Data Structures

l = [7, 3, 4, 6, 9] may not be a heap

heapify(l) makes l a heap

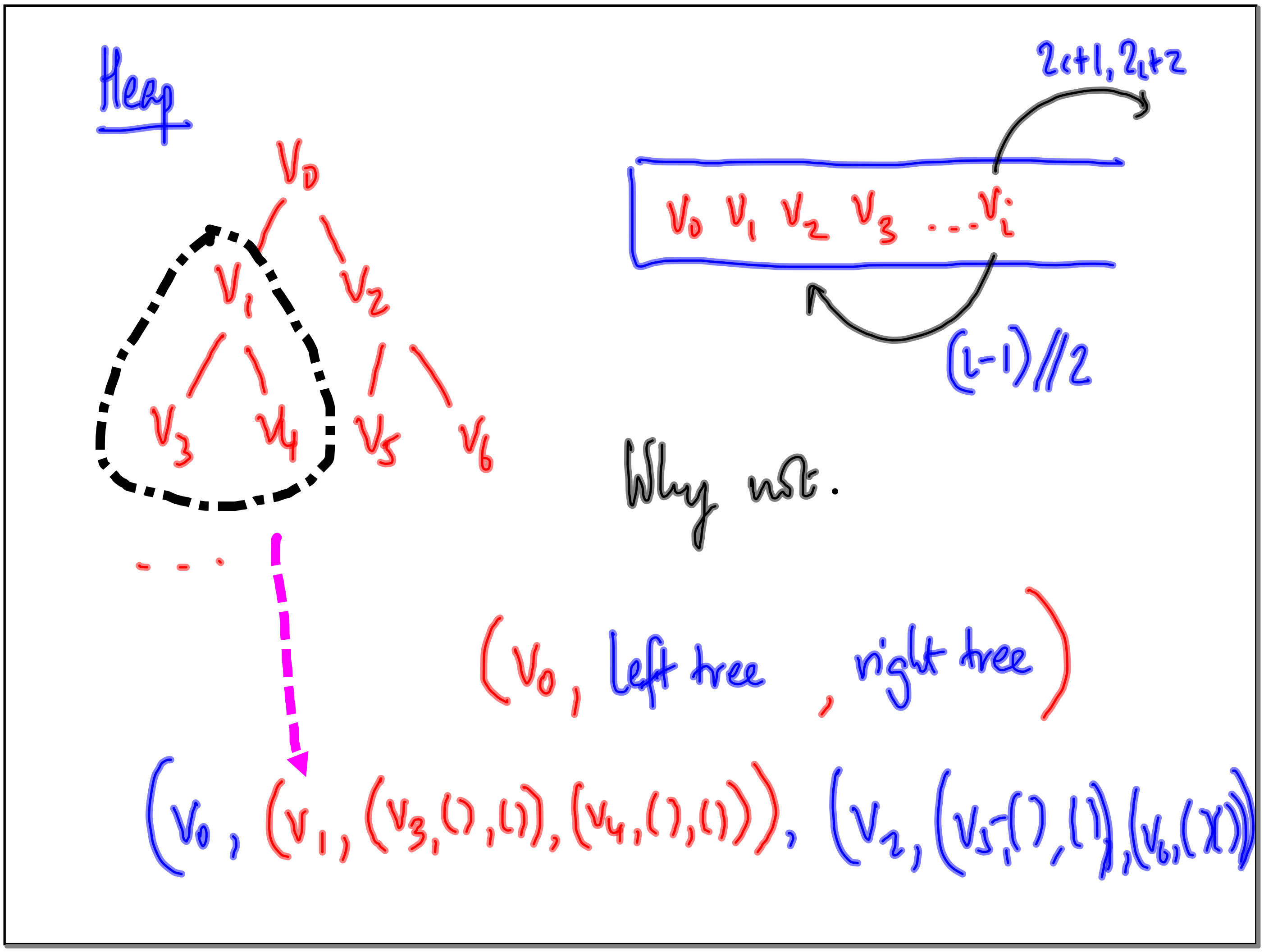
⋮

x = deletemax(l)
insert(l, v)
} assume l is a heap

What about l.append(72)

Want to "enforce" that only heap operations are used on heaps

1. Consistency is lost if other operations intervene
2. Transparency of implementation
   May have many ways to internally represent data


Heap
V0
V1 V2
V3 V4 V5 V6
2i+1, 2i+2
V0 V1 V2 V3 ... Vi
(i-1)//2
Why not.
(V0, left tree , right tree)
(V0, (V1, (V3,(),()), (V4,(),())), (V2, (V5,(),()), (V6,(),())))

Transparency

Preserve functionality independent of concrete implementation

Why enforcement?

Software is written & used by different sets of people

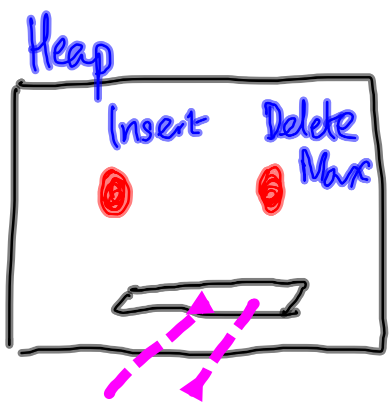
Function-preserving implementation change should not affect existing code

Need a mechanism for this

Data Structure → implementation

- reperesentation for data
- implementation of allowed functions

"Interface"



One solution: "Object Oriented Programming"

Abstract Data Type ADT

Collection of values with a well defined interface

Define a heap as an ADT in the programming language

Lists, dictionaries are built in ADTs

l = [ ]   Give me an empty list object

Till l is redefined, can only perform list operations on l

l = { }   - empty dictionary

Analogously

Q = new_empty_heap

⋮

Only insert, deletemax .. are allowed

May add "isempty", e.g.

Mechanism to

1. Define a "template" that describes how a data structure is implemented

CLASS

2. Make concrete copies of this template

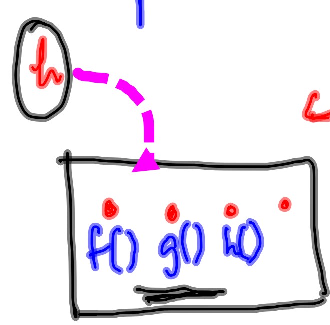
OBJECT

In Python:

class Heap:

data representation and functions that apply to heaps

Should produce



h = Heap()

Sets h to point to a new heap object

Invoking a function on a object:

h.deletemax() vs deletemax(h)

In lists

l.sort() , l.append(v), l.extend(v) || Update l in place

l[2:] || Creates a new object

len(l)

```
class Heap:
    def __init__(self):
        l = []
        self.l = []

    def deletemax(self):
        "manipulate self.l
        and return max value"
            return self.l[0]
            shift around other value to restore heap
```

h = Heap()

calls

refers to object on which manipulation is to happen

```
def insert(self, x):
    Insert x into
    self.l
```

h.insert(7)

The name "self" is (perhaps) not important, but the first argument is <u>always</u> taken to be a reference to the object

Arguments to __init__

```
def __init__(self, startlist):
    self.l = heapify(startlist)
```

h = Heap([3,4,2])

Create a new heap from values [3,4,2]

```
def __init__(self, startlist=[]):
    self.l = heapify(startlist)
```

What if we want to write

h = Heap()

to mean

h = Heap([])

```
def f(x,y):
    def g(a,b,c):
        ⋮
```

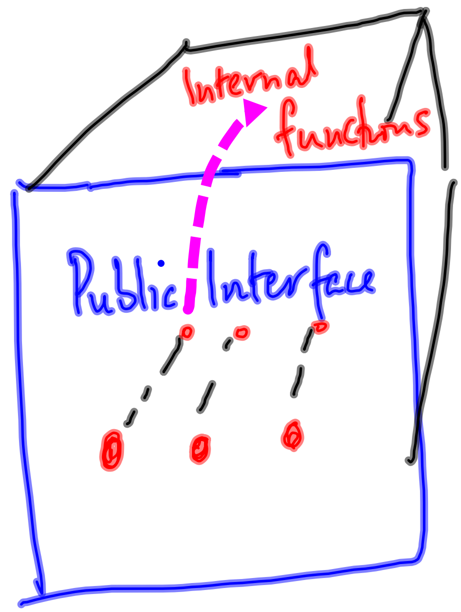
In f, can use g()
but g is not available
outside

Alternatively

```
def __init__(self, ...)

    def heapify(self):

    self.heapify()
```

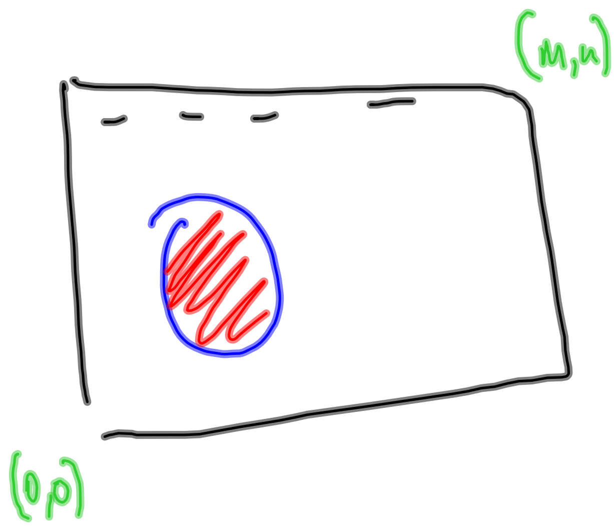


Some ~~other~~ benefits of the OO approach

Ensure consistency of the state at all times

h. Heap([x_1, x_2, .., x_n]) — sets up a heap, henceforth remains a heap

Maintain information about geometrical objects displayed on screen



Circle

Centre: (x, y)

Radius: r

Constraints:

$r > 0$

$0 \le x \le m$

$0 \le y \le n$

```
class Circle:
    def __init__(self, x, y, r):
        ⋮
```

enforce that x, y, r are "sane" values

```
    def translate(self, deltax, deltay):
```

= ensure that x, y are sane

c = Circle(4, 7, 2)

c.translate(7, -3)

x → x+7

y → y-3