Efficient Mark-Sweep

Filip Pizlo
Apple

Efficient Mark-Sweep

- Introduction
- Why mark-sweep?
- Algorithm
  - Simple Segregated Storage
  - Bump’n’pop
  - Constraint-Based Marking
  - Parallel “Draining”
  - Sticky Mark Bits
  - Conservative Roots
  - Fragmentation Mitigations

Introduction

Tracing

- An object is live if it is transitively reachable from roots

- Roots = registers, stack, global variables
- Transtive reachability = graph search over references between objects

Two Main Types of GC

- Mark-Sweep
- Copying

Mark-Sweep:

Copying:

Mark-Sweep:

Copying:

Mark-Sweep:

Copying:

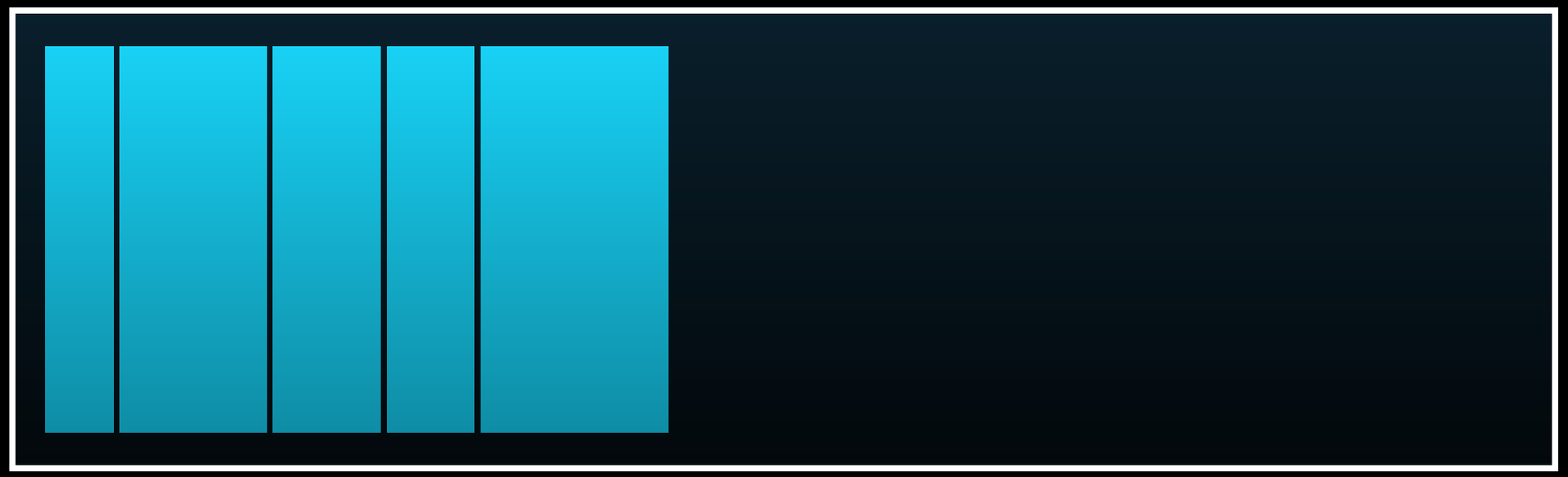
Mark-Sweep:

Copying:

Mark-Sweep:

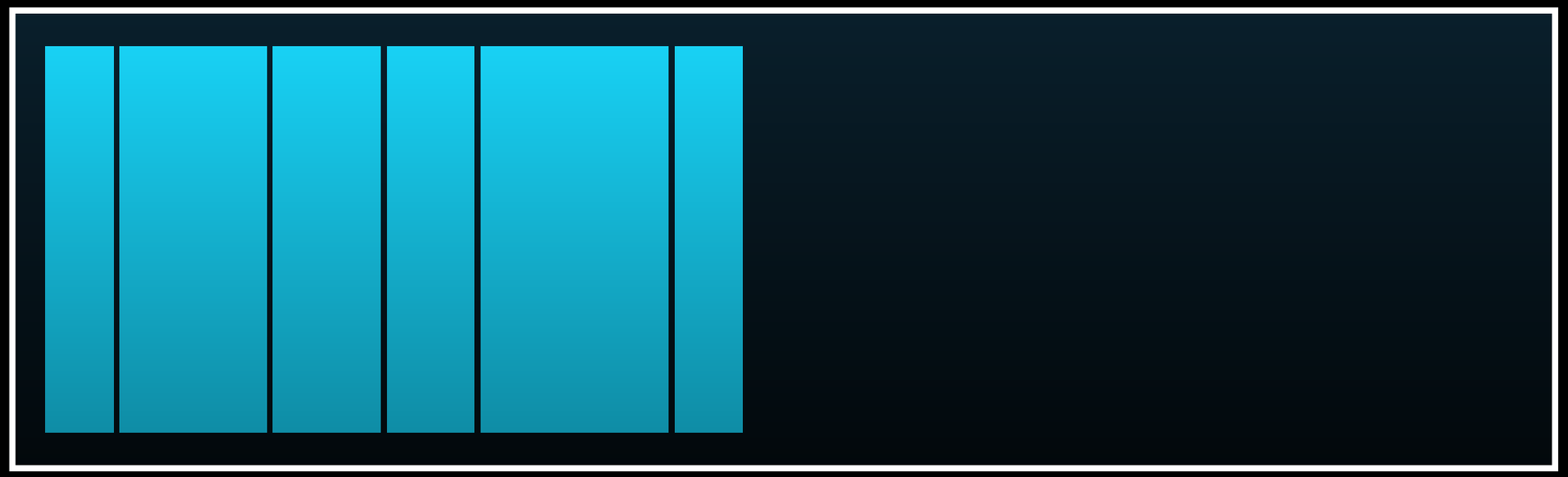
Copying:

Mark-Sweep:



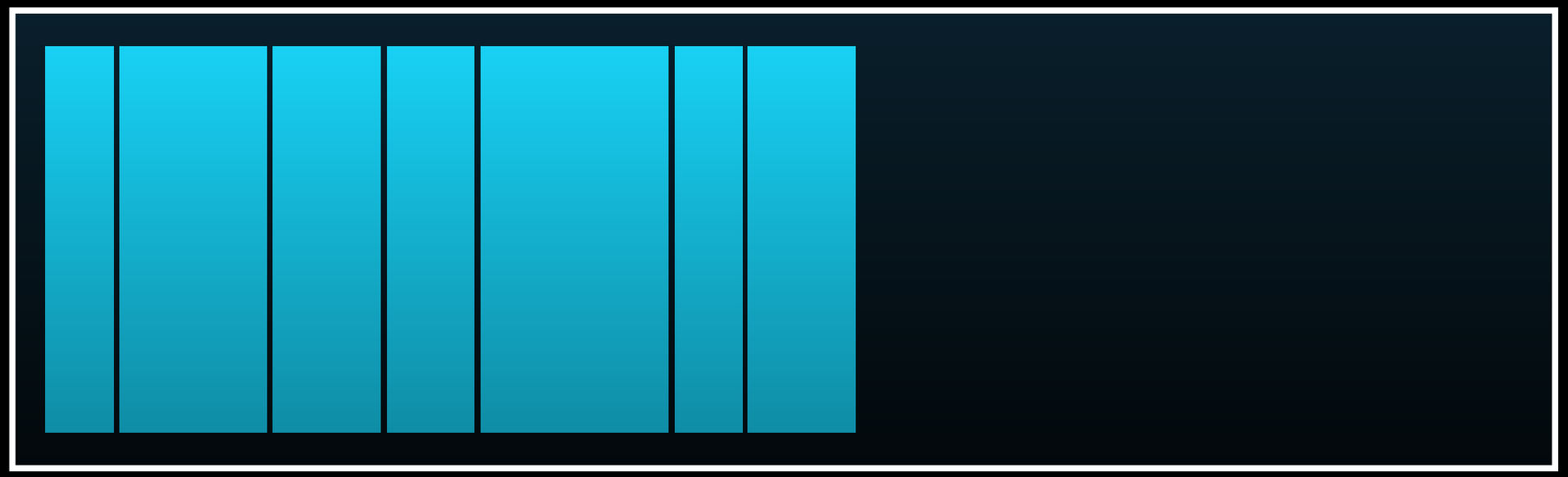
Copying:

Mark-Sweep:



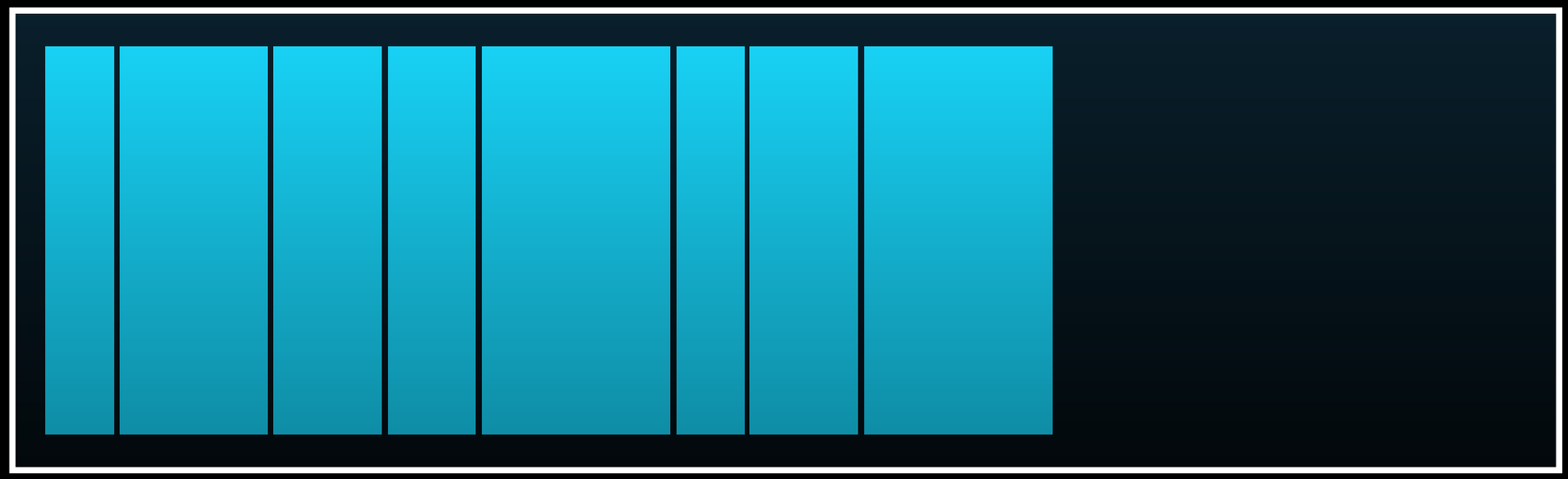
Copying:

Mark-Sweep:



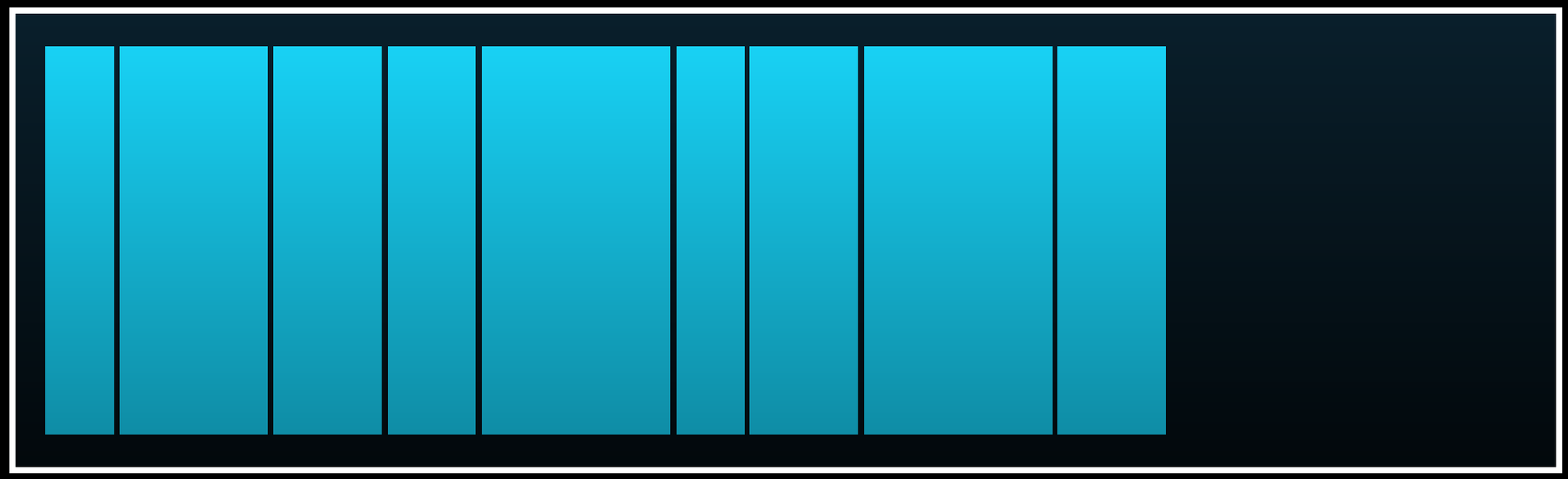
Copying:

Mark-Sweep:



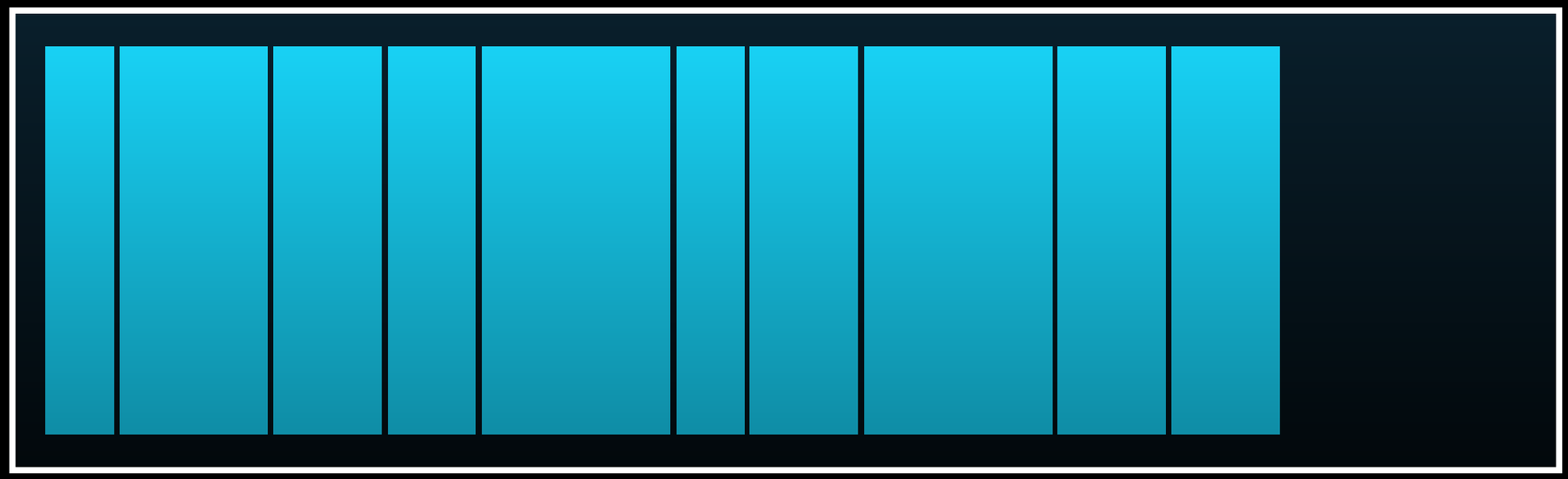
Copying:

Mark-Sweep:



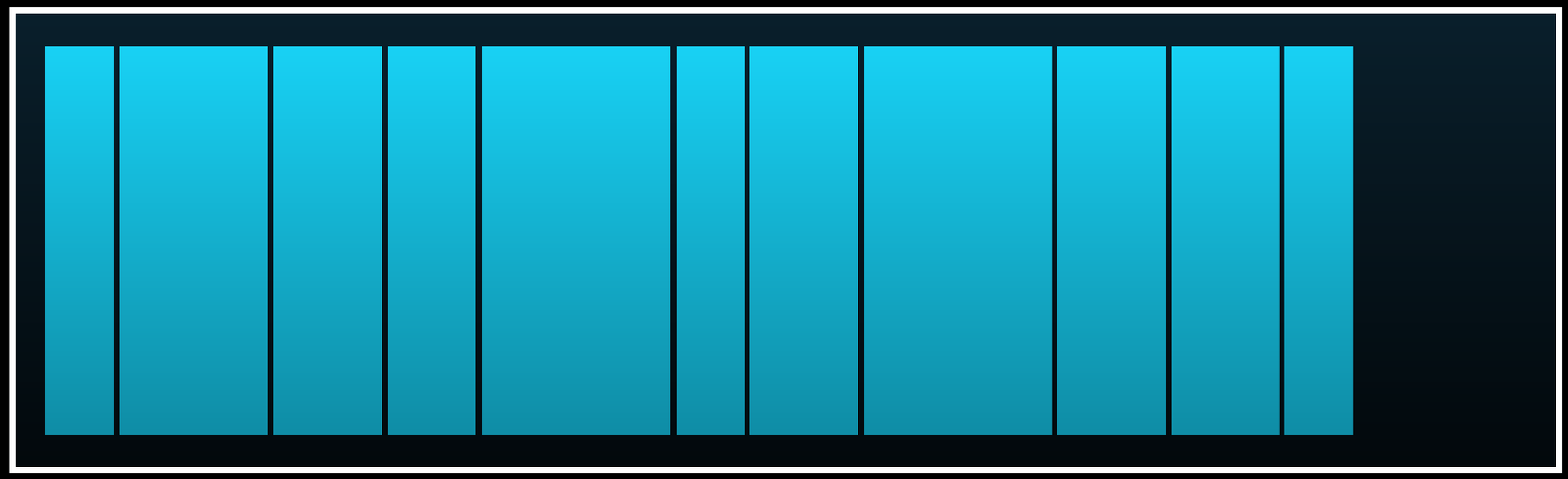
Copying:

Mark-Sweep:



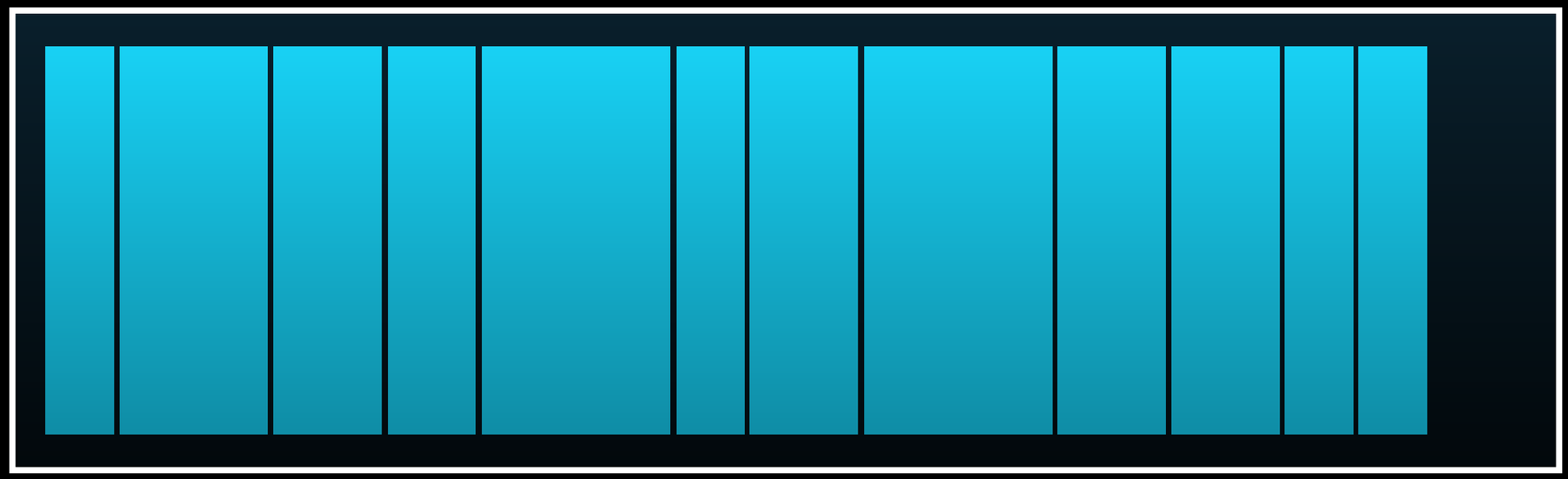
Copying:

Mark-Sweep:



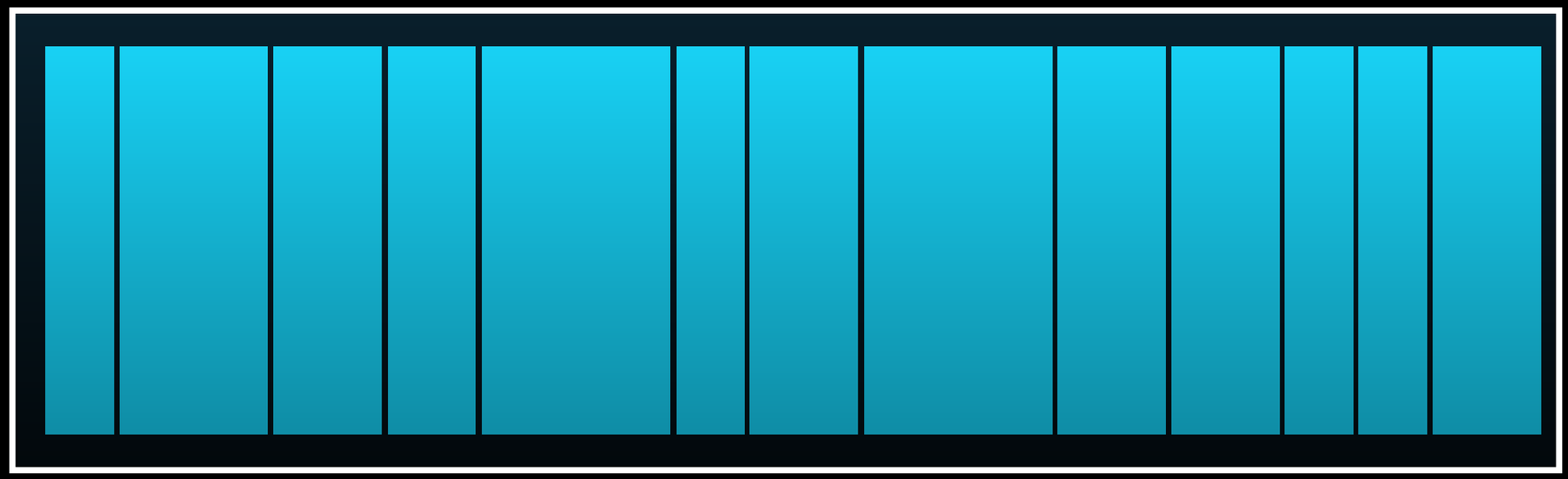
Copying:

Mark-Sweep:



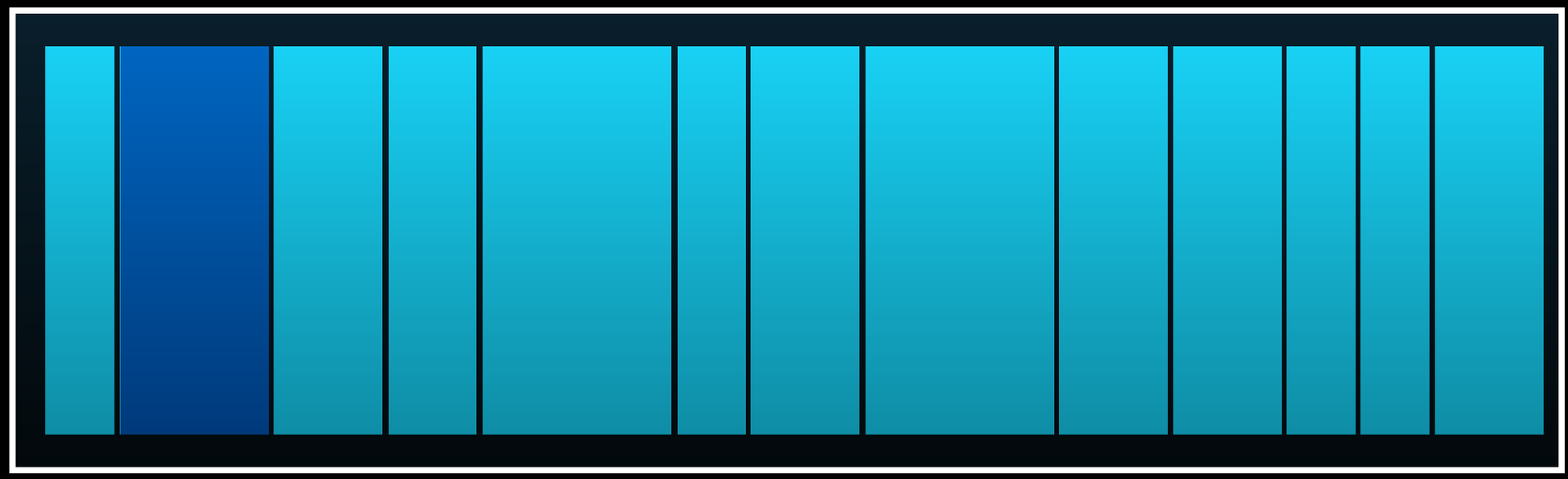
Copying:

Mark-Sweep:



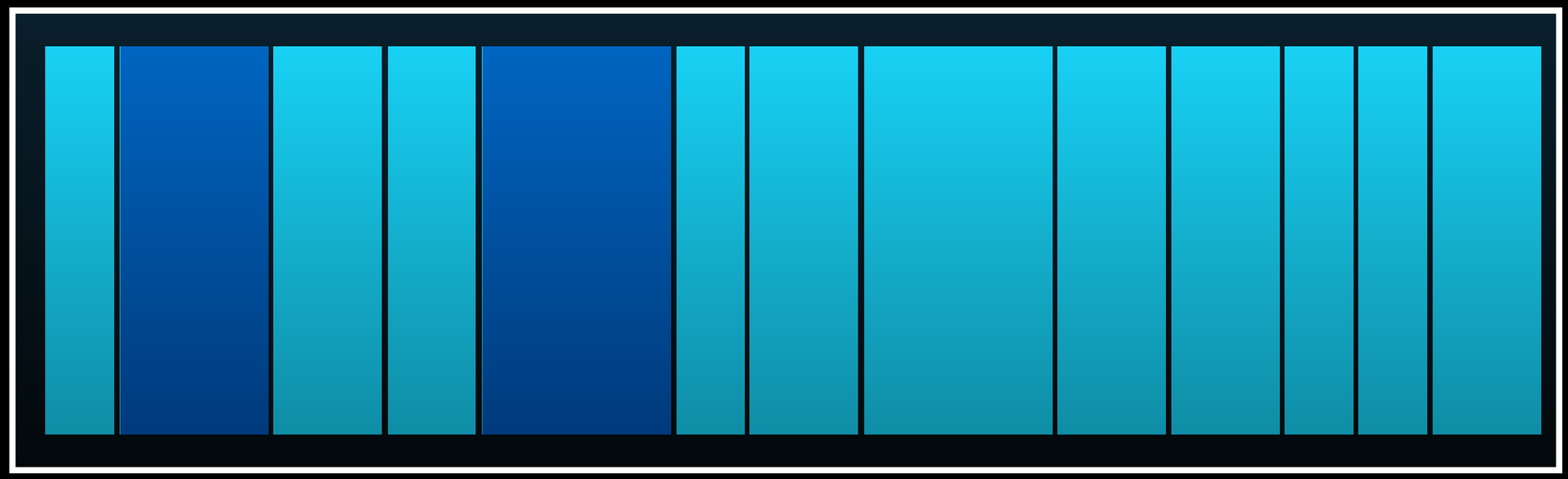
Copying:

Mark-Sweep:



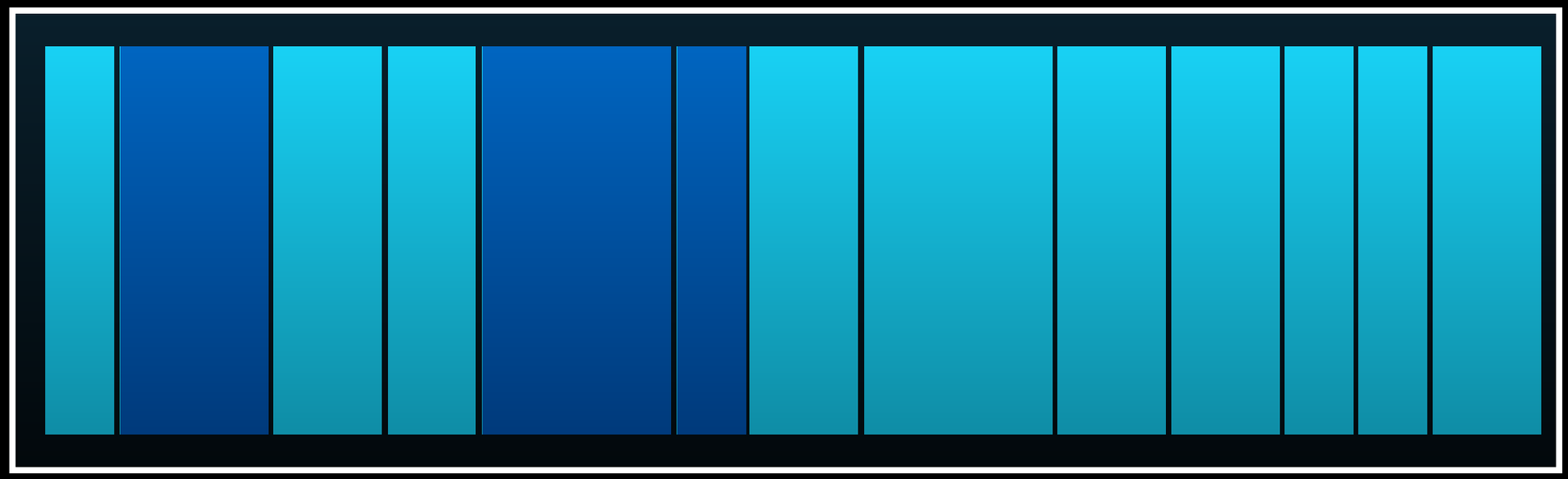
Copying:

Mark-Sweep:



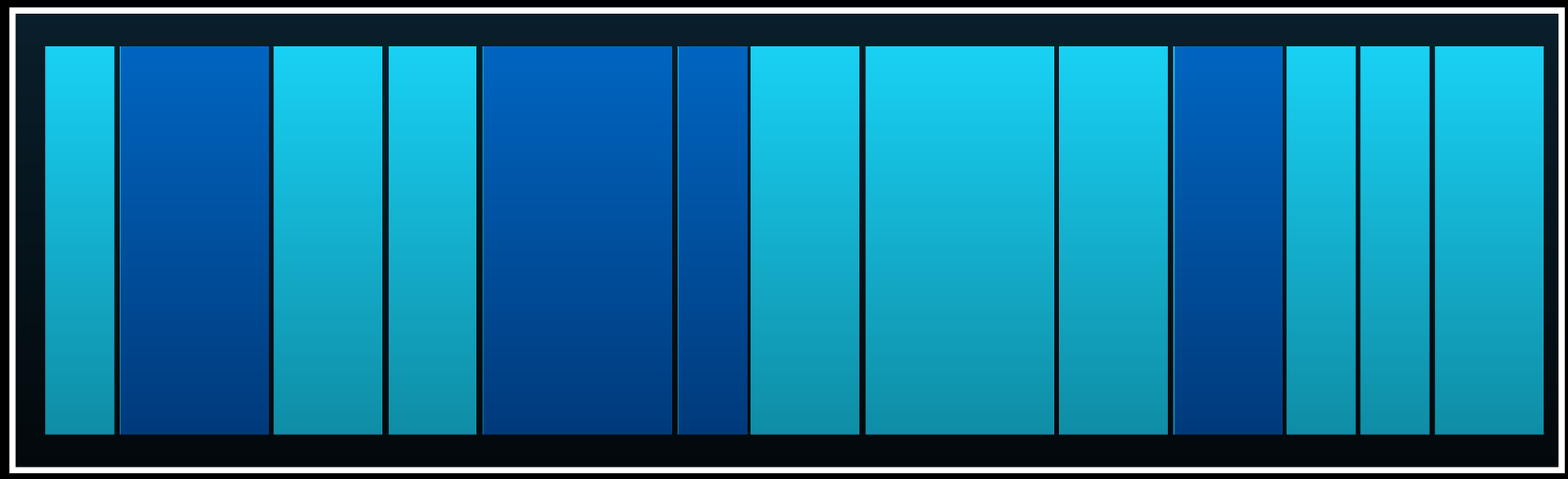
Copying:

Mark-Sweep:



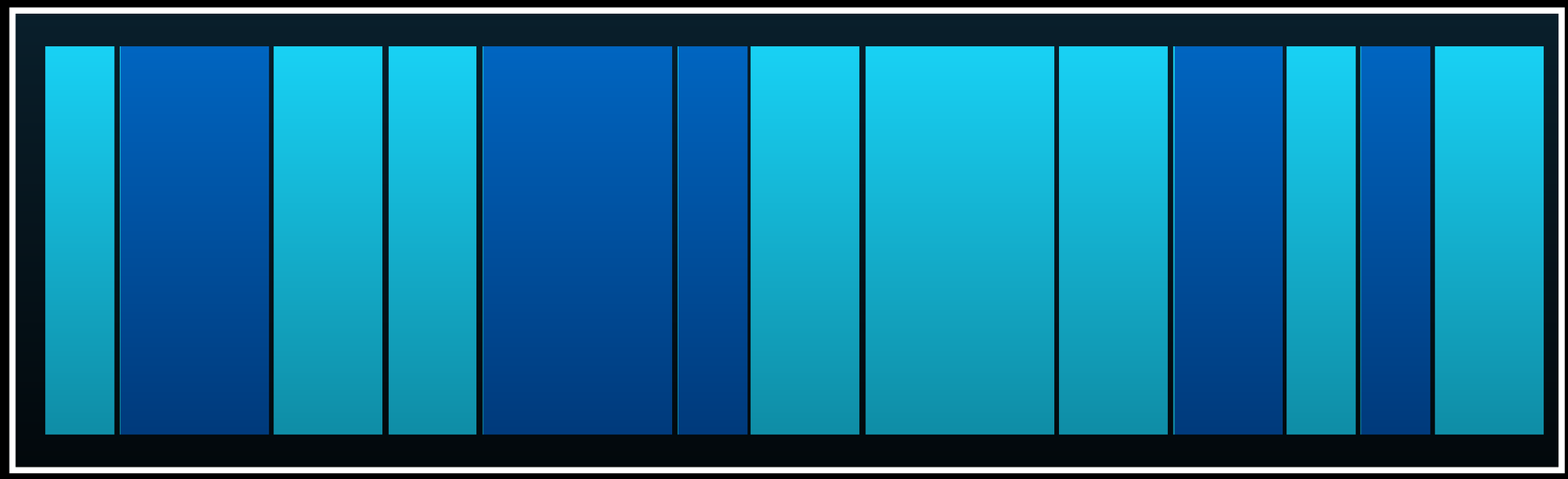
Copying:

Mark-Sweep:



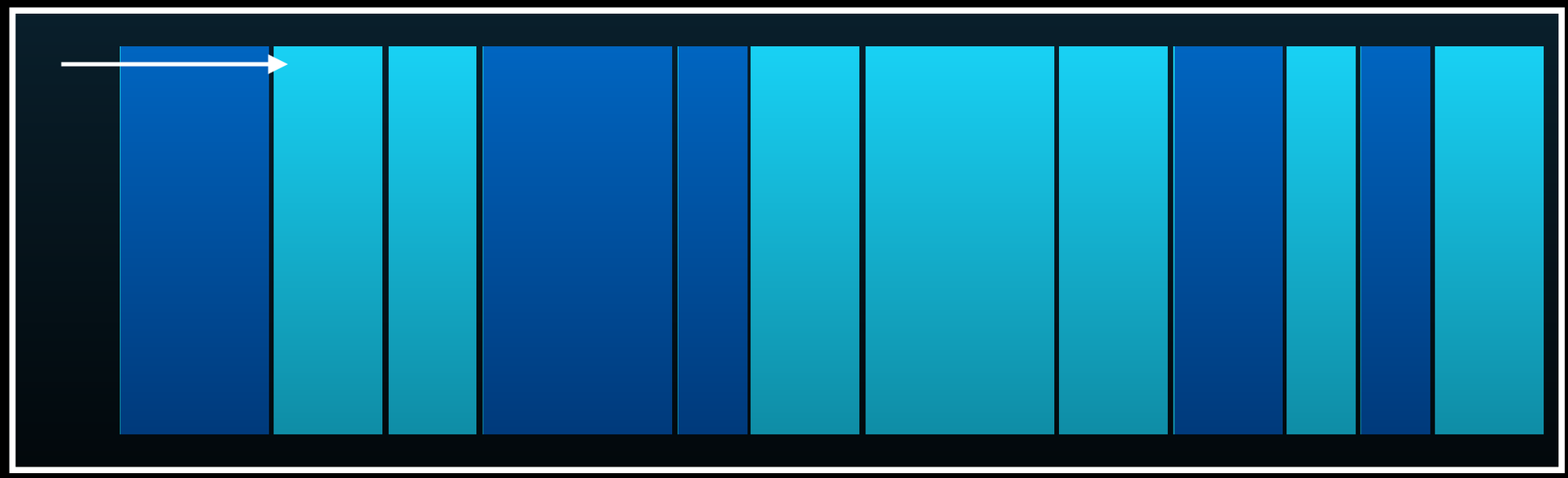
Copying:

Mark-Sweep:



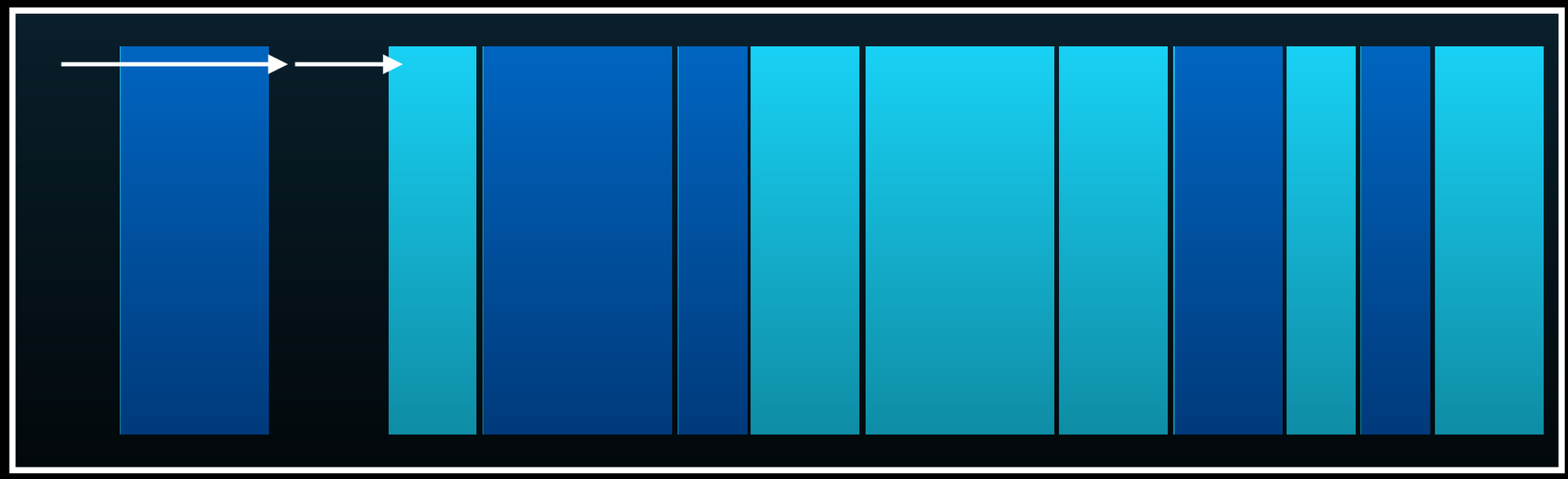
Copying:

Mark-Sweep:



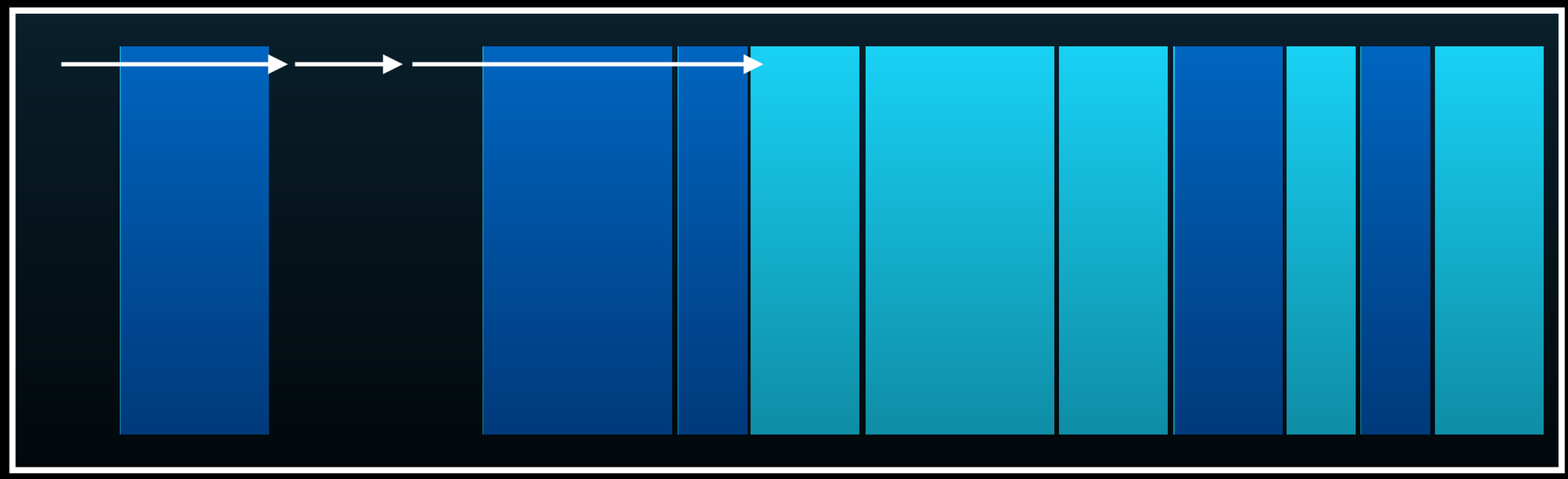
Copying:

Mark-Sweep:



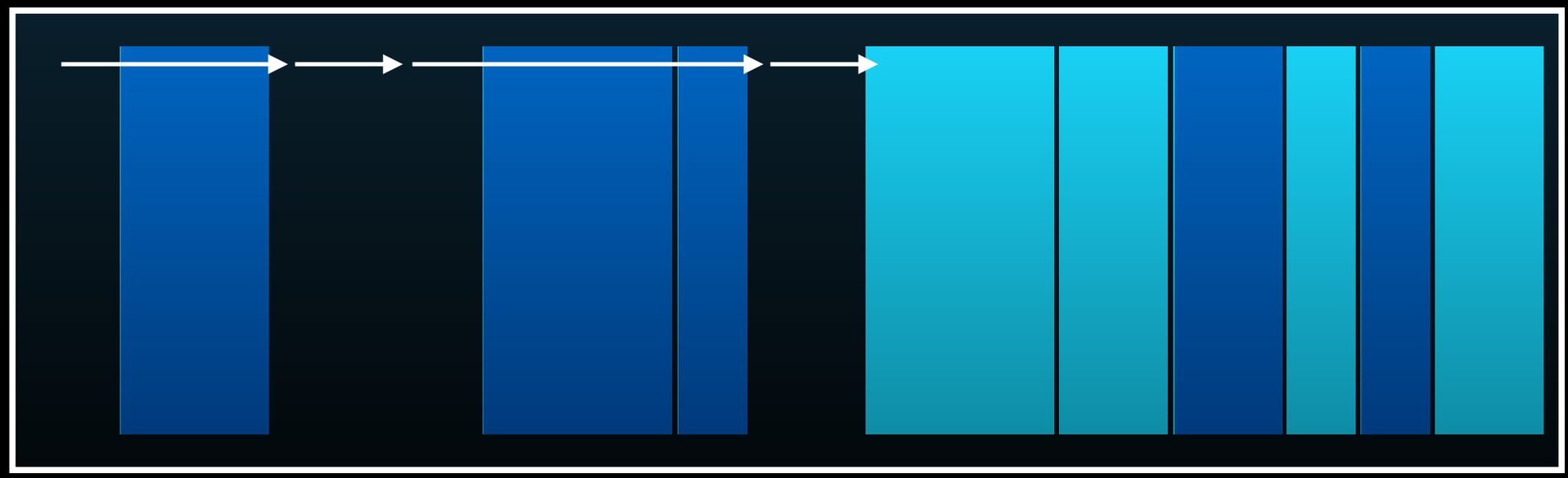
Copying:

Mark-Sweep:



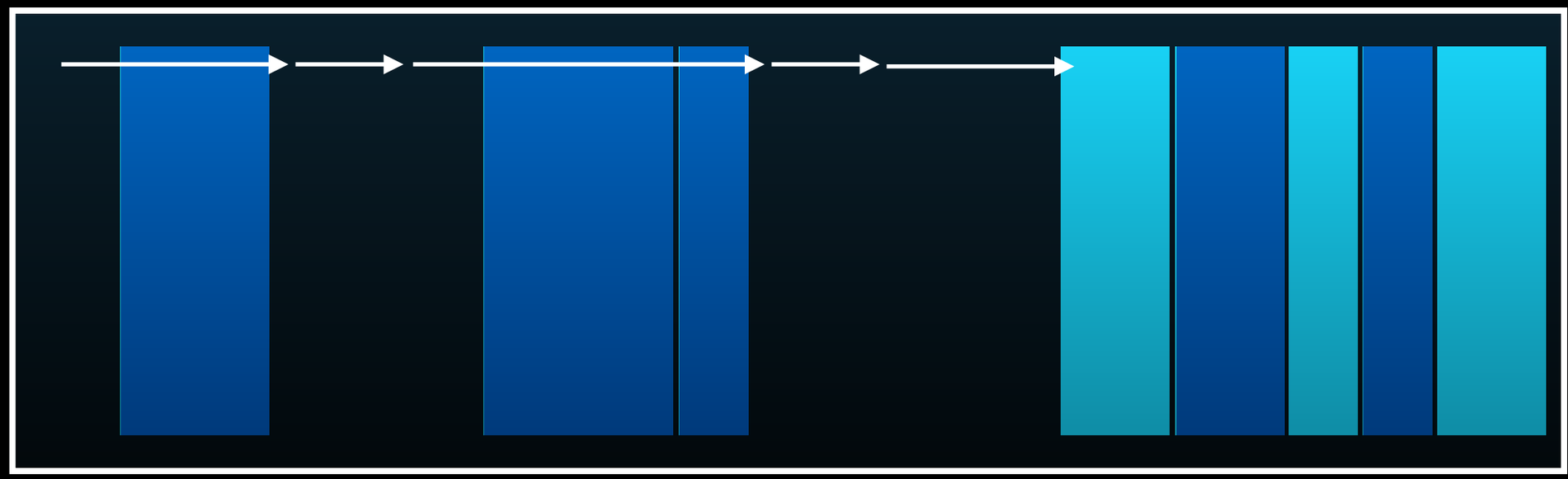
Copying:

Mark-Sweep:



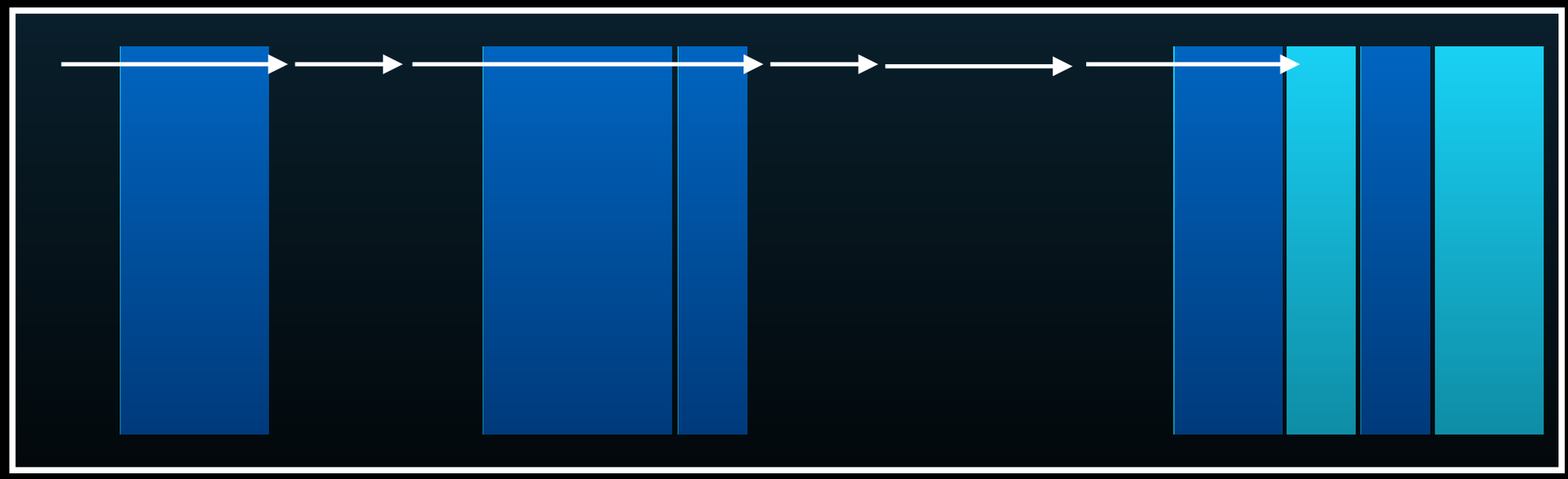
Copying:

Mark-Sweep:



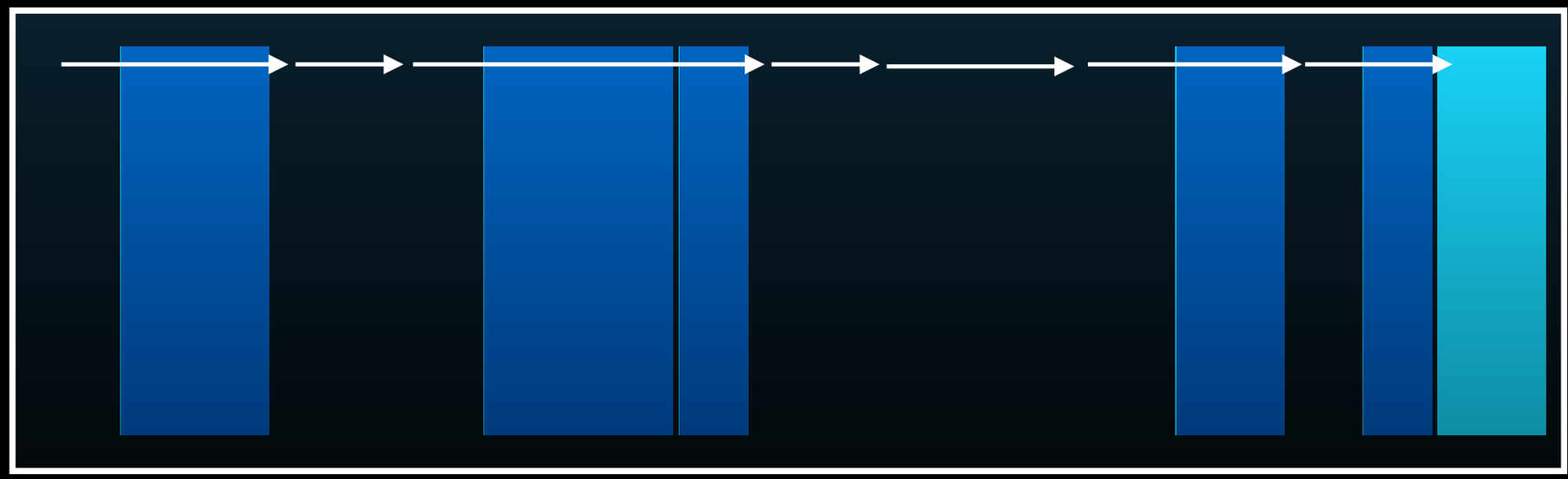
Copying:

Mark-Sweep:



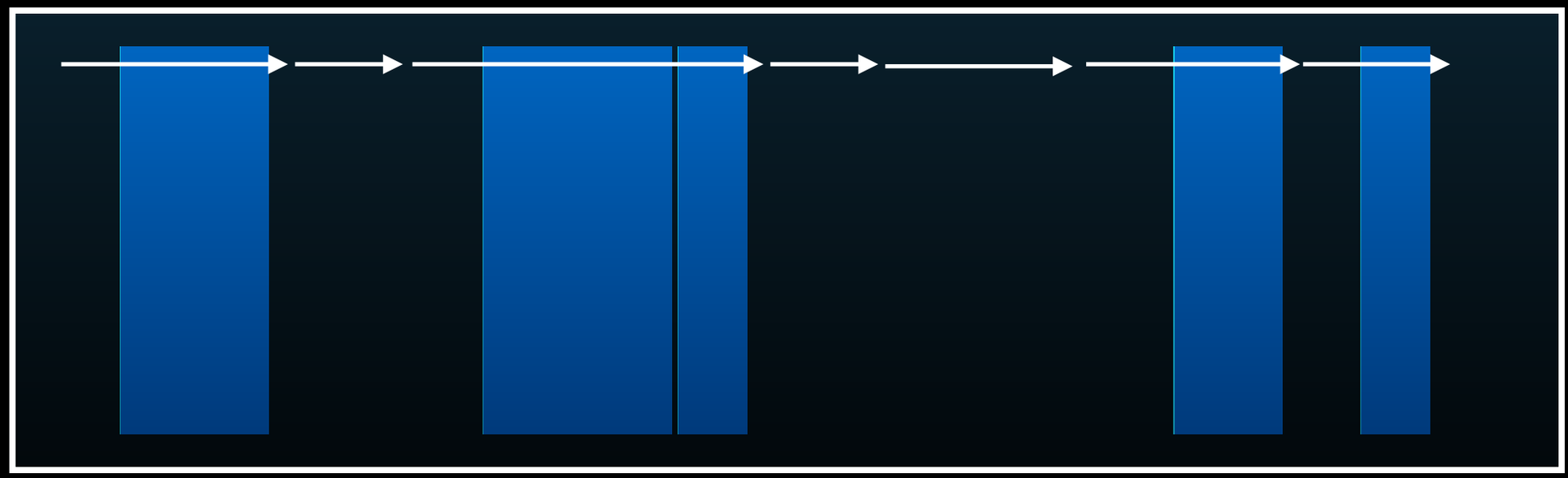
Copying:

Mark-Sweep:



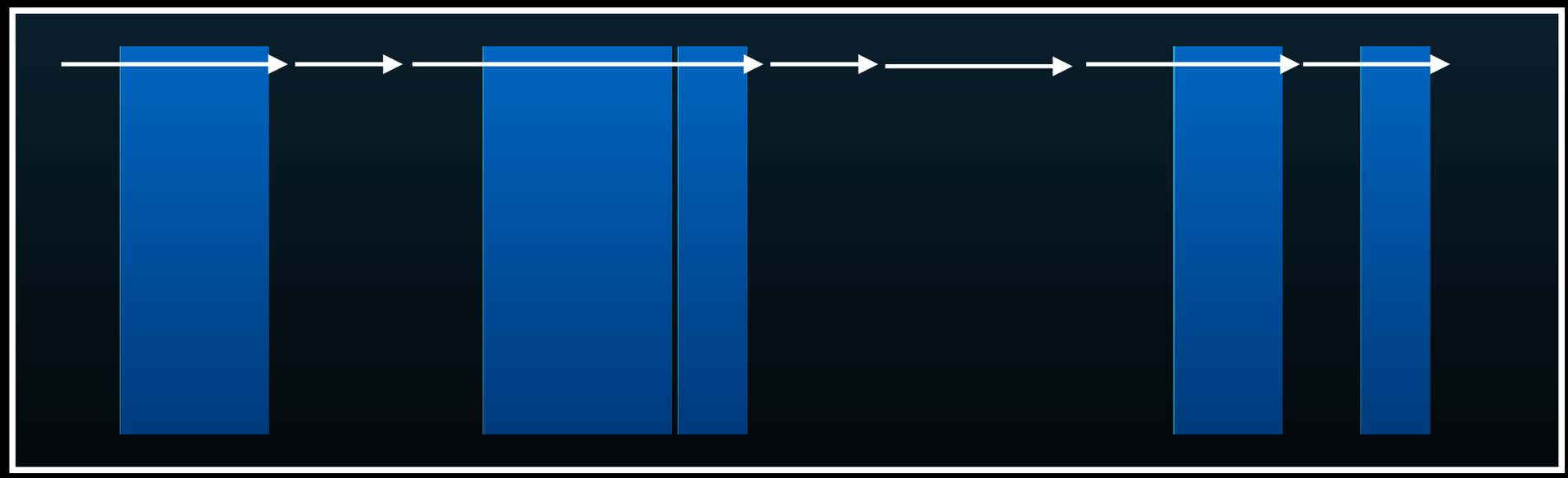
Copying:

Mark-Sweep:



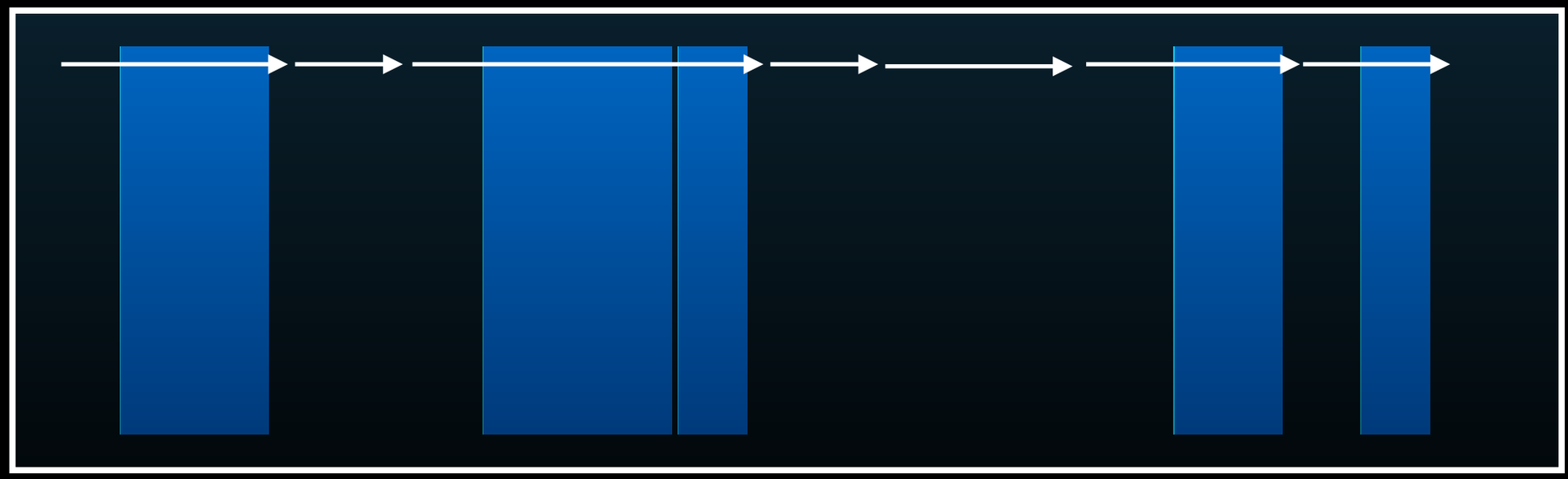
Copying:

Mark-Sweep:



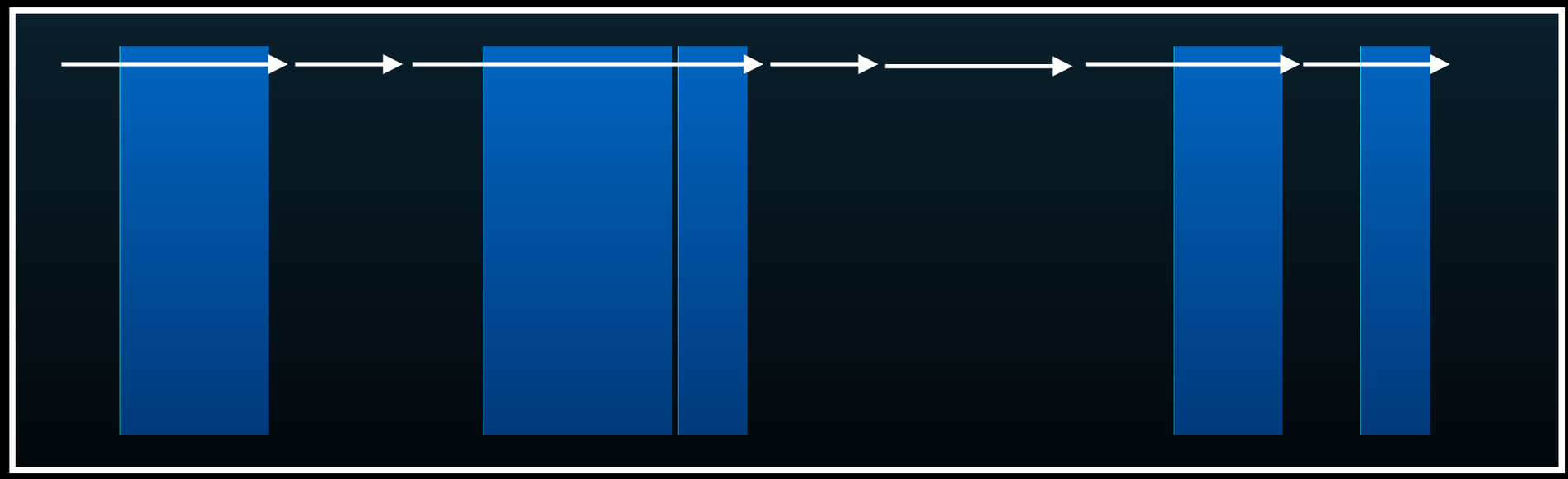
Copying:

Mark-Sweep:



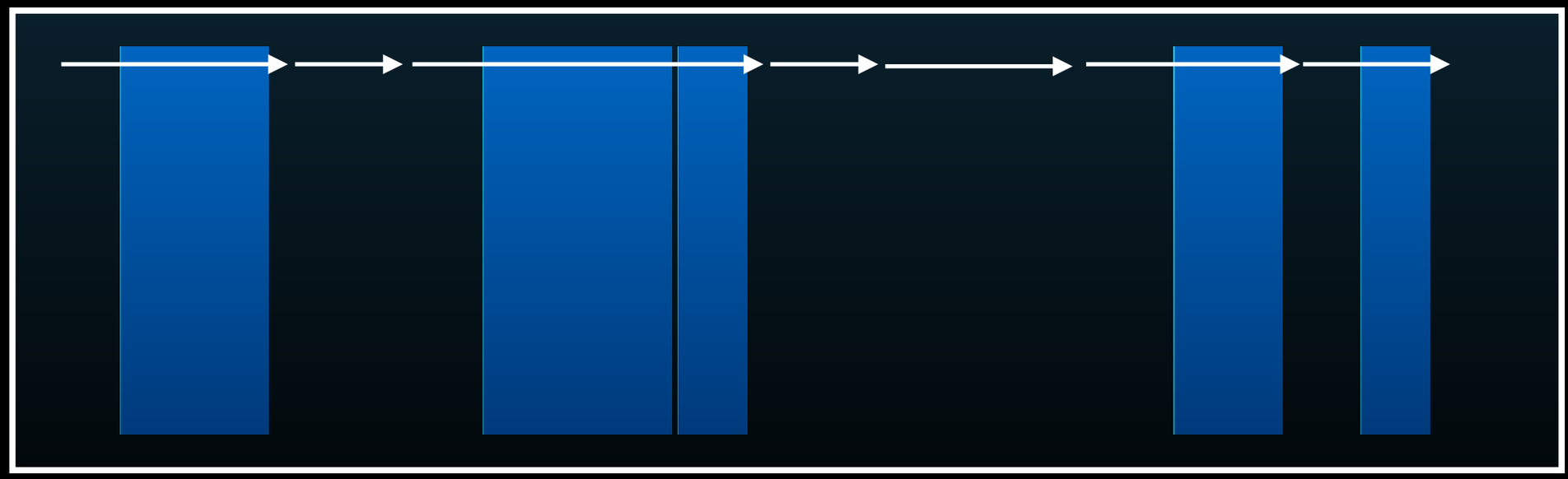
Copying:

Mark-Sweep:



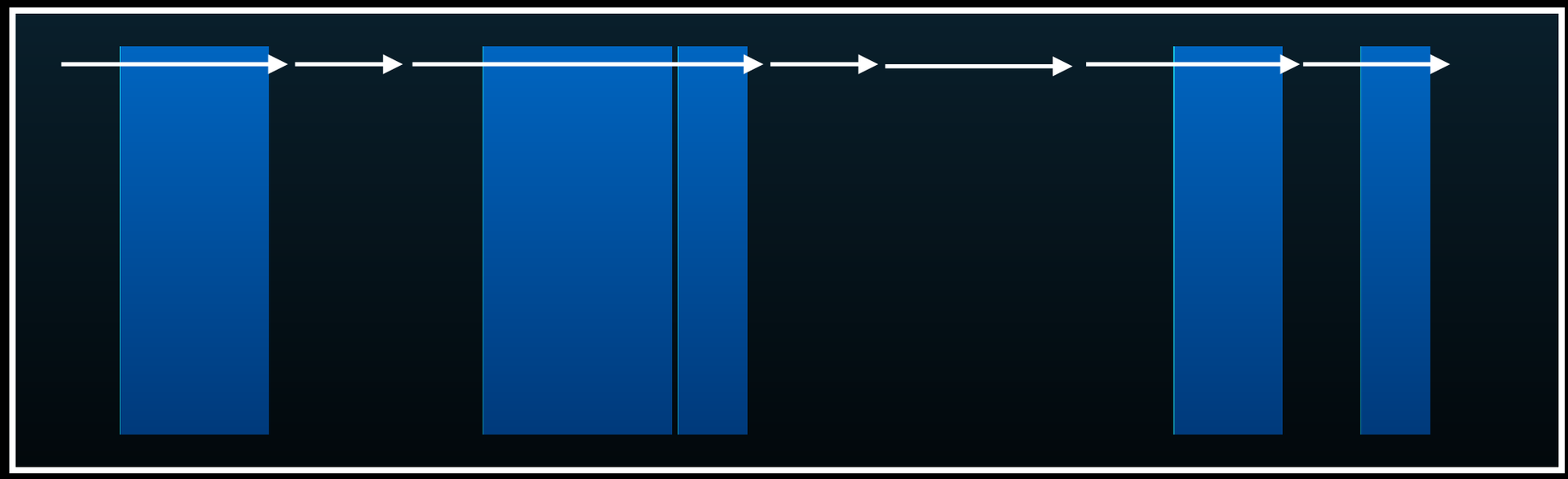
Copying:

Mark-Sweep:

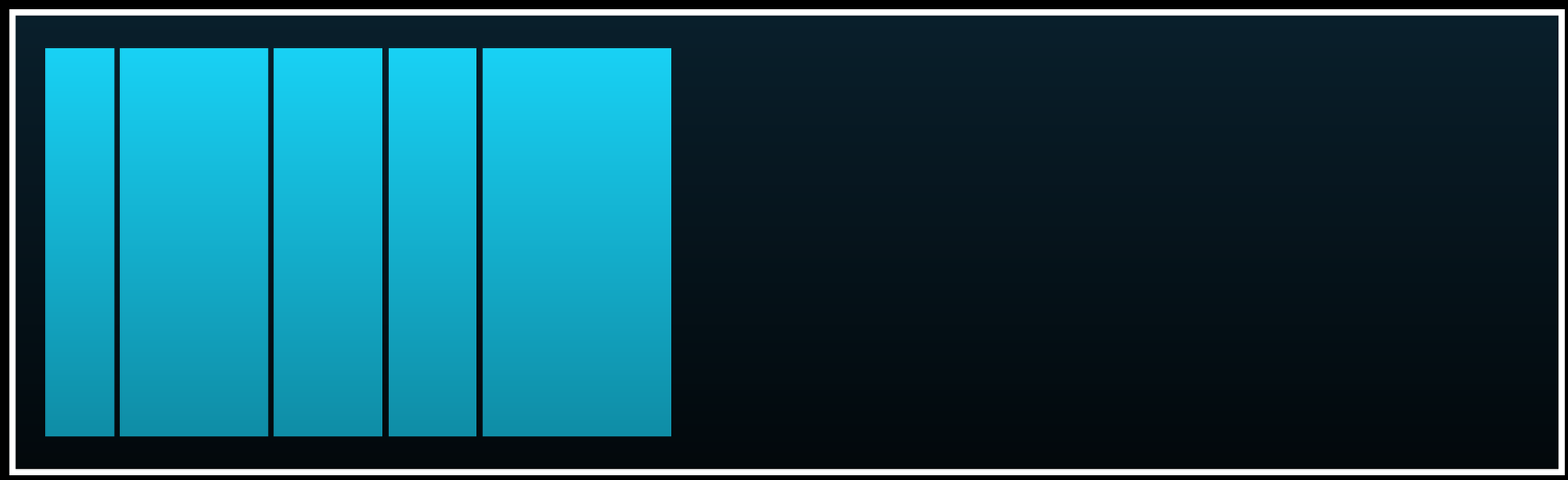


Copying:

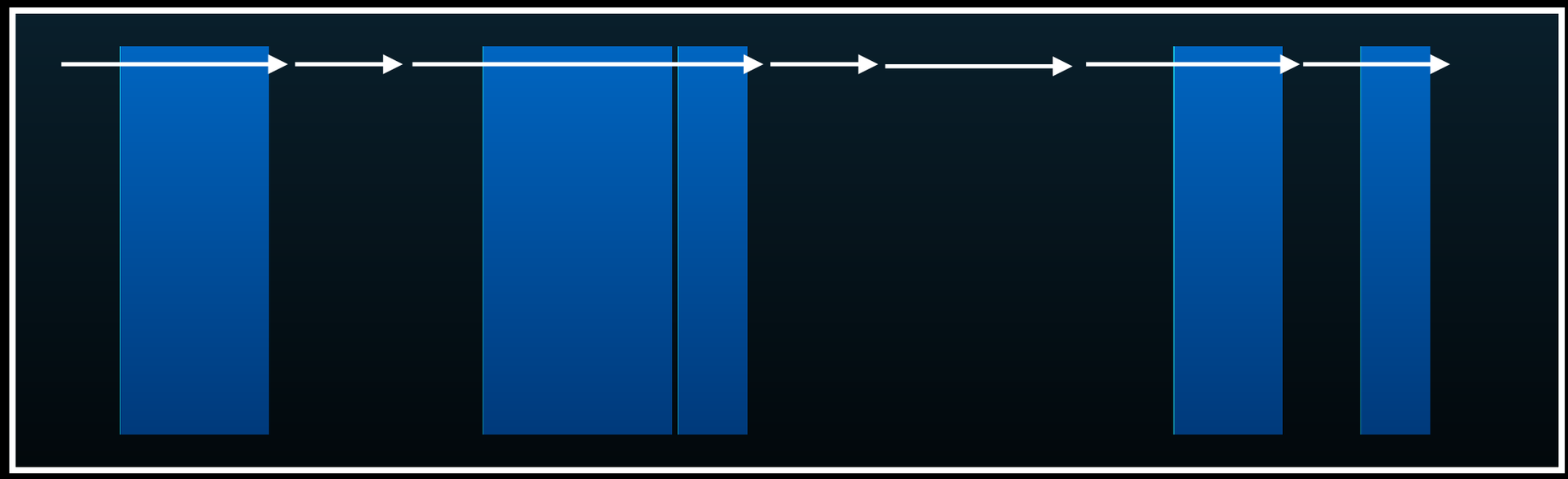
Mark-Sweep:



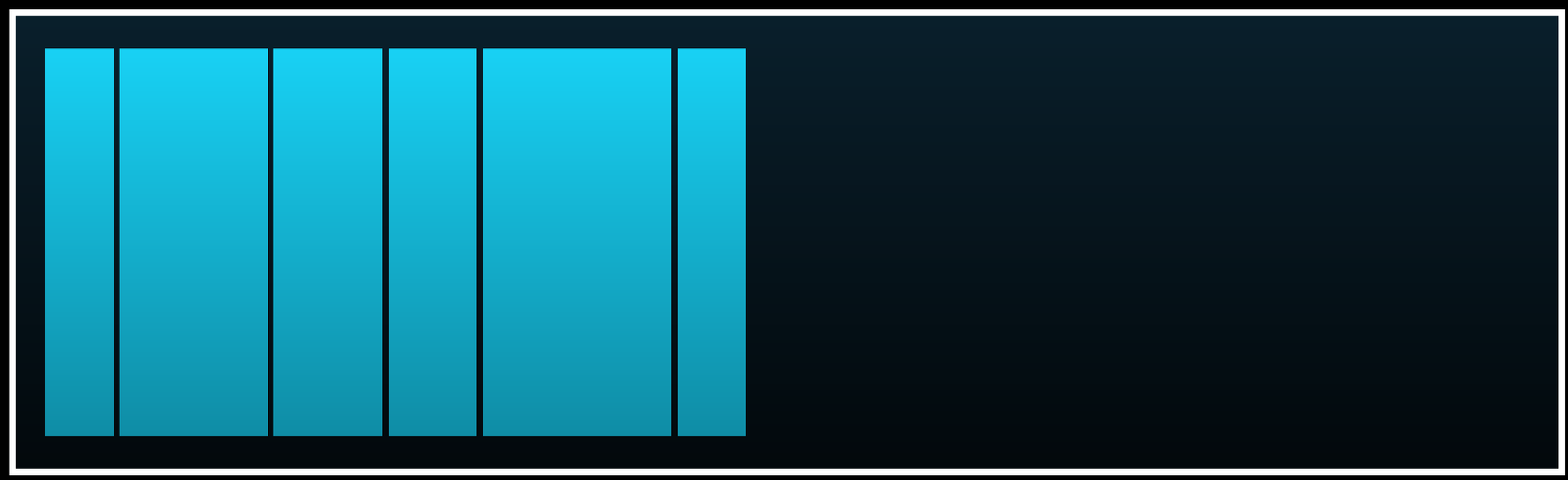
Copying:



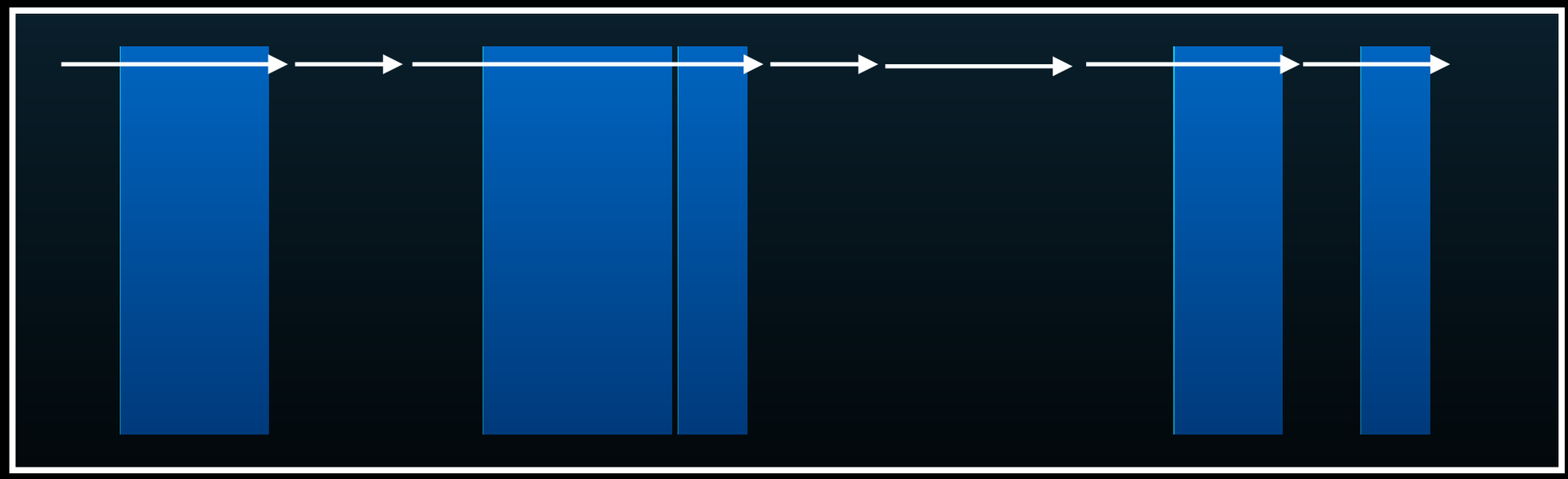
Mark-Sweep:



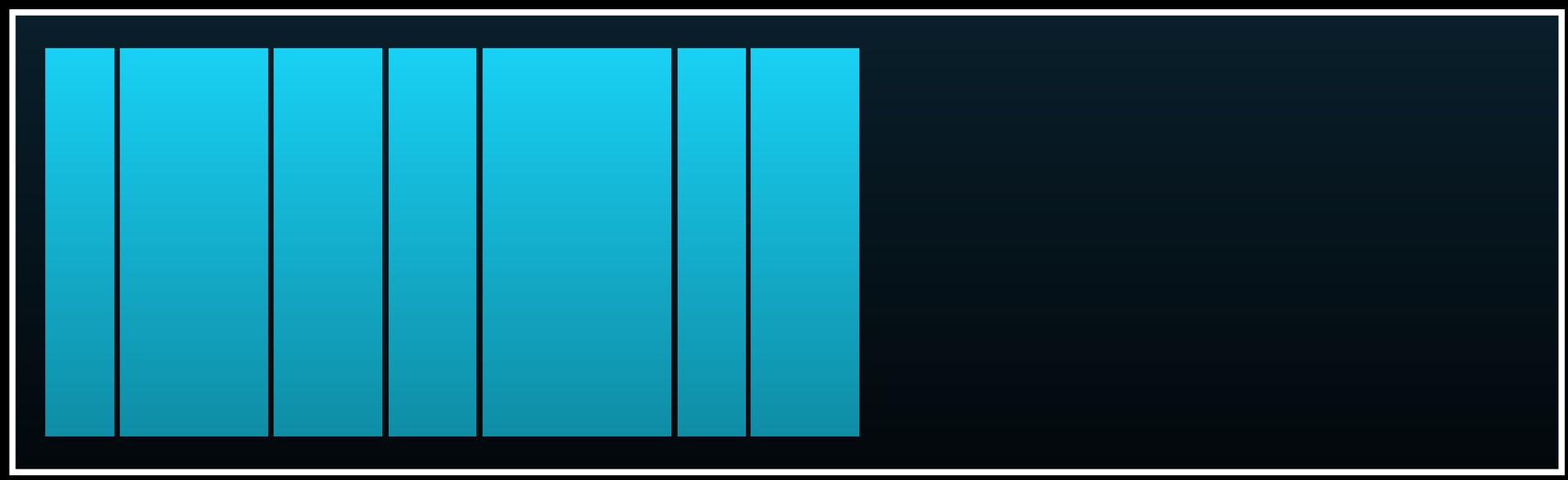
Copying:



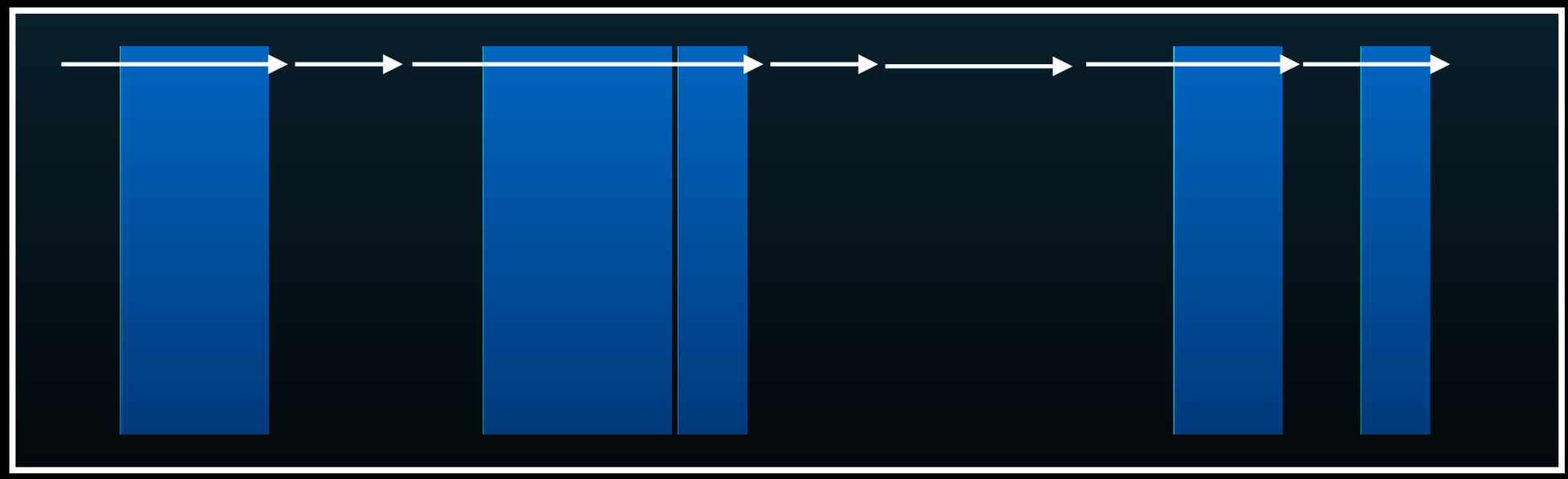
Mark-Sweep:



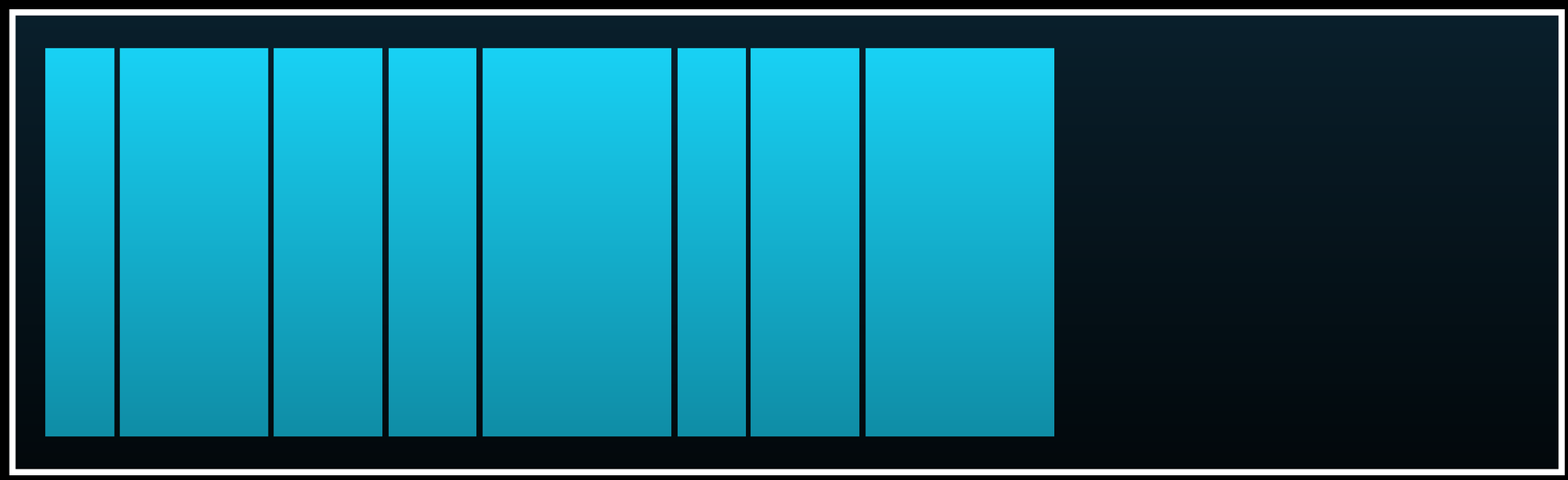
Copying:



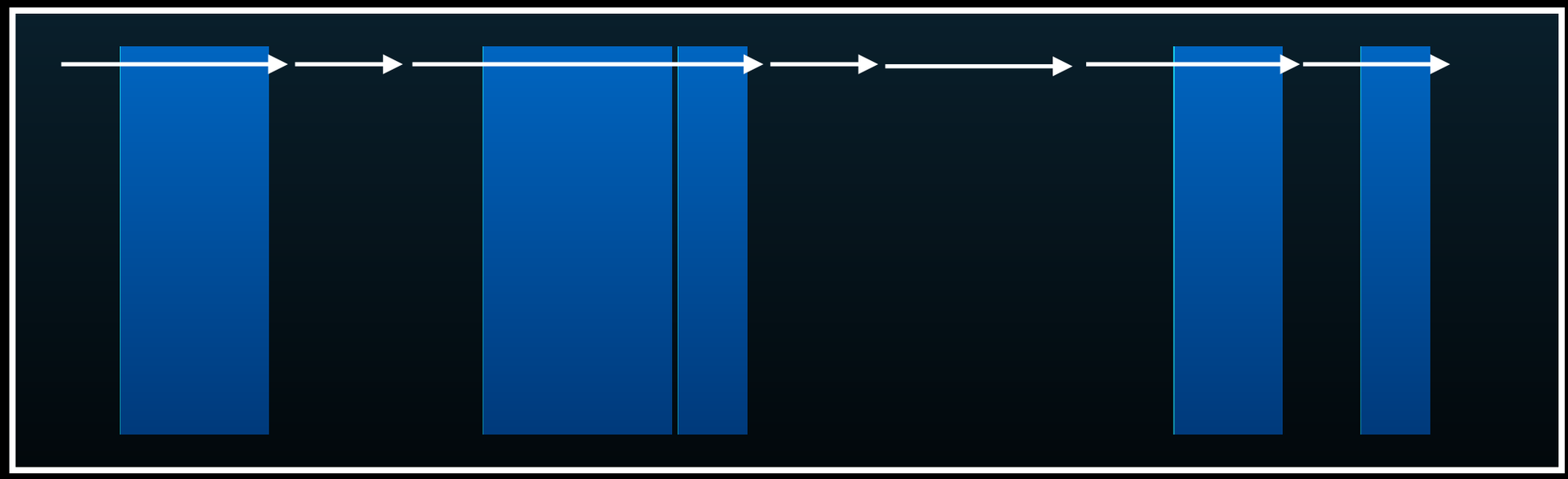
Mark-Sweep:



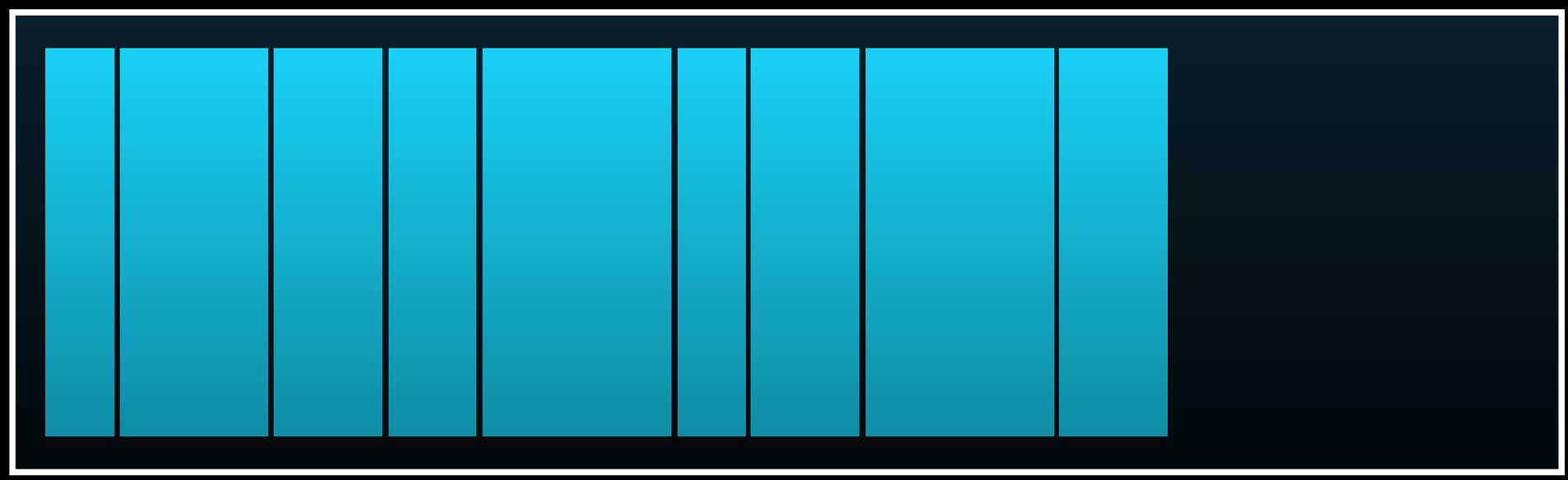
Copying:



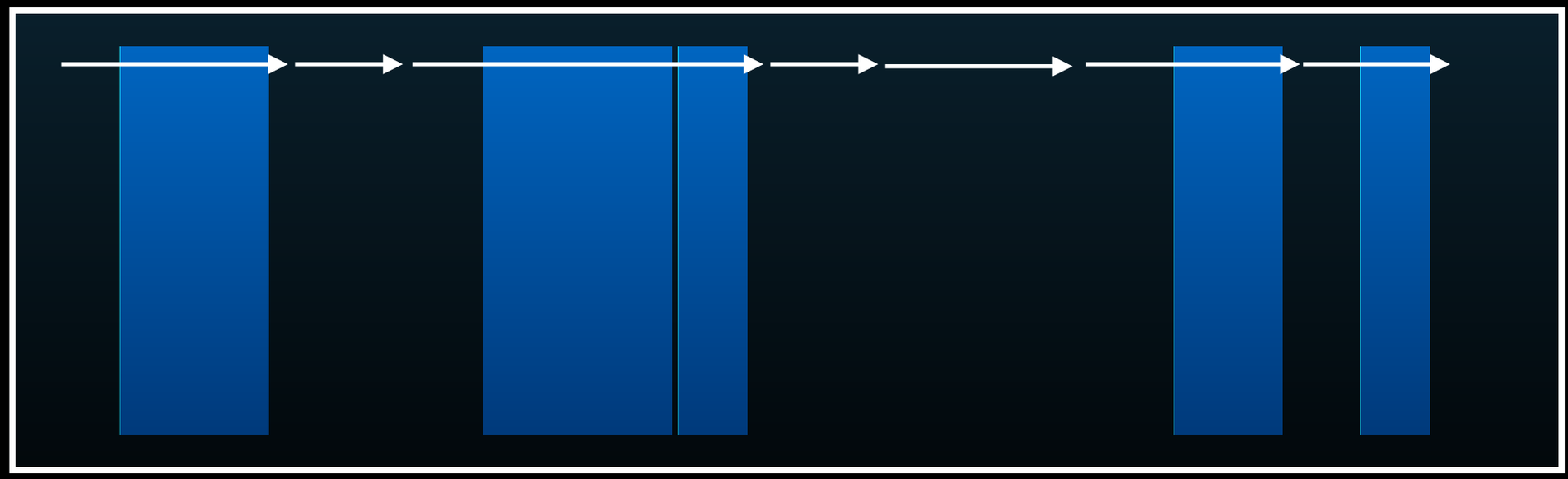
Mark-Sweep:



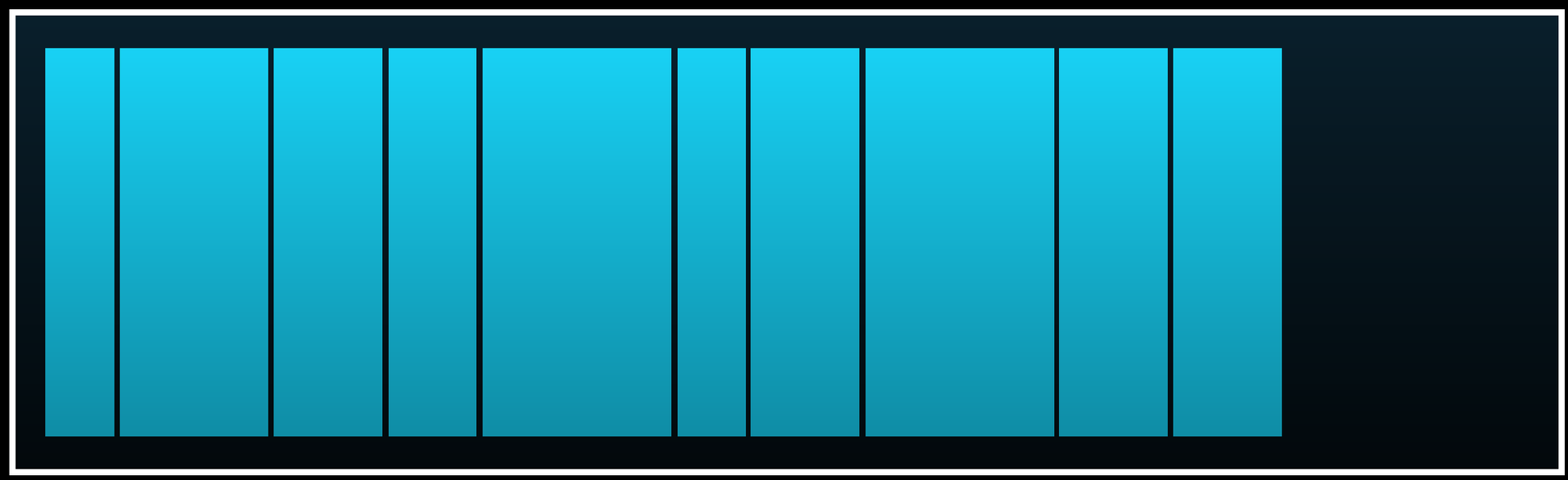
Copying:



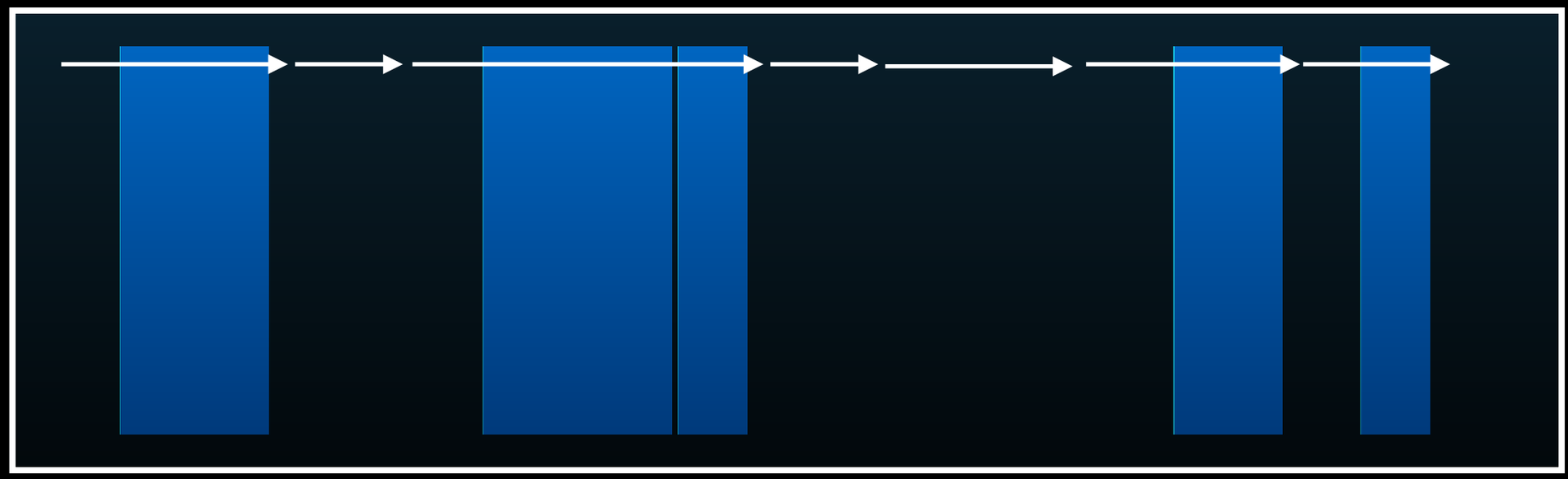
Mark-Sweep:



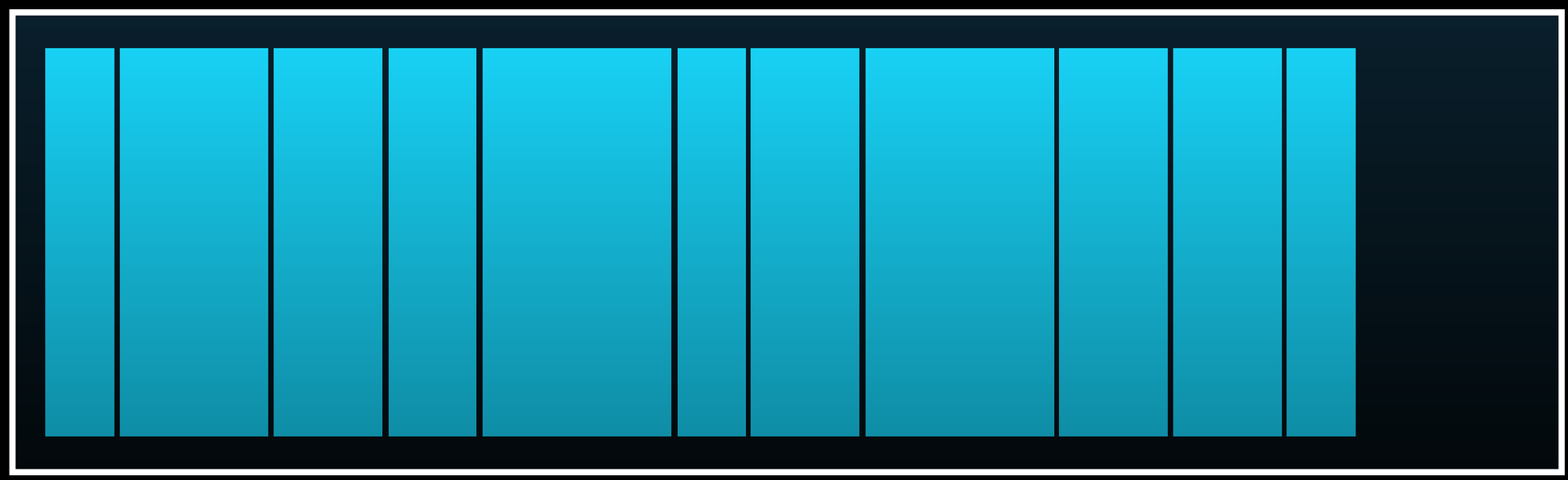
Copying:



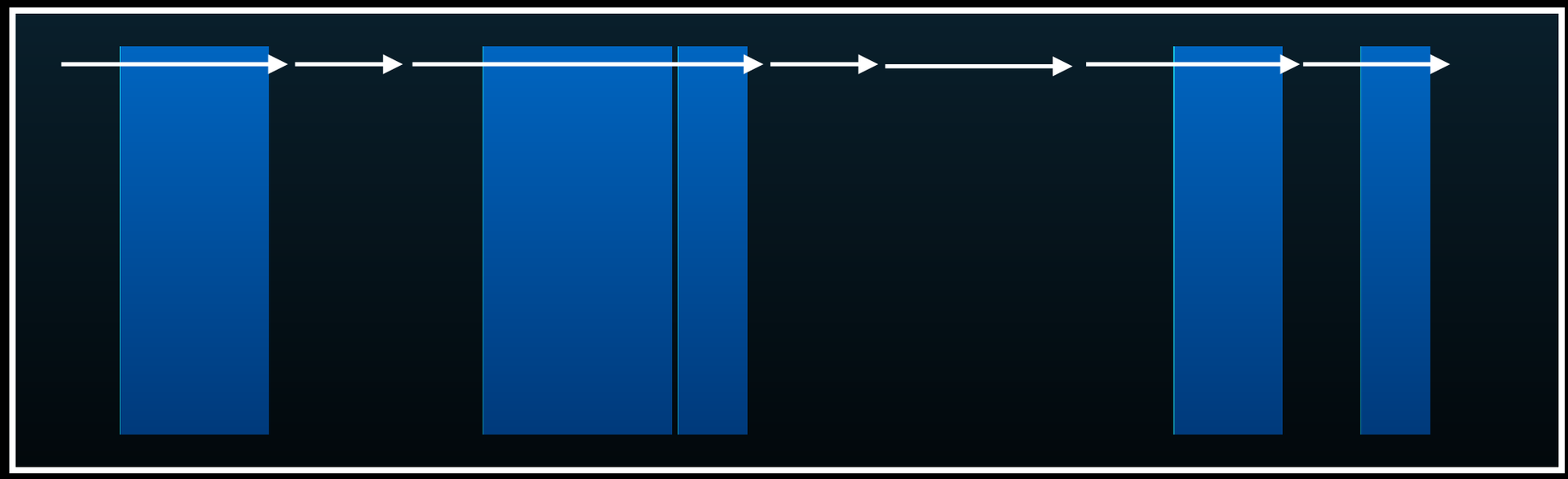
Mark-Sweep:



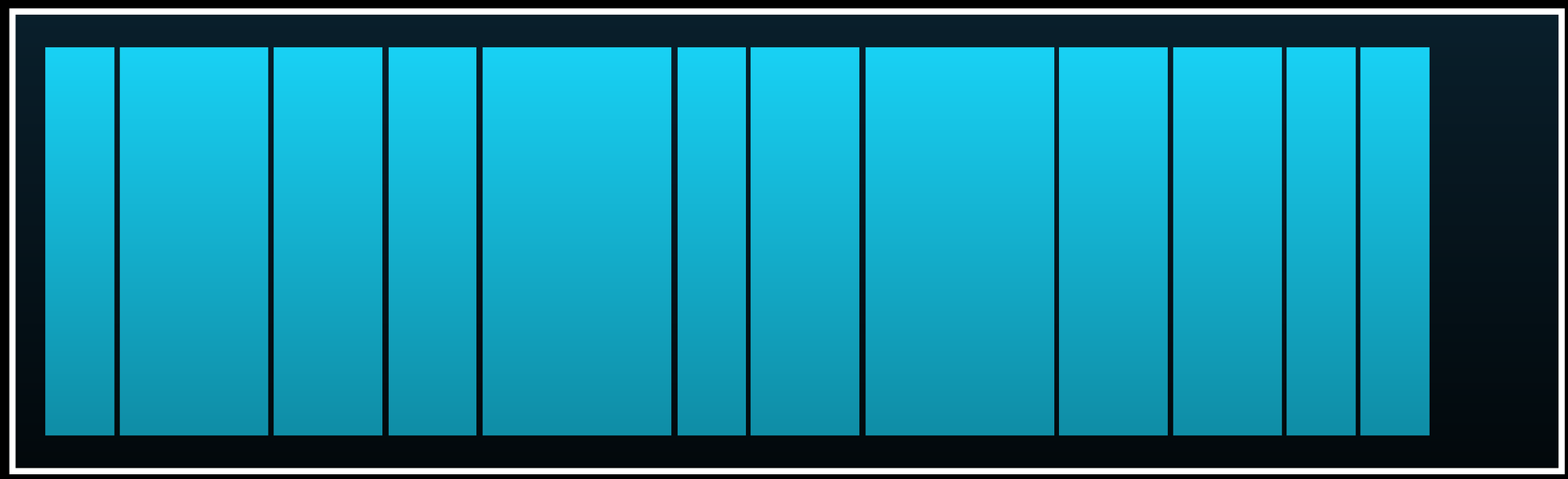
Copying:



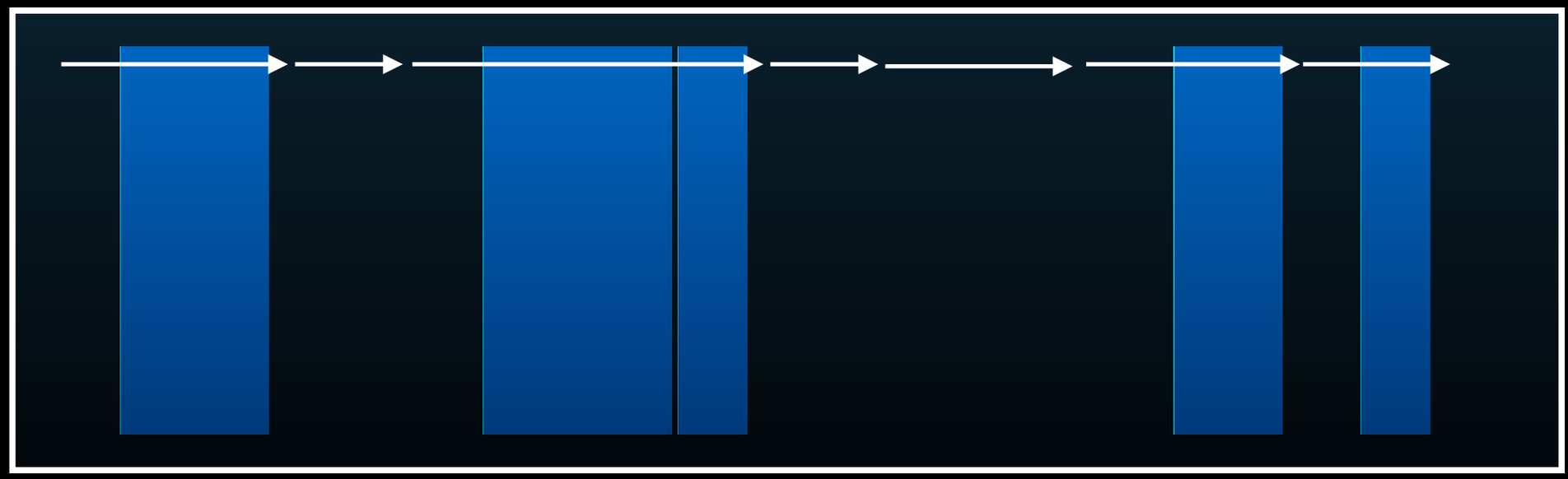
Mark-Sweep:



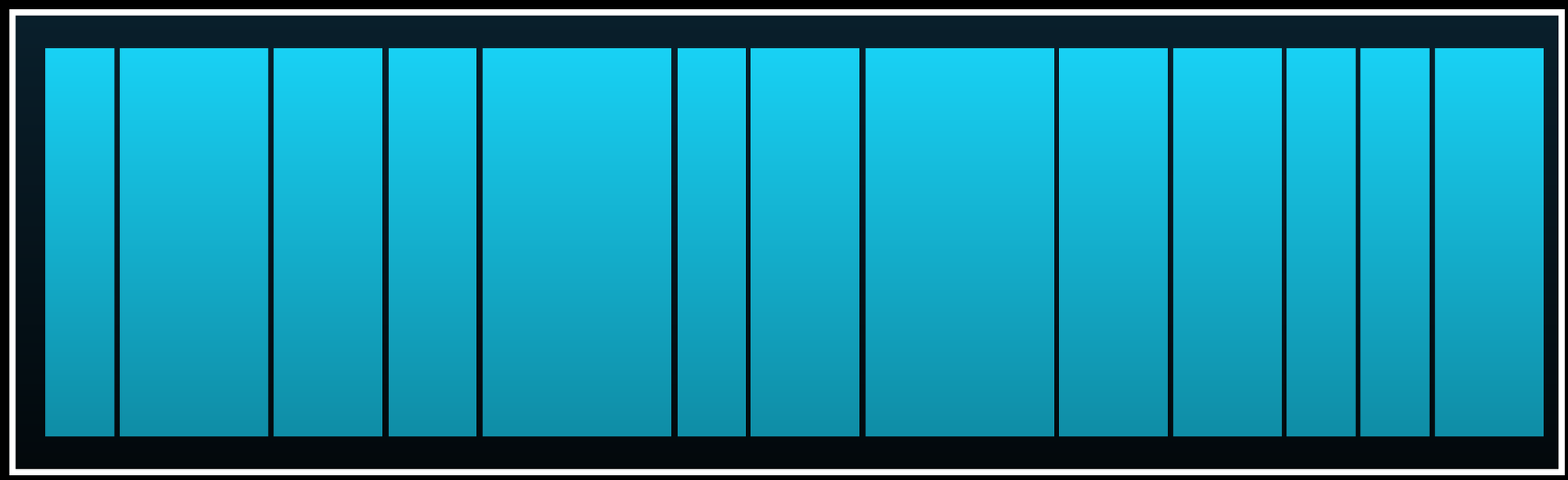
Copying:



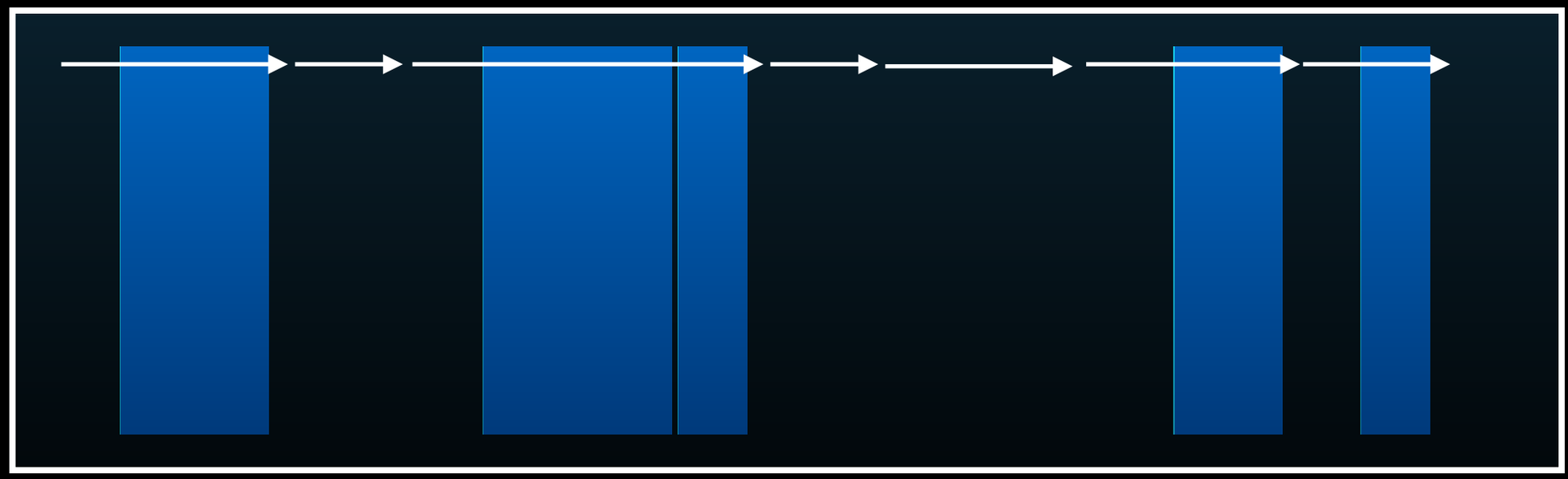
Mark-Sweep:



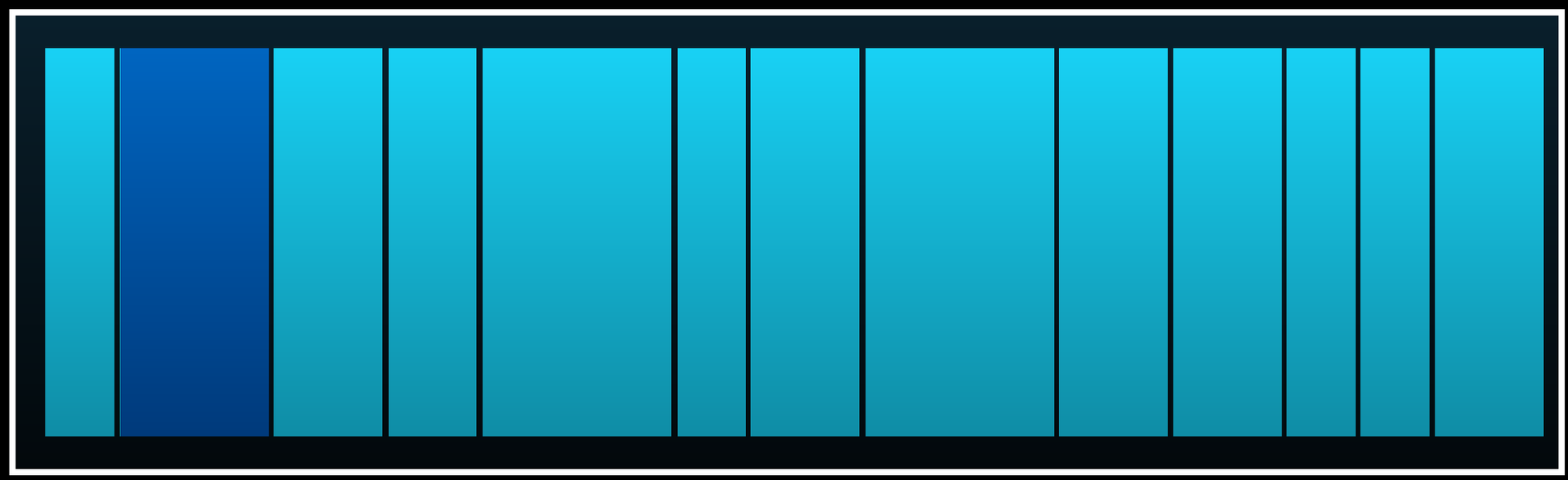
Copying:



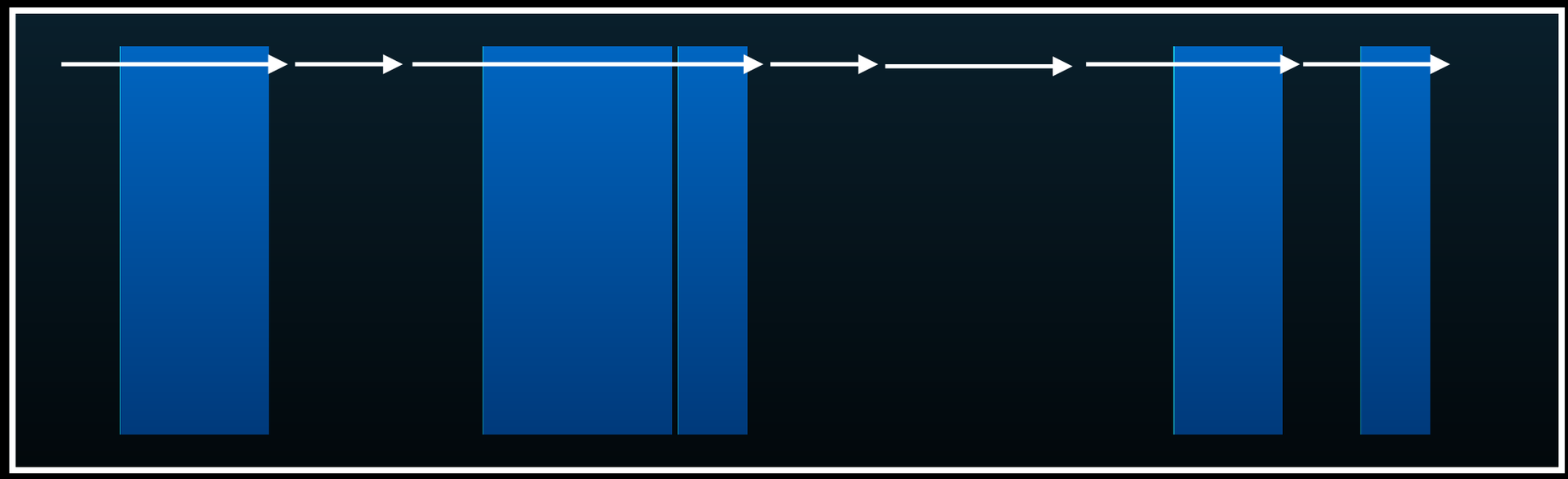
Mark-Sweep:



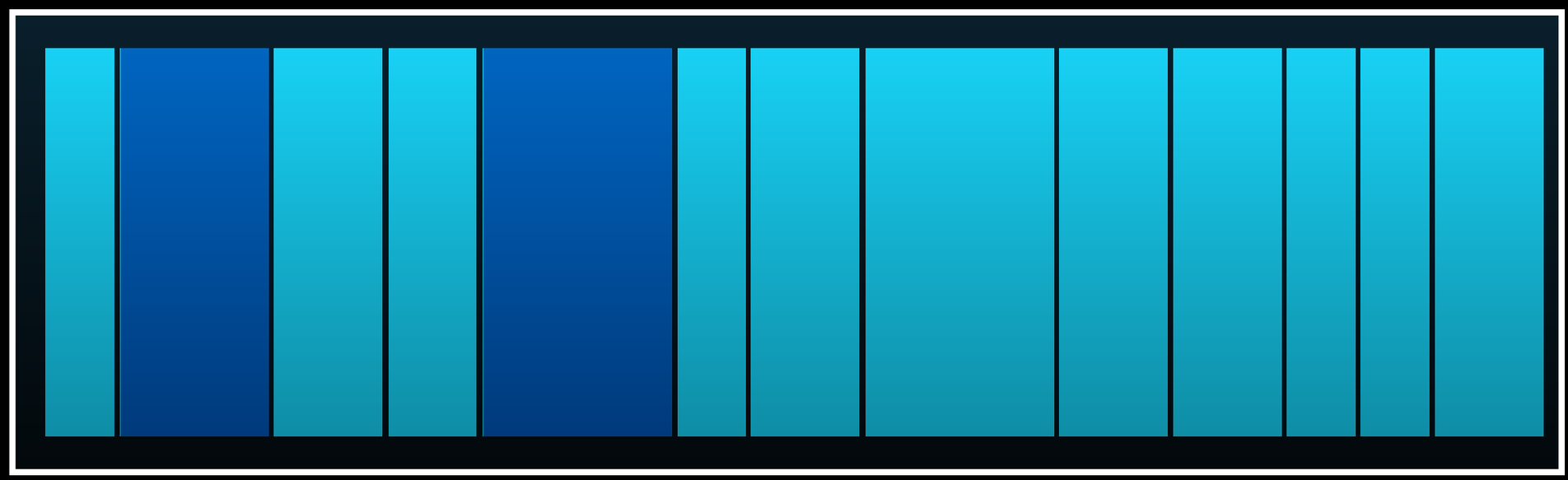
Copying:



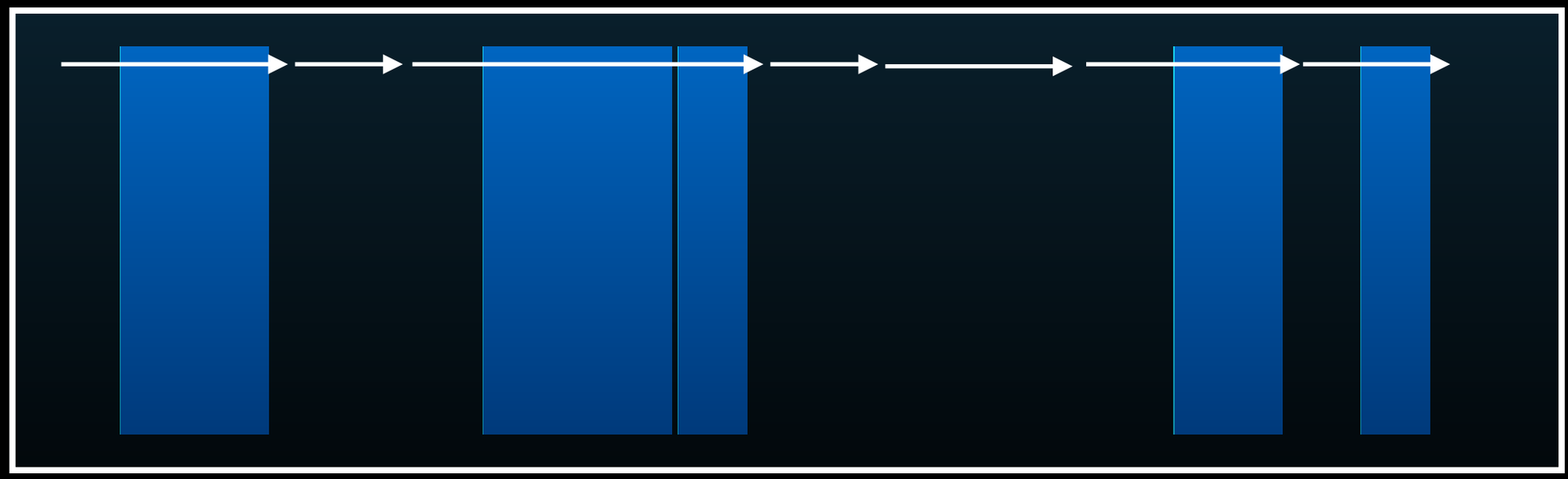
Mark-Sweep:



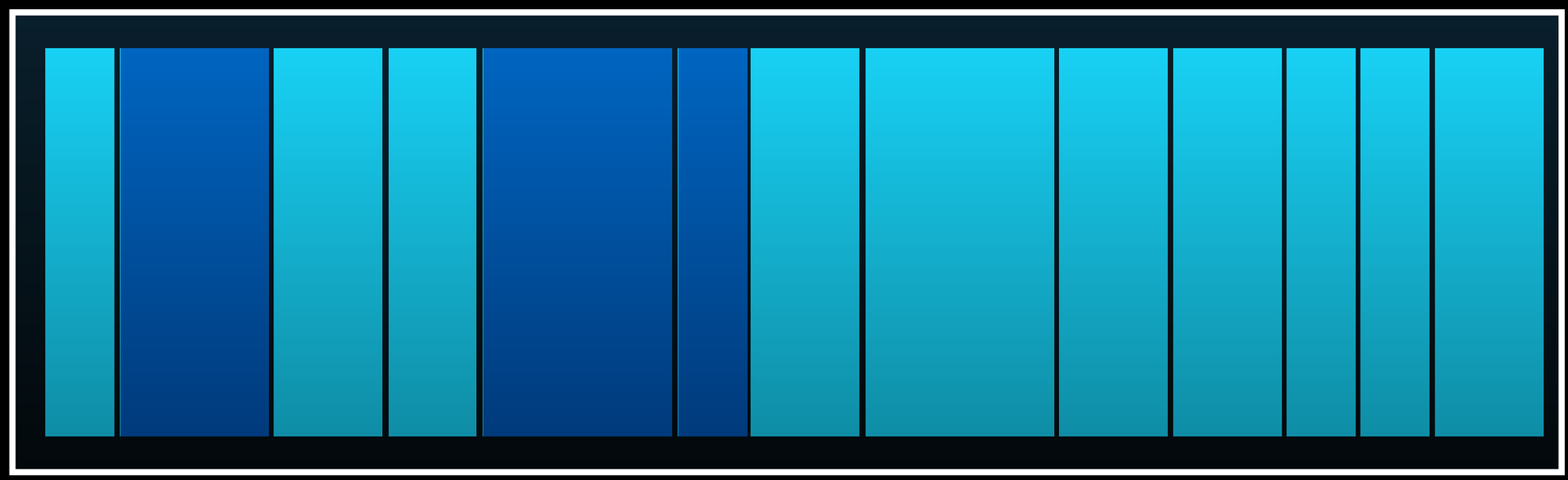
Copying:



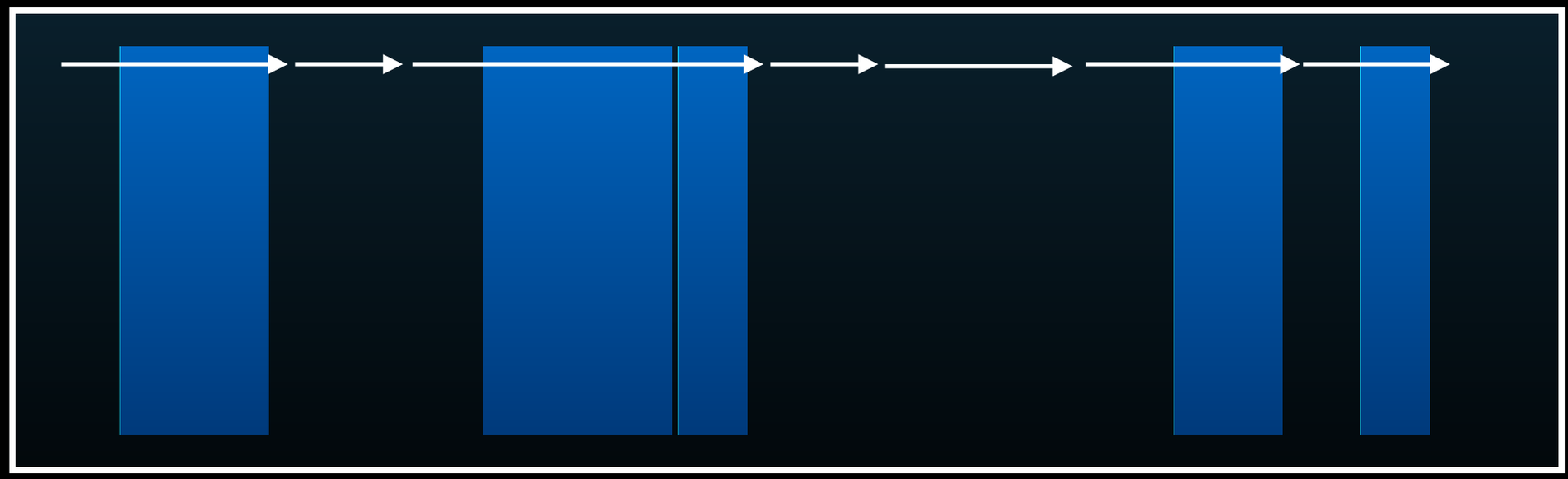
Mark-Sweep:



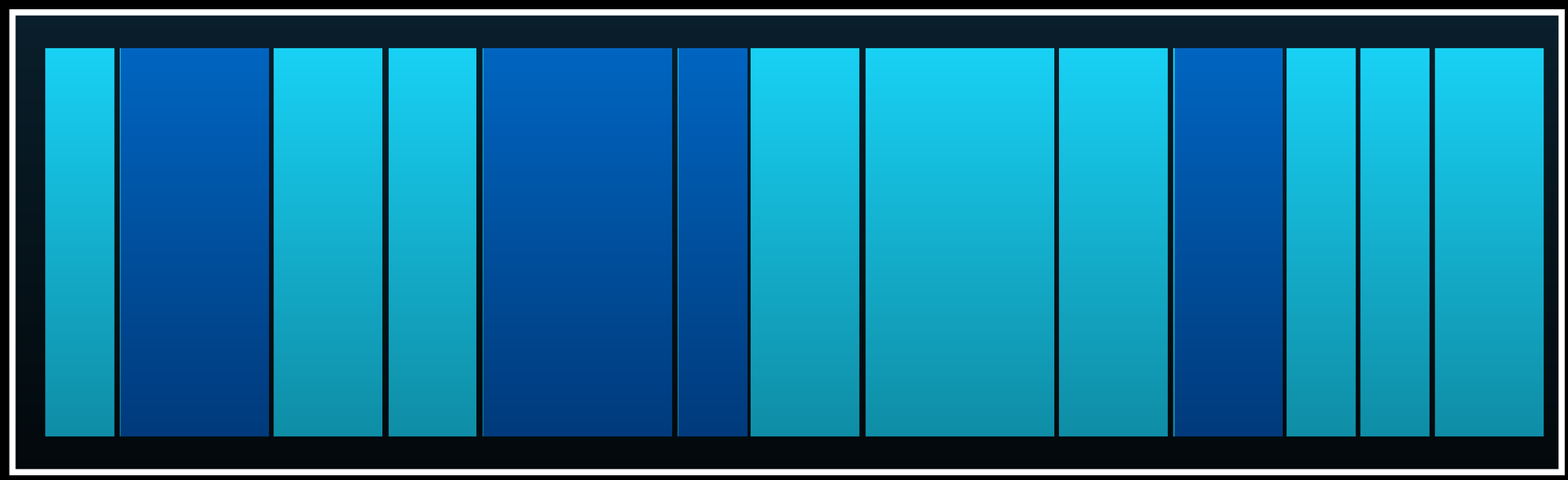
Copying:



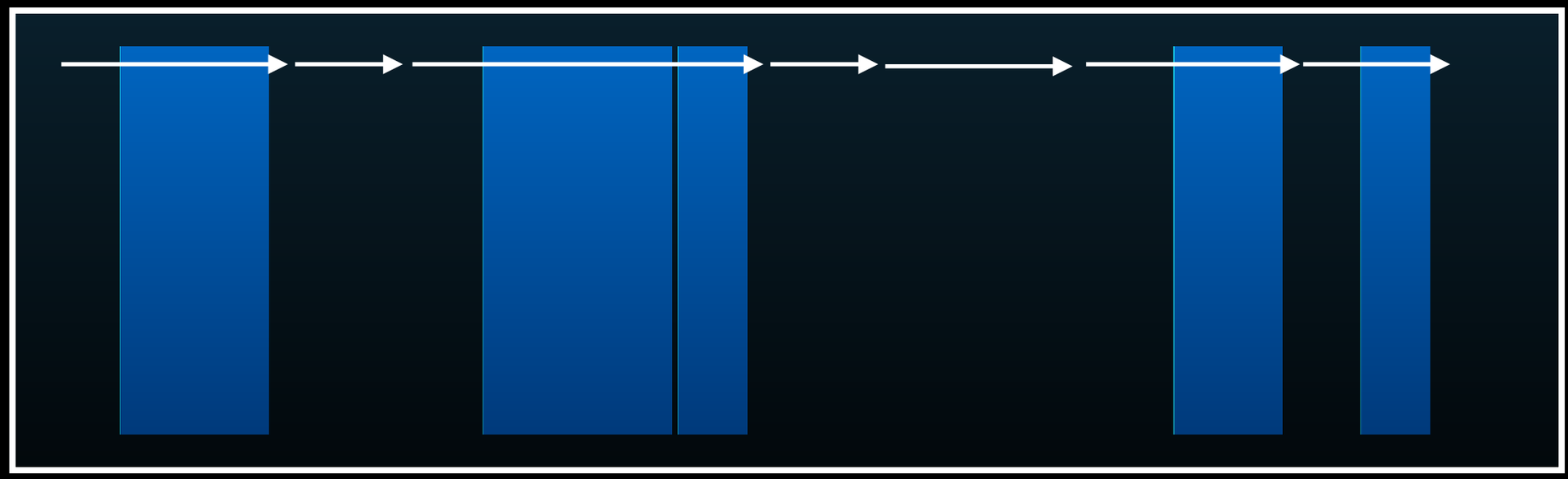
Mark-Sweep:



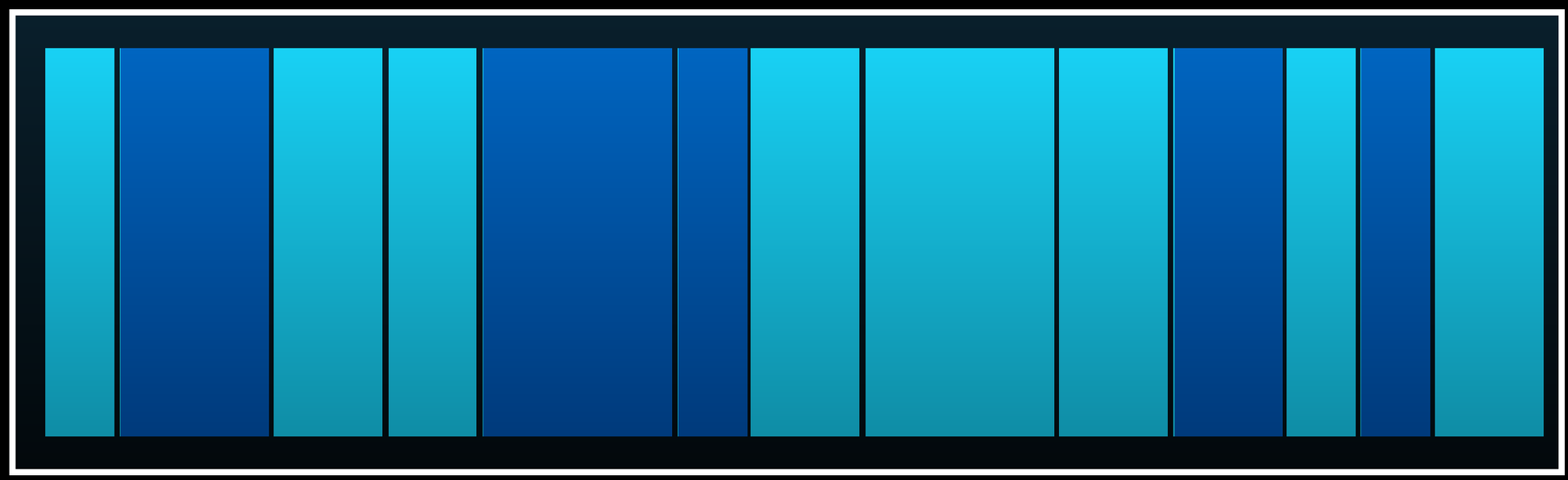
Copying:



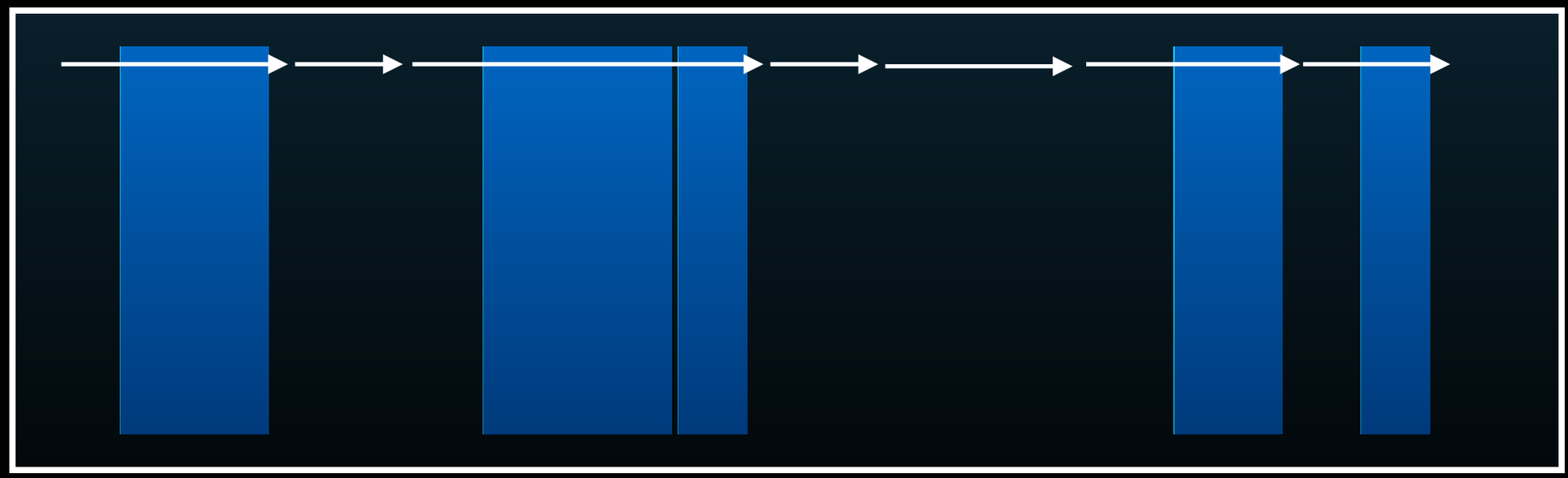
Mark-Sweep:



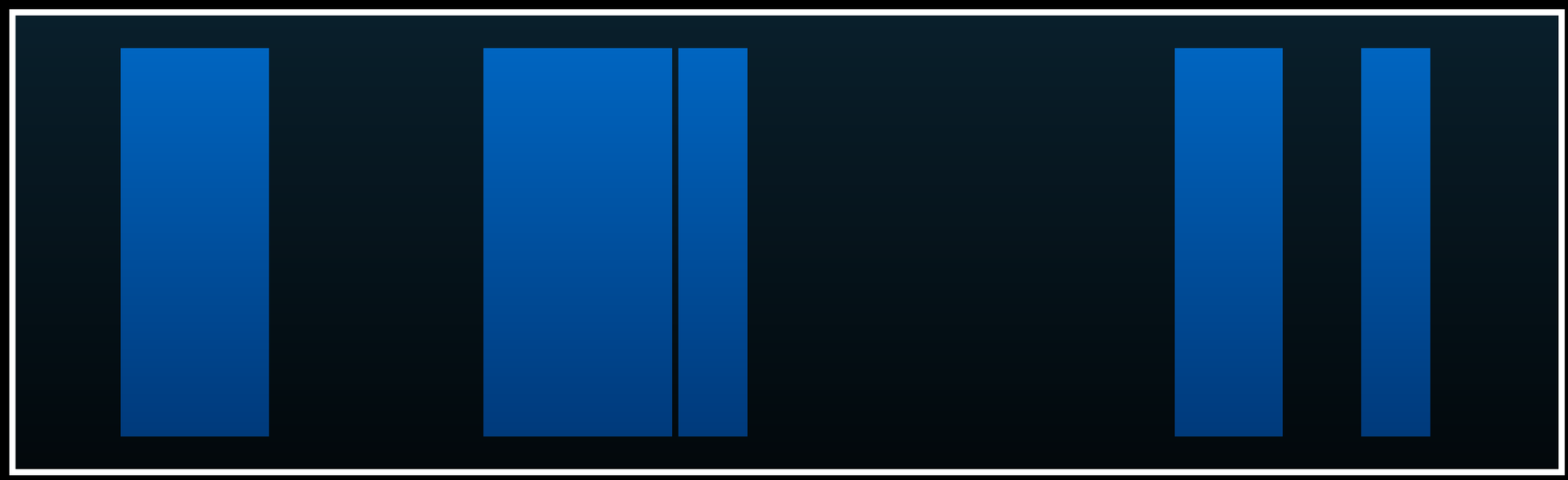
Copying:



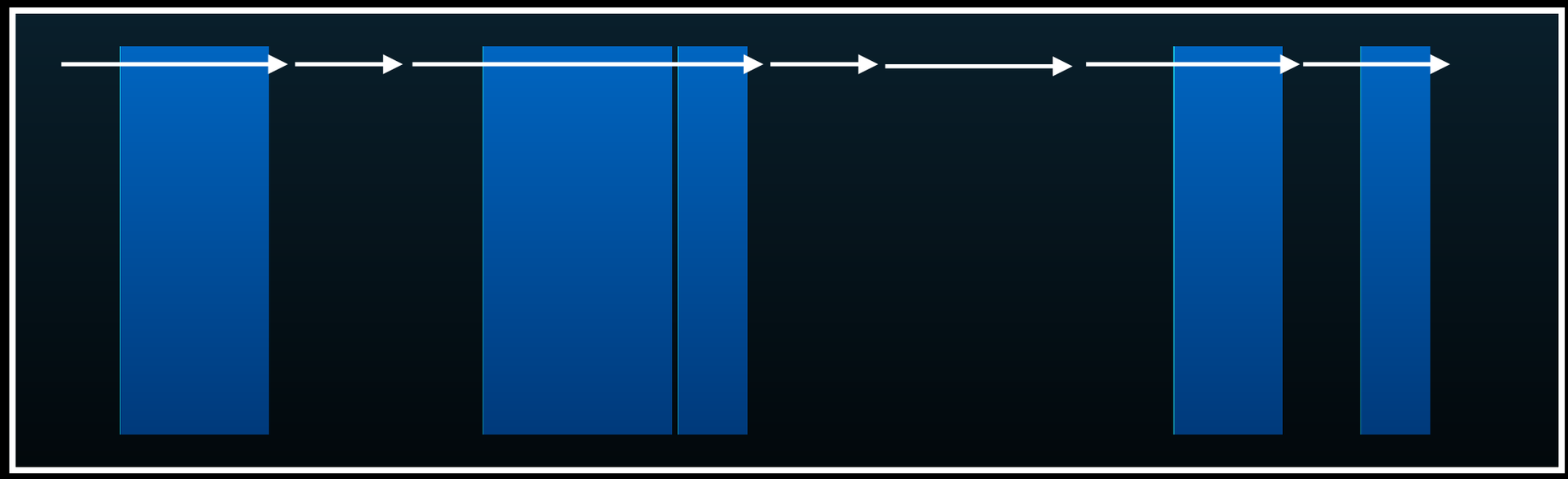
Mark-Sweep:



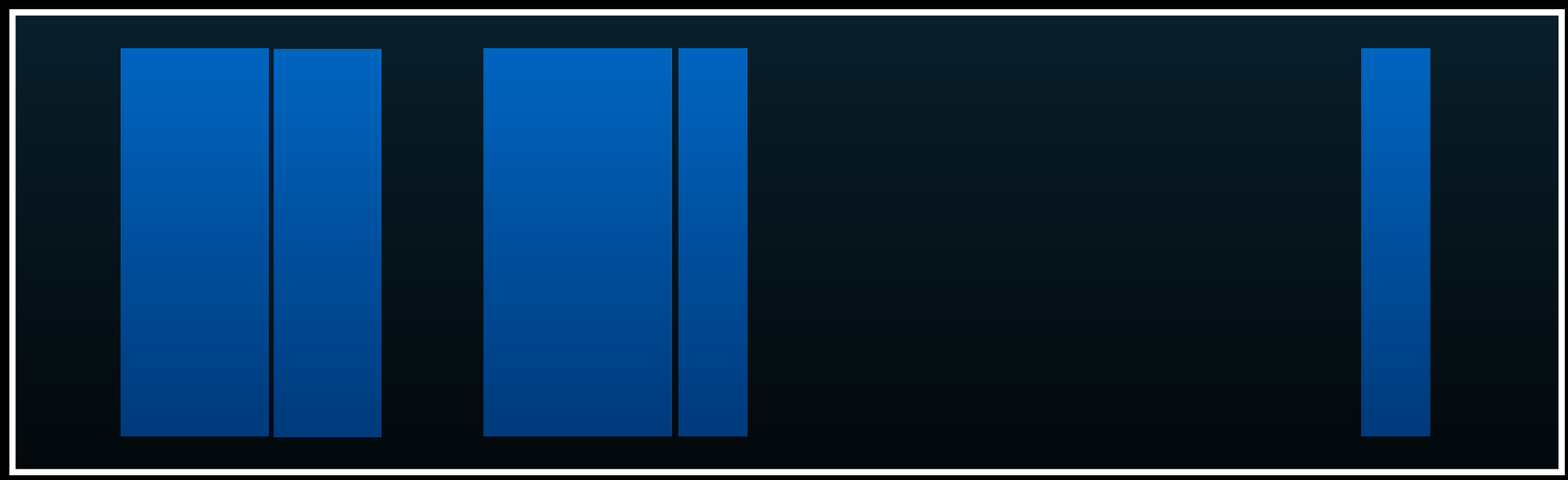
Copying:



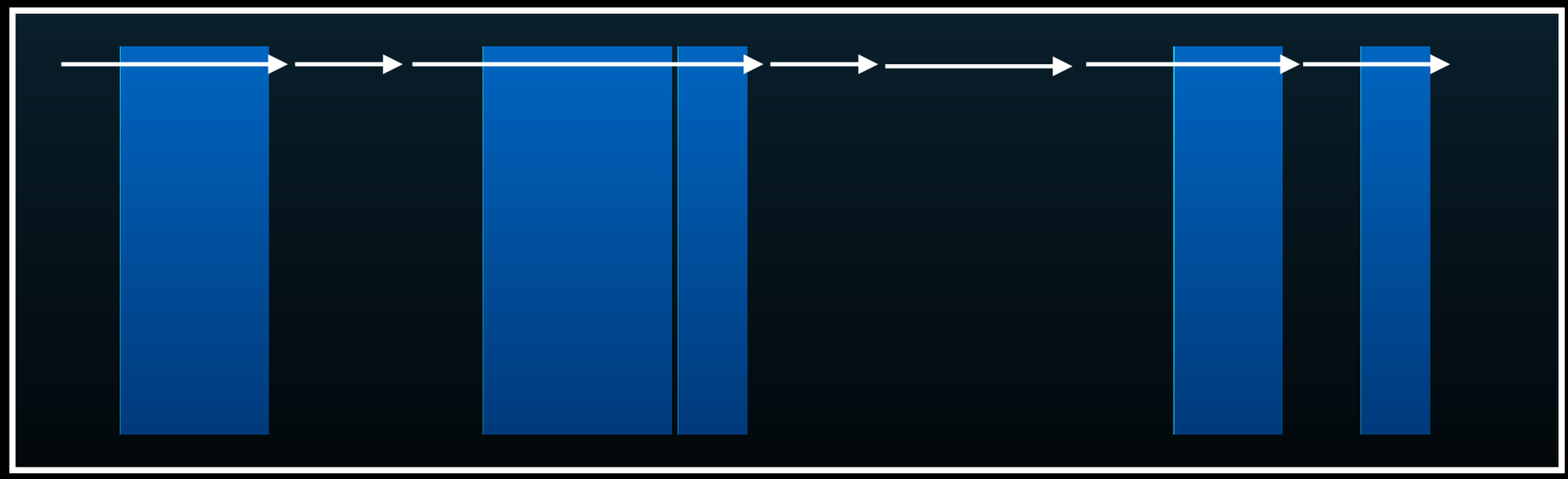
Mark-Sweep:



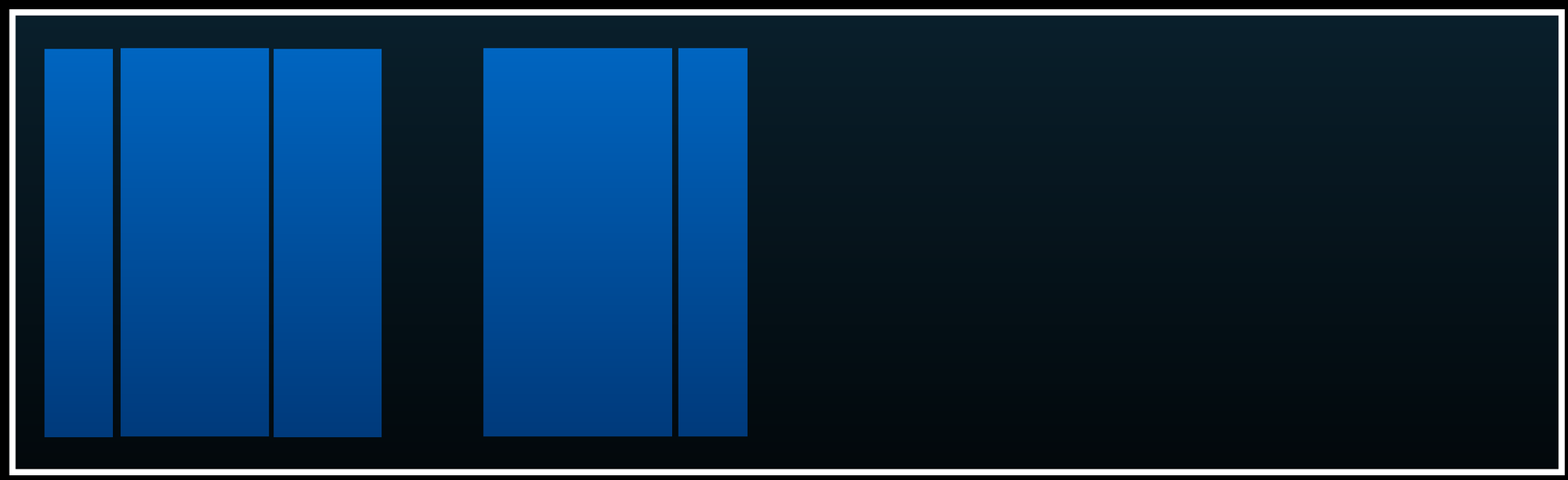
Copying:



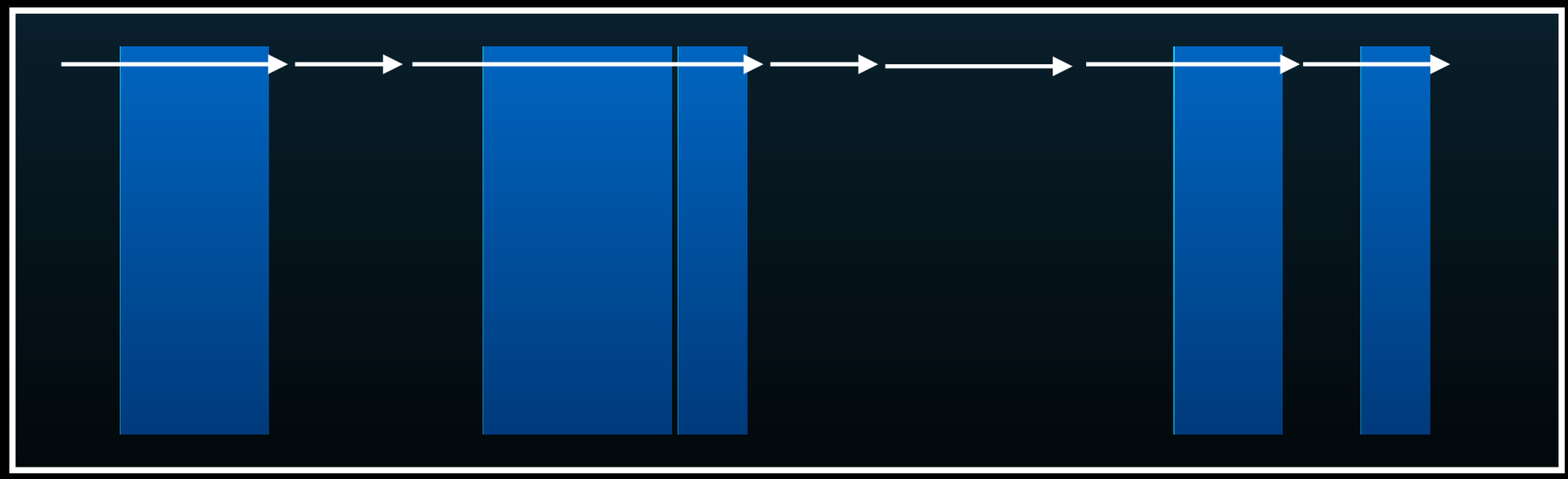
Mark-Sweep:



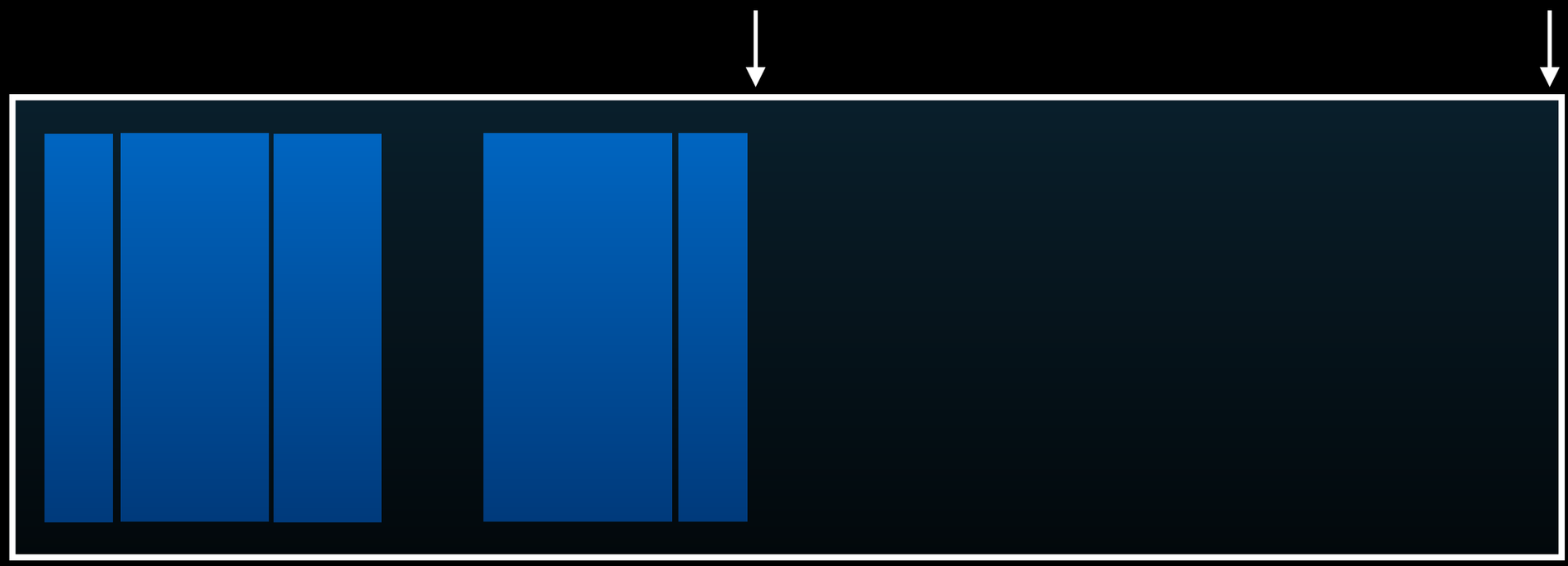
Copying:



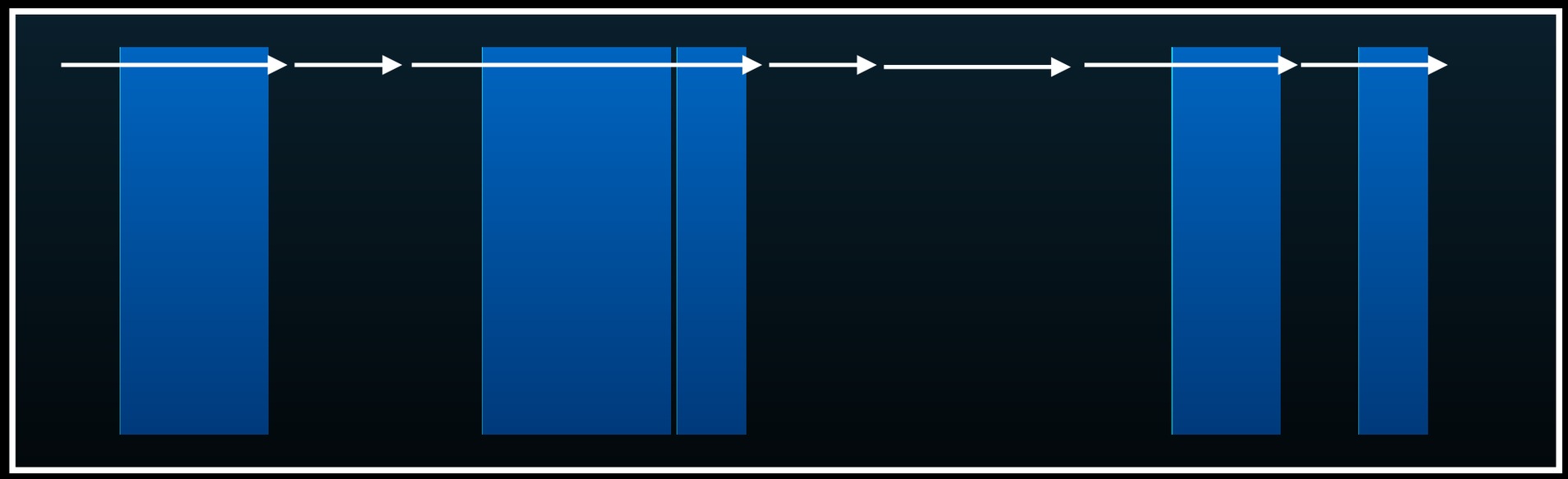
Mark-Sweep:



Copying:

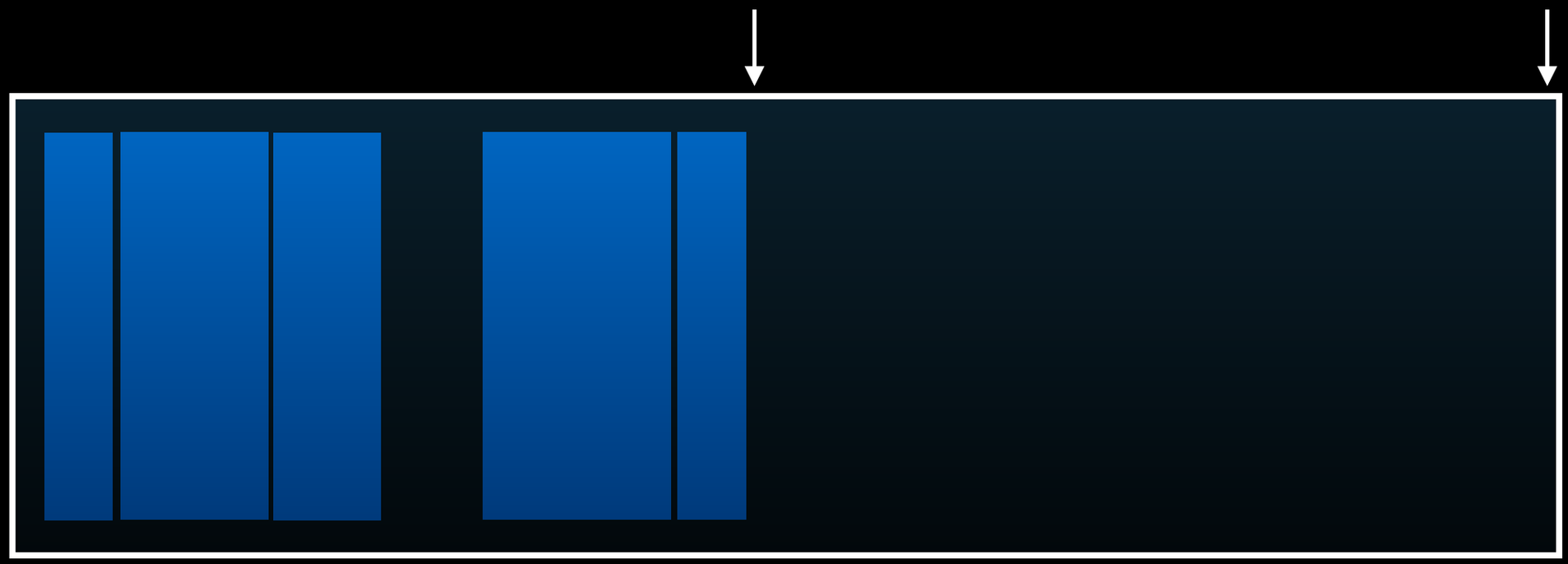


Mark-Sweep:

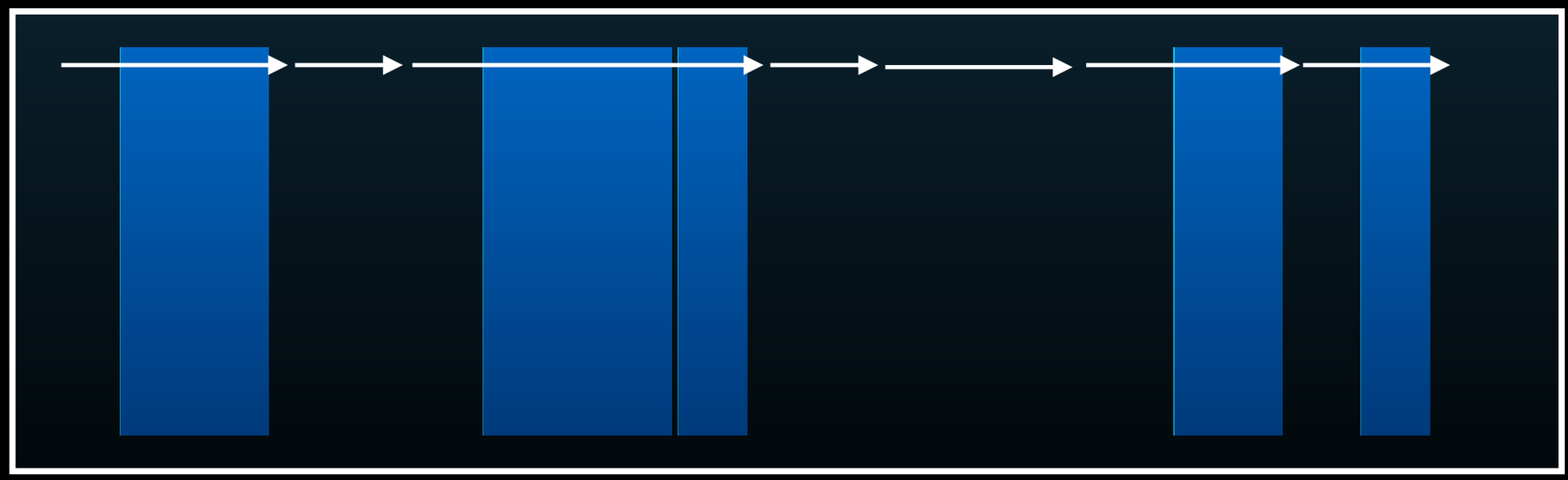


O(#live + #dead)

Copying:

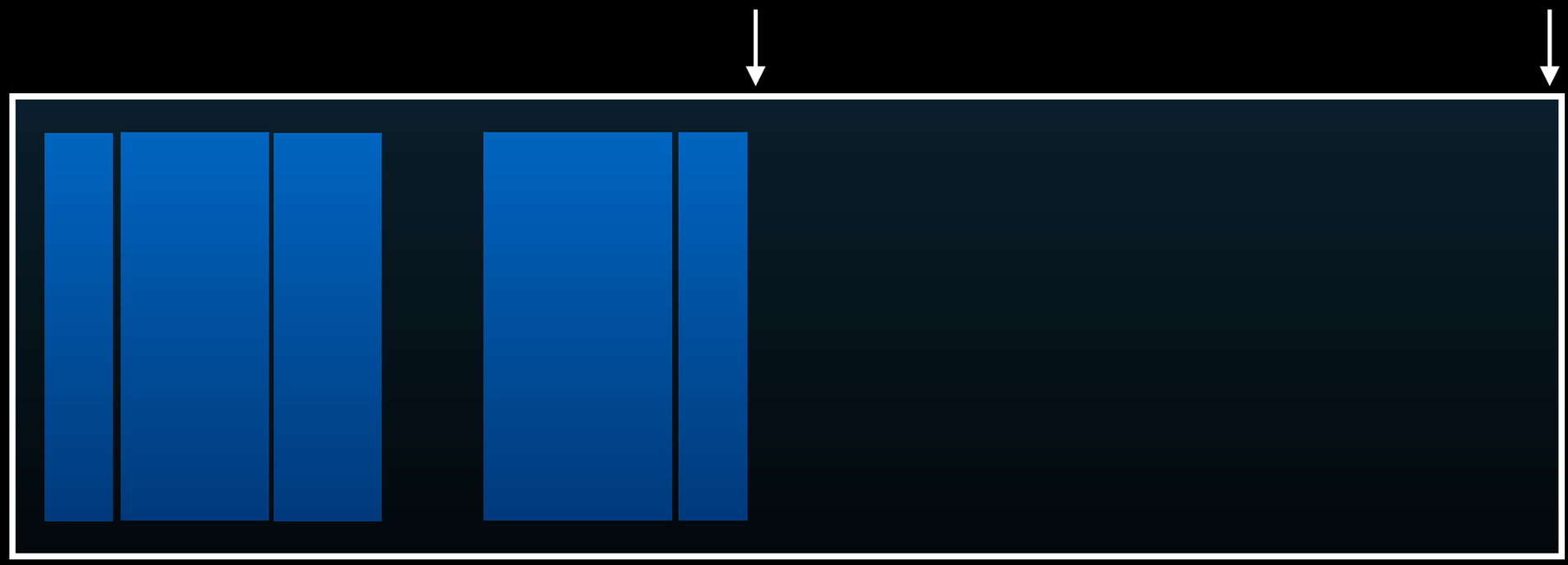


Mark-Sweep:



O(#live + #dead)

Copying:



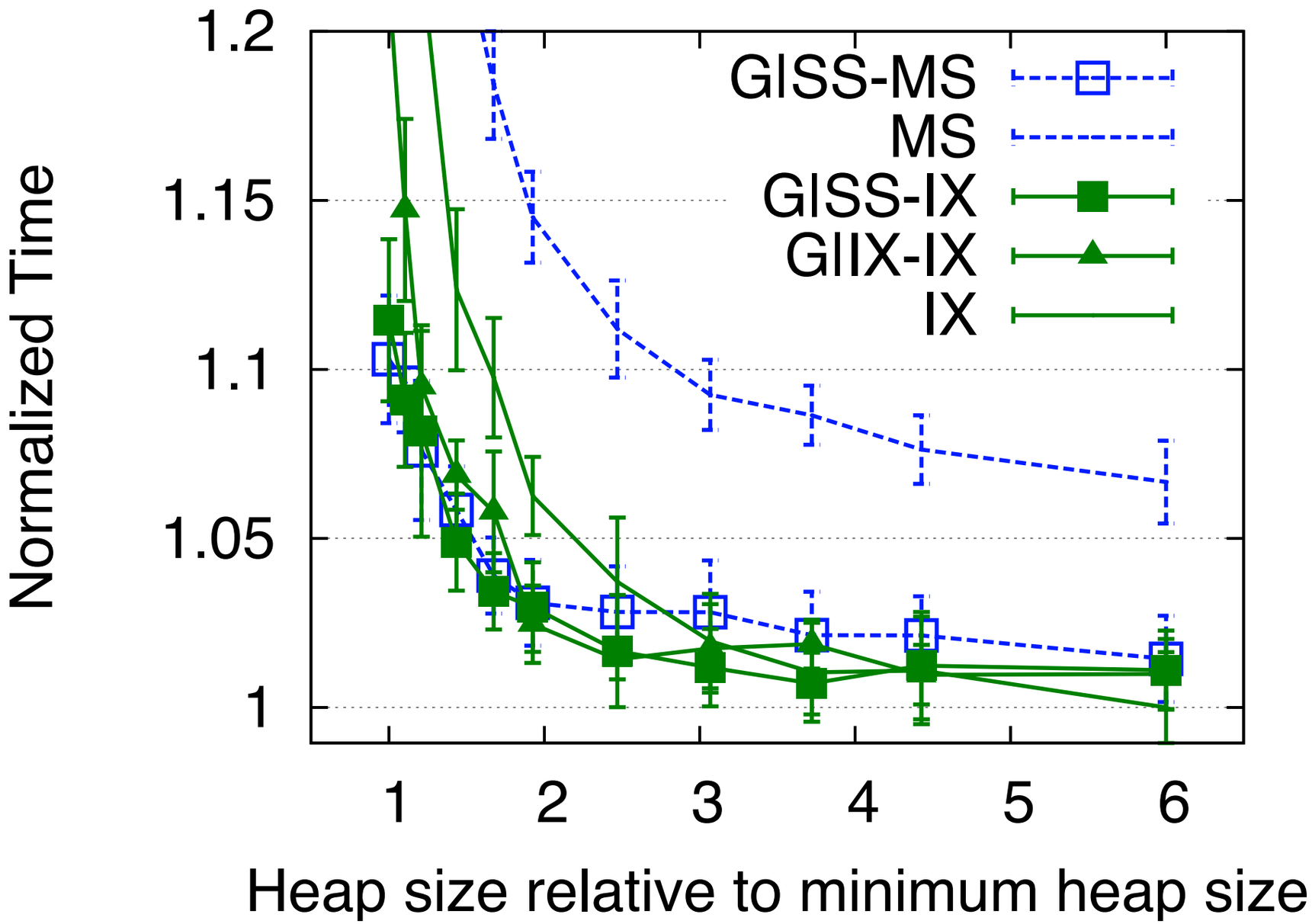
O(#live)

- Copying is an optimization for avoiding sweeping.
- Copying also reduces fragmentation.

- Is copying the only way to avoid sweeping?
- Does copying save space?

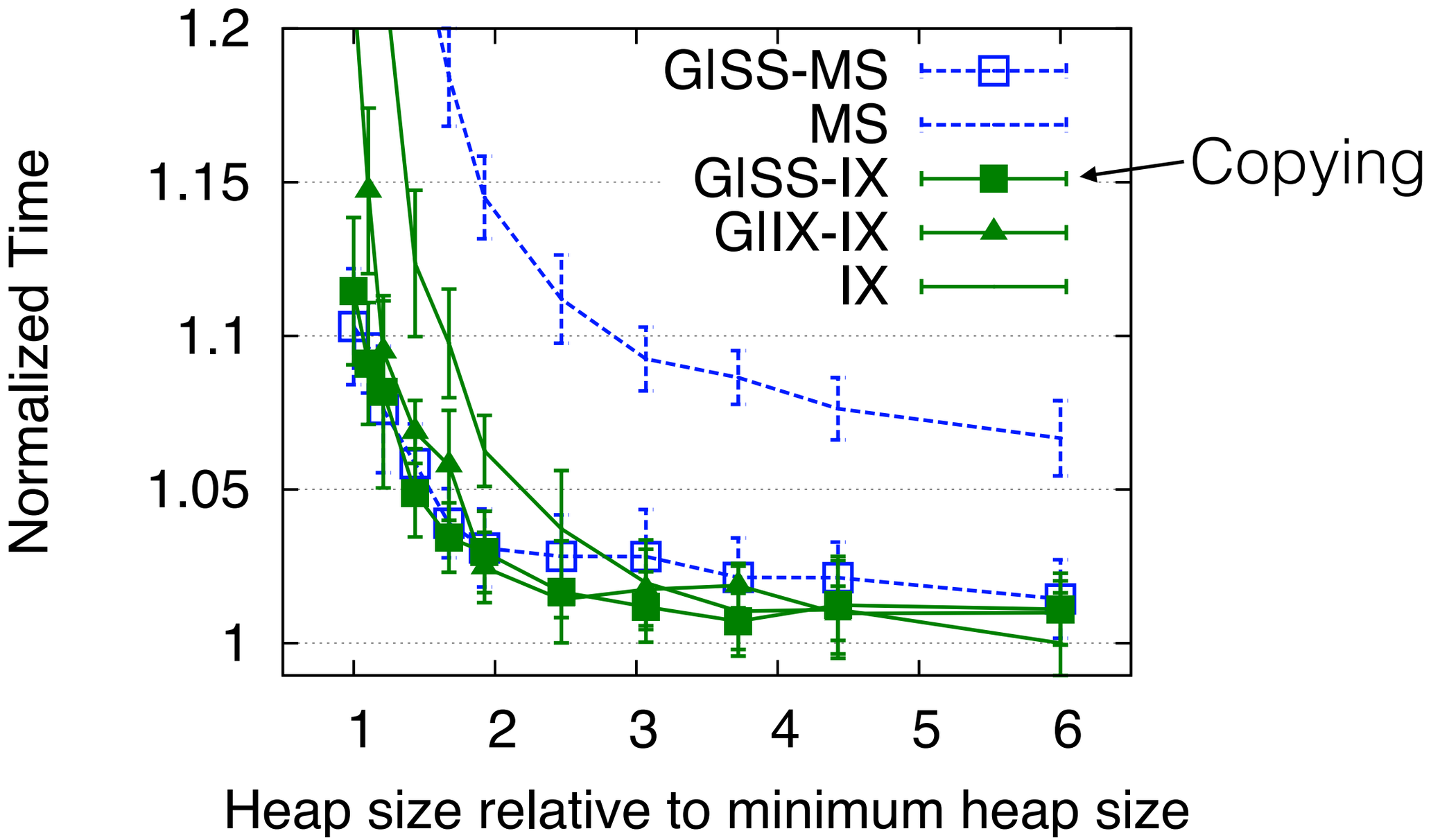
- You can avoid sweeping without copying.
- Find large free regions and bump-allocate in them.
- See:
  - Blackburn and McKinley PLDI'08
  - Berger, Zorn, and McKinley OOPSLA'02

- Fragmentation is not a big problem in practice.
- Assuming you have:
  - 48-bit address space
  - Mitigations



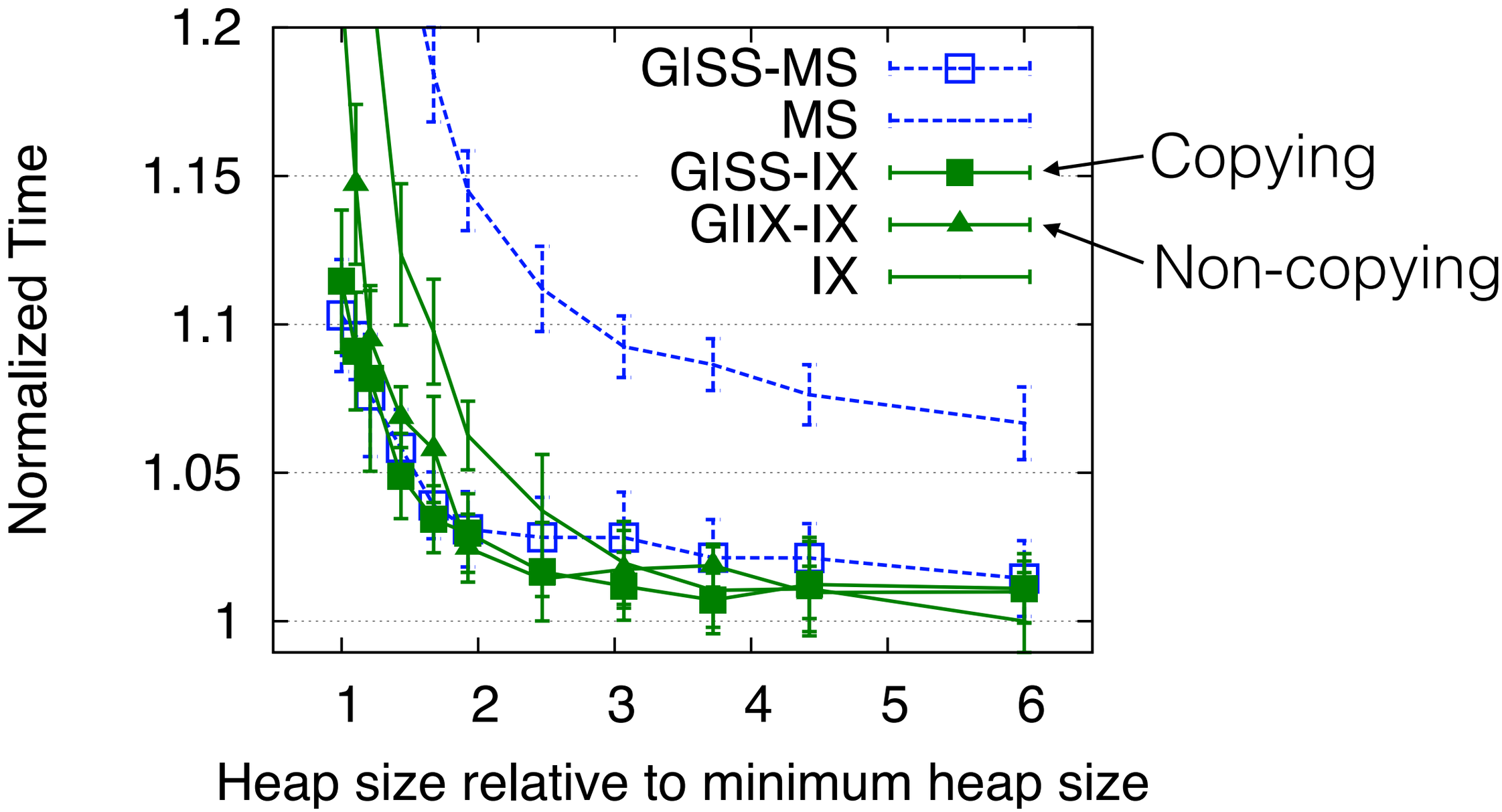
(c) Immix Variants

http://www.cs.utexas.edu/users/speedway/DaCapo/papers/immix-pldi-2008.pdf



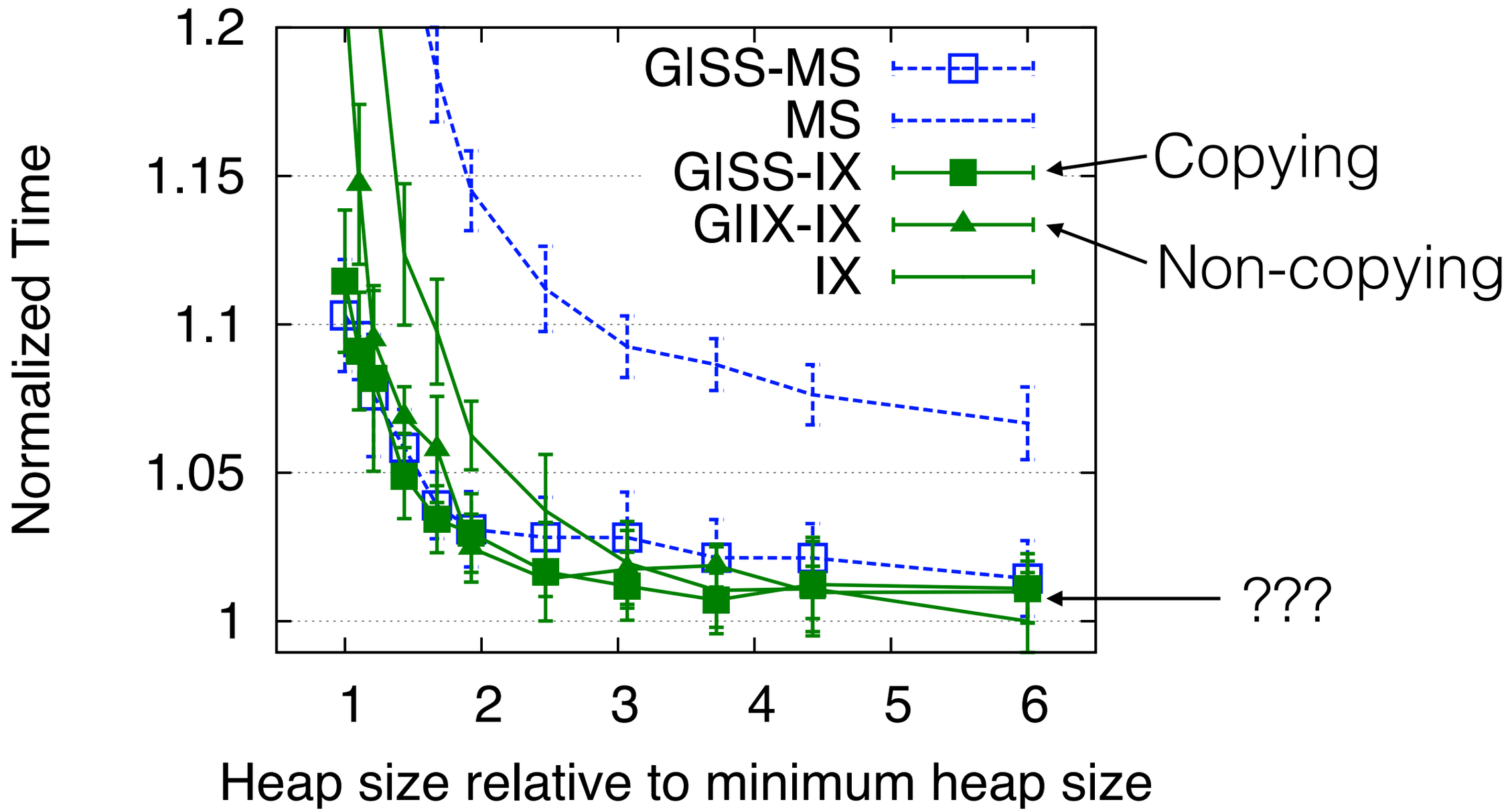
(c) Immix Variants

http://www.cs.utexas.edu/users/speedway/DaCapo/papers/immix-pldi-2008.pdf



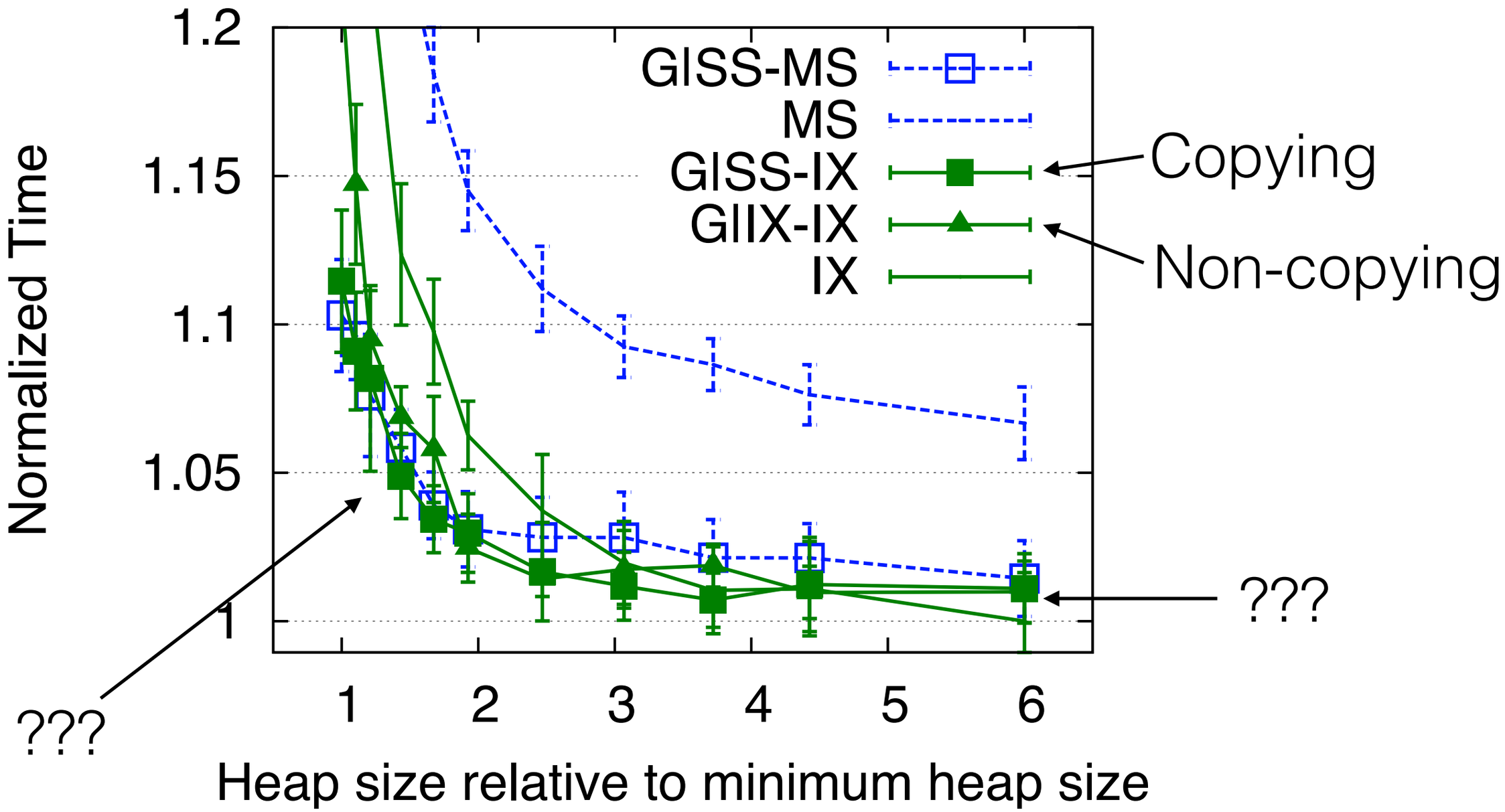
(c) Immix Variants

http://www.cs.utexas.edu/users/speedway/DaCapo/papers/immix-pldi-2008.pdf



(c) Immix Variants

http://www.cs.utexas.edu/users/speedway/DaCapo/papers/immix-pldi-2008.pdf



(c) Immix Variants

http://www.cs.utexas.edu/users/speedway/DaCapo/papers/immix-pldi-2008.pdf

- You don’t need copying to avoid sweeping.
- Copying does not save space.

- We want concurrency.

Collector Type	Barrier Overhead
Generational	Reference Write Barrier *0% overhead*
Concurrent Marking	Reference Write Barrier *0% overhead*
Concurrent Copying with To-Space Invariant	Reference Read Barrier *≥1% overhead* *cross-cutting VM changes*
Concurrent Copying with From-Space Invariant	Universal Write Barrier *forwarding pointer in header observable*

Collector Type	Barrier Overhead
Generational	Reference Write Barrier *0% overhead*
Concurrent Marking	Reference Write Barrier *0% overhead*
Concurrent Copying with To-Space Invariant	Reference Read Barrier *≥1% overhead* *cross-cutting VM changes*
Concurrent Copying with From-Space Invariant	Universal Write Barrier *forwarding pointer in header observable*

Copying Makes Everything Harder

- Harder to write native code.
- Compiler has more to worry about.
- Hashing is harder.
- Debugging VM bugs is more “exciting”.

- Not having copying enables efficient concurrent GC.
- Not having copying makes my team more productive.

- How much faster is copying?
  - *0% faster*
- Does it save space?
  - *not really*
- How much long-term maintenance cost does it incur?
  - *everything is harder*

Why mark-sweep?

Why mark-sweep?

Because it’s easier and not any worse.

Algorithm

- Huge shout out:
  - Boehm-Demers-Weiser GC
- Our GC is heavily influenced by the BDW algorithm.
- Our GC is an independent implementation of that algorithm, with major extensions:
- Some for performance.
- Some to support VM features.
  - *This makes direct perf comparison impossible.*

Algorithm

- Simple Segregated Storage
- Bump’n’pop
- Constraint-Based Marking
- Parallel “Draining”
- Sticky Mark Bits
- Conservative Roots
- Fragmentation Mitigations

Algorithm

- Simple Segregated Storage
- Bump’n’pop
- Constraint-Based Marking
- Parallel “Draining”
- Sticky Mark Bits
- Conservative Roots
- Fragmentation Mitigations

Simple Segregated Storage

Desired Property	Possible Solution
Given a size, find a free object	
Given an object, get its size	
Given an address, find if it's inside an object	
Given an object, find its adjacent objects	
Coalesce and Split	

Desired Property	Possible Solution
Given a size, find a free object	Free List per Size Class (Segregated Free Lists)
Given an object, get its size	
Given an address, find if it's inside an object	
Given an object, find its adjacent objects	
Coalesce and Split	

Desired Property	Possible Solution
Given a size, find a free object	Free List per Size Class (Segregated Free Lists)
Given an object, get its size	16KB blocks of same-size objects and a header
Given an address, find if it's inside an object	
Given an object, find its adjacent objects	
Coalesce and Split	

Desired Property	Possible Solution
Given a size, find a free object	Free List per Size Class (Segregated Free Lists)
Given an object, get its size	16KB blocks of same-size objects and a header
Given an address, find if it's inside an object	Global HashSet of blocks
Given an object, find its adjacent objects	
Coalesce and Split	

Desired Property	Possible Solution
Given a size, find a free object	Free List per Size Class (Segregated Free Lists)
Given an object, get its size	16KB blocks of same-size objects and a header
Given an address, find if it's inside an object	Global HashSet of blocks
Given an object, find its adjacent objects	16KB blocks of same-size objects and a header
Coalesce and Split	

Desired Property	Possible Solution
Given a size, find a free object	Free List per Size Class (Segregated Free Lists)
Given an object, get its size	16KB blocks of same-size objects and a header
Given an address, find if it's inside an object	Global HashSet of blocks
Given an object, find its adjacent objects	16KB blocks of same-size objects and a header
Coalesce and Split	Blocks can switch size classes when they are empty

16 byte

32 byte

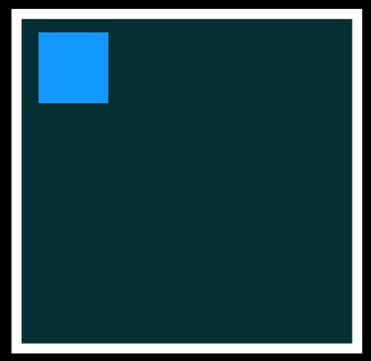
64 byte

16 byte

32 byte

64 byte

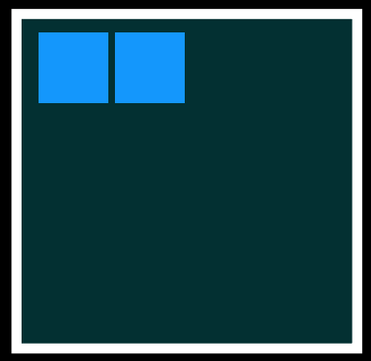
16 byte



32 byte

64 byte

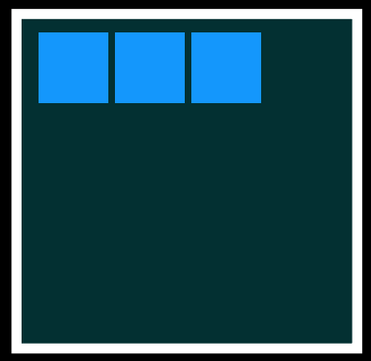
16 byte



32 byte

64 byte

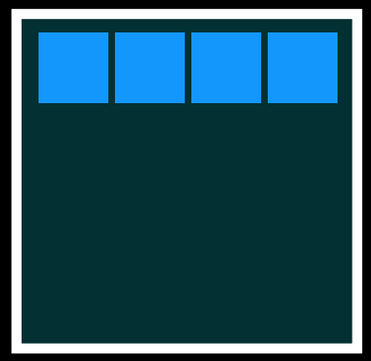
16 byte



32 byte

64 byte

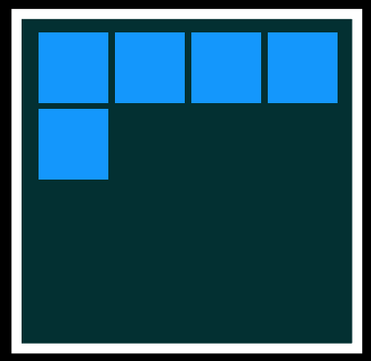
16 byte



32 byte

64 byte

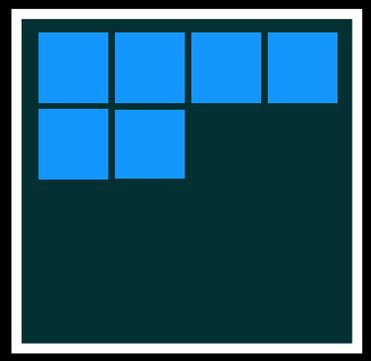
16 byte



32 byte

64 byte

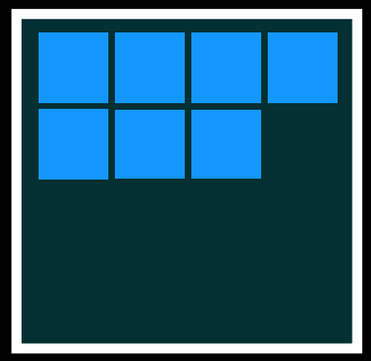
16 byte



32 byte

64 byte

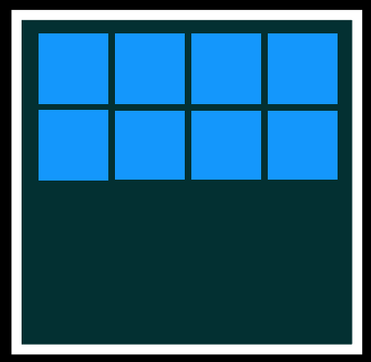
16 byte



32 byte

64 byte

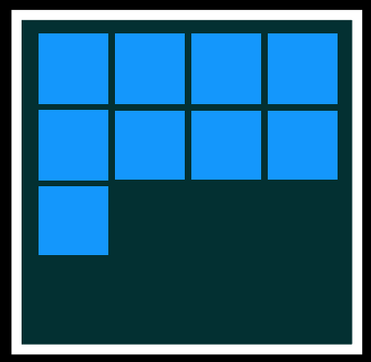
16 byte



32 byte

64 byte

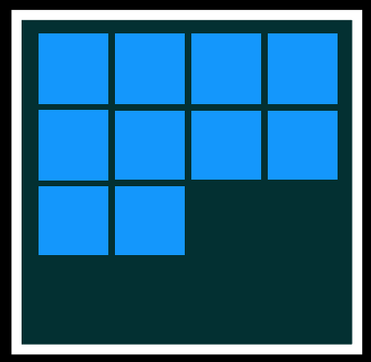
16 byte



32 byte

64 byte

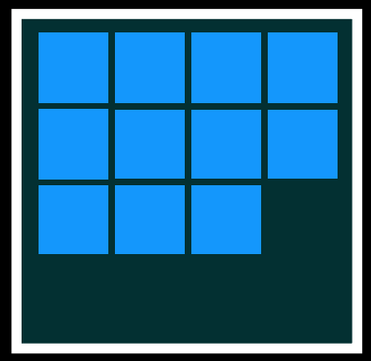
16 byte



32 byte

64 byte

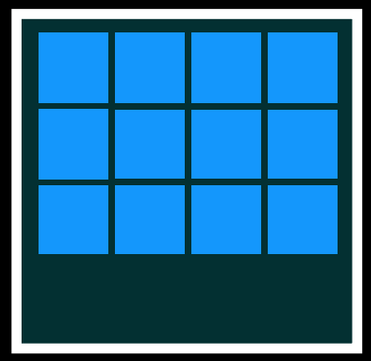
16 byte



32 byte

64 byte

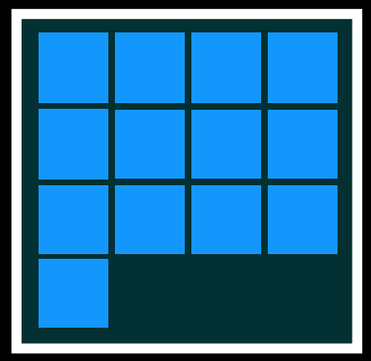
16 byte



32 byte

64 byte

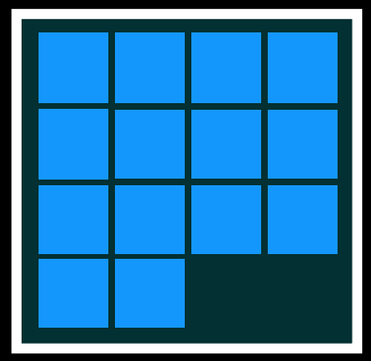
16 byte



32 byte

64 byte

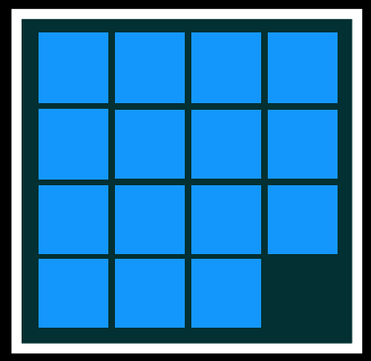
16 byte



32 byte

64 byte

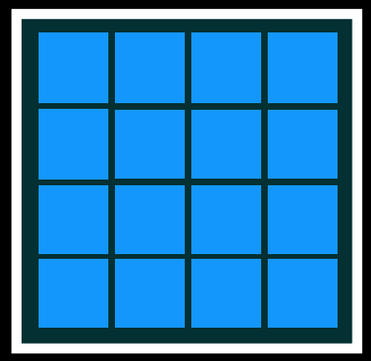
16 byte



32 byte

64 byte

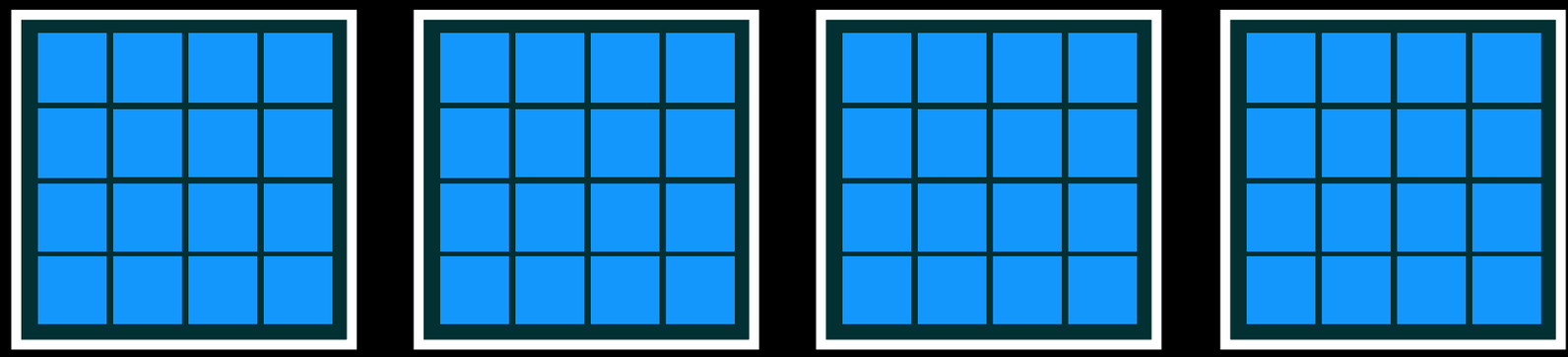
16 byte



32 byte

64 byte

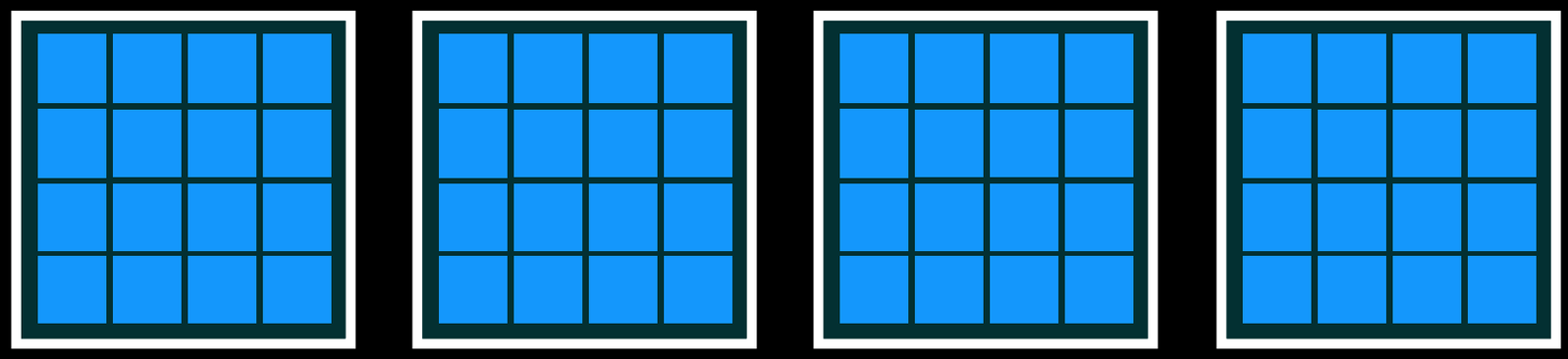
16 byte



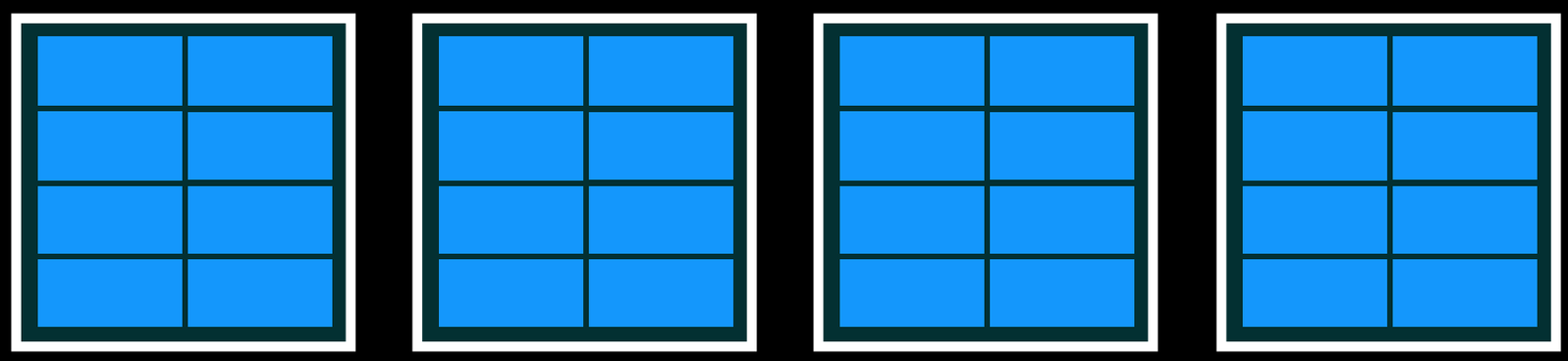
32 byte

64 byte

16 byte

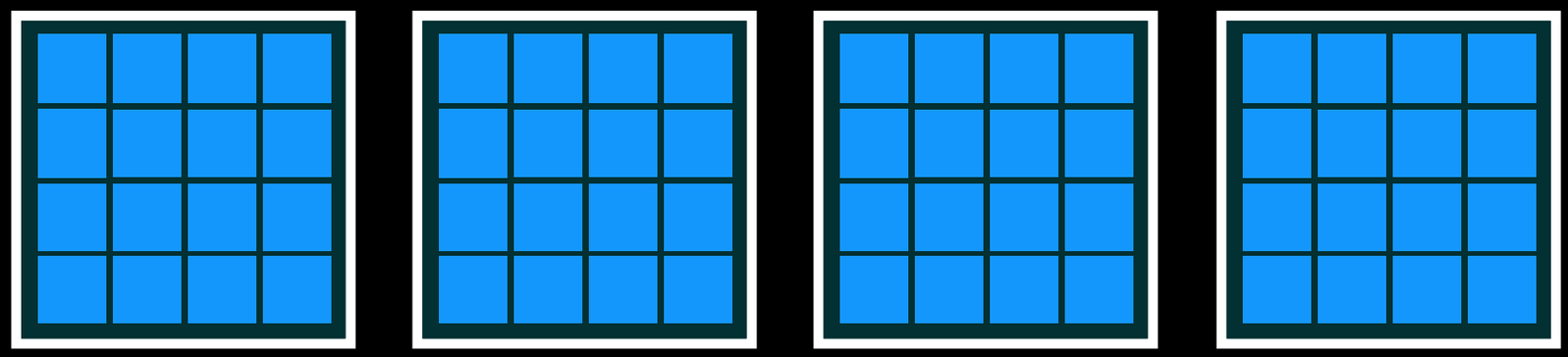


32 byte

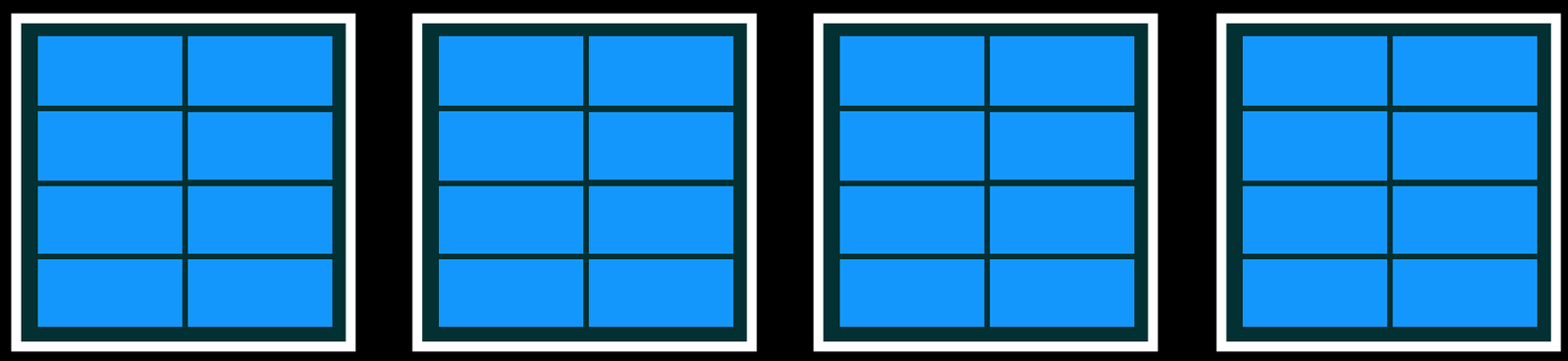


64 byte

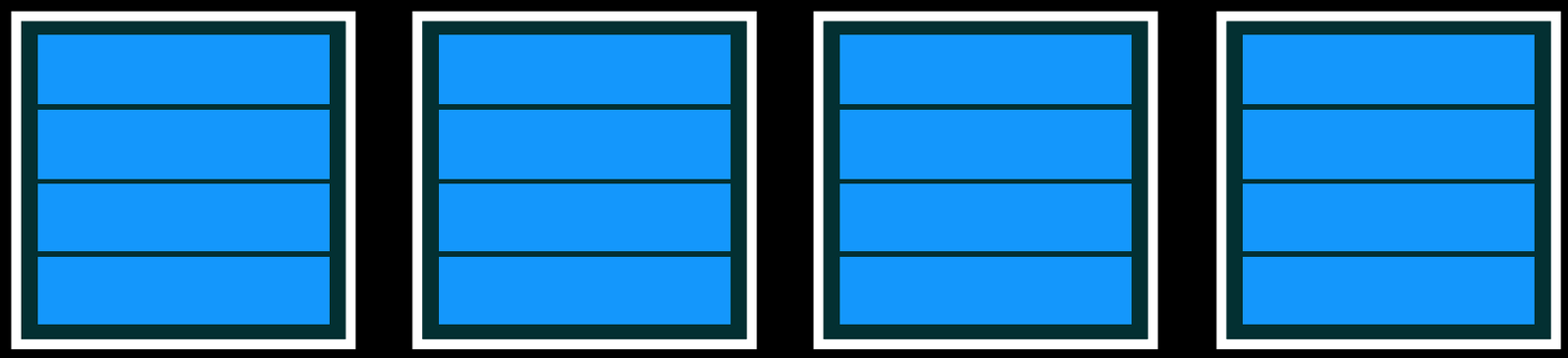
16 byte



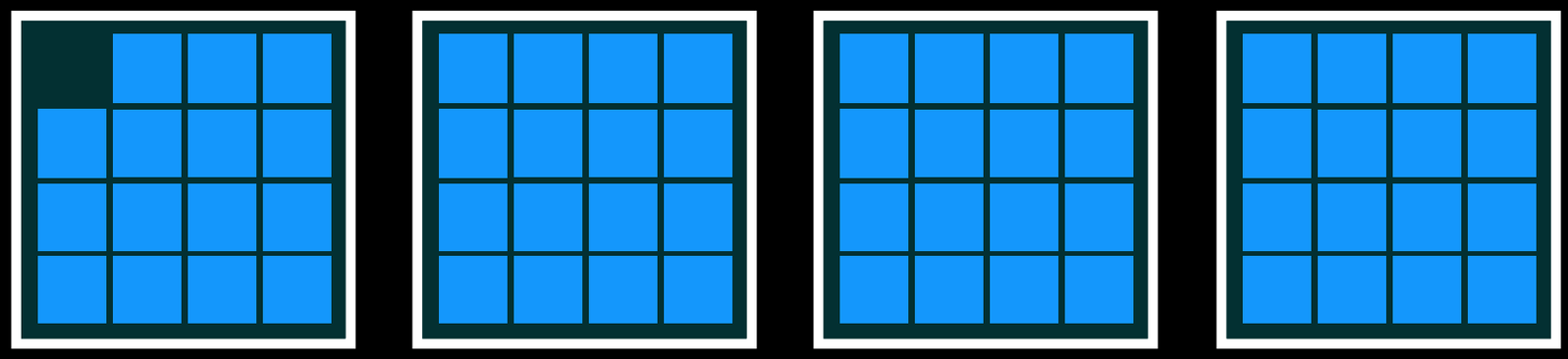
32 byte



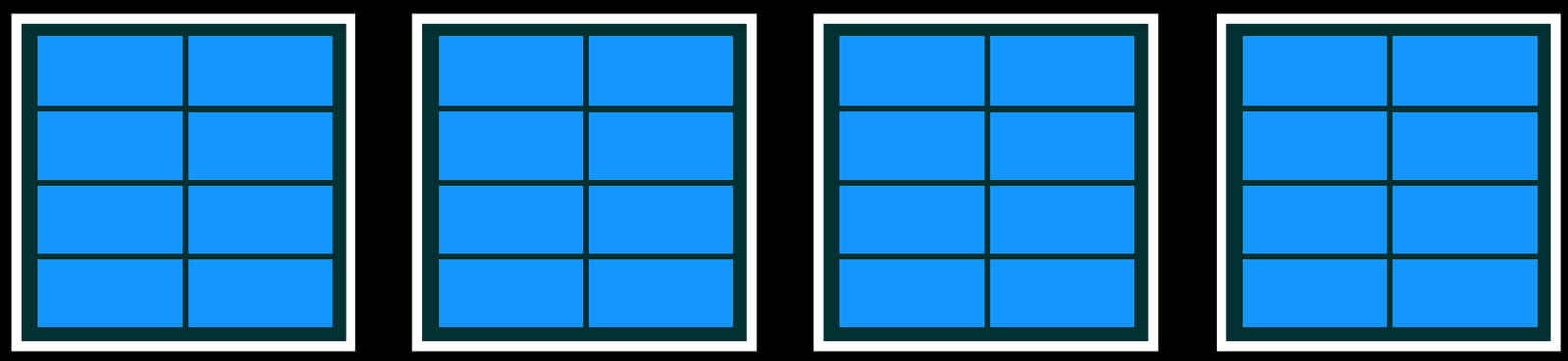
64 byte



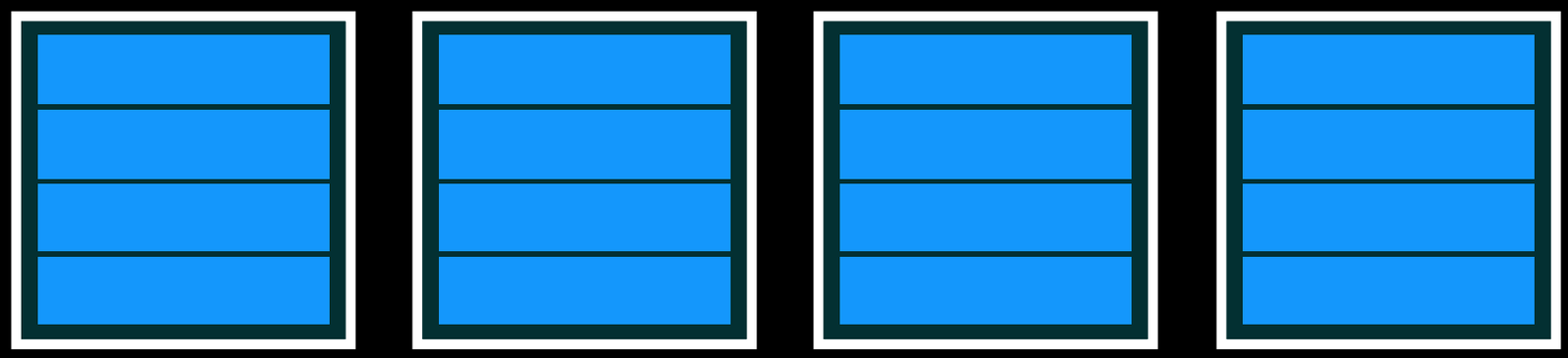
16 byte



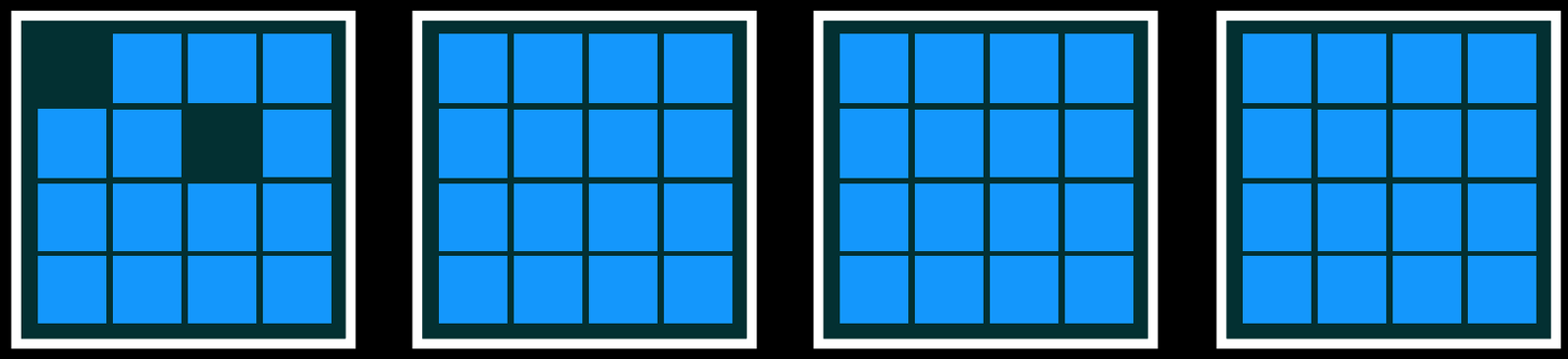
32 byte



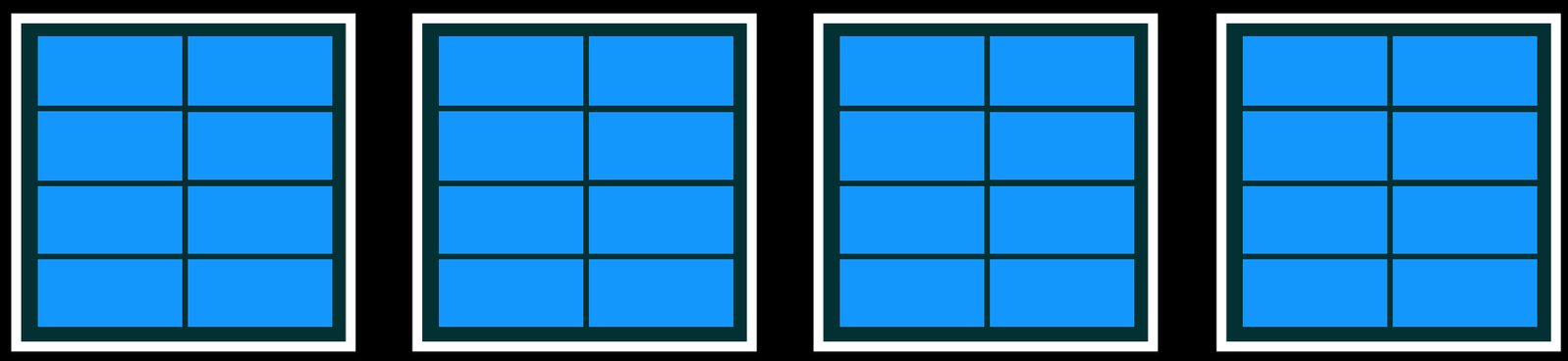
64 byte



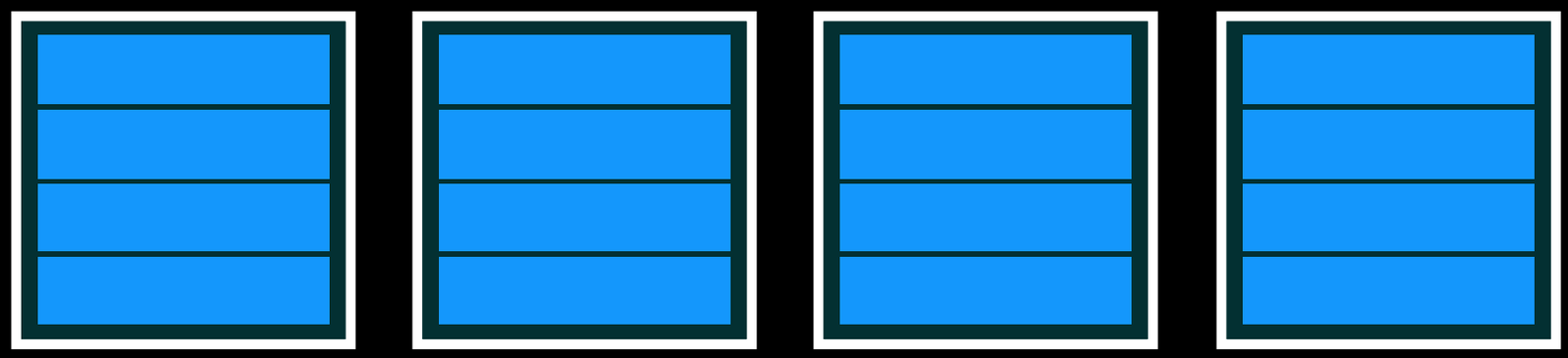
16 byte



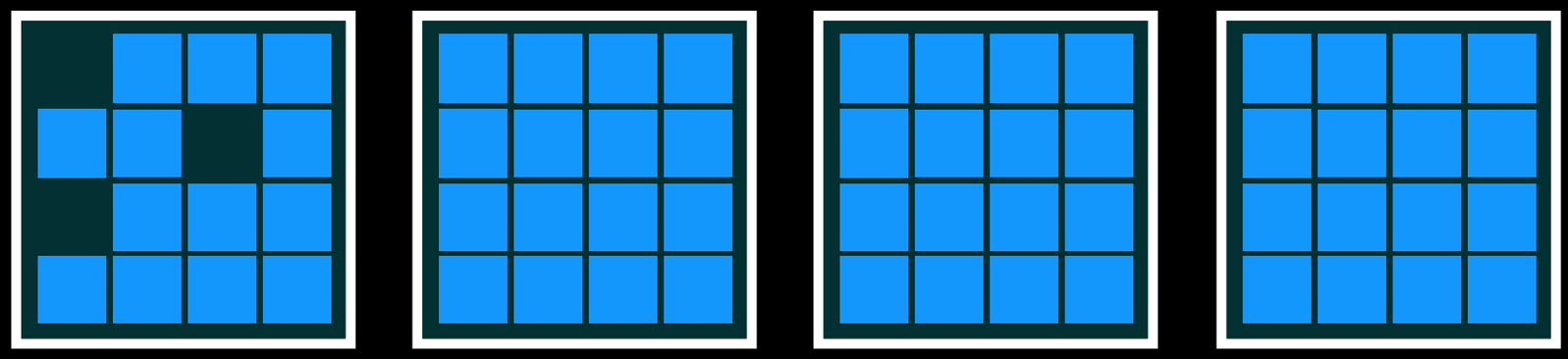
32 byte



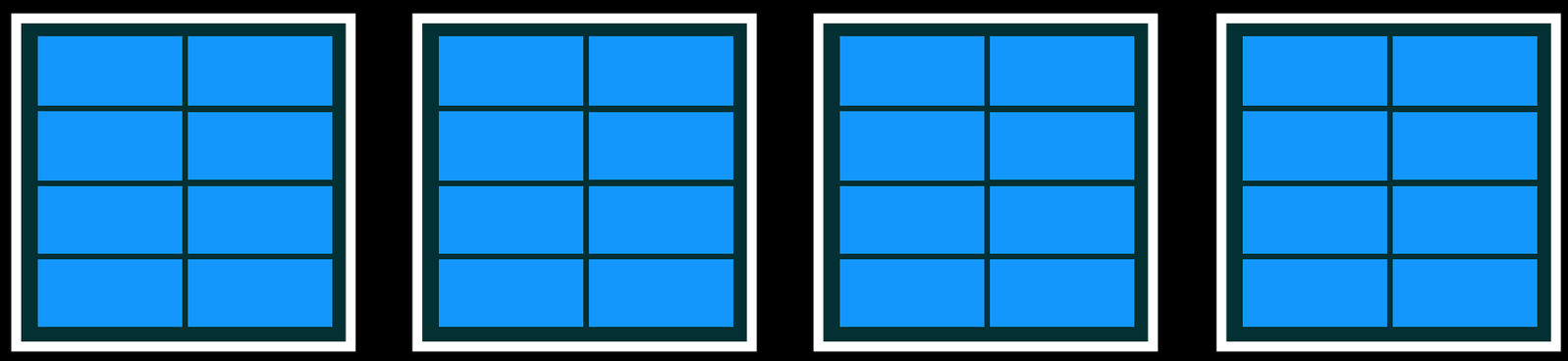
64 byte



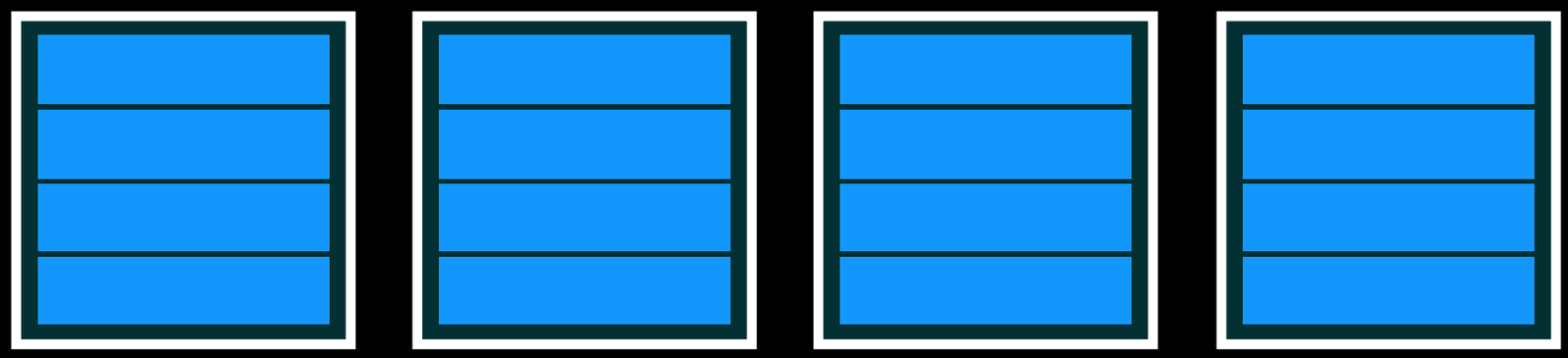
16 byte



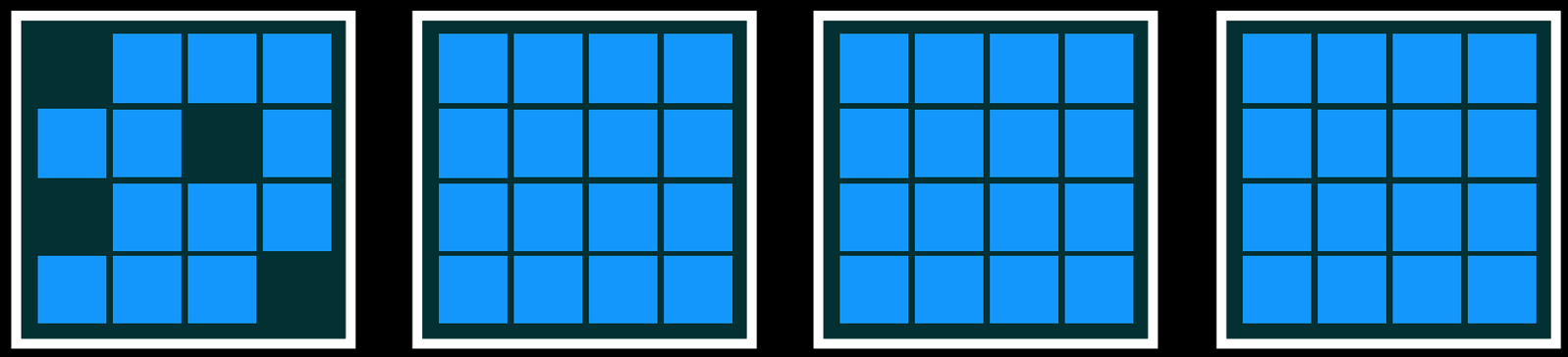
32 byte



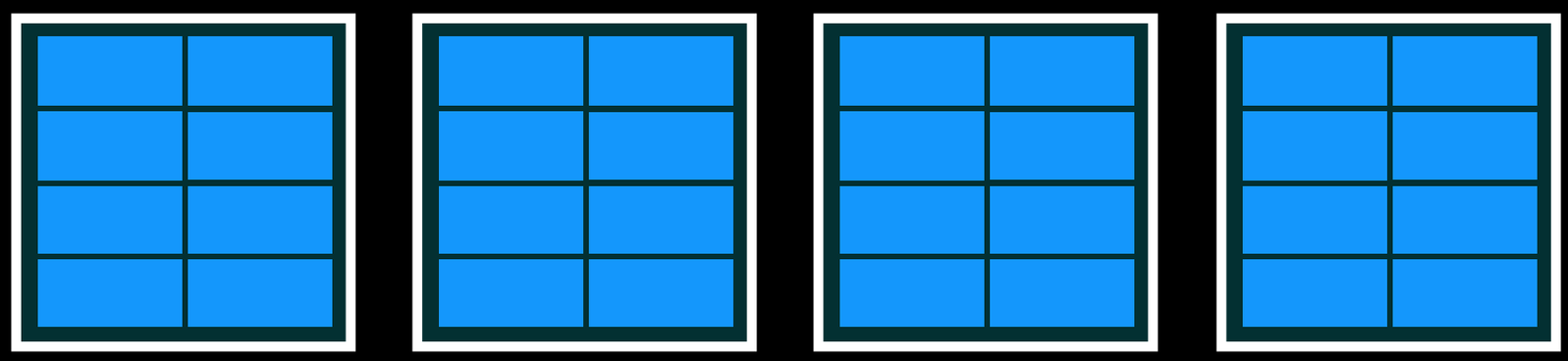
64 byte



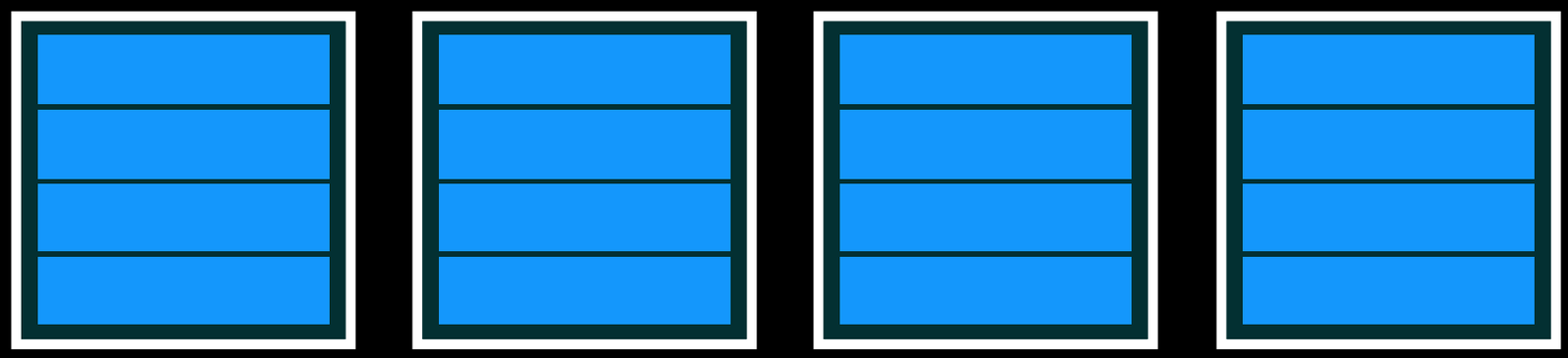
16 byte



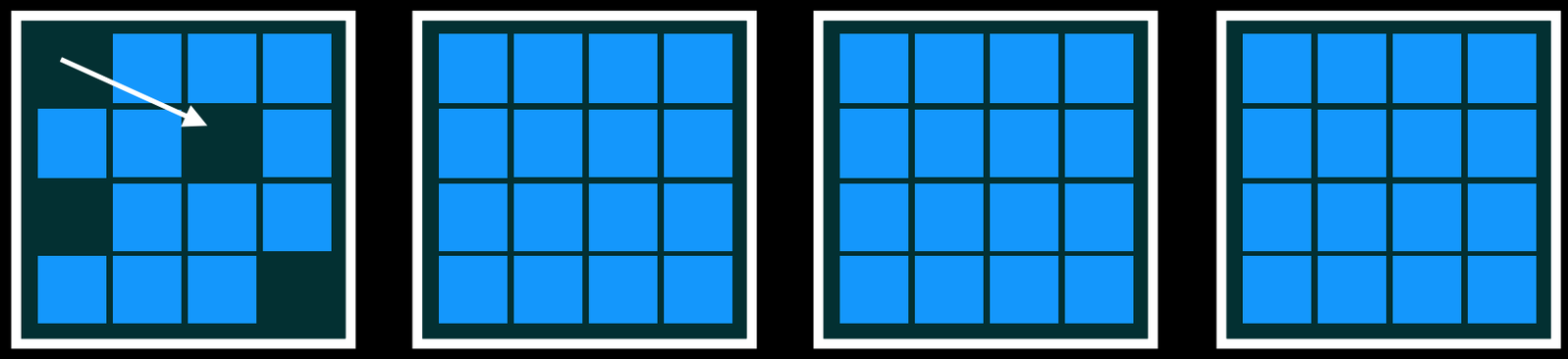
32 byte



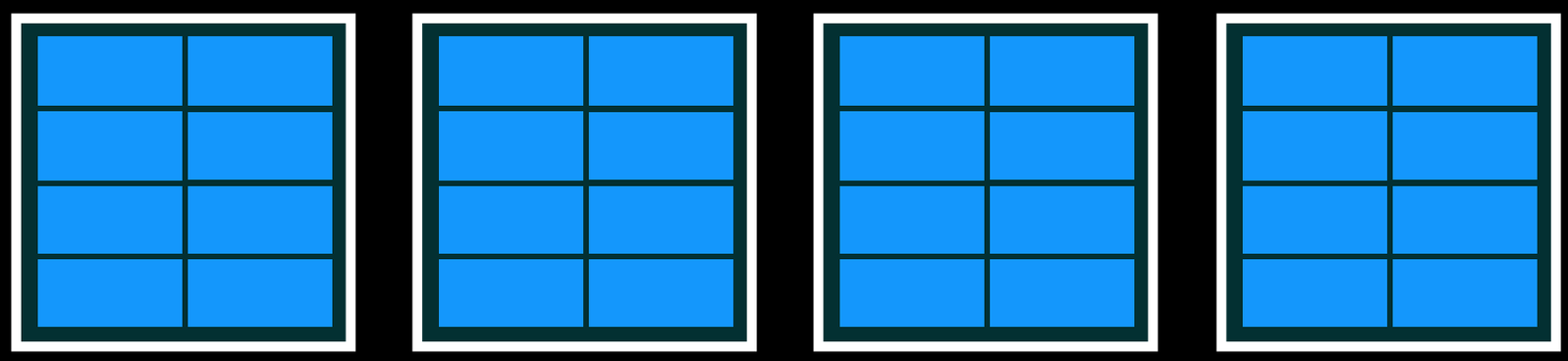
64 byte



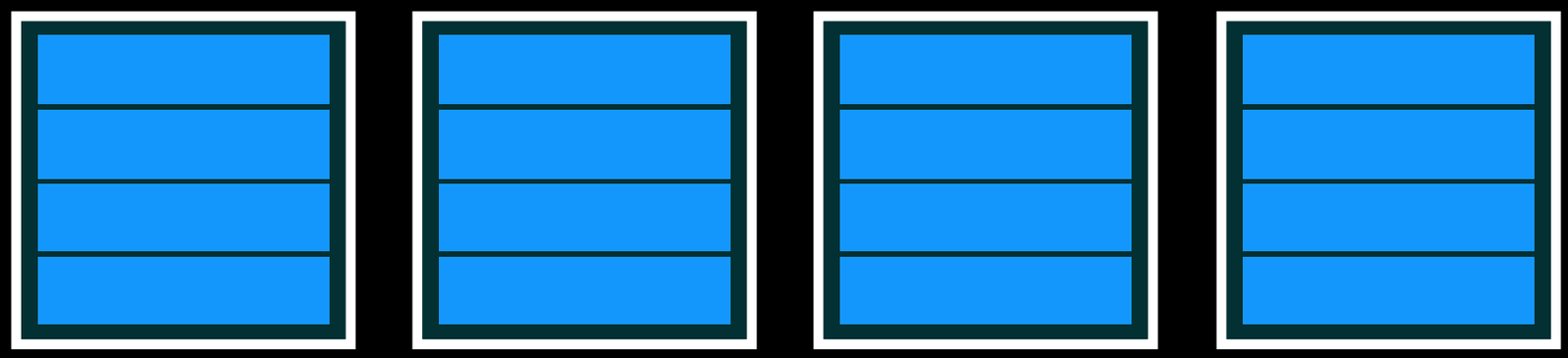
16 byte



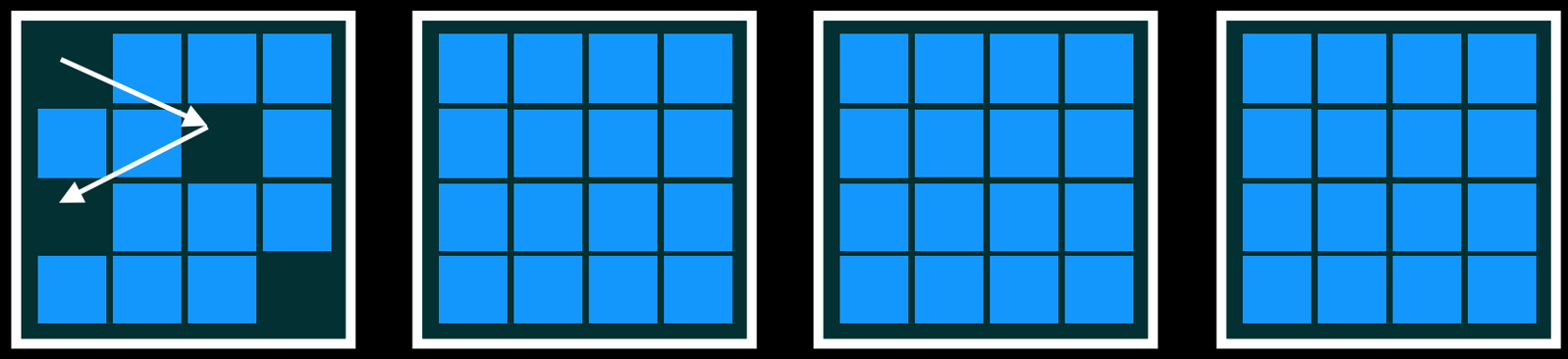
32 byte



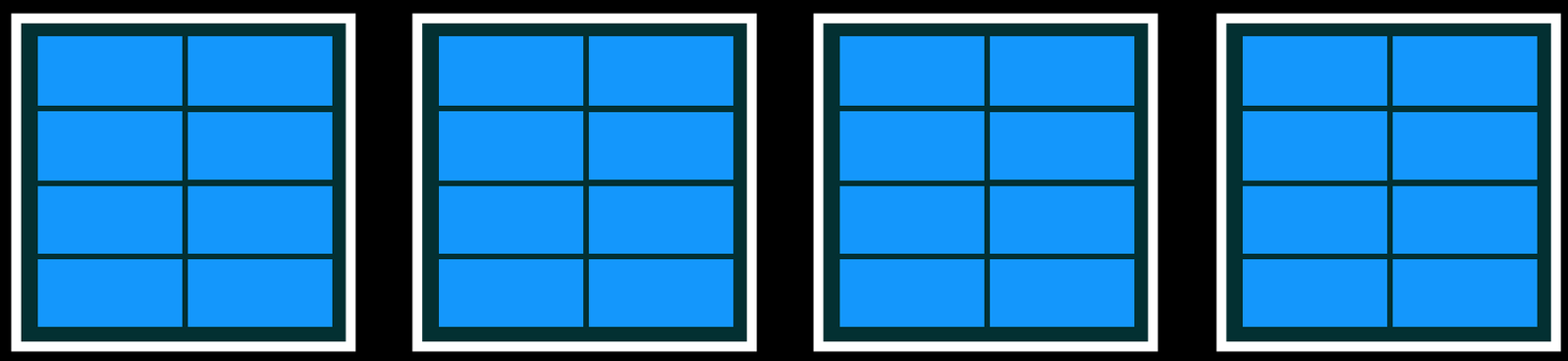
64 byte



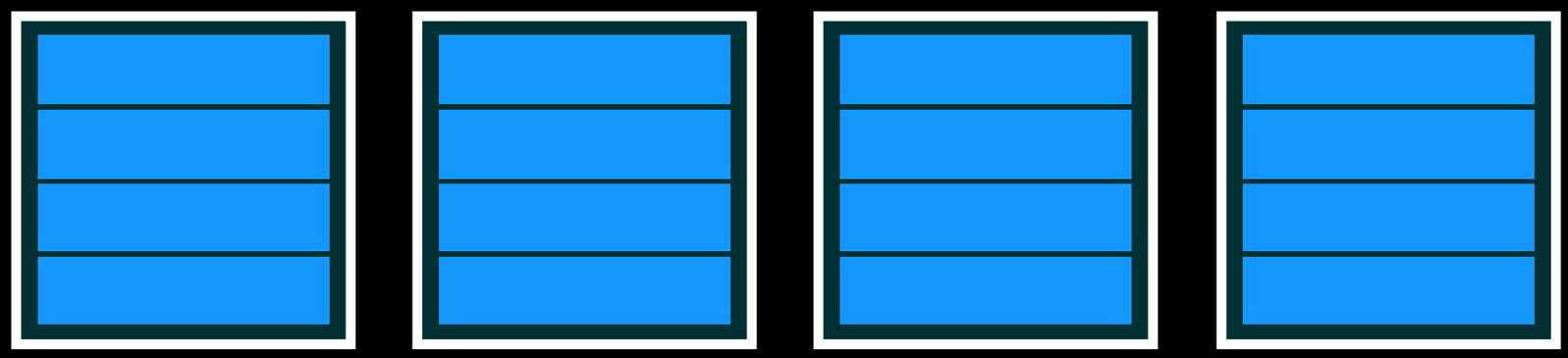
16 byte



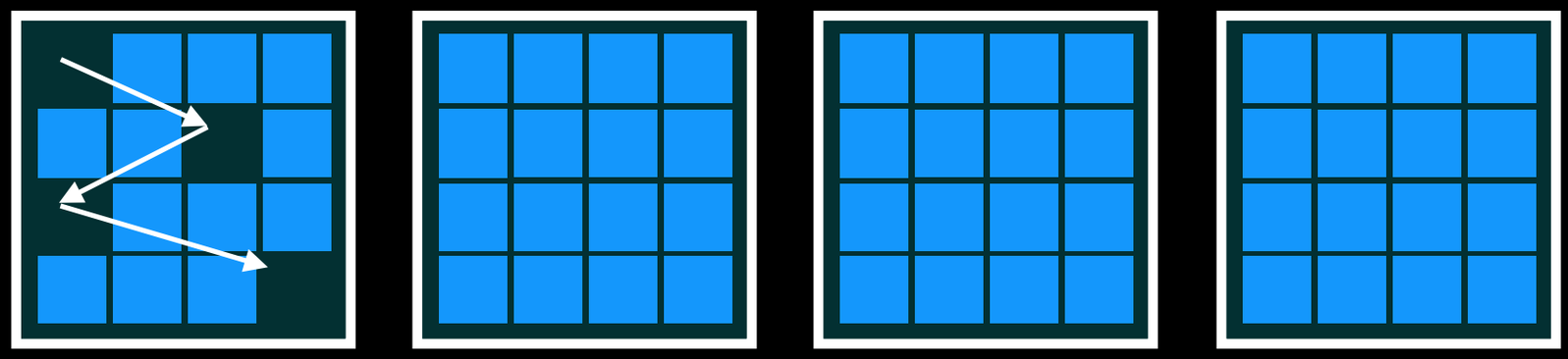
32 byte



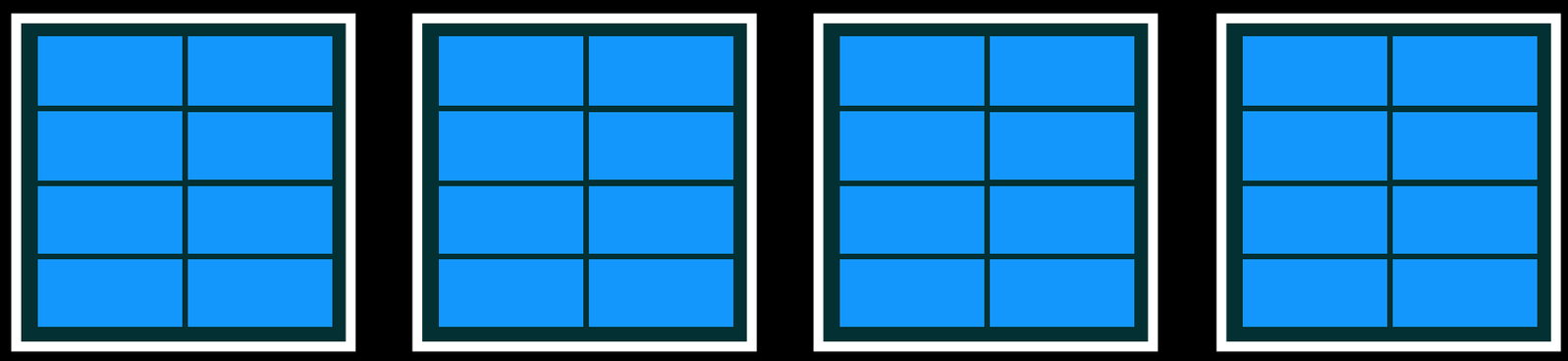
64 byte



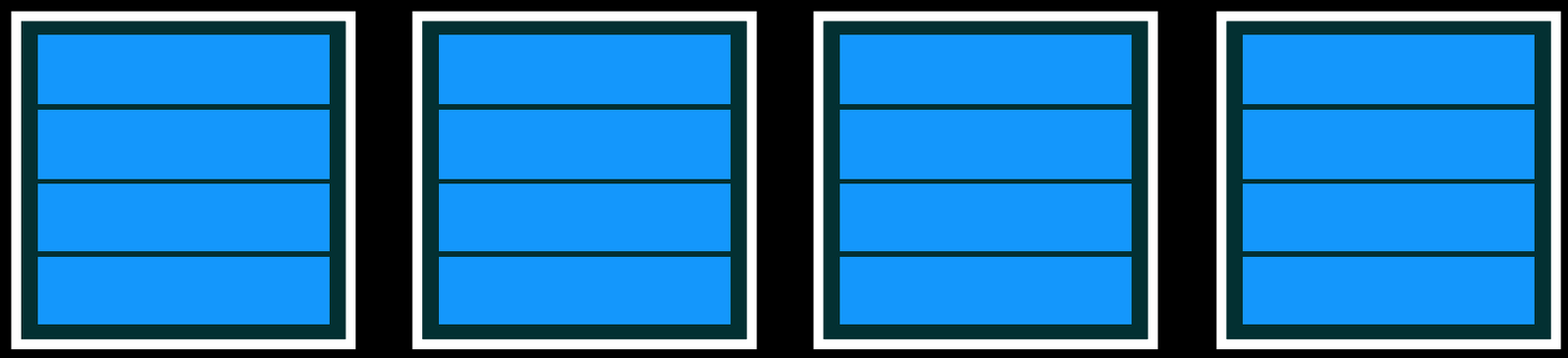
16 byte

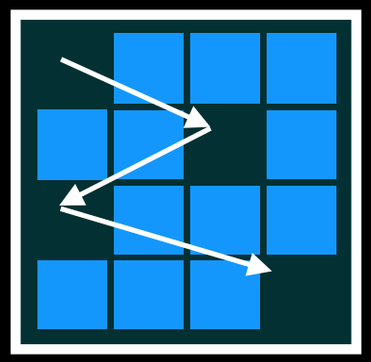


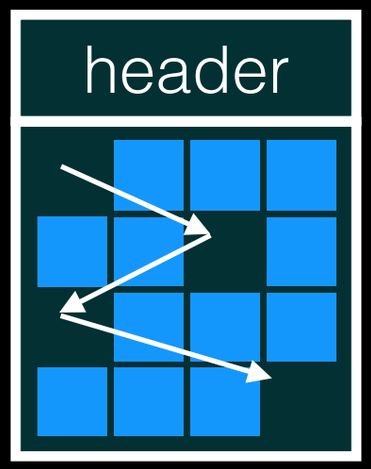
32 byte

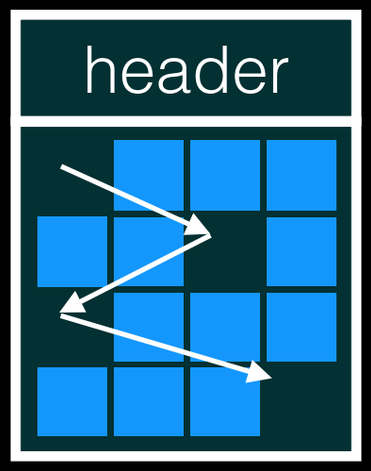


64 byte






header



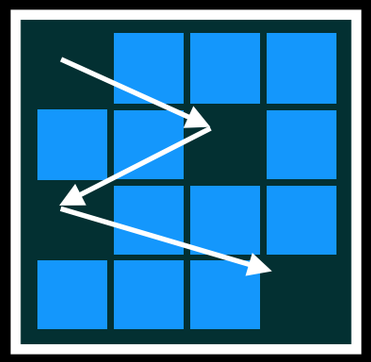
Header contains:

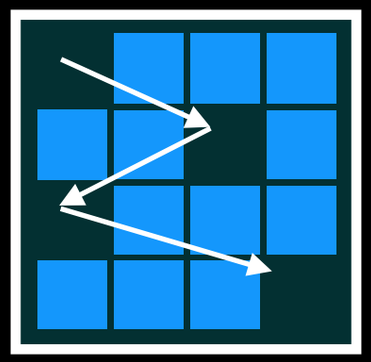
- size class
- mark bits
- marking version
- occupancy count

Simple Segregated Storage

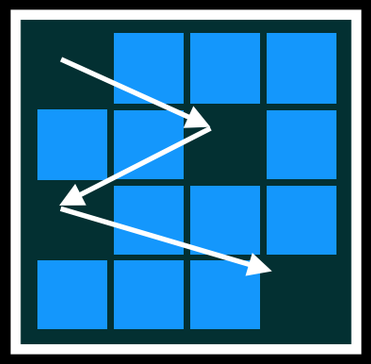
- Free list per size class
- Each 16KB block belongs to one size class
- Each block can self-identify size class
- Blocks maintain marking state
- Global HashSet of blocks

Bump’n’pop





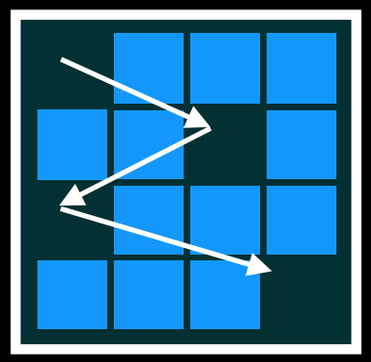
Building free-lists is *expensive*



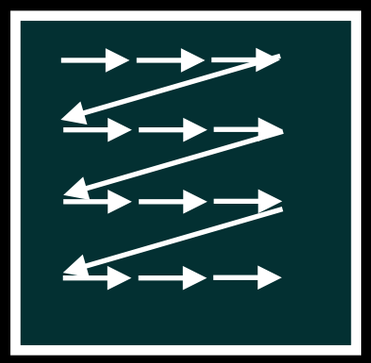
Building free-lists is *expensive*

Easily doubles cost of allocating small objects

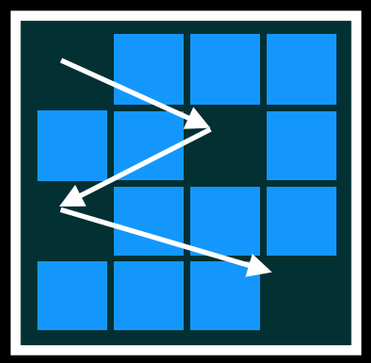
- Have to check each mark bit
  - *Our mark bits are super compact*
  - *This is as cheap as it gets*
    - *But it's still not free!*
- Have to build linked list
  - *Touch 16KB*



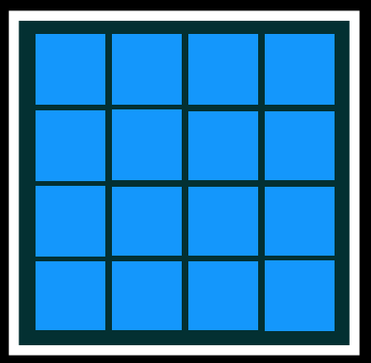
Completely Empty



Fragmented



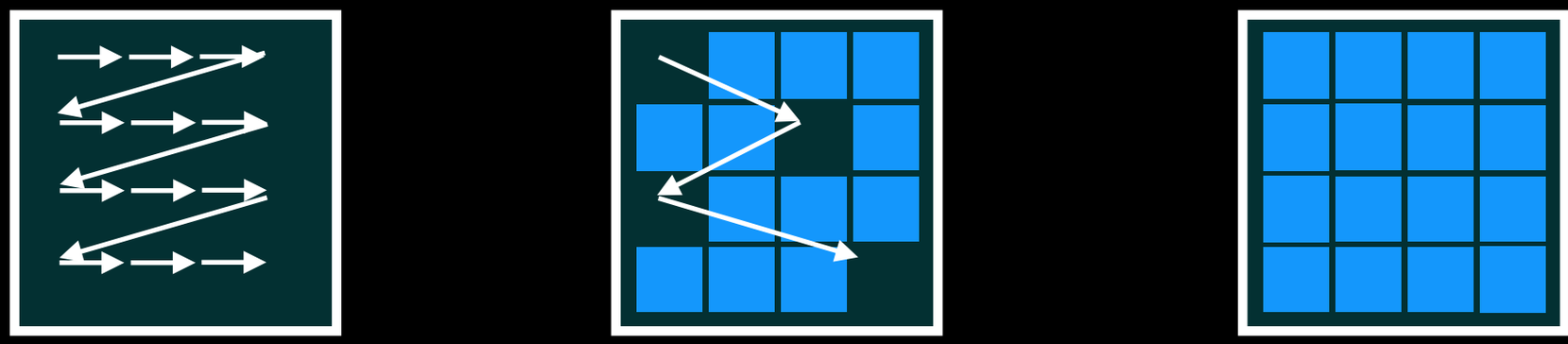
Completely Full



Completely Empty

Fragmented

Completely Full

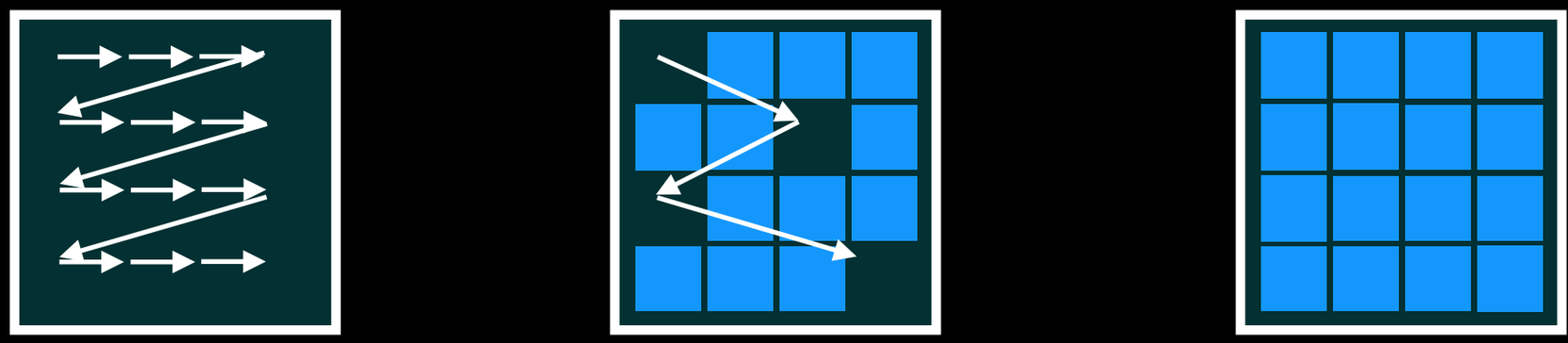


Super Common

Completely Empty

Fragmented

Completely Full



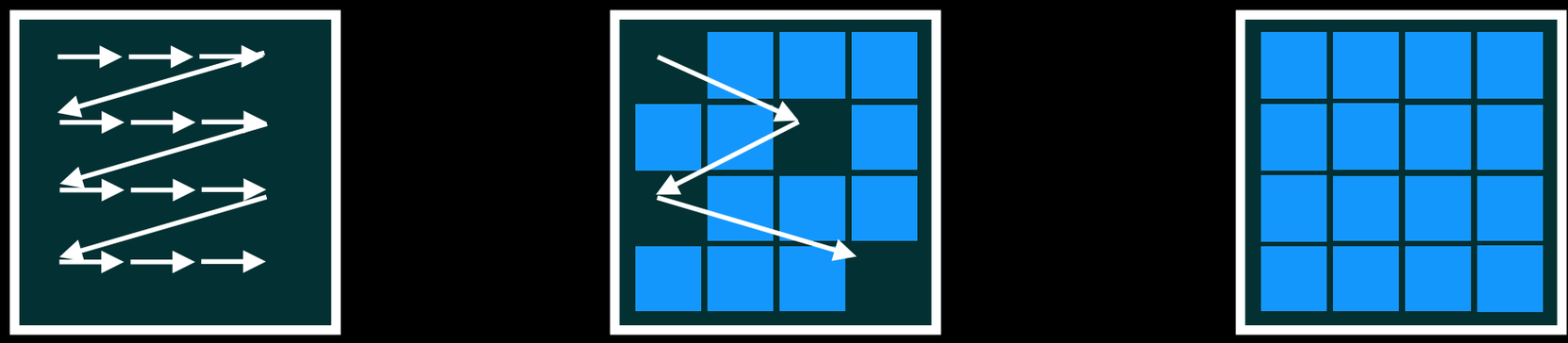
Super Common

High Infant Mortality

Completely Empty

Fragmented

Completely Full



Super Common

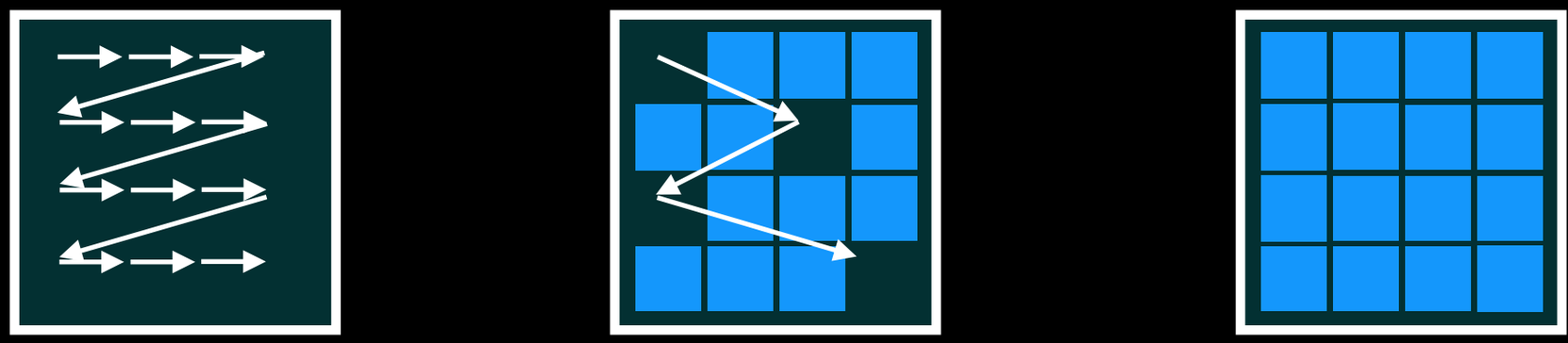
Fairly Common

High Infant Mortality

Completely Empty

Fragmented

Completely Full



Super Common

Fairly Common

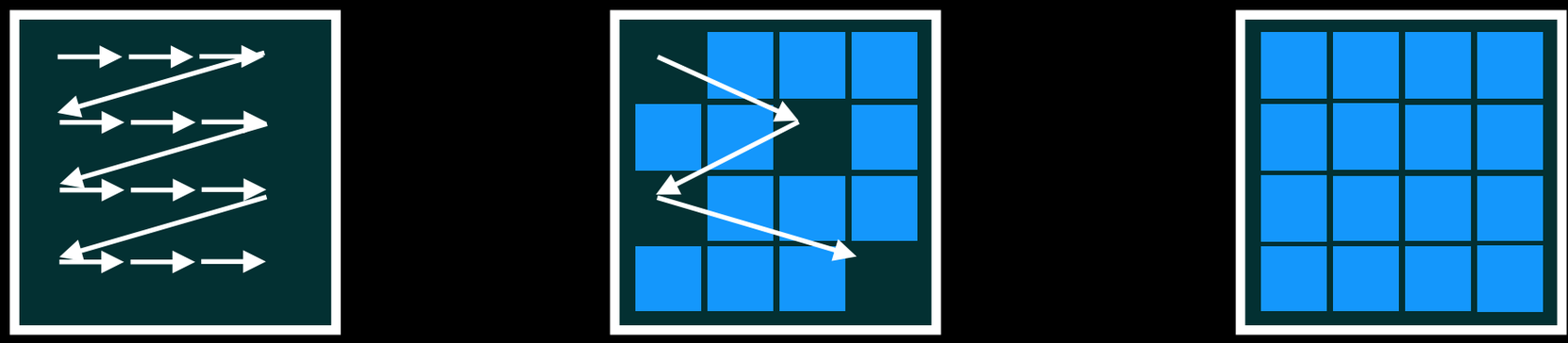
High Infant Mortality

Objects Allocated Together Have Similar Lifetimes

Completely Empty

Fragmented

Completely Full



Super Common Uncommon Fairly Common

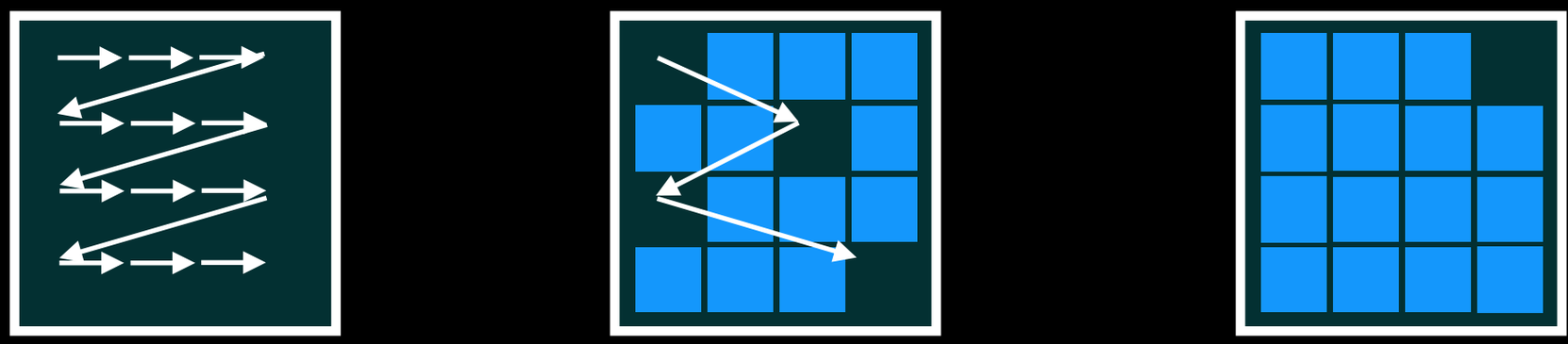
High Infant Mortality

Objects Allocated Together Have Similar Lifetimes

Completely Empty

Fragmented

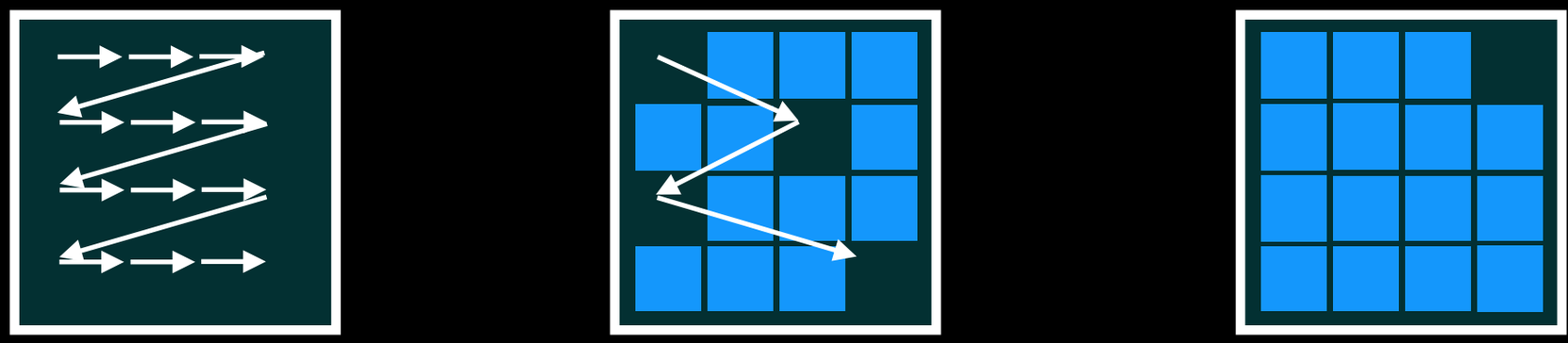
90% Full



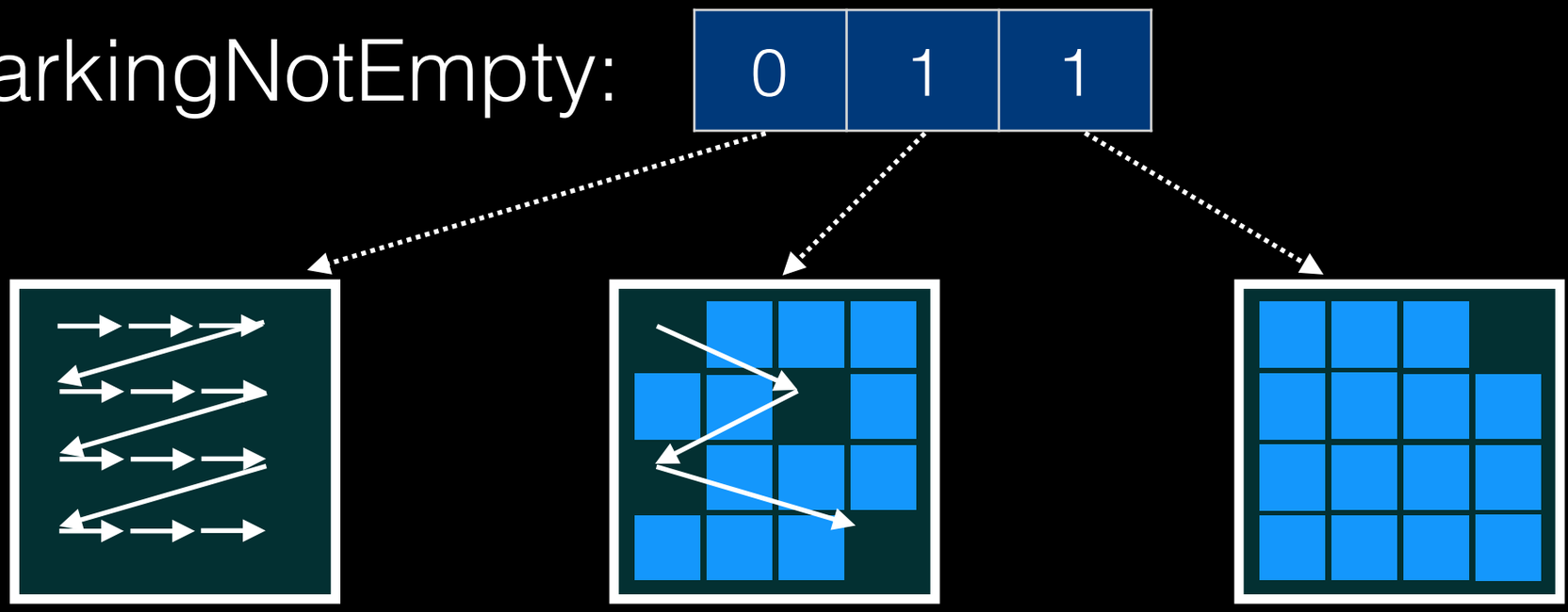
Super Common Uncommon Fairly Common

High Infant Mortality

Objects Allocated Together Have Similar Lifetimes

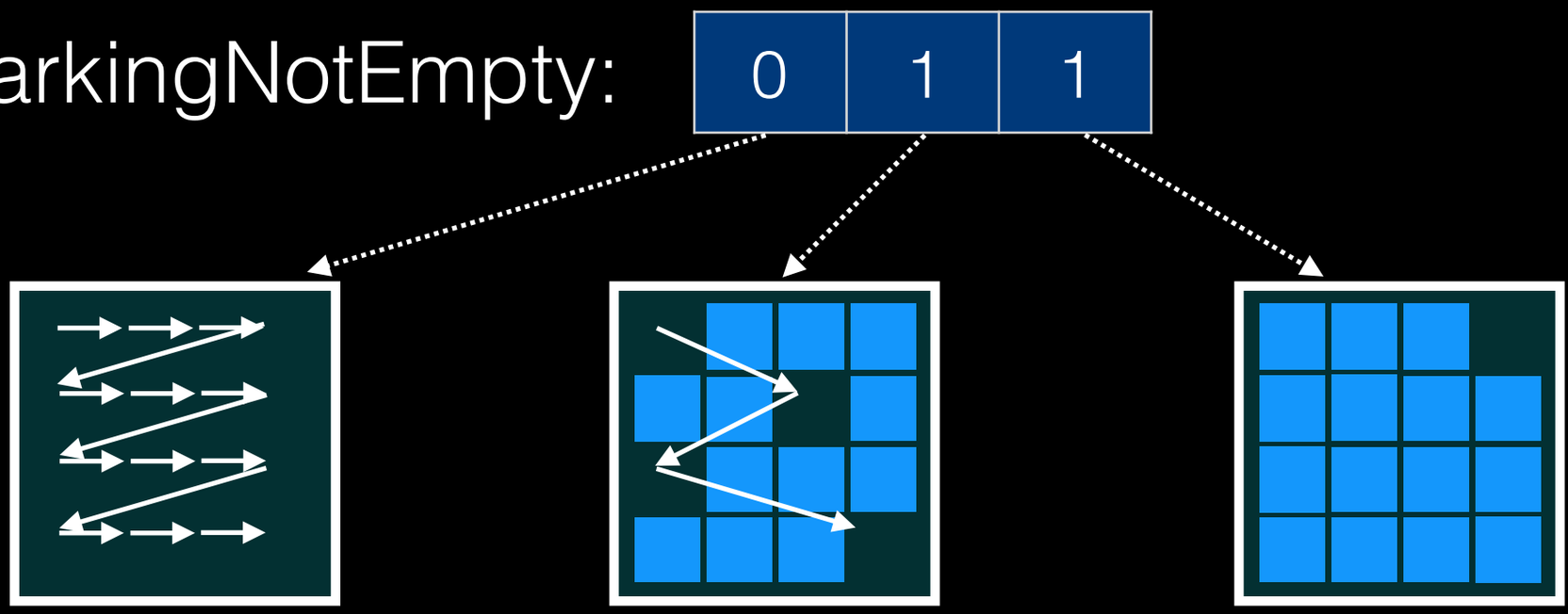


markingNotEmpty:



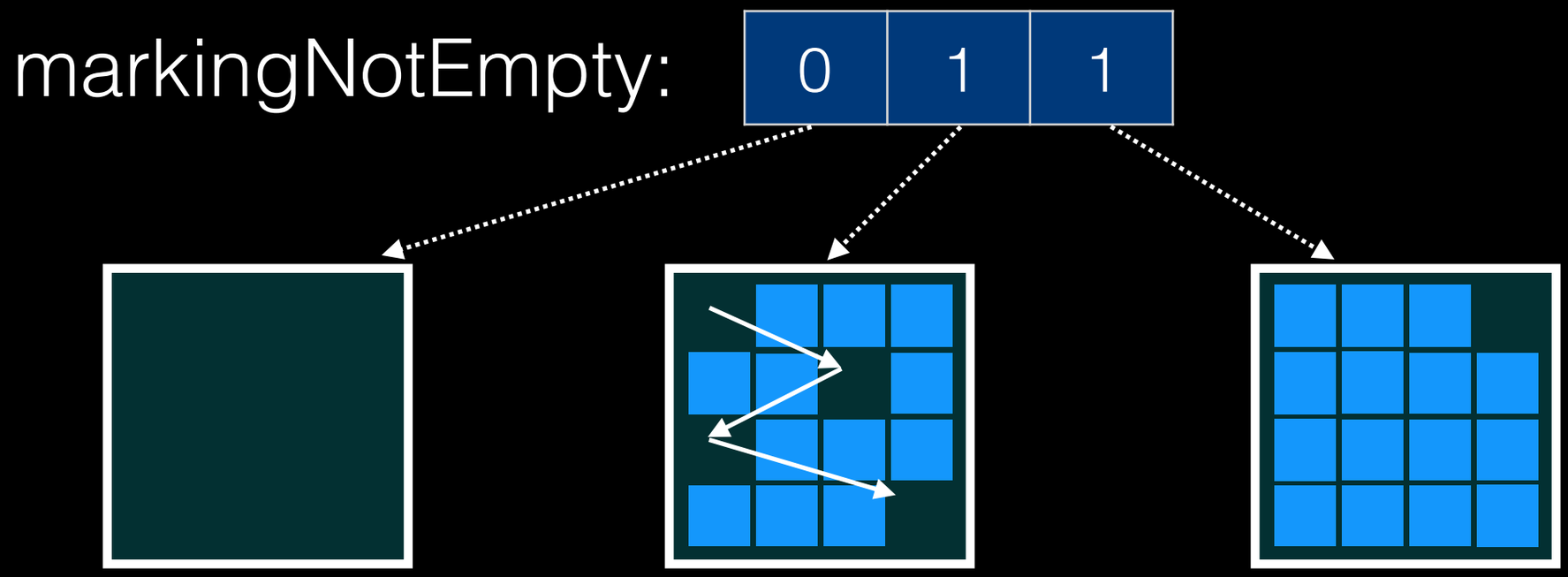
GC rapidly identifies empty blocks

markingNotEmpty:



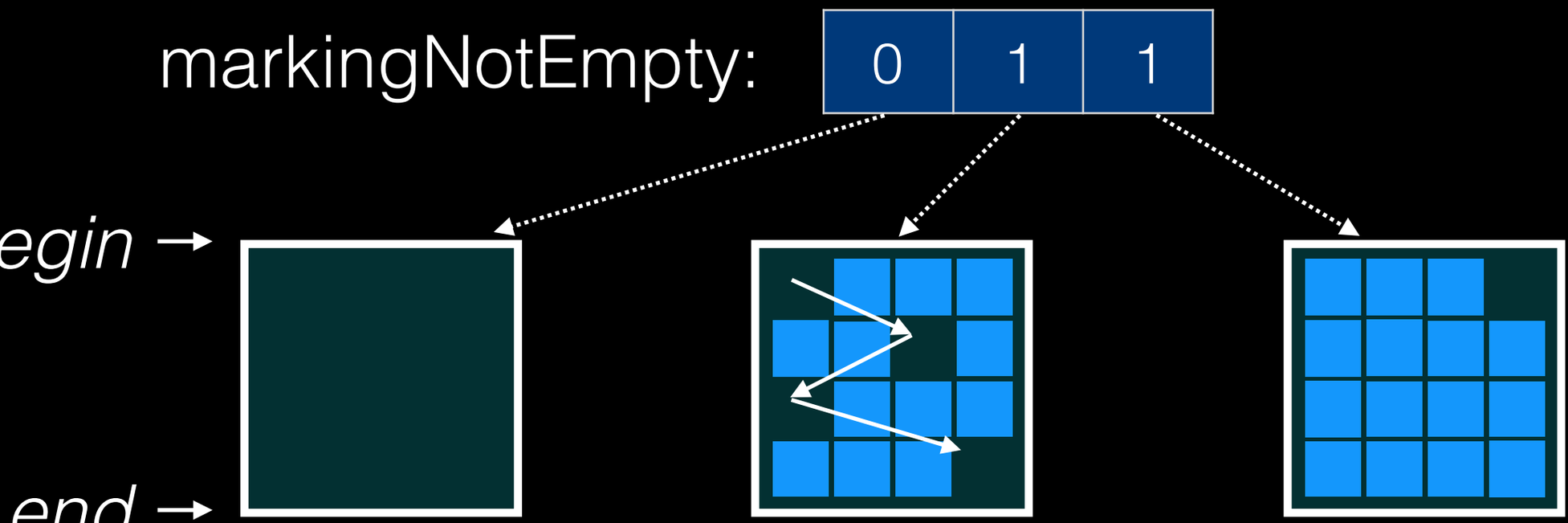
GC rapidly identifies empty blocks

No need to scan mark bits of empty blocks



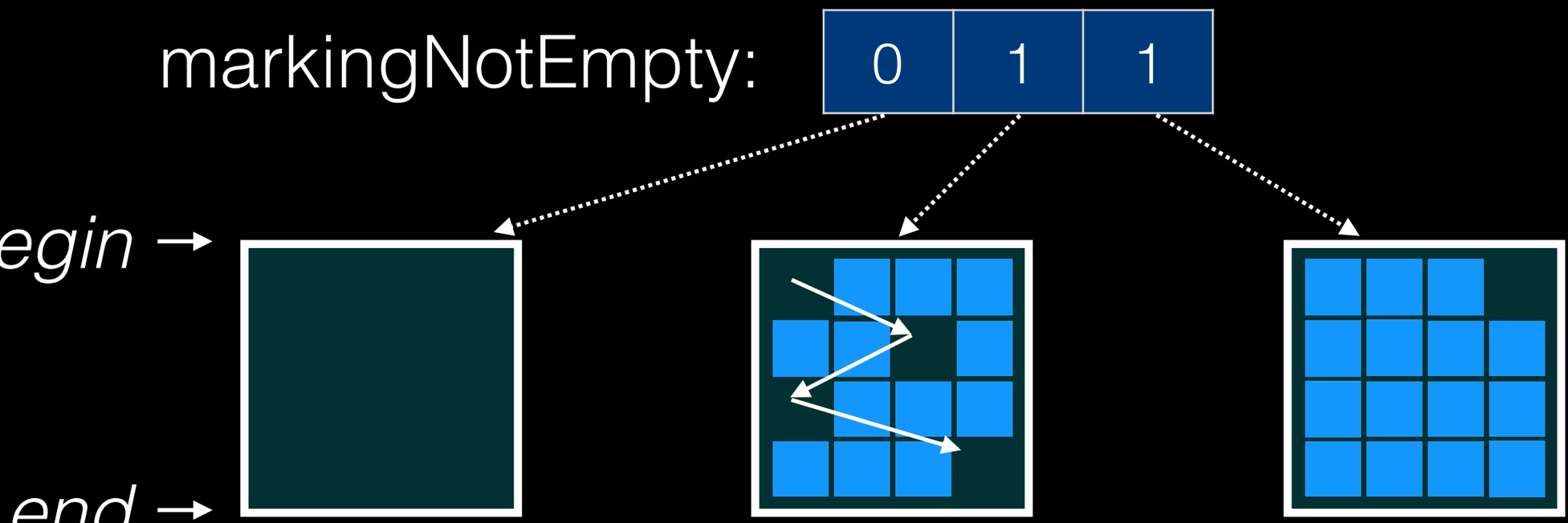
GC rapidly identifies empty blocks

No need to scan mark bits of empty blocks



GC rapidly identifies empty blocks

No need to scan mark bits of empty blocks

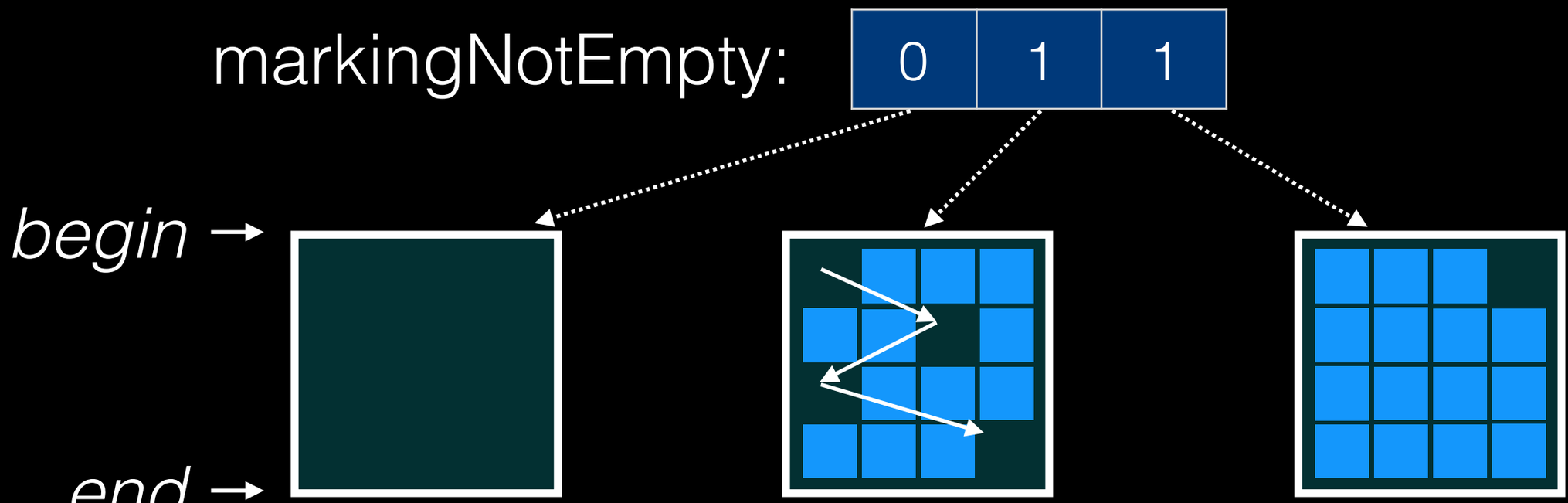


GC rapidly identifies empty blocks

No need to scan mark bits of empty blocks

Empty blocks use bump allocator

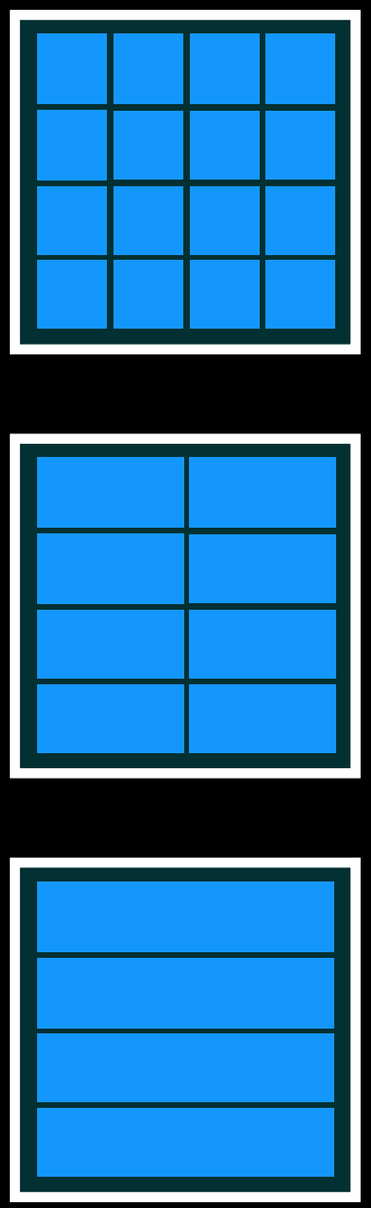
bump’n’pop



GC rapidly identifies empty blocks

No need to scan mark bits of empty blocks

Empty blocks use bump allocator



Simple segregated storage assumes that all objects in a block have the same size.

Our bump allocator preserves this.

Allocation Algorithm

1. Round up allocation size to size class
2. Allocate using size class's size
    I. First try bump
    II. Then try pop

```
template<typename Func>
HeapCell* FreeList::allocate(const Func& slowPath)
{
    unsigned remaining = m_remaining;
    if (remaining) {
        unsigned cellSize = m_cellSize;
        remaining -= cellSize;                                  // bump
        m_remaining = remaining;
        return bitwise_cast<HeapCell*>(m_payloadEnd - remaining - cellSize);
    }

    FreeCell* result = head();
    if (UNLIKELY(!result))
        return slowPath();

                                                                // pop
    m_scrambledHead = result->scrambledNext;
    return bitwise_cast<HeapCell*>(result);
}
```

```
template<typename Func>
HeapCell* FreeList::allocate(const Func& slowPath)
{
    unsigned remaining = m_remaining;
    if (remaining) {
        unsigned cellSize = m_cellSize;
        remaining -= cellSize;                                  // bump
        m_remaining = remaining;
        return bitwise_cast<HeapCell*>(m_payloadEnd - remaining - cellSize);
    }

    FreeCell* result = head();
    if (UNLIKELY(!result))
        return slowPath();

                                                                // pop
    m_scrambledHead = result->scrambledNext;
    return bitwise_cast<HeapCell*>(result);
}
```

```
template<typename Func>
HeapCell* FreeList::allocate(const Func& slowPath)
{
    unsigned remaining = m_remaining;
    if (remaining) {
        unsigned cellSize = m_cellSize;
        remaining -= cellSize;                                  // bump
        m_remaining = remaining;
        return bitwise_cast<HeapCell*>(m_payloadEnd - remaining - cellSize);
    }

    FreeCell* result = head();
    if (UNLIKELY(!result))
        return slowPath();

                                                                // pop
    m_scrambledHead = result->scrambledNext;
    return bitwise_cast<HeapCell*>(result);
}
```

```
template<typename Func>
HeapCell* FreeList::allocate(const Func& slowPath)
{
    unsigned remaining = m_remaining;
    if (remaining) {
        unsigned cellSize = m_cellSize;
        remaining -= cellSize;                                  // bump
        m_remaining = remaining;
        return bitwise_cast<HeapCell*>(m_payloadEnd - remaining - cellSize);
    }

    FreeCell* result = head();
    if (UNLIKELY(!result))
        return slowPath();

                                                                // pop
    m_scrambledHead = result->scrambledNext;
    return bitwise_cast<HeapCell*>(result);
}
```

bump'n'pop uses scrambled free-lists

```
template<typename Func>
HeapCell* FreeList::allocate(const Func& slowPath)
{
    unsigned remaining = m_remaining;
    if (remaining) {
        unsigned cellSize = m_cellSize;
        remaining -= cellSize;                                  // bump
        m_remaining = remaining;
        return bitwise_cast<HeapCell*>(m_payloadEnd - remaining - cellSize);
    }

    FreeCell* result = head();
    if (UNLIKELY(!result))
        return slowPath();

                                                                // pop
    m_scrambledHead = result->scrambledNext;
    return bitwise_cast<HeapCell*>(result);
}
```

bump'n'pop uses scrambled free-lists

scrambled free-lists haven't shipped yet!


structure ID
indexing
type
flags
cell state
butterfly pointer
native buffer


structure ID
indexing
type
flags
cell state
butterfly pointer
native buffer


structure ID
indexing
type
flags
cell state
butterfly pointer
native buffer
FreeCell::next

structure ID | indexing | type | flags | cell state | butterfly pointer | *native buffer* | FreeCell::next

structure ID	indexing	type	flags	cell state	butterfly pointer	0x31337c0defefebaadbeefaa

- Scramble the freelist with a per-freelist secret.
- Select a new secret every time we build a freelist.

- Bump’n’pop is as fast as our previous copying bump allocator, and faster than our previous pop-only allocator.

- We also tried concurrent sweeping. Bump’n’pop was faster.

- We also tried concurrent sweeping with bump’n’pop. It wasn’t any better than just bump’n’pop.

bump’n’pop

- Empty blocks don’t scan mark bits.
- Empty blocks don’t build free lists.
- 90% full blocks are treated as if they were full.

Constraint-Based Marking

Constraint-Based Marking

- Transitive reachability is not always enough
- Common examples:
  - Soft references
  - Weak map

Constraint-Based Marking

- Transitive reachability is not always enough
- WebKit examples:
  - Type inference
  - Weak map
  - DOM
  - Native code

Constraint-Based Marking

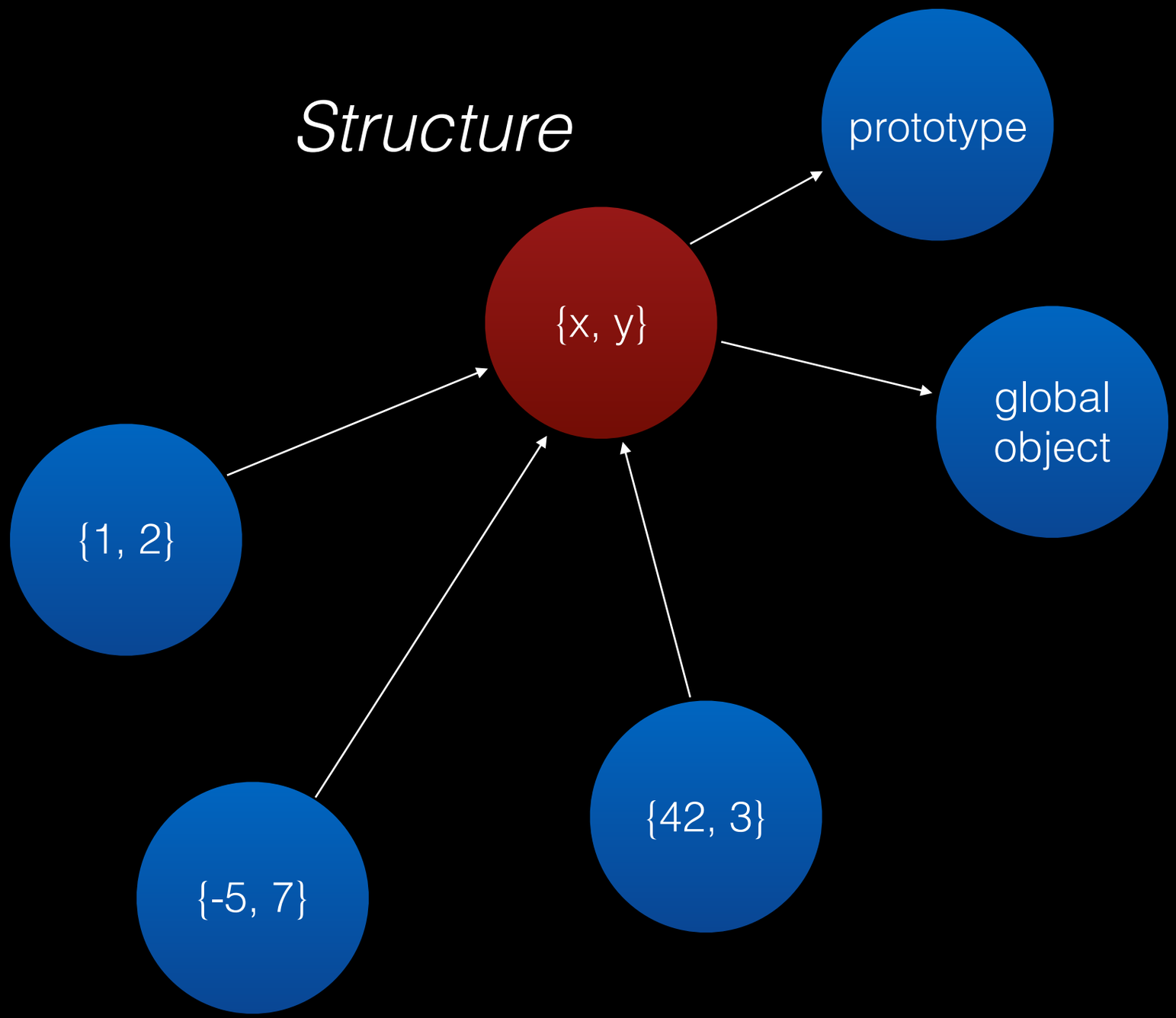
- Transitive reachability is not always enough
- WebKit examples:
  - Type inference
  - Weak map
  - DOM
  - Native code

Constraint-Based Marking

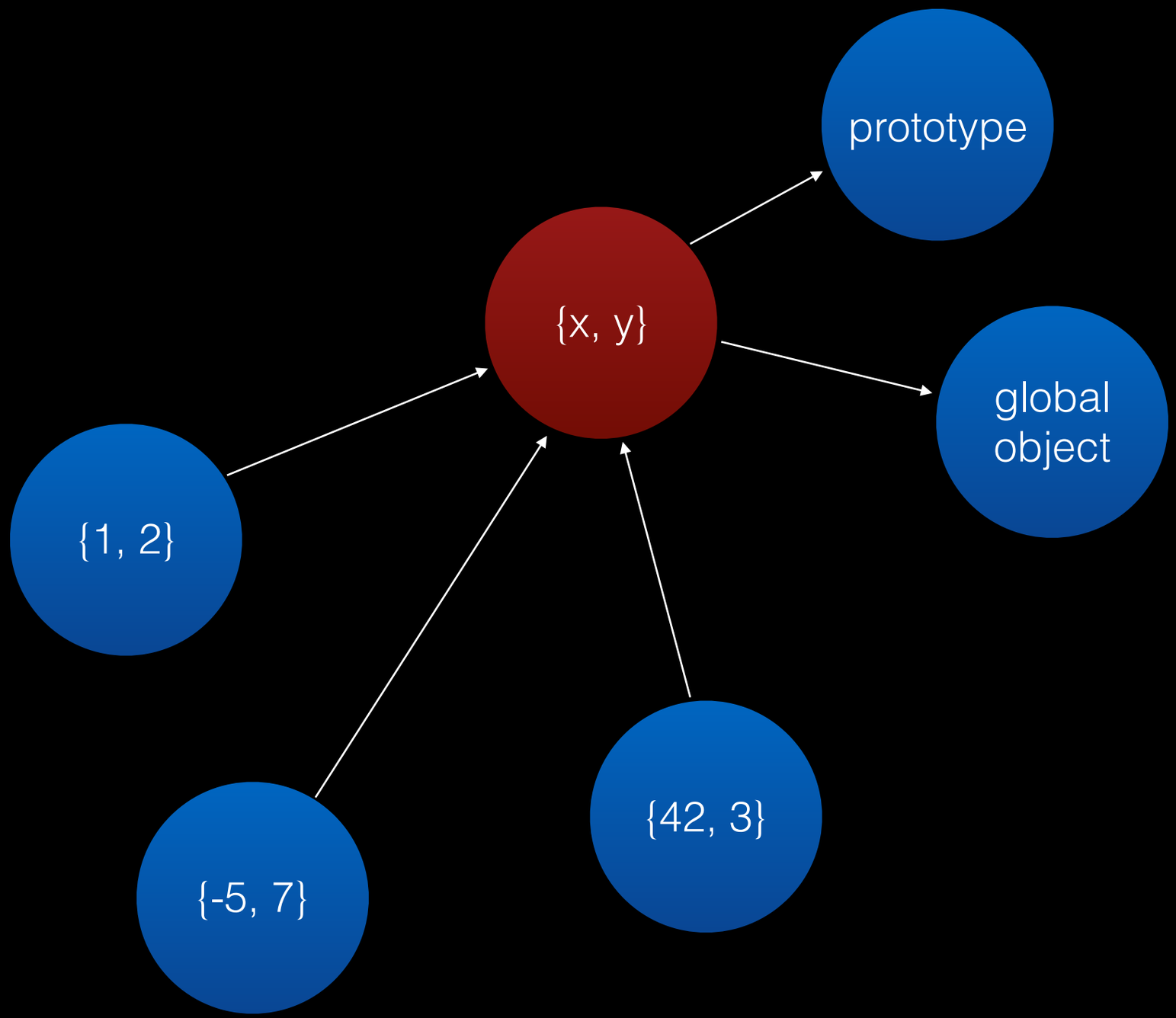
- Transitive reachability is not always enough
- WebKit examples:
  - Type inference
  - Weak map
  - DOM
  - Native code

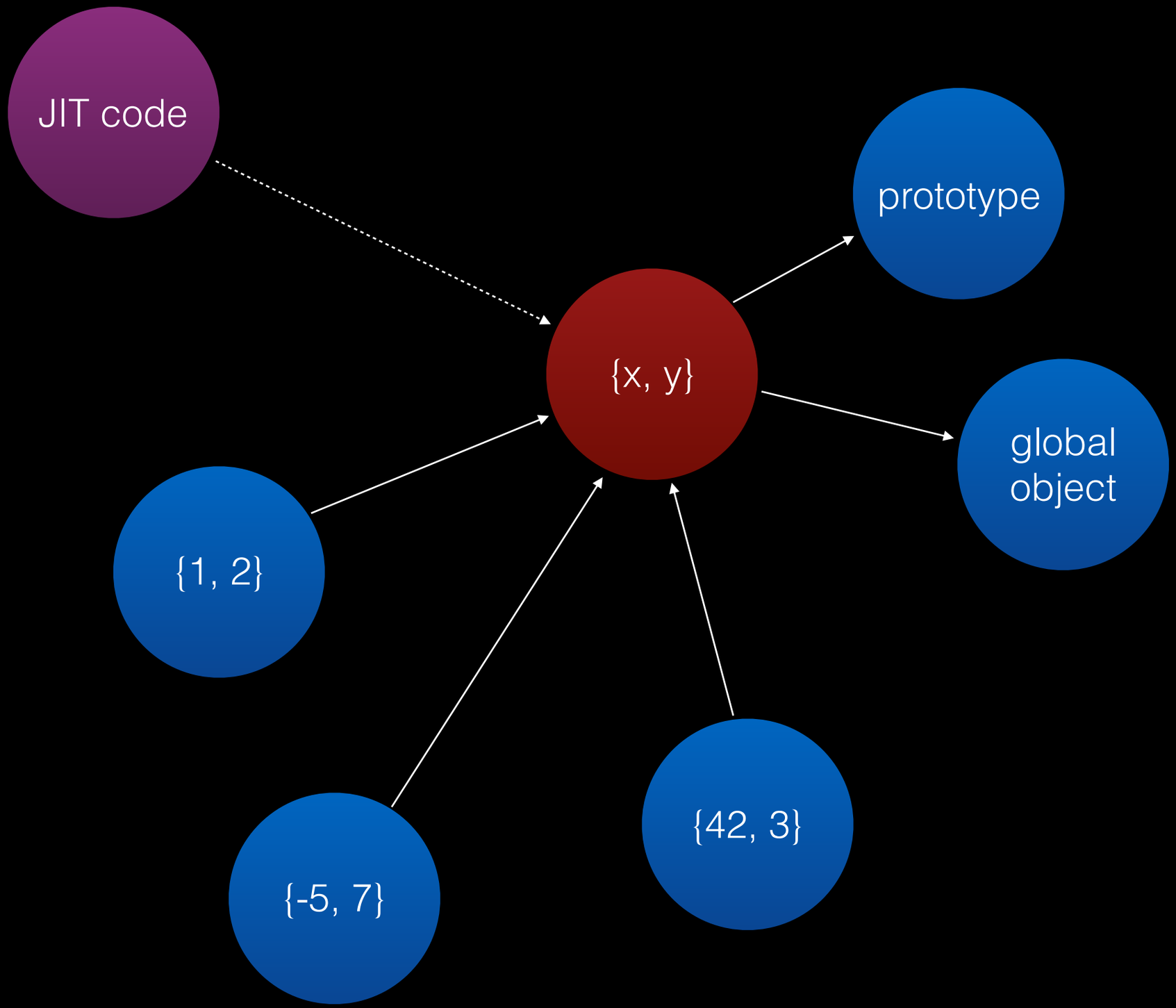
Type Inference

Structure

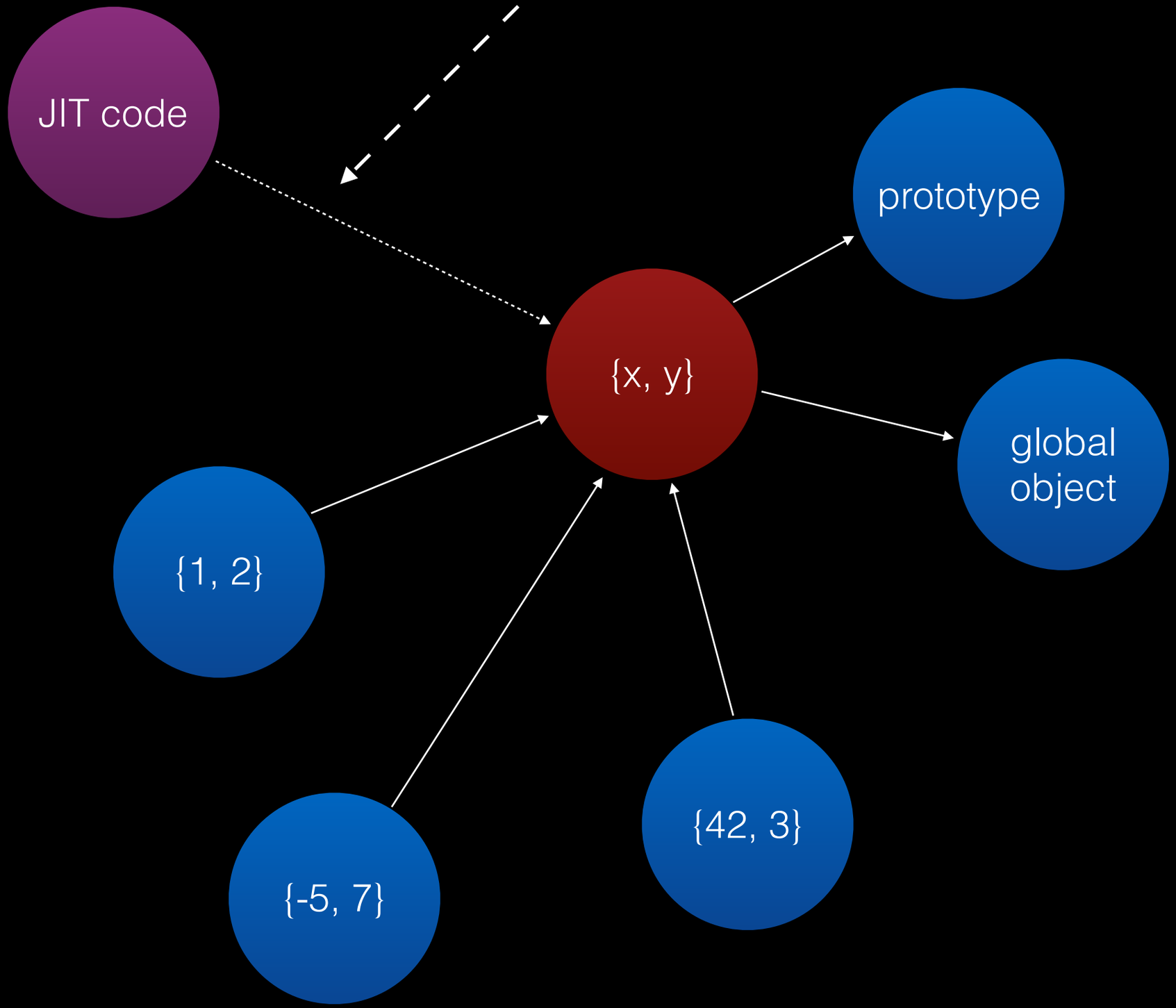


Objects


prototype
{x, y}
global
object
{1, 2}
{-5, 7}
{42, 3}


JIT code
prototype
{x, y}
global object
{1, 2}
{-5, 7}
{42, 3}

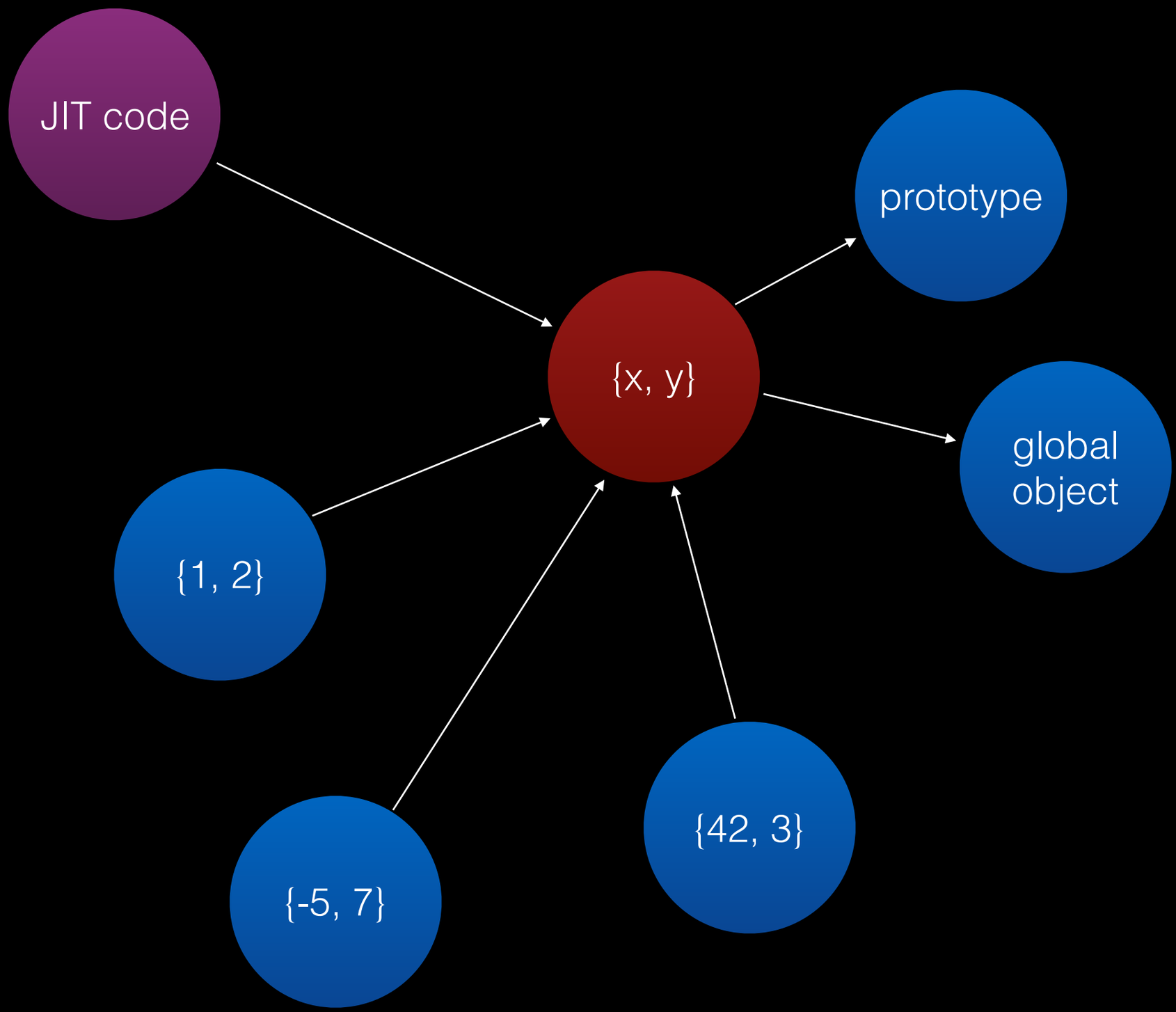
Is this a weak reference?



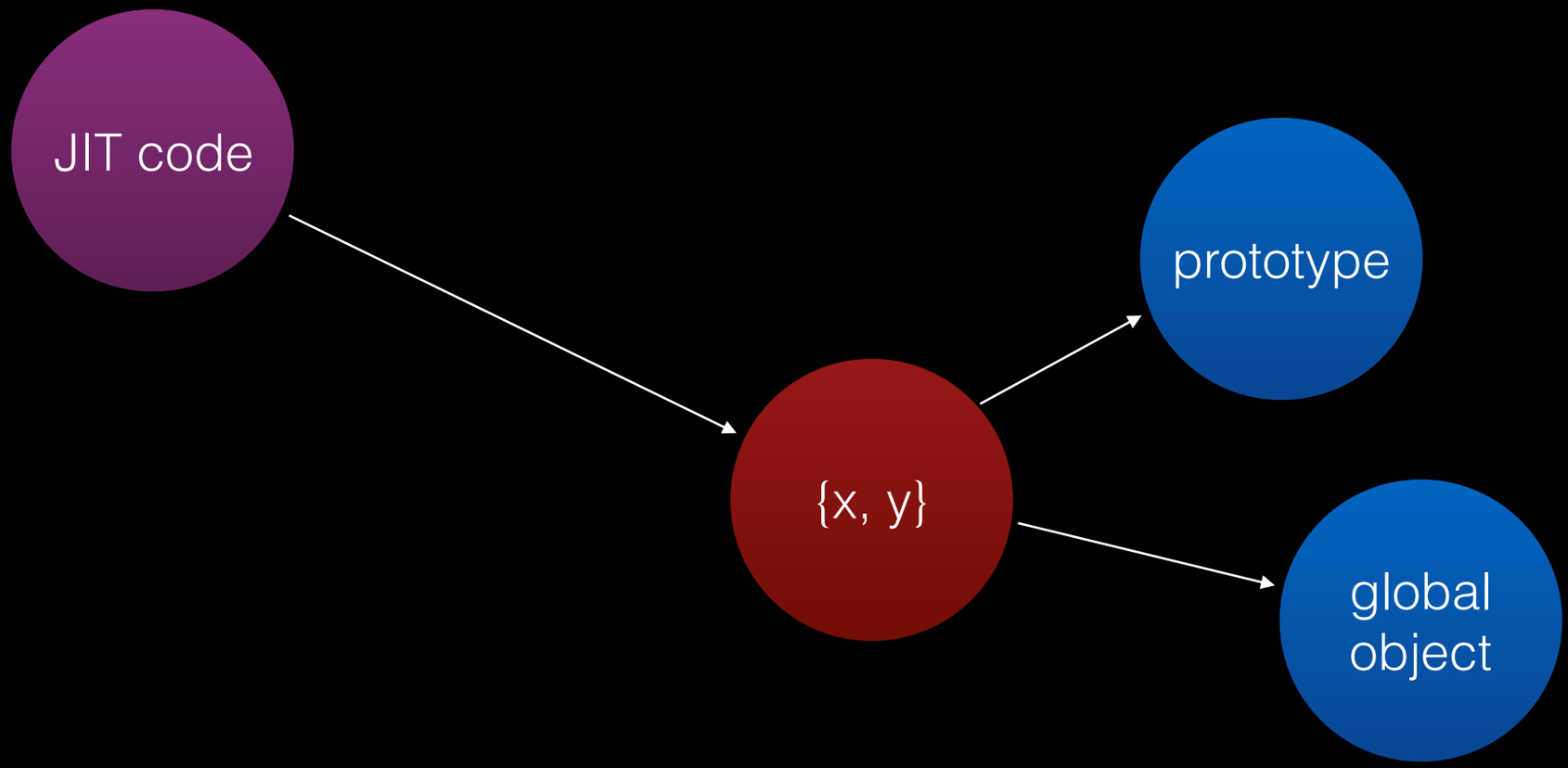
JIT code references a structure

- Strong reference?
- Weak reference?
- Marking constraint?

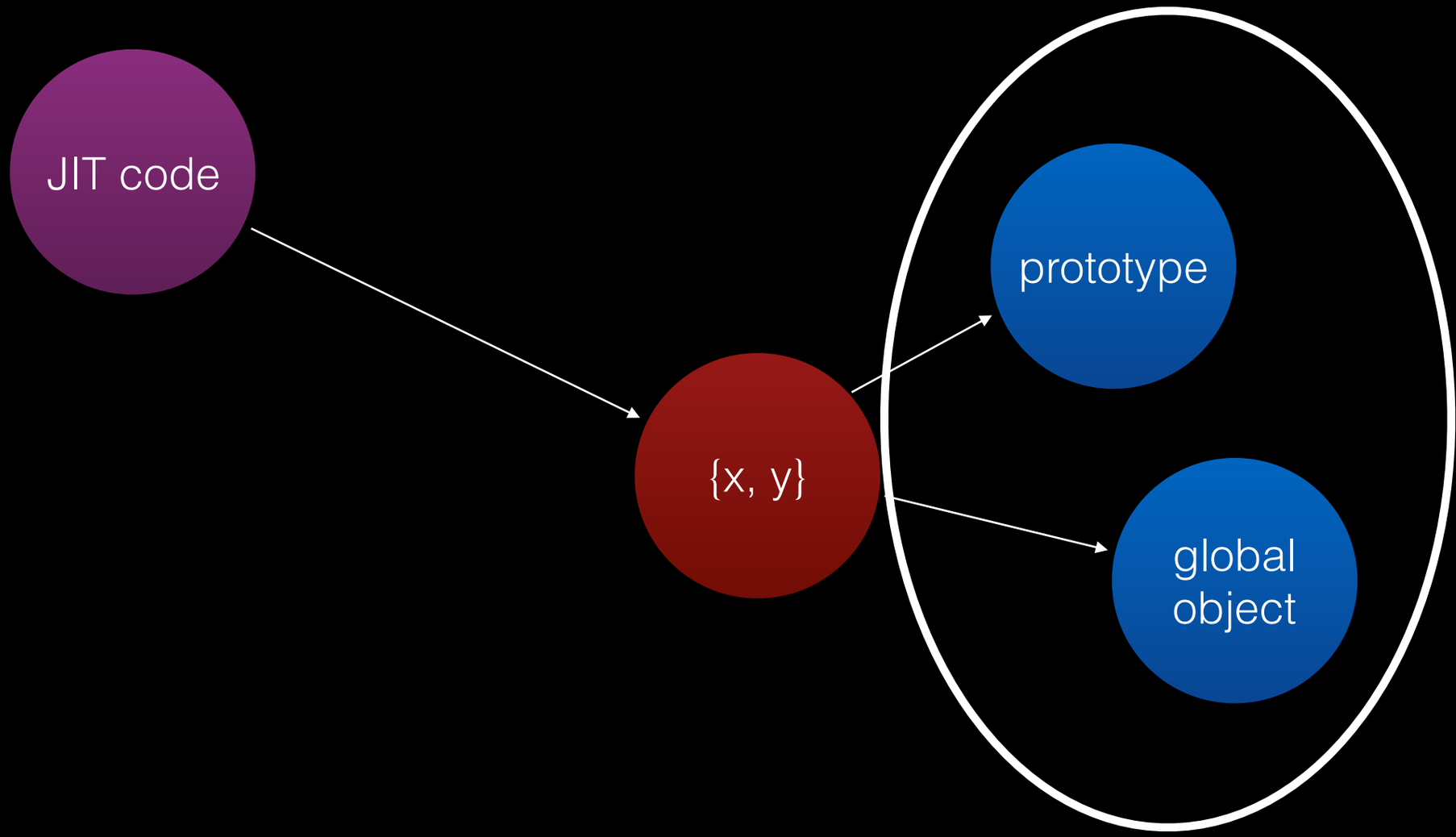
Strong reference?



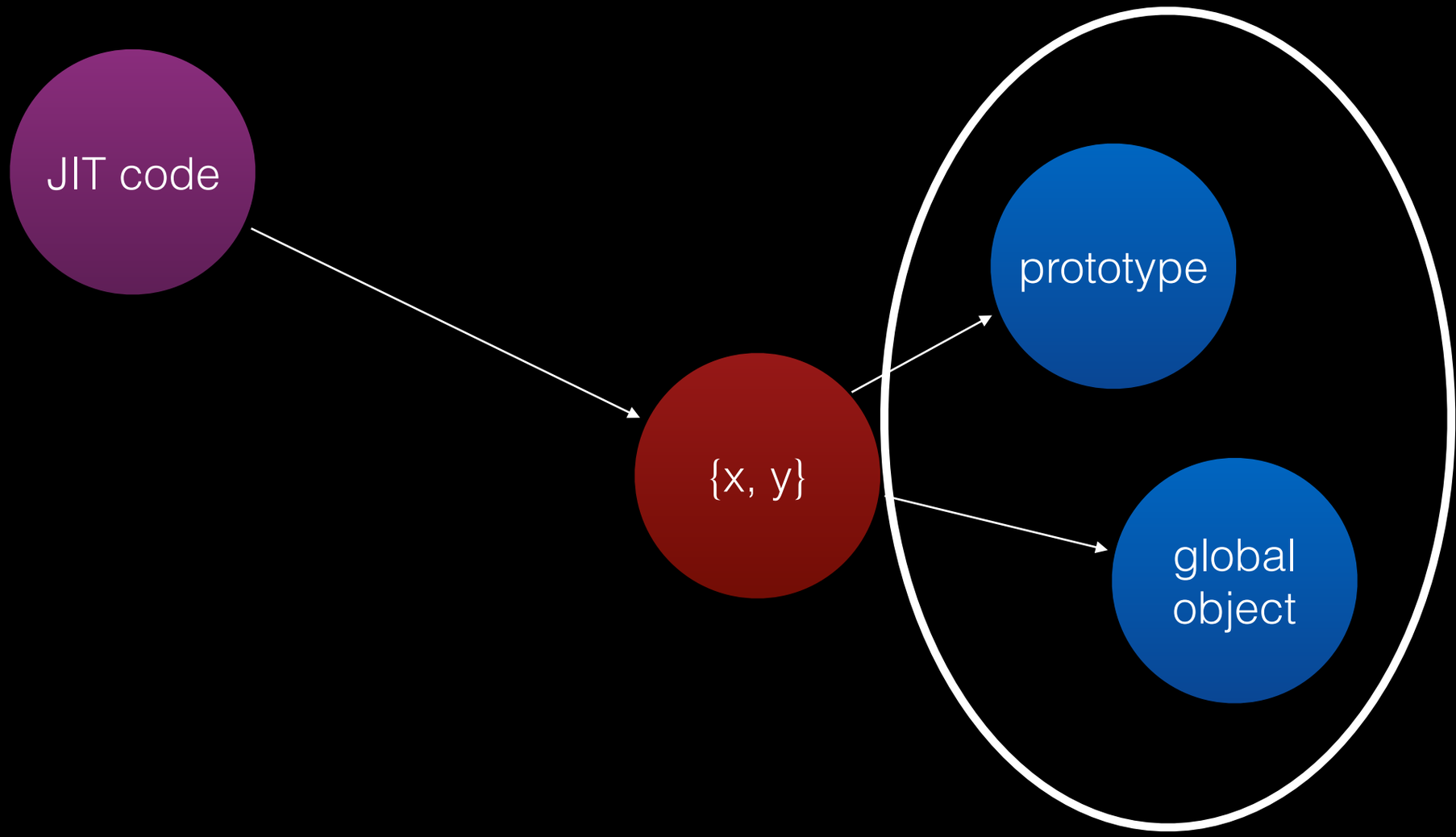
Strong reference?



Strong reference?

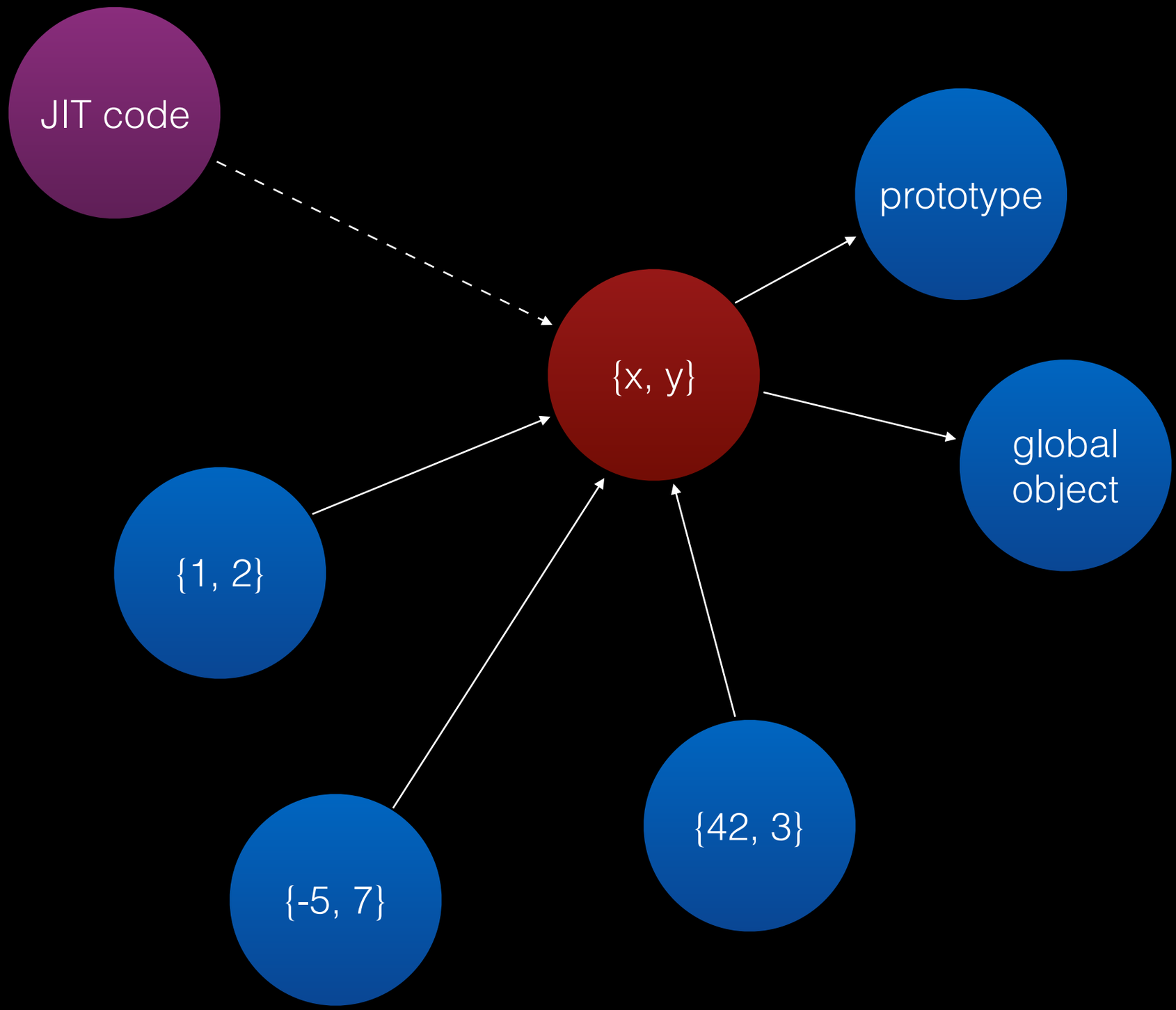


Strong reference?

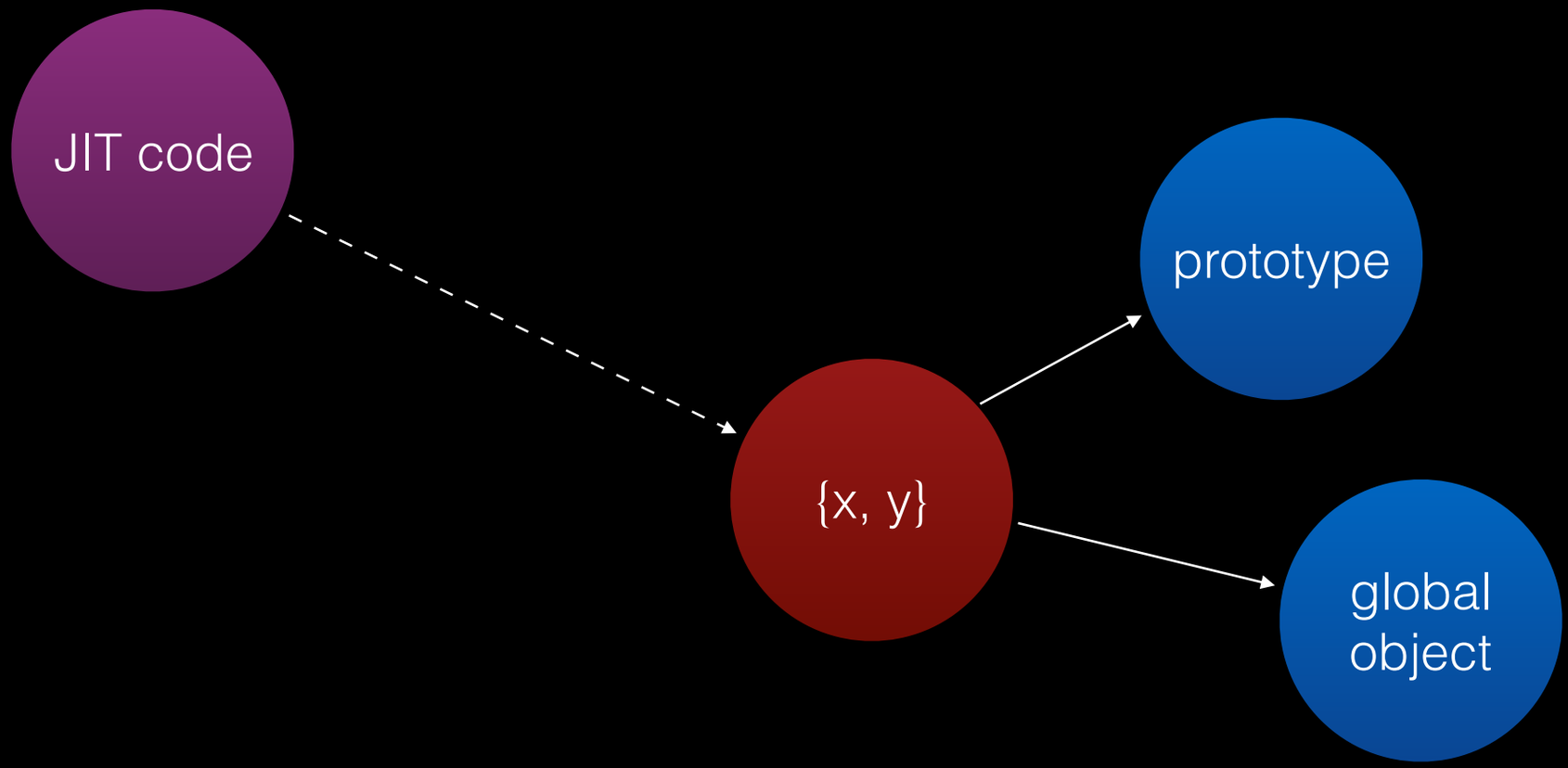


so many leaks

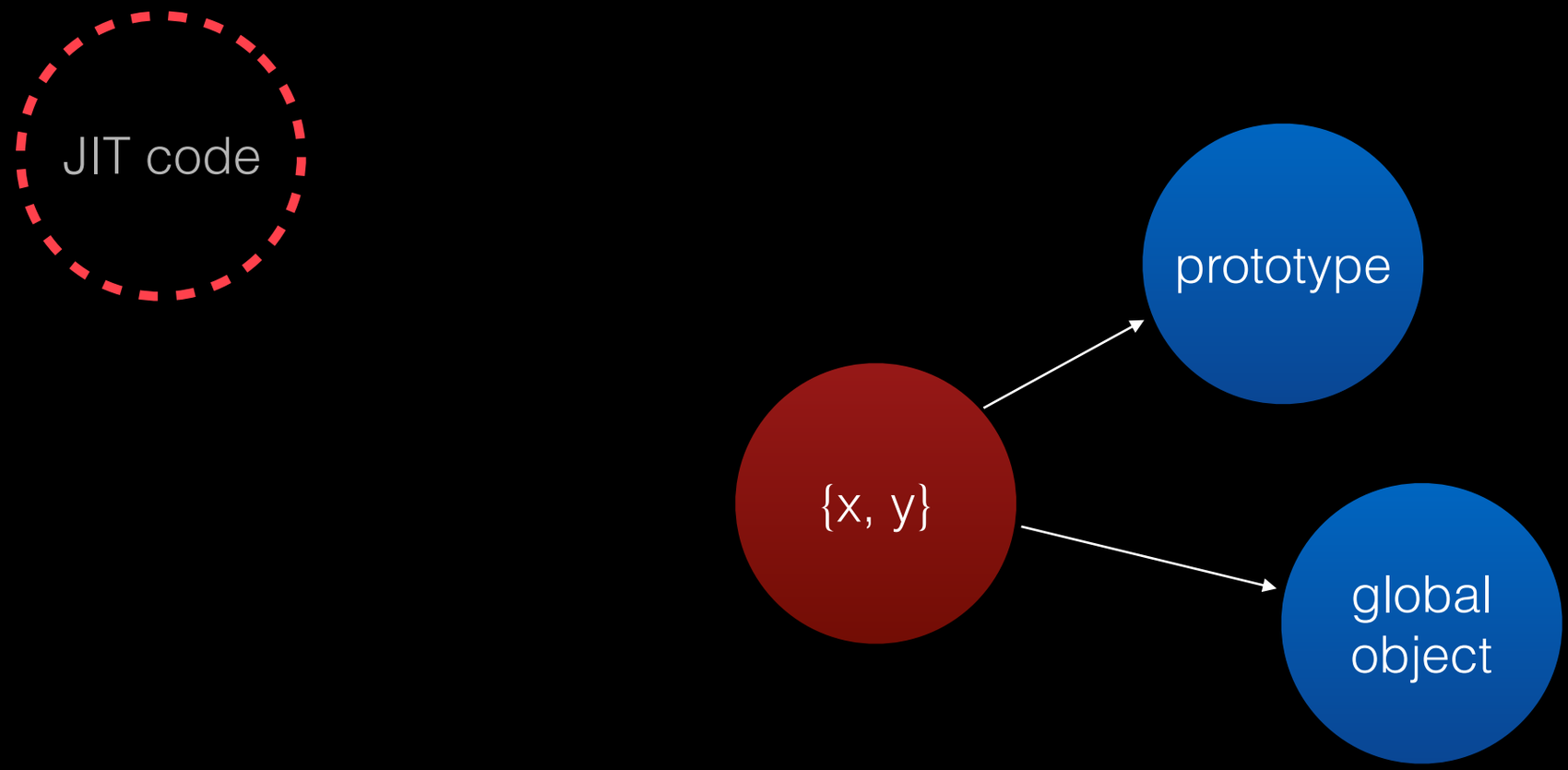
Weak reference?



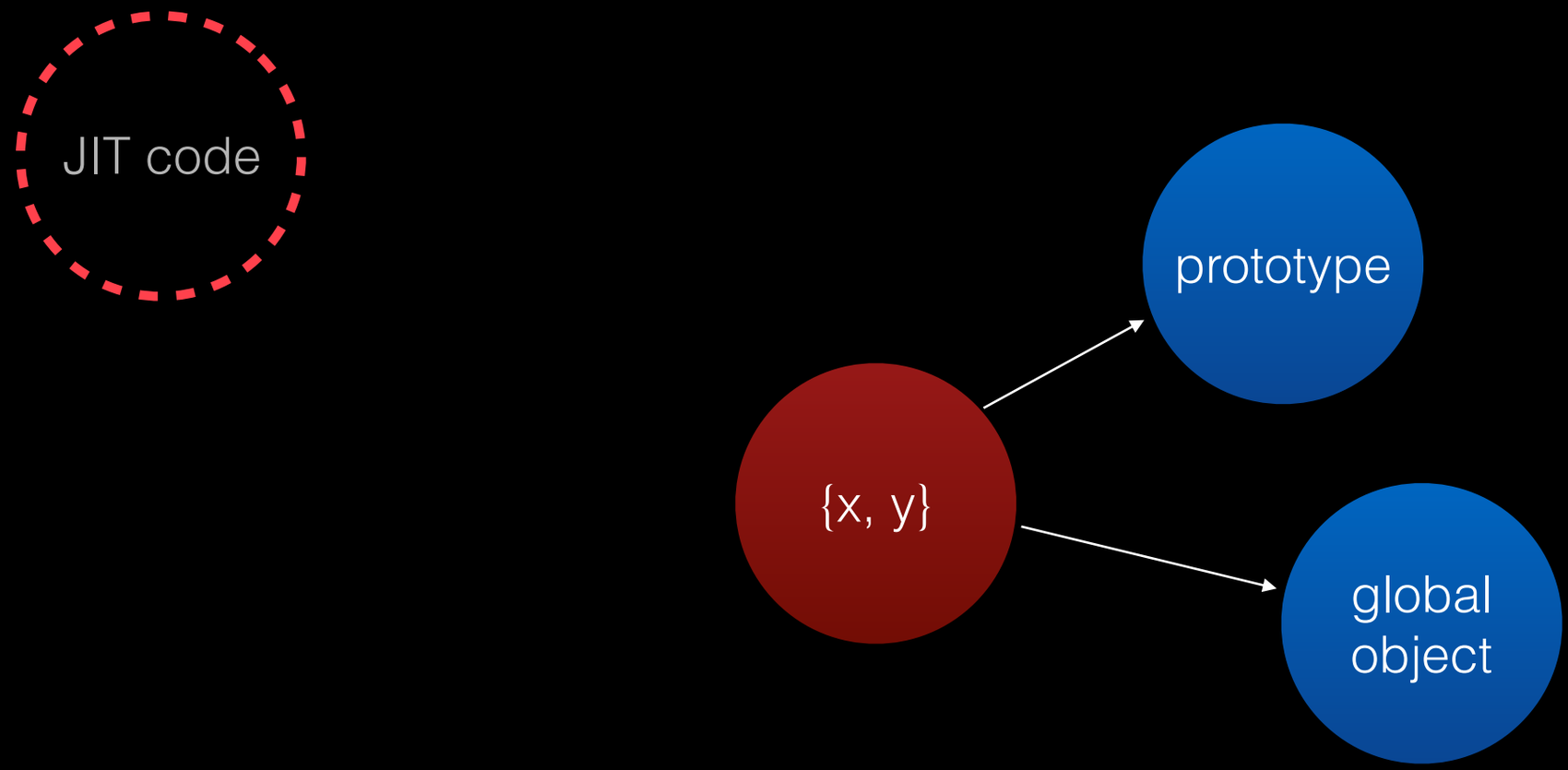
Weak reference?



Weak reference?

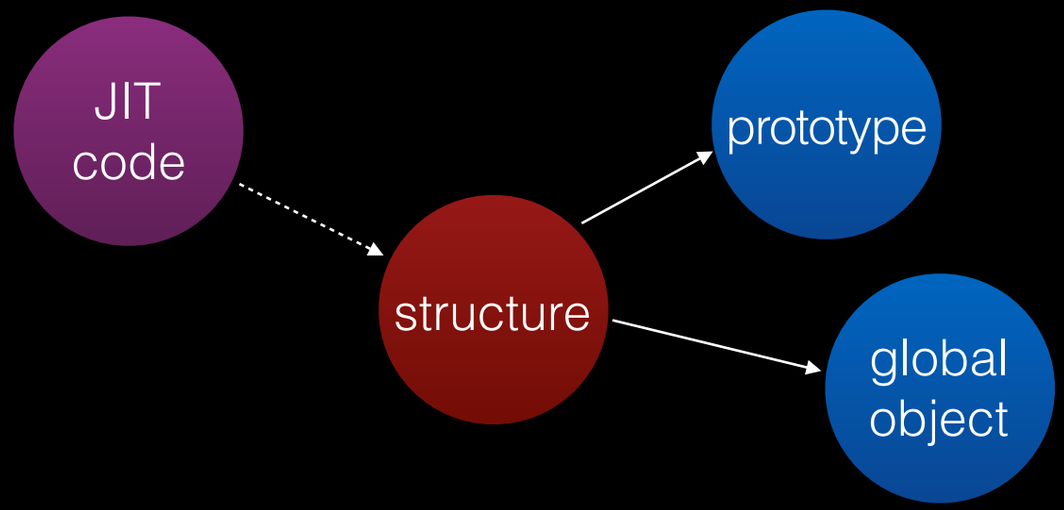


Weak reference?

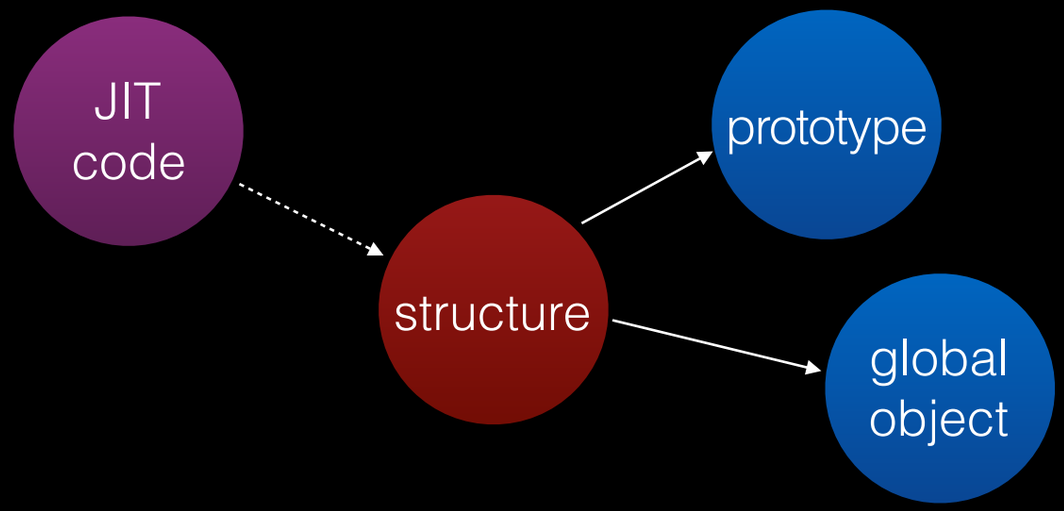


recomp storm

Marking Constraint



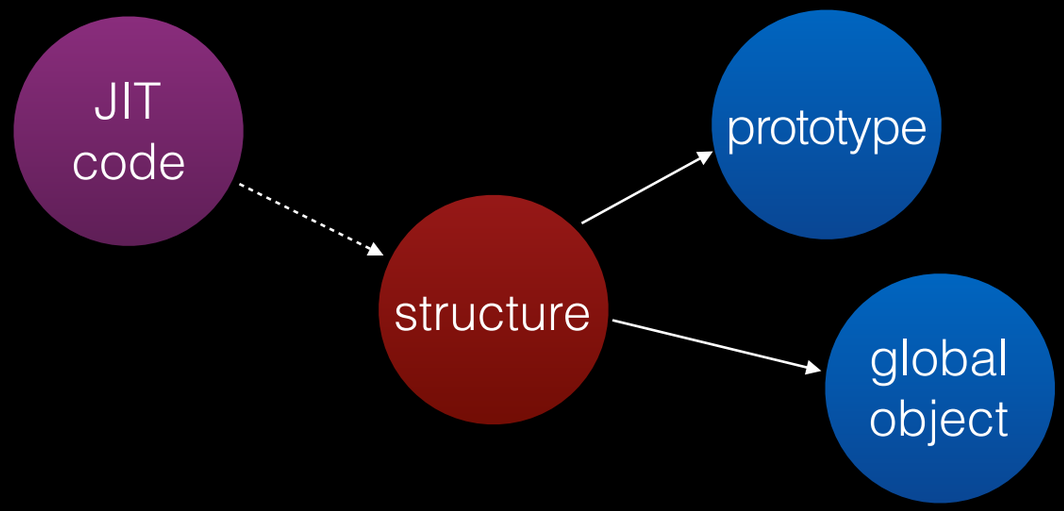
Marking Constraint



- JIT code references the structure weakly.

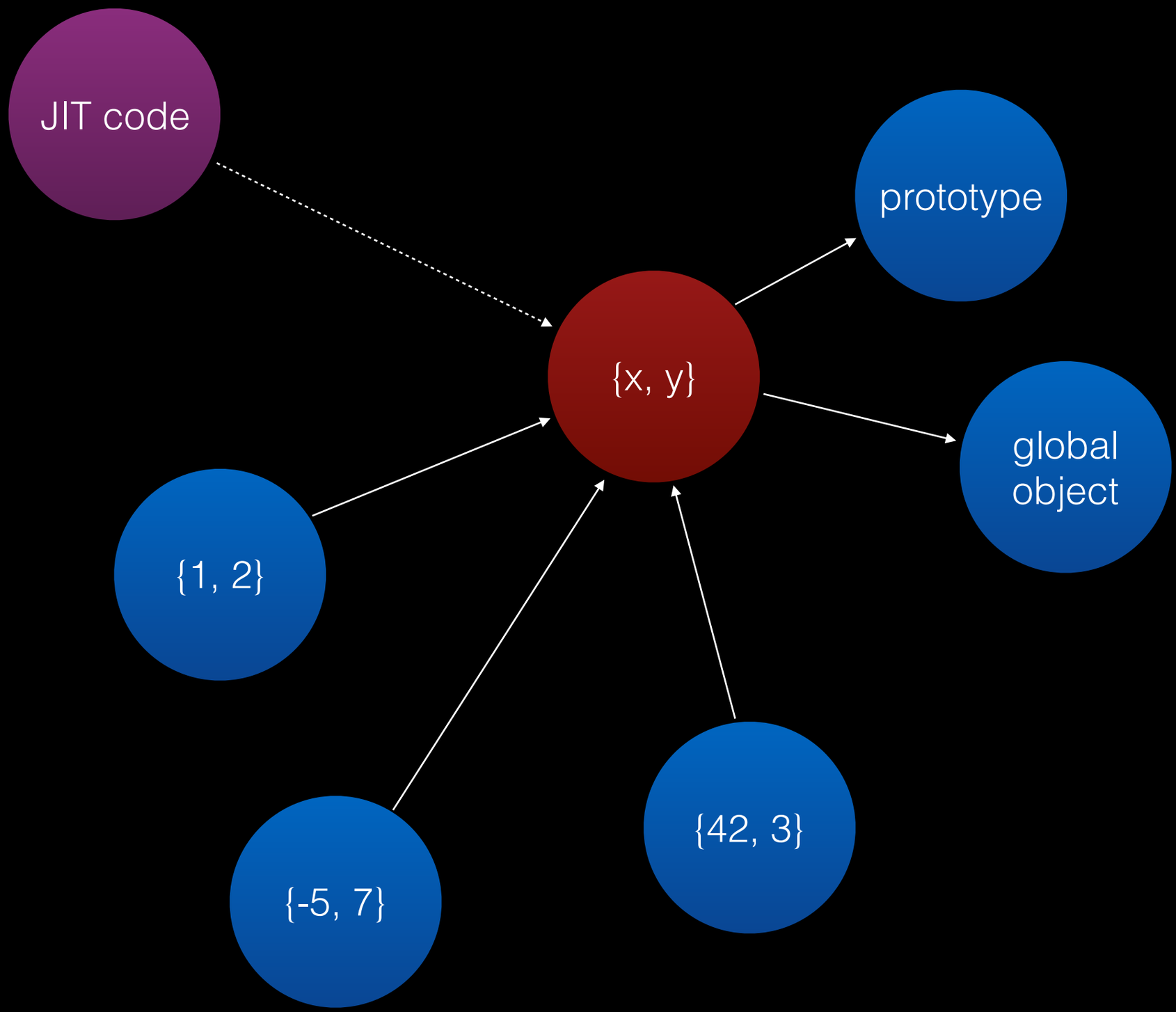
Marking Constraint

```
if (isMarked(structure->globalObject())
    && isMarked(structure->storedPrototype()))
    mark(structure);
```

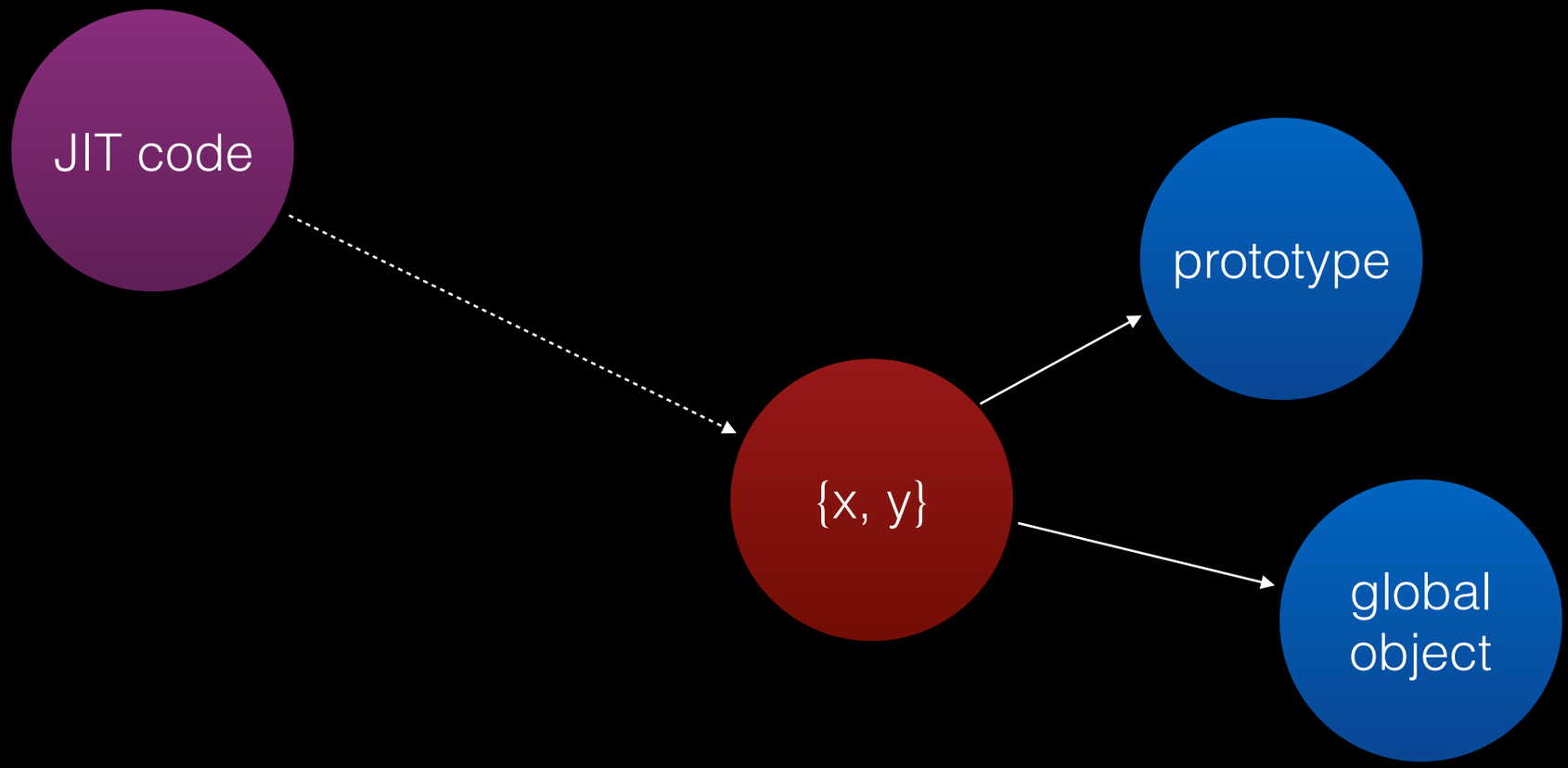


- JIT code references the structure weakly.
- JIT code also registers the above marking constraint.

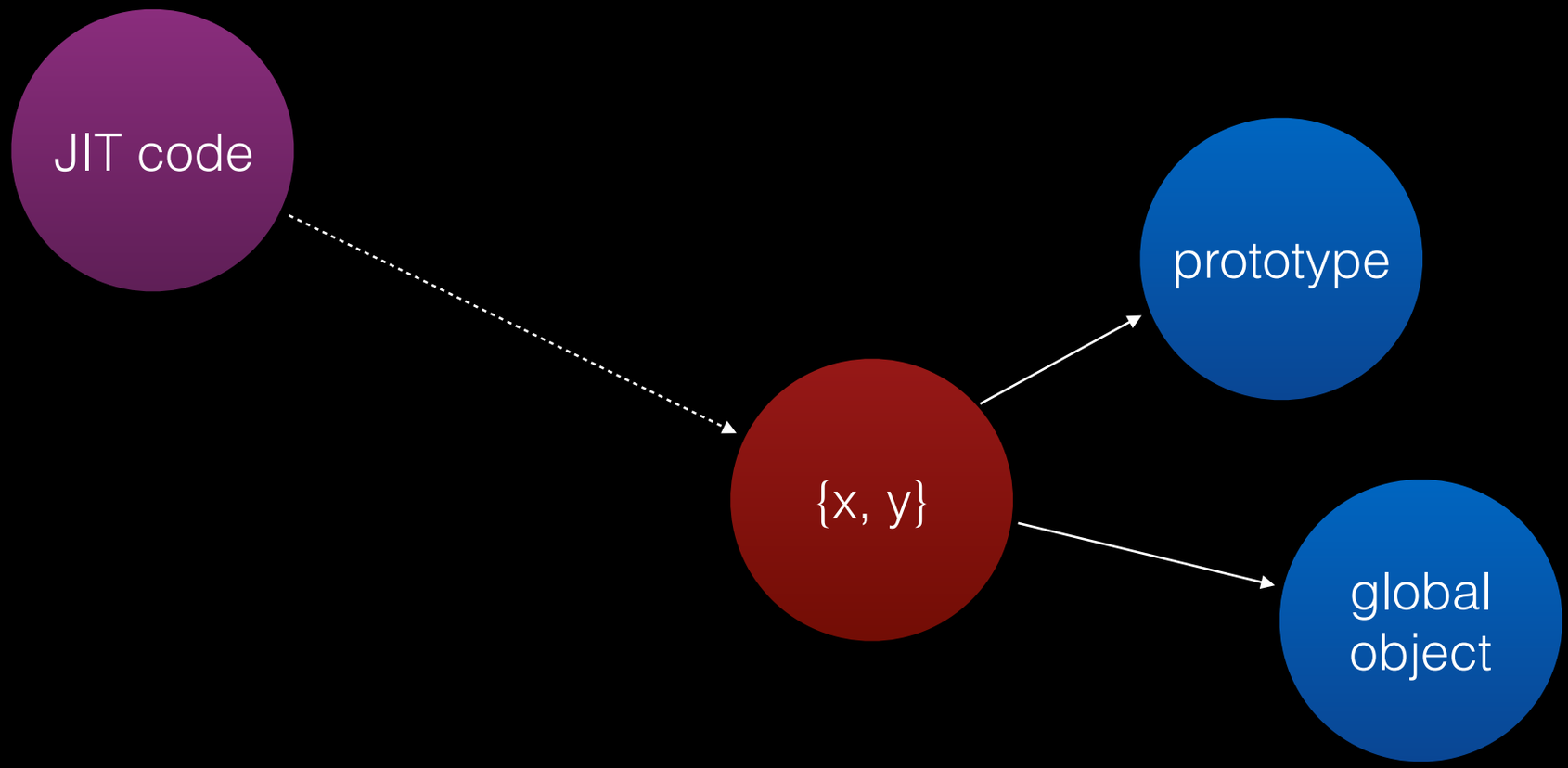
Marking Constraint!



Marking Constraint!

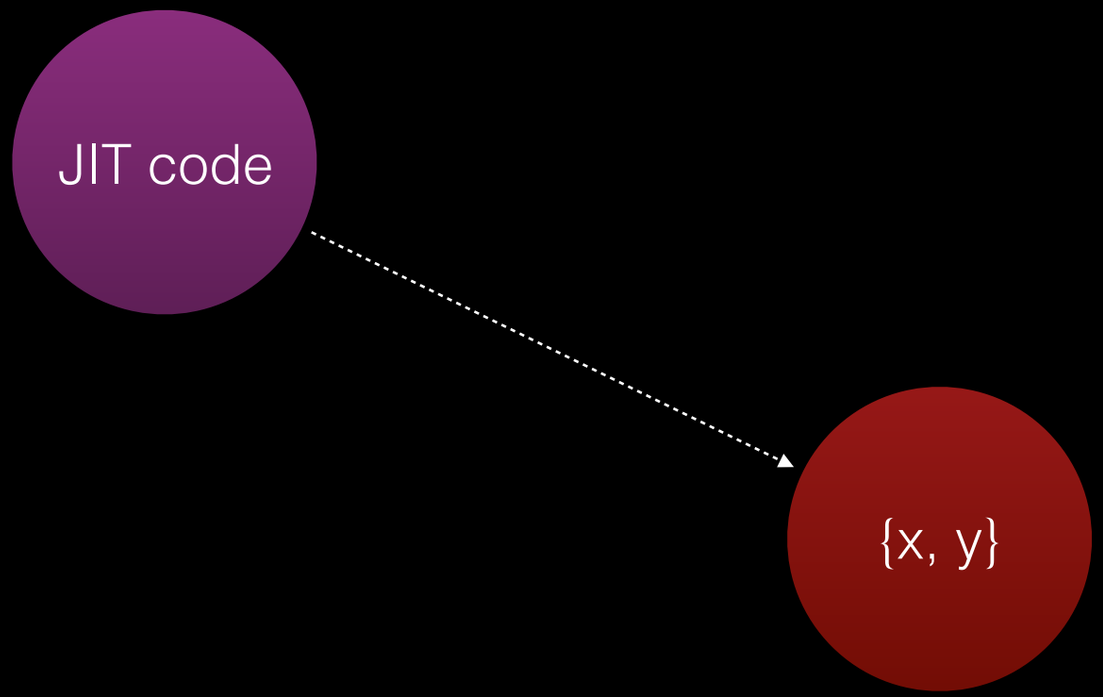


Marking Constraint!

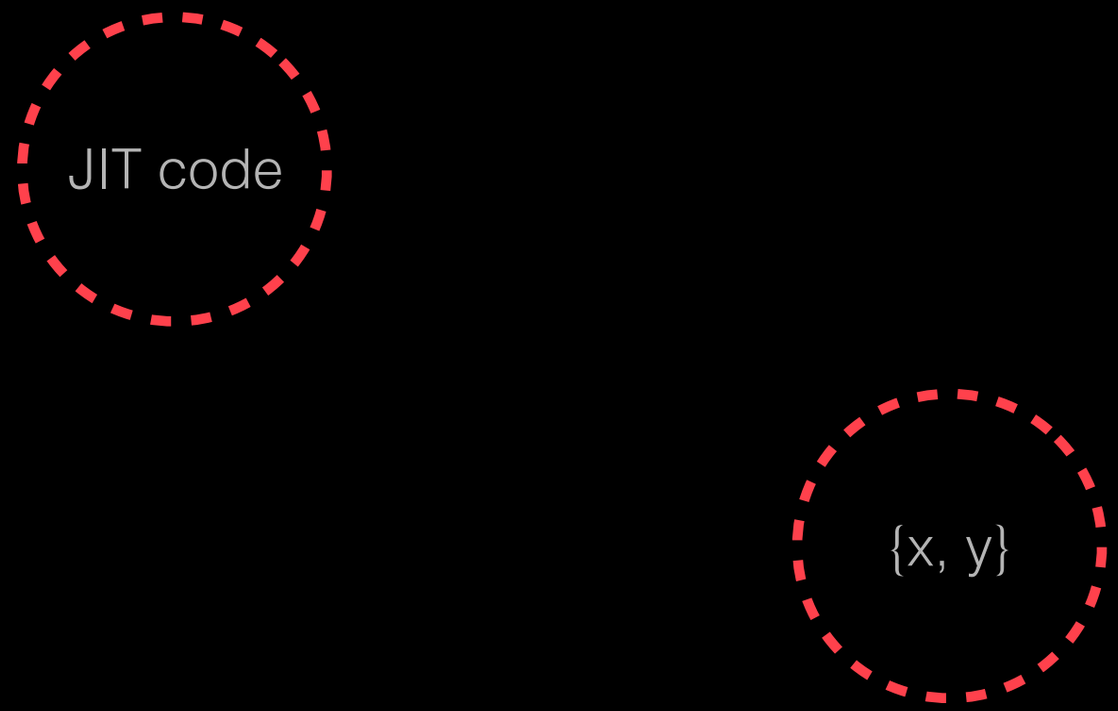


It’s cool - the prototype and global object are long-lived.

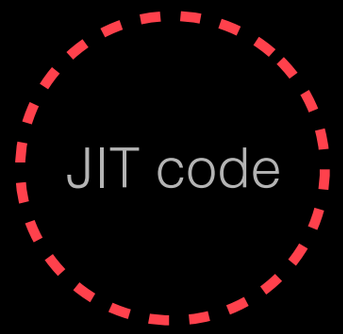
Marking Constraint!

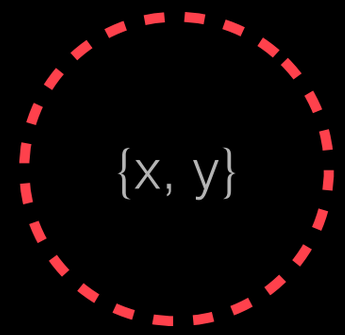


Marking Constraint!



Marking Constraint!





We want the JIT code to die in this case.

Marking Constraint!

- If the objects that use the structure die, then:
  - Keep structure alive if the user objects it points to are alive anyway.
  - Kill the structure (and the JIT code) if keeping it alive would not be safe-for-space.

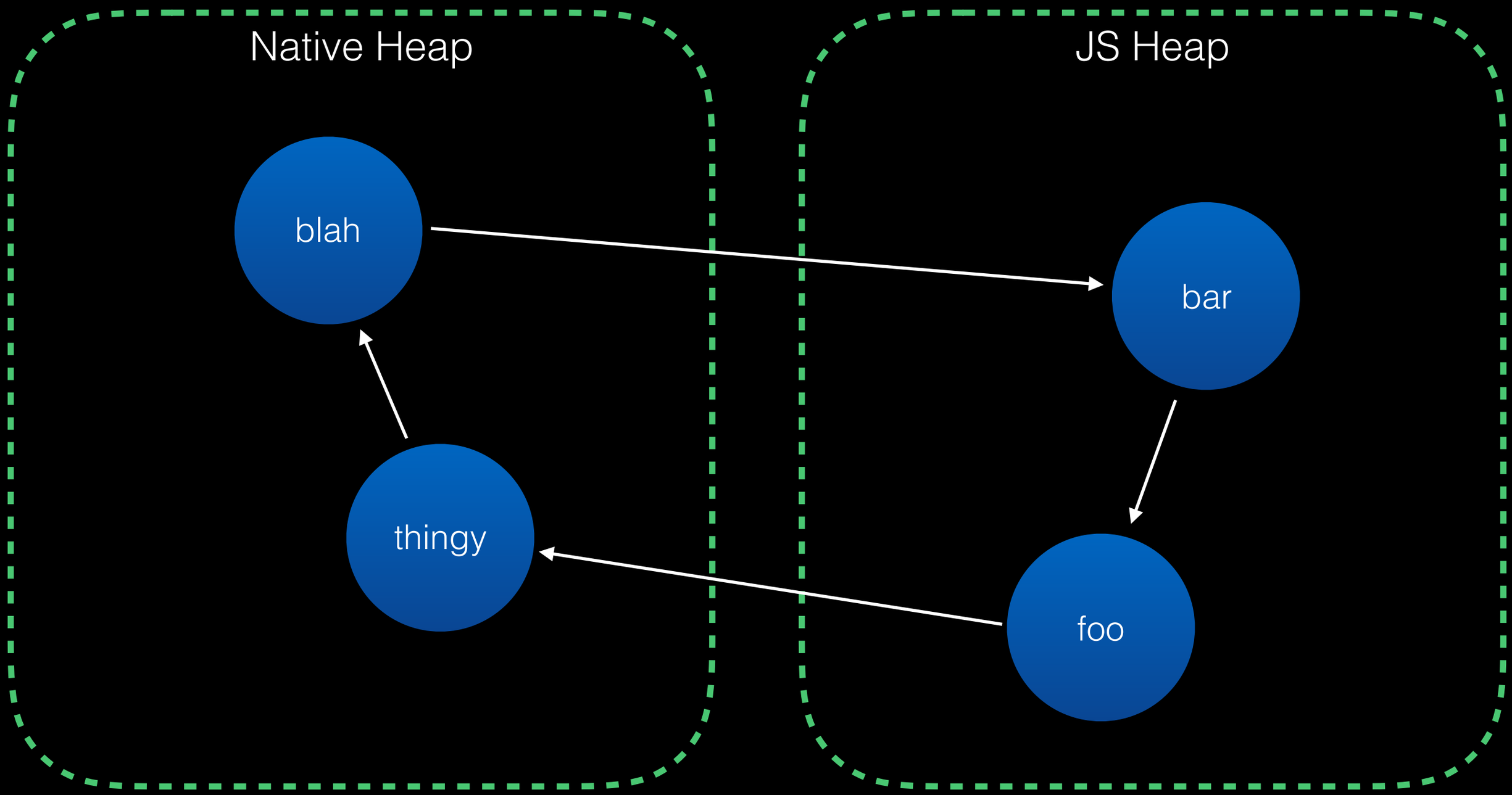
Marking Constraints

- Constraints can query which objects are marked.
- Constraints can mark objects.
- GC executes constraints to fixpoint.

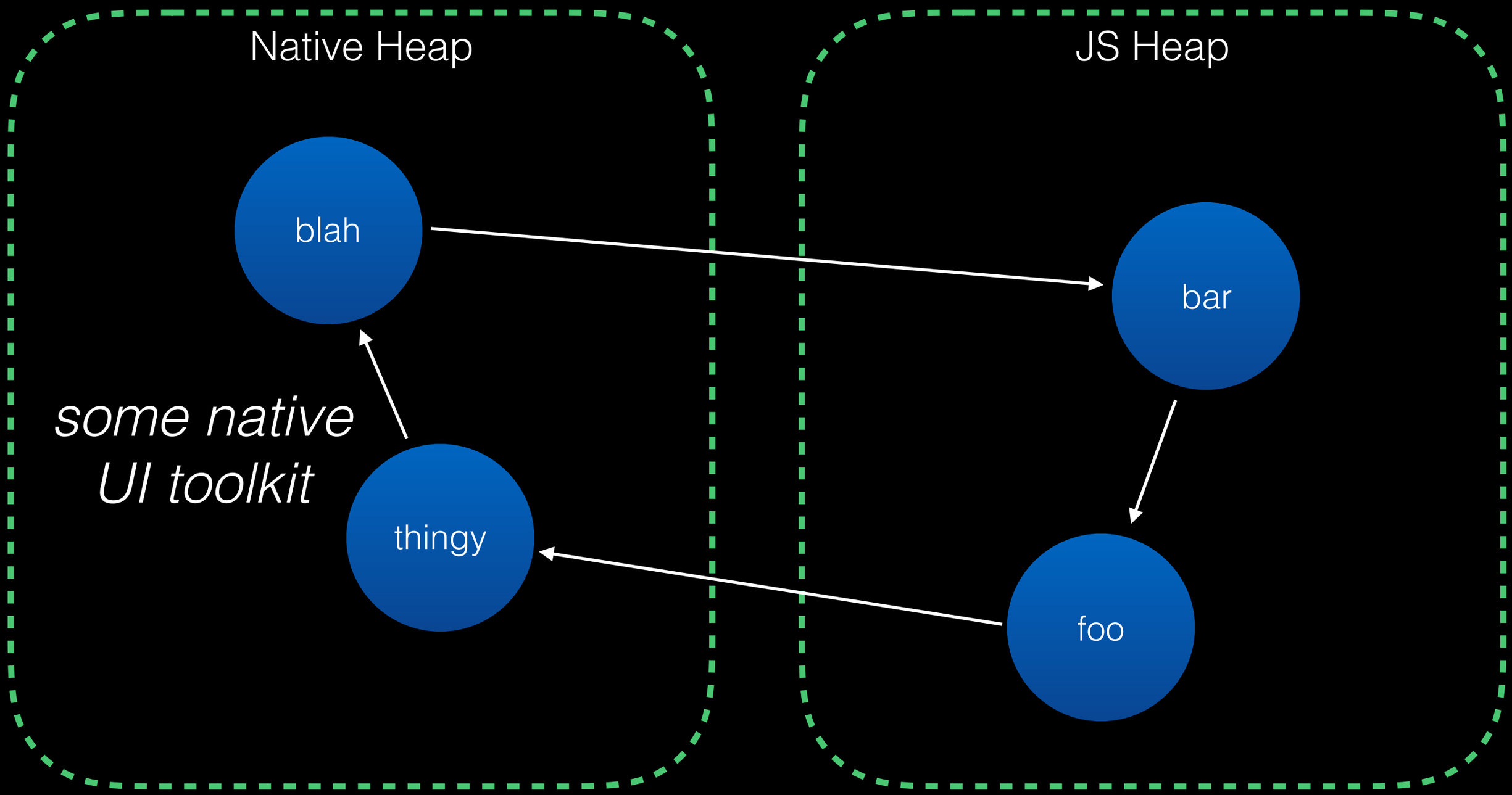
Native Code

Including the DOM

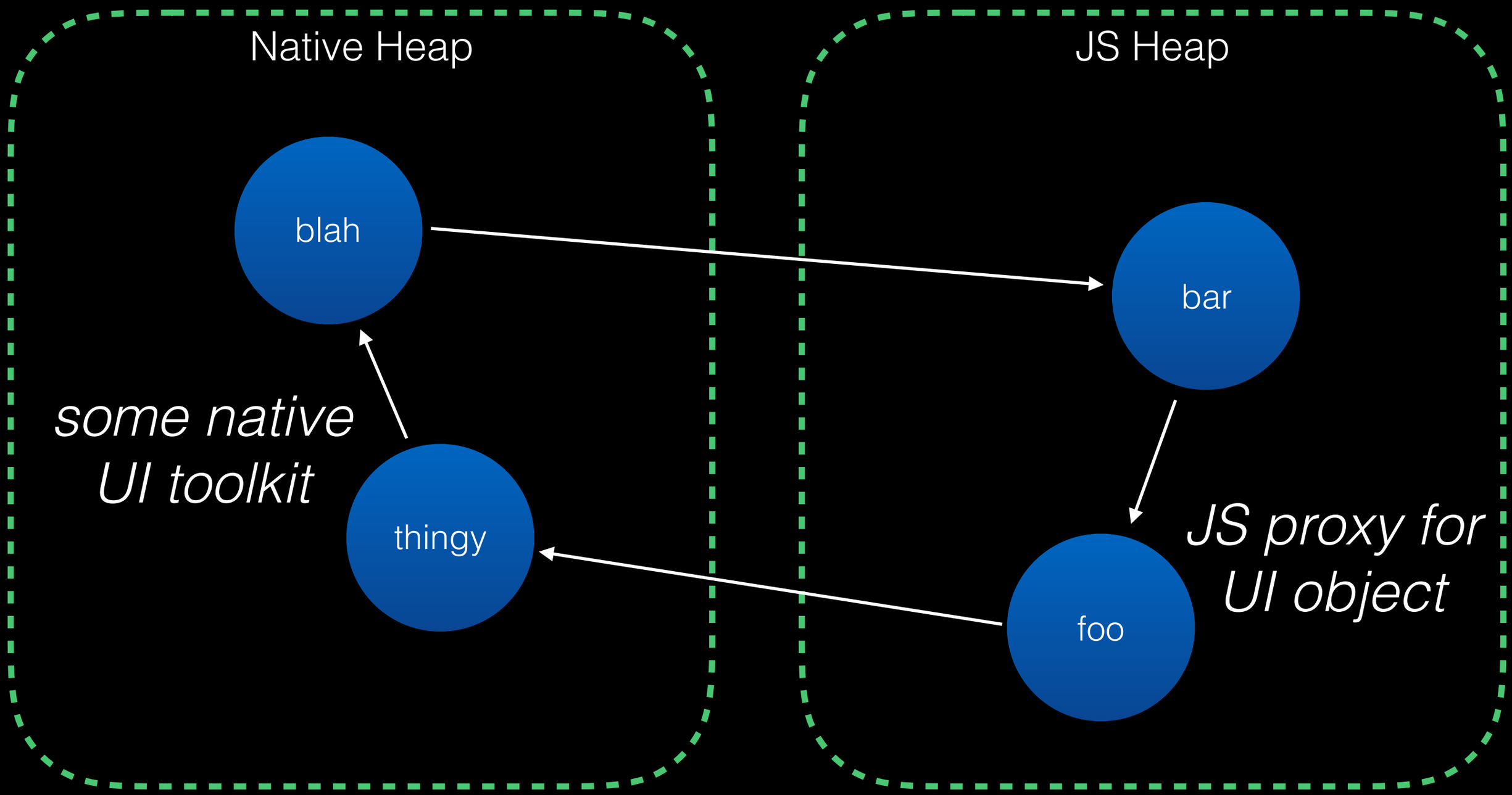
Native Code



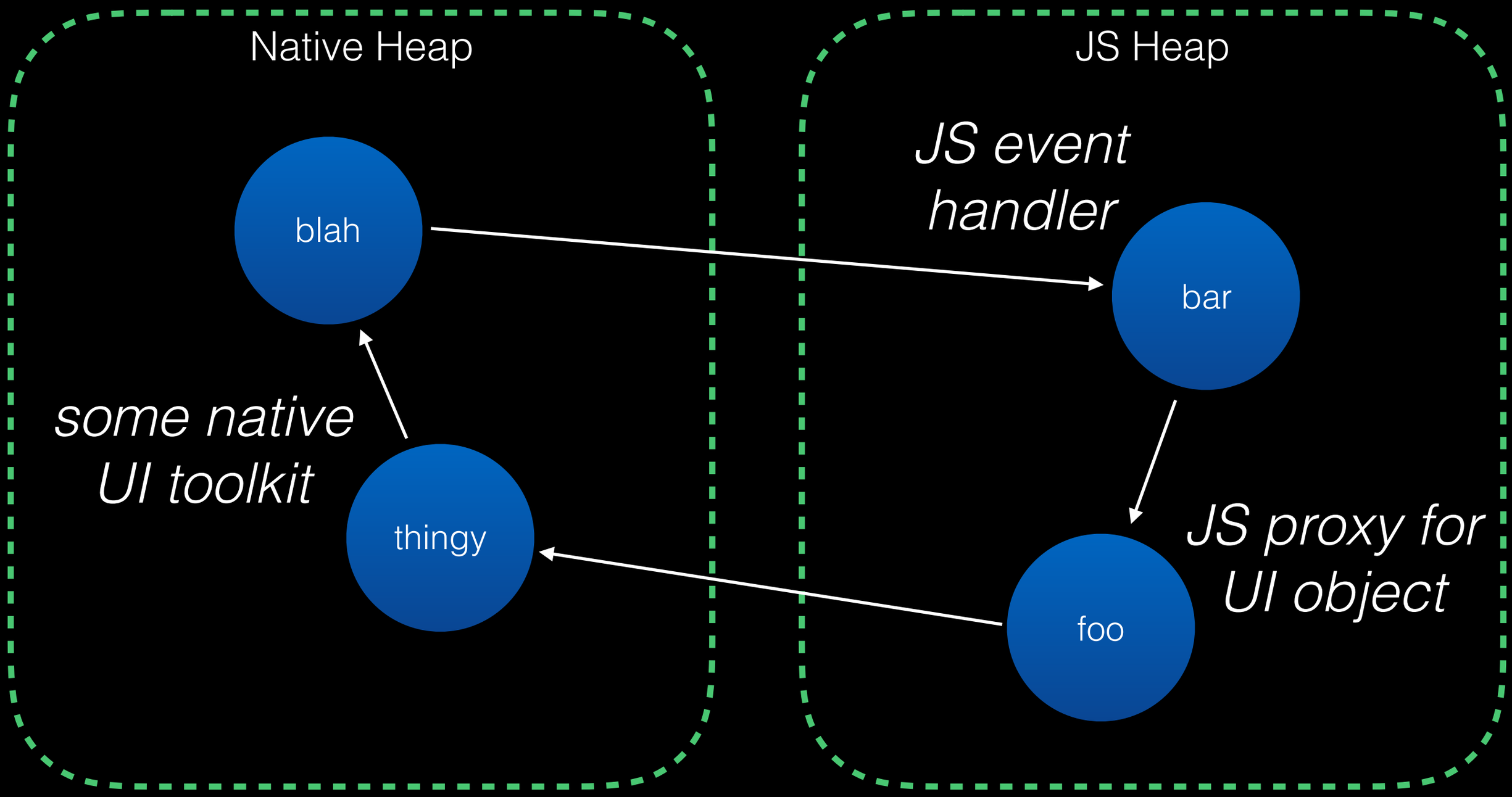
Native Code


Native Heap
JS Heap
blah
bar
some native
UI toolkit
thingy
foo

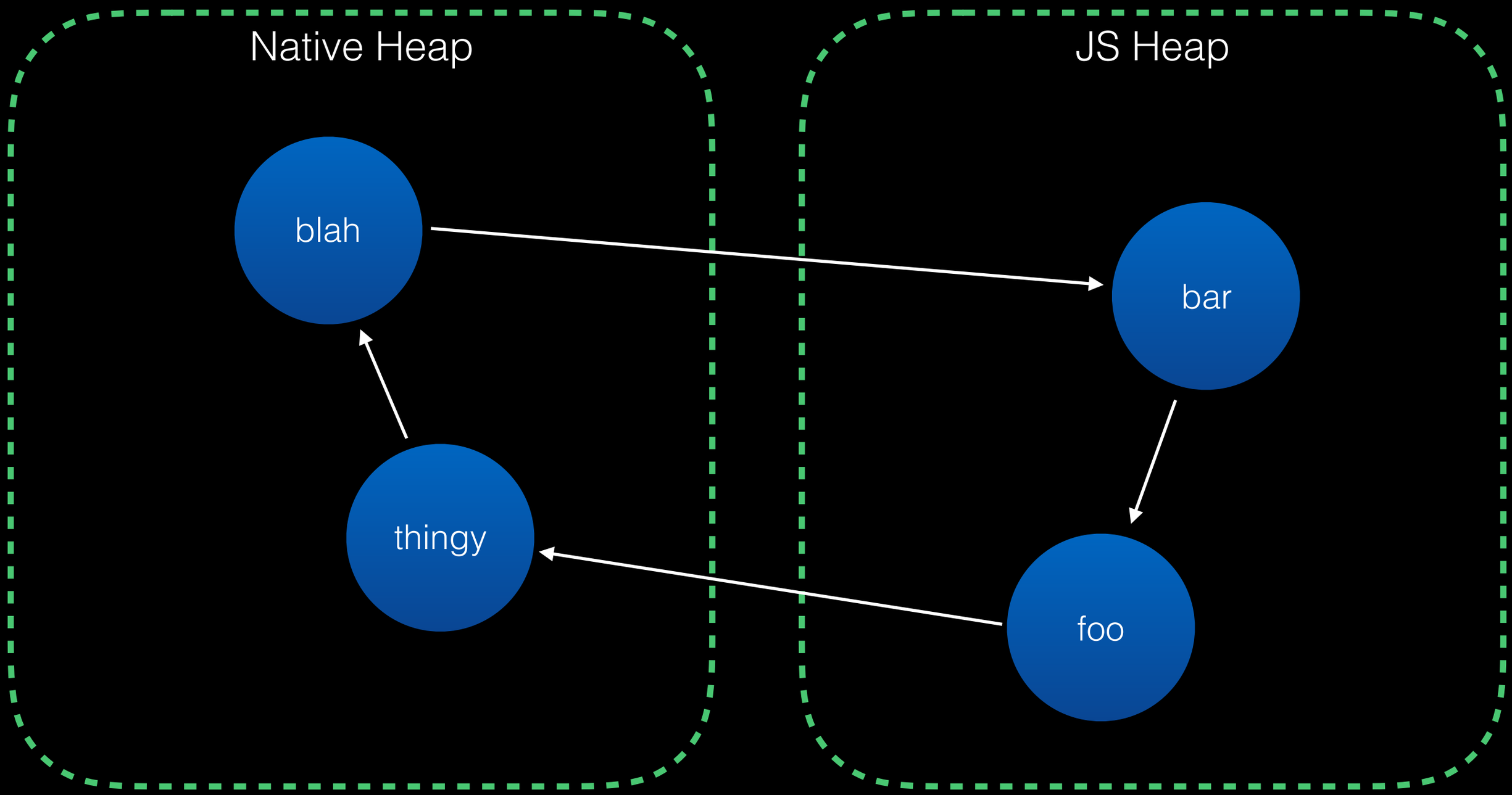
Native Code


Native Heap
JS Heap
blah
bar
some native
UI toolkit
thingy
foo
JS proxy for
UI object

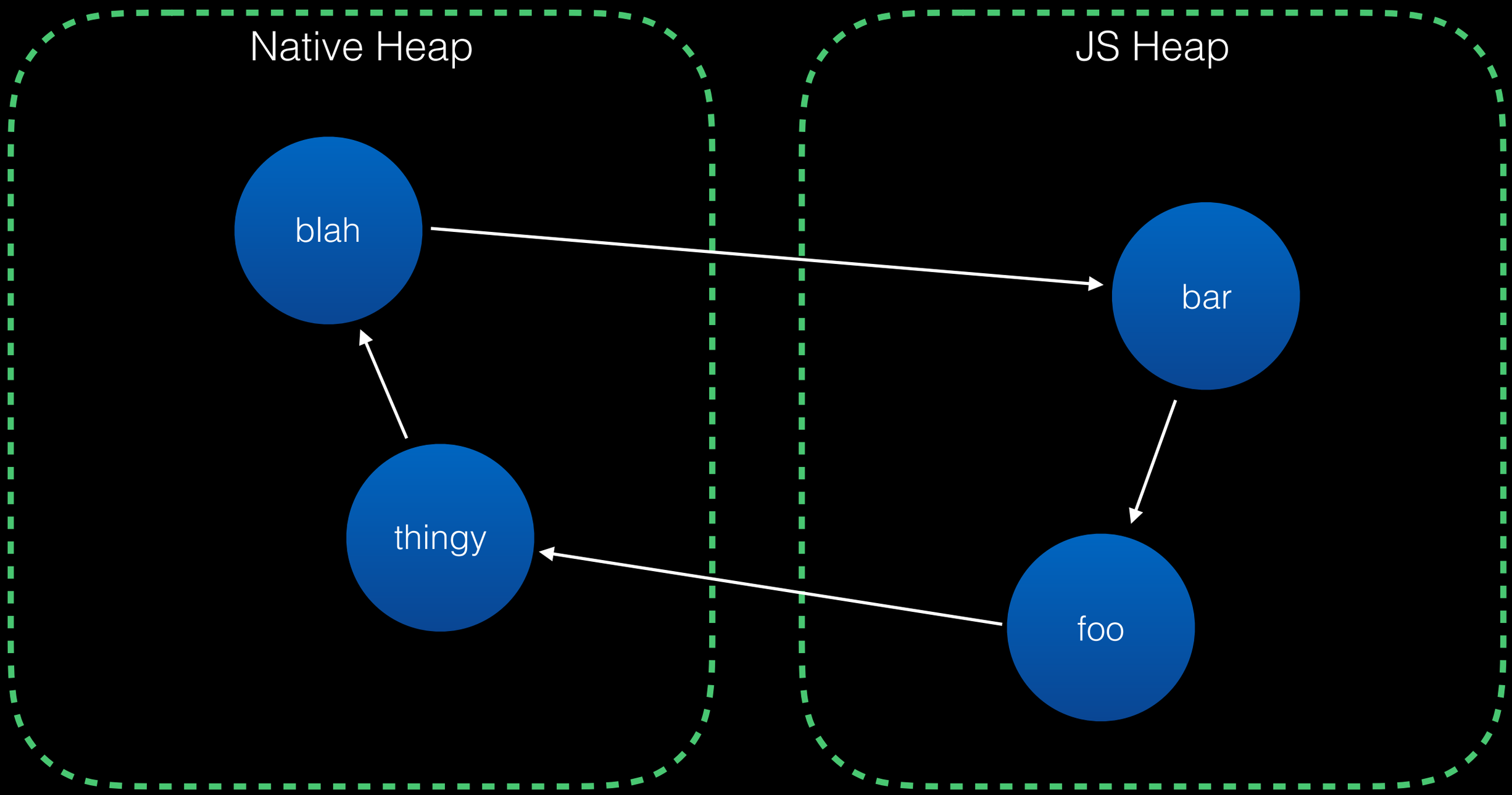
Native Code


Native Heap
JS Heap
JS event handler
blah
bar
some native UI toolkit
thingy
JS proxy for UI object
foo

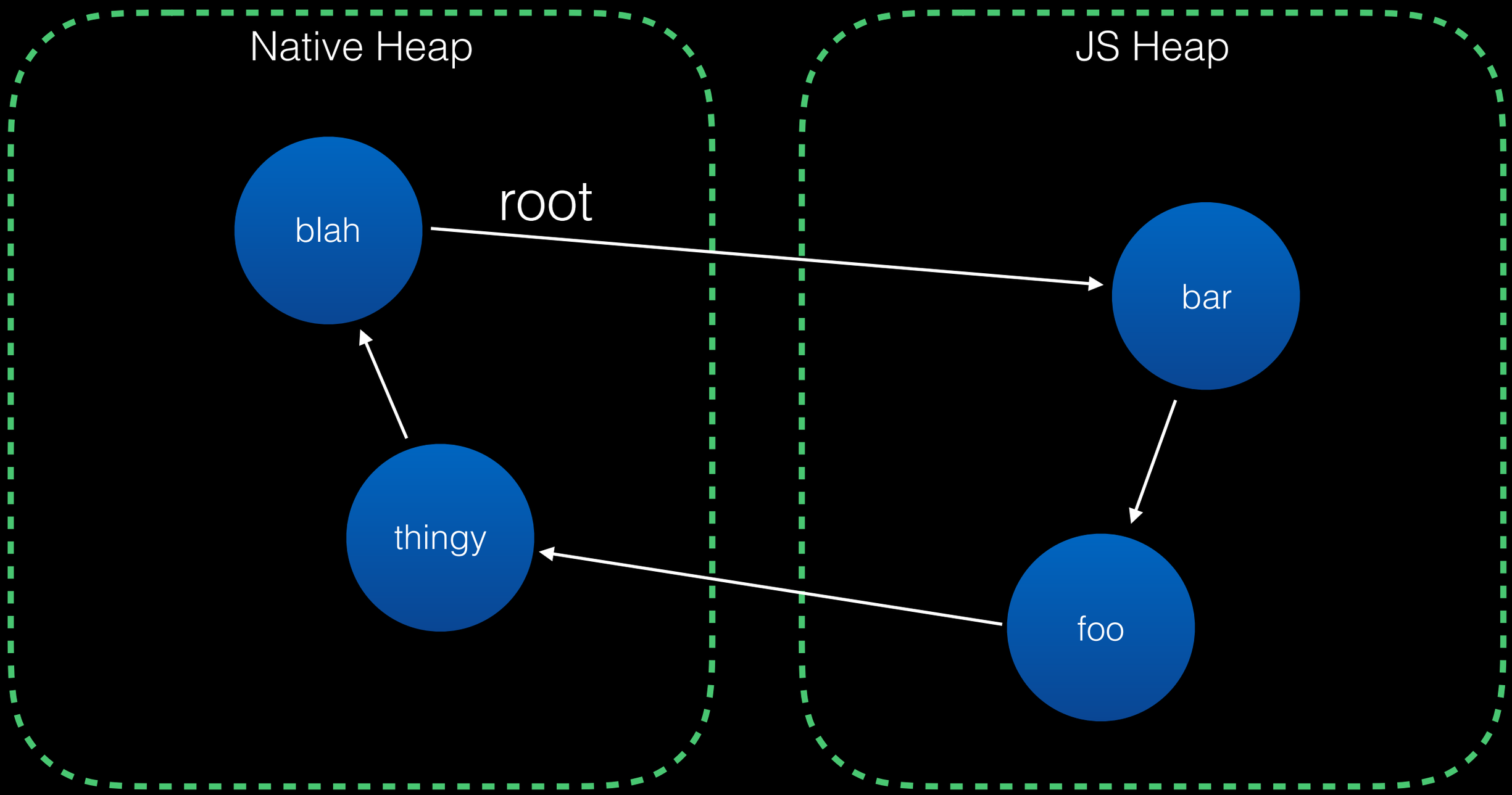
Native Code



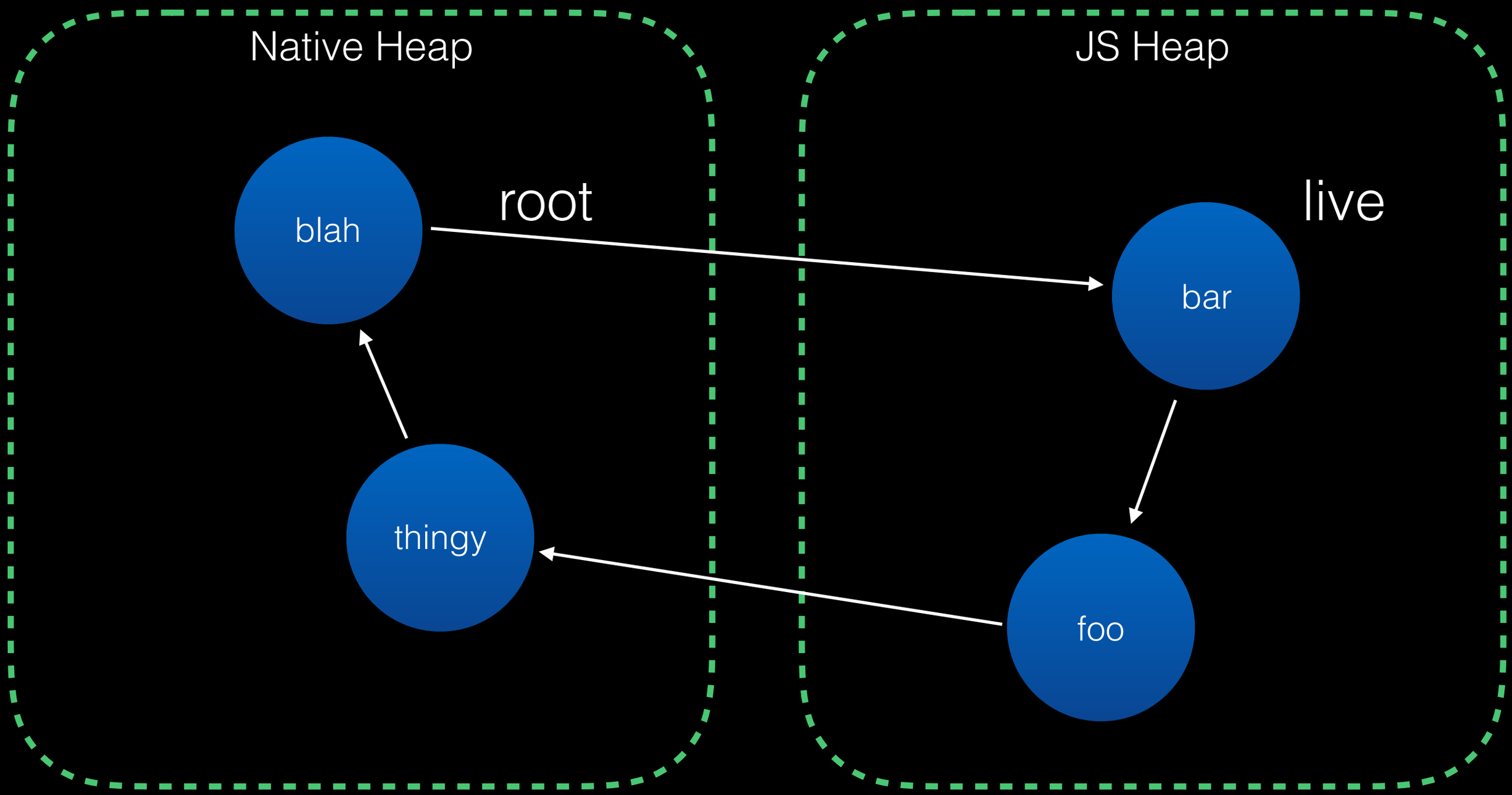
Strong Reference?



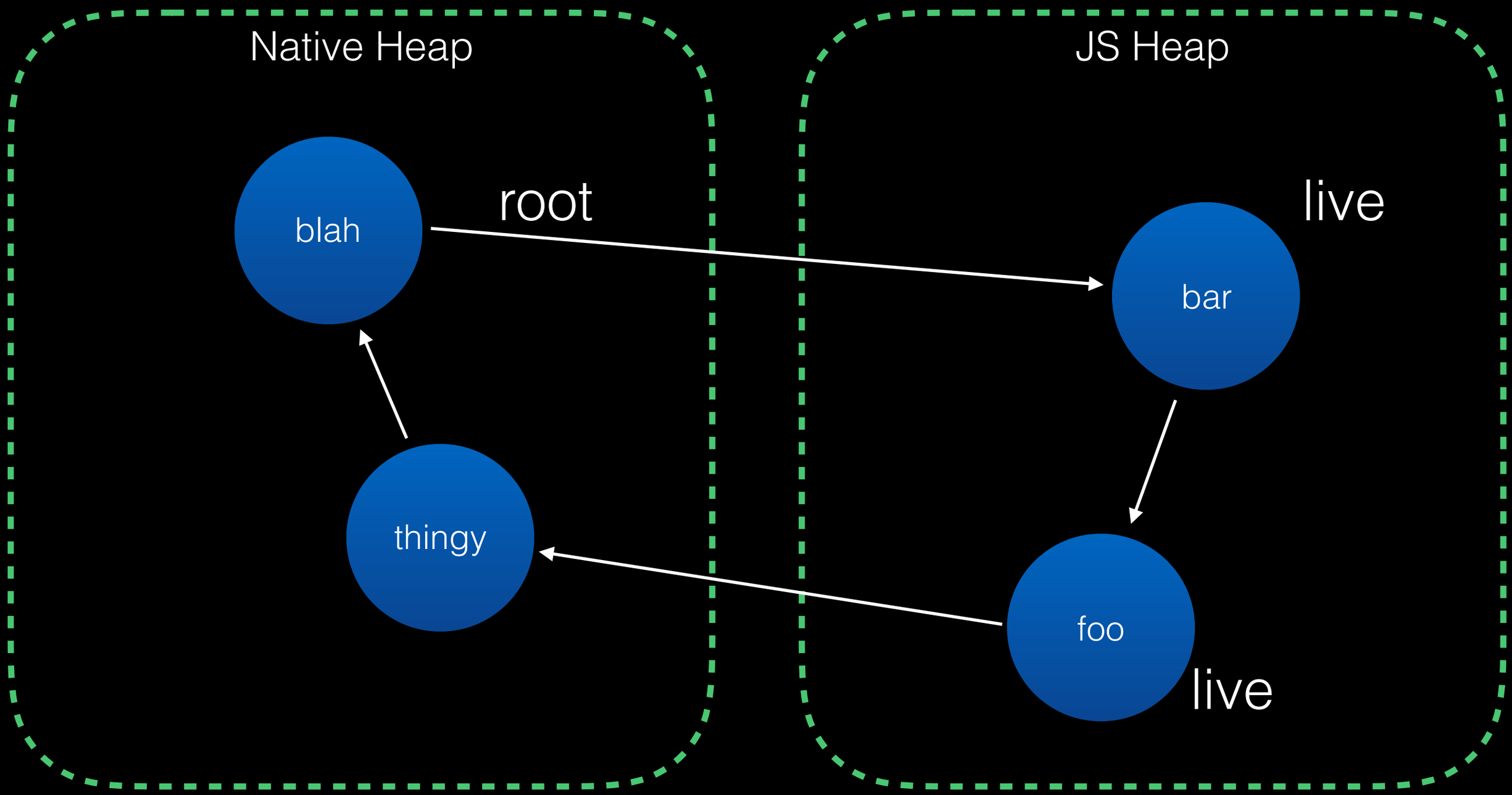
Strong Reference?



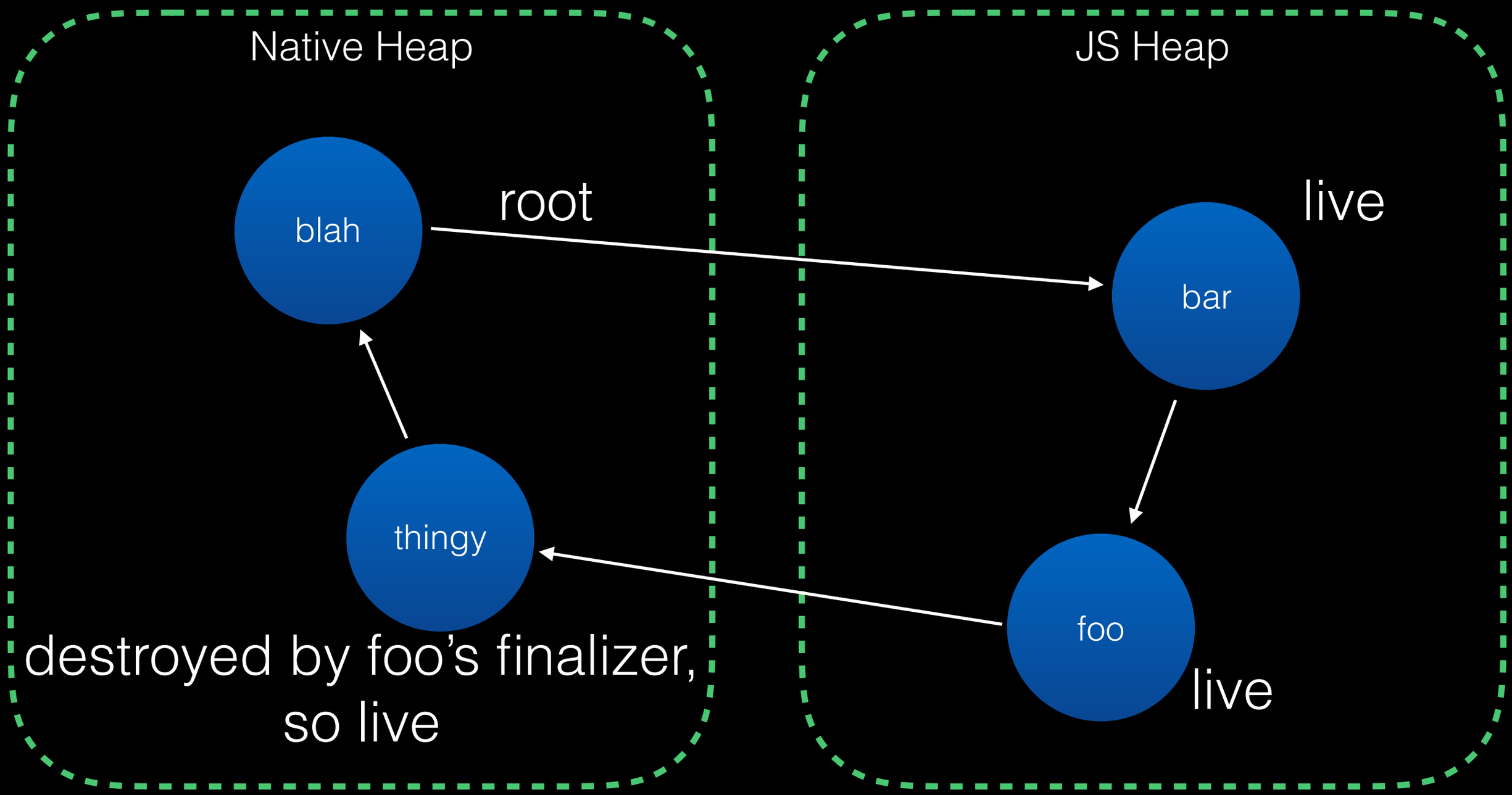
Strong Reference?



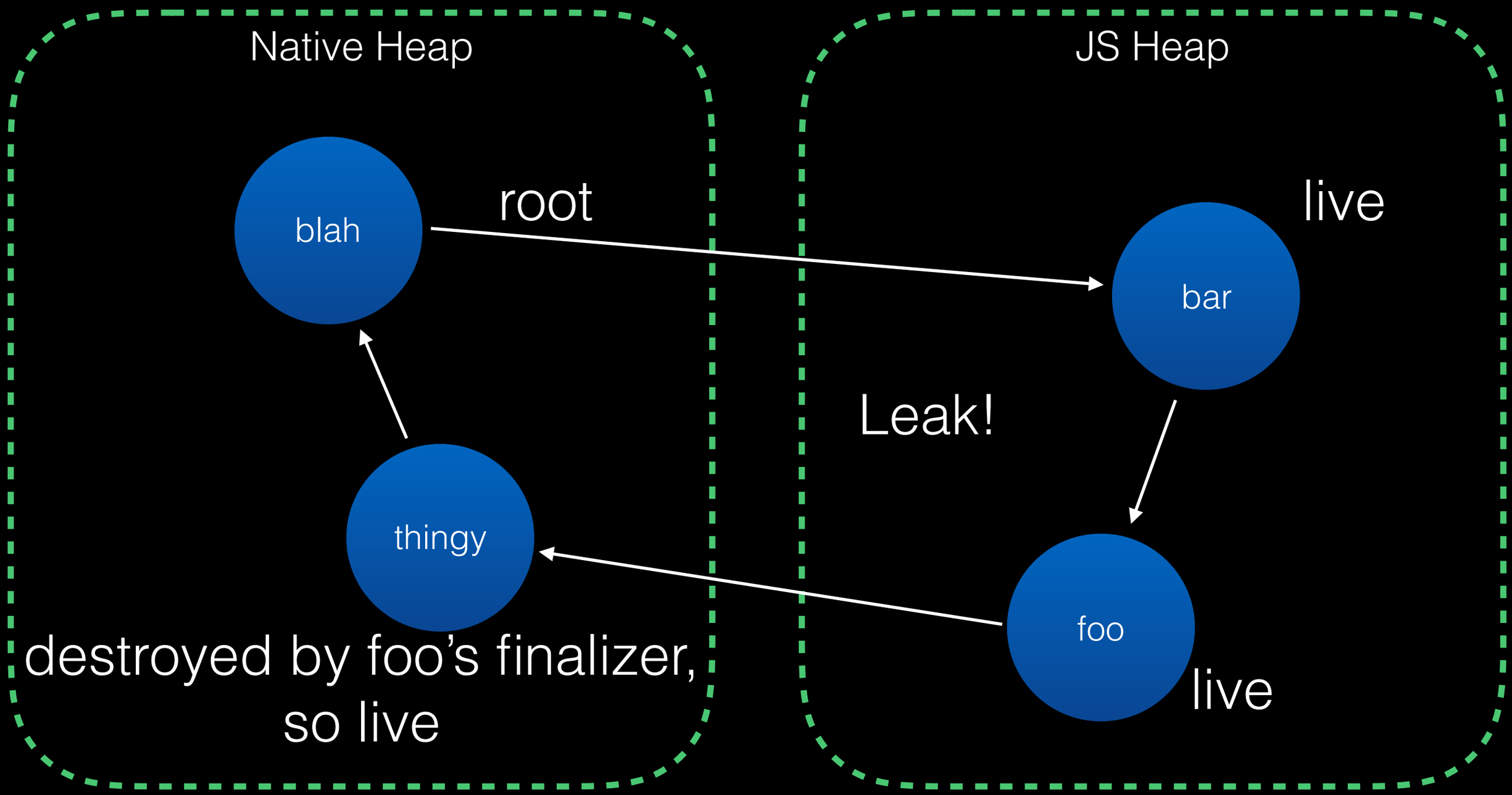
Strong Reference?



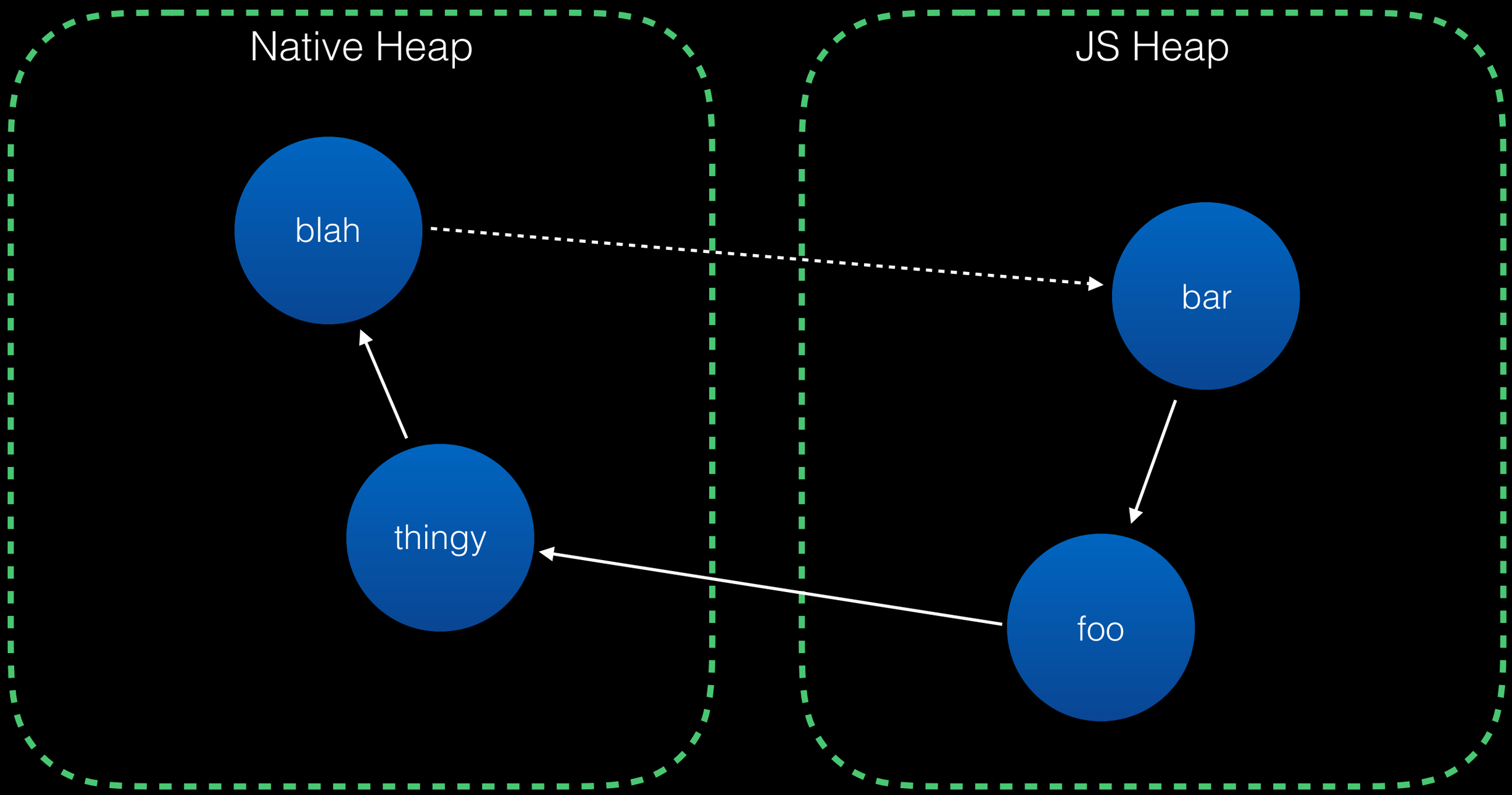
Strong Reference?



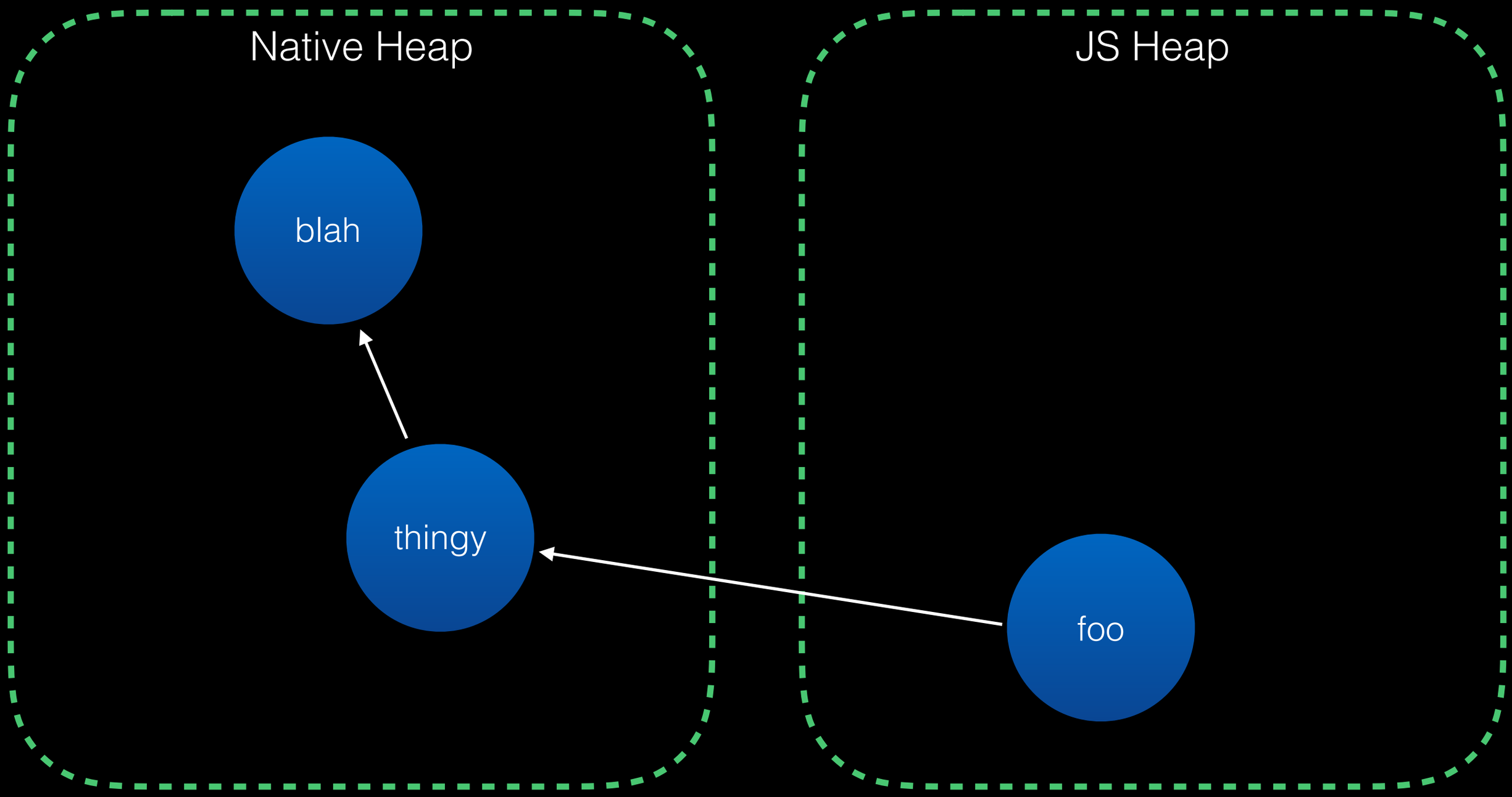
Strong Reference?



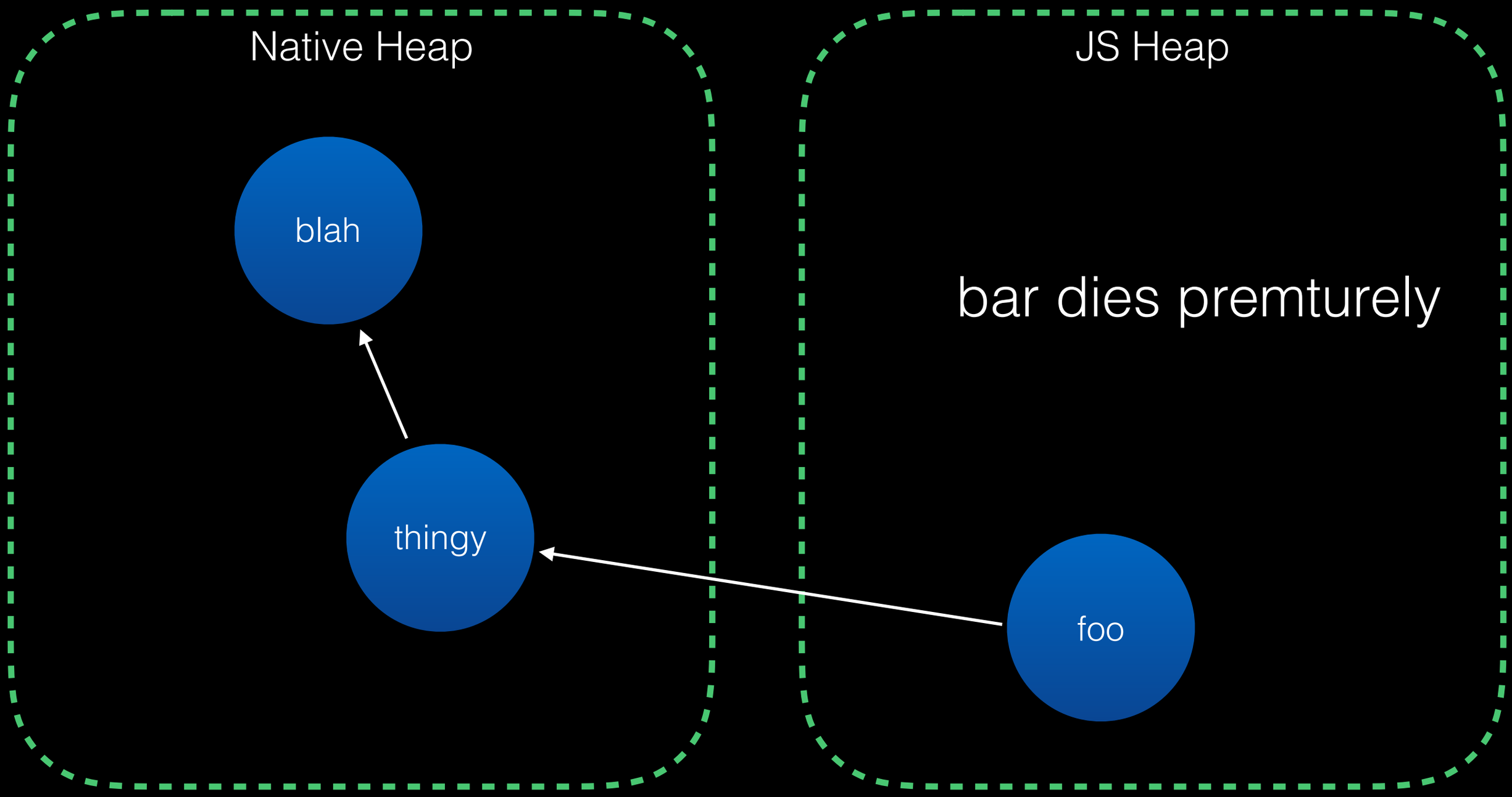
Weak Reference?



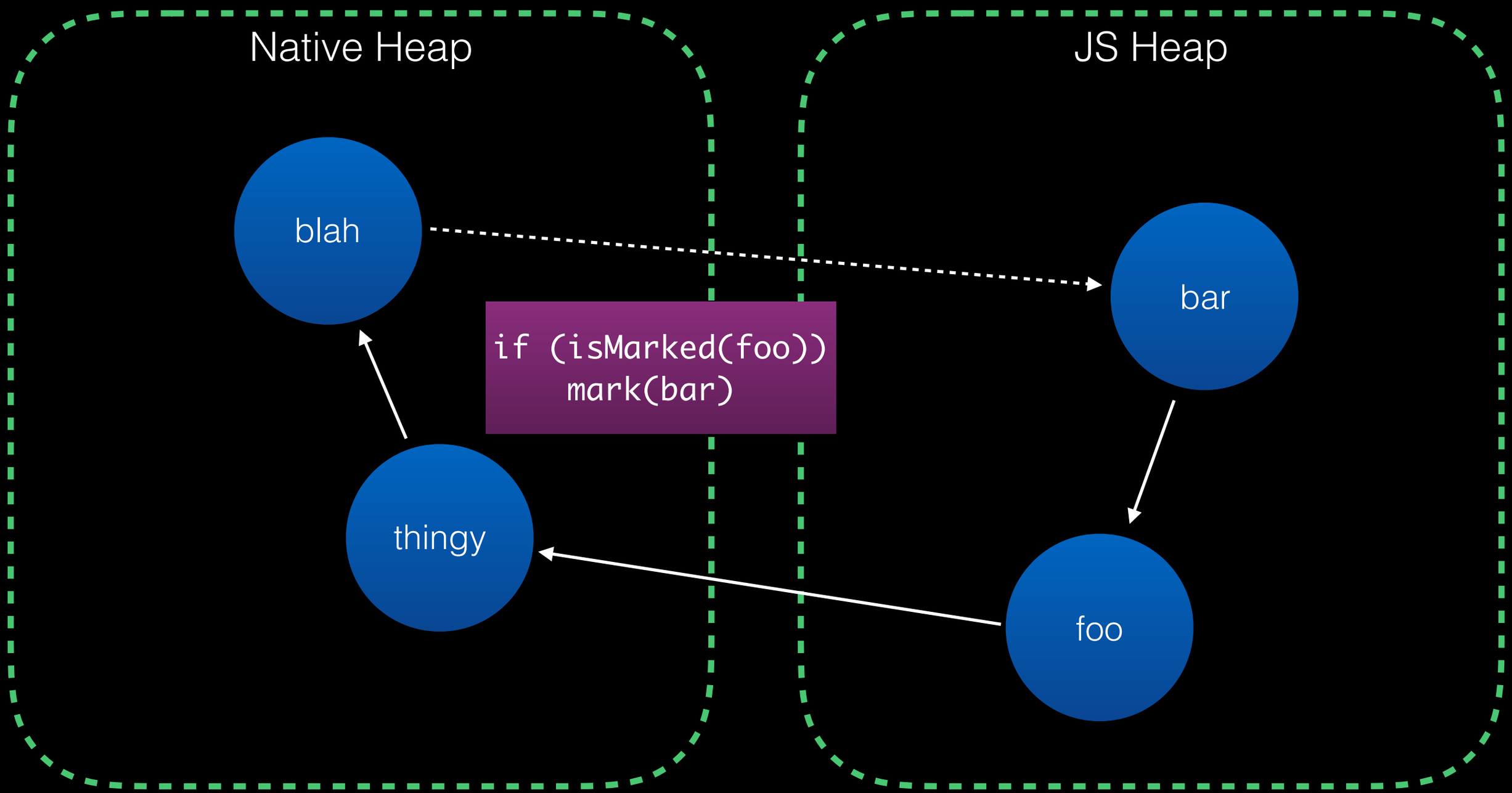
Weak Reference?



Weak Reference?



Marking Constraint!



Marking Constraints

- GC executes constraints to fixpoint.
- Useful for implementing:
  - Type Inference
  - Weak Maps
  - DOM
  - Native Code

- The most expensive marking constraint is *draining.*
- Draining = *graph search over strong references between objects*.

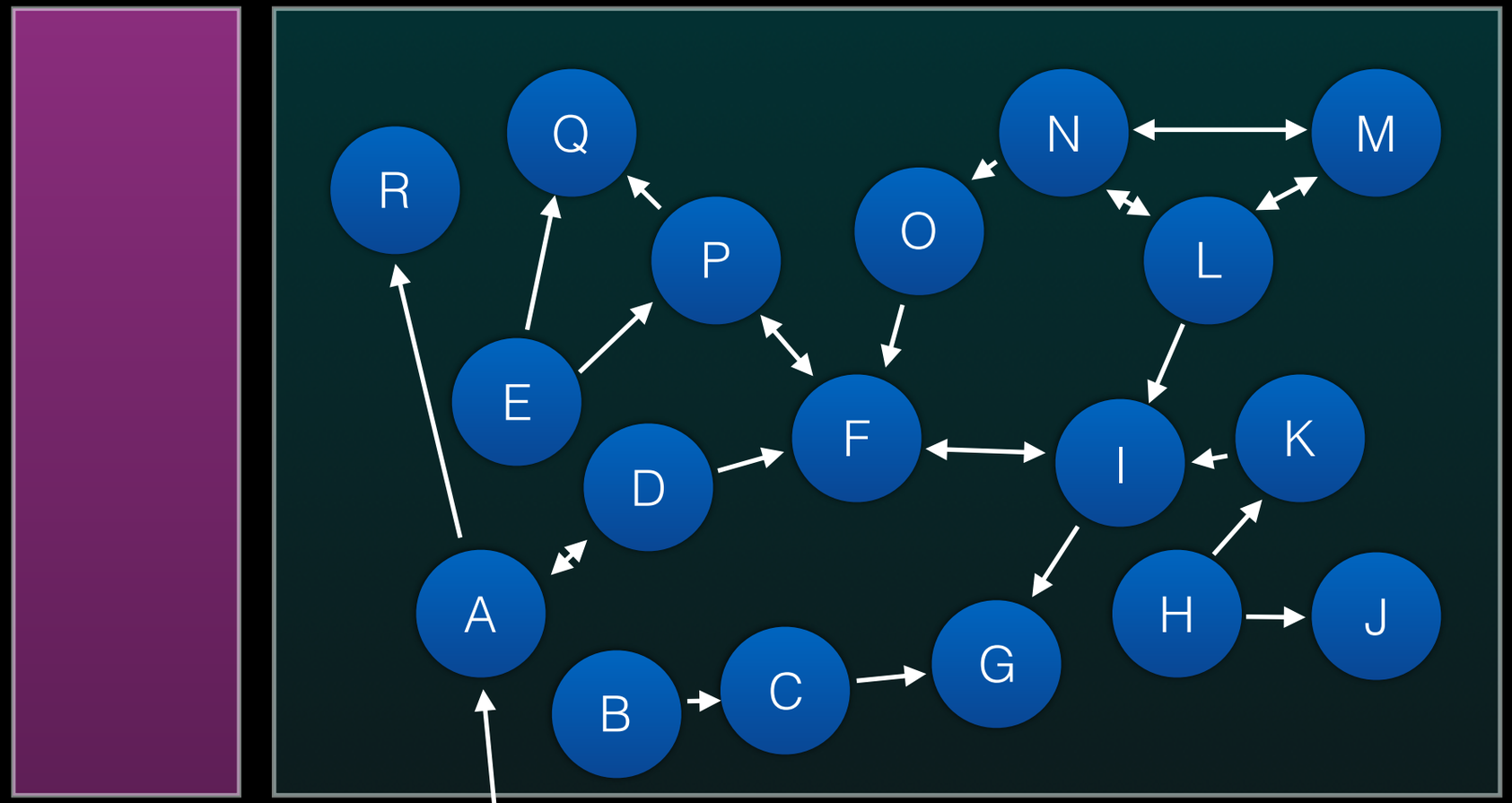
Parallel Draining

Parallel Draining

- CAS loop to set mark bits
- Handful of WTF locks
- Work donation

mark
stack

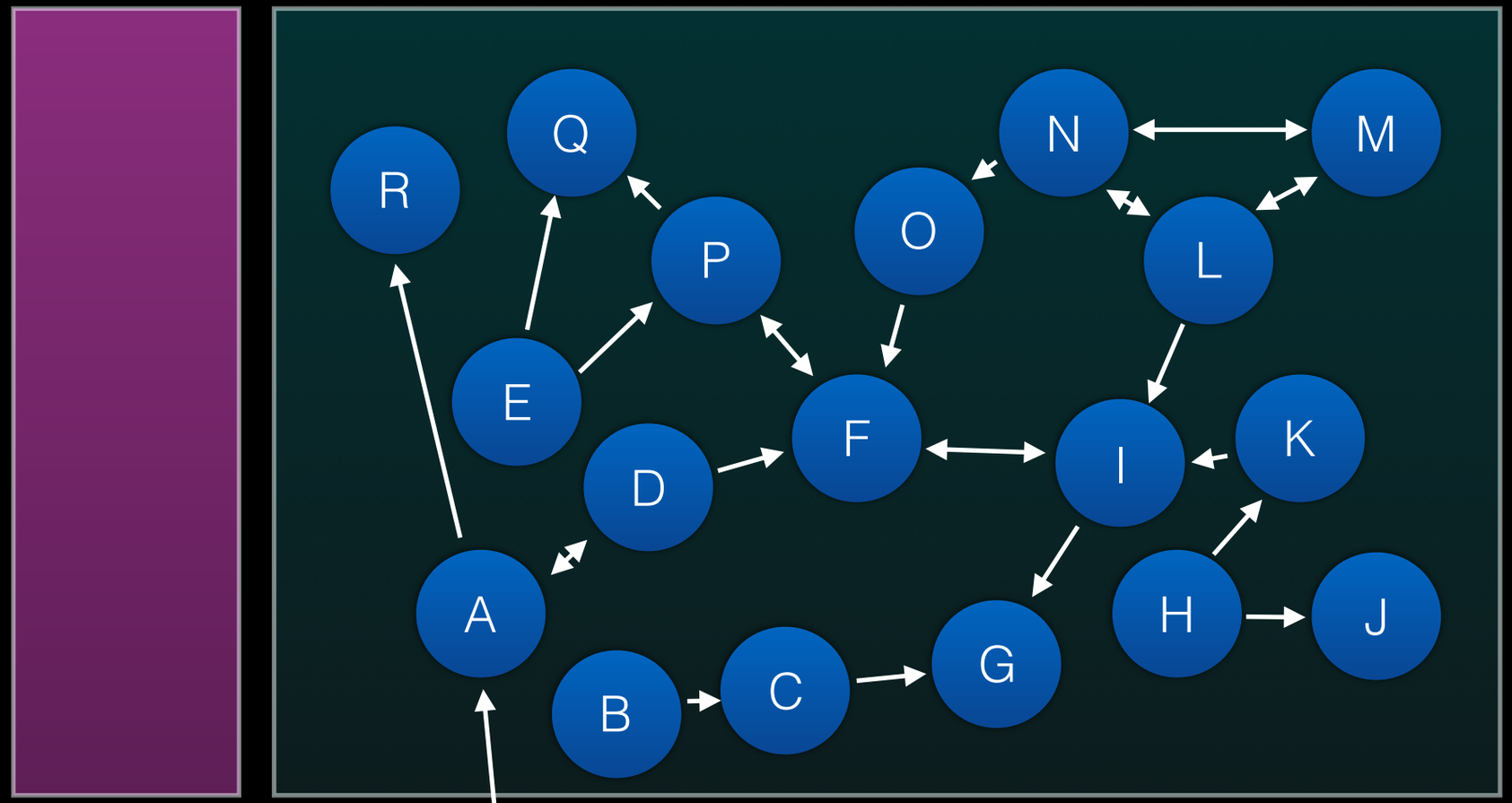
heap



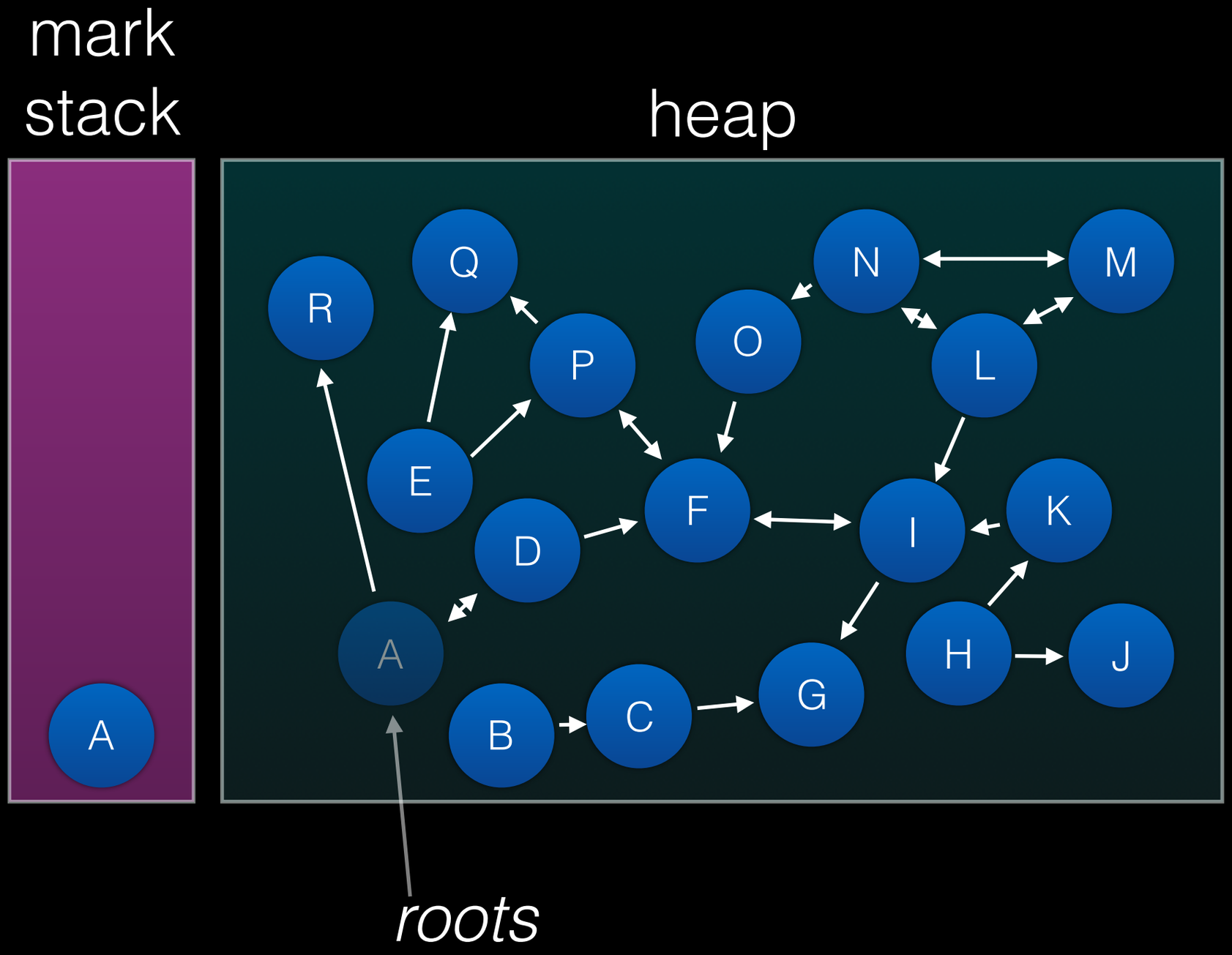
roots

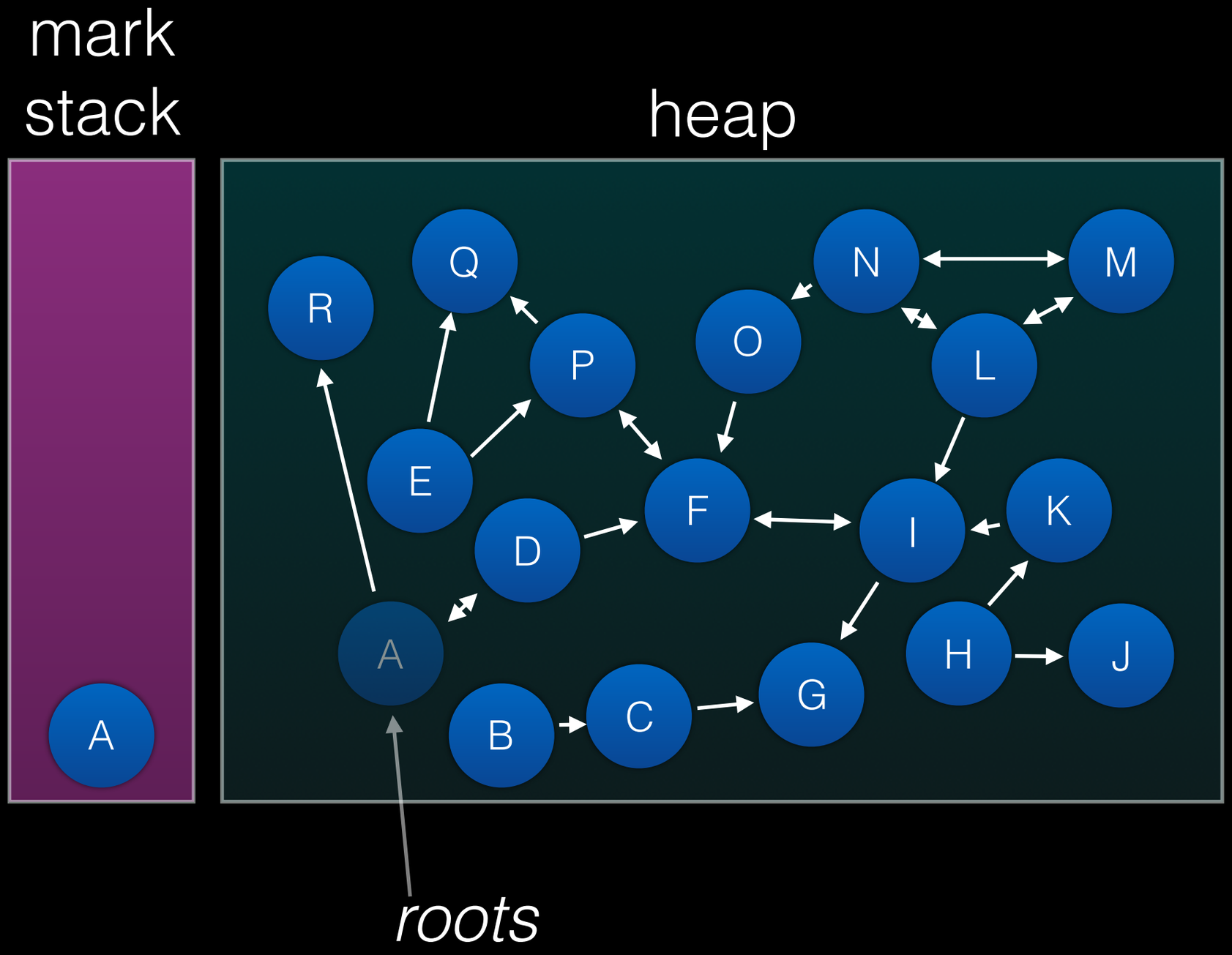
mark
stack

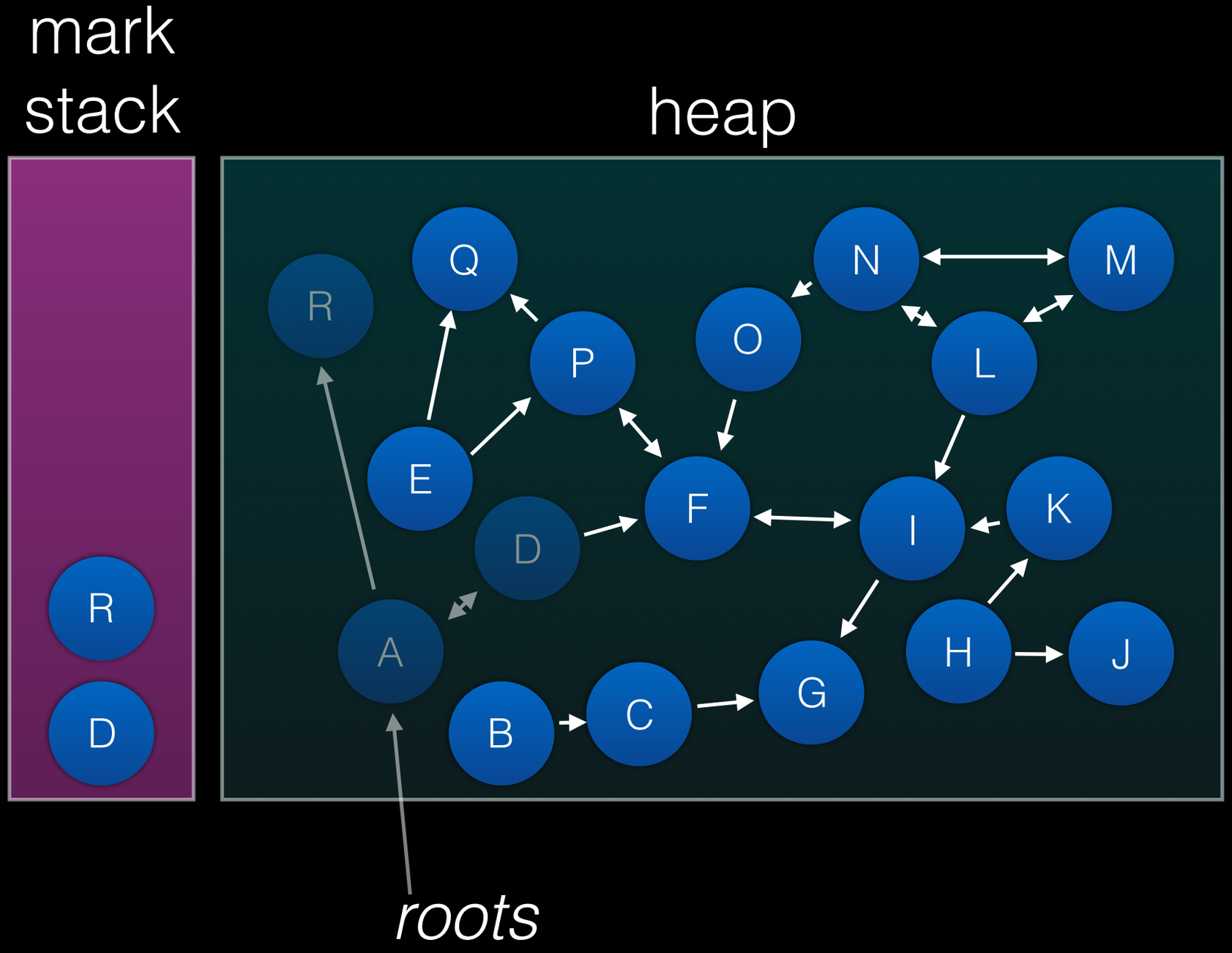
heap

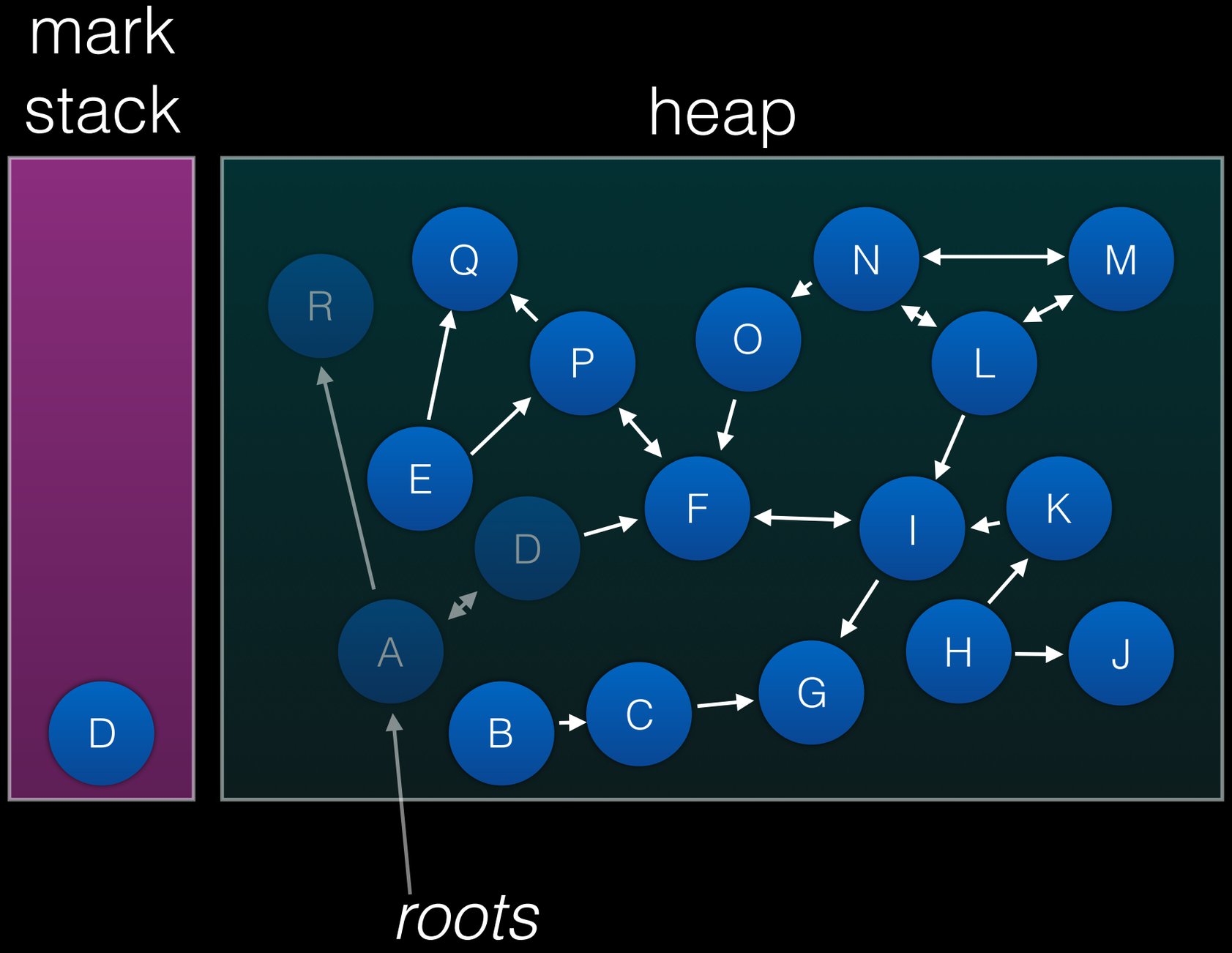


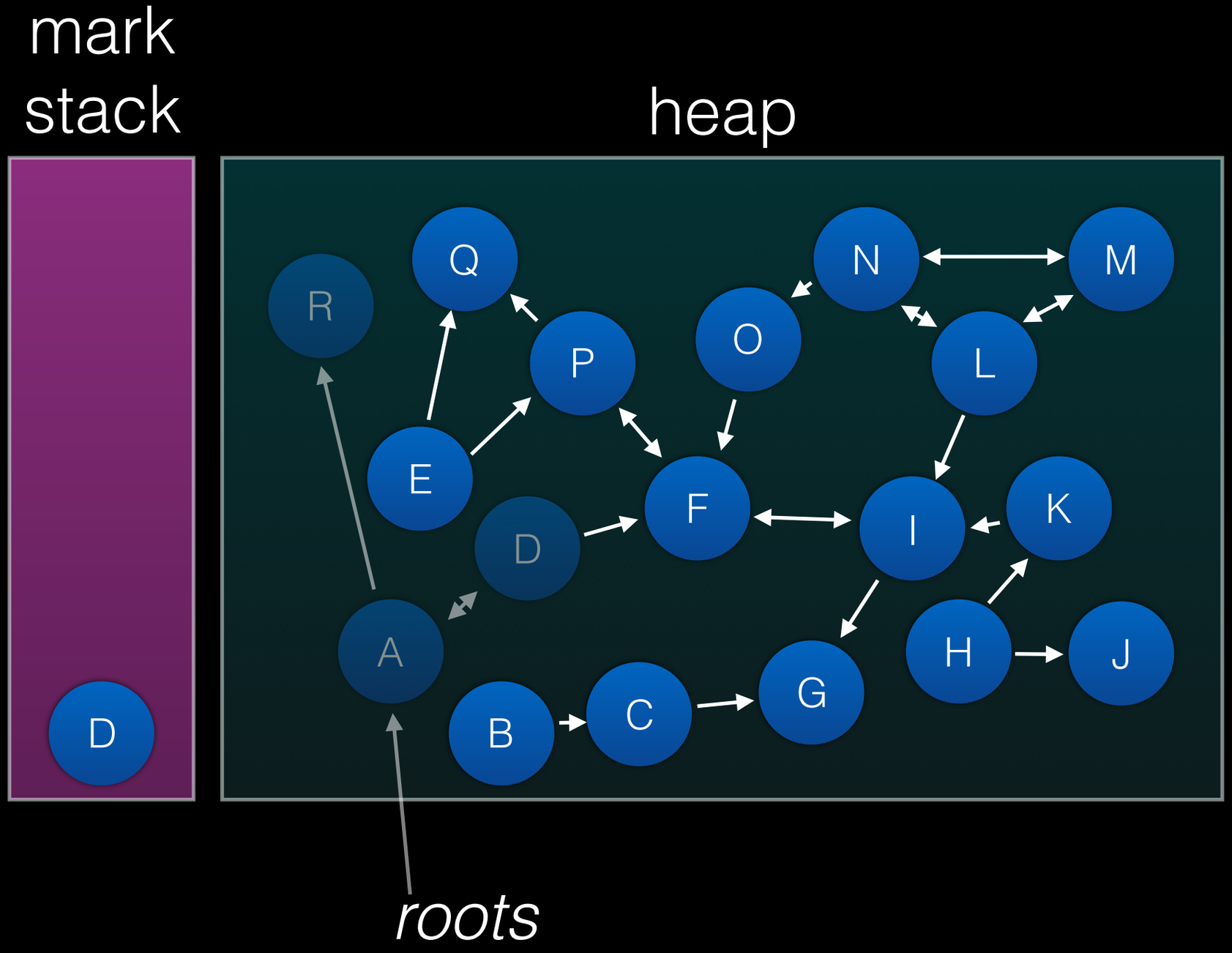
roots


mark
stack
heap
A
R
Q
P
E
D
F
N
M
O
L
I
K
A
B
C
G
H
J
roots


mark
stack
heap
A
R
Q
P
E
D
F
N
M
O
L
I
K
A
B
C
G
H
J
roots

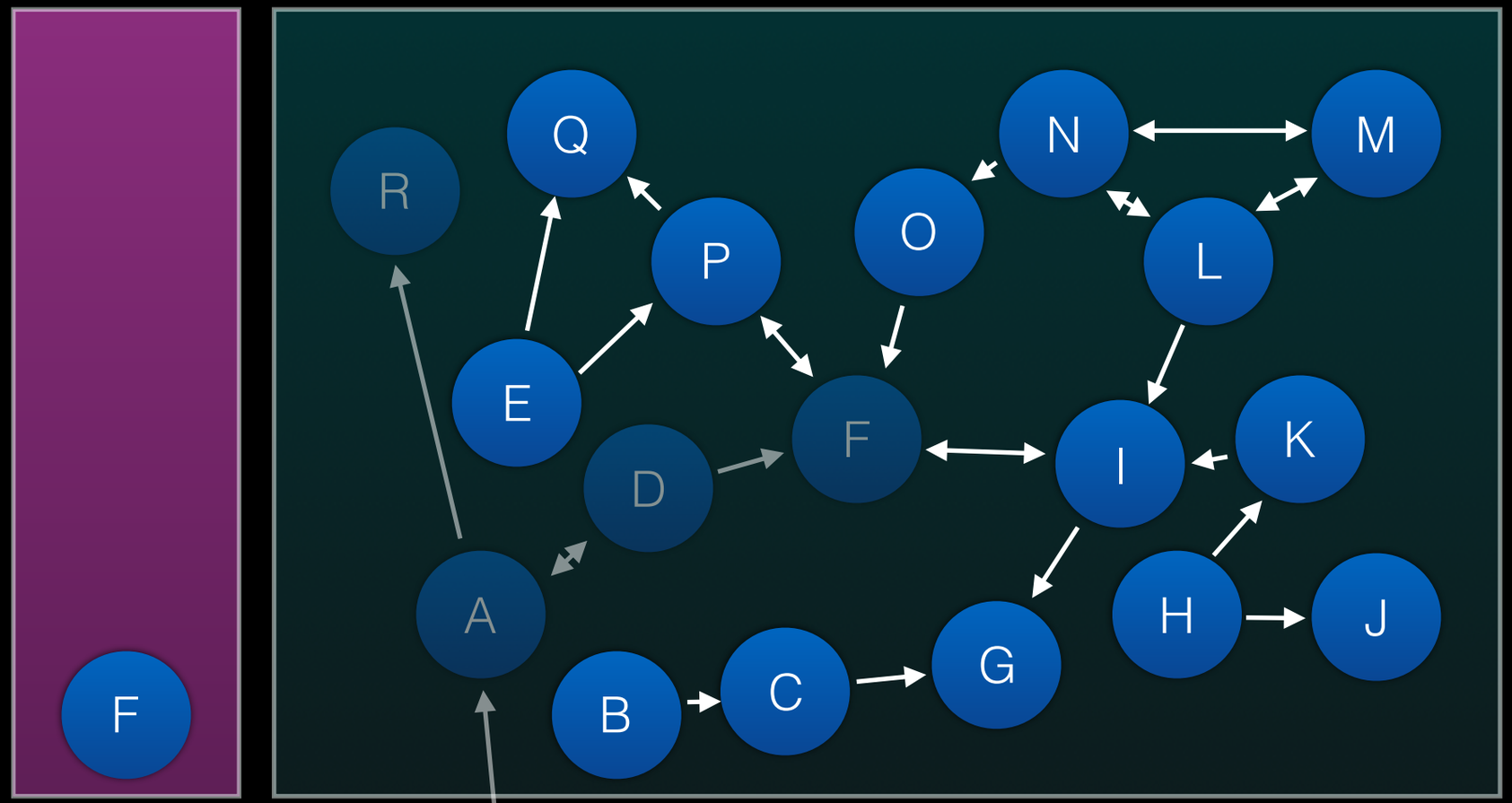

mark
stack
heap
R
D
R
Q
P
E
D
A
B
C
F
G
O
N
M
L
I
K
H
J
roots


mark
stack
heap
D
R
Q
N
M
P
O
L
E
F
I
K
D
A
H
J
B
C
G
roots

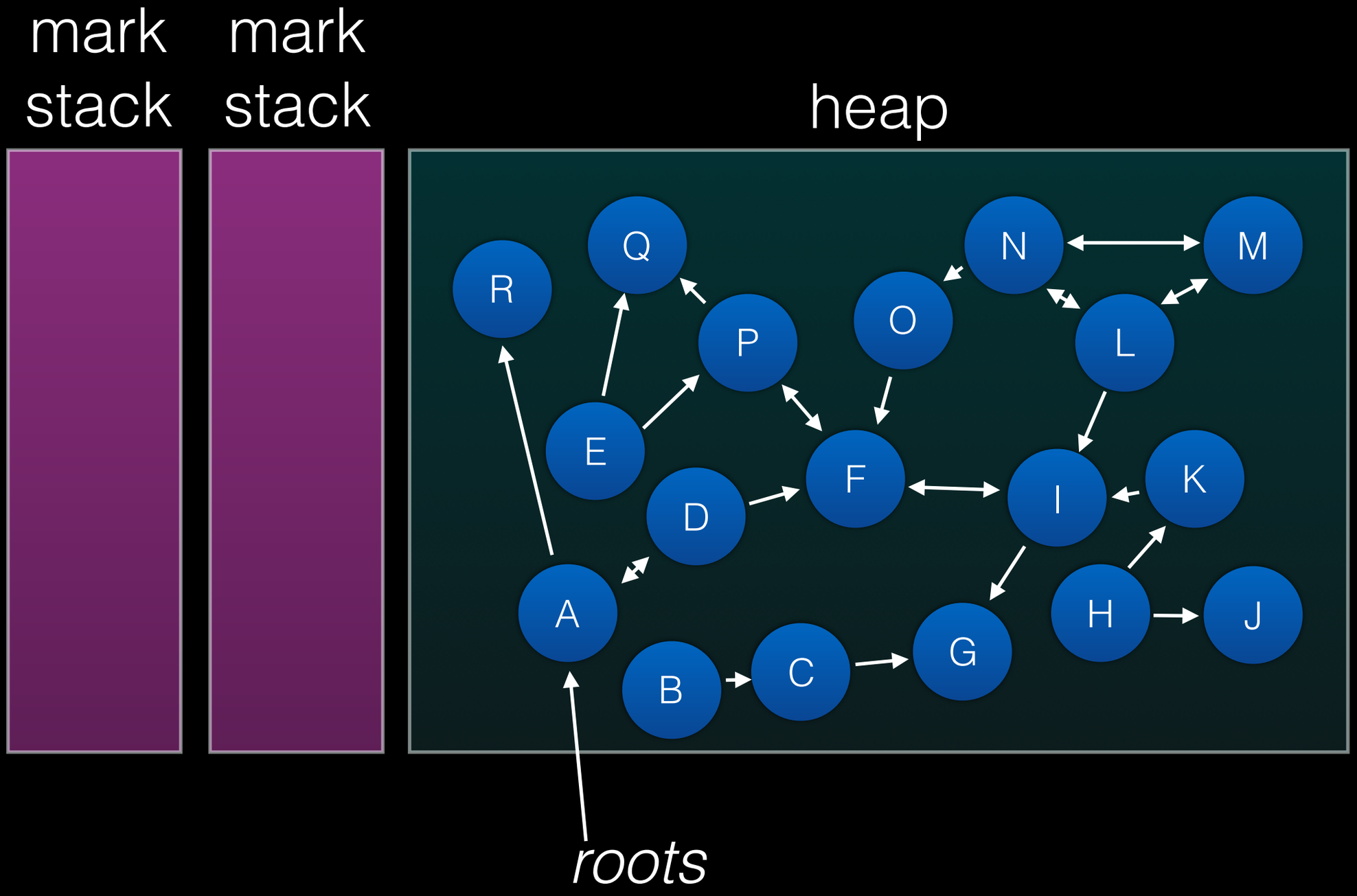

mark
stack
heap
D
R
Q
N
M
P
O
L
E
F
I
K
D
A
H
J
B
C
G
roots

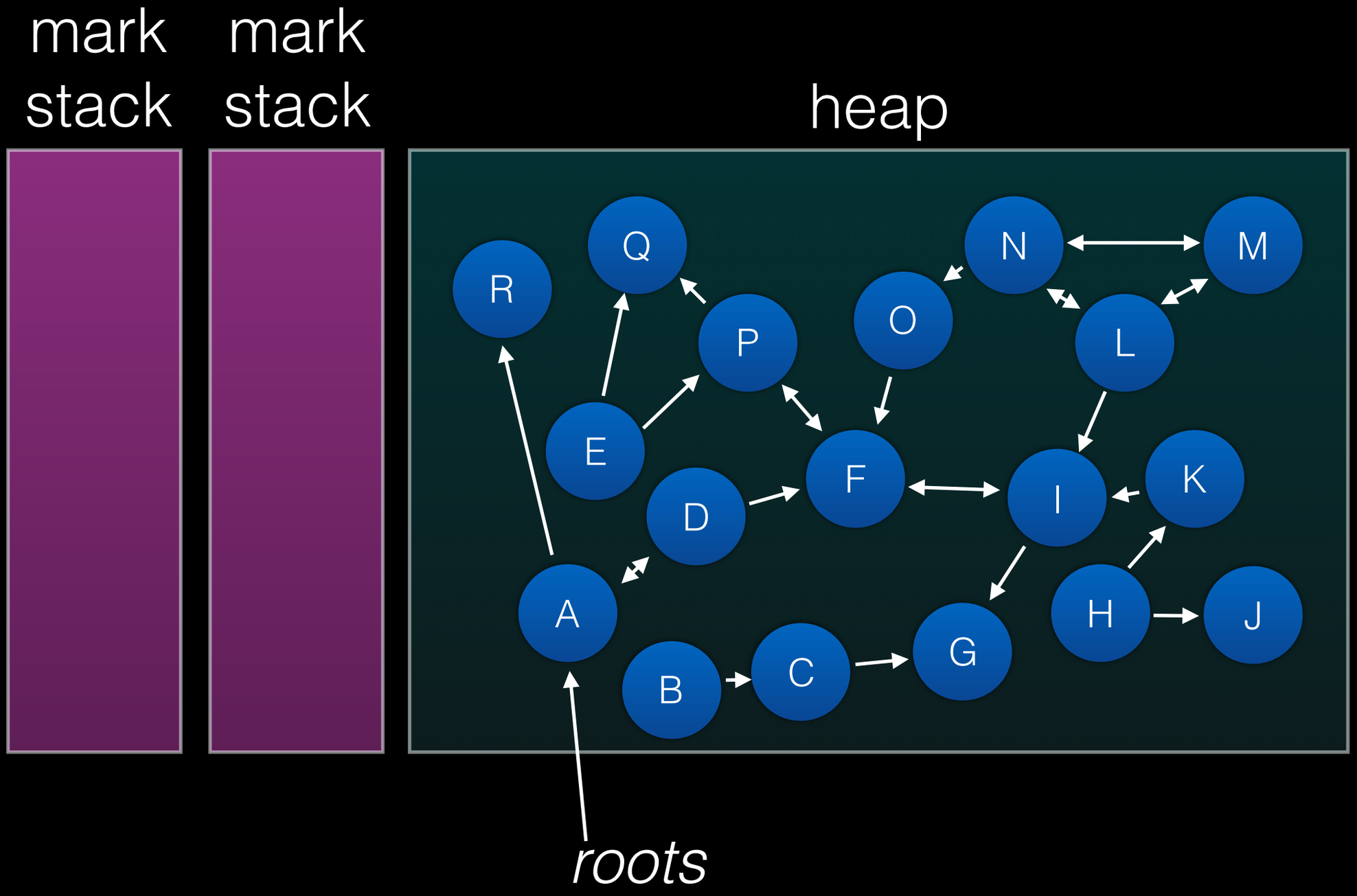
mark
stack

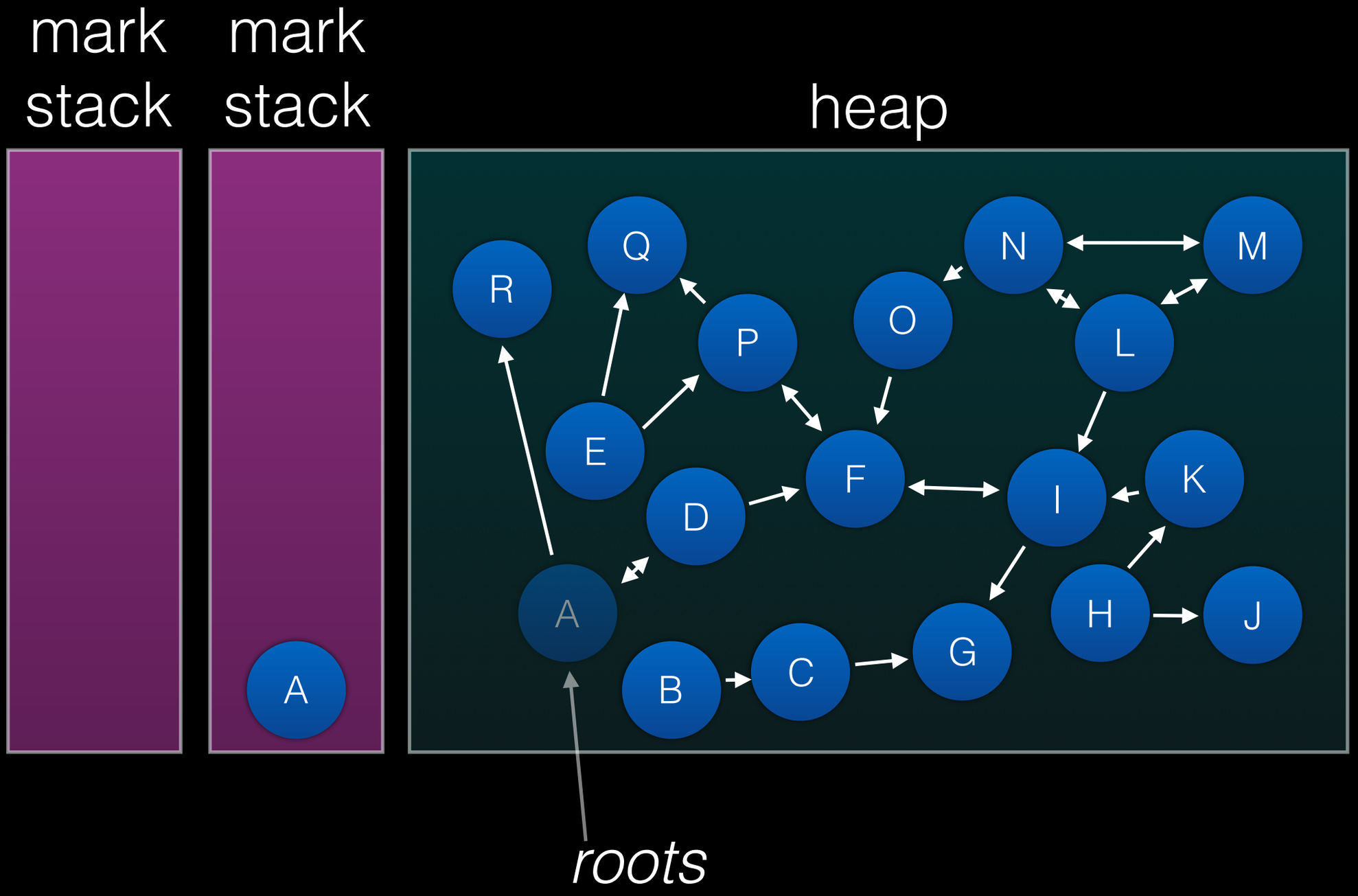
heap

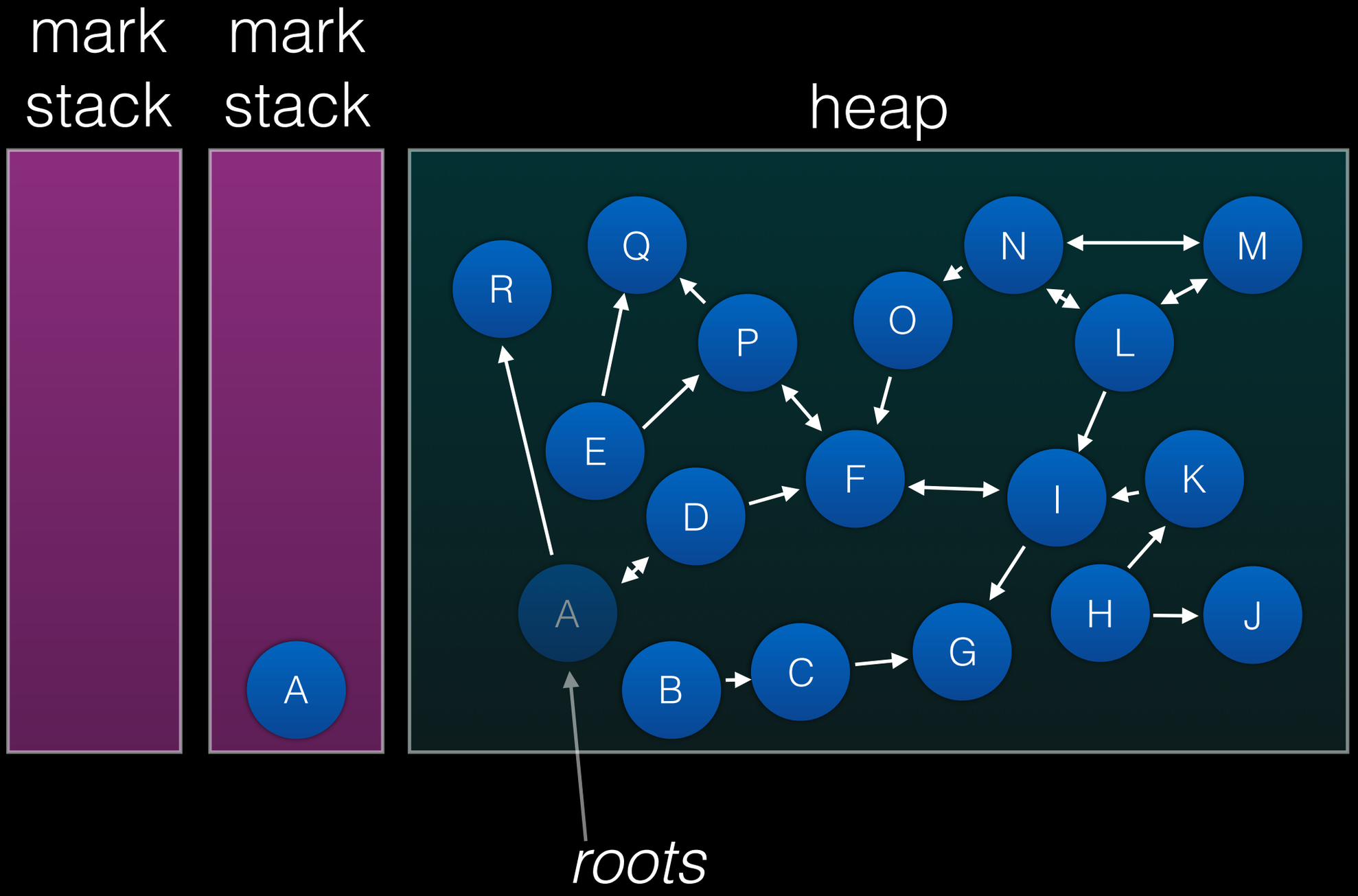


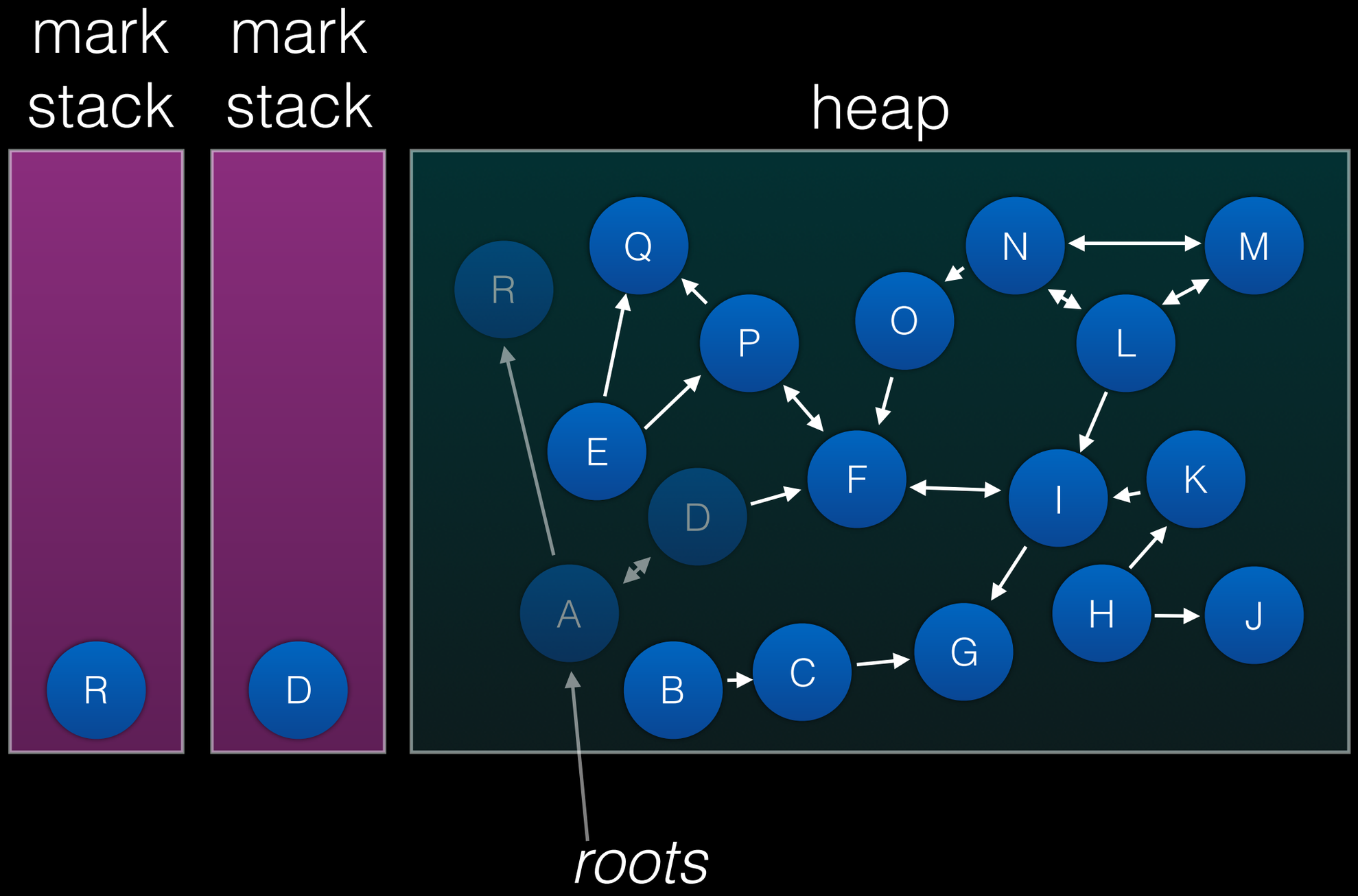
roots


mark stack
mark stack
heap
R
Q
N
M
P
O
L
E
F
I
K
D
A
H
J
B
C
G
roots


mark stack
mark stack
heap
R
Q
N
M
P
O
L
E
F
I
K
D
A
H
J
B
C
G
roots


mark stack
mark stack
heap
A
R
Q
P
E
D
F
N
M
O
L
I
K
H
J
G
C
B
A
roots


mark stack
mark stack
heap
A
R
Q
P
E
D
F
N
M
O
L
I
K
H
J
G
C
B
A
roots


mark stack
mark stack
heap
R
Q
N
M
O
P
L
E
F
I
K
D
A
H
J
B
C
G
R
D
roots

Parallel Draining

- Donate work to global mark stack when we detect *fan-out*.
- Stalled threads take 1/N of the global mark stack.

Detecting Fan-out

- Once every 100 objects visited, check if it's worth donating.
- Only donate if the global mark stack is empty and nobody holds the lock.
- Donate about half of local mark stack.

Parallel Draining

Sticky Mark Bits

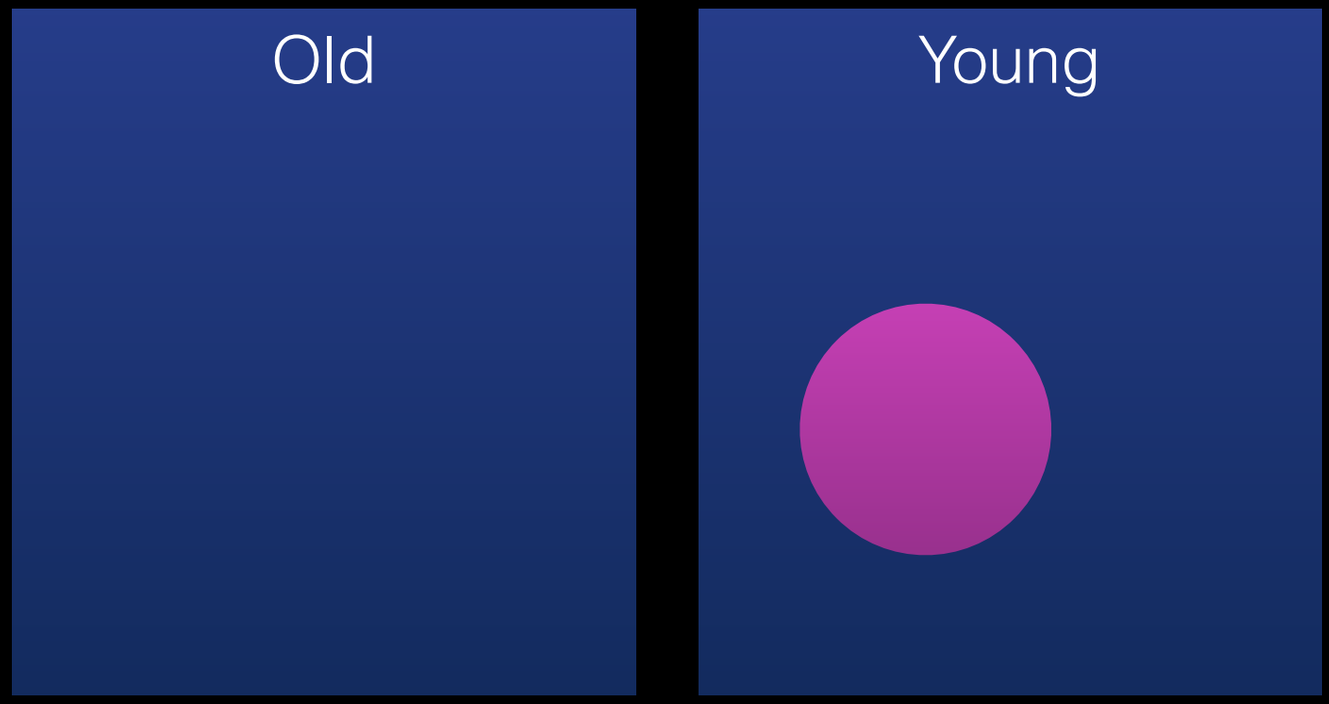
- In most cases, young objects are much more likely to die than old objects.

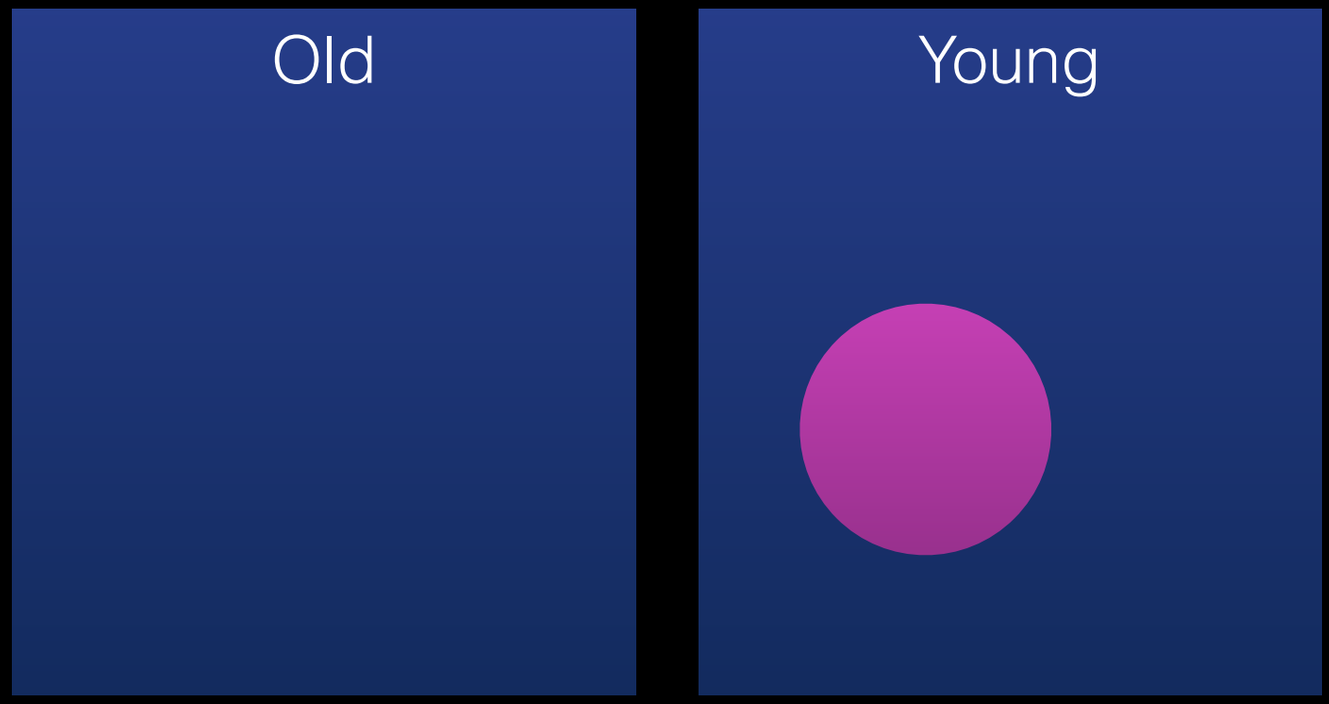
Generational GC

- Sticky Mark Bits = Generational GC without copying

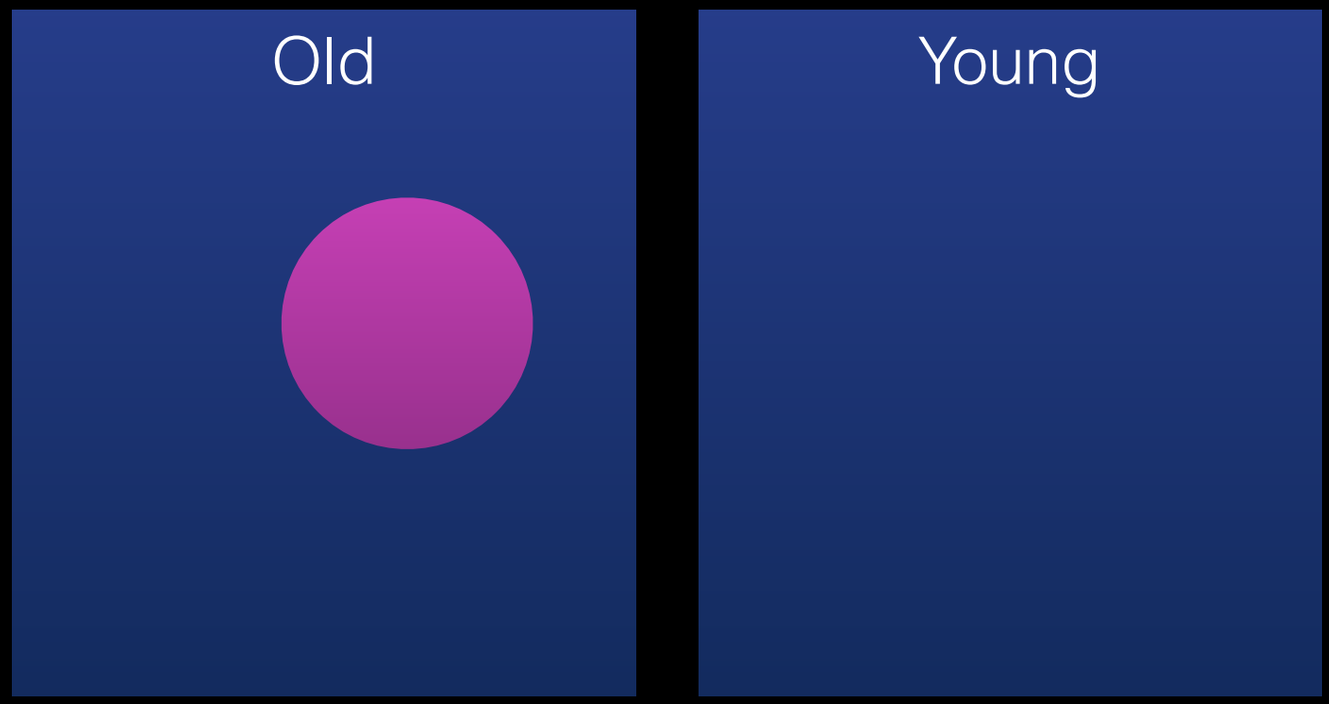
Old

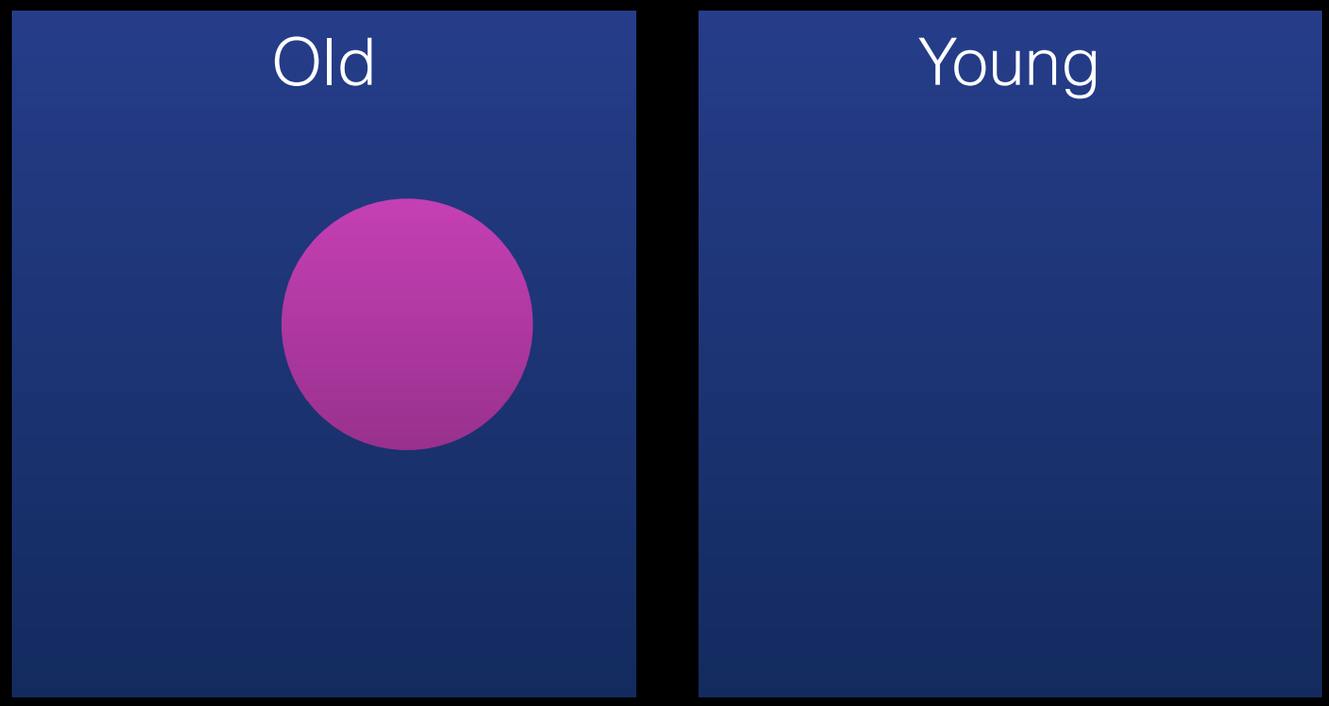
Young


Old
Young



We know that the object is young because of its address.


Old
Young



We know that the object is old because of its address.

- Traditional GenGC uses address to encode generation.
- We can just as easily use some bits in the object header.

Sticky Mark Bits

- Each object has a GC byte that tells us the generational state:
  - New
  - Remembered
  - Old
- Mark bits are not reset at the start of *eden* collections.

```
o->field = p
if (o->cellState() == Old)
    remember(o); // state becomes Remembered
```

Sticky Mark Bits

- Generational GC without copying.
- Zero-cost write barrier.
- Essential for throughput.

Conservative Roots

Style	Implications
Accutate	Native code uses handles Compiler isolates pointers Theoretically less object drag Pain in the butt
Conservative	Native code uses raw pointers Compiler treats pointers as integers Possibly more object drag *Mark pinning* Super easy

Style	Implications
Accutate	Native code uses handles Compiler isolates pointers Theoretically less object drag Pain in the butt
Conservative	Native code uses raw pointers Compiler treats pointers as integers Possibly more object drag *Mark pinning* Super easy

Is this really a thing?

- Conservative object drag totally was a thing.

- We mitigated conservative object drag:
  - 48-bit address space
  - ASLR
  - Stack sanitization

- See here for more sweet mitigations:
  http://www.hboehm.info/gc/

Fragmentation

Mitigation

- Most of WebKit's heap is malloc memory allocated by the DOM and HTML/CSS/SVG rendering code.

- malloc mitigates fragmentations by modeling first-fit.

- Not moving objects is the norm on desktop systems.

Fragmentation Mitigations

- Simulate first-fit
- Coalesce and split
- 48-bit address space

Efficient Mark-Sweep

- Simple Segregated Storage
- Bump’n’pop
- Constraint-Based Marking
- Parallel “Draining”
- Sticky Mark Bits
- Conservative Roots
- Fragmentation Mitigations

Agenda

- ~~Introduction~~
- ~~JavaScriptCore~~
- ~~Efficient Mark-Sweep~~

  - *30 minute break*

- Concurrent GC
- bmalloc
- WTF::Lock