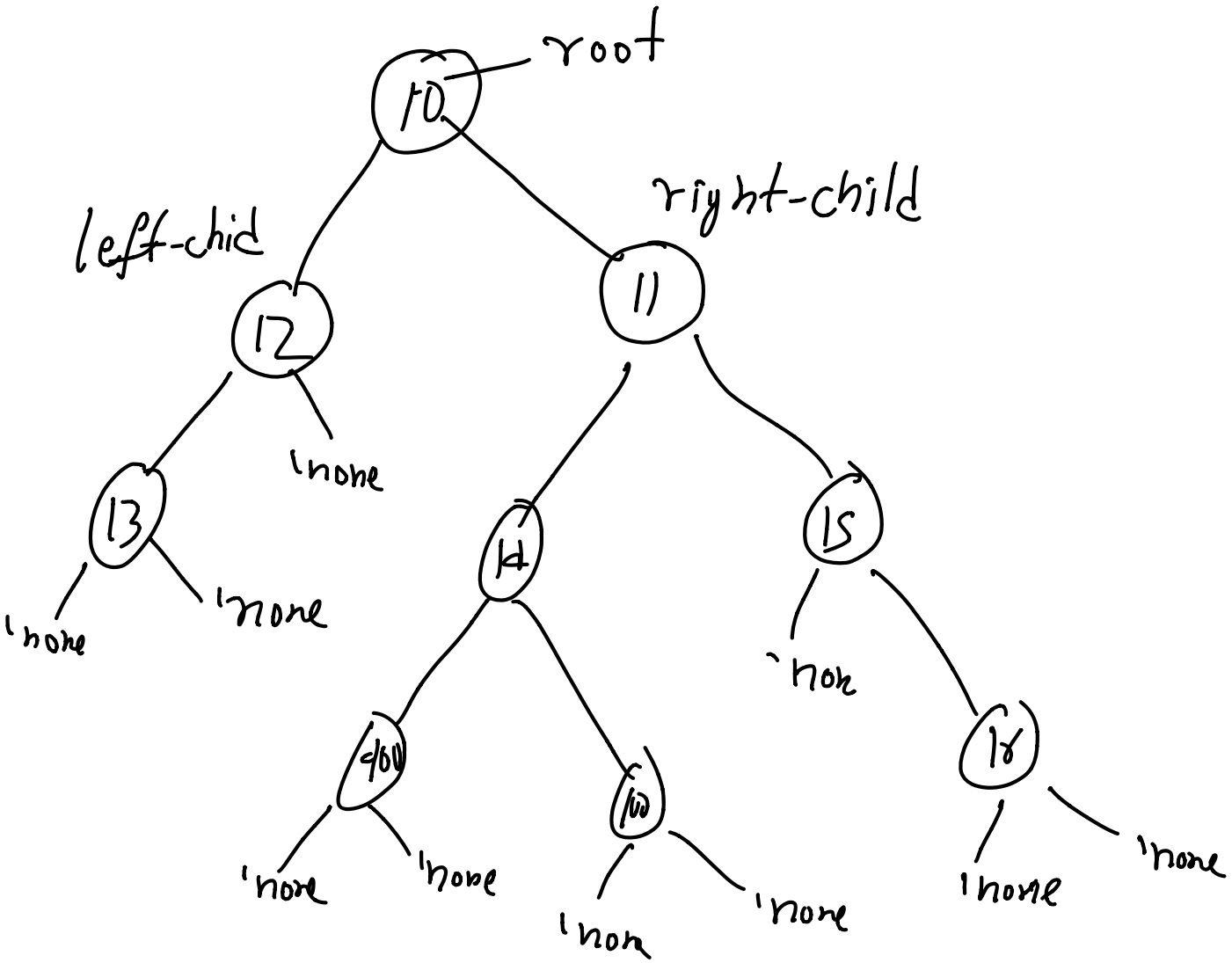
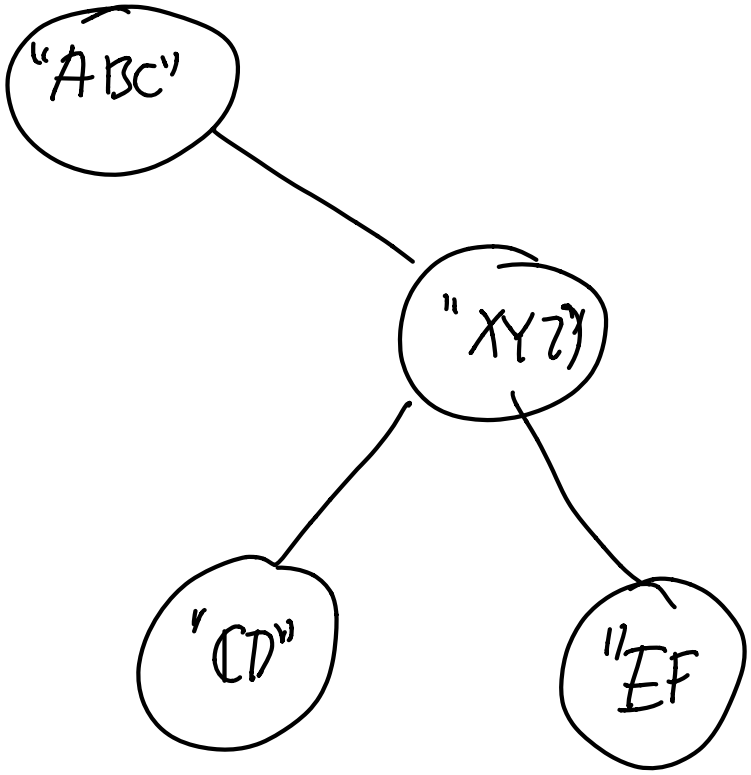
# Lecture 6

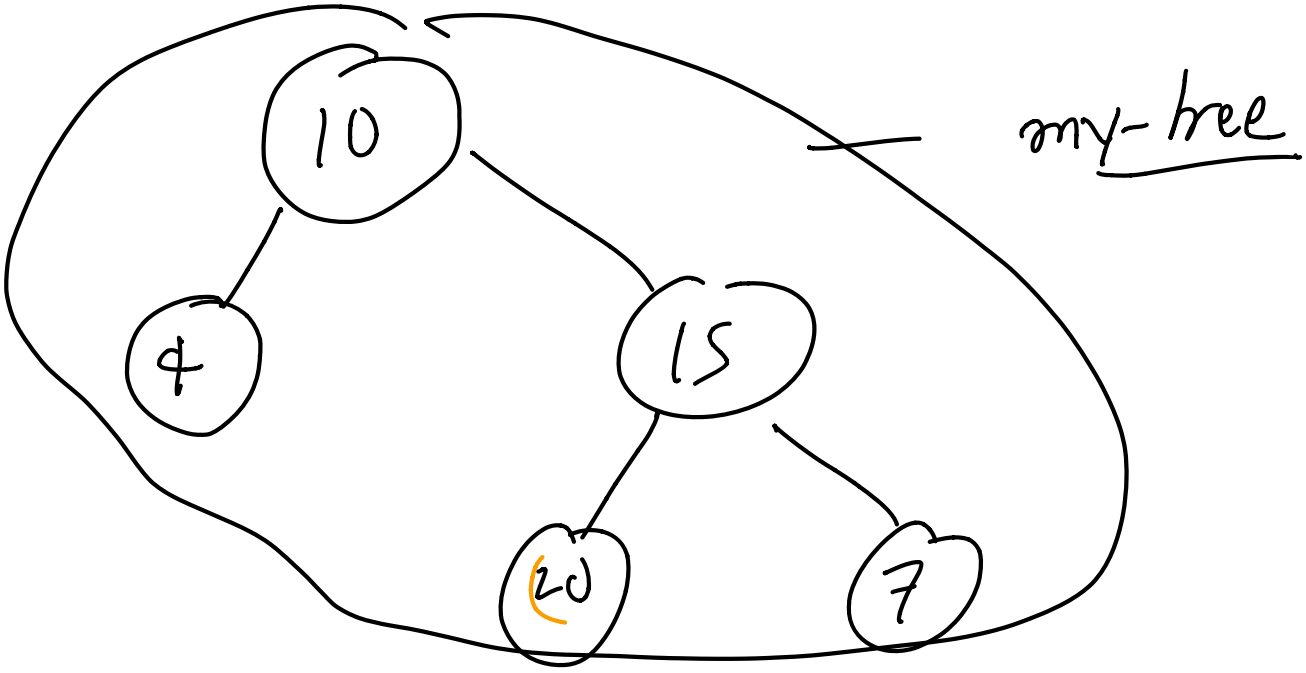
## Trees (Binary Trees)

Tree is a recursive data sbuchure.






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




```
(: right : Tree )
(define right (Tree 15 (Tree 20 'none
                            'none)
                       (Tree 7  'none
(: left : Tree)                 'none)))
(define left  (Tree 4  'none 'none))

(define my-tree (Tree 10 left right))
```

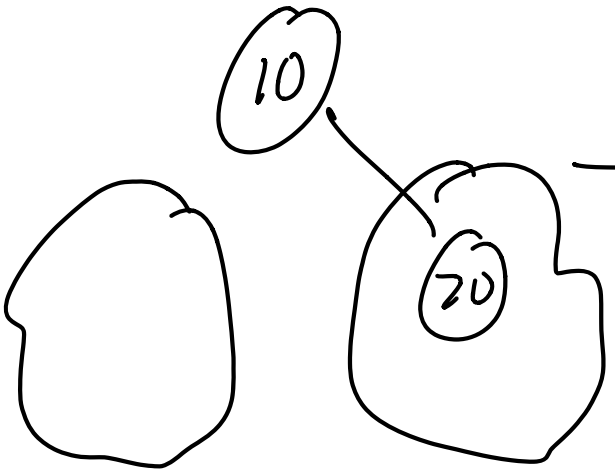
# Polymorphic Trees (Trees of arbitrary data type)

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

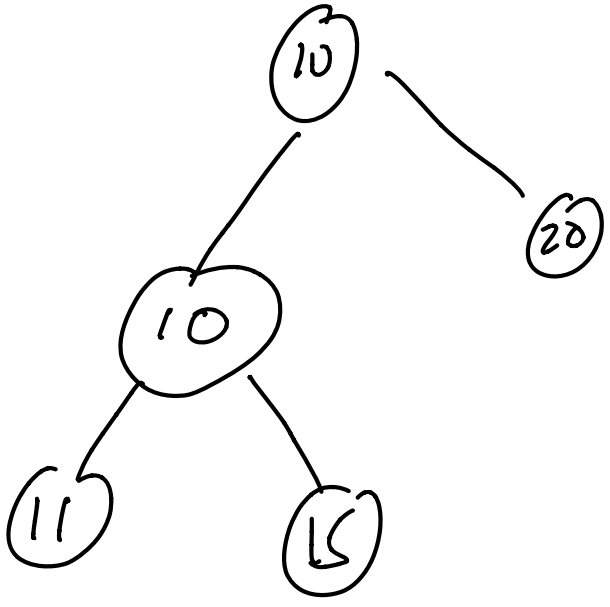
A is the data type
It can be anything.

Remark: Any function related to tree can solved recursively by solving it recursively for the left subtree and right subtree and then combining both the results to get your answer.
Classic divide and conquer approach.

Example: Write a function that finds the number of nodes in a Tree.




— Number of Nodes is 2




— Number of Nodes is 5.

Recursion

T is a tree

num_nodes(T) = num_nods (L) + num_nodes (R) + 1

L - left subtree of the T
R - Right subtree of T.

Base case.

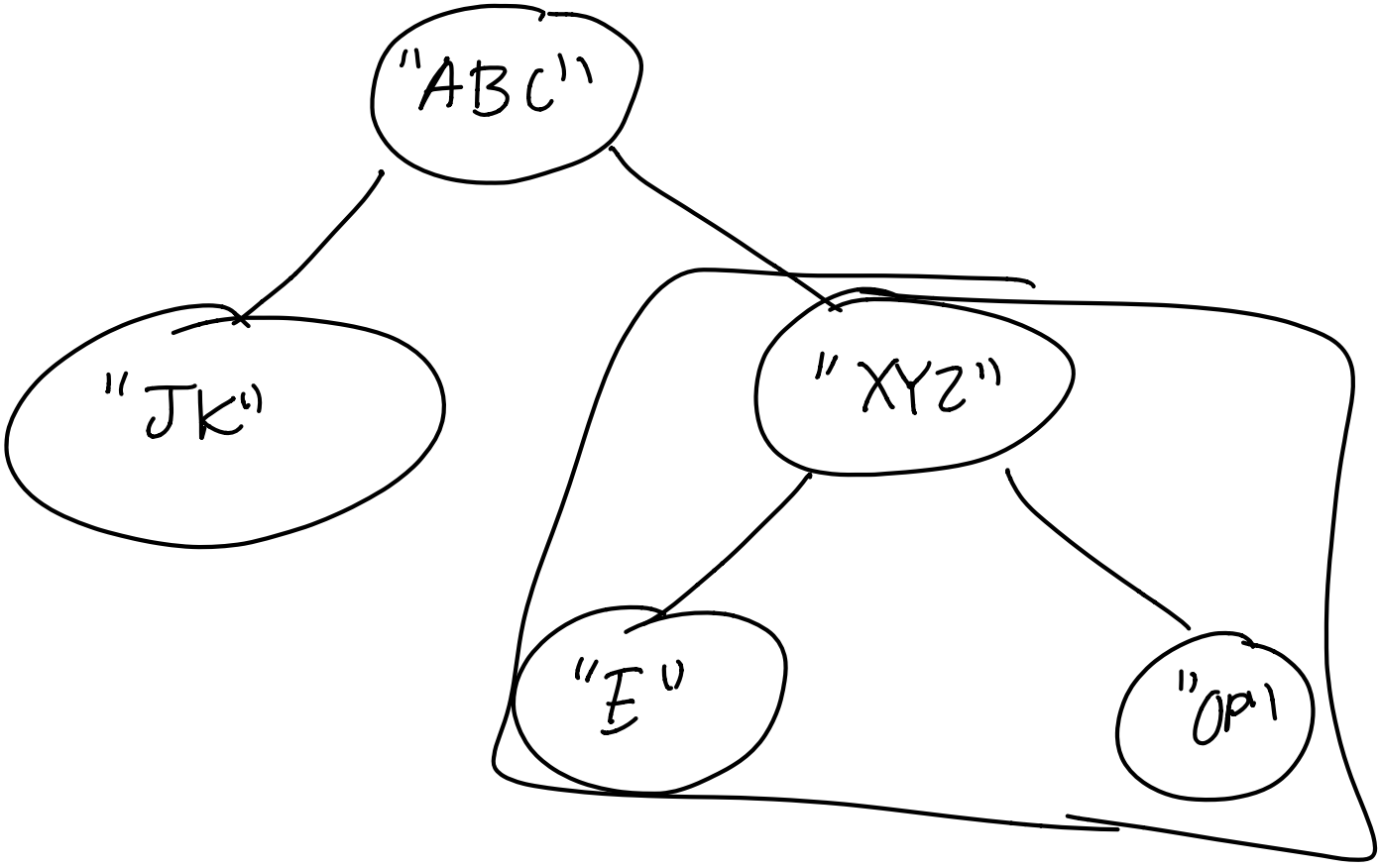
1. Tree has only one node - output 1

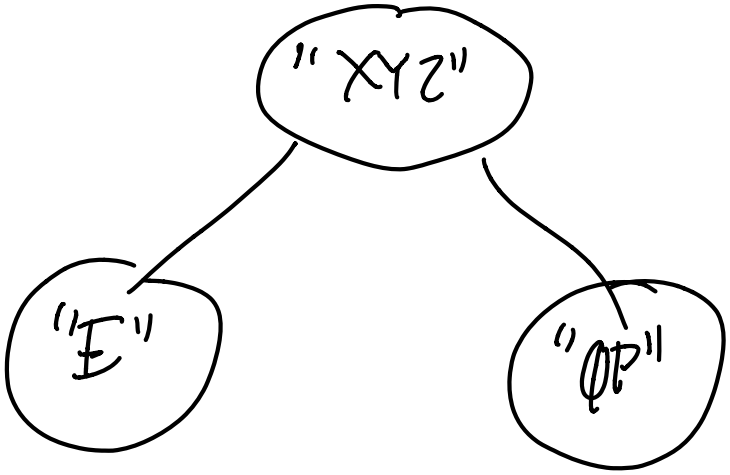
or

2. "Tree" is 'none output 0.

```
(: num_nodes : (All (A) (U (Tree A) 'none)
                    -> Integer))
(define (num_nodes t)
  (match t
    ['none 0]
    [(Tree v l r) (+ 1 (num_nodes l)
                       (num_nodes r))]))
```

Example: Write a function which takes a function and a tree as input and outputs the tree whose sotres the function.



```
(: search-xyz? : String → Boolean)
(define (search-xyz? name)
  (string=? name "xyz"))
```

```
(: search : (All (A) (U (Tree A) 'none)
                (A → Boolean) →
                (U 'none (Tree A))))
```

```
(define (search tree req)
  (match tree
    ['none 'none]
    [(Tree v l r) (cond
                    [(req v) tree]
                    [(Tree? (search l req)) (search
                                              l req)]
```

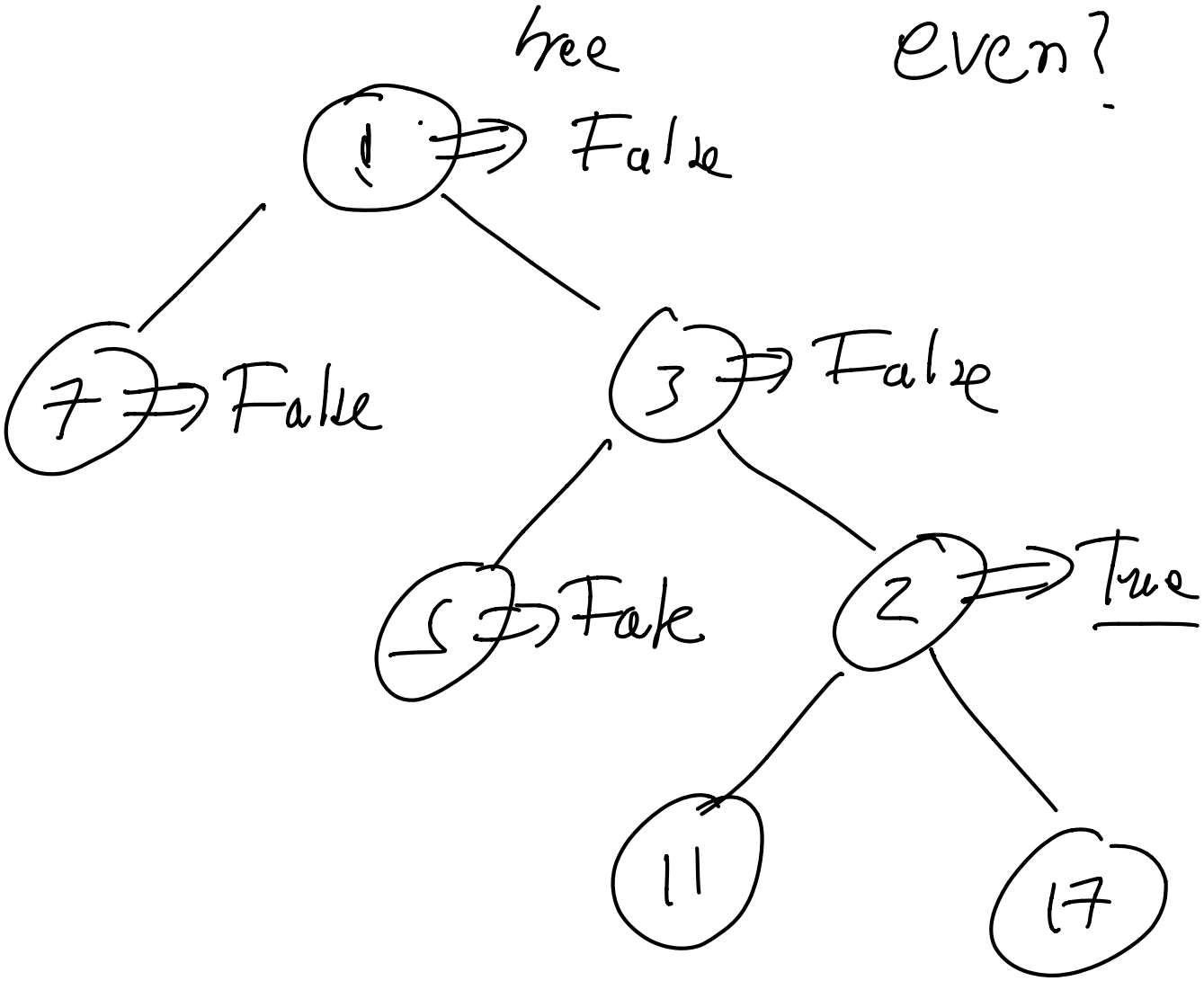
[else (search r req)]))

First check if root of the tree satisfies the condition
else check for the left subtree and check for the right subtree.

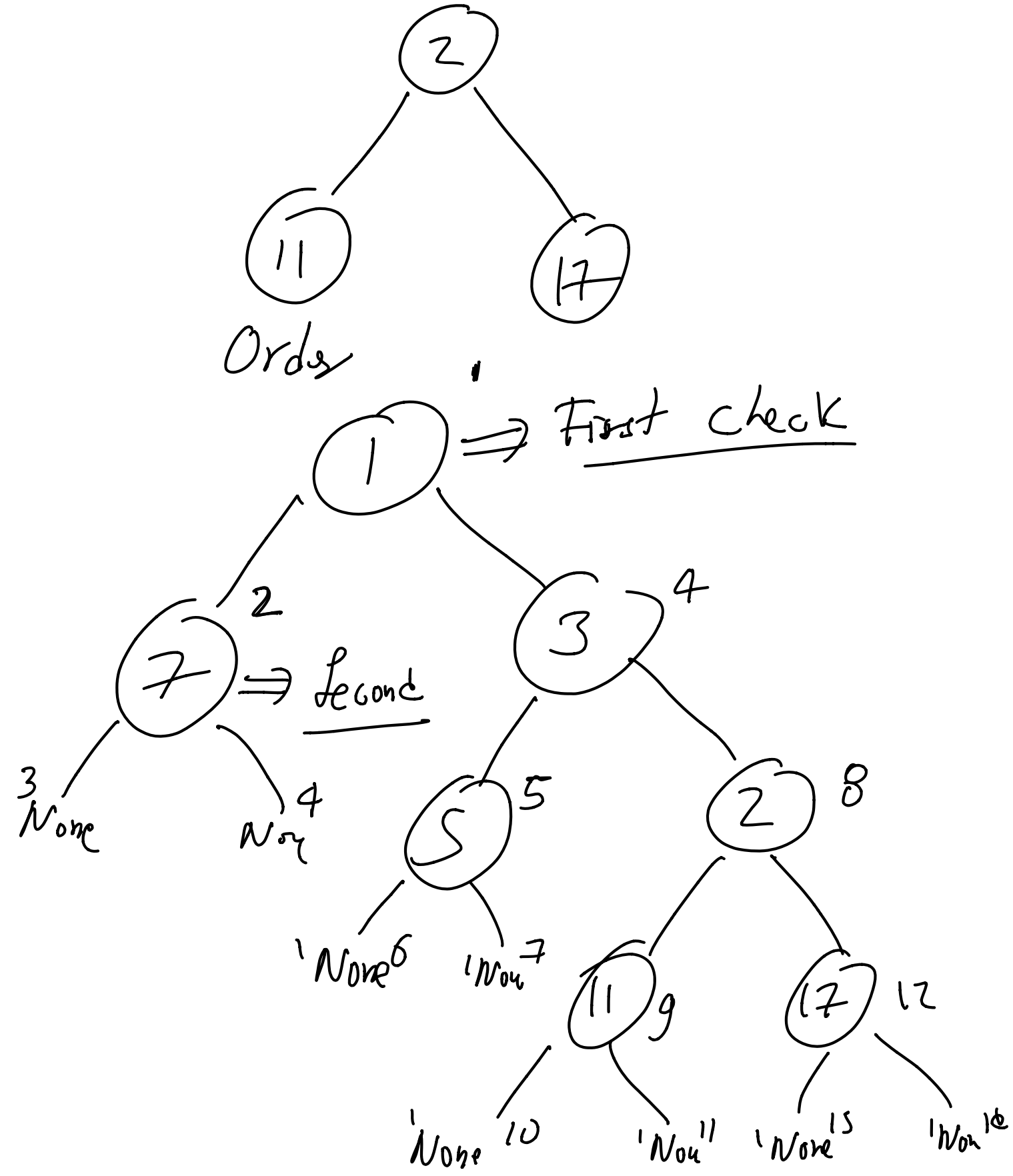
Require requires takes a value and outputs true or false.

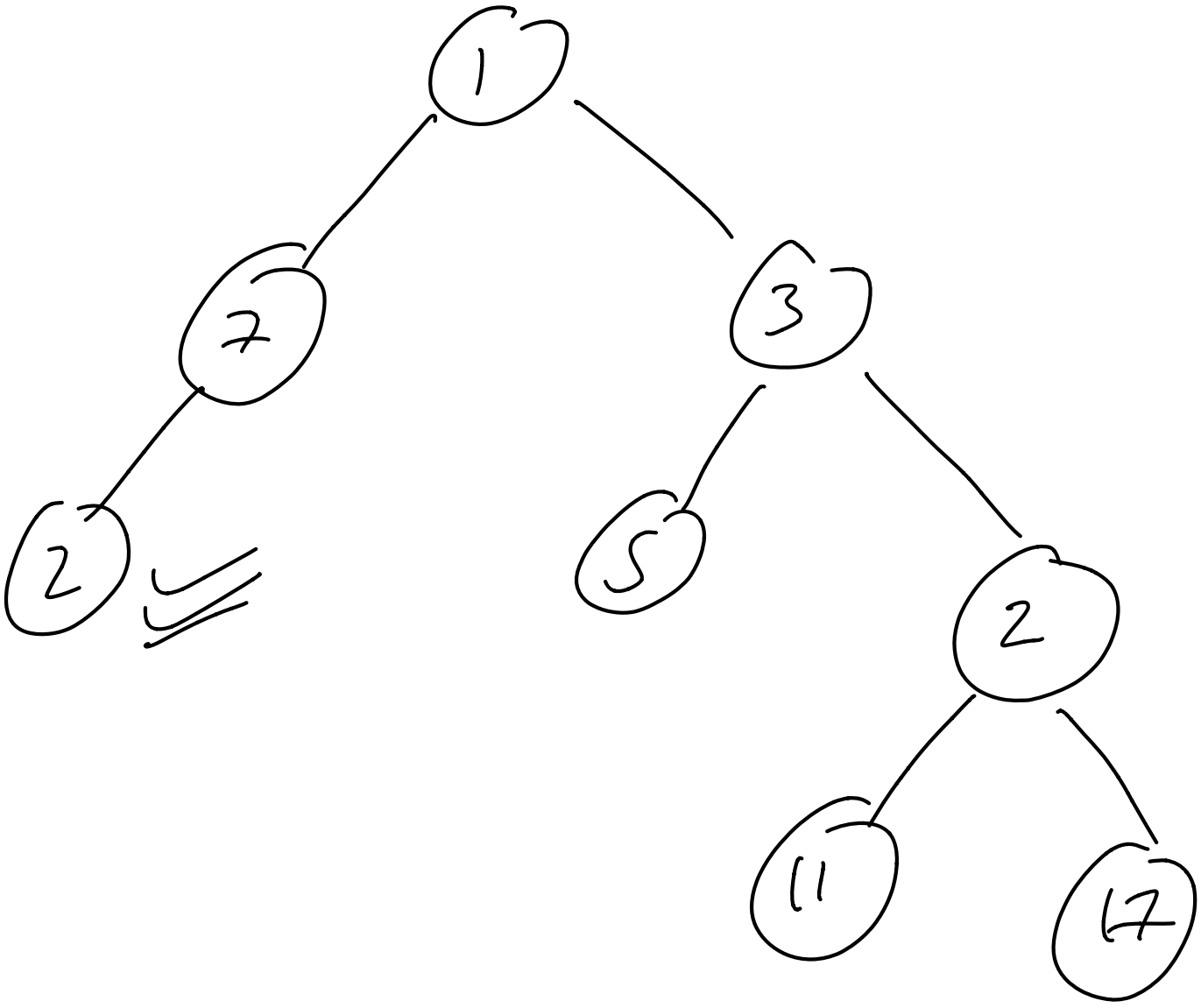
Require is check if node (val of the node) satisfies a certain condition

```
(: even? : Integer → Boolean)
(define (even? num)
  (= (remainder num 2) 0))
```

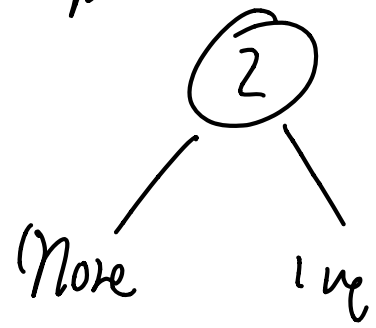



(search tree even?)


2
11
17
Ords
1
⇒ First check
7
2
⇒ Second
3
4
3
None
4
None
5
5
2
8
None 6
None 7
11
9
17
12
None 10
None 11
None 15
None 16




Output




If you want to use
search-xyz?

(search tree search-xyz?)

Implementation of list in racket.
List is a recursive data structure.

- Cons.
Data type called pair in racket.
(42 43)
("see" "saw")
('this 'that)

(cons 42 43) ⇒ (42 43)

A list in a racket is a recursive data structure using cons.

An empty list

null

Dr Racket (Welcome to Dr Racket)

> null

(List of Any)

'()

(cons 1 null)

> (cons 1 null)

'(1)

(cons 4 (cons 1 null))

→ [4 1]

(cons 7 (cons 4 (cons 1 null)))
→ [7 4 1] ✓

List of anny data type A
is pair (x y)
x: data of type A
y: List or null

null represents the empty.

Remark:
(cons 4 (cons 1 null))
≠ (cons 4 1)

The second component of the
first (cons 1 null) - (1 null)

The second component of the second 1

$1 \equiv (1 \ null)$

↑ Integer ↑ pair

Example:
Write a function which checks if two lists are equal.

Idea: Use recursion
check if the first elements of the two lists are same and then do recursion on (rest list)

```
(: list=? : (All (A) (A A → Boolean)
   (Listof A) (Listof A)
   → Boolean)

(define (list=? eq list-one list-two)
  (cond
   [ (and (empty? list-one)
          (empty? list-two)) #t ]

[ (or (empty? list-one) (emty?
       list-two)) #f ]
[ else (and (eq (first list-one)
       (first list-two)) (list=?
    eq (rest list-one) (rest list-two)))]
```

(list? string? (list "AB" "XY" "MM") (list "A" "B" "C"))

Example: Implement the map function.

(list 1 2 3 4 5)

⇓

(list 2 3 4 5 6)

(list A1 A2 A3 A4 A5)

f ⇒

(list f(A1) f(A2) f(A3) f(A4) f(A5))

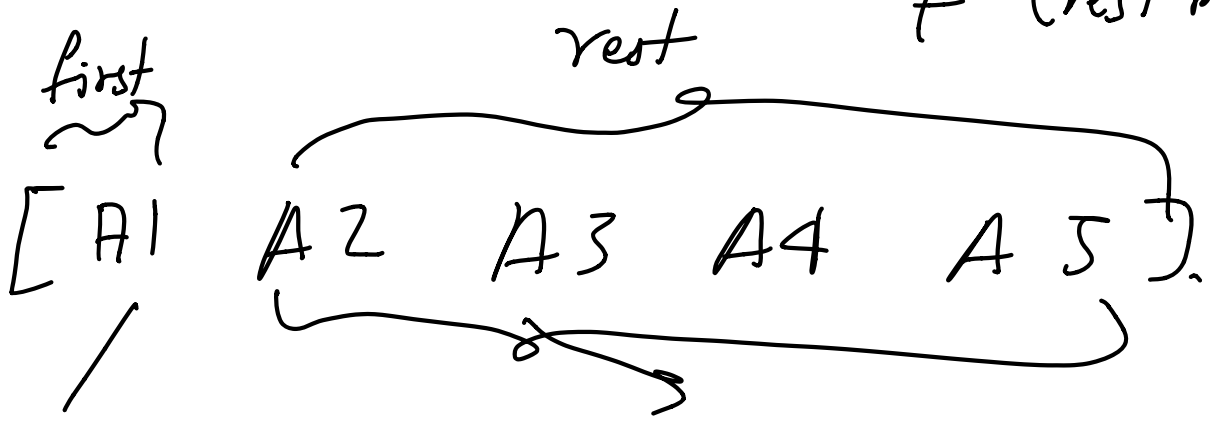
```
(: map : (All (A B) (A -> B)
   (Listof A) -> (Listof B))
(define (map f my-list)
  ( cond
  [(empty? my-list) null]
  [else (cons (f (first my-list)
     (map f (rest my-list))))]))
```

Idea: Use Recursion.

Apply f to the first element

```
(f (first my-list)) (map
        f (rest my-list))
```

(cons (f (first my-list)) (map f (rest my-list)))




(f A1) (map f rest my-list))

f(A1) [f(A2) f(A3) f(A4) f(A5)]

add to the front

(cons f(A1) [f(A2) f(A3) f(A4) f(A5)])

⇒ [f(A1) f(A2) f(A3) f(A4) f(A5)]

# Example Midterm question

Searching an integer in an Int-Set

Maybe useful

Write a function which takes a number outputs true if it is a prime number else it is false.

Input : num.

num % 2 == 0 ⇒ false

num % 2 ≠ 0

num % 3 =

num % 4

7

7 %. 2 =0 ful

7 %. 3 =0 = fa

7 %. 4 =0 = fa

7 %. 5 =0 = fals

7 %. 6 =0 = fal

7 %. 7 =0 ⟹ true

Two inputs number: d.

Find a divisor of number which is greater than or equt to d.

```
(: find-divisor : Natural Natural
                  → Natural)
(define (find-divisor number d )
(cond
[(= (remainder number d) 0) d]
[else (find-divisor number (+ 1 d))])
```

(find-divisor 5 2) ⇒ 5.

↖ remainder

5 % 2 ≟ 0

5 % 3 ≟ 0

5 % 4 ≟ 0

5 % 5 = 0

(: prime? : Natural → Boolean)

(define (prime? n)

(= (find-divisor n 2) n))

(find-divisor 9 2) ⇒ 3 ≠ 9

9 % 2 ≟ 0 ×

9 % 3 ≟ 0 ✓ ⇒ 3

(find-divisor 4 3) ⇒ 4

Check for all numbers between 2 and n

2 ——— n.