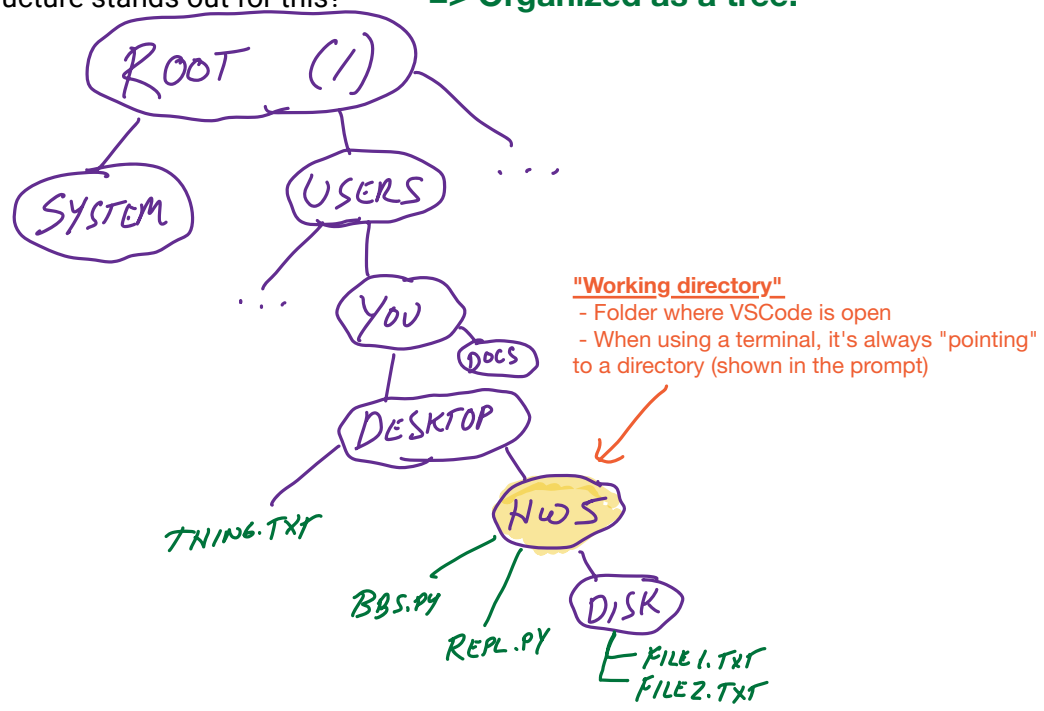
Warmup: A computer's file system organizes files into folders (also called directories): each directory contains a collection of files and other directories. These directories may themselves, also contain files and other directories, and so on …

What data structure stands out for this?

Nested, recursive structure of directories
=> Organized as a tree!



Super important things about files and folders

- Filesystem is organized like a tree. The tree has a root:
  MacOS/Linux: "/" on MacOS, "C:\" on Windows
- All programs run from a certain "working directory" (abbreviated "."), which is some folder in the tree
  => This is the directory in the prompt when you open a terminal!

```
you@your-system:~/Desktop/hw5$ pwd
/Users/you/Desktop/hw5
```

Paths: When we write paths, the path is usually relative to the working directory:

```
open("disk/file1.txt", "r")
```

=> Actual path is "./disk/file1.txt" => start at hw5, go to child directory "disk", then open "file1.txt"

FYI: Working with paths is one of those *super useful CS life skills* that will be super useful beyond CS 200. See page 4 for more notes on this!

Back to CS 200 content...

Today: using data structures to support computation and algorithms

Given: a list of numbers, and a target number

Return the positions (indices) of two numbers in the list that sum to the target (as a set)
Assume that there is exactly one solution

Ex. INDEX 0 1 2
[6, 4, 1], TARGET 7

RETURN {0, 2} SINCE 6 + 1 = 7

HOW COULD WE WRITE THIS (EFFICIENTLY)?

=> FIRST STEP: FIND NAIVE SOLUTION FIRST, THEN TRY TO IMPROVE

```
Input list : [9,4,6,8,5], target 11

If we only touch 9 once, what can we remember about it?

If we see a 9 in position 0,
 => Know:  if we later see a 2, there exists a pair that uses index 0

If we see a 4 in positions 1
 => if we later see a 7, there exists a pair with index 1

What data structure would help us remember this info???
```

IF I SEE 2, USE INDEX 0

IF I SEE 7, USE INDEX 1

=> MAP FROM (NEEDED VALUE) => (INDEX)

Idea #1 (for using data structures to speed up computation):
Ask if you can remember information as you go along, so you can use it later

=> For more notes on how this works, see recording

Idea #2: sometimes it helps to convert one data structure to another

Have: tree
Want: all nodes in tree with value > k => filter operation

=> Know how to filter on lists, but not trees


```
class BinTree:
   left: BinTree
   right: BinTree
   value


def tree_to_list(b: BinTree):
   if b == None:
      return []
   else:
        convert b.right to list
        convert b.left to list
        return (combine left list, right
list, current value)
```

Might be easier if we need to do a lot of list-like operations on this data

Examples: working with paths / Navigating around the tree

- In a path, ".." goes to the parent directory
  open("../thing.txt") => start at hw5, go to parent, open thing.txt
- Important terminal commands:
  - pwd: Show current working directory
  - cd <path>: change working directory (move around the tree!)
  - ls: show files in directory

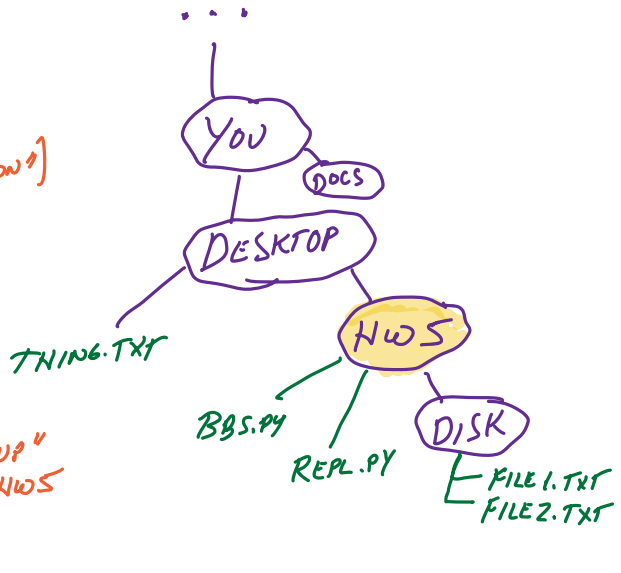
```
you@system:~/Desktop/hw5$ pwd
/Users/you/Desktop/hw5

you@system:~/Desktop/hw5$ ls
disk bbs.py repl.py

you@system:~/Desktop/hw5$ cd disk

you@system:~/Desktop/hw5/disk$ ls
file1.txt file2.txt

you@system:~/Desktop/hw5/disk$ cd ..

you@system:~/Desktop/hw5$ cd ..

you@system:~/Desktop$ ls
hw5 test.txt
```

START HERE

CHANGE TO DISK (MOVE "DOWN")

MOVE "UP" => HW5

MOVE "UP" => DESKTOP



(FYI: We're not going to test you on details about paths or terminal commands, but this is one of those really useful skills to know about working with programs that will help you in any CS course you might encounter in the future!)