QUICK CONTENT REVIEW

- Lists
  - list elements can be accessed or changed by indexing into the list using '[ ]'
    - ↳ remember we have 0-indexing!!

ie.) >> list = ['hi', 2, 'bye']

```
>> list[0]
'hi'
>> list[1]
2
>> len(list)
3
```

← remember for list we can use len(listname) to figure out the length of any list

- list slicing → kind of like indexing but getting more than one element at a time

listname[startIndex (inclusive) : stopIndex (exclusive) : stepsize]

ie.) >> list = [1, 3, 4, 2, 7]

```
>> list[1:4:2]
[3, 2]
```

← start at index 1, increment by 2 until you reach index 4 (remember we don't include index 4)

- list comprehension → quick/compact way to create a list

[<whatyouwantinlist> for <variable> in <someiteration> if <cond>]

this part is optional

ie.) [(i+1) for i in range(5) if (i%2) == 0] → [1 3 5]

i = 0, 1, 2, 3, 4

↳ will create a list of all #s from 0 → 5 that are even, incremented by 1

```
lst = []
for i in range(5):
    lst.append(i+1)
```

- List Mutation
  - ↳ can change the contents of an existing list (since lists are mutable)
  - → all mutation operations except 'pop' return 'none' ('pop' returns the popped off element)

methods

Let lst = [ ] be the list we want to modify / mutate.

① append(el) : adds an element 'el' to the end of the list whom 'append' is applied to

ex.)
```
lst.append(1)
lst
>> [1]
lst.append([1,2,3])
lst
>> [1, [1,2,3]]
```

the entire thing passed into 'append' is added as a single item to the end of the list

② extend(lst2) : extends the list by concatenating it with 'lst2' ('lst2' must be an iterable — must be some sort of sequence)

ex.)
```
lst.extend(1)
```
→ will not work / will cause an error
```
lst.append(1)
lst.extend([1,2,3])
lst
>> [1,1,2,3]
```

all items in sequence passed into 'extend' are added as their own items to the list one by one

③ insert(i, el): insert element 'el' at index i (doesn't replace previous value)

```
ex.) lst.append(1)
     lst.extend([1,2,3])
     lst
     >> [1,1,2,3]
     lst.insert(2,0)
     lst
     >> [2,1,1,2,3]
```

→ just added a new element at index 0 and moved the rest of the elements down

④ remove(el): removes the first occurence of 'el' in the list

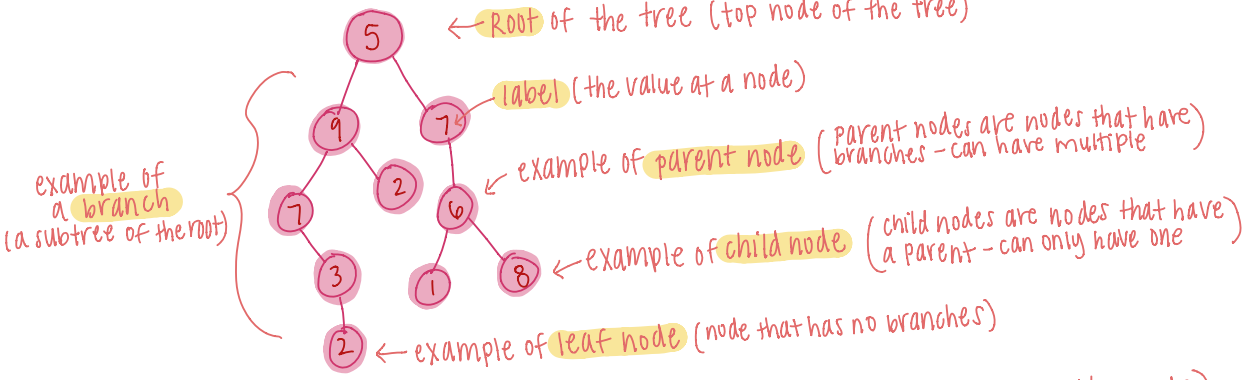
```
ex.) lst.append(1)
     lst.extend([1,2,3])
     lst
     >> [1,1,2,3]
     lst.remove(1)
     lst
     >> [1,2,3]
```

→ just removed the first occurence of '1' from the list

⑤ pop(i): removes & returns the element at index i

```
ex.) lst.append(1)
     lst.extend([1,2,3])
     lst
     >> [1,1,2,3]
     a = lst.pop(0)
     a
     >> 1
     lst
     >> [1,2,3]
```

- Trees (conceptually)
  - recursive data structure
  - root of the tree starts at the top and the leaves are at the bottom



← Root of the tree (top node of the tree)

label (the value at a node)

example of parent node (parent nodes are nodes that have branches - can have multiple)

example of child node (child nodes are nodes that have a parent - can only have one)

← example of leaf node (node that has no branches)

* depth of a node (how far away a node is from the root — # of edges between root & node)
* height of a tree (depth of the lowest leaf)

- Trees (implementation)
  - tree has value for root node and a sequence of branches
    - represent branches as lists of trees
  - constructor for 'tree' data type
    - t = tree(root_label, list_of_branches) ← if no 2nd parameter is given, the default value [] is used
    - to access label & branches

label(t) → will return root_label
branches(t) → will return list_of_branches (list of trees)

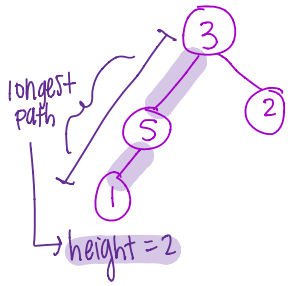
Warmup: Height (Q6)

Write a function that returns the height of a tree.

↳ length of longest path from root to leaf

```
def height(t):
    // base case? simplest case


    // recursive call(s)
```



Solution:

```
def height(t):
    if is_leaf(t):
        return 0
    else:
        all_branch_heights = [height(branch) for branch in branches(t)]
        longest_branch_height = max(all_branch_heights)
        return 1 + longest_branch_height
```

Max Product (Q2)

Q2: (Tutorial) Max Product

Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product that can be formed using non-consecutive
    elements of s.

    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
    "*** YOUR CODE HERE ***"
```

if s==[]:
    return 1

a → s[0] * max_p(s[2:])
b → max_p(s[1:])
return max(a,b)

→ Base case?
→ What choice do we have to make at every element of list?
• limitation, if we use s[i] we cannot use s[i+1] or s[i-1]

Solution:

```
def max_product(s):
    if s==[]:
        return 1
    elif len(s)==1:
        return s[0]
    else:
        dont_include_first = max_product(s[1:])
        include_first = s[0] * max_product(s[2:])
        largest_product = max(dont_include_first, include_first)
        return largest_product
```

Add this many (Q5)

modification we must make to list s

input

2.4 **Tutorial:** Write a function that takes in a value `x`, a value `el`, and a list `s` and adds as many `el`'s to the end of the list as there are `x`'s. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, s):
    """ Adds el to the end of s the number of times x occurs
    in s.
    >>> s = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, s)
    >>> s
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, s)
    >>> s
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

4:34

I will share a link in the chat. Please open it & we can collectively write our solution together. If you think you have a solution or partial solution, feel free to type it up & we will collectively go through all the solutions (more than one right answer! :)

SOLUTION:

```
def add_this_many(x, el, s):
    num_appearances = 0
    for elem in s:
        if elem == x:
            num_appearances += 1
    while num_appearances > 0:
        s.append(el)
        num_appearances -= 1
```

First iterate through all elements of s & count # of times x appears

add 'el' to end of s the # of times x occurs (num_appearances)

Key to remember: you can't append to a list while iterating through it!

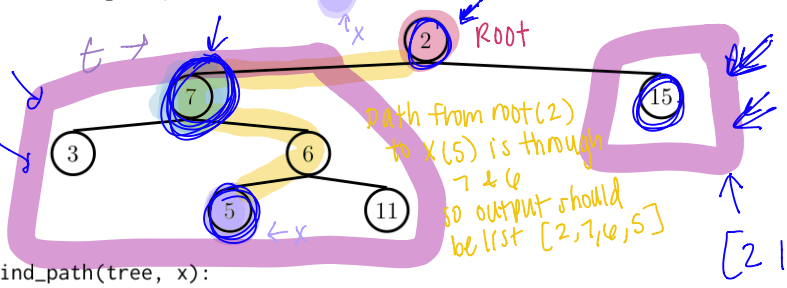
can avoid having to do this 2-part sol if we iterate through indices instead of elements (or using 'list') ↳ can just append directly in first 'for' loop

Find Path (Q8)

1.4 **Tutorial:** Write a function that takes in a tree and a value `x` and returns a list containing the nodes along the path required to get from the root of the tree to a node containing `x`.

If `x` is not present in the tree, return `None`. Assume that the entries of the tree are unique.

For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`




```
def find_path(tree, x):
    """
    >>> t = tree(2, [tree(7, [tree(3), tree(6, [tree(5), tree(11)])] ), tree(15)])
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """
    if _____:
        return _____
    for _____:
        path = _____
        if _____:
            return _____
    return _____
```

```
def find_path(tree, x):
    if label(tree) == x:
        return [label(tree)]
    for b in branches(t):
        path = find_path(b, x)
        if path:
            return [label(tree)] + path
    return None
```

COLOR CODED EXAMPLE/EXPLANATION: