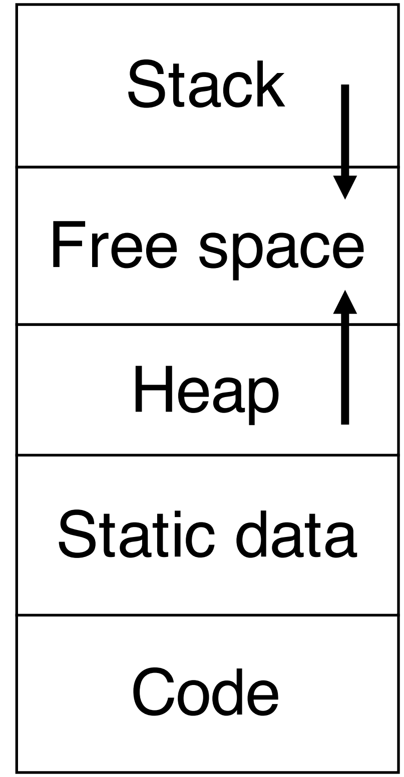
Heaps and Garbage

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Data Storage



Static area

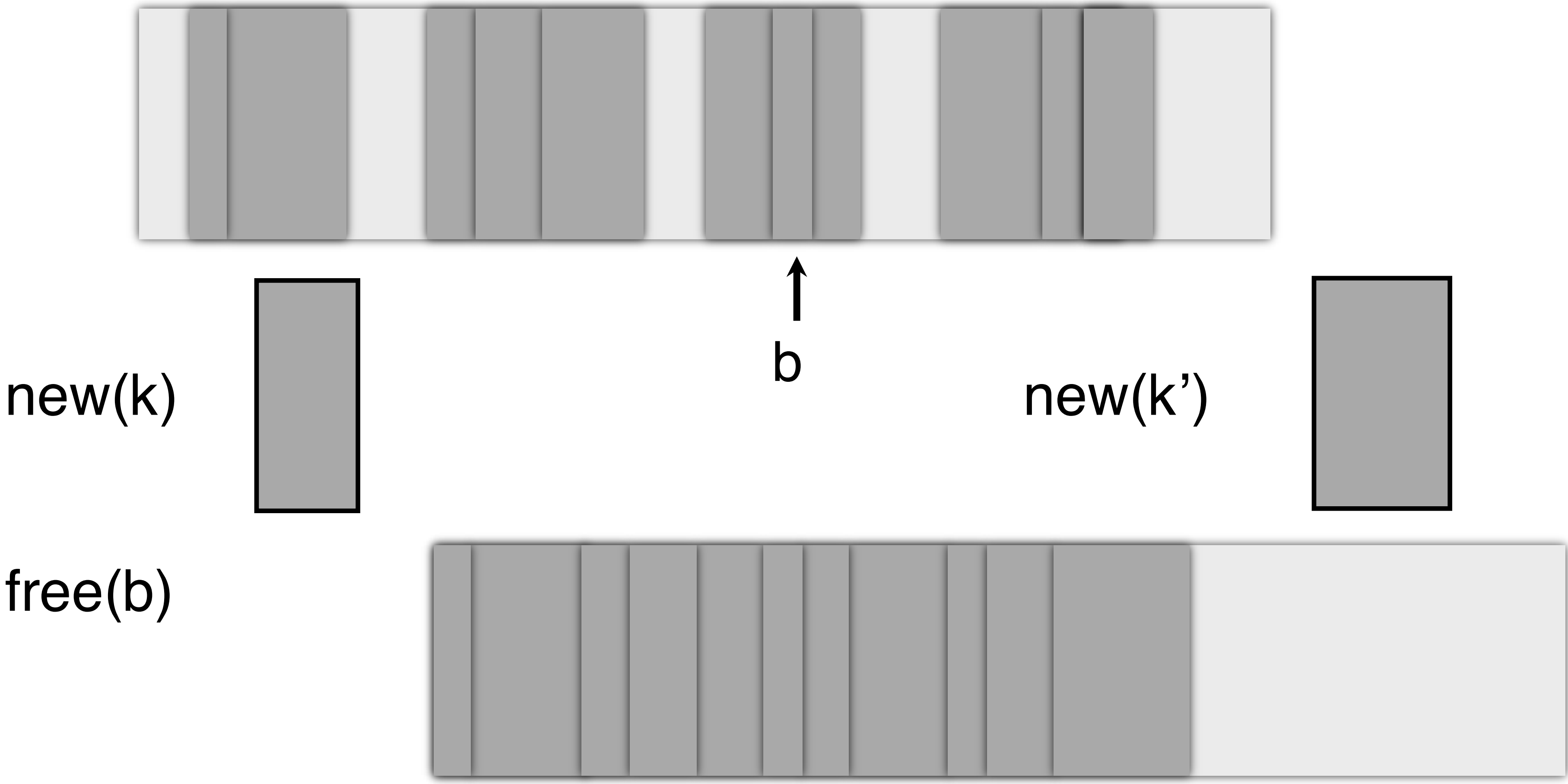
Stack

Operations: push, pop

Heap

Operations: new, delete, free, malloc

Heaps



Independent lifetimes of objects make heap management difficult

Heap Management

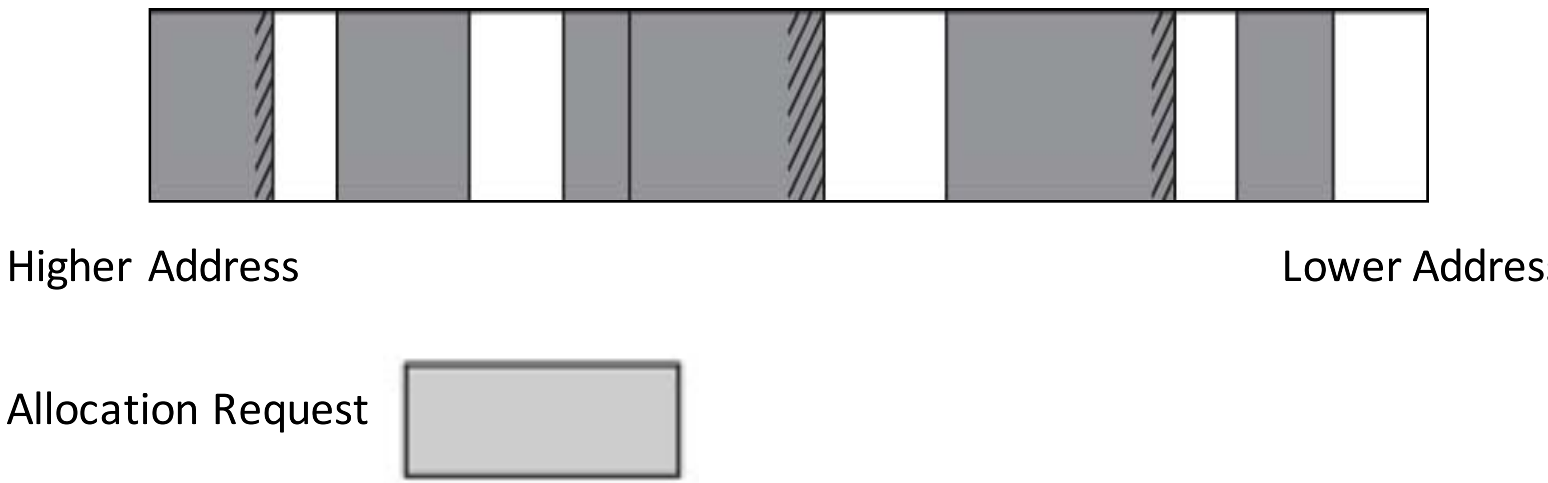
Allocator: a routine takes size of requested heap space, and search for free space

Usually, heap is managed in blocks. Allocator may return larger block than requested

Deallocator: collect free space and merge with other free space when possible

Heap compaction: move all used blocks to one end

Heap-Based Allocation



Internal fragmentation:

the allocation request is smaller than the assigned memory block

External fragmentation:

none of the scattered free space is large enough for the request

Heap Management Algorithms (not covered in this course)

First-fit: select the first free block that is large enough

Best-fit: select the smallest free block that fits

Buddy system: maintain various pools of free blocks with size of 2^k

Fibonacci heap: maintain various pools of free blocks with size of Fibonacci numbers

Heap Management

Programmer Management (C, C++)

- Pros: implementation simplicity, performance
- Cons: error prone (dangling pointers, memory leaks)

```
Node *p, *q;
p = new Node();
q = new Node();
q = p; // memory leaks
delete(p); // q becomes dangling pointer
```

Algorithms to detect dangling pointers:
Tombstones, Locks & Keys (Lec. 21)

Heap Management

Automatic Management (Java, Scheme)

- No dangling pointers, no memory leaks
- Cost: Slower than programmer management

Garbage Collection

Garbage: inaccessible heap objects

```
void foo () {
  int* a = new int[10];
  return;
}
```

Issues of heap management:

- Collect too aggressively: dangling pointers
- Collect too conservatively: memory leaks
- Key problem: collect only objects that are ***inaccessible*** from program

GC I: Reference Counting

Maintain a reference count with each heap object

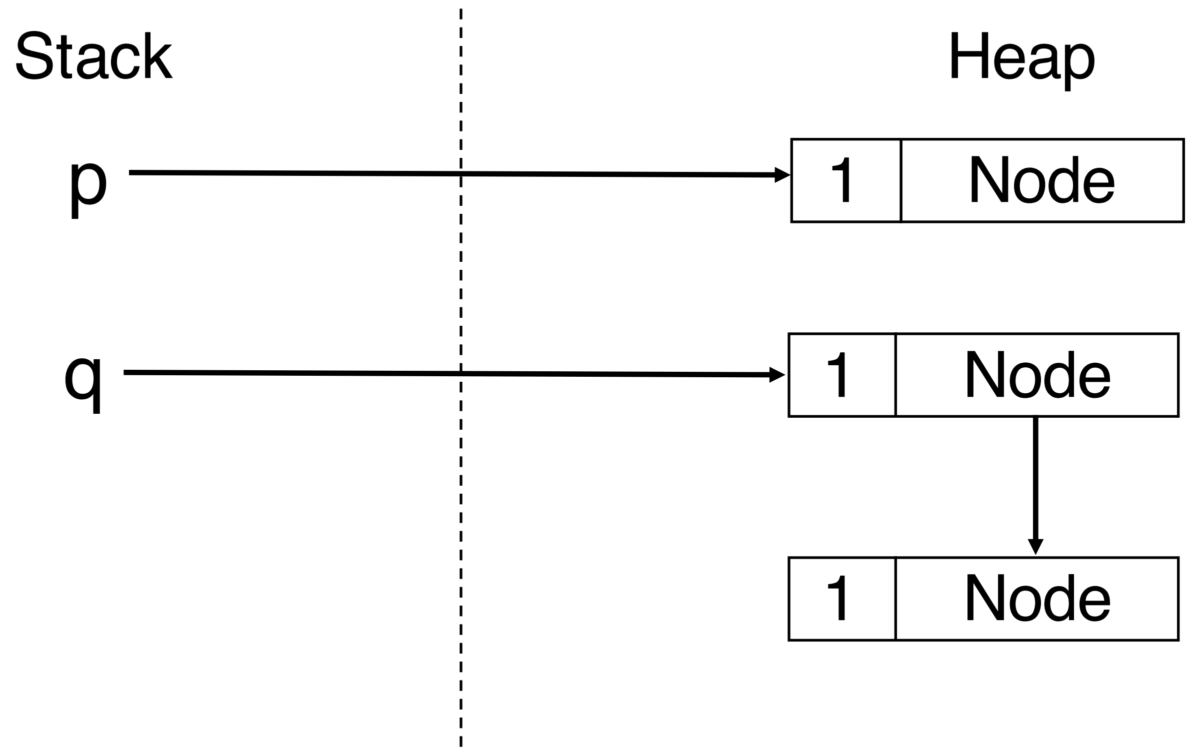
Set to 1 when object is created

Incremented each time new reference to it is created

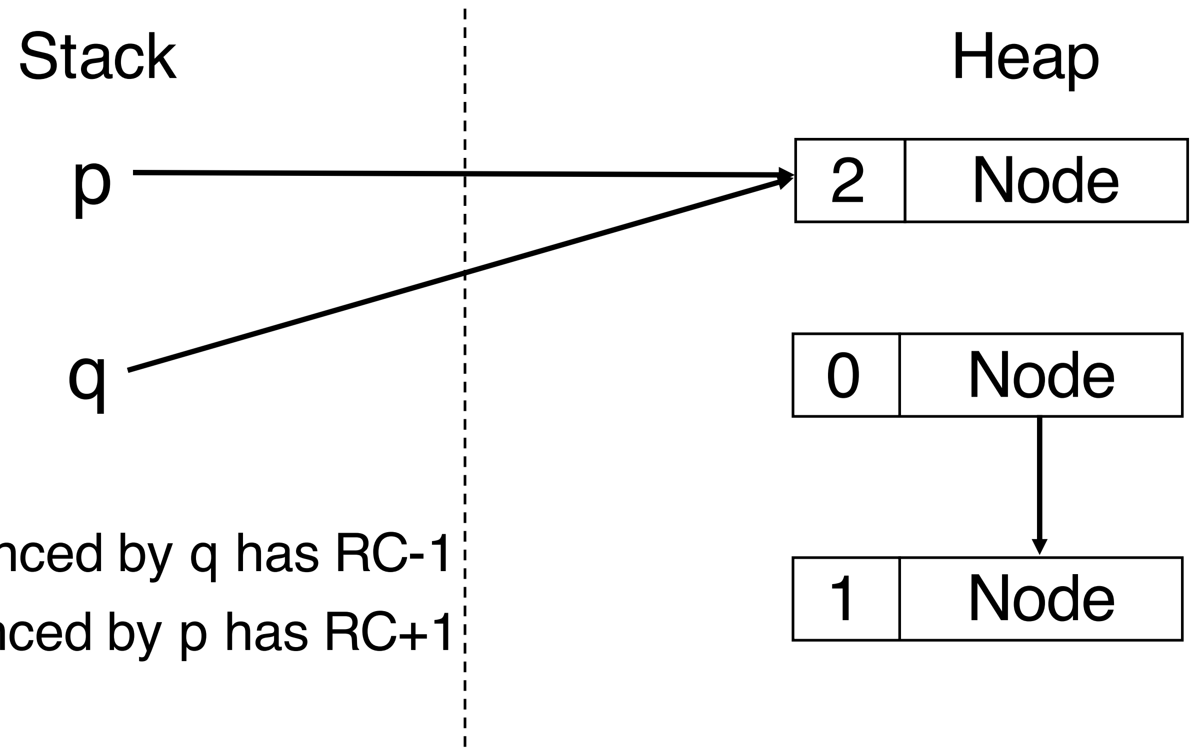
Decremented each time reference to it is deleted

Collect when count becomes 0

```
Node *p, *q;
p = new Node();
q = new Node();
q.next = new Node();
q = p;
p = null;
```



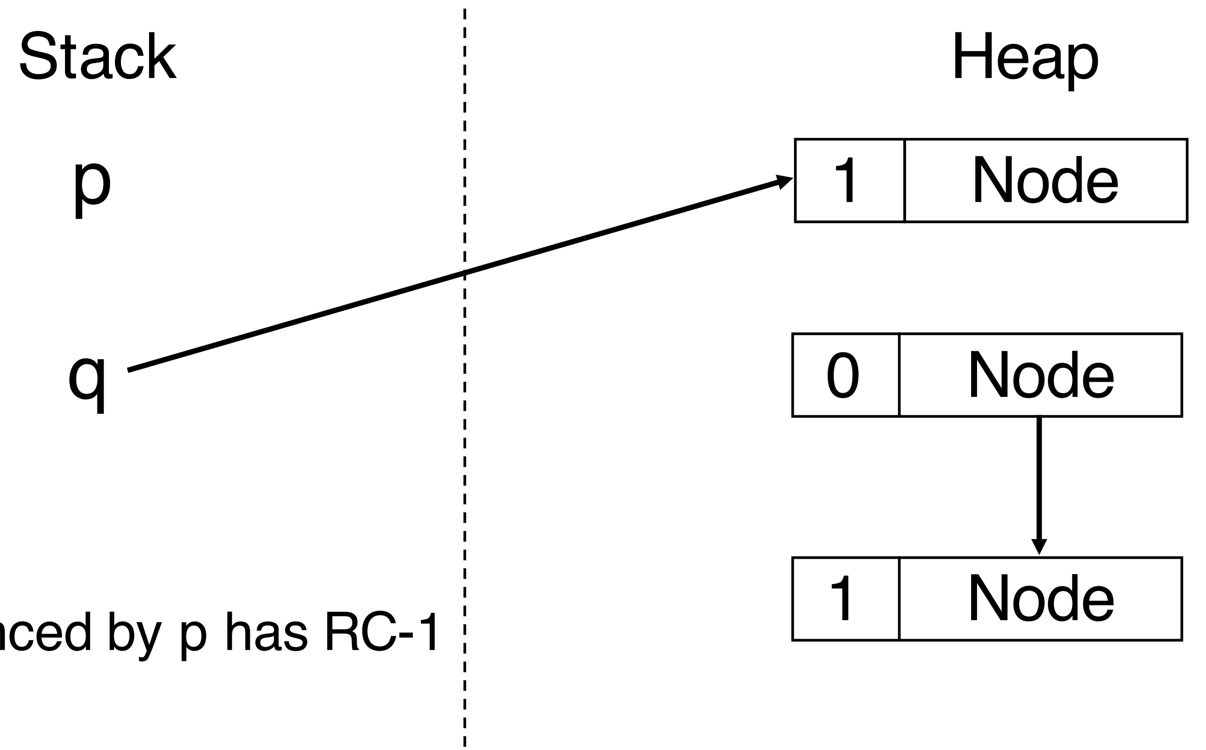
```
Node *p, *q;
p = new Node();
q = new Node();
q.next = new Node();
q = p;
p = null;
```



Object referenced by q has RC-1

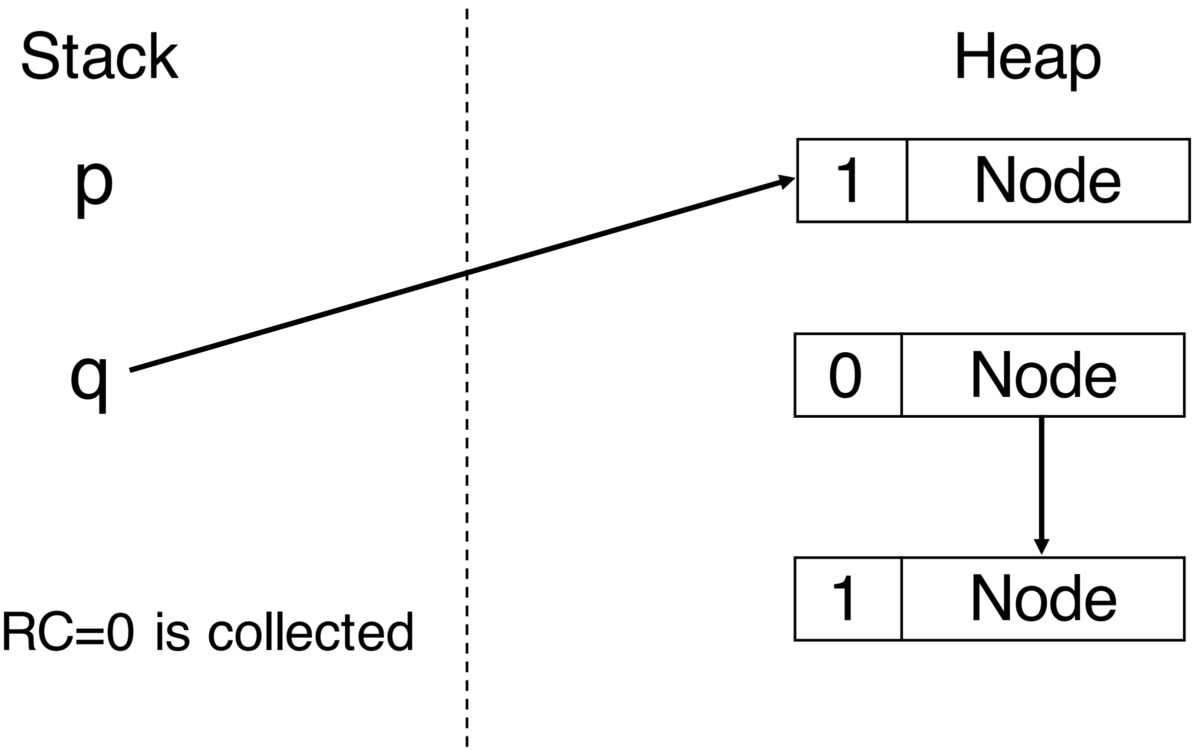
Object referenced by p has RC+1

```
Node *p, *q;
p = new Node();
q = new Node();
q.next = new Node();
q = p;
p = null;
```



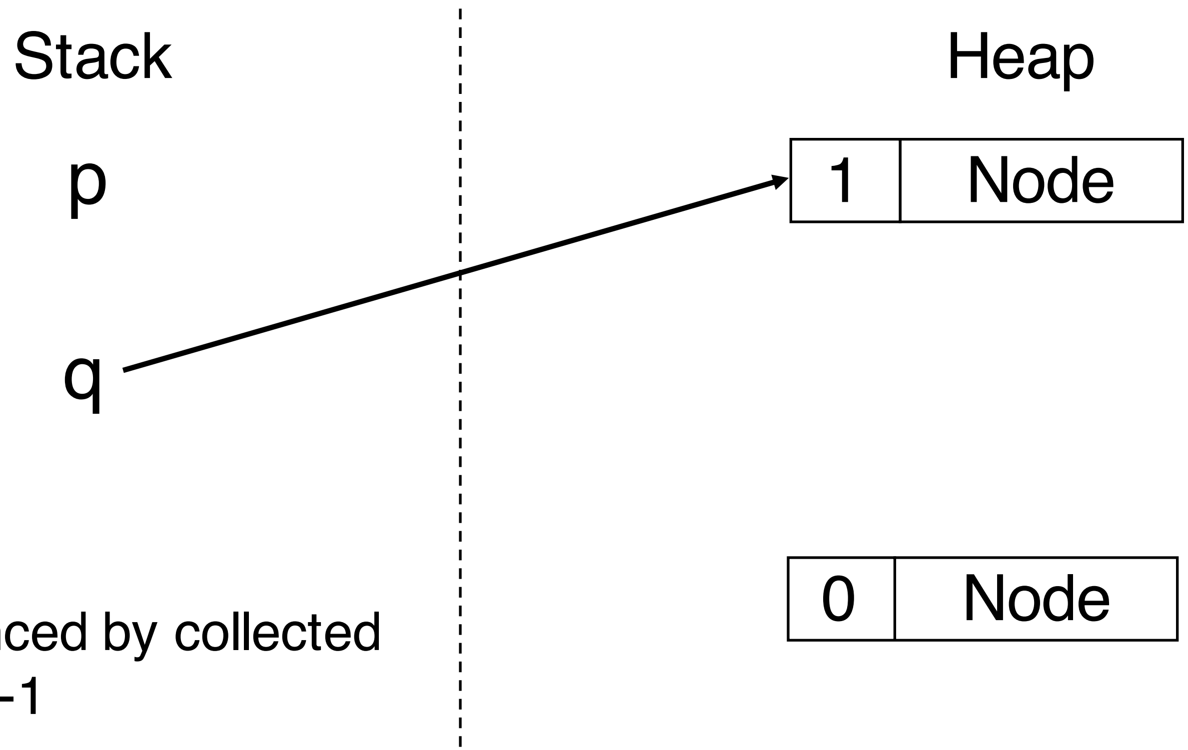
Object referenced by p has RC-1

Garbage Collection



Object where RC=0 is collected

Garbage Collection



Object referenced by collected object has RC-1

GC I: Reference Counting

When is an object dereferenced?

- Reference is LHS of Assignment
- Reference on stack is destroyed when function returns
- Reference is destroyed when an object with count 0 is collected

GC I: Reference Counting

Reference Counting is about **Object Ownership**

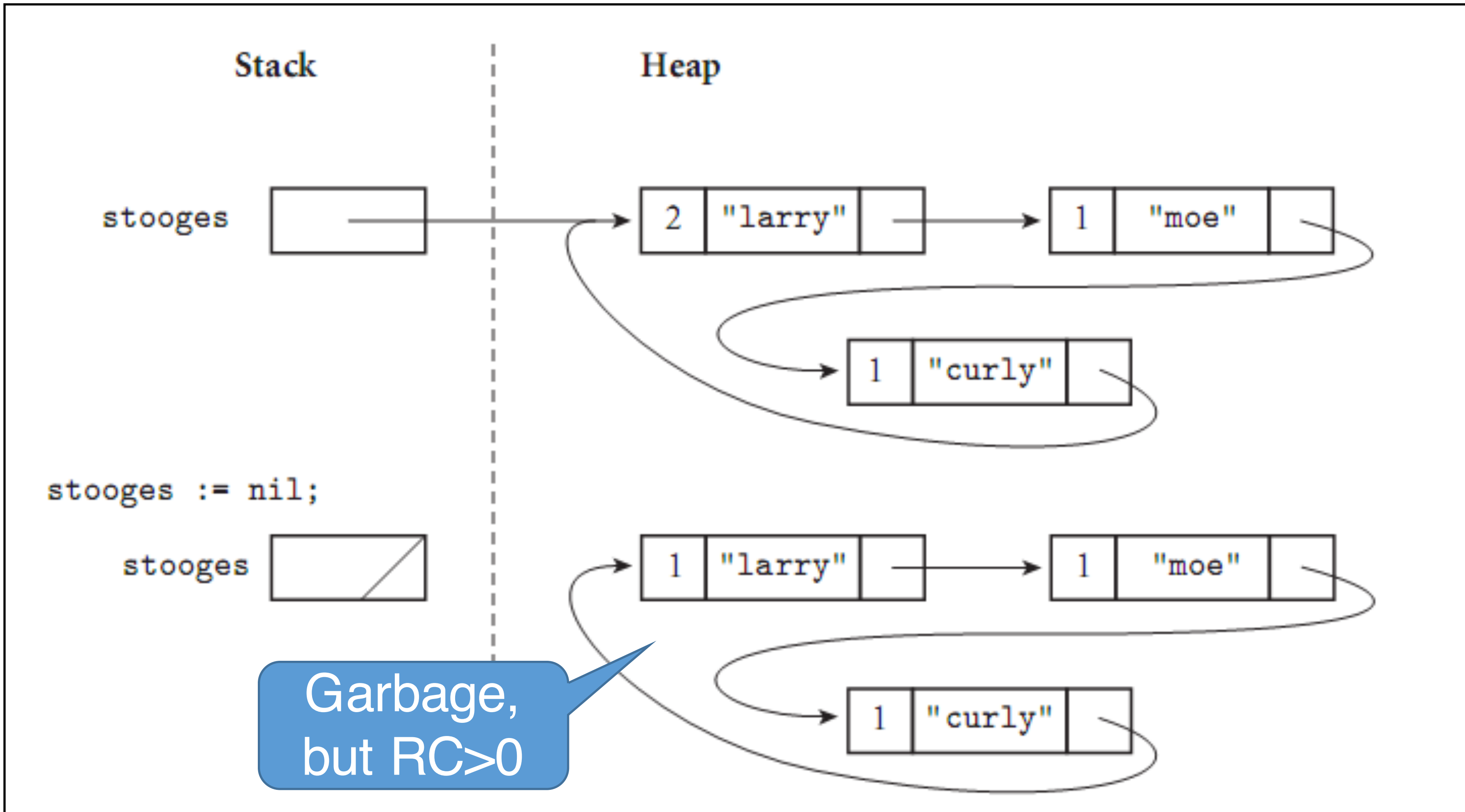
- when one object creates a reference to another object, it owns that object (retain)
- when the object deletes that reference, it relinquishes ownership (release)

Multiple owners of an object

Zero owners of an object

Problem of Reference Counting

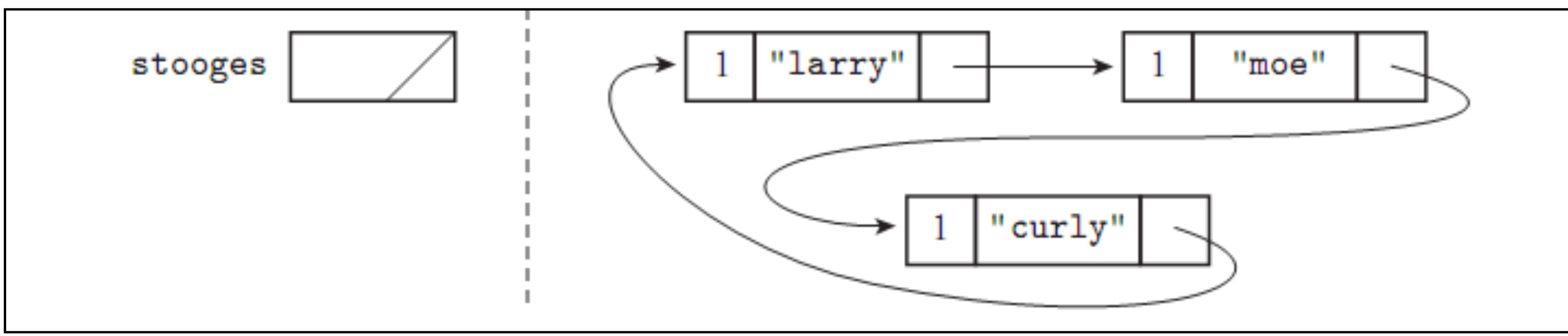
Ownership ≠ Accessibility



What Is Garbage

Ideally, any heap block not used in the future

In practice, the garbage collector identifies blocks inaccessible from program



Essentially a reachability problem (from alive variables)

We will see such algorithms in the next lecture