3110: Implementation of Functional Data Structures & Imperative Forests

Feedback on Project

- Functional Data Structure implementation
  - Linked lists
  - Red-Black Trees
- Modules

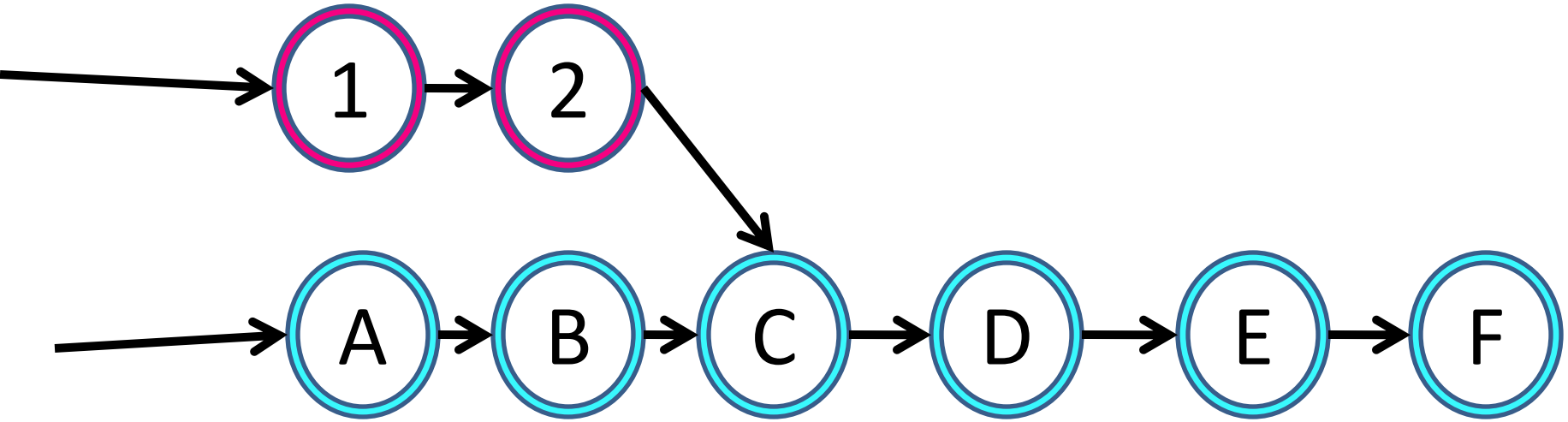
- Imperative storage of Trees
  - List representation
  - Find with path compression
  - Union by rank

Rules of Thumb

- For it to be a Functional Data Structure no destructive assignments. Ie, no side-effects.
- So far we've just copied the entire structure, if most of it is being changed can't do any better. But if the change is small can just point to the original structure.

Functional Linked List

- Given a list, want to change the 2nd element, but for it to be a functional a pointer to the original list must work, compromise:



Efficiency

- Note, it seems to create a lot of overhead, but runtime complexity is the same, O(n), as imperatively searching and destroying.

Benefit

- Can have personalized copies of a larger data structure, with no concern for error

Garbage Collection

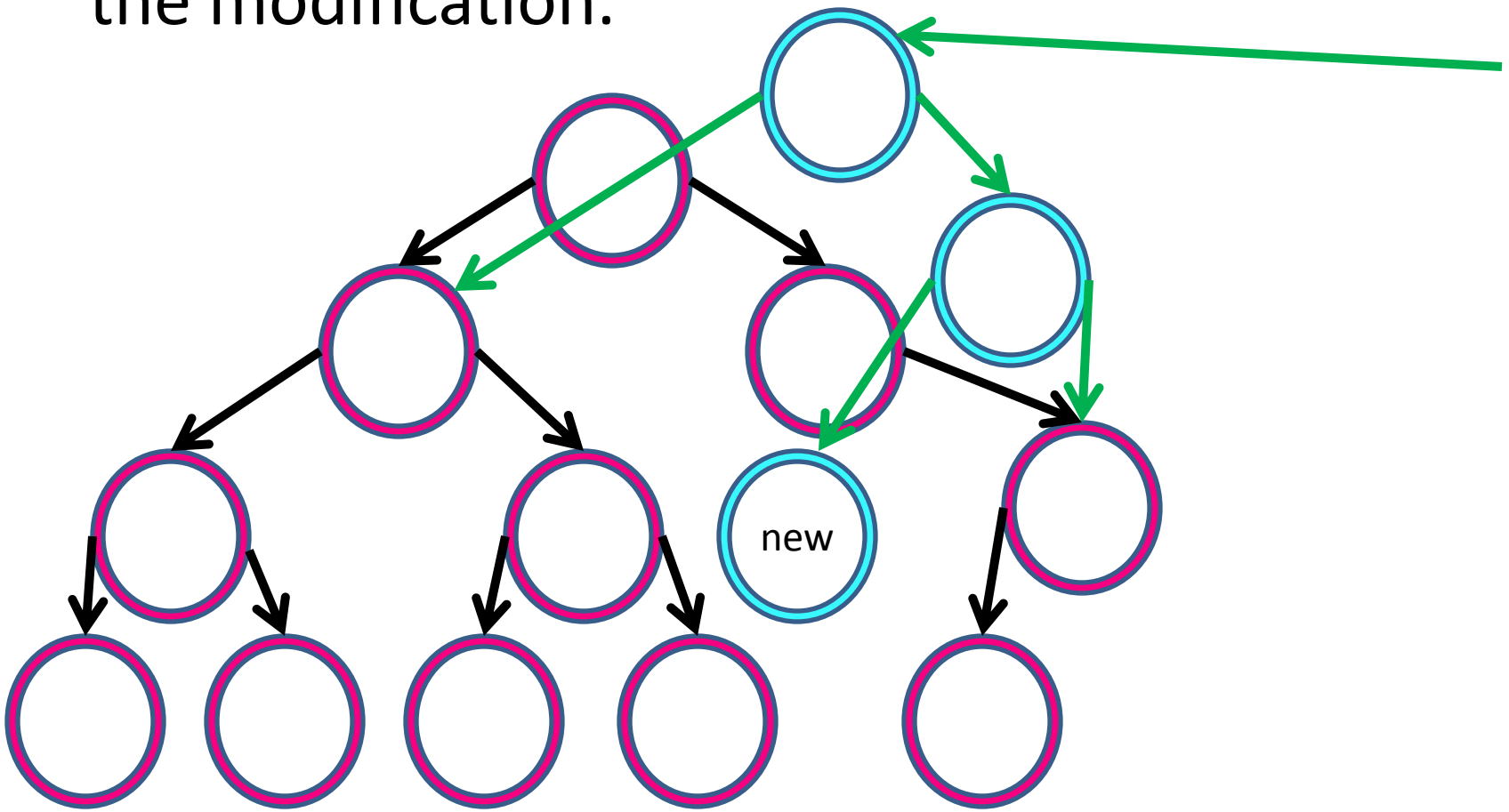
- Performing many of these operations and not needing the ‘persistent’ copies would create a lot of garbage.

- Python has a built in garbage collector. You can clean yourself with the del() function.

- More at the end of the semester.

Functional Binary Trees

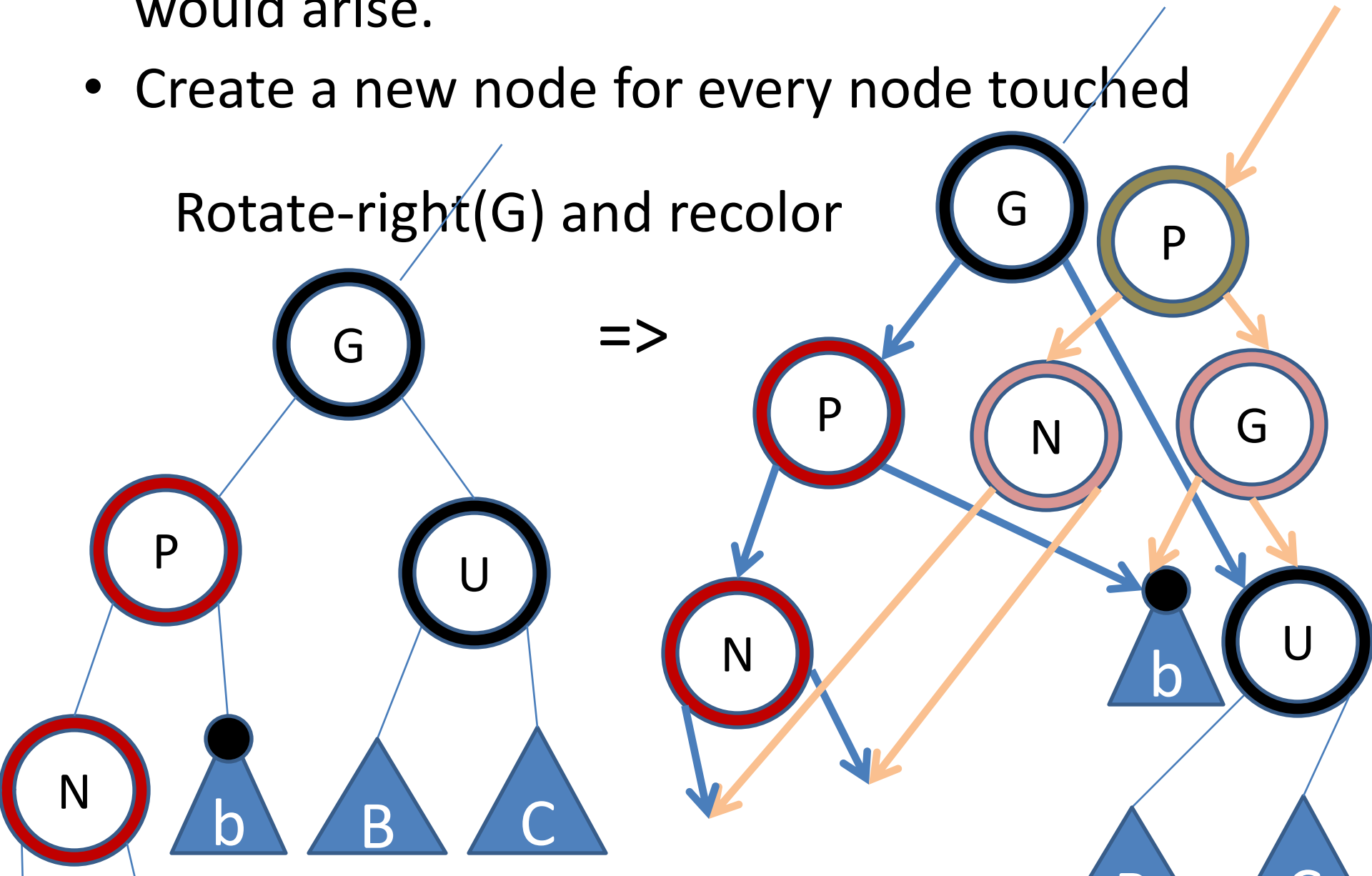
- Very similar - copy any node on the path to the modification.



Functional Red Black Trees

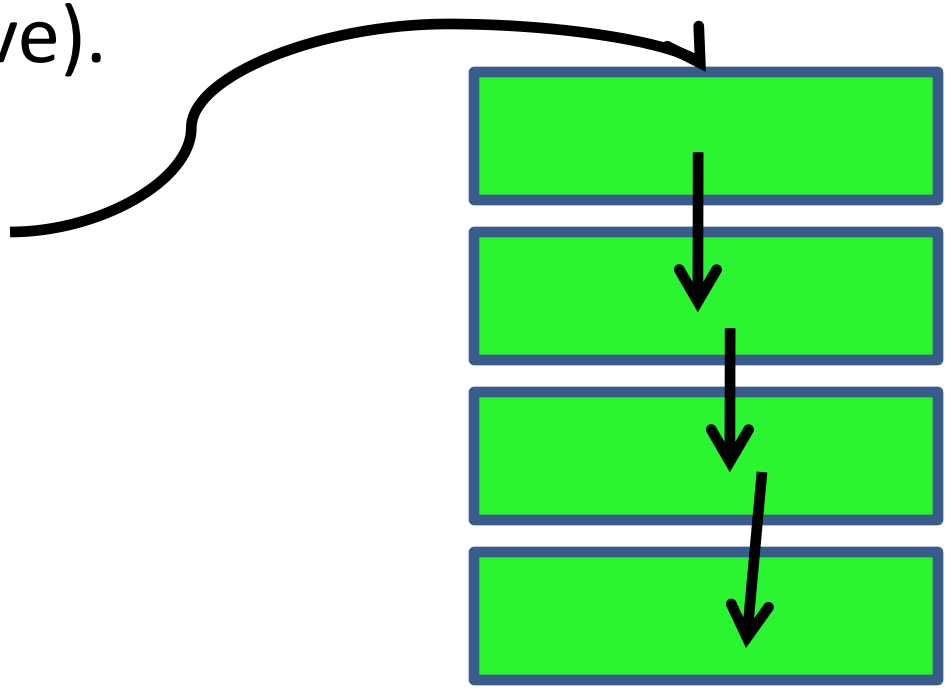
- Can not have pointers to parents. Discrepancy would arise.
- Create a new node for every node touched

Rotate-right(G) and recolor

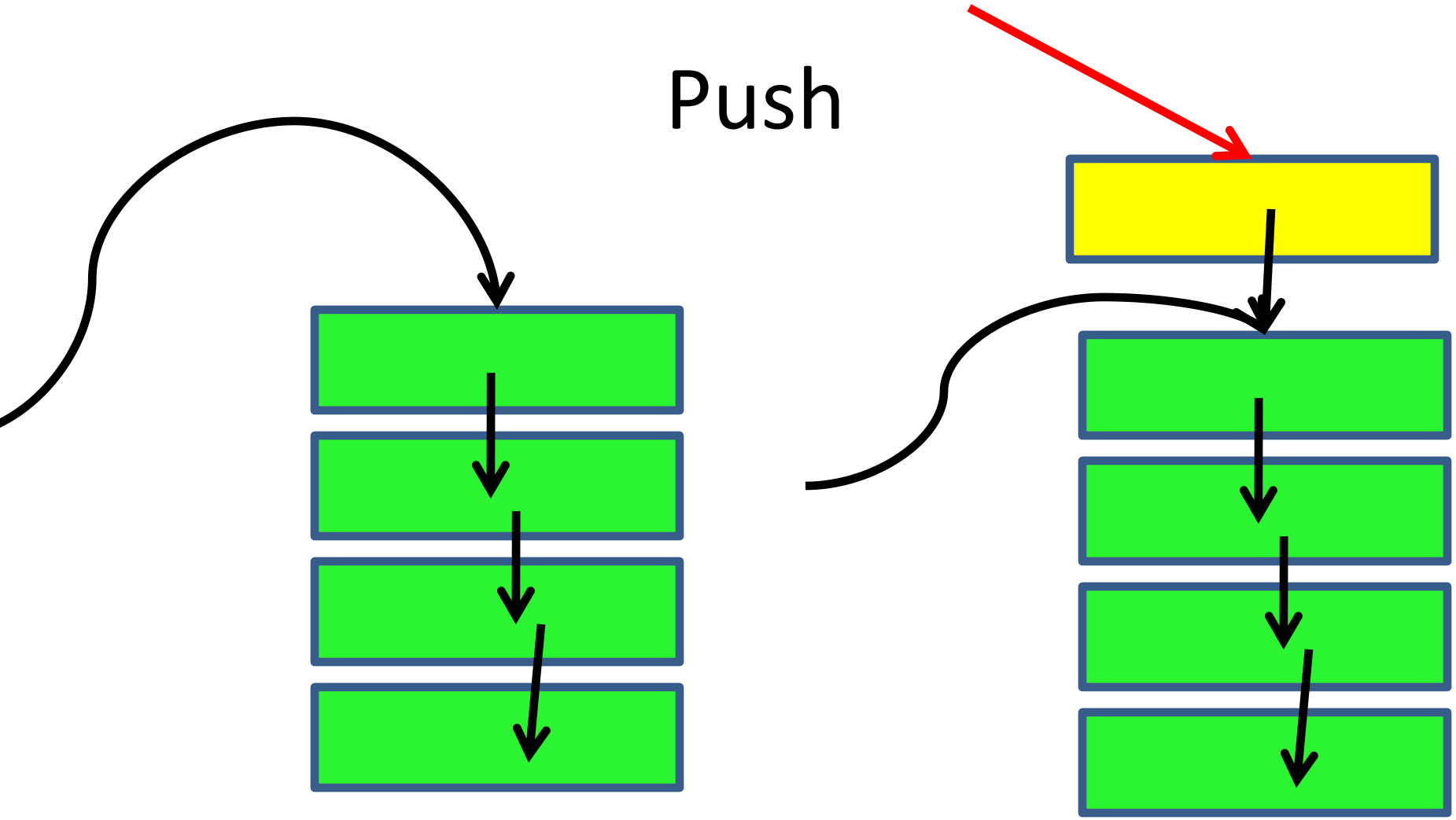


Functional Stacks

- Stacks – first in, last out.
- Implemented as a linked list.
- Generally, just 2 operations, push(add) and pop(remove).

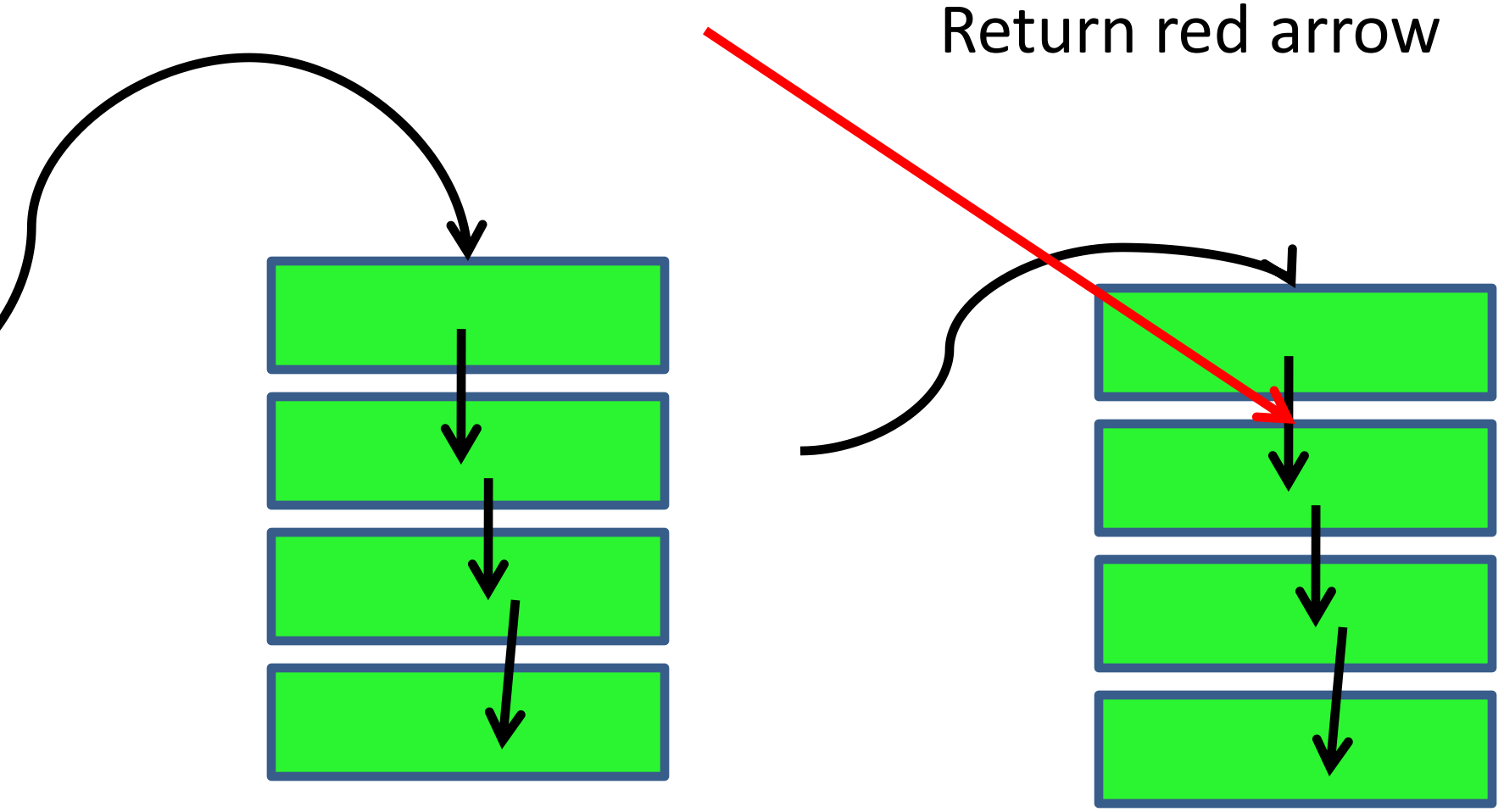


Push



Return red arrow

Pop



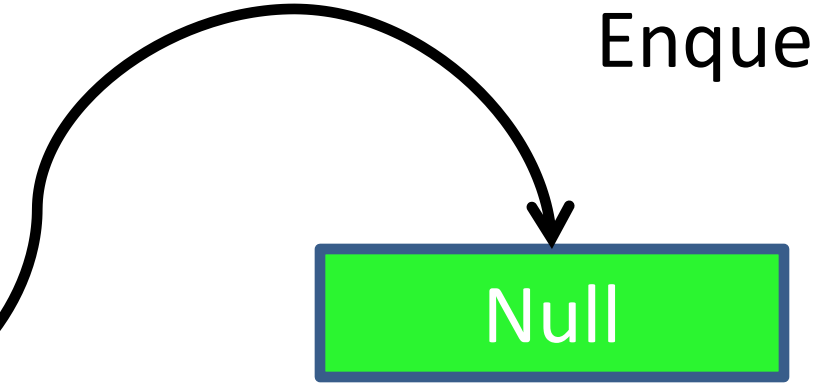
Functional Queues

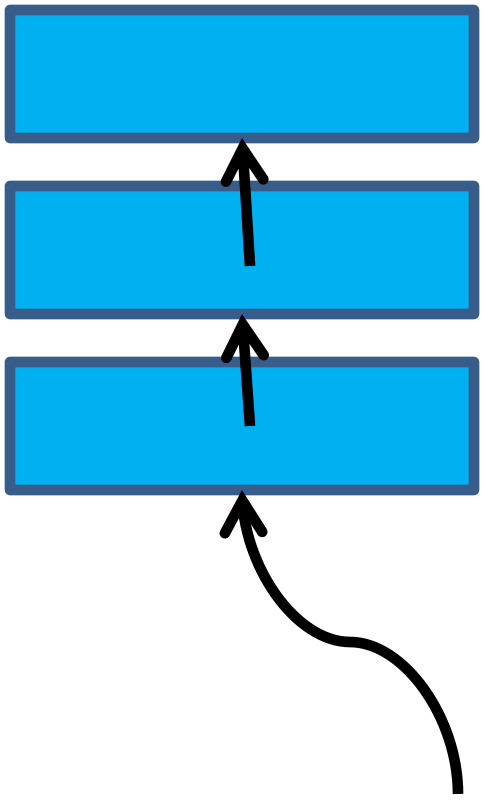
- First in, first out
- A bit trickier, if we just did a linked list, would have to copy the entire queue.
- Use 2 queues, 1 for enqueue, 1 for dequeue.

Start

Enqueue

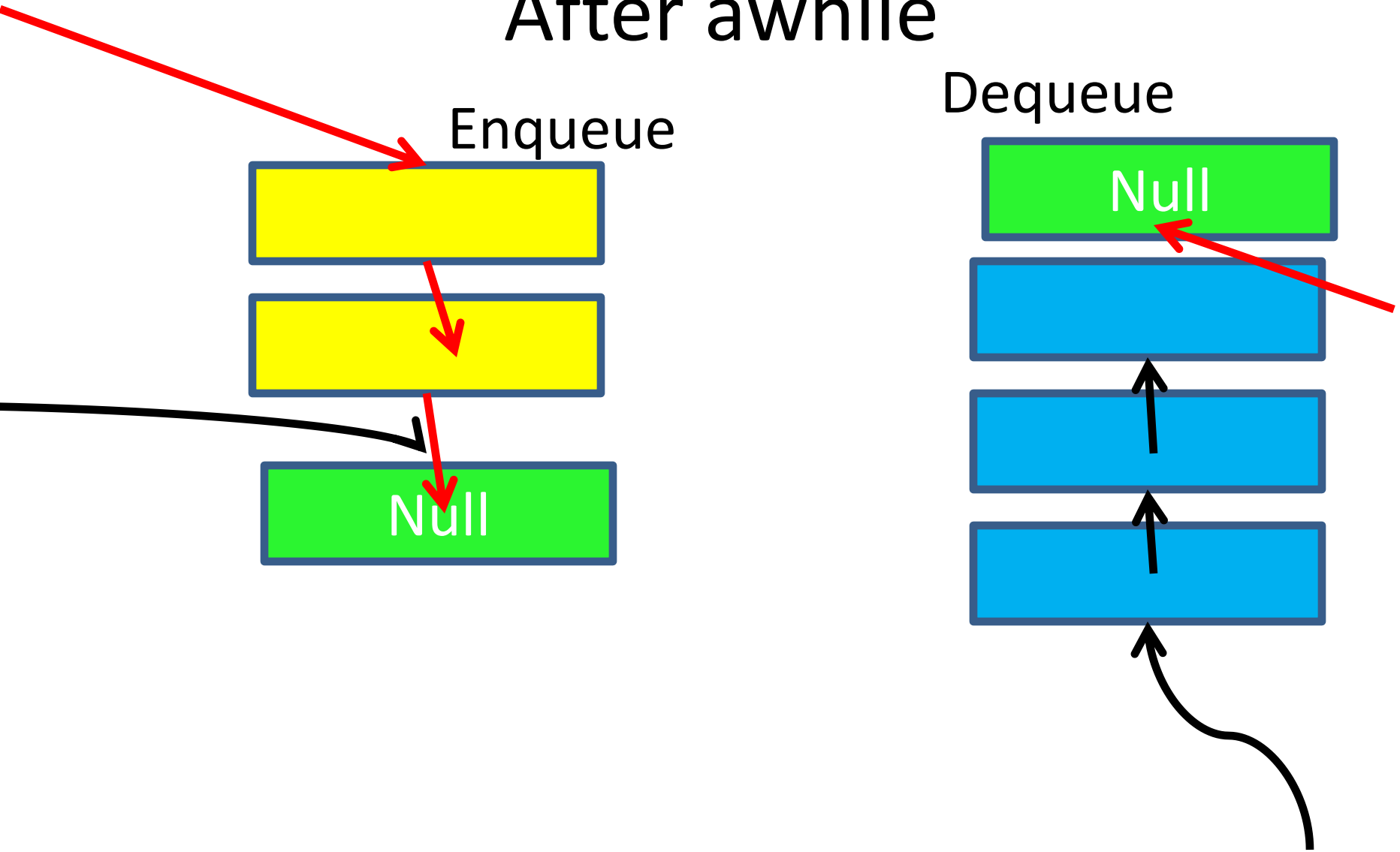
Dequeue





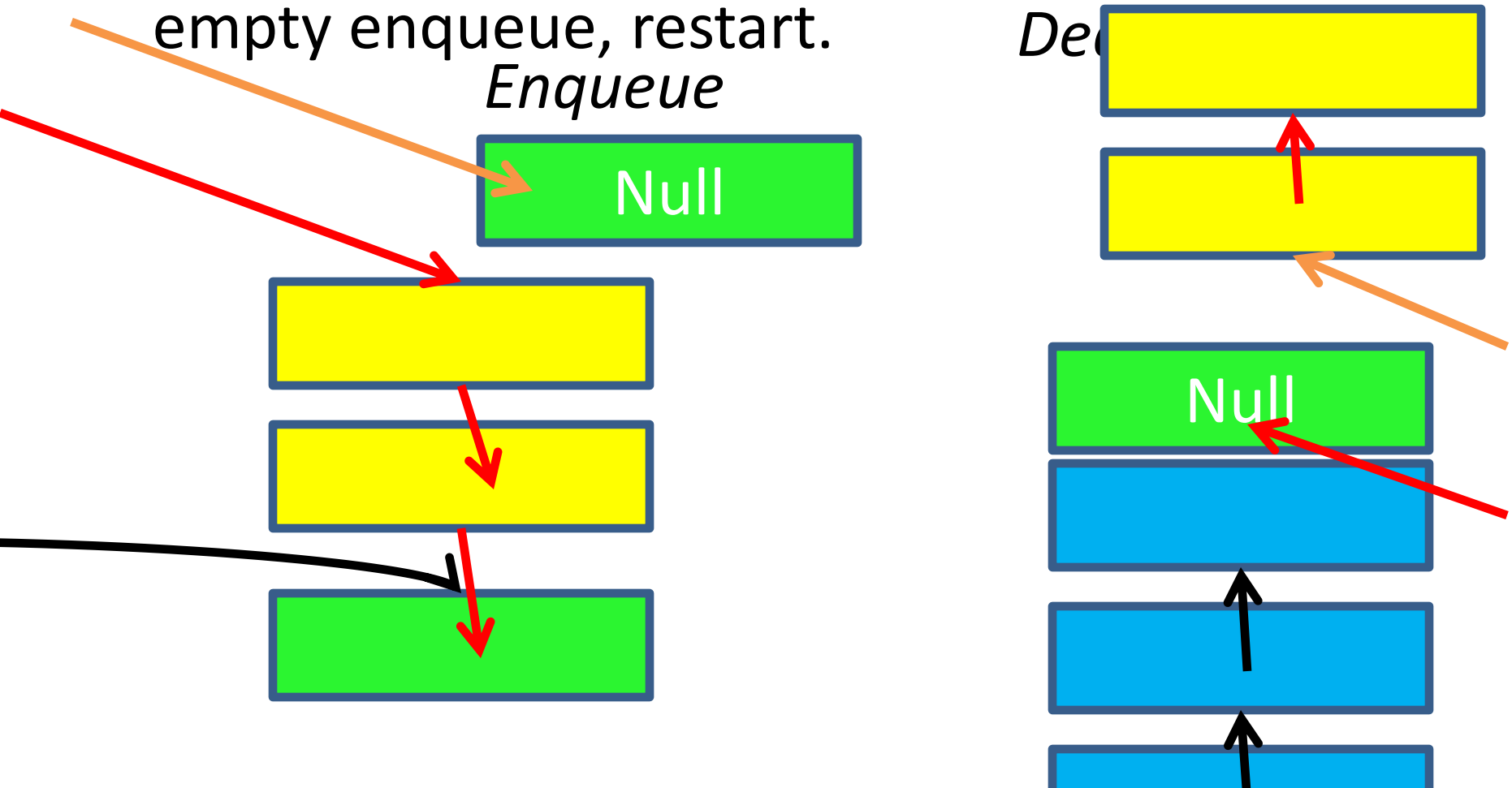
Perform operations as expected of queue

After awhile



Then have to restart

- Reverse copy, enqueue over to dequeue and empty enqueue, restart.



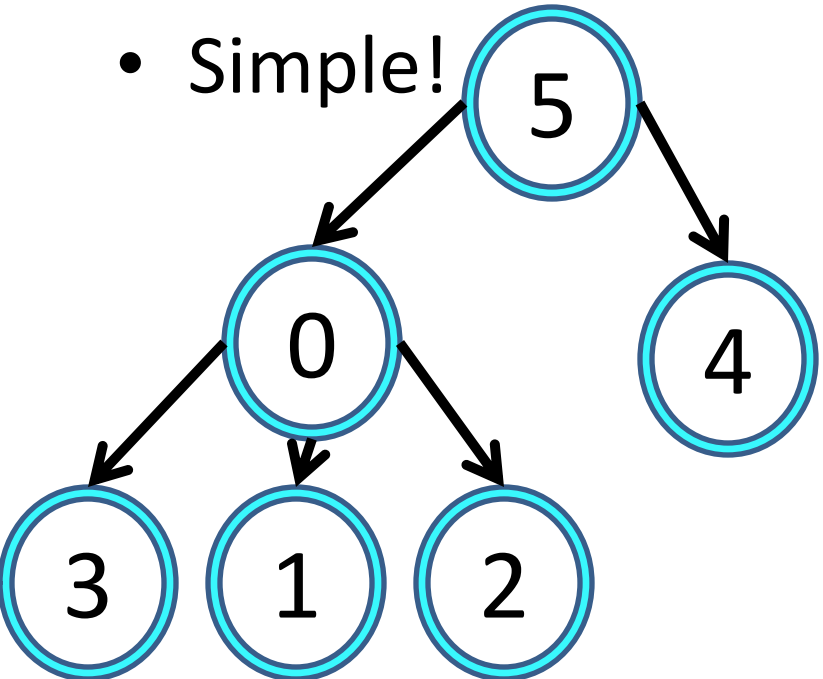
Modules

- You've been thrown in at them.
- What do you think should be required for the interface?
- We'll keep coming back to them.

First Intro to Graphs

Trees to Arrays

- Given a tree with integer valued nodes
  - Create a list as long as the maximum value, tree_list
  - For node(n), set tree_list[n] = parent(n)
- Simple!



=> [5, 0, 0, 0, 5, 5]

Arrays and Sets

- These Arrays are more powerful than just representing trees, they represent sets.
- Consider the root of the tree to be the name of the set.
- To find the set of an element perform a find operation.

Tree-Array Op

- find
- Recurse on the parent listed, until the current node is its own parent, hence it is the root, hence it is the name of the set.

Ex: given an array [0, 0, 5, 2, 4, 0, 3]

find(6) -> find(3) -> find(2) -> find(5) -> find(0)
And returns 0

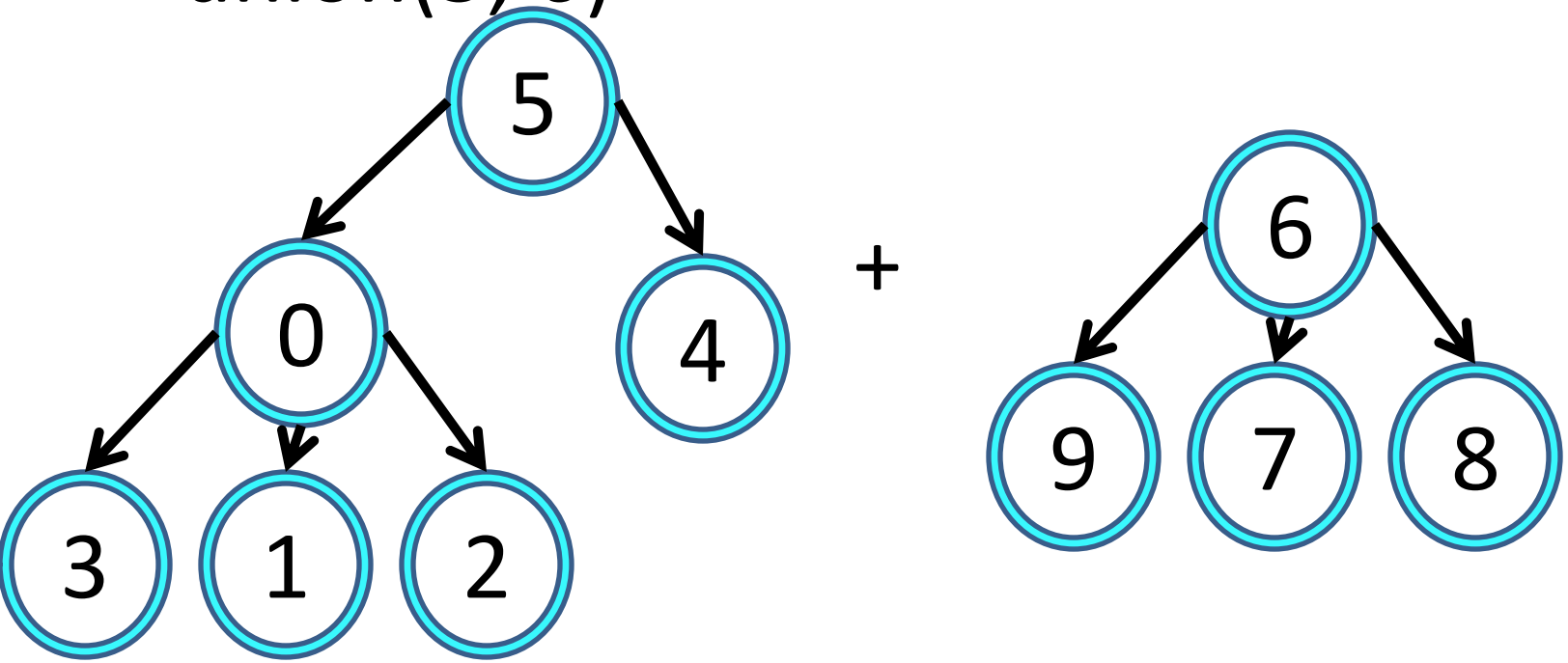
find with path compression

- No extra work, as return down the stack set values to the name of the set, so later finds will be much faster.

Calls:	Returns:	Array:
1. find(6)	1. find(0)	1. [0, 0, 5, 2, 4, 0, 3]
2. find(3)	2. find(5)	2. [0, 0, 5, 2, 4, 0, 3]
3. find(2)	3. find(2)	3. [0, 0, 5, 0, 4, 0, 3]
4. find(5)	4. find(3)	4. [0, 0, 0, 0, 4, 0, 3]
5. find(0)	5. find(6)	5. [0, 0, 0, 0, 4, 0, 0]

Operations on Tree-Arrays

- union(5, 6)



[5, 0, 0, 0, 5, 5, 6, 6, 6, 6] => [5, 0, 0, 0, 5, 5, 5, 6, 6, 6]

Need to make sure anything in the two sets containing 5 and 6 after a find will return the same.

Union

- find the root of each element
- Set one root to be the child of the other

Good union

- Arbitrarily picking 1 root to become the root of the unioned tree can create a list of

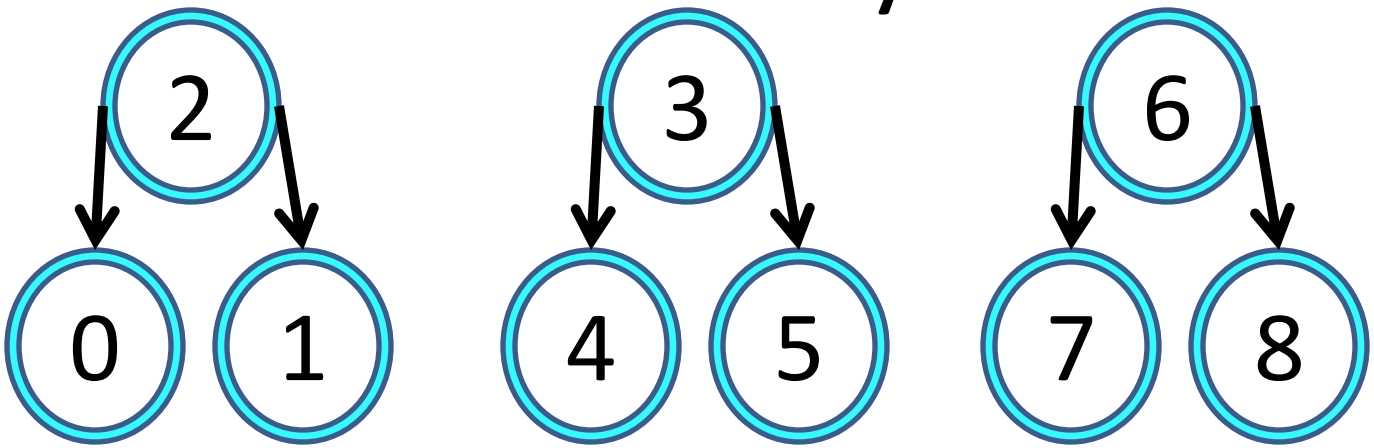
  [0, 1, 2, …] after several unions. This is bad for a find operation
- Want to keep trees as short as possible, as unions call find as well. Runtime of union is:

  O(2*O(find) + 1)

Union by Rank

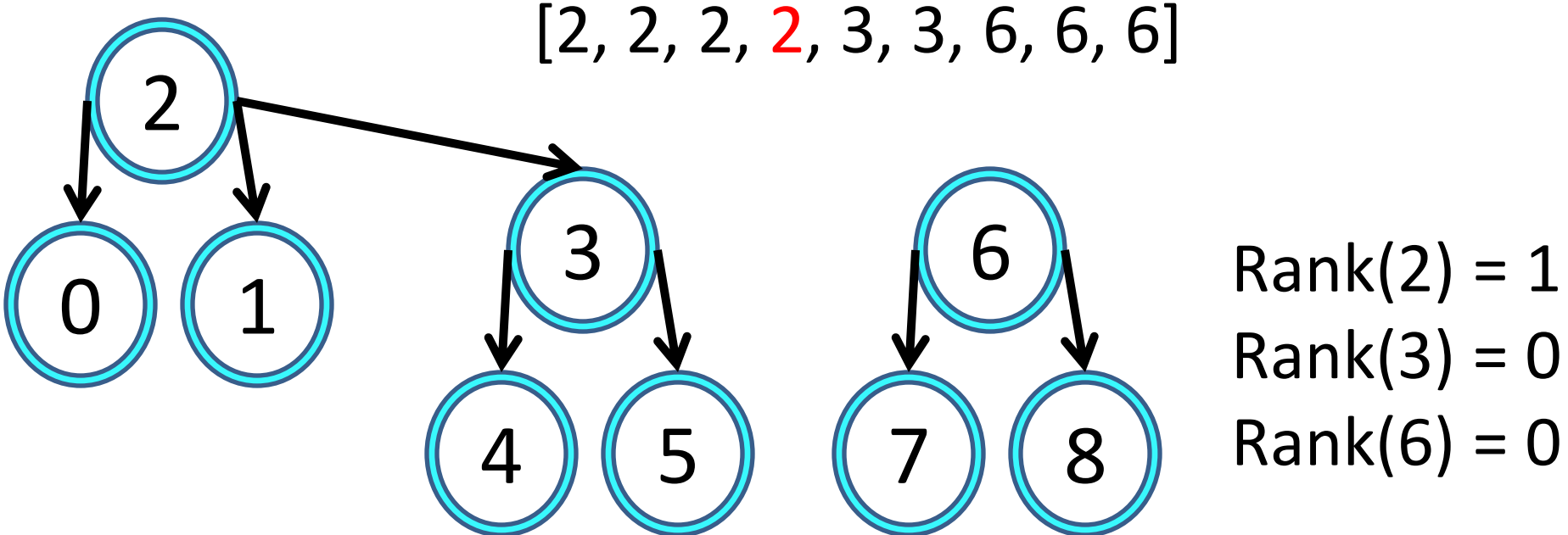
- Need to keep track of another parameter, if using find with path compression, hard to keep track of depth. Next best guess, rank.
- Rank is the number of time a root is unioned with another root of the same rank.
- Union(root1, root2), make the root with larger rank the new root.

Union by rank ex.

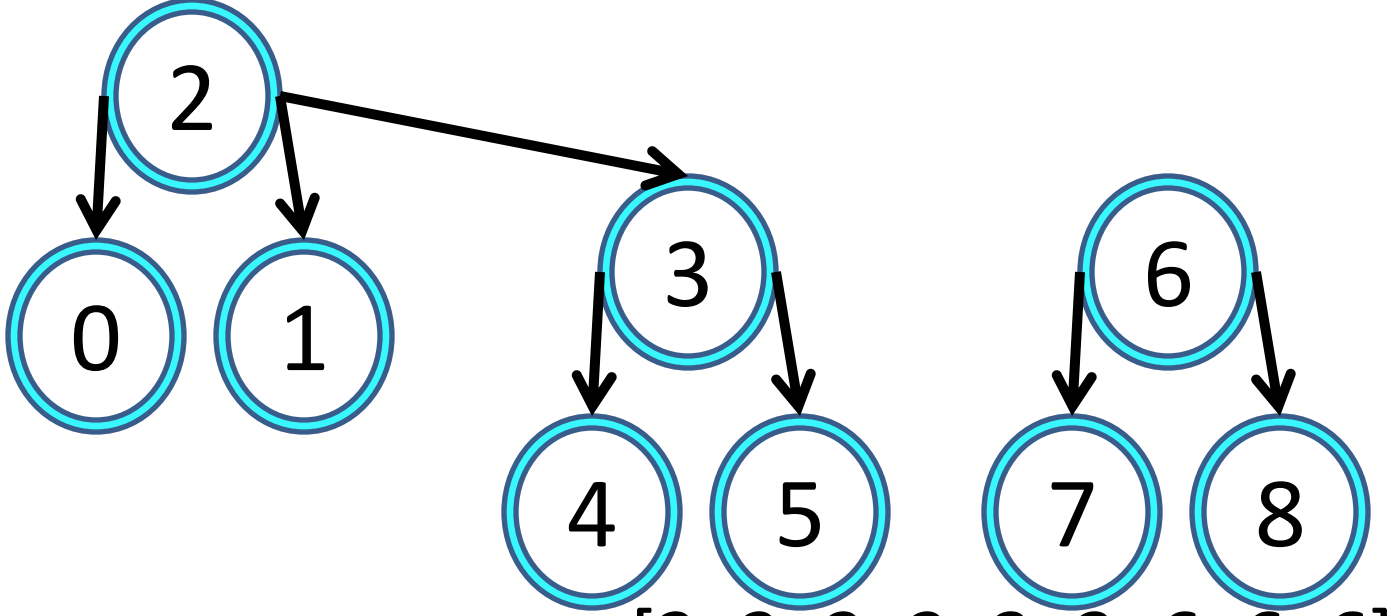


Rank(2) = 0
Rank(3) = 0
Rank(6) = 0

[2, 2, 2, 3, 3, 3, 6, 6, 6]
union(0, 4)
[2, 2, 2, 2, 3, 3, 6, 6, 6]

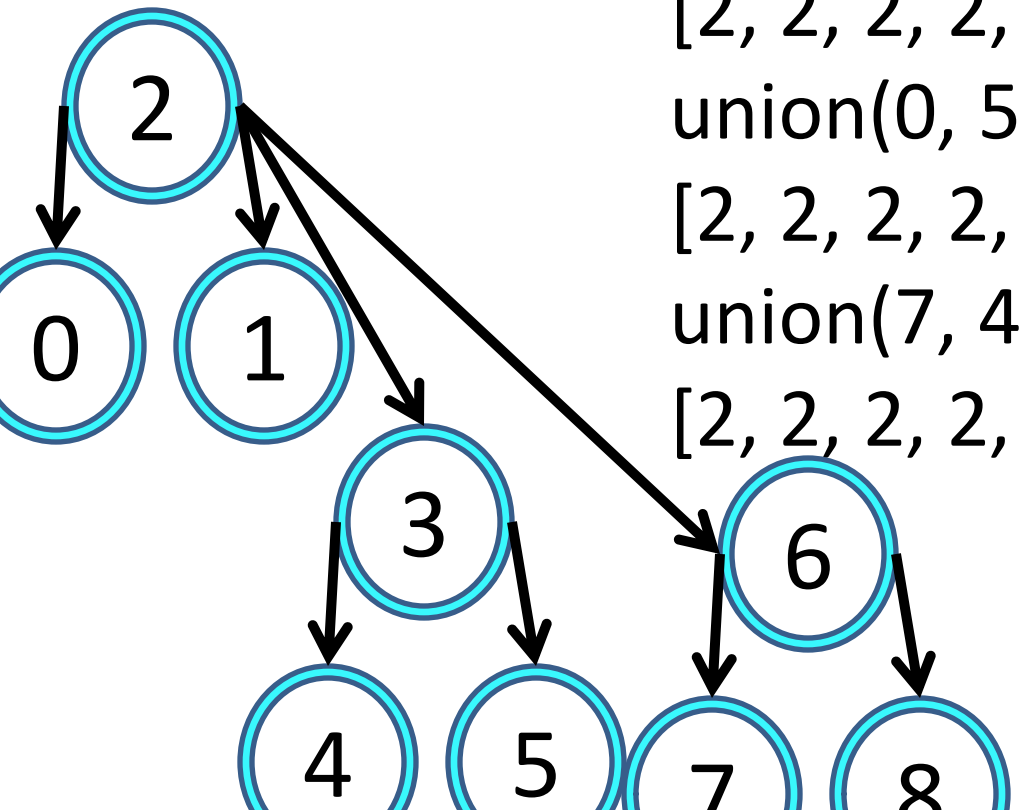


Rank(2) = 1
Rank(3) = 0
Rank(6) = 0



Rank(2) = 1
Rank(3) = 0
Rank(6) = 0

[2, 2, 2, 2, 3, 3, 6, 6, 6]
union(0, 5)
[2, 2, 2, 2, 3, 2, 6, 6, 6]
union(7, 4)
[2, 2, 2, 2, 2, 2, 2, 6, 6]



Rank(2) = 1
Rank(3) = 0
Rank(6) = 0

Plug for analysis

- Really cool function:

  log*(n)

The number of logs that need to be applied before the result is less than 1.

Ie $\log(\log(4)) < 1 \Rightarrow \log^*(4) = 2$

In practice $\log^*(n) \leq 5$

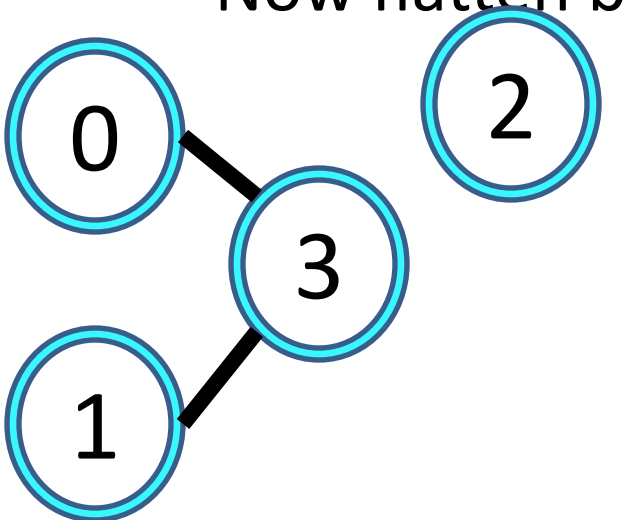
Analysis

- Given a forest(disjoint set array) with n elements, performing m union operations with path compression and union by rank. The time to do so is:

$$O((m+n)\log^*(n))$$

Finding the Connected Components

- Given a graph
  - Create a list with each node marked as its own parent ie [0, 1, 2, …]
  - For each edge in the graph call union on the two connected nodes
  - Now flatten by calling find on each node



[0, 1, 2, 3] (create)
[0, 1, 2, 0] (consider edge(0, 1))
[0, 3, 2, 0] (consider edge(1, 3))
[0, 0, 2, 0] (flatten)