# Python: Taking the GARBAGE out


**Pablo Galindo**
*@pyblogsal*
Bloomberg

**Víctor Terrón**
*@pyctor*
Google

# 1.

## What’s garbage collection?

What happens when we **delete** a variable?

```
>>> x = [2, 3, 5]
>>> x
[2, 3, 5]
>>> del x
```

Where did you go?

… or we no **longer** need it?

```python
def factorize(number):

  primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
  factors = []

  # do stuff

  return factors
```

‘primes’ is defined inside of our function, and once it goes **out of scope** we no longer need it.

**What** happens to it?

From the Python glossary:

"The process of **freeing memory** when it is not used anymore [...]"

# garbage

The memory occupied by objects that are no longer in use by the program.

# garbage collection

A form of **automatic** memory management.

# why

Because otherwise we'll eventually **run out** of memory.

# automatic

- We don’t have to do it ourselves!
- Less **thinking** → fewer **bugs** (e.g. dangling pointer)

However: can be **slower** than manual if not done properly.

Nowadays, almost nobody does it

# manual

- Classic example: Fortran.
- Modern example: Rust.
- C / C++ were designed for use with manual memory management, but have garbage-collected implementations available.

http://wiki.c2.com/?LanguagesWithoutGarbageCollection

# 2.

## Destructors

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

lasie = Dog(“Lasie”, 18)
# do stuff
del lasie
```

Frees up the memory.... right?

From the Python docs:

“It is **not guaranteed** that `__del__()` methods are called for objects that still exist when the interpreter exits”.

# why not

- Because the purpose of __del__() is **only** to free up memory when it's no longer necessary.
- It's **not designed** to execute *cleanup* code (e.g. commit changes to a database… or close a file).
- For that, we use the 'with' statement.

# 3.

## Reference counting

**Reference counting** is a

# strategy

- For each object, keep a counter of the number of references to it.
- **As soon** as the counter reaches zero the object becomes inaccessible…
- … and can be destroyed (i.e. garbage collected)

```python
x = [1, 2, 3]
y = x
```

First reference, ‘x’

Second reference, ‘y’

There are **two** references to the same list.

I want to

# try it

- For that we use **sys.getrefcount()**
- Returns the reference count of the object.
- However, from [the docs](): “The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to getrefcount()”.
- Passing as an argument **increases** the refcount.

```python
>>> import sys
>>> x = [1, 2, 3]
>>> sys.getrefcount(x)
2
>>> y = x
>>> sys.getrefcount(x)
3
>>> del x
>>> sys.getrefcount(y)
2
```

A **container** (e.g. list or set) claims

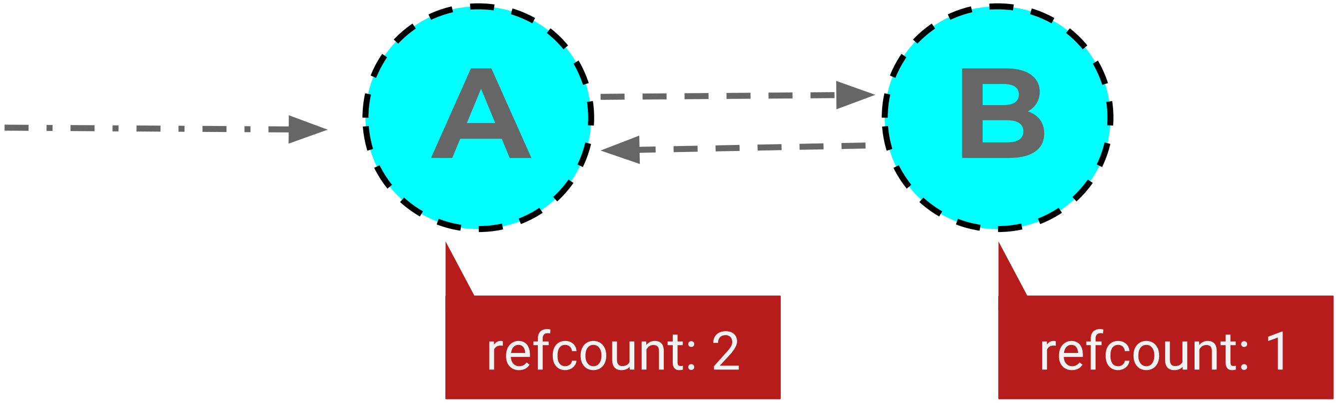
# ownership

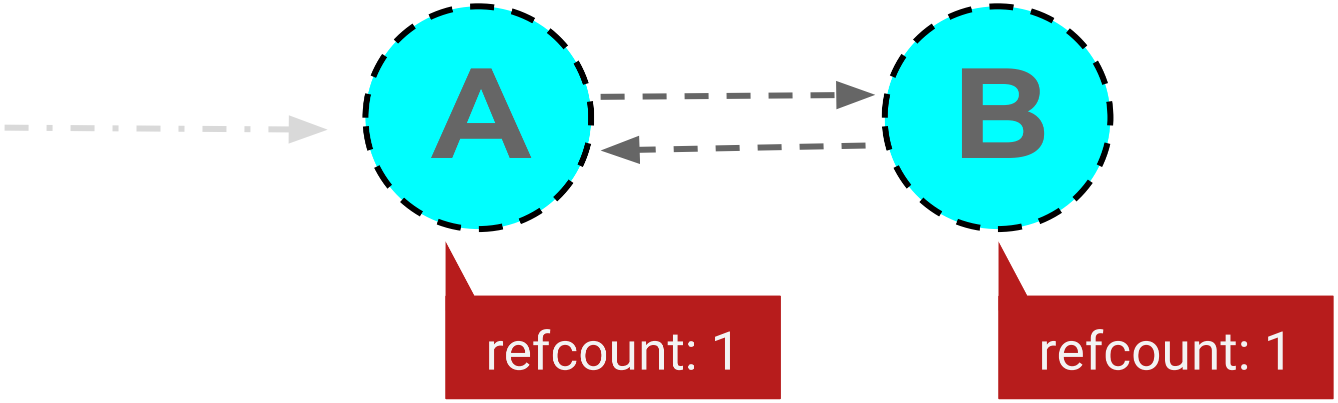
- … over all its elements.
- This **increments** their reference count by one.
- When the container is destroyed, the reference count decreases by one.

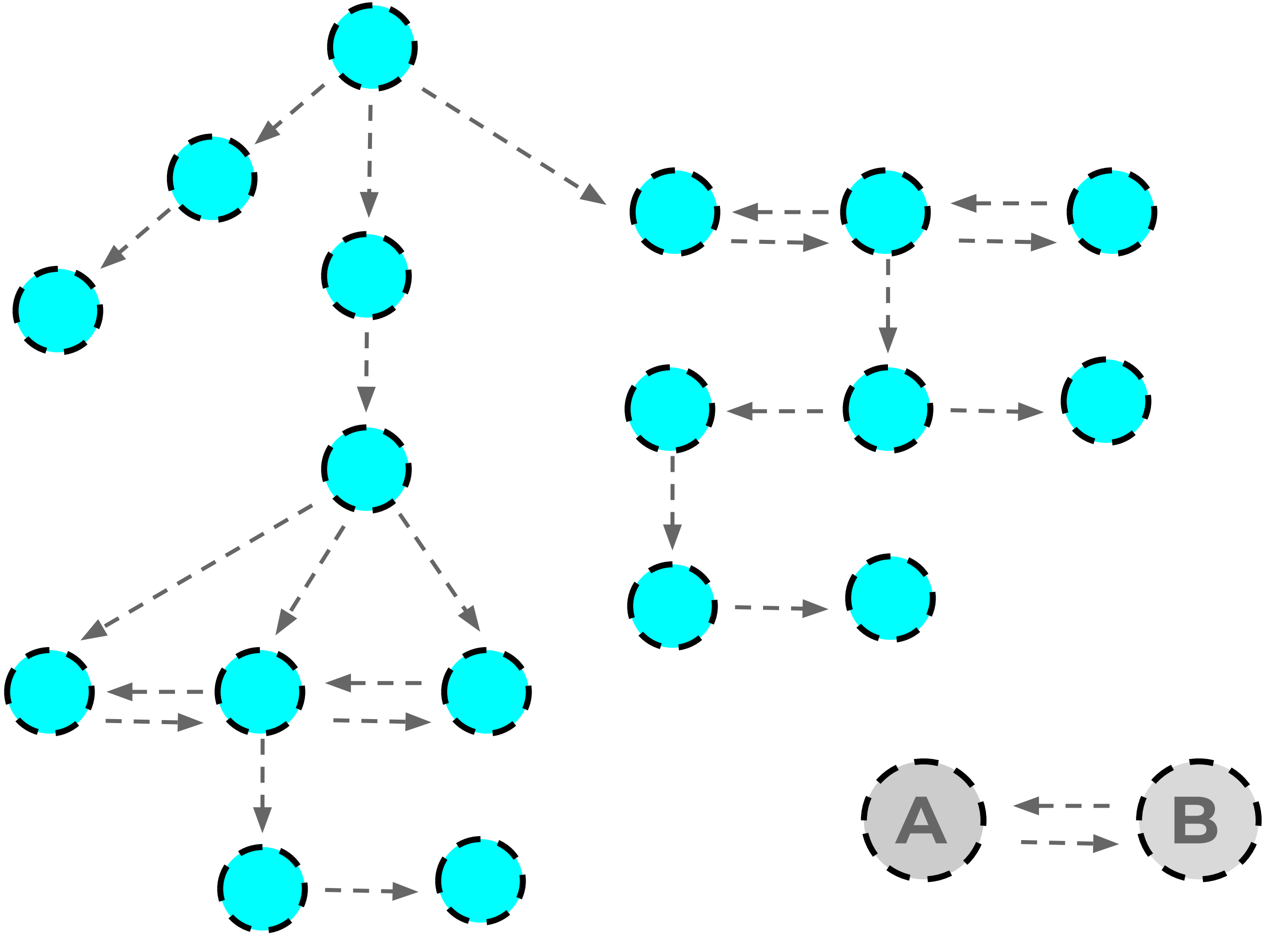
```
>>> x = "one"
>>> y = "two"
>>> sys.getrefcount(x)
2
>>> numbers = set([x, y])
>>> sys.getrefcount(x)
3
>>> sys.getrefcount(y)
3
>>> del numbers
>>> sys.getrefcount(x)
2
>>> sys.getrefcount(y)
2
```

# 4.

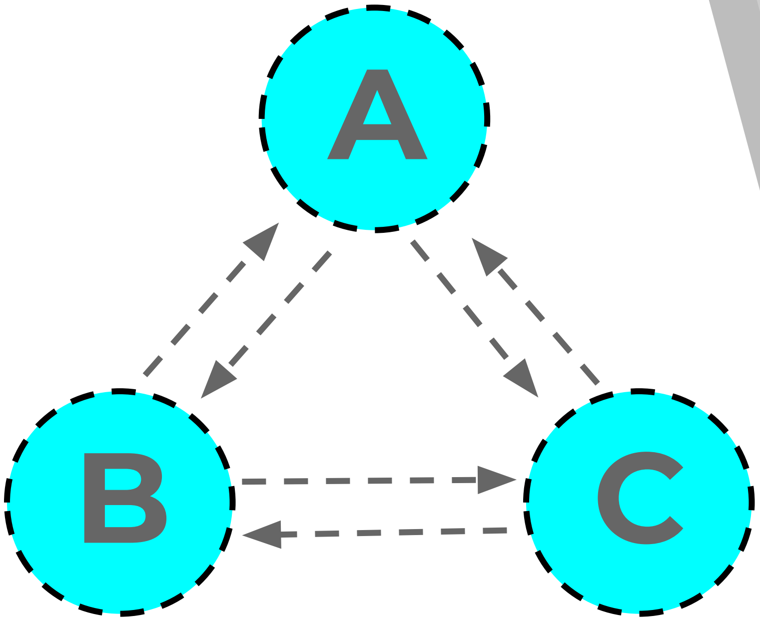
## Cycles

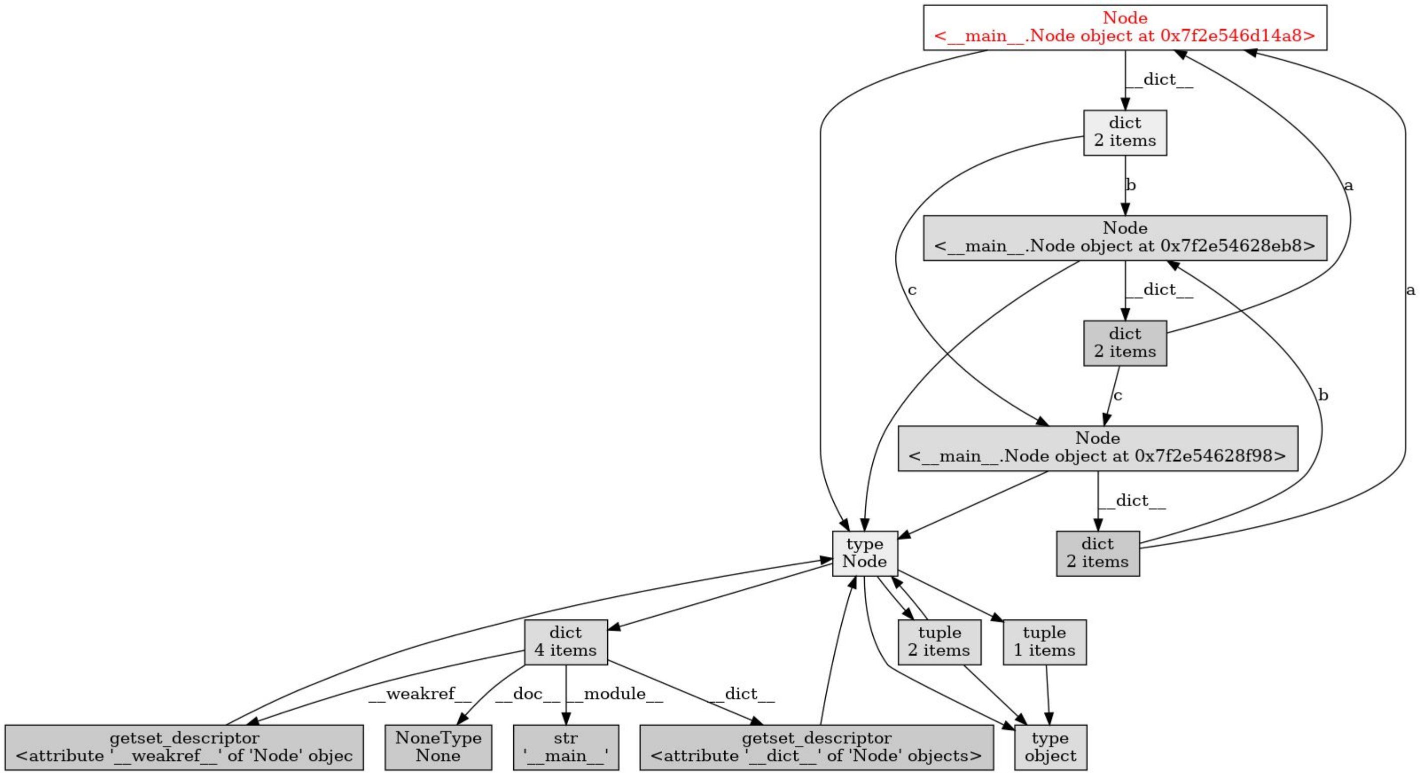

A
B
refcount: 2
refcount: 1


A
B
refcount: 1
refcount: 1


A
B

```
>>> class Node:
...     pass
...
>>> a = Node()
>>> b = Node()
>>> c = Node()
>>> a.b = b
>>> a.c = c
>>> c.a = a
>>> c.b = b
>>> b.a = a
>>> b.c = c
```





Node
<__main__.Node object at 0x7f2e546d14a8>
__dict__
dict
2 items
b
a
c
Node
<__main__.Node object at 0x7f2e54628eb8>
__dict__
dict
2 items
a
c
b
Node
<__main__.Node object at 0x7f2e54628f98>
__dict__
type
Node
dict
2 items
dict
4 items
tuple
2 items
tuple
1 items
__weakref__
__doc__
__module__
__dict__
getset_descriptor
<attribute '__weakref__' of 'Node' objec
NoneType
None
str
'__main__'
getset_descriptor
<attribute '__dict__' of 'Node' objects>
type
object

# 5.

# Generational Garbage Collection

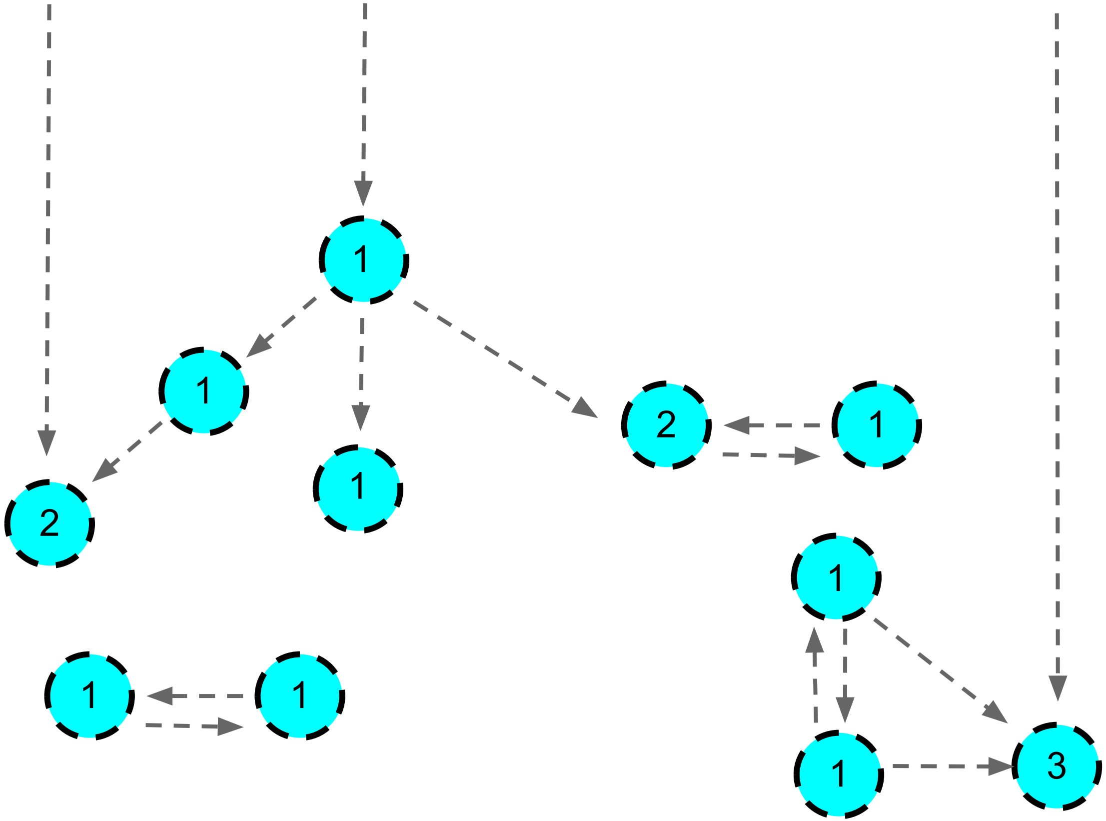
The **garbage collector** algorithm will remove

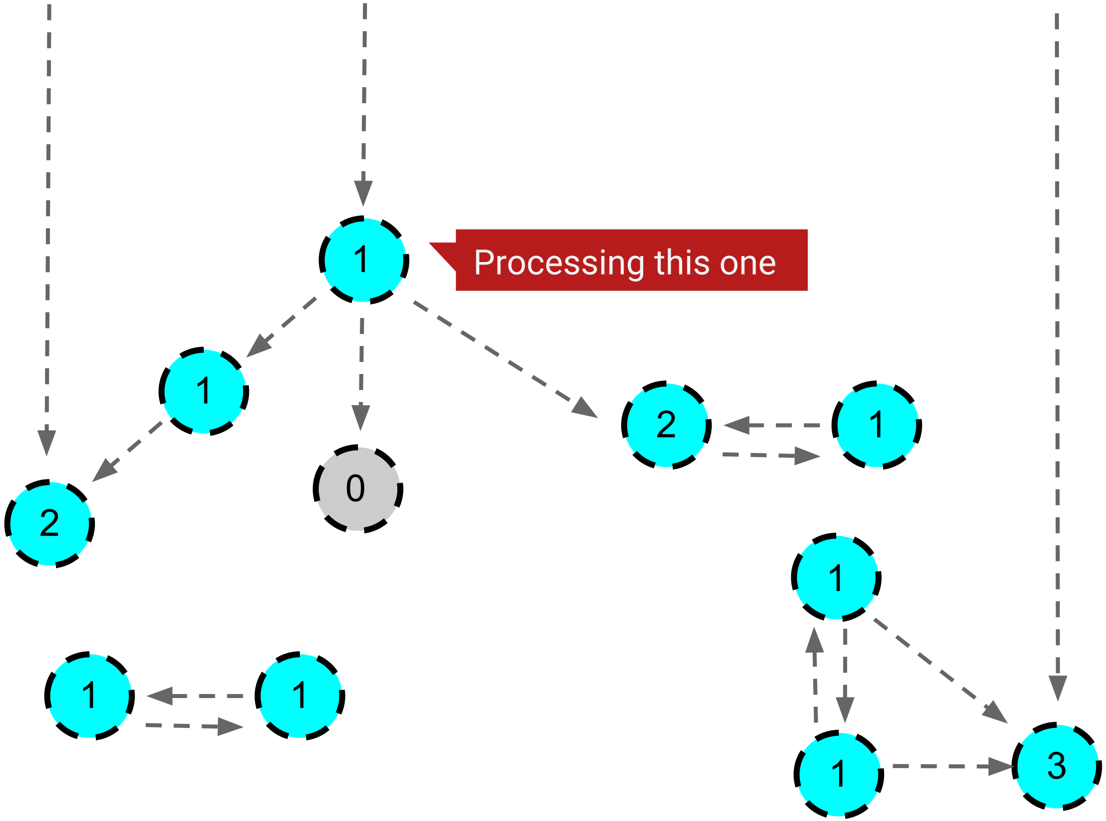
# cicles

- … that are unreachable from “outside”.
- This process **stops** the running program until is complete.

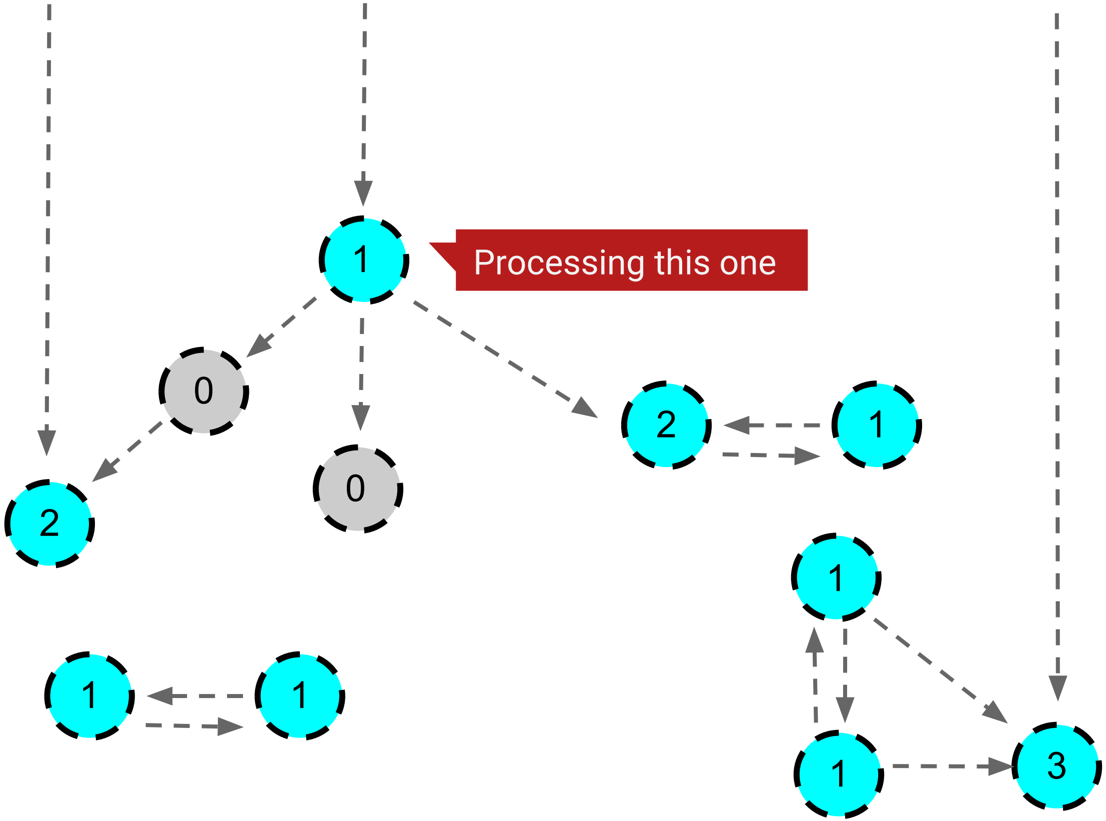
# STEP 1

1. **Decrement** references.
2. **Mark** objects with 0 references as "*maybe unreachable*"

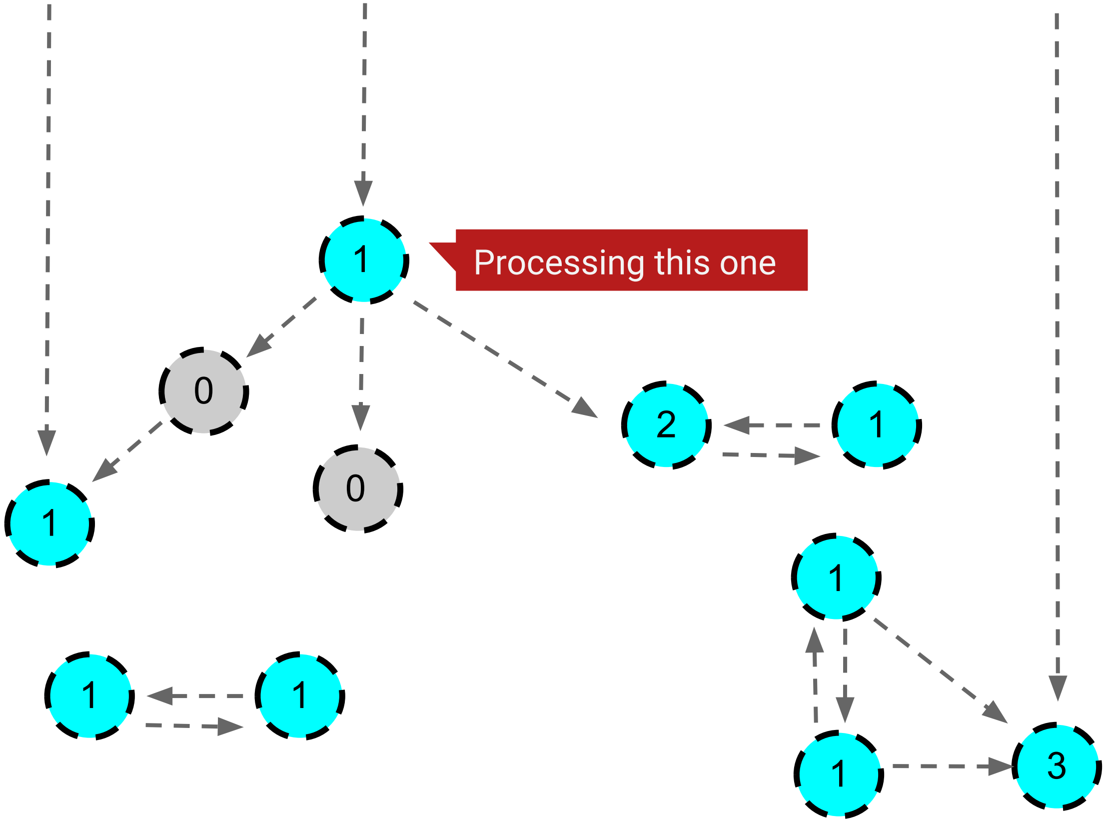

1
1
1
2
2
1
2
1
1
1
1
3

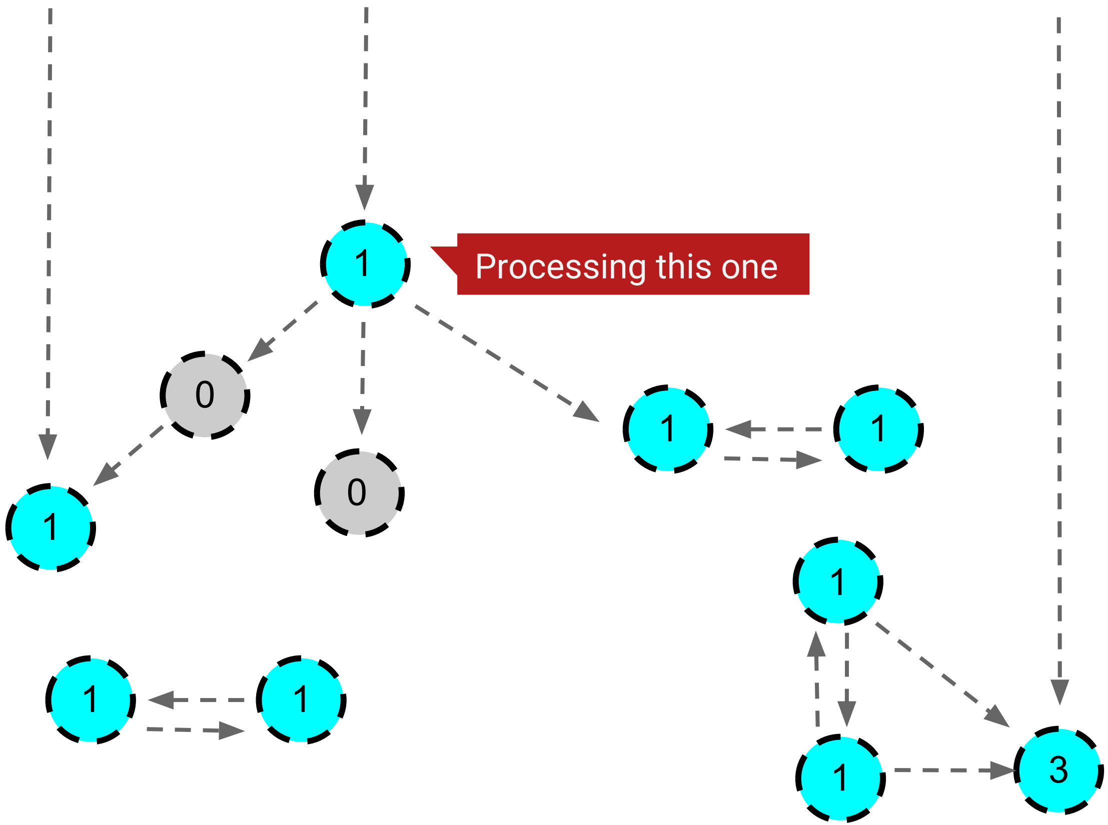


Processing this one
1
1
0
2
2
1
1
1
1
1
3

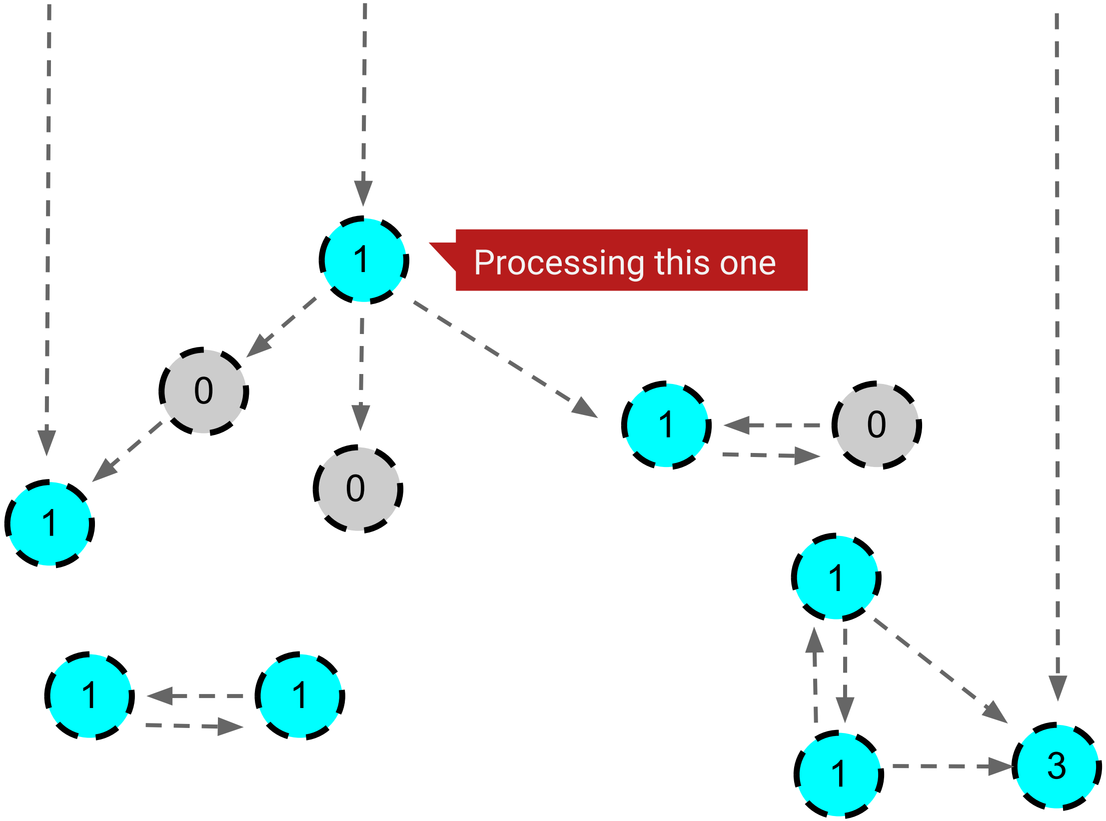


1
Processing this one
0
0
2
2
1
1
1
1
1
3

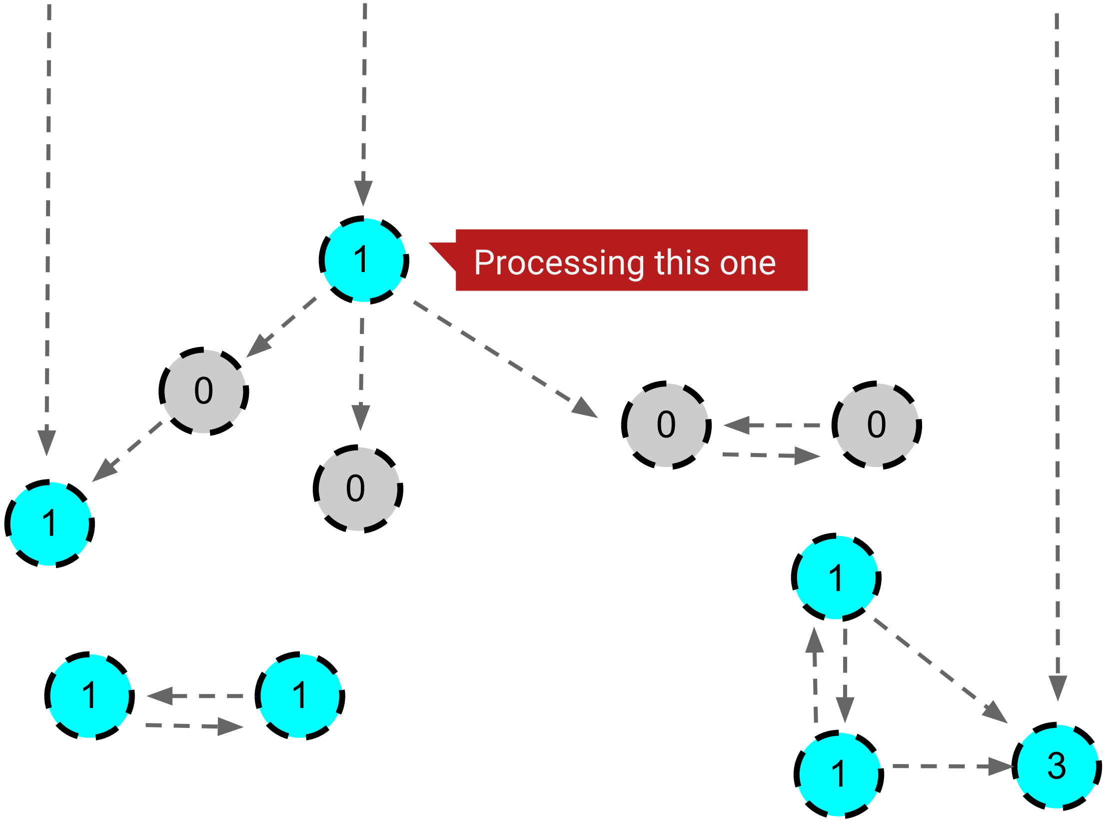

1
Processing this one
0
0
2
1
1
1
1
1
1
3
1

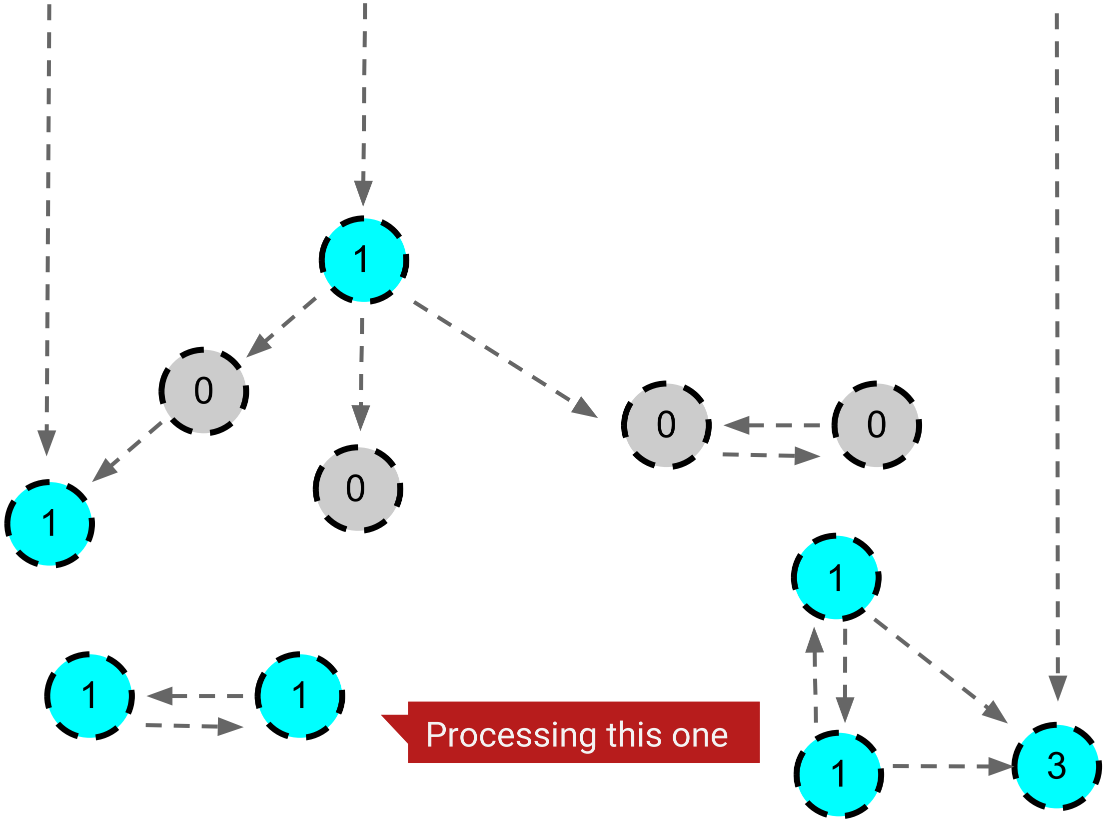


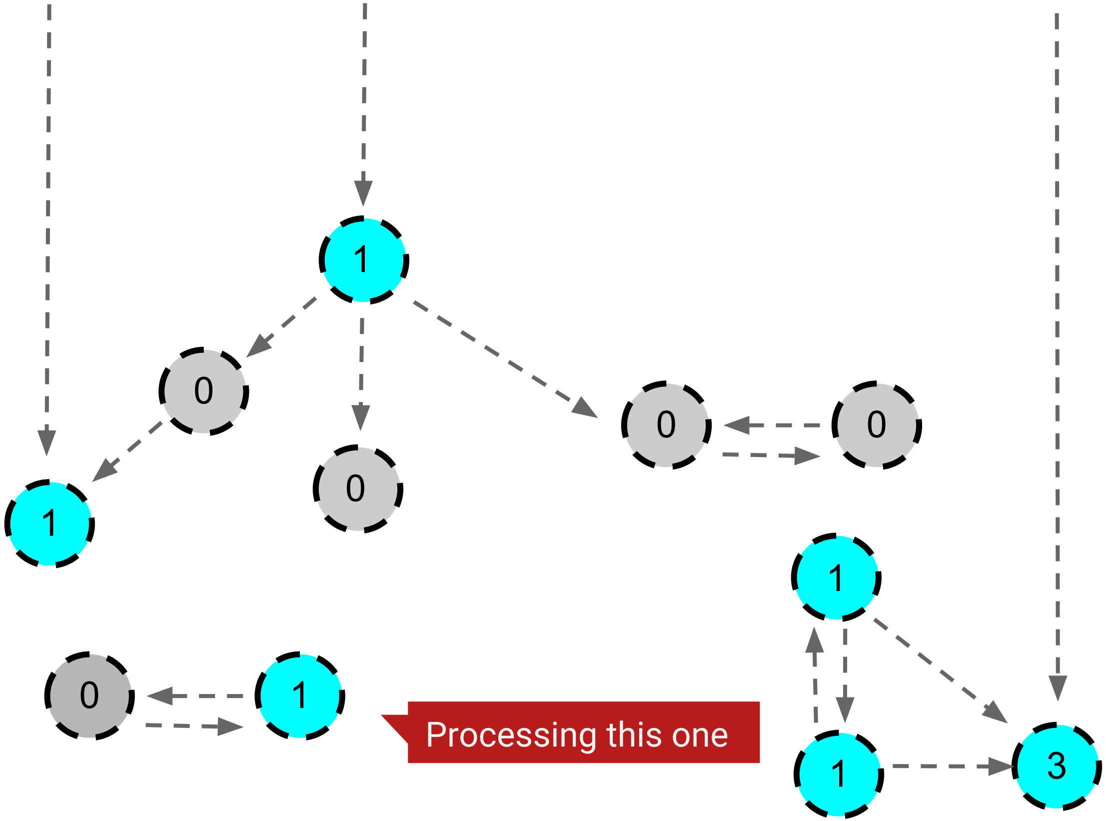
Processing this one
1
0
0
1
1
1
1
1
1
1
1
3

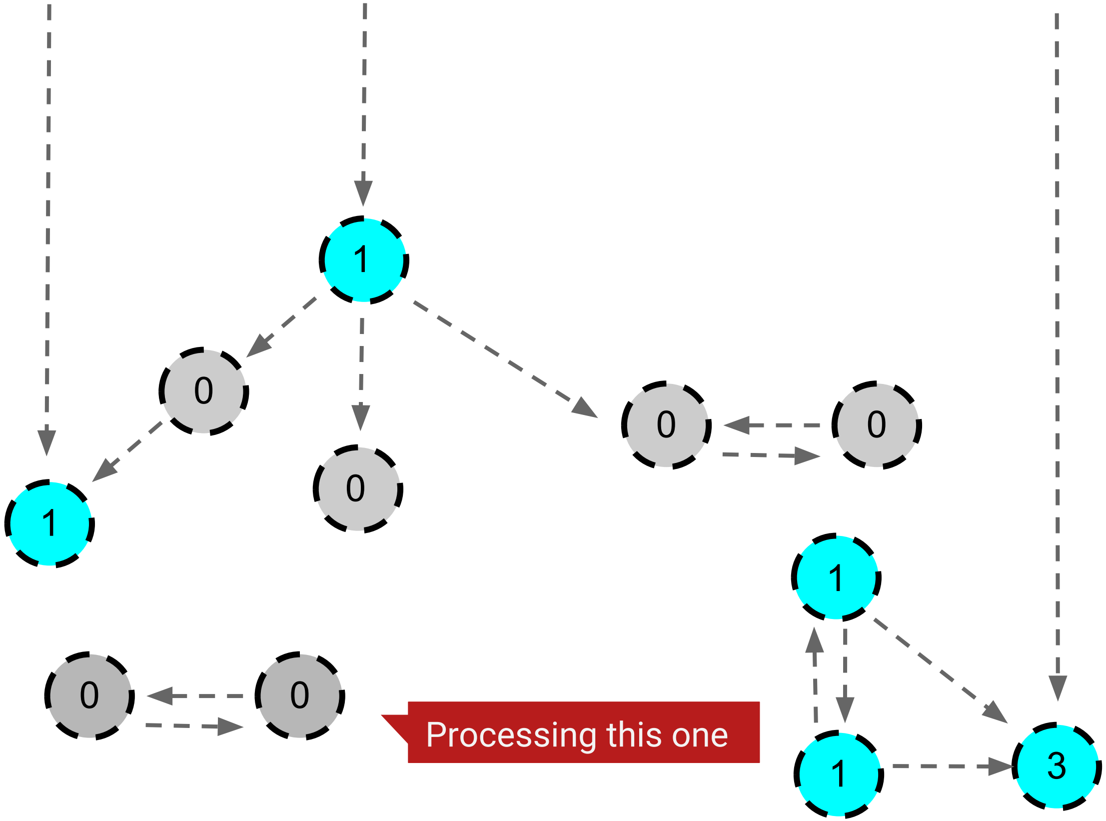


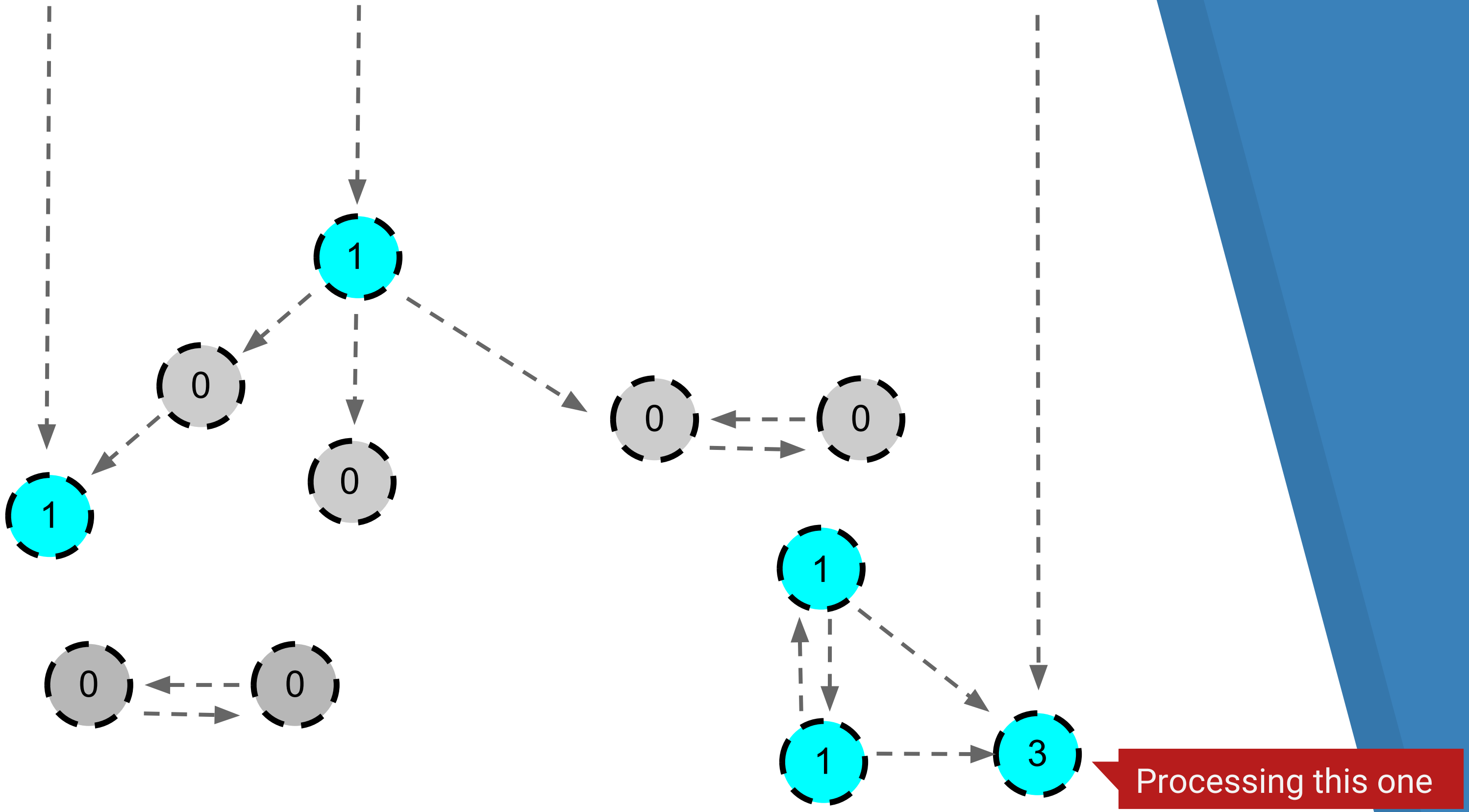
Processing this one
1
0
0
1
0
1
1
1
1
1
3

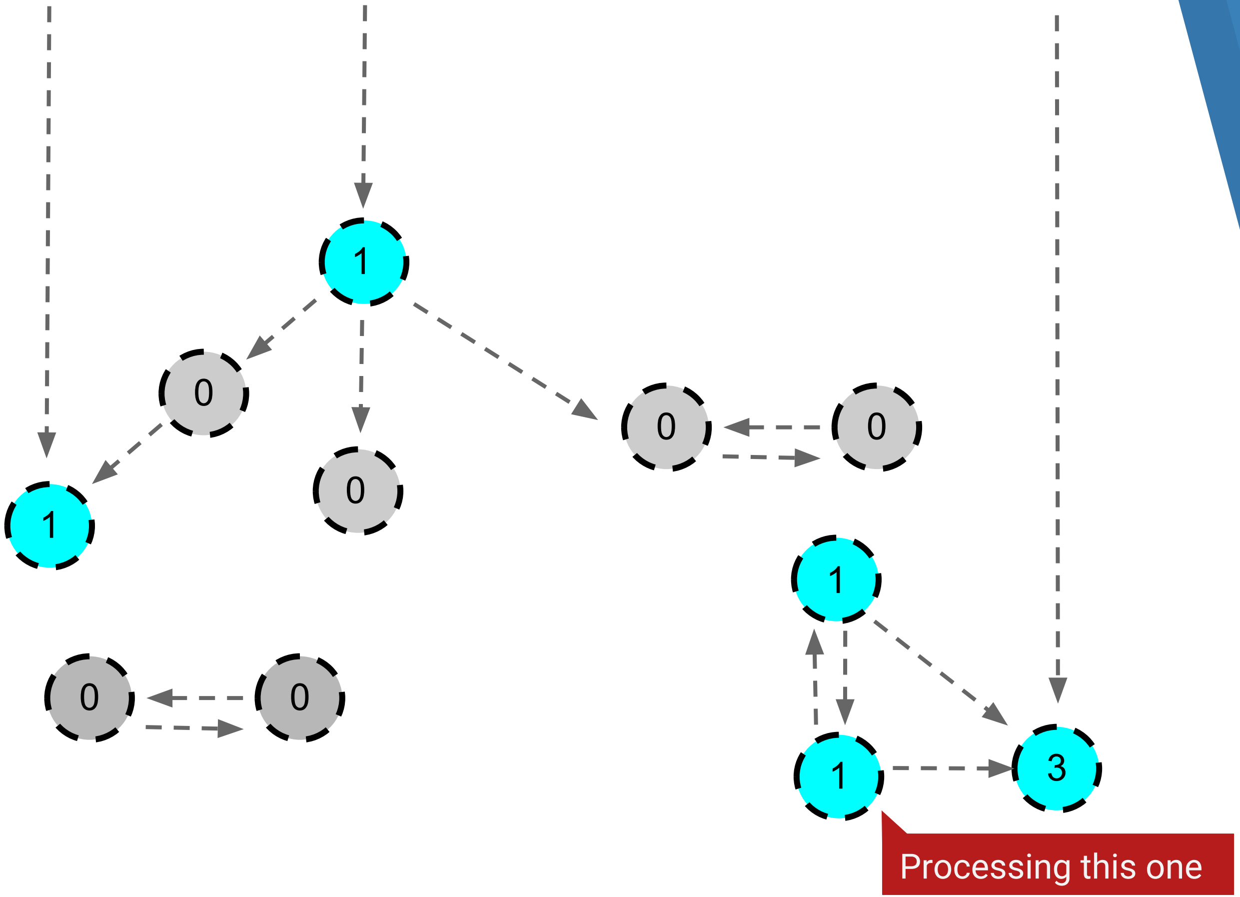

1
Processing this one
0
0
0
0
1
1
1
1
1
3

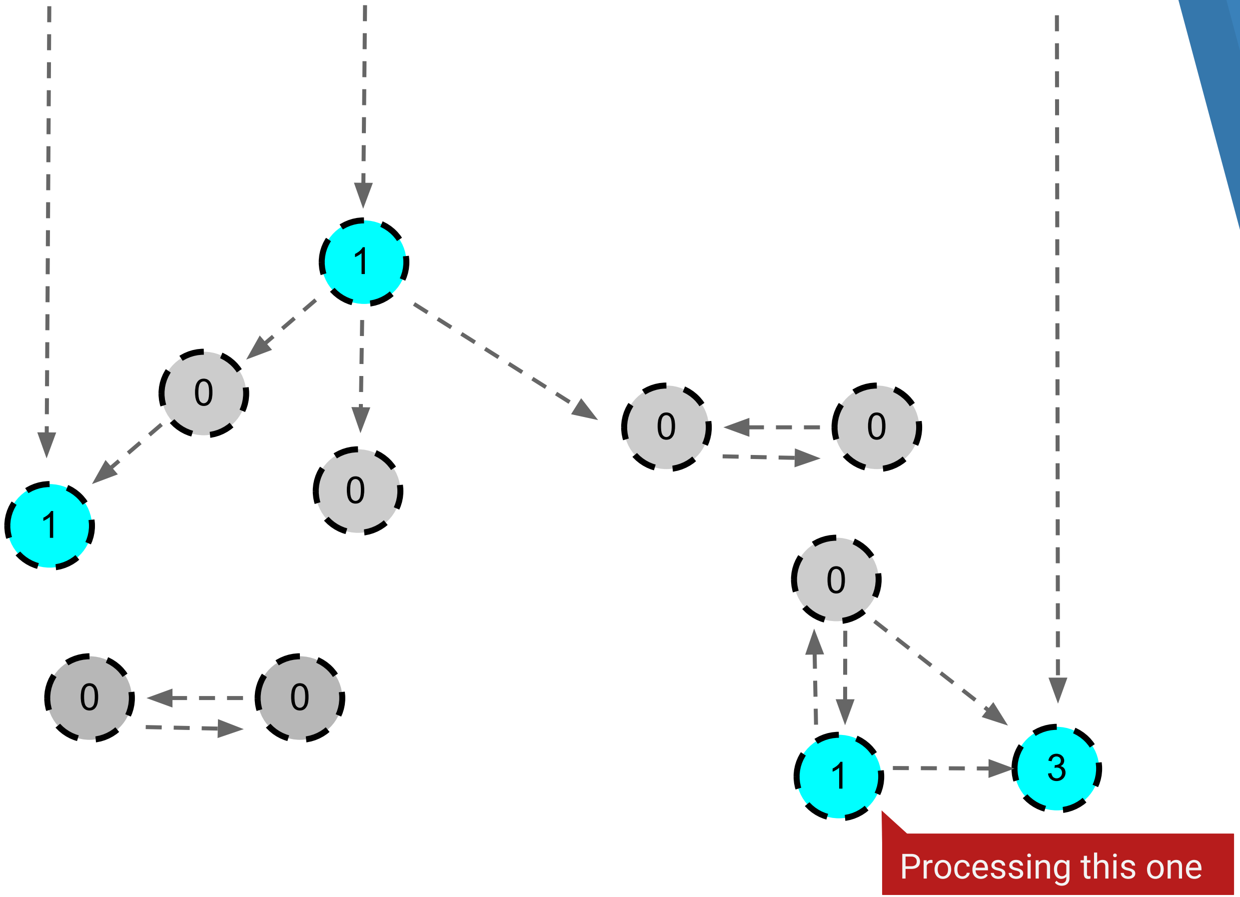

1
0
0
0
0
1
1
1
1
1
3
Processing this one

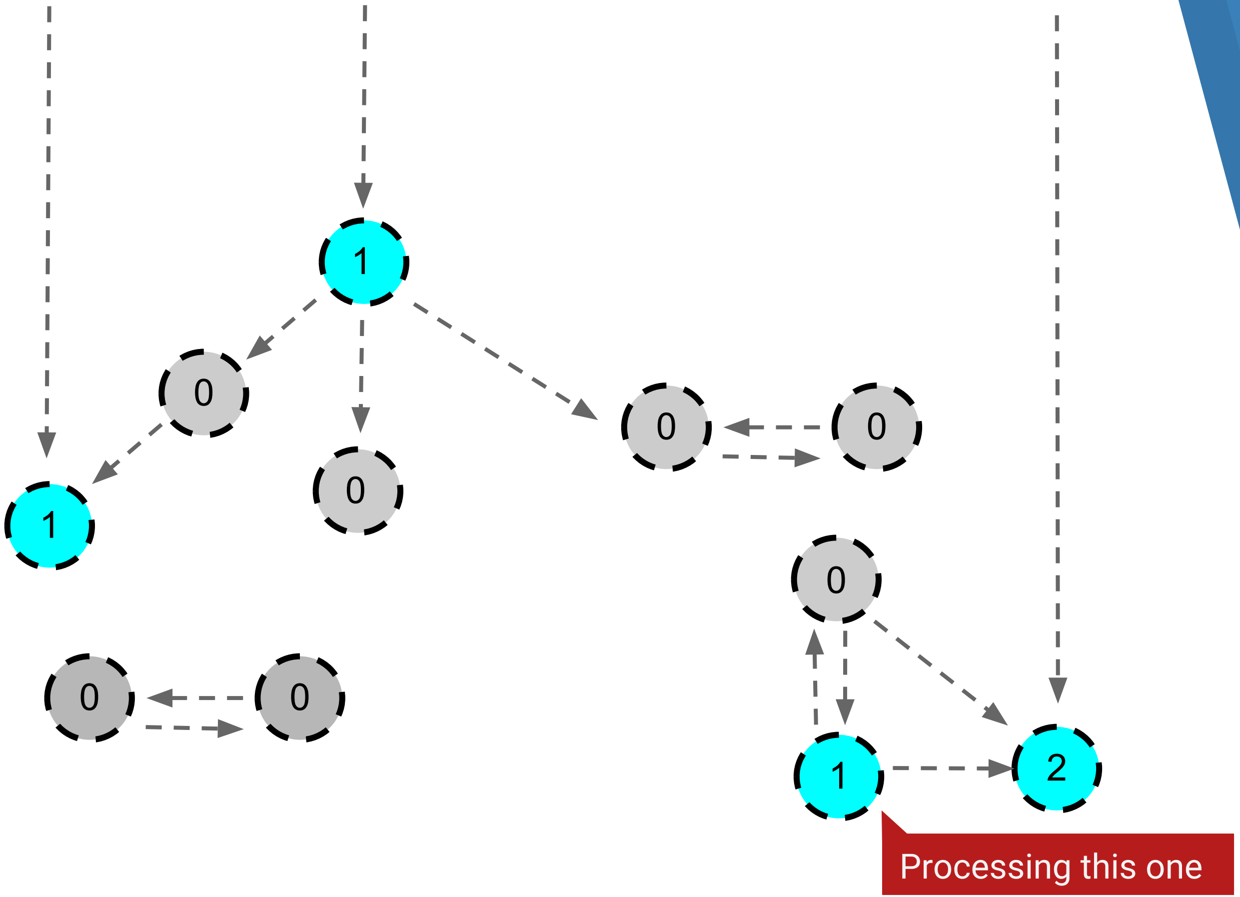

1
0
0
0
0
1
1
0
1
Processing this one
1
3

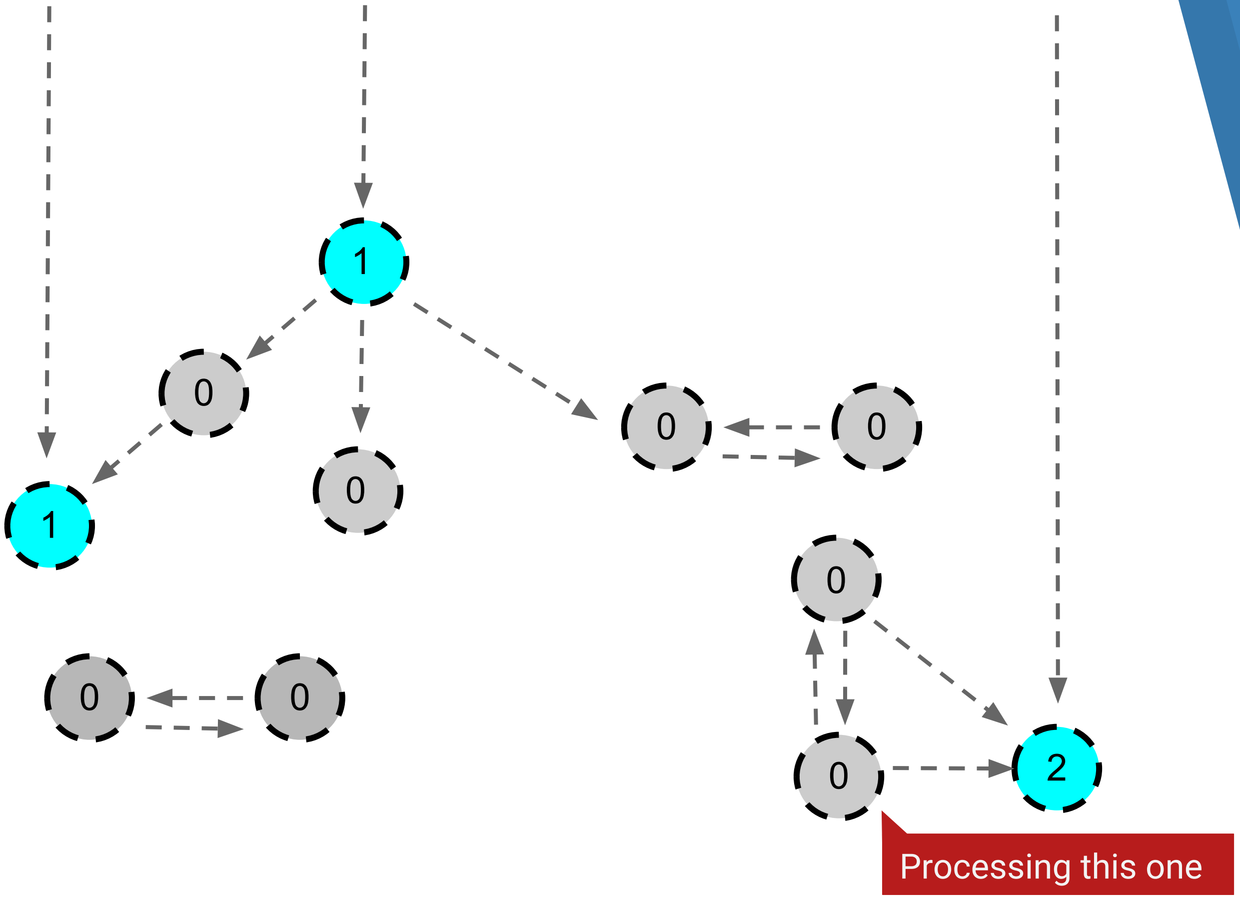

1
0
0
0
0
1
1
0
0
Processing this one
1
1
3

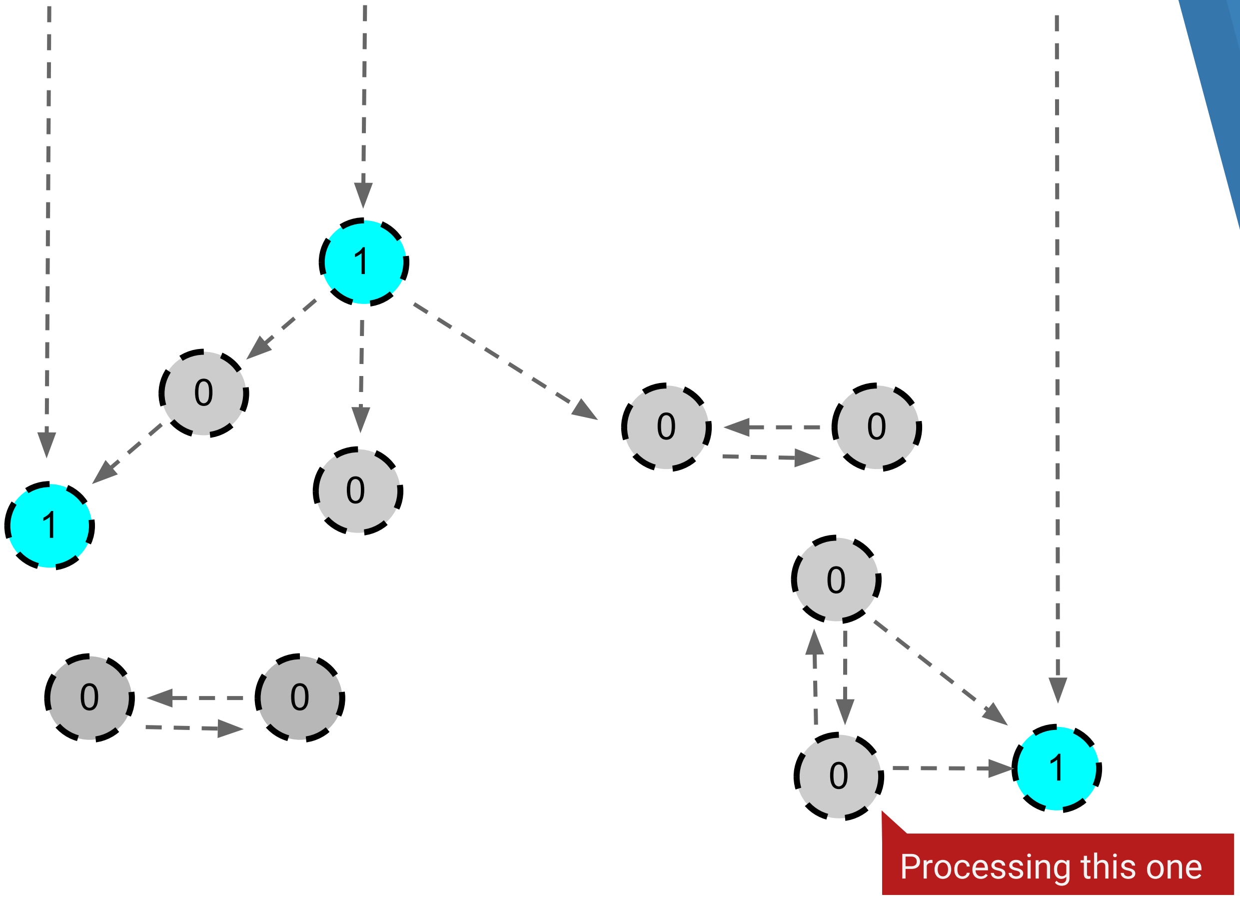

1
0
0
0
0
1
1
1
3
0
0
Processing this one


1
0
0
0
0
1
1
1
3
0
0
Processing this one


1
0
0
0
0
1
0
0
0
1
3
Processing this one


1
0
0
0
0
1
0
0
0
0
1
2
Processing this one

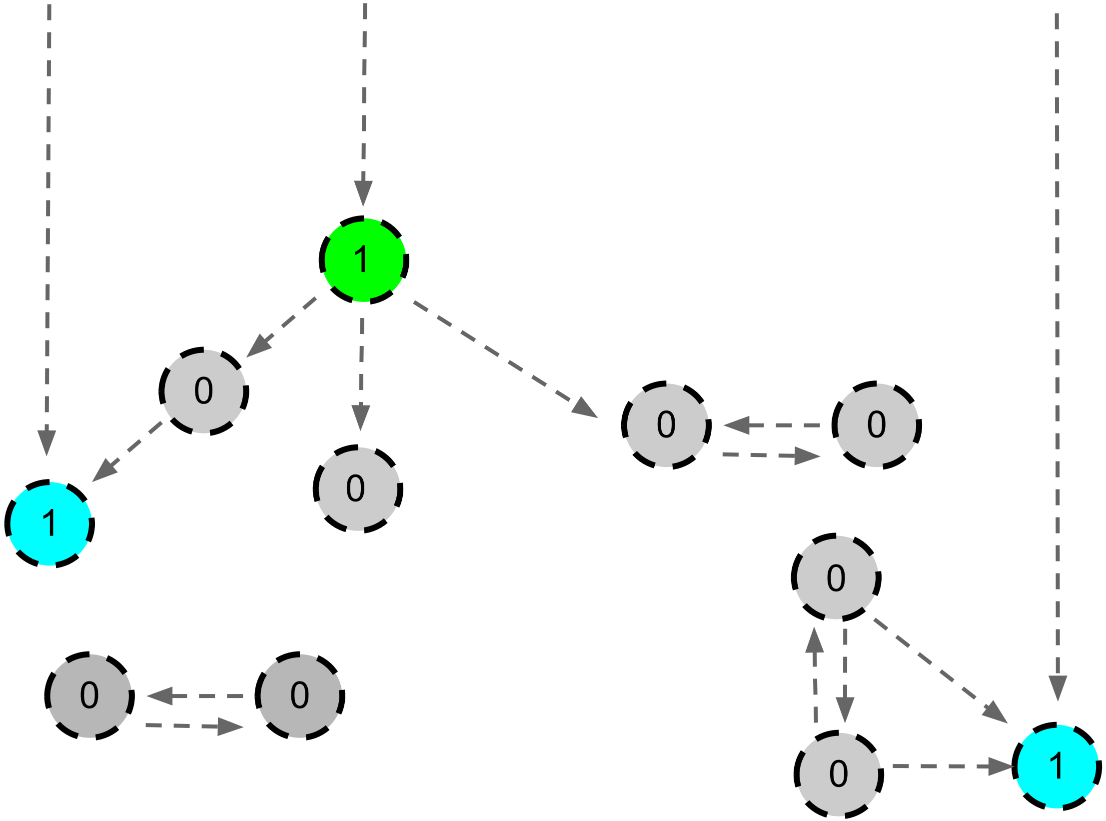

1
0
0
0
0
1
0
0
0
0
2
Processing this one

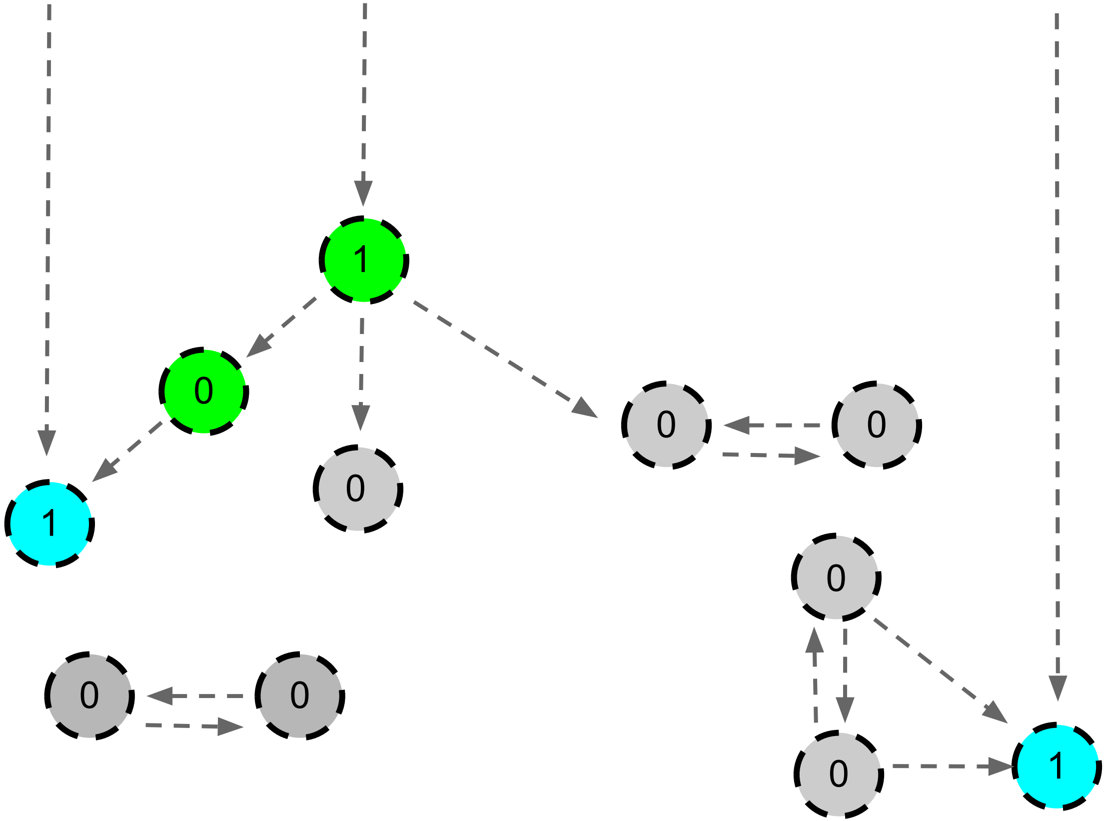


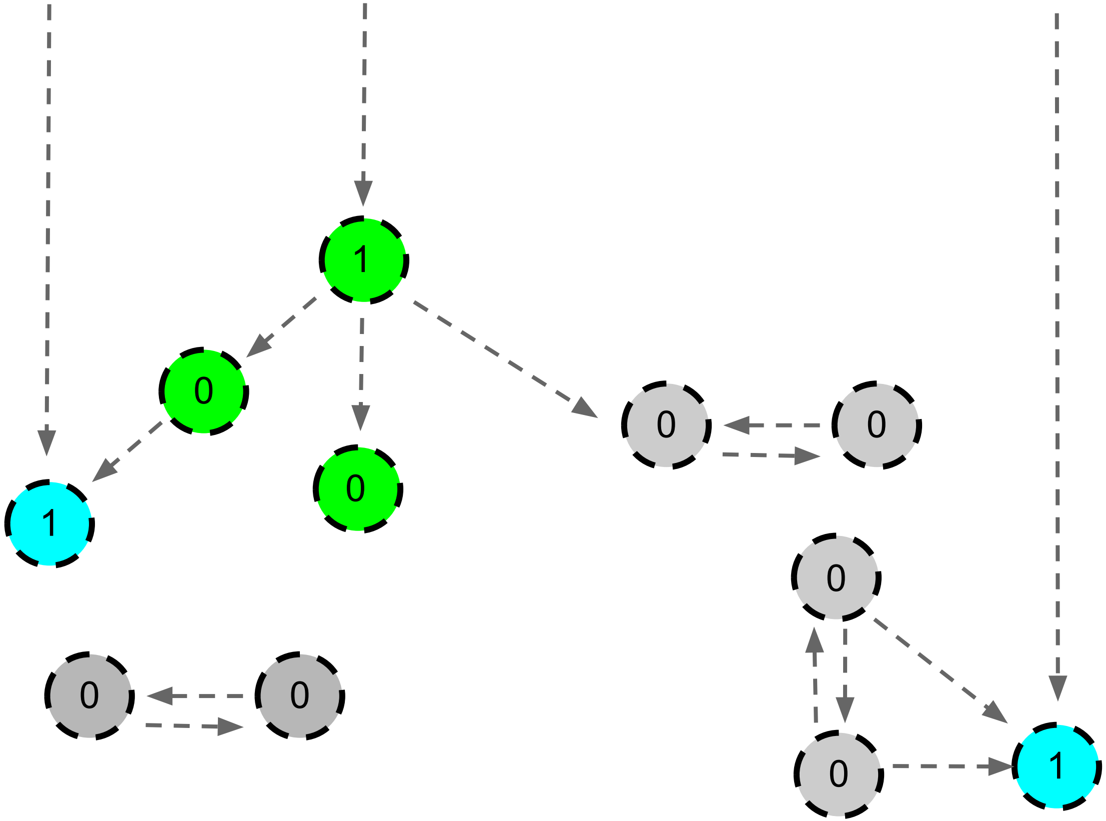
1
0
0
0
0
1
0
0
0
0
1
Processing this one

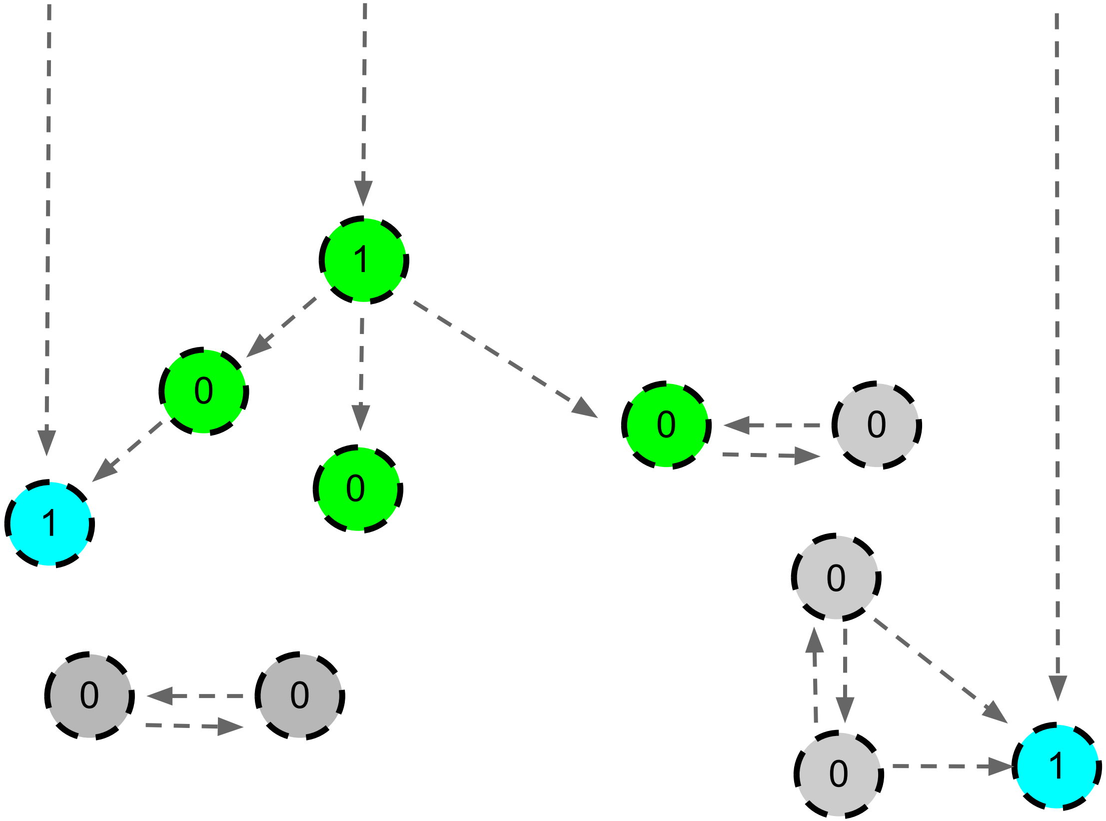
# STEP 2

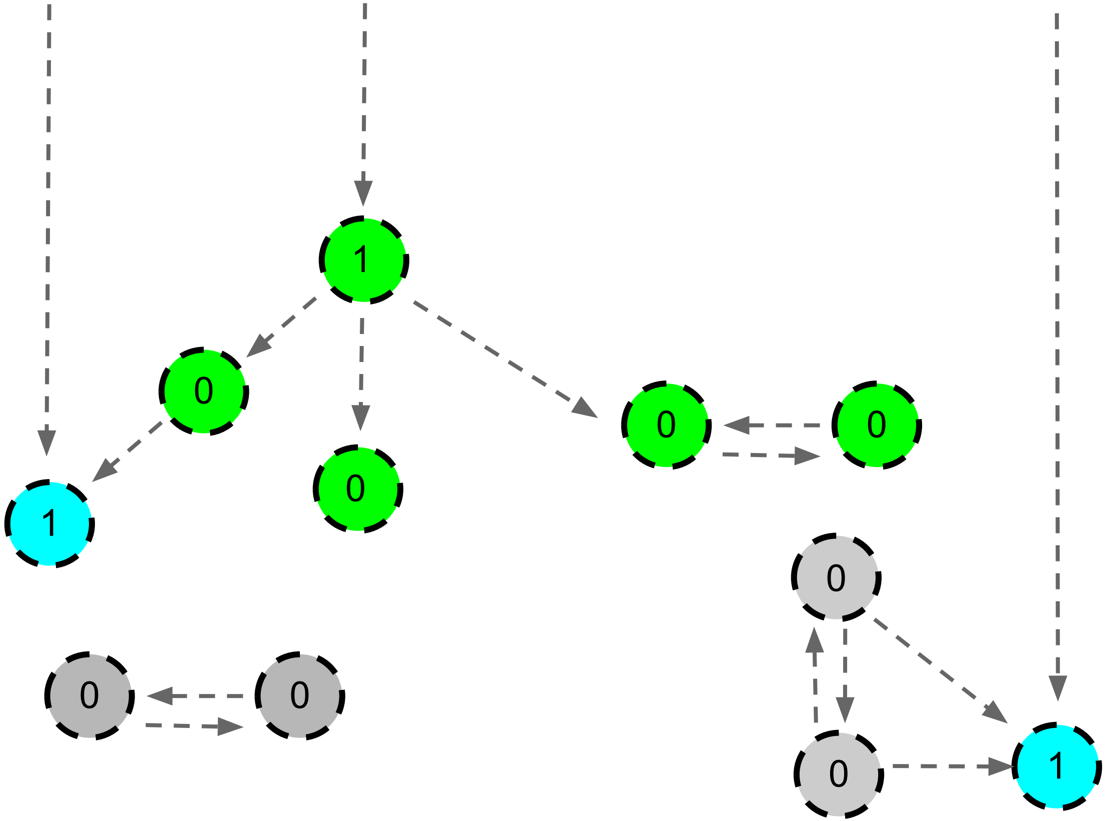
1. **Mark** objects with references != 0 as "*reachable*".
2. **Follow** object graph and mark referenced objects as "*reachable*"

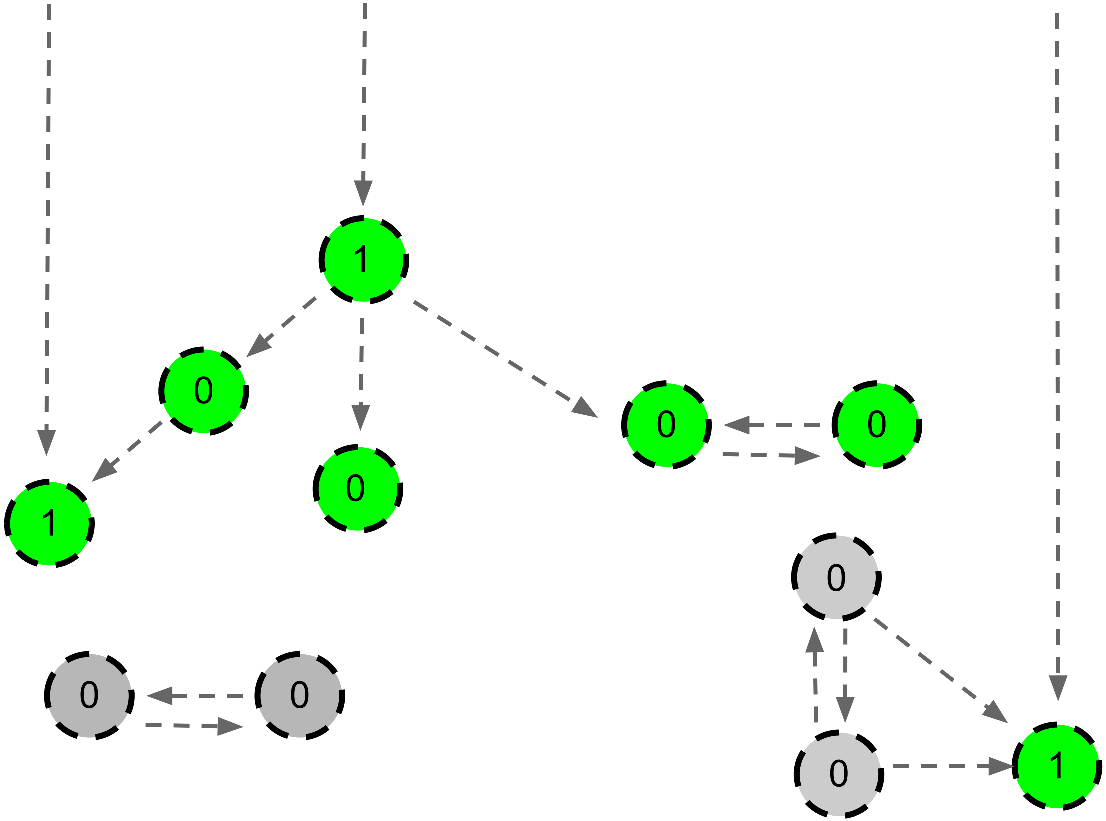

1
0
0
0
0
1
0
0
0
0
1


1
0
0
0
0
1
0
0
0
0
0
1


1
0
0
1
0
0
0
0
0
0
1

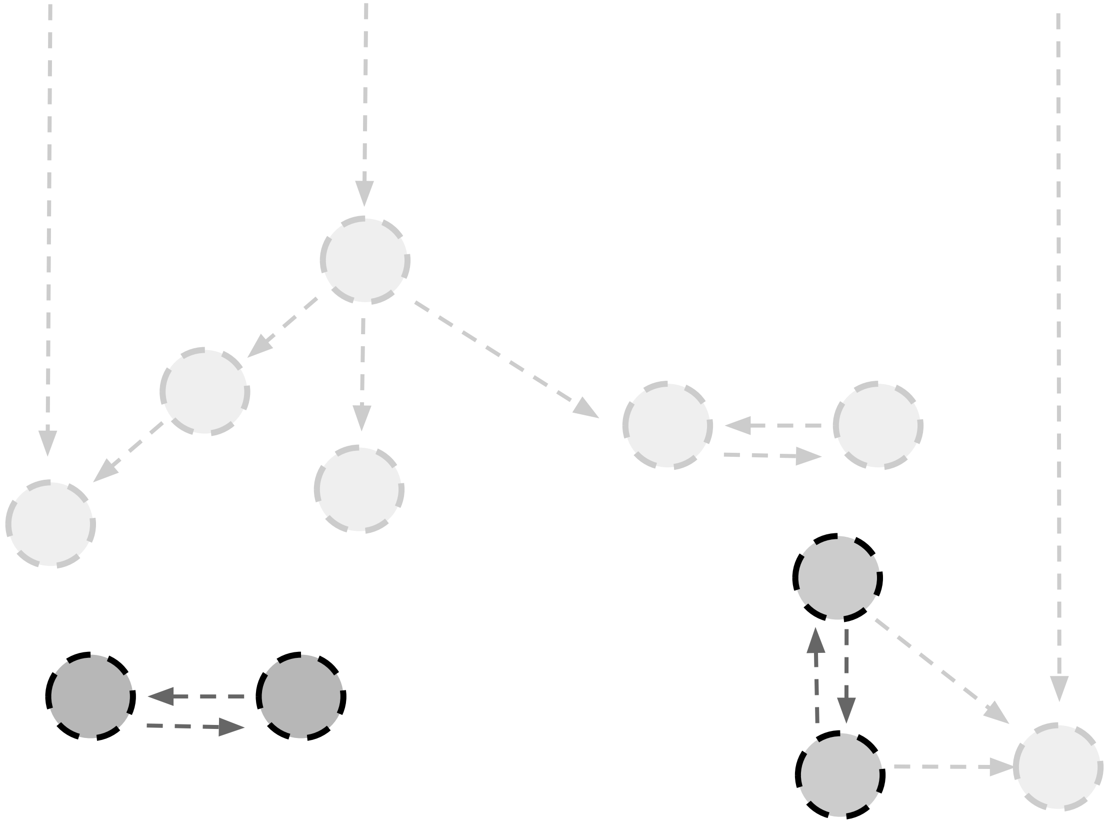

1
0
0
0
0
1
0
0
0
0
1

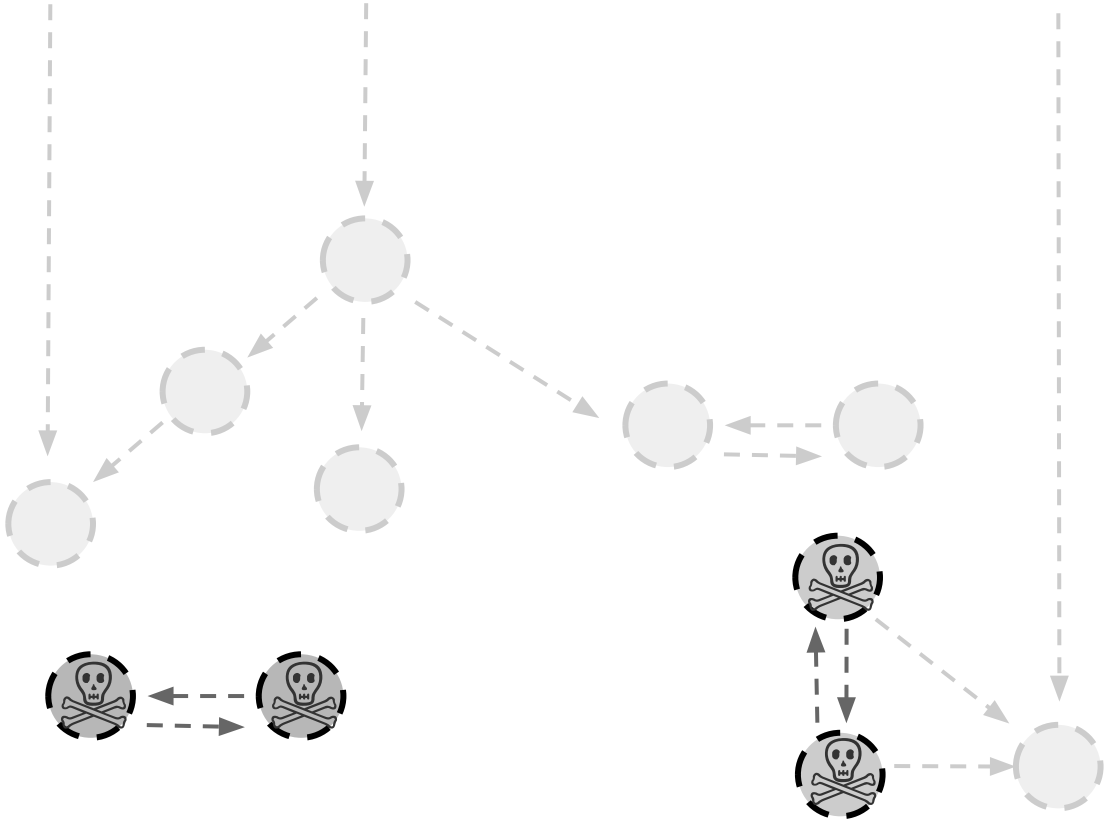

1
0
0
0
0
1
0
0
0
0
0
1


1
0
0
0
0
1
0
0
0
0
1

# STEP 3

1. **Kill** unreachable objects





Usually, objects that remain **reachable** are more likely to remain reachable in future runs of the algorithm.

To avoid doing extra work **extra work** we need

# generations

3 (older)

2

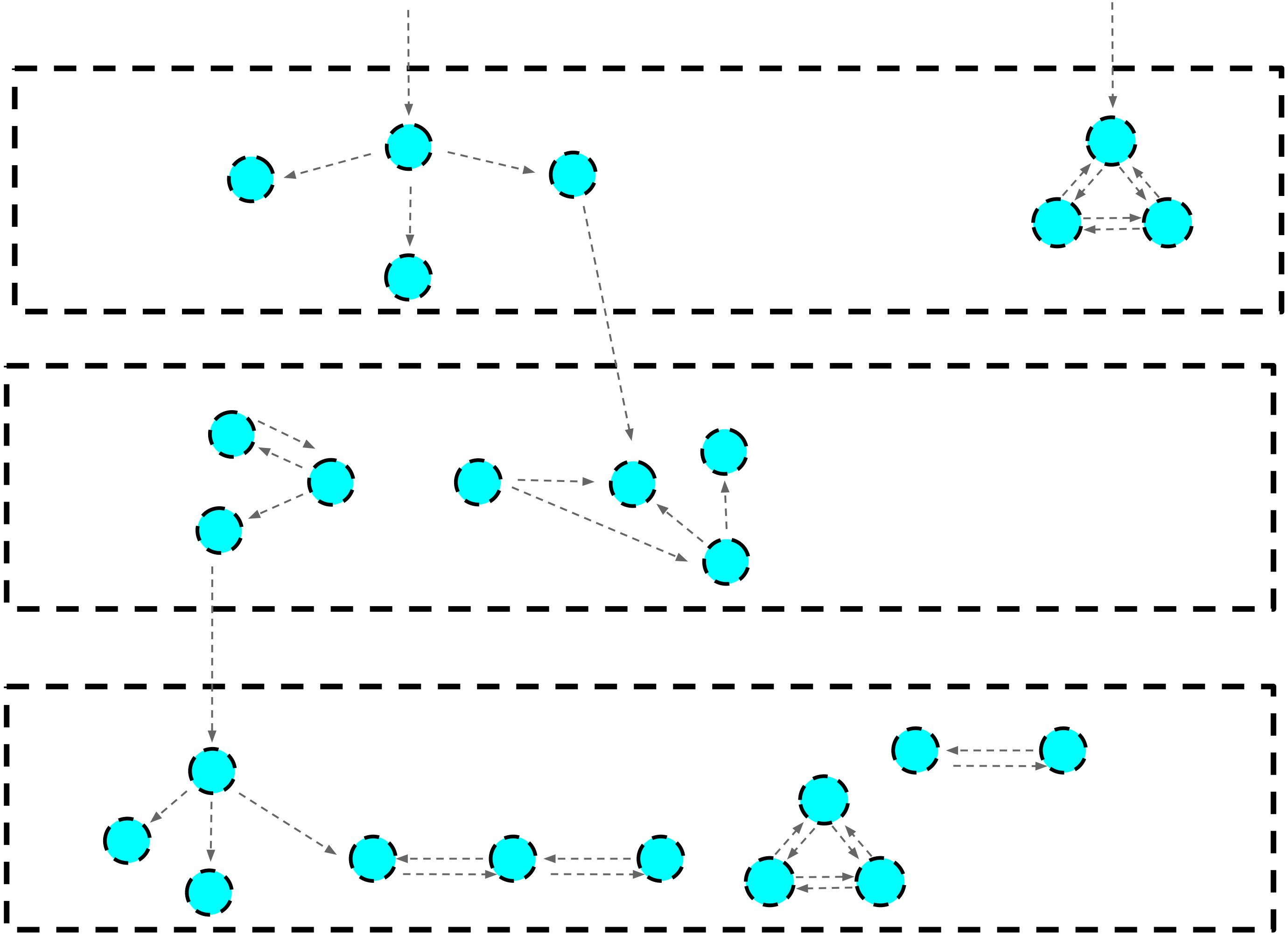
1 (younger)

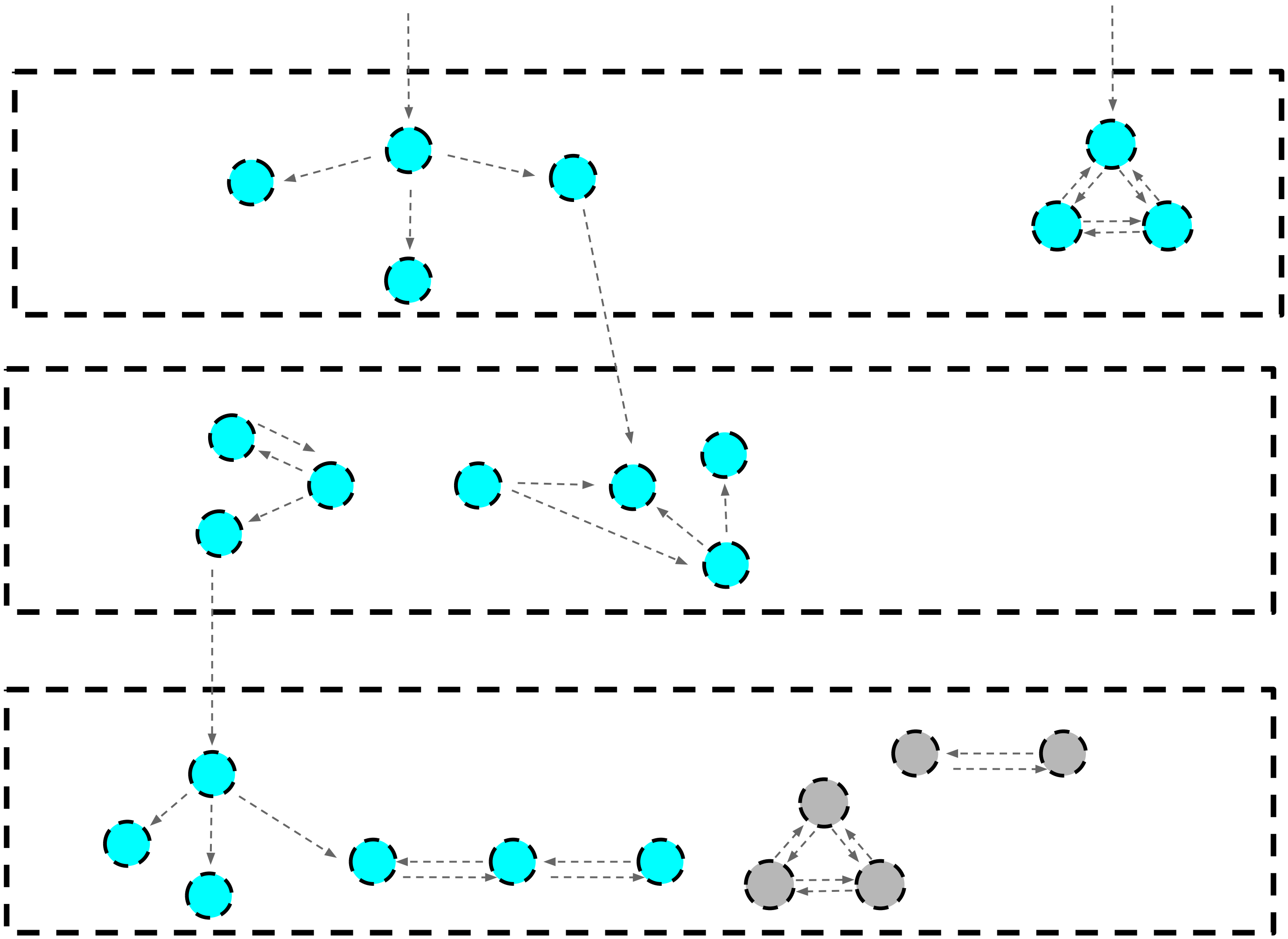
**Objects** that

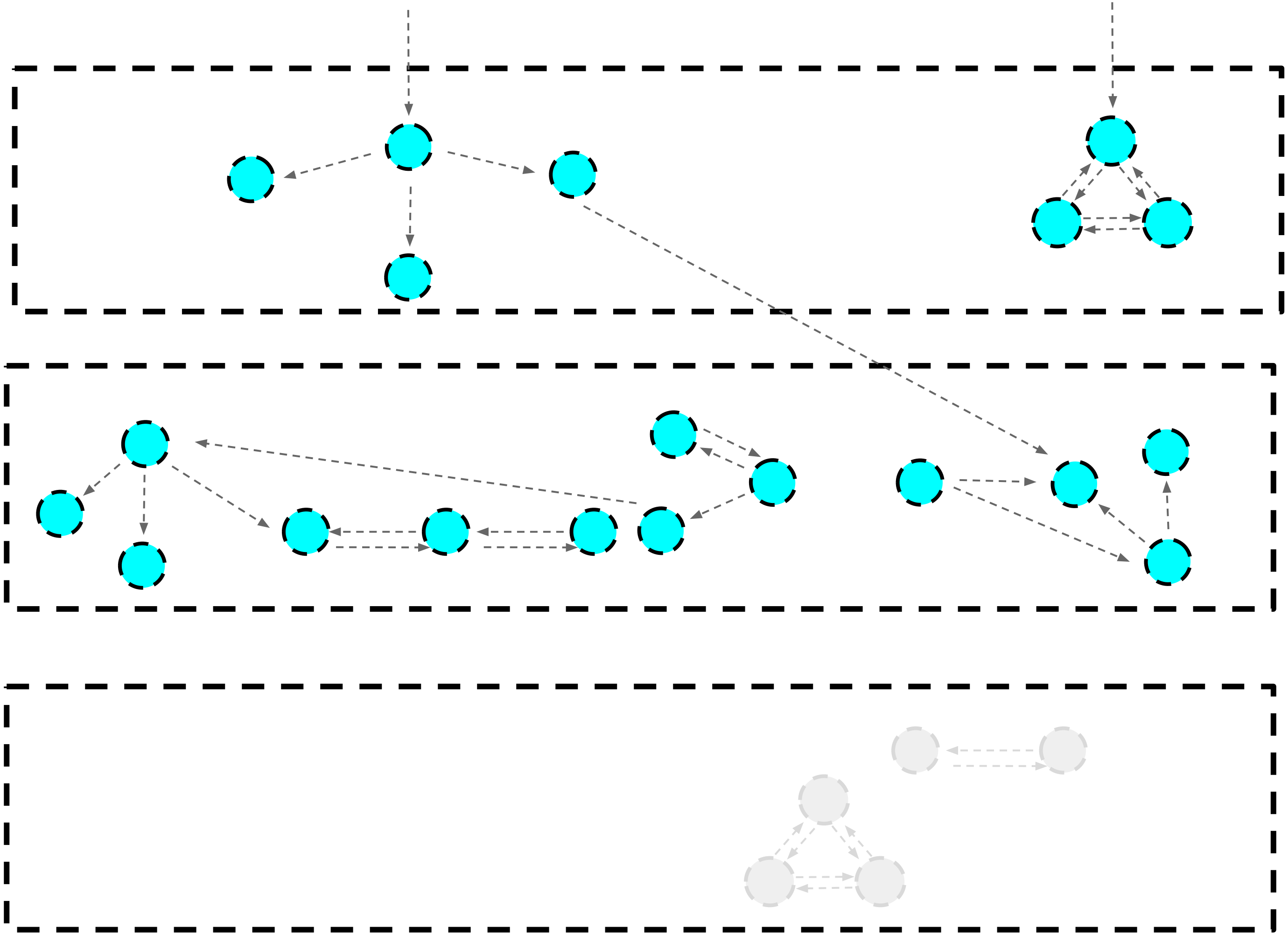
# survive

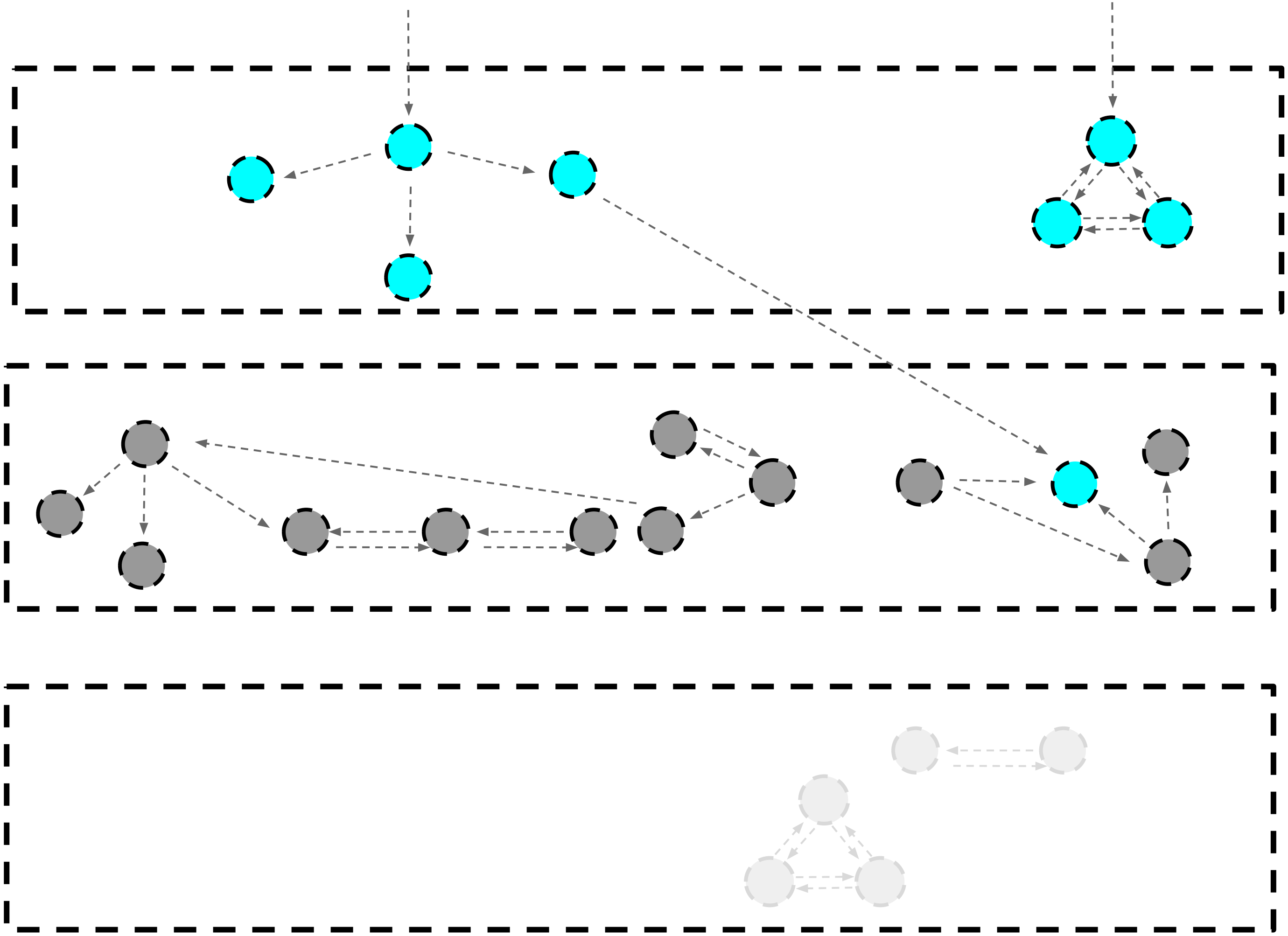
a garbage collector pass are **promoted** to the next generation.

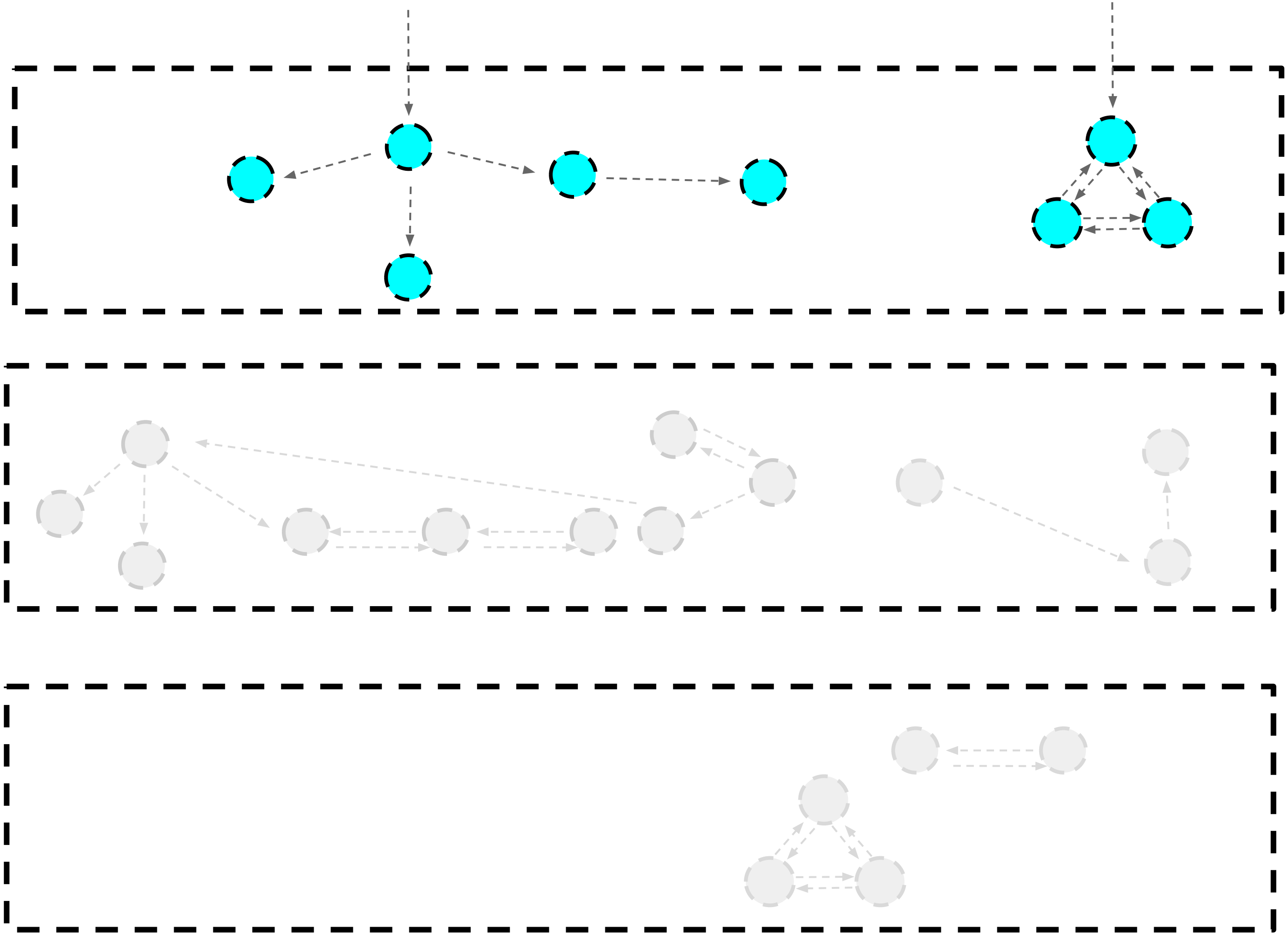
- Objects in older generations have survived more and more garbage collector passes.











The **garbage collector** makes a pass when a generation size reaches a

# threshold

This makes garbage collector passes **less frequently** in **older** generations as it is more difficult to reach them.

You can **get** and **set** the thresholds

```
>>> import gc
>>> gc.get_threshold()
(700, 10, 10)
>>> gc.set_threshold(1000,50,20)
>>> gc.get_threshold()
(1000, 50, 20)
```

You can **force** the collection of a generation

```
>>> gc.collect(generation=1)
32
```

# 6.

## The GIL

Actually, is a combination of a condition variable, a boolean and a mutex

The **Global Interpreter Lock** is a (vilified)

# mutex

- **Protects** access to Python objects.
- Prevents multiple threads from executing Python code **in parallel**.
- … so we **cannot** benefit from multi-core machines.
- For that, use [multiprocessing] instead.

We **need** it (mostly) to

# synchronize

- Every time we modify refcounts.
- We need to have an atomic count of object references.
- ... or some Python objects may fall through the cracks.

From the Python docs:

"Without the lock, even the simplest operations **could cause problems** in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being **incremented only once instead of twice.**"

Other **features** have grown to **depend** on the **guarantees** that the **GIL** enforces, but memory management is the main reason it exists.

And that's why **removing** it is **so hard**. Like *a lot*.

# 7.

## Practical applications

# DEBUGGING

## reference cycles

```python
>>> class Node:
...     pass
...
>>> a = Node()
>>> b = Node()
>>> a.b = b
>>> b.a = a
>>> import gc
>>> gc.set_debug(gc.DEBUG_SAVEALL)
>>> del a
>>> del b
>>> gc.collect()
62
>>> gc.garbage
[...,
 {'b': <__main__.Node object at 0x7f44c2f8f5c0>},
 {'a': <__main__.Node object at 0x7f44c2f8f630>}
]
```

# CALCULATING

total memory costs

```python
>>> x = [[1,2,3,4],5,6,{1:3,5:[4,5,6]}]

>>> import sys
>>> sys.getsizeof(x)
96

# That size is the size of the list + the pointers
>>> y = [1,2,3,4]
>>> sys.getsizeof(y)
96

# We can use the gc traversal to ge the total size
>>> sum(map(sys.getsizeof, gc.get_referents(x)))
392
```

The **garbage collector** is

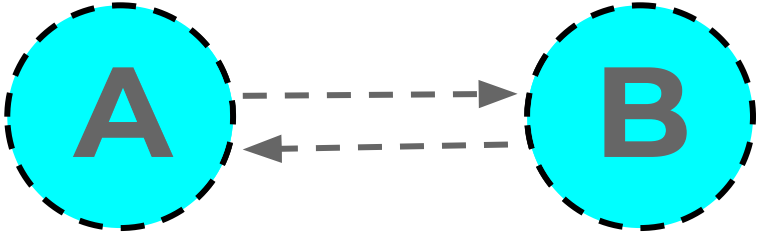
# atomic

- **Global**, not *per* thread.
- A consequence: it will **stop all the threads**.
- Because of the **GIL** the threads are **not** running in parallel anyway, but we will notice that one one of the threads takes **longer**.

# 8.

## Object finalization

# Order of destruction




- Which destructor should be called first?
- What if the destructor of A needs something in B and we deleted B first?
- What if the destructor of A calls the destructor of B?

# Resurrection

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         print("I am going to be resurrected")
...         global x
...         x = self
...
>>> a = Lazarus()
>>> del a
I am going to be resurrected
>>> x
<__main__.Lazarus object at 0x7f44c2f69fd0>
```

If an object has a __del__() method and participates in a cycle it is placed in the

# garbage

collection of the garbage collector.

- These are the **unreachable** and **uncleanable** objects.

# Python 2

```
>>> class Node:
...     def __del__(self):
...         print("Calling __del__")
...
>>> a = Node()
>>> b = Node()
>>> a.b = b
>>> b.a = a
>>> del (a,b)
>>> import gc
>>> gc.collect()
4
>>> gc.garbage
[<__main__.Node instance at 0x7f556effc680>,
 <__main__.Node instance at 0x7f556effc200>]
```

Python introduced

# PEP 442

For safe and deterministic object finalization.

Takes into account:

- Resurrection.
- Every destructor is only called once.
- Safety.

Python >3.4

```
>>> class Node:
...     def __del__(self):
...         print("Calling __del__")
...
>>> a = Node()
>>> b = Node()
>>> a.b = b
>>> b.a = a
>>> del (a,b)
>>> import gc
>>> gc.collect()
Calling __del__
Calling __del__
62
```

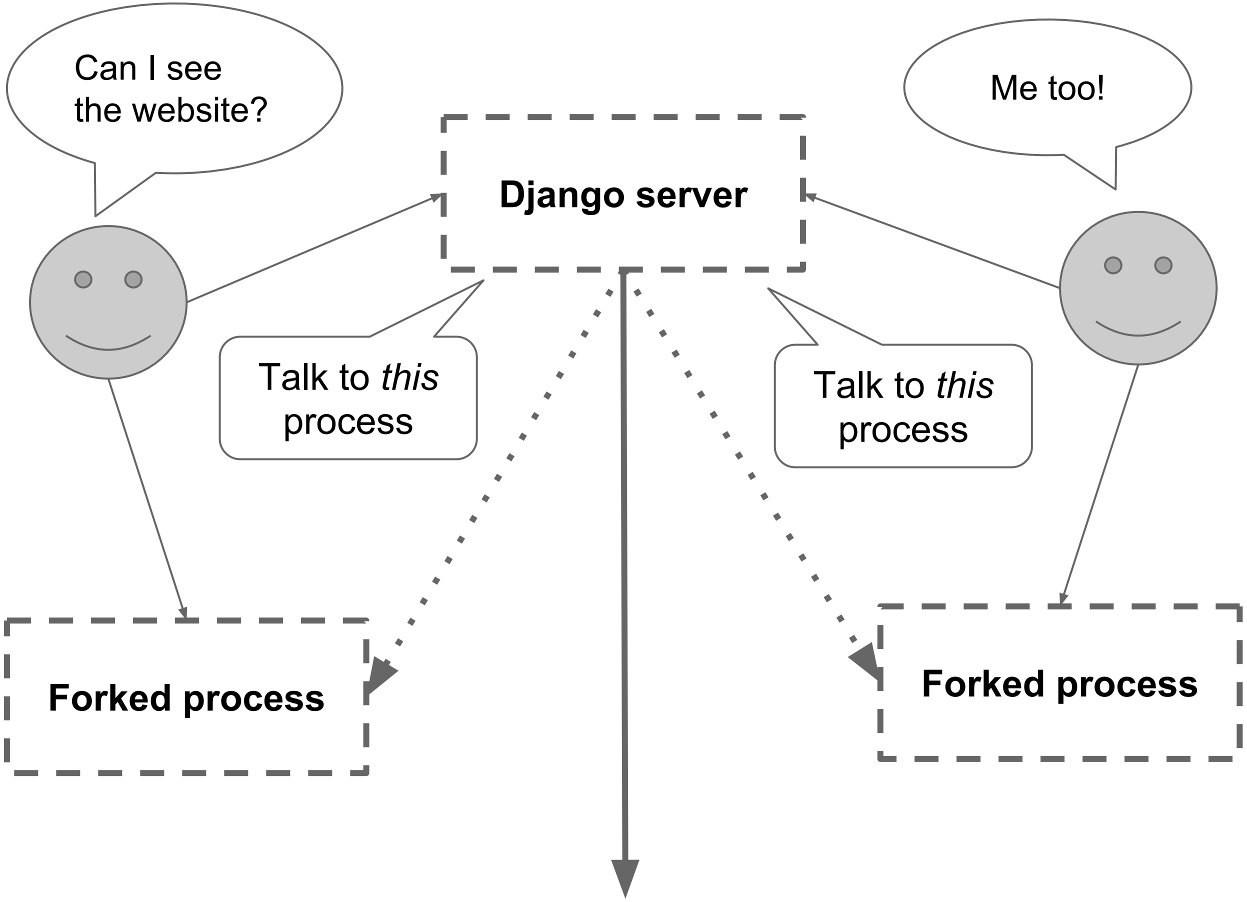
# 8.

## Copy-on-write (CoW)

How does a

# web server work?

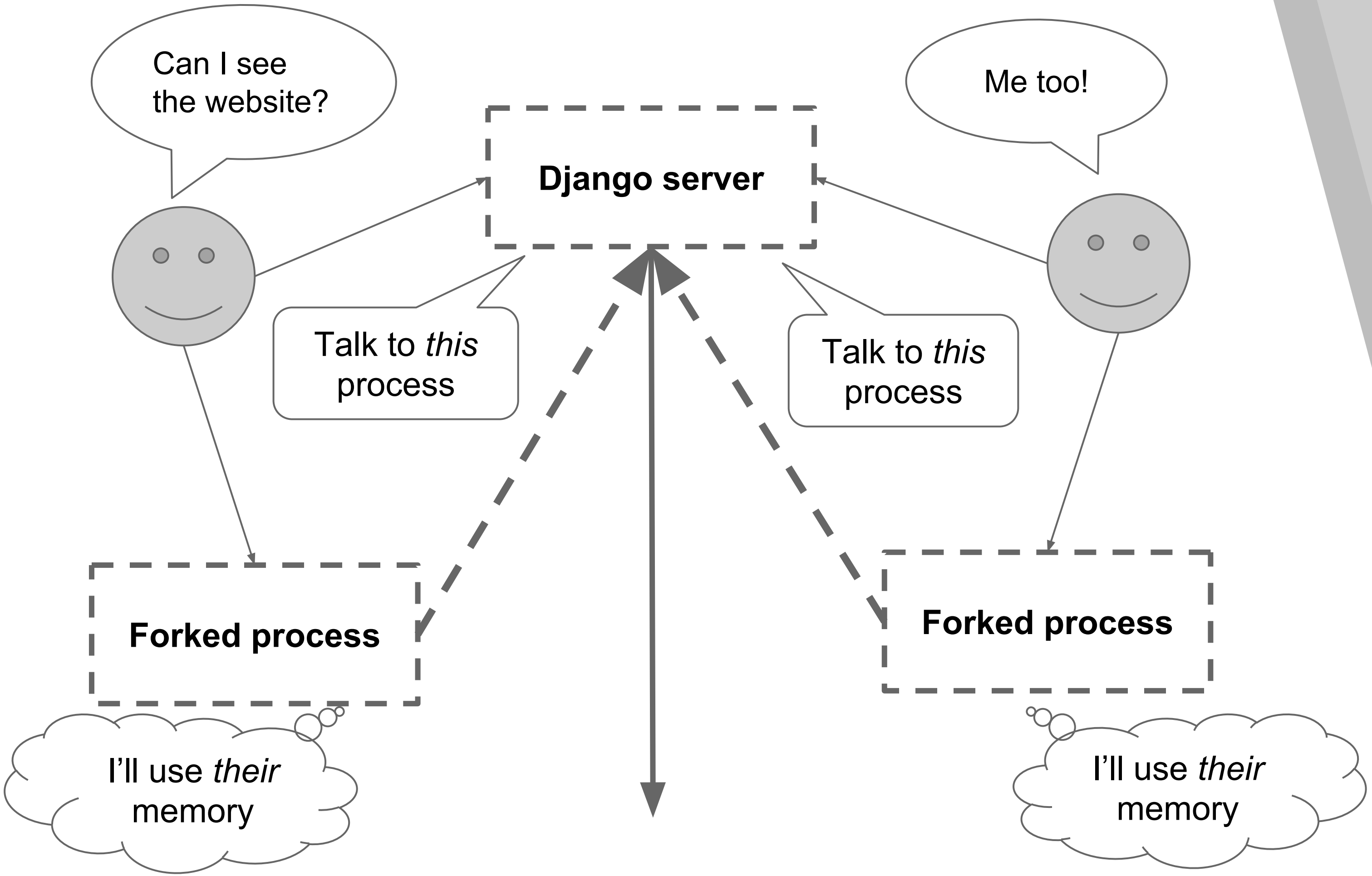
- Say Django.
- A single **master** process.
- For each incoming user request, **forks** a new process.
- Each request served by **its own** process, all in parallel.


Can I see the website?
Me too!
Django server
Talk to *this* process
Talk to *this* process
Forked process
Forked process

Copy-on-Write is an

# optimization

- Used for **forked** processes.
- Child process starts **sharing** the memory of its parent.
- We don’t make a copy until we **need to modify** it.
- If we only read it, we never get to make a copy.


Can I see the website?
Me too!
Django server
Talk to *this* process
Talk to *this* process
Forked process
Forked process
I'll use *their* memory
I'll use *their* memory

But **because** of

# reference counting

- The read itself needs to increment the refcount...
- These references are stored in the **PyObject**.
- Thus, reading the object → updating its refcount → **modifying** the object → forked process makes a copy.
- Copy-on-write (CoW) **becomes** Copy-on-Read (CoR) !!
- Our server needs 10x memory to serve 10 requests.

This is the problem

# Instagram faced

- Solution: **disable** garbage collection.
- Isn't memory usage then going to grow *indefinitely*?
- **No**. The primary mechanism to free memory is still reference count. What they disabled was just using garbage collection to **break reference cycles**.

"Dismissing Python Garbage Collection at Instagram"

# 9.

# Something good

Python uses **memory**

# arenas

- A **contiguous** piece of memory, allocated at once.
- Python hands out parts of this memory **as needed**.
- PyMalloc() instead of malloc()
- **Block** is a chunk of a certain size → a Python object.
- **Pool** is a collection of blocks of the same size.
- **Arena** is a collection of pools.

Memory arenas make **deallocation**

# fast

- We don’t release the memory to the operating system.
- Instead, we just mark it as **unused** in the arena.
- It’s always held by Python, whether used or not.
- However…

# 10.

# Something bad

**PyMalloc** also causes

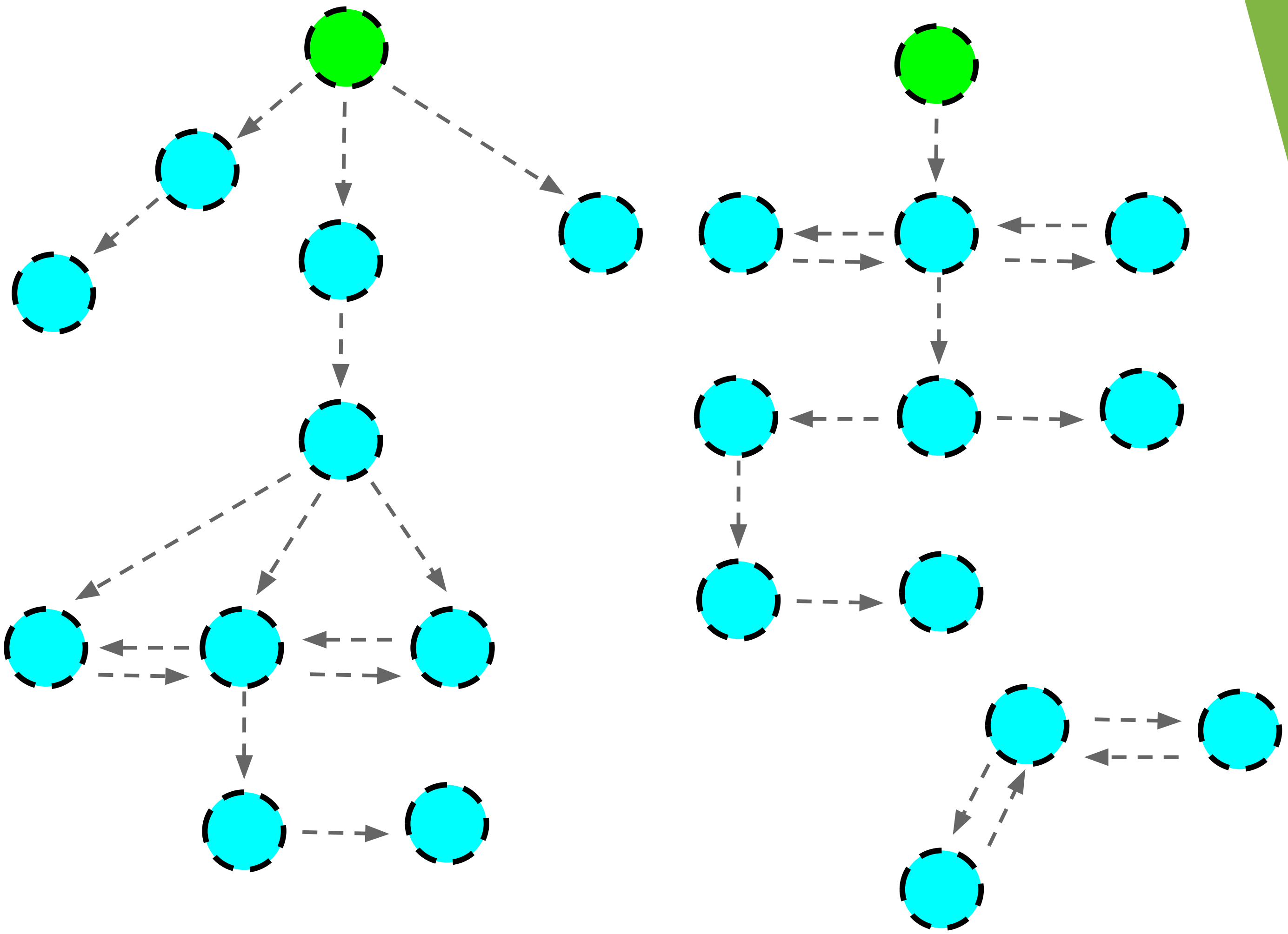
# fragmentation

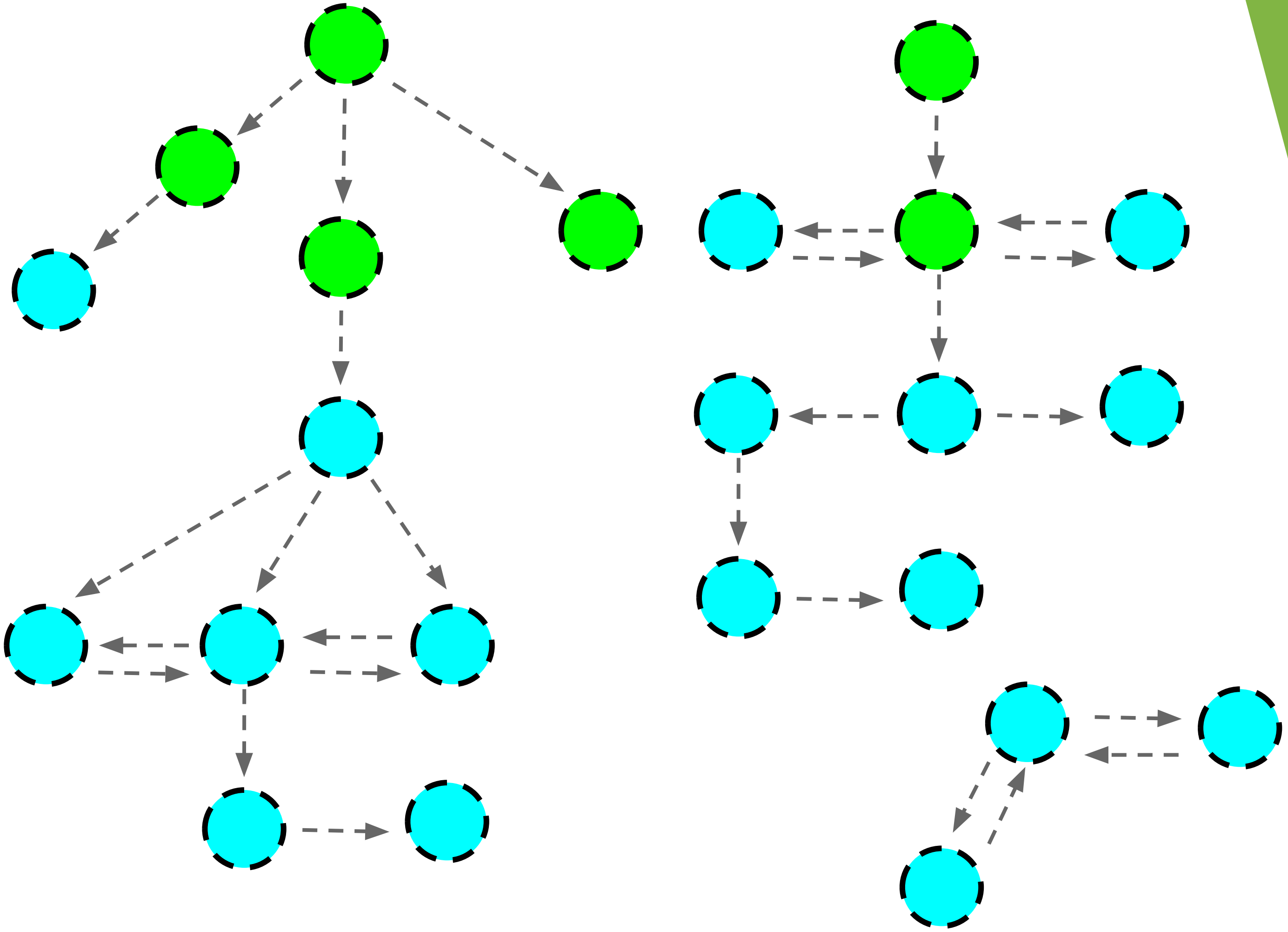
- An arena must be **completely** empty…
- … for it's memory to be **released**.
- As long as it contains **a single** object, we keep it.
- That's why Python rarely returns memory back to the operating system.

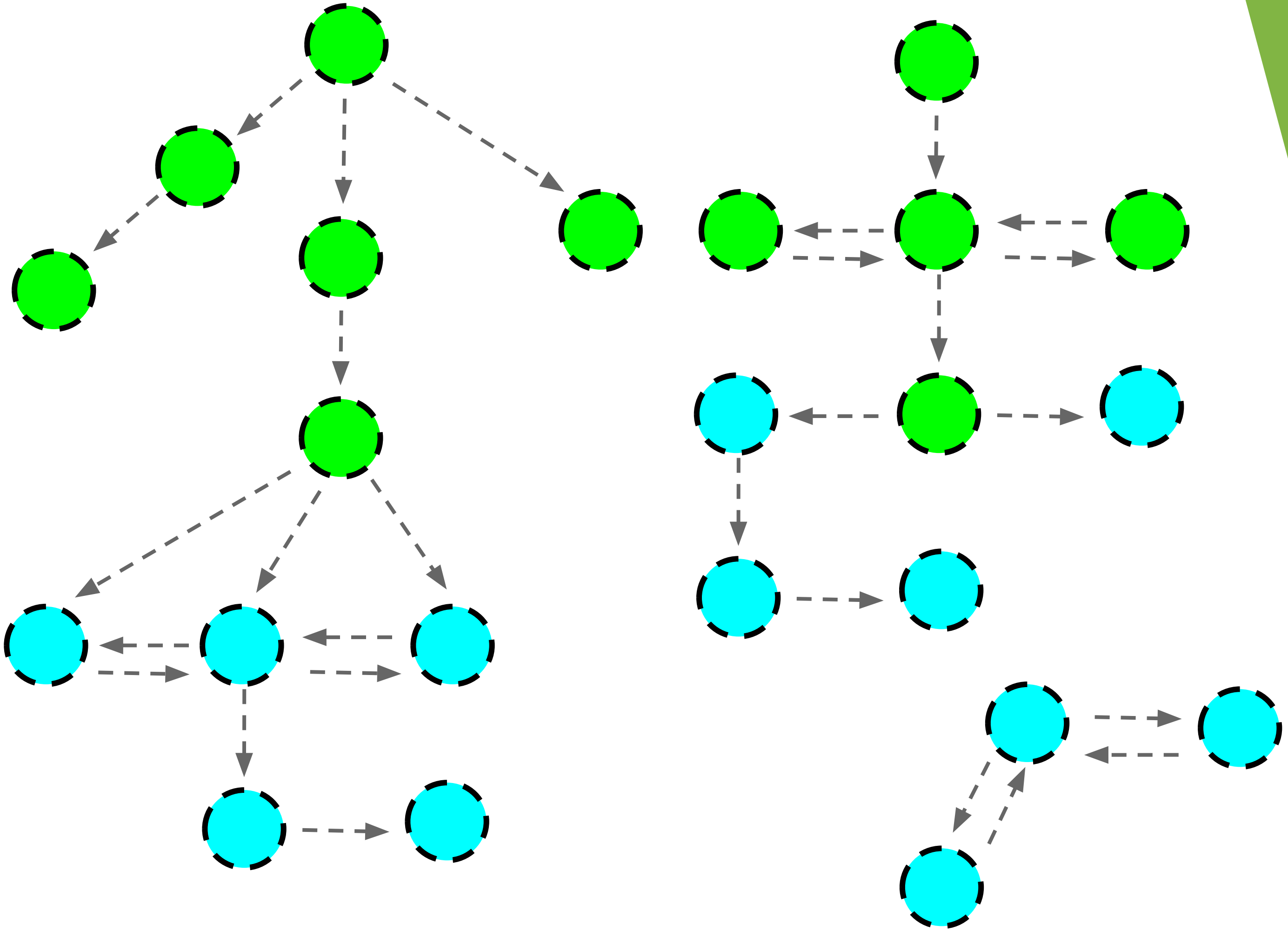
# 11.

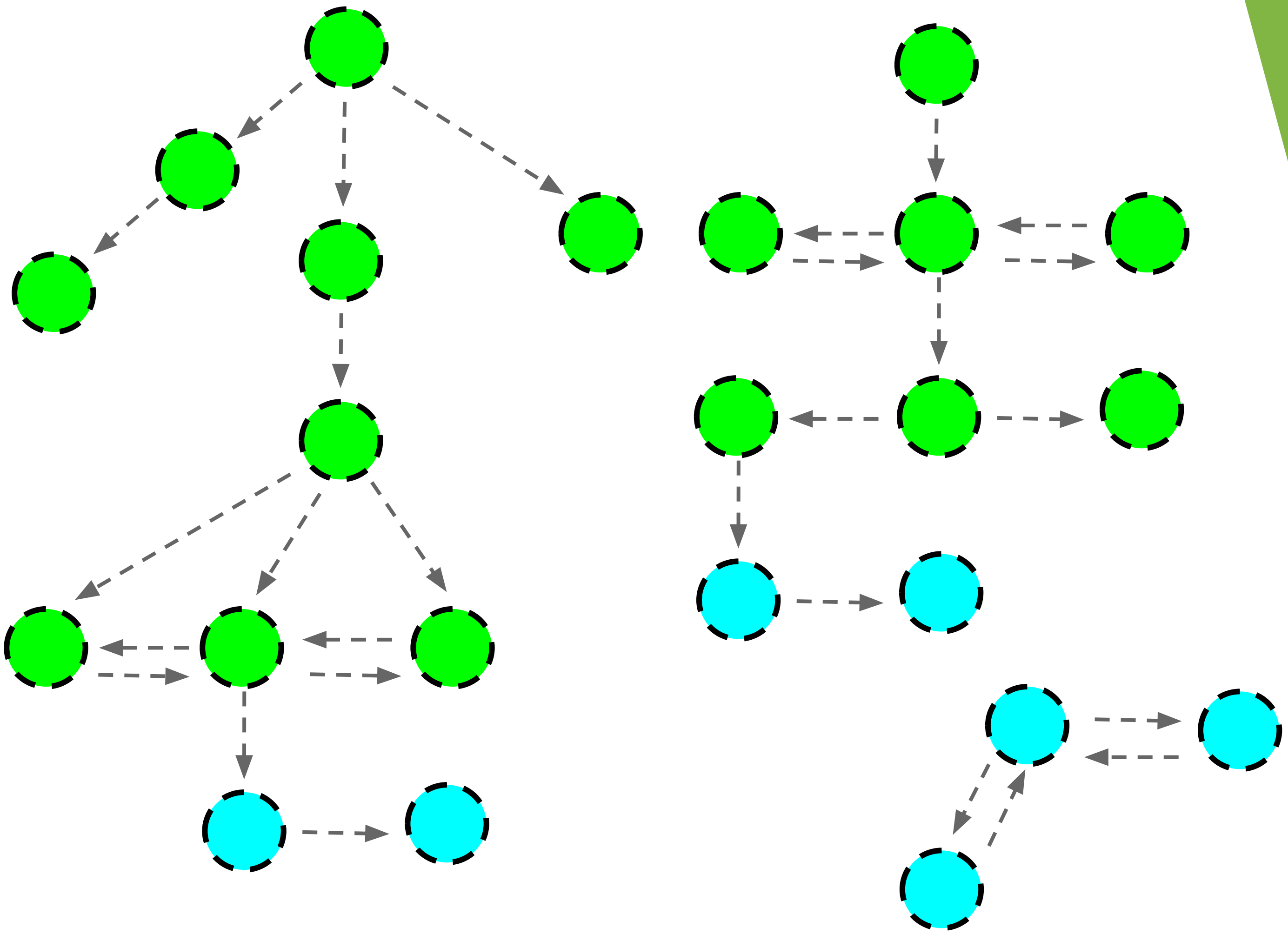
## The Future

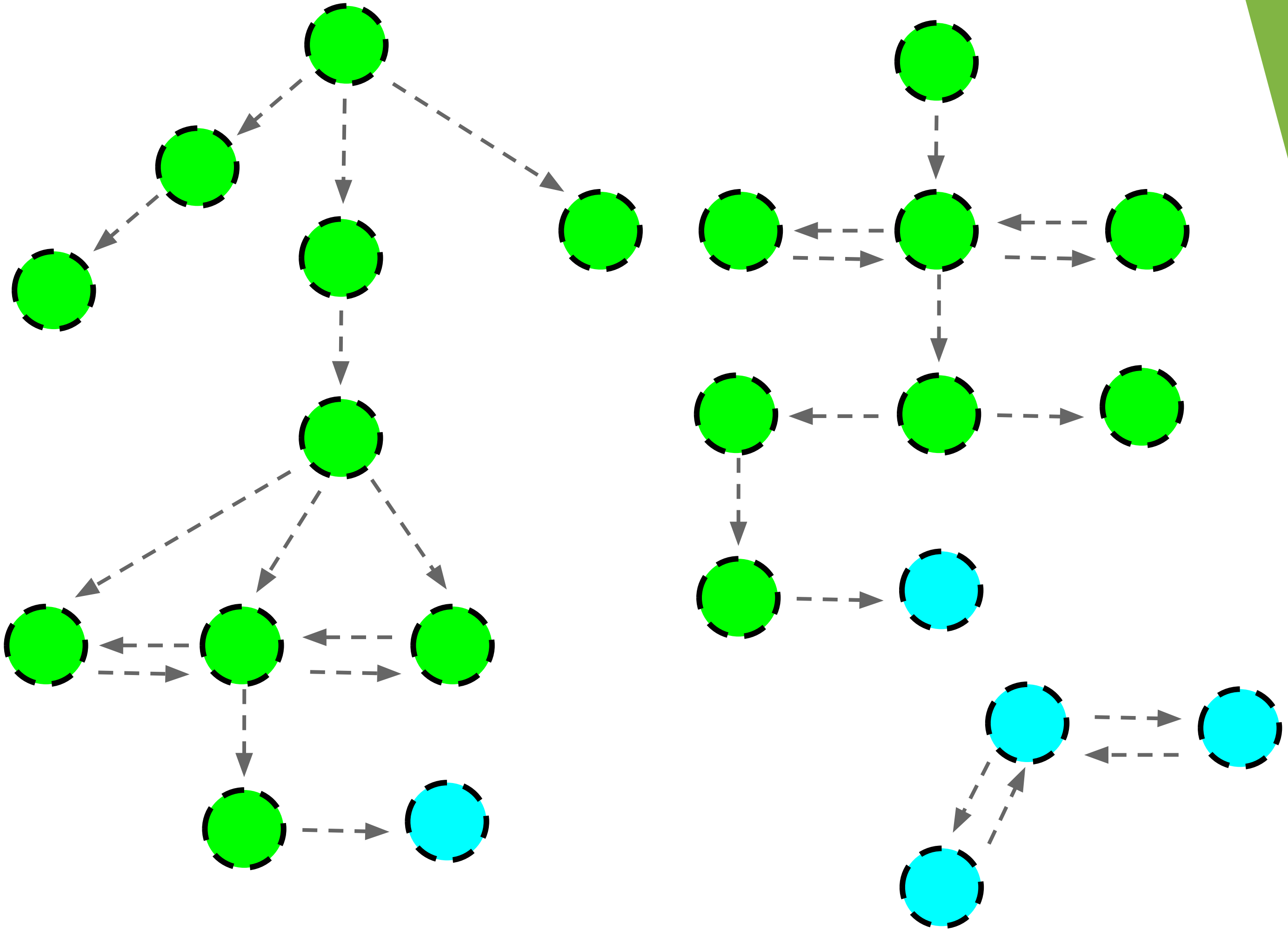
# Mark & Sweep

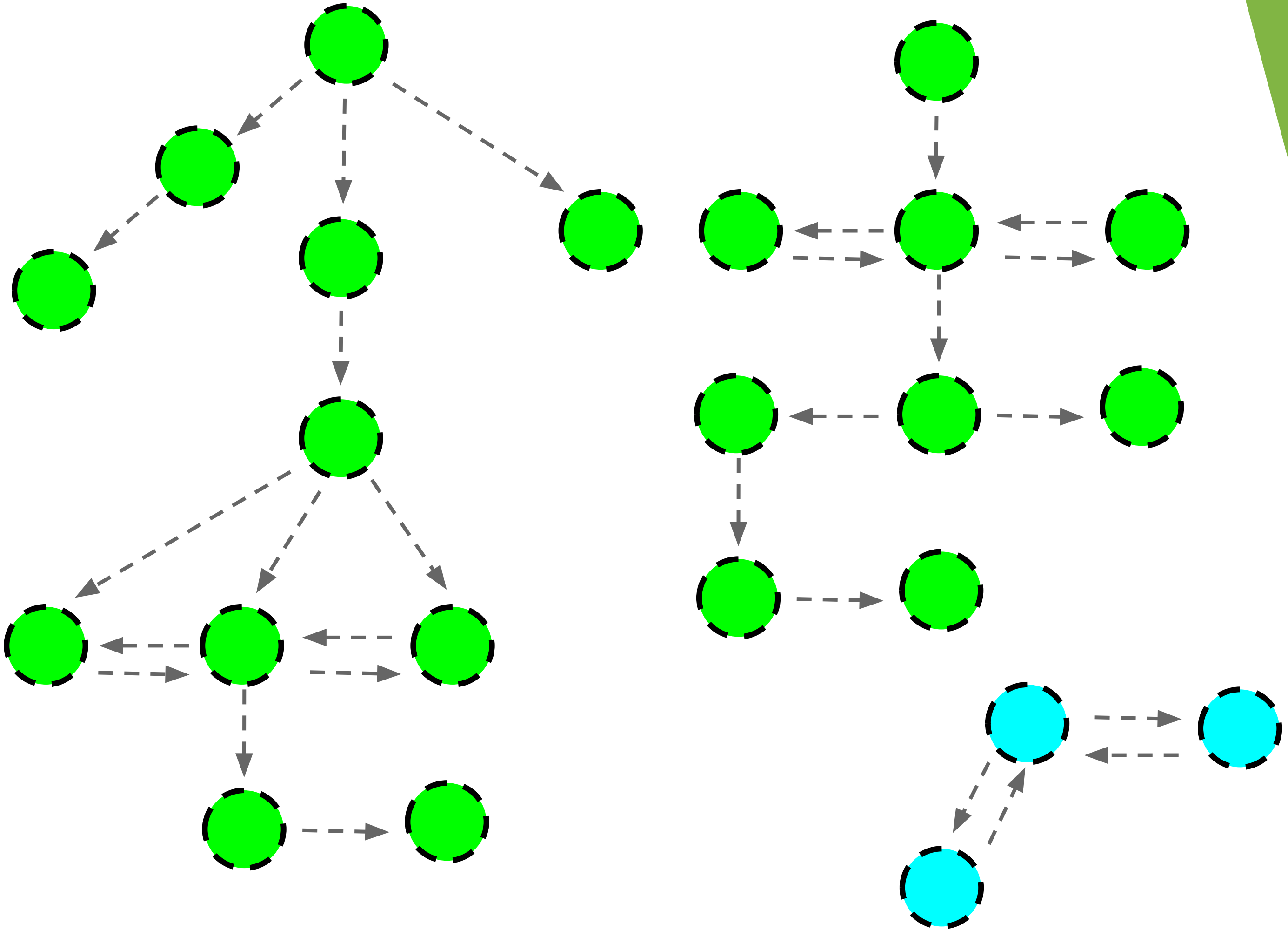


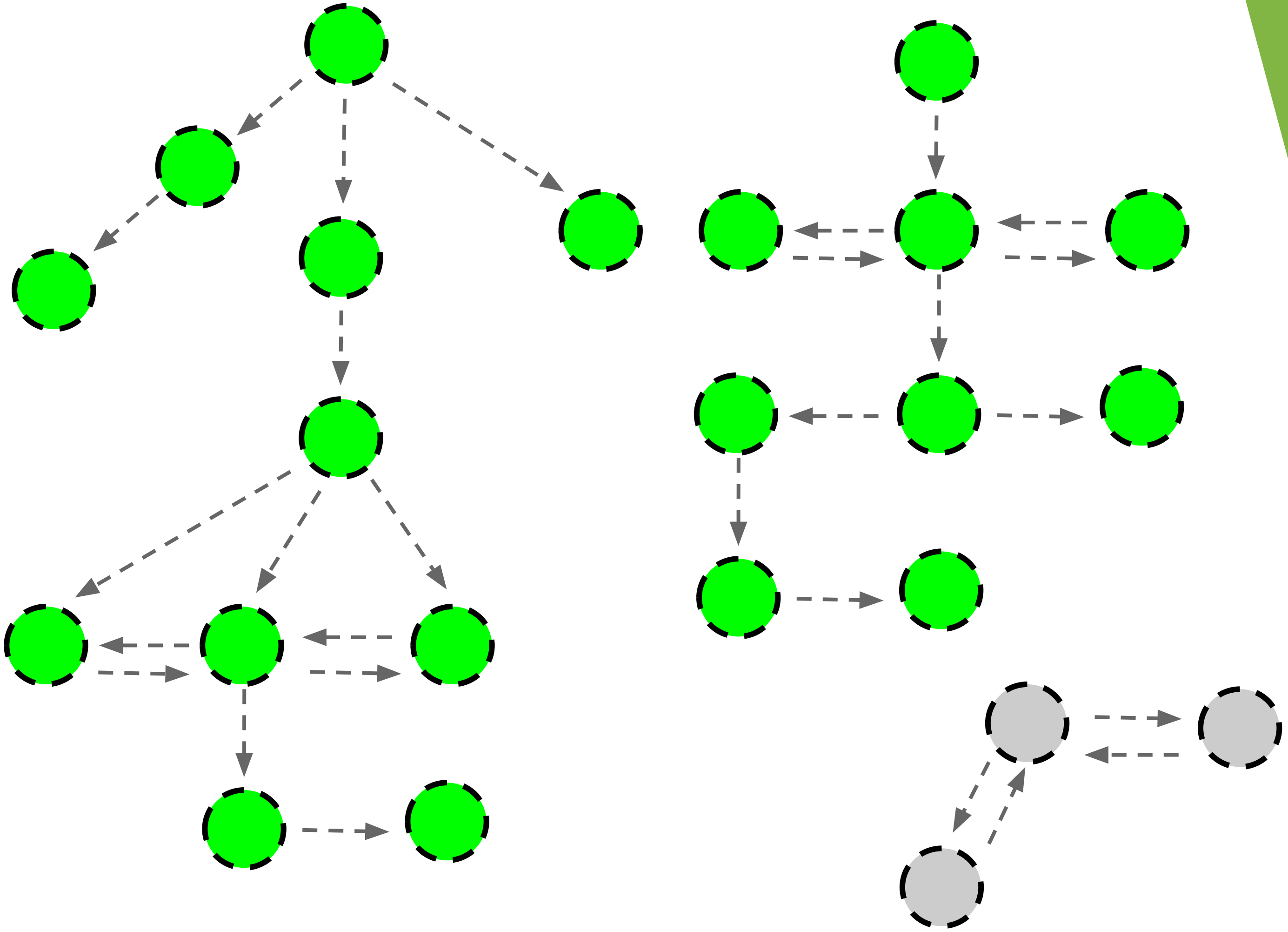


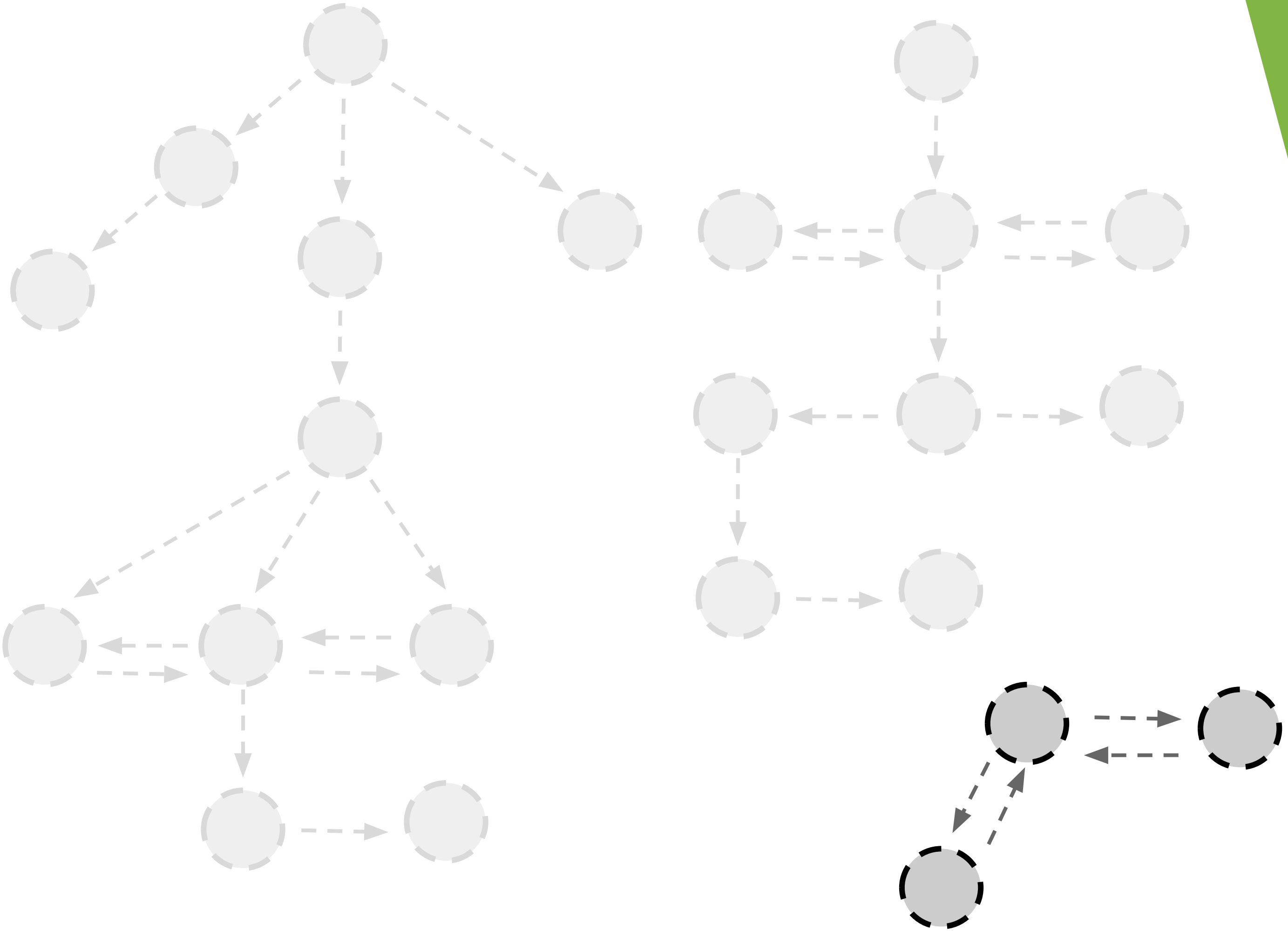


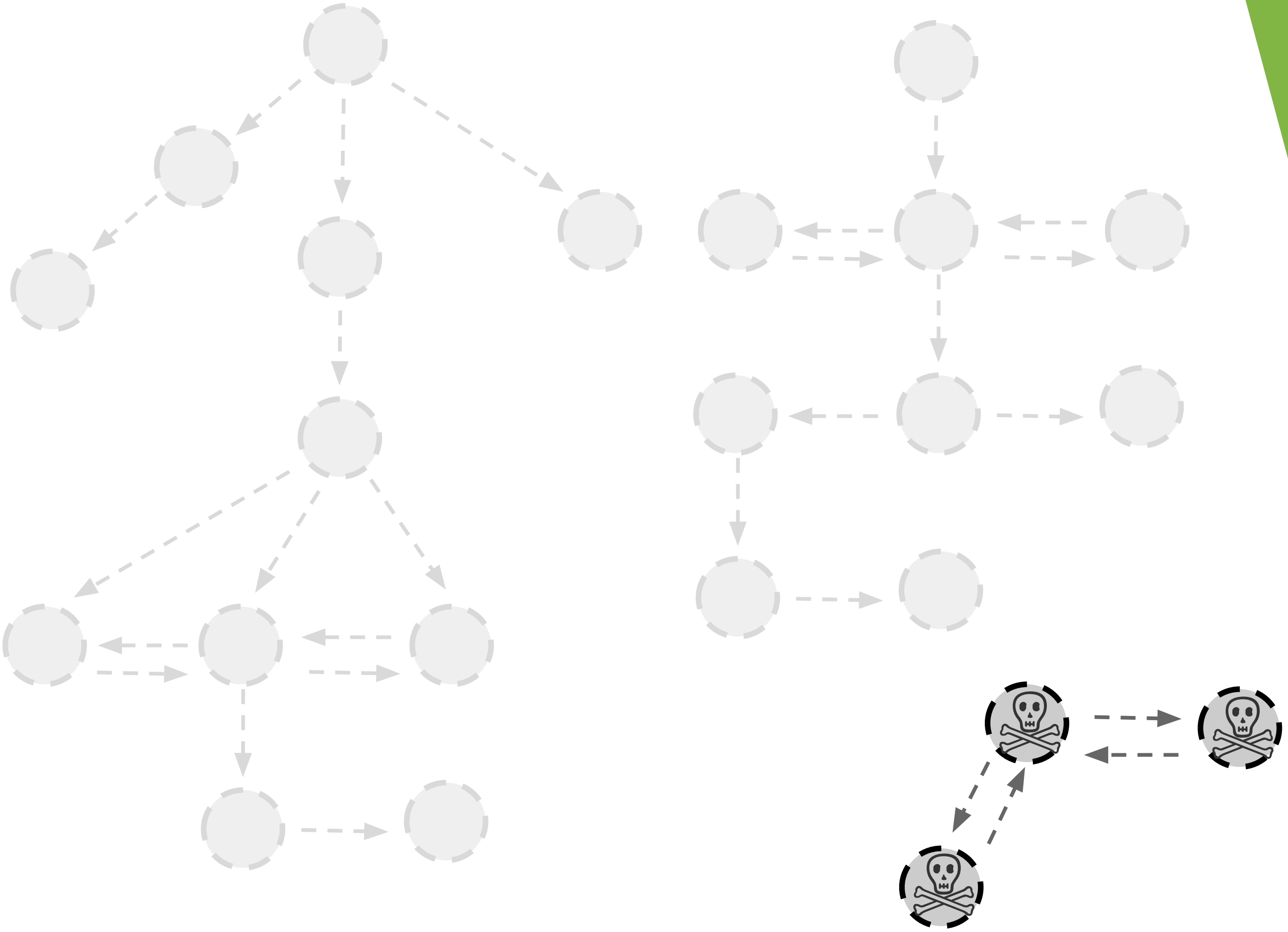




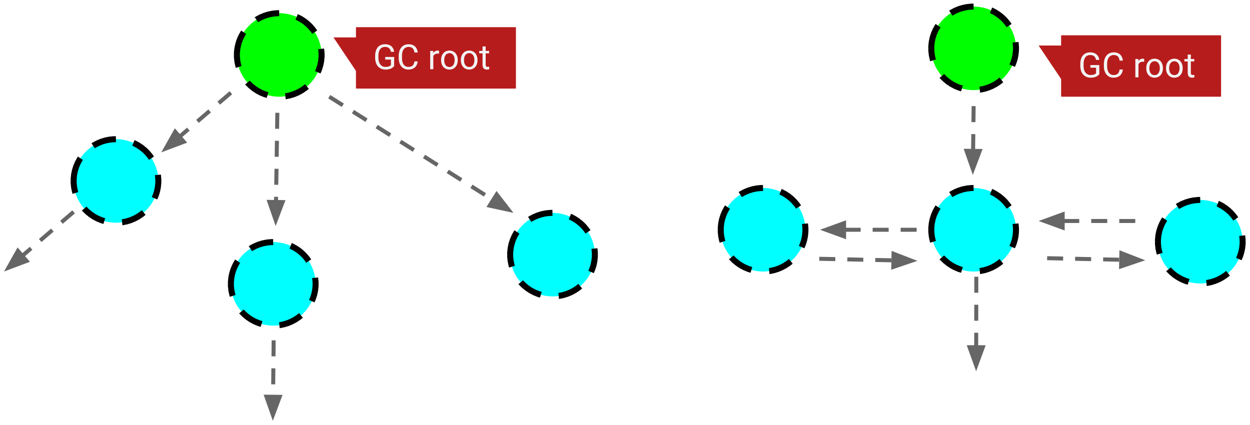








# The problem




- Finding the roots is very difficult.
- Maintaining compatibility with the C-API.
- Late garbage collection (memory can get very high).

# Recommended **resources**

https://rushter.com/blog/python-garbage-collector/

# Python: Taking the GARBAGE out

**Pablo Galindo**
*@pyblogsal*
Bloomberg

**Víctor Terrón**
*@pyctor*
Google