Heap Memory Management: Garbage Collection

CS439: Principles of Computer Systems

March 23, 2015

Last Time

- Finished Virtual Memory
  - Load control
  - Page sizes
- Discussed explicit memory management
  - Allocation policies (bump pointer, free list)
  - De-allocation policies (free)
  - Free-list management

Today's Agenda

- Garbage Collection
  - How to automatically find garbage
  - Heap management
    - Incremental vs. Whole
    - Age-based collection
- Virtual Memory Review

Garbage Collection: Automatic Memory Management

- Reduces programmer burden
  - Less user code compared to explicit memory management
  - Eliminates sources of errors
  - Less user code to get correct
  - Protects against some classes of memory errors
    - No free(), thus no premature free(), no double free(), or forgetting to free()
  - Not perfect, memory can still leak
- Integral to modern object-oriented languages
- Now part of mainstream computing

Garbage Collection brought to you by…

- "Safe" pointers!
  - Used in managed languages that provide garbage collection
  - Programs may not access arbitrary addresses in memory
  - The compiler can identify and provide to the garbage collector all the pointers, thus
  - "Once garbage, always garbage"
  - Runtime system can move objects by updating pointers
- "Unsafe" languages can do non-moving GC by assuming anything that looks like a pointer is one.

Challenge

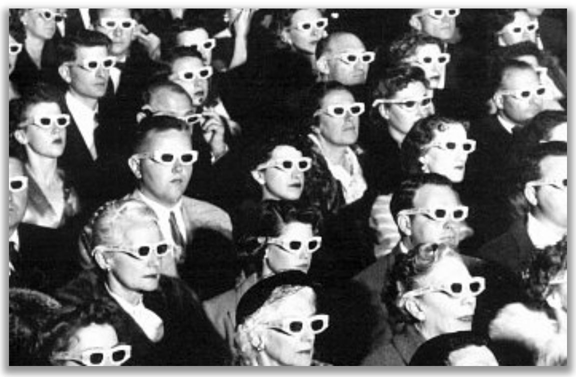
Performance efficiency!

Identifying garbage is hard and expensive.

- Time proportional to dead objects or live objects
  - Which depends on method used
- Collecting less frequently typically reduces total time but increases space requirements and pause times (Space/Time Tradeoff!)
  - *Pause time* is the time the program is paused from doing its work while the garbage is being collected

Garbage Collection is Not New

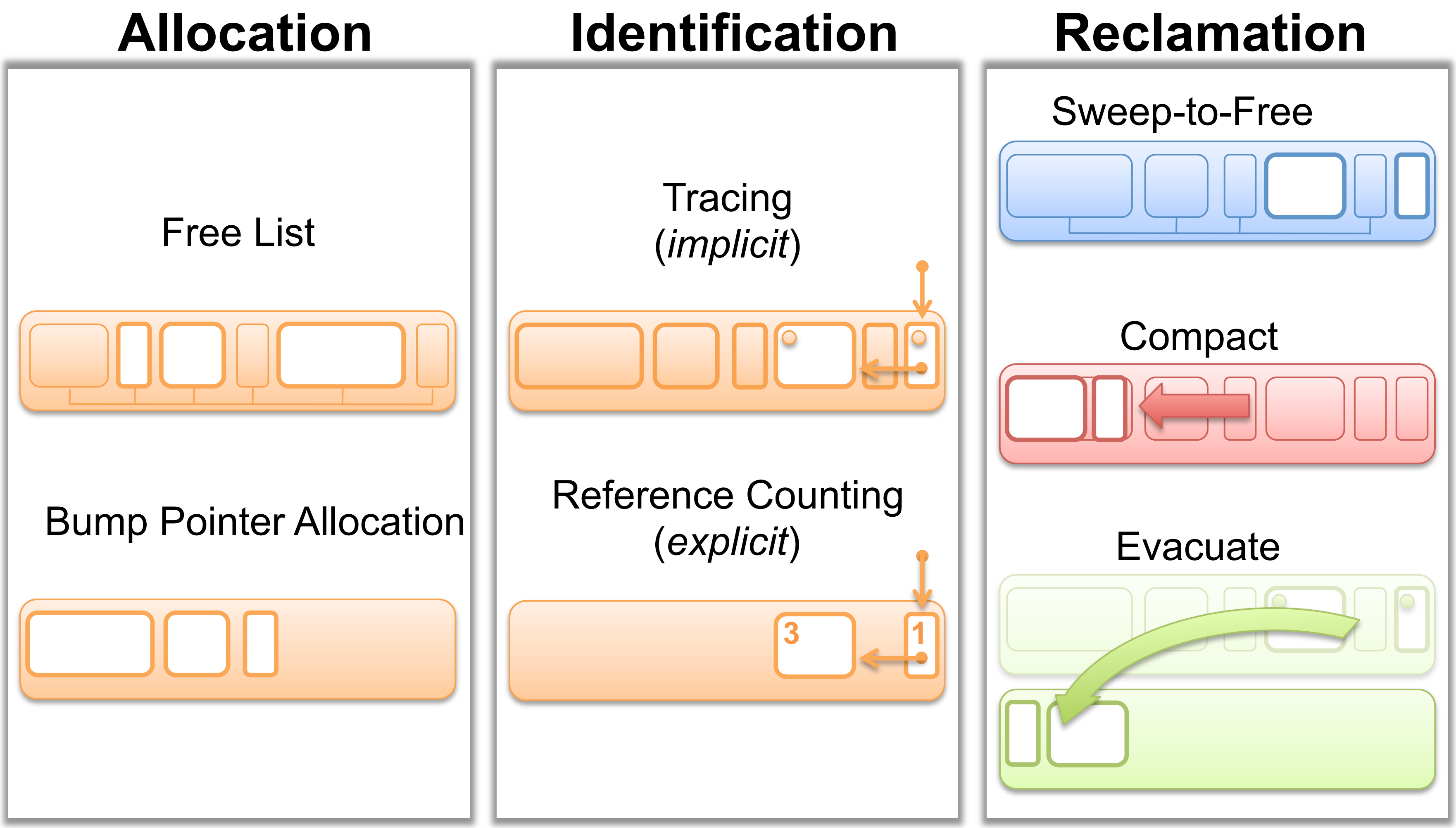
Mark-Sweep	Mark-Compact	Semi-Space
McCarthy, 1960	Styger, 1967	Cheney, 1970



revised from Blackburn & McKinley

Garbage Collection Fundamentals

Algorithmic Components



revised from Blackburn & McKinley

Garbage Collection Fundamentals: Text Description

- Algorithmic Components
  - Allocation (Same as explicit memory management)
    - Free List
    - Bump Allocation
  - Identification
    - Tracing – implicit
      - Traces pointers program objects to identify what is reachable in the program
      - Definitions in a few slides
    - Reference Counting – explicit
      - A programmer/ program keeps track of how many objects are using a pointer
  - Reclamation: taking back the heap
    - Sweep-to-Free
    - Compact
    - Evacuate
- Emphasis for this slide is on the Allocation and Identification components, description of Reclamation soon.

What is Garbage?

- In theory, any object the program will never reference again
  - Called *dead* objects
  - Dead objects cannot be identified by the compiler or runtime system
- In practice, any object the program cannot reach is garbage
  - Called *unreachable* objects

Finding Reachable Objects

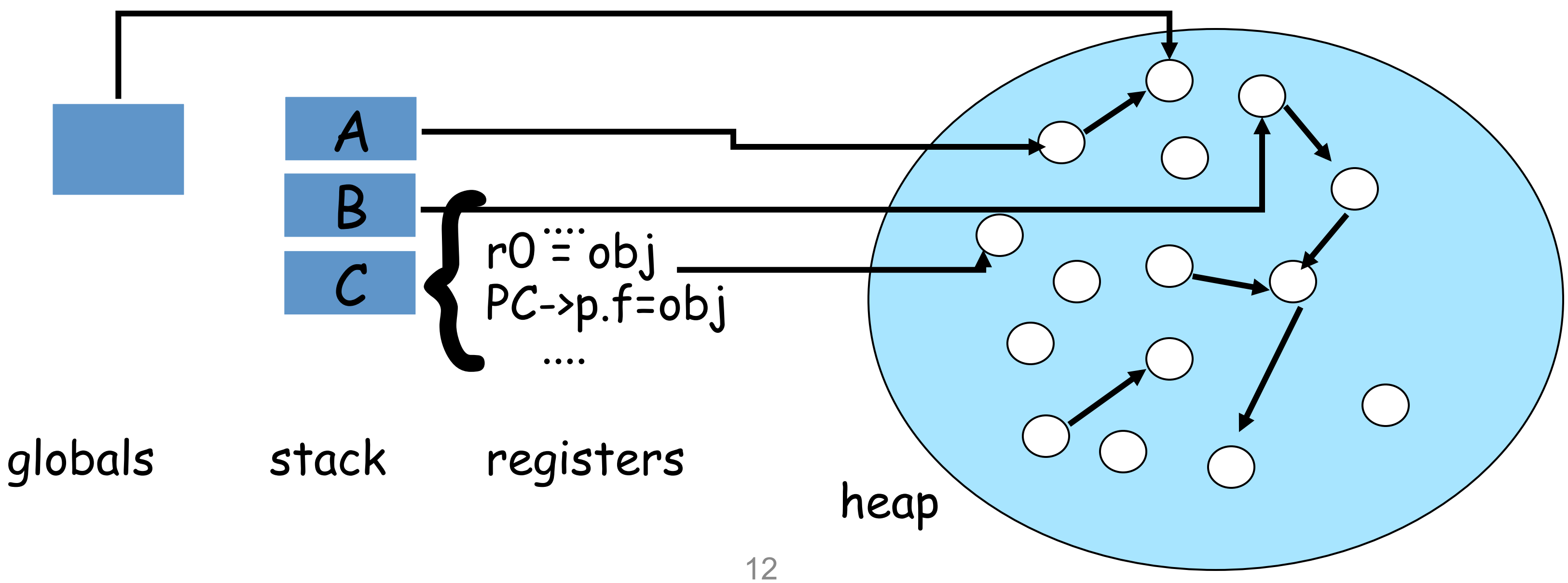
Two methods:

- Reference Counting
  - Count the number of references to each object
  - If the reference count is 0, the object is garbage
  - Does not work for cycles
- Tracing
  - Trace reachability from program *roots*
    - Roots are: registers, stack, static variables
  - Objects not traced are unreachable

Reachability

Tracing:

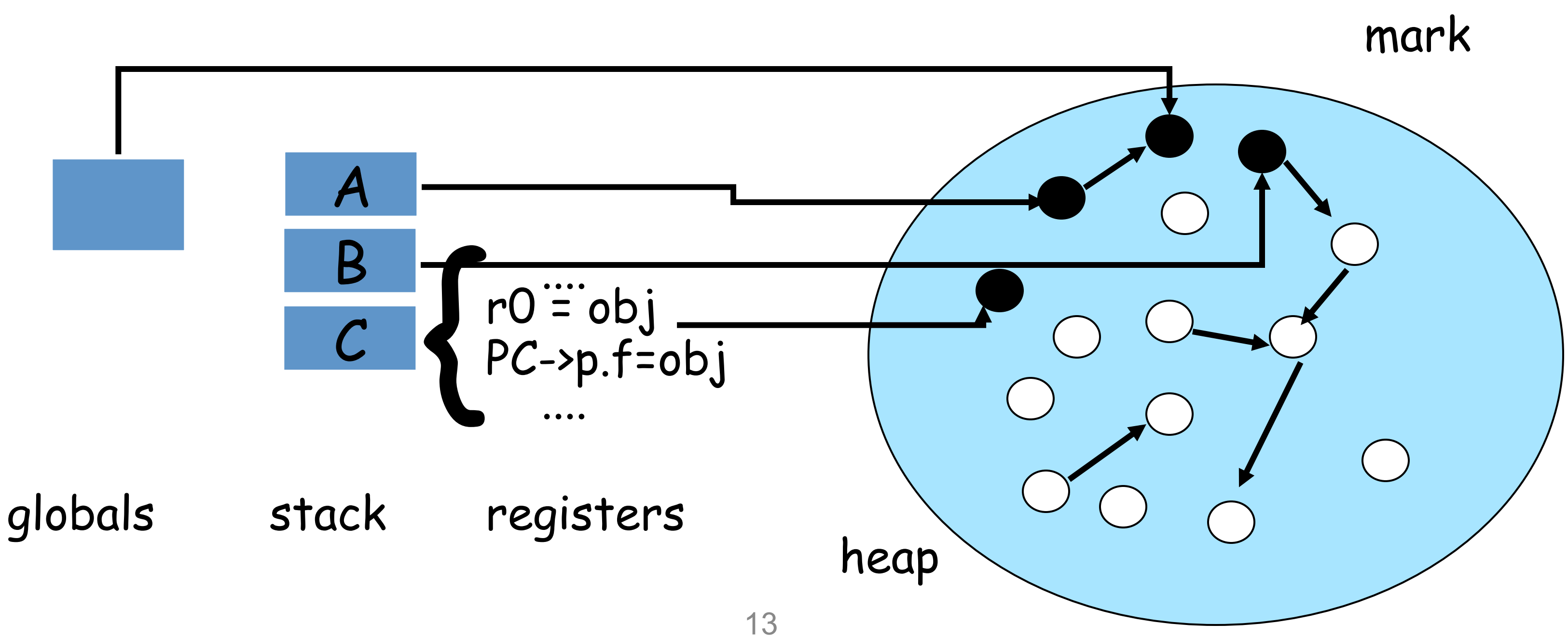
– Marks the objects reachable from the roots live, and then performs a transitive closure over them



Reachability

Tracing:

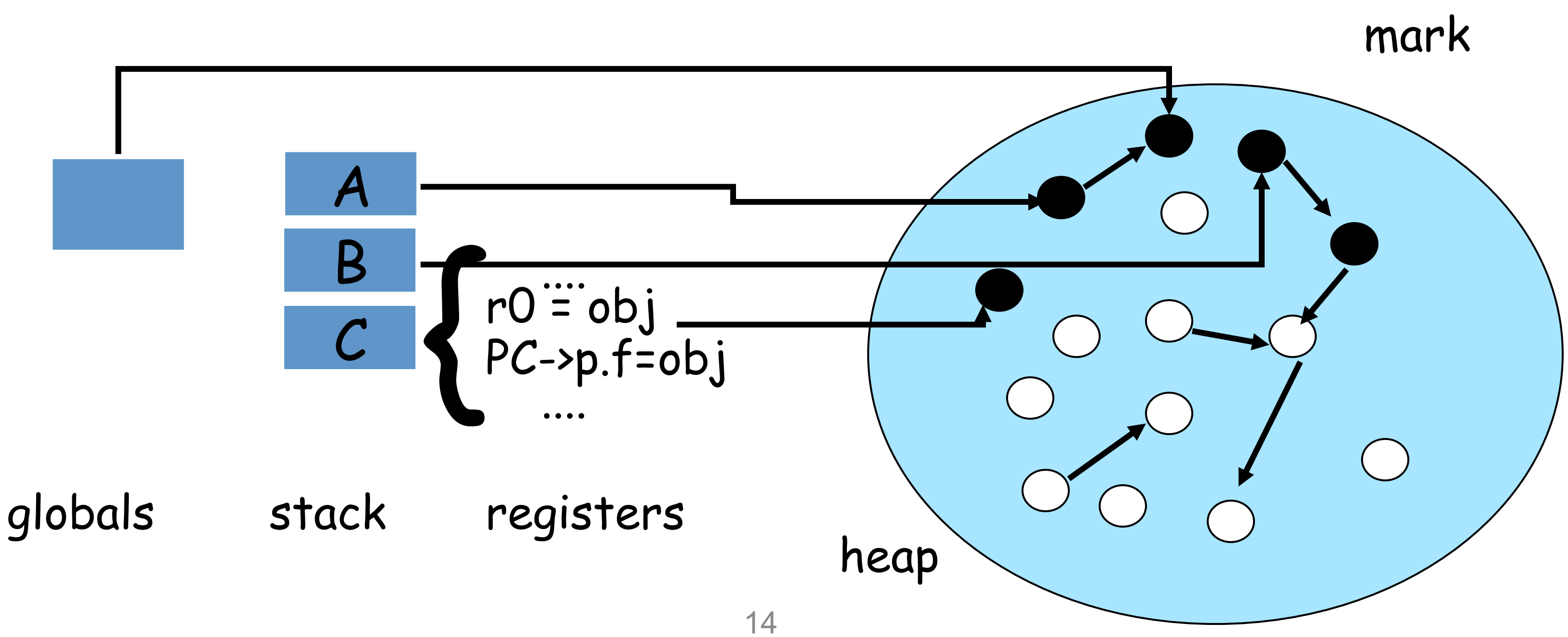
– Marks the objects reachable from the roots live, and then performs a transitive closure over them



Reachability

Tracing:

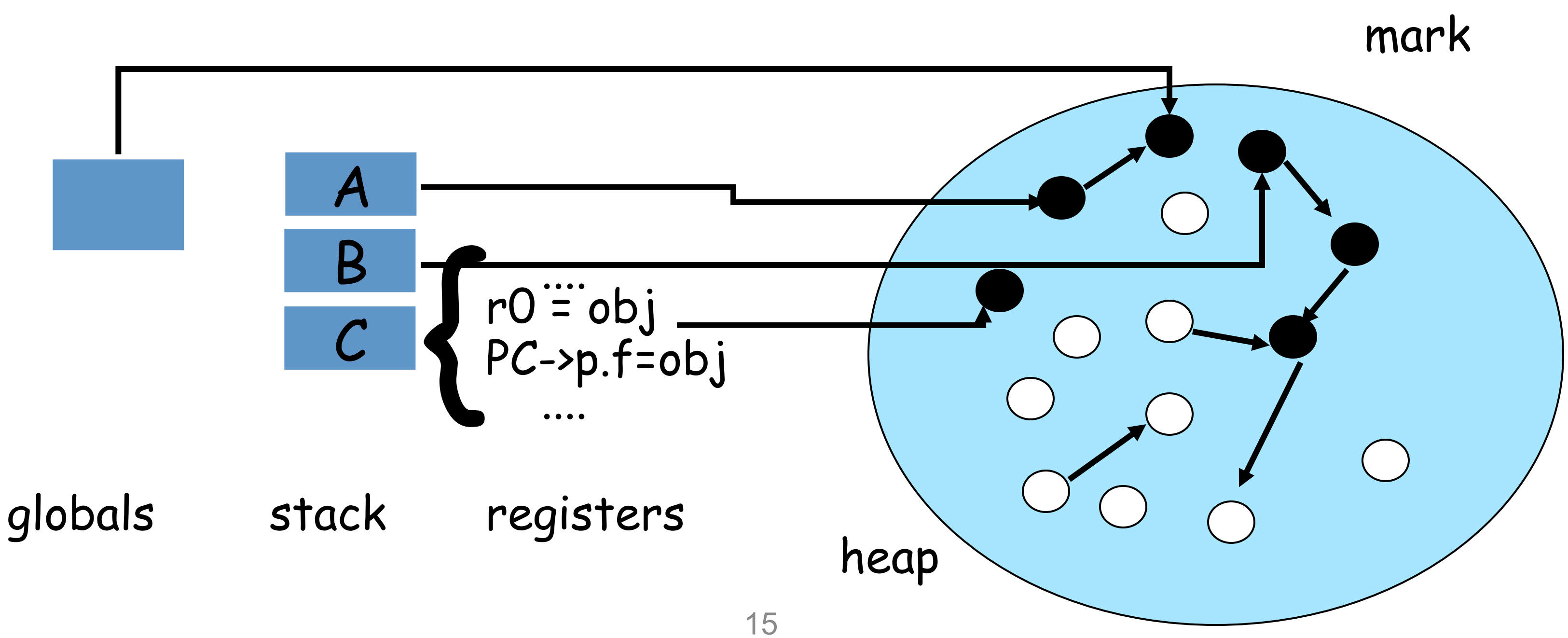
- Marks the objects reachable from the roots live, and then performs a transitive closure over them



Reachability

Tracing:

– Marks the objects reachable from the roots live, and then performs a transitive closure over them

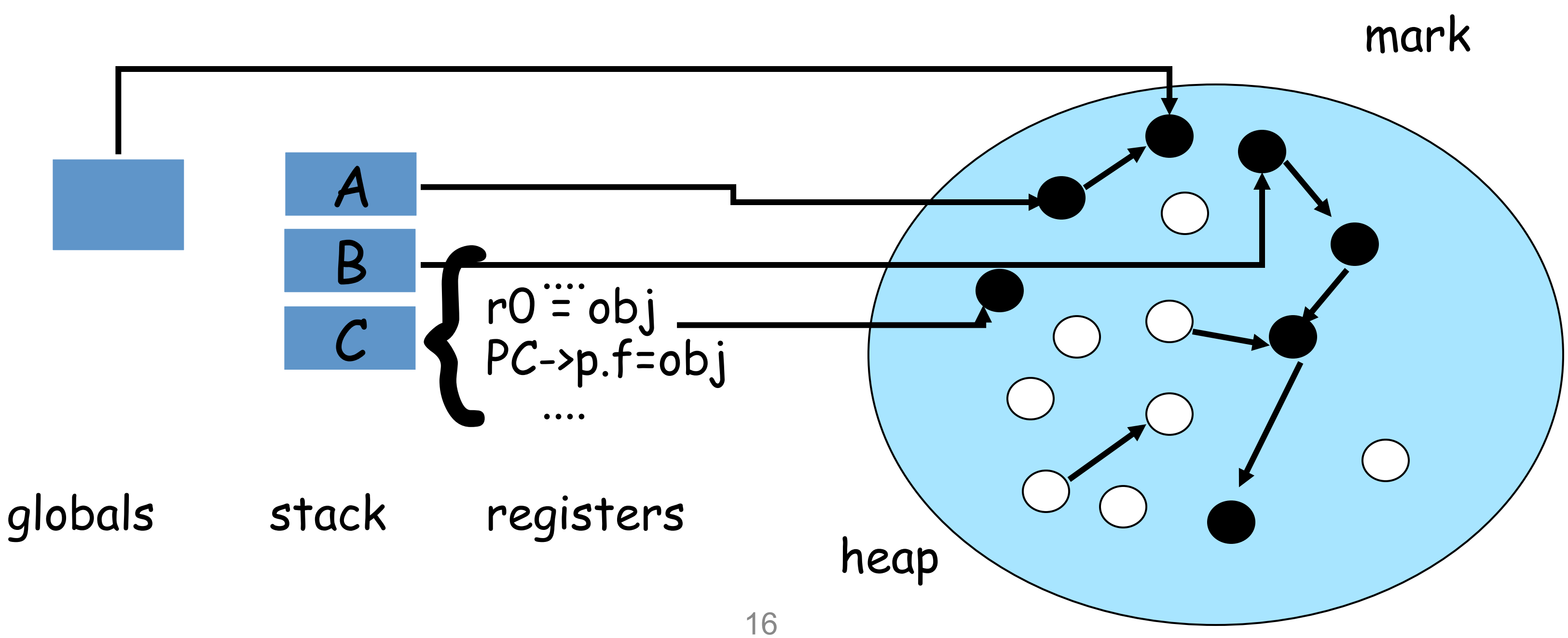


Reachability

Tracing:

– Marks the objects reachable from the roots live, and then performs a transitive closure over them

All unmarked objects are dead and can be reclaimed

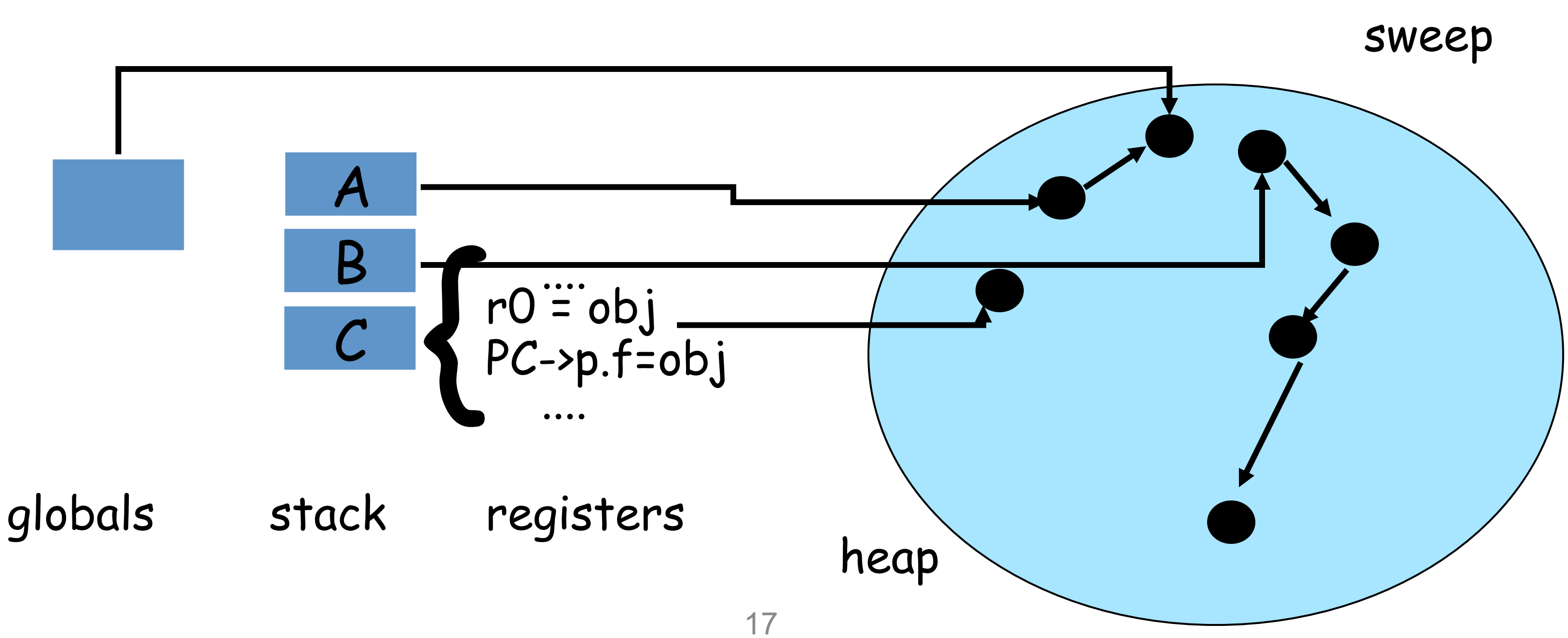


Reachability

Tracing:

- Marks the objects reachable from the roots live, and then performs a transitive closure over them

All unmarked objects are dead and can be reclaimed

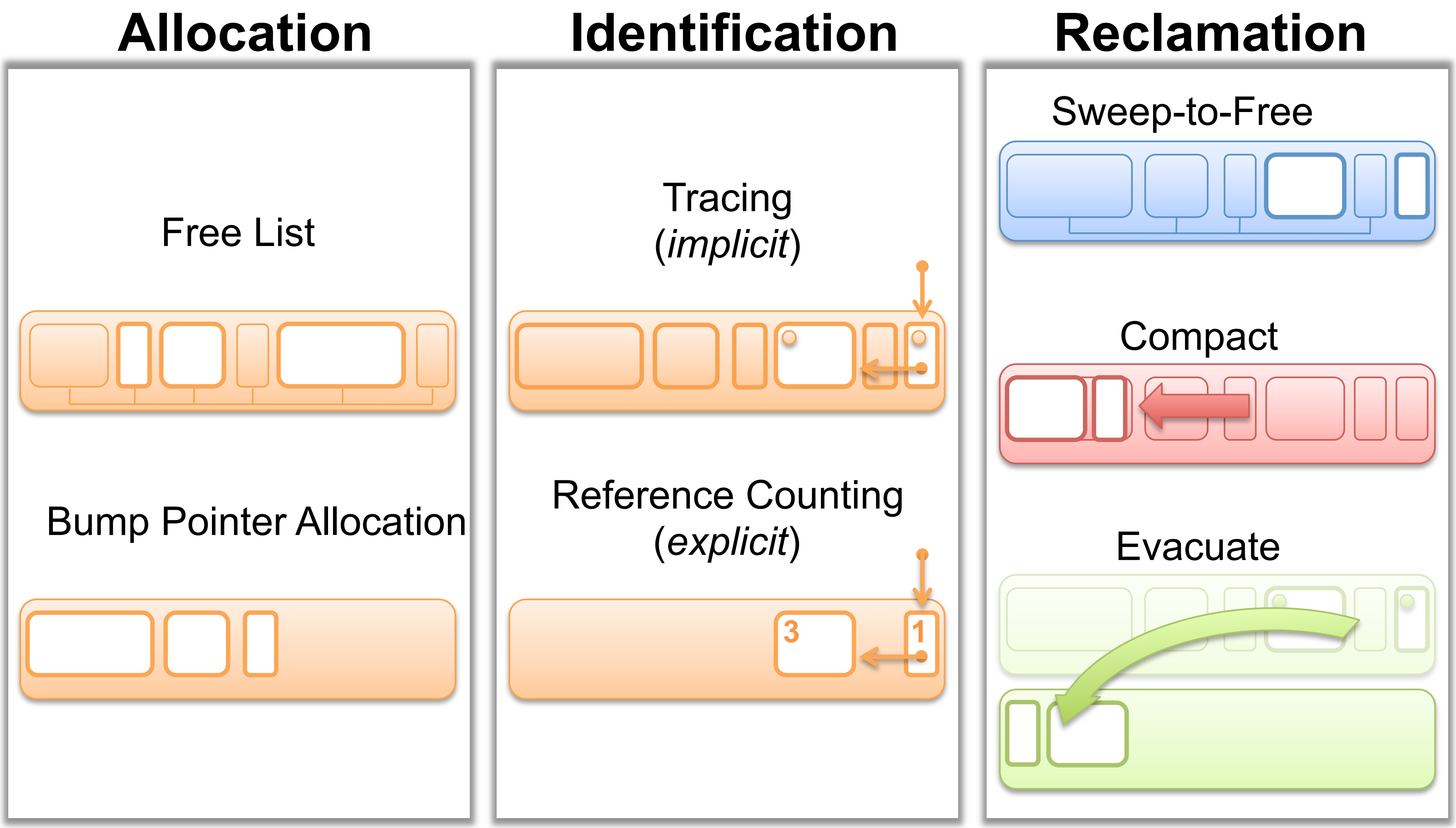


Reachability: Text Description

- Tracing
  - Marks the objects reachable from the root as reachable and then performs a transitive closure over them
- Example Setup
  - We are looking at 4 parts of the memory system
    - 1 global variable
    - 3 stack variables
    - Some register values, which are living in one of the stack variables (but we would look at the registers regardless)
    - Heap allocated memory
  - The global variable, stack variables, and registers all point to space allocated on the heap
- Steps:
  - 1. Mark the objects on the heap that are directly pointed to by our variables, which are the roots of our example
  - 2. Keep marking any objects that are reachable from those objects in the heap
    - For example, Stack variable B points to object alpha which points to object beta, which points to objects delta and omega. So alpha, beta, delta and omega are all reachable from B.
    - Must follow all trails until they end
  - 3. “Sweep” all dead objects
    - All unmarked objects are dead and can be reclaimed
    - Now that heap memory can be recycled
  - 4. Wait for garbage collection algorithm to be run again.

Garbage Collection Fundamentals

Algorithmic Components



revised from Blackburn & McKinley

Immix

Garbage Collection Fundamentals: Text Description

- This slide is a repeat of the one described in slide 9, but the emphasis has changed as we step through the algorithmic components of garbage collection.
- Algorithmic Components
  - Allocation
    - Free List
    - Bump Allocation
  - Identification
    - Tracing
    - Reference Counting
  - Reclamation - taking back the heap
    - Sweep-to-Free
    - Compact
    - Evacuate
- Emphasis for this slide is on the Reclamation component

Evaluating Garbage Collection Algorithms

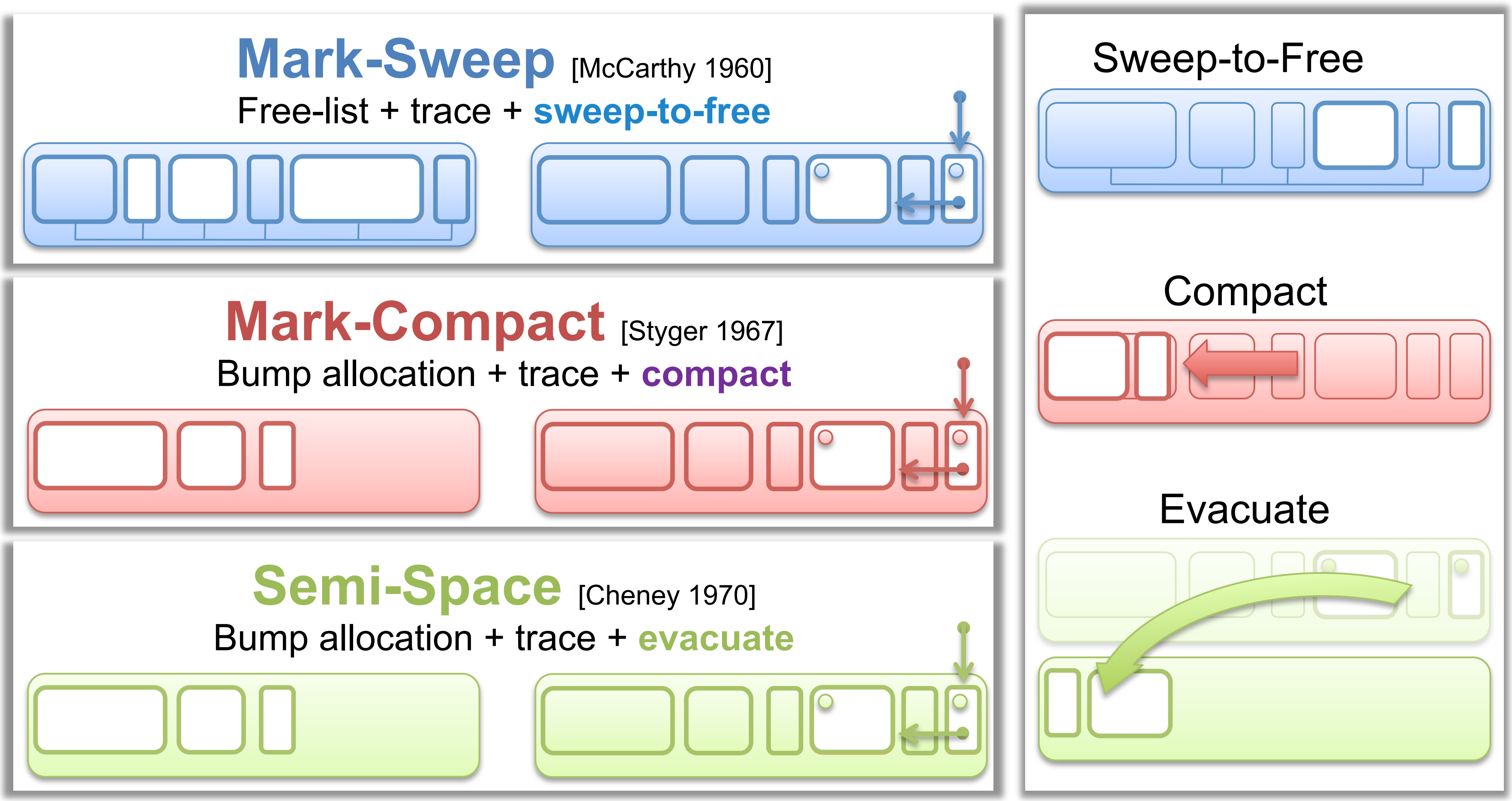
- Space efficiency
- Efficiency of allocator
  - Time to allocate
  - Locality of contemporaneously allocated objects
    - Why?
- Time to collect garbage
  - Is it fast?
  - Is incremental reclamation an option?

Reclamation

Two broad approaches:

- Non-copying
  - Uses free-list allocation and reclamation
  - Only way for explicit memory management
  - Example: Mark-Sweep
- Copying
  - (Generally) uses bump pointer allocation
  - En masse reclamation (sort of)
  - Example: Mark-Compact, Semi-Space

Garbage Collectors

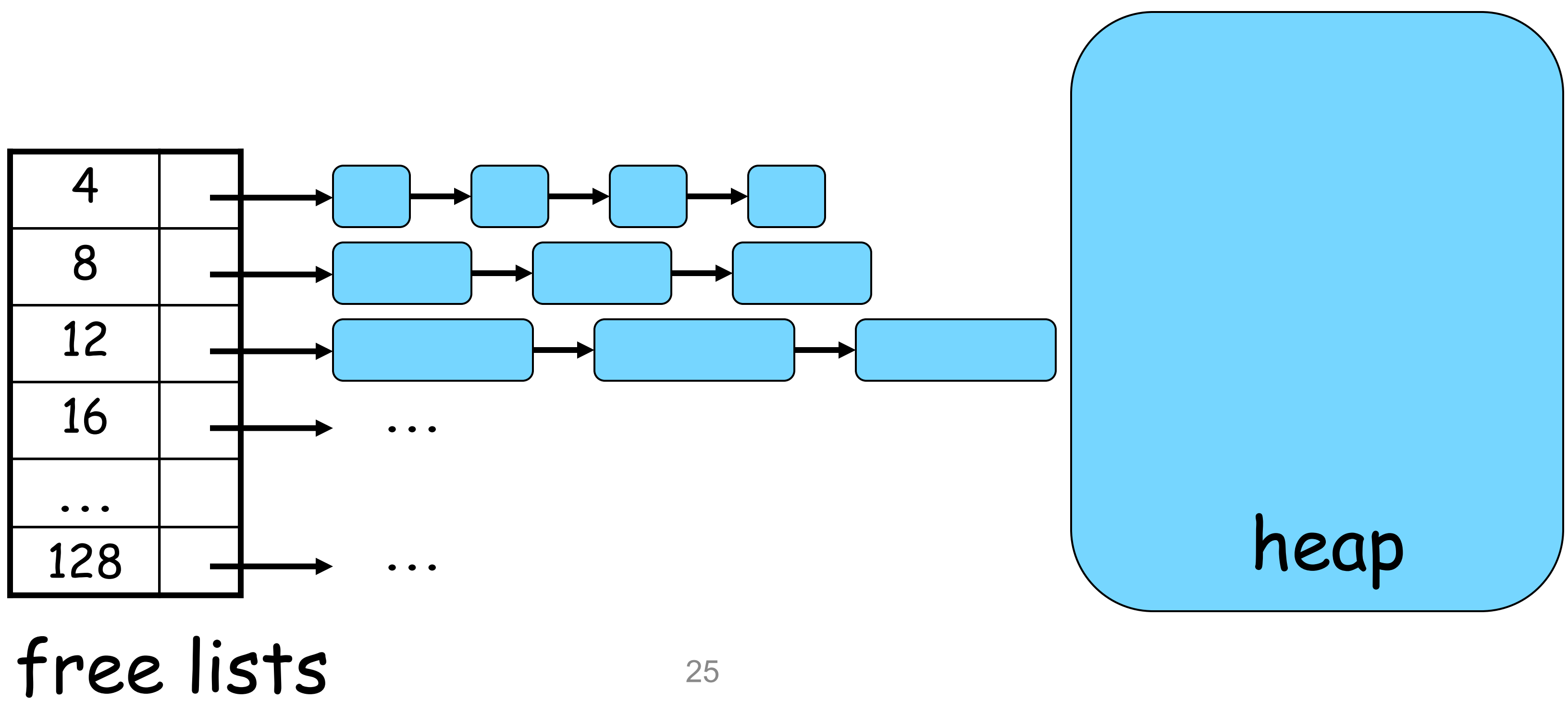


Garbage Collectors: Text Description

- Mark-Sweep - McCarthy 1960
  - free-list + trace + sweep-to-free
- Mark-Compact - Styger 1967
  - Bump allocation + trace + compact
- Semi-Space - Cheney 1970
  - Bump allocation + trace + evacuate
- Focus of this slide is Mark-Sweep, which is described in the upcoming slides

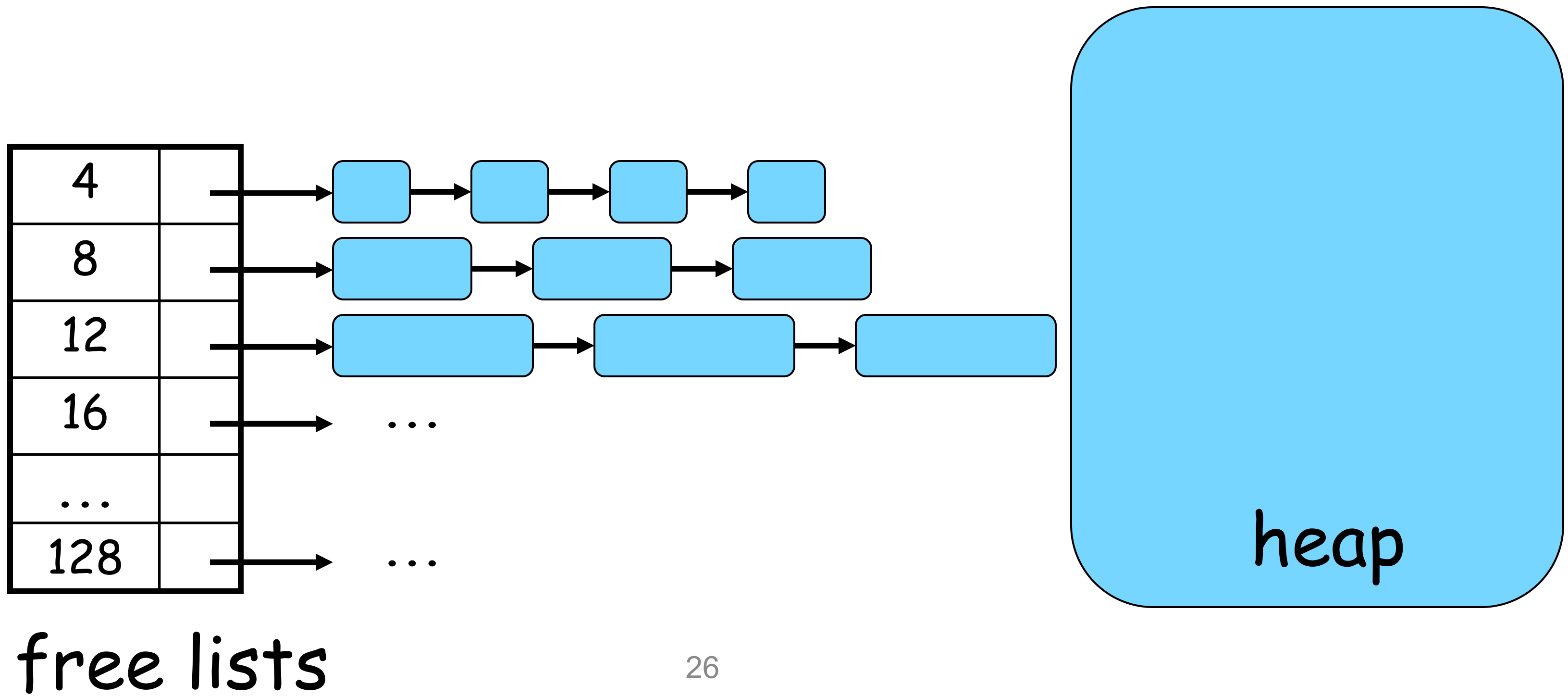
Mark-Sweep

- Free lists organized by size (binning!)
  - blocks of same size, or
  - individual objects of same size
- Most objects are small < 128 bytes



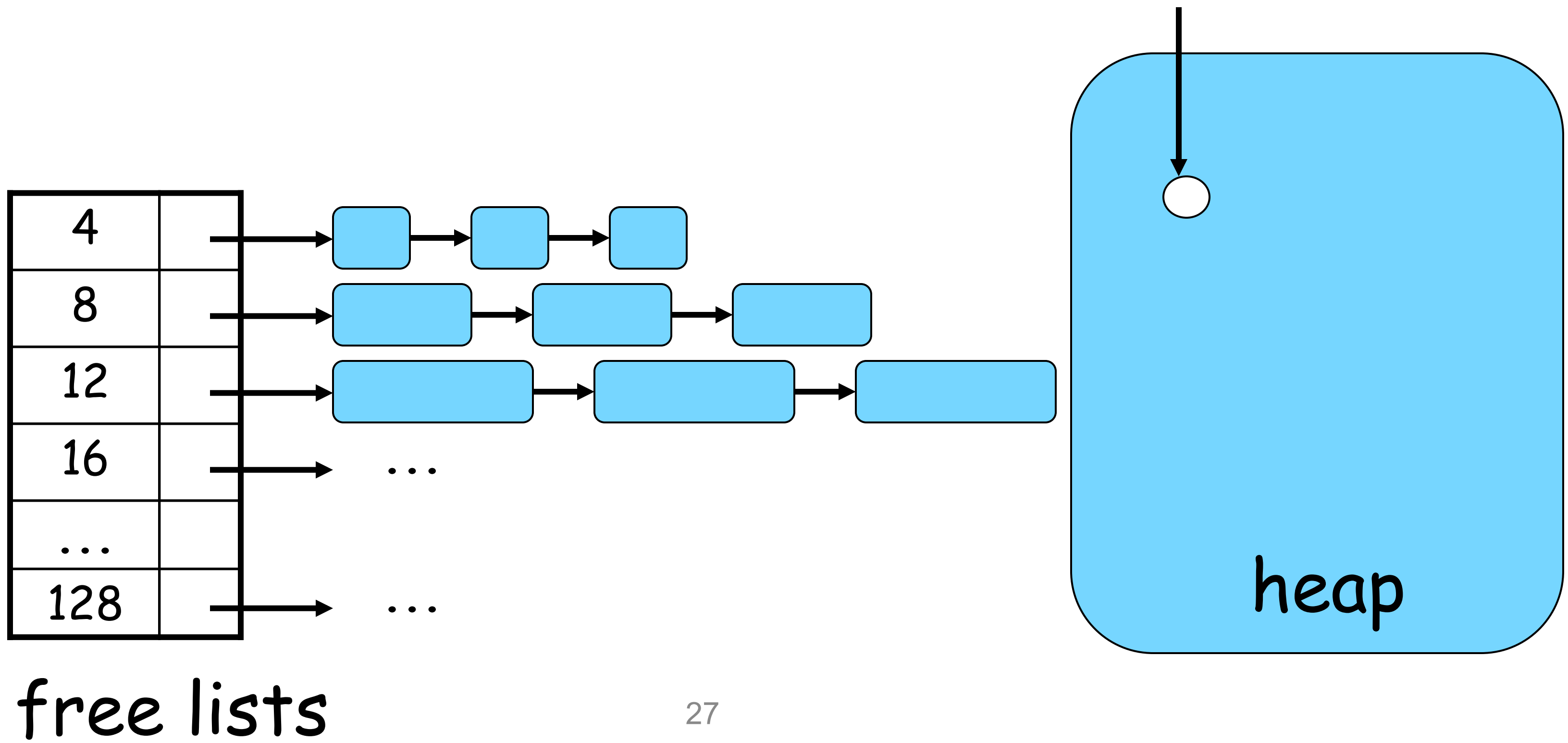
Mark-Sweep

- Allocation
  - Grab a pointer to free space off the free list



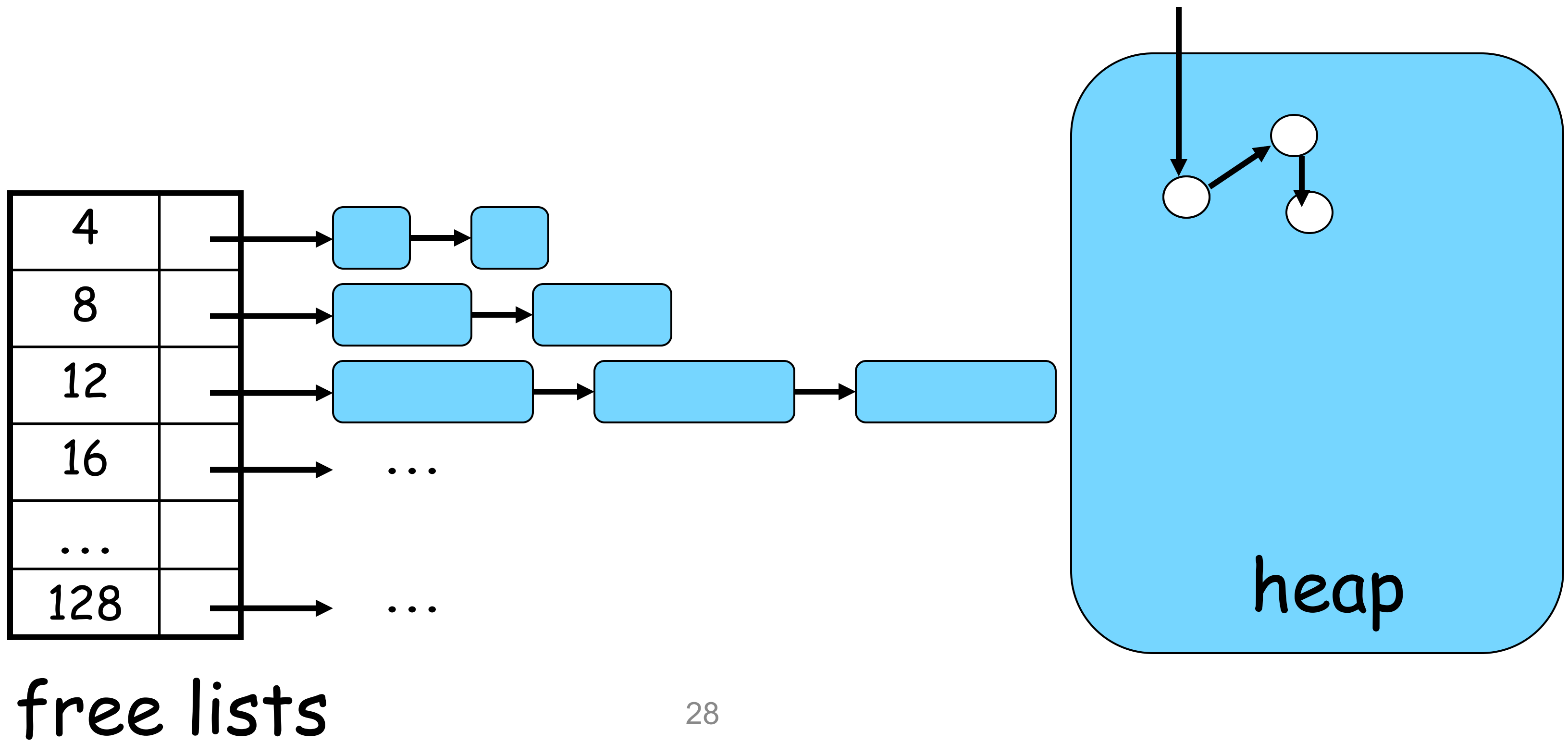
Mark-Sweep

- Allocation
  – Grab a pointer to free space off the free list



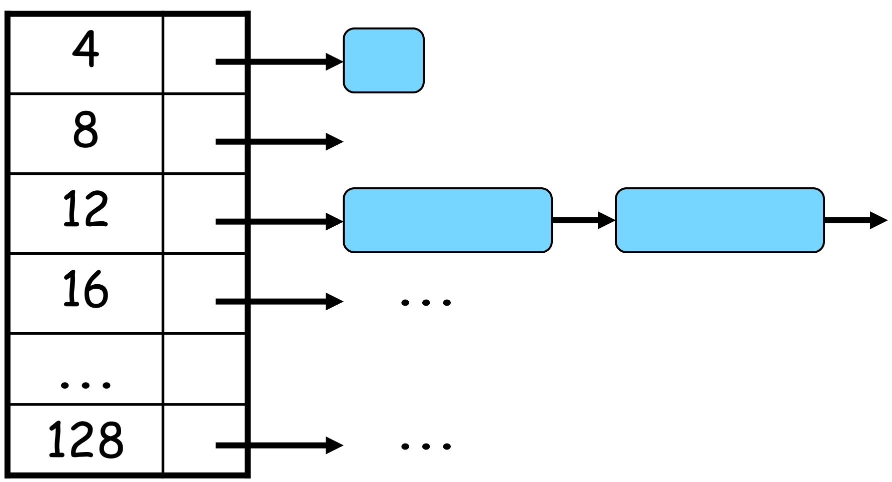
Mark-Sweep

- Allocation
  - Grab a pointer to free space off the free list

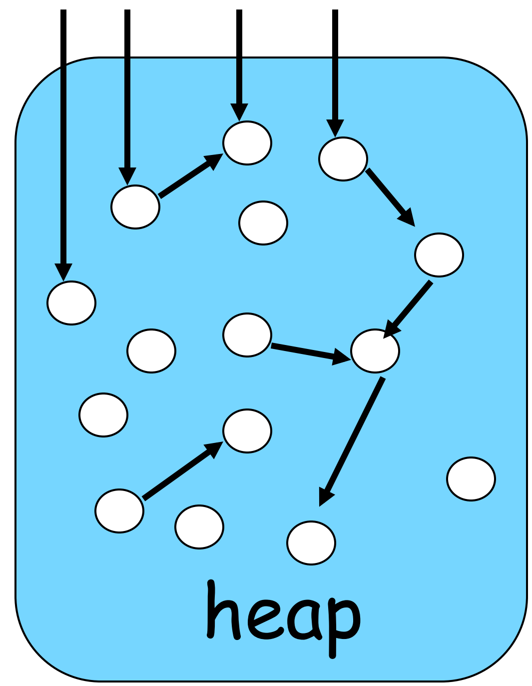


Mark-Sweep

- Allocation
  - Grab a pointer to free space off the free list
  - No more memory of the right size triggers a collection
  - Mark phase - find the reachable objects
  - Sweep phase - put unreachable ones on the free list

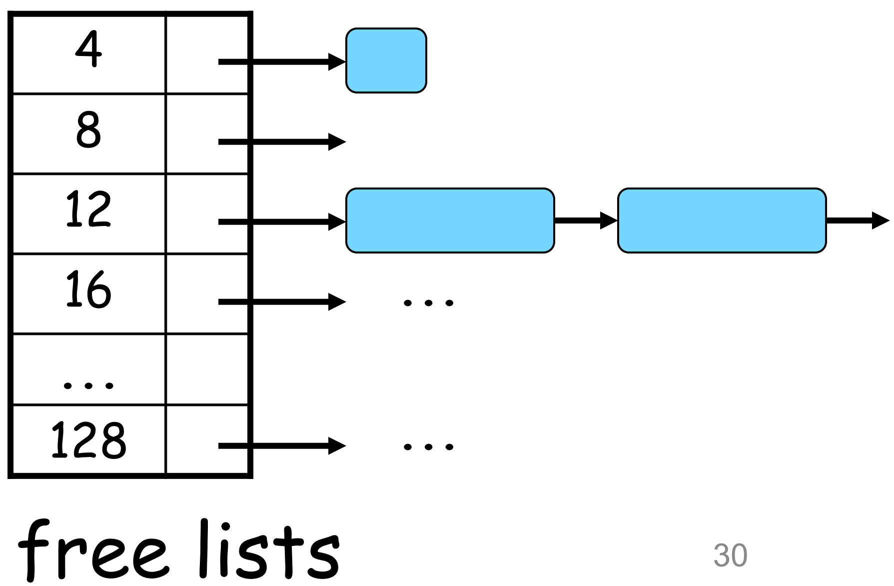


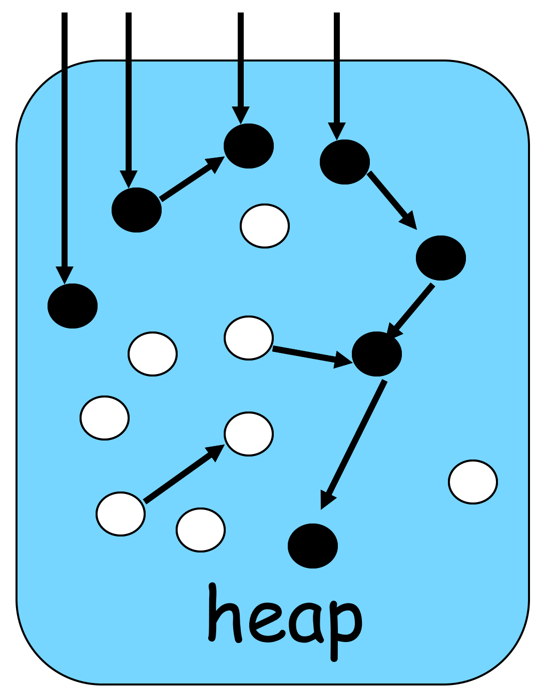
free lists



Mark-Sweep

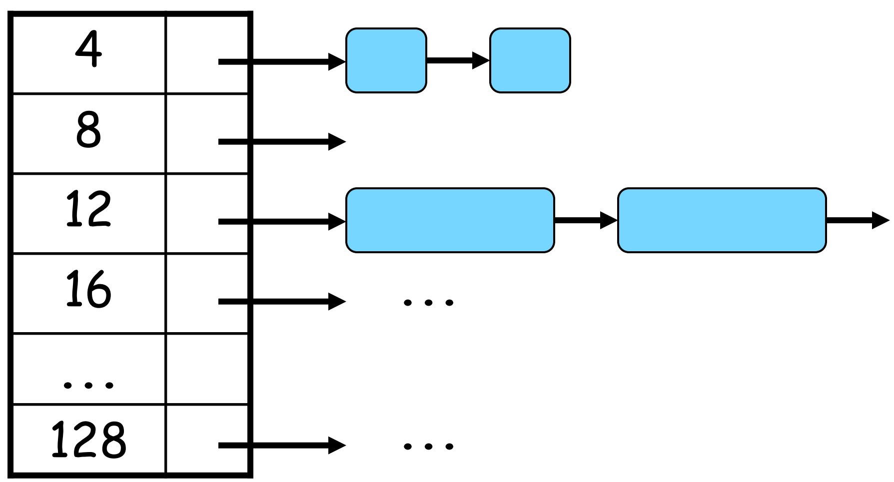
- Mark phase
  - Transitive closure marking all the reachable objects
- Sweep phase
  - sweep the memory for unreachable objects populating free list



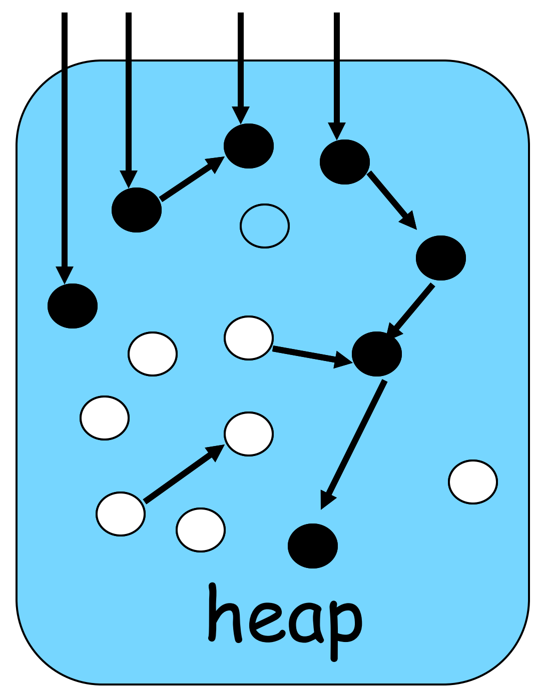


Mark-Sweep

- Mark phase
  - Transitive closure marking all the reachable objects
- Sweep phase
  - sweep the memory for unreachable objects populating free list

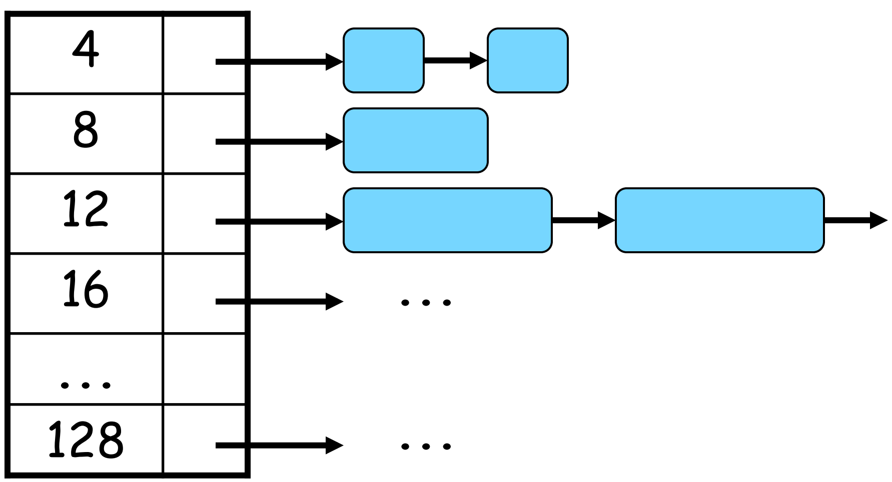


free lists

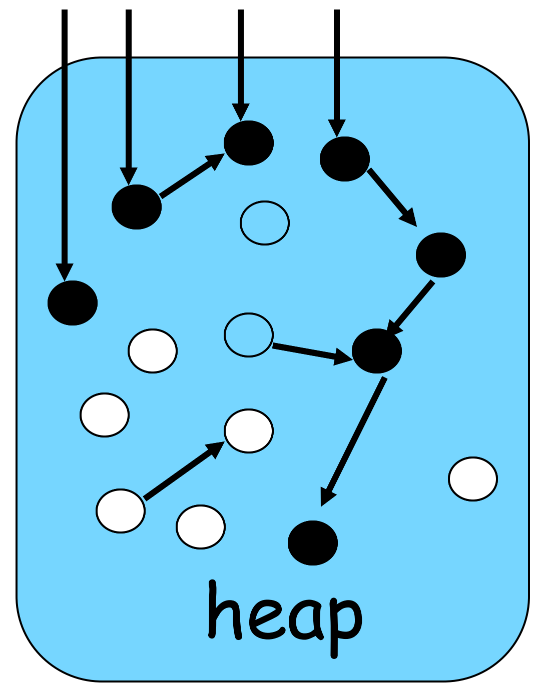


Mark-Sweep

- Mark phase
  - Transitive closure marking all the reachable objects
- Sweep phase
  - sweep the memory for unreachable objects populating free list

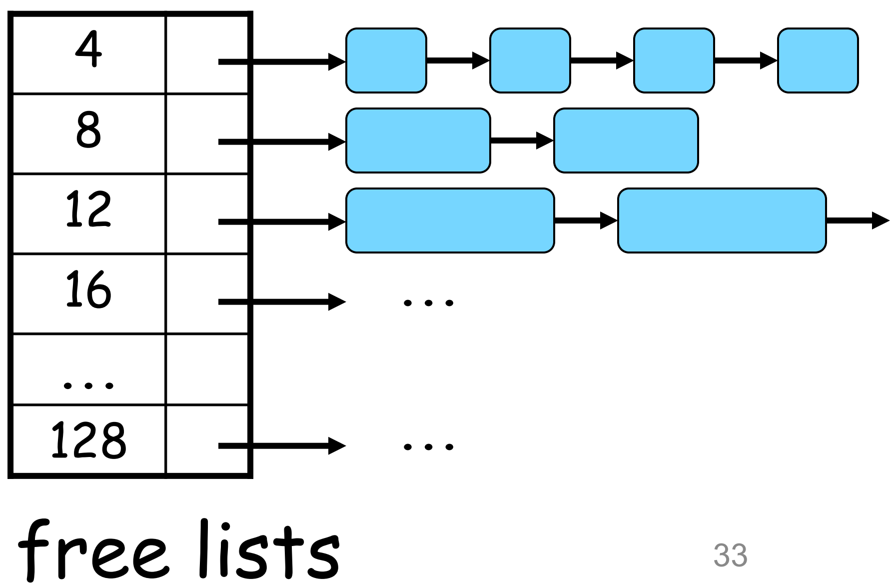


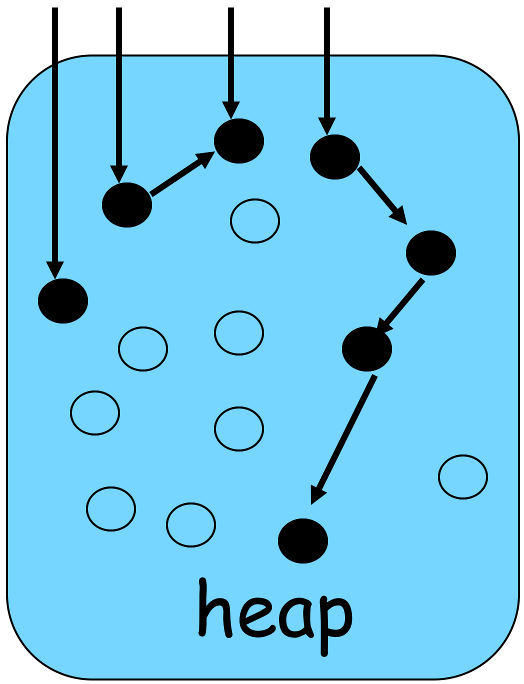
free lists



Mark-Sweep

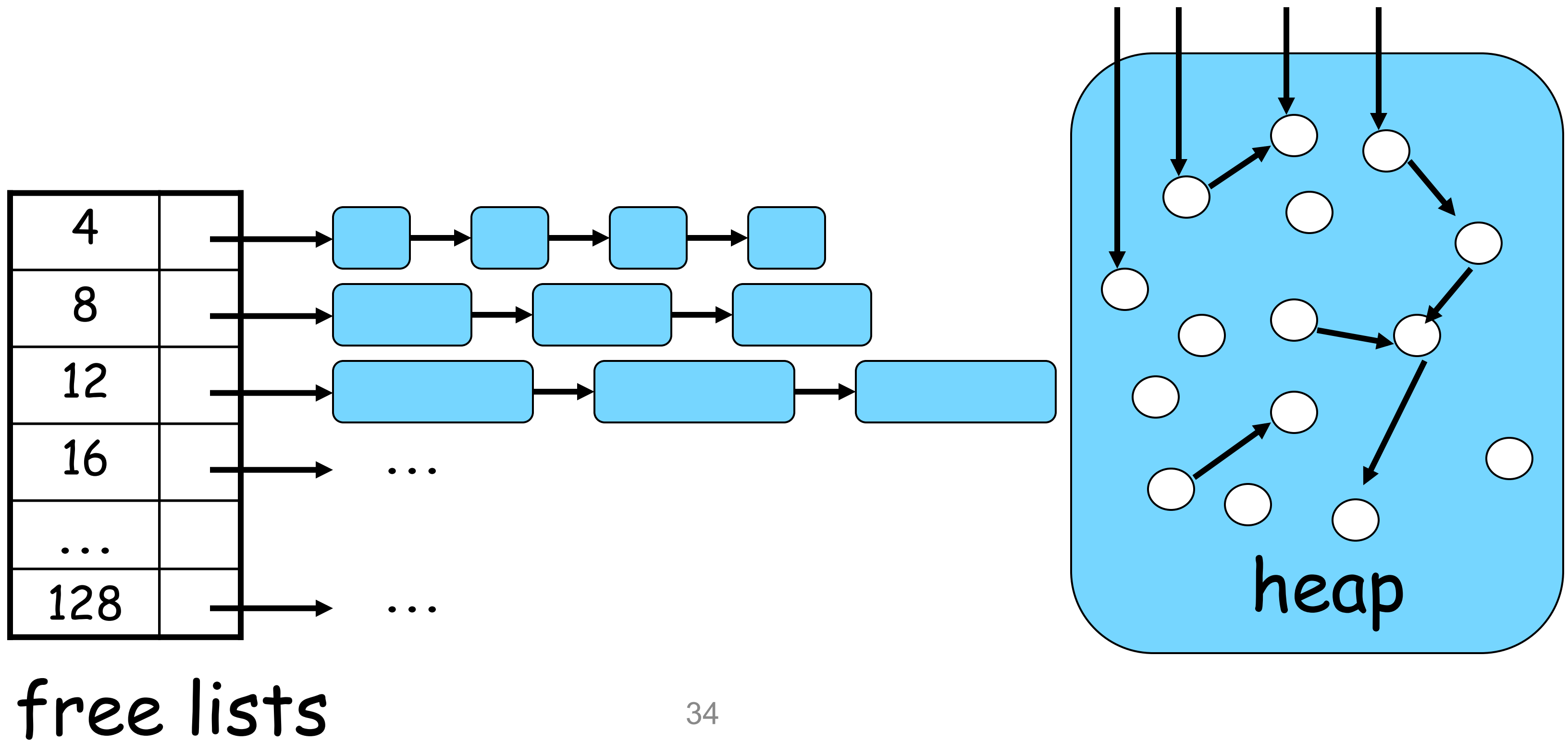
- Mark phase
  - Transitive closure marking all the unreachable objects
- Sweep phase
  - sweep the memory for unreachable objects populating free list
  - can be made incremental by organizing the heap in blocks and sweeping one block at a time on demand





Mark-Sweep

- ✔ space efficiency
- ✔ Incremental object reclamation
- ✘ relatively slower allocation time
- ✘ poor locality of contemporaneously allocated objects

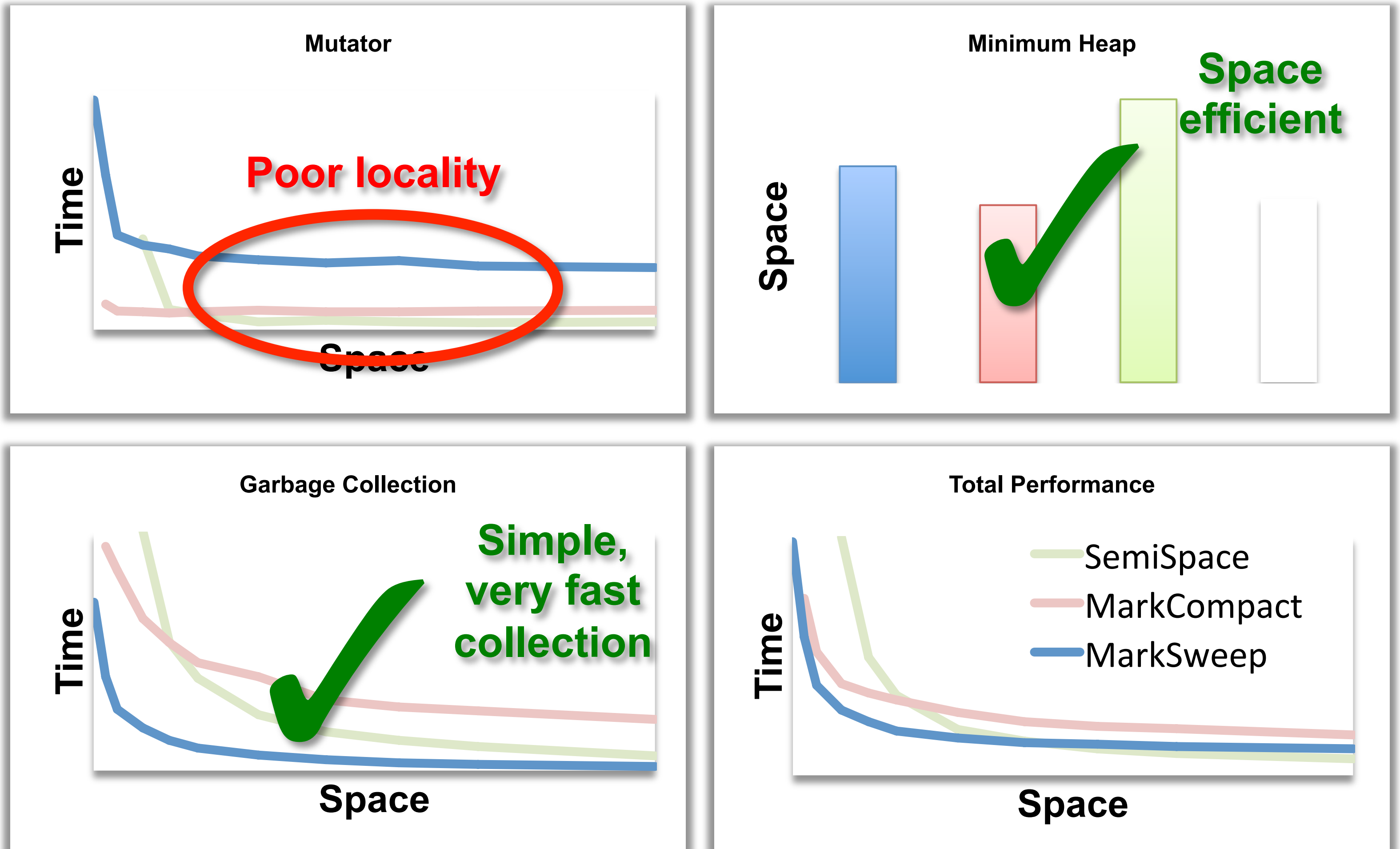


Mark-Sweep: Text Description

- Free lists organized by size (binning!)
  - blocks of same size or
  - individual objects of same size
- Most objects are small < 128 bytes
- Free List Setup:
  - have free lists for sizes 4, 8, 12, 16, … 128
  - each list has x number of free chunks of memory that it tracks
    - each chunk resides somewhere in the heap
- Allocation
  - grab a pointer to free space off the free list
  - first object is allocated on the heap
    - more objects are allocated on the heap and referenced by this first object
  - no more memory of the right size triggers a collection
- Mark phase: find the reachable objects
  - transitive closure marking all the live objects
- Sweep phase
  - sweep the memory for unreachable objects populating free lists
    - free lists grow again during this phase
  - can be made in incremental by organizing the heap in blocks and sweeping one block at a time on demand
- Result:
  - Good space efficiency
  - Incremental object reclamation possible
  - Relatively slow allocation time
  - Poor locality of contemporaneously allocated objects
    - contemporaneous means allocated close to each other in time

Mark-Sweep Evaluation

Free List Allocation + Trace + Sweep-to-Free



Actual data, taken from geomean of DaCapo, jvm98, and jbb2000 on 2.4GHz Core 2 Duo

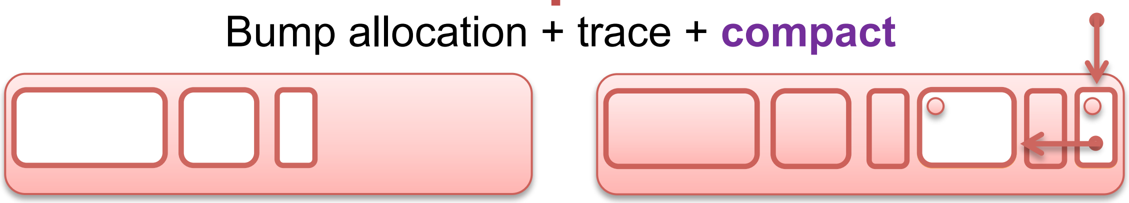
Mark-Sweep Evaluation: Text Description

- Allocator has a slow allocation time as opposed to other methods
- Algorithm is space efficient
- Allocation and recycled space method results in poor locality
- Garbage collection method is simple and very fast
- Total performance is decent

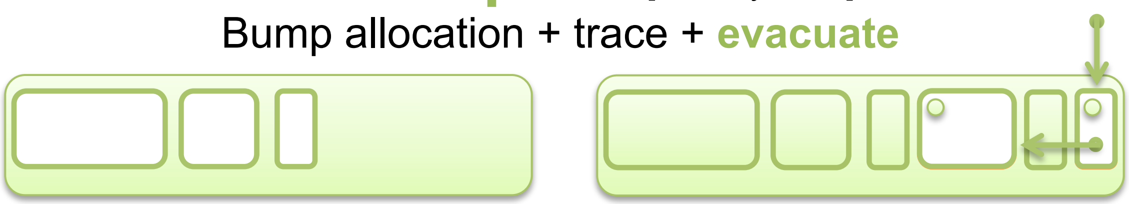
Garbage Collectors

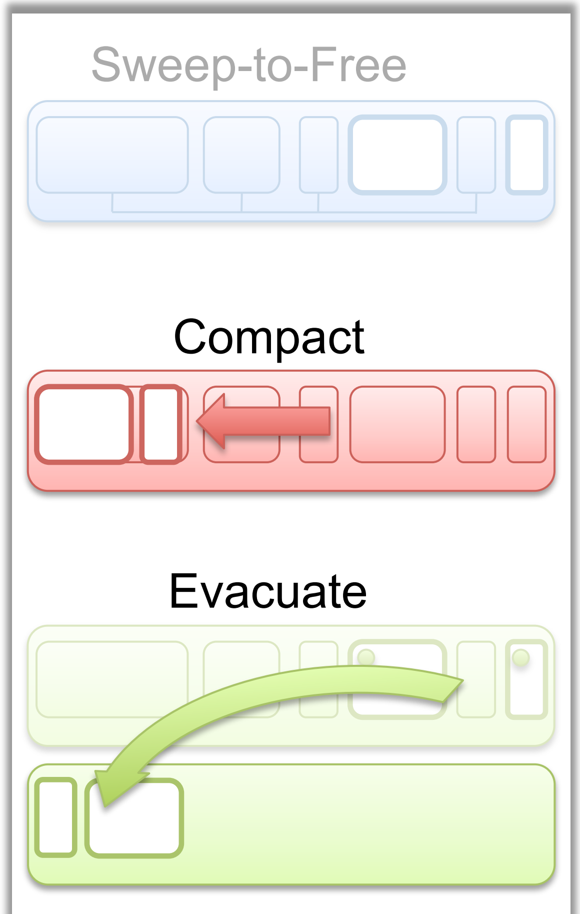
Mark-Sweep [McCarthy 1960]
Free-list + trace + **sweep-to-free**

Mark-Compact [Styger 1967]
Bump allocation + trace + **compact**



Semi-Space [Cheney 1970]
Bump allocation + trace + **evacuate**



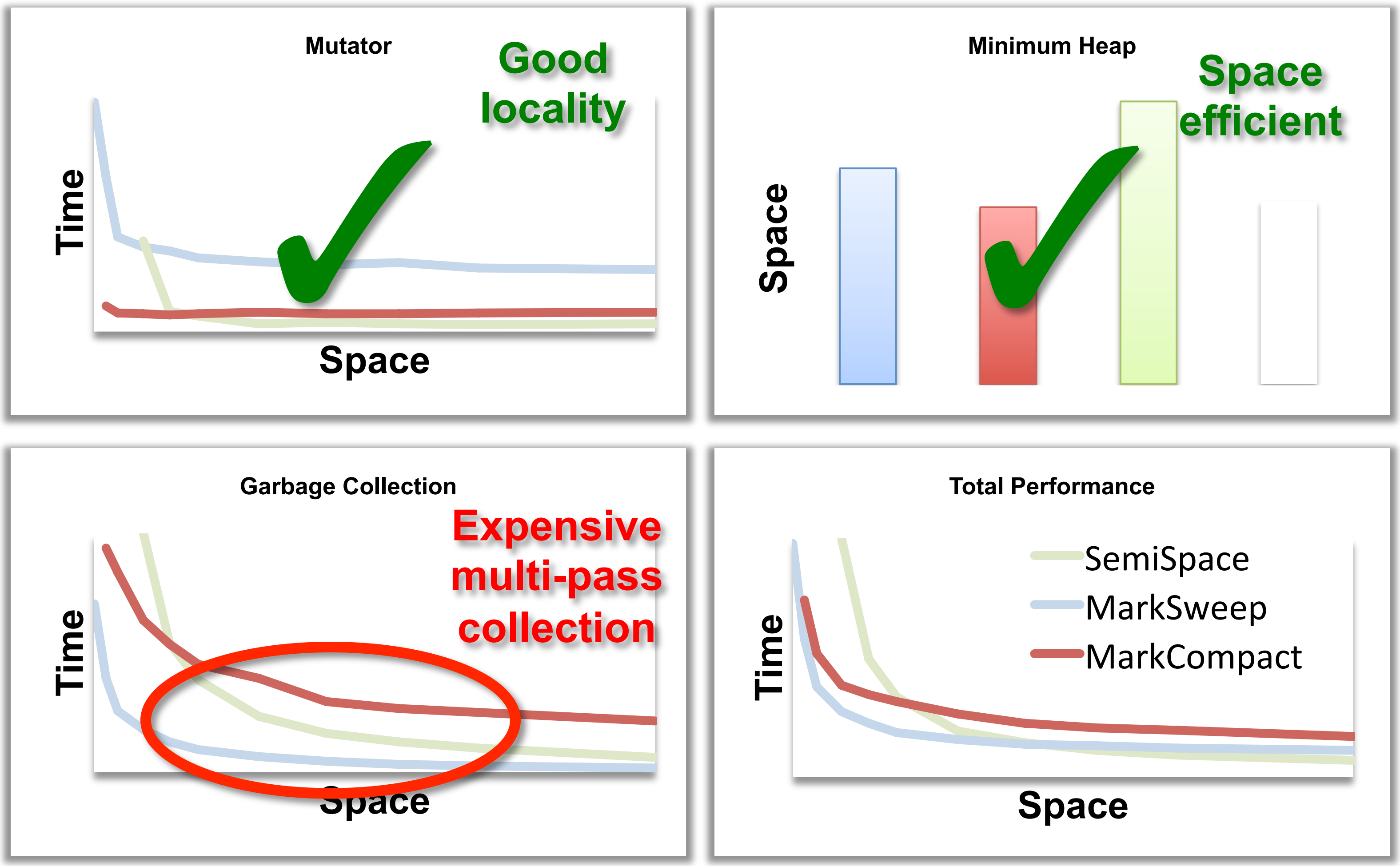


Garbage Collectors: Text Description

- This slide is the same as the one described in 24, but the emphasis is shifted to mark-compact
- Mark-Sweep - McCarthy 1960
  - free-list + trace + sweep-to-free
- Mark-Compact - Styger 1967
  - Bump pointer allocation + trace + compact
- Semi-Space - Cheney 1970
  - Bump pointer allocation + trace + evacuate
- Focus of this slide is Mark-Compact
  - Mark-compact is very similar to mark-sweep
    - One additional reclamation step: copy all objects remaining in the heap to one end of the heap (compact)
    - That additional reclamation step allows mark-compact to use bump pointer allocation (rather than free-list)
  - There are no further slides describing mark-compact

Mark-Compact Evaluation

Bump Allocation + Trace + Compact

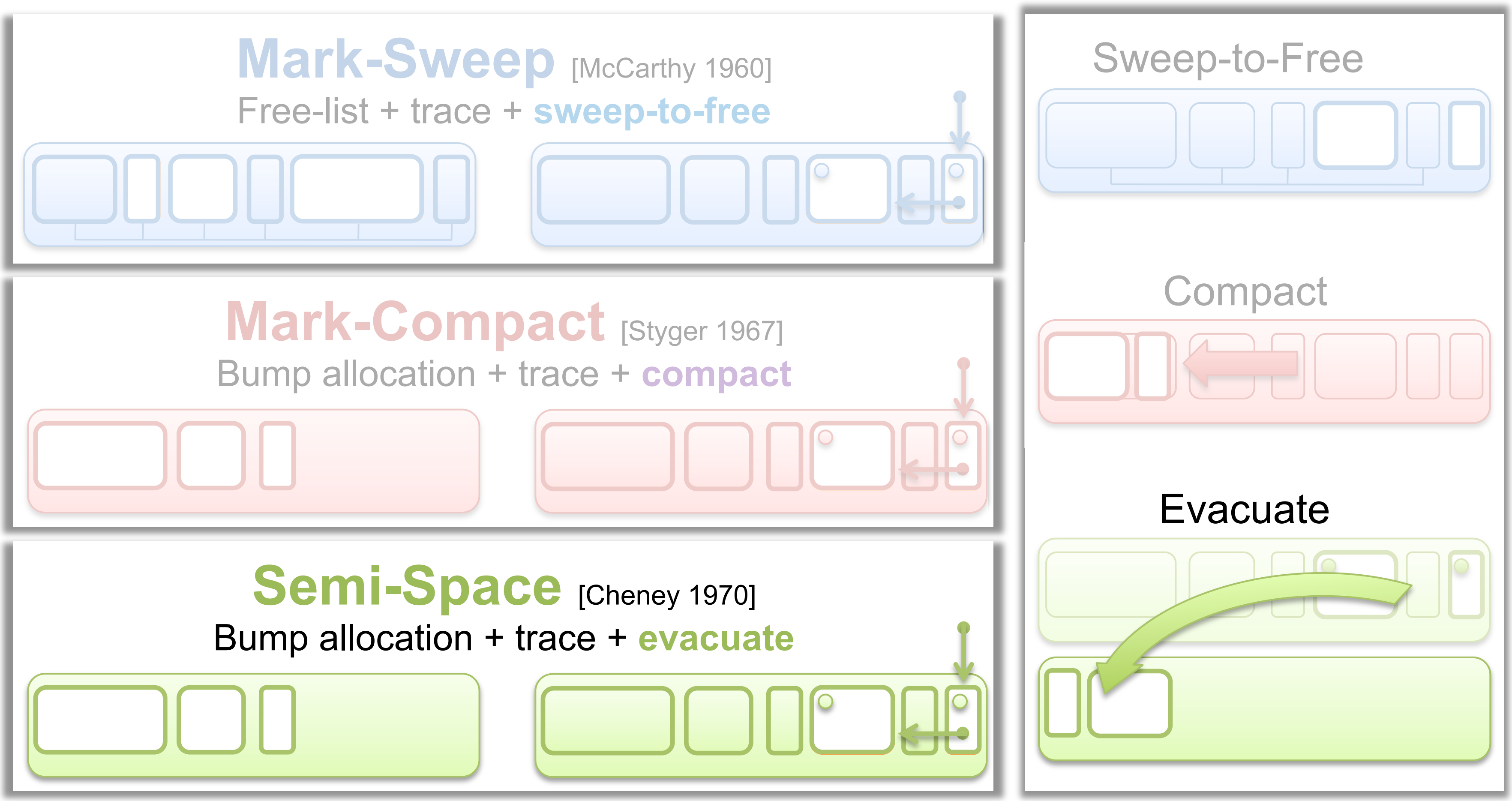


Actual data, taken from geomean of DaCapo, jvm98, and jbb2000 on 2.4GHz Core 2 Duo

Mark-Compact Evaluation: Text Description

- Allocator is fast and exhibits good locality
  - Better than mark-sweep
- Garbage collector has expensive multi-pass collection
  - Slower than mark-sweep
- Algorithm is space efficient
- Total performance is worse than that of mark-sweep

Garbage Collectors

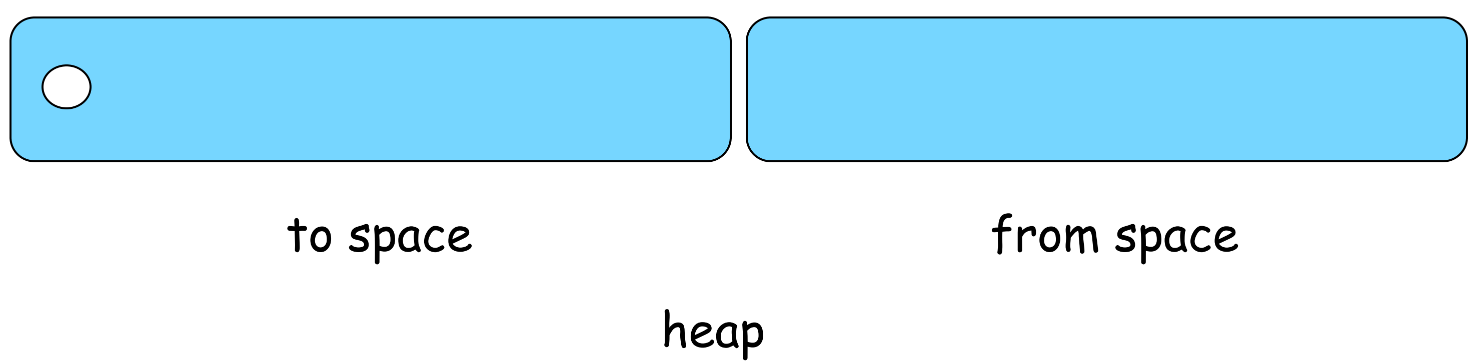


Garbage Collectors: Text Description

- This slide is the same as the one described in 24 and 39, but the emphasis is shifted to semi-space
- Mark-Sweep - McCarthy 1960
  - free-list + trace + sweep-to-free
- Mark-Compact - Styger 1967
  - Bump allocation + trace + compact
- Semi-Space - Cheney 1970
  - Bump allocation + trace + evacuate
- Focus of this slide is Semi-Space, which is described in the upcoming slides.

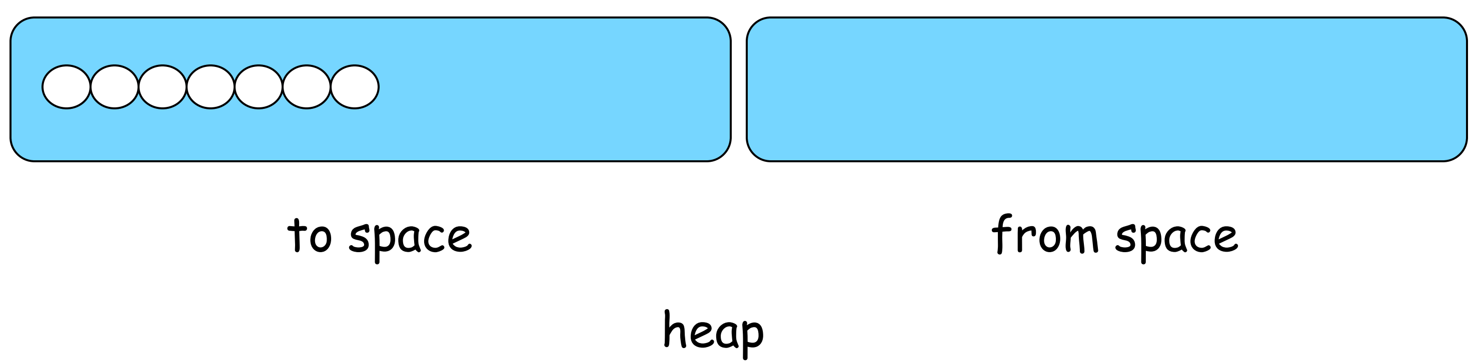
Semi-Space

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves half of the heap to copy into, in case all objects are reachable



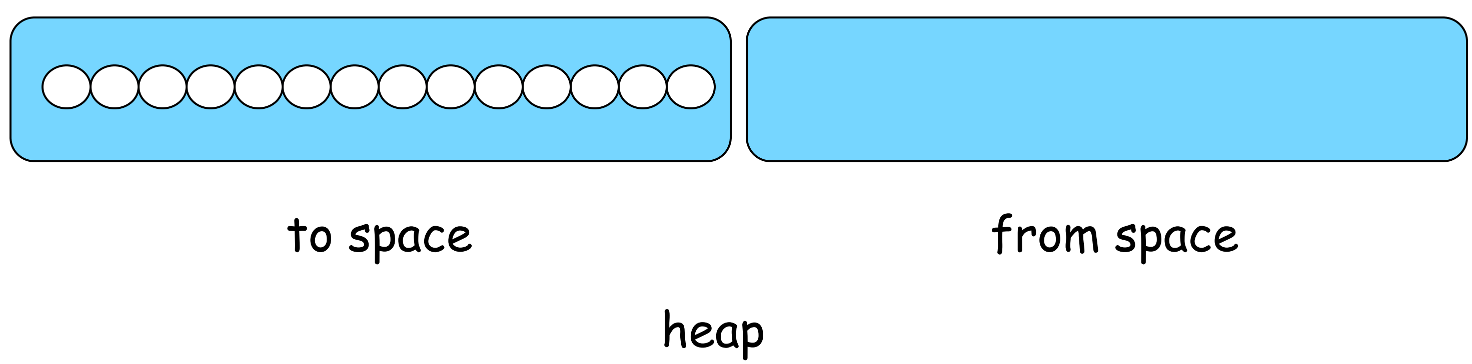
Semi-Space

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves half of the heap to copy into, in case all objects are reachable



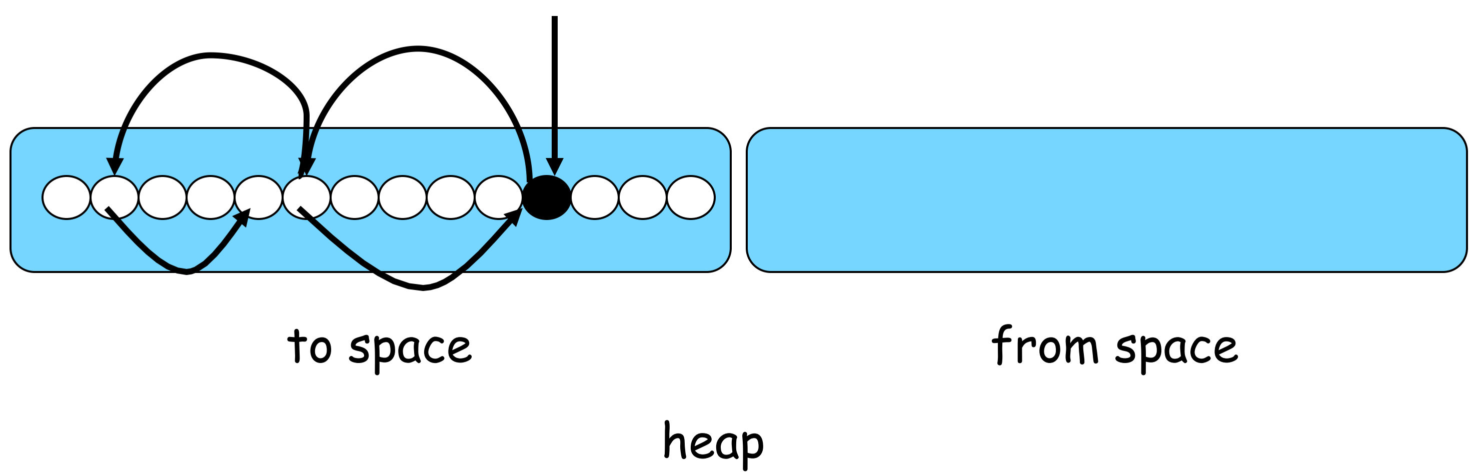
Semi-Space

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves half of the heap to copy into, in case all objects are reachable



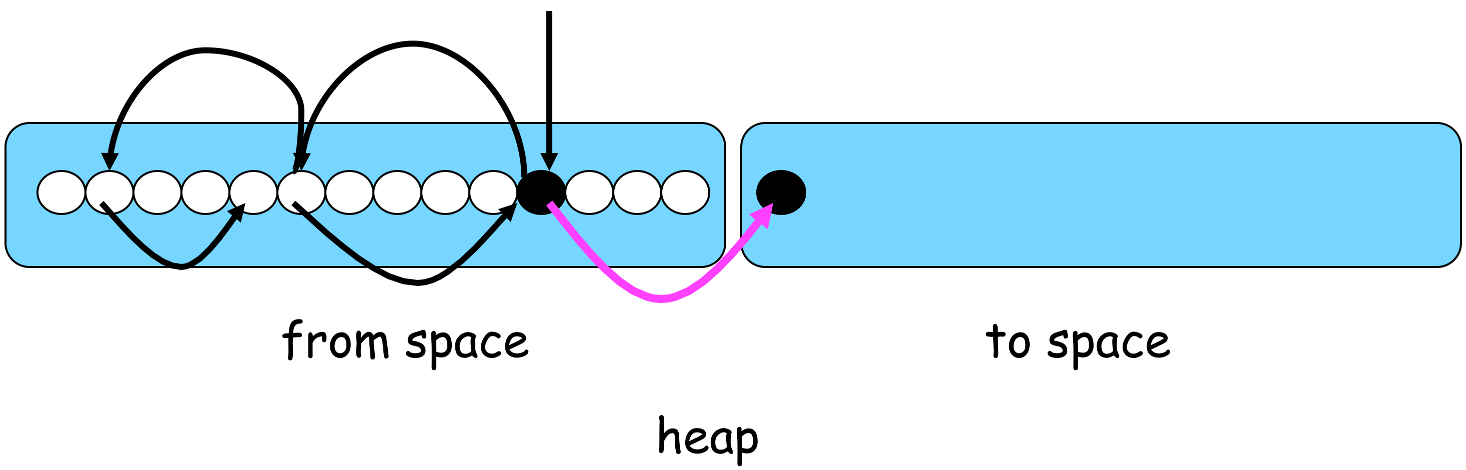
Semi-Space

- Fast **bump pointer** allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves half of the heap to copy into, in case all objects are reachable



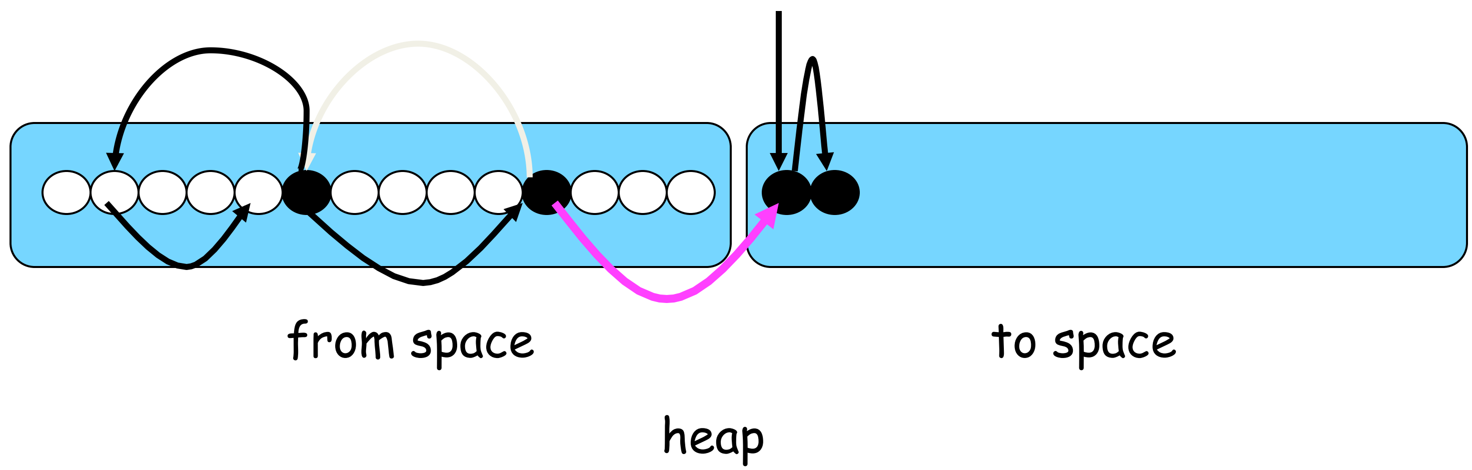
Semi-Space

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**



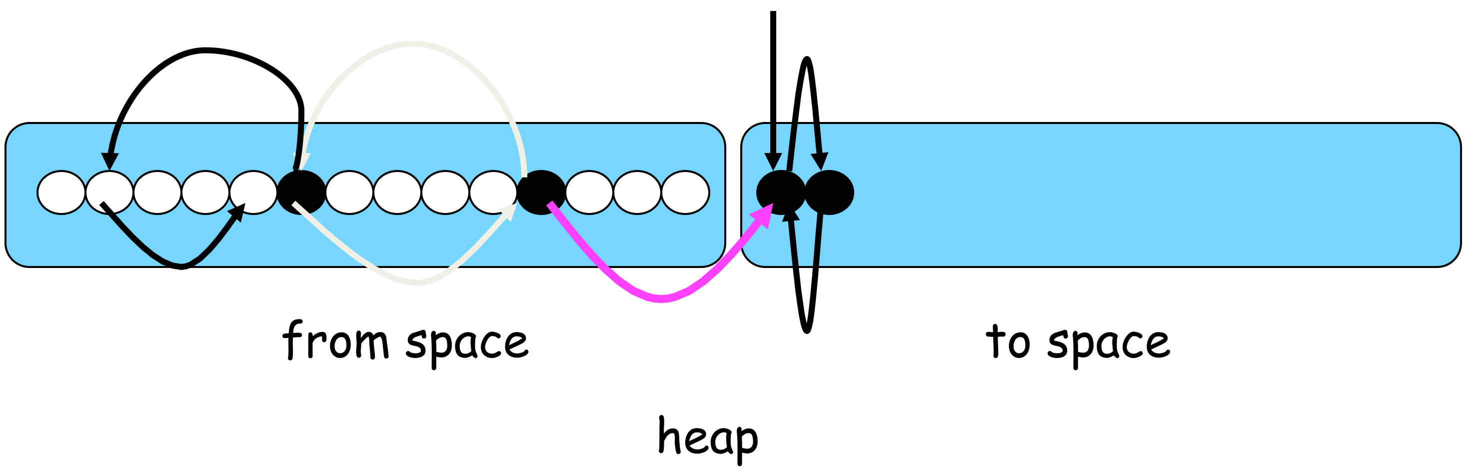
Semi-Space

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes



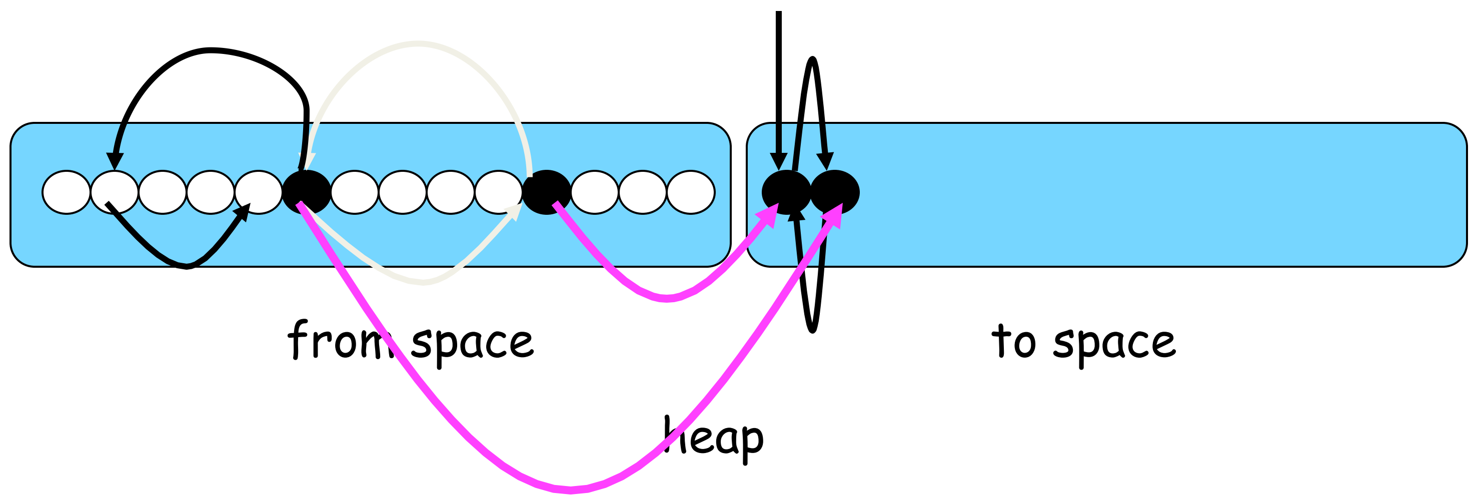
Semi-Space

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes



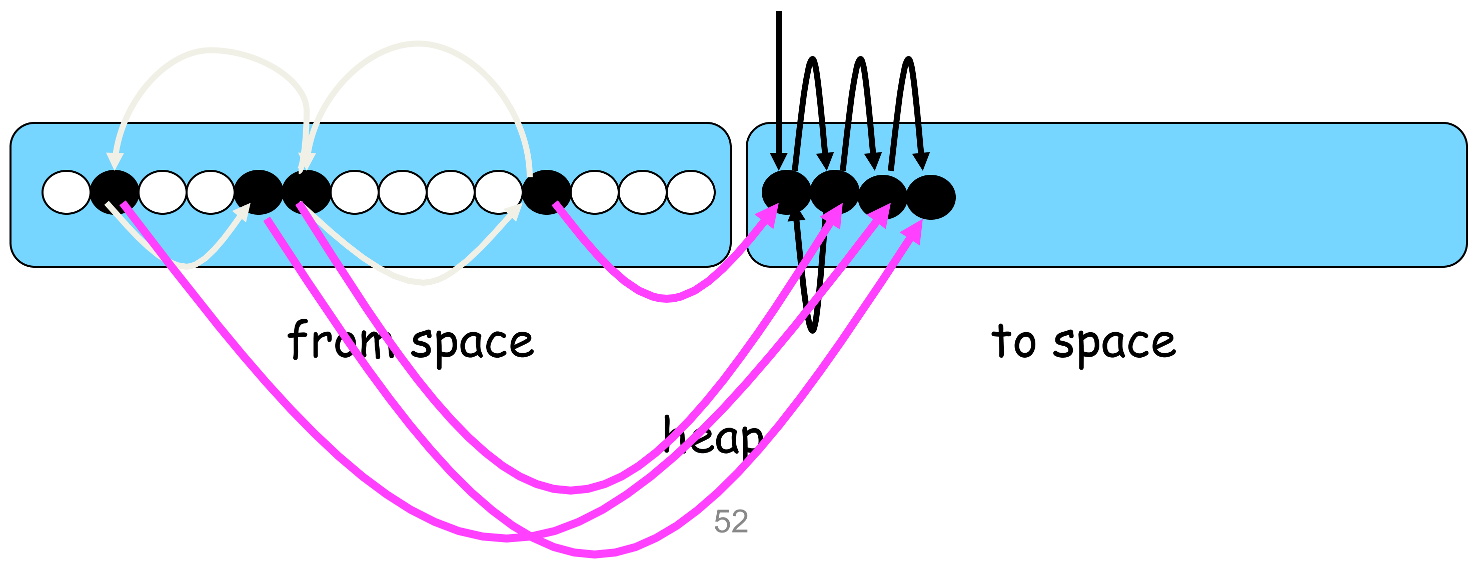
Semi-Space

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes



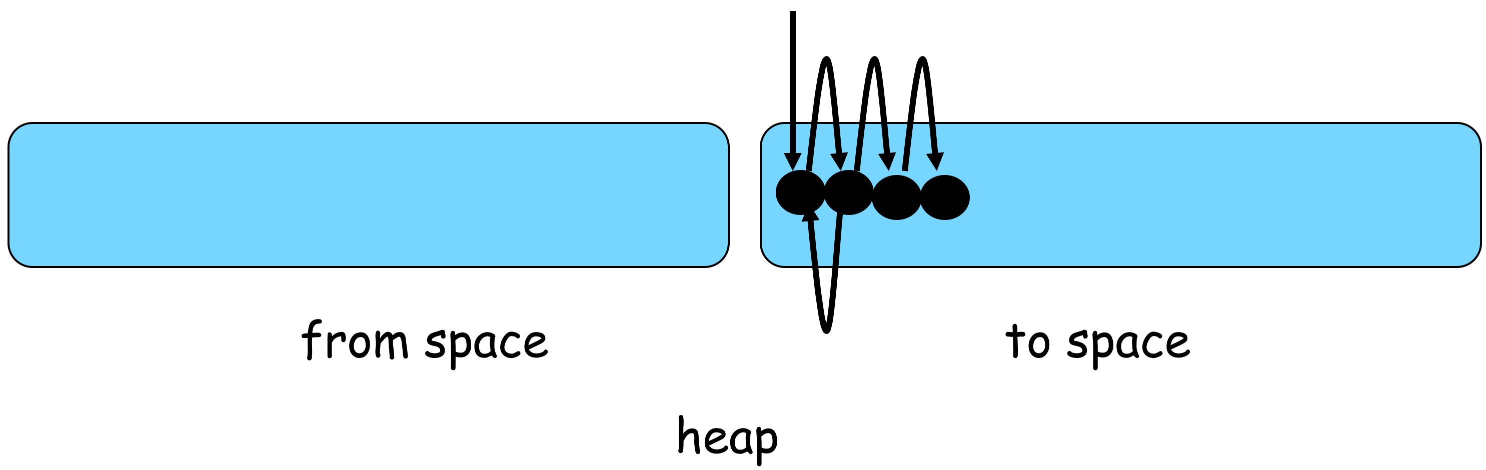
Semi-Space

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes
  - reclaims "from space" en masse



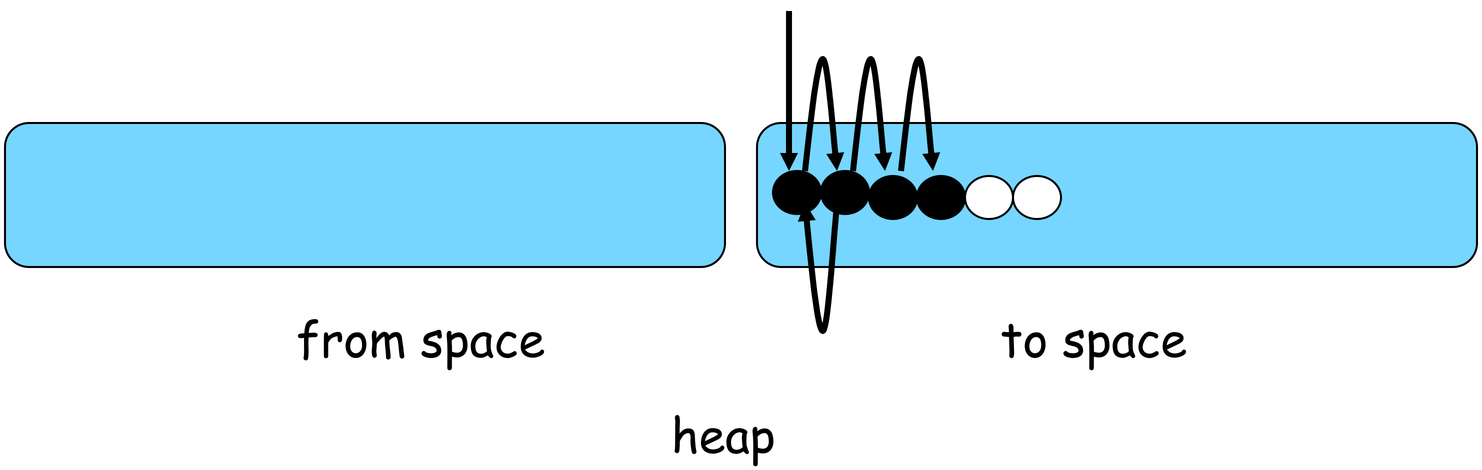
Semi-Space

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes
  - reclaims “from space” en masse
  - start allocating again into “to space”



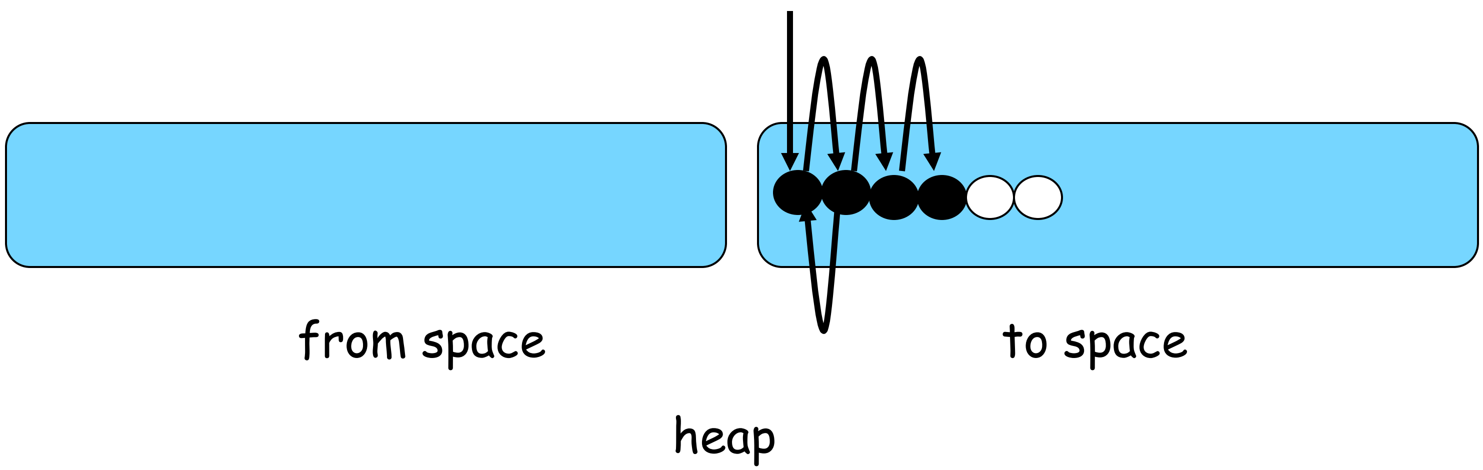
Semi-Space

- Mark phase:
  - copies object when collector first encounters it
  - installs **forwarding pointers**
  - performs transitive closure, updating pointers as it goes
  - reclaims “from space” en masse
  - start allocating again into “to space”



Semi-Space

- Notice:
  - ✔ fast allocation
  - ✔ locality of contemporaneously allocated objects
  - ✔ locality of objects connected by pointers
  - ✘ wasted space

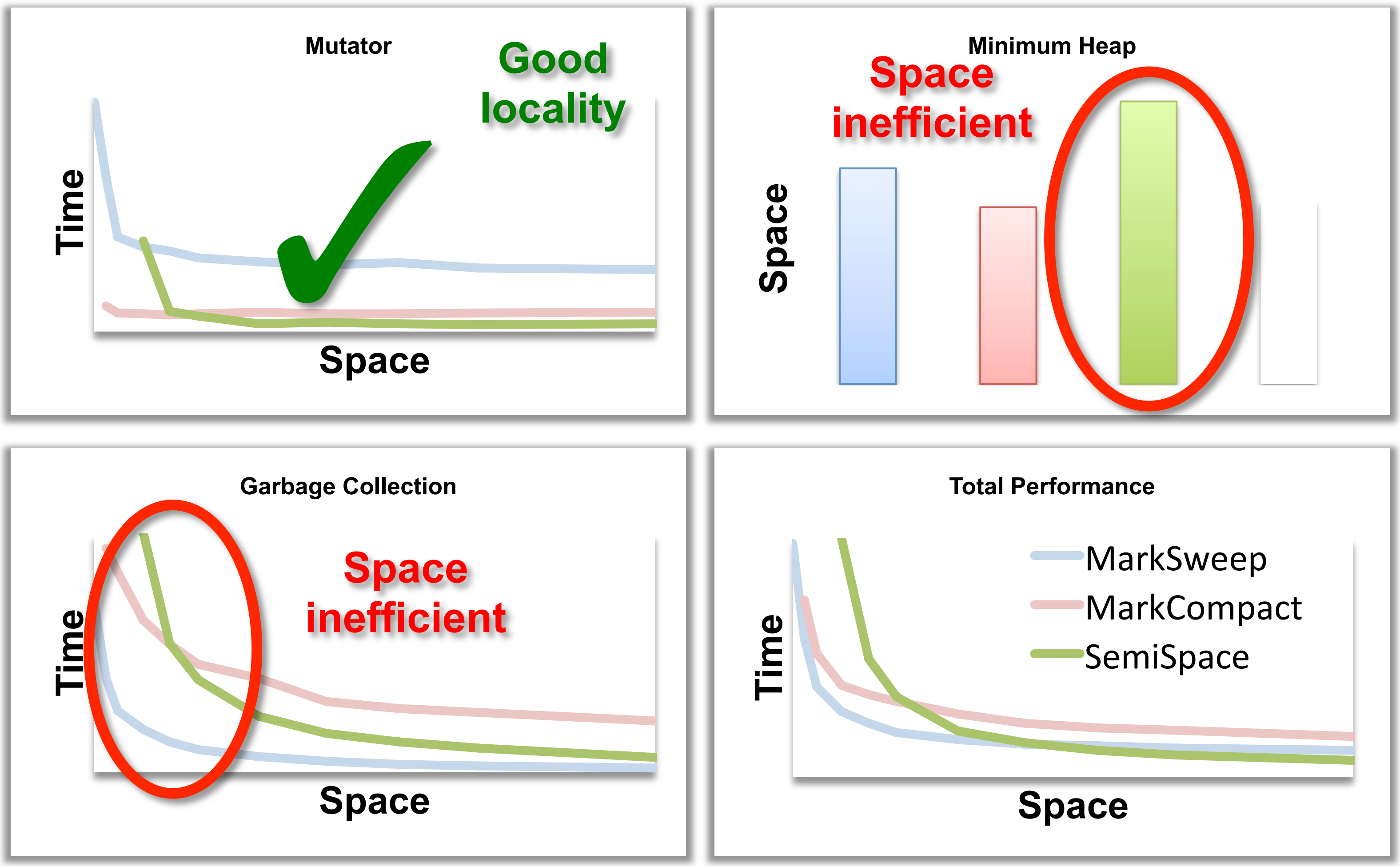


Semi-Space
Text Description

- Fast bump pointer allocation
- Reserves half the heap to copy into, in case all objects are reachable
  - 2 parts of heap: “to space” and “from space”
- Originally objects are allocated into the “to space”
  - Once the “to space” has been fully allocated the trace algorithm is run
  - The “to space” becomes the “from space” as reachable objects are copied into the other part of the heap
- Mark Phase
  - Copies object when collection first encounters it
    - object is copied into the “to space” in the other half of the heap
  - Installs forwarding pointers
    - Forwarding pointers tell where the object is now in case another pointer still points to the old spot
  - Performs transitive closures, updating pointers as it goes
    - so “to space” will end up with only reachable objects in it
  - Collects “from space” en masse
    - So once all the reachable objects have been moved over the other half of the heap is just wiped clean
  - Start allocating again into “to space”
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Result:
  - Awful space efficiency
  - Fast allocation time
  - Good locality of contemporaneously allocated objects
  - Good locality of objects connected by pointers

Semi-Space Evaluation

Bump Allocation + Trace + Evacuate



Actual data, taken from geomean of DaCapo, jvm98, and jbb2000 on 2.4GHz Core 2 Duo

Semi-Space Evaluation: Text Description

- Allocator is fast and has good locality
  - Its speed is the same as Mark-Compact (both use bump pointer)
  - Its locality is the best out of the 3
    - Copying during the trace causes objects that are related to be copied near each other
- Garbage collector is faster than mark-compact
  - Collection done in a single pass
- Algorithm is space inefficient
  - Can only use half the heap at a time
- Total performance is better than the other algorithms for large heap sizes

iClicker Question

True or false: older objects are more likely to survive garbage collection.

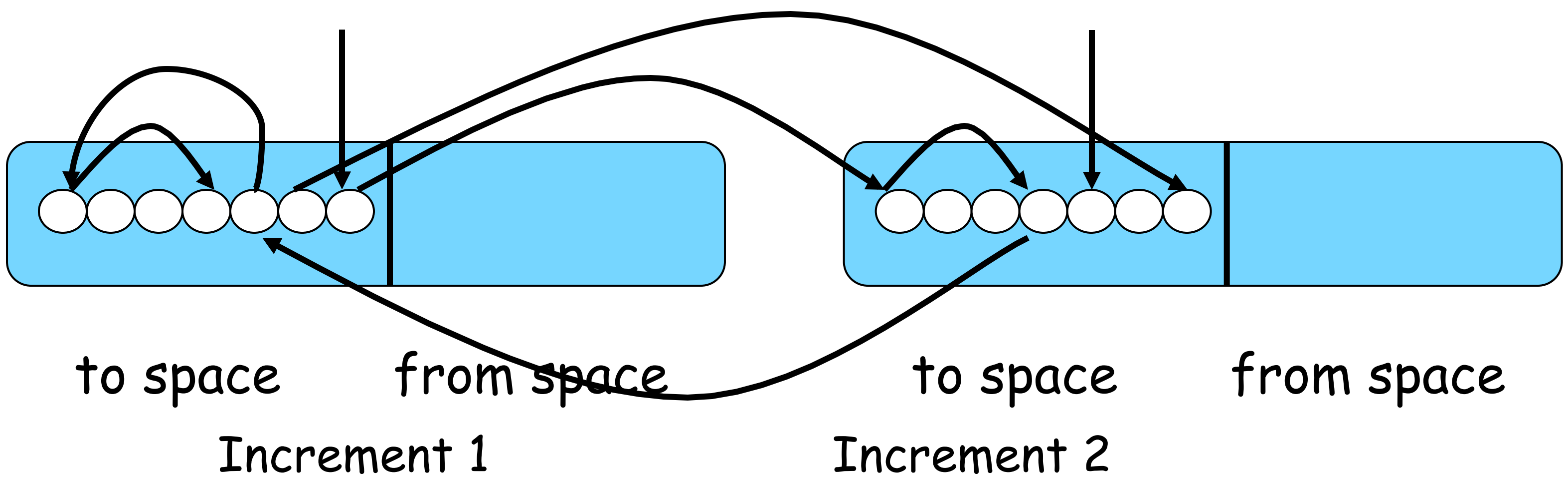
A. True
B. False

One Big Heap?

Pause time
- it takes too long to trace the whole heap at once

Throughput
- the heap contains lots of long lived objects, why collect them over and over again?

Incremental collection
- divide up the heap into increments and collect one at a time.

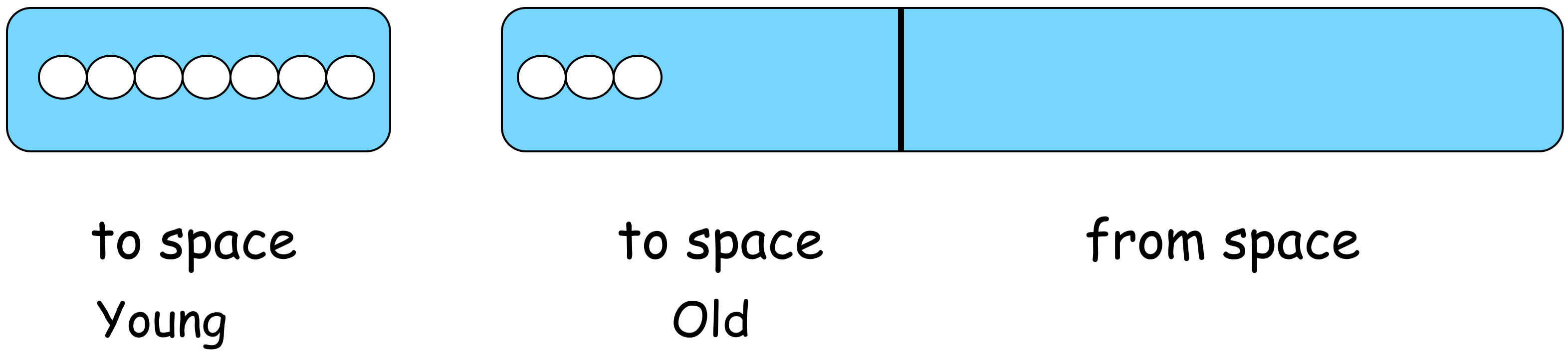


Heap Organization

What objects should we put where?

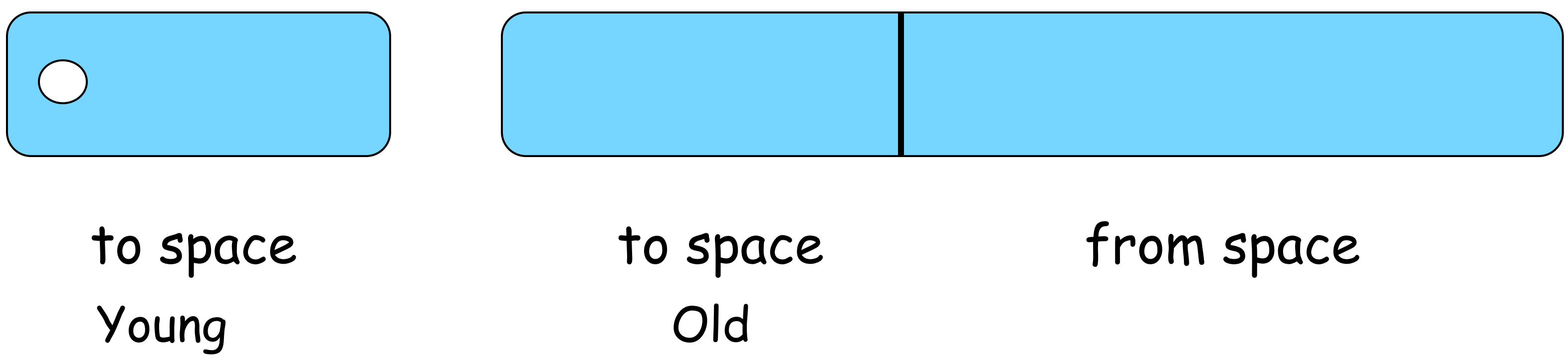
- **Generational hypothesis**
  - **young objects die more quickly than older ones** (or, most objects have short lifetimes) [Lieberman & Hewitt'83, Ungar'84]

➜ Organize the heap into young and old, collect young objects preferentially



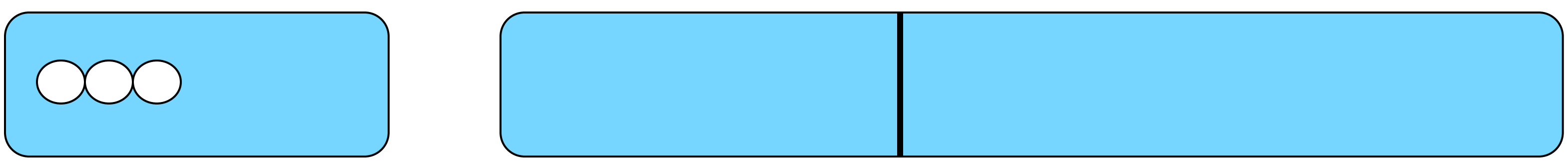
Generational Heap Organization

- **Divide the heap in to two spaces: young and old**
- Allocate into the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



Generational Heap Organization

- Divide the heap in to two spaces: young and old
- **Allocate in to the young space**
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



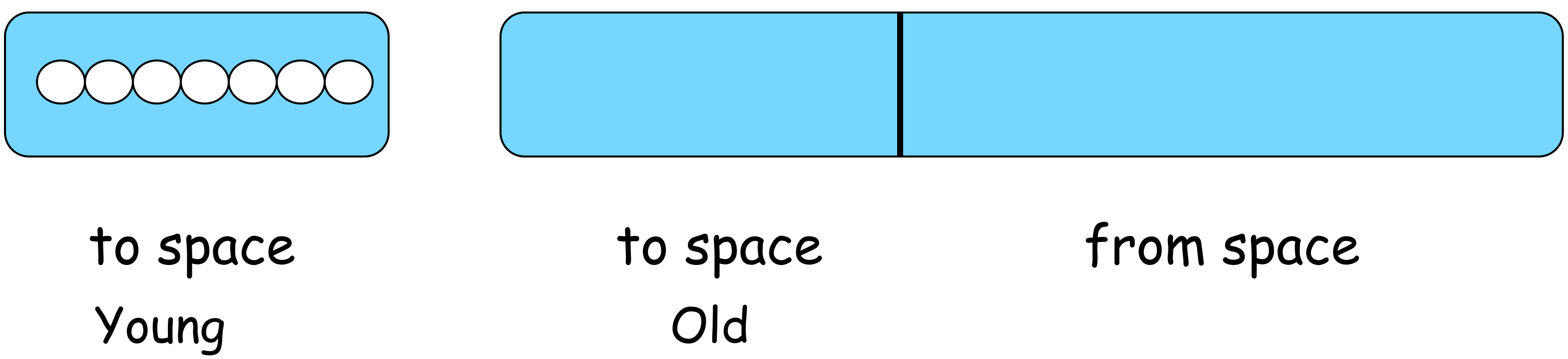
to space
Young

to space
Old

from space

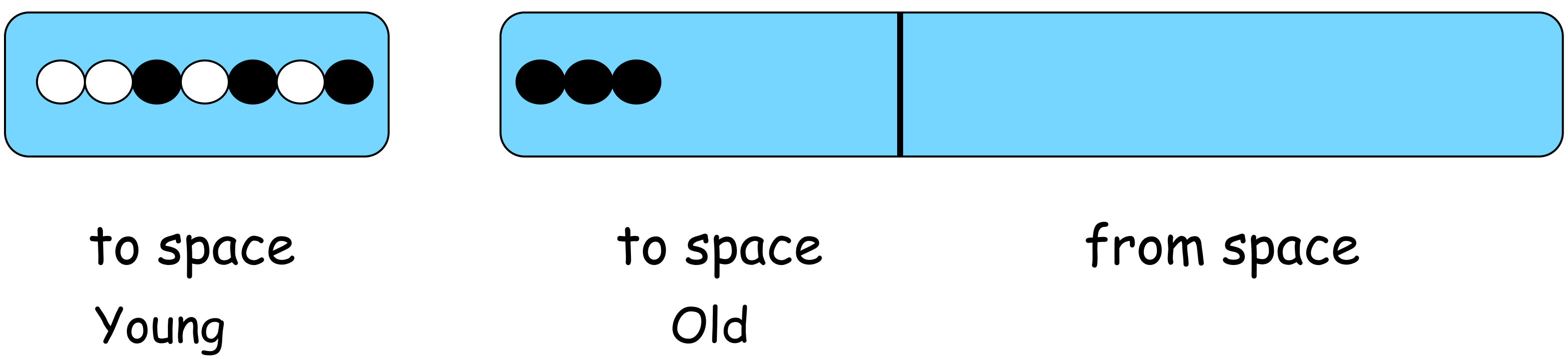
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



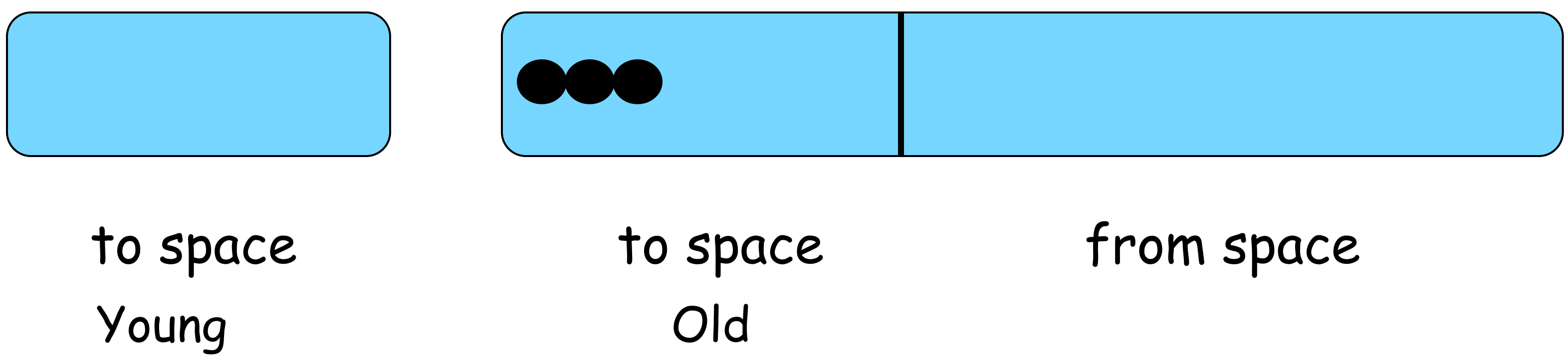
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



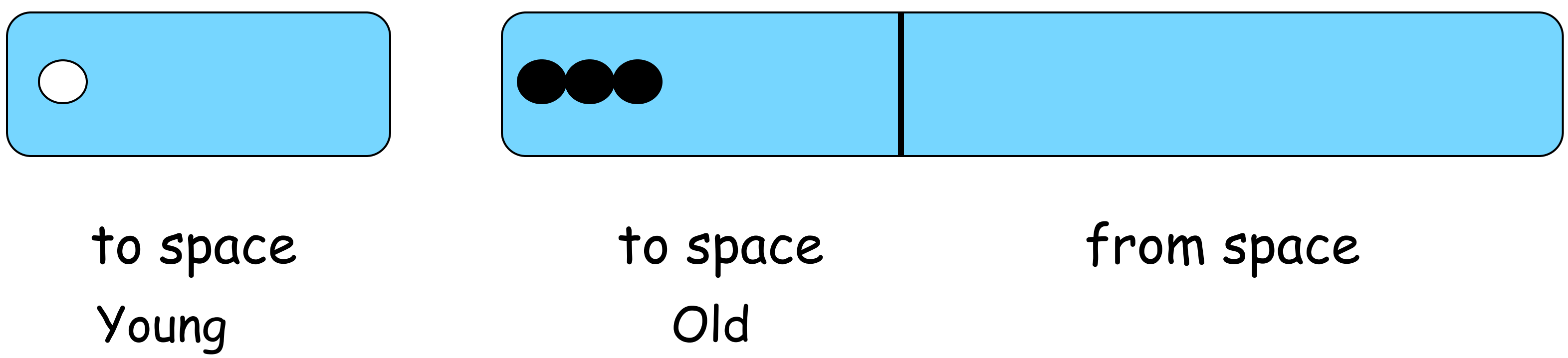
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



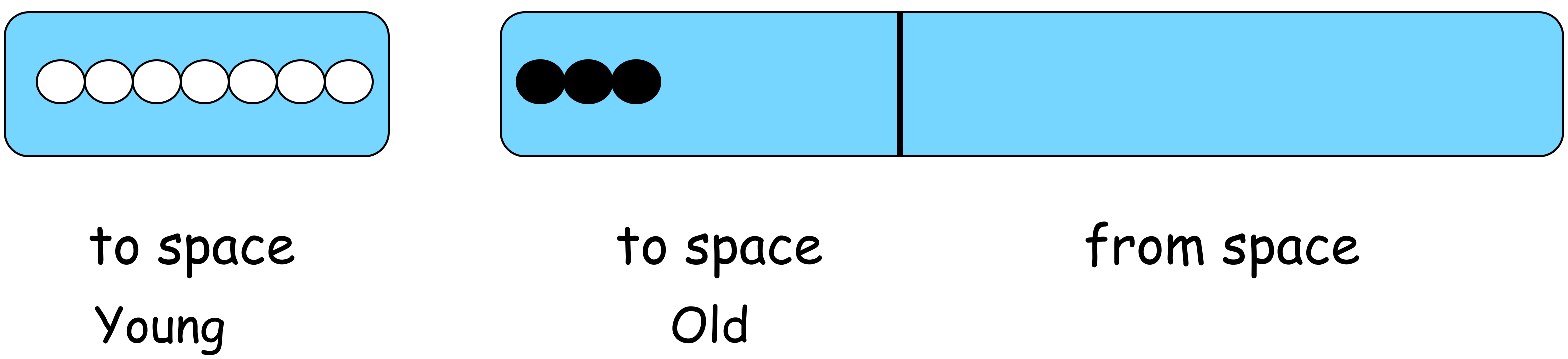
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- **Allocate in to the young space**
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



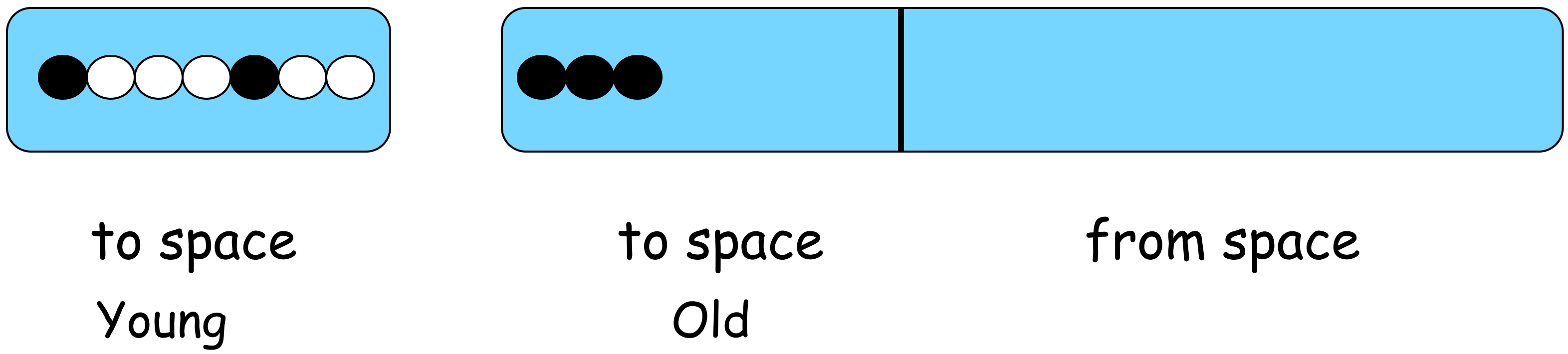
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



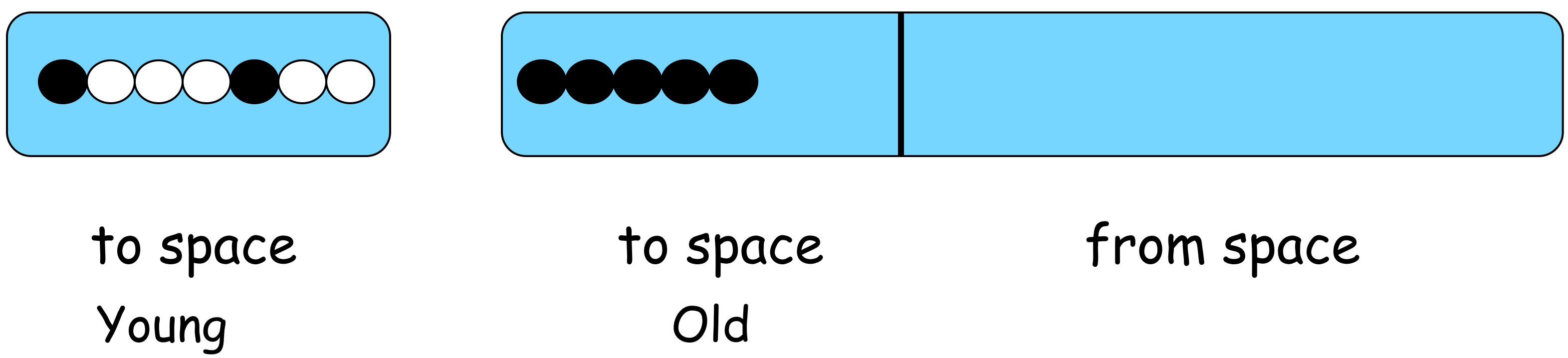
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



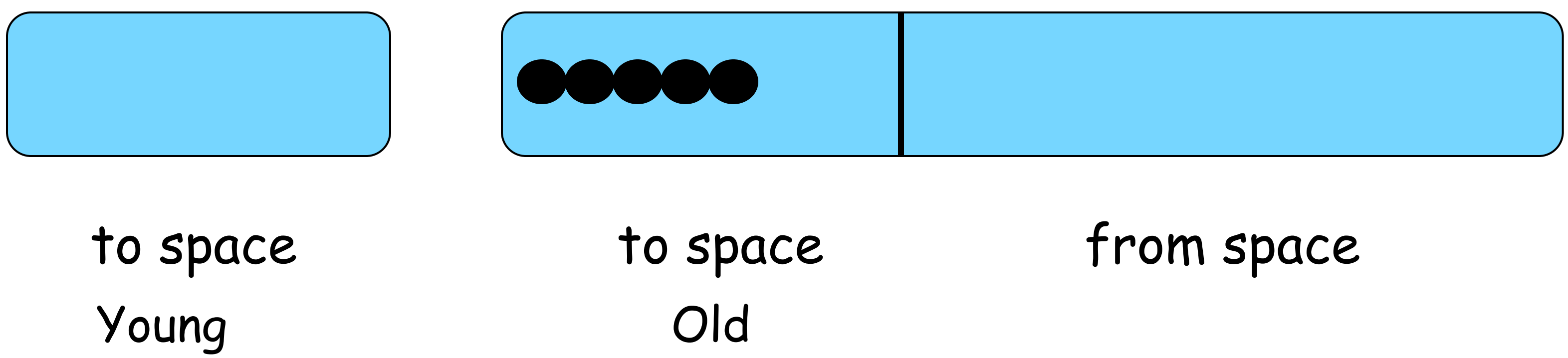
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



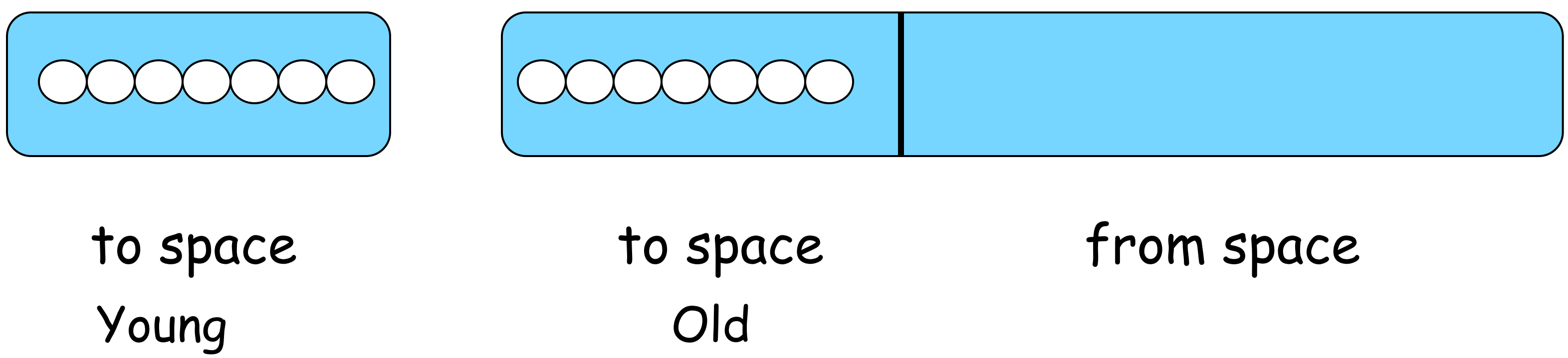
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- **When the young space fills up,**
  - **collect it, copying into the old space**
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



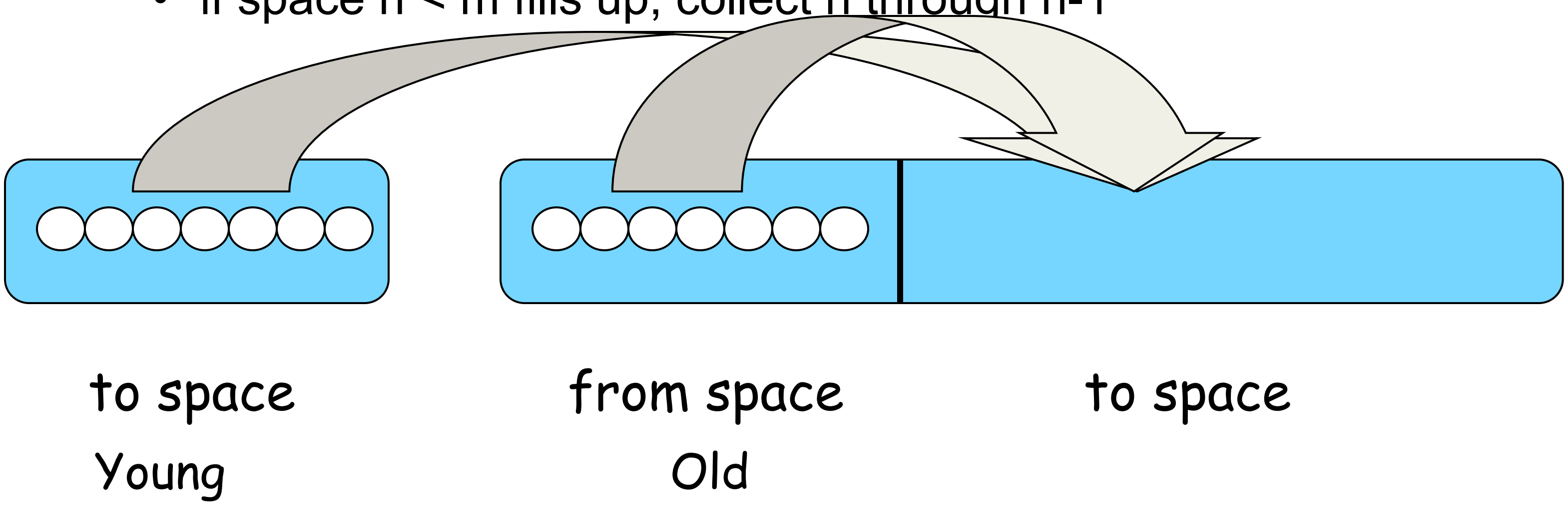
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- **When the old space fills up**
  - **collect both spaces**
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



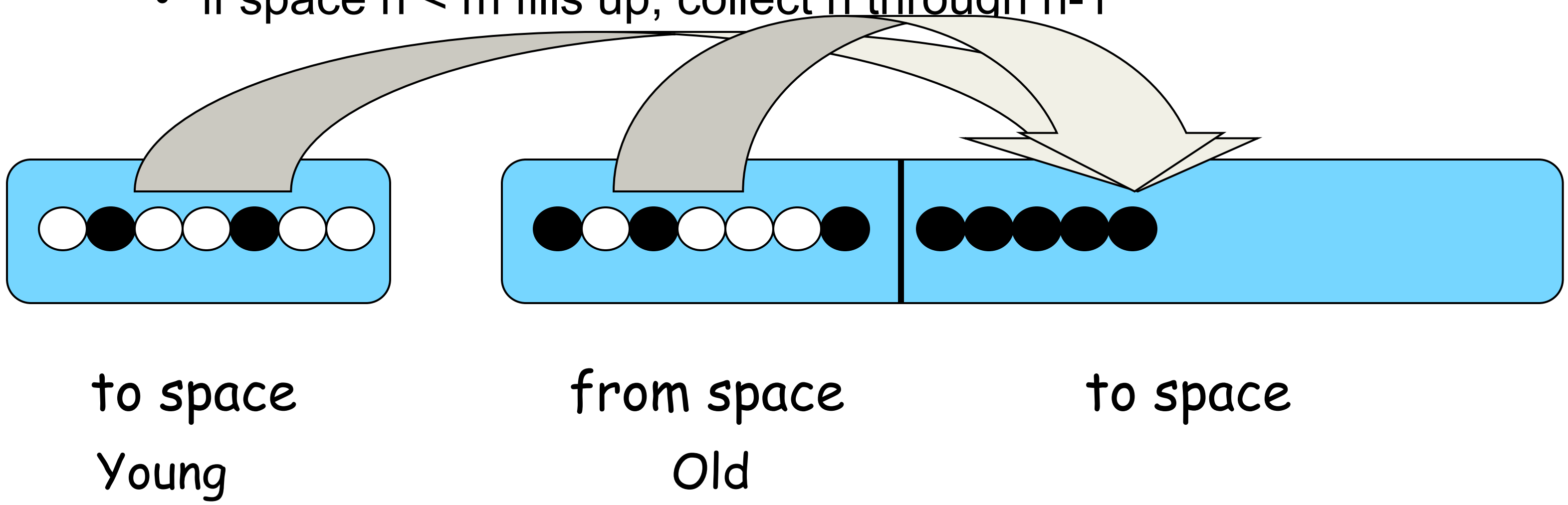
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- **When the old space fills up**
  - **collect both spaces**
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



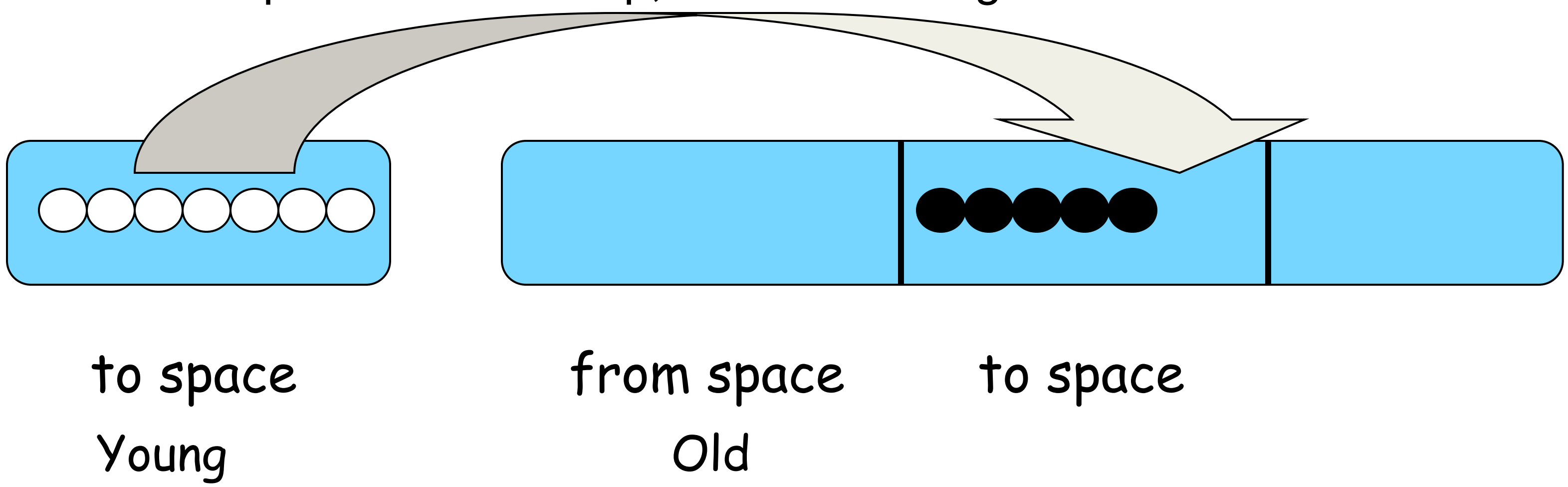
Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- **When the old space fills up**
  - **collect both spaces - ignore remembered sets**
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1



Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- **When the old space fills up**
  - **collect both spaces - ignore remembered sets**
  - Generalizing to m generations
    - if space n < m fills up, collect 1 through n-1



Generational Heap Organization: Text Description

- Divide the heap into two spaces: young and old
- Allocate into the young space
- When young space fills up
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
    - move still live objects into new “to space”
  - generalizing to m generations
    - if space n < m fills up, collect n through n-1

Garbage Collectors: Taxonomy of Design Choices

- Incrementality
  - Bounded tracing time
  - Conservative assumption: All objects in rest of heap are live
  - *Remember* pointers from rest of heap: Add *‘remembered set’* to roots for tracing
- Composability
  - Creation of hybrid collectors
- Concurrency
  - ‘Mutator’ and GC operate concurrently
- Parallelism
  - Concurrency among multiple GC threads
- Distribution
  - Distributed across multiple machines
  - Implies other four

Garbage Collection Recap

- Copying improves locality
- Incrementality improves responsiveness
- Generational hypothesis
  - Young objects: Most very short lived
    - Infant mortality: ~90% die young (within 4MB of allocation)
  - Old objects: most very long lived (bimodal)
    - Mature morality: ~5% die each 4MB of new allocation

Summary

- Discussed explicit and automatic memory management
  - Allocation policies (bump pointer, free list)
  - De-allocation policies (free, various collection algorithms)
  - Free-list management
- What principles did we learn about virtual memory management that we applied to heap memory management?
  - Programs have locality
  - Internal vs. external fragmentation
  - Keep paging to a minimum

Virtual Memory Review

Announcements

- Homework 7 due Friday 8:45a
- Project 2 due Friday 11:59p
- Project 3 is posted due Friday, 4/17
  - Project 2 must be working
    - Except multi-oom
    - No, we will not give you solutions
- Exam 2 in two weeks (Wednesday, 4/8)
  - UTC 2.122A 7p-9p