Compiler Construction

Lecture 12: garbage collection

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk

Lent 2024

Memory management

Manual memory management: programmer controls (de)allocation time/place:

```
void *malloc(size_t n)    /* allocate n bytes, return address */
void free(void *addr)     /* relinquish use of memory at addr */
```

The programmer has a lot of control. However, mistakes can be disastrous:

missing free

```
p = malloc(10);
return OK;
```

double free

```
free(p);
free(p);
```

use after free

```
free(p);
*p += 1;
```

(Observation: deallocation is much harder than allocation)

Many programming languages support heap allocation
but do not provide a deallocation operation

```
d = dict(x=3,y=4)
```

Python

```
let d = [("x",3); ("y",4)]
```

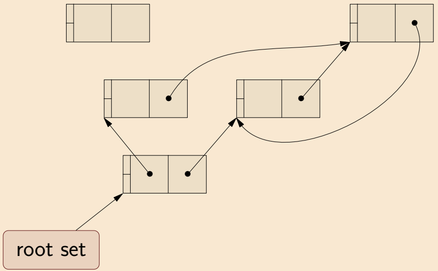
OCaml

Unless the storage is reclaimed *somehow*, memory might be exhausted.

General approach: **automatic memory management** (“garbage collection”)

Automation is based on an approximation:

If data can be reached from a **root set**, *then* it is not "garbage"



The root set might include: the stack, registers, global variables.

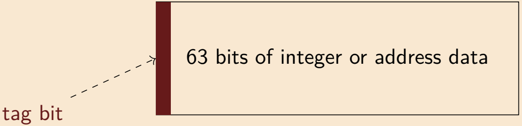
(Without loss of generality, assume a *single root*)

Reachability and representations

Ascertaining reachability requires knowledge of representations:

What is a pointer?

(typical approach: use a *tag bit* to distinguish between pointers and integers)



How are objects laid out?

(typical approach: use *headers* that carry sizes and other metadata)

Reference counting

Reference counting & tracing collection

Two basic approaches (and many variations):

Reference counting

Keep a **reference count** with each object that represents the number of pointers to it.

An object is **garbage when its count is 0**

Tracing garbage collection

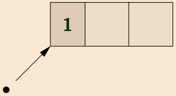
Keep alive objects **reachable** from the root set (i.e. transitive close of pointer graph)

An object is **garbage when it is unreachable**

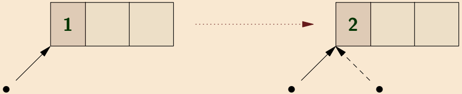
Reference counting: idea

The **reference count** tracks the number of pointers to each object.

An object's reference count is 1 when the object is created:



The count is incremented when a pointer newly references the object:

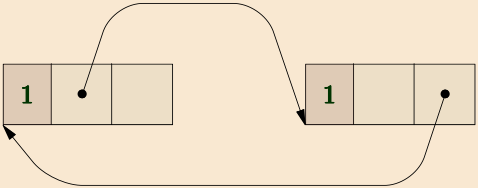


The count is decremented when a pointer no longer references the object:

The object is unreachable garbage when the reference count goes to 0:

Reference counting can't collect cyclic garbage

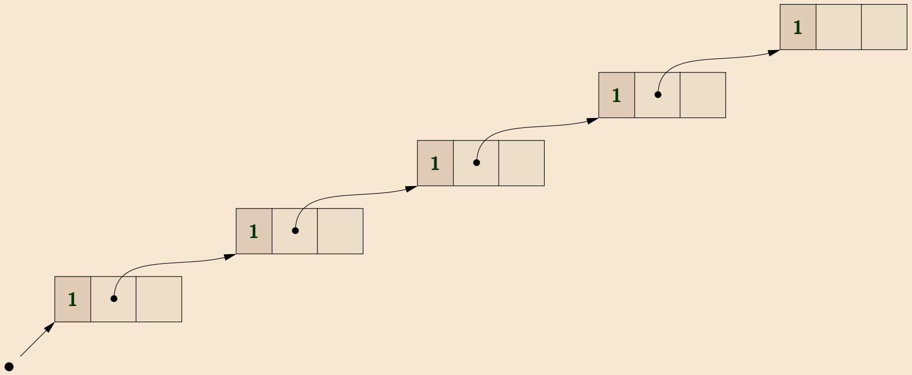
A **significant weakness** of reference counting:



There are no other references to these objects in the program but **the objects will never be collected**.

Long chains of objects make reclamation expensive

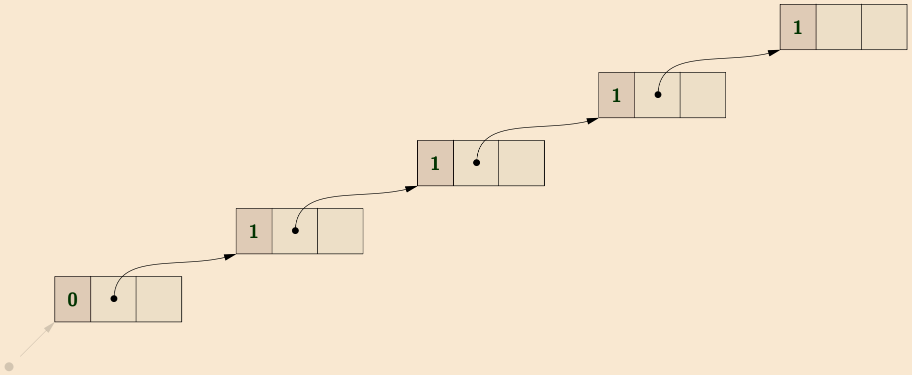
A **significant weakness** of (naive) reference counting:



Reclaiming an object can set off an unboundedly large **chain of reclamations**

Long chains of objects make reclamation expensive

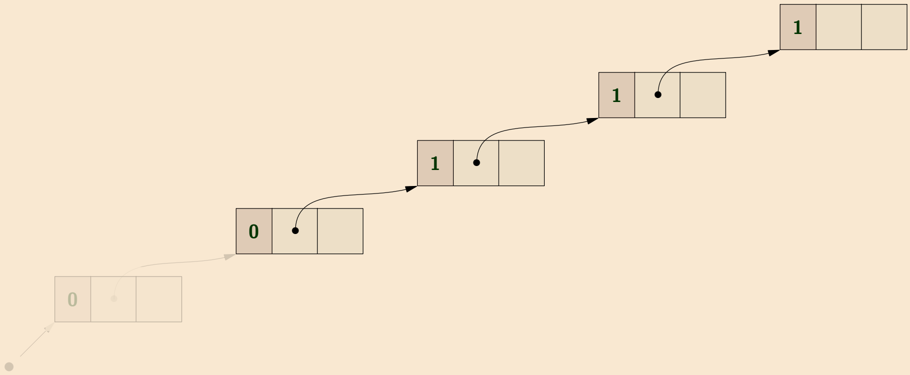
A **significant weakness** of (naive) reference counting:



Reclaiming an object can set off an unboundedly large **chain of reclamations**

Long chains of objects make reclamation expensive

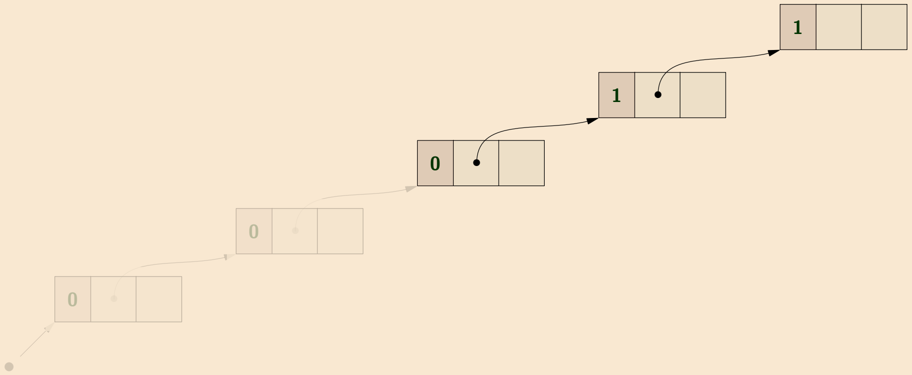
A **significant weakness** of (naive) reference counting:



Reclaiming an object can set off an unboundedly large **chain of reclamations**

Long chains of objects make reclamation expensive

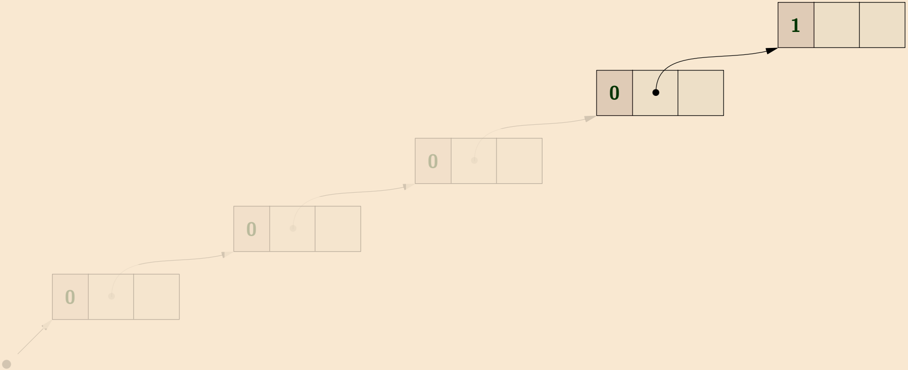
A **significant weakness** of (naive) reference counting:



Reclaiming an object can set off an unboundedly large **chain of reclamations**

Long chains of objects make reclamation expensive

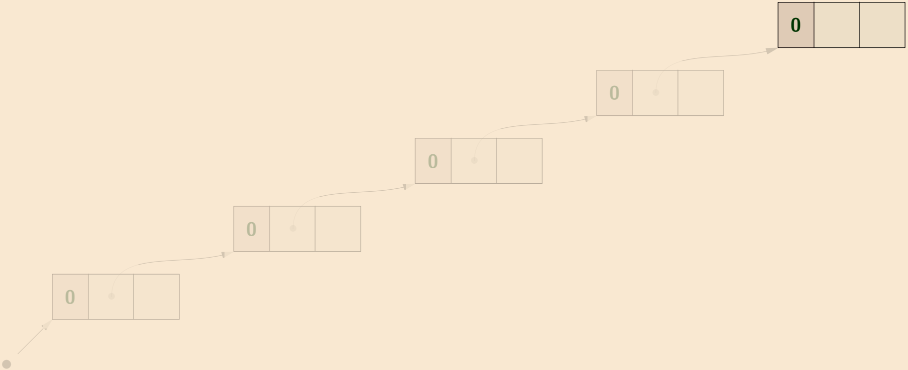
A **significant weakness** of (naive) reference counting:



Reclaiming an object can set off an unboundedly large **chain of reclamations**

Long chains of objects make reclamation expensive

A **significant weakness** of (naive) reference counting:



Reclaiming an object can set off an unboundedly large **chain of reclamations**

Reference counting: advantages and drawbacks

Advantages of reference counting:

+ Collection costs distributed through the computation
+ Allows rapid reclamation and immediate reuse

Drawbacks of reference counting:

– size overhead of storing references
– potentially high/unbounded cost on reclamation
– taking a reference involves (potentially expensive) mutation

Mark & sweep

Mark & sweep is a two-phase algorithm:

Mark phase: Traverse object graph depth first to **mark live data**

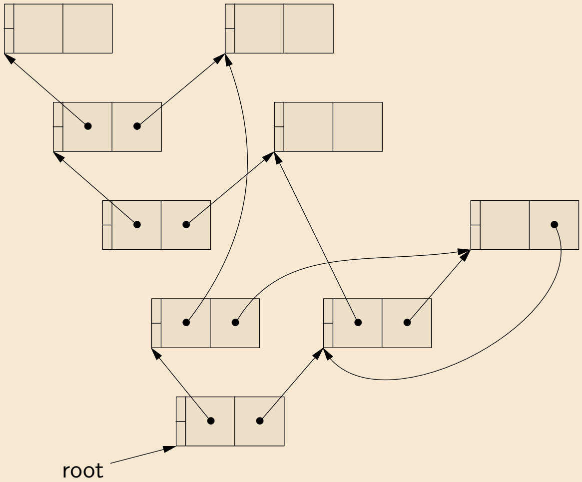
Sweep phase: iterate over entire heap, reclaiming unmarked data

Key idea: identify and reclaim **dead objects**

Marking

Mark

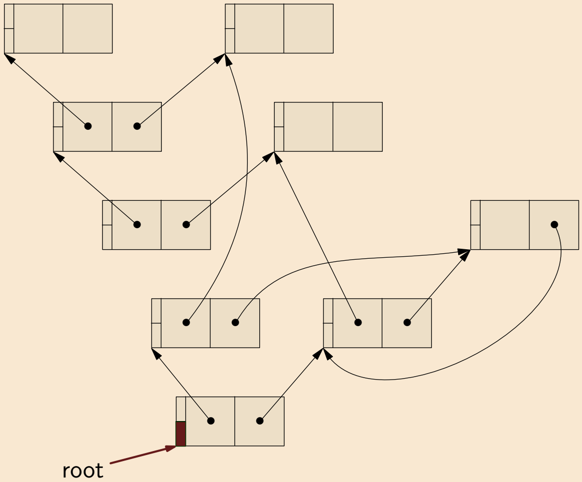
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Memory
Reference counting
Mark & sweep
Copying
Generations

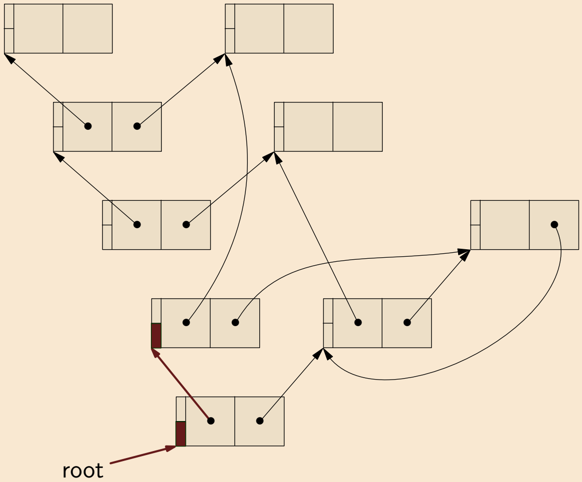
Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark

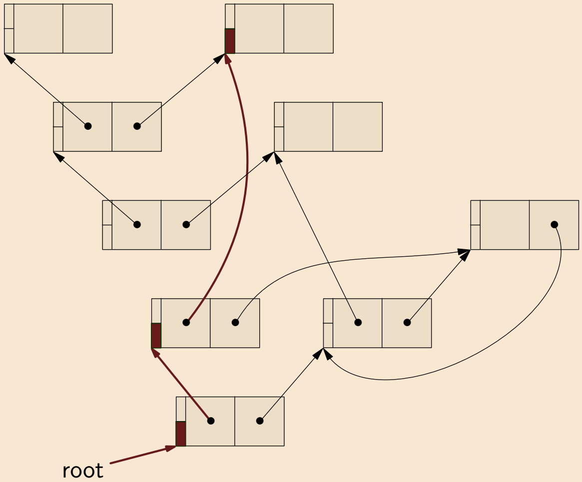
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Marking

Mark

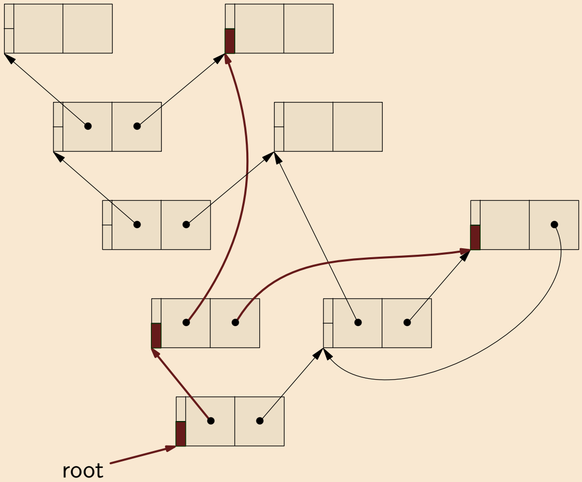
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Marking

Mark

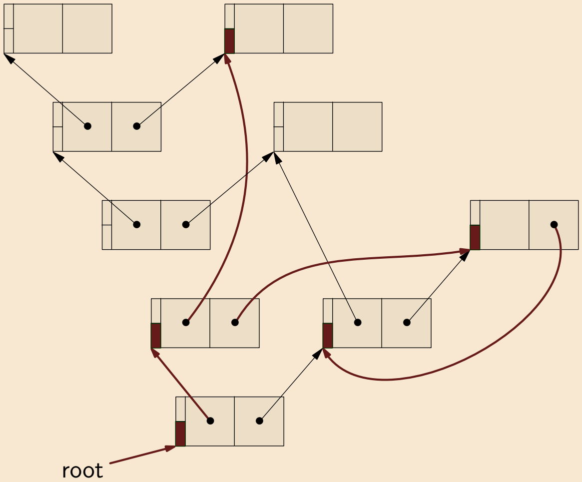
```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Marking

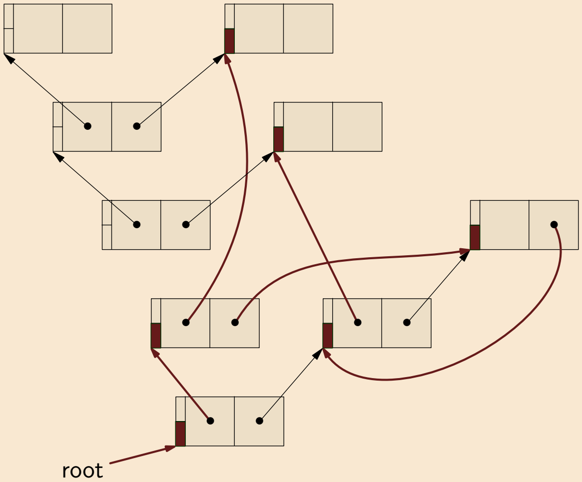
Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



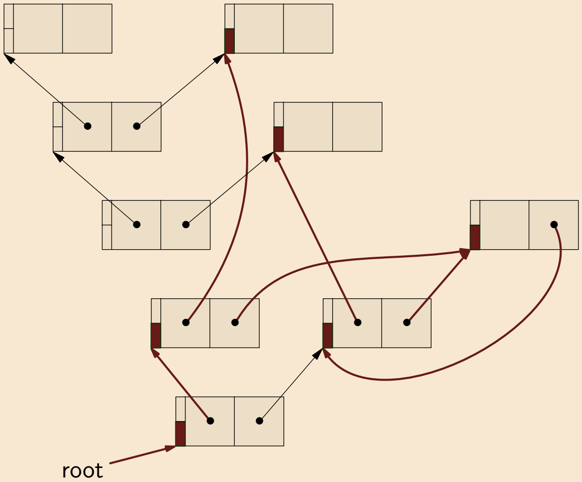
Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



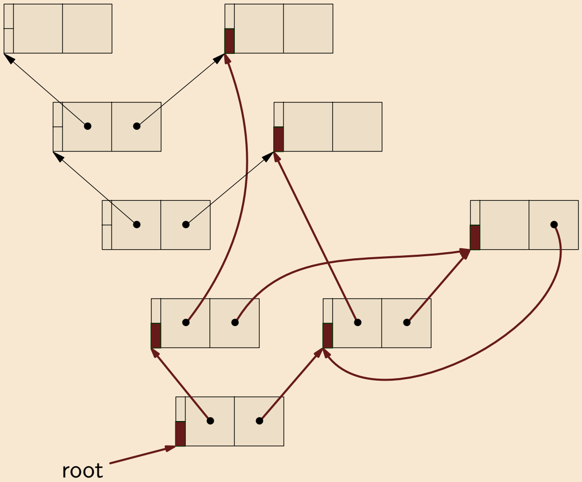
Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Mark

```
mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
```



Memory

Reference counting

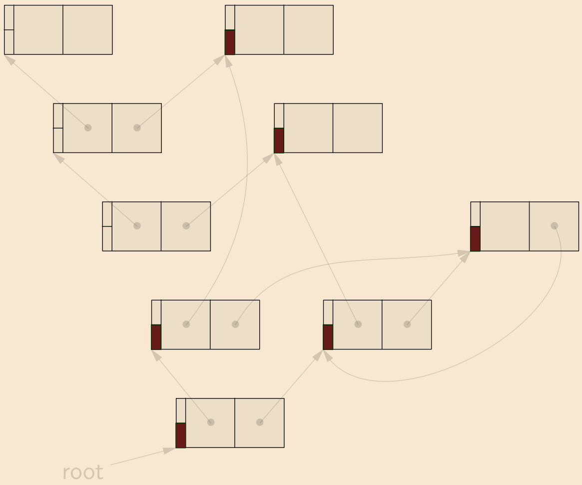
Mark & sweep

Copying

Generations

Sweep

- **linearly scan** through the heap
- **collect unmarked** blocks
- **unmark marked** blocks



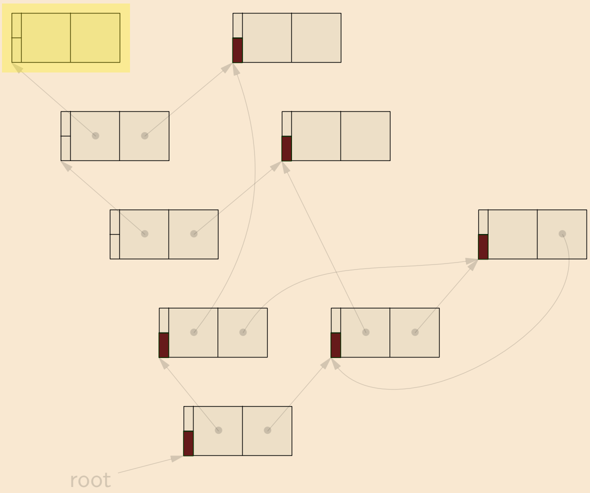
Sweeping

Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks



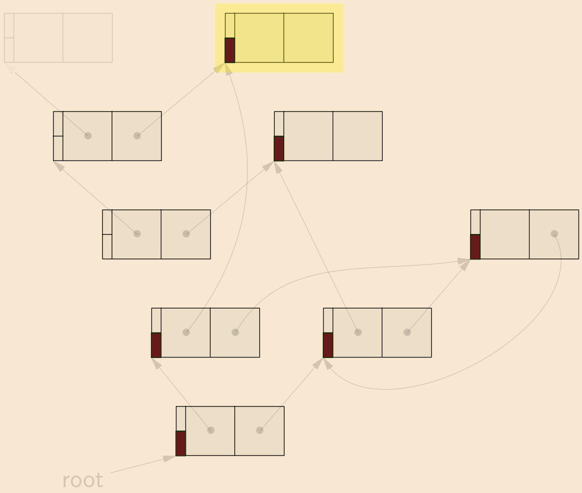
Sweeping

Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks

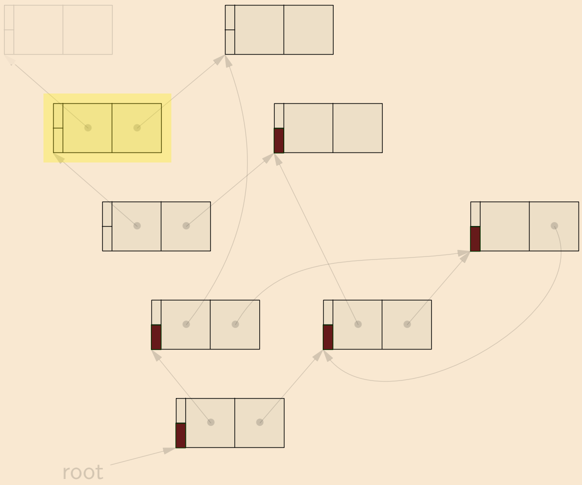


Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks



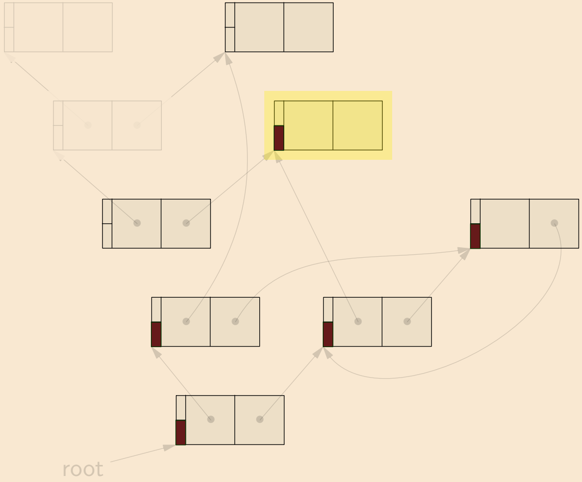
Sweeping

Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks

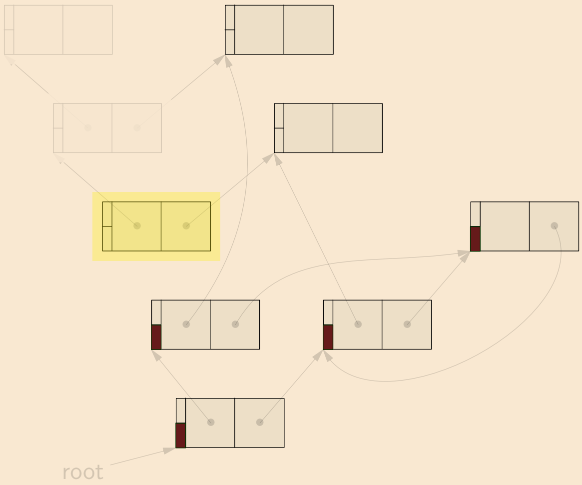


Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks

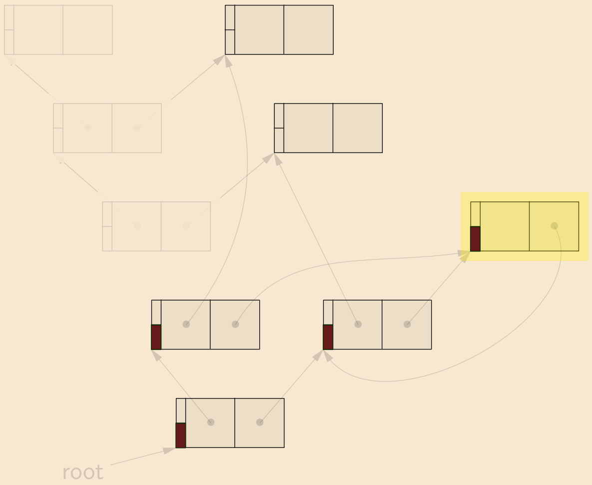


Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks

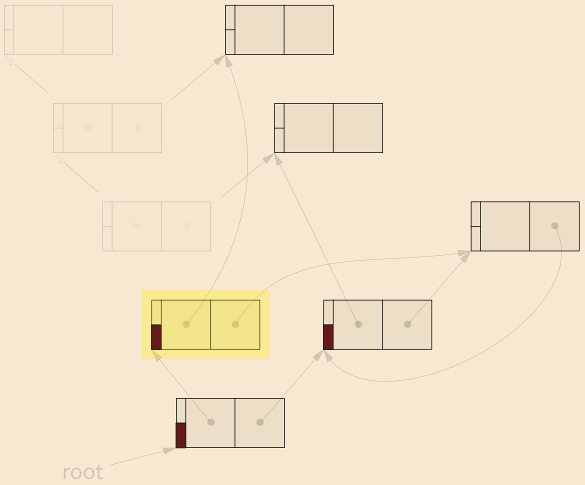


Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks

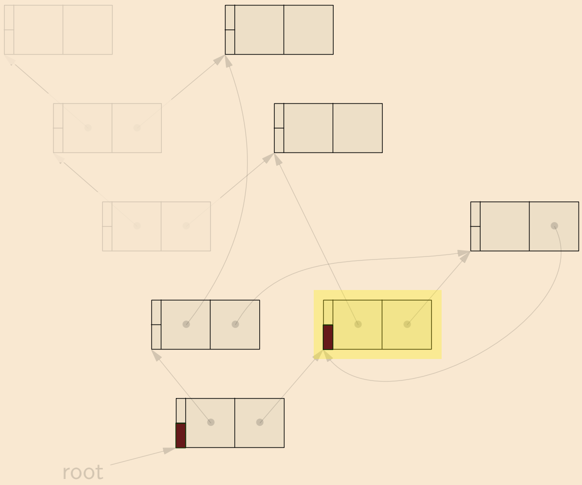


Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks

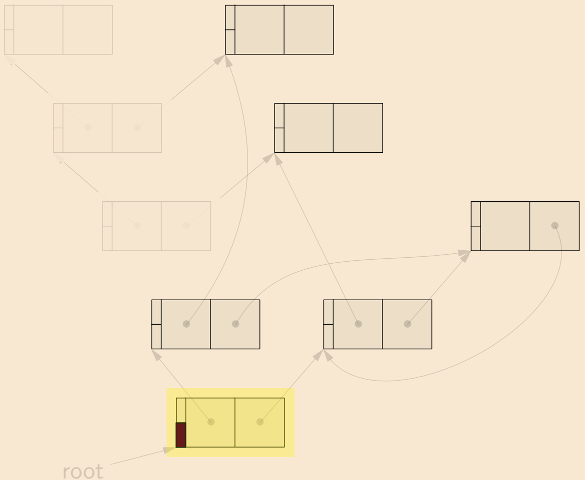


Sweep

linearly scan through the heap

collect unmarked blocks

unmark marked blocks



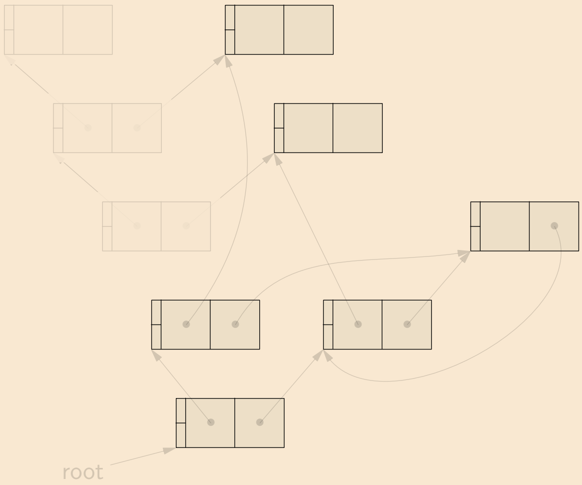
Sweeping

Sweep

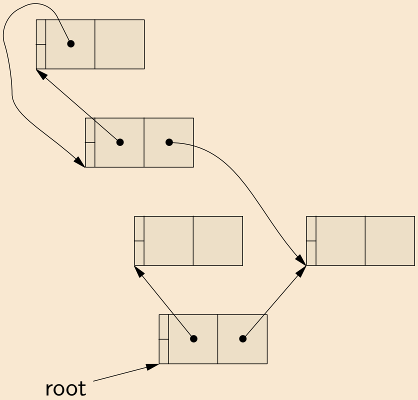
linearly scan through the heap

collect unmarked blocks

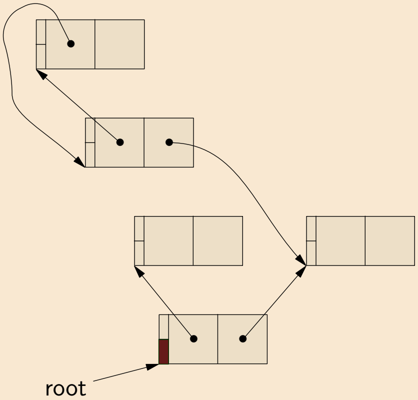
unmark marked blocks



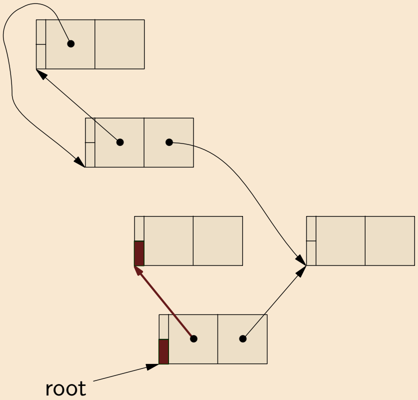
Mark & sweep is able to collect cyclic garbage:



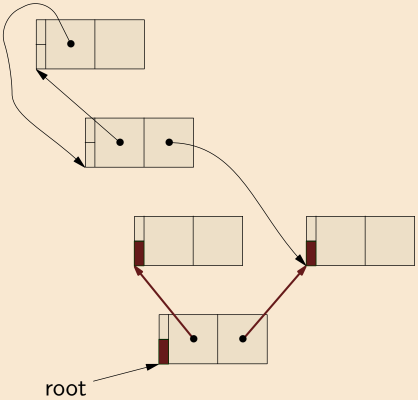
Mark & sweep is able to collect cyclic garbage:



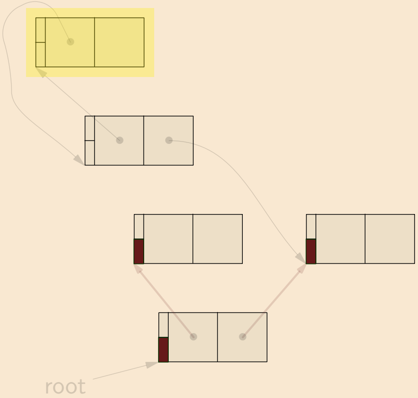
Mark & sweep is able to collect cyclic garbage:



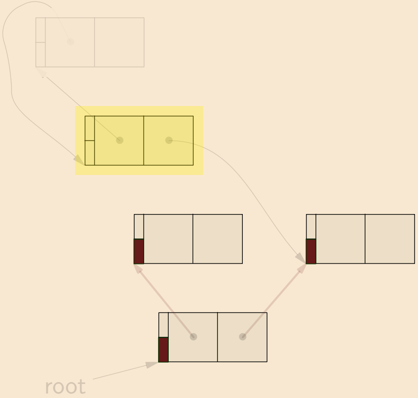
Mark & sweep is able to collect cyclic garbage:



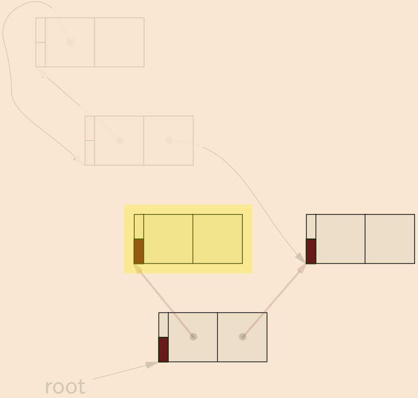
Mark & sweep is able to collect cyclic garbage:



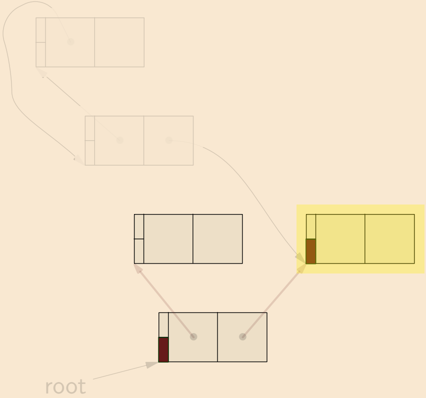
Mark & sweep is able to collect cyclic garbage:



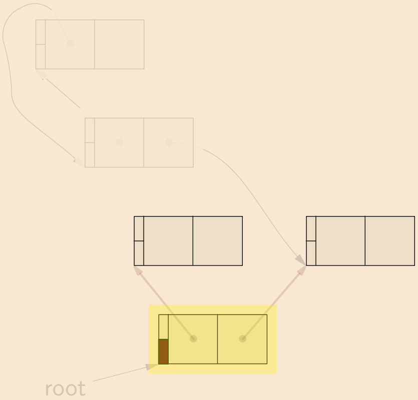
Mark & sweep is able to collect cyclic garbage:



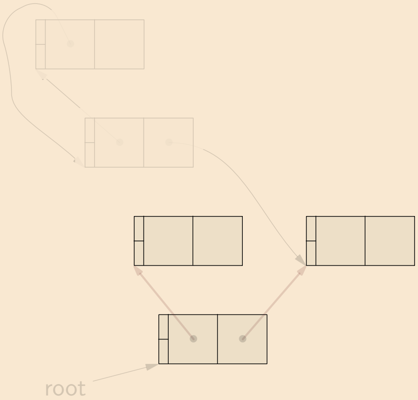
Mark & sweep is able to collect cyclic garbage:



Mark & sweep is able to collect cyclic garbage:



Mark & sweep is able to collect cyclic garbage:



Mark & sweep: advantages and drawbacks

Advantages of mark & sweep:

+ Reasonably simple
+ Collects cycles
+ Low space overhead

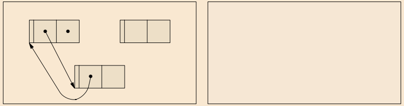
Drawbacks of mark & sweep

- Scans entire heap during sweeping
- Long (multi-second) pauses, inappropriate for interactive applications

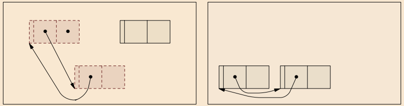
Copying collection

Copying collection: overview

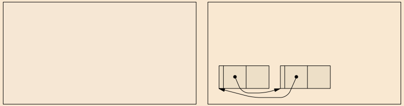
Split heap in two: **from-space** (active), **to-space** (unused)



During garbage collection: **copy** from from-space into to-space



After garbage collection: **abandon** dead objects, switch heap roles



Key idea: identify and move **live objects**

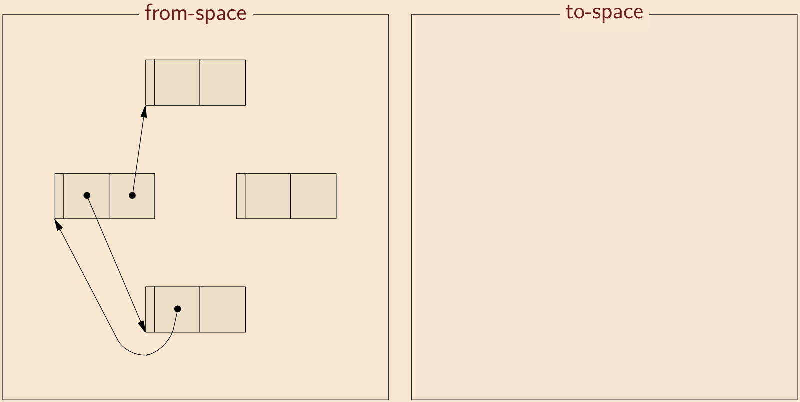
Copying collection: example

Collect

copy live blocks to to-space (starting at the root)

leave **forwarding addresses** in from-space

switch roles of spaces



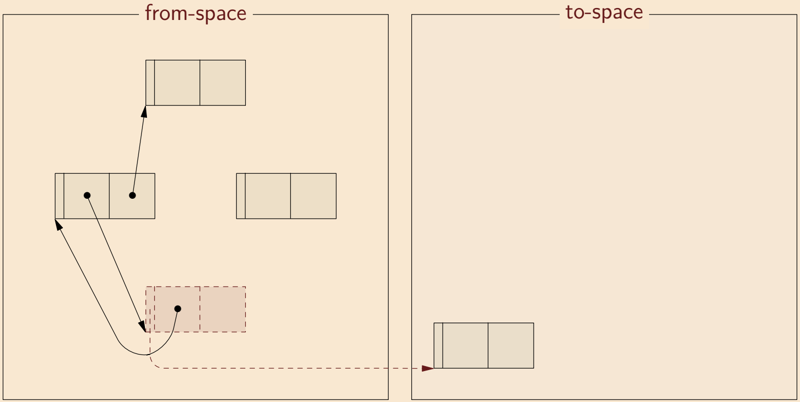
Copying collection: example

Collect

copy live blocks to to-space (starting at the root)

leave **forwarding addresses** in from-space

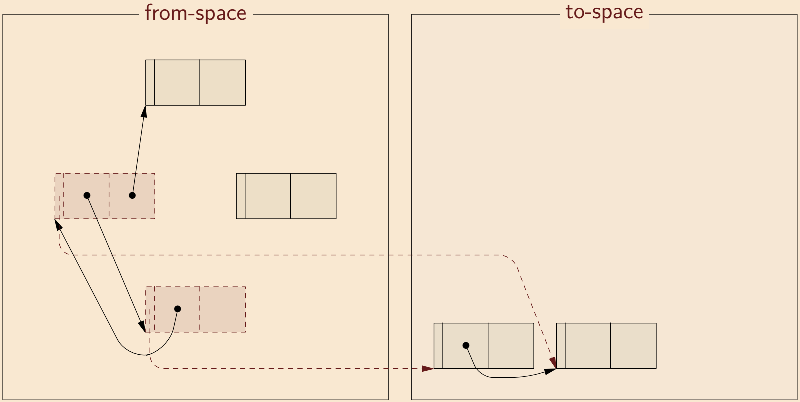
switch roles of spaces



Copying collection: example

Collect

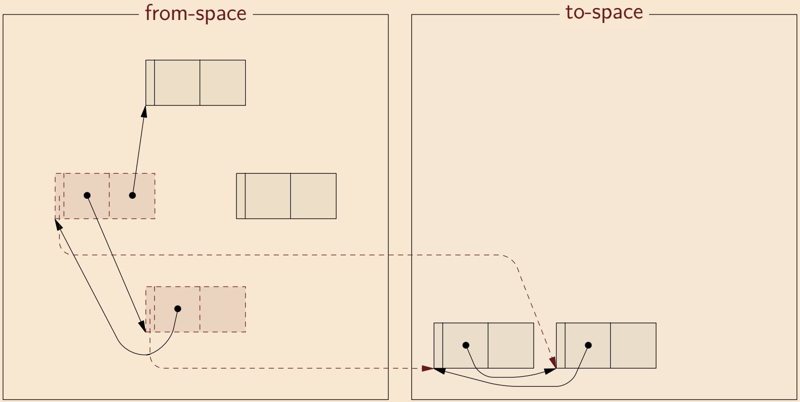
copy live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
switch roles of spaces



Copying collection: example

Collect

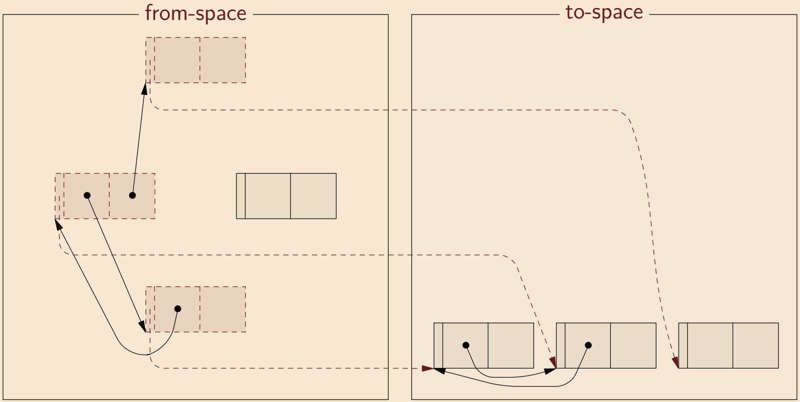
copy live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
switch roles of spaces



Copying collection: example

Collect

copy live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
switch roles of spaces



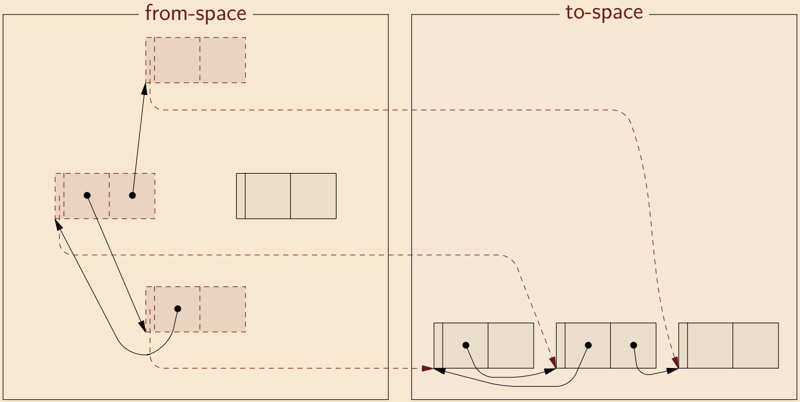
Copying collection: example

Collect

copy live blocks to to-space (starting at the root)

leave **forwarding addresses** in from-space

switch roles of spaces

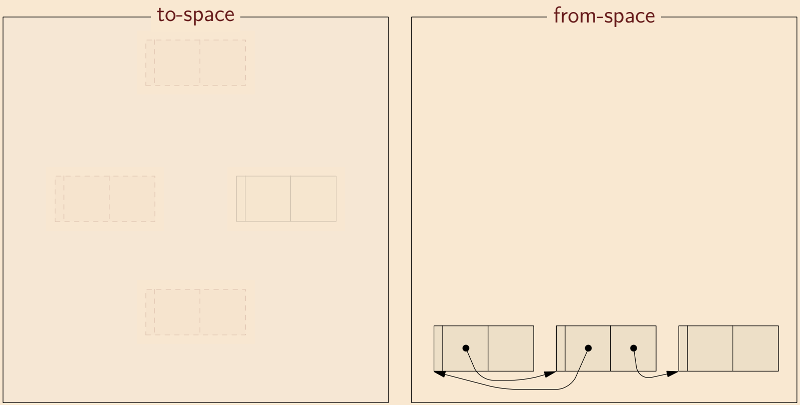


Collect

copy live blocks to to-space (starting at the root)

leave **forwarding addresses** in from-space

switch roles of spaces



Copying collection: advantages and drawbacks

Advantages of copying garbage collection:

- \+ Reasonably simple
- \+ Collects cycles
- \+ Has **running time propotional to the number of live objects**
- \+ **Automatically compacts** memory, eliminating fragmentation
- \+ Very low allocation costs (pointer bump)

Drawbacks of copying garbage collection

- – Uses **twice as much memory** as the program requires

Generational garbage collection

Generational GC: motivation

Observation: scanning all live objects takes a long time

Observation: programs often allocate a lot (hundreds of MB per second)

Observation: object lifetimes are mostly very short or relatively long

Example **evidence** (much more is available):

$> 98\%$ of collected garbage had been allocated and discarded since previous collection

(Foderaro and Fateman, 1981)

$80 - 98\%$ of objects die before 1MB old

(Wilson, 1994)

$50 - 90\%$ of Common Lisp objects die before 10KB old

(Zorn, 1989)

Generational GC: idea

Key idea: focus on young objects

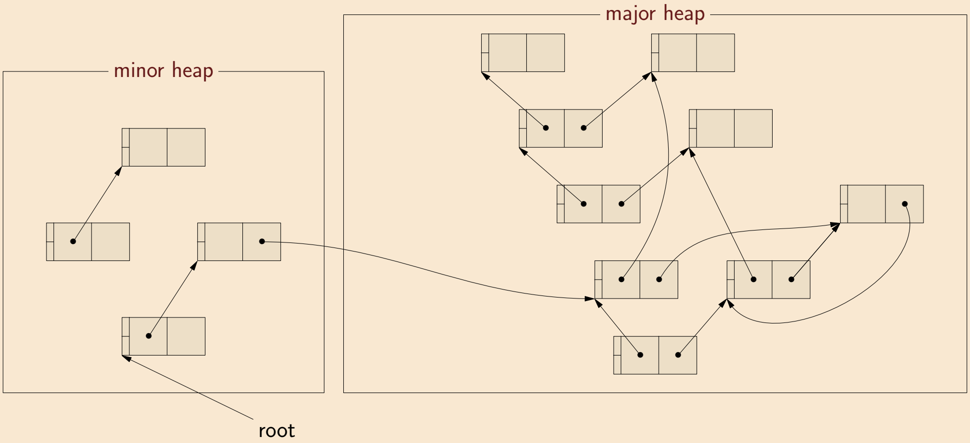
Mechanism:

- divide heap into $2+$ generations
- frequently collect young generations (fast)
- promote surviving objects to old generations
- occasionally collect old generations (slow)

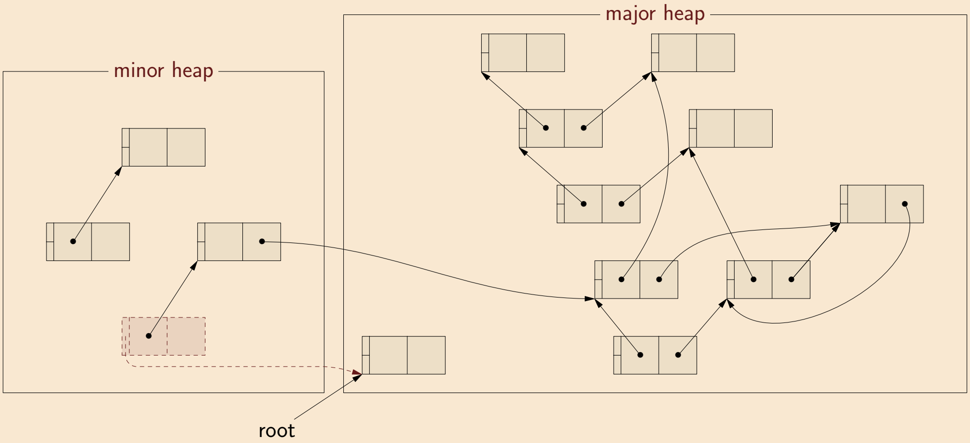
Many **variations** (e.g. generations can use different collection schemes)

Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap

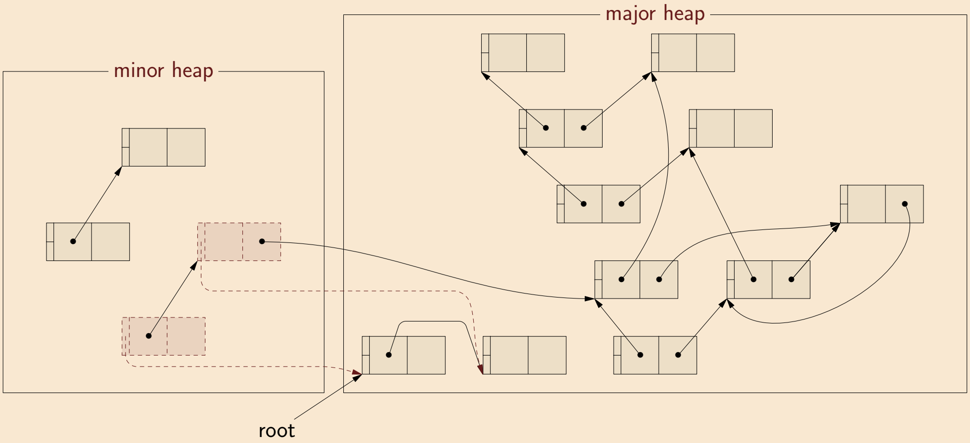


Copying collector for minor heap / mark-and-sweep for major heap



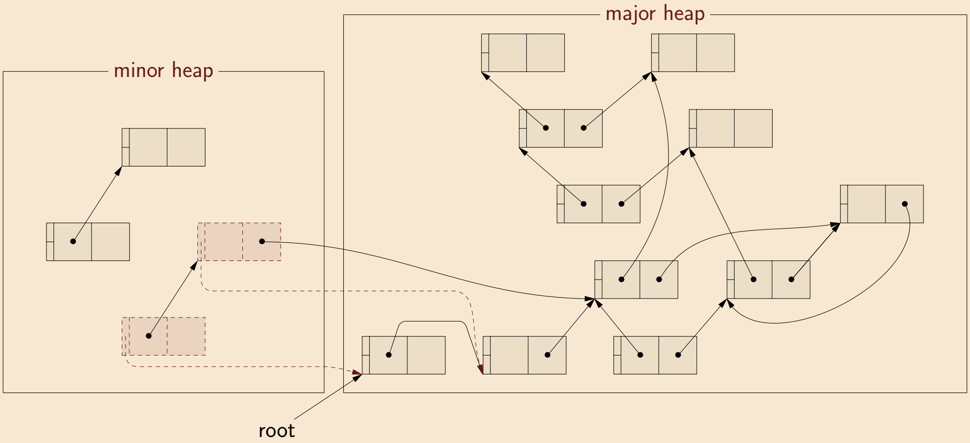
Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap



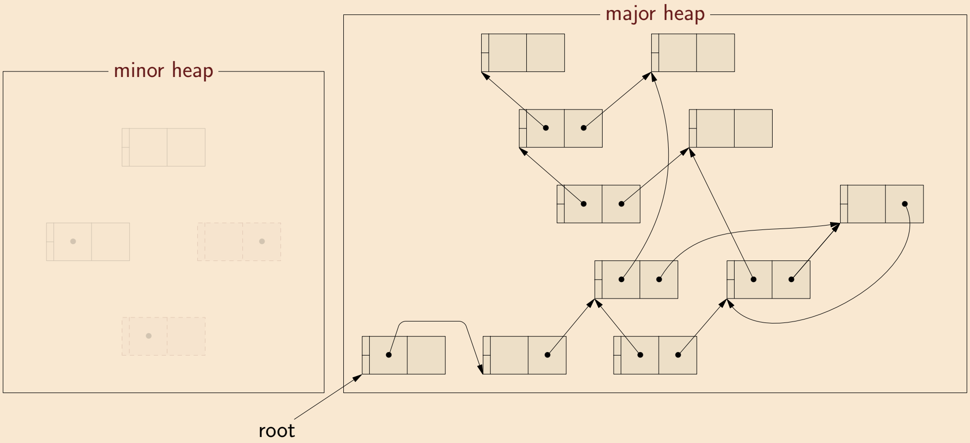
Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap



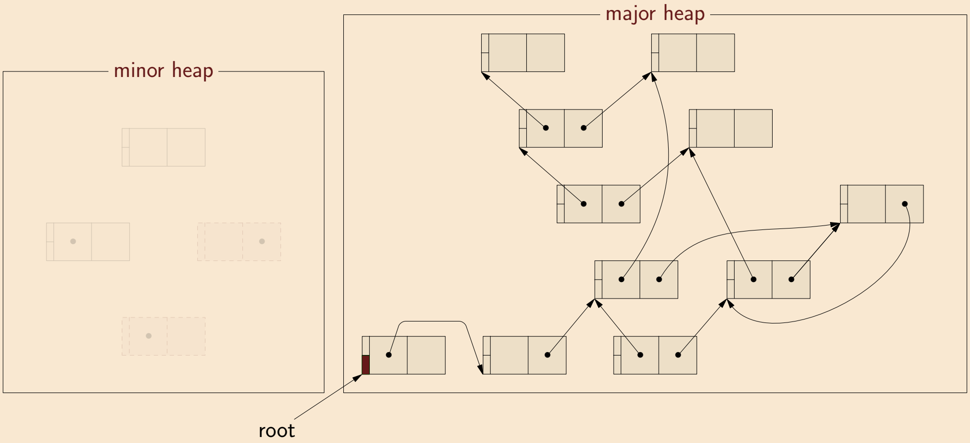
Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap



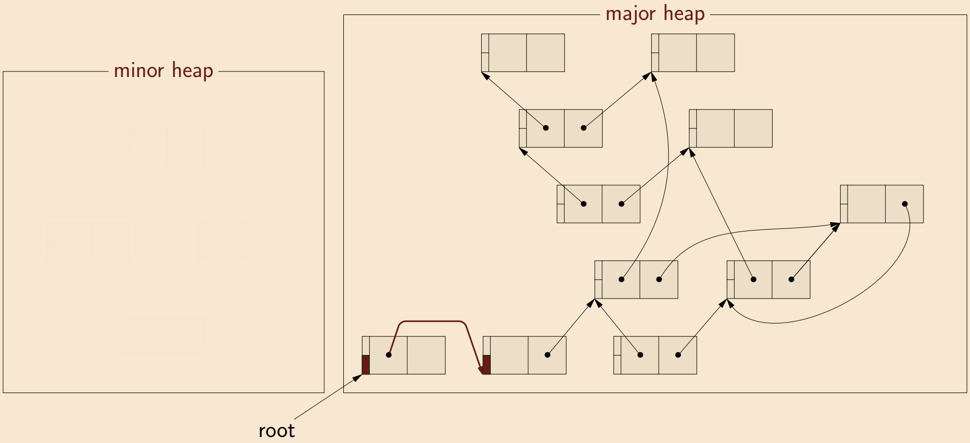
Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap

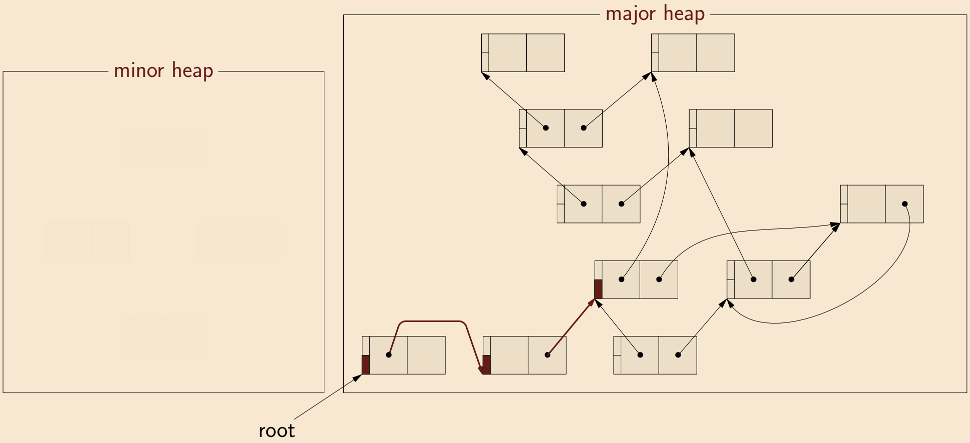


Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap

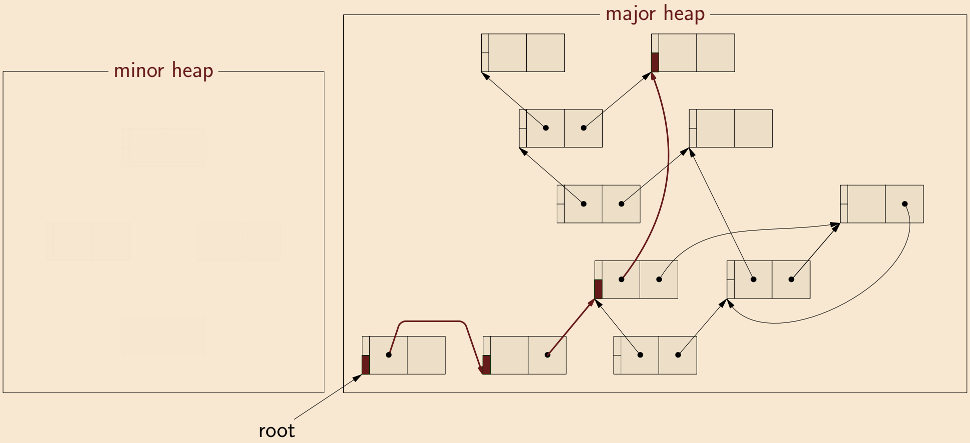


Copying collector for minor heap / mark-and-sweep for major heap

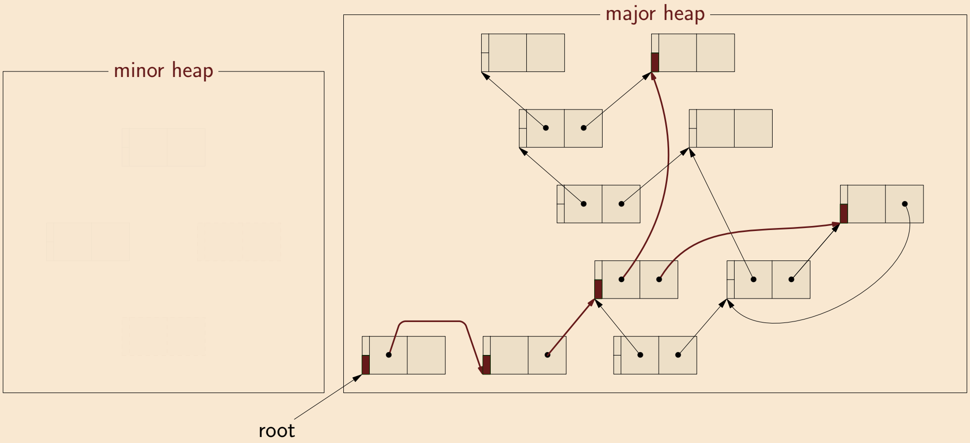


Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap

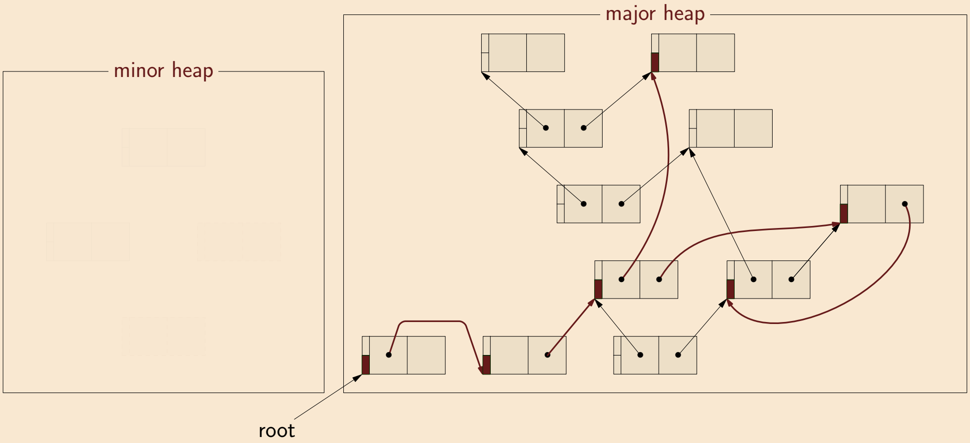


Copying collector for minor heap / mark-and-sweep for major heap



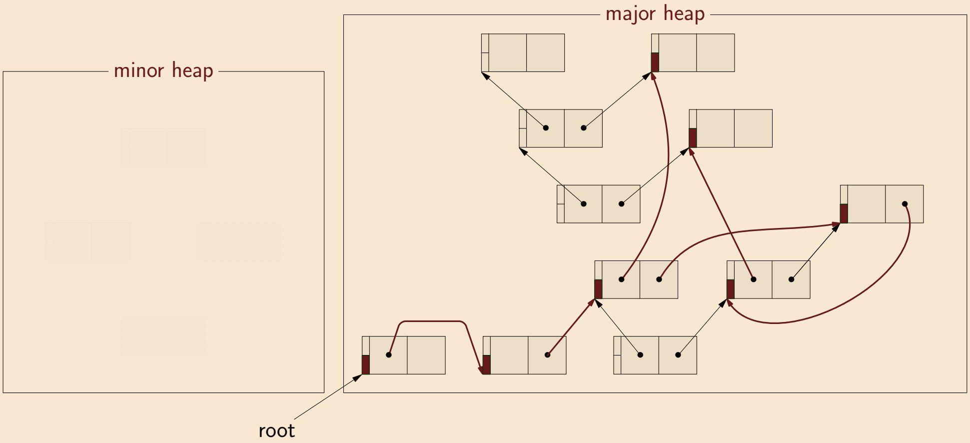
Memory
Reference counting
Mark & sweep
Copying
Generations

Copying collector for minor heap / mark-and-sweep for major heap



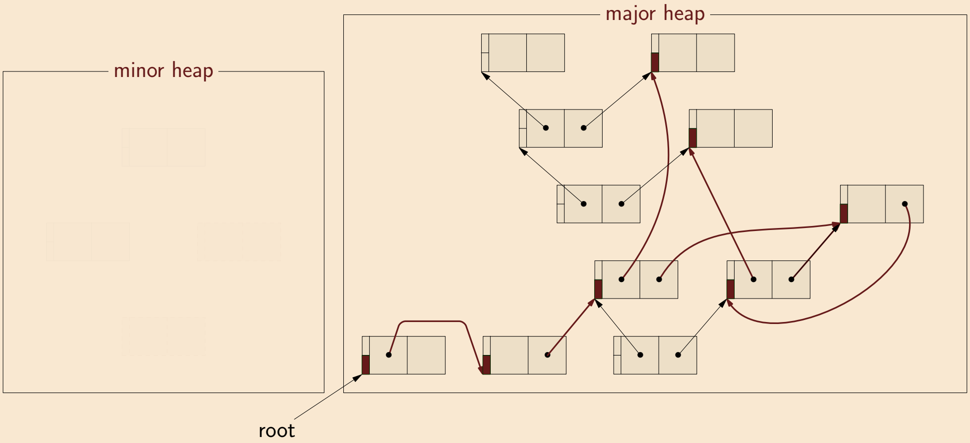
Memory
Reference counting
Mark & sweep
Copying
Generations

Copying collector for minor heap / mark-and-sweep for major heap



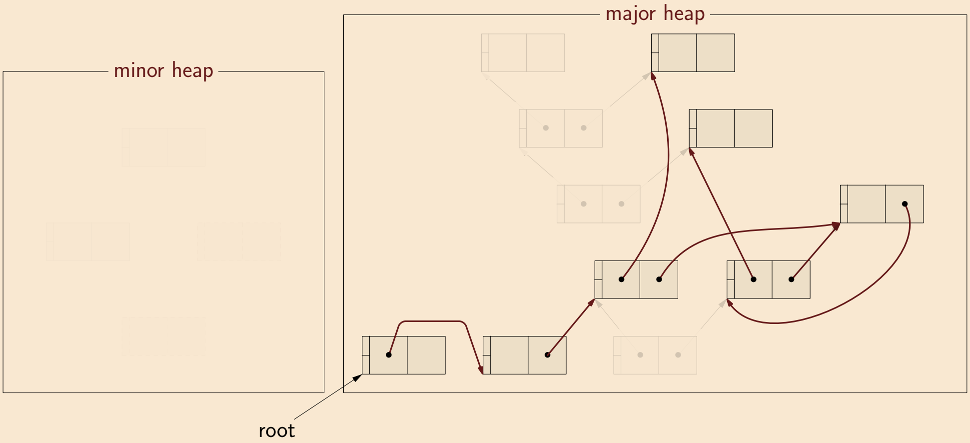
Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap



Generational GC: example

Copying collector for minor heap / mark-and-sweep for major heap



Generational GC: advantages & complexities

Advantages of generational garbage collection:

+ reduce pauses (to $100\mu s$ or less; suitable for interactive programs)
+ avoid wasted time scanning long-lived objects

Complexities of generational garbage collection:

- must distinguish between old & young pointers
- hard to find generation roots (consider pointers from old to young objects)
- can use > 2 generations, all with different policies

Next time: exceptions