Garbage Collection

Photo source: http://www.flickr.com/photos/joyoflife/101479550/ CC-By-NC-SA

Garbage Collection

Garbage Collection

Two ways to manage the heap:

Garbage Collection

Two ways to manage the heap:

- Manual (malloc, free, mmap, etc...)

Garbage Collection

Two ways to manage the heap:

- Manual (malloc, free, mmap, etc...)
  - Pros: Fast, easy for compiler, obvious semantics

Garbage Collection

Two ways to manage the heap:

- Manual (malloc, free, mmap, etc...)
  - Pros: Fast, easy for compiler, obvious semantics
  - Cons: Difficult to use, impossible* to make safe

Garbage Collection

Two ways to manage the heap:

- Manual (malloc, free, mmap, etc...)
  - Pros: Fast, easy for compiler, obvious semantics
  - Cons: Difficult to use, impossible* to make safe
- Automatic (e.g. garbage collection)

Garbage Collection

Two ways to manage the heap:

- Manual (malloc, free, mmap, etc...)
  - Pros: Fast, easy for compiler, obvious semantics
  - Cons: Difficult to use, impossible* to make safe
- Automatic (e.g. garbage collection)
  - Pros: Safe, easy for programmer

Garbage Collection

Two ways to manage the heap:

- Manual (malloc, free, mmap, etc...)
  - Pros: Fast, easy for compiler, obvious semantics
  - Cons: Difficult to use, impossible* to make safe
- Automatic (e.g. garbage collection)
  - Pros: Safe, easy for programmer
  - Cons: Unpredictability, can be slow

GC Ideas

- Want to free memory we no longer need
- Estimate this by reachability:

```
public void foo() {
     Foo x = new Foo();

     x.bar();

     x = null;

}
```

GC Ideas

- Want to free memory we no longer need
- Estimate this by reachability:

```
public void foo() {
    Foo x = new Foo();
    x.bar();

    x = null;

}
```

x still in use, cannot free

GC Ideas

- Want to free memory we no longer need
- Estimate this by reachability:

```
public void foo() {
     Foo x = new Foo();

     x.bar();

     x = null;

}
```

x still in use, cannot free

x unused, but reachable

GC Ideas

- Want to free memory we no longer need
- Estimate this by reachability:

```
public void foo() {
    Foo x = new Foo();
    x.bar();
    x = null;
}
```

x still in use, cannot free

x unused, but reachable

x unreachable, can be freed

Reachability

Reachability starts from the “roots”:

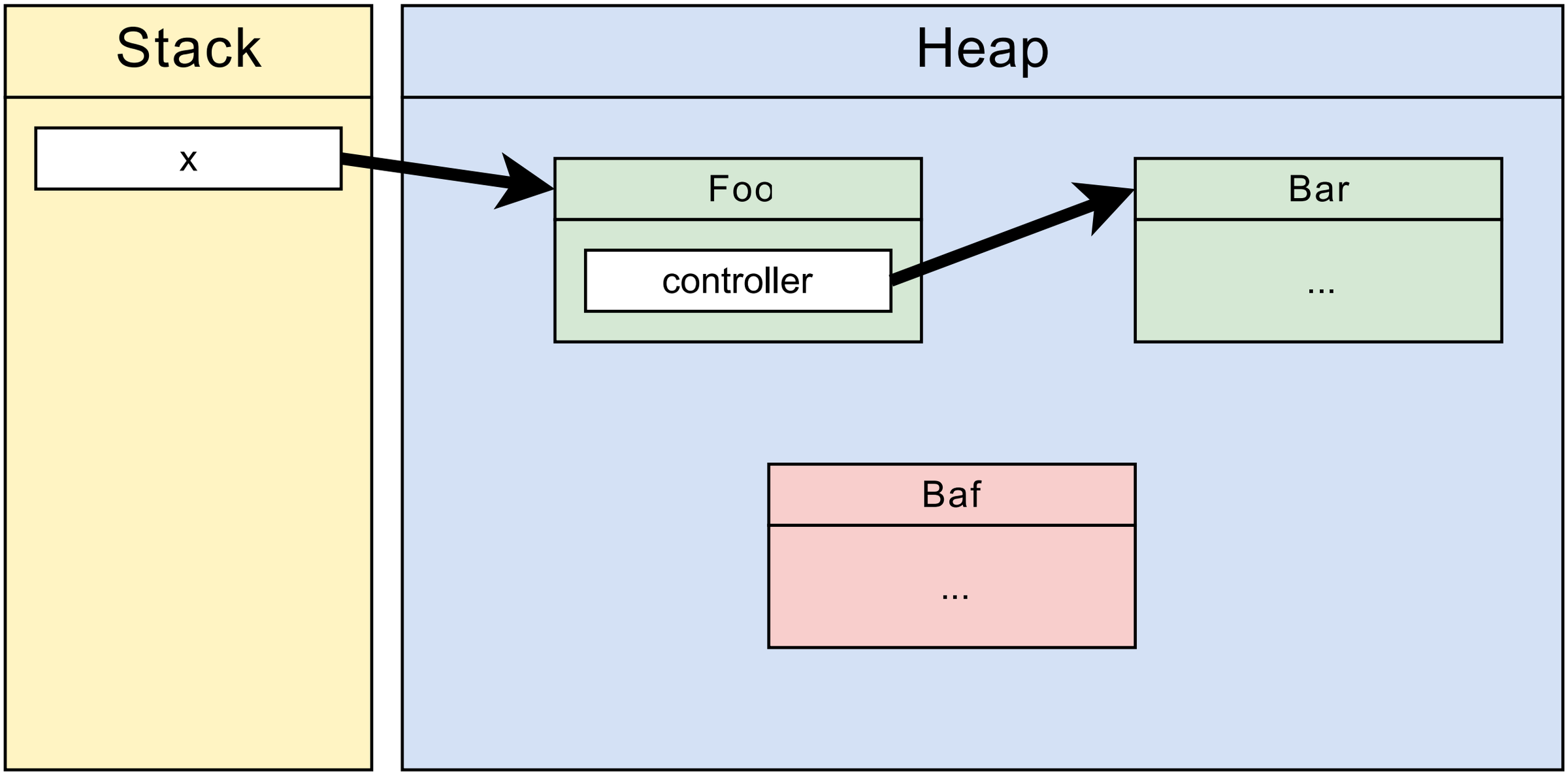
- Stack and global variables

Roots themselves are managed separately:

- Stack in activation frames
- Global variables persistent

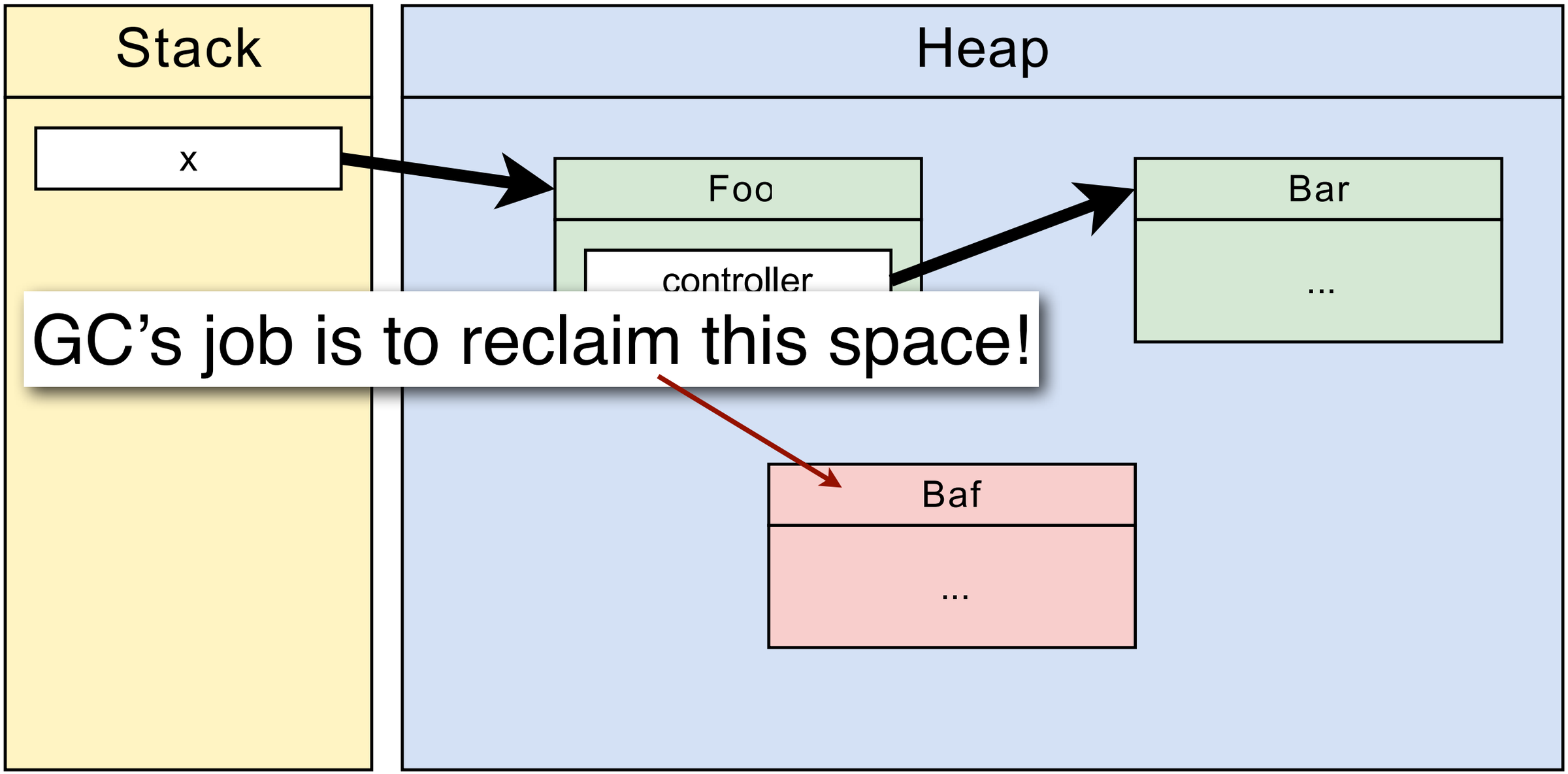
Reachability

Reachability is transitive:



Reachability

Reachability is transitive:

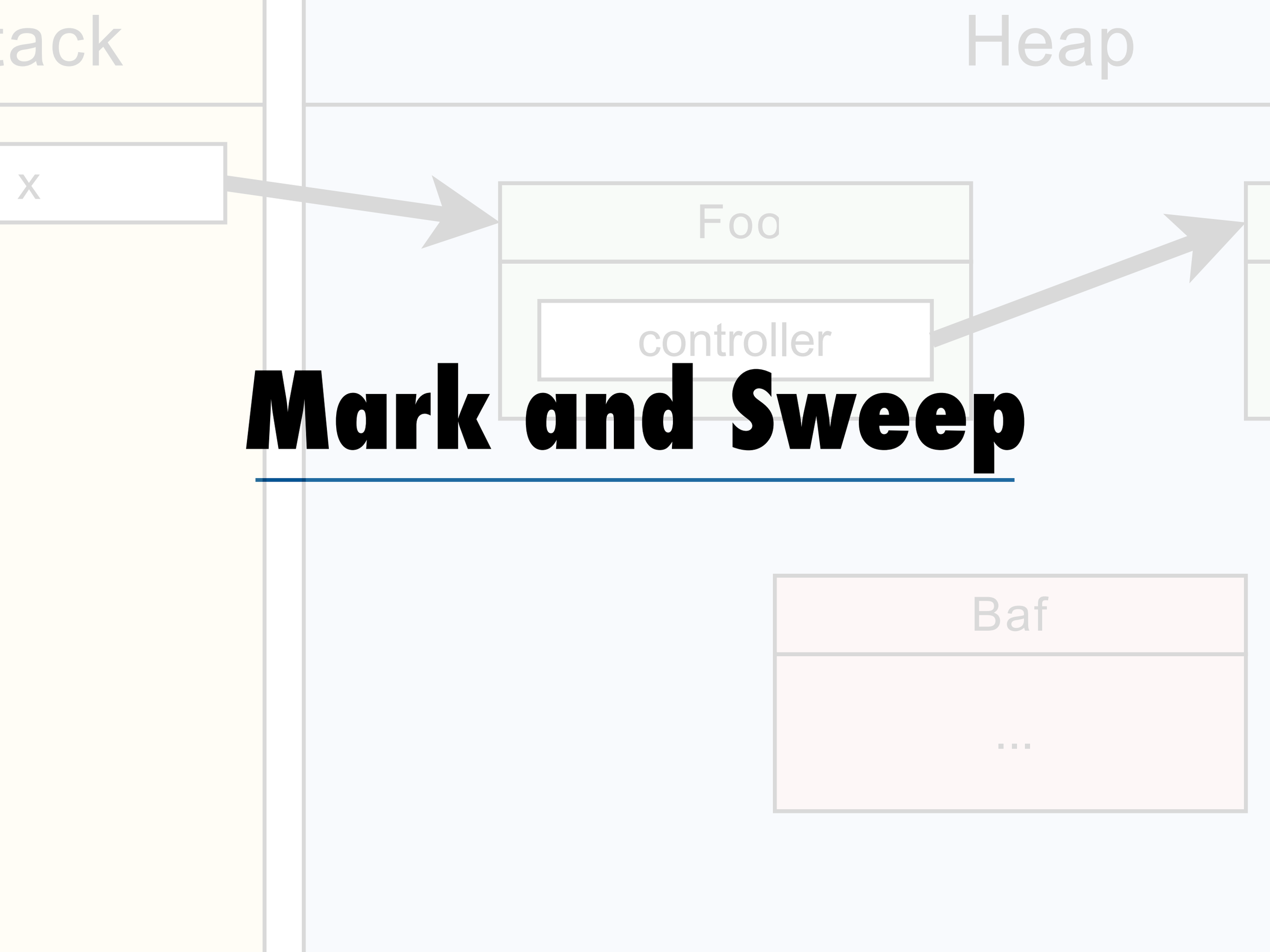


Collection

When do we perform collection?

As infrequently as possible!

(Typically, when we allocate with no free space)

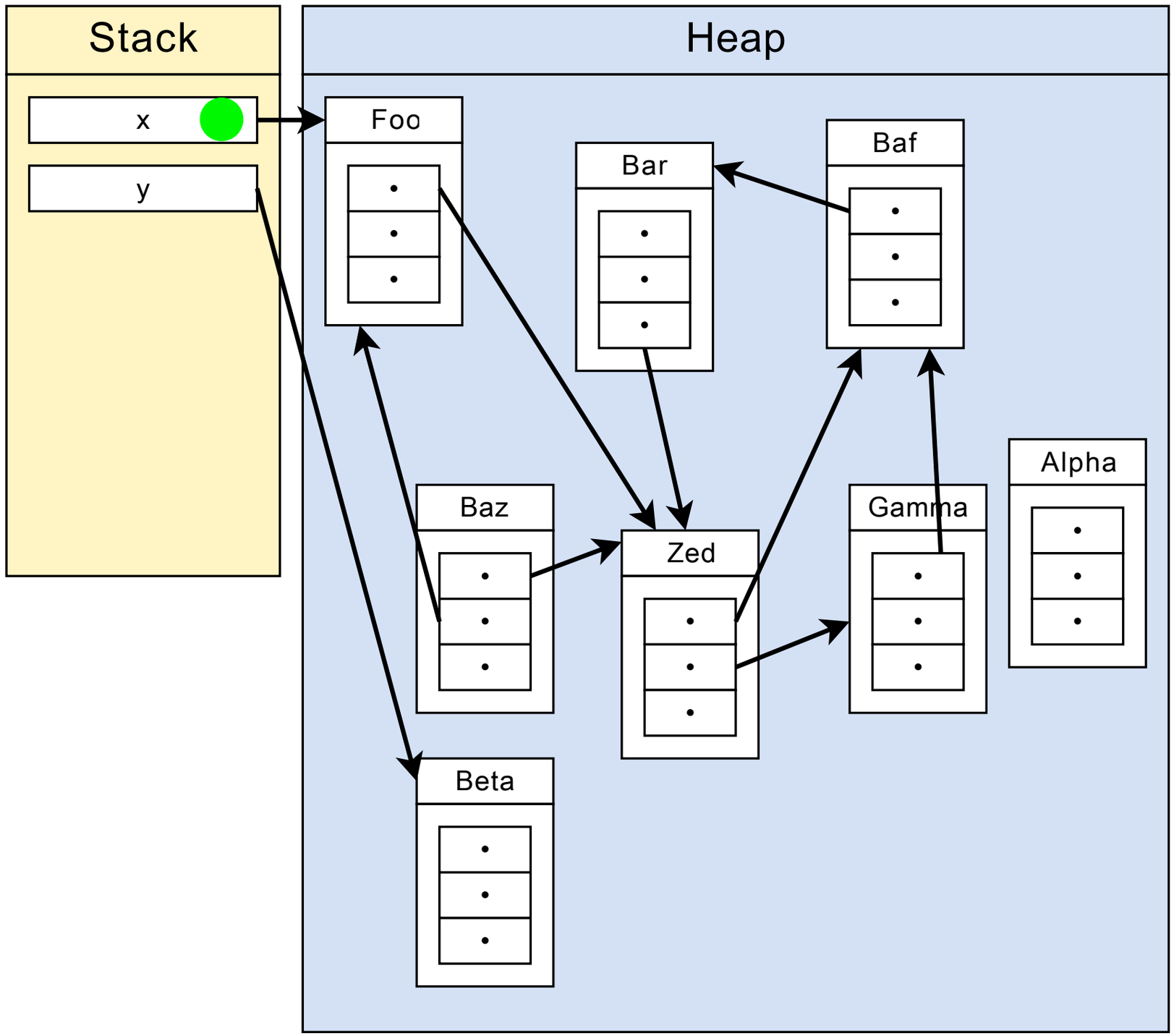


Mark and Sweep

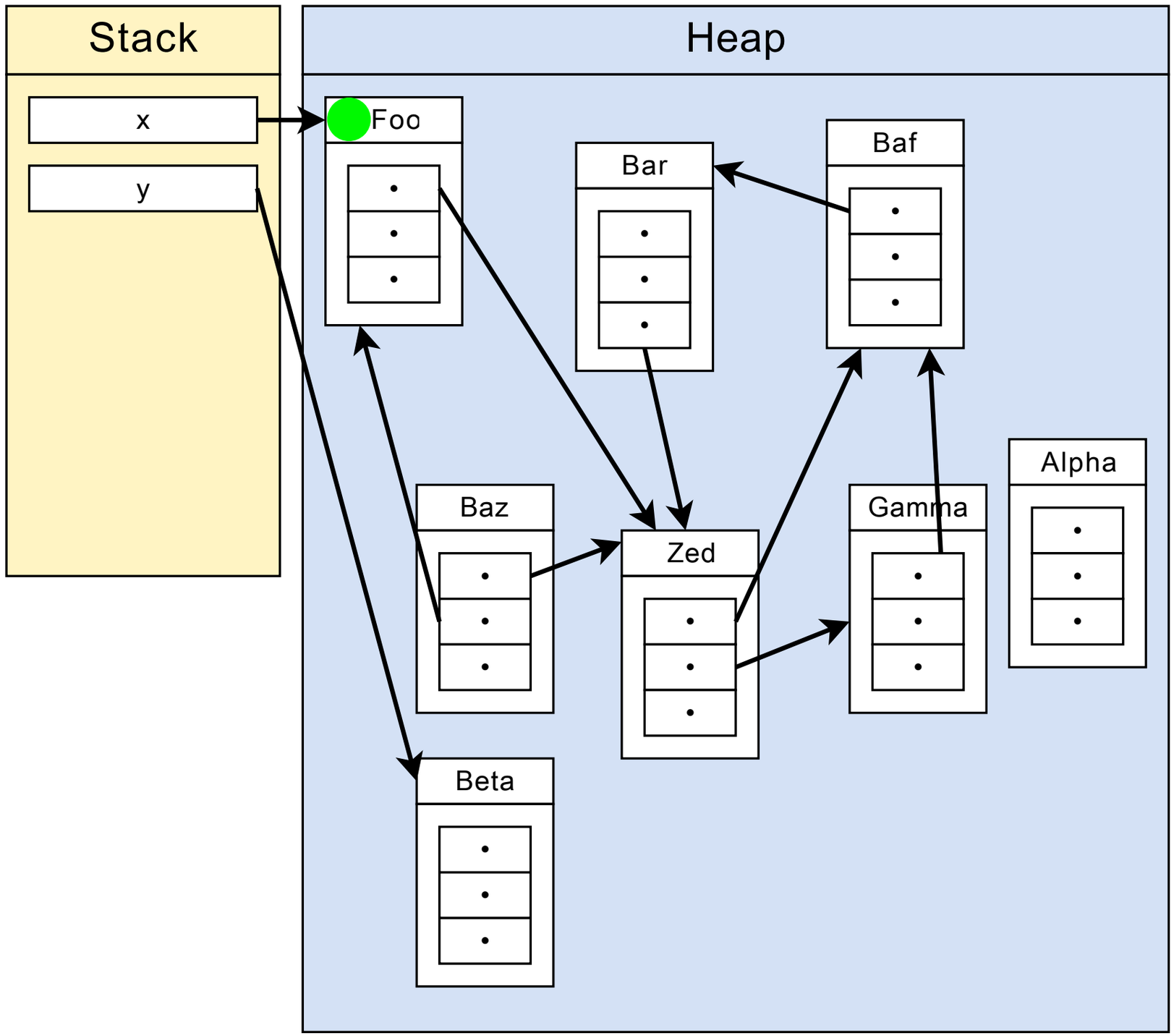
Mark and Sweep

- The heap is a graph, so:
  - Do a depth-first search to **mark** reachable objects
  - Iterate over heap to **sweep** unreachable objects

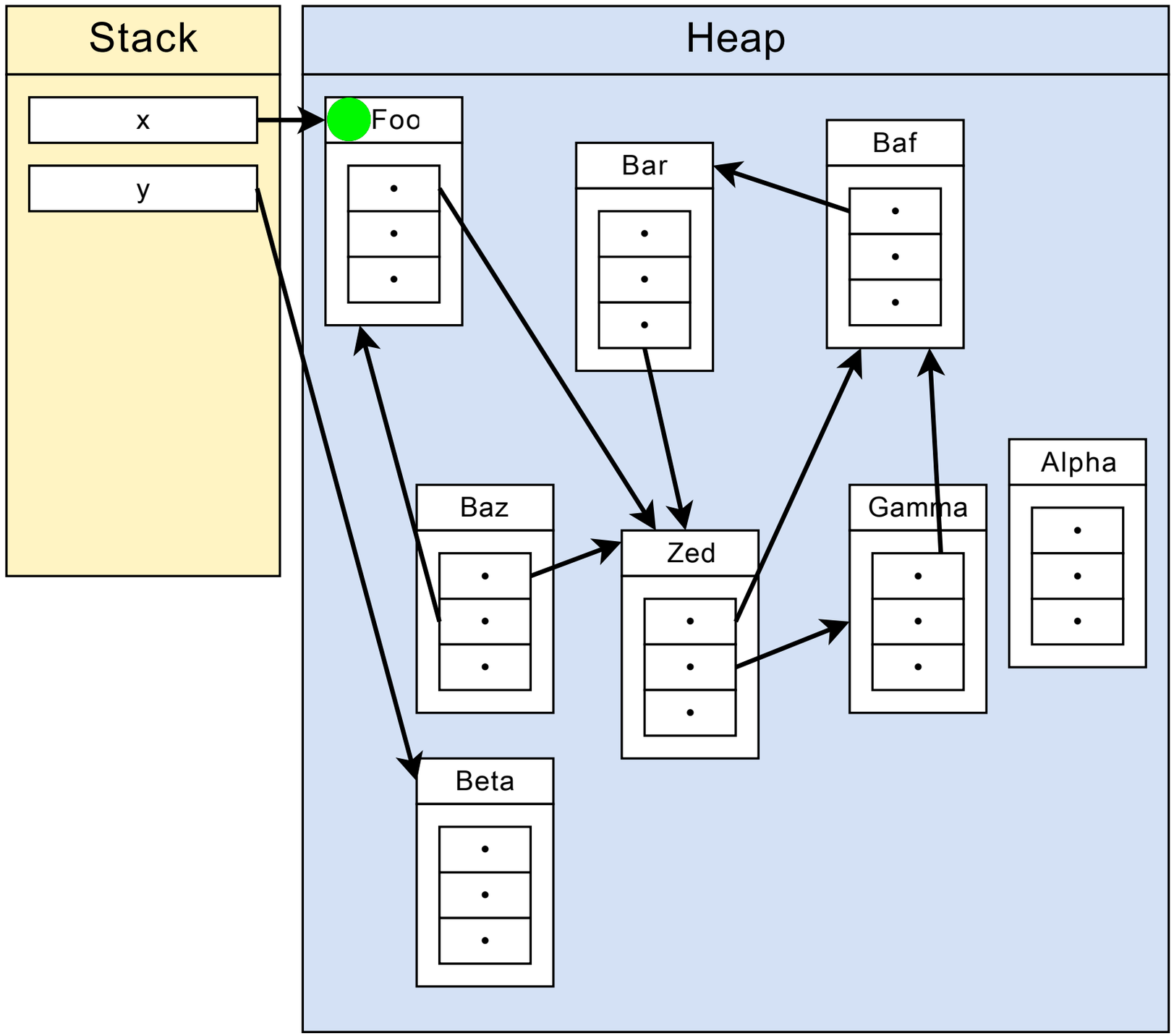
Mark



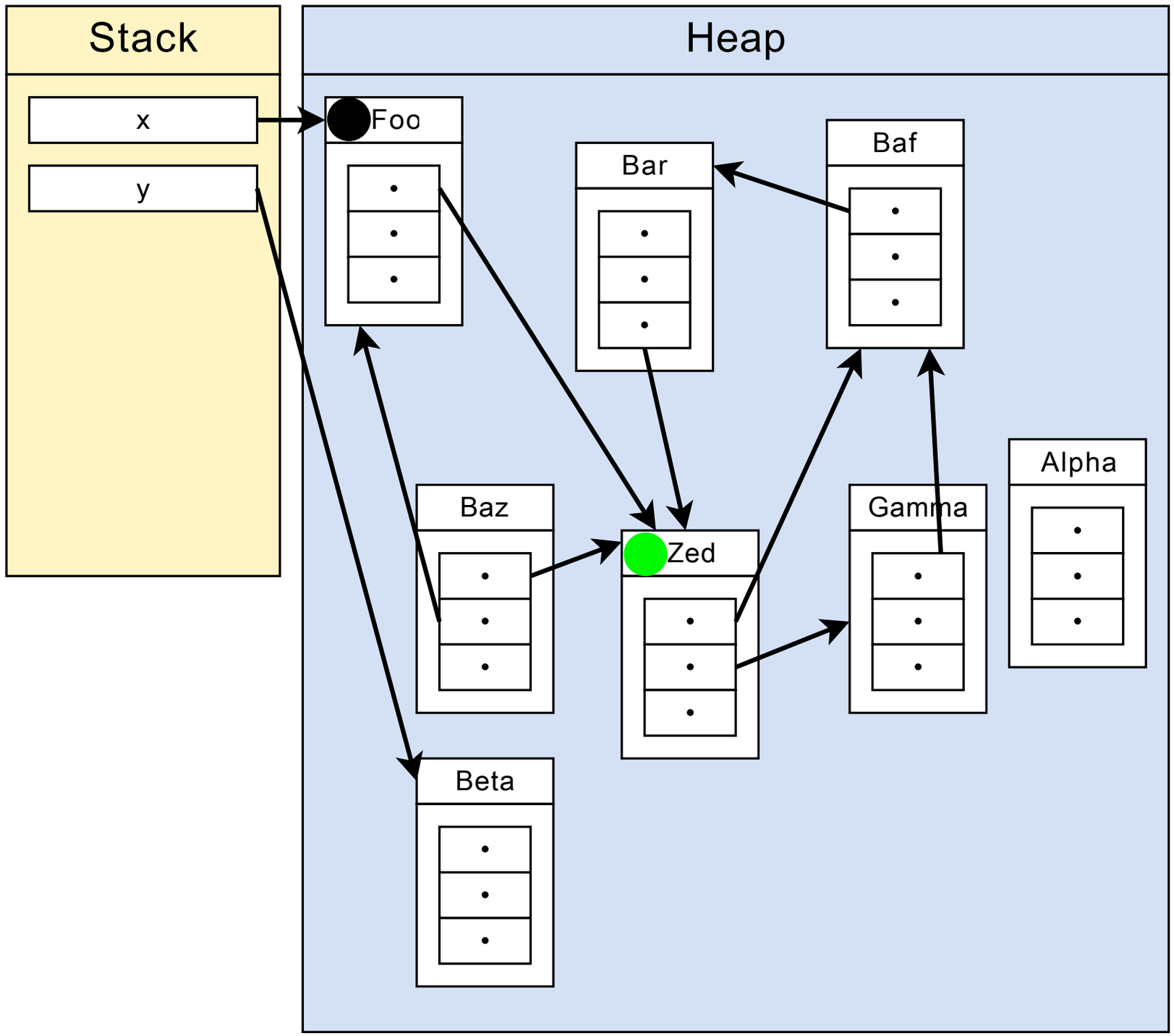
Mark



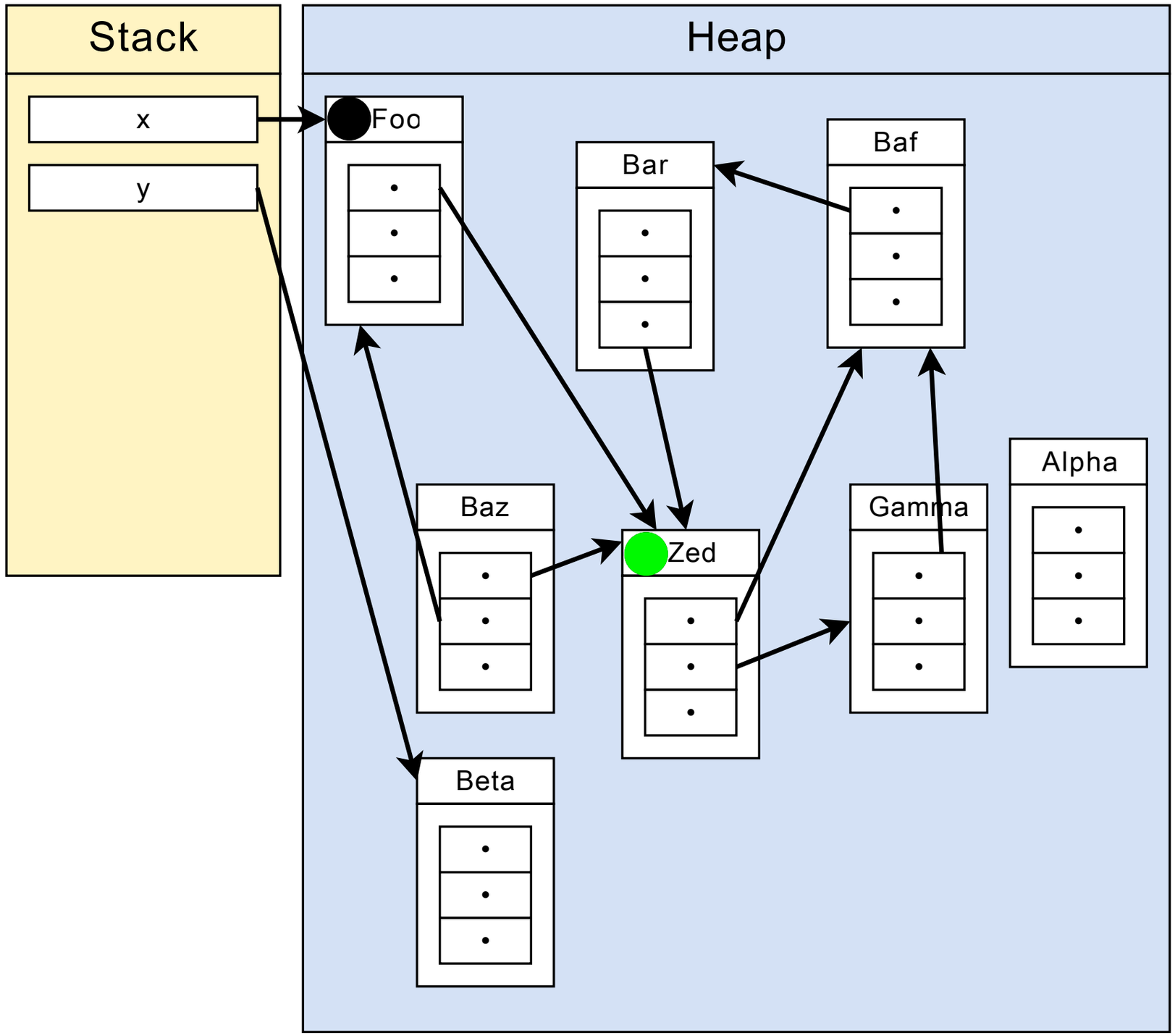
Mark



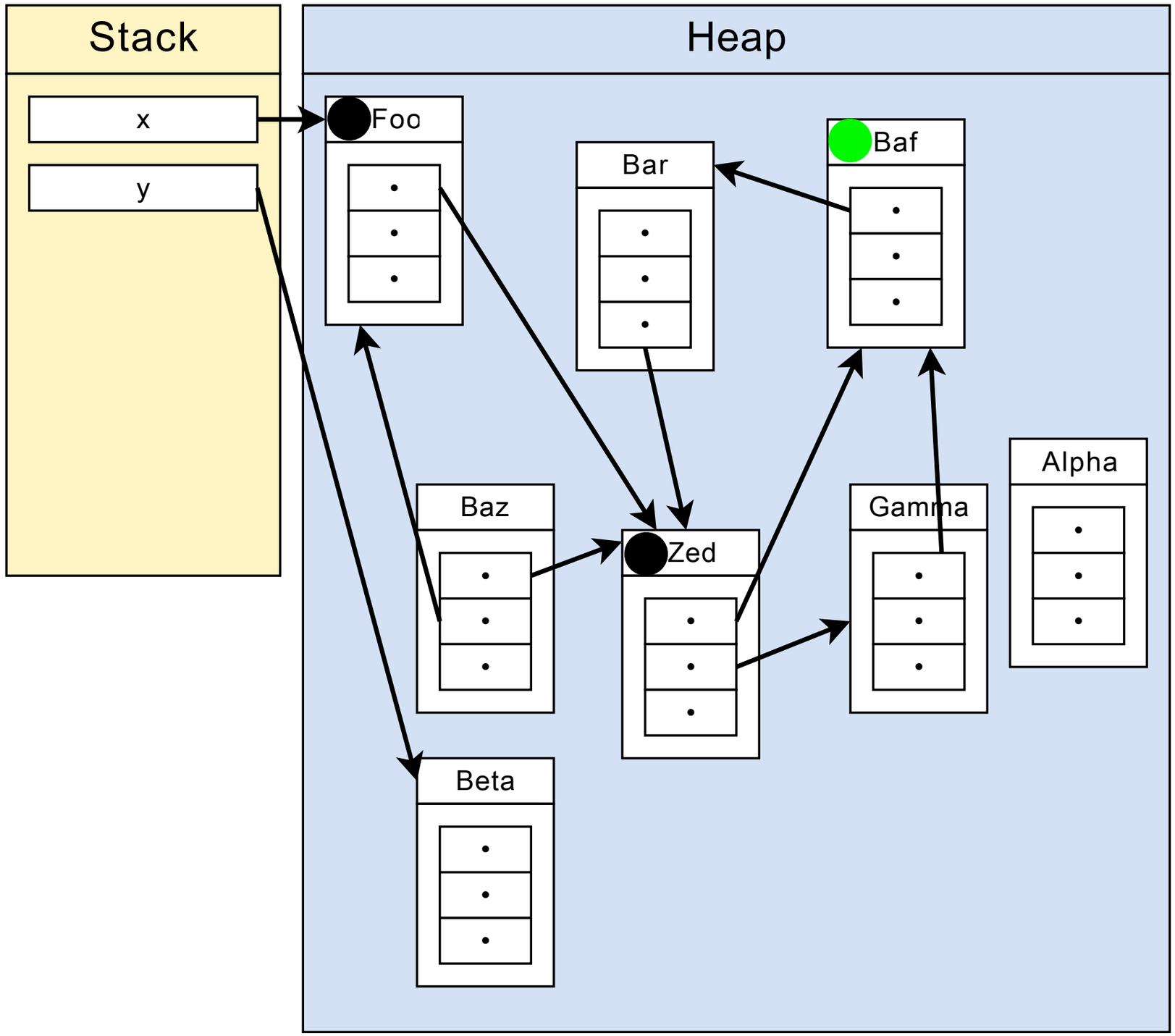
Mark



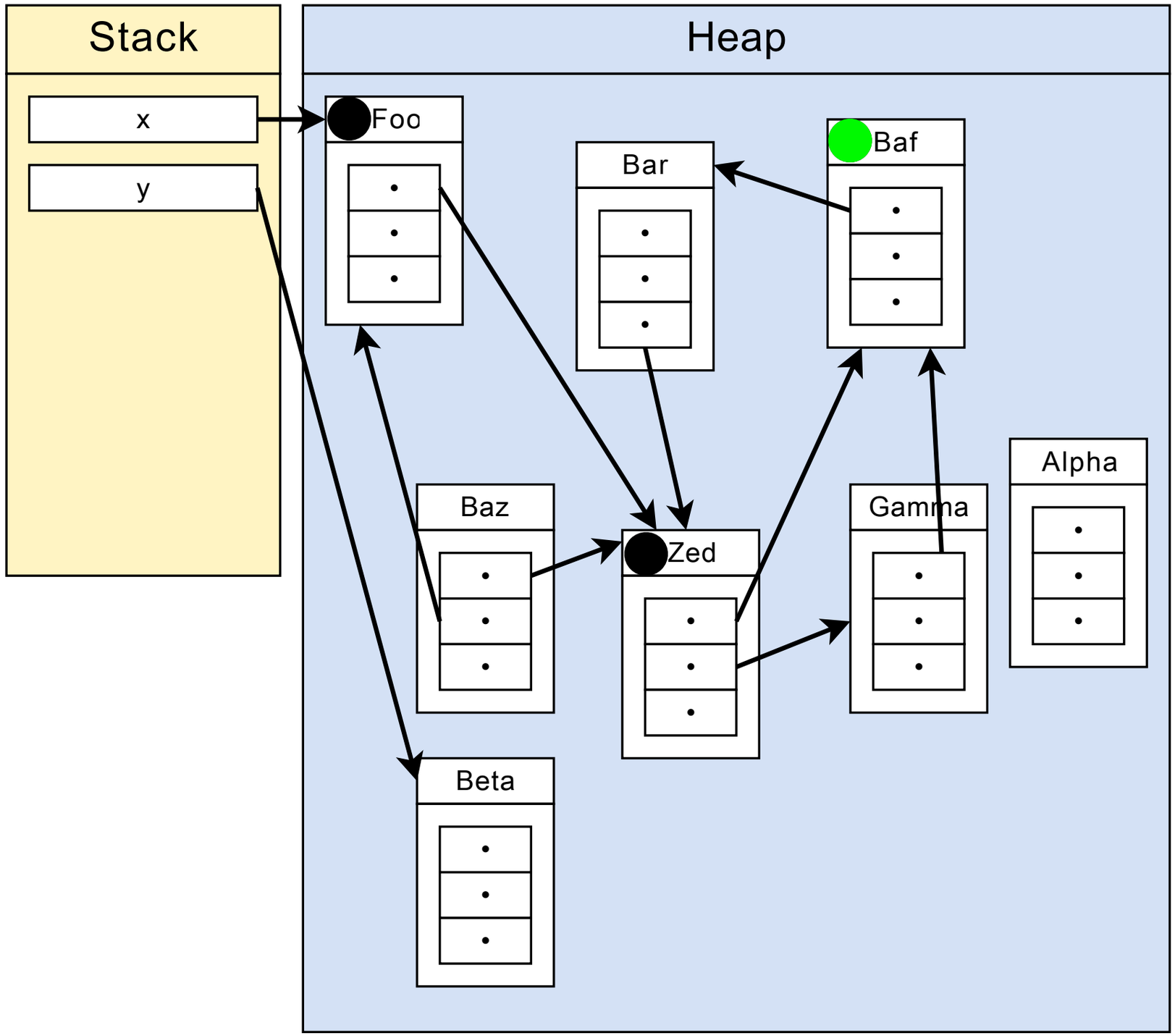
Mark



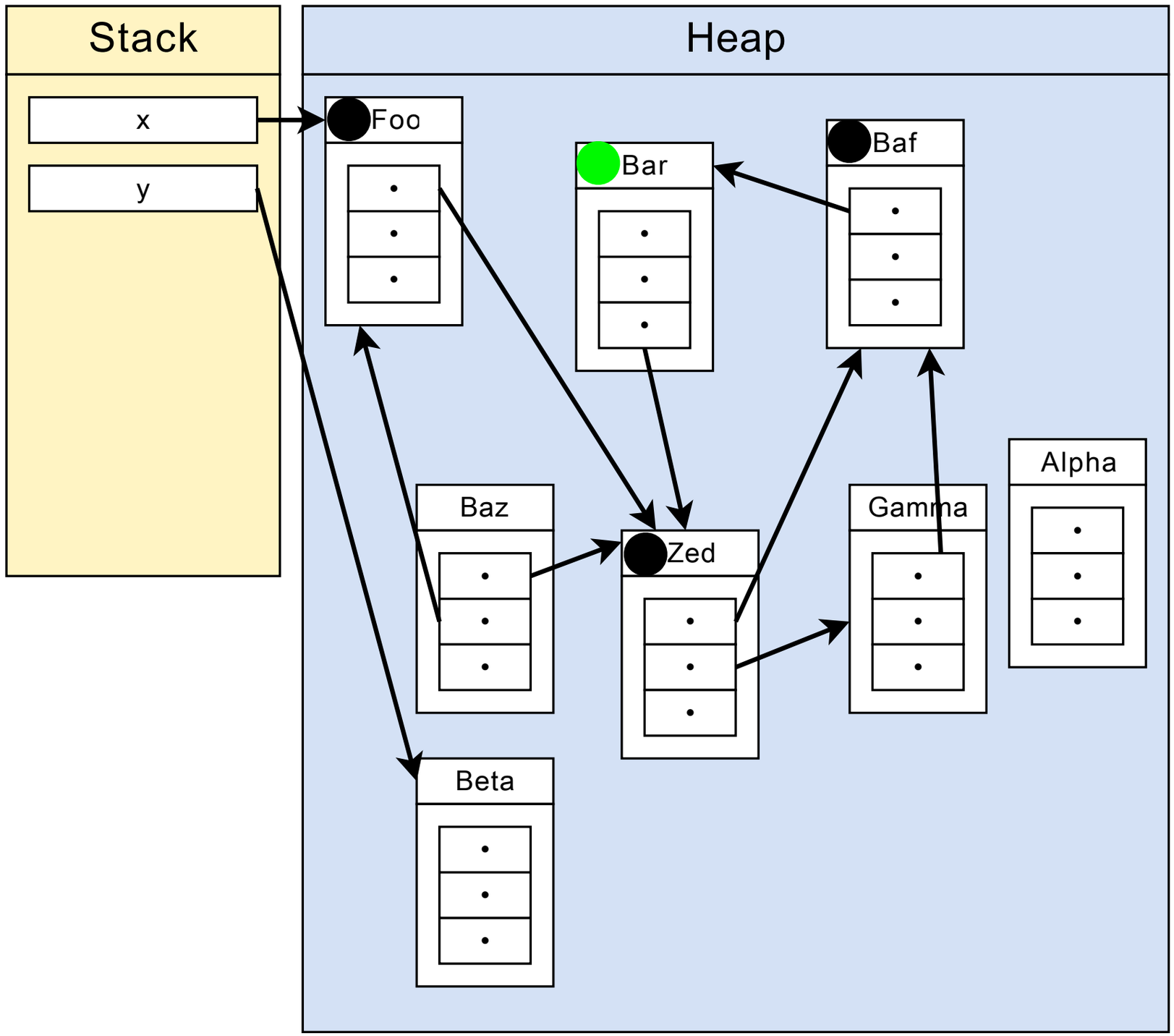
Mark



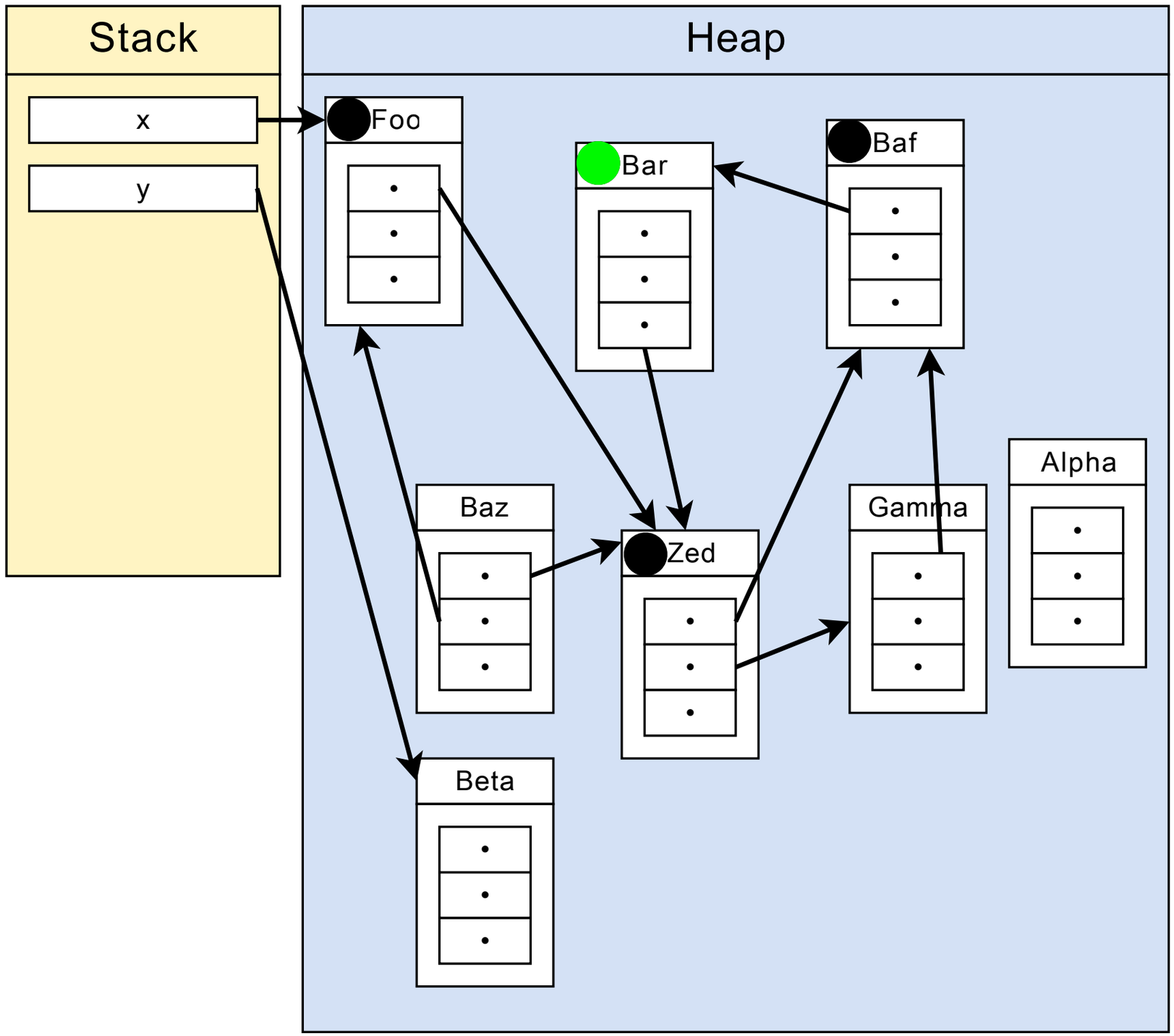
Mark



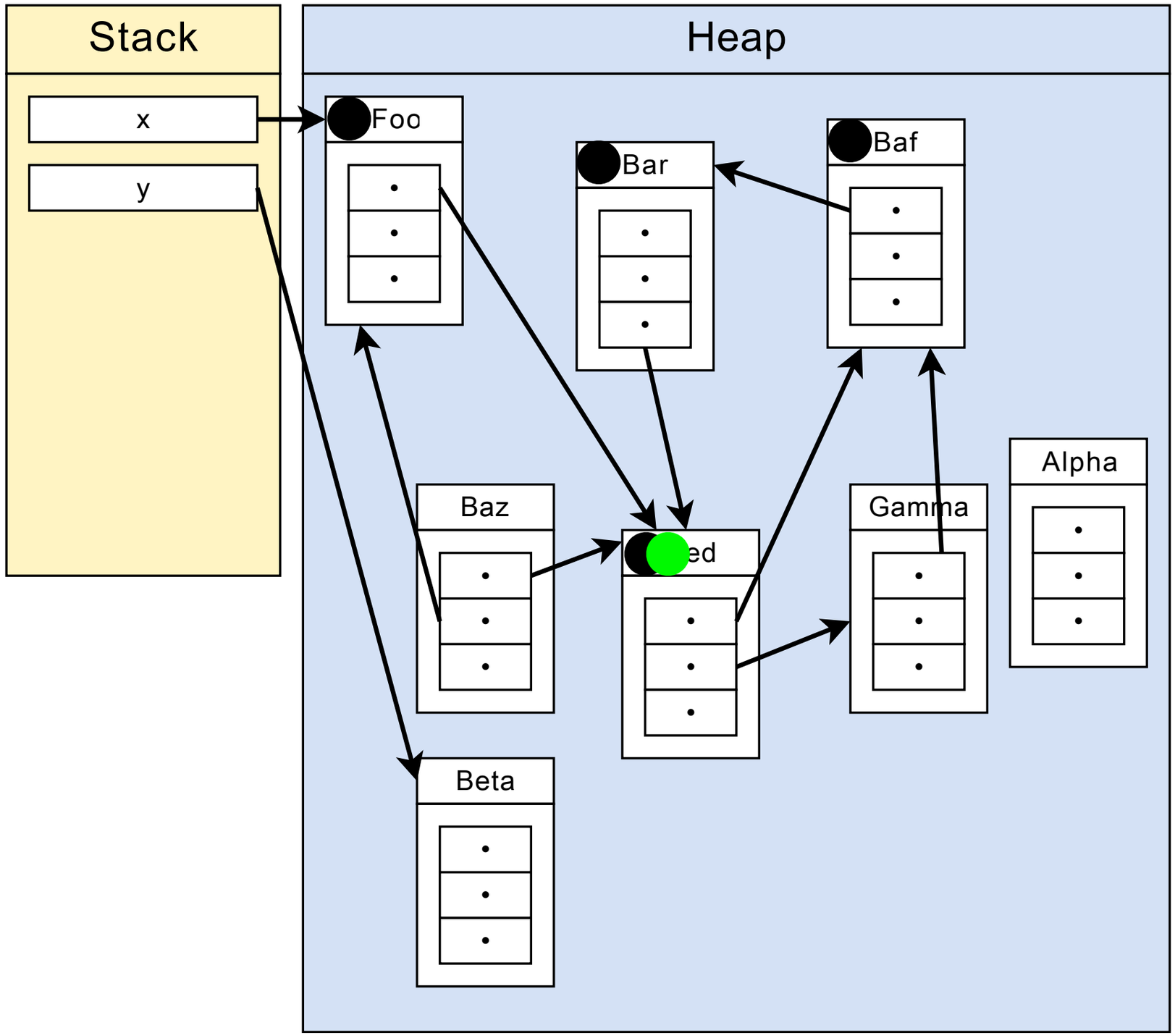
Mark



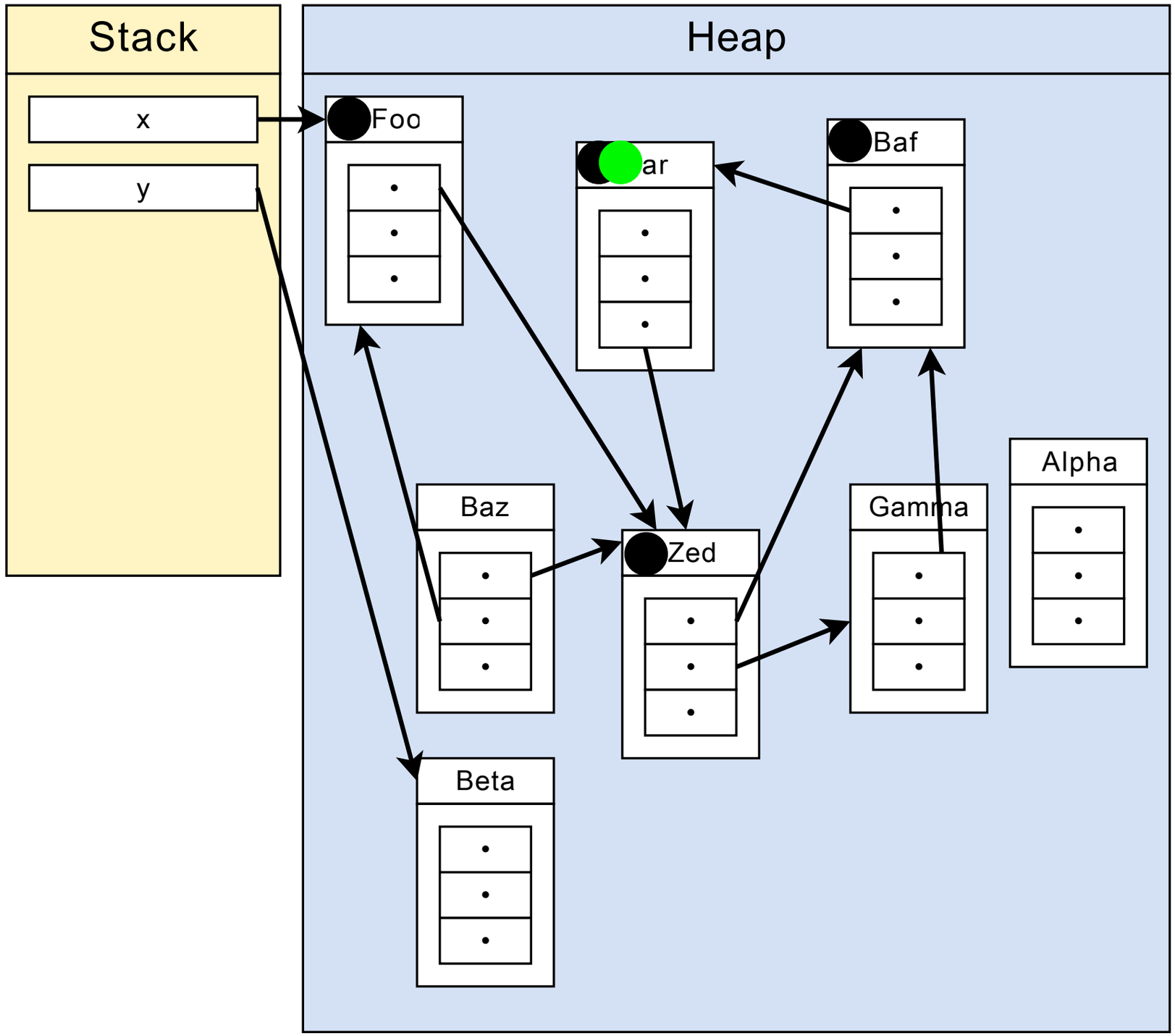
Mark



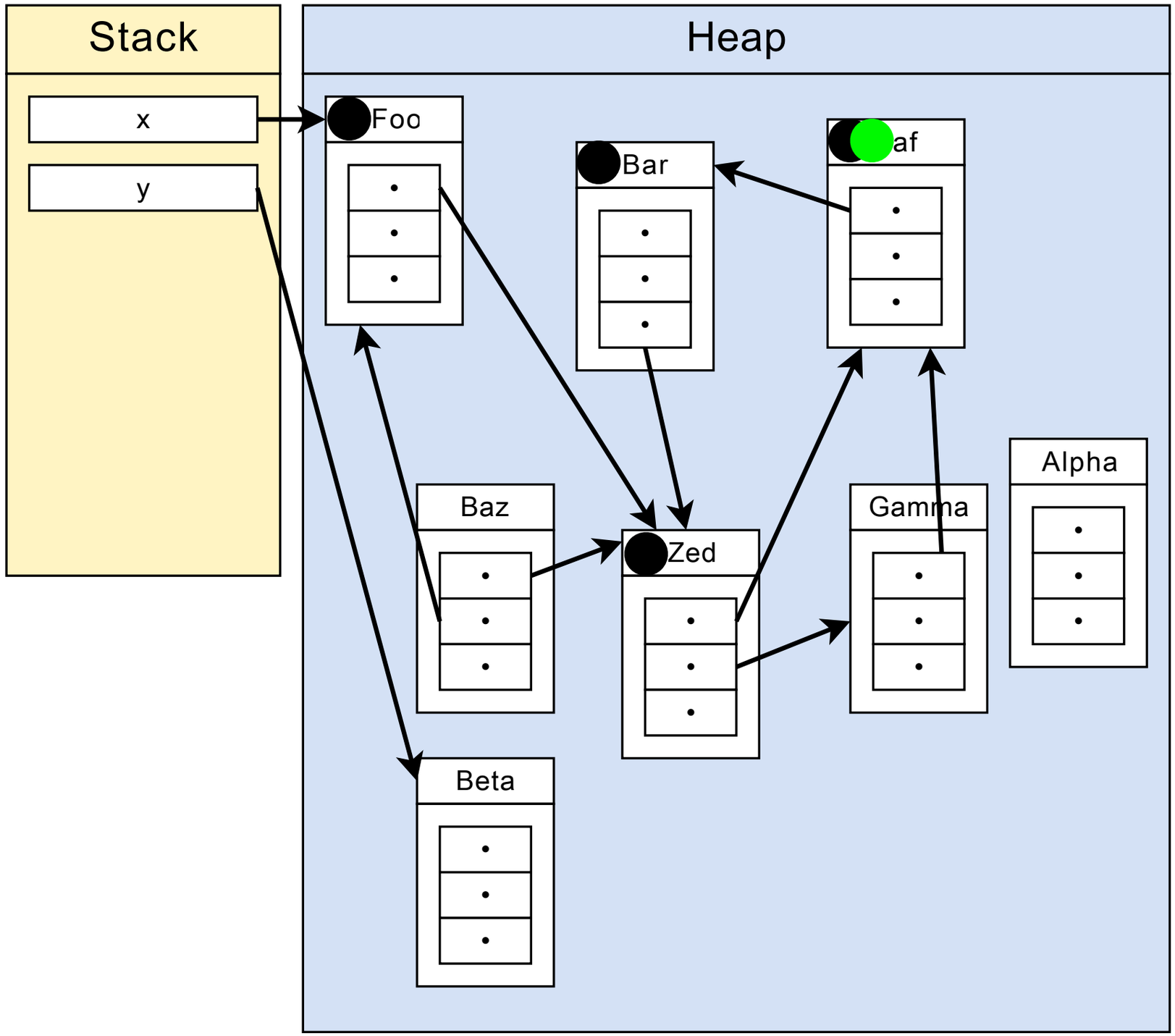
Mark



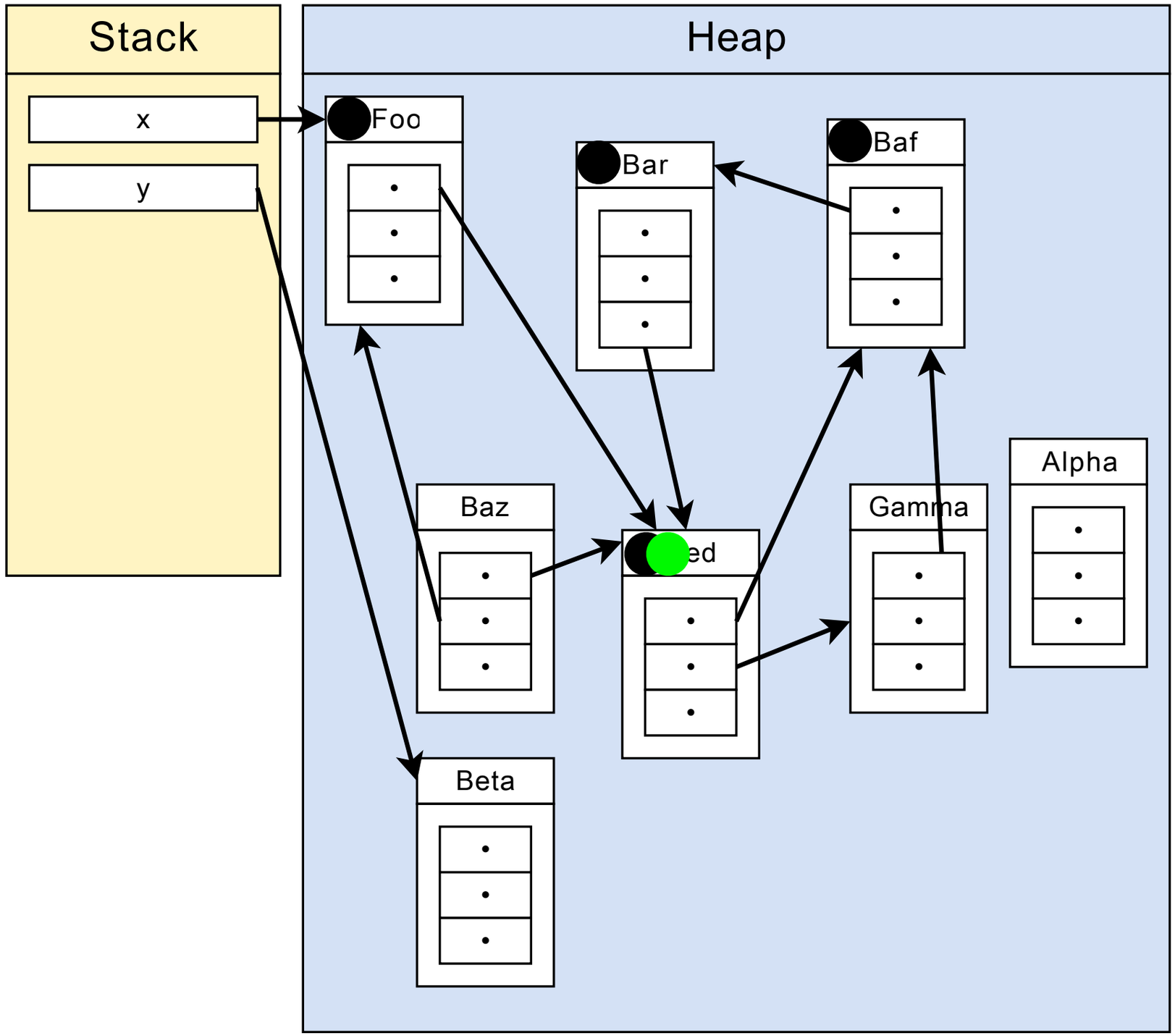
Mark



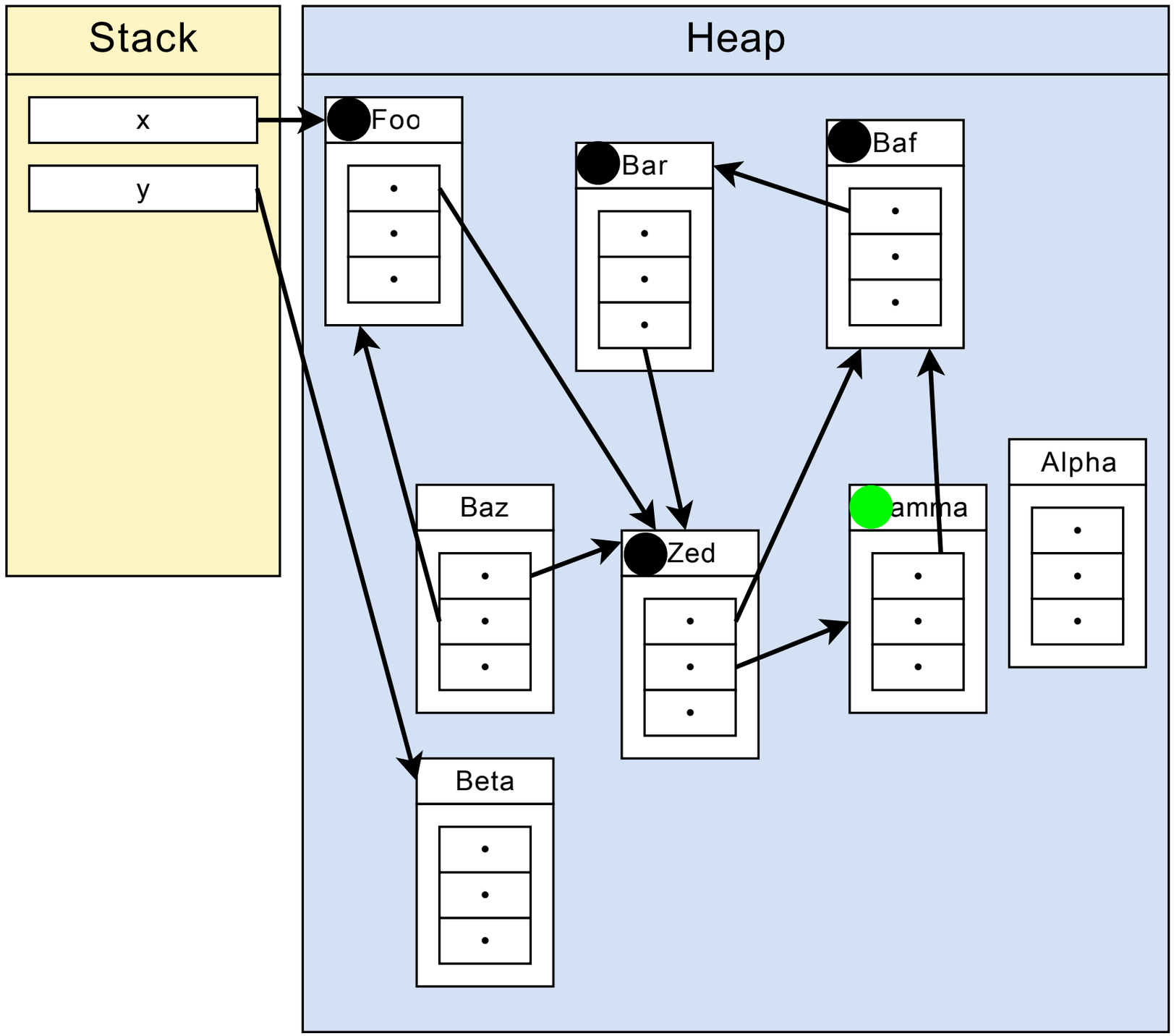
Mark



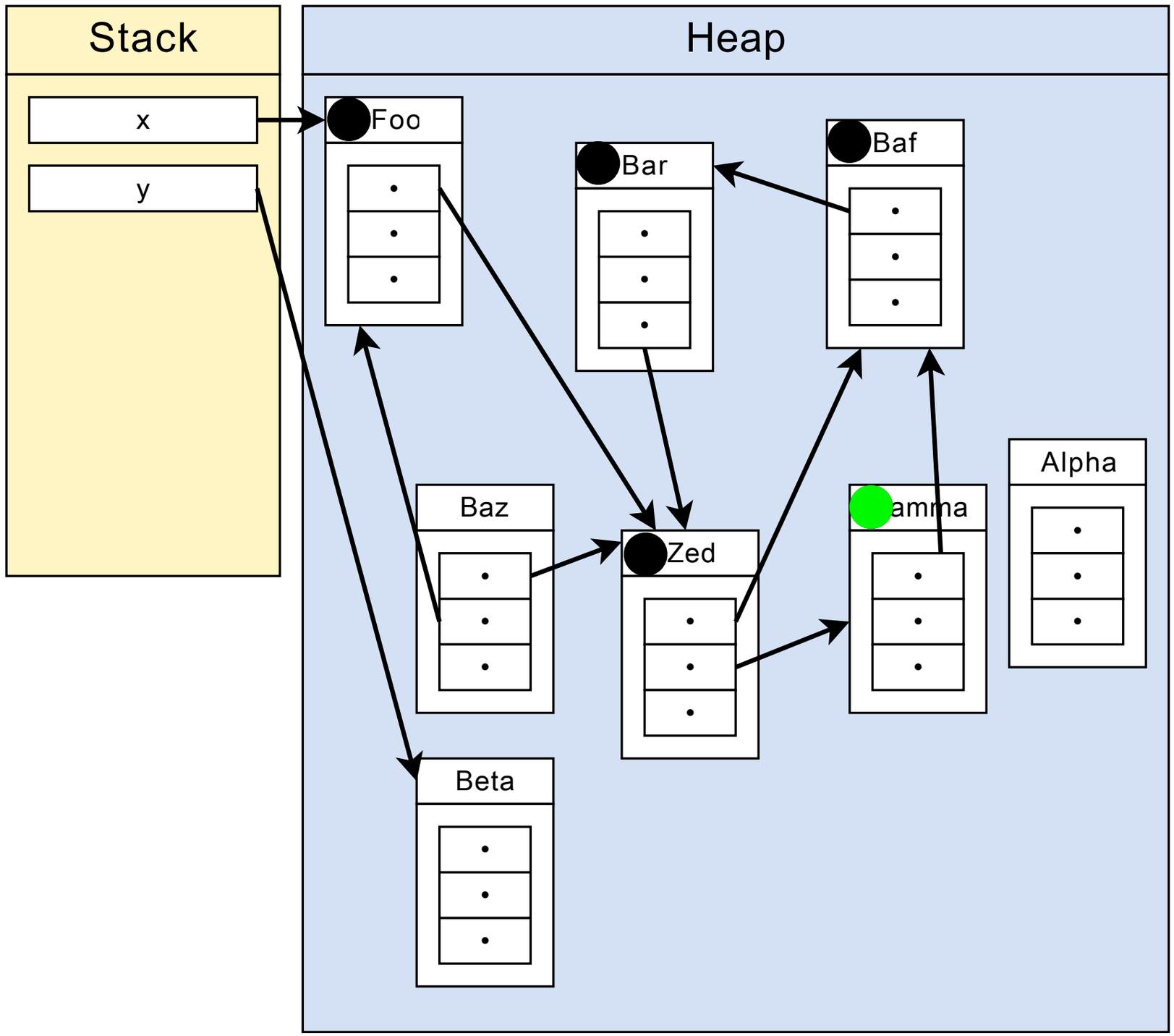
Mark



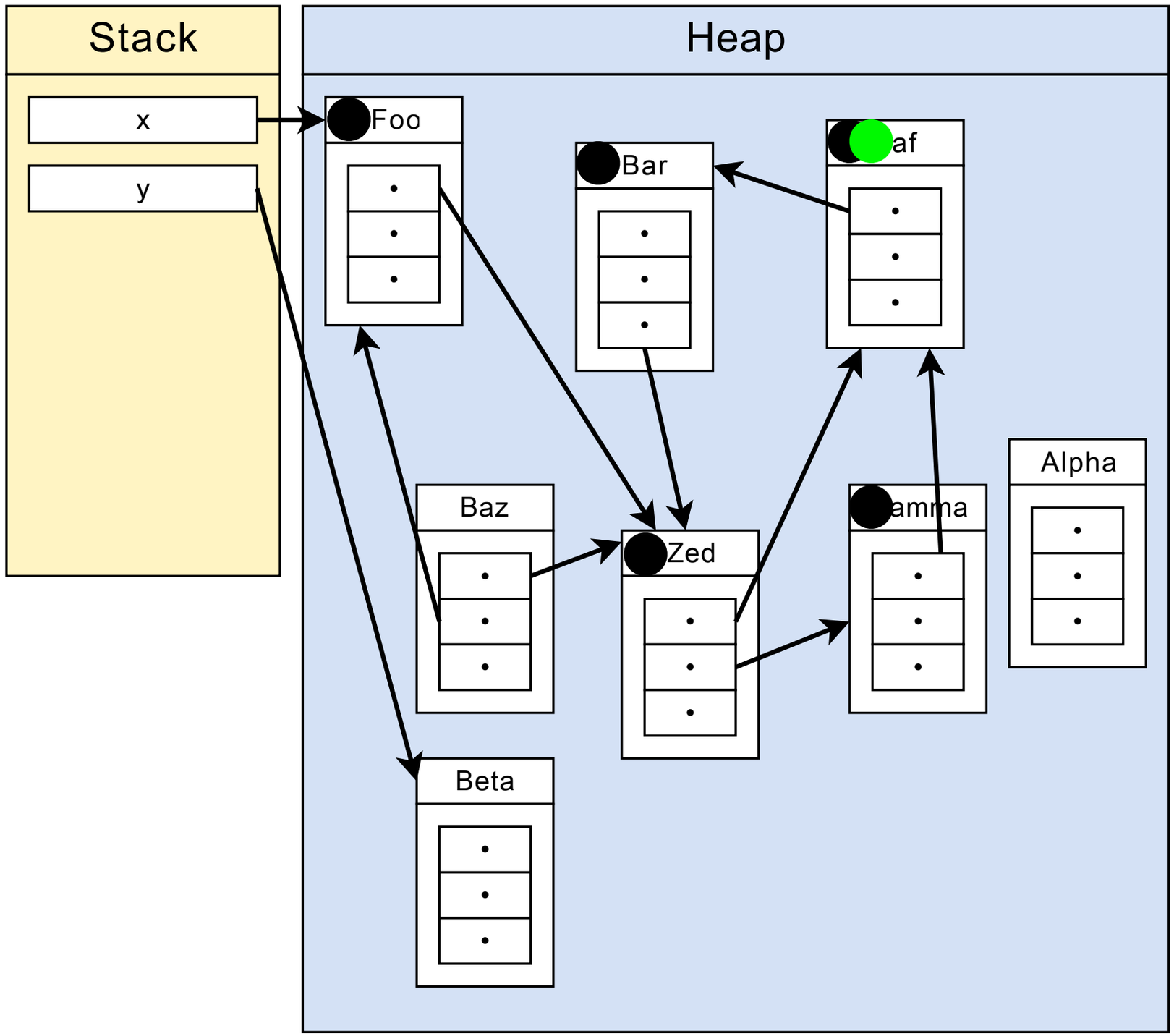
Mark



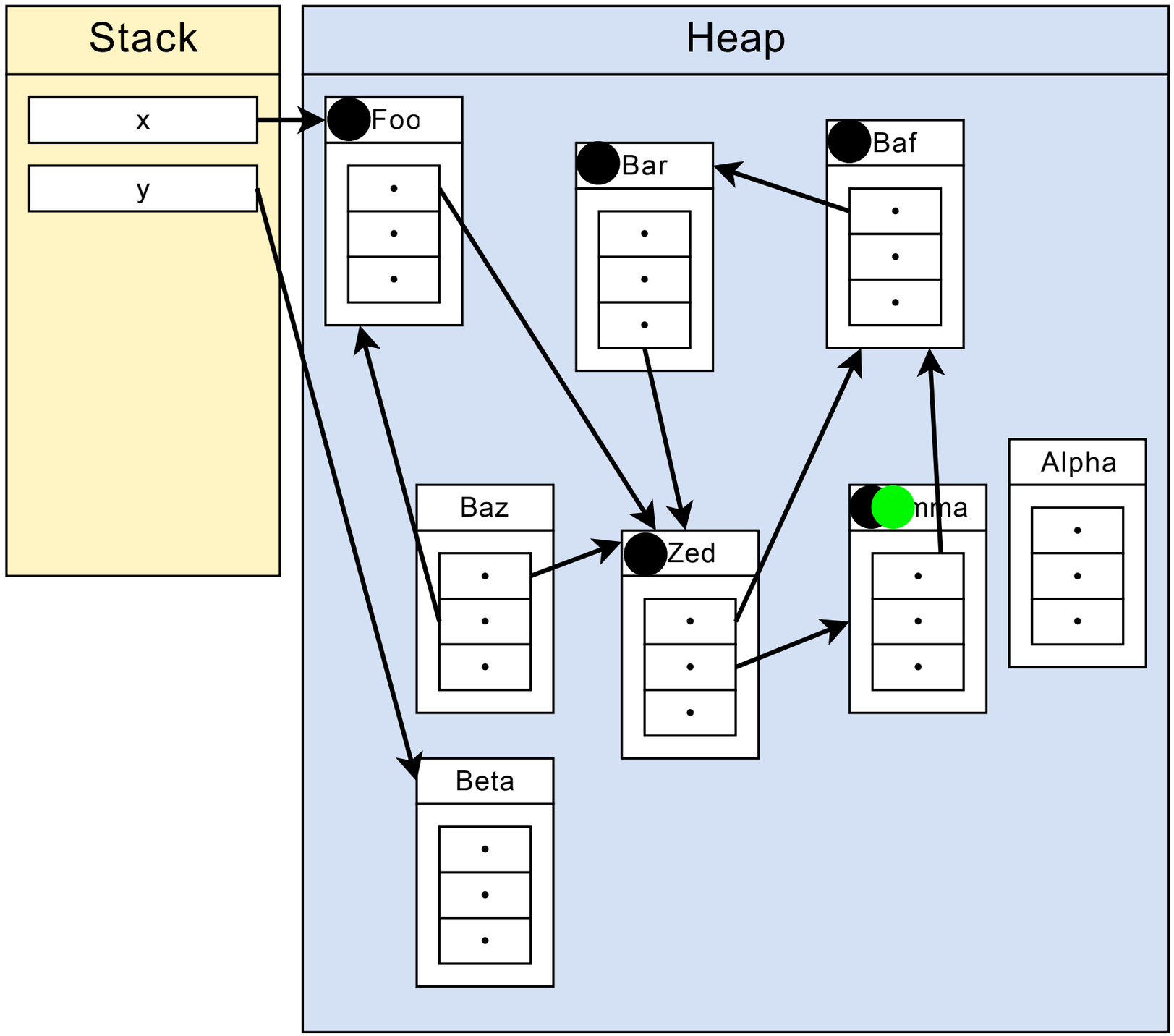
Mark



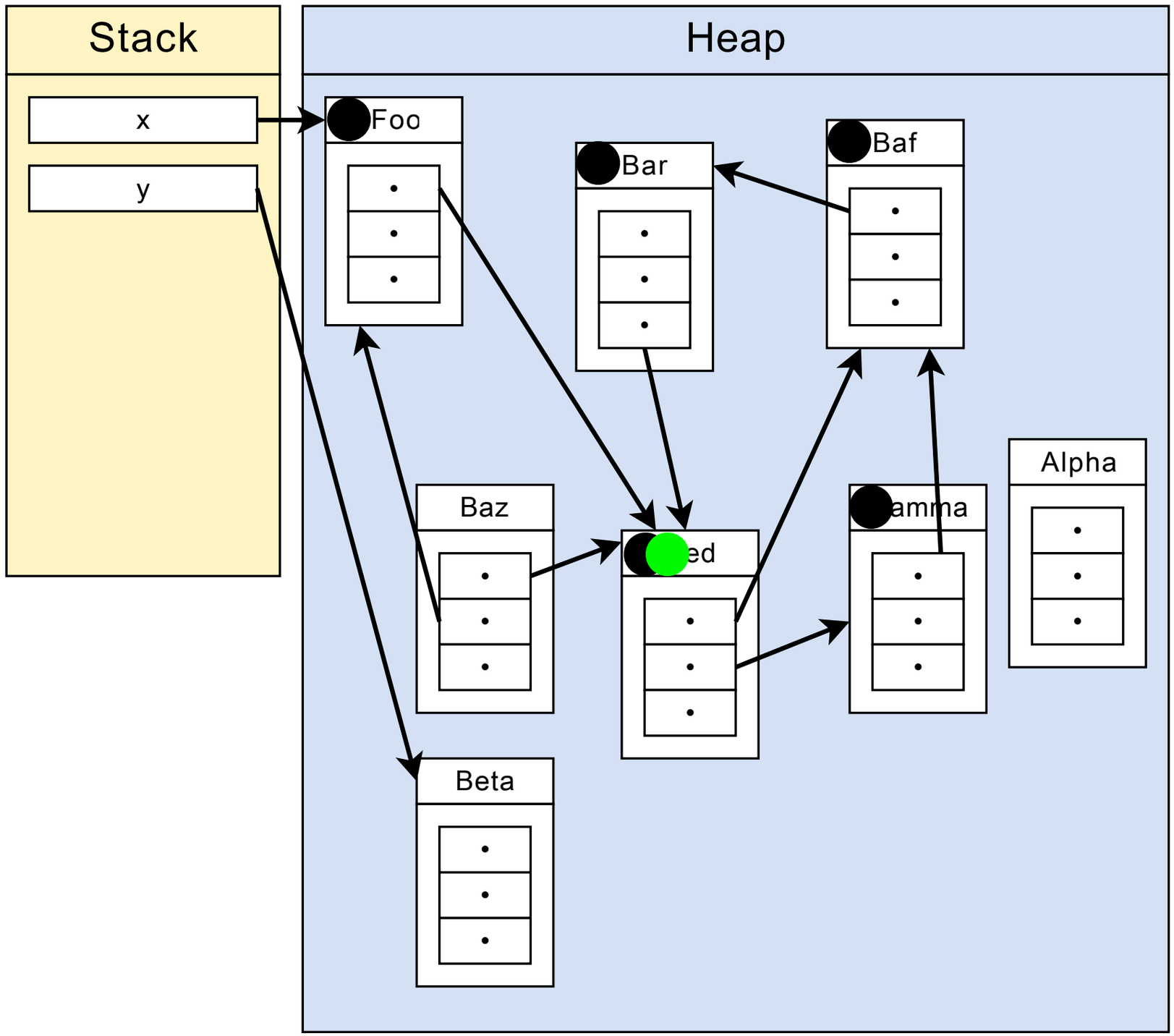
Mark



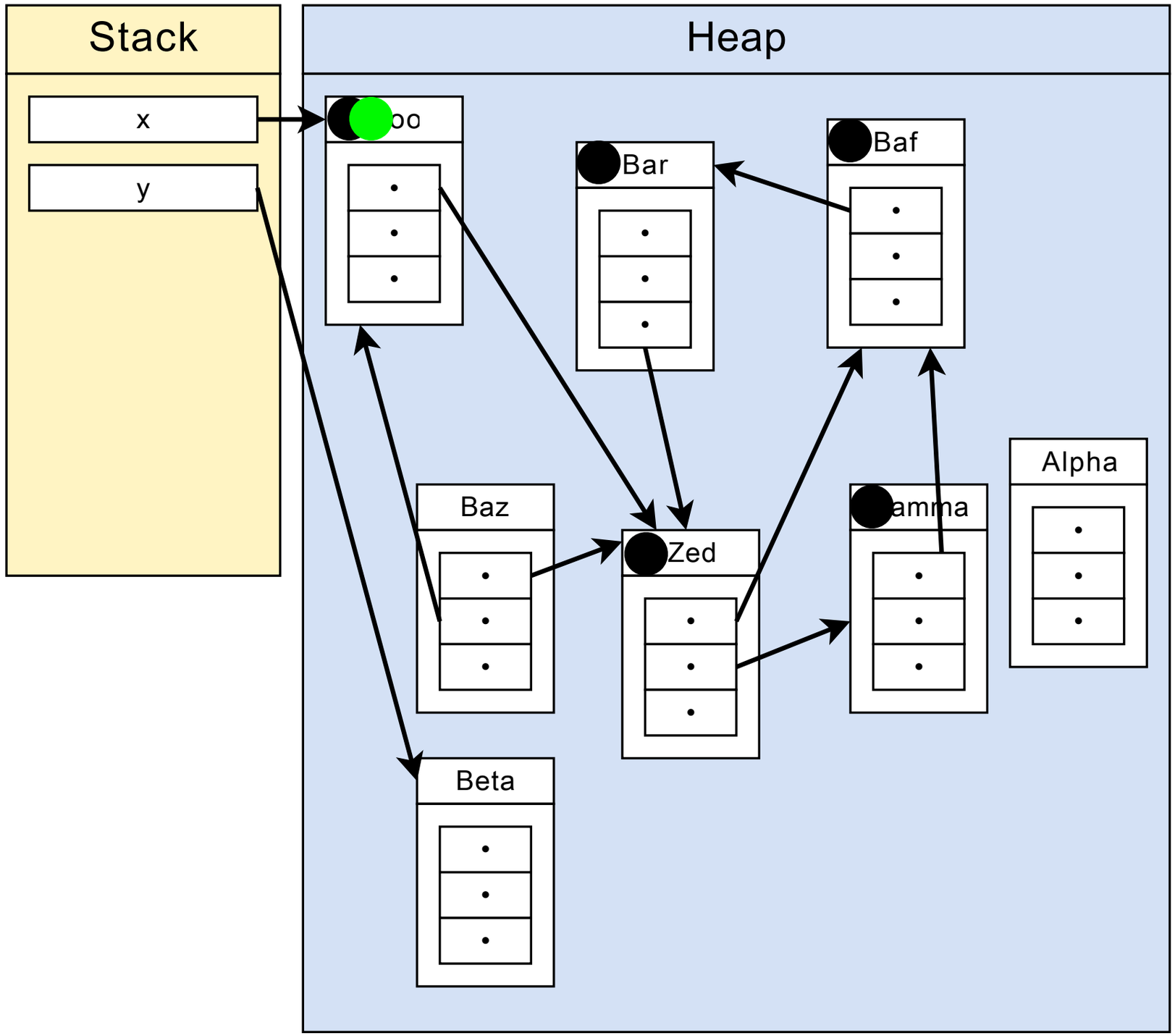
Mark



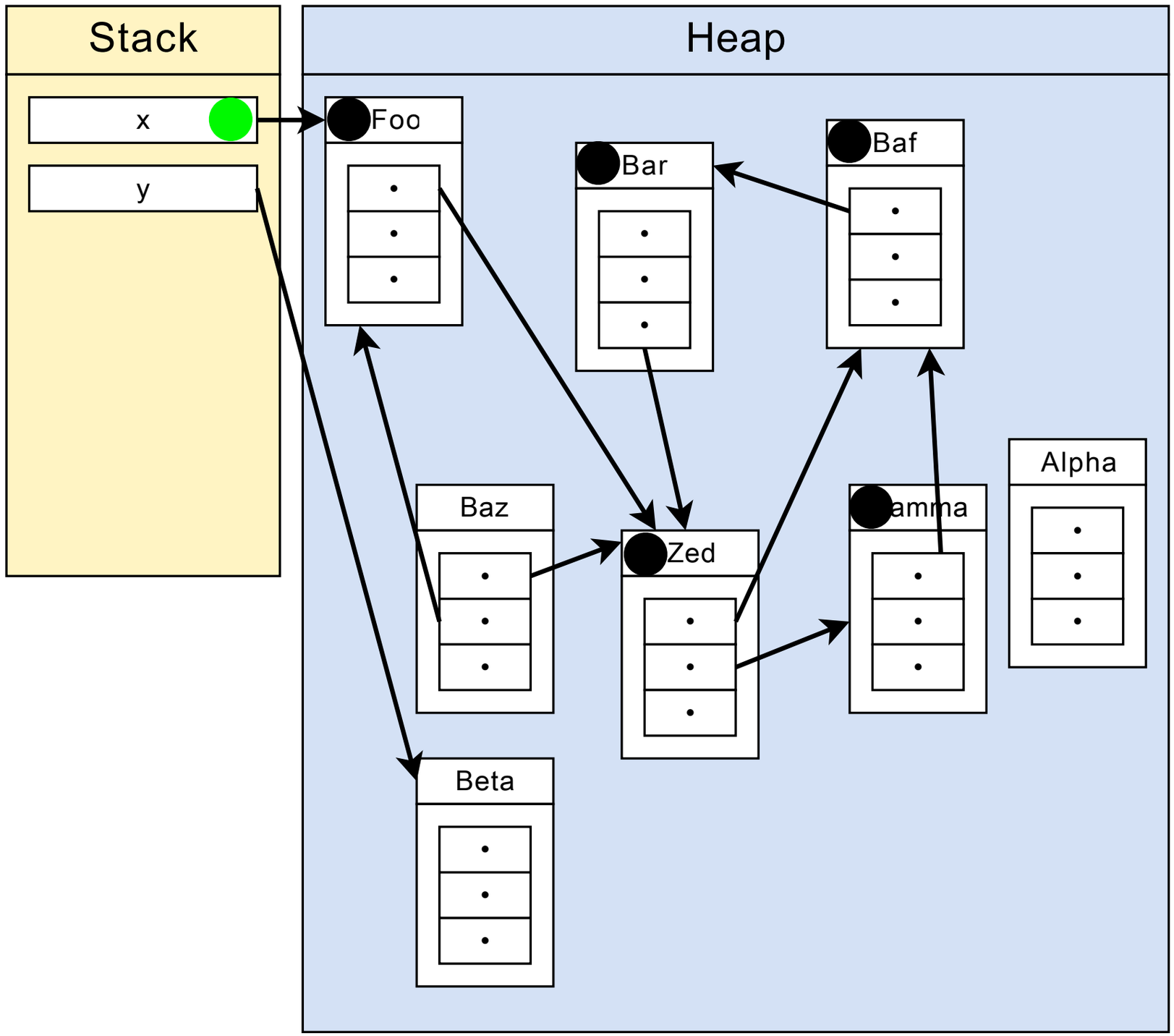
Mark



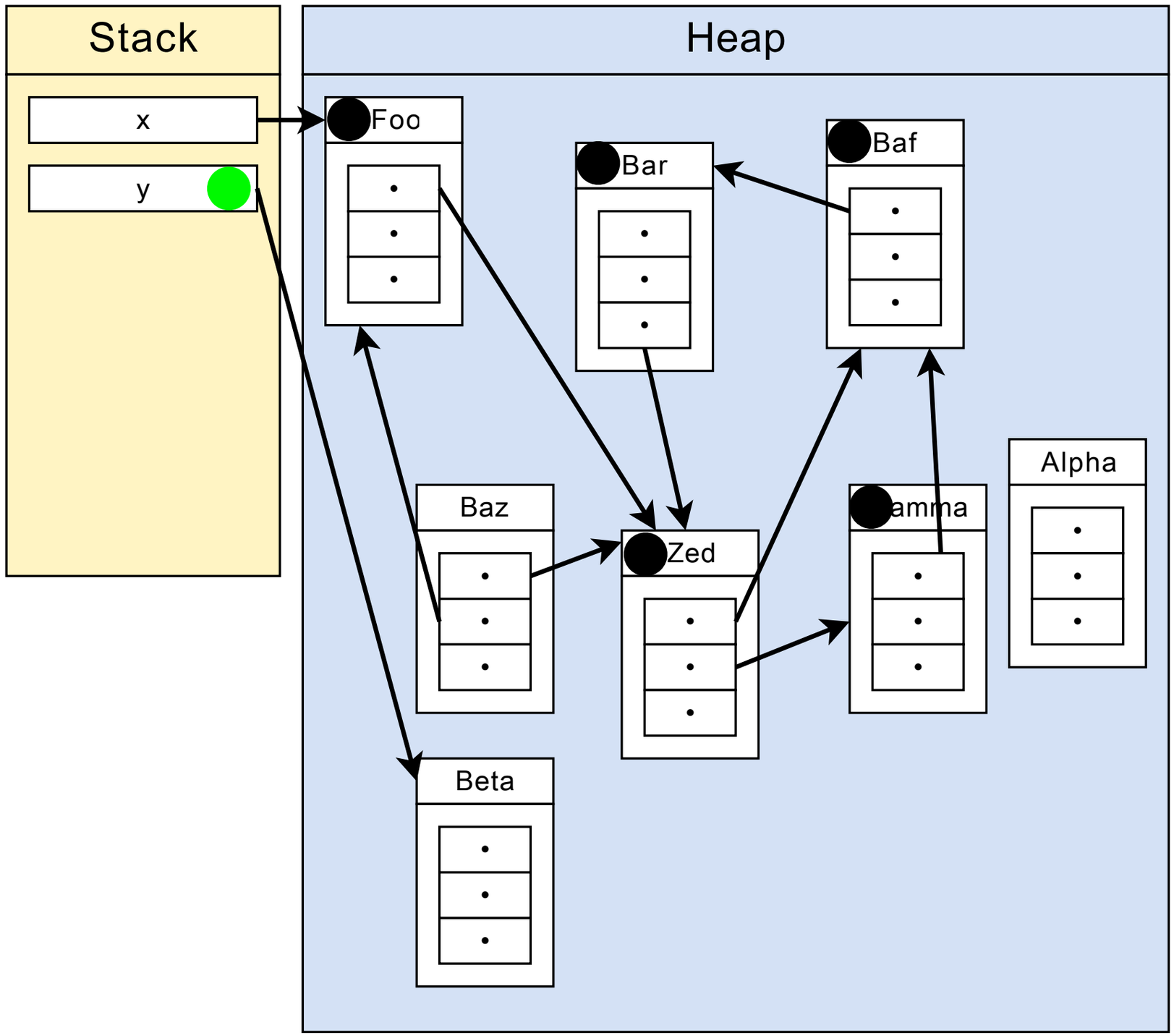
Mark



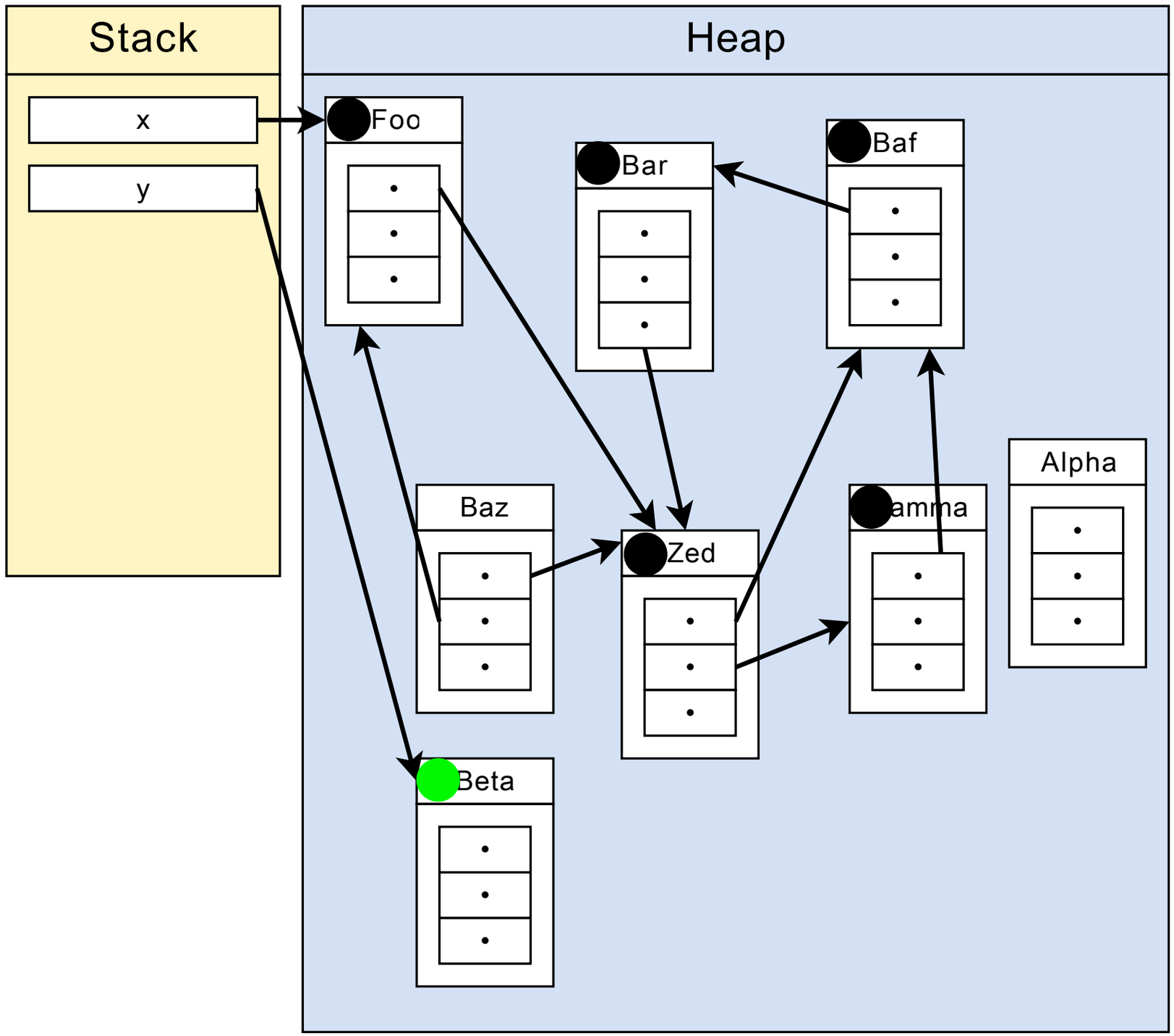
Mark



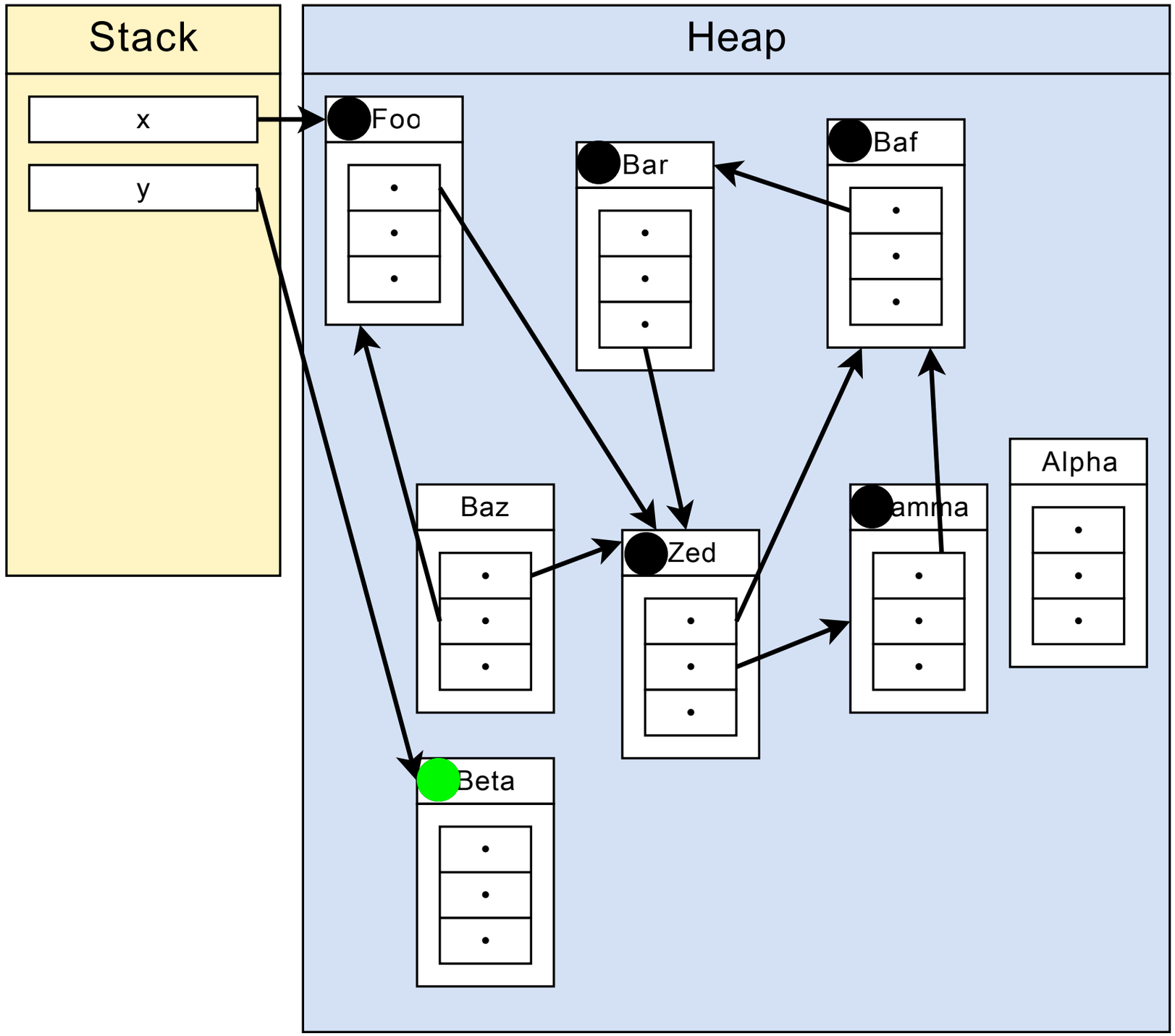
Mark



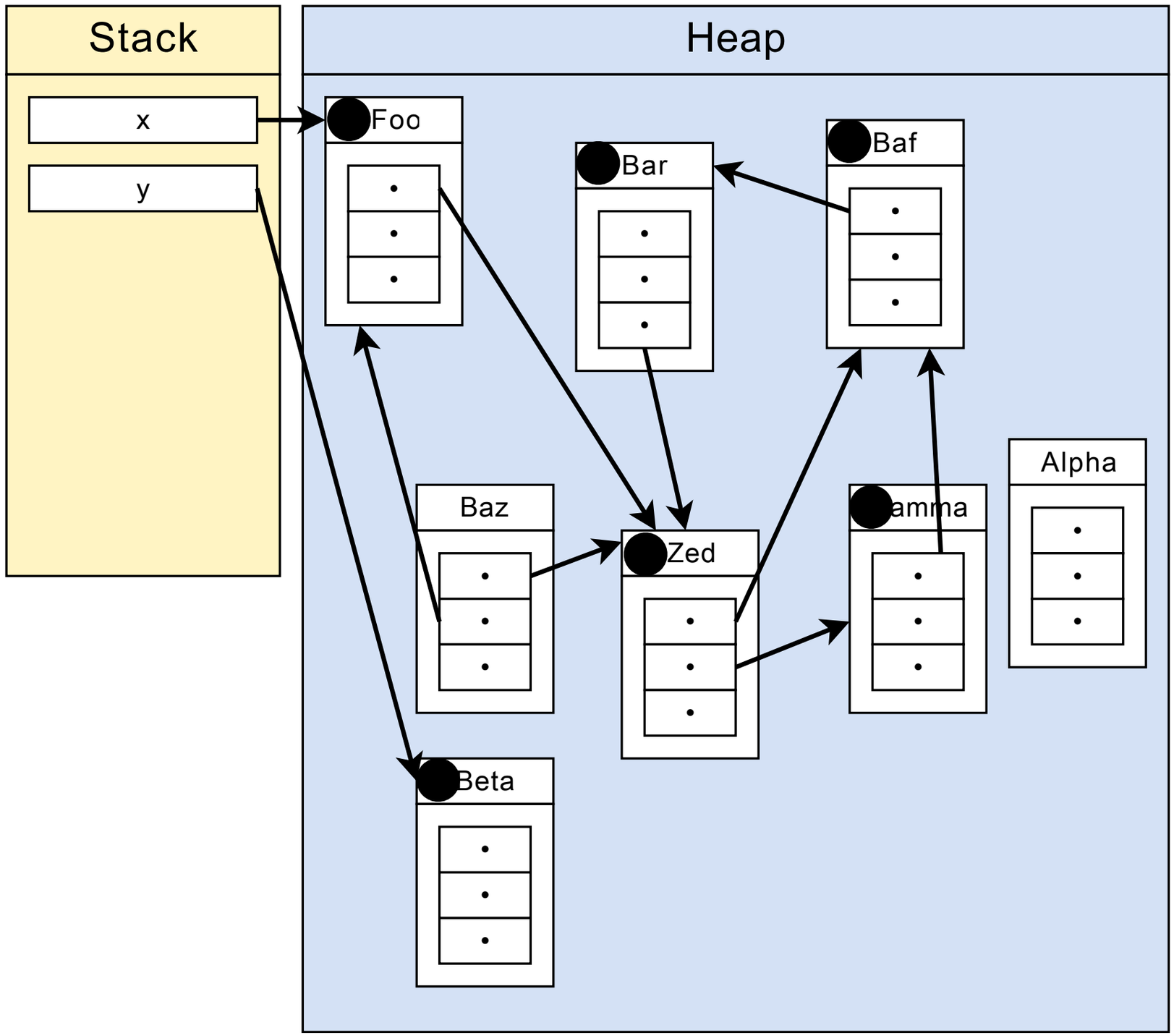
Mark



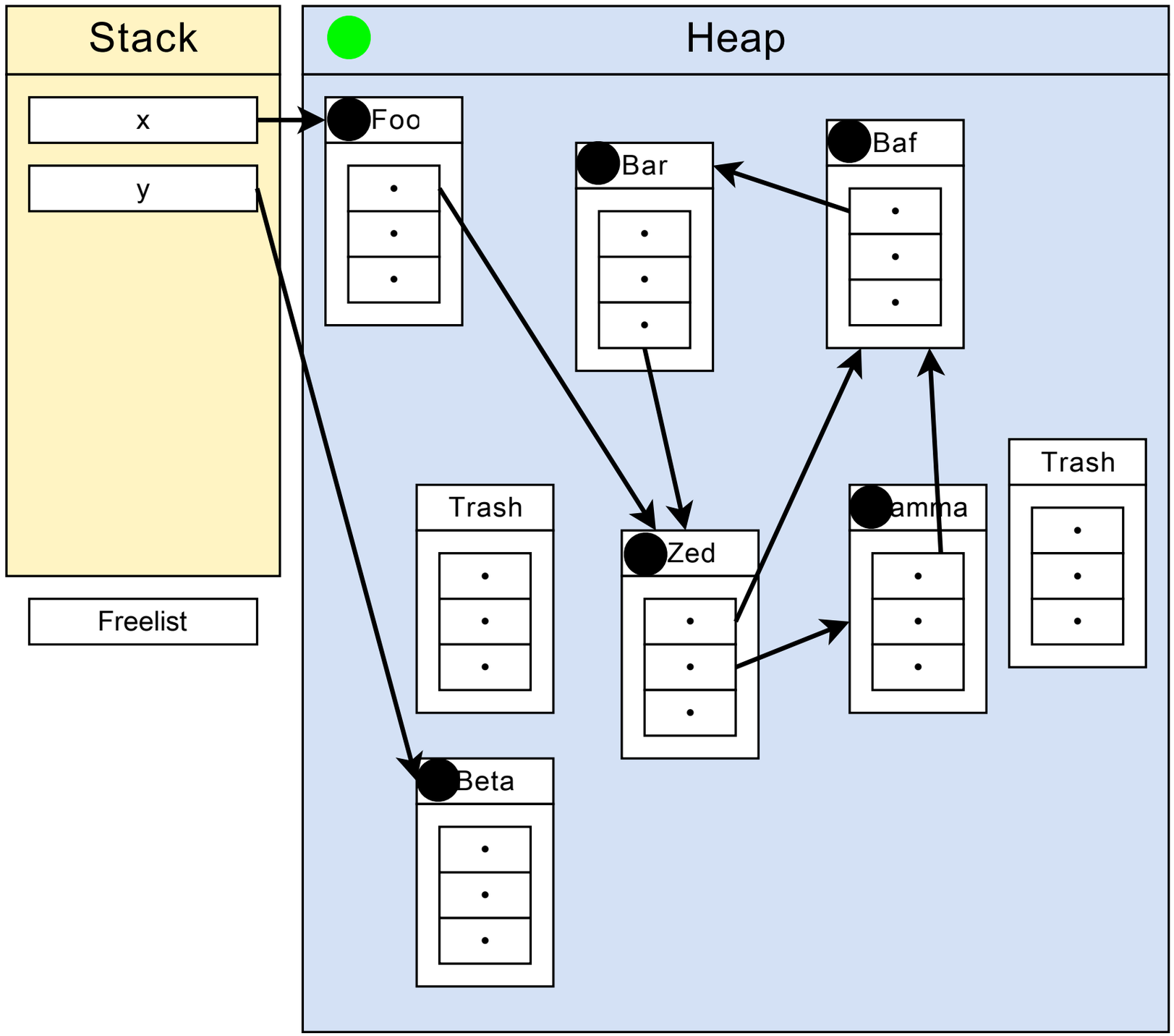
Mark



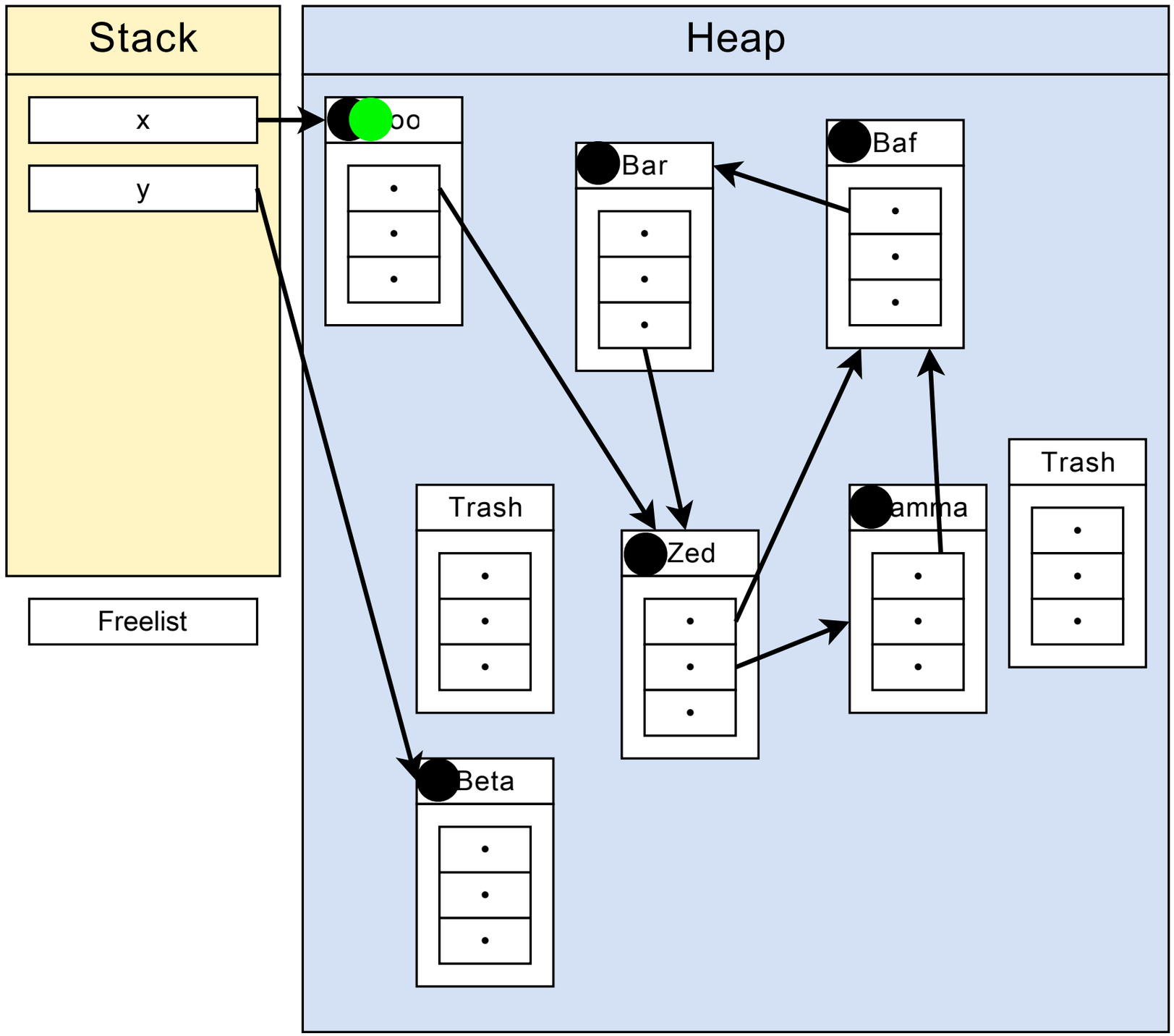
Mark



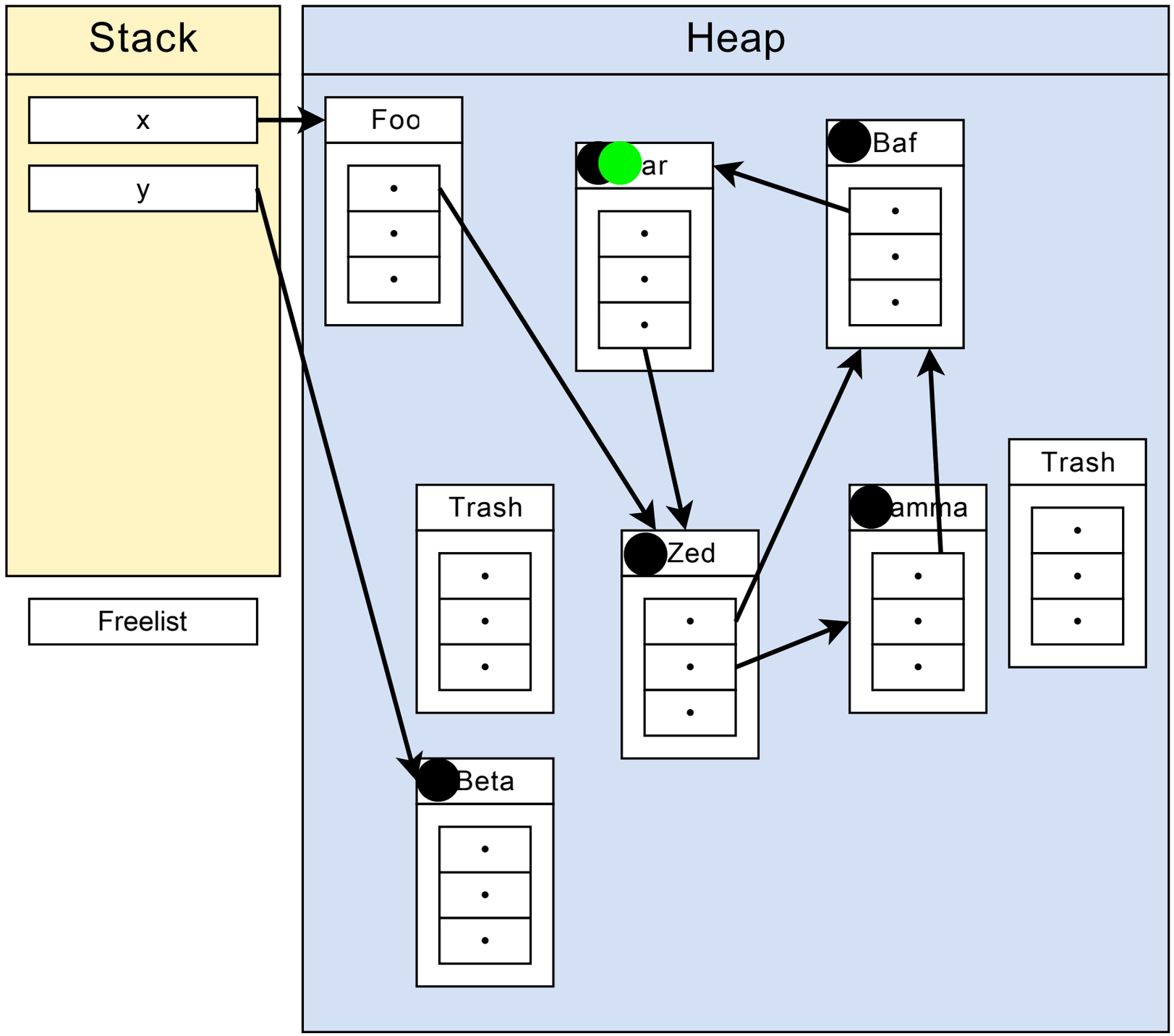
Sweep



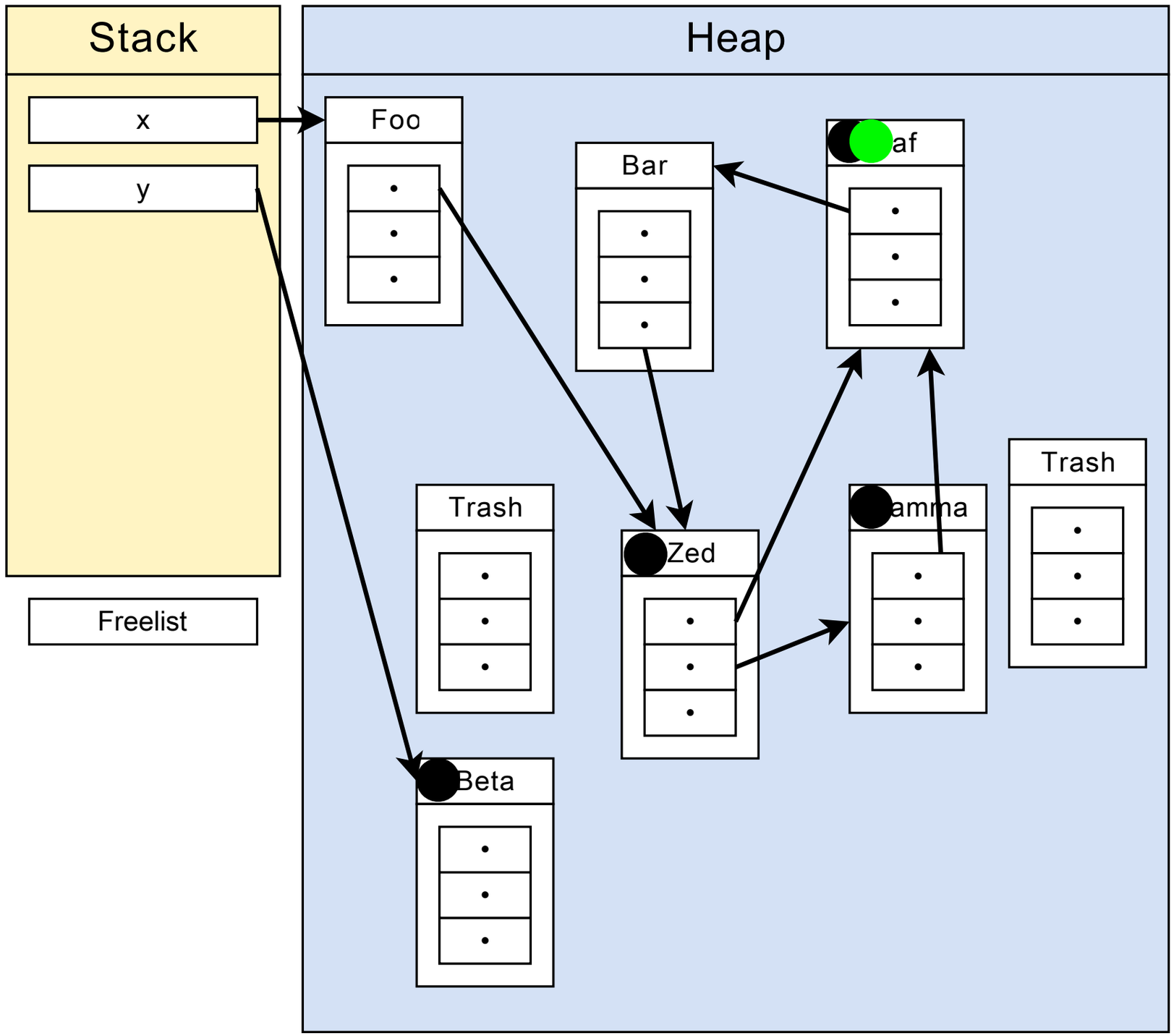
Sweep



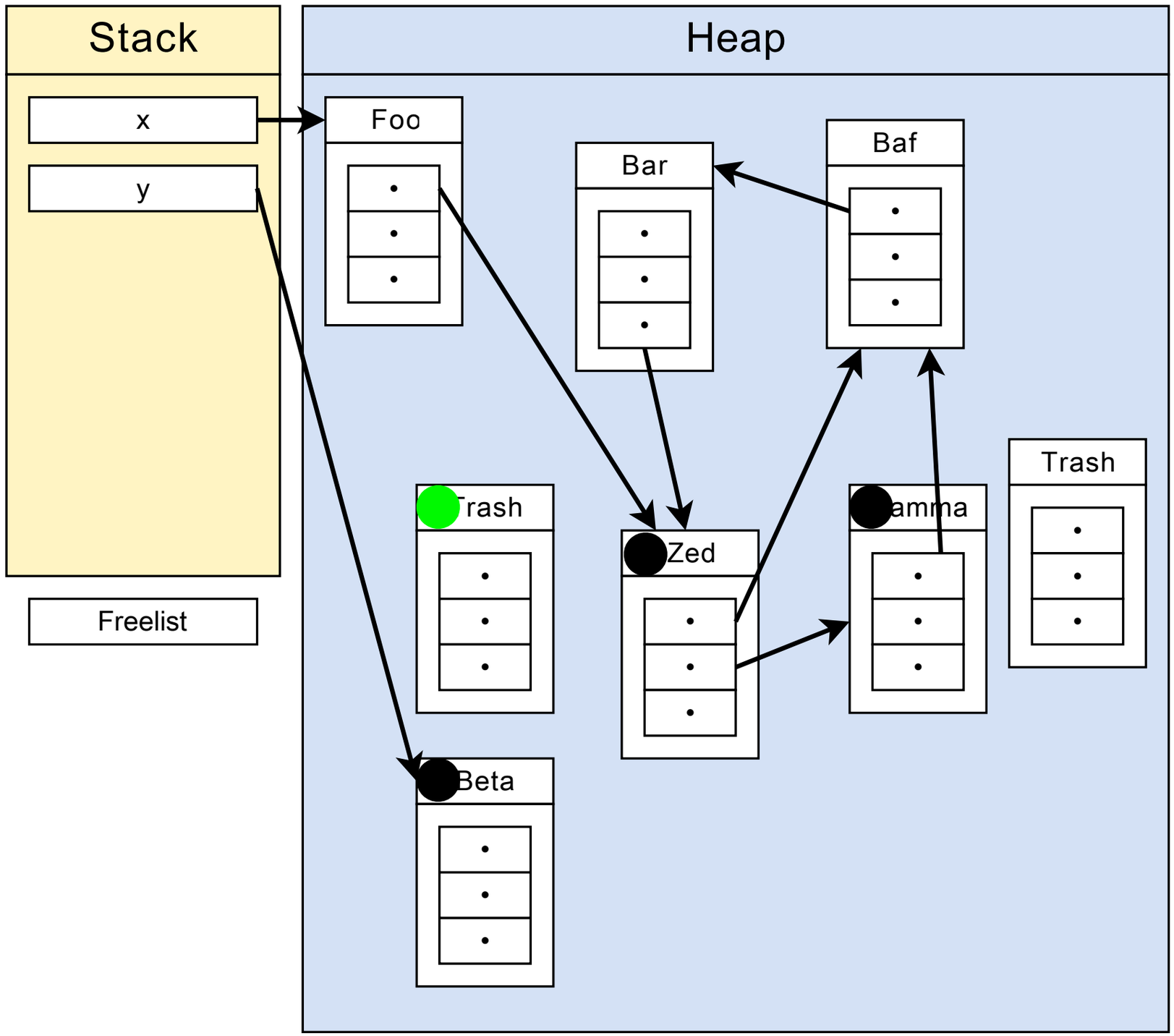
Sweep



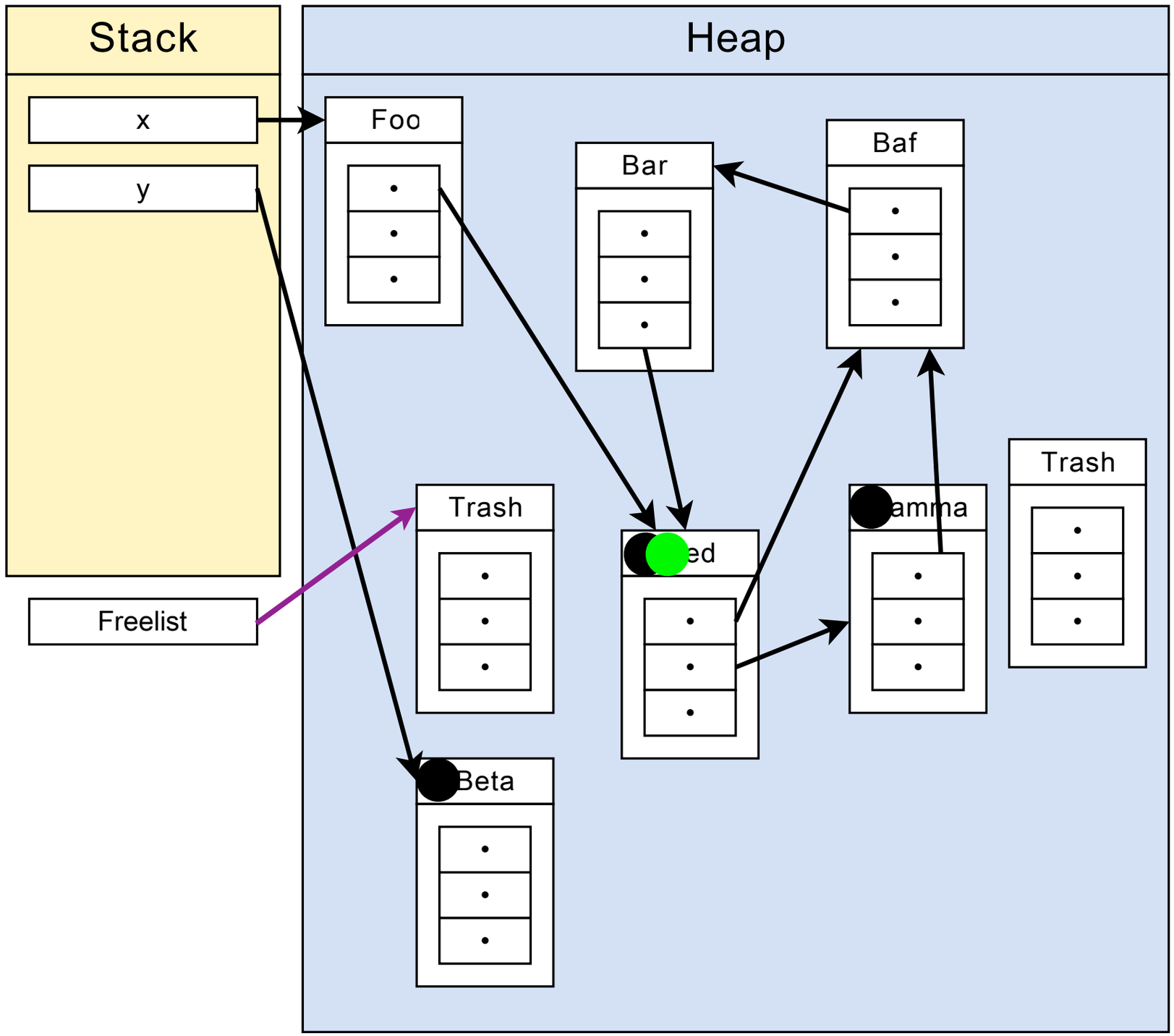
Sweep



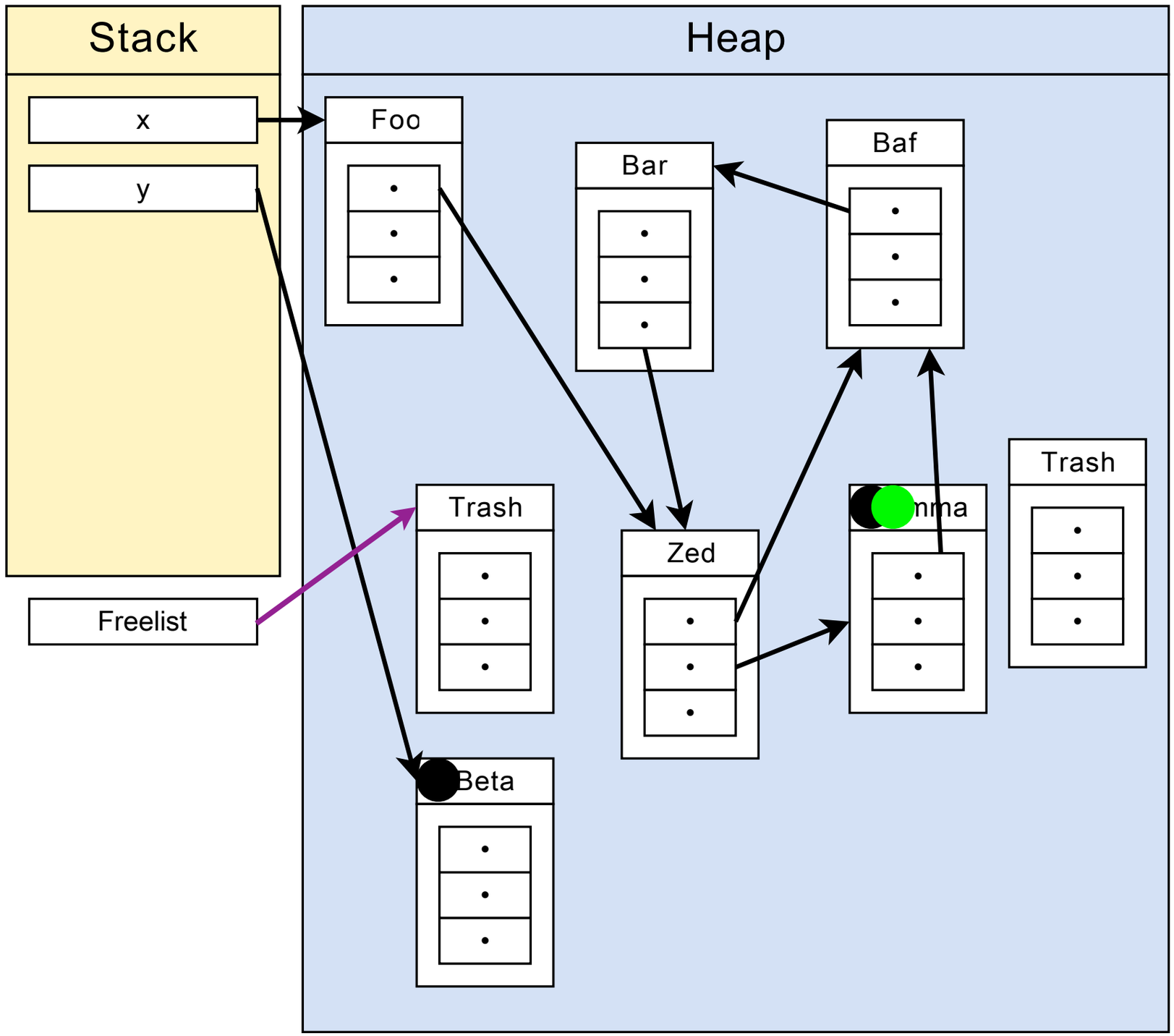
Sweep



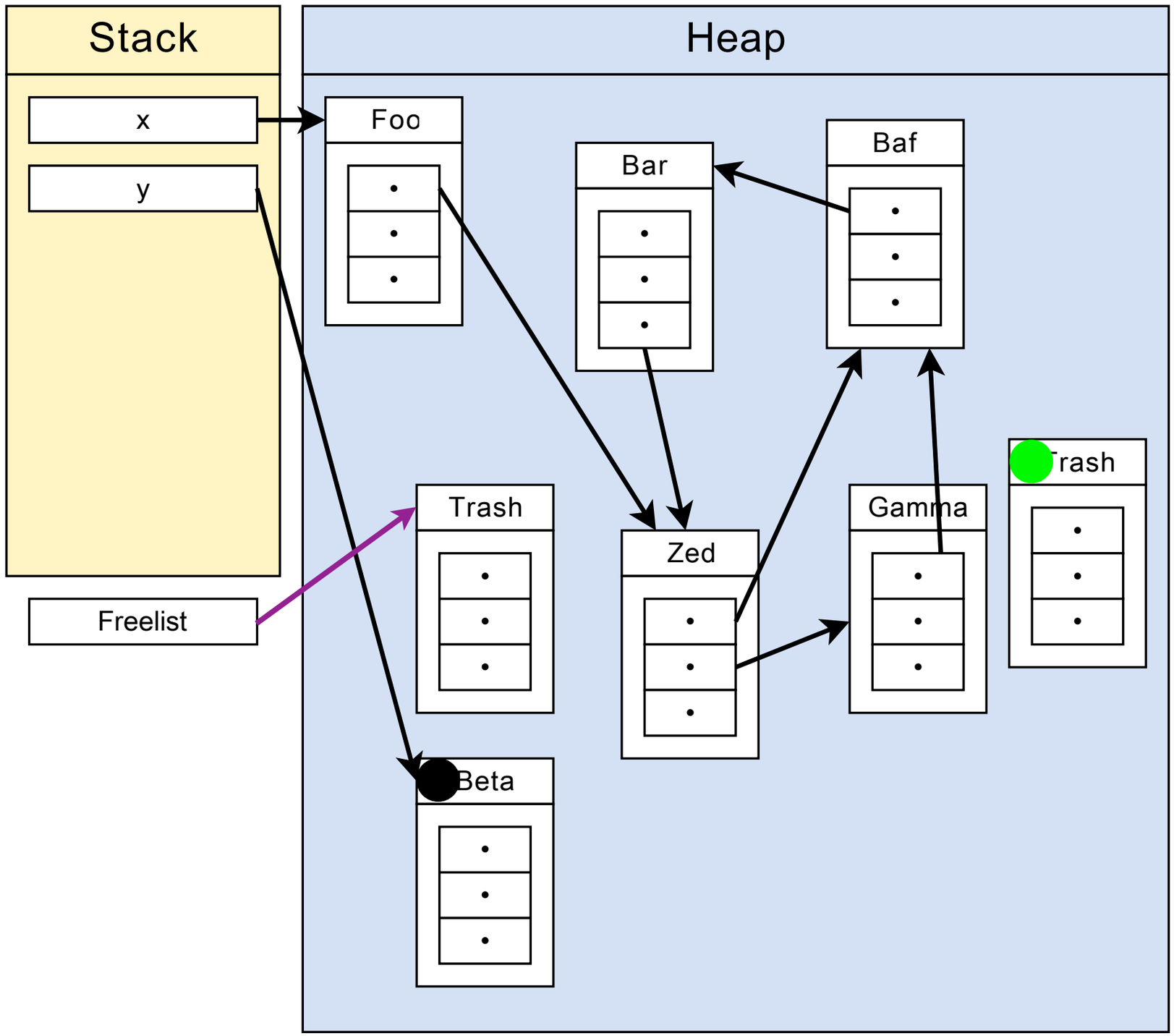
Sweep



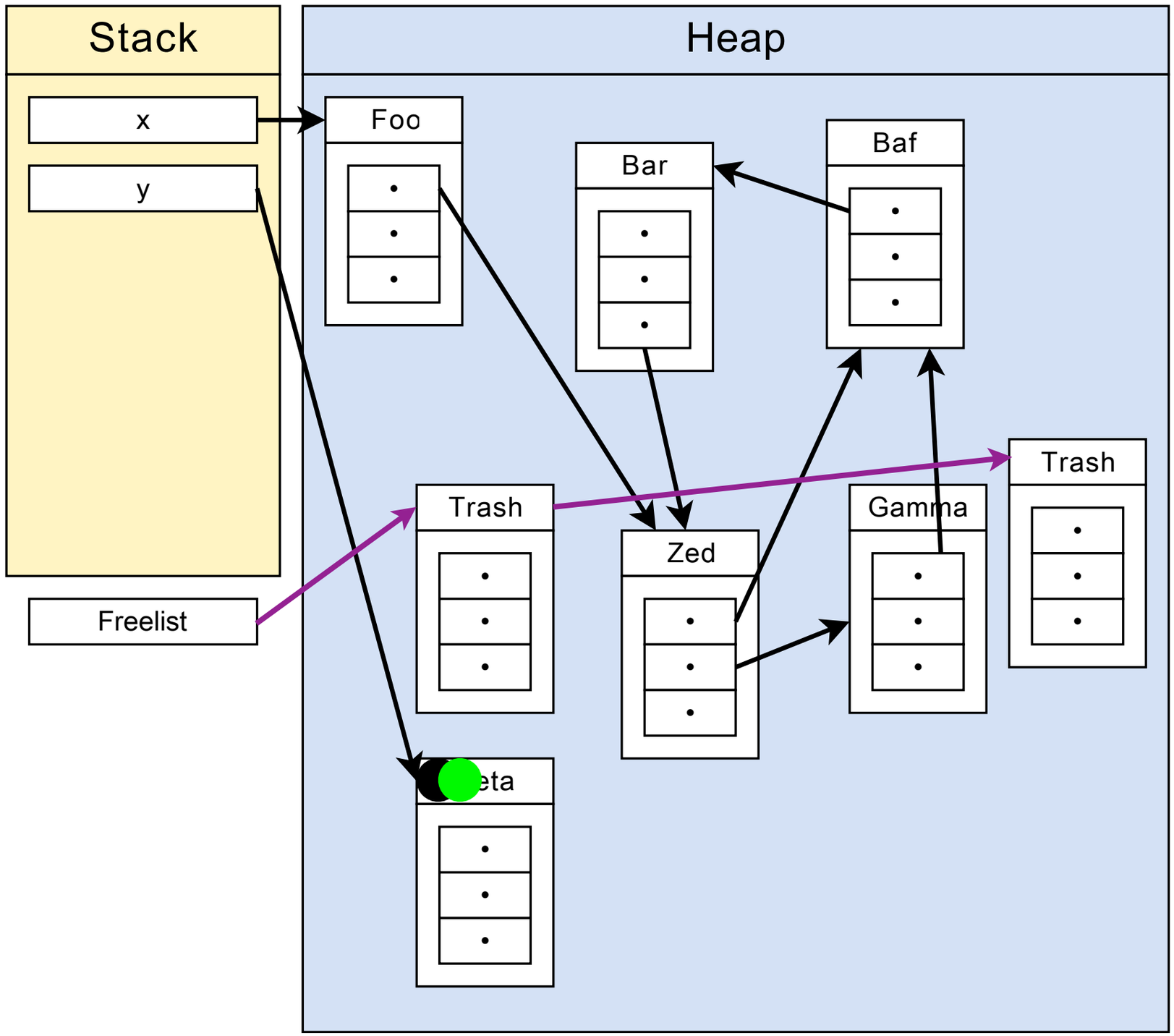
Sweep



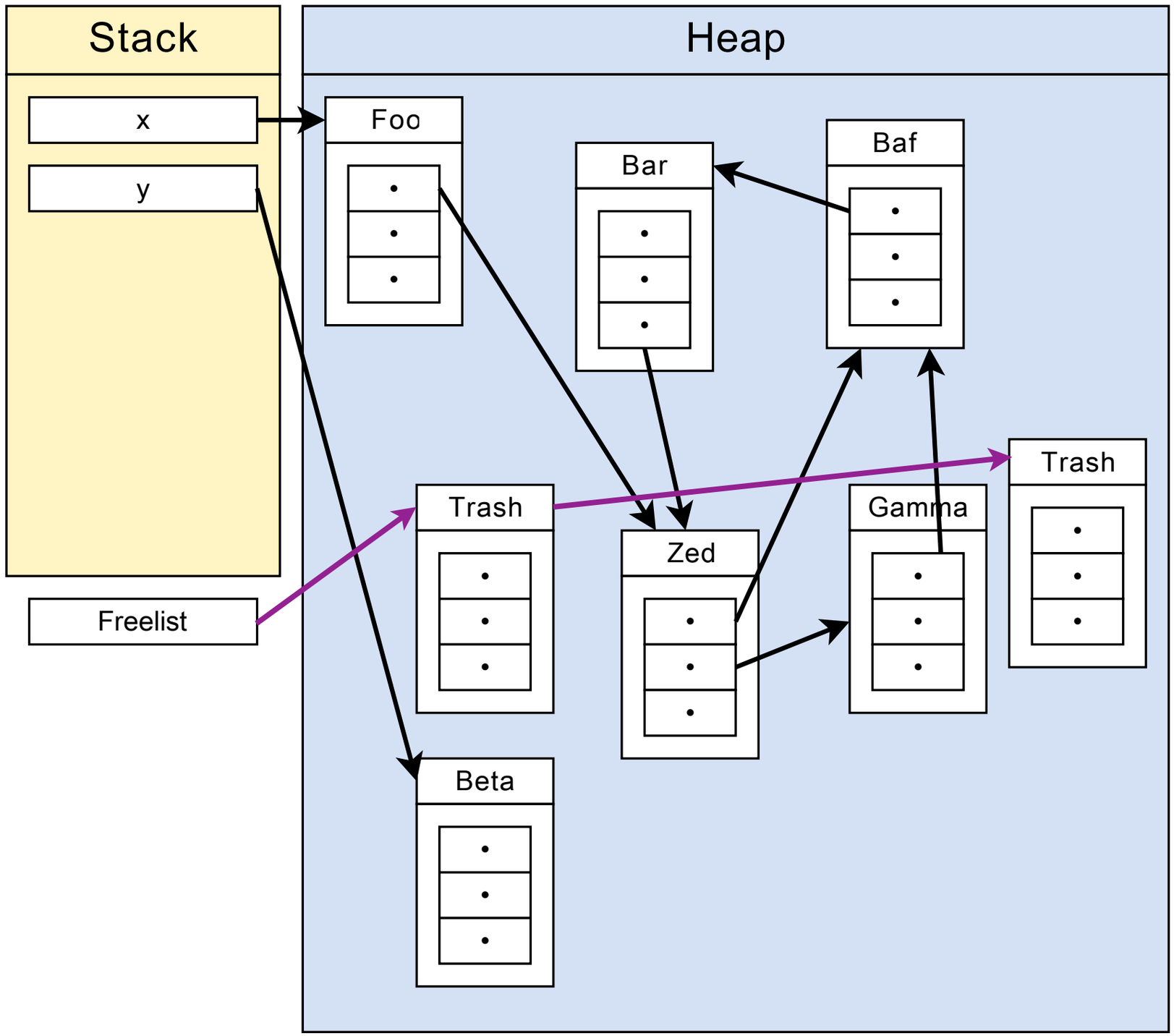
Sweep


Stack
Heap
x
y
Freelist
Foo
Bar
Baf
Trash
Zed
Gamma
Trash
Beta

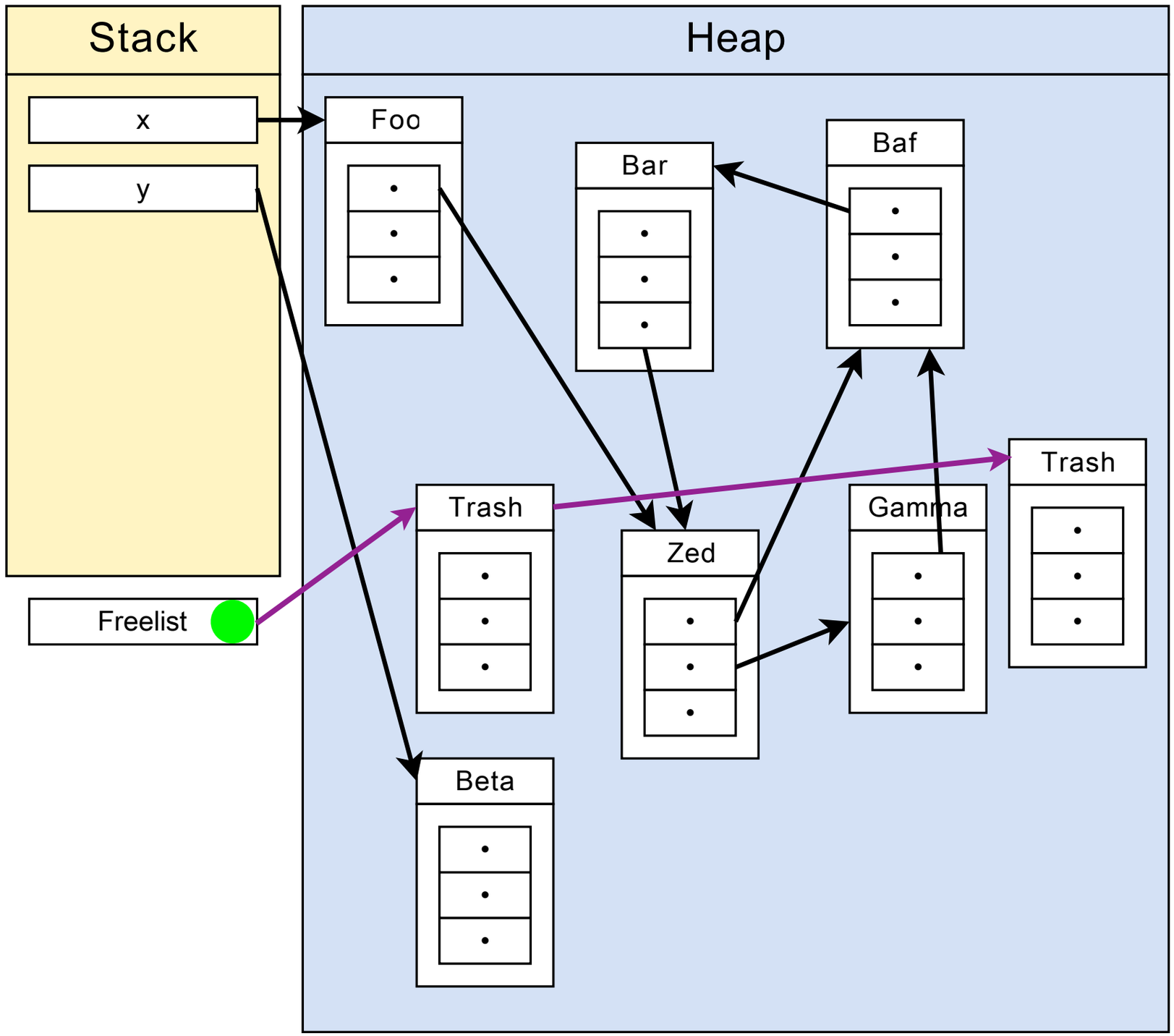
Sweep



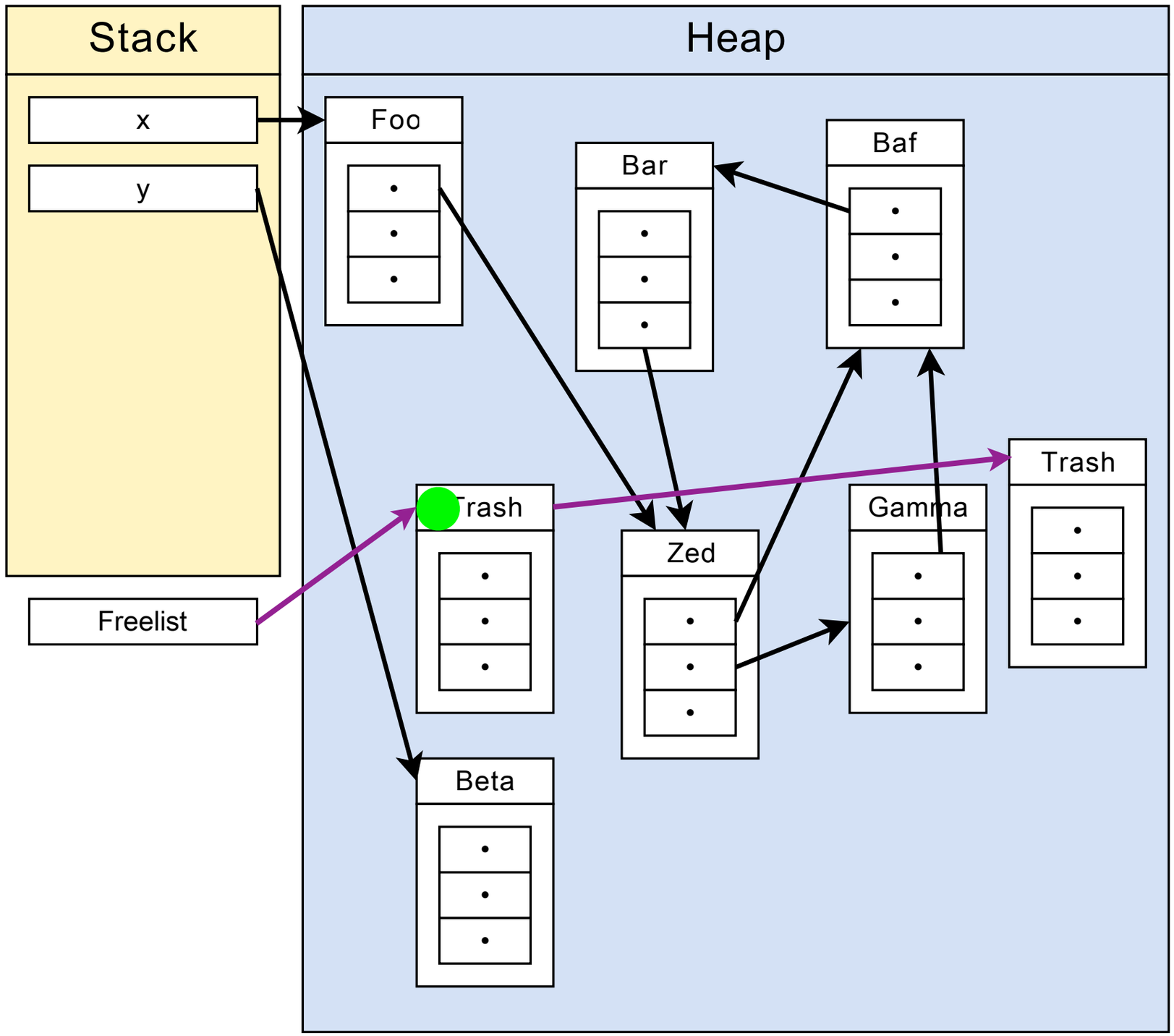
Sweep


Stack
Heap
x
y
Freelist
Foo
Bar
Baf
Trash
Zed
Gamma
Trash
Beta

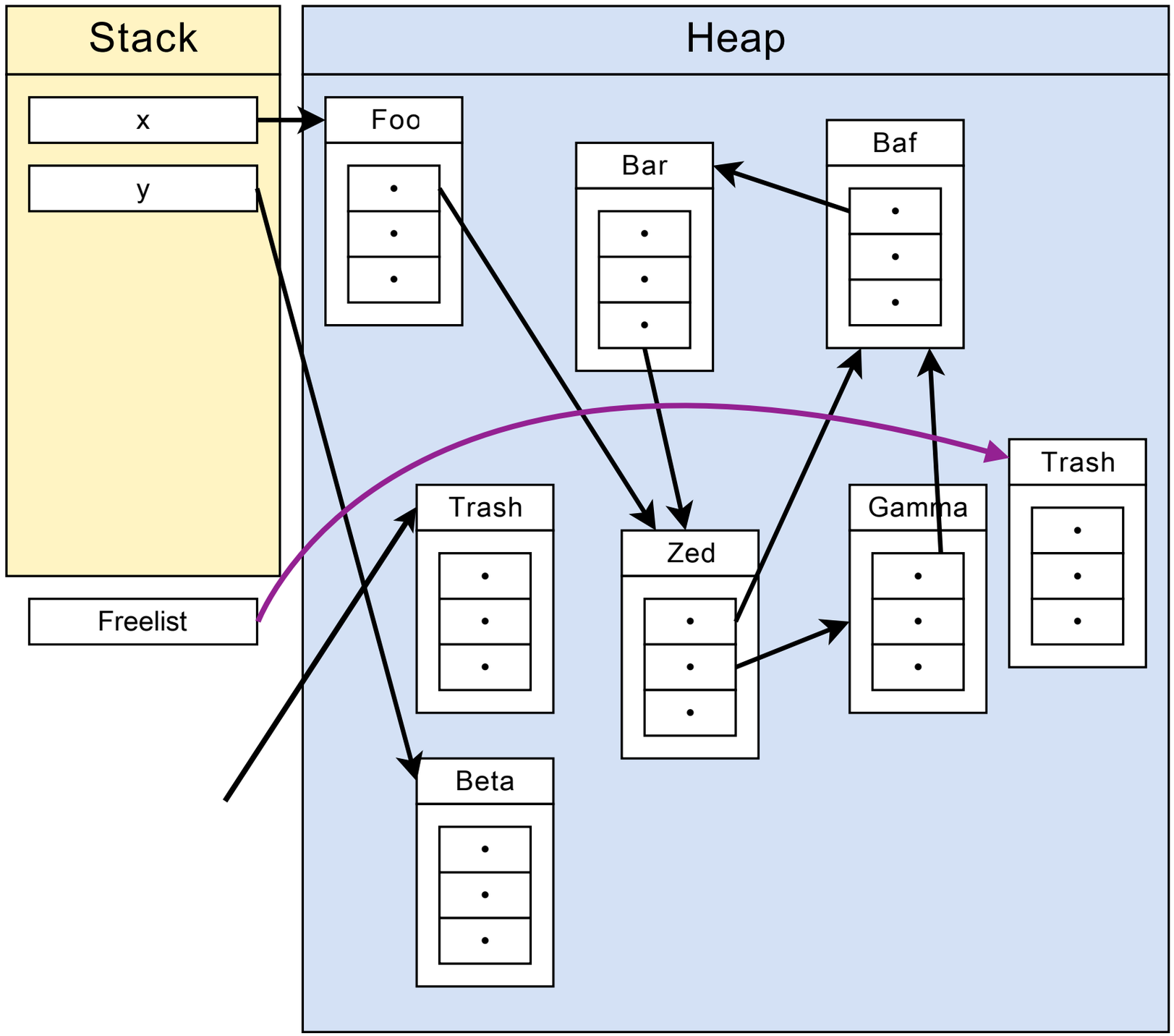
Allocation


Stack
Heap
x
y
Freelist
Foo
Bar
Baf
Trash
Zed
Gamma
Trash
Beta

Allocation


Stack
Heap
x
y
Freelist
Foo
Bar
Baf
Trash
Zed
Gamma
Trash
Beta

Allocation


Stack
Heap
x
y
Freelist
Foo
Bar
Baf
Trash
Zed
Gamma
Trash
Beta

Algorithms

Algorithms

Mark:

- Depth-first search of heap
- Mark reachable nodes

Algorithms

Mark:

- Depth-first search of heap
- Mark reachable nodes

Sweep:

- for each object in heap,
  - if marked, unmarked
  - else add to freelist

Complexity

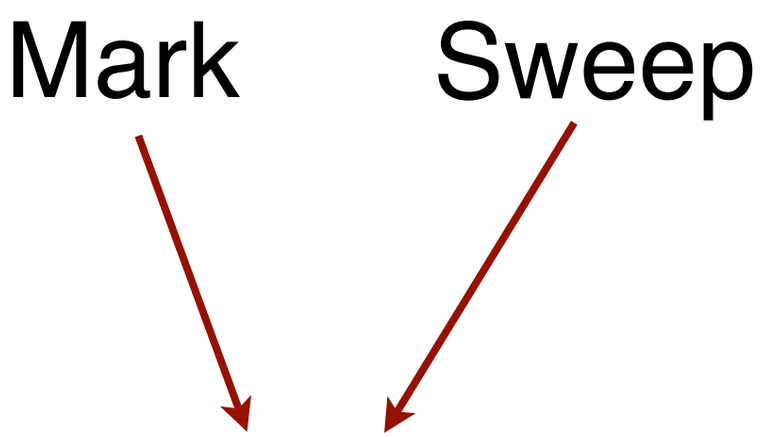
- O(R+H)
  - R: reachable objects
  - H: size of heap
- Amortized ~ (R+H)/(H-R)

Complexity

Mark

- O(R+H)
  - R: reachable objects
  - H: size of heap
- Amortized ~ (R+H)/(H-R)

Complexity



- O(R+H)
  - R: reachable objects
  - H: size of heap
- Amortized ~ (R+H)/(H-R)

Complexity

Mark Sweep

- O(R+H)
  - R: reachable objects
  - H: size of heap
- Amortized ~ (R+H)/(H-R)

GC frequency ≅ free space

Implementation

Implementation

How do we store the state of our depth-first search?

- Runtime stack: Too much recursion!
- Explicit stack: Allocated where?
- “Pointer reversal”: Use the objects we’re scanning as the stack!

Why M-and-S?

Why M-and-S?

Pros:

- Doesn't move objects
- Straightforward

Why M-and-S?

Pros:

- Doesn't move objects
- Straightforward

Cons:

- Memory fragmentation
- Can be costly

Advanced

Advanced

- Rarely have one freelist, use an array of object sizes

Advanced

- Rarely have one freelist, use an array of object sizes
- Sweep can be O(1) by keeping objects on an alloclist

Advanced

- Rarely have one freelist, use an array of object sizes
- Sweep can be O(1) by keeping objects on an alloclist
- Objects don’t move

Advanced

- Rarely have one freelist, use an array of object sizes
- Sweep can be O(1) by keeping objects on an alloclist
- Objects don’t move
  - Interior pointers

Advanced

- Rarely have one freelist, use an array of object sizes
- Sweep can be O(1) by keeping objects on an alloclist
- Objects don’t move
  - Interior pointers
  - Conservative collection

Two-Space Copying

Photo source: http://www.flickr.com/photos/photophonic/2678678123/ CC-By-NC-SA

Two-Space Copying

Two-Space Copying

- Keep two separate heaps

Two-Space Copying

- Keep two separate heaps
  - “From” space and “to” space

Two-Space Copying

- Keep two separate heaps
  - “From” space and “to” space
- Only allocate on one heap at a time

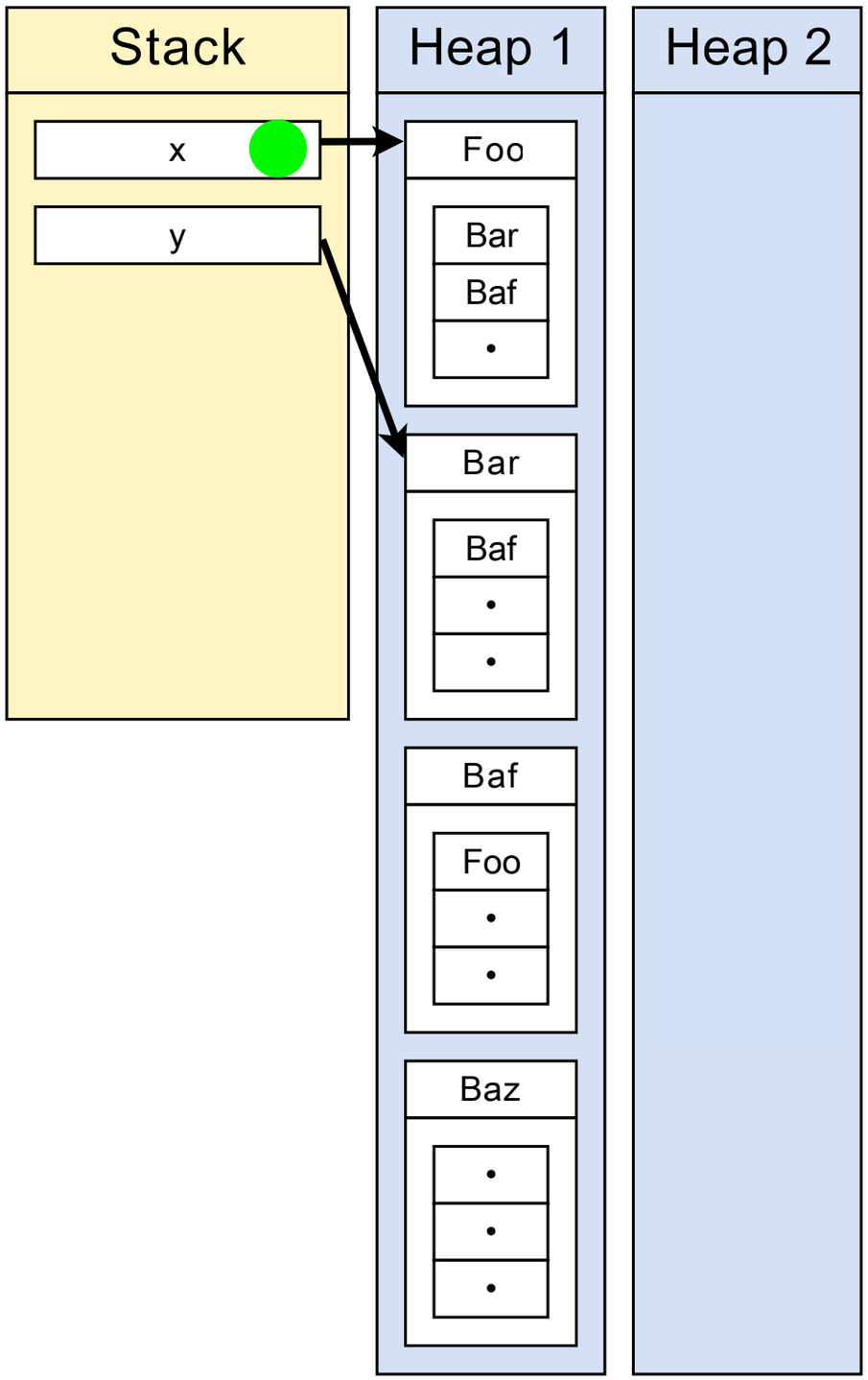
Two-Space Copying

- Keep two separate heaps
  - “From” space and “to” space
- Only allocate on one heap at a time
- Instead of marking, copy to other heap

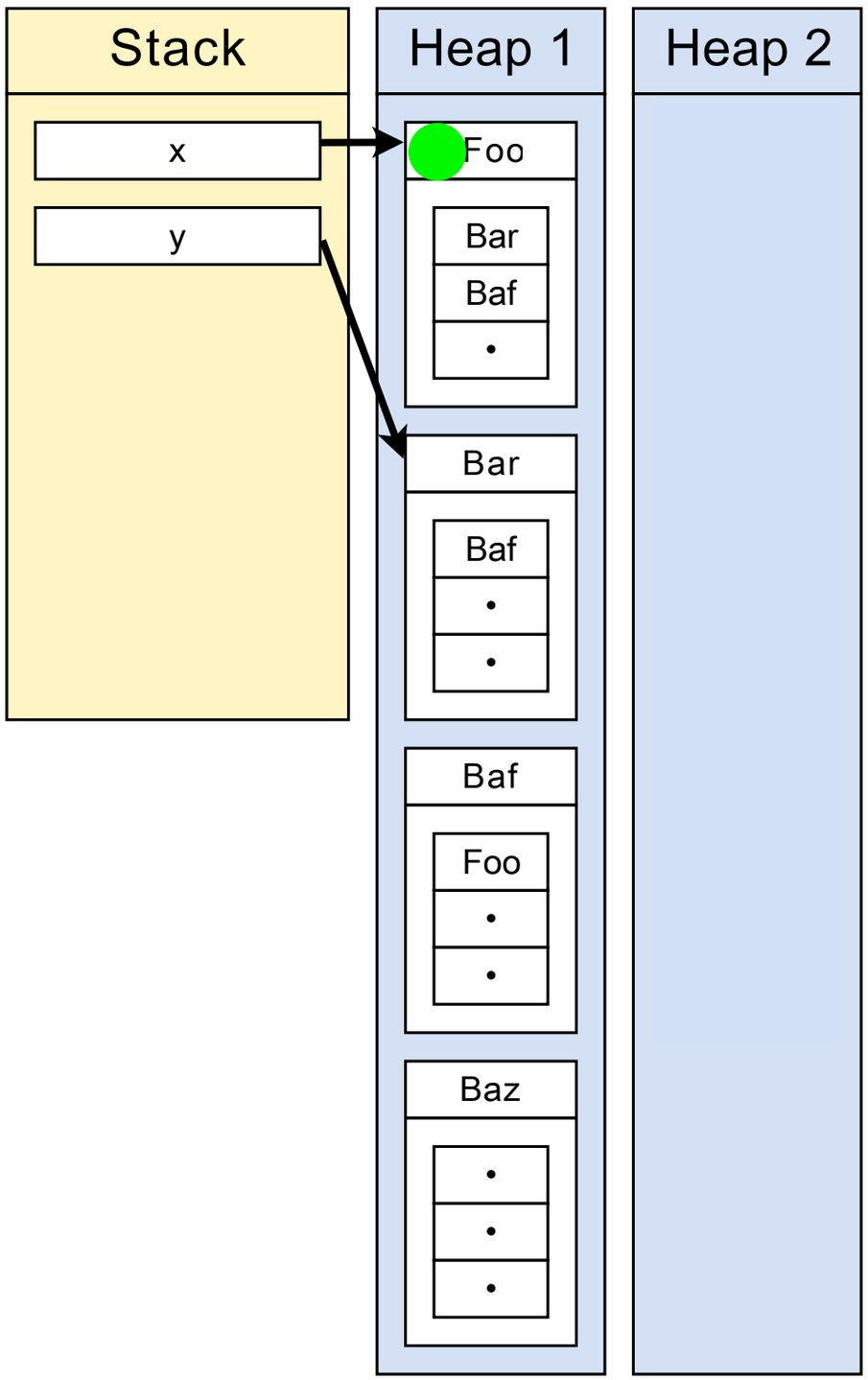
Two-Space Copying

- Keep two separate heaps
  - “From” space and “to” space
- Only allocate on one heap at a time
- Instead of marking, copy to other heap
- Redirect all pointers to new heap

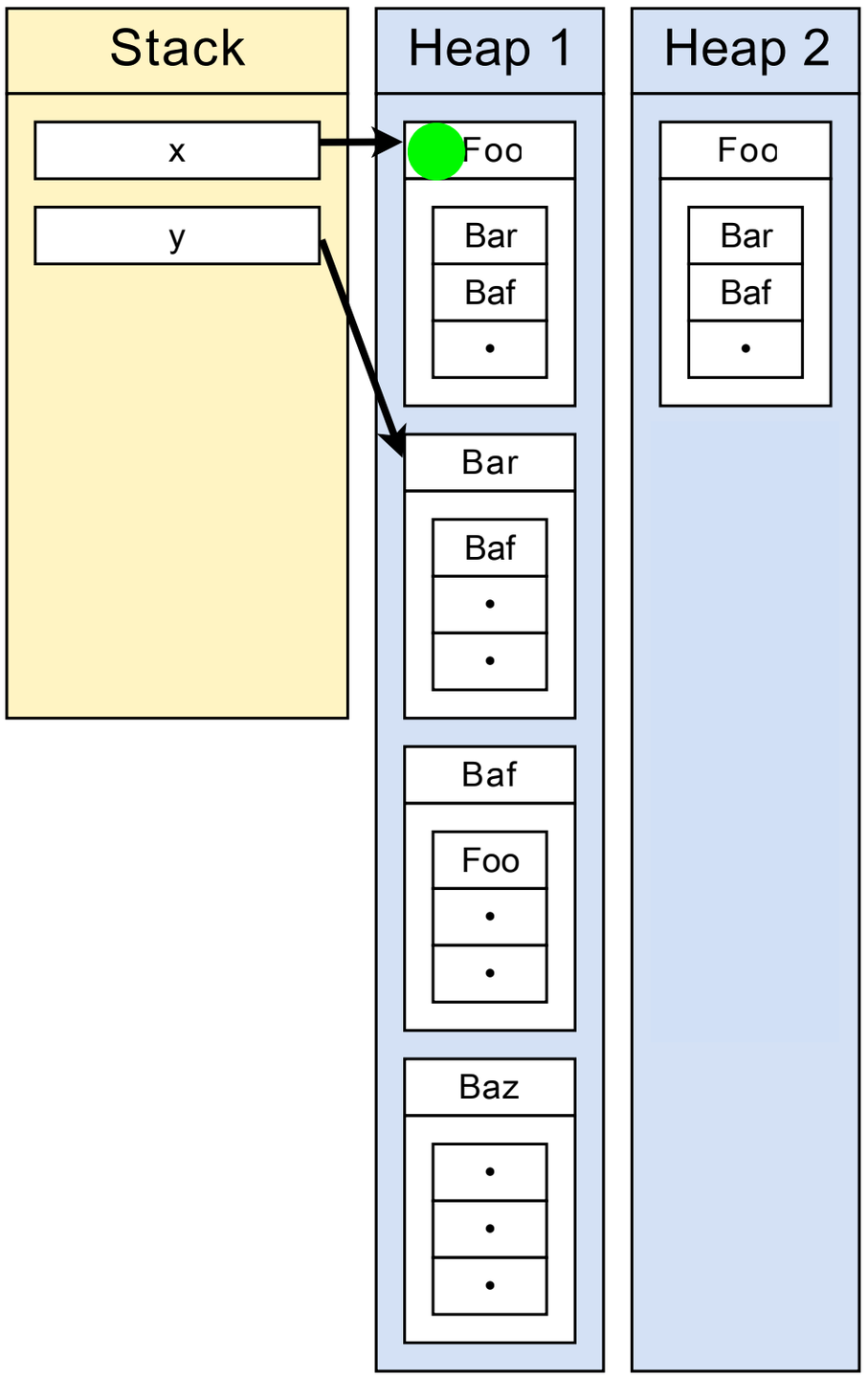
Two-Space Copying



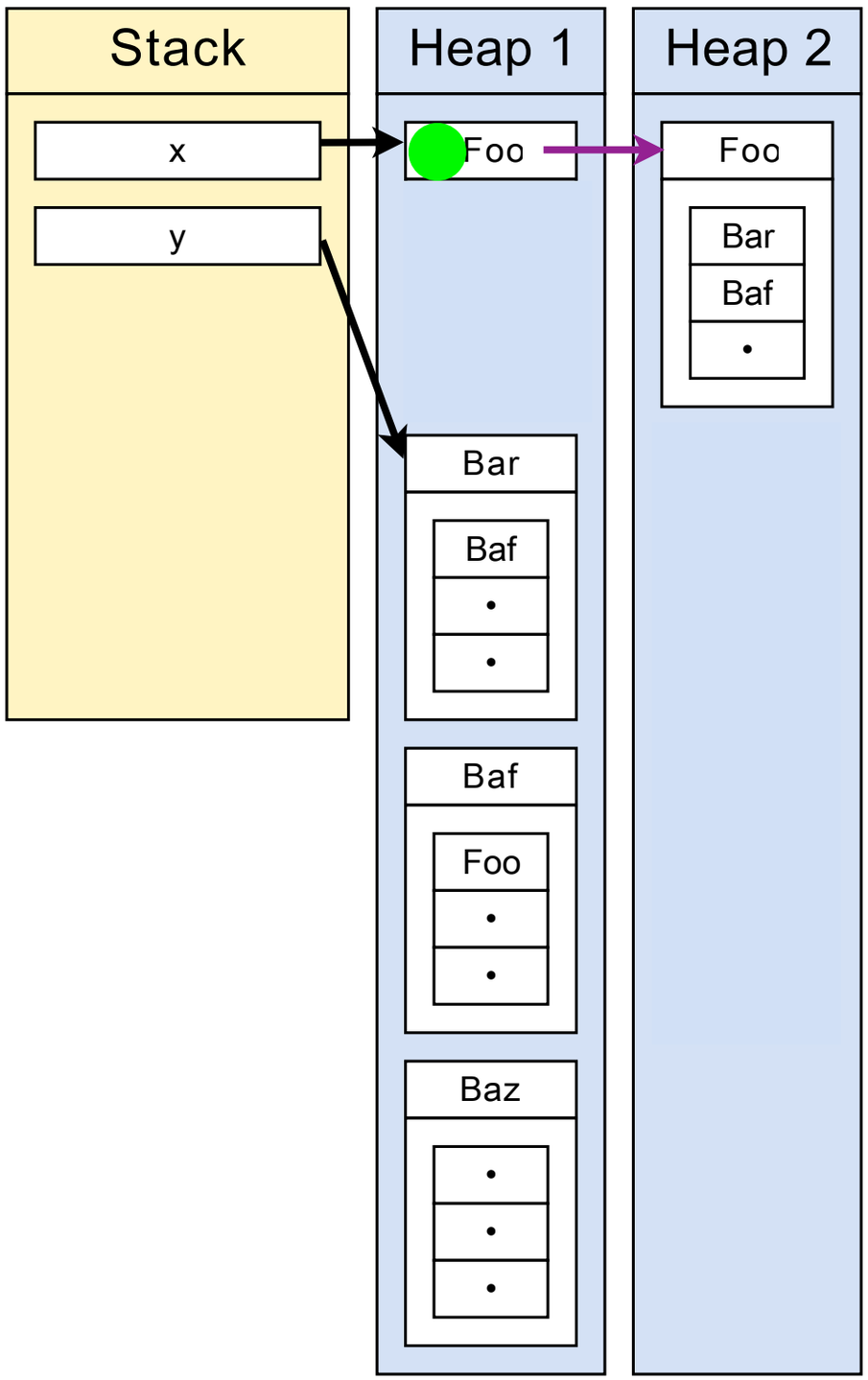
Two-Space Copying



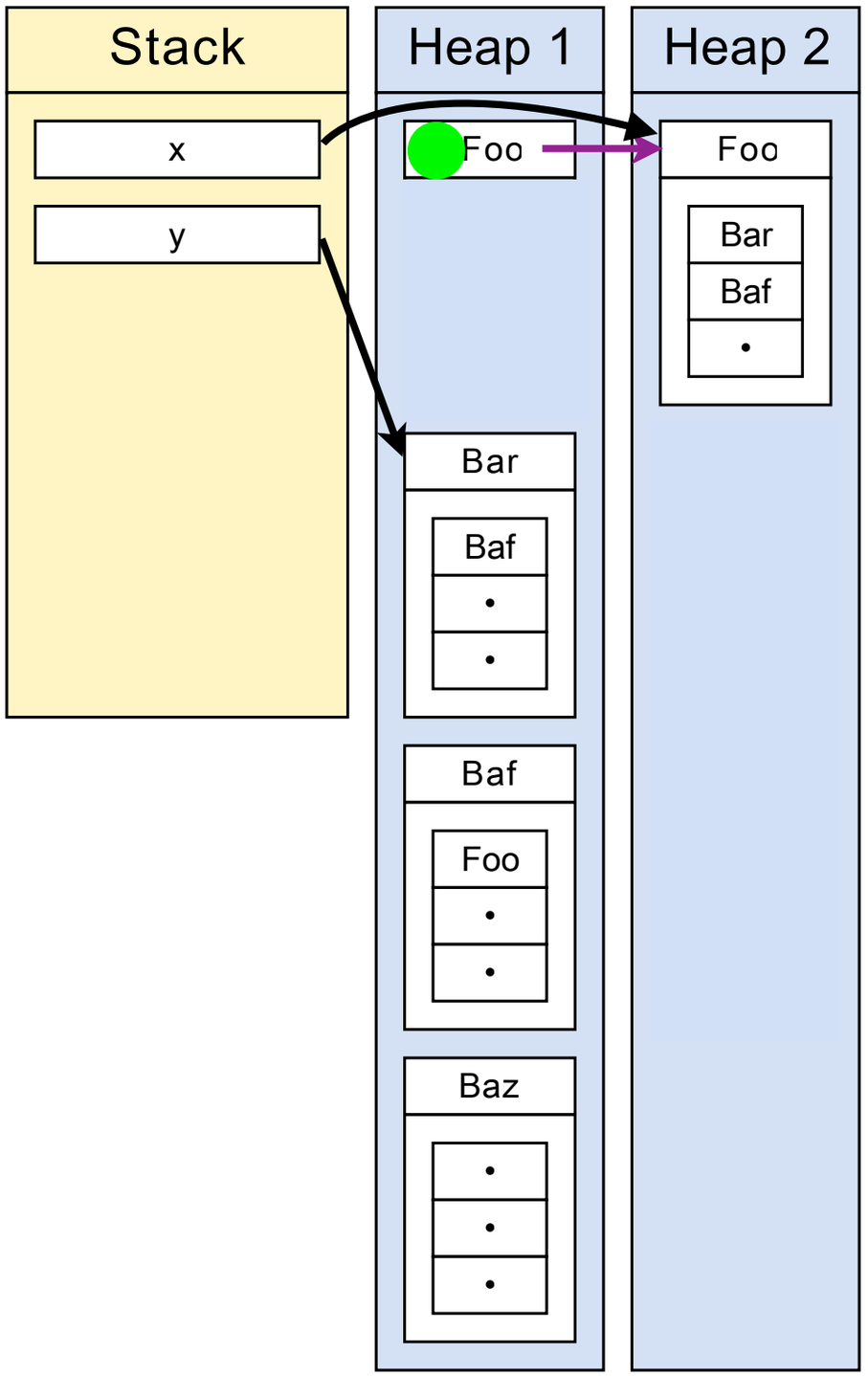
Two-Space Copying



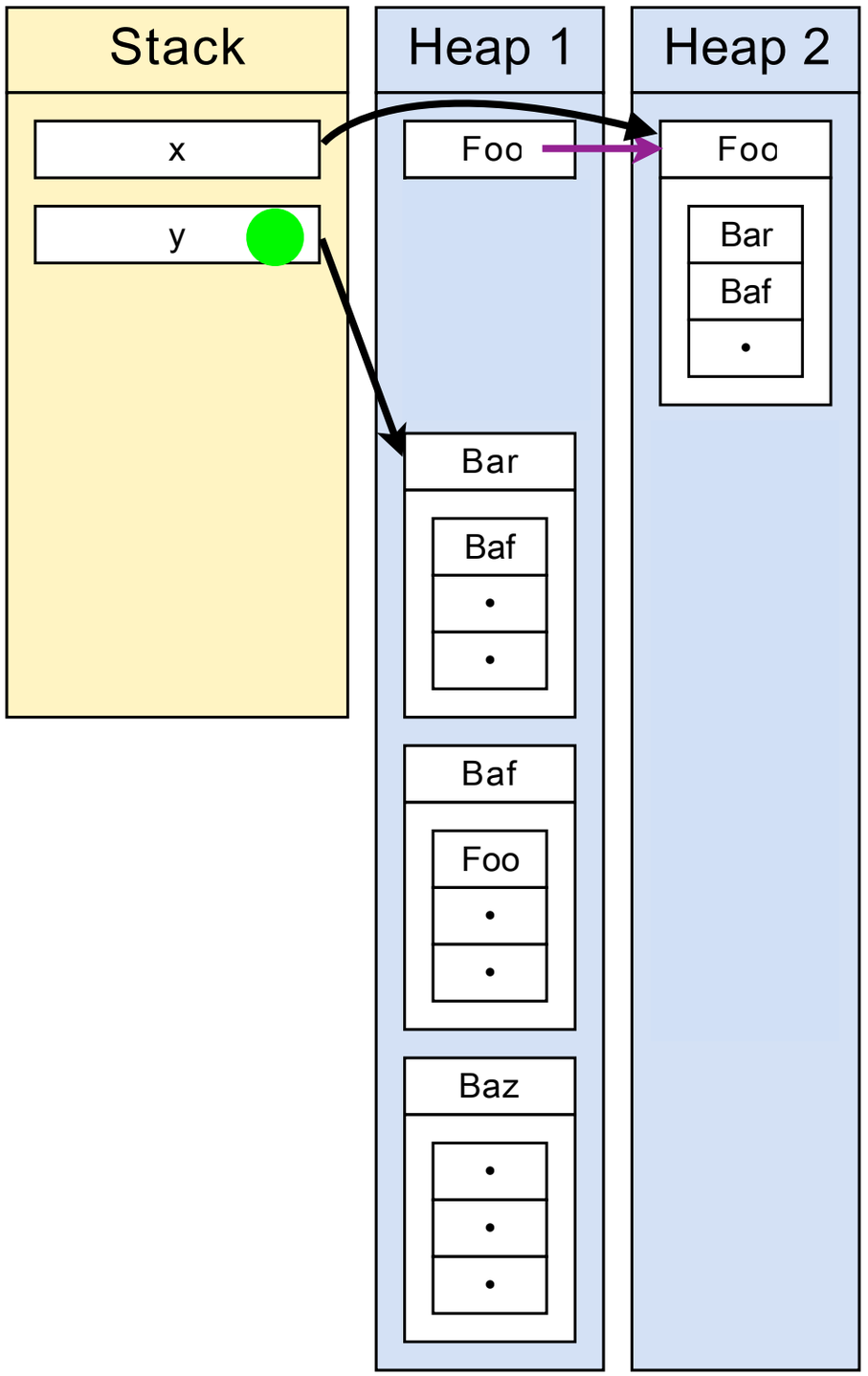
Two-Space Copying



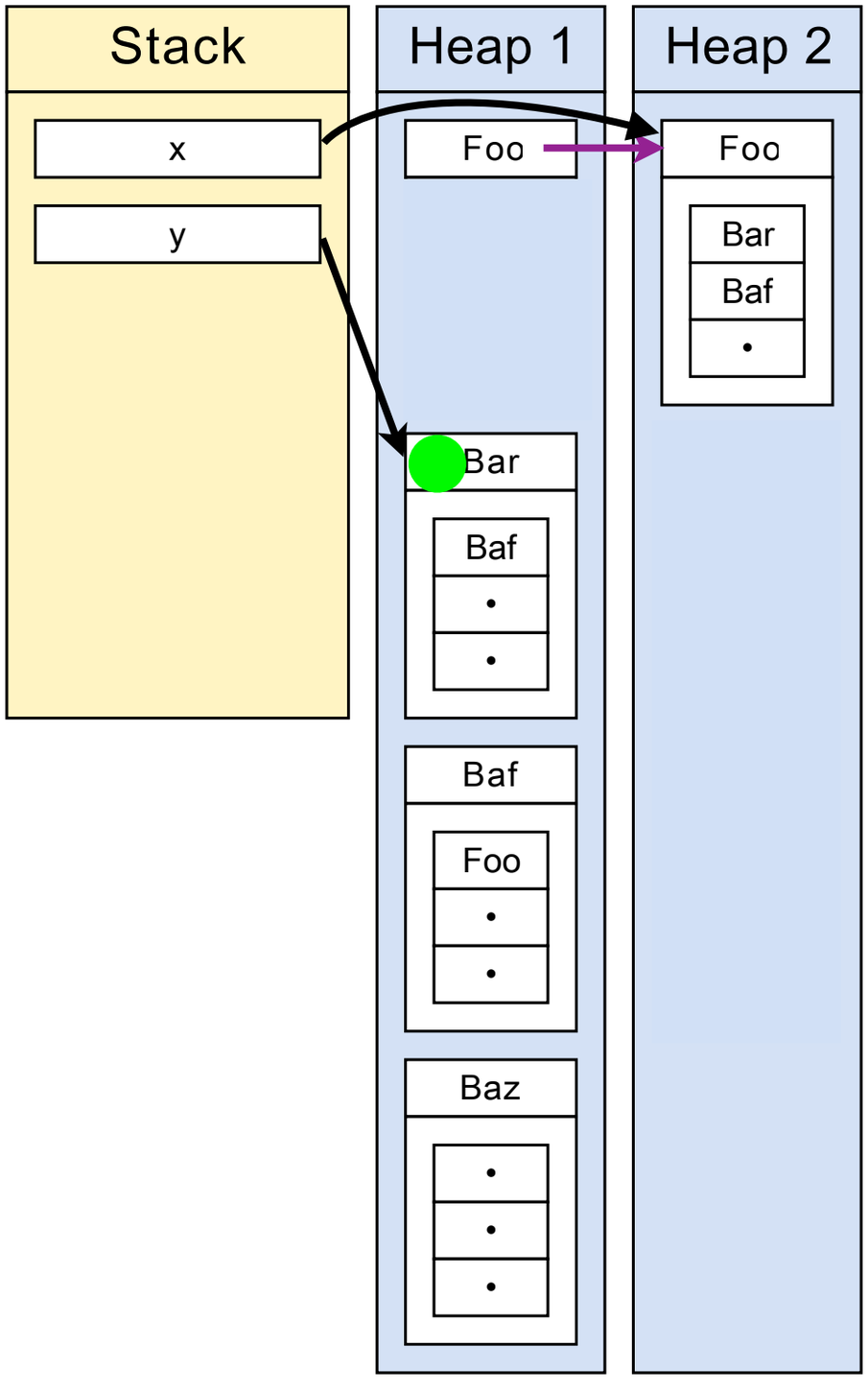
Two-Space Copying



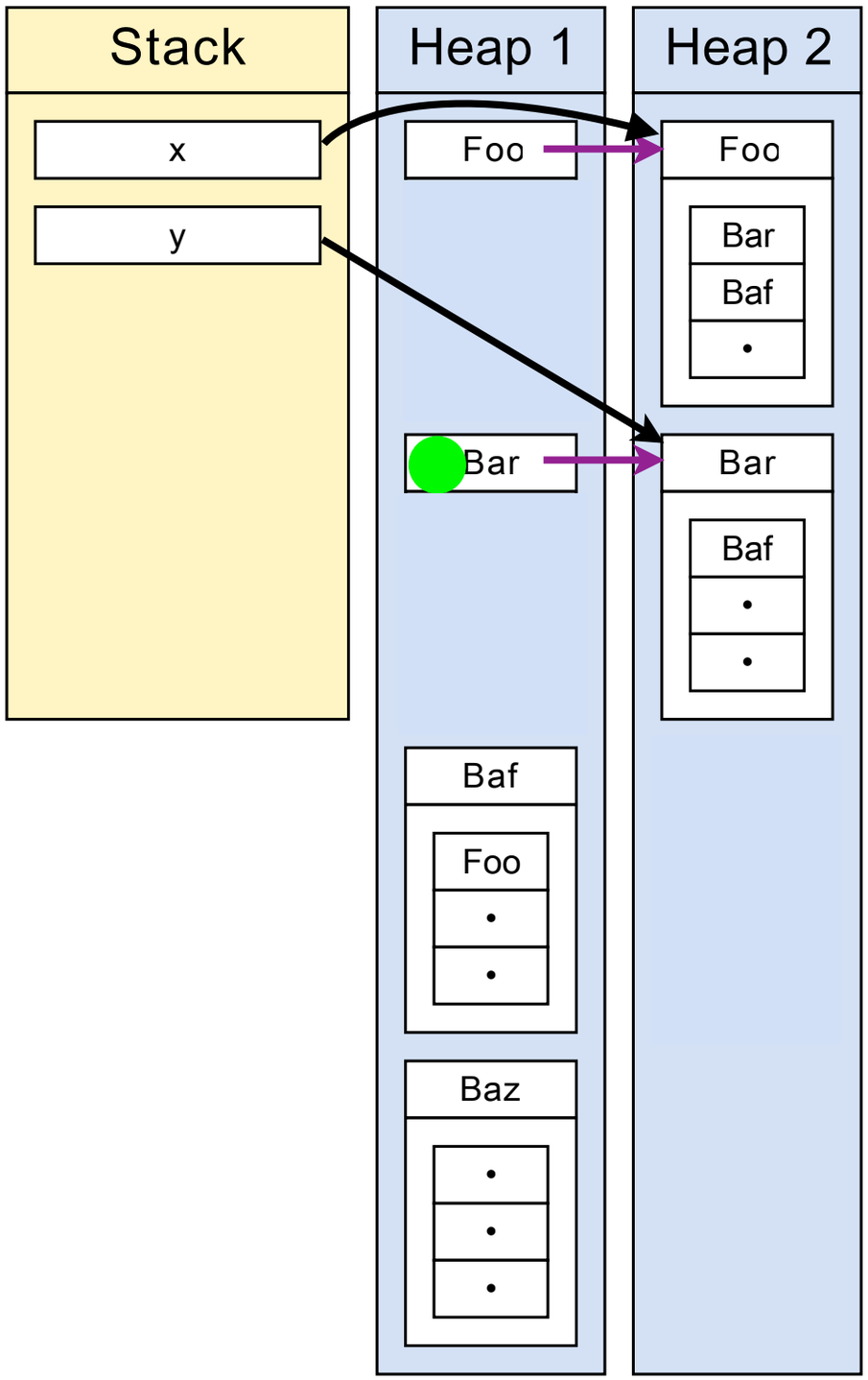
Two-Space Copying



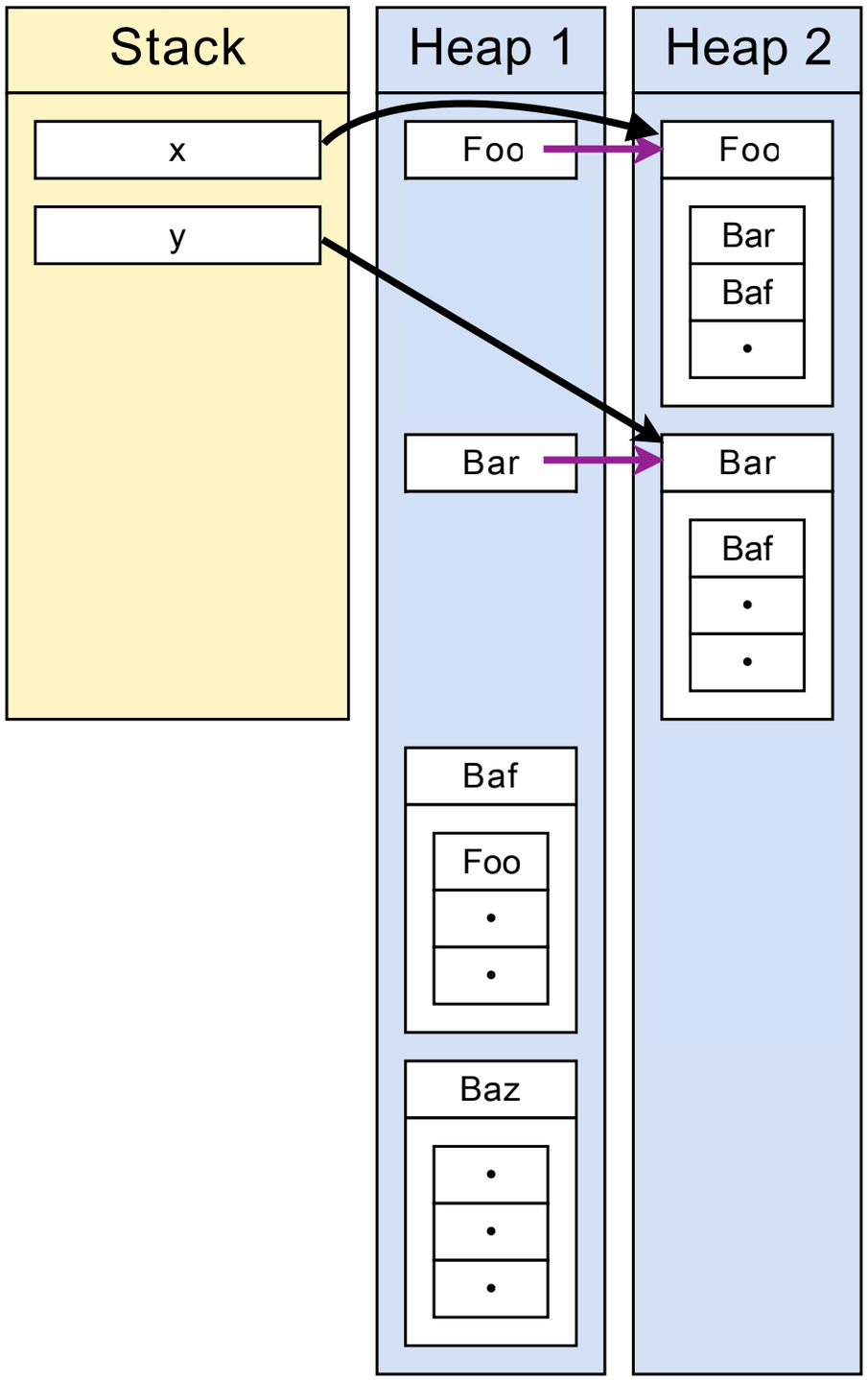
Two-Space Copying



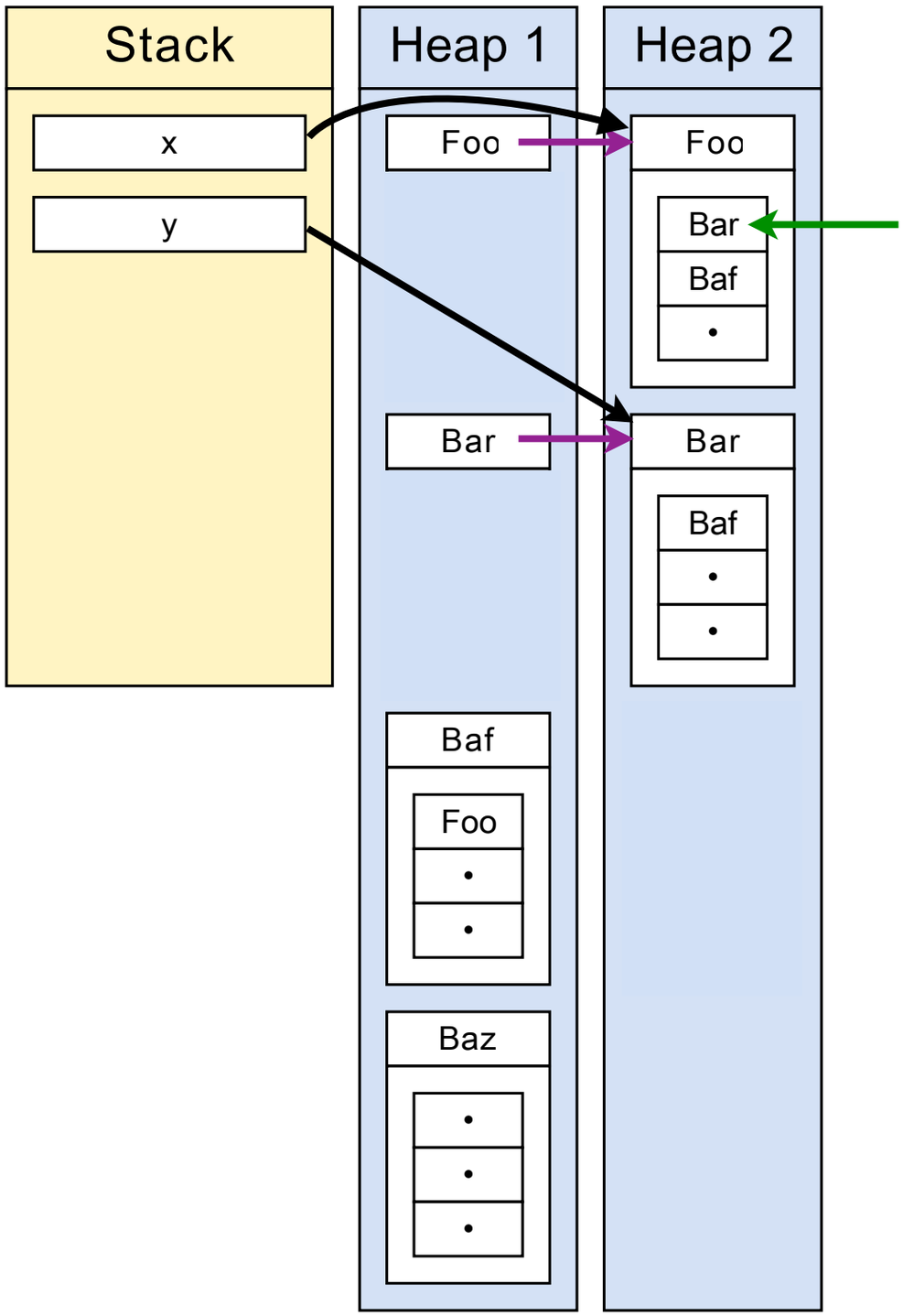
Two-Space Copying



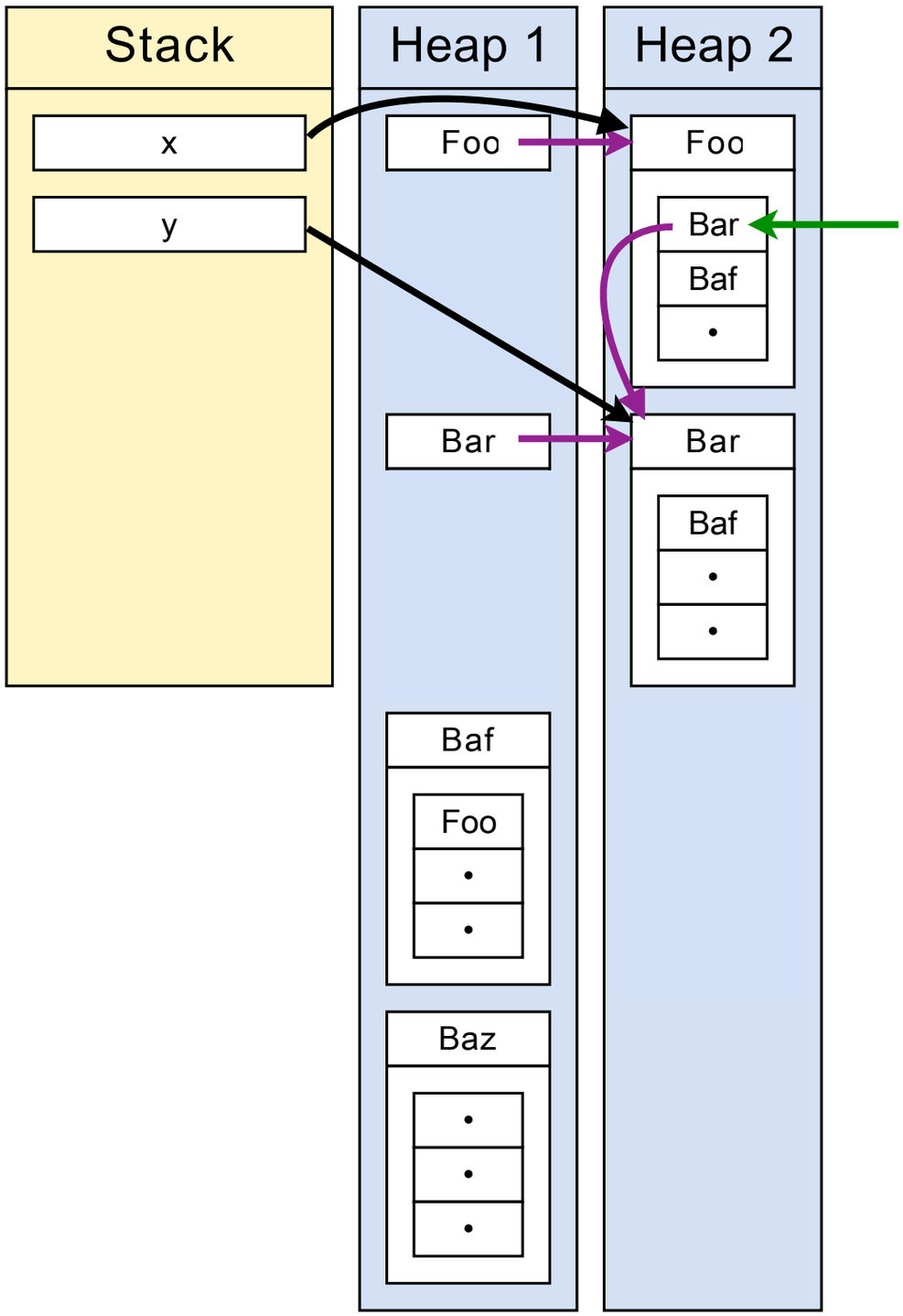
Two-Space Copying



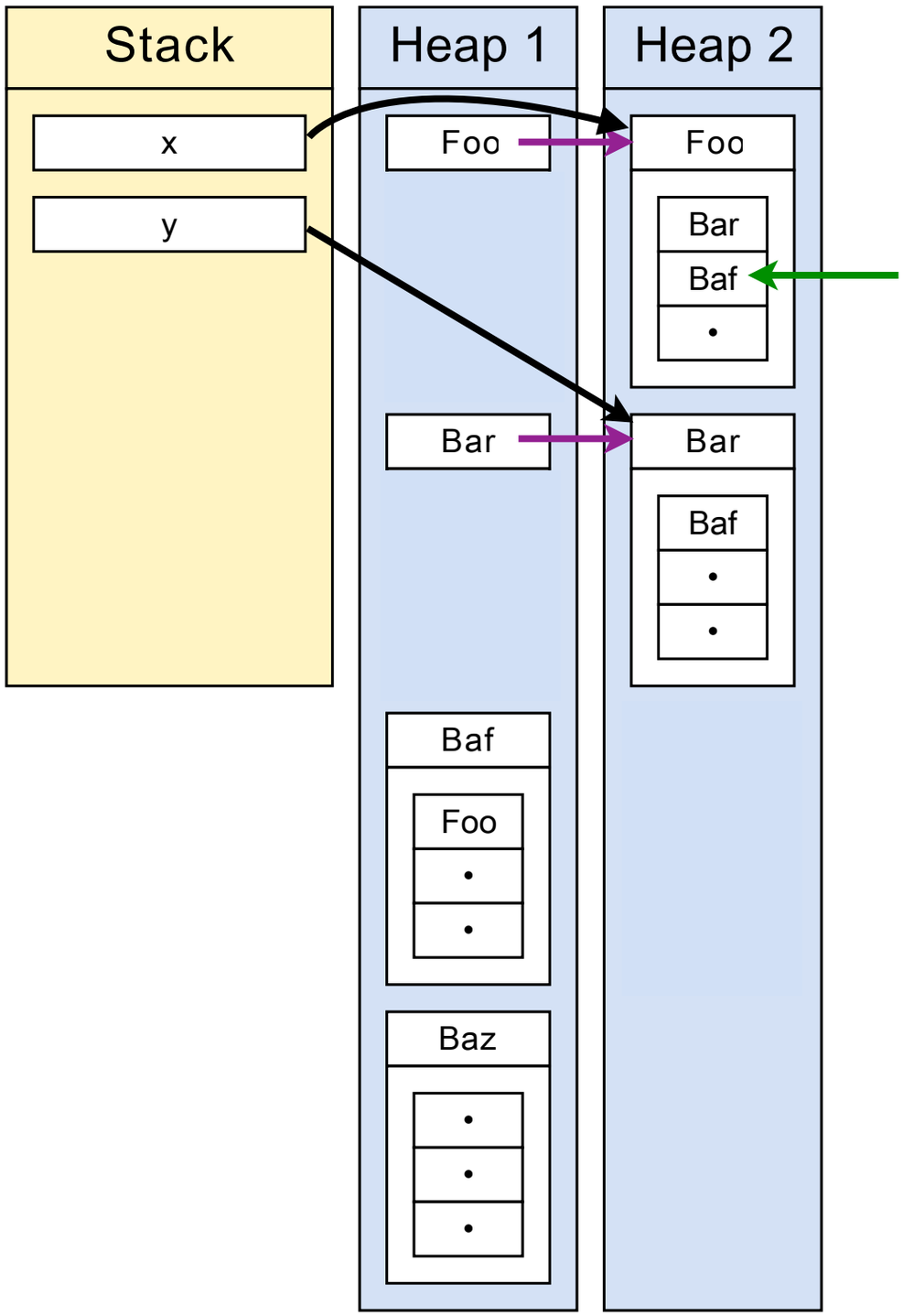
Two-Space Copying



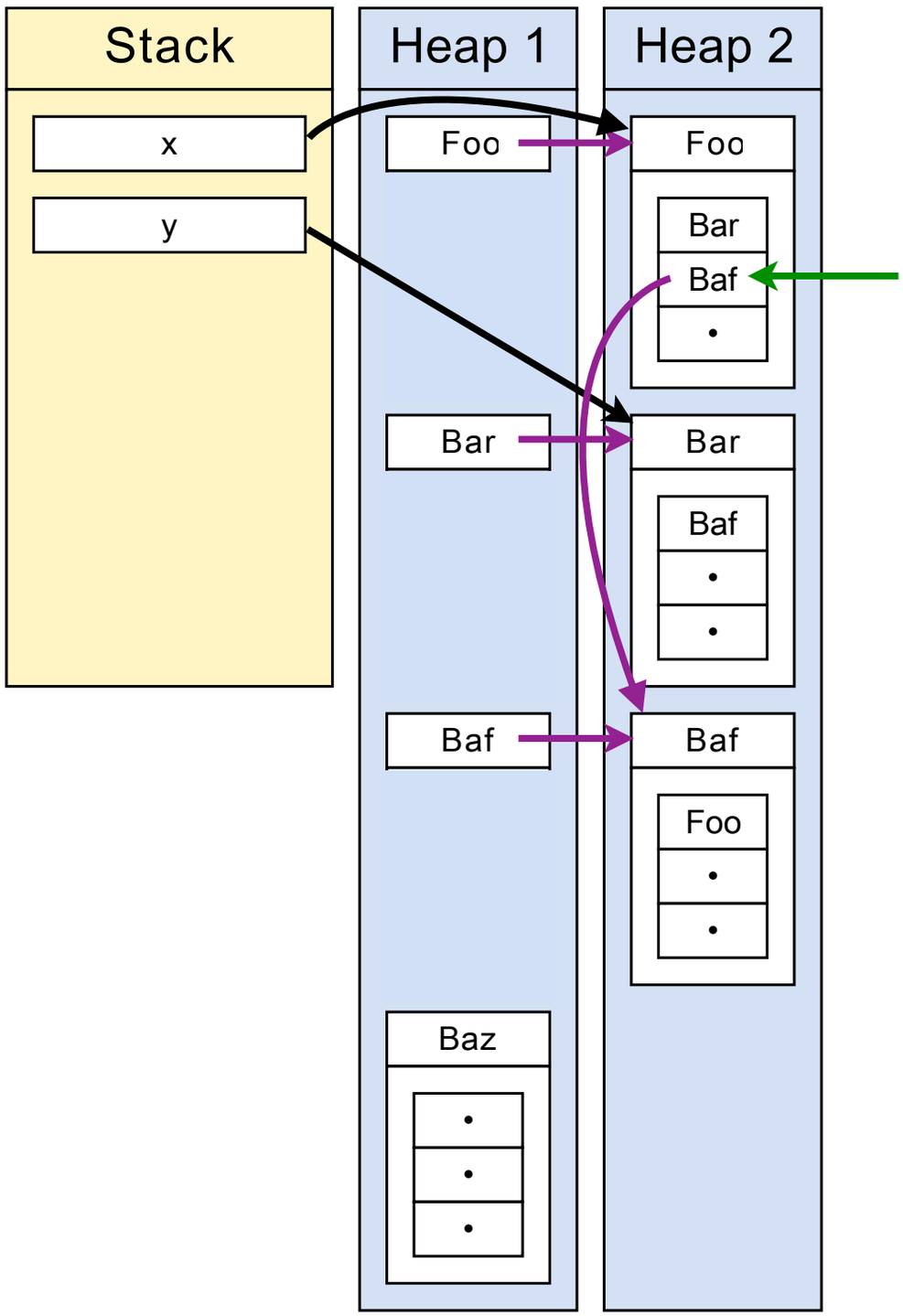
Two-Space Copying



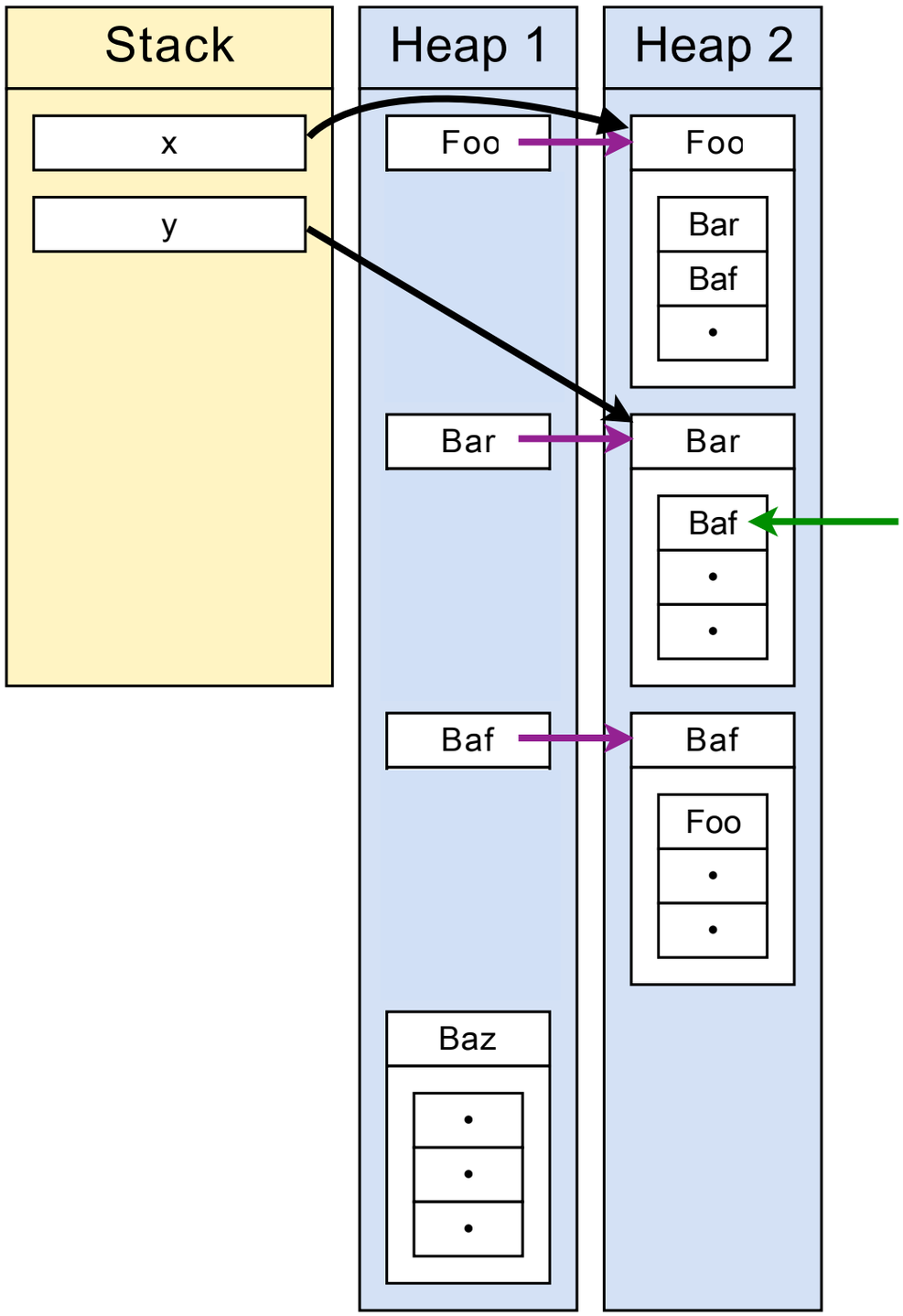
Two-Space Copying



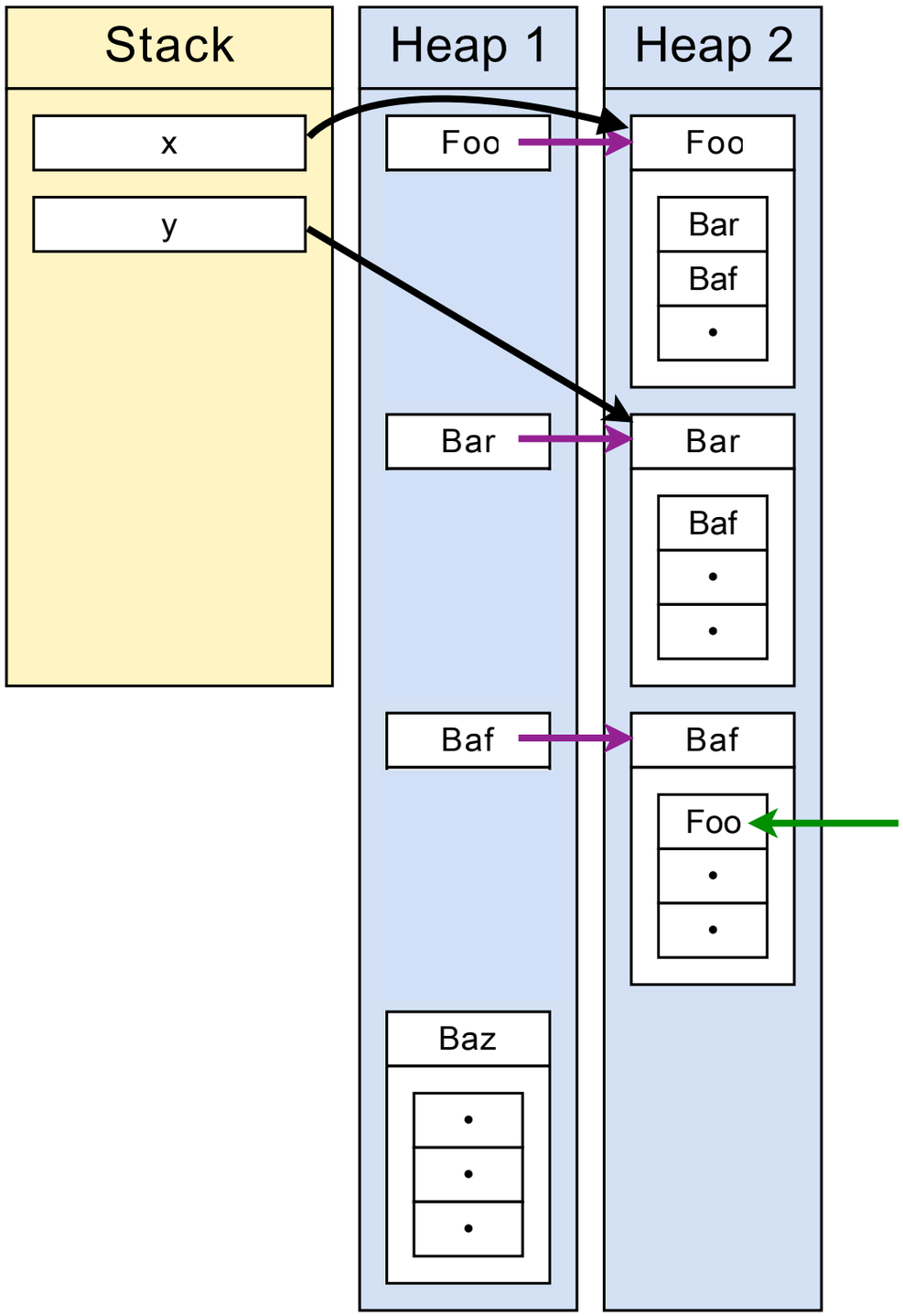
Two-Space Copying



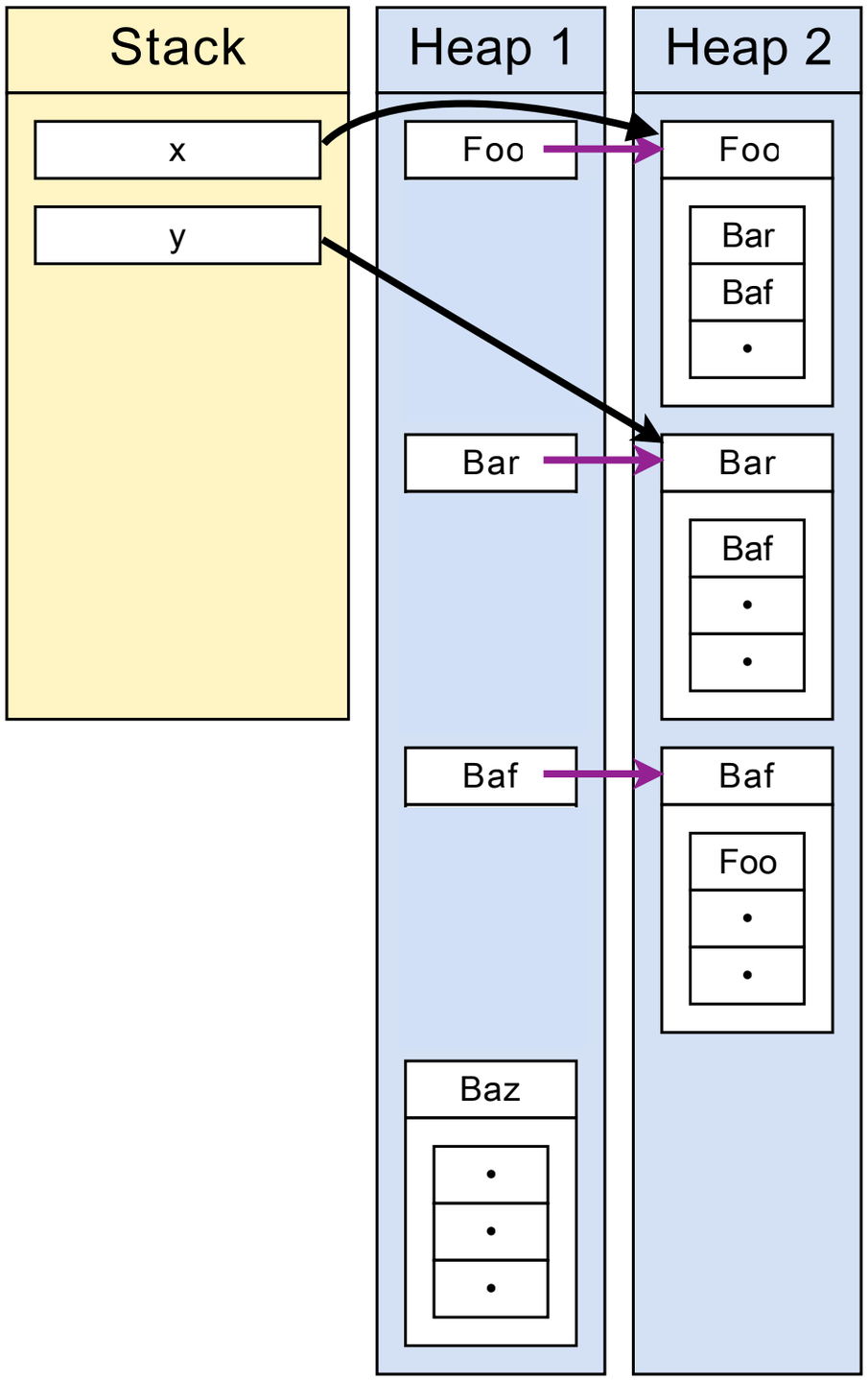
Two-Space Copying



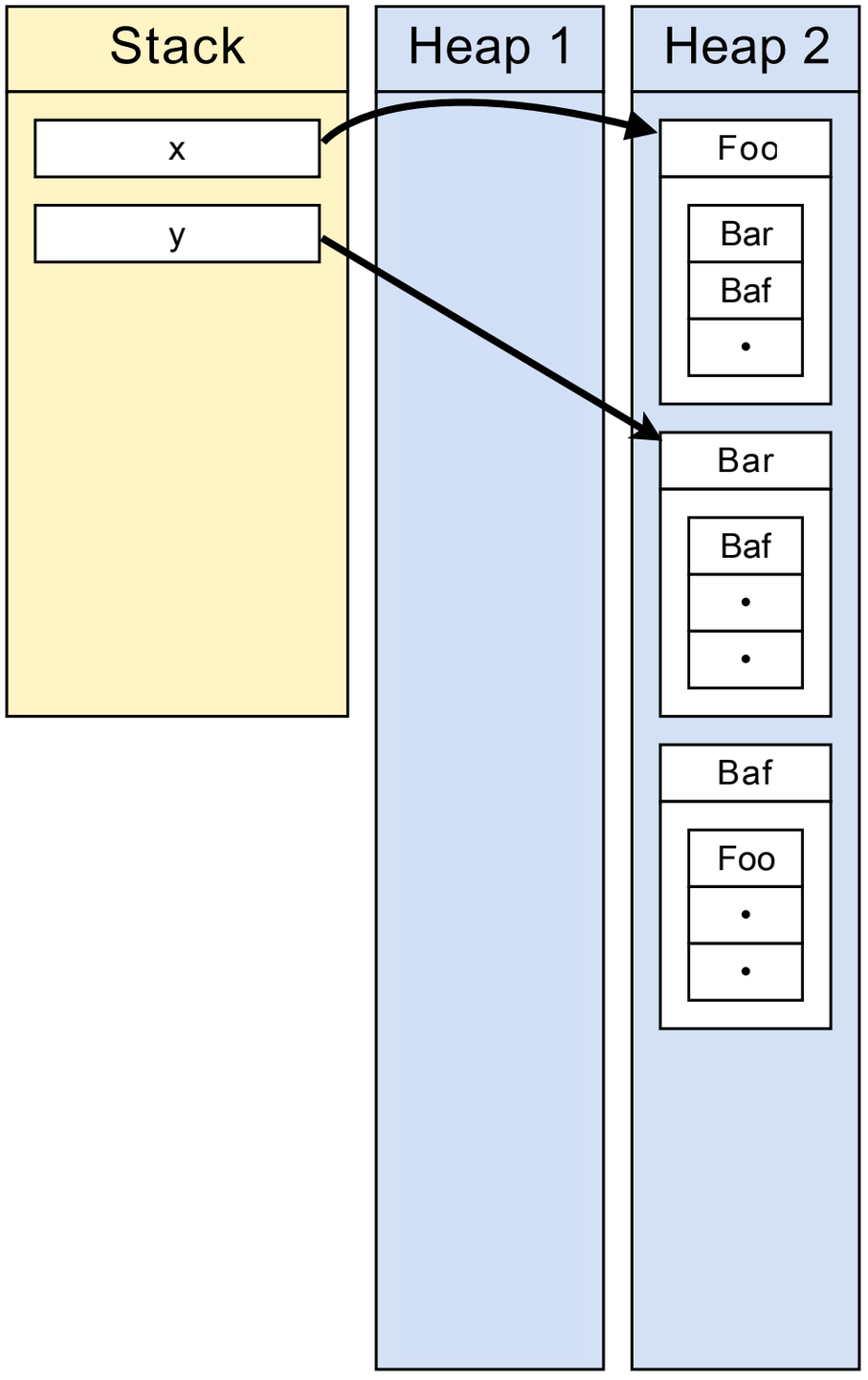
Two-Space Copying



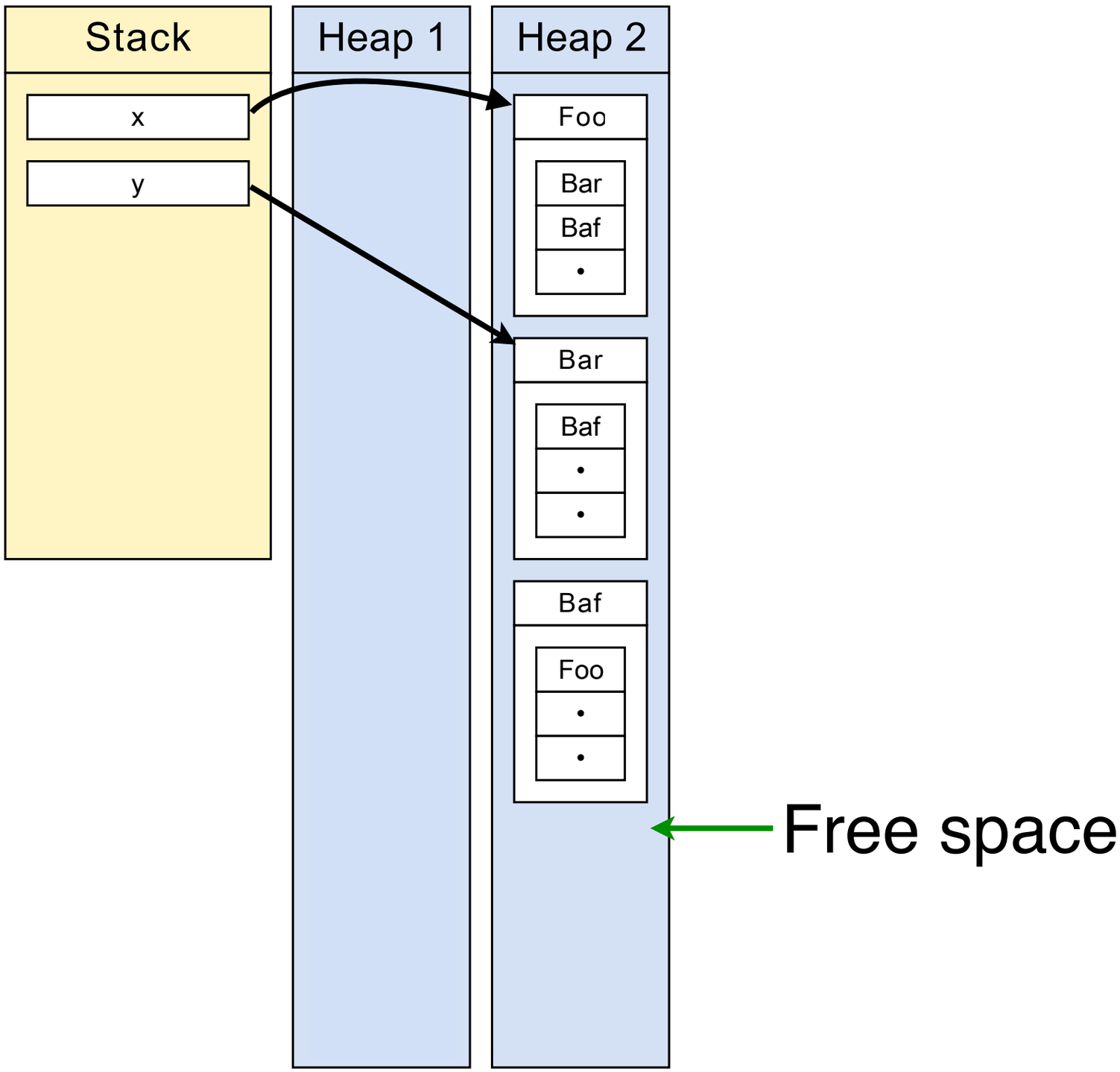
Two-Space Copying



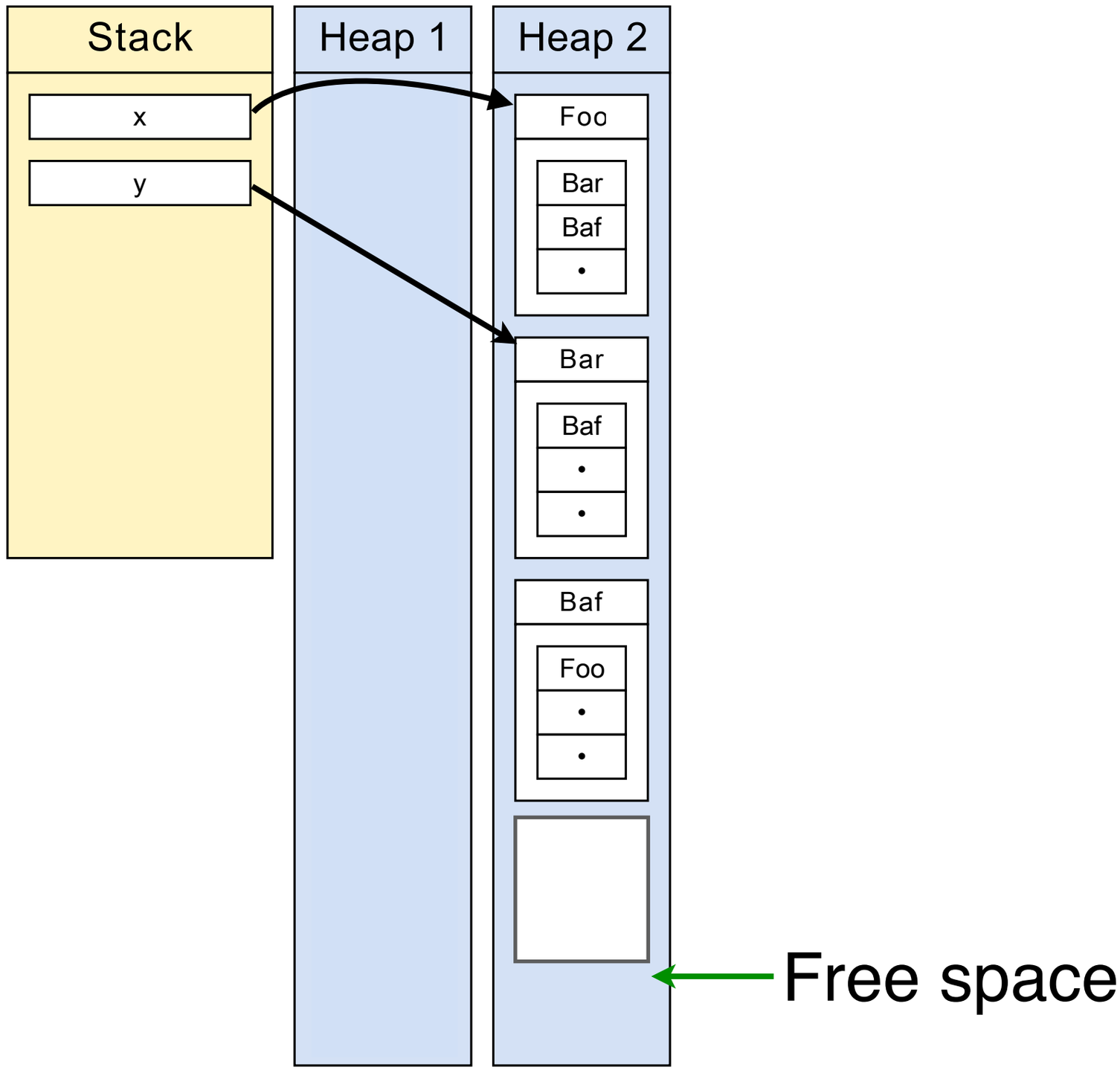
Two-Space Copying



Allocation



Allocation



Algorithm

- 1st copy objects referenced by roots
- When copying an object, leave a forwarding pointer in old heap
- Then follow heap references

Complexity

Copying?!

Isn’t that expensive?!

Complexity

- O(R)
  - No ‘H’ factor
  - (But we’ve hidden a larger constant)
- Amortized cost ~ (R)/(H/2-R)

Pros

- Objects with pointers to each other are compacted (good locality)
- Sweep phase is free, no O(H) phase
- Allocation is trivial
  - No freelists
  - Just increment a pointer

Cons

- Copying is slow
- Wasting half of heap
- Moving objects is error prone

Generational GC

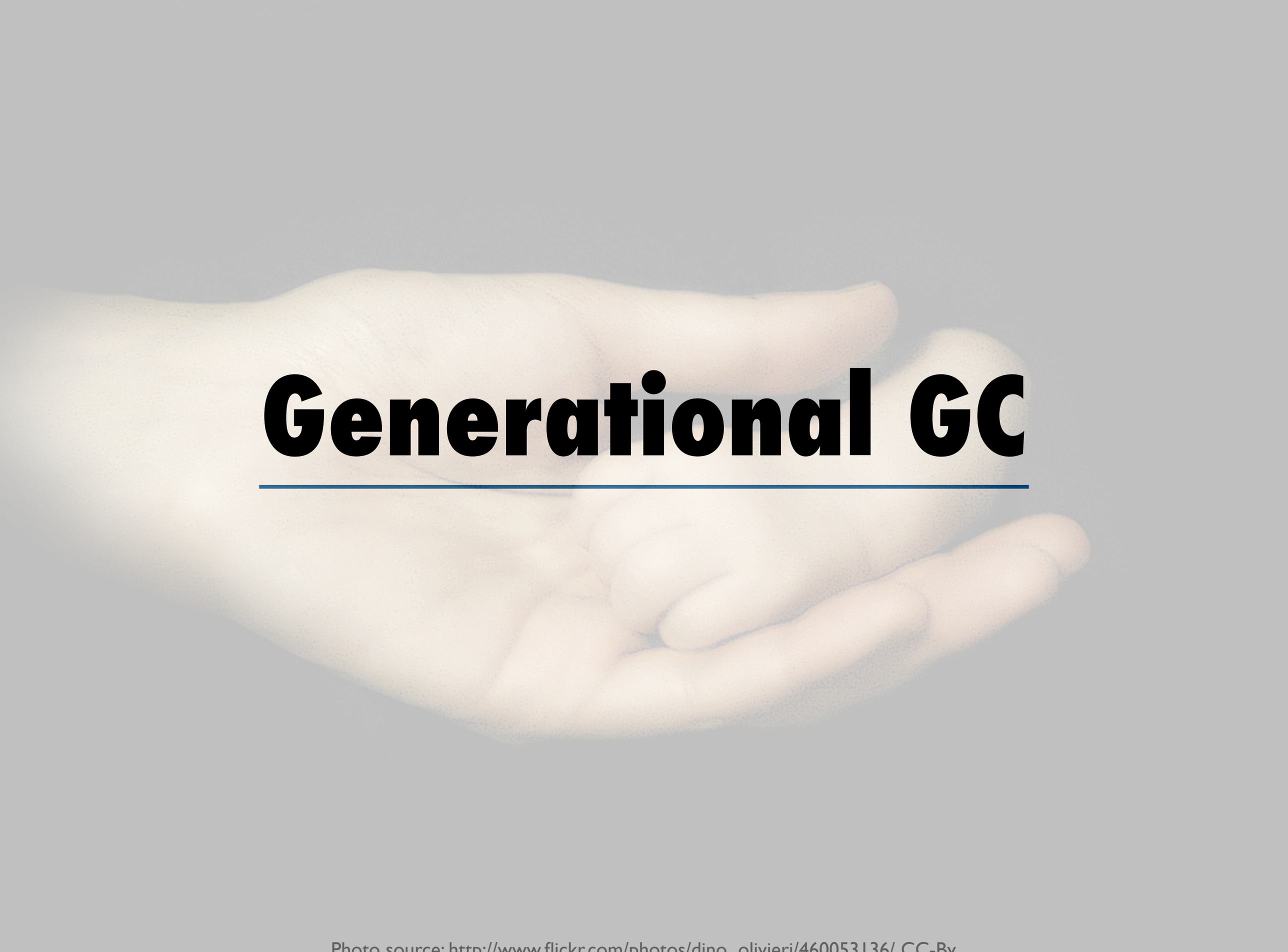


Photo source: http://www.flickr.com/photos/dino_olivieri/460053136/ CC-By

Generations

Generations

Weak generational hypothesis:

- Most allocated objects die young.

Generations

Weak generational hypothesis:

- Most allocated objects die young.

Empirically:

- (1) 80-95% of objects die young
- (2) most remaining objects have very long lives

Generational GC

- Separate the heap into n *generations* G(0)...G(n-1)
  (Typically n=2)
- Allocate in G(0)
- Collect G(x) more frequently than G(x+1)
- Long-surviving objects in G(x) moved to G(x+1)

Tracing G(0)

- Do smaller, shorter collections: Trace only G(0)
- ... but what if the only pointer to an object in G(0) is in G(1)?
- Need a *write barrier* to remember objects in G(1) with pointers to G(0)

Why Generational?

Pros:

- Usually only collect G(0)
  - Fast because most objects are dead
  - Locality, sweep, allocation benefits of copying

Cons:

- Tracing more complicated
  - Pointers from G(x+1) to G(x), write barrier
- Need separate GC for oldest generation

Compiling for GC

Allocation

Allocation

Manual memory management:

- Allocation just a function call (malloc)

Allocation

Manual memory management:

- Allocation just a function call (malloc)

Automatic memory management:

- Can’t collect mid-allocation (state inconsistent)
- Must assure that references to allocated objects are known

Collection

Collection

Manual memory management:

- free is just a function call

Collection

Manual memory management:

- free is just a function call

Automatic memory management:

- Must occasionally be in a safe state for collection
- Preemptive: Can pause at *any* point
- Yield-point based: Can pause only at compiler-generated points

Codegen

Codegen

Manual memory management:

- Pointers are just integers!

Codegen

Manual memory management:

- Pointers are just integers!

Automatic memory management:

- Pointers must always be correct
- Locations of pointers must be known
- May need a write barrier

Summary

Summary

- Languages with references can GC
- GC families: mark and sweep, copying
- GC choices affect allocation, predictability, ...
- Using a GC affects codegen, but is easier for programmers