Tree - Traversals

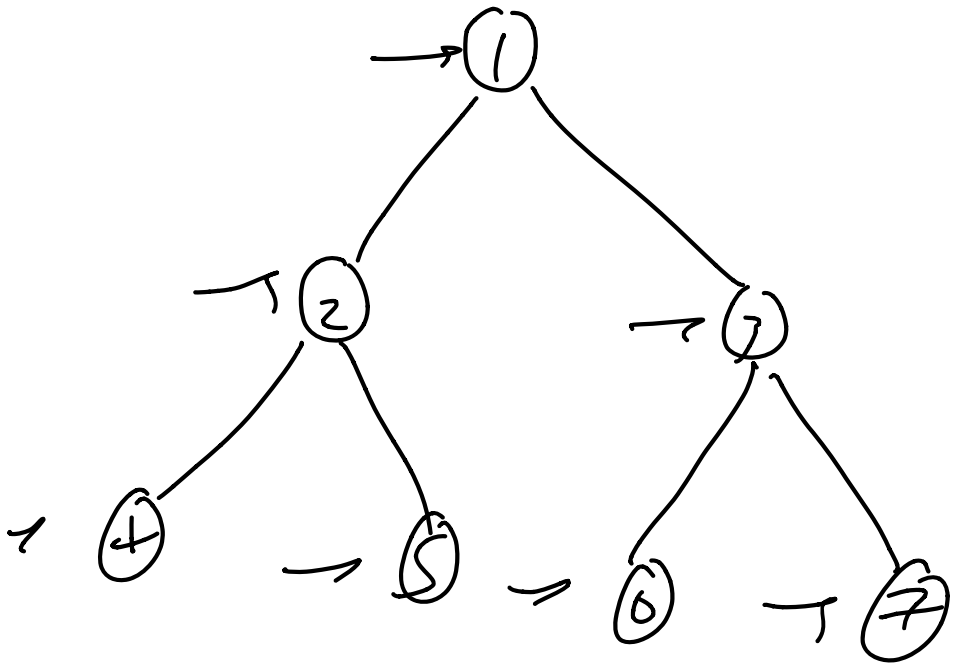
Graph Traversals (DFS).

Tree is a graph (mut

Tree is a graph with no cycle.

Binary Tree

```
(define-struct (Tree A)
  ( [value : A]
    [left-child : (U 'none (Tree A))]
    [right-child : (U 'none (Tree A))]))
```

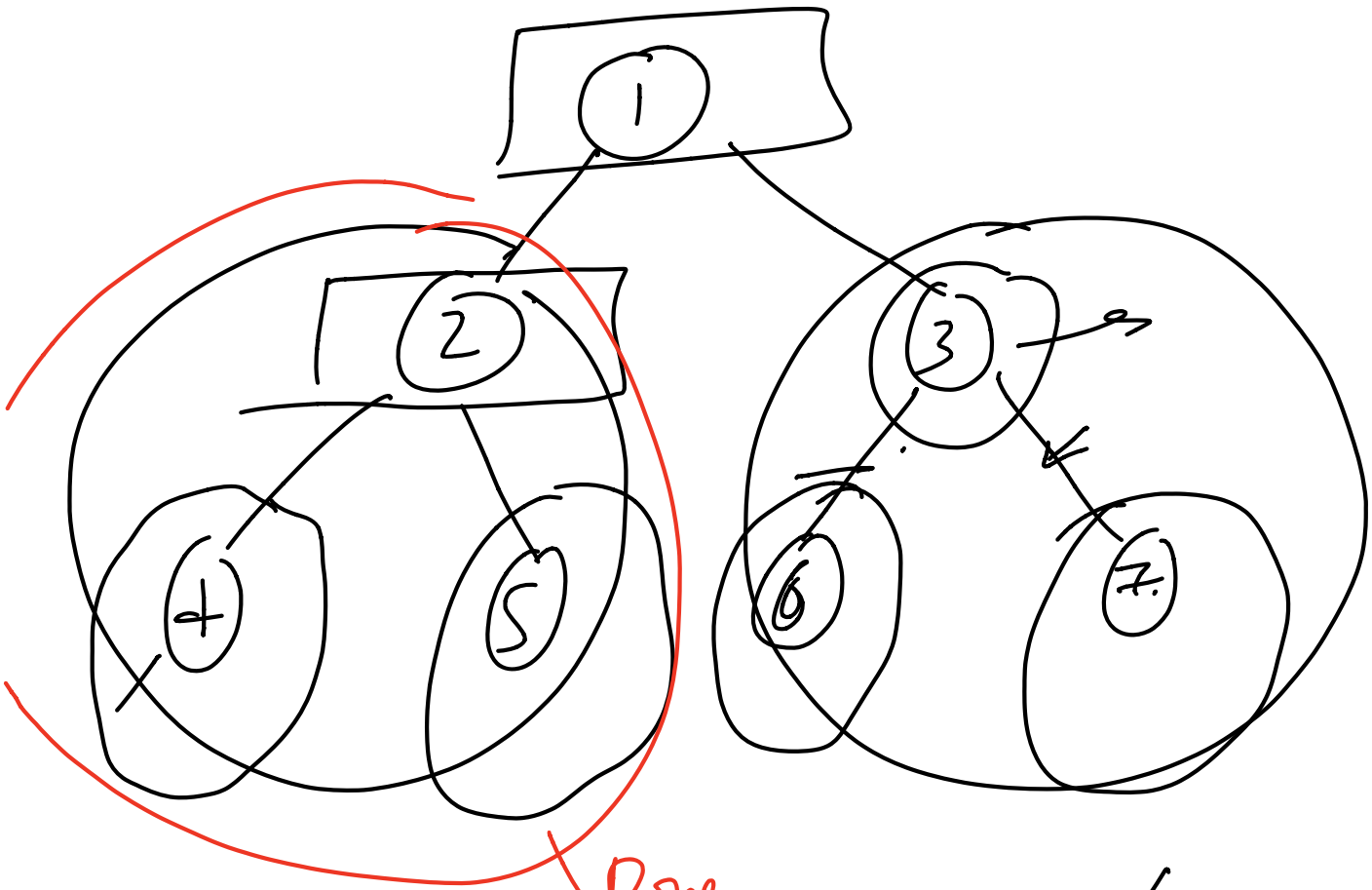


Preorder
Inorder
Postorder.

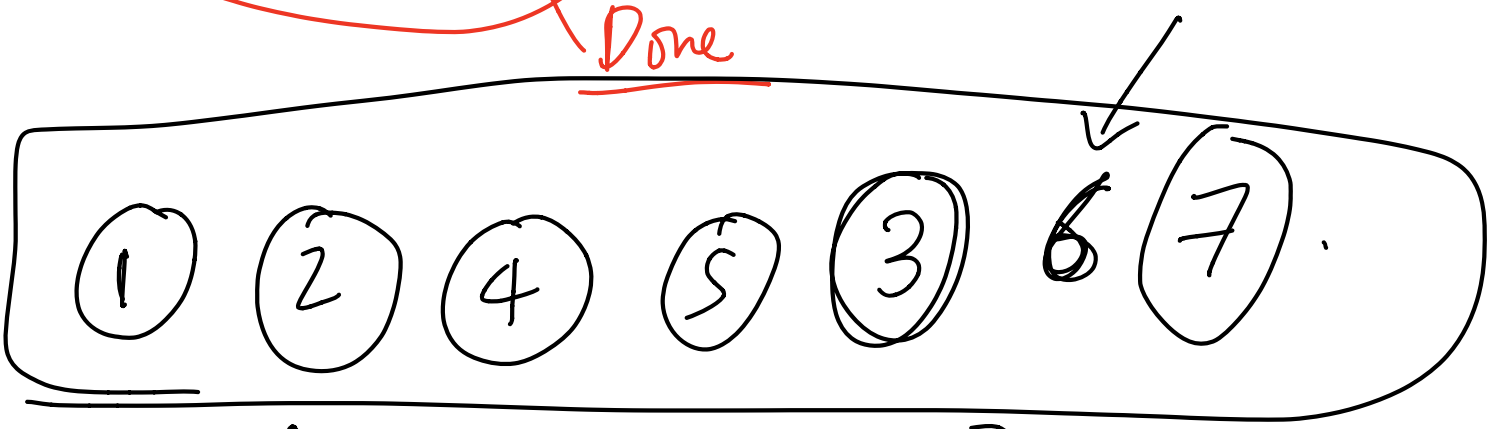
- Level order

Pre order.

1: Visit the root
Visit the left child tree (subtree)
Visit the right child tree (subtree)



Done



preorder traversal.

```
(: preorder : (All (A) (U 'none (Tree A))
                   -> (Listof A)))
(define (preorder t)
  (match
    ['none (list)]
    [(Tree v l r) (append
                   (list v) (preorder l)
                   (preorder r))]))
```

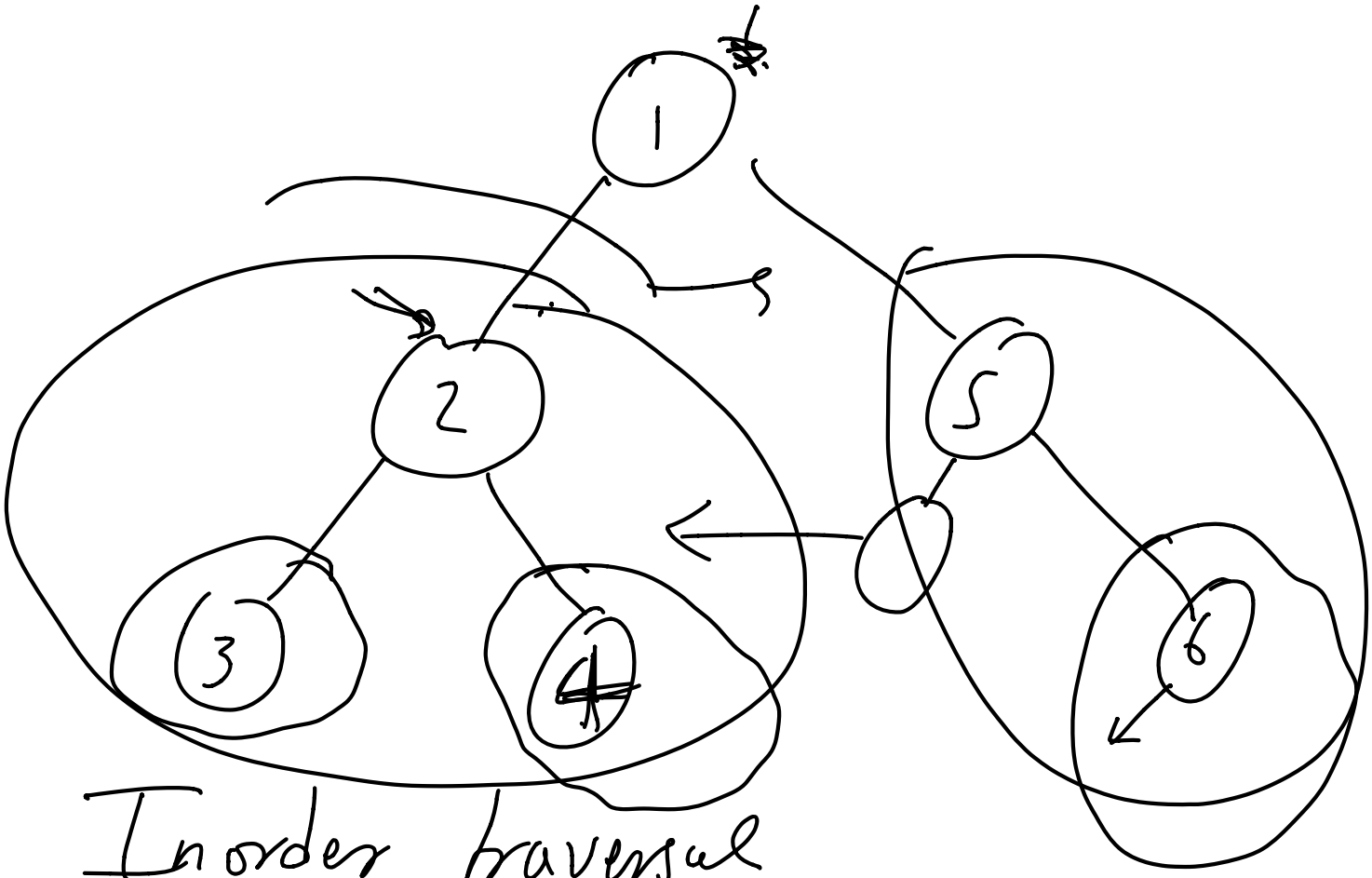
visit the left subtree

visit the root

visit the right subtree

Inorder traversal

1. visit the left subtree
2. visit the root
3. visit the right subtree



Inorder traversal

3 2 4 1 5 6

Final answer

```
(: in-order : (All (A) (U 'none (Tree A))
                -> (Listof A)))
(define (inorder t)
  (match t
    ['none (list)]
    [(Tree v l r) (append
                   (inorder l) (list v)
                   (inorder r))]))
```

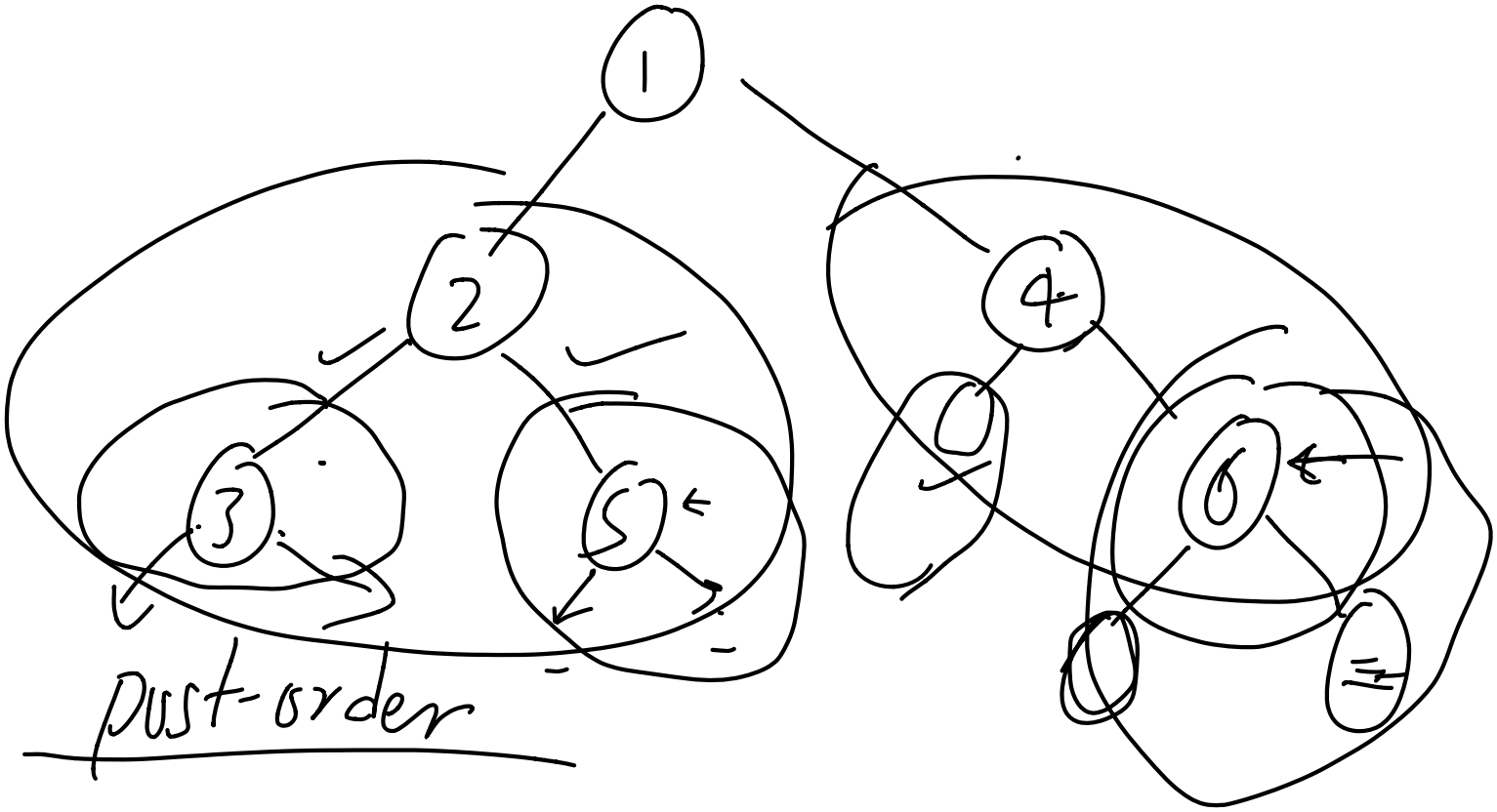
root

left subtree

right subtree

Postorder

1. Visit left-subtree
2. Visit right-subtree
3. Visit root.



Post-order

3 5 2 6 4 1

```
(: post-order : (All (A) (U 'none
                            (Tree A))
                         -> (Listof A)))
(define (post-order t)
  (match t
    ['none (list)]
    [(Tree v l r) (append (post-order l)
                          (post-order r)
                          (list v))]))
```

left-child

right-child

root

Question

If you have binary-search tree you want the output

list to be sorted.

What kind of traversal will you choose?

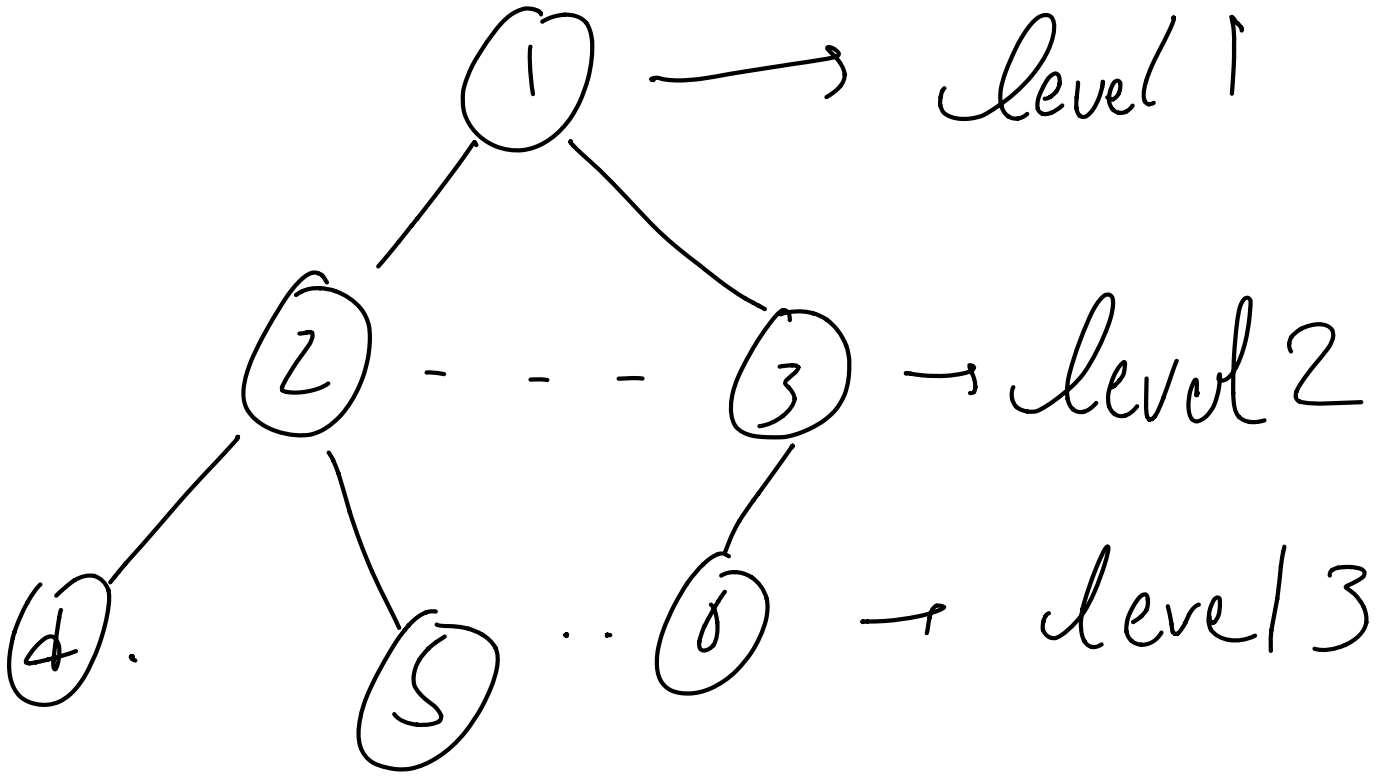
inorder -

You want to visit the smaller values before the larger values

visit left-child
visit root
visit right-child

} inorder

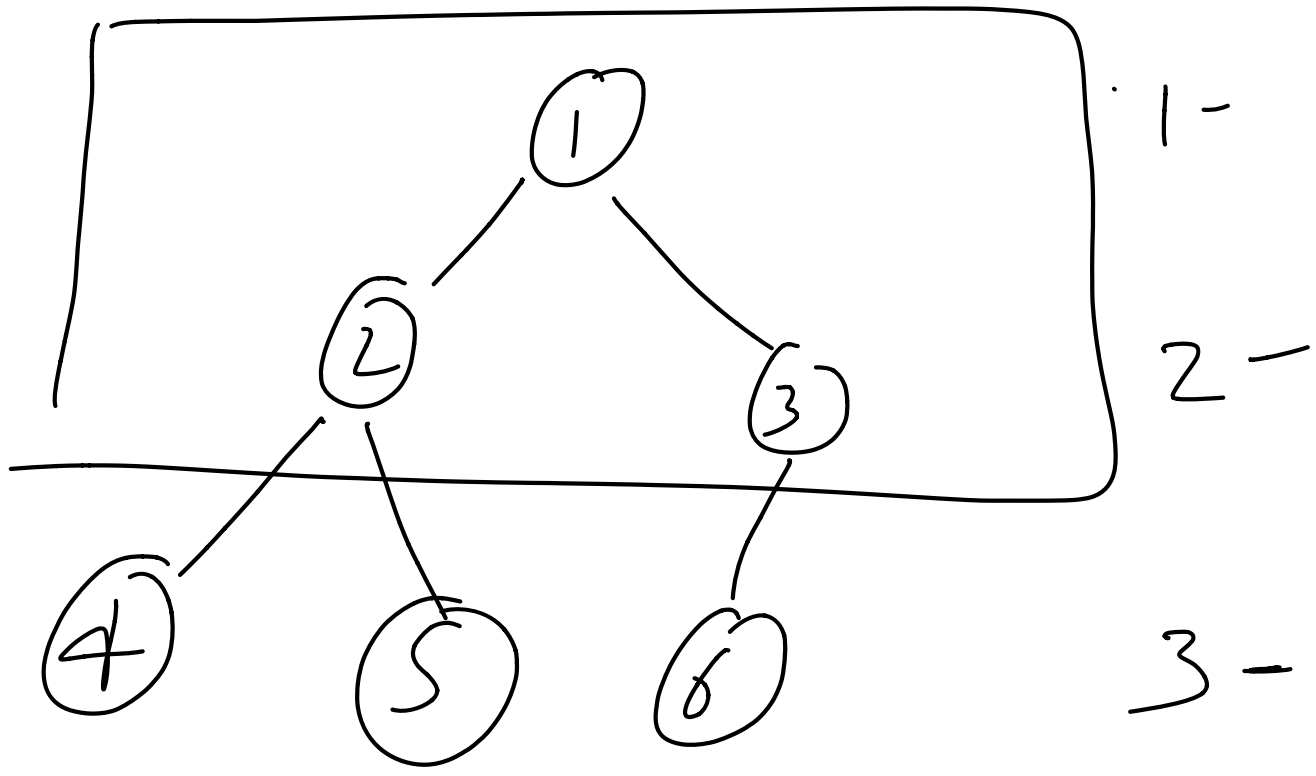
Level-order



1 2 3 4 5 6

"Natural" way of doing this is BFS (Breadth first search)

Using recursion



[1, 2, 3, 4, 5, 6].

First K levels

Fist 3 levels

First do tree-order traversal of k-1

[1, 2, 3] [4, 5, 6]

helper-function will output all nodes in the Kth level

[4, 5, 6]

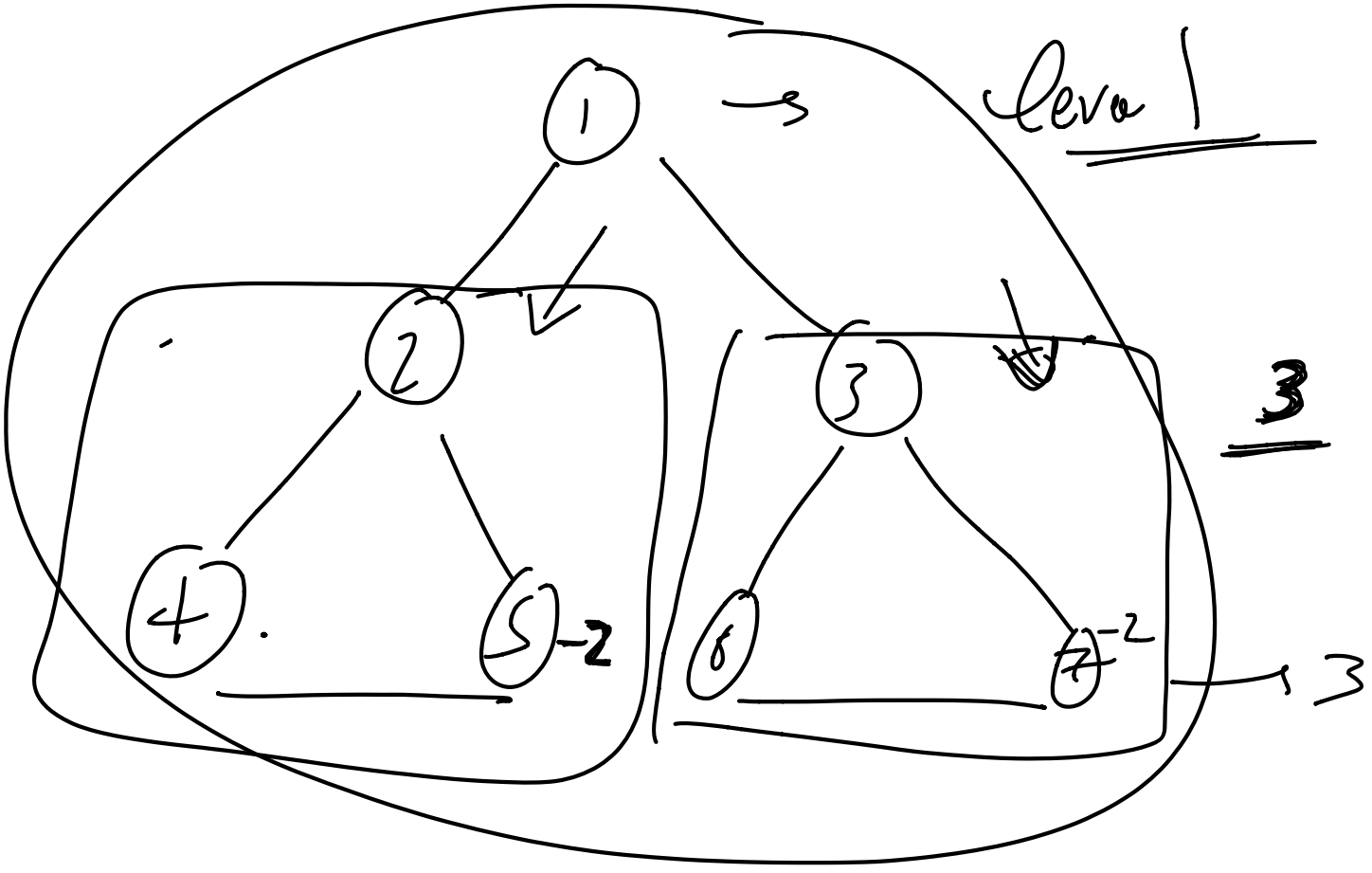
[1, 2, 3, 4, 5, 6].

```
(: level-order : (All (A) (U 'none
                             (Tree A)) Integer
                          → (Listof A)))
;; Output is the list at k level
(define (level-order t level)
```

```
(match t
  ['none (list)]
  [(Tree v l r) (cond
                  [(= level 1) (list v)]
  [else (append (level-order l (- level 1))
                (level-order r (- level 1)))])])
```

recursion

What is the recursive logic



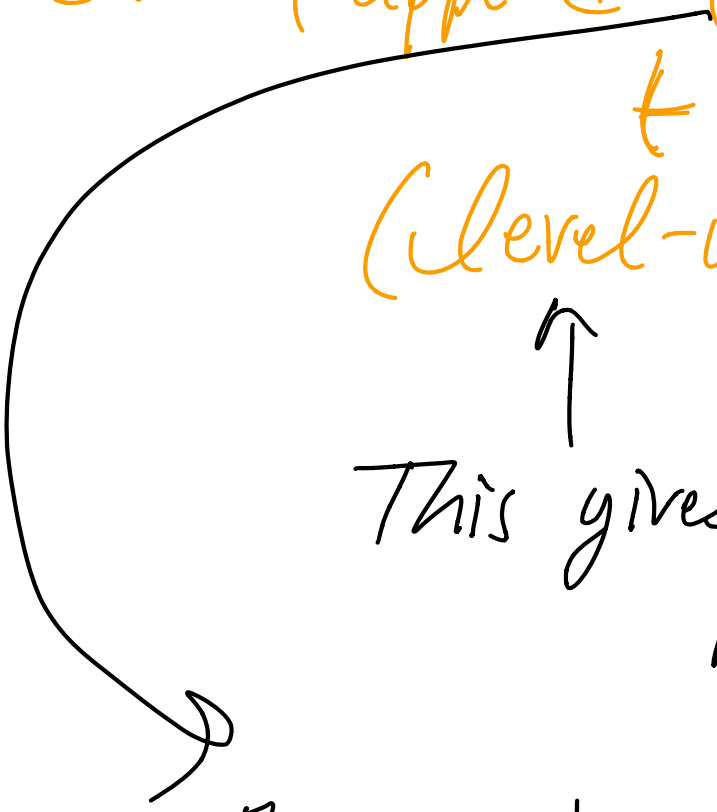
appended (4, 5, 6, 7)

level 2 (2, 3)

level 1 1

```
(: level-order-traversal
   : (All (A) (Tree A) Integer
        -> (Listof A)))
(define (level-order-traversal
          t level)
  (cond
    [(= level 1) (level-order
                   t level)]
```

```
[else (append (level-order-traversal
                t (- level 1))
              (level-order t level))]))
```

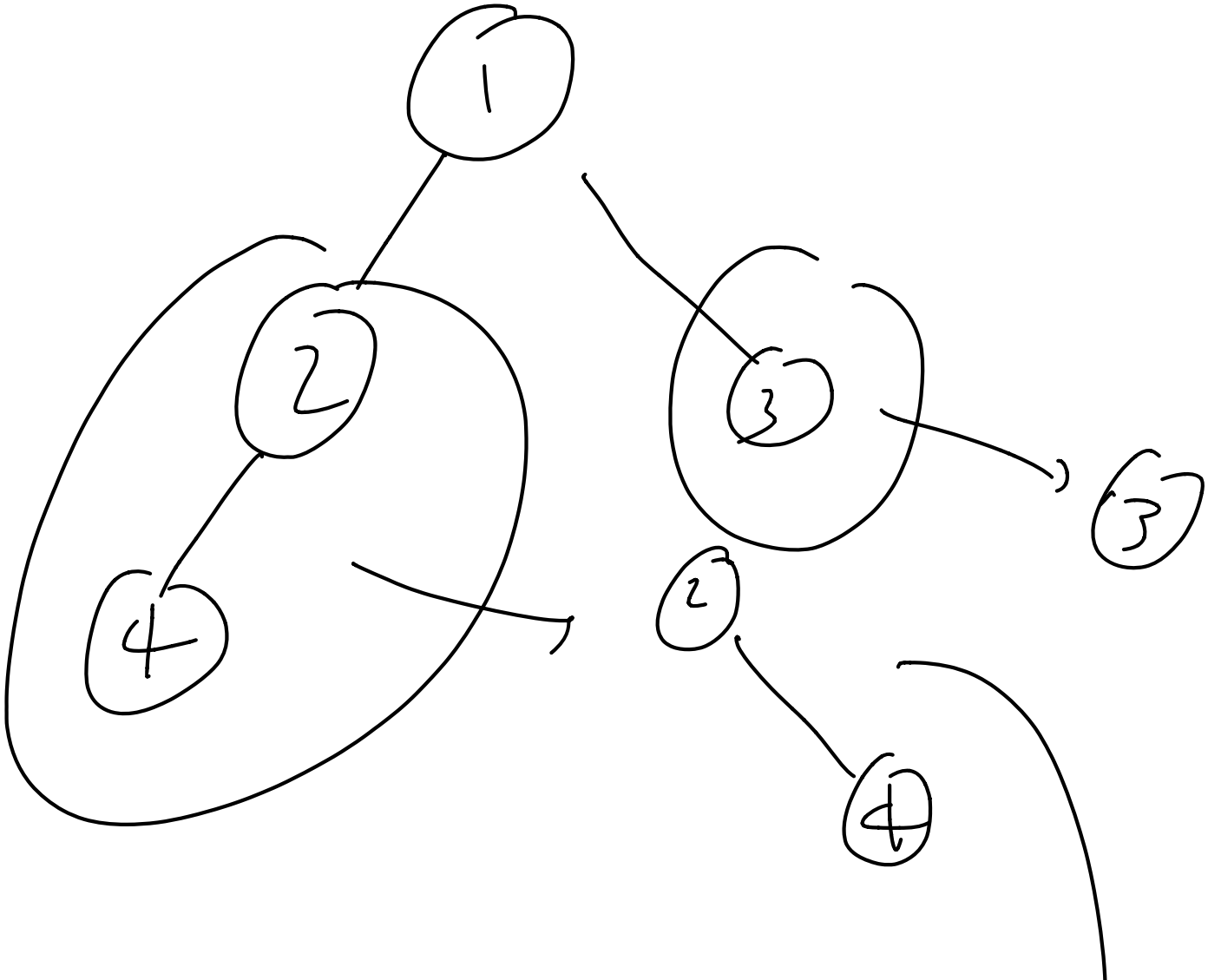


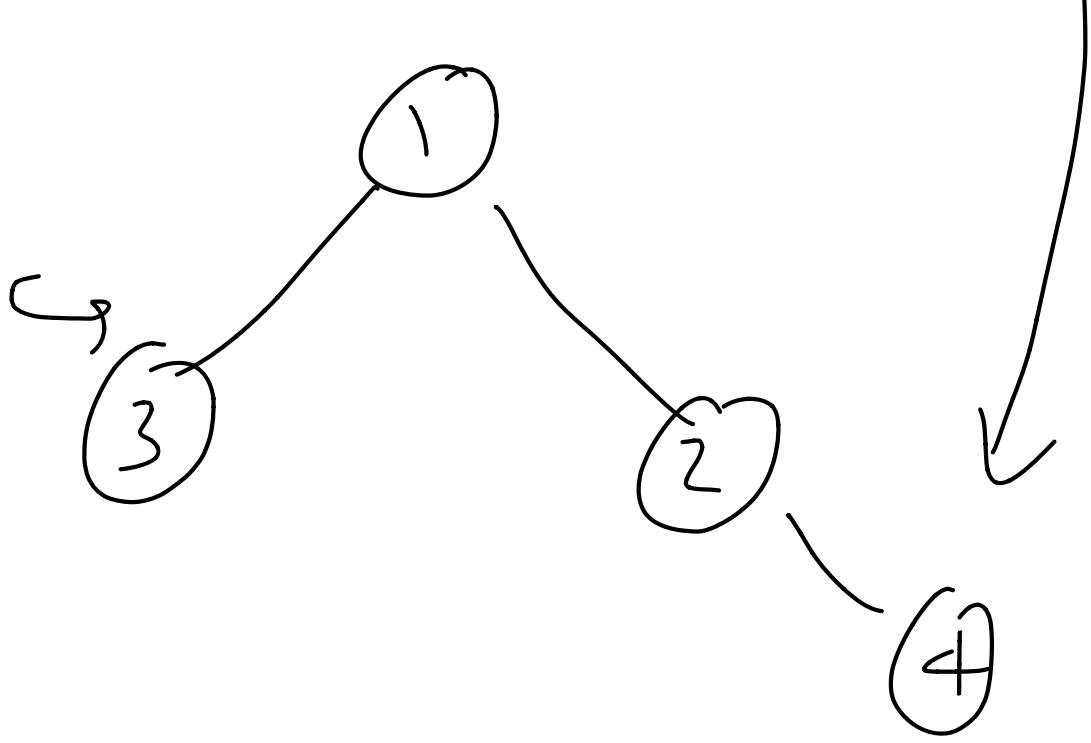
This gives all nodes in the level "level".

This nodes before the level "level"

Example problems on Trees

Write a function that takes a tree as input and outputs its mirror image.





Usual method

Assume you have solved the problem for the left subtree and right subtree

Now solve the problem for the full tree

Make left-subtree — the right-subtree

Make right-subtree left-subtree

```
(: mirror-image : (All (A)
   (U 'none (Tree A)) -> (U 'none
                             (Tree A))))
(define (mirror-image t)
  (match t
    ['none 'none]
    [(Tree v l r)
     (Tree v (mirror-image r)
             (mirror-image l))]))
```

Write a function computes the max-value of node in a tree.

Suppose max-val of left-sub-tree is max_l

max-val of the right subtree is max_r

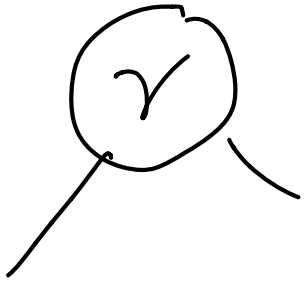
max_value of the Tree
= max(v, max_l, max_r)

v → value of the root.

```
(: max-value : (U 'none
                  (Tree Integer) -> Integer)

(define (max-value t)
  (match t
    ['none -1001] ← Assumption is required.

    [(Tree v l r) (max v (max-value l)
                        (max-value r))]))
```



max_l, max_r

-1001 -1001

max(r, max_l, max_r)

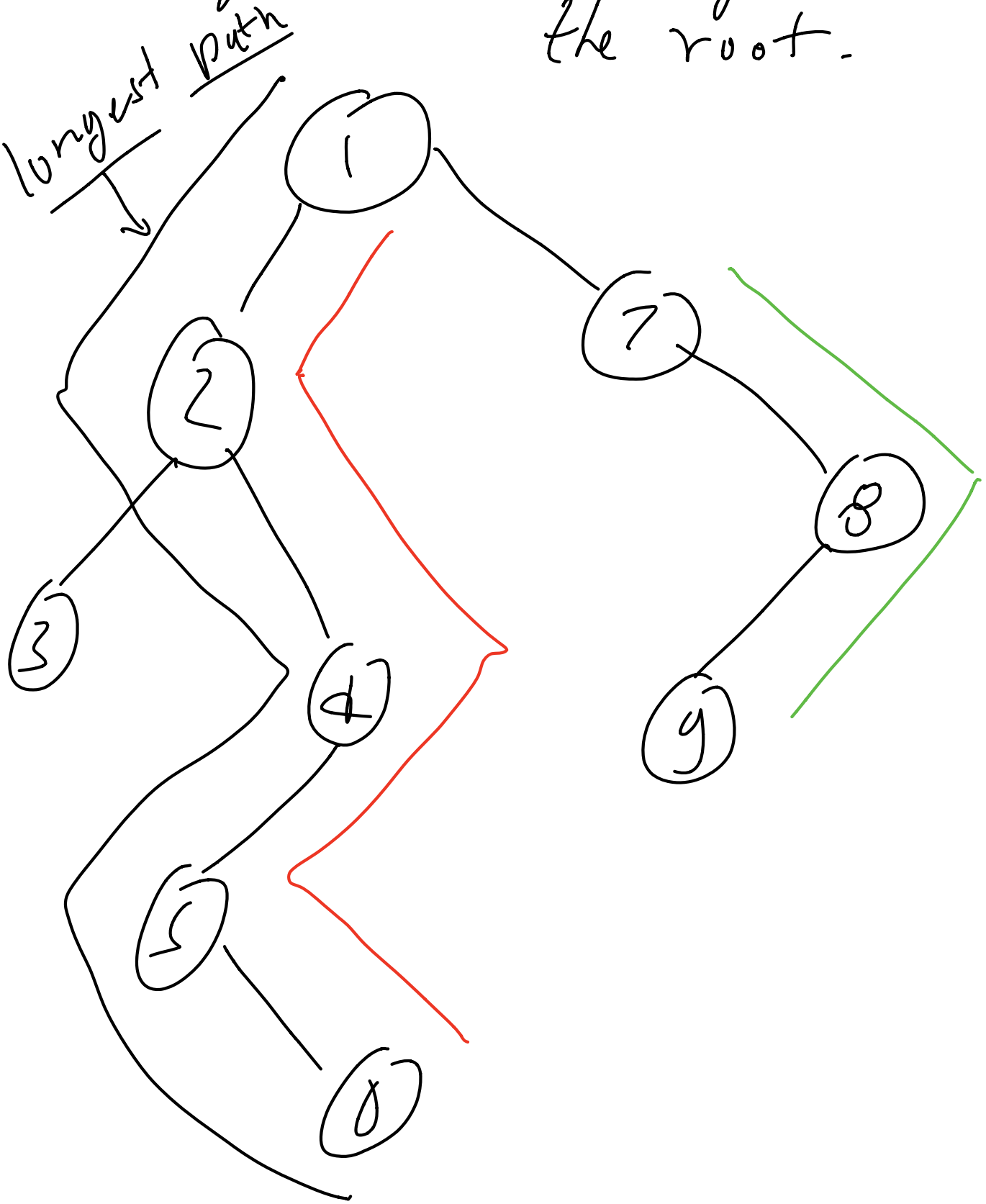
= r

max_l is the maximum of the left subtree which is empty

max_r is the maximum of the right subtree which is empty.

Suppose all my tree values are greater than -1000

Write a function that computes the longest path starting from the root.



Suppose you know the longest path of the left-subtree (longest path from the root of left-subtree) and longest of the right-subtree

```
(: longest-path-tree : (All (A)
(U (Tree A) 'none → (List of A))

(define (longest-path-tree t)
  (match t
    ['none (list)]
    [(Tree v l r) (local {
                    (define path_l (longest-path-tree
                                    l))
```

```
(define path_r (longest-path
                -tree r))
}
if (>= (length path_l)
       (length path_r))
 (append (list v) path_l)

 (append (list v) path_r))]]
```

Vectors :

Example : Write a function

Which reverse a vector

of integers.

Input: (vector 2 4 5 6)

Output: Void

(Vector 6 5 4 2)

Input

Copy my input to a different vector tmp in reverse order

Copy it back to my original

input.

(Vector 2 4 5 6) - input

(Vector 6 5 4 2) - tmp

(Vector 6 5 4 2) - input

```
(: copy-into : (Vector of Integer)
               (Vector of Integer) -> Void)
(define (copy-into v-one v-two)
  (local
    [
     (: n : Integer)
     (define n (vector-length v-one))
```

```
(: copy-into-helper : Integer -> Void)
(define (copy-into-helper i)
  (cond
    [(= i n) (void)]
    [else (begin
            (vector-set! v-two i (vector-ref
                                  v-one i))
            (copy-into-helper (+ i 1)))]))

(copy-into-helper 0))
```

start

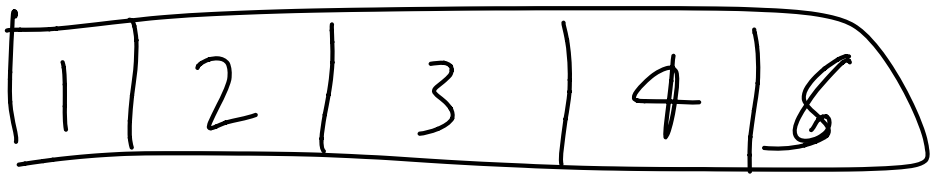
```
(: reverse-into : (Vectorof Integer)
   (Vector of Integer) -> Void)

(define (reverse-into v-one v-two)
 (local
  {
   (n : Integer)
   (define n (vector-length v-one))
  (: reverse-into-helper : Integer -> Void)
   (define (reverse-into-helper i)
    (cond
     [(= i n) (void)]
  [else (begin
          (vector-set! v-two (- n 1 i)
                       (vector-ref v-one i)))]))}
 (reverse-into-helper 0)))
```

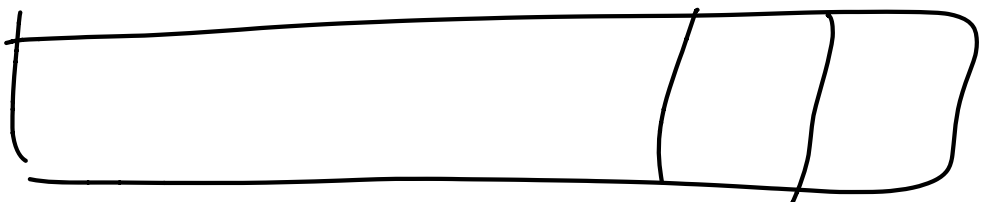
```
(: reverse : (Vectorof Integer) -> Void)
(def (reverse vec)
  (local {(: storage : (Vectorof Integer))
    (define storage (vector-map
                      (lambda ([x : integer]) 0)
                      vec))}
    (begin
      (reverse-into vec storage)
      (copy-into storage vec))))
```

$n-1-i$

v-one $i := 0$ $i=1$ $n-i-1$



v-two



$n-2$ $n-1$