Operating Systems
CMPSCI 377
Garbage Collection

Emery Berger
University of Massachusetts Amherst

Questions To Come

- Terms:
  - **Tracing**
  - **Copying**
  - **Conservative**
  - **Parallel GC**
  - **Concurrent GC**
- **Runtime** & **space** costs
  - **Live objects**
  - **Reachable objects**
  - **References**

Explicit Memory Management

- `malloc/new`
  - allocates space for an object
- `free/delete`
  - returns memory to system

- Simple, but tricky to get right
  - Forget to **`free`** ⇨ memory leak
  - **`free`** too soon ⇨ "dangling pointer"
  - Double **`free`**, invalid **`free`**...

Dangling Pointers

```
Node x = new Node (“happy”);
Node ptr = x;
delete x;           // But I’m not dead yet!
Node y = new Node (“sad”);
cout << ptr->data << endl;      // sad ☹
```

- Insidious, hard-to-track down bugs

Solution: Garbage Collection

- **Garbage collector** periodically scans objects on heap
  - No need to `free`
  - **Automatic memory management**

- Reclaims ***non-reachable*** objects
  - Won't reclaim objects until they're **dead** (actually somewhat later)

No More Dangling Pointers

```
Node x = new Node ("happy");
Node ptr = x;
// x still live (reachable through ptr)
Node y = new Node ("sad");
cout << ptr->data << endl;  // happy! ☺
```

So why not use GC
all the time?

Because This Guy Says Not To

Slightly More Technically…

- "GC **impairs performance**"
  - Extra processing
    - *collection, copying*
  - Degrades cache performance (*ibid*)
  - Degrades page locality (*ibid*)
  - Increases memory needs
    - *delayed reclamation*

On the other hand…

- No, “GC **enhances performance!**”
  - Faster allocation
    - *pointer-bumping vs. freelists*
  - Improves cache performance
    - *no need for headers*
  - Better locality
    - *can reduce fragmentation, compact data structures according to use*

Outline

- Classical GC algorithms
- Quantifying GC performance
  - A hard problem
- Oracular memory management
- GC vs. `malloc/free` bakeoff

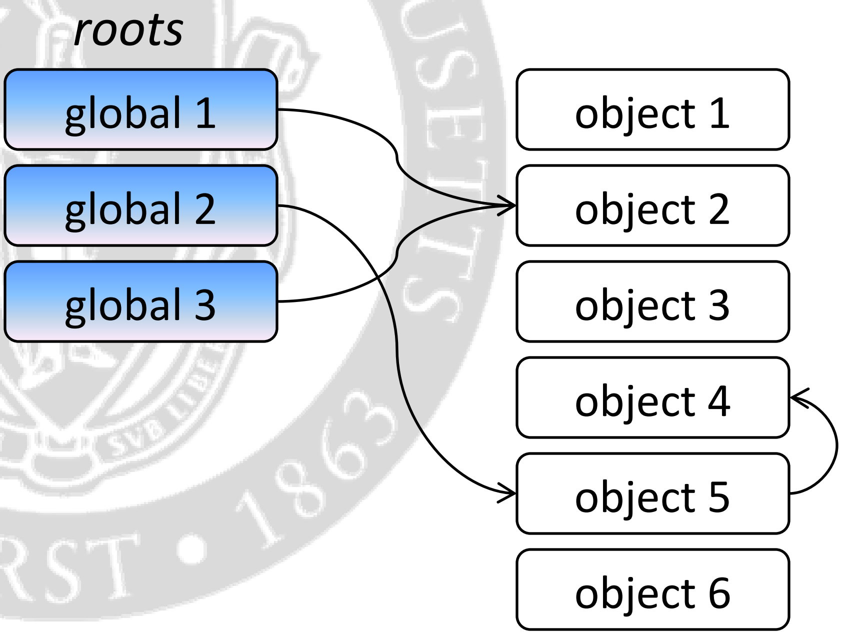
Classical Algorithms

- Three classical algorithms
  - **Mark-sweep**
  - **Reference counting**
  - **Semispace**
- Tweaks
  - **Generational garbage collection**
- Out of scope
  - **Parallel** – perform GC in parallel
  - **Concurrent** – run GC at same time as app
  - **Real-time** – ensure bounded pause times

Mark-Sweep

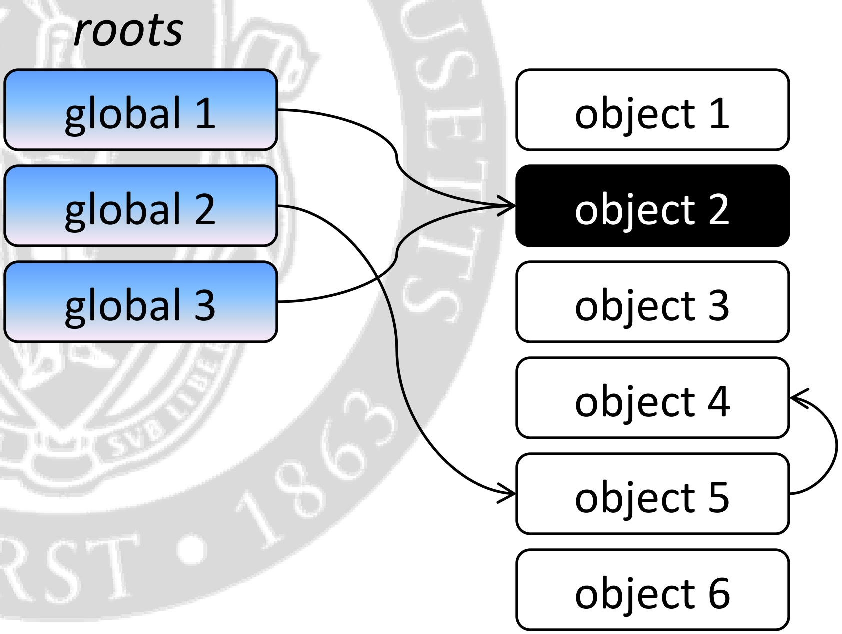
- Start with **roots**
  - Global variables, variables on stack & in registers
- Recursively visit every object through pointers
  - **Mark** every object we find (set **mark bit**)
- **Everything not marked = garbage**
  - Can then **sweep** heap for unmarked objects and free them

Mark-Sweep Example



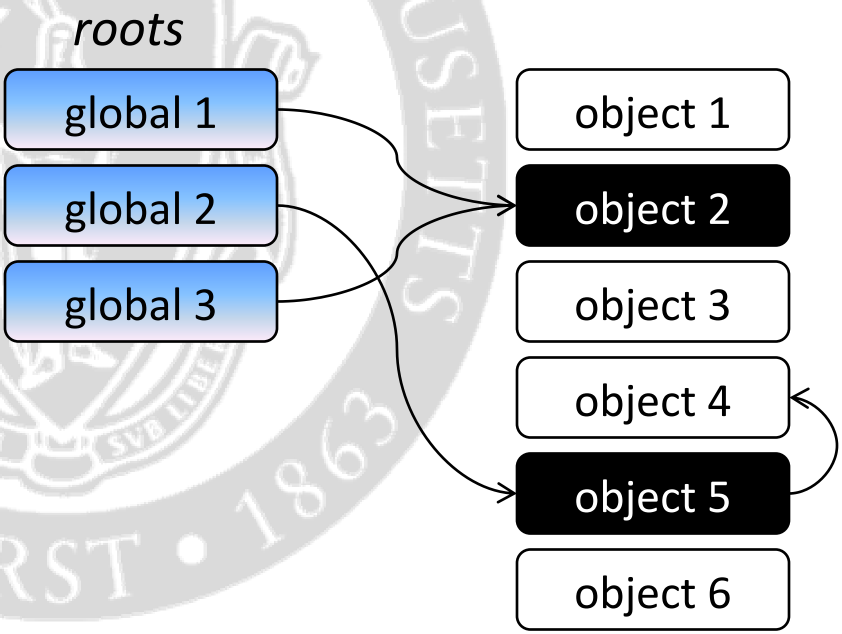
- Initially,
  all objects **white**
  (garbage)

Mark-Sweep Example



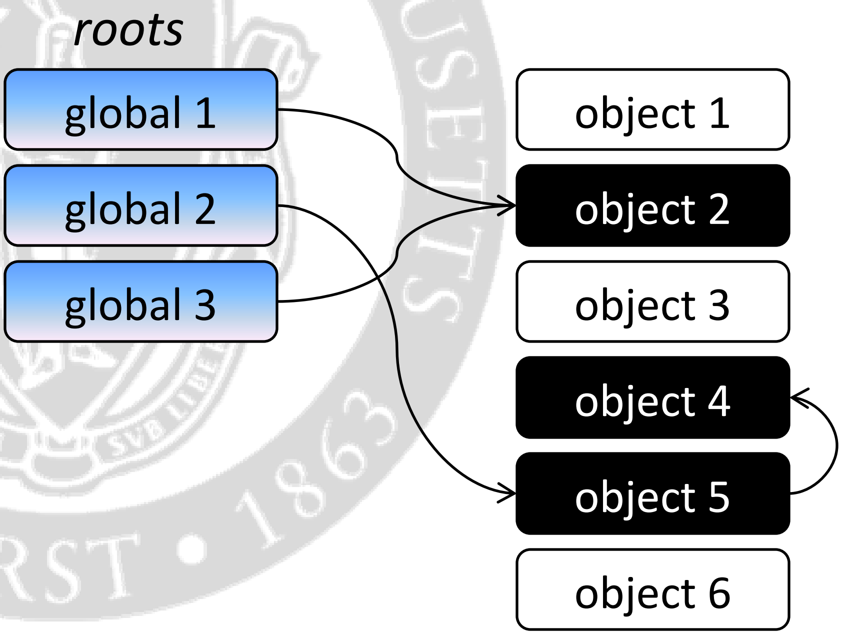
- Initially, all objects **white** (garbage)
- Visit objects, following **object graph**

Mark-Sweep Example



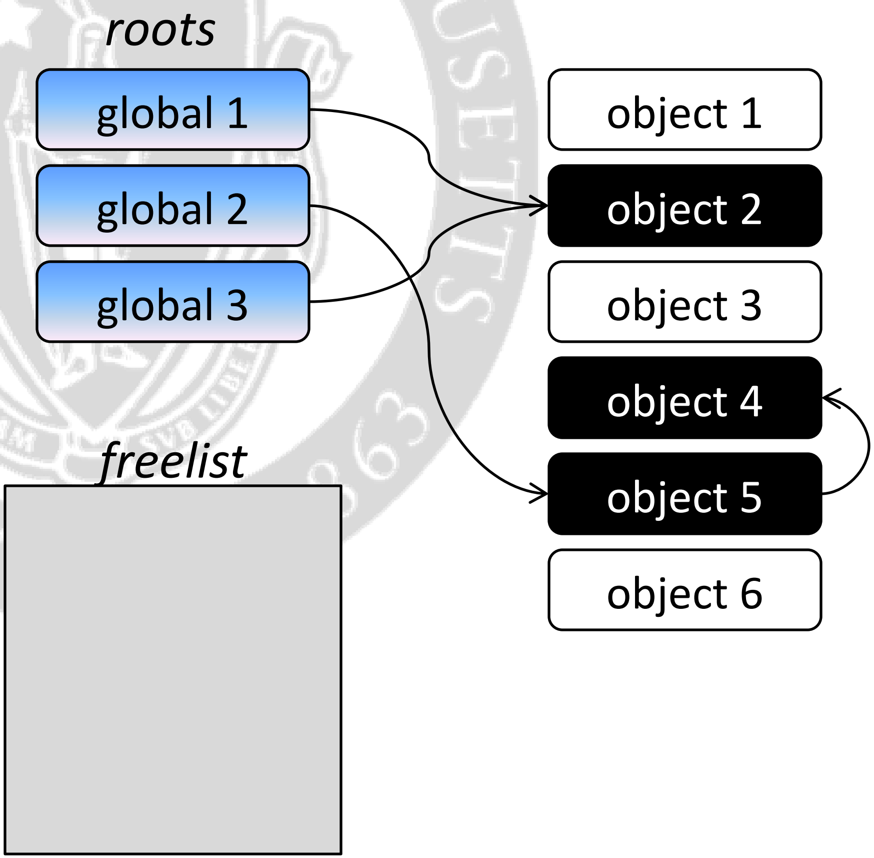
- Initially, all objects **white** (garbage)
- Visit objects, following **object graph**

Mark-Sweep Example



- Initially, all objects **white** (garbage)
- Visit objects, following **object graph**

Mark-Sweep Example

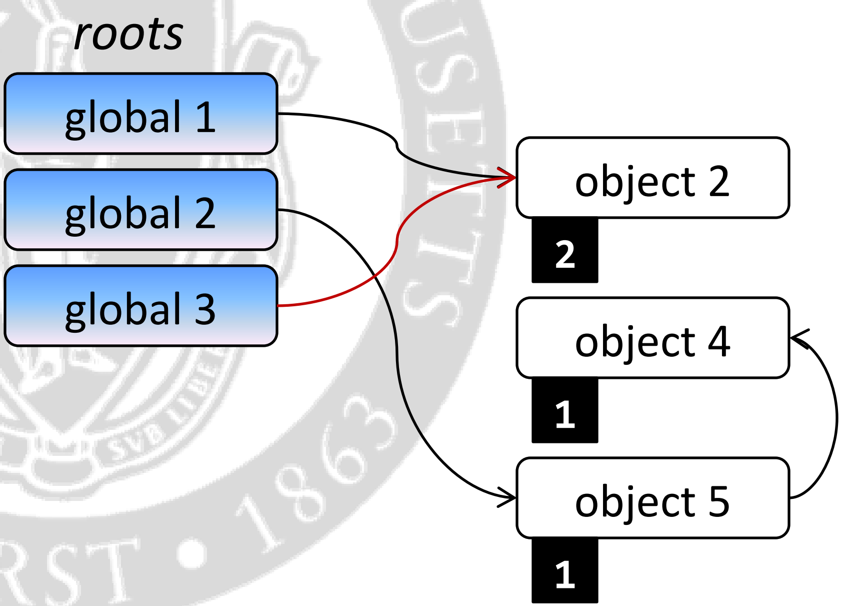


- Initially, all objects **white** (garbage)
- Visit objects, following **object graph**
- Can sweep immediately or **lazily**

Reference Counting

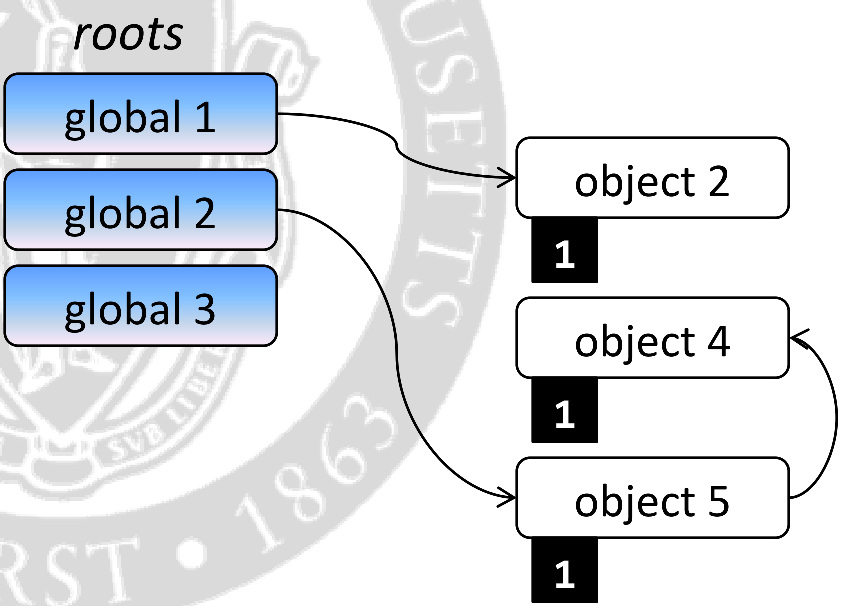
- For every object, maintain **reference count** = number of incoming pointers
  - `a->ptr = x` $\Rightarrow$ `refcount(x)++`
  - `a->ptr = y` $\Rightarrow$ `refcount(x)--; refcount(y)++`
- Reference count = 0 $\Rightarrow$ no more incoming pointers: **garbage**

Reference Counting Example



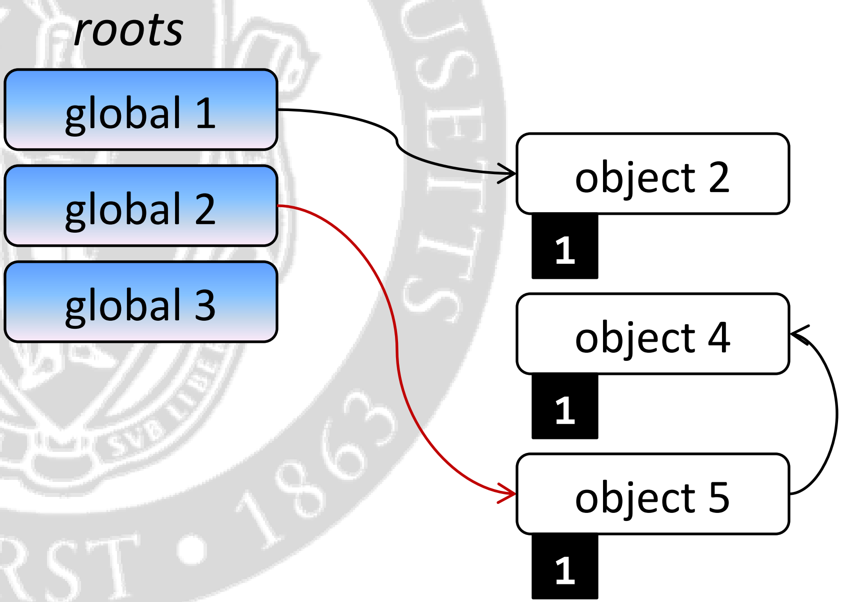
- New objects: ref count = 1

Reference Counting Example



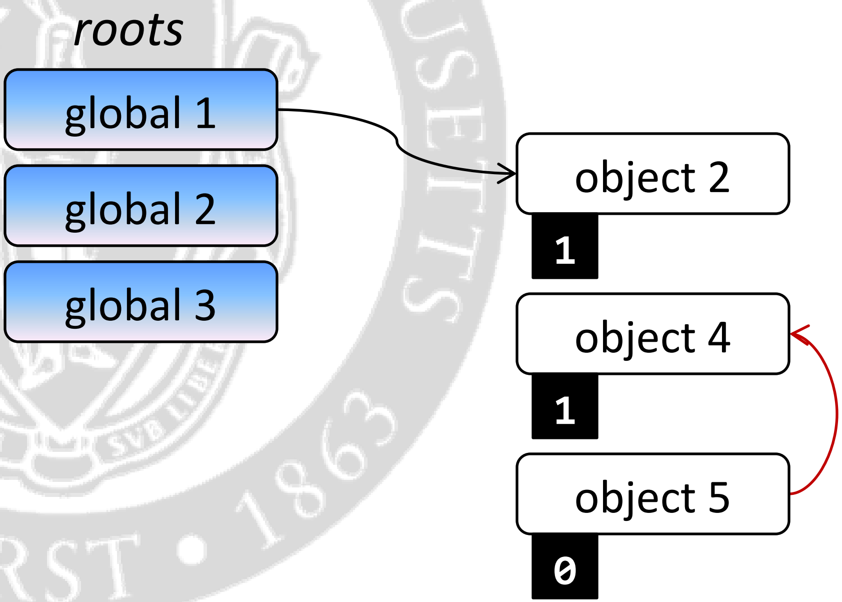
- New objects: ref count = 1
- Delete pointer: refcount--

Reference Counting Example



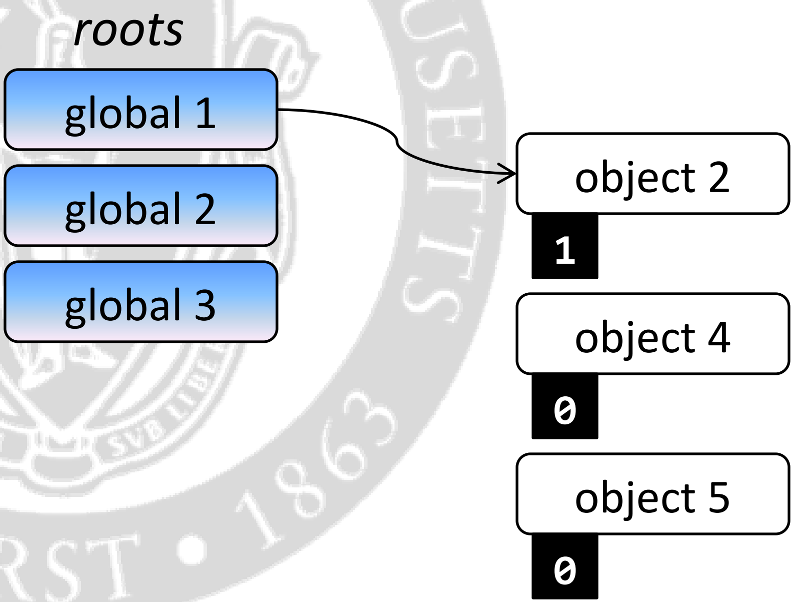
- New objects:
  ref count = 1
- Delete pointer:
  refcount--

Reference Counting Example



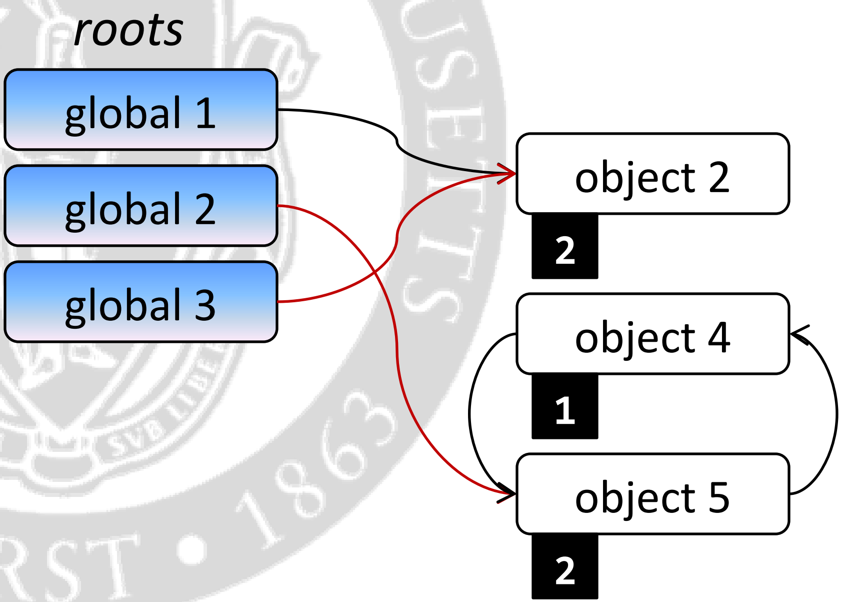
- New objects: ref count = 1
- Delete pointer: refcount--
  - And recursively delete pointers

Reference Counting Example



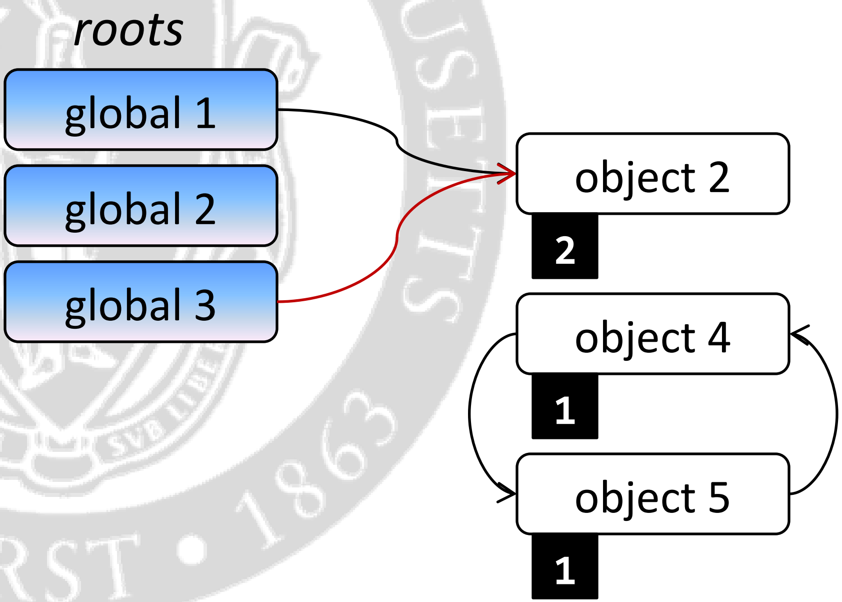
- New objects: ref count = 1
- Delete pointer: refcount--
  - And recursively delete pointers
- refcount == 0: put on freelist

Cycles & Reference Counting



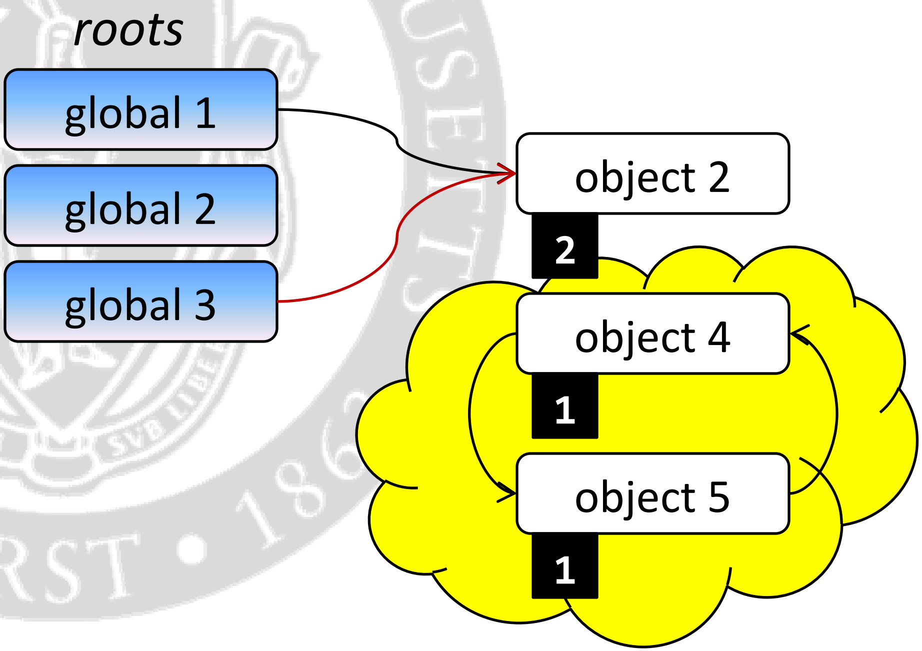
- **Big problem: cycles**

Reference Counting Example



- **Big problem: cycles**

Reference Counting Example

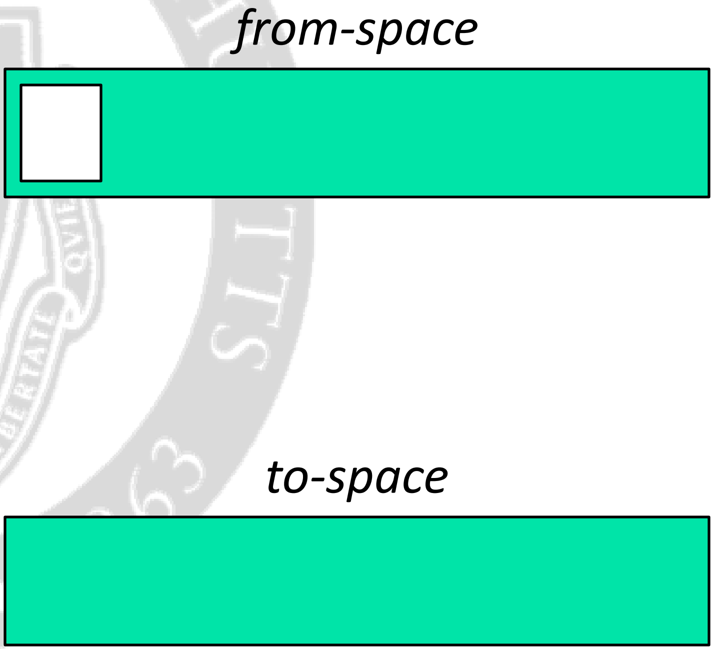


- **Big problem**: **cycles**
  - Cycles lead to unreclaimable garbage
- Need to do periodic tracing collection (e.g., mark-sweep)

Semispace

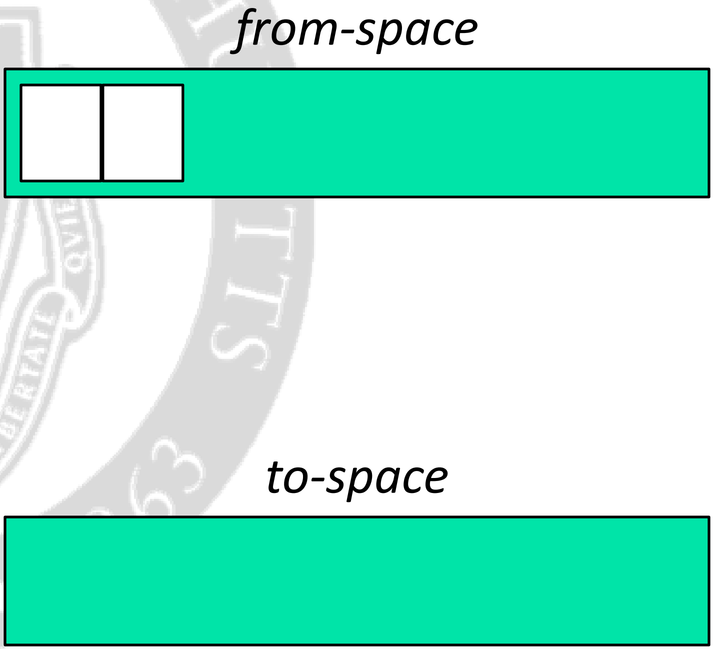
- Divide heap in two **semispaces**:
  - Allocate objects from **from-space**
- When from-space fills,
  - **Scan** from roots through live objects
  - **Copy** them into **to-space**
- When done, *switch the spaces*
  - Allocate from leftover part of to-space

Semispace Example



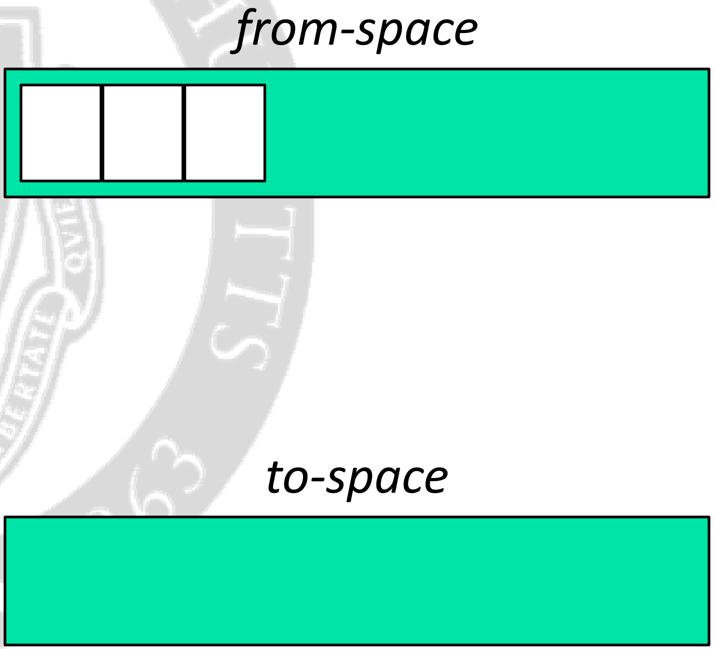
- Allocate in from-space
  - **Pointer bumping**

Semispace Example



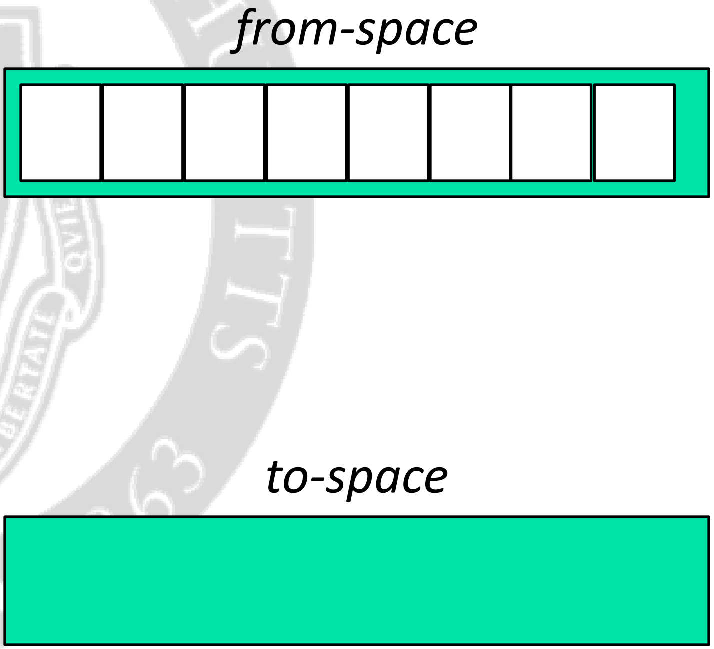
- Allocate in from-space
  - **Pointer bumping**

Semispace Example



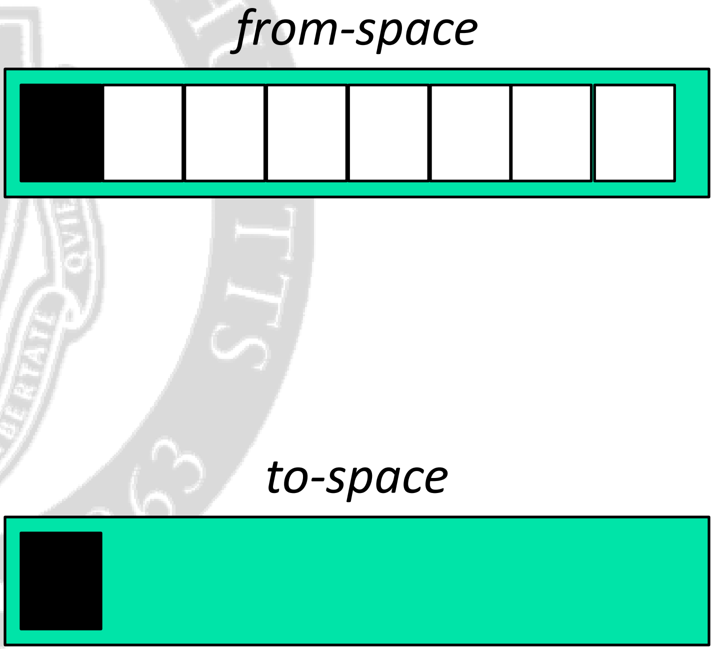
- Allocate in from-space
  - **Pointer bumping**

Semispace Example



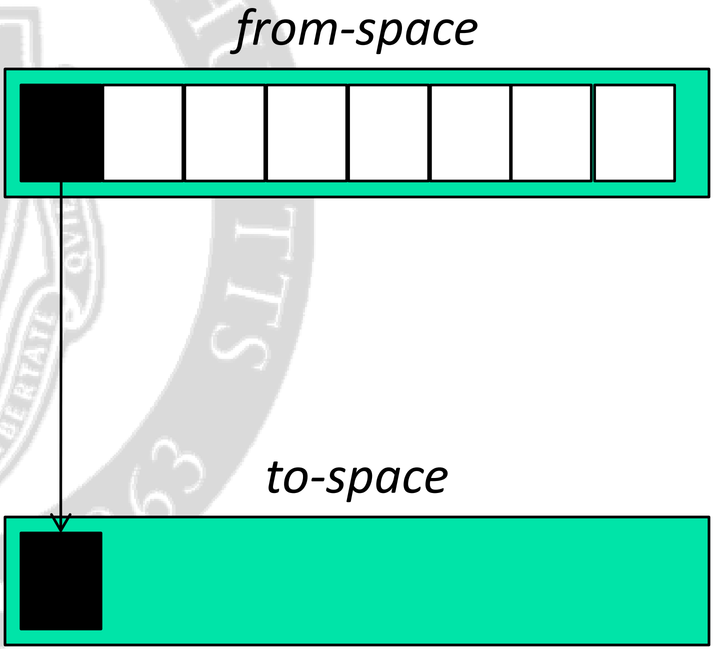
- Allocate in from-space
  - **Pointer bumping**
- Copy live objects into to-space

Semispace Example



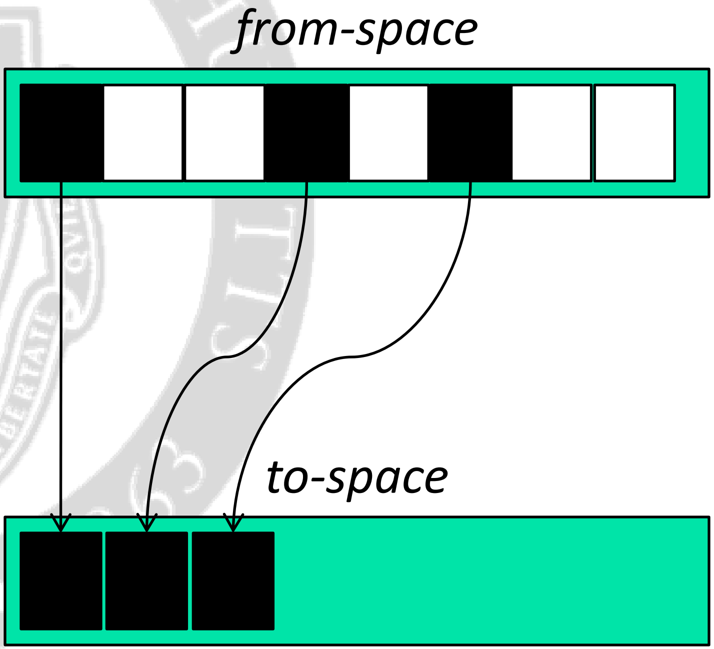
- Allocate in from-space
  - **Pointer bumping**
- Copy live objects into to-space
  - Leaves **forwarding pointer**

Semispace Example



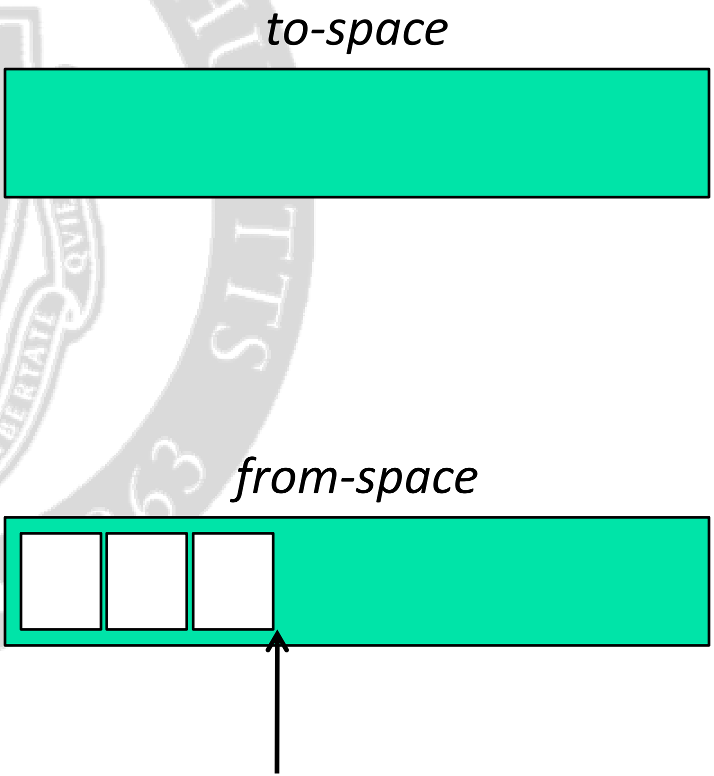
- Allocate in from-space
  - **Pointer bumping**
- Copy live objects into to-space
  - Leaves **forwarding pointer**

Semispace Example



- Allocate in from-space
  - **Pointer bumping**
- Copy live objects into to-space
  - Leaves **forwarding pointer**

Semispace Example

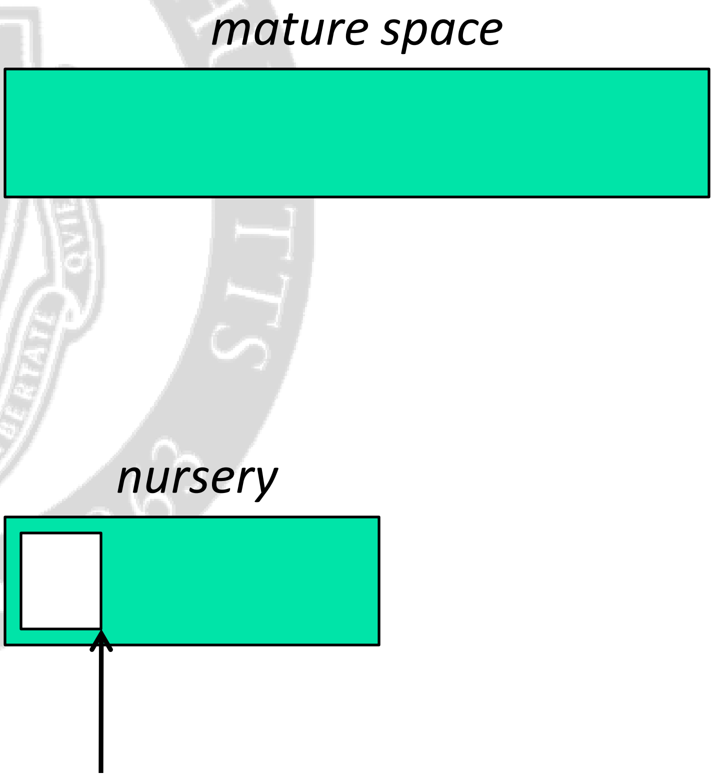


- Allocate in from-space
  - **Pointer bumping**
- Copy live objects into to-space
  - Leaves **forwarding pointer**
- **Flip** spaces; allocate from end

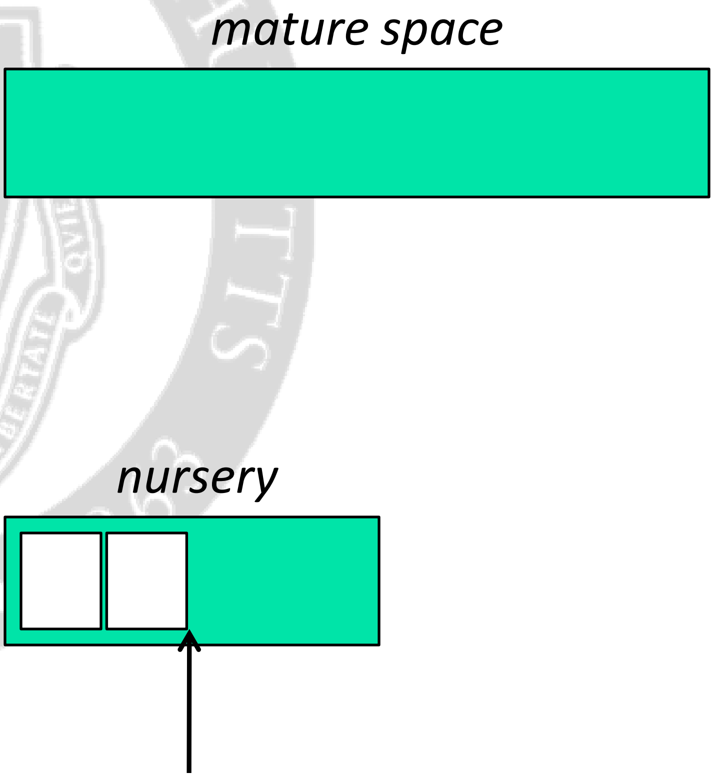
Generational GC

- Optimization for copying collectors
  - **Generational hypothesis:**
    "most objects die young"
  - **Common-case optimization**
- Allocate into **nursery**
  - Small region
  - Collect frequently
    - Copy out **survivors**
- **Key idea:** keep track of pointers from **mature space** into **nursery**

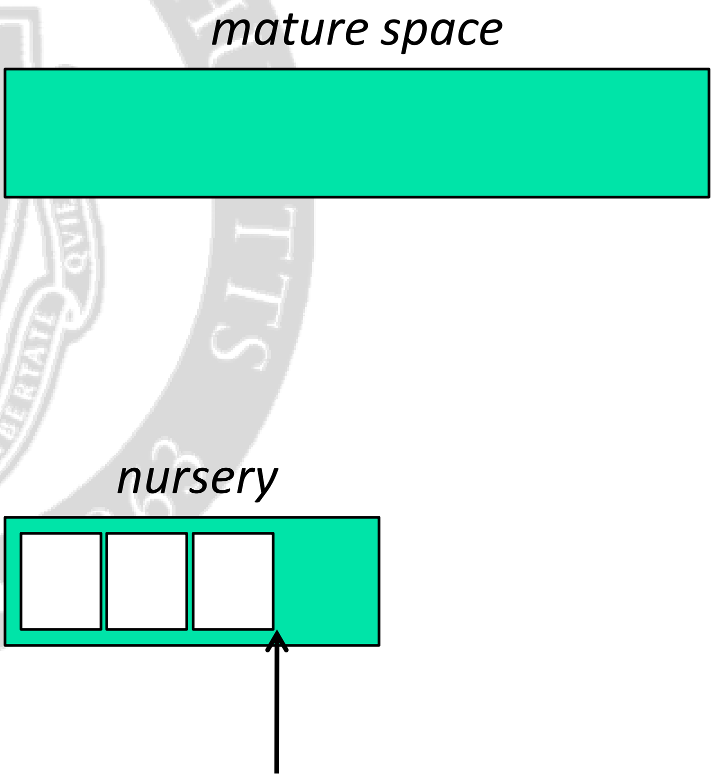
Generational GC Example



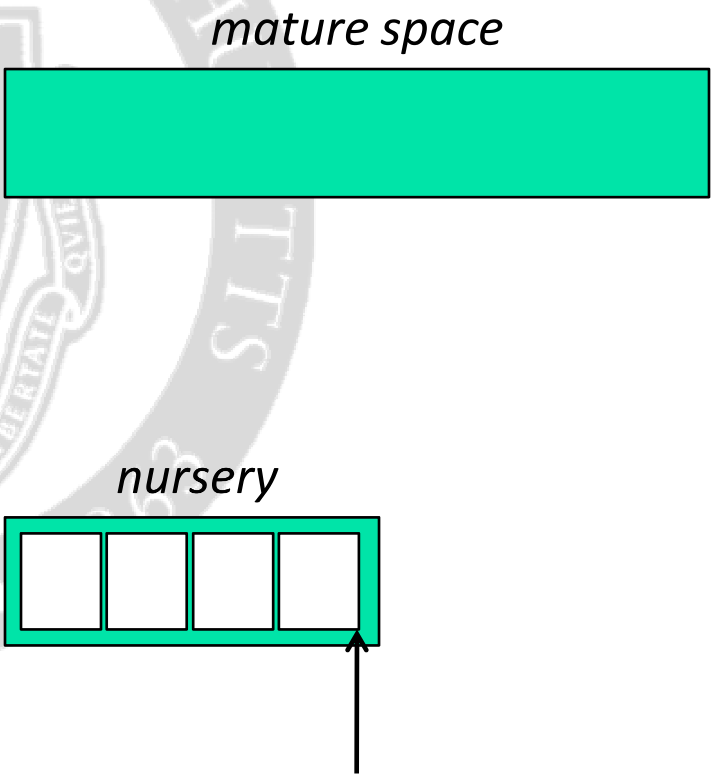
Generational GC Example



Generational GC Example

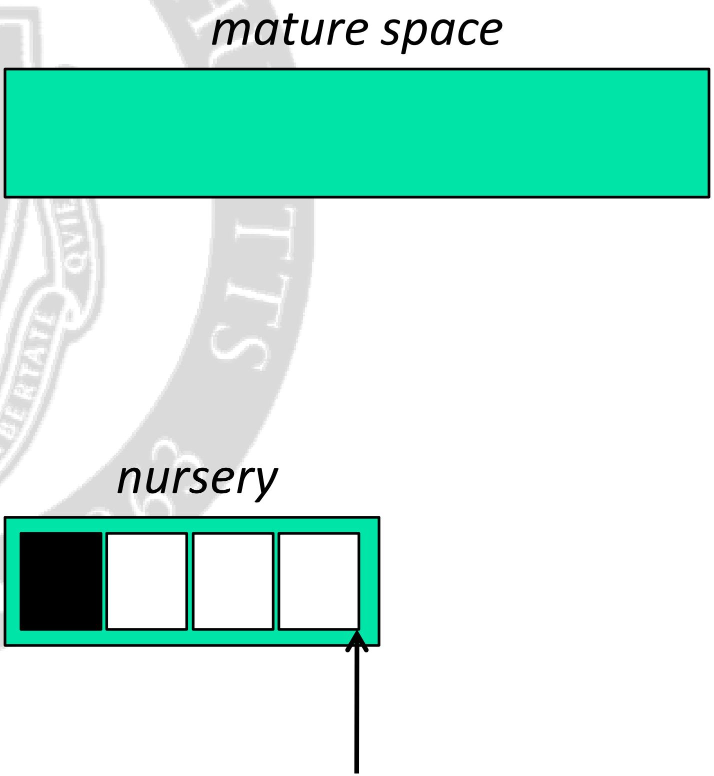


Generational GC Example



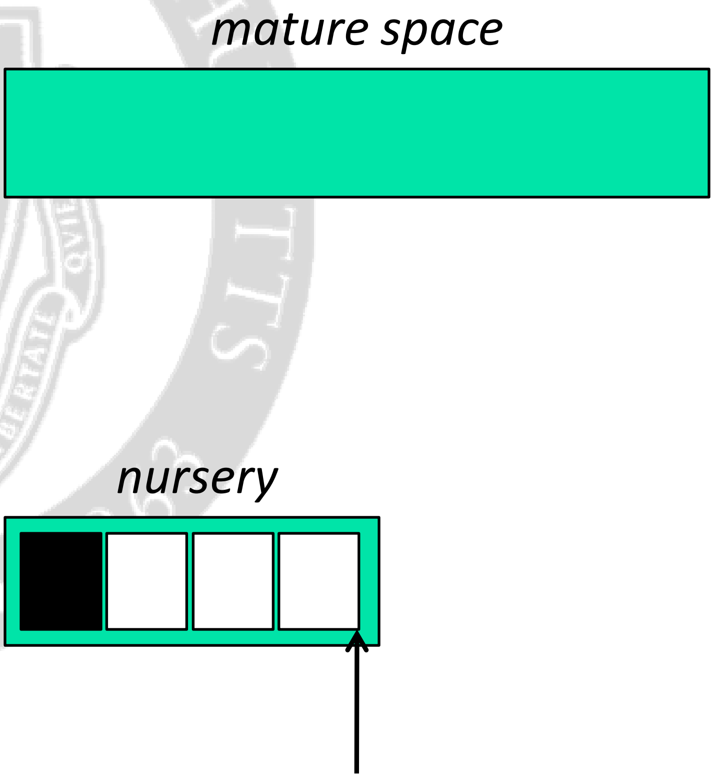
- Copy out survivors (via roots & mature space pointers)

Generational GC Example



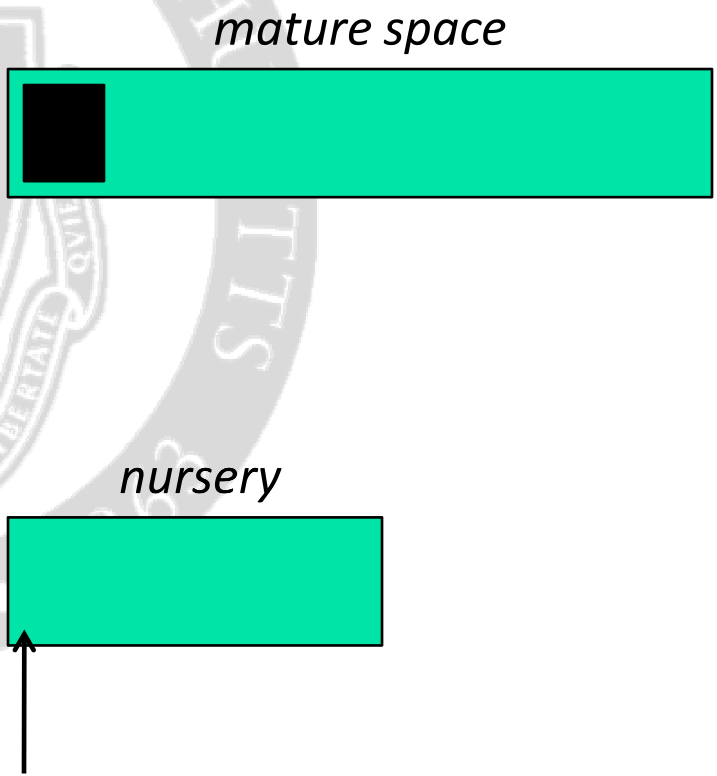
- Copy out survivors (via roots & mature space pointers)

Generational GC Example



- Copy out survivors (via roots & mature space pointers)

Generational GC Example



- Copy out survivors (via roots & mature space pointers)
- Reset allocation pointer & continue

Conservative GC

- **Non-copying** collectors for C & C++
  - Must identify pointers
    - "Duck test": if it looks like a pointer, it's a pointer
  - Trace through "pointers", marking everything
- Can link with Boehm-Demers-Weiser library ("libgc")

GC vs. malloc/free

Comparing Memory Managers

```
Node v = malloc(sizeof(Node));
v->data=malloc(sizeof(NodeData))
memcpy(v->data, old->data,
                sizeof(NodeData));
free(old->data);
v->next = old->next;
v->next->prev = v;
v->prev = old->prev;
v->prev->next = v;
free(old);
```

BDW Collector

Using GC in C/C++ is easy:

Comparing Memory Managers

```
Node v = malloc(sizeof(Node));
v->data=malloc(sizeof(NodeData));
memcpy(v->data, old->data,
           sizeof(NodeData));
free(old->data);
v->next = old->next;
v->next->prev = v;
v->prev = old->prev;
v->prev->next = v;
free(old);
```

BDW Collector

…slide in BDW and ignore calls to **free**.

What About Other Garbage Collectors?

- Compares **`malloc`** to GC, but only *conservative*, *non-copying* collectors
  - Can't reduce fragmentation, reorder objects, etc.
- But: faster *precise, copying* collectors
  - Incompatible with C/C++
  - Standard for Java…

Comparing Memory Managers

```
Node node = new Node();
node.data = new NodeData();
useNode(node);
node = null;
...
node = new Node();
...
node.data = new NodeData();
...
```

Lea Allocator

Adding **`malloc`**/**`free`** to Java:
not so easy…

Comparing Memory Managers

```
Node node = new Node();
node.data = new NodeData();
useNode(node);
node = null;
...
node = new Node();
...
node.data = new NodeData();
...
```

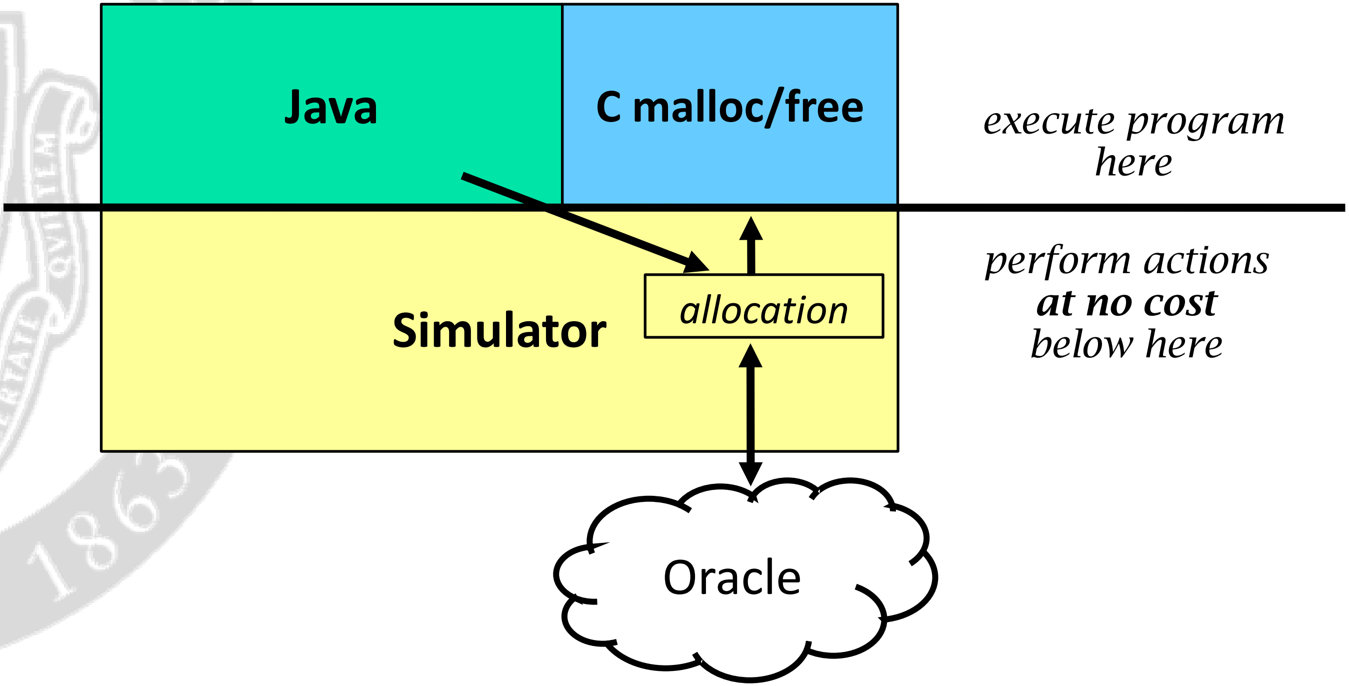
free(node)?

free(node.data)?

Lea Allocator

... need to insert **`frees`**, but where?

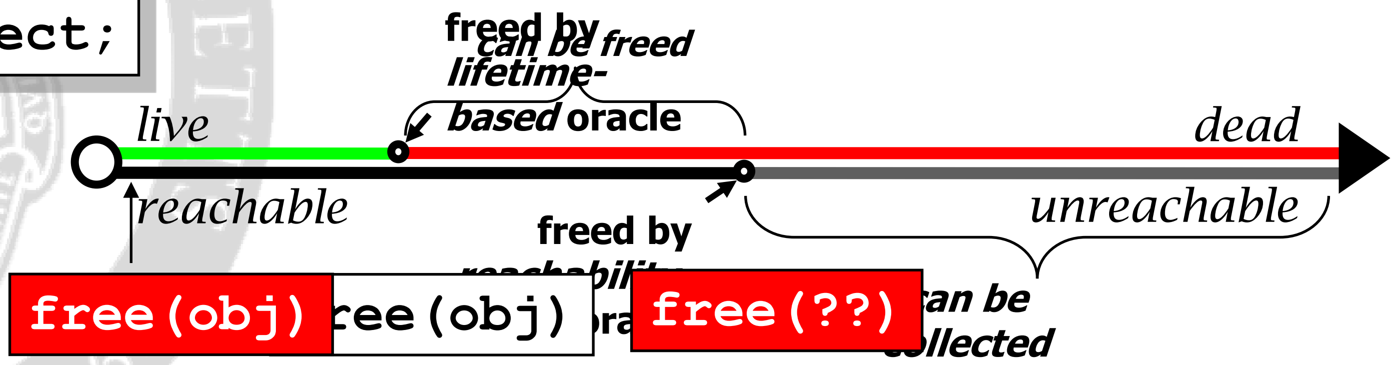
Oracular Memory Manager



- Consult oracle at each allocation
  - Oracle does not disrupt hardware state
  - Simulator invokes `free()`…

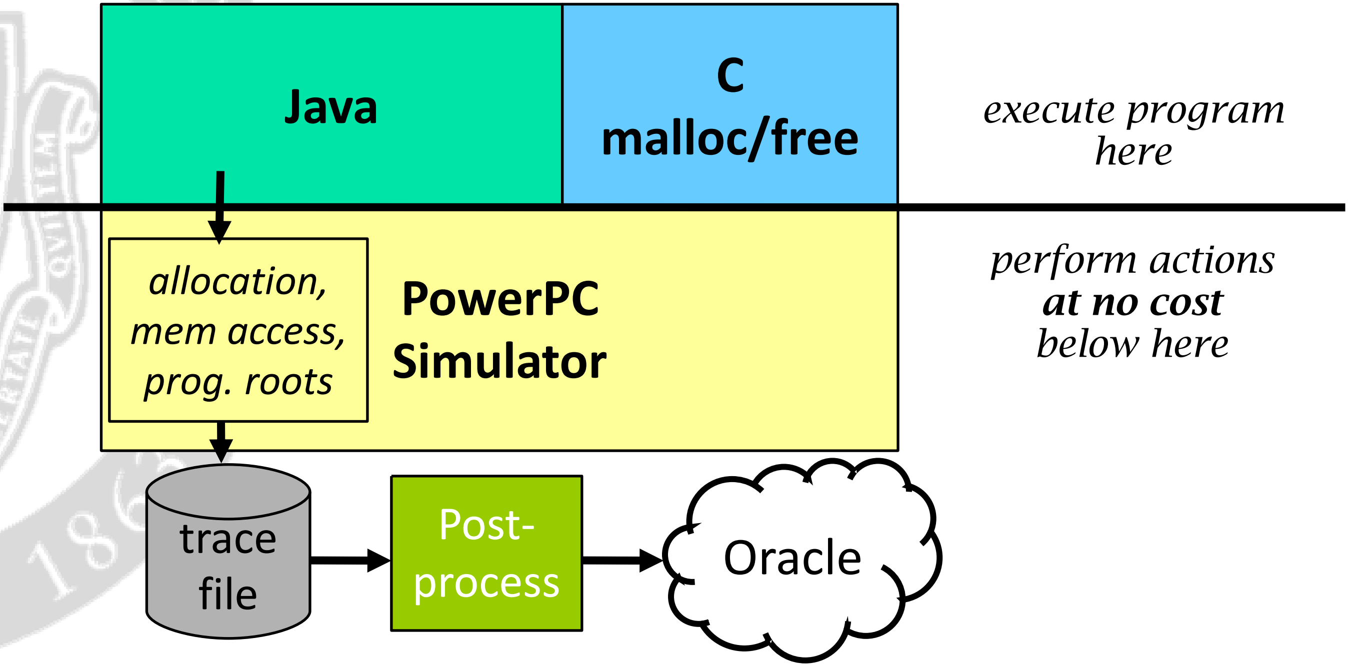
Object Lifetime & Oracle Placement

```
obj =
new Object;
```



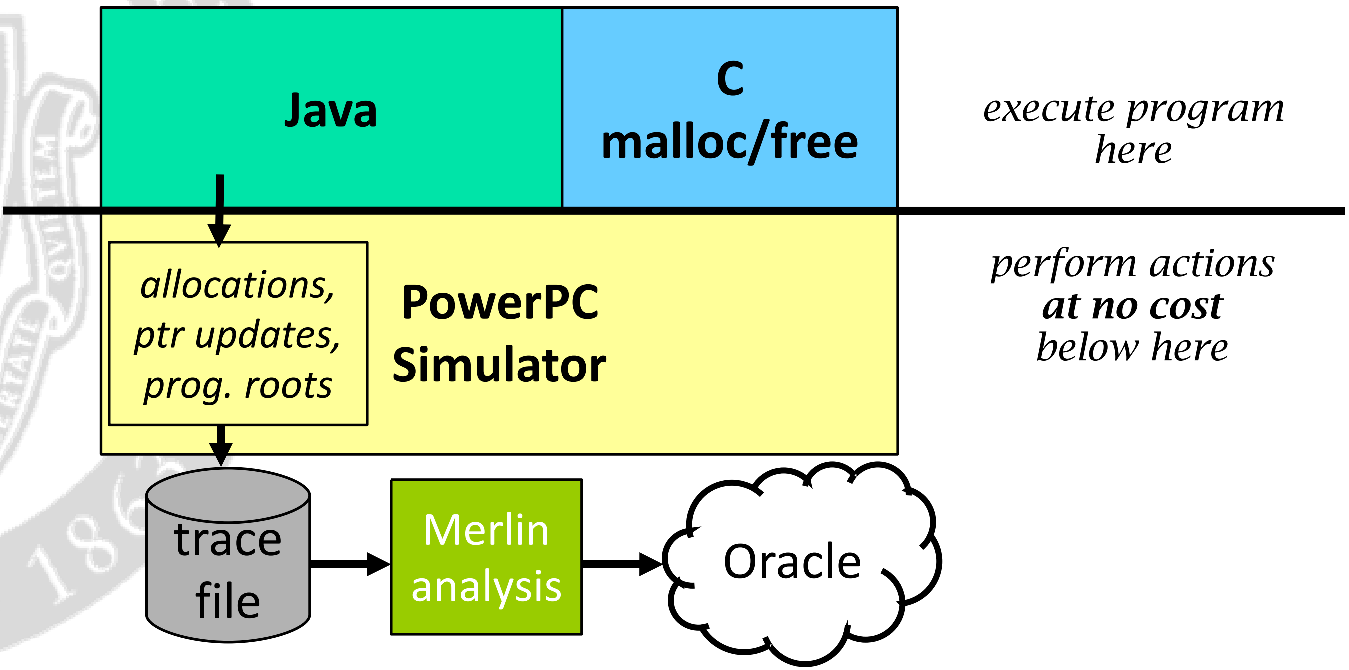
- Oracles bracket placement of **`frees`**
  - **Lifetime-based**: most aggressive
  - **Reachability-based**: most conservative

Liveness Oracle Generation



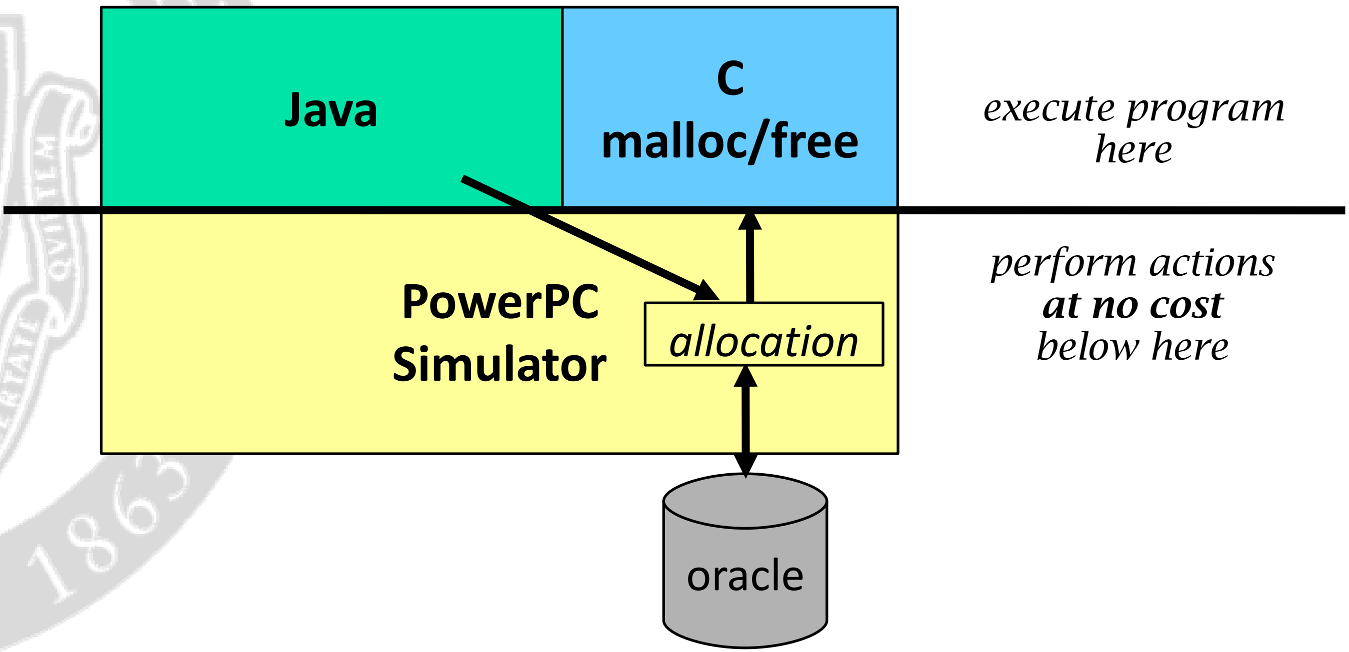
- **Liveness**: record allocs, memory accesses

Reachability Oracle Generation



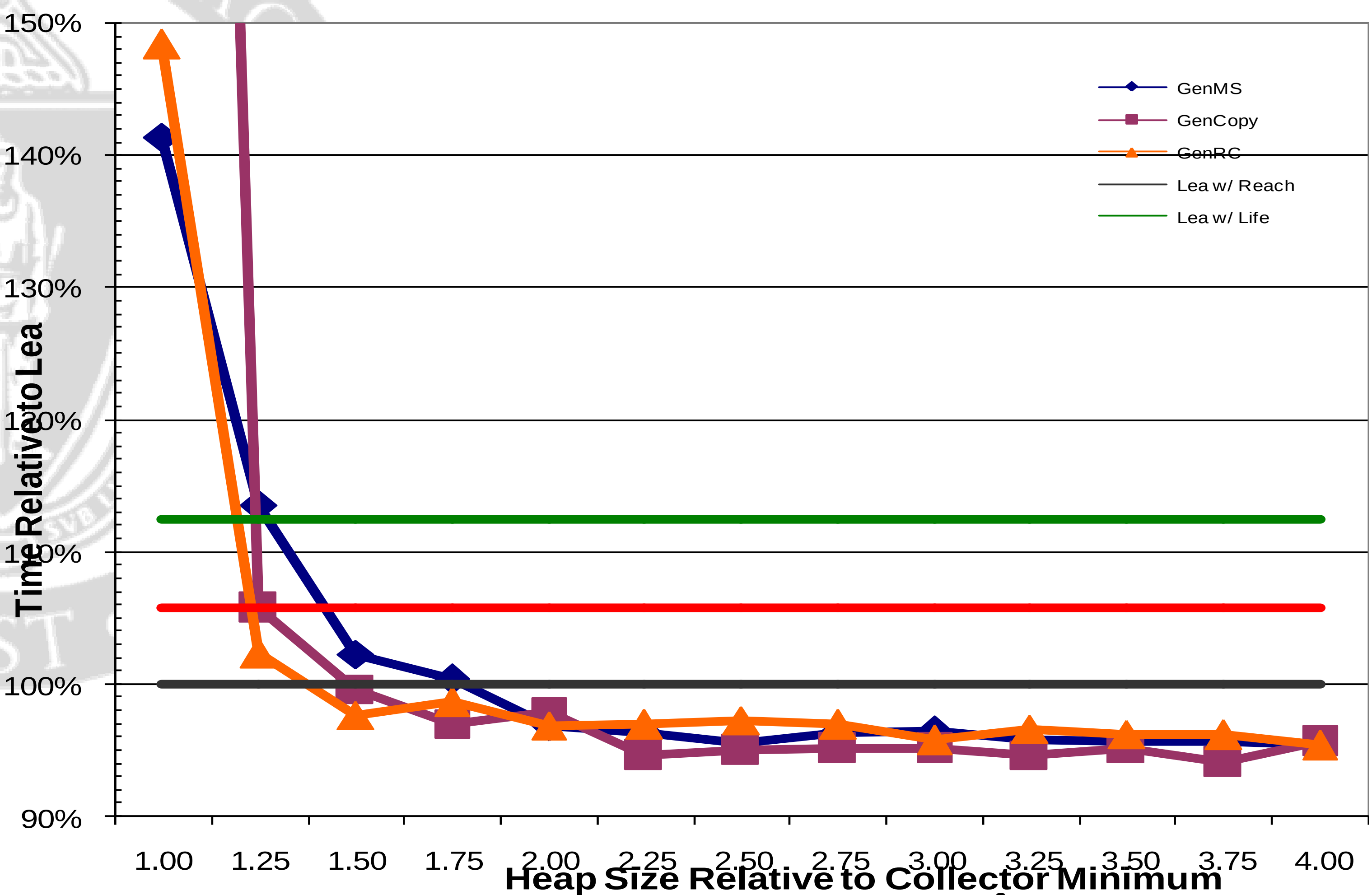
- **Reachability:**
  - Illegal instructions mark heap events (especially pointer assignments)

Oracular Memory Manager



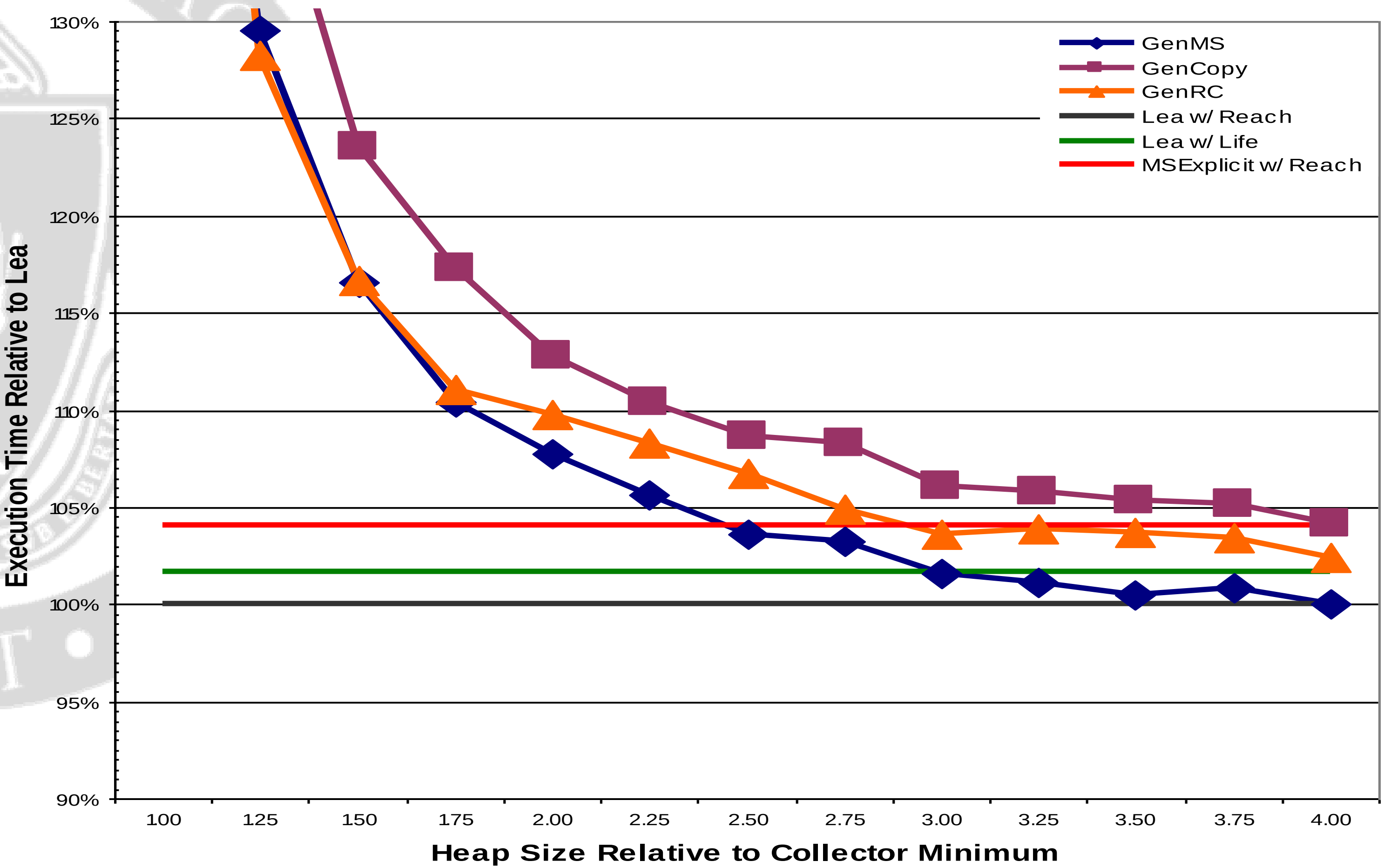
- Run & consult oracle before each allocation
  - When needed, modify instruction to call `free`
  - Extra costs (oracle access) hidden by simulator

Execution Time for pseudoJBB



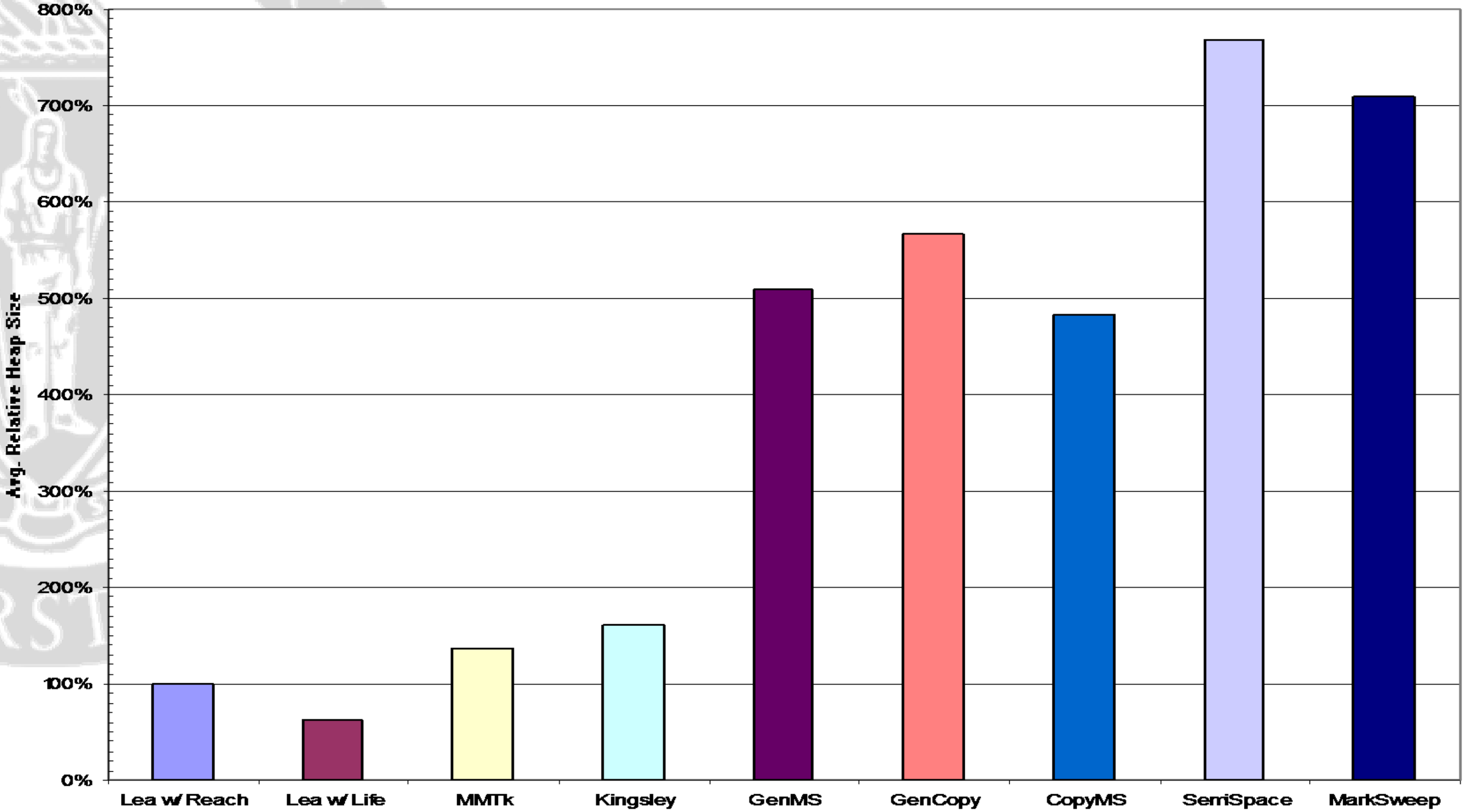
GC can be **faster** than **malloc/free**

Geo. Mean of Execution Time



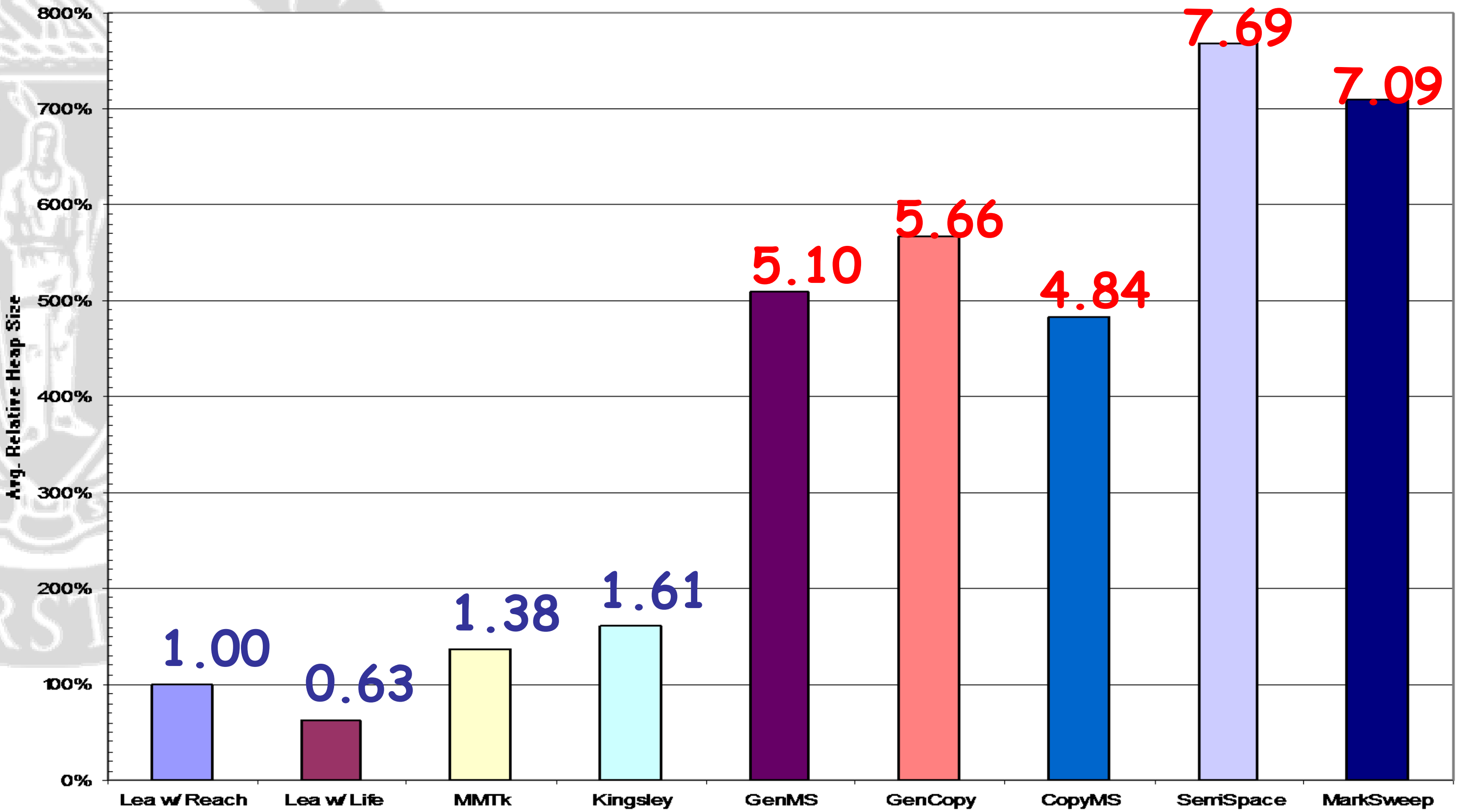
Trades space for time

Footprint at Quickest Run



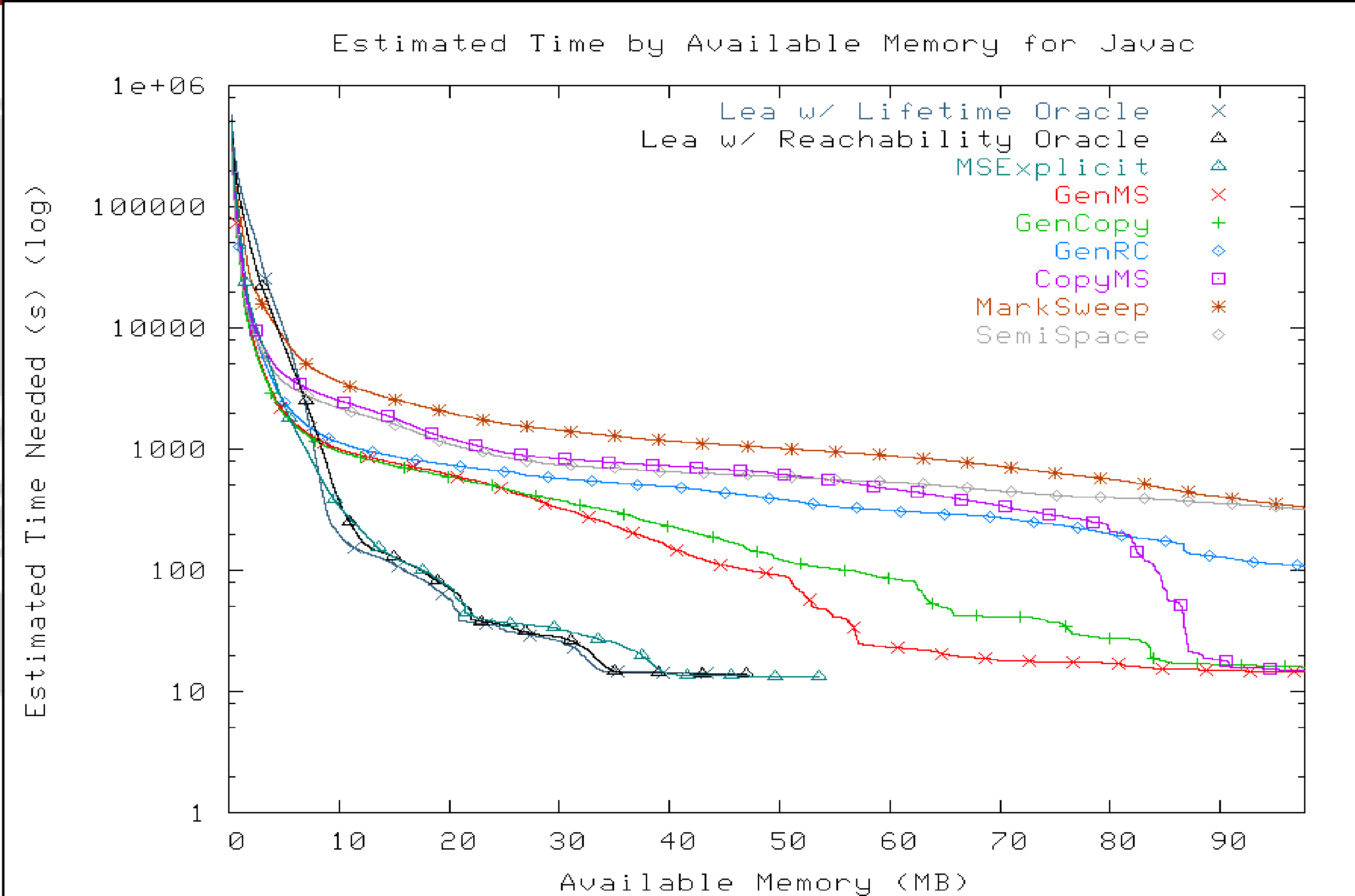
GC uses much more memory for speed

Footprint at Quickest Run



GC uses much more memory for speed

Javac Paging Performance



GC: **poor** paging performance

Summary of Results

- Best collector equals Lea's performance…
  - Up to 10% faster on some benchmarks
- … but uses more memory
  - Quickest runs require 5x or more memory
  - GenMS at least doubles mean footprint

When to Use Garbage Collection

- Garbage collection fine **if**
  - system has more than 3x needed RAM
  - and no competition with other processes
  - **or** avoiding bugs / security more important
- Not so good:
  - Limited RAM
  - Competition for physical memory
  - Depends on RAM for performance
    - In-memory database
    - Search engines, etc.

The End