Garbage Collection

Akim Demaille, Etienne Renault, Roland Levillain

June 4, 2019

Table of contents

Garbage Collection 1/2

- Fisrt apparition in LISP, 1959, McCarthy
- Garbage collection is the automatic reclamation of computer storage (heap) at **runtime**
- Automatic memory management
  - New/malloc doesn't need delete/free anymore
  - Necessary for fully modular programming.
    Otherwise some modules are responsible for allocation while others are responsible for deallocation.
  - No more memory leaks
  - Avoid dangling-pointers/references.
    Reclaiming memory too soon is no more possible

Garbage Collection 2/2

- Quite expensive relative to explicit heap management
  - Slow running programs down by (very roughly) 10 percent...
  - ... But sometime cheaper or competitive
  - Fair comparison is difficult since explicit deallocation affects the structure of programs in ways that may themselves be expensive
- Possible reduction of heap fragmentation
- Functional and logic programming languages generally incorporate garbage collection because their unpredictable execution patterns
- D, Python, Caml, Effeil, Swift, C#, Go, Java, Haskell, LISP, Dylan, Prolog, etc.

What is Garbage?

- An object is called garbage at some point during execution if it will never be used again.
- What is garbage at the indicated points?

```
int main() {
  Object x, y;
  x = new Object();
  y = new Object();
  /* Point A */
  x.doSomething();
  y.doSomething();
  /* Point B */
  y = new Object();
  /* Point C */
}
```

Approximating Garbage

- In general, it is undecidable whether an object is garbage
- An object is reachable if it can still be referenced by the program.

Goals

Detect and reclaim unreachable objects

Basics of a Garbage Collector

1. Distinguishing the live objects from the garbage ones
2. Reclaiming the garbage object' storage

Basics of a Garbage Collector

1. Distinguishing the live objects from the garbage ones
2. Reclaiming the garbage object' storage

We focus on built-in garbage collectors so that:

- allocation routines performs special actions
  - reclaim memory
  - emit specific code to recognize object format
  - *etc.*
- explicit calls to the deallocator are unnecessary
  - the allocator will call it on-time
  - the objects will be automatically destroyed

Different kind of GC

- Incremental techniques:
  - allow garbage collection to proceed piecemeal while application is running
  - my provide real-time garantees
  - can be generalized into concurrent collections
- Generationnal Schemes
  - improve efficiency/locality by garbage collecting a smaller area more often
  - avoid overhead due to long time objects
  - rely on pause to collect data

Table of contents

Reference Counting

Intuition

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object
- Each time a reference to the object is created, increase the pointed-to object's counter

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object
- Each time a reference to the object is created, increase the pointed-to object's counter
- Each time an existing reference to an object is eliminated, the counter is decremented

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object
- Each time a reference to the object is created, increase the pointed-to object's counter
- Each time an existing reference to an object is eliminated, the counter is decremented
- When the object counter equals zero, the memory can be reclaimed

Deallocation

Caution

When an object is destructed:

Transitive reclamation can be deferred by maintaining a list of freed objects

Deallocation

Caution

When an object is destructed:

- examines pointer fields

Transitive reclamation can be deferred by maintaining a list of freed objects

Deallocation

Caution

When an object is destructed:

- examines pointer fields
- for any references R contained by this object, decrement reference counter of R

Transitive reclamation can be deferred by maintaining a list of freed objects

Deallocation

Caution

When an object is destructed:

- examines pointer fields
- for any references R contained by this object, decrement reference counter of R
- If the reference counter of R becomes 0, reclaim memory

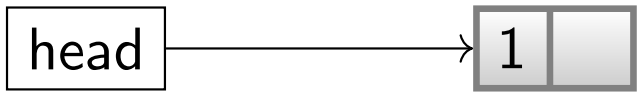
Transitive reclamation can be deferred by maintaining a list of freed objects

Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {




}
```

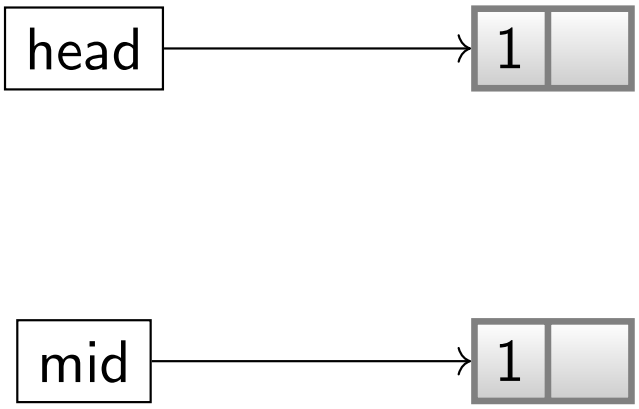
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;









}
```



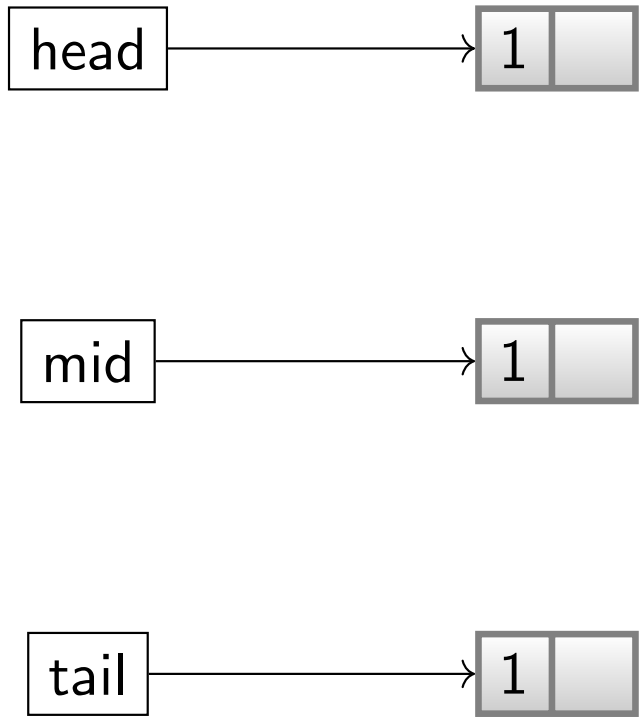
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;




}
```



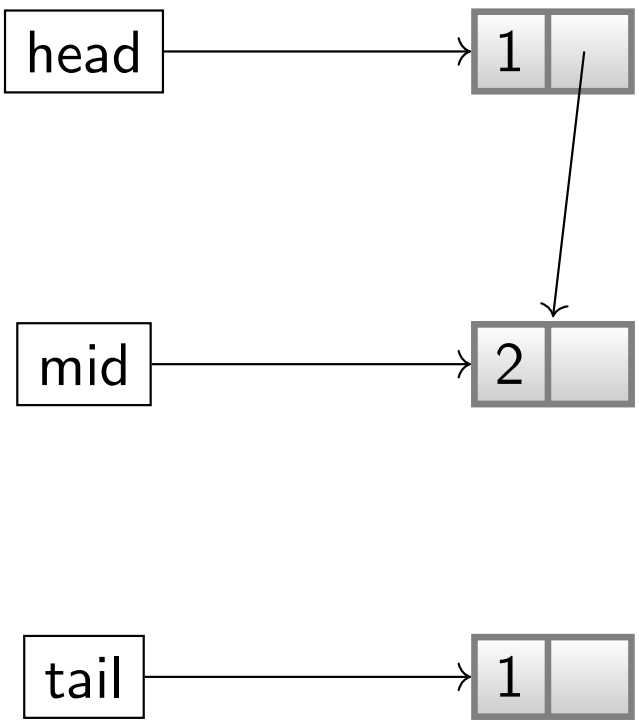
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;




}
```



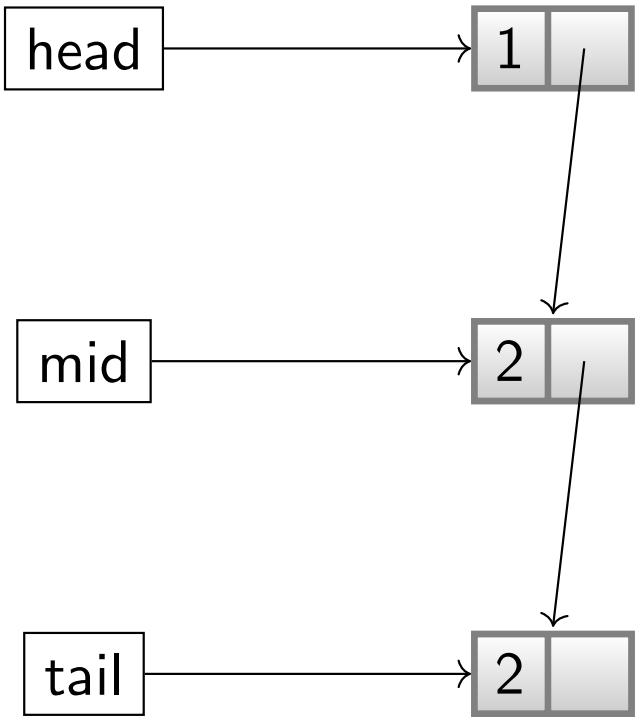
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;




}
```



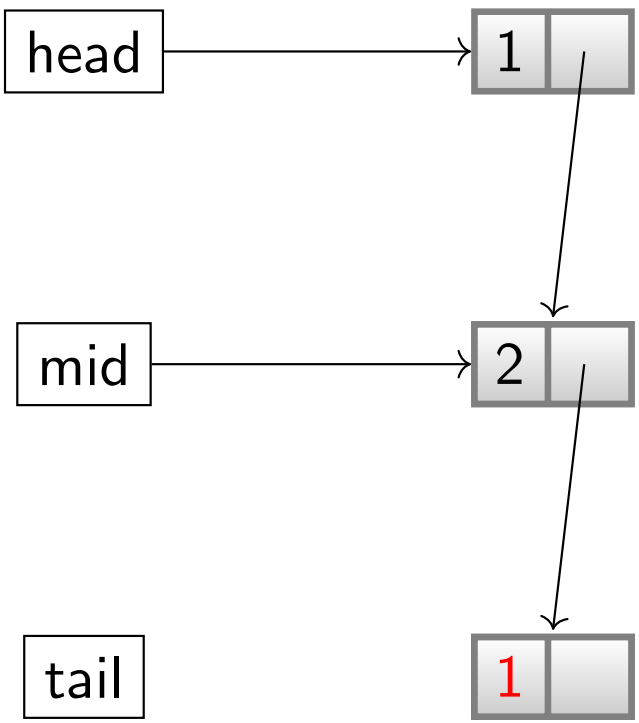
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;



}
```



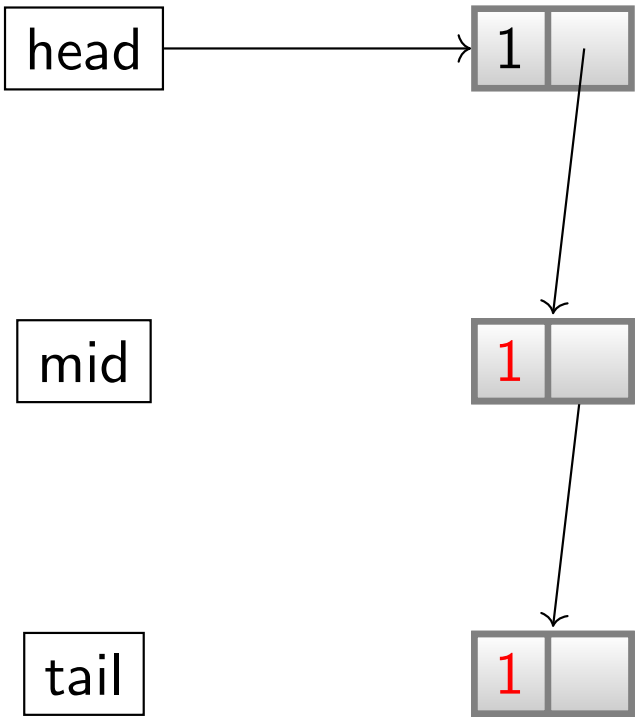
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;


}
```



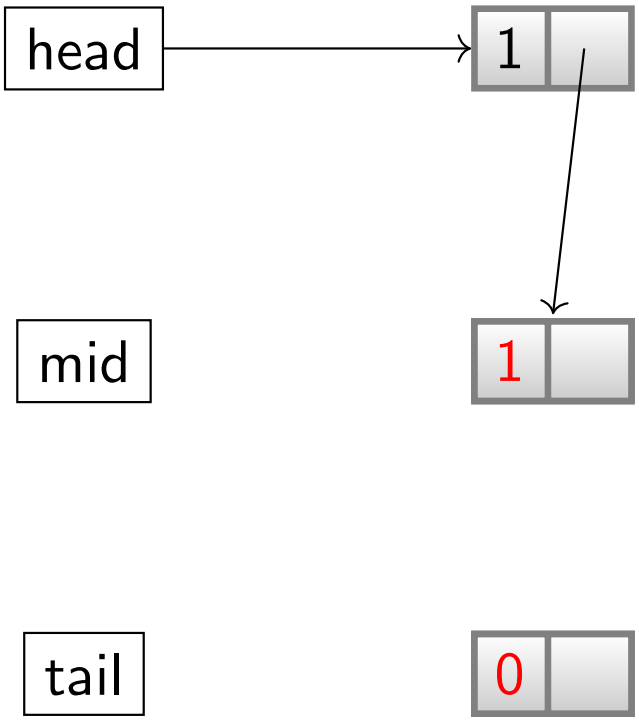
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;


}
```



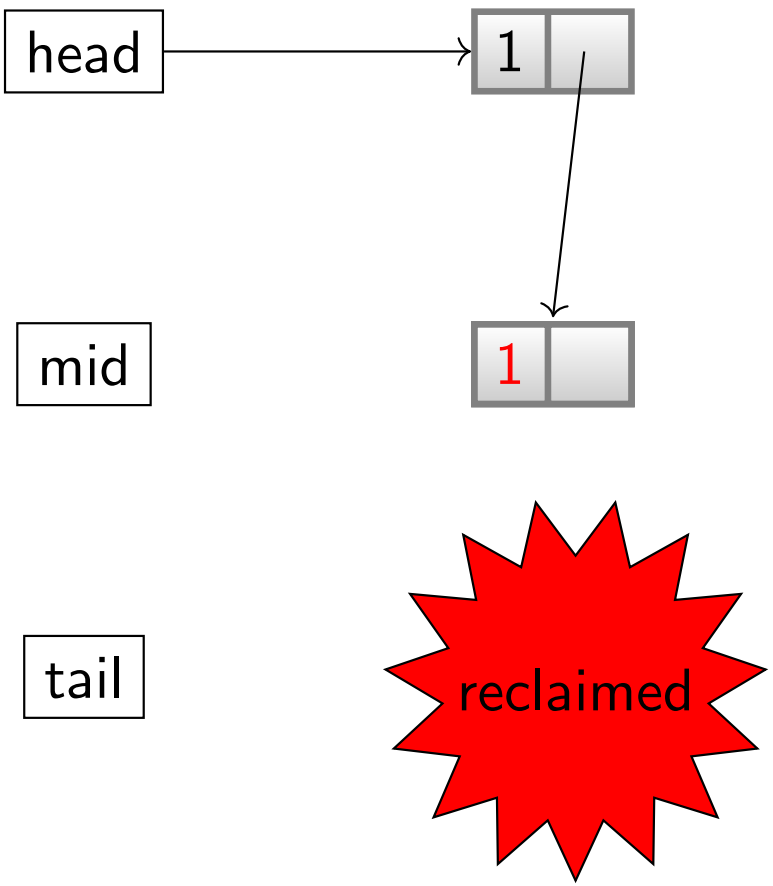
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;

}
```



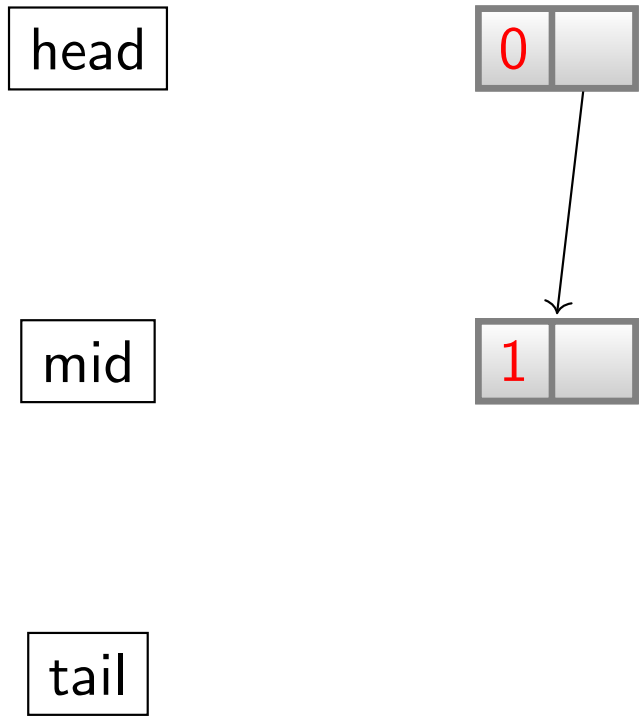
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;

}
```



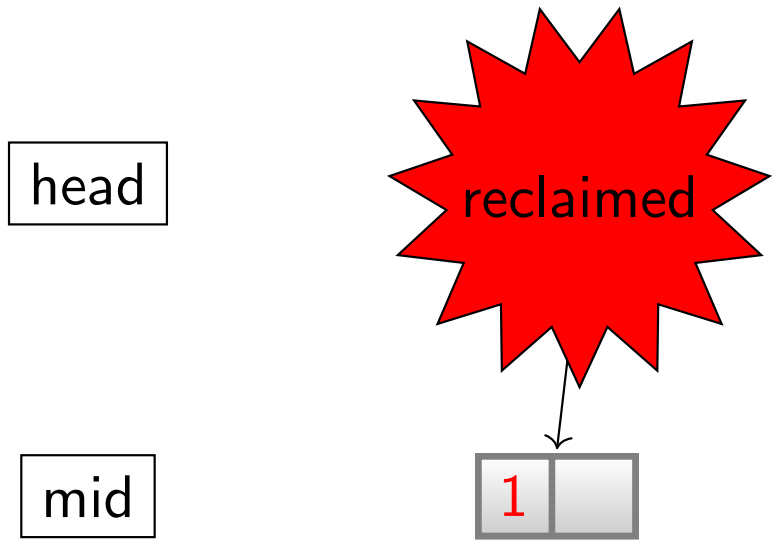
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



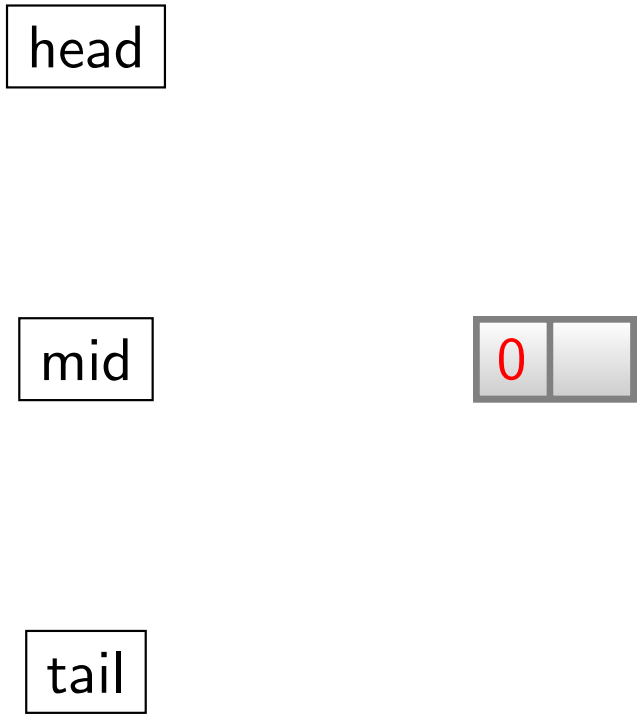
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



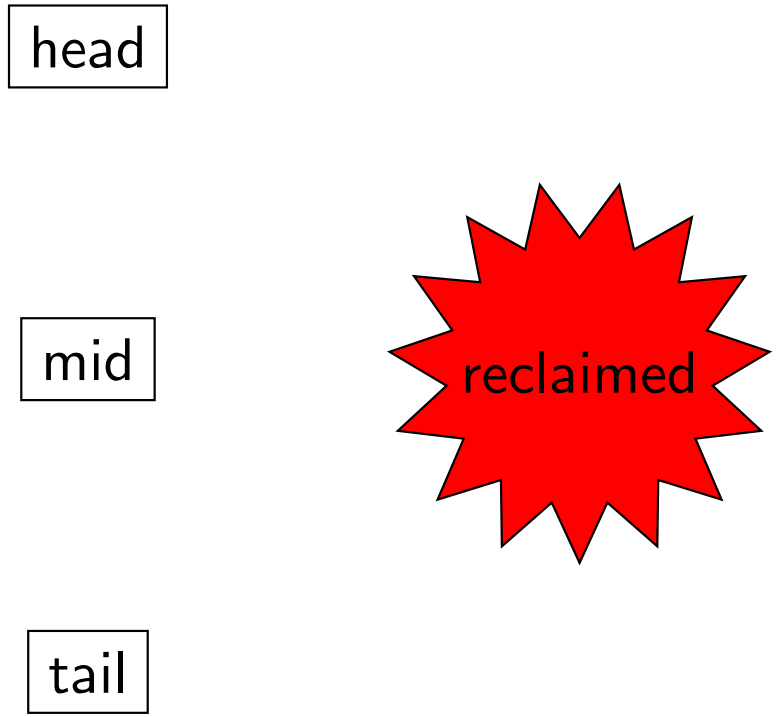
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



What about cyclic references 1

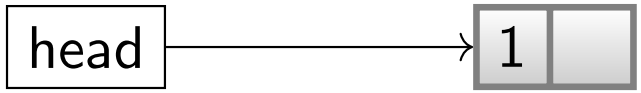
If the objects create a directed cycle, the objects references counters will never reduced to zero.

```
class LinkedList {
  LinkedList next = null;
}
int main() {









}
```

What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

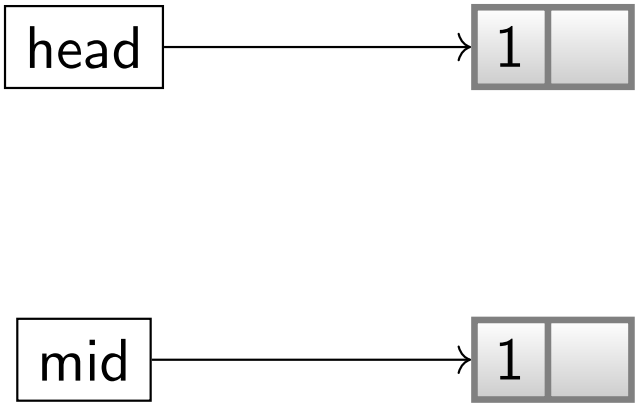
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;









}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

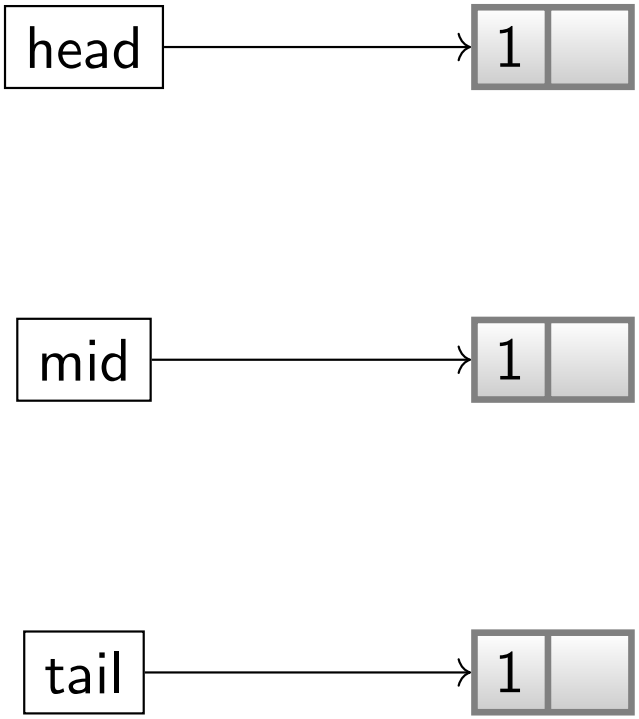
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;




}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

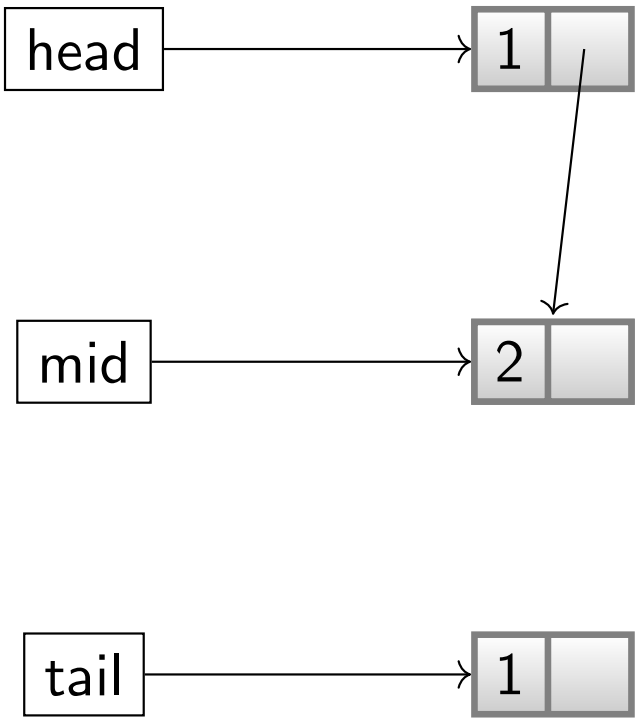
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;









}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

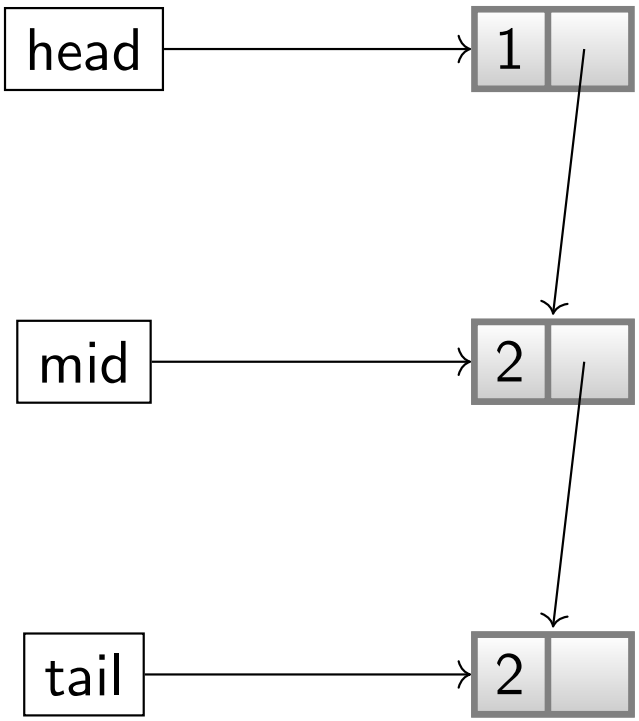
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;




}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

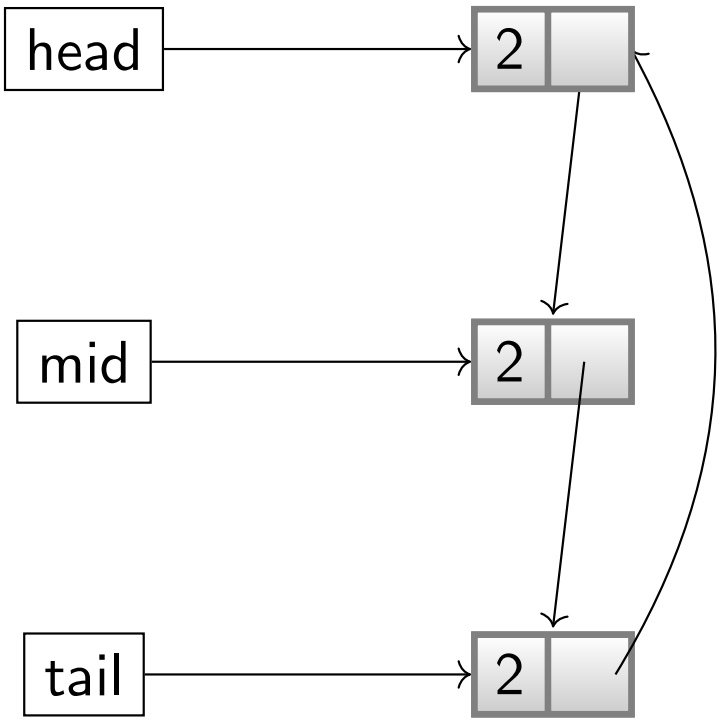
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;




}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

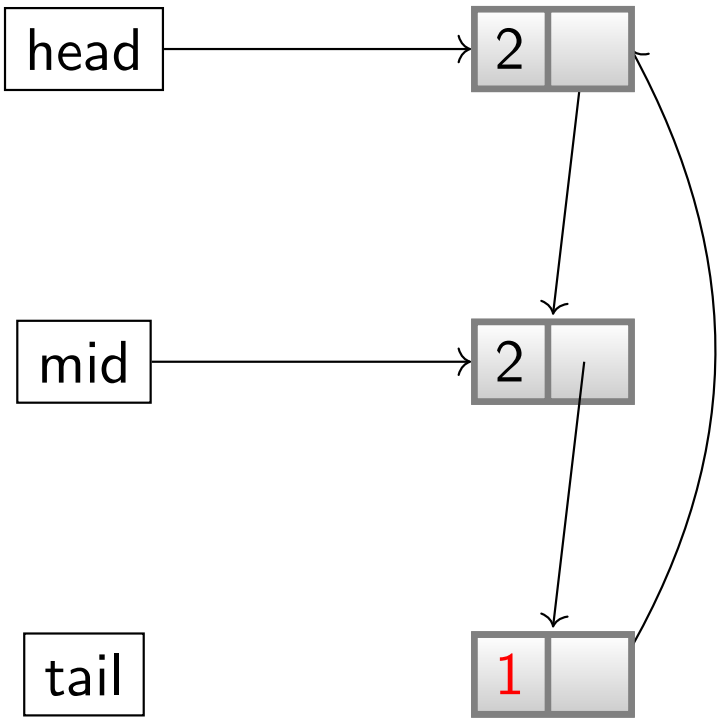
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;



}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

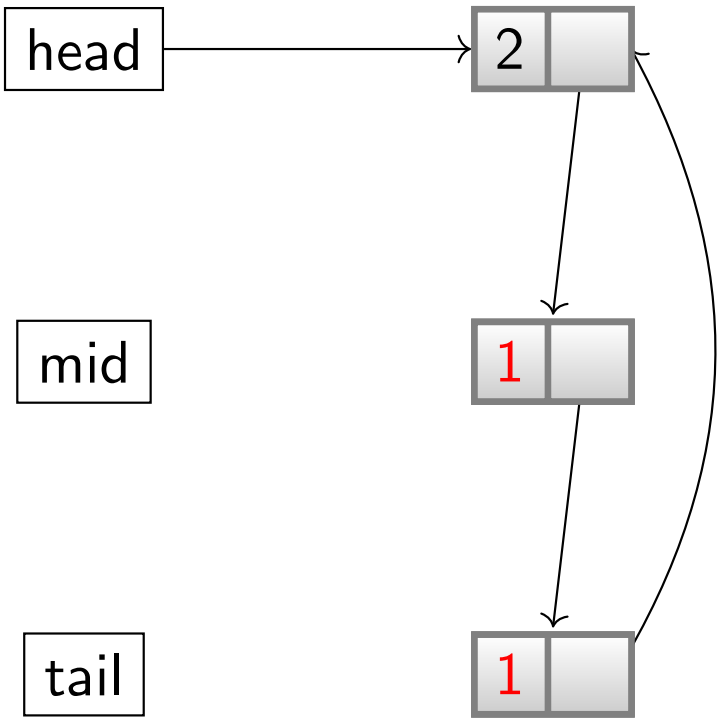
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;
 tail = null;


}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

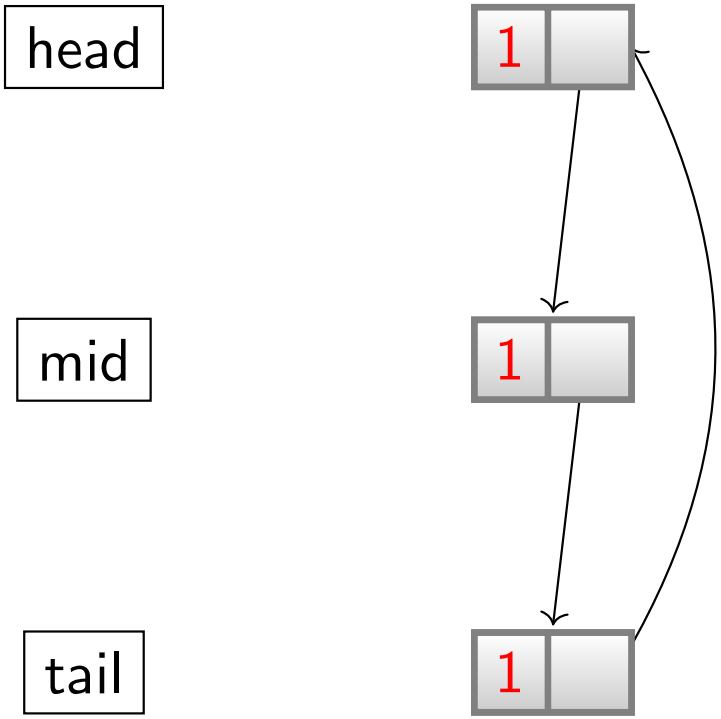
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;
 tail = null;
 mid = null;

}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;
 tail = null;
 mid = null;
 head = null;
}
```



Pros and Cons

Pros:

- Easy to implement: perl, Firefox
- Can be implemented on top of explicit memory management librairies (shared_ptr)
- Interleaved with running time
- Small overage per unit of program execution
- Transitive reclamation can be deferred by maintaining a list of freed objects
- Real-time requierements: no halt of the system.
  Necessary for application where response-time is critical

Cons:

- A whole machine word per object
- When the number of references to an object overflows, the counter is set to the maximum and the memory will never be reclaimed
- Problem with cycles
- Efficiency: cost relative to the running program

Table of contents

Analysis

- Reference counting tries to find unreachable objects by finding objects without incoming references
- These references have been forgotten !

Analysis

- Reference counting tries to find unreachable objects by finding objects without incoming references
- These references have been forgotten !

We have to trace the lifetime of objects

Intuition

Given knowledge of what's immediately accessible, find everything reachable in the program

The root set is the set of memory locations in the program that are known to be reachable

Graph Problem

Simply do a graph search starting at the root set:

- Any objects reachable from the root set are reachable
- Any objects not reachable from the root set are not reachable

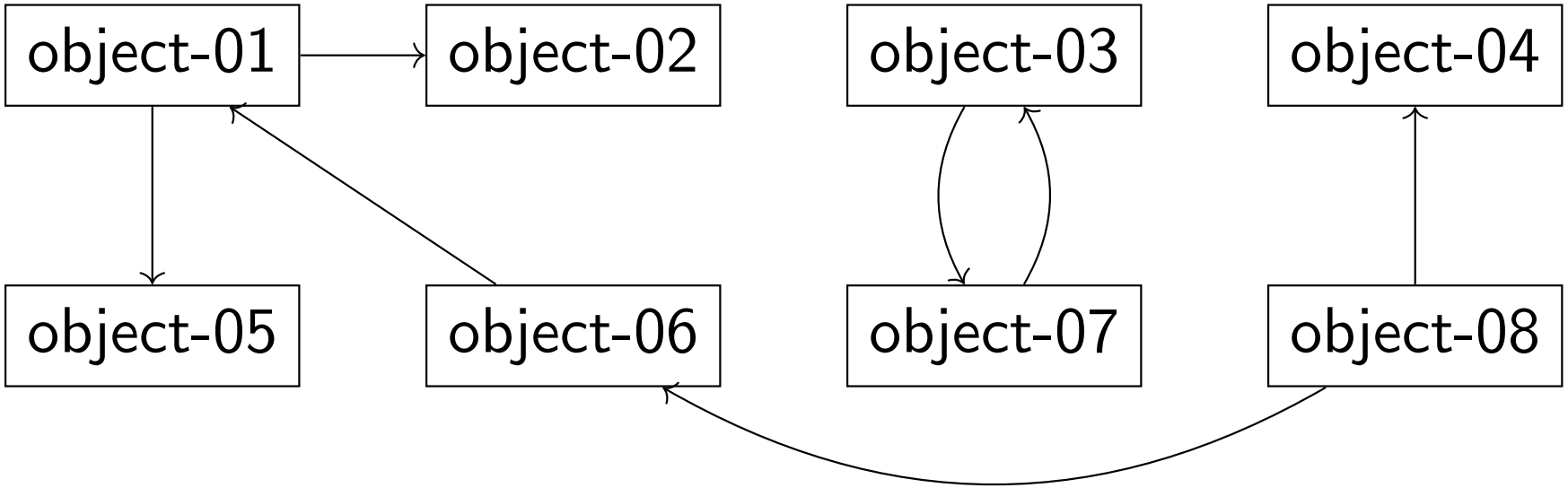
How to obtain the root set?

- static reference variables
- references registered through librairies (JNI, for instance)
- For each threads:
  - local variables
  - current method(s) arguments
  - stack
  - etc.

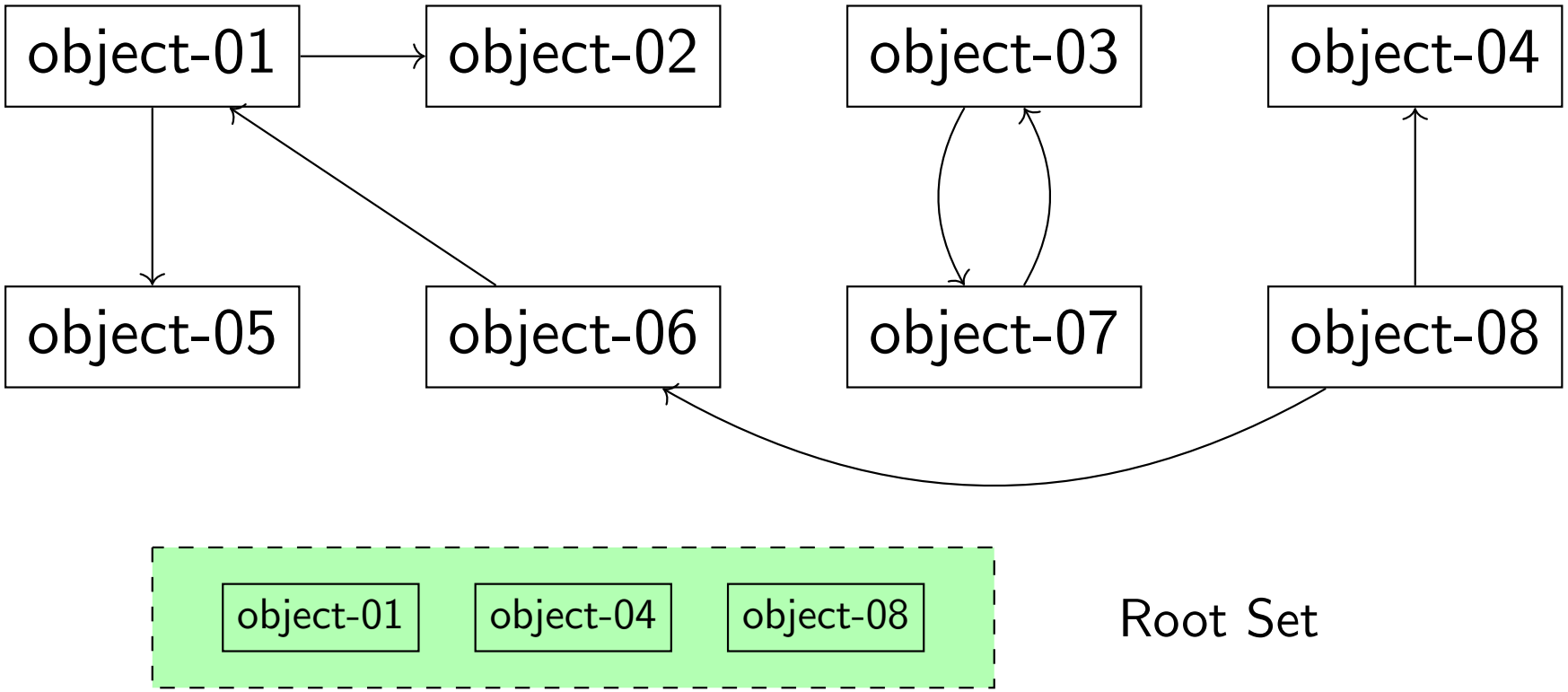
Mark-and-Sweep: the Algorithm

1. Marking phase: Find reachable objects
   - Add the root set to a worklist
   - While the worklist isn't empty
     - ★ Remove an object from the worklist
     - ★ If it is not marked, mark it and add to the worklist all objects reachable from that object
2. Sweeping phase: Reclaim free memory
   - If that object isn't marked, reclaim its memory
   - If the object is marked, unmark it

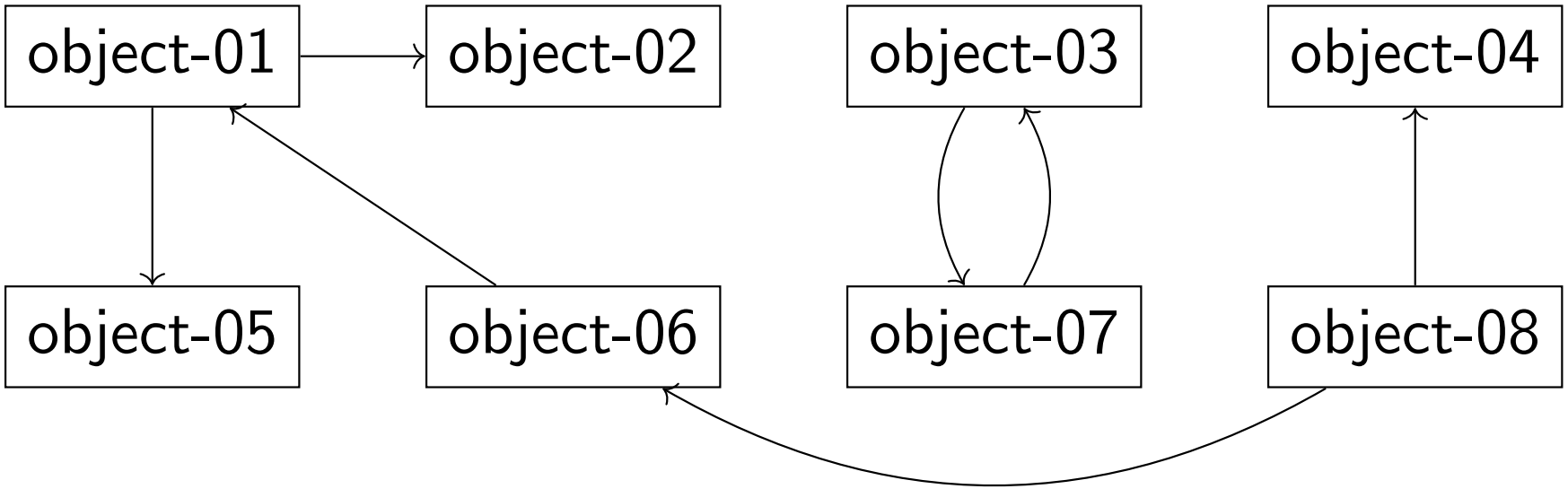
Example

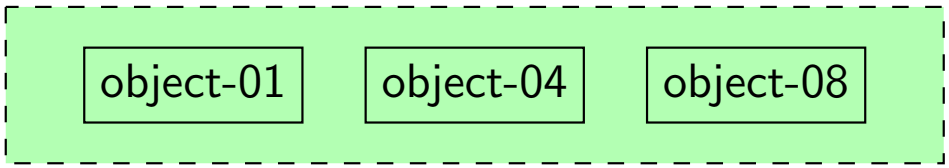

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08

Example

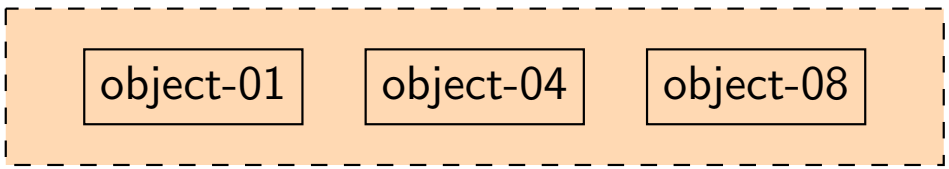

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set

Example



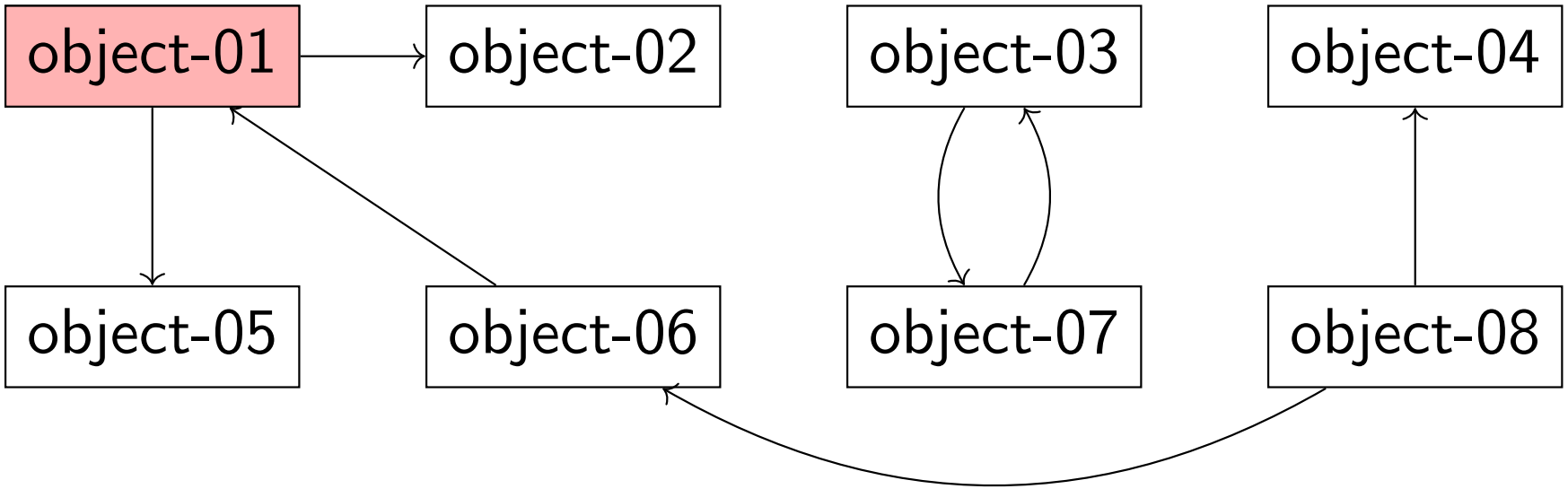


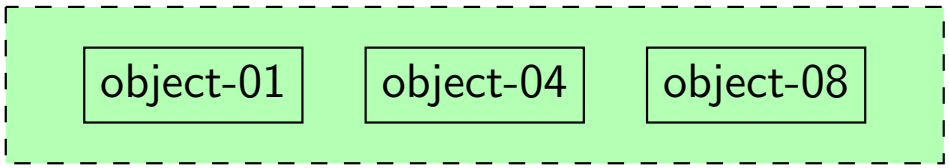
Root Set



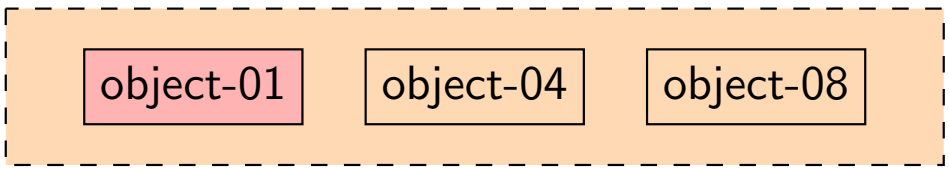
Working Set

Example



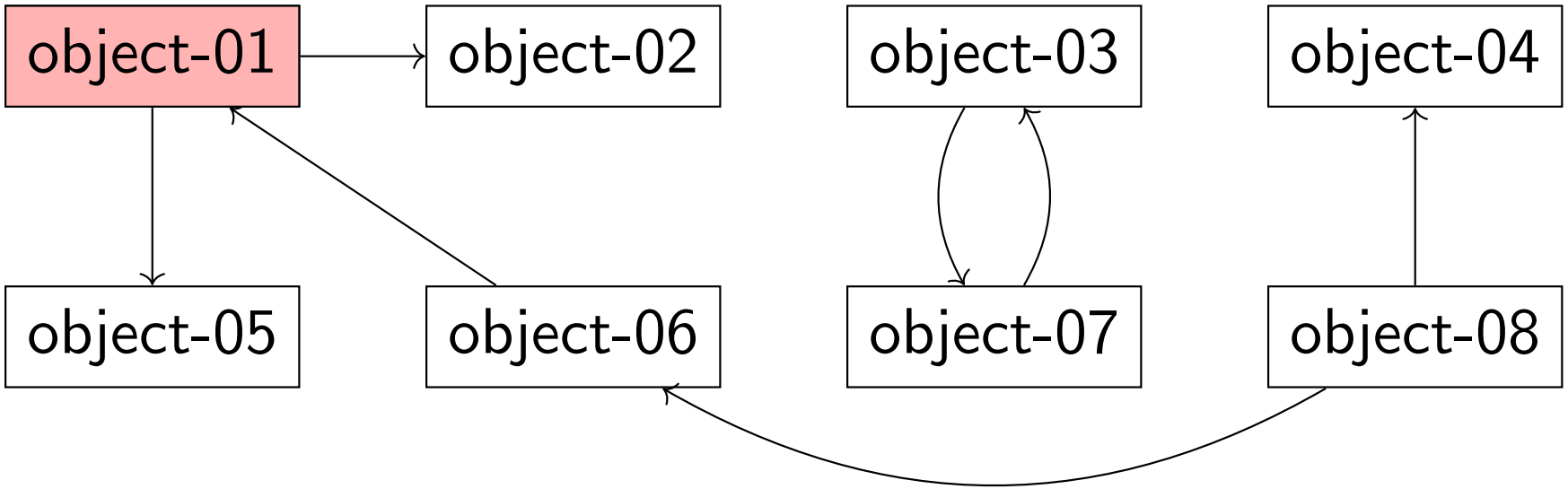


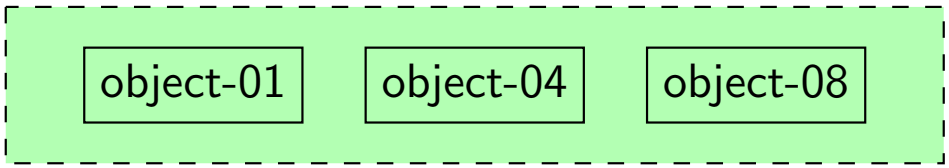
Root Set



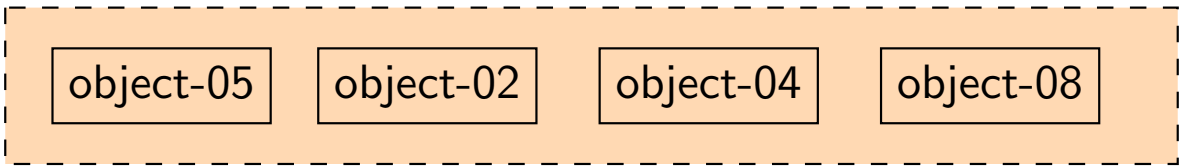
Working Set

Example



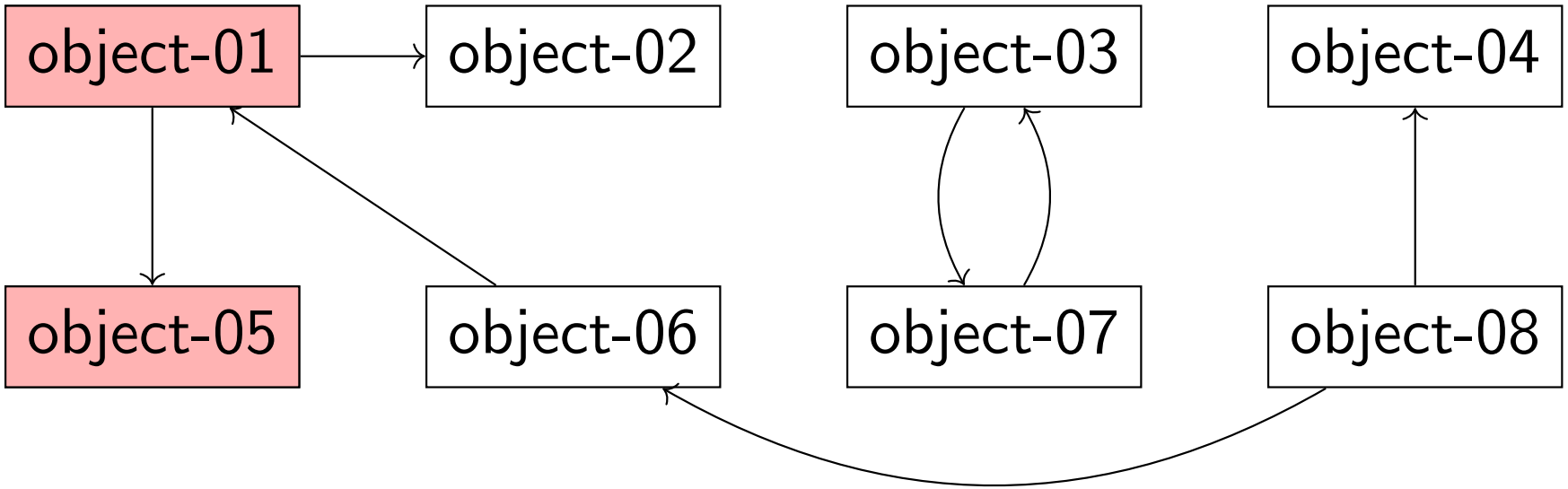


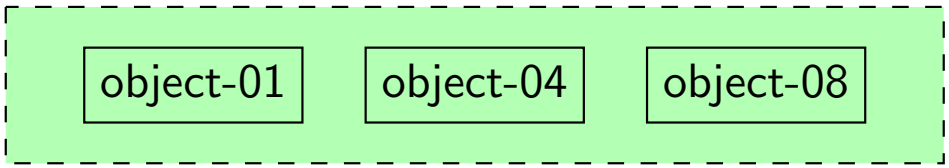
Root Set



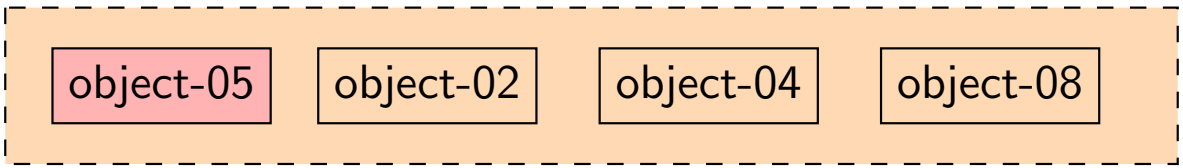
Working Set

Example



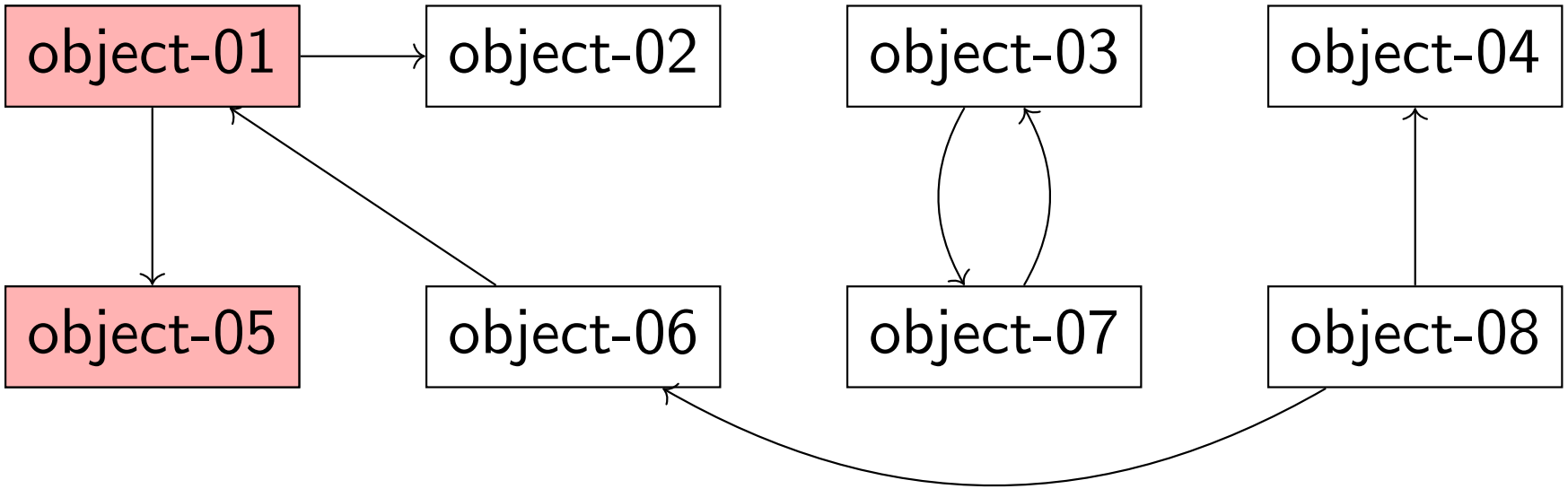


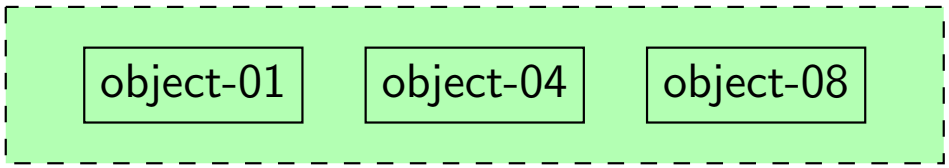
Root Set



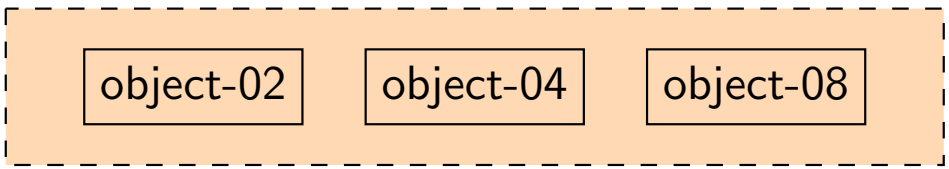
Working Set

Example



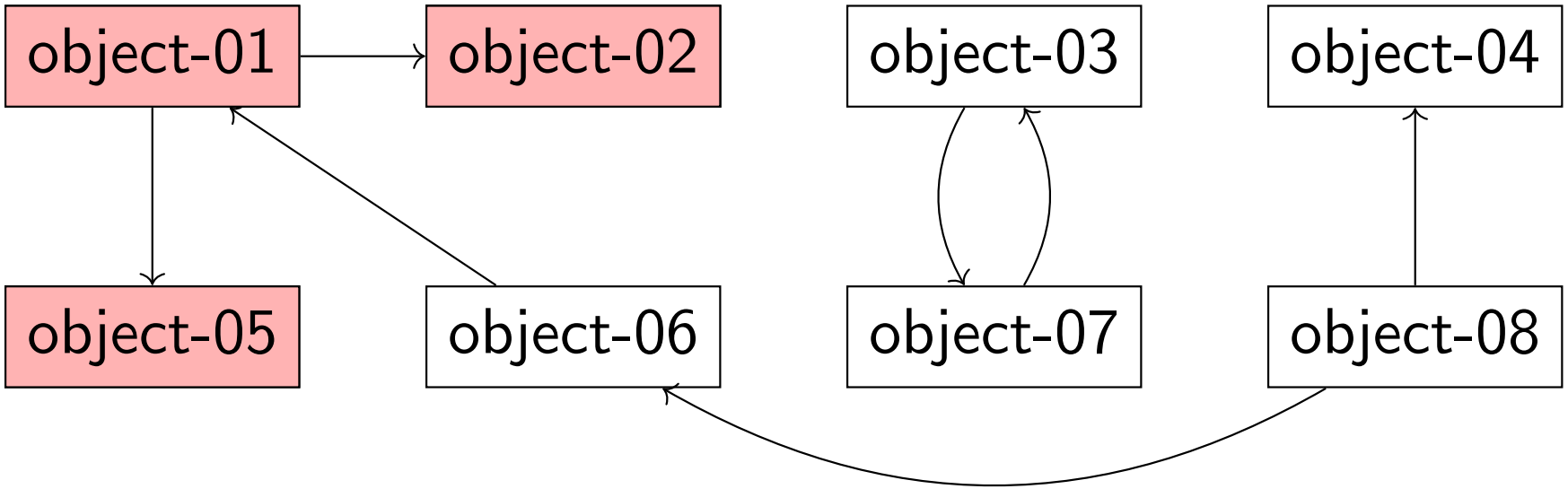


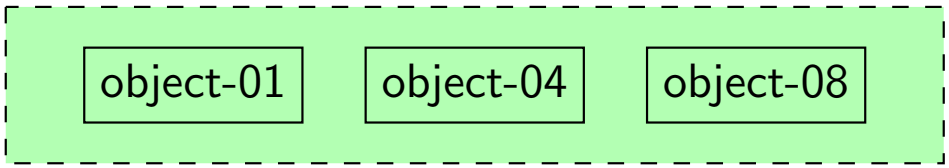
Root Set



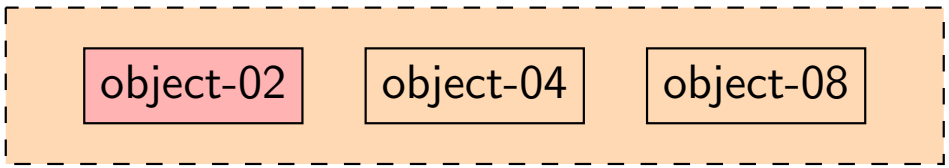
Working Set

Example



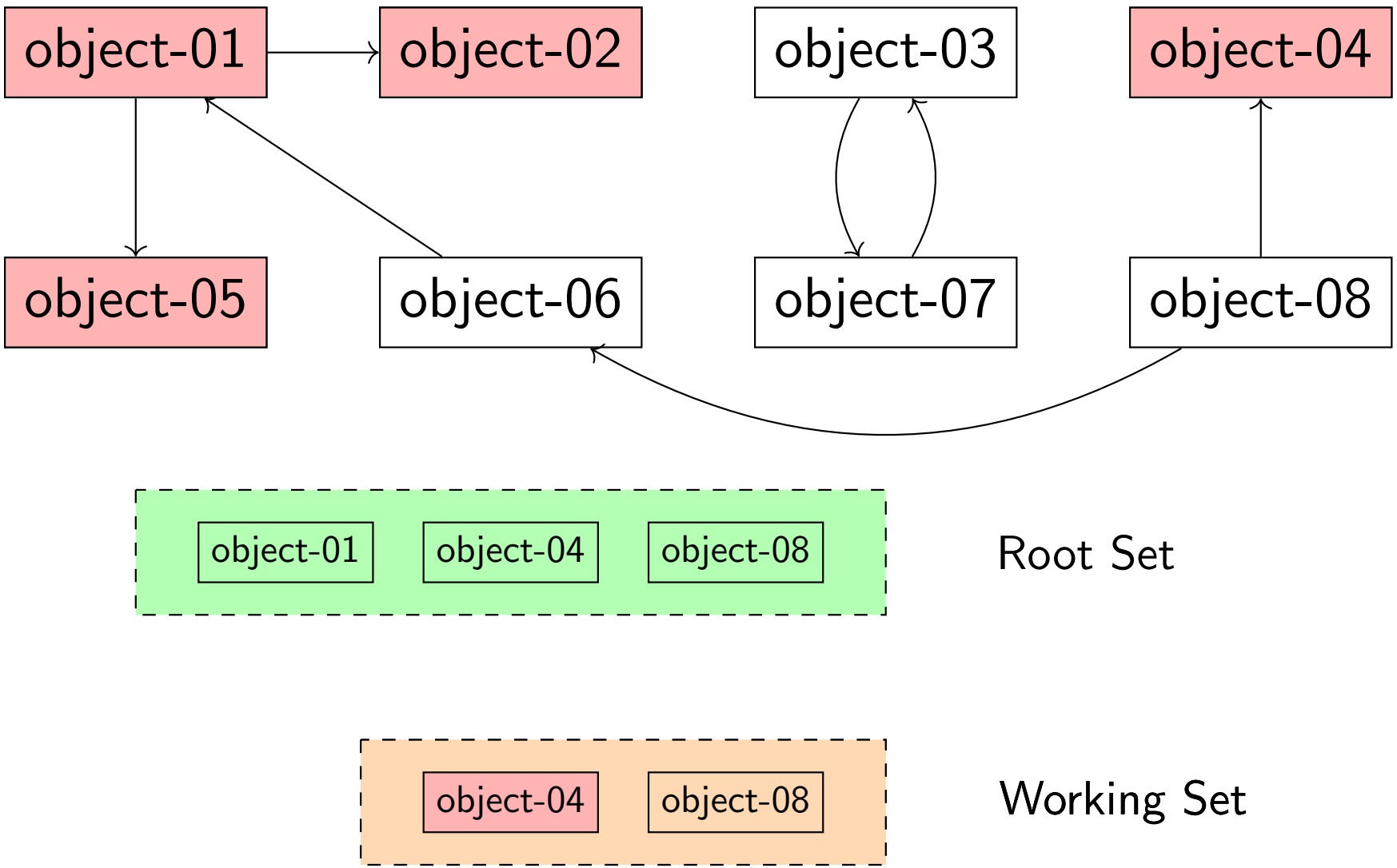


Root Set

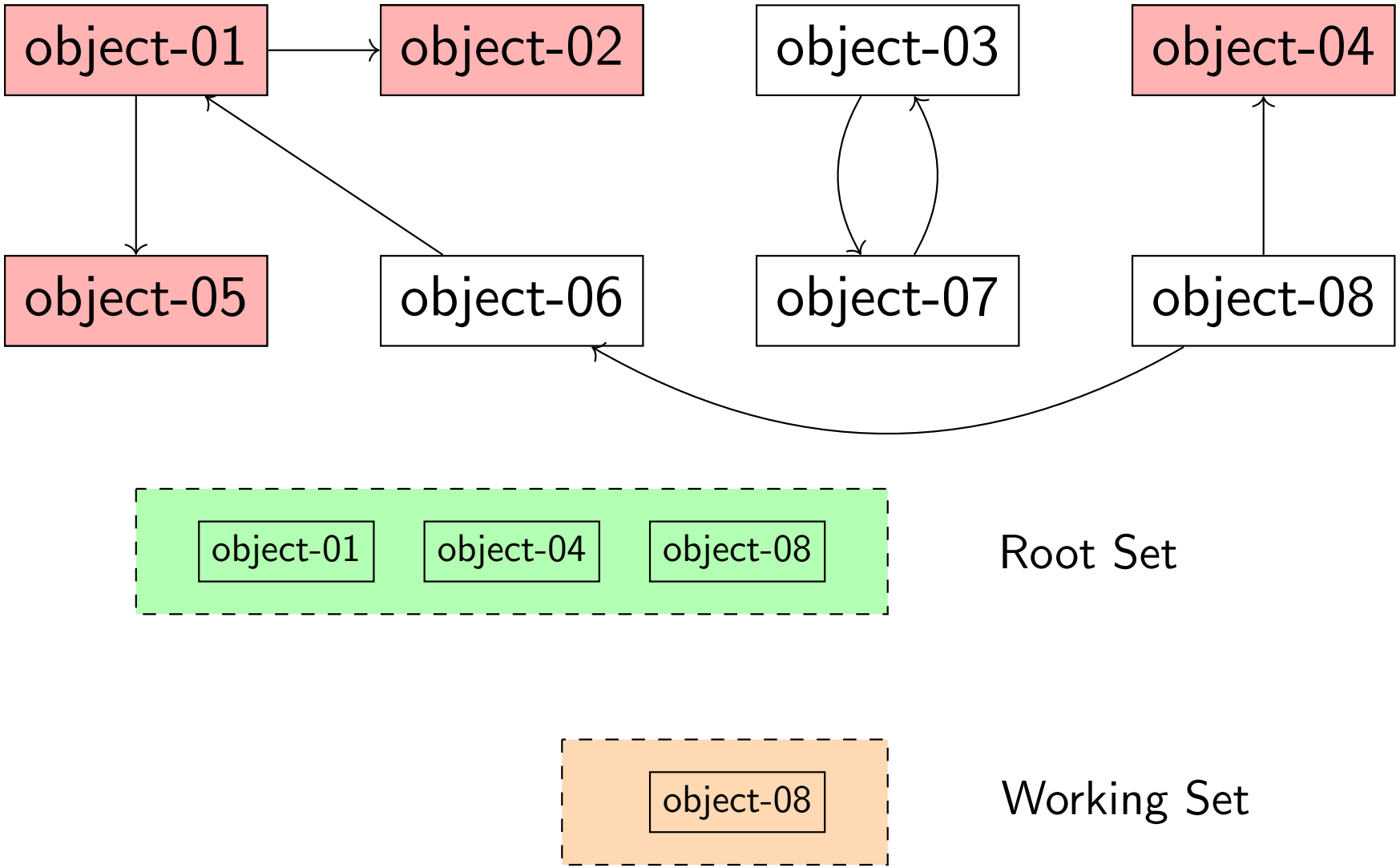


Working Set

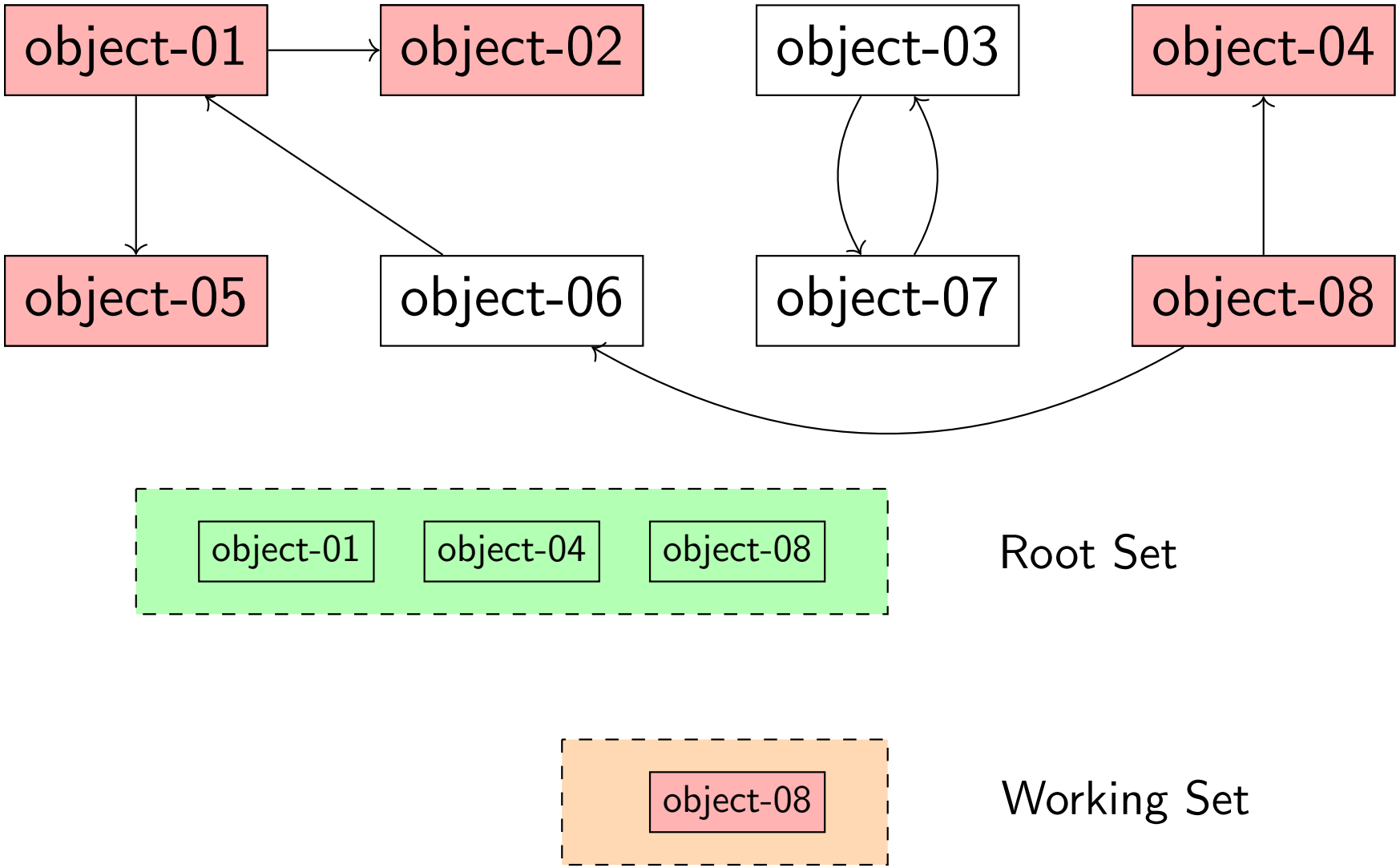
Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
object-04
object-08
Working Set

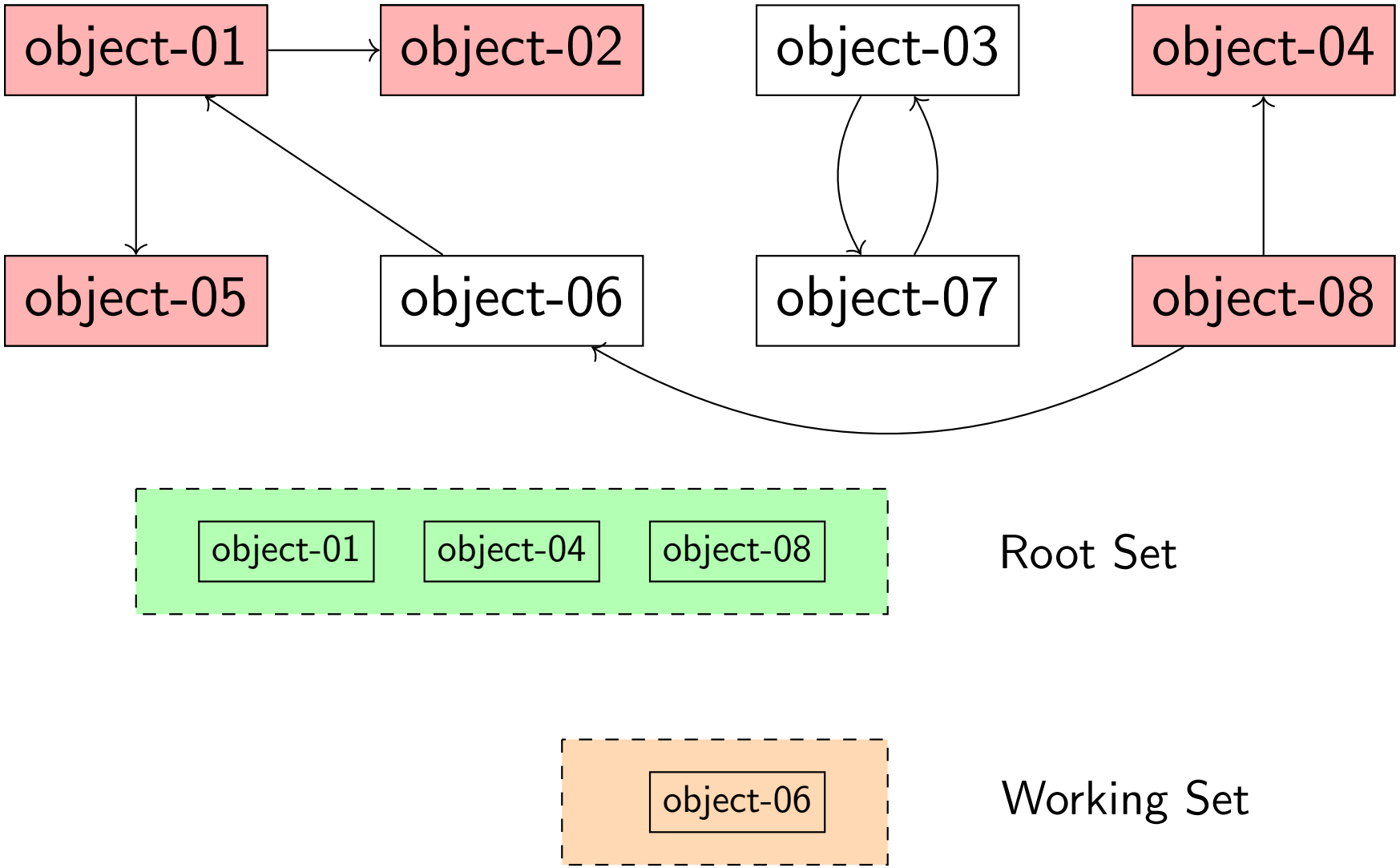
Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
object-08
Working Set

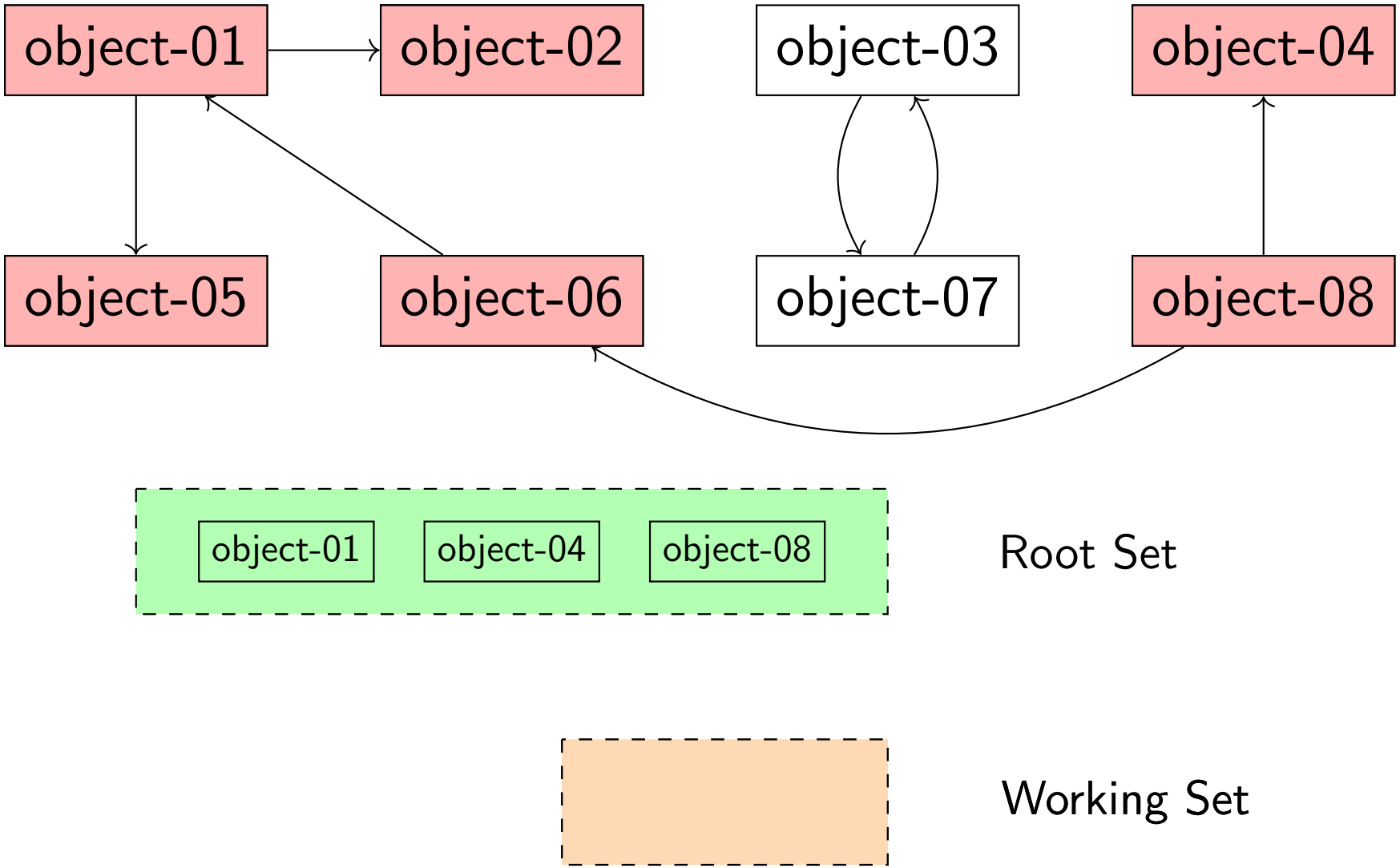
Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
object-08
Working Set

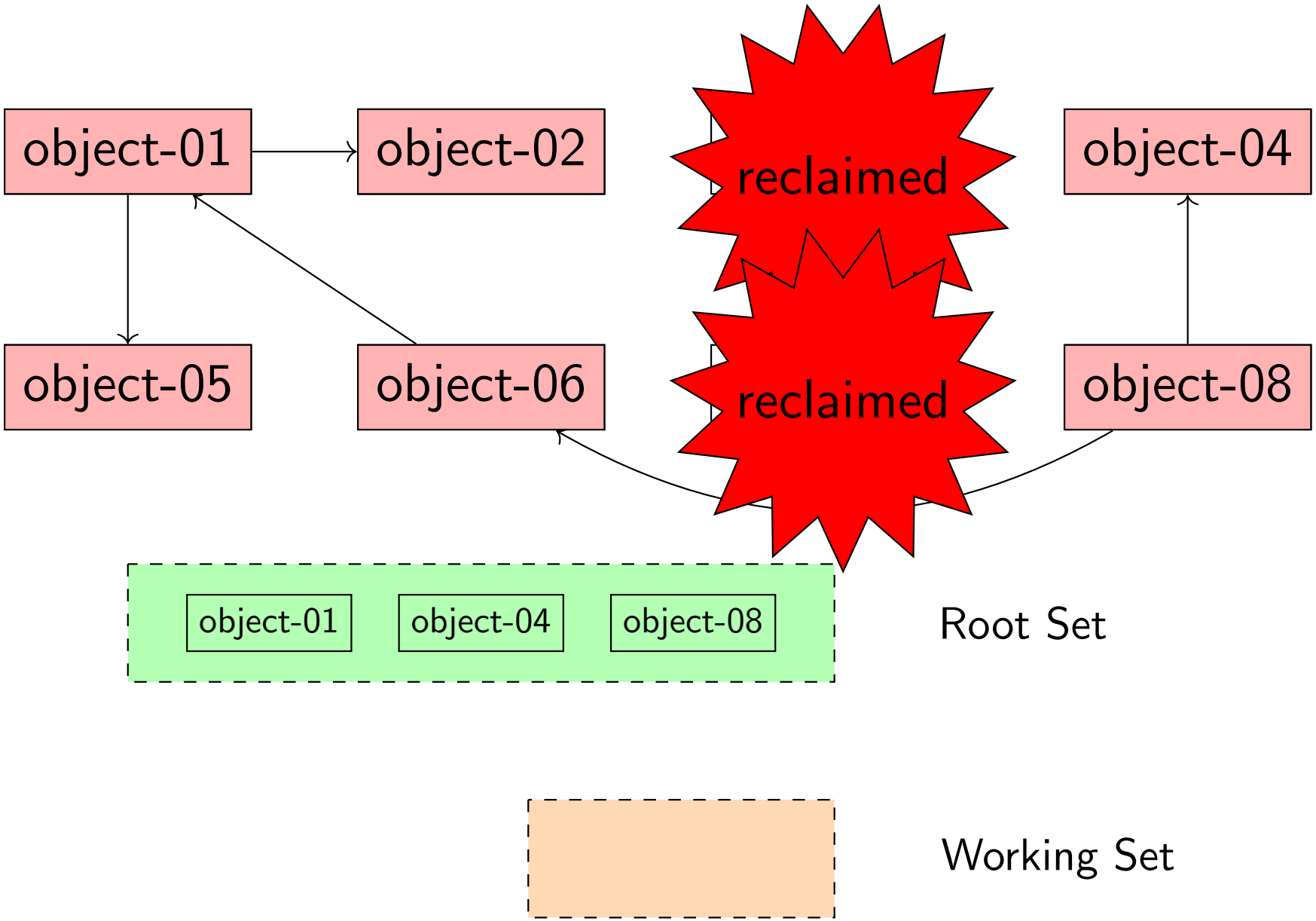
Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
object-06
Working Set

Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
Working Set

Example


object-01
object-02
reclaimed
object-04
object-05
object-06
reclaimed
object-08
object-01
object-04
object-08
Root Set
Working Set

How to sweep?

Sweeping requires to know where are unreacheable objets !

Heap :

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08

How to sweep?

Sweeping requires to know where are unreacheable objets !

Heap :

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08

Just remove from the heap all non-marked objects

Problems

- Runtime proportional to number of allocated objects
  - Sweep phase visits all objects to free them or clear marks
- Work list requires lots of memory
  - Amount of space required could potentially be as large as all of memory
  - Can't preallocate this space

Pros and Cons

Pros:

- Can free cyclic references
- 1 bits per state
- Runtime can be proportional to the number of reachable objects (Baker's algorihtm)

Cons:

- Stop the world algorithm with possibly huge pauses times
- Memory Fragmentation
- Need to walk the whole heap

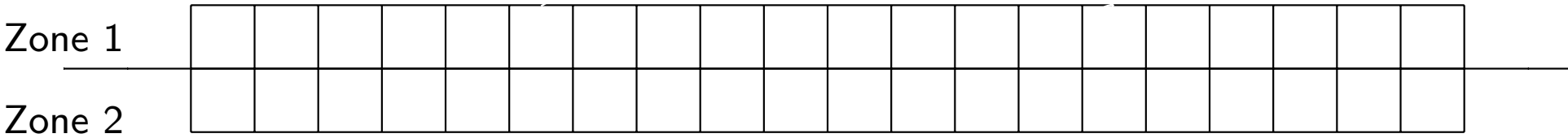
Table of contents

Analysis

- Locality can be improved
  - After garbage collection, objects are no longer closed in memory
- Allocation speed can be improved
  - After garbage collection, the free list of the allocator must be walked.

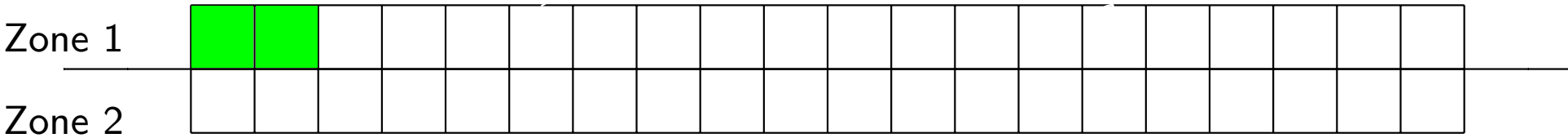
The Sweep Phase can be improved

Exemple



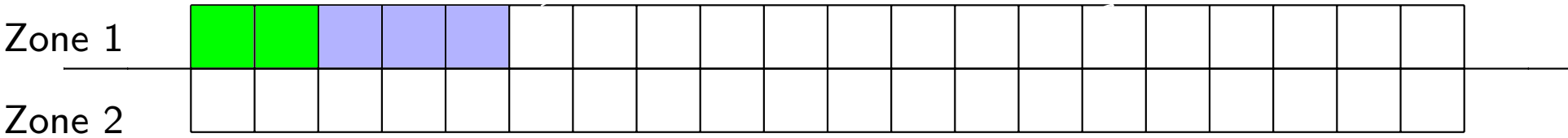
- Split memory in two pieces

Exemple



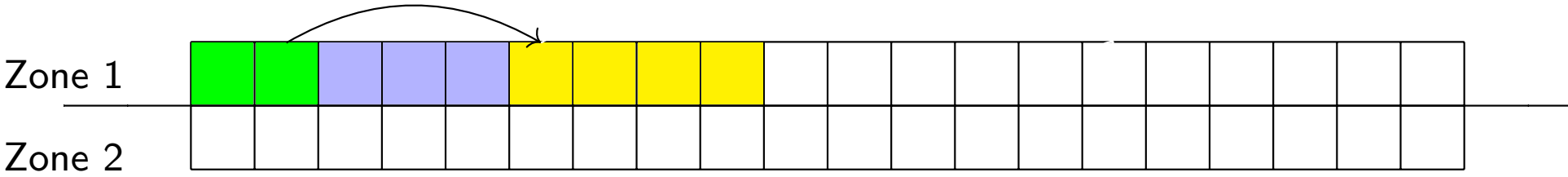
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



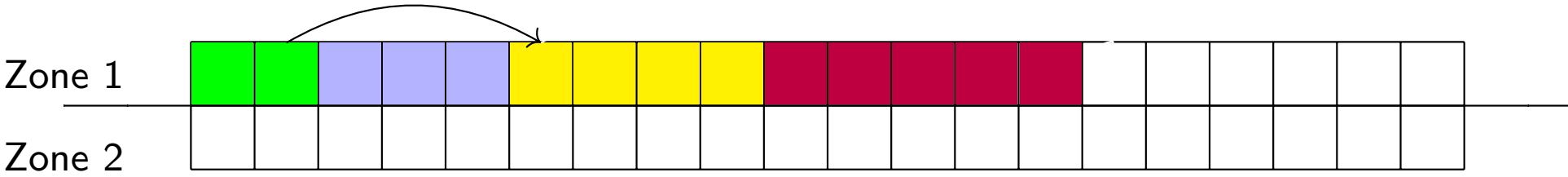
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



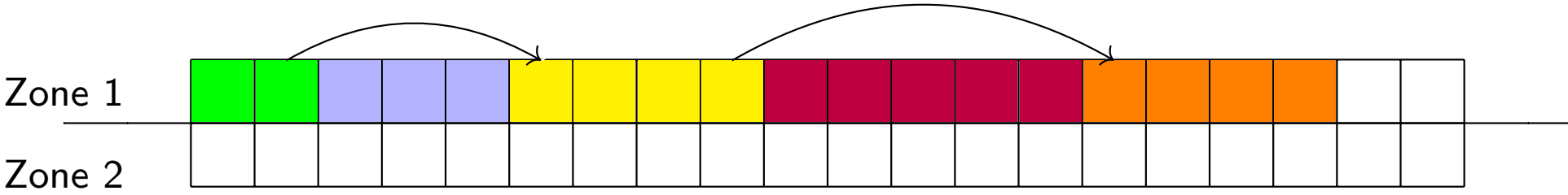
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



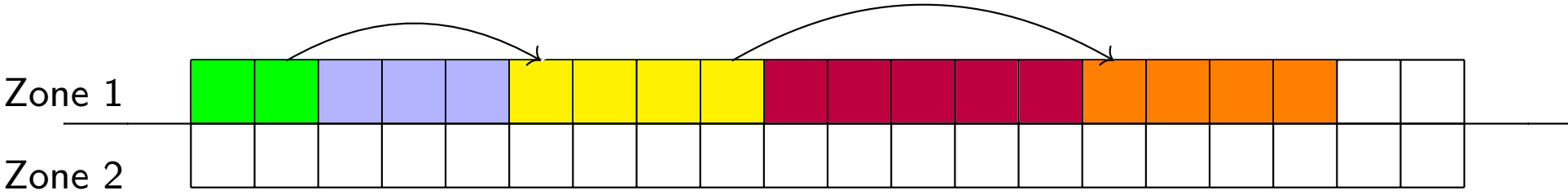
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



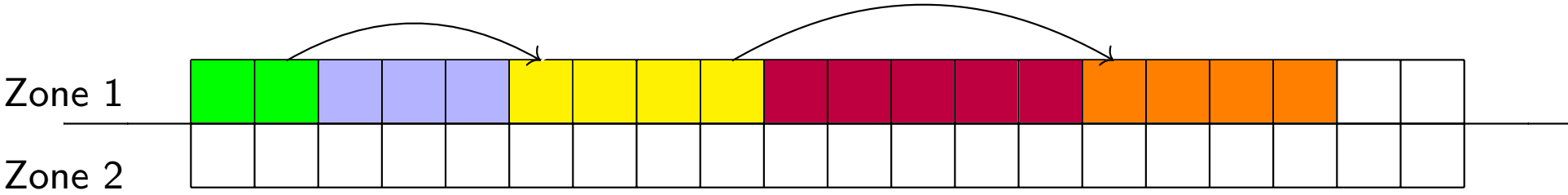
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



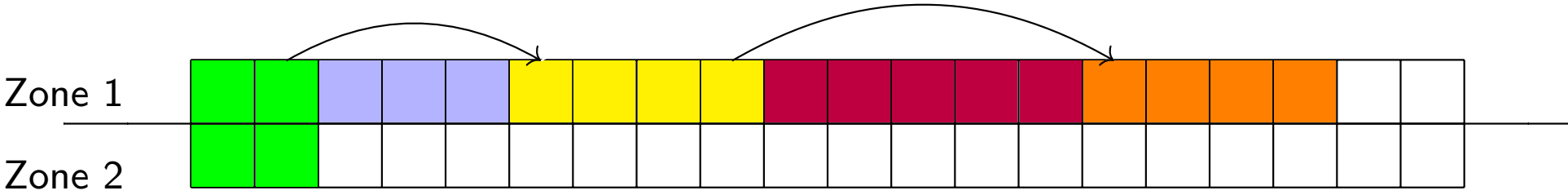
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!

Exemple



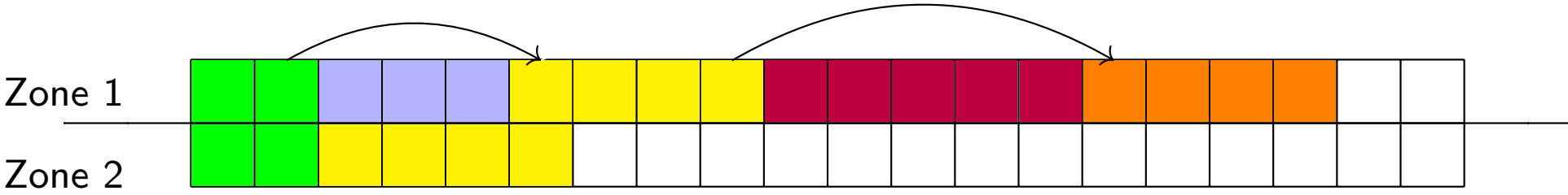
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)

Exemple



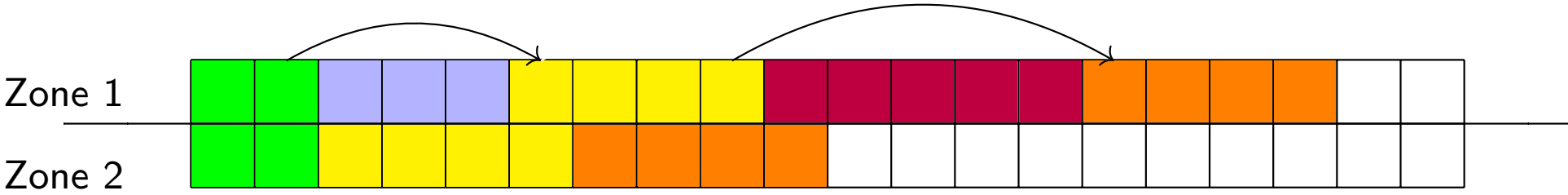
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects

Exemple



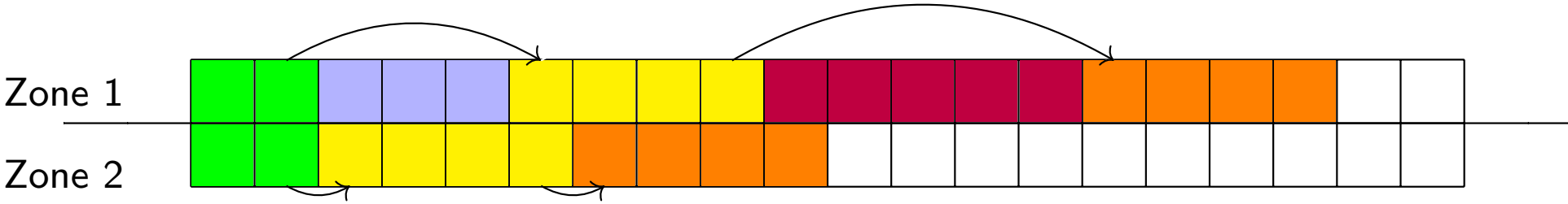
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects

Exemple



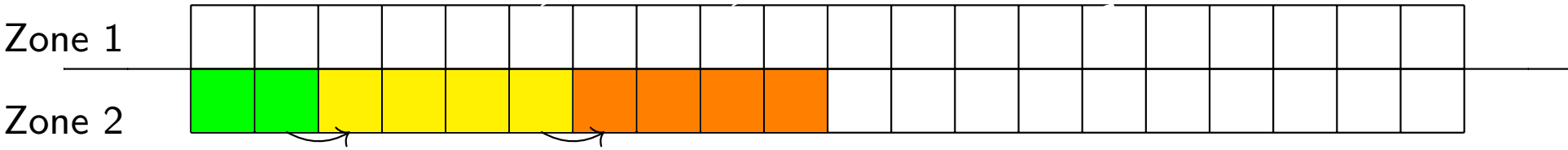
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects

Exemple



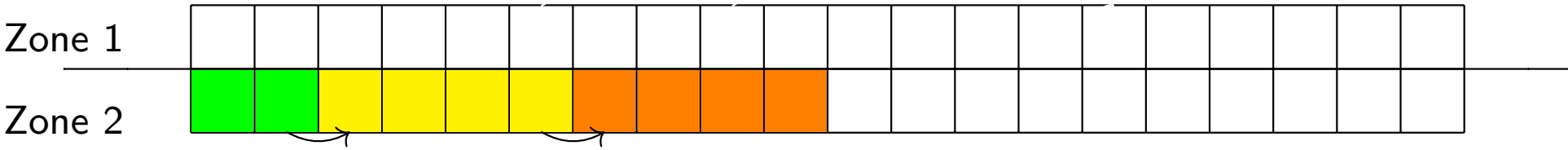
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set

Exemple



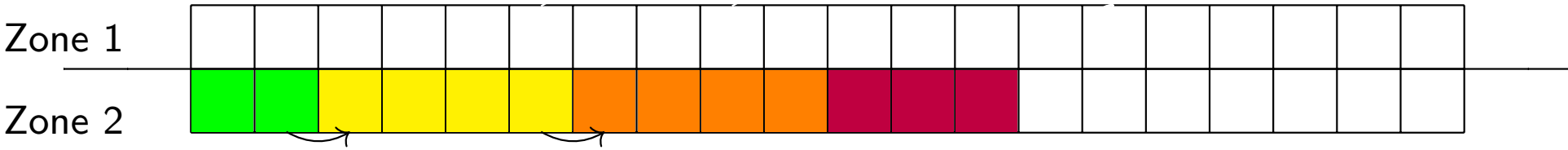
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set
- Clean zone 1 (Constant time)

Exemple



- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set
- Clean zone 1 (Constant time)
- Swap zone 1 and 2 (Now allocation will happen in zone 2)

Exemple



- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set
- Clean zone 1 (Constant time)
- Swap zone 1 and 2 (Now allocation will happen in zone 2)
- Allocate the object that have provoqued the GC

Implementation

- Partition memory into two regions: the old space and the new space.
- Keep track of the next free address in the new space.
- To allocate n bytes of memory:
- If n bytes space exist at the free space pointer, use those bytes and advance the pointer.
- Otherwise, do a copy step. To execute a copy step:
- For each object in the root set:
  - Copy that object over to the start of the old space.
  - Recursively copy over all objects reachable from that object.
- Adjust the pointers in the old space and root set to point to new locations.
- Exchange the roles of the old and new spaces.

Problems

How to adjust pointers in the copied objects correctly?

Problems

How to adjust pointers in the copied objects correctly?

1. Have each object contain a extra space for a forwarding pointer
2. First, do a complete bitwise copy of the object
3. Next, set the forwarding pointer of the original object to point to the new object
   - Follow the pointer to the object it references
   - Replace the pointer with the pointee's forwarding pointer

Pros and Cons

Pros:

- Compact the Heap
- Allocation only increments a pointer
- No sweep

Cons:

- Smaller Heap
- Copy
- Reference adjusting

Table of contents

Analysis

The best garbage collectors in use today are based on a combination of smaller garbage collectors

Objects Die Young

Most objects have extremely short lifetimes

Optimize garbage collection to reclaim young objects rapidly while spending less time on older objects

Generational Garbage Collector

- Partition memory into several generations
- Objects are always allocated in the first generation.
- When the first generation fills up, garbage collect it.
  - Runs quickly; collects only a small region of memory.
- Move objects that survive in the first generation long enough into the next generation.
- When no space can be found, run a full (slower) garbage collection on all of memory.

Garbage Collection in Java

1. Split the Heap in 3 zones: eden, survivors and tenured
2. New objects are allocated using a modified stop-and-copy collector in the Eden space.
3. When Eden runs out of space, the stop-and-copy collector moves its elements to the survivor space.
4. Objects that survive long enough in the survivor space become tenured and are moved to the tenured space.
5. When memory fills up, a full garbage collection (perhaps mark-and-sweep) is used to garbage-collect the tenured objects

Garbage Collection in C

- Boehm GC
- Mark and Sweep
- Conservative
- Consider all program variables as root set
- Easy to combine with C

Bibliography

- Uniprocessor Garbage Collection , Paul R. Wilson