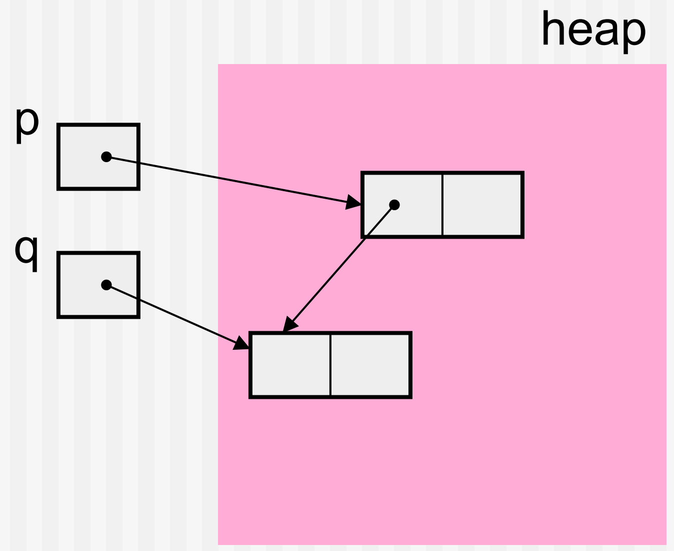
Garbage Collection

Taiichi Yuasa
Graduate School of Informatics
Kyoto University

Dynamic allocation of objects

- It is not possible to estimate the number of data objects required for execution.
- Allocate objects dynamically as required.

```
cell *p,*q;
p = (cell *) malloc(sizeof(cell));
q = (cell *) malloc(sizeof(cell));
p->left = q;
```



Disposal of dead objects

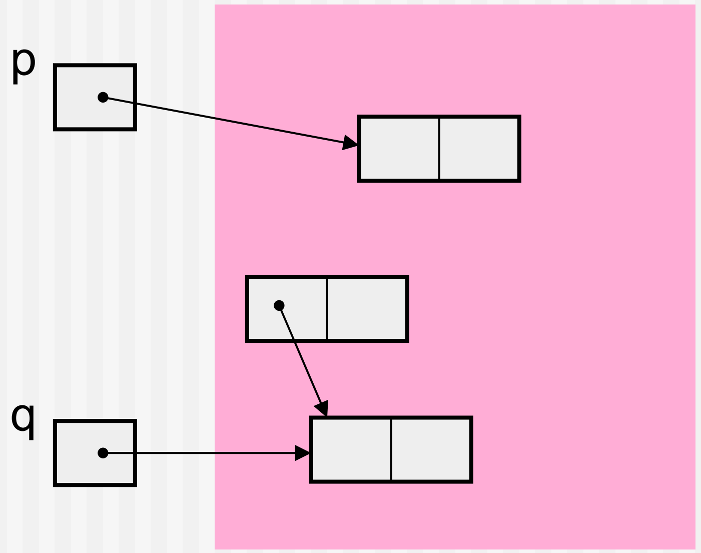
- Objects may become unnecessary during execution.

```
p->left = NULL;
```

- If not eliminated, memory leak occurs.
  ⇒ Free the memory occupied by dead objects to reuse the area.

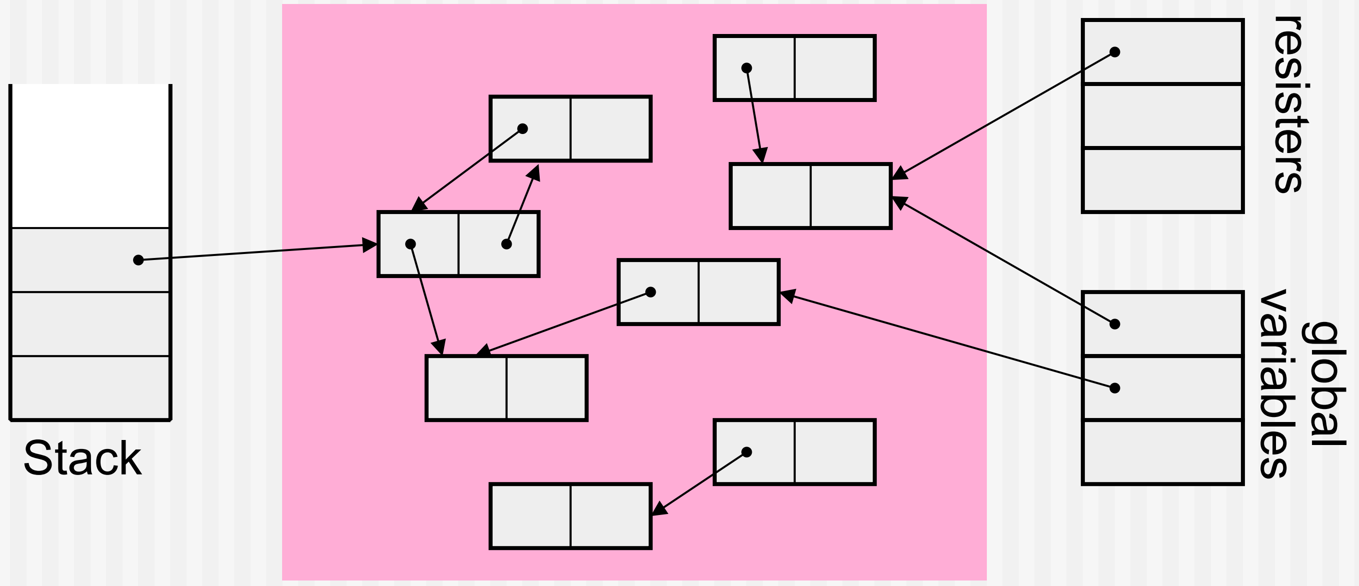
```
free(p->left);
```

- It is a difficult task.
  - forget to free ⇒ memory leak
  - actually in use ⇒ dangling pointer



(Automatic) garbage collection (GC)

- Automatically detect dead objects (garbage) to reuse the area.
  Lisp, Prolog, SmallTalk, ML, C++, Java, C#, ... Perl, Ruby, Python, ... COBOL, …
- Garbage: objects that will not be referred to.
  ≒ Objects that cannot be reached from the roots
- Root: a location which the program can directly refer to



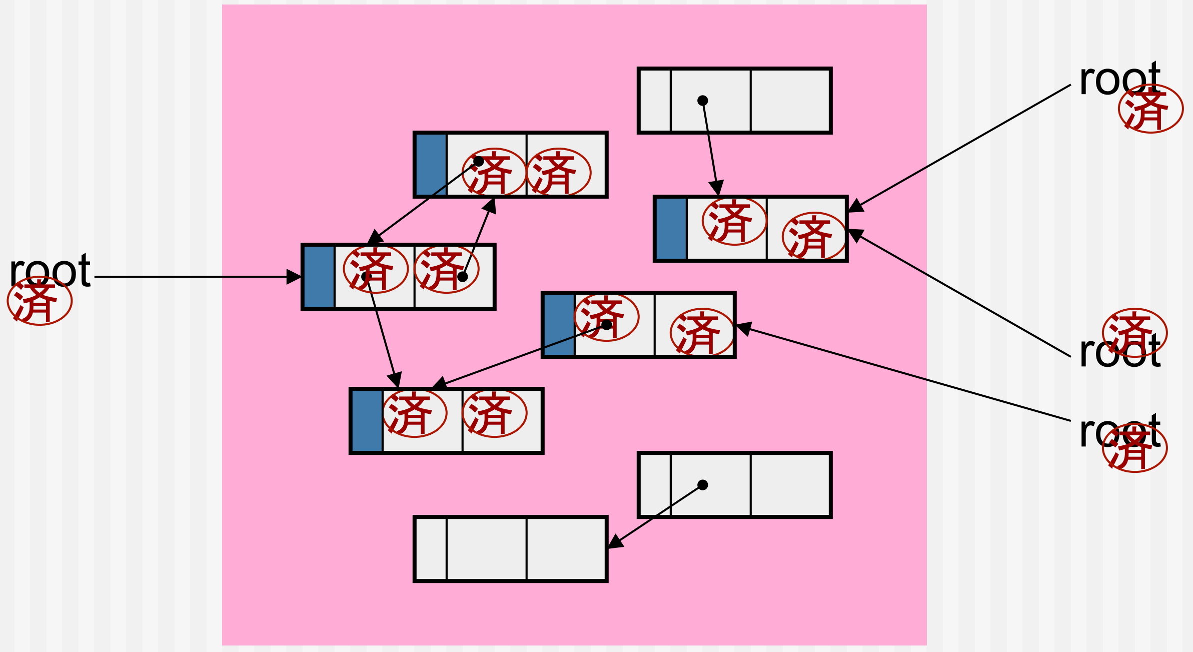
Basic algorithms

- Mark & sweep method
- Reference counting method
- Copying method

□
○

Mark & sweep method

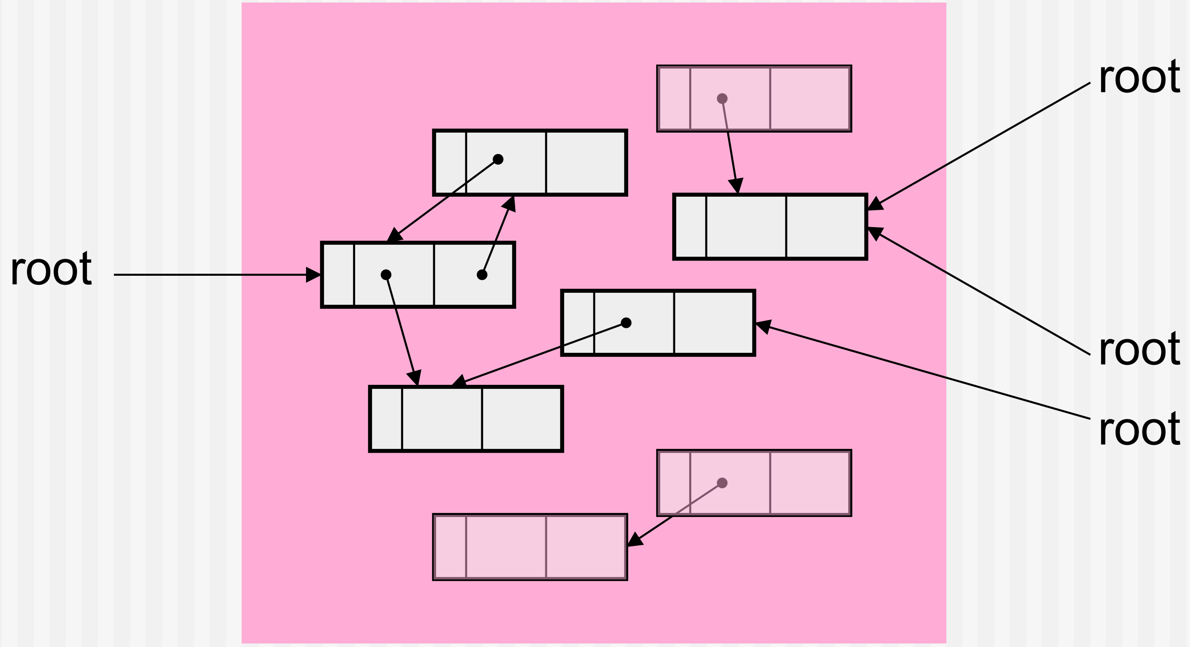
- Mark phase
  Follow pointers from the roots, and
  mark every object that are reached.
  ⇒ Mark bit



済: done

Mark & sweep method (cont)

- Sweep phase
  Scan the entire heap and collect objects that are not marked.
  Remove the mark from each object for the next GC.



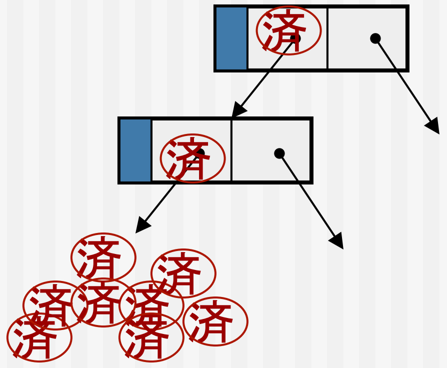
Mark & sweep method (cont)

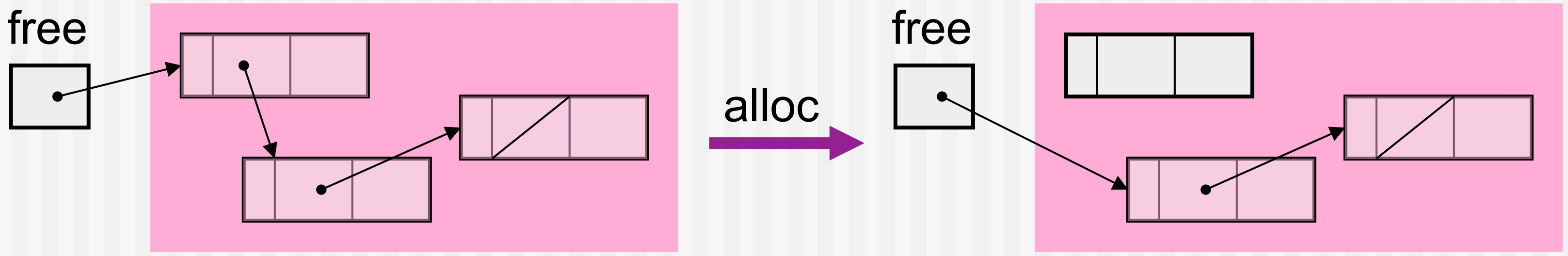
- Intuitive, easy to understand
- Execution time

  Mark: proportional to the number of live (non-garbage) objects

  Sweep: proportional to the size of the heap.
- Use a stack for GC.
- The collected areas are managed a free list.

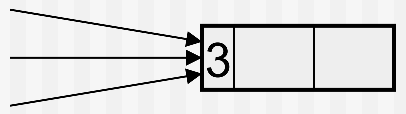
  ⇒ Possibility of memory fragmentation



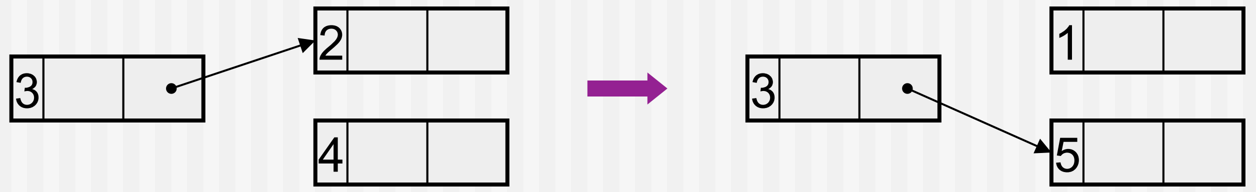


Reference counting method

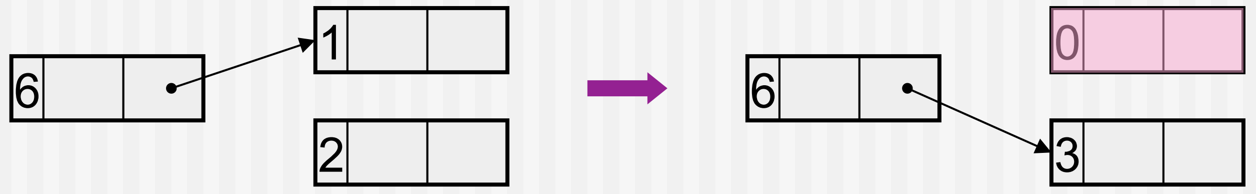
- Collect objects immediately that are not referenced by any object.
- For each object,
  prepare a reference counter.



- Update the reference counter upon pointer replacement.

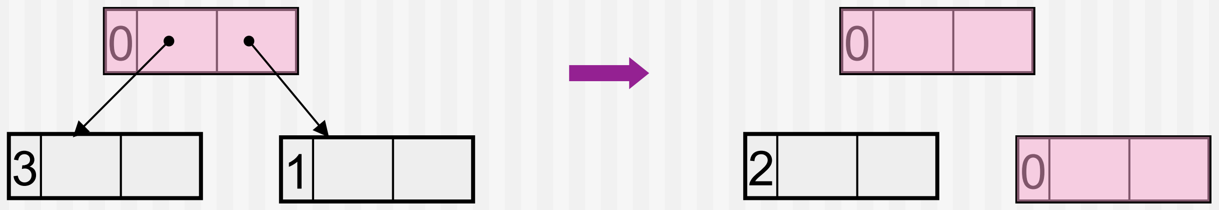


- Collect an object when its reference counter becomes 0.



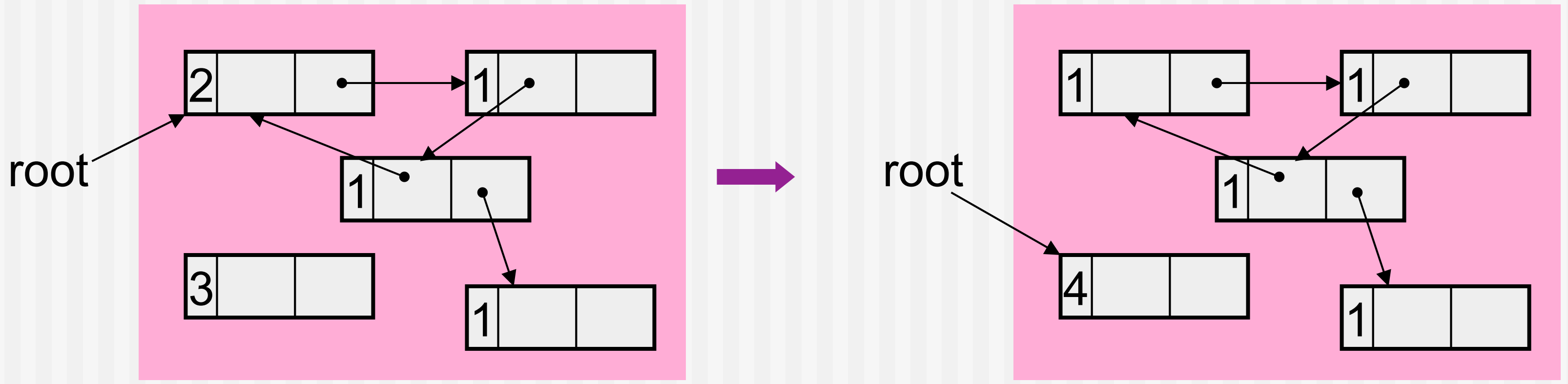
Reference counting method (cont)

- Update other counters upon collecting an object.
  - Collect also those objects whose counters became 0.



Reference counting method (cont)

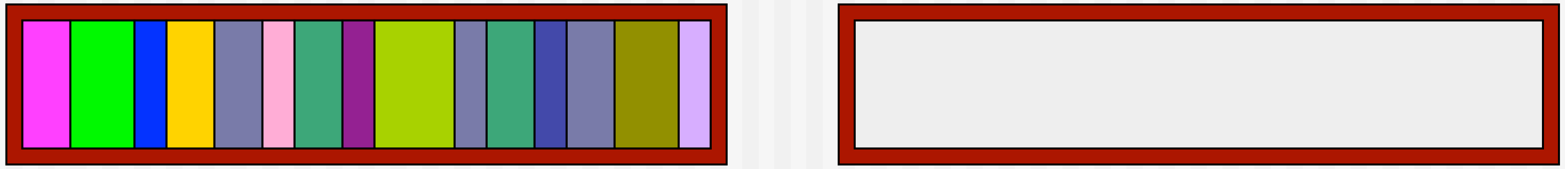
- Cannot collect objects along circular structures.
  ⇒ Used in combination with other methods (e.g., mark & sweep) .



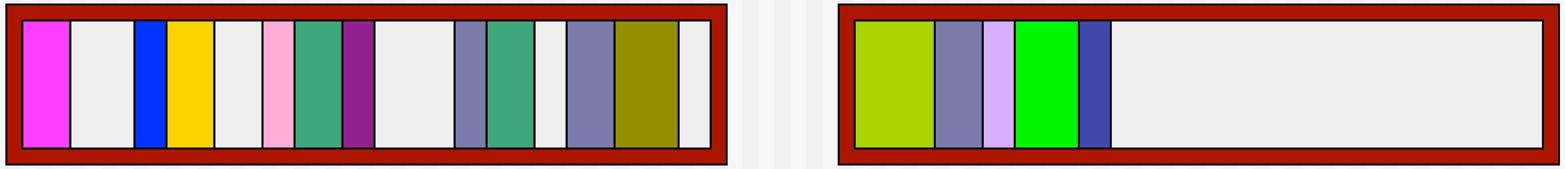
- Naive implementation results in a large system overhead.
  It is impractical to maintain the number of references from roots.
- The number of bits required for the reference counter:
  32 bits for each object?

Copying method

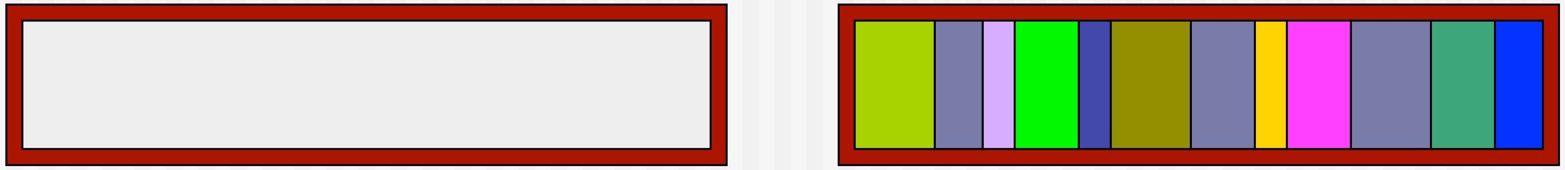
- Arrangement of books on bookshelves.



Move necessary books to the right shelf, with no space in between.



Dispose books on the left shelf, and put new books on the right shelf.



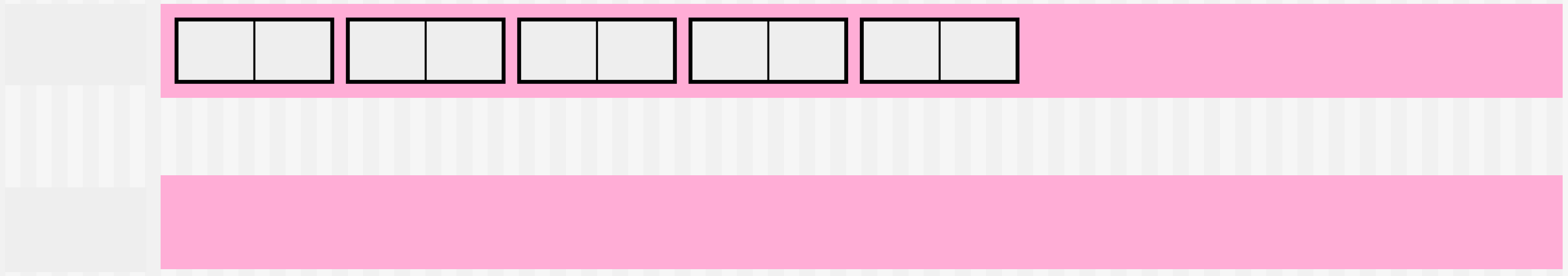
When the right shelf becomes full, use the left shelf to arrange books.

Copying method (cont)

- Divide the whole heap into two equal spaces (semispaces).
- Allocate objects from the beginning of one semispace.
- Scavenge when the semispace in use (‘from space’) becomes full.

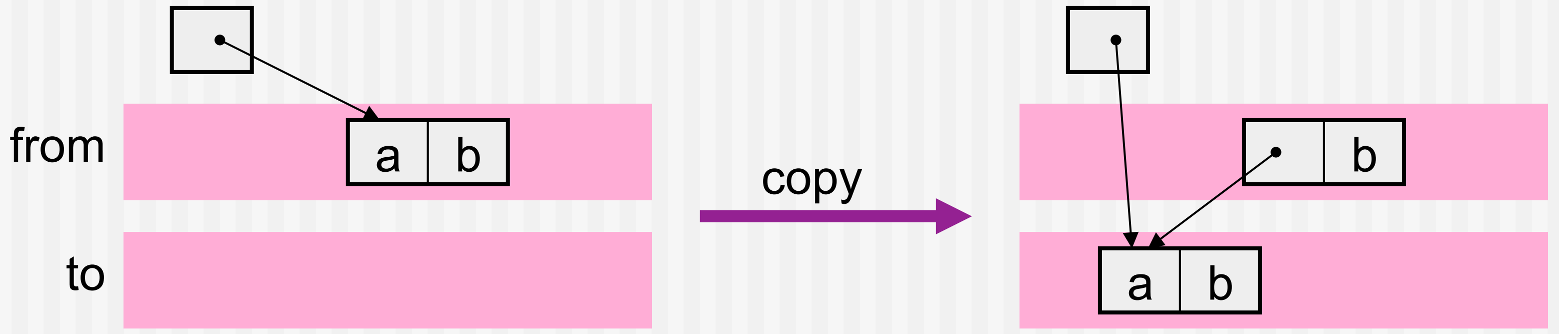
  Copy those objects that can be reached from the roots

  to the other semispace (‘to space’), packing toward one end.
- Allocate new objects in ‘to space’ packing toward one end.
- When ‘to space’ becomes full, reverse the roles of the ‘from space’ and ‘to space,’ and scavenge the corresponding space.

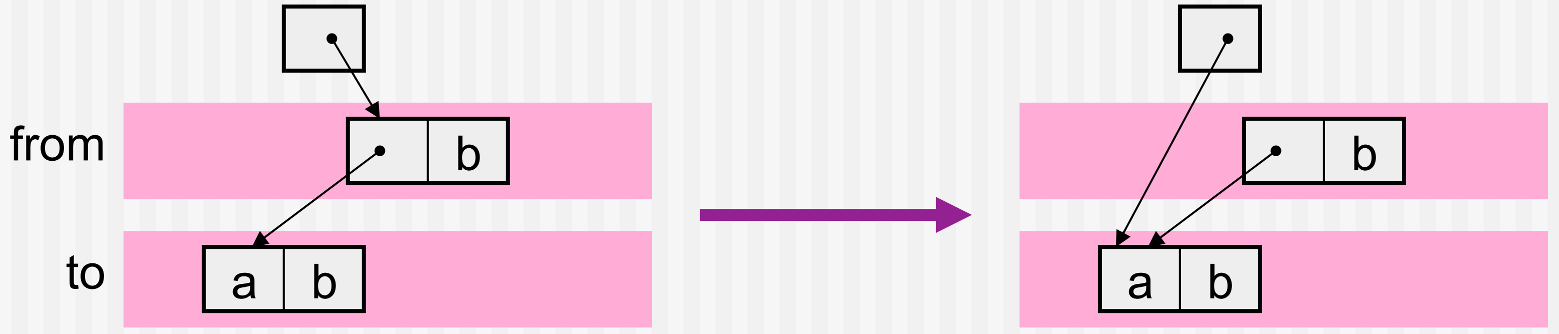


Copying method (cont)

- When copying: replace the pointer and leave a forwarding pointer.

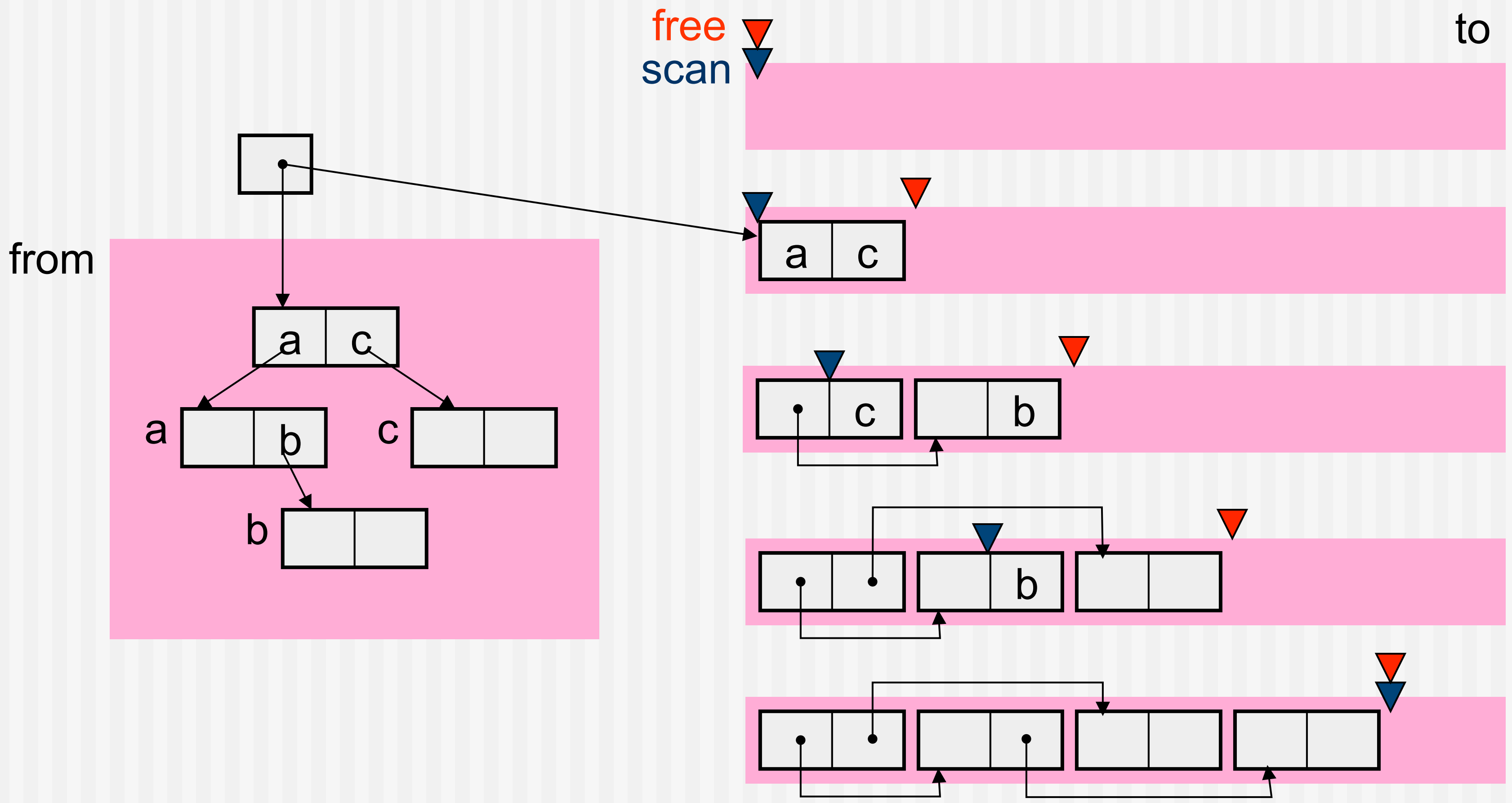


- Upon encountering an already copied object, replace the pointer.



Copying method (cont)

- Use ‘to space’ as a stack (actually, a FIFO queue).



Copying method (cont)

- Compact live objects to avoid memory fragmentation.

  Data locality ⇒ efficient when used with virtual memory and cache

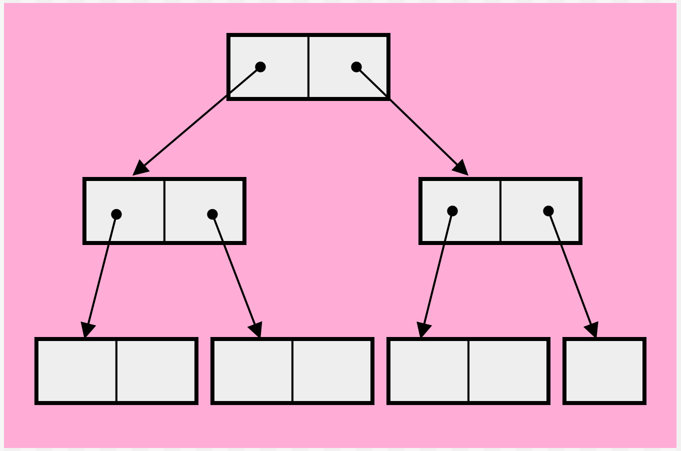
- No stack is necessary for GC.

  Unlike mark&sweep,

  objects are copied in breadth-first.

  For virtual memory and cache,

  depth-first is more suitable.



- Execution time:

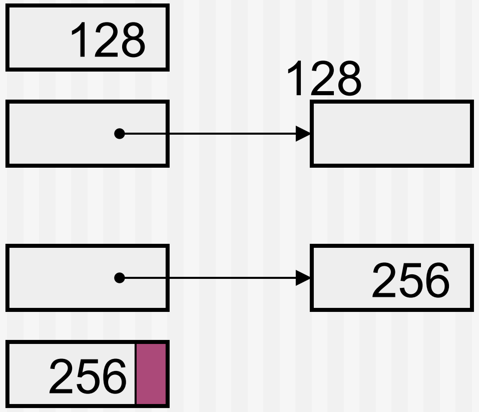
  proportional to the number of live objects

  independent of the size of the heap area.

- Only half of the heap area can be used for object allocation

Remarks for implementing GC

- Difficult to debug
  - Low repeatability
  - Probe effect
  - Temporal and spatial isolation of cause and effect
- Discrimination of pointers from others (such as numerical values)
  - Boxing and immediate data
  - Type information for each object
  - Type table for each stack frame
  - Conservative GC
    - Regard pointer-like objects as pointers.
    - Do not collect objects that may not be garbage.
    - Compaction is impossible.



Remarks for implementing GC (cont)

- Reserve GC bits (mark bit, type information).
  - For each object? Use bit tables?
- Relationship with language specifications
  - Is it possible to point the location in the middle of an object?
  - Are circulatory structures never generated?
- Finalization

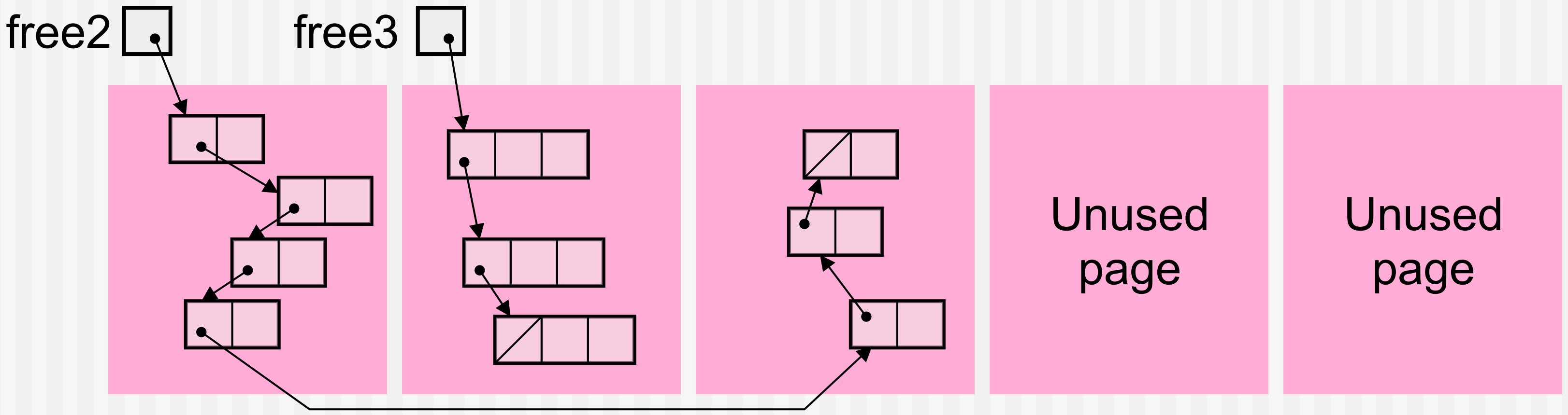
  Example: File objects & thread objects

Improvement of basic algorithms

- Mark & sweep method
  - Big bag of pages (BiBOP)
  - Pointer reversal
  - Mark & compact GC
- Reference counting method
  - Deferred reference counting
  - Sticky reference counts
  - Partial mark & sweep
- Copying method
  - Partial compaction
  - Approximately depth-first copying
  - Generational GC

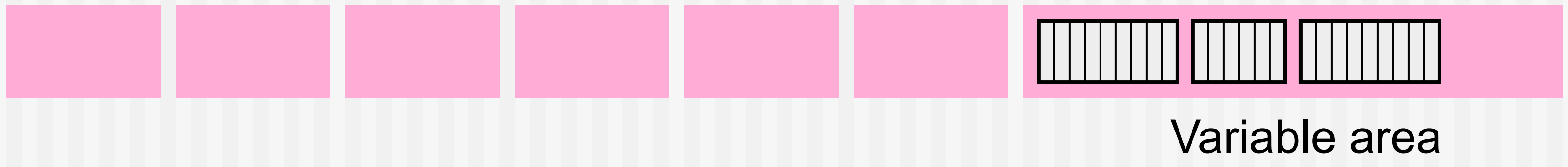
Big bag of pages (BiBOP)

- Avoid fragmentation and reduce time for sweeping
- Manage heap area in page units
- Only objects of the same size can be allocated in each page
- One free list for each object size
- When the free list for a certain object size becomes empty:
  - Use an unused page or start GC.
  - Arrange to avoid shortage of objects of a particular size.



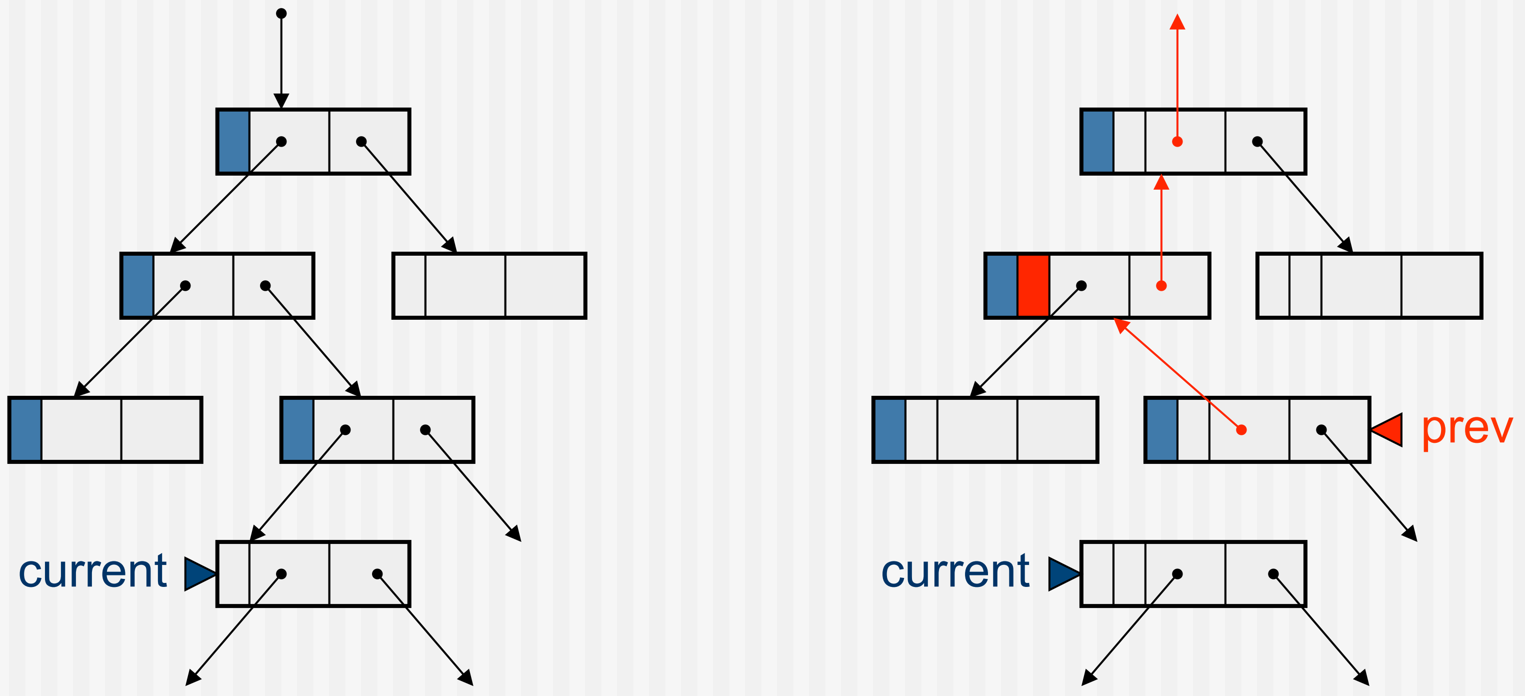
Big bag of pages (BiBOP)

- Sweep only the pages in use.
- Change the status of a page to ‘unused,’ if there is no live object in the page.
- For variable-size objects (such as arrays and character strings)
  - Allocate them in the heap area (variable area) managed in a different way.
  - Use the copying method or mark & compaction method.

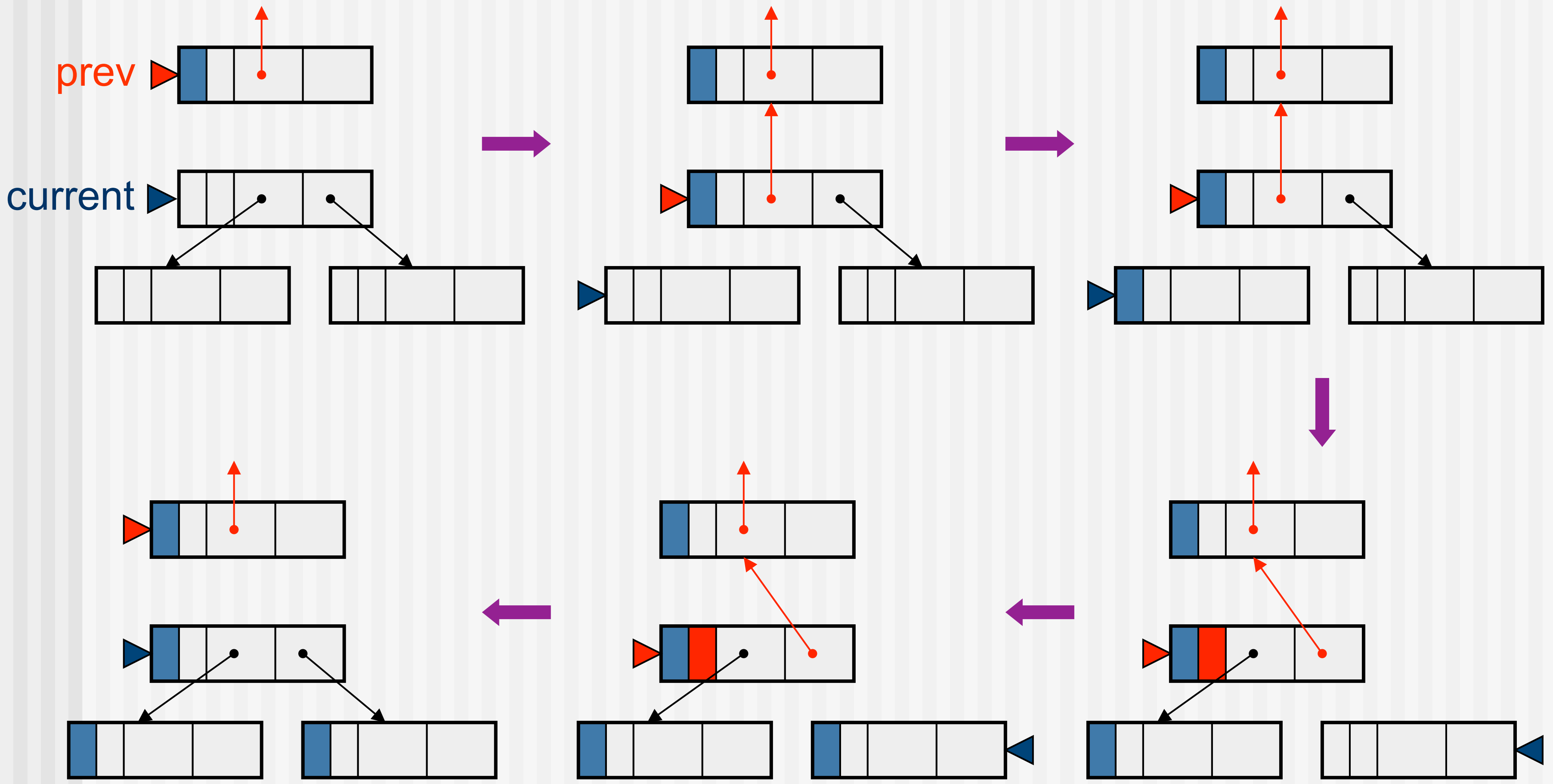


Pointer reversal

- Use objects as a GC stack.
- Flags showing the location of reversed pointers

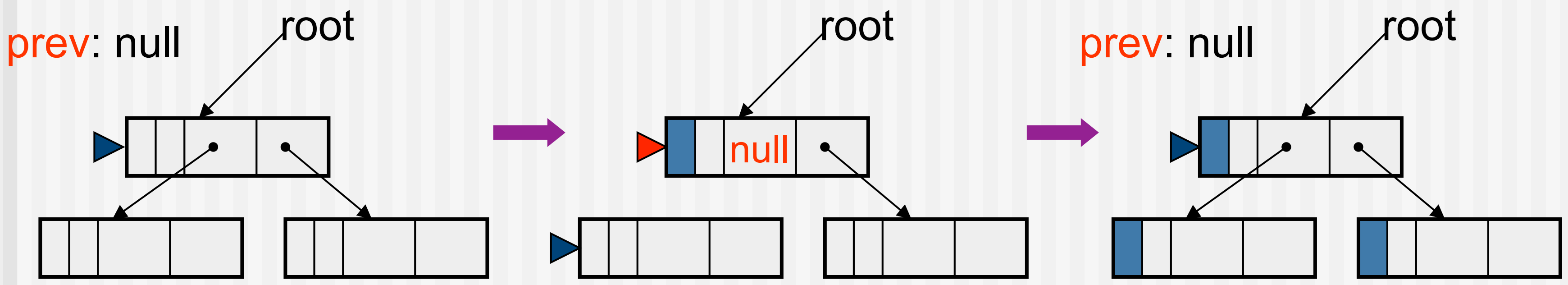


Pointer reversal

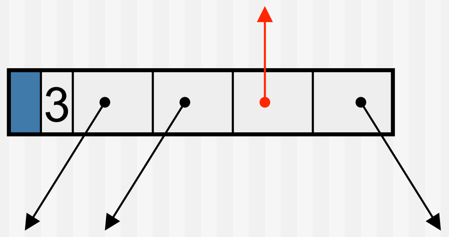


Pointer reversal

- Initialize prev to null, at the start of marking.



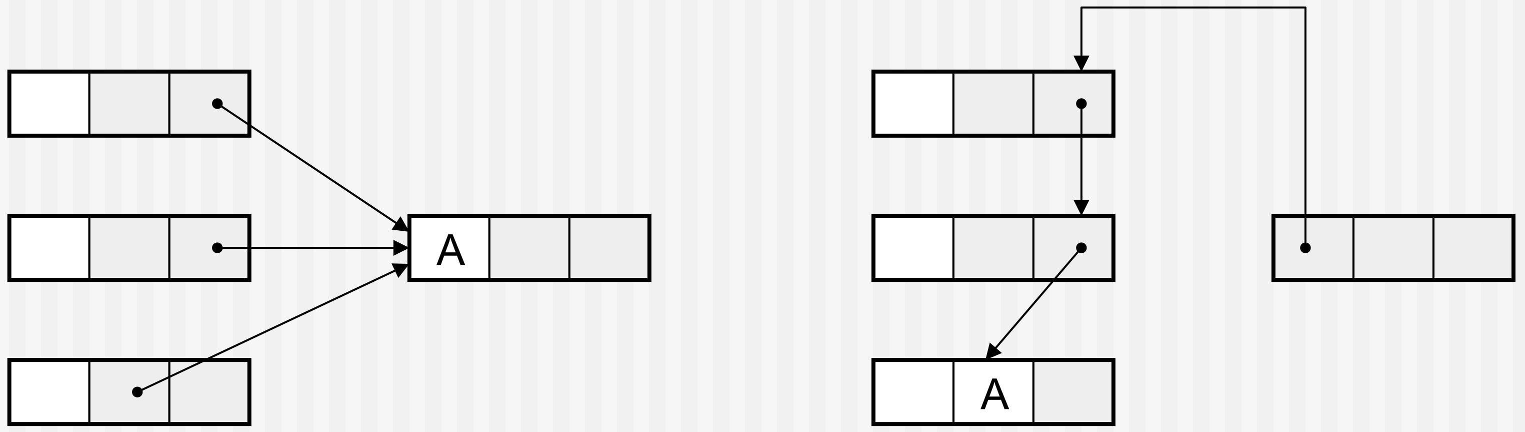
- When the number of slots is three or more, store a slot number instead of a flag.



Mark & compact GC

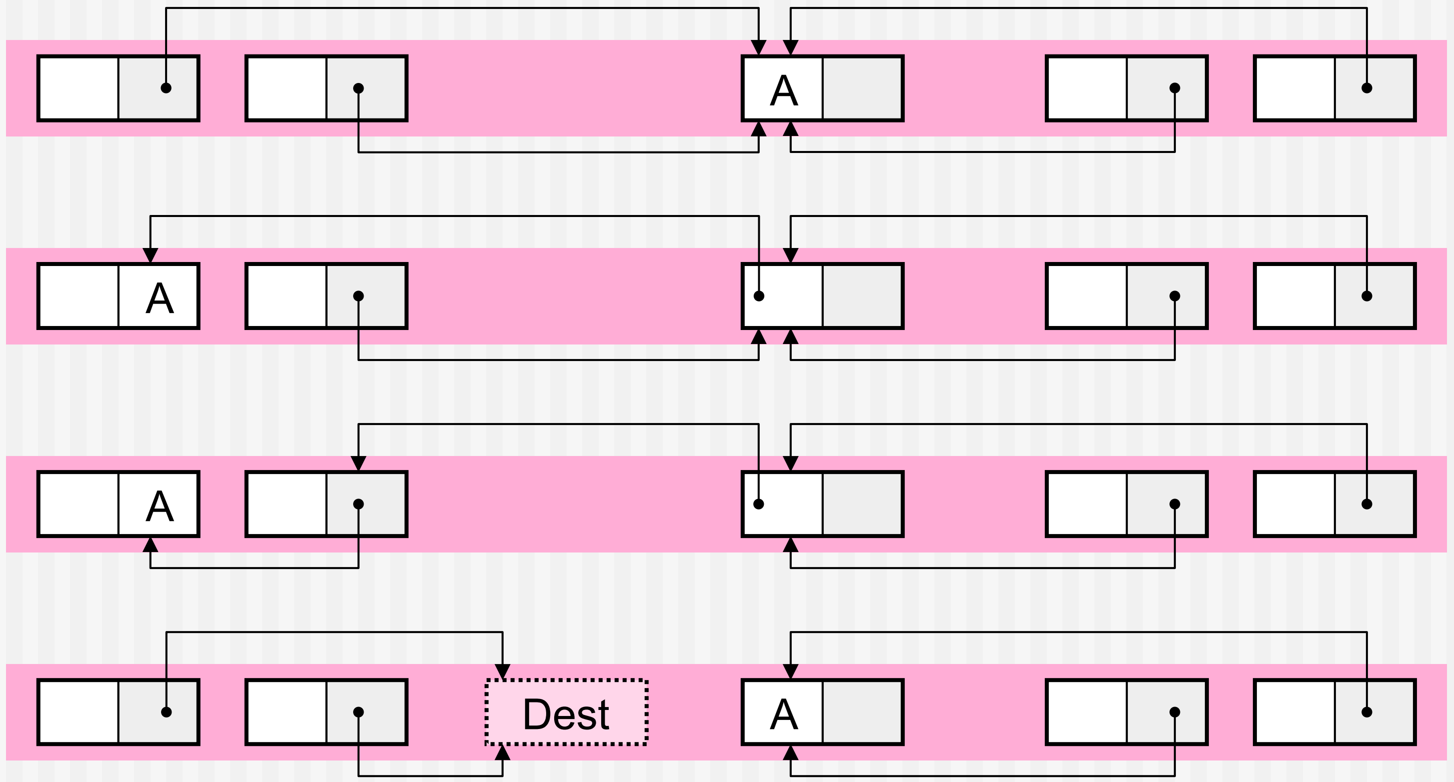
- Threaded method

  Update pointers by following the thread.
- Assumptions:
  - The header of an object (type info, mark bit) is a single word.
  - The content of the header can be distinguished from pointers.



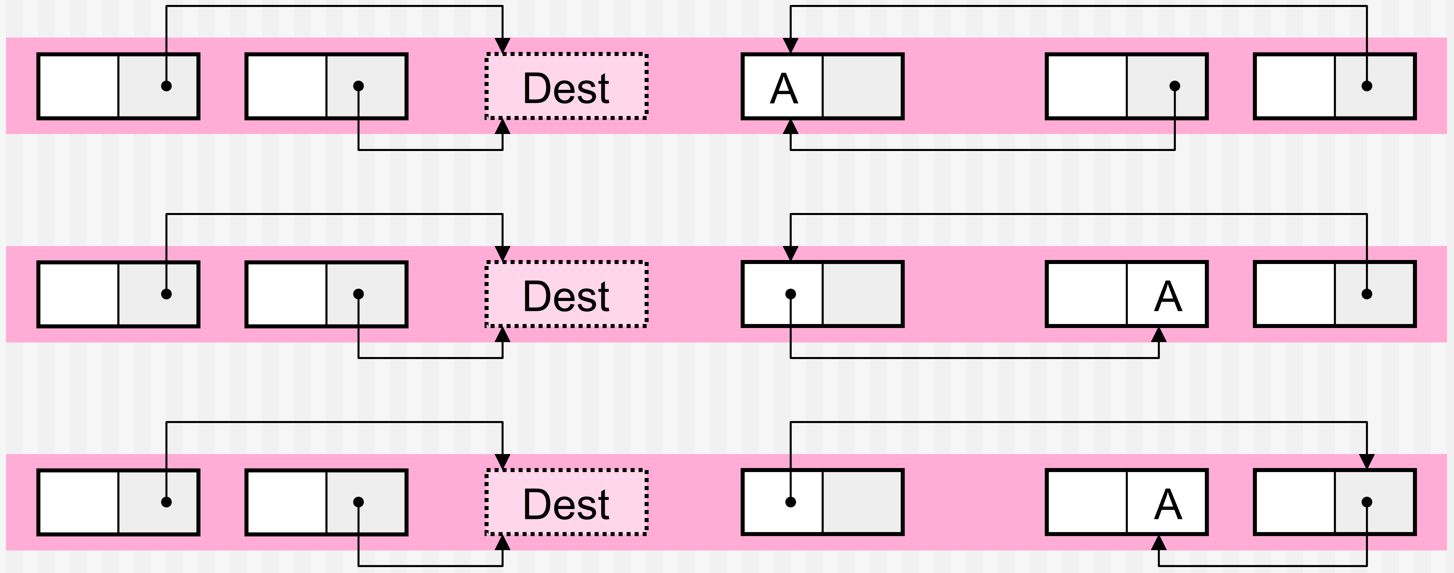
Mark & compact GC (compaction phase 1)

- Scan the heap and compute the destination to which each live object is moved.
- Replace each forward pointer (rightward in the figure) with the pointer to the destination.



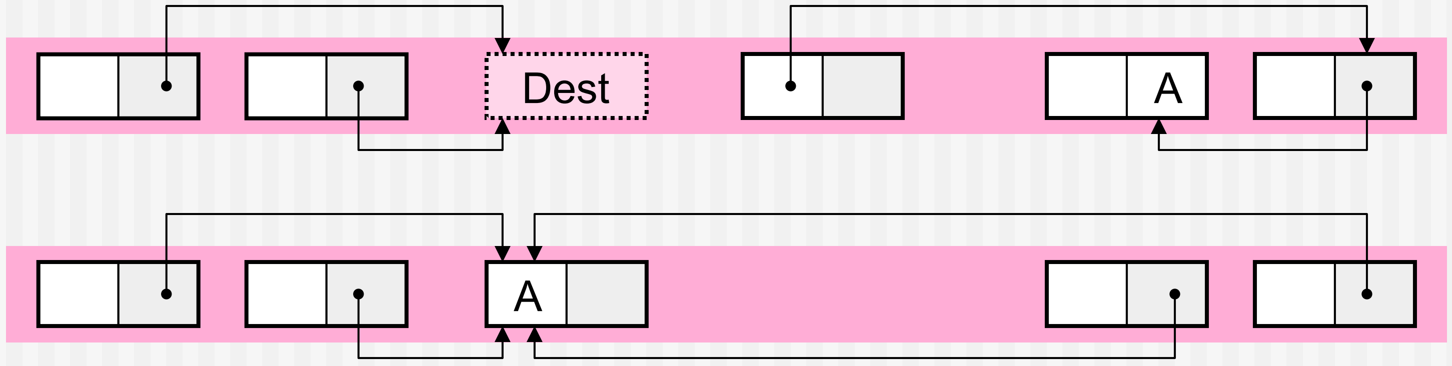
Mark & compact GC (compaction phase 1)

- Thread together backward pointers (leftward in the figure).



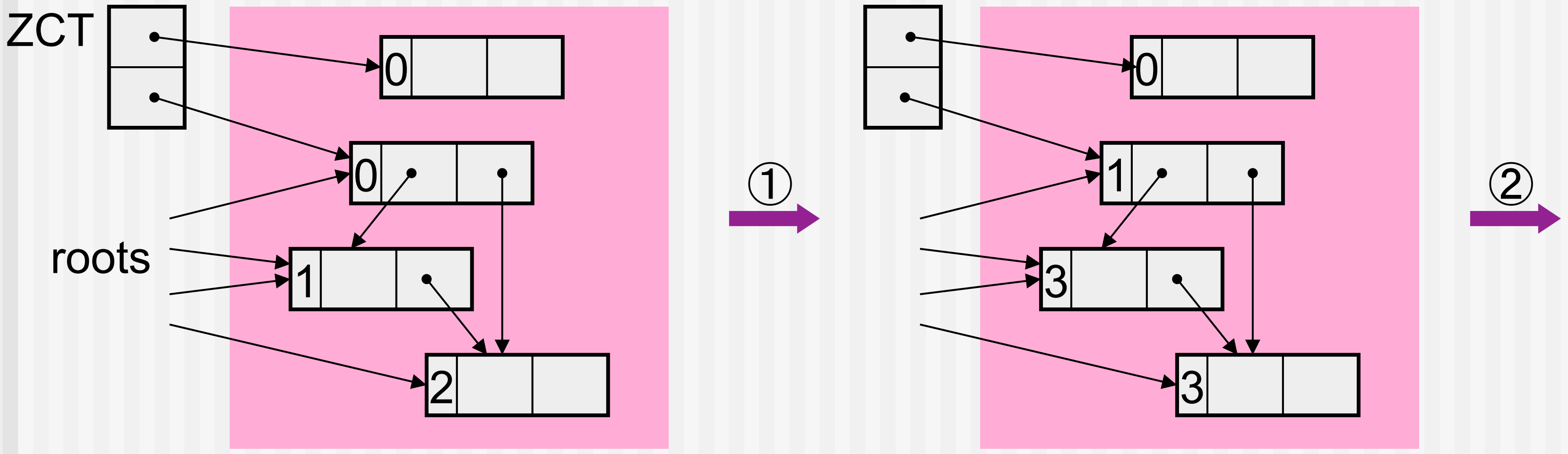
Mark & compact GC (compaction phase 2)

- Scan the heap and move live objects.
- Replace each backward pointer (leftward in the figure) with the pointer to the destination.



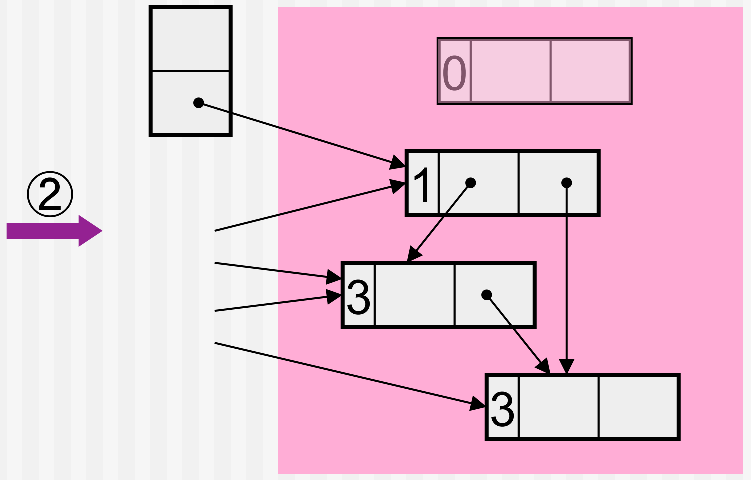
Deferred reference counting

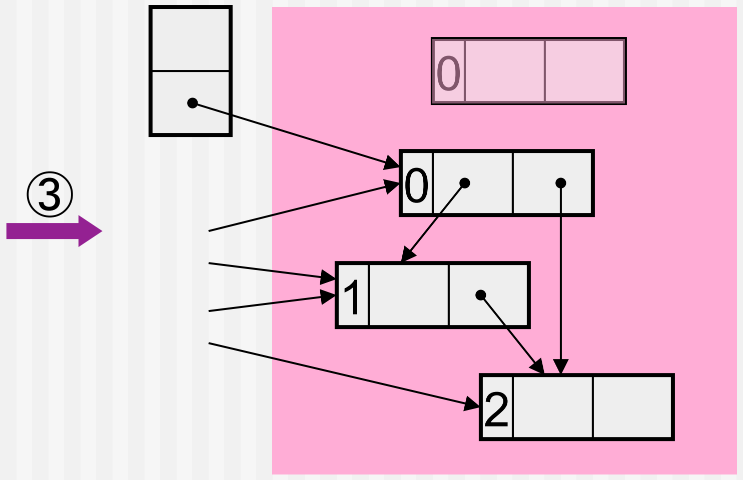
- References from the roots are not counted.
- Register an object when its counter becomes 0 in a zero-count table (ZCT).
- When ZCT becomes full.
  ① Scan the roots and increment the counter of the objects they point to.
  ② Scan the ZCT, collect objects with counter 0, and deregister them.
  ③ Scan the roots and decrement the counter of the object they point to.



Deferred reference counting

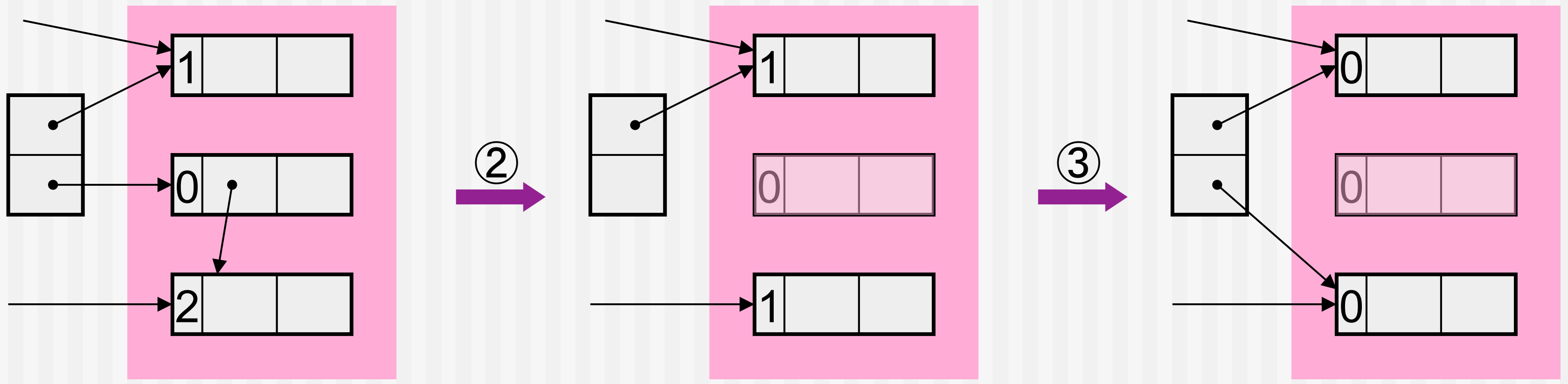
- References from the roots are not counted.
- Register an object when its counter becomes 0 in a zero-count table (ZCT).
- When ZCT becomes full.
  - ① Scan the roots and increment the counter of the objects they point to.
  - ② Scan the ZCT, collect objects with counter 0, and deregister them.
  - ③ Scan the roots and decrement the counter of the object they point to.





Deferred reference counting

- Collect objects whose counters became 0 by collecting other objects.
- During process ③, register to ZCT an object if its counter becomes 0.
    - Expand the ZCT if it becomes full.

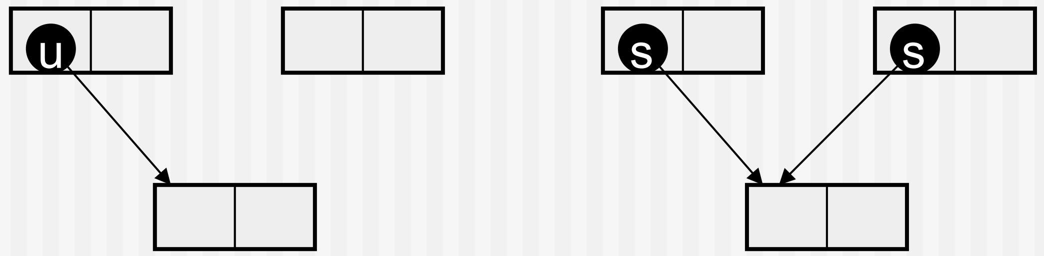


Sticky reference counts

- Use K bits as the reference counter for each object.

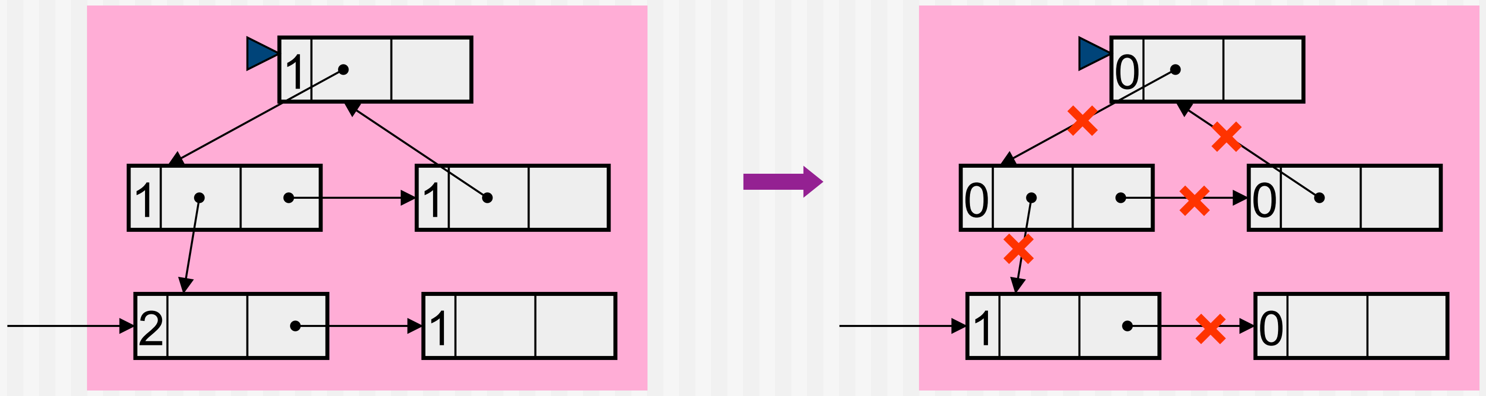
$$\begin{tikzcd}
0 \arrow[r, shift left, "+1"] & 1 \arrow[l, shift left, "-1"] \arrow[r, shift left, "+1"] & 2 \arrow[l, shift left, "-1"] \arrow[r, shift left, "+1"] & \cdots \arrow[l, shift left, "-1"] \arrow[r, shift left, "+1"] & 2^K-2 \arrow[l, shift left, "-1"] \arrow[r, "+1"] & 2^K-1 \arrow[loop above, "+1"] \arrow[loop below, "-1"]
\end{tikzcd}$$

- Use in combination with other method (e.g., mark & sweep)
- When K = 1: One bit reference count
  0 ･･･ Unique (referred to from only one place)
  1 ･･･ Shared (referred to from multiple places)



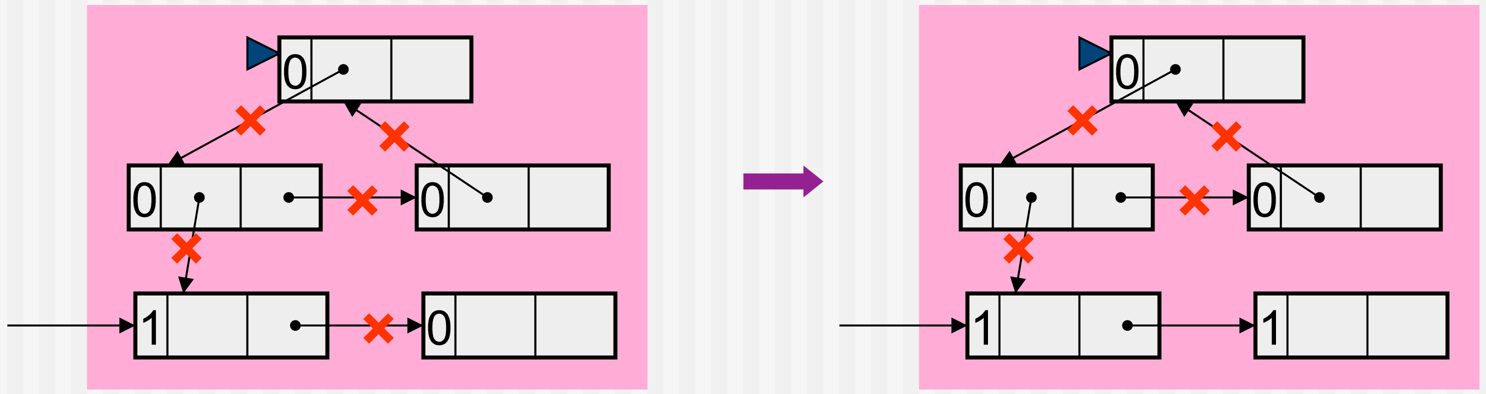
Partial mark & sweep

- Avoid marking and sweeping of the entire heap, and quickly collect garbage objects in circulatory structures.
- Disconnect all pointers that can be reached from "focused objects".

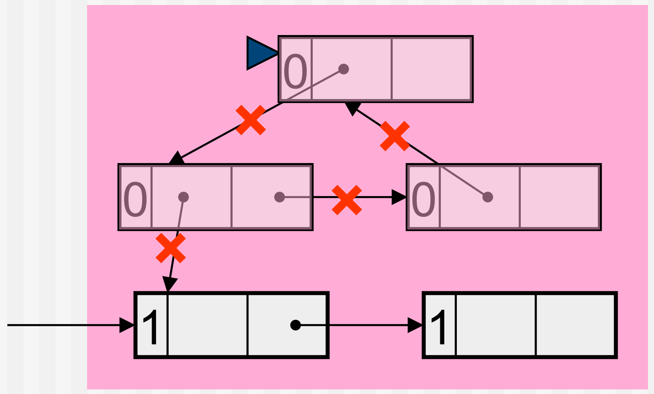


- Among the objects reached (the "group") , objects with non-zero counters are referred to from outside the group.
  ⇒ Restore all pointers that can be reached from these objects.

Partial mark & sweep

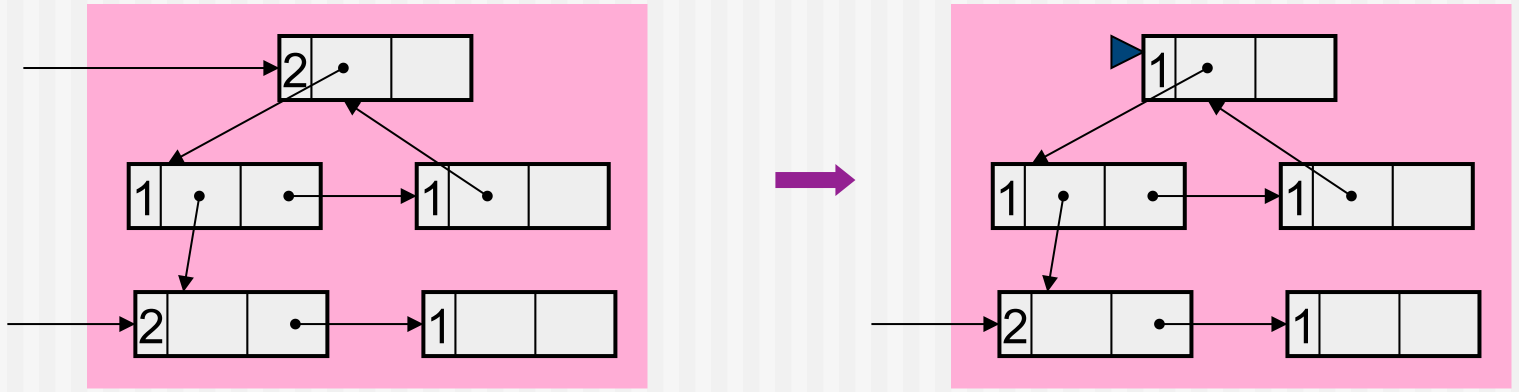


- Among the objects that can be reached from the "focused objects", collect those whose counter is 0.



Partial mark & sweep

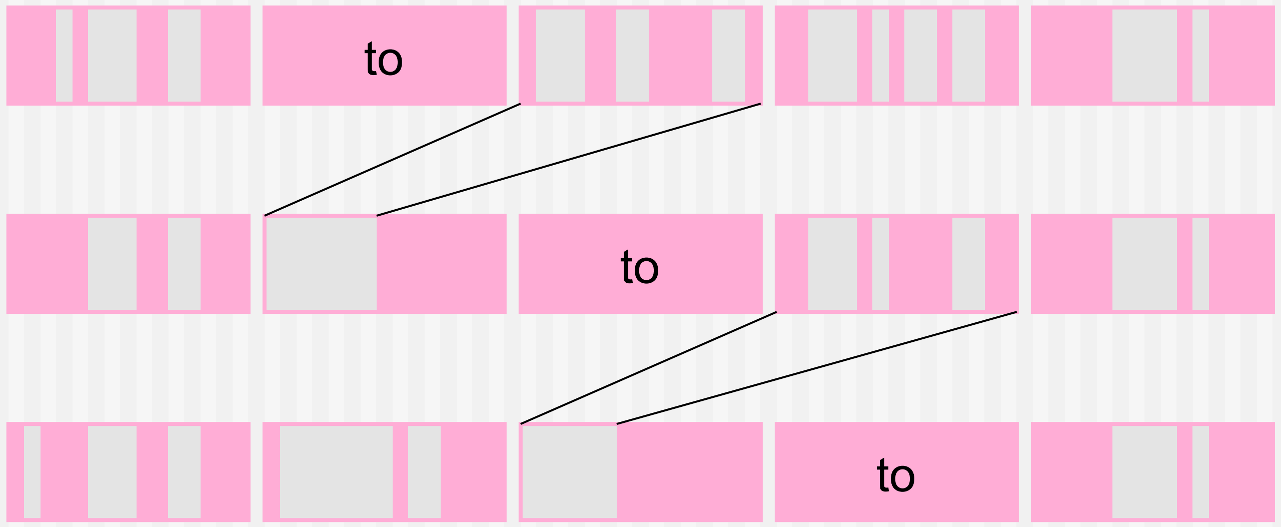
- "Focused objects" are those:
  - whose counter decreased after the previous partial mark & sweep and
  - whose counter did not increase nor whose contents did not change thereafter.
- Start partial mark & sweep when "focused objects" accumulated beyond a certain level.



- Implemented in IBM Jalapeño JVM (2001).

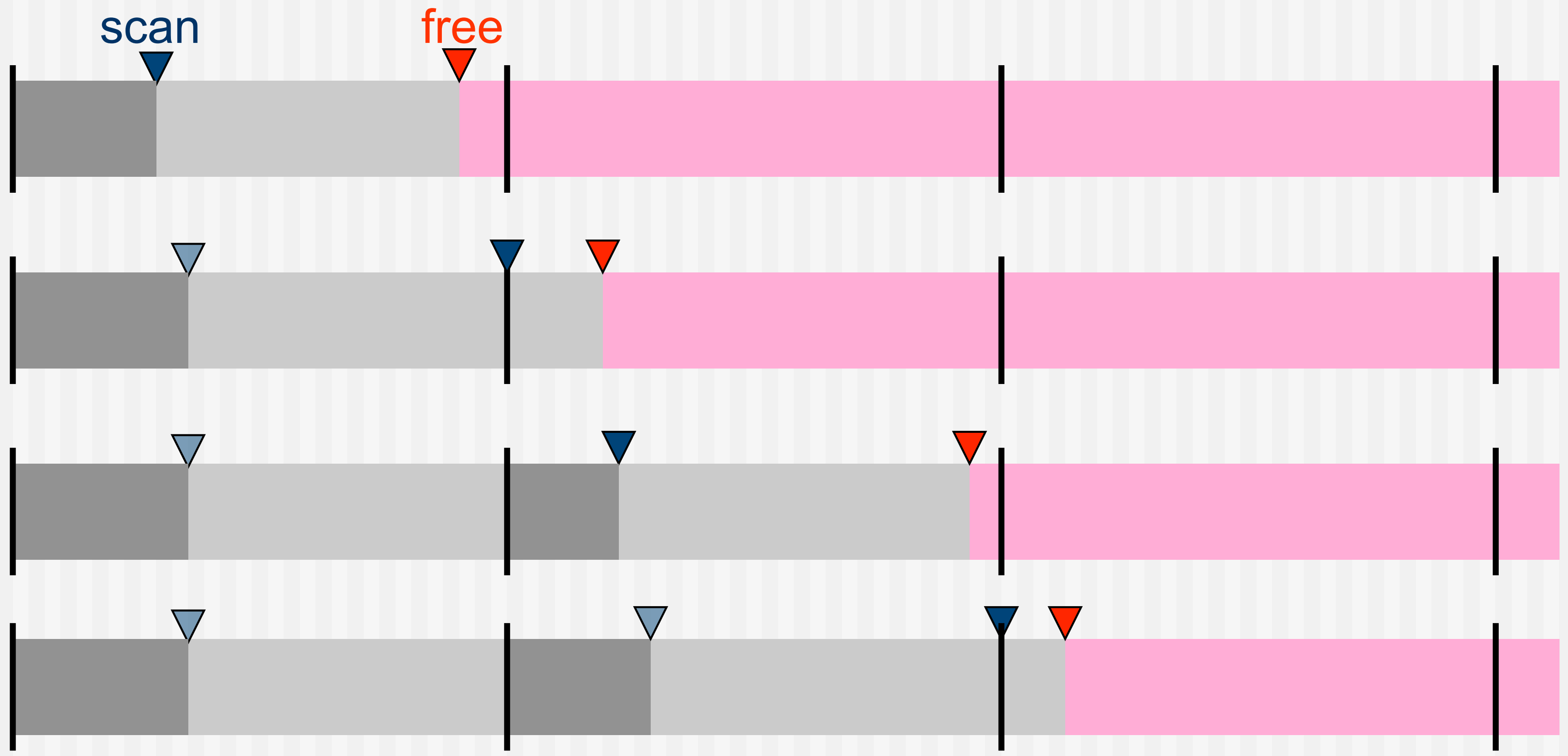
Partial compaction (Multiple-area collection)

- Divide the heap into N equal spaces (semispaces).
- Keep one semispace free as the "to space."
- Collect garbage in the semispace next to the "to space" by copying, and that in the other semispaces by mark & sweep.
- Use the "from space" as the "to space" in the next GC.



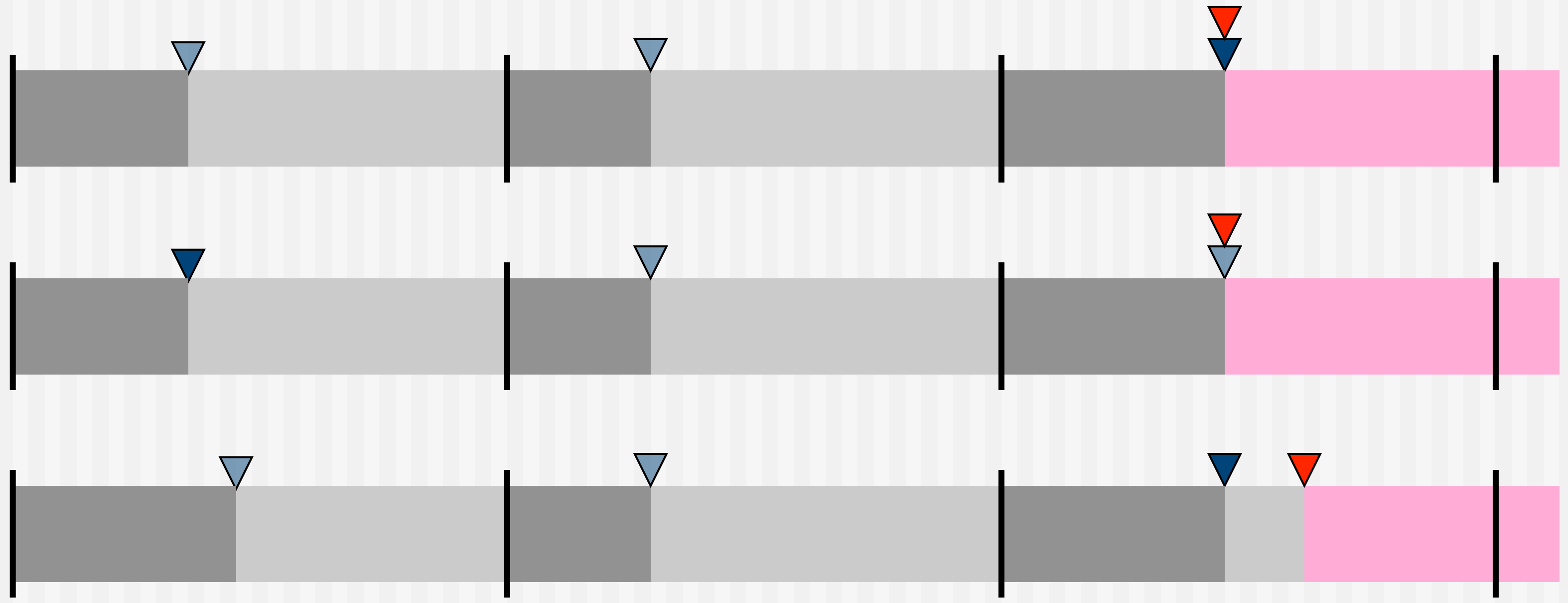
Approximately depth-first copying method

- Place an object and those objects it points to in the same page.
- When an object is copied on a new page, start scanning from that page (partial copy).



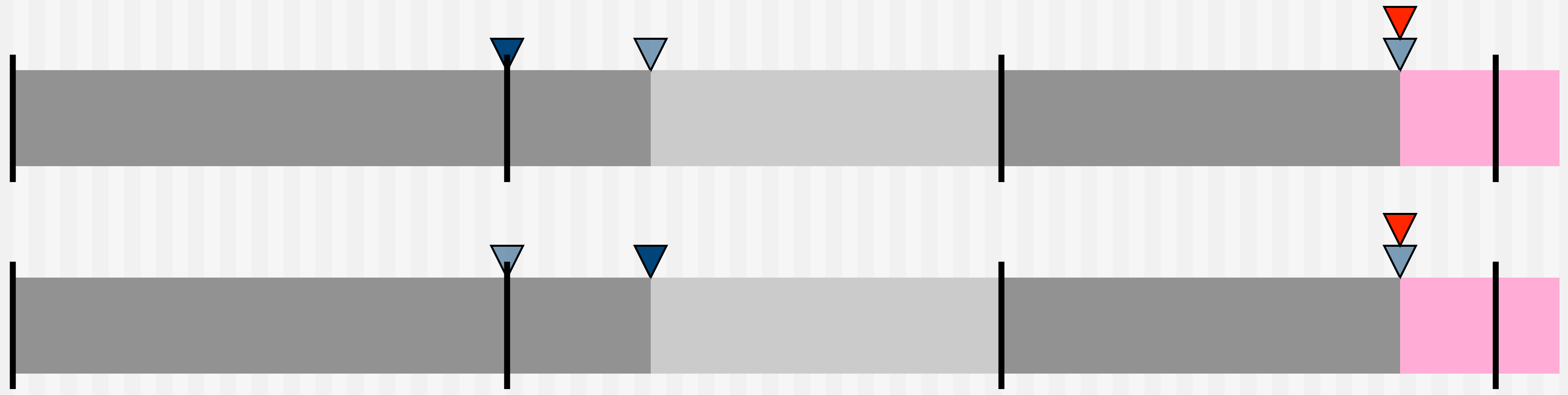
Approximately depth-first copying method

- When the partial copying is completed, continue scanning from the first page. When a copying is carried out, resume scanning the copied-to page.

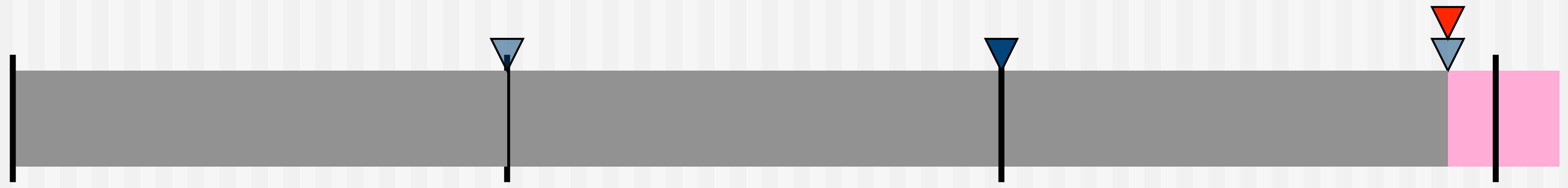


Approximately depth-first copying method

- When the scanning of a page is completed, resume scanning the next page.

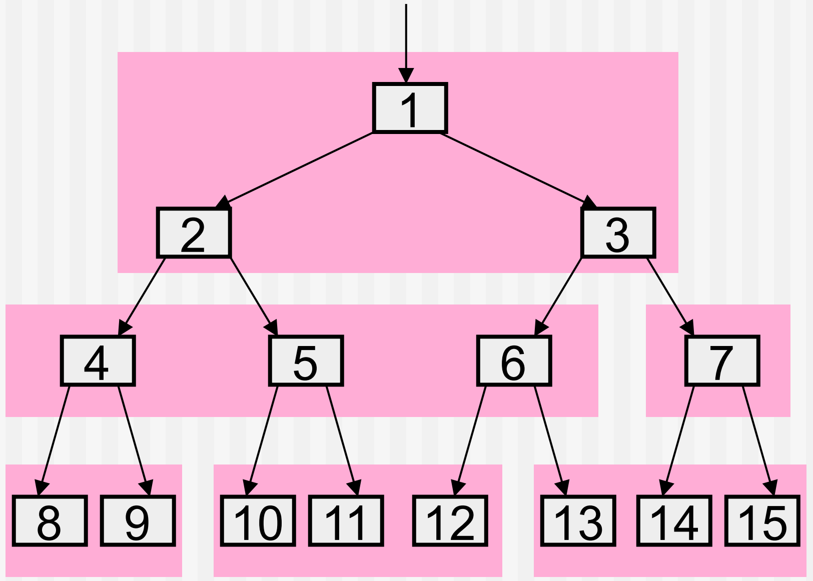


- GC ends when all the pages have been scanned.

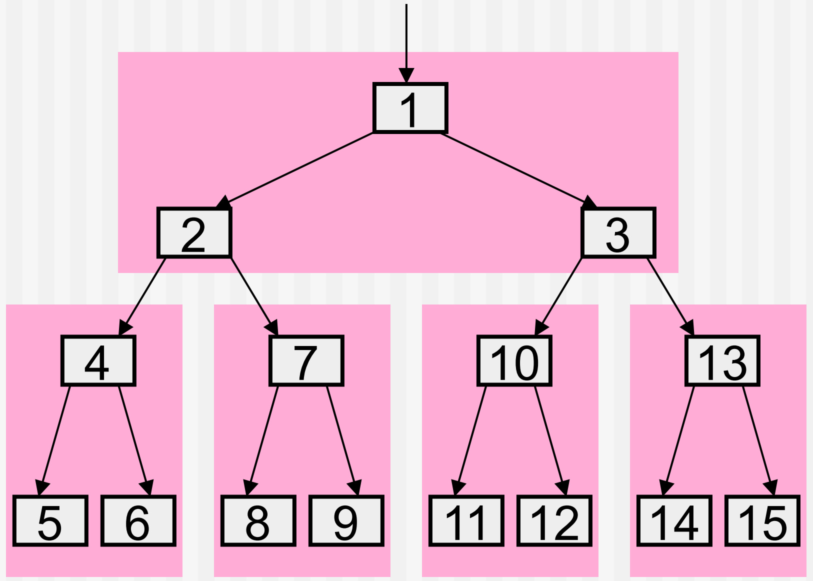


Approximately depth-first copying method

- Example of binary tree
  - Suppose that only three objects can be allocated on each page.
  - The numbers show the order of copying.



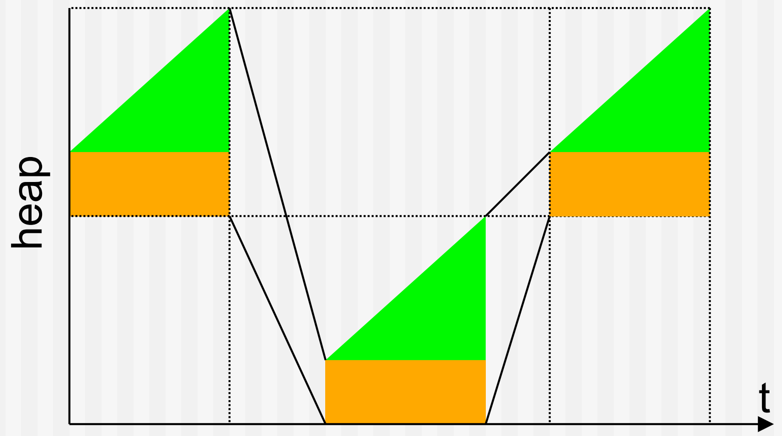
Breadth-first
(conventional copying method)

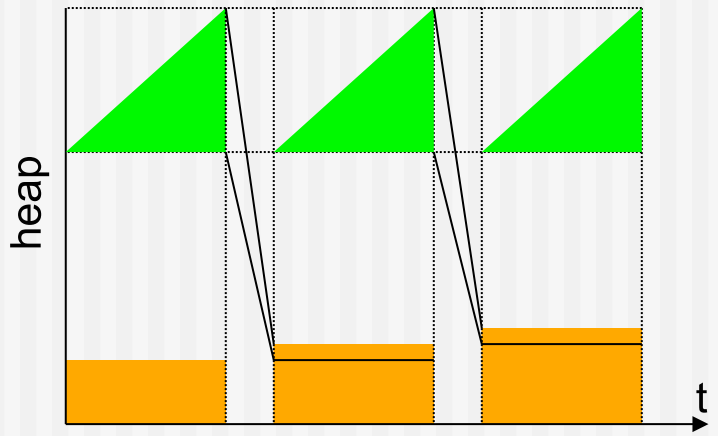


Approximately depth-first
copying method

Generational GC

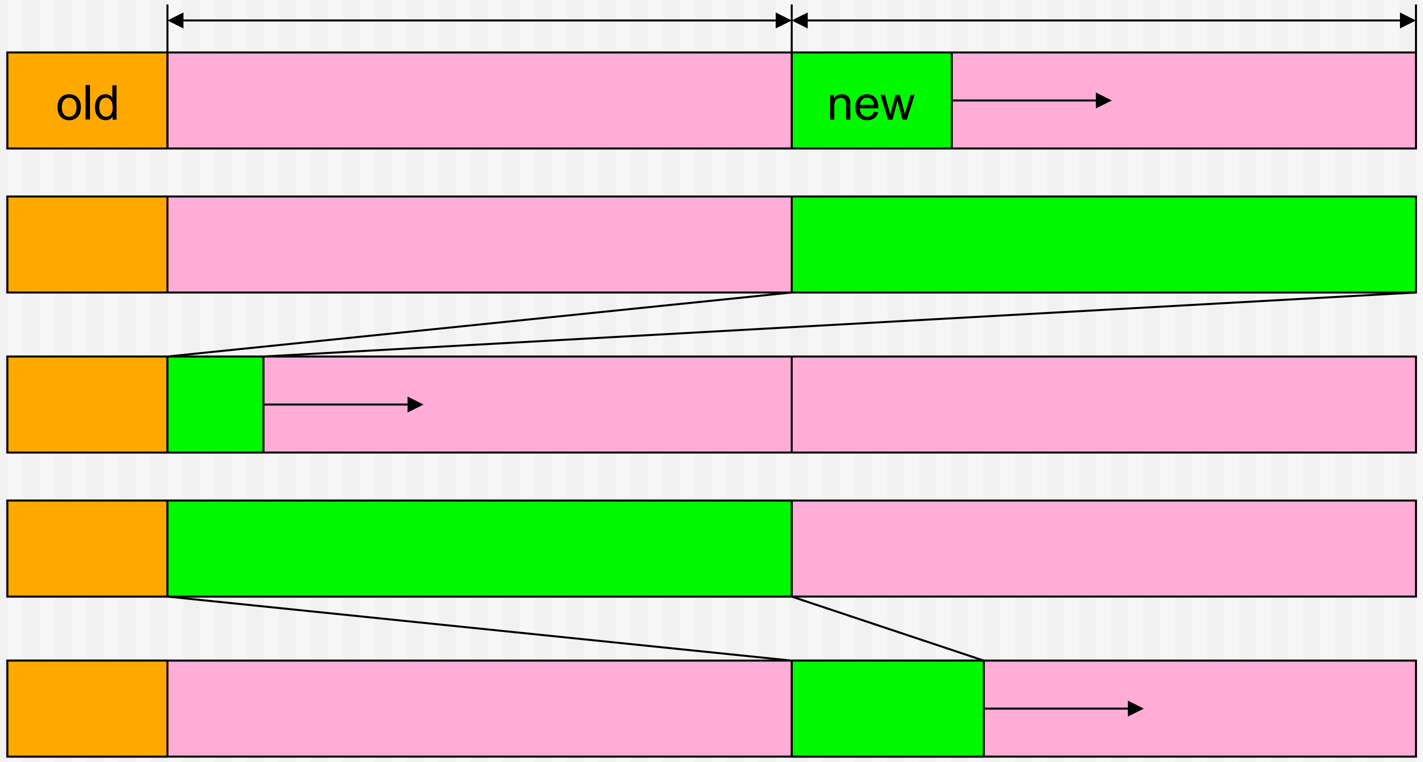
- Many objects become garbage within a short period after their creation.
- Objects that survive for a certain period tend to stay alive.
- It is a waste of time to copy all objects each time GC takes place.
- Introduce the concept of "age" to objects.
- Discriminate young (new) generation objects from old generation.
- Most of the time, collect only new-generation objects (minor GC).
    - GC is completed quickly because not many objects survive.





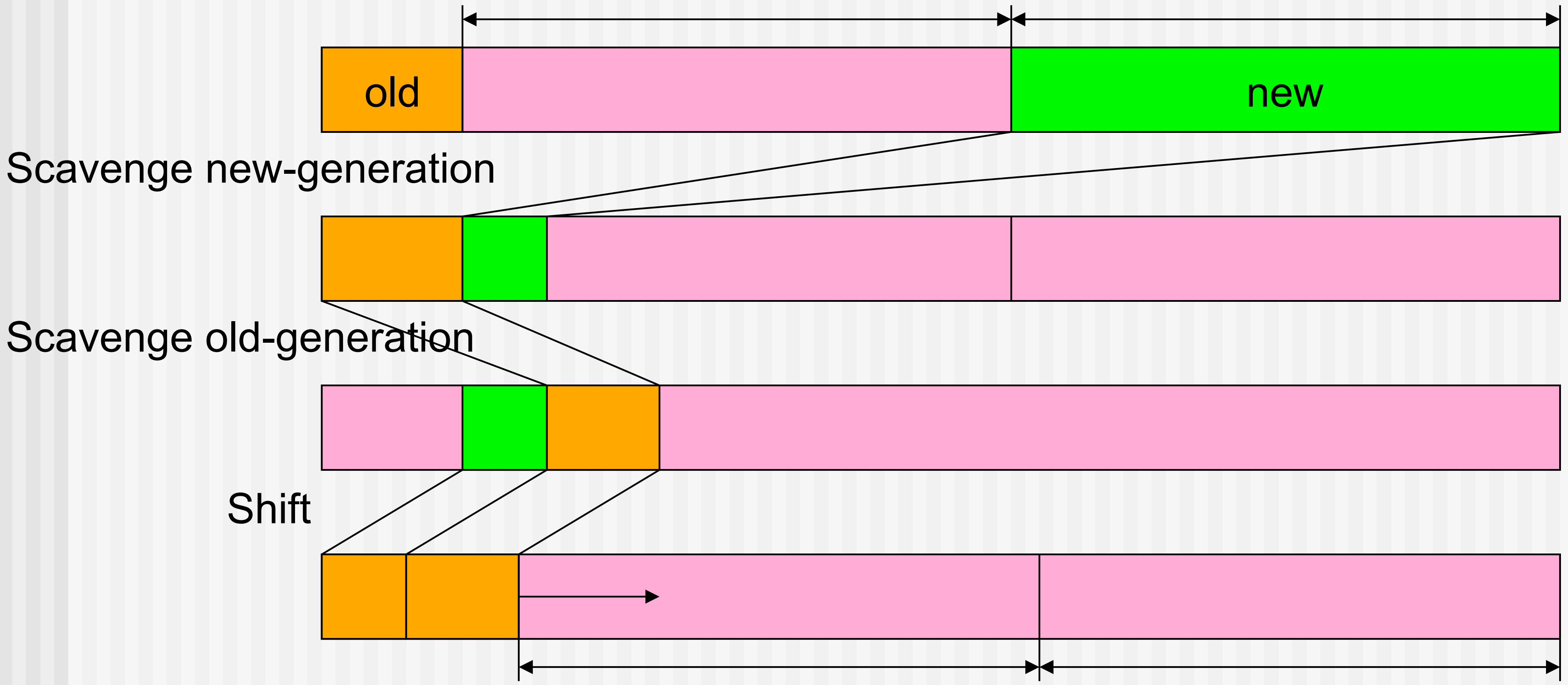
Generational GC

- Sometimes collect old generation objects (major GC) as well.
  ⇒ Disposal of old dead objects.
- An example of SML/NJ (minor GC)



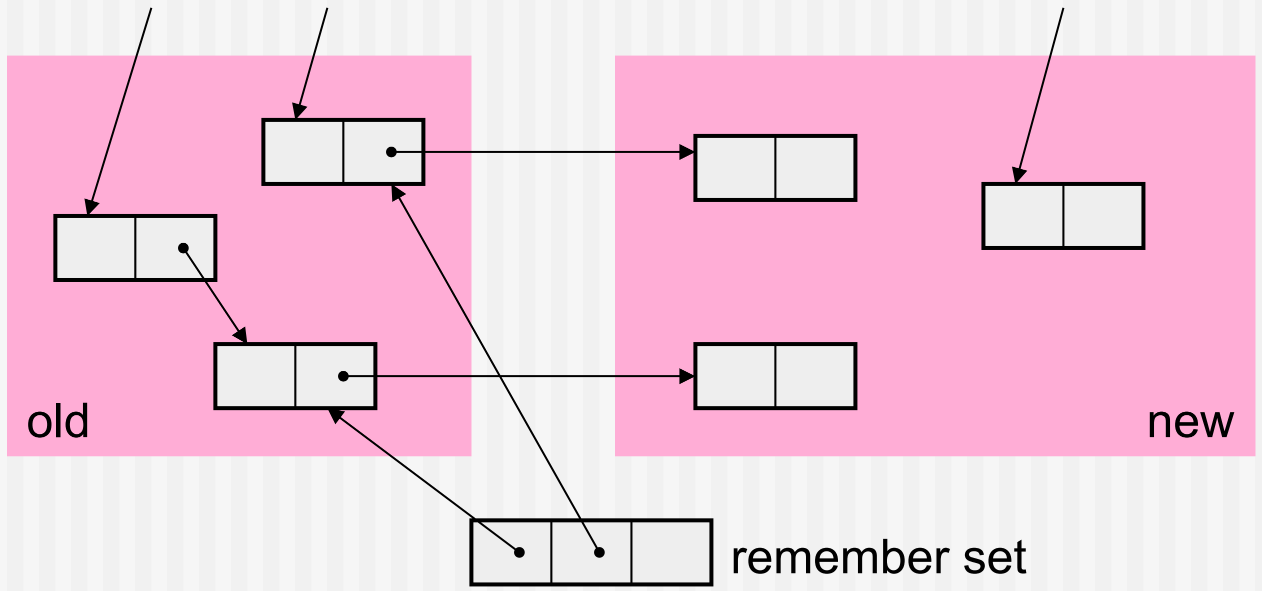
Generational GC

- An example of SML/NJ (major GC)



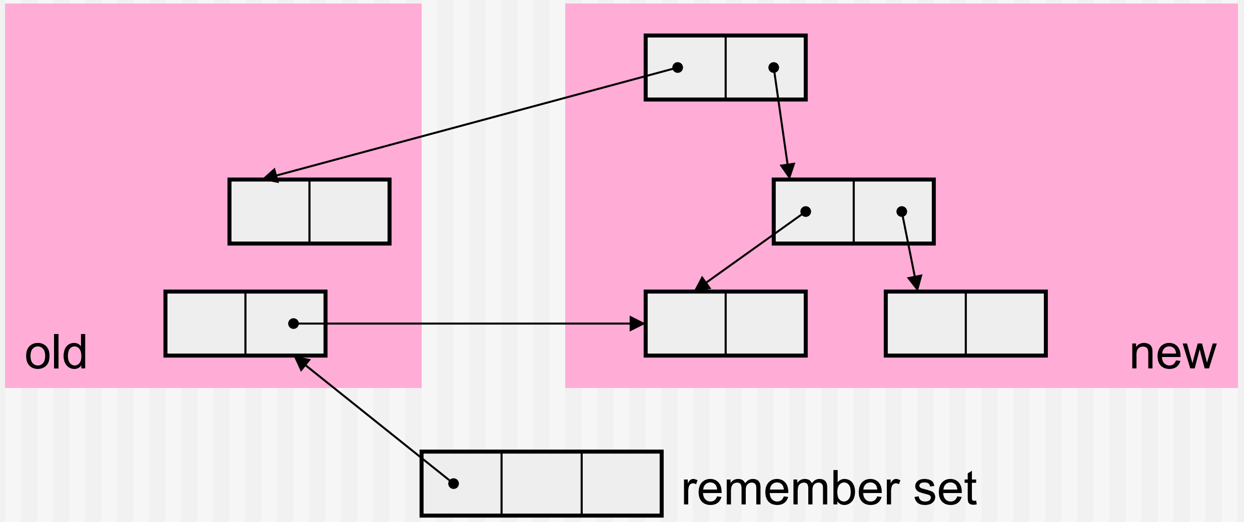
Generational GC

- Remember set
  - Minor GC needs to know references from old-generation.
  - The merit of generational GC decreases if we examine the entire old-generation.
  - Record (positions of) pointers from old-generation to new-generation.
  - These positions are also regarded as roots for minor GC.



Generational GC

- Write barrier
  - Most operations do not create old-to-new pointers.
  - Only write operations create old-to-new pointers.
  - Set up "barriers" on write operations.
  - If it is a writing of a pointer to new generation into an old-generation objects, register the position of writing in the remember set.



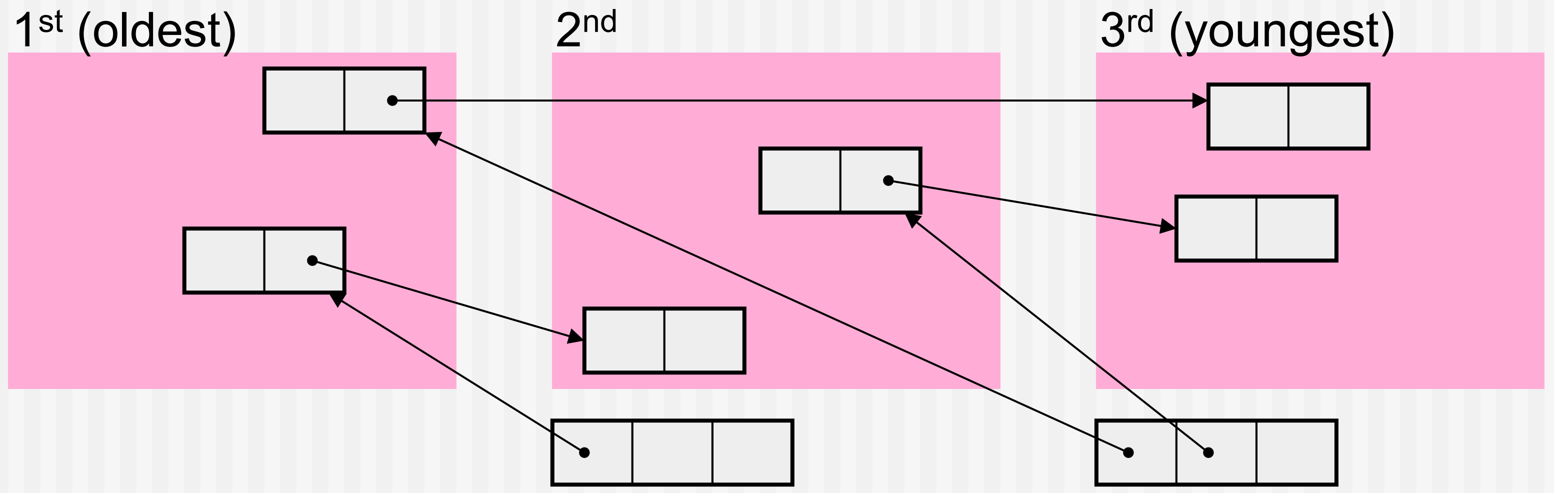
Generational GC

- Write barrier
  - Remove unnecessary barrier using a compiler.

```
aClass x = new aClass();
x.aField = aValue; // no barrier needed
```

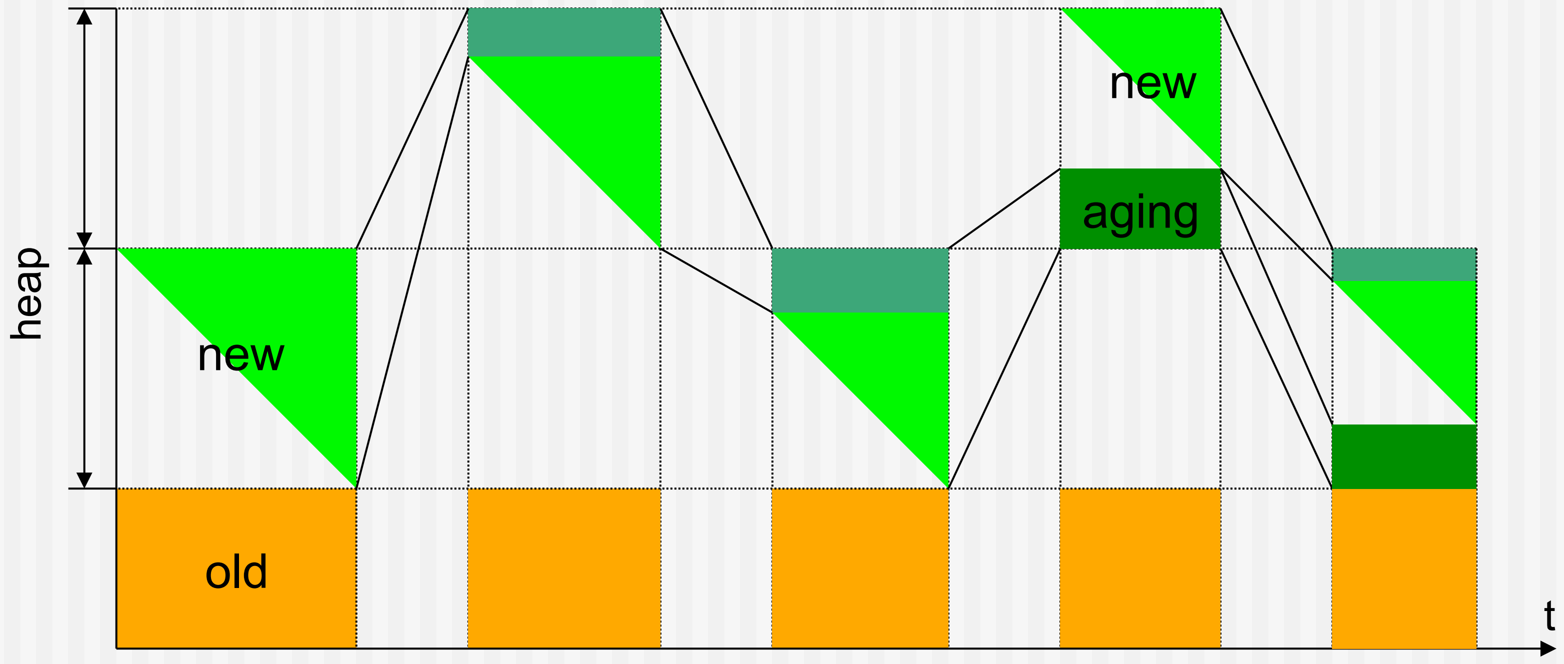
Generational GC

- Multiple generations
  - Frequency of GC: 1st (oldest) < 2nd < 3rd (youngest)
  - Prepare remember set for each generation
    Remove/Update entries during GC of old-generation.
  - Promotion: Move the object to older generation.
  - Tenure: Move the object to the oldest generation ⇒ Scarcely collected



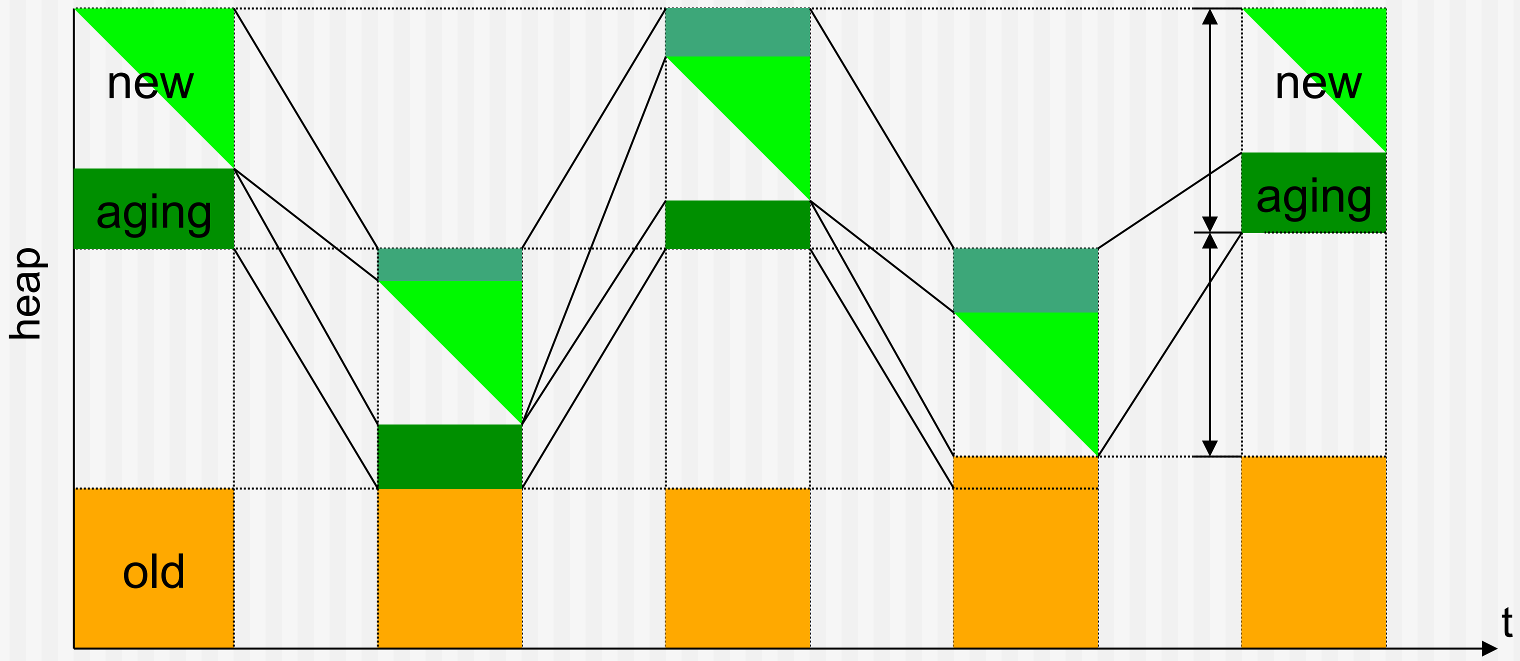
Generational GC

- Timing of promotion
  - When the object survives 1 or 2 GCs (ex. SML/NJ)
  - Associate a GC count with each object; When it exceeds N
  - Use of aging space



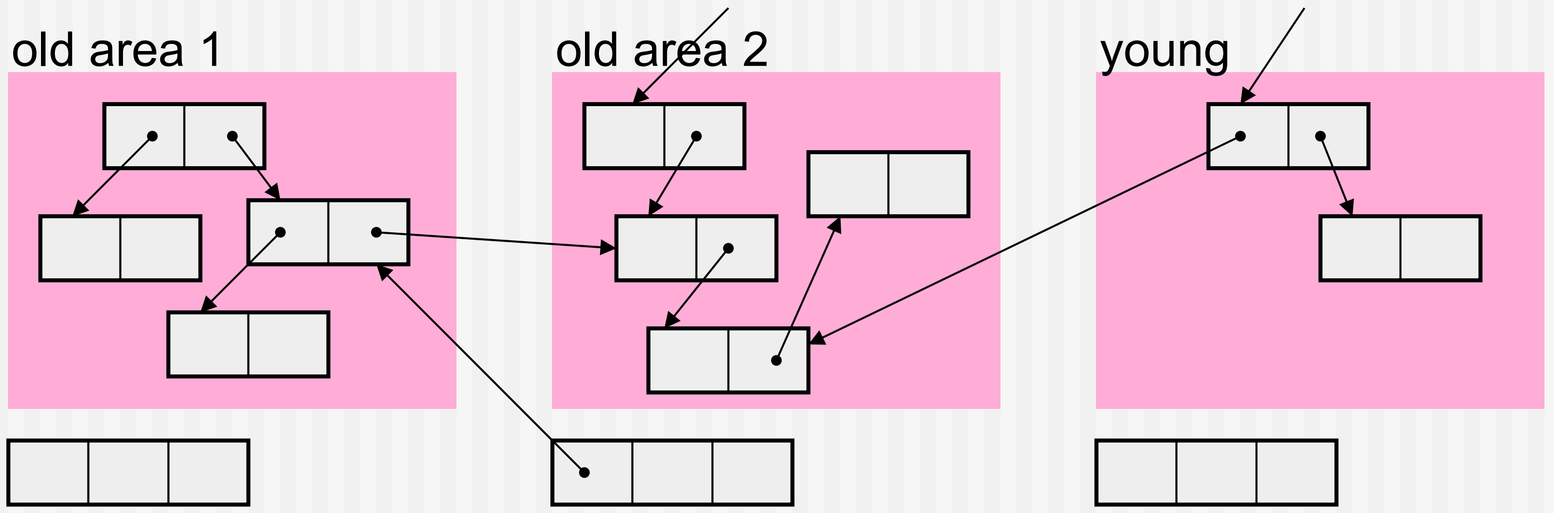
Generational GC

- Timing of promotion
  - Use of aging space



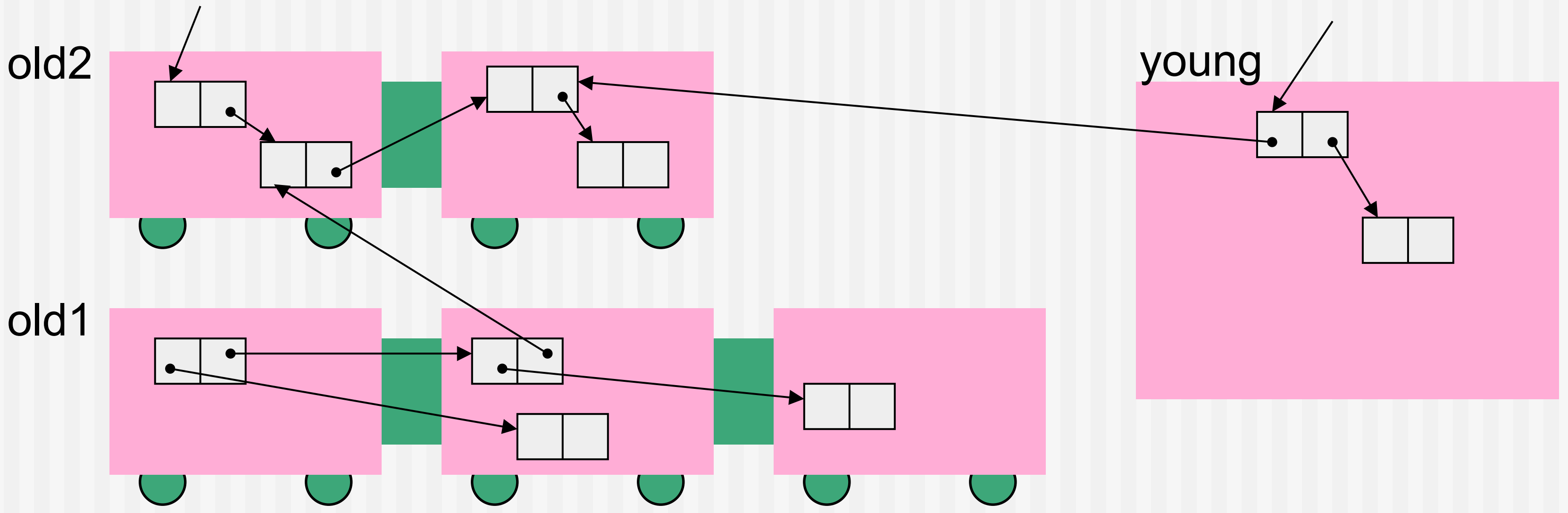
Train algorithm

- In generational GC, it is difficult to determine when to start major GC.
- Would like to carry out major GC little by little during minor GC.
- Distribute old objects in multiple areas.
  - Try to allocate a whole structure of objects in one area.
  - Scavenge one area at every K-times of minor GC.
  - Free the entire area when no objects are referred to from outside the area.



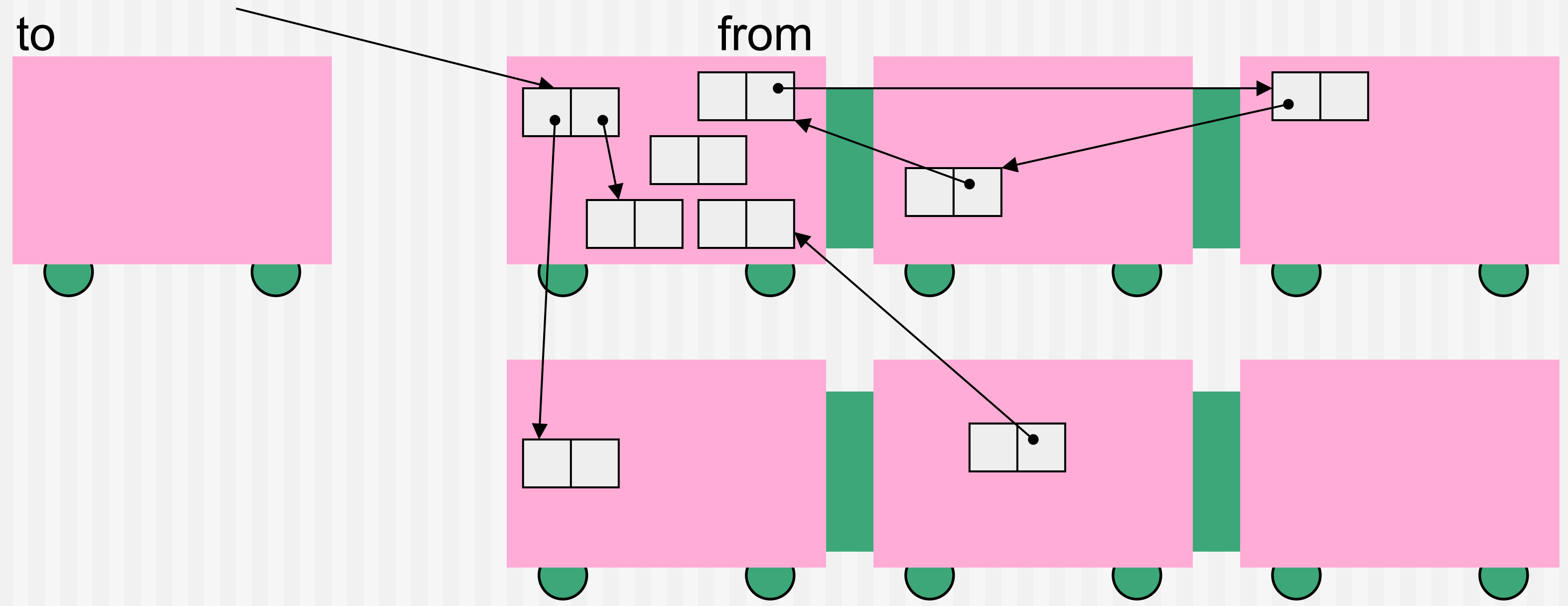
Train algorithm

- How large each area should be?
- Represent each area as a sequence ("train") of fixed-sized blocks ("cars").
  - Try to allocate a whole structure in a single train.
  - Scavenge one car at every K-times of minor GC.
  - Free the entire train when no objects are referred to from outside the train.



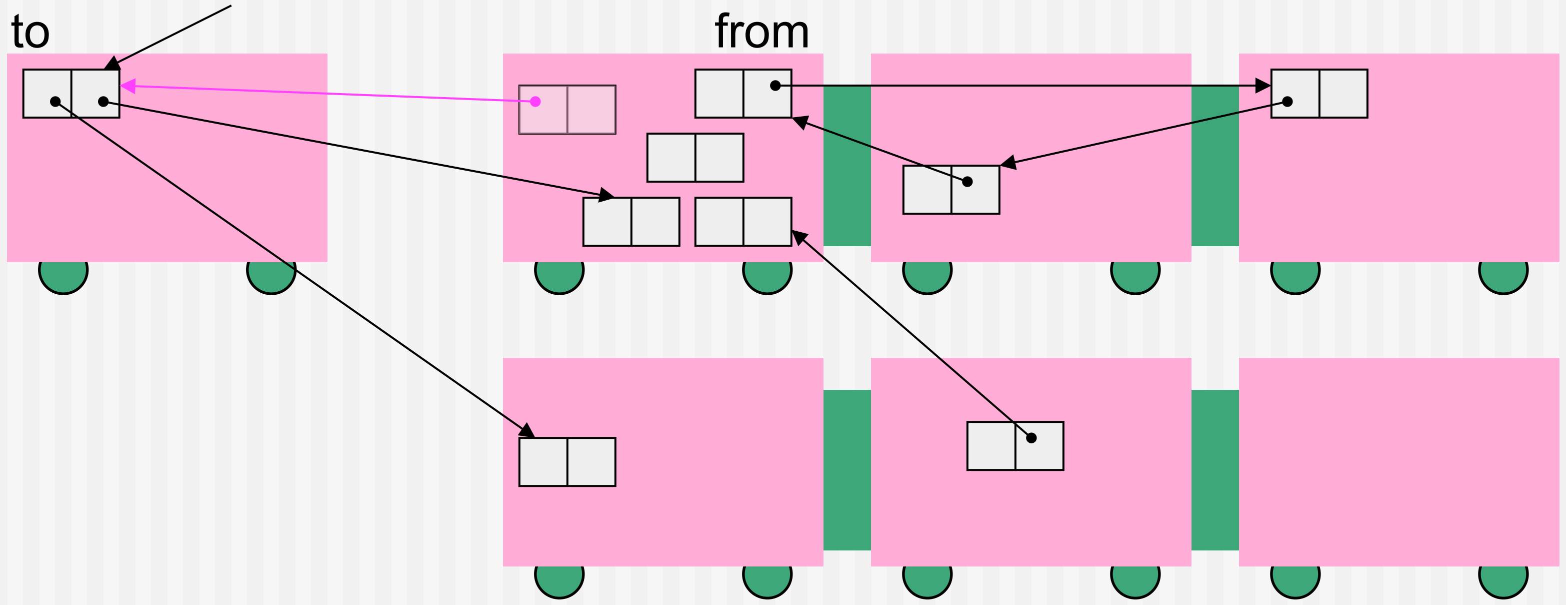
Train algorithm

1. While carrying out minor GC, copy objects in the "from" car that are pointed to from roots or live young objects, to the "to" car.
2. Scavenge from the "from" car to the "to" car.
3. Copy remaining objects that are pointed to from another train, to that train.
4. Copy remaining objects that are pointed to from another car, to that car.



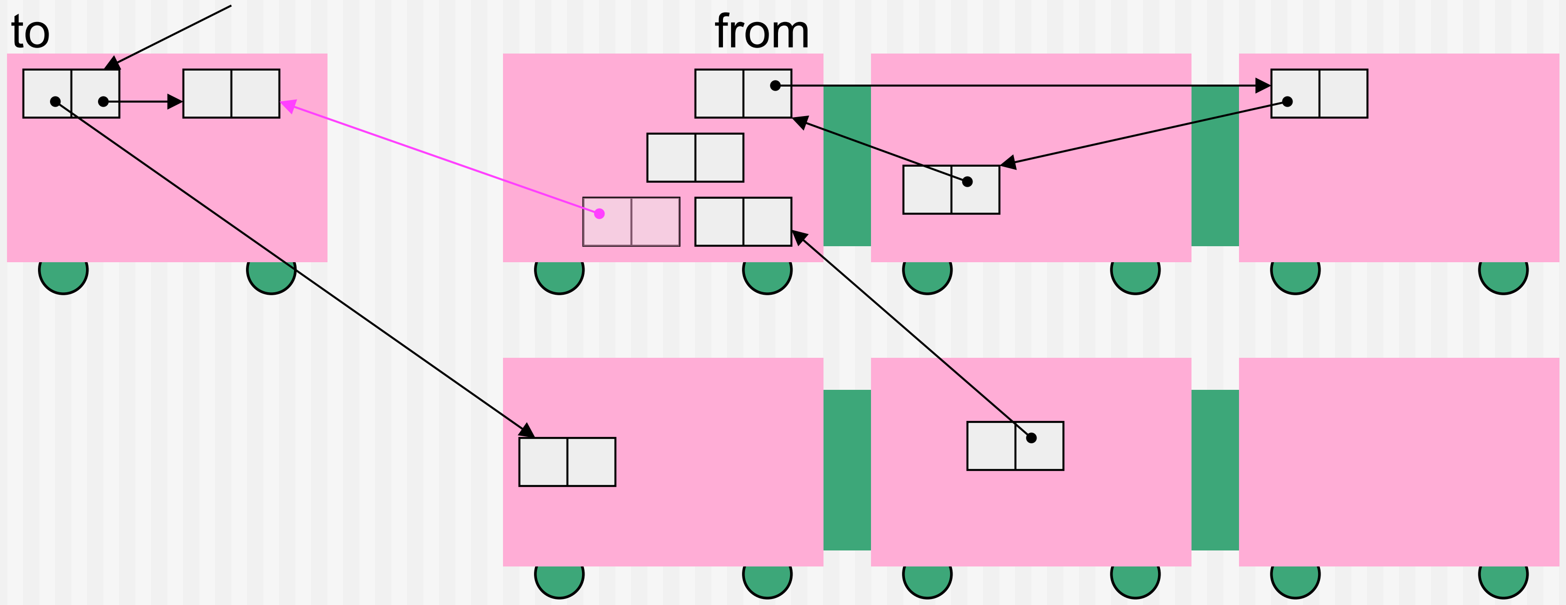
Train algorithm

1. While carrying out minor GC, copy objects in the "from" car that are pointed to from roots or live young objects, to the "to" car.
2. Scavenge from the "from" car to the "to" car.
3. Copy remaining objects that are pointed to from another train, to that train.
4. Copy remaining objects that are pointed to from another car, to that car.



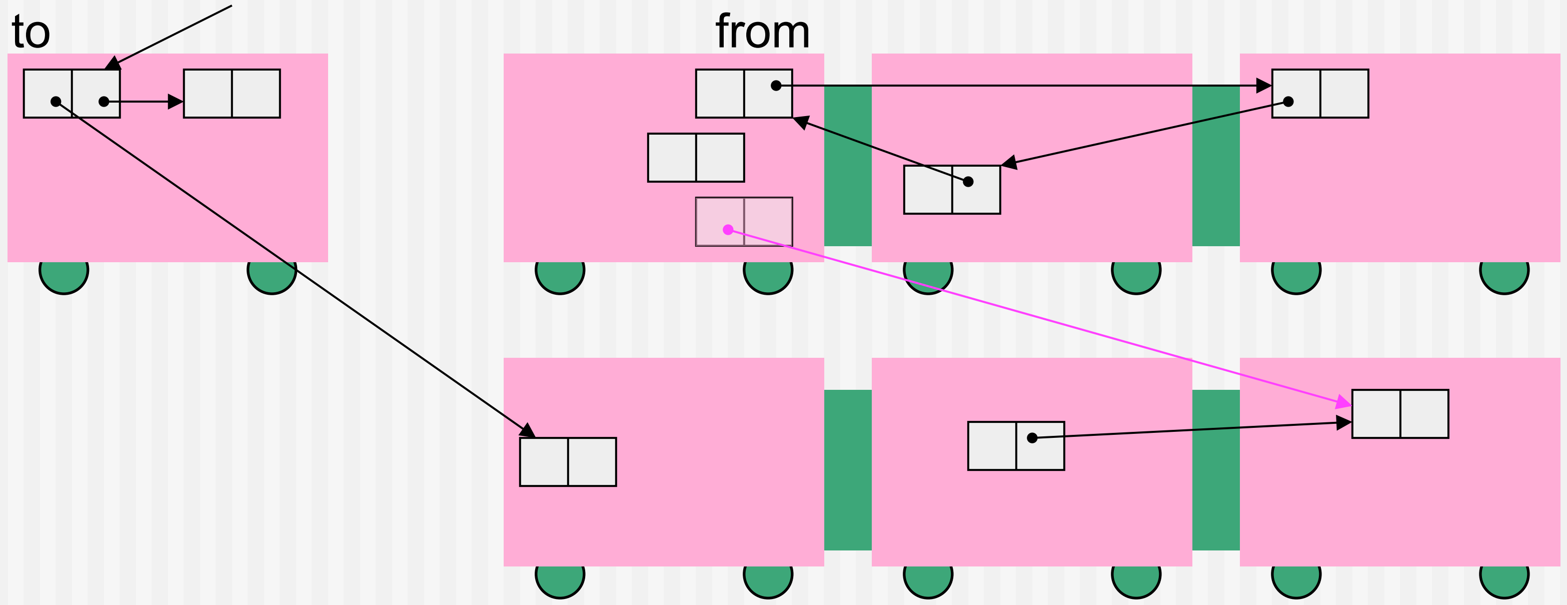
Train algorithm

1. While carrying out minor GC, copy objects in the "from" car that are pointed to from roots or live young objects, to the "to" car.
2. Scavenge from the "from" car to the "to" car.
3. Copy remaining objects that are pointed to from another train, to that train.
4. Copy remaining objects that are pointed to from another car, to that car.



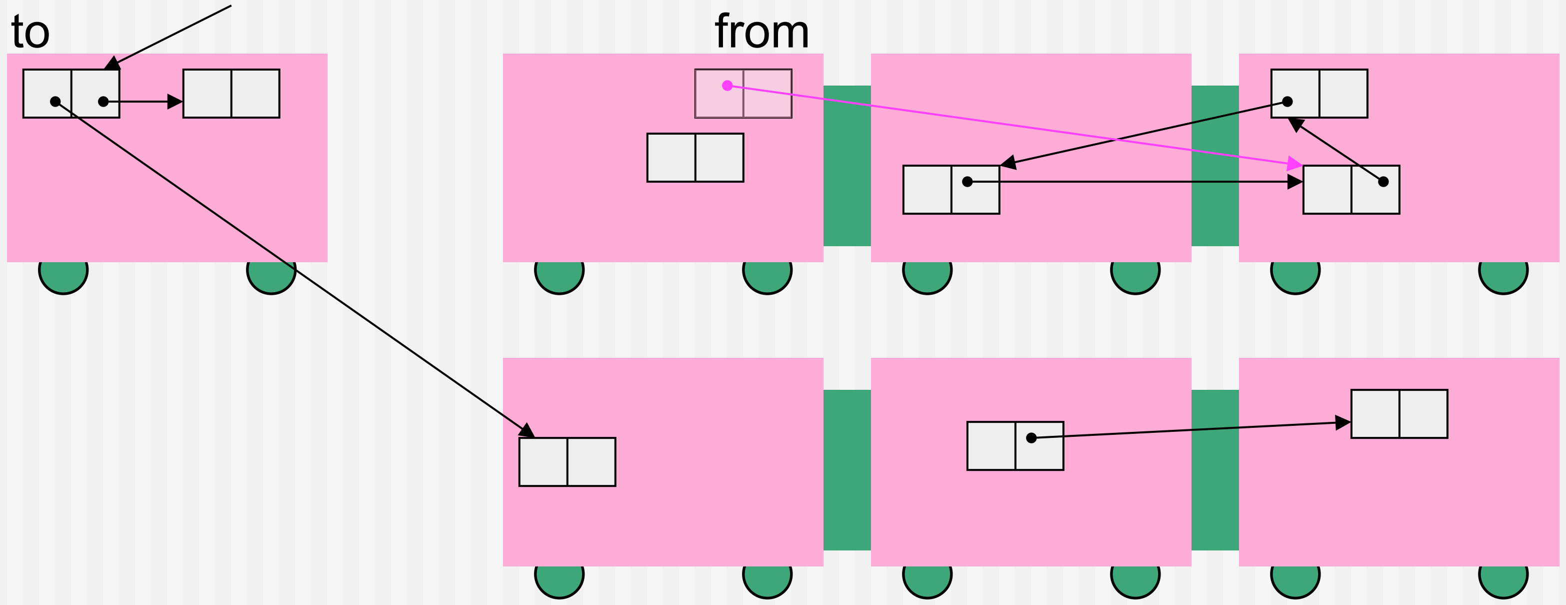
Train algorithm

1. While carrying out minor GC, copy objects in the "from" car that are pointed to from roots or live young objects, to the "to" car.
2. Scavenge from the "from" car to the "to" car.
3. Copy remaining objects that are pointed to from another train, to that train.
4. Copy remaining objects that are pointed to from another car, to that car.



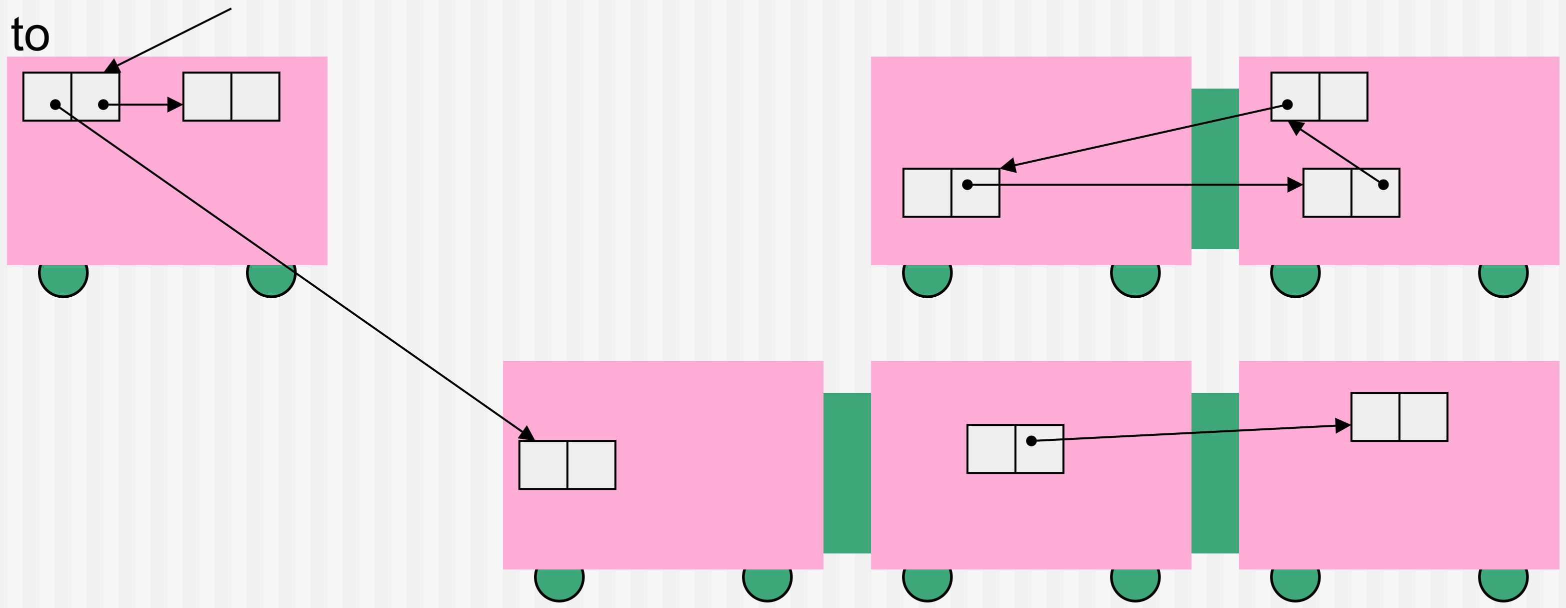
Train algorithm

1. While carrying out minor GC, copy objects in the "from" car that are pointed to from roots or live young objects, to the "to" car.
2. Scavenge from the "from" car to the "to" car.
3. Copy remaining objects that are pointed to from another train, to that train.
4. Copy remaining objects that are pointed to from another car, to that car.



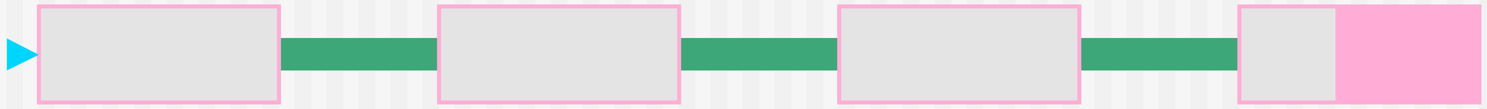
Train algorithm

1. While carrying out minor GC, copy objects in the "from" car that are pointed to from roots or live young objects, to the "to" car.
2. Scavenge from the "from" car to the "to" car.
3. Copy remaining objects that are pointed to from another train, to that train.
4. Copy remaining objects that are pointed to from another car, to that car.

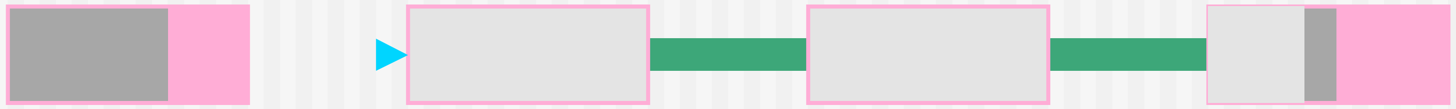


Train algorithm

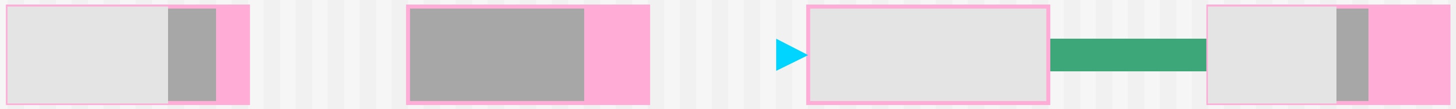
- Start with a single train.



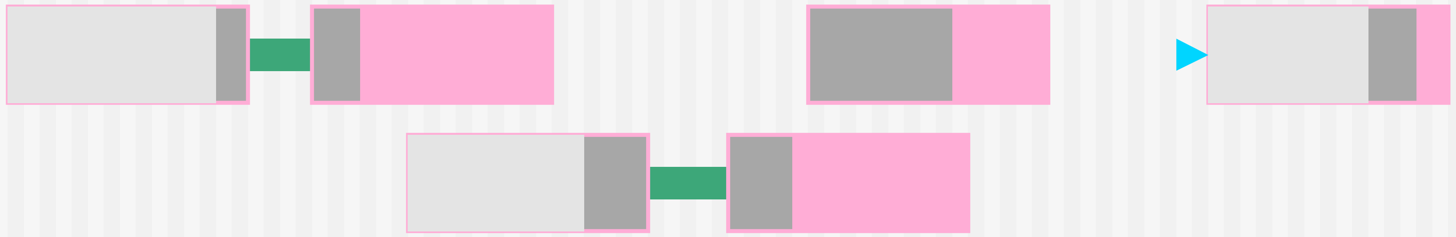
- The first car is disconnected by the 1st scavenge.



- Second scavenge...



- Third scavenge...



Boehm GC

Hans-J. Boehm

- Used as a library for C and C++ applications (including language systems)

  malloc ⇒ GC_malloc etc.
- Categories of objects
  - normal: contains pointers and is collected.
  - atomic: does not contain pointers.
  - uncollectable: contains pointers but is not collected.
  - stubborn: contains pointers and requires write barrier for *incremental marking* (explained later).
- Conservative GC, mark & sweep
- Roots:
  - Areas added by the user (uncollectable)
  - Registers ← by using setjump
  - Control stack of C ← by using a dummy variable

Users of Boehm GC (1/2)

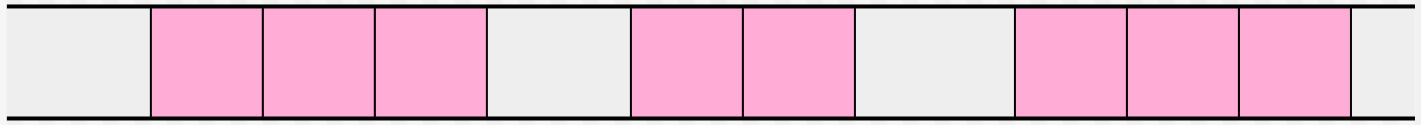
- The runtime system for GCJ, the static GNU java compiler.
- W3m, a text-based web browser.
- Some versions of the Xerox DocuPrint printer software.
- The Mozilla project, as leak detector.
- The Mono project, an open source implementation of the .NET development framework.
- The DotGNU Portable.NET project, another open source .NET implementation.
- The Irssi IRC client.
- The NAGWare f90 Fortran 90 compiler.
- Elwood Corporation's Eclipse Common Lisp system, C library, and translator.
- The Bigloo Scheme and Camloo ML compilers written by Manuel Serrano and others.
- Brent Benson's libscheme.

Users of Boehm GC (2/2)

- The MzScheme scheme implementation.
- The University of Washington Cecil Implementation.
- The Berkeley Sather implementation.
- The Berkeley Harmonia Project.
- The Toba Java Virtual Machine to C translator.
- The Gwydion Dylan compiler.
- The GNU Objective C runtime.
- Macauley 2, a system to support research in algebraic geometry and commutative algebra.
- The Vesta configuration management system.
- Visual Prolog 6.
- Asymptote LaTeX-compatible vector graphics language.

Heap management

- Heap is managed in blocks (4KB, ≒ page).
- Obtain memory space for blocks using malloc.
- Maintain a free list of unused blocks.



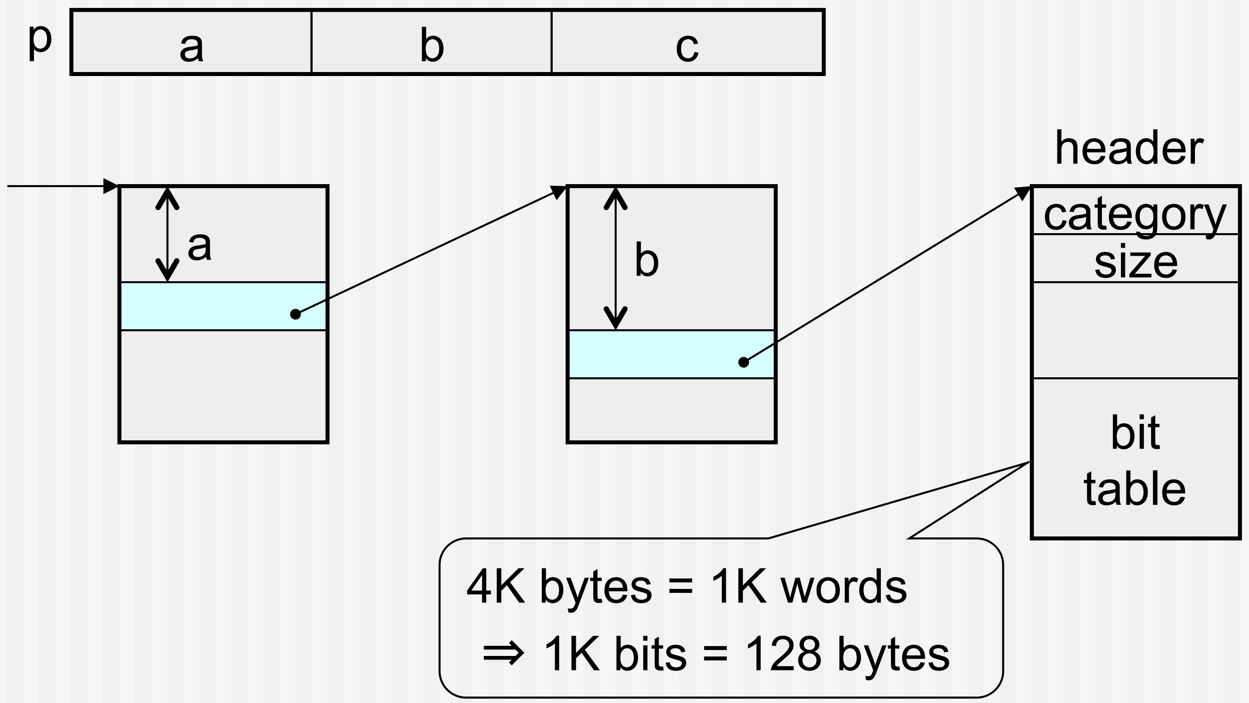
- Small objects (upto half the size of a block) ⇒ BiBOP
  - Maintain a free list for each category-size combination.
  - Objects of the same cat. and size are allocated in each block.
- Large objects
  - Allocate in multiple consecutive blocks.

Object allocation

- Small objects
  1. Free list for objects of the specified cat. and size
  2. GC
  3. Allocation of unused blocks
  4. Expansion of heap
- Large objects
  1. Search the required number of consecutive blocks from the free block list.
  2. Expansion of heap
- Expand heap if GC is frequently carried out.

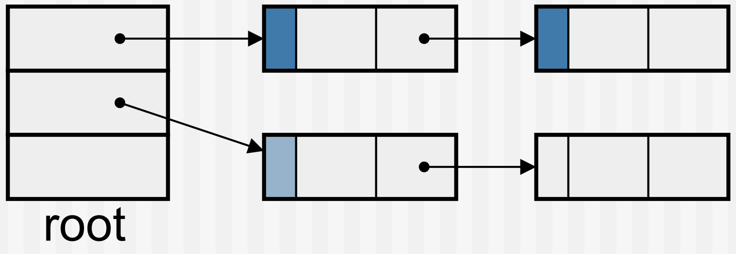
Block header

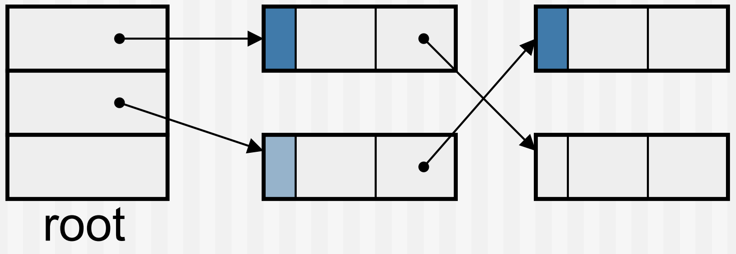
- Store information for each block.
- Two-step retrieval from address (data that might be an address)



Mark

- stop-the-world
- incremental
  - Continue marking process for a predetermined time (50 msec).
  - If the marking is not completed, carry out marking little by little, at each object allocation.
  - After the incremental marking, re-mark all the objects at once.
    - From the marked objects in those blocks that are modified during the incremental marking

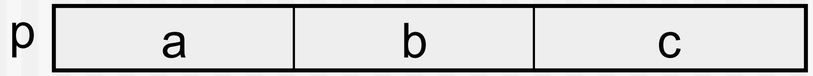




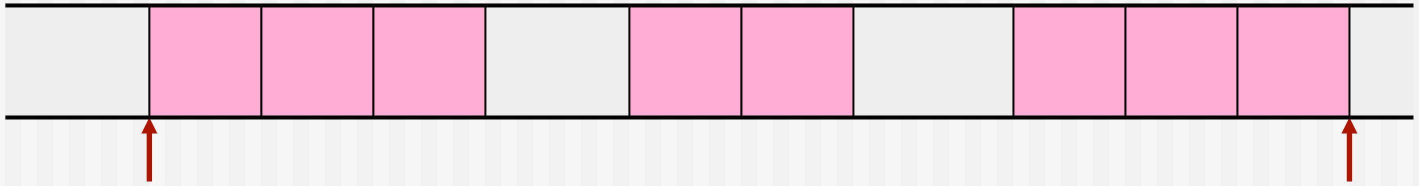
Detection of block modification

- Most of Unix and Win32
  - Use memory protection mechanism of OS
  - Set write protection for pages of entire blocks at the start of GC.
  - A signal is generated when writing occurs.
  - Upon receiving a signal, set the dirty bit (hash table) for the page, and release the write protection of the page.
- Solaris
  - Retrieve dirty bit information from /proc.
- Others
  - Write barrier for stubborn.

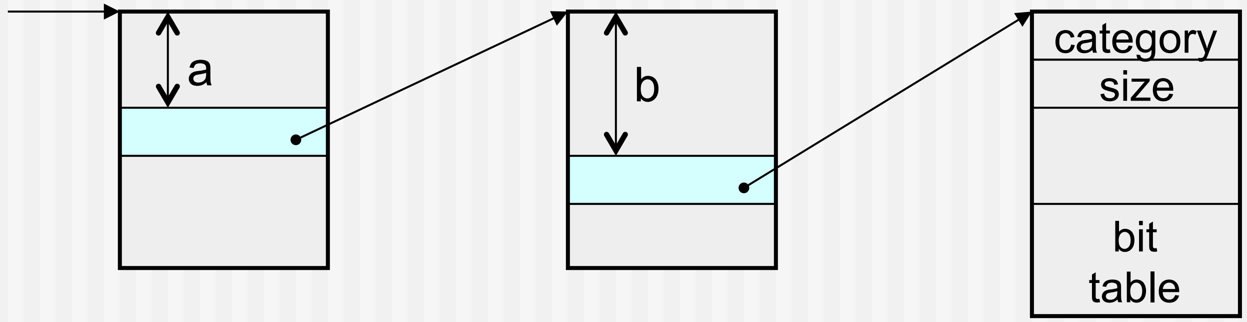
Decision of pointer possibility and marking



- Whether within the boundary of heap area or not?



- Search for a block header using significant bits (a & b)
- Compute the mark bit location using the size information in the header.



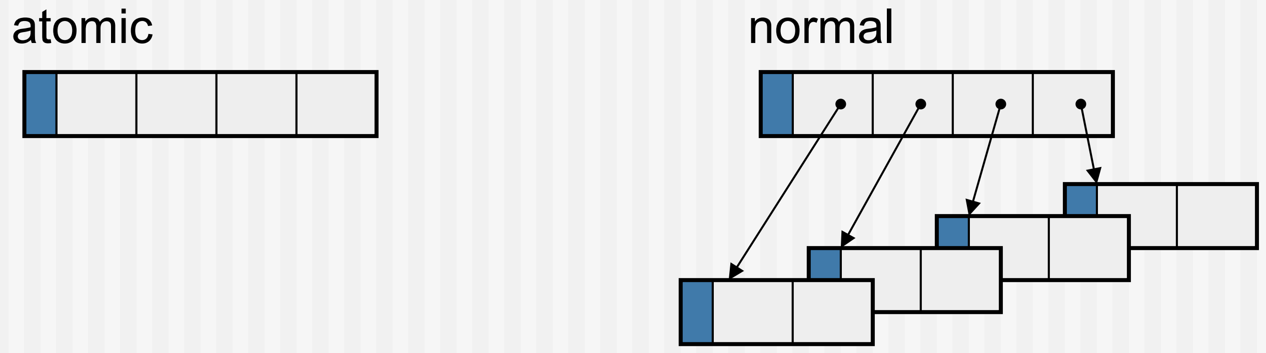
Sweep

- reclaim list: for each category and size
  - Link un-swept blocks.
- At the completion of marking
  - Add unused blocks to the free block list.
  - Add used blocks to an appropriate reclaim list.
  - Access only block headers ⇒ short operation
- While allocating objects
  - When the free list of the required category and size is empty, sweep blocks in the reclaim list of the cat. and size, one by one.
  - When the free list becomes non-empty, stop sweeping.

Black list

- To reduce the effect due to misjudge of pointers
- "Near-miss" of pointers
  - In pointer possibility decision, the header of an unused block is reached.
- Register the unused block of "near-miss" on the black list.
- Use blocks on the black list exclusively for atomic objects, which do not include a pointer.

  ⇒ Even misjudged, only the pointed object is marked by mistake.



Other topics related to GC

- Multithread GC
- Parallel GC
- GC in distributed environment
- Real-time GC
- Support of OS/HW

Reference

- Richard Jones, Antony Hosking, Eliot Moss, The Garbage Collection Handbook: The Art of Automatic Memory Management, CRC Press, ISBN 978-1-4200-8279-1, 2012

http://www.cs.ukc.ac.uk/people/staff/rej/gc.html

Richard Jones