Graphs
Describe a platypus!

You can use a drawing, or text, or code, or...
Welcome to DrRacket, version 6.3 [3m].
Language: racket; memory limit: 128 MB.

> perry ; an undefined variable

Cannot reference an identifier before its definition

> 'perry ; a symbol
'perry

> '(perry 2016) ; a symbol in a list
'(perry 2016)

> (first '(perry 2016))
'perry

> (symbol? (first '(perry 2016)))
#f

> (symbol? (second '(perry 2016)))
#f

> (define perry "platypus") ; bind the value "platypus" to the name perry

> perry
"platypus"

> 'perry
'perry

> (list perry 2016)
'("platypus" 2016)

> 'perry
'(perry 2016)
>
Racket miscellanea: member

Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 128 MB.

> (range 10)
'(0 1 2 3 4 5 6 7 8 9)
> (member 10 (range 10))
#f
> (member 0 (range 10))
'(0 1 2 3 4 5 6 7 8 9)
> (member 5 (range 10))
'(5 6 7 8 9)

docs.racket-lang.org/reference/pairs.html
Racket miscellanea: append

Welcome to DrRacket, version 6.2.1 [3m].
Language: racket [custom]; memory limit: 128 MB.
> (append (range 10) (range 11 20))
'(0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19)
> (append (range 11 20) (range 10))
'(11 12 13 14 15 16 17 18 19 0 1 2 3 4 5 6 7 8 9)
>

docs.racket-lang.org/reference/pairs.html
Racket miscellanea: remove-duplicates

Welcome to DrRacket, version 6.3 [3m].
Language: racket; memory limit: 128 MB.
> (remove-duplicates '(1 1 2 3 5))
'(1 2 3 5)
>
Can you: start at point A, cross every bridge only once, and return to point A?

Leonard Euler

No!
We need a model
Graphs!
A set of nodes/vertices (places), and a set of edges (links)
Graphs represent relationships

Like what?
A node is …
An edge is…

![Diagram of a graph showing nodes A, B, C, D and edges connecting them.](image)
Graphs represent relationships

Like Facebook
A node is a Facebook user
An edge is a “friendship”
We can “traverse the edge” in both directions. The relationship is “mutual”.

Undirected graph
Undirected graph

We can “traverse the edge” in both directions.
The relationship is “mutual”.
Graphs represent relationships

Like Twitter
A node is a Twitter user
An edge is a “follow”
Directed graph

We can “traverse the edge” in one direction.
The relationship is “one way”.

← important vocabulary!
Directed graph

We can “traverse the edge” in one direction.
The relationship is “one way”.

"source"    "destination"

A → B     A → B
A → B     A → C
A → B     A → D
B → C
B → C
B → C
B → C
B → C
D → C
D → C
Graphs represent relationships

Like highways
A node is a city
An edge is a highway from one city to another
Weighted graph

Information (usually “cost”) associated with each edge
Graphs represent relationships

Like flights

A node is a city

An edge is a flight from one city to another
Directed weighted graph

Information (usually “cost”) associated with each edge
<table>
<thead>
<tr>
<th></th>
<th>Undirected</th>
<th>Directed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unweighted</strong></td>
<td><img src="image" alt="facebook" /></td>
<td><img src="image" alt="twitter" /></td>
</tr>
<tr>
<td><strong>Weighted</strong></td>
<td><img src="image" alt="highways" /></td>
<td><img src="image" alt="flights" /></td>
</tr>
<tr>
<td></td>
<td>weights = distance, time, cost</td>
<td>weights = distance, time, cost</td>
</tr>
</tbody>
</table>
D has one adjacent edge.

C is adjacent to D.

There are two paths from A to D.

C is reachable from A.

important vocabulary!
Complete graph

There is an edge between each pair of nodes.

In other words, each node is adjacent to every other node.

*To be true in a directed graph, the edges must go in both directions.*
There is a path between each pair of nodes.

In other words, each node is reachable from every other node. 

*If this is true in a directed graph, the graph is “strongly connected”.*
a **sparse** graph has few edges

a **dense** graph has many edges

important vocabulary!
Cycle

no cycles
there are a finite number of paths to a node

cycle
there may be an infinite number of paths to a node
We’ve seen graphs before!
a linked list is a graph

graph

linked list
connected, directed graph
Designing and implementing a new data structure

- **Interface**
  Describes: **what** this data structure can do

- **Implementation: encoding**
  Describes: **how** the structure is stored, using existing data structures

- **Implementation: operations**
  Defines: **how** the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.

Our Racket graphs will be connected, weighted, directed, and functional
Designing and implementing a new data structure

- **Interface**
  Describes: what this data structure can do

**Nodes**

- **Edges**
  - construct
  - set source
  - set destination
  - set weight
  - get source
  - get destination
  - get weight

**Graphs**

- construct empty graph
- get all edges
- test if empty
- add / remove edge

**Our Racket graphs will be connected, weighted, directed, and functional**
Our Racket graphs: Interface

this interface is *not* built-in to Racket

## Edges

<table>
<thead>
<tr>
<th>Constructor</th>
<th>(make-edge source destination weight)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>(src edge) (dst edge) (weight edge)</td>
</tr>
</tbody>
</table>

## Graphs

<table>
<thead>
<tr>
<th>Constructor</th>
<th>(make-empty-graph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>(empty-graph? graph) (edge-list graph)</td>
</tr>
<tr>
<td>Modifications</td>
<td>return a new graph (add-edge edge graph) (remove-edge edge graph)</td>
</tr>
</tbody>
</table>
Our Racket graphs: examples

> (make-edge 'A 'B 10) →

> (src A B) → 'A

> (dst A B) → 'B

> (weight A B) → 10

> (edge-list) → '((A B 10))
;; edge-contains-node?
;; inputs: an edge and a node
;; outputs: true if node is either the source or the destination

(define (edge-contains-node? edge node)

---

**graph operations**
- (empty-graph? graph)
- (edge-list graph)
- (add-edge edge graph)
- (remove-edge edge graph)
- (src edge)
- (dst edge)

**list operations**
- (empty? list)
- (cons elem list)
- (list elem elem ...)
- (first list)
- (rest list)
- (map function list)
- (filter function list)
- (member elem list)
- (append list list)
- (remove-duplicates list)

and more: docs.racket-lang.org/reference/pairs.html
;; edge-contains-node?
;; inputs: an edge and a node
;; outputs: true if node is either the source or the destination
(define (edge-contains-node? edge node)
  (or (equal? (src edge) node)
      (equal? (dst edge) node)))

<table>
<thead>
<tr>
<th>graph operations</th>
<th>list operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(empty-graph? graph)</td>
<td>(empty? list)</td>
</tr>
<tr>
<td>(edge-list graph)</td>
<td>(map function list)</td>
</tr>
<tr>
<td>(add-edge edge graph)</td>
<td>(filter function list)</td>
</tr>
<tr>
<td>(remove-edge edge graph)</td>
<td>(member elem list)</td>
</tr>
<tr>
<td>(src edge)</td>
<td>(first list)</td>
</tr>
<tr>
<td>(dst edge)</td>
<td>(rest list)</td>
</tr>
<tr>
<td></td>
<td>(append list list)</td>
</tr>
<tr>
<td></td>
<td>(remove-duplicates list)</td>
</tr>
</tbody>
</table>

and more: docs.racket-lang.org/reference/pairs.html
;; node-list:
;; inputs: graph
;; outputs: list of nodes in graph (no duplicates)
(define (node-list graph)

graph operations
(empty-graph? graph)
(edge-list graph)
(add-edge edge graph)
(remove-edge edge graph)
(src edge)
(dst edge)

list operations
(empty? list)
(cons elem list)
(list elem elem ...)
(first list)
(rest list)
(map function list)
(filter function list)
(member elem list)
(append list list)
(remove-duplicates list)

and more: docs.racket-lang.org/reference/pairs.html
;; node-list:
;;   inputs: graph
;;   outputs: list of nodes in graph (no duplicates)
(define (node-list graph)
  (let* ([edges (edge-list graph)]
         [srcs (map src edges)]
         [dsts (map dst edges)])
    (remove-duplicates (append srcs dsts))))

graph operations
(empty-graph? graph)
(edge-list graph)
(add-edge edge graph)
(remove-edge edge graph)
(src edge)
(dst edge)

list operations
(empty? list)
(cons elem list)
(list elem elem ...)
(first list)
(rest list)
(map function list)
(filter function list)
(member elem list)
(append list list)
(remove-duplicates list)

and more: docs.racket-lang.org/reference/pairs.html
;; neighbors
;; inputs: a node and a graph
;; outputs: list of nodes at the other end of node’s edges
(define (neighbors node graph)

<table>
<thead>
<tr>
<th>graph operations</th>
<th>list operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(empty-graph? graph)</td>
<td>(empty? list)</td>
</tr>
<tr>
<td>(edge-list graph)</td>
<td>(cons elem list)</td>
</tr>
<tr>
<td>(add-edge edge graph)</td>
<td>(list elem elem ...)</td>
</tr>
<tr>
<td>(remove-edge edge graph)</td>
<td>(first list)</td>
</tr>
<tr>
<td>(src edge)</td>
<td>(rest list)</td>
</tr>
<tr>
<td>(dst edge)</td>
<td>(map function list)</td>
</tr>
<tr>
<td></td>
<td>(filter function list)</td>
</tr>
<tr>
<td></td>
<td>(member elem list)</td>
</tr>
<tr>
<td></td>
<td>(append list list)</td>
</tr>
<tr>
<td></td>
<td>(remove-duplicates list)</td>
</tr>
<tr>
<td></td>
<td>and more: docs.racket-lang.org/reference/pairs.html</td>
</tr>
</tbody>
</table>
;; neighbors
;; inputs: a node and a graph
;; outputs: list of nodes at the other end of node’s edges
(define (neighbors node graph)
  (let* (  
    ; get all the edges in the graph
    [allEdges (edge-list graph)]

    ; find the edges with node as the source
    [adjacent? (lambda (edge) (equal? (src edge) node))]
    [nodeEdges (filter adjacent? allEdges)]

    (map dst nodeEdges))
)

<table>
<thead>
<tr>
<th>graph operations</th>
<th>list operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(empty-graph? graph)</td>
<td>(empty? list)</td>
</tr>
<tr>
<td>(edge-list graph)</td>
<td>(map function list)</td>
</tr>
<tr>
<td>(add-edge edge graph)</td>
<td>(cons elem list)</td>
</tr>
<tr>
<td>(remove-edge edge graph)</td>
<td>(filter function list)</td>
</tr>
<tr>
<td>(src edge)</td>
<td>(list elem elem ...)</td>
</tr>
<tr>
<td>(dst edge)</td>
<td>(member elem list)</td>
</tr>
</tbody>
</table>

and more: docs.racket-lang.org/reference/pairs.html
;; contains-edge:
;;   inputs: an edge and a graph
;;   outputs: true if the graph contains the edge
(define (contains-edge edge graph)
  (list? (member edge (edge-list graph)))))

;; contains-node:
;;   inputs: a node and a graph
;;   outputs: true if the node is in the graph
(define (contains-node node graph)
  (list? (member node (node-list graph))))

graph operations
(empty-graph? graph)
(edge-list graph)
(add-edge edge graph)
(remove-edge edge graph)
(src edge)
(dst edge)

list operations
(empty? list)
(cons elem list)
(list elem elem ...)
(first list)
(rest list)

and more: docs.racket-lang.org/reference/pairs.html