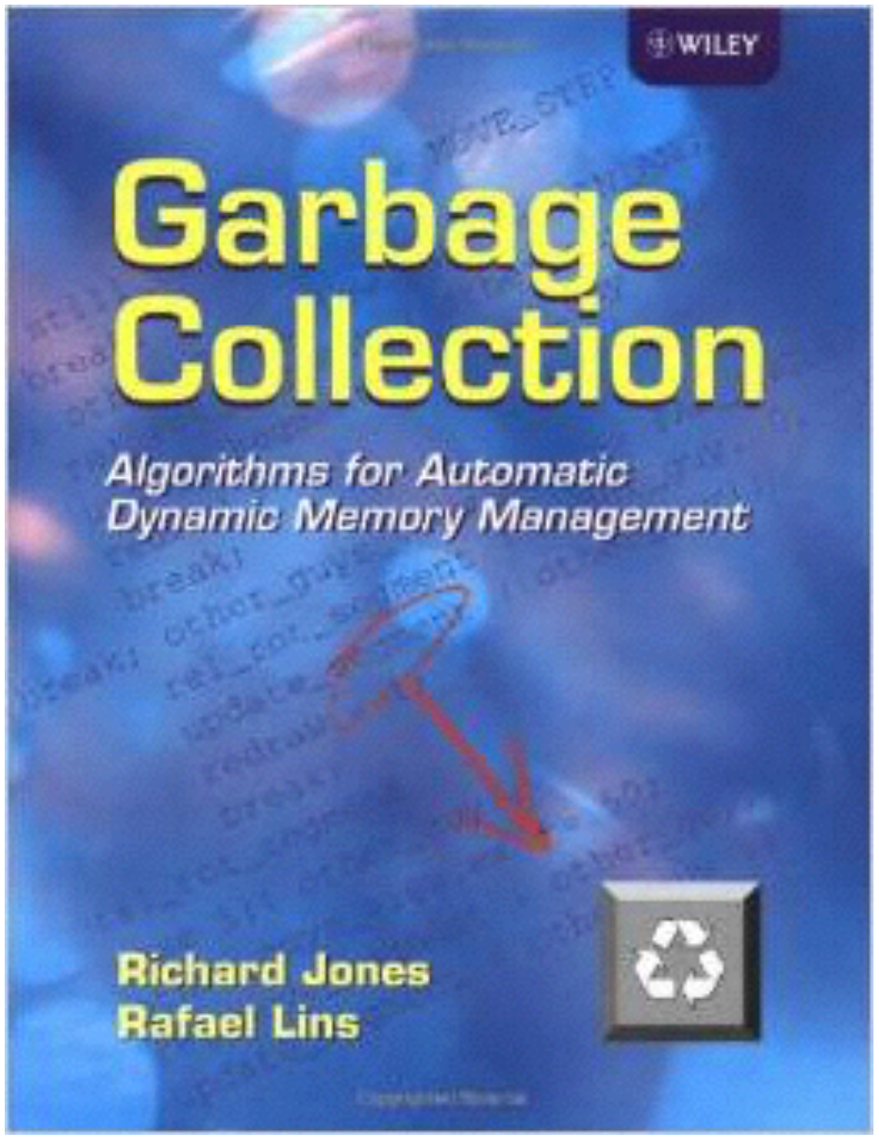
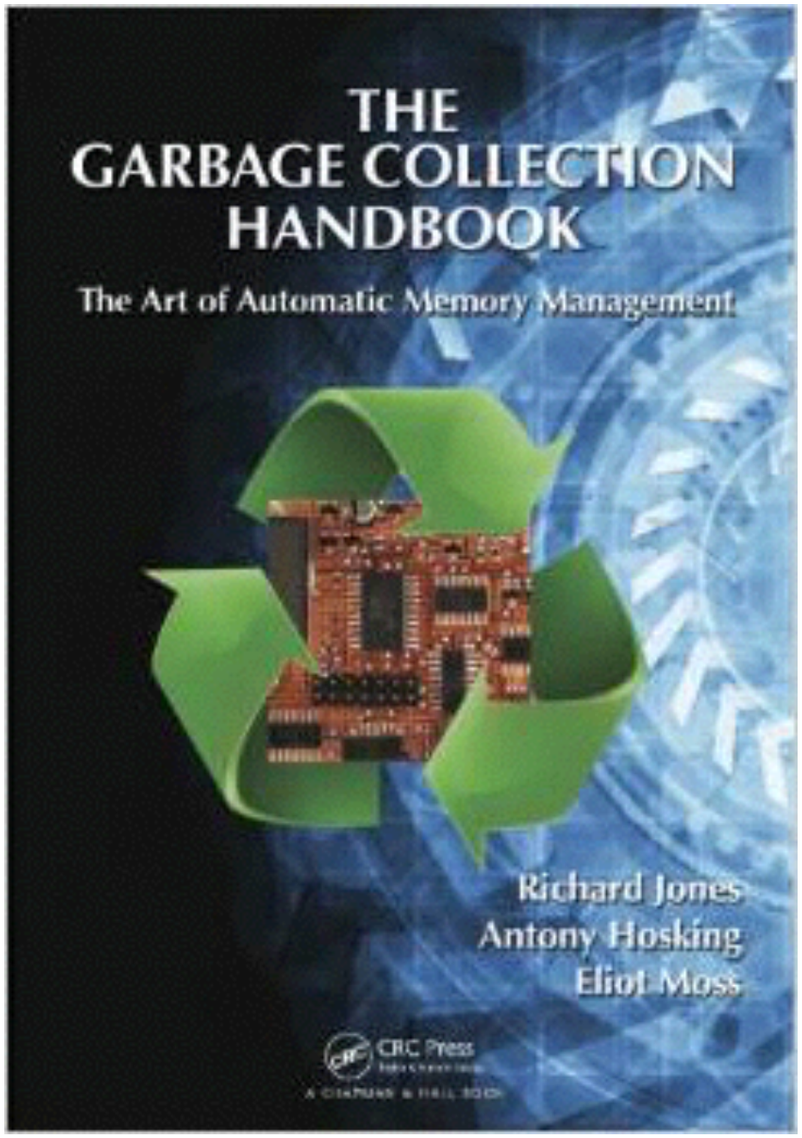
Memory Management

© 2015 Oracle, Mario Wolczko

Memory management
Part (1)
Introduction

Textbooks


WILEY
Garbage Collection
Algorithms for Automatic Dynamic Memory Management
Richard Jones
Rafael Lins


THE GARBAGE COLLECTION HANDBOOK
The Art of Automatic Memory Management
Richard Jones
Antony Hosking
Eliot Moss
CRC Press

Managing memory: 1. Static allocation

- All bindings of names to locations are made at compile/link time. All sizes are fixed at this time. There is not even a stack.
- Used in early FORTRAN, BASIC, and various languages for embedded/real-time*
- Pros: no runtime allocation and associated overheads

  Mainstream.

  - total memory usage known at compile time
  - cannot fail at runtime due to lack of memory

Static allocation

- Cons:
  - Need to allocate at compile time for the worst case (memory for all locals; maximally sized structures, etc.)
  - Fixed subdivision can't adapt to circumstances at run-time (e.g., some part needs more, temporarily)
  - Can't use recursion - return address and locals are in fixed locations
- Still used in applications where running out of memory is a critical failure
  - Java Real-Time Spec allows only static allocation in critical parts
- * Q. Why is this approach used for embedded/real-time applications?

Edge code may fail. while core component still running.

Managing memory: 2. Dynamic allocation (a) stack

- Most languages are based on procedures (aka functions, subroutines, methods)
- Invocation follows a Last-In-First-Out discipline
- Memory most easily managed as a stack
- Most processors have included a stack pointer and call/return instructions since the 1960s

Non-LIFO activations

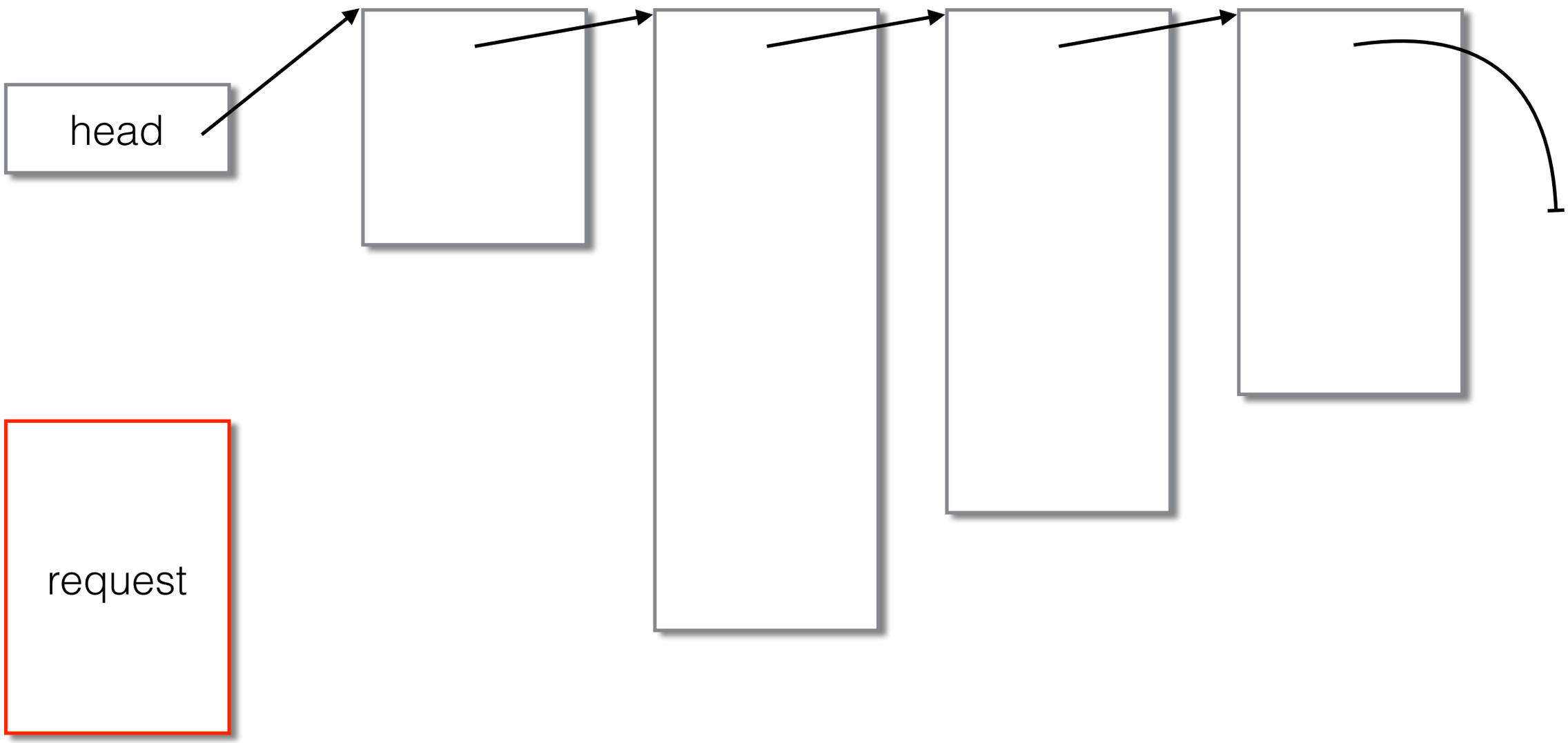
- Some languages don’t exactly follow the LIFO approach
  - Continuations in Scheme and other languages
  - Smalltalk: each activation record is a first-class object and can be referenced via *thisContext*.
- Implies that activations must be heap-allocated😱—but recall the tricks in the Smalltalk paper…

Managing memory: 2. Dynamic allocation (b) heap

The ***heap*** is a pool of memory. Allocation requests can be of any size, in any order (and can fail if they cannot be satisfied). The pool may consist of many separate contiguous areas. In some environments, new areas can be requested from the OS (and it may be possible and desirable to give back unneeded areas to the OS).

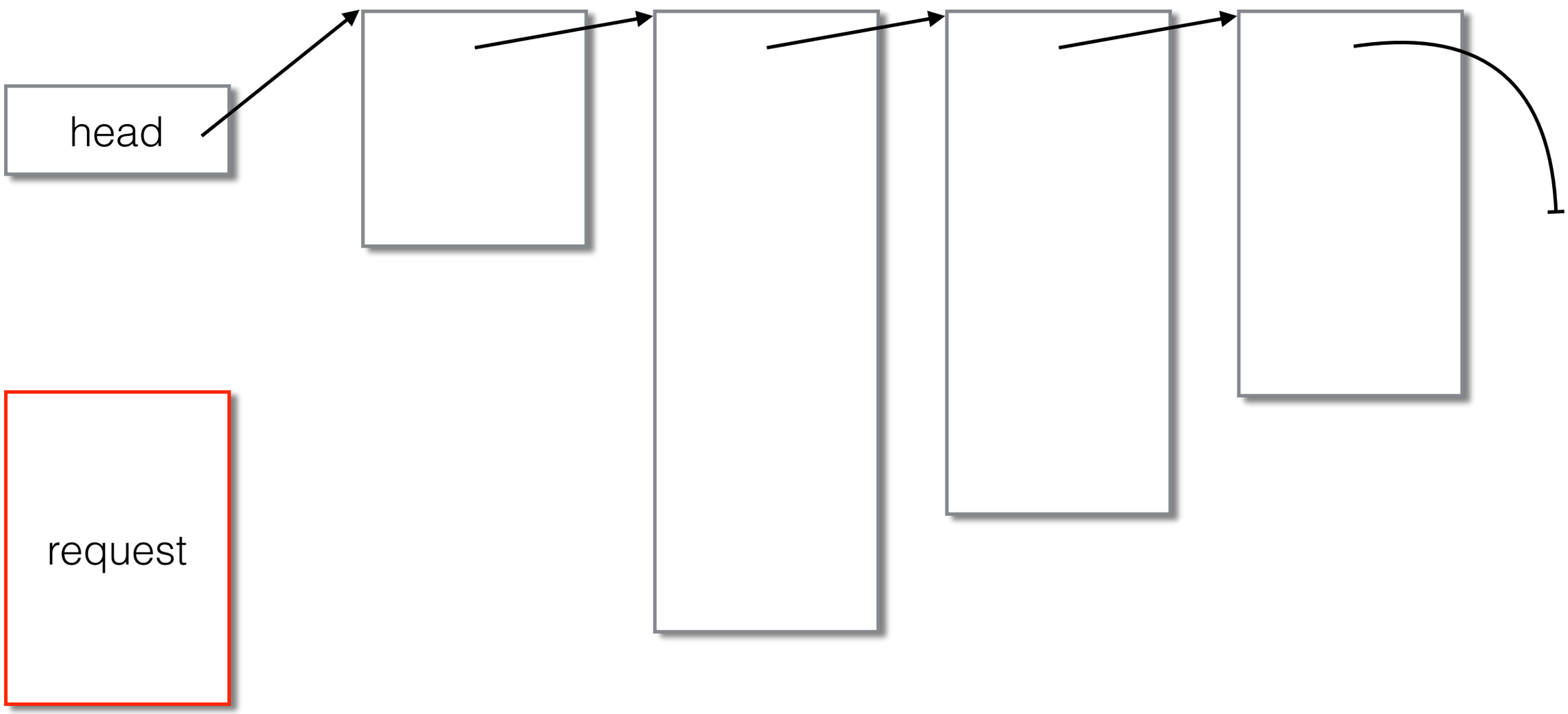
- First order problems: Allocation, Deallocation

Classic allocation from a free list

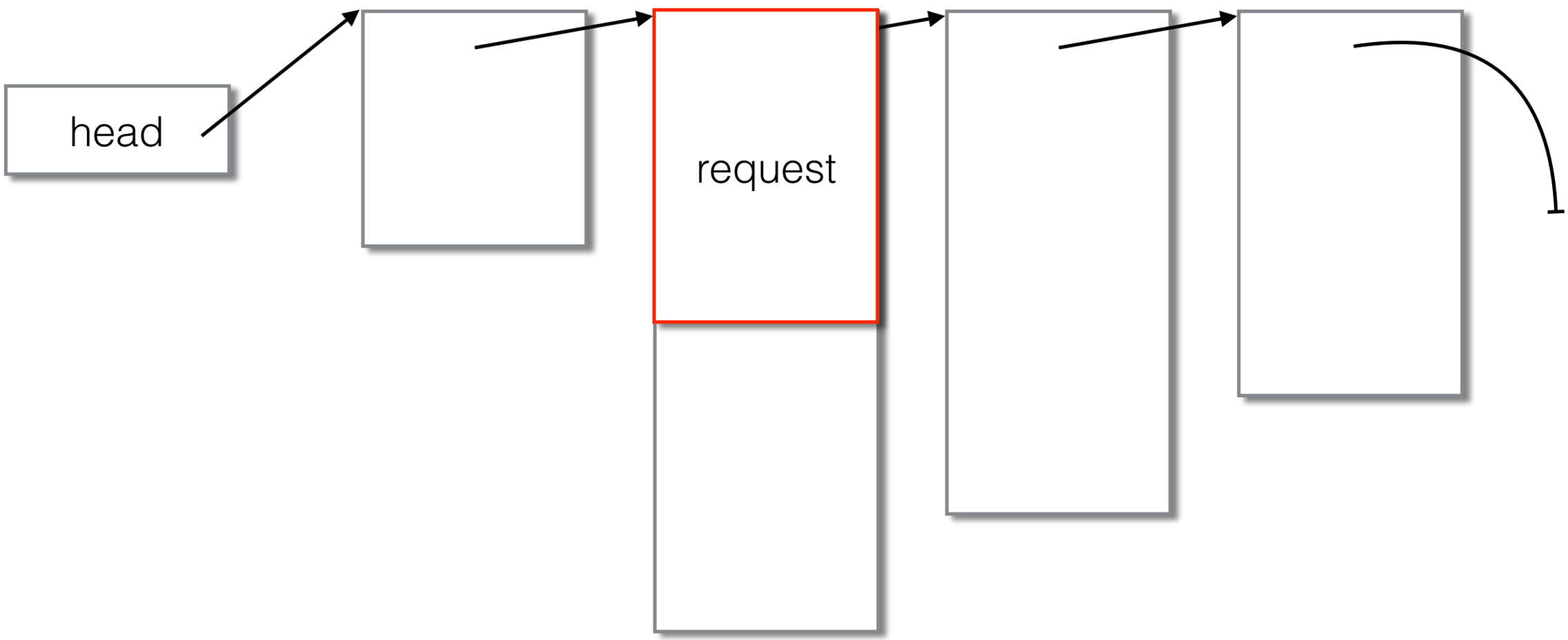


First fit: the first memory larger than request.

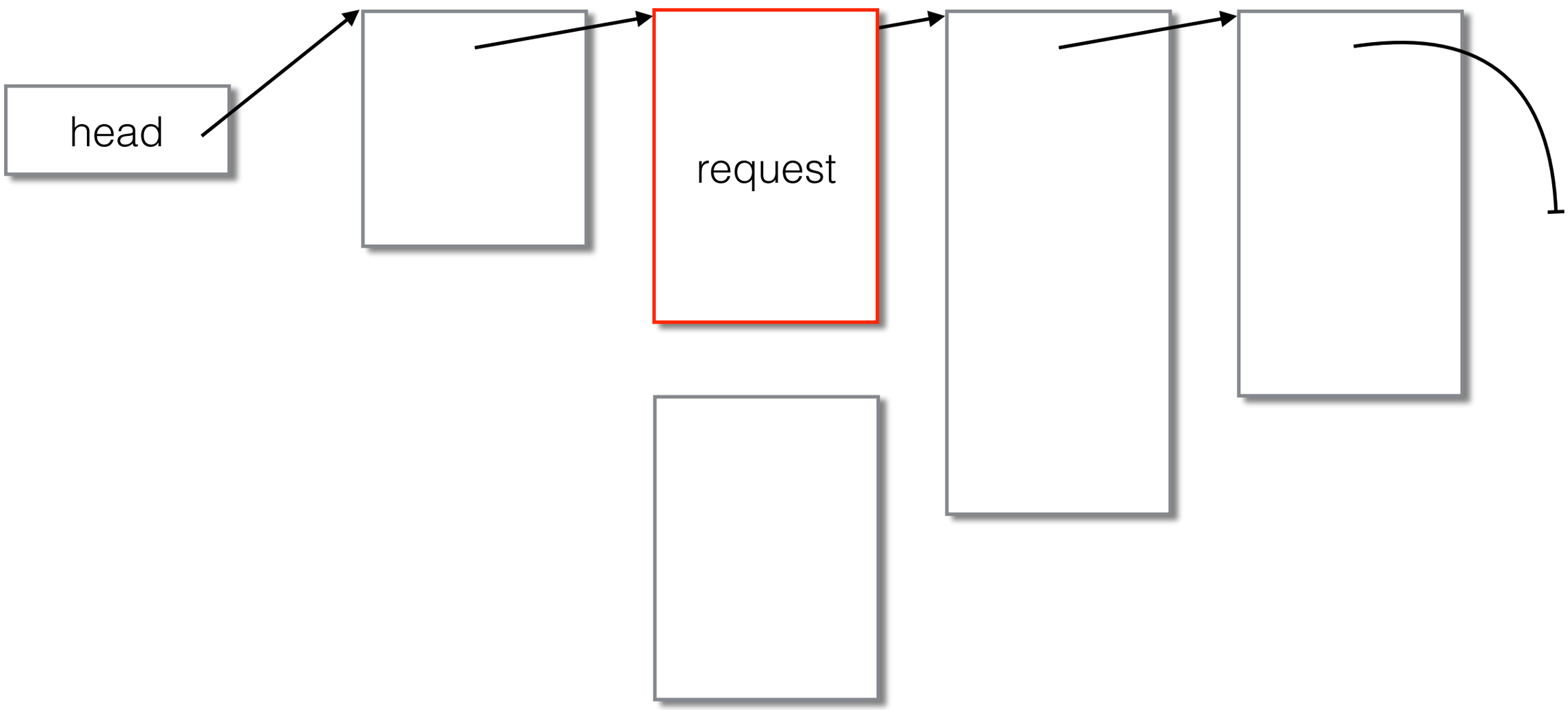
First fit



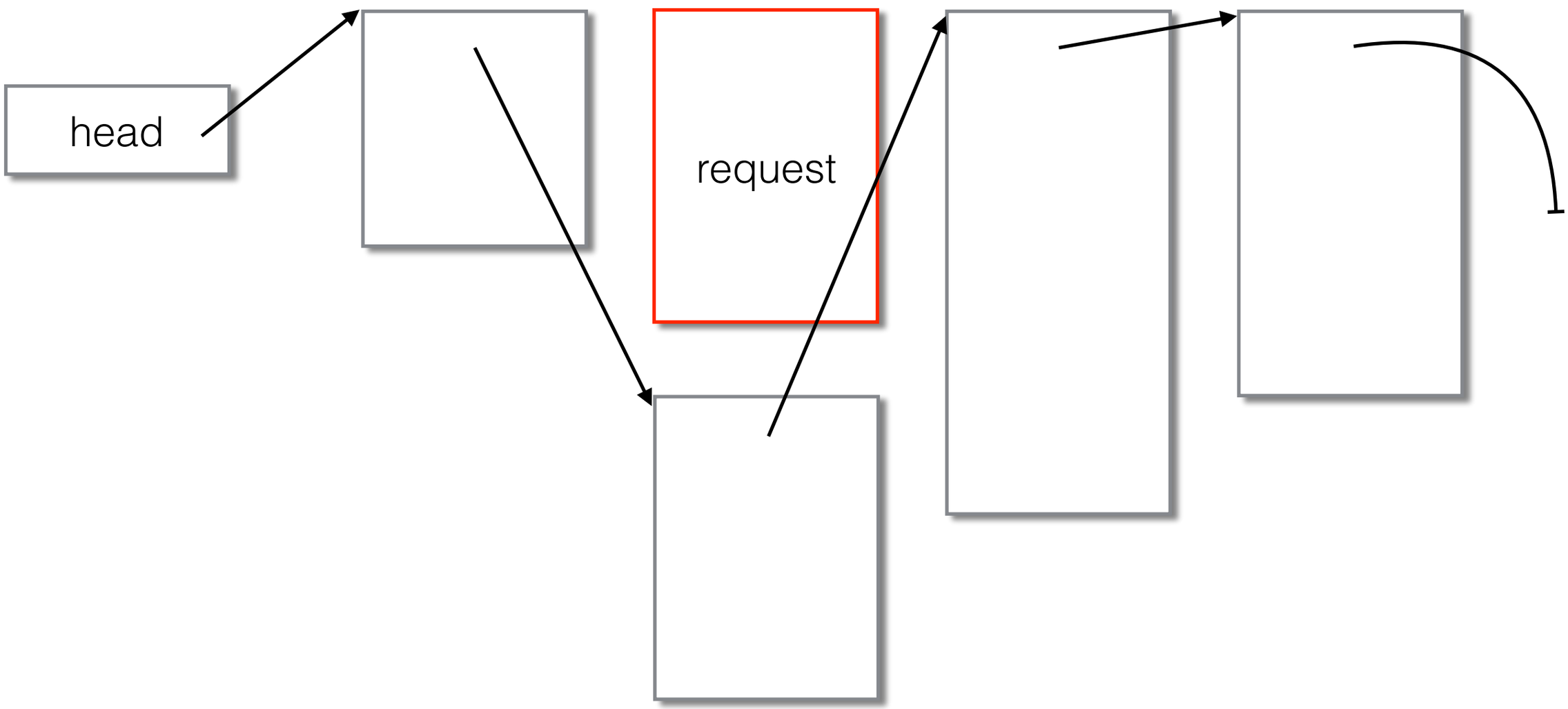
First fit



First fit



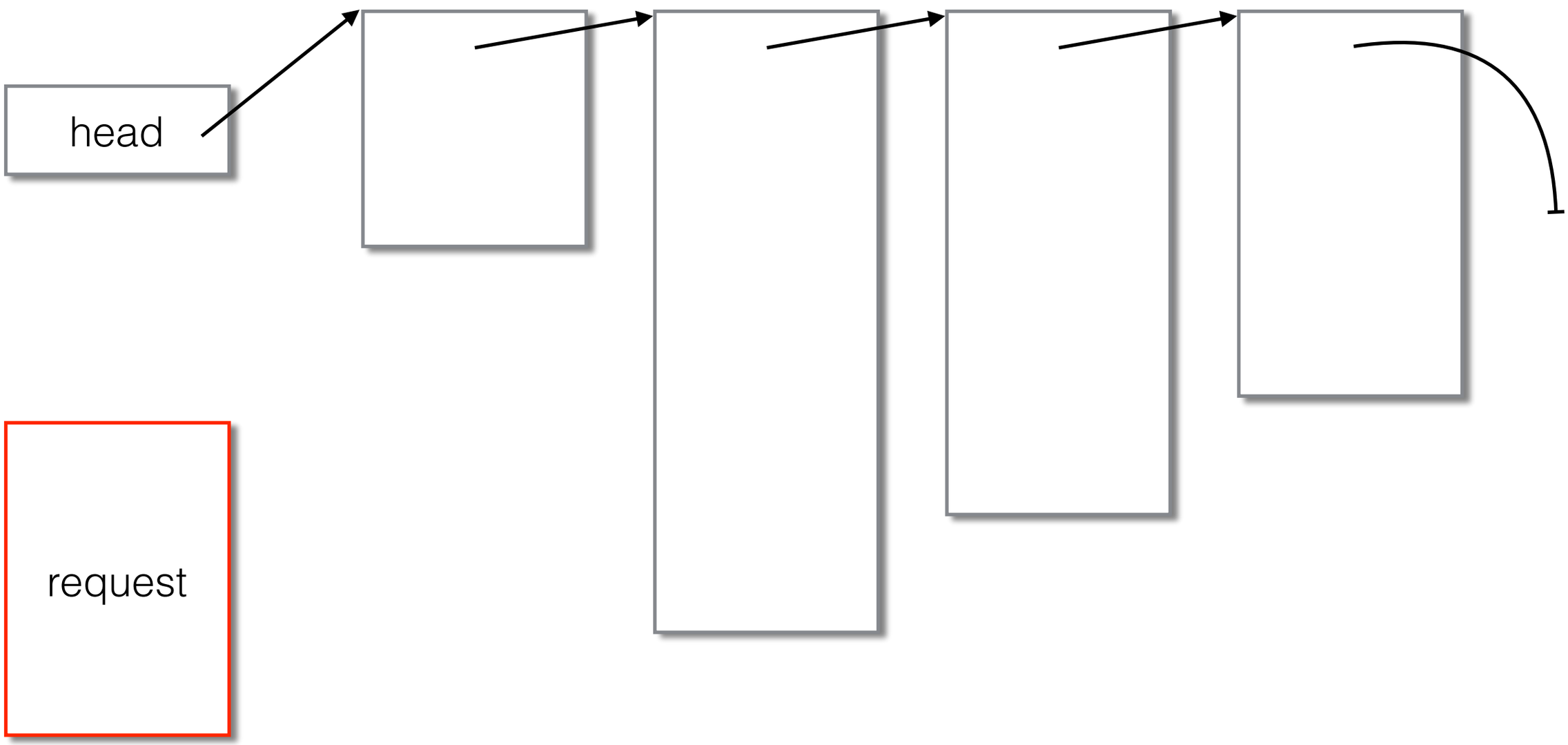
First fit



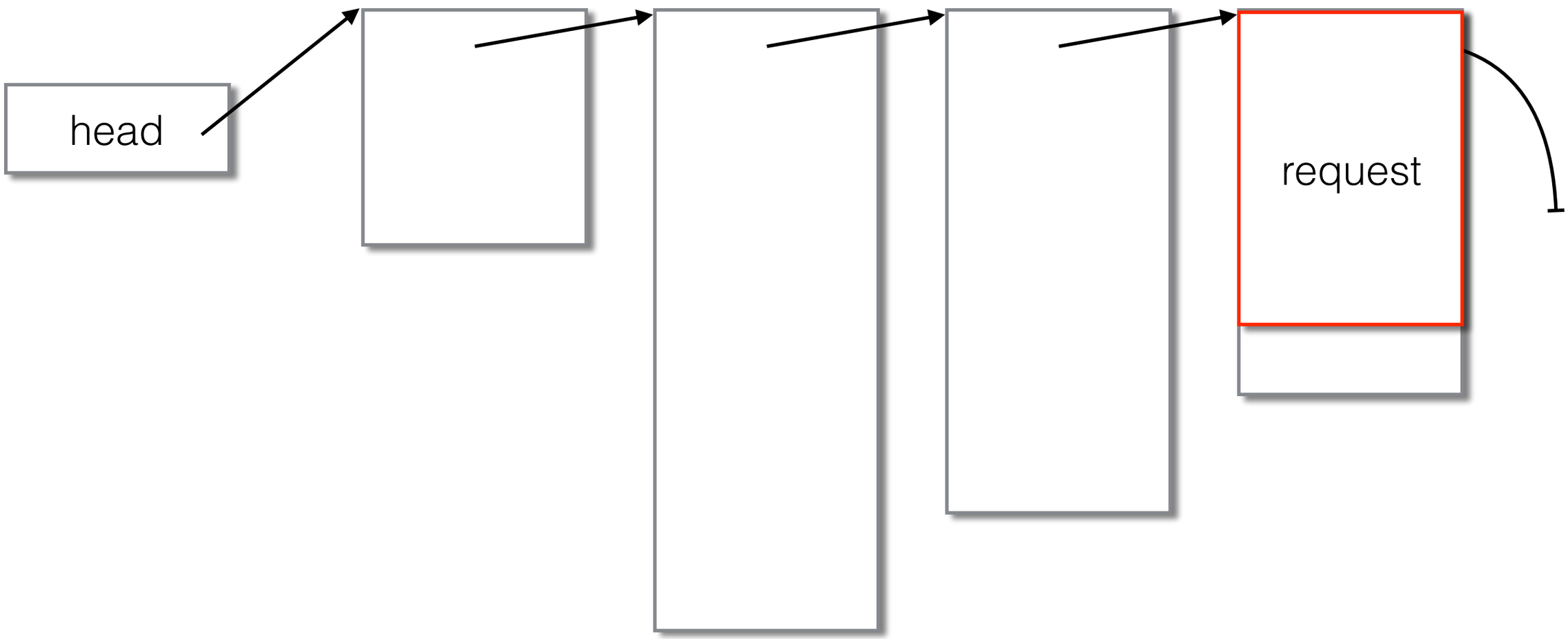
Best fit: the block's size larger than request

but has the smallest memory.

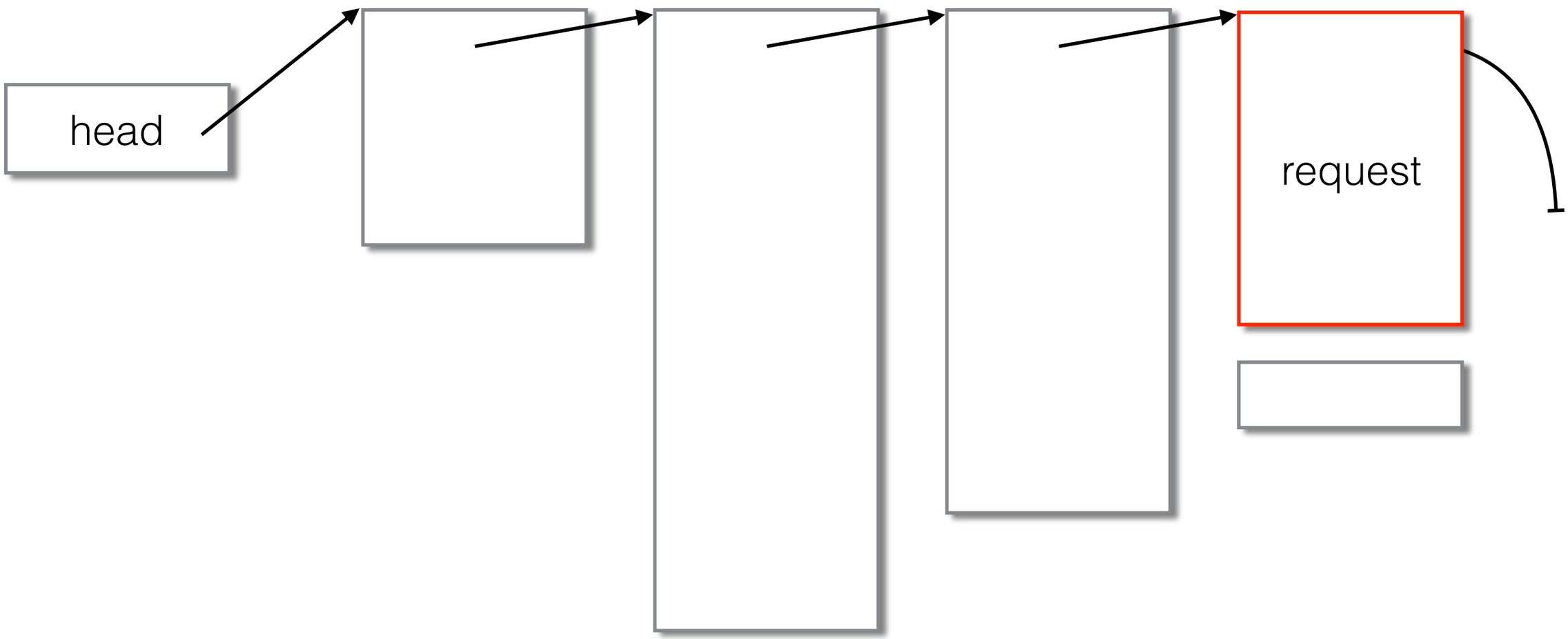
Best fit



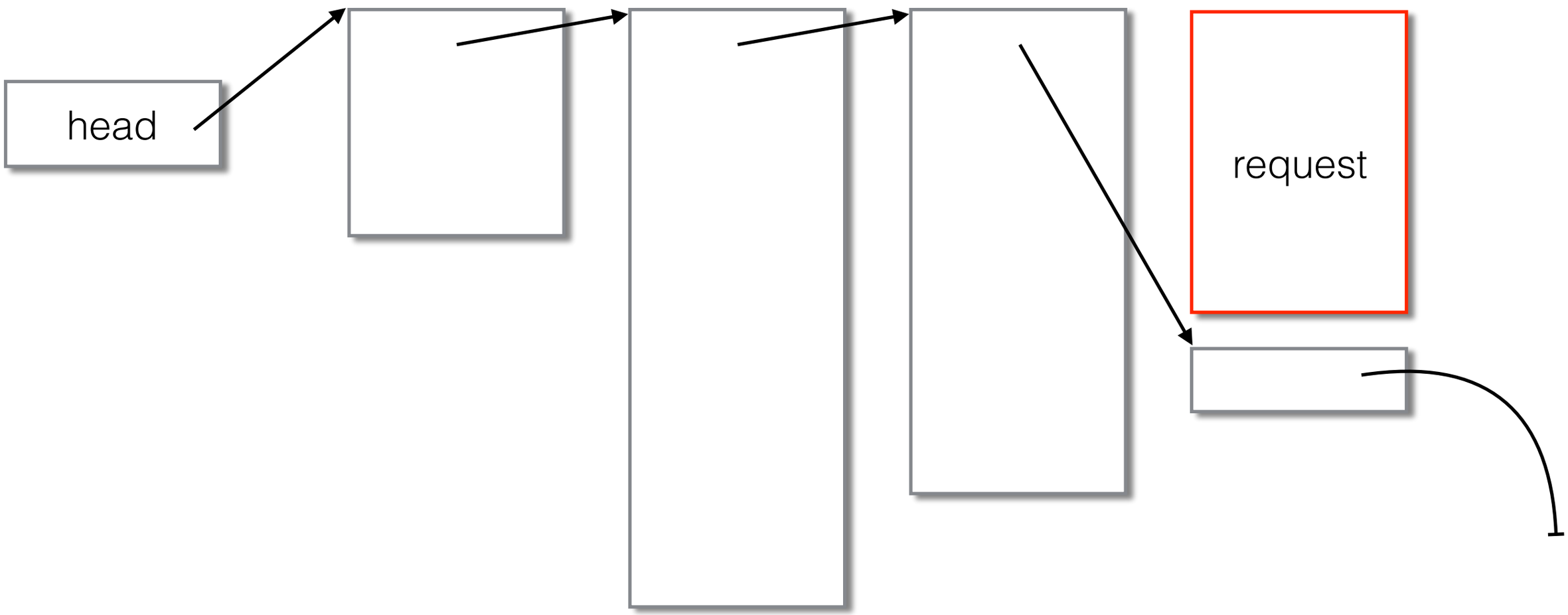
Best fit



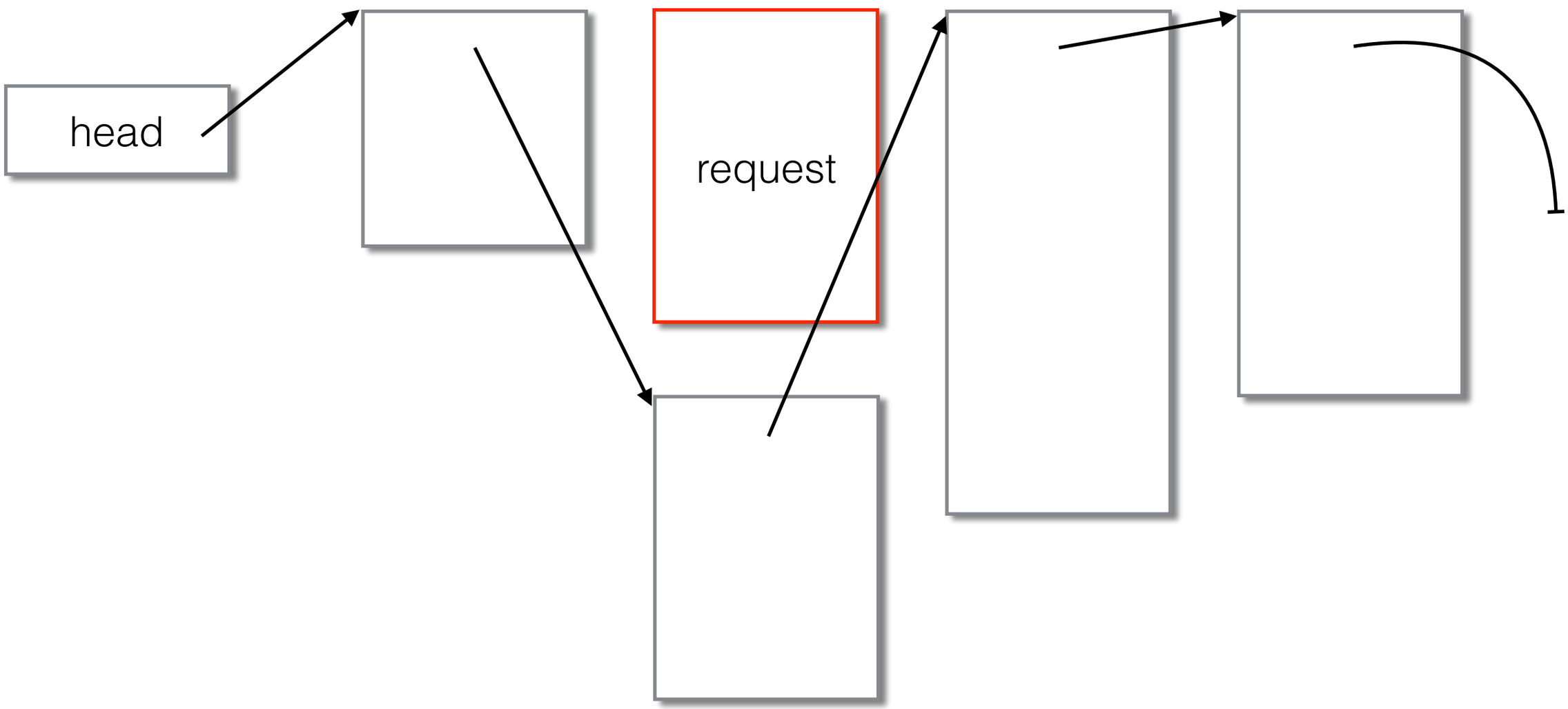
Best fit



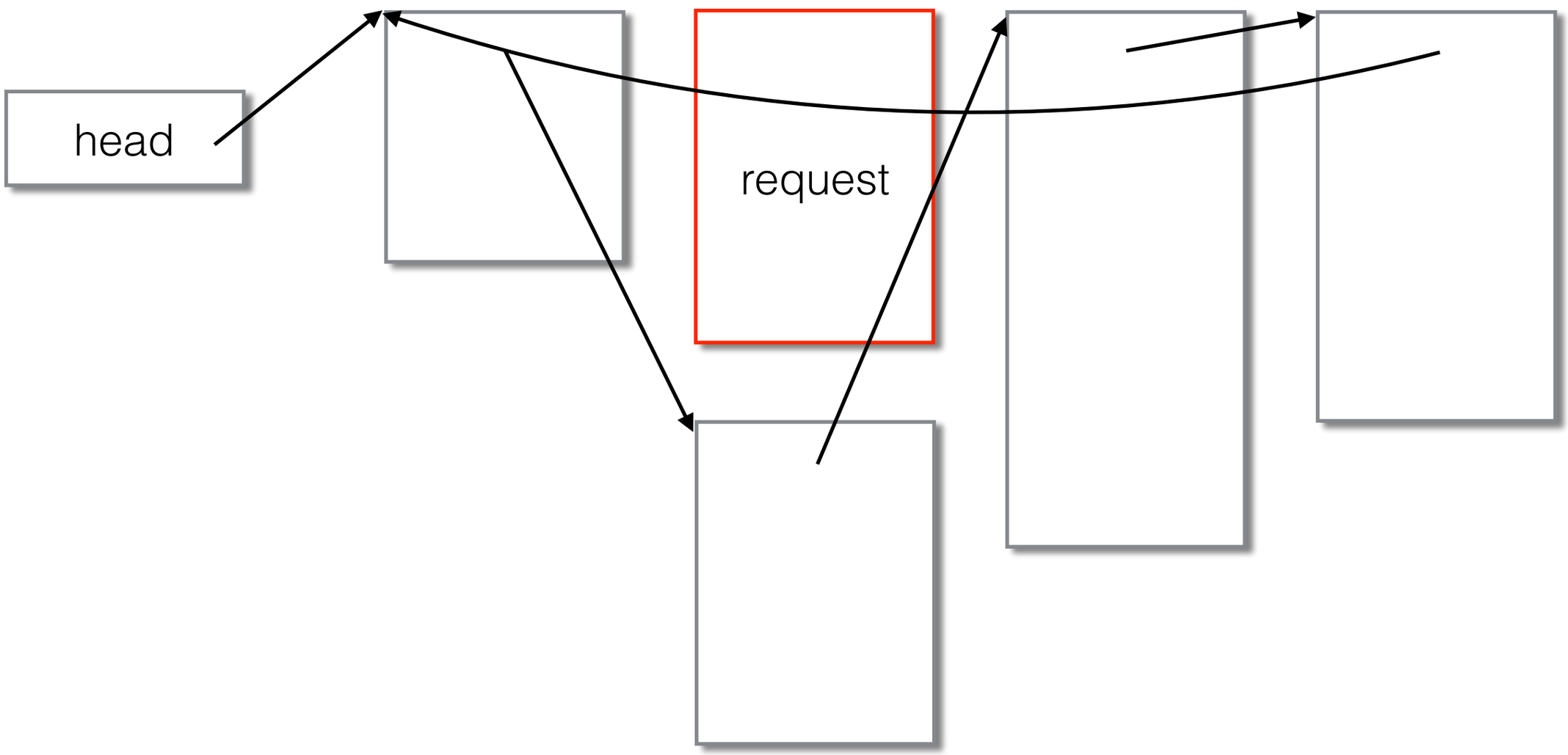
Best fit



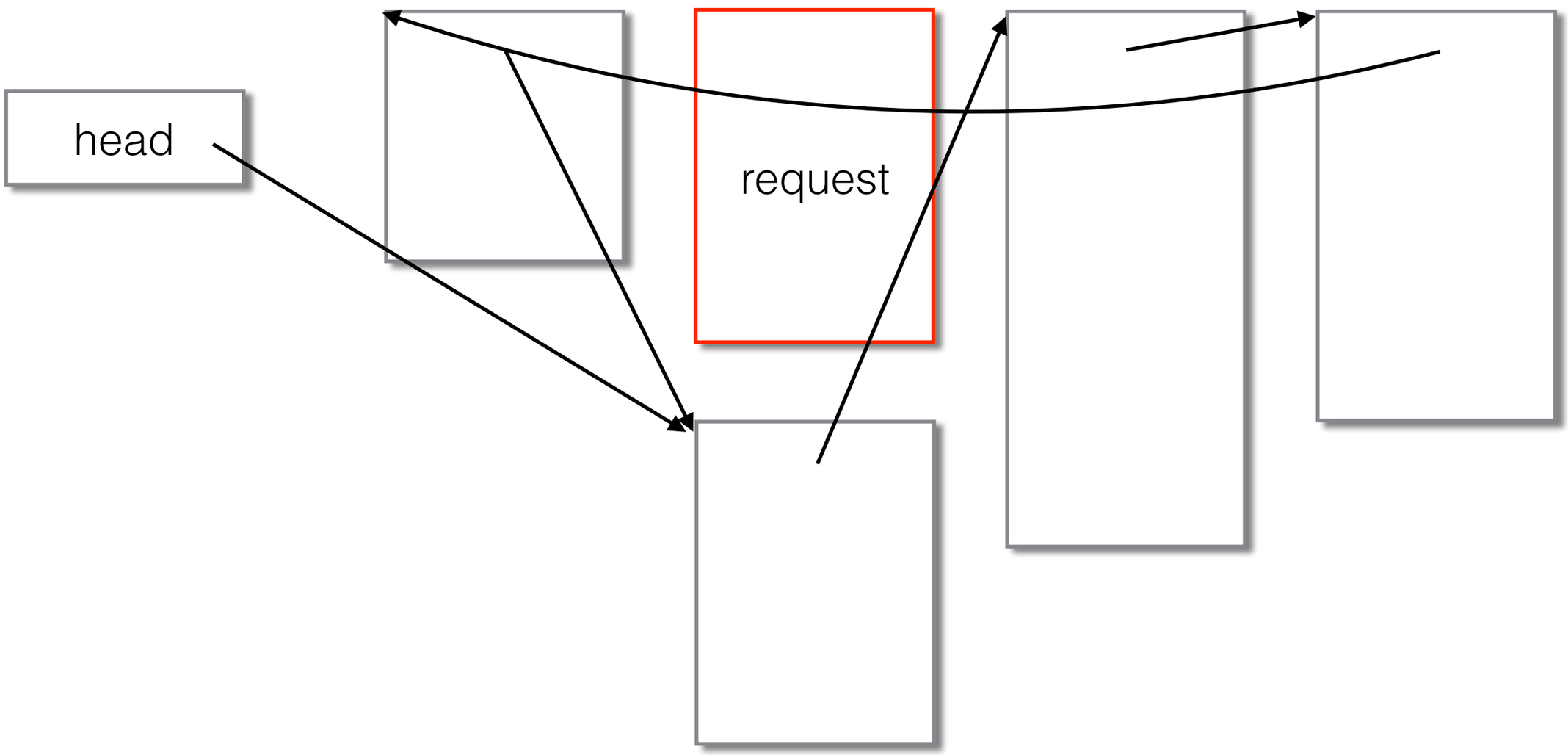
Next fit



Next fit



Next fit



Free list allocation is slow

- Needs a loop
  - And the number of iterations can be large
- Needs a size test
- Needs to be able to split a block

Object demographics

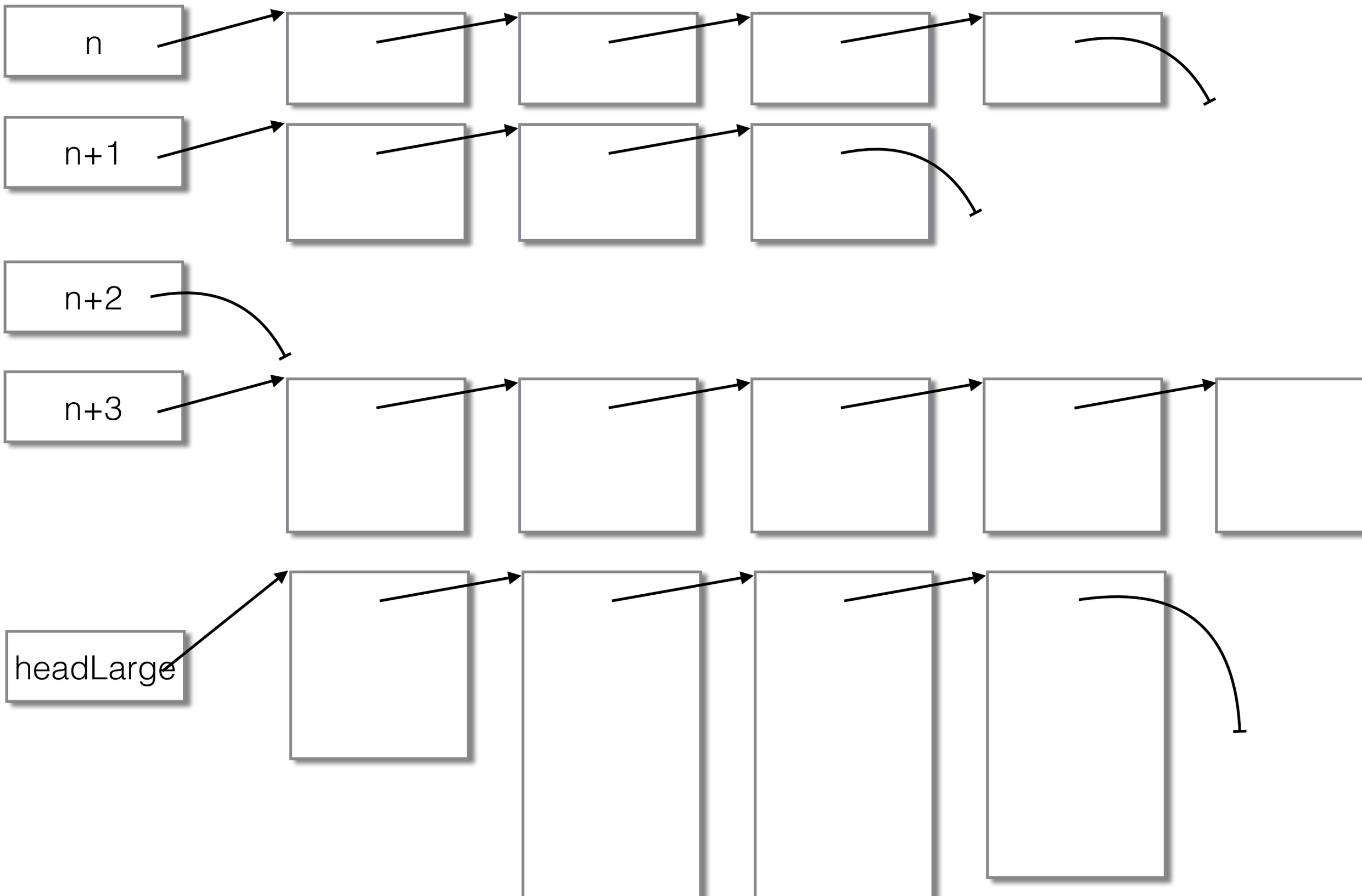
- What is the distribution of requested sizes?
- Obviously, it depends on the program. But: it is heavily influenced by the language and the available libraries.

Examples

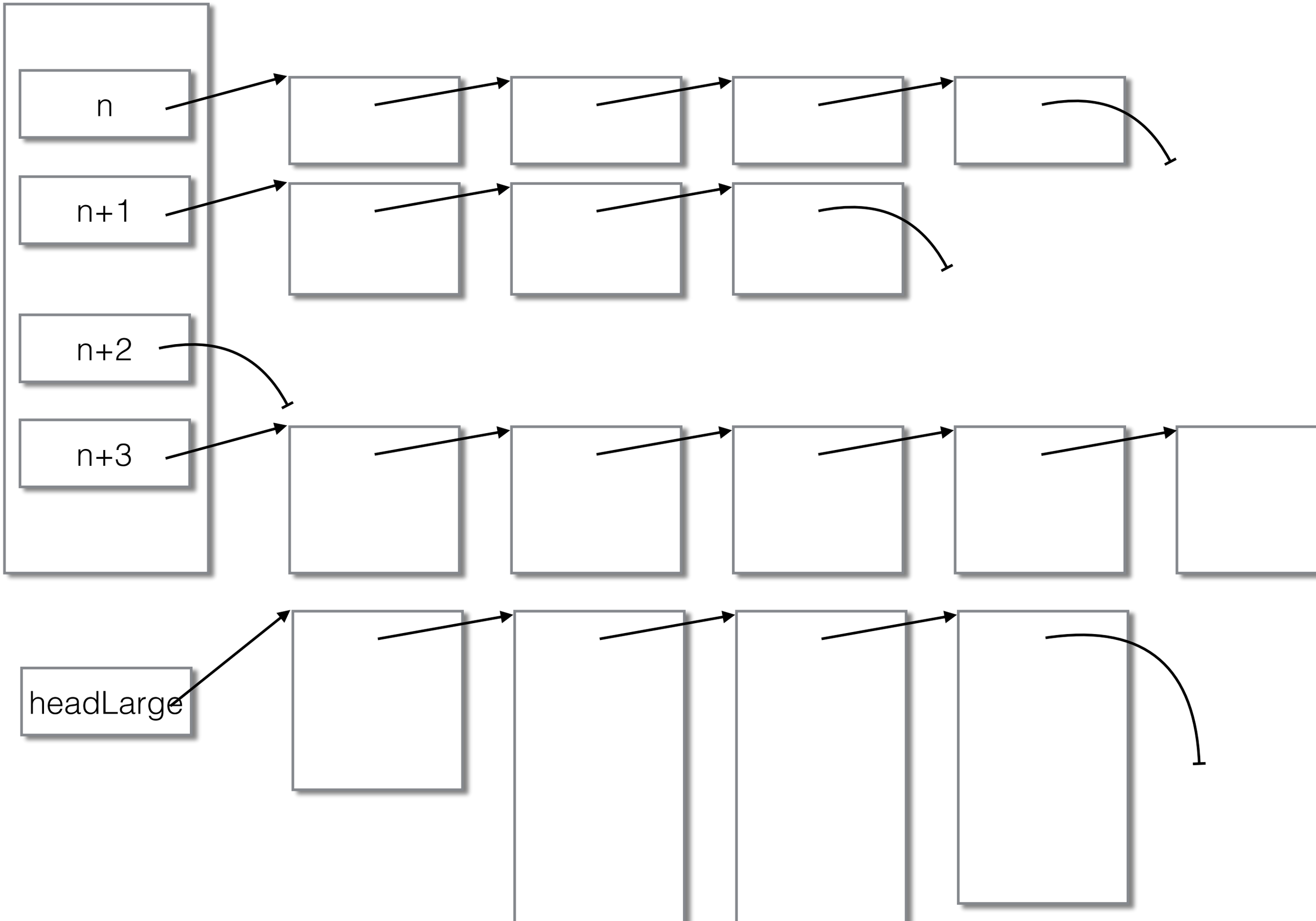
- Lisp programs heavily depend on 2-element CONS cells
- In object-oriented languages, most requests are for small objects (2-10 fields)
- Programmers tend to use named structures only up to certain cognitive limits; beyond that, indexed structures tend to be used.
- “Objects are small, arrays can be large”

Single-size free lists

- For small objects, we can have a free list per size
  - Eliminates the loop
  - Eliminates the size check
  - Eliminates splitting in the fast path
- Use a general free list only for large objects
- If the desired small free list is empty, take a large chunk and split it into small pieces.


n
n+1
n+2
n+3
headLarge

heads



Fragmentation

- As free memory gets chopped into smaller and smaller pieces, we say it is *fragmenting*.
- In the limit, we cannot satisfy a request because there are no free blocks of sufficient size, while free memory is plentiful.

Types of fragmentation

- *Internal* fragmentation is due to requests being rounded up in size, and hence some memory is wasted internally to an allocated block.
- *External* fragmentation is the process of free blocks getting smaller due to splitting.

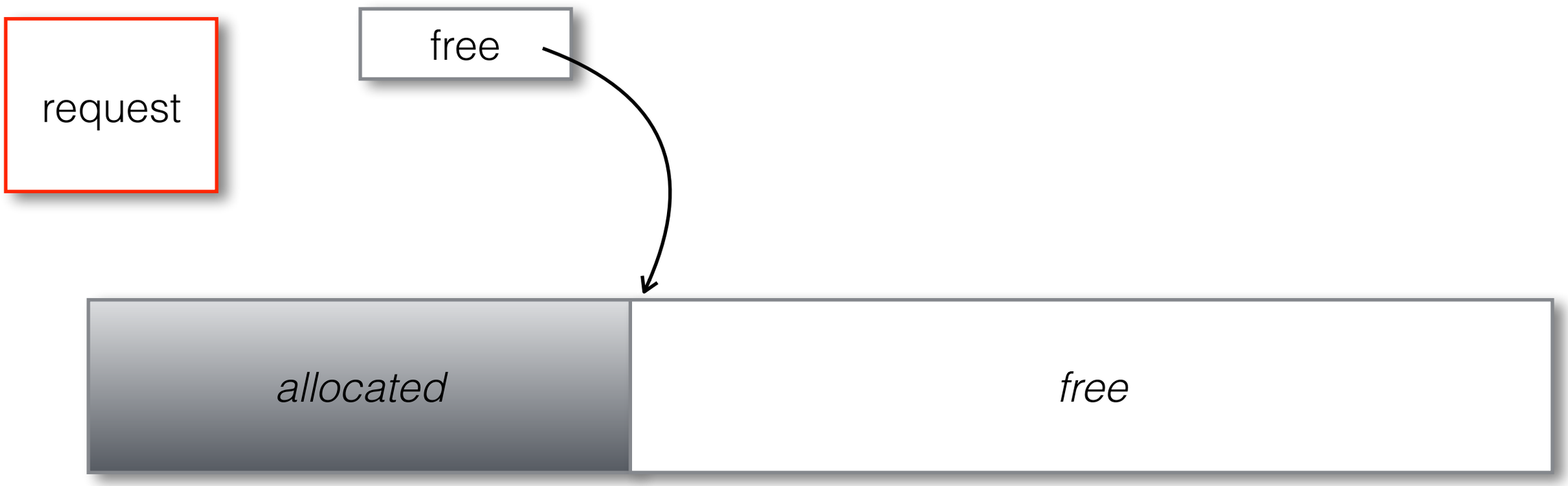
How fast does the allocator need to be?

- Objects are common, so allocation is frequent
- $O(n)$ Initialization costs are typically linear in object size
- Want to make small object creation as close to free as possible
  - Implies reclamation should be cheap too — see later

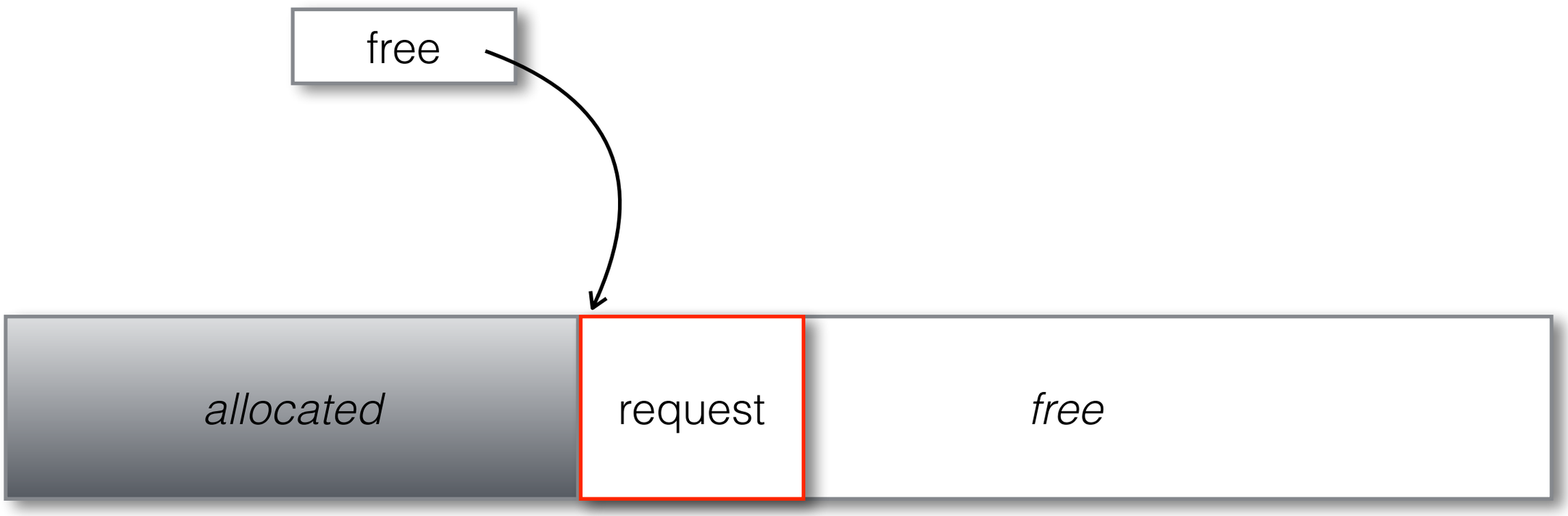
The Golden Goal

Programmer-specified abstractions should be as close to free as possible, in order to encourage well-factored, modular, reusable code.

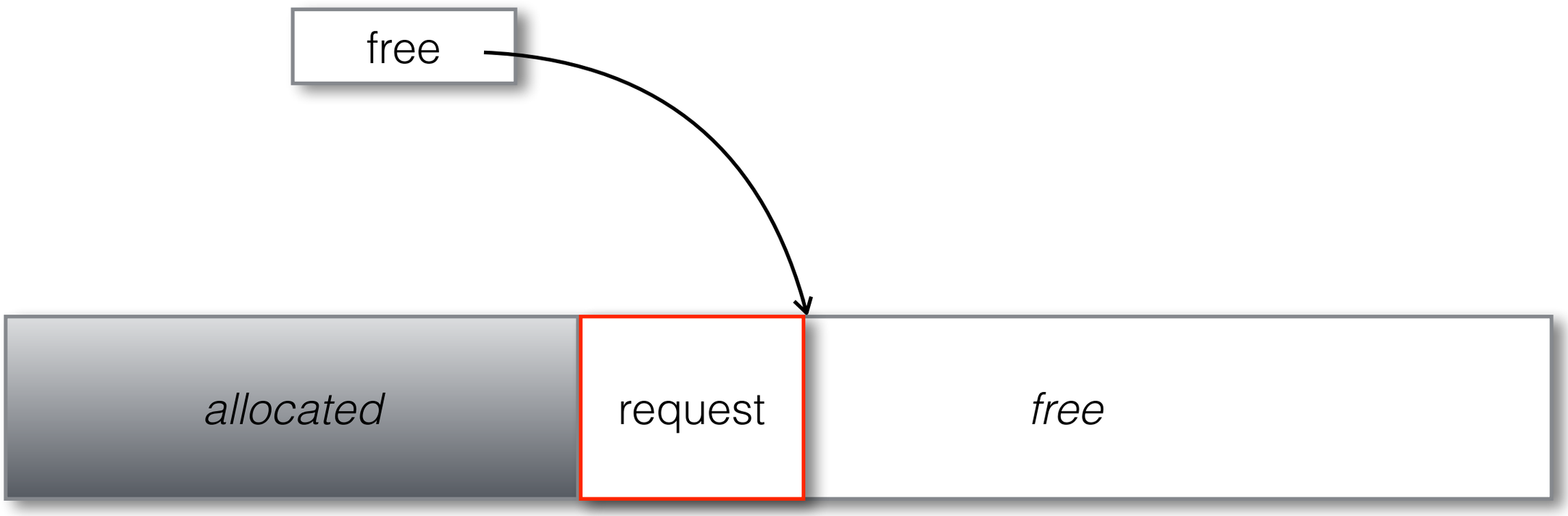
Bump allocation



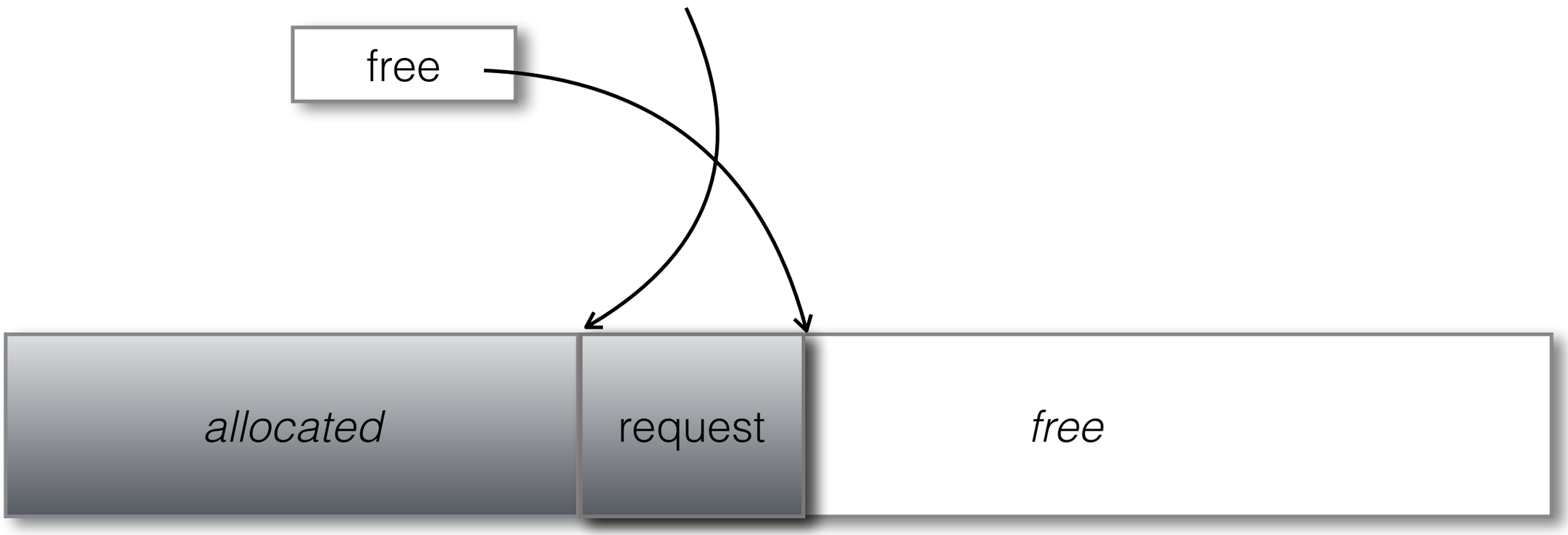
Bump allocation



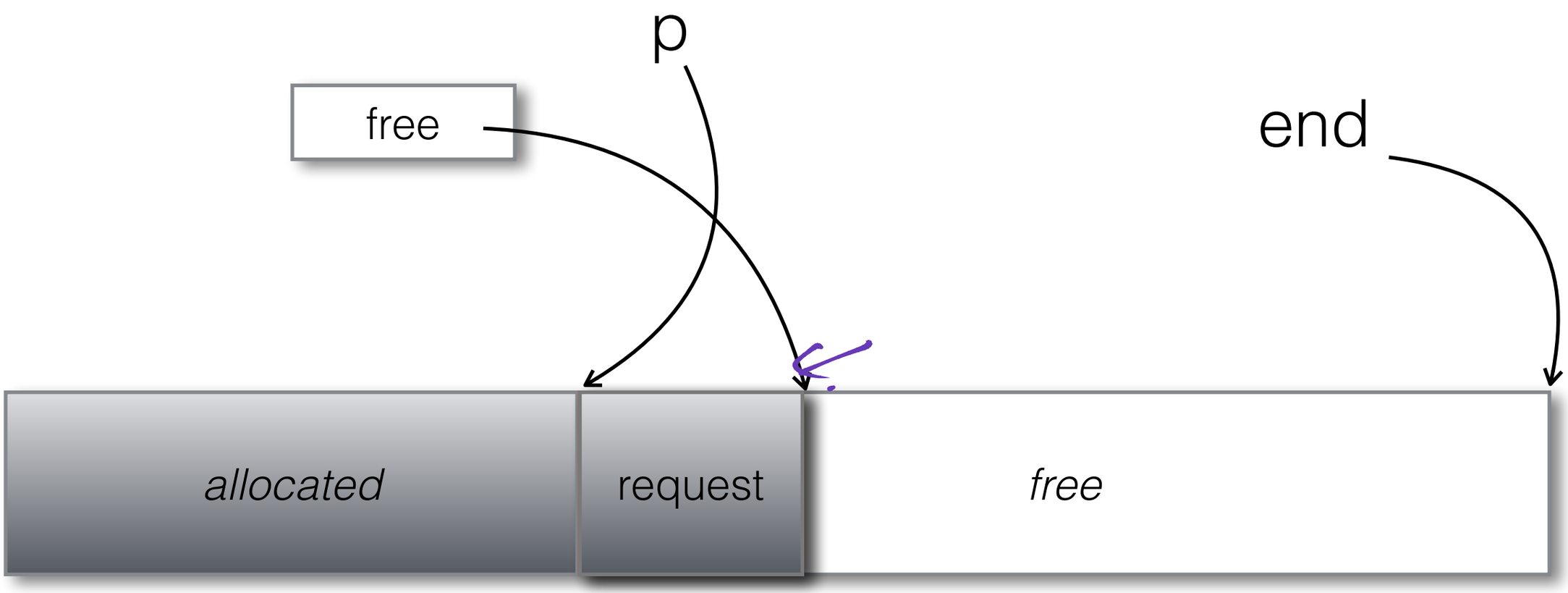
Bump allocation



Bump allocation



Bump allocation




```
newFree= free+size;
if (newFree <= end) {
  p= free; free= newFree; return p;}
```

Bump allocation — is it cheating?

- What happens when objects are freed — don't we just revert to a free list?
- The solution will be *compacting garbage collection* — but first we have to look at simpler forms of GC.

Deallocation

- If the language requires that the programmer deallocate memory, then we are done!
- But alas, most modern languages require the implementation to reclaim memory…

Why GC?

- Eliminates a class of unsafe errors — use of dangling pointers
- Saves the programmer from having to reason about when to reclaim memory
- Enables simple composition of code with sharing of objects
- However: does not prevent some kinds of memory leaks (more later)

Liveness

- When can an object be safely reclaimed?

```
public static void main(String args[]) {
  List<String> lines=
    Files.readAllLines(Paths.get(args[0]),
      Charset.defaultCharset());
  int nLines= lines.size();
  // reclaim lines??
  System.out.println(nLines);
}
```

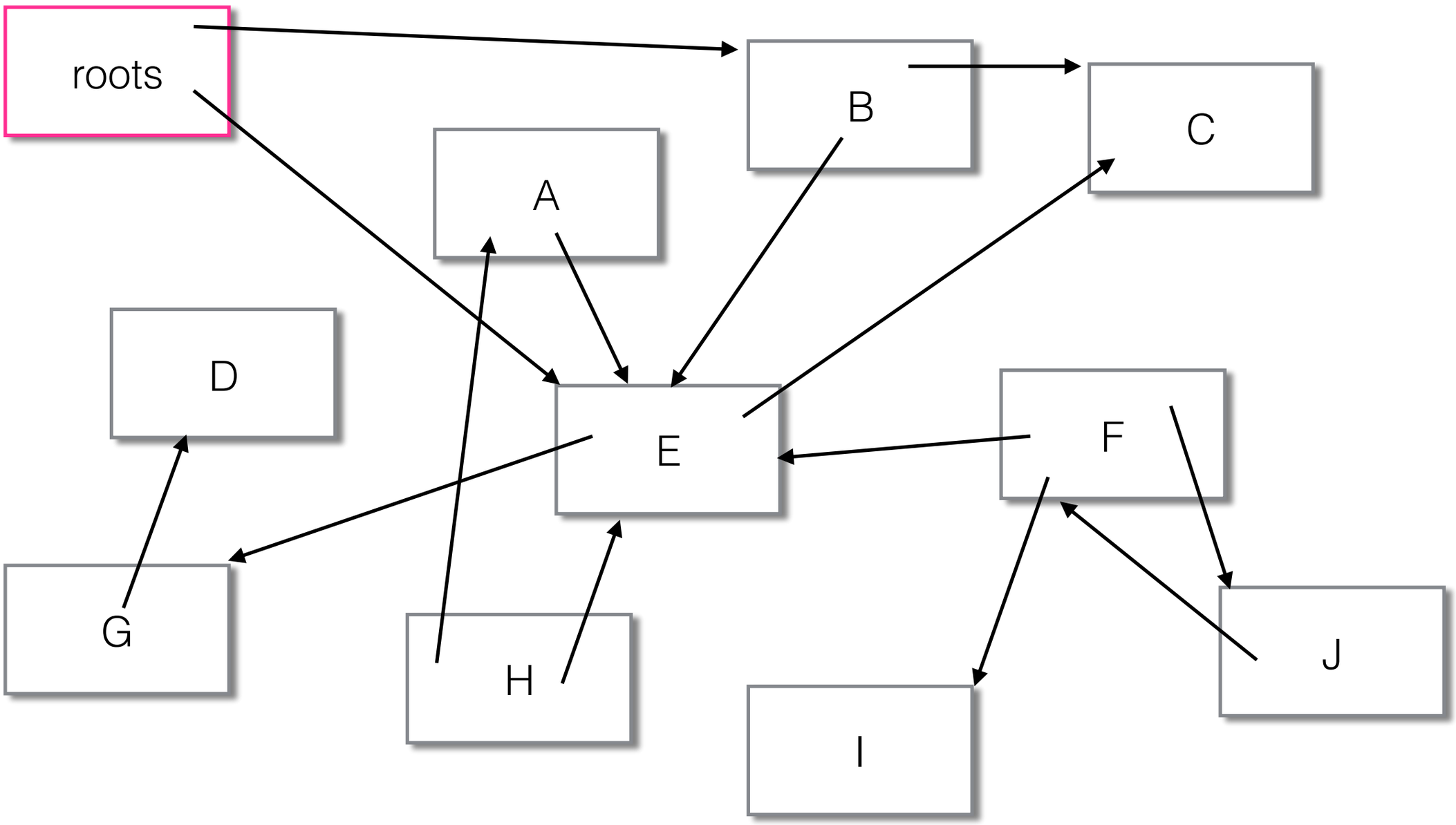
Liveness — in practice

- An ideal definition of liveness is that an object is dead when it is no longer needed.
  - But: VMs (and compilers) have severely limited crystal balls.

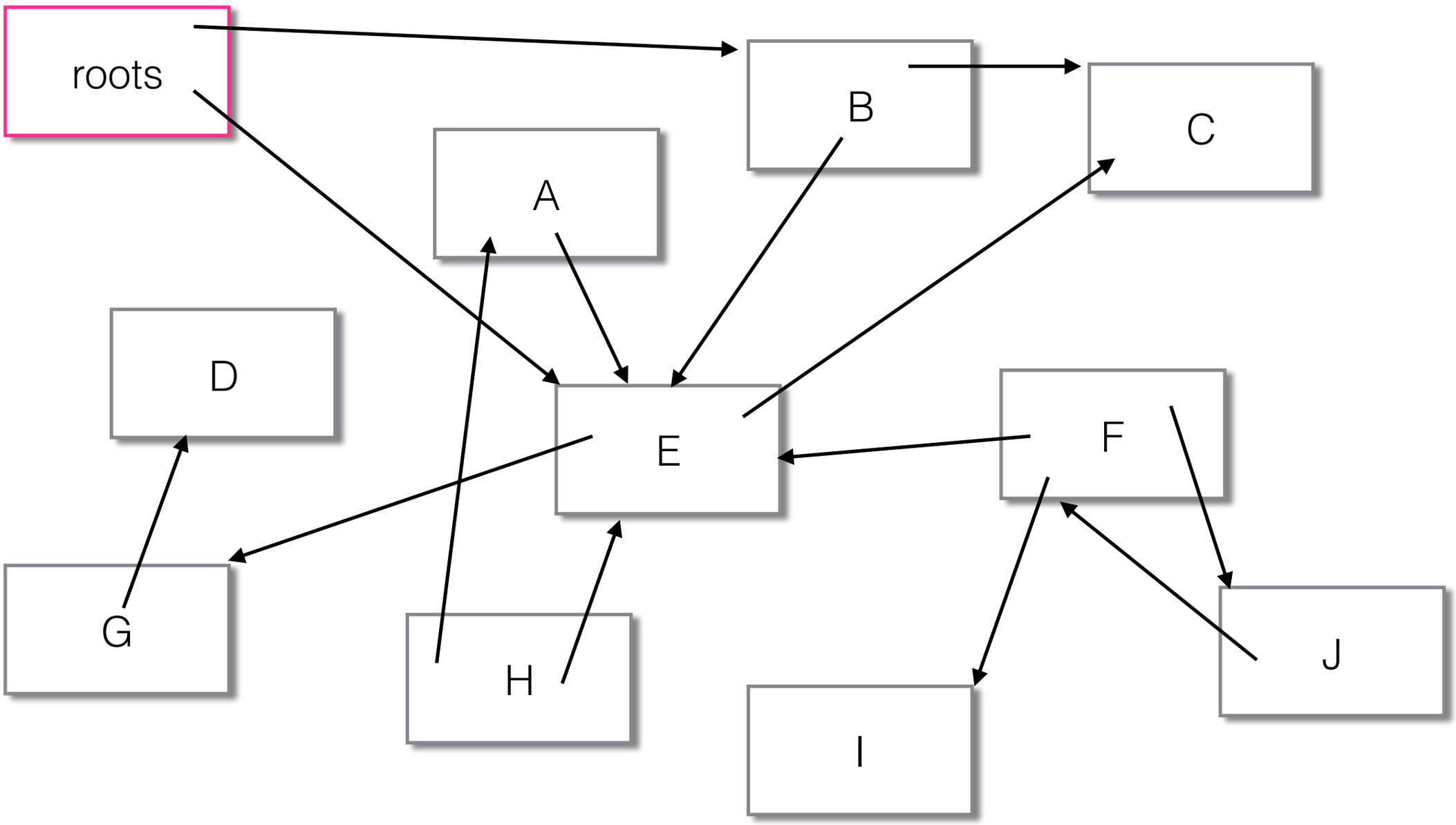
weak concepel definition.

- A more practical definition is that an object ceases to be live once it is no longer reachable from the system's *roots*.
  - The roots are typically all the variables in scope.
  - Requires a traversal of the stack(s), and enumeration of all globals

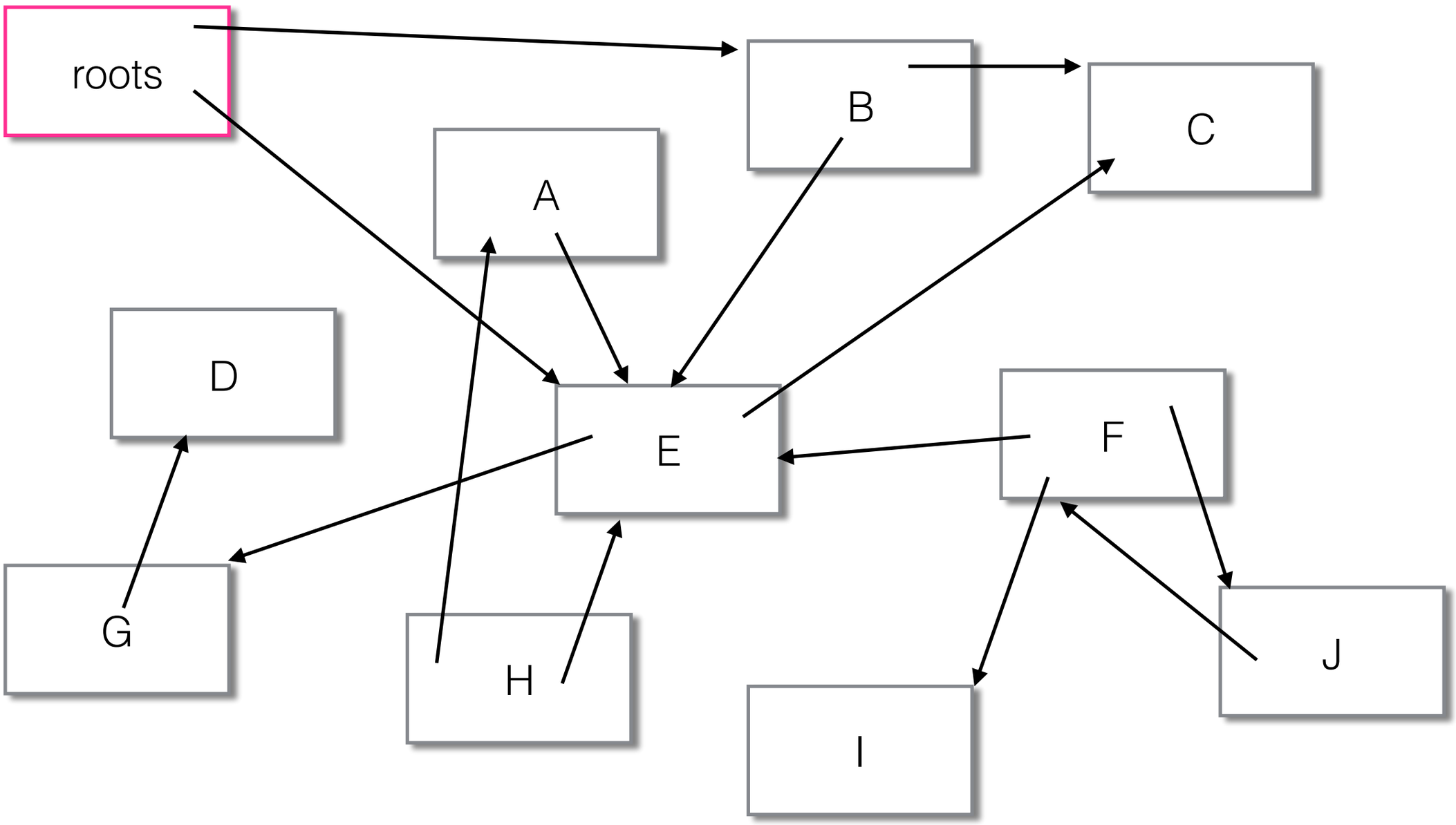
Reachability



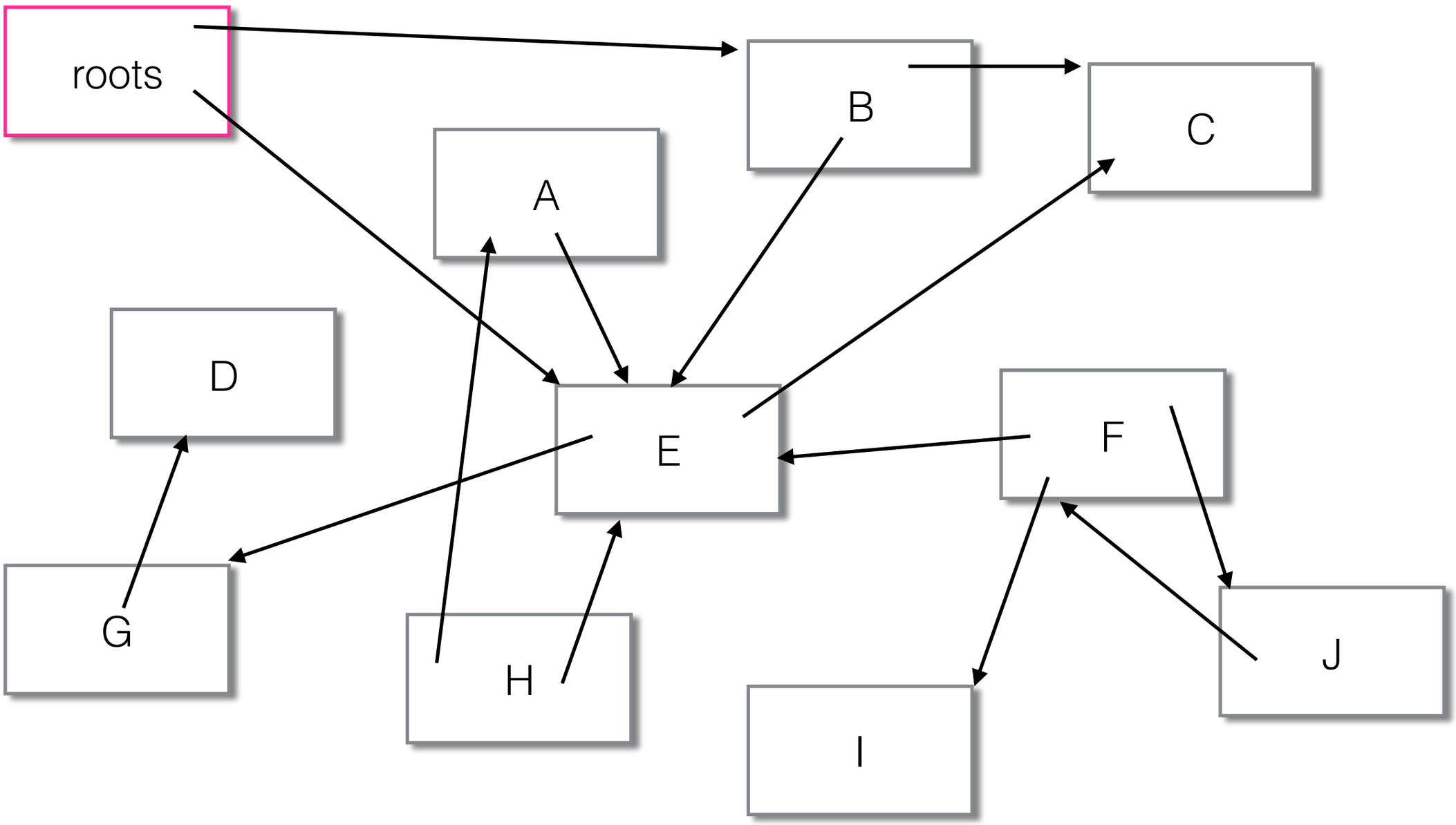
Reachability



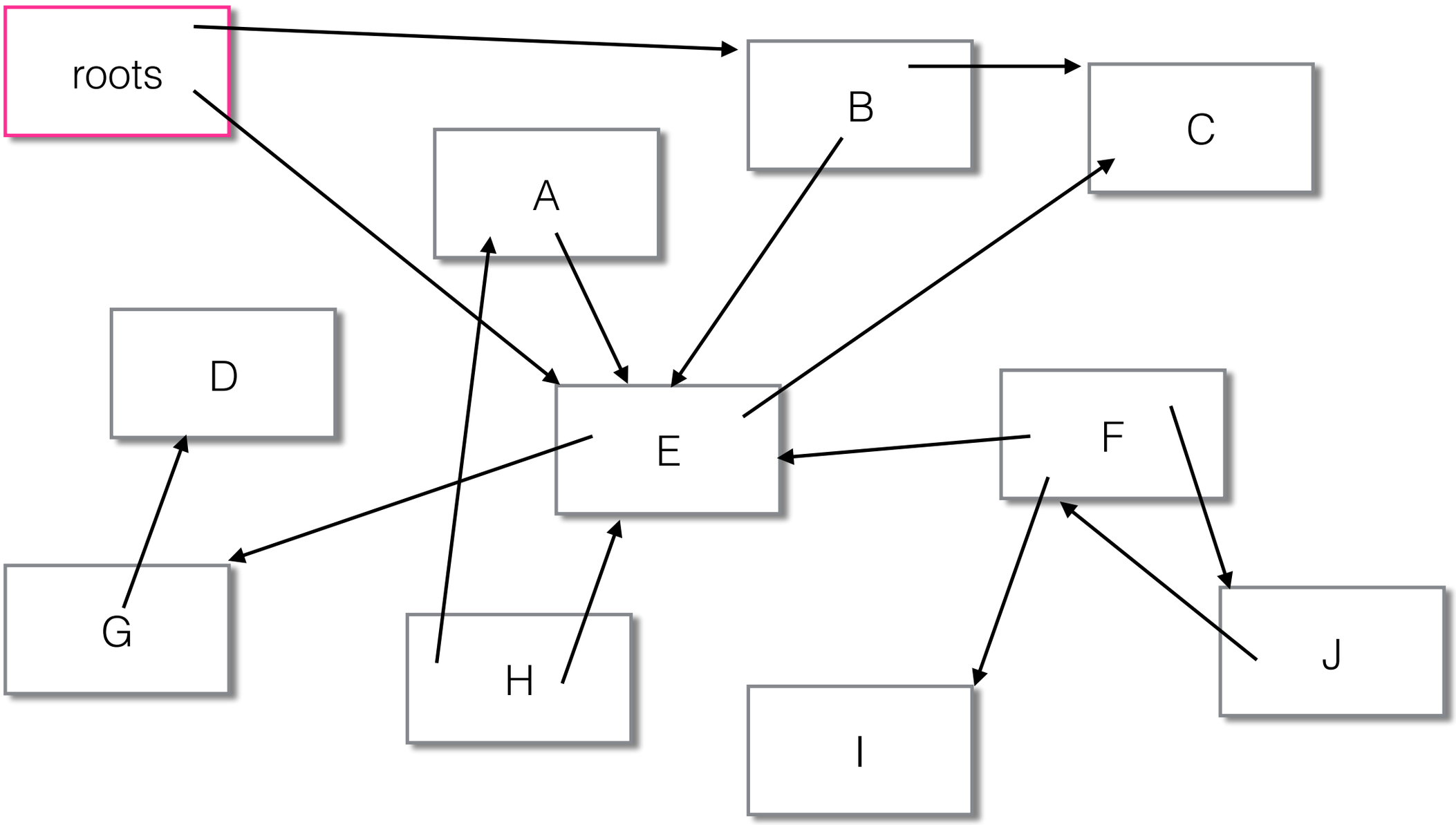
Reachability



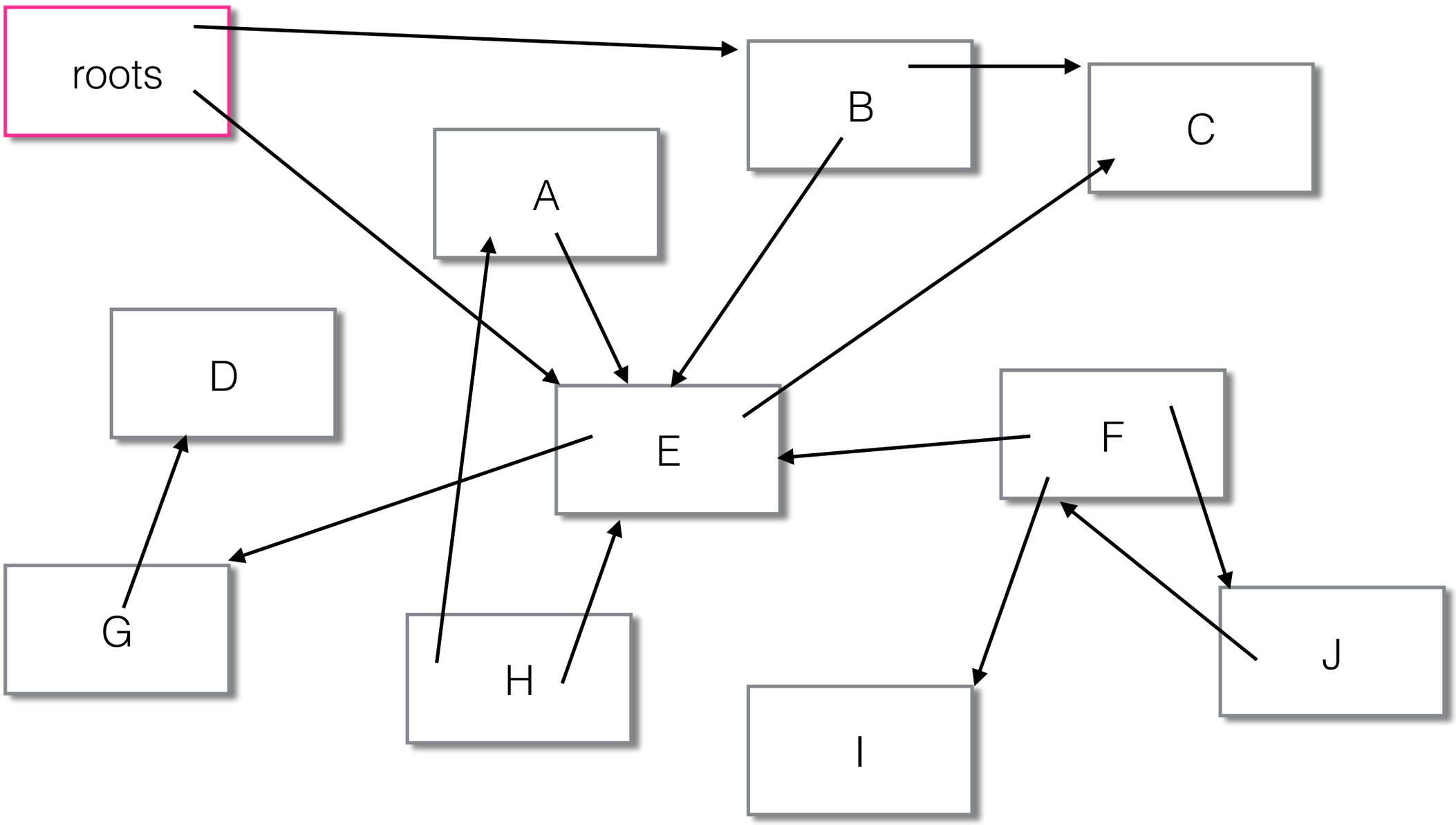
Reachability



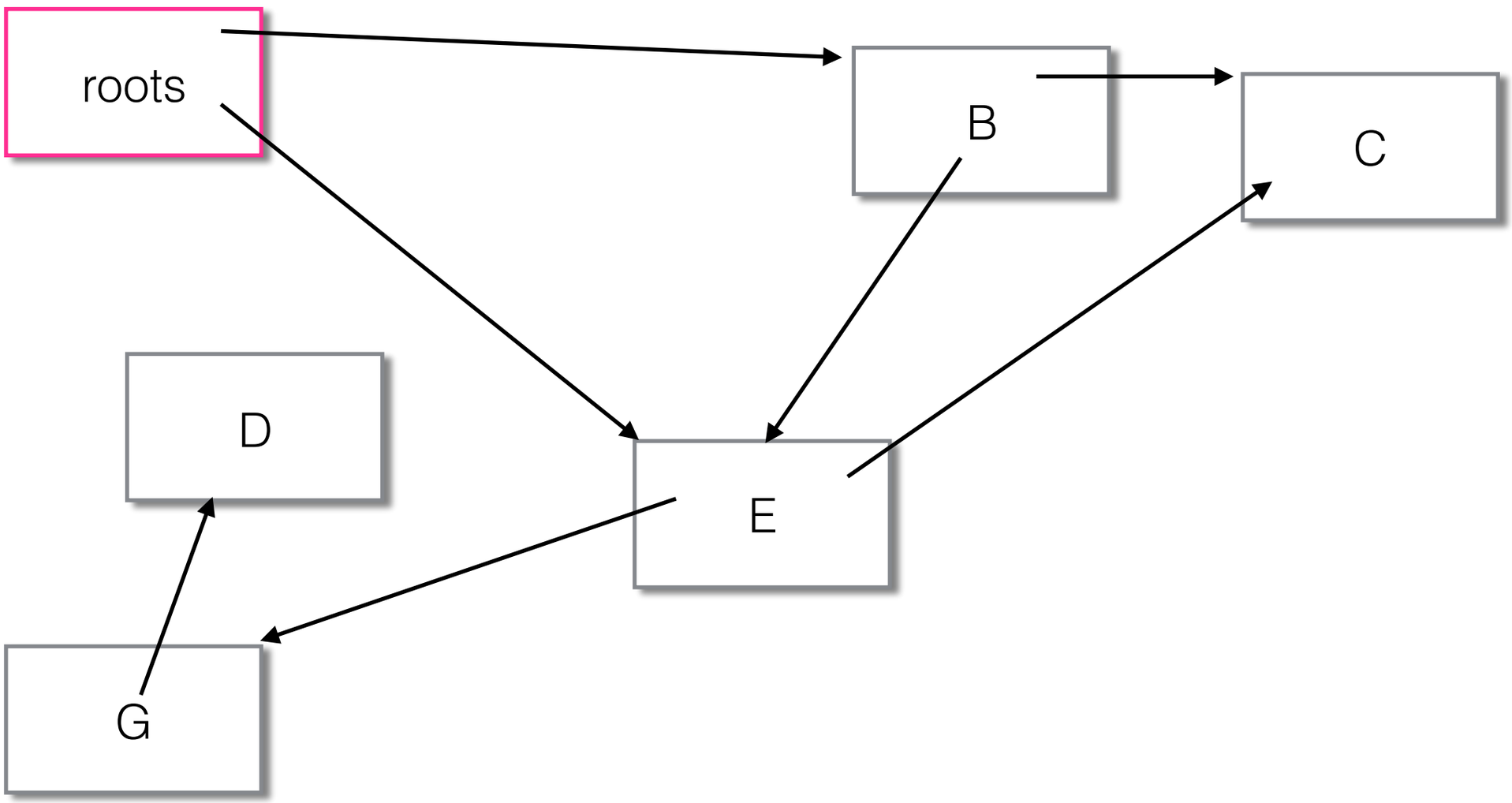
Reachability



Reachability



Reachability

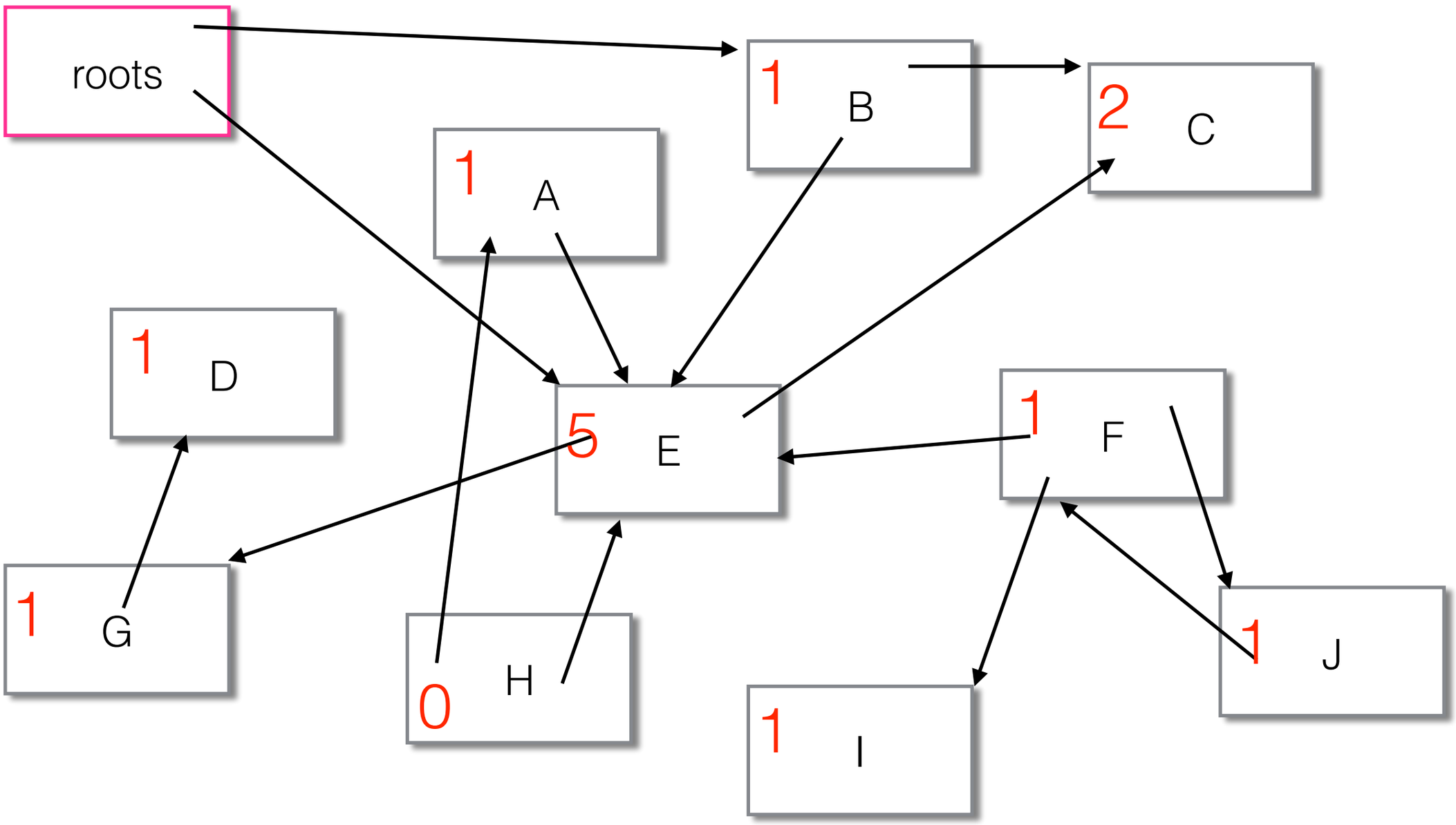


Memory management
Part (2)
Reference Counting

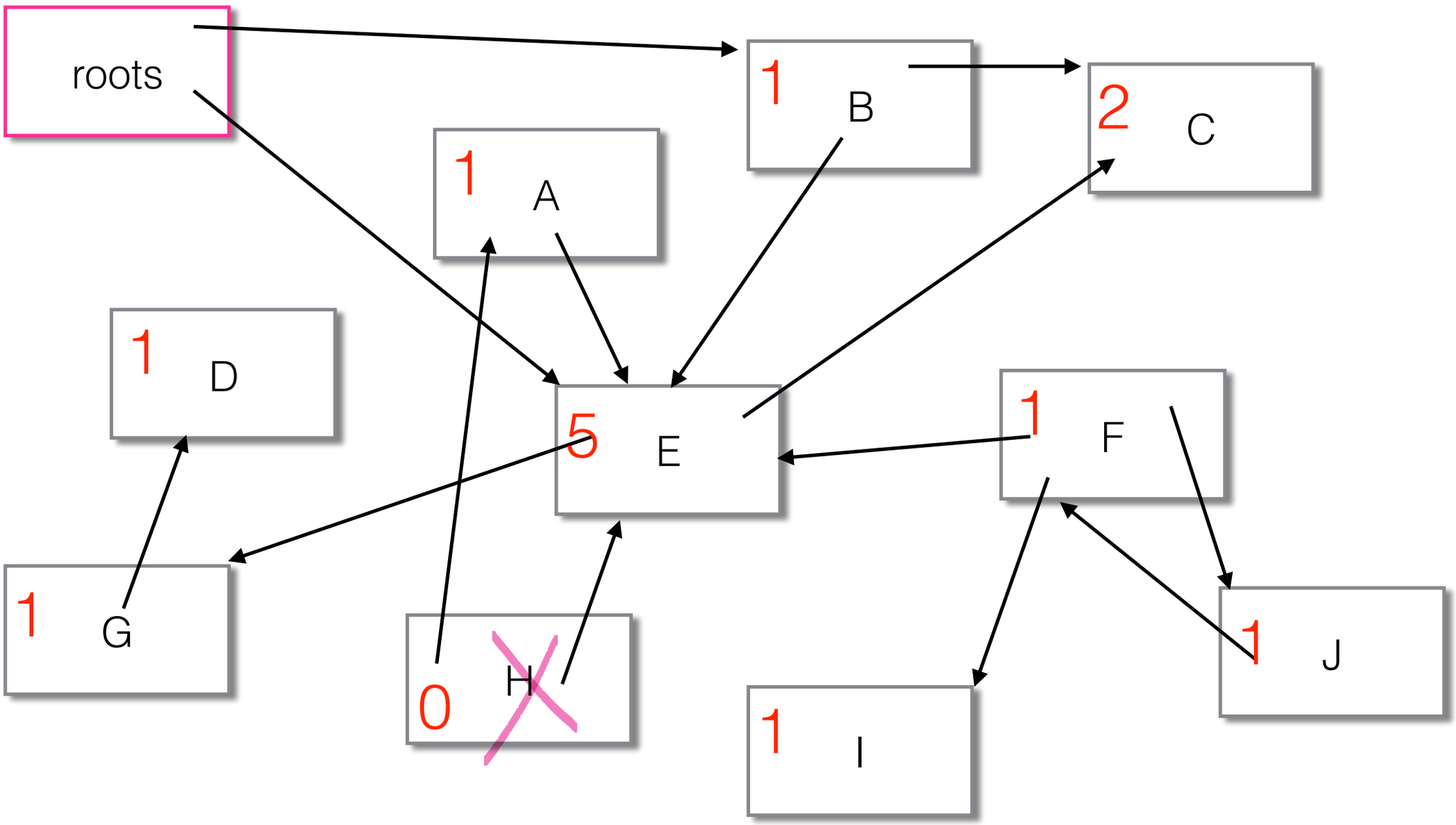
The Idea

- For each object, maintain a *reference count* — a count of how many times this object is referred to.
- ① When a reference is copied, increment the count.
- ② When a reference is destroyed, decrement the count.
- If a count is zero, the object can be reclaimed

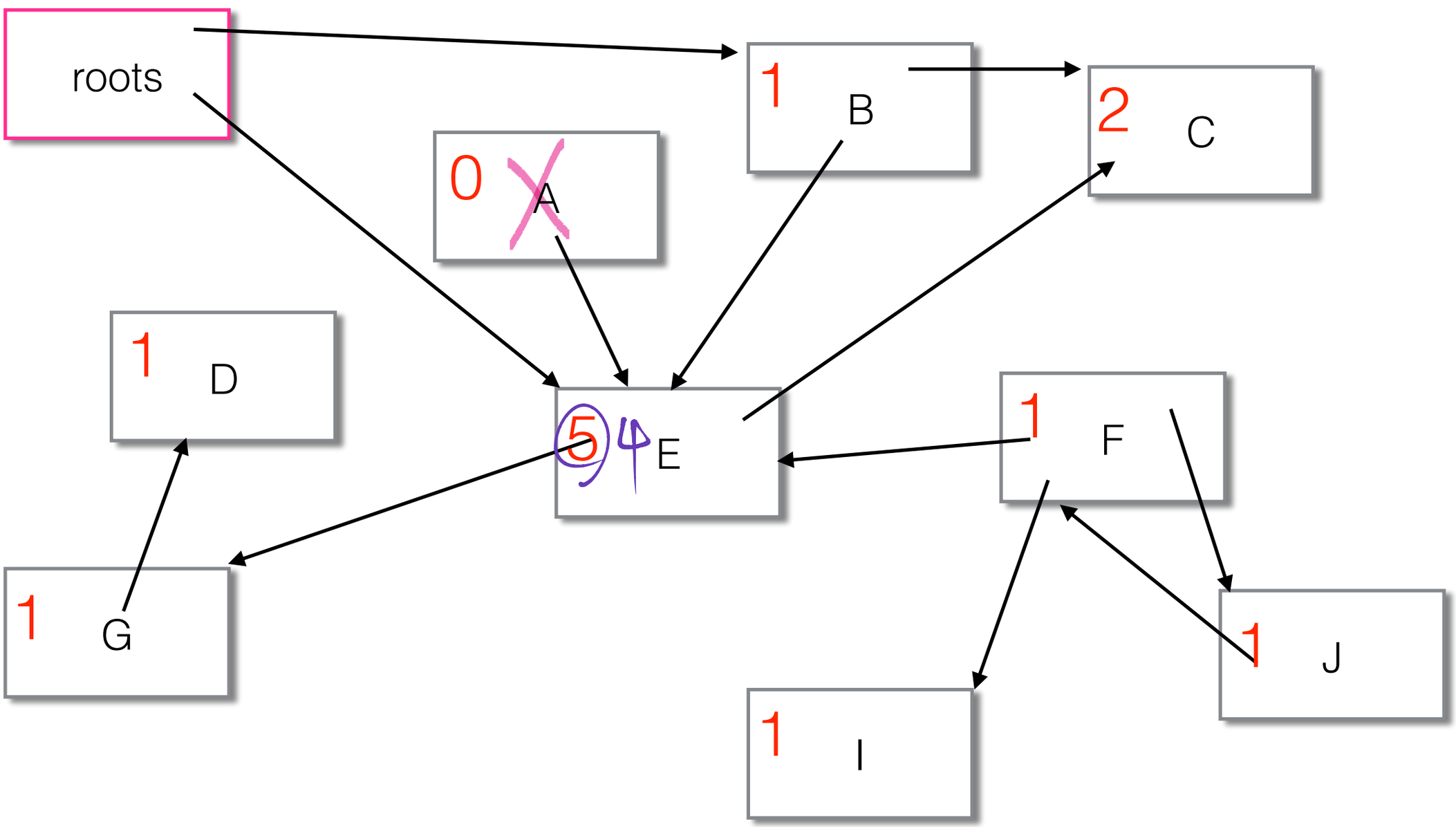
Reference counts



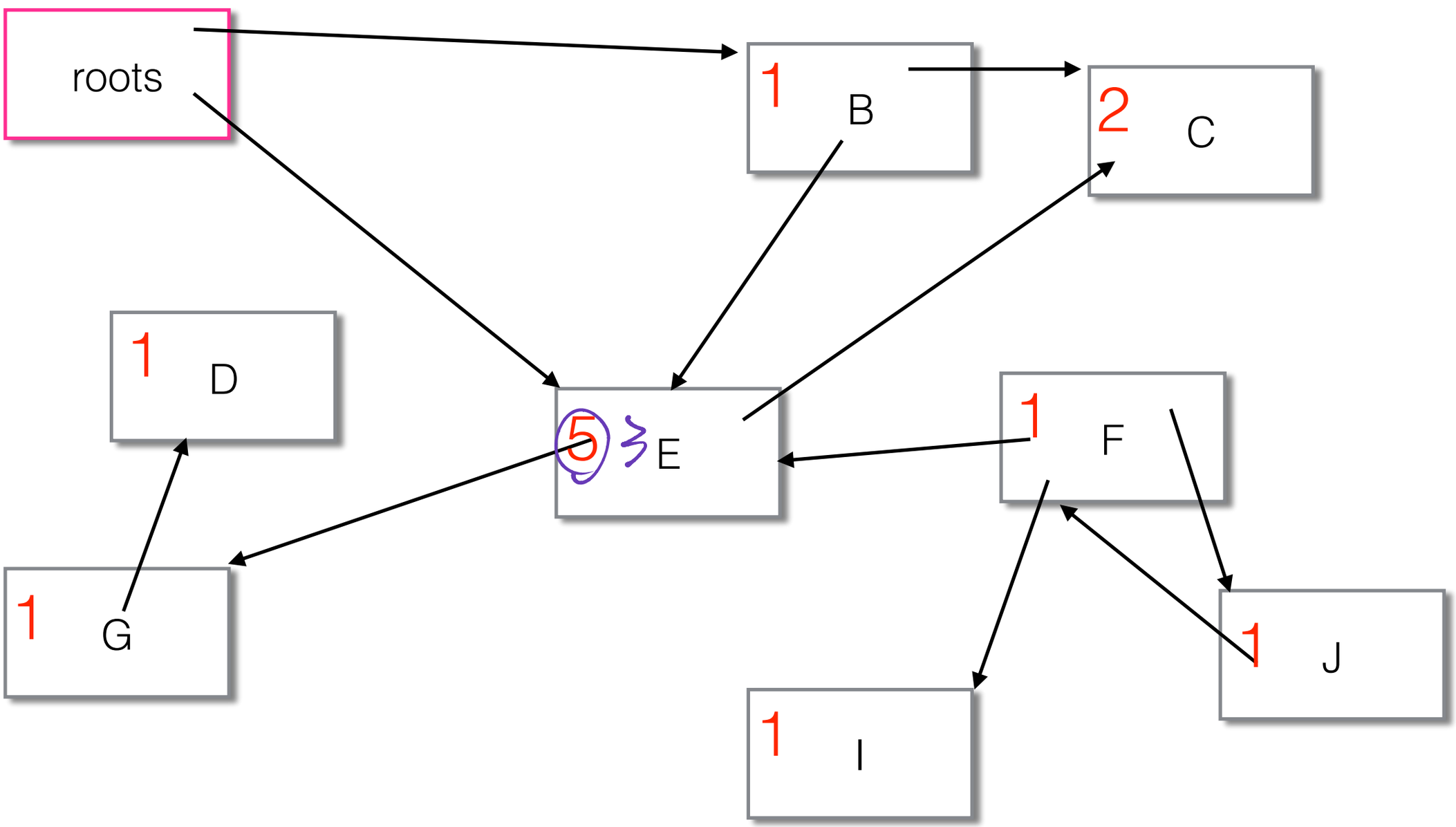
Reference counts



Reference counts



Reference counts



Reference counting implementation — take 1

Encompact obj with RC.

```
typedef struct { int RC; …fields…} *Obj;

void countUp(Obj p) { p->RC++; }

void countDown(Obj p) {
  if (--p->RC == 0) freeObj(p); }

void freeObj(Obj p){
  for (all references r in p)
    countDown(r);
  freeMem(p); }
```

Saturation

- It is usually wasteful to allocate a whole word for the count; most counts are small (often 1).
- However, once the count size is restricted, we then have to check for overflow, and when the count reaches the maximum value it saturates (or sticks) and never decreases.

Reference counting implementation — take 2

```
void countUp(Obj p) {
  if (p->RC != MAXRC) p->RC++;
}

void countDown(Obj p) {
  if (p->RC != MAXRC && --p->RC == 0)
    freeObj(p);
}
```

upper bound

Shaving off a test

- An 8-bit count is commonly used to exploit byte loads and stores. However, if we are willing to give up a bit of range, we can speed up the sequence by moving the saturation bit to the LSB

```
void countUp(Obj p) {
  unsigned char rc= p->RC + 2;
  p->RC= rc==0 ? 1 : rc;
}
```

No need to compare with MAXRC, but only need to check the value with 0.

```
void countDown(Obj p) {
  if ((p->RC -= 2) == 0)
    freeObj(p);
}
```

Reference counting in use

Recall the earlier interpretation of bytecodes:

```
case o(GET)  : push(vars[L5(*pc)]);  inc_pc(); continue;
case o(PUT)  : vars[L5(*pc)]= pop(); inc_pc(); continue;
```

Push and *pop* need to count, as does the variable assignment:

```
void push(Obj p) { countUp(p); *++sp= p; }
Obj  pop()       { Obj tos= *sp;
                   countDown(tos); sp--; return tos; }

  case o(PUT): { Obj *v= &vars[L5(*pc)]; Obj old= *v;
                 *v= *sp--; countDown(old); inc_pc(); continue; }
```

Some practicalities

- Reference counting has to be performed on all variables (stack, global and heap). Note that references in an activation record have to be decremented before deallocating the frame upon return.

- If tagging is used for ints, etc., then tag tests are also required.

- I have elided tests for NULL pointers. In some systems (e.g, Smalltalk) there are certain objects which must never be reclaimed. By assigning these objects to low-numbered addresses at startup, a single test can be used to prevent them (and NULL) from being counted. Setting MSB=1 for tagged primitives achieves the same.

- To avoid the call/return overhead, *countUp* and *countDown* are often defined as macros, or generated inline by a compiler.

More practicalities

- Decrementing the count of an over-written reference often incurs a cache miss.
- Incrementing the count of a copied reference causes a write (even if the fields are only being read)
- Concurrency: races aplenty!

Freeing when the count becomes zero

Freeing when the count becomes zero

```
void countDown(Obj p) {
  if (--p->RC == 0) freeObj(p); }

void freeObj(Obj p){
  for (all references r in p)
    countDown(r);
  freeMem(p); }
```

Freeing when the count becomes zero

```
void countDown(Obj p) {
  if (--p->RC == 0) freeObj(p); }

void freeObj(Obj p){
  for (all references r in p)
    countDown(r);
  freeMem(p); }
```

- Pause time when freeing a structure is limited only by heap size

Freeing when the count becomes zero

```
void countDown(Obj p) {
  if (--p->RC == 0) freeObj(p); }

void freeObj(Obj p){
  for (all references r in p)
    countDown(r);
  freeMem(p); }
```

- Pause time when freeing a structure is limited only by heap size
- Effectively unbounded recursion

Freeing when the count becomes zero

```
void countDown(Obj p) {
  if (--p->RC == 0) freeObj(p); }

void freeObj(Obj p){
  for (all references r in p)
    countDown(r);
  freeMem(p); }
```

- Pause time when freeing a structure is limited only by heap size
- Effectively unbounded recursion
- Solution: push zero-count objects onto a free queue, reclaimed incrementally

Freeing when the count becomes zero

```
void countDown(Obj p) {
  if (--p->RC == 0) freeObj(p); }

void freeObj(Obj p){
  for (all references r in p)
    countDown(r);
  freeMem(p); }
```

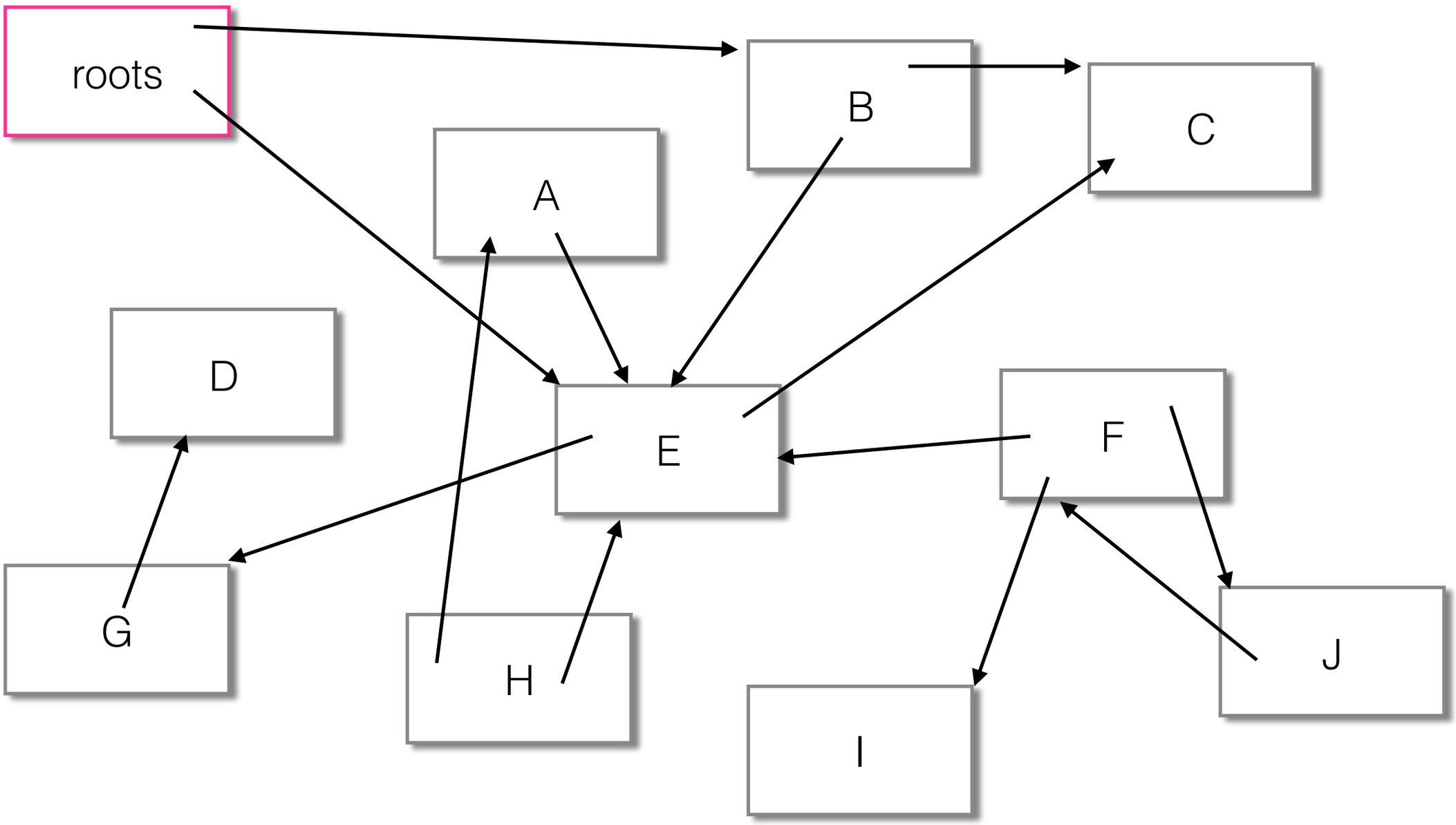
- Pause time when freeing a structure is limited only by heap size
- Effectively unbounded recursion
- Solution: push zero-count objects onto a free queue, reclaimed incrementally

Cycles

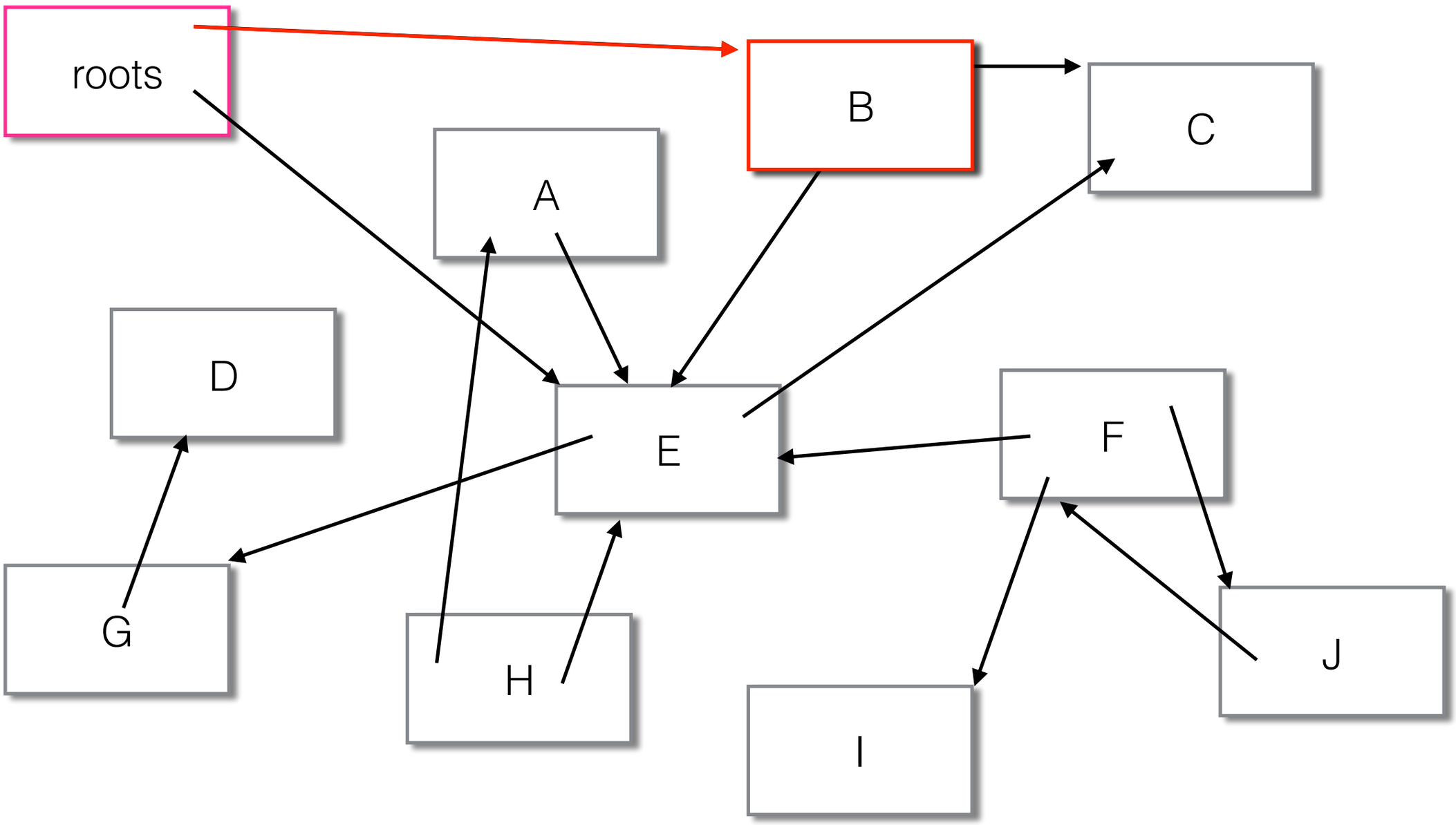
- The biggest drawback of reference counting is that it cannot reclaim cycles.
- This may not be a problem — in some systems, cycles are impossible.
- Even if possible, cycles may be infrequent, and may be reclaimed by a backup tracing collector.

Memory management
Part (3)
Tracing Collection

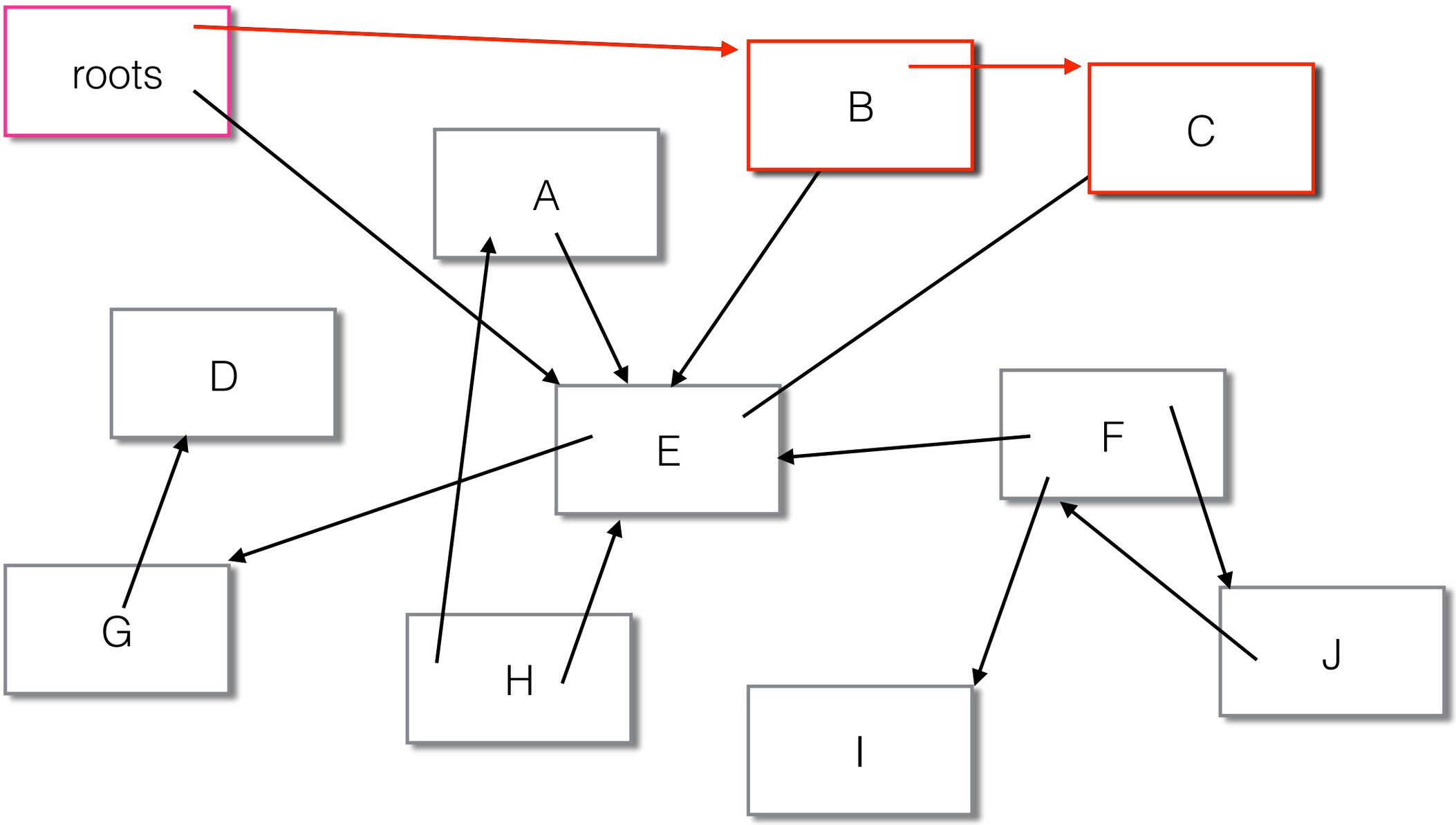
Marking



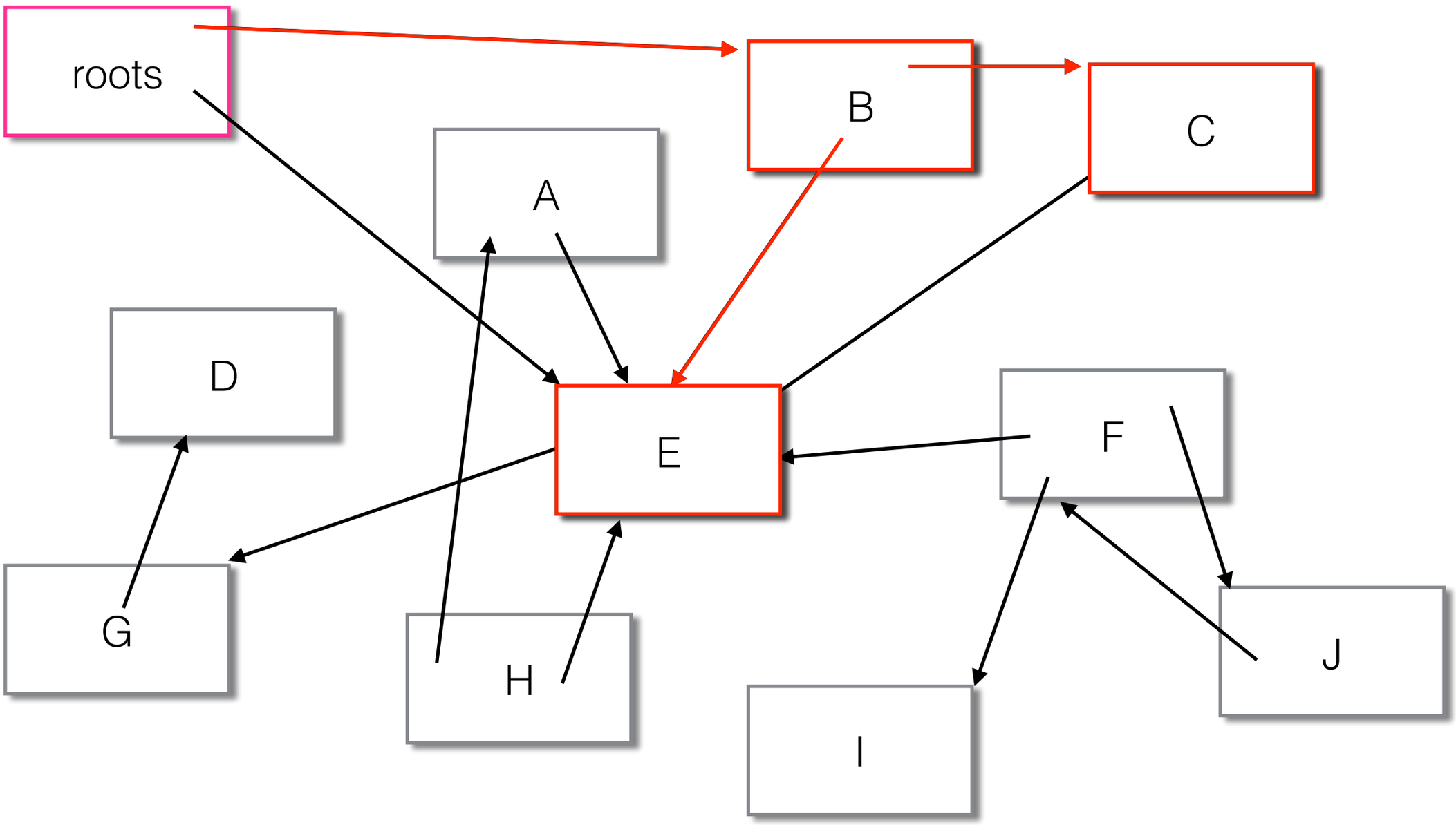
Marking



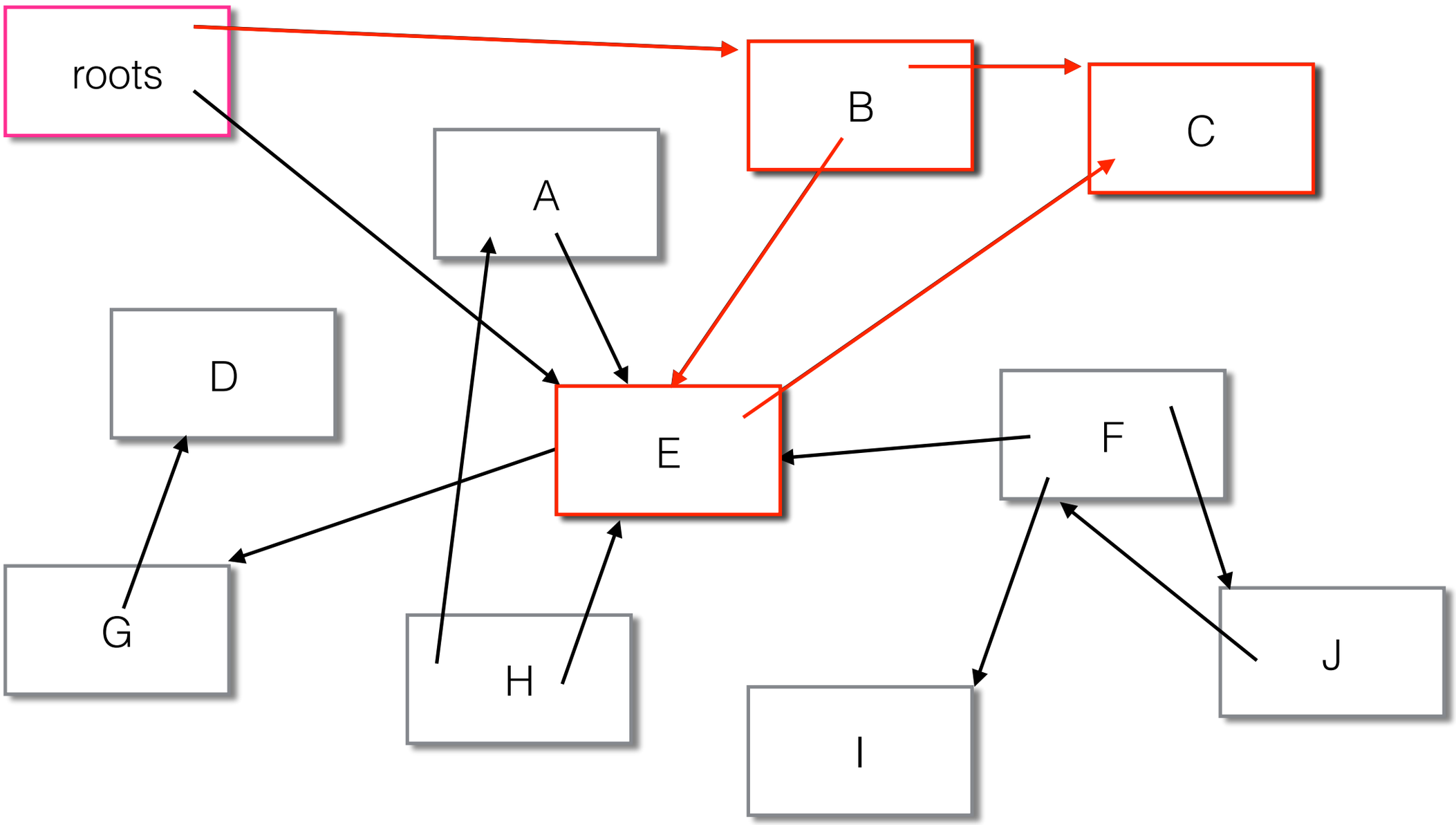
Marking



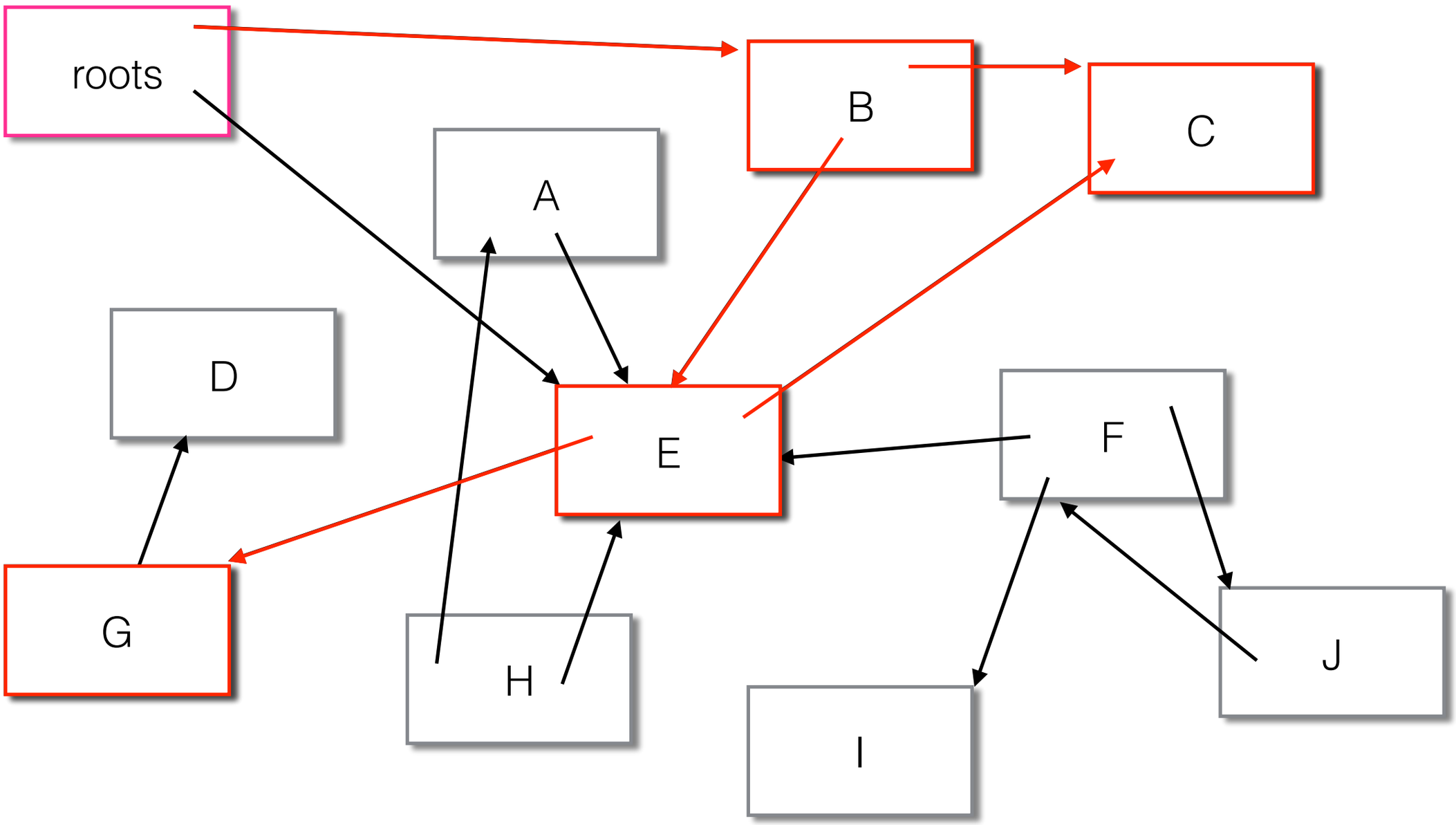
Marking



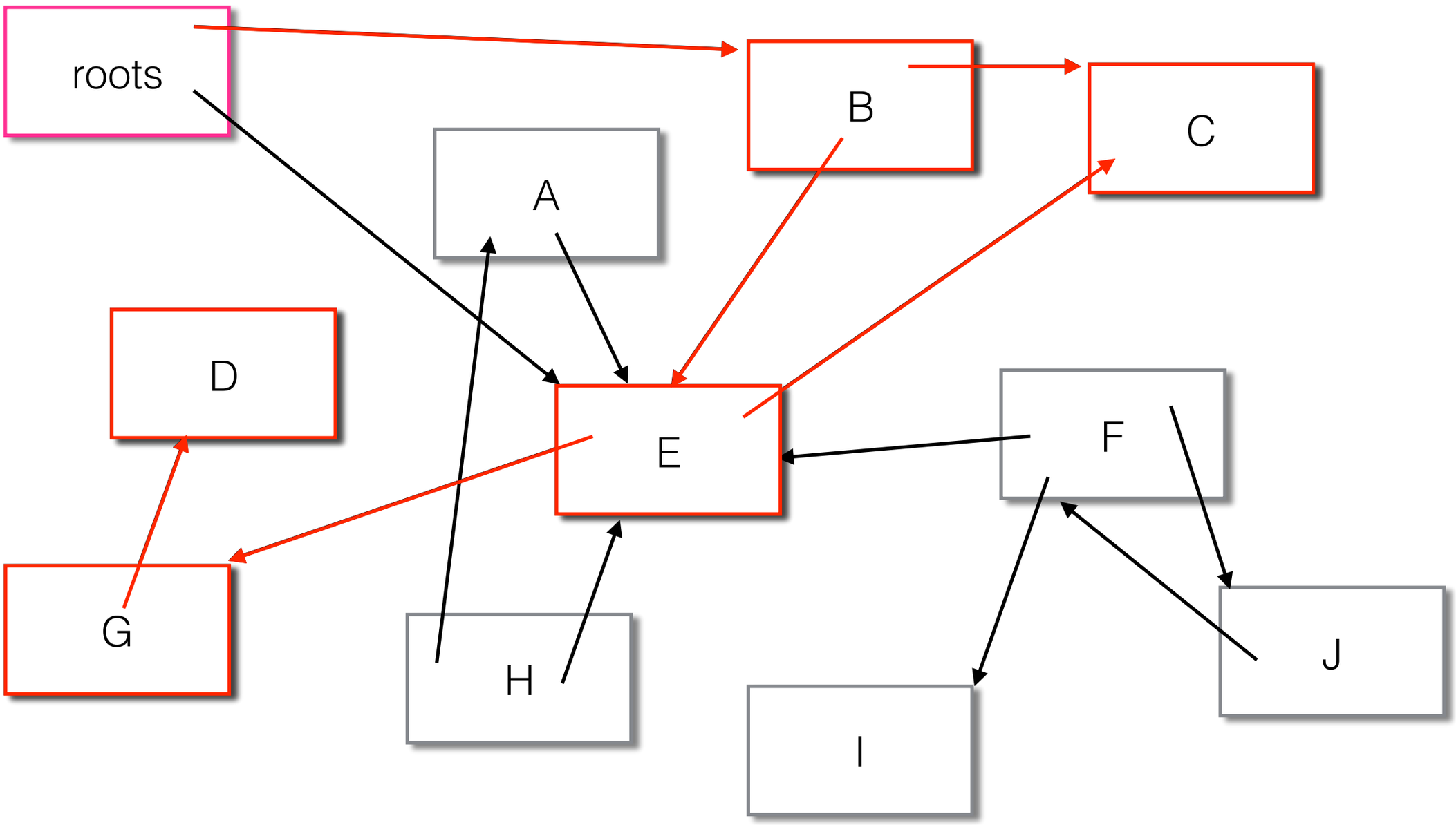
Marking



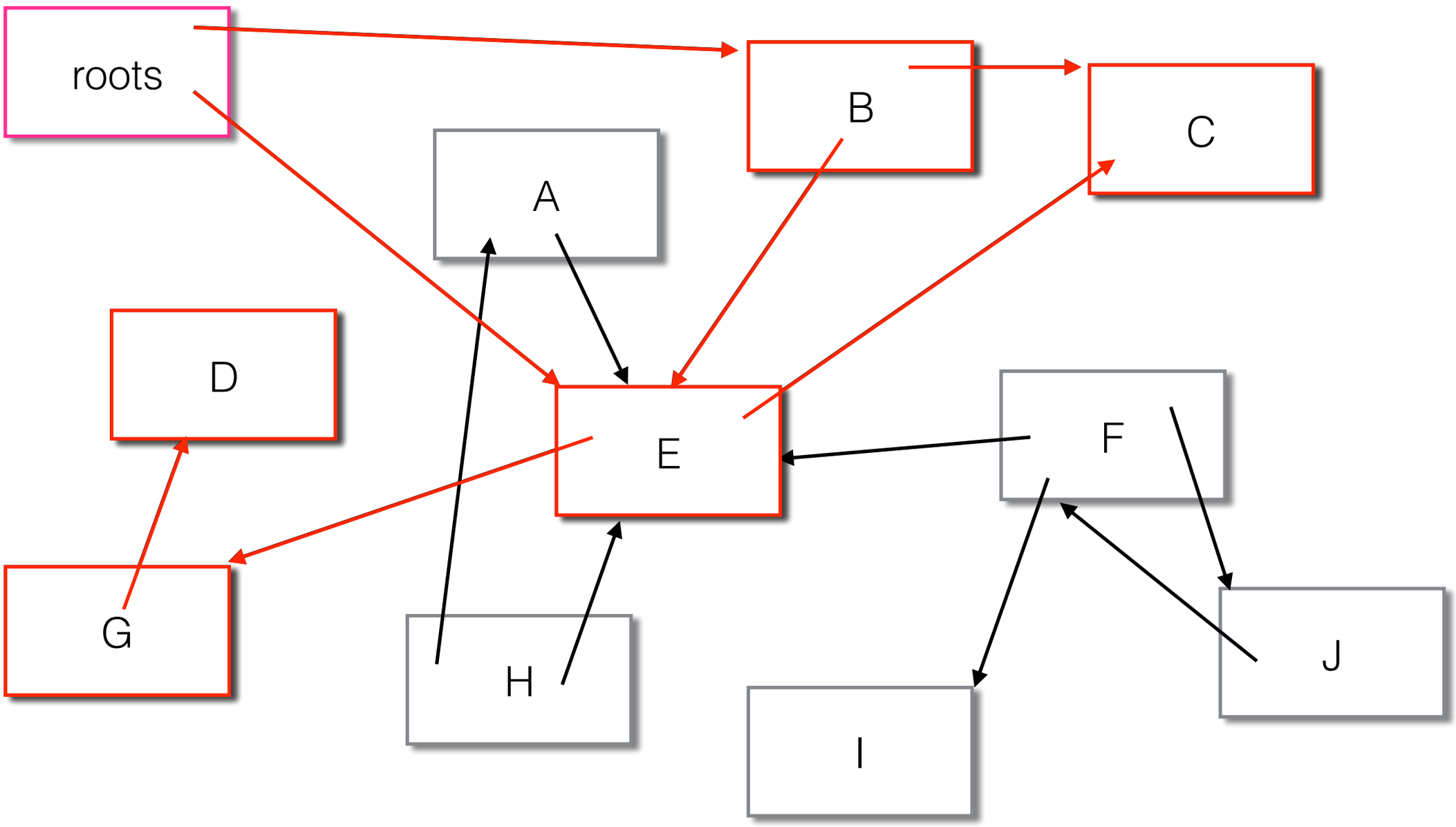
Marking



Marking



Marking



Recursive marking

```
typedef struct { int marked:1; … } *Obj;

// assumes no objects are marked to begin
for (each reference p in roots) mark(p);

void mark(Obj p) {
   if (p!=NULL && !p->marked) {
      p->marked= 1;
      for (each reference q in *p)
         mark(q);
   }
}
```

limit the recursion level.

mark the object

The mark stack

- In the worst case, the depth of recursion is bounded by the number of objects.
  - Imagine the heap containing a single linked list
- Using a work list improves this by a constant (one word per object, instead of one activation)
- Can you think of a scheme that requires only O(1) space?

Marking as a backup to reference counting

- If marking is used to deal with cycles in a reference counted system with saturating counters, the reference counts can be zeroed at the start and recomputed during marking (each mark operation increments the associated count, up to saturation)
  - This may unstick some counts.

```
void mark(Obj p) {
  if (p!=NULL && p->RC!=MAXRC) {
    p->RC++;
    for (each reference q in *p)
      mark(q);
  }
}
```

Sweeping

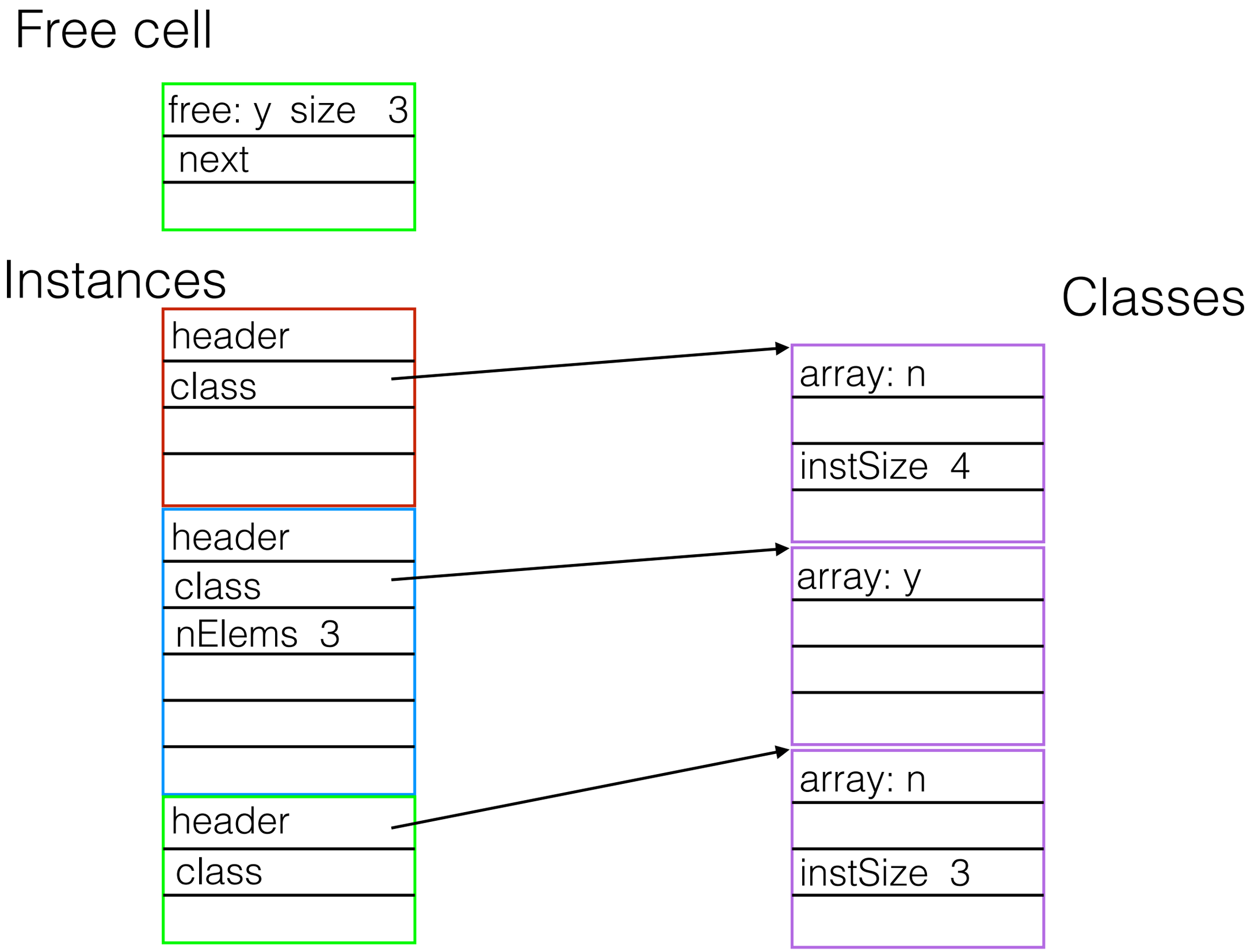
- Having finished marking, all unmarked objects are not reachable and can be reclaimed.
- One way is to sweep the heap, and add free objects to the free list(s).
- To do this we need to be able to enumerate all objects, alive or dead. This is typically done with a linear scan of the heap, and needs the heap to be *parseable* in one direction.

Parsing a heap

- Given a pointer to the start of a cell, we need to be able to find its size. We *could* store the size in the object's header.

- For instances of classes, this is usually recorded in the associated class; a pointer to the class is usually the first reference, at a fixed offset.

- For arrays, the length of the array is usually stored at a fixed offset (typically immediately after the class reference).


Free cell
free: y size 3
next
Instances
header
class
header
class
nElems 3
header
class
Classes
array: n
instSize 4
array: y
array: n
instSize 3

Addressing fragmentation with compaction

- Although we can easily coalesce adjacent free blocks during a sweep, this will not, in general, address fragmentation.

- To do this we need to use *compaction*. We can either replace the sweep with a compaction pass, or run compaction when an allocation fails despite there being sufficient free space.

- Compaction is easily implemented when there is an object table; in a direct pointer system, it is more complicated.

Sliding compaction

Sliding compaction

Sliding compaction

Sliding compaction

Sliding compaction

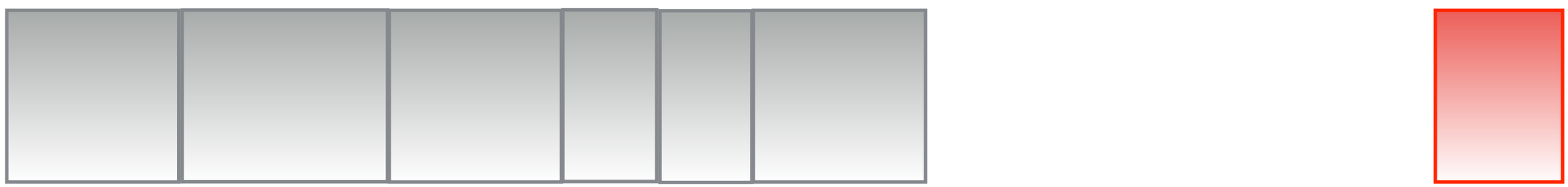
Sliding compaction

Sliding compaction

Sliding compaction

Sliding compaction

Sliding compaction



Sliding compaction

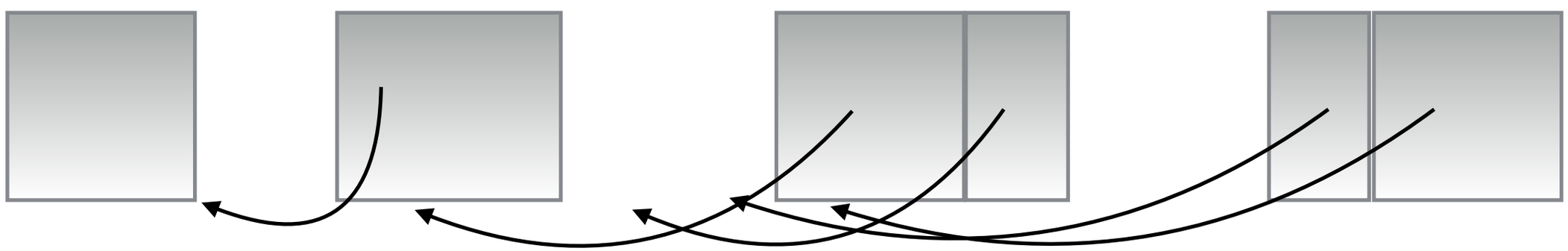
Sliding compaction

After compaction, we can use bump allocation

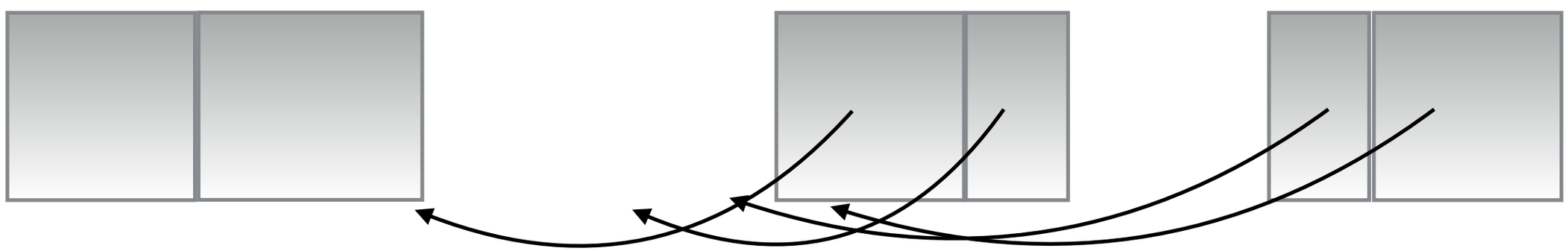
Compaction using forwarding pointers

- Compaction relocates objects, so all the pointers need to be fixed up. The easiest solution is to reserve a word in each object for a *forwarding pointer*.
- One pass through the heap computes forwarding pointers, a second pass updates all pointers using their forwarding addresses, and the final pass performs the compaction.

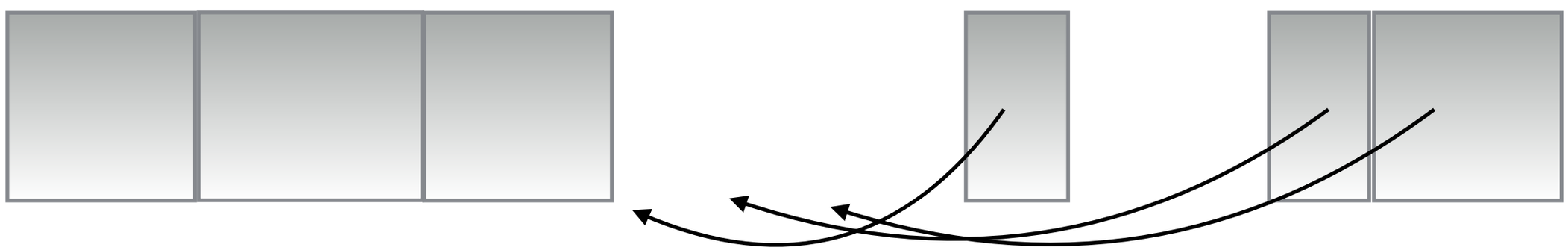
Sliding compaction using forwarding pointers



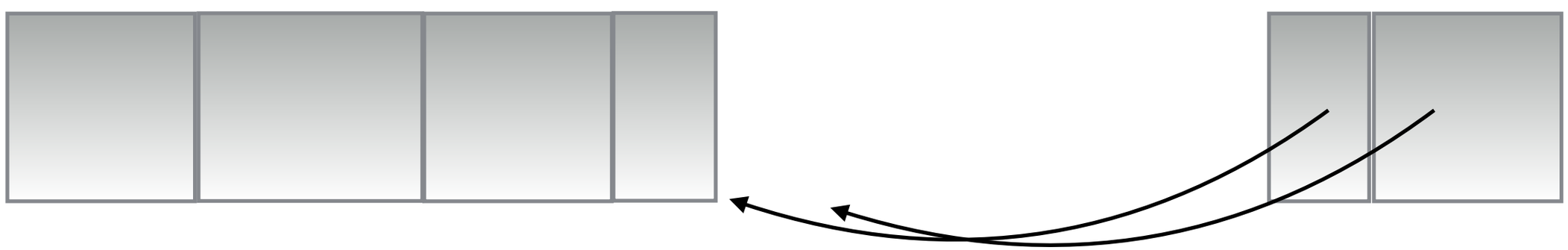
Sliding compaction using forwarding pointers



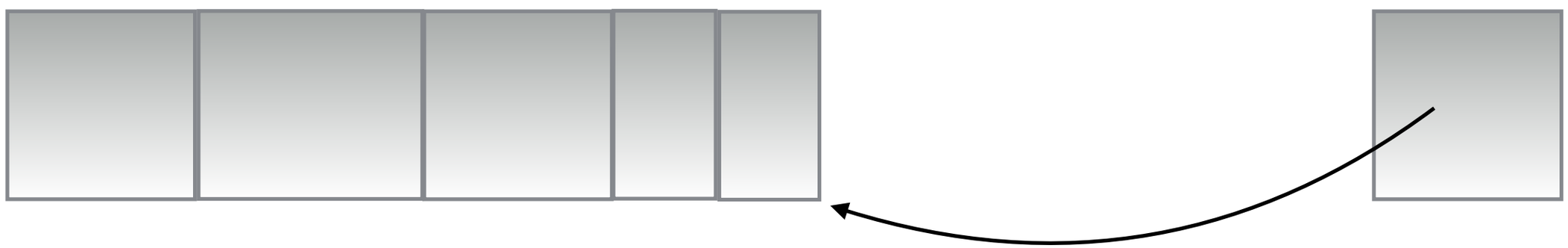
Sliding compaction using forwarding pointers



Sliding compaction using forwarding pointers



Sliding compaction using forwarding pointers



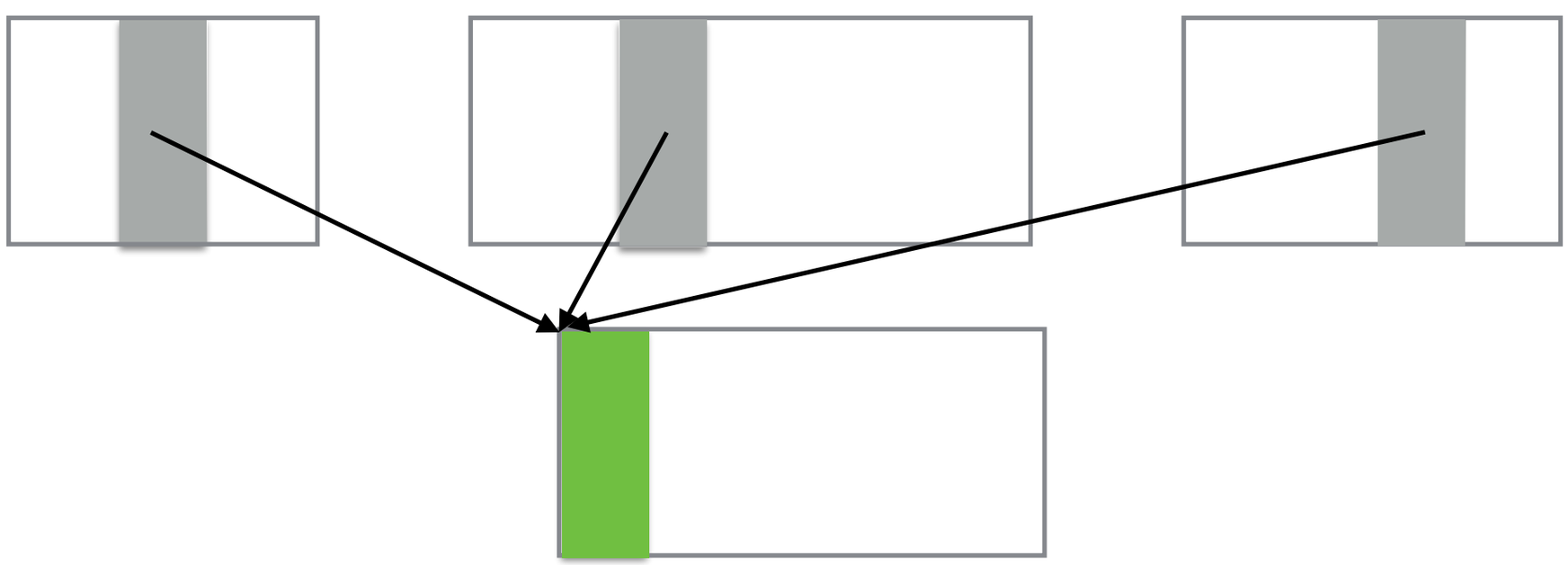
Sliding compaction using forwarding pointers

Compaction using a temporary object table

- Instead of paying for the space for forwarding pointers all the time, an alternative is to construct a separate table used just during compaction.
- The header word is replaced with a pointer to a table entry; a copy of the header is stored in the table. Another table field contains the forwarding location.

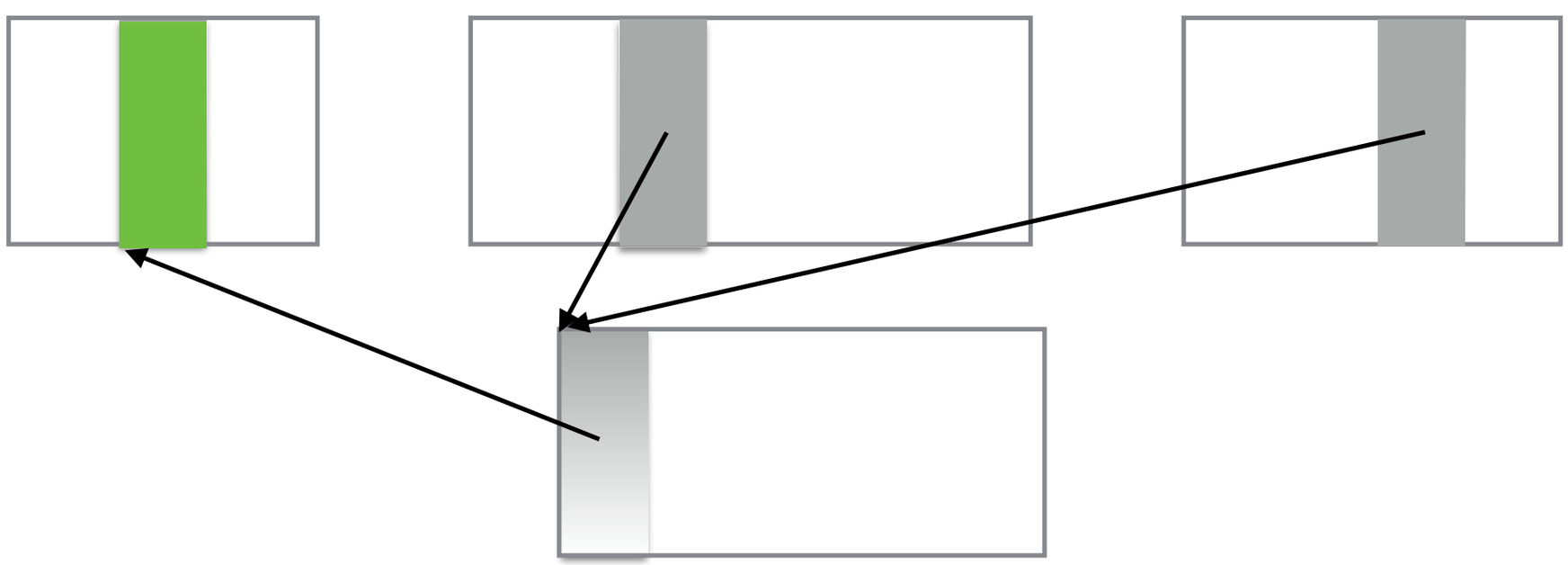
Threaded compaction

- If the use of additional space is a problem, then an alternative is to use threading (unrelated to any meaning you are already familiar with). This replaces every reference to an object with a chain of references, moving the header word to the end of the chain.



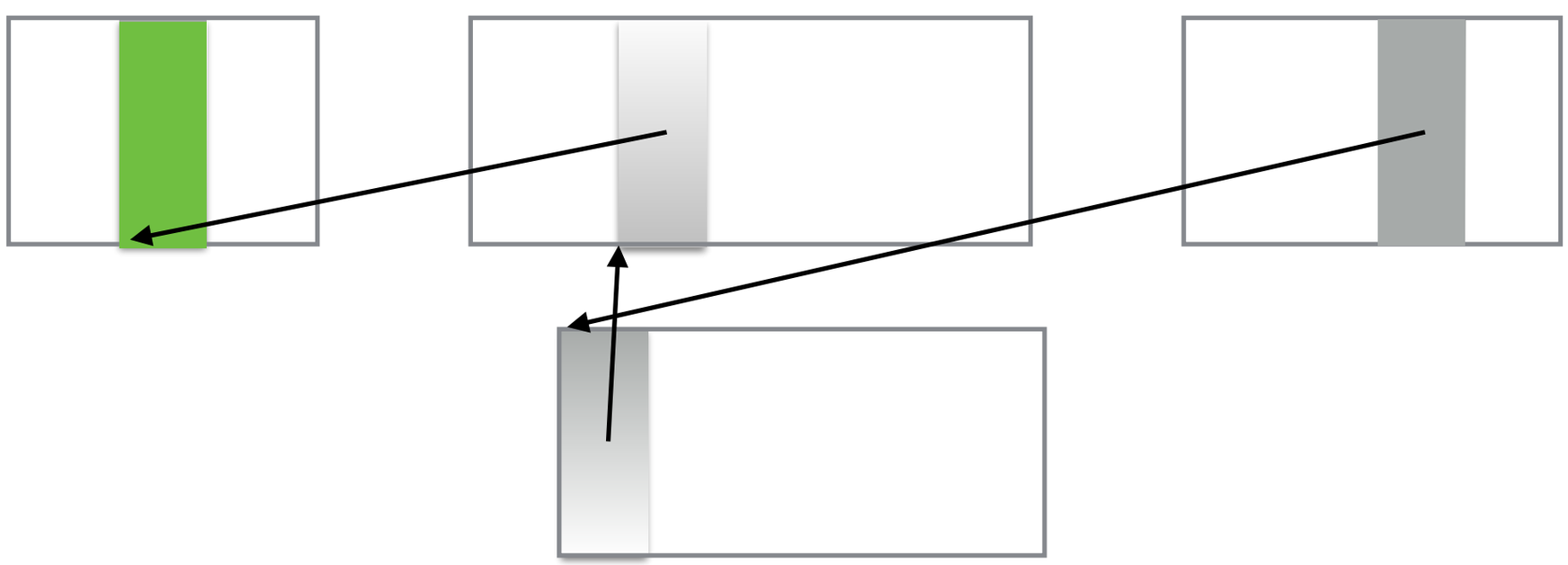
Threaded compaction

- If the use of additional space is a problem, then an alternative is to use threading (unrelated to any meaning you are already familiar with). This replaces every reference to an object with a chain of references, moving the header word to the end of the chain.



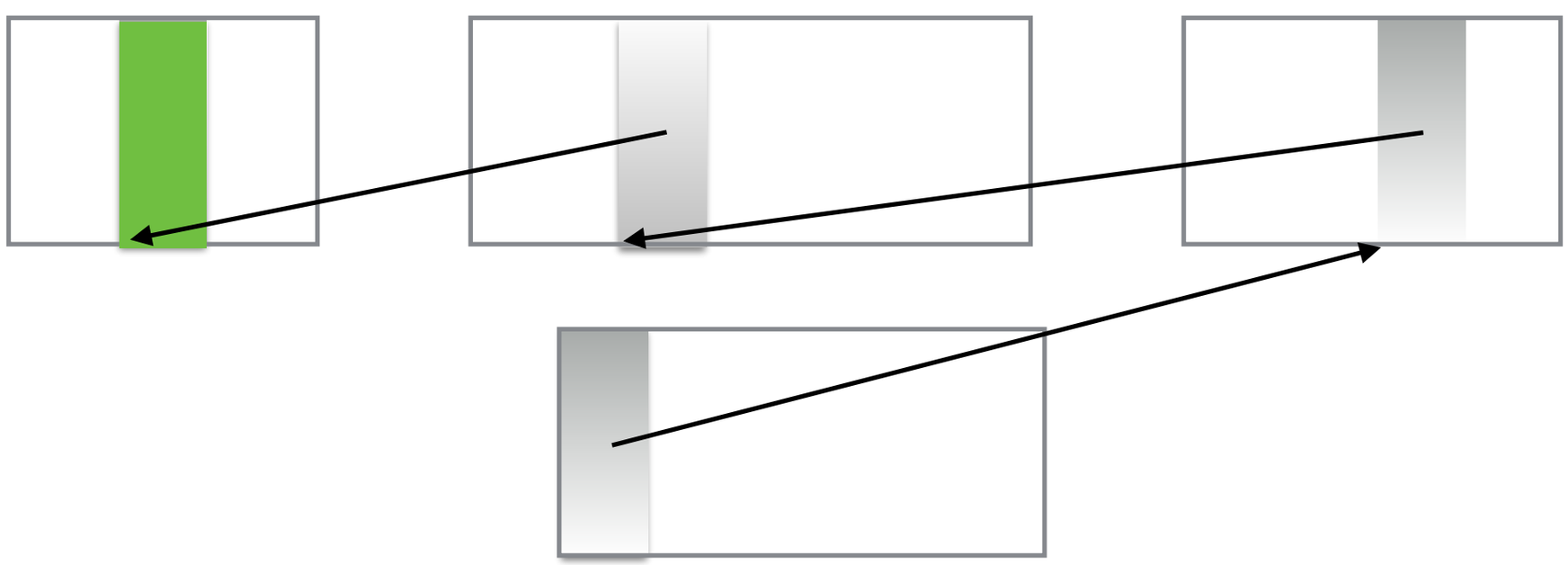
Threaded compaction

- If the use of additional space is a problem, then an alternative is to use threading (unrelated to any meaning you are already familiar with). This replaces every reference to an object with a chain of references, moving the header word to the end of the chain.



Threaded compaction

- If the use of additional space is a problem, then an alternative is to use threading (unrelated to any meaning you are already familiar with). This replaces every reference to an object with a chain of references, moving the header word to the end of the chain.



Threaded compaction

- After threading is complete, every reference to a live object is threaded from the object's header word, with the header's contents moved to the end of the thread.
- After an object is moved during compaction, the thread can be replaced with references to its new location.

Memory management
Part (4)
Copying Collection

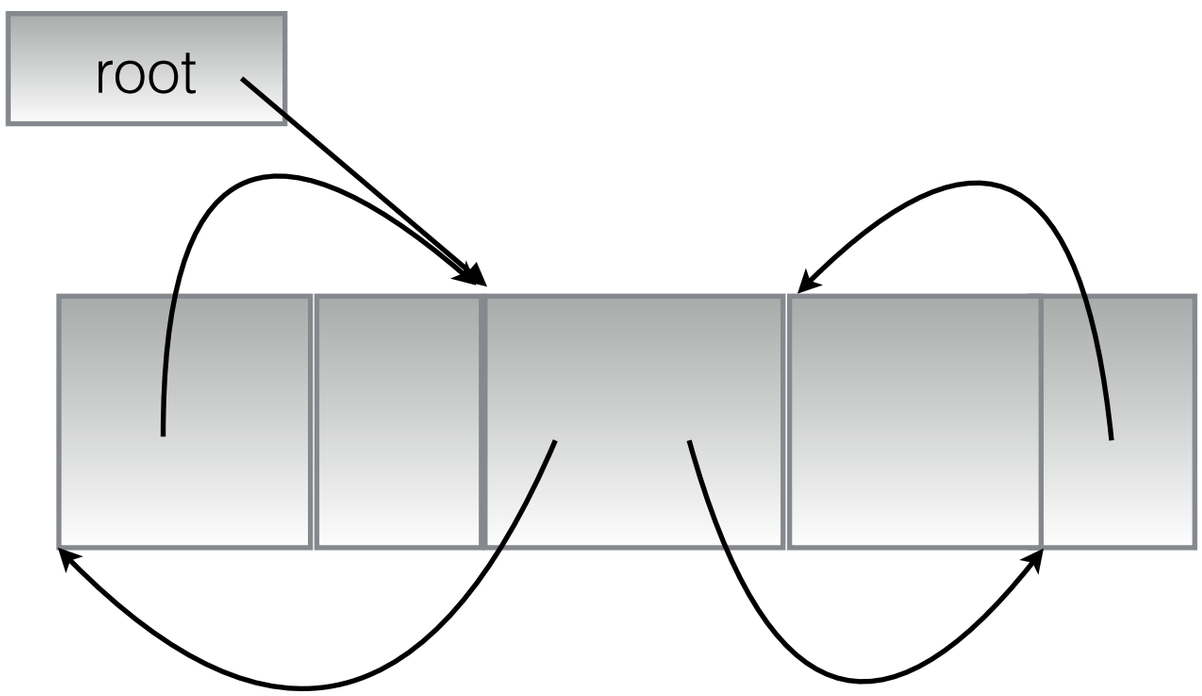
Copying collection

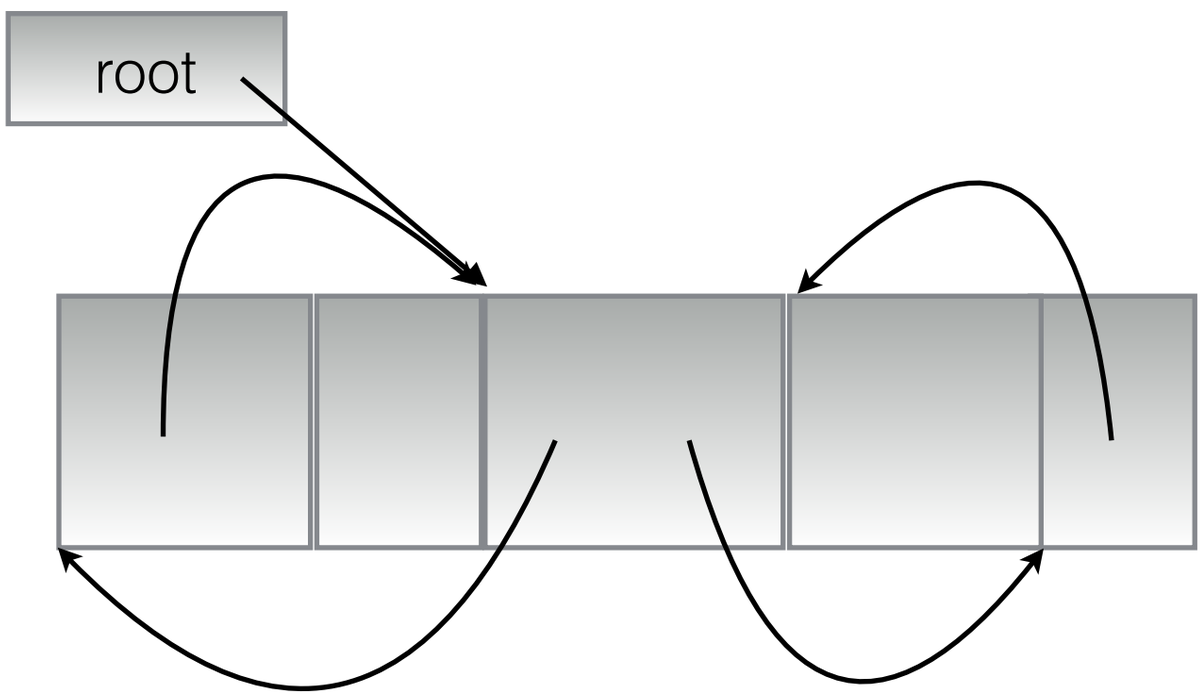
- In a copying collector, we combine tracing and compaction into a single operation.
- When an object is first traced, it is copied, and a forwarding pointer left behind.
- We divide memory into two semi-spaces: we run in a single semi-space, and copy into the other when the first fills, leaving behind all the garbage.

Copying collection: The Movie

root

root

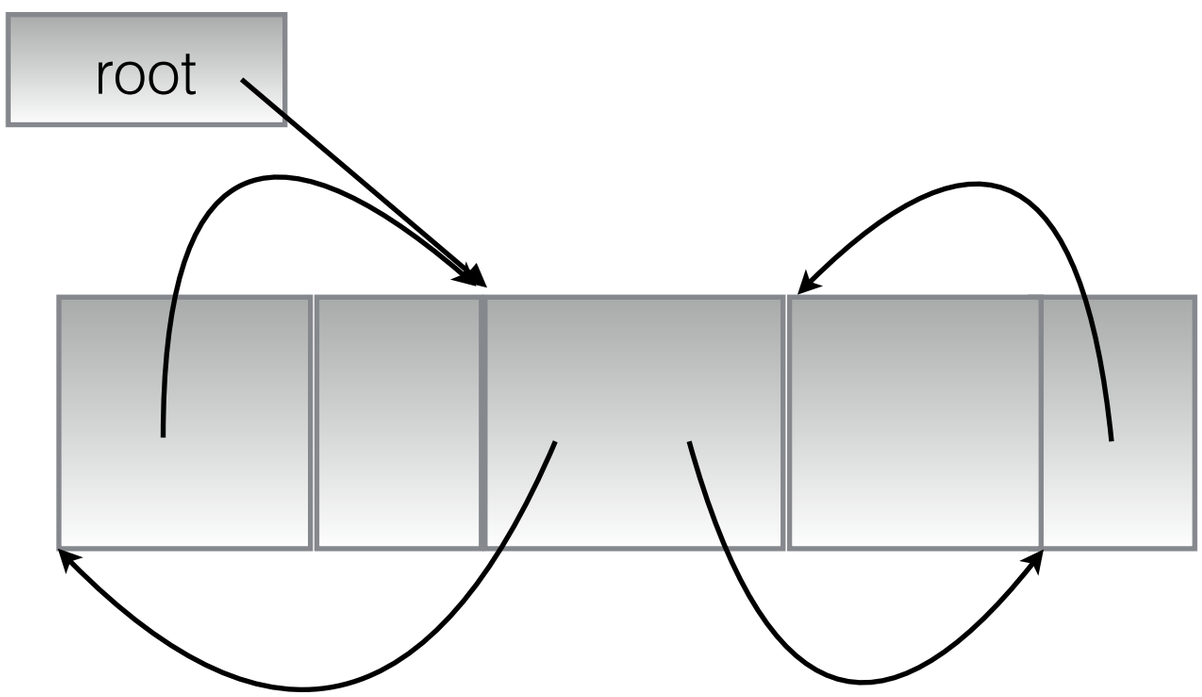

root



From

To

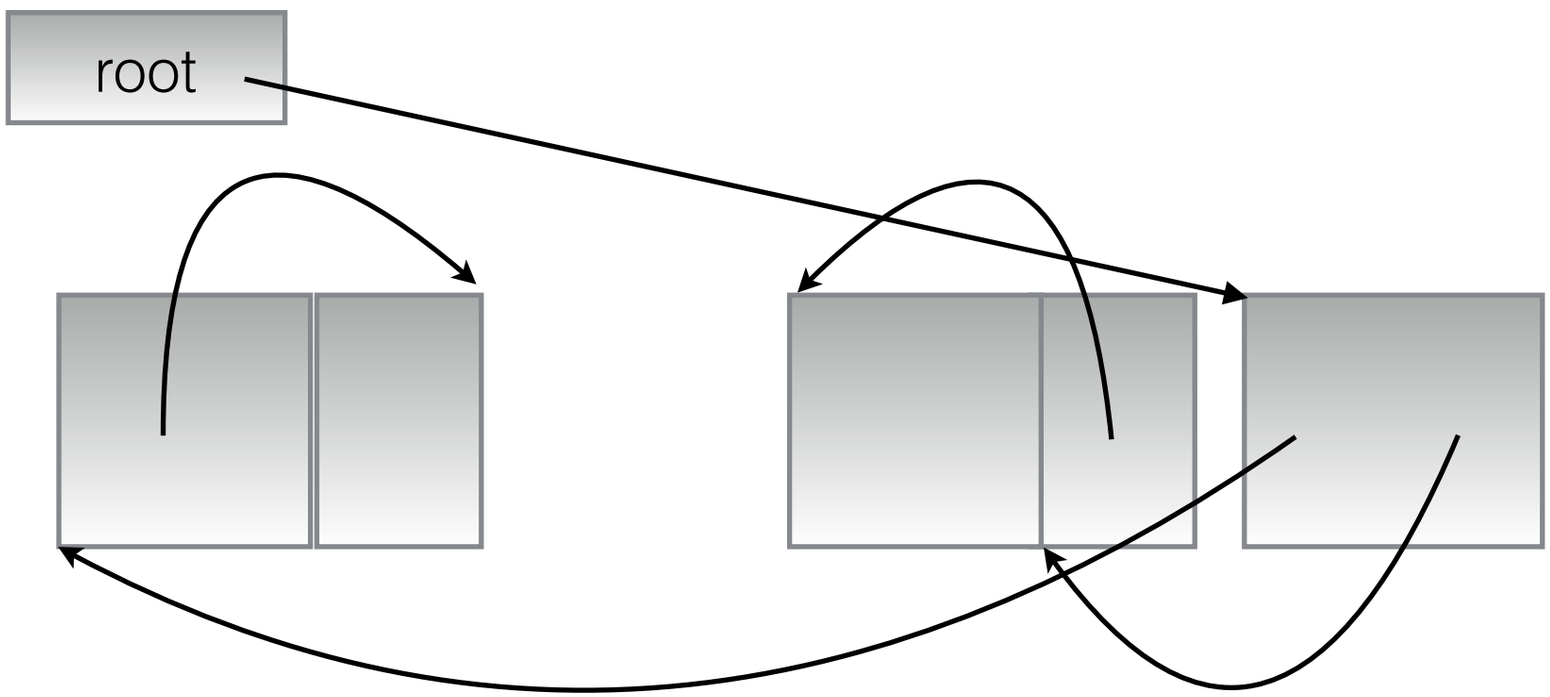
Process roots



From

To

Process roots

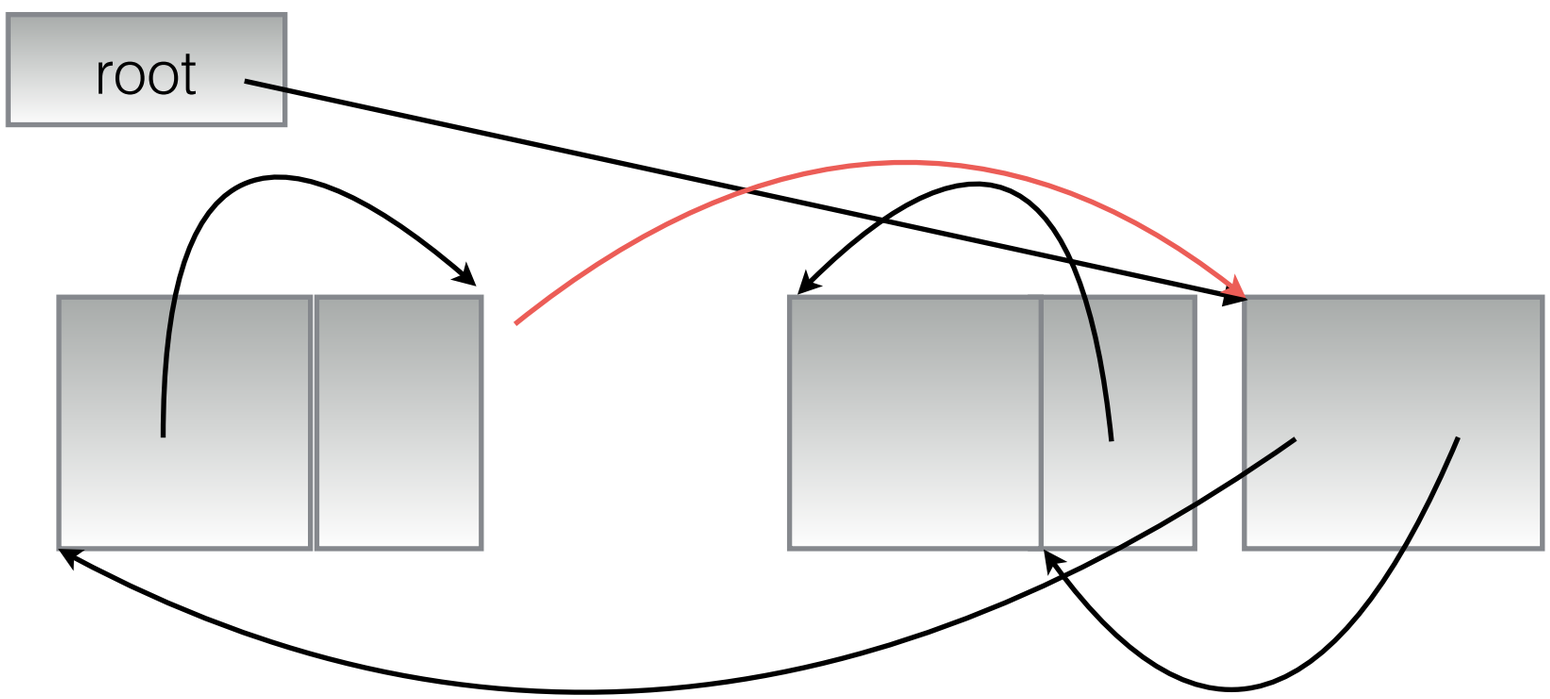


From

To

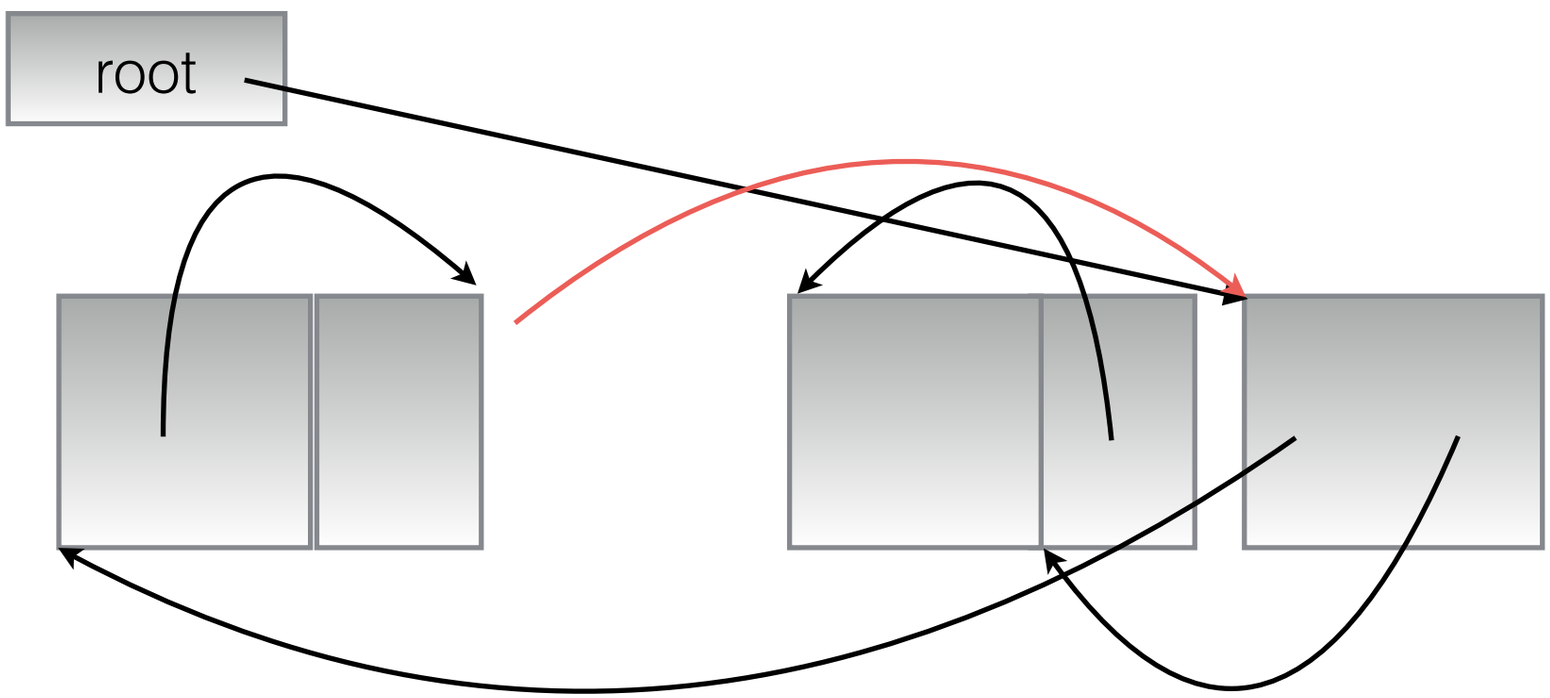
Process roots

Forwarding pointer



From

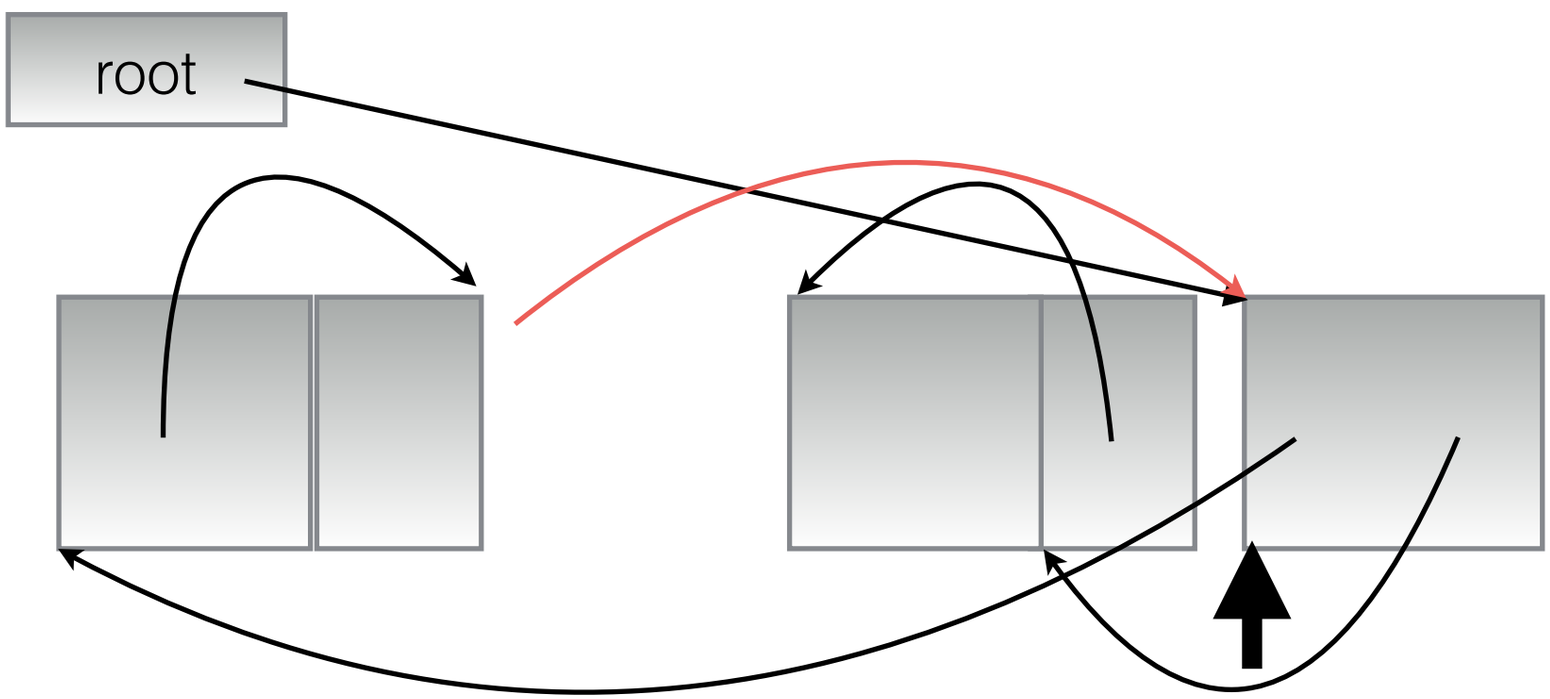
To



From

To

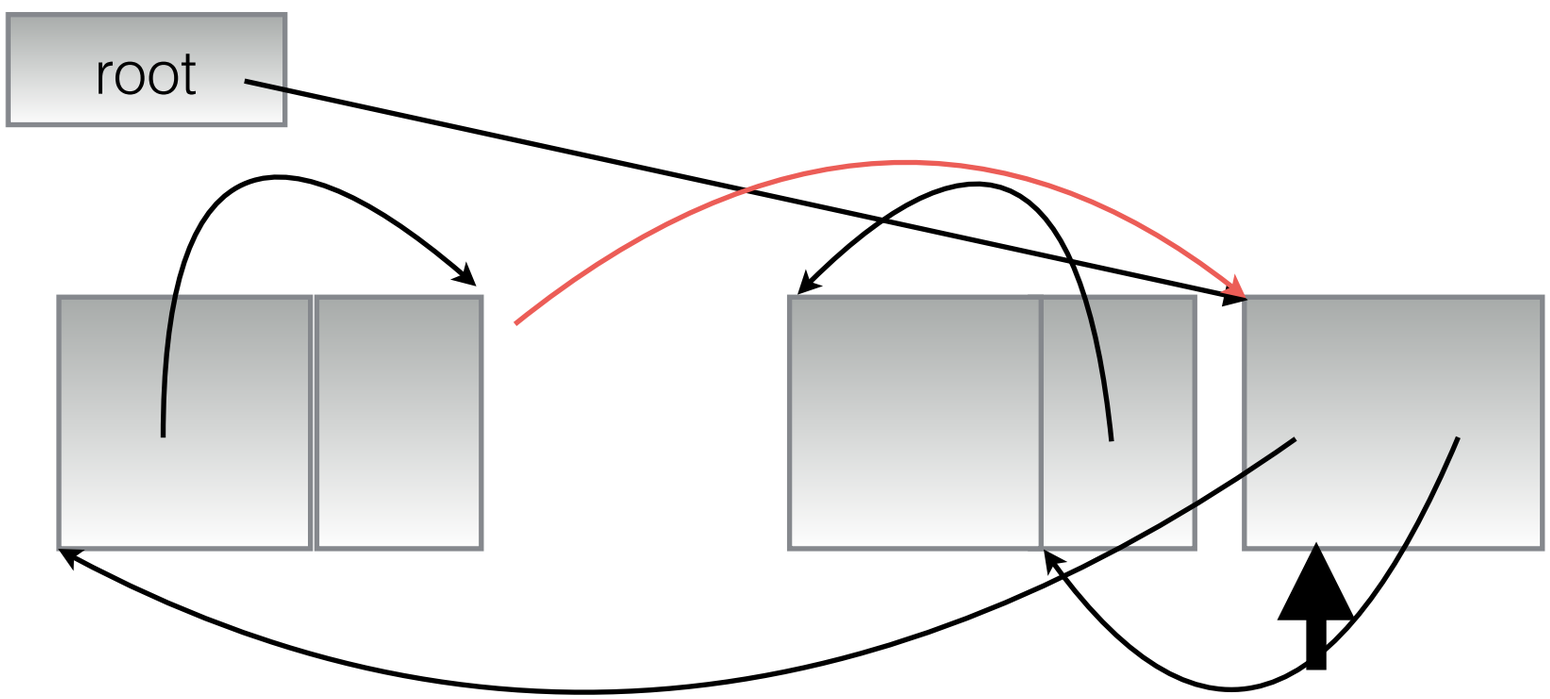
Begin scan



From

To

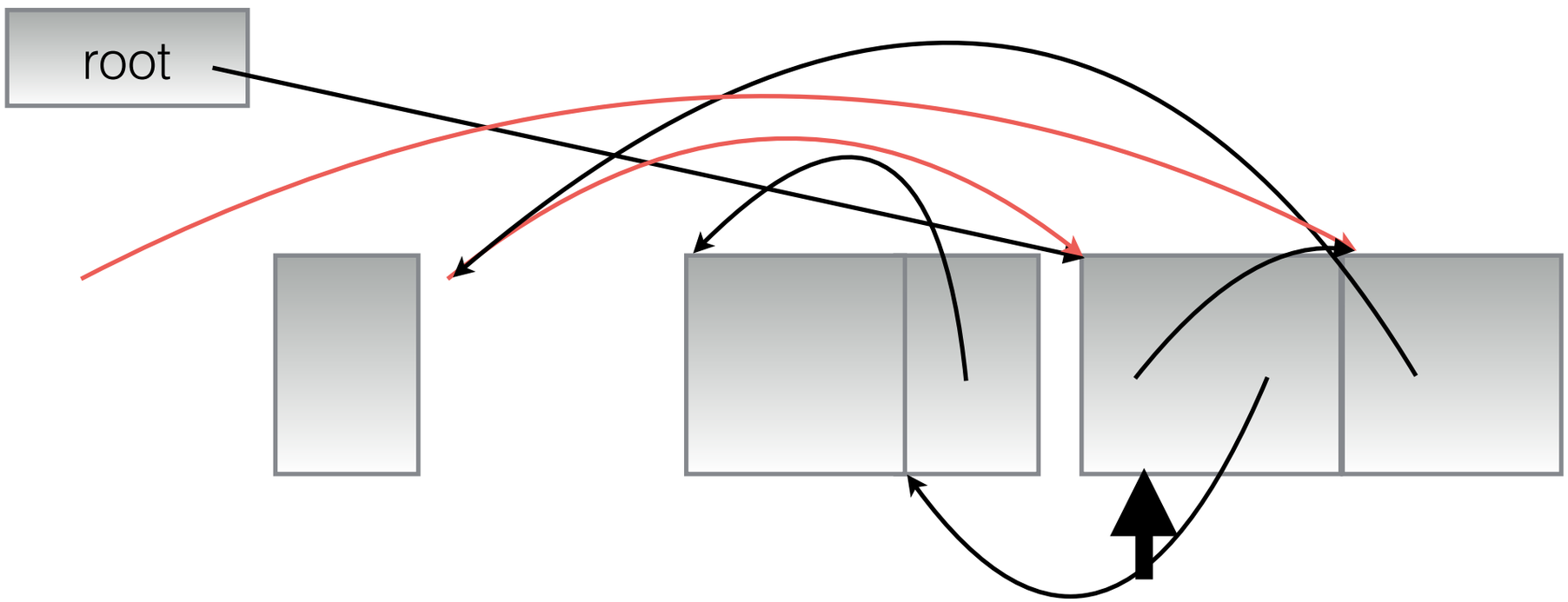
Scan



From

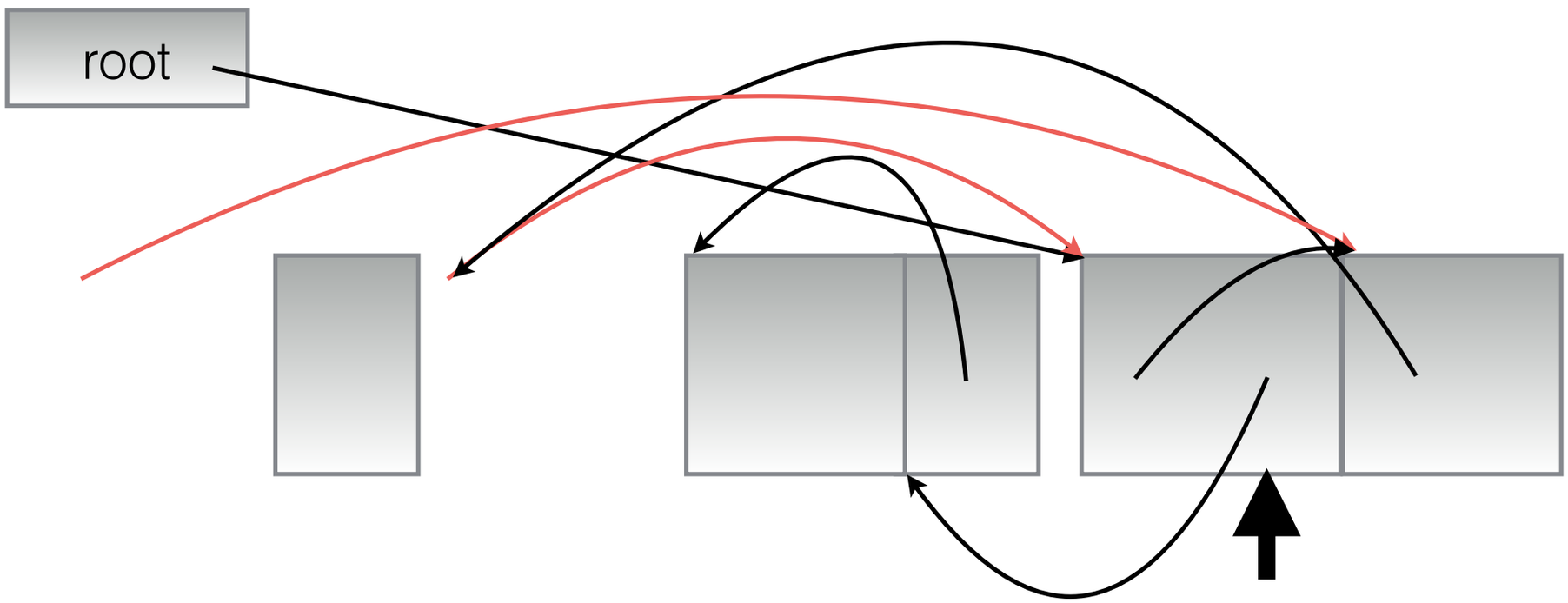
To

Scan



From To

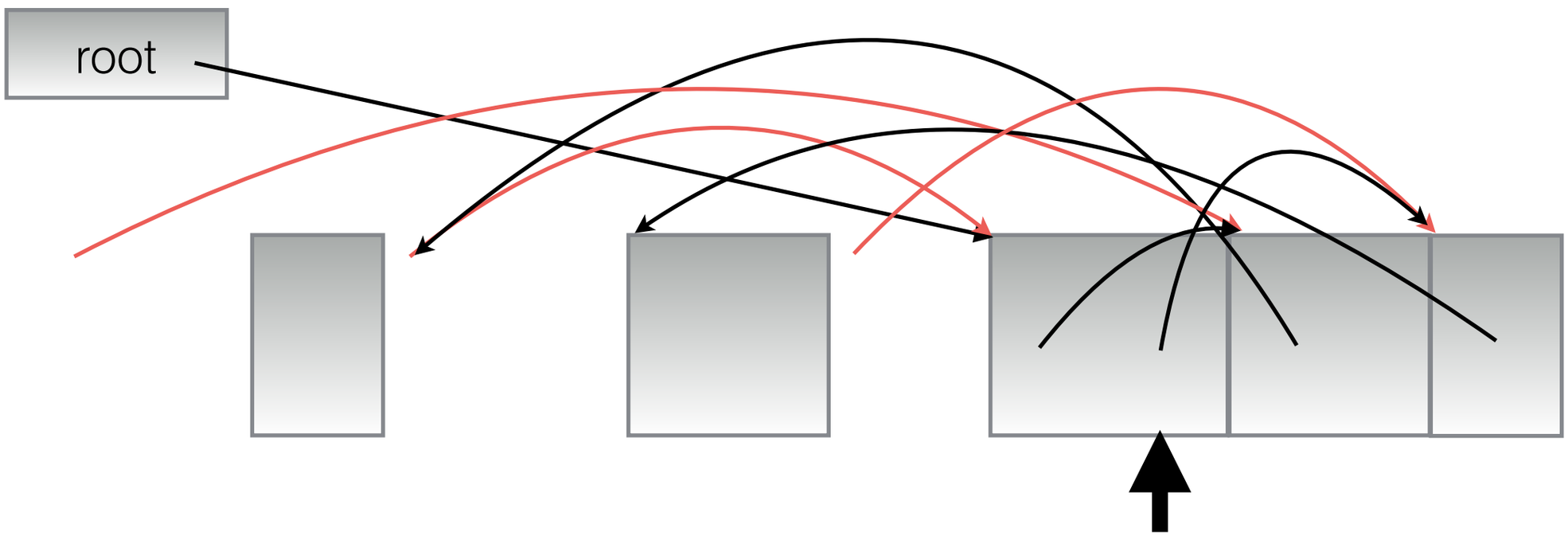
Scan



From

To

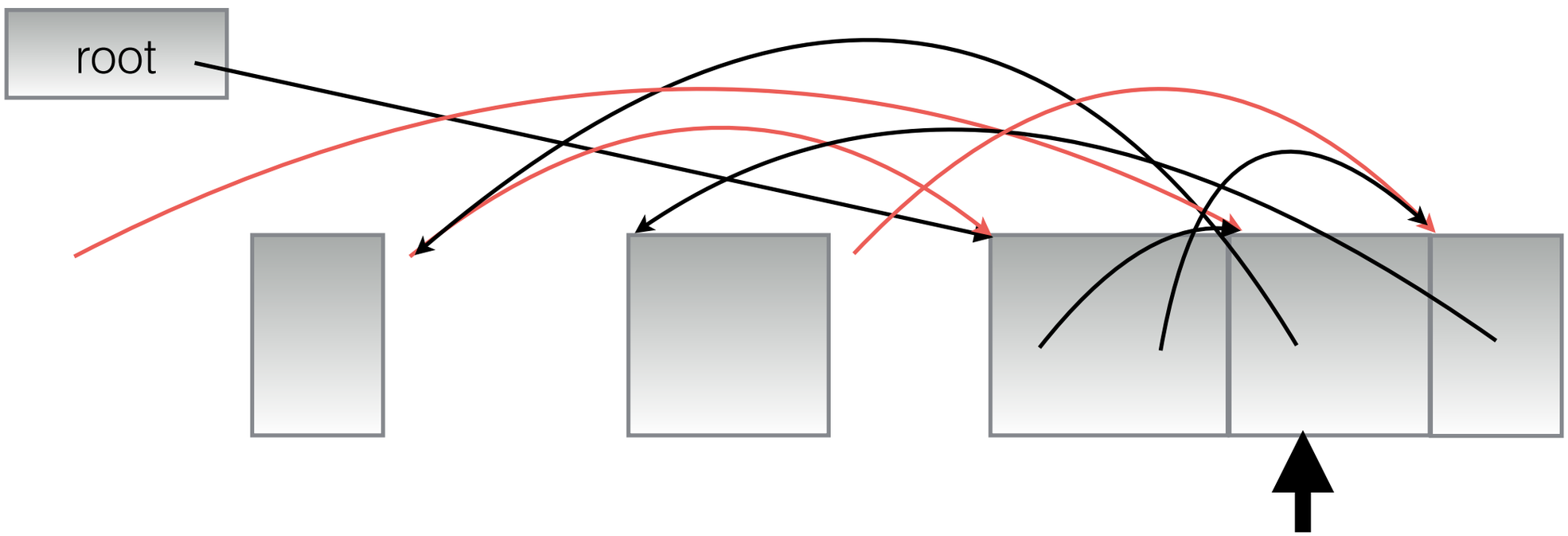
Scan



From

To

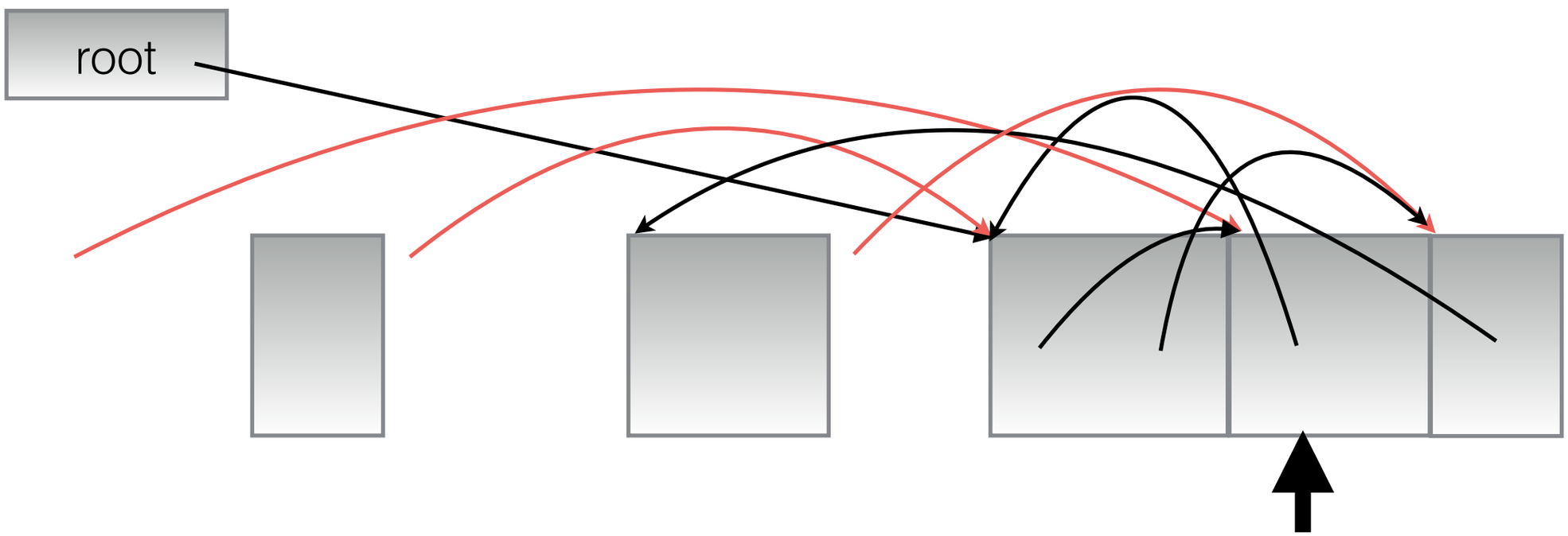
Scan



From

To

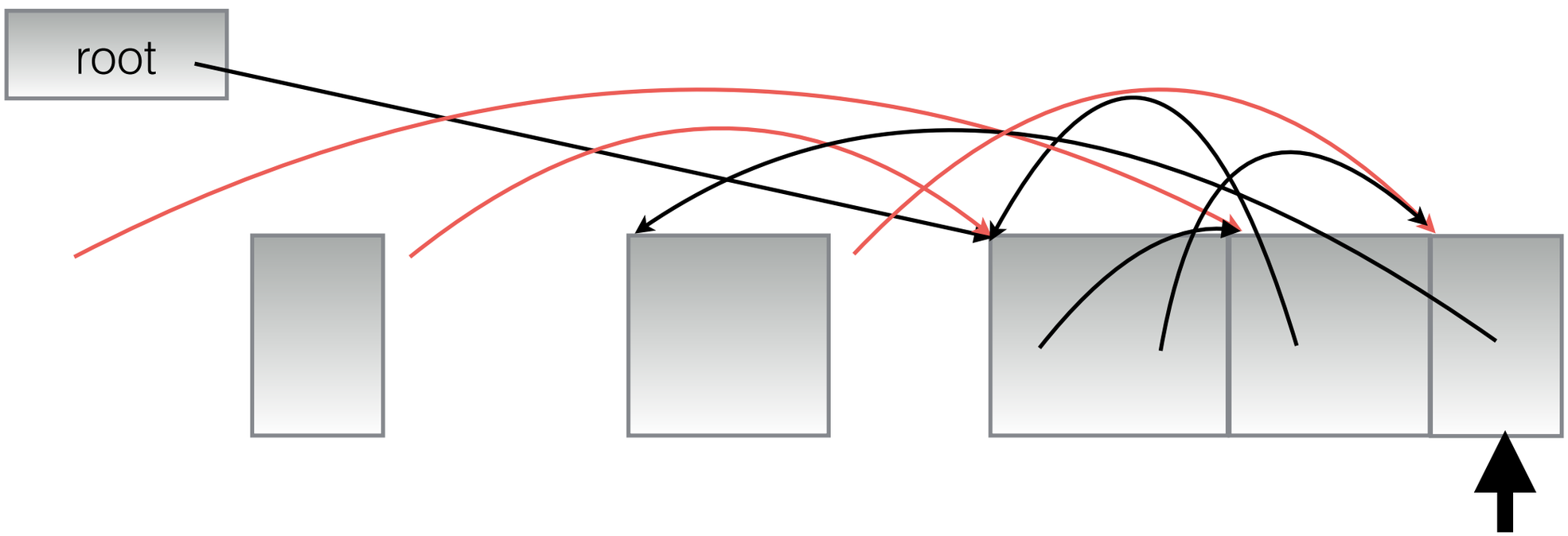
Scan



From

To

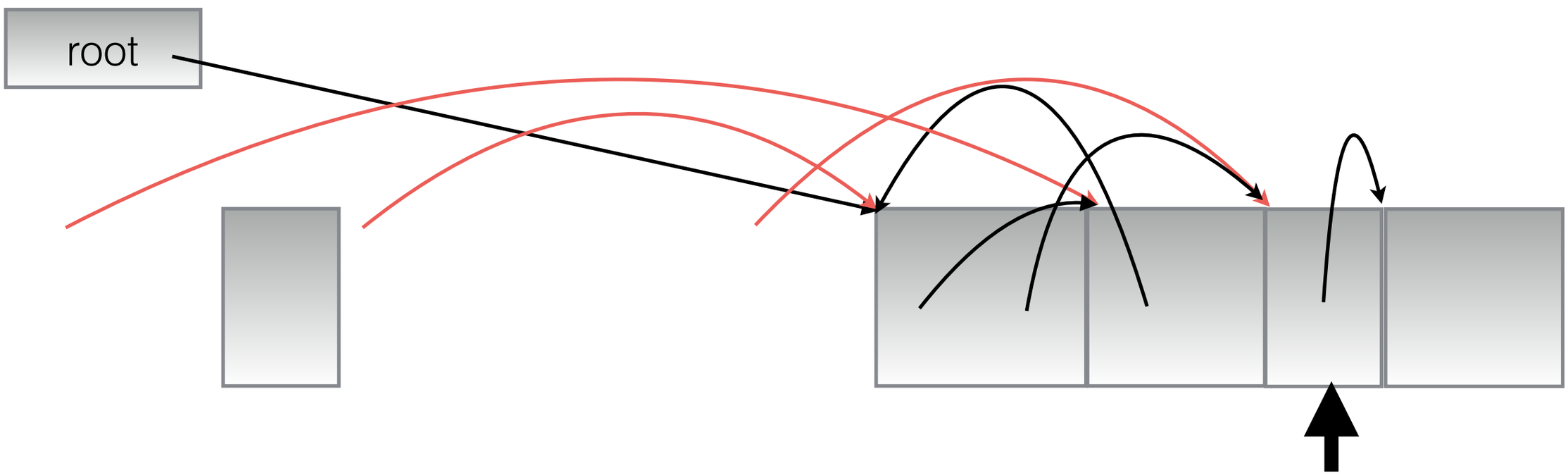
Scan



From

To

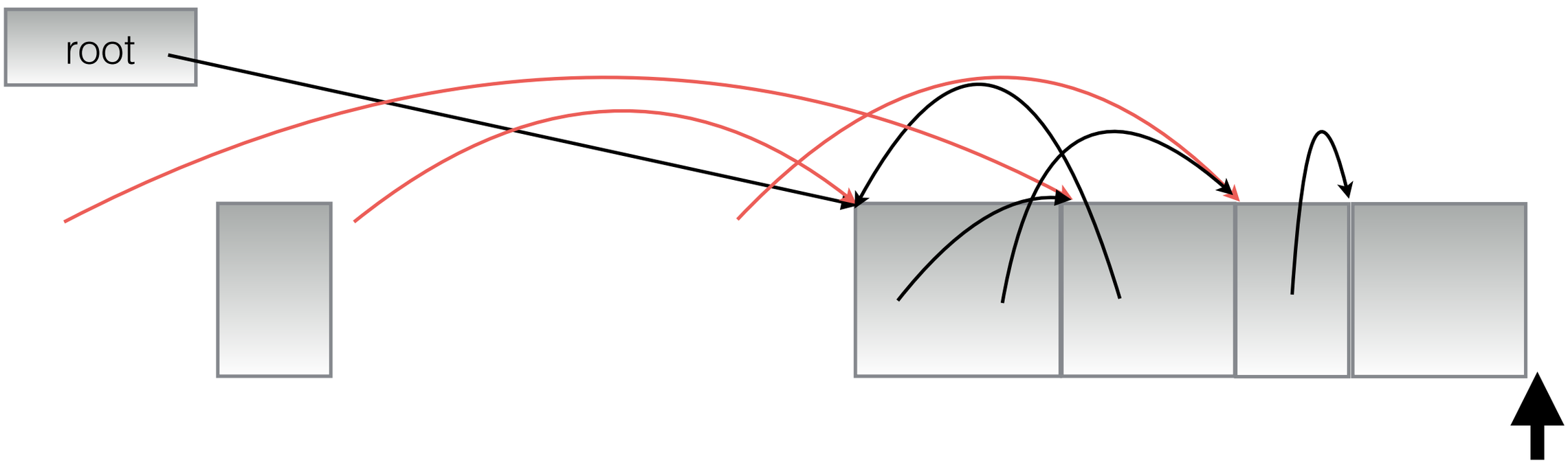
Scan



From

To

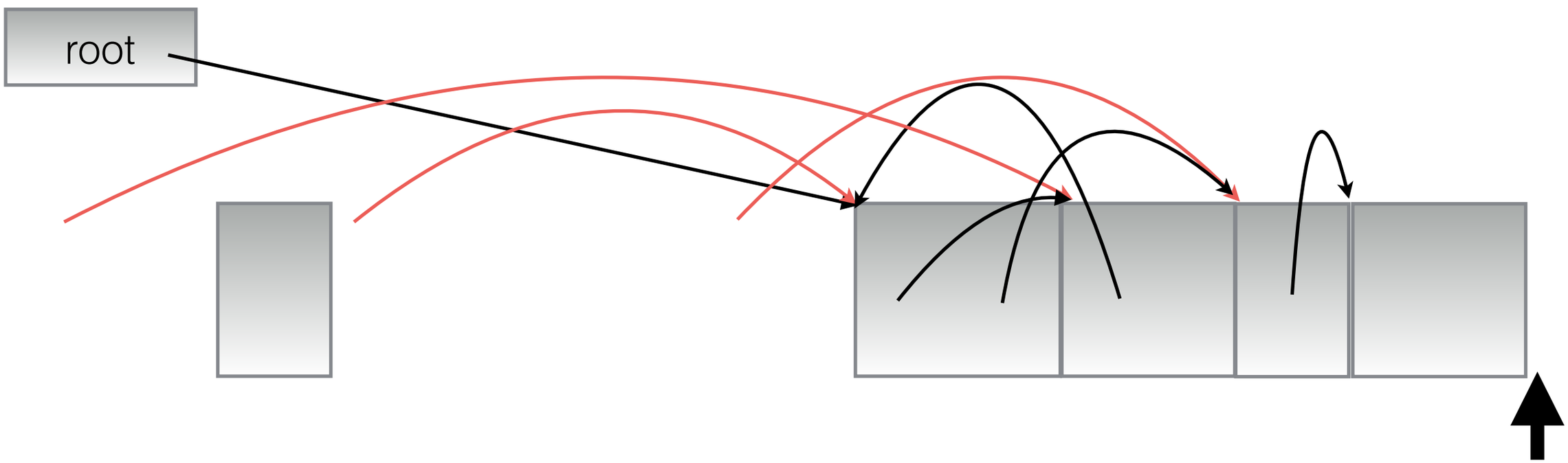
Scan



From

To

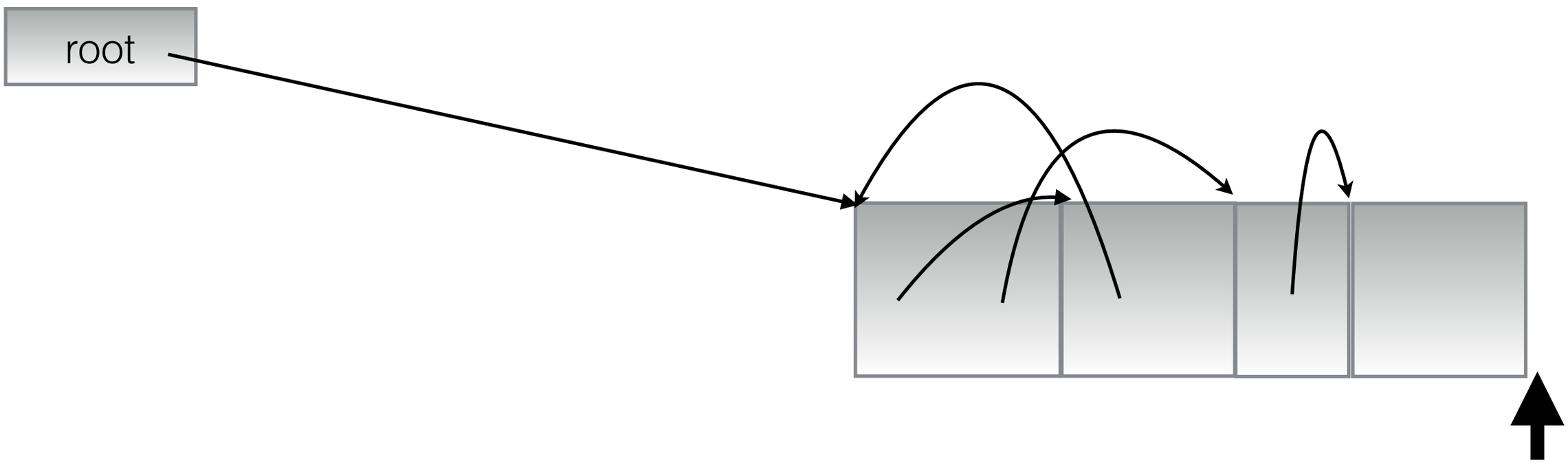
Flip



From

To

Flip



To

From

Copying

- Breadth-first, rather than depth-first
  - No need for a marking stack
  - Breadth-first usually degrades locality
- Uses twice the memory, what the hey? Virtual Memory to the rescue! Oh, wait…
- Enables bump allocation
- The linear scan to perform copying and forwarding is sometimes known as a *Cheney scan* [Cheney, 1970].

Exercise

- Add a semi-space copying collector to your Feeny bytecode interpreter and measure the changes in space and speed.

Memory management: To be continued…