Not all data can be represented as simple linked lists. Not all problems can be solved by just heading directly for the goal; sometimes you have to try one path, and if that doesn’t work, you try another one.

Let’s talk about graph structures.

Here are some of the courses you may encounter at this college.

- csu200: Discrete Structures
- csu211: Fundamentals of Computer Science I
- csu213: Fundamentals of Computer Science II
- csu370: Object Oriented Design
- csu380: Computer Organization
- csu390: Theory of Computation
- csu660: Programming Languages
- csu665: Compilers

So, there are some questions you may be asking yourself. For example “if I want to take PL, what do I need to take beforehand? Do I need to take Computer Organization? How about Compilers?”

We can represent the information about what classes have each other as prerequisites with a data structure known as a directed graph.

![Directed graph representation of course prerequisites](attachment:graph.png)
A directed graph is made up of two things: nodes, and edges. Each node is an object in the graph, and each edge is a link going from one node to another.

Directed graphs are not just used for checking dependencies between classes. They’re used for street maps (think MapQuest), attraction charts (Sarah likes Rob and Eric, Rob likes Julie, Eric likes Sarah, Julie doesn’t like anyone), and any other situation where we want to talk about objects and the relationships between them.

So, for example, we could draw another directed graph like this to represent the attraction chart I just described.

So, these graphs are quite different from the lists and trees that we’ve seen so far.
Here is a way we can represent our class dependency graph in the computer:

;; A Node is a Symbol
(define-struct edge (orig dest))
;; An Edge is a (make-edge Node Node)
;; A Graph is a [Listof Edge]
(define class-graph
  (list (make-edge 'csu213 'csu200)
        (make-edge 'csu213 'csu211)
        (make-edge 'csu370 'csu213)
        (make-edge 'csu390 'csu213)
        (make-edge 'csu380 'csu213)
        (make-edge 'csu660 'csu370)
        (make-edge 'csu660 'csu390)
        (make-edge 'csu665 'csu380)
        (make-edge 'csu665 'csu660)))
This is a list of the edges in our graph. This is not the only way to represent graphs (and it certainly isn’t the best), but it is a convenient one for our purposes.

Now, the problem is to be able to answer questions like “Do I need to take csu211 before I take csu665?”
So, let’s see what that function might look like:

;; route-exists? : Node Node Graph → Boolean
;; Is there a way to get from orig to dest in graph?
(define (route-exists? orig dest graph)
  ...
)
What now? Examples!
(equal? (route-exists? 'csu211 'csu200 class-graph) false)
(equal? (route-exists? 'csu211 'csu211 class-graph) true)
(equal? (route-exists? 'csu213 'csu211 class-graph) true)
(equal? (route-exists? 'csu665 'csu211 class-graph) true)
(equal? (route-exists? 'csu660 'csu380 class-graph) false)

Note the corner case of csu211 with itself; this is true because this program is searching for routes through a graph. If you really wanted to use this to answer questions about prerequisites, you would probably write a separate function that called route-exists?, but also handled this case differently.

So, now what?
Well, structural recursion just isn’t going to work for this sort of problem. We can’t solve the problem by decomposing the [Listof Edge] that is our graph.

Let’s think about how examples a little more...
The ones with orig=csu211 are trivial problems: csu211 has no children, so if we’re not asking to reach csu211 itself, there can’t be a route.
The one with orig=csu213 and dest=csu211 is also trivial: csu213 has csu211 as a successor, so we can quickly discover that there is a route from csu213 directly to csu211.

As for the others... well, we *can* take a recursive approach to solving them. For any orig and dest, if we can find a route from one of the successors of orig, then we can derive a route from orig to dest.

This gives us a plan for how to approach the overall problem.
First, we’ll need a way to get the successors of a node in g.

;; successors : Node Graph → [Listof Node]
;; Produces a list of nodes that a-node is immediately linked to in g.
(define (successors a-node g)
  ...)

(This is standard structural recursion; you can fill in the blanks)

So, what’s the trivial problem in ROUTE-EXISTS?

• (route-exists? orig orig G) ⇒ true

What’s the sub-problem we can generate if the problem is not trivial?

• (route-exists? CHILD dest G) for any CHILD in successors of orig

So, our plan for non-trivial problems will be to try to find a path from one of our children. If that child fails us, then we’ll try another child.

;; route-exists? : Node Node Graph → Boolean
;; Is there a way to get from orig to dest in graph?
(define (route-exists? orig dest graph)
  (cond [(symbol=? orig dest) true]
        [else ... (successors orig graph) ...]))

Hmmm. How can we work the subproblem in there? Since we can have any number of successors, we should use a auxiliary function to do the work here.

This is standard structural recursion. Follow the Design Recipe.

;; route-exists/children? : [Listof Node] Node Graph → Boolean
;; Does any node from origins have a route to dest in graph?
(define (route-exists/children? origins dest graph)
  (cond [(empty? origins) false]...)}
(else (cond
    [(route-exists? (first origins) dest graph) true]
    [else (route-exists/children? (rest origins) dest graph)])))

And, now that we have route-exists/children?, we can finish off route-exists?

;; route-exists? : Node Node Graph → Boolean
;; Is there a way to get from orig to dest in graph?
(define (route-exists? orig dest graph)
  (cond [(symbol=? orig dest) true]
        [else (route-exists/children? (successors orig graph) dest graph)]))

(equal? (successors 'csu213 class-graph) '(csu200 csu211))
(equal? (successors 'csu665 class-graph) '(csu380 csu660))
(equal? (successors 'csu211 class-graph) '())

(equal? (route-exists? 'csu211 'csu200 class-graph) false)
(equal? (route-exists? 'csu211 'csu211 class-graph) true)
(equal? (route-exists? 'csu213 'csu211 class-graph) true)
(equal? (route-exists? 'csu665 'csu211 class-graph) true)
(equal? (route-exists? 'csu660 'csu380 class-graph) false)

Great, our program works!
Or does it? There’s more to testing than just counting examples; so far, we’ve just tested several pairs of nodes in a single graph. We should really try with other graphs as well!

Hypothetically, let’s say that the Dean of the college looked at the curriculum, and decided that it would really help students if they understood Programming Languages before they took Object Oriented Design, since that way the students would have a better idea about the underlying mechanisms that dictate how object-oriented programs behave. So, the dean decides to add an edge to our graph, from csu370 to csu660.

(define class-graph
  (list (make-edge 'csu213 'csu200)
        (make-edge 'csu213 'csu211)
        (make-edge 'csu370 'csu213)
        (make-edge 'csu390 'csu213)
        (make-edge 'csu380 'csu213)
        (make-edge 'csu660 'csu370)
        (make-edge 'csu660 'csu660)
        (make-edge 'csu665 'csu380)
        (make-edge 'csu370 'csu660) ;; THIS IS THE NEW EDGE
        (make-edge 'csu665 'csu660)))

Lets run this and see what happens.

...  
Hmm. Seems to be taking a while? Why isn’t our program working?