Garbage Collection

Guido Wachsmuth, Eelco Visser

Garbage Collection
outline

Reference counting

- deallocate records with count 0

Mark & sweep

- mark reachable records
- sweep unmarked records

Copying collection

- copy reachable records

Generational collection

- collect only in young generations of records

I

Memory Safety & Management

Memory Safety

A program execution is memory safe if

- It only creates valid pointers through standard means
- Only uses a pointer to access memory that belongs to that pointer

Combines temporal safety and spatial safety

Spatial Safety

Access only to memory that pointer owns

View pointer as triple (p, b, e)

- p is the actual pointer
- b is the based of the memory region it may access
- e is the extent (bounds of that region)

Access allowed iff

- b <= p <= e - sizeof(typeof(p))

Allowed operations

- Pointer arithmetic increments p, leaves b and e alone
- Using &: e determined by size of original type

Temporal Safety
No access to undefined memory

Temporal safety violation: trying to access undefined memory

- Spatial safety assures it was to a legal region
- Temporal safety assures that region is still in play

Memory region is defined or undefined

Undefined memory is

- unallocated
- uninitialized
- deallocated (dangling pointers)

Memory Management
safety guarantees

Manual memory management

- malloc, free in C
- Easy to accidentally free memory that is still in use
- Pointer arithmetic is unsafe

Automated memory management

- Spatial safety: references are opaque (no pointer arithmetic)
- (+ array bounds checking)
- Temporal safety: no dangling pointers (only free unreachable memory)

Garbage Collector

Terminology

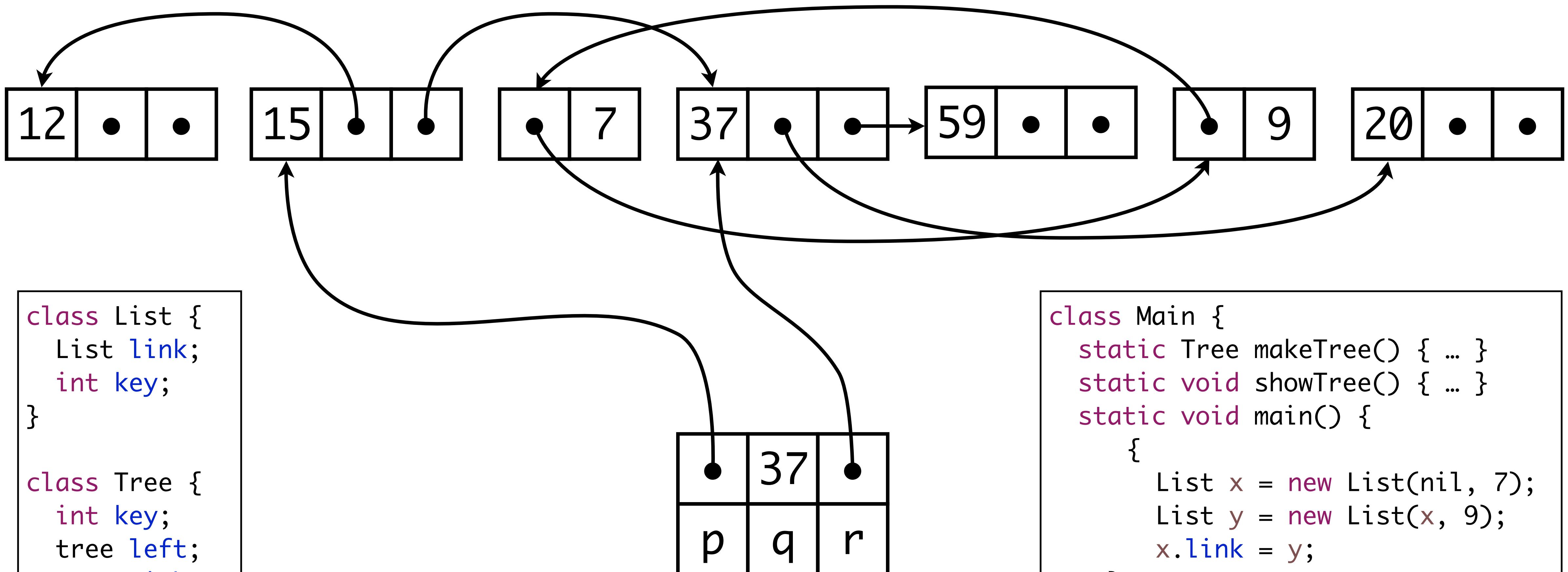
- objects that are referenced are **live**
- objects that are not referenced are **dead (garbage)**
- objects are allocated on the **heap**

Responsibilities

- allocating memory
- ensuring live objects remain in memory
- garbage collection: recovering memory from dead objects

```
class List {
  List link;
  int key;
}

class Tree {
  int key;
  tree left;
  tree right;
}
```

```
class Main {
  static Tree makeTree() { … }
  static void showTree() { … }
  static void main() {
    {
      List x = new List(nil, 7);
      List y = new List(x, 9);
      x.link = y;
    }
    {
      Tree p = maketree();
      Tree r = p.right;
      int q = r.key;
      // garbage-collect here
      showtree(p)
    }
  }
}
```




```
class List {
  List link;
  int key;
}

class Tree {
  int key;
  tree left;
  tree right;
}
```

```
class Main {
  static Tree makeTree() { … }
  static void showTree() { … }
  static void main() {
     {
       List x = new List(nil, 7);
       List y = new List(x, 9);
       x.link = y;
     }
     {
        Tree p = maketree();
        Tree r = p.right;
        int q = r.key;
        // garbage-collect here
        showtree(r)
     }
  }
}
```

II

Reference Counting

Reference Counting
idea

Counts

- how many pointers point to each record?
- store with each record

Counting

- extra instructions

Deallocate

- put on freelist
- recursive deallocation on allocation

Reference Counting

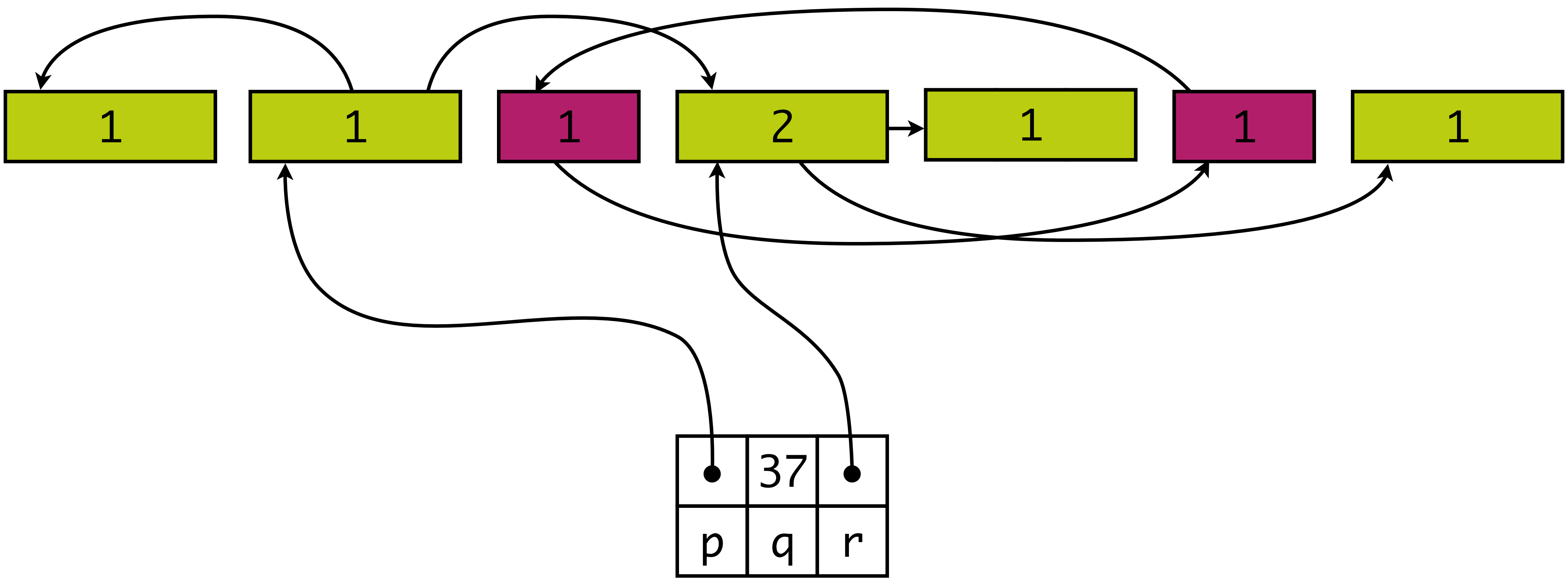
compiler instrumentation

```
x.f := p
```

```
z       := x.f
c       := z.count
c       := c - 1
z.count := c
if (c == 0) put z on free list
x.f     := p
c       := p.count
c       := c + 1
p.count := c
```

Reference Counting

example



Reference Counting
notes

Cycles

- memory leaks
- break cycles explicitly
- occasional mark & sweep collection

Expensive

- fetch, decrease, store old reference counter
- possible deallocation
- fetch, increase, store new reference counter

Reference Counting
programming languages

Languages with automatic reference counting

- Objective-C, Swift

Dealing with cycles

- strong reference: counts as a reference
- weak reference: can be nil, does not count
- unowned references: cannot be nil, does not count

III

Mark & Sweep

Mark & Sweep
idea

Mark

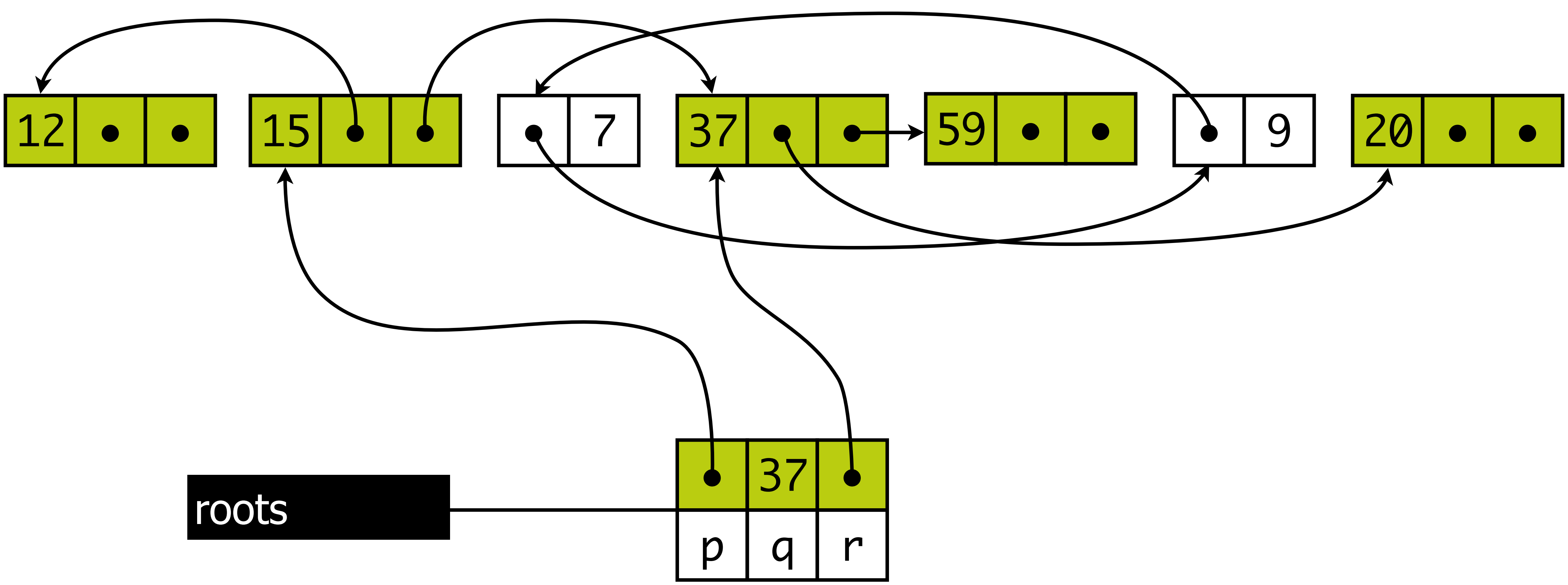
- mark reachable records
- start at variables (roots)
- follow references

Sweep

- marked records: unmark
- unmarked records: deallocate
- linked list of free records

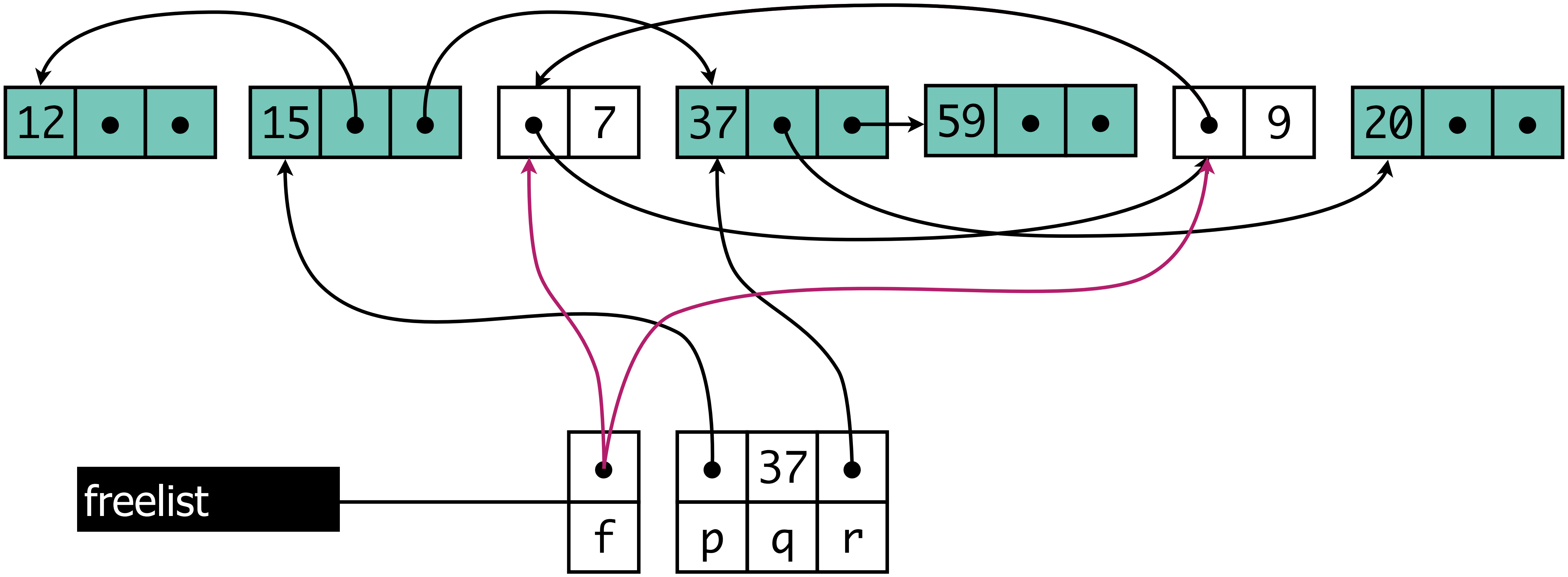
Marking

example



Sweeping

example



Mark & Sweep
algorithms

```
Mark phase:

  foreach r in roots

    DFS(r)
```

```
function DFS(x)

  if pointer(x) & !x.marked

    x.marked := true

    foreach f in fields(x)

      DFS(f)
```

```
Sweep phase:

  p := first address in heap

  while p < last address in heap
    if p.marked

      p.marked := false

    else

      f1 := first field in p
      p.f1 := freelist
      free list := p

    p := p + sizeof( p )
```

Mark & Sweep
costs

Instructions

- R reachable words in heap of size H
- Mark: c1 * R
- Sweep: c2 * H
- Reclaimed: H - R words
- Instructions per word reclaimed:

  (c1 * R + c2 * H) / (H - R)
- if (H >> R) cost per allocated word ~ c2

Mark & Sweep
costs

Memory

- DFS is recursive
- maximum depth: longest path in graph of reachable data
- worst case: H
- | stack of activation records | > H

Measures

- explicit stack
- pointer reversal

Marking: DFS with Explicit Stack

algorithms

```
function DFS(x)

  if pointer(x) & !x.marked

    x.marked = true

    foreach f in fields(x)

      DFS(f)
```

```
function DFS(x)

  if pointer(x) & !x.marked

    x.marked = true
    t = 1 ; stack[t] = x

    while t > 0

      x = stack[t] ; t = t - 1

      foreach f in fields(x)
        if pointer(f) & !f.marked

          f.marked = true
          t = t + 1 ; stack[t] = f
```

Marking: DFS with Pointer Reversal

marking without memory overhead

```
function DFS(x)
  if pointer(x) & x.done < 0
    x.done = 0 ; t = nil

    while true
      if x.done < x.fields.size
        y = x.fields[x.done]
        if pointer(y) & y.done < 0
          x.fields[x.done] = t ; t = x ; x = y ; x.done = 0
        else
          x.done = x.done + 1

      else
        y = x; x = t
        if t = nil then return
        t = x.fields[x.done]; x.fields[x.done] = y
        x.done = x.done + 1
```

Mark & Sweep
notes

Sweeping

- independent of marking algorithm
- several freelists (per record size)
- split free records for allocation

Fragmentation

- external: many free records of small size
- internal: too-large record with unused memory inside

IV

Copying Collection

Copying Collection
idea

Spaces

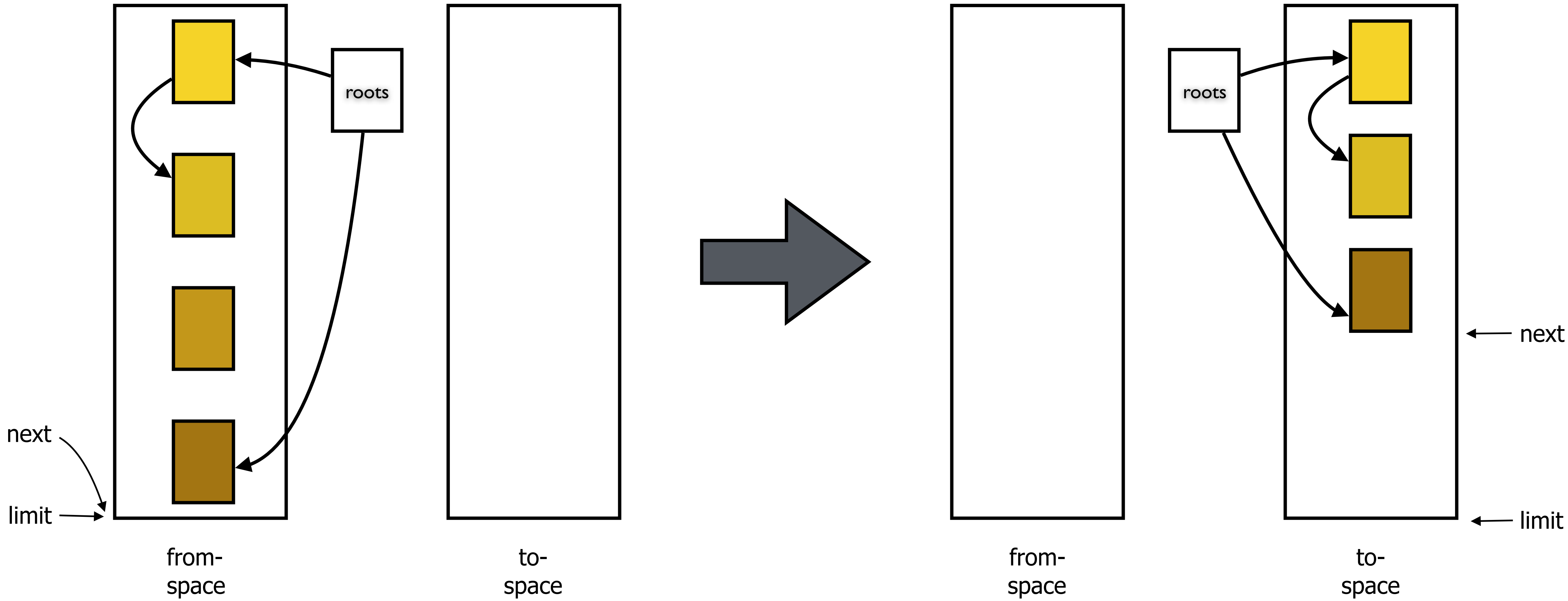
- fromspace & tospace
- switch roles after copy

Copy

- traverse reachability graph
- copy from fromspace to tospace
- fromspace unreachable, free memory
- tospace compact, **no fragmentation**

Copying Collection

idea



Copying Collection

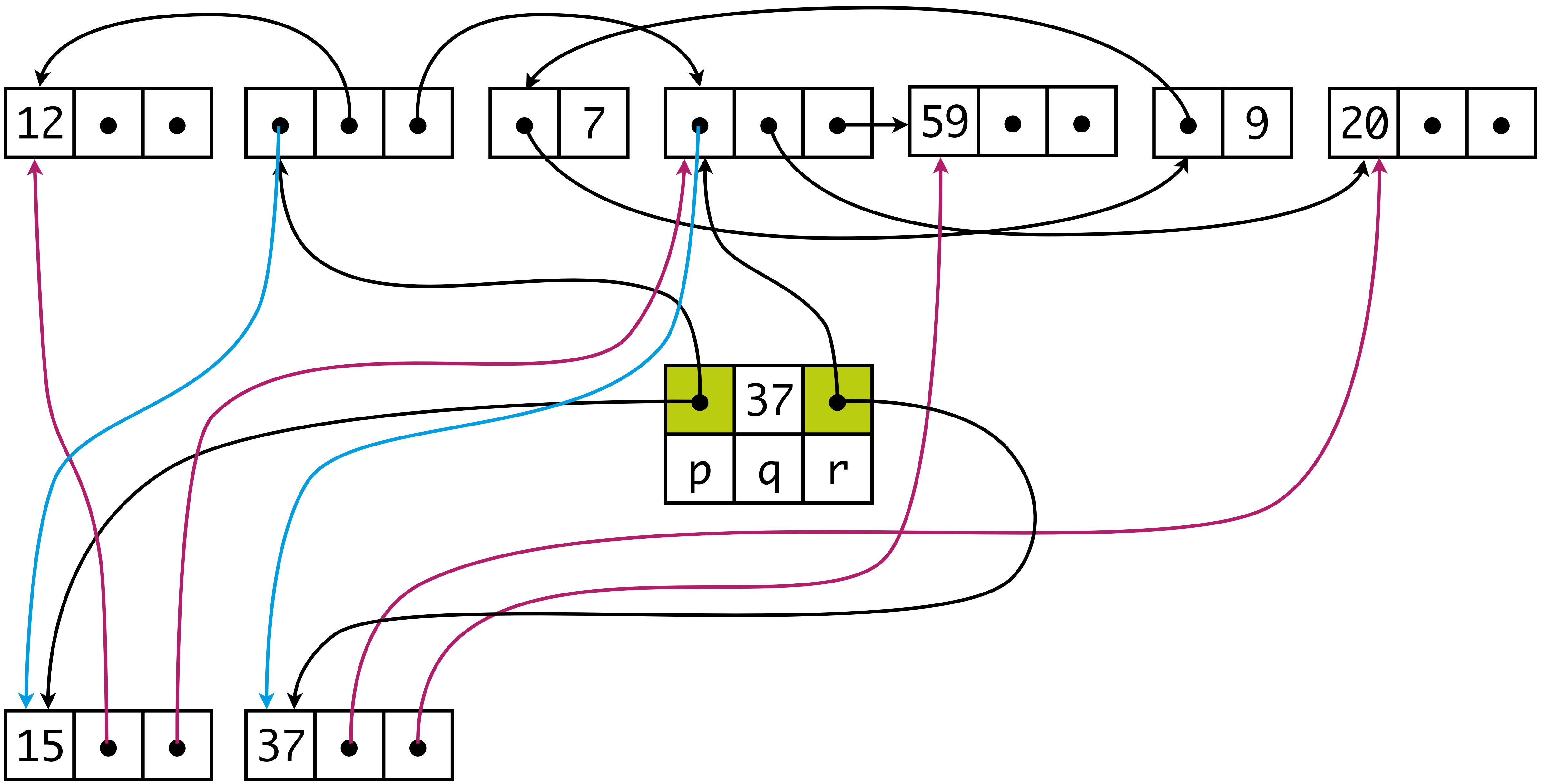
algorithms

```
function Forward(p)

  if p in fromspace
    if p.f1 in tospace
      return p.f1

    else

      foreach f in fields of p
        next.f := p.f

      p.f1 := next
      next := next + sizeof(p)
      return p.f1

  else return p
```

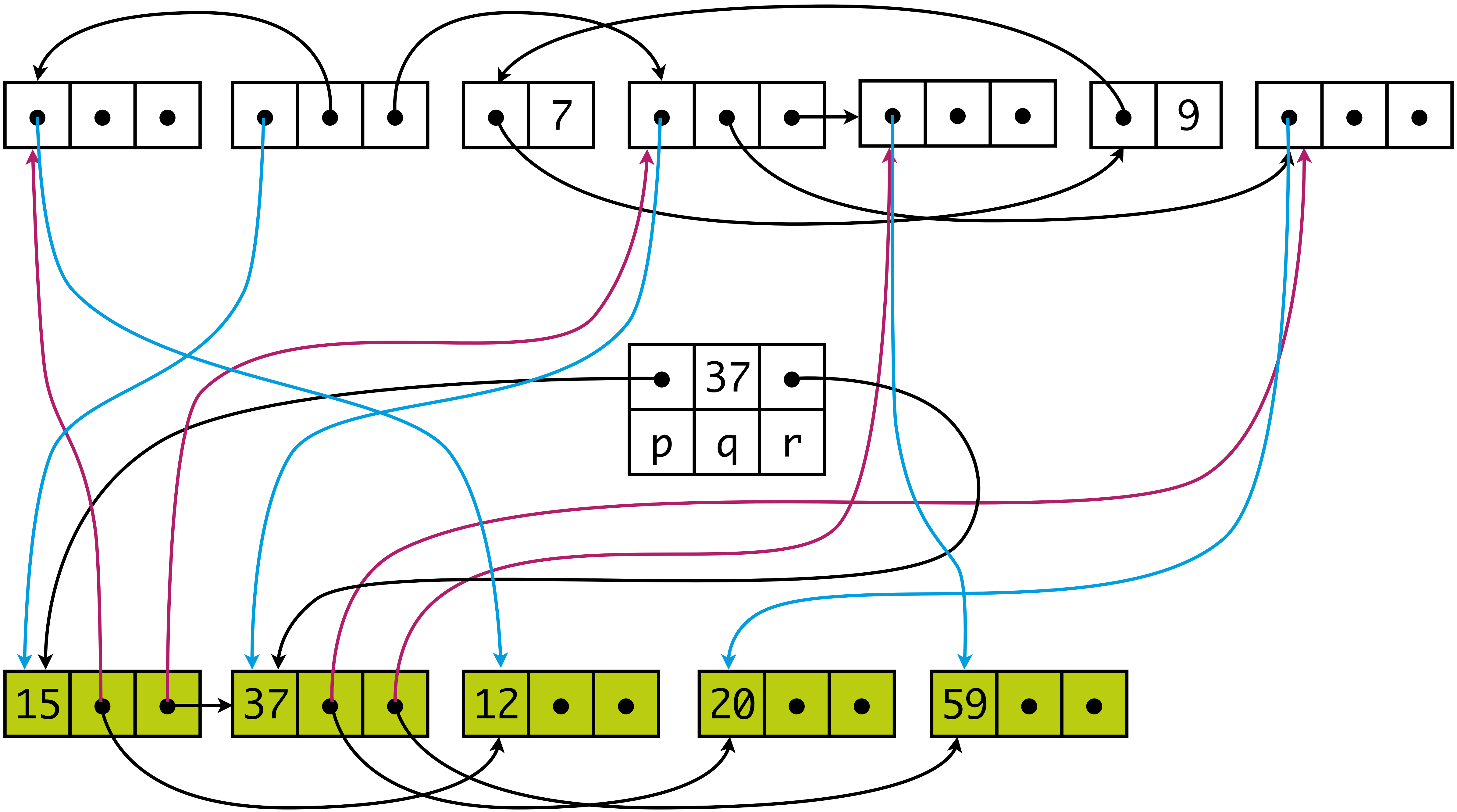
```
function BFS()

  next := scan := start(tospace)

  foreach r in roots
    r = Forward(r)

  while scan < next

    foreach f in fields of scan
      scan.f = Forward(scan.f)

    scan = scan + sizeof(scan)
```

Copying Collection

example



Copying Collection



Copying Collection
locality

Adjacent records

- likely to be unrelated

Pointers to records in records

- likely to be accessed
- likely to be far apart

Solution

- depth-first copy: slow pointer reversals
- hybrid copy algorithm

Copying Collection

costs

Instructions

- R reachable words in heap of size H
- BFS: c3 * R
- No sweep
- Reclaimed: H/2 - R words
- Instructions per word reclaimed: (c3 * R) / (H/2 - R)
- If (H >> R) : cost per allocated word => 0
- If (H = 4R) : c3 instructions per word allocated
- Solution: reduce portion of R to inspect => generational collection

V

Generational Collection

Generational Collection

idea

Generations

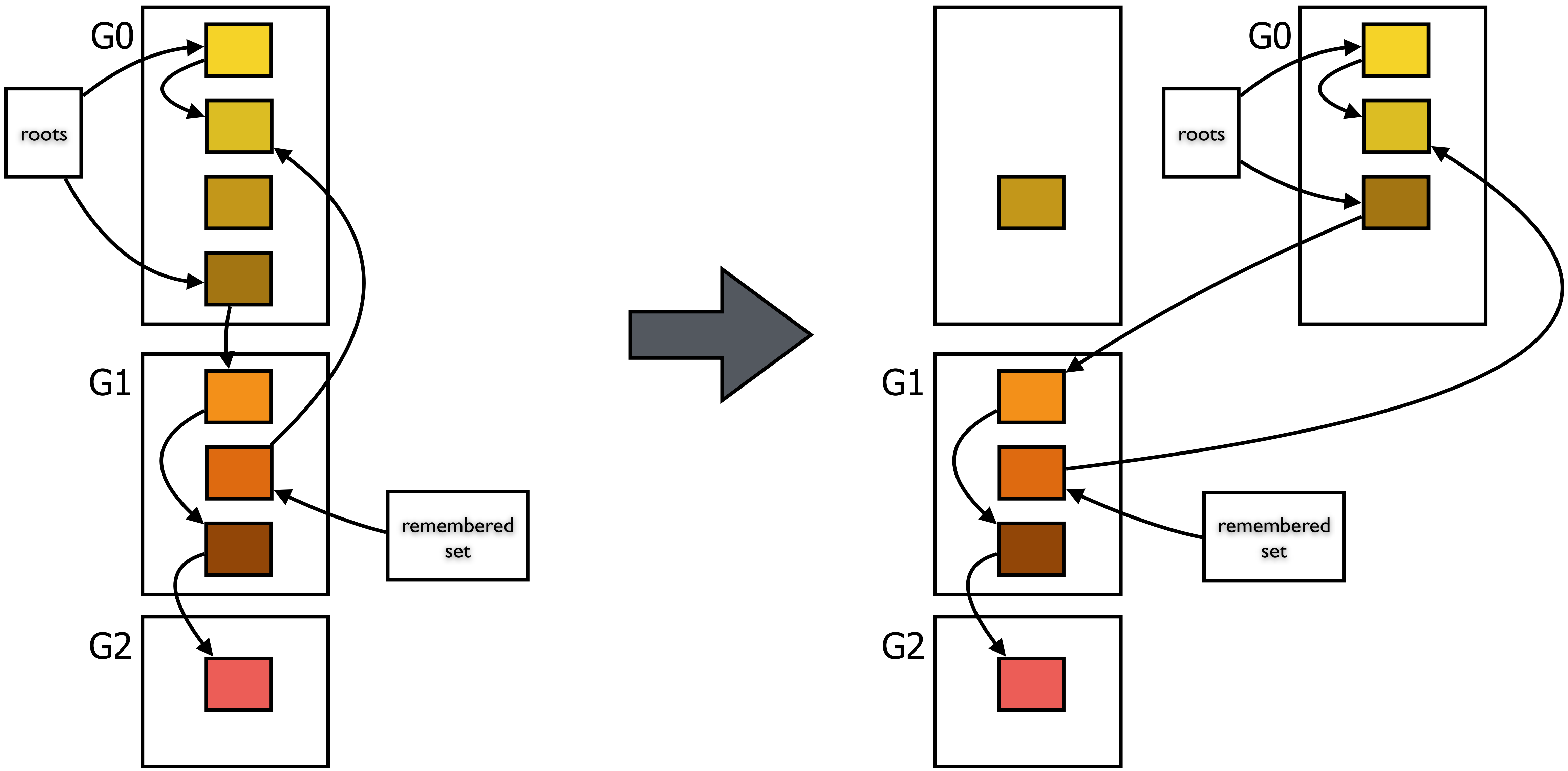
- young data: likely to die soon
- old data: likely to survive for more collections
- divide heap, collect younger generations more frequently

Collection

- roots: variables & pointers from older to younger generations
- preserve pointers to old generations
- promote objects to older generations

Generational Collection

idea



Generational Collection

costs

Instructions

- R reachable words in heap of size H
- BFS: c3 * R
- No sweep
- 10% of youngest generation is live: H/R = 10
- Instructions per word reclaimed:
  (c3 * R) / (H - R) = (c3 * R) / (10R - R) ~= c3/10
- Adding to remembered set: 10 instructions per update

Incremental Collection

idea

Interrupt by garbage collector undesirable

- interactive, real-time programs

Incremental / concurrent garbage collection

- interleave collector and mutator (program)
- incremental: per request of mutator
- concurrent: in between mutator operations

Tricolor marking

- White: not visited
- Grey: visited (marked or copied), children not visited
- Black: object and children marked

VI

Summary

Algorithms

summary

How can we collect unreachable records on the heap?

- reference counts
- mark reachable records, sweep unreachable records
- copy reachable records

How can we reduce heap space needed for garbage collection?

- pointer-reversal
- breadth-first search
- hybrid algorithms

Design Choices
summary

Serial vs Parallel

- garbage collection as sequential or parallel process

Concurrent vs Stop-the-World

- concurrently with application or stop application

Compacting vs Non-compacting vs Copying

- compact collected space
- free list contains non-compacted chunks
- copy live objects to new space; from-space is non-fragmented

Performance Metrics

summary

Throughput

- percentage of time not spent in garbage collection

GC overhead

- percentage of time spent in garbage collection

Pause time

- length of time execution is stopped during garbage collection

Frequency of collection

- how often collection occurs

Footprint

- measure of (heap) size

Garbage Collection in Java HotSpot VM
practice

Serial collector

- young generation: copying collection
- old generation: mark-sweep-compact collection

Parallel collector

- young generation: stop-the-world copying collection in parallel
- old generation: same as serial

Parallel compacting collector

- young generation: same as parallel
- old generation: roots divided in threads, marking live objects in parallel, …

Concurrent Mark-Sweep (CMS) collector

- stop-the-world initial marking and re-marking
- concurrent marking and sweeping

Literature
learn more

Andrew W. Appel, Jens Palsberg. Modern Compiler Implementation in Java, 2nd edition, 2002.

Sun Microsystems. Memory Management in the Java HotSpotTM Virtual Machine, April 2006.

copyrights