CS 3100
Data Structures and Algorithms 2
Lecture 5: Dijkstra's Shortest Path Algorithm

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

Readings in CLRS 4th edition:

- Section 22.3

Announcements

- PS2 available soon, due Wednesday
- PA1 Gradescope submission coming soon
- Office hours
  - Prof Hott Office Hours: Mondays 11a-12p, Fridays 10-11a and 2-3p
  - Prof Pettit Office Hours: Mondays and Wednesdays 2:30-4:00p
  - TA office hours posted on our website

DFS: the Strategy in Words

Depth-first search strategy

- Go as deep as can visiting un-visited nodes
  - Choose any un-visited vertex when you have a choice
- When stuck at a dead-end, backtrack as little as possible
  - Back up to where you could go to another unvisited vertex
- Then continue to go on from that point
- Eventually you'll return to where you started
  - Reach all vertices? Maybe, maybe not

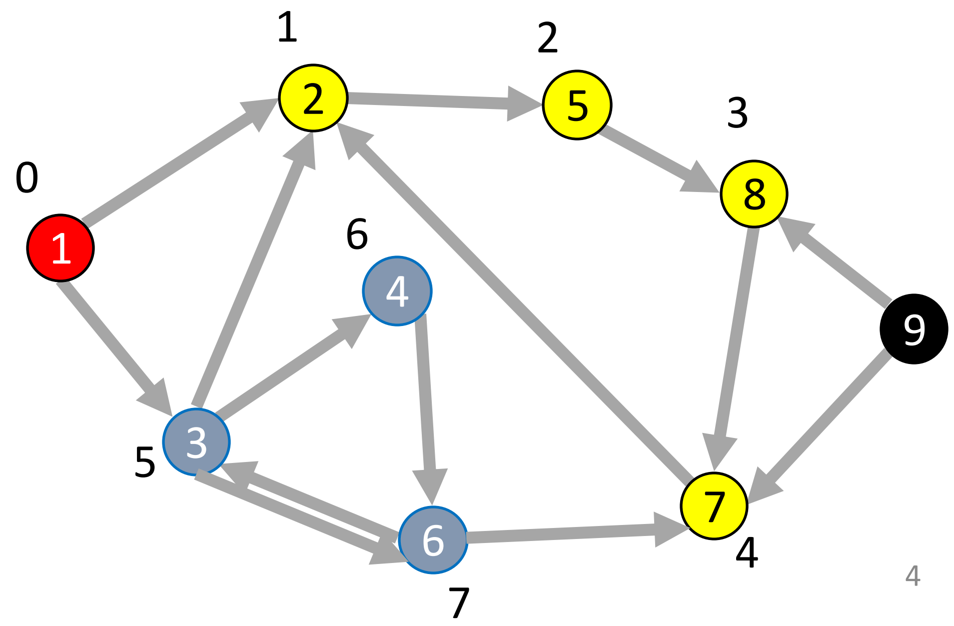
Depth-First Search

Input: a Graph G and a node s

Behavior: Start with node s, visit one neighbor of s, then all nodes reachable from that neighbor of s, then another neighbor of s,…

Output:

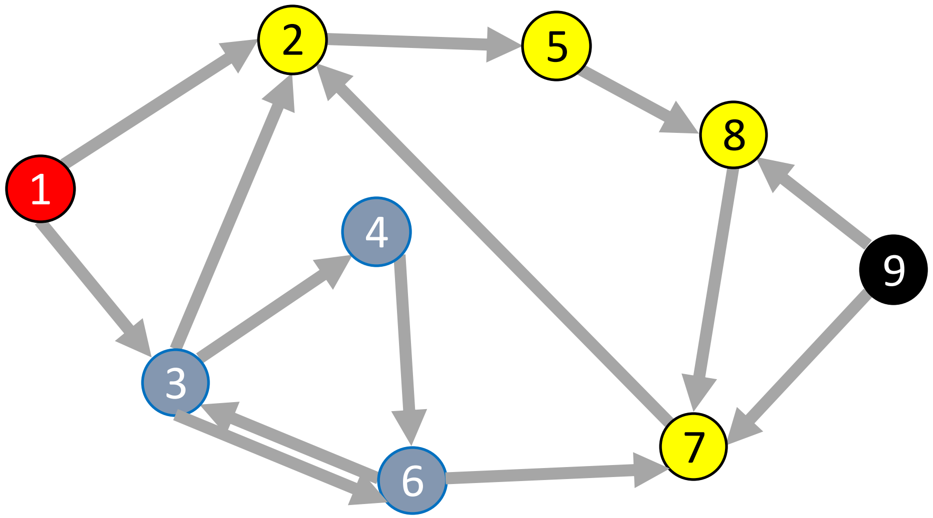
- Does the graph have a cycle?
- A topological sort of the graph.



DFS: Recursively

```
def dfs(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done)
        mark curr as seen
        for v in neighbors(current):
                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```

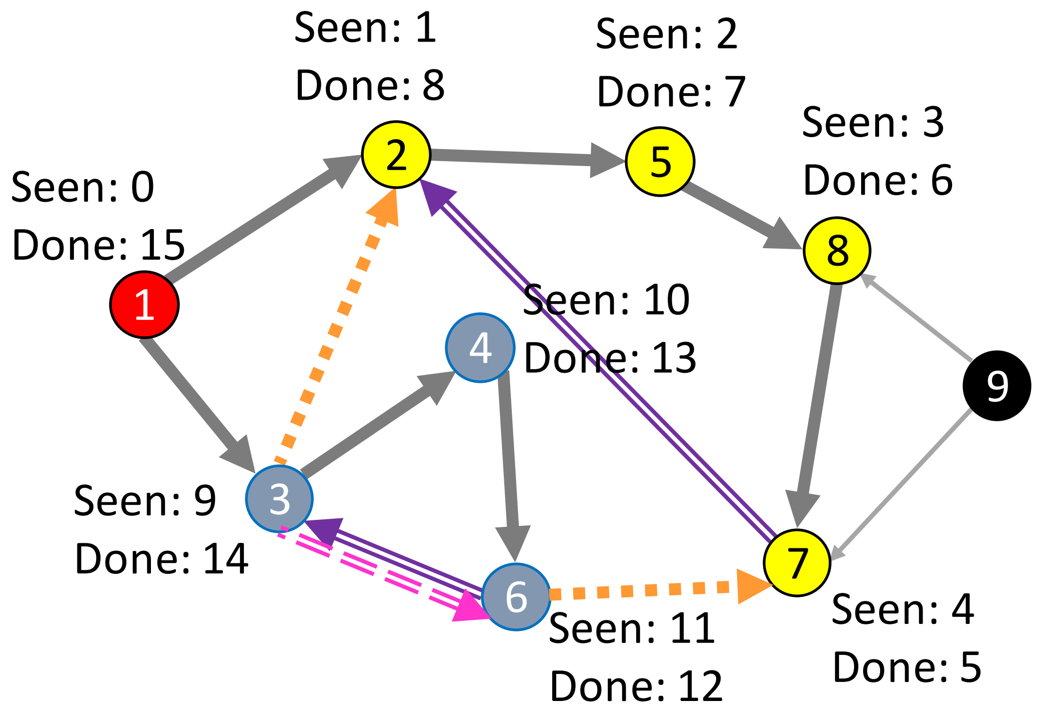


Using DFS

Consider the "seen times" and "done times"

Edges can be categorized:

- Tree Edge
    - (a, b) was followed when pushing
    - (a, b) when b was **unseen** when we were at a
- Back Edge
    - (a, b) goes to an "ancestor"
    - a and b **seen** but not **done** when we saw (a, b)
    - $t_{seen}(b) < t_{seen}(a) < t_{done}(a) < t_{done}(b)$
- Forward Edge
    - (a, b) goes to a "descendent"
    - b was **seen** and **done** between when a was **seen** and **done**
    - $t_{seen}(a) < t_{seen}(b) < t_{done}(b) < t_{done}(a)$
- Cross Edge
    - (a, b) connects "branches" of the tree
    - b was **seen** and **done** before a was ever **seen**
    - (a, b) when $t_{done}(b) > t_{seen}(a)$ and

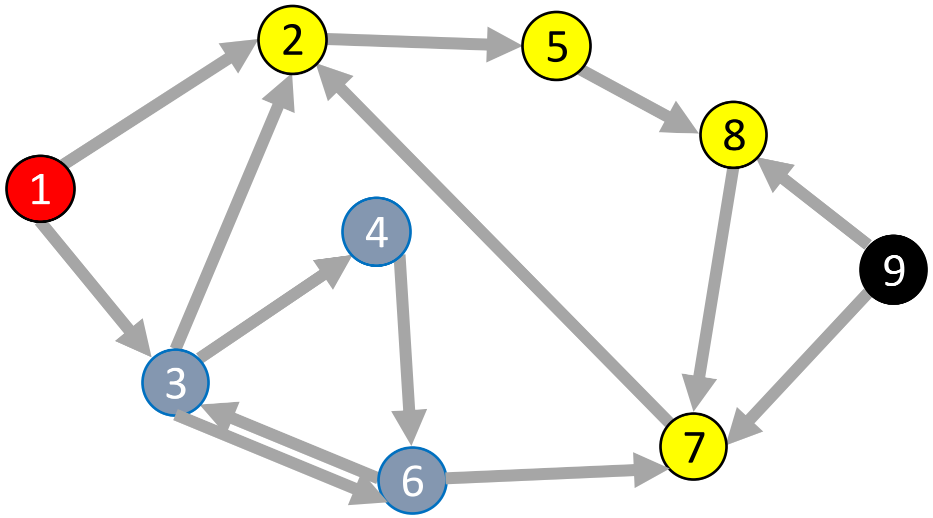


DFS: Cycle Detection

Idea: Look for a back edge!

```
def dfs(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done)
        mark curr as seen
        for v in neighbors(current):
                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```



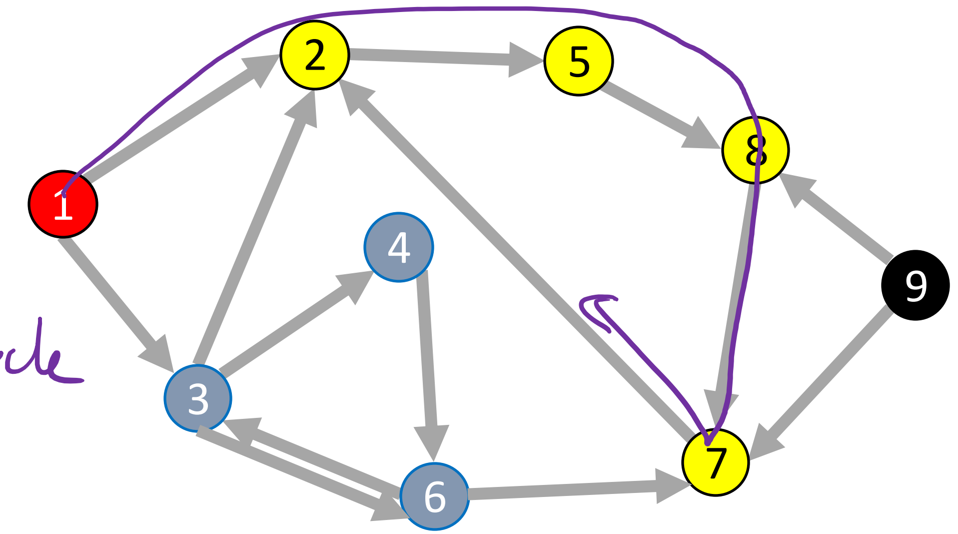
DFS: Cycle Detection

Idea: Look for a back edge!

```
def hasCycle(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
  return dfs_rec(graph, s, seen, done)

def hasCycle_rec(graph, curr, seen, done)
        cycle = False
        mark curr as seen    seen[curr] = True
        for v in neighbors(current):
back edge [   if seen[v] and not done[v]:
                  cycle = True
              if v not seen:
                  cycle = dfs_rec(graph, v, seen, done) or cycle
        mark curr as done
        return cycle
```

Seen[2] = True
Done[2] = False

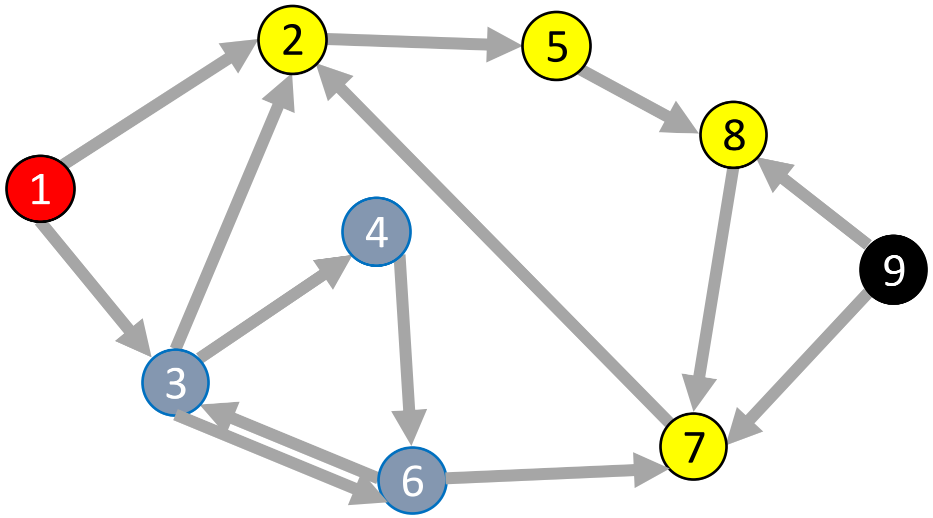


DFS: Cycle Detection

Idea: Look for a back edge!

```
def hasCycle(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        return hasCycle_rec(graph, s, seen, done)

def hasCycle _rec(graph, curr, seen, done):
        cycle = False
        mark curr as seen
        for v in neighbors(current):
                if v seen and v not done:
                        cycle = True
                elif v not seen:
                        cycle = dfs_rec(graph, v, seen, done) or cycle
        mark curr as done
        return cycle
```



Back Edges in Undirected Graphs

Finding back edges for an undirected graph is not **quite** this simple:

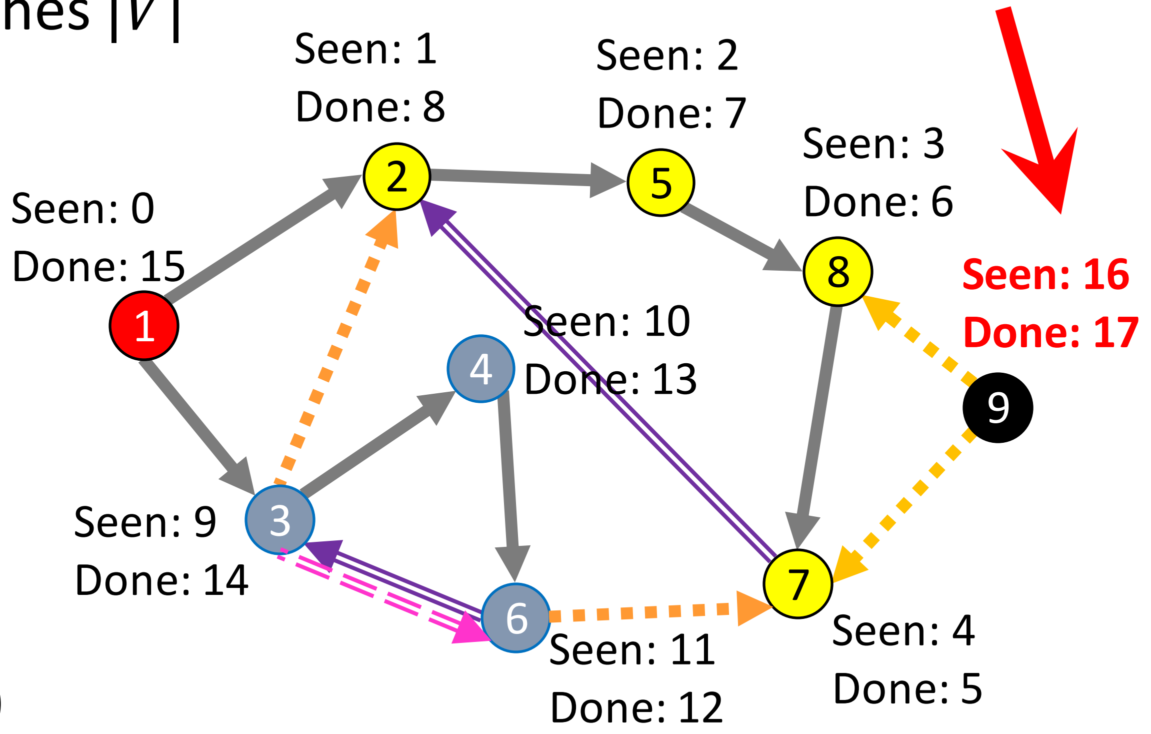
- The parent node of the current node is **seen** but not **done**
- Not a cycle, is it? It's the same edge you just traversed

Question: how would you modify our code to recognize this?

DFS "Sweep" to Process All Nodes

```
def dfs_sweep(graph):  # no start node given
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        for s in graph :  # do DFS at every vertex
            if s not seen:
                dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done) # unchanged
        mark curr as seen
        for v in neighbors(current):
                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```



CLRS's DFS Algorithm (non-recursive part)

```
DFS_sweep(G)     // CLRS calls this just dfs()
for each vertex u in G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u in G.V
    if u.color == WHITE  // if unseen
        DFS-VISIT(G, u)  // explore paths out of u
```

CLRS's DFS Algorithm (recursive part)

```
DFS-VISIT(G, u)   // sometimes called this dfs_recurse()
time = time + 1  // white vertex u has just been discovered
u.d = time  // discovery time of u
u.color = GRAY  // mark as seen
for each v in G.Adj[u]  // explore edge (u, v)
    if v.color == WHITE   // if unseen
        v.π = u
        DFS-VISIT(G, v)  // explore paths out of v (i.e., go "deeper")
u.color = BLACK  // u is finished
time = time + 1
u.f = time  // finish time of u
```

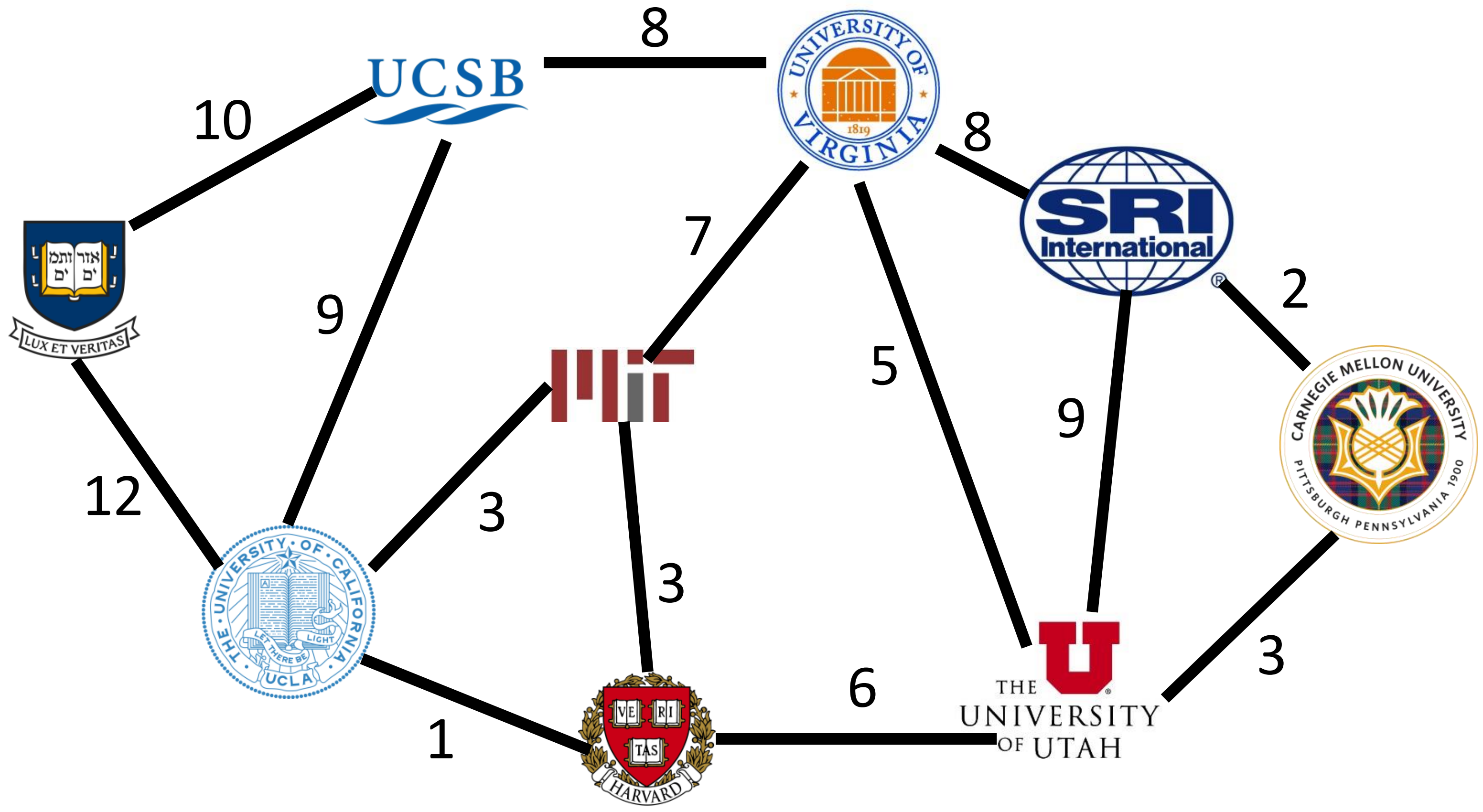
Time Complexity of DFS

For a digraph having V vertices and E edges

- Each edge is processed once in the while loop of dfs_rec() for a cost of $\Theta(E)$
  - Think about *adjacency list* data structure.
  - Traverse each list exactly once. (Never back up)
  - There are a total of **E** nodes in all the lists
- The non-recursive dfs() algorithm will do $\Theta(V)$ work even if there are no edges in the graph
- Thus over all time-complexity is $\Theta(V + E)$
  - Remember: this means the larger of the two values
  - Reminder: This is considered "linear" for graphs since there are two size parameters for graphs.
- Extra space is used for seen/done (or color) array.
  - Space complexity is $\Theta(V)$

Shortest Path

Single-Source Shortest Path Problem



Find the <u>shortest path</u> based on sum of edge-weights from UVA to each of these other places.

The problem: Given a graph $G = (V, E)$ and a start node (i.e., source) $s \in V$,

for each $v \in V$ find the minimum-weight path from $s \to v$ (call this weight $\delta(s, v)$)

Assumption (for this unit): all edge weights are positive

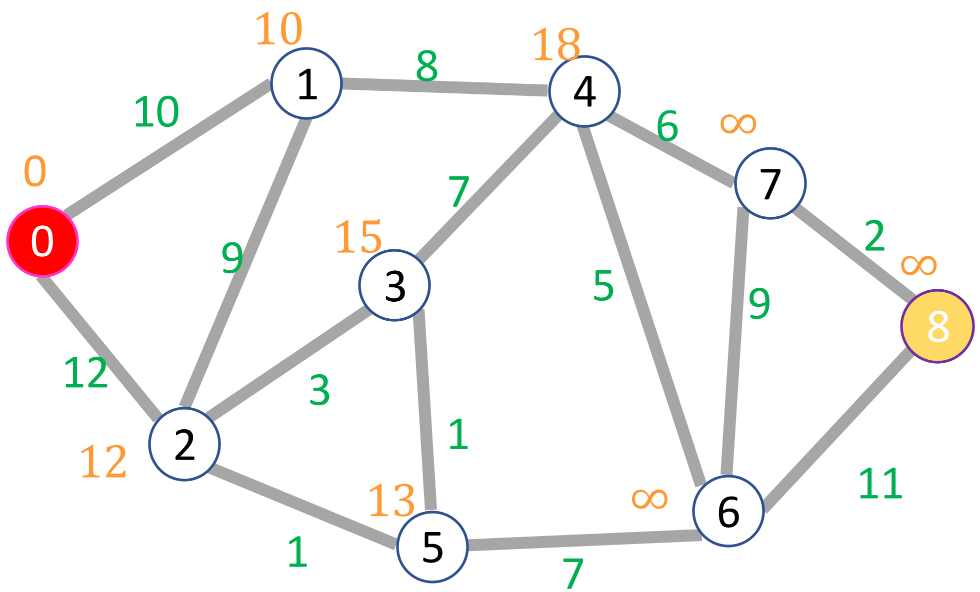
Dijkstra's Algorithm

Input: graph with **no negative edge weights**, start node s, end node t

Behavior: Start with node s, repeatedly go to the incomplete node "nearest" to s, stop when

Output:

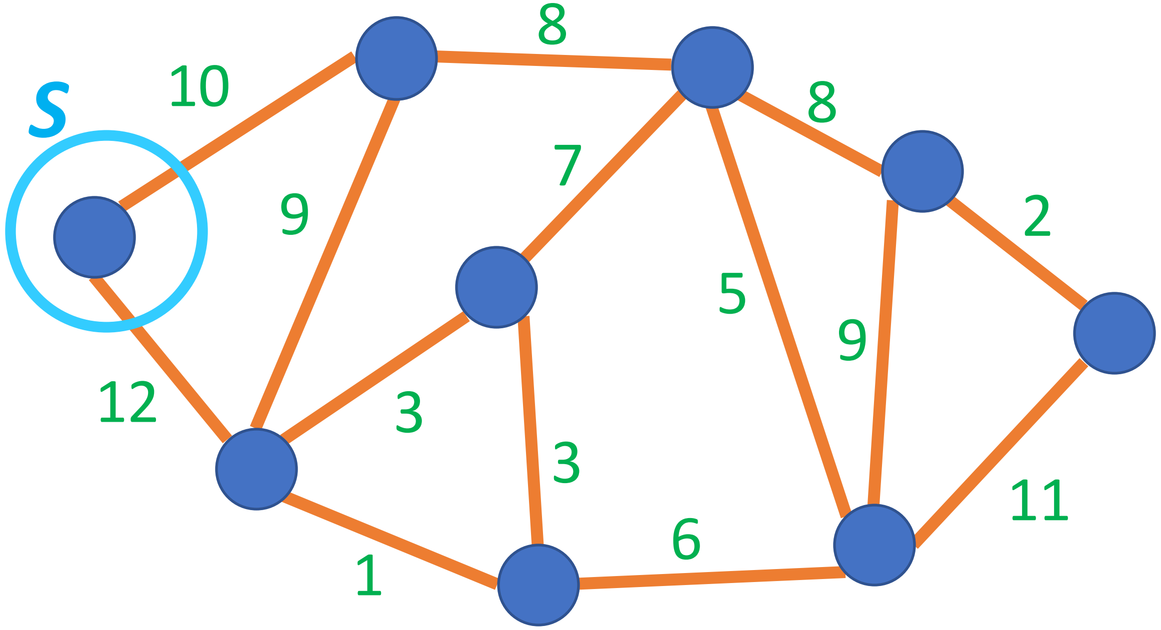
- Distance from start to end
- Distance from start to every node



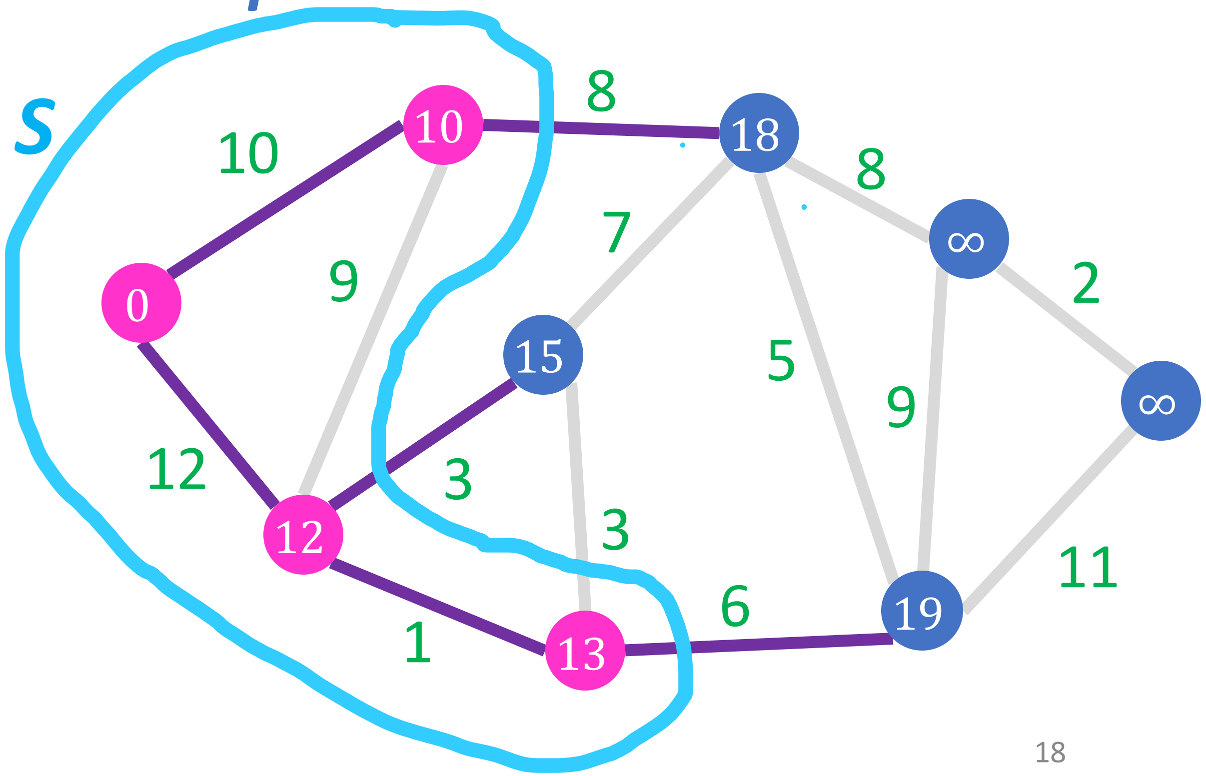
Dijkstra's Algorithm

1. Start with an empty tree S and add the source to S
2. Repeat $|V| - 1$ times:
   - At each step, add the node "nearest" to the source not yet in S to S

Initially:



At some point later:



Data Structure to Store Nodes

The strategy: At every step, choose node not in *S* that's closest to source

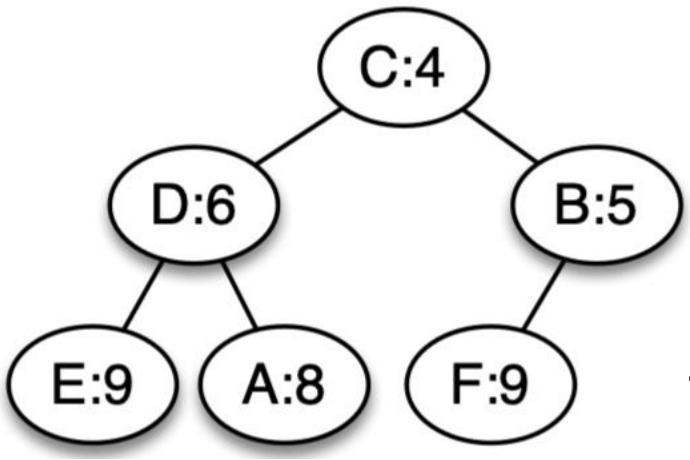
To do this efficiently, we need a data structure that:

- Stores a set of (node, distance) pairs
- Allows efficient removal of the pair with smallest distance
- Allows efficient additions and updates

This is the **Priority Queue** ADT (Abstract Data Type)!
Remember the **binary heap** data structure?

We'll need a **min-heap** (node with smallest priority at the root)

Review: Storing a Heap in an Array



Min-heap stored in array

0	1	2	3	4	5	6
:-1	C:4	D:6	B:5	E:9	A:8	F:9

Must store the key (priority) value, and maybe other info (e.g. node ID)

Store the elements in a one-dimensional array in strict left-to-right, **level order**

That is, we store all of the nodes on the tree's level *i* from left to right before storing the nodes on level *i* + 1.

- *Usually we ignore index position 0*
- *Simple formulas to find children, siblings,…*
  - ***2i:*** *left child,* ***2i+1:*** *right child*
  - ***floor(i/2):*** *parent*

Review: Heap Operations

extractMin() *perhaps called poll() in CS 2100*

- Returns and removes the item with the min key (e.g. the heap's root)
- Move last item to root and "bubble it down" to correct location
- Complexity: O(log n)

insert(item, key) *perhaps called push() in CS 2100*

- Add new item at end of array and "bubble it up" to correct location
- Complexity: O(log n)

decreaseKey(item, newKey) *not covered in CS 2100!*

- Find item in min-heap, decrease its key, and "bubble it up" to correct location
- Complexity: uh oh! Can we find item quickly, i.e. in O(log n)?
- Could sequential search the array. Then complexity is O(n)
- We **can** do this in O(log n) if we use **indirect heaps** (details later)

Dijkstra's Algorithm Implementation

1. Start with an empty tree S and add the source to S
2. Repeat $|V| - 1$ times:
   - Add the node to S that's not yet in S and that's "nearest" to source

Implementation:

distance

initialize $d_v = \infty$ for each node v

add all nodes $v \in V$ to the priority queue PQ, using d_v as the key

set $d_s = 0$

while PQ is not empty:

  $v = \mathrm{PQ.extractMin}()$

  for each $u \in V$ such that $(v, u) \in E$:

    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:

      $\mathrm{PQ.decreaseKey}(u, d_v + w(v, u))$

      $u.\mathrm{parent} = v$

cost to get to v from s

$s \longrightarrow v \xrightarrow{w} u$

each node also maintains a parent, initially NULL

key: length of shortest path $s \rightarrow u$ using nodes in PQ

Dijkstra's Algorithm Implementation

Implementation:

initialize $d_v = \infty$ for each node v

add all nodes $v \in V$ to the priority queue PQ, using d_v as the key

set $d_s = 0$

while PQ is not empty:

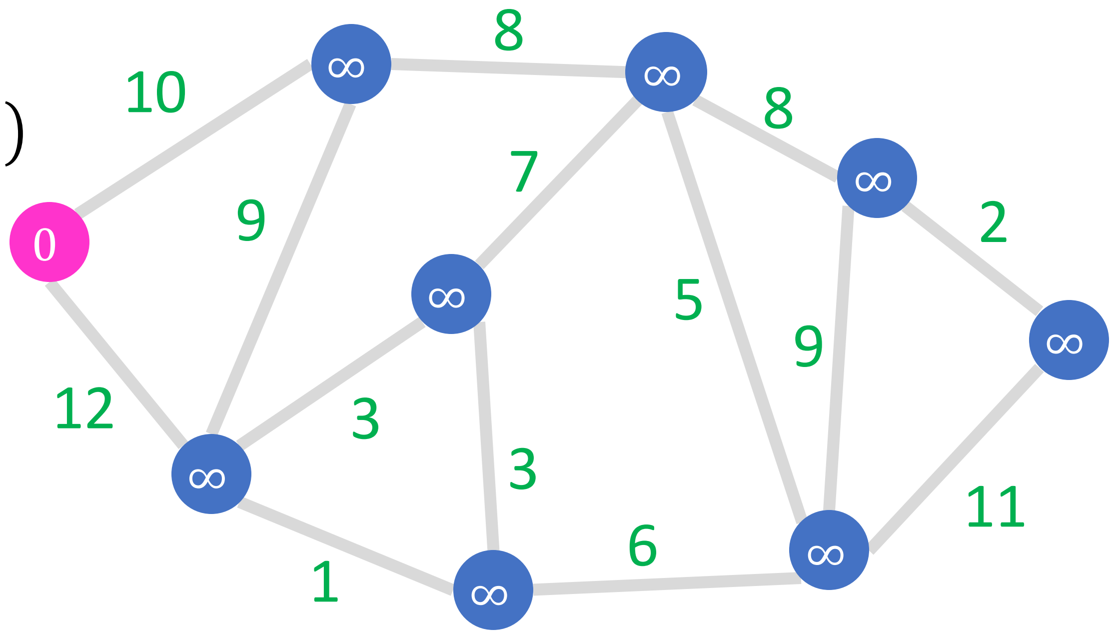
$v = \mathrm{PQ}.\mathrm{extractMin}()$

for each $u \in V$ such that $(v, u) \in E$:

if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:

$\mathrm{PQ}.\mathrm{decreaseKey}\big(u, d_v + w(v, u)\big)$

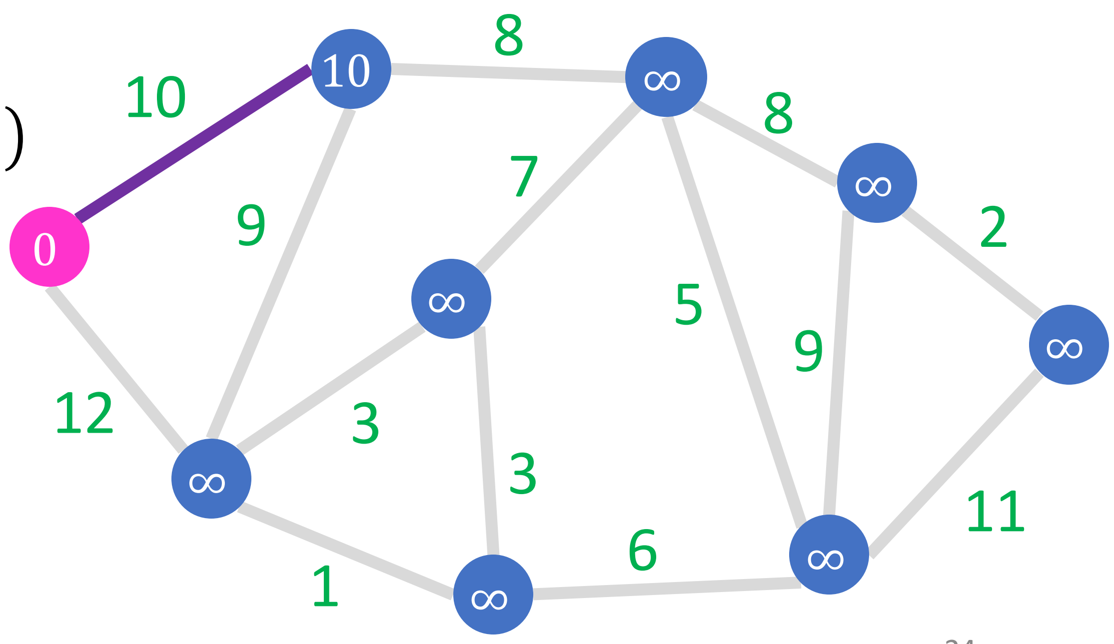
$u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

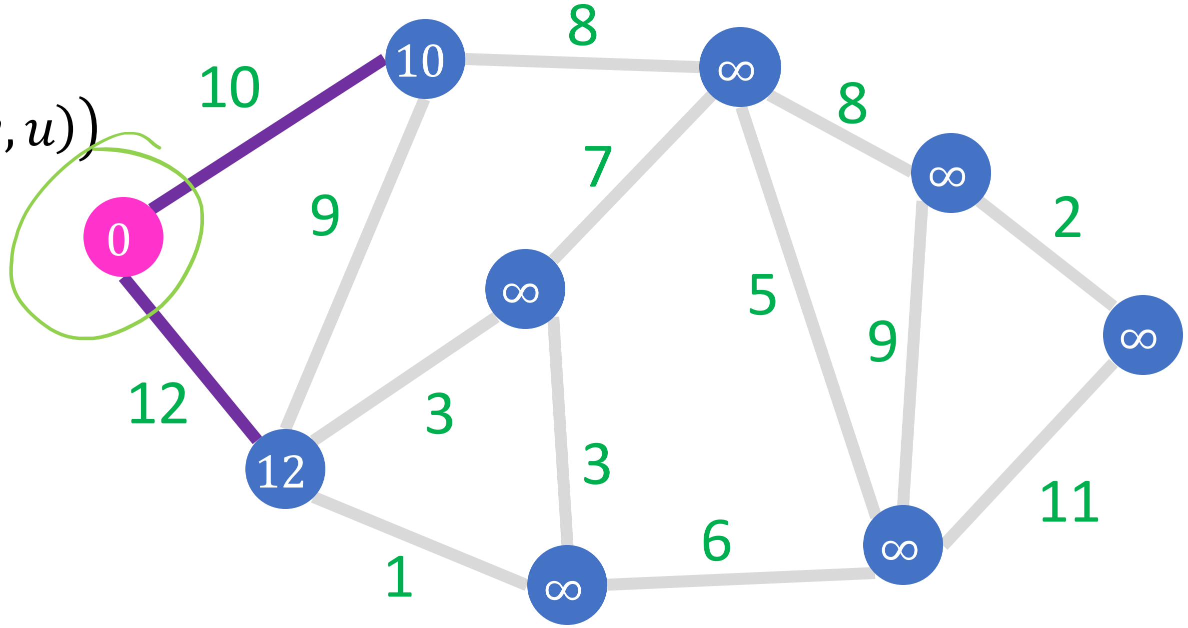
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v,u) < d_u$:
            $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v,u)\big)$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

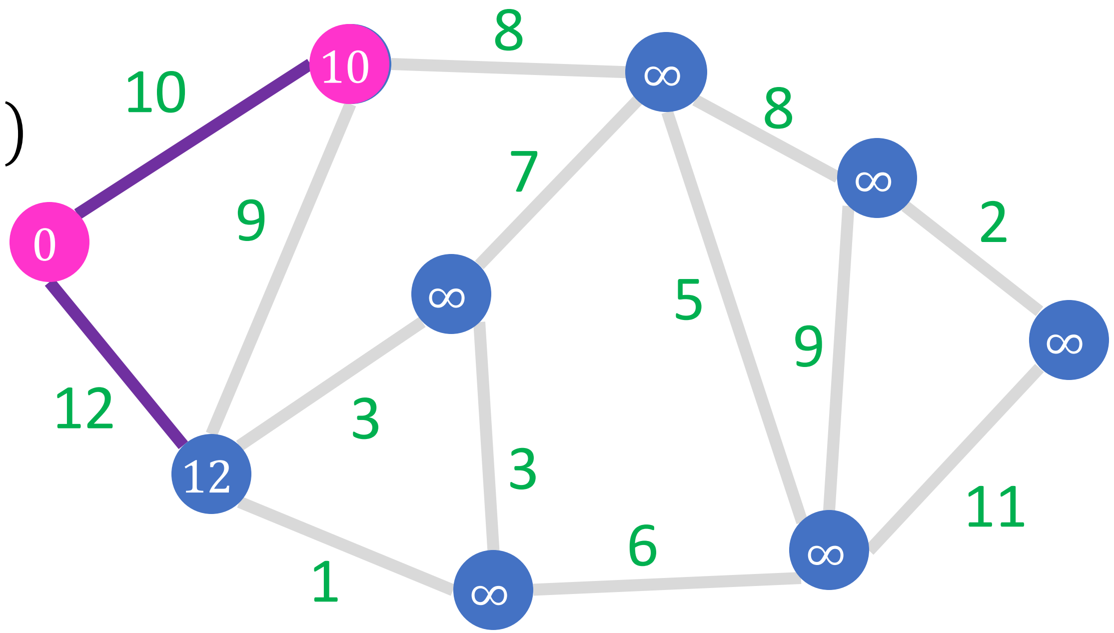
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, d_v + w(v, u))$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

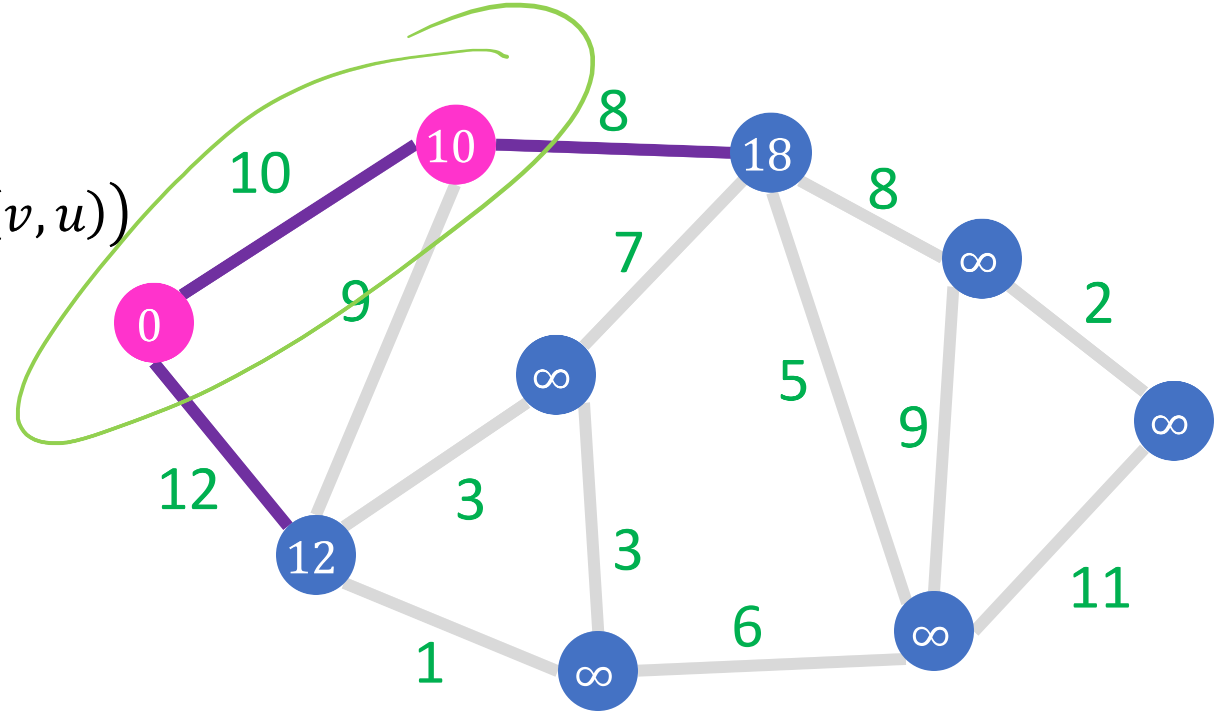
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin()}$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v,u) < d_u$:
            $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v,u)\big)$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

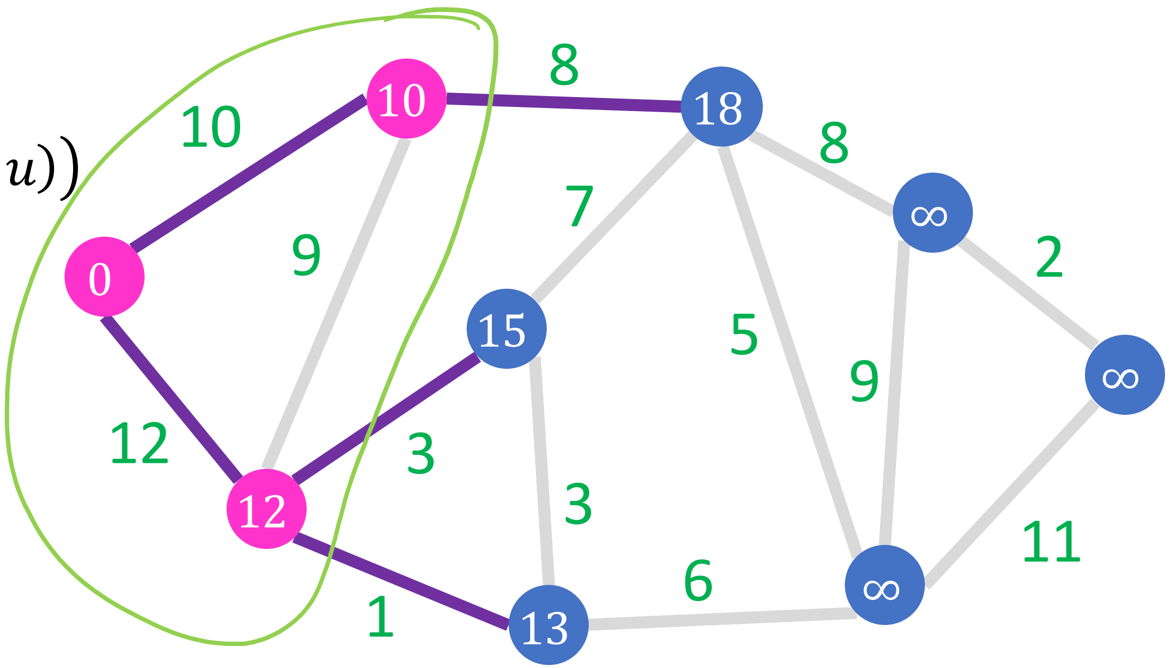
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

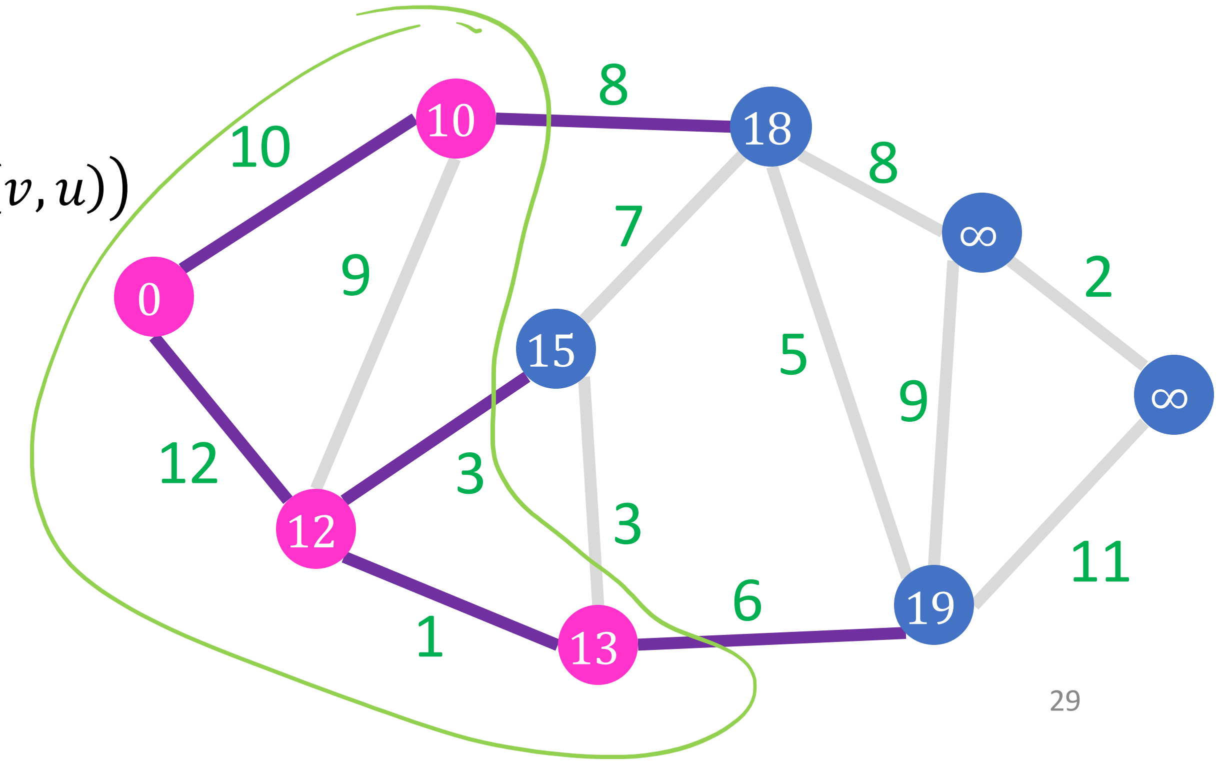
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ}.\mathrm{extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ}.\mathrm{decreaseKey}(u, d_v + w(v, u))$
      $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

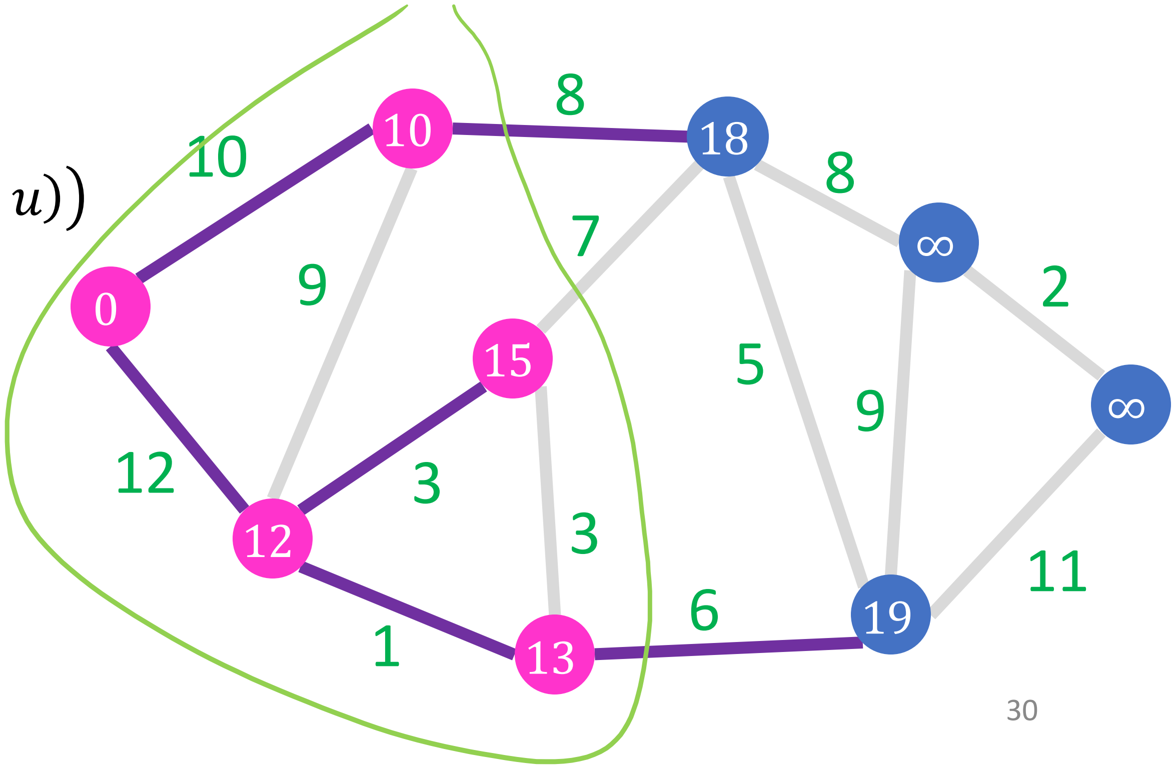
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v,u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}(u, d_v + w(v,u))$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

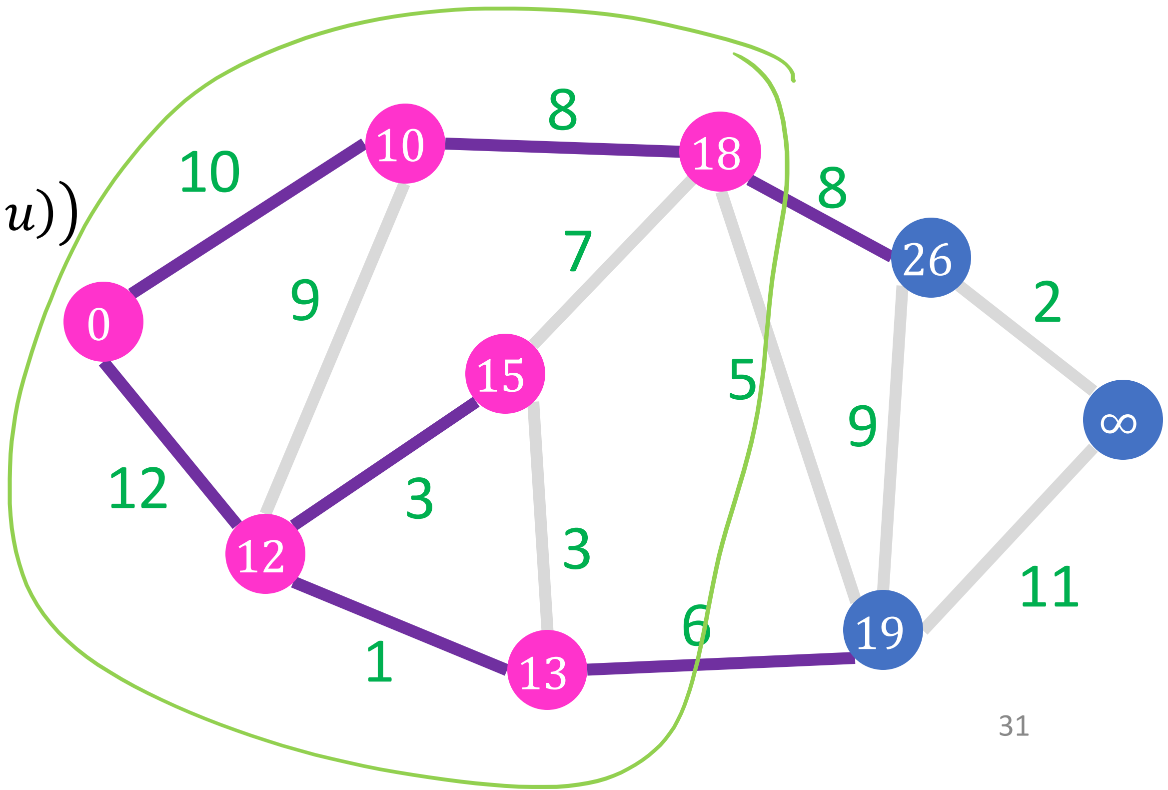
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, d_v + w(v, u)\big)$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

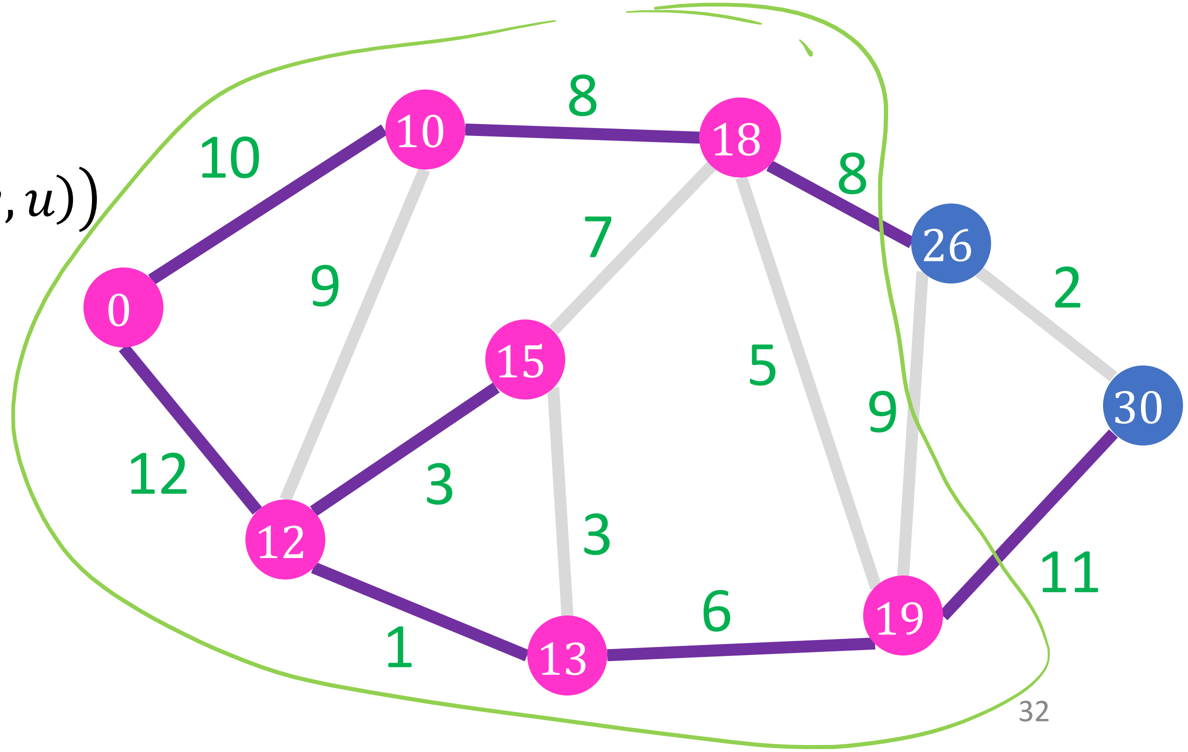
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, d_v + w(v, u))$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

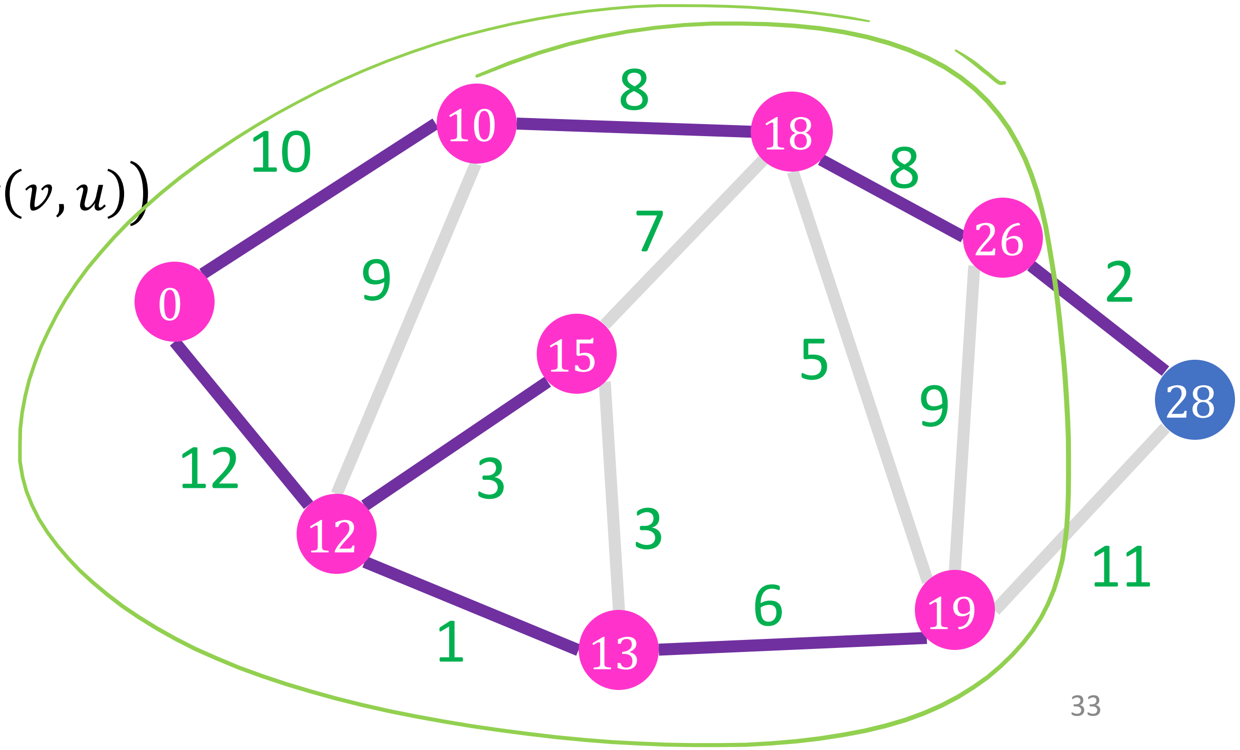
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ.extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$
      $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

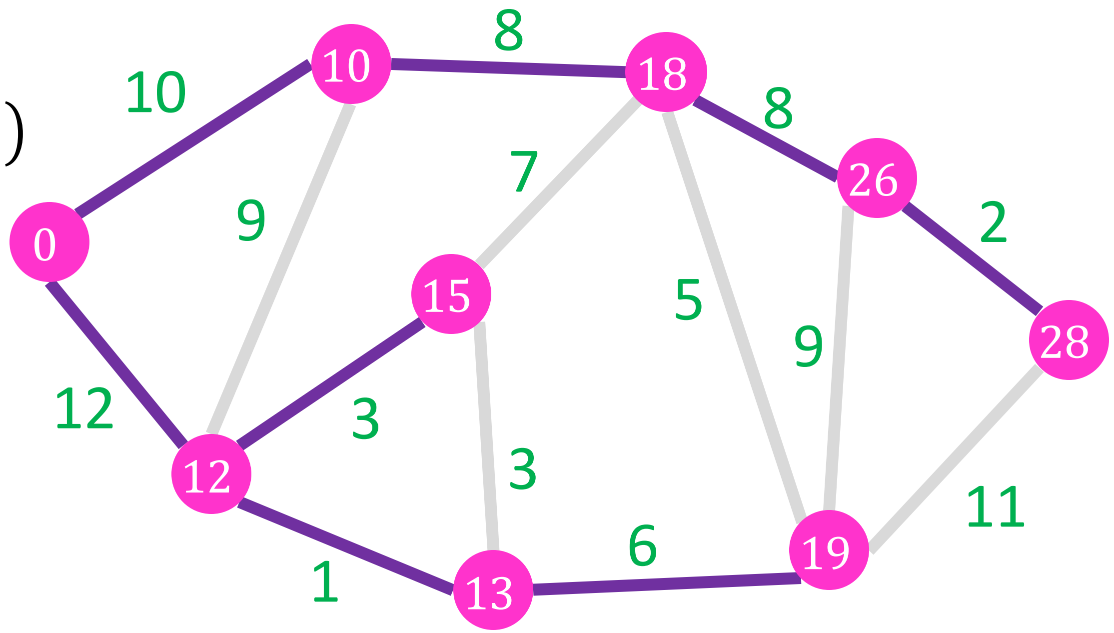
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ.extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ.decreaseKey}(u, d_v + w(v, u))$
      $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ.extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ.decreaseKey}\left(u, d_v + w(v, u)\right)$
      $u.\mathrm{parent} = v$



Observe: shortest paths from a source forms a tree, shortest path to every reachable node

Every subpath of a shortest path is itself a shortest path. (This is called the *optimal substructure property*.)

Dijkstra's Algorithm Running Time

Implementation:

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, d_v + w(v, u)\big)$
            $u.\mathrm{parent} = v$

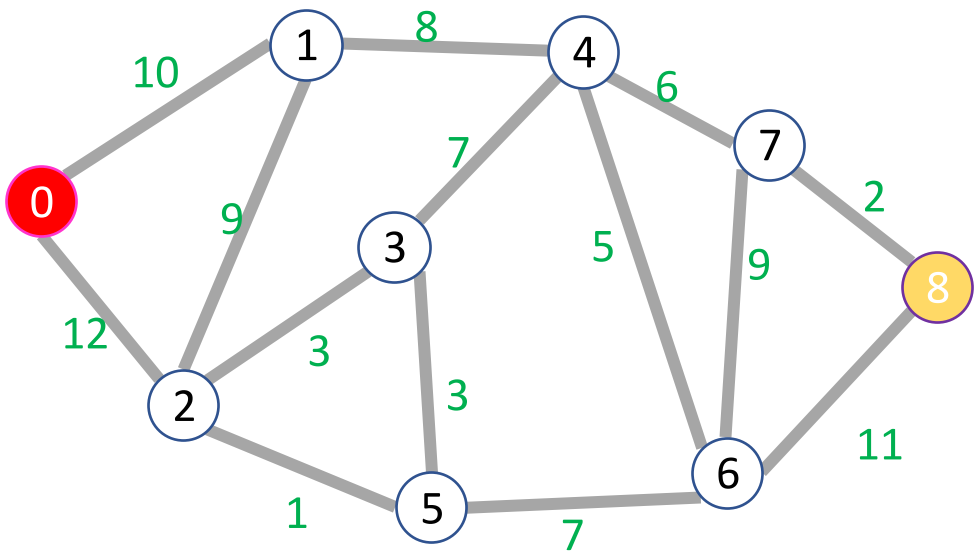
Initialization:
$O(|V|)$
+
$|V|$ iterations
$O(\log|V|)$
$|E|$ iterations total
?? $O(\log|V|)$ if we use indirect heaps

$\boxed{\begin{array}{l}|V| = n \\ |E| = m\end{array}}$

Overall running time: $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$
or, $O(m \log n)$

Python-like Code for Dijkstra's Algorithm

```
def Dijkstras(graph, start, end):
    distances = [∞, ∞, ∞,…]  # one index per node
    done = [False,False,False,…]  # one index per node
    PQ = priority queue  # e.g. a min heap
    PQ.insert((0, start))
    distances[start] = 0
    while PQ is not empty:
        current = PQ.extractmin()
        if done[current]: continue
        done[current] = True
        for each neighbor of current:
            if not done[neighbor]:
                new_dist = distances[current]+weight(current,neighbor)
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    PQ.insert((new_dist,neighbor))
    return distances[end]
```



Dijkstra's Algorithm

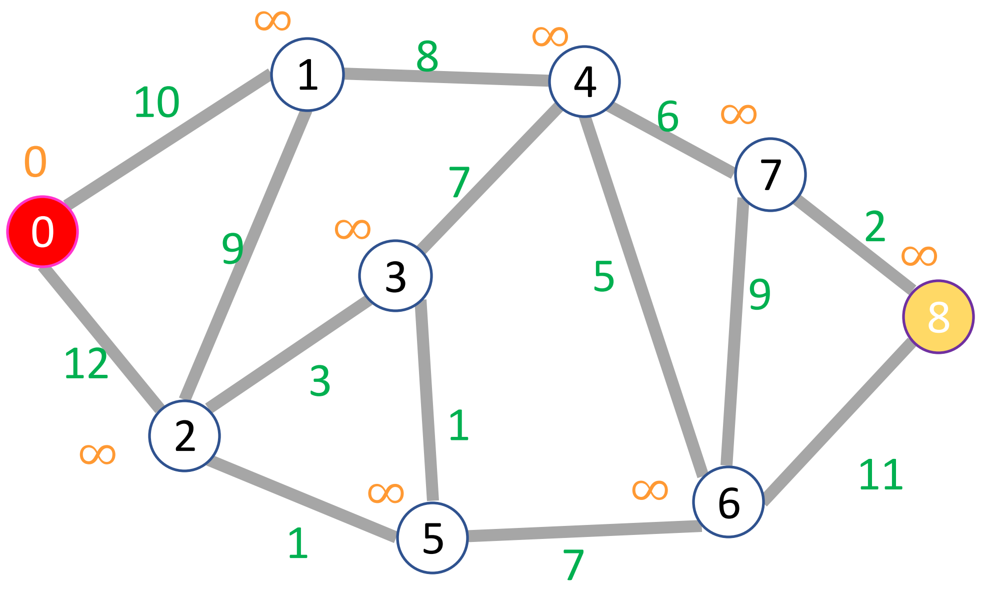
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

Node	Done?
0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞



Dijkstra's Algorithm

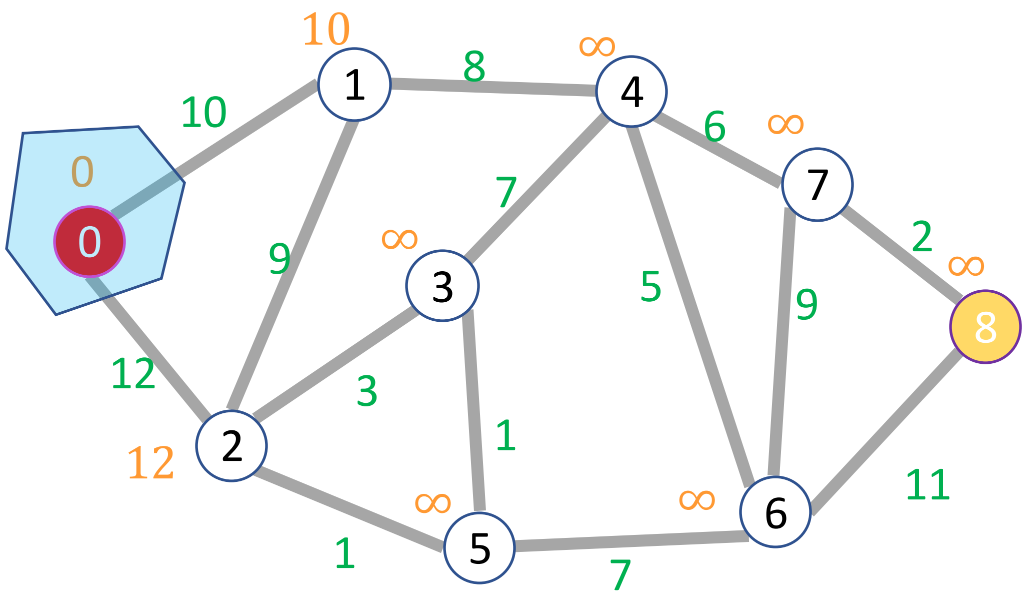
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

Node	Done?
0	T
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞



Dijkstra’s Algorithm

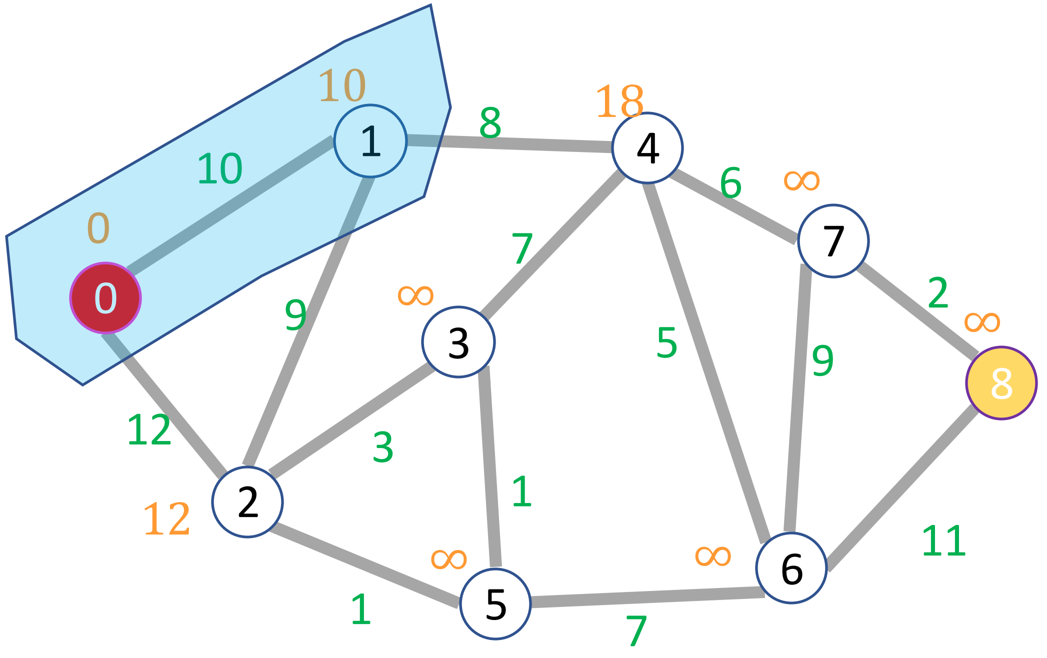
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

Node	Done?
0	T
1	T
2	F
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	∞
4	18
5	∞
6	∞
7	∞
8	∞



Dijkstra's Algorithm

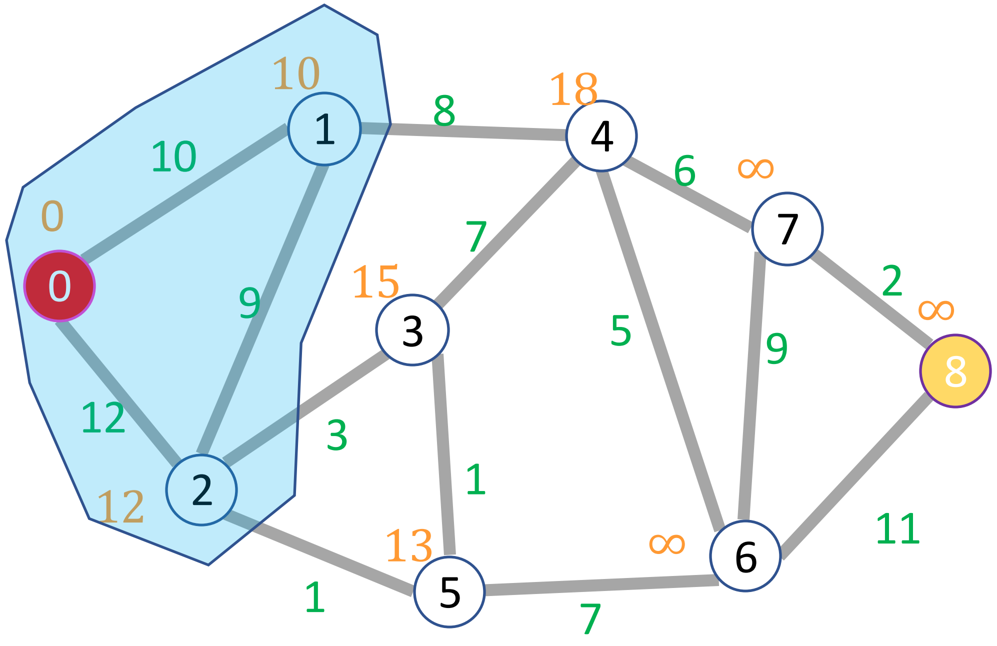
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

Node	Done?
0	T
1	T
2	T
3	F
4	F
5	F
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	15
4	18
5	13
6	∞
7	∞
8	∞



Dijkstra's Algorithm

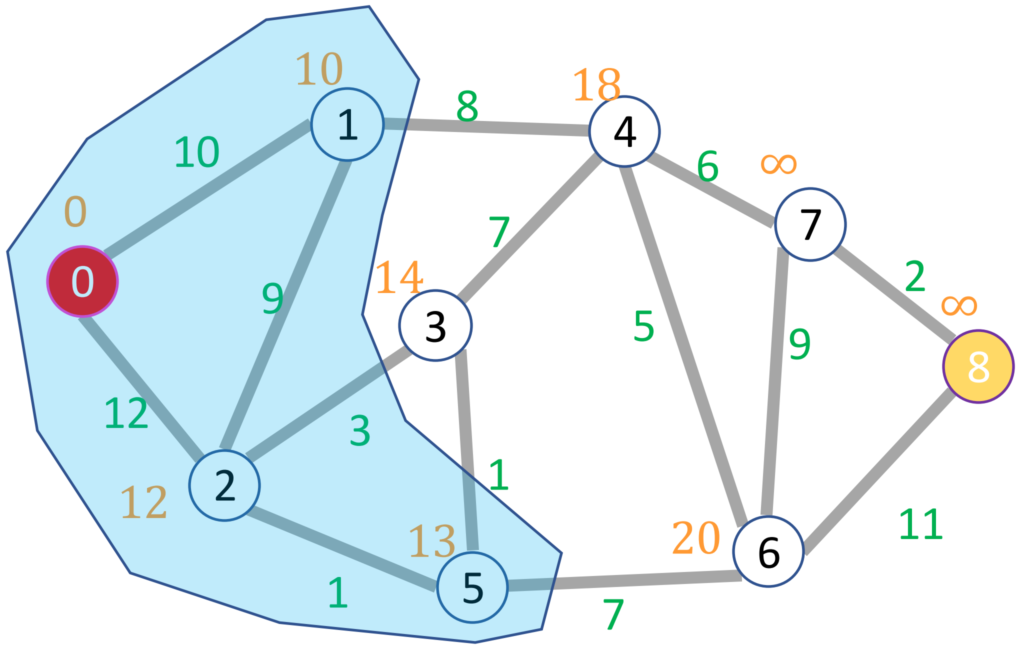
Start: 0

End: 8

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path

Node	Done?
0	T
1	T
2	T
3	F
4	F
5	T
6	F
7	F
8	F

Node	Distance
0	0
1	10
2	12
3	14
4	18
5	13
6	∞
7	20
8	∞



Dijkstra's Algorithm Implementation

Implementation:

initialize $d_v = \infty$ for each node v

add all nodes $v \in V$ to the priority queue PQ, using d_v as the key

set $d_s = 0$

while PQ is not empty:

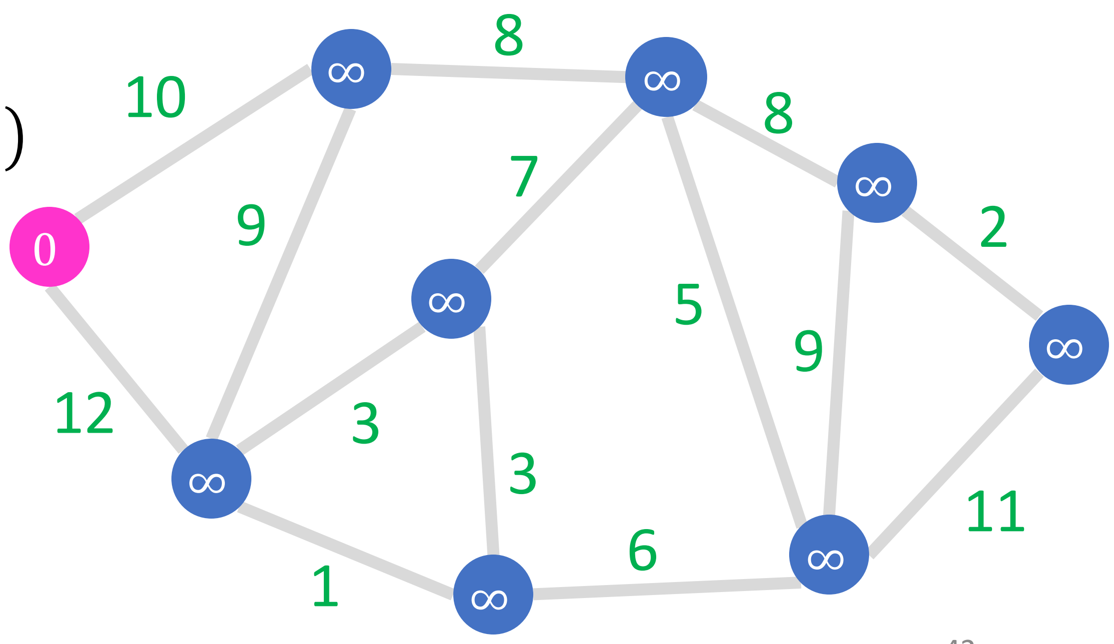
$v = \mathrm{PQ}.\mathrm{extractMin}()$

for each $u \in V$ such that $(v, u) \in E$:

if $u \in \mathrm{PQ}$ and $d_v + w(v,u) < d_u$:

$\mathrm{PQ}.\mathrm{decreaseKey}\big(u, d_v + w(v,u)\big)$

$u.\mathrm{parent} = v$



Dijkstra’s Algorithm Proof Strategy

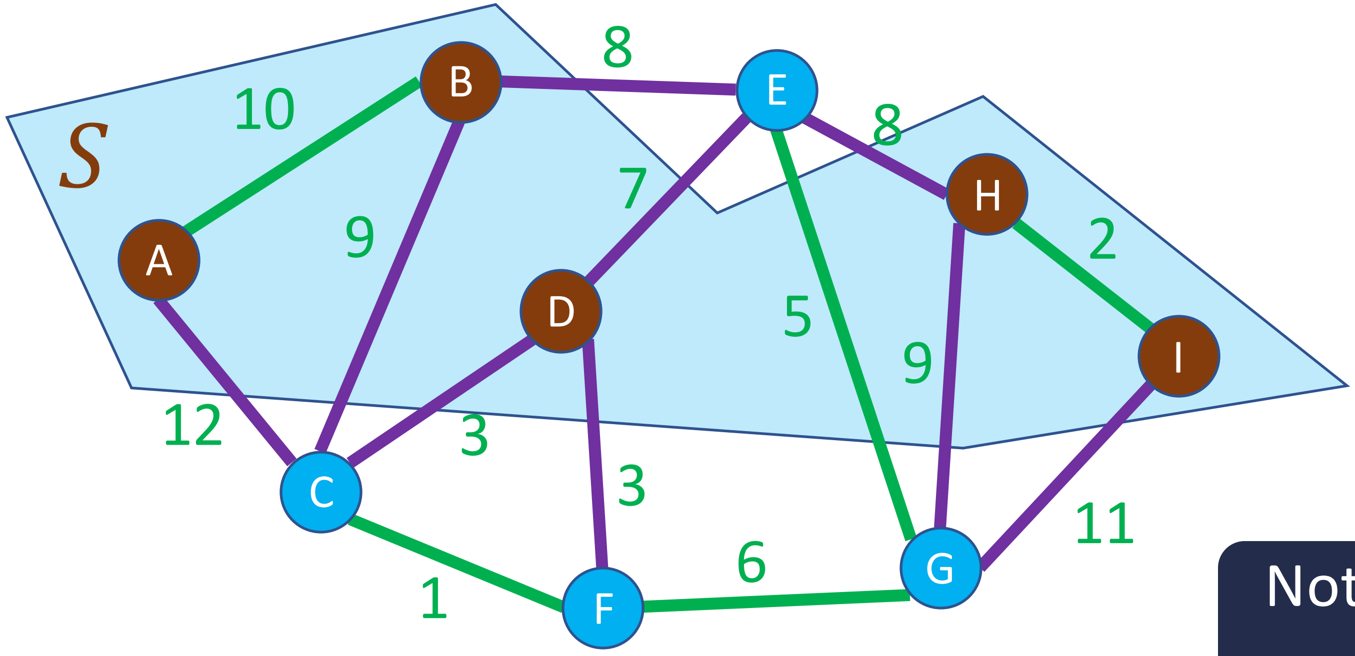
Proof by induction

Proof Idea: we will show that when node u is removed from the priority queue, $d_u = \delta(s, u)$ where $\delta(s, u)$ is the shortest distance

- **Claim 1:** There is a path of length d_u (as long as $d_u < \infty$) from s to u in G
- **Claim 2:** For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$

Graph Cuts

A **cut** of a graph $G = (V, E)$ is a partition of the nodes into two sets, S and $V - S$



Notion extends naturally to a set of edges

An edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$

An edge $(v_1, v_2) \in E$ respects a cut if $v_1, v_2 \in S$ or if $v_1, v_2 \in V - S$

Correctness of Dijkstra's Algorithm

Inductive hypothesis: Suppose that nodes $v_1 = s, \ldots, v_i$ have been removed from PQ, and for each of them $d_{v_i} = \delta(s, v_i)$, and there is a path from s to v_i with distance d_{v_i} (whenever $d_{v_i} < \infty$)

Base case:

- $i = 0$: $v_1 = s$
- Claim holds trivially

Correctness of Dijkstra's Algorithm: Claim 1

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 1: There is a path of length d_u (as long as $d_u < \infty$) from s to u in G

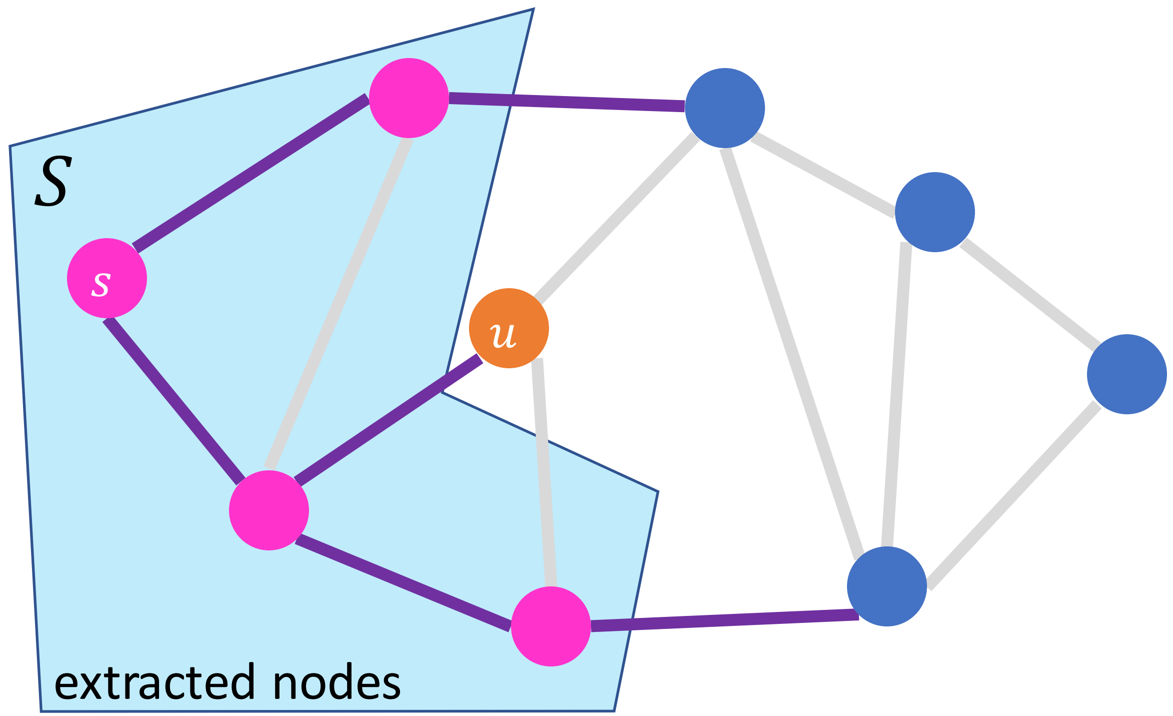
Proof:

- Suppose $d_u < \infty$
- This means that PQ.decreaseKey was invoked on node u on an earlier iteration
- Consider the last time PQ.decreaseKey is invoked on node u
- PQ.decreaseKey is only invoked when there exists an edge $(v, u) \in E$ and node v was extracted from PQ in a previous iteration
- In this case, $d_u = d_v + w(v, u)$
- By the inductive hypothesis, there is a path $s \to v$ of length d_v in G and since there is an edge $(v, u) \in E$, there is a path $s \to u$ of length d_u in G

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \ldots, u)$, $w(s, \ldots, u) \geq d_u$

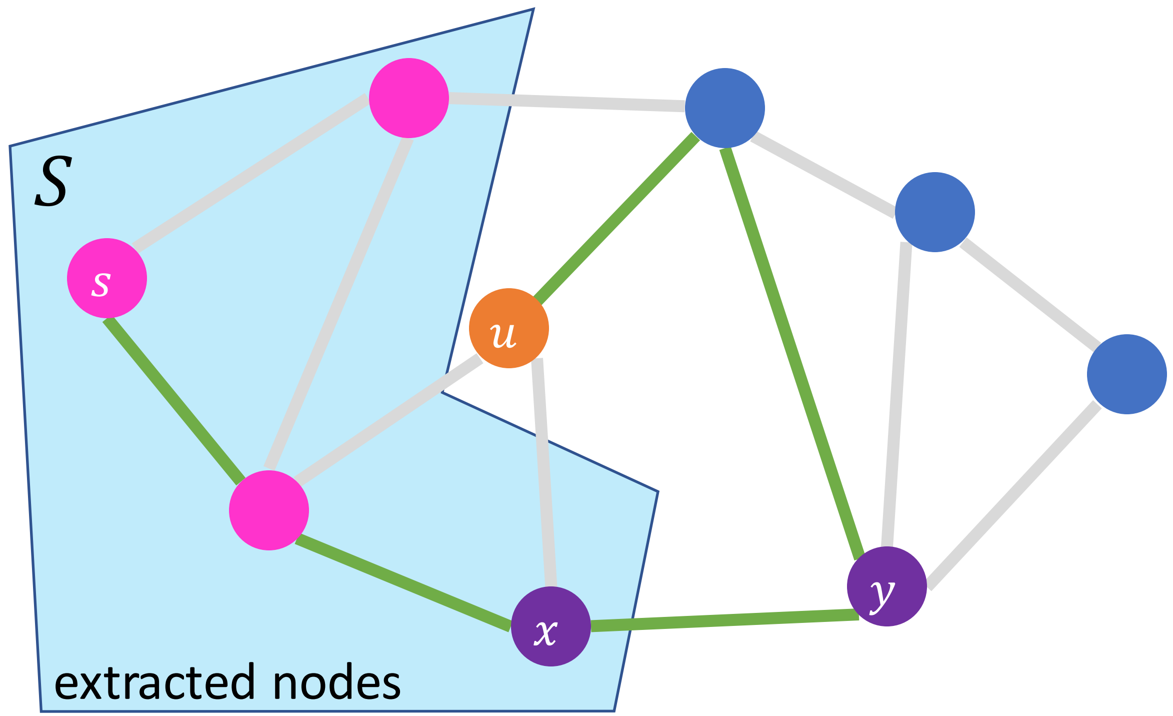


Extracted nodes "cuts" G into two subsets, $(S, V - S)$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \ldots, u)$, $w(s, \ldots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \ldots, u)$

Since $u \notin S$, $(s, \ldots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$w(s, \ldots, u) \;\geq\; \delta(s, x) + w(x, y) + w(y, \ldots, u)$$

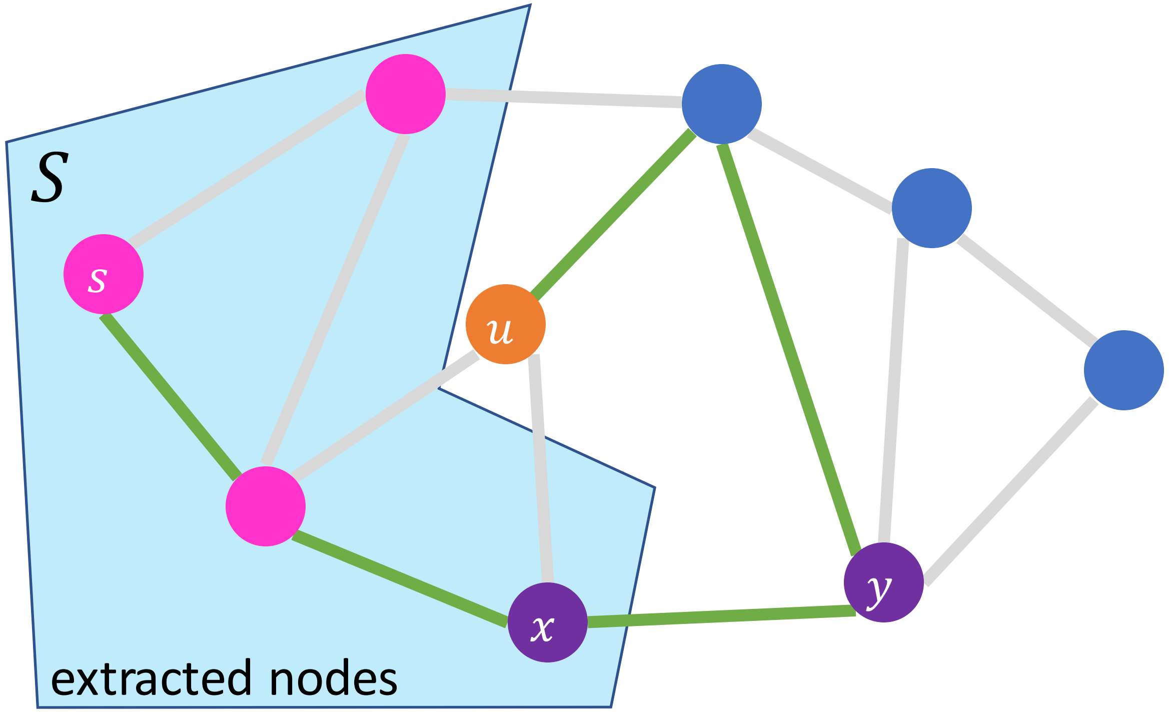
$w(s, \ldots, u) = w(s, \ldots, x) + w(x, y) + w(y, \ldots, u)$

$w(s, \ldots, x) \geq \delta(s, x)$ since $\delta(s, x)$ is weight of shortest path from s to x

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \ldots, u)$, $w(s, \ldots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \ldots, u)$

Since $u \notin S$, $(s, \ldots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

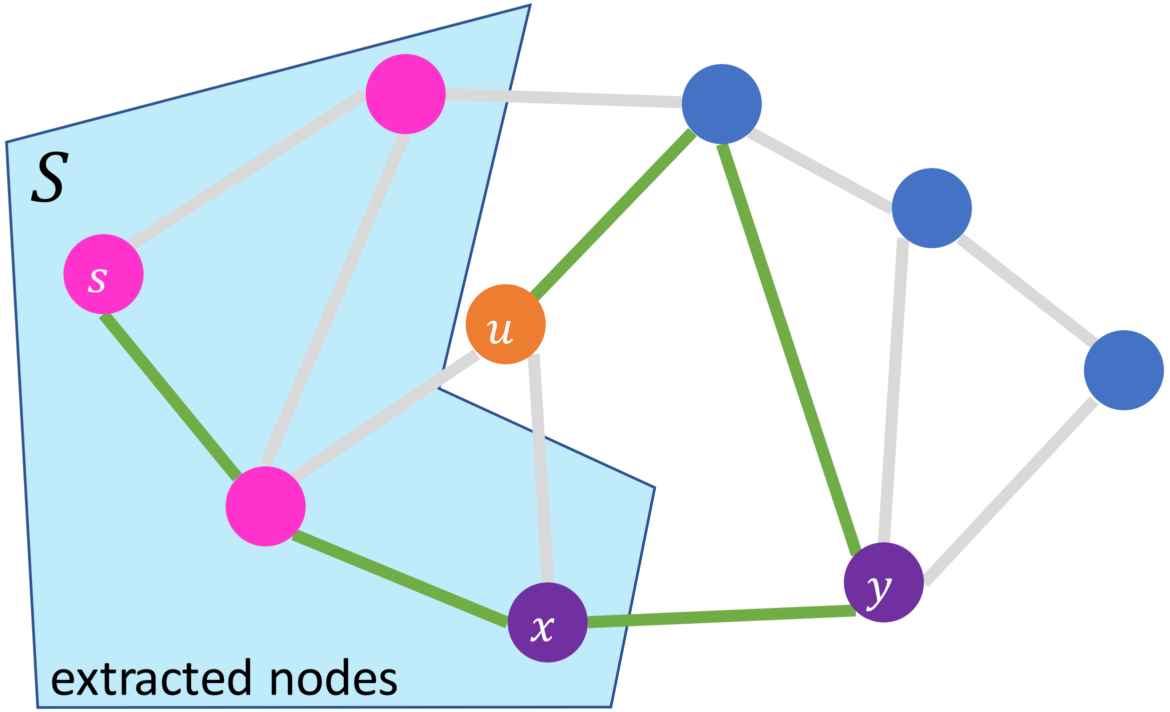
$$\begin{aligned} w(s, \ldots, u) &\geq \delta(s, x) + w(x, y) + w(y, \ldots, u) \\ &= d_x + w(x, y) + w(y, \ldots, u) \end{aligned}$$

Inductive hypothesis: since x was extracted before, $d_x = \delta(s, x)$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \ldots, u)$, $w(s, \ldots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \ldots, u)$

Since $u \notin S$, $(s, \ldots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$
\begin{aligned}
w(s, \ldots, u) &\geq \delta(s, x) + w(x, y) + w(y, \ldots, u) \\
&= d_x + w(x, y) + w(y, \ldots, u) \\
&\geq d_y + w(y, \ldots, u)
\end{aligned}
$$

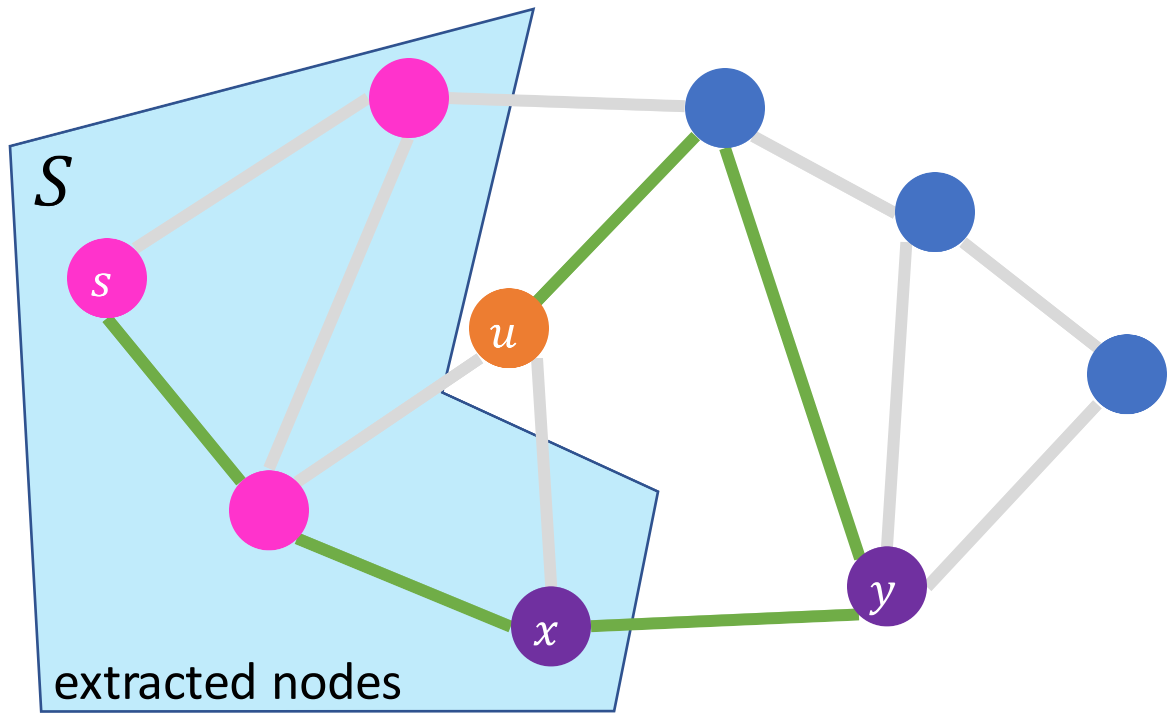
By construction of Dijkstra's algorithm, when x is extracted, d_y is updated to satisfy

$$d_y \leq d_x + w(x, y)$$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

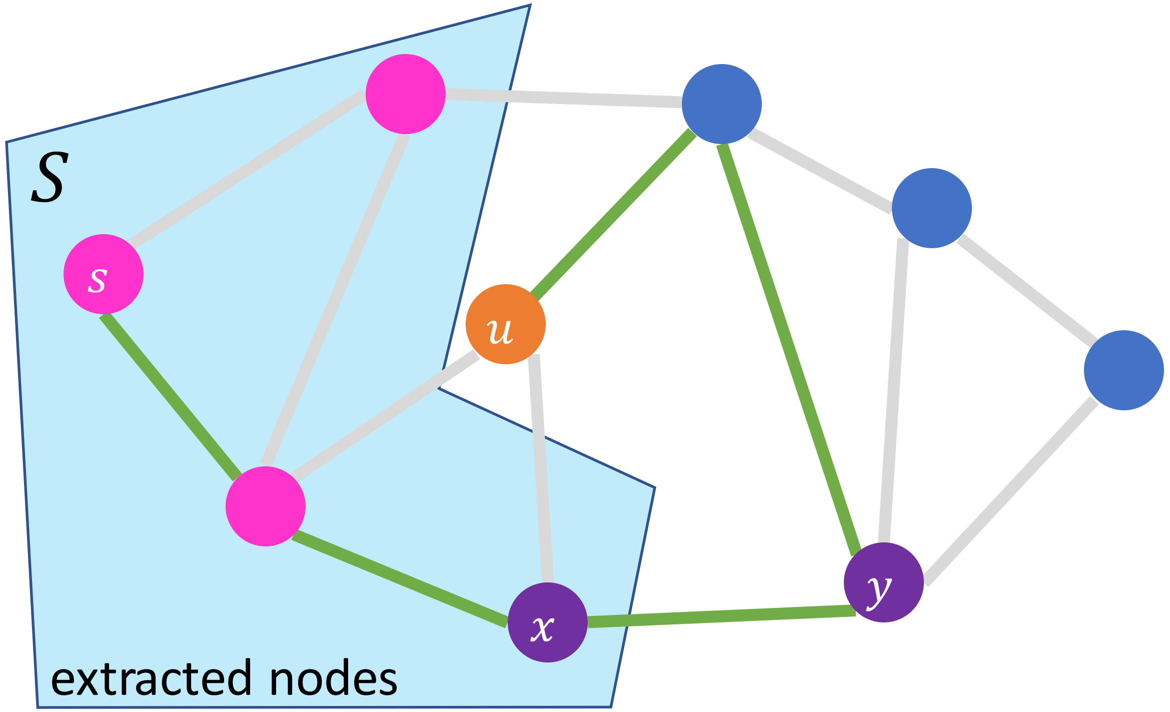
$$\begin{aligned} w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\ &= d_x + w(x, y) + w(y, \dots, u) \\ &\geq d_y + w(y, \dots, u) \\ &\geq d_u + w(y, \dots, u) \end{aligned}$$

Greedy choice property: we always extract the node of minimal distance so $d_u \leq d_y$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$\begin{aligned} w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\ &= d_x + w(x, y) + w(y, \dots, u) \\ &\geq d_y + w(y, \dots, u) \\ &\geq d_u + w(y, \dots, u) \\ &\geq d_u \end{aligned}$$

All edge weights assumed to be positive

Correctness of Dijkstra's Algorithm

Conclusion: We used proof by induction to show:

When node u is removed from the priority queue, $d_u = \delta(s, u)$

- **Claim 1:** There is a path of length d_u (as long as $d_u < \infty$) from s to u in G
- **Claim 2:** For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$

In other words, all paths $(s, \dots, u)$ are no shorter than d_u
which makes it the shortest path (or one of equally shortest paths).

Indirect Heaps

The Concern: Make decreaseKey O(log n)

Indirect heaps are an example of the common computing principle of ***indirection***:

- Simple example: an implementation of *FindMax(anArray)* that returns the array index of the max value instead of the value itself
- Pointers in languages like C and C++
- Object references in Java and Python
- A short read: https://en.wikipedia.org/wiki/Indirection

Indirect heaps:

- The idea: have some kind of "index" that, given a node's "ID", you can quickly find where that node is in the heap's tree
- Several ways to implement these
- What's shown in the next slides works well if you identify nodes with strings and you can easily use a good hashtable (dictionary)

Indirect Heap Uses >1 Data Structure

item_at_posn[i] – an array that tells us what item is stored at the position **i** in the tree

0	1	2	3	4	5	6
:-1	C:4	D:6	B:5	E:9	A:8	F:9

posn_of_item[item] – a hashtable that gives the position in the tree where a given **item** ID is stored

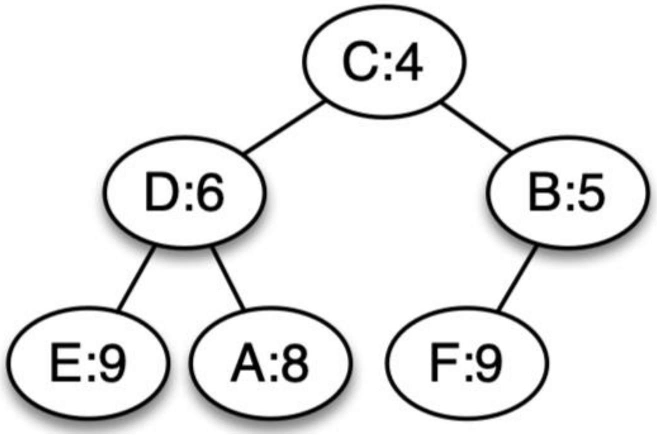
A	B	C	D	E	F
5	3	1	2	4	6

Example usage:

- What's the item at the root? **item_at_posn[1] → 'C'**
- Where in the tree is E? **posn_of_item['E'] → 4**
- What item is E's parent?

  item_at_posn[posn_of_item['E']/2] = item_at_posn[2] → 'D'

There will be some way of getting the PQ key value from the item, which we'll show as **item.key.** E.g. the min key is **item_at_posn[1].key → 4**



Is decreaseKey more efficient now?

This code shows the idea: decrease B's key and bubble it up one level:

```
item = ‘B’
item.key = 3 # it was 5
itemPosn = posn_of_item[item]    # 3
parentPosn = itemPosn / 2     # 1
parent = item_at_posn[parentPosn] # 'C'

if item.key < parent.key:  # need to swap?
   item_at_posn[parentPosn] = item     # item_at_posn[1] = 'B'
   item_at_posn[itemPosn] = parent     # item_at_posn[3] = 'C'
   posn_of_item[parent] = itemPosn     # posn_of_item['C'] = 3
   posn_of_item[item] = parentPosn     # posn_of_item['B'] = 1
```

Assuming hashtable lookup is O(1), everything here is O(1).
decreaseKey() might have to do this for the height of the tree, so O(log n) overall.