Compilation 2024

Garbage Collection

Aslan Askarov
aslan@cs.au.dk

based on slides by E.Ernst, J. Midtgaard, and M. I. Schwartzbach

Garbage

- The code in runtime.c leaks memory!
- Records and arrays are allocated but the memory is never freed

- Q: why would it ever need to be freed?

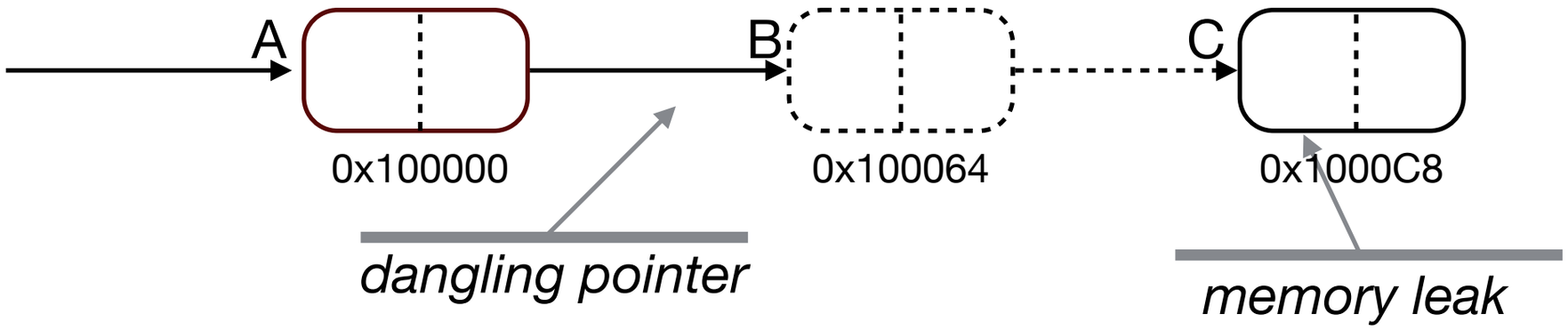
- Need memory management:
  - C-style: give programmers `free()`
  - Java-style: use automatic memory reclamation, aka ‘garbage collection’
- Ancient war: “too slow” vs “the civilized choice”

Garbage collector

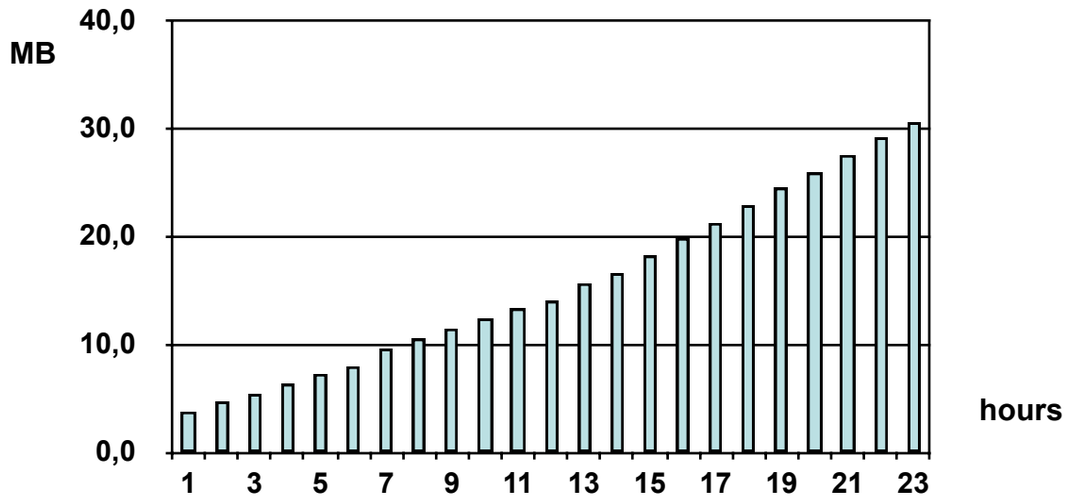
- A garbage collector is part of the runtime system
- It reclaims heap-allocated records (objects) that are no longer in use
- Garbage collector should
  - reclaim all unused records
  - spend very little time per record
  - avoid causing significant delays
  - allow all of memory to be used
- These are difficult and conflicting requirements

Life w/o garbage collection

- Unused records must be explicitly deallocated
- May be superior to automatic memory management, if done correctly
- Easy to miss some records
- Dangerous to handle pointers



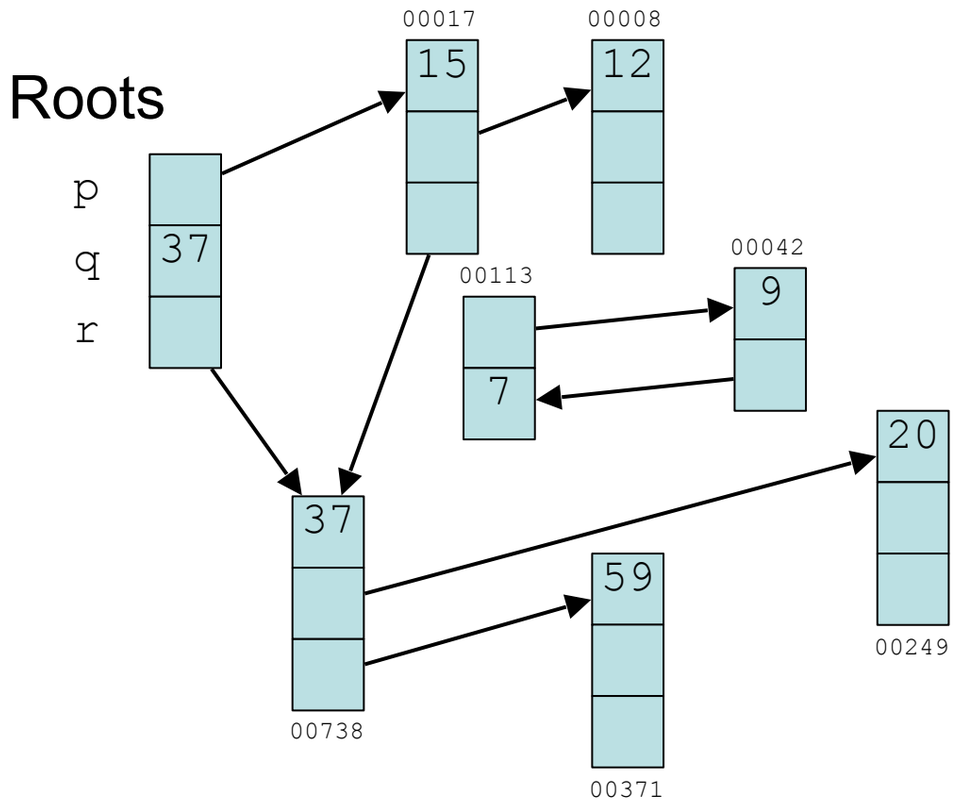
- Memory leaks in real life
- (iCal v.2.1):



Record liveness

- Which records are still “in use”?
- Ideally those that will be accessed in the future execution of the program
  - Obviously, undecidable
- But can be approximated conservatively
  - A record *is live* if it is reachable from a root location (local var, local stack, global var, etc)
  - Dead records may still point to each other

A heap with live and dead records



Mark-and-sweep algorithm

- Explore pointers starting from all root locations and *mark* all records encountered
- *Sweep* through all records in the heap and reclaim the unmarked ones
- Unmark all marked records

- Assumptions:
  - we know the start and size of each record in memory
  - we know which record fields are pointers
  - reclaimed records are kept in a `freelist`

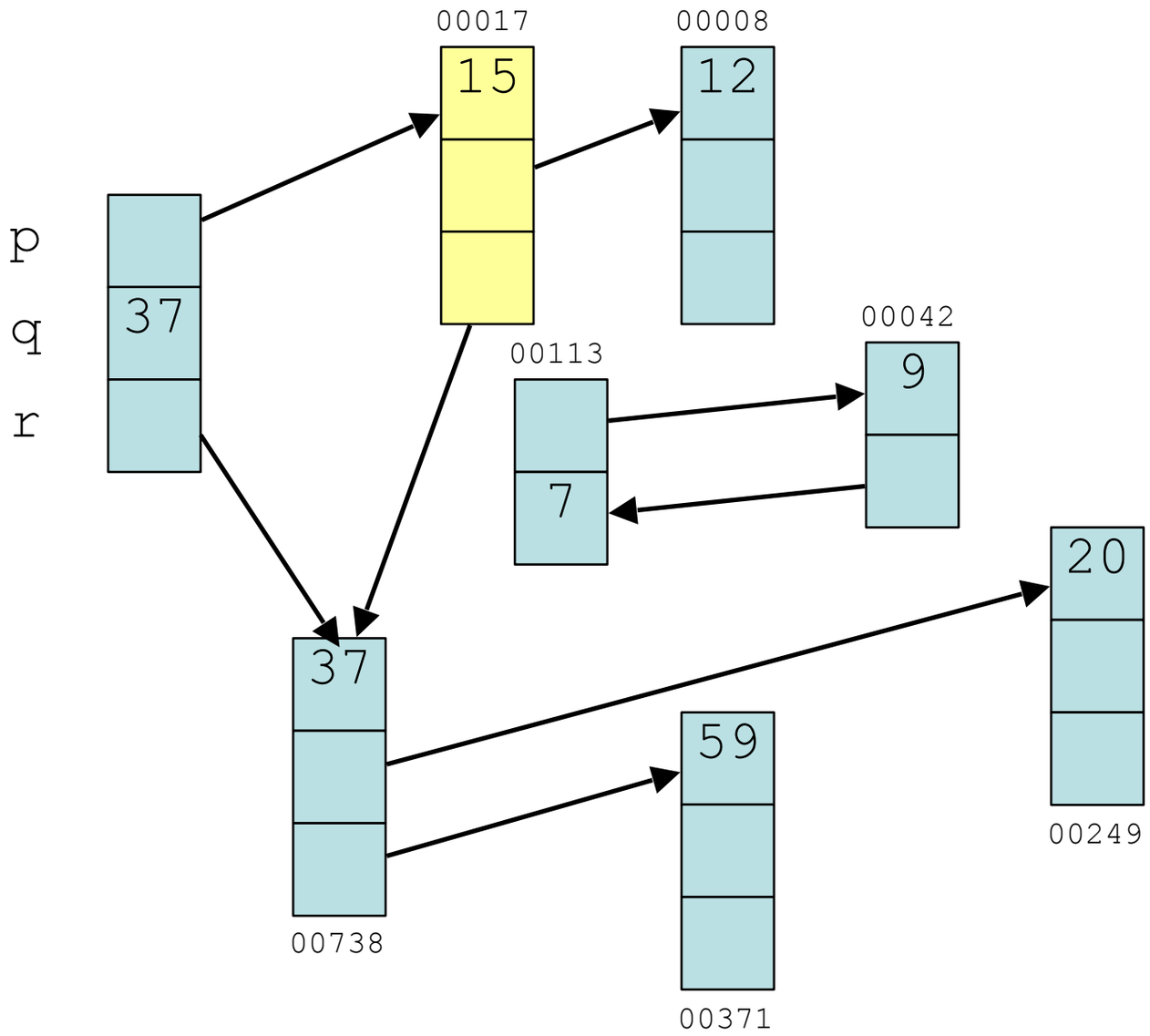
Mark-and-sweep pseudocode

```
function Mark() {
  foreach (v in a stack frame)
    DFS(v);
}
```

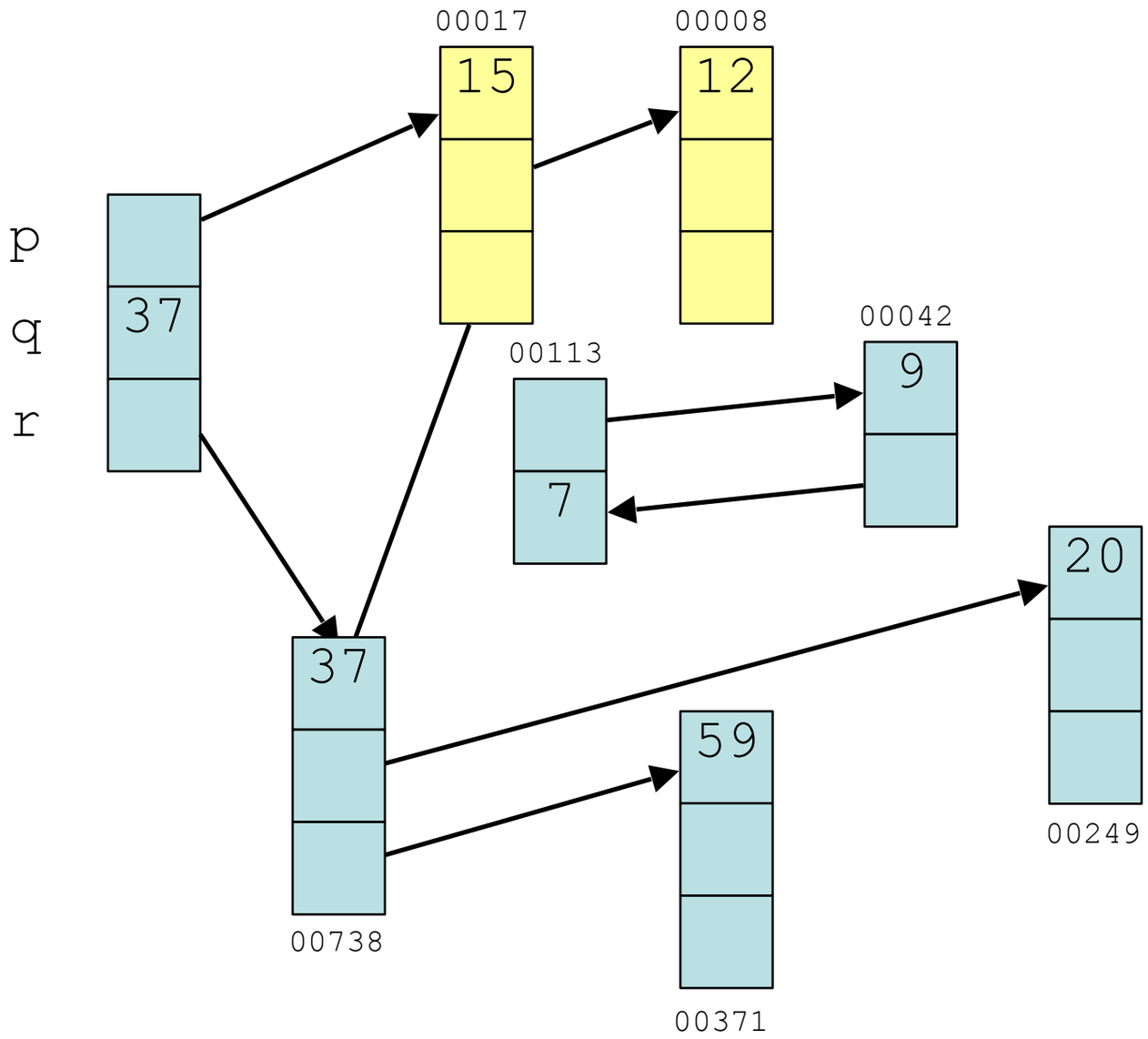
```
function DFS(x) {
  if (x is a heap pointer)
    if (x is not marked) {
      mark x;
      for (i=1; i<=|x|; i++)
        DFS(x.f_i)
    }
}
```

```
function Sweep() {
  p = first address in heap;
  while (p<last address in heap) {
    if (p is marked)
      unmark p;
    else {
      p.f_1 = freelist;
      freelist = p;
    }
    p = next object pointer after p
  }
}
```

Marking and Sweeping (1/11)

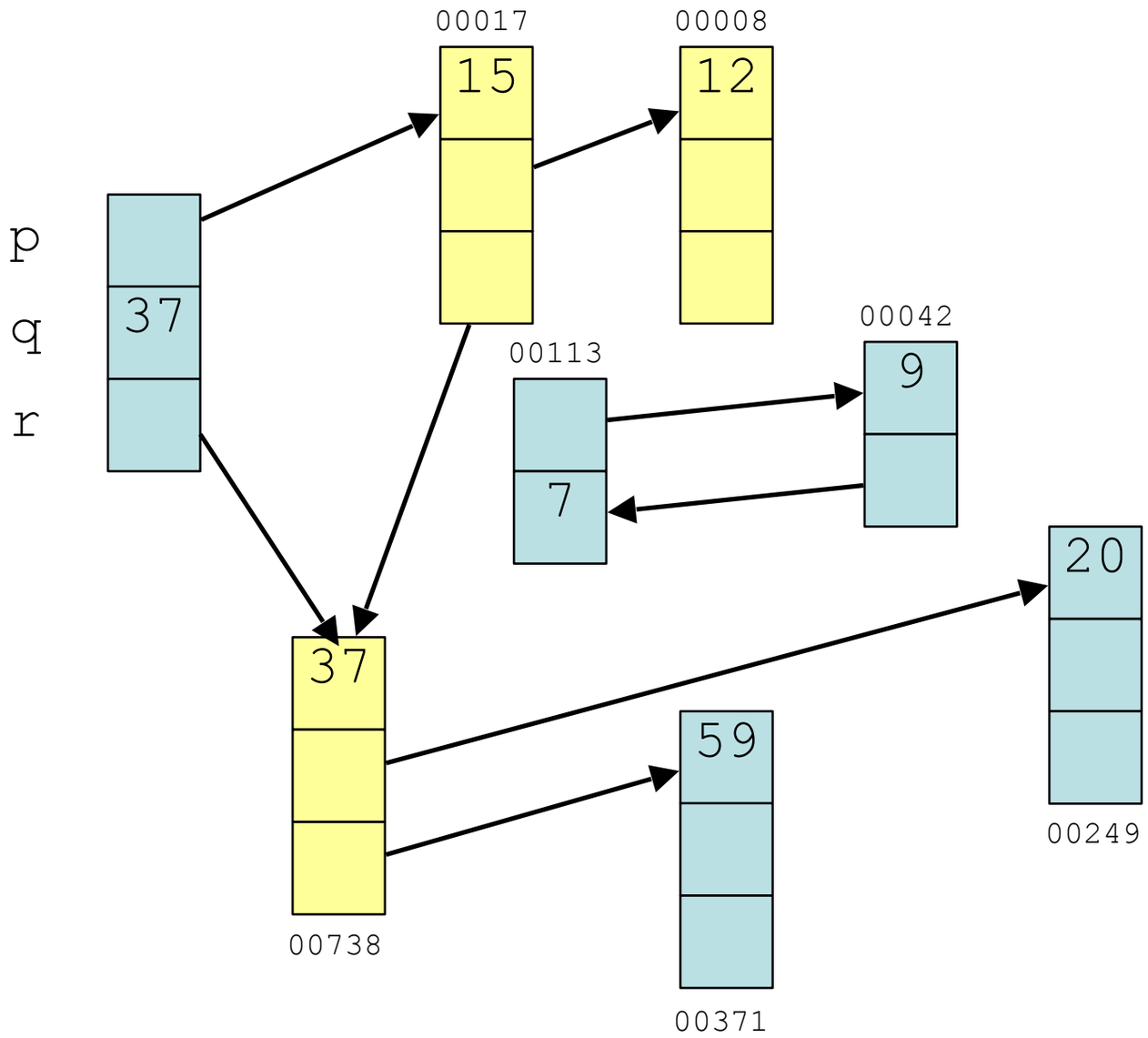


Marking and Sweeping (2/11)



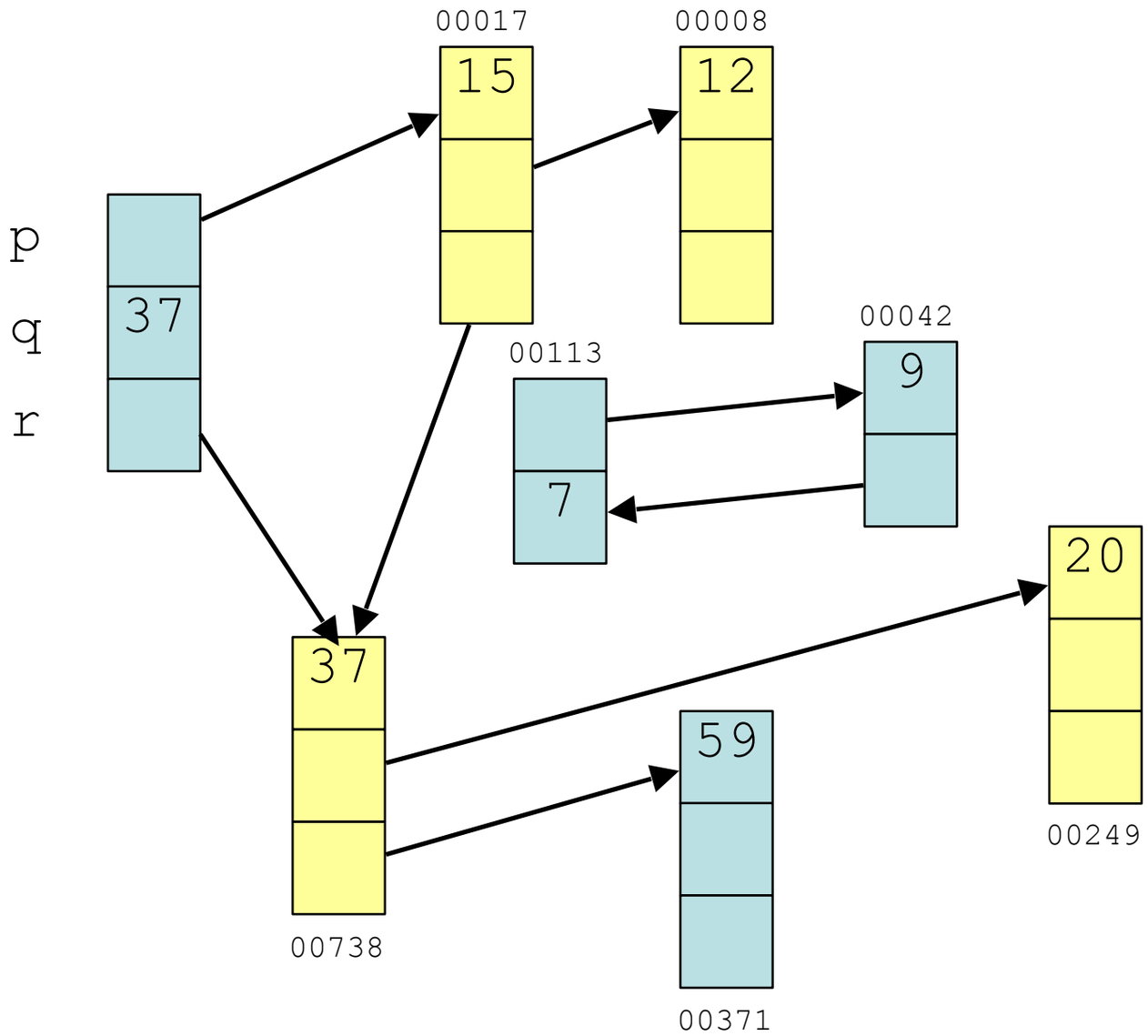
Garbage Collection

Marking and Sweeping (3/11)



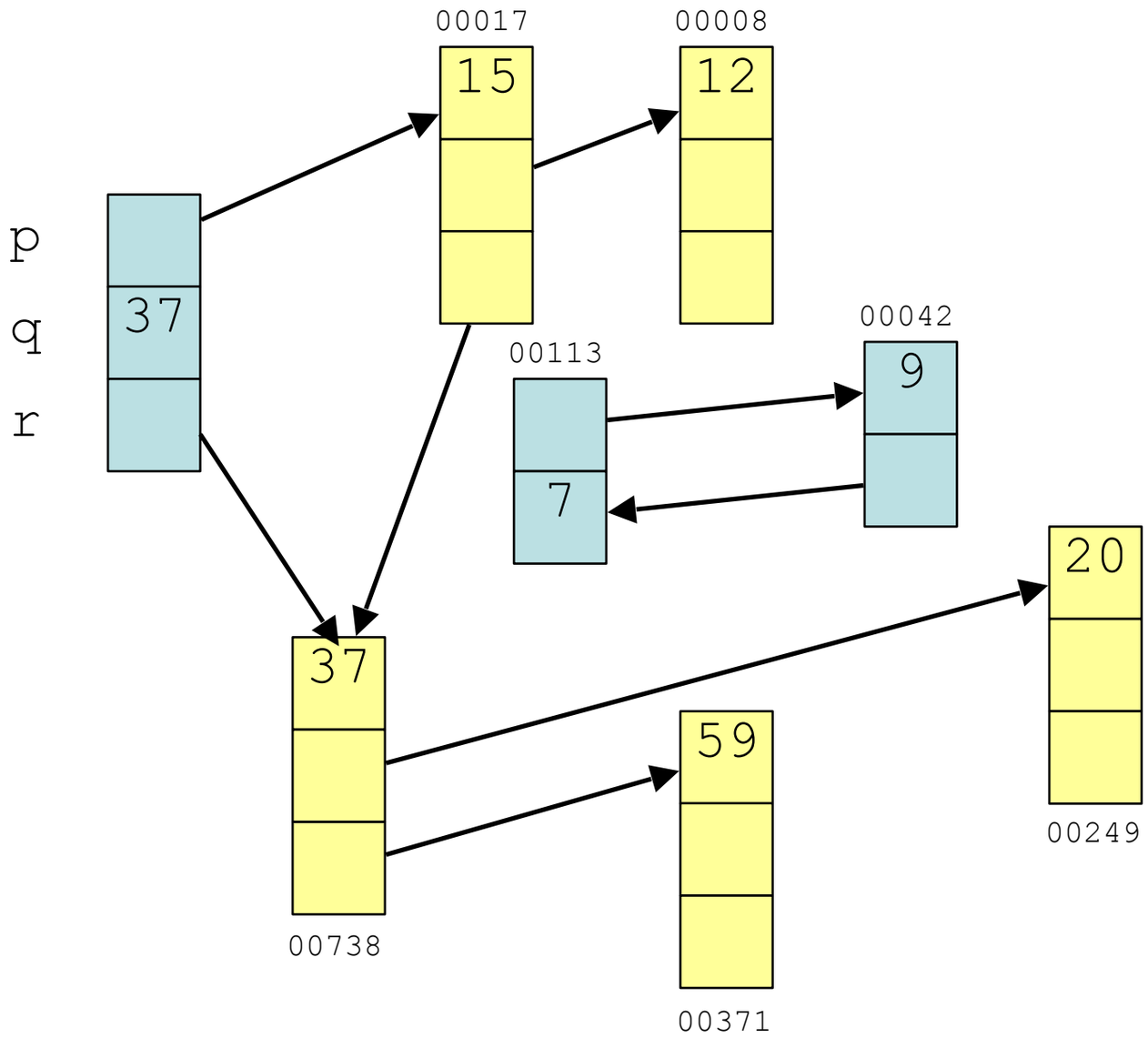
Garbage Collection

Marking and Sweeping (4/11)



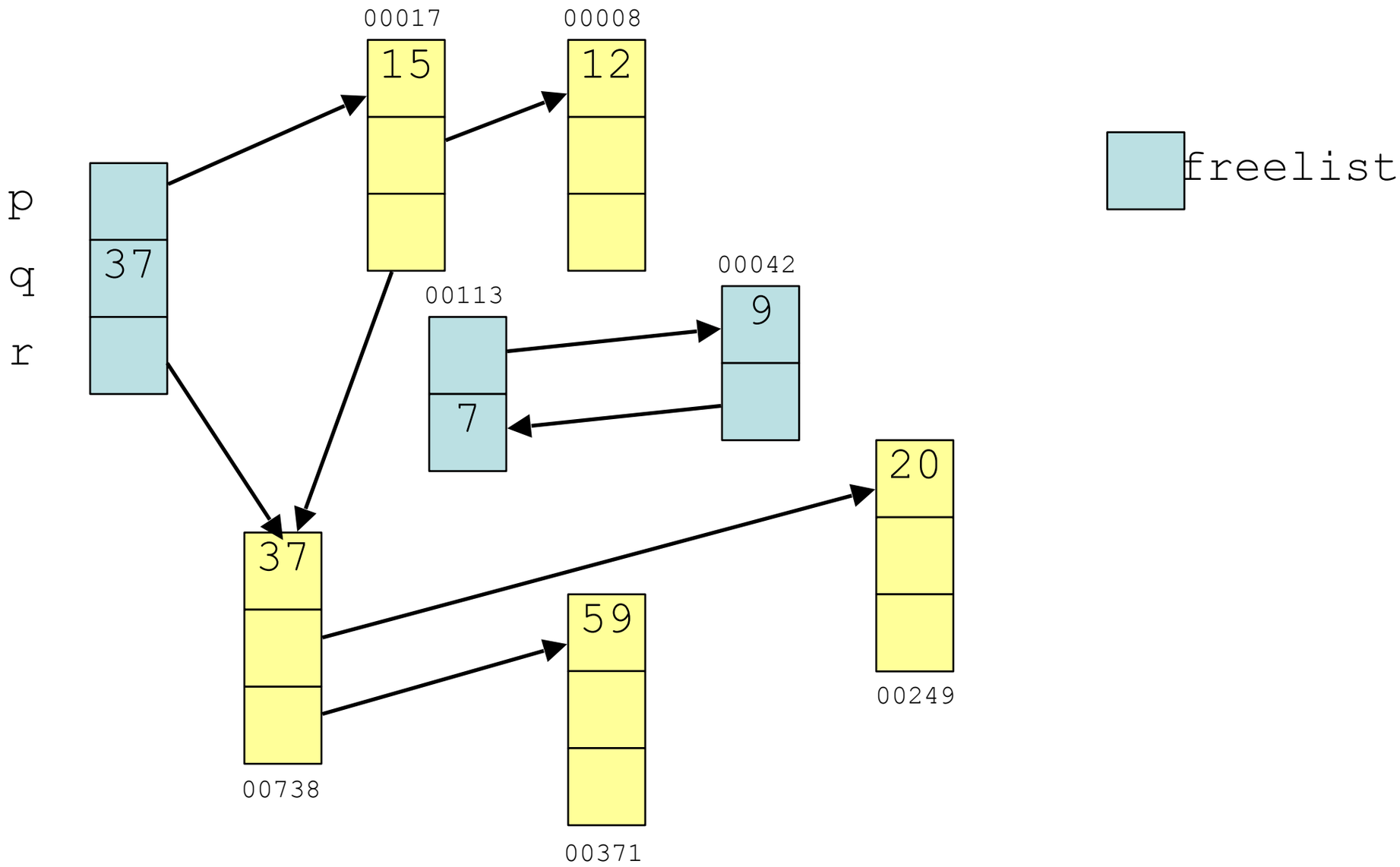
Garbage Collection

Marking and Sweeping (5/11)



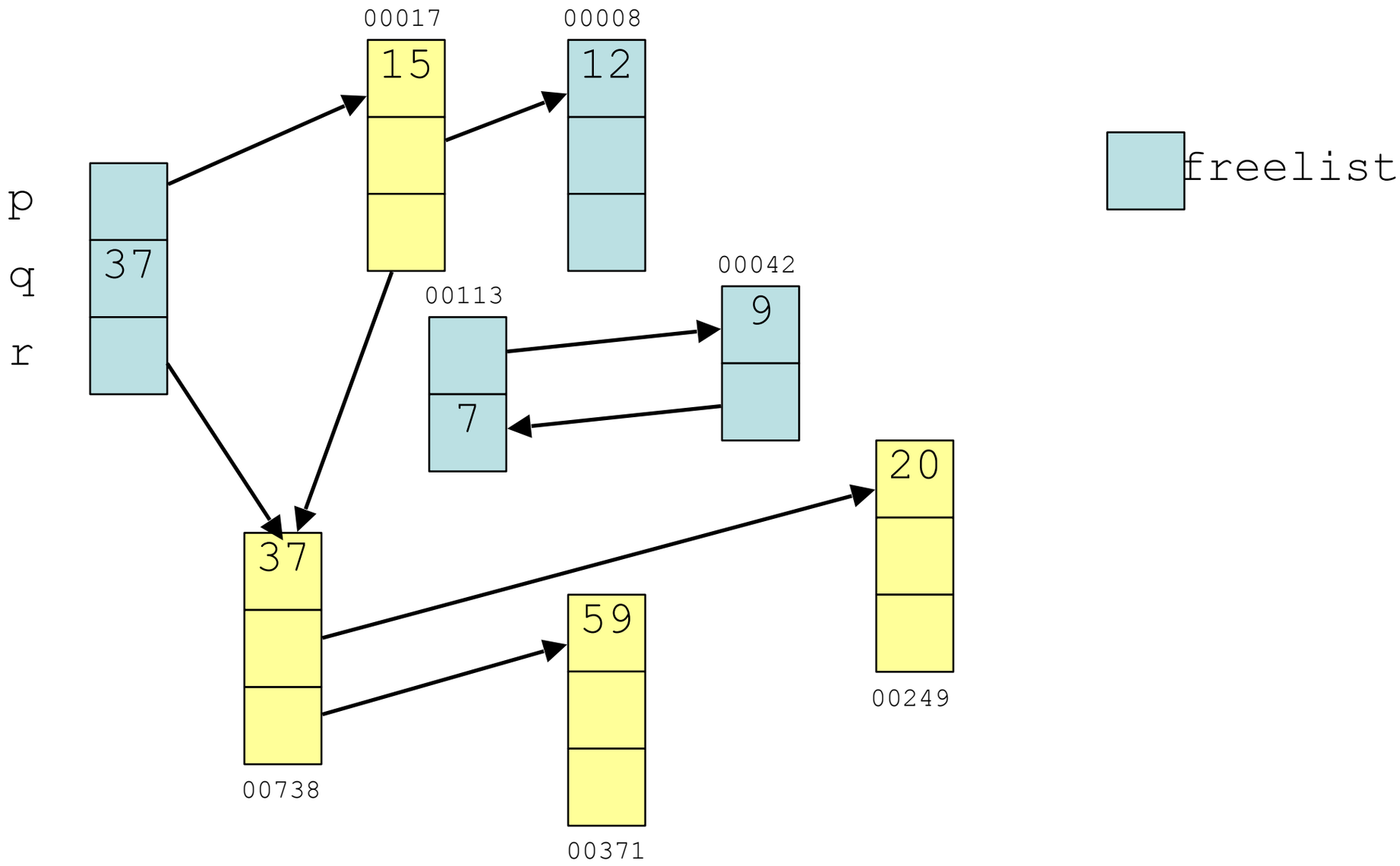
Garbage Collection

Marking and Sweeping (6/11)



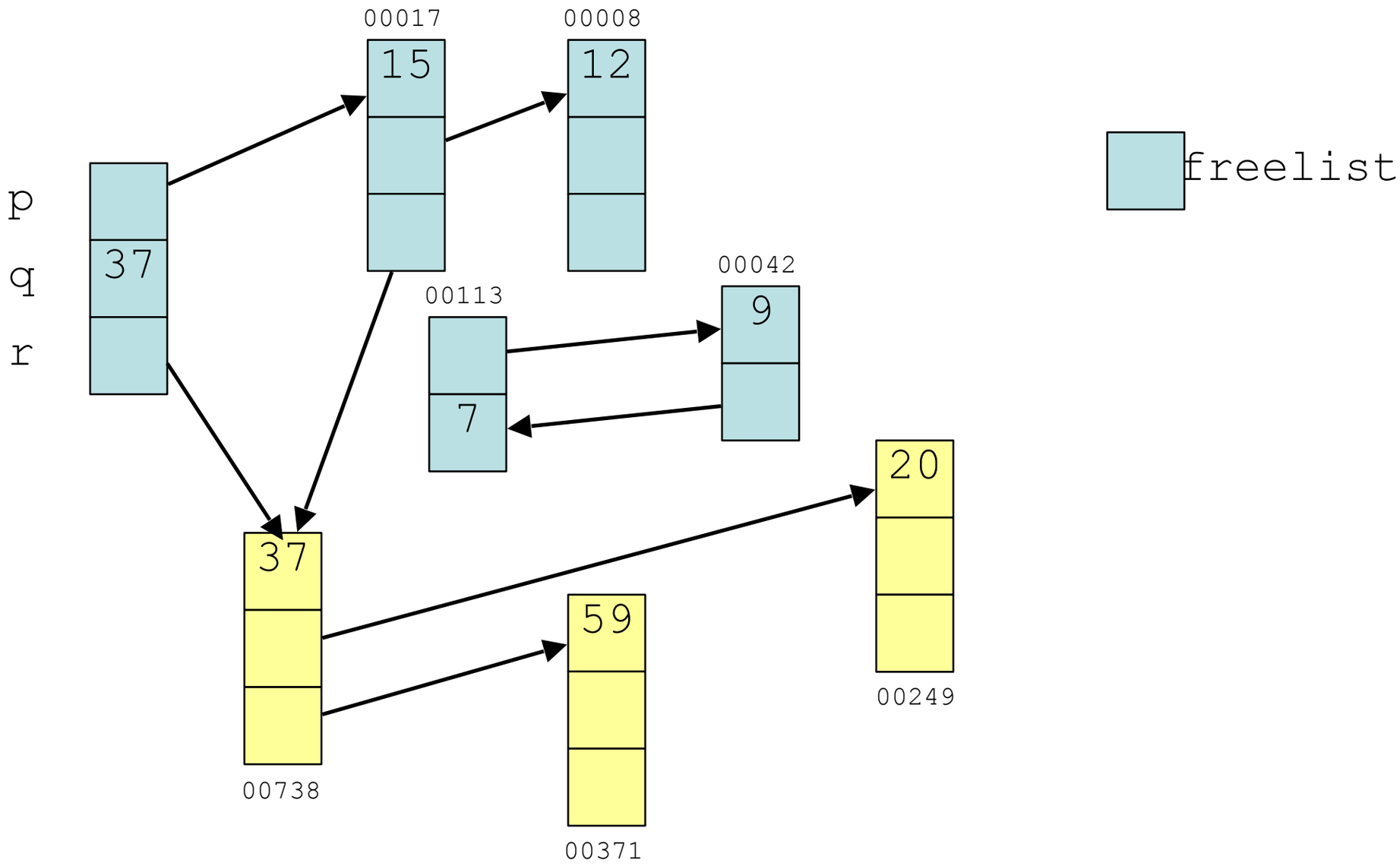
Garbage Collection

Marking and Sweeping (6/11)



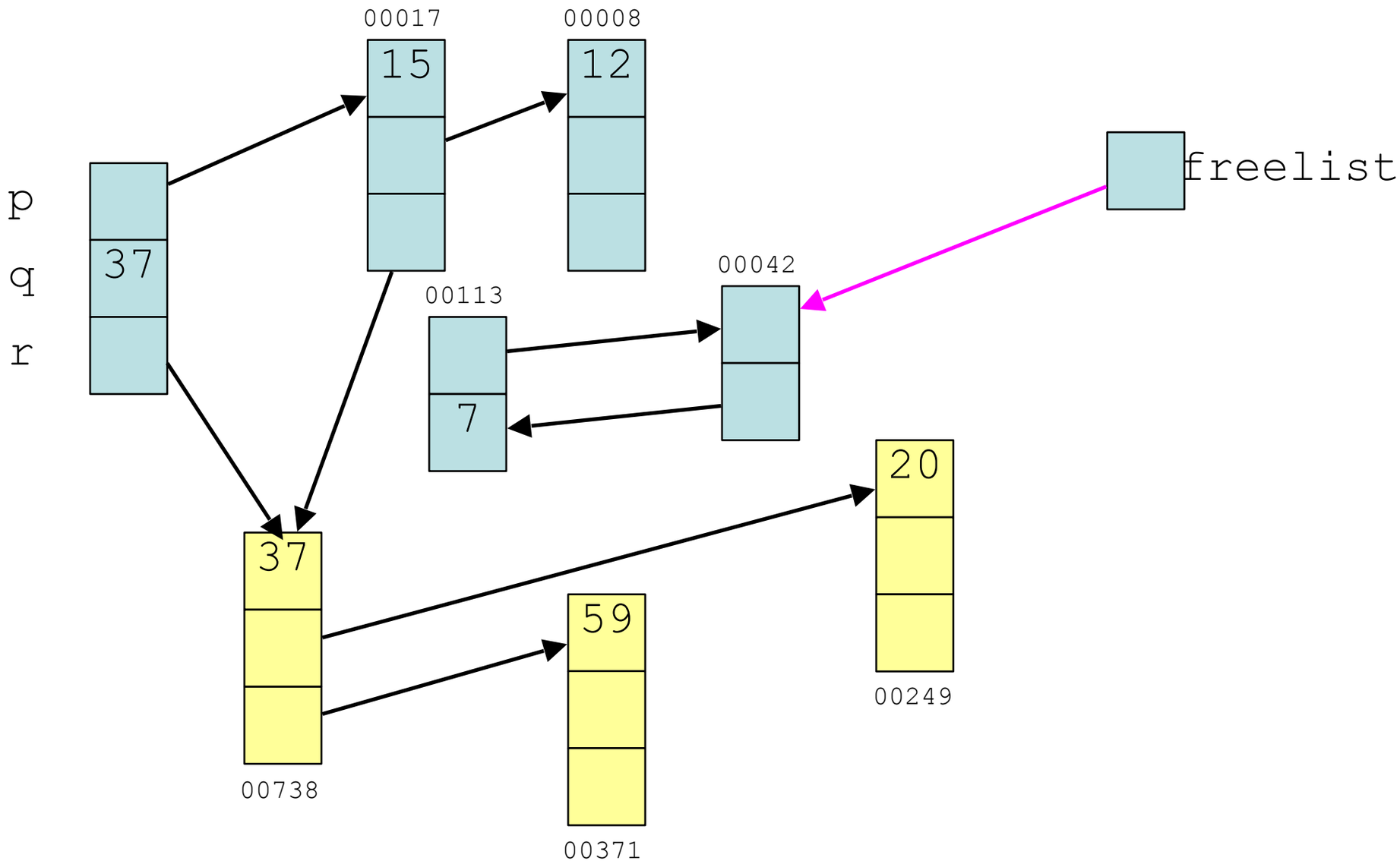
Garbage Collection

Marking and Sweeping (6/11)



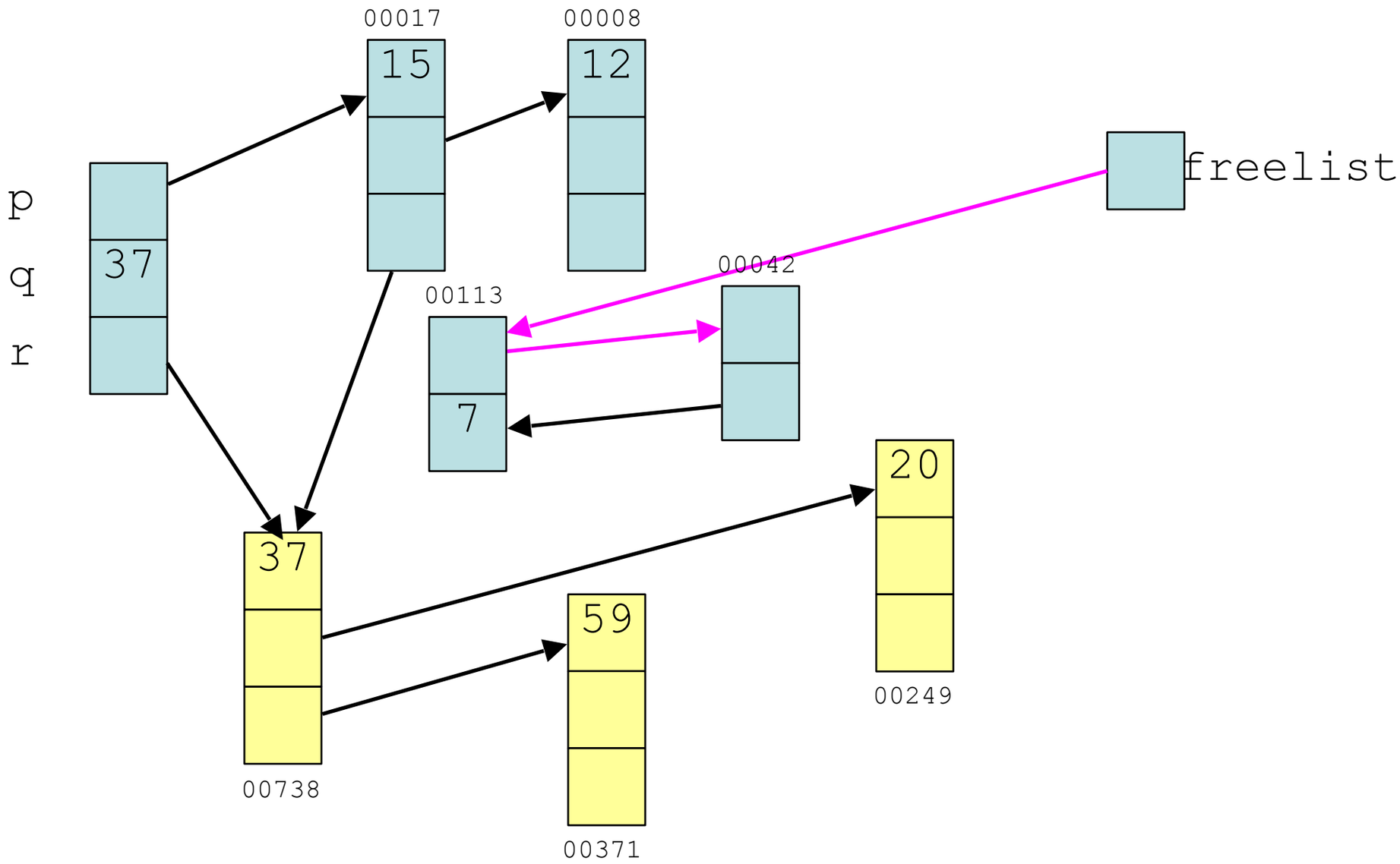
Garbage Collection

Marking and Sweeping (7/11)



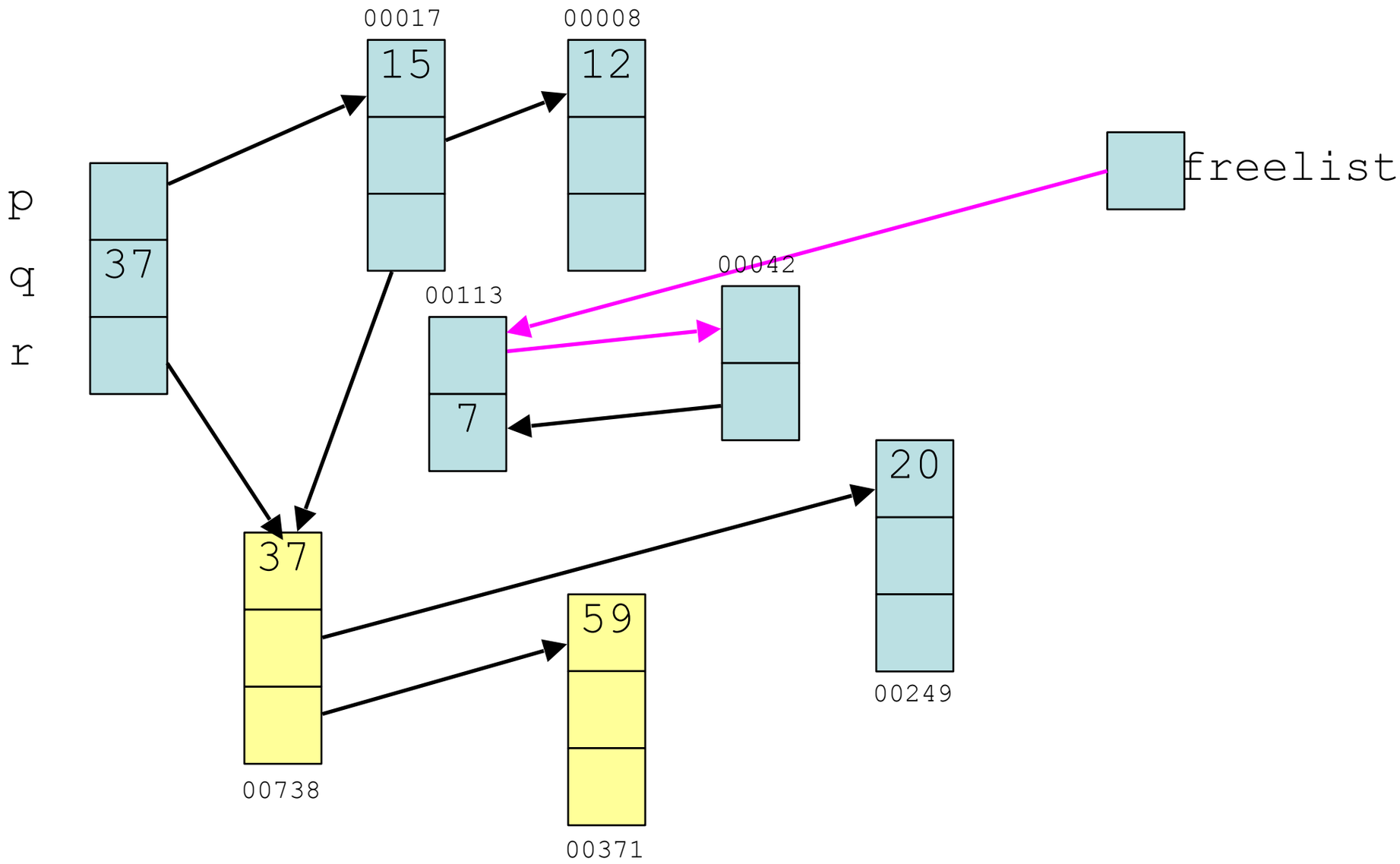
Garbage Collection

Marking and Sweeping (8/11)



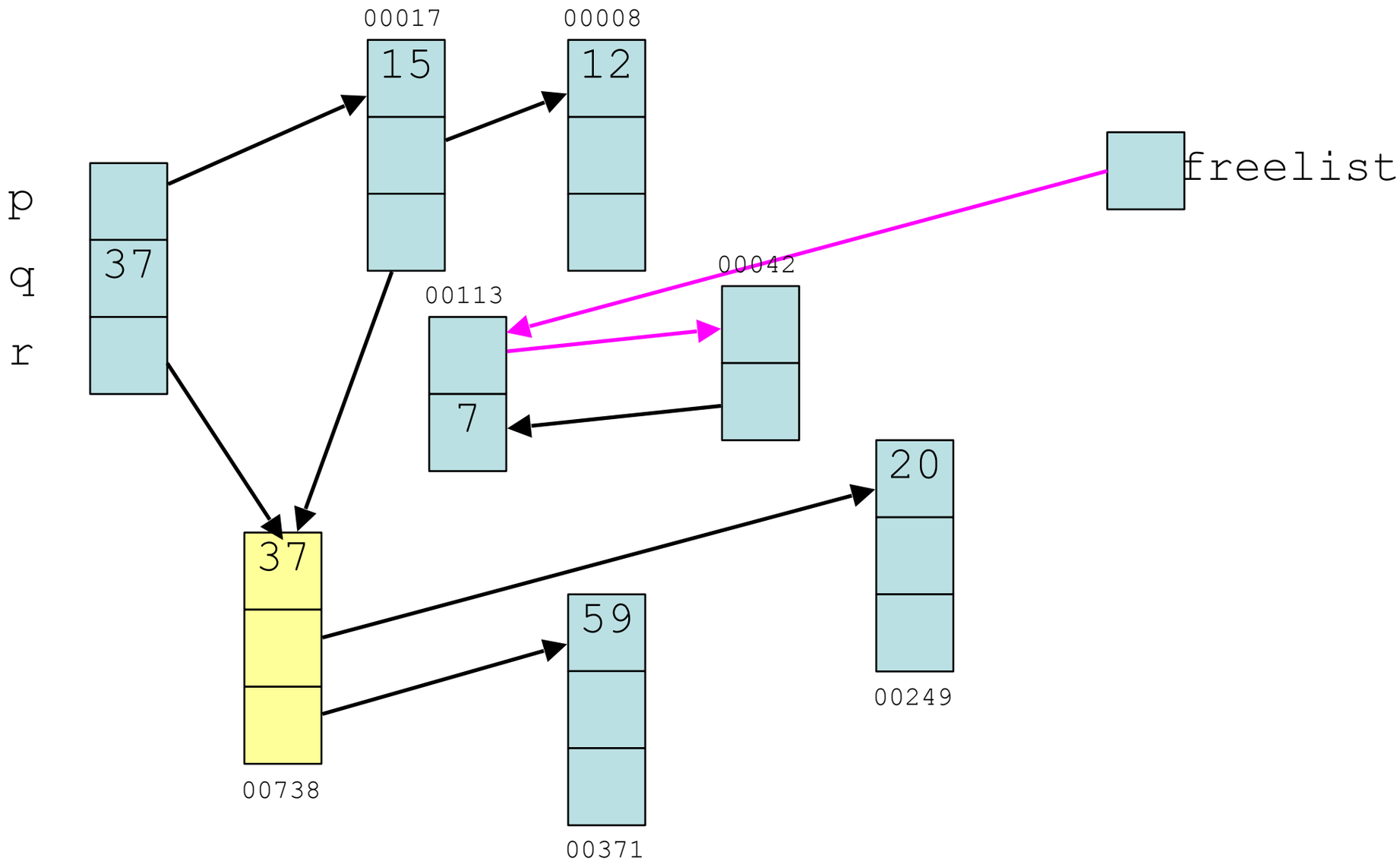
Garbage Collection

Marking and Sweeping (9/11)



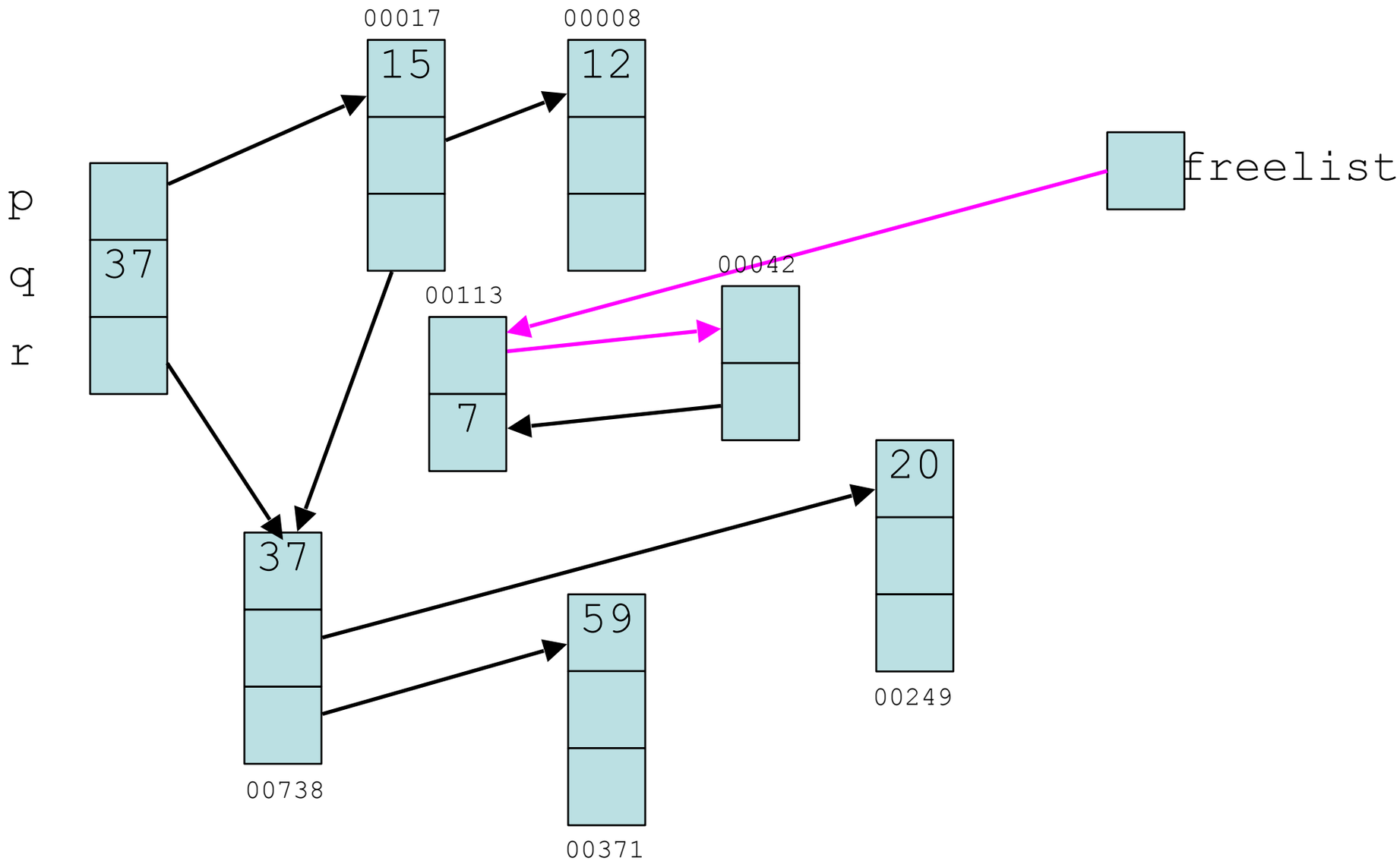
Garbage Collection

Marking and Sweeping (10/11)



Garbage Collection

Marking and Sweeping (11/11)



Garbage Collection

Analysis of mark-and-sweep

- Assume that heap has H words
- Assume that R words are reachable
- The cost of garbage collection is

$$c_1R + c_2H$$

- Utility of the collector is (H - R) of usable memory
- The cost per reclaimed word is

$$(c_1R + c_2H)/(H - R)$$

  - When R is close to H, this is expensive

Pointer reversal

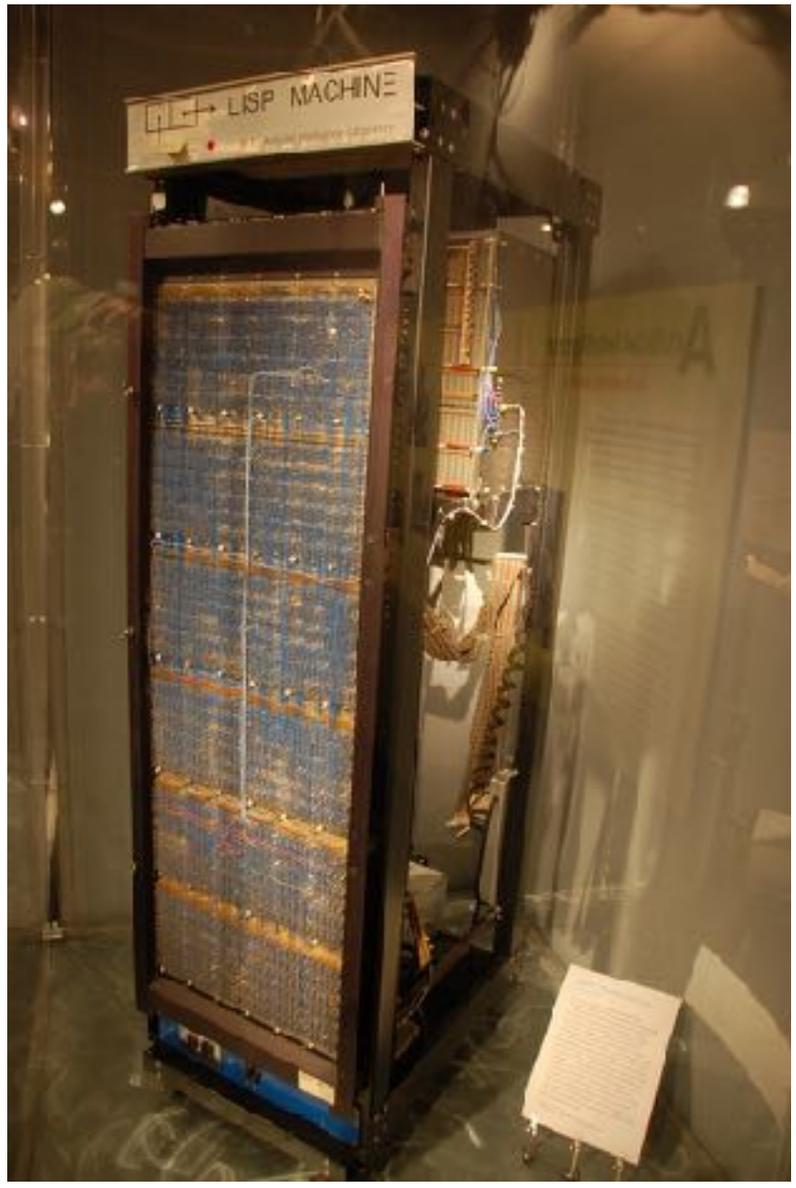
- DFS recursion stack could have size H
  - it has at least size *log* (H)
- This may be too much (after all, we're low on memory)

- The recursion stack may be *cleverly* embedded in the fields of the marked records
- This technique makes mark-and-sweep practical

Copy collectors

A historic detour: Lisp machines

"… the early MIT Lisp Machines in fact did not implement a garbage collector for quite some years; or rather, even when the garbage collector appeared, users preferred to disable it. Most of the programming tools (notably the compiler and program text editor) were designed to avoid consing and to explicitly reclaim temporary data structures whenever possible; given this, the Lisp Machine address spaces were large enough, and the virtual memory system good enough, that a user could run for several days or even a few weeks before having to save out the running "world" to disk and restart it. **Such copying back and forth to disk was equivalent to a slow,** manually triggered copying garbage collector."

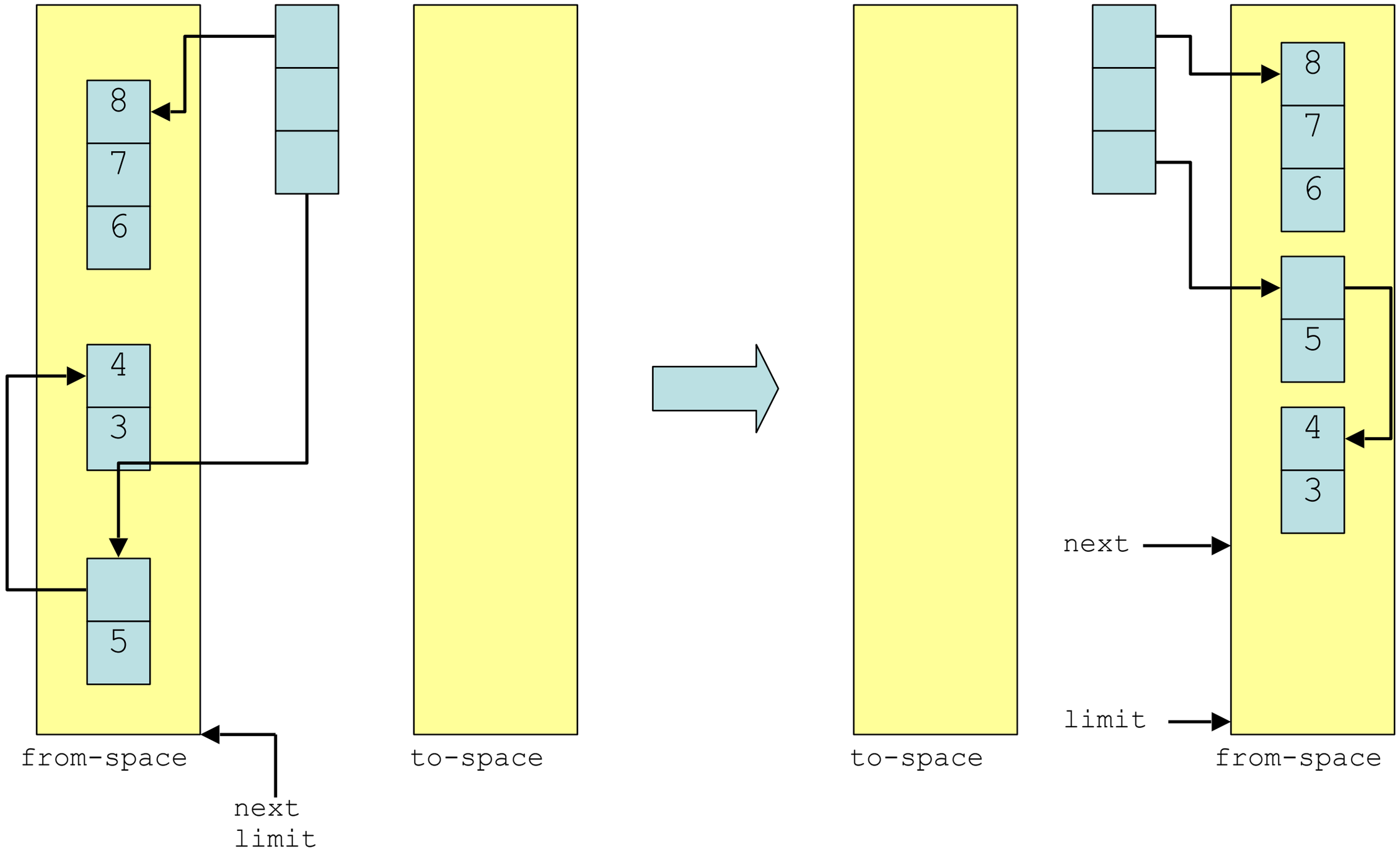


Gabriel and Steele, Evolution of Lisp 2

Stop-and-Copy algorithm

- Divide the heap space into *two parts*
- Use *only one* part at a time
- When it runs full, *copy live* records to the other part of the heap space
- Then switch the roles of the two parts
- Advantages
  - fast allocation: (no freelist)
  - avoids fragmentation
- Disadvantage:
  - wastes half your memory

Before and After Stop-and-Copy

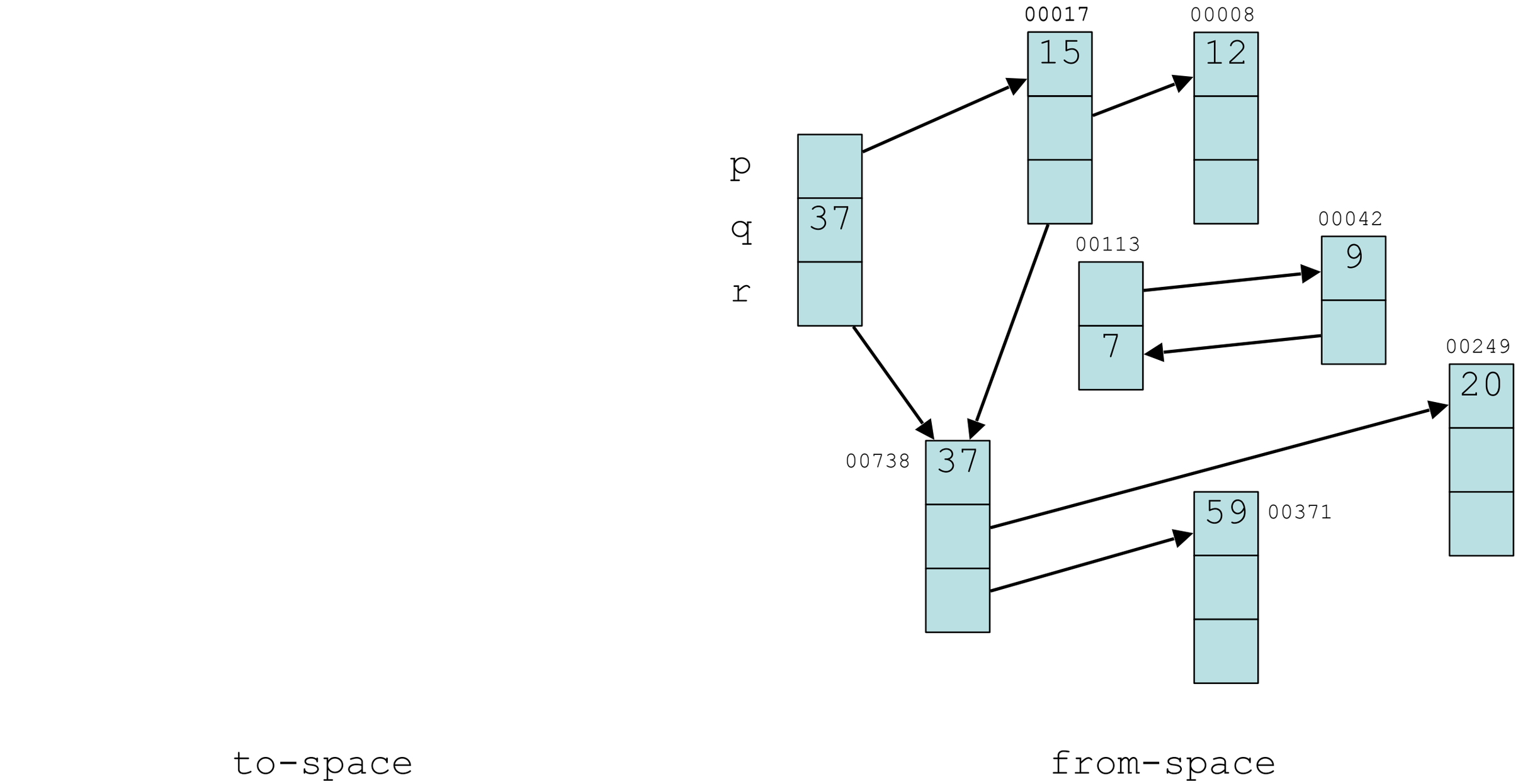


Pseudo Code for Stop-and-Copy

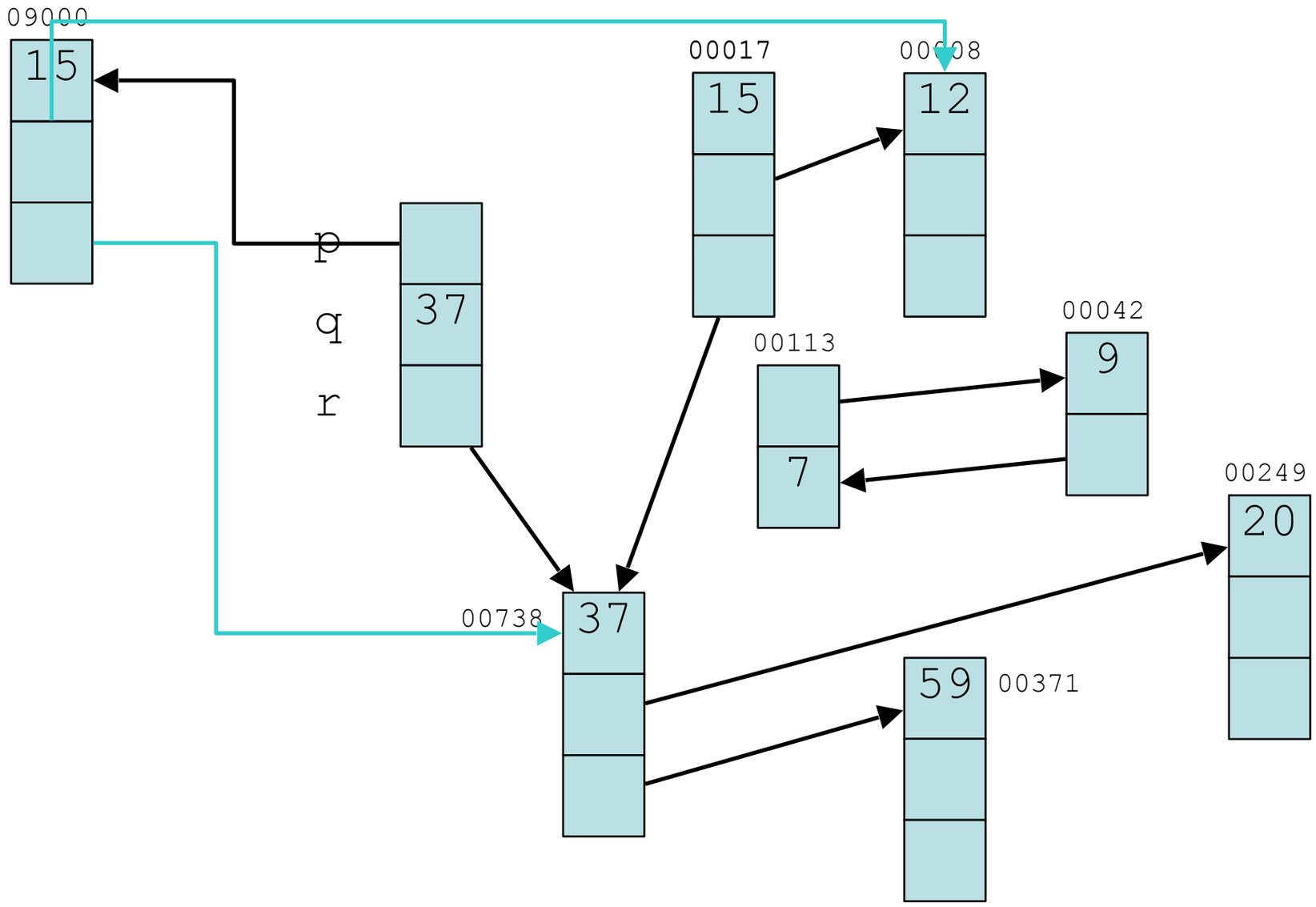
```
function Forward(x) {
  if (x ∈ from-space) {
    if (x.f1 ∈ to-space)
      return x.f1;
    else
      for (i=1; i<|x|; i++)
        next.fi = x.fi;
      x.f1 = next;
      next = next + sizeof(x);
      return x.f1;
  } else return x;
}
```

```
function Copy() {
  scan = next = start of to-space;
  foreach (v in a stack frame)
    v = Forward(v);
  while (scan < next) {
    for (i=1; i<=|scan|; i++)
      scan.fi = Forward(scan.fi);
    scan = scan + sizeof(scan);
  }
}
```

Stopping and Copying (1/13)



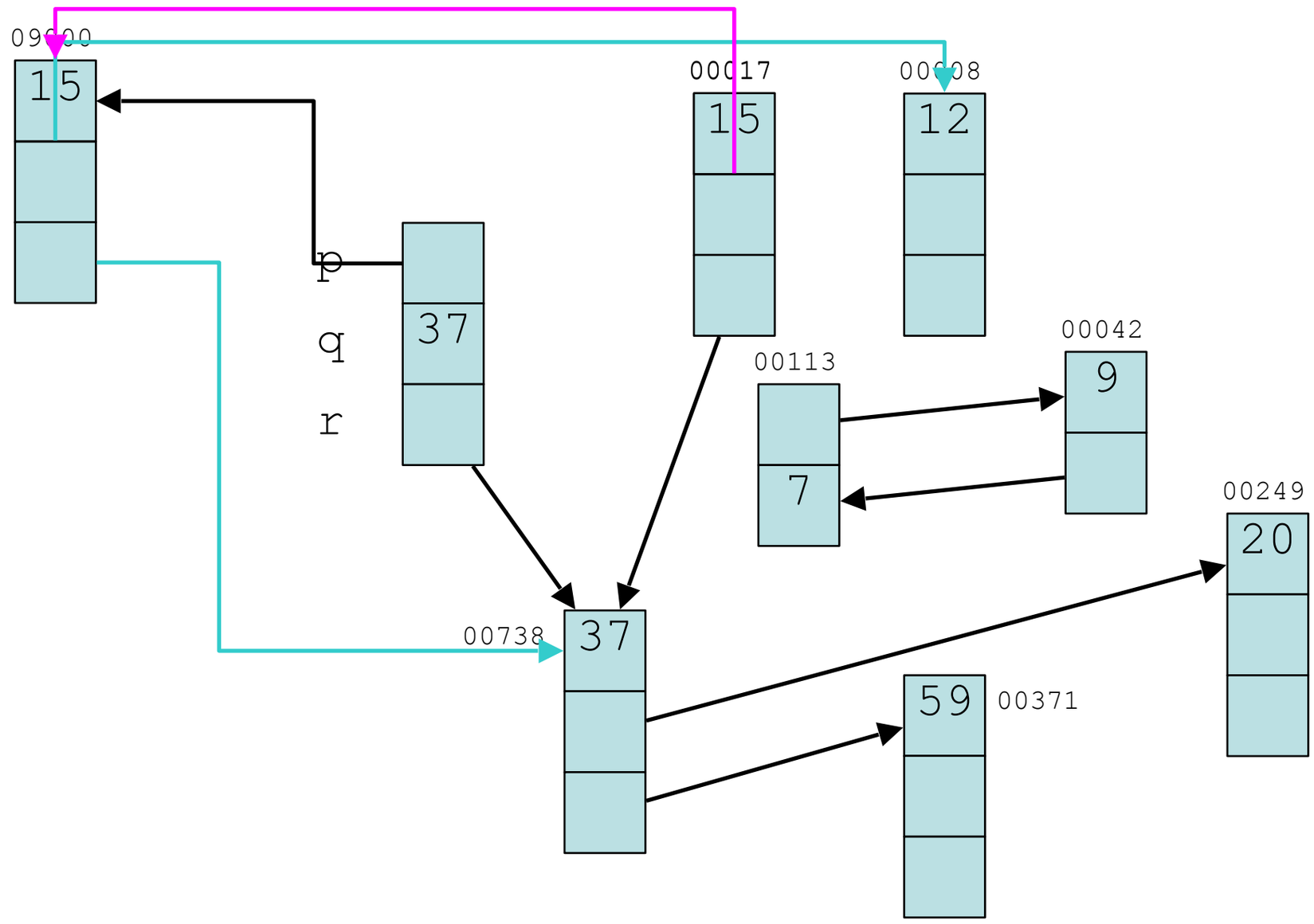
Stopping and Copying (2/13)



to-space

from-space

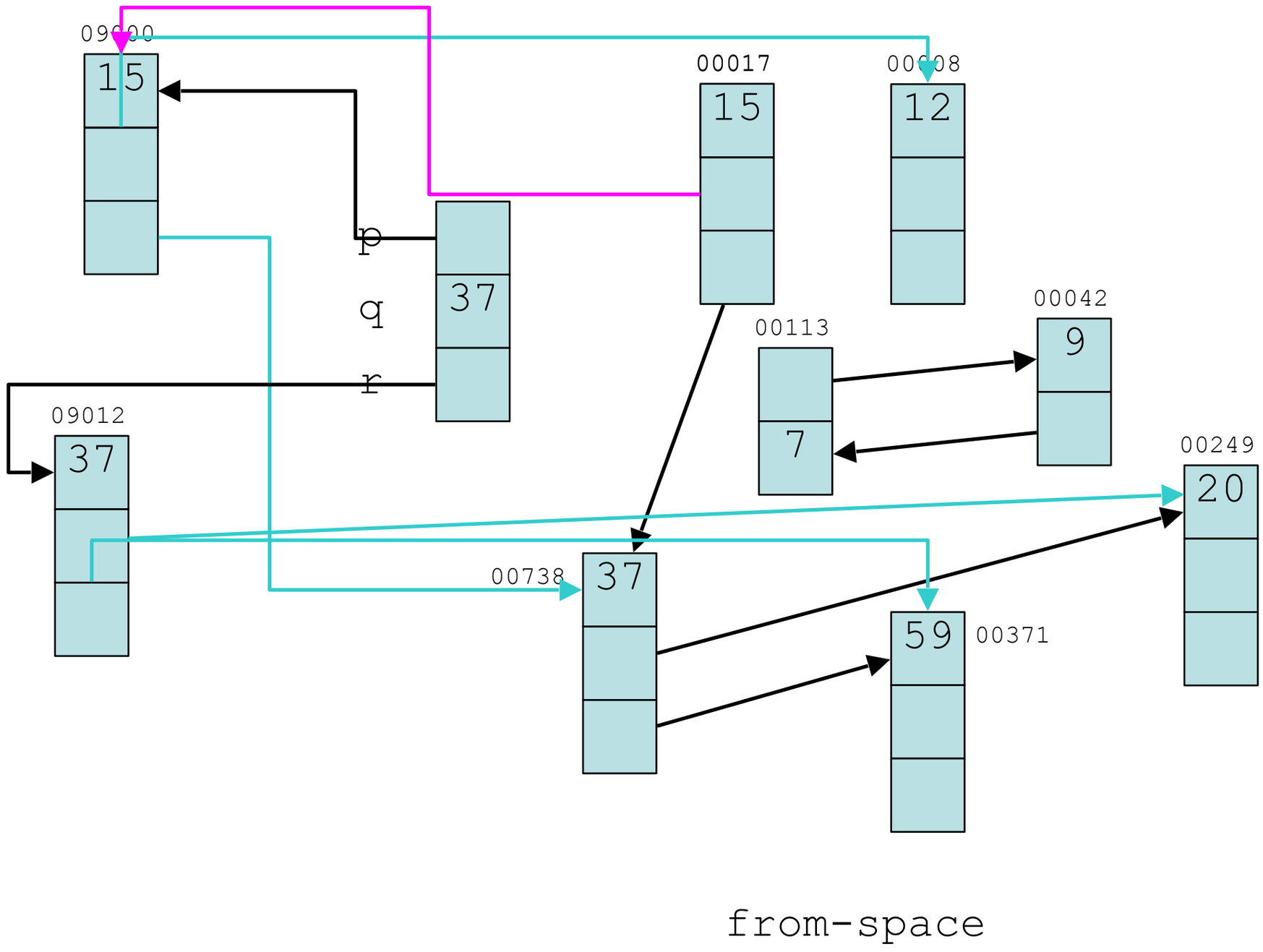
Stopping and Copying (3/13)



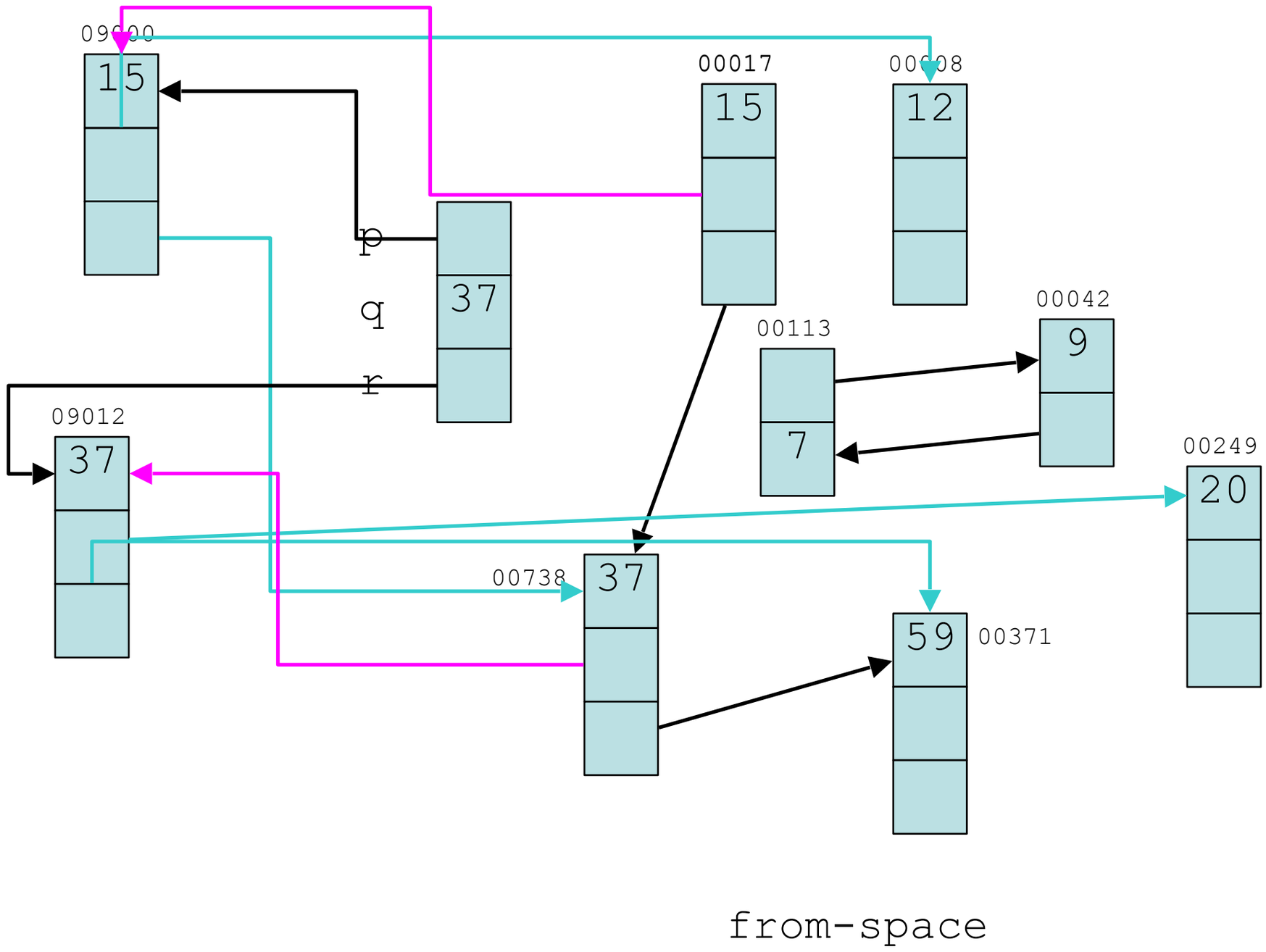
to-space

from-space

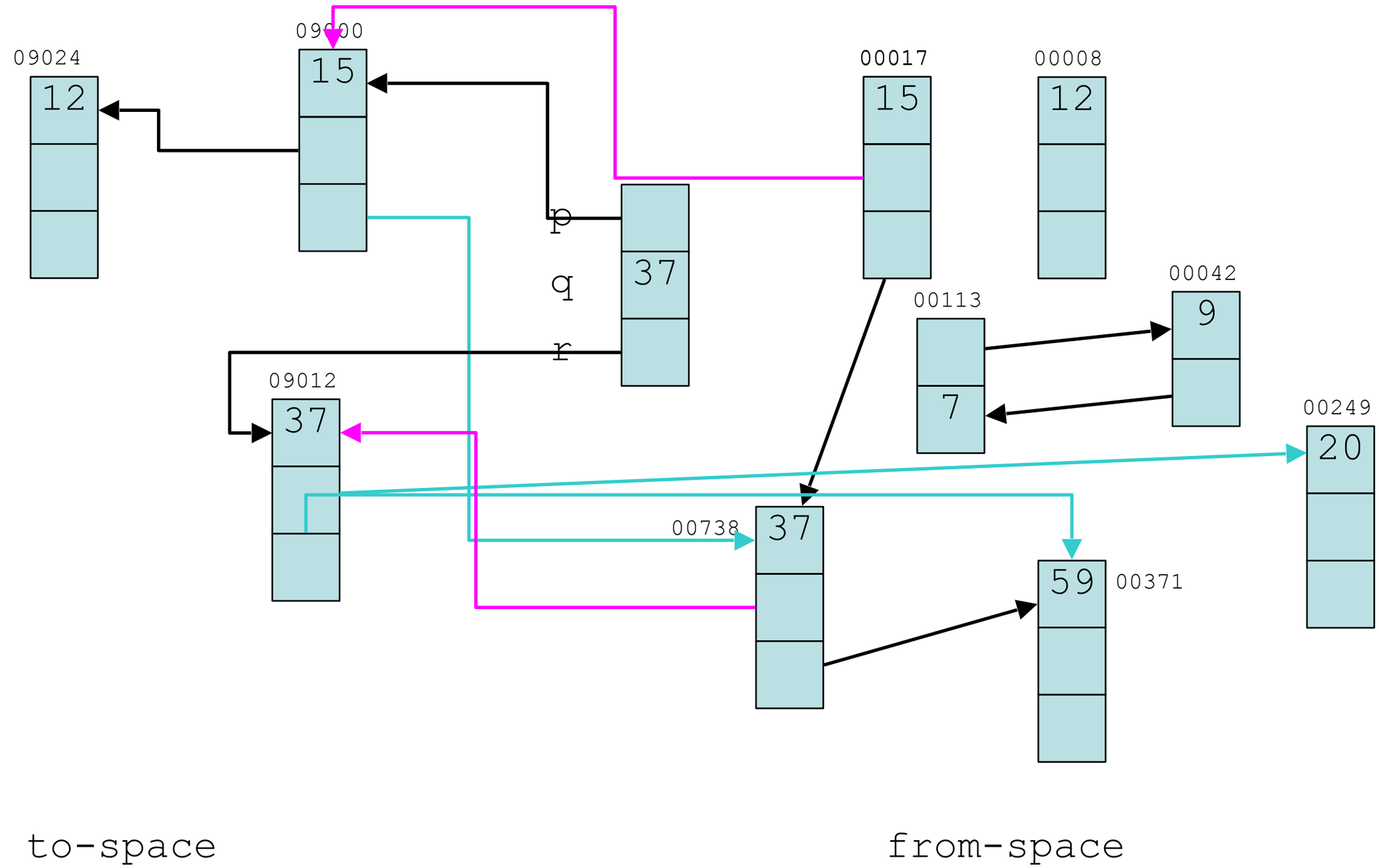
Stopping and Copying (4/13)



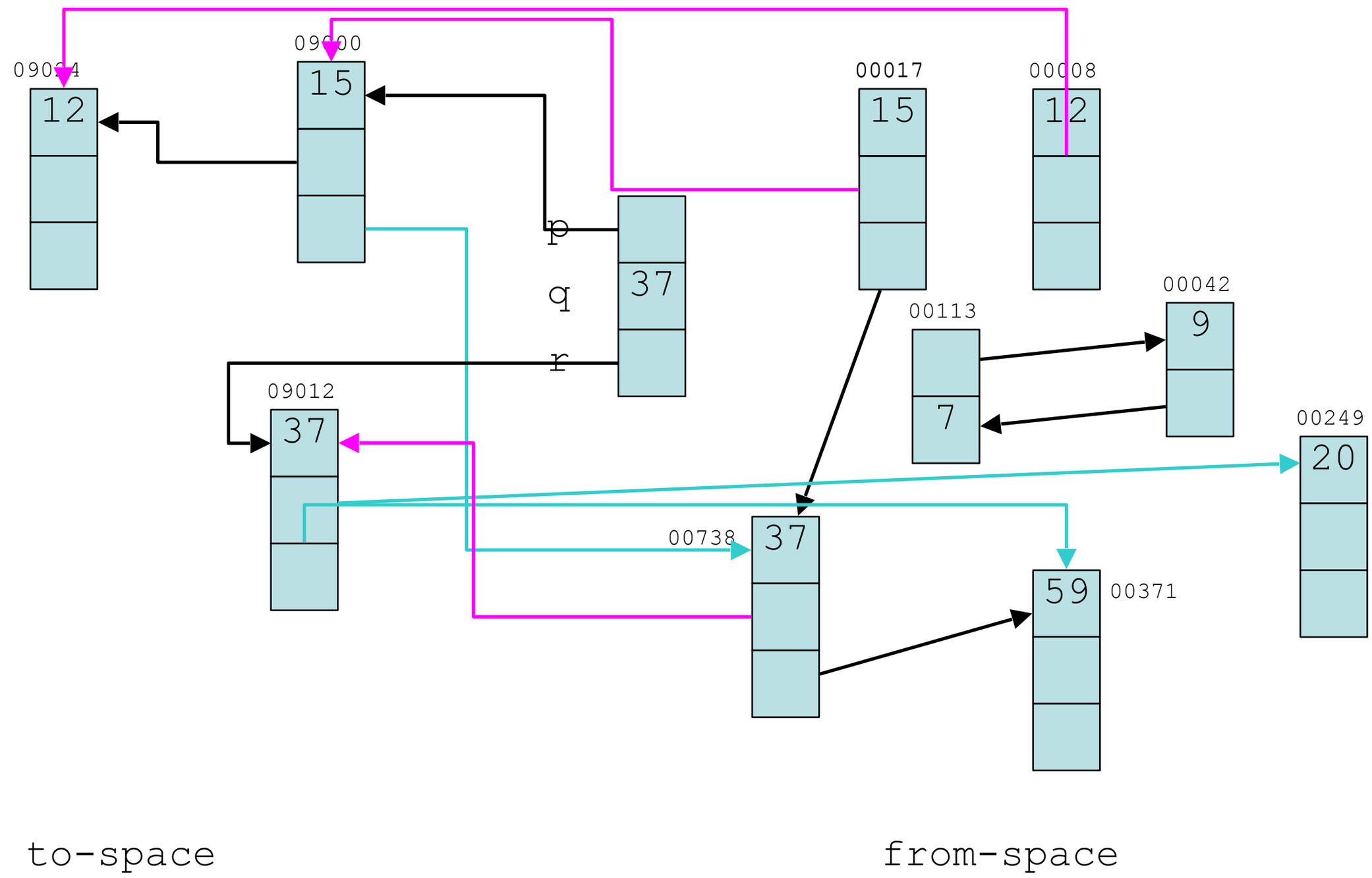
Stopping and Copying (5/13)



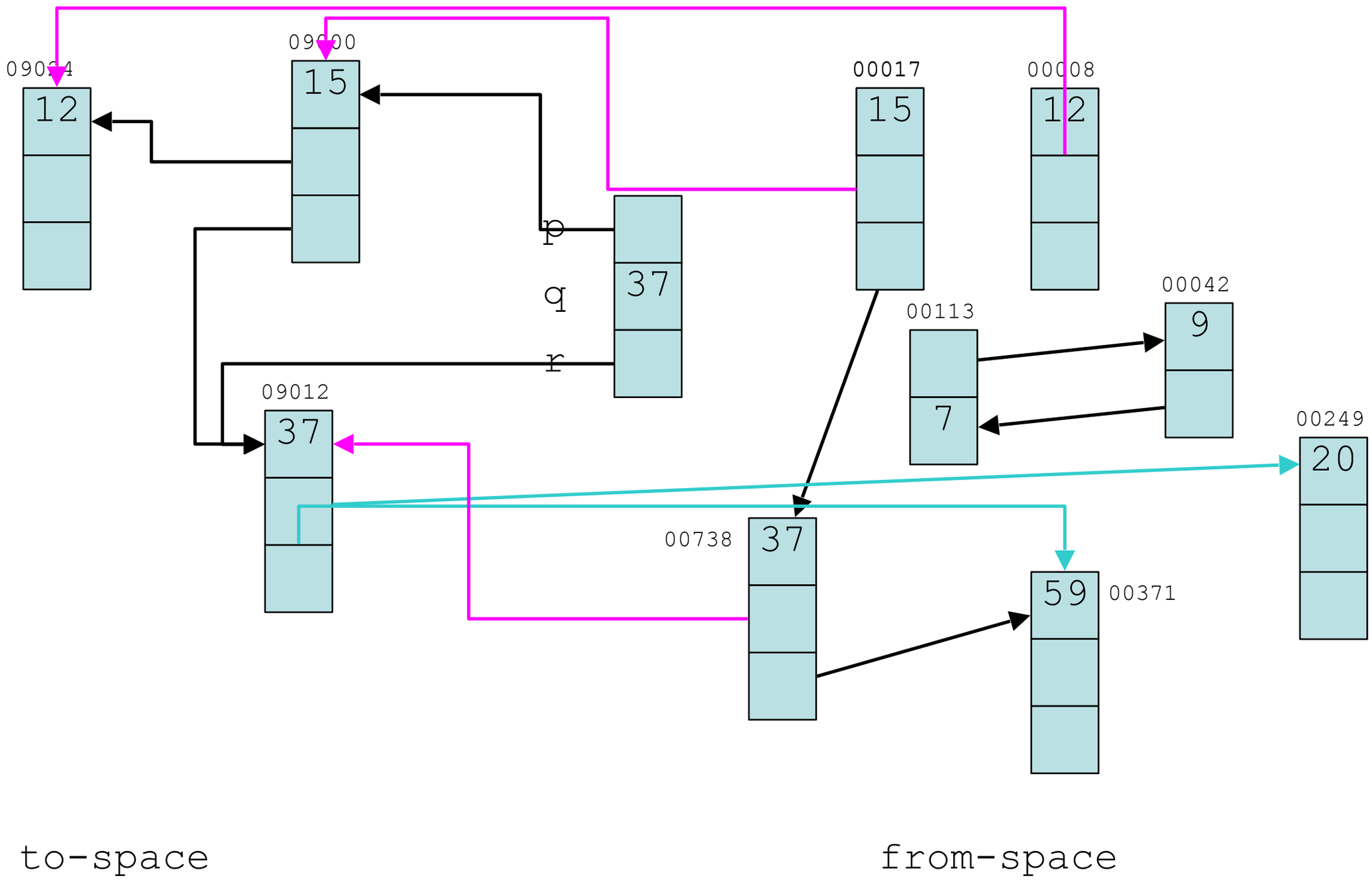
Stopping and Copying (6/13)



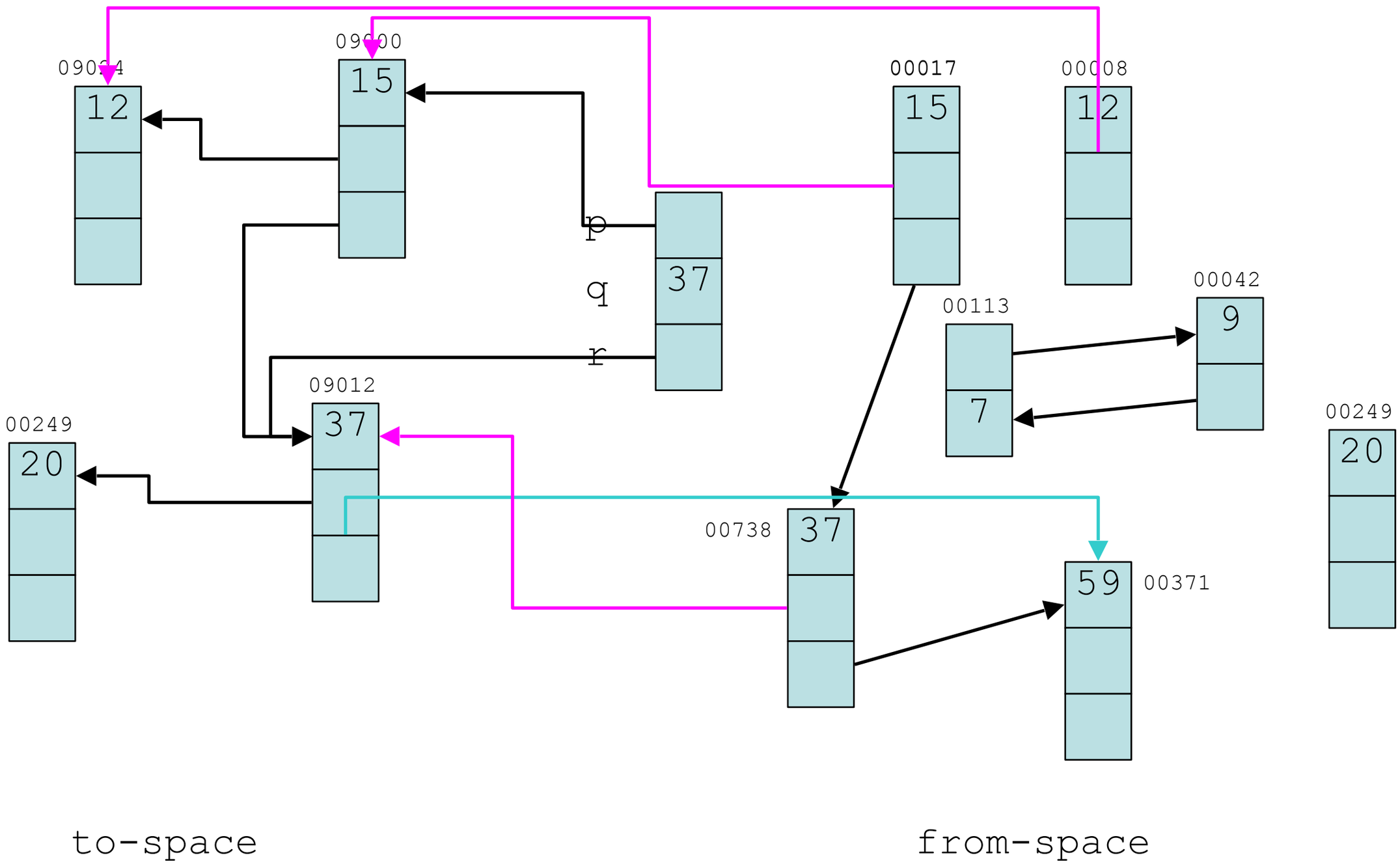
Stopping and Copying (7/13)



Stopping and Copying (8/13)

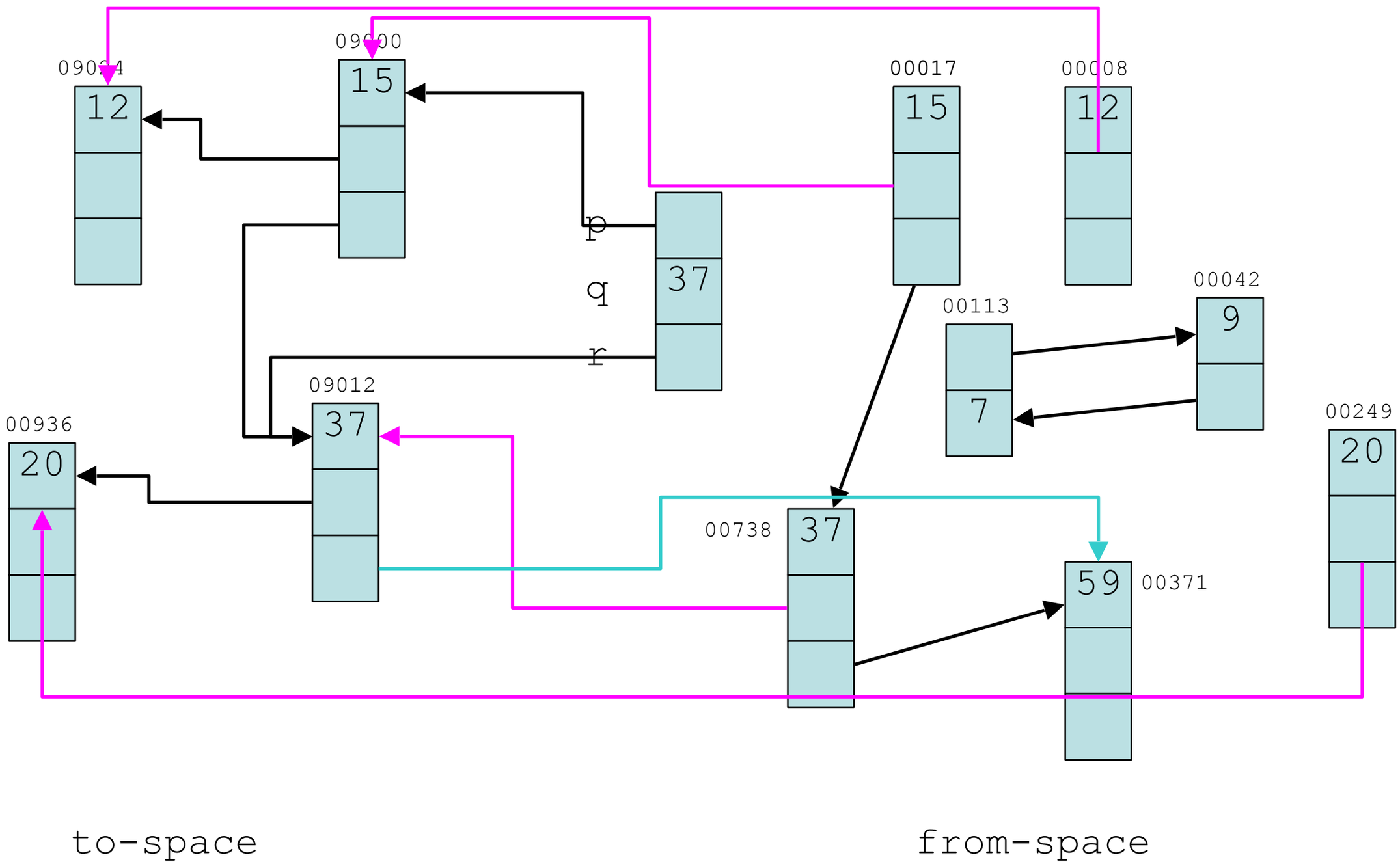


Stopping and Copying (9/13)

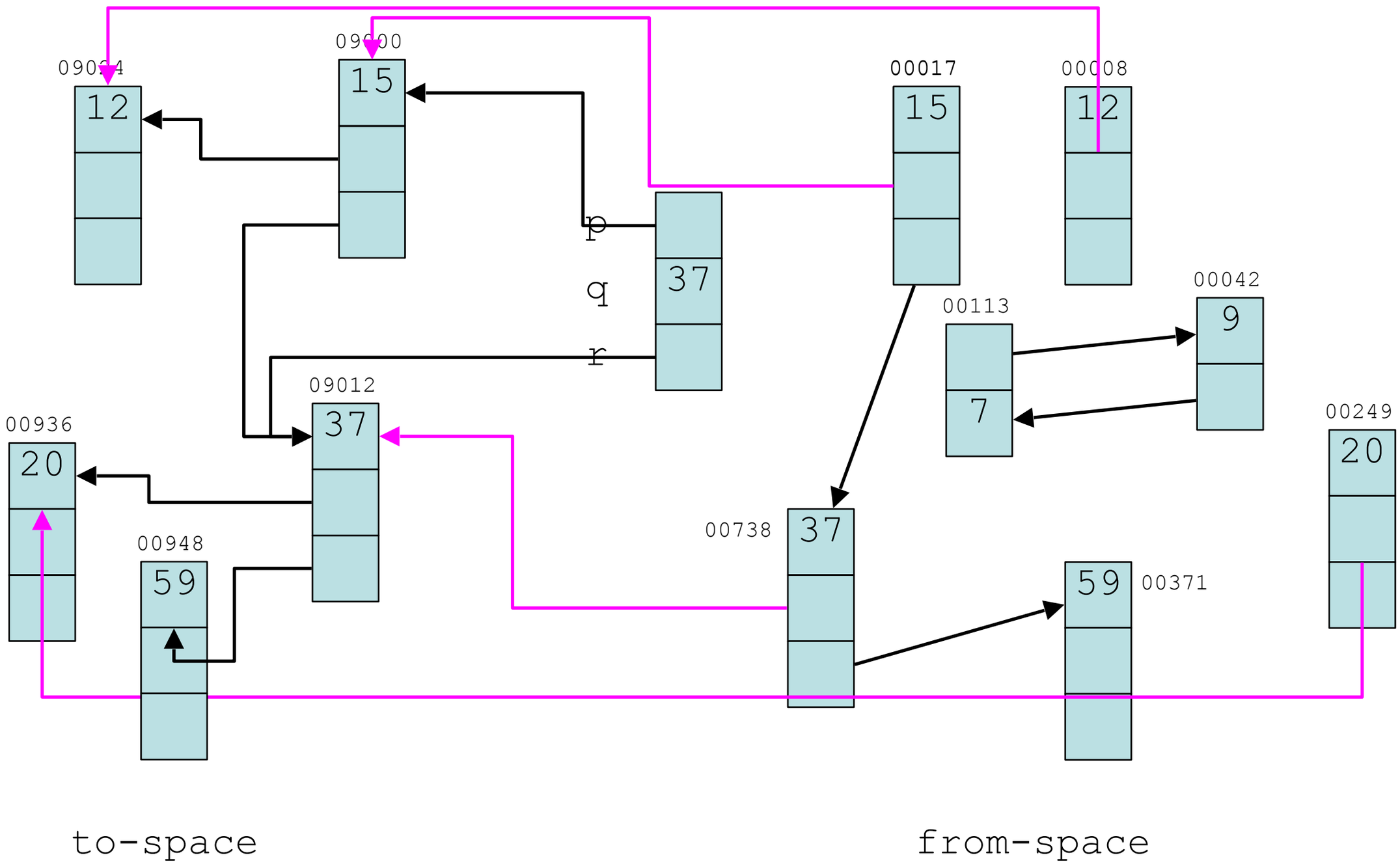


Garbage Collection

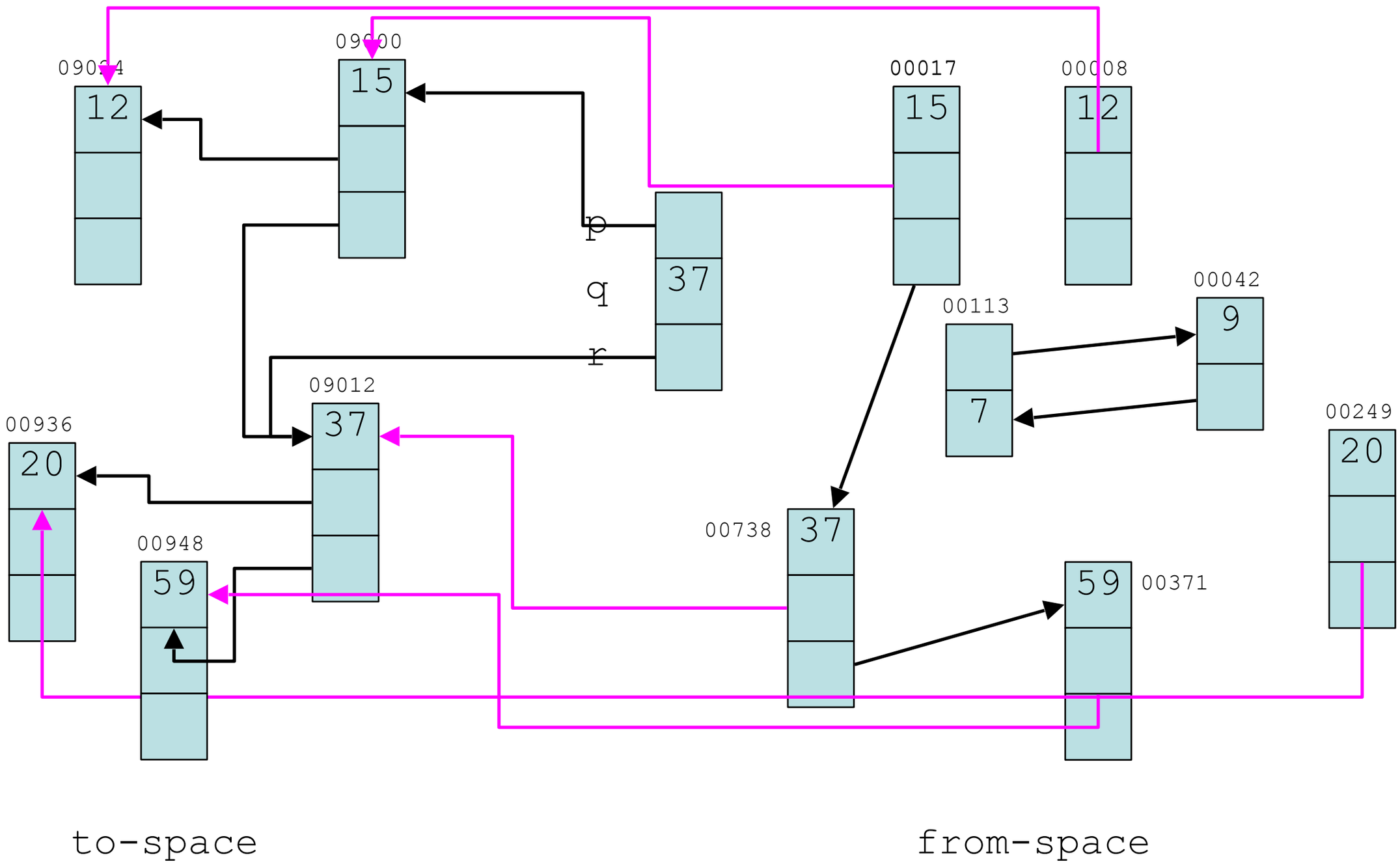
Stopping and Copying (10/13)



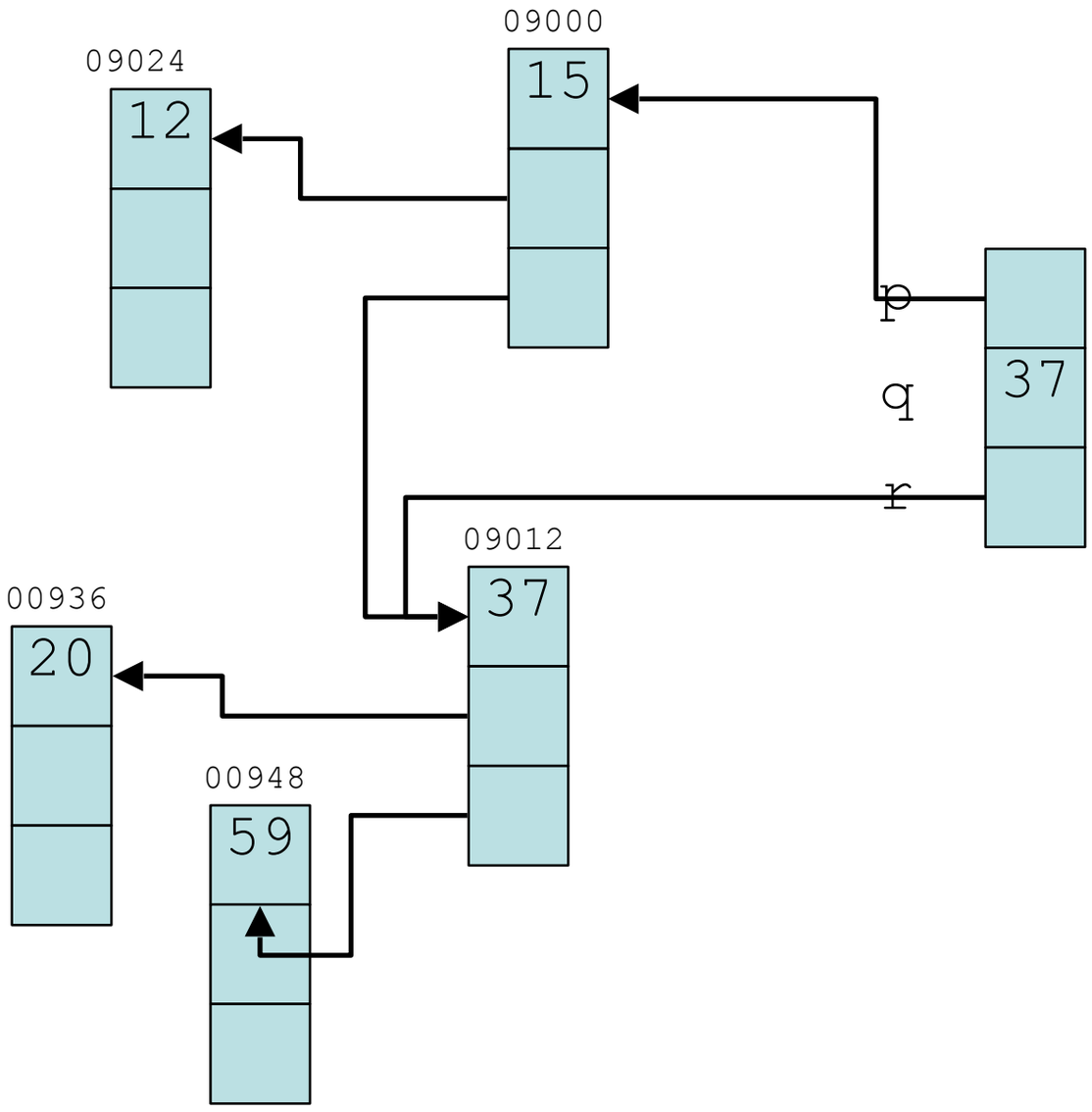
Stopping and Copying (11/13)



Stopping and Copying (12/13)



Stopping and Copying (13/13)



from-space

to-space

Analysis of Stop-and-Copy

- Assume the heap has H words
- Assume that R words are reachable
- The cost of garbage collection is: c_3R
- The cost per reclaimed word is: $c_3R/(H/2 - R)$
- This has no lower bound as H grows
- (Mark-and-sweep: $(c_1R + c_2H)/(H - R)$ per record)

Recognizing Records and Pointers

- Earlier assumptions:
  - we know the start and size of each record in memory
  - we know which record fields are pointers
- We can store the pointer information in the pointer itself (tagging) or within each record:
  - For object-oriented languages, each record already contains a pointer to a class descriptor
  - For general languages, we must sacrifice a few bytes per record
  - For the stack frame:
    - use a bit per stack location
    - use a table per program point

Generational Collection

- Observation: the young die quickly!
- The collector should focus on young records
- Divide the heap into generations: G_0, G_1, G_2, ...
- All records in G_i are younger than records in G_{i+1}
- Collect G_0 often, G_1 less often, and so on
- Promote a record from G_i to G_{i+1} when it survives several collections

Collecting a Generation

- How to collect the G_0 generation:
  - roots are no longer just stack locations, but also pointers from G_1, G_2, ...
  - it could be expensive to find those pointers
    - fortunately they are rare, so we can remember them
- Ways to remember pointers:
  - maintain a set of all updated records
  - mark pages of memory that contain updated records (using hardware or software)

Summary

- Generating and collecting garbage *is* the civilized choice
- Concept: live entities, conservatively approximated
- Basic approach: Heap as graph, live = reachable
- The Mark-and-Sweep algorithm
- The Stop-and-Copy algorithm
- The latter can be made arbitrarily cheap ;)
- Generational collection: hybrid — die young!