Single Source Shortest Paths

Priority Queues and Dijkstra’s Algorithm

Benjamin Sach
In today’s lecture we’ll be discussing the **single source shortest paths** problem in a weighted, directed graph...

![Map of Bristol](image)

*The shortest path from MVB to Temple Meads (according to Google Maps)*

In particular we’ll be interested in **Dijkstra’s Algorithm** which is based on an **abstract data-structure** called a **priority queue** 

... which can be efficiently implemented as a **binary heap**
In today’s lecture we’ll be discussing the **single source shortest paths** problem in a weighted, directed graph.

In particular we’ll be interested in **Dijkstra’s Algorithm** which is based on an **abstract data-structure** called a **priority queue**...
Part one
Priority Queues
Part one
Priority Queues

(you can forget all about graphs for the whole of part one)
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$.
Priority Queues

A **priority queue**, \( Q \) stores a set of distinct elements

Each element \( x \) has an associated value called its key - \( x.key \)

A priority queue supports the following operations:
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$.

A priority queue supports the following operations:

**INSERT**($x, k$) - inserts $x$ with $x.key = k$

**DECREASEKEY**($x, k$) - decreases the value of $x.key$ to $k$

where $k < x.key$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements. Each element $x$ has an associated value called its key - $x$.key.

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x$.key $=$ $k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x$.key to $k$

\[ \text{where } k < x\.key \]

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

\[ (\text{ties are broken arbitrarily}) \]
Priority Queues

A priority queue, $Q$ stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

\quad where $k < x.key$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

\quad (ties are broken arbitrarily)

A priority queue:

$\text{Alice} \ 3$

$\text{Bob} \ 5$

$\text{Chris} \ 8$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x\text{.key}$

A priority queue supports the following operations:

- **INSERT**($x$, $k$) - inserts $x$ with $x\text{.key} = k$
- **DECREASEKEY**($x$, $k$) - decreases the value of $x\text{.key}$ to $k$
  
  where $k < x\text{.key}$
- **EXTRACTMIN()** - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

- Alice: 3
- Bob: 5
- Chris: 8

**INSERT**(Dawn, 4)
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x$.key

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x$.key = $k$

$\text{DECREASEKey}(x, k)$ - decreases the value of $x$.key to $k$

$\text{where } k < x$.key

$\text{EXTRACTMin()}$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

A priority queue: Alice 3 Bob 5 Chris 8 Dawn 4

$\text{INSERT}(\text{Dawn}, 4)$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

**INSERT**($x, k$) - inserts $x$ with $x.key = k$

**DECREASEKEY**($x, k$) - decreases the value of $x.key$ to $k$

where $k < x.key$

**EXTRACTMIN**() - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Chris</th>
<th>Dawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

$(x, x.key)$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

\[
\begin{align*}
\text{INSERT}(x, k) & \quad \text{- inserts } x \text{ with } x.key = k \\
\text{DECREASEKEY}(x, k) & \quad \text{- decreases the value of } x.key \text{ to } k \\
& \quad \text{where } k < x.key \\
\text{EXTRACTMIN() } & \quad \text{- removes and returns the element with the smallest key} \\
& \quad \text{(ties are broken arbitrarily)}
\end{align*}
\]

A priority queue:

\[
\begin{array}{cccc}
\text{Alice} & \text{Bob} & \text{Chris} & \text{Dawn} \\
3 & 5 & 8 & 4
\end{array}
\]

\[
\text{EXTRACTMIN()}
\]
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

- \textsc{Insert}(x, k) - inserts $x$ with $x.key = k$

- \textsc{DecreaseKey}(x, k) - decreases the value of $x.key$ to $k$
  \hspace{1cm} where $k < x.key$

- \textsc{ExtractMin}() - removes and returns the element with the smallest key
  \hspace{1cm} (ties are broken arbitrarily)

A priority queue:

```
  Bob 5
  Chris 8
  Dawn 4
```

\textsc{ExtractMin}()
Priority Queues

A **priority queue**, \( Q \), stores a set of distinct elements

Each element \( x \) has an associated value called its key - \( x.\text{key} \)

A priority queue supports the following operations:

- **INSERT\((x, k)\)** - inserts \( x \) with \( x.\text{key} = k \)
- **DECREASE\(\text{KEY}\)(\(x, k\))** - decreases the value of \( x.\text{key} \) to \( k \)
  \( \text{where } k < x.\text{key} \)
- **EXTRACT\(\text{MIN}\)()** - removes and returns the element with the smallest key
  \( \text{(ties are broken arbitrarily)} \)

A priority queue:

\[
\begin{array}{ccc}
\text{Bob} & \text{Chris} & \text{Dawn} \\
5 & 8 & 4
\end{array}
\]
Priority Queues

A priority queue, \( Q \), stores a set of distinct elements. Each element \( x \) has an associated value called its key - \( x.key \).

A priority queue supports the following operations:

- **INSERT\((x, k)\)** - inserts \( x \) with \( x.key = k \)
- **DECREASE\(\text{KEY}\)(\(x, k\))** - decreases the value of \( x.key \) to \( k \)
  
  \[ \text{where } k < x.key \]

- **EXTRACT\(\text{MIN}\)()** - removes and returns the element with the smallest key
  
  (ties are broken arbitrarily)

A priority queue:

\[
\begin{array}{ccc}
\text{Bob} & \text{Chris} & \text{Dawn} \\
5 & 8 & 4
\end{array}
\]

\( \text{INSERT(Eve, 6)} \)
Priority Queues

A priority queue, \( Q \), stores a set of distinct elements

Each element \( x \) has an associated value called its key - \( x.key \)

A priority queue supports the following operations:

\[ \textup{INSERT}(x, k) - \text{inserts } x \text{ with } x.key = k \]

\[ \textup{DECREASEKEY}(x, k) - \text{decreases the value of } x.key \text{ to } k \]

\[ \text{where } k < x.key \]

\[ \textup{EXTRACTMIN}() - \text{removes and returns the element with the smallest key} \]

\[ (\text{ties are broken arbitrarily}) \]

A priority queue:

\[
\begin{array}{cccc}
\text{Bob} & \text{Chris} & \text{Dawn} & \text{Emma} \\
5 & 8 & 4 & 6 \\
\end{array}
\]

\[ \textup{INSERT}(\text{Eve}, 6) \]
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

$\text{where } k < x.key$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

Bob 5
Chris 8
Dawn 4
Emma 6
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

\[ \text{INSERT}(x, k) \] - inserts $x$ with $x.key = k$

\[ \text{DECREASEKEY}(x, k) \] - decreases the value of $x.key$ to $k$

\[ \text{EXTRACTMIN()} \] - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

EXTRACTMIN()}
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

where $k < x.key$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

\[
\begin{array}{ccc}
\text{Bob} & \text{Chris} & \text{Emma} \\
5 & 8 & 6
\end{array}
\]

\[x \quad \underline{x.key} \quad \text{\textsc{ExtractMin}()}\]
Priority Queues

A priority queue, \( Q \) stores a set of distinct elements

Each element \( x \) has an associated value called its key - \( x.key \)

A priority queue supports the following operations:

\[
\text{INSERT}(x, k) \quad - \text{inserts} \ x \ \text{with} \ x.key = k
\]

\[
\text{DECREASEKEY}(x, k) \quad - \text{decreases the value of} \ x.key \ \text{to} \ k
\]

\[
\text{where} \ k < x.key
\]

\[
\text{EXTRACTMIN()} \quad - \text{removes and returns the element with the smallest key}
\]

\[
(ties \ are \ broken \ arbitrarily)
\]
Priority Queues

A *priority queue*, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

$(ties \ are \ broken \ arbitrarily)$

A priority queue:

$\begin{align*}
\text{Bob} & \quad 5 \\
\text{Chris} & \quad 8 \\
\text{Emma} & \quad 6
\end{align*}$

$\text{DECREASEKEY}(\text{Bob}, 2)$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

$\text{where } k < x.key$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

Bob 2  Chris 8  Emma 6

$\text{DECREASEKEY}(\text{Bob}, 2)$
Priority Queues

A priority queue, \( Q \), stores a set of distinct elements

Each element \( x \) has an associated value called its key - \( x.key \)

A priority queue supports the following operations:

**INSERT\((x, k)\)** - inserts \( x \) with \( x.key = k \)

**DECREASE\(K\)EY\((x, k)\)** - decreases the value of \( x.key \) to \( k \)

\[ \text{where } k < x.key \]

**EXTRACT\(M\)IN()** - removes and returns the element with the smallest key

\( (ties \ are \ broken \ arbitrarily) \)

A priority queue:

Bob 2  Chris 8  Emma 6

\( x \)

\( x.key \)
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$.

A priority queue supports the following operations:

- $\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$
- $\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$
  \[ \text{where } k < x.key \]
- $\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key
  \[ (\text{ties are broken arbitrarily}) \]

A priority queue:

\[
\begin{array}{ccc}
\text{Bob} & \text{Chris} & \text{Emma} \\
2 & 8 & 6 \\
\end{array}
\]

$\text{INSERT}(\text{Alice}, 3)$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x$\textunderscore key.

A priority queue supports the following operations:

- **INSERT($x, k$)** - inserts $x$ with $x$\textunderscore key $= k$
- **DECREASEKEY($x, k$)** - decreases the value of $x$\textunderscore key to $k$
  
  where $k < x$\textunderscore key
- **EXTRACTMIN()** - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

- Alice 3
- Bob 2
- Chris 8
- Emma 6

\[ \text{INSERT}(\text{Alice, 3}) \]
Priority Queues

A priority queue, \( Q \), stores a set of distinct elements.

Each element \( x \) has an associated value called its key - \( x\.key \).

A priority queue supports the following operations:

\[ \text{INSERT}(x, k) \] - inserts \( x \) with \( x\.key = k \)

\[ \text{DECREASEKEY}(x, k) \] - decreases the value of \( x\.key \) to \( k \)

\[ \text{DECREASEKEY}(x, k) \text{ where } k < x\.key \]

\[ \text{EXTRACTMIN()} \] - removes and returns the element with the smallest key

\[ \text{(ties are broken arbitrarily)} \]

A priority queue:

\[
\begin{array}{cccc}
\text{Alice} & \text{Bob} & \text{Chris} & \text{Emma} \\
3 & 2 & 8 & 6
\end{array}
\]
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

where $k < x.key$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

- Alice 3
- Bob 2
- Chris 8
- Emma 6

$\text{EXTRACTMIN}()$
Priority Queues

A priority queue, \(Q\), stores a set of distinct elements.

Each element \(x\) has an associated value called its key - \(x.key\).

A priority queue supports the following operations:

- **INSERT** \((x, k)\) - inserts \(x\) with \(x.key = k\).
- **DECREASEKEY** \((x, k)\) - decreases the value of \(x.key\) to \(k\) where \(k < x.key\).
- **EXTRACTMIN()** - removes and returns the element with the smallest key (ties are broken arbitrarily).

A priority queue:

\[
\begin{array}{|c|c|c|}
\hline
& Alice & Chris & Emma \\
\hline
3 & 8 & 6 \\
\hline
\end{array}
\]

\(\text{EXTRACTMIN()}\)
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

\[\text{INSERT}(x, k)\] - inserts $x$ with $x.key = k$

\[\text{DECREASEKEY}(x, k)\] - decreases the value of $x.key$ to $k$
  where $k < x.key$

\[\text{EXTRACTMIN}()\] - removes and returns the element with the smallest key
  (ties are broken arbitrarily)

A priority queue:

\[
\begin{align*}
\text{Alice} & \quad 3 \\
\text{Chris} & \quad 8 \\
\text{Emma} & \quad 6
\end{align*}
\]
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

**INSERT** $(x, k)$ - inserts $x$ with $x.key = k$

**DECREASEKEY** $(x, k)$ - decreases the value of $x.key$ to $k$

\[ \text{where } k < x.key \]

**EXTRACTMIN()** - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

- Alice
  - key: 3
- Chris
  - key: 8
- Emma
  - key: 6

**DECREASEKEY**(Chris, 4)
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

$\text{where } k < x.key$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

Alice 3

Chris 4

Emma 6

$\text{DECREASEKEY}(\text{Chris}, 4)$
Priority Queues

A priority queue, \( Q \), stores a set of distinct elements.

Each element \( x \) has an associated value called its key - \( x.key \).

A priority queue supports the following operations:

\[ \text{INSERT}(x, k) \text{ - inserts } x \text{ with } x.key = k \]

\[ \text{DECREASEKEY}(x, k) \text{ - decreases the value of } x.key \text{ to } k \]

\[ \text{where } k < x.key \]

\[ \text{EXTRACTMIN()} \text{ - removes and returns the element with the smallest key} \]

\[ \text{ties are broken arbitrarily} \]

A priority queue:

\[ \begin{array}{ccc}
\text{Alice} & \text{Chris} & \text{Emma} \\
3 & 4 & 6 \\
\end{array} \]
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

- $\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

- $\text{DECREASE KEY}(x, k)$ - decreases the value of $x.key$ to $k$ where $k < x.key$

- $\text{EXTRACT MIN}()$ - removes and returns the element with the smallest key (ties are broken arbitrarily)

A priority queue:

\[
\begin{bmatrix}
\text{Alice} & 3 \\
\text{Chris} & 4 \\
\text{Emma} & 6 \\
\end{bmatrix}
\]

$\text{EXTRACT MIN}()$
Priority Queues

A priority queue, \( Q \), stores a set of distinct elements.

Each element \( x \) has an associated value called its key - \( x\.key \).

A priority queue supports the following operations:

- **INSERT** \((x, k)\) - inserts \( x \) with \( x\.key = k \)
- **DECREASEKEY** \((x, k)\) - decreases the value of \( x\.key \) to \( k \)
  where \( k < x\.key \)
- **EXTRACTMIN**() - removes and returns the element with the smallest key
  (ties are broken arbitrarily)

A priority queue:

\[
\begin{align*}
\text{Chris} & : 4 \\
\text{Emma} & : 6
\end{align*}
\]

\( \text{EXTRACTMIN}() \)
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x$.key

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x$.key = $k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x$.key to $k$

where $k < x$.key

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

$$\begin{array}{l}
\text{Chris} & 4 \\
\text{Emma} & 6 \\
\end{array}$$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$.

A priority queue supports the following operations:

- **INSERT($x, k$)** - inserts $x$ with $x.key = k$
- **DECREASEKEY($x, k$)** - decreases the value of $x.key$ to $k$
  where $k < x.key$
- **EXTRACTMIN()** - removes and returns the element with the smallest key
  (ties are broken arbitrarily)

A priority queue:

```
Chris 4
```

```
Emma 6
```

```
( x  
  x.key )
```

**EXTRACTMIN()**
A priority queue, \( Q \), stores a set of distinct elements.

Each element \( x \) has an associated value called its key - \( x.key \).

A priority queue supports the following operations:

**INSERT(\( x, k \))** - inserts \( x \) with \( x.key = k \)

**DECREASEKEY(\( x, k \))** - decreases the value of \( x.key \) to \( k \)

\( x < k \)

**EXTRACTMIN()** - removes and returns the element with the smallest key

(ties are broken arbitrarily)
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

$\text{where} \quad k < x.key$

$\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

$(\text{ties are broken arbitrarily})$

A priority queue:

```
(Emma 6)
```
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

$\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$

$\text{where } k < x.key$

$\text{EXTRACTMIN()}$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A priority queue:

$\text{EXTRACTMIN()}$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$.

A priority queue supports the following operations:

- **INSERT($x, k$)** - inserts $x$ with $x.key = k$

- **DECREASEKEY($x, k$)** - decreases the value of $x.key$ to $k$
  
  where $k < x.key$

- **EXTRACTMIN()** - removes and returns the element with the smallest key
  
  (ties are broken arbitrarily)

A priority queue:

$$
\begin{array}{c}
\text{EXTRACTMIN()}
\\
(x)
\\
(x.key)
\end{array}
$$
Priority Queues

A priority queue, $Q$, stores a set of distinct elements

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

- $\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

- $\text{DECREASEKEY}(x, k)$ - decreases the value of $x.key$ to $k$
  
  where $k < x.key$

- $\text{EXTRACTMIN}()$ - removes and returns the element with the smallest key

  (ties are broken arbitrarily)

A priority queue:
Priority Queues

A priority queue, $Q$, stores a set of distinct elements.

Each element $x$ has an associated value called its key - $x.key$

A priority queue supports the following operations:

\[
\text{\textbf{INSERT}}(x, k) - \text{inserts } x \text{ with } x.key = k
\]

\[
\text{\textbf{DECREASEKEY}}(x, k) - \text{decreases the value of } x.key \text{ to } k
\]

\[
\text{where } k < x.key
\]

\[
\text{\textbf{EXTRACTMIN}}() - \text{removes and returns the element with the smallest key}
\]

\[
\text{(ties are broken arbitrarily)}
\]
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... \textit{but they aren’t all efficient}

Let $n$ denote the number of elements in the queue
- \textit{our goal is to implement a queue with operations which scale well as }$n$\textit{ grows}
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let $n$ denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as $n$ grows
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let \( n \) denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as \( n \) grows

We could implement a Priority Queue using an unsorted linked list:
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient.

Let $n$ denote the number of elements in the queue.

- our goal is to implement a queue with operations which scale well as $n$ grows.

We could implement a Priority Queue using an unsorted linked list:
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren't all efficient.

Let $n$ denote the number of elements in the queue.
- our goal is to implement a queue with operations which scale well as $n$ grows.

We could implement a Priority Queue using an unsorted linked list:

\[ \text{INSERT} \text{ is very efficient,} \]
- add the new item to the head of the list in $O(1)$ time.
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... *but they aren’t all efficient*

Let \( n \) denote the number of elements in the queue

- *our goal is to implement a queue with operations which scale well as \( n \) grows*

We could implement a Priority Queue using an unsorted linked list:

\[
\begin{array}{c}
\text{Chris} & \text{Emma} & \text{Bob} & \text{Dawn} & \text{Alice} \\
7 & 6 & 5 & 4 & 3 \\
\end{array}
\]

**INSERT** is very efficient,

- add the new item to the head of the list in \( O(1) \) time
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let \( n \) denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as \( n \) grows

We could implement a Priority Queue using an unsorted linked list:

\[
\begin{align*}
\text{Chris} & \quad \text{Emma} & \quad \text{Bob} & \quad \text{Dawn} & \quad \text{Alice} \\
7 & \quad 6 & \quad 5 & \quad 4 & \quad 3
\end{align*}
\]

\text{INSERT} is very efficient,
- add the new item to the head of the list in \( O(1) \) time

\text{EXTRACTMIN} and \text{DECREASEKEY} are very inefficient, they take \( O(n) \) time
- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren't all efficient

Let $n$ denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as $n$ grows

We could implement a Priority Queue using an unsorted linked list:

![Linked List Diagram]

**INSERT** is very efficient,
- add the new item to the head of the list in $O(1)$ time

**EXTRACTMIN** and **DECREASEKEY** are very inefficient, they take $O(n)$ time
- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let \( n \) denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as \( n \) grows

We could implement a Priority Queue using an unsorted linked list:

\[
\begin{array}{c}
\text{Chris} & \text{Emma} & \text{Bob} & \text{Dawn} & \text{Alice} \\
7 & 6 & 5 & 4 & 3
\end{array}
\]

**INSERT** is very efficient,
- add the new item to the head of the list in \( O(1) \) time

**EXTRACT\text{MIN}** and **DECREASE\text{KEY}** are very *inefficient*, they take \( O(n) \) time
- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... *but they aren’t all efficient*

Let $n$ denote the number of elements in the queue

- *our goal is to implement a queue with operations which scale well as $n$ grows*

We could implement a Priority Queue using an unsorted linked list:

```
  Chris 7 → Emma 6 → Bob 5 → Dawn 4 → Alice 3
```

**INSERT** is very efficient,

- add the new item to the head of the list in $O(1)$ time

**EXTRACTMIN** and **DECREASEKEY** are very inefficient, they take $O(n)$ time

- we have to look through the entire linked list to find an item *(in the worst case)*
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let $n$ denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as $n$ grows

We could implement a Priority Queue using an unsorted linked list:

- **INSERT** is very efficient,
  - add the new item to the head of the list in $O(1)$ time

- **ExtractMin** and **DecreaseKey** are very inefficient, they take $O(n)$ time
  - we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let $n$ denote the number of elements in the queue
- *our goal is to implement a queue with operations which scale well as $n$ grows*

We could implement a Priority Queue using an unsorted linked list:

INSERT is very efficient,
- add the new item to the head of the list in $O(1)$ time

EXTRACTMIN and DECREASEKEY are very inefficient, they take $O(n)$ time
- we have to look through the entire linked list to find an item *(in the worst case)*
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let $n$ denote the number of elements in the queue
- **our goal is to implement a queue with operations which scale well as $n$ grows**

We could implement a Priority Queue using an unsorted linked list:

```
  Chris 7 -> Emma 6 -> Bob 5 -> Dawn 4 -> Alice 3

  (x
   x.key)
```

**INSERT** is very efficient,
- add the new item to the head of the list in $O(1)$ time

**ExtractMin** and **DecreaseKey** are very inefficient, they take $O(n)$ time
- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue...  

but they aren’t all efficient

Let $n$ denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as $n$ grows

Instead,
We could implement a Priority Queue using a sorted linked list:

```
(Alice: 3) -> (Dawn: 4) -> (Bob: 5) -> (Emma: 6)
```

$(x: x.key)$
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let \( n \) denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as \( n \) grows

Instead,
We could implement a Priority Queue using a sorted linked list:

ExtractMin is very efficient,
- remove the head of the list in \( O(1) \) time
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let \( n \) denote the number of elements in the queue

- our goal is to implement a queue with operations which scale well as \( n \) grows

Instead,

We could implement a Priority Queue using a sorted linked list:

\[
\begin{array}{c}
\text{Dawn} & \text{Bob} & \text{Emma} \\
4 & 5 & 6 \\
\end{array}
\]

\( \text{ExtractMin} \) is very efficient,

- remove the head of the list in \( O(1) \) time
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let $n$ denote the number of elements in the queue

- our goal is to implement a queue with operations which scale well as $n$ grows

Instead,

We could implement a Priority Queue using a sorted linked list:

![Linked List Diagram]

**ExtractMin** is very efficient,

- remove the head of the list in $O(1)$ time

**Insert** and **DecreaseKey** are very inefficient, they take $O(n)$ time

- we have to look through the entire linked list (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue. . .

*but they aren’t all efficient*

Let $n$ denote the number of elements in the queue

- *our goal is to implement a queue with operations which scale well as $n$ grows*

Instead,

We could implement a Priority Queue using a *sorted* linked list:

- ExtractMin is very efficient,
  - remove the head of the list in $O(1)$ time

- Insert and DecreaseKey are very inefficient, they take $O(n)$ time
  - we have to look through the entire linked list
  
*(in the worst case)*
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let $n$ denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as $n$ grows

Instead,
We could implement a Priority Queue using a sorted linked list:

\[ x \]
\[ x.\text{key} \]

**ExtractMin** is very efficient,
- remove the head of the list in $O(1)$ time

**Insert** and **DecreaseKey** are very inefficient, they take $O(n)$ time
- we have to look through the entire linked list (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... 
but they aren’t all efficient

Let $n$ denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as $n$ grows

Instead,
We could implement a Priority Queue using a sorted linked list:

\[ \begin{array}{c}
\text{Chris} & 7 \\
\text{Dawn} & 4 & \text{Bob} & 5 & \text{Emma} & 6 \\
\end{array} \]

\[ \text{ExtractMin} \text{ is very efficient,} \]
- remove the head of the list in $O(1)$ time

\[ \text{Insert} \text{ and DecreaseKey} \text{ are very inefficient, they take } O(n) \text{ time} \]
- we have to look through the entire linked list (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue . . .

but they aren’t all efficient

Let $n$ denote the number of elements in the queue

- our goal is to implement a queue with operations which scale well as $n$ grows

Instead,

We could implement a Priority Queue using a sorted linked list:

ExtractMin is very efficient,

- remove the head of the list in $O(1)$ time

Insert and DecreaseKey are very inefficient, they take $O(n)$ time

- we have to look through the entire linked list
  (in the worst case)
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren’t all efficient

Let $n$ denote the number of elements in the queue

- our goal is to implement a queue with operations which scale well as $n$ grows

Instead,

We could implement a Priority Queue using a sorted linked list:

```
    Chris
     7
       ↓
      Dawn
       4
       ↓
      Bob
      5
      ↓
    Emma
     6
     \( x \)
```

\textbf{ExtractMin} is very efficient,

- remove the head of the list in \(O(1)\) time

\textbf{Insert} and \textbf{DecreaseKey} are very \textit{inefficient}, they take \(O(n)\) time

- we have to look through the entire linked list \textit{(in the worst case)}
Using a Linked List as a Priority Queue

There are many ways in which we could implement a priority queue... but they aren't all efficient

Let \( n \) denote the number of elements in the queue
- our goal is to implement a queue with operations which scale well as \( n \) grows

Instead,

We could implement a Priority Queue using a sorted linked list:

\[ \begin{align*}
\text{Chris} & \quad 7 \\
\text{Dawn} & \quad 4 \\
\text{Bob} & \quad 5 \\
\text{Emma} & \quad 6
\end{align*} \]

\text{ExtractMin} is very efficient,
- remove the head of the list in \( O(1) \) time

\text{Insert} and \text{DecreaseKey} are very inefficient, they take \( O(n) \) time
- we have to look through the entire linked list (in the worst case)
A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right.
A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right.
A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right.

The height is \( O(\log n) \)

\( n \) is the number of elements in the Heap.
A binary heap is an ‘almost complete’ binary tree, where every level is full… except (possibly) the lowest, which is filled from left to right.

The height is $O(\log n)$

$n$ is the number of elements in the Heap
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right

The height is $O(\log n)$

$n$ is the number of elements in the Heap

**Heap Property** Any element has a key *less than or equal to* the keys of its children.
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right

The height is $O(\log n)$

$n$ is the number of elements in the Heap

**Heap Property** Any element has a key less than or equal to the keys of its children.
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right

The height is $O(\log n)$

$n$ is the number of elements in the Heap

**Heap Property** Any element has a key less than or equal to the keys of its children.
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full . . .
except (possibly) the lowest, which is filled from left to right.

The height is $O(\log n)$

$n$ is the number of elements in the Heap.

Heap Property Any element has a key less than or equal to the keys of its children.
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right.

The height is $O(\log n)$

$n$ is the number of elements in the Heap

**Heap Property** Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array $A$: 
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right.

The height is \(O(\log n)\)

\(n\) is the number of elements in the Heap.

**Heap Property** Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array \(A\):

```
2 2 3 5 3 4 6 7 6 5 4 9
```
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right

The height is $O(\log n)$

$n$ is the number of elements in the Heap

**Heap Property** Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array $A$:

```
 2 2 3 5 3 4 6 7 6 5 4 9
```
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right.

The height is $O(\log n)$.

$n$ is the number of elements in the Heap.

**Heap Property** Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array $A$:
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full… *except (possibly) the lowest, which is filled from left to right*

The height is $O(\log n)$

$n$ is the number of elements in the Heap

**Heap Property** Any element has a key *less than or equal to* the keys of its children.

A binary heap can be efficiently stored implicitly as an array $A$:
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right.

the height is \( O(\log n) \)

\( n \) is the number of elements in the Heap

Heap Property Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array \( A \):

2 2 3 5 3 4 6 7 6 5 4 9
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right

the height is $O(\log n)$

$n$ is the number of elements in the Heap

Heap Property Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array $A$:

```
2 2 3 5 3 4 6 7 6 5 4 9
```
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full . . . except (possibly) the lowest, which is filled from left to right.

The height is \( O(\log n) \)

\( n \) is the number of elements in the Heap

**Heap Property** Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array \( A \):

\[
\begin{array}{cccccccccccc}
2 & 2 & 3 & 5 & 3 & 4 & 6 & 7 & 6 & 5 & 4 & 9 \\
\end{array}
\]

Moving around using: \( \text{PARENT}(i) = \lfloor i/2 \rfloor \)  \( \text{LEFT}(i) = 2i \)  \( \text{RIGHT}(i) = 2i + 1 \)
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full... except (possibly) the lowest, which is filled from left to right

the height is \( O(\log n) \)

\( n \) is the number of elements in the Heap

**Heap Property** Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array \( A \):

\[
\begin{array}{cccccccc}
2 & 2 & 3 & 5 & 3 & 4 & 6 & 7 & 6 & 5 & 4 & 9 \\
\end{array}
\]

Moving around using: \( \text{PARENT}(i) = \lfloor i/2 \rfloor \quad \text{LEFT}(i) = 2i \quad \text{RIGHT}(i) = 2i + 1 \)
Binary Heaps

A binary heap is an ‘almost complete’ binary tree, where every level is full . . .
except (possibly) the lowest, which is filled from left to right

The height is \( O(\log n) \)

\( n \) is the number of elements in the Heap

**Heap Property** Any element has a key less than or equal to the keys of its children.

A binary heap can be efficiently stored implicitly as an array \( A \):

\[
\begin{array}{cccccccccc}
2 & 2 & 3 & 5 & 3 & 4 & 6 & 7 & 6 & 5 & 4 & 9
\end{array}
\]

Moving around using: \( \text{PARENT}(i) = \lfloor i/2 \rfloor \), \( \text{LEFT}(i) = 2i \), \( \text{RIGHT}(i) = 2i + 1 \)
Using a Binary Heap as a Priority Queue

We will now see how to use a Binary Heap to implement the required operations:

\[ \text{INSERT}(x, k) \] - inserts \( x \) with \( x.key = k \)

\[ \text{DECREASE KEY}(x, k) \] - decreases the value of \( x.key \) to \( k \) where \( k < x.key \)

\[ \text{EXTRACT MIN}() \] - removes and returns the element with the smallest key 
\[ (\text{ties are broken arbitrarily}) \]

Each in \( O(\log n) \) time per operation

Assumption we can find the location of any element \( x \) in the Heap in \( O(1) \) time

This is a little fiddly but essentially you just build a look-up table
**DECREASEKEY with a Binary Heap**

**DECREASEKEY** \((x, k)\) - decreases the value of \(x.key\) to \(k\)

where \(k < x.key\)

- **Step 1:** Find element \(x\)
- **Step 2:** Check that \(k \leq x.key\), otherwise raise an error
- **Step 3:** Set \(x.key = k\)
- **Step 4:** While \(x.key\) is smaller than its parent’s: (stop if \(x\) becomes the root) swap \(x\) with its parent
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**(*x, k*) - decreases the value of *x.key* to *k*  

where *k < x.key*

---

**DECREASEKEY**(*E, 2*)

Step 1: Find element *x*

Step 2: Check that *k ≤ x.key*, otherwise raise an error

Step 3: Set *x.key = k*

Step 4: While *x.key* is smaller than its parent’s: *(stop if x becomes the root)*  
swap *x* with its parent
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**\((x, k)\) - decreases the value of \(x.key\) to \(k\) where \(k < x.key\)

**DECREASEKEY**\((E, 2)\)

**Step 1:** Find element \(x\)

**Step 2:** Check that \(k \leq x.key\), otherwise raise an error

**Step 3:** Set \(x.key = k\)

**Step 4:** While \(x.key\) is smaller than its parent’s: (stop if \(x\) becomes the root) swap \(x\) with its parent
**DECREASEKEY** with a Binary Heap

**DECREASEKEY**\((x, k)\) - decreases the value of \(x.key\) to \(k\)

where \(k < x.key\)

**DECREASEKEY**\((E, 2)\)

\[
\begin{array}{c}
2 \\
/ \\
5 3 \\
/ / \\
7 6 5 4 \\
/ / \\
4 6 \\
/ \\
3
\end{array}
\]

Step 1: Find element \(x\)

Step 2: Check that \(k \leq x.key\), otherwise raise an error

Step 3: Set \(x.key = k\)

Step 4: While \(x.key\) is smaller than its parent’s: *(stop if \(x\) becomes the root)*

swap \(x\) with its parent
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**\((x, k)\) - decreases the value of \(x.key\) to \(k\)

\[\text{where } k < x.key\]

**DECREASEKEY**\((E, 2)\)

---

**Step 1:** Find element \(x\)

**Step 2:** Check that \(k \leq x.key\), otherwise raise an error

**Step 3:** Set \(x.key = k\)

**Step 4:** While \(x.key\) is smaller than its parent’s: (stop if \(x\) becomes the root)

swap \(x\) with its parent
**DECREASEKey with a Binary Heap**

**DECREASEKey** \((x, k)\) - decreases the value of \(x.key\) to \(k\)

\(\text{where } k < x.key\)

**DECREASEKey** \((E, 2)\)

**Step 1:** Find element \(x\)

**Step 2:** Check that \(k \leq x.key\), otherwise raise an error

**Step 3:** Set \(x.key = k\)

**Step 4:** While \(x.key\) is smaller than its parent’s:  
   (stop if \(x\) becomes the root)  
   swap \(x\) with its parent
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**($x, k$) - decreases the value of $x.key$ to $k$

where $k < x.key$

**DECREASEKEY**($E, 2$)

It's a heap again!

**Step 1:** Find element $x$

**Step 2:** Check that $k \leq x.key$, otherwise raise an error

**Step 3:** Set $x.key = k$

**Step 4:** While $x.key$ is smaller than its parent’s: (stop if $x$ becomes the root) swap $x$ with its parent
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**($x, k$) - decreases the value of $x.key$ to $k$

where $k < x.key$

---

**Step 1:** Find element $x$

**Step 2:** Check that $k \leq x.key$, otherwise raise an error

**Step 3:** Set $x.key = k$

**Step 4:** While $x.key$ is smaller than its parent's: (stop if $x$ becomes the root) swap $x$ with its parent
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**\((x, k)\) - decreases the value of \(x.key\) to \(k\)

where \(k < x.key\)

---

**Step 1:** Find element \(x\) \(\rightarrow O(1)\) time

**Step 2:** Check that \(k \leq x.key\), otherwise raise an error

**Step 3:** Set \(x.key = k\)

**Step 4:** While \(x.key\) is smaller than its parent’s: (stop if \(x\) becomes the root)

swap \(x\) with its parent
DECREASEKEY with a Binary Heap

\textbf{DECREASEKEY}(x, k) - decreases the value of }x.\text{key} \text{ to } k \text{ where } k < x.\text{key}

\begin{itemize}
  \item \textbf{Step 1:} Find element }x\text{ \textit{O}(1) time
  \item \textbf{Step 2:} Check that }k \leq x.\text{key}, \text{otherwise raise an error
  \item \textbf{Step 3:} Set }x.\text{key} = k
  \item \textbf{Step 4:} While }x.\text{key} \text{ is smaller than its parent's: (stop if }x\text{ becomes the root)
    \begin{itemize}
      \item swap }x\text{ with its parent
    \end{itemize}
\end{itemize}
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**(\(x, k\)) - decreases the value of \(x\).key to \(k\)

where \(k < x\).key

---

**Step 1:** Find element \(x\) \(\rightarrow O(1) \text{ time}\)

**Step 2:** Check that \(k \leq x\).key, otherwise raise an error

**Step 3:** Set \(x\).key = \(k\) \(\rightarrow O(1) \text{ time}\)

**Step 4:** While \(x\).key is smaller than its parent’s: (stop if \(x\) becomes the root) swap \(x\) with its parent
**DECREASEKEY with a Binary Heap**

**DECREASEKEY** \((x, k)\) - decreases the value of \(x.key\) to \(k\)

\[\text{where } k < x.key\]

---

**Step 1:** Find element \(x\) \(\rightarrow O(1)\) time

**Step 2:** Check that \(k \leq x.key\), otherwise raise an error

**Step 3:** Set \(x.key = k\) \(\rightarrow O(1)\) time

**Step 4:** While \(x.key\) is smaller than its parent’s: (stop if \(x\) becomes the root) swap \(x\) with its parent

---

Each swap takes \(O(1)\) time
**DECREASEKEY with a Binary Heap**

**DECREASEKEY** \((x, k)\) - decreases the value of \(x.key\) to \(k\)

*where \(k < x.key\)*

---

**Step 1:** Find element \(x\)  \(\mathcal{O}(1)\) time

**Step 2:** Check that \(k \leq x.key\), otherwise raise an error

**Step 3:** Set \(x.key = k\)  \(\mathcal{O}(1)\) time

**Step 4:** While \(x.key\) is smaller than its parent’s: (stop if \(x\) becomes the root)

  - swap \(x\) with its parent

---

The height is \(\mathcal{O}(\log n)\)

---

Each swap takes \(\mathcal{O}(1)\) time
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**($x, k$) - decreases the value of $x.key$ to $k$

where $k < x.key$

Step 1: Find element $x$  
$O(1)$ time

Step 2: Check that $k \leq x.key$, otherwise raise an error

Step 3: Set $x.key = k$  
$O(1)$ time

Step 4: While $x.key$ is smaller than its parent’s:  
(stop if $x$ becomes the root)

swap $x$ with its parent

The height of the tree is $O(\log n)$ so there are $O(\log n)$ swaps
**DECREASEKEY with a Binary Heap**

**DECREASEKEY**\((x, k)\) - decreases the value of \(x\).key to \(k\) where \(k < x\).key

**Diagram:**
- The height is \(O(\log n)\)
- Element \(E\)

**Step 1:** Find element \(x\)

**Step 2:** Check that \(k \leq x\).key, otherwise raise an error

**Step 3:** Set \(x\).key = \(k\)

**Step 4:** While \(x\).key is smaller than its parent’s: *(stop if \(x\) becomes the root)*
- swap \(x\) with its parent
**DECREASEKEY** with a Binary Heap

**DECREASEKEY**\((x, k)\) - decreases the value of \(x.key\) to \(k\)

*where \(k < x.key\)*

---

**Step 1:** Find element \(x\)

**Step 2:** Check that \(k \leq x.key\), otherwise raise an error

**Step 3:** Set \(x.key = k\)

**Step 4:** While \(x.key\) is smaller than its parent's: *(stop if \(x\) becomes the root)*

- swap \(x\) with its parent

Overall this takes \(O(\log n)\) time

---

The height is \(O(\log n)\)
**INSERT with a Binary Heap**

\[ \text{INSERT}(x, k) \] - inserts \( x \) with \( x.\text{key} = k \)

**Step 1:** Put element \( x \) in the next free slot

**Step 2:** Run \text{DECREASE}K\text{EY}(x,k).
**INSERT with a Binary Heap**

**INSERT**\((x, k)\) - inserts \(x\) with \(x.key = k\)

- **Step 1:** Put element \(x\) in the next free slot
- **Step 2:** Run **DECREASE KEY**\((x,k)\).
**INSERT with a Binary Heap**

\[ \text{INSERT}(x, k) - \text{inserts } x \text{ with } x.\text{key} = k \]

\[ \text{INSERT}(F, 1) \]

**Step 1:** Put element \( x \) in the next free slot

**Step 2:** Run \( \text{DECREASEKEY}(x, k) \).
**INSERT with a Binary Heap**

\[ \text{INSERT}(x, k) \] - inserts \( x \) with \( x\.key = k \)

**Step 1:** Put element \( x \) in the next free slot

**Step 2:** Run \text{DECREASEKEY}(x, k)\).

![Binary Heap Diagram]
**INSERT with a Binary Heap**

\[ \text{INSERT}(x, k) \] - inserts \( x \) with \( x.\text{key} = k \)

**Step 1:** Put element \( x \) in the next free slot

**Step 2:** Run \( \text{DECREASE\text{KEY}}(x, k) \).
**INSERT with a Binary Heap**

**INSERT**(x, k) - inserts x with x.key = k

**INSERT**(F, 1)

**Step 1:** Put element x in the next free slot

**Step 2:** Run **DECREASEKEY**(x, k).
**INSERT with a Binary Heap**

**INSERT**($x$, $k$) - inserts $x$ with $x.key = k$

**INSERT**($F$, 1)

**Step 1:** Put element $x$ in the next free slot \(\text{O}(1)\) time

**Step 2:** Run **DECREASEKEY**($x$, $k$).
**INSERT with a Binary Heap**

\[ \text{INSERT}(x, k) - \text{inserts } x \text{ with } x.\text{key} = k \]

**Step 1:** Put element \( x \) in the next free slot \( \mathcal{O}(1) \) time

**Step 2:** Run \( \text{DECREASE\text{KEY}}(x, k) \). \( \mathcal{O}(\log n) \) time
**INSERT with a Binary Heap**

**INSERT**(\( x, k \)) - inserts \( x \) with \( x\.\text{key} = k \)

**Step 1:** Put element \( x \) in the next free slot \( O(1) \) time

**Step 2:** Run \( \text{DECREASE KEY}(x, k) \). \( O(\log n) \) time

Overall this takes \( O(\log n) \) time.
**INSERT with a Binary Heap**

$\text{INSERT}(x, k)$ - inserts $x$ with $x.key = k$

**Step 1:** Put element $x$ in the next free slot \( O(1) \) time

**Step 2:** Run $\text{DECREASEKEY}(x, k)$.

\( O(\log n) \) time

Overall this takes \( O(\log n) \) time
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN**() - removes and returns the element with the smallest key

- **Step 1:** Extract the element at the root
  *by definition, it is the minimum*

- **Step 2:** Move the rightmost element in the bottom level to the root
  *(call this element *y*)

- **Step 3:** While *y*.key is larger than one of its children’s: *(stop if *y* becomes a leaf)*
  swap *y* with the child with the *smaller key*
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

**Diagram:**

- **Step 1:** Extract the element at the root
  - *by definition, it is the minimum*

- **Step 2:** Move the rightmost element in the bottom level to the root
  - *(call this element \(y\))*

- **Step 3:** While \(y.key\) is larger than one of its children’s:
  - *(stop if \(y\) becomes a leaf)*
  - swap \(y\) with the child with the *smaller key*
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

---

**Step 1:** Extract the element at the root

*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root

*(call this element \(y\))*

**Step 3:** While \(y\).key is larger than one of its children’s:

*(stop if \(y\) becomes a leaf)*

swap \(y\) with the child with the smaller key
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

---

**Step 1:** Extract the element at the root

*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root

*(call this element $y$)*

**Step 3:** While $y.key$ is larger than one of its children’s:

*(stop if $y$ becomes a leaf)*

swap $y$ with the child with the *smaller key*
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

**Step 1:** Extract the element at the root
*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root
*(call this element \( y \))*

**Step 3:** While \( y.\text{key} \) is larger than one of its children’s:
*(stop if \( y \) becomes a leaf)*
swap \( y \) with the child with the smaller key
**ExtractMin with a Binary Heap**

**ExtractMin()** - removes and returns the element with the smallest key

---

**Step 1:** Extract the element at the root  
*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root  
*(call this element y)*

**Step 3:** While *y.key* is larger than one of its children’s:  
*(stop if y becomes a leaf)*  
swap *y* with the child with the *smaller key*
**EXTRACTMIN** with a Binary Heap

**EXTRACTMIN()** - removes and returns the element with the smallest key

---

**Step 1:** Extract the element at the root  
*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root  
*(call this element $y$)*

**Step 3:** While $y.key$ is larger than one of its children’s:  
*(stop if $y$ becomes a leaf)*  
swap $y$ with the child with the smaller key
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

**Step 1:** Extract the element at the root
   *by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root
   *(call this element y)*

**Step 3:** While \( y \text{.key} \) is larger than one of its children’s:
   *(stop if \( y \) becomes a leaf)*
   swap \( y \) with the child with the smaller key
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

**Step 1:** Extract the element at the root
   *by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root
   *(call this element y)*

**Step 3:** While *y.key* is larger than one of its children’s:
   *(stop if y becomes a leaf)*
   swap *y* with the child with the *smaller key*

\[O(1)\] time
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

**Step 1:** Extract the element at the root
*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root
*call this element \( y \)*

**Step 3:** While \( y . \text{key} \) is larger than one of its children's:
*stop if \( y \) becomes a leaf*
swap \( y \) with the child with the **smaller key**
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

Step 1: Extract the element at the root
      *by definition, it is the minimum*

Step 2: Move the rightmost element in the bottom level to the root
        *(call this element *y*)

Step 3: While *y.key* is larger than one of its children’s:
        *(stop if *y* becomes a leaf)*
        swap *y* with the child with the *smaller key*
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

---

**Step 1:** Extract the element at the root 
*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root 
*(call this element \(y\))*

**Step 3:** While \(y.key\) is larger than one of its children’s:  
*(stop if \(y\) becomes a leaf)*  
swap \(y\) with the child with the *smaller key*

*The height of the tree is \(O(\log n)\) so there are \(O(\log n)\) swaps (again)*
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

- **Step 1:** Extract the element at the root
  - *by definition, it is the minimum*

- **Step 2:** Move the rightmost element in the bottom level to the root
  - *(call this element $y$)*

- **Step 3:** While $y$.key is larger than one of its children’s:
  - *(stop if $y$ becomes a leaf)*
  - swap $y$ with the child with the *smaller key*
**EXTRACTMIN with a Binary Heap**

**EXTRACTMIN()** - removes and returns the element with the smallest key

---

**Step 1:** Extract the element at the root

*by definition, it is the minimum*

**Step 2:** Move the rightmost element in the bottom level to the root

*(call this element $y$)*

**Step 3:** While $y.key$ is larger than one of its children’s:

*stop if $y$ becomes a leaf*

swap $y$ with the child with the smaller key

---

Overall this takes $O(\log n)$ time
Priority queue Summary

We have seen three different priority queue implementations each supporting the following operations:

- **INSERT**\((x, k)\) - inserts \(x\) with \(x.key = k\)
- **DECREASEKEY**\((x, k)\) - decreases the value of \(x.key\) to \(k\)
  where \(k < x.key\)
- **EXTRACTMIN()** - removes and returns the element with the smallest key

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>DECREASEKEY</th>
<th>EXTRACTMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>
Priority queue Summary

We have seen three different priority queue implementations each supporting the following operations:

\[ \text{INSERT}(x, k) \] - inserts \( x \) with \( x.\text{key} = k \)

\[ \text{DECREASEKEY}(x, k) \] - decreases the value of \( x.\text{key} \) to \( k \) where \( k < x.\text{key} \)

\[ \text{EXTRACTMIN}() \] - removes and returns the element with the smallest key

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>DECREASEKEY</th>
<th>EXTRACTMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>

Is this the best possible?
Priority queue Summary

We have seen three different priority queue implementations each supporting the following operations:

\[ \text{INSERT}(x, k) - \text{inserts } x \text{ with } x.\text{key} = k \]

\[ \text{DECREASEKEY}(x, k) - \text{decreases the value of } x.\text{key} \text{ to } k \]

\[ \text{where } k < x.\text{key} \]

\[ \text{EXTRACTMIN}() - \text{removes and returns the element with the smallest key} \]

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>DECREASEKEY</th>
<th>EXTRACTMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

*Is this the best possible? actually, no :)*
Priority queue Summary

We have seen three different priority queue implementations each supporting the following operations:

\[ \text{INSERT}(x, k) \] - inserts \( x \) with \( x.key = k \)

\[ \text{DECREASEKEY}(x, k) \] - decreases the value of \( x.key \) to \( k \)

\[ \text{EXTRACTMIN()} \] - removes and returns the element with the smallest key

\[
\begin{array}{|c|c|c|c|}
\hline
 & \text{INSERT} & \text{DECREASEKEY} & \text{EXTRACTMIN} \\
\hline
\text{Unsorted Linked List} & O(1) & O(n) & O(n) \\
\hline
\text{Sorted Linked List} & O(n) & O(n) & O(1) \\
\hline
\text{Binary Heap} & O(\log n) & O(\log n) & O(\log n) \\
\hline
\text{Fibonacci Heap} & O(1) & O(1) & O(\log n) \\
\hline
\end{array}
\]

Is this the best possible? actually, no :)}
Priority queue Summary

We have seen three different priority queue implementations each supporting the following operations:

- **INSERT**(\(x, k\)) - inserts \(x\) with \(x\).key = \(k\)
- **DECREASE KEY**(\(x, k\)) - decreases the value of \(x\).key to \(k\) where \(k < x\).key
- **EXTRACT MIN**() - removes and returns the element with the smallest key

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>DECREASE KEY</th>
<th>EXTRACT MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unssorted Linked List</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

Is this the best possible? actually, no :)

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>DECREASE KEY</th>
<th>EXTRACT MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci Heap</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

...but Fibonacci Heaps are complicated, amortised and have large hidden constants
One more thing. . .

Take an array of elements of length $n$

\[ A \]

**INSERT** every element into a priority queue:

**EXTRACT MIN** from the priority queue $n$ times

*and put the elements in $A'$ in the order they come out*

\[ A' \]

*what is $A'$?*
One more thing... 

Take an array of elements of length $n$ 

\[ A \]

**INSERT** every element into a priority queue: 

**EXTRACTMin** from the priority queue $n$ times 

*and put the elements in $A'$ in the order they come out* 

\[ A' \]

What is $A'$? It's A in sorted order
One more thing... 

Take an array of elements of length $n$

Insert every element into a priority queue:

If you implement the priority queue as a Binary Heap

You can use this to sort in $O(n \log n)$ time

ExtractMin from the priority queue $n$ times 
and put the elements in $A'$ in the order they come out

What is $A'$? it's $A$ in sorted order
HeapSort

Take an array of elements of length $n$

$A$

$A'$

**INSERT** every element into a priority queue:

If you implement the priority queue as a *Binary Heap*

*You can use this to sort in $O(n \log n)$ time*

**EXTRACTMIN** from the priority queue $n$ times

*and put the elements in $A'$ in the order they come out*

$A' \ ?$ *it's A in sorted order*
End of part one
Part two

Dijkstra’s Algorithm
In today’s lecture we’ll be discussing the **single source shortest paths** problem in a weighted, directed graph.

The shortest path from MVB to Temple Meads (according to Google Maps)

In particular we’ll be interested in **Dijkstra’s Algorithm** which is based on an **abstract data-structure** called a **priority queue**

... which can be efficiently implemented as a **binary heap**
In today’s lecture we’ll be discussing the **single source shortest paths** problem in a weighted, directed graph.

In particular we’ll be interested in **Dijkstra’s Algorithm**

which is based on an **abstract data-structure** called a **priority queue**

... which can be efficiently implemented as a **binary heap**

The shortest path from MVB to Temple Meads
(according to Google Maps)

Vertices are junctions
Edges are roads
Edge weights are in miles
Directed edges are one-way streets
Dijkstra’s Algorithm solves the **single source shortest paths** problem in a **weighted**, directed graph...
Dijkstra’s Algorithm solves the **single source shortest paths** problem in a **weighted**, directed graph. . .

It finds the shortest path from a given *source* vertex to *every* other vertex.
Dijkstra’s Algorithm solves the **single source shortest paths** problem in a **weighted**, directed graph. . .

It finds the shortest path from a given *source* vertex to every other vertex

*The weights have to be non-negative*
Dijkstra’s Algorithm solves the **single source shortest paths** problem in a **weighted**, directed graph. . .

It finds the shortest path from a given *source* vertex to *every* other vertex

*The weights have to be non-negative*

The graph is stored as an **Adjacency List**
Dijkstra’s Algorithm solves the **single source shortest paths** problem in a **weighted**, directed graph. It finds the shortest path from a given **source** vertex to every other vertex. The weights have to be non-negative. The graph is stored as an **Adjacency List**. The time complexity will depend on how efficient the priority queue used is.
Dijkstra’s Algorithm solves the **single source shortest paths** problem in a **weighted**, directed graph...

It finds the shortest path from a given *source* vertex to *every* other vertex

*The weights have to be non-negative*

The graph is stored as an **Adjacency List**

The time complexity will depend on how efficient the priority queue used is

Remember from last lecture that in **unweighted**, directed graphs, Breadth First Search solves this problem in $O(|V| + |E|)$ time

$|V|$ is the number of vertices and $|E|$ is the number of edges
Dijkstra’s algorithm

We assume that we have a priority queue, supporting
\texttt{INSERT}, \texttt{DECREASEKEY} and \texttt{EXTRACTMIN}

\begin{center}
\begin{tabular}{|c|}
\hline
\texttt{Dijkstra}(s) \\
\hline
\end{tabular}
\end{center}

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \texttt{INSERT}(\( v \), \text{dist}(v))
while the queue is not empty
\begin{align*}
\text{dist}(v) &= \text{dist}(u) + \text{weight}(u,v) \\
&\quad \text{if dist}(v) > \text{dist}(u) + \text{weight}(u,v)
\end{align*}
\texttt{DECREASEKEY}(\( v \), \text{dist}(v))

\( (u, v) \in E \) iff there is an edge from \( u \) to \( v \)

\( \text{weight}(u, v) \) is the weight of the edge from \( u \) to \( v \)

\( \text{dist}(v) \) is the length of the best path between \( s \) and \( v \), found so far

\textbf{Claim} when Dijkstra’s algorithm terminates,
for each vertex \( v \), \( \text{dist}(v) \) is the distance between \( s \) and \( v \)
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
    $u = \text{EXTRACTMIN}()$
    for every edge $(u, v) \in E$
        if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
            $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
            $\text{DECREASEKEY}(v, \text{dist}(v))$

$\text{DIJKSTRA}(s)$

We’re going to simulate $\text{DIJKSTRA}(A)$
i.e. $s = A$

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
$v.key = \text{dist}(v)$
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
\hspace{1em} $u = \text{EXTRACTMIN}()$
\hspace{1em} for every edge $(u, v) \in E$
\hspace{2em} if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
\hspace{3em} $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
\hspace{2em} $\text{DECREASEKEY}(v, \text{dist}(v))$

$\text{DIJKSTRA}(s)$

We're going to simulate $\text{DIJKSTRA}(A)$
i.e. $s = A$

$\text{dist}$:

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

$\text{dist}(v)$ is the length of the shortest
path between $s$ and $v$, found so far

at all times, for each vertex $v$,
\begin{align*}
    v.\text{key} &= \text{dist}(v)
\end{align*}
We're going to simulate \texttt{Dijkstra(A)} i.e. \( S = A \)

\begin{itemize}
  \item \texttt{Dijkstra(s)}
  \item For all \( v \), set \( \text{dist}(v) = \infty \)
  \item Set \( \text{dist}(s) = 0 \)
  \item For each \( v \), do \texttt{INSERT}(v,dist(v))
  \item While the queue is not empty
    \begin{itemize}
      \item \( u = \texttt{EXTRACTMIN}() \)
      \item For every edge \((u,v) \in E\)
        \begin{itemize}
          \item If \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
            \begin{itemize}
              \item \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \)
              \item \texttt{DECREASEKEY}(v,\text{dist}(v))
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

\textbf{dist:}
\begin{tabular}{cccccccc}
  A & B & C & D & E & F & G \\
  0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{tabular}

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

\textit{at all times}, for each vertex \( v \),

\( v.\text{key} = \text{dist}(v) \)
We’re going to simulate
\textsc{Dijkstra}(A)
\[\text{i.e. } S = A\]

\textbf{Dijkstra}(s)

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \textsc{Insert}(v,\text{dist}(v))
while the queue is not empty
\[ u = \text{ExtractMin}() \]
for every edge \( (u,v) \in E \)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \]
\textsc{DecreaseKey}(v,\text{dist}(v))

new path to \( B = 0 + 1 = 1 \)

\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & \infty & \infty & \infty & \infty & \infty & \infty \\
\end{array}

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

\textit{at all times}, for each vertex \( v \),
\[ v.\text{key} = \text{dist}(v) \]
We're going to simulate Dijkstra(A)

\[ s = A \]

\[ \text{Dijkstra(s)} \]

For all \( v \), set \( \text{dist}(v) = \infty \)

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

For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)

while the queue is not empty

\[ u = \text{EXTRACTMIN}() \]

for every edge \( (u,v) \in E \)

if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)

\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \]

\[ \text{DECREASEKEY}(v, \text{dist}(v)) \]

new path to \( B = 0 + 1 = 1 \)

\[ \text{dist:} \begin{array}{cccccccc}
A & B & C & D & E & F & G \\
0 & 1 & \infty & \infty & \infty & \infty & \infty & \infty
\end{array} \]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),

\[ v.key = \text{dist}(v) \]
Dijkstra($s$)

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
  $u = \text{EXTRACTMIN}()$
  for every edge $(u, v) \in E$
    if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
      $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
      $\text{DECREASEKEY}(v, \text{dist}(v))$

We're going to simulate Dijkstra(A)
i.e. $s = A$

this is called relaxing edge $(u, v)$

new path to B = $0 + 1 = 1$

$\text{A B C D E F G}$
$\text{dist: 0 1 } \infty \infty \infty \infty \infty \infty$

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
$v.key = \text{dist}(v)$
For all $v$, set $\text{dist}(v) = \infty$ 
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty 
\hspace{1em} $u = \text{EXTRACTMIN}()$
\hspace{1em} for every edge $(u, v) \in E$
\hspace{2em} if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
\hspace{3em} $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
\hspace{3em} $\text{DECREASEKEY}(v, \text{dist}(v))$

$\text{DIJKSTRA}(s)$

We're going to simulate $\text{DIJKSTRA}(A)$
\hspace{1em} i.e. $s = A$

this is called relaxing edge $(u, v)$

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$, $v.key = \text{dist}(v)$
**Dijkstra**

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
\( u = \text{EXTRACTMIN}() \)
for every edge \((u,v) \in E\)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
\( \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \)
\( \text{DECREASEKEY}(v, \text{dist}(v)) \)

---

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
\( u = \text{EXTRACTMIN}() \)
for every edge \((u,v) \in E\)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
\( \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \)
\( \text{DECREASEKEY}(v, \text{dist}(v)) \)

---

**Dijkstra\((s)\)**

We're going to simulate **Dijkstra\((A)\)**
i.e. \( s = A \)

this is called **relaxing edge** \((u, v)\)

---

\[ \begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\hline
0 & 1 & \infty & \infty & \infty & \infty & \infty \\
\end{array} \]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\( v.\text{key} = \text{dist}(v) \)
We’re going to simulate Dijkstra\( (A) \) i.e. \( s = A \)

\[ \text{dist}(v) \text{ is the length of the shortest path between } s \text{ and } v, \text{ found so far} \]

\[ \forall v, \text{set } \text{dist}(v) = \infty \]

\[ \text{set } \text{dist}(s) = 0 \]

For each \( v \), do INSERT\( (v, \text{dist}(v)) \)

while the queue is not empty

\[ u = \text{ExtractMin}() \]

for every edge \( (u, v) \in E \)

if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)

\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]

DECREASE\( \text{KEY}(v, \text{dist}(v)) \)

\[ V = \{ A, B, C, D, E, F, G \} \]

\[ \begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\hline
0 & 1 & \infty & \infty & \infty & \infty & \infty & \infty
\end{array} \]
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
    $u = \text{ExtractMin}()$
    for every edge $(u, v) \in E$
        if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
            $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
            $\text{DECREASEKEY}(v, \text{dist}(v))$

$\text{Dijkstra}(s)$

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
$v.\text{key} = \text{dist}(v)$

We’re going to simulate $\text{Dijkstra}(A)$
i.e. $s = A$

this is called relaxing edge $(u, v)$

settled vertices
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
while the queue is not empty
  \( u = \text{EXTRACTMIN}() \)
  for every edge \( (u, v) \in E \)
    if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
    \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)
    \( \text{DECREASEKEY}(v, \text{dist}(v)) \)

new path to \( C = 1 + 2 = 3 \)

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & \infty & \infty & \infty & \infty & \infty
\end{array}
\]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\( v.\text{key} = \text{dist}(v) \)
We’re going to simulate $\text{DIJKSTRA}(A)$ i.e. $s = A$

dist$(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$, $v\.key = \text{dist}(v)$
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
\begin{itemize}
  \item $u = \text{EXTRACTMIN}()$
  \item for every edge $(u, v) \in E$
    \begin{itemize}
      \item if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
      \item $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
      \item $\text{DECREASEKEY}(v, \text{dist}(v))$
    \end{itemize}
\end{itemize}

new path to $D = 1 + 4 = 5$

$\begin{array}{ccccccccc}
A & B & C & D & E & F & G \\
\hline
0 & 1 & 3 & \infty & \infty & \infty & \infty & \infty
\end{array}$

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,$$
v.\text{key} = \text{dist}(v)$$
For all $v$, set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each $v$, do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
\[ u = \text{EXTRACTMIN}() \]
for every edge $(u, v) \in E$
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]
\text{DECREASEKEY}(v, \text{dist}(v))

We’re going to simulate \text{Dijkstra}(A)
i.e. $s = A$

\[ \text{dist}(v) \] is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
\[ v.\text{key} = \text{dist}(v) \]
We're going to simulate \text{Dijkstra}(A) i.e. \( S = A \)

\begin{itemize}
\item For all \( v \), set \( \text{dist}(v) = \infty \)
\item set \( \text{dist}(s) = 0 \)
\item For each \( v \), do \text{INSERT}(v, \text{dist}(v))
\item while the queue is not empty
  \begin{itemize}
  \item \( u = \text{EXTRACTMIN}() \)
  \item for every edge \((u,v) \in E\)
    \begin{itemize}
    \item if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
      \begin{itemize}
      \item \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \)
      \item \text{DECREASEKEY}(v, \text{dist}(v))
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
\end{itemize}

\text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far \( \text{at all times} \), for each vertex \( v \), \( v.\text{key} = \text{dist}(v) \)
**Dijkstra\((s)\)**

For all \(v\), set \(\text{dist}(v) = \infty\)

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

For each \(v\), do \text{INSERT}(v, \text{dist}(v))

while the queue is not empty

\(u = \text{EXTRACTMIN()}\)

for every edge \((u,v) \in E\)

if \(\text{dist}(v) > \text{dist}(u) + \text{weight}(u,v)\)

\(\text{dist}(v) = \text{dist}(u) + \text{weight}(u,v)\)

\text{DECREASEKEY}(v, \text{dist}(v))

---

We're going to simulate **Dijkstra**(\(A\))

i.e. \(s = A\)

this is called **relaxing edge** \((u, v)\)

---

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & 3 & 5 & \infty & \infty & \infty
\end{array}
\]

\(\text{dist}(v)\) is the length of the shortest path between \(s\) and \(v\), found so far

\textit{at all times}, for each vertex \(v\),

\(v.\text{key} = \text{dist}(v)\)
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
while the queue is not empty
\[ u = \text{EXTRACTMIN}() \]
for every edge \( (u, v) \in E \)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]
\[ \text{DECREASEKEY}(v, \text{dist}(v)) \]

We're going to simulate \( \text{DIJKSTRA}(A) \)
i.e. \( s = A \)
this is called relaxing edge \((u, v)\)

\[ \text{dist}(v) \] is the length of the shortest path between \( s \) and \( v \), found so far
\[ \text{at all times}, \text{for each vertex } v, \]
\[ v.\text{key} = \text{dist}(v) \]
**Dijkstra(s)**

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
while the queue is not empty
  \( u = \text{EXTRACTMIN()} \)
for every edge \( (u, v) \in E \)
  if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
    \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)
    \( \text{DECREASEKEY}(v, \text{dist}(v)) \)

We're going to simulate \( \text{Dijkstra}(A) \)
i.e. \( s = A \)

this is called relaxing edge \((u, v)\)

new path to \( D = 3 + 1 = 4 \)

\[
\text{dist: } \begin{array}{cccccccc}
A & B & C & D & E & F & G \\
0 & 1 & 3 & 5 & \infty & \infty & \infty \\
\end{array}
\]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\( v.\text{key} = \text{dist}(v) \)
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
\( u = \text{EXTRACTMIN()} \)
for every edge \( (u,v) \in E \)
\( \text{if } \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
\( \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \)
\( \text{DECREASEKEY}(v, \text{dist}(v)) \)

\text{Dijkstra}(s)

We’re going to simulate \text{Dijkstra}(A)
\text{i.e. } s = A

this is called relaxing edge \((u,v)\)

new path to \( D = 3 + 1 = 4 \)

\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\hline
0 & 1 & 3 & 4 & \infty & \infty & \infty
\end{array}

\text{dist}(v) \text{ is the length of the shortest path between } s \text{ and } v, \text{ found so far}

\text{at all times}, \text{ for each vertex } v, \ \ v.key = \text{dist}(v)
We're going to simulate \( \text{DIJKSTRA}(A) \)

\[ \text{i.e. } s = A \]

\[ \text{this is called relaxing edge } (u, v) \]

\[ \text{new path to } G = 3 + 5 = 8 \]

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\hline
0 & 1 & 3 & 4 & \infty & \infty & \infty \\
\end{array}
\]

\[ \text{dist}(v) \text{ is the length of the shortest path between } s \text{ and } v, \text{ found so far} \]

\[ \text{at all times, for each vertex } v, \]

\[ v.\text{key} = \text{dist}(v) \]
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do \text{INSERT}$(v, \text{dist}(v))$
while the queue is not empty
  $u = \text{EXTRACTMIN}()$
  for every edge $(u,v) \in E$
    if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u,v)$
      $\text{dist}(v) = \text{dist}(u) + \text{weight}(u,v)$
      \text{DECREASEKEY}(v, \text{dist}(v))$

new path to $G = 3 + 5 = 8$

$\text{dist:}$
\begin{tabular}{cccccccc}
  A & B & C & D & E & F & G \\
  0 & 1 & 3 & 4 & $\infty$ & $\infty$ & 8
\end{tabular}

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
$v.\text{key} = \text{dist}(v)$

We’re going to simulate $\text{DIJKSTRA}(A)$
i.e. $s = A$
this is called relaxing edge $(u, v)$
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
\[ u = \text{EXTRACTMIN}() \]
for every edge \((u, v) \in E\)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]
\[ \text{DECREASEKEY}(v, \text{dist}(v)) \]

\[ \text{DIJKSTRA}(s) \]

We’re going to simulate \text{DIJKSTRA}(A)\]
i.e. \( S = A \)
this is called \text{relaxing edge} \((u, v)\)

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & 3 & 4 & \infty & \infty & 8 \\
\end{array}
\]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\[ v.key = \text{dist}(v) \]
**Dijkstra(s)**

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
\[ u = \text{EXTRACTMIN}(\) \]
for every edge $(u,v) \in E$
\[ \text{if } \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \]
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \]
\[ \text{DECREASEKEY}(v, \text{dist}(v)) \]

We're going to simulate $\text{Dijkstra(A)}$
i.e. $s = A$

This is called relaxing edge $(u, v)$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>8</td>
</tr>
</tbody>
</table>

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
\[ v.\text{key} = \text{dist}(v) \]
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
while the queue is not empty
\begin{align*}
  u & = \text{EXTRACTMIN}() \\
  \text{for every edge } (u, v) & \in E \\
  \text{if } \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \\
  \text{dist}(v) & = \text{dist}(u) + \text{weight}(u, v) \\
  \text{DECREASEKEY}(v, \text{dist}(v))
\end{align*}
\( \text{DIJKSTRA}(s) \)

We’re going to simulate \( \text{DIJKSTRA}(A) \)
i.e. \( s = A \)
this is called relaxing edge \((u, v)\)

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & 3 & 4 & \infty & \infty & 8
\end{array}
\]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\( v.\text{key} = \text{dist}(v) \)
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
$u = \text{EXTRACTMIN}()$
for every edge $(u, v) \in E$
if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
$\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
$\text{DECREASEKEY}(v, \text{dist}(v))$

Dijkstra($s$)

We're going to simulate Dijkstra(A)
i.e. $s = A$
this is called relaxing edge $(u, v)$

new path to $A = 0 + 2 = 2$

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & 3 & 4 & \infty & \infty & 8 \\
\end{array}
\]

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
$v.key = \text{dist}(v)$
We’re going to simulate \texttt{Dijkstra}(A) i.e. $s = A$

We start by setting all distances to infinity, except for the source $s$.

For each vertex $v$, we insert it into the priority queue with its initial distance, $\text{dist}(v) = \infty$.

We then repeatedly extract the vertex with the smallest distance from the priority queue and relax all outgoing edges from that vertex.

This process continues until the queue is empty.

We can see from the diagram that the new path to $E$ is $4 + 2 = 6$.

The distance table is shown at each step:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>dist:</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>\infty</td>
<td>\infty</td>
<td>8</td>
</tr>
</tbody>
</table>

The distance function, $\text{dist}(v)$, is the length of the shortest path between the source $s$ and vertex $v$, found so far.

At all times, for each vertex $v$, \texttt{v.key} = $\text{dist}(v)$
We're going to simulate \textsc{Dijkstra}(A) i.e. $s = A$

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
$u = \text{ExtractMin}()$
for every edge $(u, v) \in E$
if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
$\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
$\text{DECREASEKEY}(v, \text{dist}(v))$

new path to $E = 4 + 2 = 6$

\begin{tabular}{ccccccccc}
A & B & C & D & E & F & G \\
0 & 1 & 3 & 4 & 6 & $\infty$ & 8
\end{tabular}

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
$v.key = \text{dist}(v)$
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
while the queue is not empty
\( u = \text{EXTRACTMIN}() \)
for every edge \( (u, v) \in E \)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\( \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)
\( \text{DECREASEKEY}(v, \text{dist}(v)) \)

\[ \begin{array}{cccccccc}
A & B & C & D & E & F & G \\
0 & 1 & 3 & 4 & 6 & \infty & \infty
\end{array} \]

new path to \( F = 4 + 2 = 6 \)

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\( v.\text{key} = \text{dist}(v) \)
For all \( v \), set \( \text{dist}(v) = \infty \)

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

For each \( v \), do \text{INSERT}(v, \text{dist}(v))

while the queue is not empty

\( u = \text{EXTRACTMIN}() \)

for every edge \((u, v) \in E\)

if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)

\( \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)

\text{DECREASEKEY}(v, \text{dist}(v))

We’re going to simulate \( \text{DIJKSTRA}(A) \)

i.e. \( s = A \)

this is called \textit{relaxing edge} \((u, v)\)

new path to \( F = 4 + 2 = 6 \)

\[ \begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & 3 & 4 & 6 & 6 & 8 \\
\end{array} \]

dist\((v)\) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),

\( v.\text{key} = \text{dist}(v) \)
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
    \( u = \text{EXTRACTMIN}() \)
for every edge \( (u, v) \in E \)
    if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
        \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)
        \text{DECREASEKEY}(v, \text{dist}(v))

\text{DIJKSTRA}(s)

We're going to simulate \text{DIJKSTRA}(A) \ i.e. \( s = A \)

This is called \text{relaxing} \text{ edge} \ (u, v) \)

\[ \begin{array}{cccccccc}
\text{dist:} & A & B & C & D & E & F & G \\
\hline
0 & 1 & 3 & 4 & 6 & 6 & 6 & 8 \\
\end{array} \]

\( \text{dist}(v) \) \ is the length of the shortest path between \( s \) and \( v \), \text{found so far} \)

\textit{At all times, for each vertex} \( v \), \textit{v.key = dist}(v) \)
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
  $u = \text{EXTRACTMIN}()$
  for every edge $(u, v) \in E$
    if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
      $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
      $\text{DECREASEKEY}(v, \text{dist}(v))$

We're going to simulate $\text{DIJKSTRA}(A)$
i.e. $s = A$
this is called relaxing edge $(u, v)$

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far
at all times, for each vertex $v$, $v.\text{key} = \text{dist}(v)$
**Dijkstra**

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
\[ u = \text{EXTRACTMIN}() \]
for every edge \((u, v) \in E\)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]
\text{DECREASEKEY}(v, \text{dist}(v))

---

**We’re going to simulate**

\[ \text{Dijkstra}(A) \]
\[ \text{i.e. } s = A \]

This is called **relaxing edge** \((u, v)\)

---

**dist**: 0 1 3 4 6 6 6 8

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\[ v.\text{key} = \text{dist}(v) \]
**Dijkstra(s)**

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
while the queue is not empty
\( u = \text{EXTRACTMIN}() \)
for every edge \((u, v) \in E\)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\( \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)
\( \text{DECREASEKEY}(v, \text{dist}(v)) \)

We’re going to simulate **Dijkstra(A)**
i.e. \( s = A \)

this is called **relaxing edge** \((u, v)\)

\[
\begin{align*}
\text{new path to } B &= 6 + 1 = 7 \\
A & B C D E F G \\
\text{dist:} & \begin{array}{ccccccc}
0 & 1 & 3 & 4 & 6 & 6 & 8 \\
\end{array}
\end{align*}
\]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), **found so far**

\textit{at all times}, for each vertex \( v \),
\( v.key = \text{dist}(v) \)
We’re going to simulate \texttt{Dijkstra}(A)
i.e. \( s = A \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

\( \text{at all times}, \) for each vertex \( v \),
\( v.\text{key} = \text{dist}(v) \)
**Dijkstra**($s$)

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$

For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty

$u = \text{EXTRACTMIN}()$

for every edge $(u, v) \in E$

if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$

$\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$

$\text{DECREASEKEY}(v, \text{dist}(v))$

---

We’re going to simulate $\text{Dijkstra}(A)$
i.e. $s = A$

this is called relaxing edge $(u, v)$

---

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,

$v.key = \text{dist}(v)$
We’re going to simulate `DIJKSTRA(A)` i.e. \( S = A \)

This is called relaxing edge \((u, v)\)

\(\text{DIJKSTRA}(s)\)

For all \(v\), set \(\text{dist}(v) = \infty\)
set \(\text{dist}(s) = 0\)
For each \(v\), do \(\text{INSERT}(v, \text{dist}(v))\)
while the queue is not empty
\[ u = \text{EXTRACTMIN}() \]
for every edge \((u, v) \in E\)
if \(\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)\)
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]
\[\text{DECREASEKEY}(v, \text{dist}(v))\]

\(\text{dist}(v)\) is the length of the shortest path between \(s\) and \(v\), found so far

\(at\ all\ times\), for each vertex \(v\),
\[ v.\text{key} = \text{dist}(v) \]
**Dijkstra(s)**

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
  $u = \text{EXTRACTMIN}()$
  for every edge $(u, v) \in E$
    if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
      $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
      $\text{DECREASEKEY}(v, \text{dist}(v))$

We’re going to simulate $\text{Dijkstra}(A)$
i.e. $s = A$
this is called **relaxing edge** $(u, v)$

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & 3 & 4 & 6 & 6 & 8 \\
\end{array}
\]

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$,
$v.key = \text{dist}(v)$
For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
  $u = \text{EXTRACTMIN}()$
  for every edge $(u,v) \in \mathcal{E}$
    if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u,v)$
      $\text{dist}(v) = \text{dist}(u) + \text{weight}(u,v)$
      $\text{DECREASEKEY}(v, \text{dist}(v))$

new path to $G = 6 + 1 = 7$

\begin{tabular}{cccccccc}
A & B & C & D & E & F & G \\
\hline 
0 & 1 & 3 & 4 & 6 & 6 & 6 & 8 \\
\end{tabular}

$\text{dist}(v)$ is the length of the shortest path between $s$ and $v$, found so far

at all times, for each vertex $v$, $v\cdot\text{key} = \text{dist}(v)$
**DIJKSTRA(\(s\))**

For all \(v\), set \(\text{dist}(v) = \infty\)

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

For each \(v\), do \text{INSERT}(v, \text{dist}(v))

while the queue is not empty

\(u = \text{EXTRACTMIN}()\)

for every edge \((u,v) \in E\)

if \(\text{dist}(v) > \text{dist}(u) + \text{weight}(u,v)\)

\(\text{dist}(v) = \text{dist}(u) + \text{weight}(u,v)\)

\(\text{DECREASEKEY}(v, \text{dist}(v))\)

---

We’re going to simulate **DIJKSTRA(A)**

e.i. \(s = A\)

this is called **relaxing edge** \((u, v)\)

---

new path to \(G = 6 + 1 = 7\)

\[\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
\text{dist:} & 0 & 1 & 3 & 4 & 6 & 6 & 7 \\
\end{array}\]

\(\text{dist}(v)\) is the length of the shortest path between \(s\) and \(v\), found so far

*at all times*, for each vertex \(v\),

\(v\).key = \(\text{dist}(v)\)
**Dijkstra\((s)\)**

For all \(v\), set \(\text{dist}(v) = \infty\)

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

For each \(v\), do \text{INSERT}(v, \text{dist}(v))

while the queue is not empty

\[ u = \text{EXTRACTMIN()} \]

for every edge \((u,v) \in E\)

if \(\text{dist}(v) > \text{dist}(u) + \text{weight}(u,v)\)

\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \]

\text{DECREASEKEY}(v, \text{dist}(v))

---

**Dijkstra\((A)\)**

i.e. \(s = A\)

We're going to simulate

this is called relaxing edge \((u, v)\)

---

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
0 & 1 & 3 & 4 & 6 & 6 & 6 & 7
\end{array}
\]

\text{dist}(v) \text{ is the length of the shortest path between } s \text{ and } v, \text{ found so far}

\text{at all times}, for each vertex \(v\),

\(v\.\text{key} = \text{dist}(v)\)
We’re going to simulate \textsc{Dijkstra}(A) i.e. $s = A$

\begin{algorithm}
\DontPrintSemicolon
\textbf{Dijkstra}(s) \\
\ForAll{$v$, set $\text{dist}(v) = \infty$} \\
\text{set } \text{dist}(s) = 0 \\
\ForEach{$v$}, do \textsc{Insert}(v, \text{dist}(v)) \\
\While{the queue is not empty}
\quad $u = \text{ExtractMin}()$ \\
\quad \ForEvery{edge $(u,v) \in E$}
\quad \quad \text{if } \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v)$
\quad \quad \quad $\text{dist}(v) = \text{dist}(u) + \text{weight}(u,v)$ \\
\quad \quad \textsc{DecreaseKey}(v, \text{dist}(v))
\end{algorithm}

\begin{itemize}
\item \text{dist}$(v)$ is the length of the shortest path between $s$ and $v$, found so far \\\n\text{at all times, for each vertex } v, \quad v.key = \text{dist}(v)$
\end{itemize}

\begin{table}
\begin{tabular}{cccccccc}
A & B & C & D & E & F & G \\
\hline
0 & 1 & 3 & 4 & 6 & 6 & 6 & 7
\end{tabular}
\end{table}
For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
while the queue is not empty
\[ u = \text{EXTRACTMIN}() \]
for every edge \( (u, v) \in E \)
if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]
\[ \text{DECREASEKEY}(v, \text{dist}(v)) \]

We’re going to simulate \( \text{DIJKSTRA}(s) \)
i.e. \( s = A \)

shortest paths from \( s = A \):

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
0 & 1 & 3 & 4 & 6 & 6 & 6 & 7 \\
\end{array}
\]

\( \text{dist}(v) \) is the length of the shortest path between \( s \) and \( v \), found so far

at all times, for each vertex \( v \),
\[ v.\text{key} = \text{dist}(v) \]
Proof of Correctness

Claim when Dijkstra’s algorithm terminates, for each vertex $v$, $\text{dist}(v) = \delta(s, v)$

where $\delta(s, v)$ is the true distance between $s$ and $v$

\[
\text{Dijkstra}(s)
\]

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
\[
\begin{align*}
& u = \text{EXTRACTMIN}() \\
& \text{for every edge } (u, v) \in E \\
& \quad \text{if } \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \\
& \quad \quad \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \\
& \quad \quad \text{DECREASEKEY}(v, \text{dist}(v))
\end{align*}
\]
Proof of Correctness

**Claim** when Dijkstra’s algorithm terminates, for each vertex \( v \), \( \text{dist}(v) = \delta(s,v) \)

where \( \delta(s,v) \) is the true distance between \( s \) and \( v \)

\[
\text{Dijkstra}(s)
\]

- For all \( v \), set \( \text{dist}(v) = \infty \)
- set \( \text{dist}(s) = 0 \)
- For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)
- while the queue is not empty
  - \( u = \text{EXTRACTMIN}() \)
  - for every edge \( (u,v) \in E \)
    - if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
      - \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \)
      - \( \text{DECREASEKEY}(v, \text{dist}(v)) \)

**Observation** At all times, \( \text{dist}(v) \) is the length of some path from \( s \) to \( v \)

(unless \( \text{dist}(v) = \infty \))

Therefore, for each vertex \( v \), \( \delta(s,v) \leq \text{dist}(v) \)
Proof of Correctness

Claim when Dijkstra’s algorithm terminates, for each vertex \( v \), \( \text{dist}(v) = \delta(s, v) \)
where \( \delta(s, v) \) is the true distance between \( s \) and \( v \)

\[
\text{DIJKSTRA}(s)
\]

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \text{INSERT}(v, \text{dist}(v))
while the queue is not empty
\[
\begin{align*}
\text{u} &= \text{EXTRACTMIN}() \\
\text{for every edge } (u, v) \in E \\
\text{if } \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \text{ then} \\
\text{dist}(v) &= \text{dist}(u) + \text{weight}(u, v) \\
\text{DECREASEKEY}(v, \text{dist}(v))
\end{align*}
\]
Proof of Correctness

**Claim** when Dijkstra’s algorithm terminates, for each vertex \( v \), \( \text{dist}(v) = \delta(s, v) \)

where \( \delta(s, v) \) is the true distance between \( s \) and \( v \)

---

**DIJKSTRA**\((s)\)

For all \( v \), set \( \text{dist}(v) = \infty \)

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

For each \( v \), do \( \text{INSERT}(v, \text{dist}(v)) \)

while the queue is not empty

\[
u = \text{EXTRACTMIN}()
\]

for every edge \((u, v) \in E\)

if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)

\[
\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)
\]

\( \text{DECREASEKEY}(v, \text{dist}(v)) \)

---

Further, observe that after a vertex \( v \) is \( \text{EXTRACTED} \),

\( \text{dist}(v) \) certainly doesn’t increase
Proof of Correctness

Claim when Dijkstra’s algorithm terminates, for each vertex $v$, $\text{dist}(v) = \delta(s, v)$
where $\delta(s, v)$ is the true distance between $s$ and $v$

```
Dijkstra(s)
```

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
while the queue is not empty
    $u = \text{EXTRACTMIN}()$
    for every edge $(u, v) \in E$
        if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
            $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
            $\text{DECREASEKEY}(v, \text{dist}(v))$

Further, observe that after a vertex $v$ is $\text{EXTRACTED}$,
$\text{dist}(v)$ certainly doesn’t increase

So we focus on proving that for all $v$,
when vertex $v$ is $\text{EXTRACTED}$, $\text{dist}(v) = \delta(s, v)$
Proof of Correctness

**Lemma** Whenever a vertex $v$ is EXTRACTED, $\text{dist}(v) = \delta(s, v)$

**Proof**
Proof of Correctness

Lemma Whenever a vertex $v$ is EXTRACTED, $\text{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let $v$ be the first vertex to be EXTRACTED with $\text{dist}(v) \neq \delta(s, v)$
Proof of Correctness

Lemma  Whenever a vertex $v$ is EXTRACTED, $\text{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let $v$ be the first vertex to be EXTRACTED with $\text{dist}(v) \neq \delta(s, v)$.

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s, s)$.
Proof of Correctness

Lemma  Whenever a vertex $v$ is \texttt{EXTRACTED}, $\text{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let $v$ be the first vertex to be \texttt{EXTRACTED} with $\text{dist}(v) \neq \delta(s, v)$.

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)
Proof of Correctness

**Lemma** Whenever a vertex \( v \) is extracted, \( \text{dist}(v) = \delta(s,v) \)

**Proof**

To derive a contradiction, we let \( v \) be the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s,v) \).

\( v \) cannot be the source, \( s \) because \( \text{dist}(s) = 0 = \delta(s,s) \). (this is in the algorithm description)

There must be a path from \( s \) to \( v \), otherwise \( \text{dist}(v) = \infty = \delta(s,v) \)
Proof of Correctness

**Lemma** Whenever a vertex $v$ is **Extracted**, $\text{dist}(v) = \delta(s, v)$

**Proof**

To derive a contradiction, we let $v$ be the first vertex to be **Extracted** with $\text{dist}(v) \neq \delta(s, v)$.

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s, s)$  
(\textit{this is in the algorithm description})

There must be a path from $s$ to $v$, otherwise

\[ \text{dist}(v) = \infty = \delta(s, v) \]
(\textit{Dijkstra doesn’t find paths that aren’t there})
Proof of Correctness

Lemma Whenever a vertex $v$ is **EXTRACTED**, $\text{dist}(v) = \delta(s,v)$

Proof

To derive a contradiction, we let $v$ be the first vertex to be **EXTRACTED** with $\text{dist}(v) \neq \delta(s,v)$

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s,s)$

(this is in the algorithm description)

There must be a path from $s$ to $v$, otherwise

$$\text{dist}(v) = \infty = \delta(s,v)$$

(Dijkstra doesn’t find paths that aren’t there)

Consider the point in the algorithm immediately before $v$ is **EXTRACTED**
Proof of Correctness

Lemma Whenever a vertex \( v \) is \textsc{Extracted}, \( \text{dist}(v) = \delta(s, v) \)

Proof

To derive a contradiction, we let \( v \) be the first vertex to be \textsc{Extracted} with \( \text{dist}(v) \neq \delta(s, v) \)

\( v \) cannot be the source, \( s \) because \( \text{dist}(s) = 0 = \delta(s, s) \) \( (this\ is\ in\ the\ algorithm\ description) \)

There must be a path from \( s \) to \( v \), otherwise

\[ \text{dist}(v) = \infty = \delta(s, v) \] \( (Dijkstra\ doesn’t\ find\ paths\ that\ aren’t\ there) \)

Consider the point in the algorithm immediately before \( v \) is \textsc{Extracted}

In particular consider a shortest path from \( s \) to \( v \):
Proof of Correctness

**Lemma** Whenever a vertex \( v \) is extracted, \( \text{dist}(v) = \delta(s,v) \)

**Proof**

To derive a contradiction, we let \( v \) be the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s,v) \). 

\( v \) cannot be the source, \( s \) because \( \text{dist}(s) = 0 = \delta(s,s) \)  

(this is in the algorithm description)

There must be a path from \( s \) to \( v \), otherwise 

\[ \text{dist}(v) = \infty = \delta(s,v) \]  

(Dijkstra doesn’t find paths that aren’t there)

Consider the point in the algorithm immediately before \( v \) is extracted

In particular consider a shortest path from \( s \) to \( v \):

![Diagram of a shortest path from s to v](image)
Proof of Correctness

**Lemma** Whenever a vertex $v$ is extracted, $\text{dist}(v) = \delta(s,v)$

**Proof**

To derive a contradiction, we let $v$ be the first vertex to be extracted with $\text{dist}(v) \neq \delta(s,v)$.

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s,s)$

There must be a path from $s$ to $v$, otherwise

$$\text{dist}(v) = \infty = \delta(s,v)$$

(this is in the algorithm description)

$\text{(Dijkstra doesn’t find paths that aren’t there)}$

Consider the point in the algorithm immediately before $v$ is extracted.

In particular consider a shortest path from $s$ to $v$:
Proof of Correctness

**Lemma** Whenever a vertex $v$ is **Extracted**, $\text{dist}(v) = \delta(s, v)$

**Proof**

To derive a contradiction, we let $v$ be the first vertex to be **Extracted** with $\text{dist}(v) \neq \delta(s, v)$

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

There must be a path from $s$ to $v$, otherwise

$$\text{dist}(v) = \infty = \delta(s, v)$$

(Dijkstra doesn’t find paths that aren’t there)

Consider the point in the algorithm immediately before $v$ is **Extracted**

In particular consider a shortest path from $s$ to $v$:

these are the vertices still in the queue
Proof of Correctness

Lemma Whenever a vertex $v$ is EXTRACTED, $\text{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let $v$ be the first vertex to be EXTRACTED with $\text{dist}(v) \neq \delta(s, v)$.

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s, s)$ (this is in the algorithm description).

There must be a path from $s$ to $v$, otherwise

$$\text{dist}(v) = \infty = \delta(s, v)$$

(Dijkstra doesn’t find paths that aren’t there).

Consider the point in the algorithm immediately before $v$ is EXTRACTED.

In particular consider a shortest path from $s$ to $v$:

these are the vertices still in the queue ($v$ is in but $s$ isn’t)
Proof of Correctness

**Lemma** Whenever a vertex $v$ is **extracted**, $\text{dist}(v) = \delta(s, v)$

**Proof**

To derive a contradiction, we let $v$ be the first vertex to be **extracted** with $\text{dist}(v) \neq \delta(s, v)$.

$v$ cannot be the source, $s$ because $\text{dist}(s) = 0 = \delta(s, s)$

There must be a path from $s$ to $v$, otherwise $\text{dist}(v) = \infty = \delta(s, v)$

*(this is in the algorithm description)*

*(Dijkstra doesn’t find paths that aren’t there)*

Consider the point in the algorithm immediately before $v$ is **extracted**

In particular consider a shortest path from $s$ to $v$:

These are the vertices still in the queue

($v$ is in but $s$ isn’t)

$y$ is the first vertex on the path that is in the queue

(there must be one because $v$ is in the queue but $s$ isn’t)

$x$ is the vertex before $y$ on the path

(it’s definitely not in the queue)
Proof of Correctness

$v$ is the first vertex to be EXTRACTED with $\text{dist}(v) \neq \delta(s, v)$

Consider the point in the algorithm immediately before $v$ is EXTRACTED.

In particular consider a shortest path from $s$ to $v$:

- $y$ is the first vertex on the path that is in the queue.
- $x$ is the vertex before $y$ on the path.
- These are the vertices still in the queue.
Proof of Correctness

\( v \) is the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is extracted.

In particular consider a shortest path from \( s \) to \( v \):

- these are the vertices still in the queue
- \( y \) is the first vertex on the path that is in the queue
- \( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path
Proof of Correctness

\( v \) is the first vertex to be **Extracted** with \( \text{dist}(v) \neq \delta(s,v) \)

Consider the point in the algorithm immediately before \( v \) is **Extracted**

In particular consider a shortest path from \( s \) to \( v \):

- these are the vertices still in the **queue**
- \( y \) is the first vertex on the path that is in the **queue**
- \( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path
Proof of Correctness

\( v \) is the first vertex to be \textbf{EXTRACTED} with \( \text{dist}(v) \neq \delta(s,v) \)

Consider the point in the algorithm immediately before \( v \) is \textbf{EXTRACTED}

In particular consider a shortest path from \( s \) to \( v \):

\( y \) is the first vertex on the path that is in the \textbf{queue}

\( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path
Proof of Correctness

\( v \) is the first vertex to be \textsc{Extracted} with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is \textsc{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

- \( y \) is the first vertex on the path that is in the queue
- \( x \) is the vertex before \( y \) on the path
- \( v \) is the first vertex to be \textsc{Extracted}
- \( s \) is the start vertex

The path shown from \( s \) to \( y \) is a shortest path \( \text{(otherwise, the path to } v \text{ isn't shortest)} \)
Proof of Correctness

\( \nu \) is the first vertex to be \textbf{EXTRACTED} with \( \text{dist}(\nu) \neq \delta(s, \nu) \)

Consider the point in the algorithm immediately before \( \nu \) is \textbf{EXTRACTED}

In particular consider a shortest path from \( s \) to \( \nu \):

\( \nu \) is the first vertex to be \textbf{EXTRACTED}

\( y \) is the first vertex on the path that is in the queue

\( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path \( \text{(otherwise, the path to \( \nu \) isn't shortest)} \)
Proof of Correctness

\( v \) is the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is extracted.

In particular consider a shortest path from \( s \) to \( v \):

The path shown from \( s \) to \( y \) is a shortest path \( \text{otherwise, the path to } v \text{ isn't shortest} \)

therefore, \( \delta(s, y) \leq \delta(s, v) \)
Proof of Correctness

\( v \) is the first vertex to be Extracted with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is Extracted

In particular consider a shortest path from \( s \) to \( v \):

The path shown from \( s \) to \( y \) is a shortest path \( (\text{otherwise, the path to } v \text{ isn’t shortest}) \)

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is Extracted from the queue before \( v \)
Proof of Correctness

\( v \) is the first vertex to be *Extracted* with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is *Extracted*

In particular consider a shortest path from \( s \) to \( v \):

- \( v \) is the first vertex to be *Extracted* with \( \text{dist}(v) \neq \delta(s, v) \)
- The vertex \( x \) is *Extracted* from the queue before \( v \)

The path shown from \( s \) to \( y \) is a shortest path *otherwise, the path to \( u \) isn’t shortest*

\[ \delta(s, y) \leq \delta(s, v) \]

The vertex \( x \) is *Extracted* from the queue before \( v \)

\[ \text{dist}(x) = \delta(s, x) \]
Proof of Correctness

\( v \) is the first vertex to be \textbf{Extracted} with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is \textbf{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

\( y \) is the first vertex on the path that is in the queue

\( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path\;\;\;\;(\text{othersise, the path to } v \text{ isn't shortest})

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is \textbf{Extracted} from the queue before \( v \)

therefore \( \text{dist}(x) = \delta(s, x) \)\;\;\;\;(v \text{ is the first vertex Extracted with the wrong distance})
Proof of Correctness

\( v \) is the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is extracted

In particular consider a shortest path from \( s \) to \( v \):

The path shown from \( s \) to \( y \) is a shortest path \( \text{(otherwise, the path to} \ v \text{isn’t shortest)} \)

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is extracted from the queue before \( v \)

therefore \( \text{dist}(x) = \delta(s, x) \) \( \text{(} v \text{is the first vertex} \text{ extracted with the wrong distance)} \)

Further, when \( x \) was extracted we relaxed edge \( (x, y) \)
Proof of Correctness

\( v \) is the first vertex to be \textbf{Extracted} with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is \textbf{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

- \( y \) is the first vertex on the path that is in the queue
- \( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path \( (\text{otherwise, the path to } v \text{ isn't shortest}) \)

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is \textbf{Extracted} from the queue \textit{before} \( v \)

therefore \( \text{dist}(x) = \delta(s, x) \) \( (v \text{ is the first vertex } \textbf{Extracted} \text{ with the wrong distance}) \)

Further, when \( x \) was \textbf{Extracted} we relaxed edge \((x, y)\)

therefore \( \text{dist}(y) \leq \text{dist}(s, x) + \text{weight}(x, y) \)
Proof of Correctness

\( v \) is the first vertex to be \textit{Extracted} with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is \textit{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

- these are the vertices still in the queue
- \( y \) is the first vertex on the path that is in the queue
- \( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path \( \text{otherwise, the path to } v \text{ isn't shortest} \)

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is \textit{Extracted} from the queue \textit{before} \( v \)

therefore \( \text{dist}(x) = \delta(s, x) \) \( v \text{ is the first vertex } \textit{Extracted} \text{ with the wrong distance} \)

Further, when \( x \) was \textit{Extracted} we relaxed edge \((x, y)\)

therefore \( \text{dist}(y) \leq \text{dist}(s, x) + \text{weight}(x, y) \)
Proof of Correctness

\( v \) is the first vertex to be \textit{Extracted} with \( \text{dist}(v) \neq \delta(s,v) \)

Consider the point in the algorithm immediately before \( v \) is \textit{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

The path shown from \( s \) to \( y \) is a shortest path \( \text{(otherwise, the path to } v \text{ isn't shortest)} \)

therefore, \( \delta(s,y) \leq \delta(s,v) \)

The vertex \( x \) is \textit{Extracted} from the queue \textit{before} \( v \)

therefore \( \text{dist}(x) = \delta(s,x) \) \( \text{ (} v \text{ is the first vertex } \textit{Extracted} \text{ with the wrong distance)} \)

Further, when \( x \) was \textit{Extracted} we relaxed edge \( (x,y) \)

therefore \( \text{dist}(y) \leq \delta(s,x) + \text{weight}(x,y) \)
Proof of Correctness

\( v \) is the first vertex to be Extracted with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is Extracted

In particular consider a shortest path from \( s \) to \( v \):

- these are the vertices still in the queue
- \( y \) is the first vertex on the path that is in the queue
- \( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path (otherwise, the path to \( v \) isn't shortest)

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is Extracted from the queue before \( v \)

therefore \( \text{dist}(x) = \delta(s, x) \) (\( v \) is the first vertex Extracted with the wrong distance)

Further, when \( x \) was Extracted we relaxed edge \((x, y)\)

therefore \( \text{dist}(y) \leq \delta(s, x) + \text{weight}(x, y) \)
Proof of Correctness

\( v \) is the first vertex to be \texttt{Extracted} with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is \texttt{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

The path shown from \( s \) to \( y \) is a shortest path \textit{(otherwise, the path to} \( v \) \textit{isn’t shortest)}

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is \texttt{Extracted} from the queue \textit{before} \( v \)

therefore \( \text{dist}(x) = \delta(s, x) \) \textit{(\( v \) is the first vertex \texttt{Extracted} with the wrong distance)}

Further, when \( x \) was \texttt{Extracted} we relaxed edge \( (x, y) \)

therefore \( \text{dist}(y) \leq \delta(s, x) + \text{weight}(x, y) = \delta(s, y) \)
Proof of Correctness

\( v \) is the first vertex to be \text{EXTRACTED} with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is \text{EXTRACTED}

In particular consider a shortest path from \( s \) to \( v \):

- \( y \) is the first vertex on the path that is in the queue
- \( x \) is the vertex before \( y \) on the path

The path shown from \( s \) to \( y \) is a shortest path \( (\text{otherwise, the path to } v \text{ isn’t shortest}) \)

therefore, \( \delta(s, y) \leq \delta(s, v) \)

The vertex \( x \) is \text{EXTRACTED} from the queue \text{before} \( v \)

therefore \( \text{dist}(x) = \delta(s, x) \) \( (v \text{ is the first vertex } \text{EXTRACTED} \text{ with the wrong distance}) \)

Further, when \( x \) was \text{EXTRACTED} we \text{relaxed} edge \( (x, y) \)

therefore \( \text{dist}(y) \leq \delta(s, x) + \text{weight}(x, y) = \delta(s, y) \) \( (\text{the path shown is shortest}) \)
Proof of Correctness

\( v \) is the first vertex to be \textsc{Extracted} with \( \text{dist}(v) \neq \delta(s,v) \)

Consider the point in the algorithm immediately before \( v \) is \textsc{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

We have shown that: \( \text{dist}(y) \leq \delta(s,y) \) and \( \delta(s,y) \leq \delta(s,v) \)
Proof of Correctness

\(v\) is the first vertex to be \textit{Extracted} with \(\text{dist}(v) \neq \delta(s, v)\)

Consider the point in the algorithm immediately before \(v\) is \textit{Extracted}.

In particular consider a shortest path from \(s\) to \(v\):

We have shown that: \(\text{dist}(y) \leq \delta(s, y)\) and \(\delta(s, y) \leq \delta(s, v)\)

\textit{We are almost there :)}
Proof of Correctness

\( v \) is the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is extracted

In particular consider a shortest path from \( s \) to \( v \):

We have shown that: \( \text{dist}(y) \leq \delta(s, y) \) and \( \delta(s, y) \leq \delta(s, v) \)

We are almost there :)

Finally, we have that \( \text{dist}(v) \leq \text{dist}(y) \)
Proof of Correctness

\( \nu \) is the first vertex to be extracted with \( \text{dist}(\nu) \neq \delta(s, \nu) \)

Consider the point in the algorithm immediately before \( \nu \) is extracted

In particular consider a shortest path from \( s \) to \( \nu \):

We have shown that: \( \text{dist}(y) \leq \delta(s, y) \) and \( \delta(s, y) \leq \delta(s, \nu) \)

We are almost there :)

Finally, we have that \( \text{dist}(v) \leq \text{dist}(y) \)

because \( \nu \) is extracted next \( \text{(so has the minimum dist)} \)
Proof of Correctness

\( v \) is the first vertex to be \texttt{EXTRACTED} with \( \text{dist}(v) \neq \delta(s, v) \)

Consider the point in the algorithm immediately before \( v \) is \texttt{EXTRACTED}

In particular consider a shortest path from \( s \) to \( v \):

We have shown that: \( \text{dist}(y) \leq \delta(s, y) \) and \( \delta(s, y) \leq \delta(s, v) \)

\textit{We are almost there :)}

Finally, we have that \( \text{dist}(v) \leq \text{dist}(y) \)

because \( v \) is \texttt{EXTRACTED} next \hspace{1cm} (so has the minimum \text{dist})

\textit{Putting it all together,} \hspace{1cm} \text{dist}(v) \leq \text{dist}(y) \leq \delta(s, y) \leq \delta(s, v)
Proof of Correctness

\( v \) is the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s,v) \)

Consider the point in the algorithm immediately before \( v \) is extracted

In particular consider a shortest path from \( s \) to \( v \):

- \( y \) is the first vertex on the path that is in the queue
- \( x \) is the vertex before \( y \) on the path

We have shown that: \( \text{dist}(y) \leq \delta(s,y) \) and \( \delta(s,y) \leq \delta(s,v) \)

We are almost there :)

Finally, we have that \( \text{dist}(v) \leq \text{dist}(y) \)

because \( v \) is extracted next (so has the minimum \( \text{dist} \))

Putting it all together, \( \text{dist}(v) \leq \delta(s,v) \)
Proof of Correctness

\( v \) is the first vertex to be \textsc{Extracted} with \( \text{dist}(v) \neq \delta(s,v) \)

Consider the point in the algorithm immediately before \( v \) is \textsc{Extracted}

In particular consider a shortest path from \( s \) to \( v \):

\[ \text{We have shown that: } \text{dist}(y) \leq \delta(s,y) \text{ and } \delta(s,y) \leq \delta(s,v) \]

\textit{We are almost there :)}

Finally, we have that \( \text{dist}(v) \leq \text{dist}(y) \)

because \( v \) is \textsc{Extracted} next \ \ (so has the minimum \text{dist})

\textit{Putting it all together, } \text{dist}(v) \leq \delta(s,v) \]
**Proof of Correctness**

\( v \) is the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s,v) \)

Consider the point in the algorithm immediately before \( v \) is extracted.

In particular, consider a shortest path from \( s \) to \( v \):

- \( v \) is the first vertex on the path that is in the queue.
- \( y \) is the first vertex on the path that is in the queue.
- \( x \) is the vertex before \( y \) on the path.

\[ \text{these are the vertices still in the queue} \]

**Summary**

We assumed that \( v \) was the first vertex to be extracted with \( \text{dist}(v) \neq \delta(s,v) \).

We proved that \( \text{dist}(v) \leq \delta(s,v) \).

However, we also have that, \( \text{dist}(v) \geq \delta(s,v) \) (Dijkstra only finds actual paths).

So we have that \( \text{dist}(v) = \delta(s,v) \) \( \text{Contradiction!} \) there is no such \( v \).

*(in other words all distances are correct)*
Time Complexity

The time complexity of Dijkstra’s algorithm depends on the time complexities of \texttt{INSERT}, \texttt{DECREASE\textsc{Key}} and \texttt{EXTRACT\textsc{Min}} supported by the queue.

\begin{verbatim}
Dijkstra(s)

For all $v$, set $\text{dist}(v) = \infty$
set $\text{dist}(s) = 0$
For each $v$, do \texttt{INSERT}(v, \text{dist}(v))
while the queue is not empty
    $u = \texttt{EXTRACT\textsc{Min}}()$
    for every edge $(u, v) \in E$
        if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
            $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
            \texttt{DECREASE\textsc{Key}}(v, \text{dist}(v))
\end{verbatim}
The time complexity of Dijkstra’s algorithm depends on the time complexities of \textsc{Insert}, \textsc{DecreaseKey} and \textsc{ExtractMin} supported by the queue.

\begin{algorithm}
\textbf{\textsc{Dijkstra}(s)}

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \textsc{Insert}(\( v, \text{dist}(v) \))
while the queue is not empty
\hspace{1em} \( u = \text{ExtractMin}() \)
\hspace{1em} for every edge \((u, v) \in E\)
\hspace{2em} if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)
\hspace{3em} \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)
\hspace{3em} \textsc{DecreaseKey}(v, \text{dist}(v))
\end{algorithm}

\( O(|V|) \) time
The time complexity of Dijkstra’s algorithm depends on the time complexities of \textsc{Insert}, \textsc{DecreaseKey} and \textsc{ExtractMin} supported by the queue.

\begin{itemize}
  \item \textbf{Dijkstra}(s)
  \end{itemize}

\begin{itemize}
  \item For all \( v \), set \( \text{dist}(v) = \infty \)
  \item set \( \text{dist}(s) = 0 \)
  \item For each \( v \), do \textsc{Insert}(\( v, \text{dist}(v) \))
  \item while the queue is not empty
    \begin{itemize}
      \item \( u = \text{ExtractMin}() \)
      \item for every edge \( (u,v) \in E \)
        \begin{itemize}
          \item if \( \text{dist}(v) > \text{dist}(u) + \text{weight}(u,v) \)
            \begin{itemize}
              \item \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u,v) \)
              \item \textsc{DecreaseKey}(v, \text{dist}(v))
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

\( O(|V|) \) time

(we store \textit{dist} in an array of length \(|V|\))
The time complexity of Dijkstra’s algorithm depends on the time complexities of **INSERT**, **DECREASEKEY** and **EXTRACTMIN** supported by the queue.

### Dijkstra(s)

For all $v$, set $\text{dist}(v) = \infty$

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

For each $v$, do $\text{INSERT}(v, \text{dist}(v))$

while the queue is not empty

\[ u = \text{EXTRACTMIN}() \]

for every edge $(u, v) \in E$

\[ \text{if dist}(v) > \text{dist}(u) + \text{weight}(u, v) \]

\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]

\[ \text{DECREASEKEY}(v, \text{dist}(v)) \]

$O(|V|)$ time

(we store $\text{dist}$ in an array of length $|V|$)

$O(1)$ time
**Time Complexity**

The time complexity of Dijkstra’s algorithm depends on the time complexities of **INSERT**, **DECREASEKEY** and **EXTRACTMIN**

supported by the queue

**Dijkstra\((s)\)**

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(</td>
<td>V</td>
</tr>
<tr>
<td>(O(1)) time</td>
<td></td>
</tr>
<tr>
<td>(O(</td>
<td>V</td>
</tr>
</tbody>
</table>

For all \(v\), set \(\text{dist}(v) = \infty\)

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

For each \(v\), do **INSERT**\((v, \text{dist}(v))\)

while the queue is not empty

\[ u = \text{EXTRACTMIN}() \]

for every edge \((u, v) \in E\)

if \(\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)\)

\[ \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \]

**DECREASEKEY**\((v, \text{dist}(v))\)
Time Complexity

The time complexity of Dijkstra’s algorithm depends on the time complexities of \texttt{INSERT}, \texttt{DECREASEKEY} and \texttt{EXTRACTMIN} supported by the queue.

\begin{algorithm}
\begin{algorithmic}
\caption{Dijkstra(s)}
\end{algorithmic}
\end{algorithm}

\begin{itemize}
\item For all \( v \), set \( \text{dist}(v) = \infty \)
\item set \( \text{dist}(s) = 0 \)
\item For each \( v \), do \text{INSERT}(v, \text{dist}(v))
\end{itemize}

\begin{algorithmic}
\While{the queue is not empty}
\State \( u = \text{EXTRACTMIN}() \)
\For{every edge \((u, v) \in E\)}
\If{\( \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \)}
\State \( \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \)
\State \text{DECREASEKEY}(v, \text{dist}(v))
\EndIf
\EndFor
\EndWhile
\end{algorithmic}

\( O(|V| \cdot T_{\text{INSERT}}) \) time \( for \ the \ setup \)
Time Complexity

The time complexity of Dijkstra’s algorithm depends on the time complexities of **Insert**, **DecreaseKey** and **ExtractMin** supported by the queue.

\[
\text{Dijkstra}(s)
\]

For all \( v \), set \( \text{dist}(v) = \infty \)
set \( \text{dist}(s) = 0 \)
For each \( v \), do \( \text{Insert}(v, \text{dist}(v)) \)
while the queue is not empty
\[
\begin{aligned}
& u = \text{ExtractMin}() \\
& \text{for every edge } (u, v) \in E \\
& \quad \text{if } \text{dist}(v) > \text{dist}(u) + \text{weight}(u, v) \\
& \quad \quad \text{dist}(v) = \text{dist}(u) + \text{weight}(u, v) \\
& \quad \quad \text{DecreaseKey}(v, \text{dist}(v))
\end{aligned}
\]

\[O(|V| \cdot T_{\text{Insert}})\] time

for the setup

\[O(T_{\text{ExtractMin}})\] time per iteration
Time Complexity

The time complexity of Dijkstra’s algorithm depends on the time complexities of 
\textbf{\textsc{Insert}}, \textbf{\textsc{DecreaseKey}} and \textbf{\textsc{ExtractMin}}
supported by the queue

\begin{algorithm}
\textsc{Dijkstra}(s)
\begin{enumerate}
  \item For all $v$, set $\text{dist}(v) = \infty$
  \item set $\text{dist}(s) = 0$
  \item For each $v$, do \text{\textsc{Insert}}($v$, $\text{dist}(v)$)
  \item while the queue is not empty
    \begin{enumerate}
      \item $u = \text{\textsc{ExtractMin}}()$
      \item for every edge $(u, v) \in E$
        \begin{enumerate}
          \item if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)$
            \begin{enumerate}
              \item $\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)$
              \item \text{\textsc{DecreaseKey}}($v$, $\text{dist}(v)$)
            \end{enumerate}
        \end{enumerate}
    \end{enumerate}
\end{enumerate}
\end{algorithm}

$O(|V| \cdot T_{\text{\textsc{Insert}}})$ time
\textit{for the setup}

$O(T_{\text{\textsc{ExtractMin}}})$ time per iteration

relaxing an edge takes $O(T_{\text{\textsc{DecreaseKey}}})$ time
**Time Complexity**

The time complexity of Dijkstra’s algorithm depends on the time complexities of $\text{INSERT}$, $\text{DECREASE KEY}$ and $\text{EXTRACT MIN}$ supported by the queue.

---

**$\text{DIJKSTRA}(s)$**

- For all $v$, set $\text{dist}(v) = \infty$
- set $\text{dist}(s) = 0$
- For each $v$, do $\text{INSERT}(v, \text{dist}(v))$
- while the queue is not empty
  - $u = \text{EXTRACT MIN}()$
  - for every edge $(u,v) \in E$
    - if $\text{dist}(v) > \text{dist}(u) + \text{weight}(u,v)$
      - $\text{dist}(v) = \text{dist}(u) + \text{weight}(u,v)$
      - $\text{DECREASE KEY}(v, \text{dist}(v))$

---

We do $O(|V|)$ iterations of the while loop and relax each edge at most once...
Time Complexity

The time complexity of Dijkstra’s algorithm depends on the time complexities of \textsc{Insert}, \textsc{DecreaseKey} and \textsc{ExtractMin} supported by the queue.

\begin{itemize}
  \item \textbf{\textsc{Dijkstra}(s)}
    \begin{enumerate}
      \item For all \(v\), set \(\text{dist}(v) = \infty\)
      \item set \(\text{dist}(s) = 0\)
      \item For each \(v\), do \textsc{Insert}(\(v, \text{dist}(v)\))
      \item while the queue is not empty
        \begin{enumerate}
          \item \(u = \text{ExtractMin}()\)
          \item for every edge \((u, v) \in E\)
            \begin{enumerate}
              \item if \(\text{dist}(v) > \text{dist}(u) + \text{weight}(u, v)\)
              \item \(\text{dist}(v) = \text{dist}(u) + \text{weight}(u, v)\)
              \item \textsc{DecreaseKey}(v, \text{dist}(v))
            \end{enumerate}
        \end{enumerate}
    \end{enumerate}
\end{itemize}

We do \(O(|V|)\) iterations of the while loop and \textit{relax} each edge at most once...

so overall this takes:

\[ O(|V| \cdot T_{\text{insert}}) \text{ time for the setup} \]
\[ O(T_{\text{extractmin}}) \text{ time per iteration} \]
\[ O(T_{\text{decreasekey}}) \text{ time for relaxing an edge} \]

\[ O(|V|) \text{ iterations of the while loop} \]
Time Complexity

The time complexity of Dijkstra’s algorithm depends on the time complexities of \texttt{INSERT}, \texttt{DECREASE\textsc{Key}} and \texttt{EXTRACT\textsc{Min}} supported by the queue.

\begin{algorithm}
\textbf{Dijkstra}(s)
\begin{algorithmic}
\State For all \textit{v}, set dist(\textit{v}) = \infty
\State set dist(s) = 0
\State For each \textit{v}, do \texttt{INSERT(v,dist(\textit{v}))}
\While{the queue is not empty}
\State \textit{u} = \texttt{EXTRACT\textsc{Min}()}
\For{every edge (\textit{u},\textit{v}) \in E}
\If{dist(\textit{v}) > dist(\textit{u}) + weight(\textit{u},\textit{v})}
\State dist(\textit{v}) = dist(\textit{u}) + weight(\textit{u},\textit{v})
\State \texttt{DECREASE\textsc{Key}(v,dist(\textit{v}))}
\EndIf
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

We do $O(|V|)$ iterations of the while loop and 	extit{relax} each edge at most once.

so overall this takes:

$$O(|V| \cdot T_{\text{INSERT}} + |V| \cdot T_{\text{EXTRACT\textsc{Min}}} + |E| \cdot T_{\text{DECREASE\textsc{Key}}})$$ time
Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>DECREASEKEY</th>
<th>EXTRACTMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

The time complexity of Dijkstra’s algorithm depends on the time complexities of \textit{INSERT}, \textit{DECREASEKEY} and \textit{EXTRACTMIN} supported by the queue.
Summary

The time complexity of Dijkstra’s algorithm depends on the time complexities of

\textbf{Insert}, \textbf{DecreaseKey} and \textbf{ExtractMin} supported by the queue

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th></th>
<th>\textbf{Insert}</th>
<th>\textbf{DecreaseKey}</th>
<th>\textbf{ExtractMin}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

What is $n$?
Summary

The time complexity of Dijkstra’s algorithm depends on the time complexities of \textsc{Insert}, \textsc{DecreaseKey} and \textsc{ExtractMin} supported by the queue.

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th></th>
<th>\textsc{Insert}</th>
<th>\textsc{DecreaseKey}</th>
<th>\textsc{ExtractMin}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

What is $n$?

$n$ denotes the number of elements in the queue.
Summary

The time complexity of Dijkstra’s algorithm depends on the time complexities of \textbf{INSERT}, \textbf{DECREASE\textsc{Key}} and \textbf{EXTRACT\textsc{Min}} supported by the queue.

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th>Queue Type</th>
<th>INSERT</th>
<th>DECREASE\textsc{Key}</th>
<th>EXTRACT\textsc{Min}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

What is $n$?

$n$ denotes the number of elements in the queue

so $n \leq |V|$ (one element per vertex)
**Summary**

The time complexity of Dijkstra’s algorithm depends on the time complexities of **Insert**, **DecreaseKey** and **ExtractMin** supported by the queue.

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>DecreaseKey</th>
<th>ExtractMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>$O(1)$</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log</td>
</tr>
</tbody>
</table>

What is $n$?

$n$ denotes the number of elements in the queue

so $n \leq |V|$ (one element per vertex)
Summary

The time complexity of Dijkstra’s algorithm depends on the time complexities of \textsc{Insert}, \textsc{DecreaseKey} and \textsc{ExtractMin} supported by the queue.

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th>Priority Queue</th>
<th>\textsc{Insert}</th>
<th>\textsc{DecreaseKey}</th>
<th>\textsc{ExtractMin}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>(O(1))</td>
<td>(O(</td>
<td>V</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>(O(\log</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(\log</td>
</tr>
</tbody>
</table>
The time complexity of Dijkstra’s algorithm depends on the time complexities of \textbf{INSERT}, \textbf{DECREASEKEY} and \textbf{EXTRACTMIN} supported by the queue.

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th></th>
<th>\textbf{INSERT}</th>
<th>\textbf{DECREASEKEY}</th>
<th>\textbf{EXTRACTMIN}</th>
<th>\textbf{DIJKSTRA} run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>$O(1)$</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
<td>$O(</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log</td>
<td>V</td>
<td>)$</td>
<td>$O(\log</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log</td>
<td>V</td>
</tr>
</tbody>
</table>

|
Summary

The time complexity of Dijkstra’s algorithm depends on the time complexities of
\textbf{Insert}, \textbf{DecreaseKey} and \textbf{ExtractMin}
supported by the queue

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th>Priority Queue</th>
<th>Insert</th>
<th>DecreaseKey</th>
<th>ExtractMin</th>
<th>Dijkstra run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>$O(1)$</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
<td>$O(</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log</td>
<td>V</td>
<td>)$</td>
<td>$O(\log</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log</td>
<td>V</td>
</tr>
</tbody>
</table>

... but Fibonacci Heaps are \textit{complicated}, \textit{amortised} and have \textit{large hidden constants}
Summary

The time complexity of Dijkstra’s algorithm depends on the time complexities of \texttt{INSERT,DECREASEKEY} and \texttt{EXTRACTMIN} supported by the queue.

Recall from earlier the complexities of some priority queues:

<table>
<thead>
<tr>
<th>Priority Queue</th>
<th>\texttt{INSERT}</th>
<th>\texttt{DECREASEKEY}</th>
<th>\texttt{EXTRACTMIN}</th>
<th>\texttt{DIJKSTRA} run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>(O(1))</td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>(O(</td>
<td>V</td>
<td>))</td>
<td>(O(</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>(O(\log</td>
<td>V</td>
<td>))</td>
<td>(O(\log</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(\log</td>
<td>V</td>
</tr>
</tbody>
</table>

…but Fibonacci Heaps are complicated, amortised and have large hidden constants.

Dijkstra’s algorithm solves the single source shortest path algorithm on weighted, directed graphs… with non-negative edge weights.