CS 61A Discussion 5

Mutation and Trees

Raymond Chan
Discussion 121
UC Berkeley

Agenda

- Announcements
- Linked Lists
- Trees
- Mutation
- Dictionaries
- Quiz (not after a midterm)

Announcements

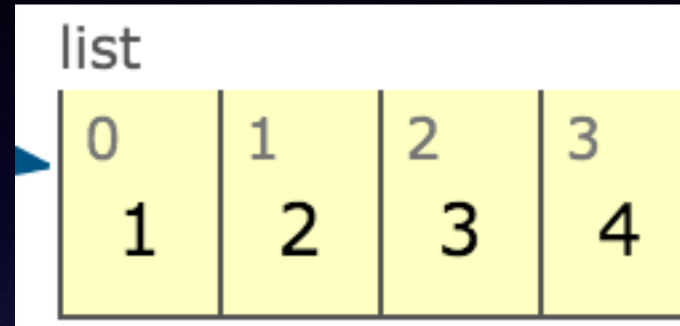
- HW 3 due Friday 2/26
- Maps Project due Tuesday 3/1
- CSM Adjunct Sections sign-ups available again
  - http://csmscheduler.herokuapp.com/

Linked Lists

- A type of sequence that connects multiple *links.*
- Each link has *first* element and a *rest* element.
  - The last link has “empty” as the rest element.
- Think of connected chains with each chain containing information.

Linked Lists

Python List



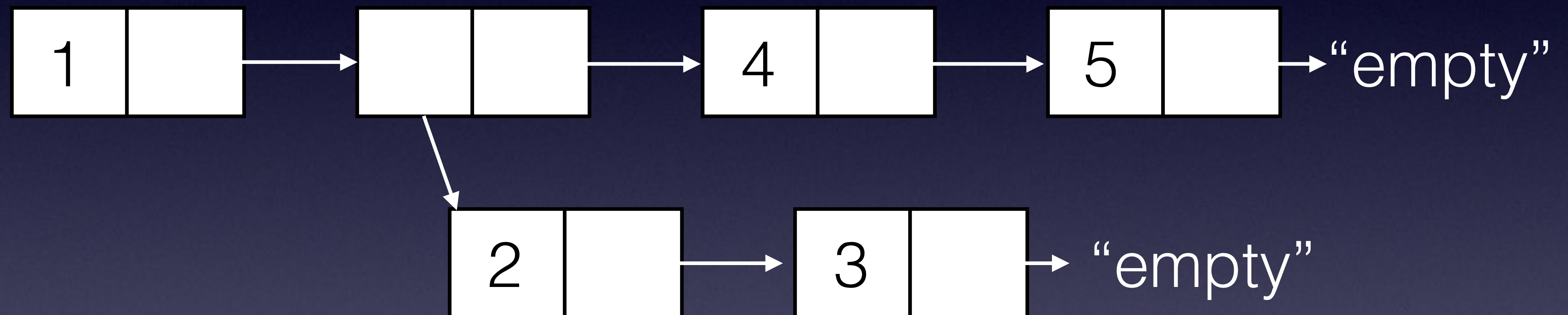
Linked List ADT

To form this linked list, use the constructor:

```
link(1, link(2, link(3, link(4, empty))))
```

Linked Lists

- The first element can also be another linked list.

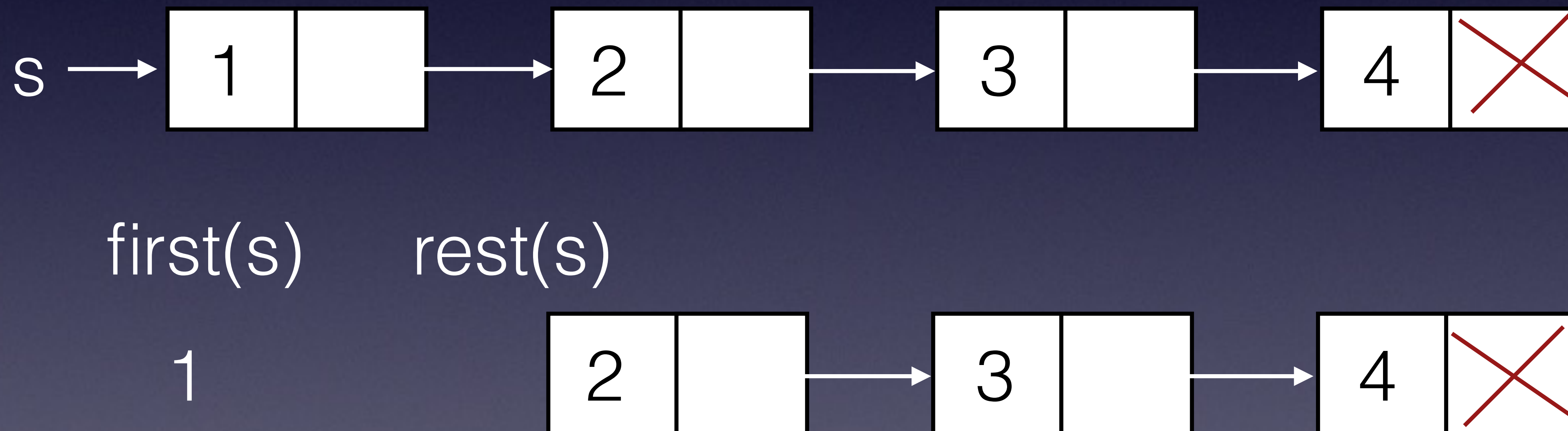


Linked Lists

- For each link box, you need to call the link constructor
  - Recursive data structure
- Selectors **first(s)** obtains the first element and **rest(s)** obtains the rest of the elements of the linked list **s**
  - **rest(s)** always returns a linked list or an empty linked list

Linked Lists

- It is very natural to use recursion for linked lists as we can split it up to **first(s)** and **rest(s)**

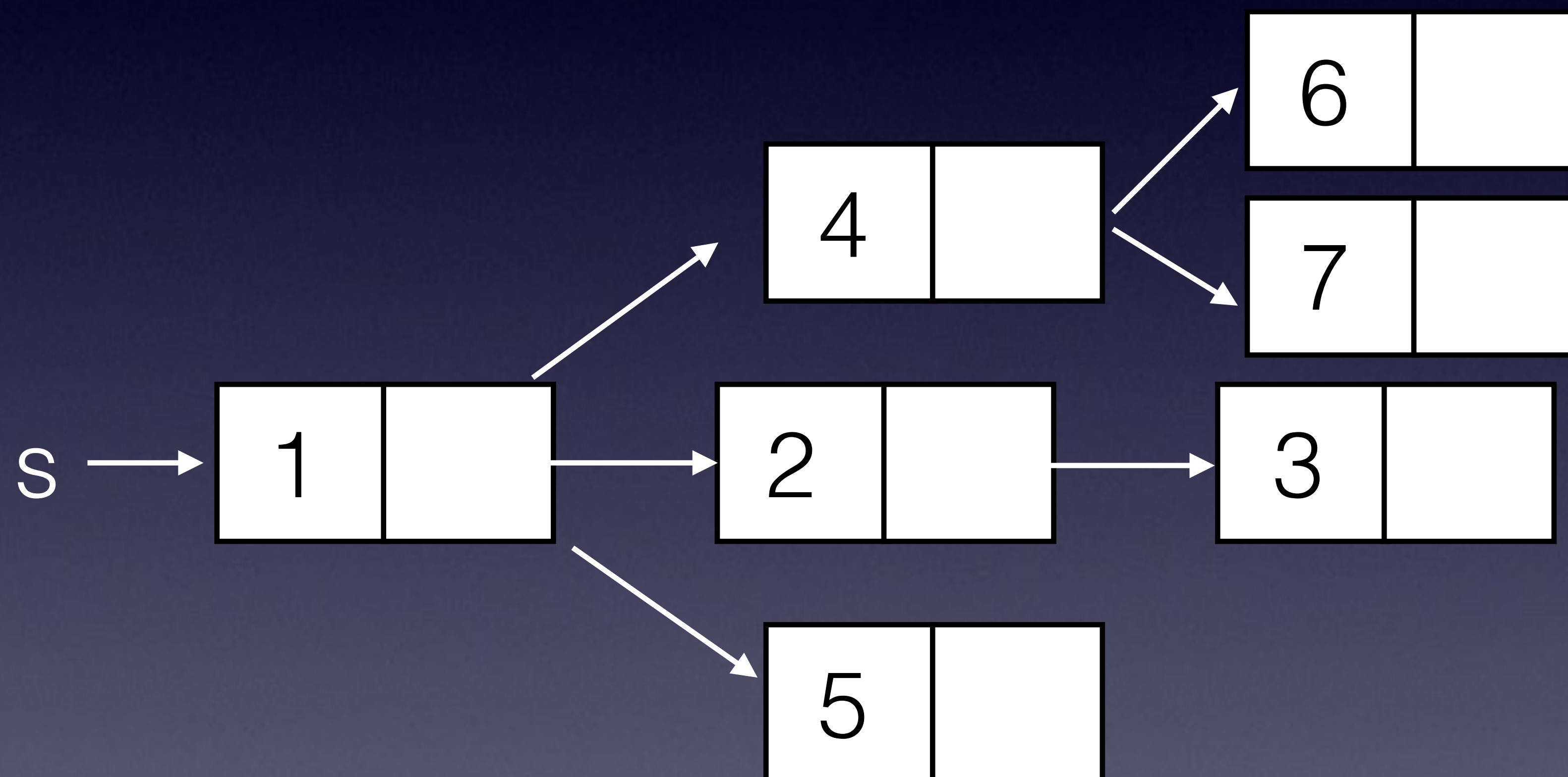


Trees

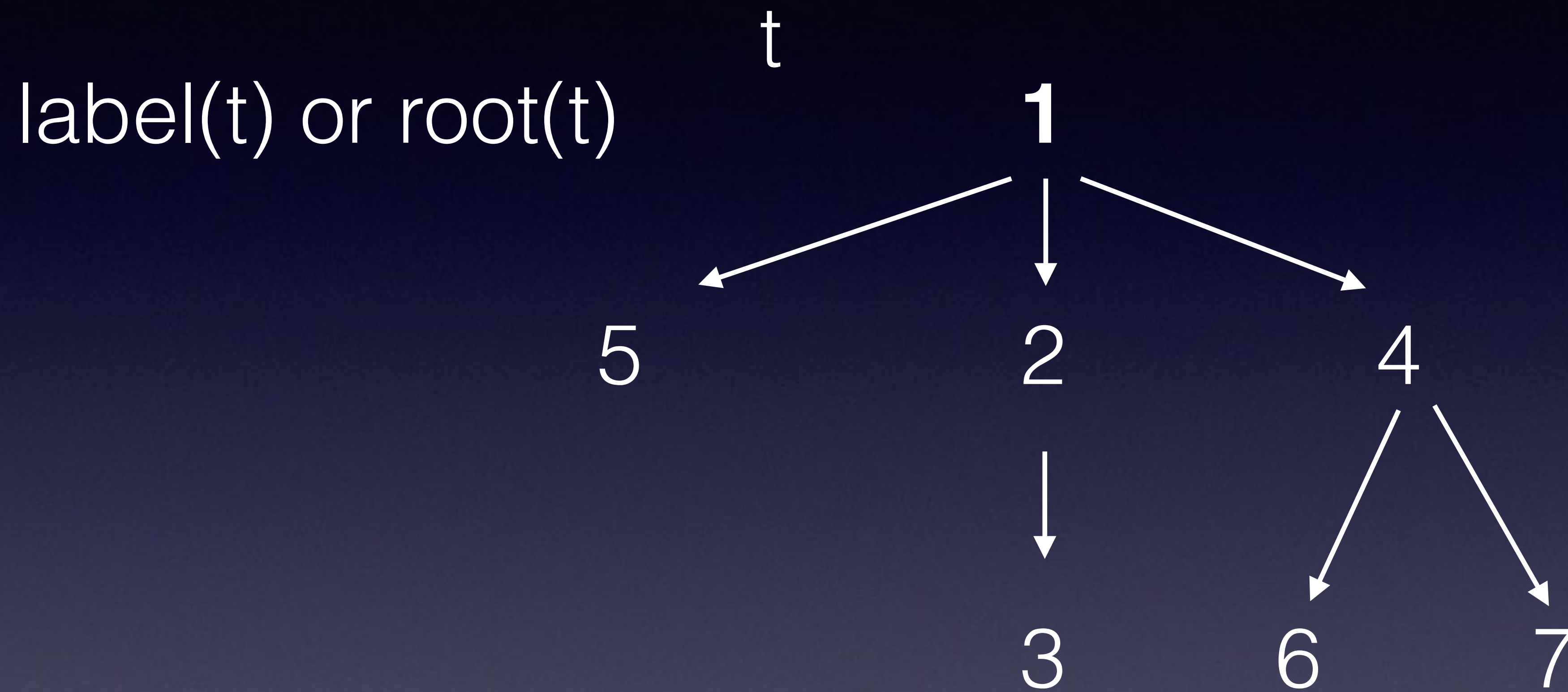
- What if a linked list’s rest can contains more than 1 link?

Trees

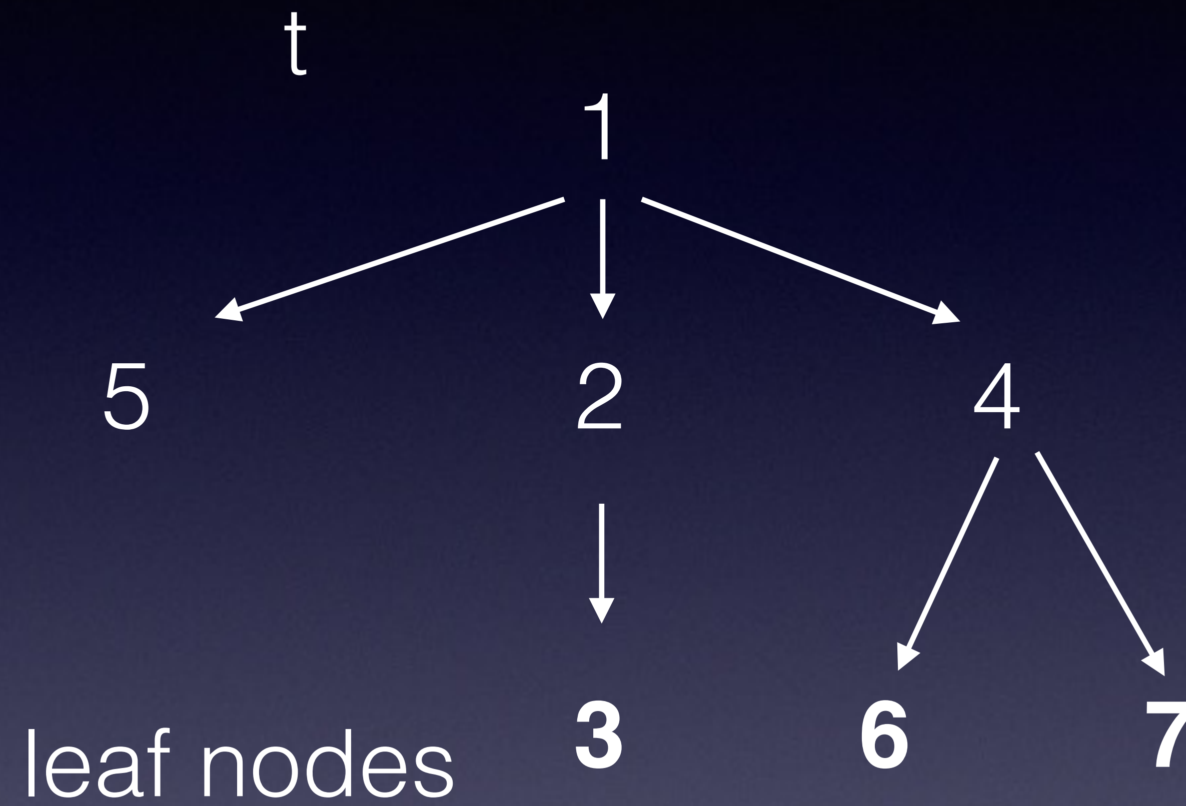
- What if a linked list's rest can contains more than 1 link?



Trees



Trees

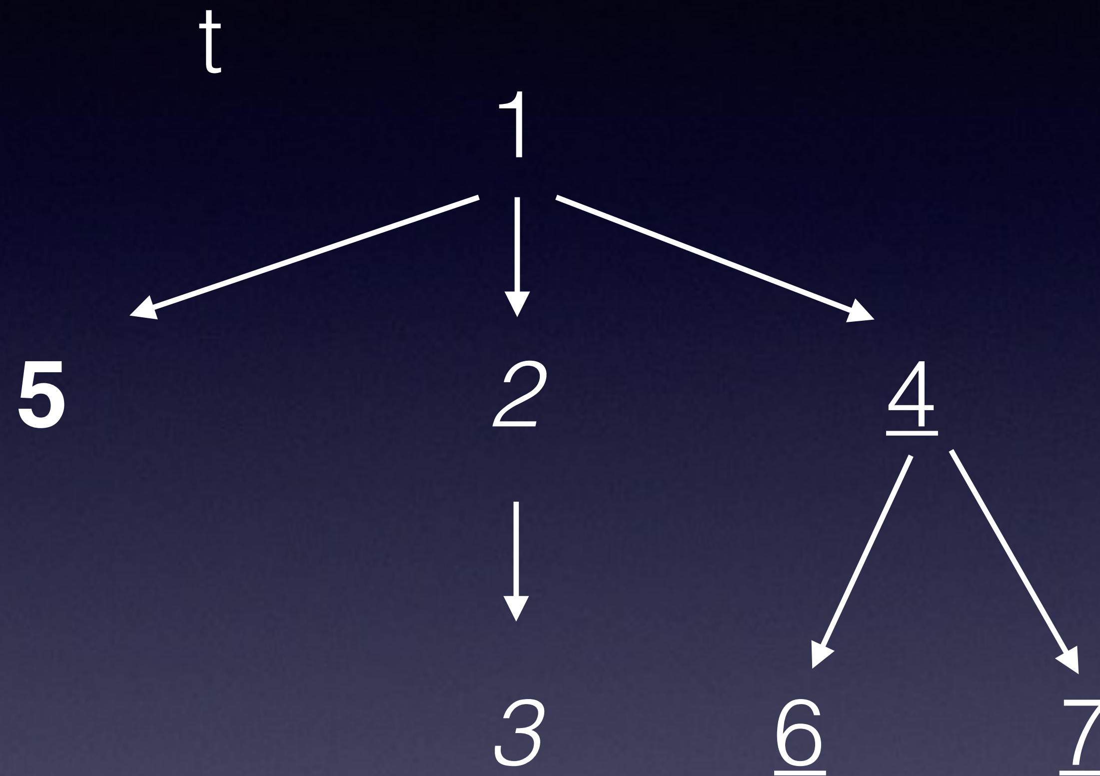


Trees

Trees

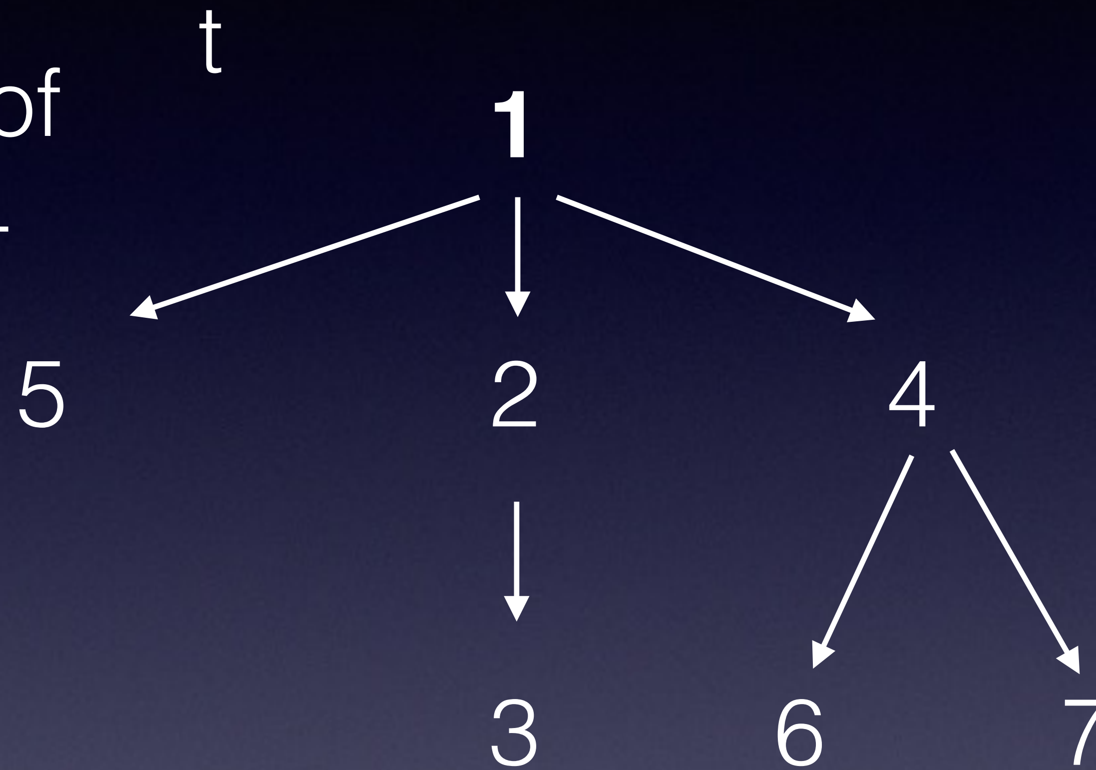
Trees are also a recursive data structure

The children of the root are smaller subtrees



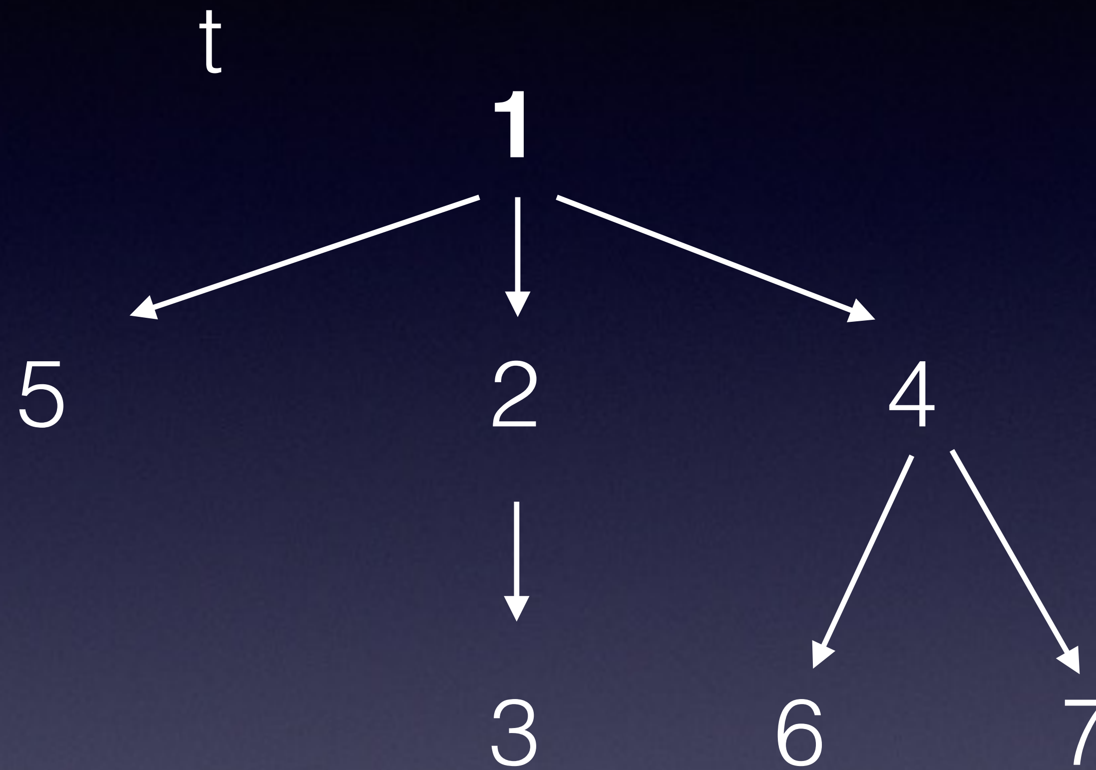
Trees

1 is parent node of nodes 5, 2, and 4

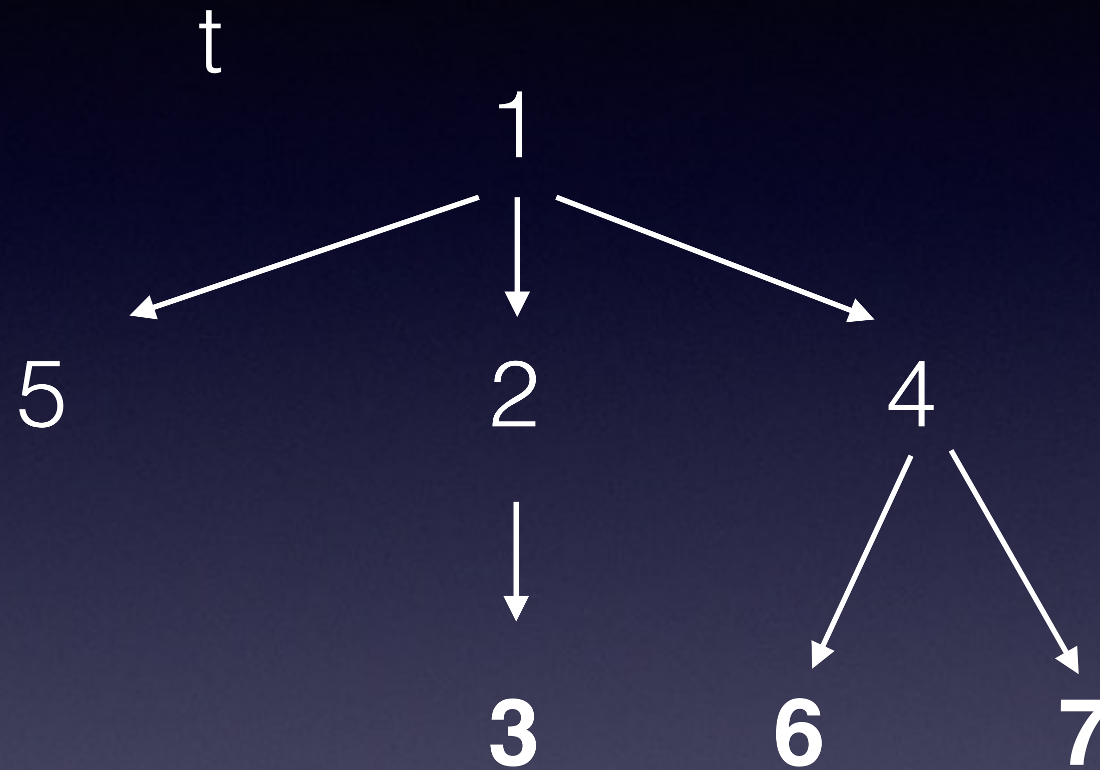


Trees

the root node
has no parent

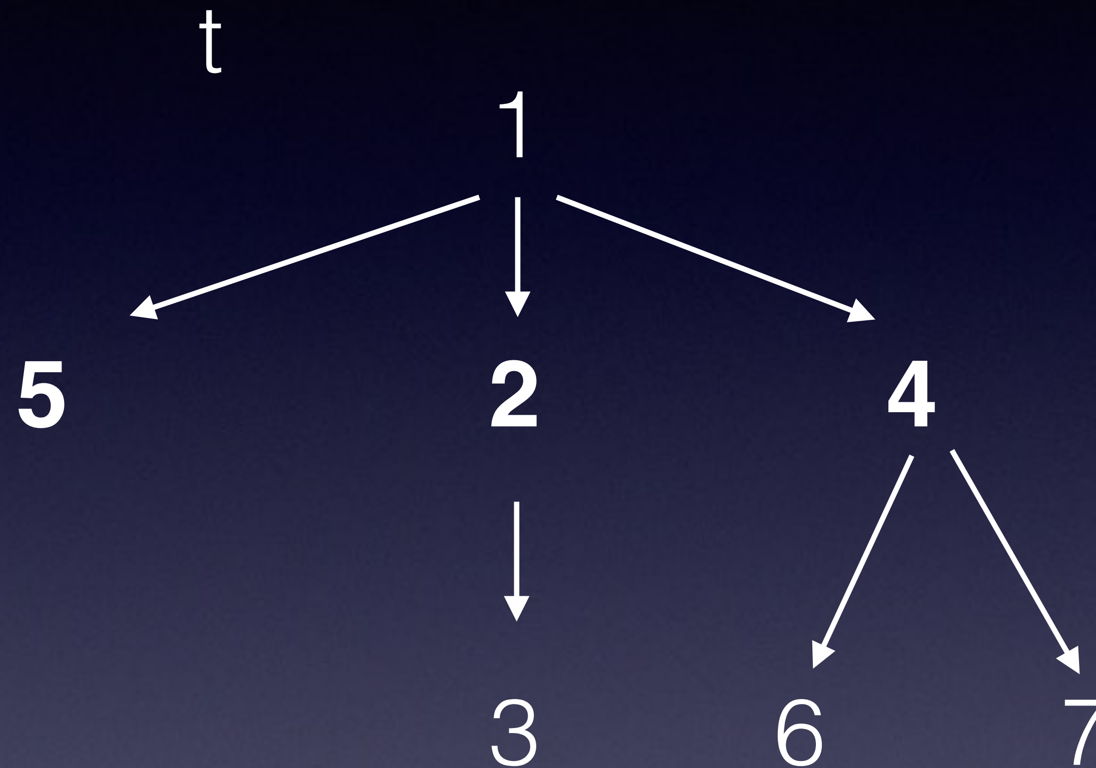


Trees



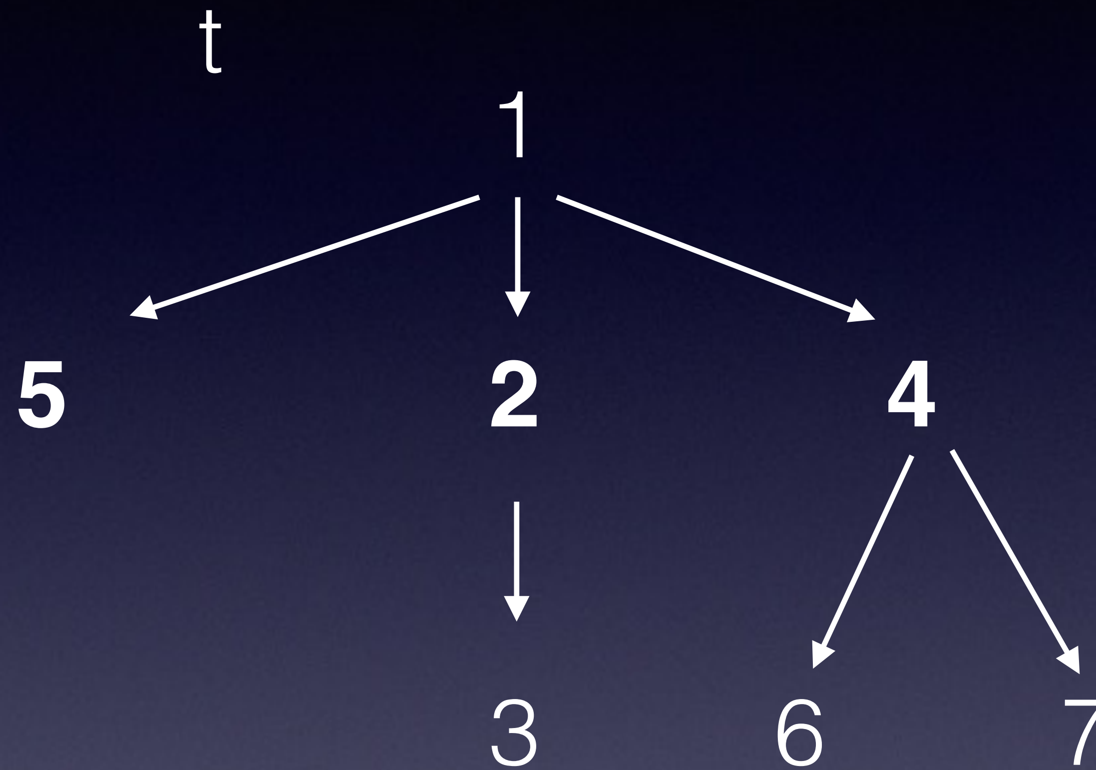
leaves do not have any children

Trees



all other nodes have a parent and at least 1 child if the size of the tree is > 1

Trees



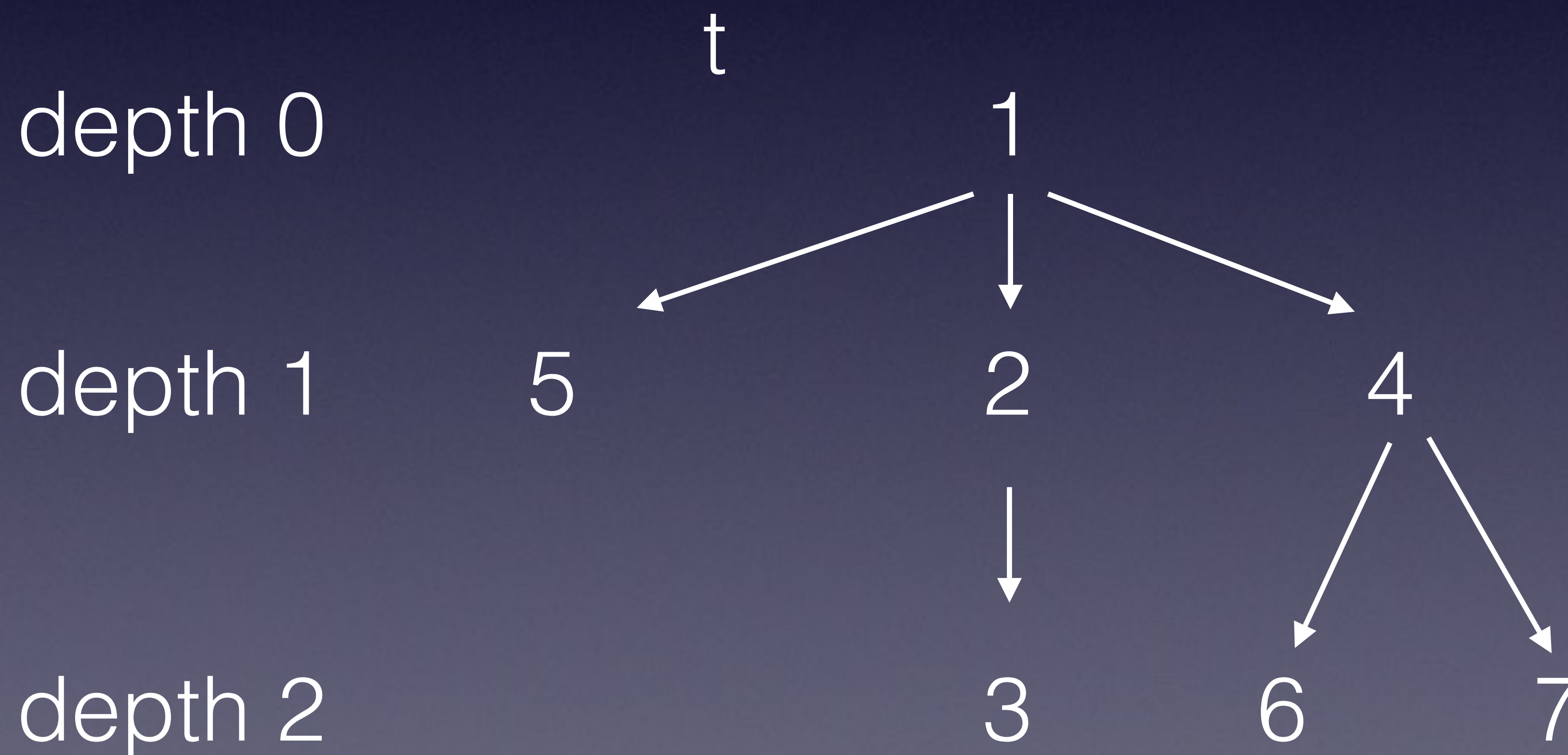
each node can only have one parent

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.

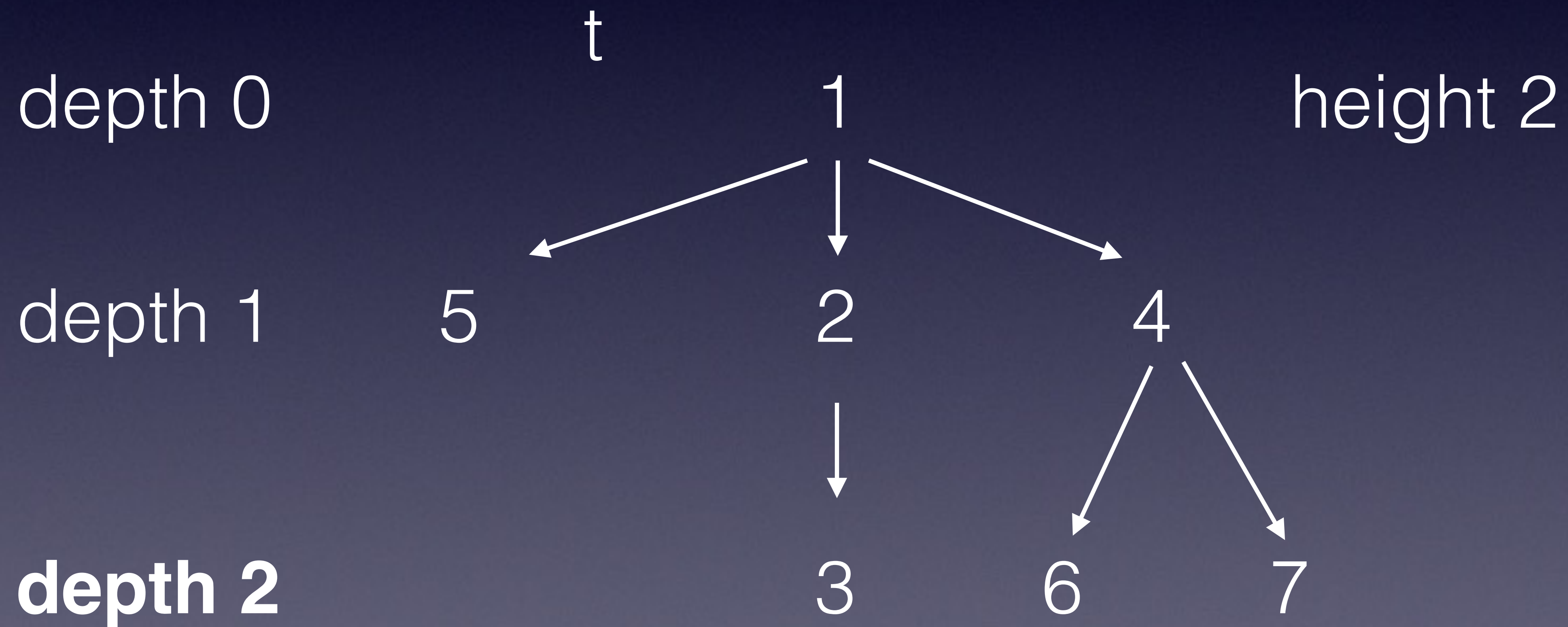


Trees

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

Trees

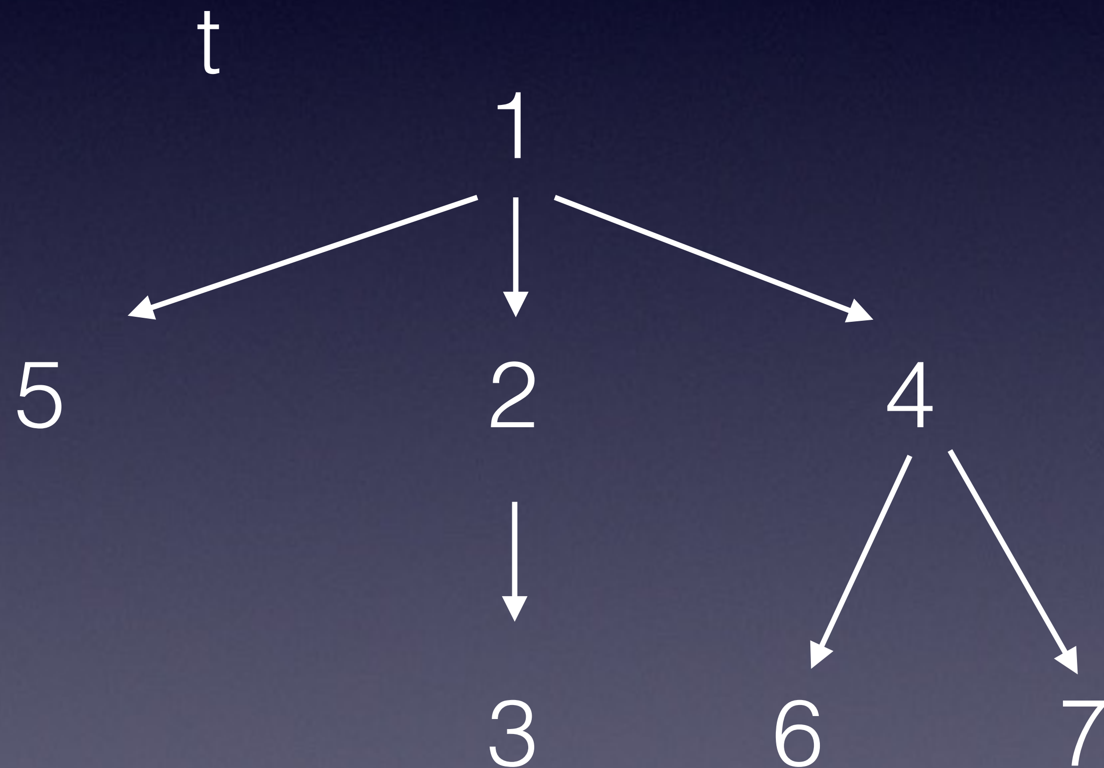
- The height of a tree is the depth of the lowest leaves.



Trees

- Our tree(label, children=[]) constructor is implemented via Python Lists

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



Trees

- children(t) returns a sequence of subtrees
- We usually need to iterate over the children and make recursive calls to each child

Trees

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

Discussion Worksheet

- Section 1.1: 1 and 3

Mutation

- When we define functions, we created function objects in environment diagrams.
- When we create lists, we create list objects.
- We can change the elements of list objects after we’ve created it.

Mutation

- When we define functions, we created function objects in environment diagrams.
- When we create lists, we create list objects.
- We can change the elements of list objects after we've created it.

```
>>> a = [1, 2, 3]
>>> a
```

Mutation

- When we define functions, we created function objects in environment diagrams.
- When we create lists, we create list objects.
- We can change the elements of list objects after we’ve created it.

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
```

Mutation

- When we define functions, we created function objects in environment diagrams.
- When we create lists, we create list objects.
- We can change the elements of list objects after we've created it.

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> a[2] = 100
>>> a
```

Mutation

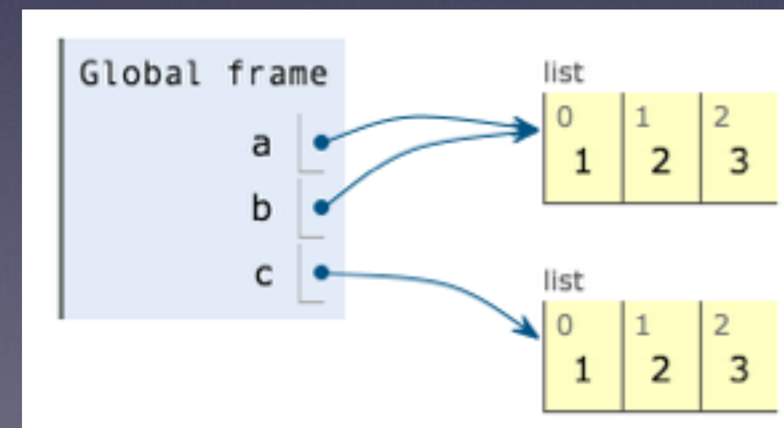
- When we define functions, we created function objects in environment diagrams.
- When we create lists, we create list objects.
- We can change the elements of list objects after we've created it.

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> a[2] = 100
>>> a
[1, 2, 100]
```

Mutation

- If I assign this variable **a** to variable **b**, **b** receives the reference.
- **a** and **b** ***is*** the same list as they are both referencing the same list object
- **a**, **b**, and **c** have the same elements, but **a** and **c** are not the same list

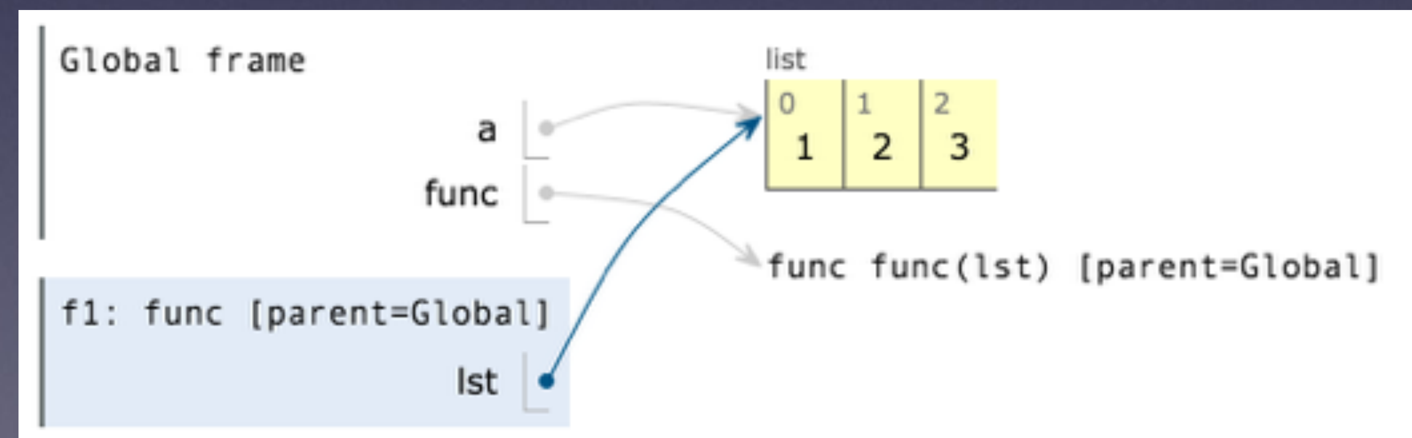
```
1  a = [1, 2, 3]
2  b = a
3  c = [1,2, 3]
```



Mutation

- When we assign a list to a variable, the variable references the list object.
- If I pass in a variable that references a list to a function argument, I am passing in the reference.
    - This is similar to passing in a function object.

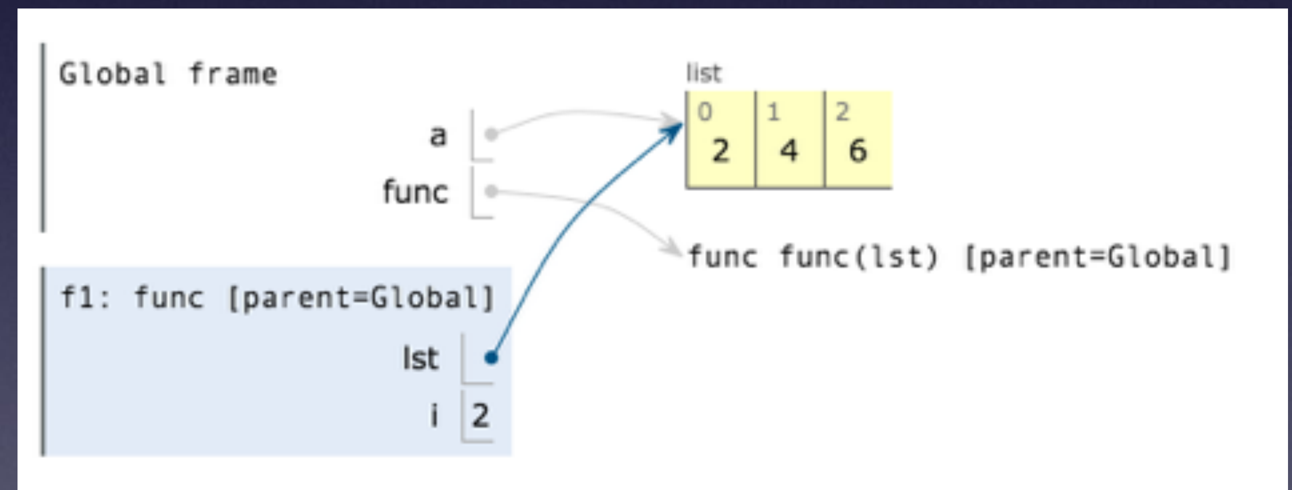
```
1  a = [1, 2, 3]
2  def func(lst):
3      for i in range(0, len(lst)):
4          lst[i] = lst[i] * 2
5
6  func(a)
```



Mutation

- Within the body of func, lst's values are changed. Notice that a's values are also changed because lst references the same list a is point to.

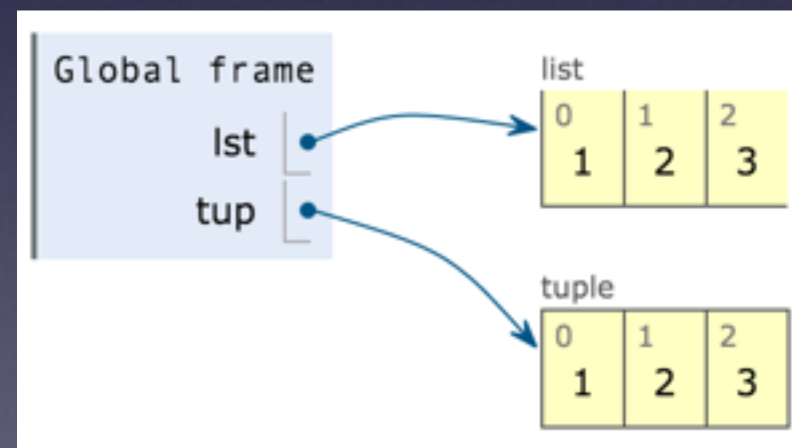
```
1  a = [1, 2, 3]
2  def func(lst):
3      for i in range(0, len(lst)):
4          lst[i] = lst[i] * 2
5
6  func(a)
```



Mutation

- Lists and dictionaries are mutable.
- Tuples and strings are immutable. Once they are created, they cannot be changed.

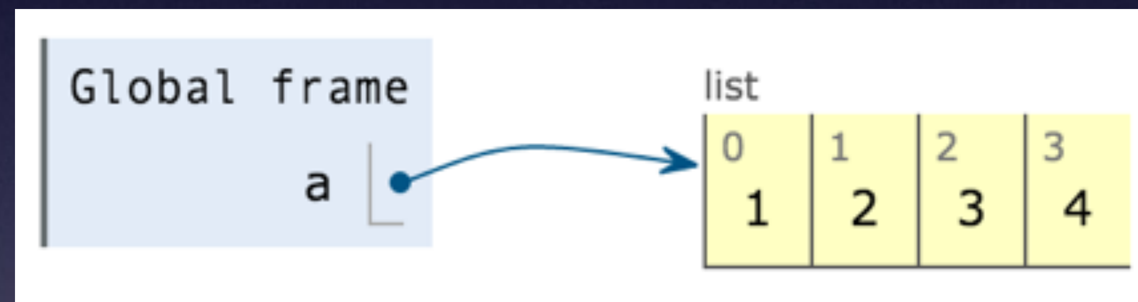
```
1  lst = [1, 2, 3]
2  tup = (1, 2, 3)
```

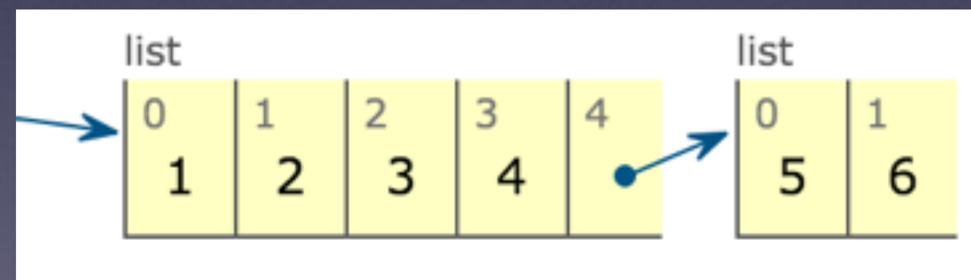


Mutation

- append(x) adds **x** to the end of the list

```
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> a.append([5, 6])
>>> a
[1, 2, 3, 4, [5, 6]]
>>> len(a)
5
```

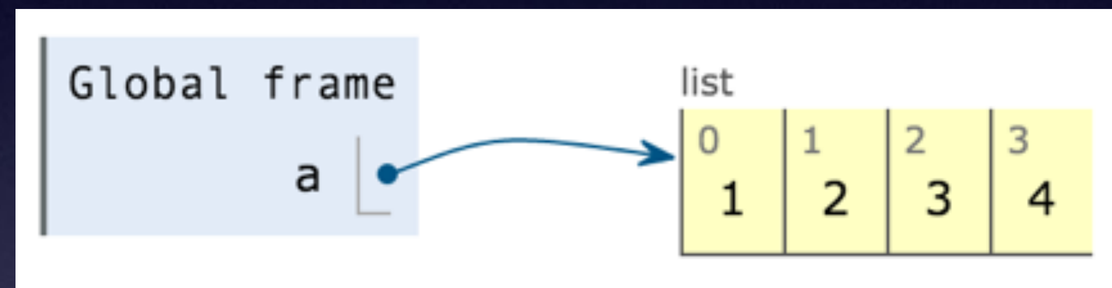


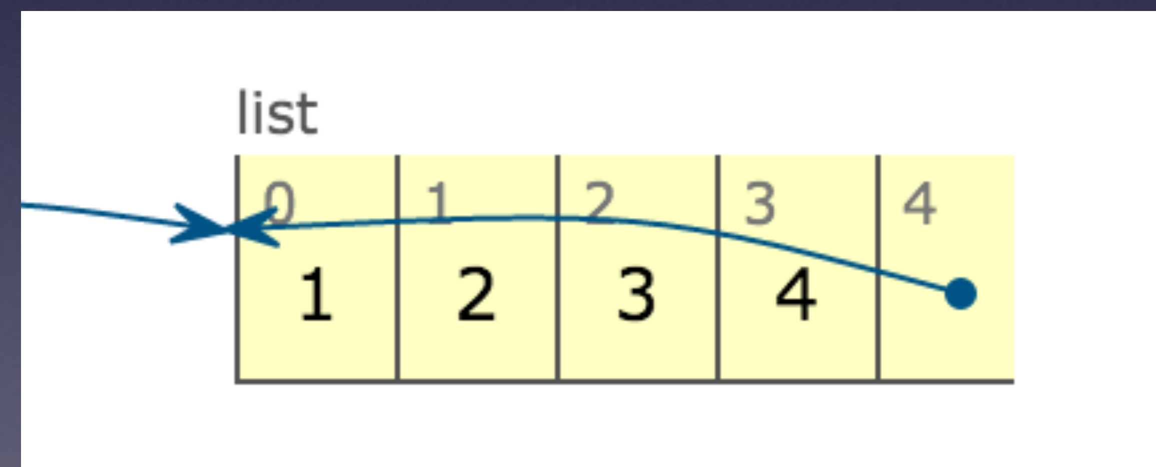


Mutation

- A list can append itself.

```
>>> a = [1, 2, 3, 4]
>>> a.append(a)
>>> a
[1, 2, 3, 4, [...]]
>>> a[4][3]
4
>>> a[4][4][4][2]
3
```





Mutation

- += for lists mutates the original list
- += is different than a = a + [1] because this re-assigns the original list

```
>>> a = [1, 2, 3, 4]
>>> b = a
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
```

```
>>> a = a + [6]
>>> a
[1, 2, 3, 4, 5, 6]
>>> b
[1, 2, 3, 4, 5]
```

Mutation

- lst1 += [lst2] -> lst1 appends each element of lst2
- which leads us to…

```
>>> a = [1, 2, 3, 4]
>>> b = a
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
```

```
>>> a = a + [6]
>>> a
[1, 2, 3, 4, 5, 6]
>>> b
[1, 2, 3, 4, 5]
```

Mutation

- extend(seq) appends each element of seq to list.

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> a
[1, 2, 3, 4]
>>> b
[3, 4]
```

Mutation

- insert(i, x) inserts **x** at index **i** by adding a new element and not replace the original element at **i**

```
>>> a = [1, 2, 3]
>>> a.insert(1, 55)
>>> a
[1, 55, 2, 3]
```

Mutation

- remove(x) removes the first time we see **x** in a list, otherwise errors

```
>>> a = [1, 2, 3, 2, 5, 1]
>>> a.remove(2)
>>> a
[1, 3, 2, 5, 1]
```

Mutation

- pop(i) returns and removes the element at index **i**. By default, **i** is the last element

```
>>> a = [1, 2, 3, 2, 4, 1]
>>> a.pop()
1
>>> a.pop(3)
2
>>> a
[1, 2, 3, 4]
```

Mutation Q1

http://tinyurl.com/mutation-q1

Mutation Questions

Q2 on page 9

Dictionaries

- Dictionaries map keys to values.
- Python dictionaries are unordered.
- We can obtain a key's mapped value by indexing into the dictionary via the key.
- We can add key-value pairs anytime and can also replace a key's value with something else.

Dictionaries

- A dictionary key can be any **immutable** value.
- If we try to place an entry with a mutable key (i.e. list), we will get an unhashable type error.
- We can check whether a dictionary contains a key with **in**.
- However, to check if a dictionary contains a value, need to iterate through the dictionary

Dictionaries

```
>>> numerals = {"I":1, "II":2, “III":3}
>>> numerals[“II"]
2
>>> numerals["IV"] = 4
>>> numerals
{"I":1, "II":2, "III":3, “IV":4}
>>> numerals[“I"] = 100
>>> numerals
{"I":100, "II":2, "III":3, “IV”:4}
>>> "I" in numerals
True
>>> 100 in numerals
False
```

Dictionaries

- We can iterate over a dictionary's keys.

Dictionaries

- We can iterate over a dictionary's keys.

```
for key in dictionary
```

Dictionaries

- We can iterate over a dictionary's keys.

```
for key in dictionary

for key in dictionary.keys()
```

Dictionaries

- We can iterate over a dictionary's keys.

```
for key in dictionary

for key in dictionary.keys()
```

- We can iterate over a dictionary's values.

Dictionaries

- We can iterate over a dictionary's keys.

```
for key in dictionary

for key in dictionary.keys()
```

- We can iterate over a dictionary's values.

```
for value in dictionary.values()
```

Dictionaries

- We can iterate over a dictionary's keys.

```
for key in dictionary

for key in dictionary.keys()
```

- We can iterate over a dictionary's values.

```
for value in dictionary.values()
```

- We can iterate over a dictionary's keys and values at the same time.

Dictionaries

- We can iterate over a dictionary's keys.

```
for key in dictionary

for key in dictionary.keys()
```

- We can iterate over a dictionary's values.

```
for value in dictionary.values()
```

- We can iterate over a dictionary's keys and values at the same time.

```
for key, value in dictionary.items()
```

Dictionaries

- We can delete a dictionary’s key-value pair.

Dictionaries

- We can delete a dictionary’s key-value pair.

```
>>> a = {"a":1, "b":2, "c":3, "d":4}
>>> del a["a"]
>>> a
{"b":2, "c":3, "d":4}
```

Dictionaries

- We can delete a dictionary's key-value pair.

```
>>> a = {"a":1, "b":2, "c":3, "d":4}
>>> del a["a"]
>>> a
{"b":2, "c":3, "d":4}
```

- We can delete a key and return its value.

Dictionaries

- We can delete a dictionary's key-value pair.

```
>>> a = {"a":1, "b":2, "c":3, "d":4}
>>> del a["a"]
>>> a
{"b":2, "c":3, "d":4}
```

- We can delete a key and return its value.

```
>>> a.pop(“d")
4
>>> a
{"b":2, "c":3}
```

Dictionaries Questions

- Q2, Q3, and Q4

Recap

- Linked lists chains of link elements where the first is some information and the rest is another linked list.
- Trees are recursive data structures that have root values and maybe other trees as their children.
- Dictionaries contain key value pairs to store information.
- Lists and dictionaries are mutable. Tuples and strings are immutable.
- Python list objects are references with pointers. When calling functions that takes a list, we pass in the reference (or pointer) and not create a new list.