RTGC

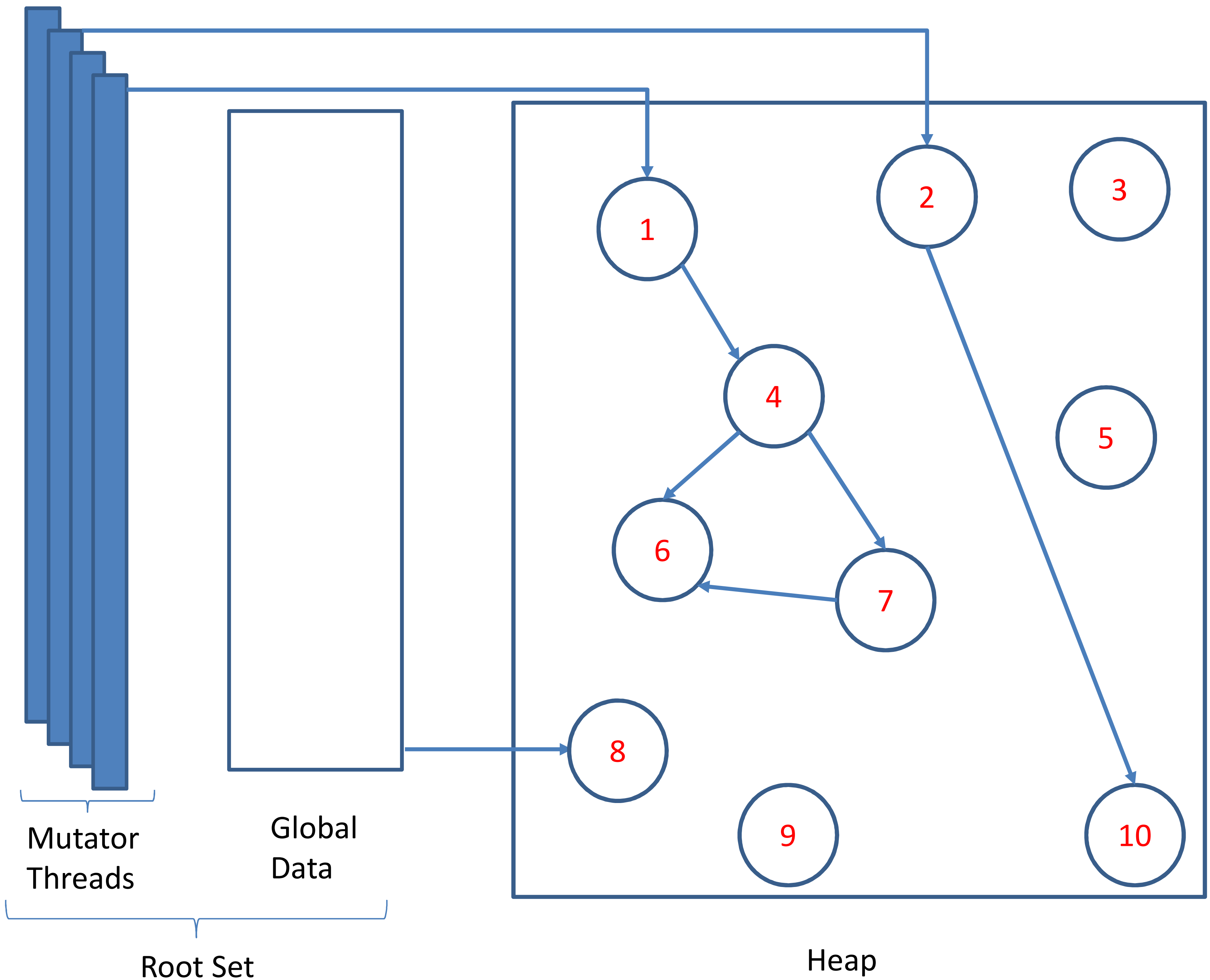
Real-Time
Multi-Core
Continuously Concurrent
Garbage Collection

Two Talks in One

- Part 1: Atomic GC
- Part 2: Real-time Concurrent GC

A Visual Representation of Memory

- We need a visual representation of memory to discuss garbage collection.
- That won’t fit on this slide, so it’s on the next one.



1
2
3
4
5
6
7
8
9
10
Mutator Threads
Global Data
Root Set
Heap

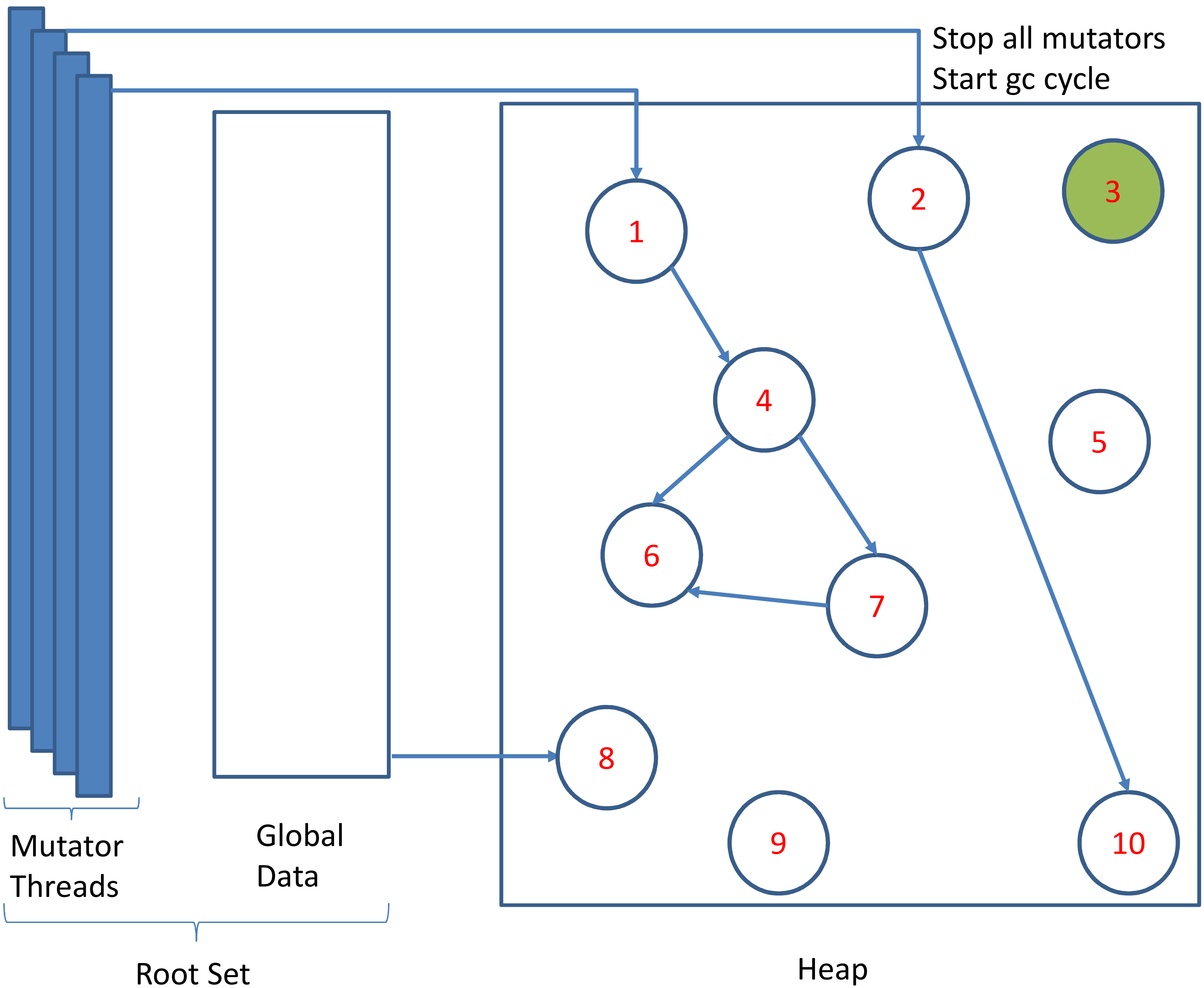
Part 1:
A Simple Atomic Tracing Collector

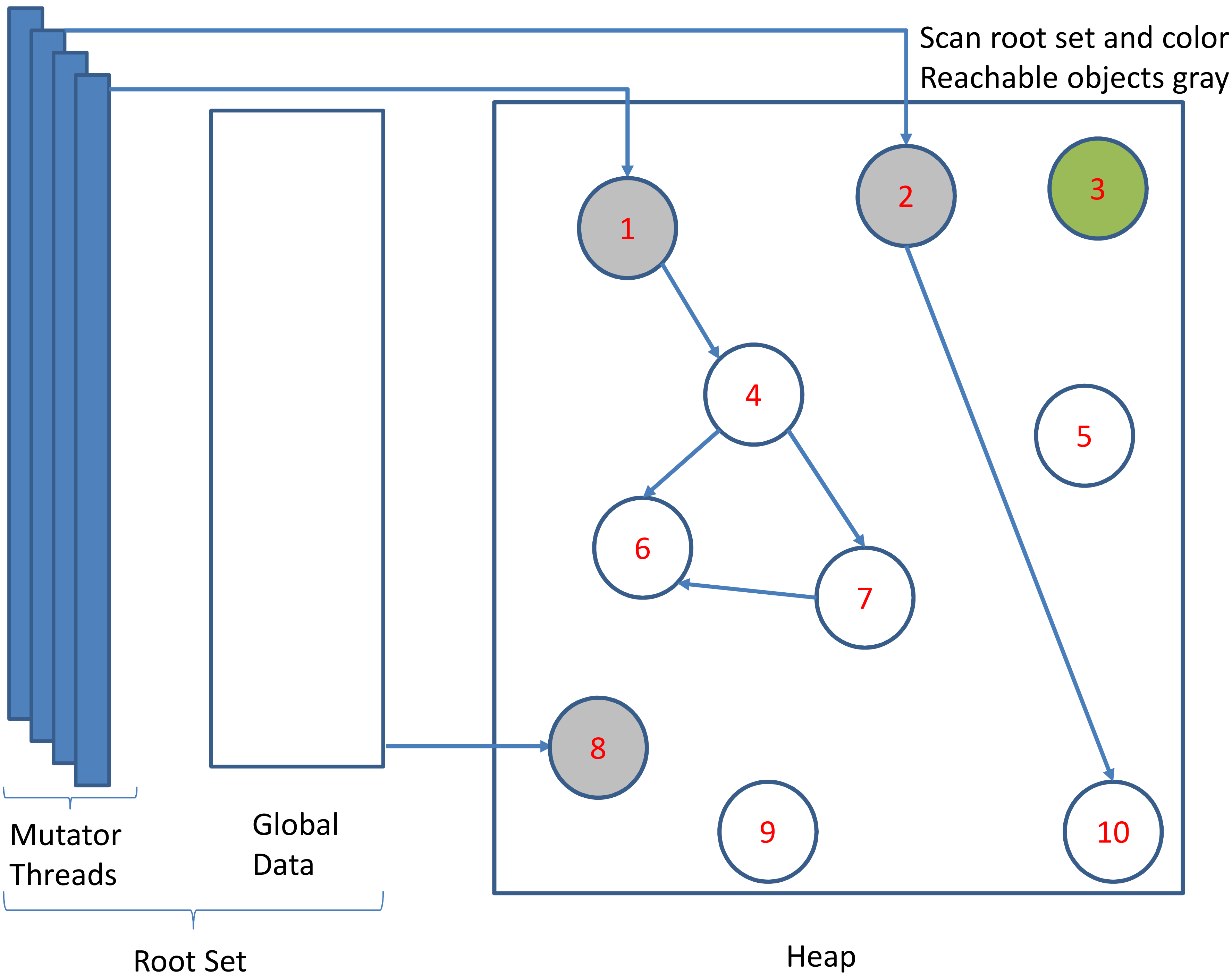
- “Atomic” == “Stop the world” - The gc stops the execution of every mutator thread. Mutators and the gc cannot run at the same time.
- In a “gc cycle” we recursively trace the graph of all reachable objects, starting from the “root set” of pointers.
- Reclaim/collect all “garbage” objects that weren’t reachable by tracing so they may be allocated as new objects.
- No way for mutators to alter the object pointer graph while the gc works because they are all stopped.
- Resume all mutator threads after a gc cycle has completed.
- Your program is either “mutating” or “collecting”, but not both at the same time.

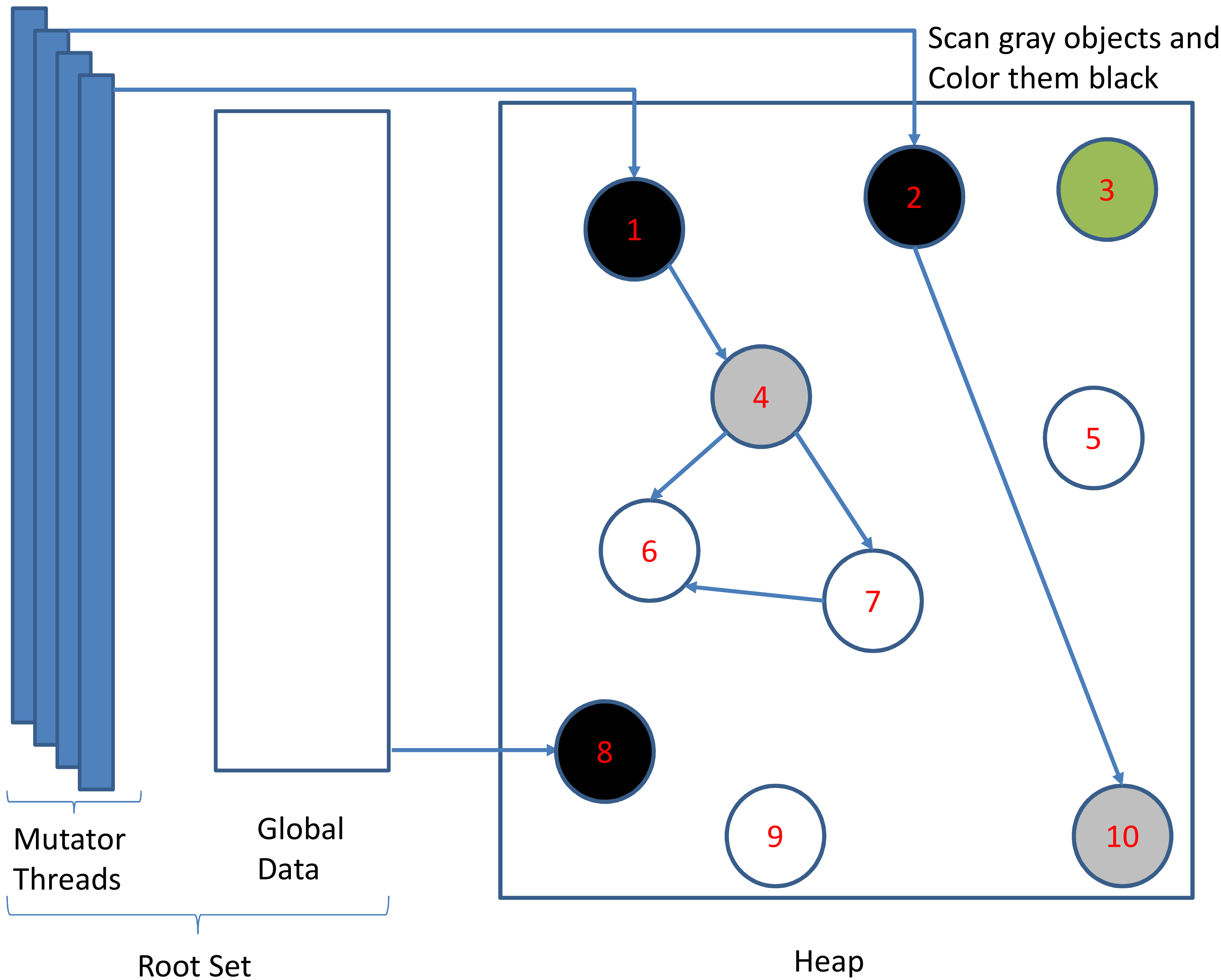
Quad-Color marking

- Assign each object a color:
  - White - All allocated objects are white at the beginning of a gc cycle. This means they may be reachable, or they may be garbage. We don't know yet.
  - Gray - A gray object has been found to be reachable. However, we don't know anything about other objects it may point at recursively.
  - Black - A black object is not only reachable, but we have recursively scanned it for pointers to any other objects in the graph.
  - Green – free objects that can be allocated as new objects.
- All reachable objects change from white to gray to black in a gc cycle.
- Any objects that remain white at the end of a gc cycle are garbage and are changed to green.

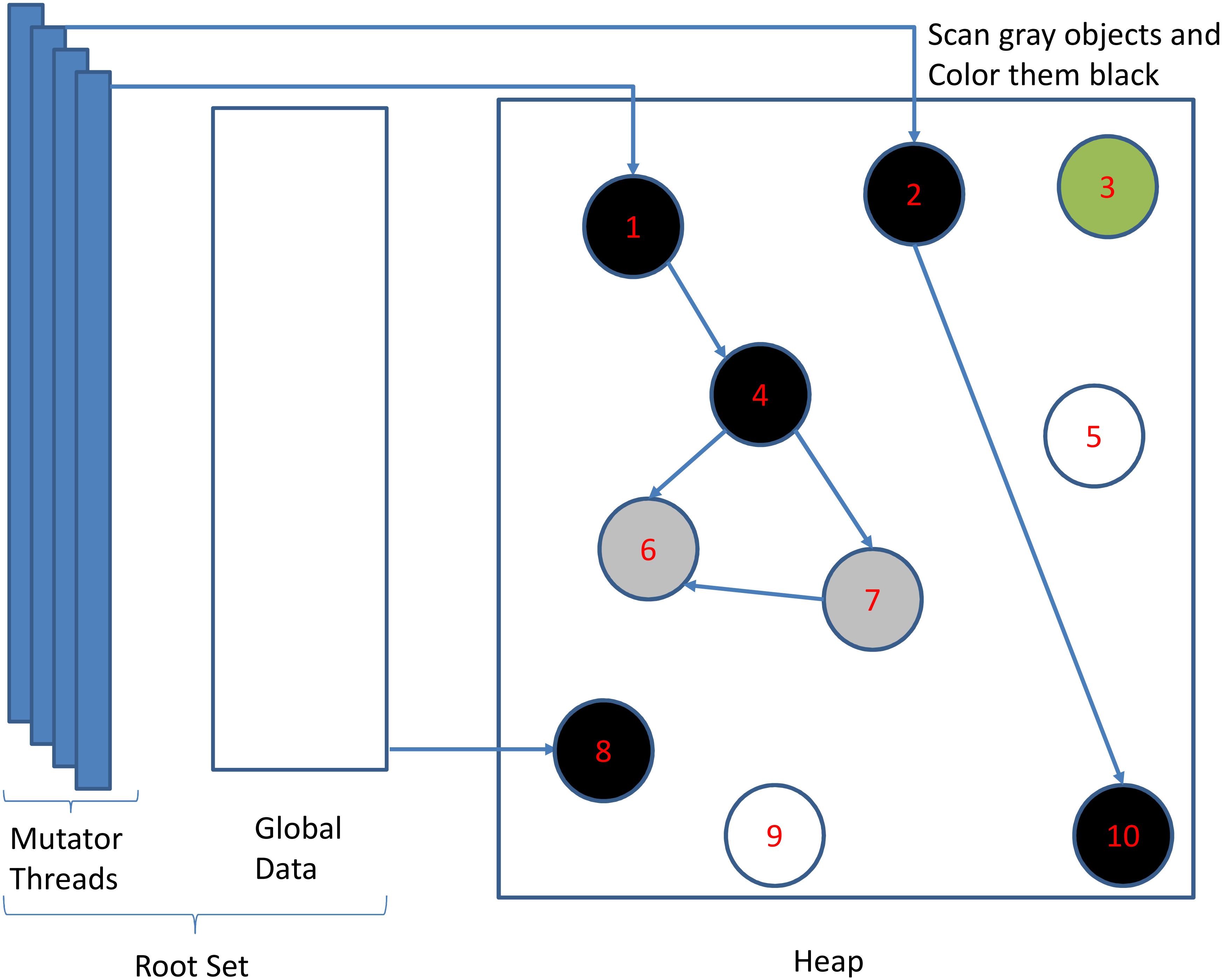
Lets look at a visual example of this process.


Stop all mutators
Start gc cycle
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

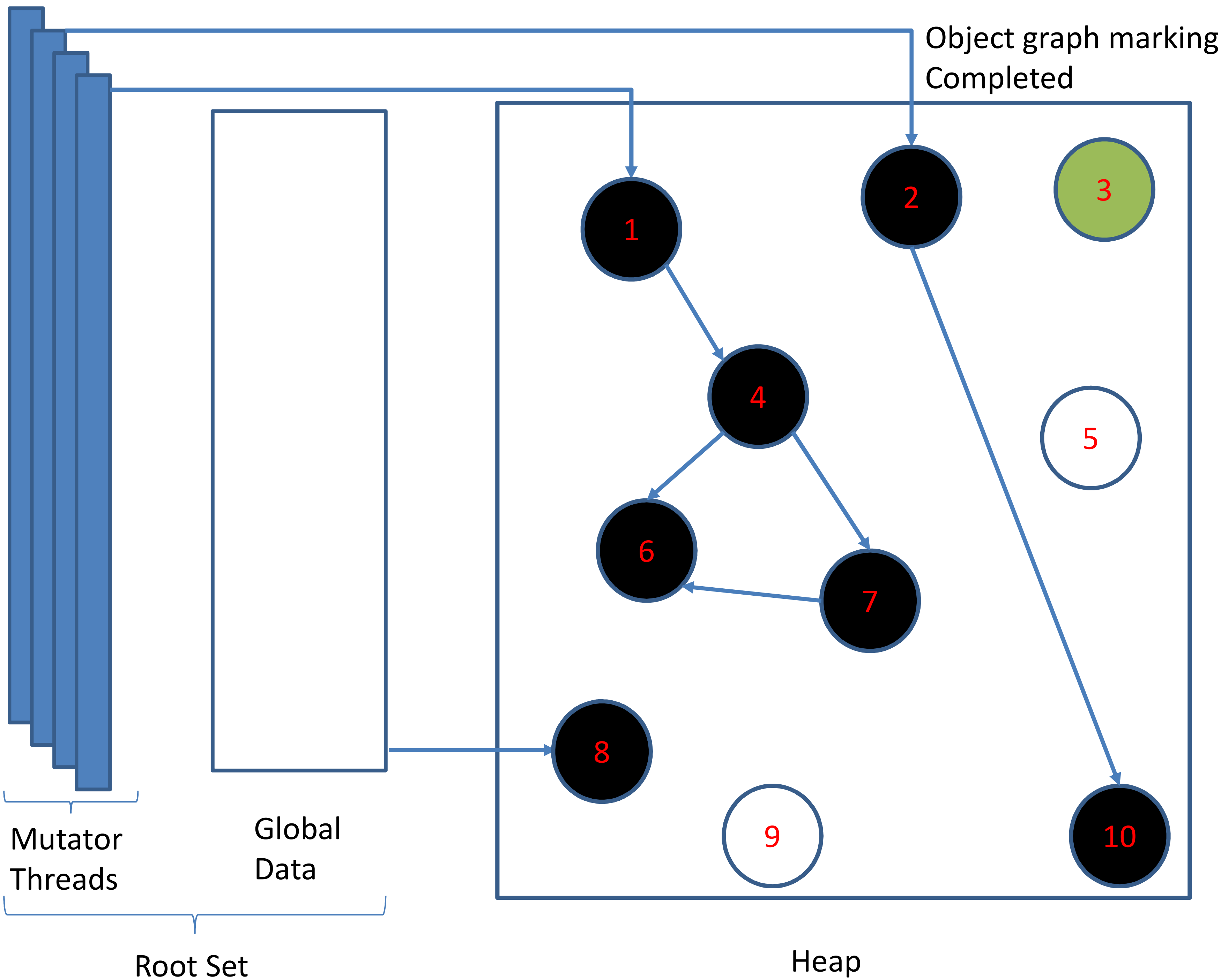

Scan root set and color
Reachable objects gray
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

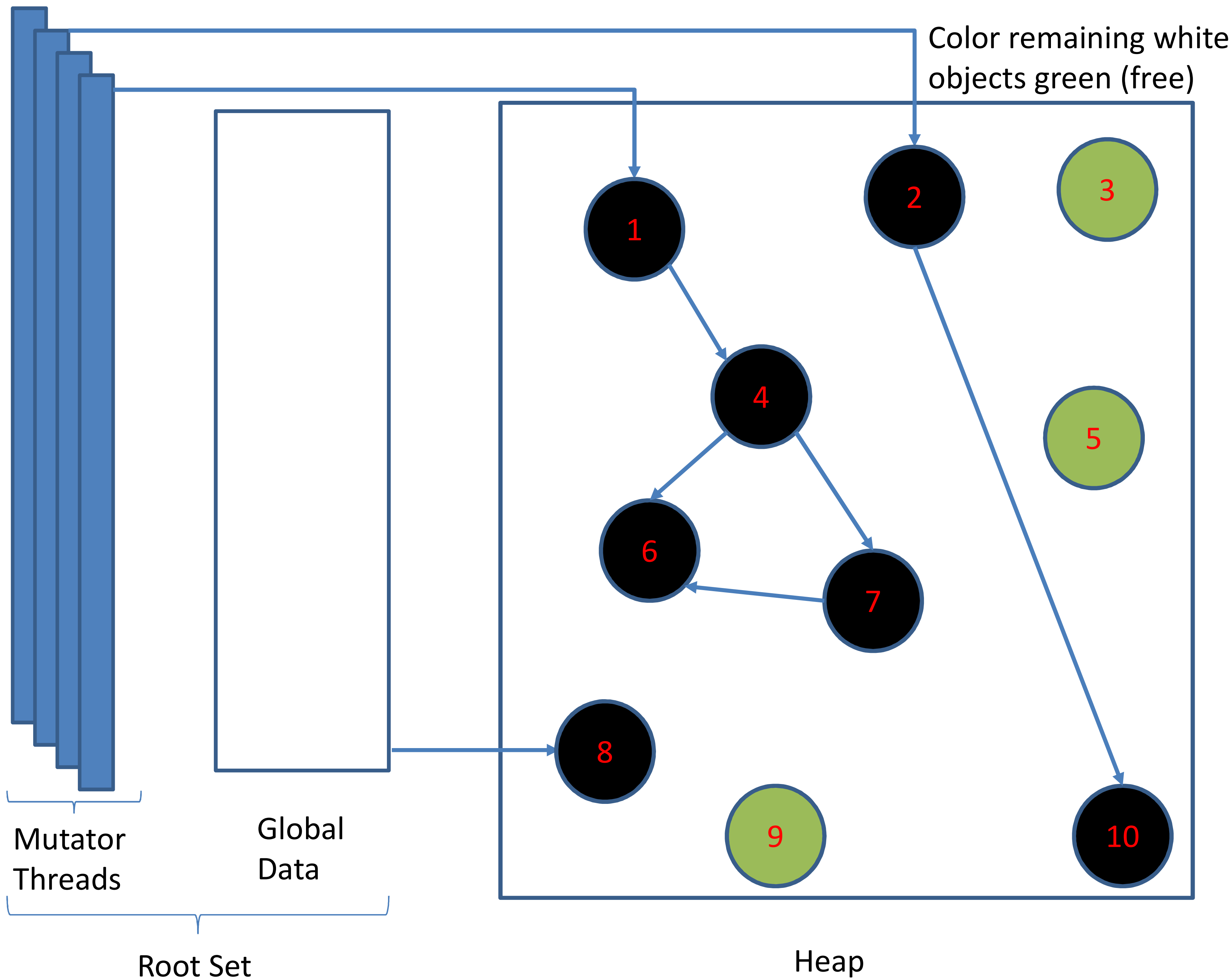


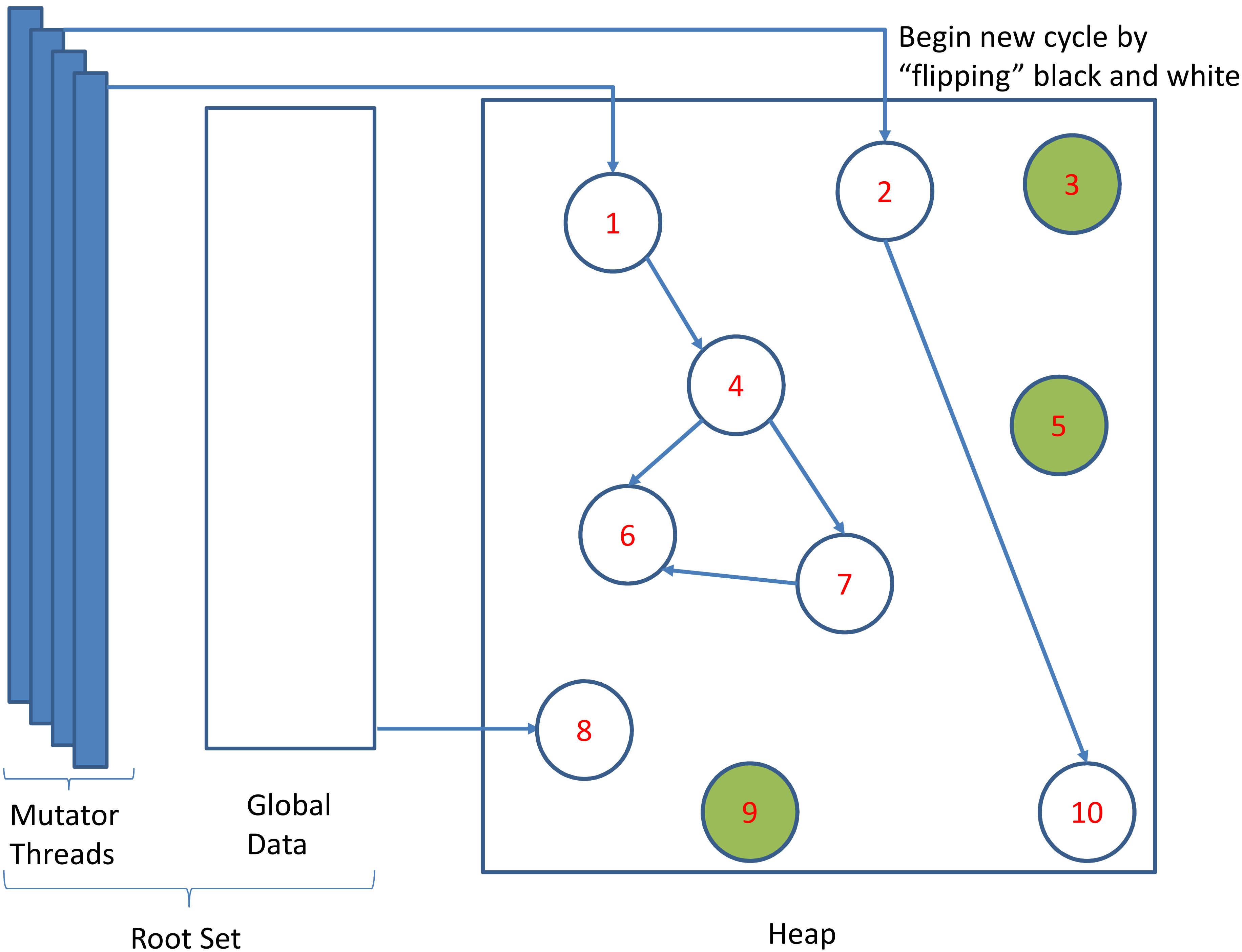
Scan gray objects and
Color them black
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap



Scan gray objects and
Color them black
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

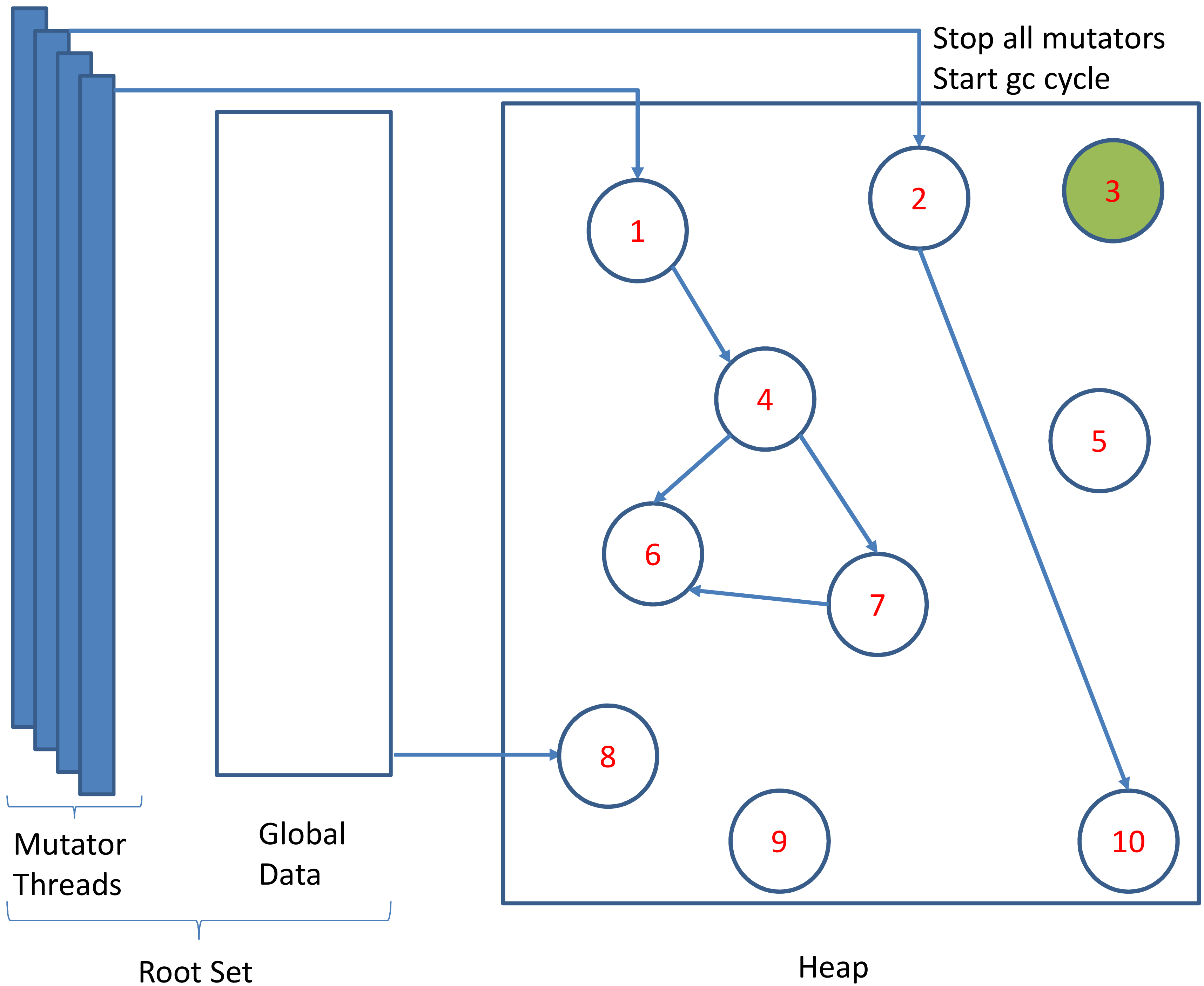

Object graph marking
Completed
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap


Color remaining white objects green (free)
1
2
3
4
5
6
7
8
9
10
Mutator Threads
Global Data
Root Set
Heap

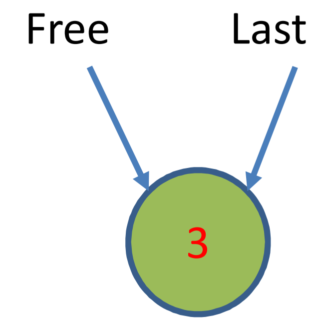

Begin new cycle by
“flipping” black and white
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

One Problem

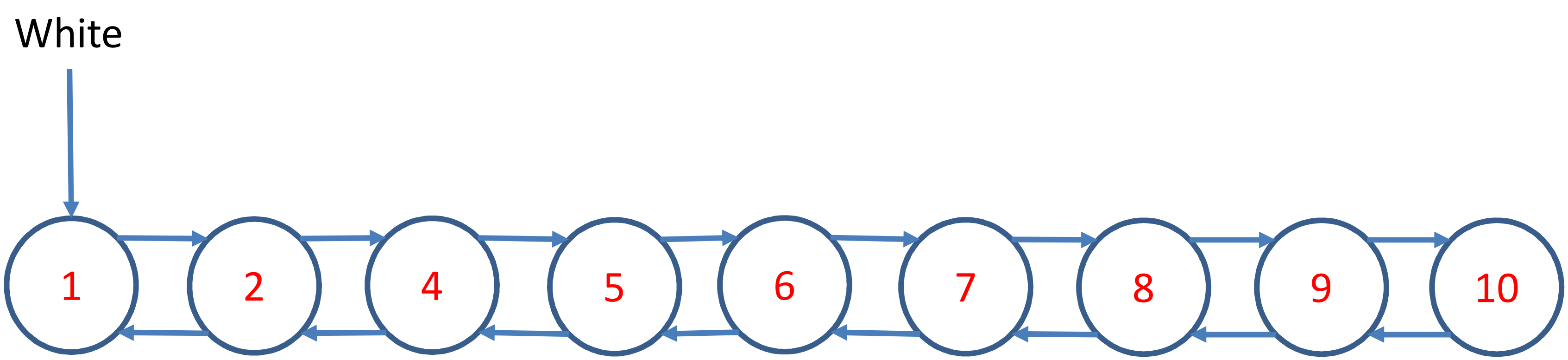
- We repeatedly had to look at every single object to determine which ones were gray.
- Let's fix this by treating object colors as sets that are redundantly represented by doubly-linked lists.

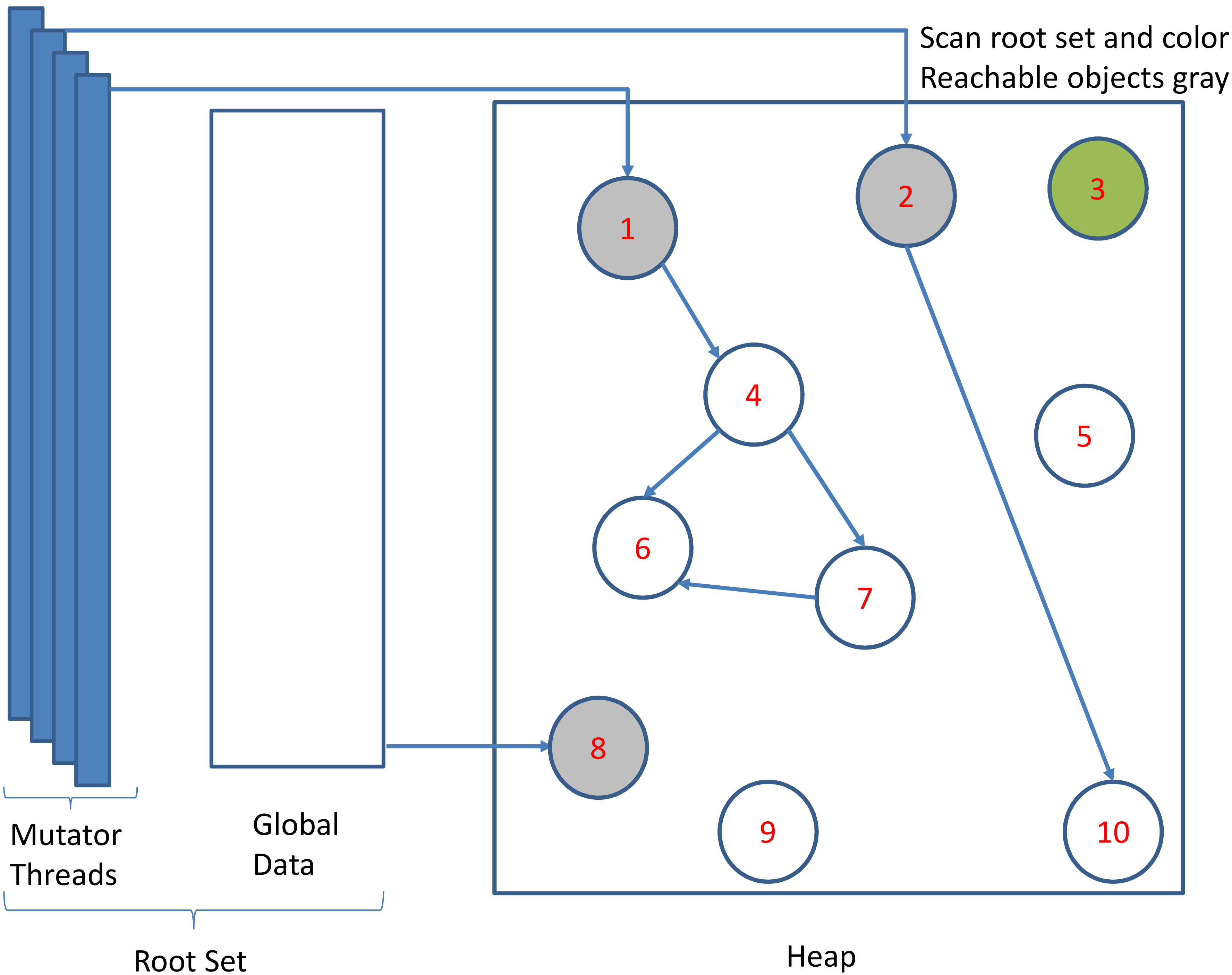

Stop all mutators
Start gc cycle
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

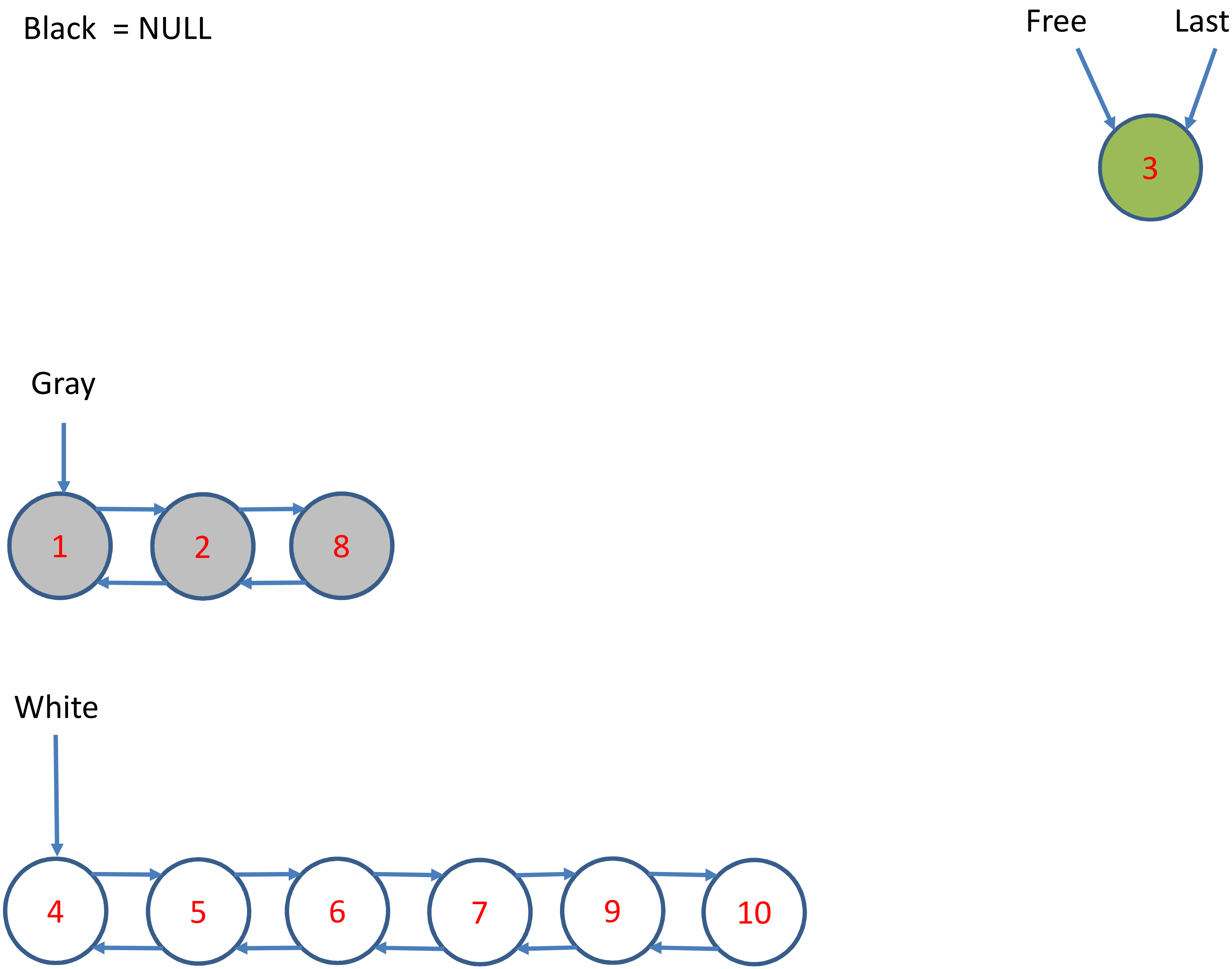
Black = NULL

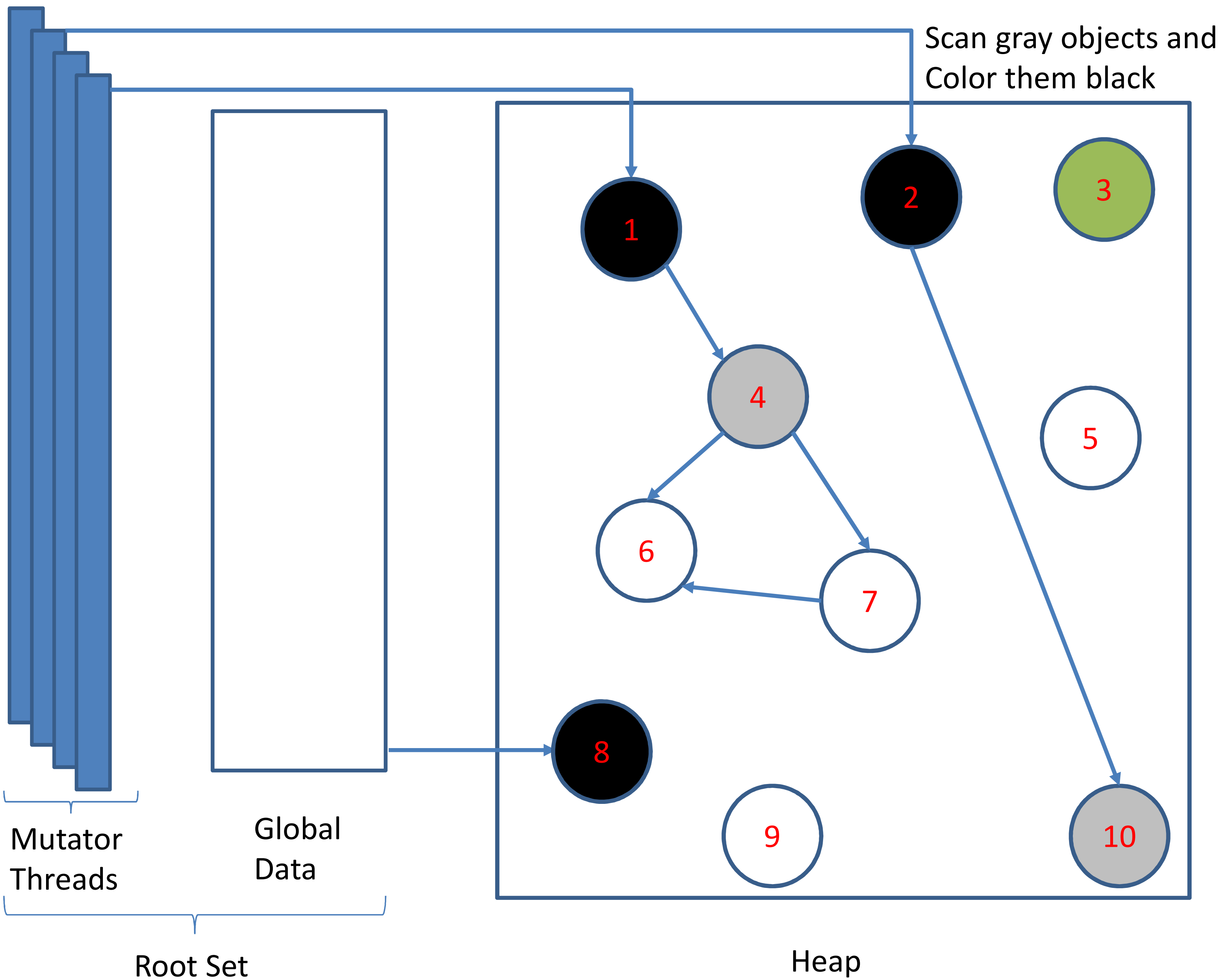


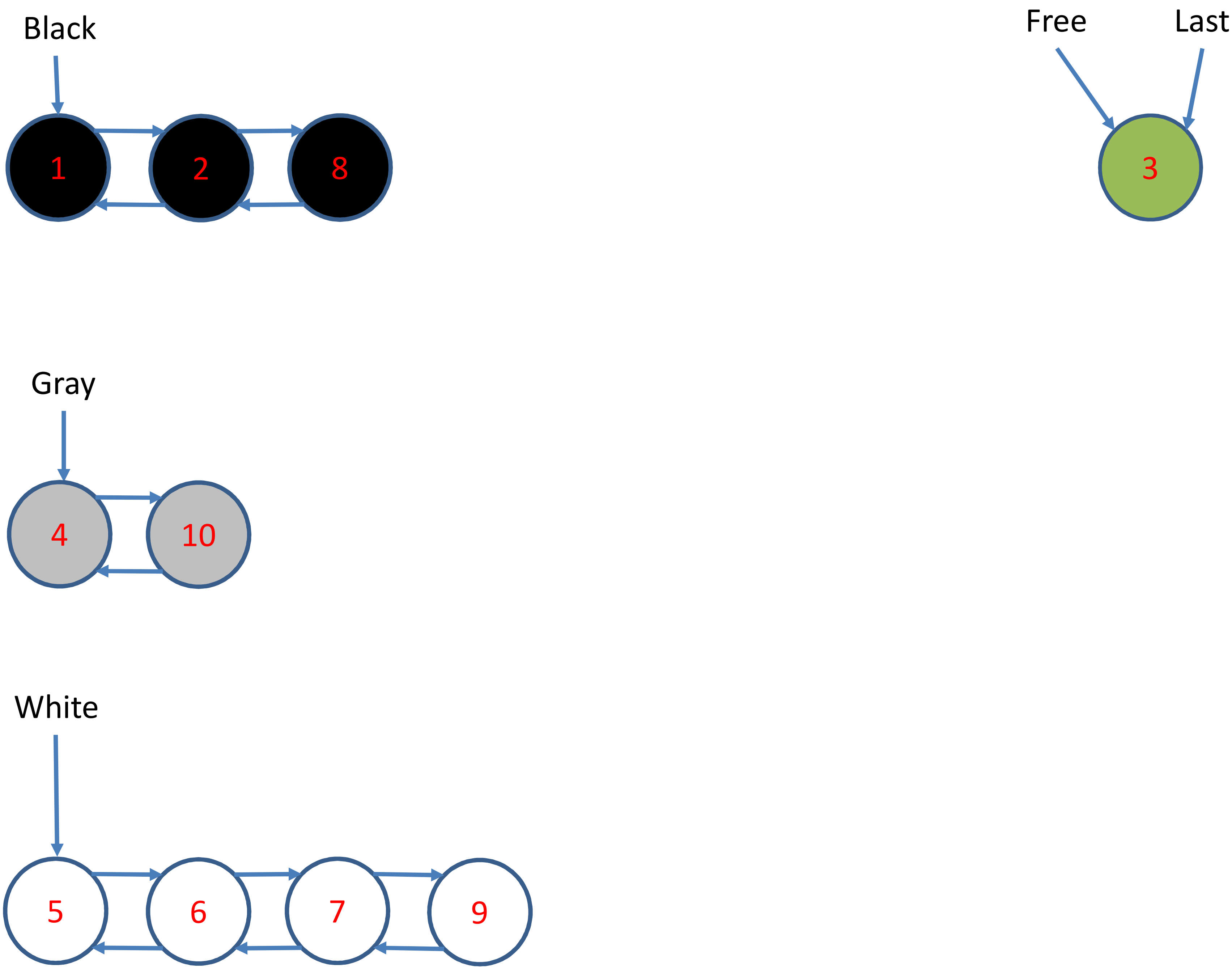
Gray = NULL

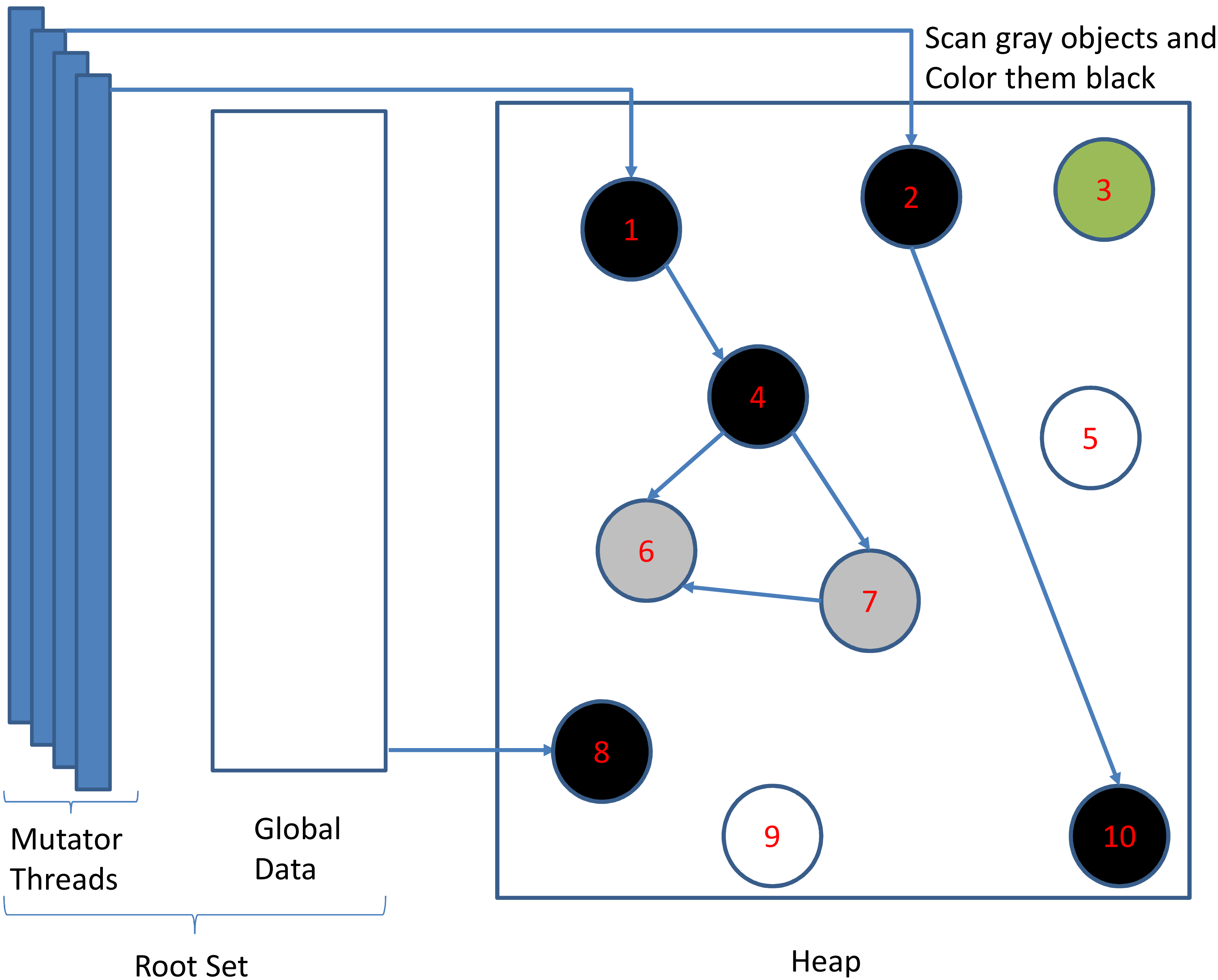


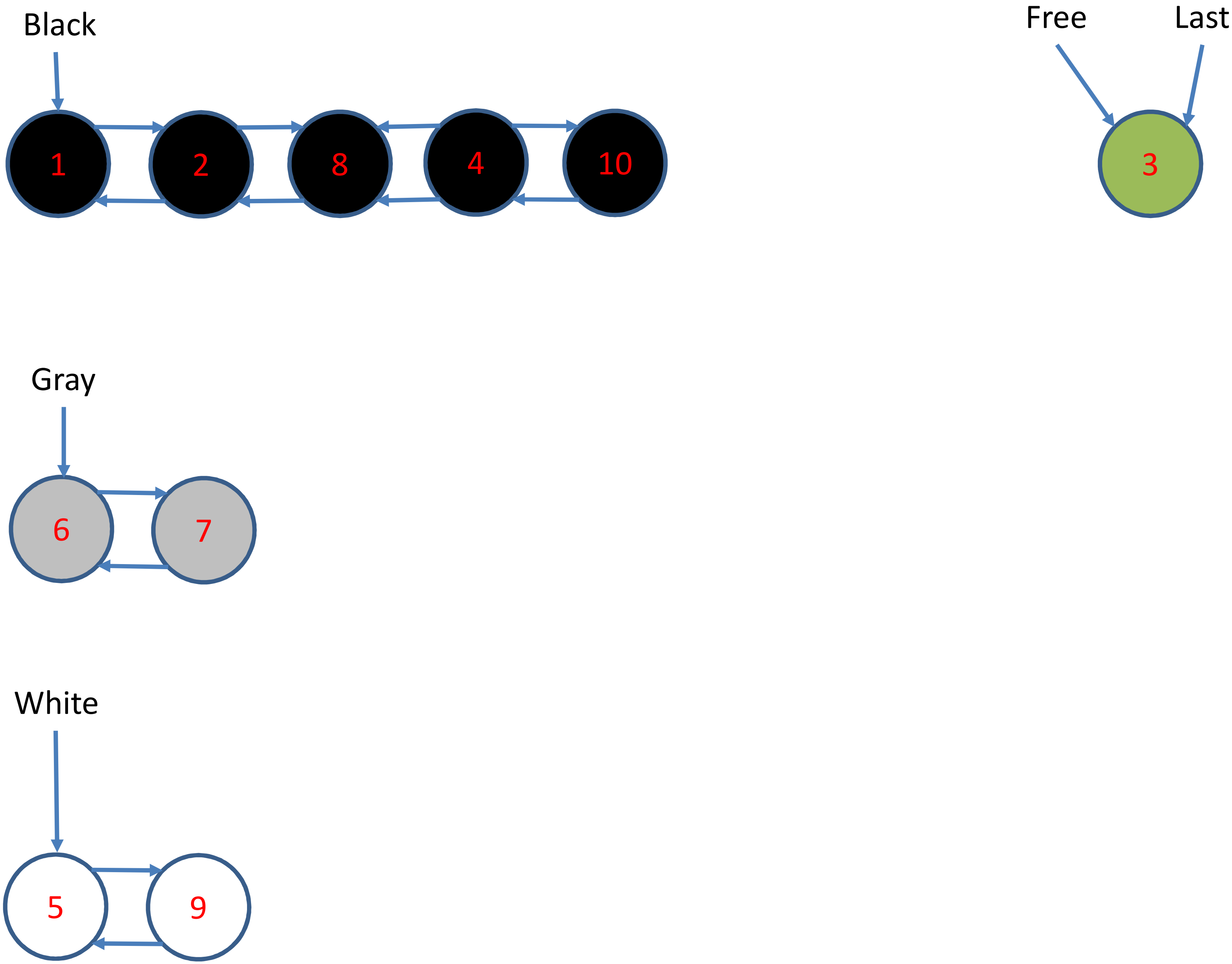

Scan root set and color
Reachable objects gray
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

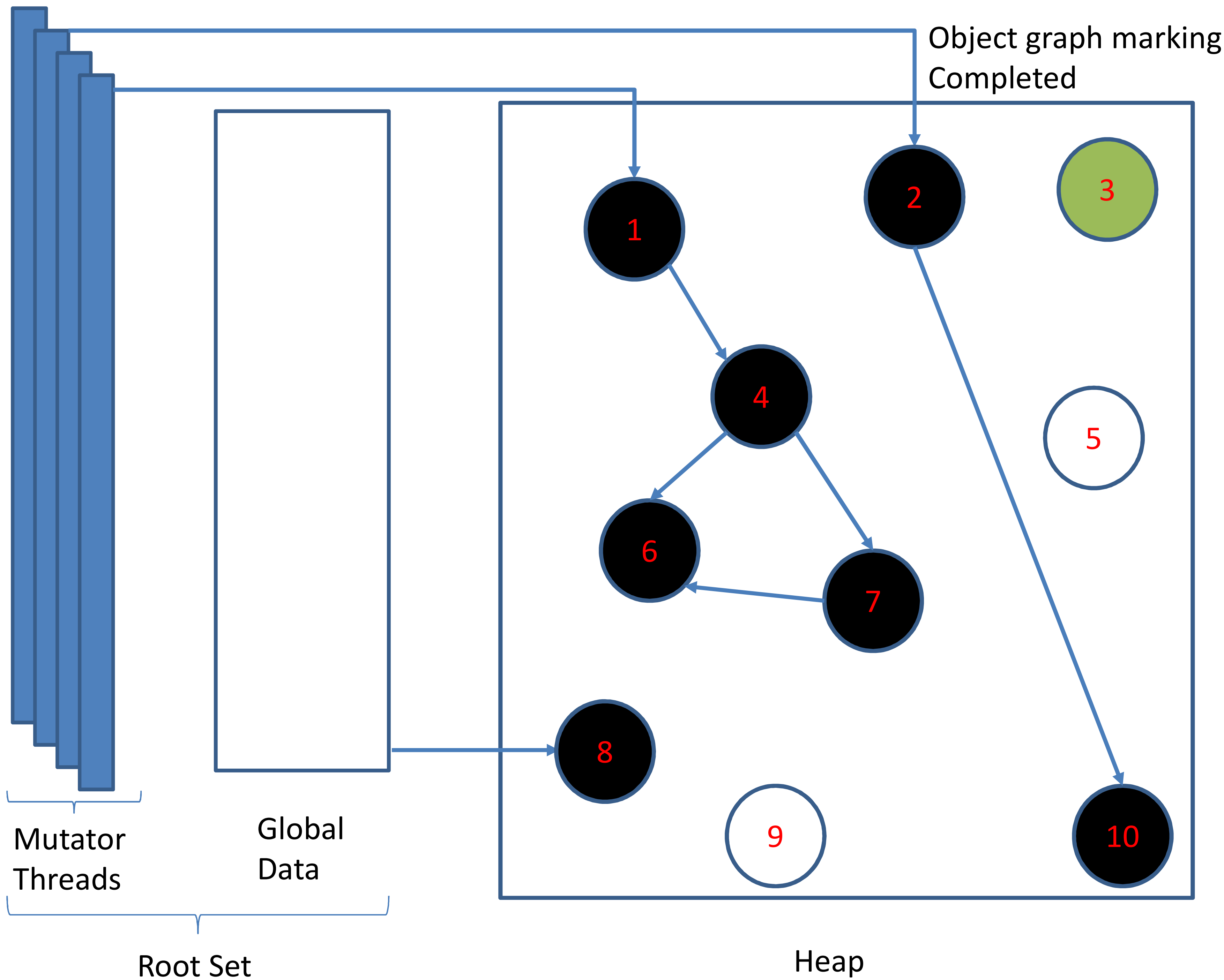

Black = NULL
Free
Last
3
Gray
1
2
8
White
4
5
6
7
9
10

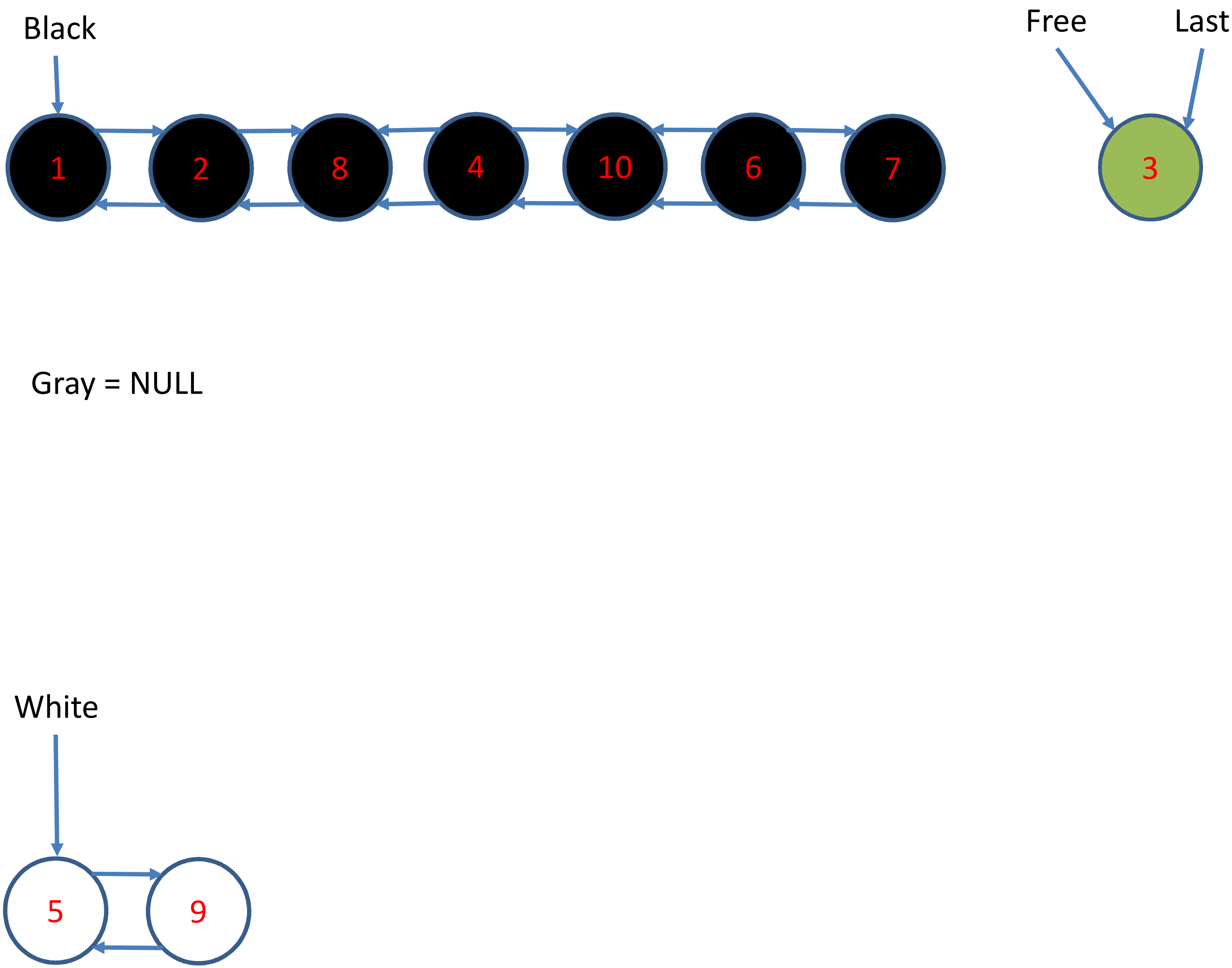

Scan gray objects and
Color them black
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

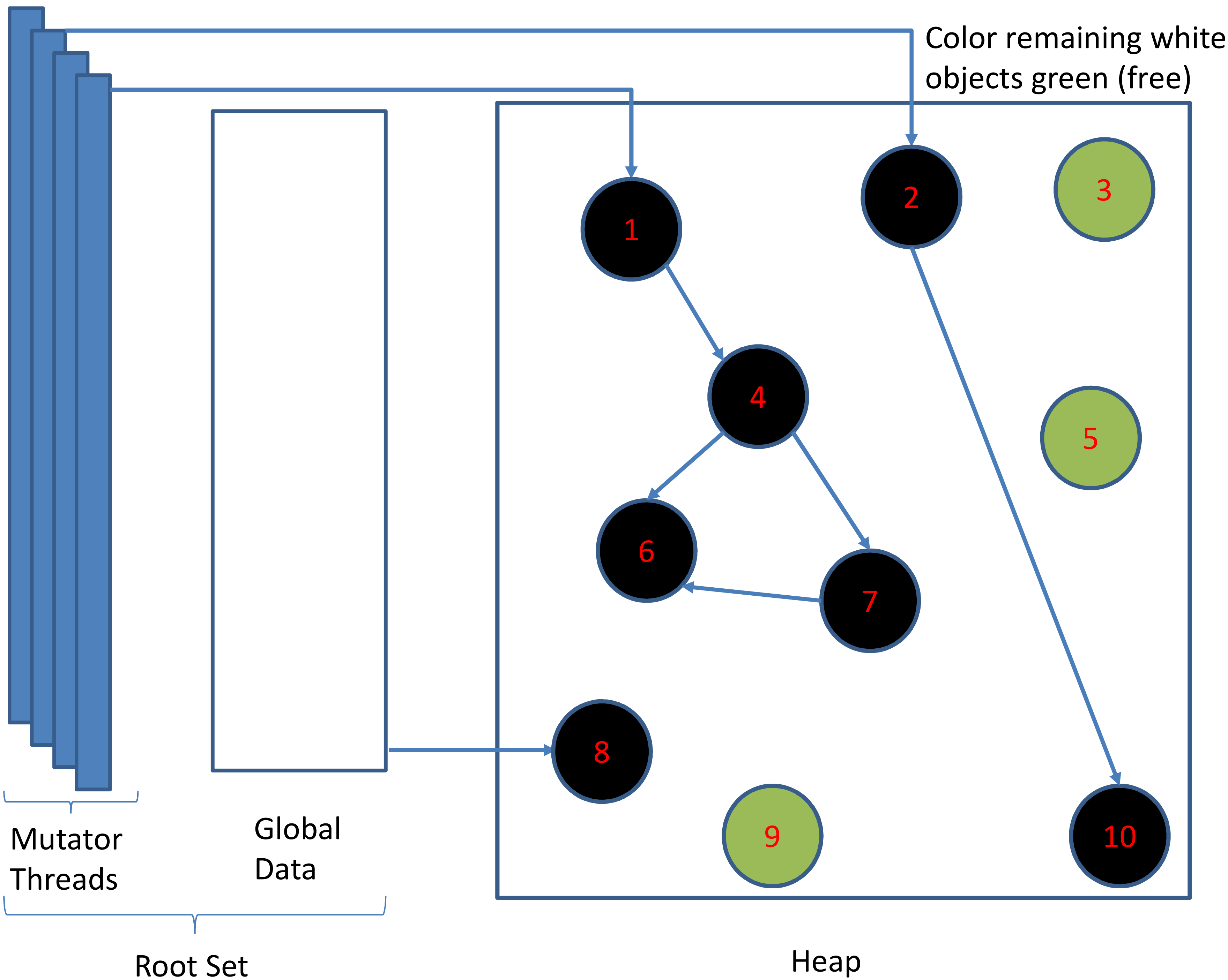

Black
1
2
8
Free
Last
3
Gray
4
10
White
5
6
7
9

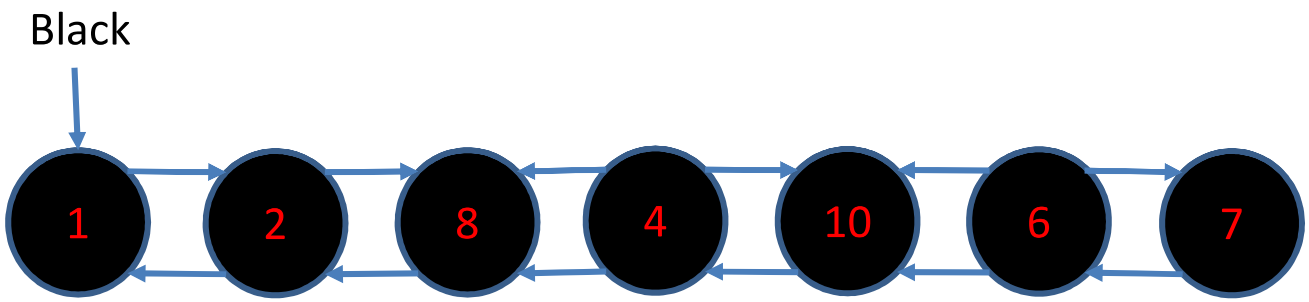

Scan gray objects and
Color them black
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap


Black
1
2
8
4
10
Free
Last
3
Gray
6
7
White
5
9

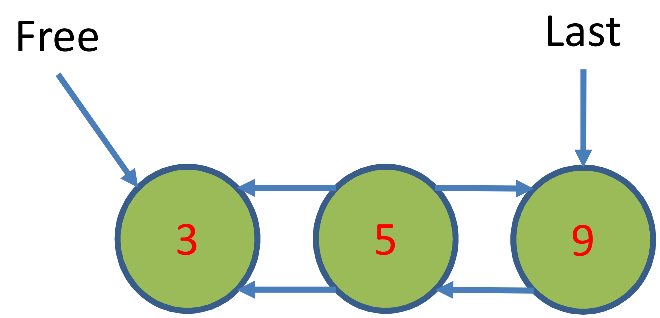

Object graph marking
Completed
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap


Black
1
2
8
4
10
6
7
Free
Last
3
Gray = NULL
White
5
9

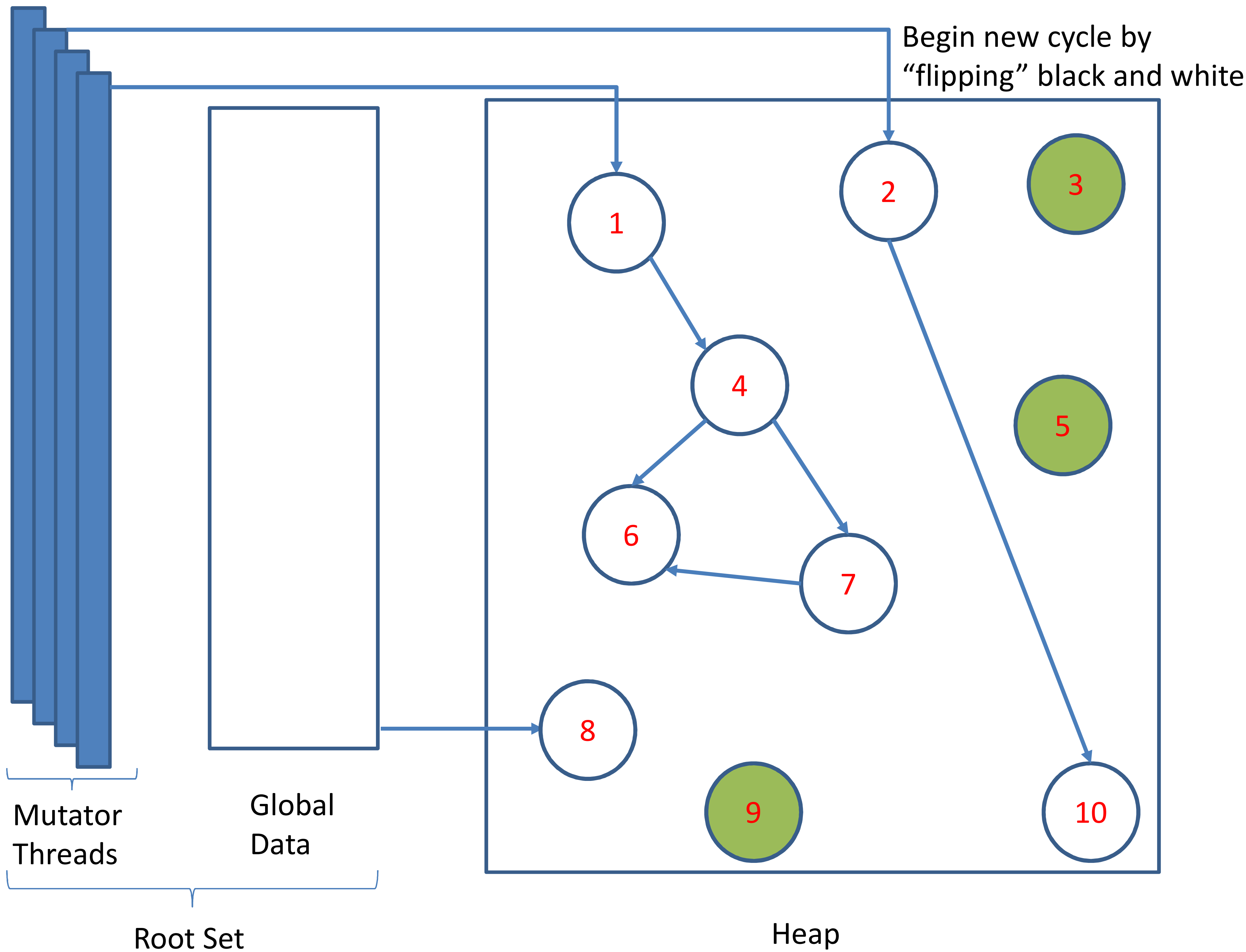

Color remaining white
objects green (free)
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

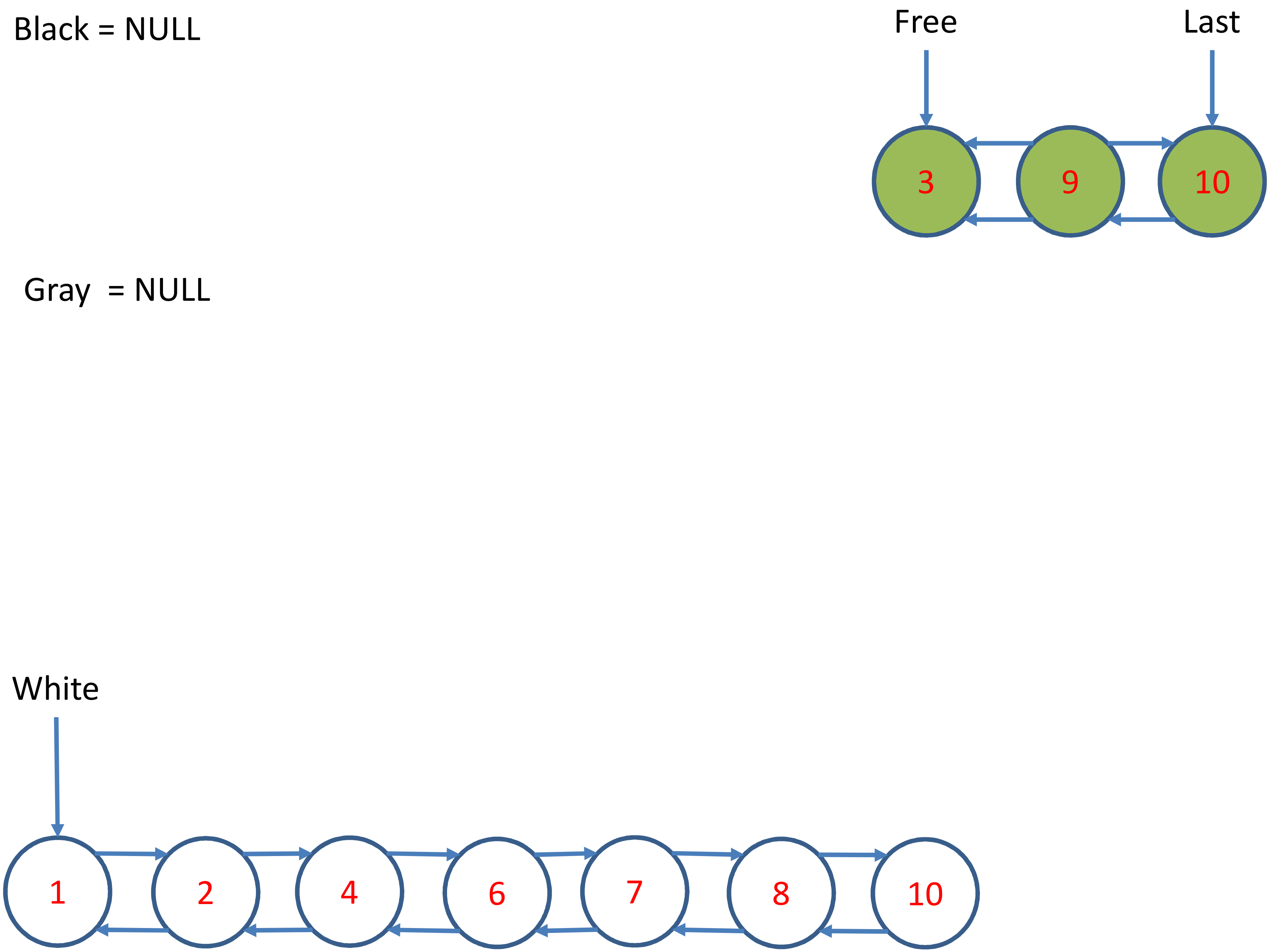


Gray = NULL



White = NULL


Begin new cycle by
“flipping” black and white
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap

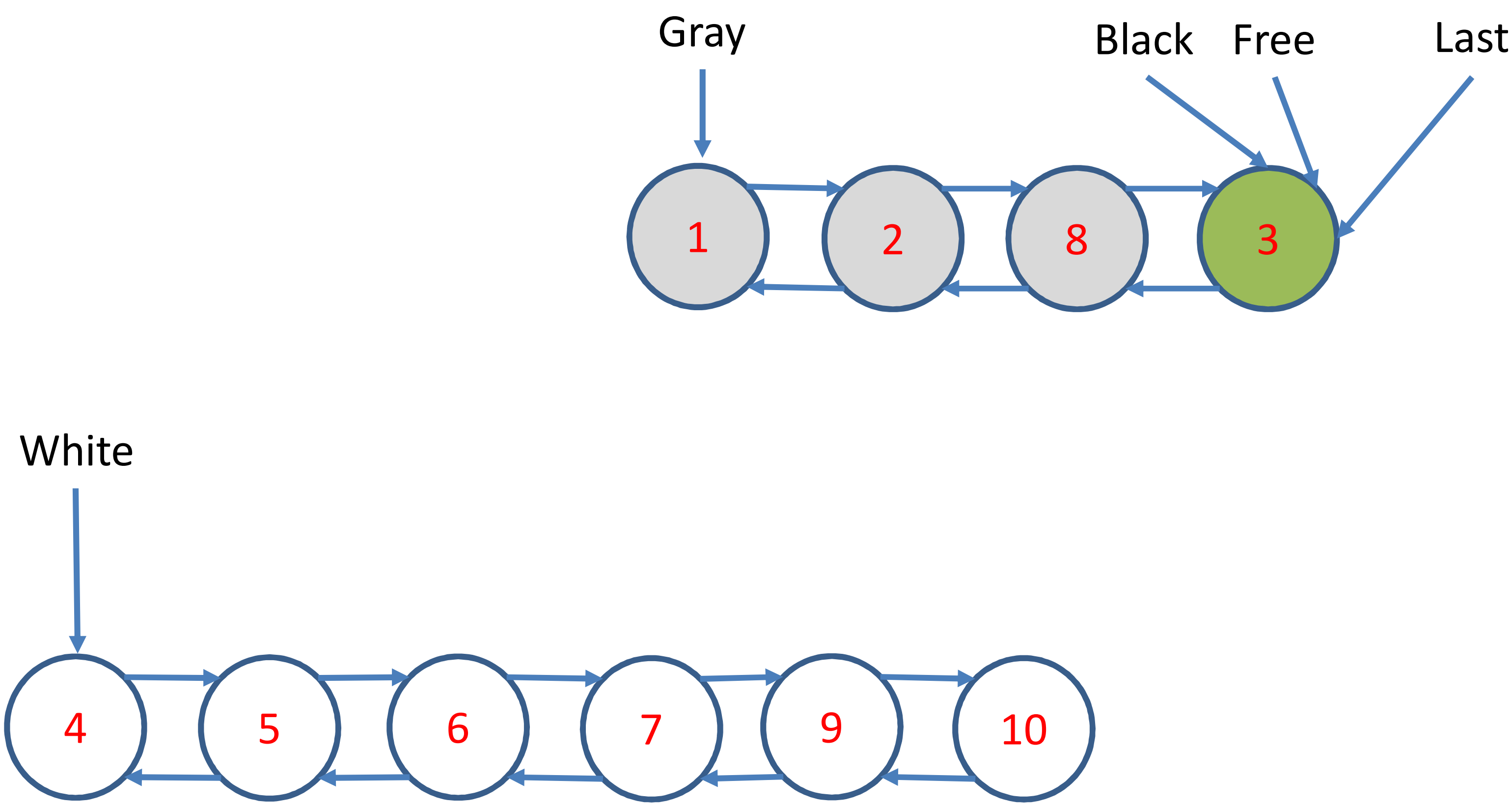


Black = NULL
Free
Last
3
9
10
Gray = NULL
White
1
2
4
6
7
8
10

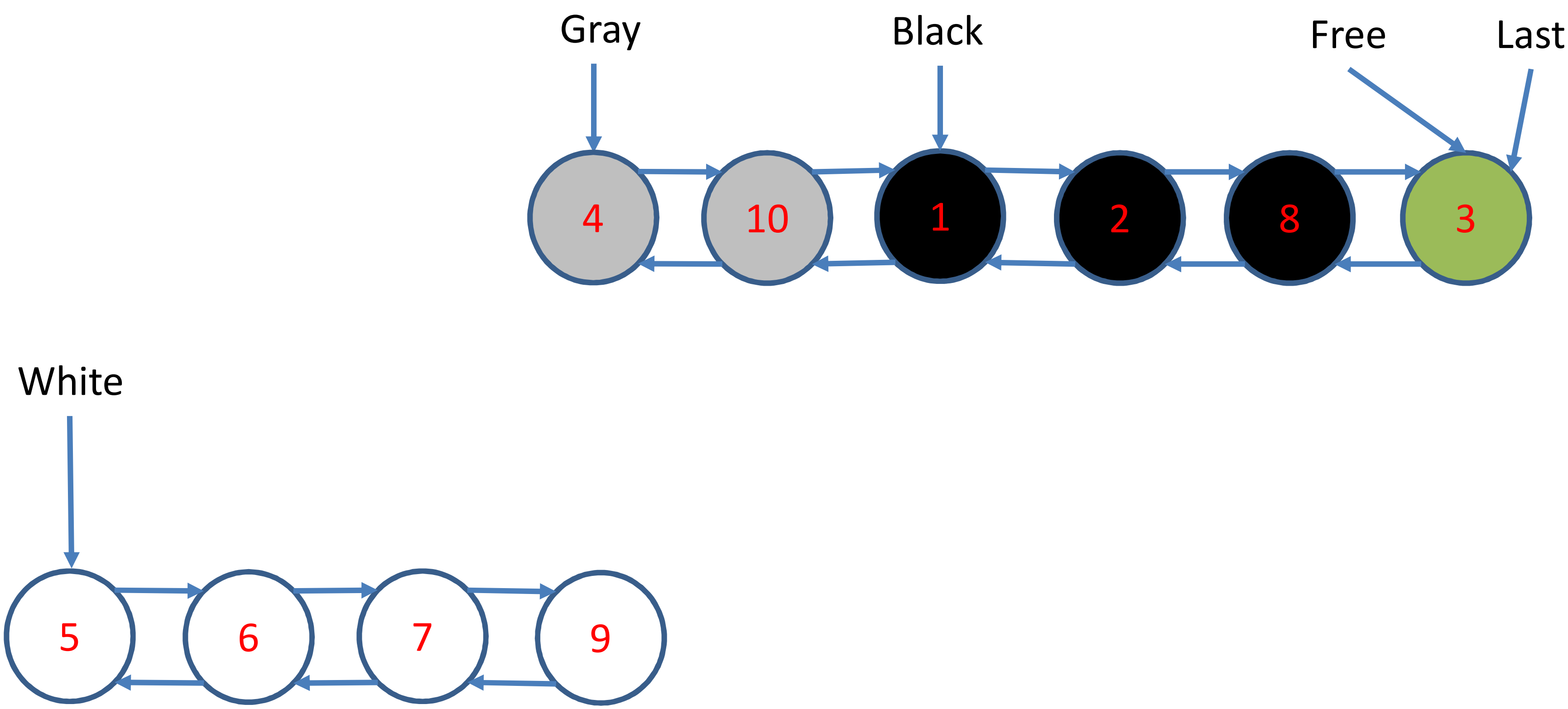
Let’s improve this

- Notice that every live object changes from white, to gray, to black.
- We can make this process more efficient and automatic by connecting some of the linked lists.
- We are also going to color newly allocated objects black.
- We’ll examine why later.

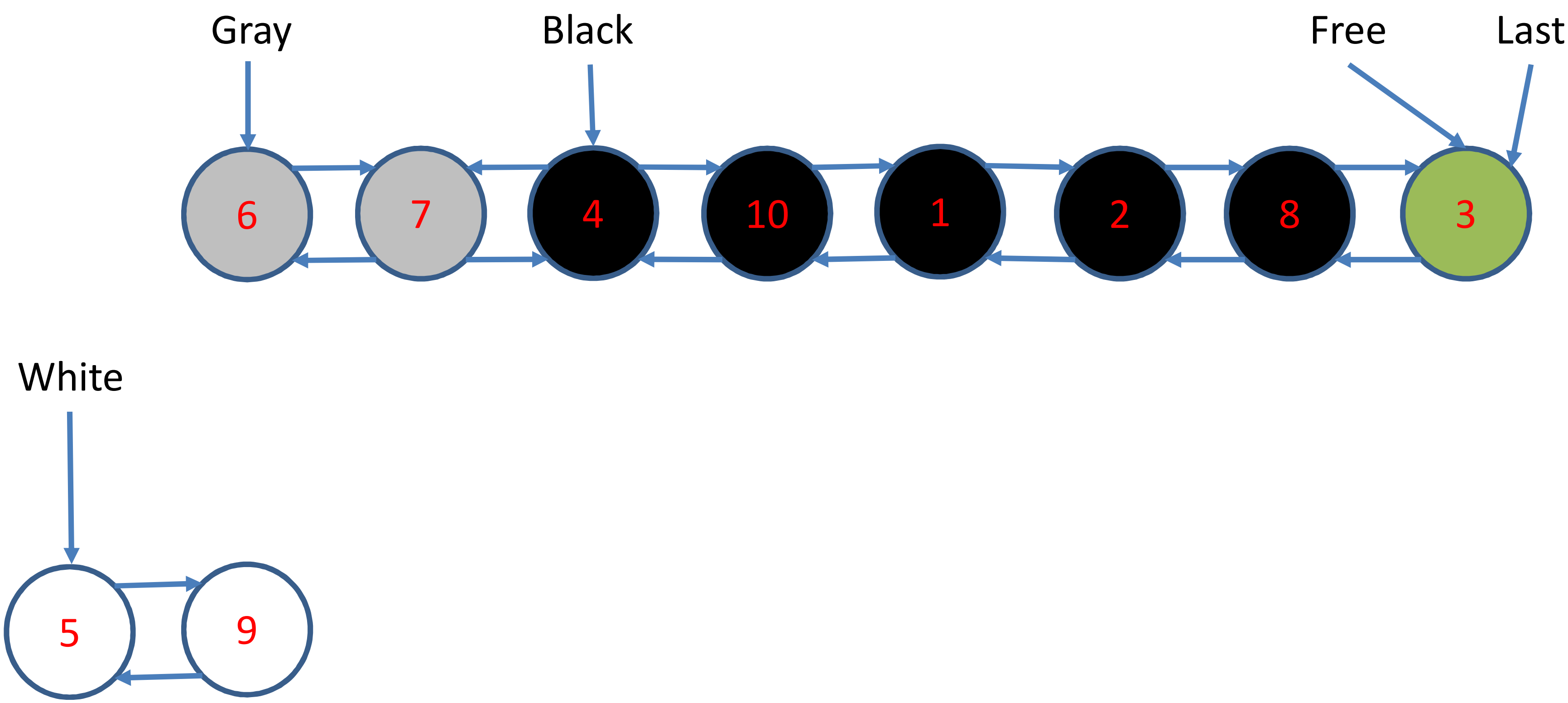
Scan/Trace root set



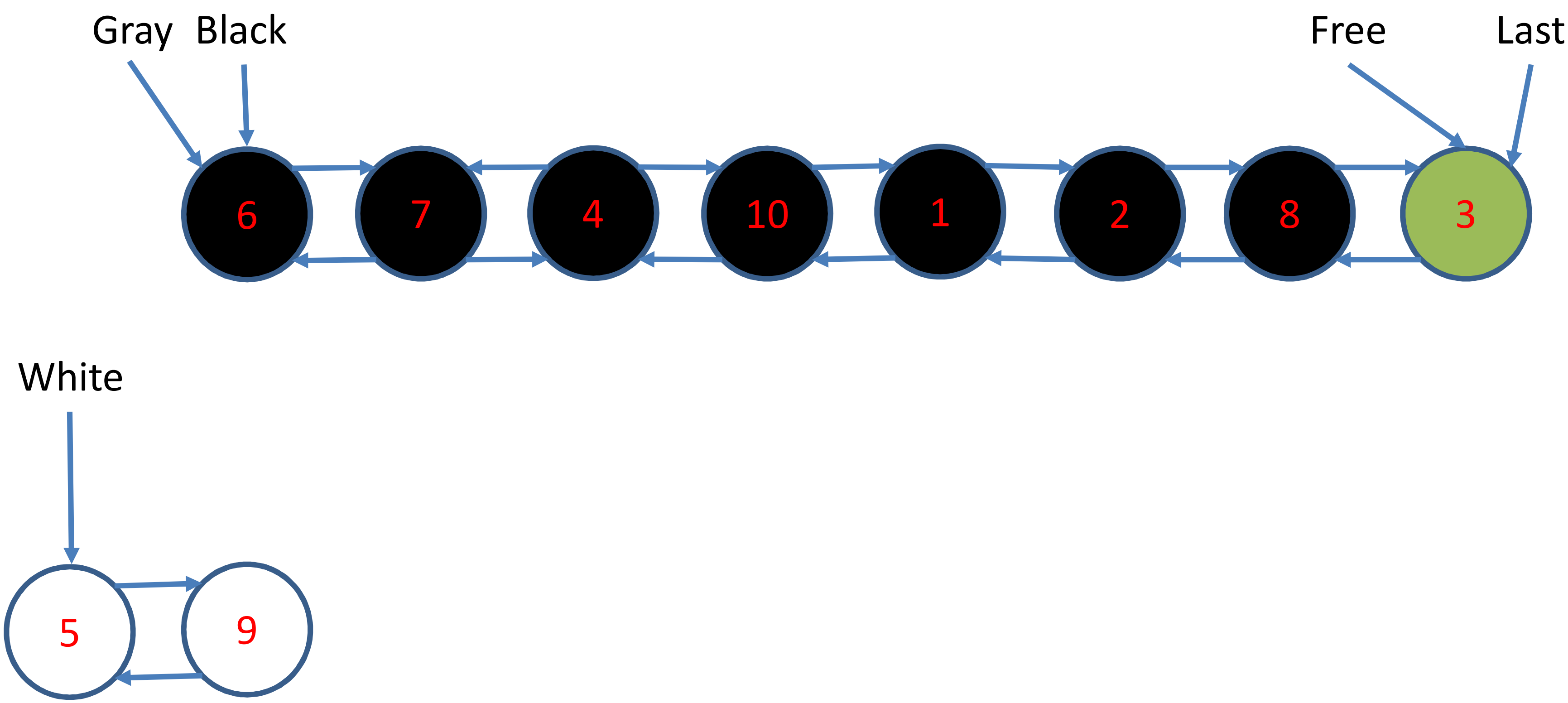
Tracing object graph



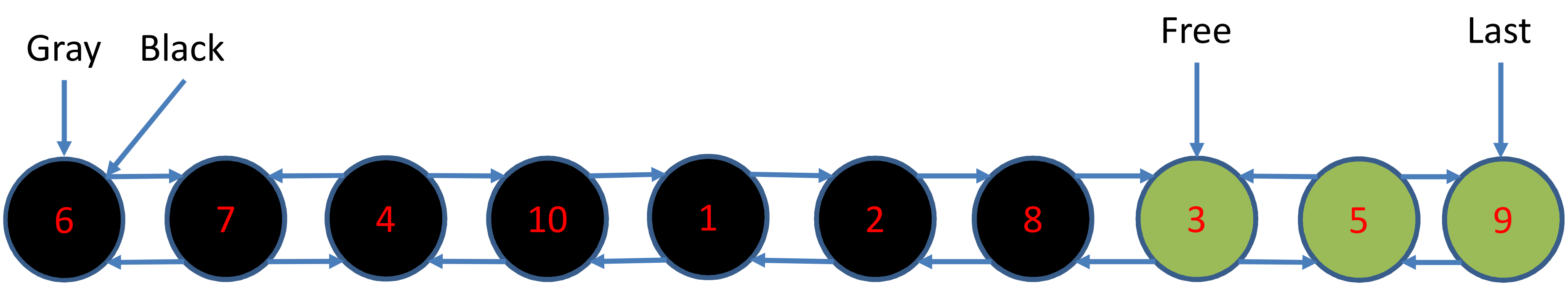
Tracing object graph



Gc cycle complete when Gray == Black



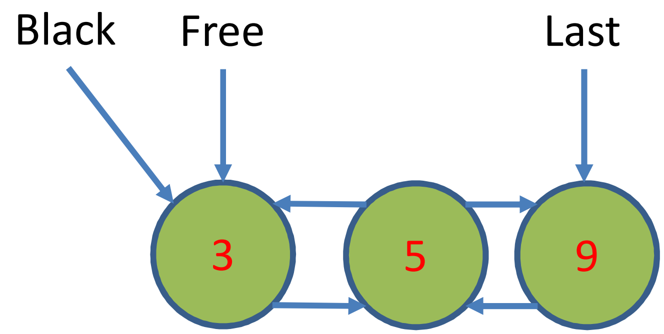
Add garbage to free set

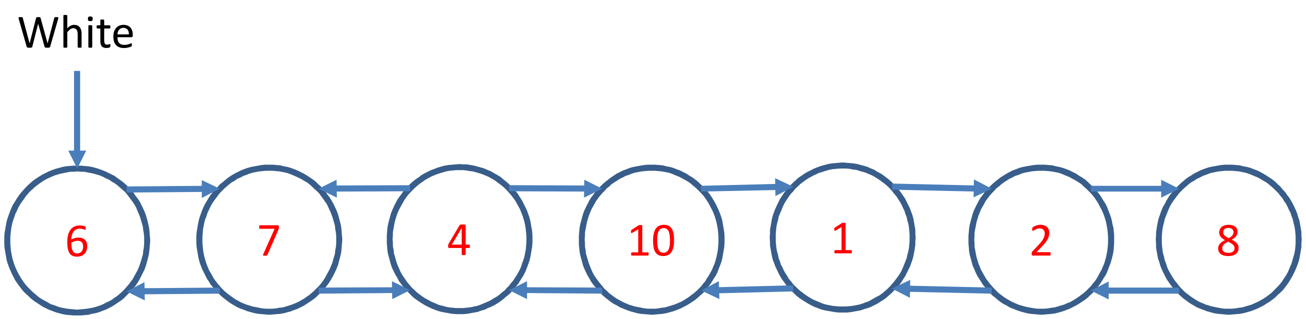


White = NULL

Flip to start new gc cycle

Gray = NULL





Atomic GC Inner loop

```
void full_gc() {
    flip();
    scan_root_set();

    do {
        scan_gray_set();
    } while (gray != black);

    free_garbage();

   gc_count = gc_count + 1;
}
```

Part 2:
Real-time Concurrent GC

- How do we make the atomic collector a concurrent collector?

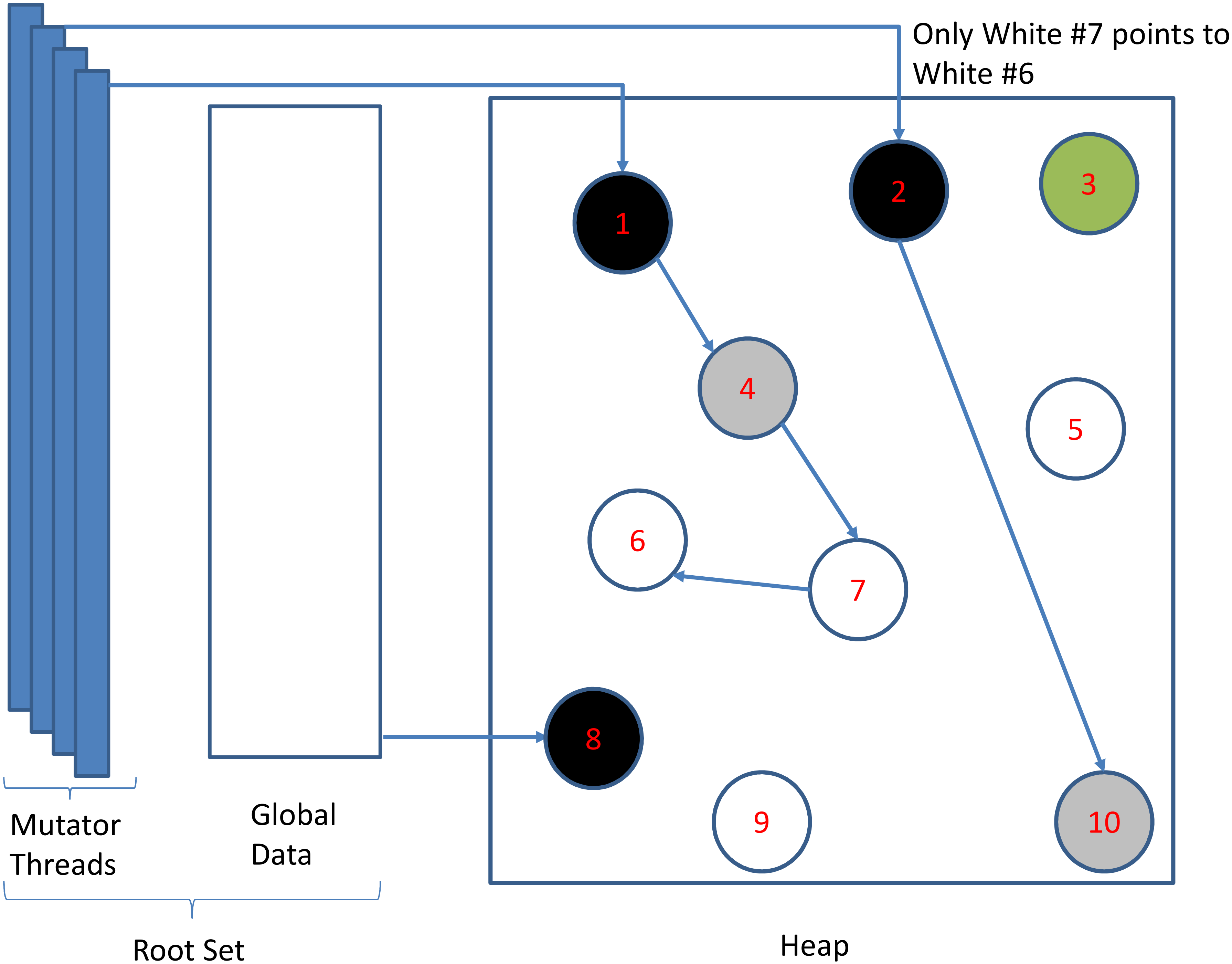
1. Only stop the mutators briefly at the beginning of a new gc cycle when flipping.
2. Introduce a “write barrier” to allow the mutator and gc threads to run concurrently after that for the rest of the cycle.

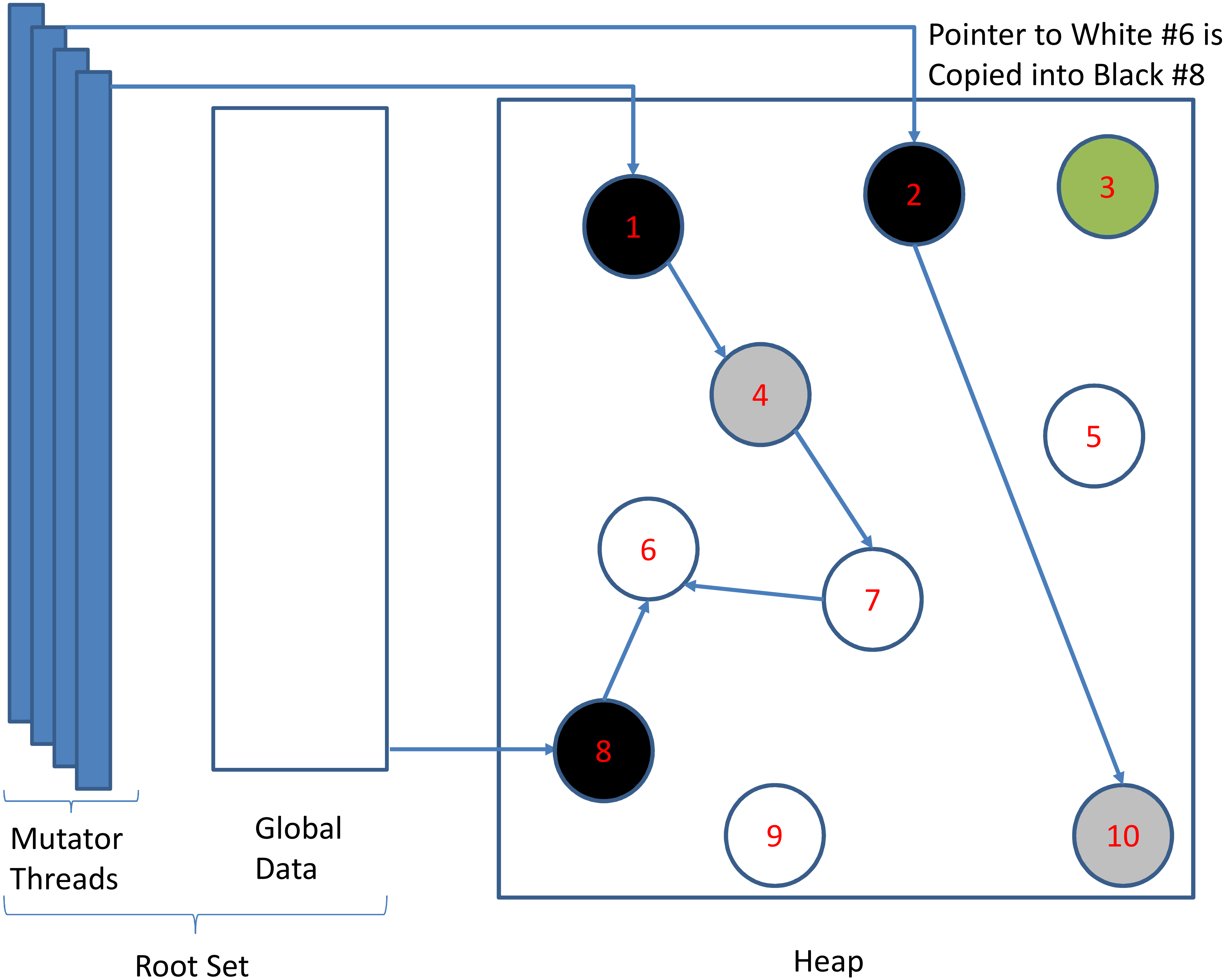
Snap-shot at Beginning

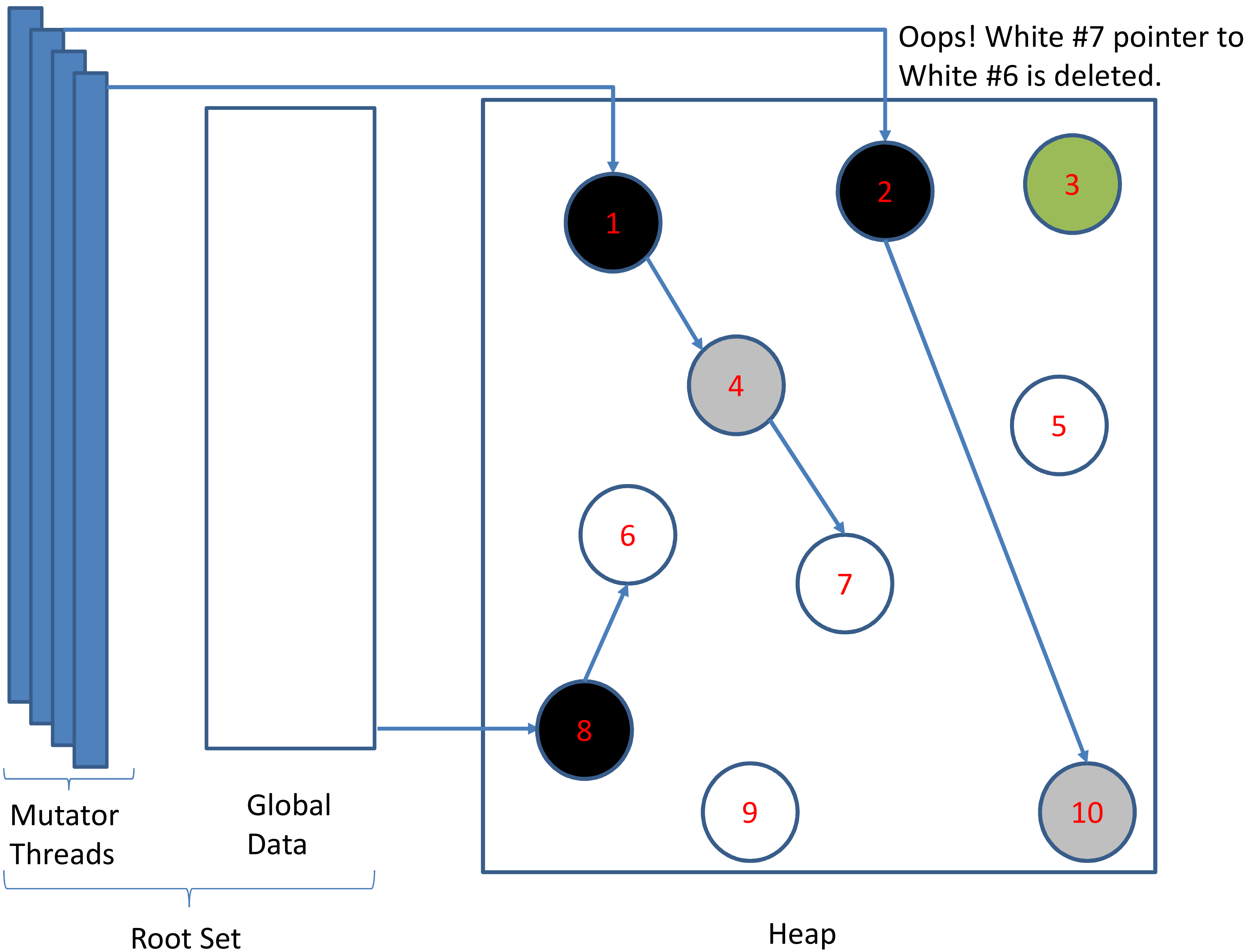
- Conceptually take a complete “snap-shot” of the memory graph when we flipped.
- We are going to retain all objects that are reachable at that time, even if some of them become garbage later during this gc cycle.
- The write barrier allows this by intercepting any mutator pointer writes that might corrupt the snapshot.

Why do we need a Snap-shot?

- Suppose a pointer to a white object is stored in exactly one place.
- What if we copy that pointer into another object that is already marked black, and then overwrite the original pointer.
- We have now lost the original pointer, and placed a copy into a black object that will not be scanned again.
- Without a snap-shot, the white object will incorrectly become garbage because it was not traced before getting overwritten.
- With a snap-shot the object stays alive because it was alive when we flipped.


Only White #7 points to
White #6
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap


Pointer to White #6 is
Copied into Black #8
1
2
3
4
5
6
7
8
9
10
Mutator
Threads
Global
Data
Root Set
Heap


Oops! White #7 pointer to White #6 is deleted.
1
2
3
4
5
6
7
8
9
10
Mutator Threads
Global Data
Root Set
Heap

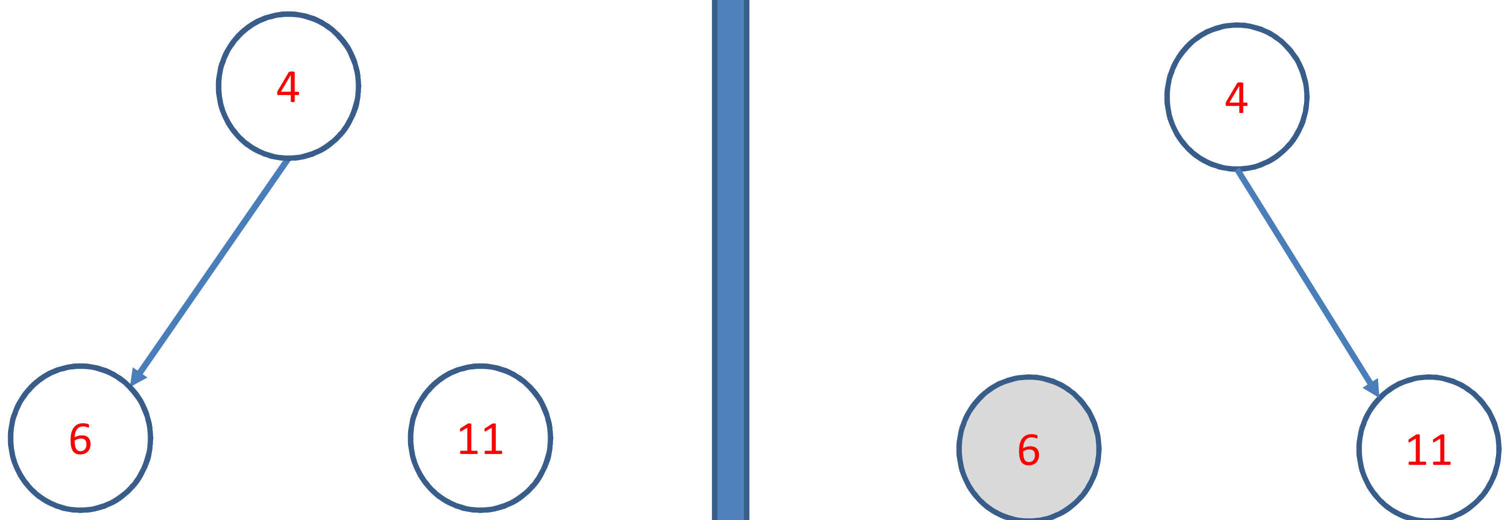
The Write barrier

- The write barrier intercepts pointer writes and examines the pointer about to be overwritten.
- If a pointer to a white object is about to be overwritten, we must make that object gray now, or at some later point in this gc cycle.
- We don’t need to apply this to initializing pointer writes (i.e – Writing pointers into a newly allocated object).
- We’ll also see soon why we also don’t need to apply this to stack or register writes.

Write Barrier

Before Write

After Write



If we are about to overwrite a pointer to a white object then

We must make that object gray now or at some later point in the current gc cycle

Flipping

- When flipping we pause every mutator thread.
- Each paused mutator copies its stack and register state, and then it may resume.
- We don’t need to apply the write barrier to the stack or registers because we atomically copy them somewhere when we flip.
- Each mutator does this independently and only sees a pause time proportional to it’s own stack and register copy time.
- Flipping completes after every mutator has resumed. This isn’t necessary, but it’s simple and usually quick.

Does this really work? - Induction

- Prove: All objects alive at the start of a gc cycle will be retained at the end of the cycle.
- Base case: All objects directly reachable from the saved stack and register state after a flip will be retained after tracing.
- Induction step: Tracing the transitive closure of the object graph reachable from the base case, in conjunction with a mutator write barrier, guarantees that all objects alive at the start of the gc cycle are retained at the end of the cycle.

Flipping and the Write Barrier

- We can trade off flip pause time and efficiency.
- If we want no mutator pauses when flipping, we need to apply the write barrier to stack and register pointer writes.
- This costs a lot in performance.
- In exchange, we don’t ever need to explicity stop the mutators and copy the stacks and registers.
- We only need to restrict all mutator allocation for the short time it takes to swap the marked and unmarked color. No explict pauses or copying are required.
- This probably only makes sense for extremely pause time sensitive applications.

Conservative versus Perfect Collection

- Perfect collection means we can accurately identify all heap pointers, including pointers from stacks and registers.
- Perfect collection only requires Tri-Color marking (no green).
- Conservative collection loosens this constraint and treats anything that looks like a pointer as a pointer.
- Conservative collection uses Quad-Color marking so conservative pointers don't try to "retain" free objects that might look like garbage.
- Conservative collection is helpful in using gc with languages like C, because we do not have "perfect" metadata for stack and register pointers.
- Providing pointer metadata when allocating helps to produce a practical blend of conservative and perfect collection

Allocate Black

- We allocate black because new objects were not alive when we flipped, so they are not part of the snap-shot.
- New objects could be incorrectly classified as garbage at the end of the gc cycle if we didn't allocate black.
- Unfortunately, this extends the time it takes to reclaim newly allocated objects that quickly become garbage. They can only be reclaimed in the gc cycle that starts after their allocation.

Continuously Concurrent GC

- There is no Oracle to tell you when an object becomes garbage.
- A “maximally efficient” tracing gc would collect garbage the moment it is created.
- If you have cores (and power) to burn, then continuously gc on as many threads as make sense for your application.
- This can dramatically reduce the maximum heap size.
- Trade off is faster gc vs larger heaps.
- Doing gc work only in response to allocation work is often a mistake. See first bullet point.
- Only slow/stop gc when you have no live objects or you want to save power or use some gc cores for mutator work.

The Coalescer

- Coalescing is an optional but valuable way to minimize heap size.
- Coalescing competes with allocation on mutator threads to “unallocate” free objects and return whole pages of free objects to their original unallocated state.

Status

- Single threaded gc runs continuously in a simple test program running multiple mutator threads.
- The WCL implementation of Common Lisp was instrumented with the write barrier and converted to use RTGC. Thousands of self recompiles run while the gc continuously runs concurrently.
- Using the coalescer reduced the WCL self recompile heap size by a factor of 8.

Next Directions

1. Integrate with a popular language runtime – probably the JDK, but maybe another language.
2. Use RTGC as a replacement for malloc/free in conventional programs C/C++.
3. Use RTGC as a replacement for ARC (Automatic Reference Counting) Apple's Objective-C and Swift language. Reclaim circular memory graphs without burdening developers!
4. Measure and improve RTGC cache friendliness.