DFS = Depth first search
We used DFS for:
1) Connected components of undirected graphs
2) Topological sorting of DAGs
3) Strongly connected components (SCCs) of directed graphs

BFS = breadth first search
= explore graph in layers

Example:

Run BFS starting at B:

```
**BFS(G, s)**

**Input:** G = (V, E) in adj. list. representation (G can be directed or undirected)

**Output:** for all \( w \in V \), \( \text{dist}(w) = \min \) # of edges to go from \( s \) to \( w \)

for all \( w \in V \), \( \text{dist}(w) = \infty \)

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

\( Q = \{ s \} \) (create a new queue just containing \( s \))

While \( Q \neq \emptyset \)

\( w = \text{Dequeue}(Q) \)

for all \((w, z) \in E\)

if \( \text{dist}(z) = \infty \)

then \( \text{enqueue}(Q, z) \)

\( \text{dist}(z) = \text{dist}(w) + 1 \)

**Running time:** \( O(|V| + |E|) = O(n + m) \)

**Queue:**

\[ \text{enqueue} \quad \text{FIFO} = \text{first-in, first-out} \quad \text{dequeue} \]
Generalize BFS to allow positive lengths on edges.

Let \( l(e) = \text{length of edge } e \). 
\[ l(e) > 0 \]

For a path \( P : v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \), its length is:
\[ l(P) = \sum_{i=0}^{k-1} l(v_i, v_{i+1}) \]

\( \text{dist}(v, w) = \text{length of shortest path from } v \text{ to } w \)
\[ = \min_{P} \{ l(P) : P \text{ is a path from } v \text{ to } w \} \]

Goal: for given set \( V \), find \( \text{dist}(s, v) \) for all \( v \in V \).

BFS solves it when \( l(e) = 1 \) for all \( e \in E \).
Suppose every $l(e)$ is a positive integer.
For every edge $e$, replace $e$ by a path of length $l(e)$ & give each new edge length 1.
Run BFS on this new graph.

Problem: running time depends on lengths $l(e)$ which might be HUGE.

Most of the time BFS is processing "dummy" vertices that we added.
Can we skip such times & only consider times to process original vertices?
Set "alarm clocks" for times of interest.

Example:

```
\begin{center}
\begin{tikzpicture}
\node[shape=circle,draw=black] (a) at (0,0) {a};
\node[shape=circle,draw=black] (b) at (-1,-1) {b};
\node[shape=circle,draw=black] (c) at (1,-1) {c};
\node[shape=circle,draw=black] (d) at (0,-2) {d};
\node[shape=circle,draw=black] (e) at (0,-3) {e};
\node[shape=circle,draw=black] (f) at (-1,-4) {f};
\node[shape=circle,draw=black] (g) at (1,-4) {g};
\draw (a) -- (b) node [midway] {$10$};
\draw (b) -- (c) node [midway] {$2$};
\draw (c) -- (d) node [midway] {$4$};
\draw (d) -- (e) node [midway] {$10$};
\draw (e) -- (f) node [midway] {$2$};
\draw (f) -- (g) node [midway] {$4$};
\draw (g) -- (a) node [midway] {$2$};
\end{tikzpicture}
\end{center}
```
time 0: see s
1: see d & f_i
2: see d & f_k
3: 
4: see a & f_j
5: 
10: see b & e_v
11: 
12: see c

Idea: each vertex w has an alarm clock set at d(w) when we see a new path to w, check if need to adjust w's clock.

High-level algorithm:
Initialize: for s set its alarm for d(s)=0 for all w \in V - s, set d(w) = \infty
Repeat until no more alarms (ignore \infty-time alarms)
Let w be the next alarm clock to go off at time T.
Then:
set dist(w) = T
for every (w, z) \in E:
    if d(z) > T + l(w, z)
    then d(z) = T + l(w, z).
To implement alarm clocks, use min-heaps data structure priority queue.

**Min-heap data structure** H:

- H contains a set of elements (in our case, element = vertex)
- each element has a key (in our case, key = dist(w))

**Operations**:

- **Insert (H, w, dist(w))**: add element w with key dist(w) to H

- **Decrease key (H, w, dist(w))**: for w ∈ H, decrease its key to dist(w)

- **Delete Min (H)**: output the element in H with smallest & delete it from H

For a heap with ≤ n elements, \(O(\log n)\) time per operation.
Dijkstra (G, l, s)

input: \( G = (V,E) \) with \( l(e) > 0 \) for each \( e \in E \)
& vertex \( s \in V \)

output: for all \( w \in V \)

\[ \text{dist}(w) = \text{length of shortest } s \rightarrow w \text{ path} \]
\[ \text{Prev}(w) = \text{parent of } w \text{ on this path} \]

for all \( w \in V \), \[
\begin{align*}
\text{dist}(w) &= 0 \\
\text{Prev}(w) &= \text{NULL}
\end{align*}
\]

\[ \text{dist}(s) = 0 \]
\[ H = \emptyset \]
for all \( w \in V \), \[ \text{Insert}(H, w, \text{dist}(w)) \]
while \( H \neq \emptyset \):

\[ w = \text{deleteMin}(H) \]

for all \( (w, z) \in E \):

if \[ \text{dist}(w) > \text{dist}(w) + l(w, z) \]

then:

\[ \begin{align*}
\text{dist}(z) &= \text{dist}(w) + l(w, z) \\
\text{Prev}(z) &= w \\
& \text{Decrease key } (H, z, \text{dist}(z))
\end{align*} \]
Running time:

- \( n \) inserts & \( n \) deletes \( \Rightarrow O(n \log n) \) time.
- \( \leq M \) decrease keys \( \Rightarrow O(m \log n) \) time

\( \Rightarrow O((n+m) \log n) \) total time.

if \( G \) is connected then

\( m \geq n-1 \), so it's \( O(m \log n) \) total time.

How to implement min-heaps?

Use complete binary tree: every level is full except possibly the bottom level

fill bottom level from left to right

Example:

```
      1
     / \    
    2   3
   / \ /\  
  4  5 6 7
 / \ /   /  
8 9 10 11 12 ...
```

Can use an array \( A[1 \ldots 12] \) to store the keys for tree
For node in position $i$ of $A_{\Sigma I}$
Parent is in position $\left\lfloor \frac{i}{2} \right\rfloor$
Left child is at $2i$,
Right child is at $2i + 1$.

Let $p(i) = \left\lfloor \frac{i}{2} \right\rfloor$

Min-heap property: $A[p(i)] \leq A[i]$
Min key is at root.

Insert: add new element to bottom level, leftmost open spot
This maintains desired shape
Then "bubble-up" to maintain min-heap property
To decrease-key: lower its value & then Bubble-up

To delete-Min:
1) output value at root
2) Delete root
3) Take element from bottom-right & put it at the root
4) Sift-down:

Example: