Code Platoon

Code Camp for Veterans

Recursion

Topics Covered / Goals

- Why? The basic use case for recursion
- What? Defining the recursive approach
- How? the call stack and stack overflow
- Recursive Data Structures: Trees, Binary Trees
- Solving Binary Search using a Binary Tree

Why? The basic use case for recursion ★

- Recursion is ultimately just another approach to solving algorithmic problems
- It is important to consider as it ‘fits’ some problems better than a standard imperative approach
- Let’s first compare it to the standard imperative approach

Factorial (Imperative)

- Definition:

$$n! = n * (n-1) * (n-2) * \ldots * 1$$

- Examples:

1!	1	= 1
2! =	2 * 1	= 2
3! =	3 * 2 * 1	= 6
4! =	4 * 3 * 2 * 1	= 24

Imperative solutions

- Imperative: a ‘step by step’ solution
    - Loops and Conditionals are the main tools
    - Variables help us build up our solution as we go

Factorial implemented (imperative)

See curriculum/week-03/day3/code/1-simple-factorial.py

scratchpad.py

```python

def factorial(n):
    result = 1

    for i in range(n, 1, -1):
        result *= i

    return result


print(factorial(5))

```

Factorial imperative (walkthrough)

- So what is actually happening when we run this code?
- We build up our solution through multiple passes of a loop, using a variable to store/update the solution

Factorial imperative (walkthrough)

- Call factorial(5)
    - Enter function body

Factorial imperative (walkthrough)

- Inside factorial body
    - n = 5
    - result = 1
    - Iterate i through [5, 4, 3, 2, 1]

Factorial imperative (walkthrough)

- Inside factorial body
    - i = 5
    - result = 1 * 5

Factorial imperative (walkthrough)

- Inside factorial body
    - i = 4
    - result = 5 * 4

Factorial imperative (walkthrough)

- Inside factorial body
    - i = 3
    - result = 20 * 3

Factorial imperative (walkthrough)

- Inside factorial body
  - i = 2
  - result = 60 * 2

Factorial imperative (walkthrough)

- Inside factorial body
    - result = 120
    - Return result

- factorial (5) is now replaced with 120 at calling site

Recursion

- Alternative Factorial Definition:

$$n! = n * (n-1)!$$

- **Recursion:** defining a function *in terms of itself*

Factorial implemented (recursive)

See curriculum/week-03/day3/code/2-recursive-factorial.py

scratchpad.py

```python

def factorial(n):
    if n == 1:
        return 1

    return n * factorial(n-1)


print(factorial(5))



```

Recursive solution

- Recursion consists of two core ideas:

1) The ‘*recursive step*’: building up a solution in terms of a ‘recursive’ call to the same function, with modified inputs

2) The ‘*base case*’: A ‘terminating’ case, does not involve a call to itself, so allows the string of recursive calls to ‘collapse’

The Call Stack

- With recursion we can call a function from within itself
- This works because of an idea called the '*call stack*'
- As you may have guessed, the call stack is an example of a Stack - first-in, last out!
- First, let's explore the call stack without reference to recursion

Call Stack (example function)

scratchpad.py

```python
database = {
    "Benjamin": {
        "age": 35,
    }
}


def get_from_DB(name, key):
    return database[name][key]


def get_info_string(name):
    age = get_from_DB(name, "age")
    return f"Benjamin is {age} years old"


print(get_info_string("Benjamin"))

```

Call Stack (walkthrough)

Call stack starts out empty

Call Stack → [ ]

Call Stack (walkthrough)

Encounter first function call (print), push it onto the stack

Call Stack ⇒ [

print(get_info_string("Benjamin"))

]

Call Stack (walkthrough)

print can't be called until we compute it's parameter,

so push **get_info_string** onto the stack

Call Stack ⇨ [

get_info_string("Benjamin")
print(...)

]

Call Stack (walkthrough)

get_info_string calls a function within its own body,

so push that onto the stack

```
Call Stack  ⇨  [
        get_from_DB("Benjamin", "age"),
        get_info_string("Benjamin")
        print(...)
]
```

Call Stack (walkthrough)

get_from_DB("Benjamin", "age") returns an actual value, so pop it from the stack and continue computing

Call Stack → [

get_info_string("Benjamin")
print(...)

]

Call Stack (walkthrough)

No more function calls in **get_info_string** means we can return its value and pop it off the stack

Call Stack ⇨ [

print(“Benjamin is 35 years old”)

]

Call Stack (walkthrough)

Finally we can execute print, return it’s value (None)

and pop it off the call stack

Call Stack → []

Call Stack (factorial)

Back to factorial, how does the call stack handle our recursive function?

Call Stack → [
factorial(5)
]

Call Stack (factorial)

First push new function calls onto the stack

Call Stack → [
factorial(4)
5 * factorial(4)
]

Call Stack (factorial)

We will continually add new frames to the stack on our recursive step

Call Stack → [

factorial(3)

4 * factorial(3)

5 * factorial(4)

]

Call Stack (factorial)

We will continually add new frames to the stack on our recursive step …

```
Call Stack            [
        factorial(2)
        3 * factorial(2)
        4 * factorial(3)
        5 * factorial(4)
]
```

Call Stack (factorial)

… **until …**

```
Call Stack  =>  [
          factorial(1)
          2 * factorial(1)
          3 * factorial(2)
          4 * factorial(3)
          5 * factorial(4)
]
```

Call Stack (factorial)

We hit the base case, now we can replace calls to factorial(1) with 1 …

```
Call Stack  ⇨  [
        1
        2 * factorial(1)
        3 * factorial(2)
        4 * factorial(3)
        5 * factorial(4)
]
```

Call Stack (factorial)

And factorial(2) with 2 …

Call Stack [
2 * 1
3 * factorial(2)
4 * factorial(3)
5 * factorial(4)
]

Call Stack (factorial)

And on and on we ‘collapse’ the chain of function calls

```
Call Stack  ⇨  [
         3 * 2
         4 * factorial(3)
         5 * factorial(4)
]
```

Call Stack (factorial)

And on and on we ‘collapse’ the chain of function calls

```
Call Stack  ⇨  [
          4 * 6
          5 * factorial(4)
]
```

Call Stack (factorial)

And on and on we ‘collapse’ the chain of function calls

```
Call Stack  ⇨  [
          5 * 24
]
```

Call Stack (factorial)

Until we have a single value, to replace factorial(5) at it’s calling site

Call Stack ⇨ [
120
]

Stack Overflow

Call Stacks cannot grow forever!

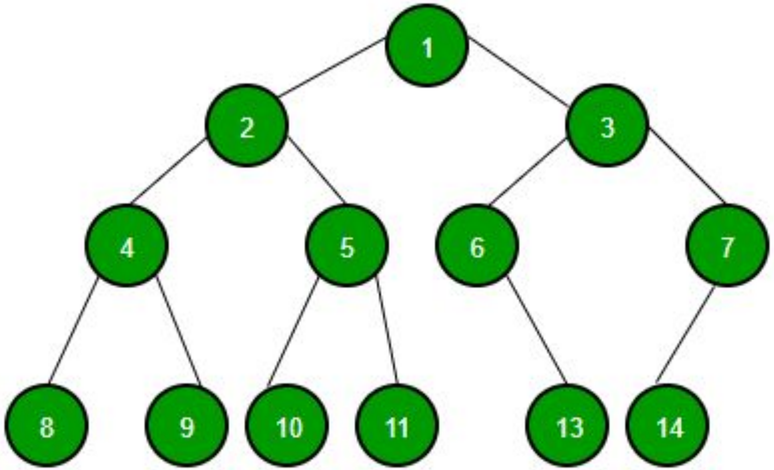
```
# This is missing it's base case on purpose, try running it to see what happens

def factorial(n):
    return n * factorial(n-1)


print(factorial(5))  # RecursionError: maximum recursion depth exceeded
```

Data Structures

- So what does all this have to do with data structures?
- Just like a function can be recursive (self-referential), a Data Structure can have a similar quality called '*optimal substructure*'
- *Optimal substructure* means that a data structure is 'defined in terms of itself'
- Trees are a perfect example

Trees

This is a ‘binary tree’, because each node has a maximum of two children



Trees (definiton)

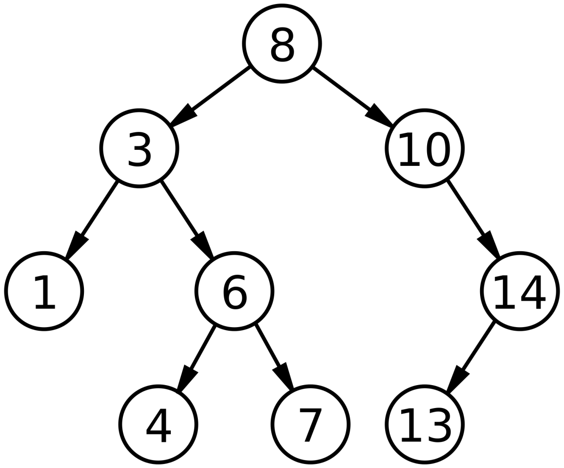
- Similar to our recursive function definition, Trees can have one of two forms:
  - A ‘leaf’: this is our base base, a Tree with 0 children
  - A ‘tree node’: our recursive case, a Tree 1 or 2 children

Binary Search Trees

- In addition to being a binary tree, a binary search tree has two additional properties:
  - The left node (and all of its children) have a value smaller than the parent node
  - The right node (and all of its children) have a value greater than the parent node

Binary Search Trees

Confirm for yourself this counts as a BST



Trees (implemented)

See curriculum/week-03/day3/code/4-binary-search-tree.py

Trees (implemented)

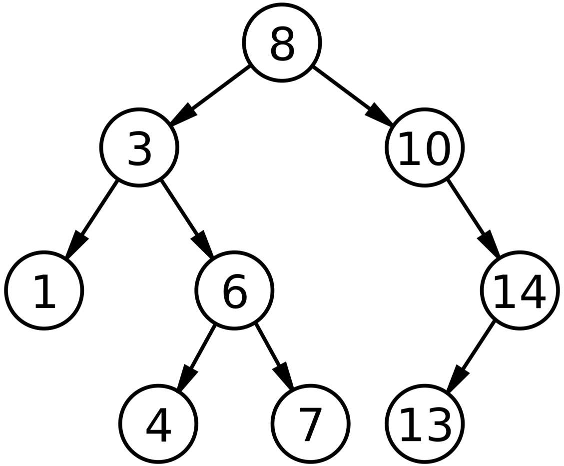
See curriculum/week-03/day3/code/4-binary-search-tree.py

Binary Search

- Given a Binary Search Tree, can we find a given value?
- Because of our guarantees we always know to look in the left *or* right (not both)

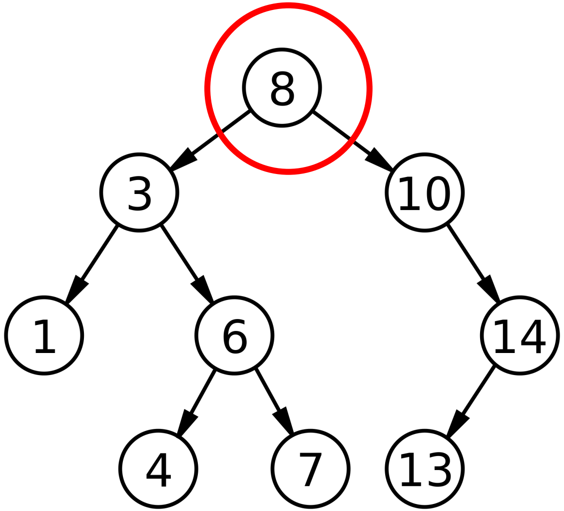
Binary Search

Looking for 7



Binary Search

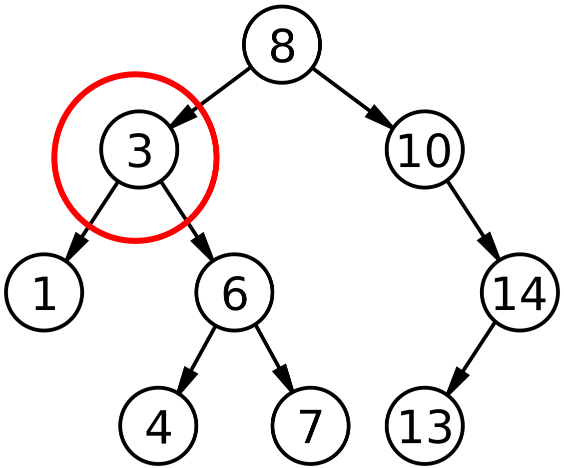
Looking for 7



- 7 != 8
- 7 < 8
- Search left subtree

Binary Search

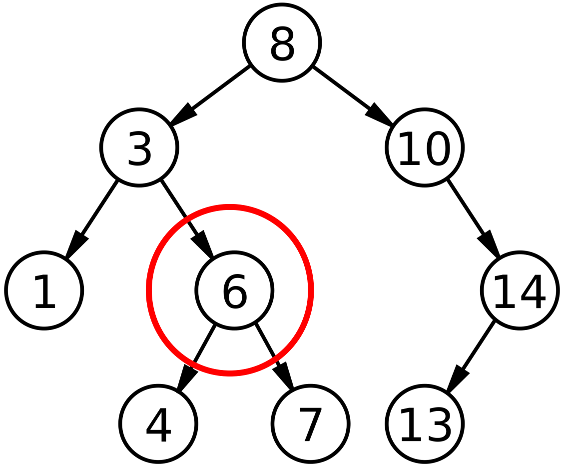
Looking for 7



- 7 != 3
- 3 < 7
- Search right subtree

Binary Search

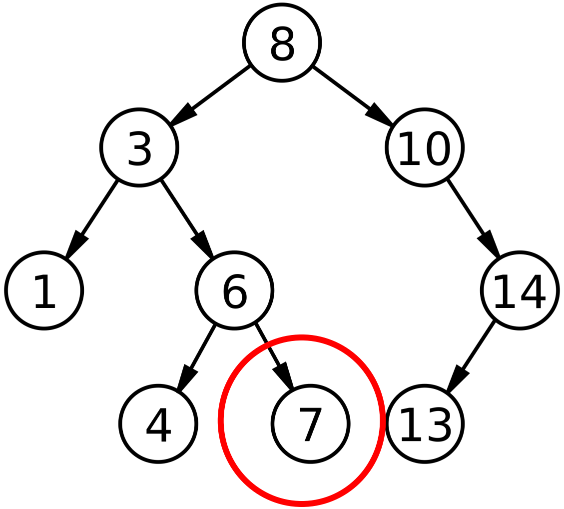
Looking for 7



- 7 != 6
- 6 < 7
- Search right subtree

Binary Search

Looking for 7



- 7 == 7
- Done!

Big-O Teaser

- What sort of ‘guarantee’ do we have with Binary Search?
- How does the runtime change as our input grows?
  - If I had a tree with depth 5, how many total elements do I need to look at to know if it is in the tree?
  - What about a tree with depth 100? Can we generalize this for a tree with depth N?