

# Memory Management

**Eelco Visser**



**CS4200 | Compiler Construction | December 2, 2021**

# Garbage Collection

## Reference counting

- deallocate records with count 0

# Garbage Collection

## Reference counting

- deallocate records with count 0

## Mark & sweep

- mark reachable records
- sweep unmarked records

# Garbage Collection

## Reference counting

- deallocate records with count 0

## Mark & sweep

- mark reachable records
- sweep unmarked records

## Copying collection

- copy reachable records

# Garbage Collection

## Reference counting

- deallocate records with count 0

## Mark & sweep

- mark reachable records
- sweep unmarked records

## Copying collection

- copy reachable records

## Generational collection

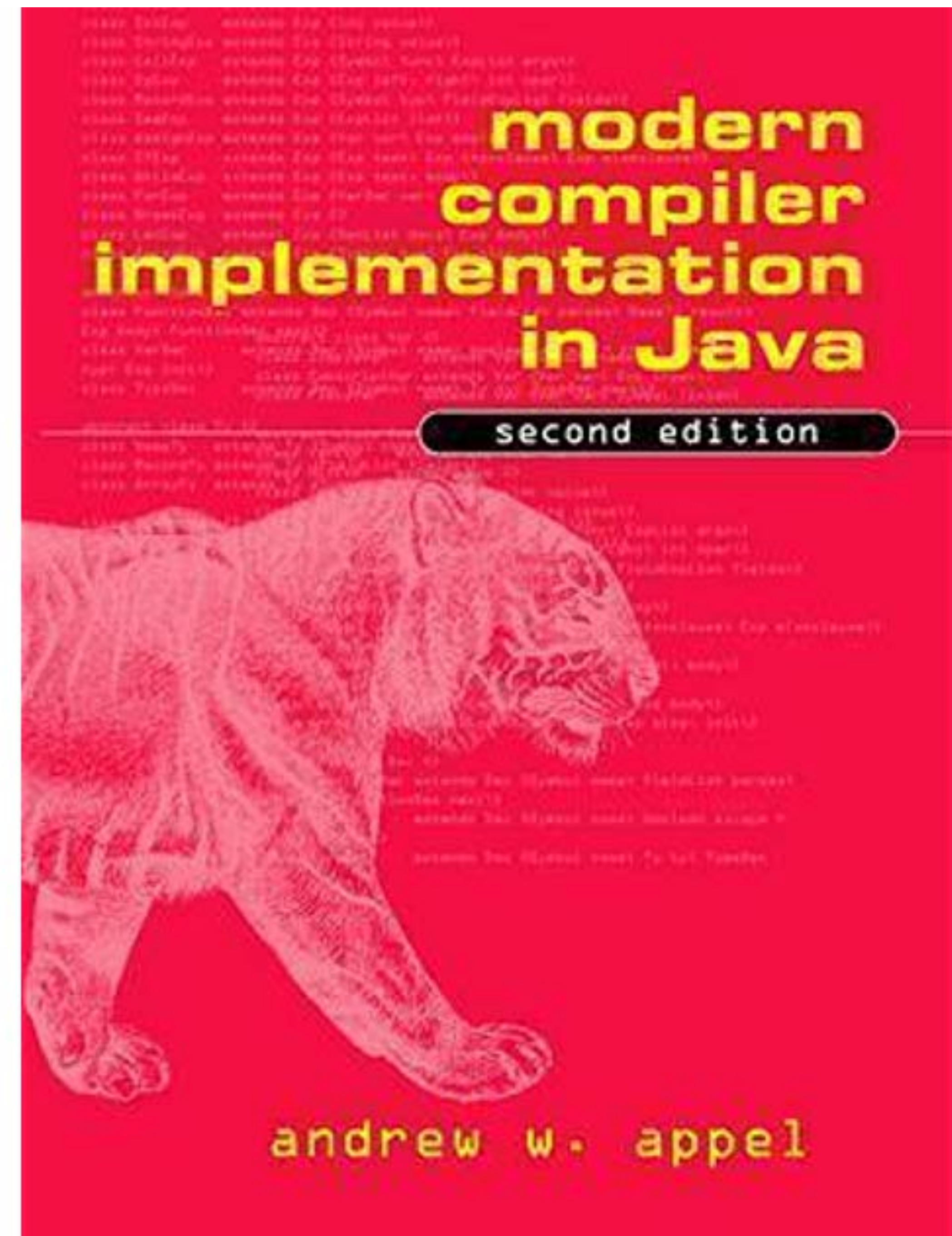
- collect only in young generations of records

# Reading Material



Andrew W. Appel and Jens Palsberg (2002). Garbage Collection. Chapter In Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press.

The lecture closely follows the discussion of mark-and-sweep collection, reference counts, copying collection, and generational collection in this chapter. This chapter also provides detailed cost analyses and discusses advantages and disadvantages of the different approaches to garbage collection.

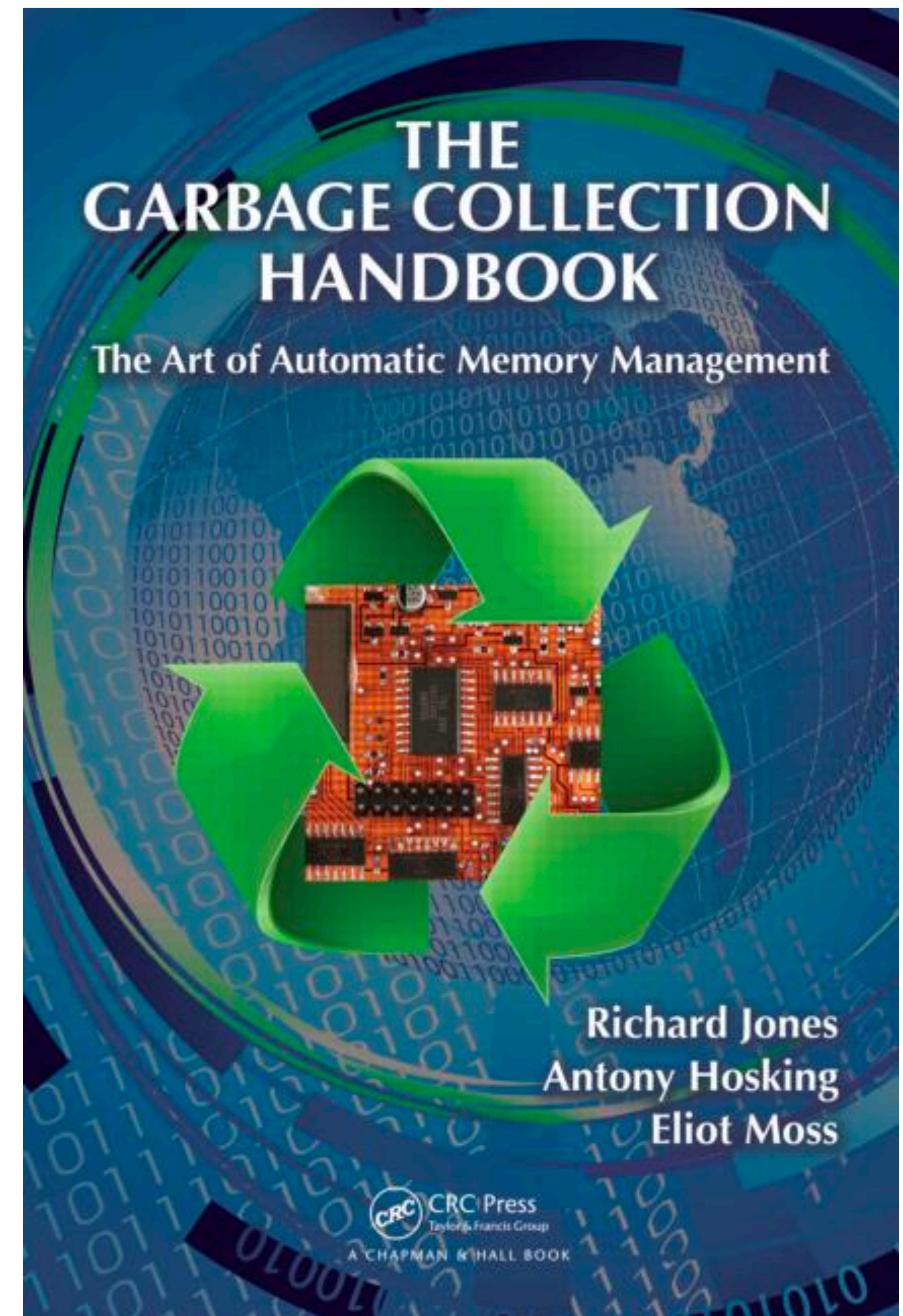




Richard Jones, Antony Hosking, Eliot Moss. The Garbage Collection Handbook. The Art of Automatic Memory Management.

A systematic overview of garbage collection algorithms.

Dig deeper





# Memory Safety & Memory 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 base of the memory region it may access
- e is the extent (bounds of that region)

**Access allowed iff**

- $b \leq p \leq e - \text{sizeof}(\text{typeof}(p))$

**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

## 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

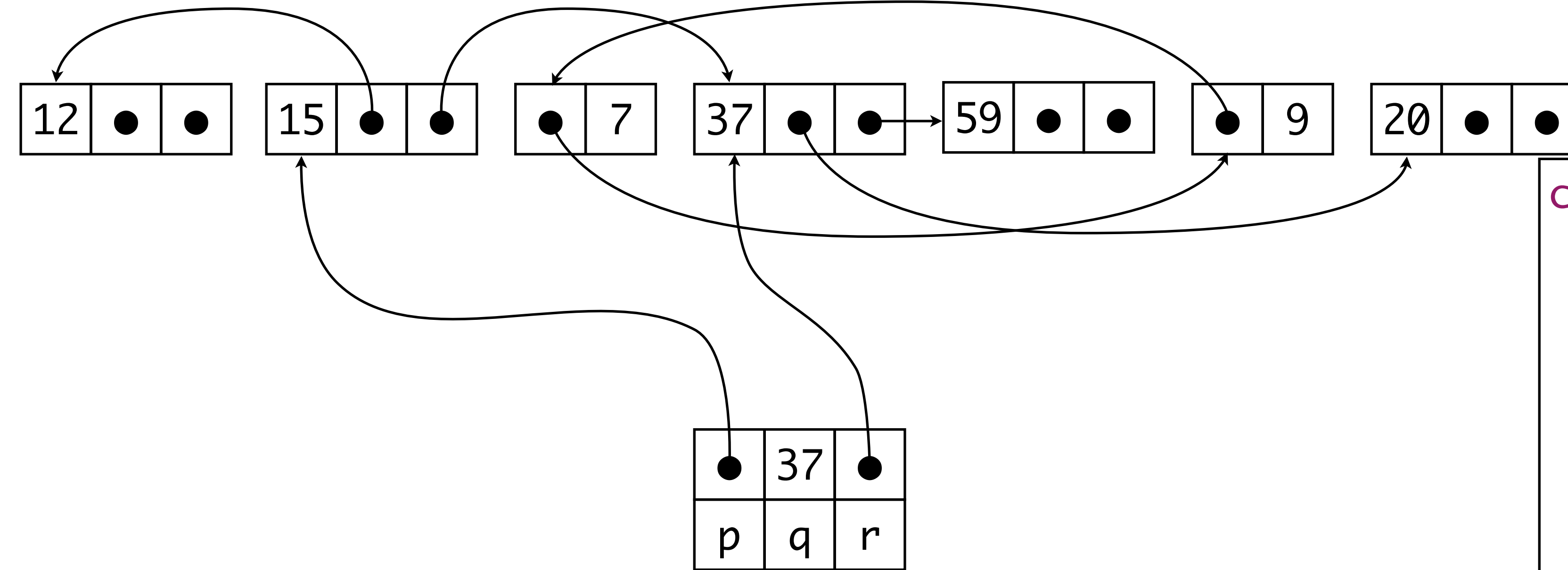
- objects that are referenced are live
- objects that are not referenced are dead (garbage)
- objects are allocated on the heap

## Responsibilities

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



# An Example Program



```
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)
        }
    }
}
```

# Reference Counting

# Reference Counting

## Counts

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

## Counting

- extra instructions

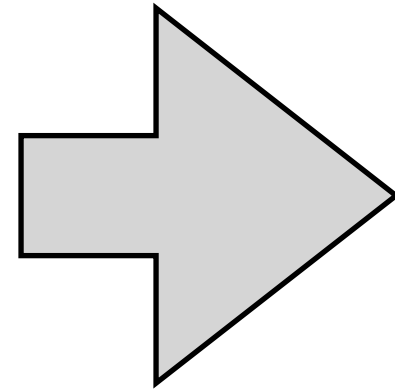
## Deallocate

- put on freelist
- recursive deallocation on allocation



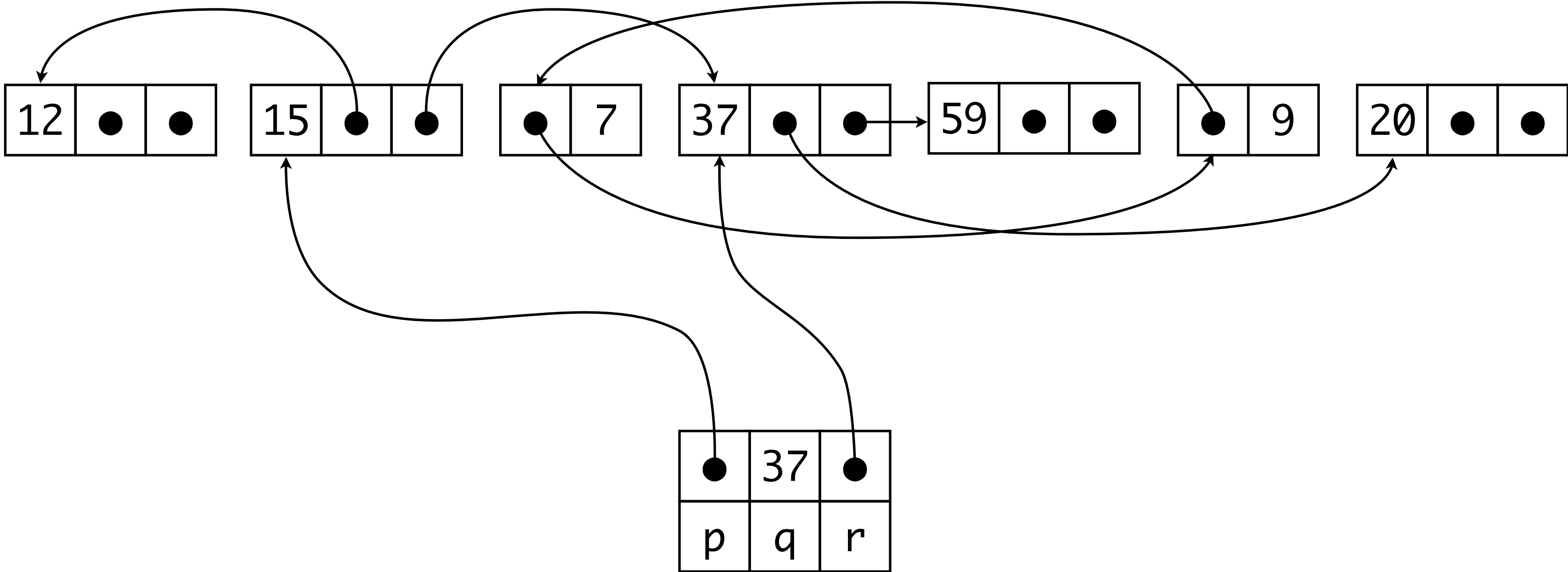
# Reference Counting: Instrumentation

`x.f := p`

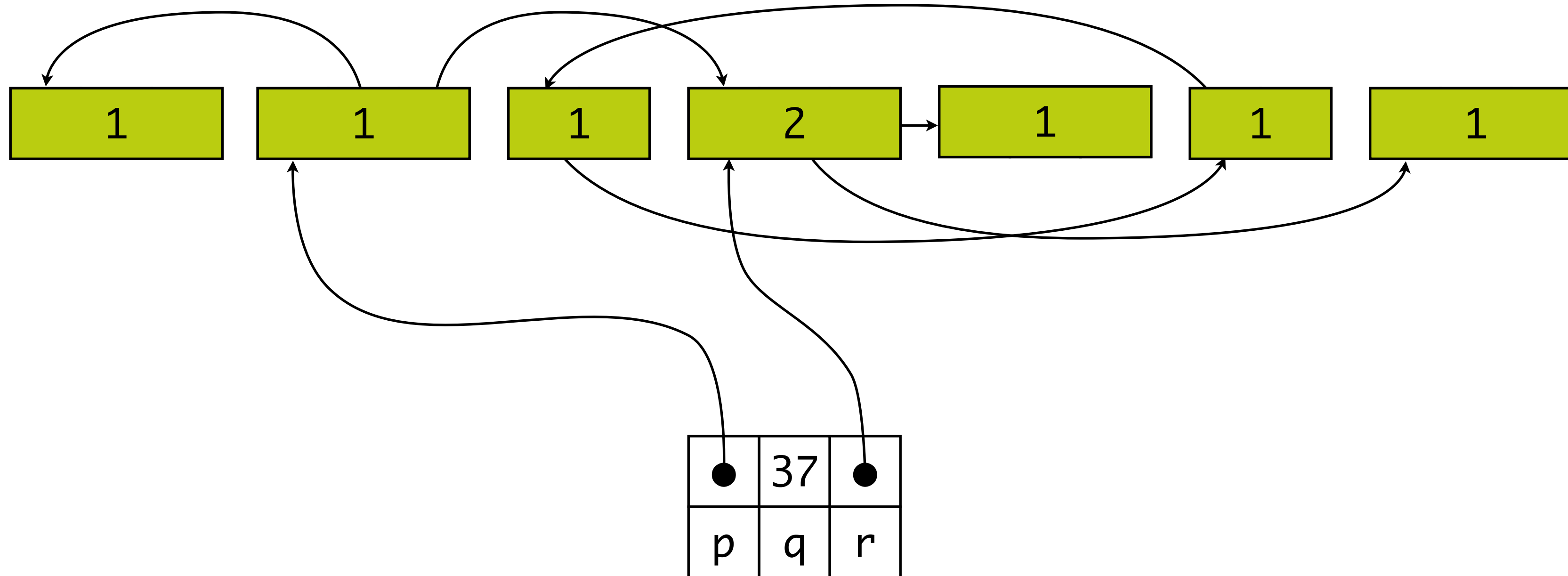


```
z      := x.f
c      := z.count
c      := c - 1
z.count := c
if (c == 0) put z on free list
x.f     := p
c       := p.count
c       := c + 1
p.count := c
```

# Reference Counting

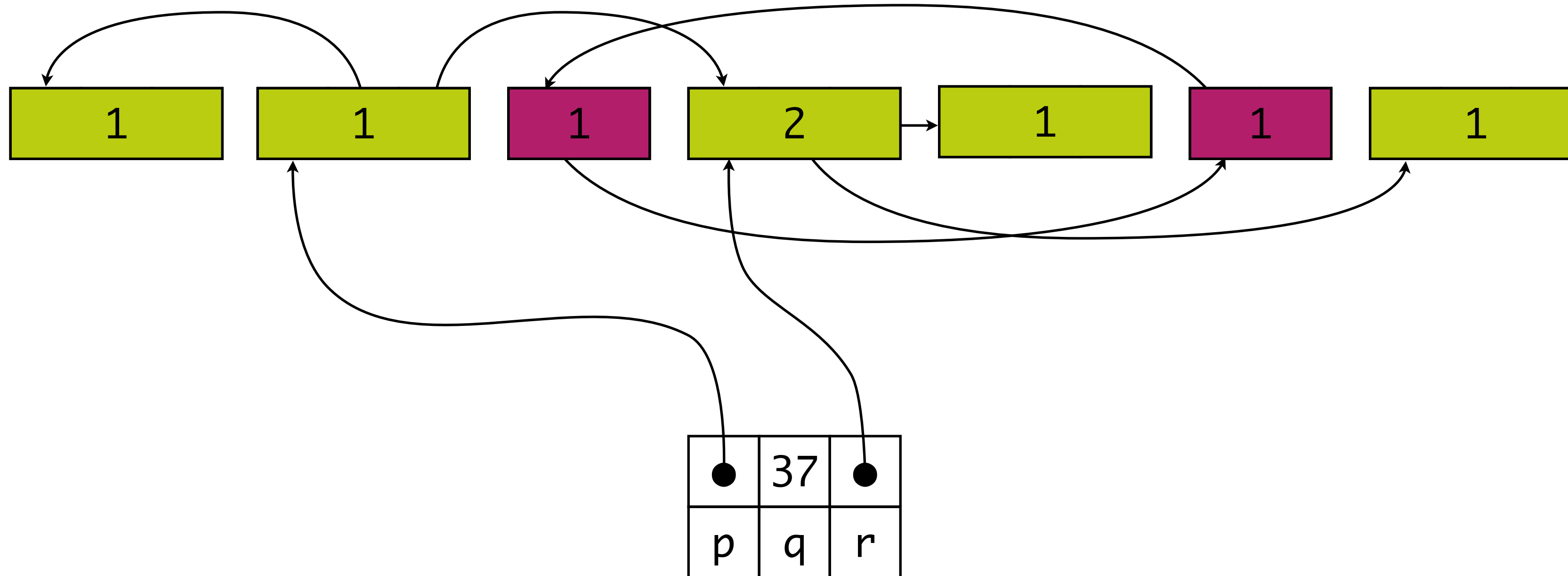


# Reference Counting





# Reference Counting



# 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

# Programming Languages using Reference Counting

## 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

# Mark & Sweep



# Mark & Sweep: Idea

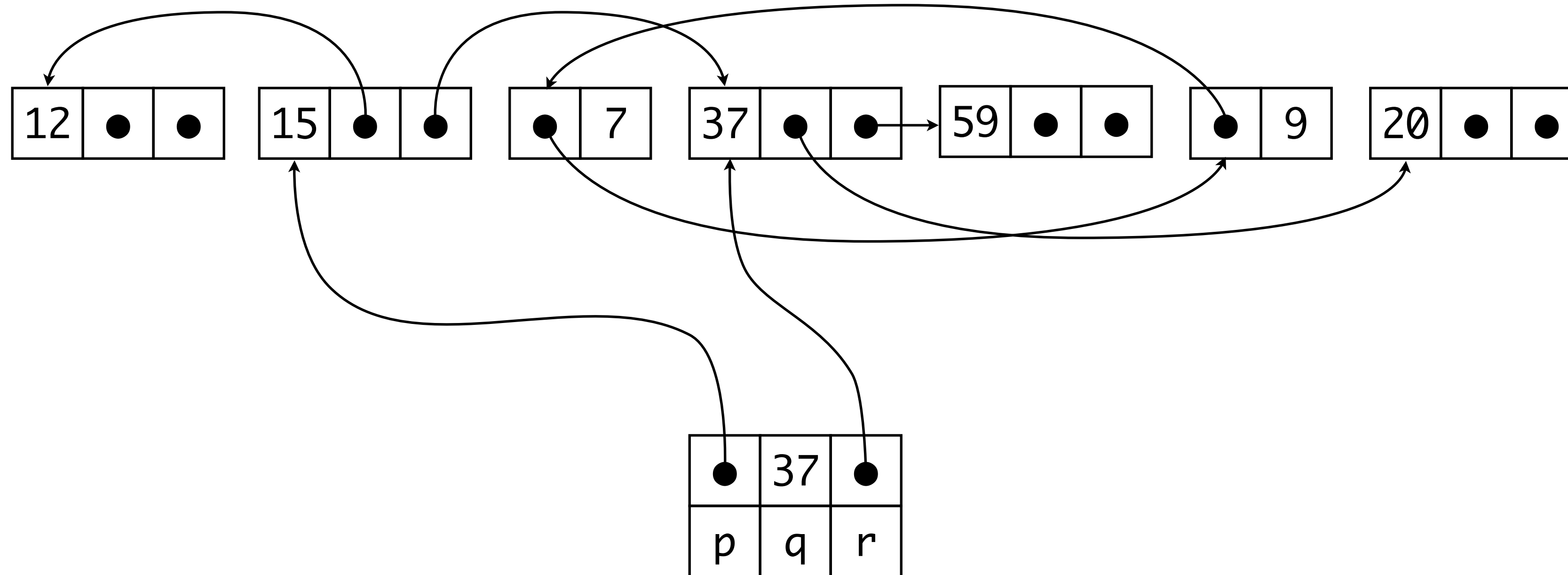
## Mark

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

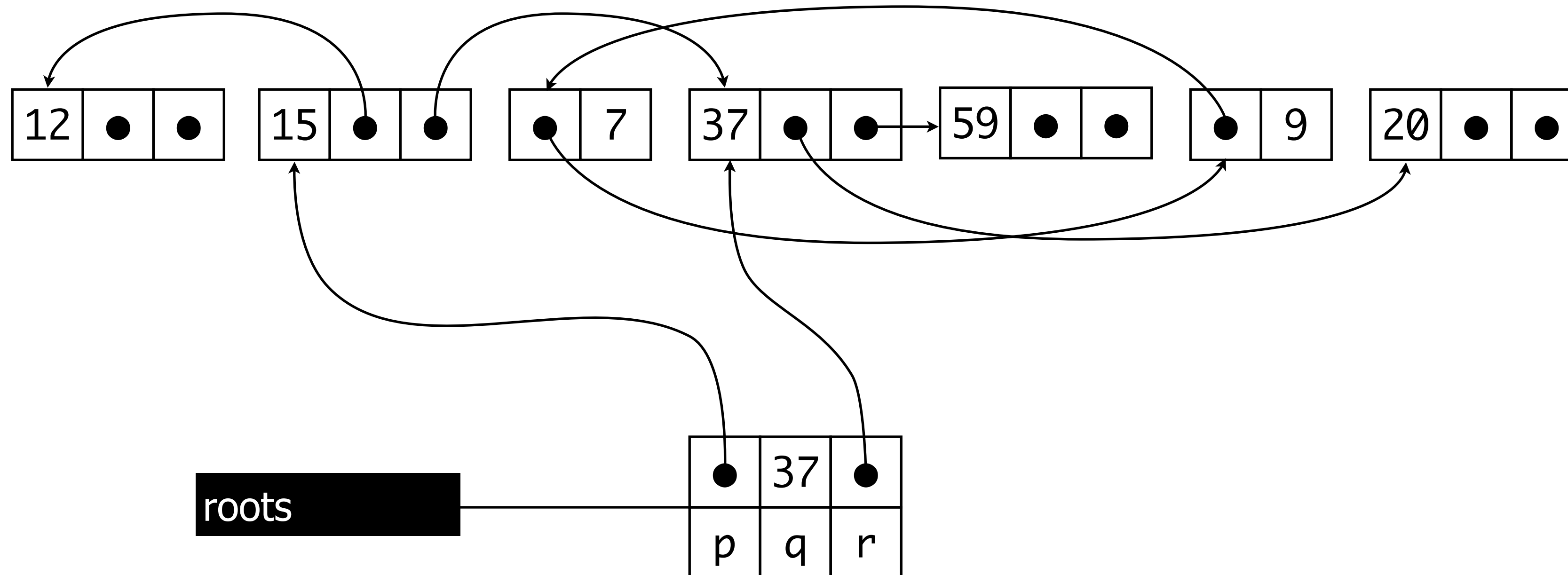
## Sweep

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

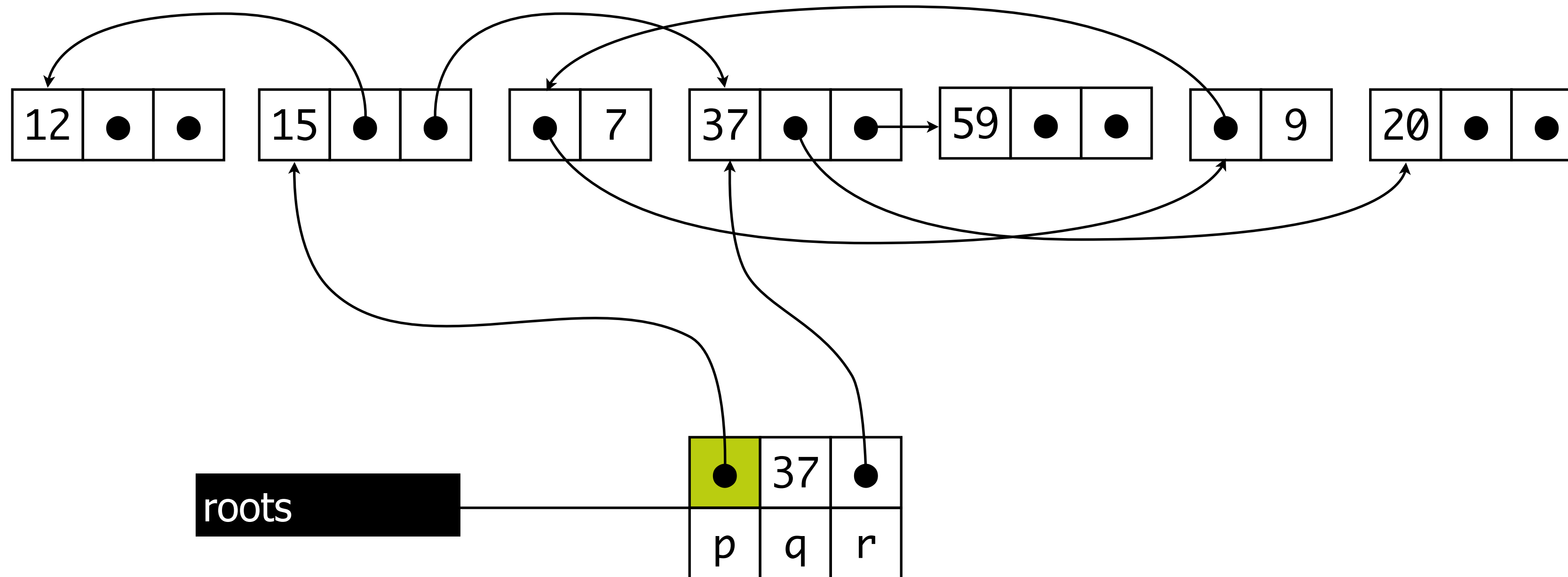
# Marking



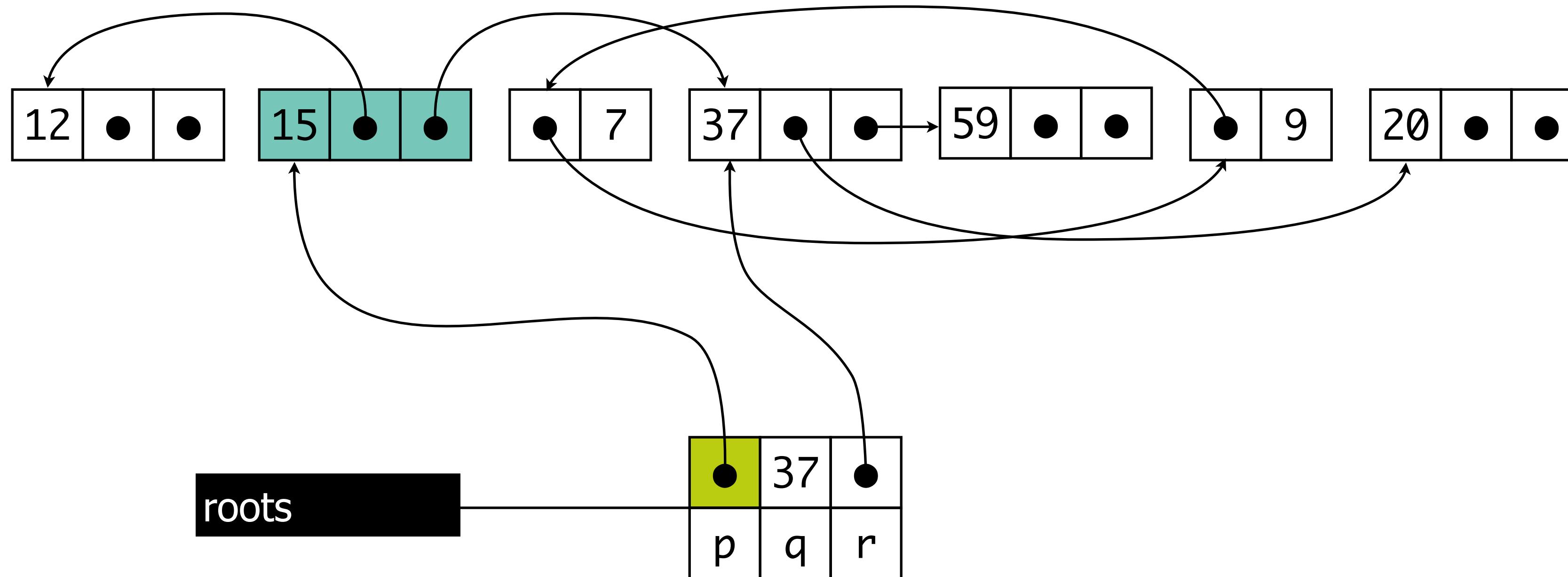
# Marking



# Marking

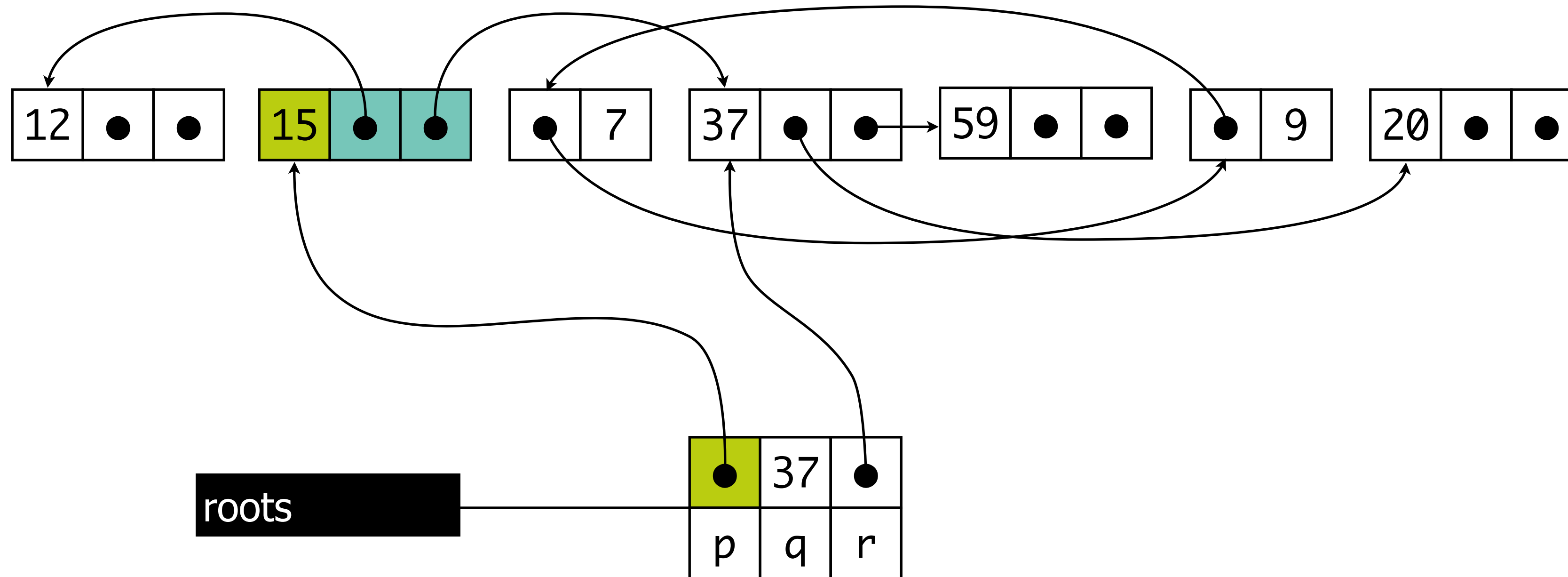


# Marking

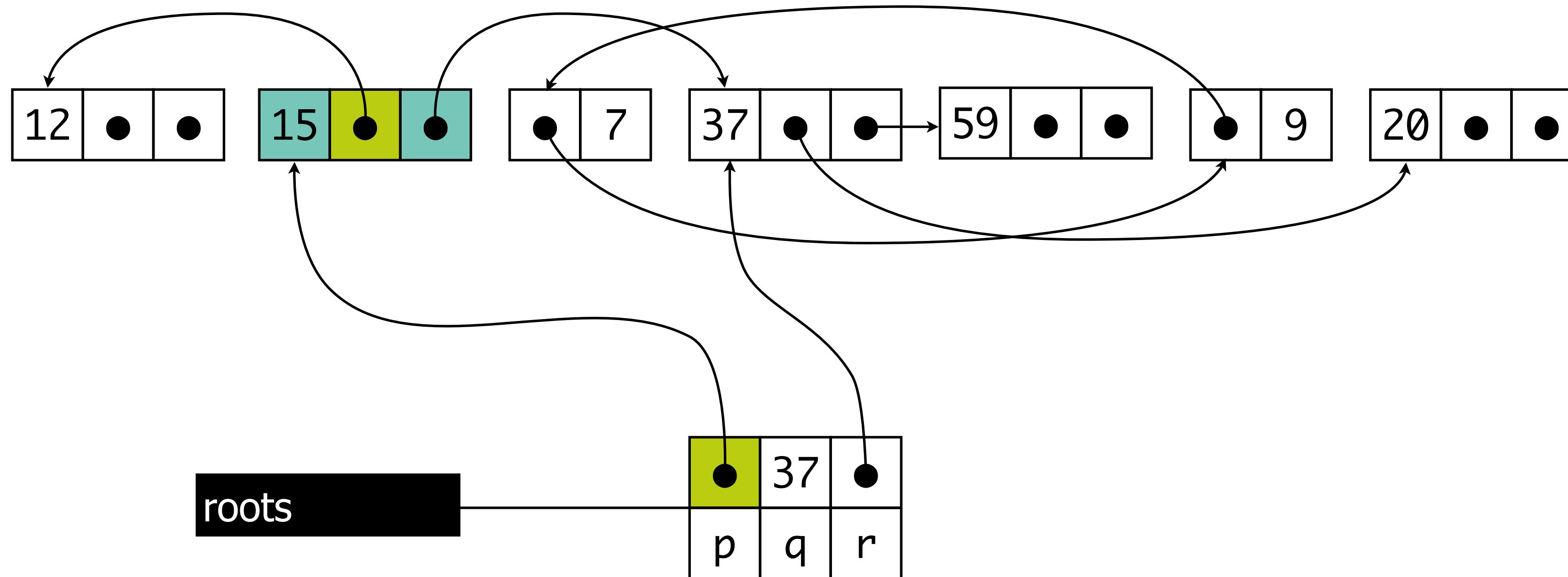




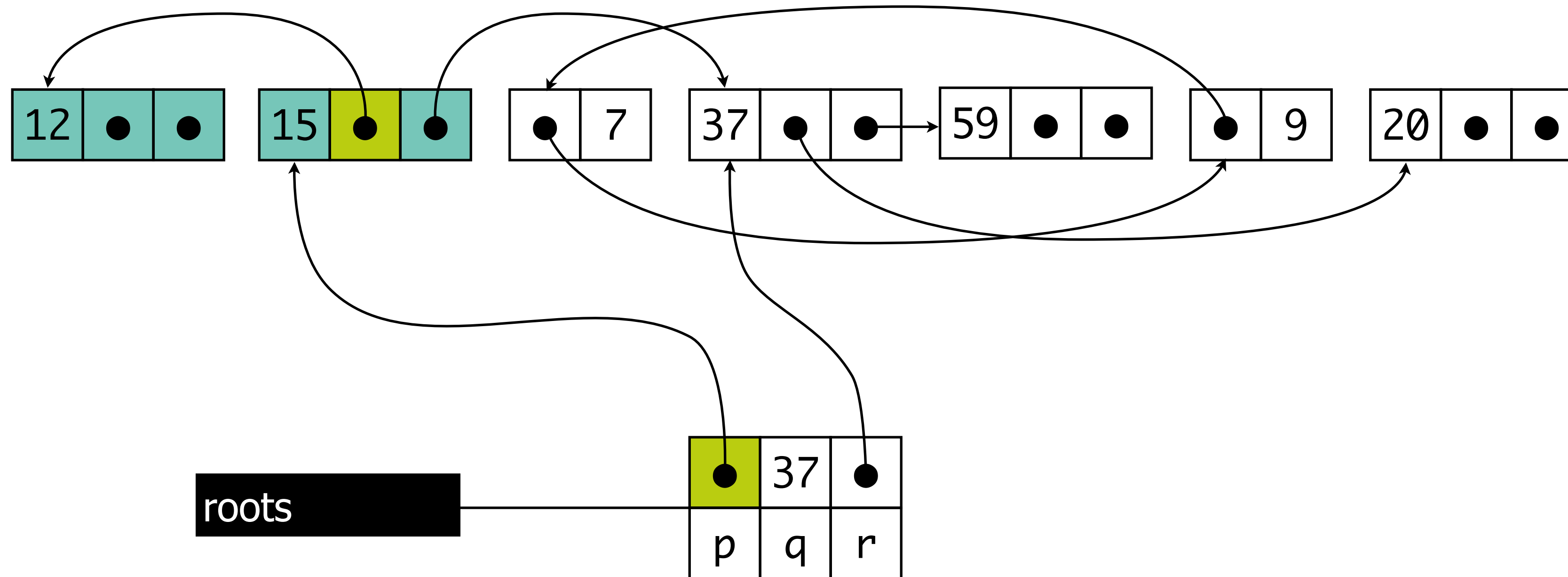
# Marking



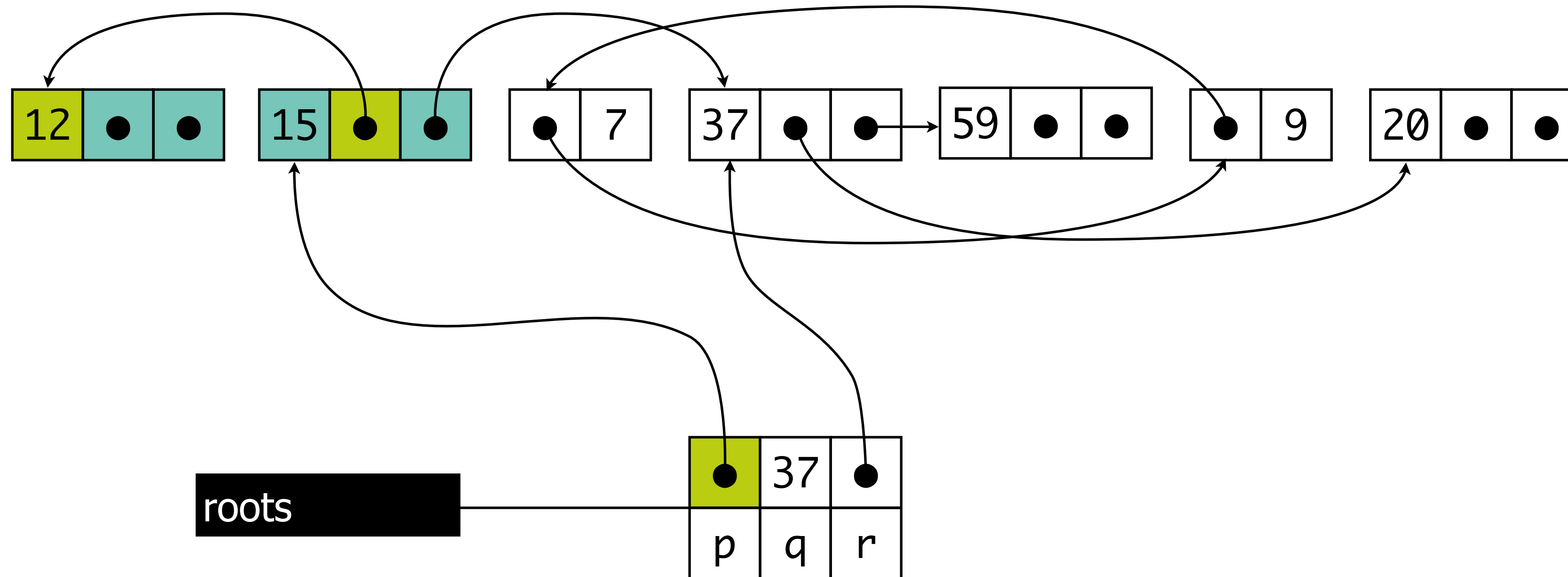
# Marking



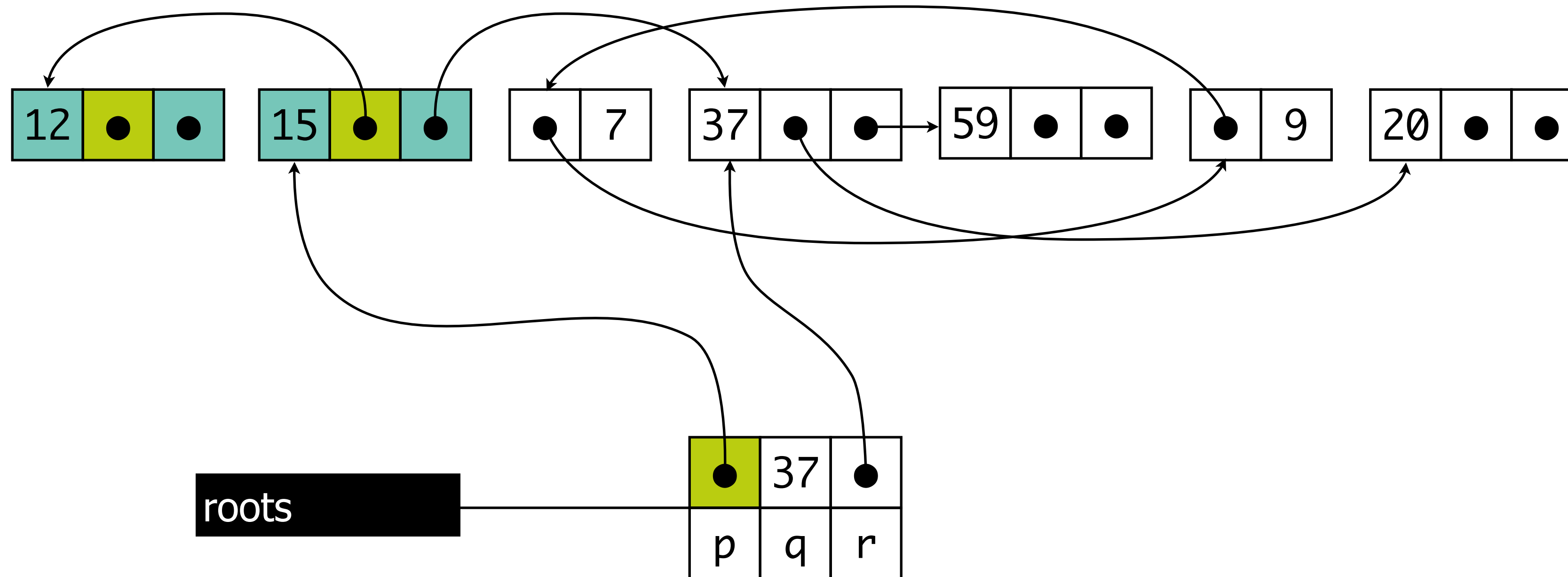
# Marking



# Marking

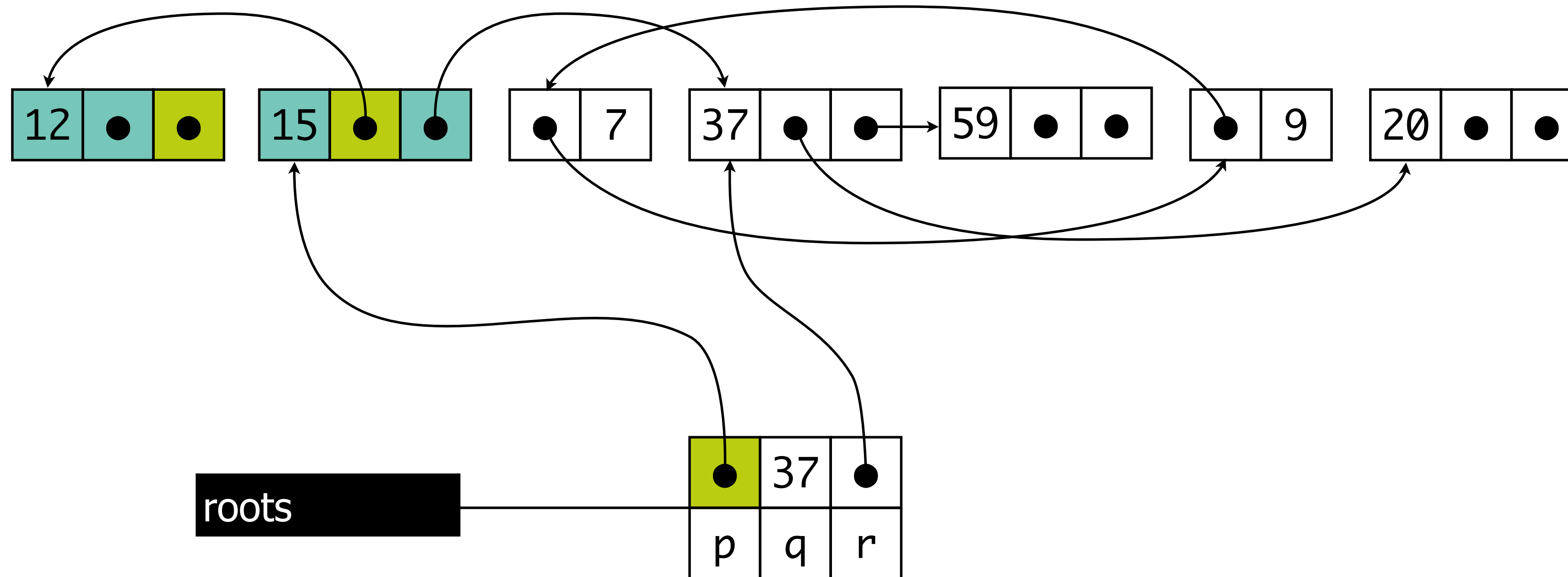


# Marking

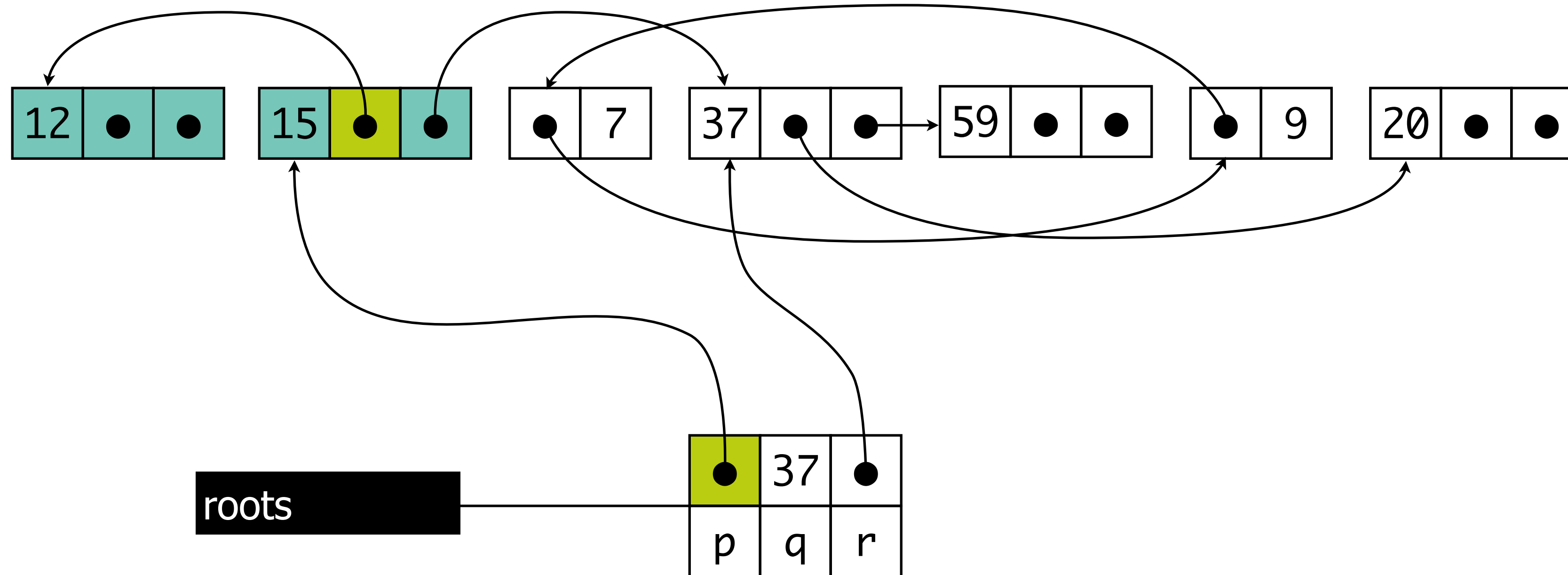




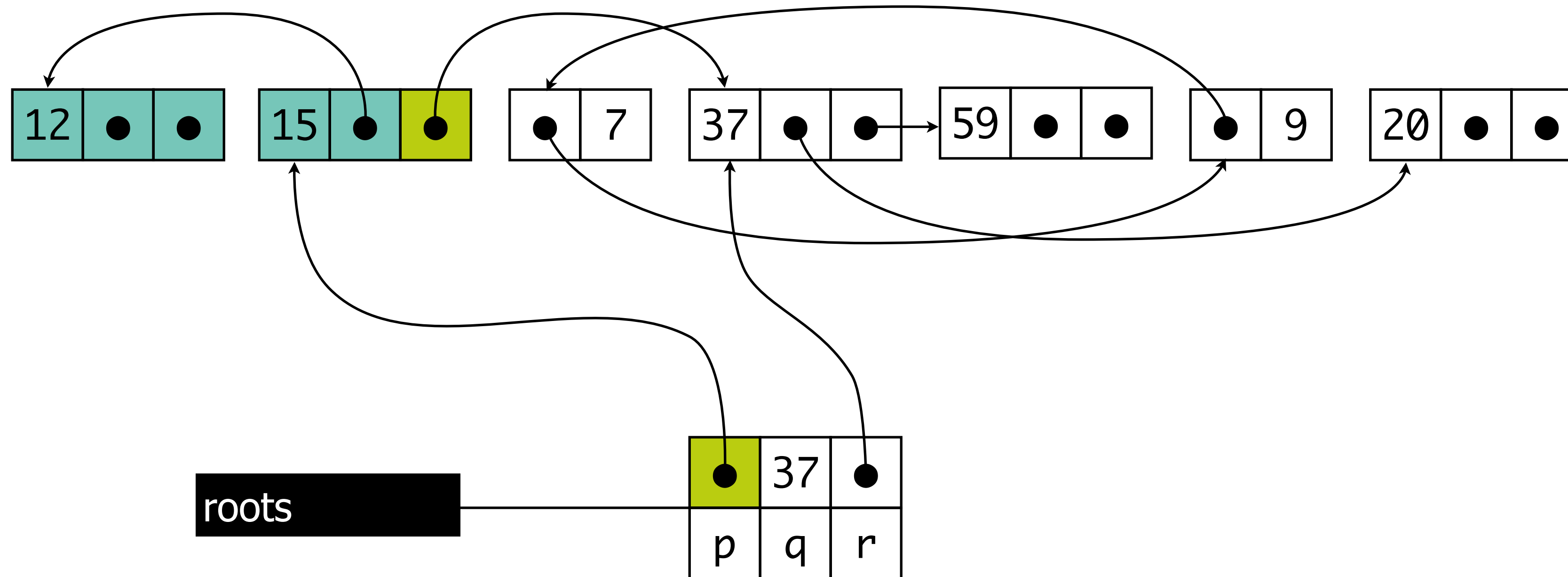
# Marking



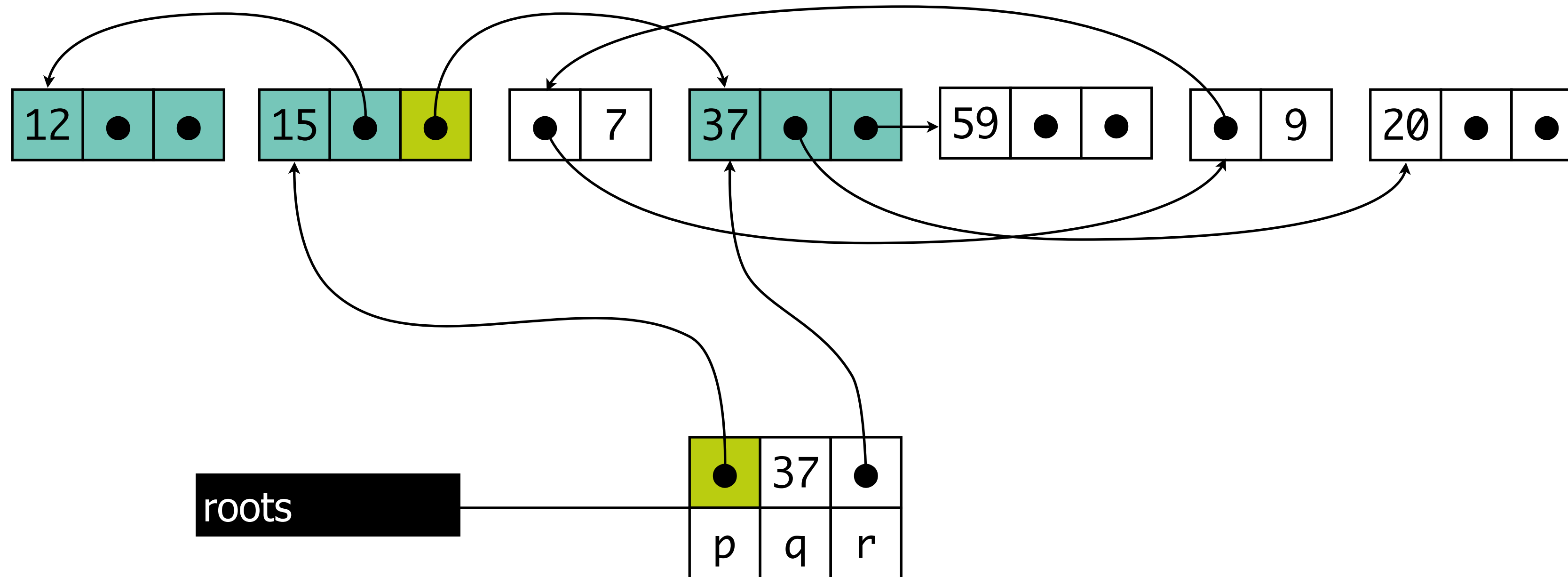
# Marking



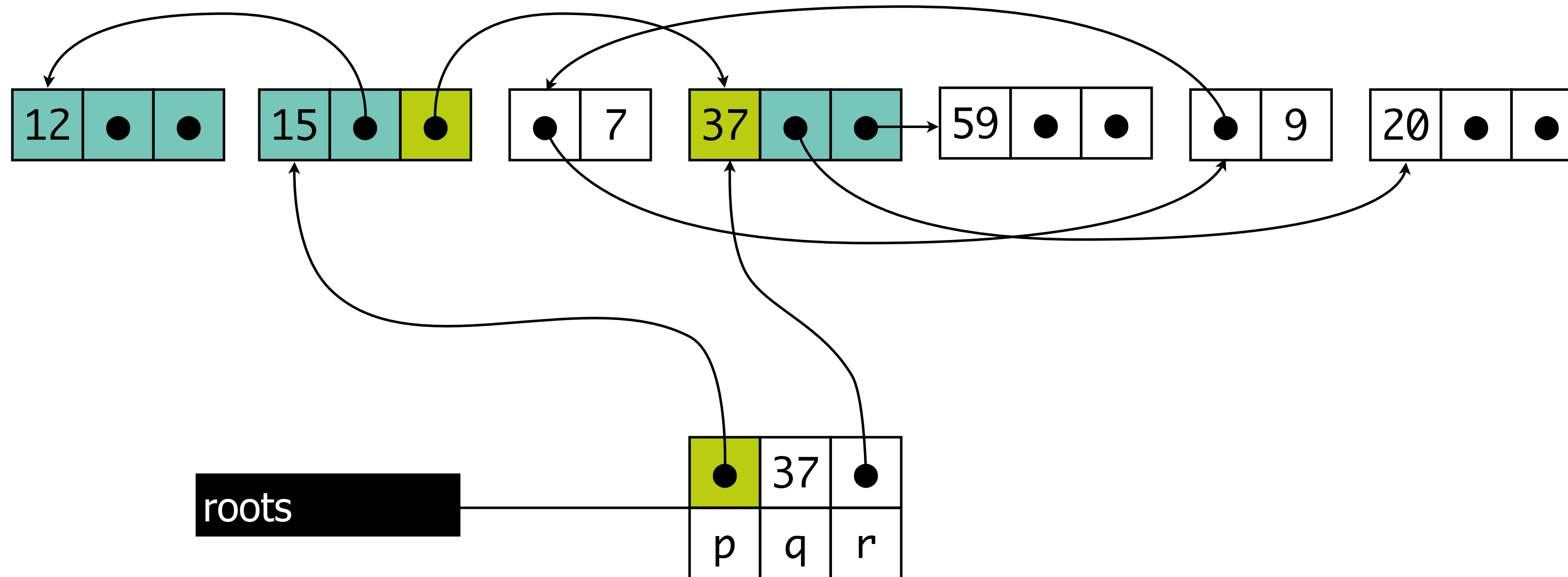
# Marking



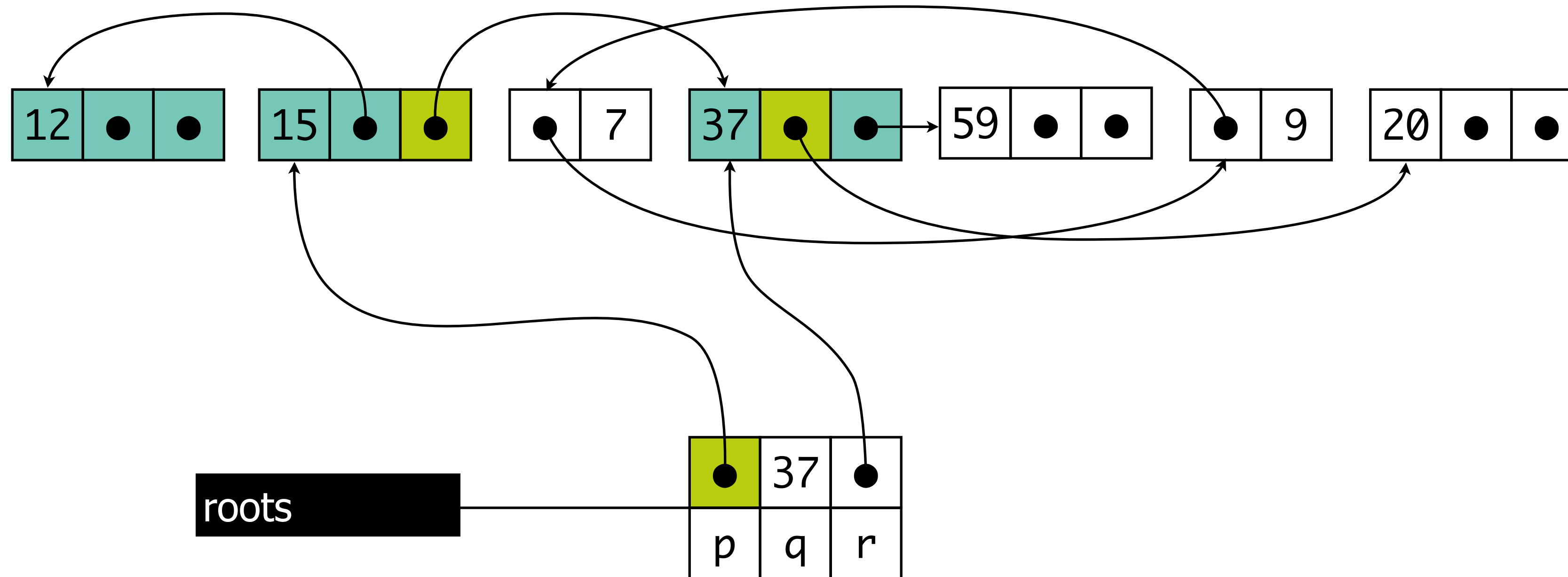
# Marking



# Marking

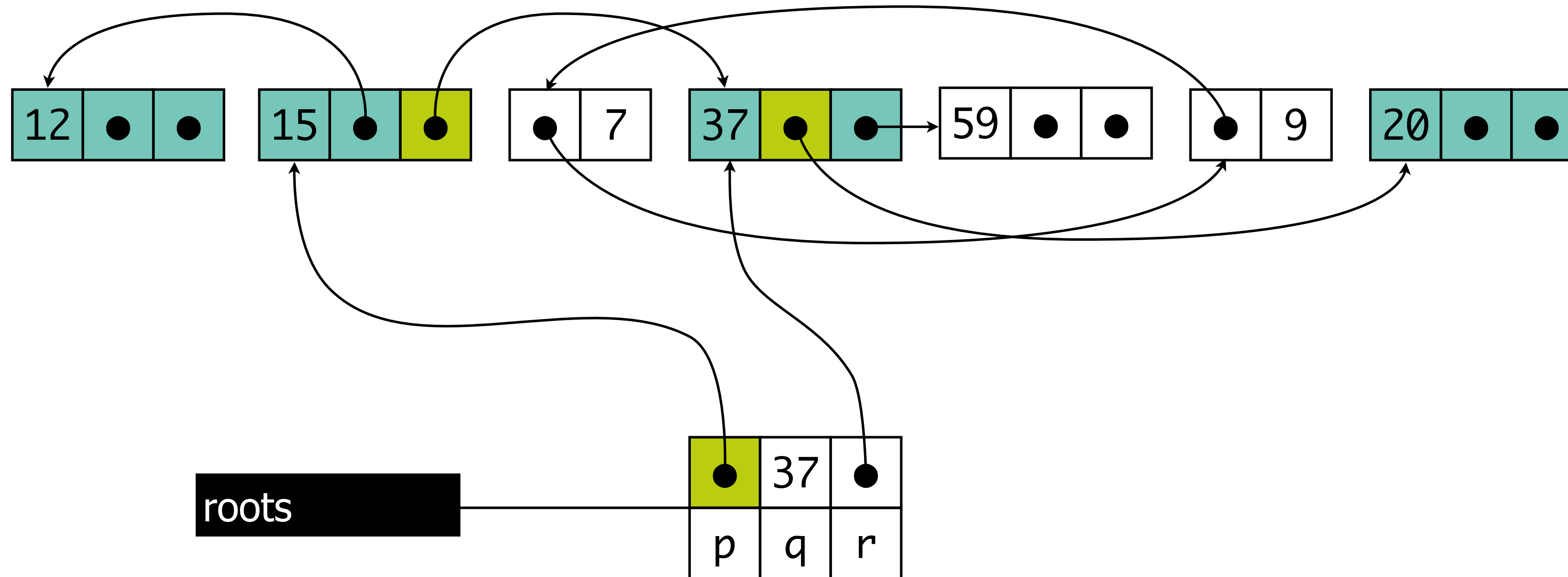


# Marking

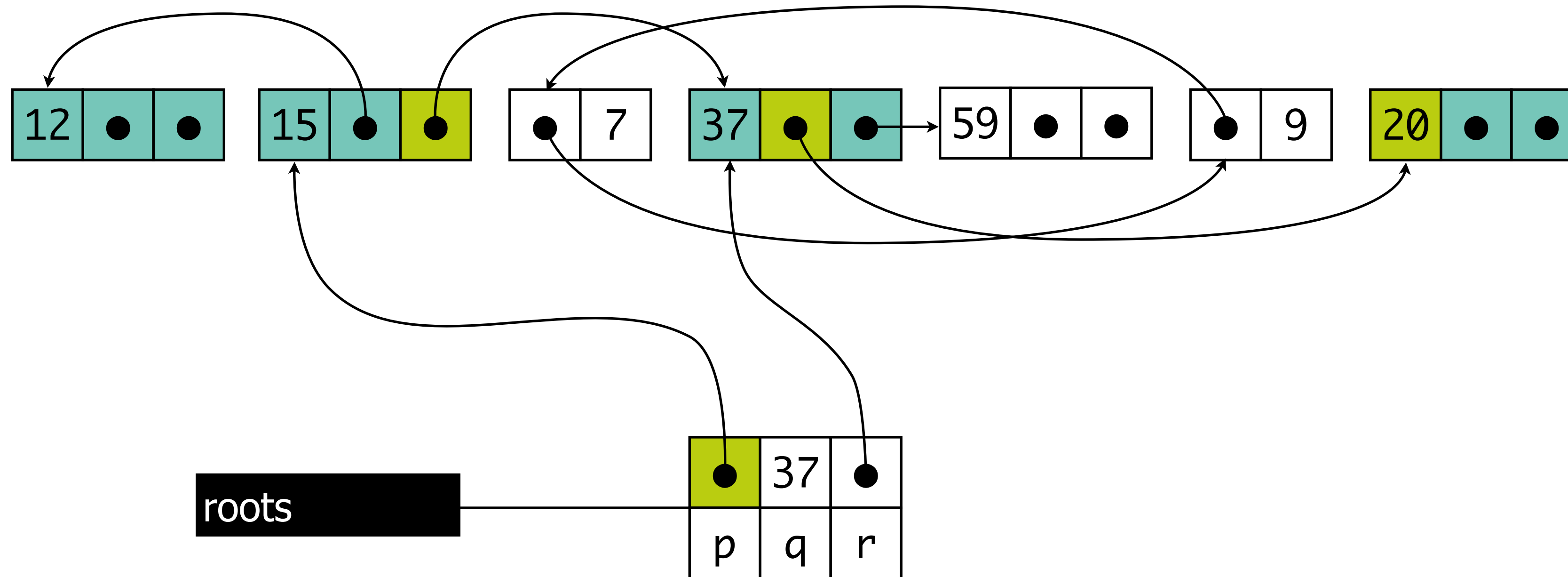




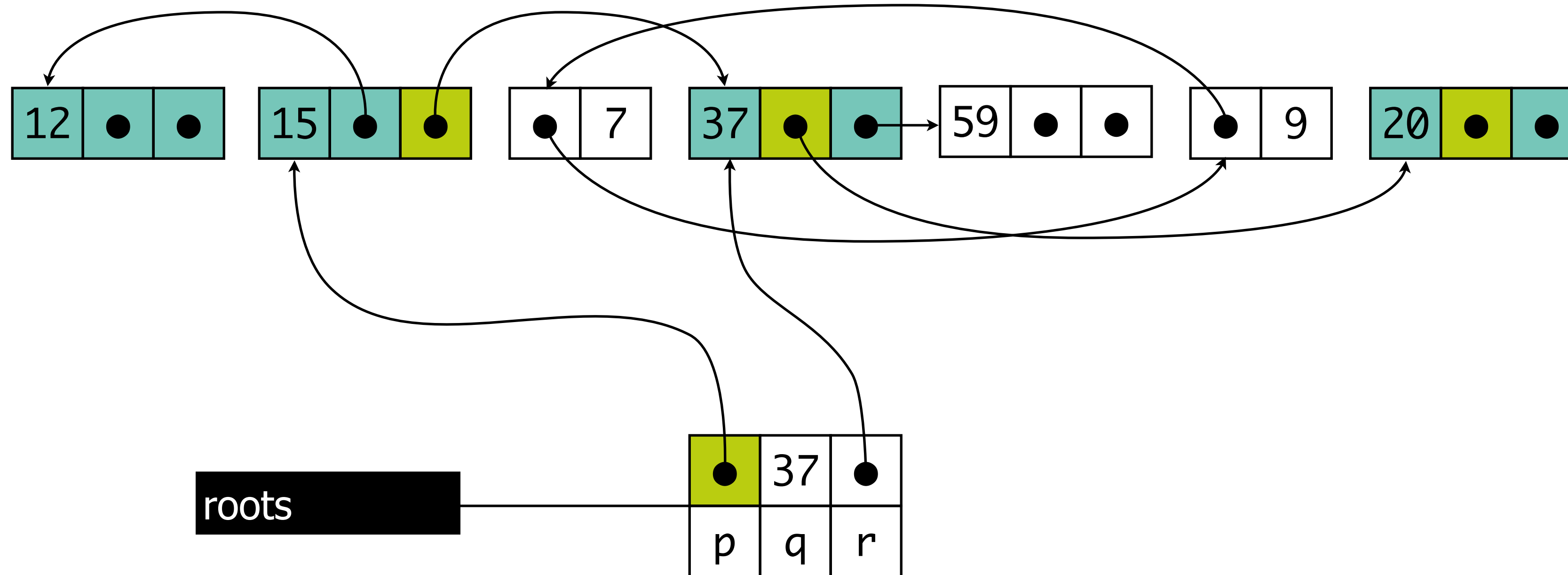
# Marking



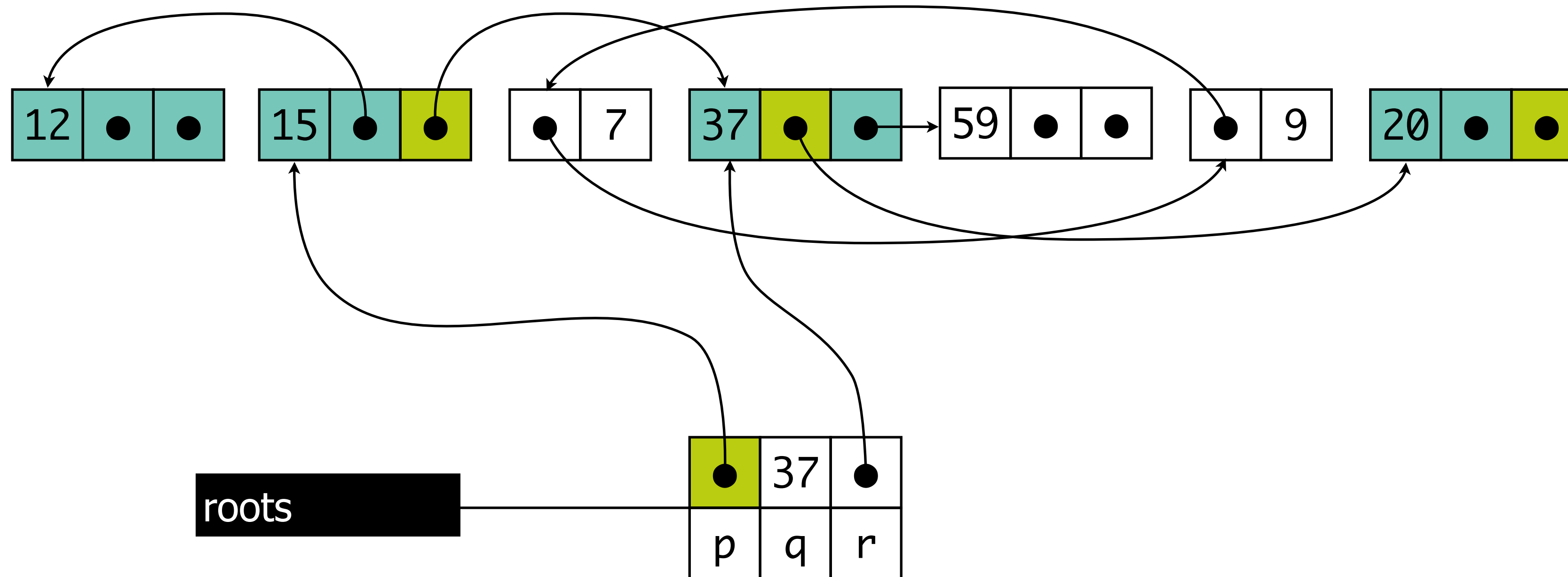
# Marking



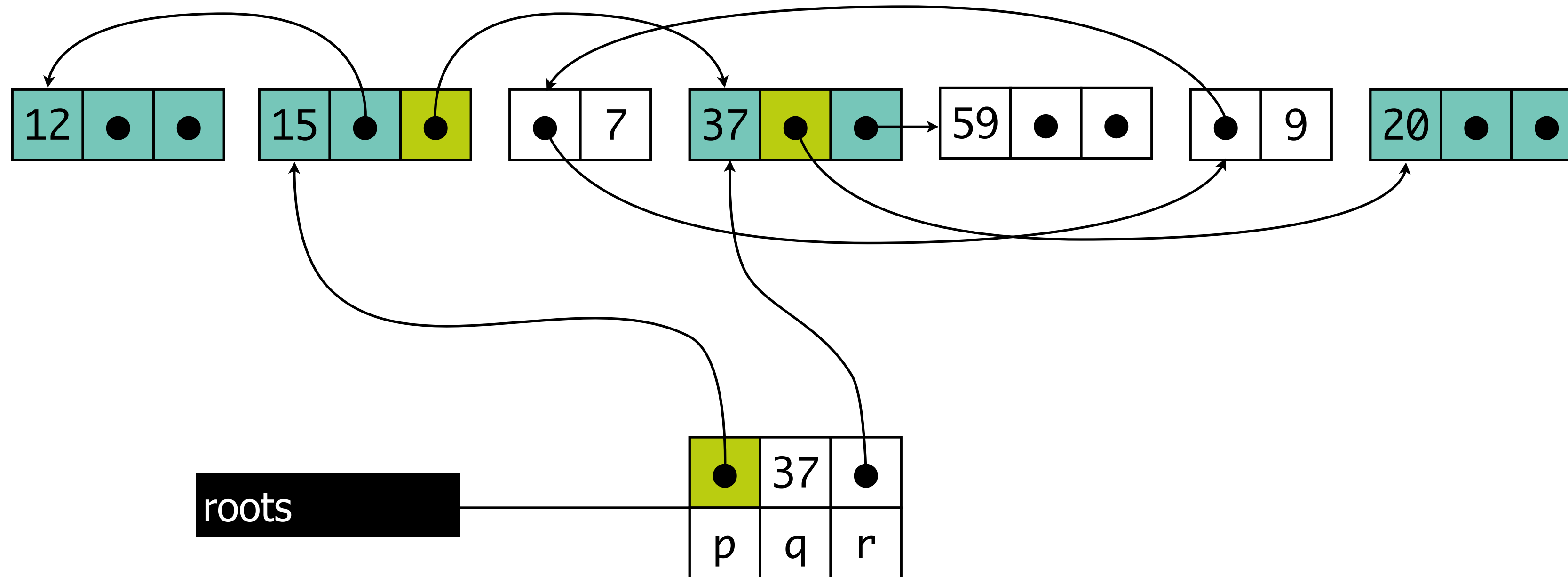
# Marking



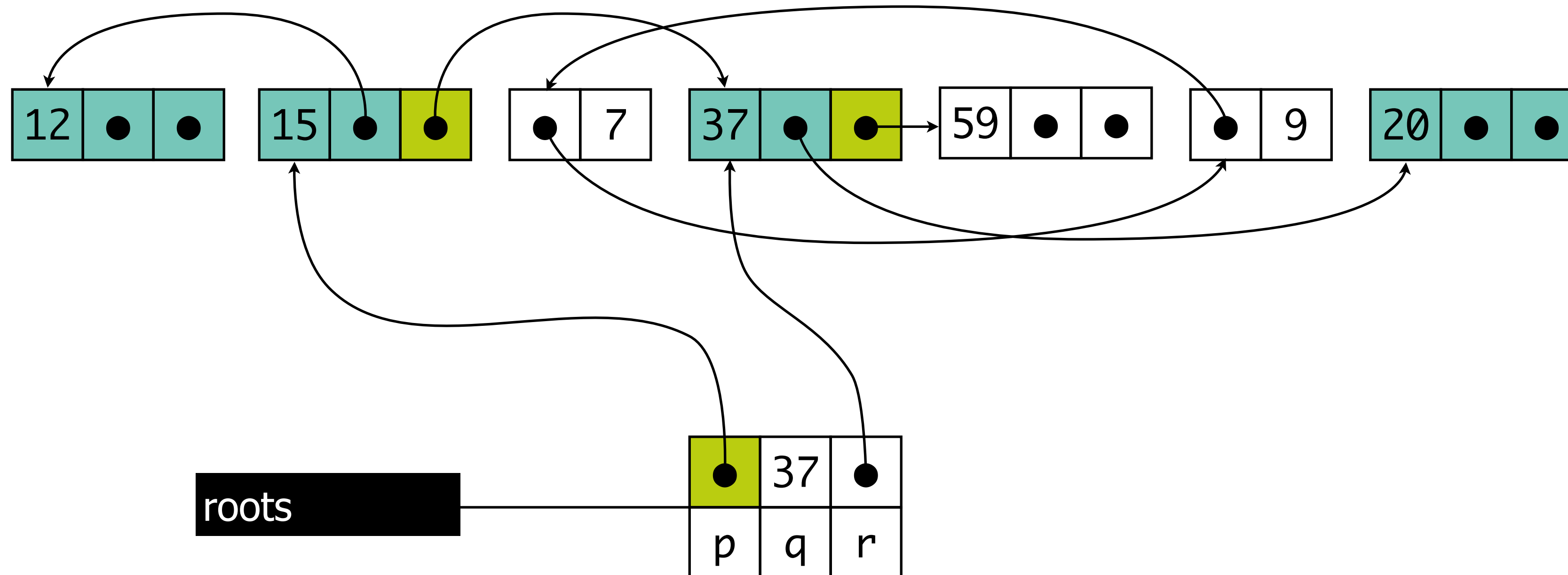
# Marking



# Marking

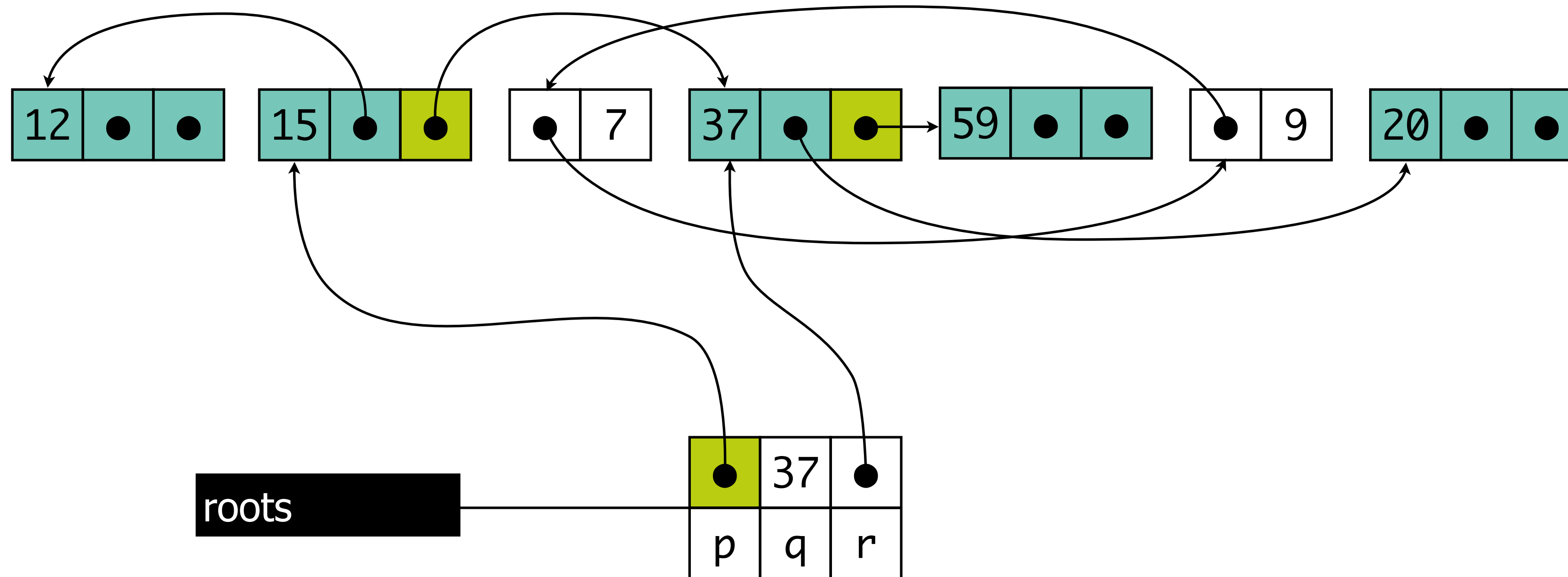


# Marking

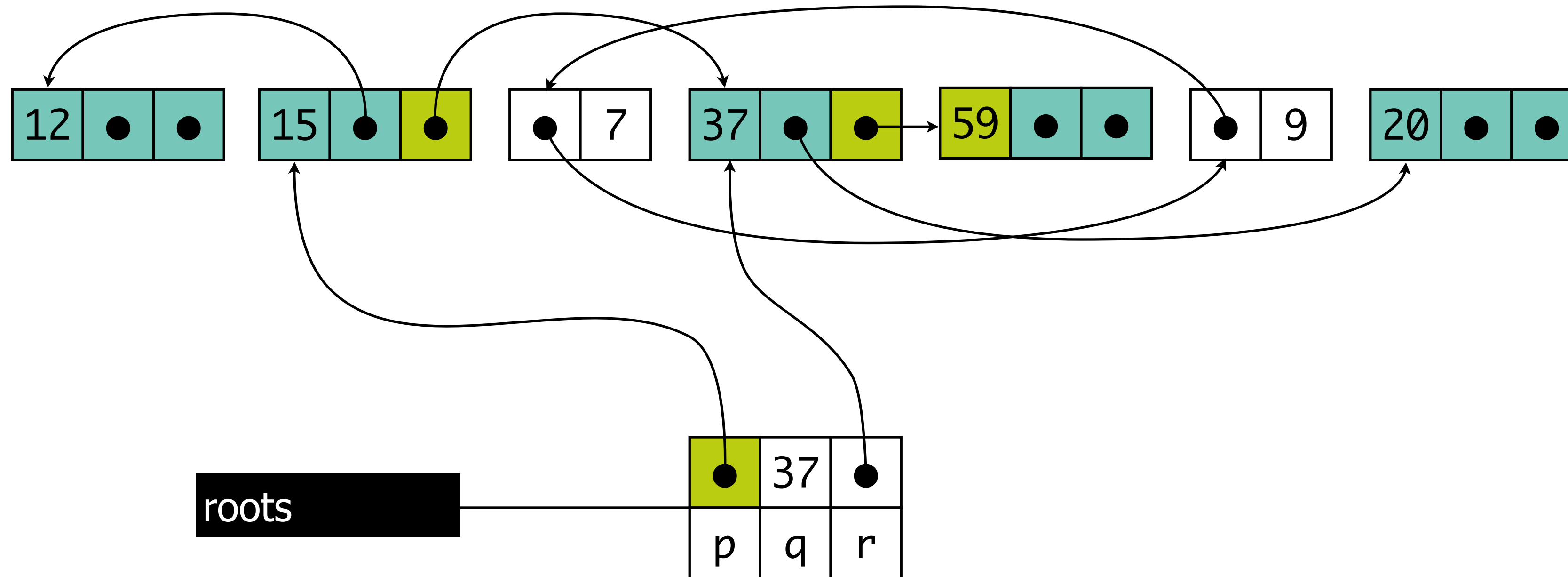




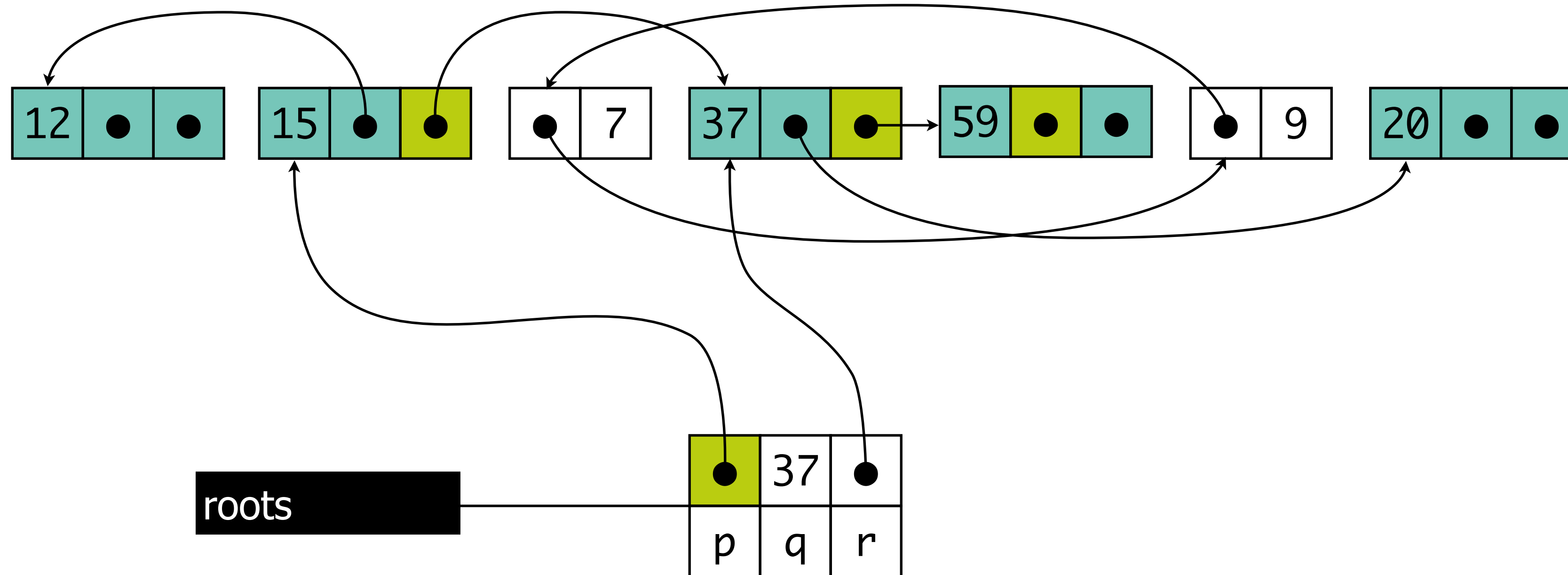
# Marking



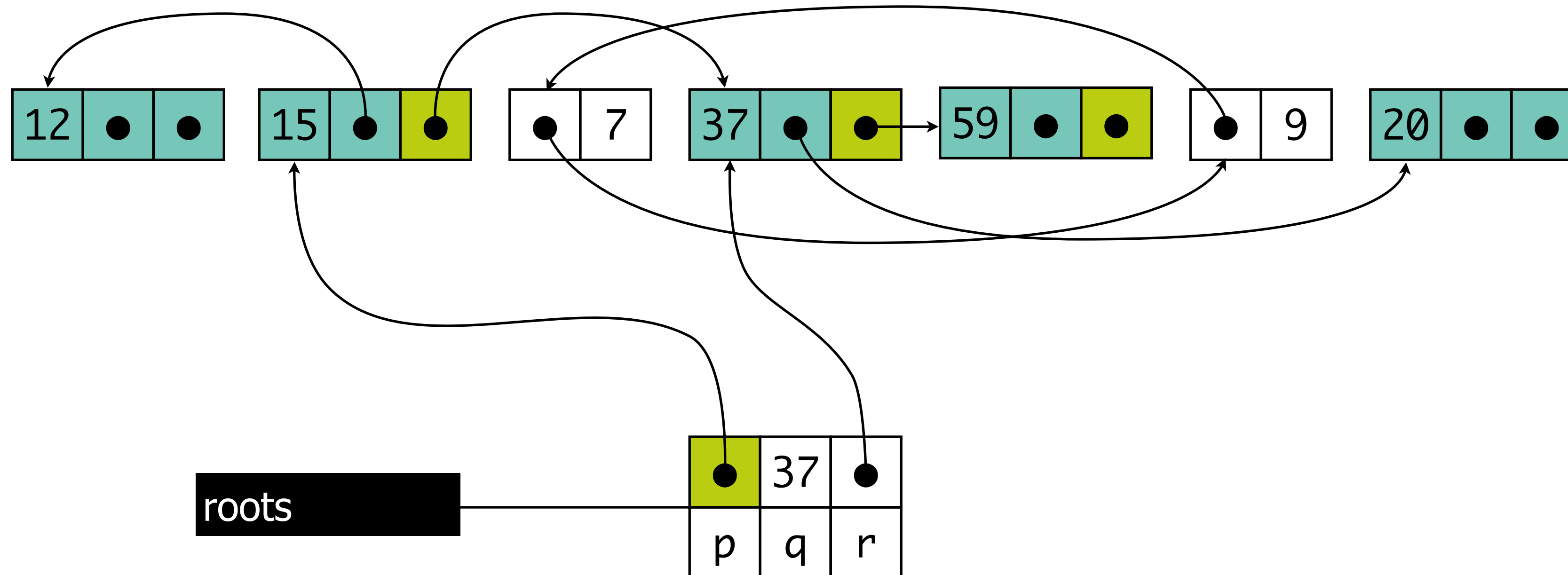
# Marking



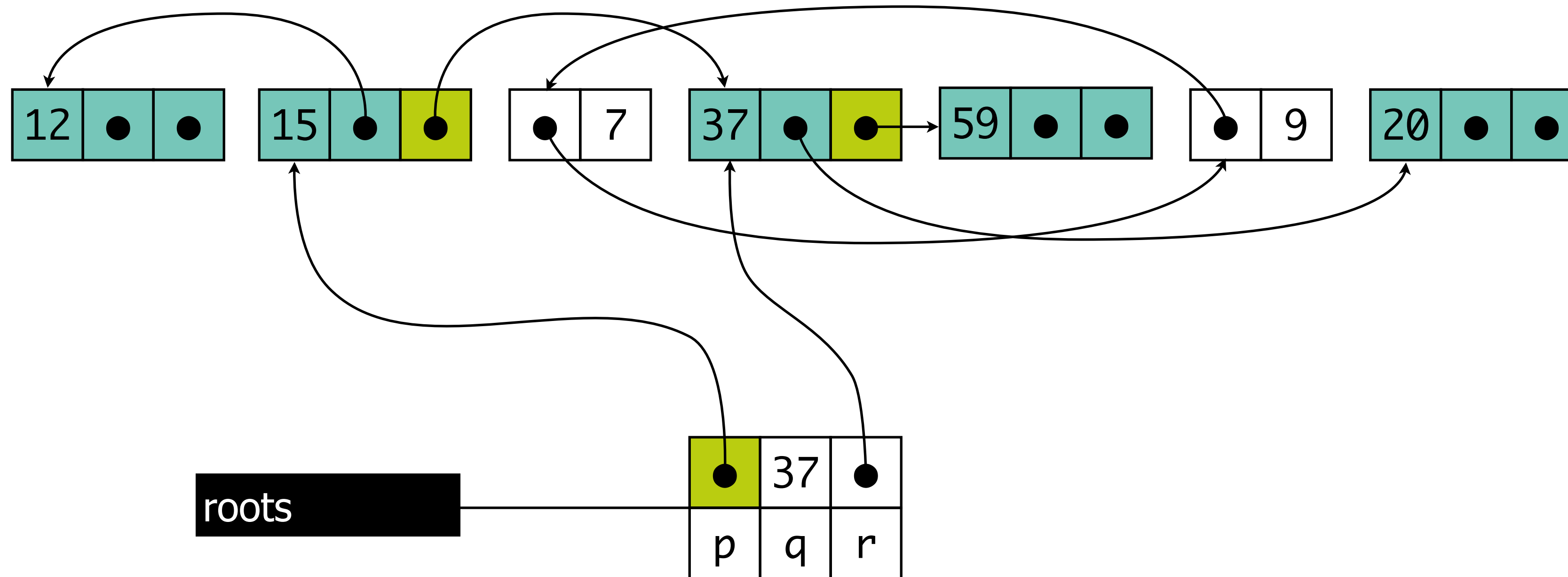
# Marking



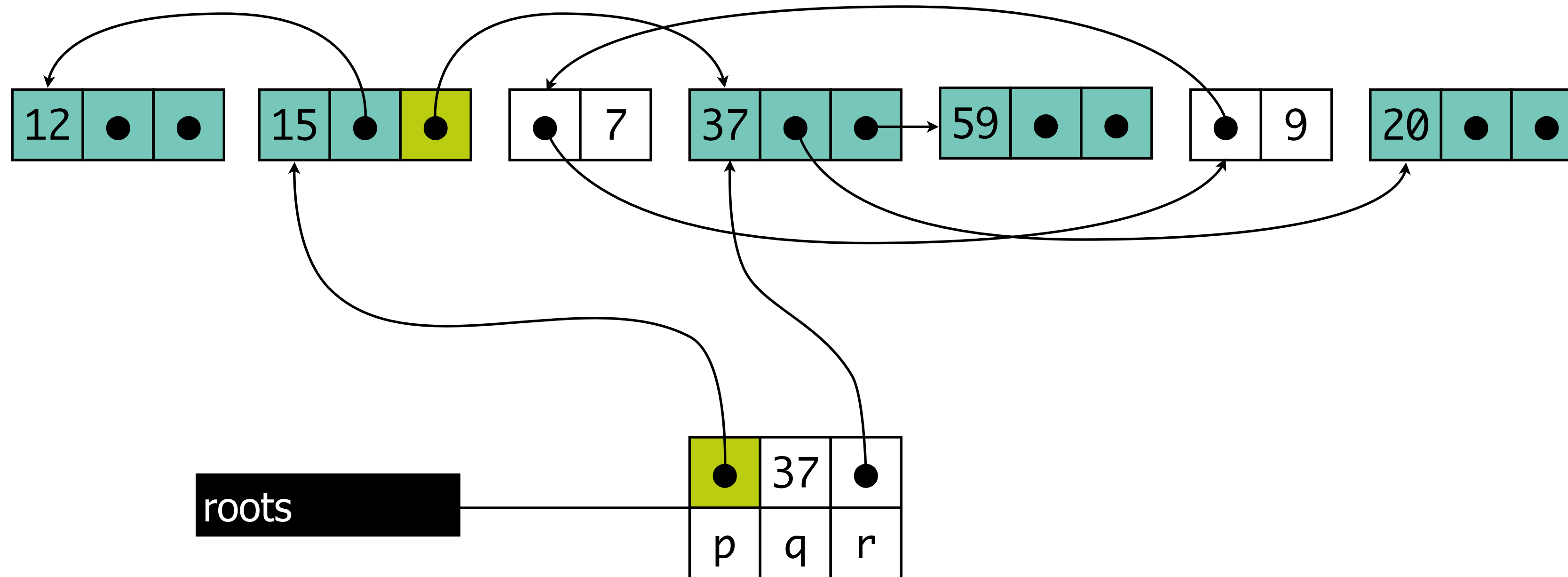
# Marking



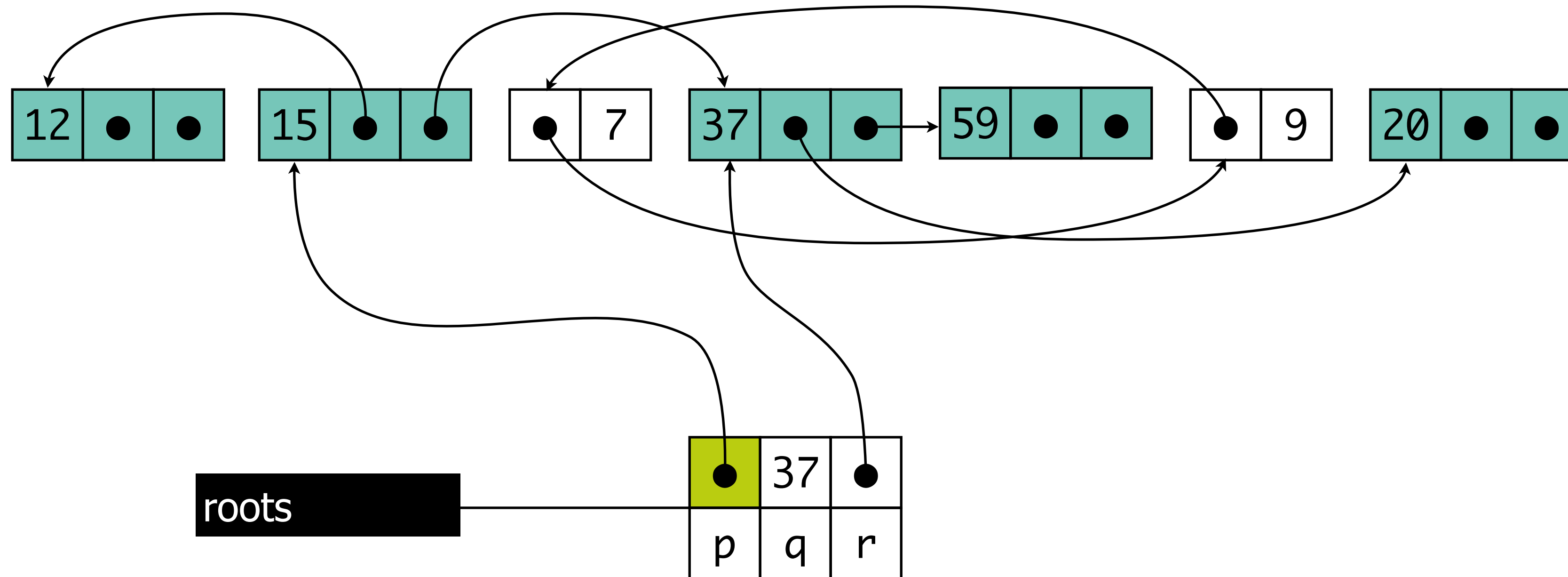
# Marking



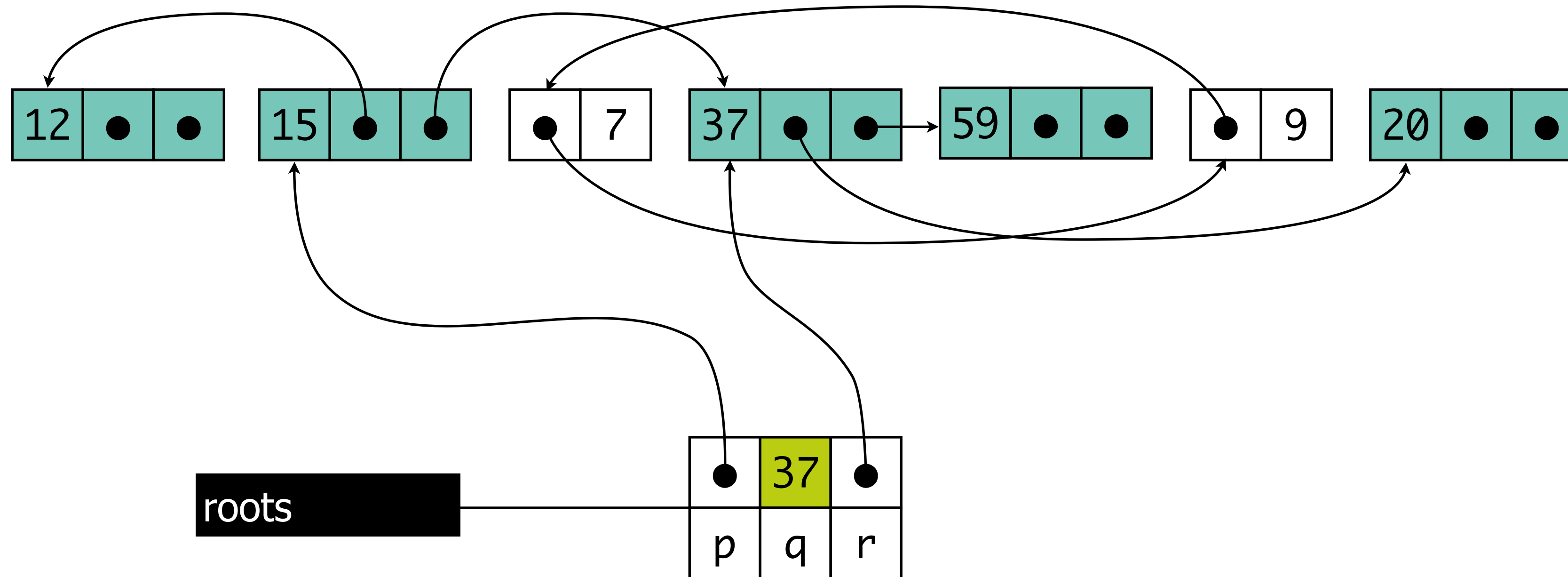
# Marking



# Marking

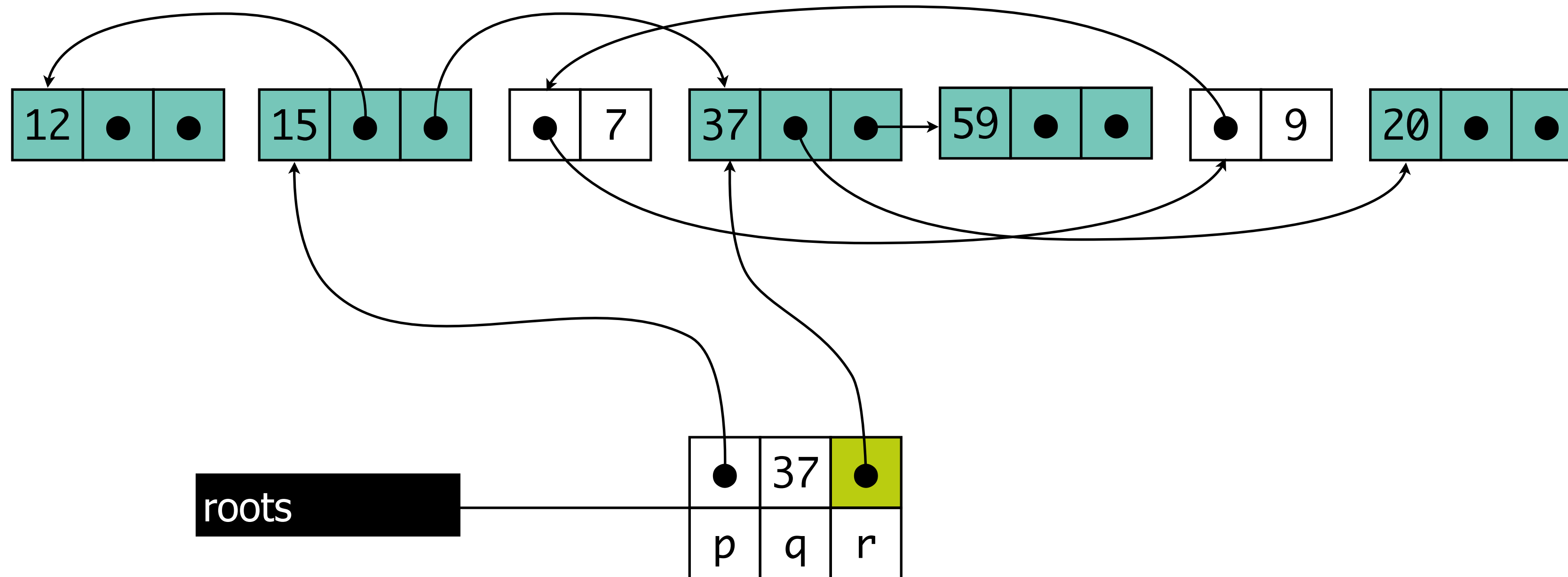


# Marking

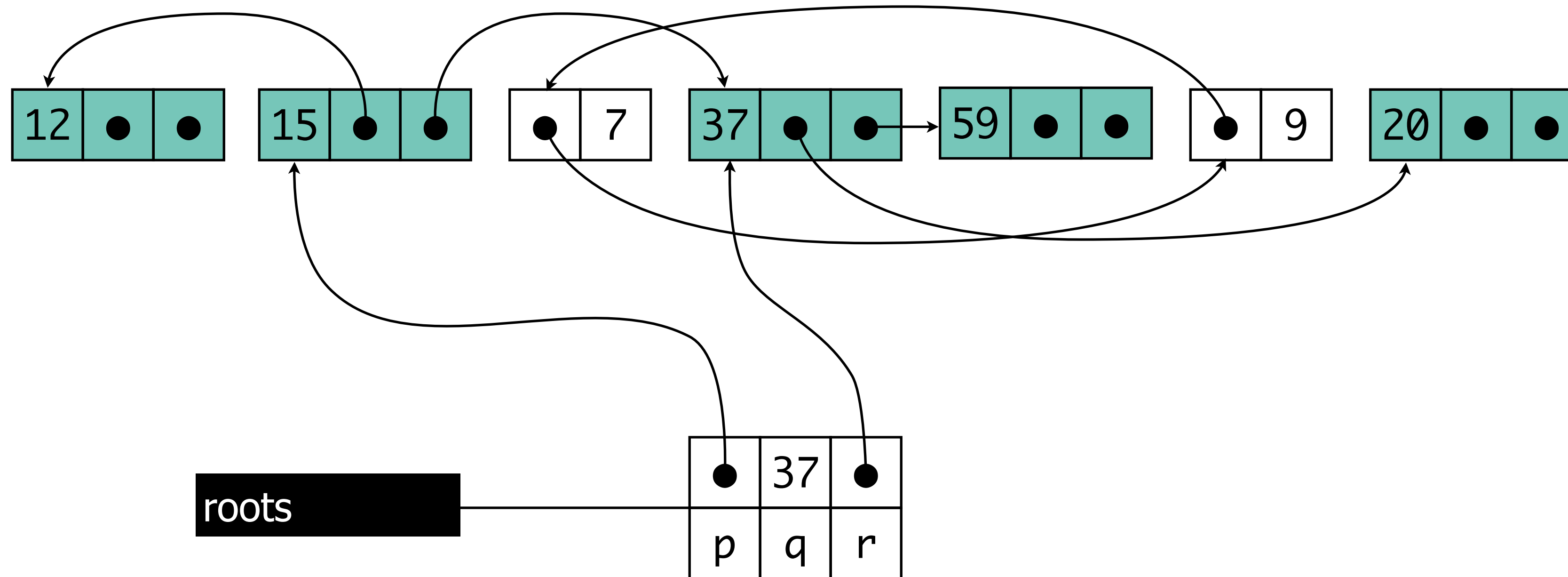




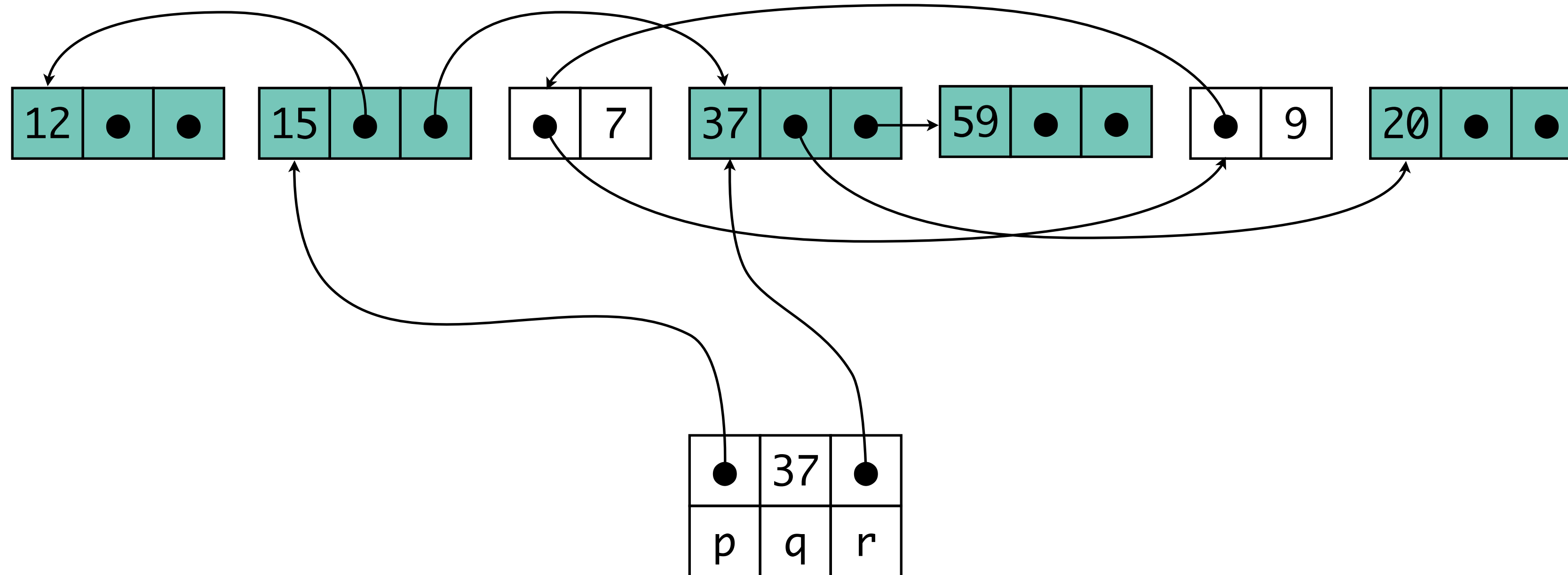
# Marking



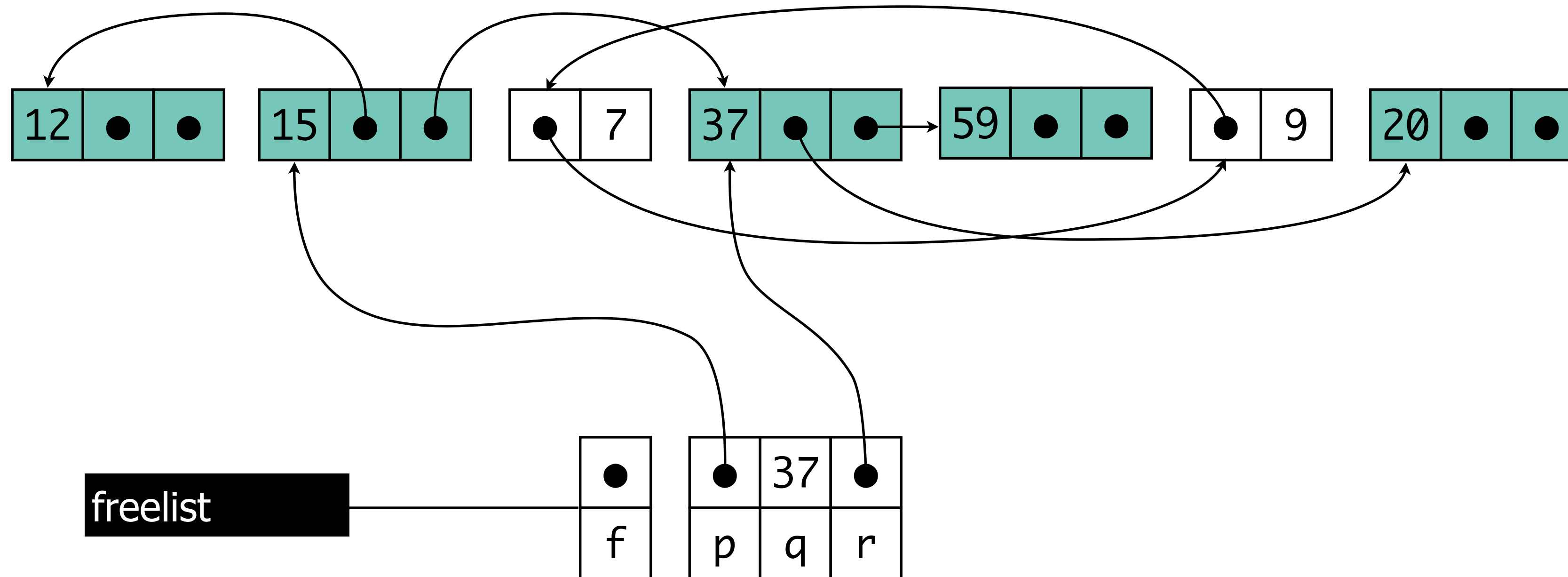
# Marking



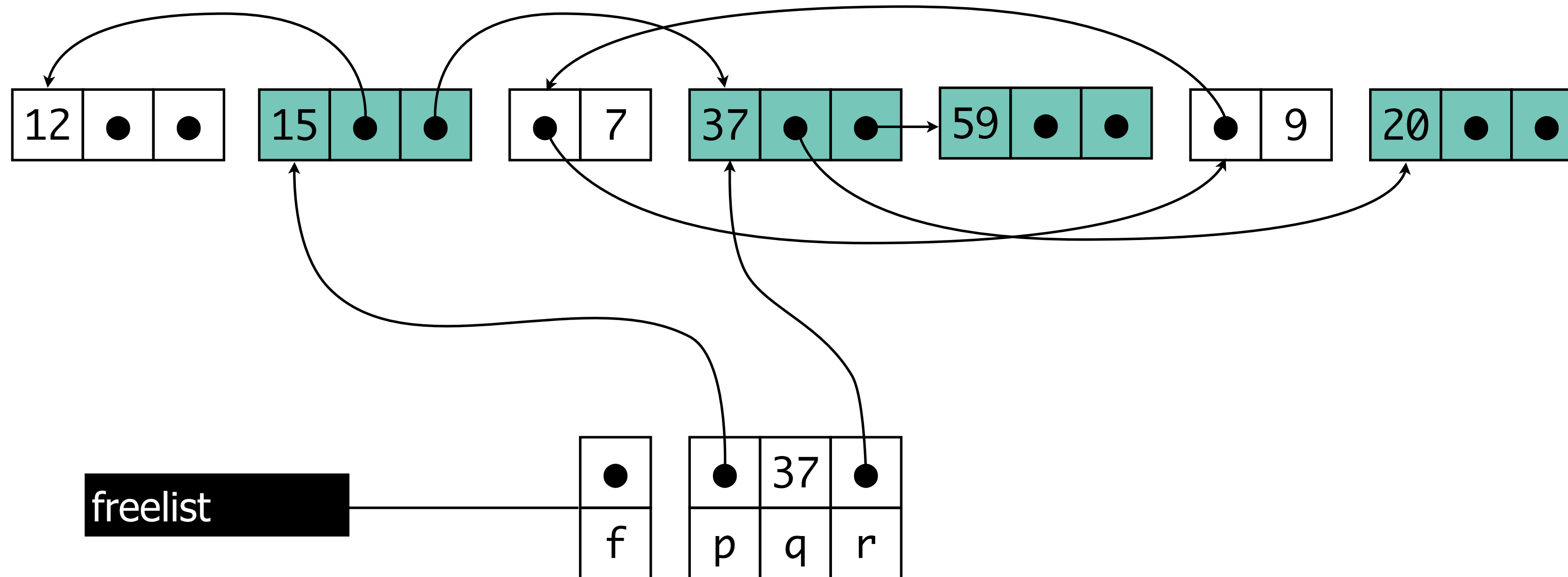
# Sweeping



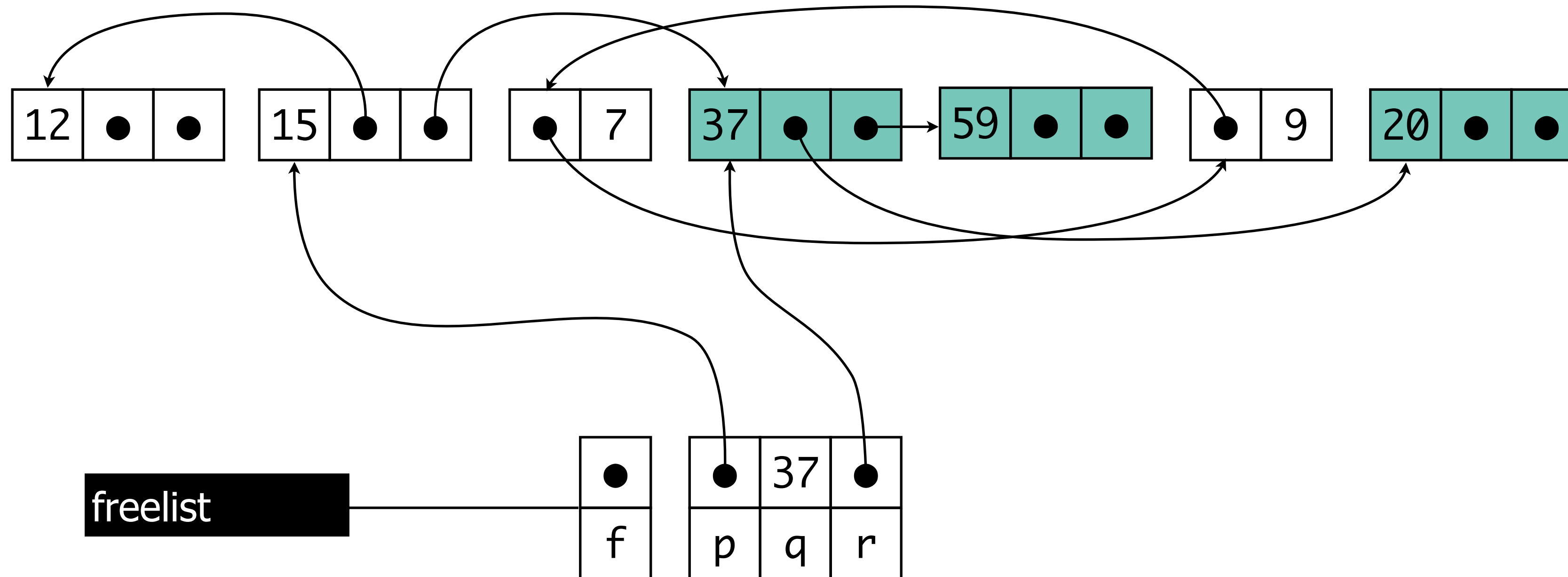
# Sweeping



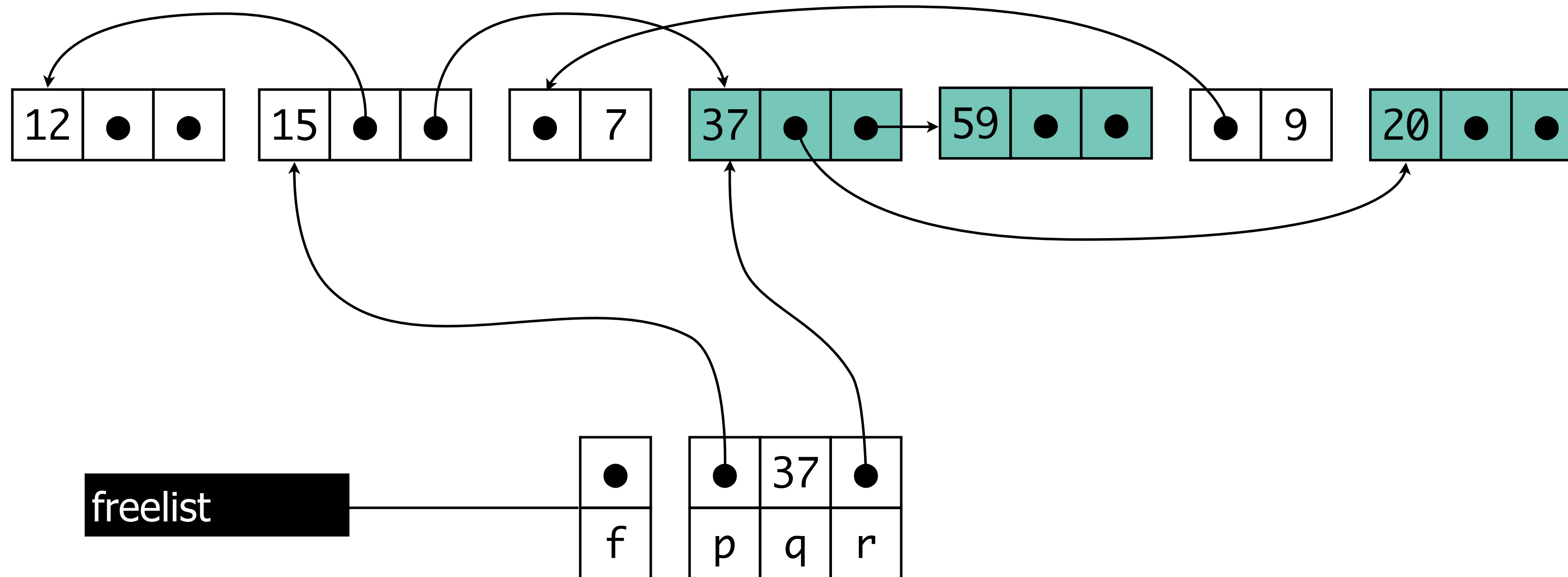
# Sweeping



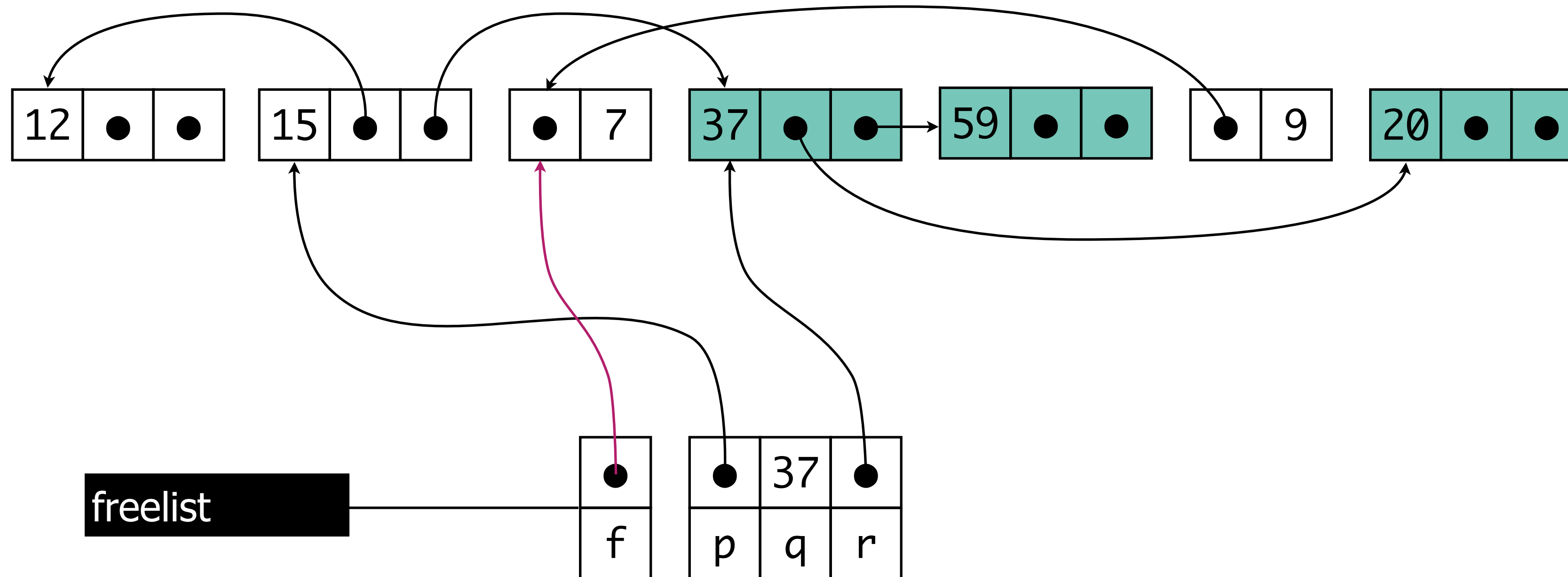
# Sweeping



# Sweeping

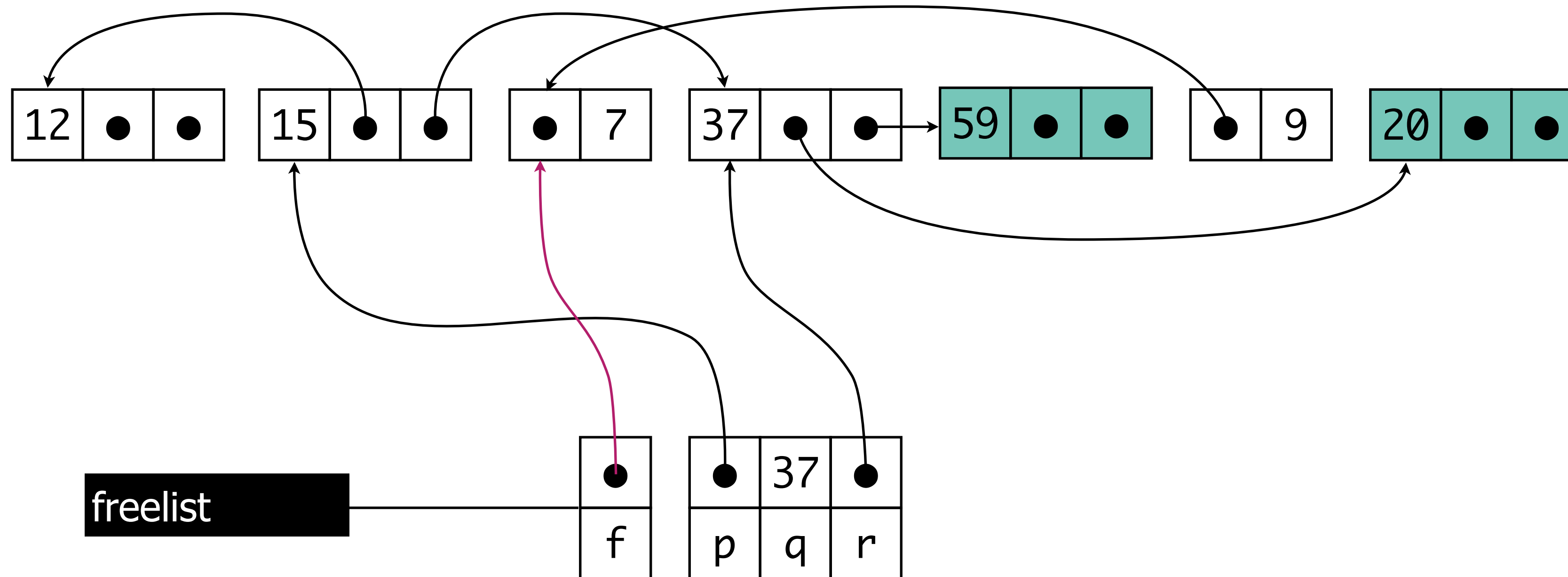


# Sweeping

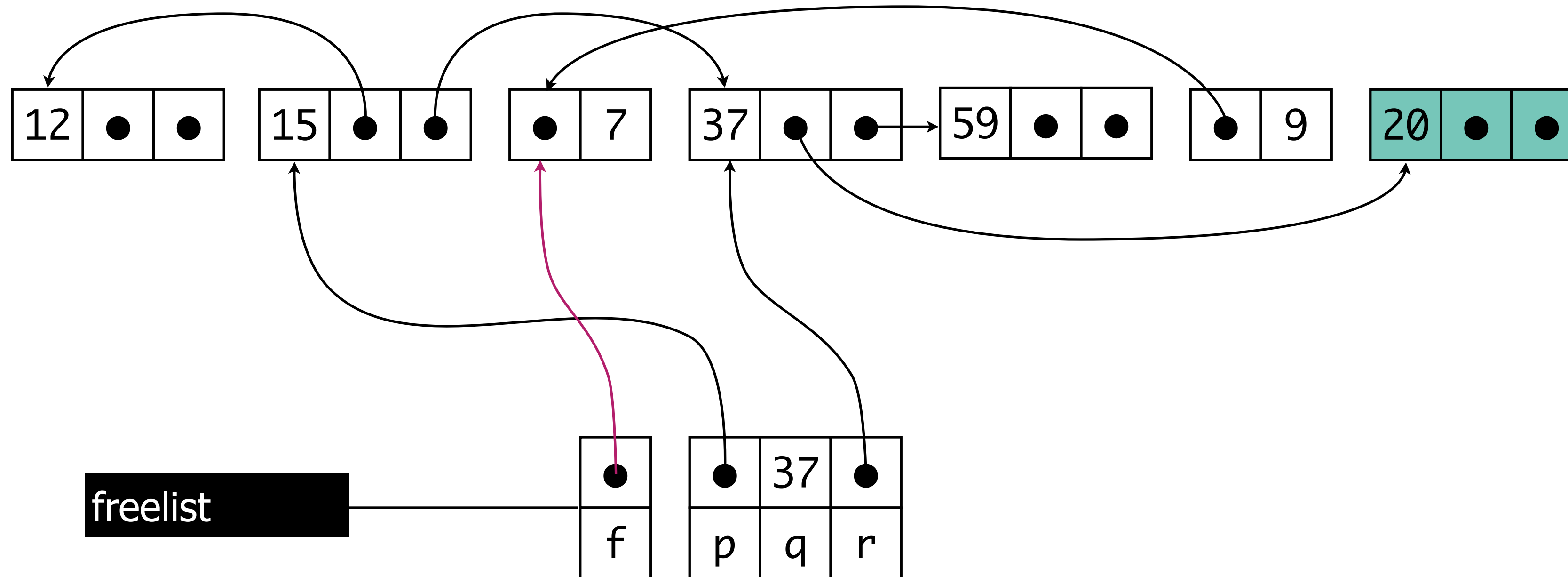




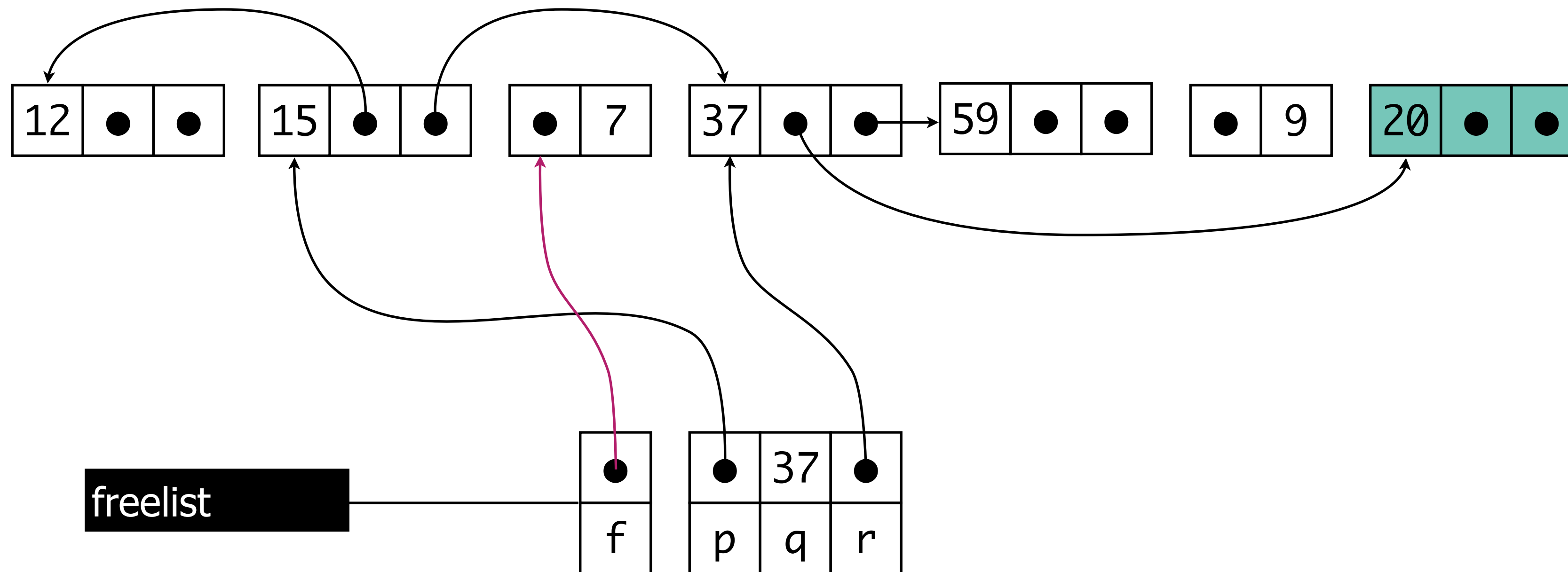
# Sweeping



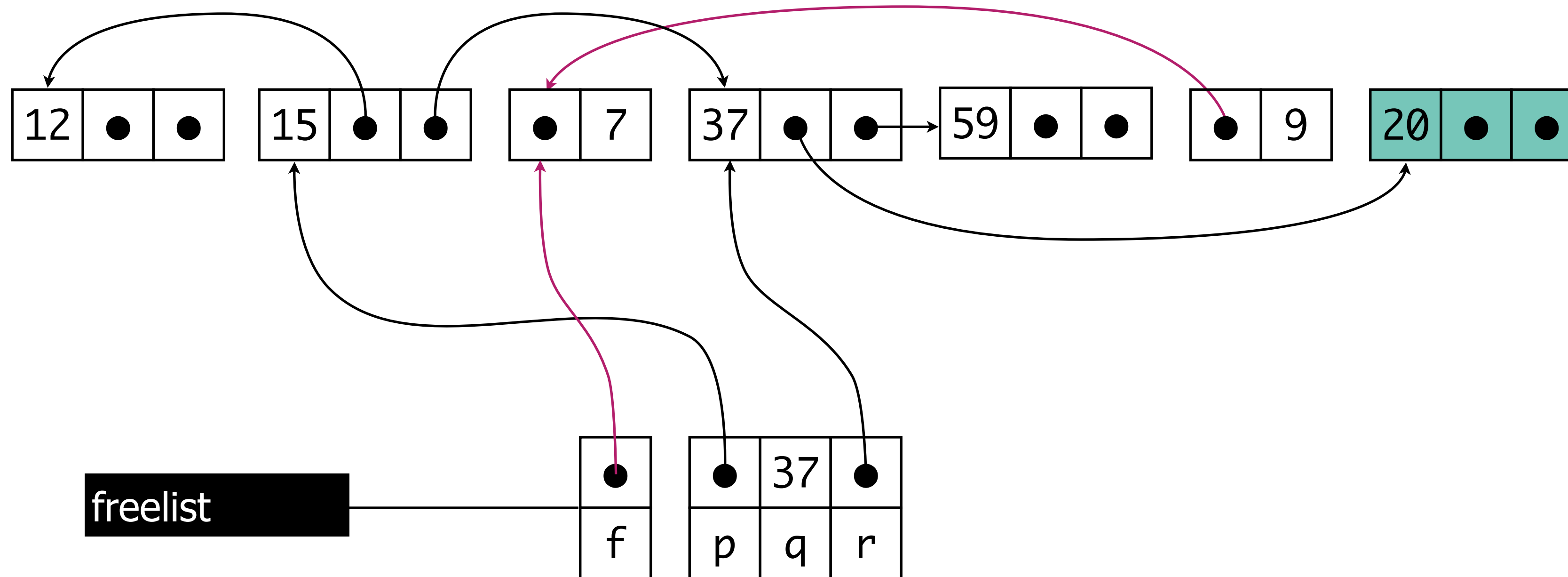
# Sweeping



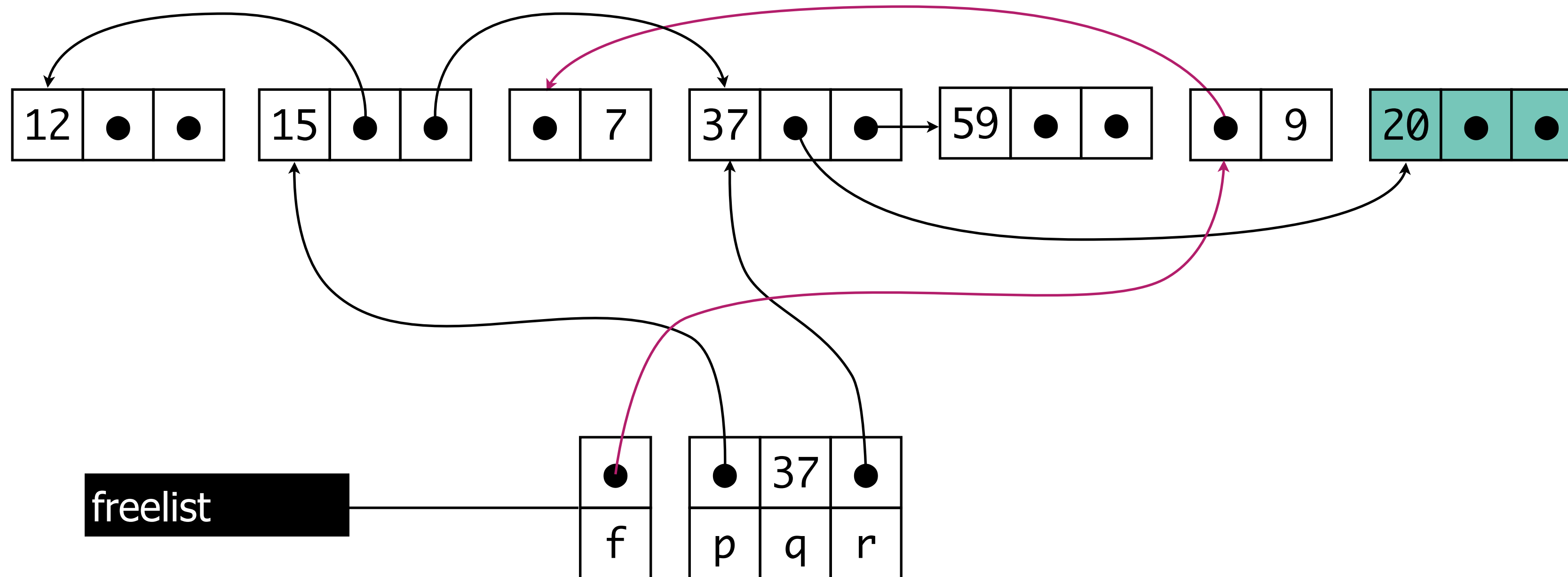
# Sweeping



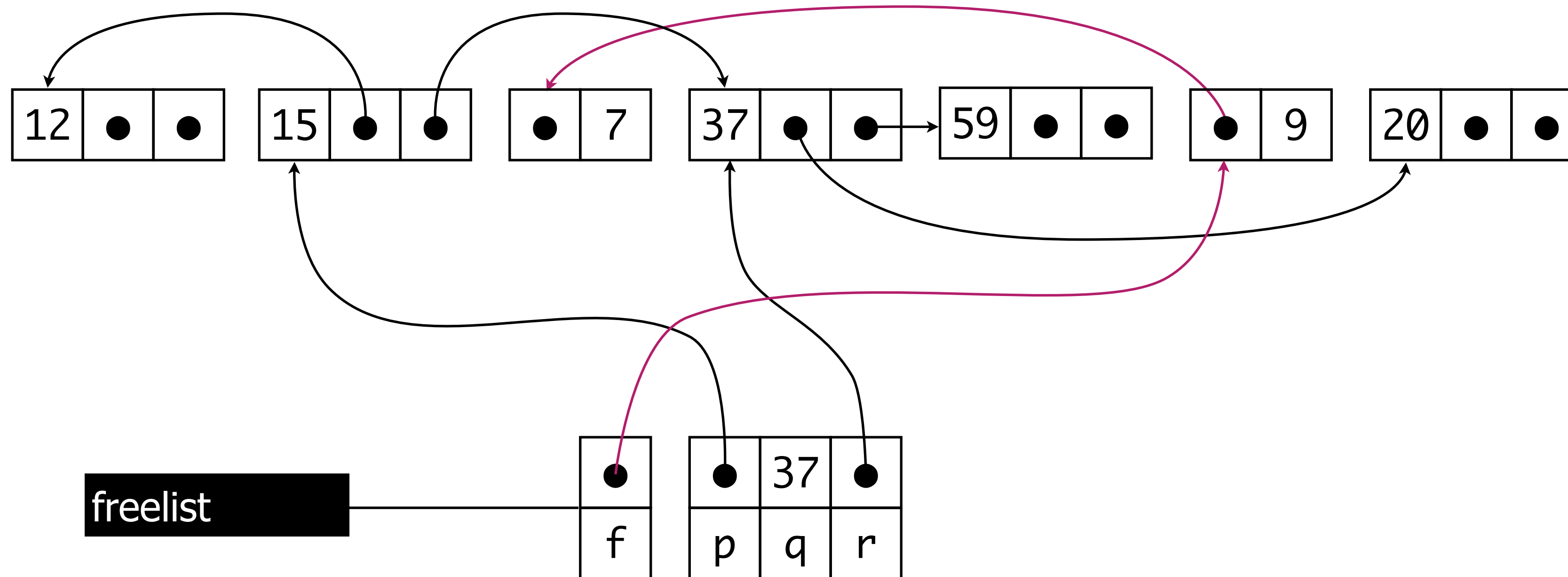
# Sweeping



# Sweeping



# Sweeping



# Mark & Sweep: Algorithms

```
function DFS(x)
  if pointer(x) & !x.marked
    x.marked := true
    foreach f in fields(x)
      DFS(f)
```

# Mark & Sweep: Algorithms

```
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  $(H \gg R)$  cost per allocated word  $\sim c2$

# Mark & Sweep: Costs

## Memory

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

## Measures

- explicit stack
- pointer reversal

# Marking: DFS with Explicit Stack: Algorithms

# Marking: DFS with Explicit Stack: Algorithms

```
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

```
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
```

marking without memory overhead

# Mark & Sweep

## 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

# Copying Collection

# Copying Collection: Idea

## Spaces

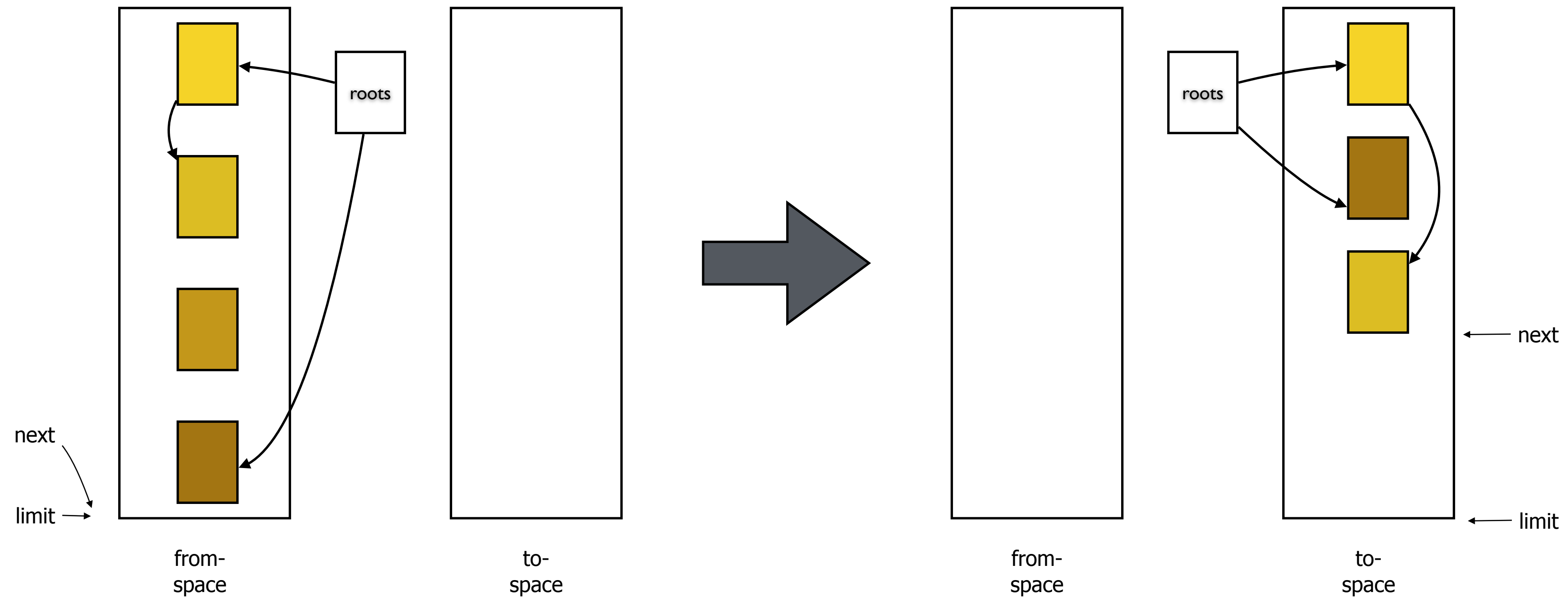
- fromspace & tospace
- switch roles after copy

## Copy

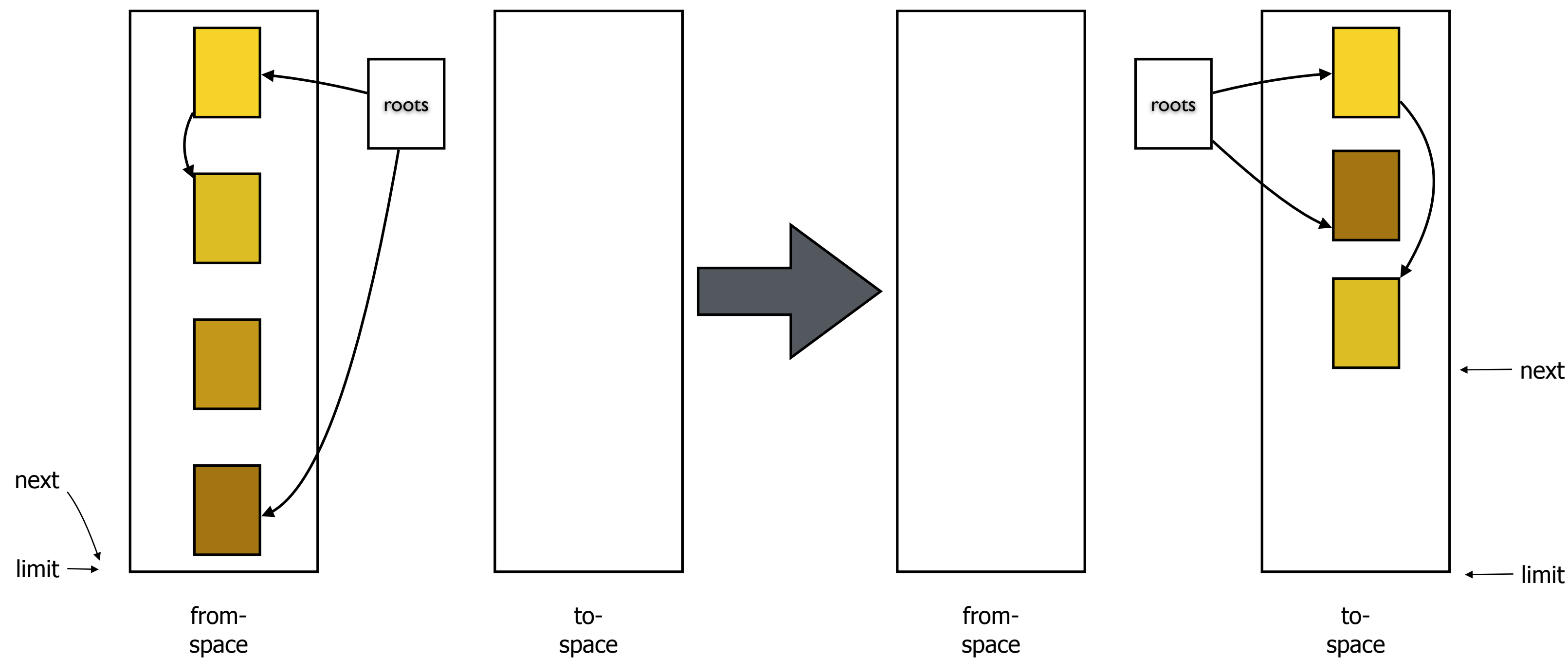
- traverse reachability graph
- copy from fromspace to tospace
- fromspace unreachable, free memory
- tospace compact, **no fragmentation**



# Copying Collection: Idea

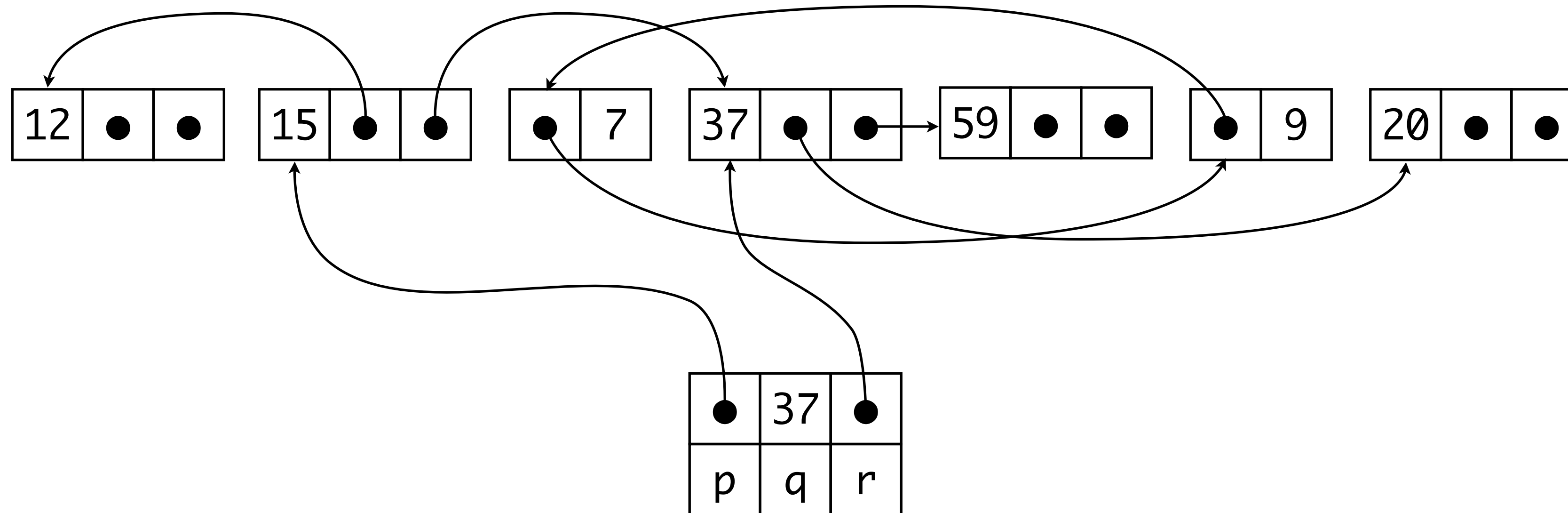


# Copying Collection: Algorithm

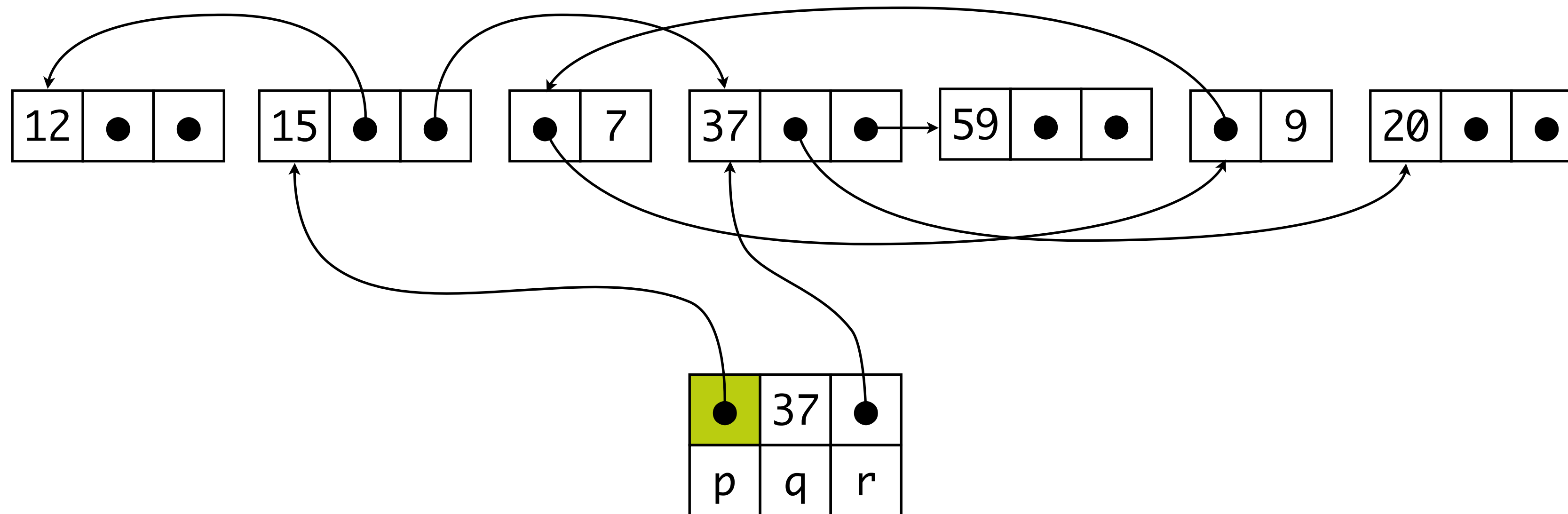


```
function BFS()  
  
  next := scan := start(tospace)  
  
  foreach r in roots  
    r = Forward(r)  
  
  while scan < next  
  
    foreach f in fields of scan  
      scan.f = Forward(scan.f)  
  
    scan = scan + sizeof(scan)
```

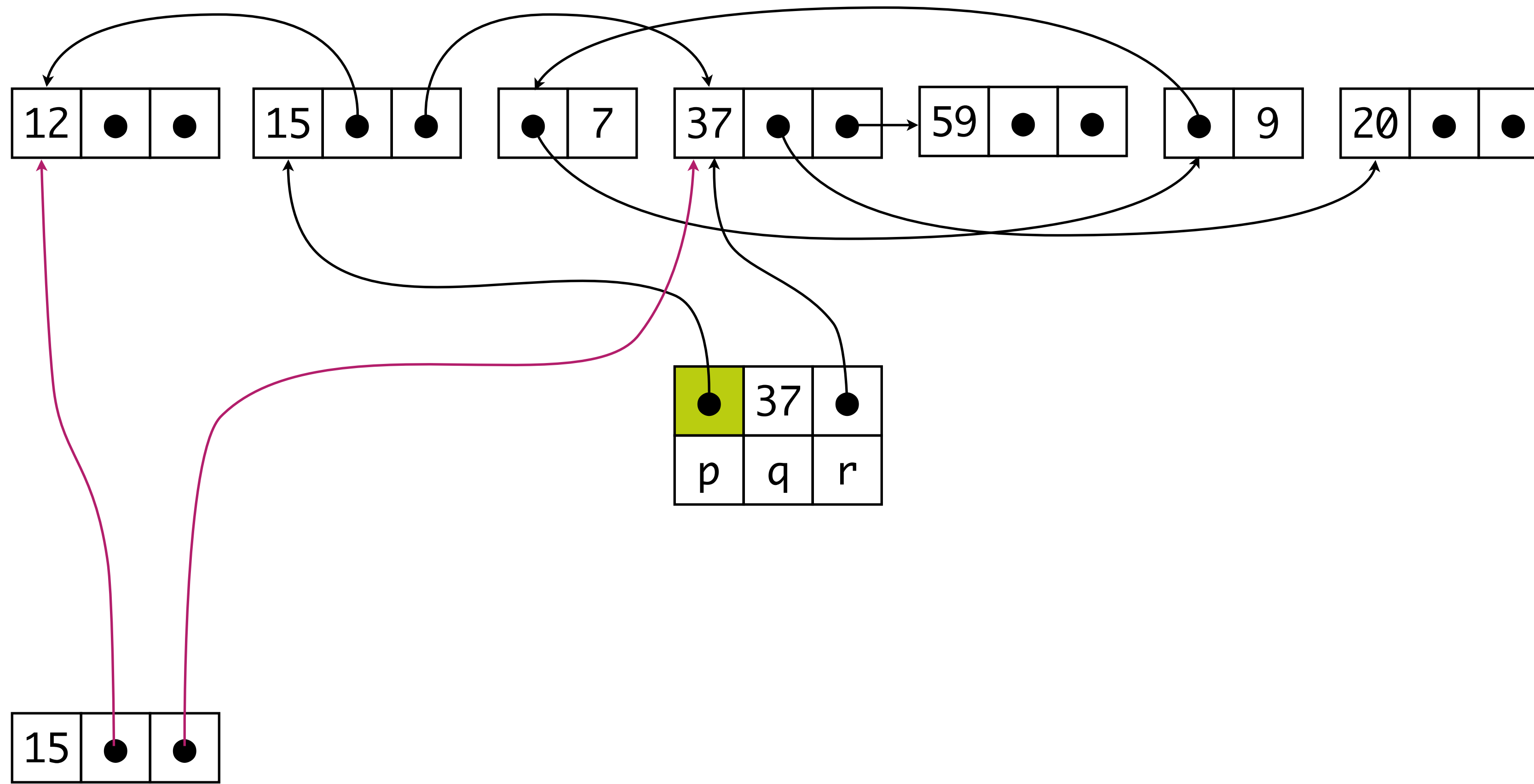
# Copying Collection: Example



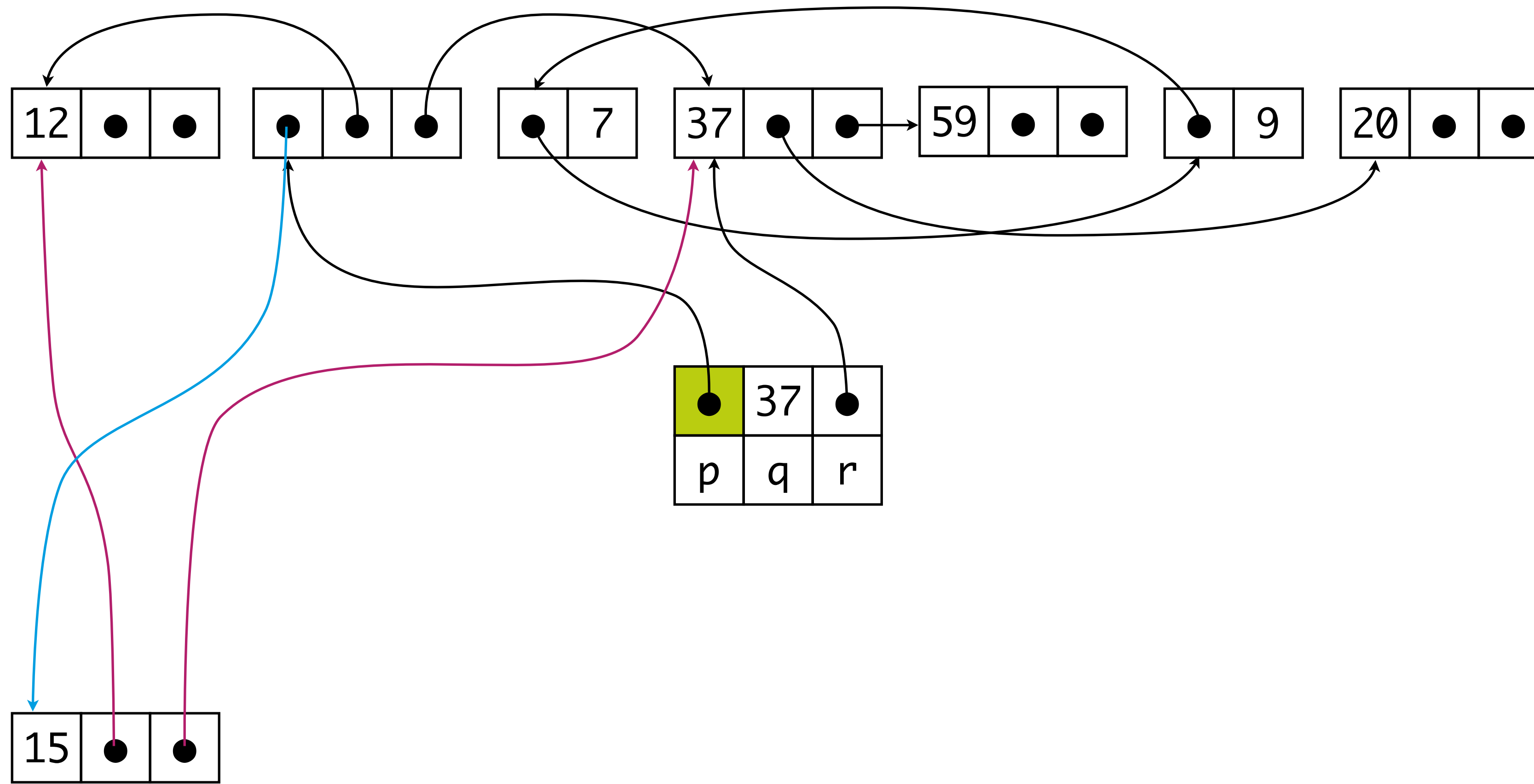
# Copying Collection: Example



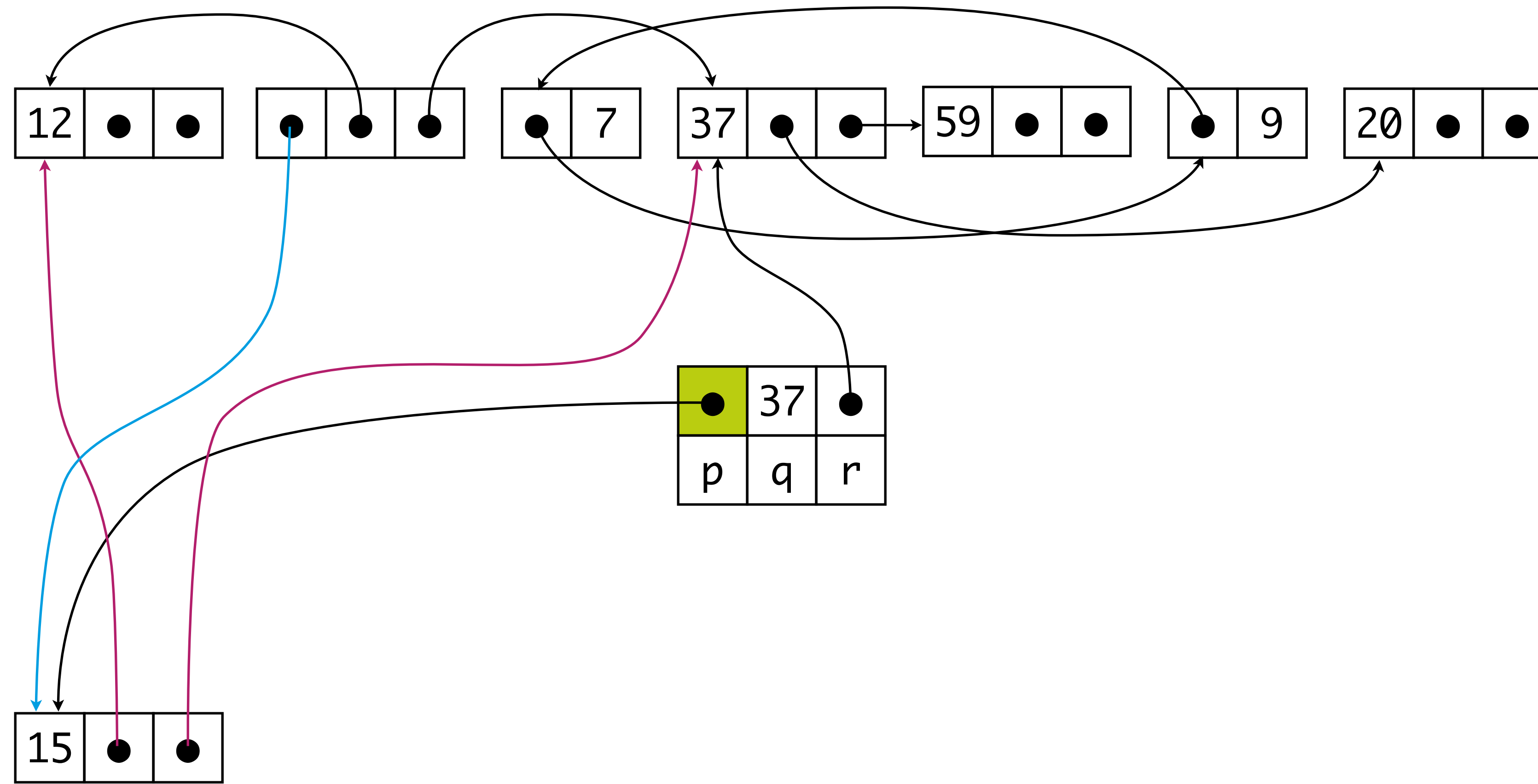
# Copying Collection: Example



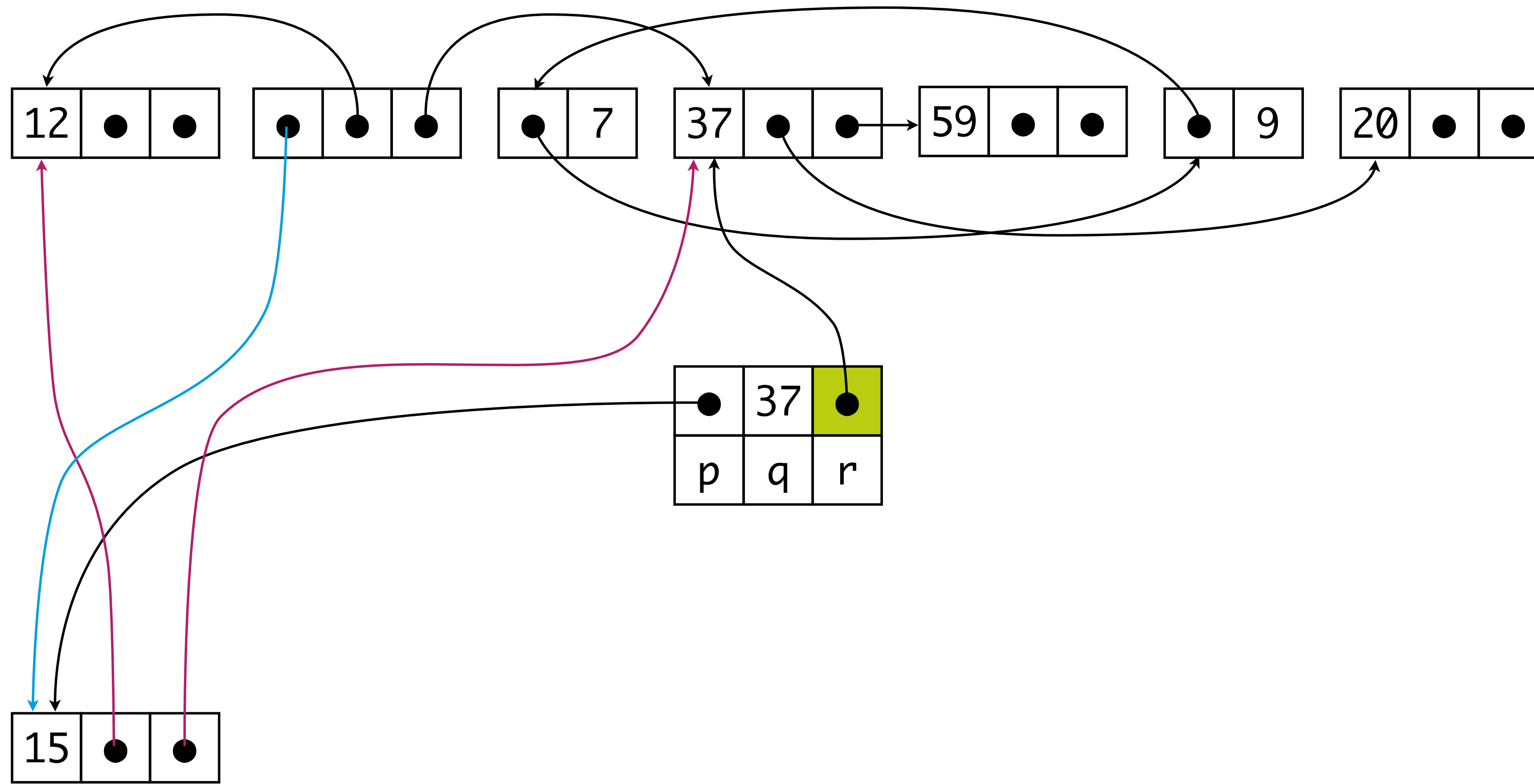
# Copying Collection: Example



# Copying Collection: Example

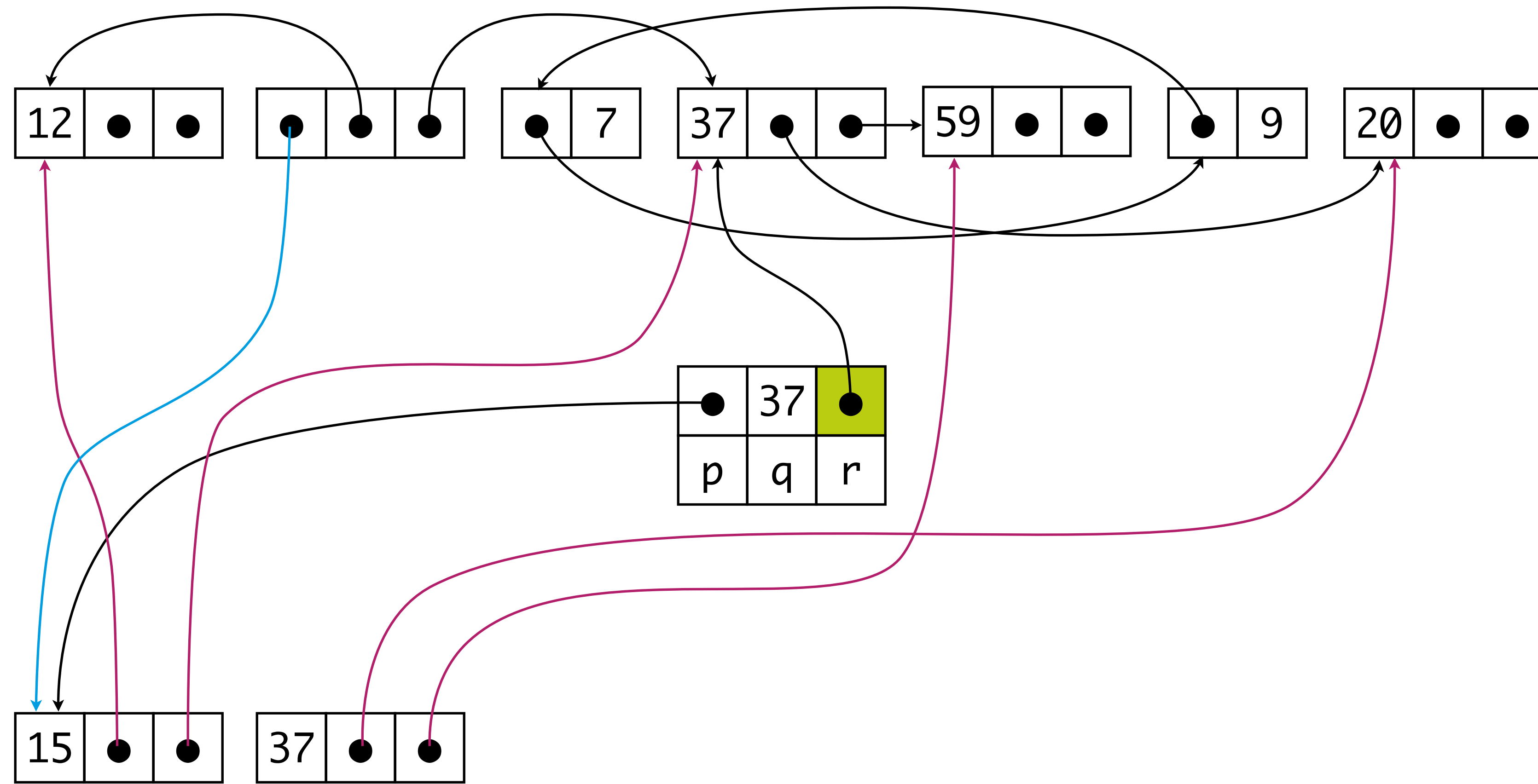


# Copying Collection: Example

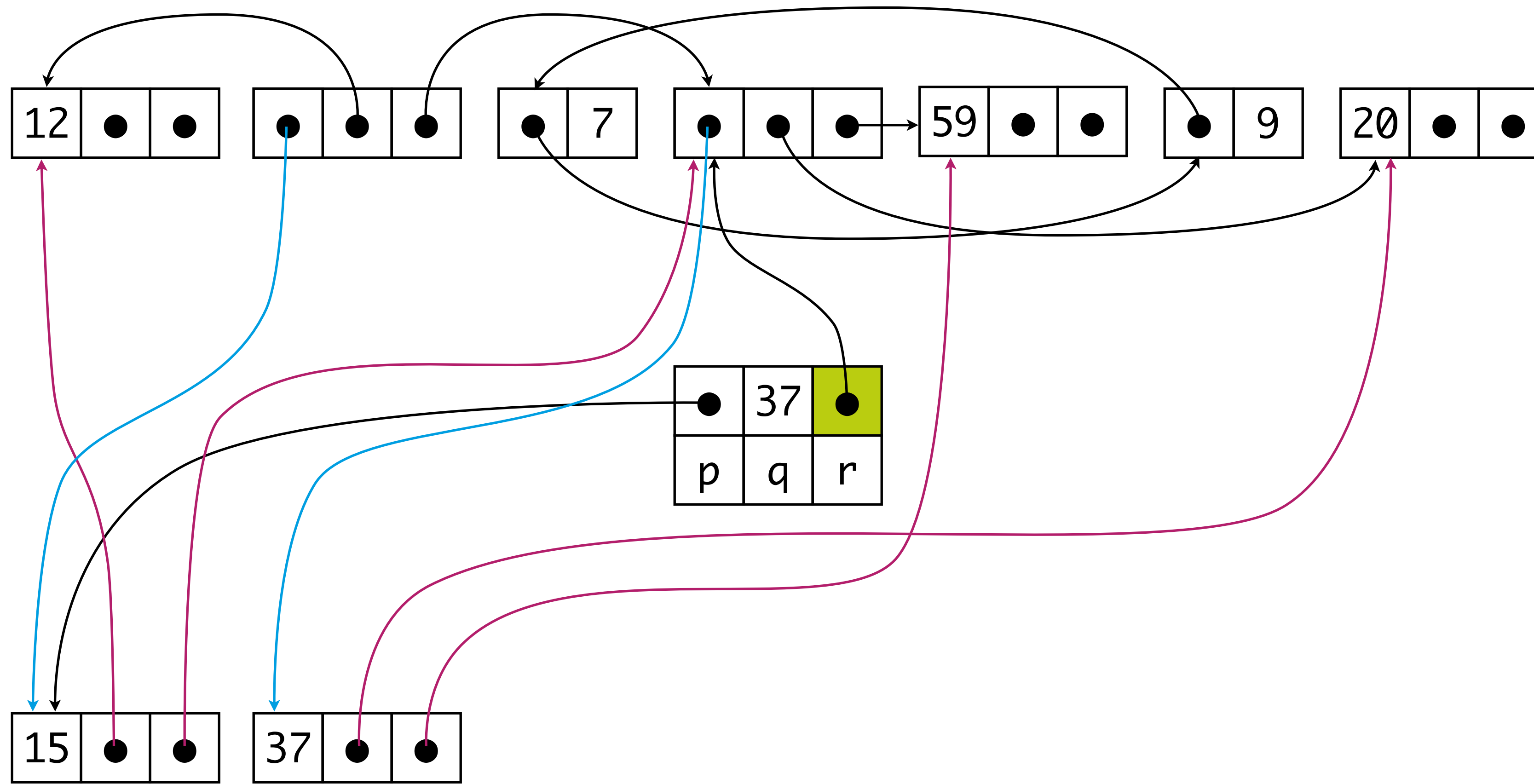




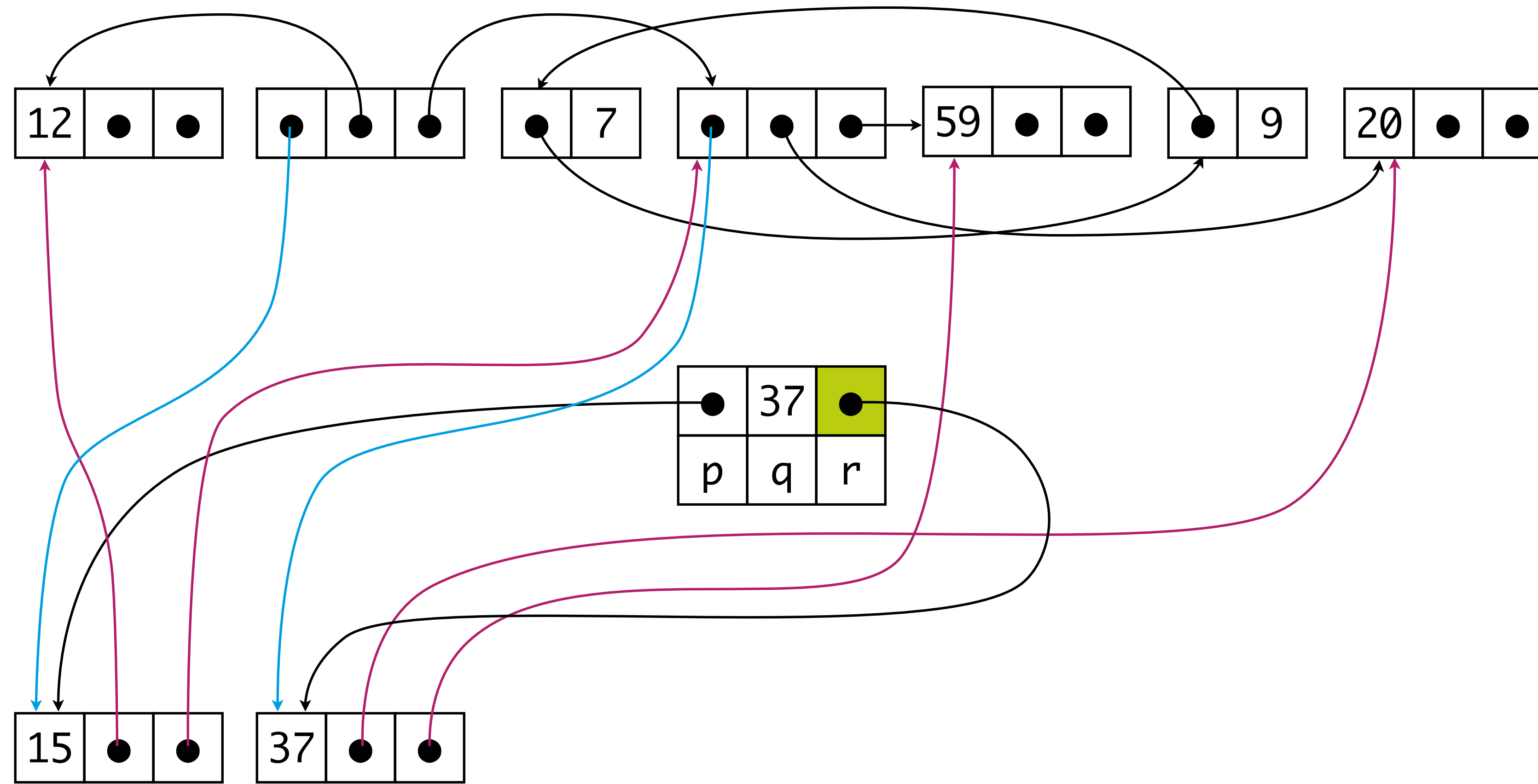
# Copying Collection: Example



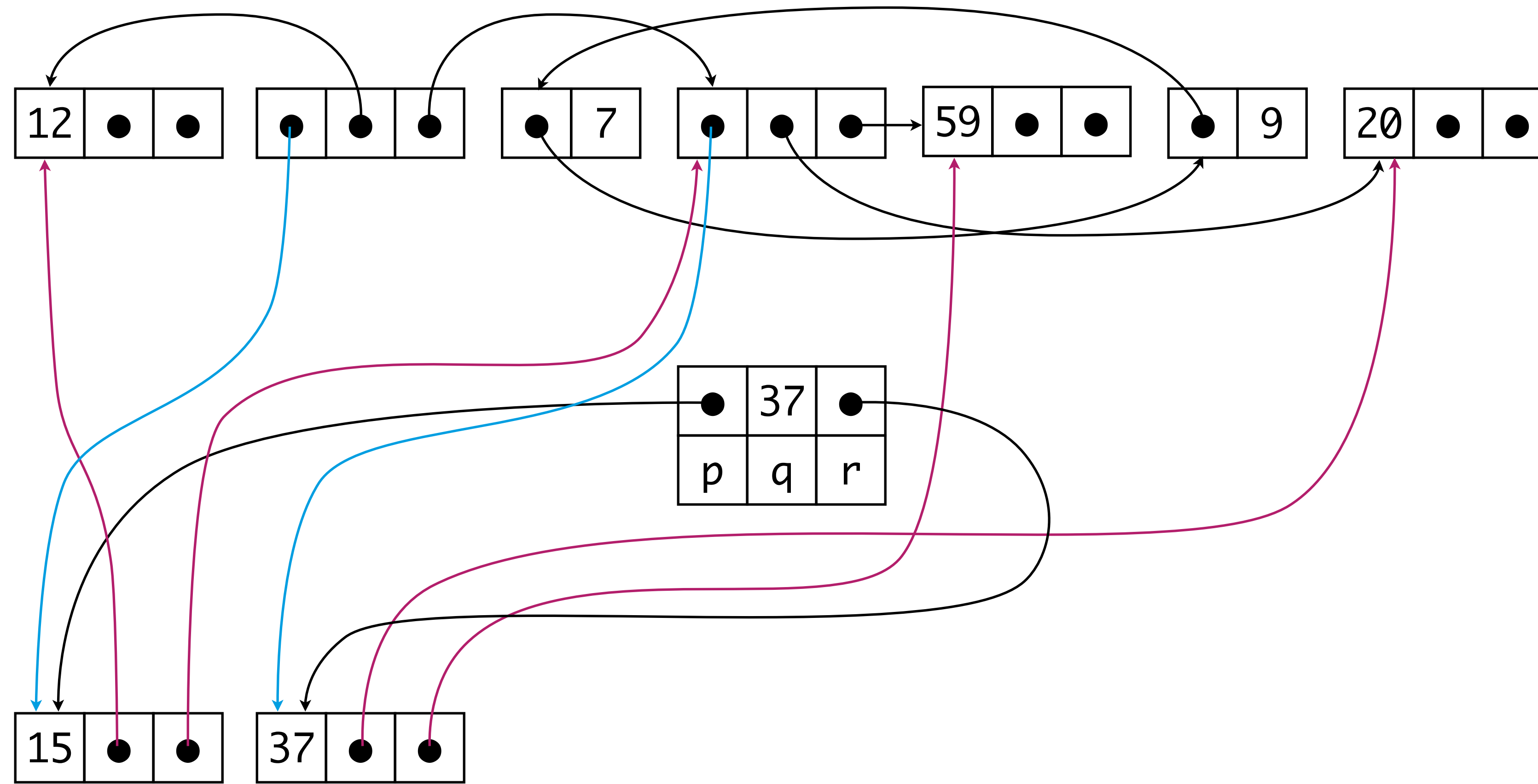
# Copying Collection: Example



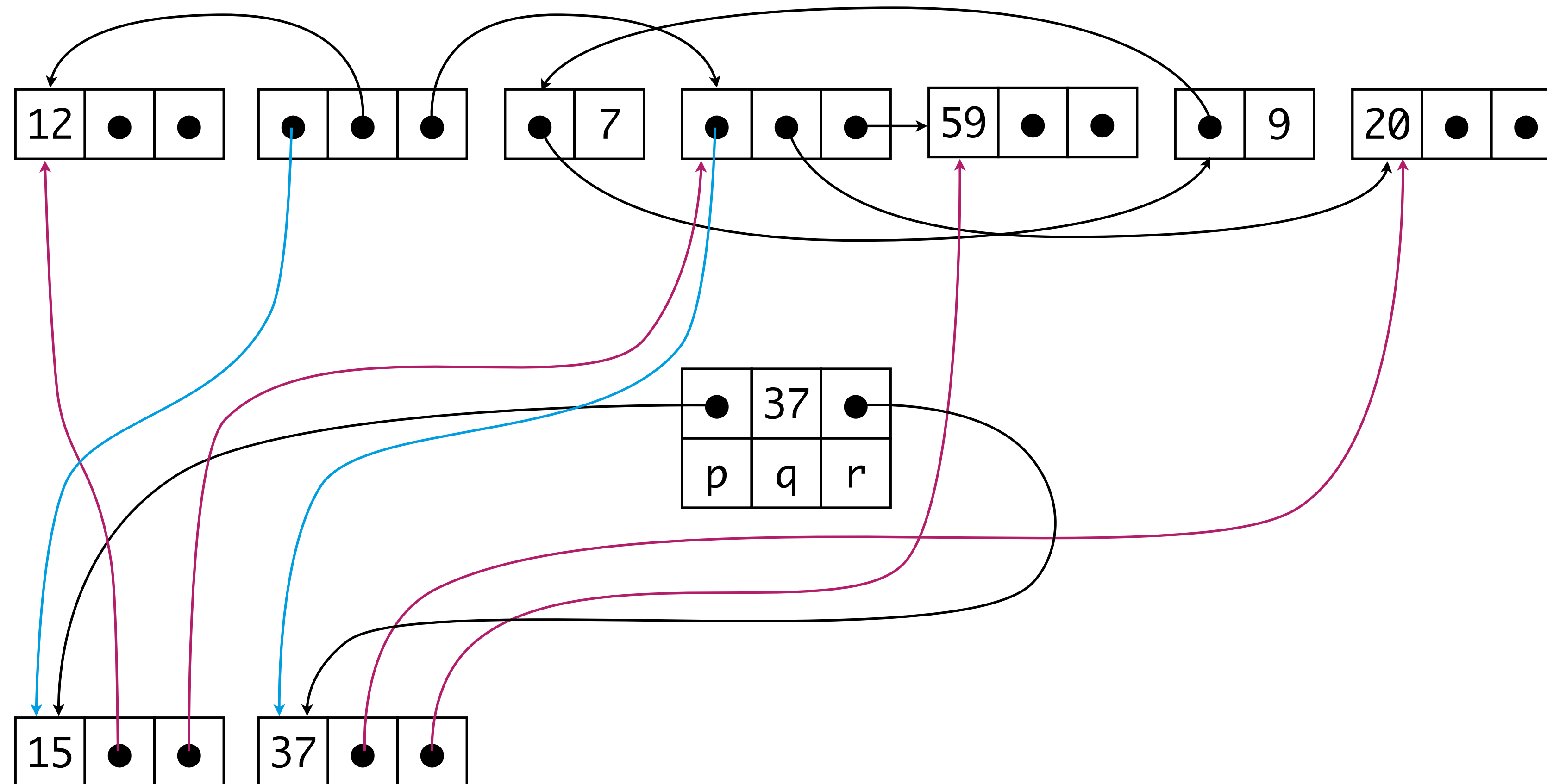
# Copying Collection: Example



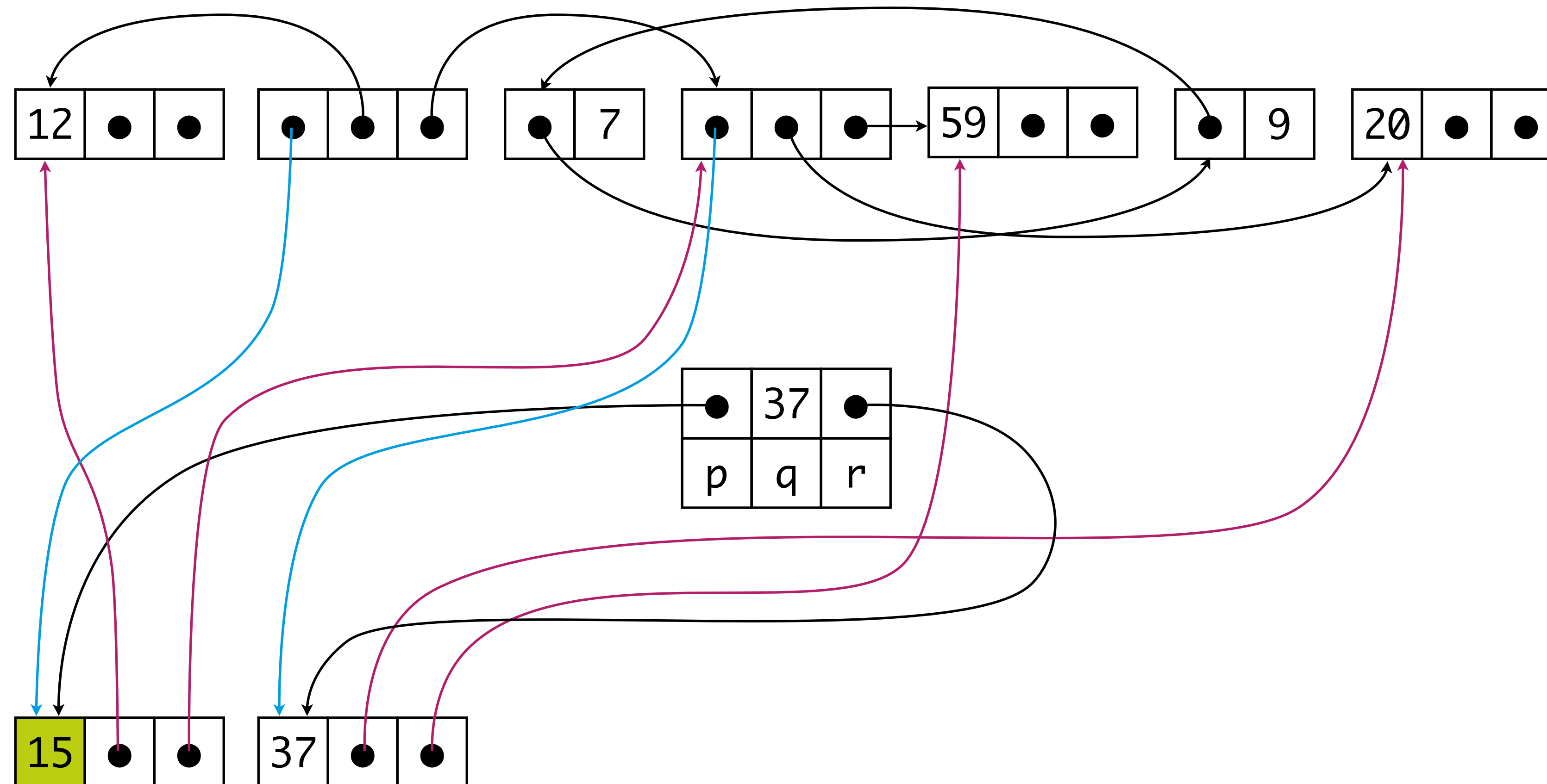
# Copying Collection: Example



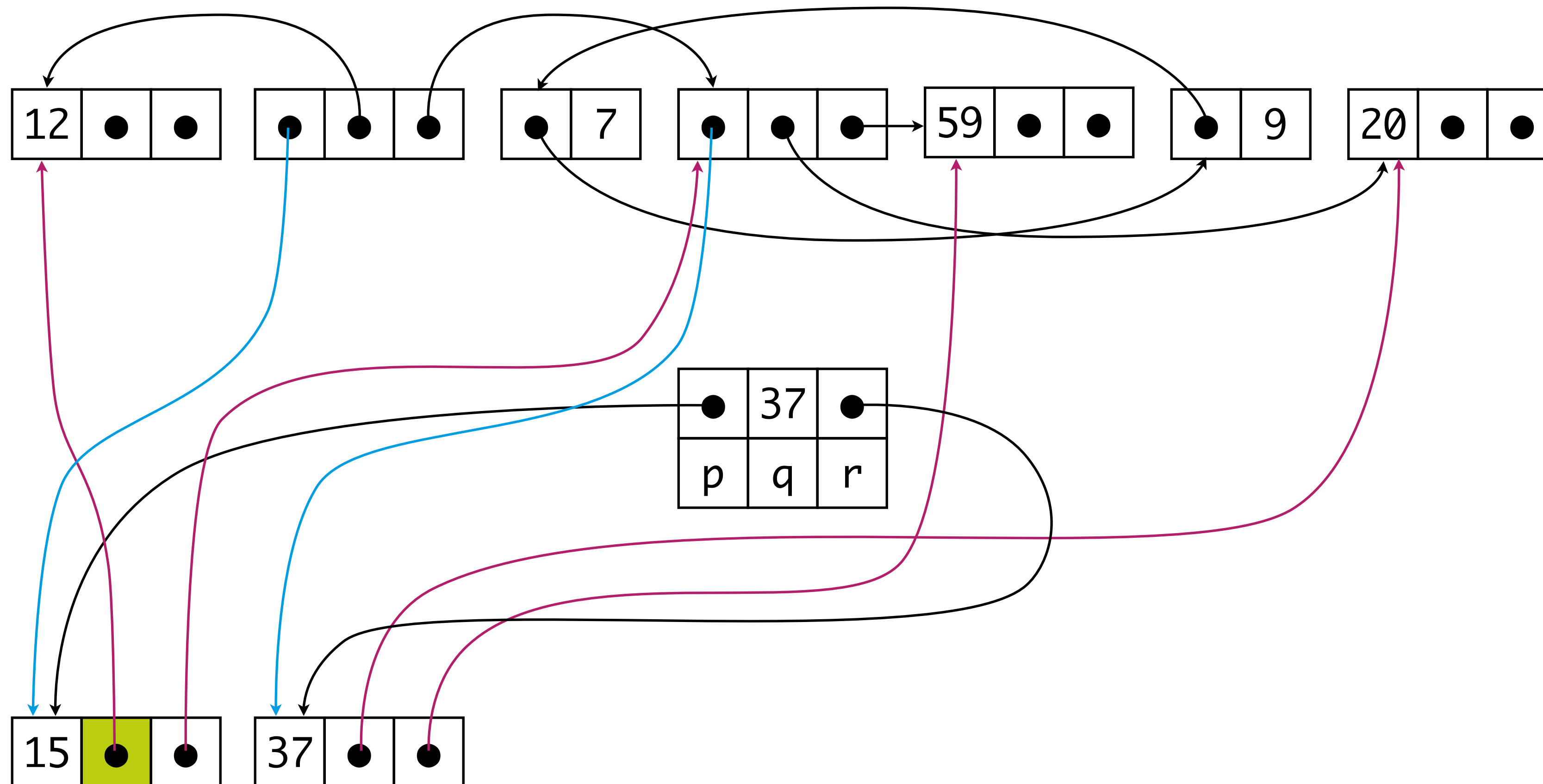
# Copying Collection: Example



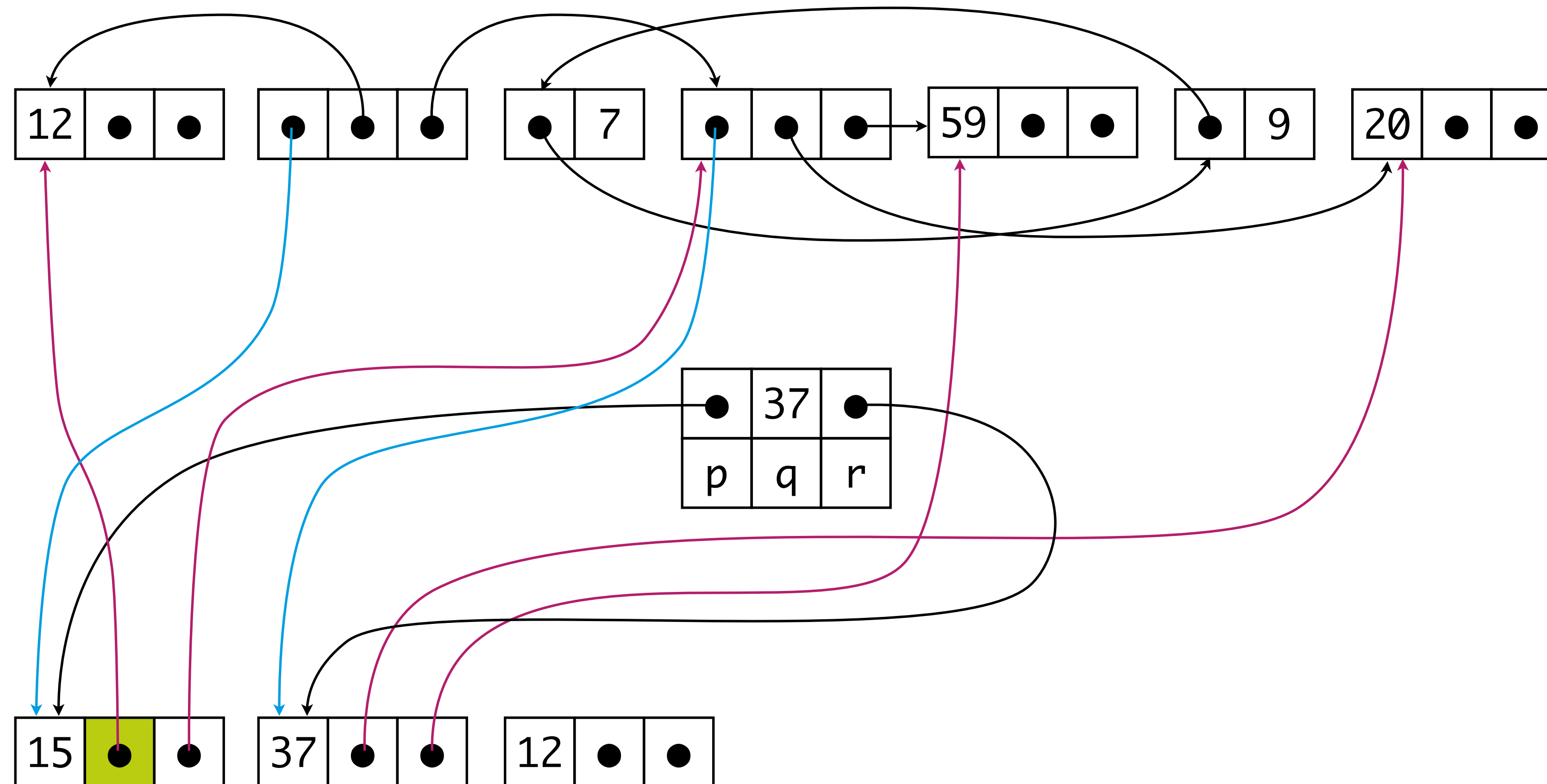
# Copying Collection: Example



# Copying Collection: Example

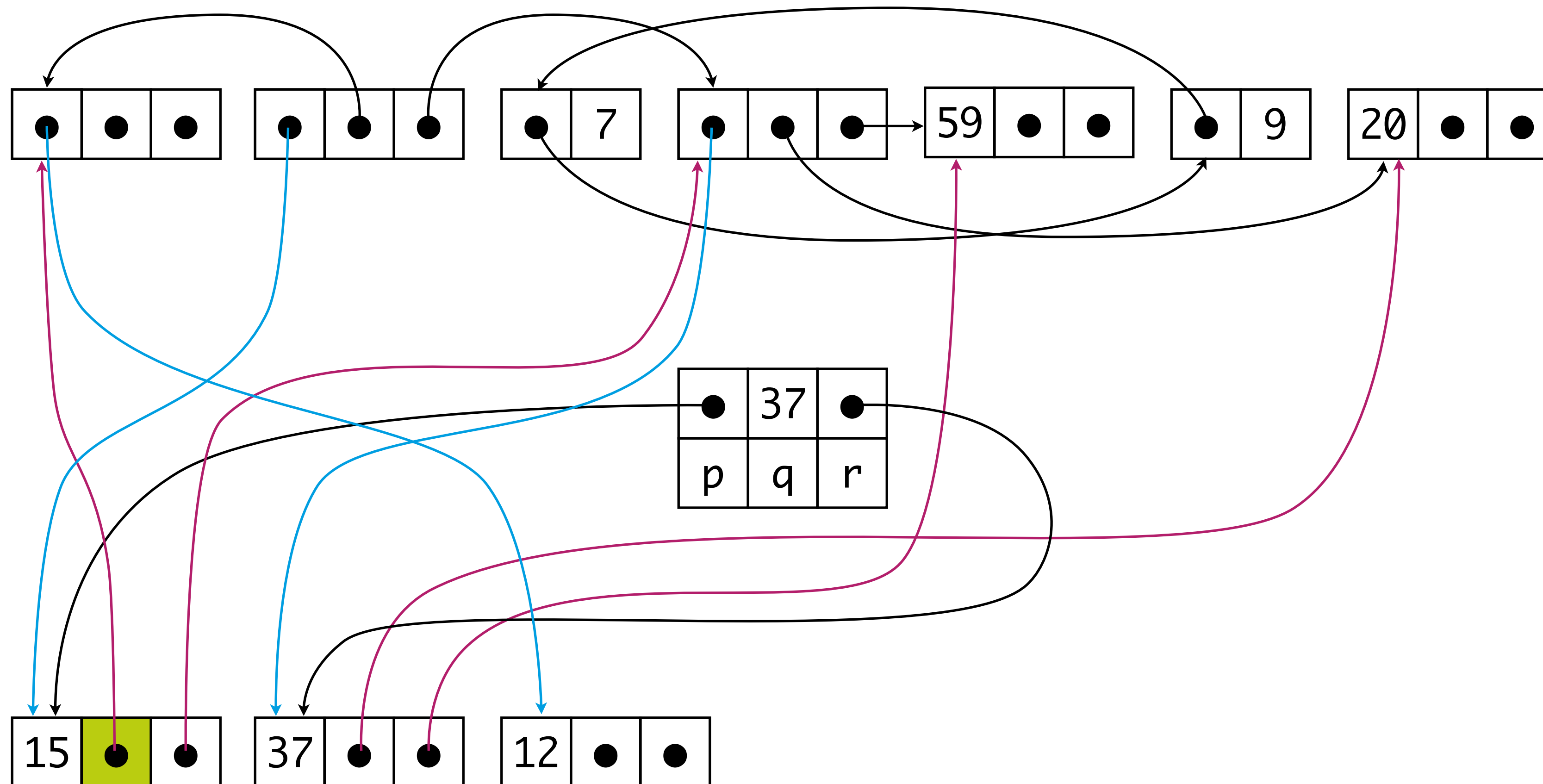


# Copying Collection: Example

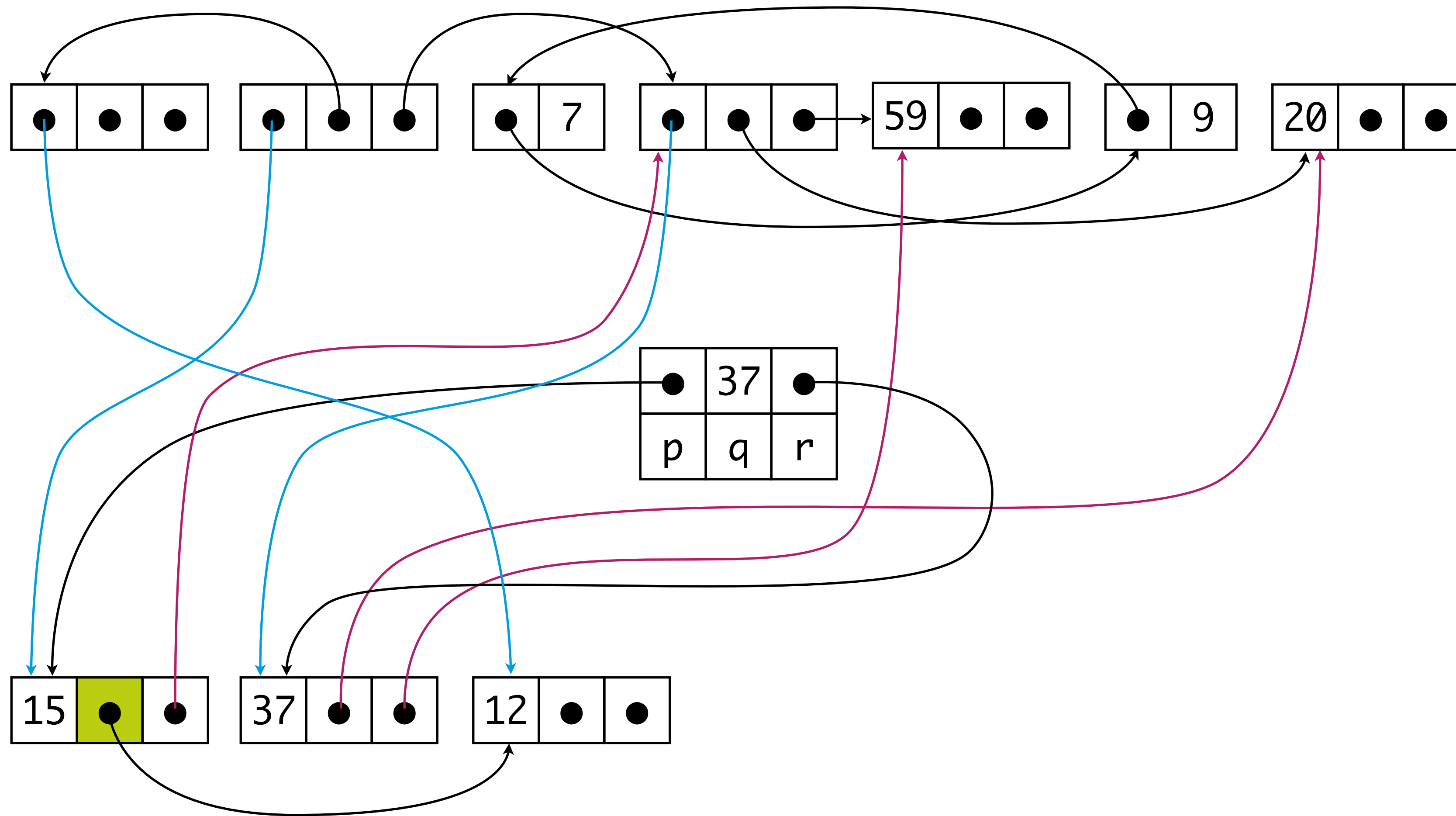




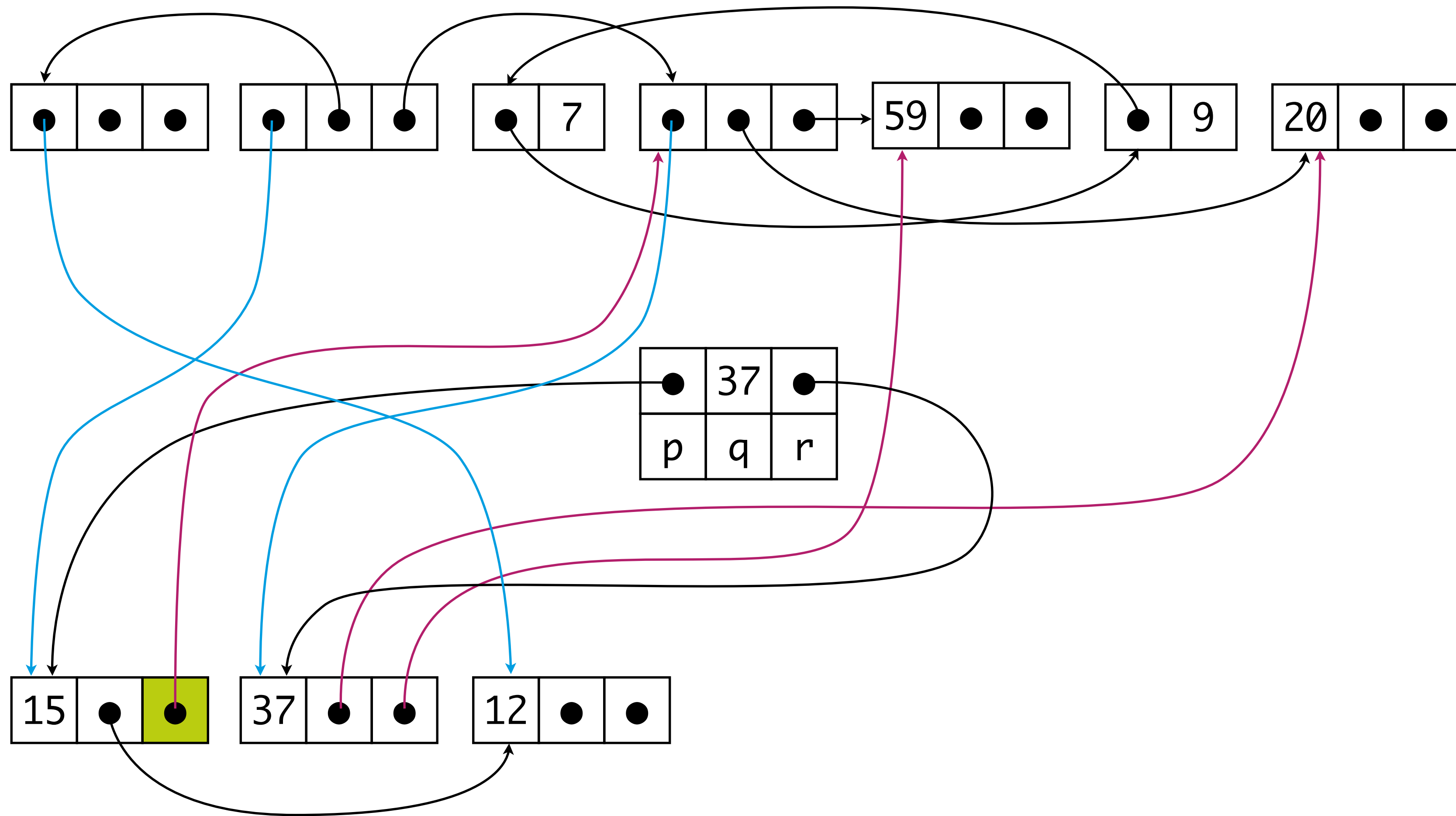
# Copying Collection: Example



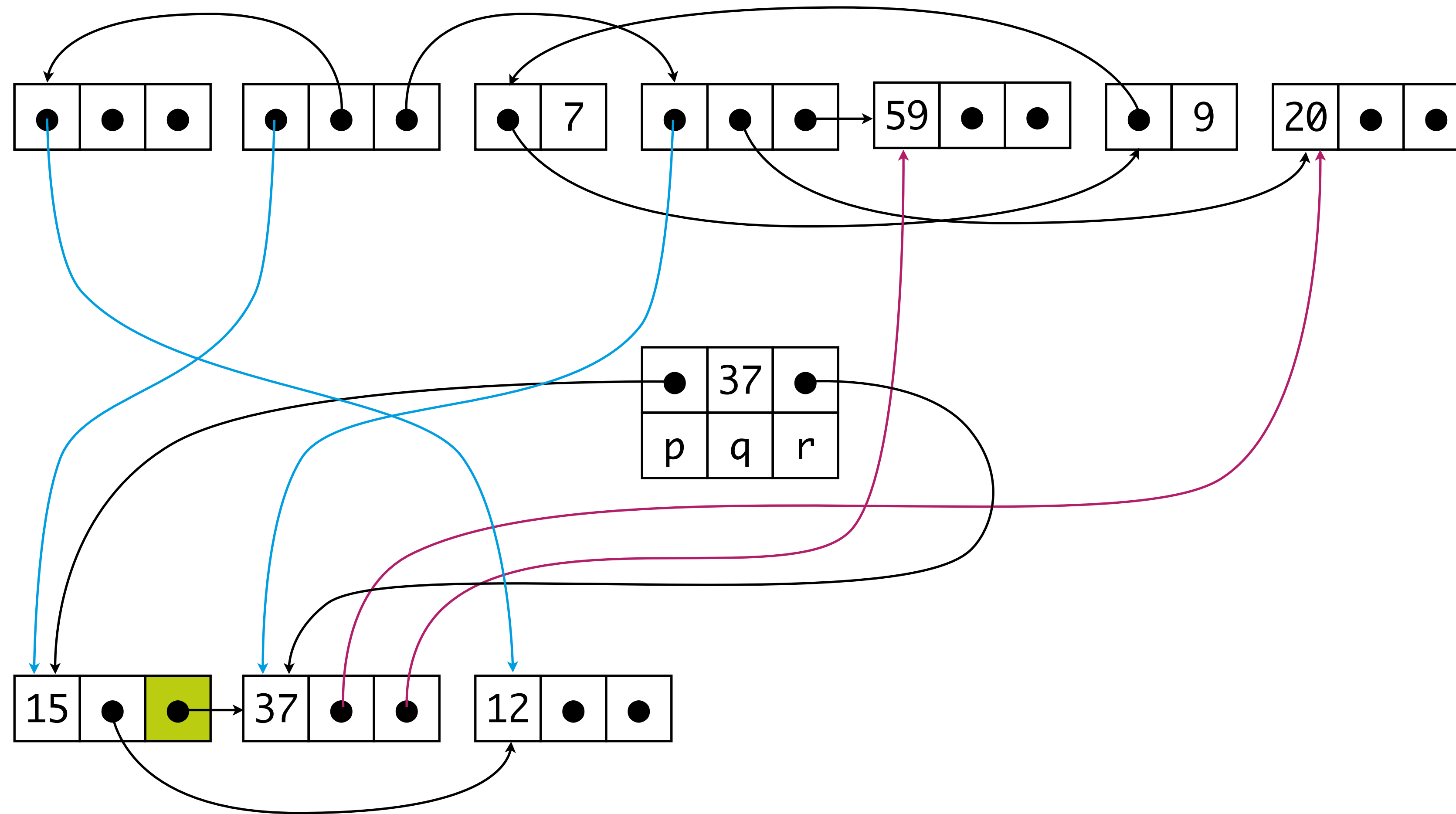
# Copying Collection: Example



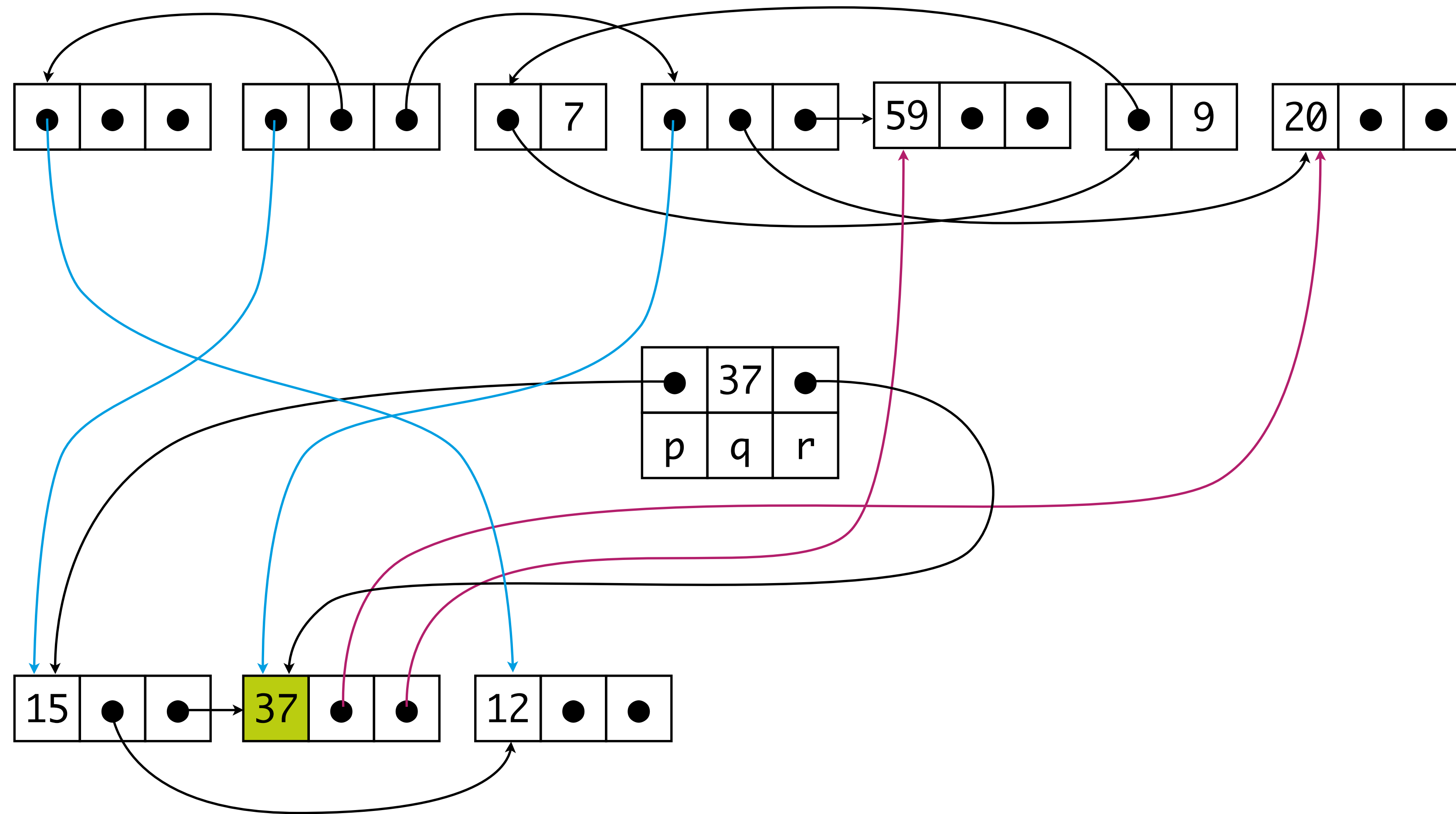
# Copying Collection: Example



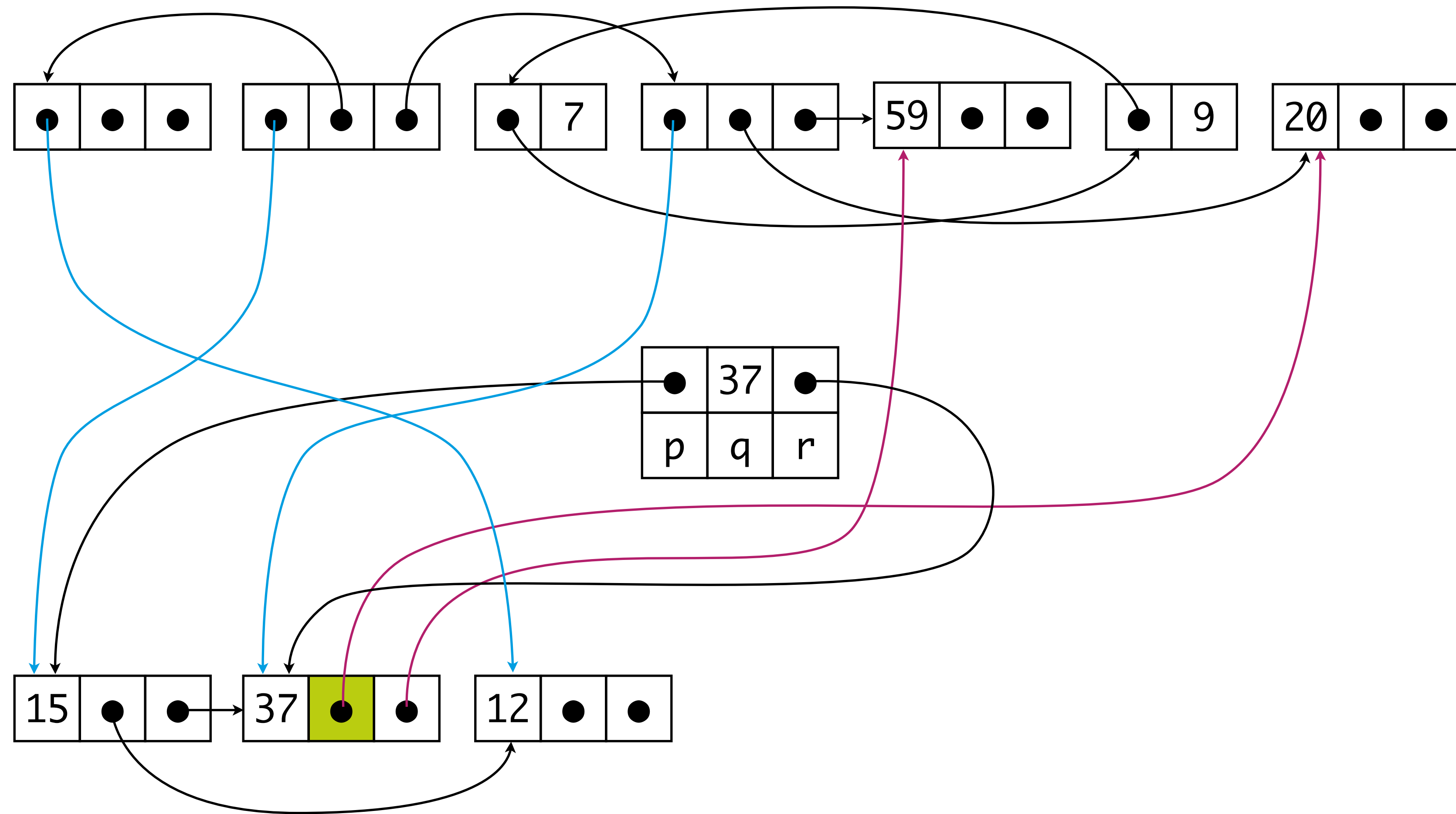
# Copying Collection: Example



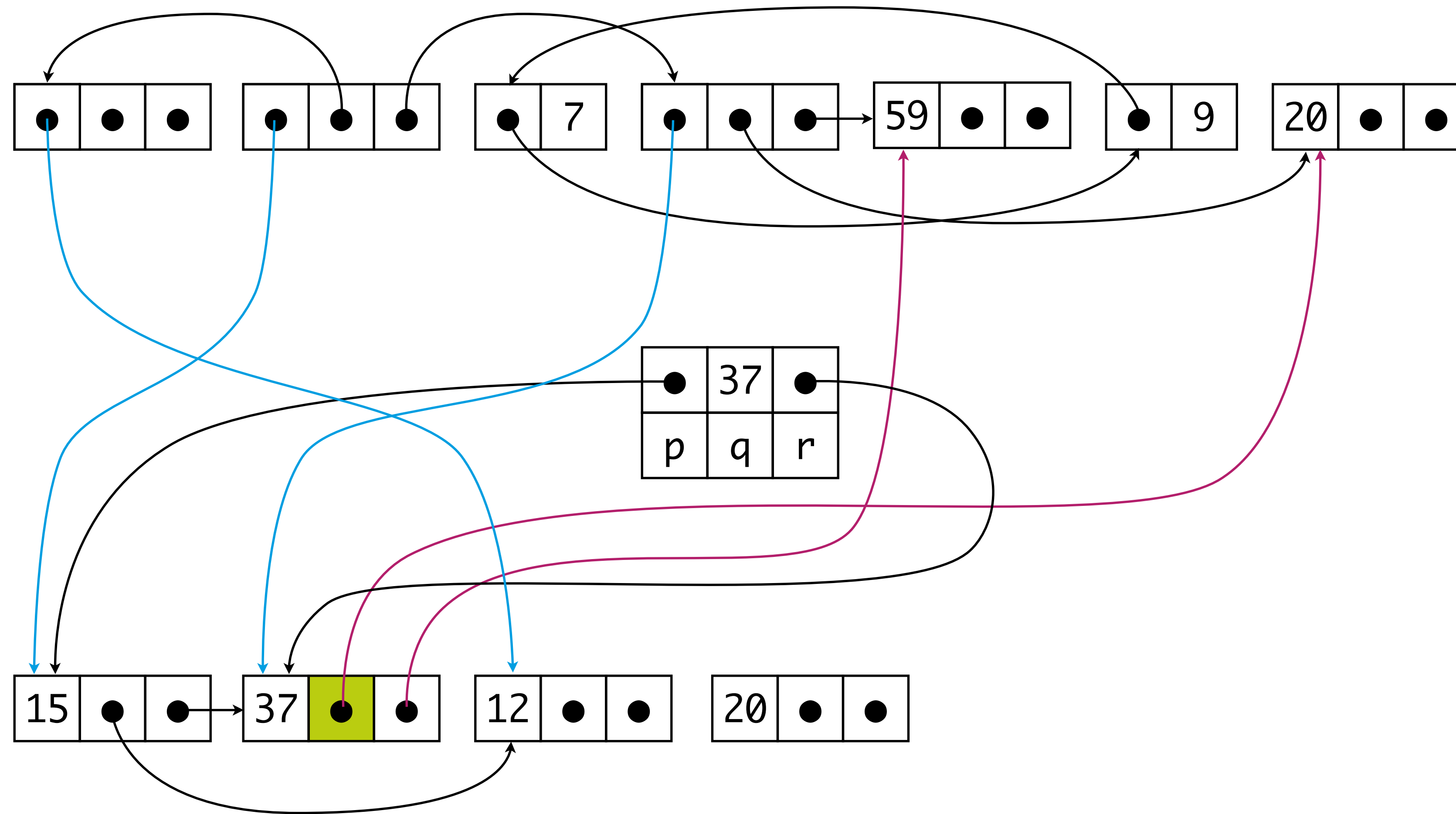
# Copying Collection: Example



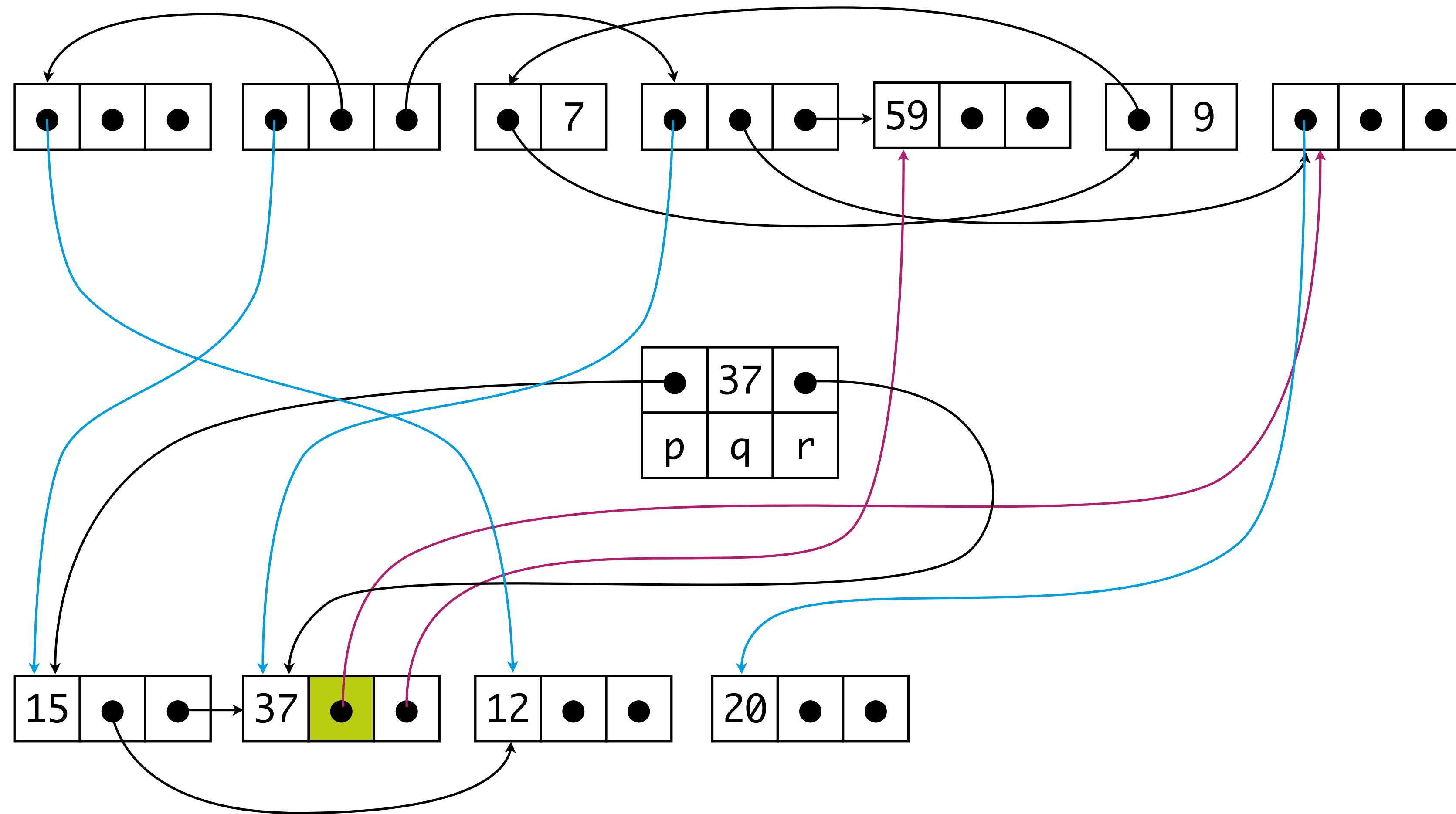
# Copying Collection: Example



# Copying Collection: Example

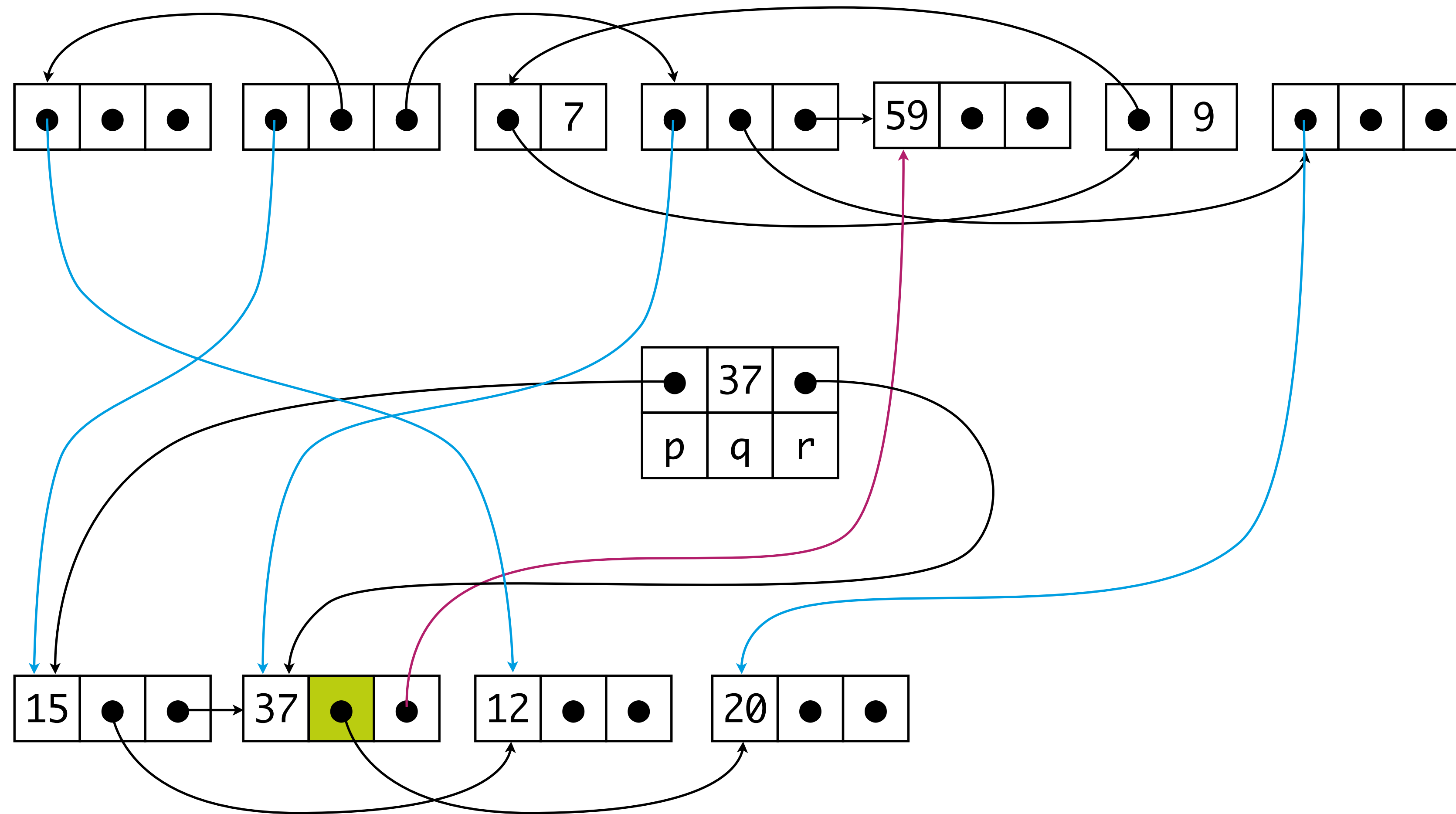


# Copying Collection: Example

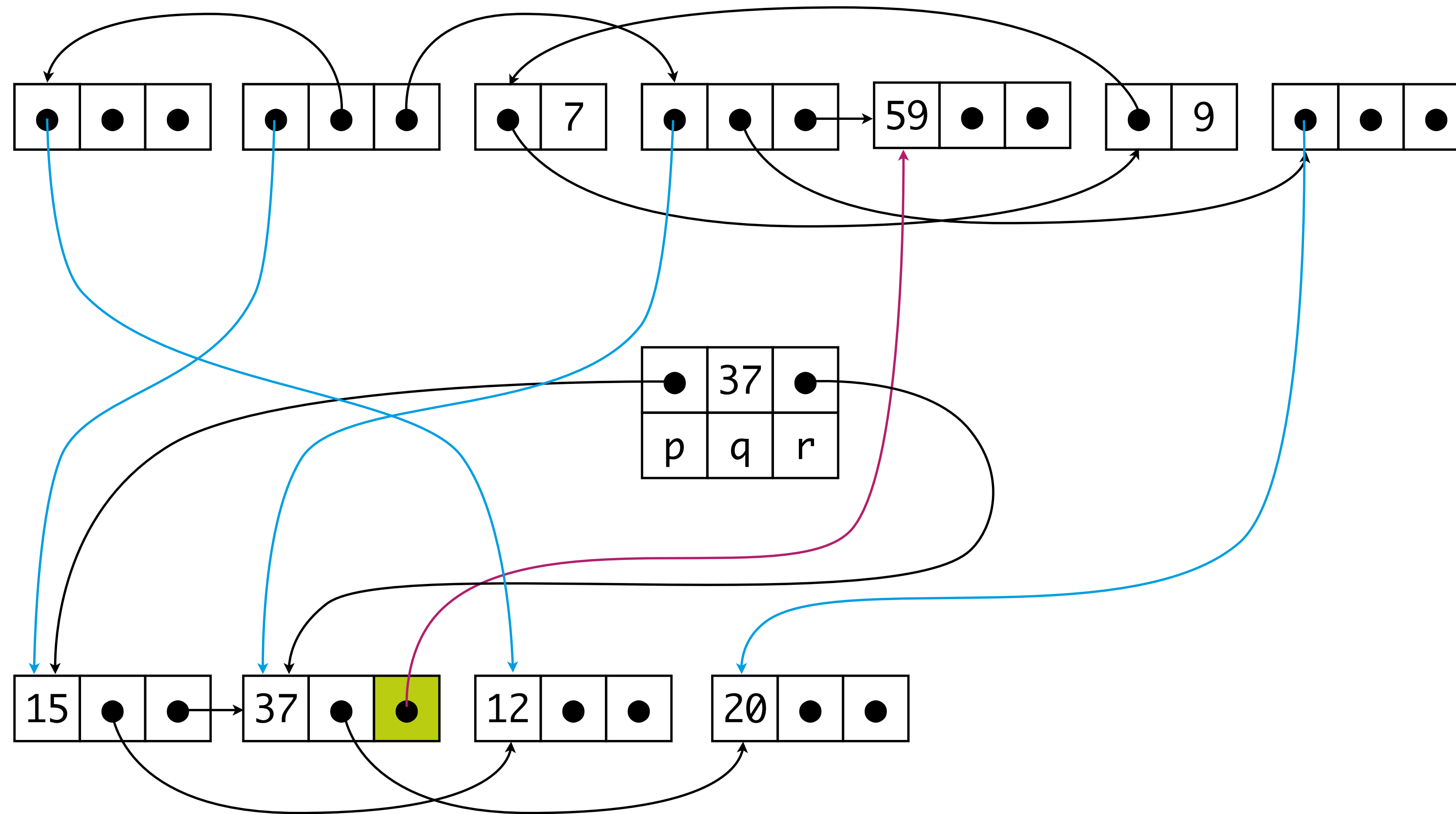




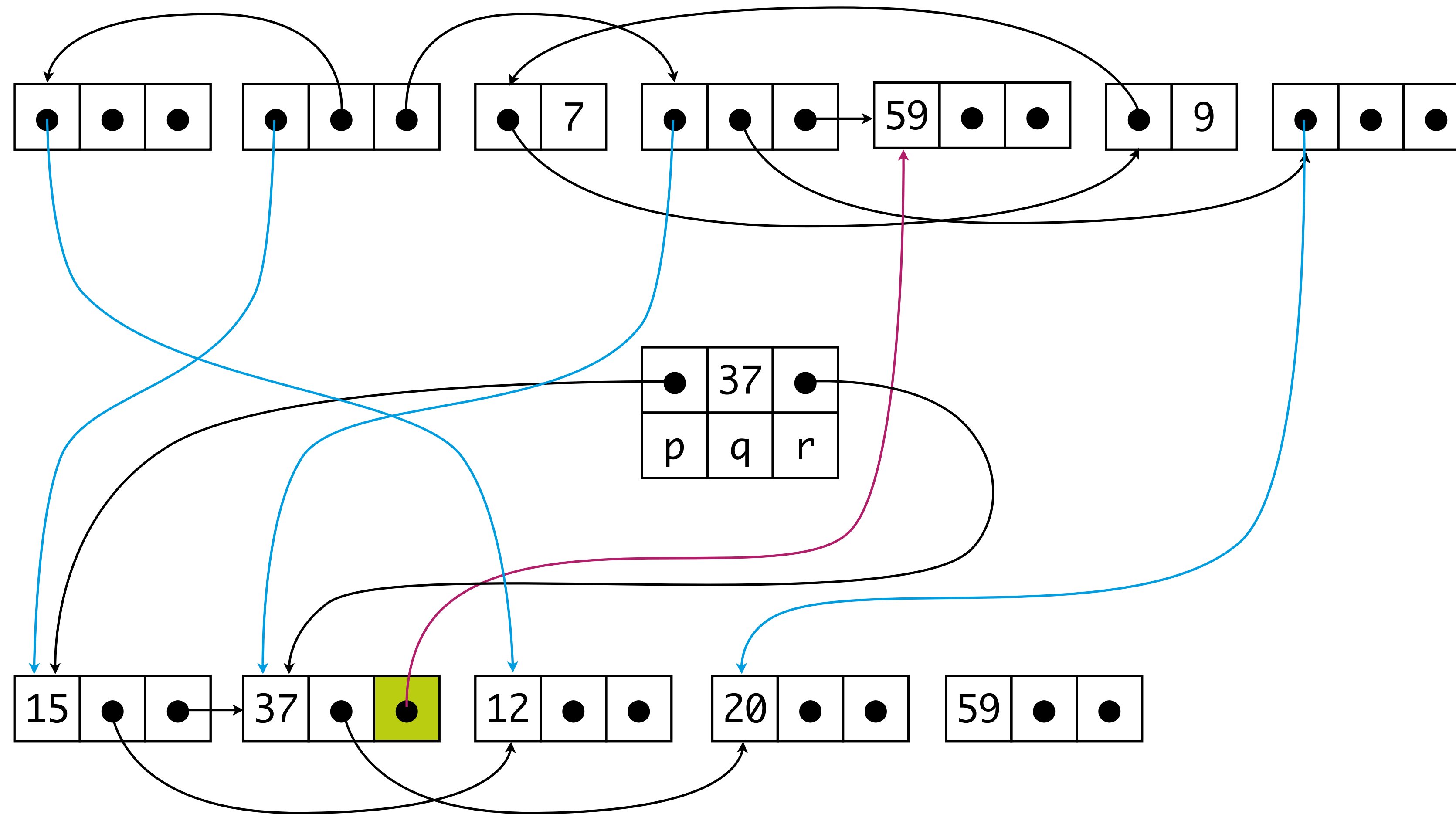
# Copying Collection: Example



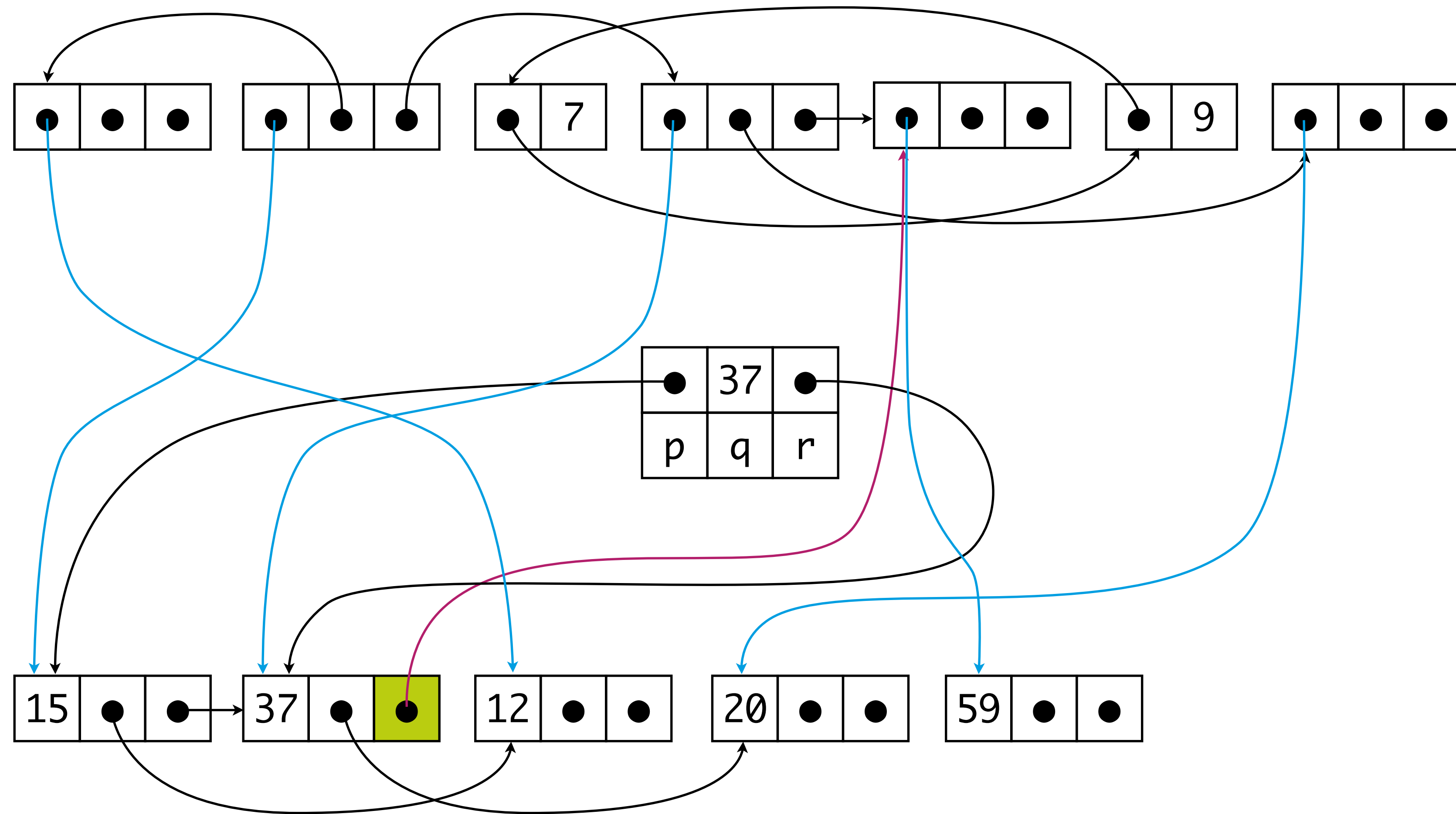
# Copying Collection: Example



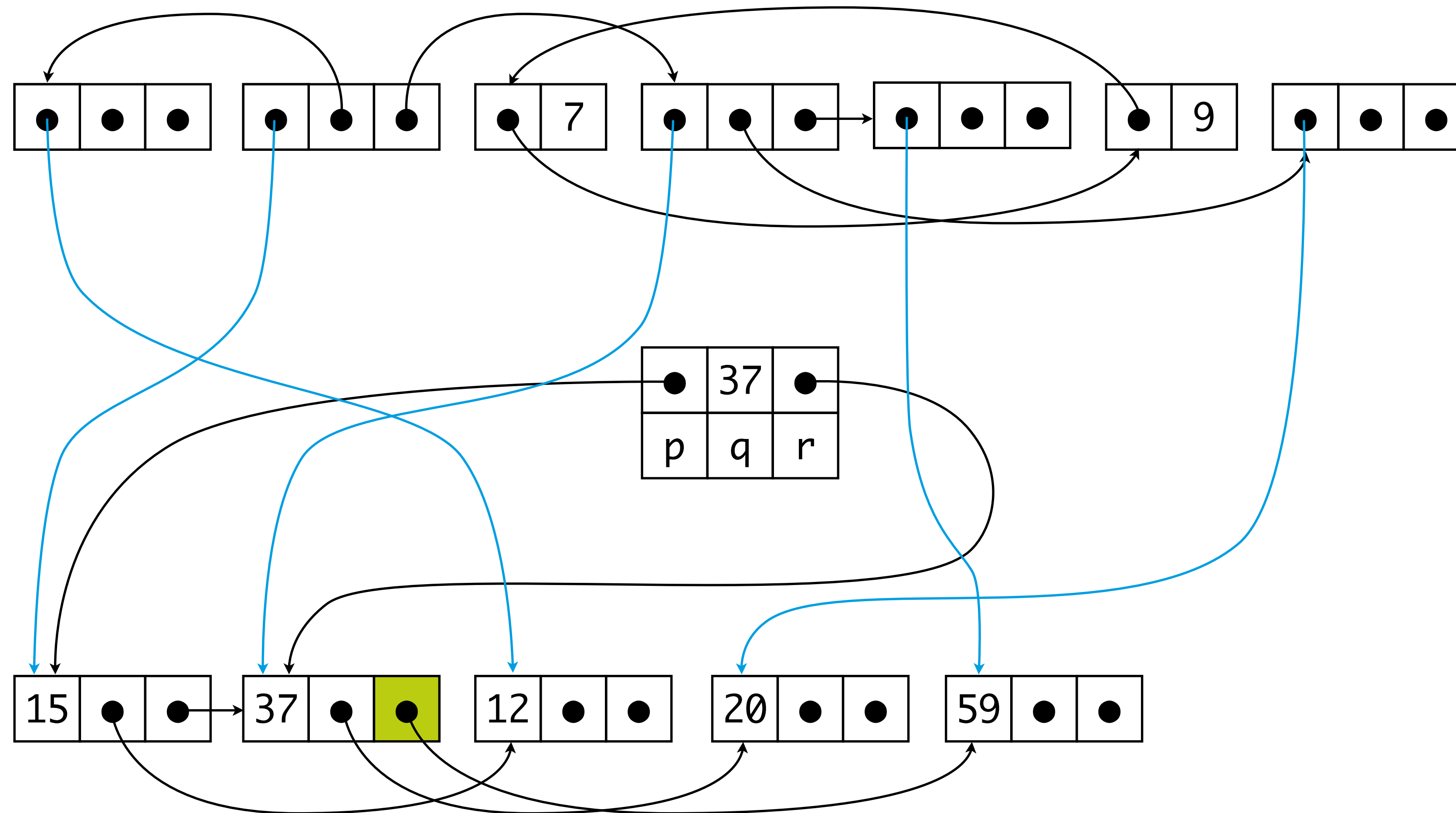
# Copying Collection: Example



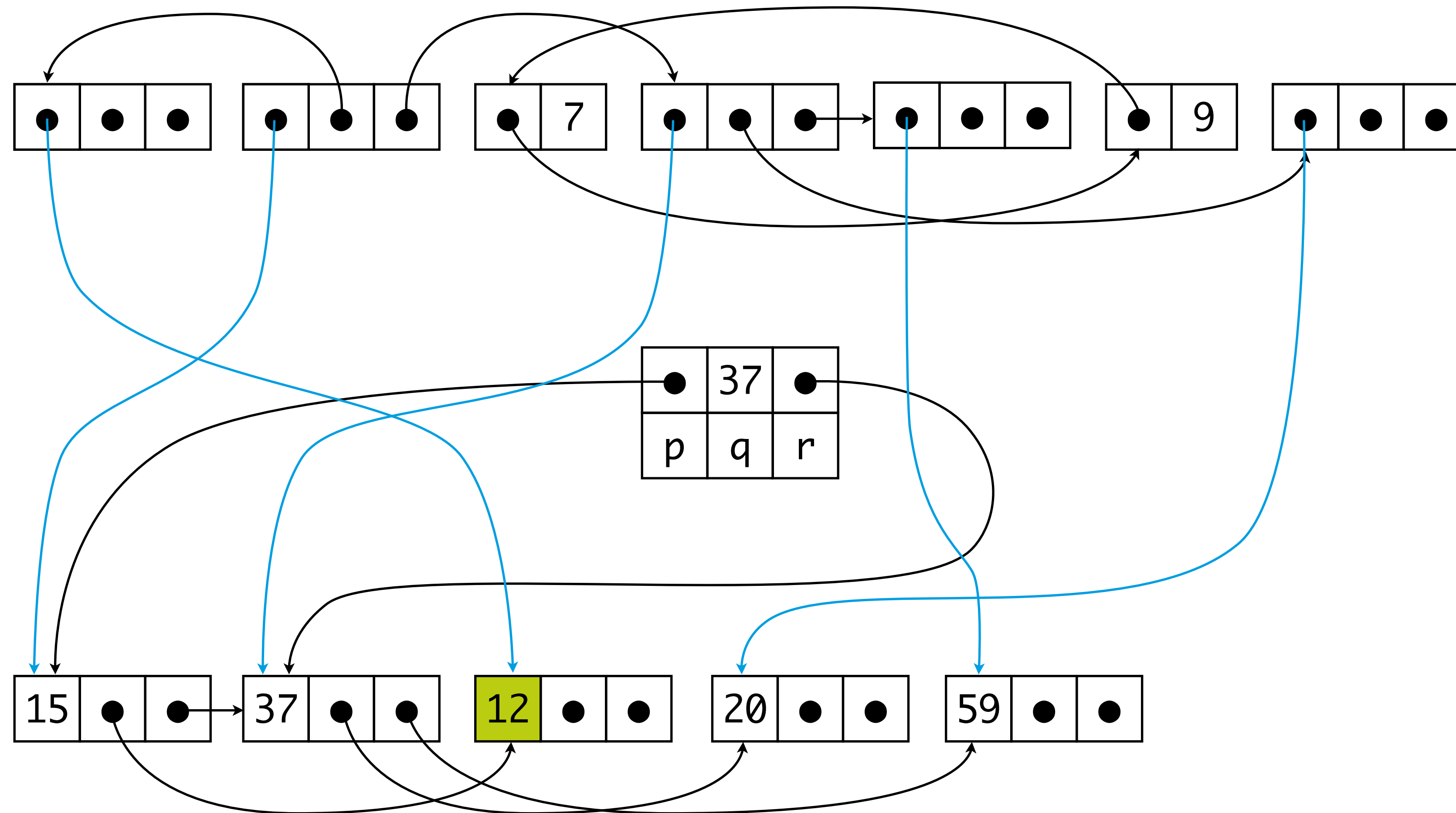
# Copying Collection: Example



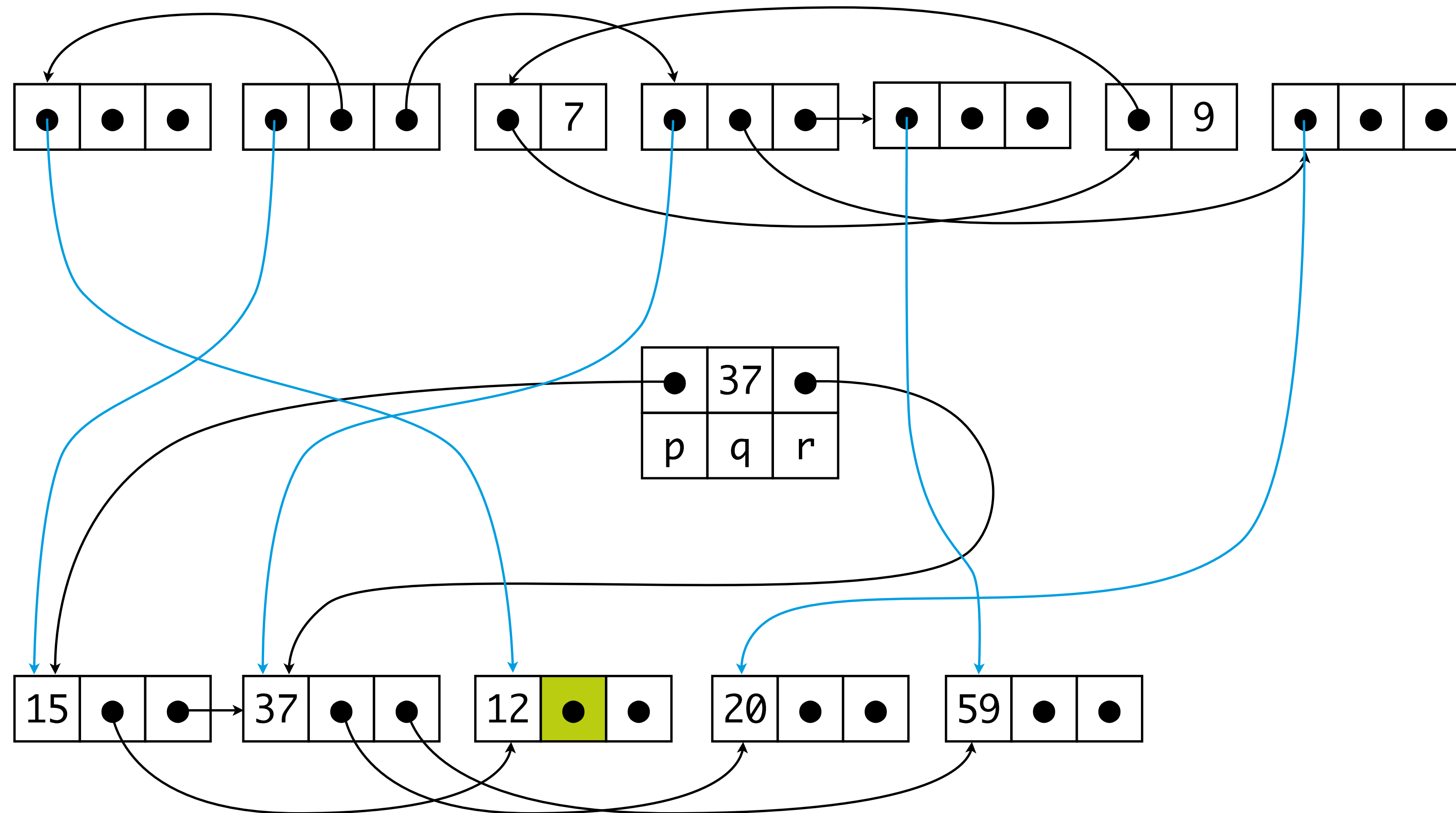
# Copying Collection: Example



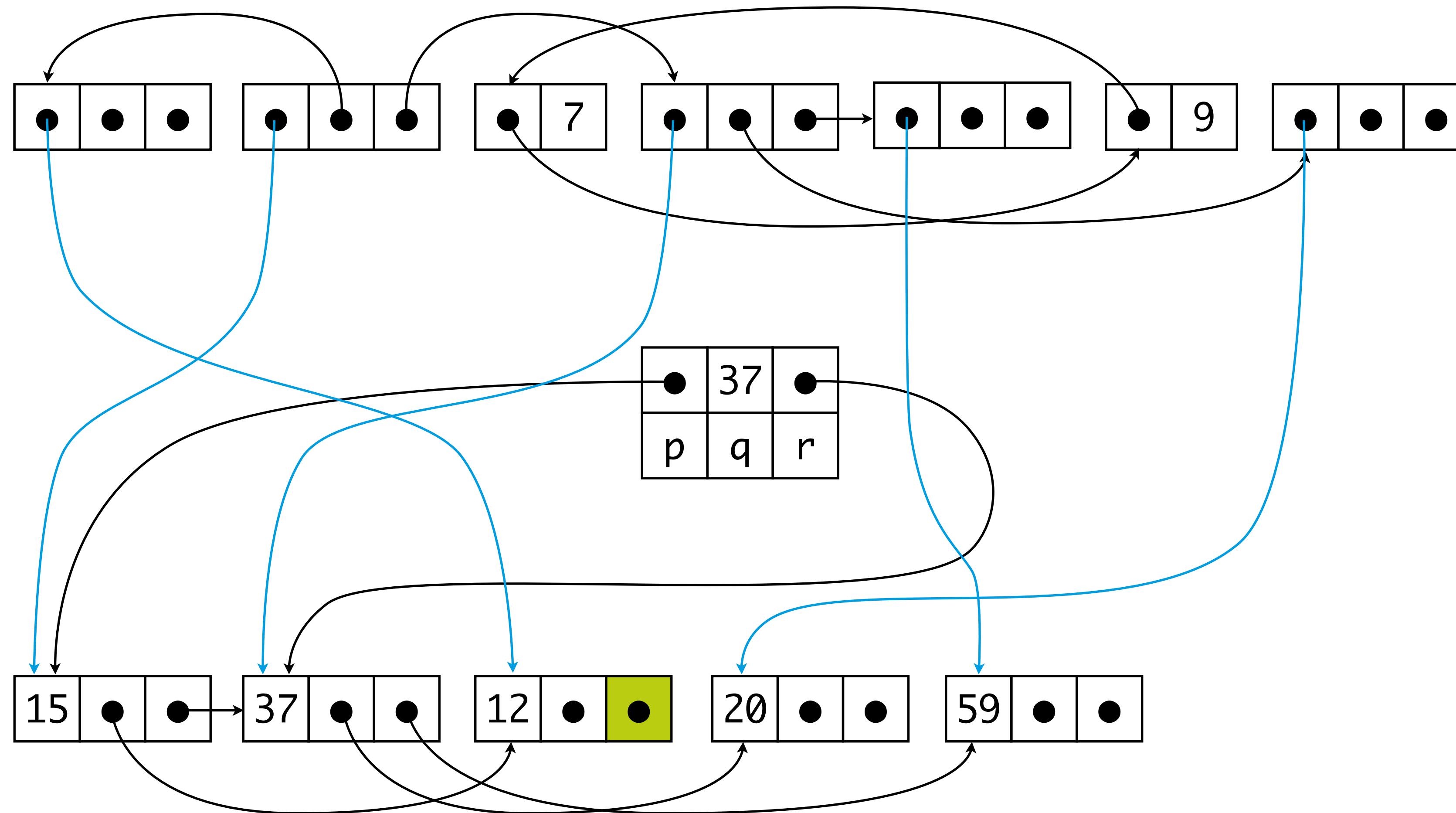
# Copying Collection: Example



# Copying Collection: Example

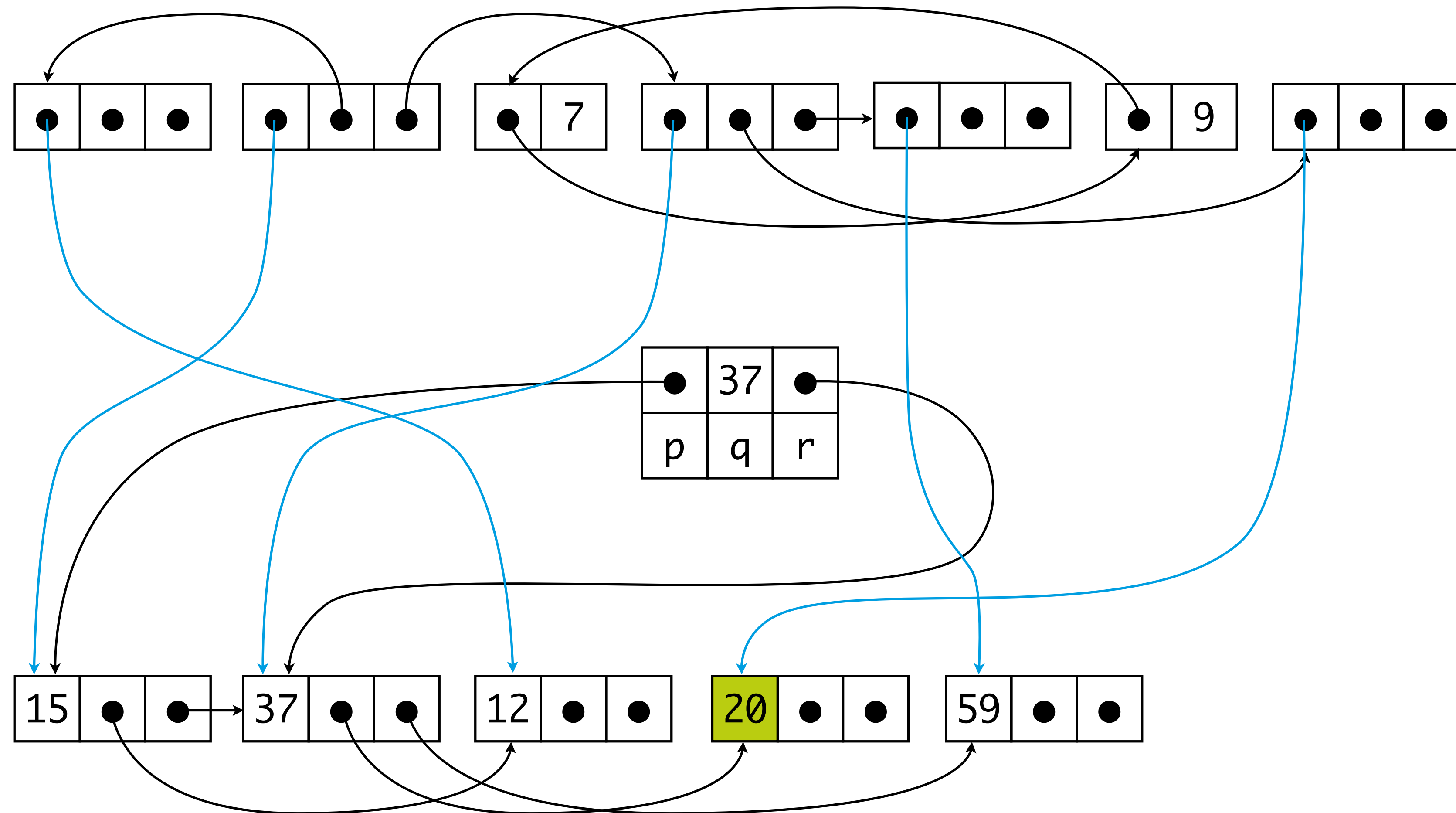


# Copying Collection: Example

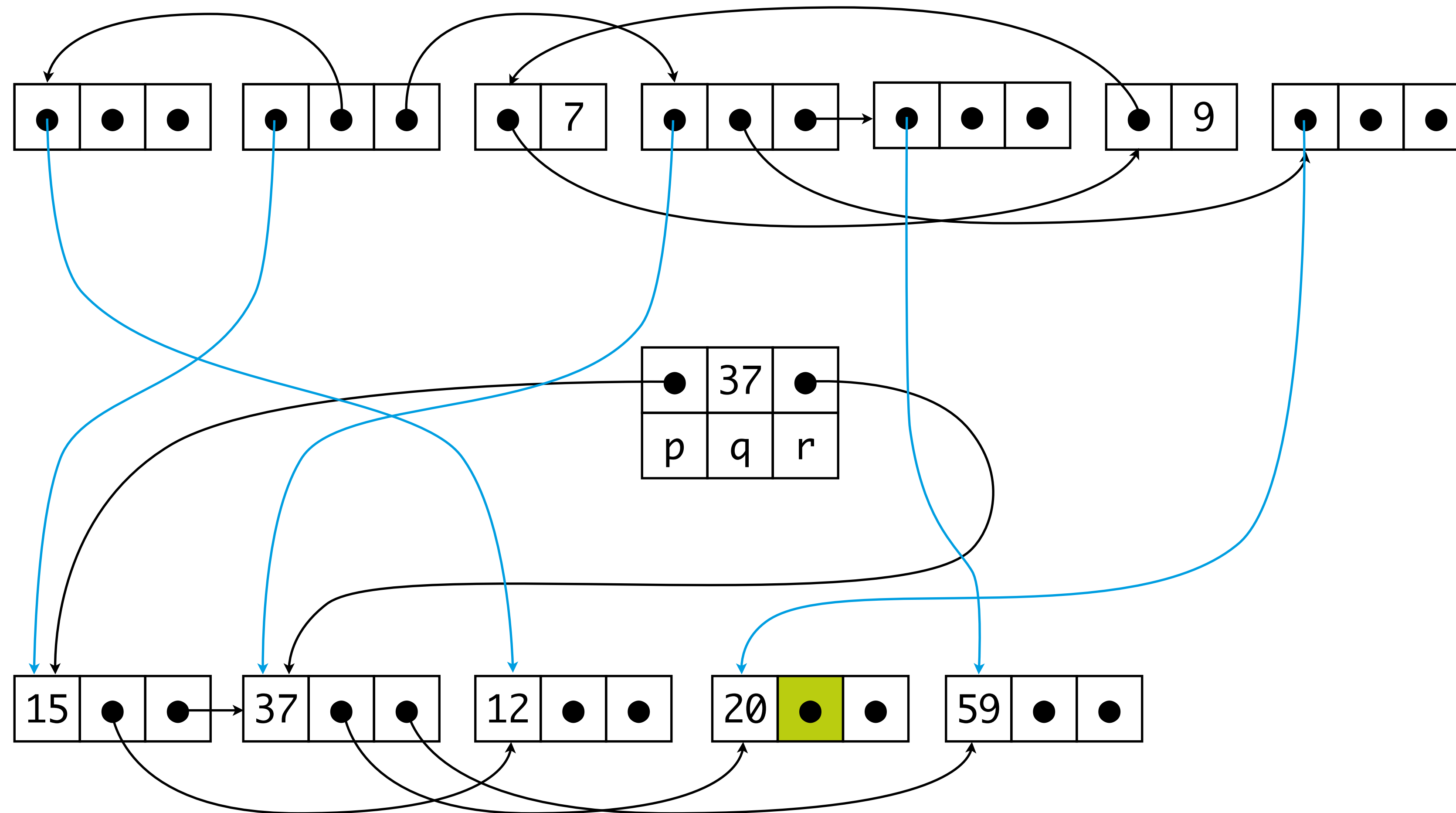




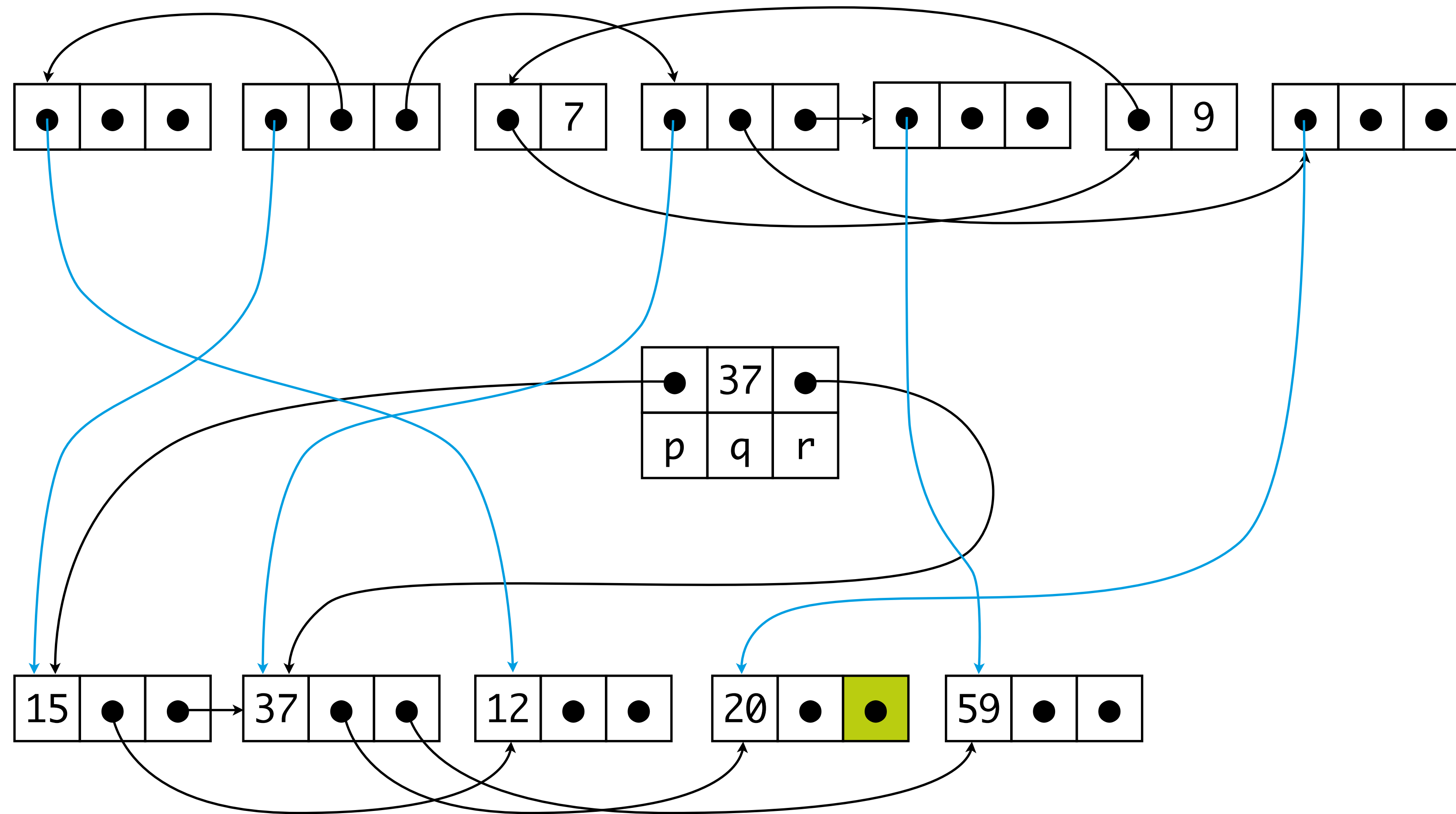
# Copying Collection: Example



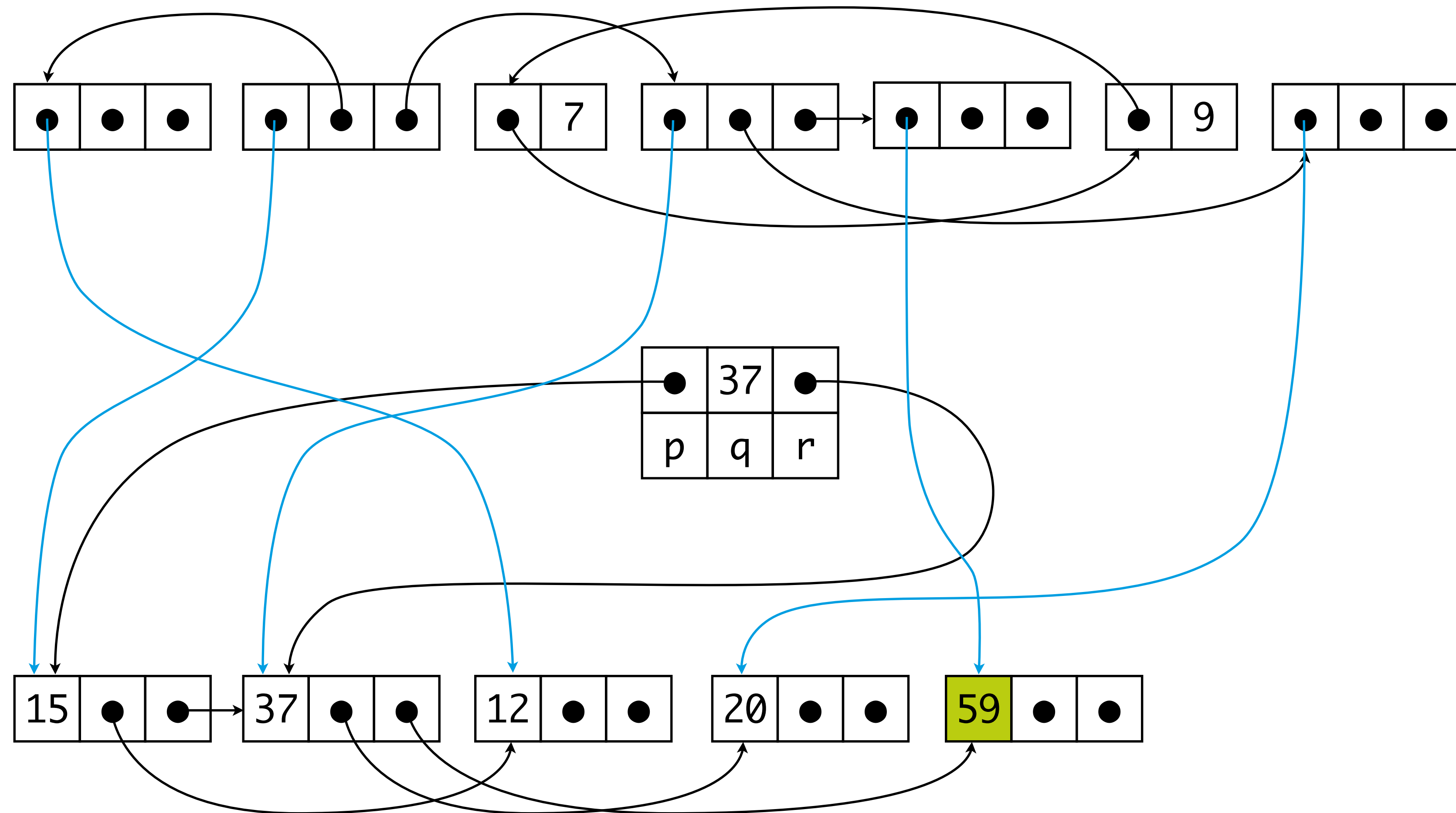
# Copying Collection: Example



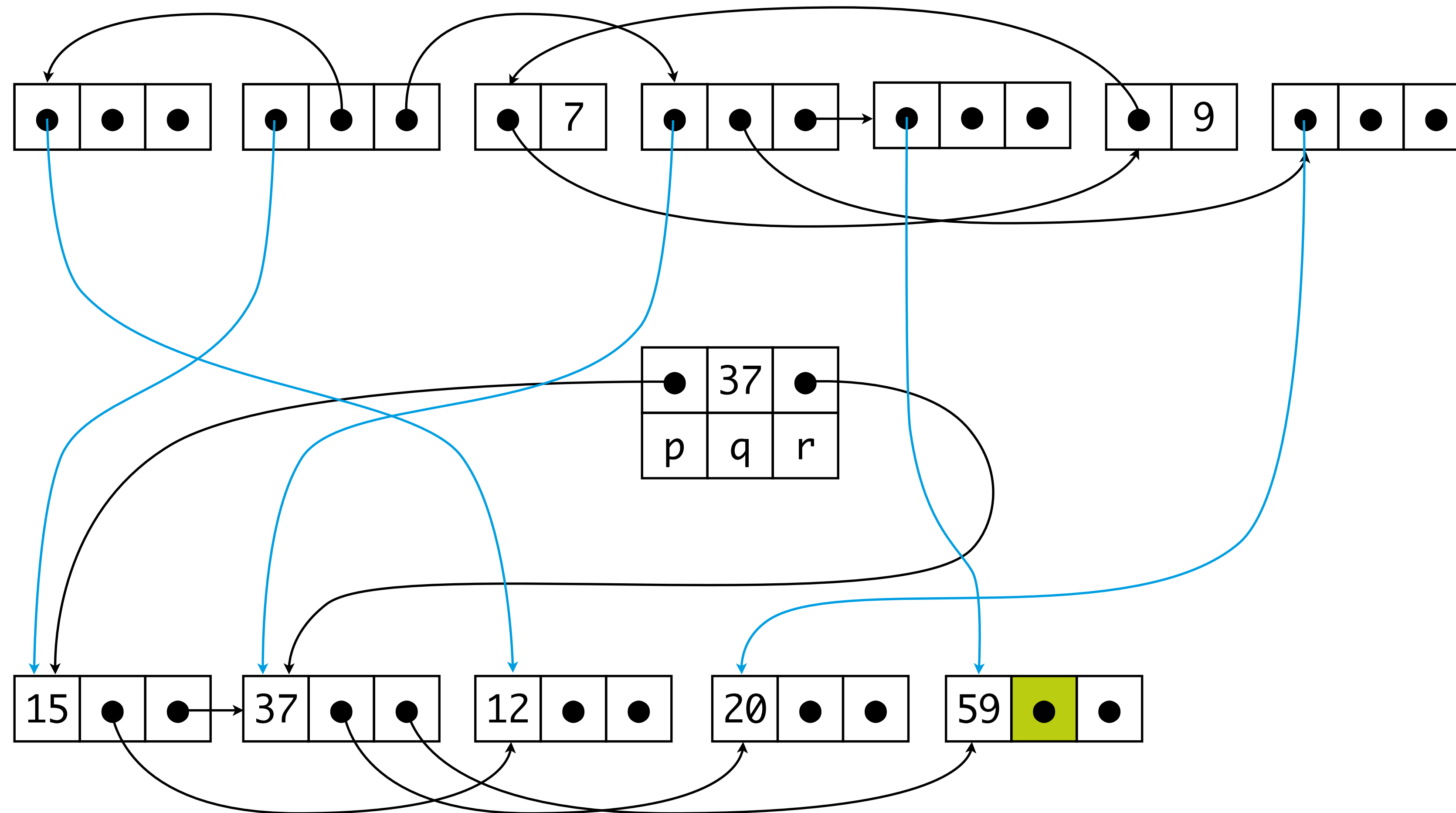
# Copying Collection: Example



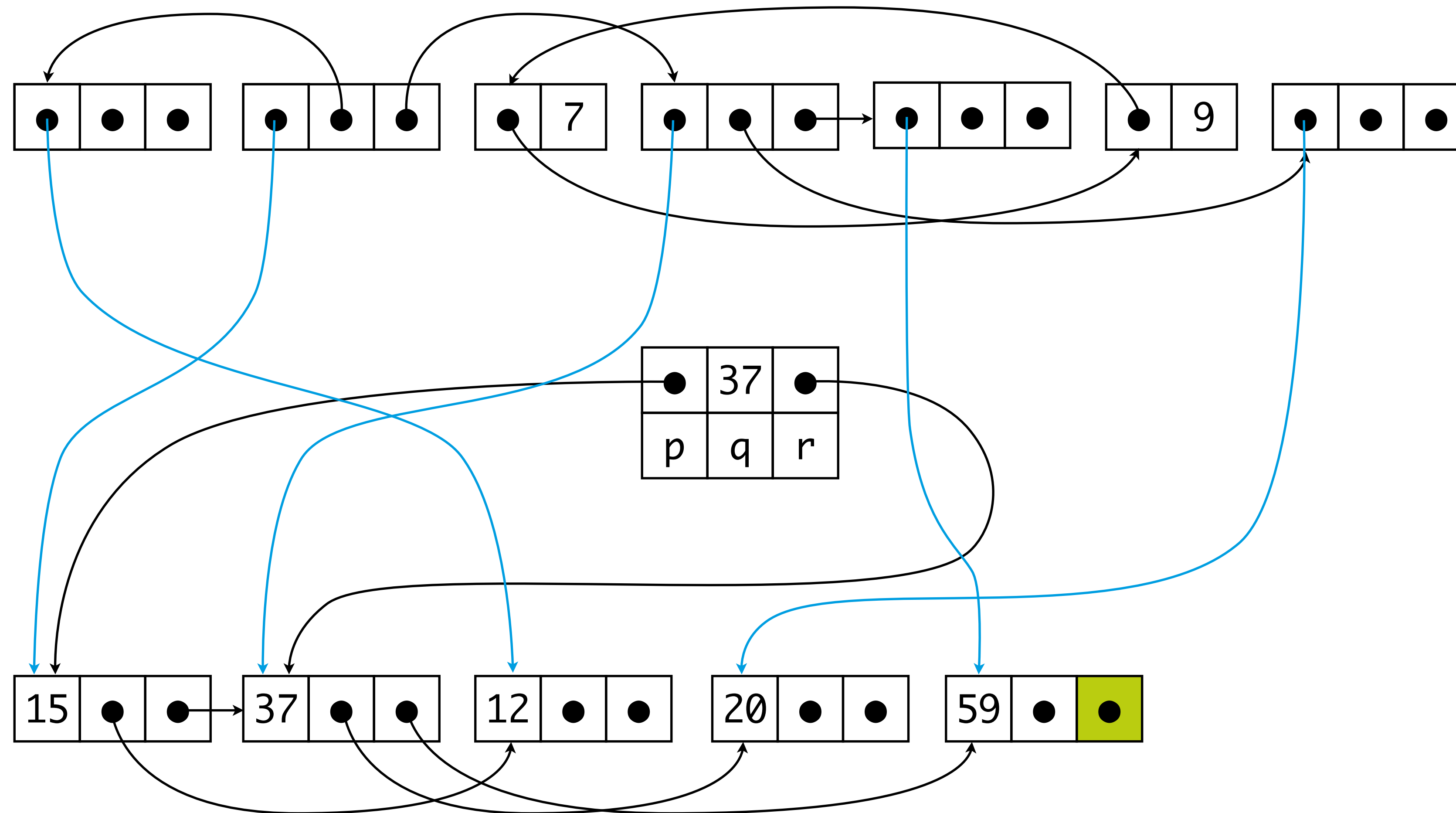
# Copying Collection: Example



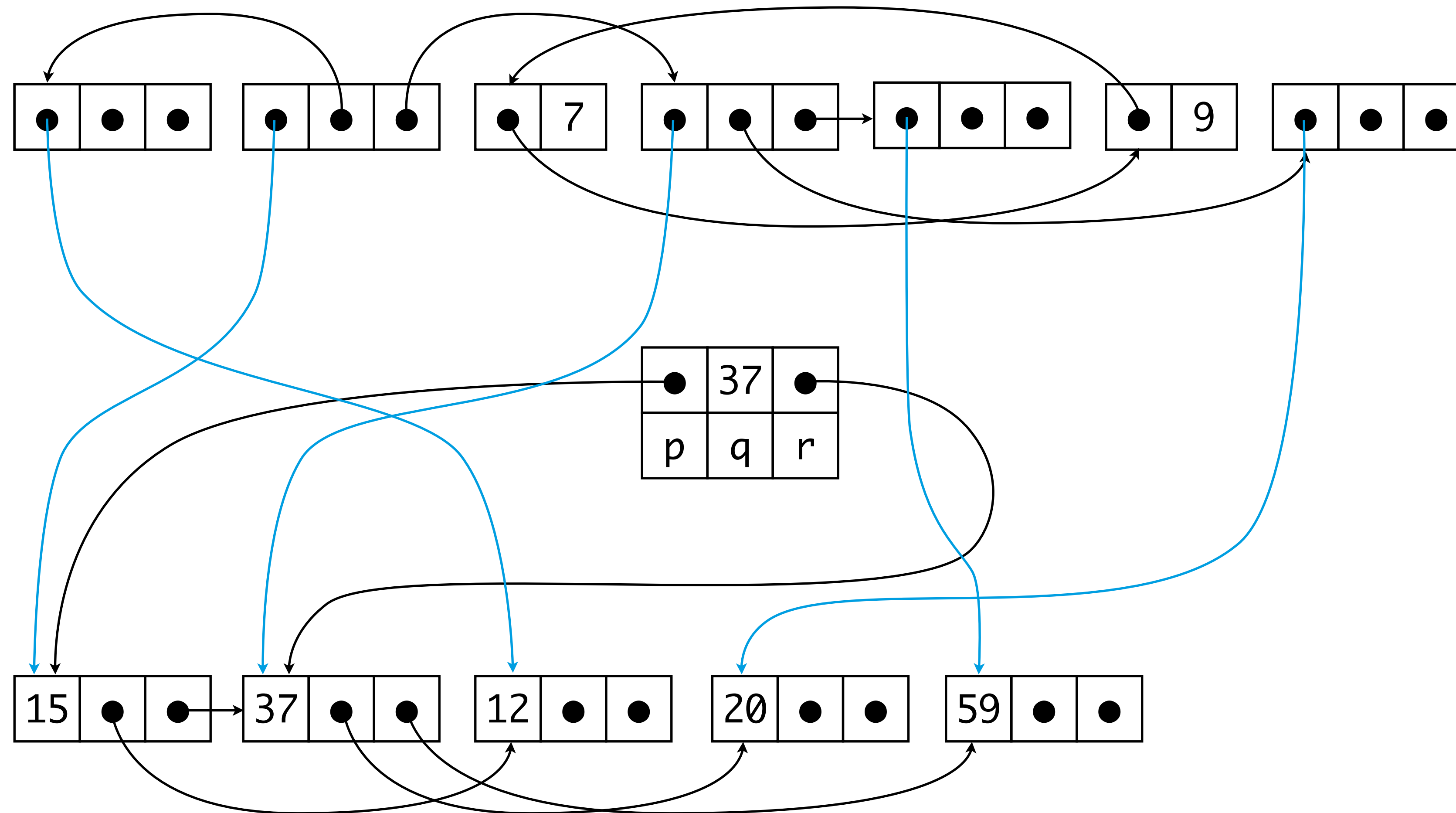
# Copying Collection: Example



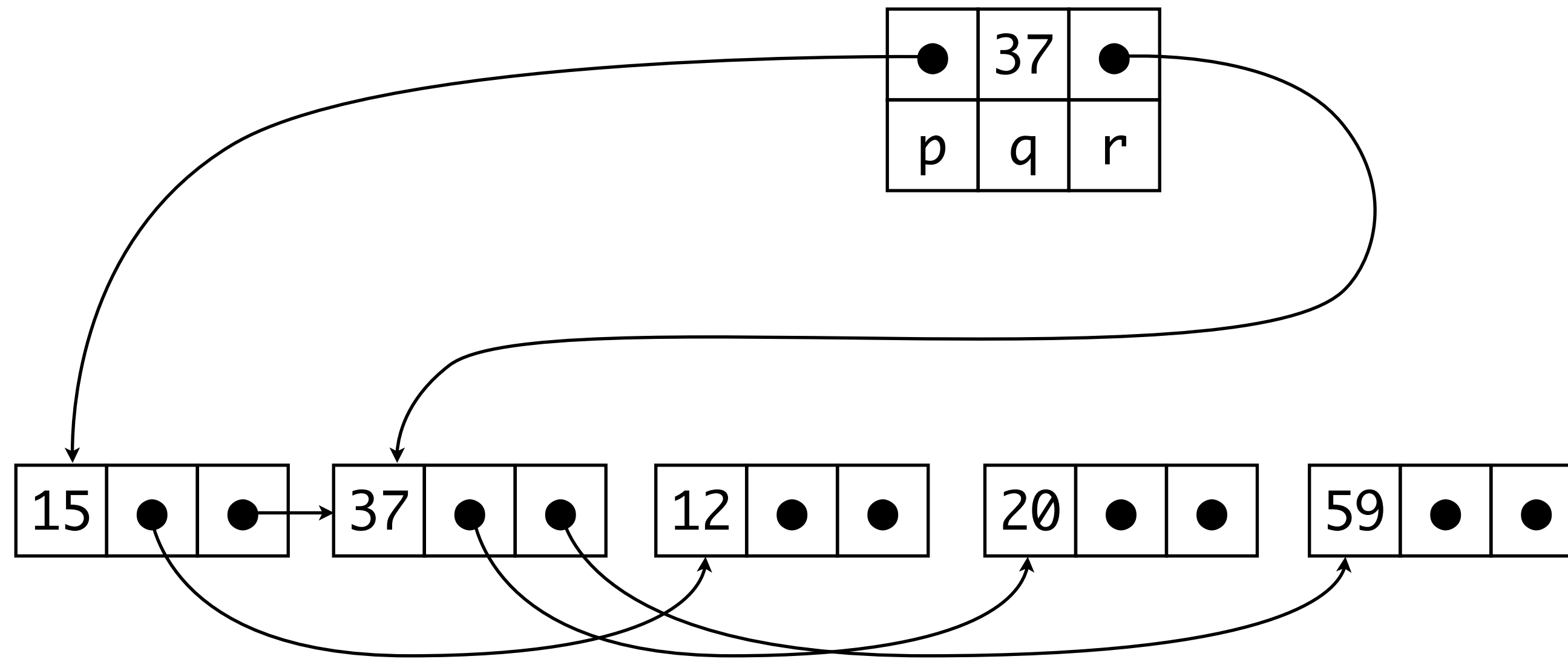
# Copying Collection: Example



# Copying Collection: Example



# Copying Collection: Example





## 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  $(H \gg R)$  : cost per allocated word  $\Rightarrow 0$
- If  $(H = 4R)$  :  $c3$  instructions per word allocated
- Solution: reduce portion of R to inspect  $\Rightarrow$  generational collection

# Generational Collection

# Generational Collection

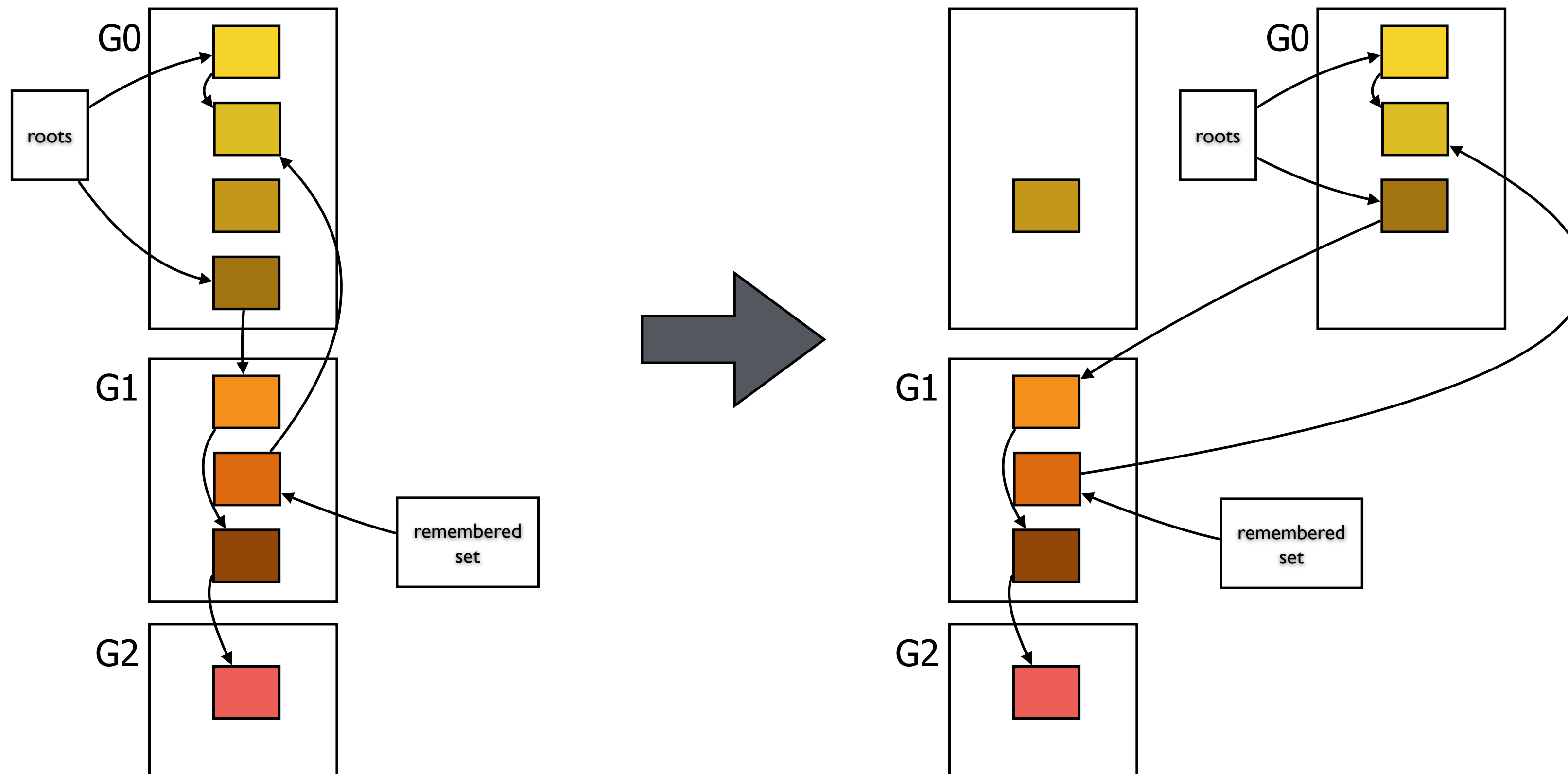
## Generations

- 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



# Generational Collection: Costs

## Instructions

- R reachable words in heap of size H
- BFS:  $c_3 * R$
- No sweep
- 10% of youngest generation is live:  $H/R = 10$
- Instructions per word reclaimed:  
$$(c_3 * R) / (H - R) = (c_3 * R) / (10R - R) \approx c_3/10$$
- Adding to remembered set: 10 instructions per update

# Incremental Collection

## 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

# 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

## 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

## 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

## 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

- Andrew W. Appel, Jens Palsberg. Modern Compiler Implementation in Java, 2nd edition, 2002.
- Sun Microsystems. Memory Management in the Java HotSpot™ Virtual Machine, April 2006.
- Richard Jones, Antony Hosking, Eliot Moss. The Garbage Collection Handbook. The Art of Automatic Memory Management.

# Language-Parametric Memory Management?

# Language-Parametric Memory Management?

## Garbage collectors are language-specific

- Representation of objects in memory
- Roots of heap in stack

## Can we derive garbage collector from language definition?

## A uniform model for memory layout

- Scopes describe static binding structure
- Frames instantiate scopes at run time
- Language-parametric memory management
- Language-parametric type safety

# Language-Parametric Type Safety?

## Type Safety: Well-typed programs don't go wrong

- A program that type checks does not have run-time type errors
- Preservation
  - ▶  $e : t \ \& \ e \rightarrow v \Rightarrow v : t$
- Progress
  - ▶  $e \rightarrow e' \Rightarrow e' \text{ is a value } \vee e' \rightarrow e''$
- (Slightly different for big step semantics as in definitional interpreters)

## Proving type safety

- Easier to establish with an interpreter
- Bindings complicate proof
- How to maintain?
- Can we automate verification of type safety?



Traditionally, operational semantics specifications use ad hoc mechanisms for representing the binding structures of programming languages.

This paper introduces frames as the dynamic counterpart of scopes in scope graphs.

This provides a uniform model for the representation of memory at run-time.

We are currently experimenting with specializing DynSem interpreters using scopes and frames using Truffle/Graal with encouraging results (200x speed-ups).

ECOOP 2016

<http://dx.doi.org/10.4230/LIPICs.ECOOP.2016.20>

# Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics (Artifact)\*

Casper Bach Poulsen<sup>1</sup>, Pierre Néron<sup>2</sup>, Andrew Tolmach<sup>3</sup>, and Eelco Visser<sup>4</sup>

- 1 Delft University of Technology  
c.b.poulsen@tudelft.nl
- 2 French Network and Information Security Agency (ANSSI)  
pierre.neron@ssi.gouv.fr
- 3 Portland State University  
tolmach@pdx.edu
- 4 Delft University of Technology  
visser@acm.org

## Abstract

Our paper introduces a systematic approach to the alignment of names in the static structure of a program, and memory layout and access during its execution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs,

and provides the basis for a language-independent specification of sound reachability-based garbage collection.

This Coq artifact showcases how our uniform model for memory layout in dynamic semantics provides structure to type soundness proofs. The artifact contains type soundness proofs mechanized in Coq for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent framework formalizing scope graphs and frame heaps.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs  
Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics  
Digital Object Identifier 10.4230/DARTS.2.1.10  
Related Article Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, and Eelco Visser, “Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics”, in Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016), LIPIcs, Vol. 56, pp. 20:1–20:26, 2016.  
<http://dx.doi.org/10.4230/LIPICs.ECOOP.2016.20>  
Related Conference 30th European Conference on Object-Oriented Programming (ECOOP 2016), July 18–22, 2016, Rome, Italy

## 1 Scope

The artifact is designed to document and support repeatability of the type soundness proofs in the companion paper [2], using the Coq proof assistant.<sup>1</sup> In particular, the artifact provides a

\* This work was partially funded by the NWO VICI *Language Designer’s Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.  
<sup>1</sup> <https://coq.inria.fr/>



# Specializing a Meta-Interpreter

JIT Compilation of DynSem Specifications on the Graal VM

Vlad Vergu  
TU Delft  
The Netherlands  
v.a.vergu@tudelft.nl

Eelco Visser  
TU Delft  
The Netherlands  
visser@acm.org

## ABSTRACT

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. DynSem specifications can be executed to interpret programs in the language under development. To enable fast turnaround during language development, we have developed a meta-interpreter for DynSem specifications, which requires minimal processing of the specification. In addition to fast development time, we also aim to achieve fast run times for interpreted programs.

In this paper we present the design of a meta-interpreter for DynSem and report on experiments with JIT compiling the application of the meta-interpreter on the Graal VM. By interpreting specifications directly, we have minimal compilation overhead. By specializing pattern matches, maintaining call-site dispatch chains and using native control-flow constructs we gain significant run-time performance. We evaluate the performance of the meta-interpreter when applied to the Tiger language specification running a set of common benchmark programs. Specialization enables the Graal VM to JIT compile the meta-interpreter giving speedups of up to factor 15 over running on the standard Oracle Java VM.

## CCS CONCEPTS

• **Software and its engineering** → **Interpreters; Domain specific languages; Semantics;**

## KEYWORDS

dynamic semantics, interpretation, JIT, run-time optimization

### ACM Reference Format:

Vlad Vergu and Eelco Visser. 2018. Specializing a Meta-Interpreter: JIT Compilation of DynSem Specifications on the Graal VM. In *15th International Conference on Managed Languages & Runtimes (ManLang’18), September 12–14, 2018, Linz, Austria*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3237009.3237018>

## 1 INTRODUCTION

The dynamic semantics of a programming language defines the run time execution behavior of programs in the language. Ideally,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*ManLang’18, September 12–14, 2018, Linz, Austria*

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6424-9/18/09...\$15.00

<https://doi.org/10.1145/3237009.3237018>

the design of a programming language starts with the specification of its dynamic semantics to provide a high-level readable and unambiguous definition. However, understanding the design of a programming language also requires experimentation by actually running programs. Therefore, this ideal route is rarely taken, but language designs are embodied in the implementation of interpreters or compilers instead.

We have previously designed DynSem [33], a high-level meta-DSL for dynamic semantics specifications of programming languages, with the aim of supporting readable *and* executable specification. It supports the definition of modular and concise semantics by means of reduction rules with implicit propagation of contextual information. DynSem’s executable semantics entails that specifications can be used to interpret object language programs.

In our early prototypes, DynSem specifications were compiled to an interpreter. The process of generating a Java implementation of an interpreter and compiling that generated code caused long turnaround times during language prototyping. In order to support rapid prototyping with short turnaround times, we turned to interpreting specifications directly instead of compiling them. A DynSem interpreter is a *meta-interpreter* since the programs it interprets are themselves interpreters. Figure 1 depicts the high-level architecture of the DynSem meta-interpreter. First, a DynSem specification is desugared (explicated) to make implicit passing of semantic components explicit. The resulting specification in DynSem Core is then loaded into the meta-interpreter together with the AST of the interpreted object program. The interpreter consumes the program as input enacting the specification. This produces the desired result of a short turnaround time for experimenting with dynamic semantics specifications.

Meta-interpretation reduces the turnaround time at the expense of execution performance. At run time there are two interpreter layers operating (the meta-language interpreter and the object-language interpreter) which introduces substantial overhead. While we envision DynSem as a convenient way to prototype the dynamic semantics of programming languages, ultimately we also envision it as a convenient way to bridge the gap between the prototyping and production phases of a programming language’s lifecycle. Thus, we not only want an interpreter fast, but we also want a fast interpreter, which raises the question: Can we achieve fast object-language interpreters by optimizing the meta-interpretation of dynamic semantics specifications?

Direct vanilla interpreters are in general slow to begin with, even when they are implemented in a host language that is JIT-ed. This is because the host JIT is unable to see patterns in the object language and to meaningfully optimize the interpreter. The task of optimizing an interpreter has traditionally been long and

# Scopes and Frames Improve Meta-Interpreter Specialization

Vlad Vergu

Delft University of Technology, Delft, The Netherlands

[v.a.vergu@tudelft.nl](mailto:v.a.vergu@tudelft.nl)

Andrew Tolmach 

Portland State University, Portland, OR, USA

[tolmach@pdx.edu](mailto:tolmach@pdx.edu)

Eelco Visser 

Delft University of Technology, Delft, The Netherlands

[e.visser@tudelft.nl](mailto:e.visser@tudelft.nl)

## Abstract

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. To maintain a short definition-to-execution cycle, DynSem specifications are meta-interpreted. Meta-interpretation introduces runtime overhead that is difficult to remove by using interpreter optimization frameworks such as the Truffle/Graal Java tools; previous work has shown order-of-magnitude improvements from applying Truffle/Graal to a meta-interpreter, but this is still far slower than what can be achieved with a language-specific interpreter. In this paper, we show how specifying the meta-interpreter using *scope graphs*, which encapsulate static name binding and resolution information, produces much better optimization results from Truffle/Graal. Furthermore, we identify that JIT compilation is hindered by large numbers of calls between small polymorphic rules and we introduce *rule cloning* to derive larger monomorphic rules at run time as a countermeasure. Our contributions improve the performance of DynSem-derived interpreters to within an order of magnitude of a handwritten language-specific interpreter.

**2012 ACM Subject Classification** Software and its engineering → Interpreters

**Keywords and phrases** Definitional interpreters, partial evaluation

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2019.4

**Funding** This research was partially funded by the NWO VICI *Language Designer’s Workbench* project (639.023.206) and by a gift from the Oracle Corporation.

**Acknowledgements** We thank the anonymous reviewers for their feedback on previous versions of this paper, and we thank Laurence Tratt for his guidance on obtaining reliable runtime measurements and analyzing the resulting time series.

## 1 Introduction

A *language workbench* [9, 36] is a computing environment that aims to support the rapid development of programming languages with a quick turnaround time for language design experiments. Meeting that goal requires that (a) turning a language design idea into an executable prototype is easy; (b) the delay between making a change to the language and starting to execute programs in the revised prototype is short; and (c) the prototype runs programs reasonably quickly. Moreover, once the language design has stabilized, we will need a way to run programs at production speed, as defined for the particular language and application domain.




© Vlad Vergu, Andrew Tolmach, and Eelco Visser;

licensed under Creative Commons License CC-BY

33rd European Conference on Object-Oriented Programming (ECOOP 2019).

Editor: Alastair F. Donaldson; Article No. 4; pp. 4:1–4:30

Leibniz International Proceedings in Informatics

 LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



A desirable property for programming languages is type safety: well-typed programs don't go wrong.

Demonstrating type safety for language implementations requires a proof. Such a proof is hard (at least tedious) for language models, and rarely done for language implementations.

Can we automatically check type safety for language implementations?

This paper shows how to do that at least for definitional interpreters for non-trivial languages. (By using scopes and frames to represent bindings.)

POPL 2018

<https://doi.org/10.1145/3158104>

**Intrinsically-Typed Definitional Interpreters for Imperative Languages**

CASPER BACH POULSEN, Delft University of Technology, The Netherlands  
ARJEN ROUVOET, Delft University of Technology, The Netherlands  
ANDREW TOLMACH, Portland State University, USA  
ROBBERT KREBBERS, Delft University of Technology, The Netherlands  
EELCO VISSER, Delft University of Technology, The Netherlands

A definitional interpreter defines the semantics of an object language in terms of the (well-known) semantics of a host language, enabling understanding and validation of the semantics through execution. Combining a definitional interpreter with a separate type system requires a separate type safety proof. An alternative approach, at least for pure object languages, is to use a dependently-typed language to encode the object language type system in the definition of the abstract syntax. Using such intrinsically-typed abstract syntax definitions allows the host language type checker to verify automatically that the interpreter satisfies type safety. Does this approach scale to larger and more realistic object languages, and in particular to languages with mutable state and objects?

In this paper, we describe and demonstrate techniques and libraries in Agda that successfully scale up intrinsically-typed definitional interpreters to handle rich object languages with non-trivial binding structures and mutable state. While the resulting interpreters are certainly more complex than the simply-typed  $\lambda$ -calculus interpreter we start with, we claim that they still meet the goals of being concise, comprehensible, and executable, while guaranteeing type safety for more elaborate object languages. We make the following contributions: (1) A *dependent-passing style* technique for hiding the weakening of indexed values as they propagate through monadic code. (2) An Agda library for programming with *scope graphs* and *frames*, which provides a uniform approach to dealing with name binding in intrinsically-typed interpreters. (3) Case studies of intrinsically-typed definitional interpreters for the simply-typed  $\lambda$ -calculus with references (STLC+Ref) and for a large subset of Middleweight Java (MJ).

CCS Concepts: • **Theory of computation** → **Program verification**; *Type theory*; • **Software and its engineering** → **Formal language definitions**;

Additional Key Words and Phrases: definitional interpreters, dependent types, scope graphs, mechanized semantics, Agda, type safety, Java

**ACM Reference Format:**  
Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (January 2018), 34 pages. <https://doi.org/10.1145/3158104>

Authors' addresses: Casper Bach Poulsen, Delft University of Technology, The Netherlands, [c.b.poulsen@tudelft.nl](mailto:c.b.poulsen@tudelft.nl); Arjen Rouvoet, Delft University of Technology, The Netherlands, [a.j.rouvoet@tudelft.nl](mailto:a.j.rouvoet@tudelft.nl); Andrew Tolmach, Portland State University, Oregon, USA, [tolmach@pdx.edu](mailto:tolmach@pdx.edu); Robbert Krebbers, Delft University of Technology, The Netherlands, [r.j.krebbers@tudelft.nl](mailto:r.j.krebbers@tudelft.nl); Eelco Visser, Delft University of Technology, The Netherlands, [e.visser@tudelft.nl](mailto:e.visser@tudelft.nl).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2018 Copyright held by the owner/author(s).  
2475-1421/2018/1-ART16  
<https://doi.org/10.1145/3158104>

Except where otherwise noted, this work is licensed under

