# Welcome to CS61A Disc 5 Sect. 29/47 :D

Dickson Tsai
dickson.tsai+cs61a@berkeley.edu
OH: Tu, Th 4-5pm 411 Soda

Previous: Data Abstraction **>>>**

**Today: Lists and Trees >>>**

Next stop: Object oriented programming

## Section Quiz Logistics

- Please take out a piece of paper
- Put your name, login (e.g. cs61a-ty), SID, and section number (#47 for 11-12:30 pm, #29 for 6:30-8pm)
- Graded on effort: effort = revise your quiz with diff color

## Section Quiz

1. How can I make your lab/discussion experience more worthwhile?
2. Review: What is data abstraction?
3. From lab: What would Python print?

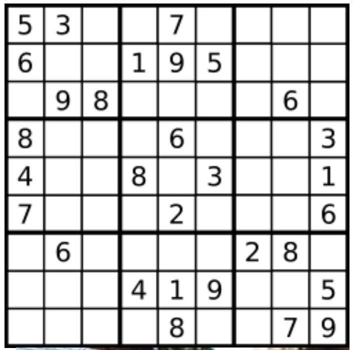
```
>>> a = [1, 2, 3, 4]
>>> a[2:]
____
>>> a[-2:]
_____
>>> a[-2::-1]
_____
>>> a + 3
_____
```

```
>>> a + [3]
_____
>>> a
_____
```

## Section Quiz [Solutions]

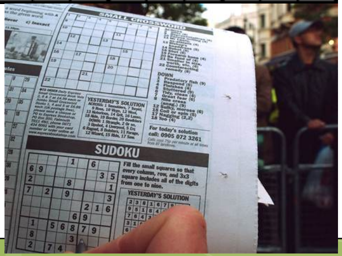
1. What is data abstraction? Think of a real life example.
   1. A methodology by which functions enforce an abstraction barrier between representation and use

The grid is represented by pixels on a screen



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

The grid is represented by a newspaper



In both grids:
- lookup(board,position)
- fill_in(board,position)
- take_note(board, position, possibility)

- is_valid(solution)

(Sudoku has the same rules no matter the form!)

## Section Quiz [Solutions]

1. From lab: What would Python print?

```
>>> a = [1, 2, 3, 4]
>>> a[2:]
[3, 4]
>>> a[-2:]
[3, 4]
>>> a[-2::-1]
[3, 2, 1]
>>> a + 3
TypeError: can only concatenate list (not “int”) to list
(Bonus): >>> 3 + a
TypeError: unsupported operand type(s) for ‘+’: ‘int’ and ‘list’
```

```
>>> a + [3]
[1, 2, 3, 4, 3]
>>> a
[1, 2, 3, 4]
(Bonus): >>> a[5:]
[]
```

Explanations:

- Slicing and concatenation are non-mutating operations. The original list remains unchanged. You get a new object every time!
- If slicing is invalid, you will get an empty list

## Anuncios

- HW4 due next Tuesday 10/7.
- Project 2 Trends due next Wed 10/8 for one extra point, or 10/9
- Reminder: My OH are Tuesdays, Thursdays 4-5 PM in Soda 411
- Do not post any of your code on a public-viewable platform online. (e.g. Github, Pastebin)
- Go to dicksontsai.com/meet to schedule an appointment if you would like to talk to me about anything related to the course.

## 5-min Review Recap

- Pingpong (should require) translating an iterative solution to a recursive solution.

```
def fib_iter(n):
  curr, prev = 0, 1
  while n > 0:
    curr, prev = curr + prev, curr
    n -= 1
  return curr

def fib_trans(n):
  def helper(n, curr, prev):
    if n == 0:
      return curr
    return helper(n-1, curr+prev, curr)
  return helper(n, 0, 1)
```

1. Variables to track
2. Initial values
3. When to stop (note – they are opposite)
4. Update values. In recursion, assignment happens by argument passing
5. Loop
6. Return

# Tip of the Day

- Appreciate the power of the tools you have!(Think of more reasons!)
  - Why are while loops powerful?
    - You just have to state how a general iteration should go
    - You can control when the loop ends generally (with variables)
  - Why are functions powerful?
    - State a piece of computation generally. Manipulate the computation via the parameters of the function.
    - Store the code until the code needs to be called up.
  - Why is recursion powerful?
    - Instead of coming up with whole solution at once, build up from a smaller solution. Lets you think more about concepts than code
  - Why is data abstraction powerful?
    - Custom types = more flexibility, precision in manipulating data
    - Specialized tweet functions can take in tweet objects, know what the objects store

# New Way of Running Disc

- Form groups of 2-3 as usual
- Write up your solution (function boxes + answers) on the board!
  - If you are concerned about putting about an incorrect answer, you shouldn't be. It's better to make mistakes here than on the test.
    - Plus, everyone can see common mistakes and learn together.
  - More practice with writing code
  - Chalkboards should make you feel smart.
  - Rotate people every problem so everyone gets a chance.

## List Comprehension

```
[<map_expr> for <name> in <iter_exp> if <cond_expr>]
```

A list comprehension is an expression that evaluates to a list.

To create the list:

1. Evaluate the iter exp. <name> will take on each value in the resulting sequence
2. Evaluate the cond exp. If false, move on to the next value.
3. Evaluate the map exp under the context of the name. The value of the map exp will be an element of the new list.

## List Comprehension Examples

1. Iter exp 2. cond exp 3. map exp

A list of 50 1's:

```
[1 for i in range(50)]
```

A list of odd squares from a to b:

```
[x * x for x in range(a, b) if x % 2 == 1]
```

The list of fibonacci numbers for odd squares from a to b if odd square ends with a 7

```
[fib(x) for x in [x*x for x in range(a, b) \
       if x % 2 == 1] if x % 10 == 7]
```

## List Comprehension Examples

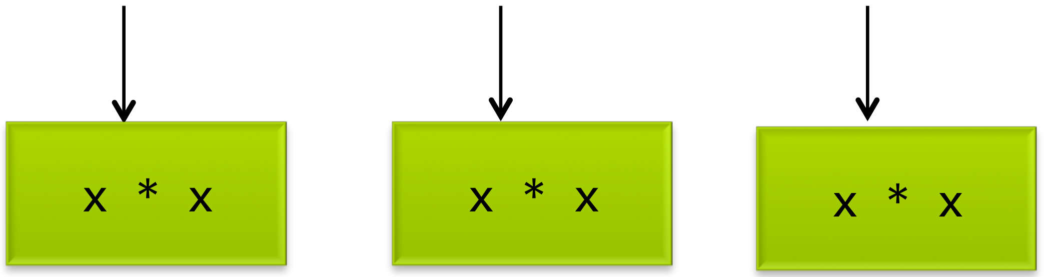
1. Iter exp 2. cond exp 3. map exp

A list of odd squares from 1 to 6:

```
[x * x for x in range(1, 9) if x % 2 == 1]
```

| | | | | | | |
|---|---|---|---|---|---|---|
| Iter exp seq: | 1 | 2 | 3 | 4 | 5 | 6 |
| Cond exp eval: | 1 | **2** | 3 | **4** | 5 | **6** |

Map exp:



```
[1, 9, 25]
```

## Why use list comprehensions?

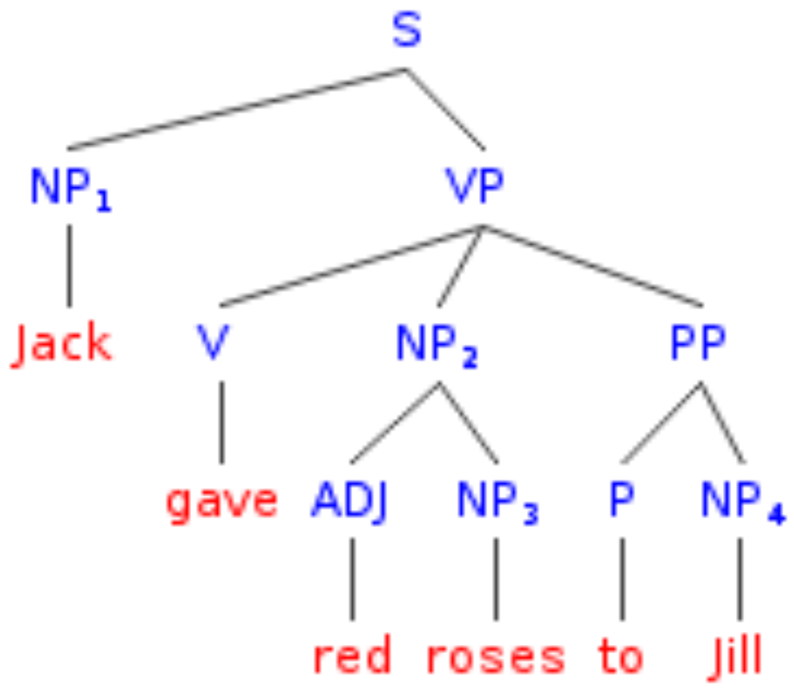
- “Concise way to create lists” – Python documentation
- Writing out a whole list is tedious and high-maintenance
  - `[tweet_text(tweets[0]), tweet_text(tweets[1])]`
- Q: How can one piece of code generate all elements
  - Q: What does each element share? -> map expression
  - Q: What varies from element to element? -> iter expression
    - Have another list that will determine the differences. You can filter that list -> cond expression

## Why use list comprehensions?

- Q: How can one piece of code generate all elements
  - Q: What does each element share? -> map expression
  - Q: What varies from element to element? -> iter expression
    - Have another list that will determine the differences. You can filter that list -> cond expression
- Example: collecting text of tweets
  - Part that's the <u>same</u>: tweet_text(…)
  - Part that's <u>different</u>: tweets[0], tweets[1]
    - The differences are [tweets[0], tweets[1], …,] -> tweets!
  - Now, as long as we can <u>plug in difference </u>under the same name "tweet", we can write all elements using same code:

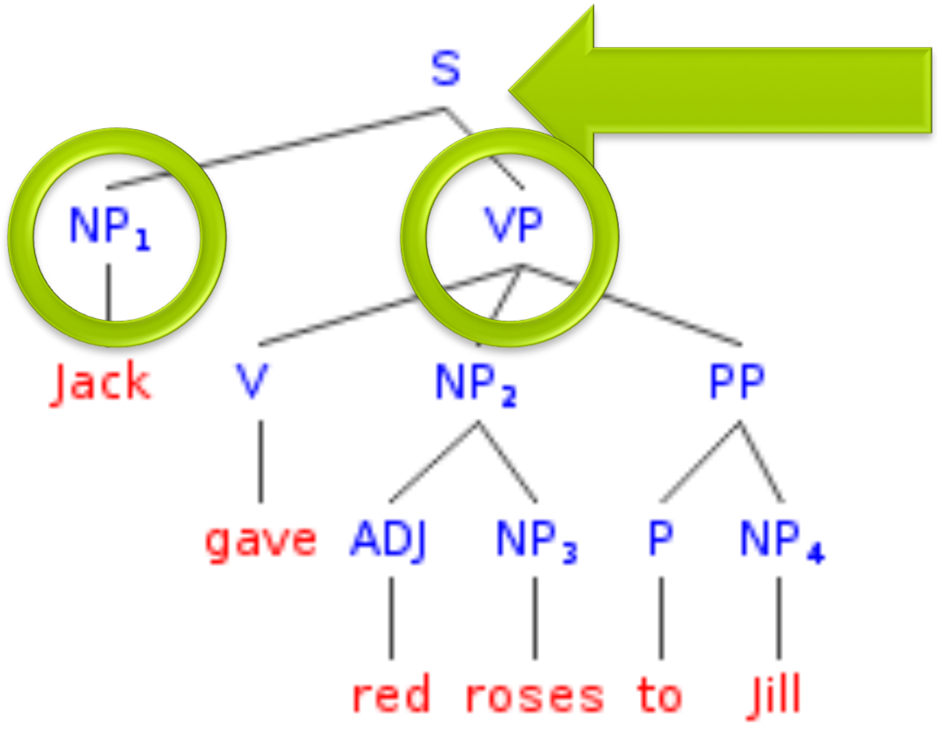
```
[tweet_text(tweet) for tweet in tweets]
```

## Tree Terminology



- Tree – a graph without cycles (there is only one path from one node to another) -> Hierarchy

## Tree Terminology – Parent/Child

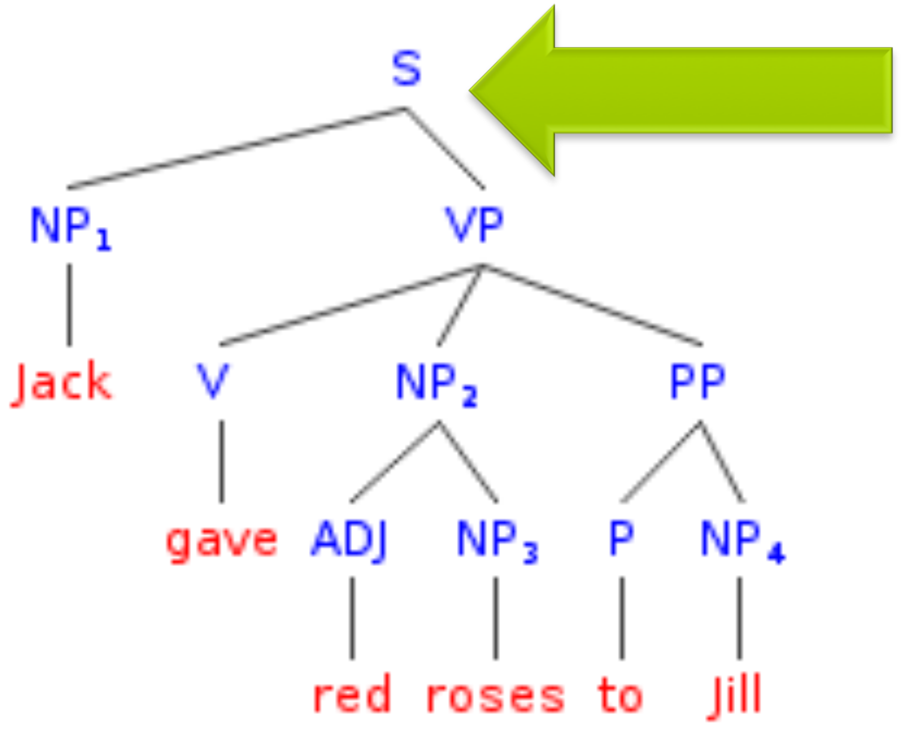


S is the parent of $NP_1$ and VP

$NP_1$ and VP are children of S

- Parent – a node that has children.
- Child – a node that has a parent

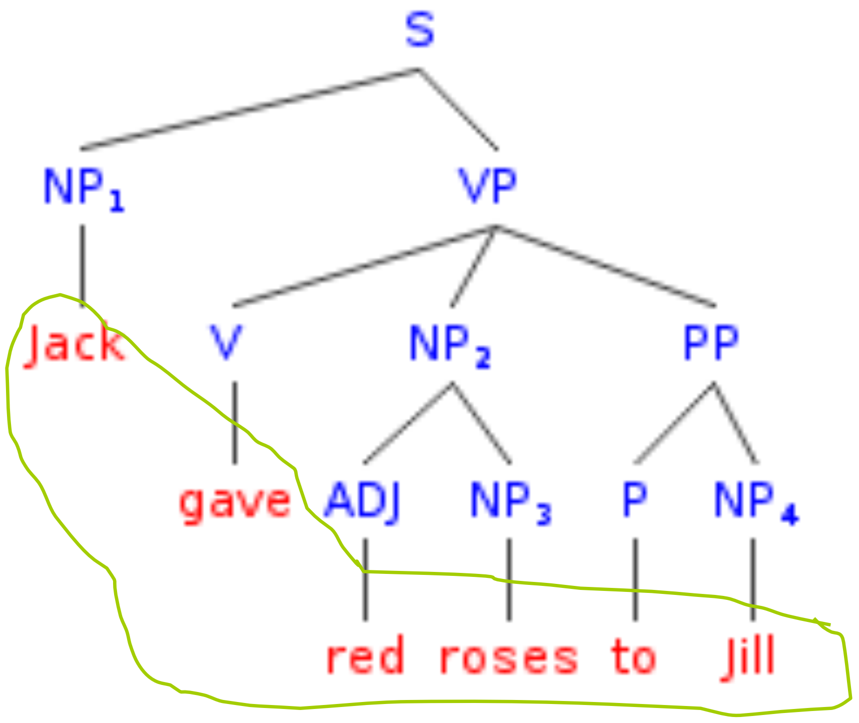
# Tree Terminology - Root



S is the root of this tree

- Root – a node with no parent
  - To refer to a tree, usually have a reference to the root

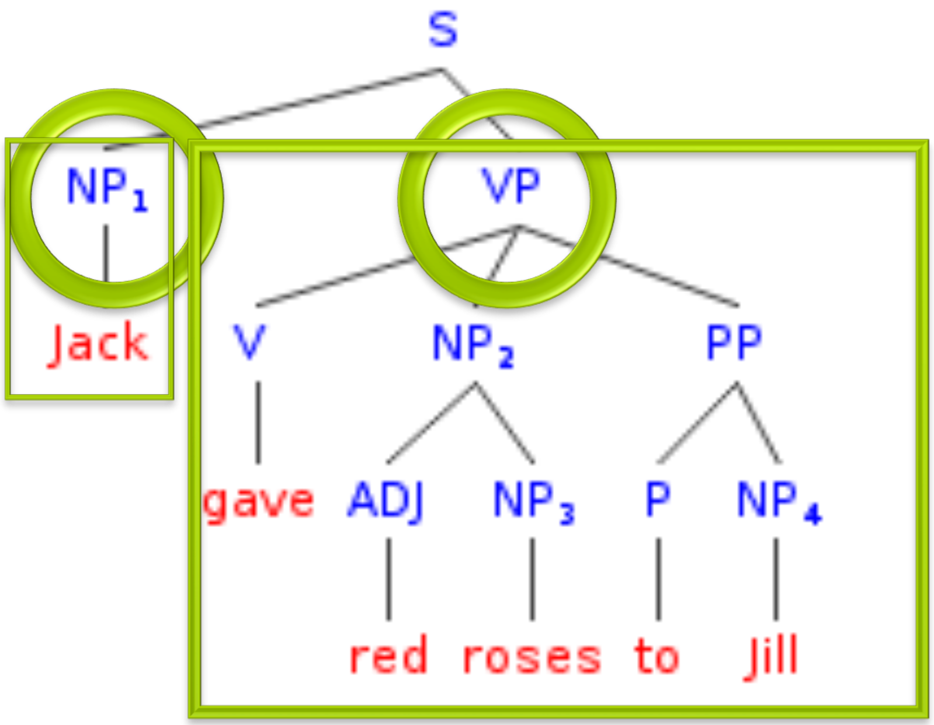
## Tree Terminology - Leaf



The words are leaves here

- Leaf – a node without any child
  - Most involved with base cases???

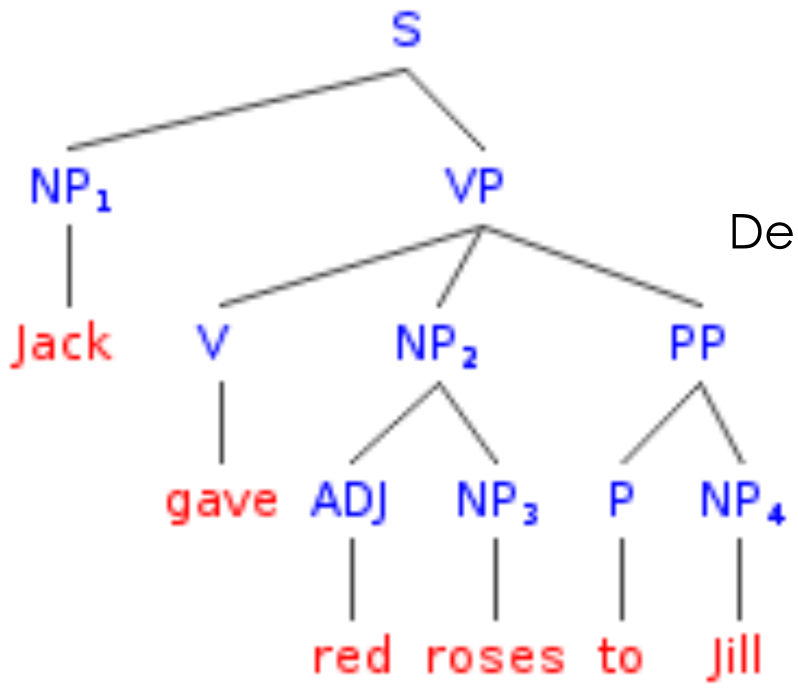
## Tree Terminology - Subtree



The tree corresponding to a child is a subtree.

- Subtree – the tree corresponding to a child. The child is the root of this tree

# Tree Terminology – Depth/height



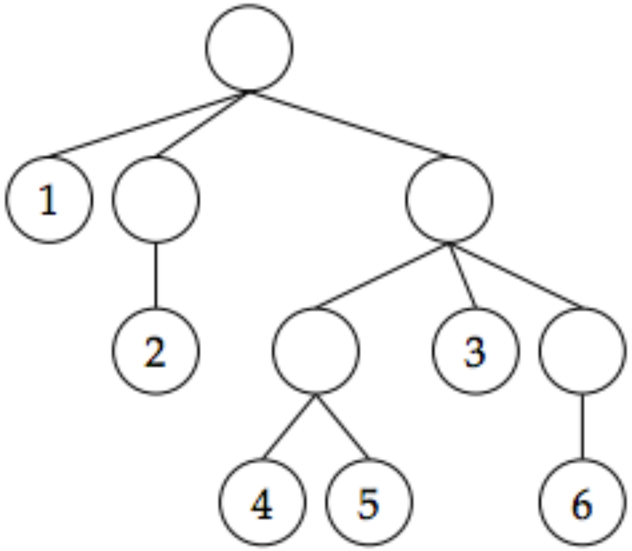
Depth of PP: 2

Height: 4

- Depth(node) – the distance from root to given node
- Height – the depth of the lowest leaf: min{depth(node)}

## Non-rooted Tree

```
tree = [1, [2], [[4, 5], 3, [6]]]
```

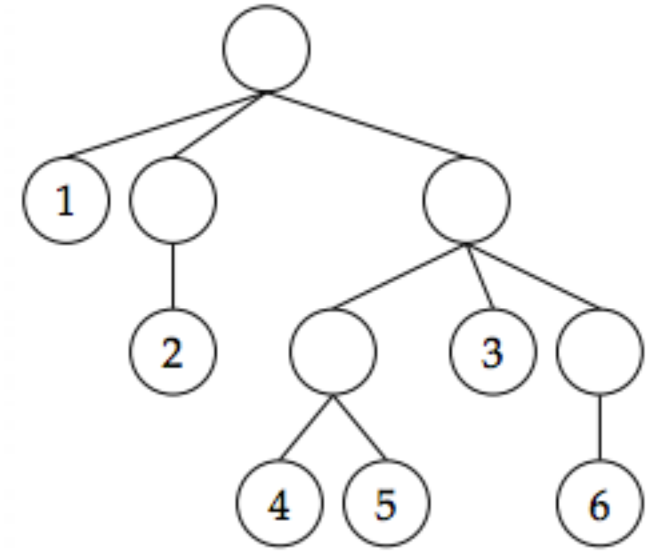


Q: What do you notice about this tree?

A: There are only values at the leaves!

# Non-rooted Tree ADT

```
tree = [1, [2], [[4, 5], 3, [6]]]
```



<u>Representation</u>

Leaves are **ints**

every other node is a **list**

---

<u>Use</u>

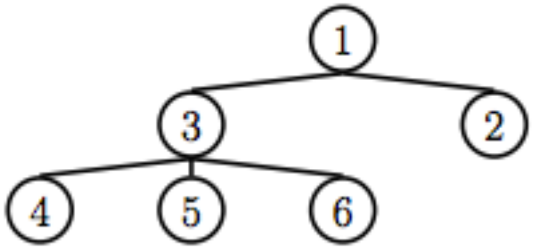
is_leaf(tree)

apply_to_leaves(fn, tree)

To get values, just return tree

## Rooted Tree

```
tree = rooted(1, [rooted(3, [leaf(4), leaf(5),
leaf(6)]), leaf(2)])
```



Q: What do you notice about this tree?

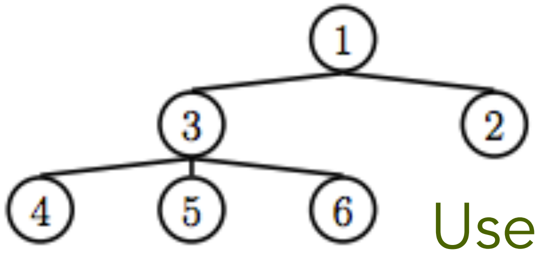
A: There are values at every node!

# Rooted Tree ADT

```
tree = rooted(1, [rooted(3, [leaf(4), leaf(5),
leaf(6)]), leaf(2)])
```

<u>Representation</u>

Each tree = value + list of children (list of rooted trees)



<u>Use - Constructors</u>

rooted(value, branches)

leaf(value) # makes branches an empty list

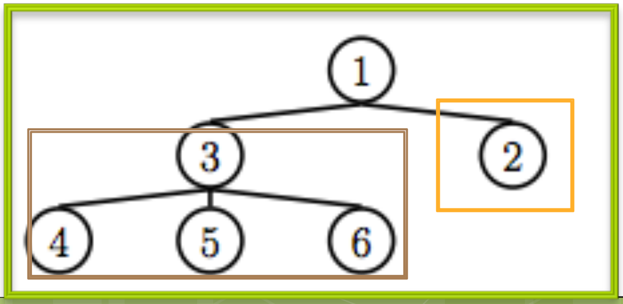
<u>Use – Selectors:</u> root(tree) branches(tree)

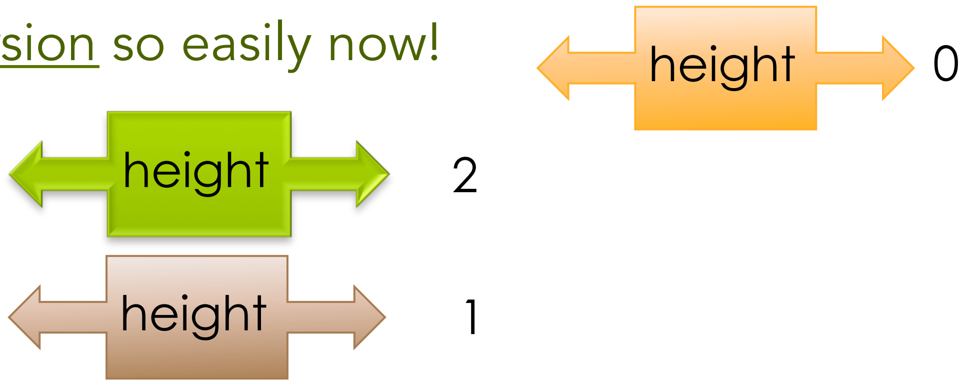
<u>Use – Functions:</u> is_rooted_leaf(tree)

Note: Selecting is more complicated now!

# Closure Property

- The result of combination can itself be combined using same method
  - For trees: To create a tree, combine value of list of trees. Now we can put this new tree into another list of trees and value to produce yet another tree!
  - i.e. Every subtree of a tree is a tree
    - -> You can do recursion so easily now!





# Optional Conceptual Questions (Wednesday 10/1 lecture)

1. What are the restrictions on dictionary keys?
   1. Note that strings can be keys. What does this say about string objects?
2. What are the restrictions on dictionary values?
   1. Can I have a dictionary whose value is another dictionary?
3. What are the behavior conditions of a linked list?
4. Draw the box-and-pointer diagram (if possible) for:
   1. link(1, 2, 3, 4, None)                3. link(1, link('hi', link(3, None)))
   2. link(1, link(link(2, link(3, None)), link(4, None)))
5. What functions do we have to add to our linked list to satisfy the sequence abstraction?
6. How are rooted trees different from non-rooted trees? Why might the difference be useful?
7. What're the constructors and selectors of the linked list? The rooted tree?