Garbage Collection

Automatic Heap Memory Management

Diogenes Nunez
April 3, 2019

Garbage collection is in most languages

- Java (desktop and Android)
- Javascript
- Swift (or Objective-C)
- Python
- Haskell
- Lisp/Scheme
- ML/OCaML
- PHP
- Go
- ...

Memory Allocation

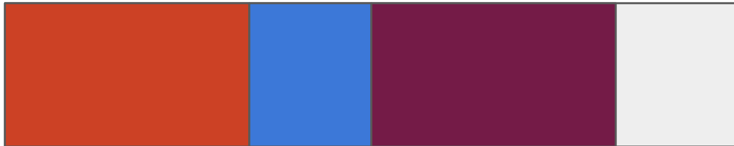
- Memory can be allocated in different locations
    - Stack
    - Heap
    - Global
    - Filesystem

Objects

- An **object** is an allocated chunk of memory for use by the application.
- Examples
  - C stack objects: Static arrays, local variables
  - C heap objects: Anything allocated with malloc
  - Java stack objects: Local variables
  - Java heap objects: Anything allocated with new, Class definitions
- A **reference** is a pointer to an object.

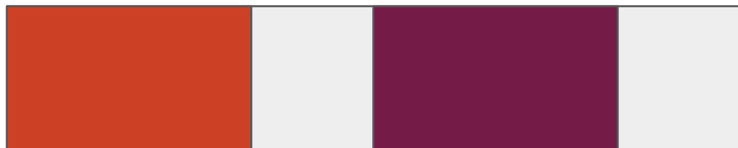
Incorrect heap memory management causes issues

- Leads to memory corruption errors when wrong
  - Use after free
  - Dangling pointer
  - Double free
- Can create space leaks by not deallocating
- Can fragment the heap



Incorrect heap memory management causes issues

- Leads to memory corruption errors when wrong
  - Use after free
  - Dangling pointer
  - Double free
- Can create space leaks by not deallocating
- Can fragment the heap



Liveness

- A heap object is **live** if the object will be accessed by the program in the future.
- A heap object is **dead** if the object will never be accessed by the program.

Garbage Collector

- **Garbage collector** (GC) is a program that reclaims dead objects automatically for an application to reuse.
- GC runs when the application fails to allocate a new heap object.

Perfect GC can’t exist.

- Detecting liveness of a heap object in an arbitrary program is undecidable.
- Therefore, GC must *estimate* liveness.
    - Must be conservative in liveness.

1. Garbage Collection (GC)
2. GC Algorithms
    a. Reference Counting
    b. Mark Sweep
    c. Copying
    d. Generational and other Modifiers
3. GC in Practice

Roots

The **roots** of a program are memory addresses directly accessible by the application without following a pointer.

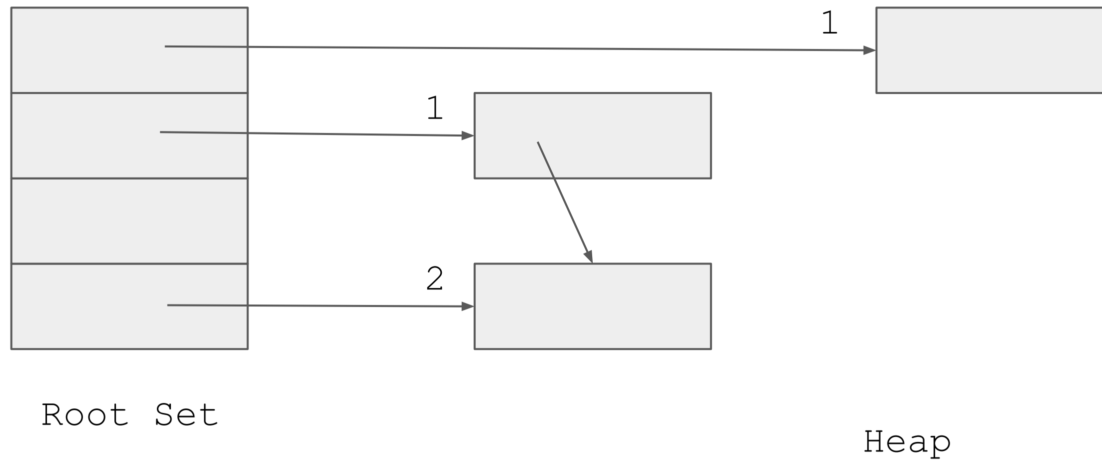
- Stack objects
- Global objects
- Register values

The collection of roots is called the **root set**.

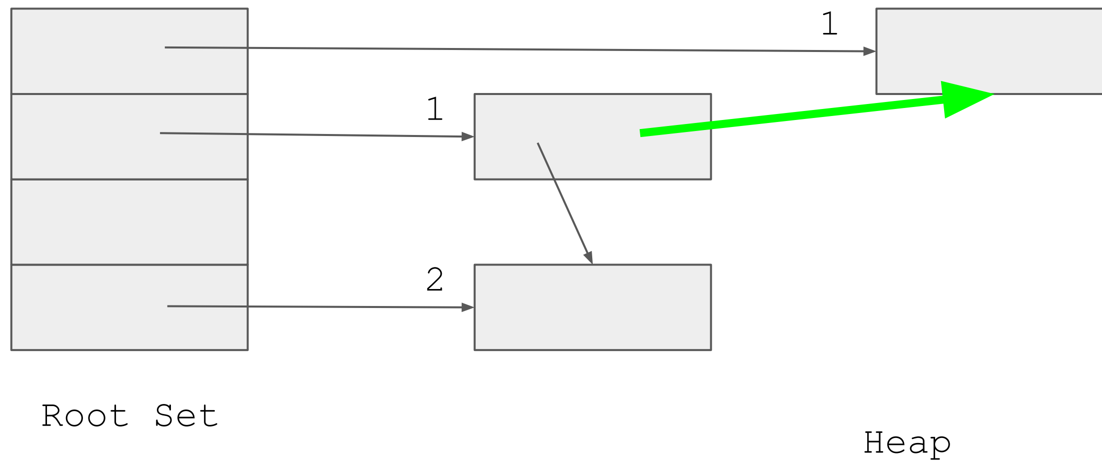
Reference Counting Algorithm: Insight

- If a heap object has no incoming pointers, that object can never be accessed.
- Count the number of incoming pointers on each heap object.
- Pointer updates change the count.
  - +1 when adding an incoming pointer
  - -1 when removing an incoming pointer
  - If 0 incoming pointers, remove object's outgoing references and then reclaim object

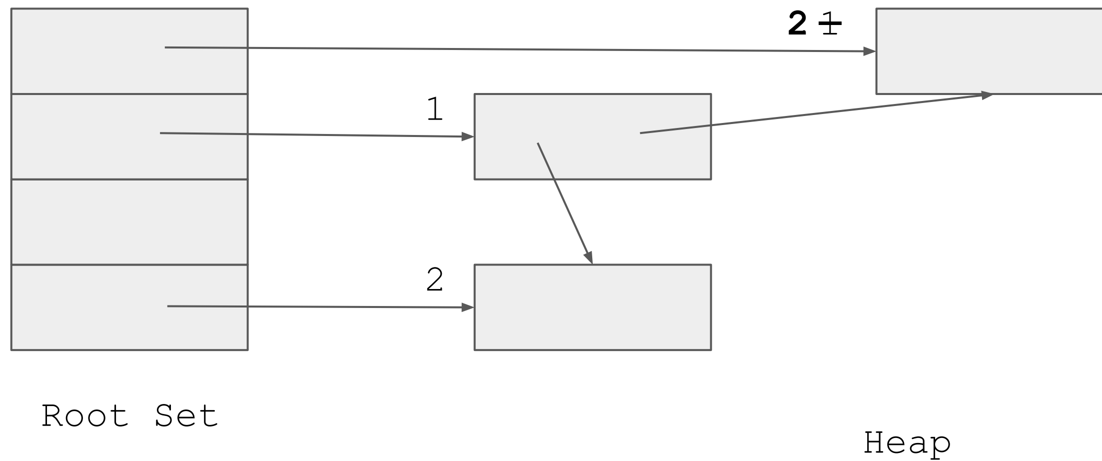
RC: Adding a new pointer



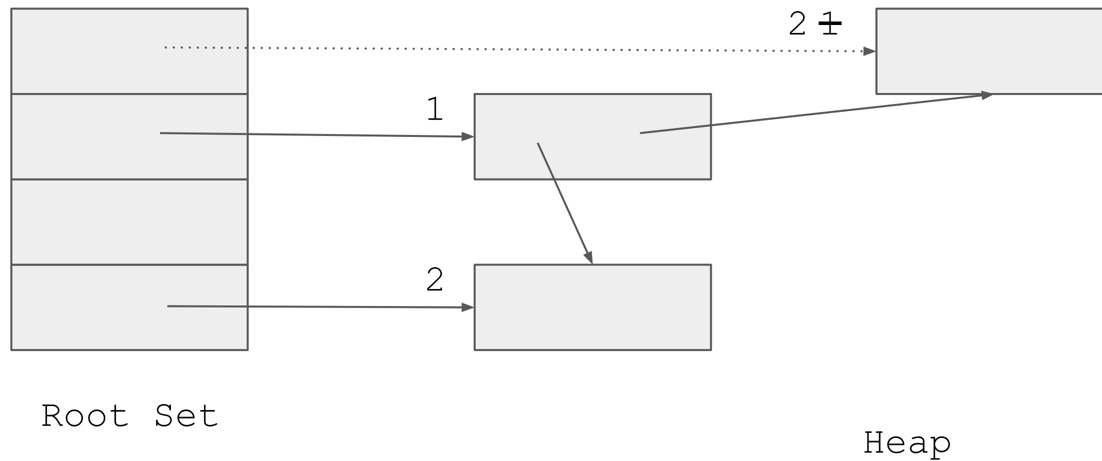
RC: Adding a new pointer



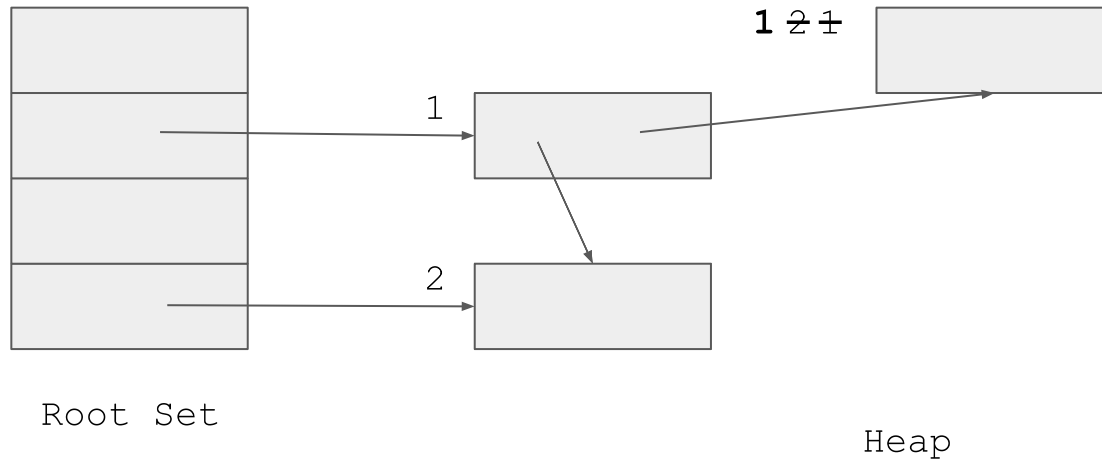
RC: Adding a new pointer



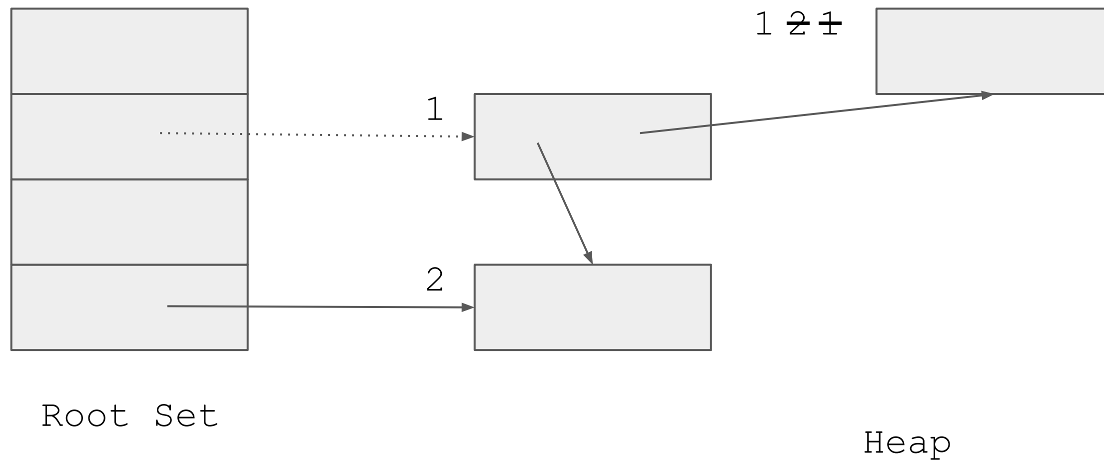
RC: Removing a pointer



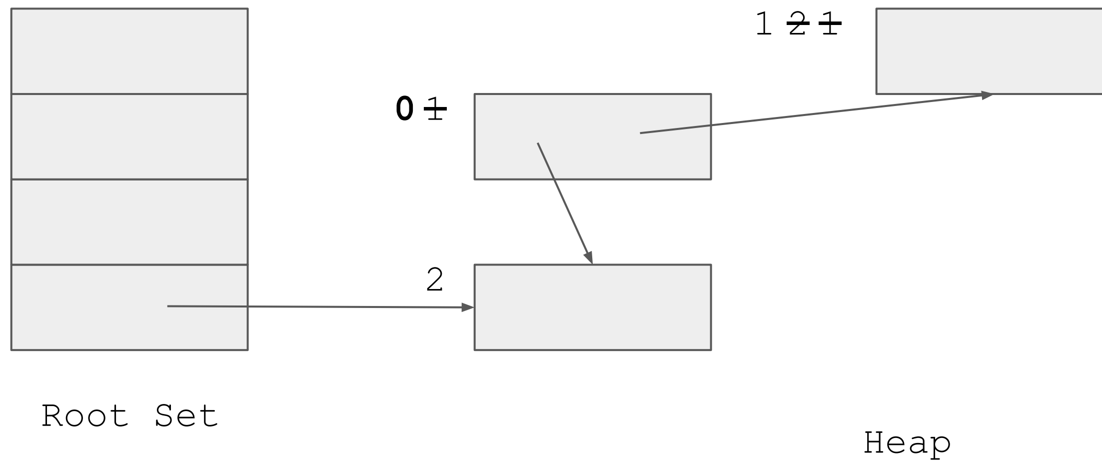
RC: Removing a pointer



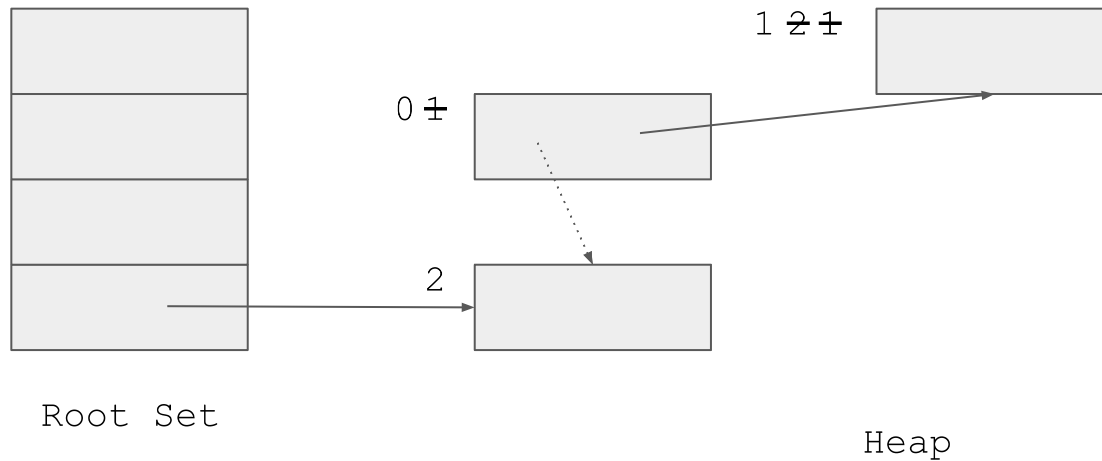
RC: Dealing with dead objects



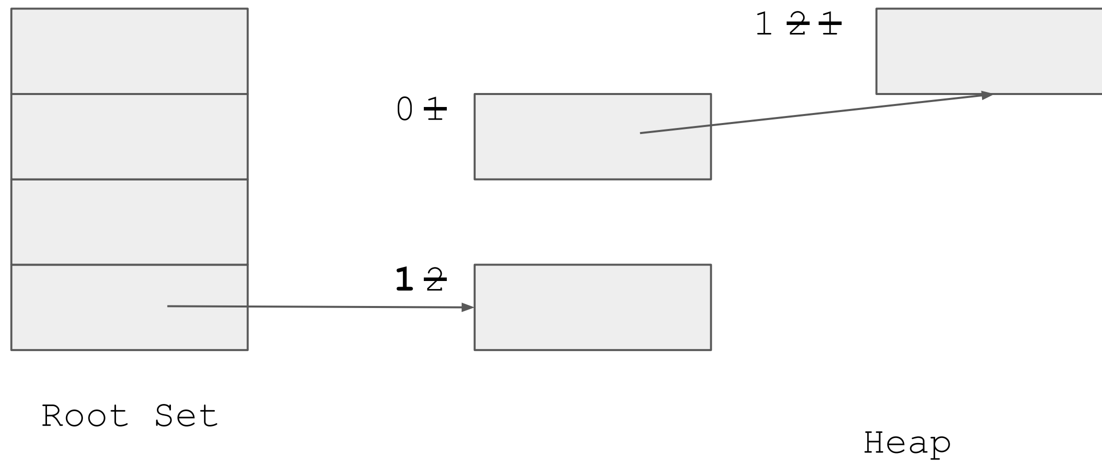
RC: Dealing with dead objects



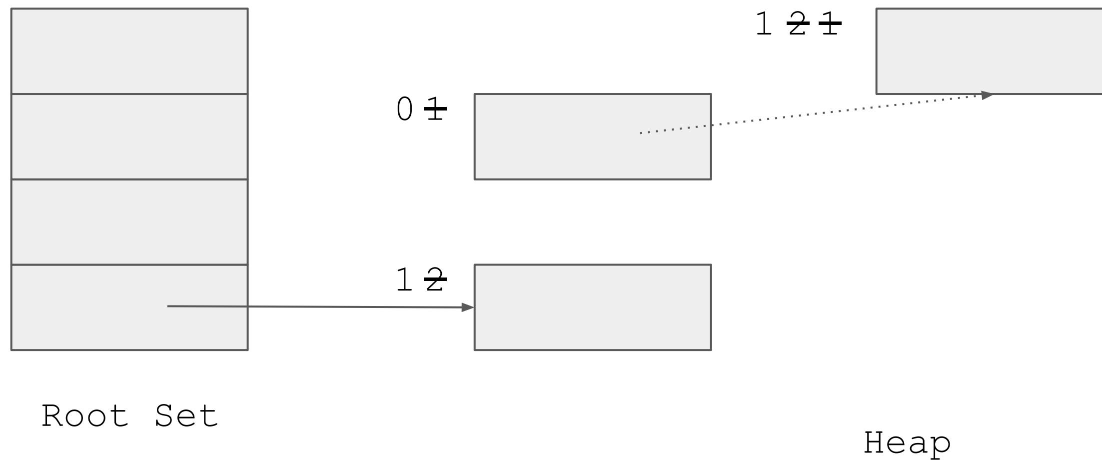
RC: Dealing with dead objects



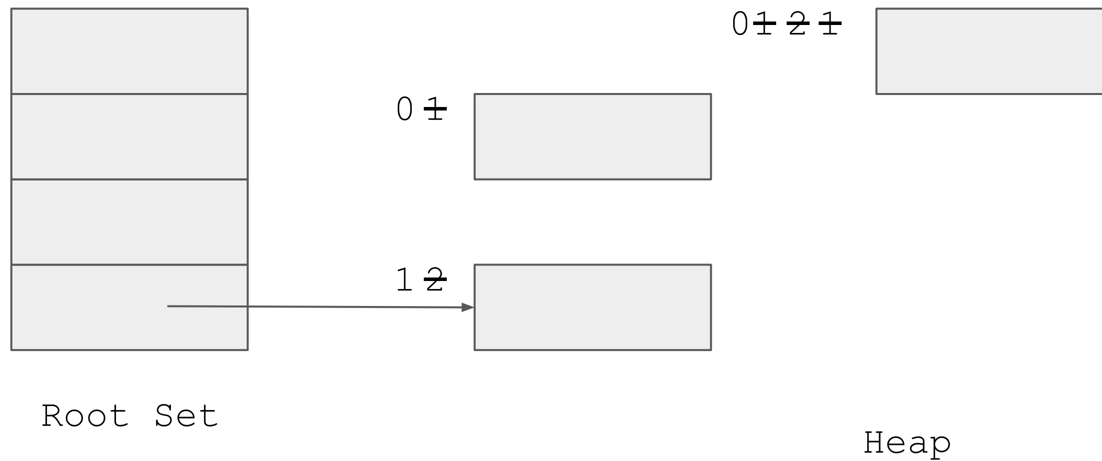
RC: Dealing with dead objects



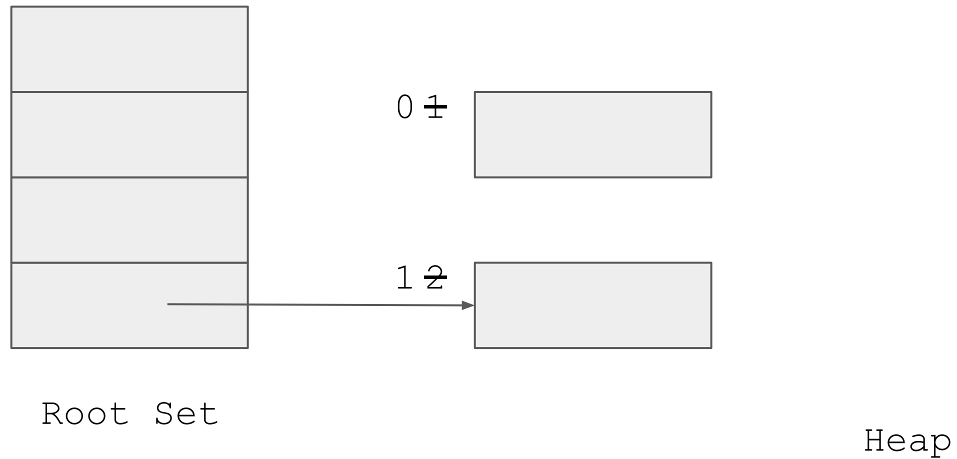
RC: Dealing with dead objects



RC: Dealing with dead objects

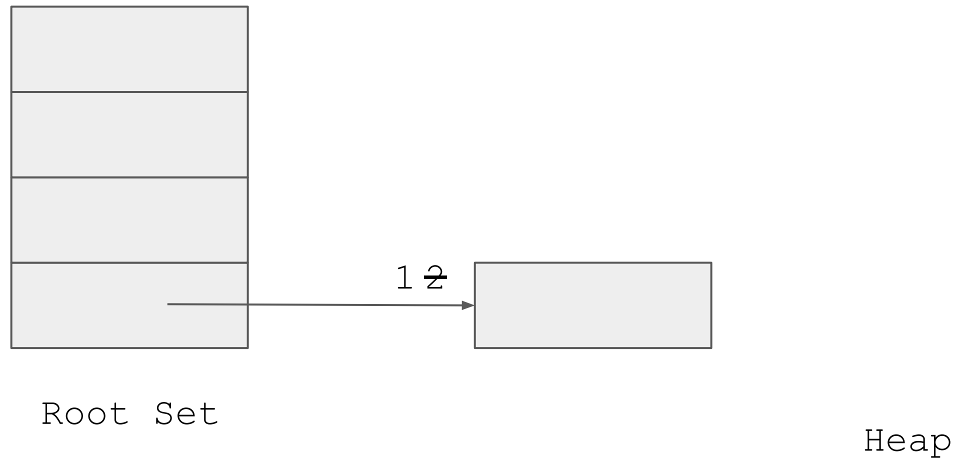


RC: Dealing with dead objects



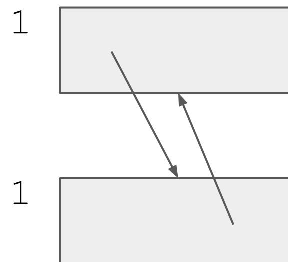
Heap

RC: Dealing with dead objects



RC: Pros and Cons

- Pros
  - Immediately reclaim objects upon death
  - Counter updates are incremental
- Cons
  - Overhead on each pointer update
  - Decrementing a count can cause a lengthy pause
  - Can cause fragmentation
  - Cannot reclaim dead object cycles

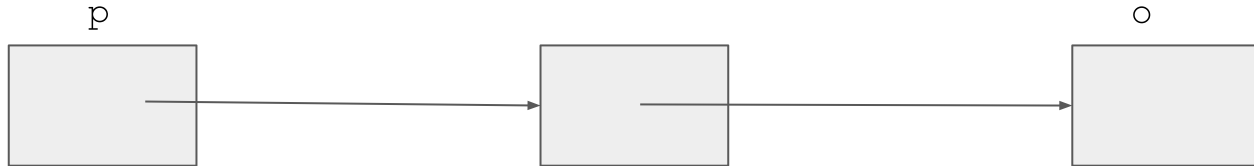


Heap

1. Garbage Collection (GC)
2. GC Algorithms
    a. Reference Counting
    b. Mark Sweep
    c. Copying
    d. Generational and other Modifiers
3. GC in Practice

Mark Sweep Algorithm: Reachability

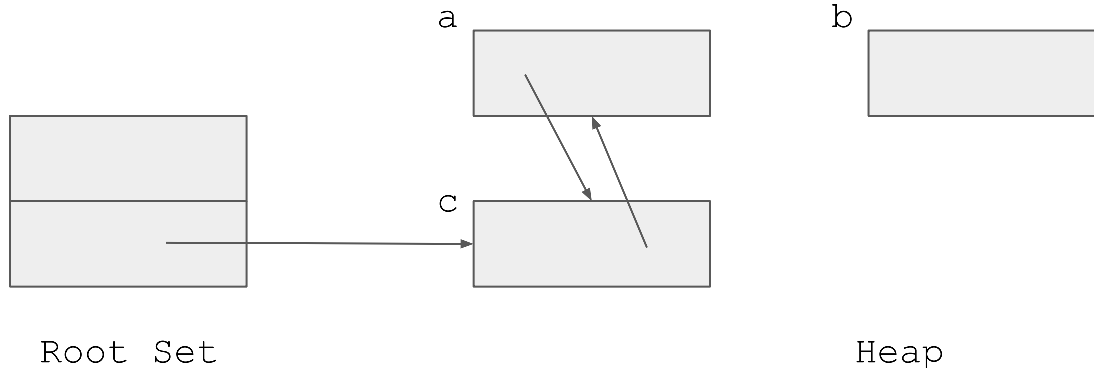
- Object o is **reachable** from object p if there exists a path of pointers from p to o.



Mark Sweep estimates liveness with reachability.

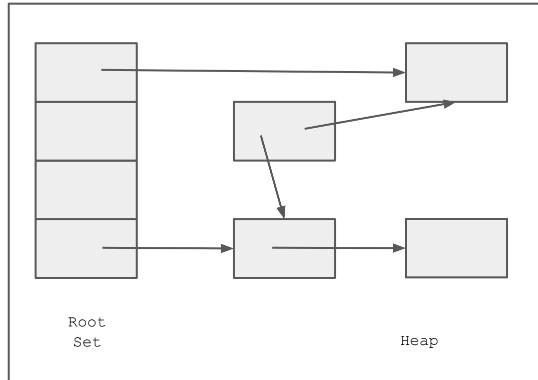
A heap object is live iff it is reachable from any root.

- By transitivity, if an object is live, so are any objects it points to.

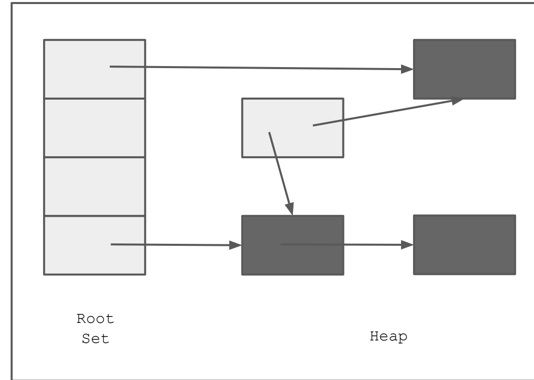


Mark Sweep Algorithm: Two Phases

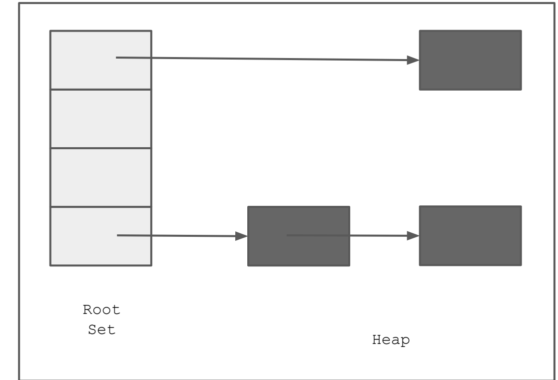
- Mark phase: Use BFS or DFS to mark object reachable from roots as live
- Sweep phase: Traverse heap and reclaim all unmarked objects



Initial Heap

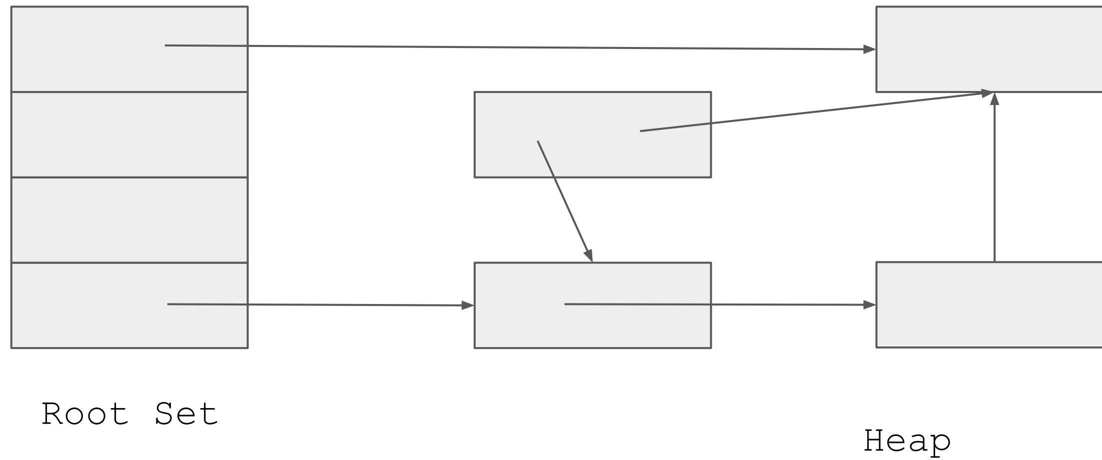


After Mark Phase

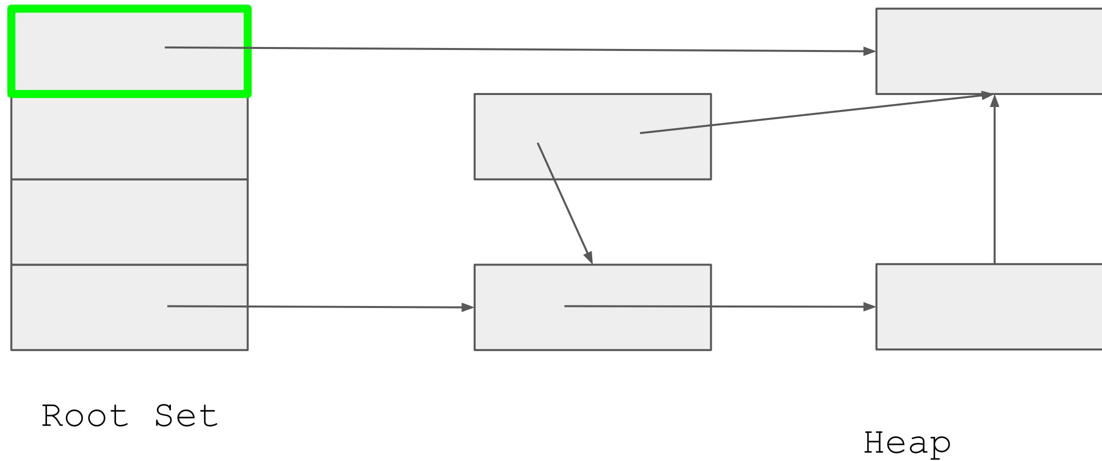


After Sweep Phase

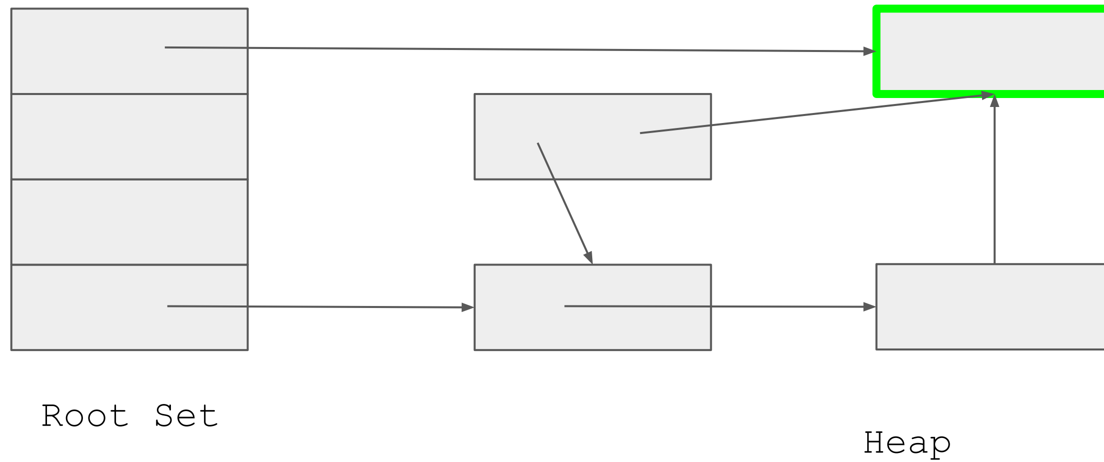
MS: Mark Phase



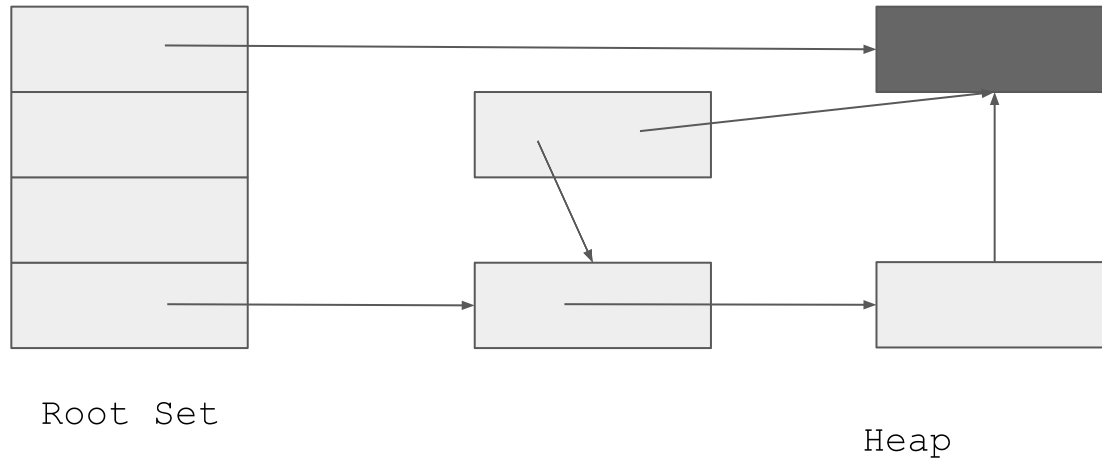
MS: Mark Phase



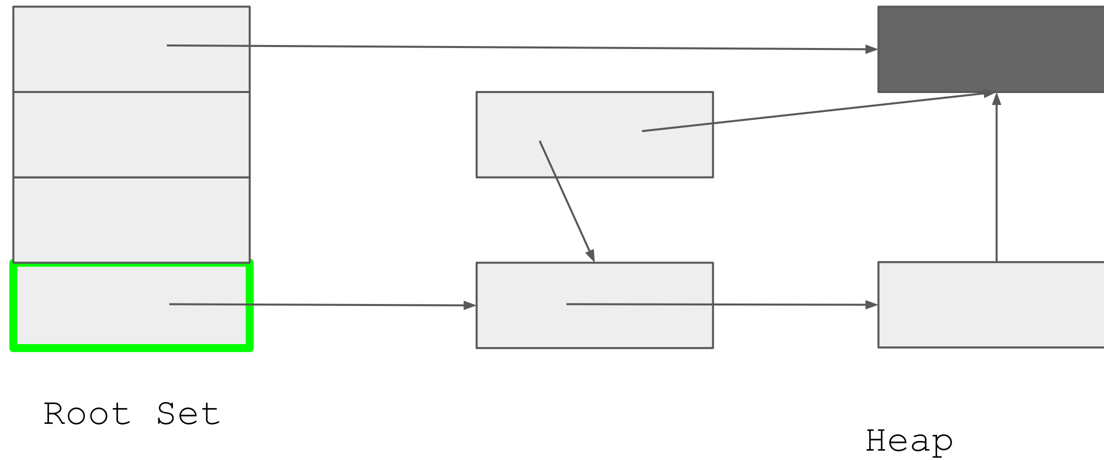
MS: Mark Phase



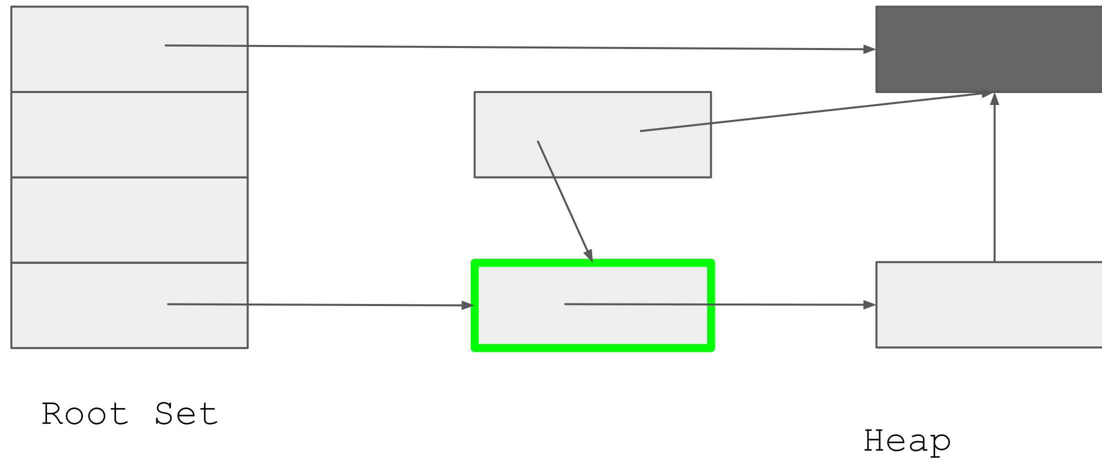
MS: Mark Phase



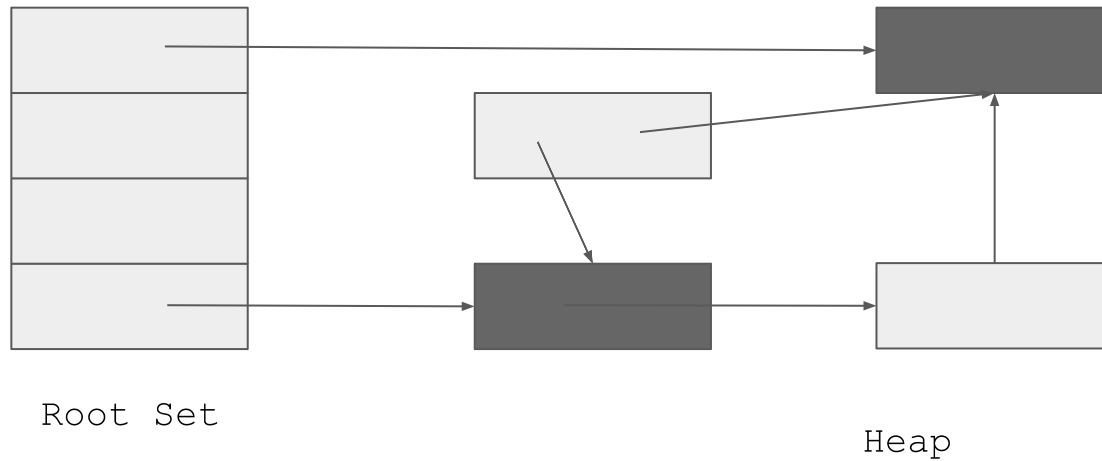
MS: Mark Phase



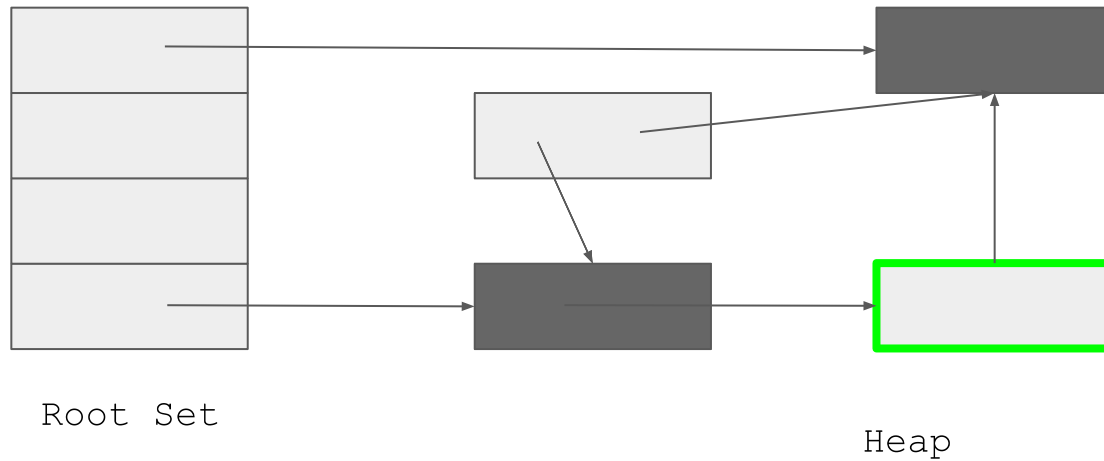
MS: Mark Phase



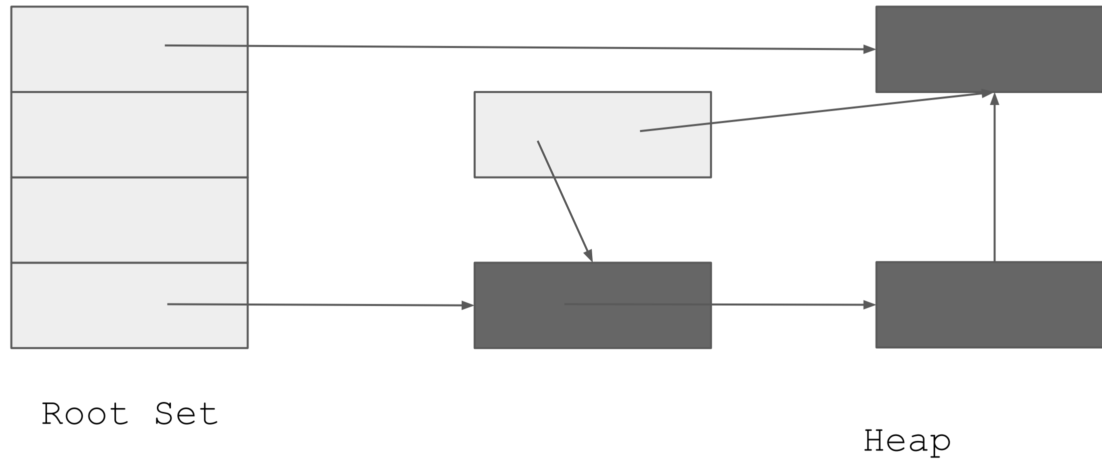
MS: Mark Phase



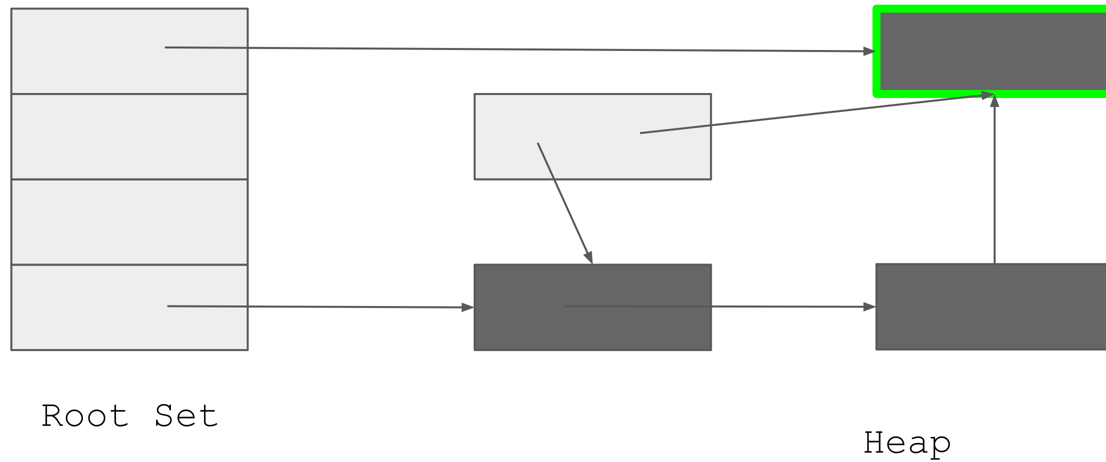
MS: Mark Phase



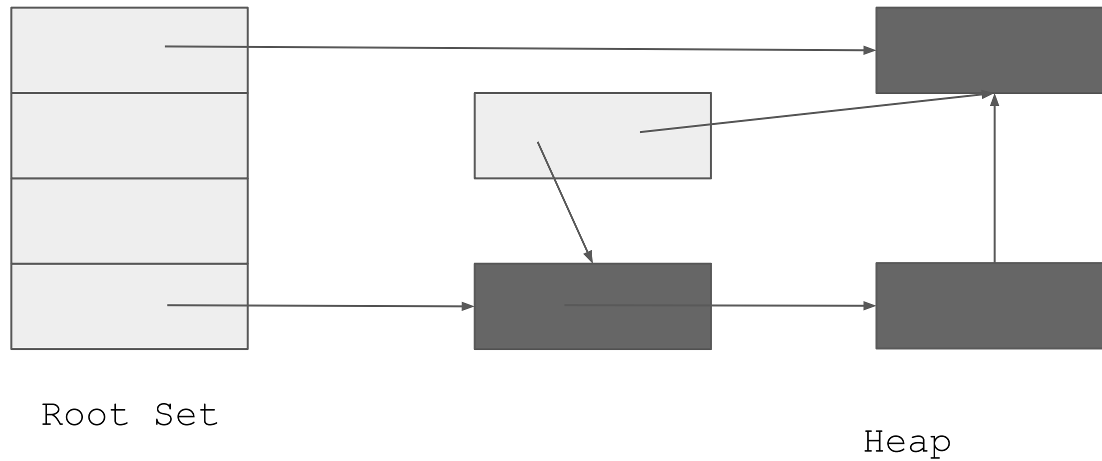
MS: Mark Phase



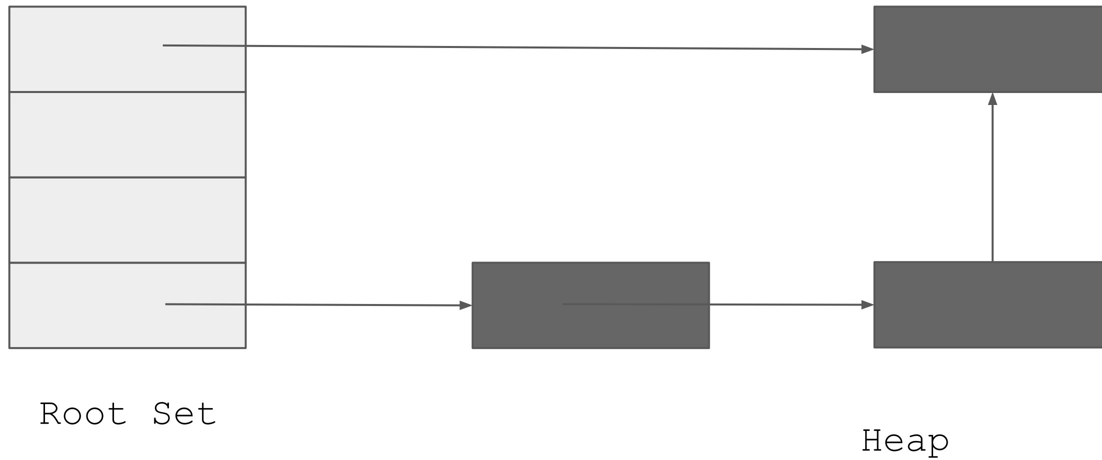
MS: Mark Phase



MS: Sweep Phase



MS: Sweep Phase



Mark Sweep: Pros and Cons

- Pros
    - Can collect dead object cycles
    - Marking an object is inexpensive
    - Little to no overhead while application runs
- Cons
    - Must pause the application to mark and sweep
    - Marking phase is slow if there are a lot of live objects
    - Must traverse the whole heap to sweep dead objects
    - Still creates fragmentation

1. Garbage Collection (GC)
2. GC Algorithms
    a. Reference Counting
    b. Mark Sweep
    c. Copying
    d. Generational and other Modifiers
3. GC in Practice

Aside: Copying Collector has many names

- Semispace Collector
- Scavenging Collector
- Stop and Copy collector
- **Copying Collector**

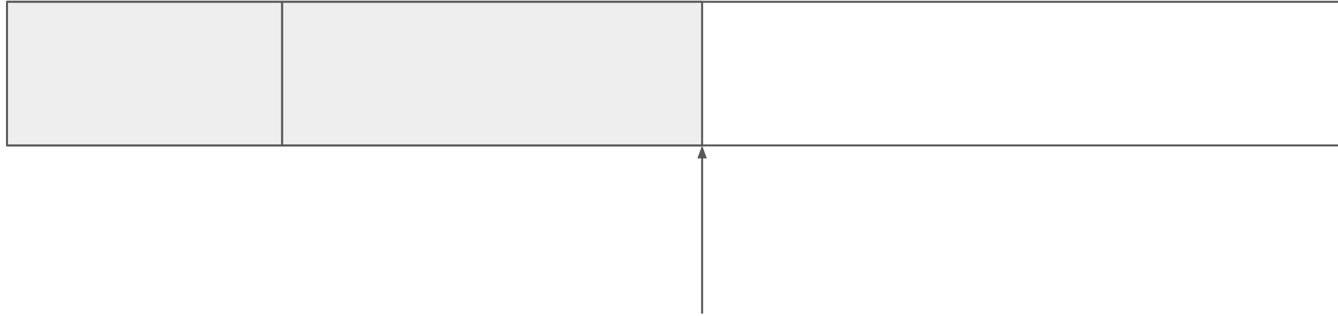
Copying Algorithm: Heap Layout

- Divide the heap into two equal-sized parts, From-Space and To-Space
- Application allocates into the From-Space

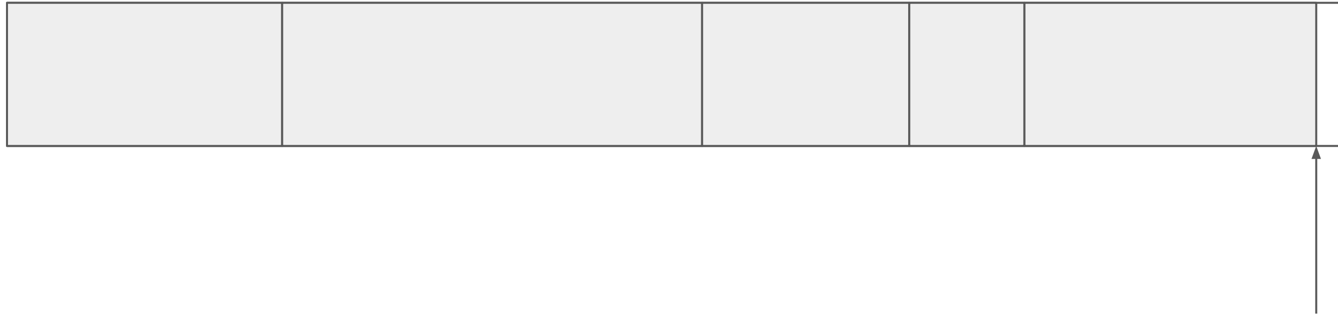
Copying Algorithm: Bump Pointer Allocation

Copying Algorithm: Bump Pointer Allocation

Copying Algorithm: Bump Pointer Allocation



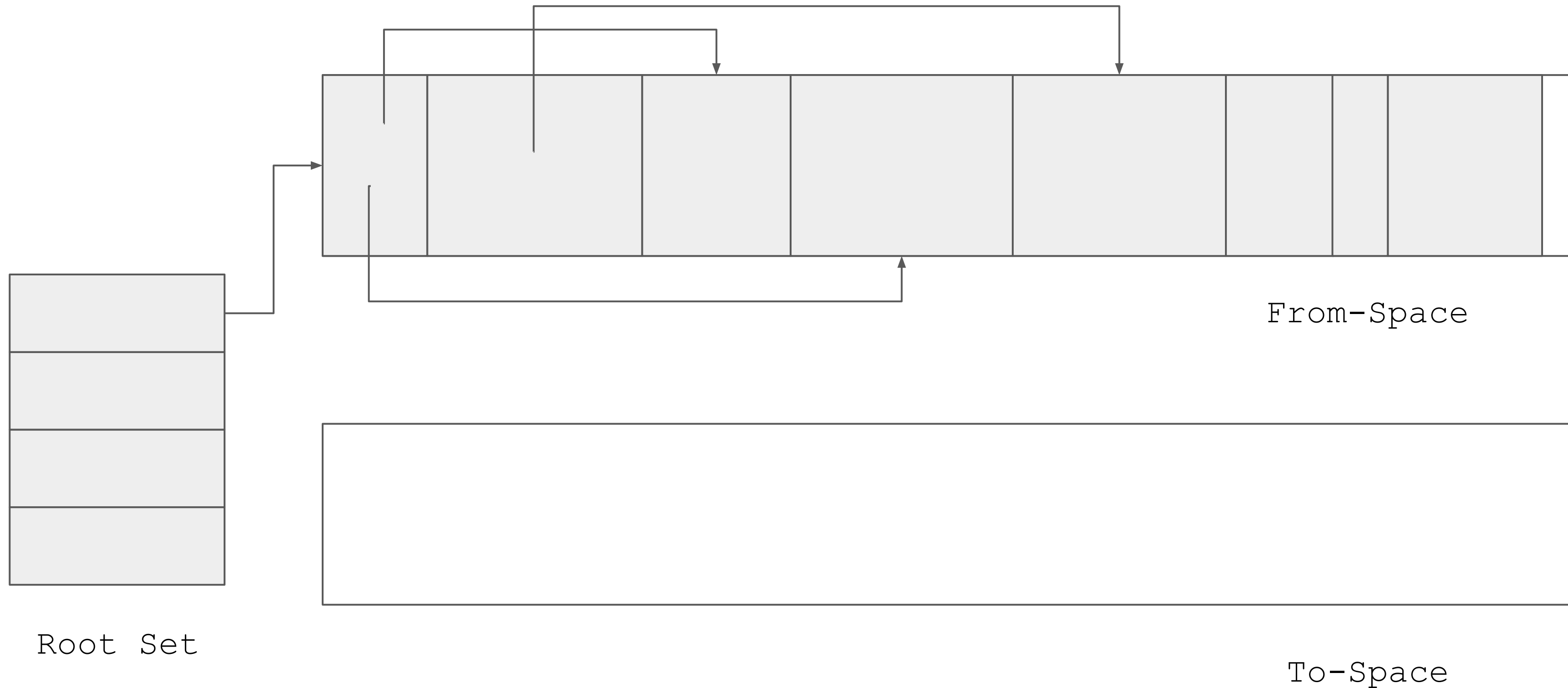
Copying Algorithm: Bump Pointer Allocation



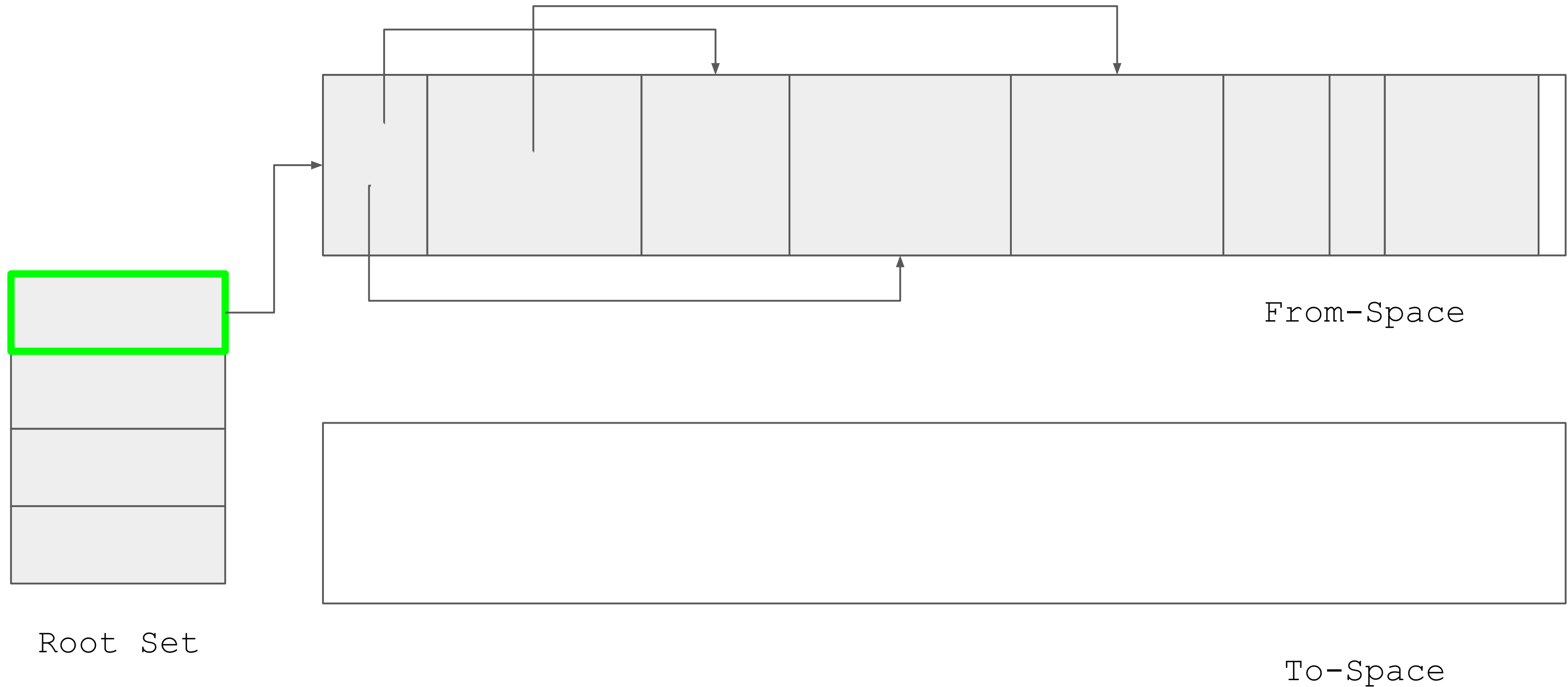
Copying Algorithm: Collection

- Visit objects using BFS starting at roots
- When visiting an object
    - Copy the object over to the To-Space
    - Update incoming pointer to the visited object
- At the end of GC, flip the two halves

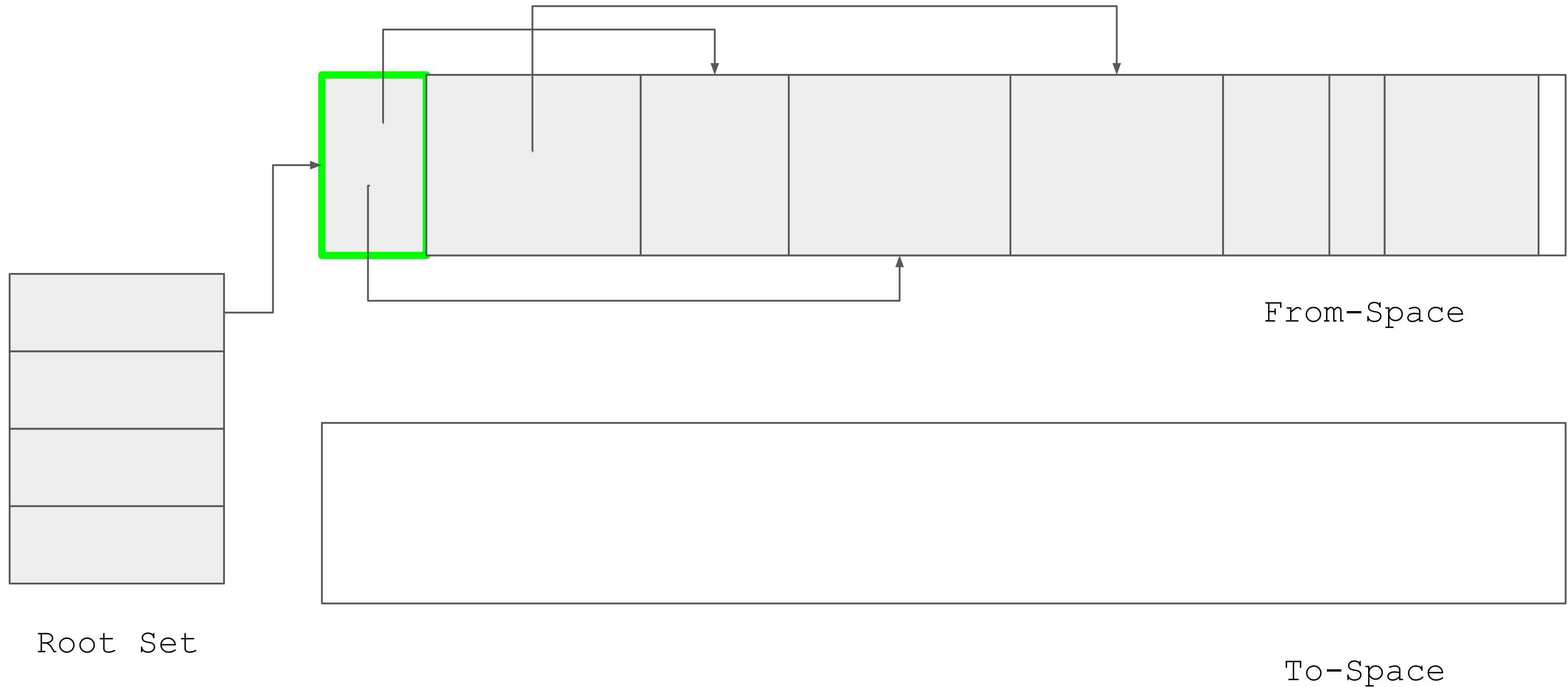
Copying Algorithm: Example



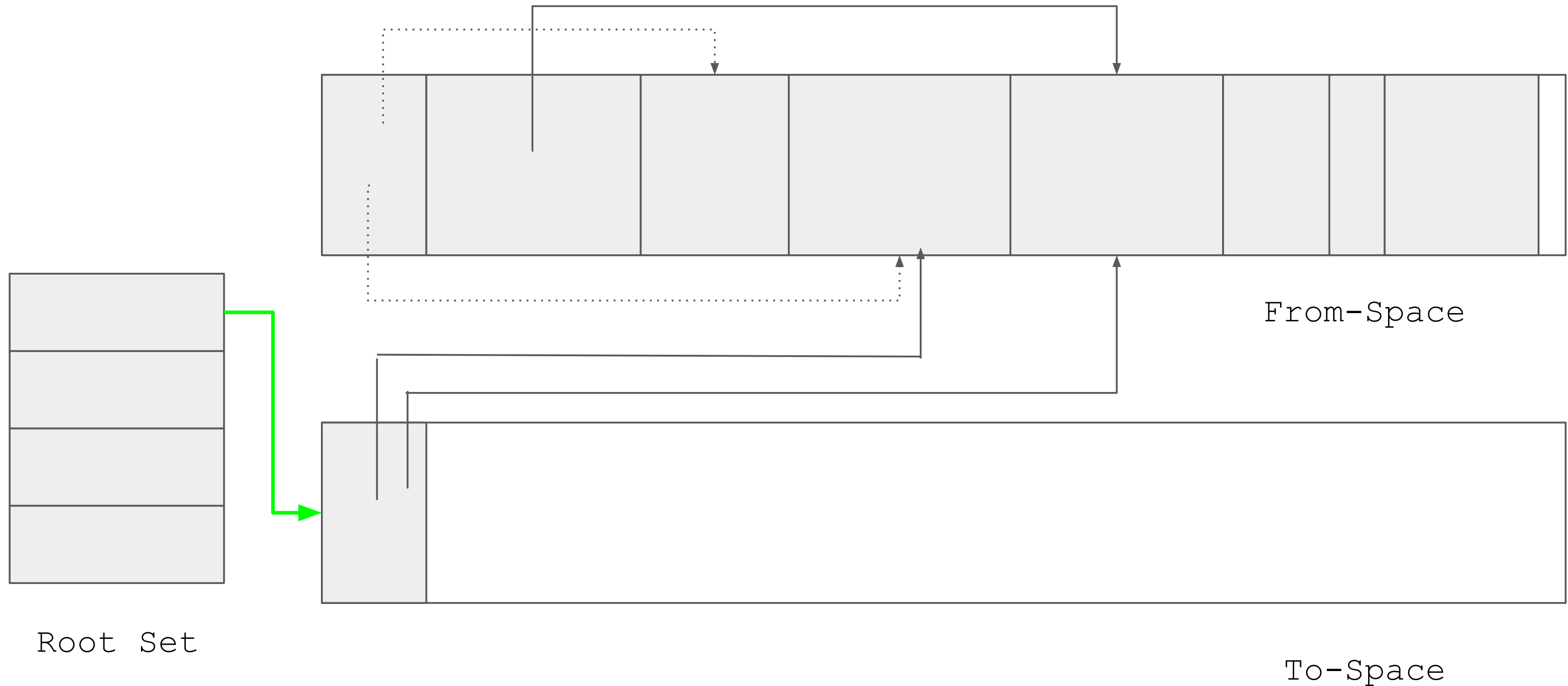
Copying Algorithm: Example



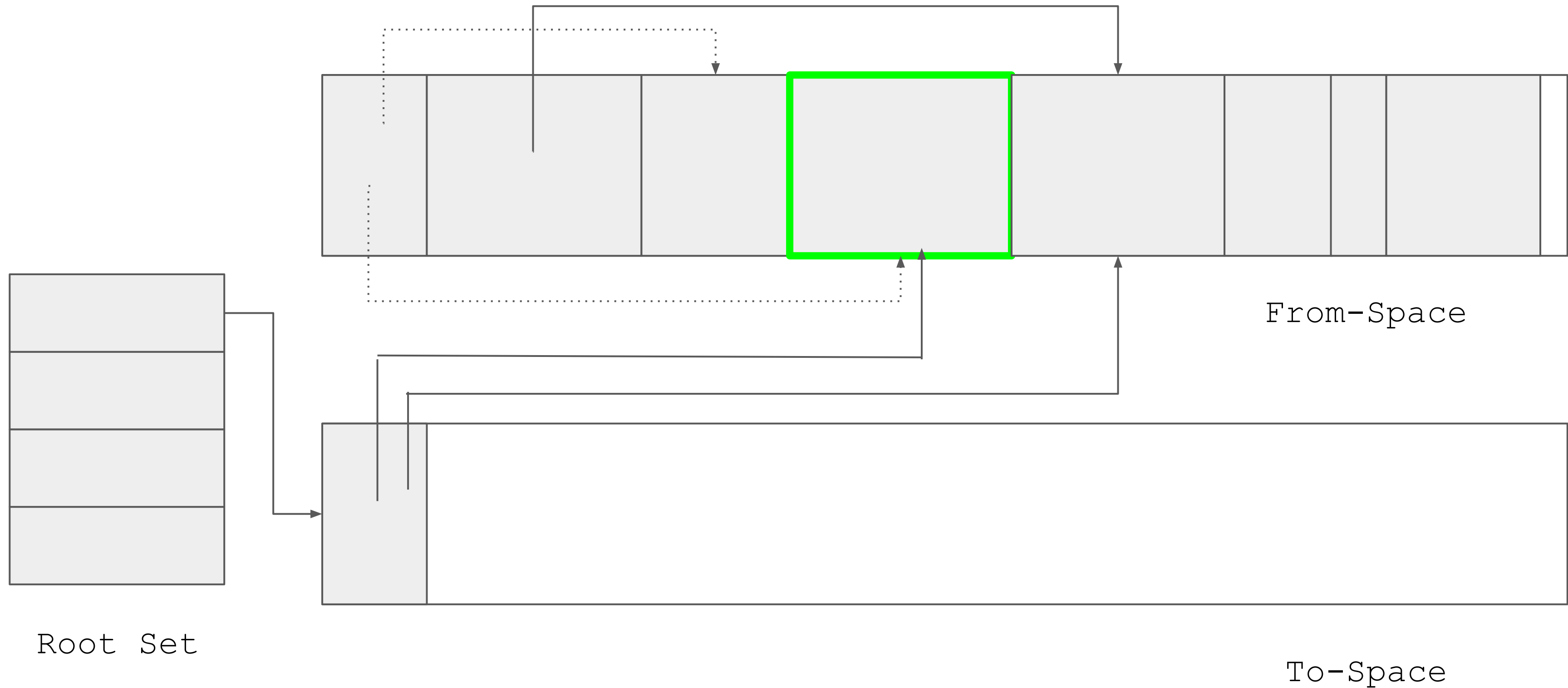
Copying Algorithm: Example



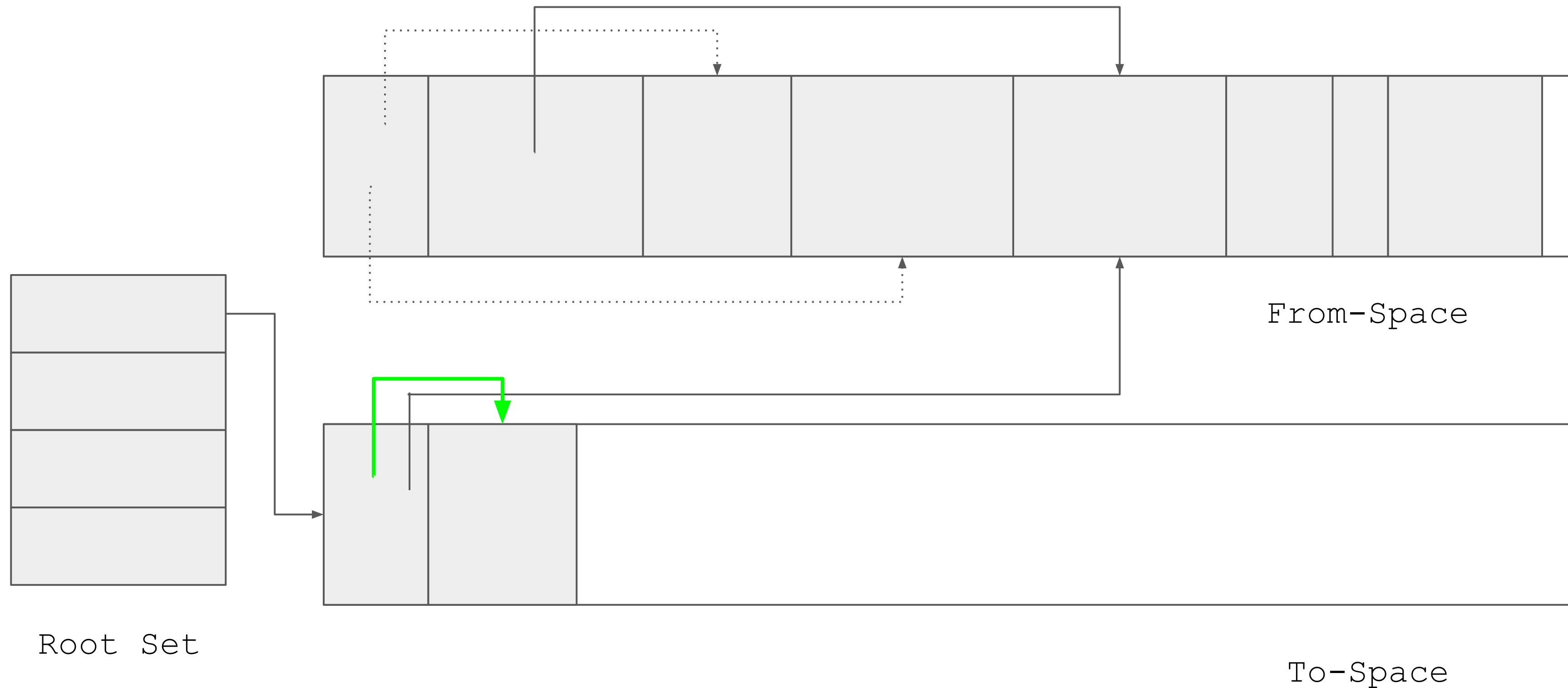
Copying Algorithm: Example



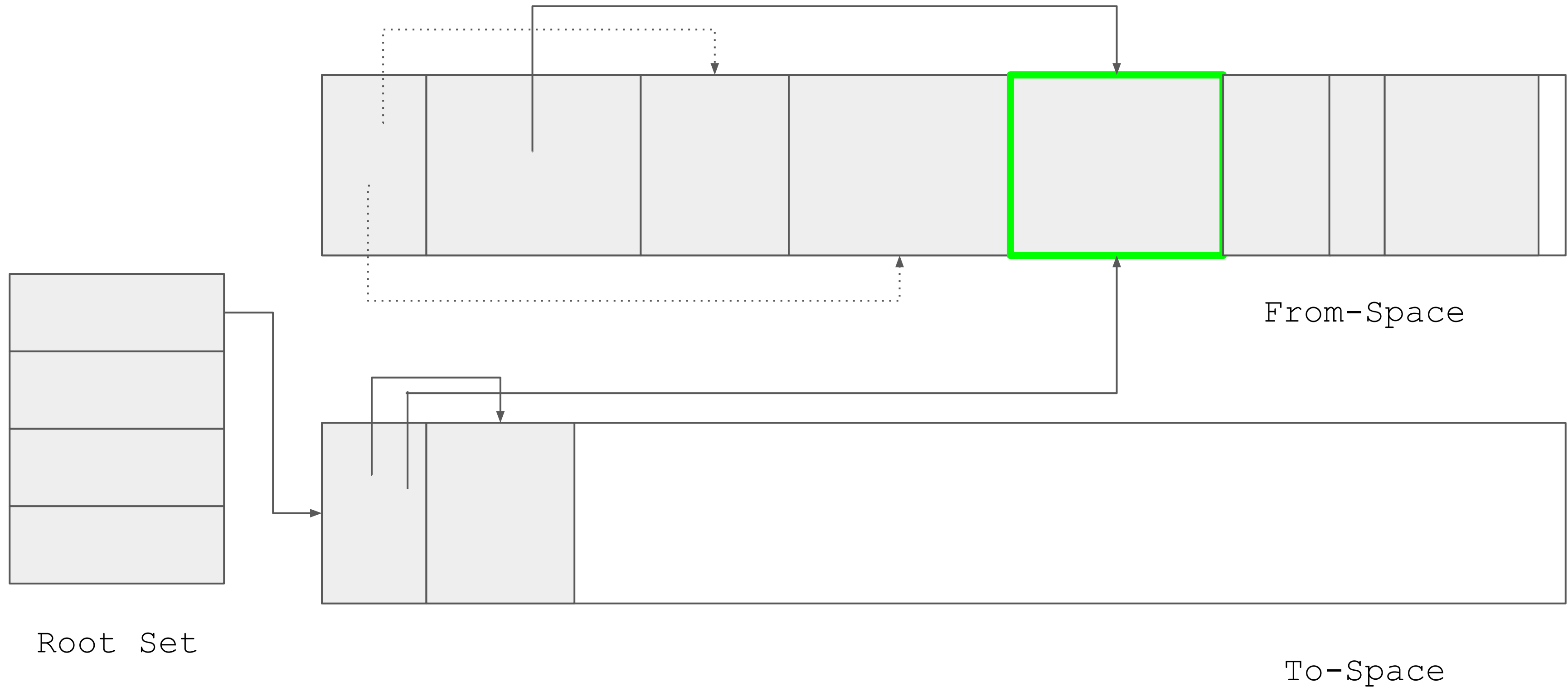
Copying Algorithm: Example



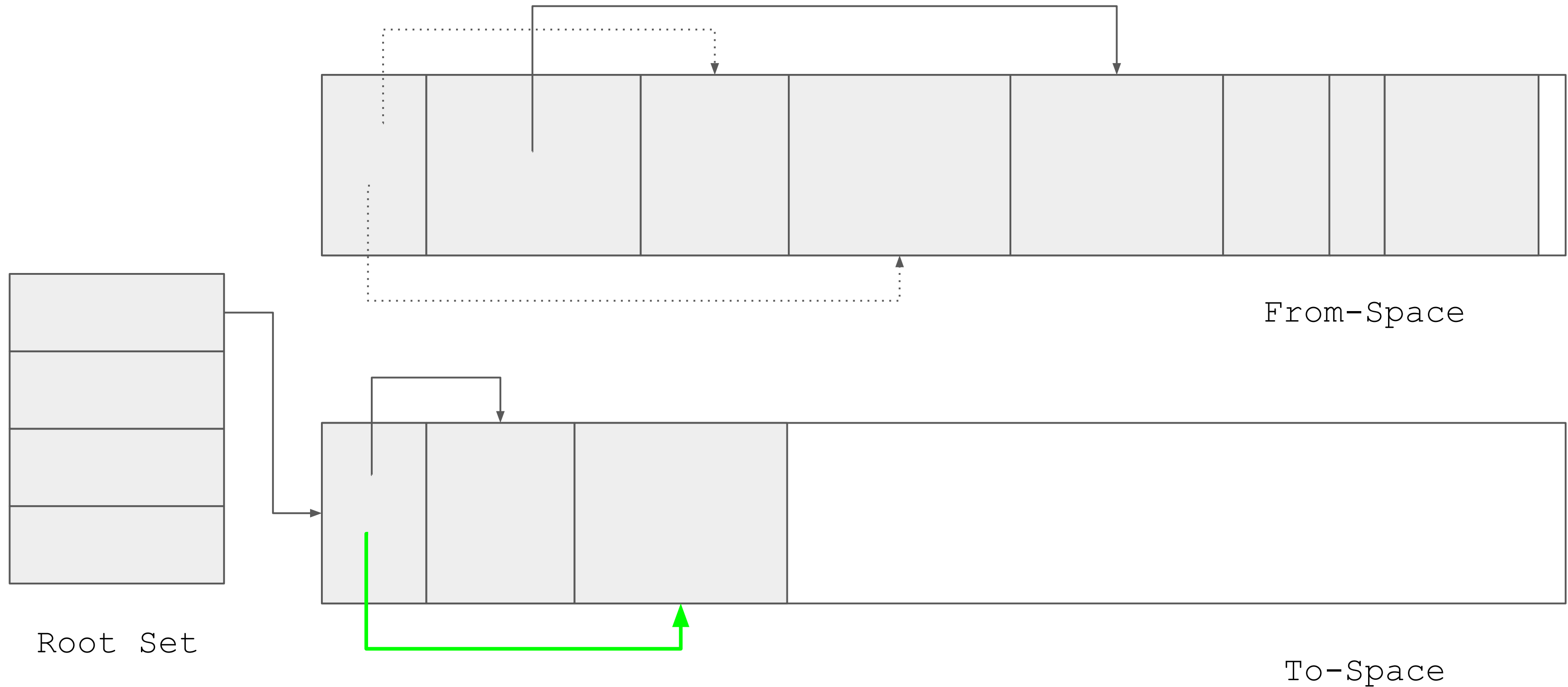
Copying Algorithm: Example



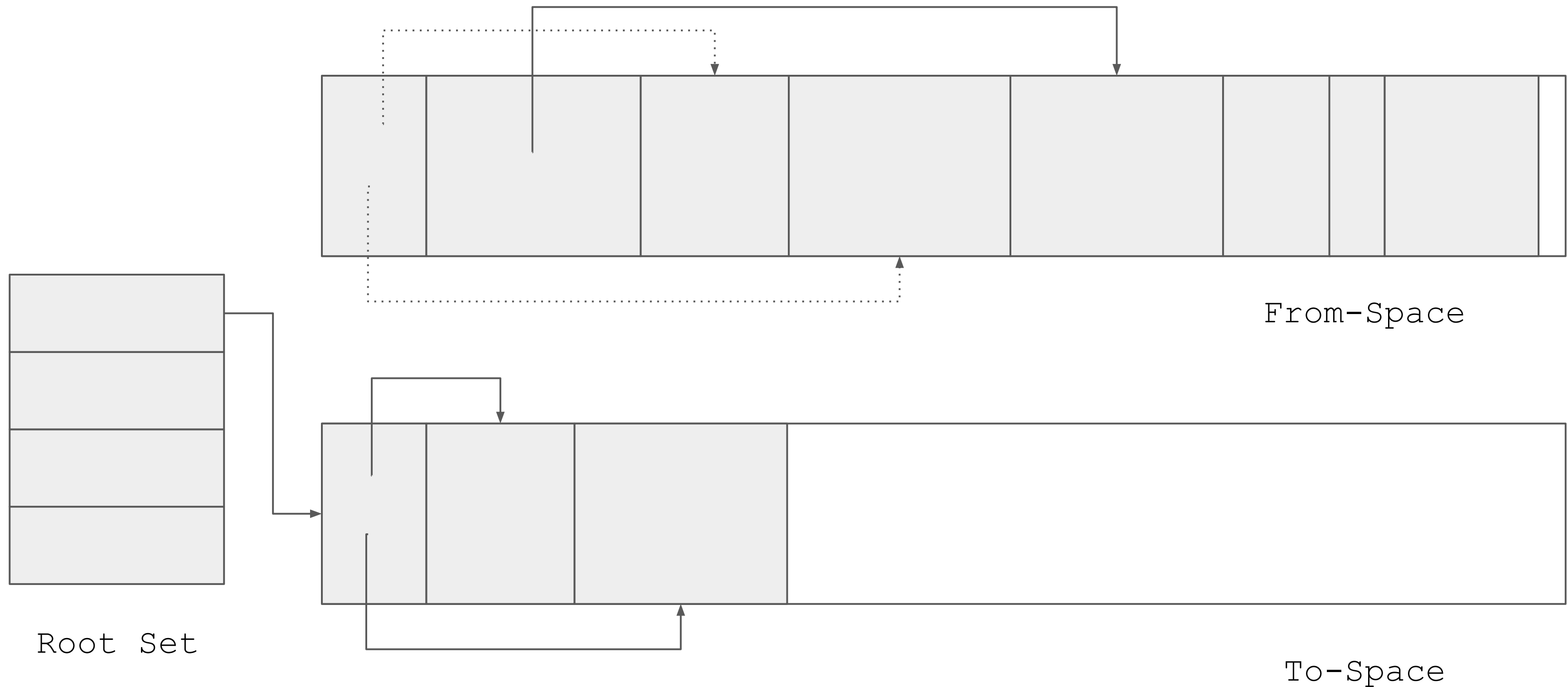
Copying Algorithm: Example



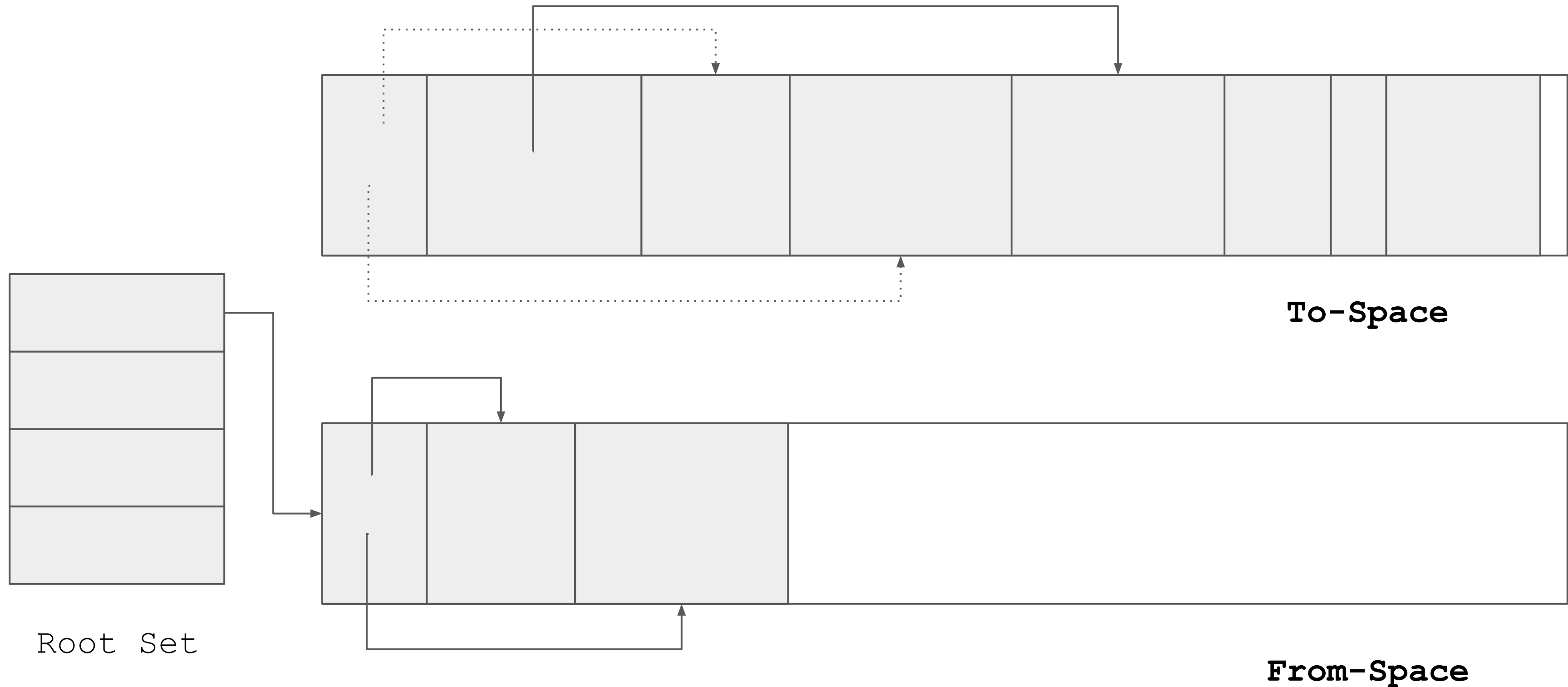
Copying Algorithm: Example



Copying Algorithm: Example



Copying Algorithm: Example



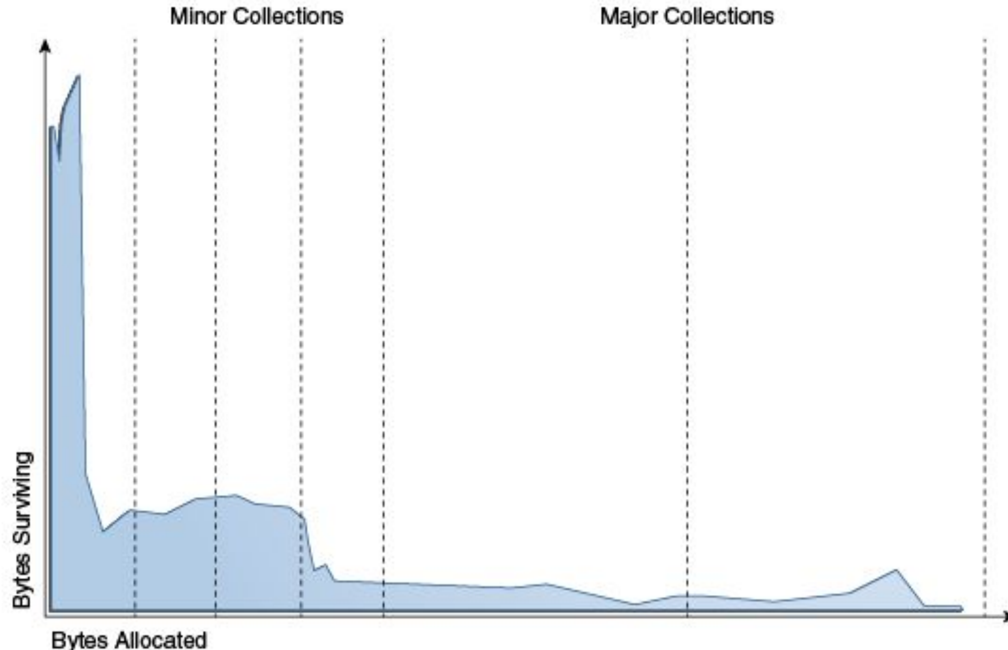
Copying: Pros and Cons

- Pros
  - Removes fragmentation
  - Doesn’t traverse whole heap to reclaim memory
  - Fast allocation
- Cons
  - Need 2x the heap
  - Must pause entire program to move objects

1. Garbage Collection (GC)
2. GC Algorithms
    a. Reference Counting
    b. Mark Sweep
    c. Copying
    d. Generational and other Modifiers
3. GC in Practice

Exploit generation hypothesis for faster GC

- Generational Hypothesis: “Most objects die young.” [Ungar 1984]



Picture from https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html

Exploit generation hypothesis for faster GC

- Split the heap into young and old partitions called **generations**.
  - Young generation usually called ***nursery*** or *eden*.
  - Old generation usually called ***mature space*** or *tenured generation*.
- Allocate into the nursery exclusively
- Collection Algorithm
  - Collect nursery when allocation fails.
  - Move young survivors to the mature space.
  - Collect mature space when whole heap is full.
- In practice, nursery collected using a copying collector.
- Collector for the mature space varies.

Algorithms can be improved further.

- Concurrent Marking
  - GC does some marking while application is running
- Compacting Pass
  - Modern MS implementations compact the heap after a GC.
- Add a MS or copying collector as a backup
  - RC collectors usually have a MS collector to clean up object cycles every so often
- Different heap layouts
  - e.g. Copying collector’s heap layout
  - e.g. Generations
  - e.g. Split heap into buckets of similar sized objects
  - ...

Algorithms can be improved further.

- Lazy Sweeping
  - Sweep the heap as needed while application runs
- Compiler Optimizations
  - In RC, remove increments immediately followed by decrements to same object

Actual GCs in Java Right Now

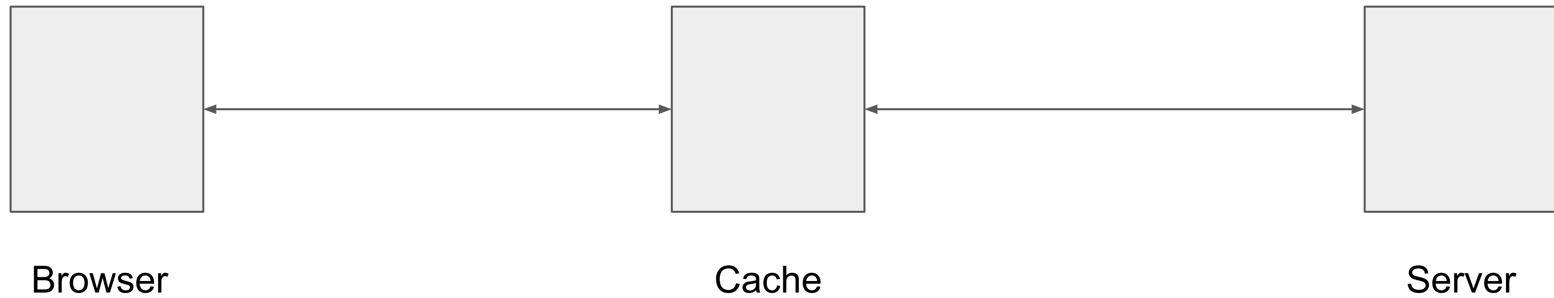
- Generational Mark Sweep (-XX:+UseSerialGC)
  - Old generation collected using compacting Mark Sweep.
- Concurrent Generational Mark Sweep (-XX:+UseParallelGC)
  - Same as above, but marks concurrently with user application.
- G1 (-XX:+UseG1GC)
  - Concurrent generational Mark Sweep GC that guarantees a pause time.
  - Divides the heap into equally sized regions.
  - Each region is either a nursery or mature region.
  - Uses statistics to determine which mature regions have the most dead objects.
- These are *real* java flags
  - `java -XX:+UseG1GC -verbose:gc -XX:+PrintGCDetails HelloWorld`

1. Garbage Collection (GC)
2. GC Algorithms
    a. Reference Counting
    b. Mark Sweep
    c. Copying
    d. Generational and other Modifiers
3. GC in Practice

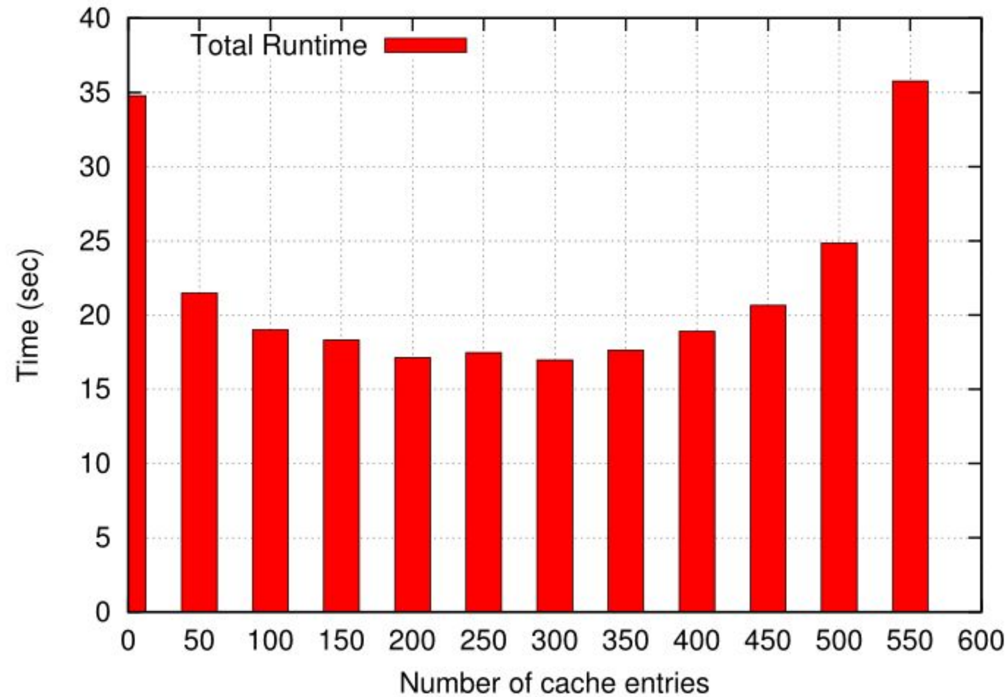
Memory correctness is not an issue

- Dangling pointers and use-after-free can’t happen
  - Would imply GC collected a live object.
- Double-free can’t happen
  - GC will only reclaim dead objects.
  - GC will reclaim a dead object exactly once.

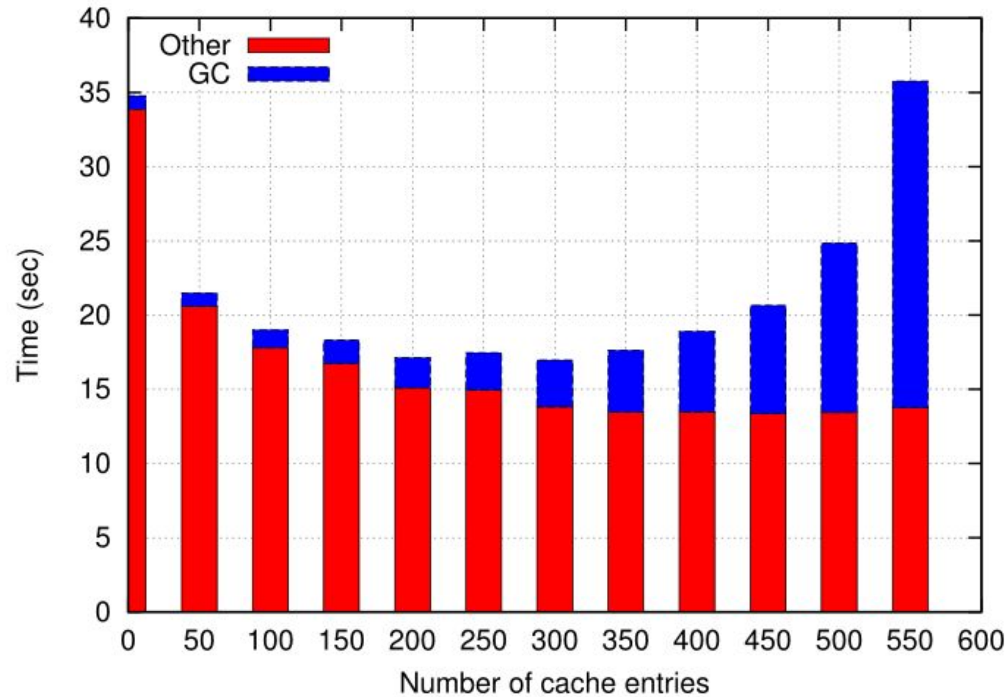
Example: Caching Web Requests



Performance is an issue



GC dominates runtime



Dealing with Performance Issues Due to GC

- Allocate only the memory you need
  - Reduces the amount of live data and speeds up MS and Copying

- Null pointers to objects you know are dead
  - Allows GC to recover more memory from dead objects

```
public void foo() {
    Object o = new Object();
    // … use o only here
    o = null; // o is now dead
}
```

Questions?