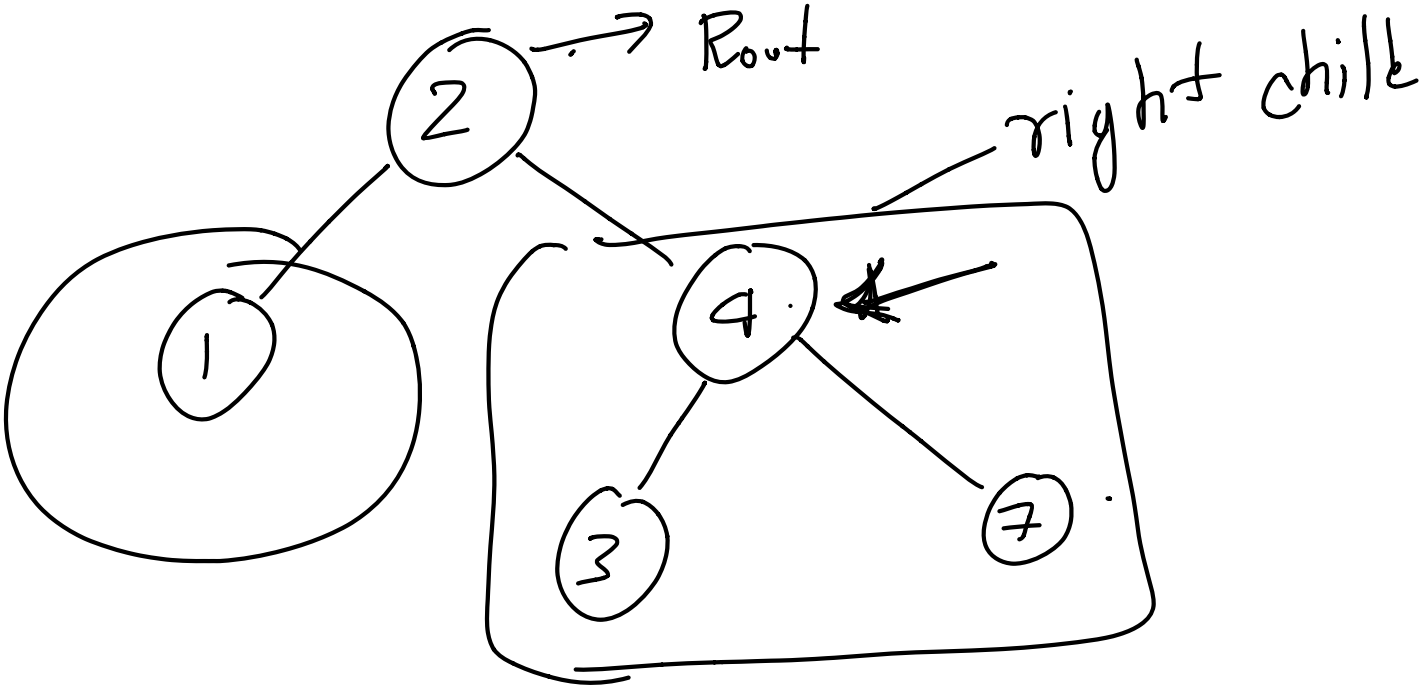
Regarding HW 11

HW 11 changed

Type definition of Graph was changed.
Make sure you use the type definition.

Binary Search Trees

It's a tree whose nodes/vertices satisfy "binary search invariance" property.

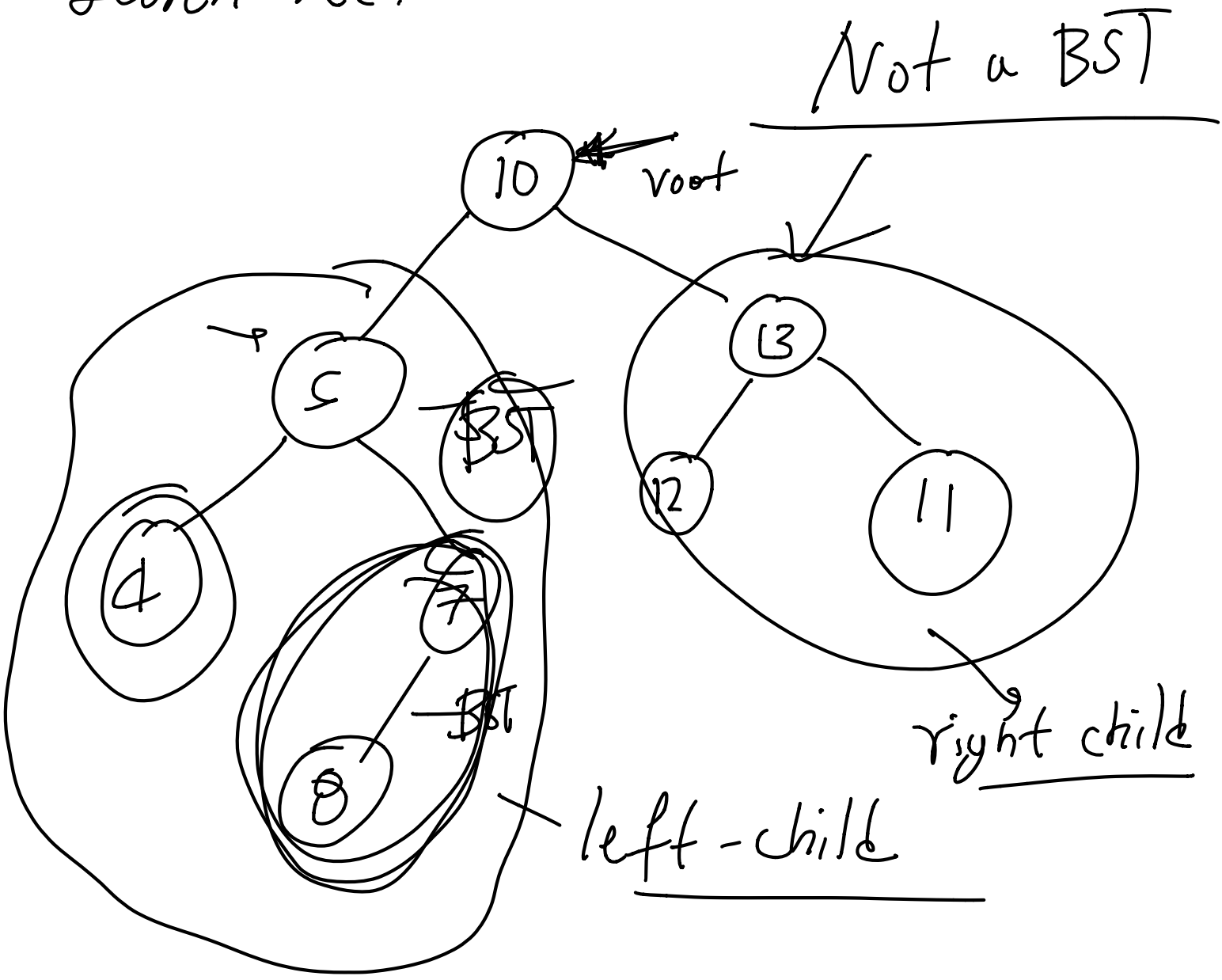


All nodes of the left child of the root are less than value of that node.

All nodes of the right child of the root are greater than or equal to value of that node.

Left child of the root is again a binary search tree.

Right child of the root is again a binary search tree.

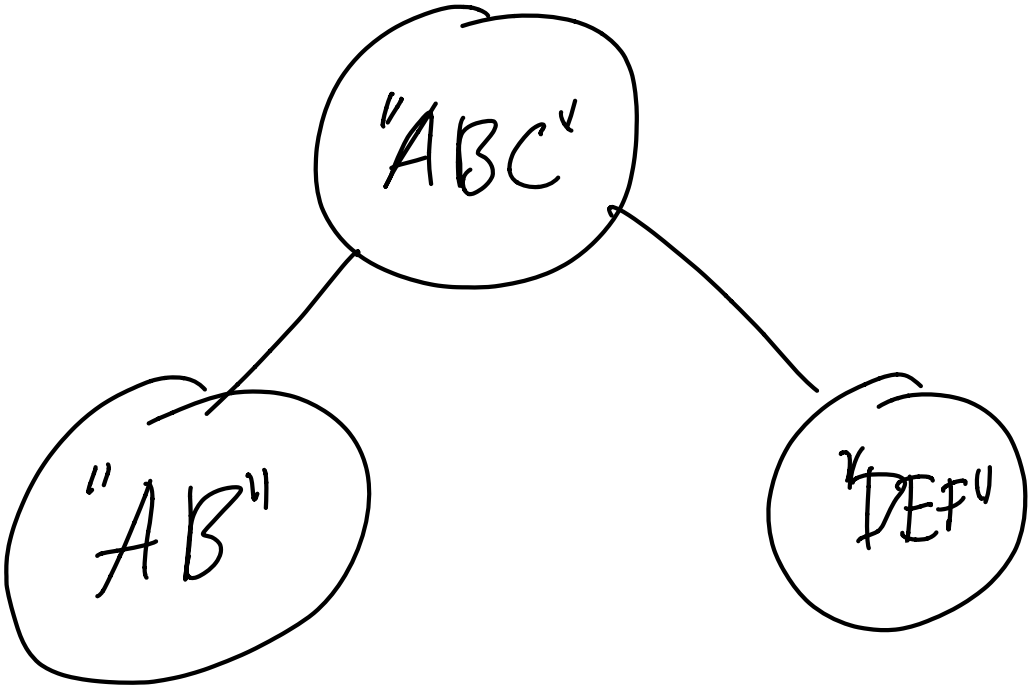


Satisfies condition 1

Satisfies condition 2

Check Left child is a BST

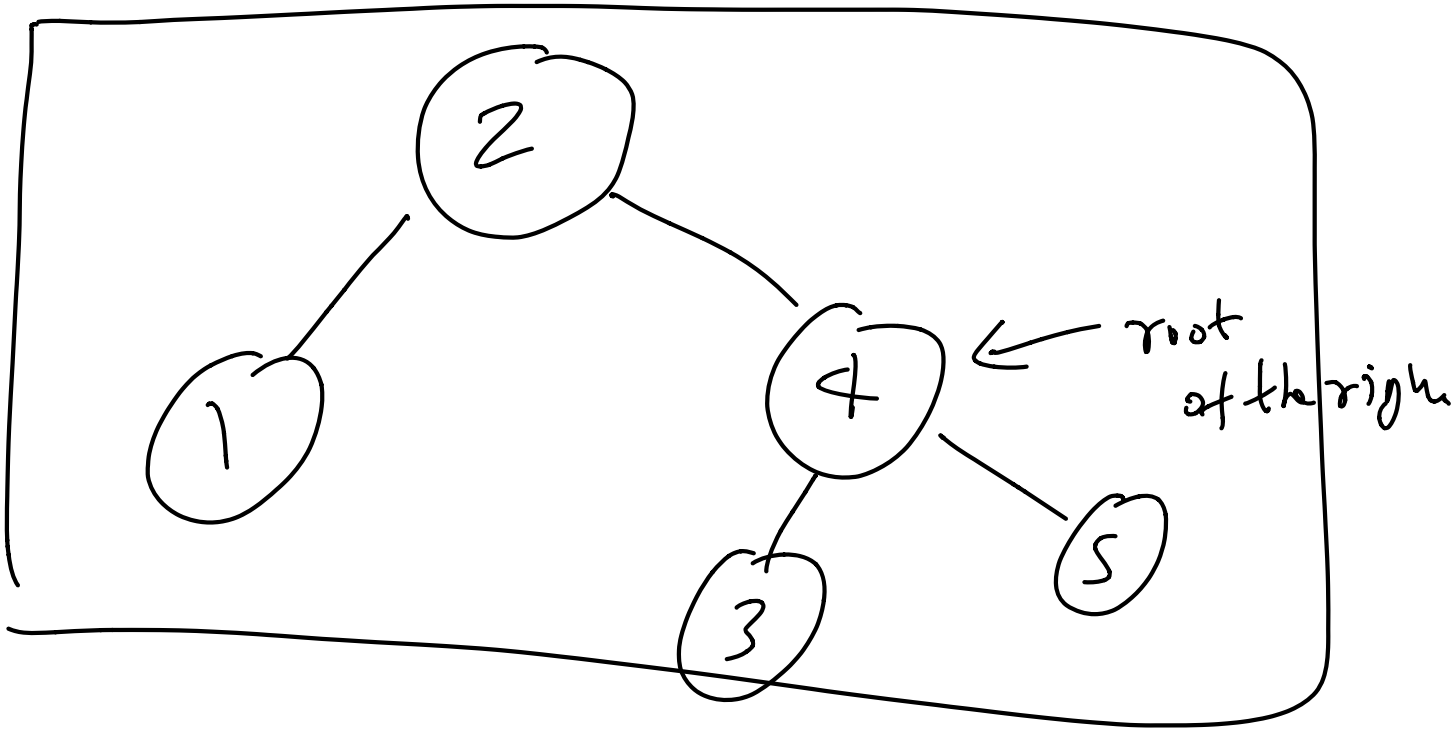
False



"AB" < "ABC" < "DEF"

Racket Definition

```
(define-struct (BST A)
  ([le? : (A A -> Boolean)]
   [bst : (U 'none (Node A))]))

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

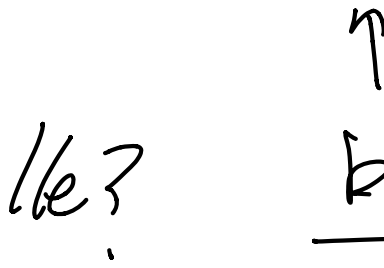


Creak the tree.

```
(: root : (Node Integer))
(define root (Node 2 (Node 1
                           'none
                           'none)
                     (Node 4 (Node 3
                                   'none
                                   'none)
                             (Node 5
                                   'none 'none))))
```

left child

```
(: my-bst : (BST Integer))
(define my-bst (BST <= root))
```



Using lambda

```
(define my-bst (BST (lambda([a : Integer]
                            [b : Integer] (<= a b))
                      root))
```

1. BST search

Search for target

1. Check if root == target (output → root)

2. if target > root

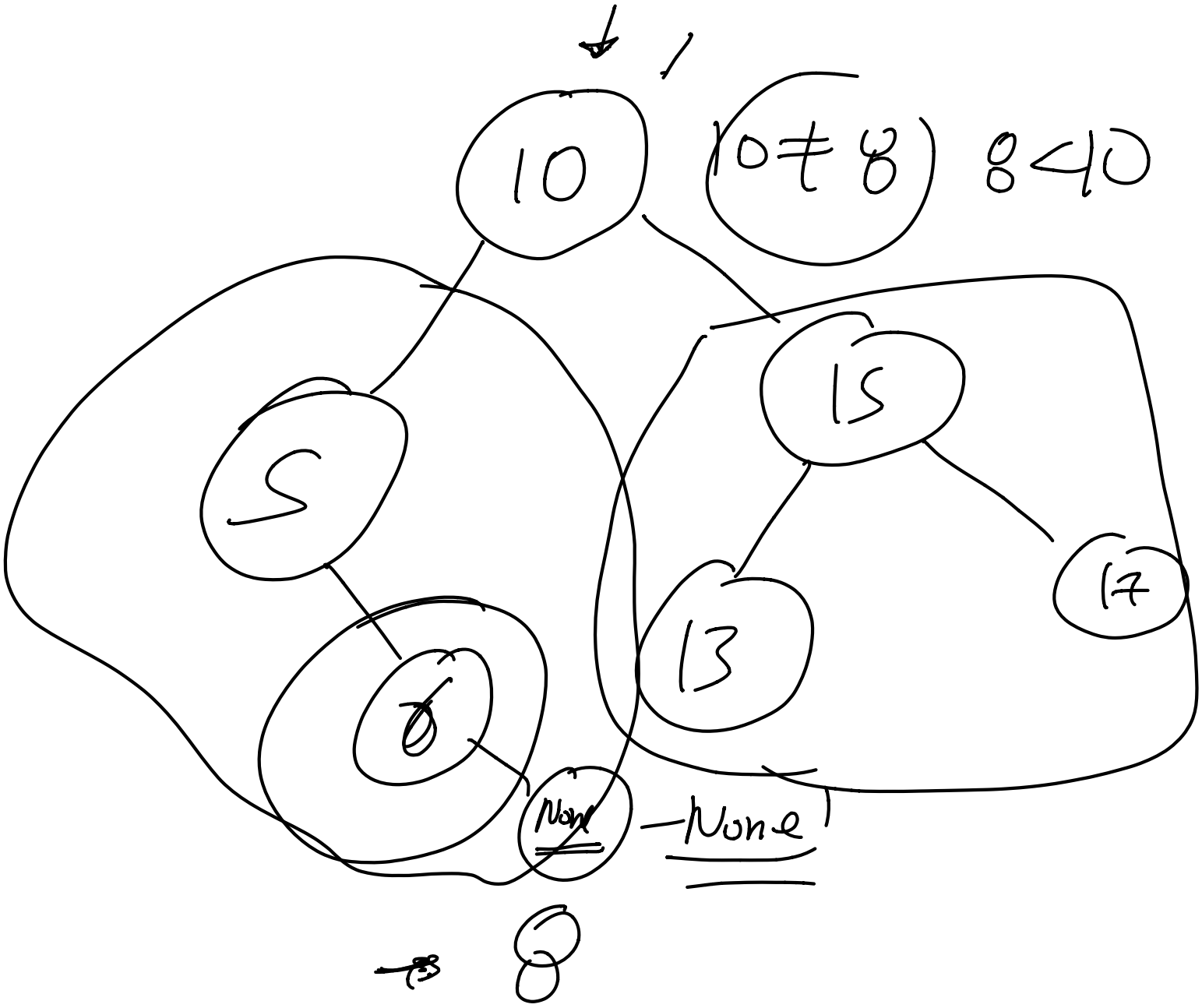
target if its exists can only exist in the right subtree.
Recursively check for target in the right subtree

3. if target < root

if target is less than root
that means if target exists
it can only exist in left
subtree.

Recursively check for target
in left subtree.

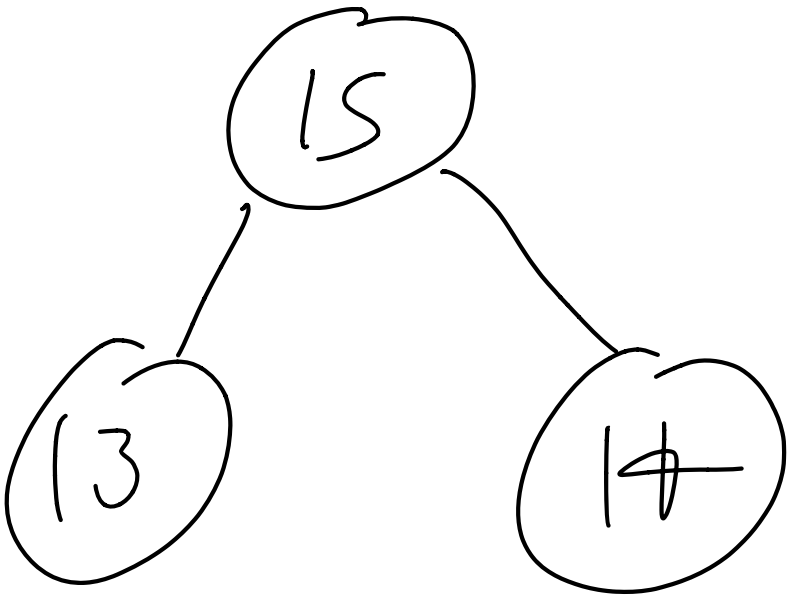
Base.
If my tree is empty.
output None.



15 check with the root 10

15 > 10

go to right subtree




```
(: BST-search : (All (A) (BST A)
                     A -> (U 'none (BST A))

(define (BST-search t target)
  (match t
    [(BST _ 'none) 'none]
    [(BST lte? (Node my-val lc rc))
     (cond
       [(cond (lte? target my-val) (
```

$a \le b$ and $b \le a \Rightarrow a = b$

```
(lte? my-val target)) ?]

[(lte? target my-val)
        (BST-search (BST lte? lc)
                    target)]
[(lte? my-val target) (BST-search
        (BST lte? rc)
                    target)]
[else (error "Error")]))
```

$$\left(aRb, bRa \Rightarrow a = b \right)$$

Insert something in to BST

Suppose you want insert target in to BST.

Base case: BST is none, Create a new BST

Check if root < target

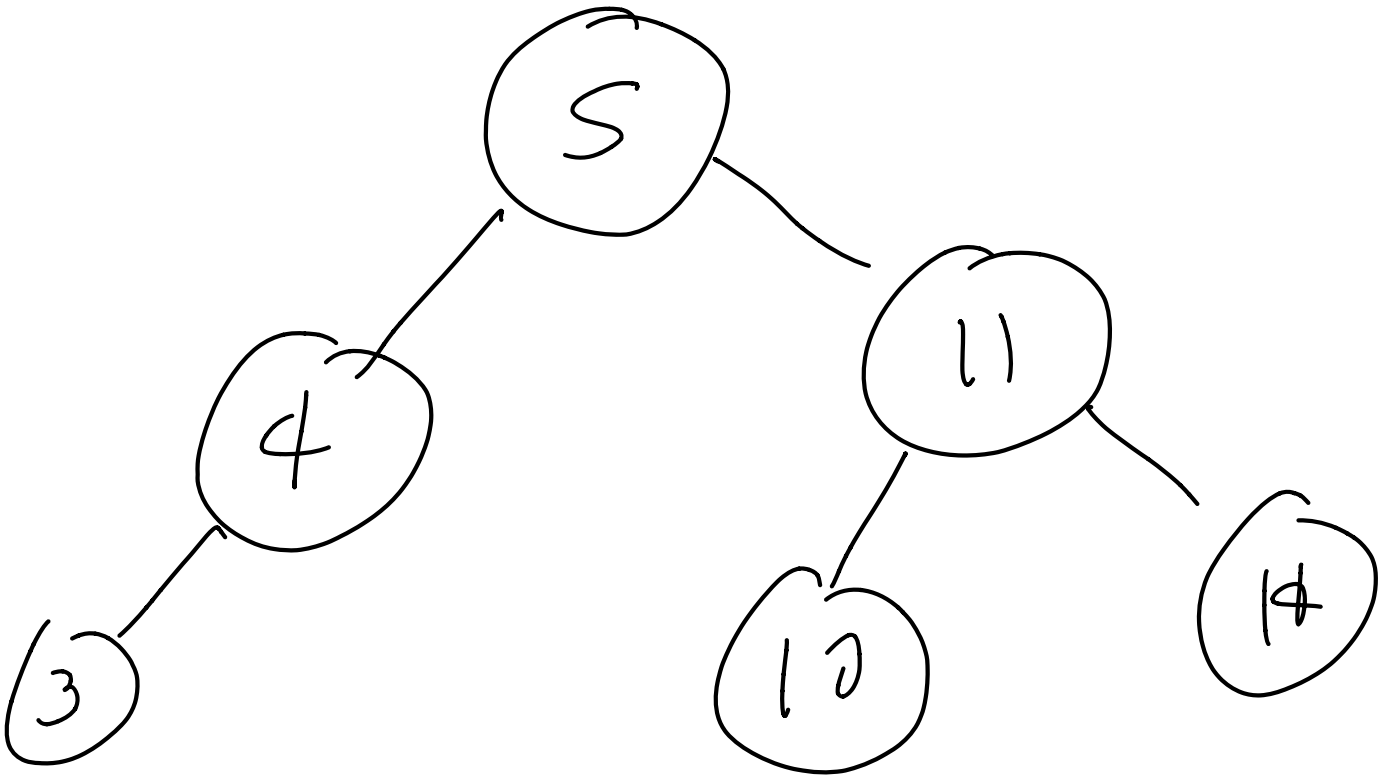
recursively insert target into the right child.

if target < root
recursively insert target
into the left child.

```
(: BST-insert : (All (A) A (BST A)
      → (BST A)))
(define (BST-insert value T)
  (local
    [
     (lte? : A A → Boolean)
     (define lte? (BST-lte? T))

     (: node-insert : (U 'none (Node A))
                      A → (Node A))
     (define (node-insert node value)
```

```
(match node
  ['none (Node value 'none 'none)]
[(Node my-val lc rc)
  (cond
    [(<= value my-val)
       (node-insert lc value)]

    [(<= my-val value)
       (node-insert rc value)]
    [else (error "Error")])])

(BST lte? (node-insert (BST-bst t)
                       value)))
```

Deletion



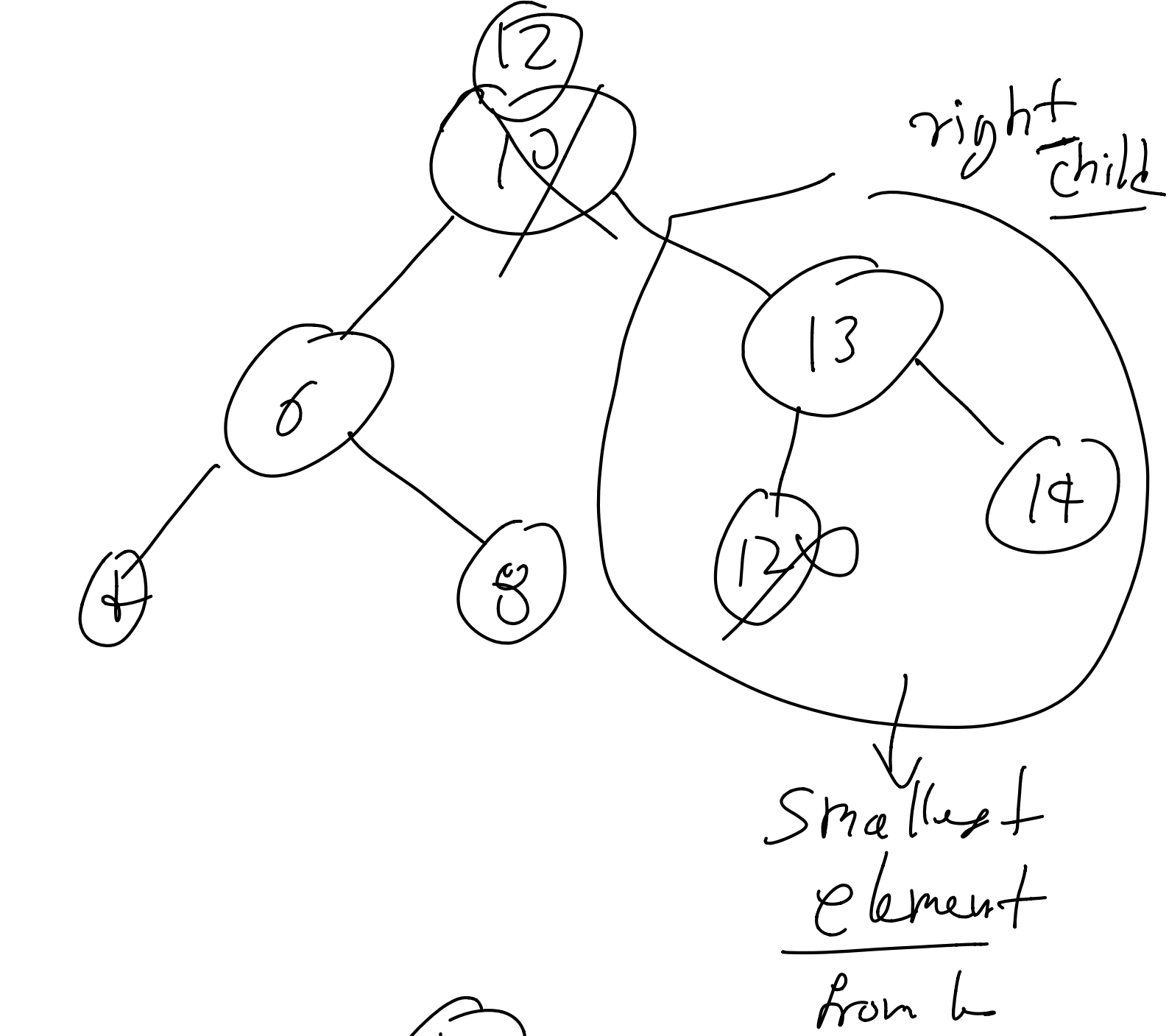
Step 1: If target > root

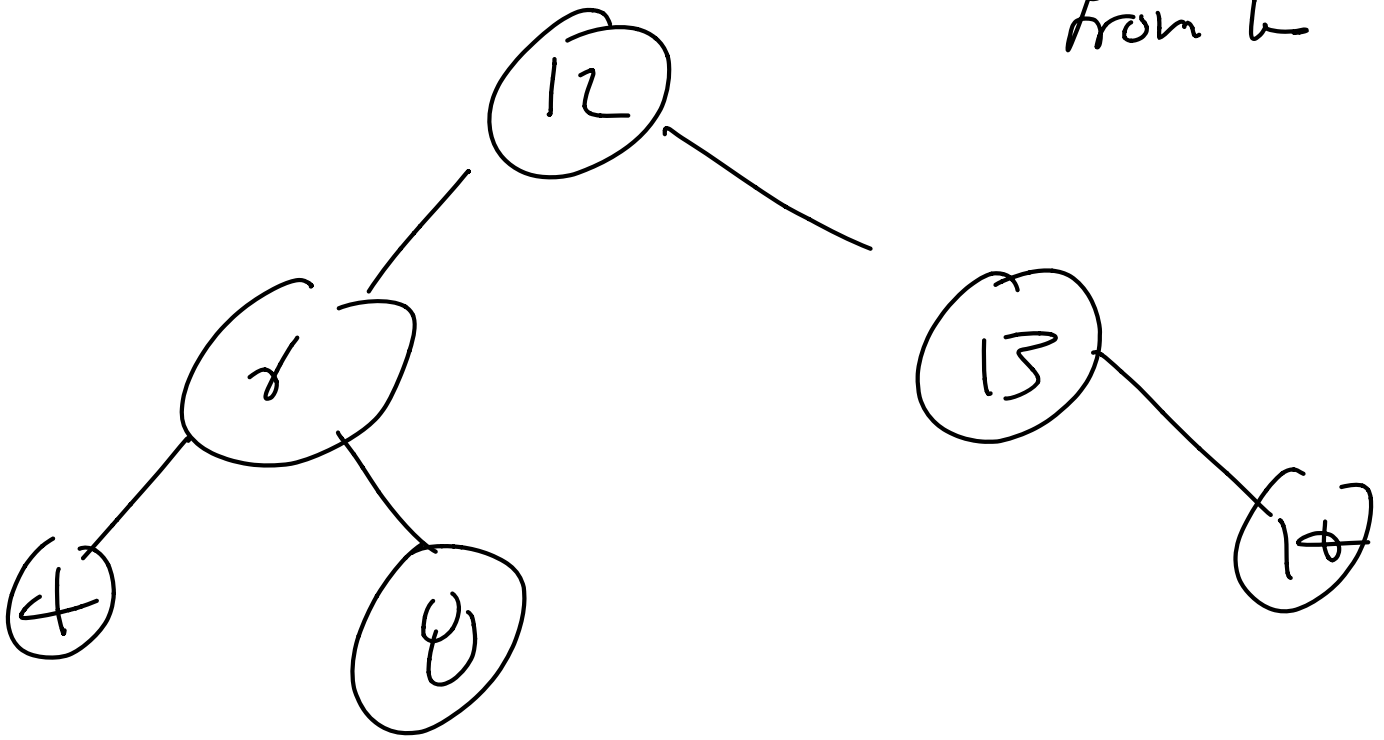
Then delete from the right subtree

Step 2: If target < root

Then delete from the left subtree

If the target is root?


12
10
right child
6
13
4
8
12
14
Smallest element from h


12
6
13
4
8
14

If target == root

Find smallest in the right child → s

Delete s from the right child.

make s the root

Create the tree

```
(: BST-delete : (All (A) A (BST A)
                    → (BST A)))

(define (BST-delete value t)
  (local
    {
     (: lte? : A A → Boolean)
     (define lte? (BST-lte? t))

     (: node-min : (Node A) → A)
     (define (node-min node)
       (match node
         [(Node v 'none r) v]
         [(Node v l r)
          (node-min l)]))
```

← left child is missing

```
(: root-delete : (Node A)
   → (U 'none (Node A)))
(define (root-delete node)
  (match node
  [(Node _ 'none r) r]
  [(Node _ l 'none) l]
  [(Node v l r)
   (local {(define min-val (node-min r))}
    (Node min-val l (node-delete
                     min-val r)))]))

(: node-delete : A (U 'none
                      (Node A)) → (U 'none
                                     (Node A)))
(define node-delete value node)
```

no left child

no right child

```
(match node
  ['none 'none]
  [(Node my-val lc rc)
(cond
 [(and (lte? value my-val)
       (lte? my-val value))
  (root-delete node)]
 [(lte? value my-val)
  (Node my-val (Node-delete value lc) rc)]
 [(lte? my-val value)
  (Node my-val lc (Node-delete value rc))]
```

value == root. value

root delete

root value

Delete from the left child

root value.

Delete from right child

```
[else (error "Error")]]]}
```

```
(BST-left? (node-delek value
                (BST-bst t))))
```

Graphs — Undirected version

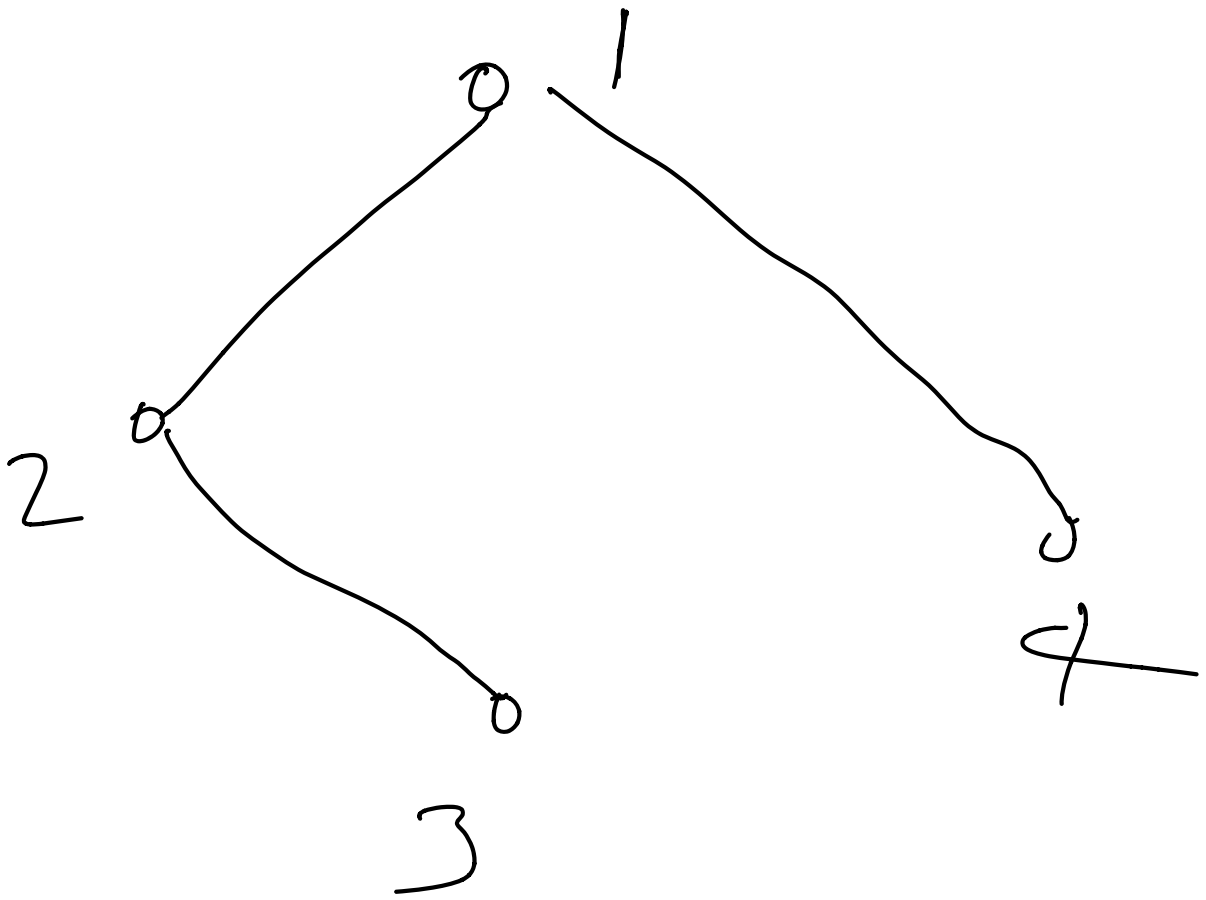
$G = (V, E)$

→ vertices

$V = \{1, 2, 3, 4\}$

$E = \{\{1,2\}, \{2,3\}, \{4,1\}\}$.

$\{1,2\} = \{2,1\}$



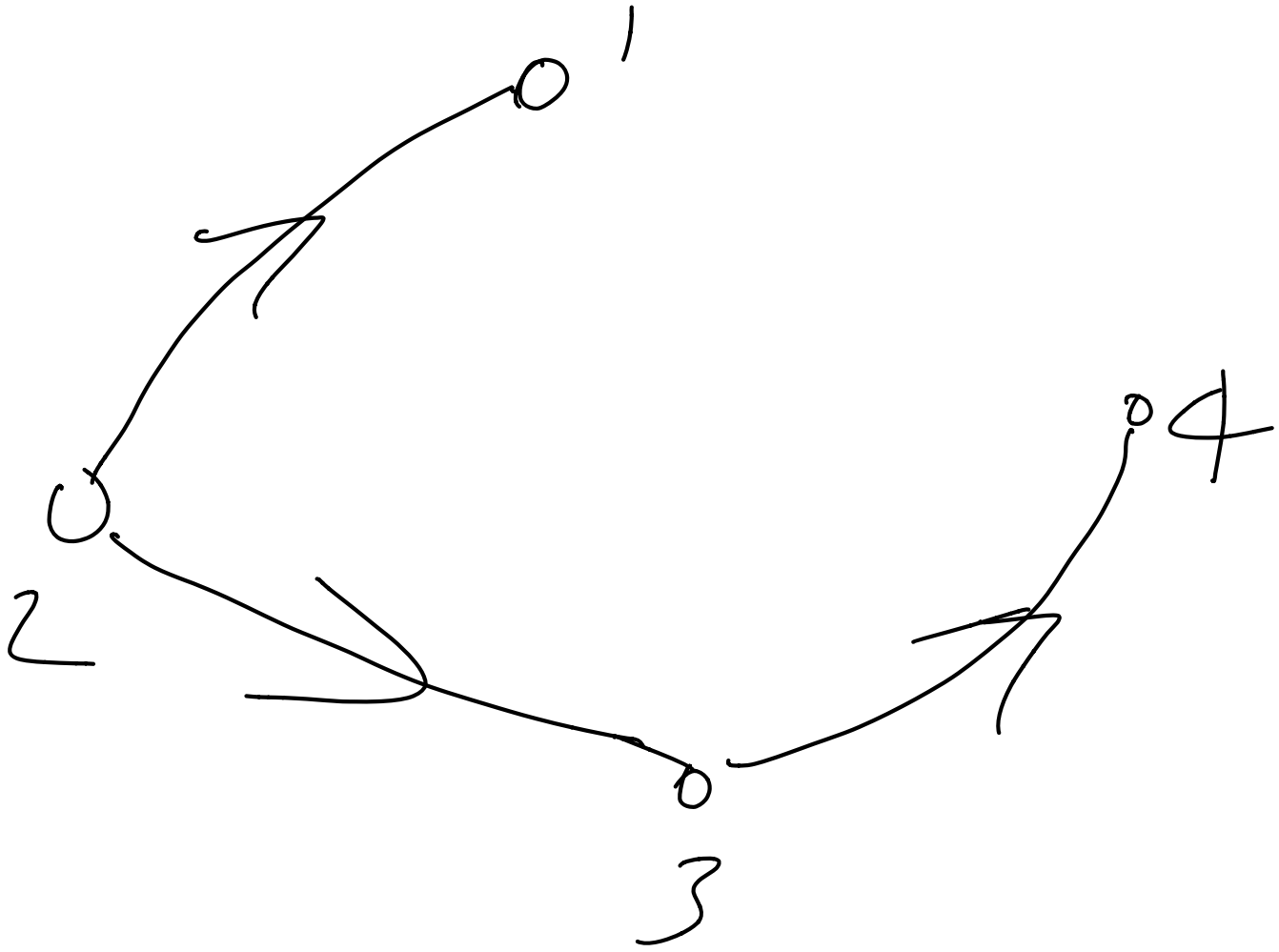
$G = (V, E)$

$V = \{1, 2, 3, 4\}$

$E = \{(2,1), (3,4), (2,3)\}$

↑ ordered pair

$(2,1) \neq (1,2)$



Adjacency list

1 - [ ]

2 - [1, 3]

3 - [4]

4 - [ ]

0	[ ]
1	[1 3]
2	[4]
3	[ ]

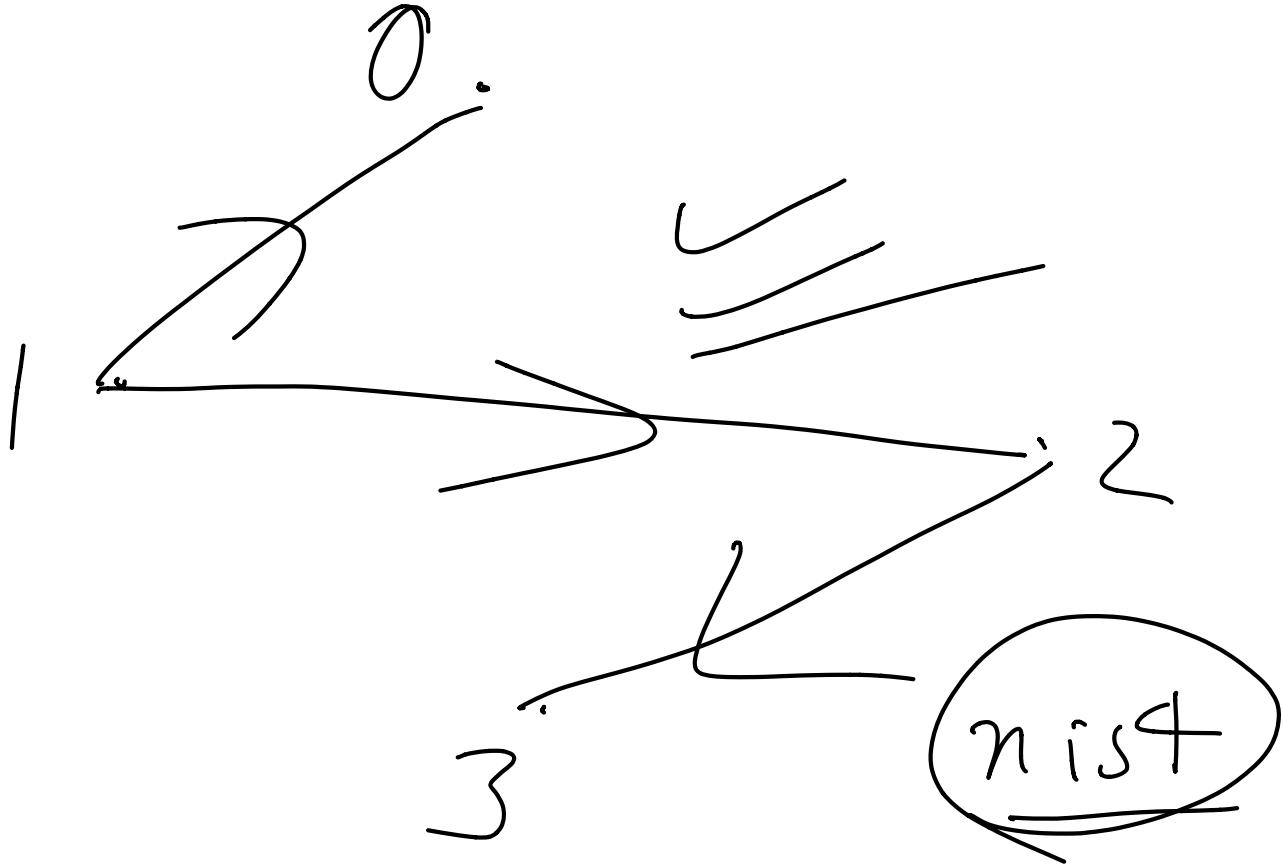
We will adjacency list representation.

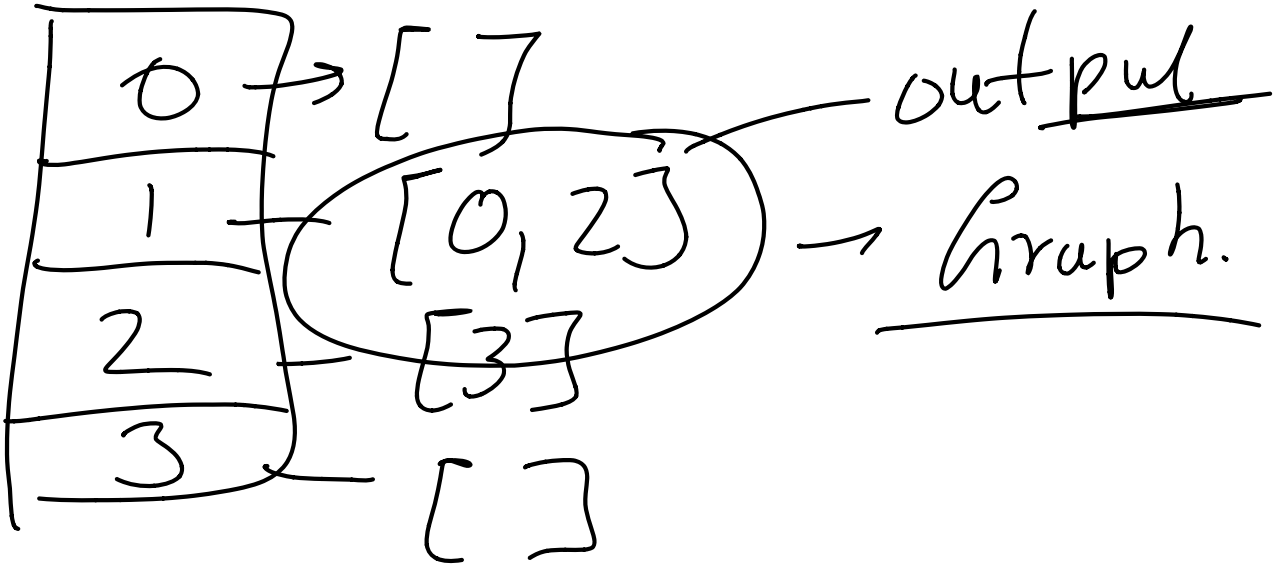
```
(define-type Vertex Integer)
(define-struct Graph
  ([n : Integer]   ⇒ number of nodes
   [adj : (Vectorof (Listof Vertex))]))
```

Why not use list of list
Because vector-ref is O(1)
list-ref is not.

Example: Write a function
which takes a Graph as

input and outputs
the list of vertices
adjacent to that vertex.



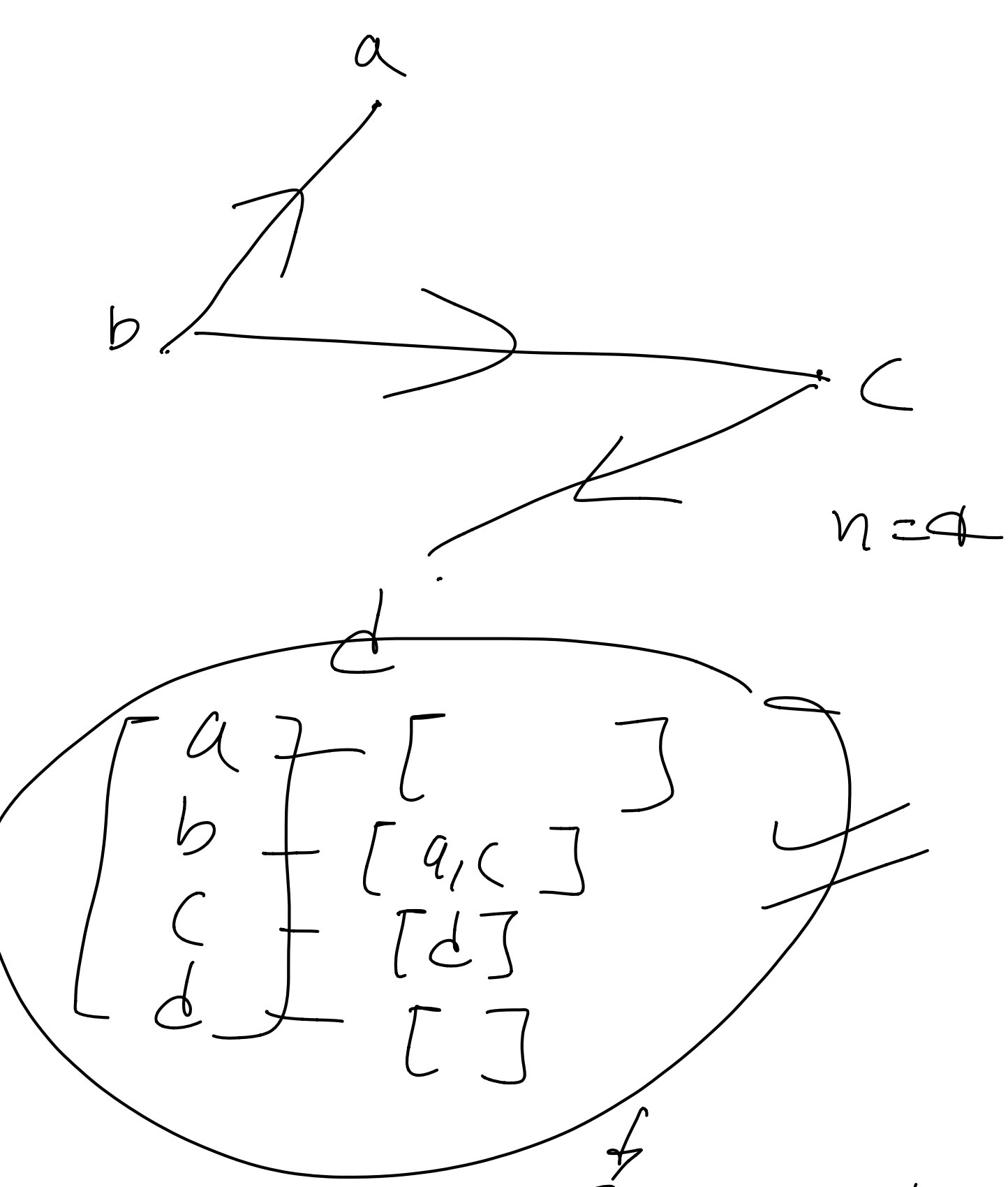


(vector (list) (list 0 2) (list 3)

Vertex 1 (list))

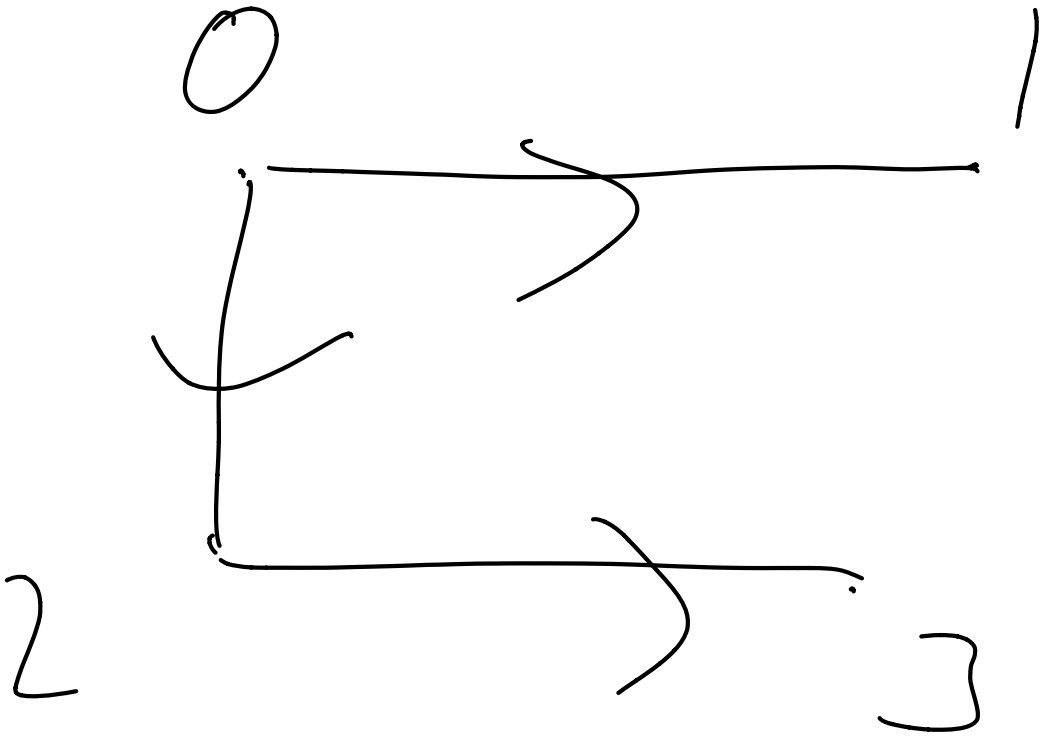
Output should be
[0, 2]

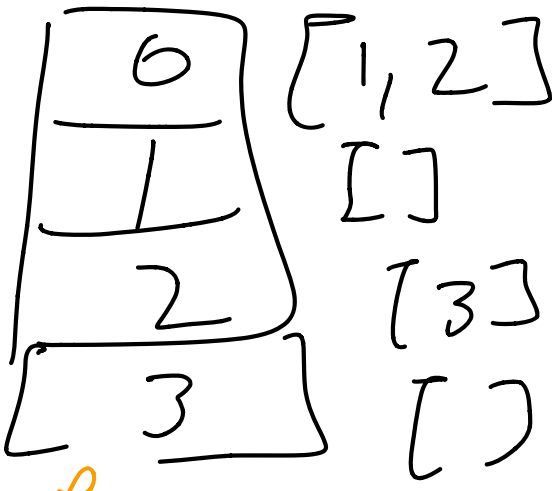
```
(: get-nbrs : Graph Vertex
   → (Listof Vertex))
(define (get-nbrs g v)
  (vector-ref (graph-adj g) v))
```



hash map [e] — output a list

Graph example

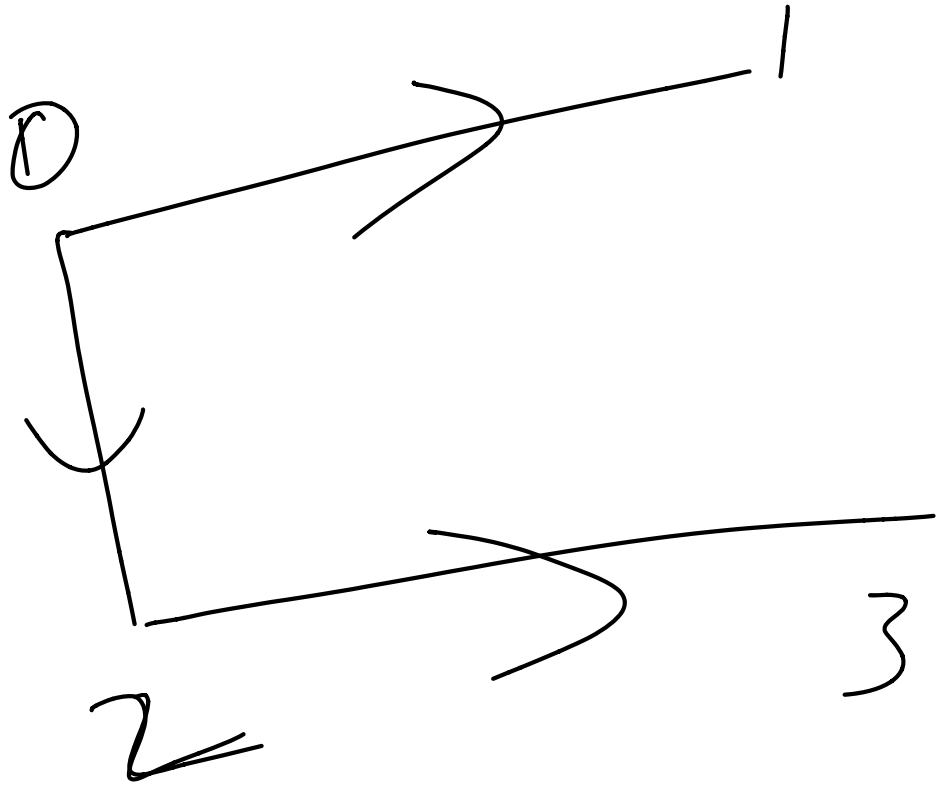





```
(define my-graph (graph 3
                        (vector (list 1 2) (list)
                                (list 3) (list))))
```

in-degree of a vertex is number of incoming edges into the vertex

out-degree of a vertex is the number of outgoing edges from the vertex.



out-degree of 0 is 2
in-degree of 0 is 0

out-degree of 2 is 1
in-degree of 2 is 1

Problem 2 BST

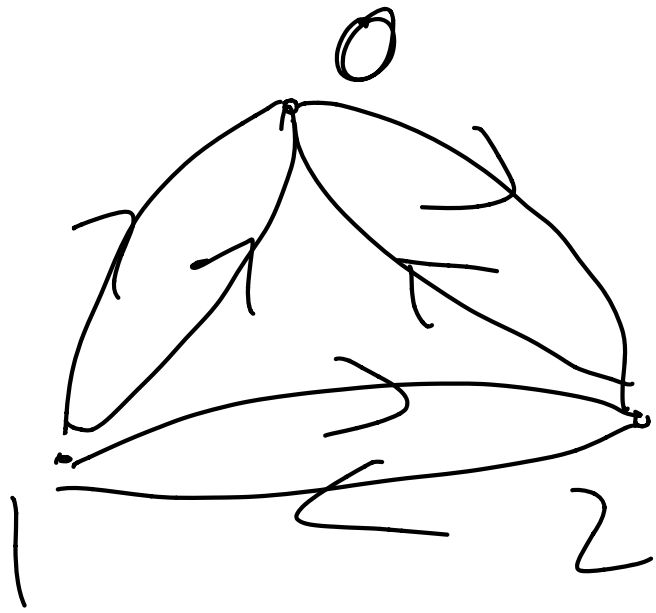
BST-delete function

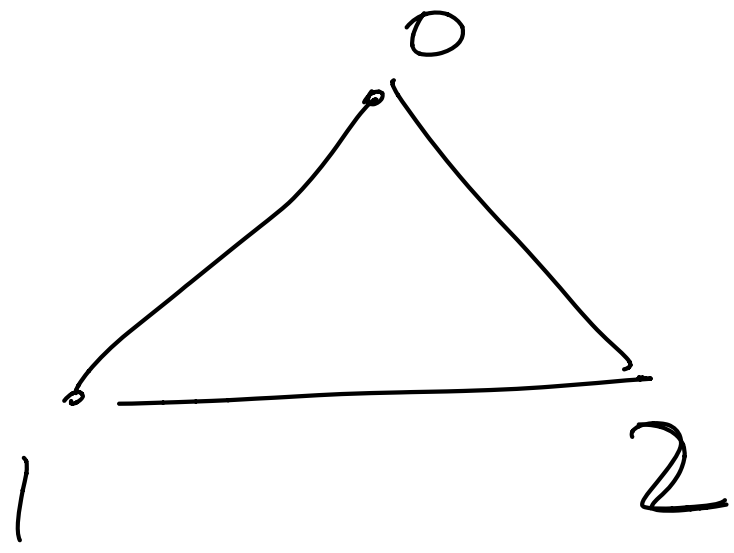
BST.rkt lab last

Save this file in tk include

(require "./include/BST.rkt")

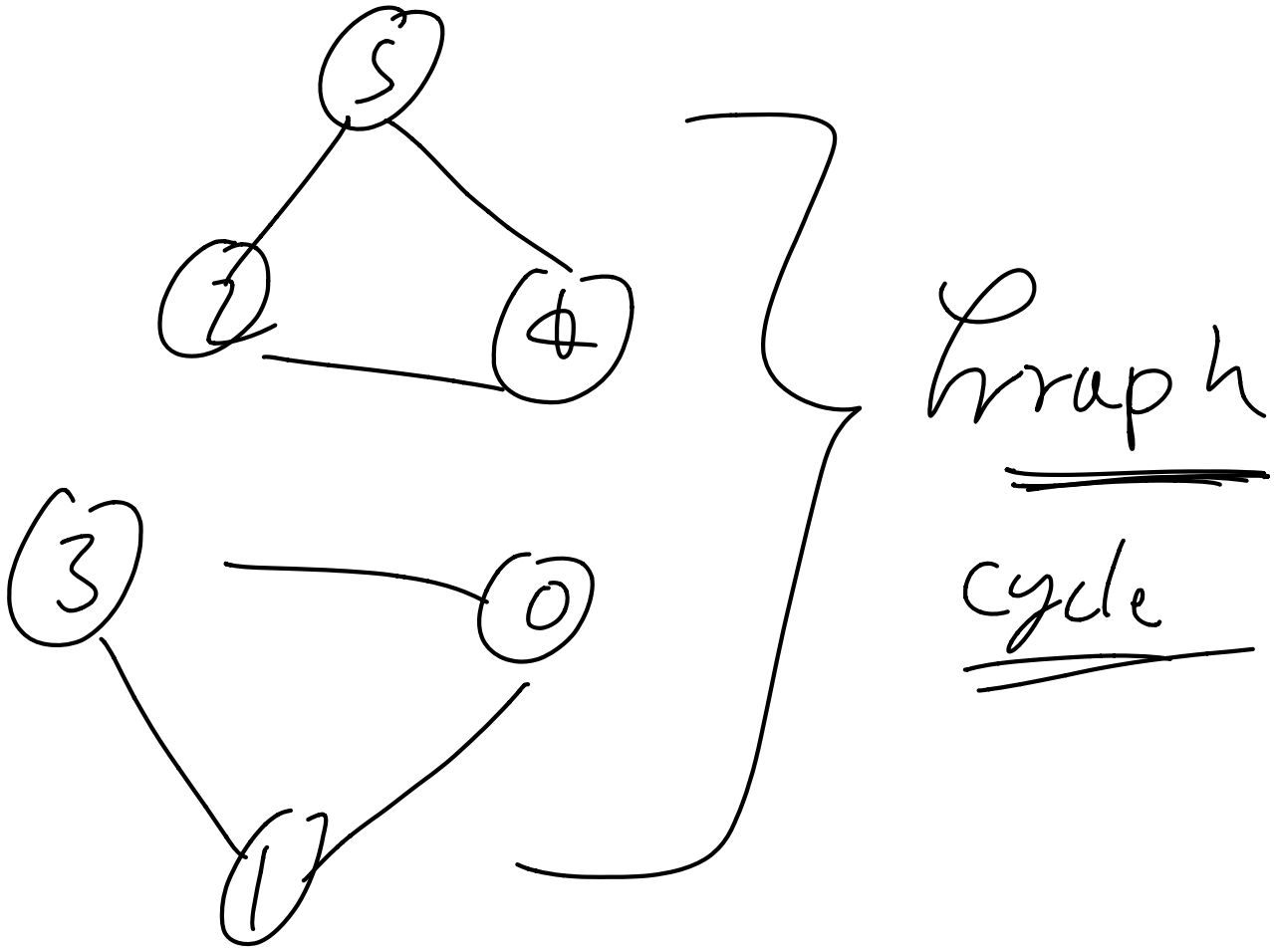
Undirected graph





in-degree of 2 is 2

out-degree of 2 is 2


5
2
4
3
0
1
Graph
cycle