Lecture 22

another way to generate graphs
two functions on those graphs
tandem worklists

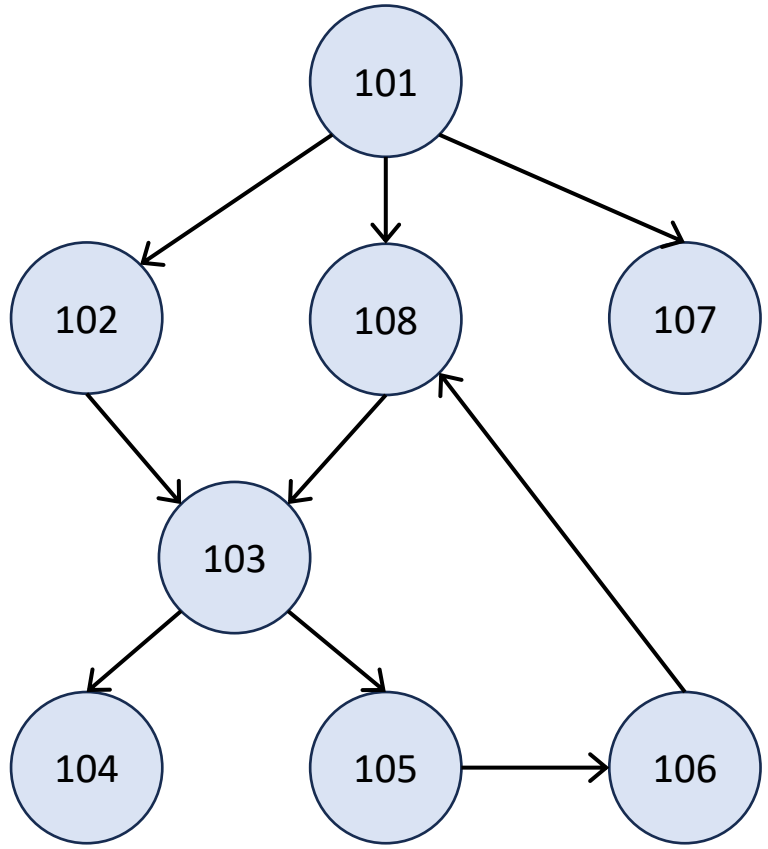
Mazes as generated graphs

```
(define M4
  (list O O O O O
        O W W W O
        O W O O O
        O W O W W
        W W O O O))
```

`(make-pos 0 0)` has no field that holds a list of subs

BUT

`(next-positions (make-pos 0 0))` generates `(list (make-pos 0 1) (make-pos 1 0))`

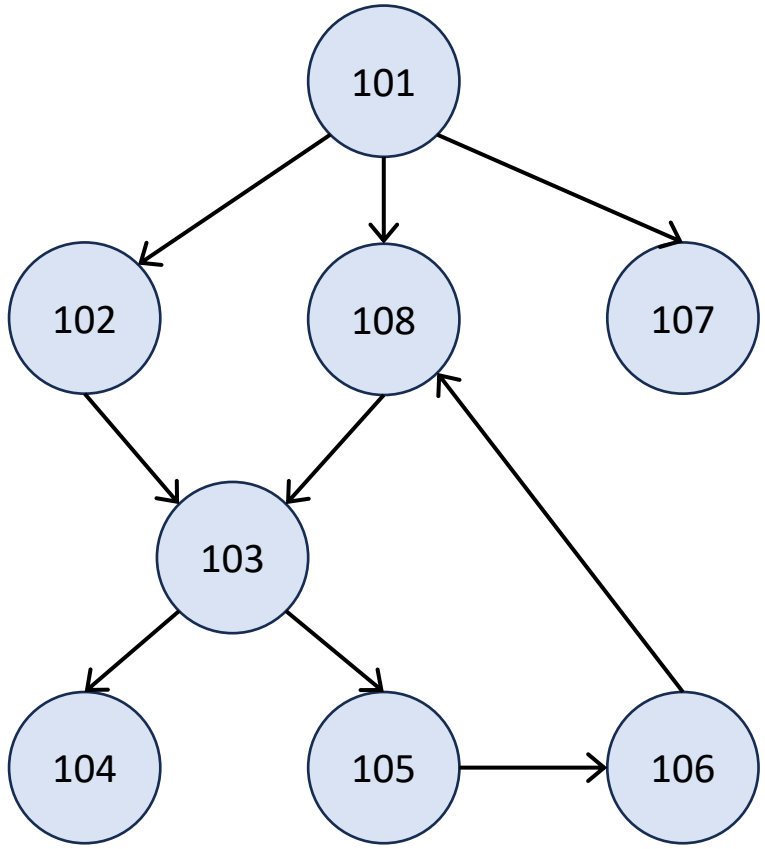

101
102
108
107
103
104
105
106

```
(@htdd Node)
(define-struct node (number nexts))
;; Node is (make-node Natural (listof Natural))
;; interp. node's number, and list of numbers of nodes that the arrows point to

(define N101 (make-node 101 (list 102 108 107)))
```

For example:

Nodes are like web pages

Node numbers are like URLs



```
(@htdd Node)
(define-struct node (number nexts))
;; Node is (make-node Natural (listof Natural))
;; interp. node's number, and list of numbers of nodes that the arrows point to

(define N101 (make-node 101 (list 102 108 107)))
```

```
(@htdd Map)
#|
 A Map is AN OPAQUE DATA STRUCTURE that represents one or more maps.
 OPAQUE means you can't look inside it.  THE ONLY THING YOU ARE  ALLOWED TO DO
 WITH A MAP IS PASS IT TO generate-node.
|#
(@htdf generate-node)
(@signature Map Natural -> Node)
;; Give map and node number (name), generate corresponding node
(define (generate-node map number)
```

treat this as a primitive

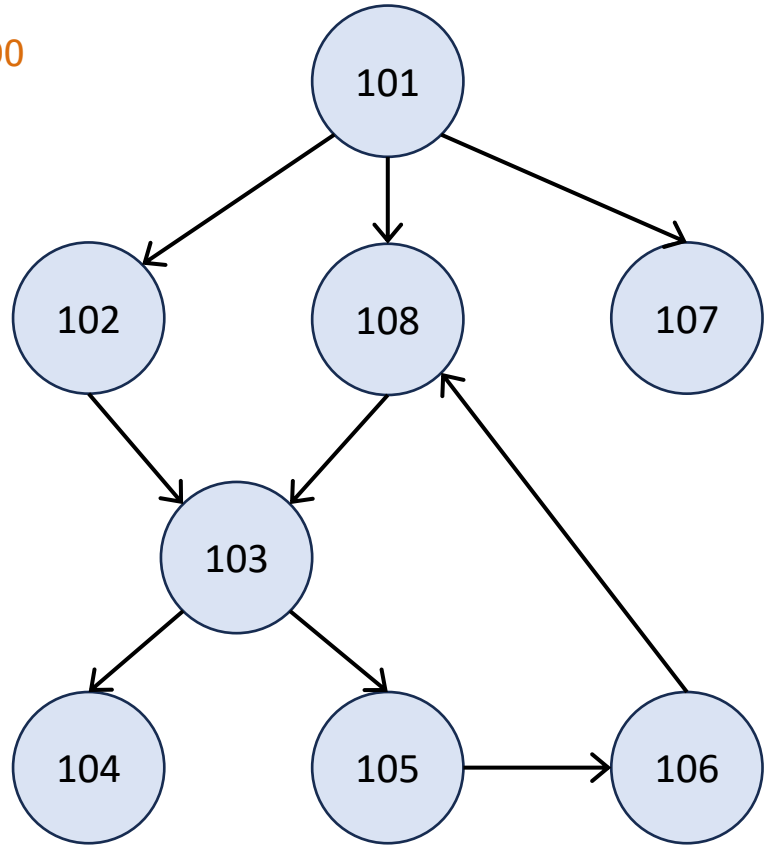
(generate-node MAP 101) generates

```
(make-node 101 (list 102 108 107)))
```

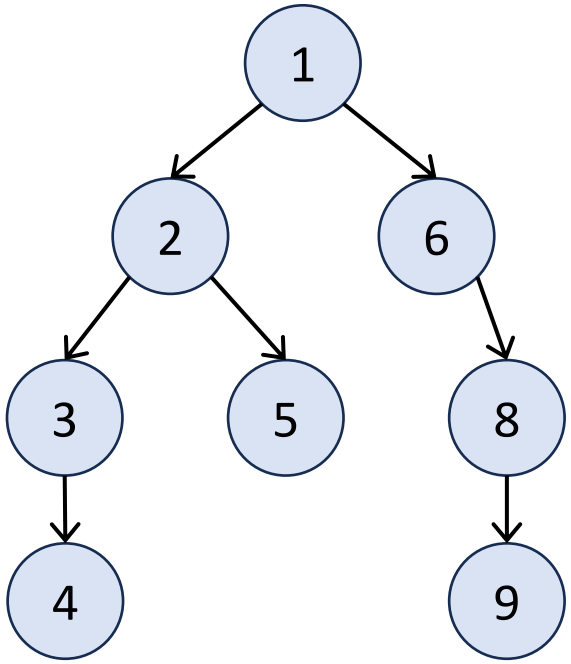
For example:

Nodes are like web pages

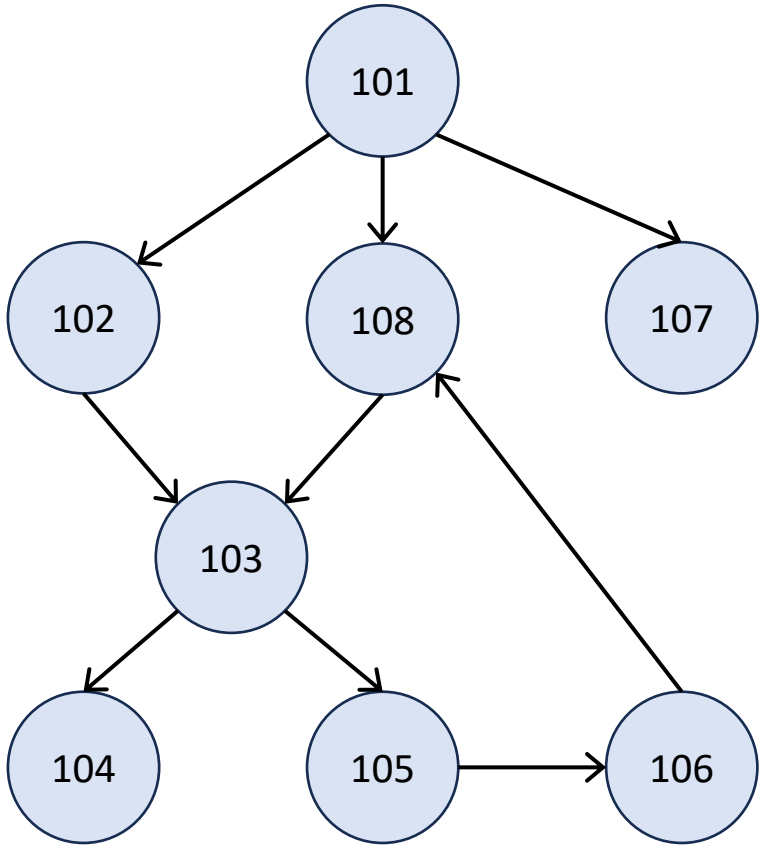
Node numbers are like URLs

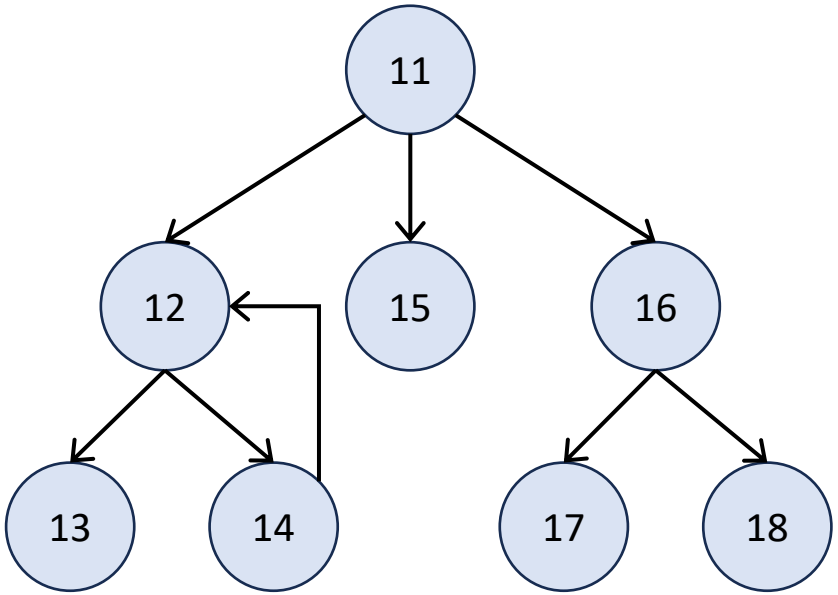


the provided MAP has 3 separate graphs in it

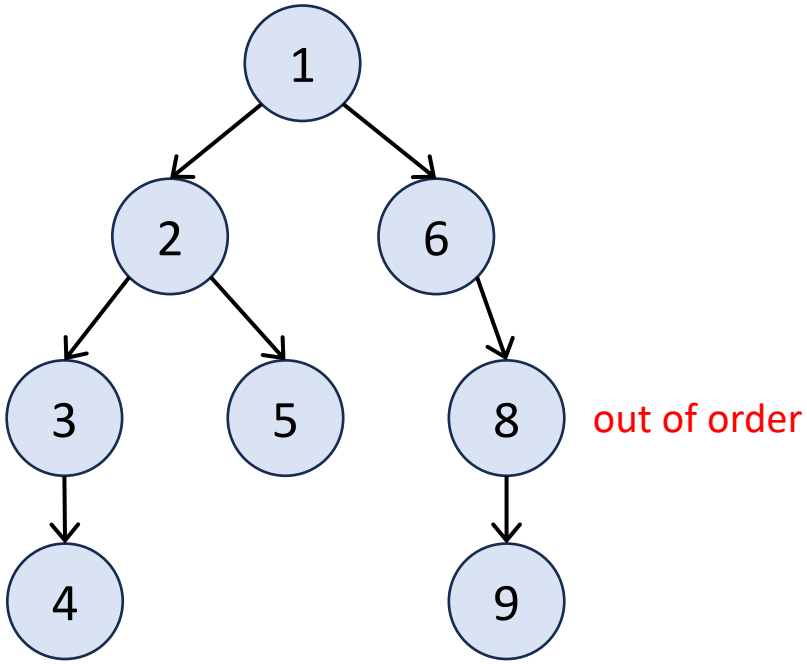


a graph *can* have cycles and joins, it doesn't have to



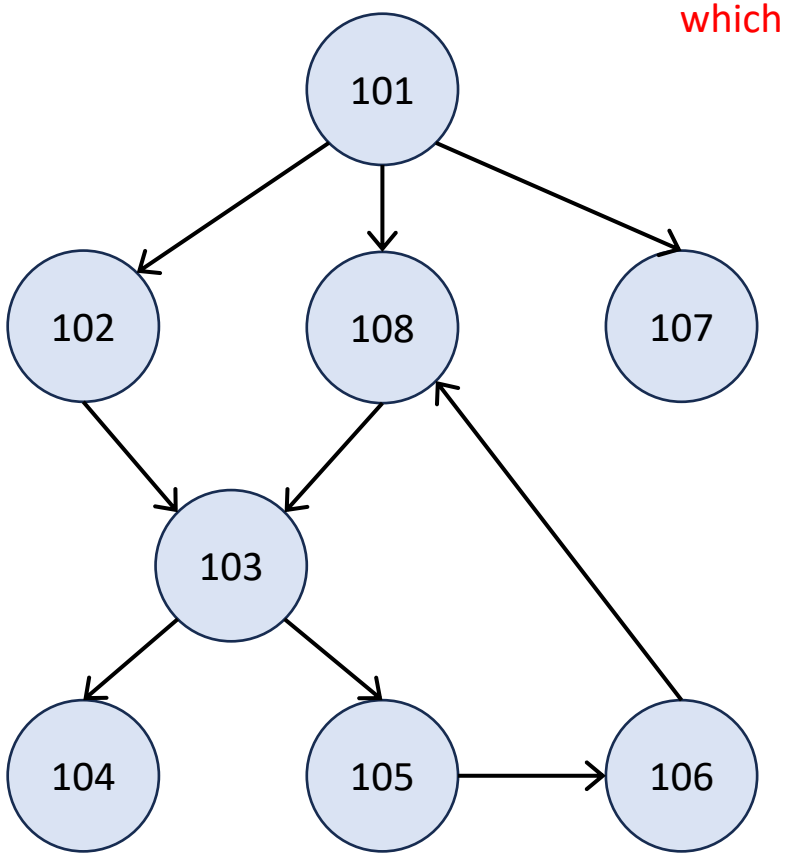


```
(@problem 1)

(@htdf first-out-of-order)
(@signature Map Natural -> Natural or false)
;; first node number in a TR traversal from num0 that is not +1 of previous node

(check-expect (first-out-of-order MAP   1) 8)
(check-expect (first-out-of-order MAP  11) false)
(check-expect (first-out-of-order MAP 101) 108)

(@template-origin genrec arb-tree accumulator)
```



```
(@problem 1)

(@htdf first-out-of-order)
(@signature Map Natural -> Natural or false)
;; first node number in a TR traversal from num0 that is not +1 of previous node

(check-expect (first-out-of-order MAP   1) 8)
(check-expect (first-out-of-order MAP  11) false)
(check-expect (first-out-of-order MAP 101) 108)

(@template-origin genrec arb-tree accumulator)
```

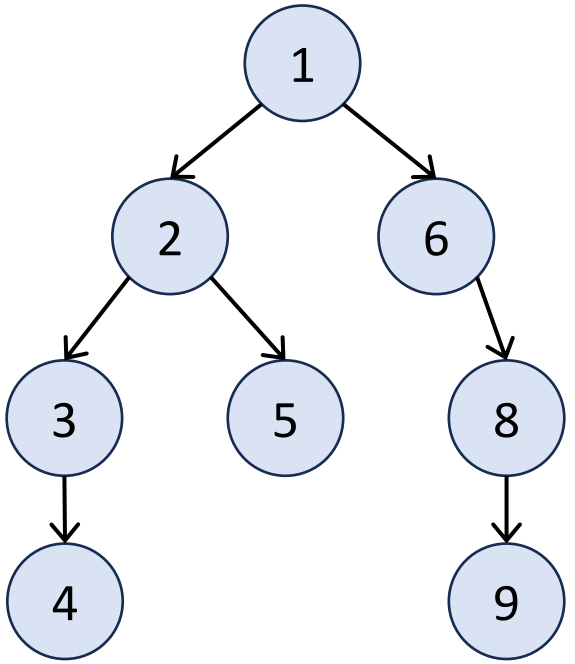
is one out of order?
which one?

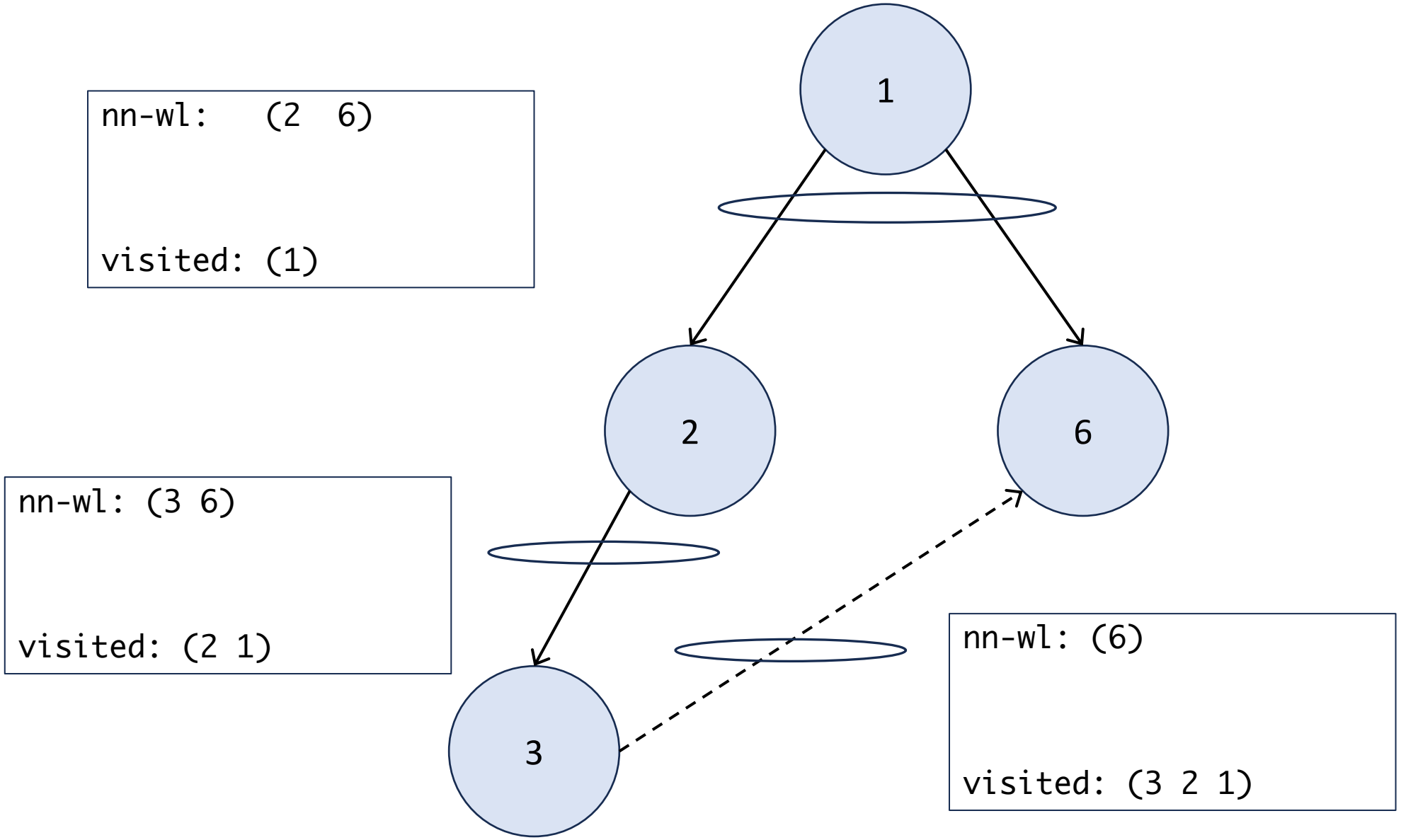


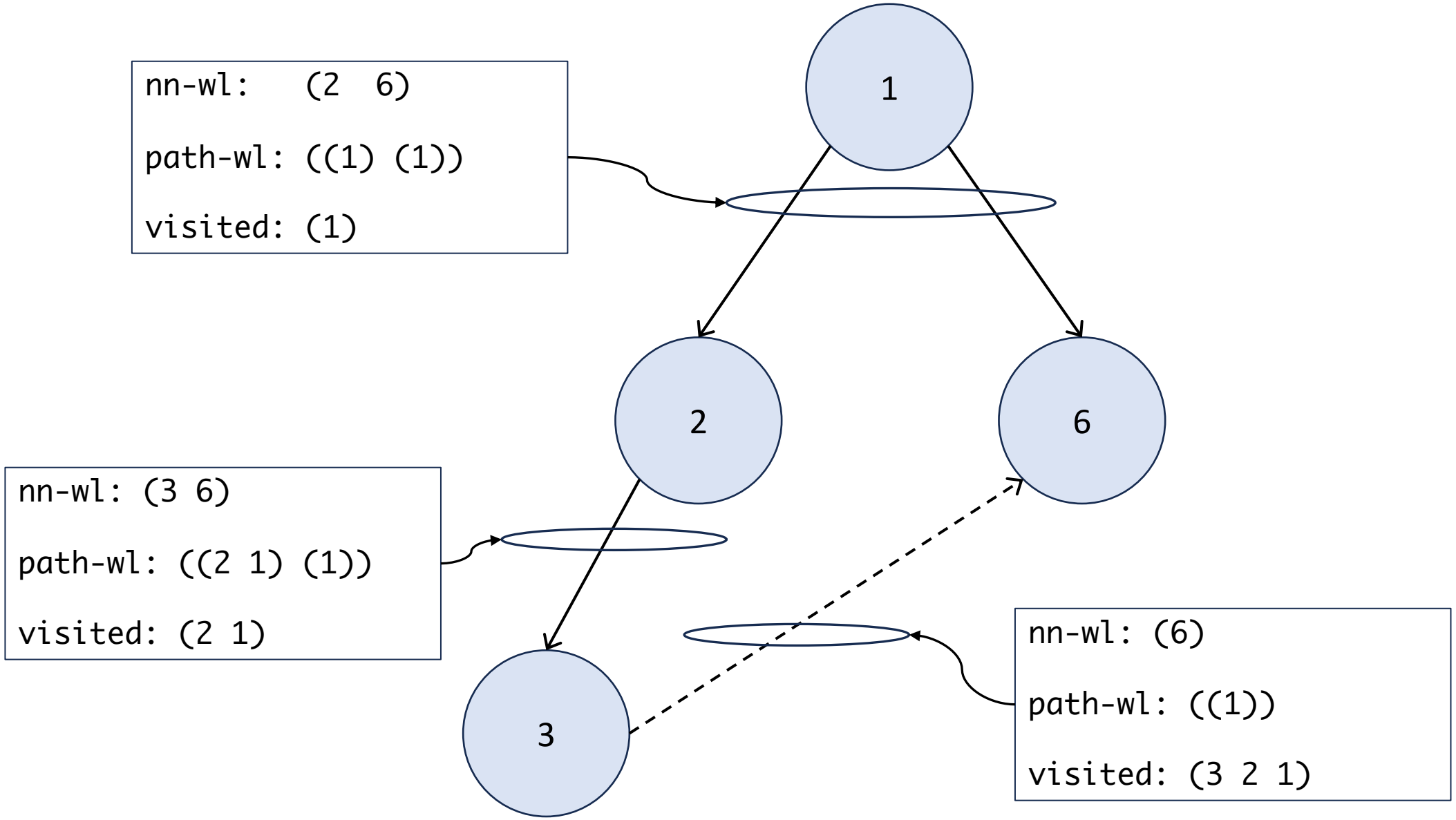
```
(define (fn-for-graph/tr map num0)
  ;; nn-wl is (listof Natural); node number worklist
  ;; fn-for-node adds the unvisited direct subs of n
  ;; fn-for-lonn takes node numbers off one at a time to call fn-for-node
  (local [(define (fn-for-node n nn-wl)
            (local [(define num (node-number n))
                    (define nexts (node-nexts n))]
              (cond [...<stop cycles>...]
                    [else
                     (fn-for-lonn (append nexts nn-wl))])))

          (define (fn-for-lonn nn-wl visited)
            (cond [(empty? nn-wl) (...)]
                  [else
                   (fn-for-node (generate-node map (first nn-wl))
                                (rest nn-wl))]))]

    (fn-for-? ...num0)))
```

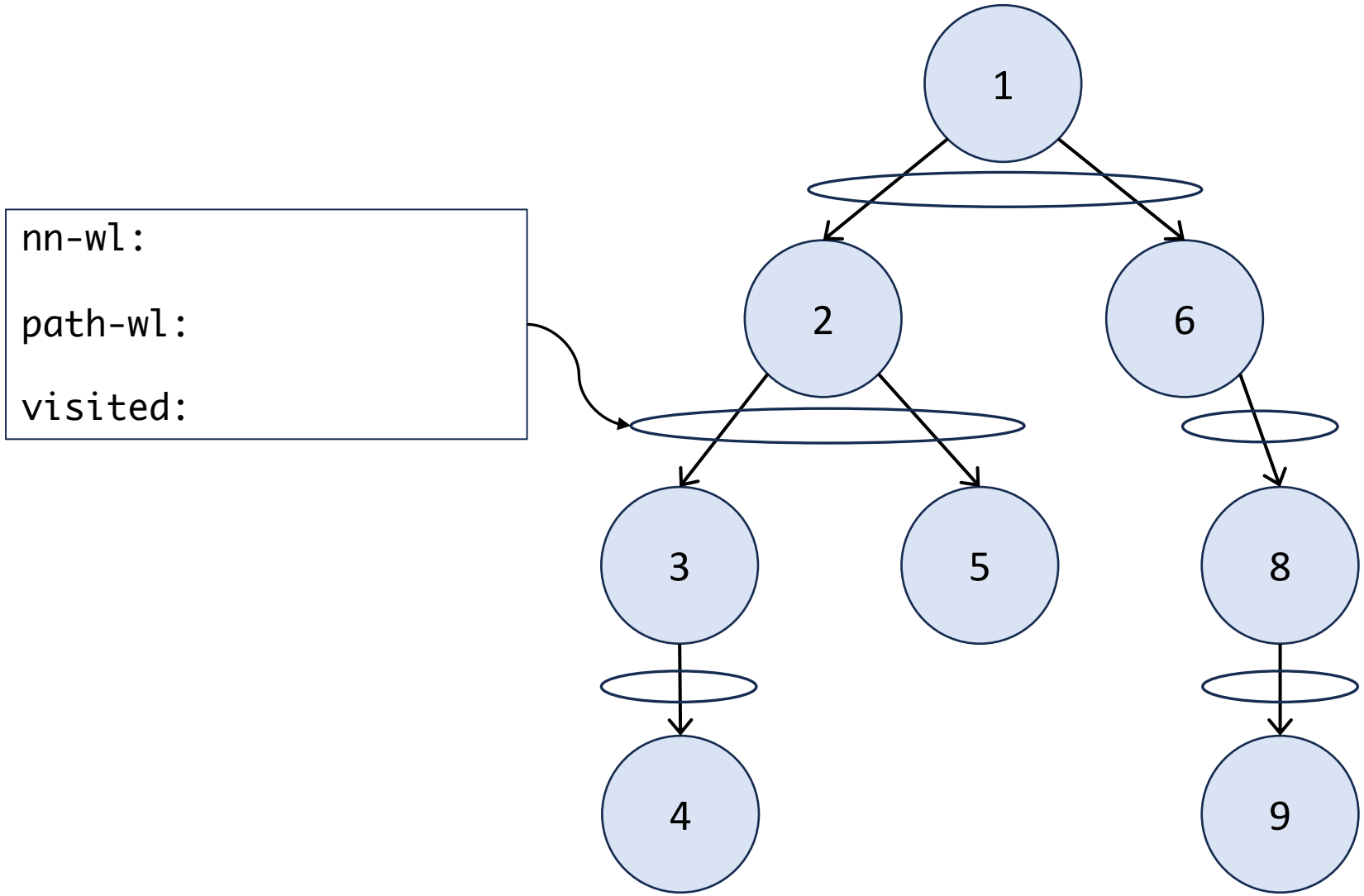
```
(define (first-out-of-order map num0)
  ;; nn-wl is (listof Natural);   worklist of node numbers
  ;; visited is (listof Natural)
  ;; Numbers of nodes already visited in the tr. (first visited) is always
  ;; the previous node's number which implies visited is never empty
  (local [(define (fn-for-node n nn-wl visited)
            (local [(define num (node-number n))
                    (define nexts (node-nexts n))
                    (define nvisited (cons num visited))]
              (cond [(member? num visited) (fn-for-lonn nn-wl visited)]
                    [(not (= num (add1 (first visited)))) num]
                    [else
                     (fn-for-lonn (append nexts nn-wl) nvisited)])))

          (define (fn-for-lonn nn-wl visited)
            (cond [(empty? nn-wl) false]
                  [else
                   (fn-for-node (generate-node map (first nn-wl))
                                (rest nn-wl)
                                visited)]))]
    ;; must start at fn-for-lonn to satisfy visited invariant
    (fn-for-lonn (node-nexts (generate-node map num0))
                 (list num0))))
```

```
(@htdf first-out-of-order-path)
(@signature Map Natural -> Natural or false)
;; in TR traversal of graph from n, produce path if first out of sequence node

(check-expect (first-out-of-order-path MAP   1) (list 1 6 8))
(check-expect (first-out-of-order-path MAP  11) false)
(check-expect (first-out-of-order-path MAP 101) (list 101 102 103 105 106 108))

(@template-origin genrec arb-tree accumulator)
```

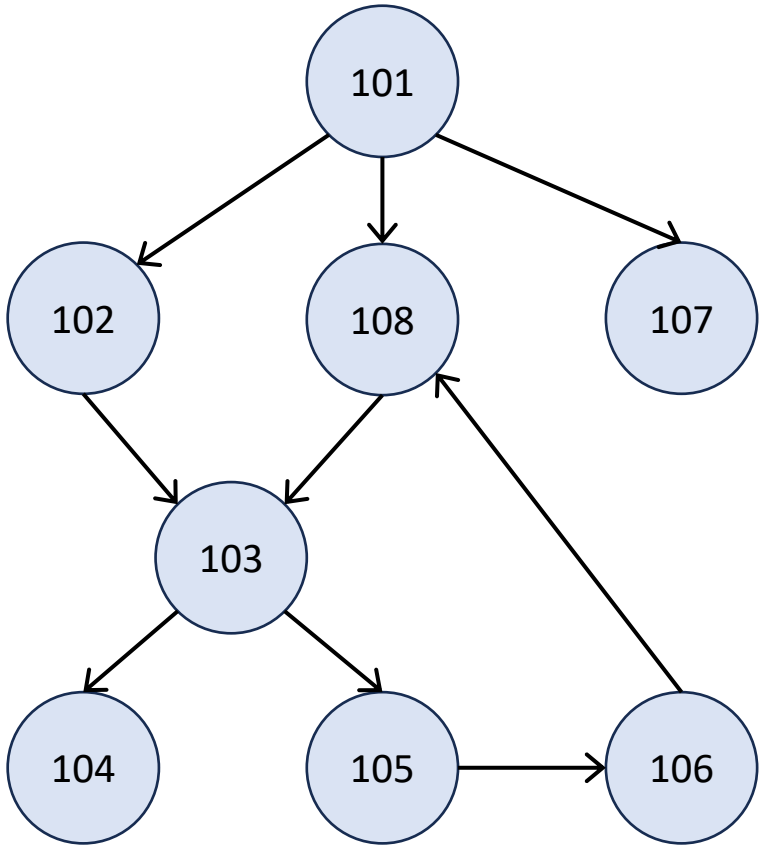



nn-wl: (2 6)
visited: (1)
1
2
6
nn-wl: (3 6)
visited: (2 1)
3
nn-wl: (6)
visited: (3 2 1)


nn-wl: (2 6)
path-wl: ((1) (1))
visited: (1)
1
2
6
nn-wl: (3 6)
path-wl: ((2 1) (1))
visited: (2 1)
3
nn-wl: (6)
path-wl: ((1))
visited: (3 2 1)


nn-wl:
path-wl:
visited:
1
2
6
3
5
8
4
9

```
(@htdf first-out-of-order-path)
(@signature Map Natural -> Natural or false)
;; in TR traversal of graph from n, produce path if first out of sequence node

(check-expect (first-out-of-order-path MAP   1) (list 1 6 8))
(check-expect (first-out-of-order-path MAP  11) false)
(check-expect (first-out-of-order-path MAP 101) (list 101 102 103 105 106 108))

(@template-origin genrec arb-tree accumulator)
```



```
(define (fn-for-graph/tr map num0)
  ;; nn-wl is (listof Natural); node number worklist
  ;; fn-for-node adds the unvisited direct subs of n
  ;; fn-for-lonn takes node numbers off one at a time to call fn-for-node
  (local [(define (fn-for-node n nn-wl)
            (local [(define num (node-number n))
                    (define nexts (node-nexts n))]
              (cond [...<stop cycles>...]
                    [else
                     (fn-for-lonn (append nexts nn-wl))])))

          (define (fn-for-lonn nn-wl visited)
            (cond [(empty? nn-wl) (...)]
                  [else
                   (fn-for-node (generate-node map (first nn-wl))
                                (rest nn-wl))]))]

    (fn-for-? ...num0)))
```

```
(define (first-out-of-order-path map num0)

  ;; nn-wl   is (listof Natural);          worklist of node numbers
  ;; path-wl is (listof (listof Natural)); tandem worklist of paths
  ;; visited is (listof Natural)
  ;; Numbers of nodes already visited in the tr. (first visited) is always
  ;; the previous node's number which implies visited is never empty

  (local [(define (fn-for-node n path nn-wl path-wl  visited)
            (local [(define num      (node-number n))
                    (define nexts    (node-nexts n))
                    (define npath    (cons num path))
                    (define nvisited (cons num visited))]
              (cond [(member? num visited) (fn-for-lonn nn-wl path-wl visited)]
                    [(not (= num (add1 (first visited)))) (reverse npath)]
                    [else
                     (fn-for-lonn (append nexts
                                          nn-wl)
                                  (append (make-list (length nexts) npath)
                                          path-wl)
                                  nvisited)])))

          (define (fn-for-lonn nn-wl path-wl visited)
            (cond [(empty? nn-wl) false]
                  [else
                   (fn-for-node (generate-node map (first nn-wl))
                                (first path-wl)
                                (rest nn-wl)
                                (rest path-wl)
                                visited)]))]

    ;; must start at fn-for-lonn to satisfy visited invariant
    (fn-for-lonn (node-nexts (generate-node map num0))
                 (make-list (length (node-nexts (generate-node map num0)))
                            (list num0))
                 (list num0))))
```