CSCI 136:
Data Structures
and
Advanced Programming

Lecture 33

Priority Queues / Dijkstra's Algorithm

Instructor: Dan Barowy

Williams

Topics

Student Course Surveys

Priority Queues

Dijkstra's algorithm

Your to-dos

1. **Review readings** from *Bailey*.
2. **Study** for the final exam.
   a. Pro tip: **review quizzes**.
   b. **Do problems** in study guide/practice exam.
   c. **Don't stress out!** Just be methodical and do your best.
3. **Work on resubmissions** you plan to submit.

Announcements

1. **No lab** this week.
2. **No colloquium** this week.
3. Instead: **end of year ice cream social** on Friday.

Evaluation Forms

(all of these are anonymous)

We care a lot about what you say in these forms. Please take your time and write thoughtful responses.

Your feedback is very valuable to us!

Purpose of SCS Forms

"[T]he SCS provides instructors with feedback regarding their courses and teaching. The faculty legislation governing the SCS provides that SCS results are made available to the appropriate department chair, the Dean of the Faculty, and at appropriate times, to members of the Committee on Appointments and Promotions (CAP). The results are considered in matters of faculty reappointment, tenure, and promotion."

—Office of the Provost, Williams College

Purpose of "Blue Sheets"

Student comments on the blue sheets […] are solely for your benefit. They are not made available to department or program chairs, the Dean of the Faculty, or the CAP for evaluation purposes.

—Office of the Provost, Williams College

Blue sheet prompts:

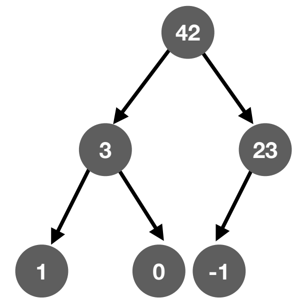
* What **course topic** did you **enjoy the most**?

* What **course topic** did you **least enjoy**? Do you think that it was valuable to learn anyway?

* Are there **other aspects** of the course that you **liked** or **disliked**? (E.g., *office hours*, *TAs*, *assignments*, *course structure*, *meeting times*, etc.)
Feel free to suggest an alternative approach.

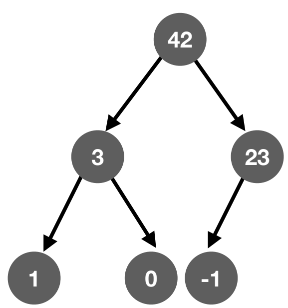
* Did you **look forward to coming to class**?

(Binary) max heap



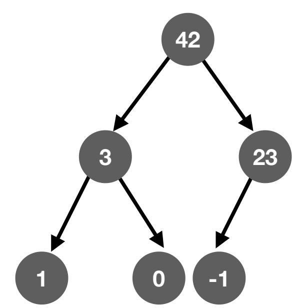
Max heap property: for any given node **n**, if **p** is a parent node of **n**, then the **key** of **p** is ≥ the **key** of **n**.

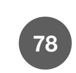
Insertion



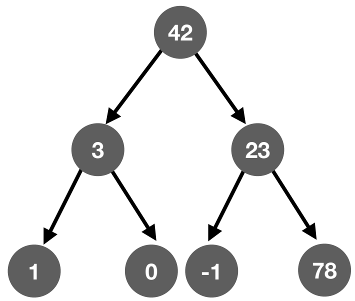
A **binary heap** is usually implemented as an **always-complete binary tree**.

Insertion



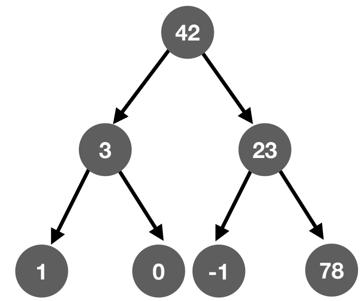
Suppose we want to insert a new node, 

Insertion



First, **insert** the new node at the first available position in the tree that **maintains completeness**.

Insertion

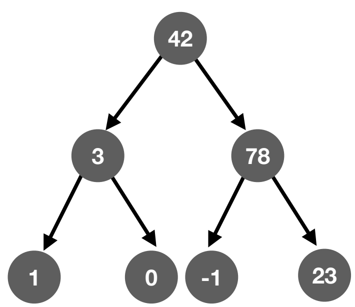


23 ≥ 78 **?**

No.

Next, **compare** the new node with its parent.

Insertion

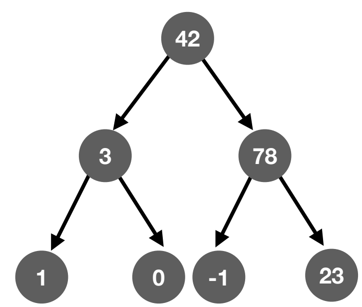


23 ≥ 78 **?**

No.

If the **max heap property** is violated, **swap**.

Insertion

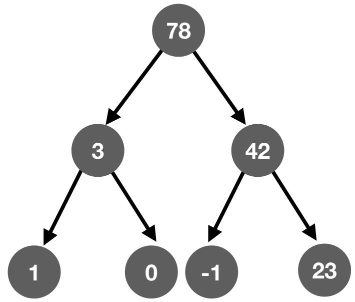


42 ≥ 78 **?**

No.

Continue swapping the new node with parents until the **max heap property is satisfied**.

Insertion

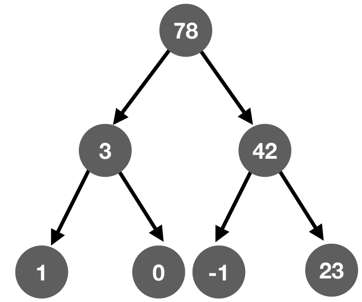


42 ≥ 78 ?

No.

Continue swapping the new node with parents until the **max heap property is satisfied** (parent ≥ node or no parents remain).

Insertion

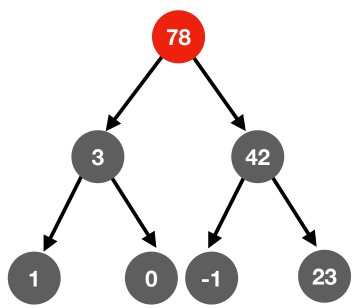


42 ≥ 78 ?

No.

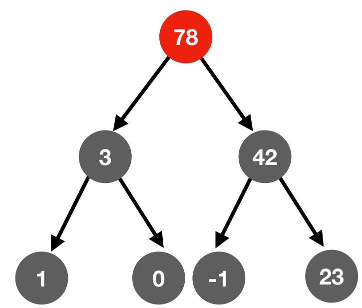
The **swapping procedure** performed on **insert** is often referred to as **heap-up** or **percolate-up**.

Find-max



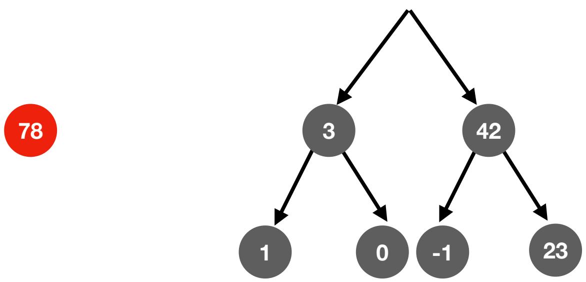
To find the **maximum element** in a max heap, simply **return** the **root**.

Extract



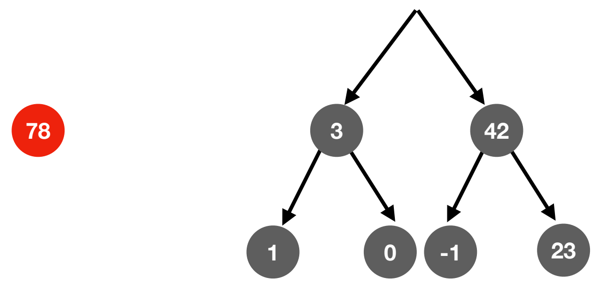
To **remove and return** the **maximum element** in a max heap, first perform **find-max**.

Extract



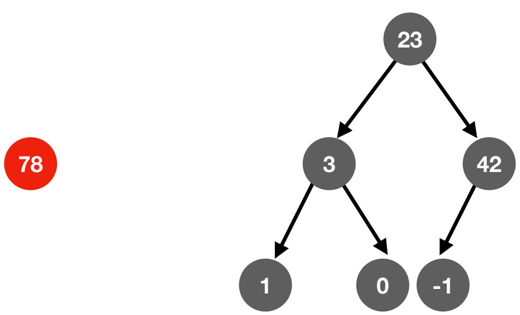
Temporarily store the max element.

Extract



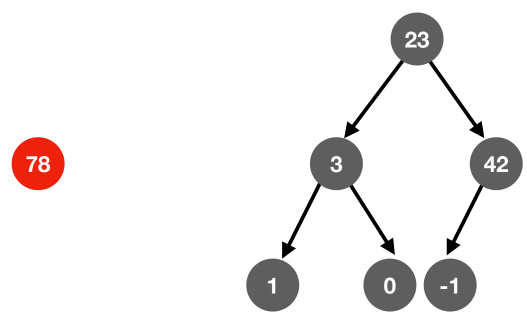
Replace the **root** with the **last element** in the complete tree.

Extract



Replace the **root** with the **last element** in the complete tree.

Extract

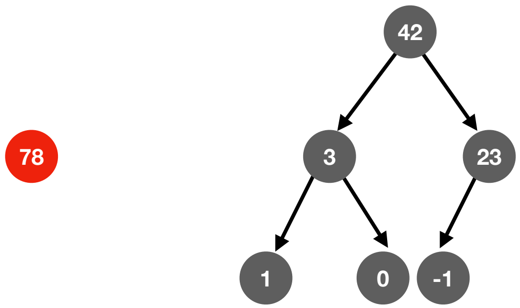


$23 \geq 42$?

No.

Compare the root with its children. **Swap** the **root** with **the largest element**.

Extract

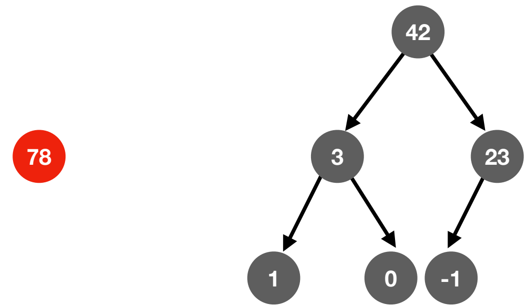


23 ≥ 42 ?

No.

Compare the root with its children. **Swap** the **root** with **the largest element**.

Extract

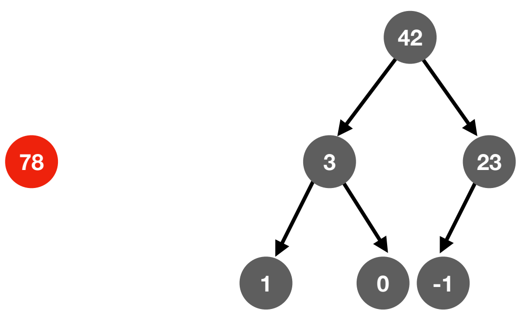


23 ≥ -1 ?

Yes.

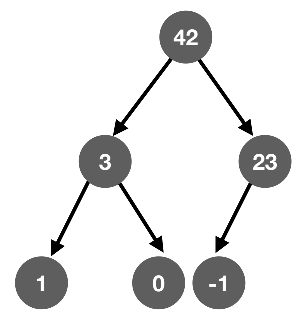
Continue swapping until the **max heap property is satisfied** (parent ≥ node or no parents remain).

Extract



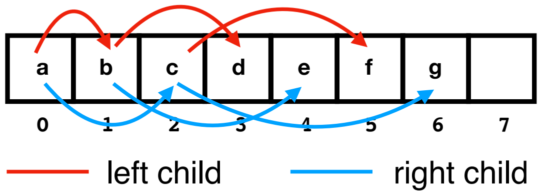
Return the saved maximum element.

Extract



The **swapping procedure** performed on **extract** is often referred to as **heap-down** or **percolate-down**.

Implementation



A binary heap is often implemented using an implicit binary tree data structure. In other words, heap nodes are actually stored in an array or vector.

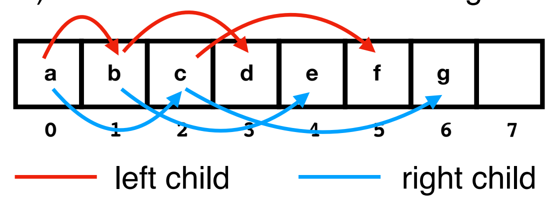
$$\texttt{leftChild(i)} = 2 \times i + 1$$
$$\texttt{rightChild(i)} = 2 \times i + 2$$
$$\texttt{parent(i)} = \left\lfloor (i - 1) / 2) \right\rfloor$$

Max heap in action

Build a max heap from the following elements:

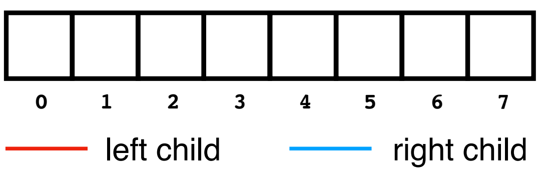
56 5 57 0 -7 99

But store the elements in an array (i.e., an implicit binary tree). Process nodes from left to right.



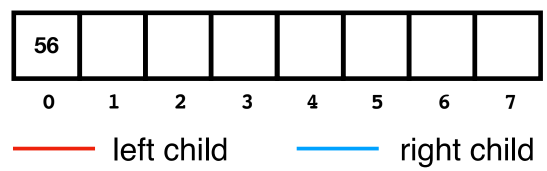
$$\texttt{leftChild(i)} = 2 \times i + 1$$
$$\texttt{rightChild(i)} = 2 \times i + 2$$
$$\texttt{parent(i)} = \left\lfloor (i - 1) / 2) \right\rfloor$$

Max heap in action



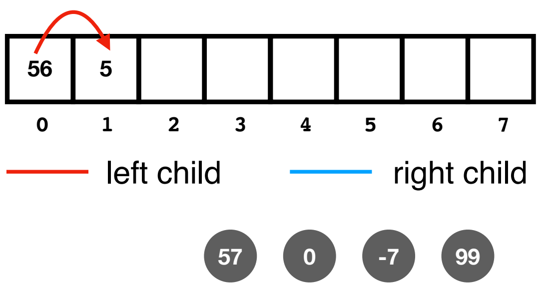
56 5 57 0 -7 99

Max heap in action

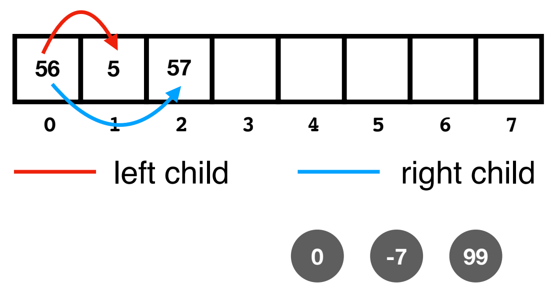


5 57 0 -7 99

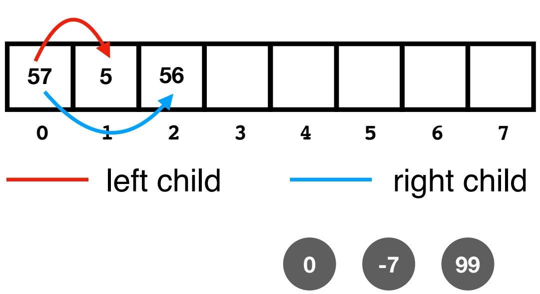
Max heap in action



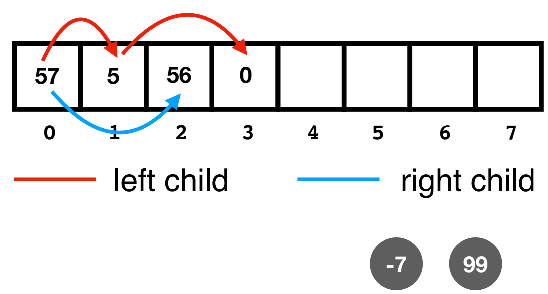
Max heap in action



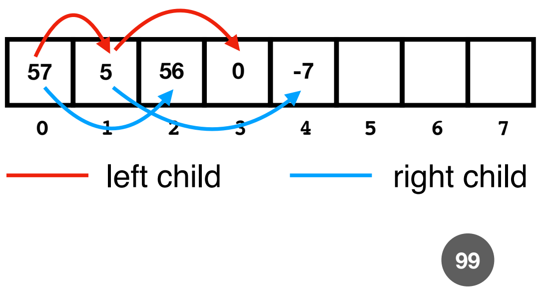
Max heap in action



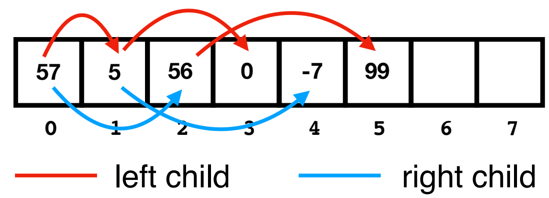
Max heap in action



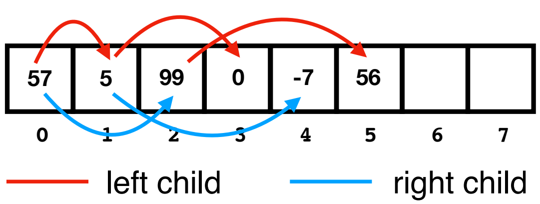
Max heap in action



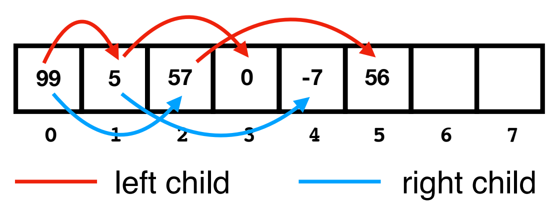
Max heap in action



Max heap in action

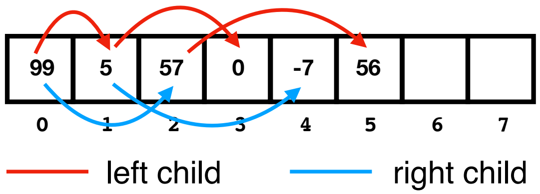


Max heap in action



Done!

Max heap in action



Advantages:

find-max: O(1)

insert: O(log n)

extract: O(log n)

Lots of interesting variants on heaps!

Summary of running times [edit]

In the following time complexities[5] $O(f)$ is an asymptotic upper bound and $\Theta(f)$ is an asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	find-min	delete-min	insert	decrease-key	merge
Binary[5]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[5]	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$[a]	$\Theta(\log n)$	$O(\log n)$[b]
Fibonacci[5][6]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Pairing[7]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$o(\log n)$[a][c]	$\Theta(1)$
Brodal[10][d]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[12]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Strict Fibonacci[13]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap	?	$O(\log n)$[a]	$O(\log n)$[a]	$\Theta(1)$	?

a. ^ a b c d e f g h i Amortized time.

b. ^ n is the size of the larger heap.

c. ^ Lower bound of $\Omega(\log\log n)$,[8] upper bound of $O(2^{2\sqrt{\log\log n}})$.[9]

d. ^ Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with n elements can be constructed bottom-up in $O(n)$.[11]

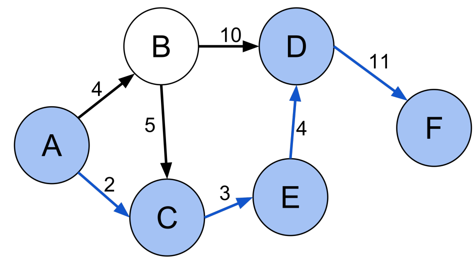
From **Wikipedia**: **priority queue** page.

Recall the example from our first class

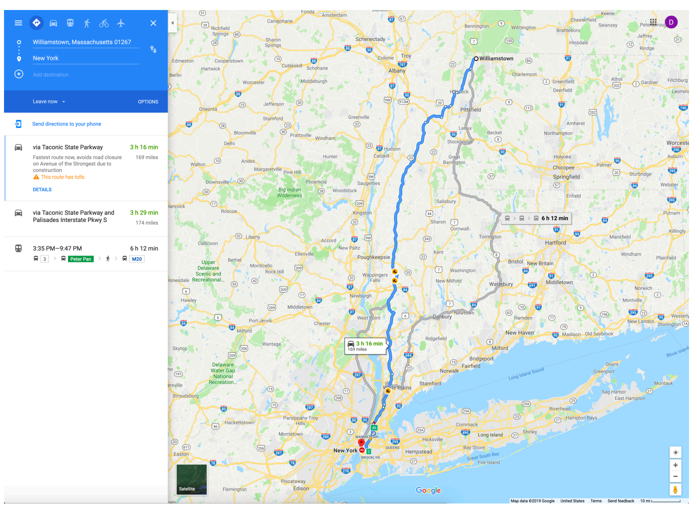
Graphs: shortest paths

Shortest path problem

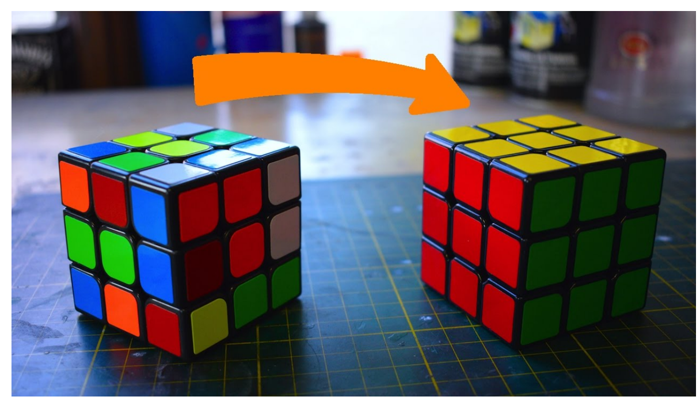
The **shortest path problem** is the problem of finding a **path between two vertices** in a graph such that **the sum** of the weights of its constituent edges **is minimized**.



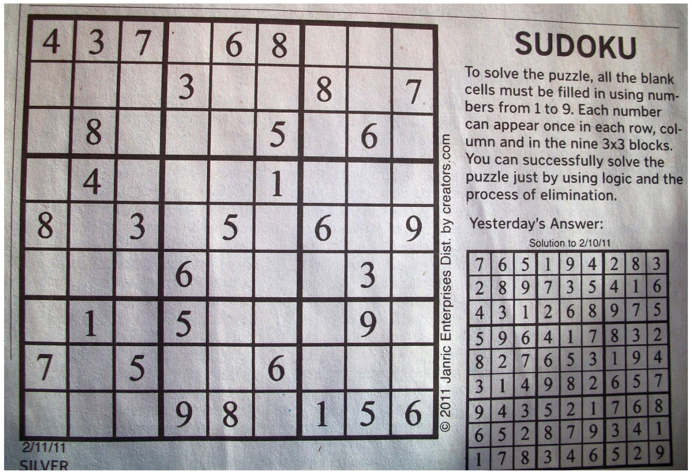
Applications



Applications

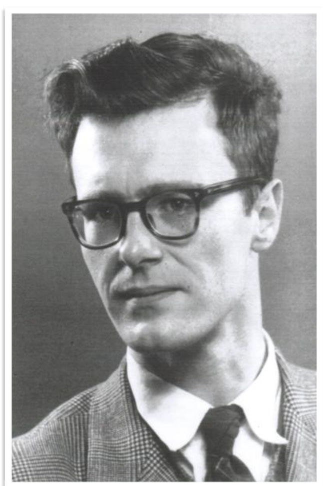


Applications



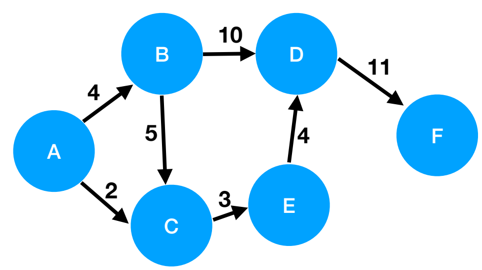
Applications

Dijkstra's algorithm



- Invented by Edsgar Dijkstra in 1959.
- The original version used a min-priority queue.
- Designed using pencil and paper; algorithm was intended to demonstrate to non-technical people how computers could be useful.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



Looking for path from A to F.

dist

A	∞
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{A, B, C, D, E, F}

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

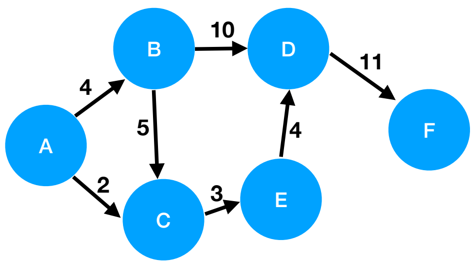
A	0
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{A, B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

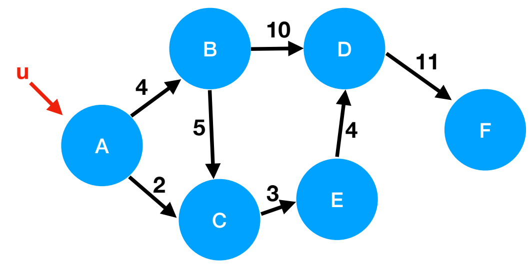
A	0
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

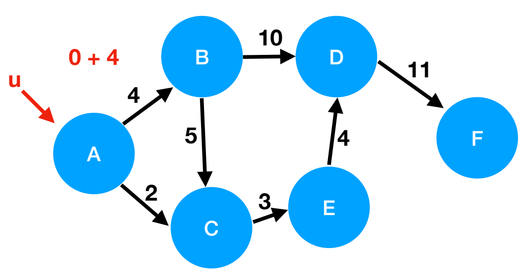
A	0
B	4
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

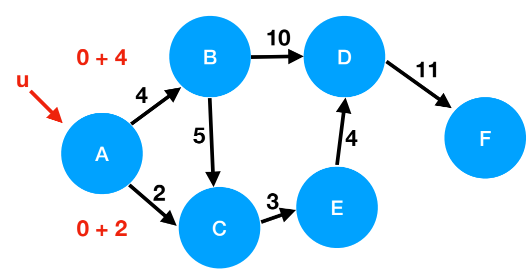
A	0
B	4
C	2
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	undef
F	undef
G	undef

Q

{B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

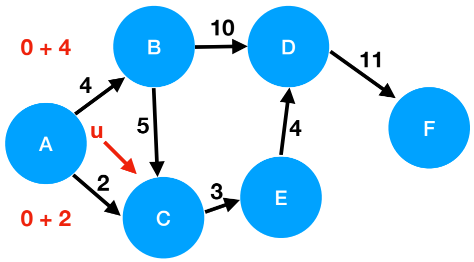
A	0
B	4
C	2
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	undef
F	undef
G	undef

Q

{B, D, E, F}



Looking for path from A to F.


```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```


dist

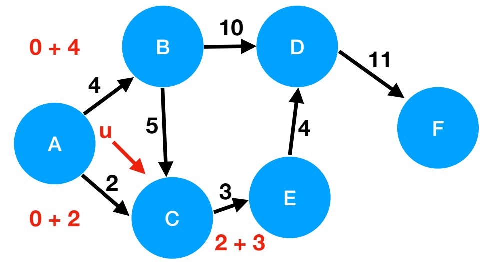
A	0
B	4
C	2
D	∞
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	C
F	undef
G	undef

Q

{B, D, E, F}



Looking for path from A to F.


```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```


dist

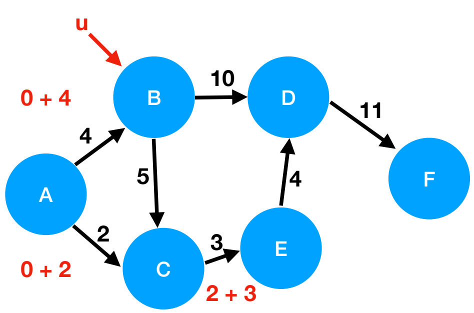
A	0
B	4
C	2
D	∞
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	C
F	undef
G	undef

Q

{D, E, F}



Looking for path from A to F.


```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```


dist

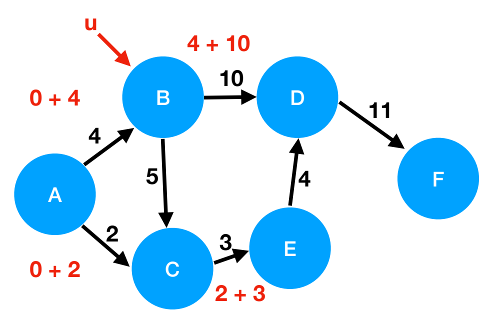
A	0
B	4
C	2
D	14
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	B
E	C
F	undef
G	undef

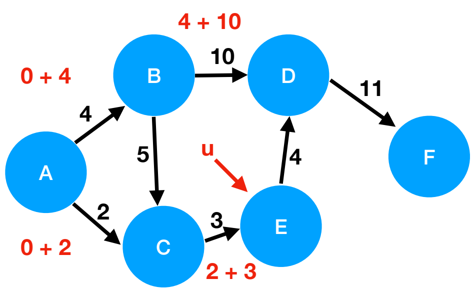
Q

{D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



dist

A	0
B	4
C	2
D	14
E	5
F	∞
G	∞

prev

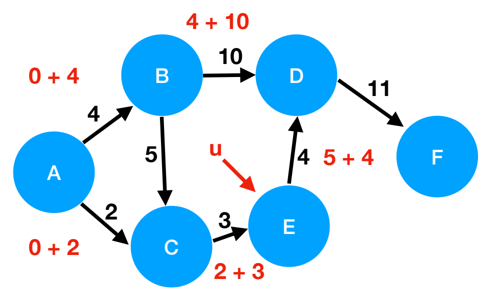
A	undef
B	A
C	A
D	B
E	C
F	undef
G	undef

Q

{D, F}

Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



dist

A	0
B	4
C	2
D	9
E	5
F	∞
G	∞

prev

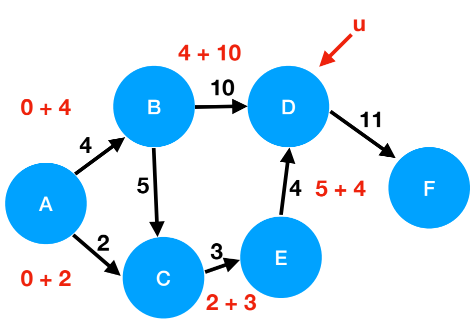
A	undef
B	A
C	A
D	E
E	C
F	undef
G	undef

Q

{D, F}

Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



dist

A	0
B	4
C	2
D	9
E	5
F	∞
G	∞

prev

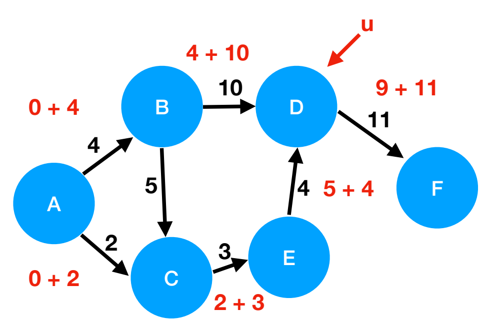
A	undef
B	A
C	A
D	E
E	C
F	undef
G	undef

Q

{F}

Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef

Q

{F}

Looking for path from A to F.

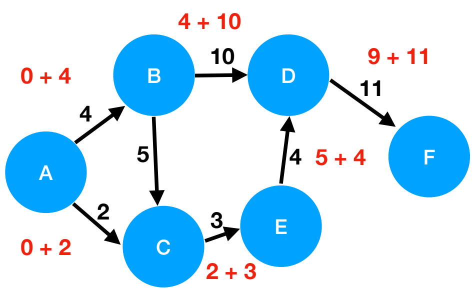
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef



Q

{}

Done!

Looking for path from A to F.

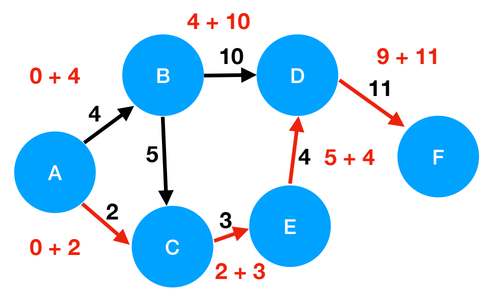
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef



Q

{}

Done!

Read prev backward from F and reverse.

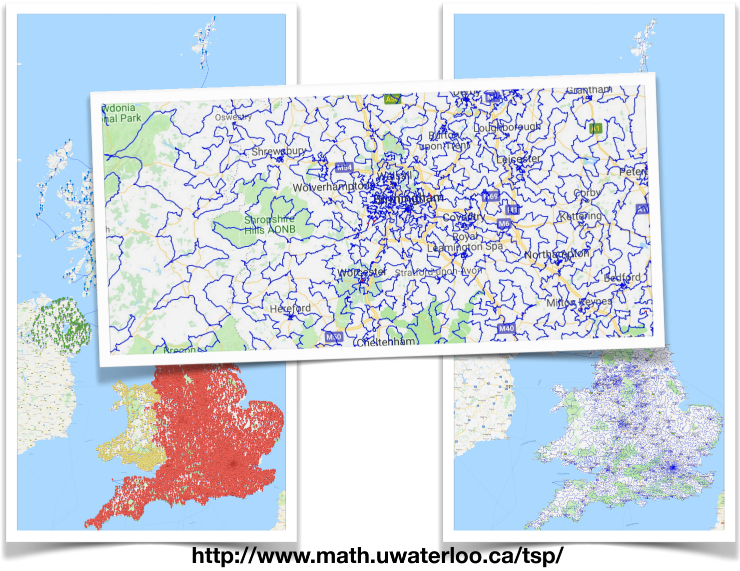
Graphs: traveling salesperson

Applications

Delivery routes.

Applications

Optimal 49,687-stop pub crawl



http://www.math.uwaterloo.ca/tsp/

Recap & Next Class

Today:

Priority queues

Heaps

Next class:

Dijkstra’s algorithm