CS 61A DISCUSSION 3

TREES AND SEQUENCES

Raymond Chan
Discussion 134
UC Berkeley Fall 16

AGENDA

- Announcements
- Lists
- List Comprehension
- Trees
- Appendix
    - Data Abstraction

ANNOUNCEMENTS

- Midterm 1 Regrade Requests via Gradescope by Sunday.
- Homework 4 due tonight.
- Homework 5 due 9/27, next Tues.
- Lab 4 due Friday.

SEQUENCES

- Ordered collection of values
- Length
- Element Selection

LIST

- Sequence - order collection of values
- Python list is a type of sequence of whatever values we want.
    - numbers, strings, functions, lists
- Create a list using [] (square brackets).
    - [1, 2, 3, 4, 5]
- List content can contain different types.
    - [1, “two”, lambda : 3, 4, True]

LIST

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
  - First element is at index 0
  - *i*-th element is indexed at *i* -1
- Can have negative index
  - If a list has a length of *n*, we can index from -*n* to *n* -1.

LIST

- We can access, or index, any element with square brackets.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
```

- Lists are **zero-indexed**.
  - First element is at index 0
  - *i*-th element is indexed at *i* -1
- Can have negative index
  - If a list has a length of *n*, we can index from -*n* to *n* -1.

LIST

- We can access, or index, any element with square brackets.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
```

- Lists are **zero-indexed**.
  - First element is at index 0
  - *i*-th element is indexed at *i* -1
- Can have negative index
  - If a list has a length of *n*, we can index from -*n* to *n* -1.

LIST

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
  - First element is at index 0
  - *i*-th element is indexed at *i* -1

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
```

- Can have negative index
  - If a list has a length of *n*, we can index from -*n* to *n* -1.

LIST

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
    - First element is at index 0
    - *i*-th element is indexed at *i* -1
- Can have negative index
    - If a list has a length of *n*, we can index from -*n* to *n* -1.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
```

LIST

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
    - First element is at index 0
    - *i*-th element is indexed at *i* -1
- Can have negative index
    - If a list has a length of *n*, we can index from -*n* to *n* -1.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
```

LIST

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
  - First element is at index 0
  - *i*-th element is indexed at *i* -1

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
Index OutOfBounds Error
```

- Can have negative index
  - If a list has a length of *n*, we can index from -*n* to *n* -1.

LIST

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
  - First element is at index 0
  - *i*-th element is indexed at *i* -1
- Can have negative index
  - If a list has a length of *n*, we can index from -*n* to *n* -1.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
Index OutOfBounds Error
>>> L[-4]
```

LIST

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
    - First element is at index 0
    - *i*-th element is indexed at *i* -1
- Can have negative index
    - If a list has a length of *n*, we can index from -*n* to *n* -1.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
Index OutOfBounds Error
>>> L[-4]
2
```

LIST

- With multiple lists, we can concatenated them together using +

```
>>> odds = [1, 3, 5, 7]
>>> evens = [2, 4, 6]
>>> odds + evens
[1, 3, 5, 7, 2, 4, 6]
```

LIST

- To obtain the length of a sequence, use the **len** built-in function

```
>>> odds = [1, 3, 5, 7]
>>> len(odds)
4
>>> odds[len(odds) - 1]
7
```

LIST

- Check if an element exists in a list with **in**
- Cannot look into nested lists

```
>>> odds = [1, 3, 5, 7]
>>> 5 in odds
True
>>> 3 in odds
False
```

```
>>> lst = [1, [2, 3], 5, 7]
>>> 3 in lst
False
```

LIST

WHAT WOULD PYTHON PRINT?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])

>>> len(a)

>>> 2 in a

>>> 4 in a

>>> a[3][0]
```

LIST

WHAT WOULD PYTHON PRINT?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
1 3
>>> len(a)

>>> 2 in a

>>> 4 in a

>>> a[3][0]
```

LIST

WHAT WOULD PYTHON PRINT?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
1 3
>>> len(a)
5
>>> 2 in a

>>> 4 in a

>>> a[3][0]
```

LIST

WHAT WOULD PYTHON PRINT?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
1 3
>>> len(a)
5
>>> 2 in a
False
>>> 4 in a

>>> a[3][0]
```

Cannot look into nested list [2, 3]

LIST

WHAT WOULD PYTHON PRINT?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
1 3
>>> len(a)
5
>>> 2 in a
False
>>> 4 in a
True
>>> a[3][0]
```

LIST

WHAT WOULD PYTHON PRINT?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
1 3
>>> len(a)
5
>>> 2 in a
False
>>> 4 in a
True
>>> a[3][0]
2
```

a[3] returns the nested list

LIST SLICING

- We can get a certain part of a list via slicing
- list[<start>:<stop>:<step>]
- Our new list beings at ***start***, takes every ***step***-th element (or jump by ***step***), and ends at index before ***stop.***
- If it cannot reach ***stop***, it will return an empty list.
- By default step is 1
- Slicing will **always** create a **new list.**
- **step** can be positive (go right) or negative (go left).

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
[‘is’, ‘a’]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']    >>> lst[4:2]
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
[‘is’, ‘a’]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
[‘is’, ‘a’]
```

```
>>> lst[4:2]
[]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
[‘is’, ‘a’]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
[‘is’, ‘a’]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', ‘so']
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
[‘is’, ‘a’]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', ‘so']
>>> lst[2:7:-4]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
[‘is’, ‘a’]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', ‘so']
>>> lst[2:7:-4]
[]
>>> lst[:5]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
['is', 'a']
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', ‘so']
>>> lst[2:7:-4]
[]
>>> lst[:5]
[‘c','s','6','1','a']
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
['is', 'a']
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', ‘so']
>>> lst[2:7:-4]
[]
>>> lst[:5]
[‘c’,'s','6','1','a']
>>> lst[7:]
```

LIST SLICING

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
>>> lst[-3:-5:-1]
['is', 'a']
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', ‘so']
>>> lst[2:7:-4]
[]
>>> lst[:5]
[‘c’,'s','6','1','a']
>>> lst[7:]
['fun']
```

LIST SLICING

WHAT WOULD PYTHON PRINT?

```
>>> a =[3, 1, 4, 2, 5, 3]
>>> a[1::2]

>>> a[:]

>>> a[4:2]

>>> a[1:-2]

>>> a[::-1]
```

LIST SLICING

WHAT WOULD PYTHON PRINT?

```
>>> a =[3, 1, 4, 2, 5, 3]
>>> a[1::2]
[1, 2, 3]
>>> a[:]

>>> a[4:2]

>>> a[1:-2]

>>> a[::-1]
```

LIST SLICING

WHAT WOULD PYTHON PRINT?

```
>>> a =[3, 1, 4, 2, 5, 3]
>>> a[1::2]
[1, 2, 3]
>>> a[:]
[3, 1, 4, 2, 5, 3]
>>> a[4:2]

>>> a[1:-2]

>>> a[::-1]
```

Slicing always creates and returns a new list

LIST SLICING

WHAT WOULD PYTHON PRINT?

```
>>> a =[3, 1, 4, 2, 5, 3]
>>> a[1::2]
[1, 2, 3]
>>> a[:]
[3, 1, 4, 2, 5, 3]
>>> a[4:2]
[]
>>> a[1:-2]

>>> a[::-1]
```

Step by default is 1.
Cannot reach 2 from 4 when going right.

LIST SLICING

WHAT WOULD PYTHON PRINT?

```
>>> a =[3, 1, 4, 2, 5, 3]
>>> a[1::2]
[1, 2, 3]
>>> a[:]
[3, 1, 4, 2, 5, 3]
>>> a[4:2]
[]
>>> a[1:-2]
[1, 4, 2]
>>> a[::-1]
```

LIST SLICING

WHAT WOULD PYTHON PRINT?

```
>>> a =[3, 1, 4, 2, 5, 3]
>>> a[1::2]
[1, 2, 3]
>>> a[:]
[3, 1, 4, 2, 5, 3]
>>> a[4:2]
[]
>>> a[1:-2]
[1, 4, 2]
>>> a[::-1]
[3, 5, 2, 4, 1, 3]
```

Without passing **start** and **stop**, the defaults will change with **step**.

LIST SLICING

WHAT WOULD PYTHON PRINT?

- Default **start** and **step** changes with the sign of **s.**
- If **s** is a positive step, then lst[<start>:<step>:s] becomes lst[0:len(lst):s].
- lst[<start>:<step>:-s] becomes lst[len(lst)-1: -(len(lst)+1) : -s]
  - -(len(lst)+1) because we need to count backwards and cross the 0-th index.

```
>>> a[::-1]
[3, 5, 2, 4, 1, 3]
```

FOR LOOPS

- Another method of iteration
- for <variable> in <sequence>

```
>>> for i in range(0, 6):
…       print(i)
0
1
2
3
4
5
```

```
>>> lst [1, 2, 3]
>>> for x in lst:
…       print(x)
1
2
3
```

```
>>> lst [1, 2, 3]
>>> for i in range(len(lst)):
…       print(lst[i])
1
2
3
```

FOR LOOPS

- range(<start>, <stop>,<step>)
- Allows a for loop to iterate through a sequence from ***start*** up to and excluding ***stop****,* taking every ***step***-th element.
- Default **step** is 1.
- Default **start** is 0.
- Must have **stop.**

```
for i in range(0, 5, 2)
for i in range(2, 5)
for i range(5)
```

LIST COMPREHENSION

- Compact way to create a list
- [<map exp> for <name> in <iter exp> if <filter exp>]
- if clause is optional

```
nums = [1, 2, 3, 4, 5, 6, 7]
lst = []
for x in nums:
    if x % 2 == 0:
        lst += [x+3]

[x + 3 for x in nums if x % 2 == 0]
```

LIST COMPREHENSION

- Don’t use an else at the end.

- Move it to the expression.

- `x + 3 if not x % 2 else 100` is a **ternary expression**

```
>>> [x + 3 for x in nums if x % 2 == 0 else 100]
Error
>>> [x + 3 if x % 2 == 0 else 100 for x in nums]
[5, 7, 9]
```

LIST COMPREHENSION

SLICING

- Another way to see slicing

```
>>> nums = [1, 2, 3, 4, 5, 6, 7]
>>> [nums[i] for i in range(1, 5, 2)]
[2, 4]
>>> nums[1:5:2]
[2, 4]
```

LIST COMPREHENSION

WHAT WOULD PYTHON PRINT?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]

>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]

>>>[[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```

LIST COMPREHENSION

WHAT WOULD PYTHON PRINT?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
[3, 5]
>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]

>>>[[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```

LIST COMPREHENSION

WHAT WOULD PYTHON PRINT?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
[3, 5]
>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]
[20, 6, 6]
>>>[[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```

LIST COMPREHENSION

WHAT WOULD PYTHON PRINT?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
[3, 5]
>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]
[20, 6, 6]
>>>[[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
[[2, 4], [4, 6], [6, 8], [8, 10]]

lst = []
for x in [1, 2, 3, 4]:
    for y in [x, x + 1]:
        lst += [[y * 2]]
```

LIST COMPREHENSION

WHAT WOULD PYTHON PRINT?

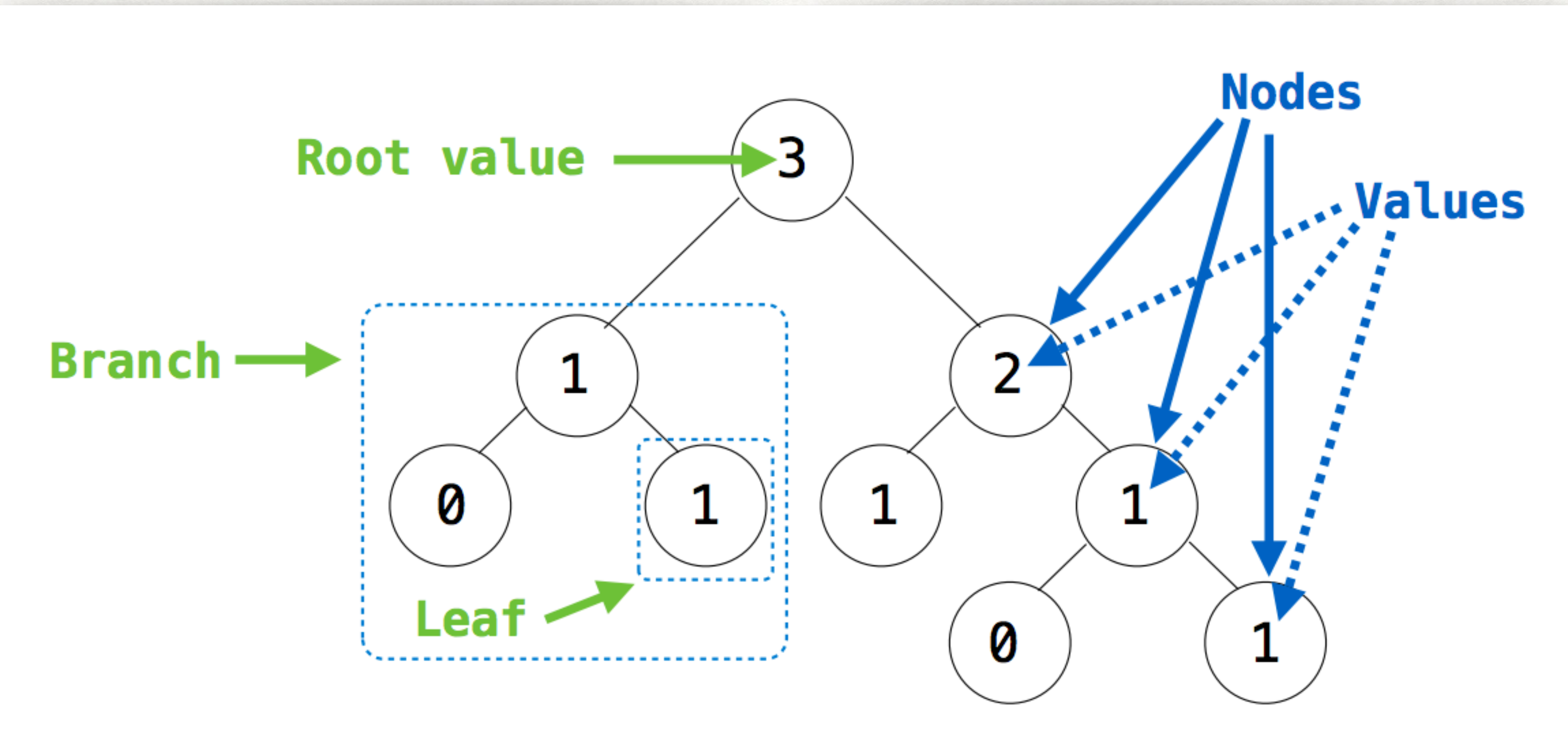
```
[[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]

[x=1, x=2, x=3, x=4]

[[y * 2 for y in [1, 2], [y * 2 for y in [2, 3], …]

[[2, 4], [4, 6], [6, 8], [8, 10]]
```

```
lst = []
for x in [1, 2, 3, 4]:
    for y in [x, x + 1]:
        lst += [[y * 2]]
```

Nested list comprehension start with outer variable.

Do inner list comprehension.

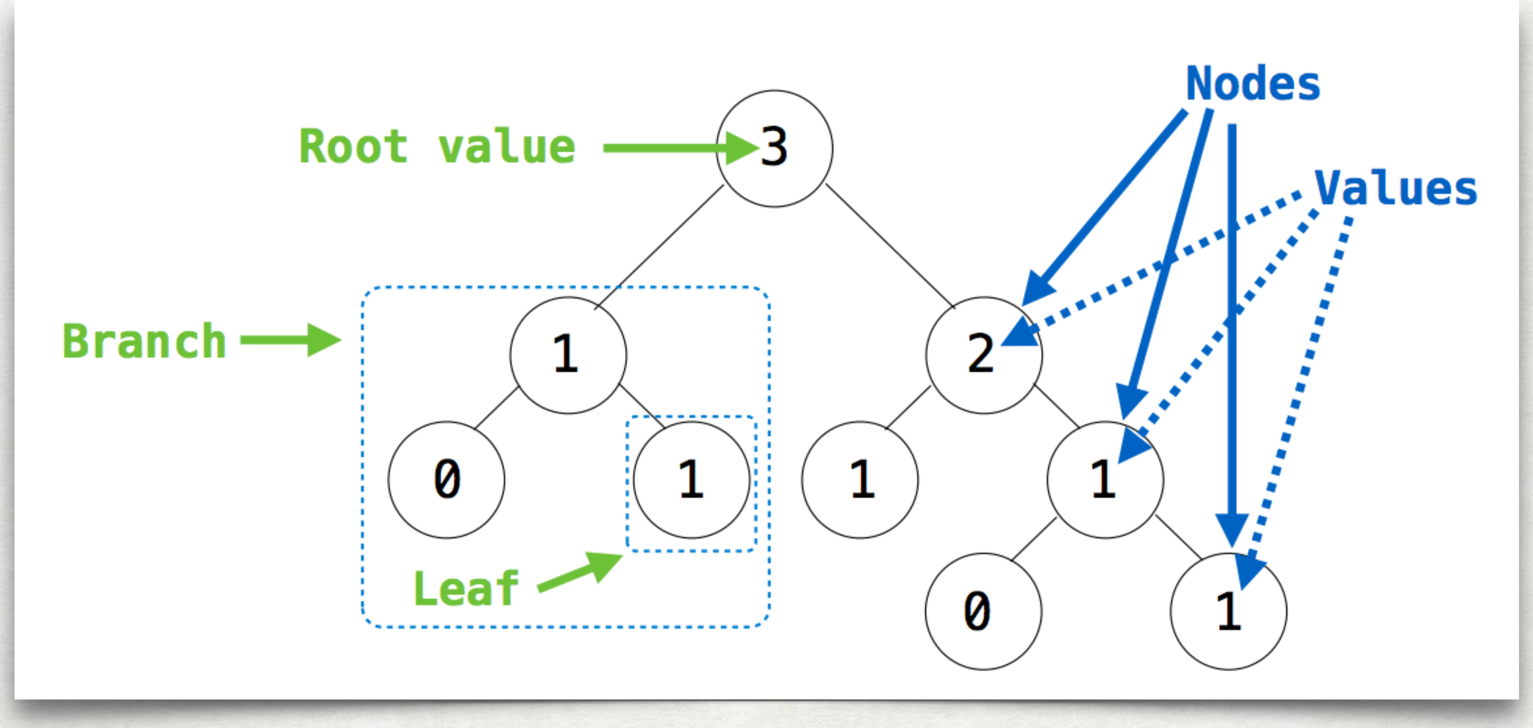
TREES



From John Denero

TREES

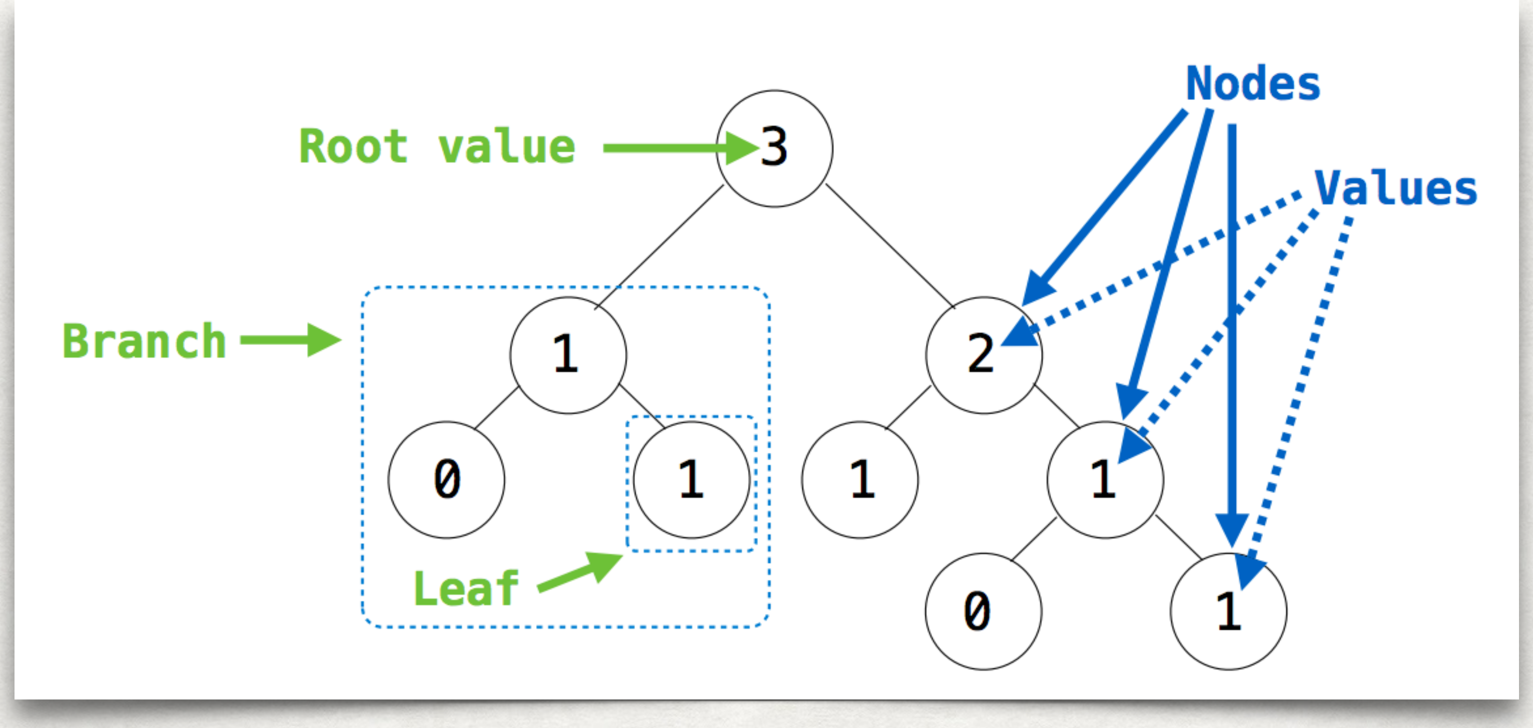
- A tree has a root. The value of the root is called the root value.
- Each branch, or subtree, is a tree and it has a root.
- Nodes are the circle and the value is within.
- Leaf nodes have no branches (or children).



From John Denero's Slides

TREES

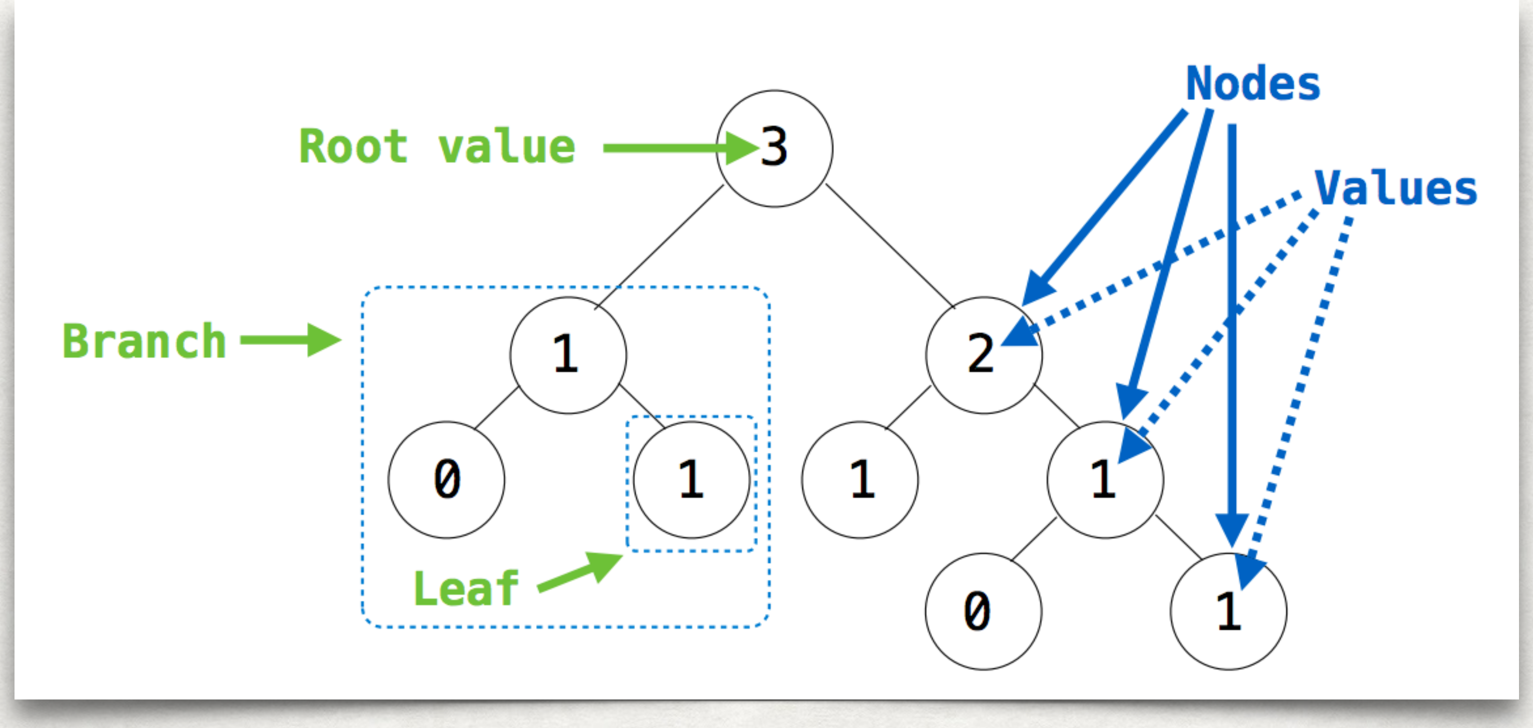
- Except for the upper most root (3), every node in the tree has only 1 **parent.**
- All nodes except for leaves have child(ren).
- Trees are recursive because subtrees and leaves are also trees.



From John Denero's Slides

TREES

- The node of 3 is the parent of the node with 1 and node with 2.
- Simpler: 3 is the parent of 1 and 2, and 2 is the child of 3.
- Note: nodes are the circle, or position at the tree. You need to actually get the value.



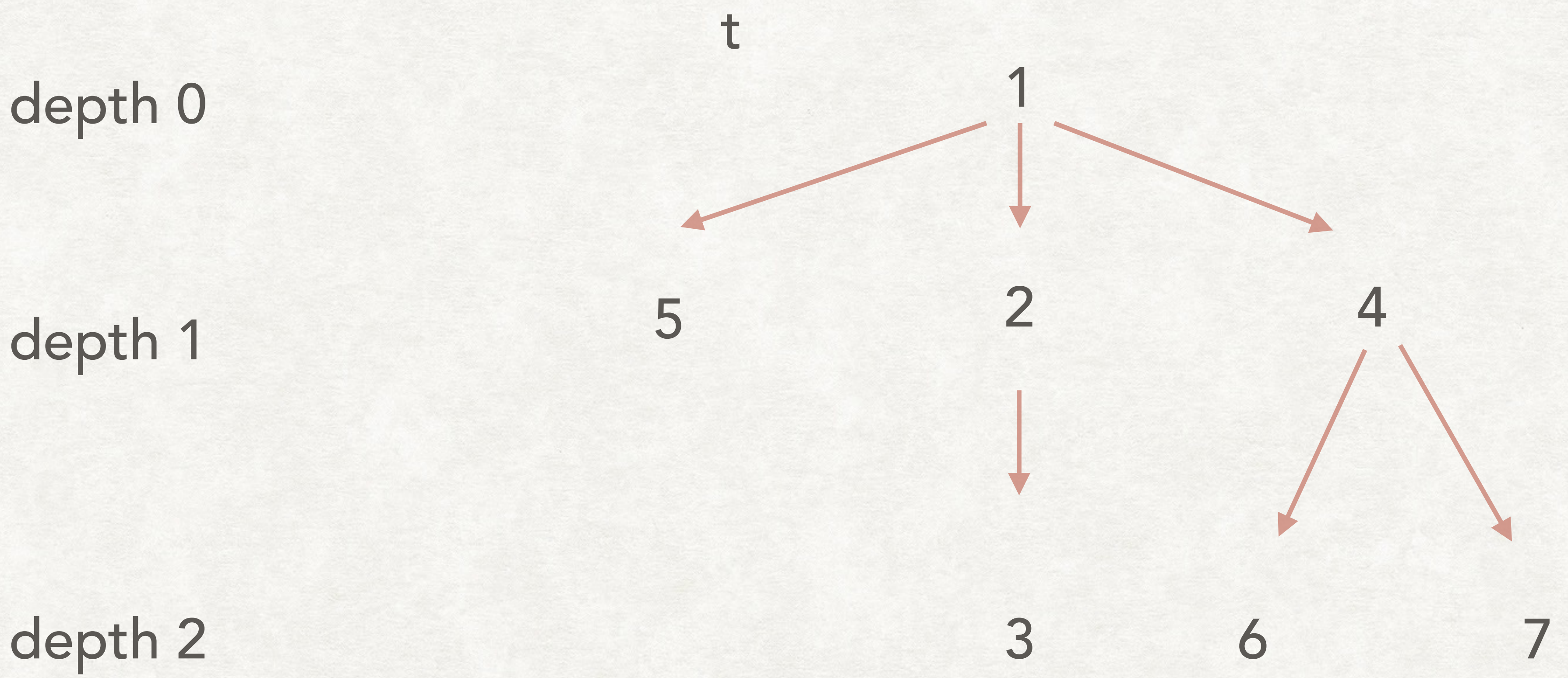
From John Denero's Slides

TREES

- The depth of a node is how far it is away from the root.
- Or count the number of edges from the root to the node.

TREES

- The depth of a node is how far it is away from the root.
- Or count the number of edges from the root to the node.

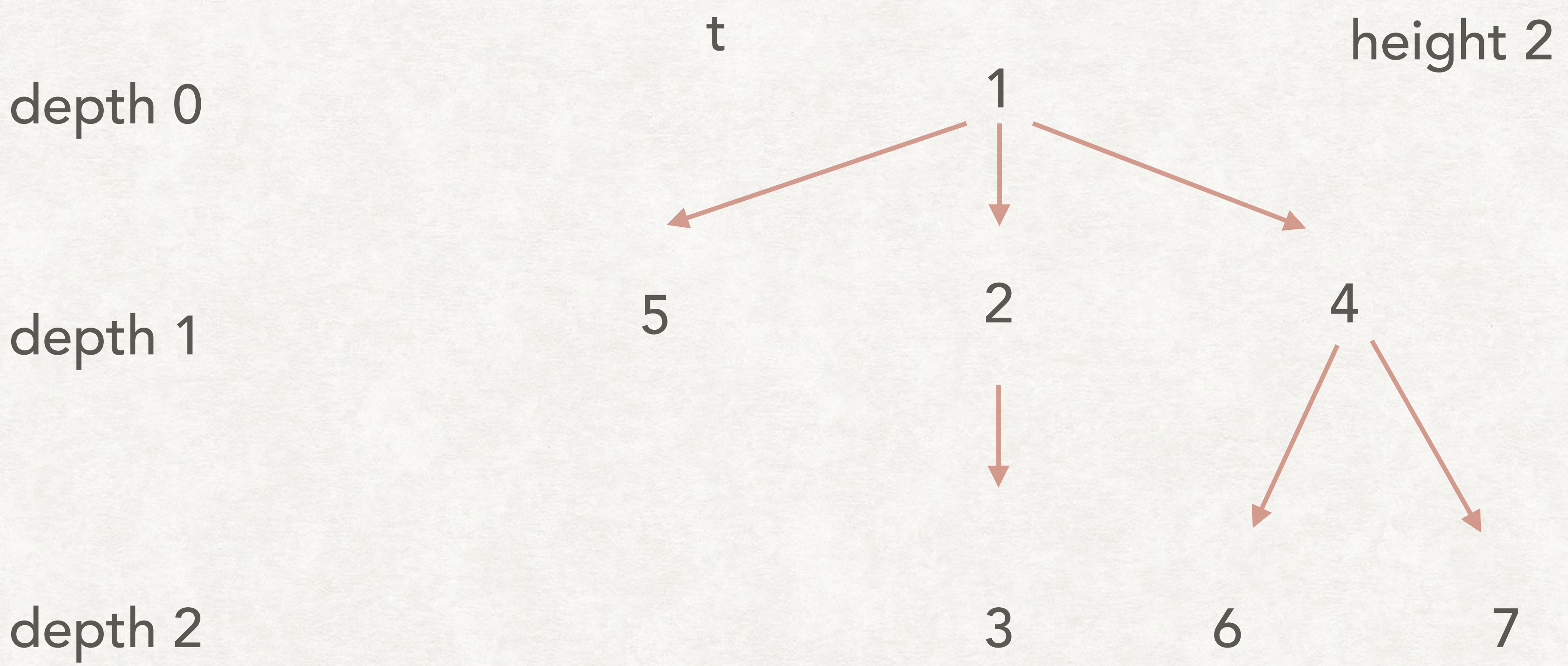


TREES

- The height of a tree is the depth of the lowest leaves.

TREES

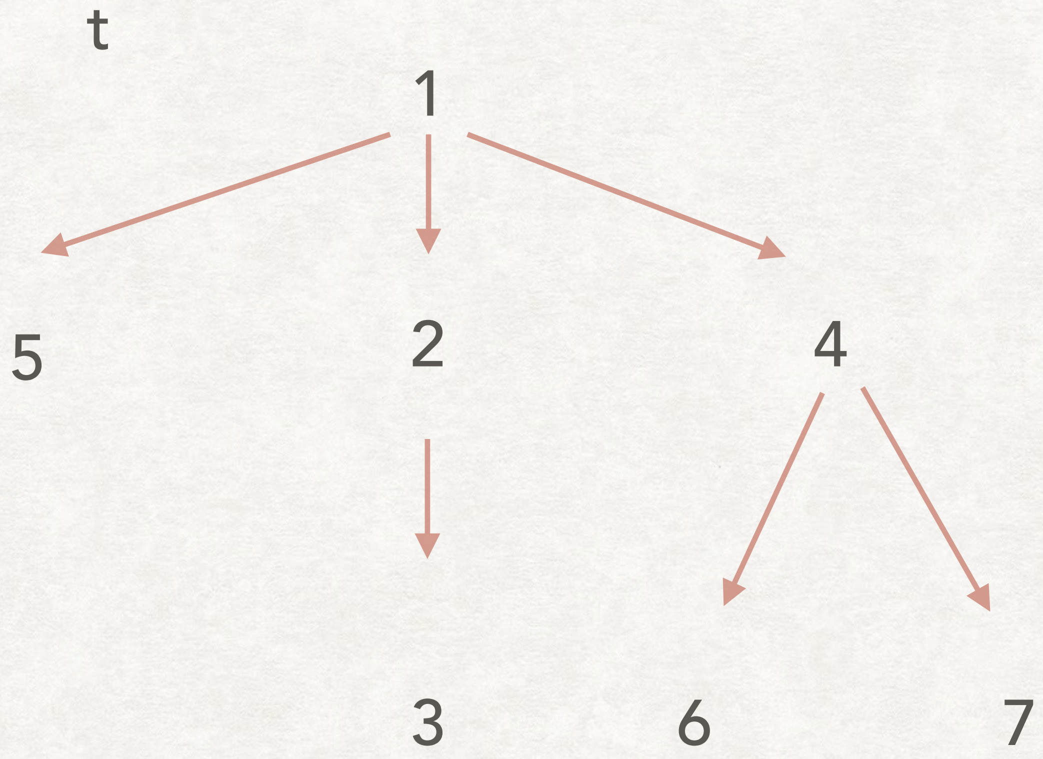
- The height of a tree is the depth of the lowest leaves.



TREES

- Our tree(root, branches=[]) constructor is implemented via Python Lists

```
t = tree(1,
     [tree(5),
     tree(2,
        [tree(3)]),
     tree(4,
        [tree(6),
        tree(7)])
     ])
```



TREES

- branches(t) returns a sequence of subtrees.
- We usually need to iterate over the branches and make recursive calls for each subtree/branch.

TREES

- For tree questions, we typically do something with the root of the tree and then for each of the tree’s branches, make the recursive call.
- The smaller problems are the tree’s subtrees, which can be accessed via the tree’s branches.

TREES

```
#Constructor
def tree(root, branches=[]):
    return [root] + list(branches)

#Selectors
def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not branches(tree)
```

tree creates a tree.

root obtains the value of the tree.

branches obtains a list of the tree's branches.

is_leaf checks if the tree has no more branches.

TREES

Return a tree with the square of every element of t

```
def square_tree(t):
```

TREES

```
def square_tree(t):
    if is_leaf(t):
        return tree(root(t)**2)
    new_branches = []
    for branch in branches(t):
        new_branches += [square_tree(branch)]
    return tree(root(t)**2, new_branches)
```

TREES

- Base case is check if tree is a leaf.
- Since each branch is a subtree, we need to make recursive calls to every branch.
- Leap of faith that **square_tree(branch)** returns the subtree with values squared.

```
def square_tree(t):
    if is_leaf(t):
        return tree(root(t)**2)
    new_branches = []
    for branch in branches(t):
        new_branches += [square_tree(branch)]
    return tree(root(t)**2, new_branches)
```

TREES

- Notice that if there are no branches, then the for loop does not iterative over anything.
- **new_branches** becomes an empty list, and the return function would work.

```
def square_tree(t):
    new_branches = []
    for branch in branches(t):
        new_branches += [square_tree(branch)]
    return tree(root(t)**2, new_branches)
```

TREES

Return a tree with the square of every element of t

```
def square_tree(t):
    new_branches = []
    for branch in branches(t):
        new_branches += [square_tree(branch)]
    return tree(root(t)**2, new_branches)

def square_tree(t):
    return tree(root(t)**2, [square_tree(branch) for branch in branches(t)])
```

TREES

Return the height of the tree

```
def height(t):
```

TREES

Return the height of the tree

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

TREES

- Since we now dealing with numbers, we need to have base case check for leaves.

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

RECAP

- Lists contain a sequence of values of which we can access via indexing.
- List slicing creates a new list of a certain portion of the original list.
- For loops are a way to iterate through sequences.
- List comprehension creates a new list in one line.
- Trees are recursive data structures that have root values and maybe other trees as their children.

APPENDIX

- Data Abstraction

DATA ABSTRACTION

- Most of the time we need to work on code that was implemented by someone else.
- Via data abstraction, we don’t need to worry about how the implementation of the data.
- We just need to know how to use the data.
- Why is it useful?

DATA ABSTRACTION

- Why is it useful?

- If we were to change the implementation of a ADTs, we only need to change the constructors and selectors.

- Any functions we wrote that used the selectors **do not** need to be changed!

DATA ABSTRACTION

- We can treat data as abstract data types
- Constructors create these ADTs
- Selectors are used to retrieve information from ADTs

DATA ABSTRACTION

Constructor:

```
def make_city(city, latitude, longitude):
    return [city, latitude, longitude]
```

Selectors:

```
def get_name(city):
    return city[0]
def get_lat(city):
    return city[1]
def get_lon(city):
    return city[2]
```

DATA ABSTRACTION VIOLATIONS

- When we use the direct implementation of an ADT rather than its selectors when writing functions, we are violating data abstraction barriers!

- This is bad because we are making an assumption on how the data is implemented.

DATA ABSTRACTION VIOLATIONS

- When we use the direct implementation of an ADT rather than its selectors when writing functions, we are violating data abstraction barriers!

```
def distance(city1, city2):
    lat_1, lon_1 = get_lat(city1), get_lon(city1)
    lat_2, lon_2 = get_lat(city2), get_lon(city2)
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

```
def distance(city1, city2):
    lat_1, lon_1 = city[1], city[2]
    lat_2, lon_2 = city[1], city[2]
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

DATA ABSTRACTION VIOLATIONS

- When we use the direct implementation of an ADT rather than its selectors when writing functions, we are violating data abstraction barriers!

```
def distance(city1, city2):
    lat_1, lon_1 = get_lat(city1), get_lon(city1)
    lat_2, lon_2 = get_lat(city2), get_lon(city2)
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

GOOD

```
def distance(city1, city2):
    lat_1, lon_1 = city[1], city[2]
    lat_2, lon_2 = city[1], city[2]
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

BAD