Suppose we have a graph, and do DFS search from $S$:
We should get the following tree:
We should get the following tree:

```
  S  
 /   \\  /   \  \
A     B  
   \  /    \\
   C D     
    \   /   \\
     E  
```

But if we were interested in getting to $C$, this gives a rather inefficient route of $S \rightarrow A \rightarrow B \rightarrow C$ instead of $S \rightarrow C$. 
Imagine the graph as a physical set of marbles connected by string:
When we pick up the graph by $S$, we see immediately how to get to $C$:
BREADTH-FIRST SEARCH (BFS)

- Proceed layer by layer
- Find layer $d + 1$ by scanning neighbors of layer $d$
procedure $\text{bfs}(G, s)$

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$
Output: For all vertices $u$ reachable from $s$, $\text{dist}(u)$ is set to the distance from $s$ to $u$.

for all $u \in V$:
    $\text{dist}(u) = \infty$

$\text{dist}(s) = 0$

$Q = [s]$ (queue containing just $s$)

while $Q$ is not empty:
    $u = \text{eject}(Q)$
    for all edges $(u, v) \in E$:
        if $\text{dist}(v) = \infty$:
            $\text{inject}(Q, v)$
            $\text{dist}(v) = \text{dist}(u) + 1$
Let's run BFS on our graph, starting at $S$:
Let's run BFS on our graph, starting at $S$:

<table>
<thead>
<tr>
<th>Order of visitation</th>
<th>Queue contents after processing node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$[S]$</td>
</tr>
<tr>
<td>$A$</td>
<td>$[A \ C \ D \ E]$</td>
</tr>
<tr>
<td>$C$</td>
<td>$[C \ D \ E \ B]$</td>
</tr>
<tr>
<td>$D$</td>
<td>$[D \ E \ B]$</td>
</tr>
<tr>
<td>$E$</td>
<td>$[E \ B]$</td>
</tr>
<tr>
<td>$B$</td>
<td>$[B]$</td>
</tr>
</tbody>
</table>
IS BFS CORRECT?

That is, for each $d$, at some point:

- all nodes with distance $\leq d$ are correctly set
- all other nodes have distance set to $\infty$
- the queue contains exactly the nodes at distance $d$
BFS RUNNING TIME

- Each vertex put onto queue once
- Examine each edge once (directed) or twice (undirected)

\[ O(|V| + |E|) \]
Edges are often weighted with some kind of length:
We could compute shortest paths using BFS:

- Break edges into unit lengths with dummy nodes!
Suppose this is our graph:
We can break up each edge based on its length:

And then run BFS as usual
But what if this was our graph?
Then BFS is a slow, boring process:

\[ G' \]
Then BFS is a slow, boring process:

$G'$:

What if we could tell BFS to wake us up when it gets to the interesting part?
Wake us up when it gets to the interesting part:

\[ G:\]

\[ S \quad 100 \text{ } 50 \quad 200 \quad B \quad A \]
Wake us up when it gets to the interesting part:

$G:$

- ETAs for $A$ and $B$ are 100 and 200
Wake us up when it gets to the interesting part:

• ETAs for $A$ and $B$ are 100 and 200
• Set alarms for 100 and 200
Wake us up when it gets to the interesting part:

- ETAs for $A$ and $B$ are 100 and 200
- Set alarms for 100 and 200
- Wake up at time 100
Wake us up when it gets to the interesting part:

- ETAs for $A$ and $B$ are 100 and 200
- Set alarms for 100 and 200
- Wake up at time 100
- New ETA for $B$ is 150
Wake us up when it gets to the interesting part:

- ETAs for $A$ and $B$ are 100 and 200
- Set alarms for 100 and 200
- Wake up at time 100
- New ETA for $B$ is 150
- Change $B$'s alarm to 150
Wake us up when it gets to the interesting part:
Wake us up when it gets to the interesting part:

- Set alarm for node $s$ at time $0$
Wake us up when it gets to the interesting part:

- Set alarm for node $s$ at time 0
- Repeat until no more alarms:
Wake us up when it gets to the interesting part:

- Set alarm for node $s$ at time 0
- Repeat until no more alarms:
  
  Say the next alarm goes off at time $T$ for node $u$
Wake us up when it gets to the interesting part:

- Set alarm for node $s$ at time 0
- Repeat until no more alarms:
  
  Say the next alarm goes off at time $T$ for node $u$
  
  - Then the distance from $s$ to $u$ is $T$
Wake us up when it gets to the interesting part:

- Set alarm for node $s$ at time 0
- Repeat until no more alarms:

  Say the next alarm goes off at time $T$ for node $u$
  - Then the distance from $s$ to $u$ is $T$
  - For each neighbor $v$ of $u$: 
Wake us up when it gets to the interesting part:

- Set alarm for node $s$ at time 0
- Repeat until no more alarms:
  
  Say the next alarm goes off at time $T$ for node $u$
  
  - Then the distance from $s$ to $u$ is $T$
  - For each neighbor $v$ of $u$:
    - If there's no alarm for $v$, set one for $T + l(u, v)$
Wake us up when it gets to the interesting part:

- Set alarm for node $s$ at time 0
- Repeat until no more alarms:
  
  Say the next alarm goes off at time $T$ for node $u$

  - Then the distance from $s$ to $u$ is $T$
  - For each neighbor $v$ of $u$:
    - If there's no alarm for $v$, set one for $T + l(u, v)$
    - If $v$'s alarm is later than that, reduce it to this
This is essentially Dijkstra's algorithm!
(This is essentially Dijkstra)
DIJKSTRA'S ALGORITHM

Given a graph $G$ and a starting vertex $s$, find shortest paths to all reachable vertices
DIJKSTRA'S ALGORITHM

We need to use a priority queue with these operations:

- insert
- decrease-key
- delete-min
- make-queue
procedure dijkstra($G, l, s$)

Input: Graph $G = (V, E)$, directed or undirected; positive edge lengths $\{l_e : e \in E\}$; vertex $s \in V$

Output: For all vertices $u$ reachable from $s$, $dist(u)$ is set to the distance from $s$ to $u$.

for all $u \in V$:
   $dist(u) = \infty$
   $prev(u) = nil$
$dist(s) = 0$

$H = makequeue(V)$ (using $dist$-values as keys)

while $H$ is not empty:
   $u = deletemin(H)$
   for all edges $(u, v) \in E$:
      if $dist(v) > dist(u) + l(u, v)$:
         $dist(v) = dist(u) + l(u, v)$
         $prev(v) = u$
         decreasekey($H, v$)
Finally, we get this tree for paths from $A$:
Dijkstra's algorithm is basically just BFS:

• Instead of a regular queue,
• use a priority queue to account for lengths
Another way of looking at this:

- Start at $s$
- Grow the "known region" $R$ to larger distances
- Next node added to $R$ is the one closest to $s$
How do we find the next node $v$ outside $R$?

- Consider shortest path $s \rightarrow \cdots \rightarrow u \rightarrow v$
- $u$ closer to $s$ than $v$
- $u \in R$
- So $s \rightarrow v$ path extends a currently known path by one edge
How do we find the next node $v$ outside $R$?

- Try all single-edge extentions, find the shortest path
- Its endpoint is $v$

That is, $v$ is the node outside $R$ where $\text{distance}(s, u) + l(u, v)$ is minimized, for all $u \in R$
Initialize \( \text{dist}(s) \) to 0, other \( \text{dist}() \) values to \( \infty \)
\( R = \{ \} \) (the ‘‘known region’’)
while \( R \neq V \):
   Pick the node \( v \notin R \) with smallest \( \text{dist}(\cdot) \)
   Add \( v \) to \( R \)
   for all edges \( (v, z) \in E \):
      if \( \text{dist}(z) > \text{dist}(v) + l(v, z) \):
         \( \text{dist}(z) = \text{dist}(v) + l(v, z) \)
DIJKSTRA'S RUNNING TIME

- makequeue: at most $|V|$ insert operations
- $|V|$ deletemin operations
- $|V| + |E|$ insert/decreasekey operations

Time depends on implementation, but if we use a binary heap:

$$O((|V| + |E|) \log |V|)$$
How can we implement the priority queue?

- Array
- Binary heap
- $d$-ary heap
- Fibonacci heap
ARRAY

- insert, decreasekey: $O(1)$
- deletemin: $O(n)$
BINARY HEAP

- Complete binary tree (except last level)
- Key value $\leq$ that of its children
BINARY HEAP

- insert:
  - place node at bottom of tree and "bubble up"
- decreasekey:
  - "bubble up" (it's already in the tree)
- deletemin:
  - remove (and return) root
  - move last element to root
  - "sift down"
Let's say we have this initial heap:
We insert 7:
Bubble up:
Bubble up:
Now let's run delete-min:
Move last node to root:
Sift down:
Sift down:
**$d$-ARY HEAP**

- Like binary heap, but with $d$ children for each node
- Height reduces to

\[ \Theta \left( \log_d n \right) = \Theta \left( \left( \log n \right) / \left( \log d \right) \right) \]

- Insert/decreasekey slightly faster
- Deletemin slightly slower
# Which implementation is best?

| Implementation   | deletemin      | insert/decreasekey | $|V| \times \text{deletemin} + (|V| + |E|) \times \text{insert}$ |
|------------------|----------------|--------------------|------------------------------------------------------|
| Array            | $O(|V|)$       | $O(1)$             | $O(|V|^2)$                                            |
| Binary heap      | $O(\log |V|)$  | $O(\log |V|)$      | $O((|V| + |E|) \log |V|)$                              |
| $d$-ary heap     | $O\left(\frac{d \log |V|}{\log d}\right)$ | $O\left(\frac{\log |V|}{\log d}\right)$ | $O\left((|V| \cdot d + |E|) \frac{\log |V|}{\log d}\right)$ |
| Fibonacci heap   | $O(\log |V|)$  | $O(1)$ (amortized) | $O(|V| \log |V| + |E|)$                               |
Suppose now that the graph has negative edges:
Can we just shift everything into the positive range?
Note that the distances in Dijkstra's algorithm are always \( \leq \) the true distance.
procedure update \((u, v) \in E\)
\[
dist(v) = \min\{dist(v), dist(u) + l(u, v)\}
\]

- If \(u\) is the second-last node in shortest path to \(v\), gives exact distance
- Cannot set \(dist(v)\) smaller than the true distance

Dijkstra's algorithm can be seen as a sequence of these updates
Consider the shortest path from $s$ to some node $t$:

- This path has at most $|V| - 1$ edges
- If we did updates in this order:

$$(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_k, t)$$

we would get the correct distance to $t$. 
Consider the shortest path from $s$ to some node $t$:

- How can we update the right edges in the right order, without already knowing the shortest paths?
- Just update all edges $|V| - 1$ times!

$\Rightarrow$ Bellman-Ford algorithm, $O(|V| \cdot |E|)$. 
Bellman-Ford algorithm:

**procedure shortest-paths**\((G, l, s)\)

**Input:** Directed graph \(G = (V, E)\);
- edge lengths \(\{l_e : e \in E\}\) with no negative cycles;
- vertex \(s \in V\)

**Output:** For all vertices \(u\) reachable from \(s\), \(dist(u)\) is set to the distance from \(s\) to \(u\).

for all \(u \in V\):
- \(dist(u) = \infty\)
- \(prev(u) = \text{nil}\)

\(dist(s) = 0\)

repeat \(|V| - 1\) times:
  for all \(e \in E\):
    update\((e)\)
Bellman-Ford algorithm:

**Diagram:**
- Nodes: S, A, B, C, D, E, F, G
- Edges and weights:
  - S to A: 10
  - S to G: 8
  - A to S: 1
  - A to B: 2
  - A to C: 1
  - B to E: 1
  - C to B: 1
  - C to D: 3
  - D to E: 1
  - E to D: -1
  - F to E: -4
  - G to F: 1

**Table:**

<table>
<thead>
<tr>
<th>Node</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
</tr>
<tr>
<td>G</td>
<td>∞</td>
</tr>
</tbody>
</table>
What if there is a negative cycle?
Two kinds of graphs that cannot have negative cycles:

- graphs with no negative edges
- dags
SHORTEST PATHS IN DAGS

Key idea:

• In any path, the vertices appear in increasing linearized order
SHORTEST PATHS IN DAGS

- Linearize the dag using DFS
- Visit the vertices in sorted order
- Update all outgoing edges
procedure dag-shortest-paths\( (G, l, s) \)

Input: \( \text{Dag } G = (V, E); \)
edge lengths \( \{l_e : e \in E\} \); vertex \( s \in V \)

Output: For all vertices \( u \) reachable from \( s \), \( \text{dist}(u) \) is set to the distance from \( s \) to \( u \).

for all \( u \in V \):
\[
\text{dist}(u) = \infty
\]
\[
\text{prev}(u) = \text{nil}
\]

\( \text{dist}(s) = 0 \)
Linearize \( G \)
for each \( u \in V \), in linearized order:
for all edges \( (u, v) \in E \):
update \( (u, v) \)
Recap: How can we linearize a dag algorithmically?

- List nodes in decreasing order of post numbers
Order: $B, D, A, C, F, E$
Now let's add some lengths:

Order: $B, D, A, C, F, E$
$B, D, A, C, F, E$
\[ B, D, A, C, F, E \]

\[
\begin{array}{ccc}
\text{0} & \text{1} \\
A & \infty & 3 \\
B & 0 & 0 \\
C & \infty & \infty \\
D & \infty & 1 \\
E & \infty & \infty \\
F & \infty & \infty \\
\end{array}
\]
$B, D, A, C, F, E$
\[ B, D, A, C, F, E \]
\[ B, D, A, C, F, E \]