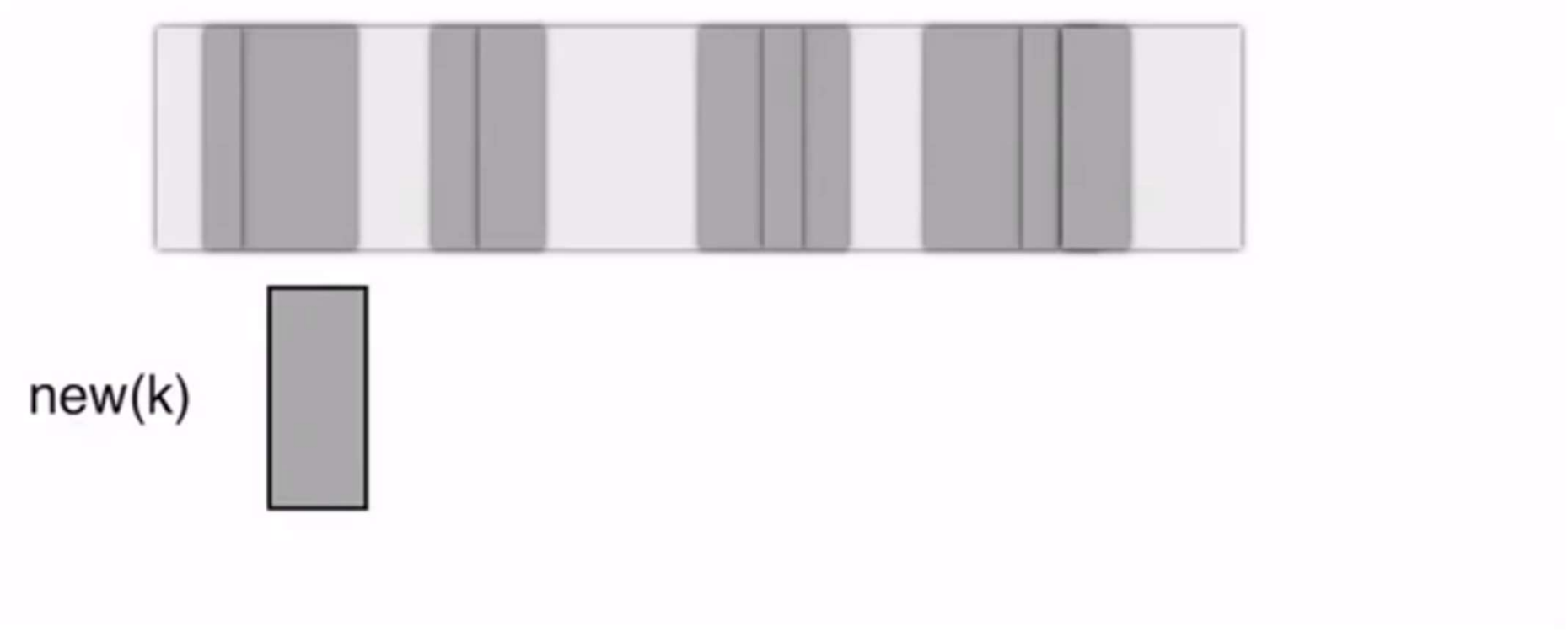
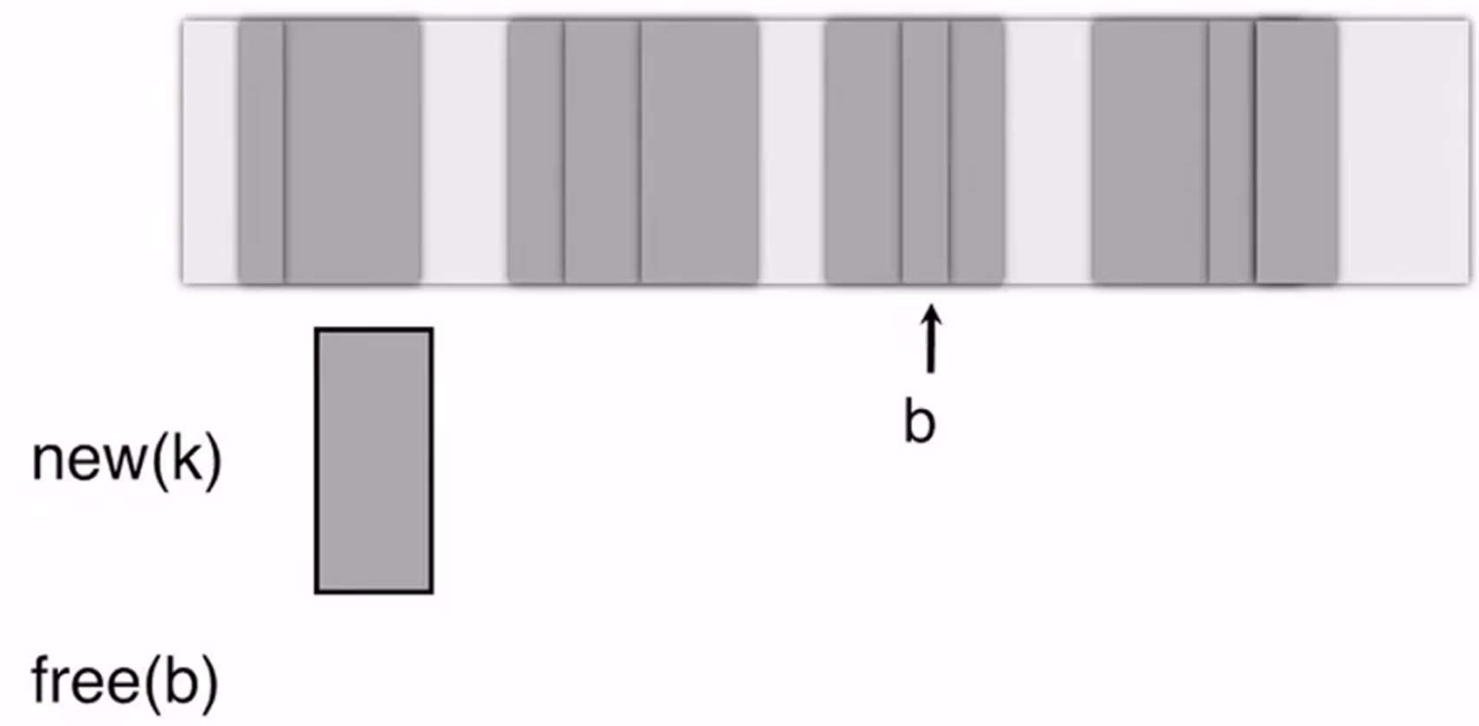
Memory Management

Professor: Suman Saha

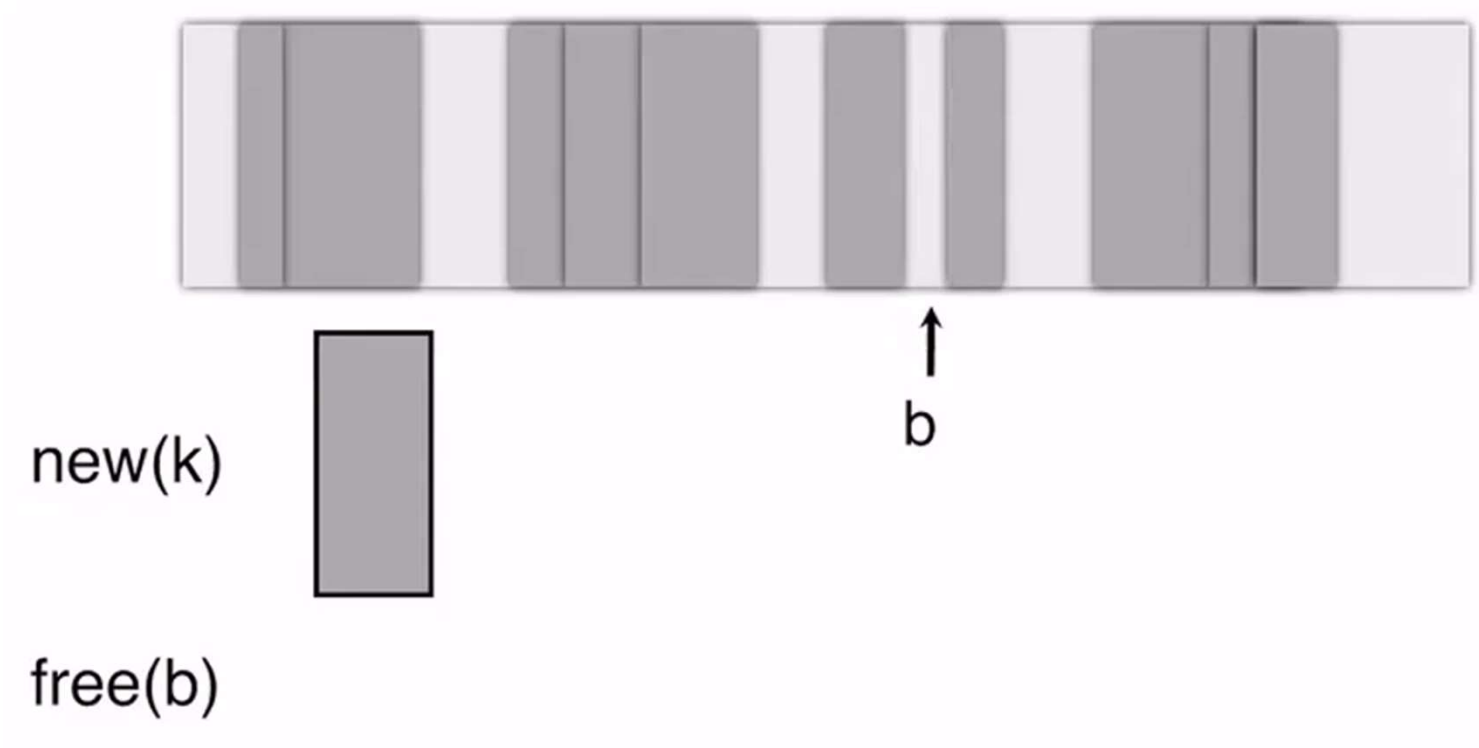
Heaps



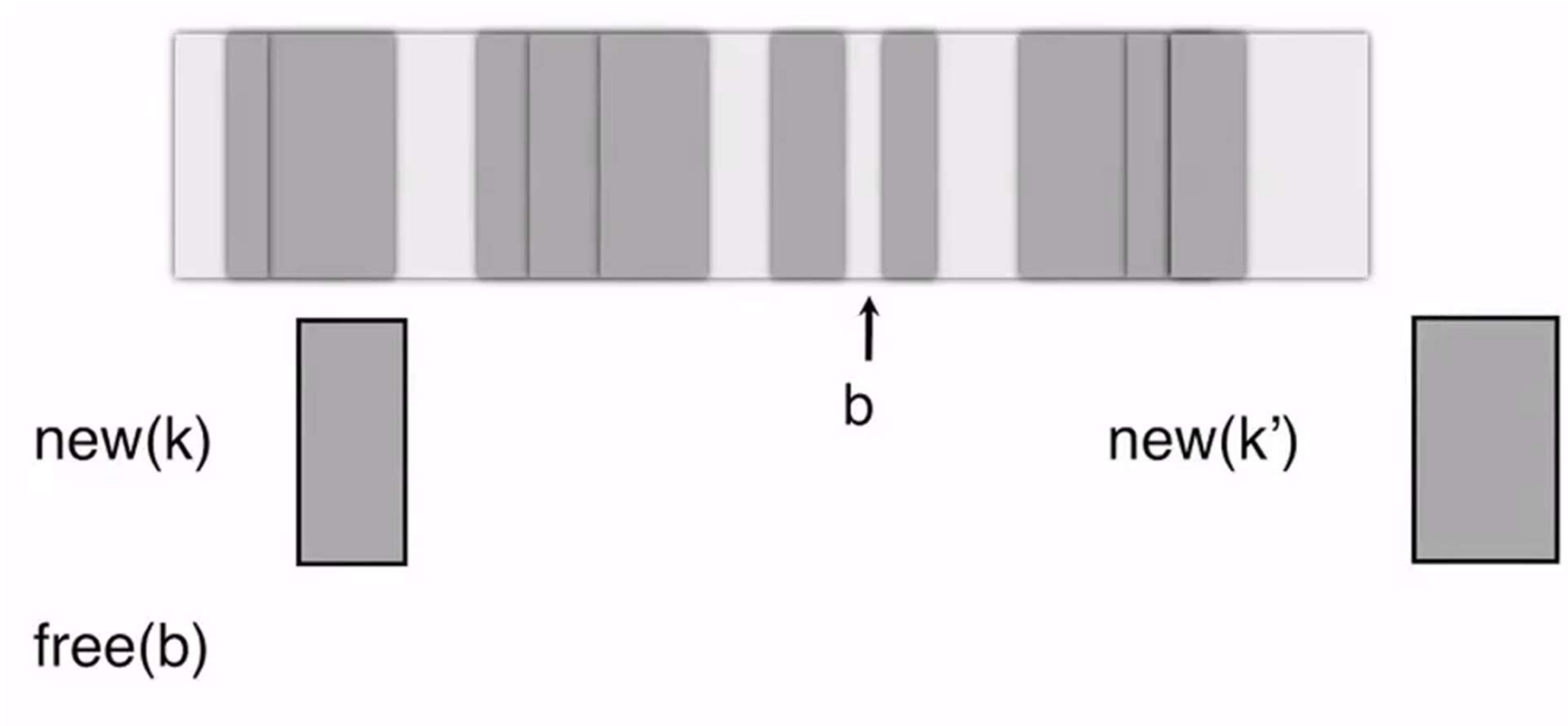
Heaps



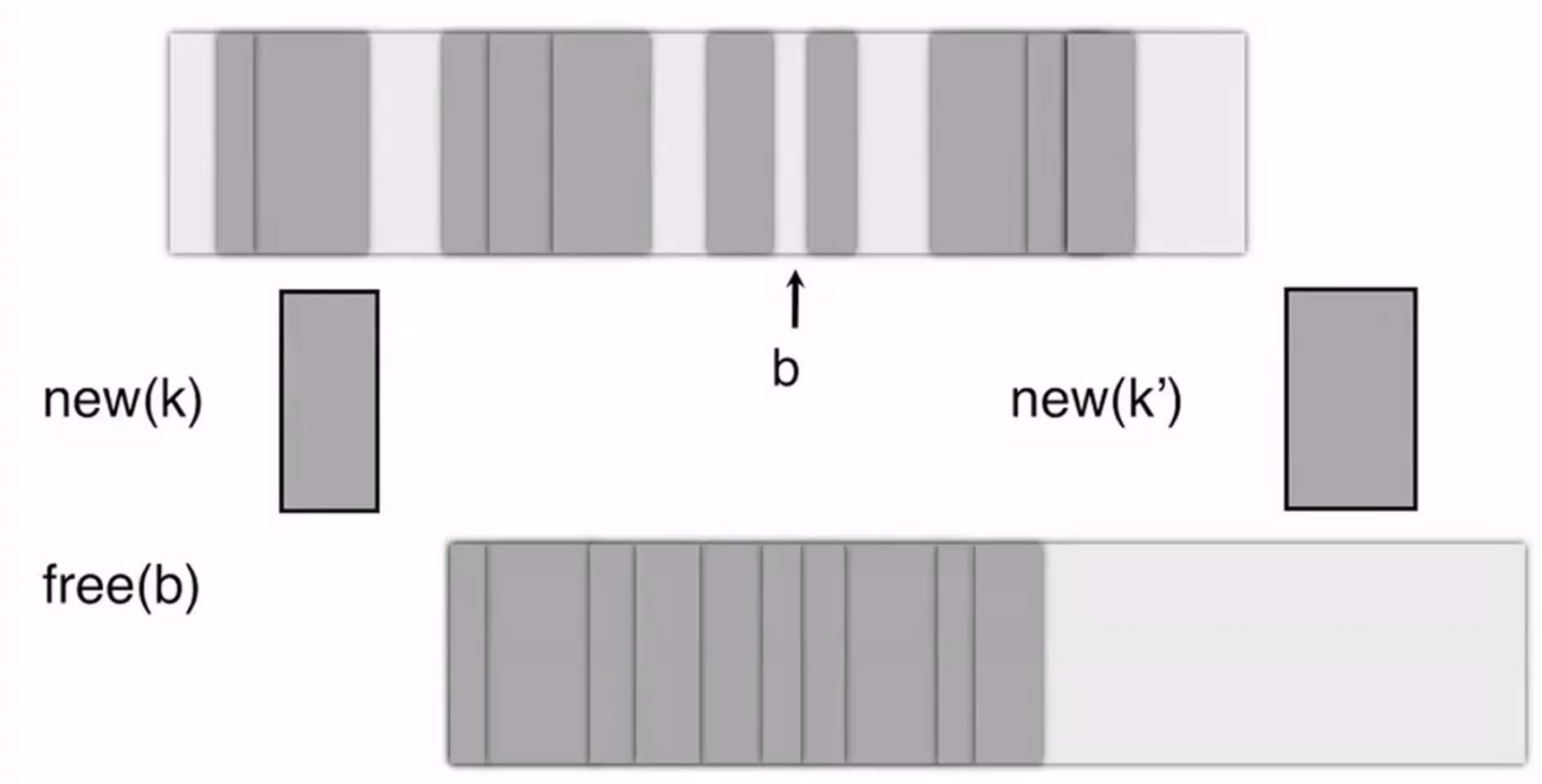
Heaps



Heaps



Heaps



Heap Management

PennState

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

Higher Address Lower Address

Allocation Request 

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)

PennState

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

PennState

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
```

Heap Management

PennState

Automatic Management (Java, Scheme)

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

Garbage Collection

PennState

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

PennState

Maintain a reference count with each heap object

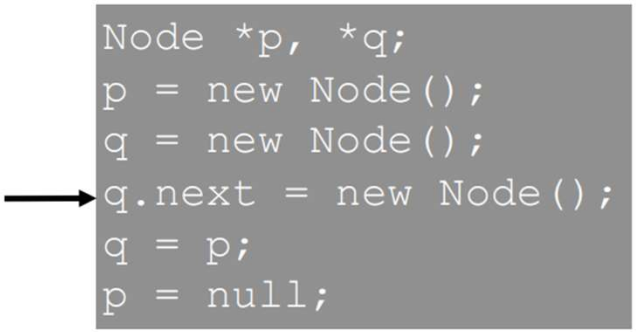
Set to 1 when object is created

Incremented each time new reference to it is created

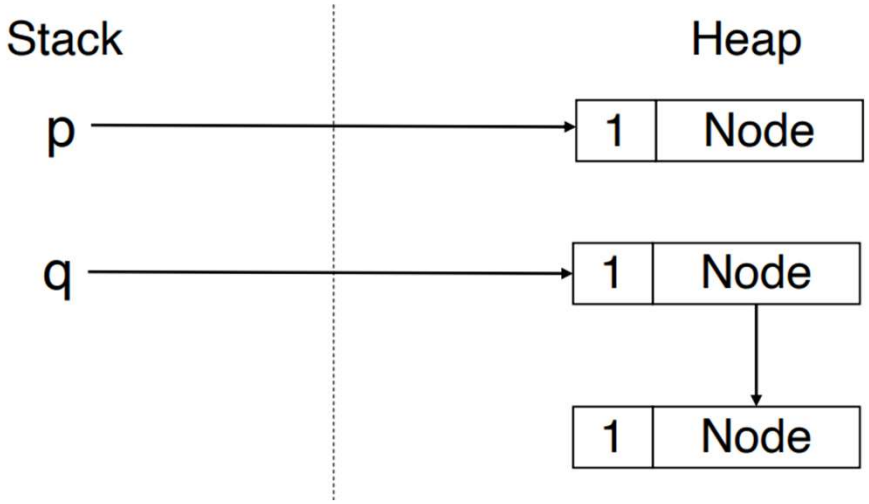
Decremented each time reference to it is deleted

Collect when count becomes 0

GC I: Reference Counting



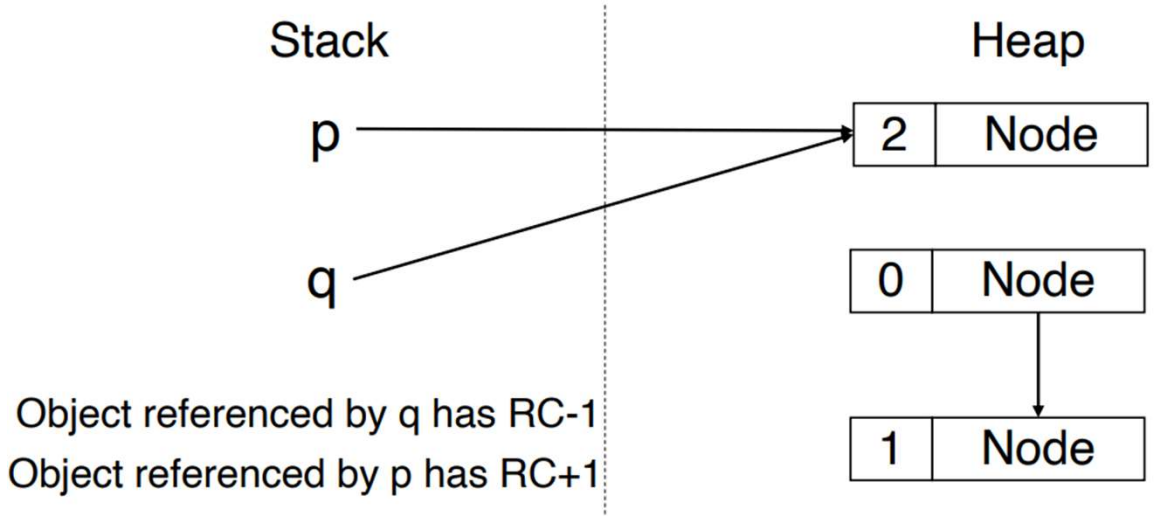
```
Node *p, *q;
p = new Node();
q = new Node();
q.next = new Node();
q = p;
p = null;
```



GC I: Reference Counting

PennState

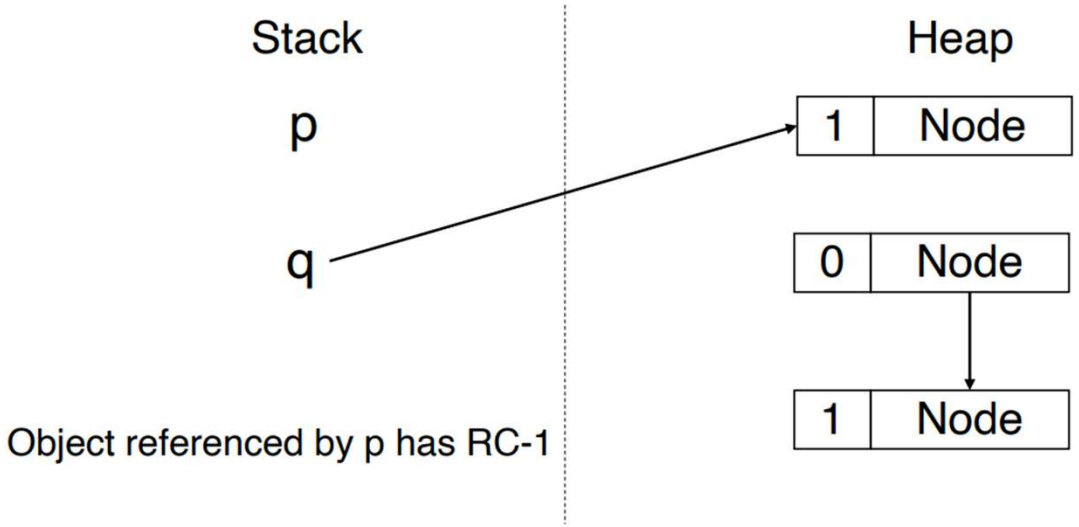
```
Node *p, *q;
p = new Node();
q = new Node();
q.next = new Node();
q = p;
p = null;
```



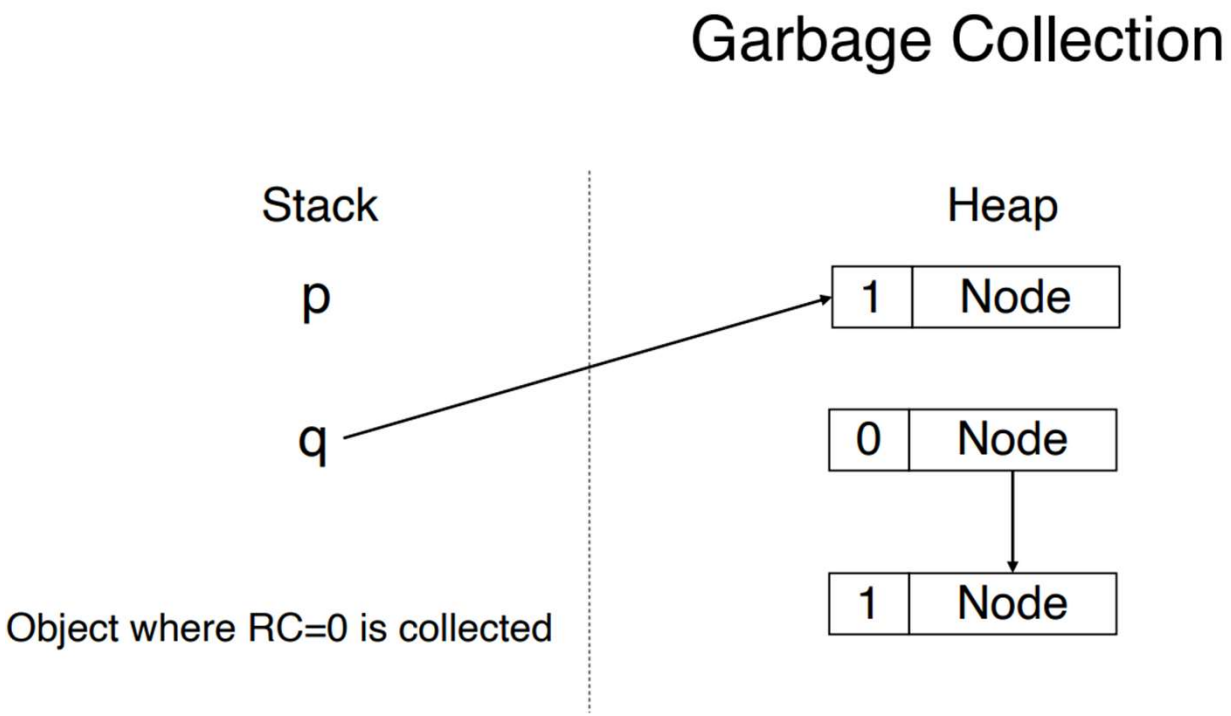
GC I: Reference Counting

PennState

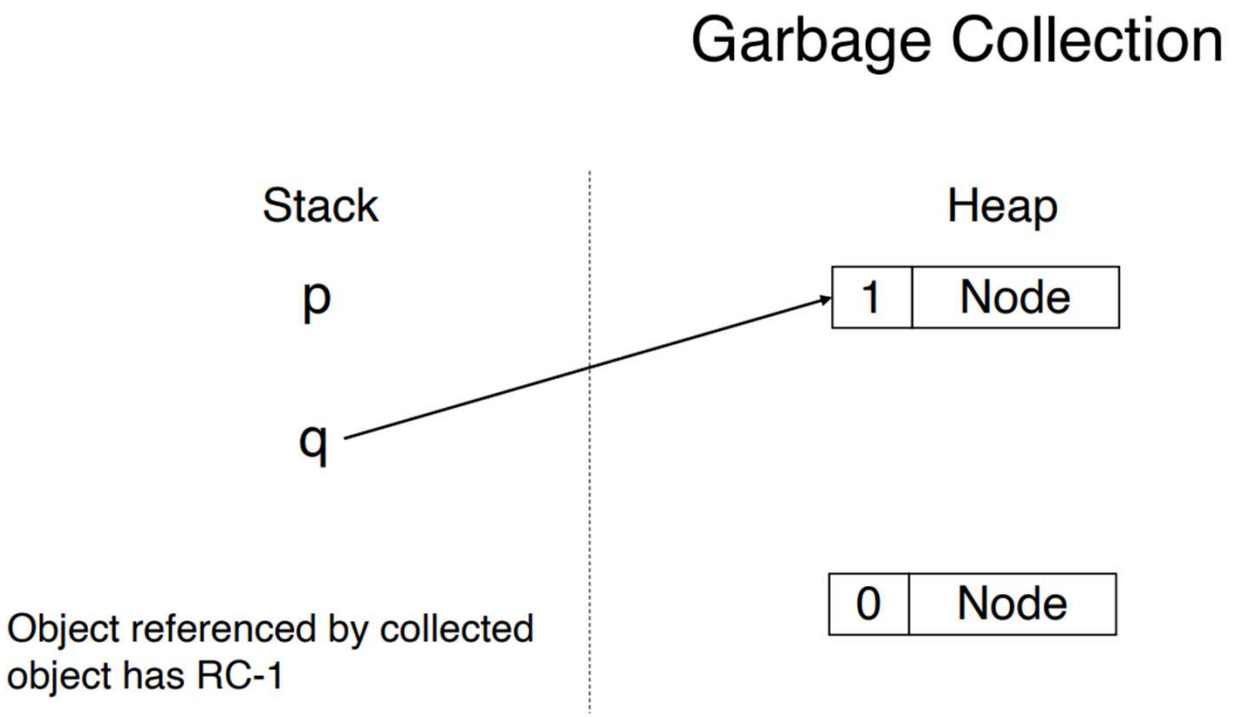
```
Node *p, *q;
p = new Node();
q = new Node();
q.next = new Node();
q = p;
p = null;
```



GC I: Reference Counting



GC I: Reference Counting



GC I: Reference Counting

PennState

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

PennState

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)

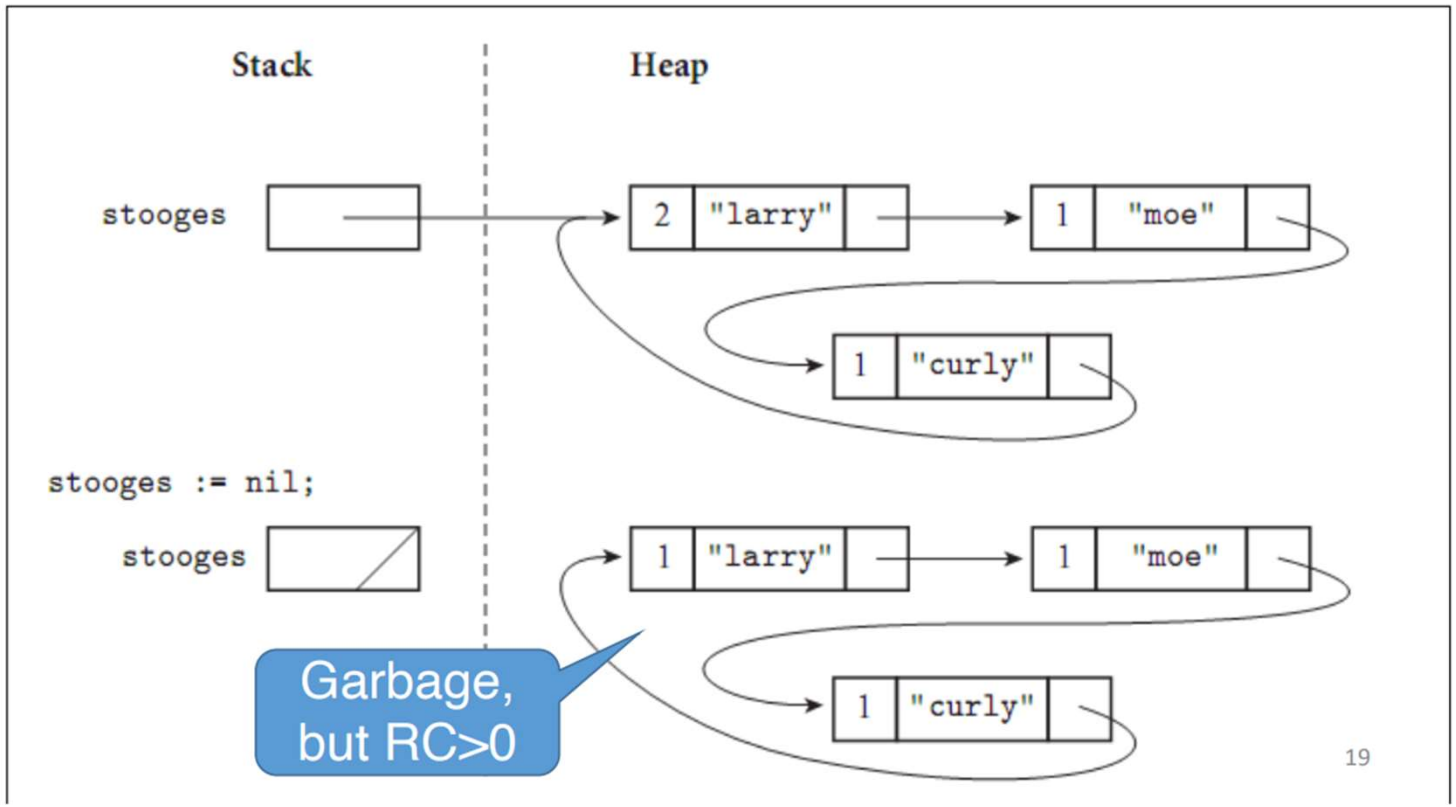
Multiple owners of an object

Zero owners of an object

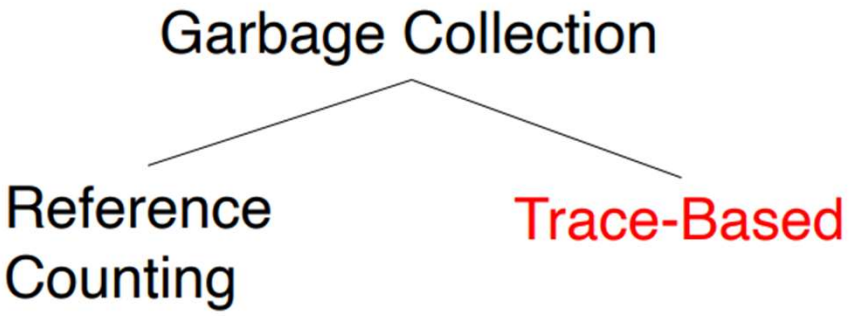
Problem of Reference Counting

PennState

Ownership ≠ Accessibility



Taxonomy

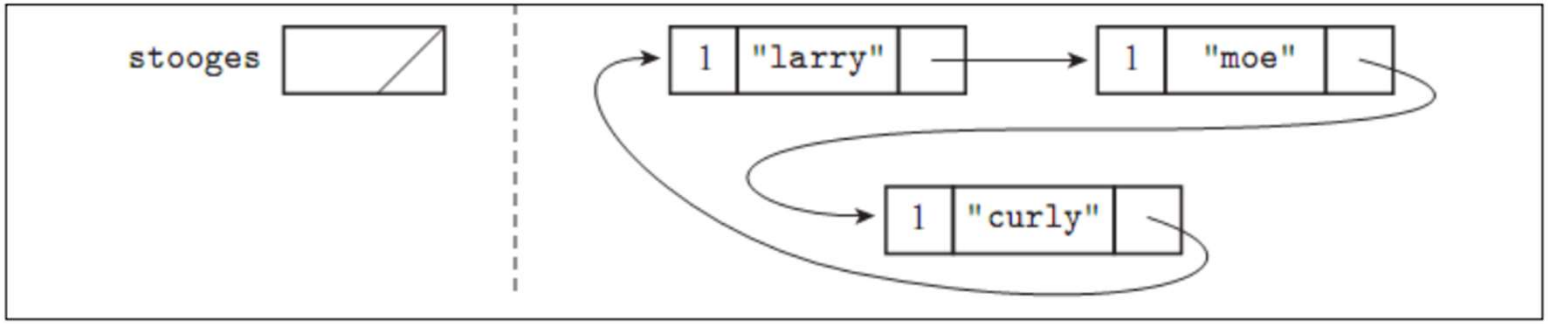


What is Garbage?

PennState

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)

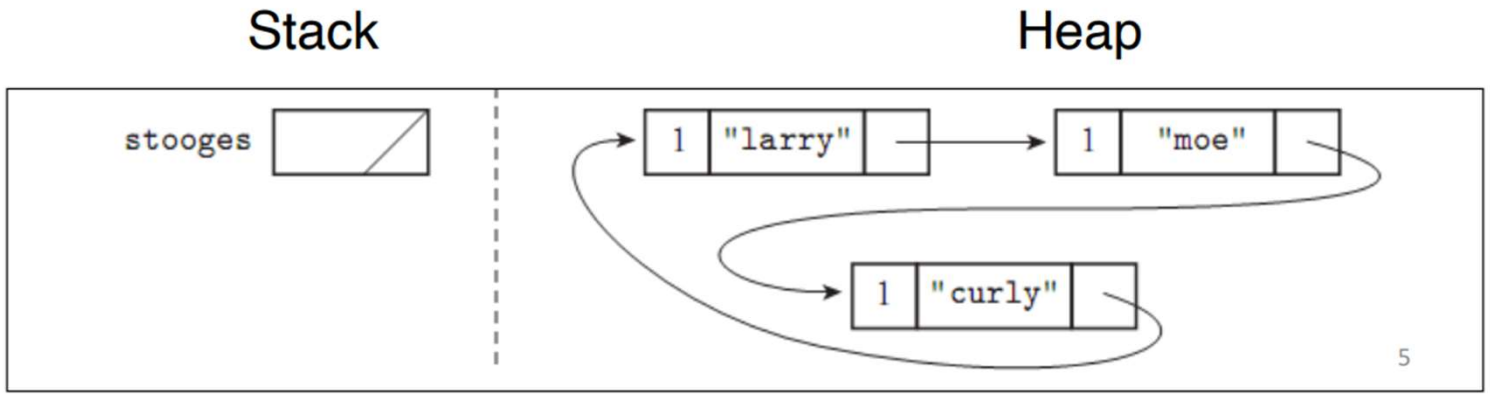
GC II: Tracing Collection

PennState

What allocated blocks are still accessible (live)?

- Reference in registers, static area, or stack
- Reference from reachable objects to other objects

Essentially a reachability problem (heap as directed graph)



Abstraction of Tracing

PennState

Root set: references in registers, static area or stack

Heap graph: node→object, edge→pointer/reference

Accessible set = Reachable object from root set

Overview

PennState

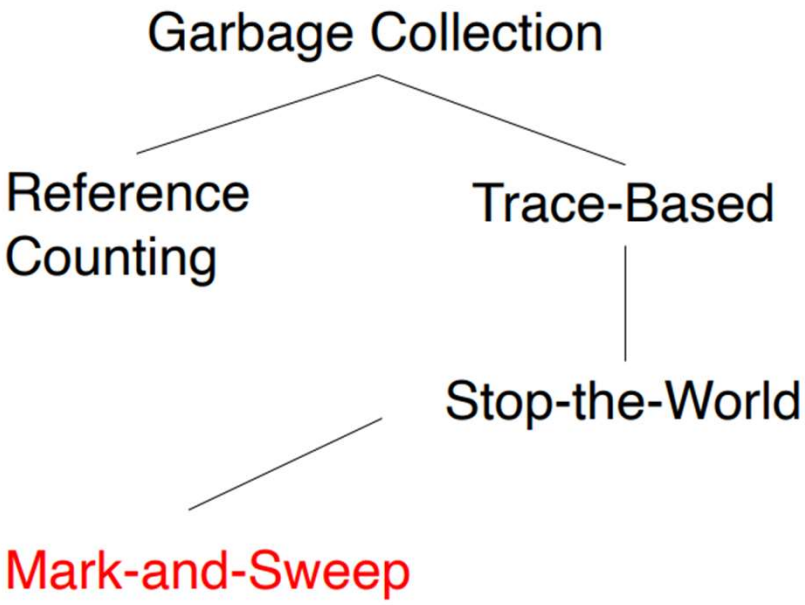
Essentially a reachability problem (heap as directed graph)

- Start from references in registers/stack/static area
- Traverse the graph to identify reachable blocks
- Remove unreachable blocks

Tradeoffs

- Time overhead - % of time spent on collection
- Space overhead – memory needed for collection algorithm
- Pause time – the time when a program is interrupted
- Frequency of collection
- Promptness – time between a block becomes garbage and it is collected

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Mark-and-Sweep

Mark-And-Sweep

PennState

Triggered when the heap is full, interrupt program

1 Mark Bit (MB) for each object, initially 0

Phase 1 (Mark)

- Traverse graph, set MB to 1 for visited node

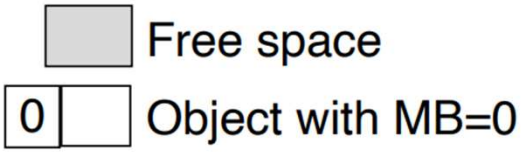
Phase 2 (Sweep)

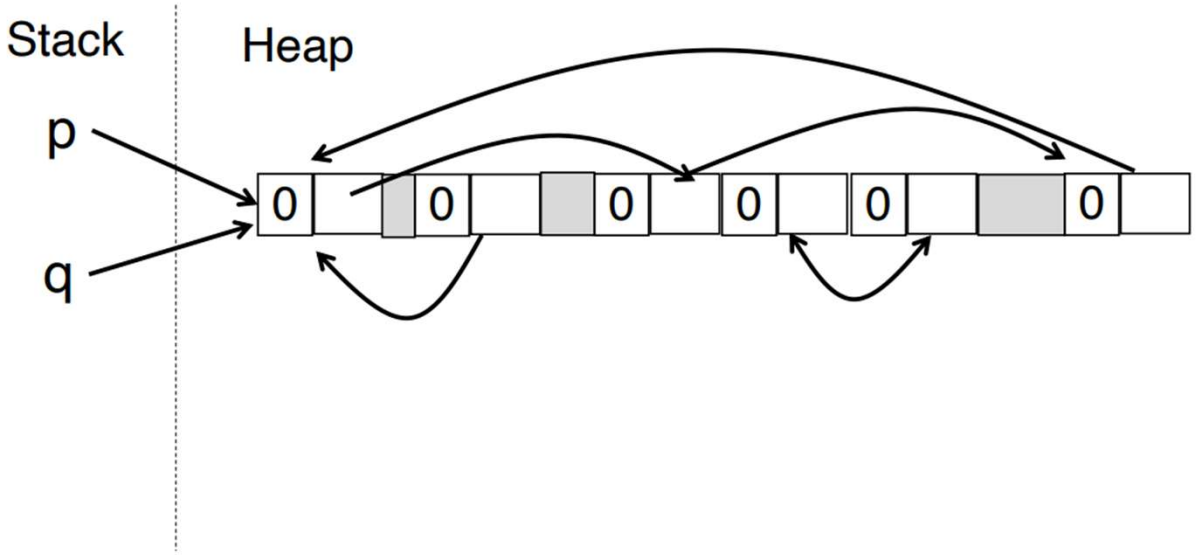
- Traverse all objects in heap, collect objects whose MB=0; otherwise, set MB to 0 (prepare for the next collection)

Mark-And-Sweep

Example

Initial state



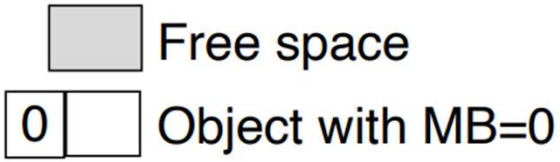


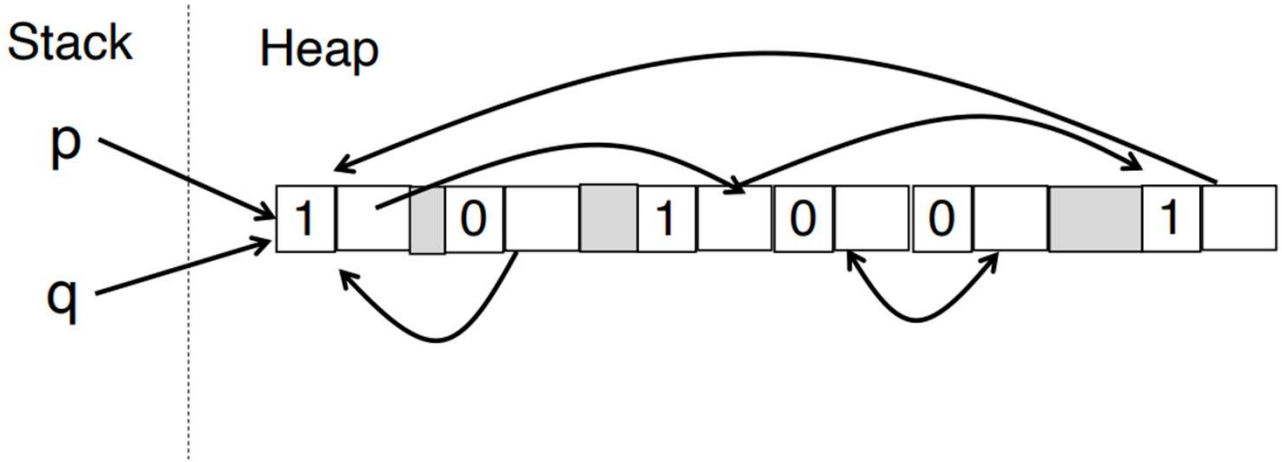
Mark-And-Sweep

PennState

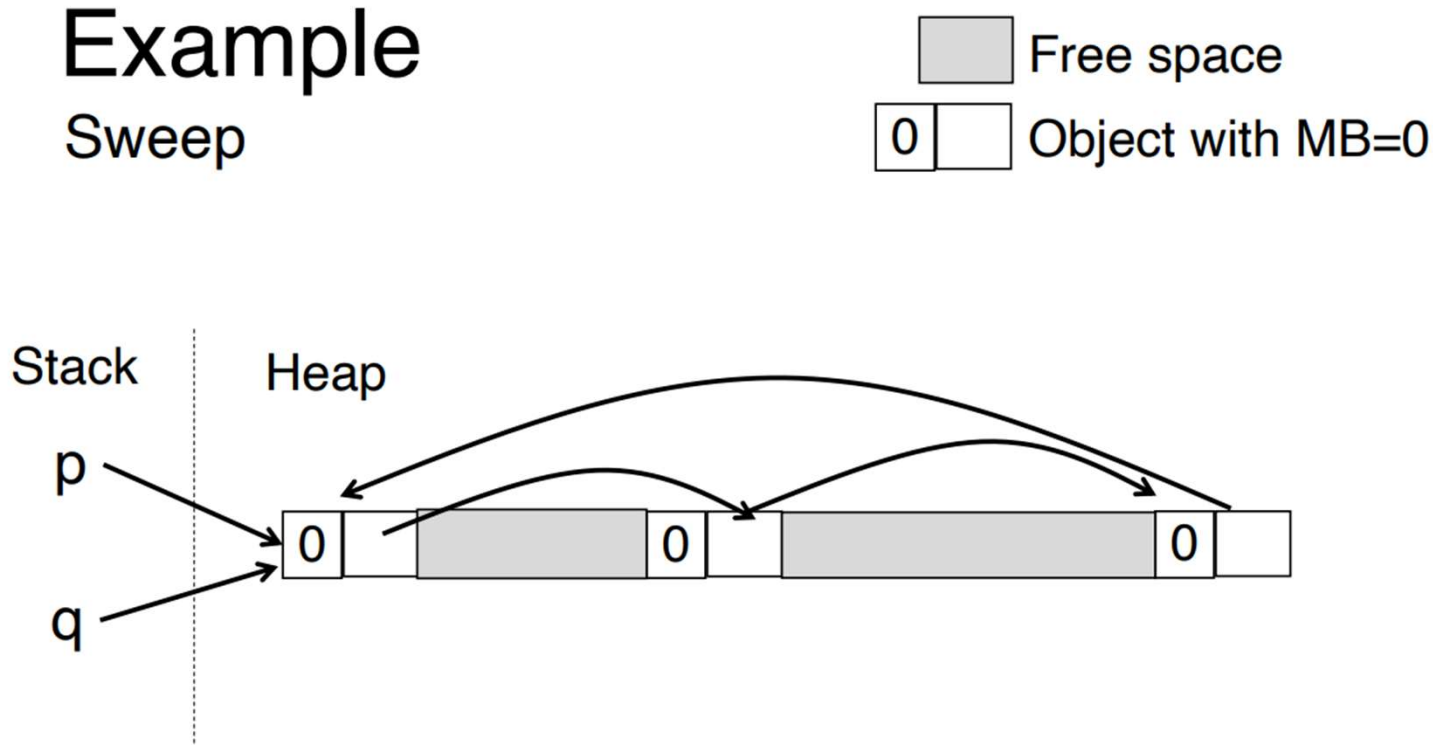
Example

Mark





Mark-And-Sweep



Mark-And-Sweep

PennState

1 Mark Bit (MB) for each object, initially 0

A: number of alive objects; N: all objects on heap

Phase 1 (Mark) – O(A)

- Traverse graph, set MB to 1 for visited node

Phase 2 (Sweep) – O(N)

- Traverse entire heap, collect objects whose MB=0; otherwise, set MB to 0 (prepare for the next collection)

Stop-the-World Collectors

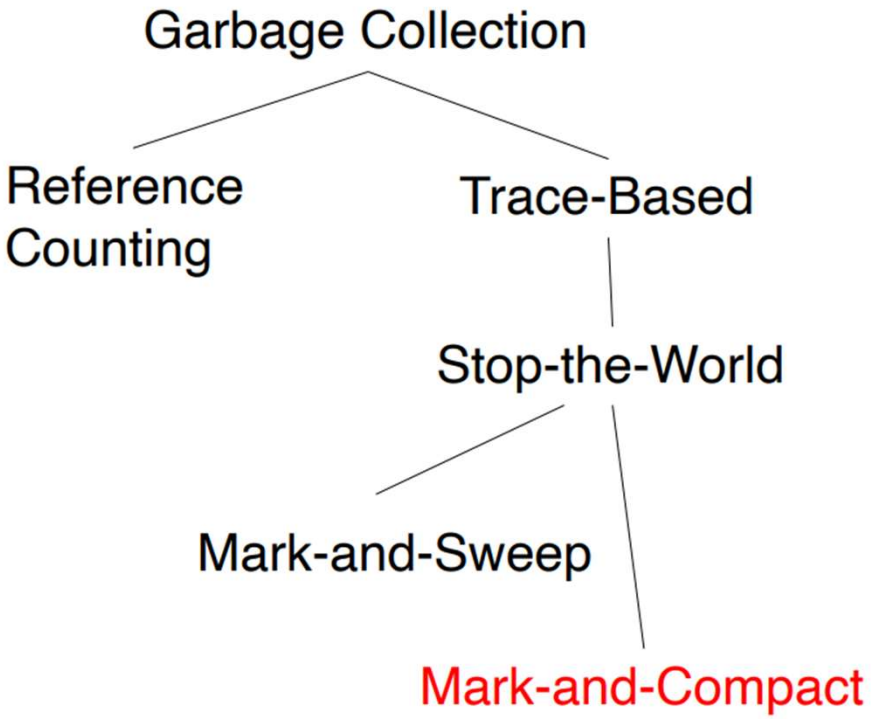
PennState

A: number of alive objects; N: all objects on heap

H: total heap size

	Mark-And-Sweep
Complexity	O(N)
Fragmentation?	Yes
Memory move	No
Effective heap size	H

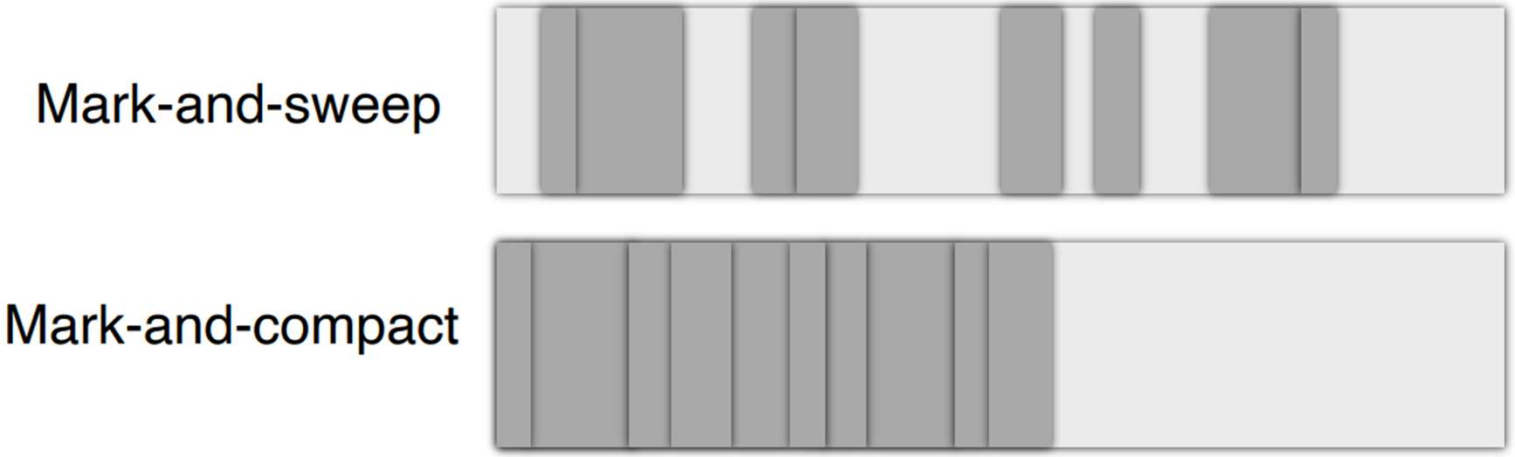
Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Mark-and-Sweep
Mark-and-Compact

Mark-and-Compact

Similar to mark-and-sweep, except

Compact: move reachable object to one end



Why Compact?

PennState

Locality: fewer pages or cache-lines needed to hold the active data.

Reduce fragmentation: available space merged to store large objects

Mark-and-Compact?

PennState

Triggered when the heap is full, interrupt program

Phase 1 (Mark): same as mark-and-sweep

Phase 2 (Compact)

a) Maintain M, a map from object to new location

I, a pointer initialized to the start of heap

For each heap object o from low address to high address

if MB=1, M(o)=I, I = I + size(o)

Mark-and-Compact?

PennState

Triggered when the heap is full, interrupt program

Phase 1 (Mark): same as mark-and-sweep

Phase 2 (Compact)

b) For each heap object o from low address to high address

if MB=1, move o to location M(o), update pointers in o (use the map M), set MB to 0

if MB=0, collect o

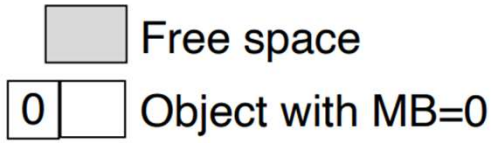
c) Retarget root references (use M)

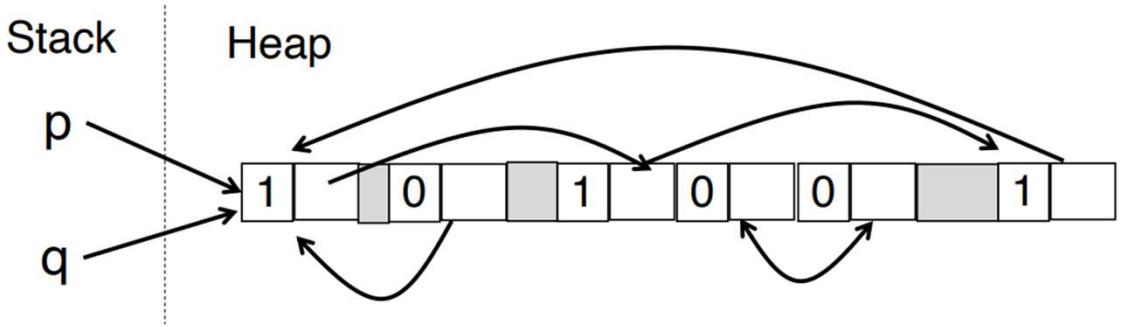
Mark-and-Compact?

PennState

Example

Mark



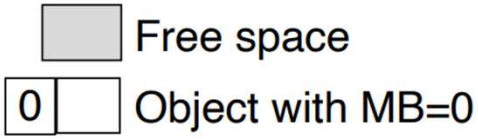


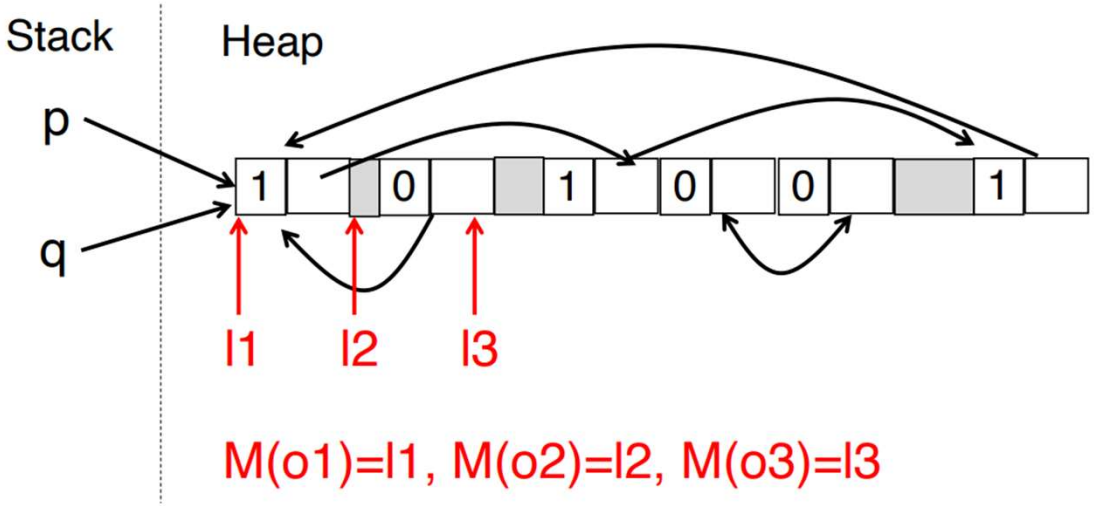
Mark-and-Compact?

PennState

Example

Compact a



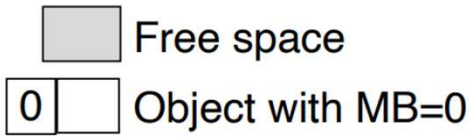


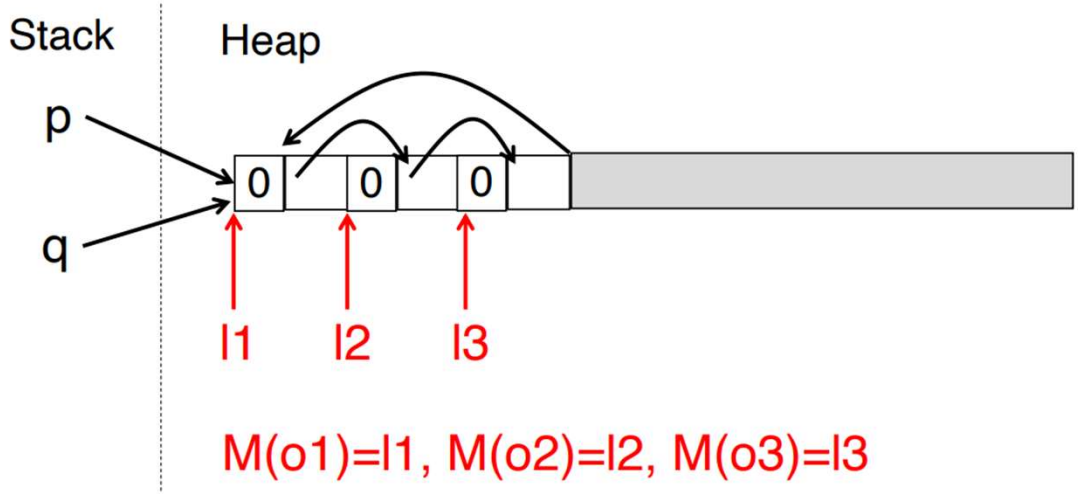
Mark-and-Compact?

PennState

Example

Compact b&c





Mark-and-Compact?

PennState

Pros:

Eliminates fragmentation

Great with long-living objects

Cons:

Bad with short-lived data

Collection time is proportional to the heap size

Mark-and-Compact?

PennState

A: number of alive objects; N: all objects on heap

Phase 1 (Mark) – O(A)

- same as mark-and-sweep

Phase 2 (Compact) – O(N)

- Compute new locations for objects
- Move new objects to new location

Stop-the-World Collectors

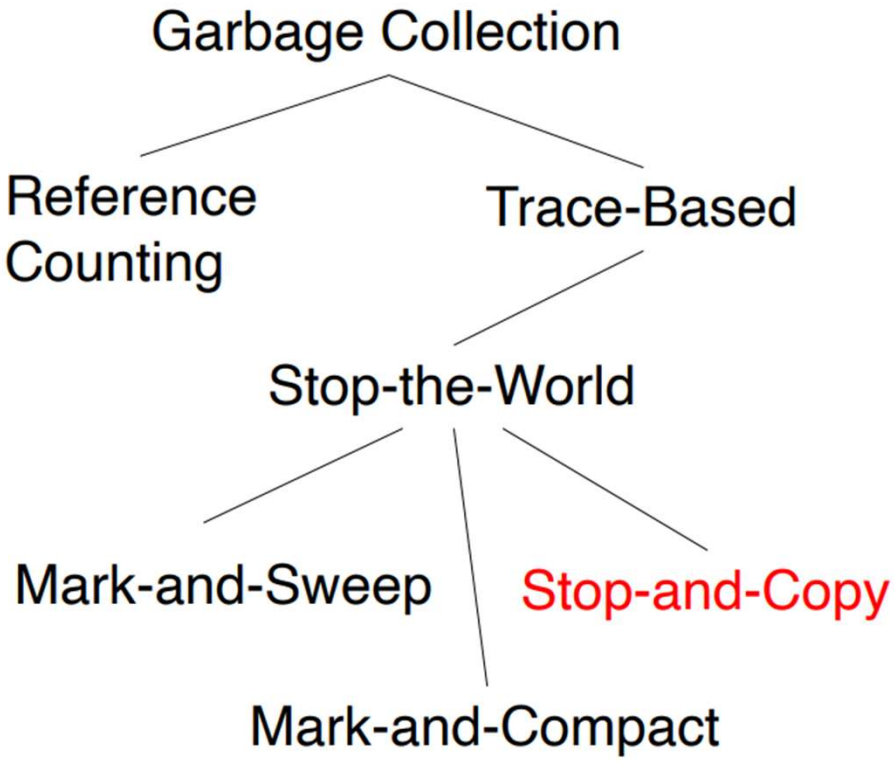
PennState

A: number of alive objects; N: all objects on heap

H: total heap size

	Mark-And-Sweep	Mark-And-Compact
Complexity	O(N)	O(N)
Fragmentation?	Yes	No
Memory move	No	Yes
Effective heap size	H	H

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Mark-and-Sweep
Stop-and-Copy
Mark-and-Compact

Stop-and-Copy (Copying Collector)

PennState

Time-Space Tradeoff:

- 2 Heaps: allocate space in one, copy to second when the other is full (half of the heap is unused)
- Only one pass over live objects is needed (much faster than previous algorithms)

MB bit is not needed

Stop-and-Copy

PennState

Triggered when heap in use is full, interrupt program

For each visited object

1. Copy it to the other heap (no fragmentation)
2. For the object in the old heap, keep a *forwarding pointer* to the new address
3. For each object it points to:

   a) if visited: update links (follow forwarding pointer)

   b) otherwise, (recursively) visit object and update link

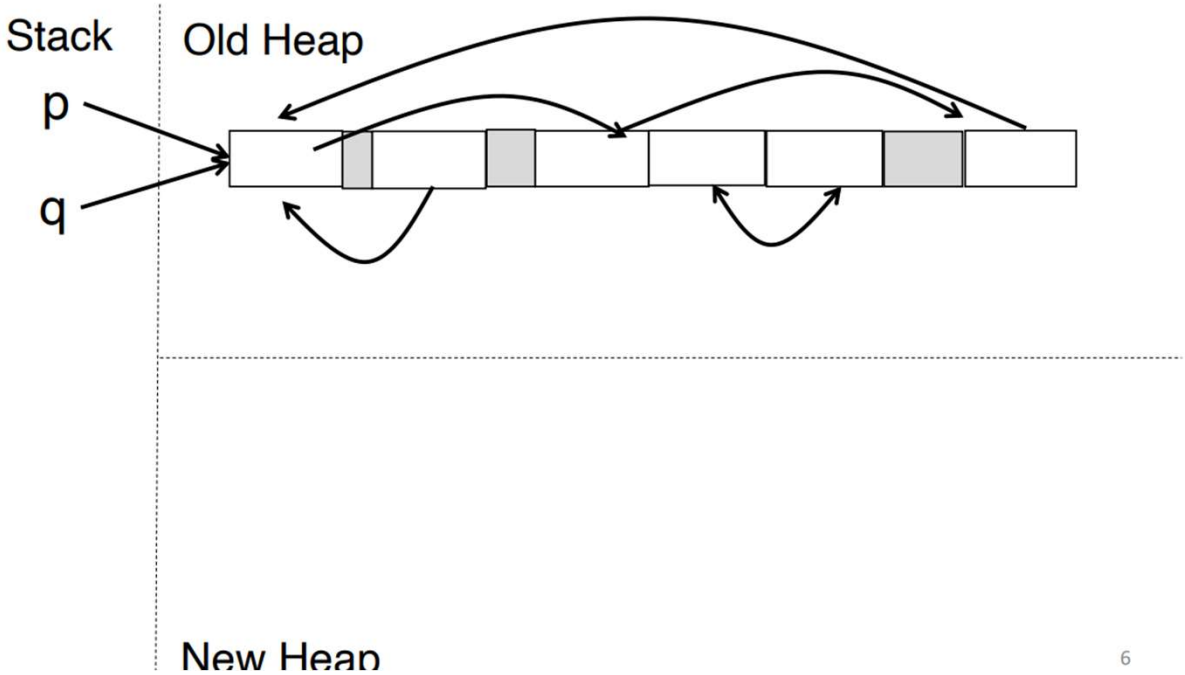
Retarget root references

Switch heaps

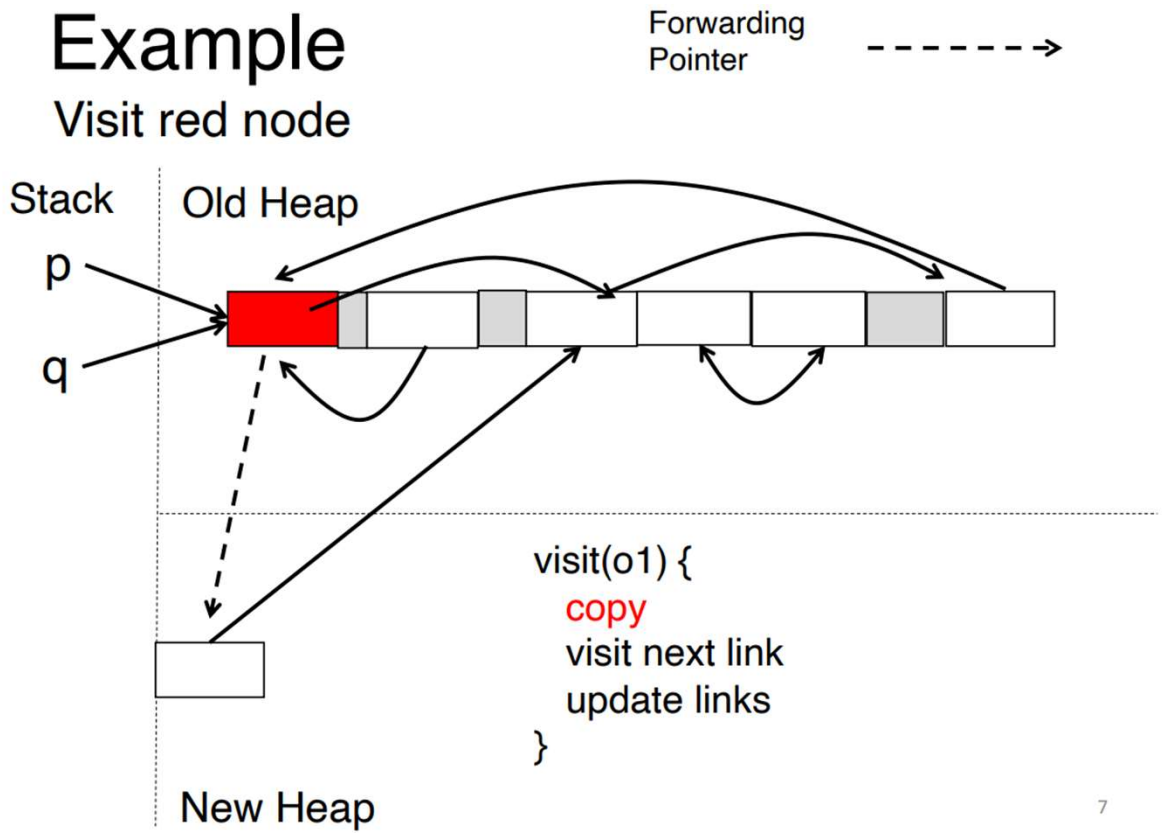
Stop-and-Copy

Example

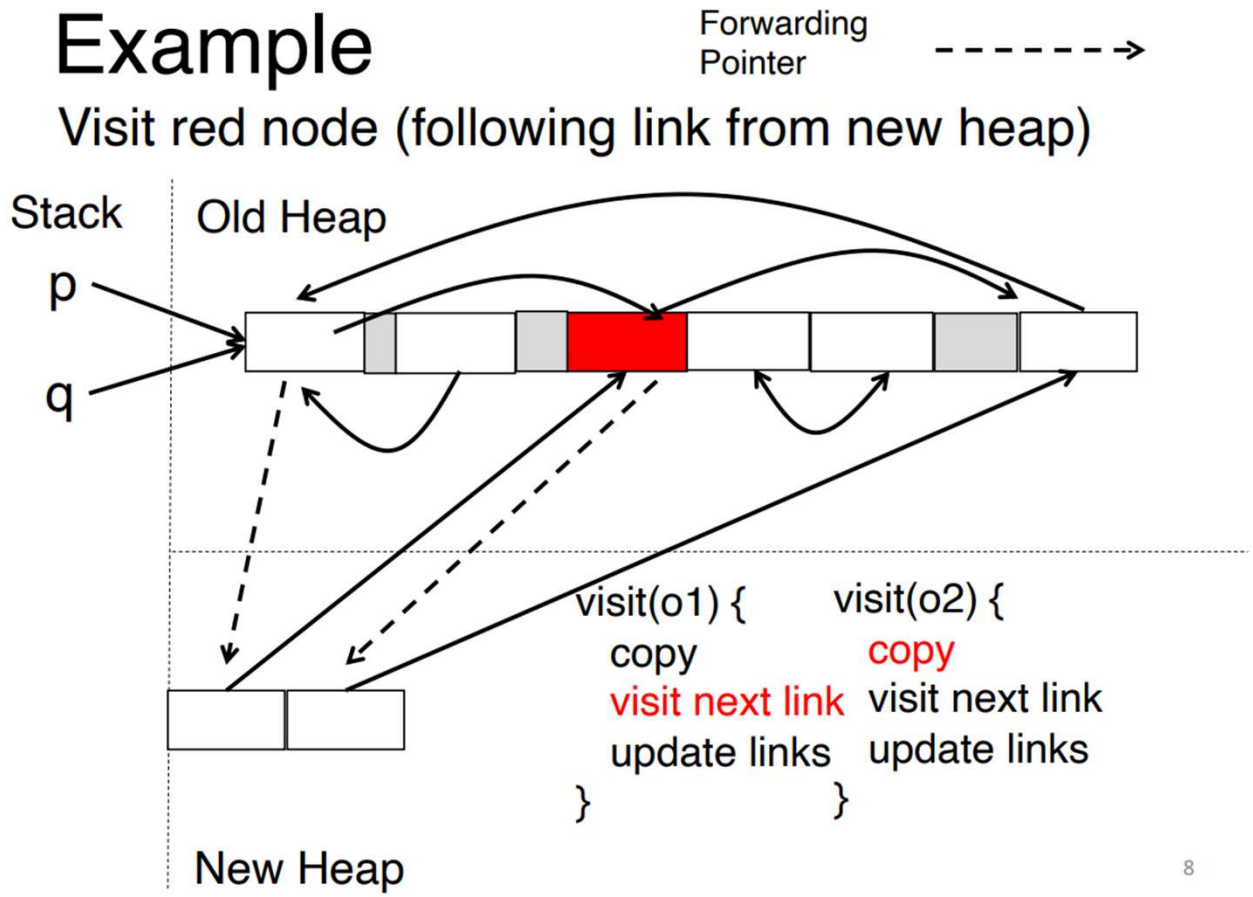
Initial state



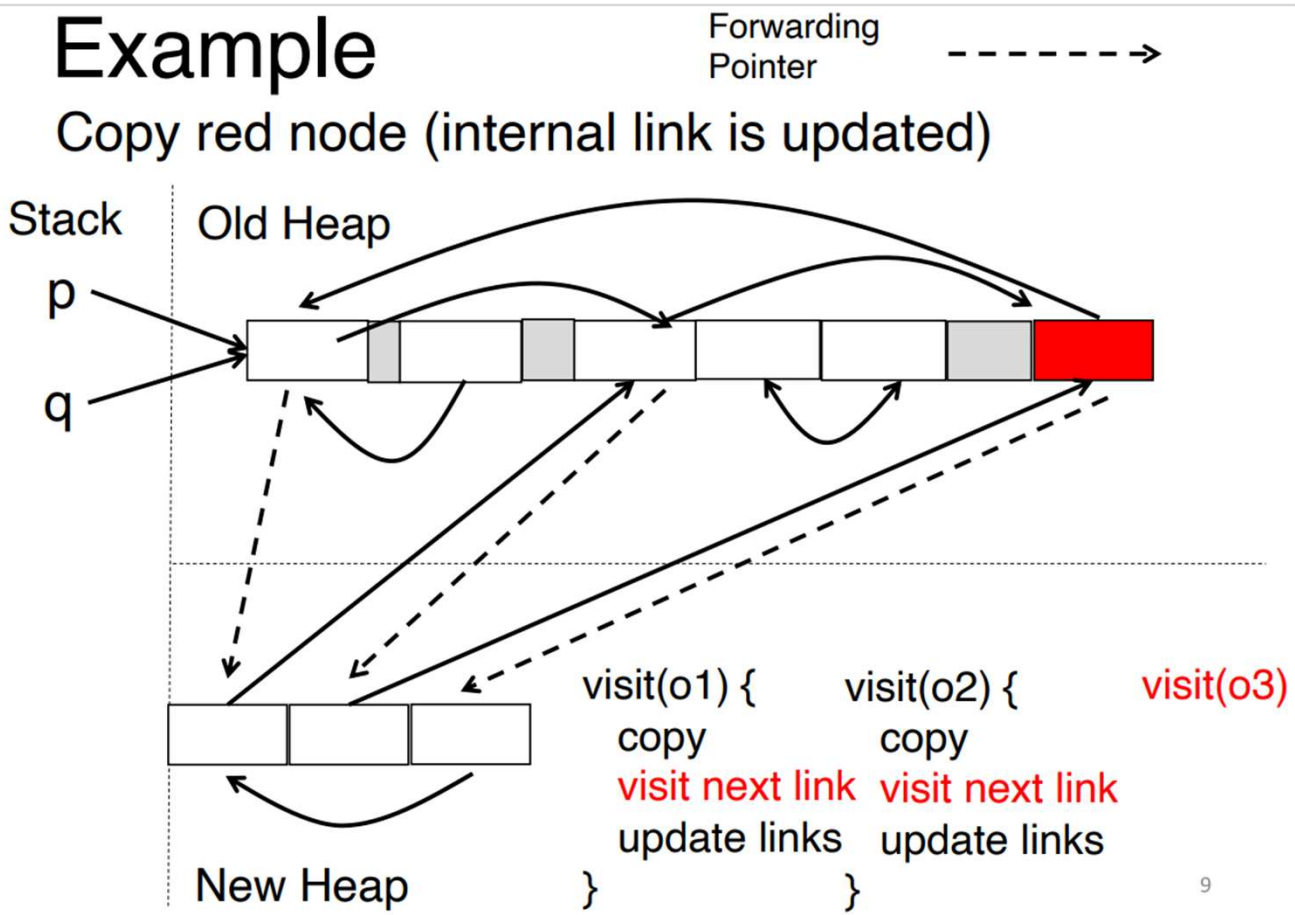
Stop-and-Copy



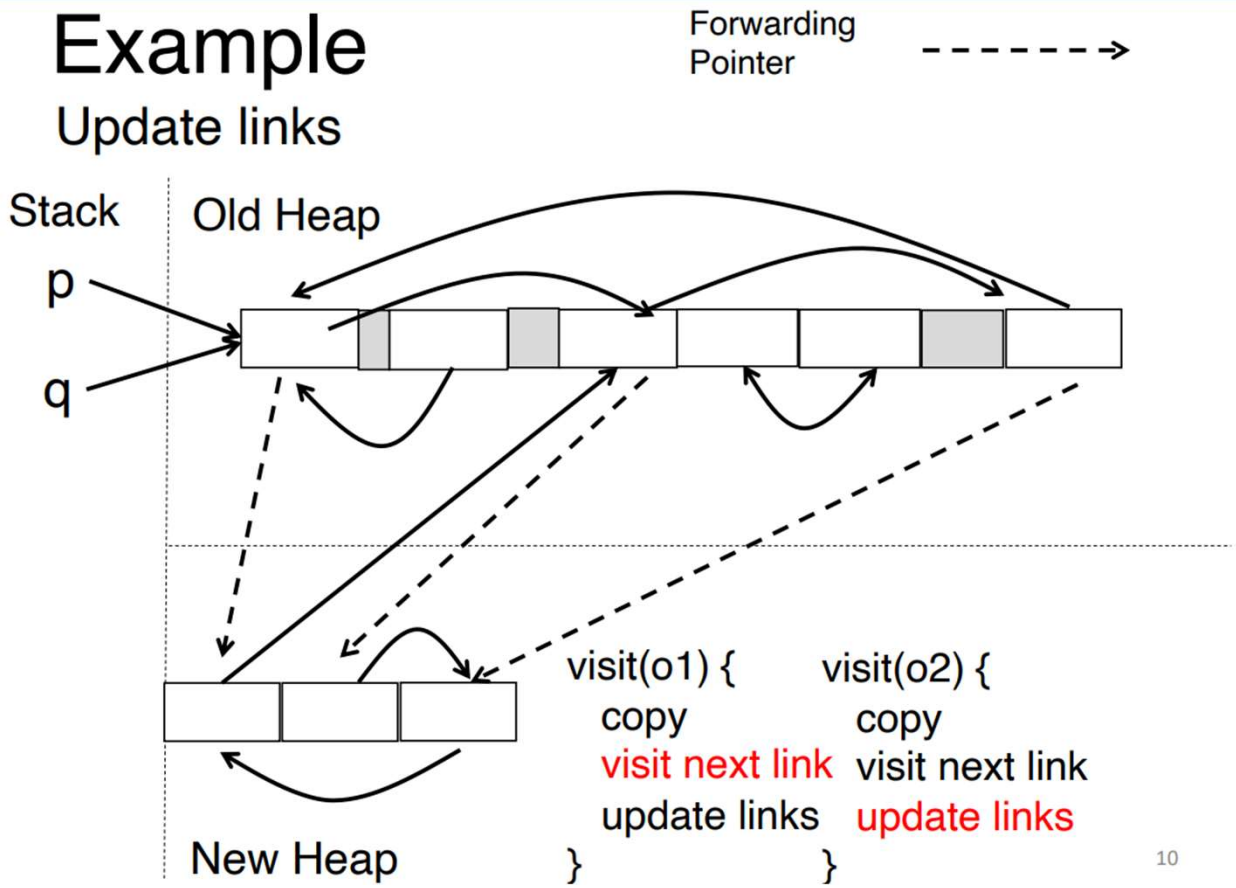
Stop-and-Copy



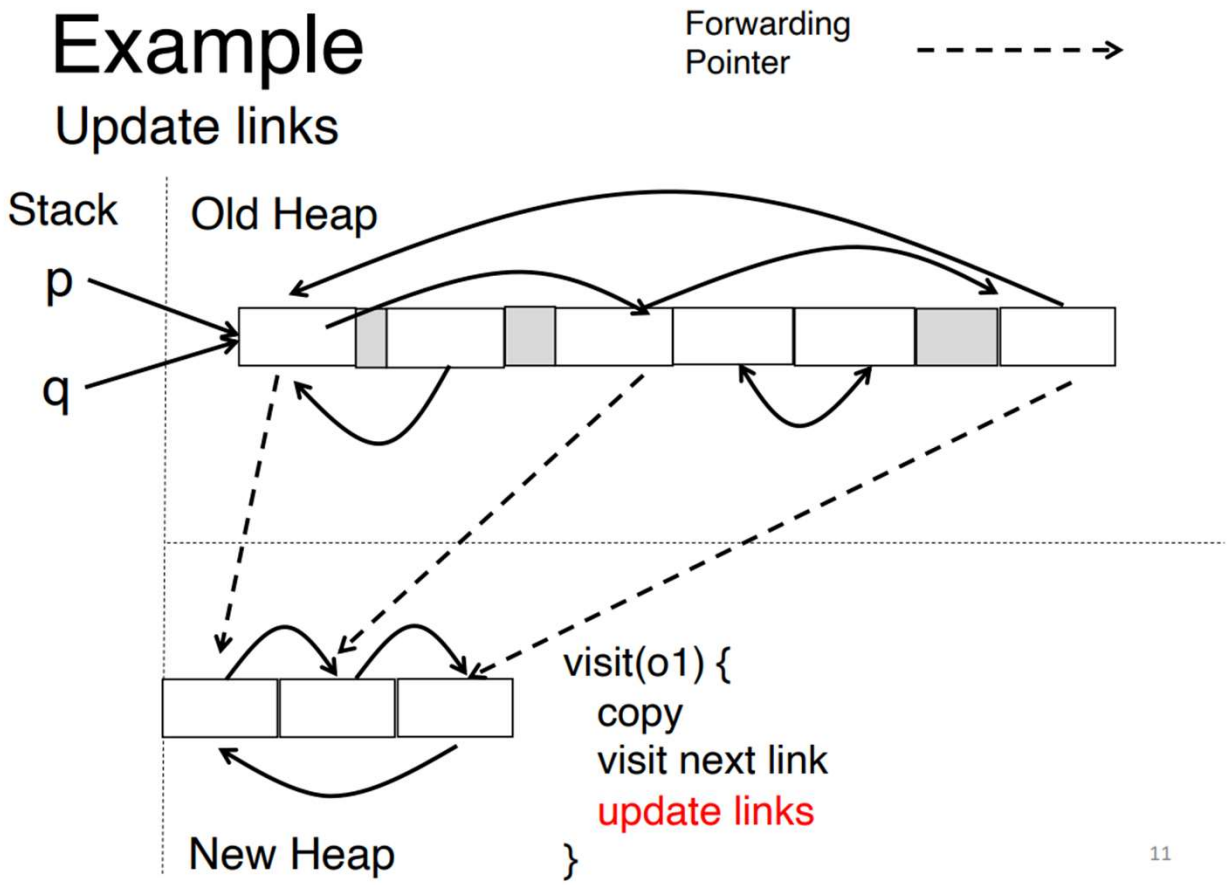
Stop-and-Copy



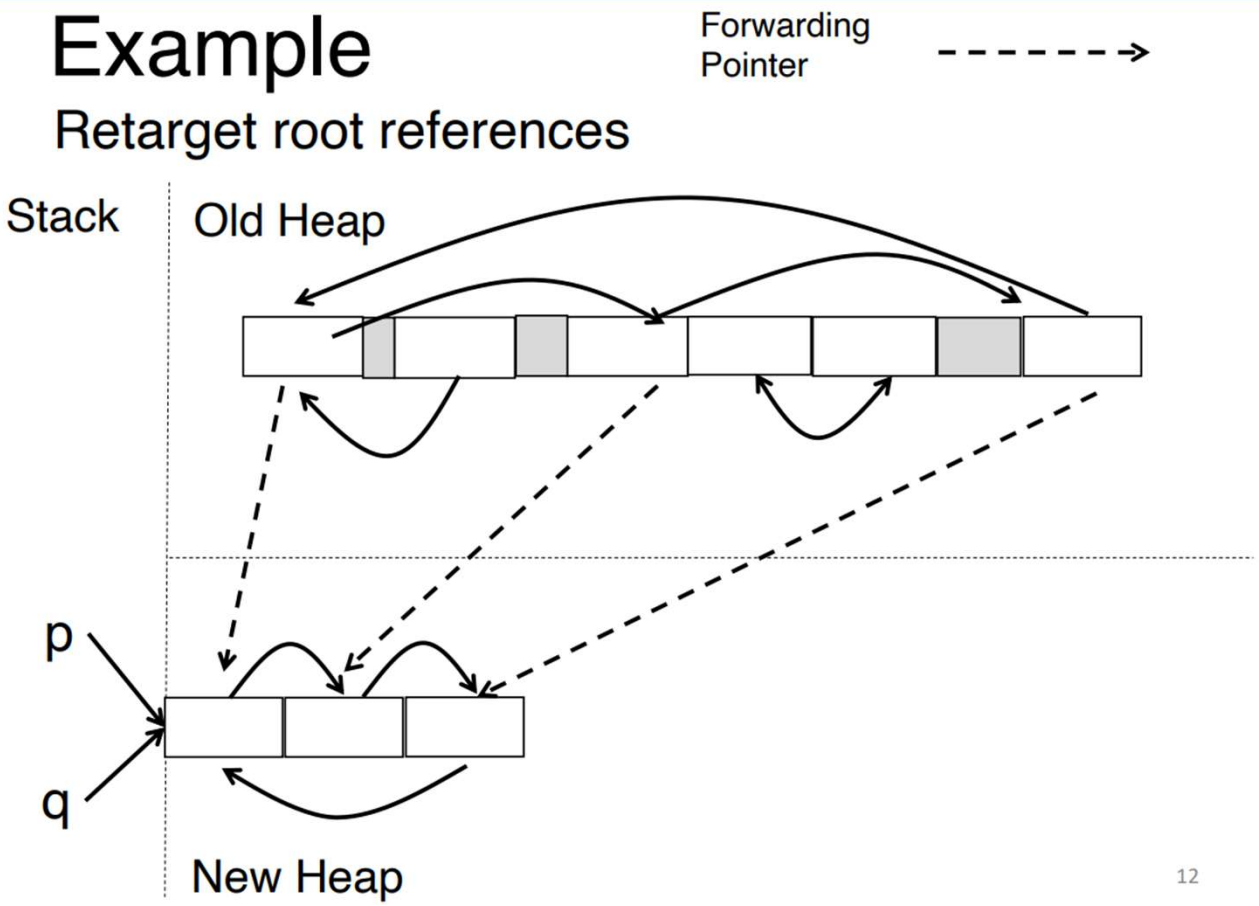
Stop-and-Copy



Stop-and-Copy



Stop-and-Copy

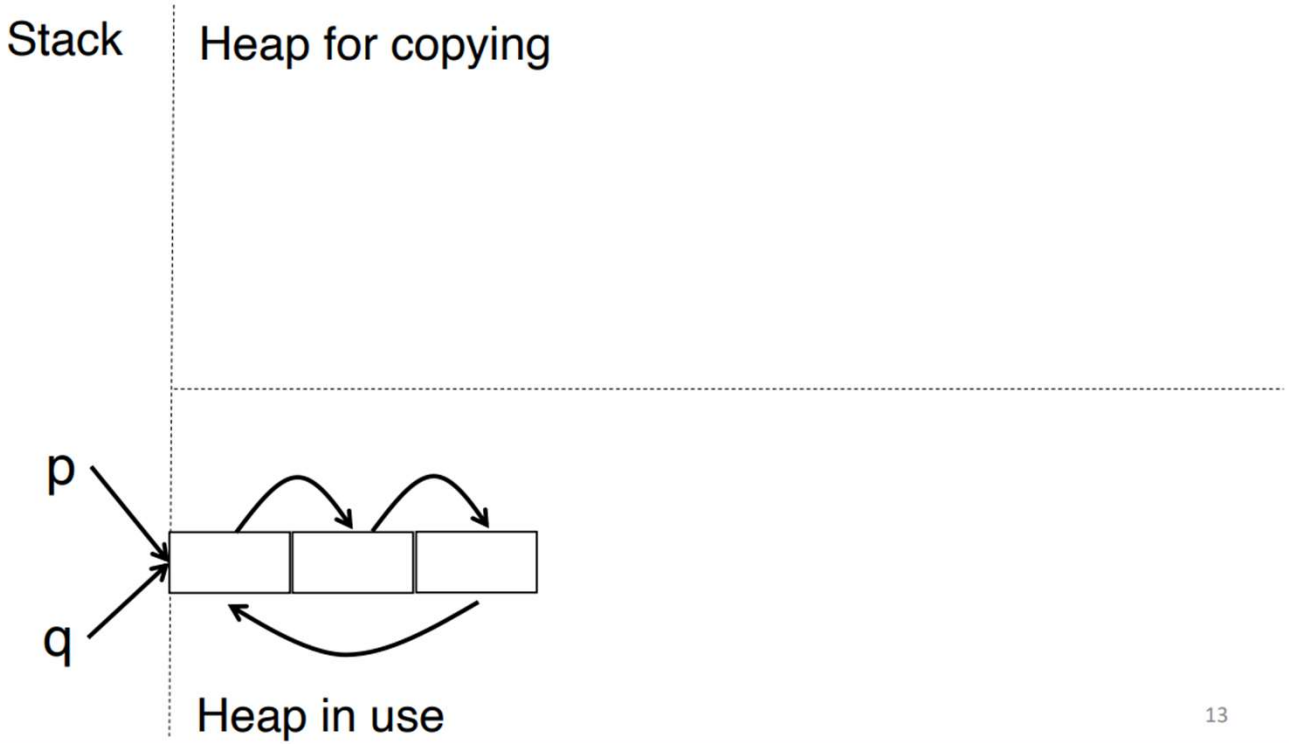


Stop-and-Copy

PennState

Example

Swap Heaps (no need to clean the old one)



Stop-the-World-Collectors

PennState

A: number of alive objects; N: all objects on heap

H: total heap size

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

Stop-and-Copy

PennState

Pros:

Eliminate fragmentation

Collection time proportional to ***live*** objects

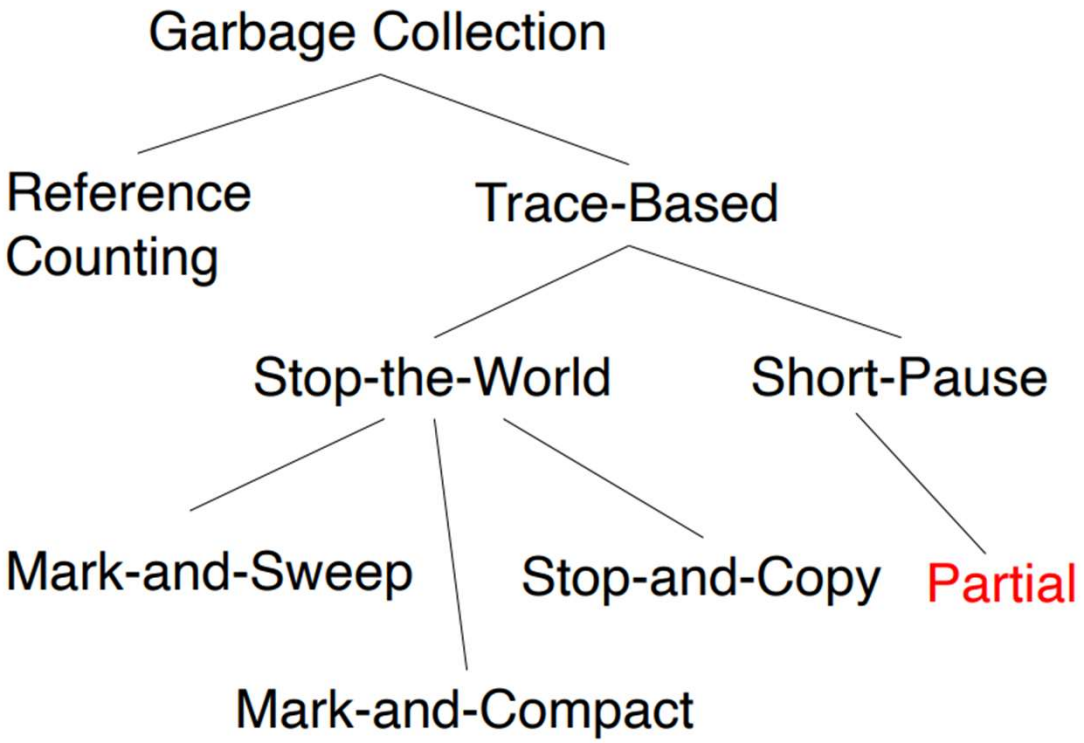
Great for short-lived data

Cons:

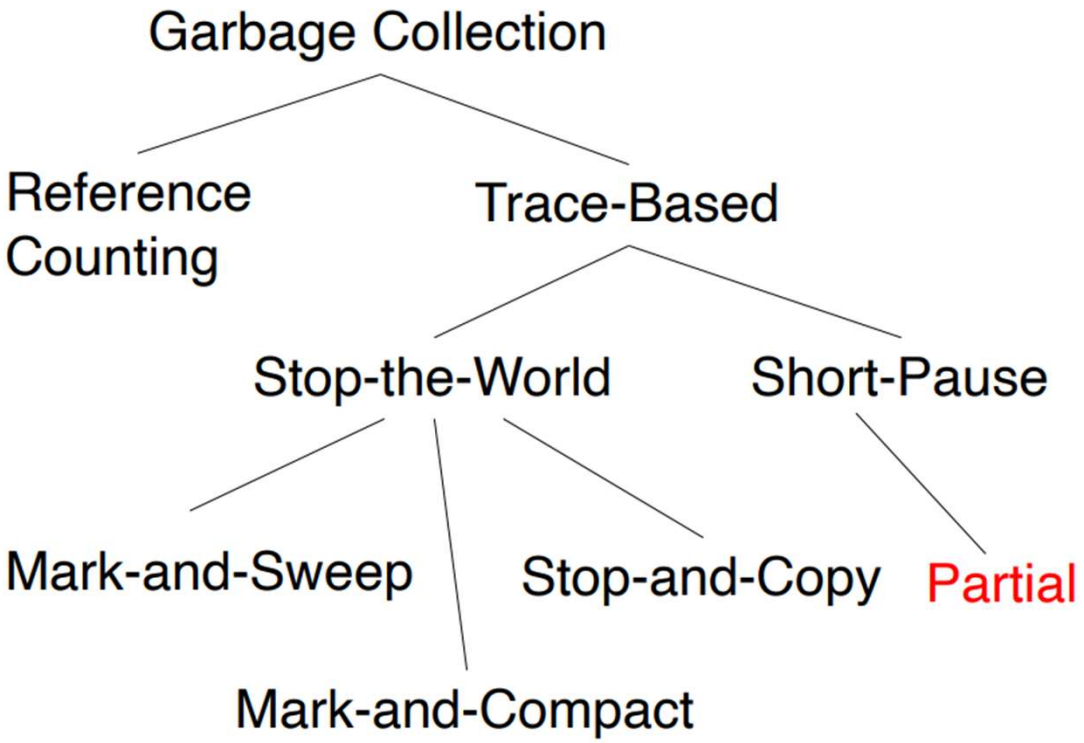
Only half of heap is in use, requires more collecting
(virtual memory alleviates this limitation)

Bad for long living data (copying data back and forth)

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Stop-and-Copy
Partial

Short-Pause Collection



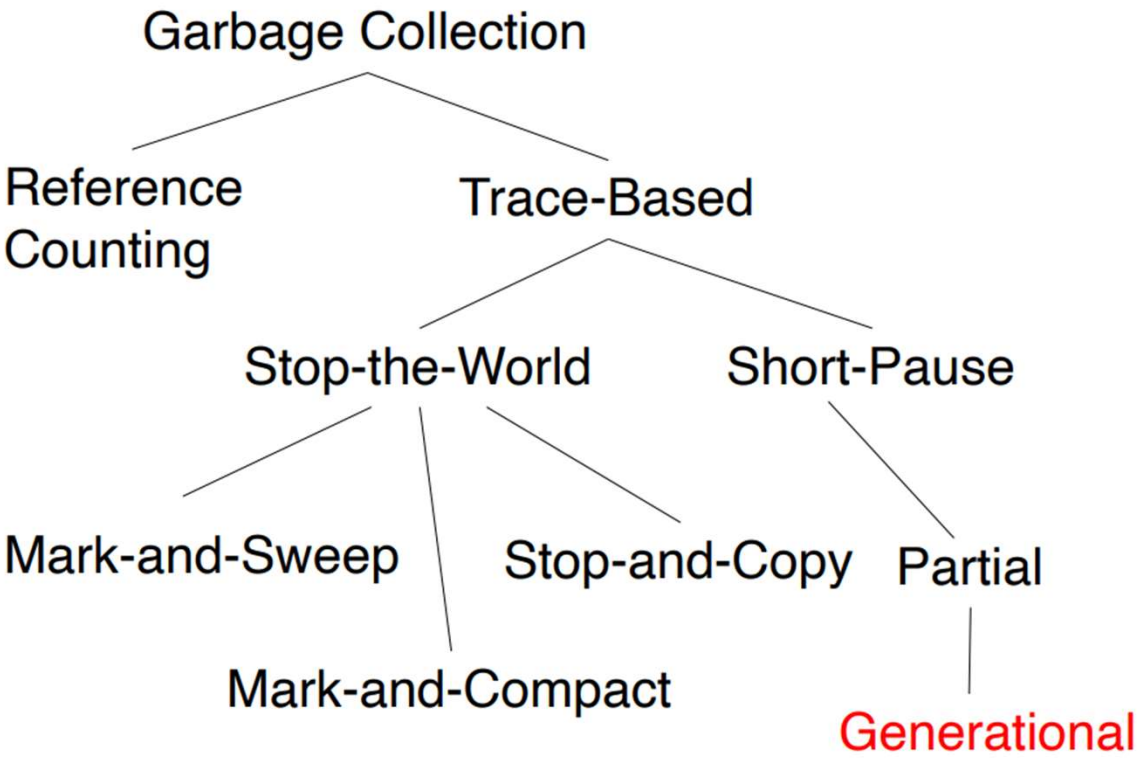
Short-Pause Collection

PennState

Partial

- stop the program, but only briefly, to garbage collect **a part of** the heap

Taxonomy


Garbage Collection
Reference Counting
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Stop-and-Copy
Partial
Mark-and-Compact
Generational

The Object Life-Cycle

PennState

"Most objects die young."

- But those that survive one GC are likely to survive many

Idea: tailor GC to spend more time on regions of the heap where objects have just been created

A better ratio of reclaimed space per unit time

Generational

PennState

Divide heap into generations g_1, g_2 …

- g_i holds older objects than g_{i-1}

Create new objects in g_1, until it fills up

GC g_1 only; move reachable objects to g_2 after several collections (typically one collection)

Generational

PennState

When g_2 fills, garbage collect g_1 and g_2, and put the reachable objects in g_3

In general: When g_i fills, collect g_1, g_2, …, g_i, and put the reachable objects in g_{i+1}

What GC algorithm is better for young generation?
How about old generation?

Algorithm for Young Generation

PennState

A: number of alive objects; N: all objects on heap

$A \ll N$

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

Stop-and-copy is the most efficient

Algorithm for Old Generation

PennState

A: number of alive objects; N: all objects on heap

A is close to N

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

Mark-And-Compact removes fragmentation

Algorithm for Old Generation

PennState

A: number of alive objects; N: all objects on heap

What if most objects have the same size?

	Mark-And-Sweep	Mark-And-Compact	Stop-And-Copy
Complexity	O(N)	O(N)	O(A)
Fragmentation?	Yes	No	No
Memory move	No	Yes	Yes
Effective heap size	H	H	H/2

Mark-And-Sweep saves the cost of moving objects

Generational

PennState

Pros:

Divide heap according to lifetimes of data

Great for data with mixed lifetimes

Cons:

Garbage might survive a collection

Small heap size for each generation

More frequent collection