Data Structures and Algorithms – COMS21103

2015/2016

Single Source Shortest Paths

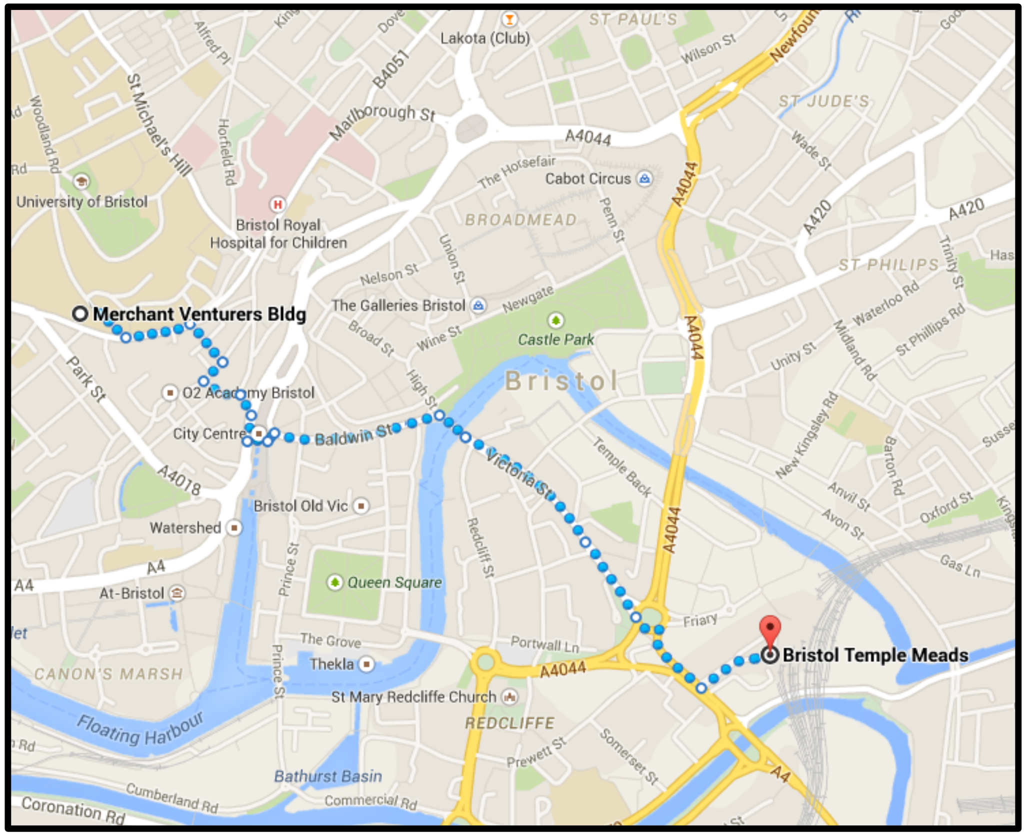
Priority Queues and Dijkstra's Algorithm

Benjamin Sach

University of BRISTOL

University of BRISTOL

In today's lectures we'll be discussing the **single source shortest paths** problem
in a weighted, directed graph. . .



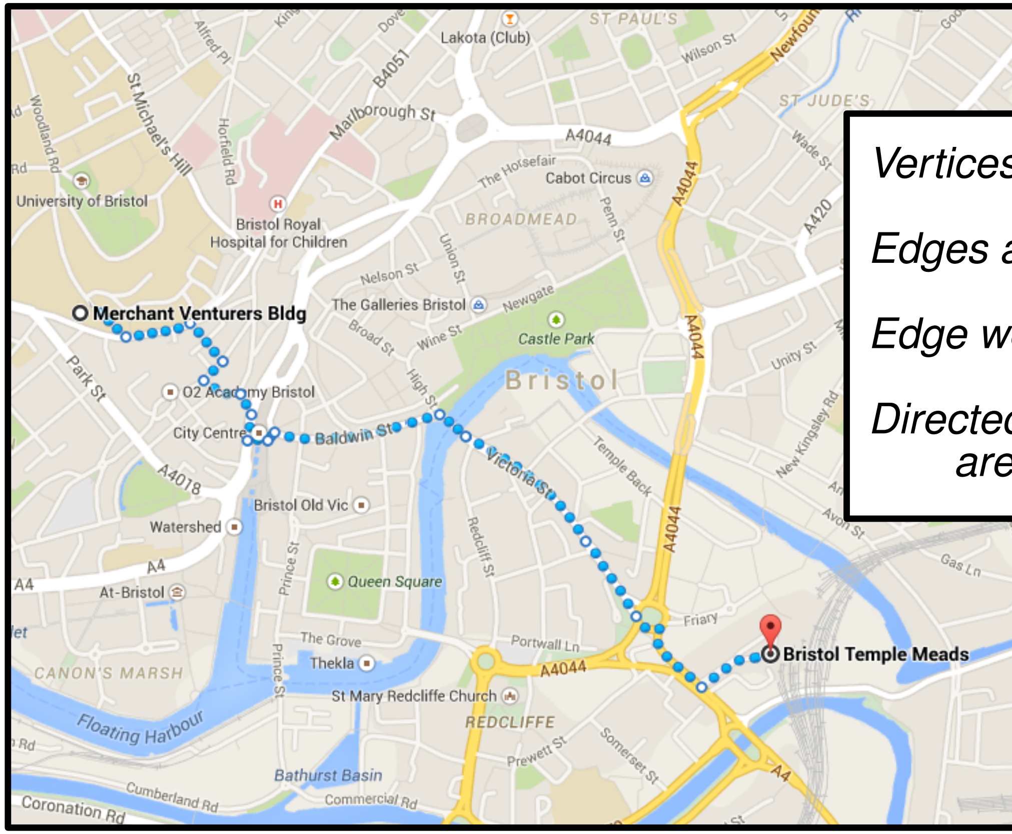
The shortest path from MVB to Temple Meads
(according to Google Maps)

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 lectures we'll be discussing the **single source shortest paths** problem
in a weighted, directed graph. . .



Vertices are junctions

Edges are roads

Edge weights are in miles

Directed edges are one-way streets

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

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 - $x.\text{key}$

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:

$\textsc{Insert}(x, k)$ - 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:

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

$\textsc{DecreaseKey}(x, k)$ - 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:

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

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

where $k < x$.key

$\textsc{ExtractMin}()$ - 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:

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

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**: $\boxed{\textbf{Alice}\ 3}$ $\boxed{\textbf{Bob}\ 5}$ $\boxed{\textbf{Chris}\ 8}$

$\left(\boxed{x \quad x.\text{key}}\right)$

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 $x.\text{key} = k$

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	Bob	Chris
3	5	8

$\left(\begin{array}{|c|}\hline x \\ x.\text{key} \\ \hline\end{array}\right)$

$\textsc{Insert}(\text{Dawn}, 4)$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	Bob	Chris	Dawn
3	5	8	4

$\left(\begin{array}{|c|}\hline x \\ x.\text{key} \\ \hline\end{array}\right)$

$\textsc{Insert}(\text{Dawn}, 4)$

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**:

Alice	Bob	Chris	Dawn
3	5	8	4

$\left(\boxed{\begin{array}{c} x \\ x\text{.key} \end{array}}\right)$

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 $x.\text{key} = k$

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	Bob	Chris	Dawn
3	5	8	4

$\left(\begin{array}{|c|}\hline x \\ x.\text{key} \\ \hline\end{array}\right)$

$\textsc{ExtractMin}()$

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:

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

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	Chris	Dawn
5	8	4

$\left(\boxed{\begin{matrix} x \\ x.\text{key} \end{matrix}} \right)$

$\textsc{ExtractMin}()$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	Chris	Dawn
5	8	4

$\left(\begin{array}{|c|}\hline x \\ x.\text{key} \\ \hline\end{array}\right)$

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 $x.\text{key} = k$

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	**Chris**	**Dawn**
5	8	4

$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}}\right)$

$\textsc{Insert}(\text{Emma}, 6)$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	Chris	Dawn	Emma
5	8	4	6

$\left(\begin{array}{|c|}\hline x \\ x.\text{key} \\ \hline\end{array}\right)$

$\textsc{Insert}(\text{Emma}, 6)$

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:

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

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	**Chris**	**Dawn**	**Emma**
5	8	4	6

$\left(\boxed{\begin{array}{c} x \\ x\text{.key} \end{array}}\right)$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	**Chris**	**Dawn**	**Emma**
5	8	4	6

$\left(\boxed{\begin{matrix} x \\ x.\text{key} \end{matrix}}\right)$

$\textsc{ExtractMin}()$

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:

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

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	**Chris**	**Emma**
5	8	6

$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}} \right)$

$\textsc{ExtractMin}()$

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:

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

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

where $k < x$.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	**Chris**	**Emma**
5	8	6

$\left(\begin{array}{c} x \\ x\text{.key} \end{array} \right)$

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

where $k < x$.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob	Chris	Emma
5	8	6

$\left(\boxed{\begin{matrix} x \\ x.\text{key} \end{matrix}}\right)$

$\textsc{DecreaseKey}(\text{Bob}, 2)$

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 $x.\text{key} = k$

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

Bob	Chris	Emma
2	8	6

$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}}\right)$

DECREASEKEY(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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob 2	**Chris** 8	**Emma** 6	$\left(\boxed{x \; x.\text{key}} \right)$

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Bob 2 | **Chris** 8 | **Emma** 6

$\left(\boxed{x \quad x\text{.key}} \right)$

$\textsc{Insert}(\text{Alice}, 3)$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	**Bob**	**Chris**	**Emma**	x
3	2	8	6	x.key

$\textsc{Insert}(\text{Alice}, 3)$

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:

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

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

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 $x.\text{key} = k$

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	Bob	Chris	Emma
3	2	8	6

$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}}\right)$

$\textsc{ExtractMin}()$

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:

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

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

where $k < x$.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	Chris	Emma
3	8	6

$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}}\right)$

$\textsc{ExtractMin}()$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	**Chris**	**Emma**	x
3	8	6	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:

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

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

where $k < x$.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	Chris	Emma
3	8	6

$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}} \right)$

$\textsc{DecreaseKey}(\text{Chris}, 4)$

Priority Queues

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

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

A priority queue supports the following operations:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice 3	**Chris** 4	**Emma** 6	$\left(\begin{array}{c} x \\ x.\text{key} \end{array}\right)$

$\textsc{DecreaseKey}(\text{Chris}, 4)$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice	Chris	Emma	$\left(\boxed{\begin{matrix} x \\ x.\text{key} \end{matrix}}\right)$
3	4	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:

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

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Alice 3	**Chris** 4	**Emma** 6	$\left(\boxed{x \quad x\text{.key}} \right)$

$\textsc{ExtractMin}()$

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:

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

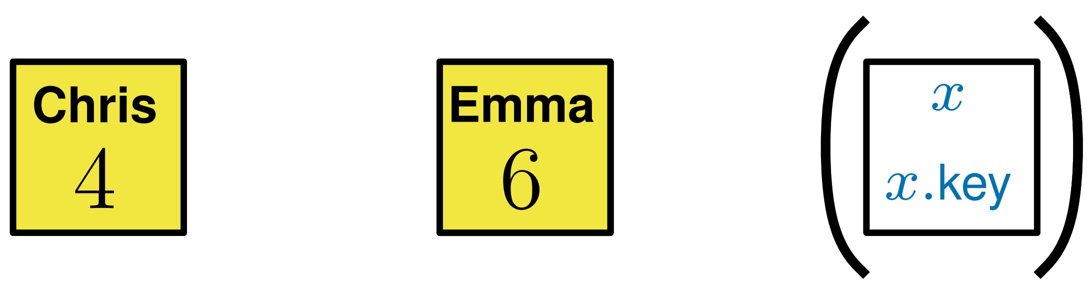
$\textsc{DecreaseKey}(x, k)$ - decreases the value of x.key to k

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:



$\textsc{ExtractMin}()$

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:

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

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

where $k < x$.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Chris	Emma	x
4	6	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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Chris	Emma	x
4	6	x.key

$\textsc{ExtractMin}()$

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:

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

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

where $k < x$.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Emma
6

$\left(\boxed{\begin{matrix} x \\ x\text{.key} \end{matrix}} \right)$

$\textsc{ExtractMin}()$

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:

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

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Emma
6

$\left(\boxed{\begin{matrix} x \\ x\text{.key} \end{matrix}} \right)$

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

Emma
6

$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}}\right)$

$\textsc{ExtractMin}()$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

$$\left(\boxed{\begin{matrix} x \\ x.\text{key} \end{matrix}}\right)$$

$\textsc{ExtractMin}()$

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:

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

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

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

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

A **priority queue**:

$$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}}\right)$$

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

where $k <$ x.key

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

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*

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

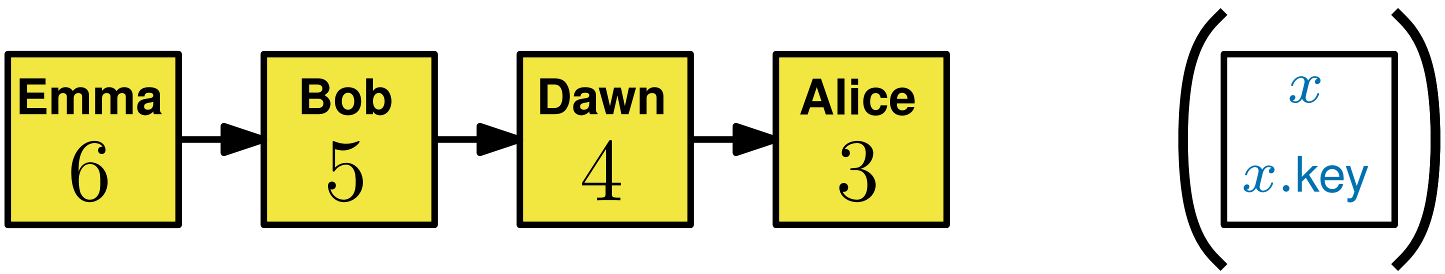
There are many ways in which we could implement a priority queue. . .

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

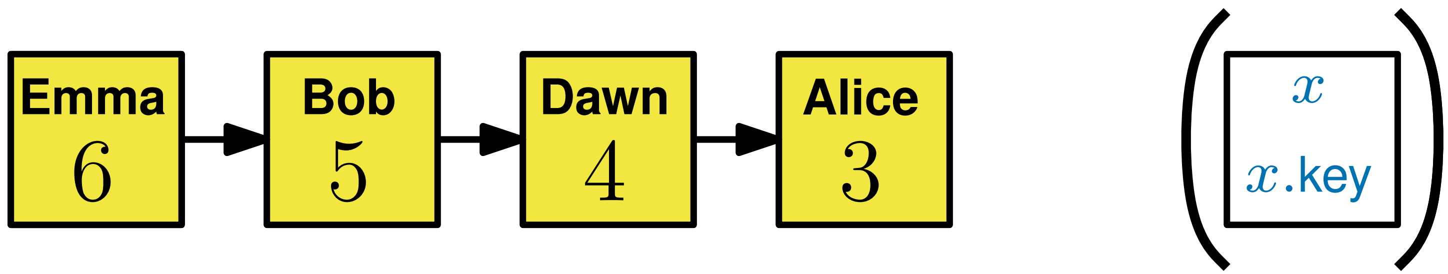
There are many ways in which we could implement a priority queue...

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:



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

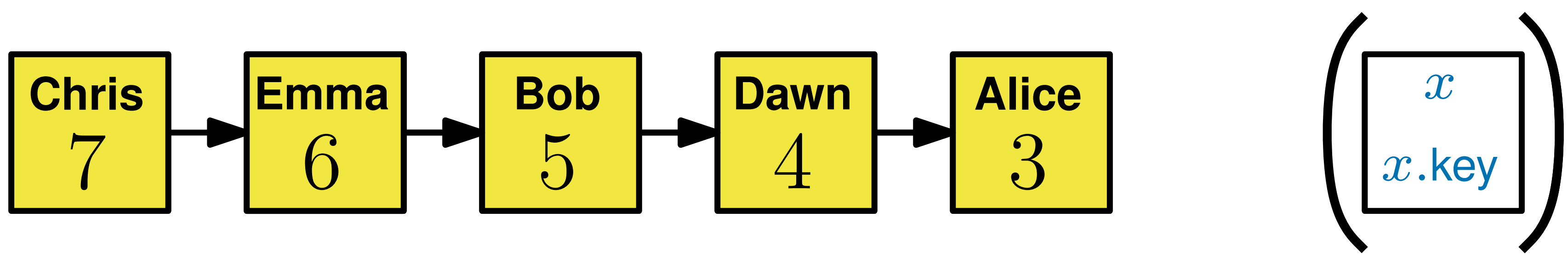
There are many ways in which we could implement a priority queue...

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:



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

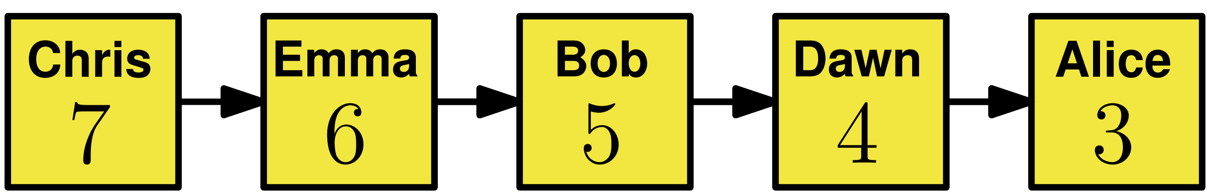
There are many ways in which we could implement a priority queue...

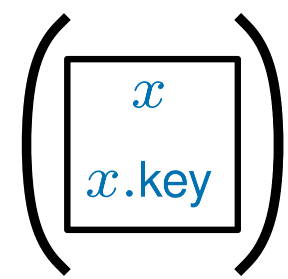
but they aren't all efficient

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:





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

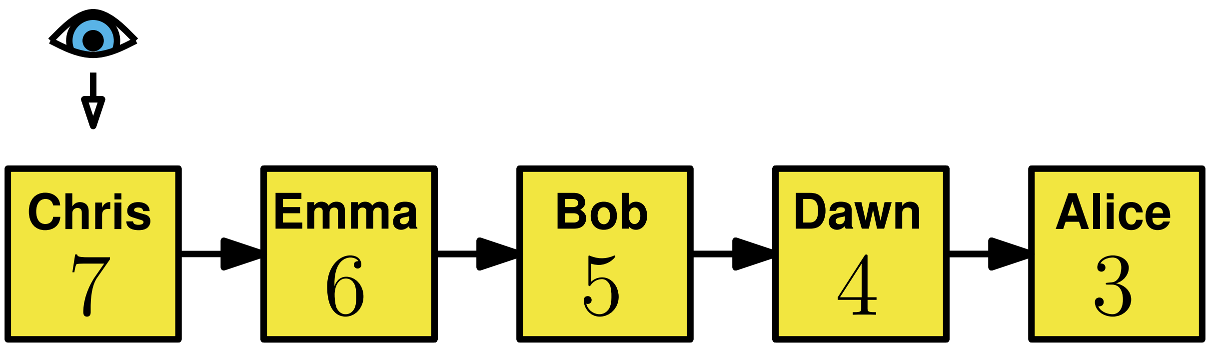
There are many ways in which we could implement a priority queue...

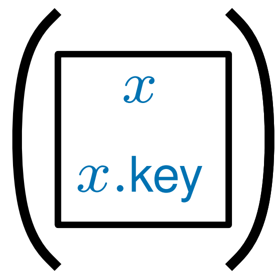
but they aren't all efficient

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

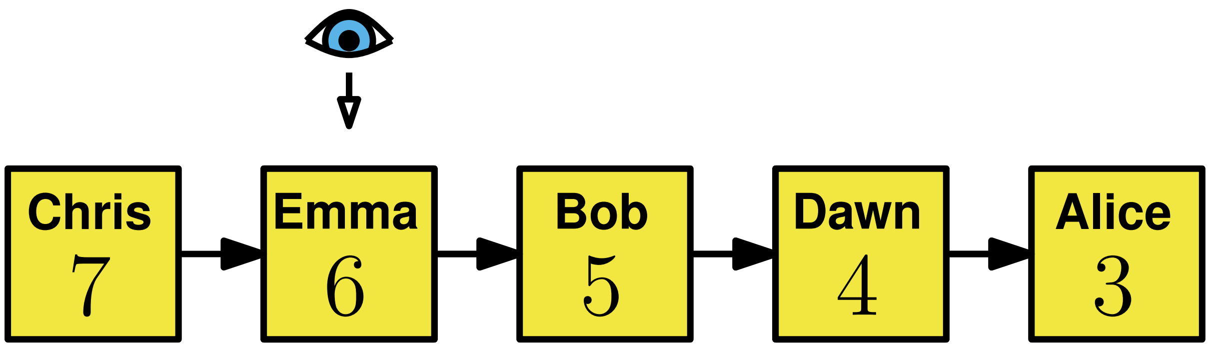
There are many ways in which we could implement a priority queue...

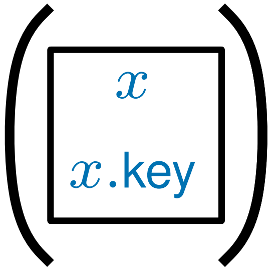
but they aren't all efficient

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

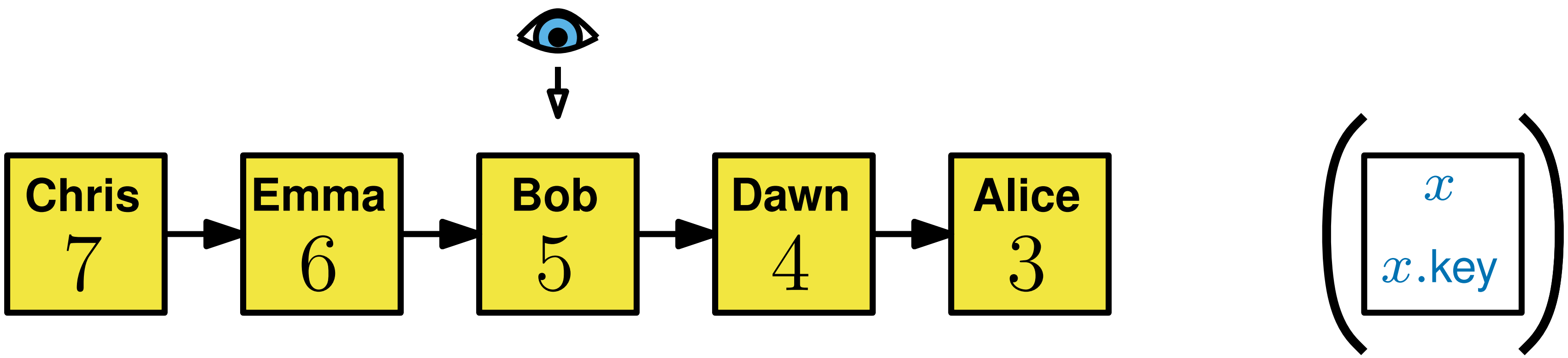
There are many ways in which we could implement a priority queue. . .

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:



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

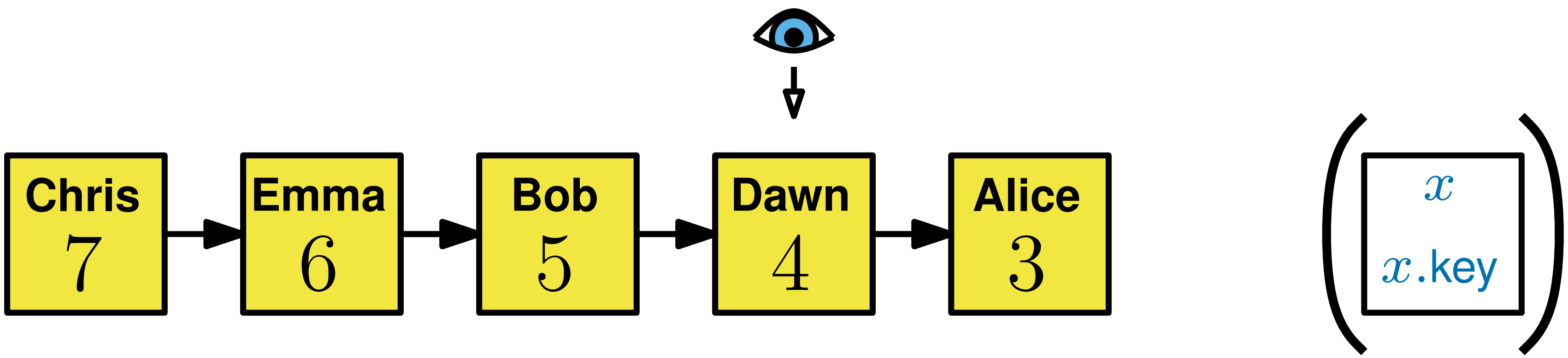
There are many ways in which we could implement a priority queue. . .

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

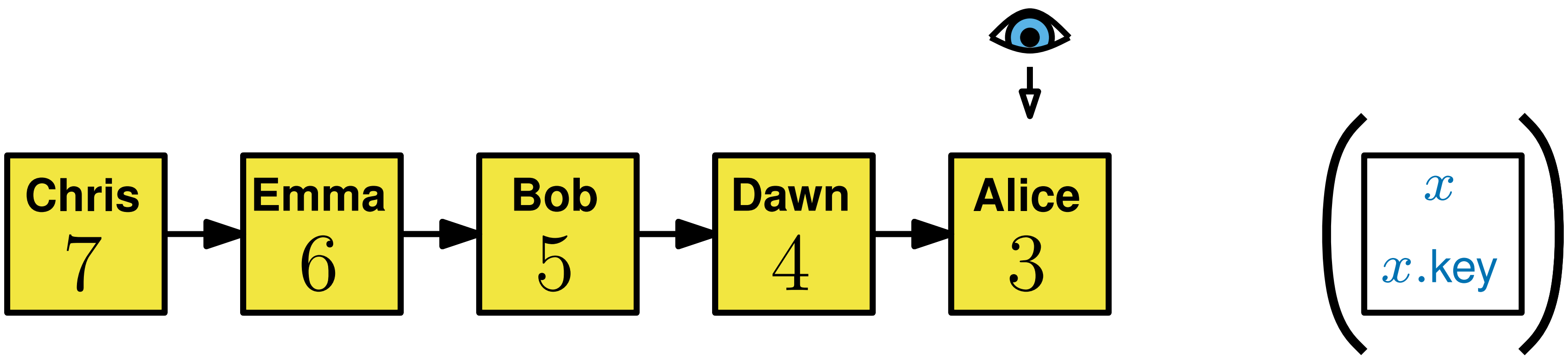
There are many ways in which we could implement a priority queue...

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

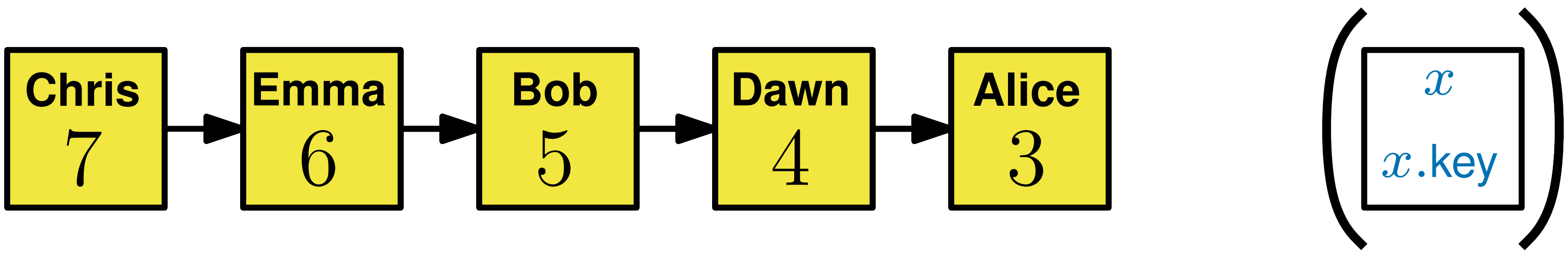
There are many ways in which we could implement a priority queue...

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:



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...

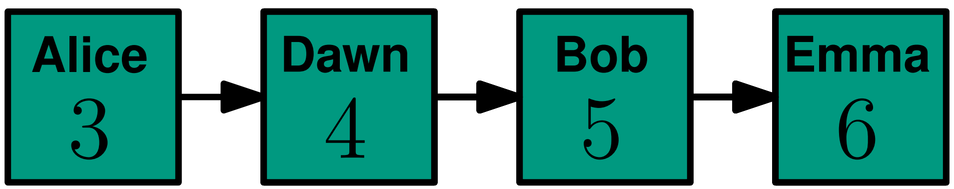
but they aren't all efficient

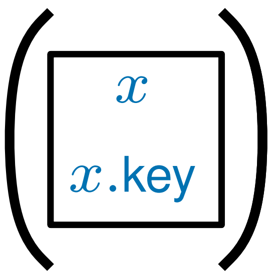
Let n denote the number of elements in the queue

- 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:





Using a Linked List as a Priority Queue

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

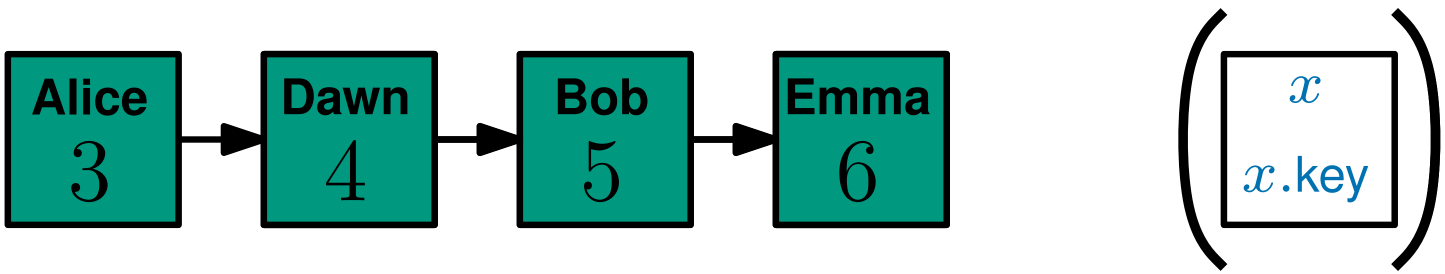
but they aren't all efficient

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. . .

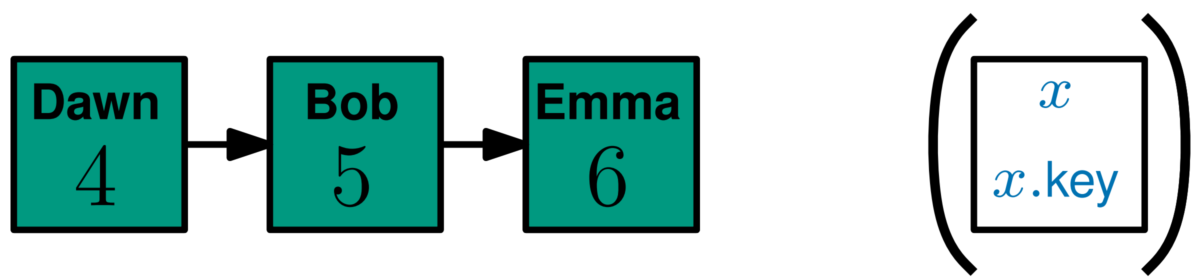
but they aren't all efficient

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:



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...

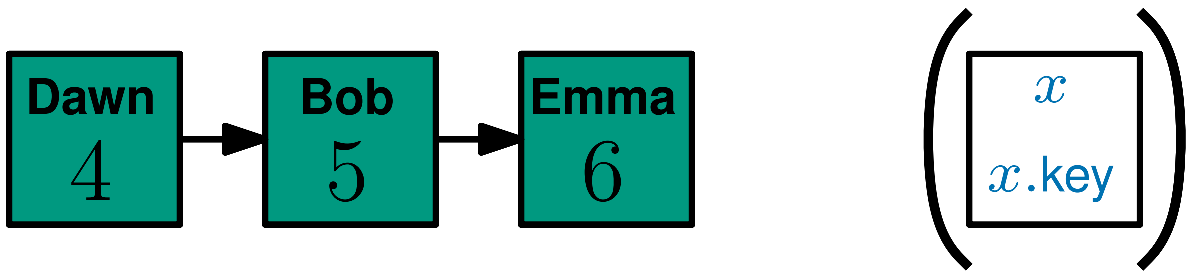
but they aren't all efficient

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:



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. . .

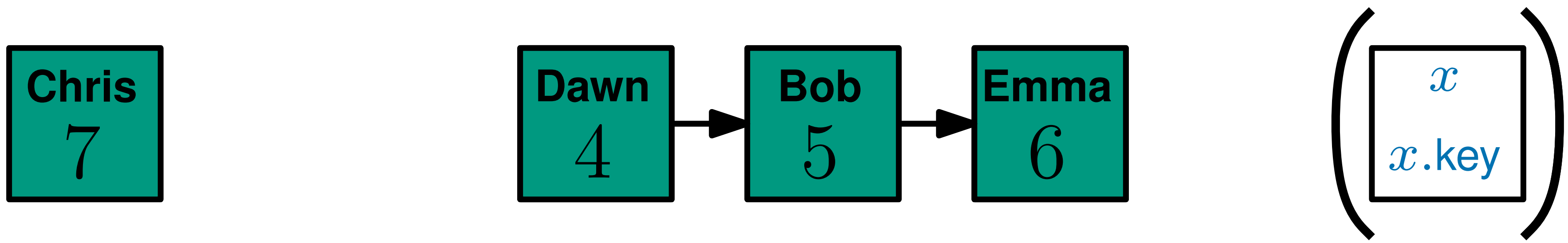
but they aren't all efficient

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...

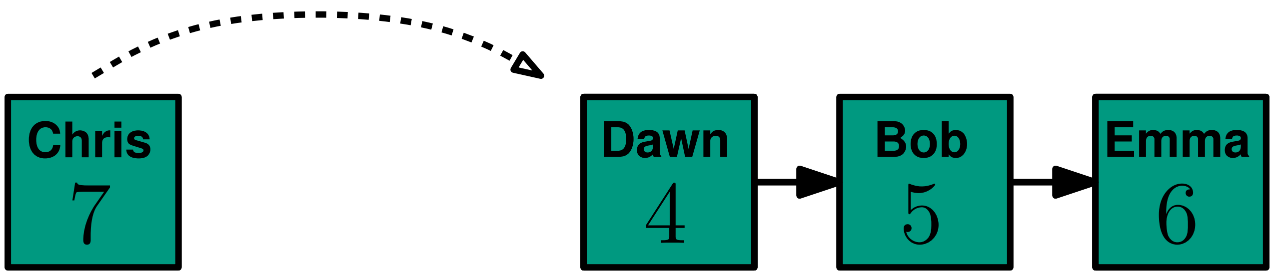
but they aren't all efficient

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:



$\left(\boxed{\begin{array}{c} x \\ x.\text{key} \end{array}}\right)$

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. . .

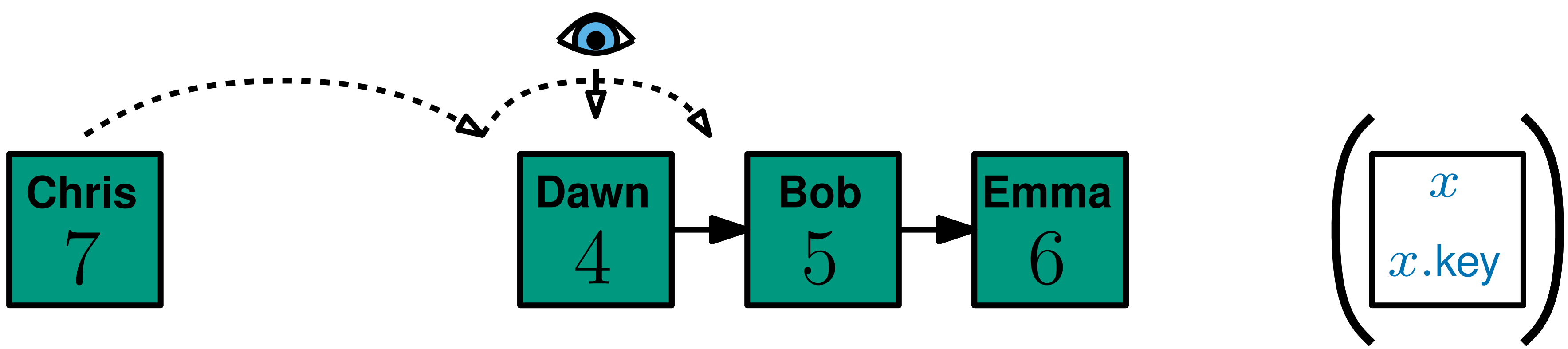
but they aren't all efficient

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...

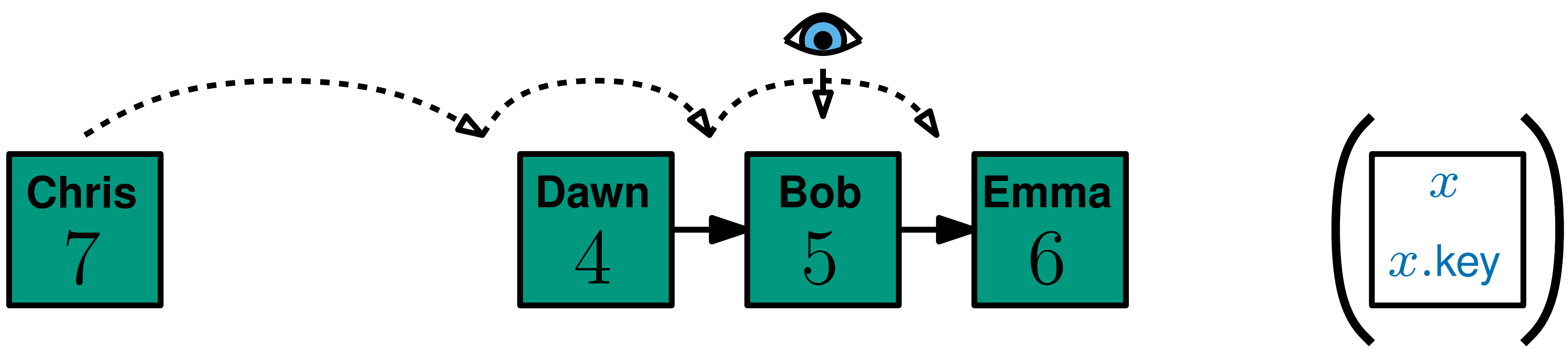
but they aren't all efficient

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:



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...

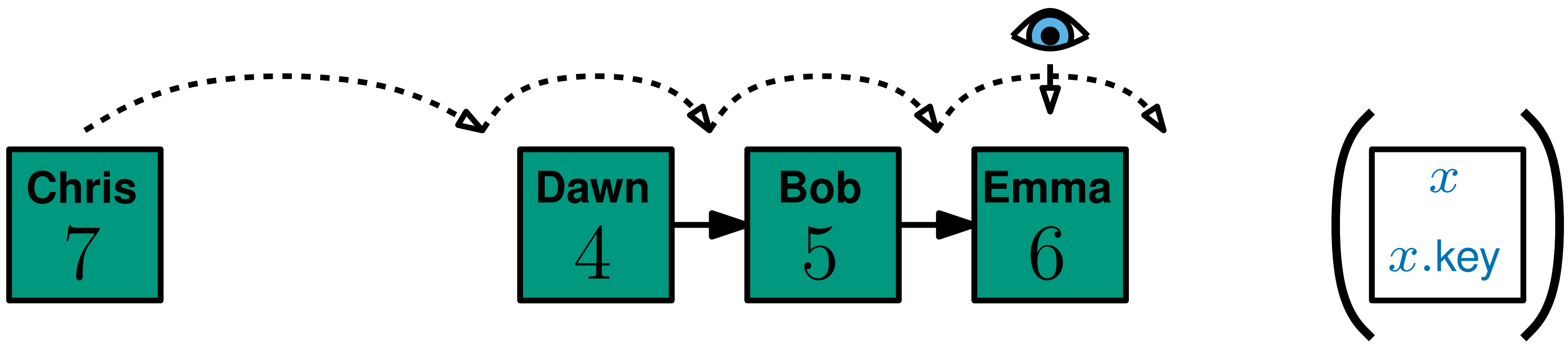
but they aren't all efficient

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...

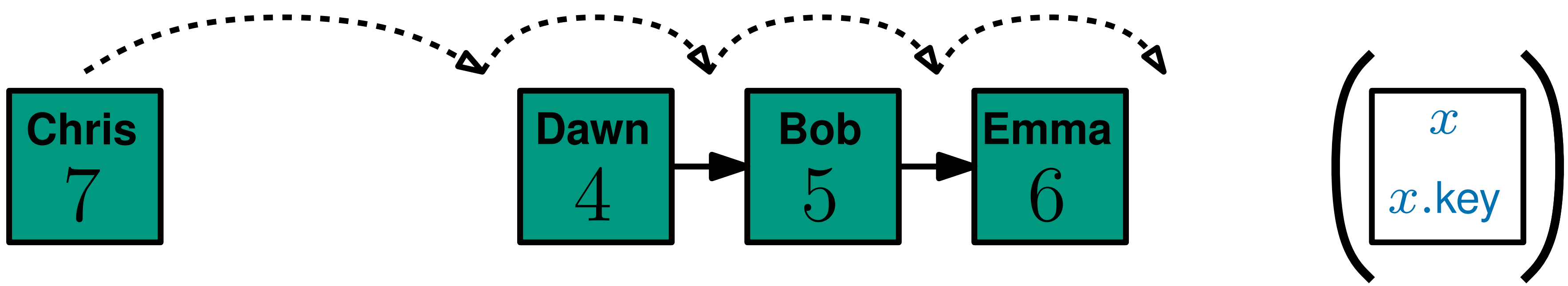
but they aren't all efficient

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:



EXTRACTMIN is very efficient,

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

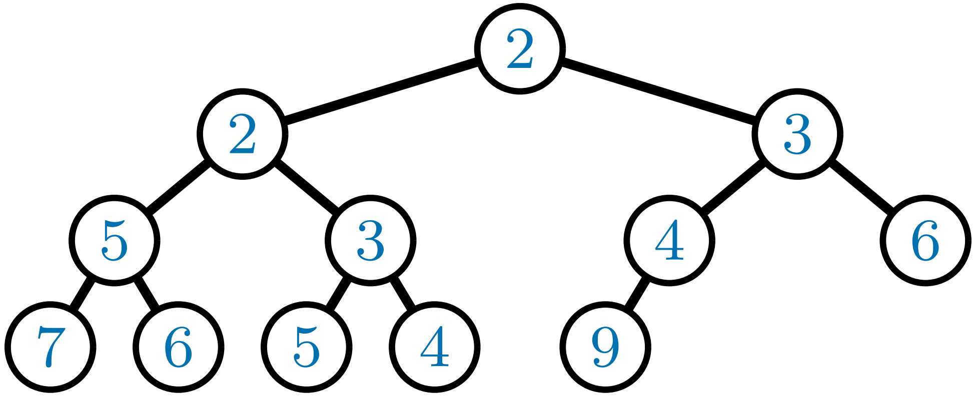
INSERT and DECREASEKEY are very *inefficient*, they take $O(n)$ time

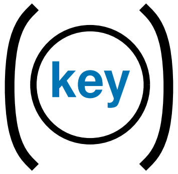
- we have to look through the entire linked list

(in the worst case)

Binary Heaps

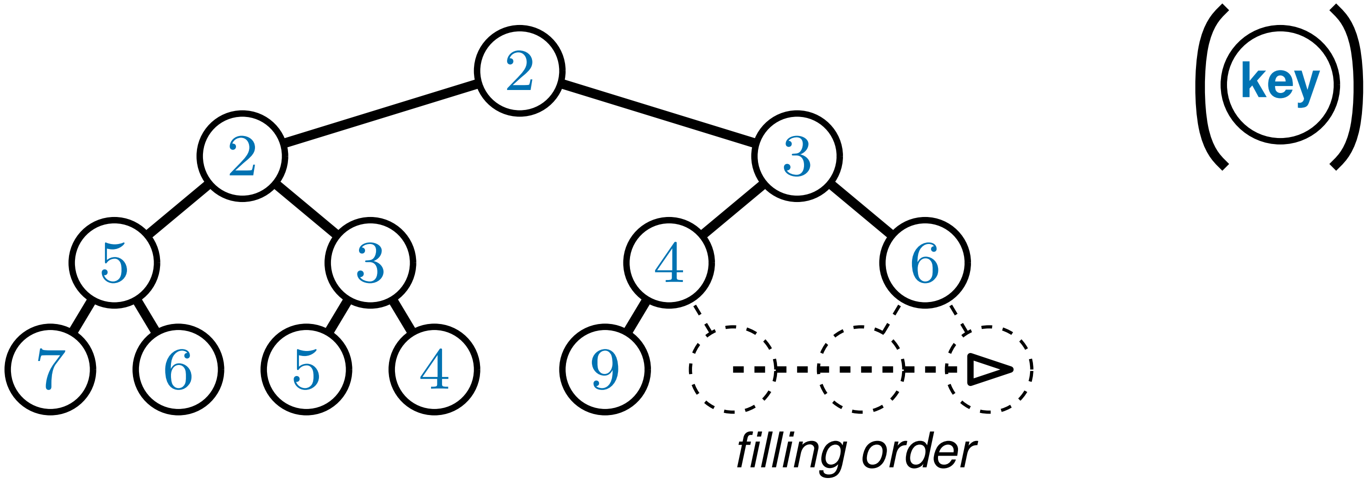
A binary heap is an 'almost complete' binary tree, where every level is full. . .
except (possibly) the lowest, which is filled from left to right





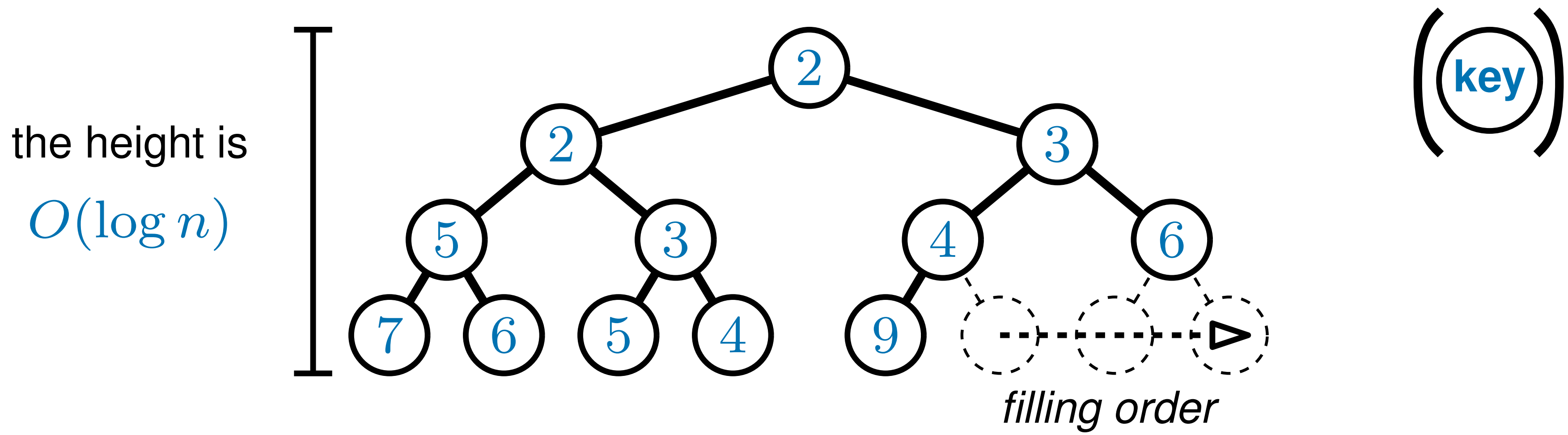
Binary Heaps

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



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

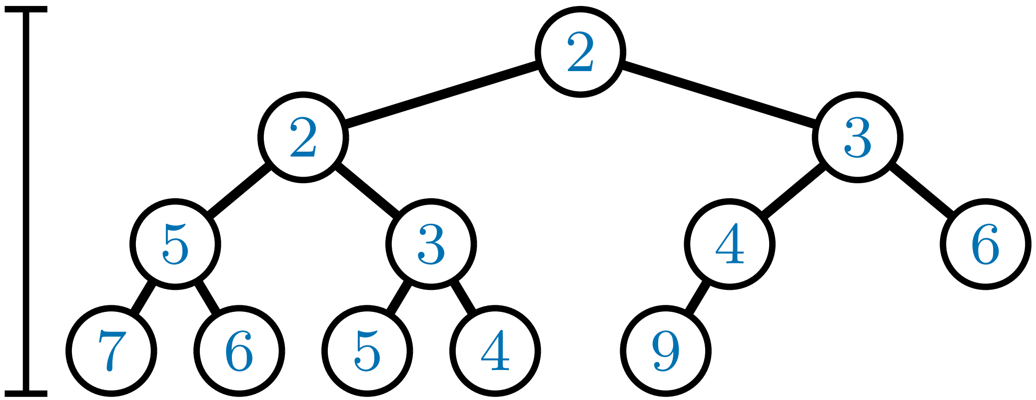


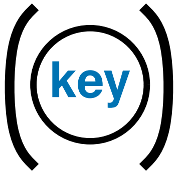
n is the number of elements in the Heap

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
$O(\log n)$



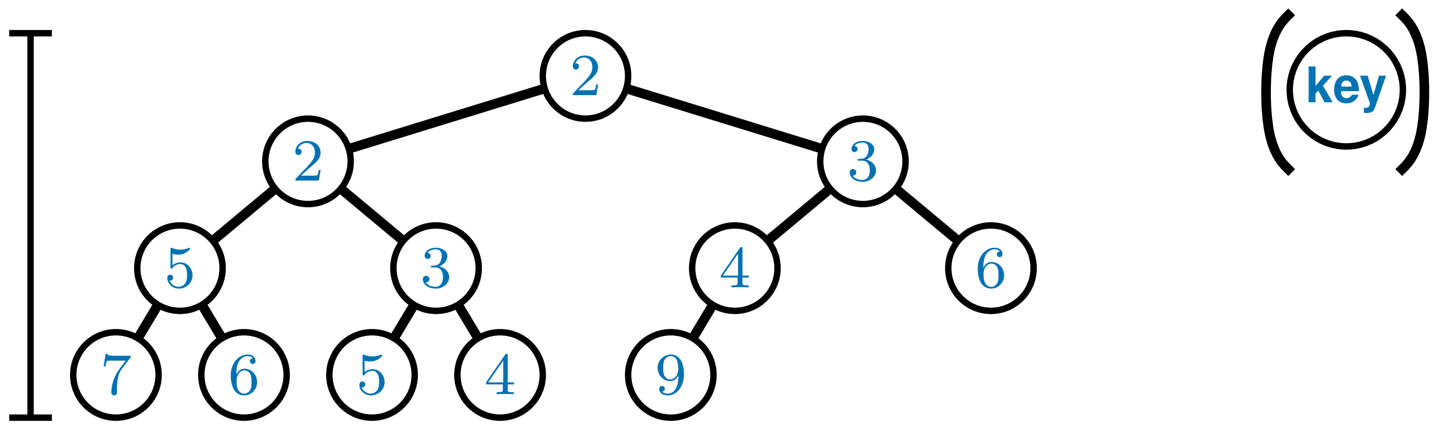


n is the number of elements in the Heap

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

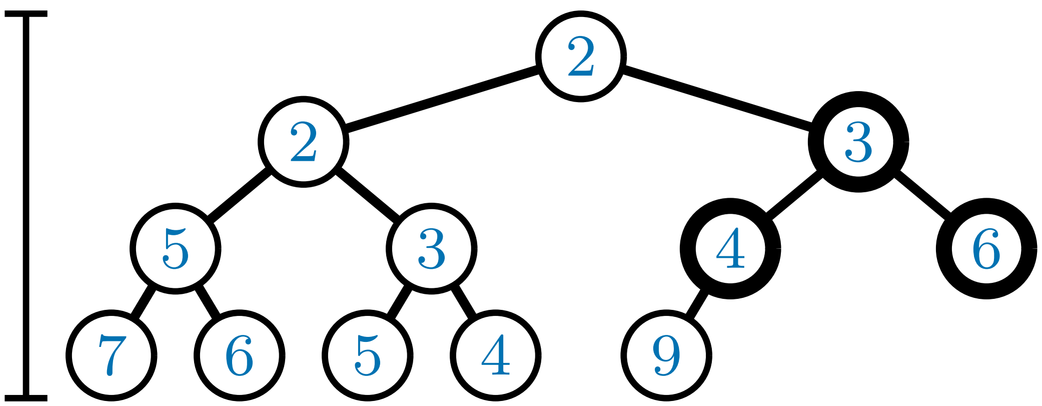
Heap Property Any element has a key *less than or equal to* the keys of its children.

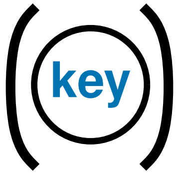
Binary Heaps

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

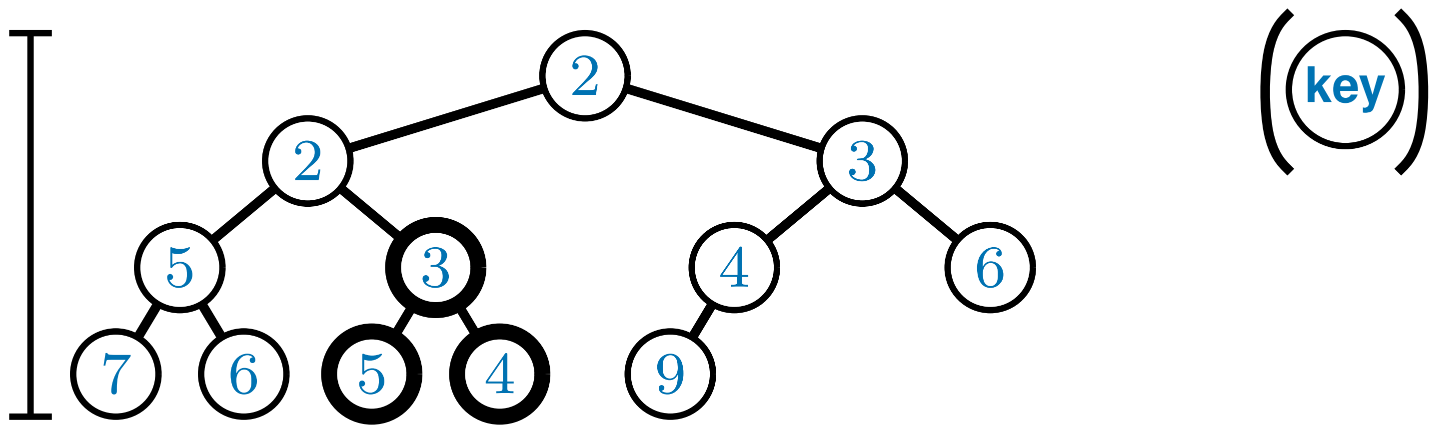
Heap Property Any element has a key *less than or equal to* the keys of its children.

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

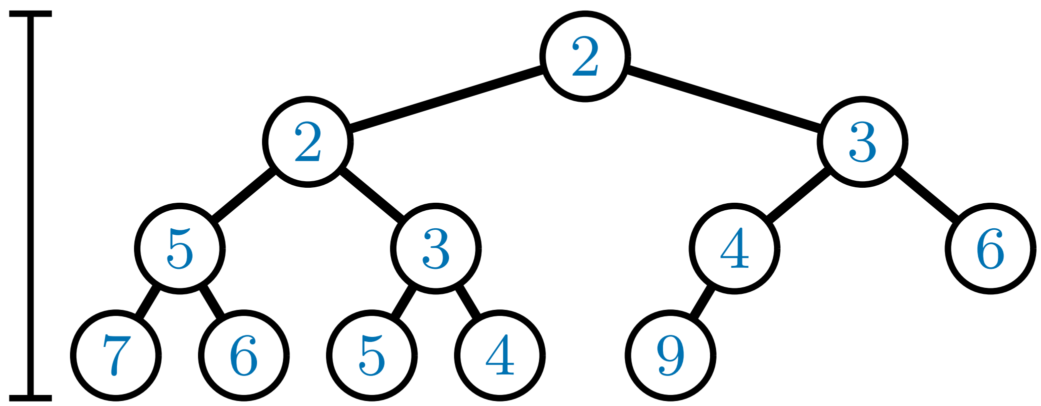
Heap Property Any element has a key *less than or equal to* the keys of its children.

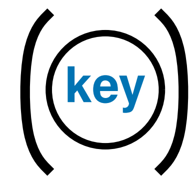
Binary Heaps

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

$O(\log n)$





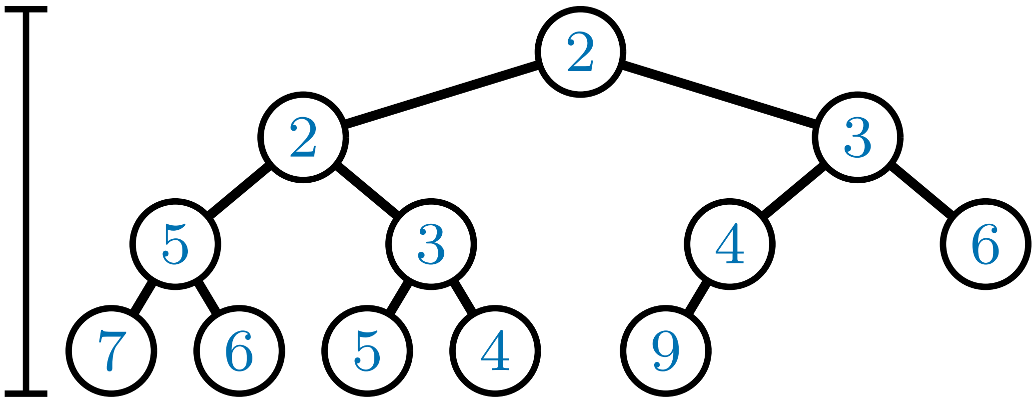
n is the number of elements in the Heap

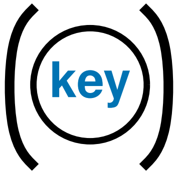
Heap Property Any element has a key *less than or equal to* the keys of its children.

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
$O(\log n)$





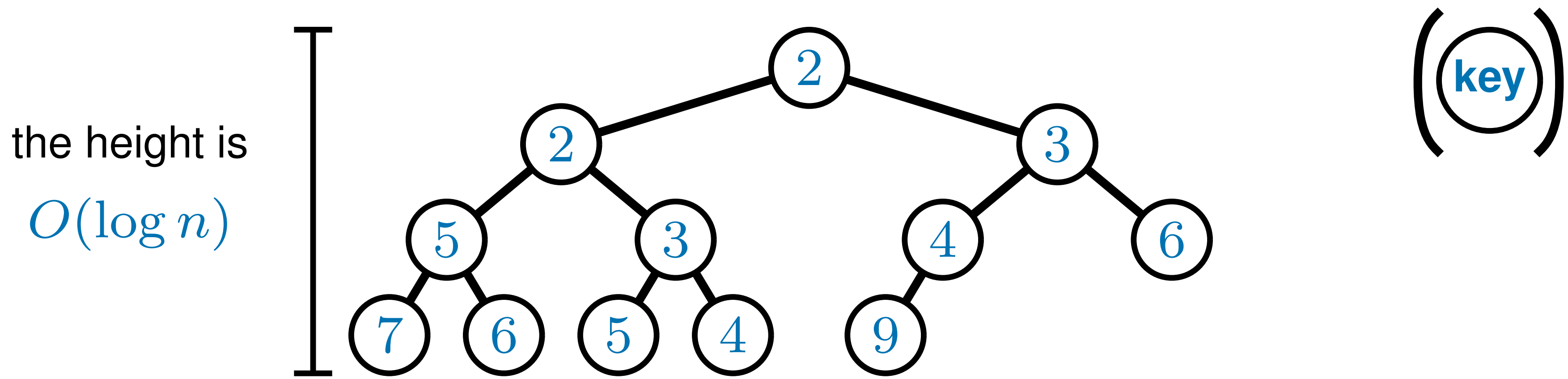
n is the number of elements in the Heap

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



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.

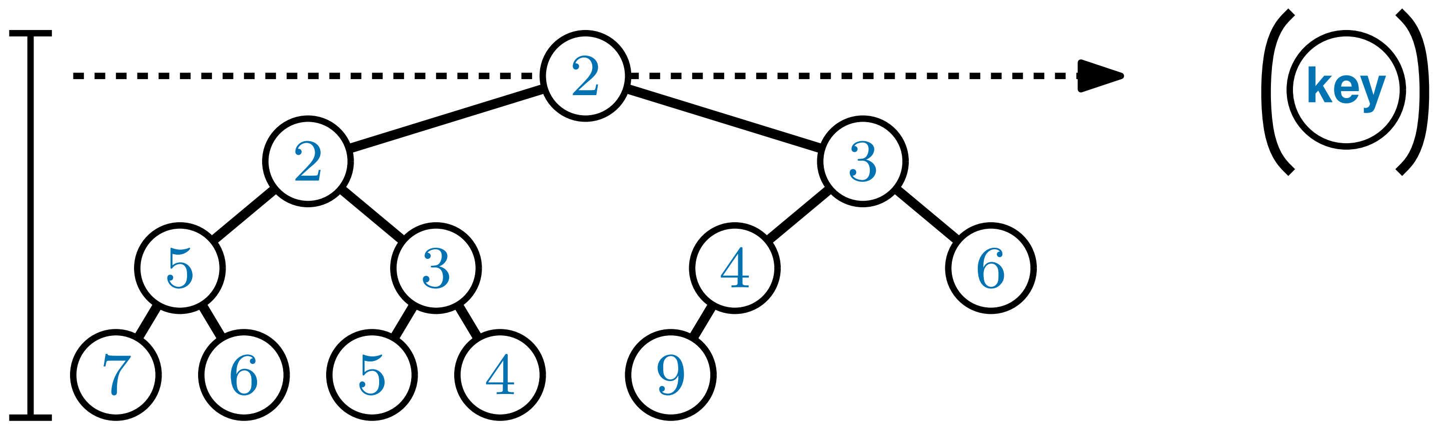
A binary heap can be efficiently stored implicitly as an array A:

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.

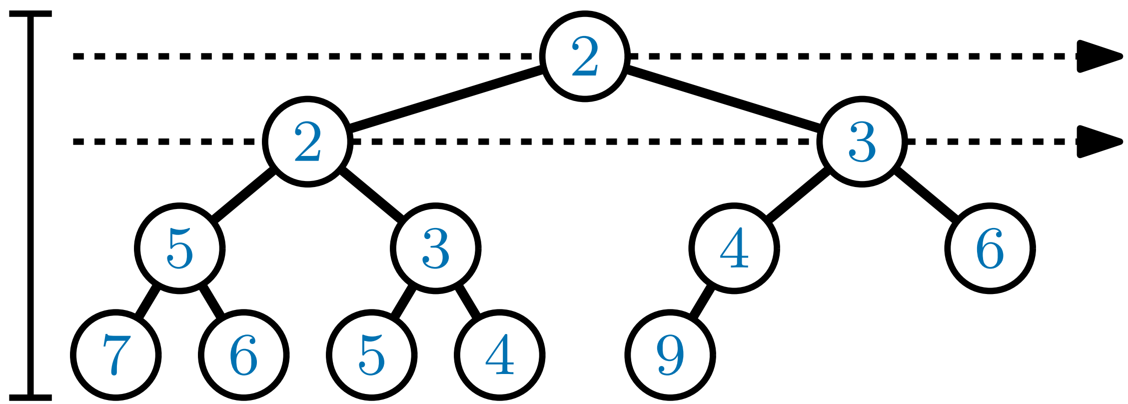
A binary heap can be efficiently stored implicitly as an array A:

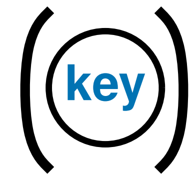
2	2	3	5	3	4	6	7	6	5	4	9

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.

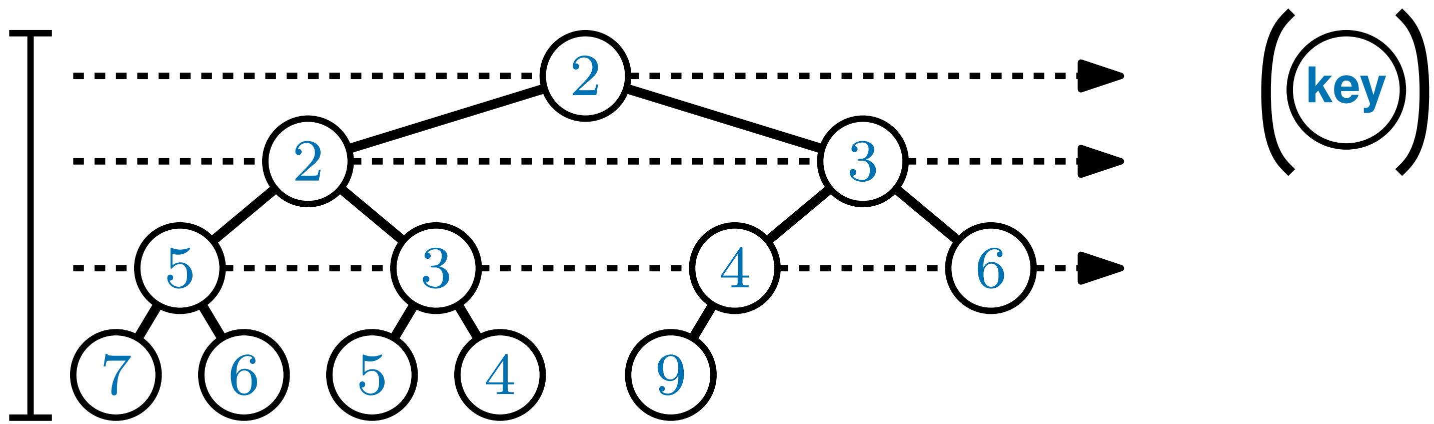
A binary heap can be efficiently stored implicitly as an array A:

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

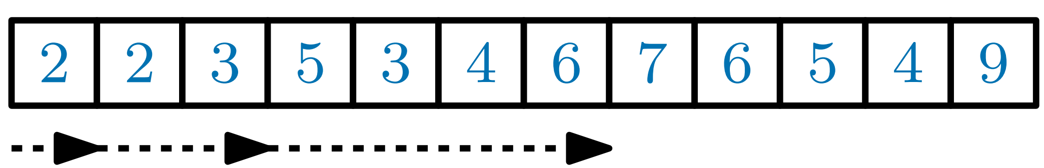
the height is
$O(\log n)$



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.

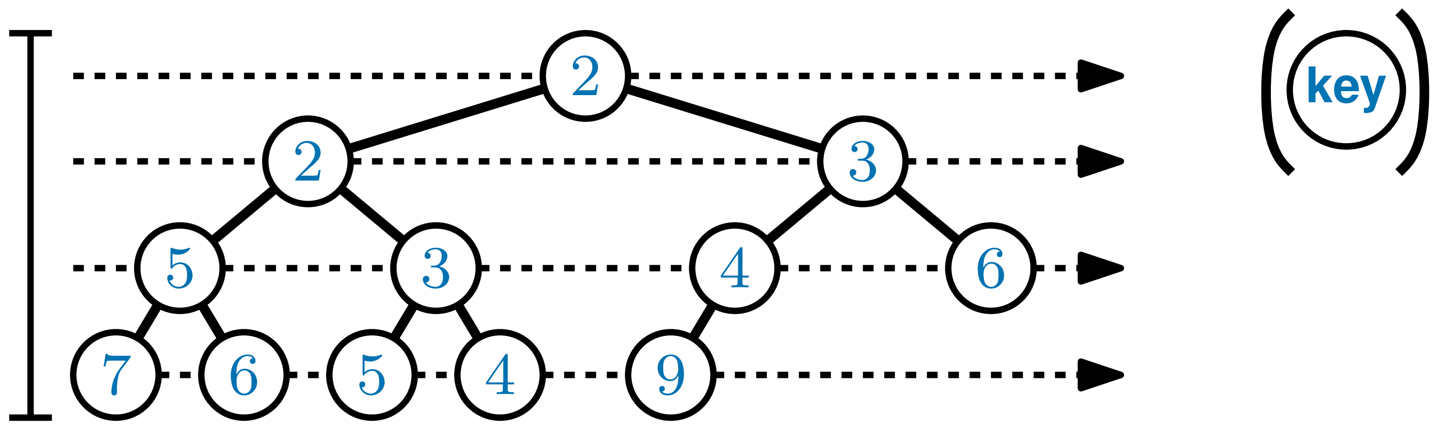
A binary heap can be efficiently stored implicitly as an array A:



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.

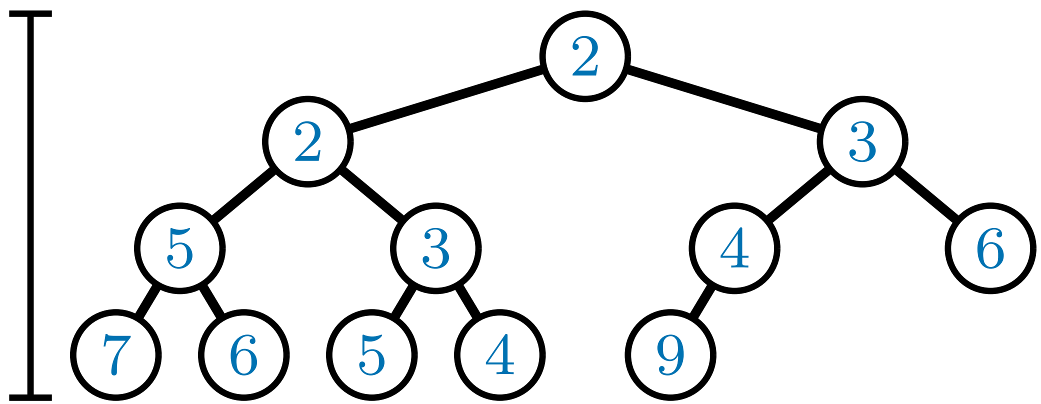
A binary heap can be efficiently stored implicitly as an array A:

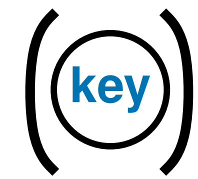
2	2	3	5	3	4	6	7	6	5	4	9

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.

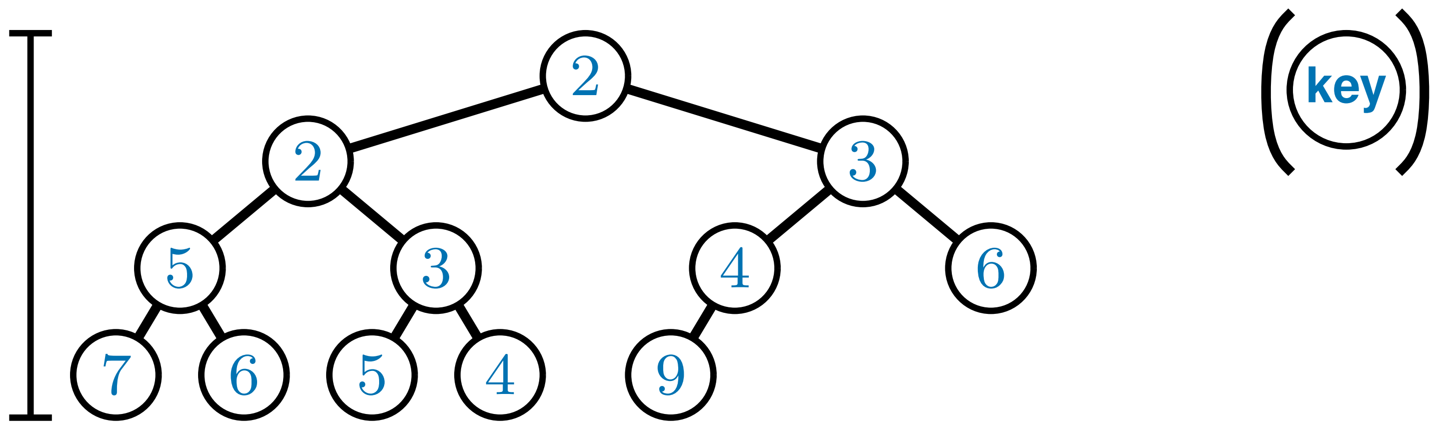
A binary heap can be efficiently stored implicitly as an array A:

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:

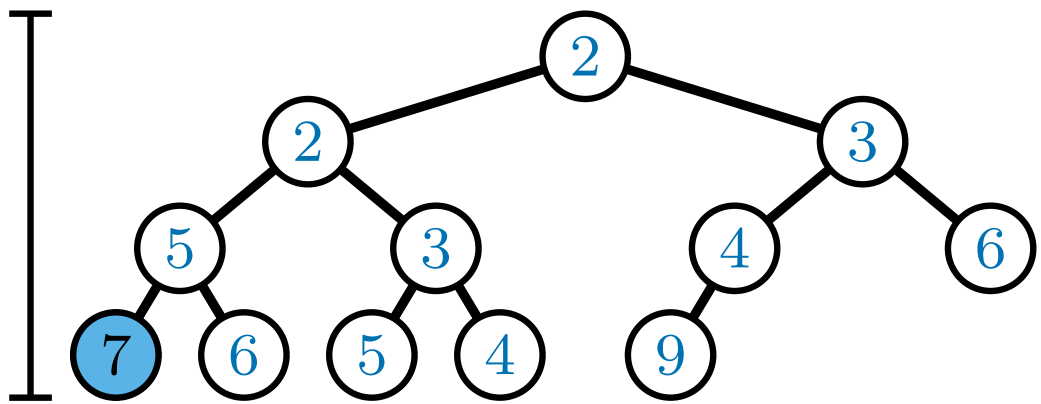
2	2	3	5	3	4	6	7	6	5	4	9

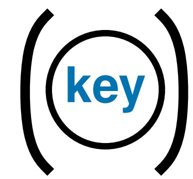
Moving around using: $\textsc{Parent}(i) = \lfloor i/2 \rfloor$ $\textsc{Left}(i) = 2i$ $\textsc{Right}(i) = 2i + 1$

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:

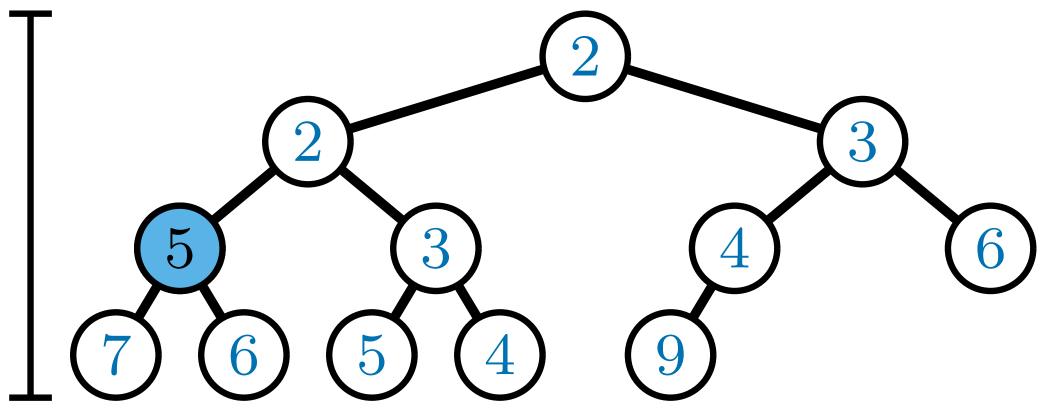
2	2	3	5	3	4	6	7	6	5	4	9

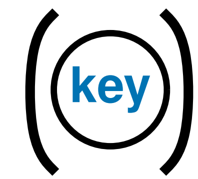
Moving around using: $\textsc{Parent}(i) = \lfloor i/2 \rfloor$ $\textsc{Left}(i) = 2i$ $\textsc{Right}(i) = 2i + 1$

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:

2	2	3	5	3	4	6	7	6	5	4	9

Moving around using: $\textsc{Parent}(i) = \lfloor i/2 \rfloor$ $\textsc{Left}(i) = 2i$ $\textsc{Right}(i) = 2i + 1$

Using a Binary Heap as a Priority Queue

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

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

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

where $k < x.key$

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

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

Using a Binary Heap as a Priority Queue

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

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

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

where $k < x.key$

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(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

Using a Binary Heap as a Priority Queue

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

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

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

where $k < x.key$

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

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

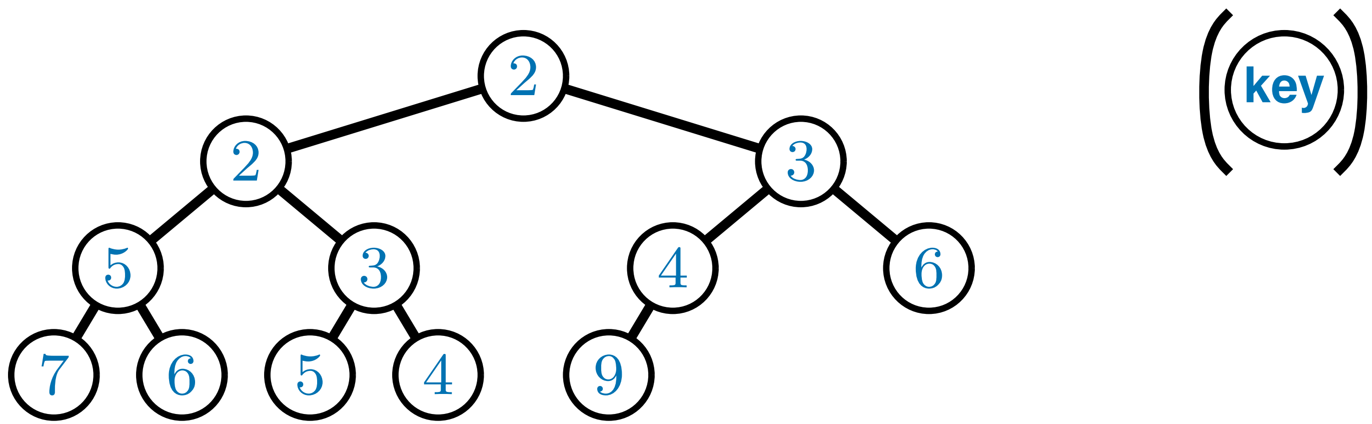
Assumption we can find the location of any element x in the Heap in $O(1)$ time

This is a little fiddly. . . we'll come back to it at the end of the lecture

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 $k \leqslant x$.key, *otherwise raise an error*

Step 3: Set x.key $= k$

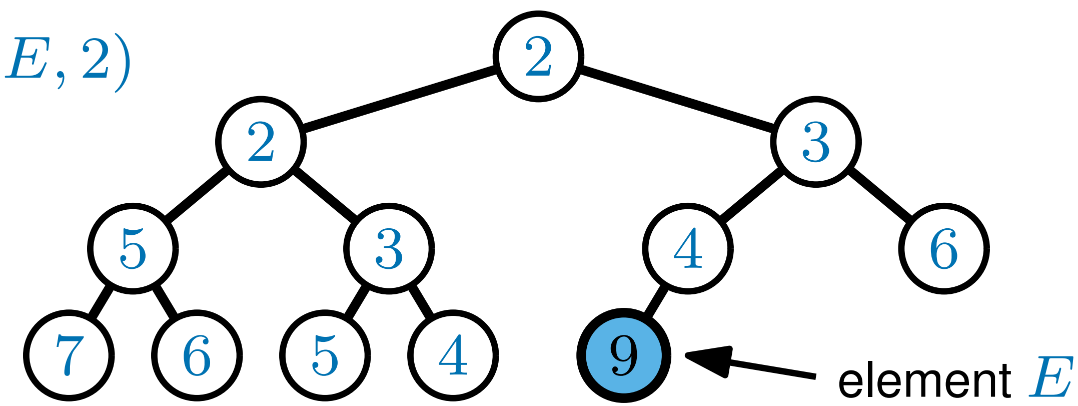
Step 4: While x.key is smaller than its parent's: *(stop if x becomes the root)*
swap x with its parent

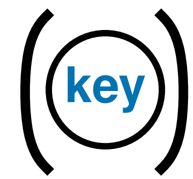
DecreaseKey with a Binary Heap

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 $k \leqslant x$.key, *otherwise raise an error*

Step 3: Set x.key $= k$

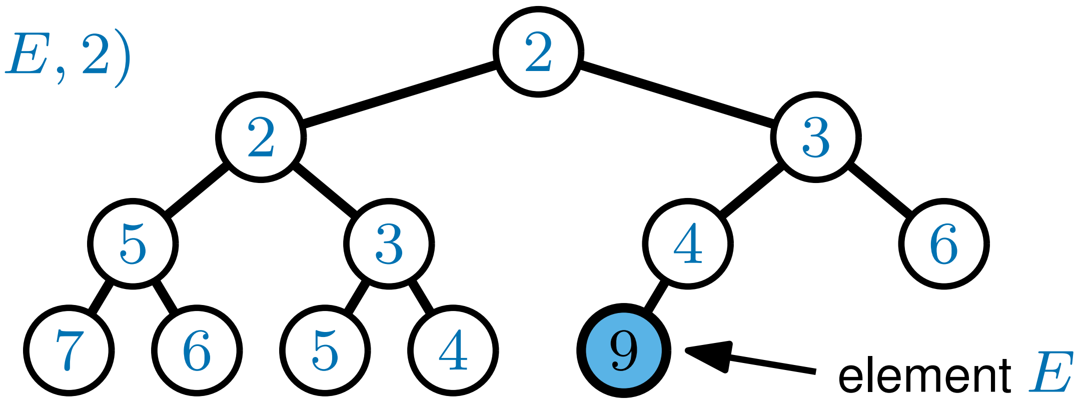
Step 4: While x.key is smaller than its parent's: *(stop if x becomes the root)*
swap x with its parent

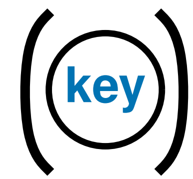
DecreaseKey with a Binary Heap

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 $k \leqslant x$.key, *otherwise raise an error*

Step 3: Set x.key $= k$

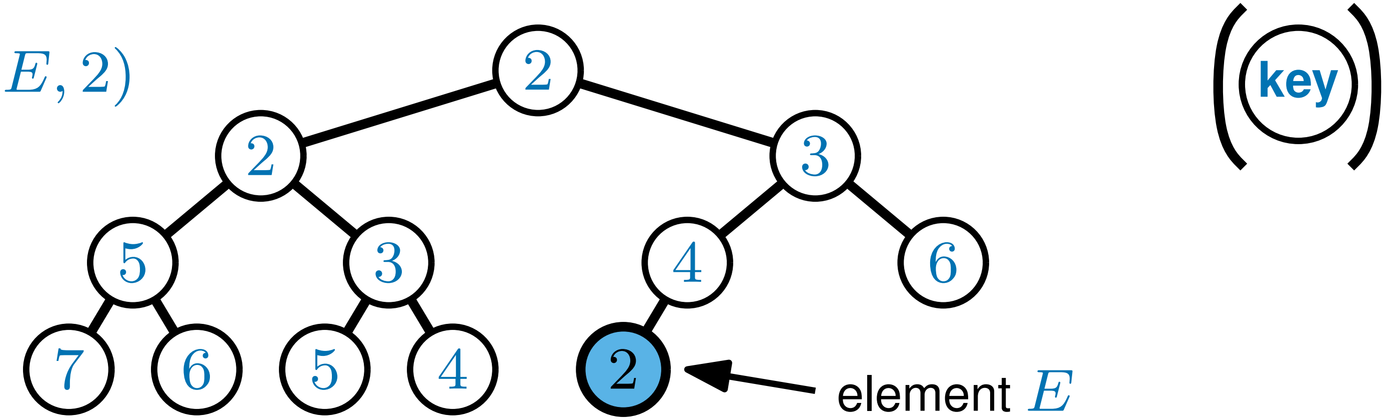
Step 4: While x.key is smaller than its parent's: *(stop if x becomes the root)*
swap x with its parent

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 $k \leqslant x$.key, *otherwise raise an error*

Step 3: Set x.key $= k$

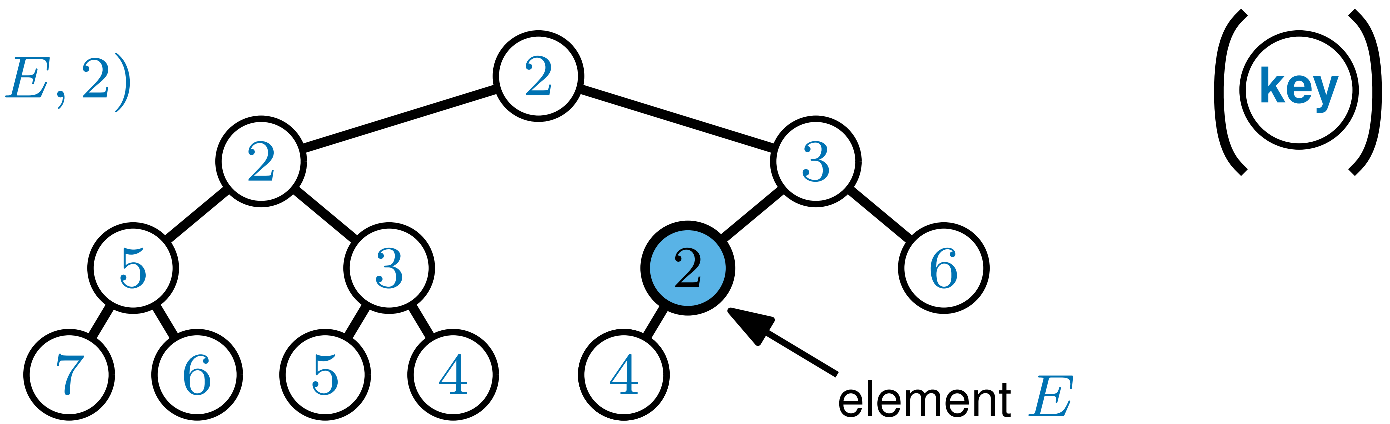
Step 4: While x.key is smaller than its parent's: *(stop if x becomes the root)*
swap x with its parent

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 $k \leqslant x$.key, *otherwise raise an error*

Step 3: Set x.key $= k$

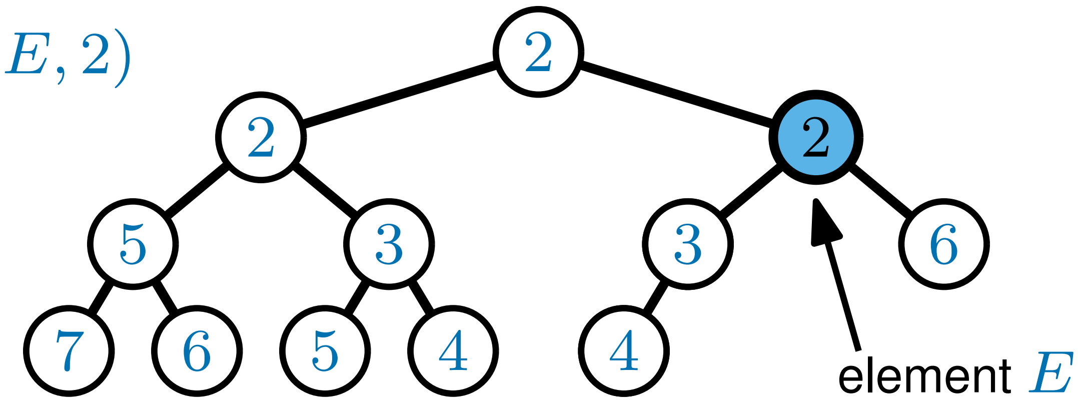
Step 4: While x.key is smaller than its parent's: *(stop if x becomes the root)*
swap x with its parent

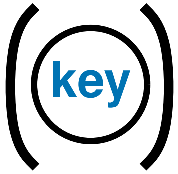
DecreaseKey with a Binary Heap

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 $k \leqslant x$.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

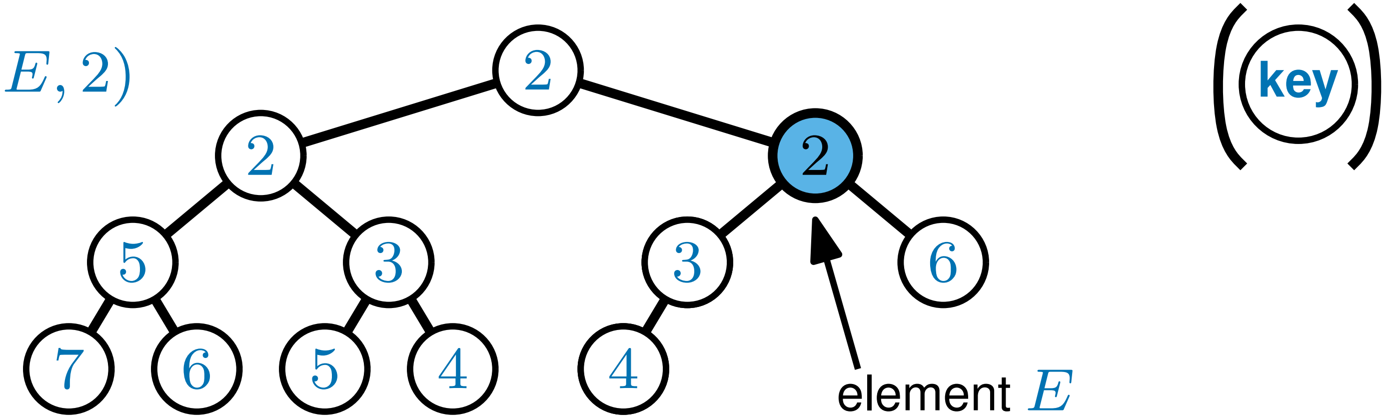
DecreaseKey with a Binary Heap

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 $k \leqslant x$.key, *otherwise raise an error*

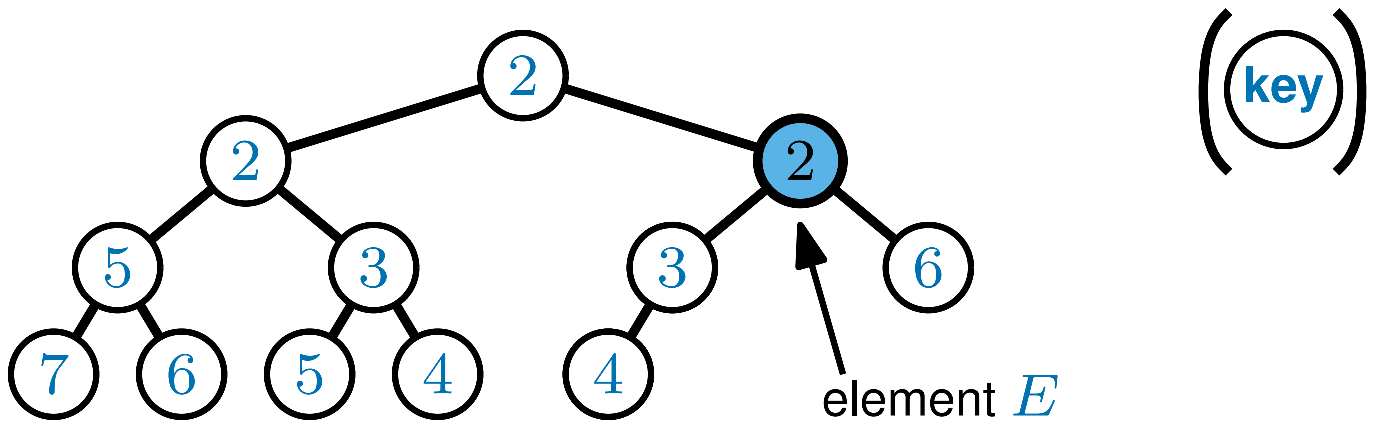
Step 3: Set x.key $= k$

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 $k \leqslant x$.key, *otherwise raise an error*

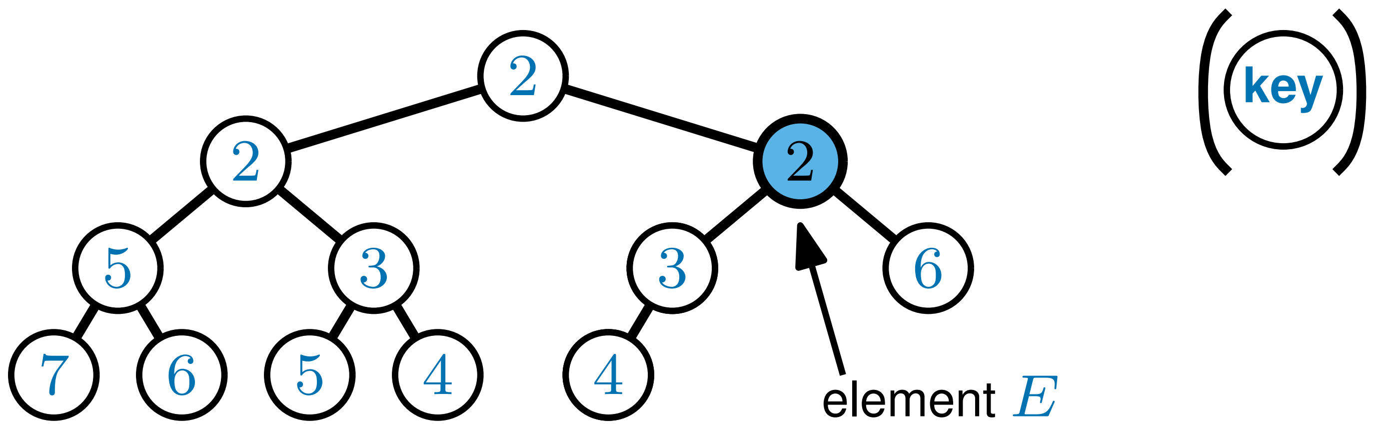
Step 3: Set x.key $= k$

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 ← $O(1)$ time

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

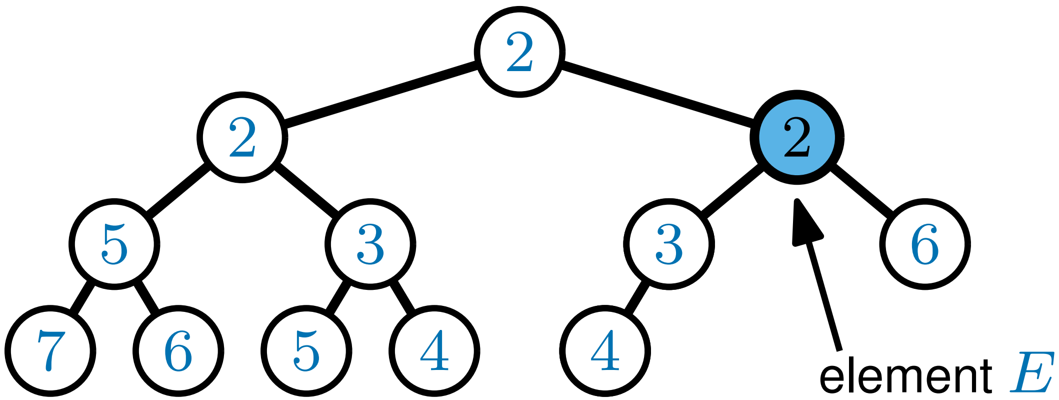
Step 3: Set x.key $= k$

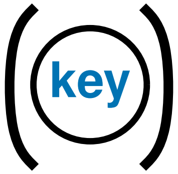
Step 4: While x.key is smaller than its parent's: *(stop if x becomes the root)*
swap x with its parent

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 ← $O(1)$ time

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

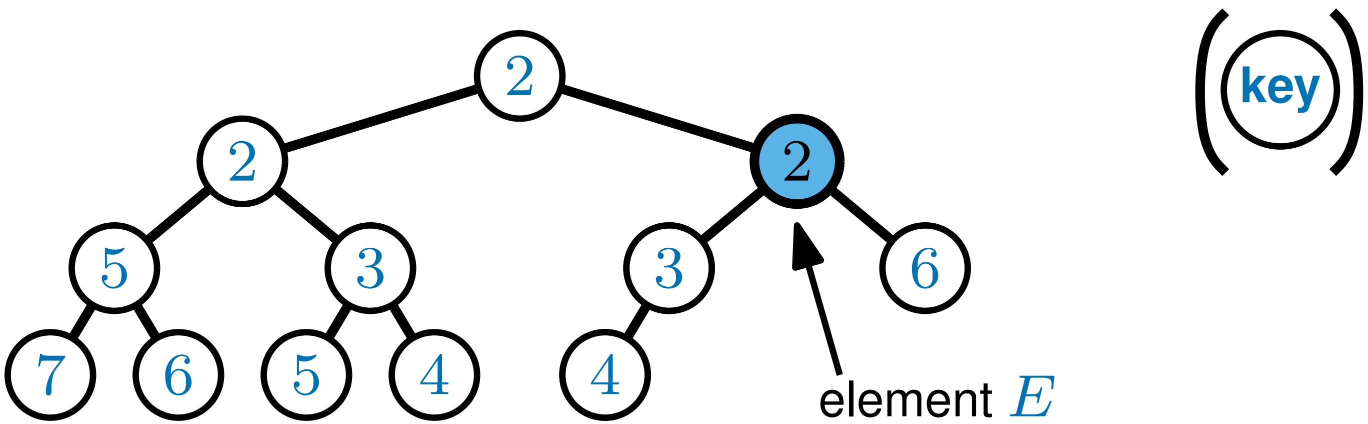
Step 3: Set x.key $= k$

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 ← $O(1)$ time

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

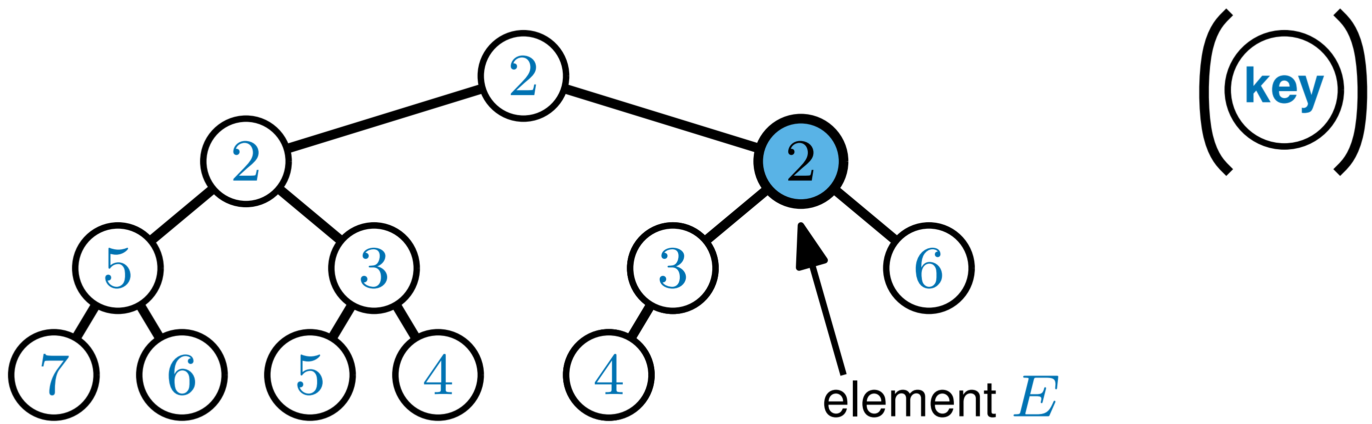
Step 3: Set x.key $= k$ ← $O(1)$ time

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

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 $k \leqslant x$.key, *otherwise raise an error*

Step 3: Set x.key $= k$ ← $O(1)$ time

Each swap takes $O(1)$ time

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

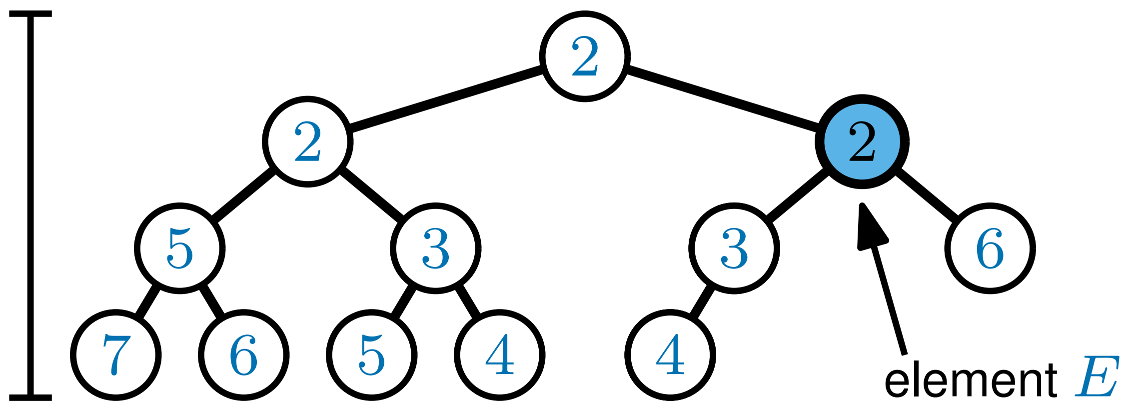
DecreaseKey with a Binary Heap

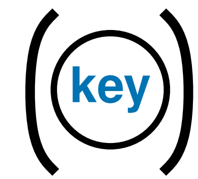
DecreaseKey(x, k) - decreases the value of x.key to k

where $k < x.key$

the height is

$O(\log n)$





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

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

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

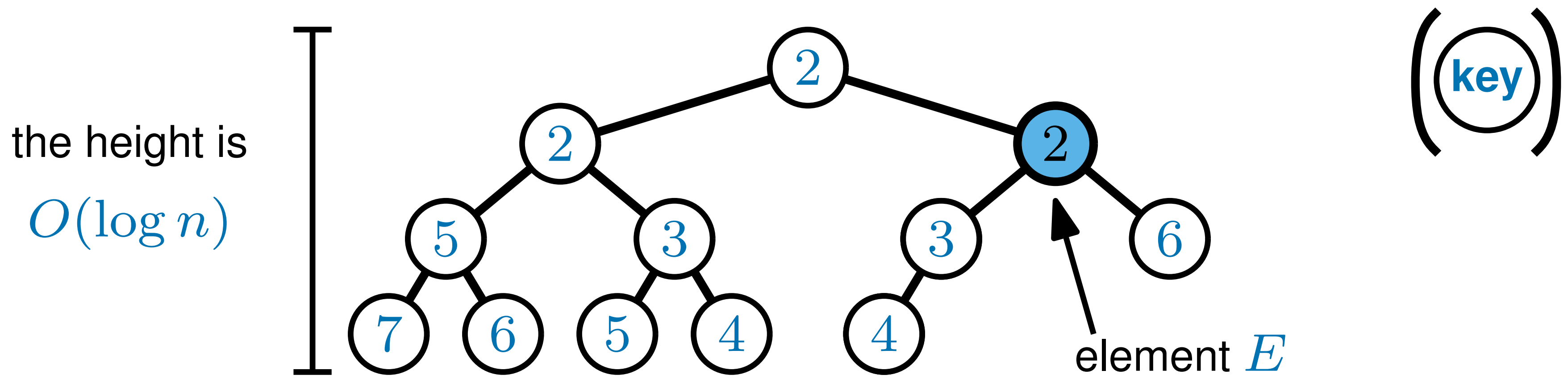
Each swap takes $O(1)$ time

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

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 $k \leqslant x$.key, *otherwise raise an error*

Step 3: Set x.key $= k$ ← $O(1)$ time

Each swap takes $O(1)$ time

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

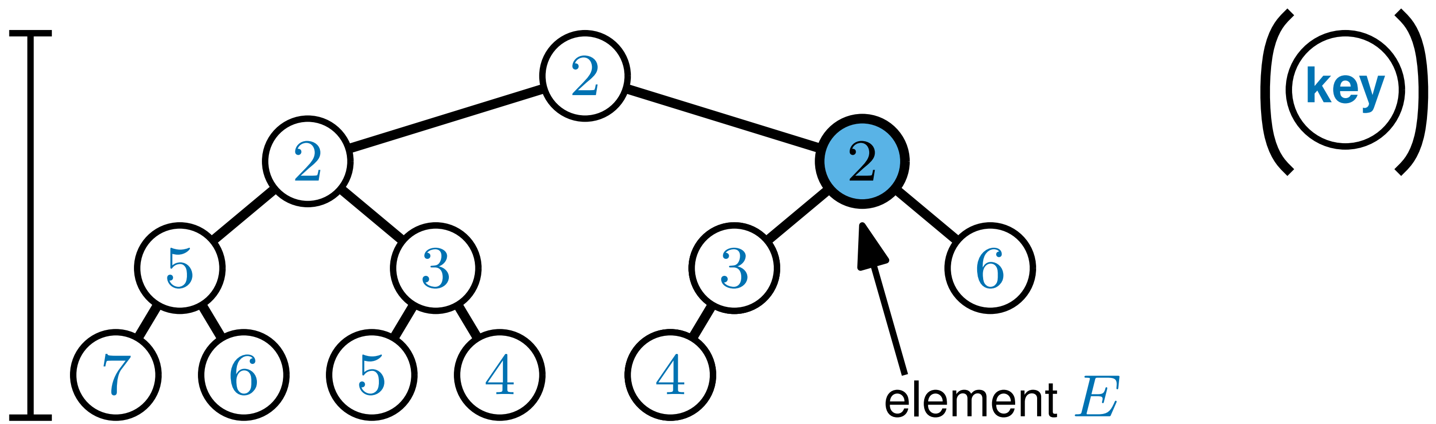
The height of the tree is $O(\log n)$ *so there are* $O(\log n)$ *swaps*

DecreaseKey with a Binary Heap

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

where $k < x.key$

the height is $O(\log n)$



Step 1: Find element x

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

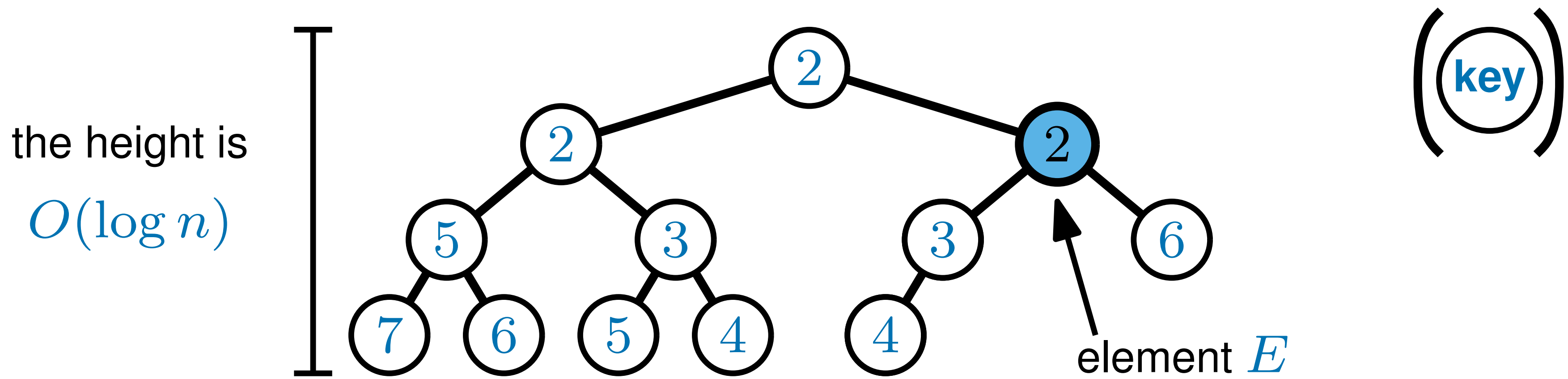
Step 3: Set x.key $= k$

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 $k \leqslant x$.key, *otherwise raise an error*

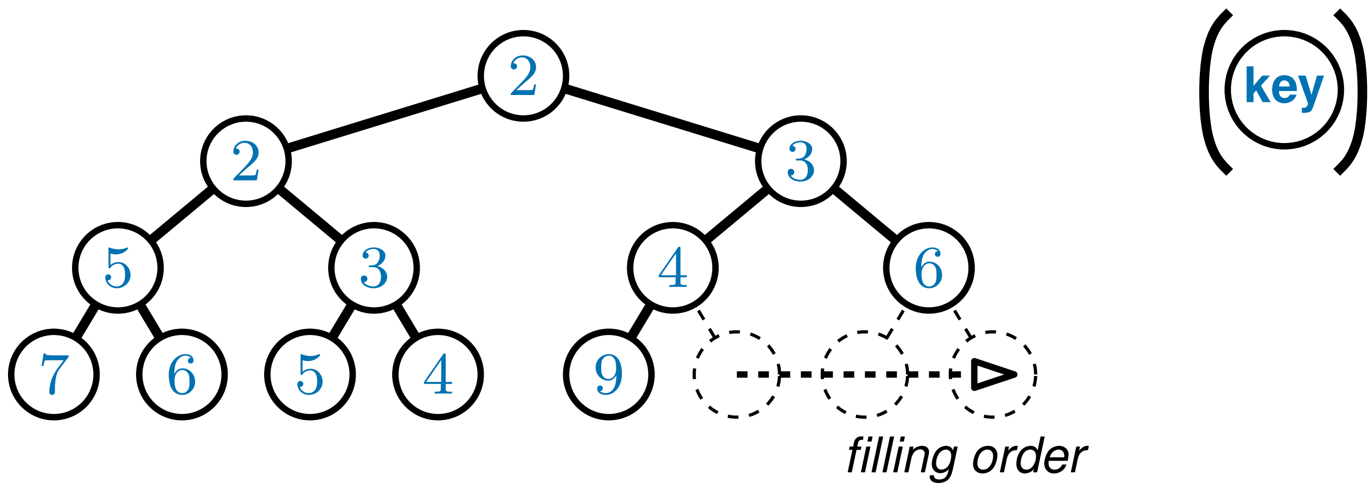
Step 3: Set x.key $= k$

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

INSERT with a Binary Heap

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



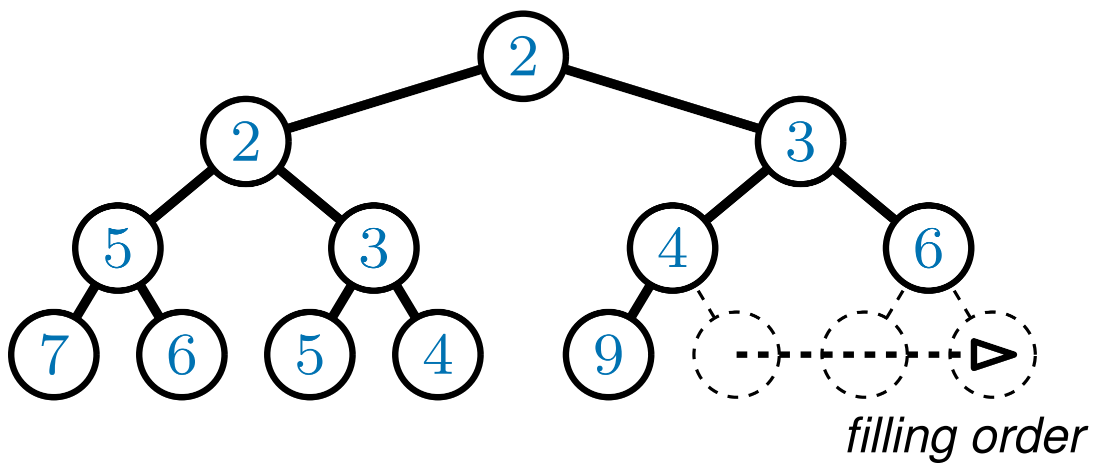
Step 1: Put element x in the next free slot

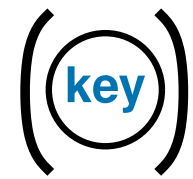
Step 2: Run DECREASEKEY(x,k).

INSERT with a Binary Heap

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

INSERT$(F, 1)$





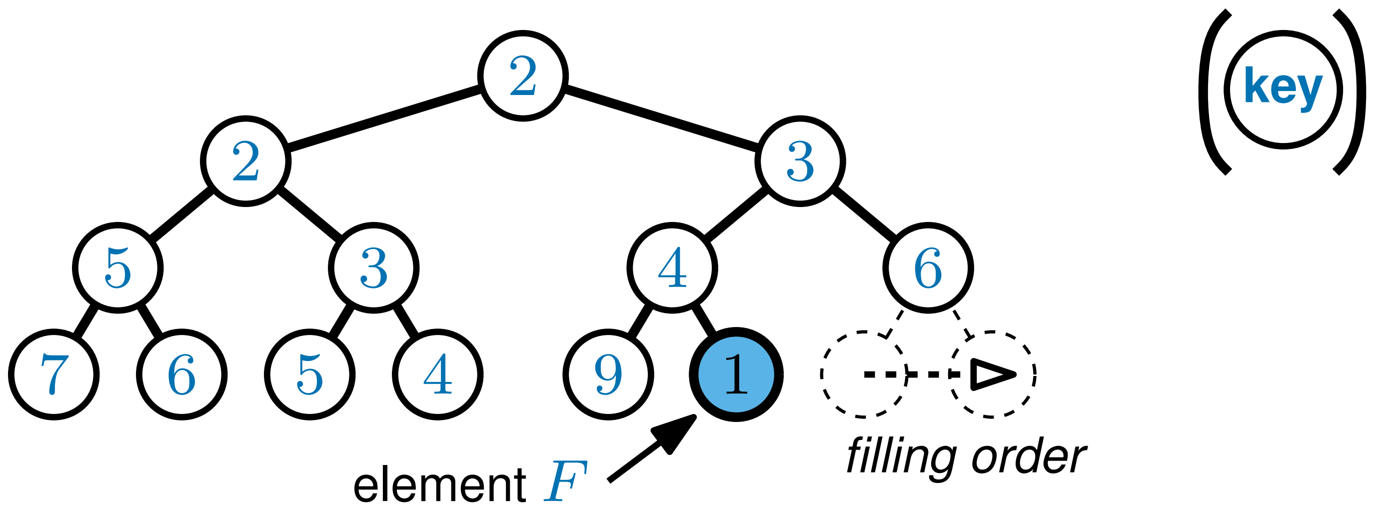
Step 1: Put element x in the next free slot

Step 2: Run DECREASEKEY(x,k).

Insert with a Binary Heap

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

Insert$(F, 1)$



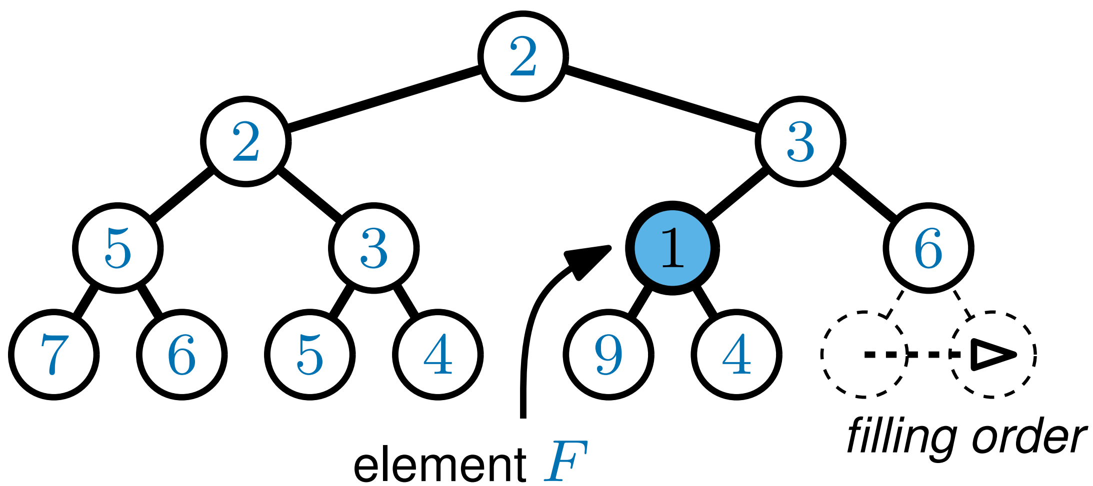
Step 1: Put element x in the next free slot

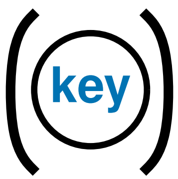
Step 2: Run DecreaseKey(x,k).

INSERT with a Binary Heap

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

INSERT$(F, 1)$





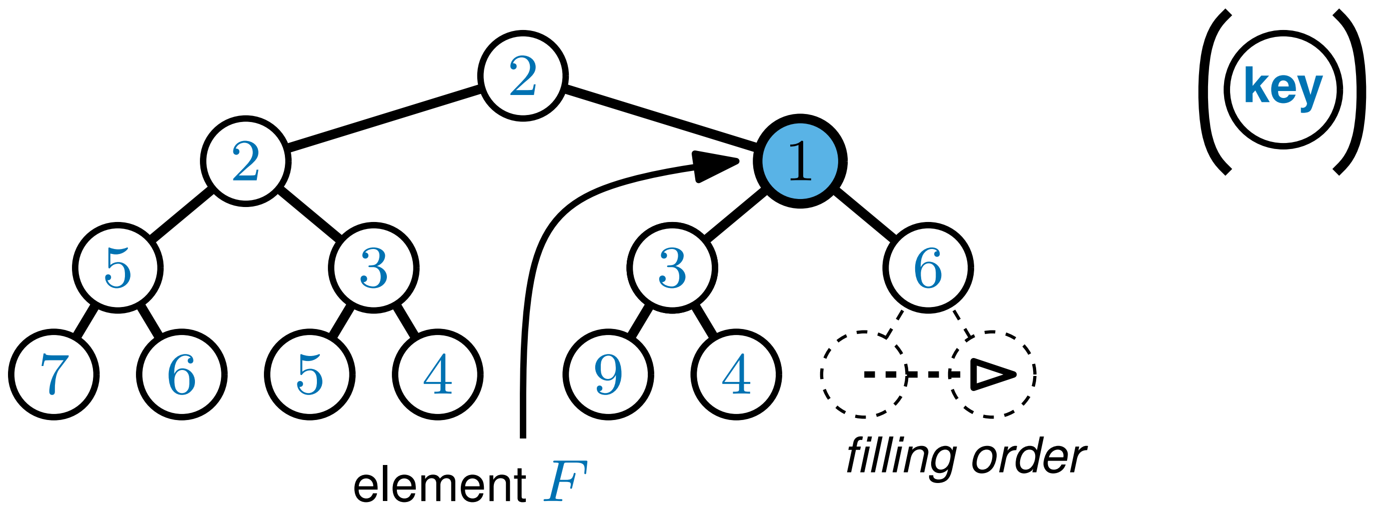
Step 1: Put element x in the next free slot

Step 2: Run DECREASEKEY(x,k).

INSERT with a Binary Heap

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

INSERT$(F, 1)$



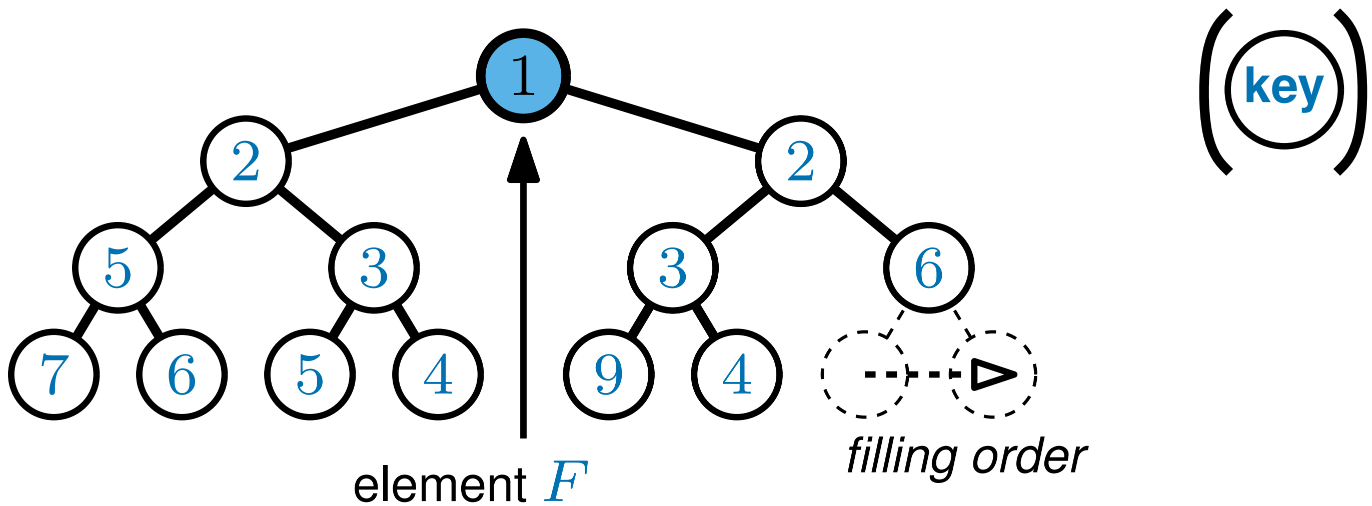
Step 1: Put element x in the next free slot

Step 2: Run DECREASEKEY(x,k).

INSERT with a Binary Heap

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

INSERT$(F, 1)$



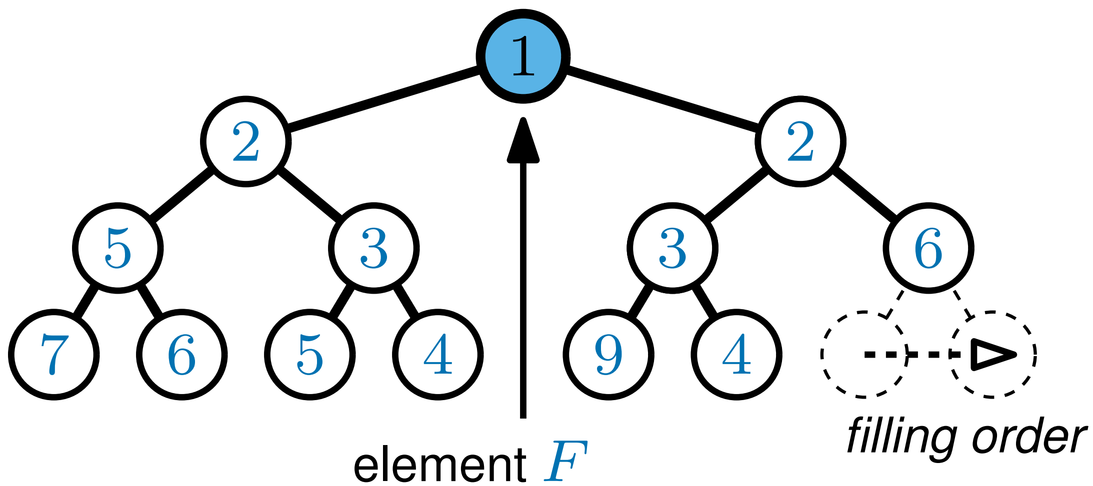
Step 1: Put element x in the next free slot

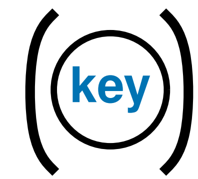
Step 2: Run DECREASEKEY(x,k).

Insert with a Binary Heap

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

Insert$(F, 1)$





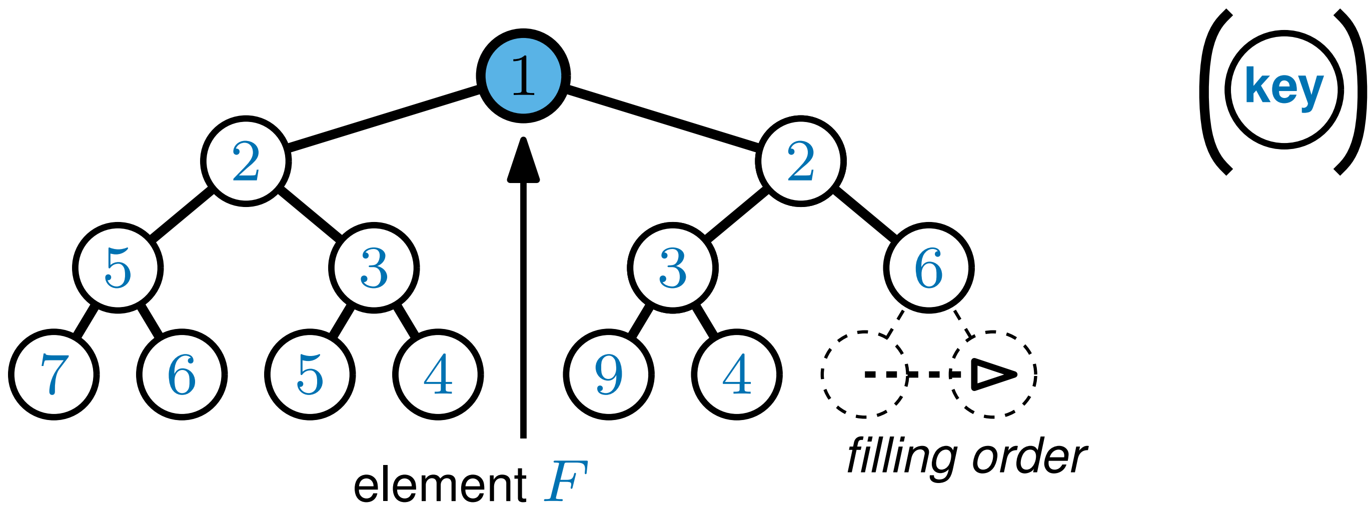
Step 1: Put element x in the next free slot ← $O(1)$ time

Step 2: Run DecreaseKey(x,k).

INSERT with a Binary Heap

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

INSERT$(F, 1)$



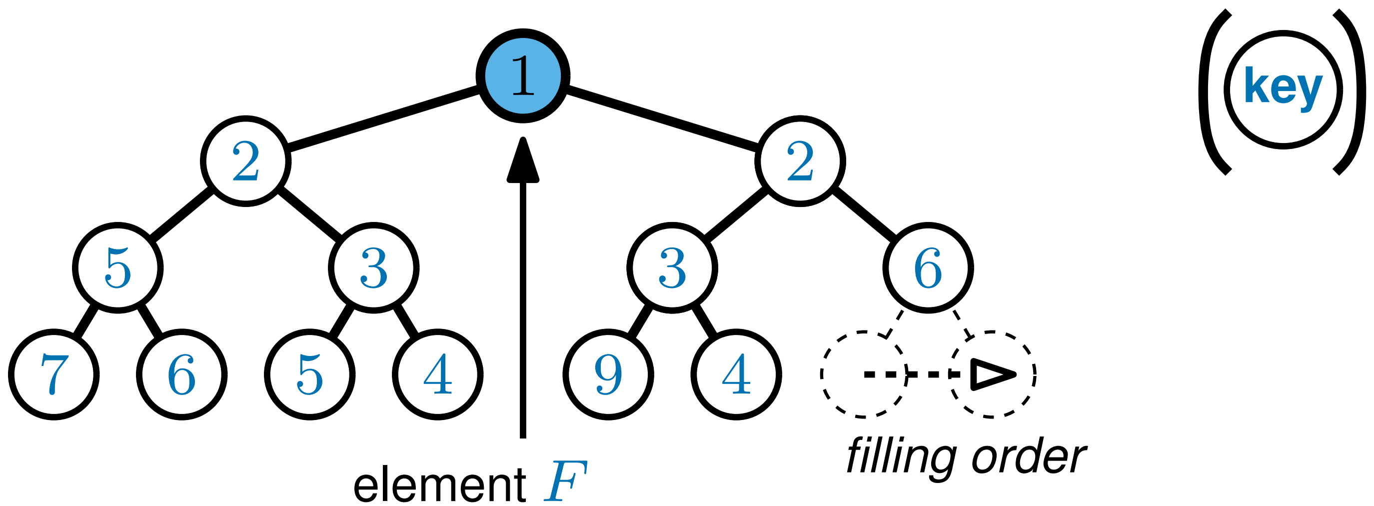
Step 1: Put element x in the next free slot ← $O(1)$ time

Step 2: Run DECREASEKEY(x,k). ← $O(\log n)$ time

$\textsc{Insert}$ with a Binary Heap

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

$\textsc{Insert}(F, 1)$



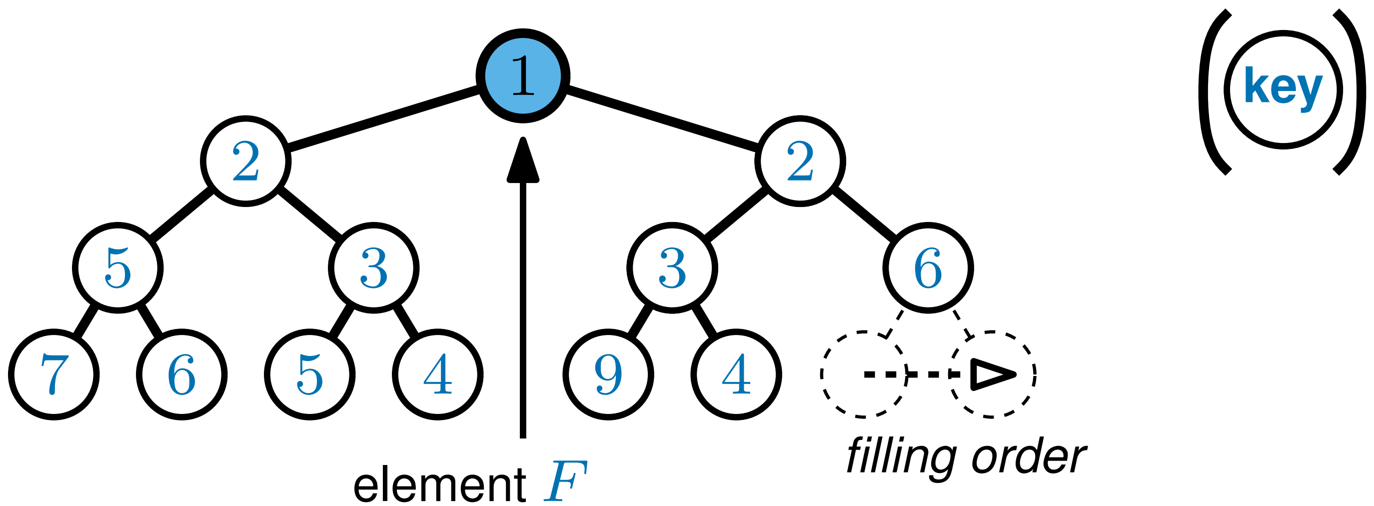
Step 1: Put element x in the next free slot ← $O(1)$ time

Step 2: Run $\textsc{DecreaseKey}$(x,k). ← $O(\log n)$ time

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

INSERT with a Binary Heap

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



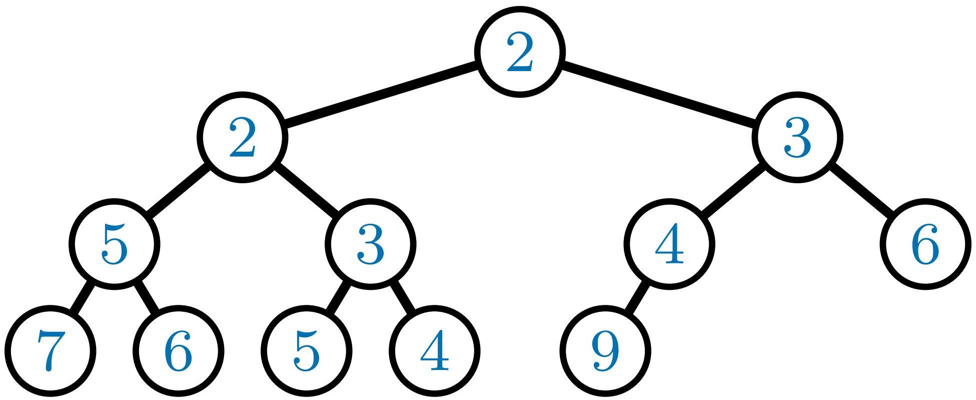
Step 1: Put element x in the next free slot ← $O(1)$ time

Step 2: Run 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

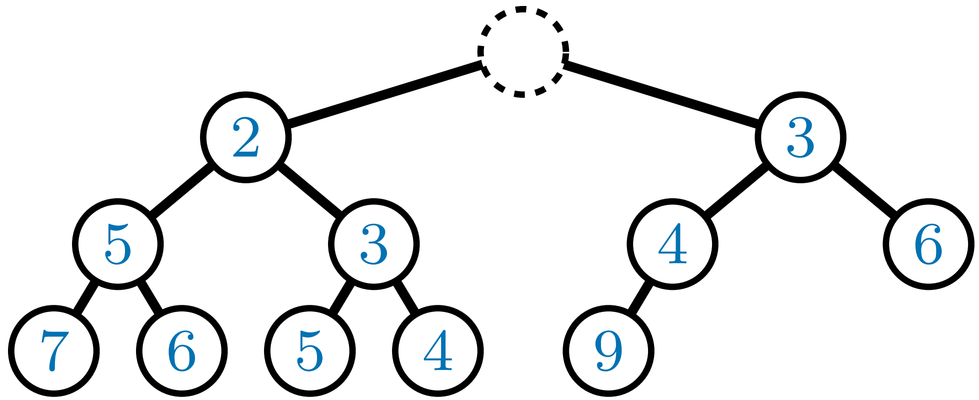
(call this element y)

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



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

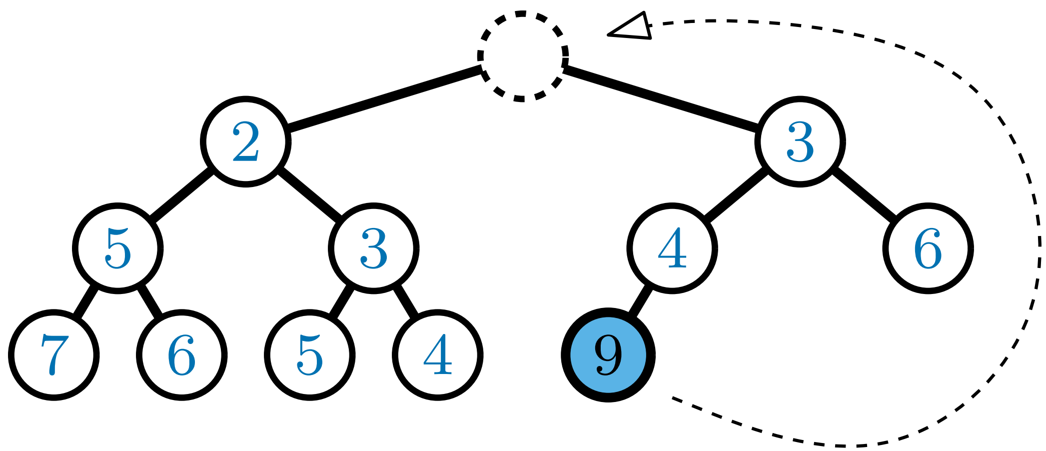
(call this element y)

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

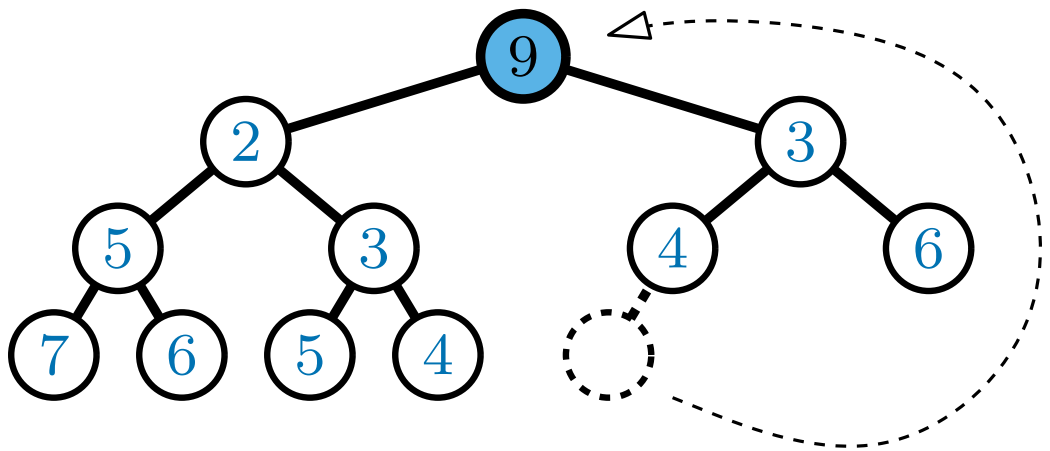
(call this element y)

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

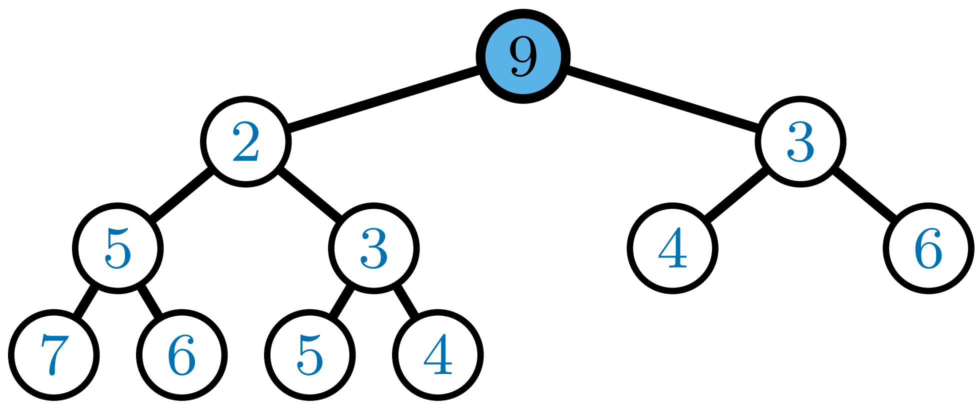
(call this element y)

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

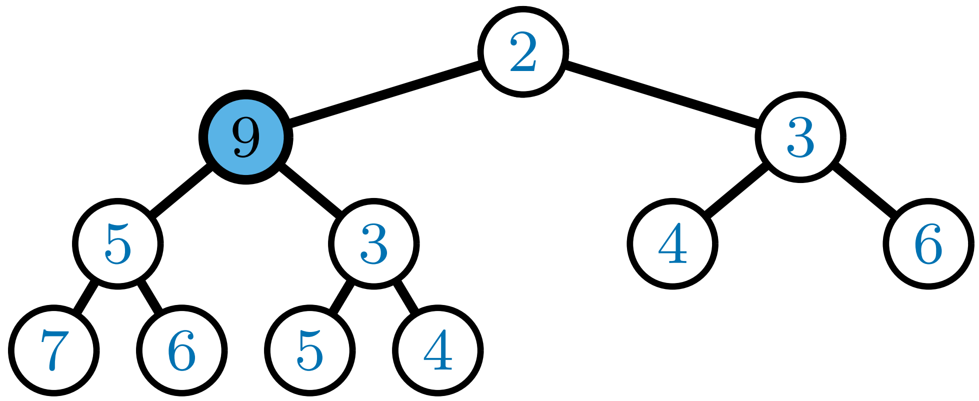
(call this element y)

Step 3: While y.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

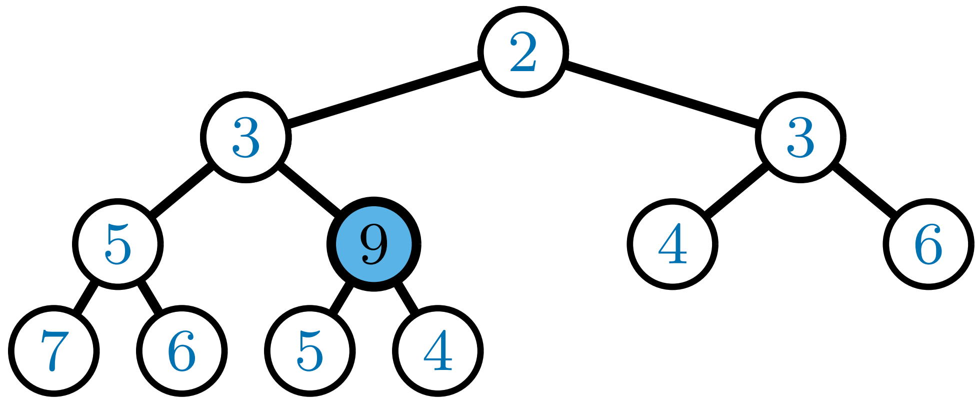
(call this element y)

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

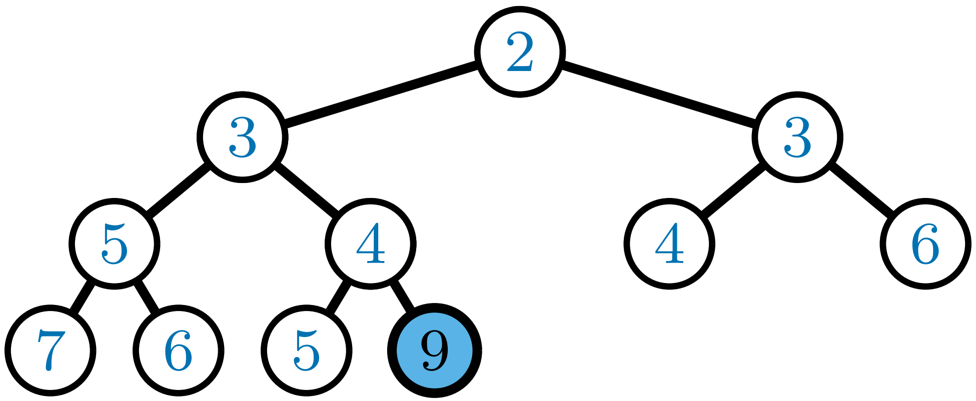
(call this element y)

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

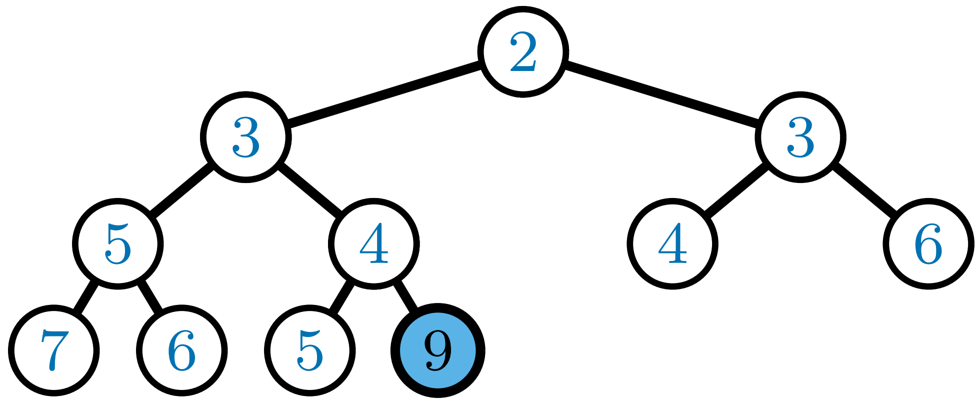
(call this element y)

Step 3: While y.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



$O(1)$ time

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

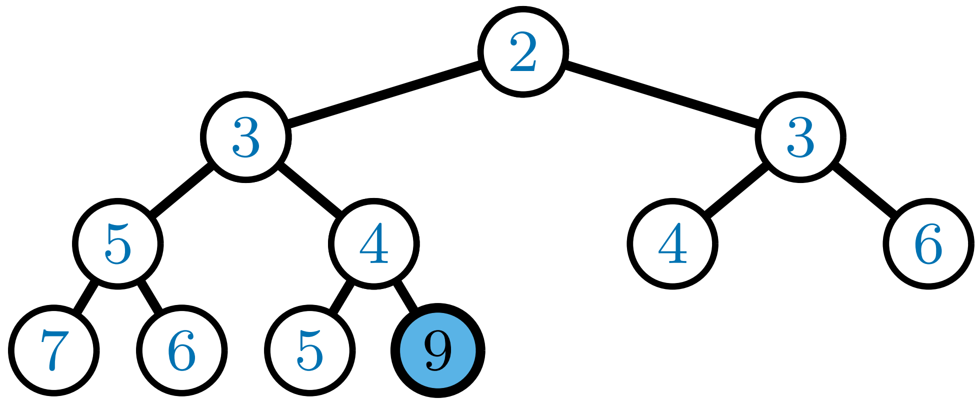
(call this element y)

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



$O(1)$ time

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

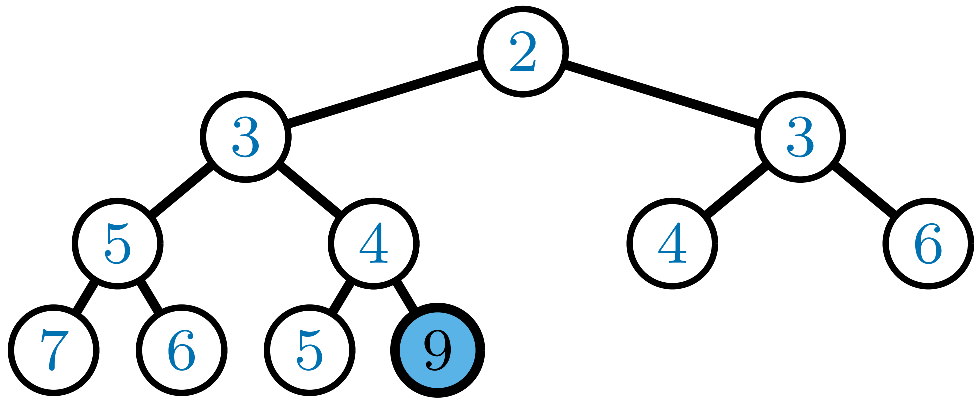
(call this element y)

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



$O(1)$ time

Each swap takes

$O(1)$ time

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

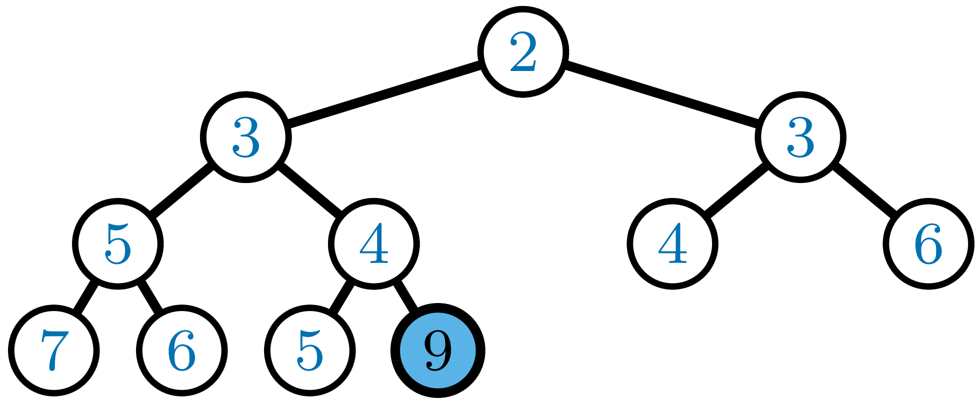
(call this element y)

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



$O(1)$ time

Each swap takes

$O(1)$ time

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)

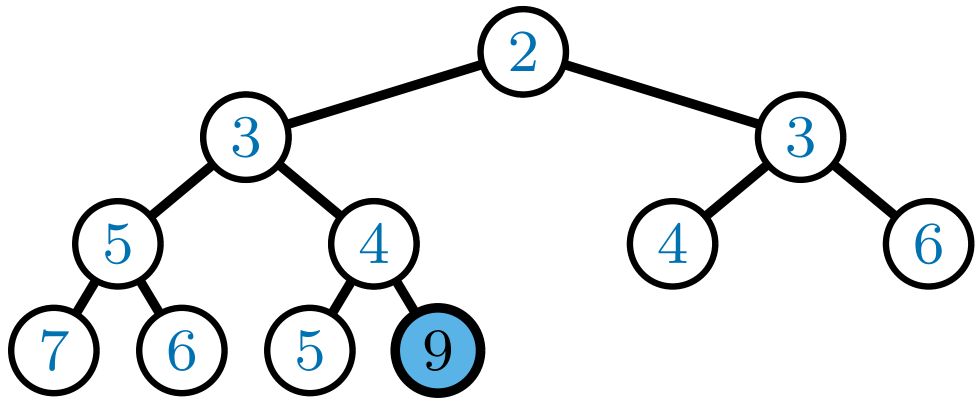
Step 3: While y.key is larger than one of its children's: *(stop if y becomes a leaf)*

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

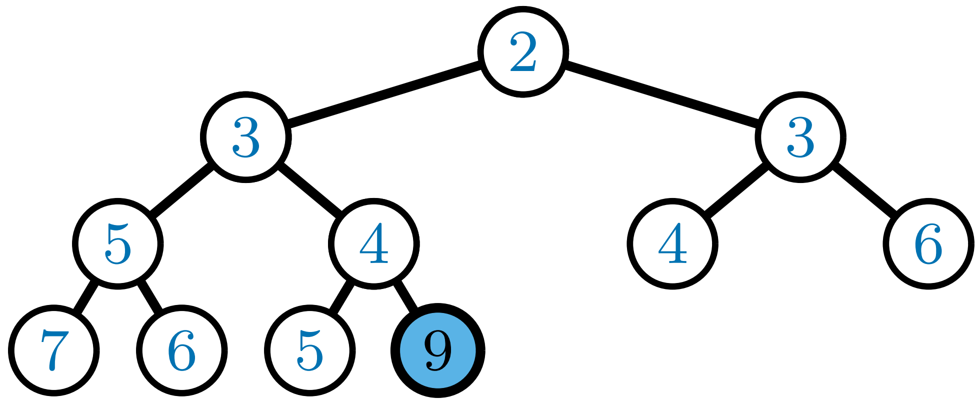
(call this element y)

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

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$

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

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$

What happened to that assumption?

Priority queue Summary

We have seen three different **priority queue** implementations each supporting 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

where $k < x.key$

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

	$\textsc{Insert}$	$\textsc{DecreaseKey}$	$\textsc{ExtractMin}$
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$

What happened to that assumption?

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

That pesky **assumption**. . .

That pesky **assumption**...

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

That pesky **assumption**. . .

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

Previously we said that. . .

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

That pesky **assumption**...

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

Previously we said that...

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

New (more reasonable) Assumption

Each element x also has an associated (unique) positive integer **ID** - x.ID $\leqslant N$

That pesky **assumption**...

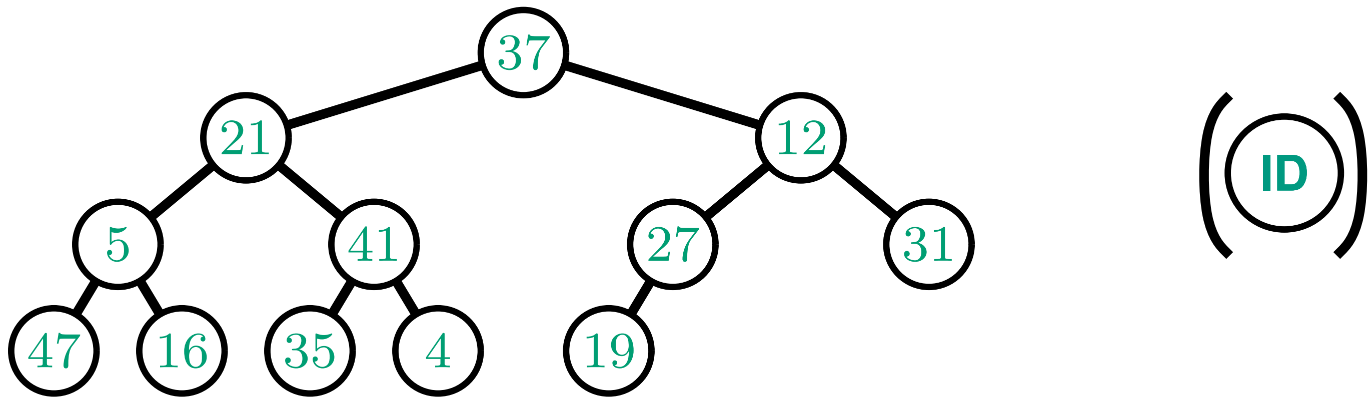
Old Assumption we can find the location of any element x in the Heap in $O(1)$ time

Previously we said that...

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

New (more reasonable) Assumption

Each element x also has an associated (unique) positive integer **ID** - x.ID $\leqslant N$



That pesky **assumption**...

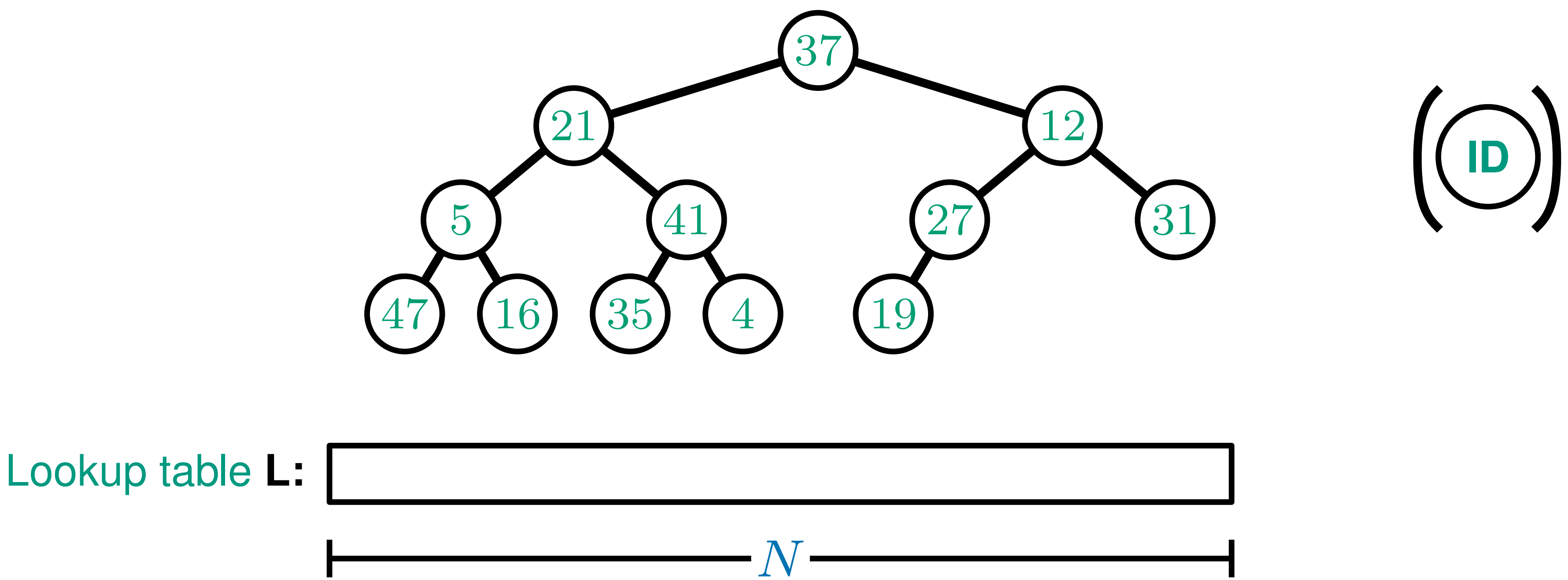
Old Assumption we can find the location of any element x in the Heap in $O(1)$ time

Previously we said that...

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

New (more reasonable) Assumption

Each element x also has an associated (unique) positive integer **ID** - x.ID $\leqslant N$



That pesky **assumption**. . .

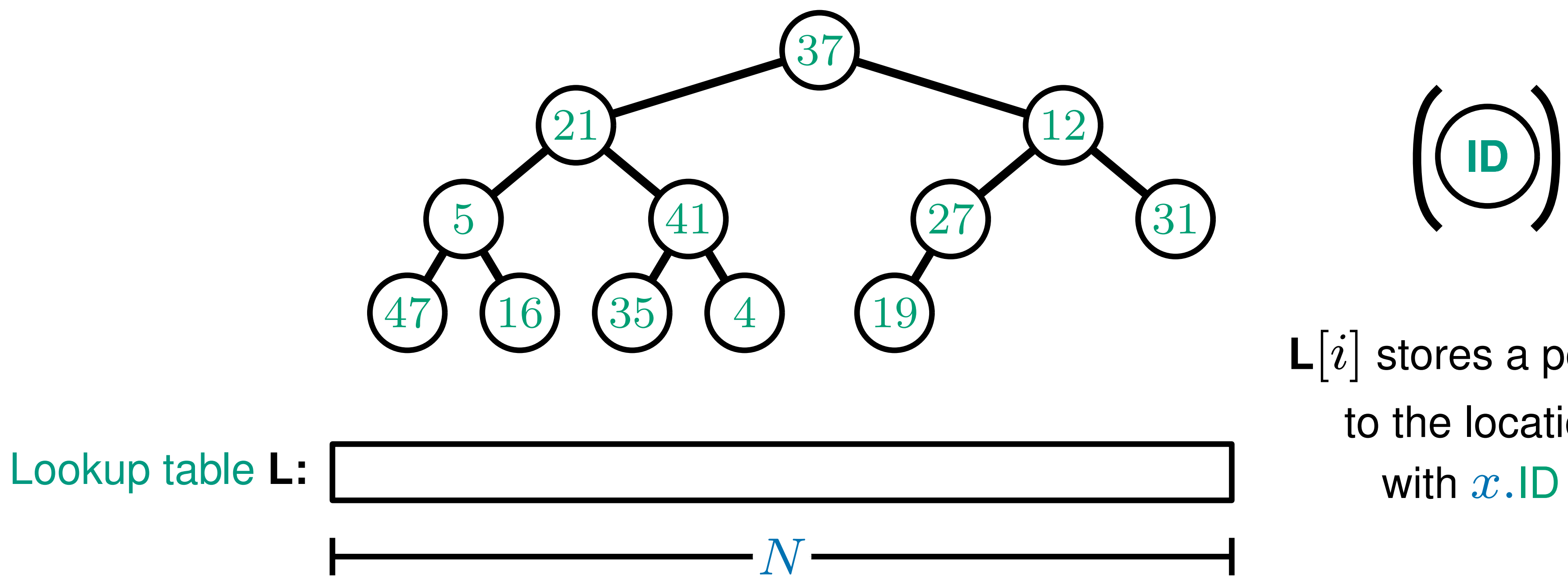
Old Assumption we can find the location of any element x in the Heap in $O(1)$ time

Previously we said that. . .

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

New (more reasonable) Assumption

Each element x also has an associated (unique) positive integer **ID** - $x.\text{ID} \leqslant N$



L$[i]$ stores a pointer

to the location of x

with $x.\text{ID} = i$

That pesky **assumption**. . .

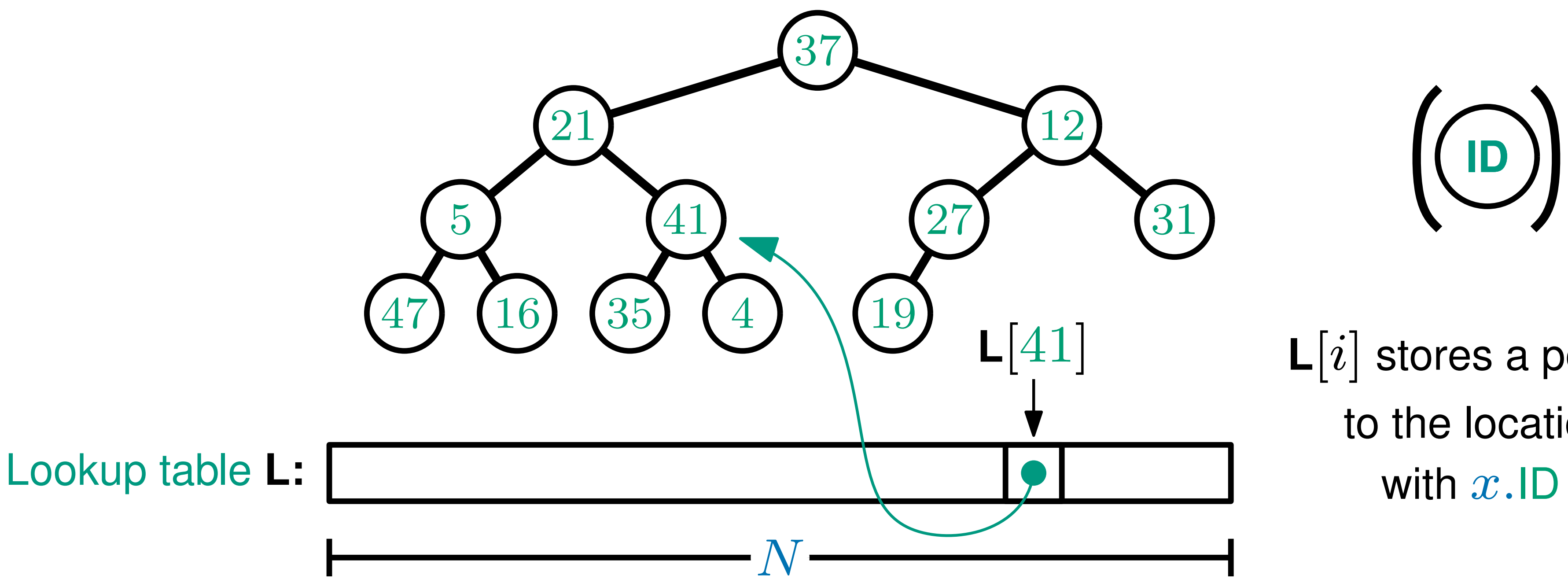
Old Assumption we can find the location of any element x in the Heap in $O(1)$ time

Previously we said that. . .

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

New (more reasonable) Assumption

Each element x also has an associated (unique) positive integer **ID** - $x.\text{ID} \leqslant N$



L$[i]$ stores a pointer

to the location of x

with $x.\text{ID} = i$

That pesky **assumption**...

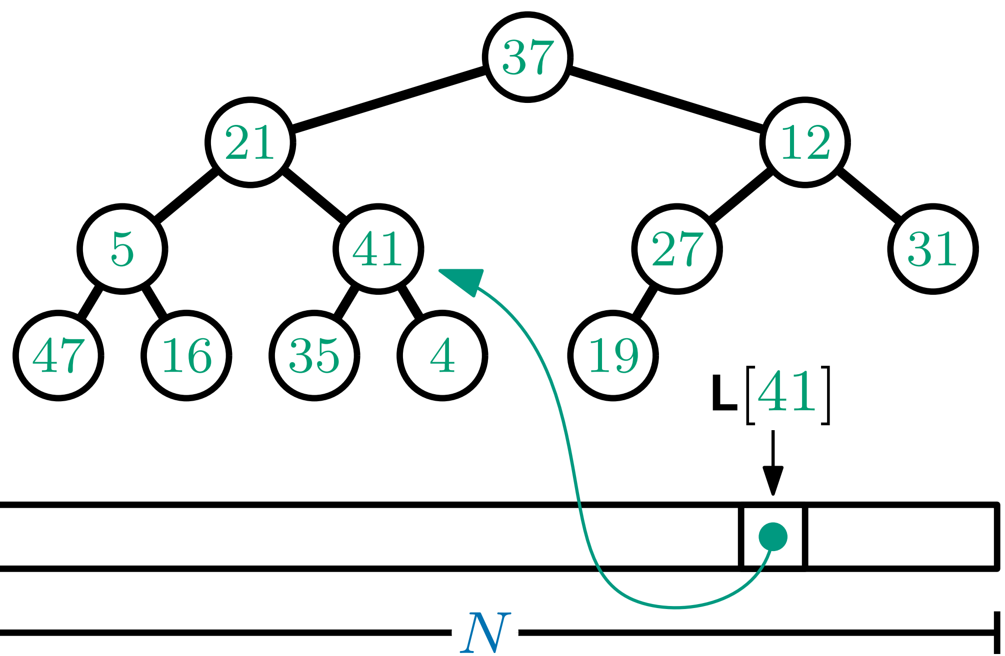
Old Assumption we can find the location of any element x in the Heap in $O(1)$ time

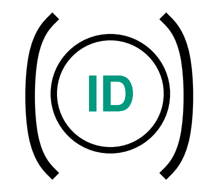
Previously we said that...

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

New (more reasonable) Assumption

Each element x also has an associated (unique) positive integer **ID** - $x.\text{ID} \leqslant N$





Lookup table **L:**

L$[i]$ stores a pointer to the location of x with $x.\text{ID} = i$

Whenever we move an element we update **L** in $O(1)$ time

That pesky **assumption**. . .

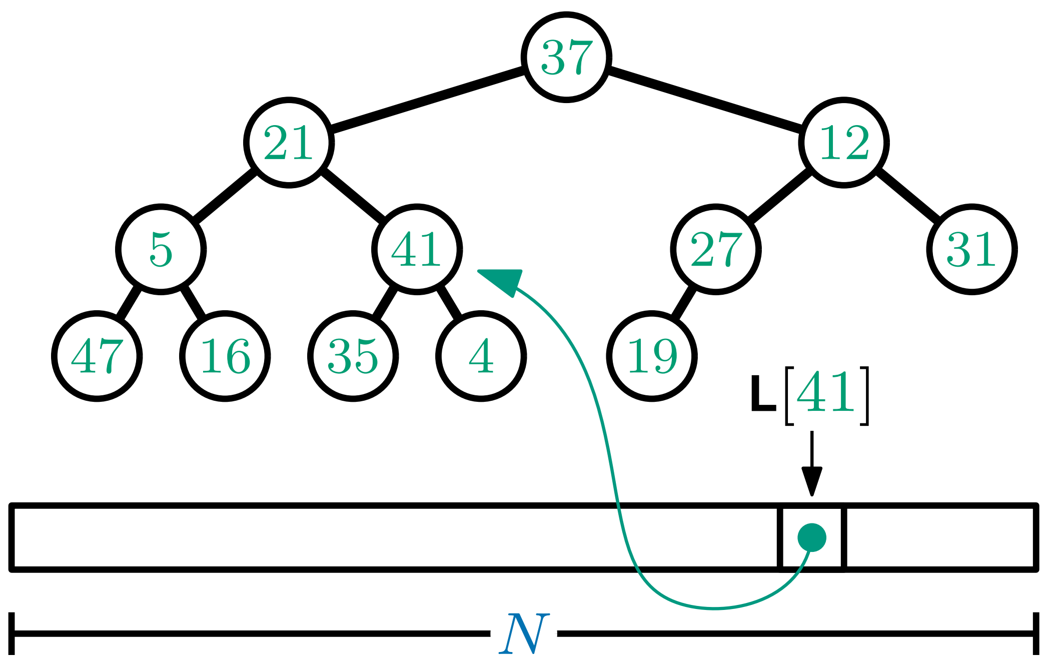
Old Assumption we can find the location of any element x in the Heap in $O(1)$ time

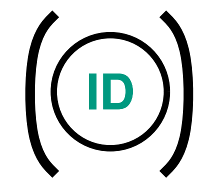
Previously we said that. . .

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

New (more reasonable) Assumption

Each element x also has an associated (unique) positive integer **ID** - $x.\text{ID} \leqslant N$





Lookup table **L:**

L$[i]$ stores a pointer to the location of x with $x.\text{ID} = i$

Whenever we move an element we update **L** in $O(1)$ time

Finding element x takes $O(1)$ time as required

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

	INSERT	DECREASEKEY	EXTRACTMIN	SPACE
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(N)$

Priority queue Summary

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

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

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

where $k < x.key$

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

	INSERT	DECREASEKEY	EXTRACTMIN	SPACE
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(N)$

Spoiler: for the shortest path problem, $N = O(n)$

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

	INSERT	DECREASEKEY	EXTRACTMIN	SPACE
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(N)$

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

where $k < x.key$

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

	INSERT	DECREASEKEY	EXTRACTMIN	SPACE
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(N)$

Is this the best possible?

Priority queue Summary

We have seen three different **priority queue** implementations each supporting 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

where $k < x.key$

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

	Insert	DecreaseKey	ExtractMin	Space
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(N)$

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

Priority queue Summary

We have seen three different **priority queue** implementations each supporting 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

where $k < x.key$

$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

	INSERT	DECREASEKEY	EXTRACTMIN	SPACE
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(N)$

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

Fibonacci Heap	$O(1)$	$O(1)$	$O(\log n)$	$O(n)$

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 $k < x.key$

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

	INSERT	DECREASEKEY	EXTRACTMIN	SPACE
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(N)$

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

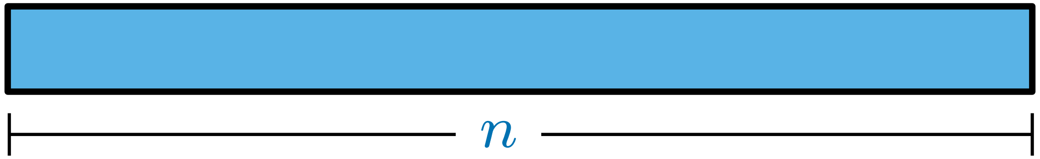
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log n)$	$O(n)$

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

One more thing. . .

Take an array of elements of length n

$\mathbf{A}$



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

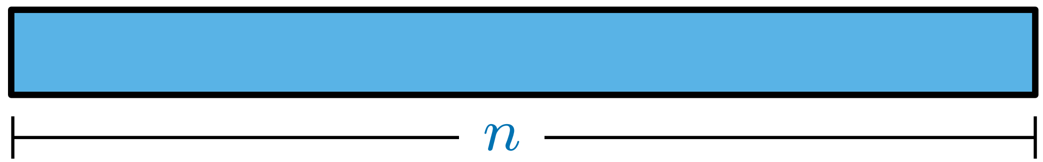
$\mathbf{A}'$

what is $\mathbf{A}'$?

One more thing. . .

Take an array of elements of length n

$\mathbf{A}$

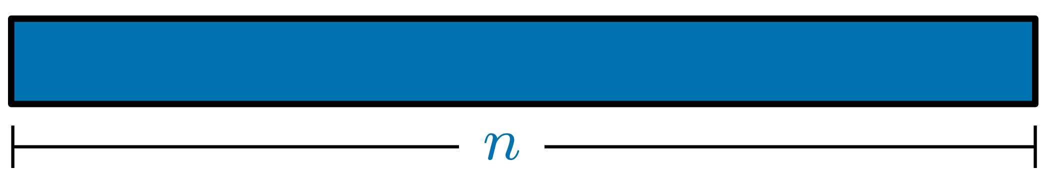


INSERT every element into a priority queue:

EXTRACTMIN from the priority queue n times

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

$\mathbf{A}'$



what is $\mathbf{A}'$? *it's A in sorted order*

One more thing...

Take an array of elements of length n

$\mathbf{A}$ — 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

$\mathbf{A}'$ — n

what is $\mathbf{A}'$? *it's A in sorted order*

HeapSort

Take an array of elements of length n

$\mathbf{A}$

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

$\mathbf{A}'$

n

what is $\mathbf{A}'$? *it's A in sorted order*

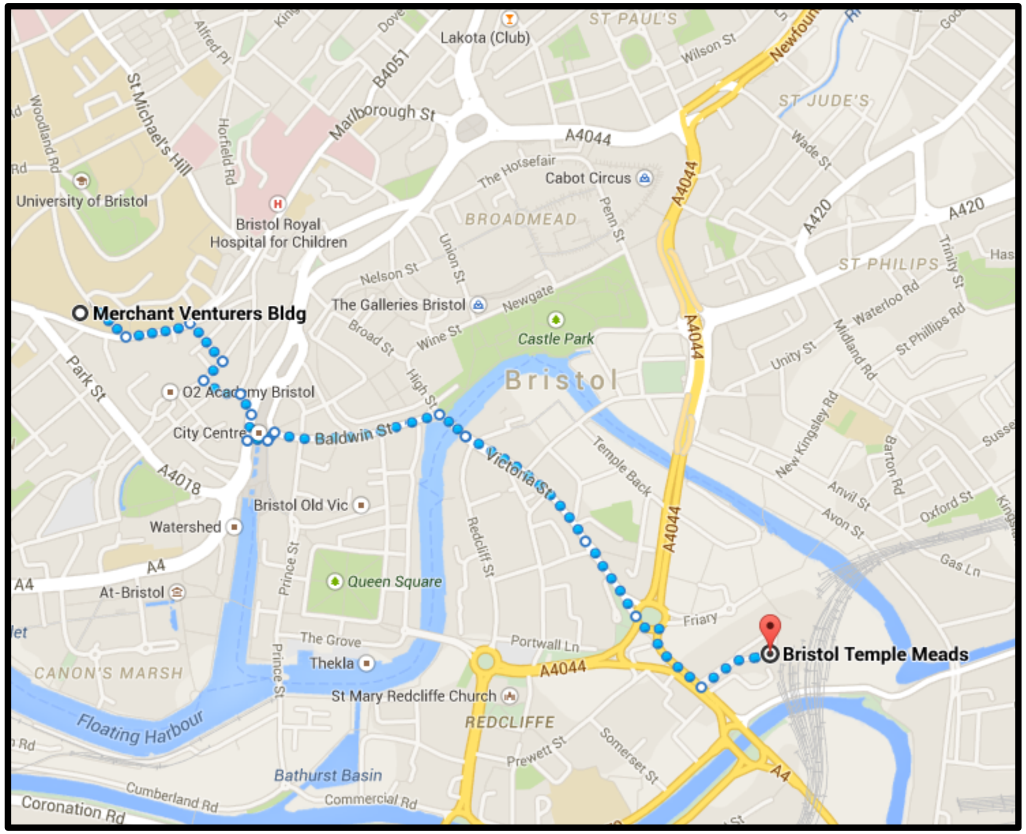
University of BRISTOL

End of part one

Part two

Dijkstra’s Algorithm

In today's lectures we'll be discussing the **single source shortest paths** problem
in a weighted, directed graph. . .

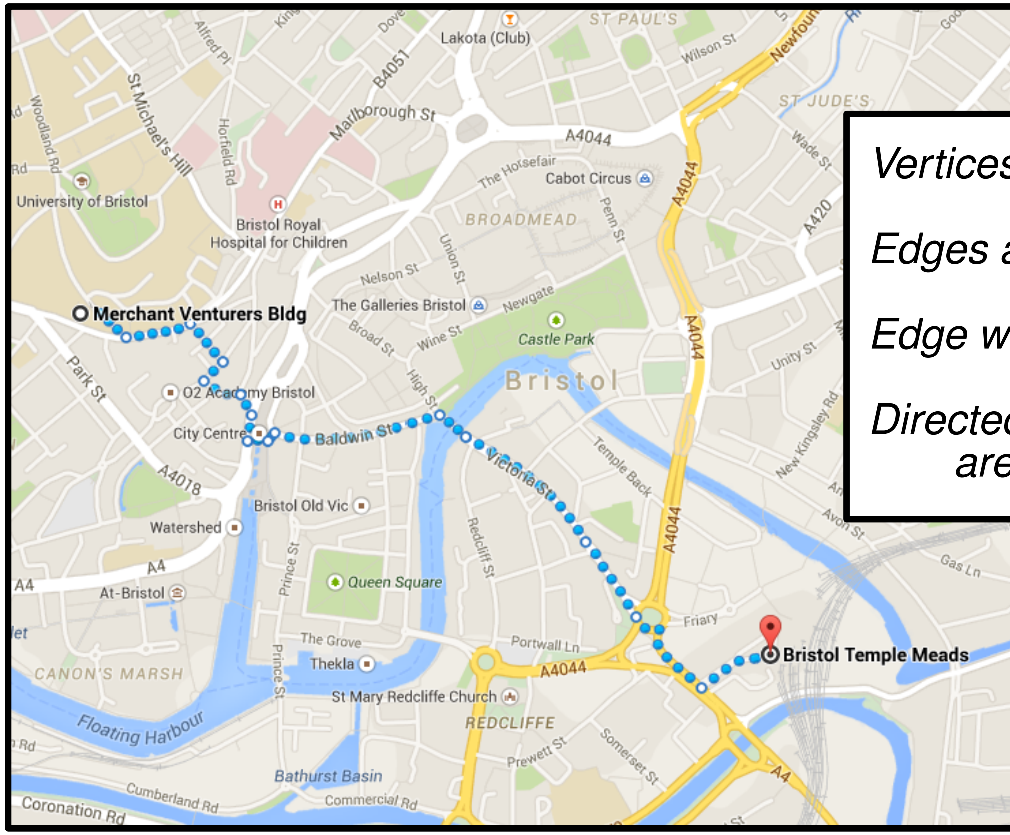


The shortest path from MVB to Temple Meads
(according to Google Maps)

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 lectures we'll be discussing the **single source shortest paths** problem
in a weighted, directed graph. . .



Vertices are junctions

Edges are roads

Edge weights are in miles

Directed edges
are one-way streets

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

Post-lunch Priority Queue refresher

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

where $k < x.key$

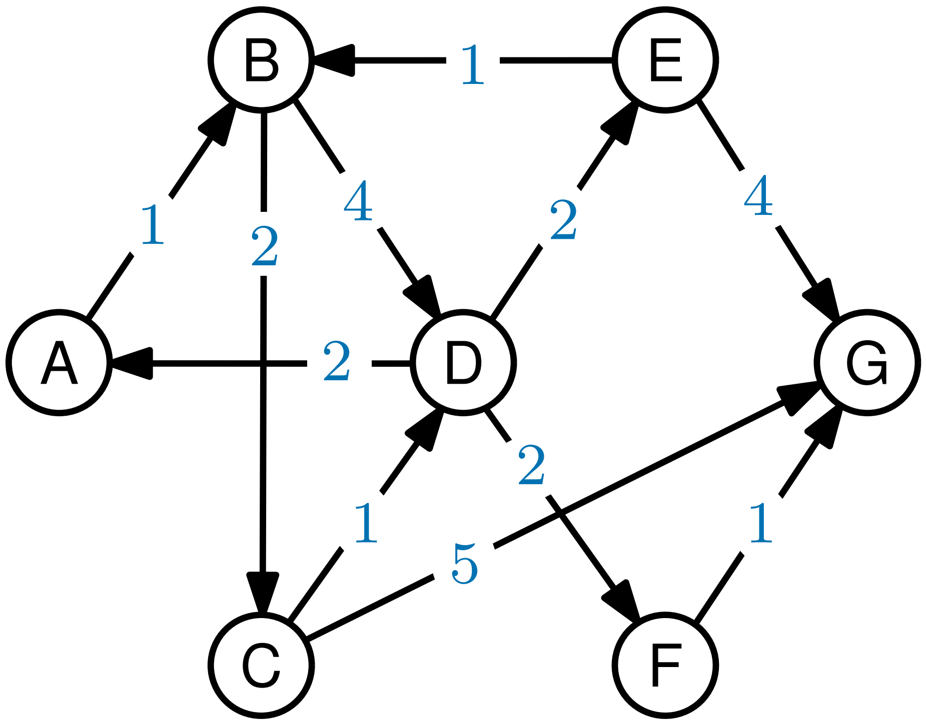
$\textsc{ExtractMin}()$ - removes and returns the element with the smallest key

(ties are broken arbitrarily)

Single source shortest paths

Dijkstra's Algorithm solves the **single source shortest paths** problem

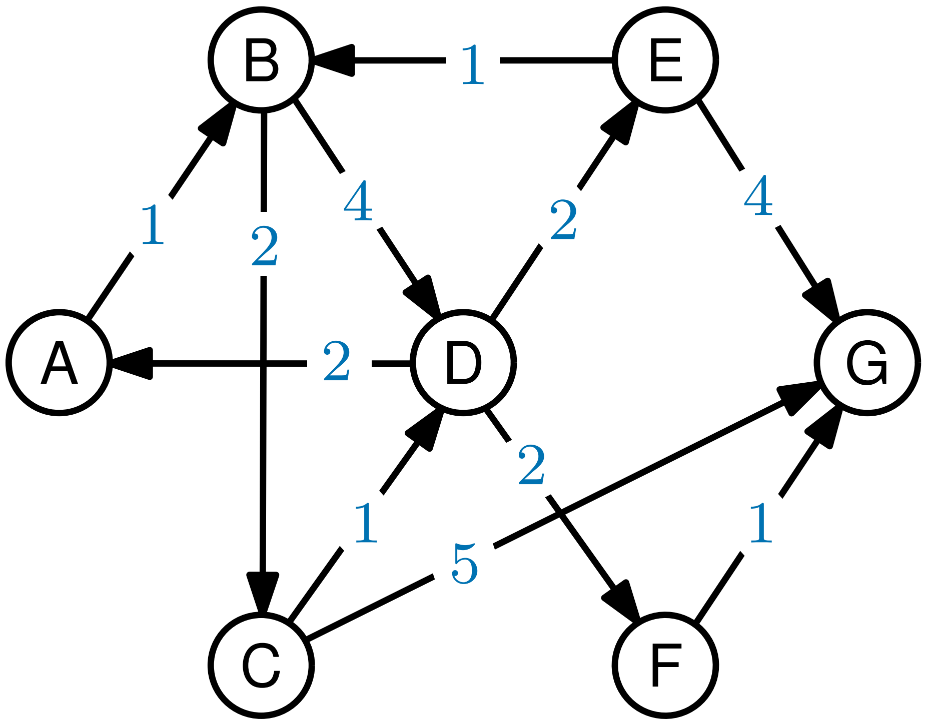
in a **weighted**, directed graph...



Single source shortest paths

Dijkstra's Algorithm solves the **single source shortest paths** problem

in a **weighted**, directed graph...



It finds the shortest path from

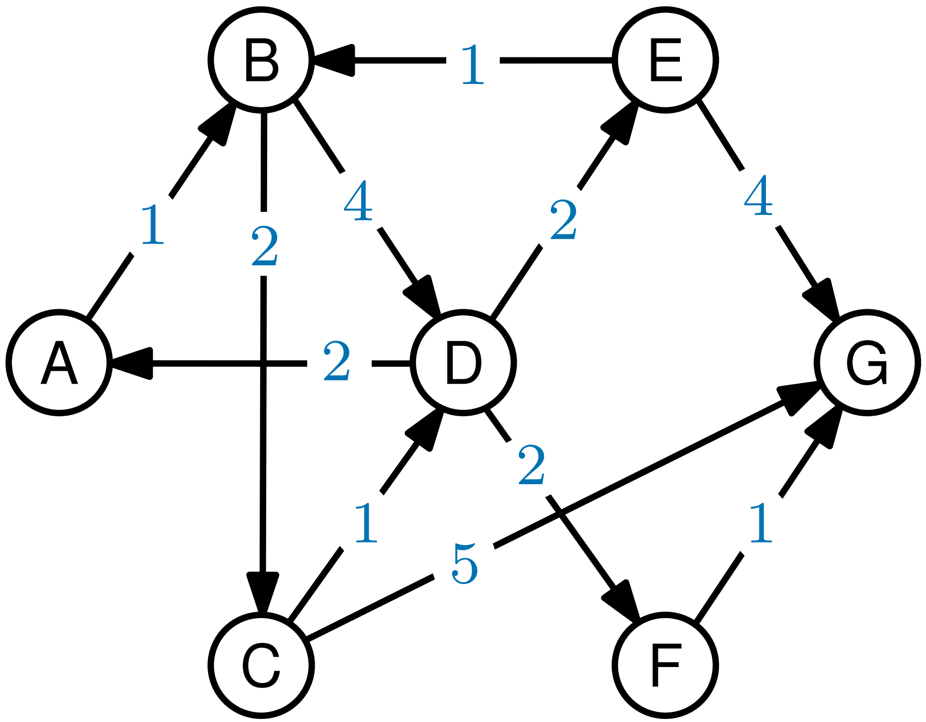
a given *source* vertex

to every other vertex

Single source shortest paths

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

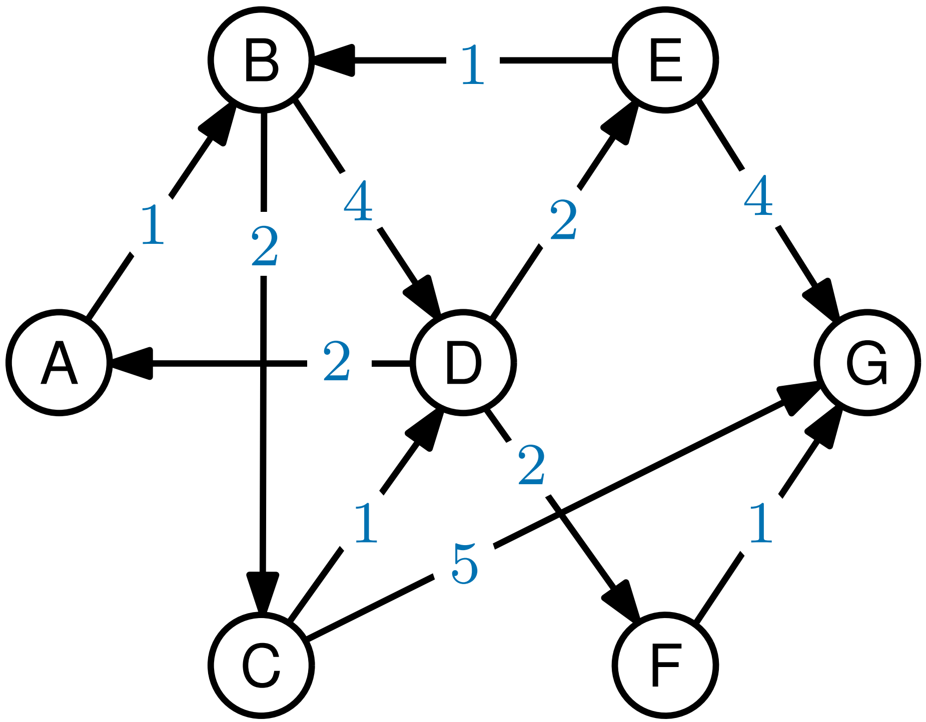
to every other vertex

The weights have to be non-negative

Single source shortest paths

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

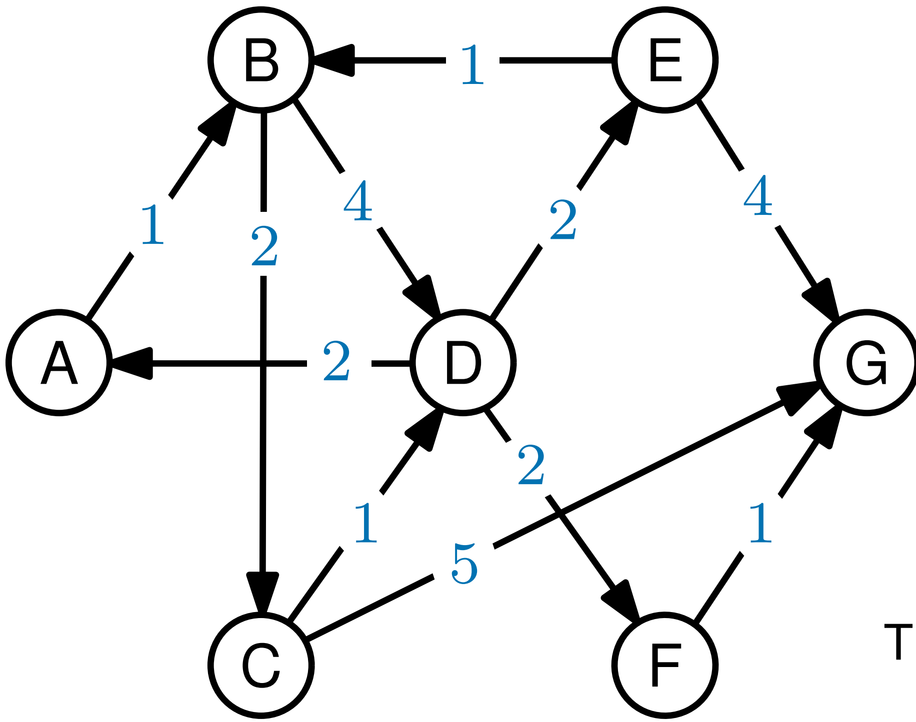
The weights have to be non-negative

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

Single source shortest paths

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

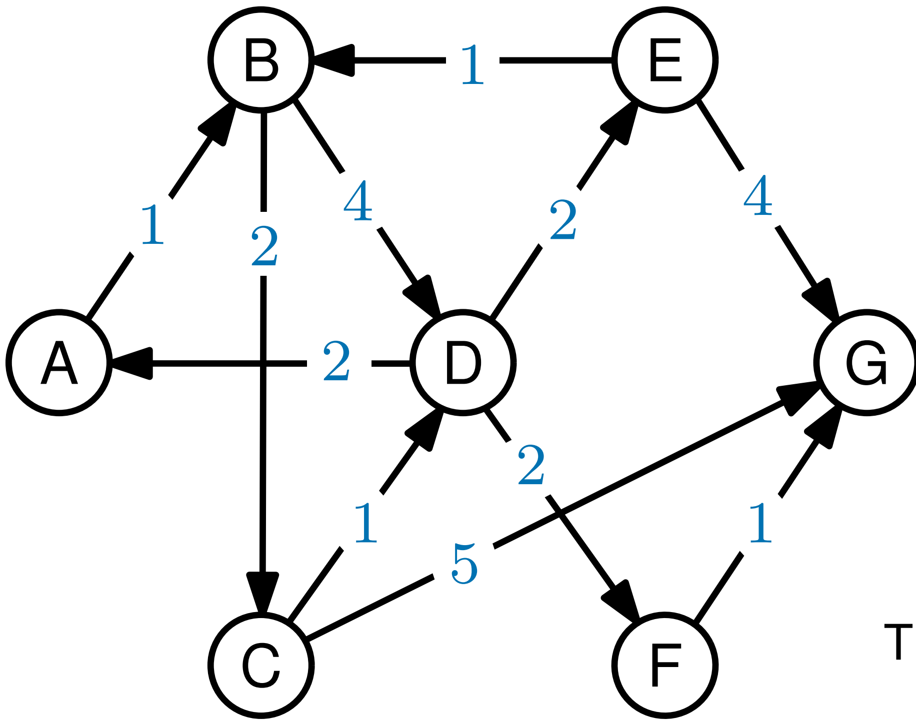
The graph is stored as an **Adjacency List**

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

Single source shortest paths

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 Monday's 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
INSERT, DECREASEKEY and EXTRACTMIN

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

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

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

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

Claim when Dijkstra's algorithm terminates,

for each vertex v, $\texttt{dist}(v)$ is the distance between s and v

$\textsc{Dijkstra}(s)$

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

We're going to simulate

$\textsc{Dijkstra}(A)$

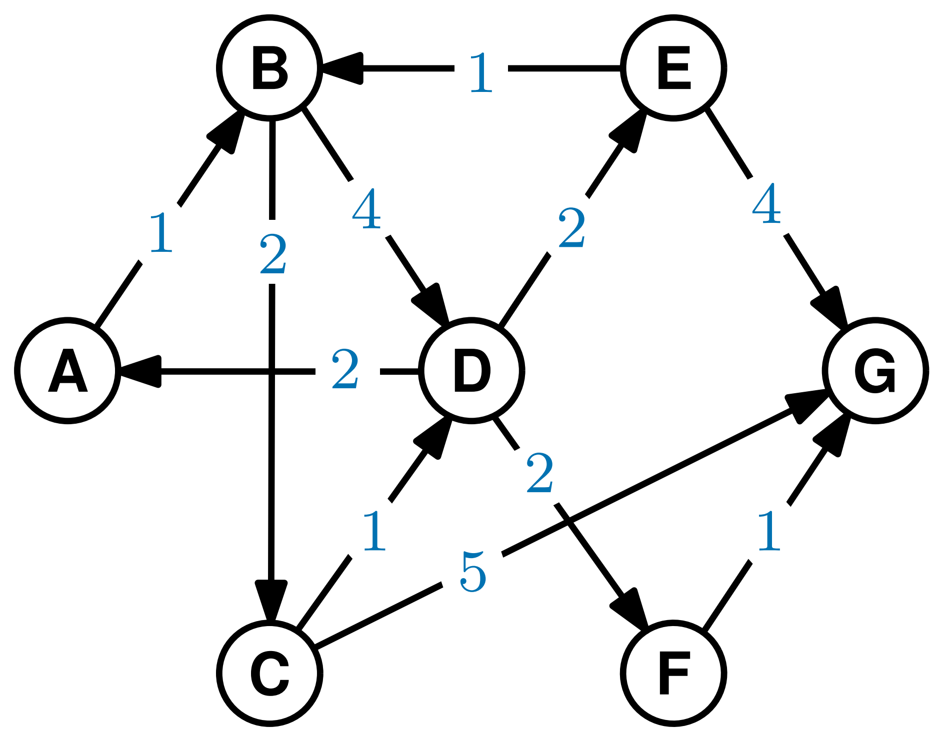
i.e. $s = A$

	A	B	C	D	E	F	G
dist:							

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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$



$\textsc{Dijkstra}(s)$

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

We're going to simulate $\textsc{Dijkstra}(A)$

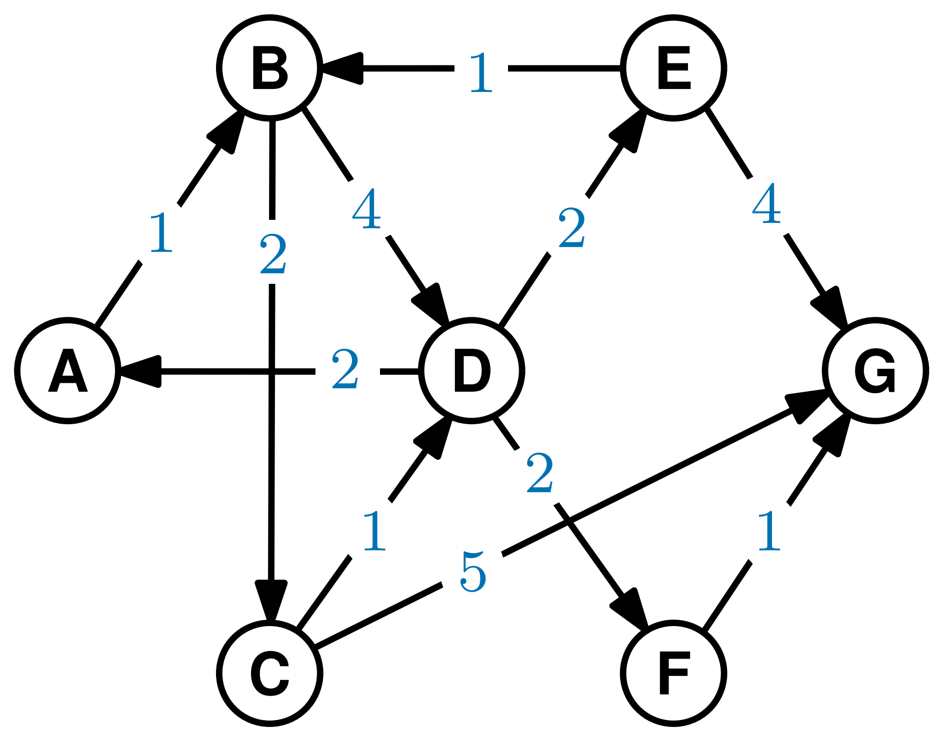
i.e. $s = A$

	A	B	C	D	E	F	G
dist:	0	∞	∞	∞	∞	∞	∞

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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$



University of BRISTOL

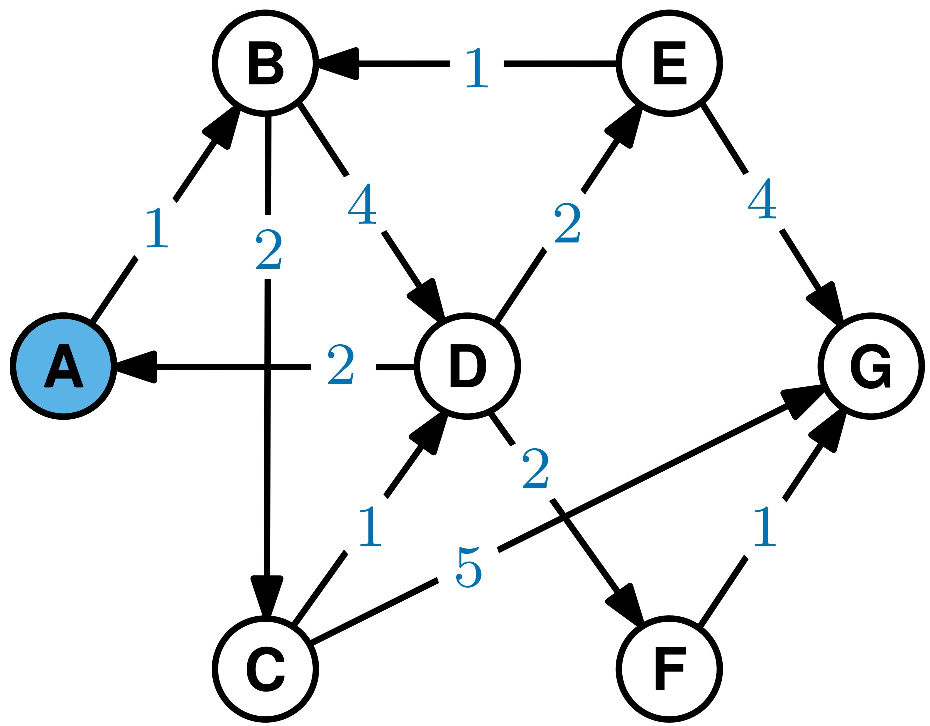
DIJKSTRA(s)

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

We're going to simulate DIJKSTRA(A)

i.e. s = A

	A	B	C	D	E	F	G
dist:	0	∞	∞	∞	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

University of BRISTOL

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

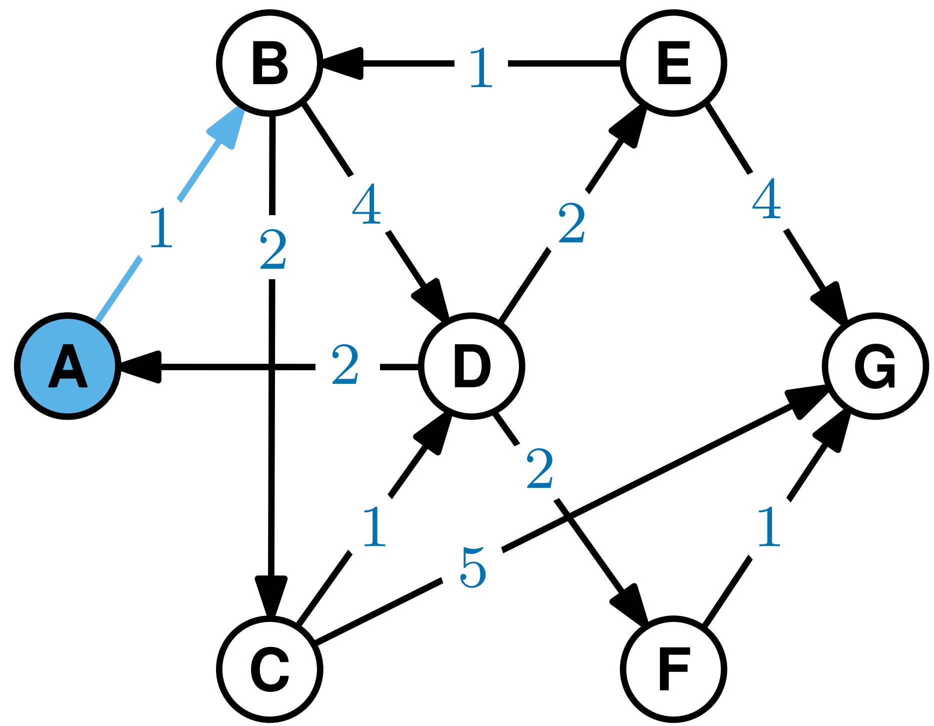
We're going to simulate

DIJKSTRA(A)

i.e. s = A

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

	A	B	C	D	E	F	G
dist:	0	∞	∞	∞	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

DIJKSTRA(s)

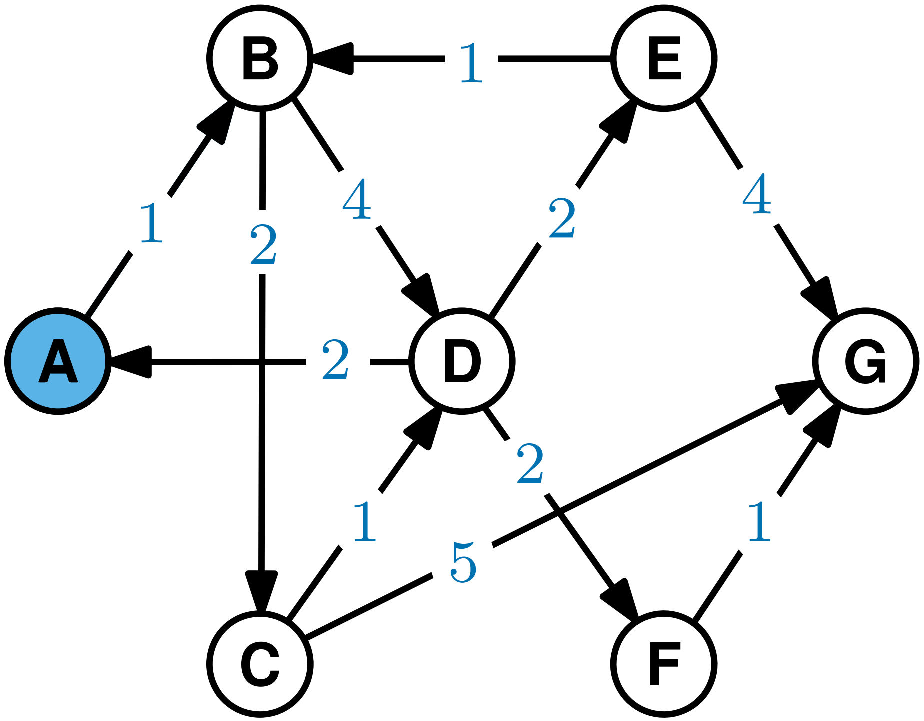
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

We're going to simulate DIJKSTRA(A)

i.e. s = A

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

	A	B	C	D	E	F	G
dist:	0	1	∞	∞	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \mathtt{dist}(v)$

```
DIJKSTRA(s)
```

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

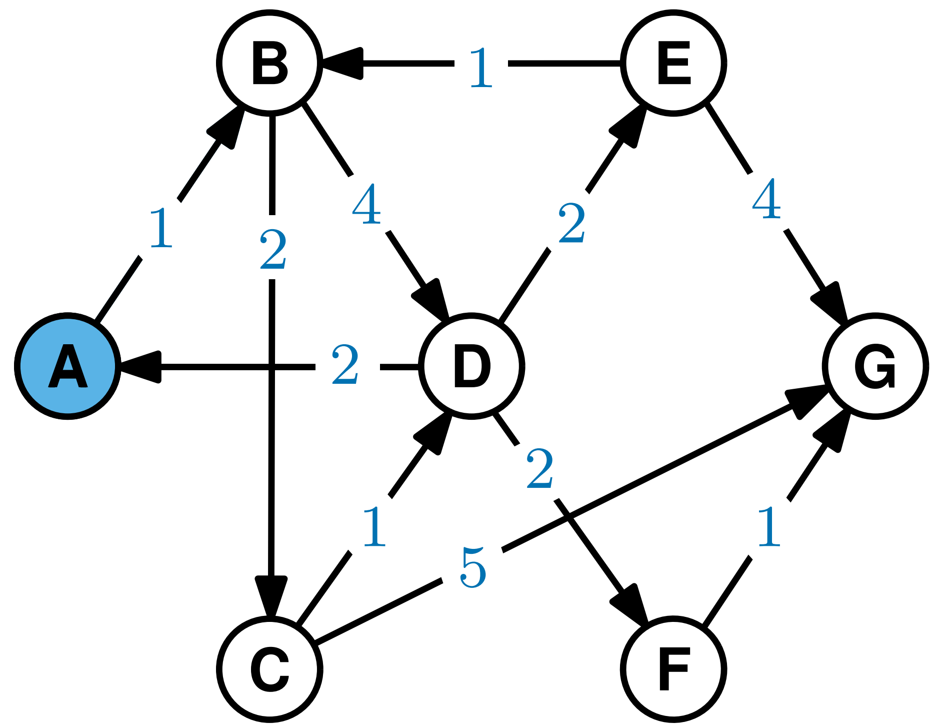
We're going to simulate DIJKSTRA(A)

i.e. $s = A$

this is called relaxing edge (u, v)

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

dist:	A	B	C	D	E	F	G
	0	1	∞	∞	∞	∞	∞



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

at all times, for each vertex v,

$v.\text{key} = \mathtt{dist}(v)$

DIJKSTRA(s)

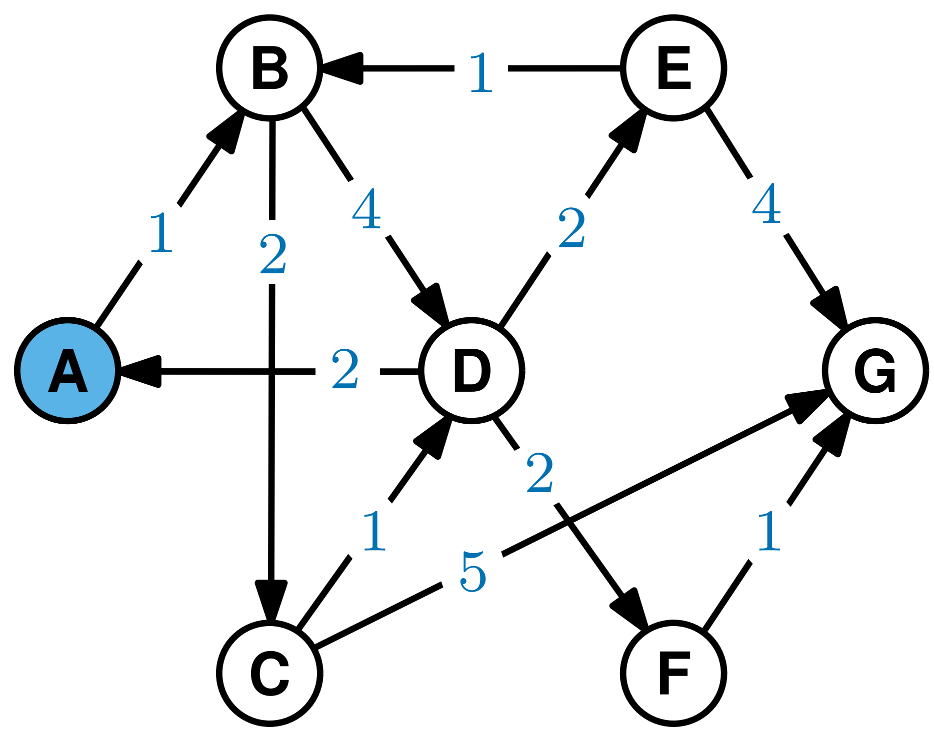
For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do INSERT(v, $\texttt{dist}(v)$)
While the queue is not empty
  $u =$ EXTRACTMIN()
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      DECREASEKEY(v, $\texttt{dist}(v)$)

this is called relaxing edge (u, v)

We're going to simulate DIJKSTRA(A)

i.e. $s =$ A

	A	B	C	D	E	F	G
dist:	0	1	∞	∞	∞	∞	∞



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

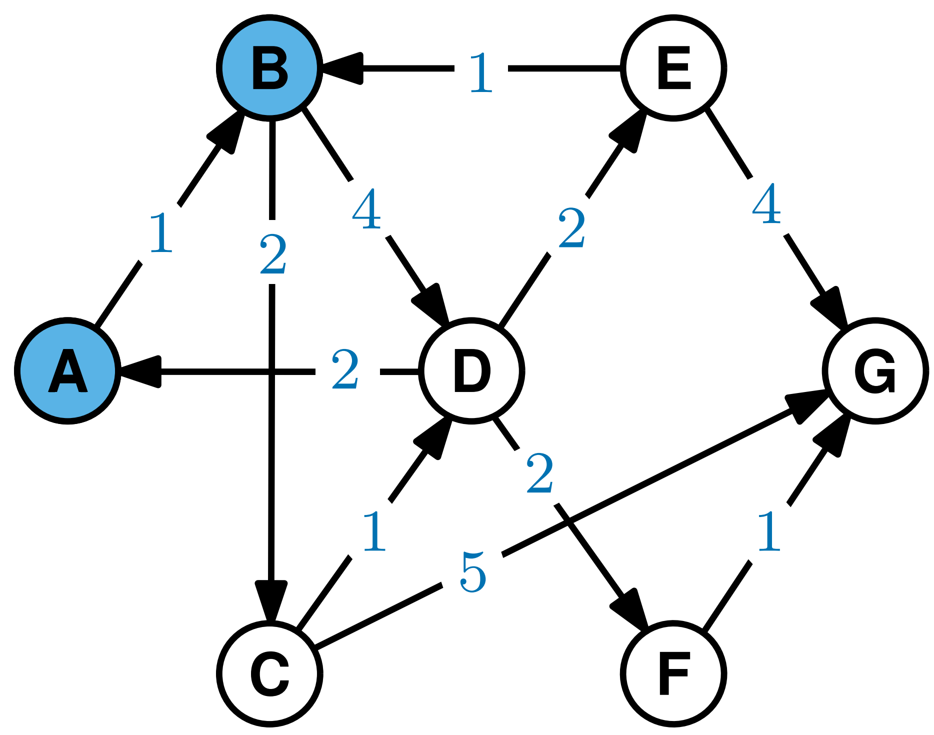
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

this is called relaxing edge (u, v)

We're going to simulate DIJKSTRA(A)

i.e. s = A

	A	B	C	D	E	F	G
dist:	0	1	∞	∞	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \mathtt{dist}(v)$

DIJKSTRA(s)

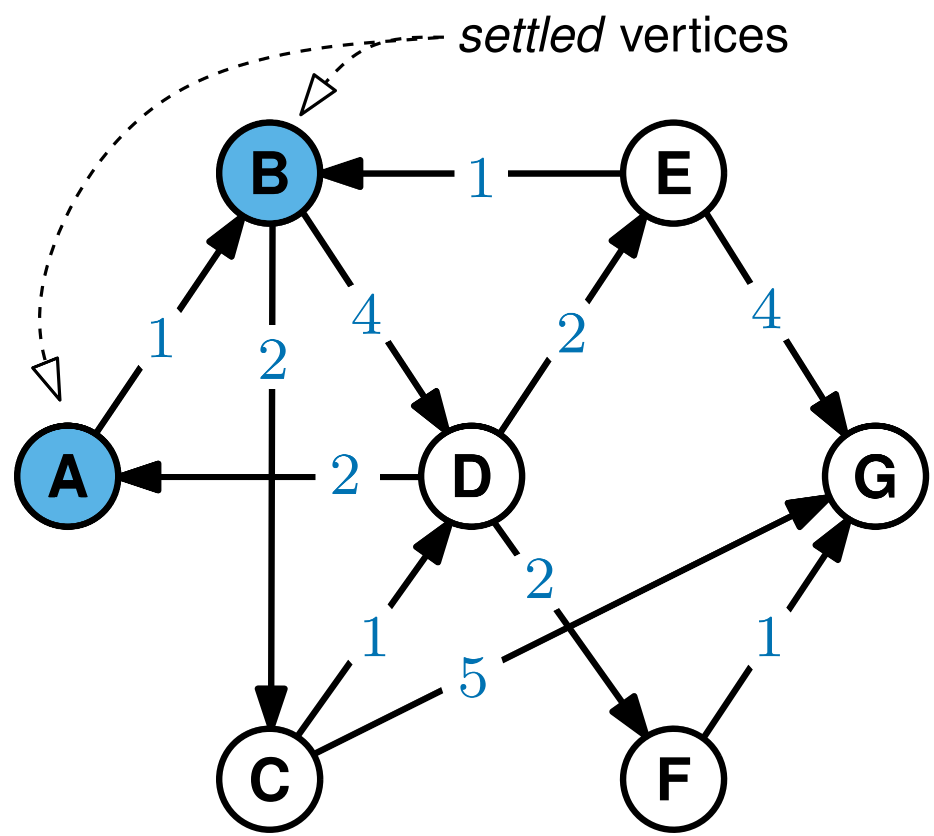
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

this is called relaxing edge (u, v)

We're going to simulate DIJKSTRA(A)

i.e. $s =$ A

	A	B	C	D	E	F	G
dist:	0	1	∞	∞	∞	∞	∞



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

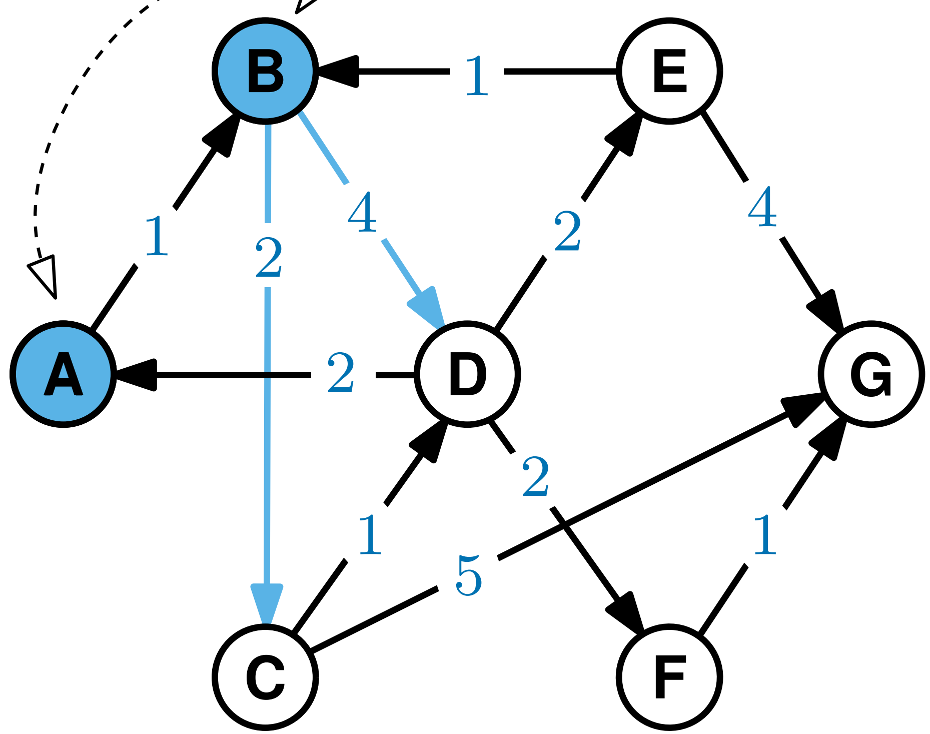
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u,v) ∈ E
      If dist(v) > dist(u) + weight(u,v)
         dist(v) = dist(u) + weight(u,v)
         DECREASEKEY(v, dist(v))
```

We're going to simulate DIJKSTRA(A)

i.e. $s = $ A

this is called relaxing edge (u, v)

settled vertices



	A	B	C	D	E	F	G
dist:	0	1	∞	∞	∞	∞	∞

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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

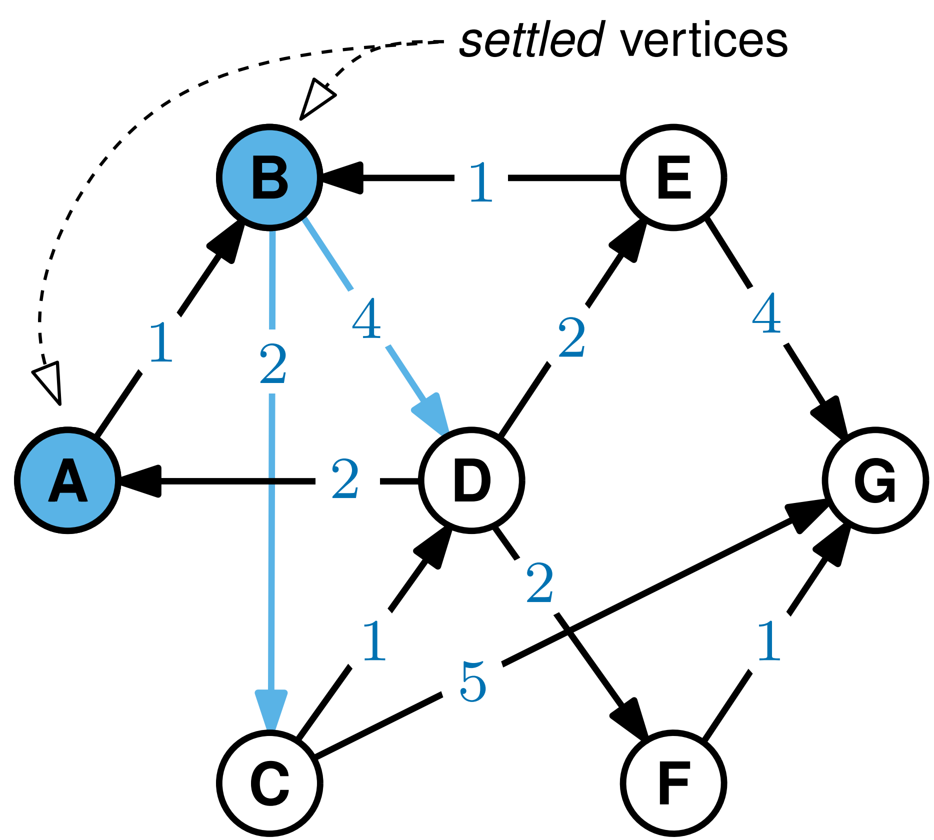
DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)



new path to **C** = $1 + 2 = 3$

	A	B	C	D	E	F	G
dist:	0	1	∞	∞	∞	∞	∞

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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

```
DIJKSTRA(s)
```

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

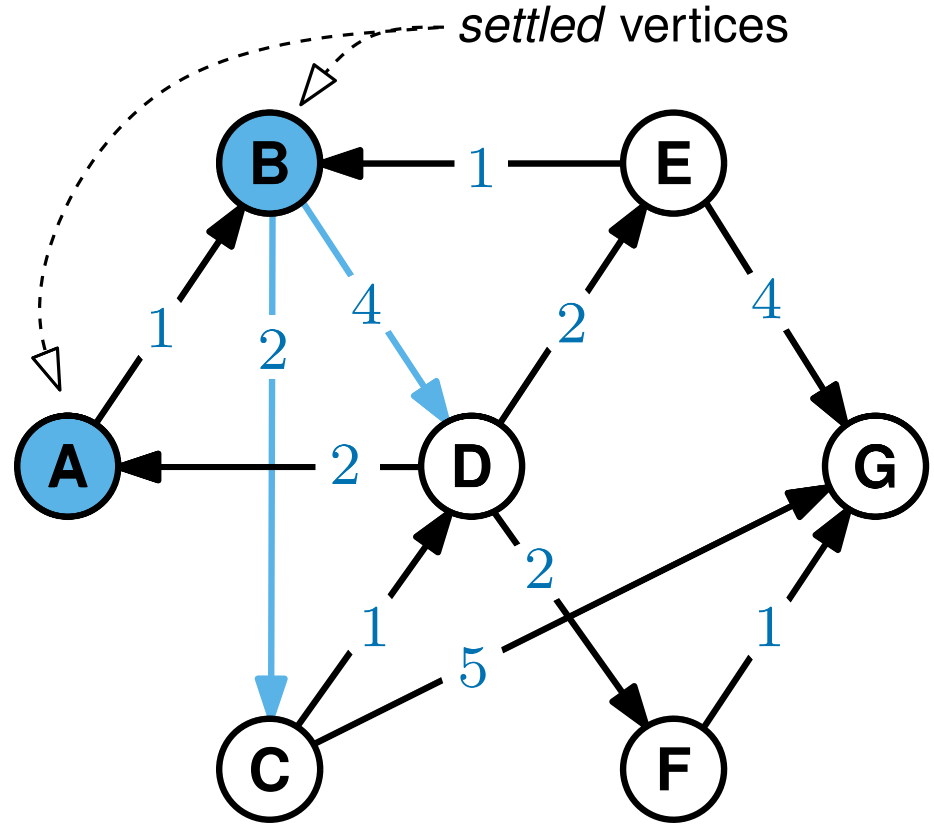
We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

new path to **C** = $1 + 2 = 3$

	A	B	C	D	E	F	G
dist:	0	1	3	∞	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

We're going to simulate DIJKSTRA(A)

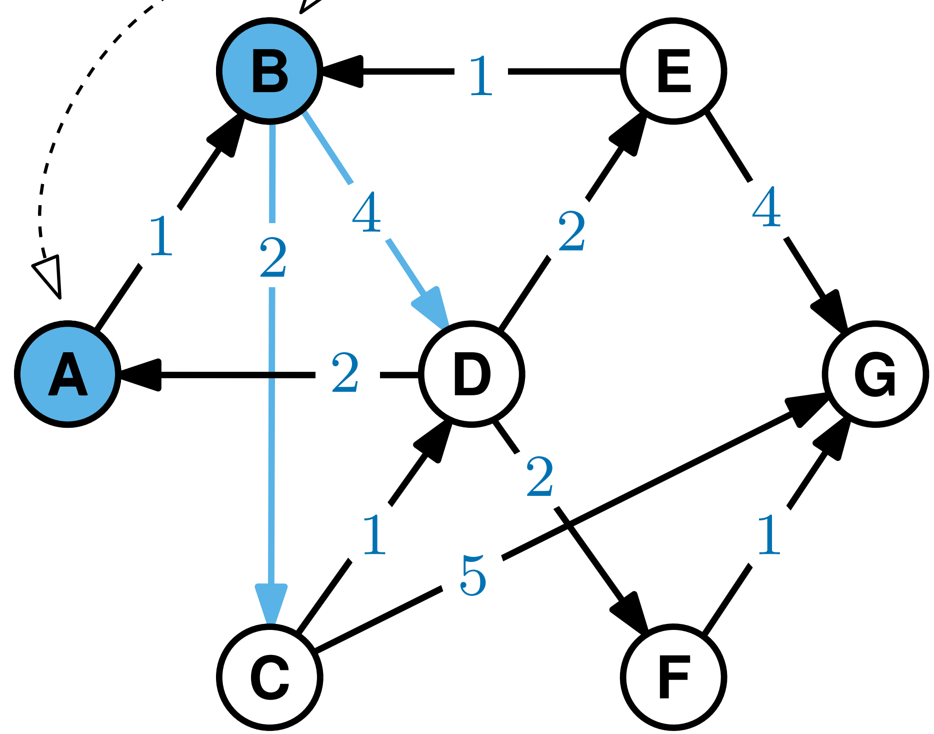
i.e. s = A

this is called relaxing edge (u, v)

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

	A	B	C	D	E	F	G
dist:	0	1	3	∞	∞	∞	∞

settled vertices



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

We're going to simulate DIJKSTRA(A)

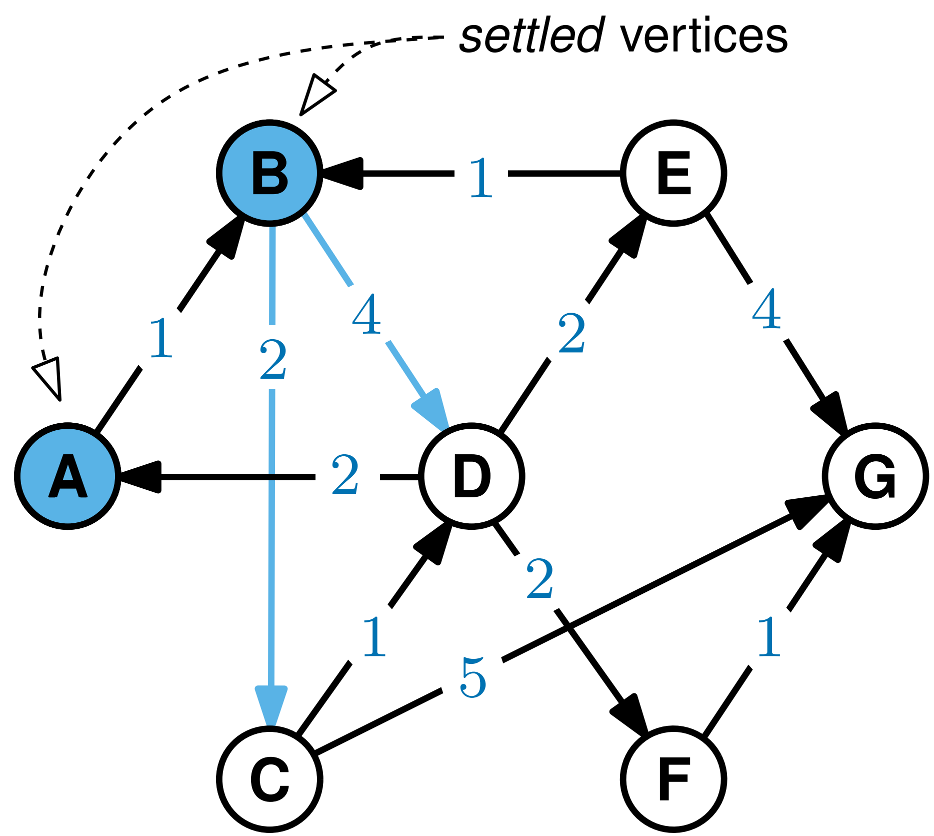
i.e. $s = \text{A}$

this is called relaxing edge (u, v)

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

dist:

A	B	C	D	E	F	G
0	1	3	5	∞	∞	∞



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

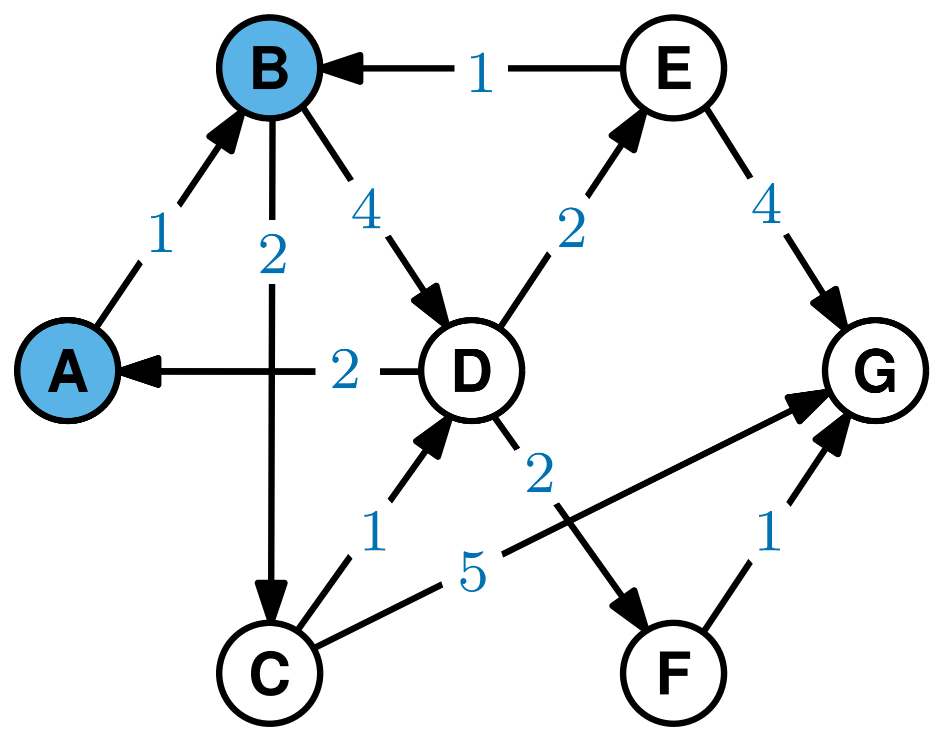
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u,v) ∈ E
    If dist(v) > dist(u) + weight(u,v)
      dist(v) = dist(u) + weight(u,v)
      DECREASEKEY(v, dist(v))
```

We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	5	∞	∞	∞



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 = dist(v)

DIJKSTRA(s)

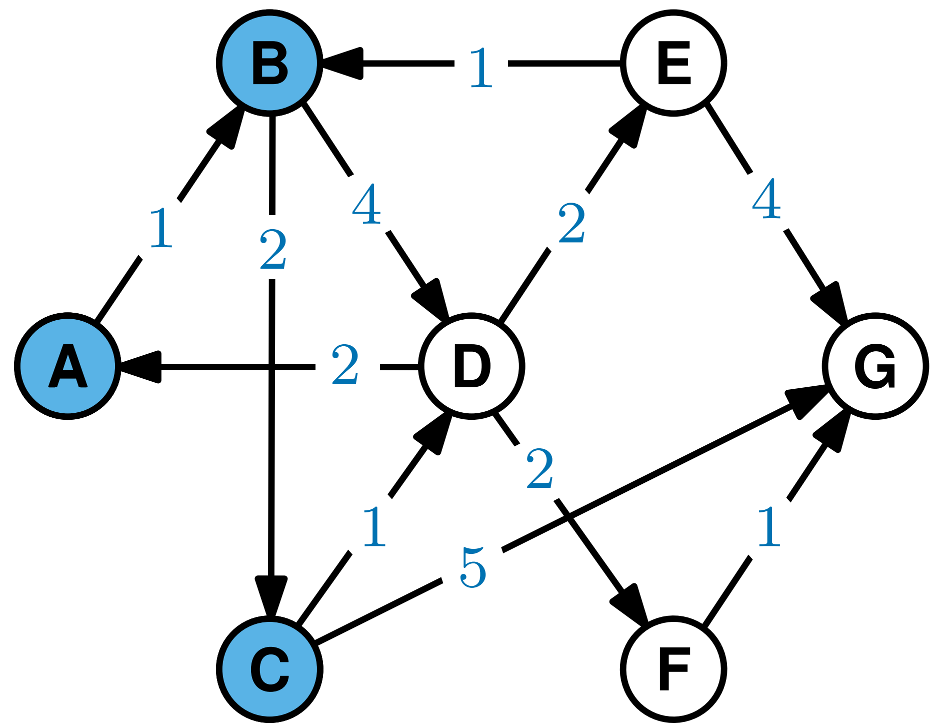
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

this is called relaxing edge (u, v)

We're going to simulate DIJKSTRA(A)

i.e. s = A

	A	B	C	D	E	F	G
dist:	0	1	3	5	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

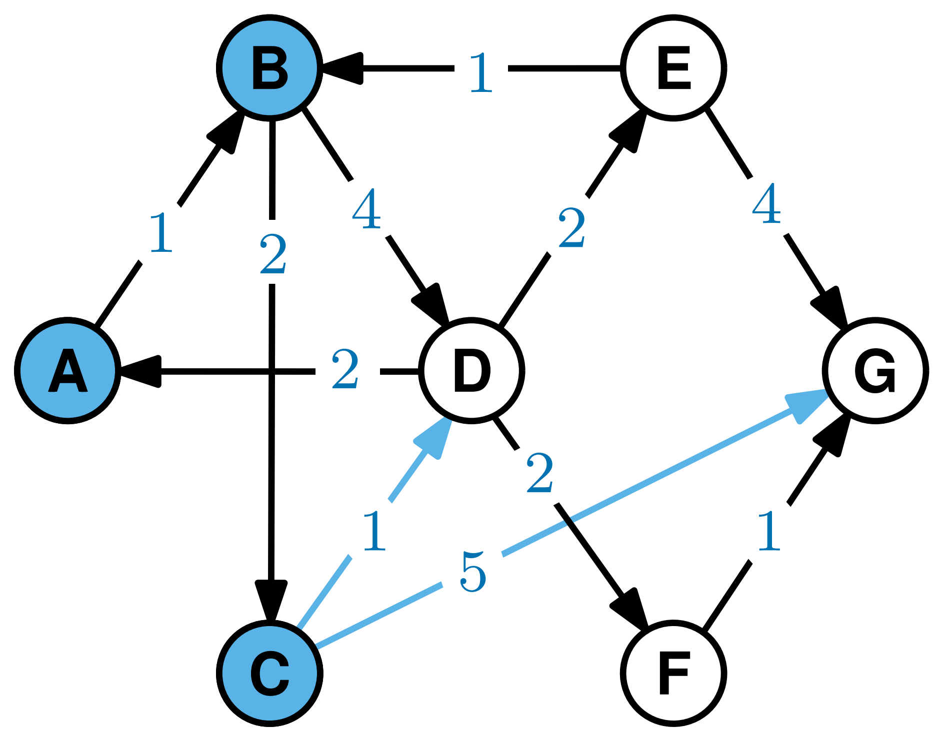
this is called relaxing edge (u, v)

We're going to simulate DIJKSTRA(A)

i.e. s = A

dist:

A	B	C	D	E	F	G
0	1	3	5	∞	∞	∞



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 = dist(v)

$\textsc{Dijkstra}(s)$

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

We're going to simulate

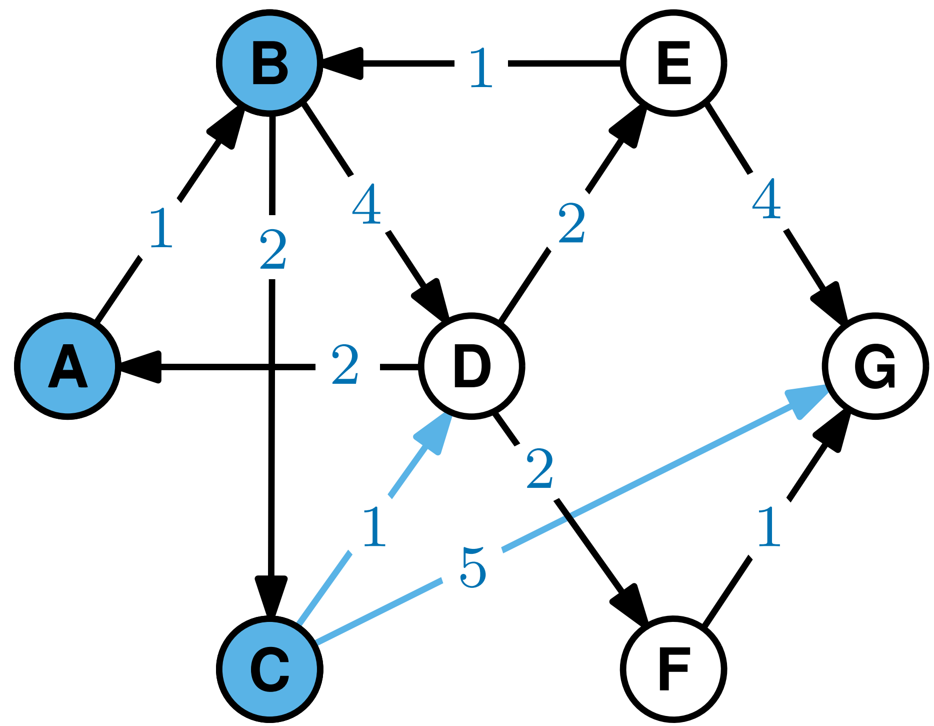
$\textsc{Dijkstra}(\text{A})$

i.e. $s = \text{A}$

this is called relaxing edge (u, v)

new path to **D** = $3 + 1 = 4$

	A	B	C	D	E	F	G
dist:	0	1	3	5	∞	∞	∞



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do INSERT(v, $\texttt{dist}(v)$)
While the queue is not empty
  u = EXTRACTMIN()
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      DECREASEKEY(v, $\texttt{dist}(v)$)

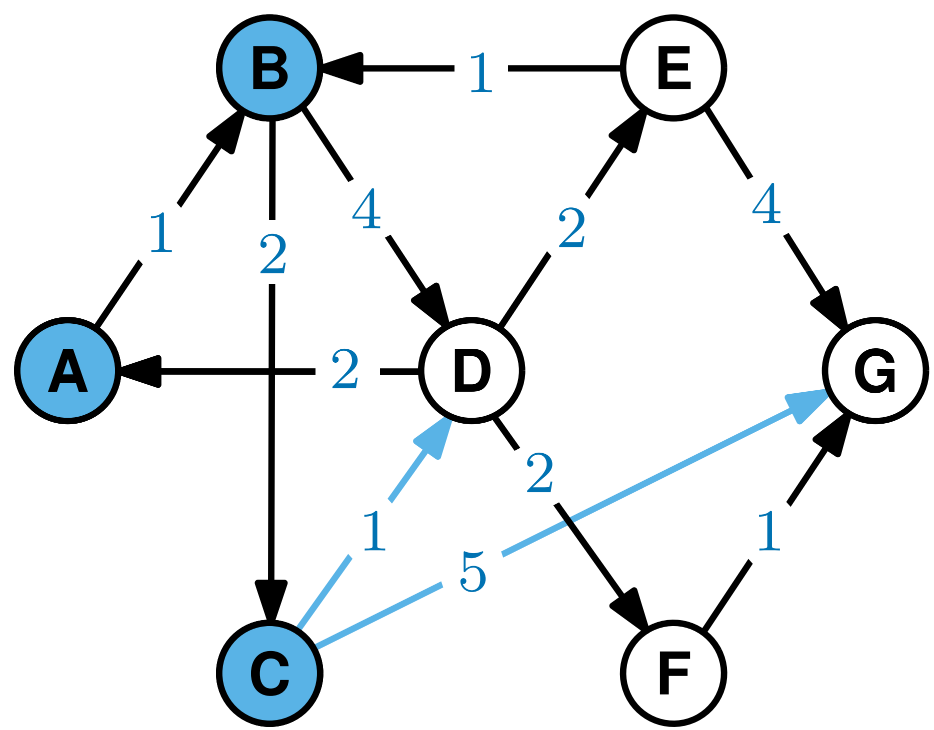
this is called relaxing edge (u, v)

We're going to simulate DIJKSTRA(A)

i.e. s = A

new path to **D** = $3 + 1 = 4$

	A	B	C	D	E	F	G
dist:	0	1	3	4	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

We're going to simulate

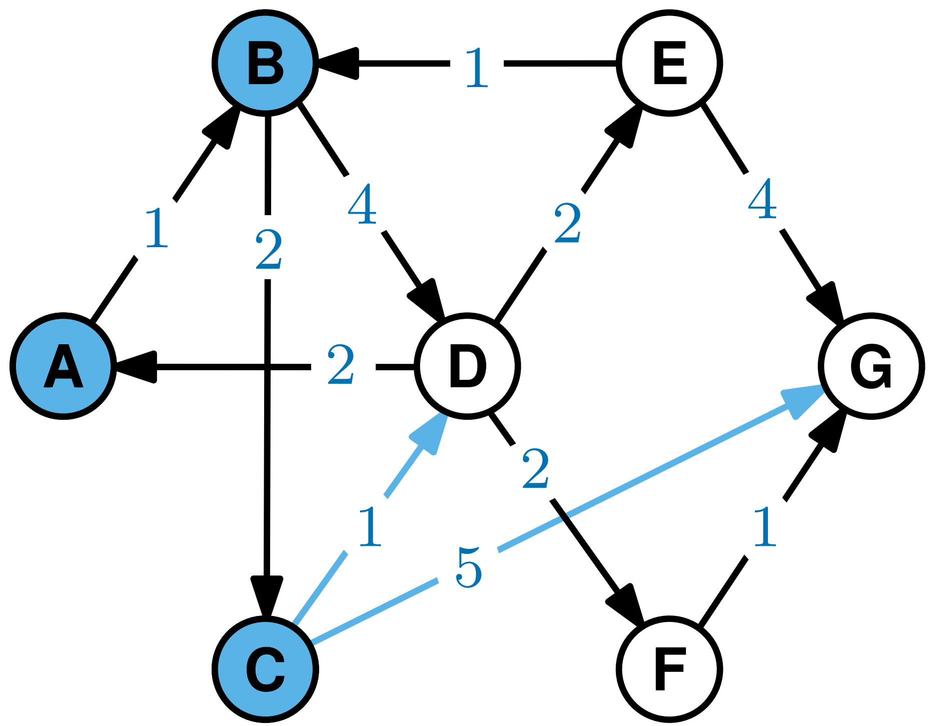
DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

new path to **G** = $3 + 5 = 8$

	A	B	C	D	E	F	G
dist:	0	1	3	4	∞	∞	∞



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

at all times, for each vertex v,

v.key $= \mathtt{dist}(v)$

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

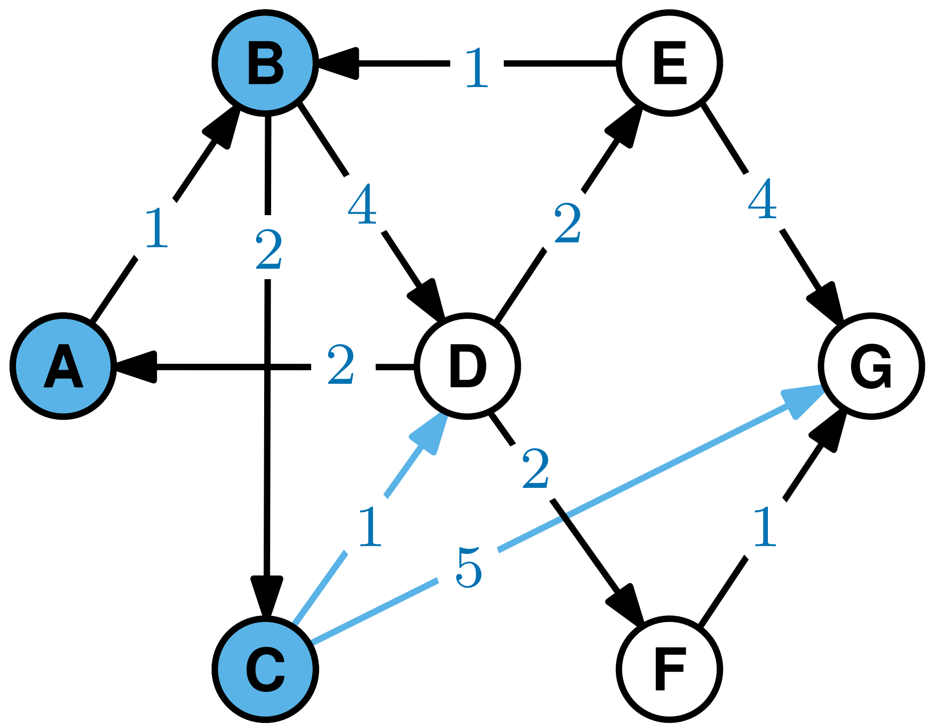
We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

new path to **G** = $3 + 5 = 8$

	A	B	C	D	E	F	G
dist:	0	1	3	4	∞	∞	8



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

at all times, for each vertex v,

v.key $= \mathtt{dist}(v)$

$\textsc{Dijkstra}(s)$

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

this is called relaxing edge (u, v)

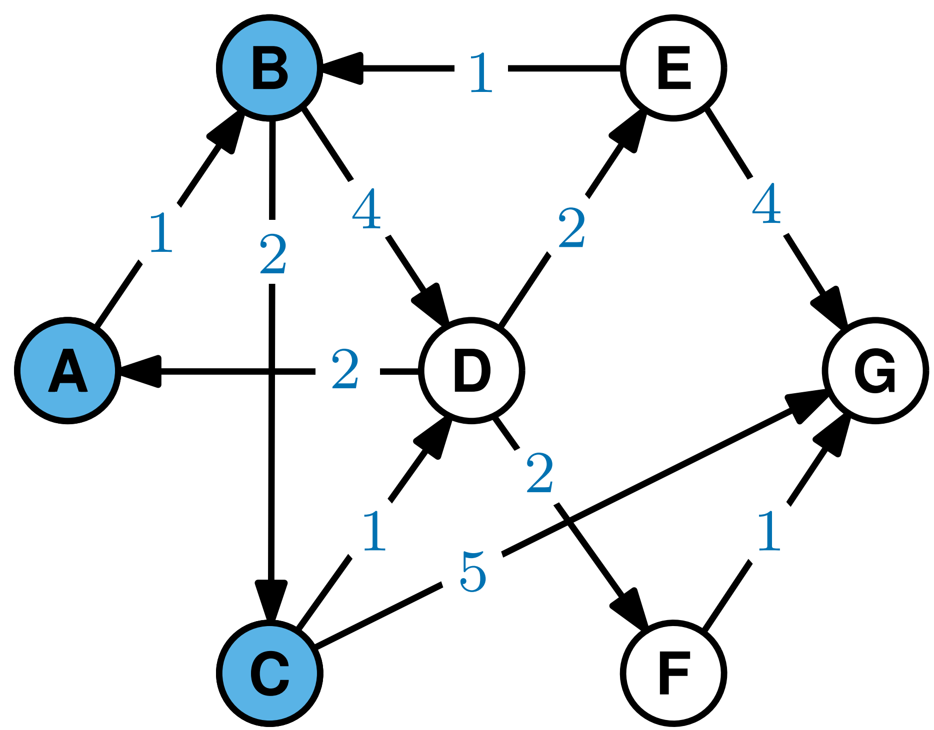
We're going to simulate

$\textsc{Dijkstra}(A)$

i.e. $s = A$

dist:

A	B	C	D	E	F	G
0	1	3	4	∞	∞	8



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

$\textsc{Dijkstra}(s)$

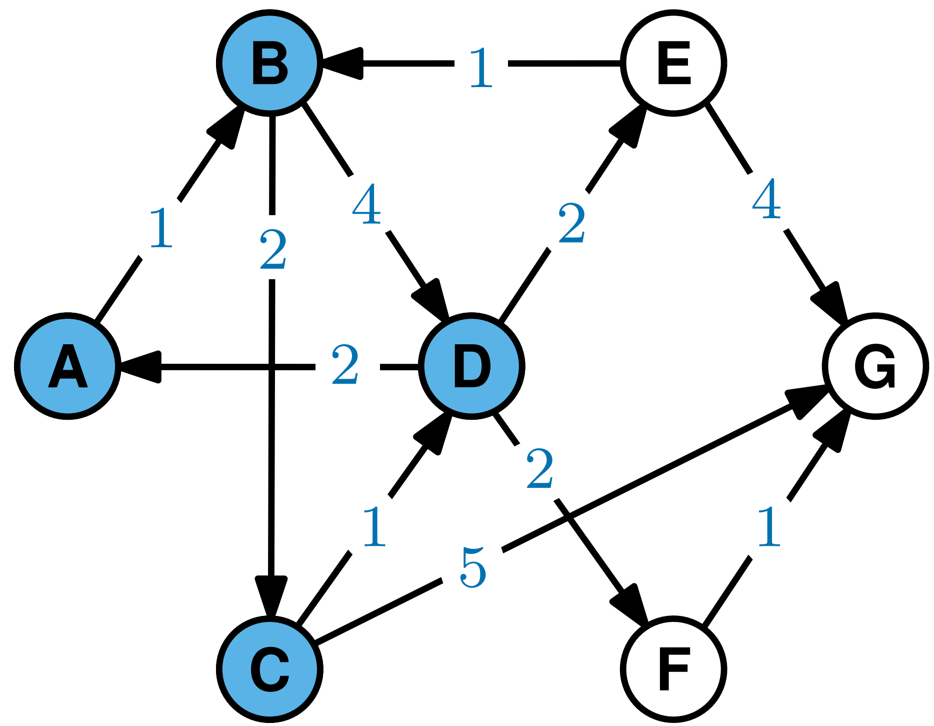
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

We're going to simulate $\textsc{Dijkstra}(A)$

i.e. $s =$ A

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	∞	∞	8



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

$\text{DIJKSTRA}(s)$

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

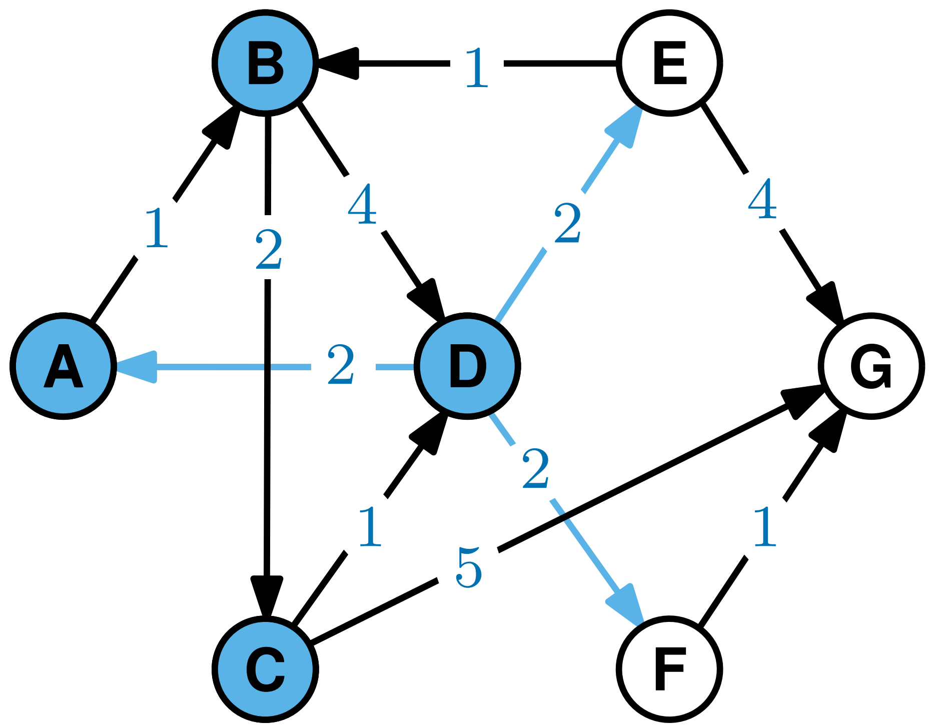
We're going to simulate

$\text{DIJKSTRA}(A)$

i.e. $s = A$

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	∞	∞	8



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

at all times, for each vertex v,

$v.\text{key} = \mathtt{dist}(v)$

University of BRISTOL

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

We're going to simulate DIJKSTRA(A)

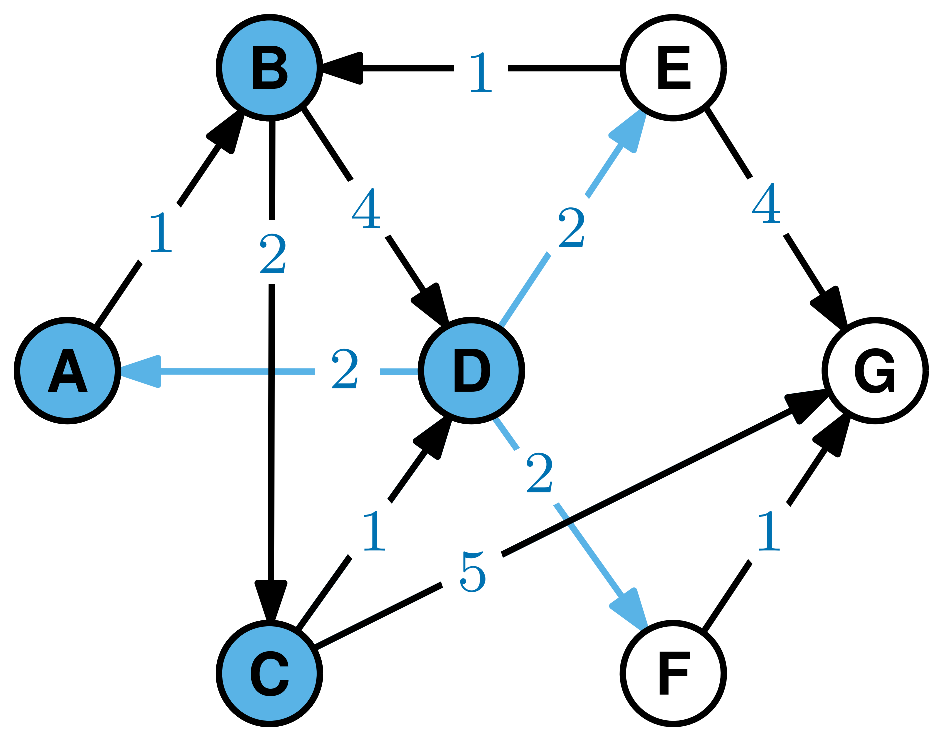
i.e. s = A

this is called relaxing edge (u, v)

new path to **A** = $0 + 2 = 2$

dist:

A	B	C	D	E	F	G
0	1	3	4	∞	∞	8



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

this is called relaxing edge (u, v)

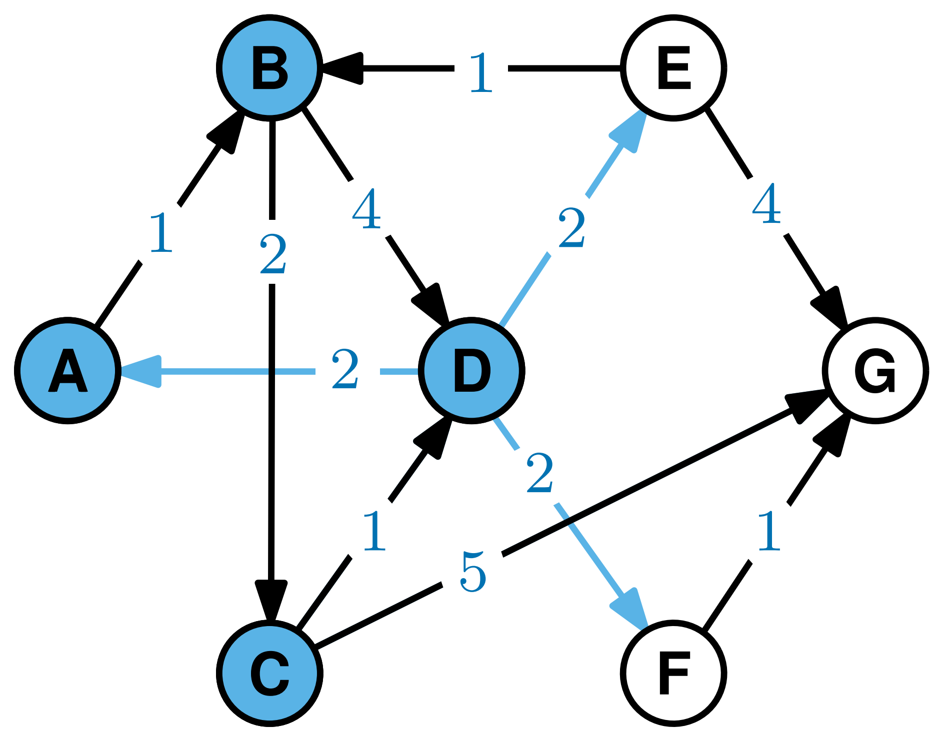
We're going to simulate

DIJKSTRA(A)

i.e. s = A

new path to **E** = $4 + 2 = 6$

	A	B	C	D	E	F	G
dist:	0	1	3	4	∞	∞	8



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

University of BRISTOL

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

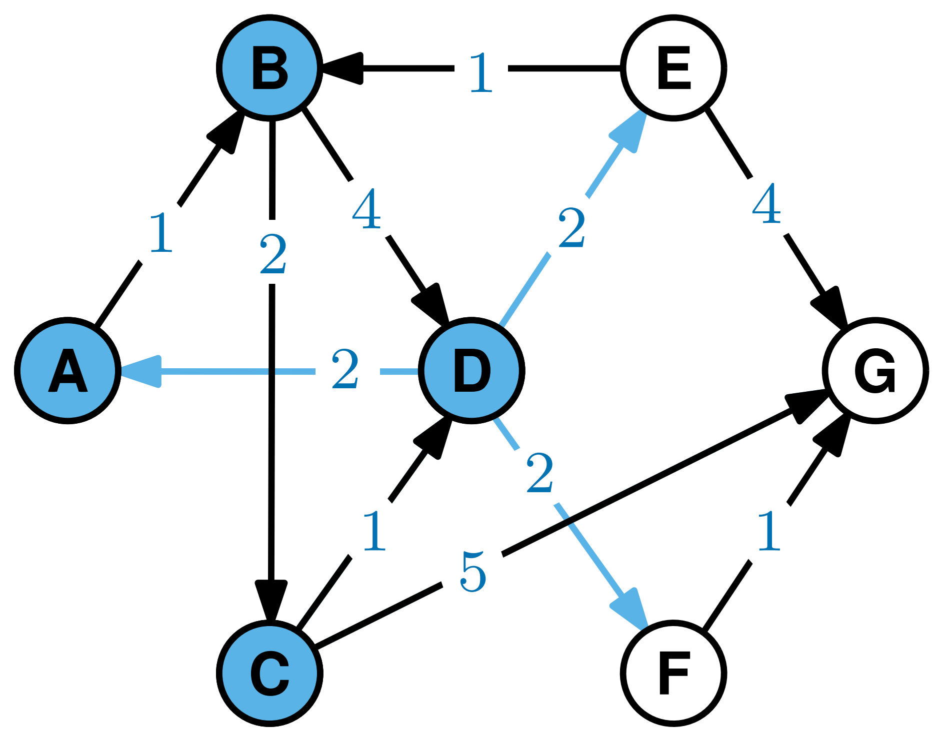
We're going to simulate DIJKSTRA(A)

i.e. $s = $ A

this is called relaxing edge (u, v)

new path to **E** = $4 + 2 = 6$

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	∞	8



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 = dist(v)

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

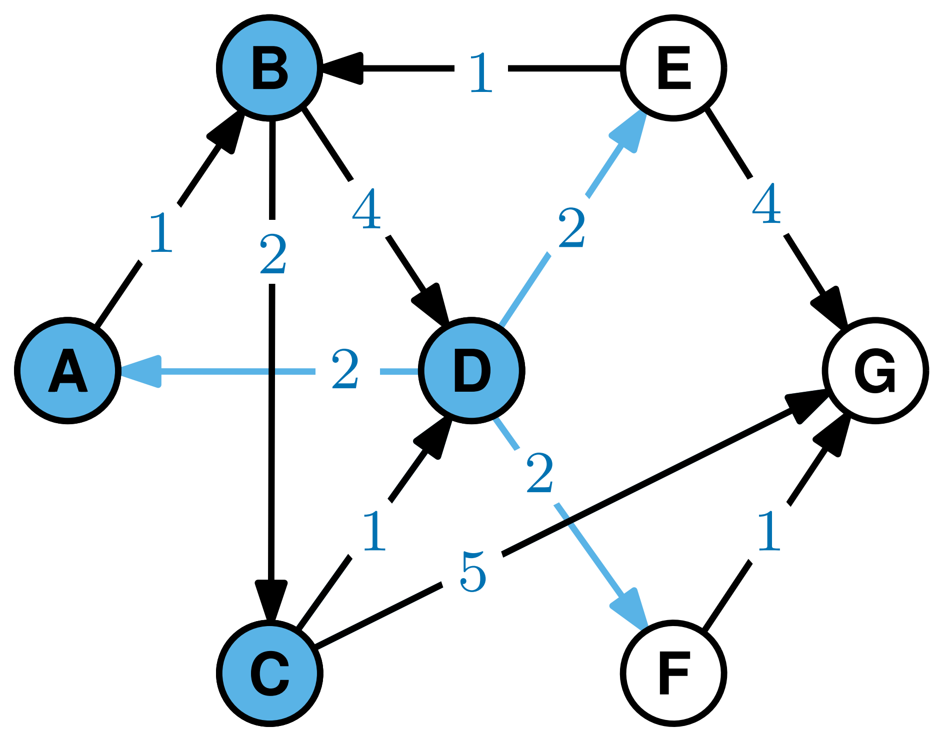
We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

new path to **F** = $4 + 2 = 6$

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	∞	8



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

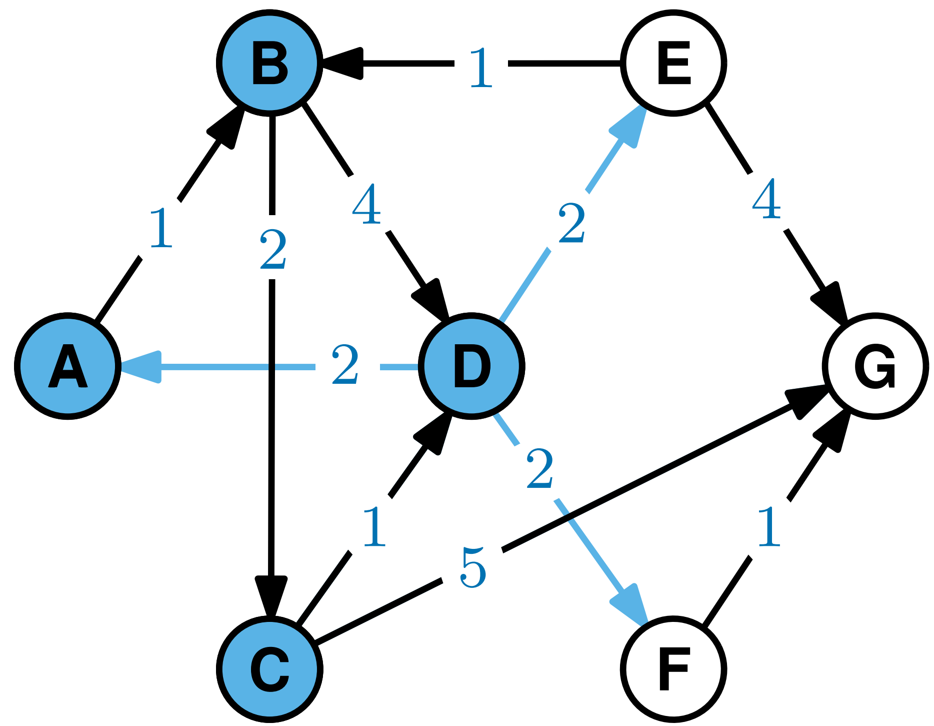
We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

new path to **F** = $4 + 2 = 6$

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

$\textsc{Dijkstra}(s)$

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

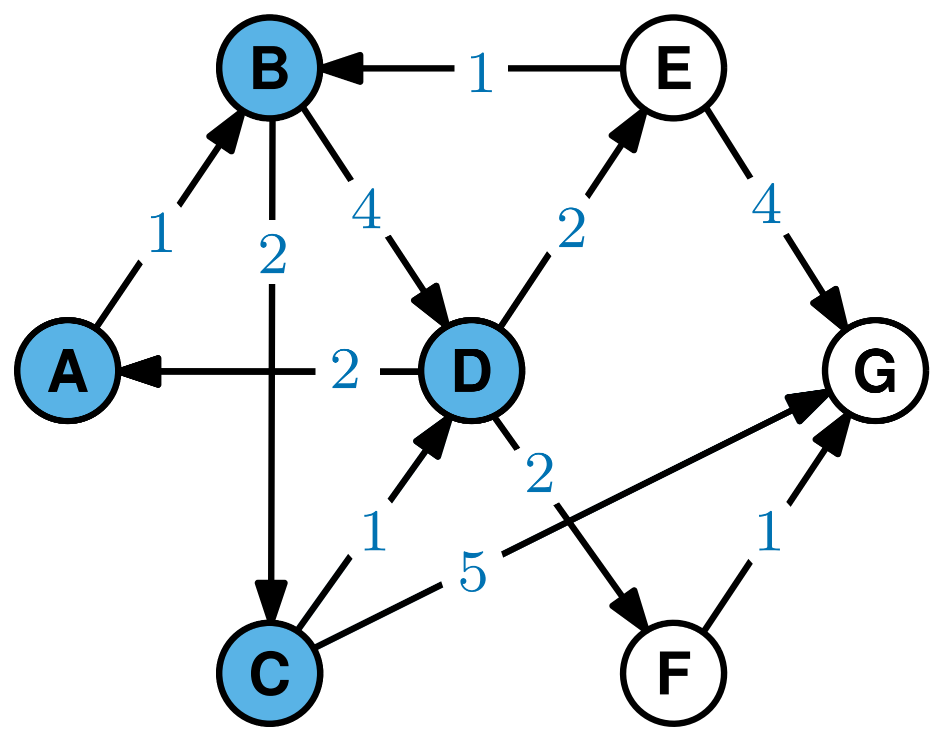
We're going to simulate $\textsc{Dijkstra}(\text{A})$

i.e. $s =$ A

this is called relaxing edge (u, v)

dist:

A	B	C	D	E	F	G
0	1	3	4	6	6	8



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

at all times, for each vertex v,

$$v.\text{key} = \texttt{dist}(v)$$

University of BRISTOL

DIJKSTRA(s)

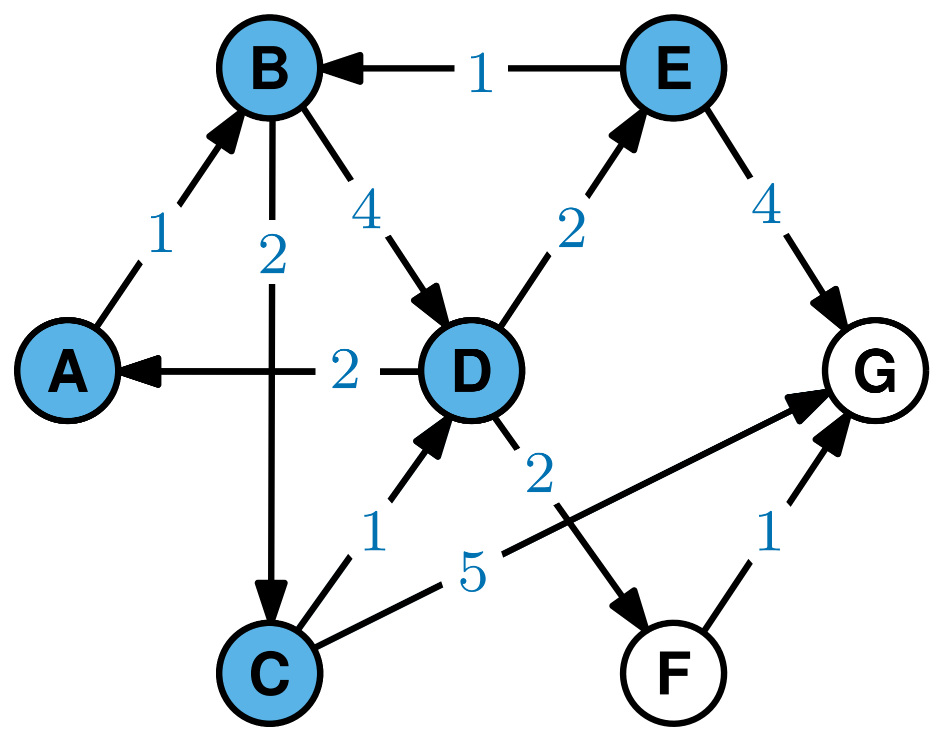
For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do INSERT$(v, \texttt{dist}(v))$
While the queue is not empty
  $u = $ EXTRACTMIN()
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      DECREASEKEY$(v, \texttt{dist}(v))$

We're going to simulate DIJKSTRA(A)

i.e. $s =$ A

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

DIJKSTRA(s)

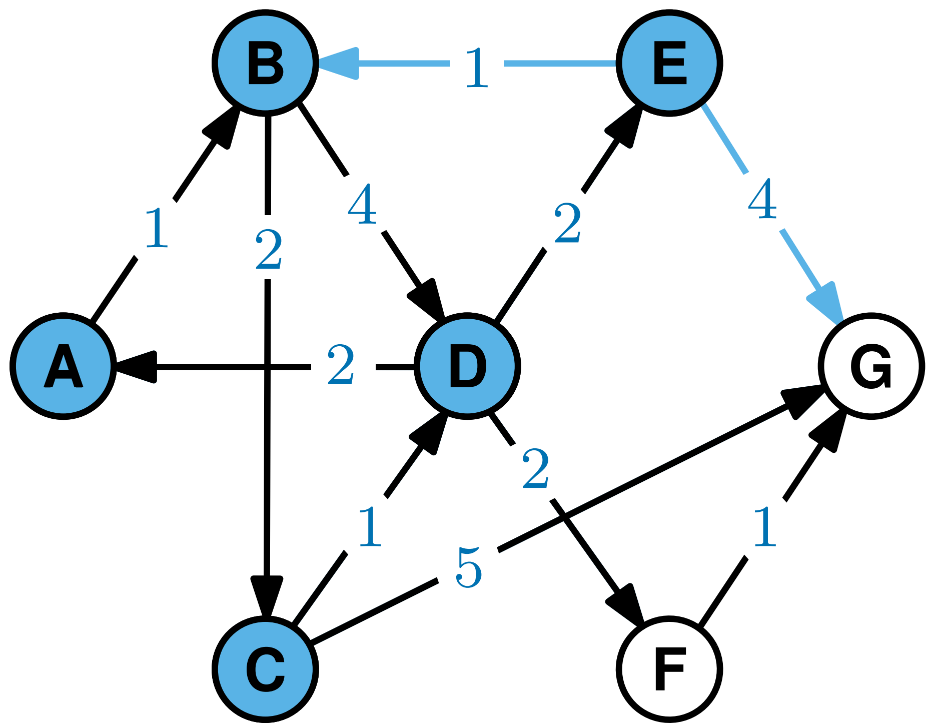
```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

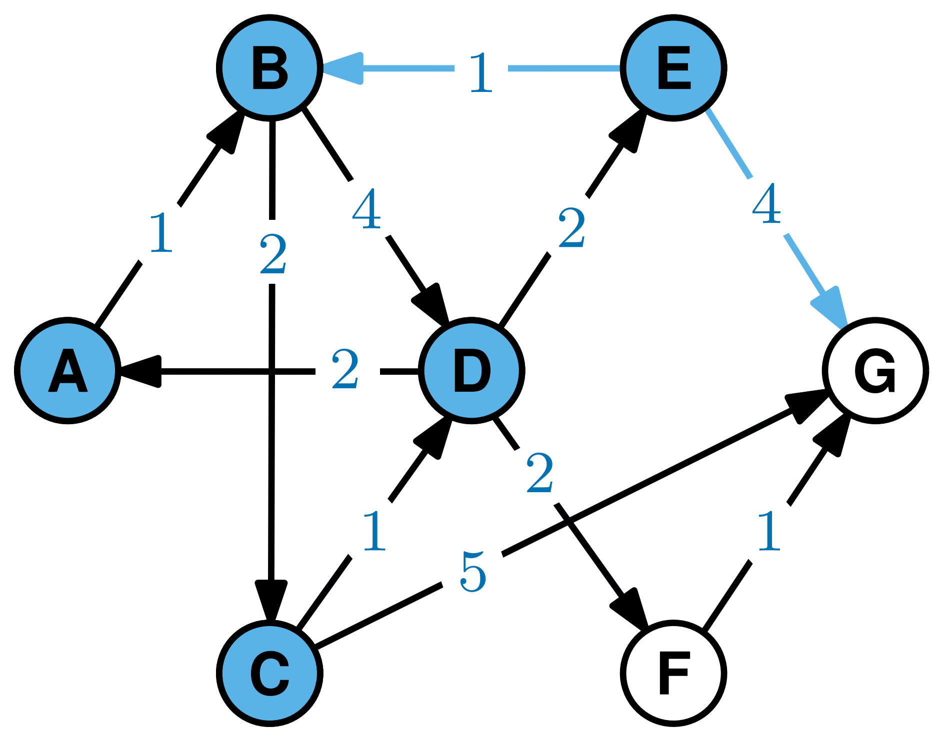
We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

new path to **B** = $6 + 1 = 7$

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

University of BRISTOL

DIJKSTRA(s)

For all v, set dist$(v) = \infty$
set dist$(s) = 0$
For each v, Do INSERT$(v,$ dist$(v))$
While the queue is not empty
  $u =$ EXTRACTMIN()
  For every edge $(u, v) \in E$
    If dist$(v) >$ dist$(u) +$ weight(u, v)
      dist$(v) =$ dist$(u) +$ weight(u, v)
      DECREASEKEY$(v,$ dist$(v))$

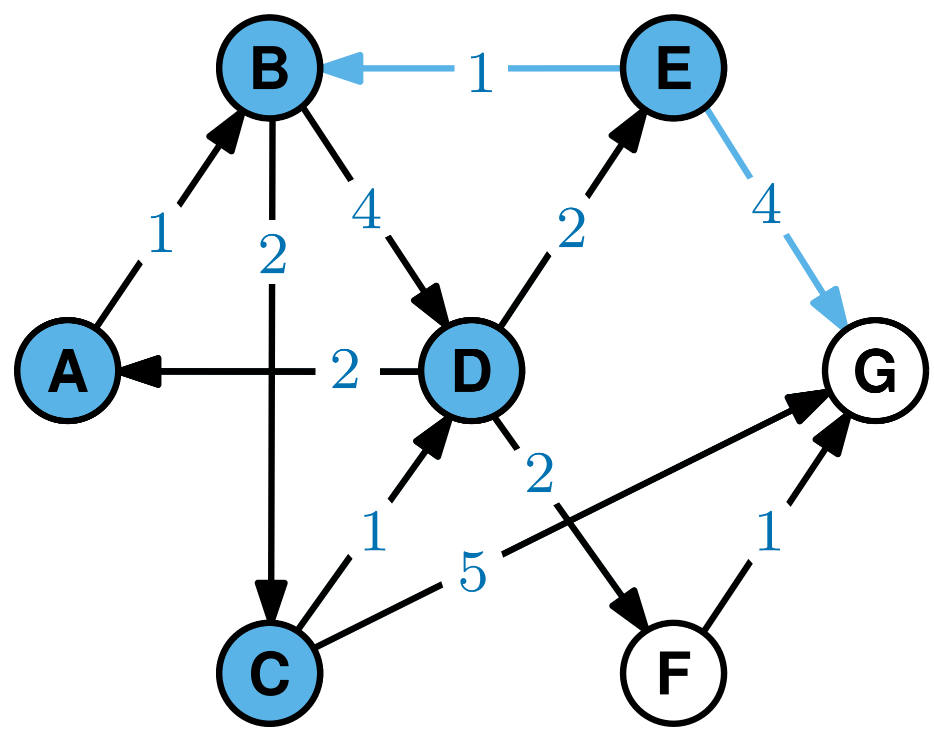
We're going to simulate DIJKSTRA(A)

i.e. $s =$ A

this is called relaxing edge (u, v)

new path to **G** = $6 + 4 = 10$

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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 $=$ dist(v)

DIJKSTRA(s)

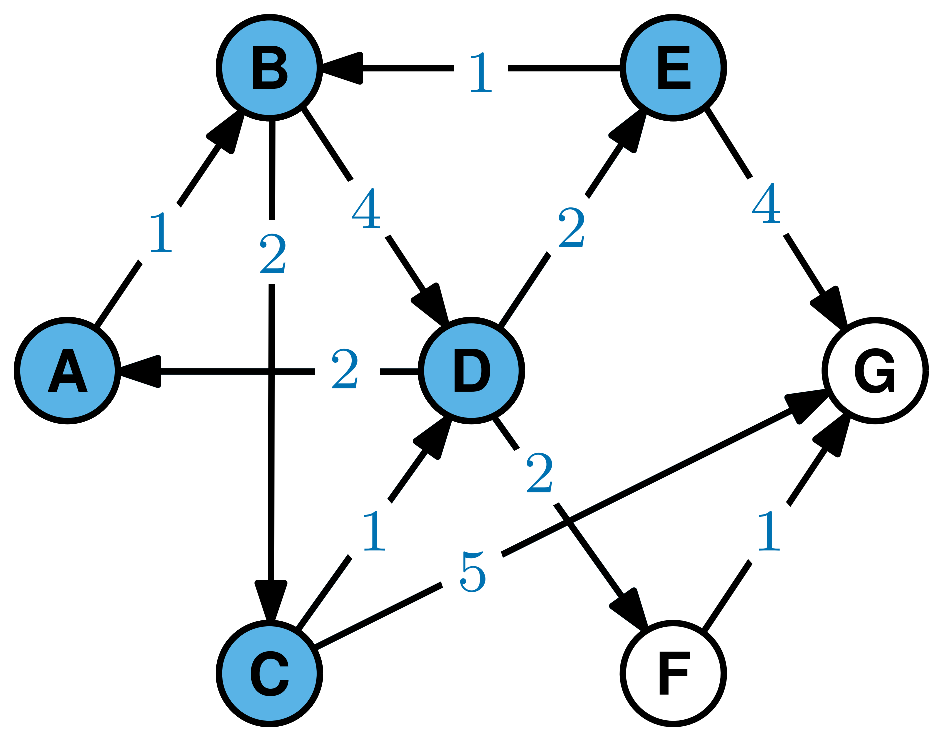
For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

Dijkstra(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do Insert(v, dist(v))
While the queue is not empty
   u = ExtractMin()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DecreaseKey(v, dist(v))
```

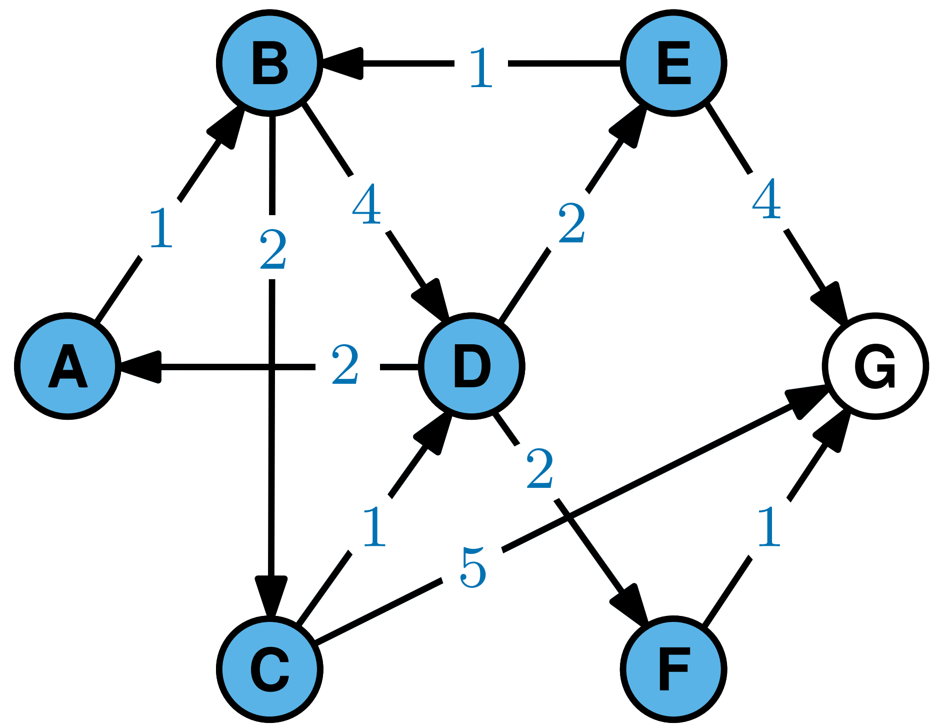
We're going to simulate

Dijkstra(A)

i.e. s = A

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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 = dist(v)

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

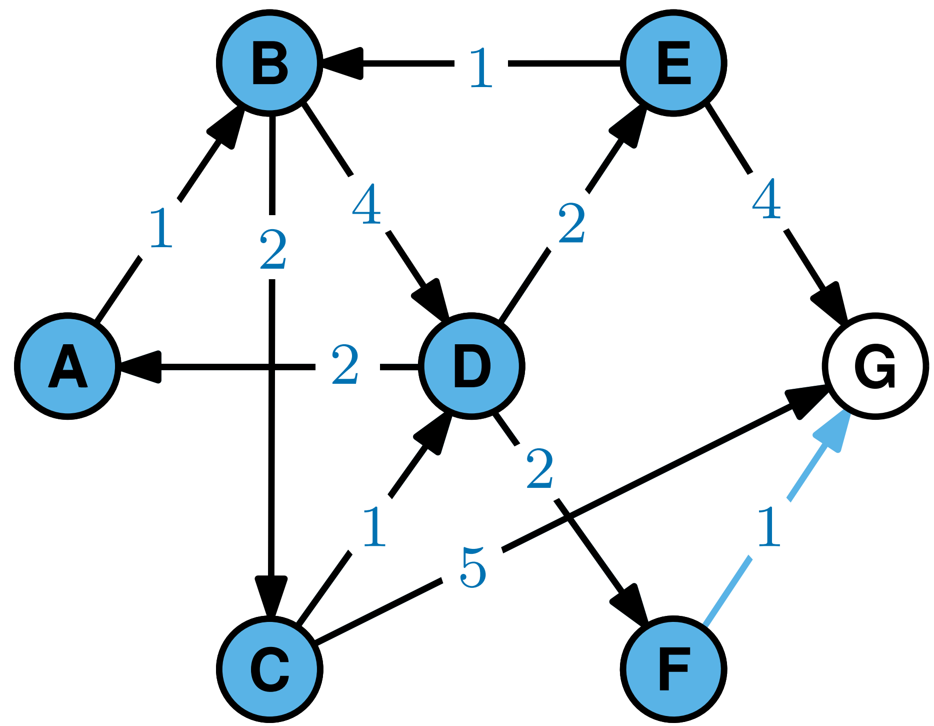
We're going to simulate

DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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

at all times, for each vertex v,

v.key $= \texttt{dist}(v)$

University of BRISTOL

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

We're going to simulate

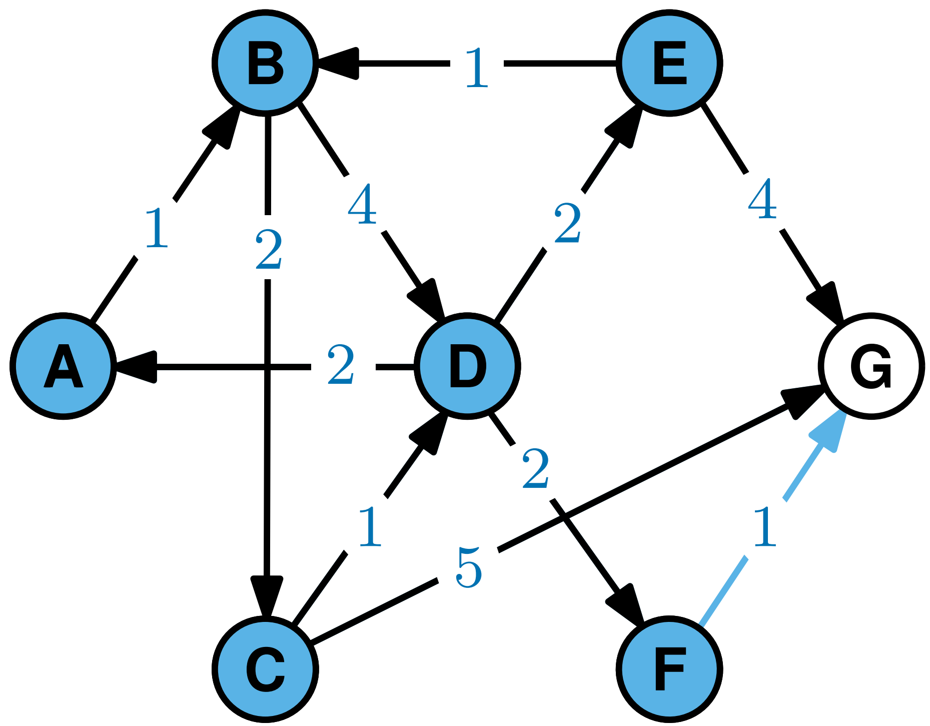
DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

new path to **G** = $6 + 1 = 7$

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	8



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the queue is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

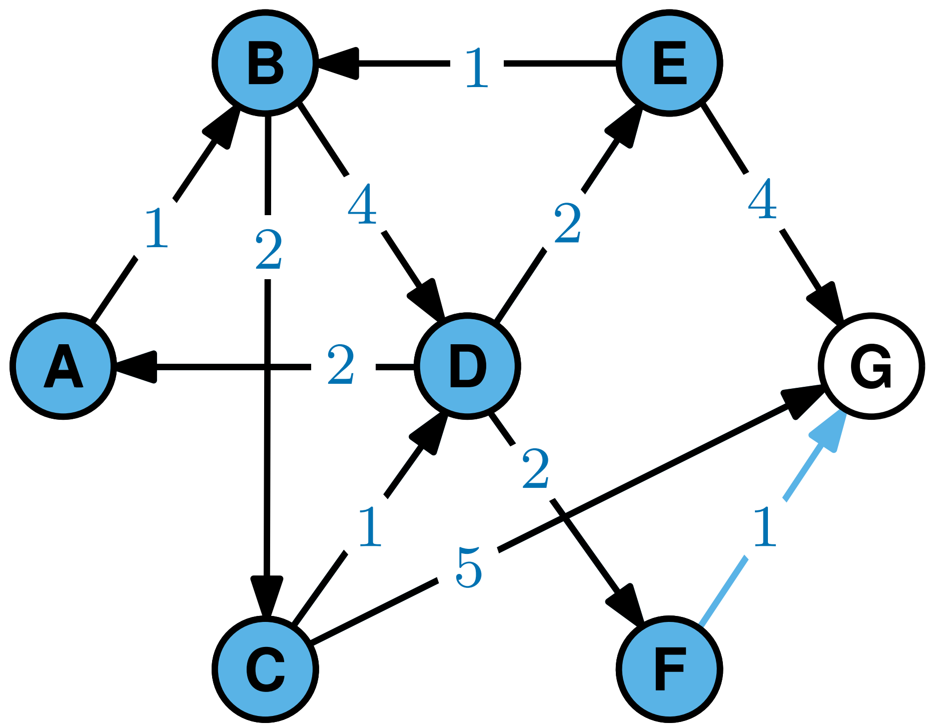
We're going to simulate DIJKSTRA(A)

i.e. $s =$ A

this is called relaxing edge (u, v)

new path to **G** = $6 + 1 = 7$

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	7



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

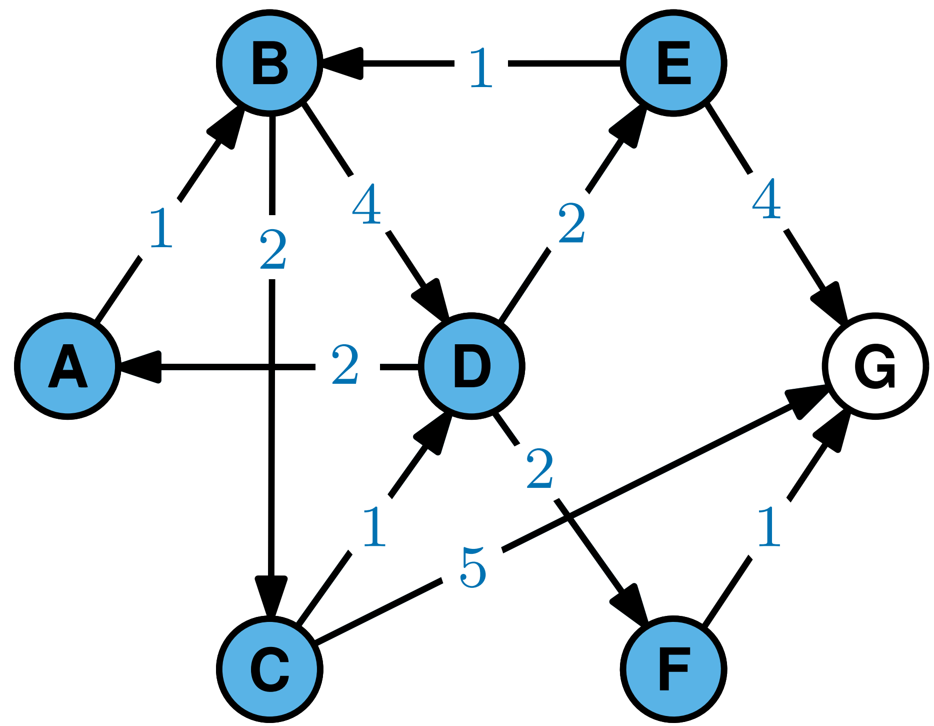
this is called relaxing edge (u, v)

We're going to simulate DIJKSTRA(A)

i.e. s = A

dist:

A	B	C	D	E	F	G
0	1	3	4	6	6	7



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

at all times, for each vertex v,

v.key = $\texttt{dist}(v)$

$\textsc{Dijkstra}(s)$

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u,v) ∈ E
      If dist(v) > dist(u) + weight(u,v)
         dist(v) = dist(u) + weight(u,v)
         DECREASEKEY(v, dist(v))
```

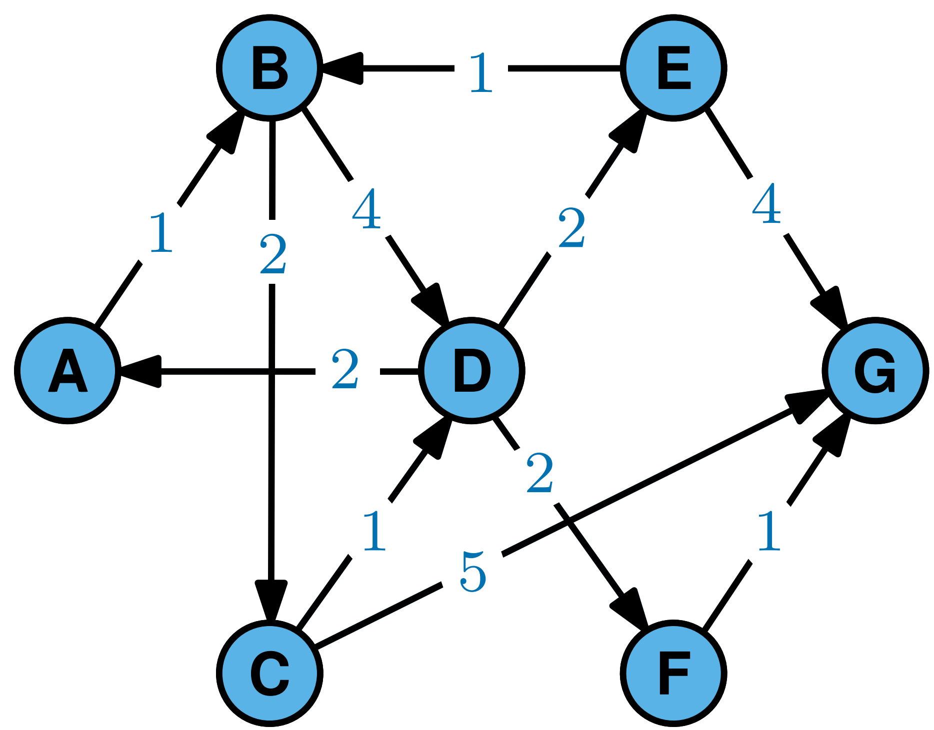
We're going to simulate

$\textsc{Dijkstra}(A)$

i.e. $s = A$

this is called relaxing edge (u, v)

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	7



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do $\textsc{Insert}(v, \texttt{dist}(v))$
While the $\texttt{queue}$ is not empty
  $u = \textsc{ExtractMin}()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\textsc{DecreaseKey}(v, \texttt{dist}(v))$

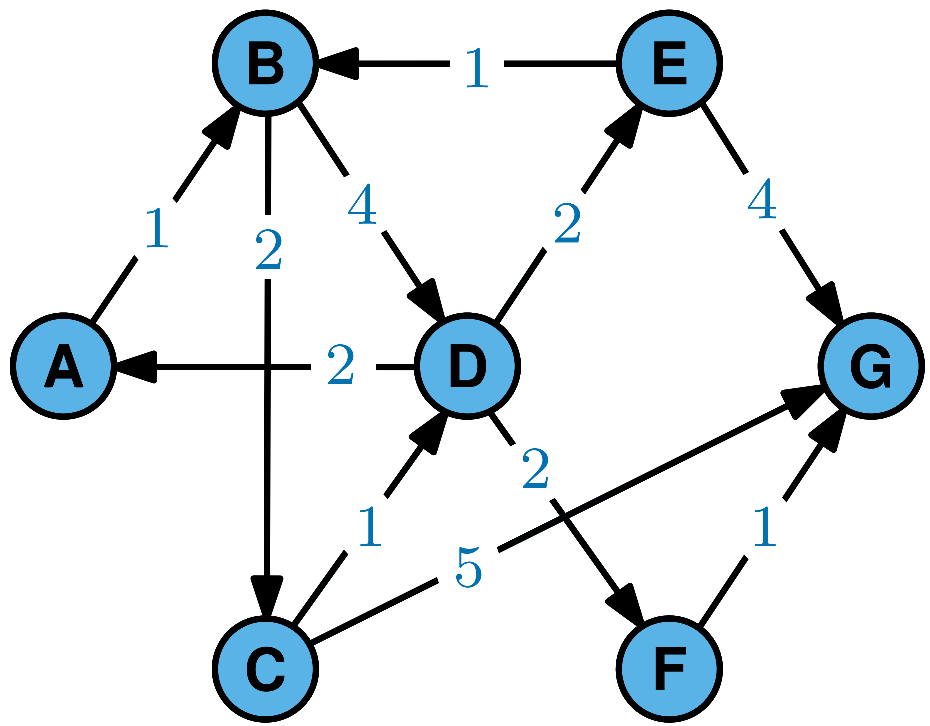
We're going to simulate DIJKSTRA(A)

i.e. s = A

this is called relaxing edge (u, v)

shortest paths from s = A:

	A	B	C	D	E	F	G
dist:	0	1	3	4	6	6	7



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

at all times, for each vertex v,

$v.\text{key} = \texttt{dist}(v)$

Proof of Correctness

Claim when Dijkstra's algorithm terminates, for each vertex v, $\mathtt{dist}(v) = \delta(s, v)$

where $\delta(s, v)$ is the *true* distance between s and v

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

Proof of Correctness

Claim when Dijkstra's algorithm terminates, for each vertex v, $\mathtt{dist}(v) = \delta(s, v)$

where $\delta(s, v)$ is the *true* distance between s and v

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

Observation At all times, $\mathtt{dist}(v)$ is the length of *some* path from s to v

(unless $\mathtt{dist}(v) = \infty$)

Therefore, for each vertex v, $\delta(s, v) \leqslant \mathtt{dist}(v)$

Proof of Correctness

Claim when Dijkstra's algorithm terminates, for each vertex v, $\texttt{dist}(v) = \delta(s, v)$

where $\delta(s, v)$ is the *true* distance between s and v

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

Proof of Correctness

Claim when Dijkstra's algorithm terminates, for each vertex v, $\texttt{dist}(v) = \delta(s, v)$

where $\delta(s, v)$ is the *true* distance between s and v

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

Further, observe that after a vertex v is EXTRACTED,

$\texttt{dist}(v)$ certainly doesn't increase

Proof of Correctness

Claim when Dijkstra's algorithm terminates, for each vertex v, $\mathtt{dist}(v) = \delta(s, v)$

where $\delta(s, v)$ is the *true* distance between s and v

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

Further, observe that after a vertex v is EXTRACTED,

$\mathtt{dist}(v)$ certainly doesn't increase

So we focus on proving that for all v,

when vertex v is EXTRACTED, $\mathtt{dist}(v) = \delta(s, v)$

Proof of Correctness

the true distance
between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

Proof of Correctness

the true distance *between s and v*

Lemma Whenever a vertex v is EXTRACTED, $\mathtt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\mathtt{dist}(v) \neq \delta(s, v)$

Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\mathtt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\mathtt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\mathtt{dist}(s) = 0 = \delta(s, s)$

Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

Proof of Correctness

the true distance
between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

There must be a path from s to v, otherwise

$$\texttt{dist}(v) = \infty = \delta(s, v)$$

Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

There must be a path from s to v, otherwise

$$\texttt{dist}(v) = \infty = \delta(s, v)$$

(Dijkstra doesn't find paths that aren't there)

Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\mathtt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\mathtt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\mathtt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

There must be a path from s to v, otherwise

$$\mathtt{dist}(v) = \infty = \delta(s, v)$$

(Dijkstra doesn't find paths that aren't there)

Consider the point in the algorithm immediately before v is EXTRACTED

Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\mathtt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\mathtt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\mathtt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

There must be a path from s to v, otherwise

$$\mathtt{dist}(v) = \infty = \delta(s, v)$$

(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:

Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

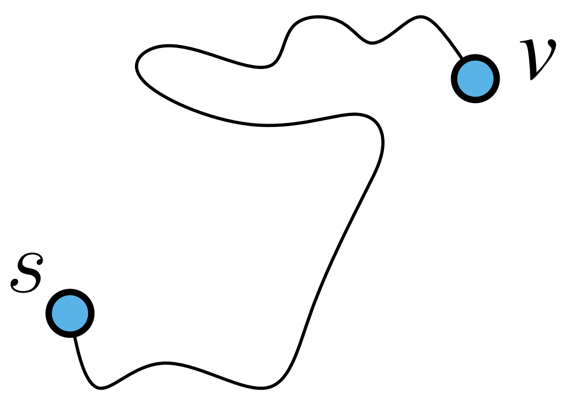
There must be a path from s to v, otherwise

$$\texttt{dist}(v) = \infty = \delta(s, v)$$

(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:



Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

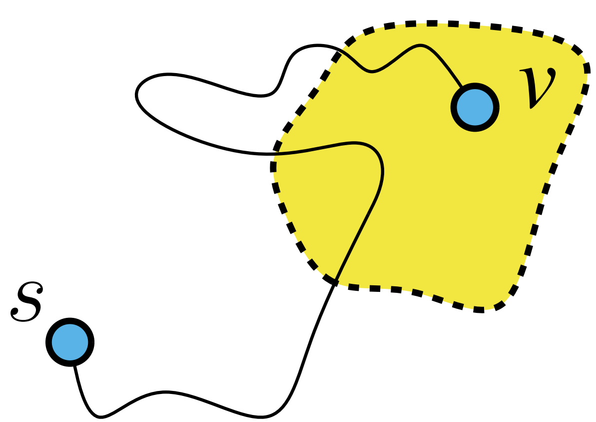
There must be a path from s to v, otherwise

$$\texttt{dist}(v) = \infty = \delta(s, v)$$

(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:



Proof of Correctness

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

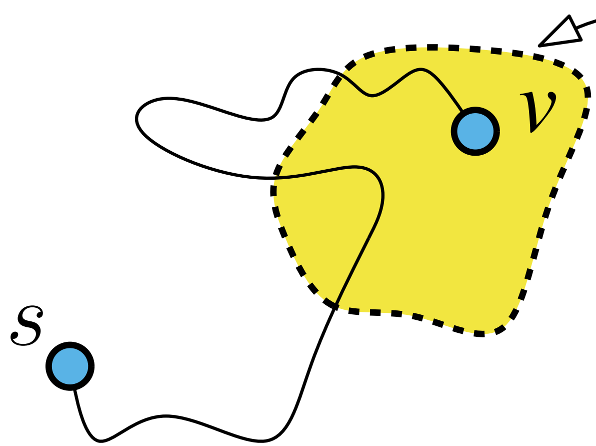
There must be a path from s to v, otherwise

$$\texttt{dist}(v) = \infty = \delta(s, v)$$

(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

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

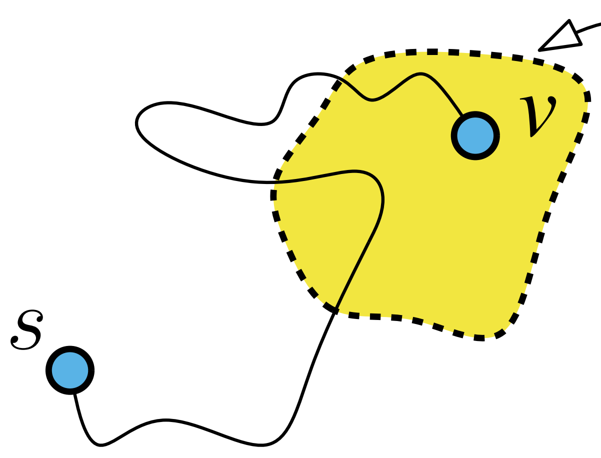
There must be a path from s to v, otherwise

$$\texttt{dist}(v) = \infty = \delta(s, v)$$

(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

the true distance between s and v

Lemma Whenever a vertex v is EXTRACTED, $\texttt{dist}(v) = \delta(s, v)$

Proof

To derive a contradiction, we let v be the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

v cannot be the source, s because $\texttt{dist}(s) = 0 = \delta(s, s)$

(this is in the algorithm description)

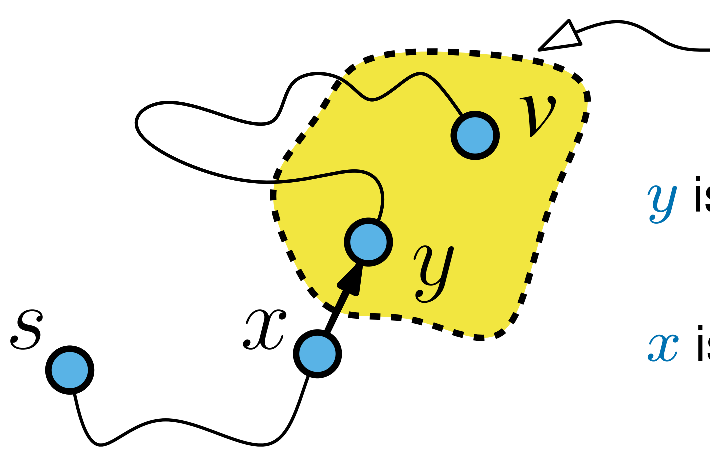
There must be a path from s to v, otherwise

$$\texttt{dist}(v) = \infty = \delta(s, v)$$

(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)

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)

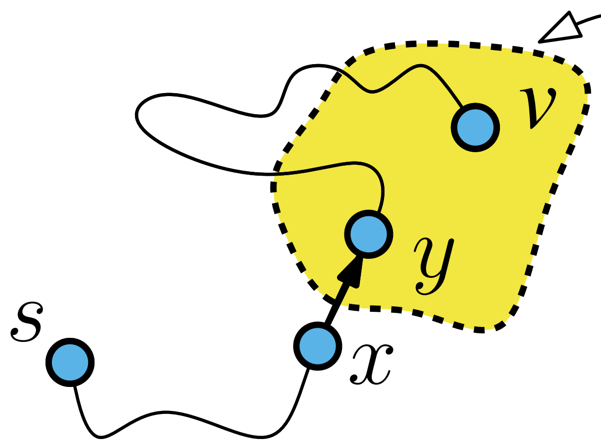
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

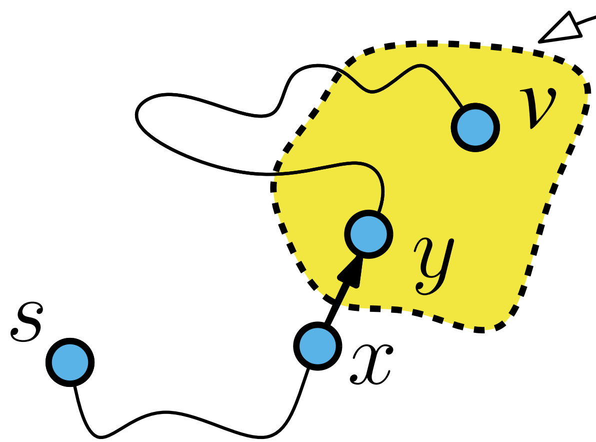
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s,v)$

the true distance between s and 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

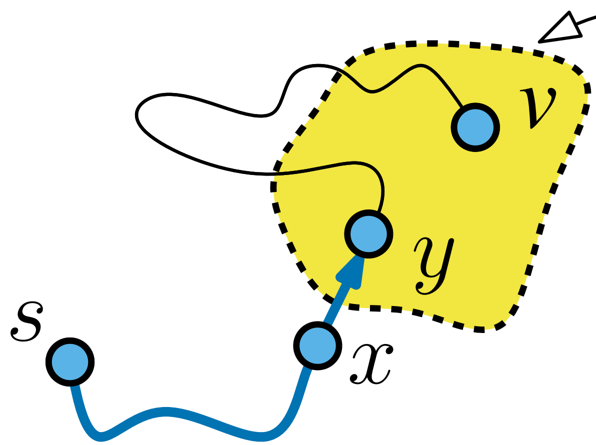
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

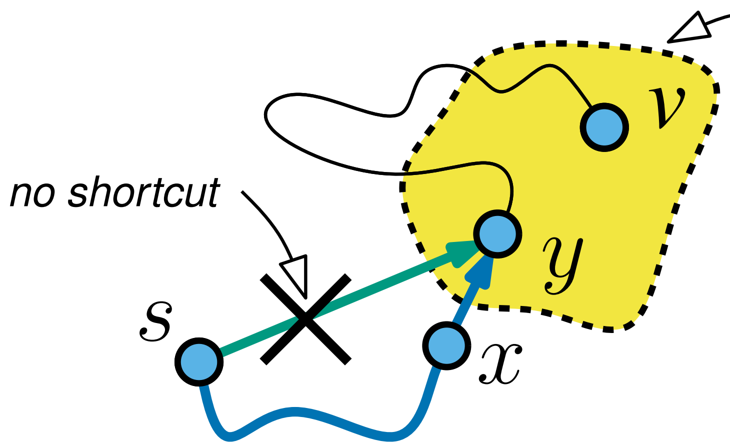
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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

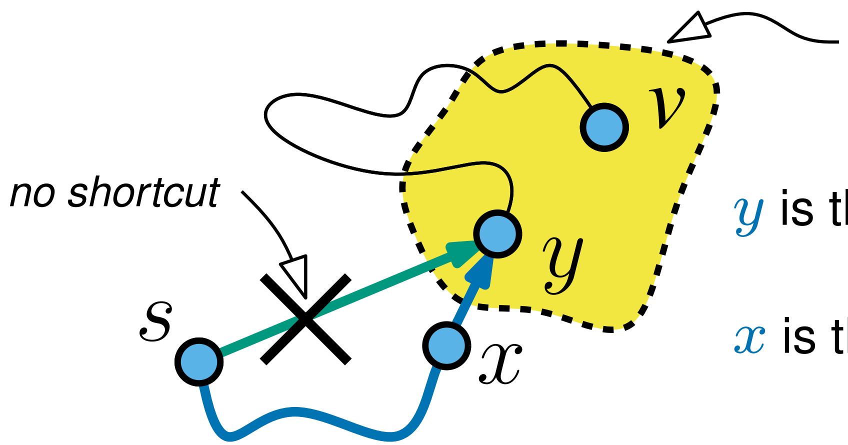
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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)*

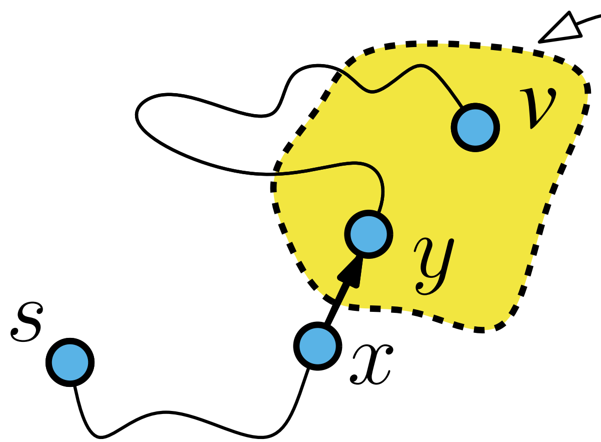
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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)*

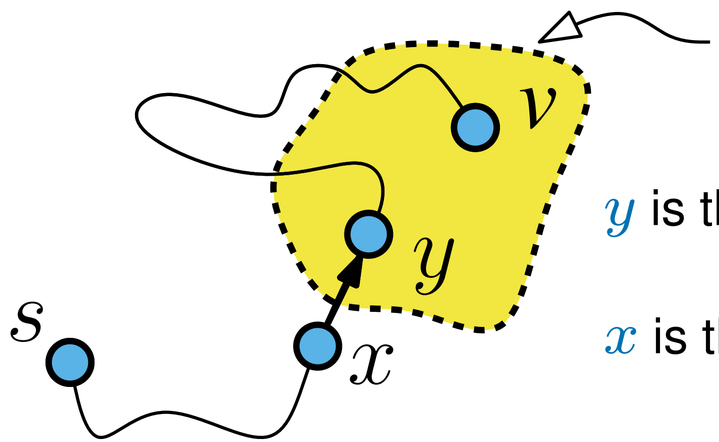
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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, $\delta(s, y) \leqslant \delta(s, v)$

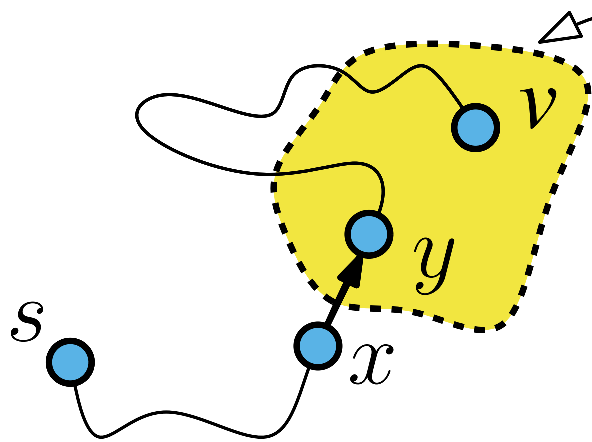
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\mathtt{dist}(v) \neq \delta(s, v)$

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, $\delta(s, y) \leqslant \delta(s, v)$

The vertex x is EXTRACTED from the queue *before* v

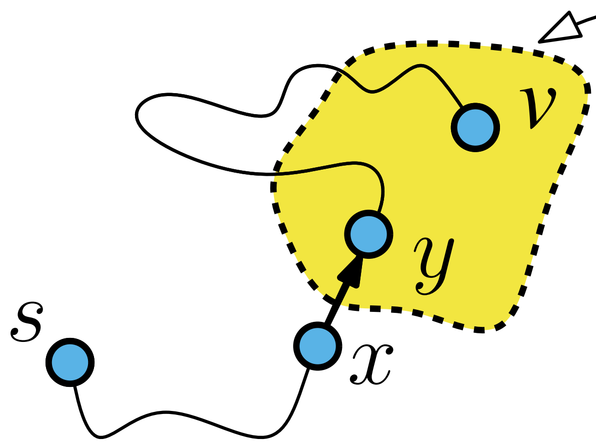
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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, $\delta(s, y) \leqslant \delta(s, v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s, x)$

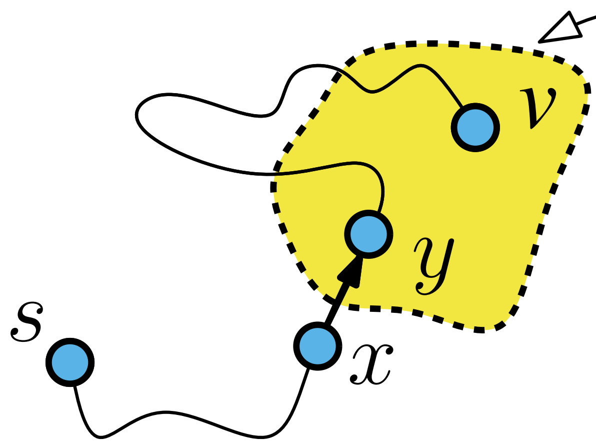
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s,v)$

the true distance
between s and 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, $\delta(s,y) \leqslant \delta(s,v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s,x)$ *(v is the first vertex EXTRACTED with the wrong distance)*

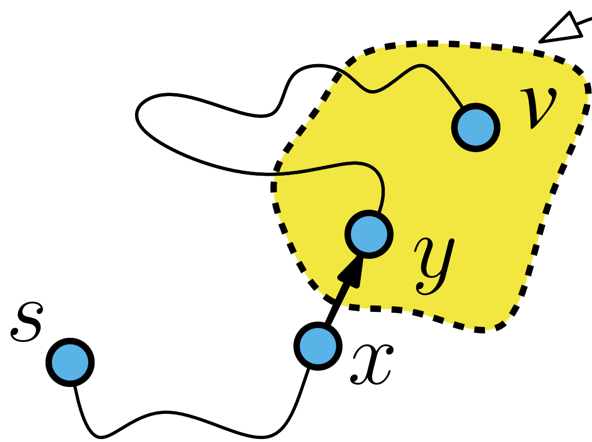
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s,v)$

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, $\delta(s,y) \leqslant \delta(s,v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s,x)$ *(v is the first vertex EXTRACTED with the wrong distance)*

Further, when x was EXTRACTED we relaxed edge (x,y)

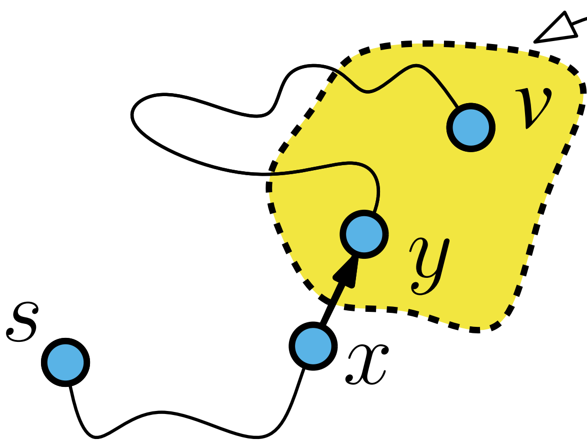
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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, $\delta(s, y) \leqslant \delta(s, v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s, x)$ *(v is the first vertex EXTRACTED with the wrong distance)*

Further, when x was EXTRACTED we relaxed edge (x, y)

therefore $\texttt{dist}(y) \leqslant \texttt{dist}(x) + \texttt{weight}(x, y)$

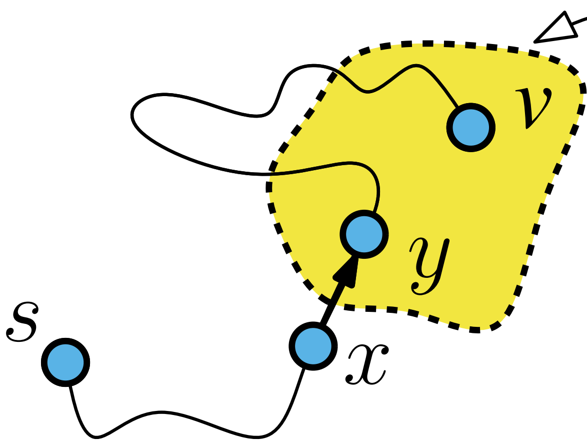
Proof of Correctness

v is the first vertex to be EXTRACTED with $\mathtt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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, $\delta(s, y) \leqslant \delta(s, v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\mathtt{dist}(x) = \delta(s, x)$ *(v is the first vertex EXTRACTED with the wrong distance)*

Further, when x was EXTRACTED we relaxed edge (x, y)

therefore $\mathtt{dist}(y) \leqslant \mathtt{dist}(x) + \mathtt{weight}(x, y)$

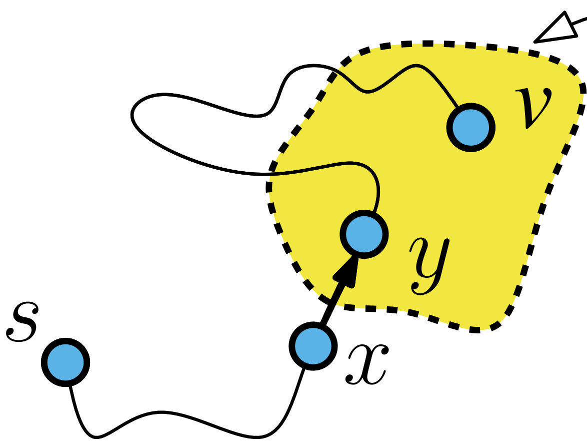
Proof of Correctness

v is the first vertex to be Extracted with $\texttt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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, $\delta(s, y) \leqslant \delta(s, v)$

The vertex x is Extracted from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s, x)$ *(v is the first vertex Extracted with the wrong distance)*

Further, when x was Extracted we relaxed edge (x, y)

therefore $\texttt{dist}(y) \leqslant \delta(s, x) + \texttt{weight}(x, y)$

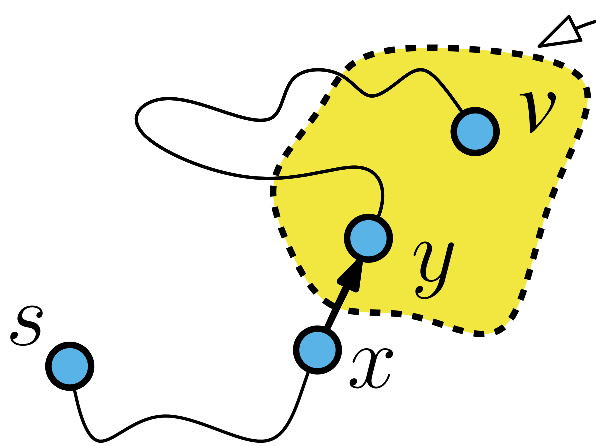
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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, $\delta(s, y) \leqslant \delta(s, v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s, x)$ *(v is the first vertex EXTRACTED with the wrong distance)*

Further, when x was EXTRACTED we relaxed edge (x, y)

therefore $\texttt{dist}(y) \leqslant \delta(s, x) + \texttt{weight}(x, y)$

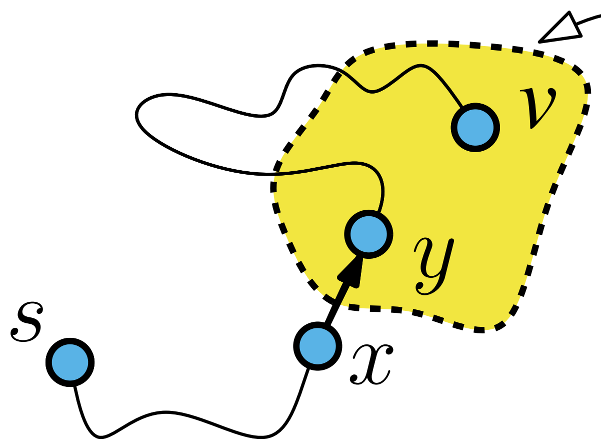
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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, $\delta(s, y) \leqslant \delta(s, v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s, x)$ *(v is the first vertex EXTRACTED with the wrong distance)*

Further, when x was EXTRACTED we relaxed edge (x, y)

therefore $\texttt{dist}(y) \leqslant \delta(s, x) + \texttt{weight}(x, y) = \delta(s, y)$

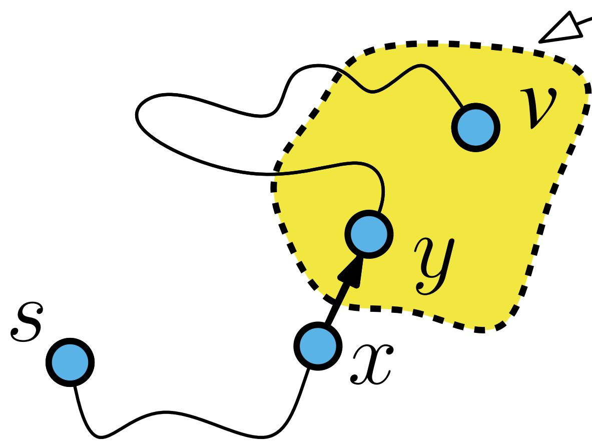
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s,v)$

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, $\delta(s,y) \leqslant \delta(s,v)$

The vertex x is EXTRACTED from the queue *before* v

therefore $\texttt{dist}(x) = \delta(s,x)$ *(v is the first vertex EXTRACTED with the wrong distance)*

Further, when x was EXTRACTED we relaxed edge (x,y)

therefore $\texttt{dist}(y) \leqslant \delta(s,x) + \texttt{weight}(x,y) = \delta(s,y)$ *(the path shown is shortest)*

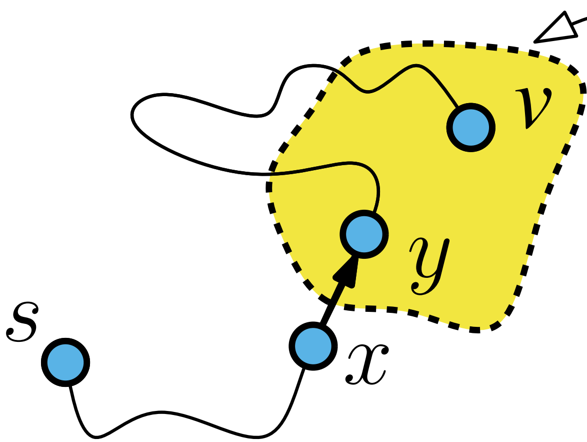
Proof of Correctness

the true distance
between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s,v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s,y)$ and $\delta(s,y) \leqslant \delta(s,v)$

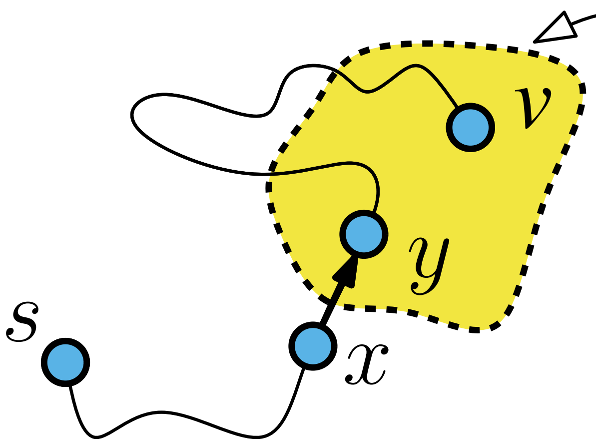
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

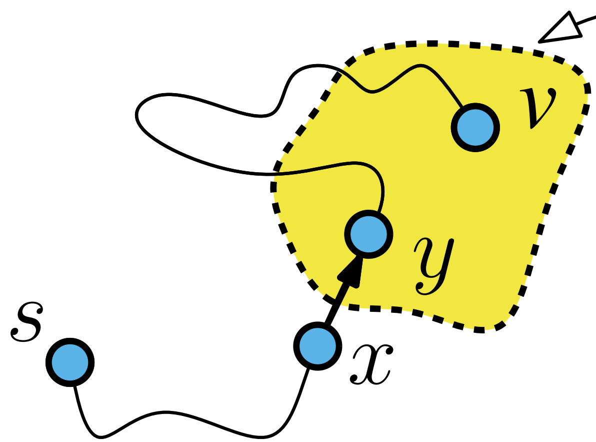
Proof of Correctness

the true *distance between* s *and* v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

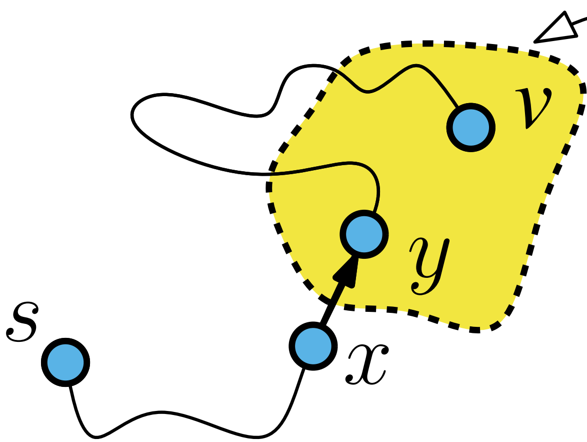
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum $\texttt{dist}$)*

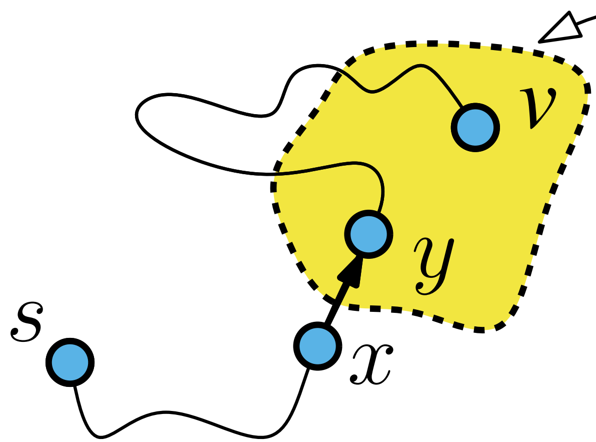
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s,v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s,y)$ and $\delta(s,y) \leqslant \delta(s,v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum* $\texttt{dist}$*)*

Putting it all together, $\texttt{dist}(v) \leqslant \texttt{dist}(y) \leqslant \delta(s,y) \leqslant \delta(s,v)$

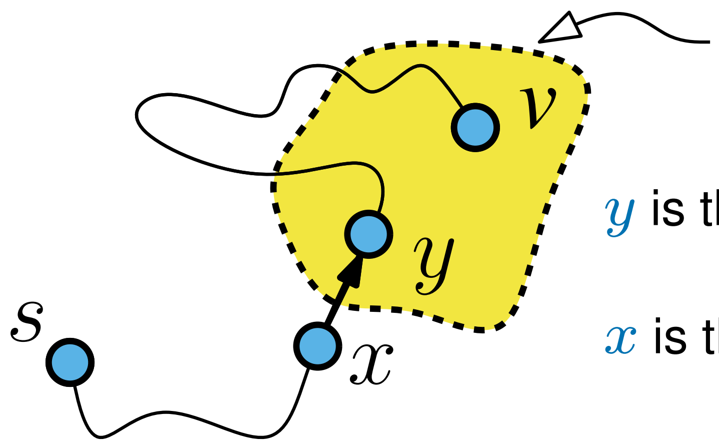
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum* $\texttt{dist}$*)*

Putting it all together, $\texttt{dist}(v) \leqslant \texttt{dist}(y) \leqslant \delta(s, y) \leqslant \delta(s, v)$

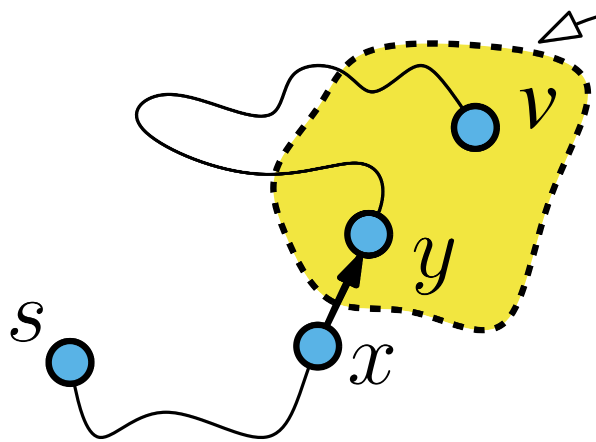
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum* $\texttt{dist}$*)*

Putting it all together, $\texttt{dist}(v) \leqslant \texttt{dist}(y) \leqslant \delta(s, y) \leqslant \delta(s, v)$

University of BRISTOL

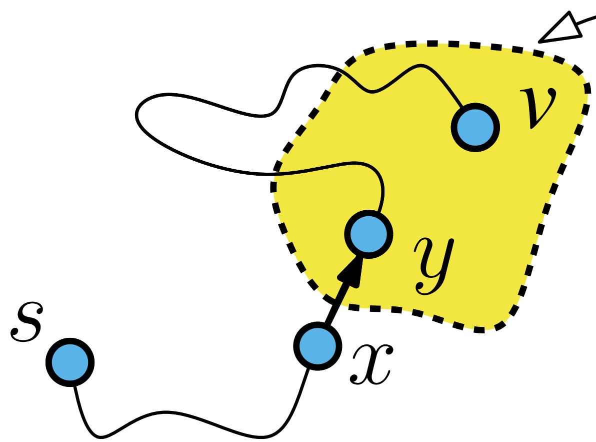
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum $\texttt{dist}$)*

Putting it all together, $\texttt{dist}(v) \leqslant \texttt{dist}(y) \leqslant \delta(s, y) \leqslant \delta(s, v)$

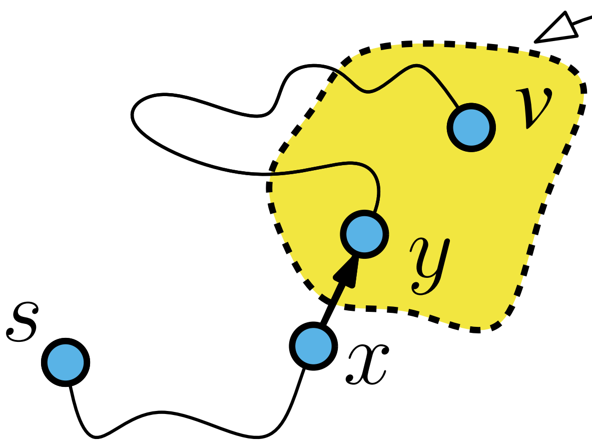
Proof of Correctness

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

the true distance between s and 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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum* $\texttt{dist}$*)*

Putting it all together, $\texttt{dist}(v) \leqslant \texttt{dist}(y) \leqslant \delta(s, y) \leqslant \delta(s, v)$

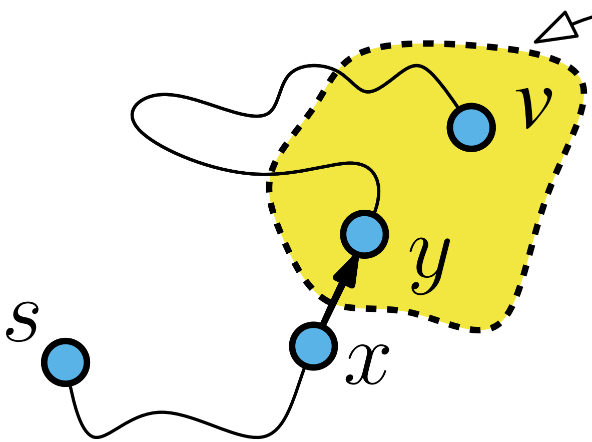
Proof of Correctness

the true distance between s and v

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum* $\texttt{dist}$*)*

Putting it all together, $\texttt{dist}(v)$ $\leqslant \delta(s, v)$

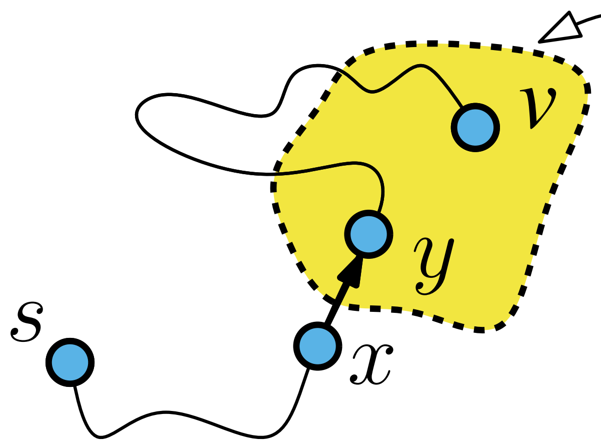
Proof of Correctness

the true *distance* *between s and v*

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

We have shown that: $\texttt{dist}(y) \leqslant \delta(s, y)$ and $\delta(s, y) \leqslant \delta(s, v)$

We are almost there :)

Finally, we have that $\texttt{dist}(v) \leqslant \texttt{dist}(y)$

because v is EXTRACTED next *(so has the minimum* $\texttt{dist}$*)*

Putting it all together, $\texttt{dist}(v) \leqslant \delta(s, v)$

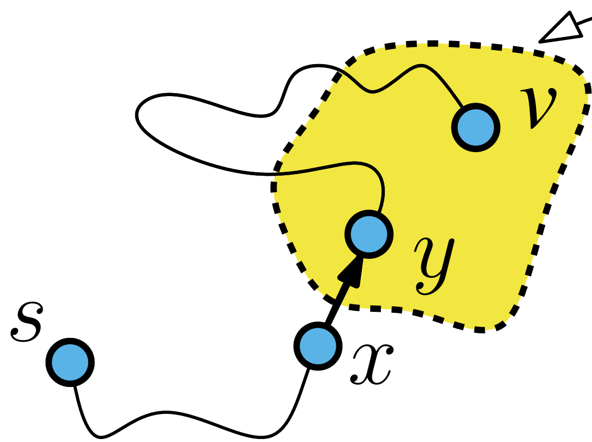
Proof of Correctness

the true distance *between s and v*

v is the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

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

Summary

we assumed that v was the first vertex to be EXTRACTED with $\texttt{dist}(v) \neq \delta(s, v)$

we proved that $\texttt{dist}(v) \leqslant \delta(s, v)$

however, we also have that, $\texttt{dist}(v) \geqslant \delta(s, v)$ *(Dijkstra only finds actual paths)*

so we have that $\texttt{dist}(v) = \delta(s, v)$ **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 INSERT, DECREASEKEY and EXTRACTMIN supported by the queue

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
  u = EXTRACTMIN()
  For every edge (u, v) ∈ E
    If dist(v) > dist(u) + weight(u, v)
      dist(v) = dist(u) + weight(u, v)
      DECREASEKEY(v, dist(v))
```

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)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V|)$ 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)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V|)$ time

(*we store* dist *in an array of length* $|V|$)

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)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V|)$ time

(*we store* 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)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V|)$ time

(*we store* `dist` *in an array of length* $|V|$)

$O(1)$ time

$O(|V| \cdot T_{\text{INSERT}})$ 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)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V| \cdot T_{\text{INSERT}})$ 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

DIJKSTRA(s)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V| \cdot T_{\text{INSERT}})$ 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
INSERT, DECREASEKEY and EXTRACTMIN
supported by the queue

DIJKSTRA(s)

For all v, set $\texttt{dist}(v) = \infty$
set $\texttt{dist}(s) = 0$
For each v, Do INSERT$(v, \texttt{dist}(v))$
While the queue is not empty
  $u = $ EXTRACTMIN$()$
  For every edge $(u, v) \in E$
    If $\texttt{dist}(v) > \texttt{dist}(u) + \texttt{weight}(u, v)$
      $\texttt{dist}(v) = \texttt{dist}(u) + \texttt{weight}(u, v)$
      DECREASEKEY$(v, \texttt{dist}(v))$

$O(|V| \cdot T_{\text{INSERT}})$ time

for the setup

$O(T_{\text{EXTRACTMIN}})$ time per iteration

relaxing an edge takes
$O(T_{\text{DECREASEKEY}})$ 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)

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V| \cdot T_{\text{INSERT}})$ time

for the setup

$O(T_{\text{EXTRACTMIN}})$ time per iteration

relaxing an edge takes
$O(T_{\text{DECREASEKEY}})$ time

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
INSERT, DECREASEKEY and EXTRACTMIN
supported by the queue

```
DIJKSTRA(s)
```

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u, v) ∈ E
      If dist(v) > dist(u) + weight(u, v)
         dist(v) = dist(u) + weight(u, v)
         DECREASEKEY(v, dist(v))
```

$O(|V| \cdot T_{\text{INSERT}})$ time

for the setup

$O(T_{\text{EXTRACTMIN}})$ time per iteration

relaxing an edge takes
$O(T_{\text{DECREASEKEY}})$ time

We do $O(|V|)$ iterations of the while loop
and relax each edge at most once...

so overall this takes:

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)
```

```
For all v, set dist(v) = ∞
set dist(s) = 0
For each v, Do INSERT(v, dist(v))
While the queue is not empty
   u = EXTRACTMIN()
   For every edge (u,v) ∈ E
      If dist(v) > dist(u) + weight(u,v)
         dist(v) = dist(u) + weight(u,v)
         DECREASEKEY(v, dist(v))
```

$O(|V| \cdot T_{\text{INSERT}})$ time
for the setup

$O(T_{\text{EXTRACTMIN}})$ time per iteration

relaxing an edge takes
$O(T_{\text{DECREASEKEY}})$ time

We do $O(|V|)$ iterations of the while loop
and relax each edge at most once...

so overall this takes:

$$O(|V| \cdot T_{\text{INSERT}} + |V| \cdot T_{\text{EXTRACTMIN}} + |E| \cdot T_{\text{DECREASEKEY}}) \text{ time}$$

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of INSERT, DECREASEKEY and EXTRACTMIN supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log n)$

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of INSERT, DECREASEKEY and EXTRACTMIN supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log n)$

What is n?

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of INSERT, DECREASEKEY and EXTRACTMIN supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log n)$

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
INSERT, DECREASEKEY and EXTRACTMIN
supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log n)$

What is n?

n denotes the number of elements in the queue
so $n \leqslant |V|$ (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 last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(\lvert V\rvert)$	$O(\lvert V\rvert)$
Sorted Linked List	$O(\lvert V\rvert)$	$O(\lvert V\rvert)$	$O(1)$
Binary Heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log \lvert V\rvert)$

What is n?

n denotes the number of elements in the queue
so $n \leqslant \lvert V\rvert$ (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 last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(\|V\|)$	$O(\|V\|)$
Sorted Linked List	$O(\|V\|)$	$O(\|V\|)$	$O(1)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\log \|V\|)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log \|V\|)$

What is n?

n denotes the number of elements in the queue
so $n \leqslant |V|$ (one element per vertex)

remember that for the Binary Heap n to find an element in $O(1)$ time
we needed each element to have an **ID** $\leqslant N \ldots$ *here* $N = |V|$

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of
INSERT, DECREASEKEY and EXTRACTMIN
supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN
Unsorted Linked List	$O(1)$	$O(\|V\|)$	$O(\|V\|)$
Sorted Linked List	$O(\|V\|)$	$O(\|V\|)$	$O(1)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\log \|V\|)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log \|V\|)$

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of INSERT, DECREASEKEY and EXTRACTMIN supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN	DIJKSTRA run time
Unsorted Linked List	$O(1)$	$O(\|V\|)$	$O(\|V\|)$	$O(\|V\|^2 + \|V\|\|E\|)$
Sorted Linked List	$O(\|V\|)$	$O(\|V\|)$	$O(1)$	$O(\|V\|^2 + \|V\|\|E\|)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log \|V\|)$	$O(\|E\| + \|V\| \log \|V\|)$

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of
INSERT,DECREASEKEY and EXTRACTMIN
supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN	DIJKSTRA run time
Unsorted Linked List	$O(1)$	$O(\|V\|)$	$O(\|V\|)$	$O(\|V\|^2 + \|V\|\|E\|)$
Sorted Linked List	$O(\|V\|)$	$O(\|V\|)$	$O(1)$	$O(\|V\|^2 + \|V\|\|E\|)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log \|V\|)$	$O(\|E\| + \|V\| \log \|V\|)$

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

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of INSERT, DECREASEKEY and EXTRACTMIN supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN	DIJKSTRA run time
Unsorted Linked List	$O(1)$	$O(\|V\|)$	$O(\|V\|)$	$O(\|V\|^2 + \|V\|\|E\|)$
Sorted Linked List	$O(\|V\|)$	$O(\|V\|)$	$O(1)$	$O(\|V\|^2 + \|V\|\|E\|)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log \|V\|)$	$O(\|E\| + \|V\| \log \|V\|)$

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

all of these solutions use $O(|V| + |E|)$ space

Summary

The time complexity of Dijkstra's algorithm depends on the time complexities of
INSERT, DECREASEKEY and EXTRACTMIN
supported by the queue

Recall from last lecture, the complexities of some priority queues:

	INSERT	DECREASEKEY	EXTRACTMIN	DIJKSTRA run time
Unsorted Linked List	$O(1)$	$O(\|V\|)$	$O(\|V\|)$	$O(\|V\|^2 + \|V\|\|E\|)$
Sorted Linked List	$O(\|V\|)$	$O(\|V\|)$	$O(1)$	$O(\|V\|^2 + \|V\|\|E\|)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\log \|V\|)$	$O(\|E\| + \|V\| \log \|V\|)$

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

all of these solutions use $O(|V| + |E|)$ space

Dijkstra's algorithm solves the single source shortest path algorithm
on weighted, directed graphs. . .
with **non-negative** edge weights