Organizing Data Efficiently with Common Data Structures

Rasmus Resen Amossen

http://rasmus.resen.org

Data Structures

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

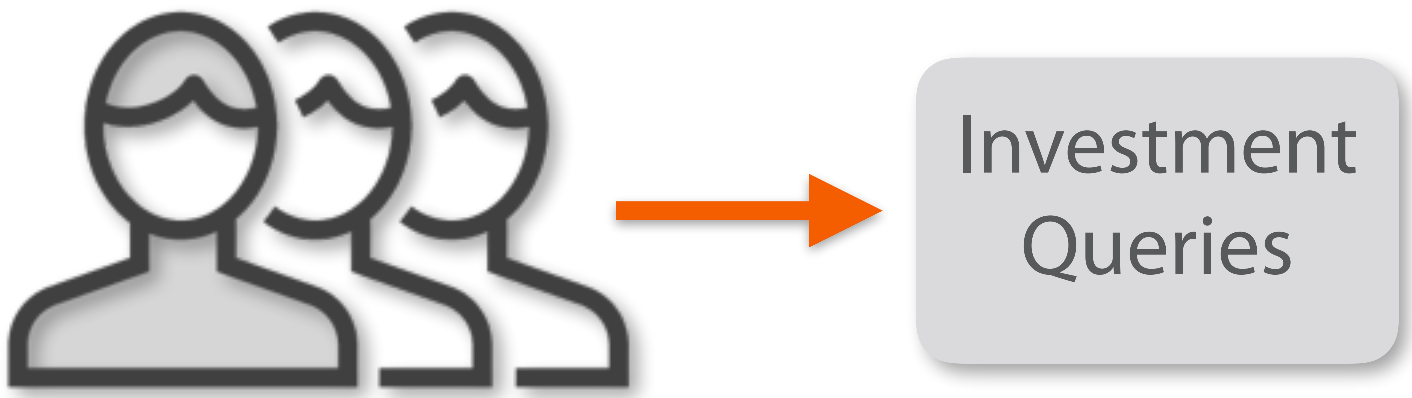
Data Structures

Dynamic array

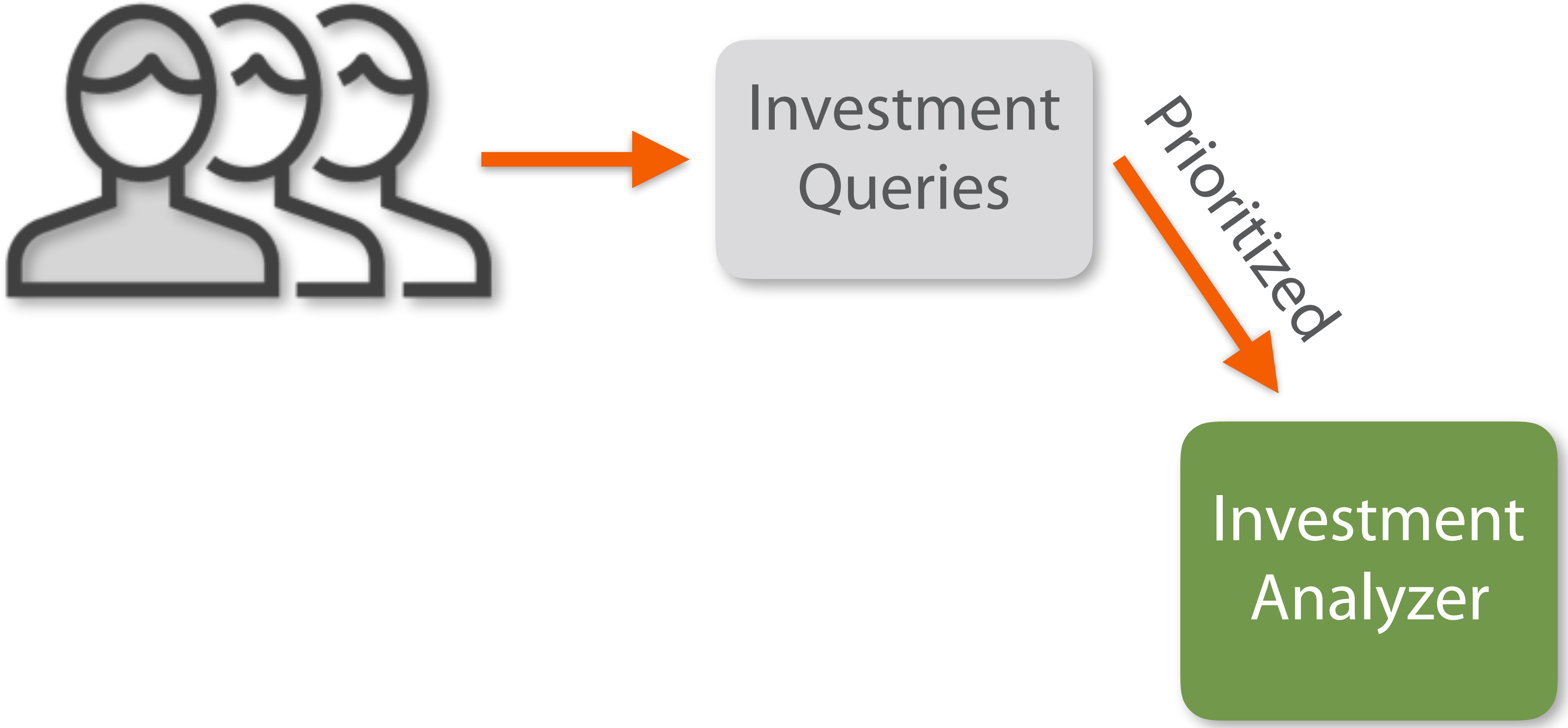
Hash table

Linked list

Priority queue



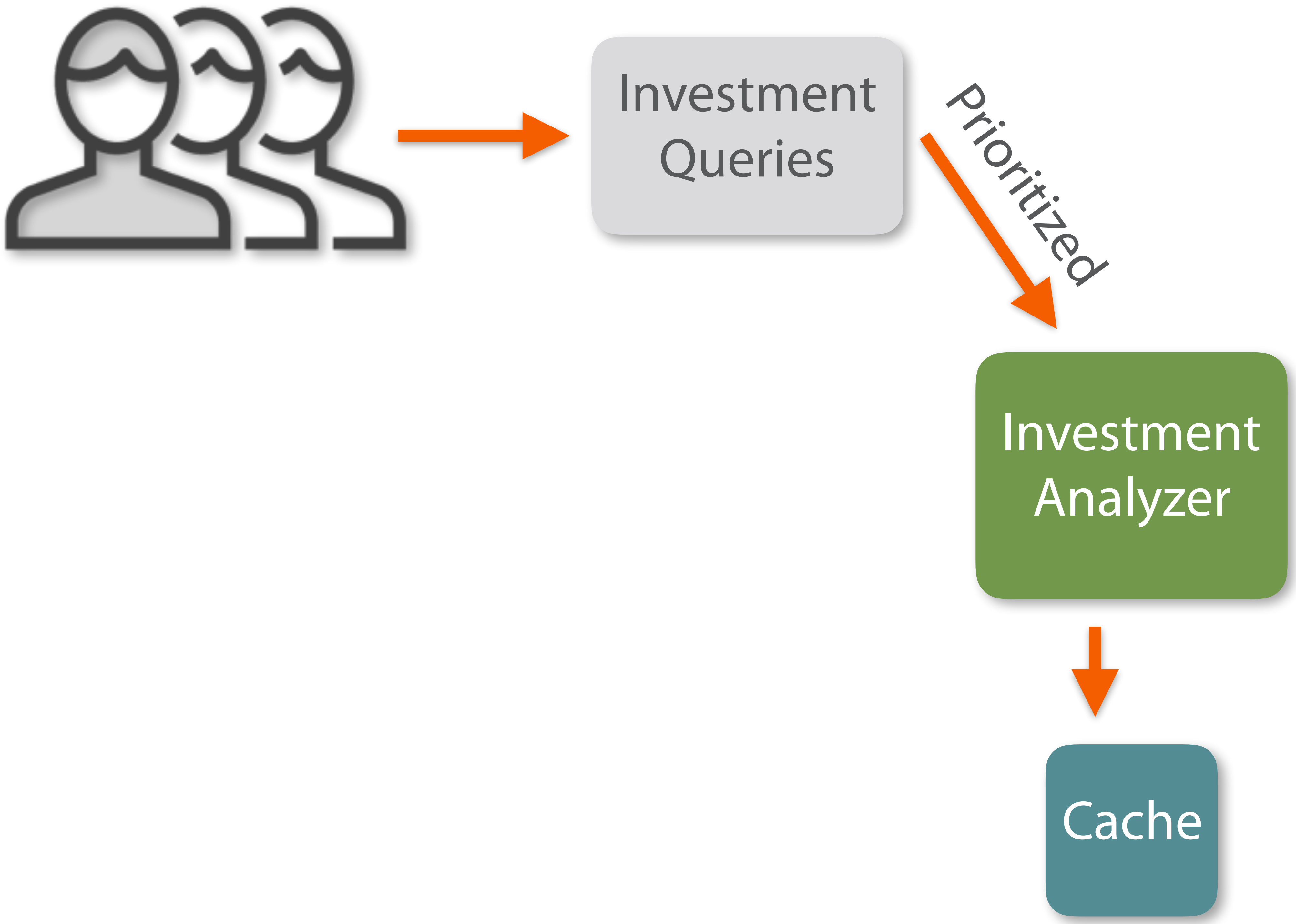
Investment Queries



Investment Queries

Prioritized

Investment Analyzer

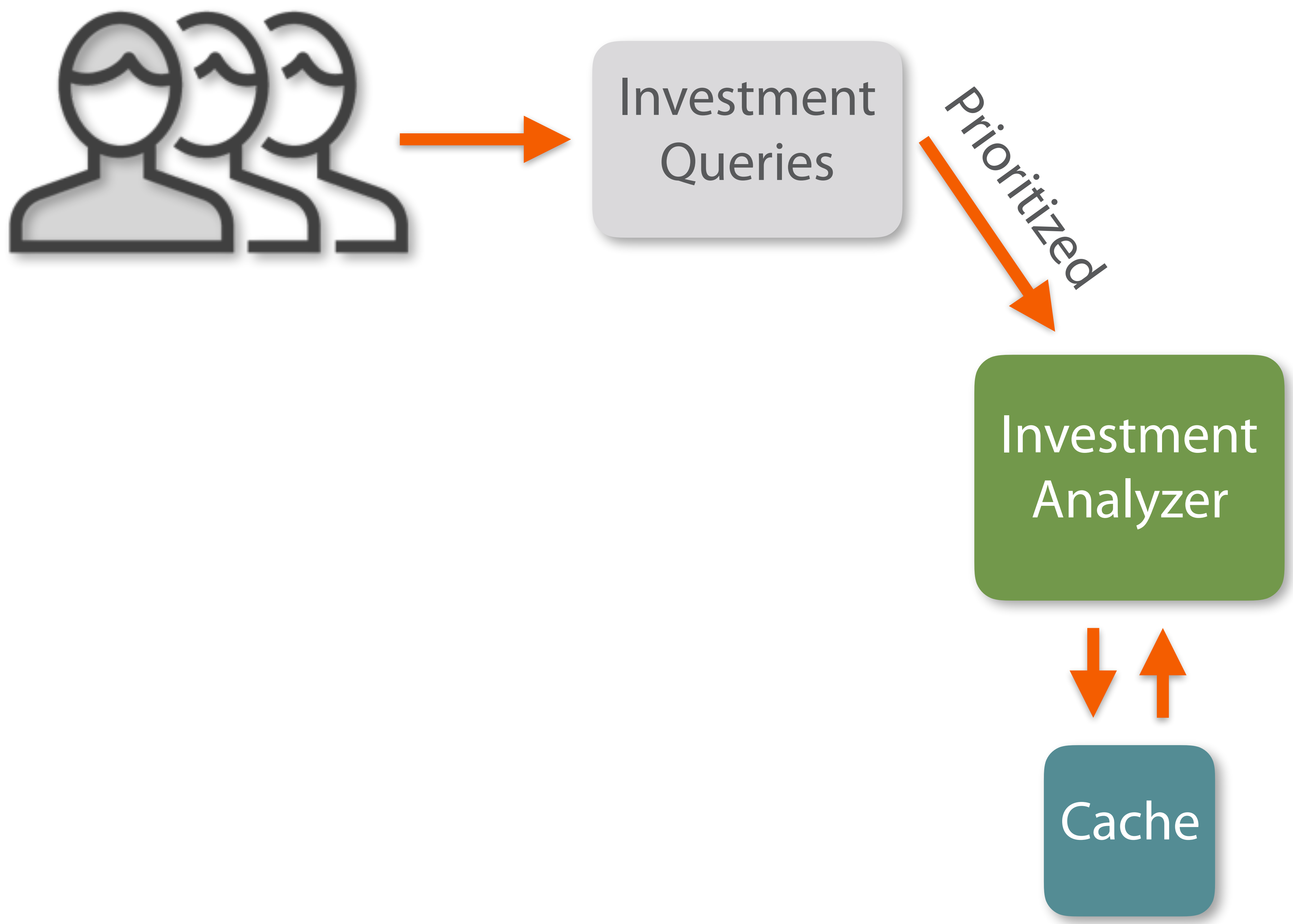


Investment Queries

Prioritized

Investment Analyzer

Cache

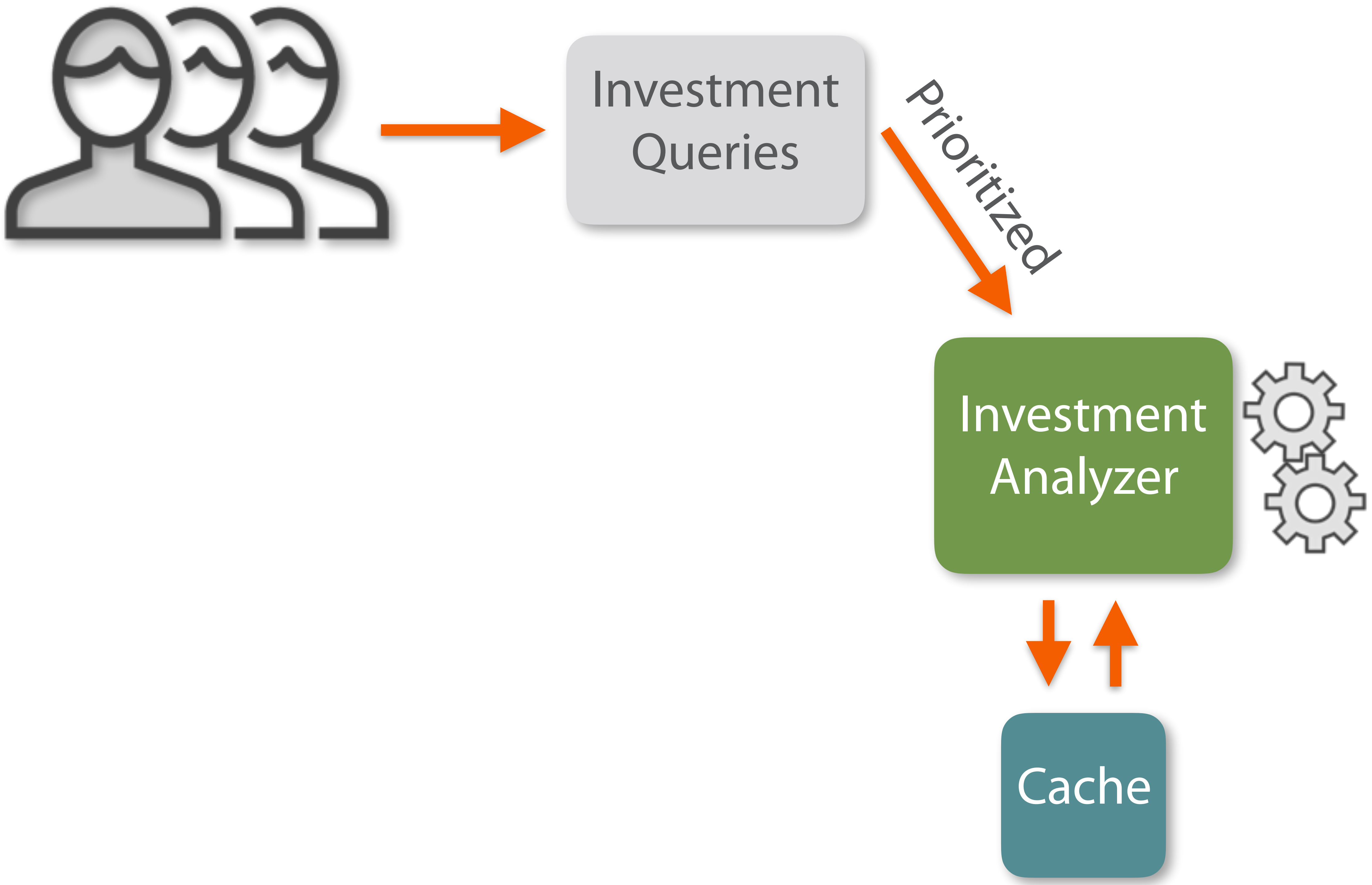


Investment Queries

Prioritized

Investment Analyzer

Cache

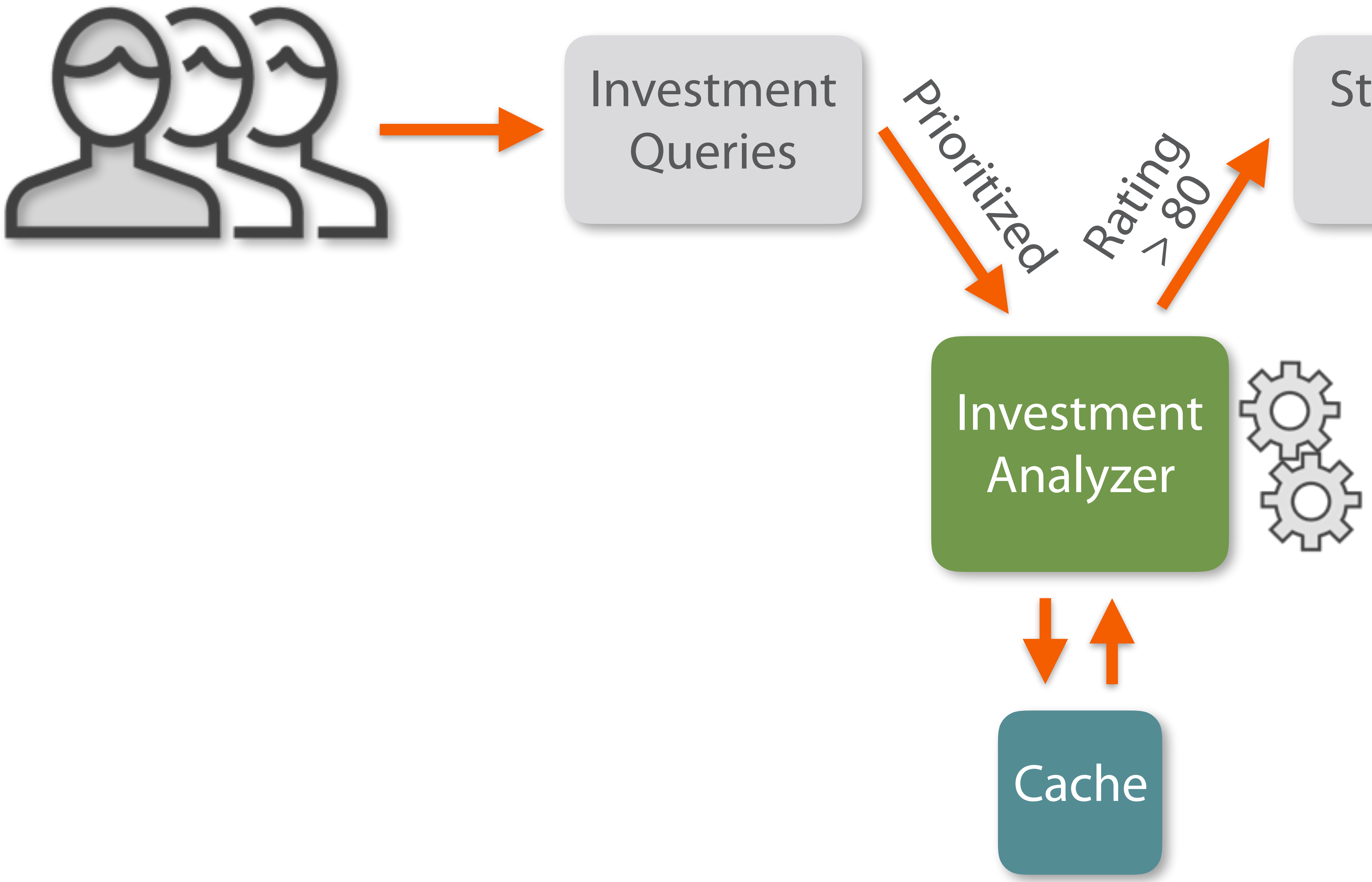


Investment Queries

Prioritized

Investment Analyzer

Cache



Investment Queries

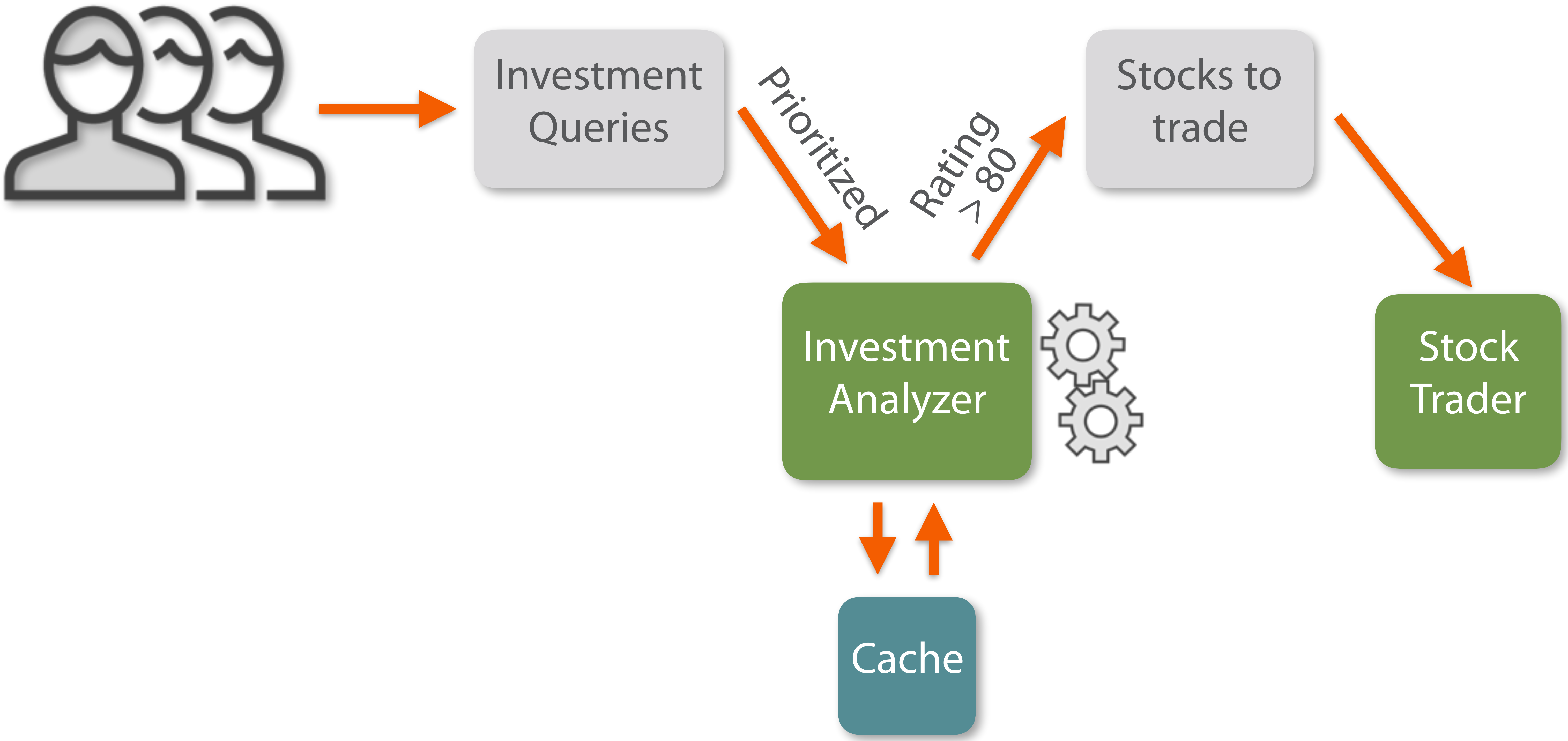
Prioritized

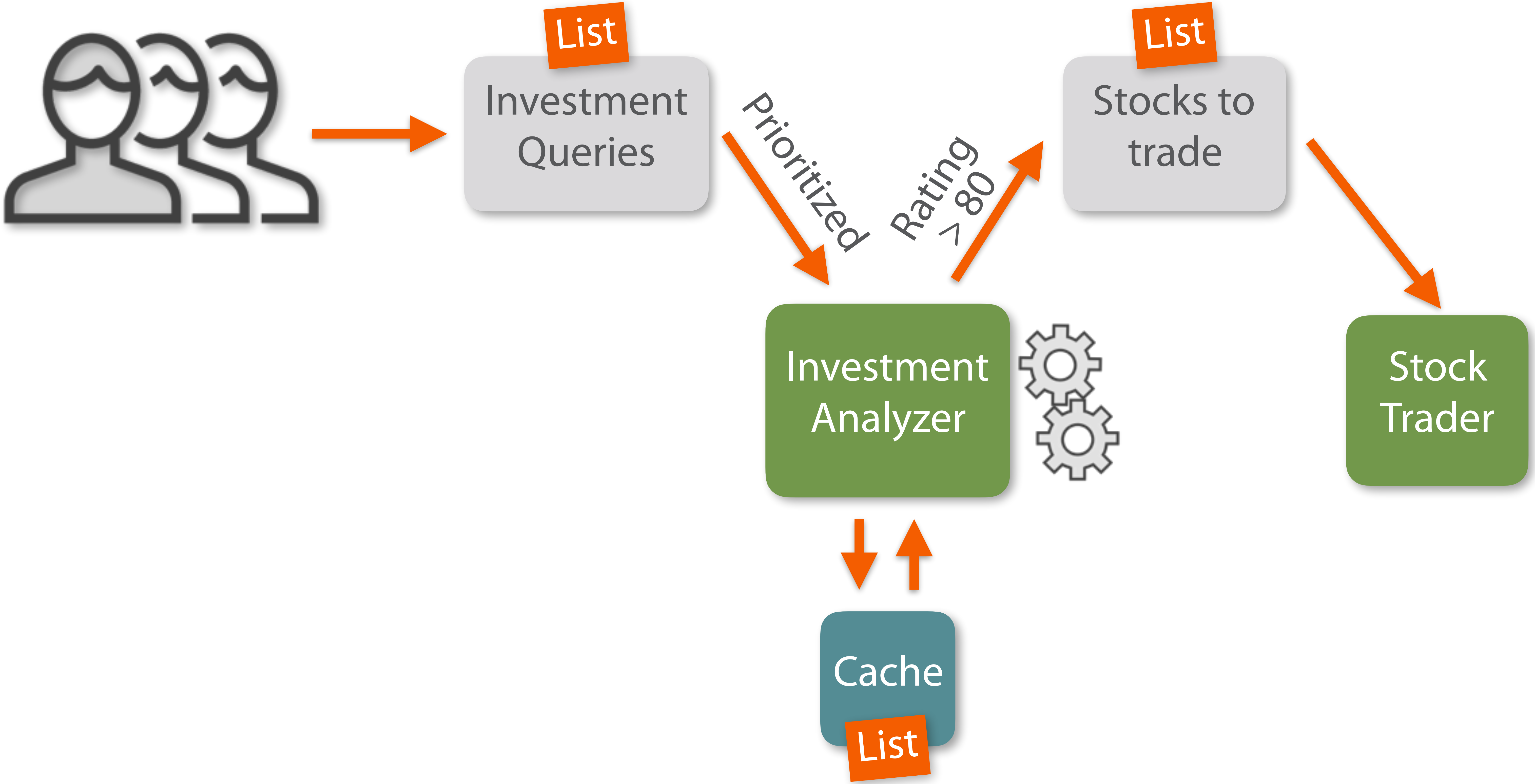
Rating > 80

Stocks to trade

Investment Analyzer

Cache


Investment Queries
Prioritized
Rating > 80
Stocks to trade
Investment Analyzer
Stock Trader
Cache



List

Investment Queries

Prioritized

Investment Analyzer

Cache

List

Rating > 80

List

Stocks to trade

Stock Trader

Investment Analyser

An investment analyzer done wrong

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

Data Structures

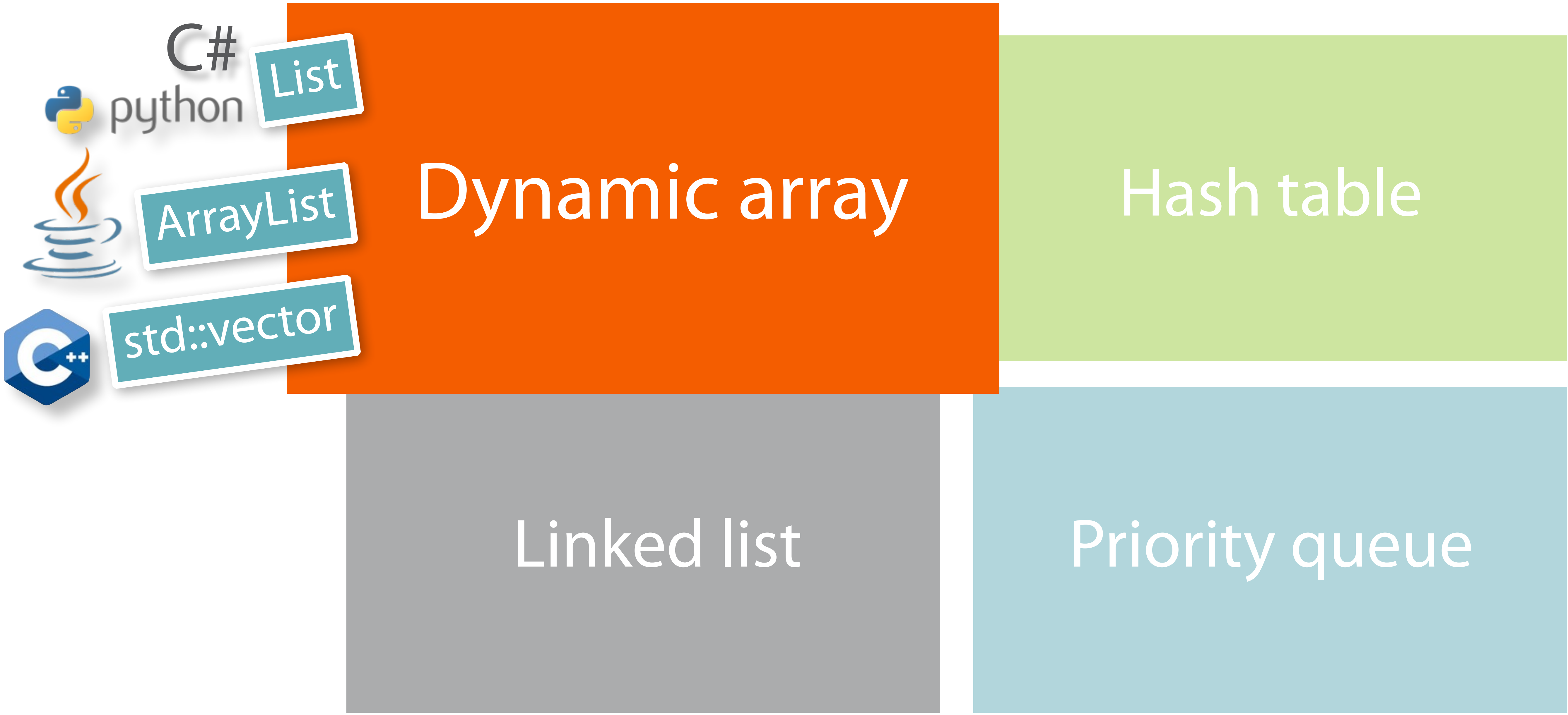
Dynamic array

Hash table

Linked list

Priority queue

Data Structures



C#

python

List

ArrayList

std::vector

Dynamic array

Hash table

Linked list

Priority queue

Add Element

Size: 8

4	*2*	*6*	*9*	*2*	*1*		

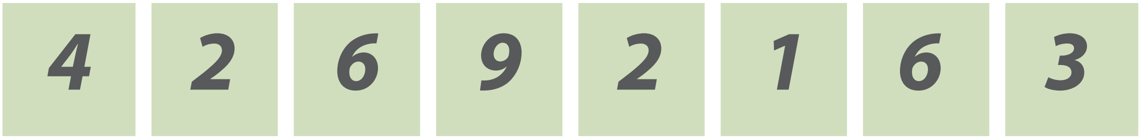
Add Element

Size: 8

4	*2*	*6*	*9*	*2*	*1*	*6*	

Add Element

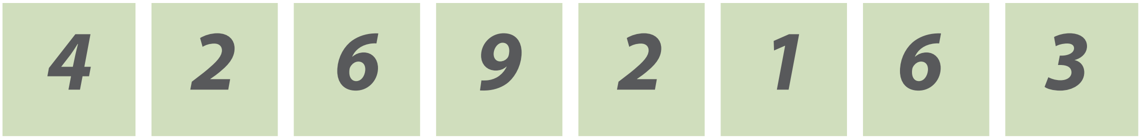
Size: 8



4	*2*	*6*	*9*	*2*	*1*	*6*	*3*

Add Element

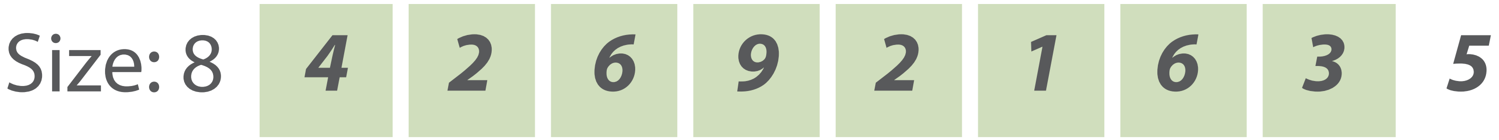
Size: 8



4	*2*	*6*	*9*	*2*	*1*	*6*	*3*

Common case: $O(1)$

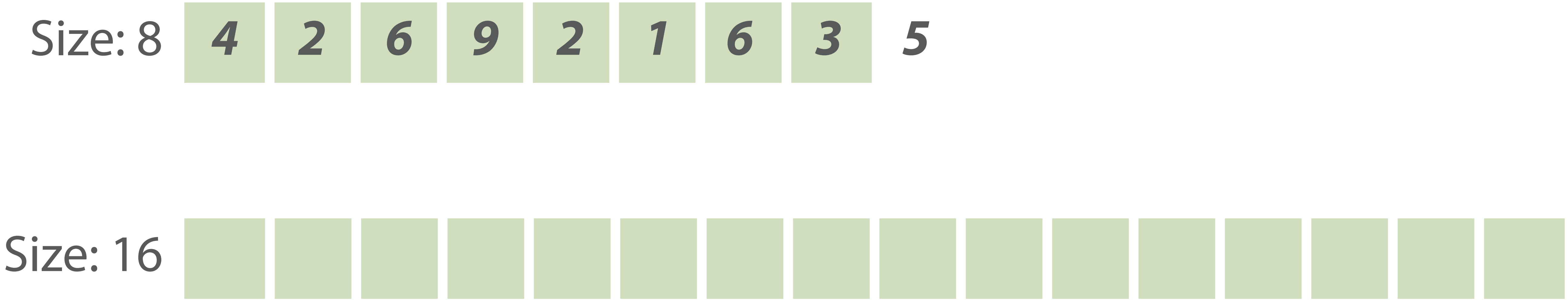
Add Element



Size: 8 *4* *2* *6* *9* *2* *1* *6* *3* *5*

Common case: $O(1)$

Add Element

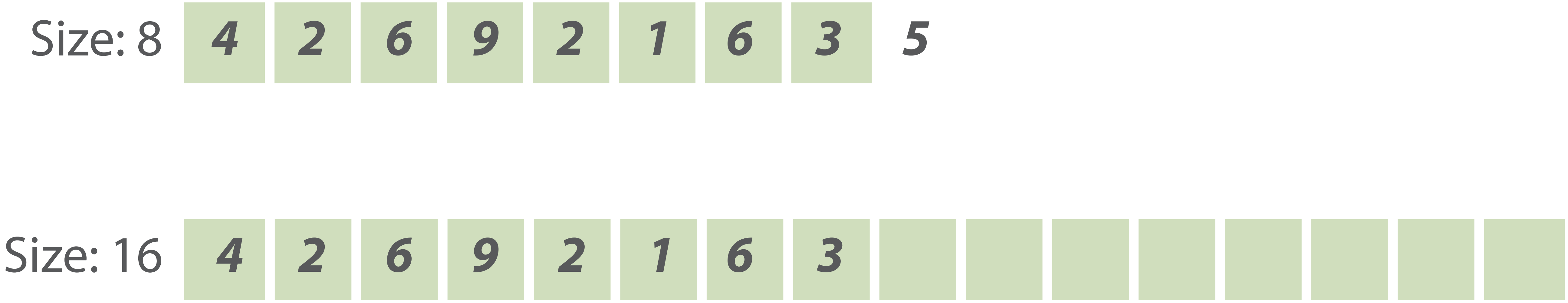


Size: 8 *4* *2* *6* *9* *2* *1* *6* *3* *5*

Size: 16

Common case: *O*(1)

Add Element



Size: 8	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*	*5*

Size: 16	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*								

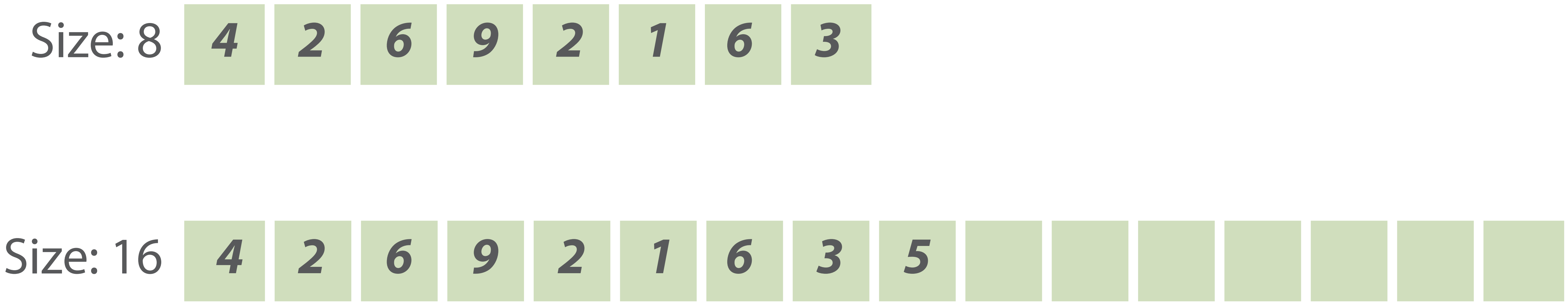
Common case: *O*(1)

Add Element

Size: 8	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*								
Size: 16	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*	*5*							

Common case: $O(1)$

Add Element



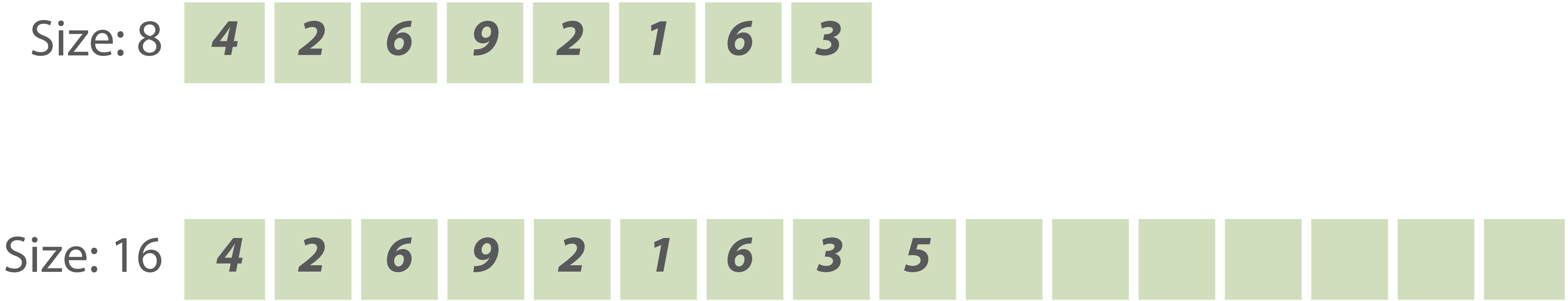
Size: 8	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*

Size: 16	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*	*5*							

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Add Element



Size: 8	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*								
Size: 16	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*	*5*							

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?

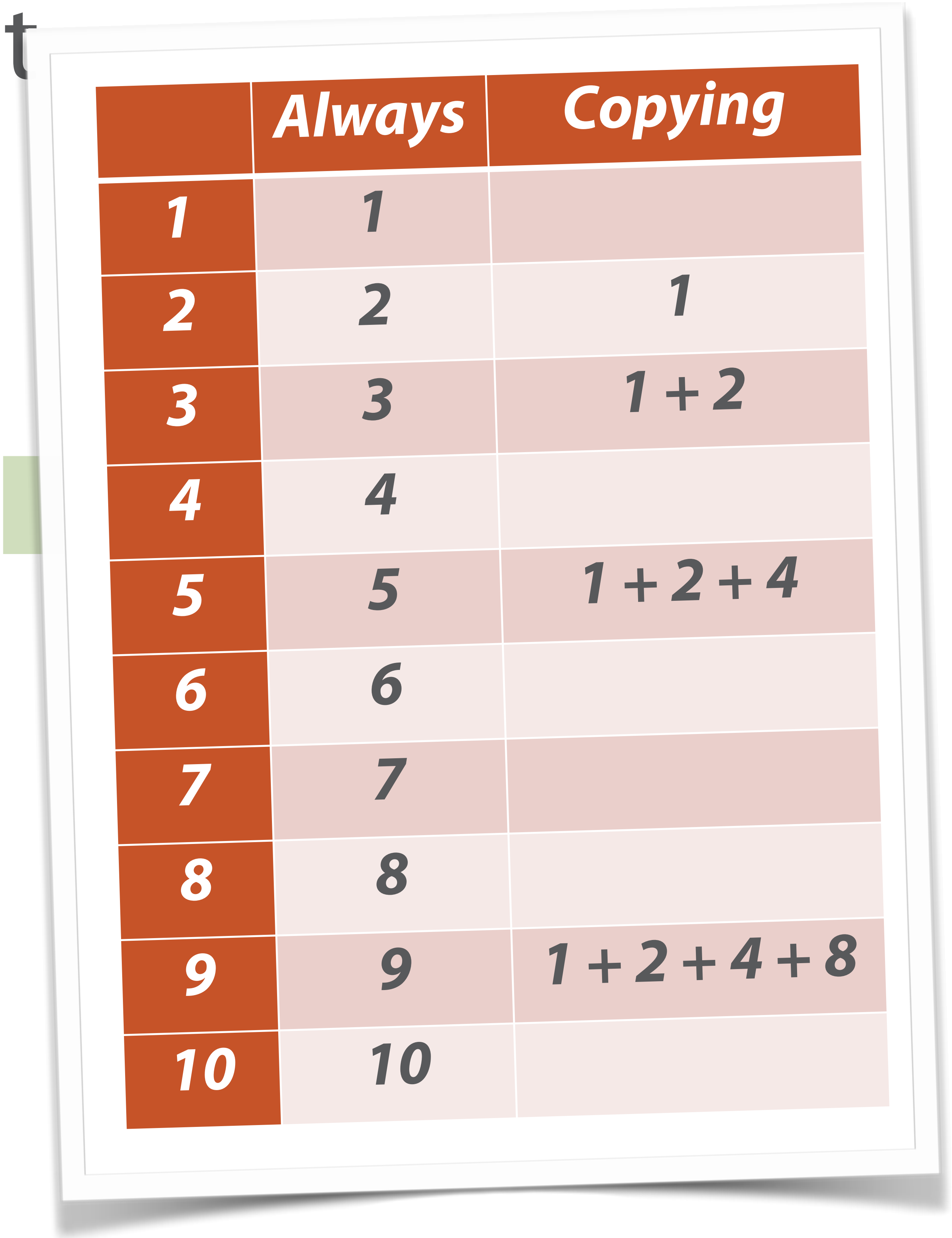
Add Element

Size: 8	4	2	6	9	2	1	6	3			
Size: 16	4	2	6	9	2	1	6	3	5		

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

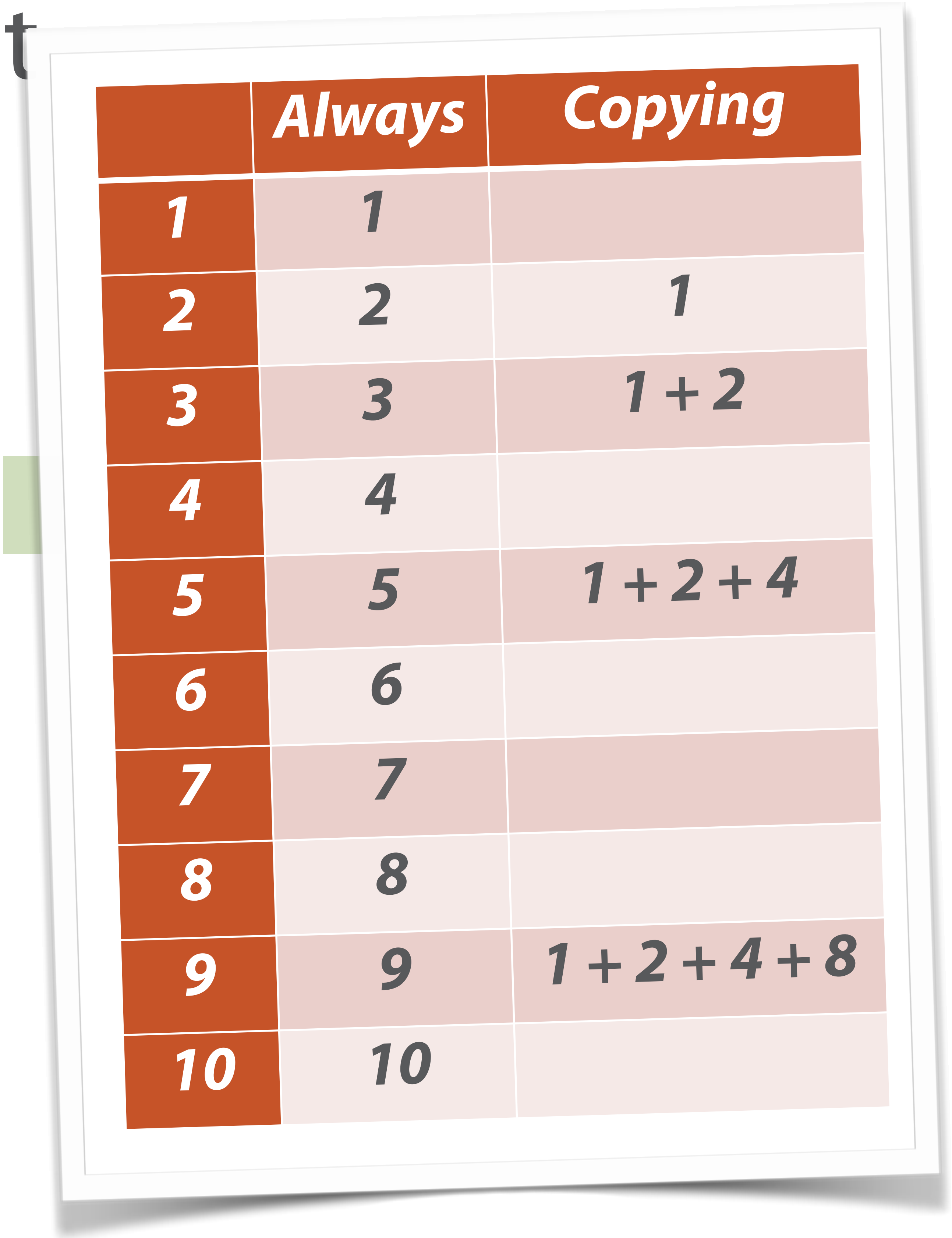
Size: 8	4	2	6	9	2	1	6	3

Size: 16	4	2	6	9	2	1	6	3	5		

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

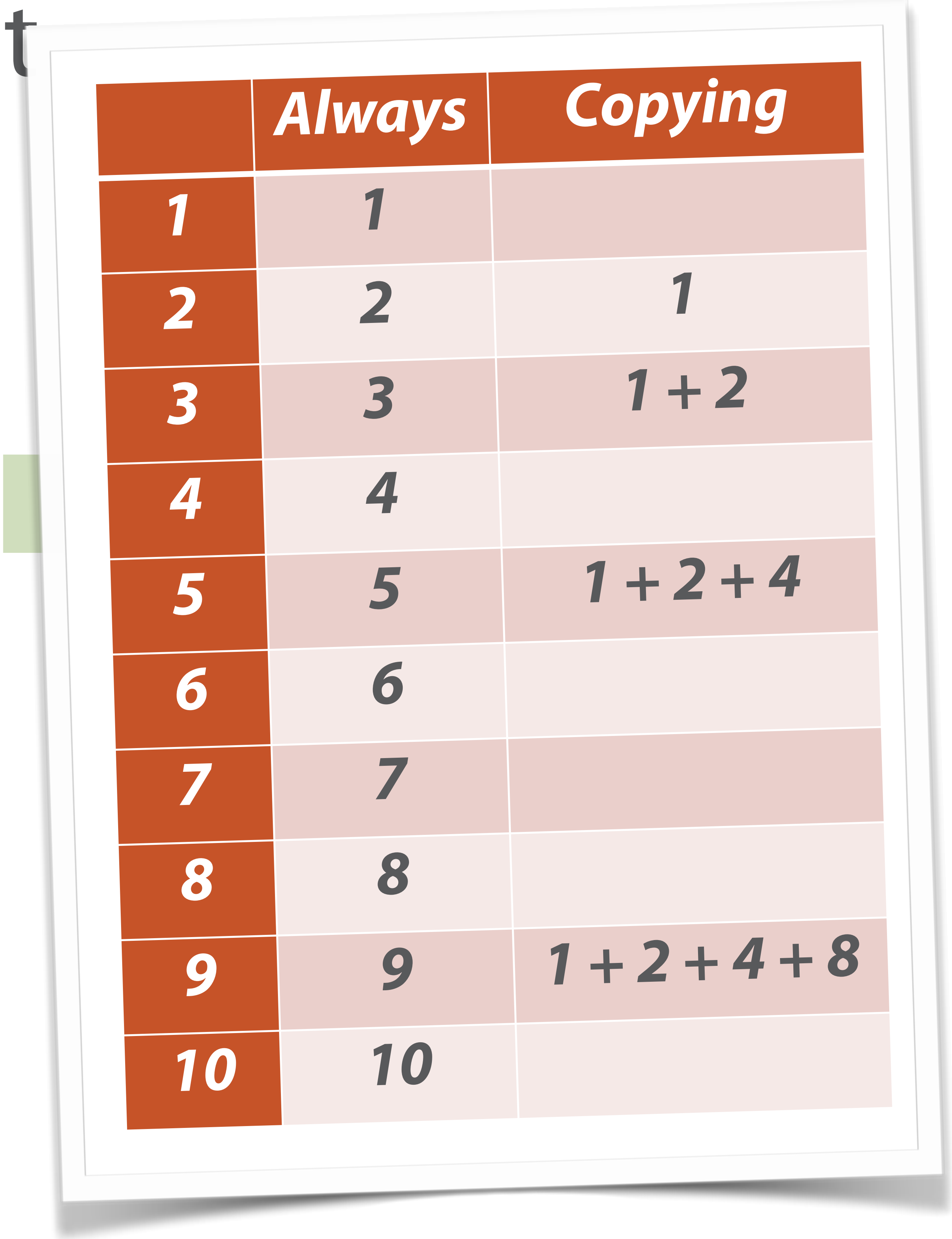
Size: 8	4	2	6	9	2	1	6	3

Size: 16	4	2	6	9	2	1	6	3	5		

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

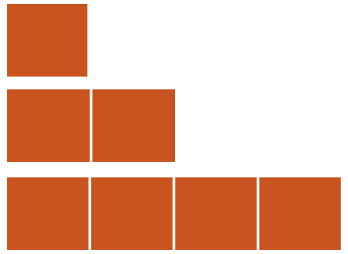
Size: 8 | 4 | 2 | 6 | 9 | 2 | 1 | 6 | 3

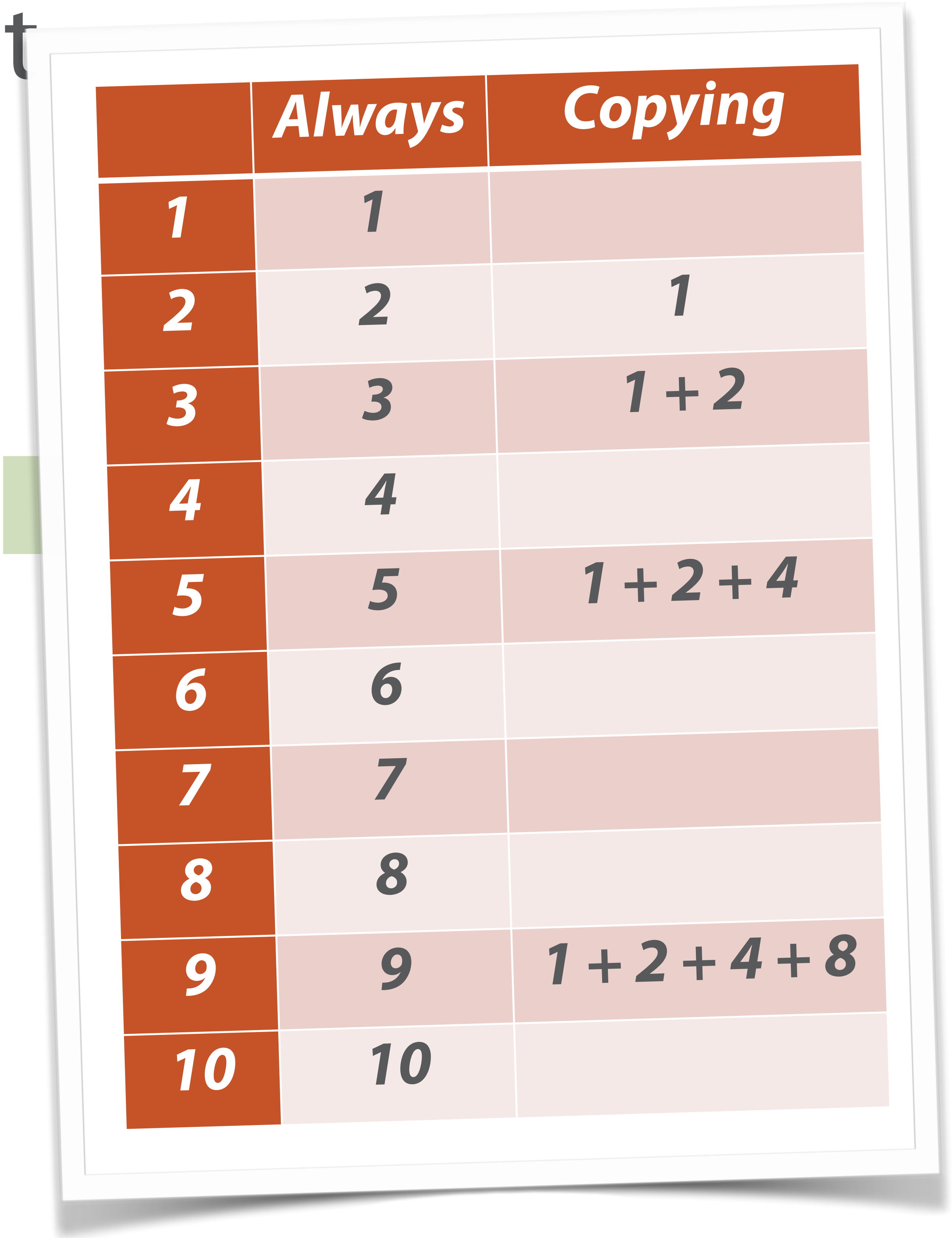
Size: 16 | 4 | 2 | 6 | 9 | 2 | 1 | 6 | 3 | 5 | |

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?





	Always	*Copying*
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

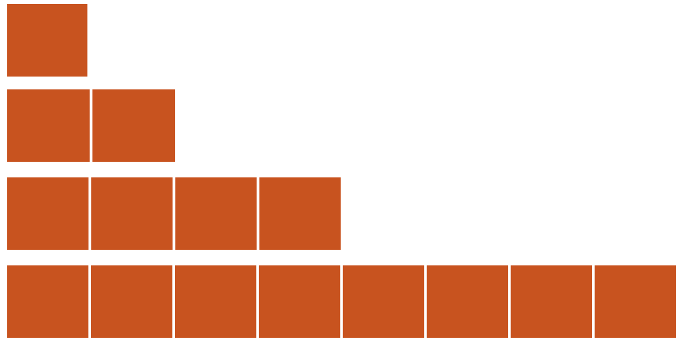
Add Element

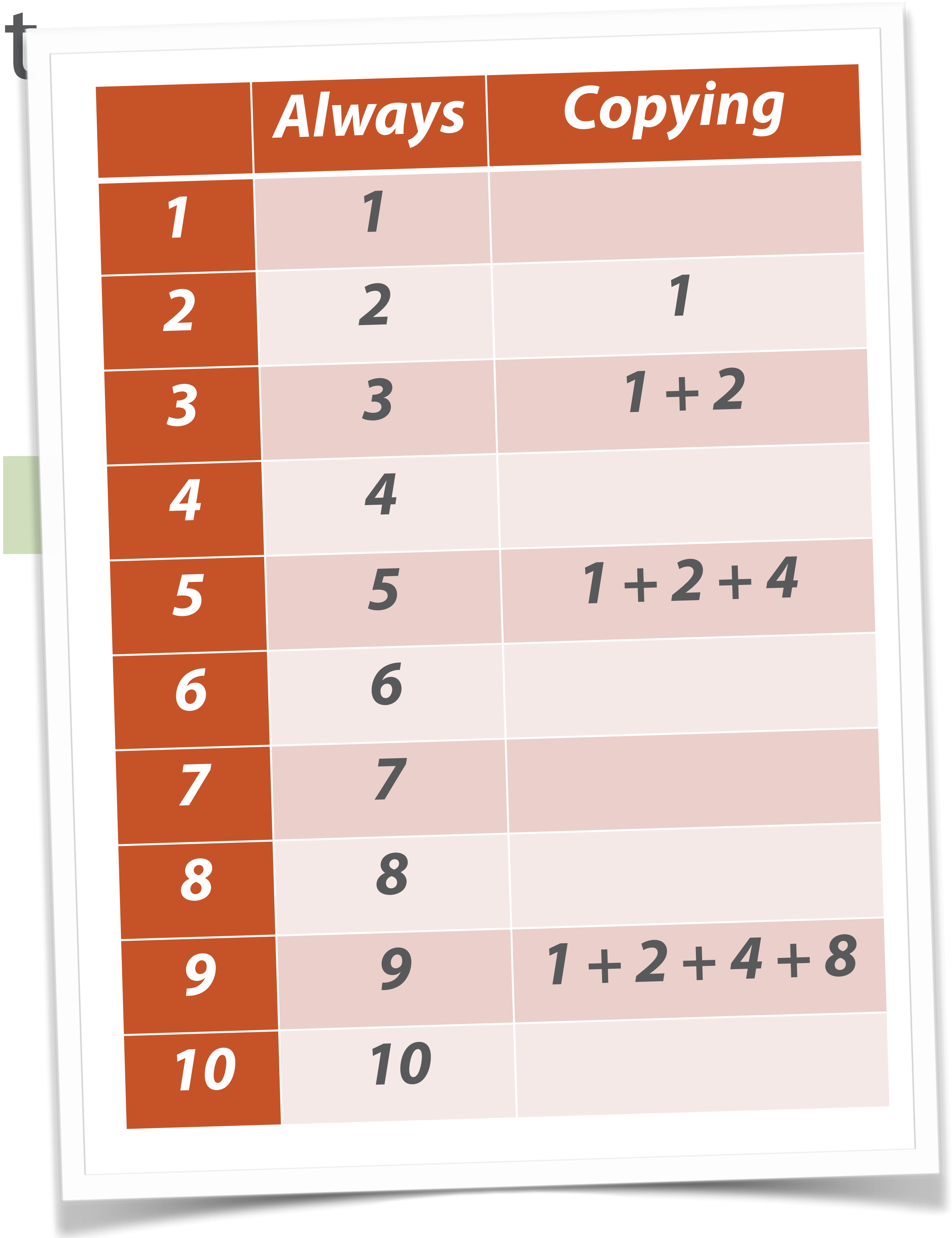
Size: 8	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*		
Size: 16	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*	*5*	

Common case: *O*(1)

Worst case: *O*(*N*) (array size: *N*)

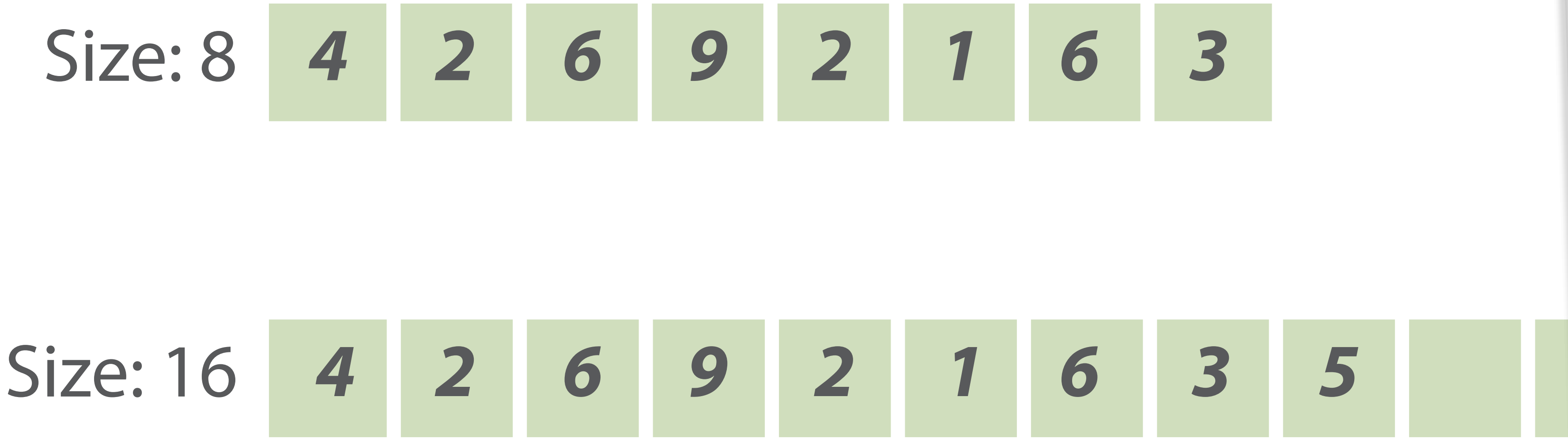
Amortized: *O*(*N*²)?





	Always	***Copying***
1	*1*	
2	*2*	*1*
3	*3*	*1 + 2*
4	*4*	
5	*5*	*1 + 2 + 4*
6	*6*	
7	*7*	
8	*8*	
9	*9*	*1 + 2 + 4 + 8*
10	*10*	

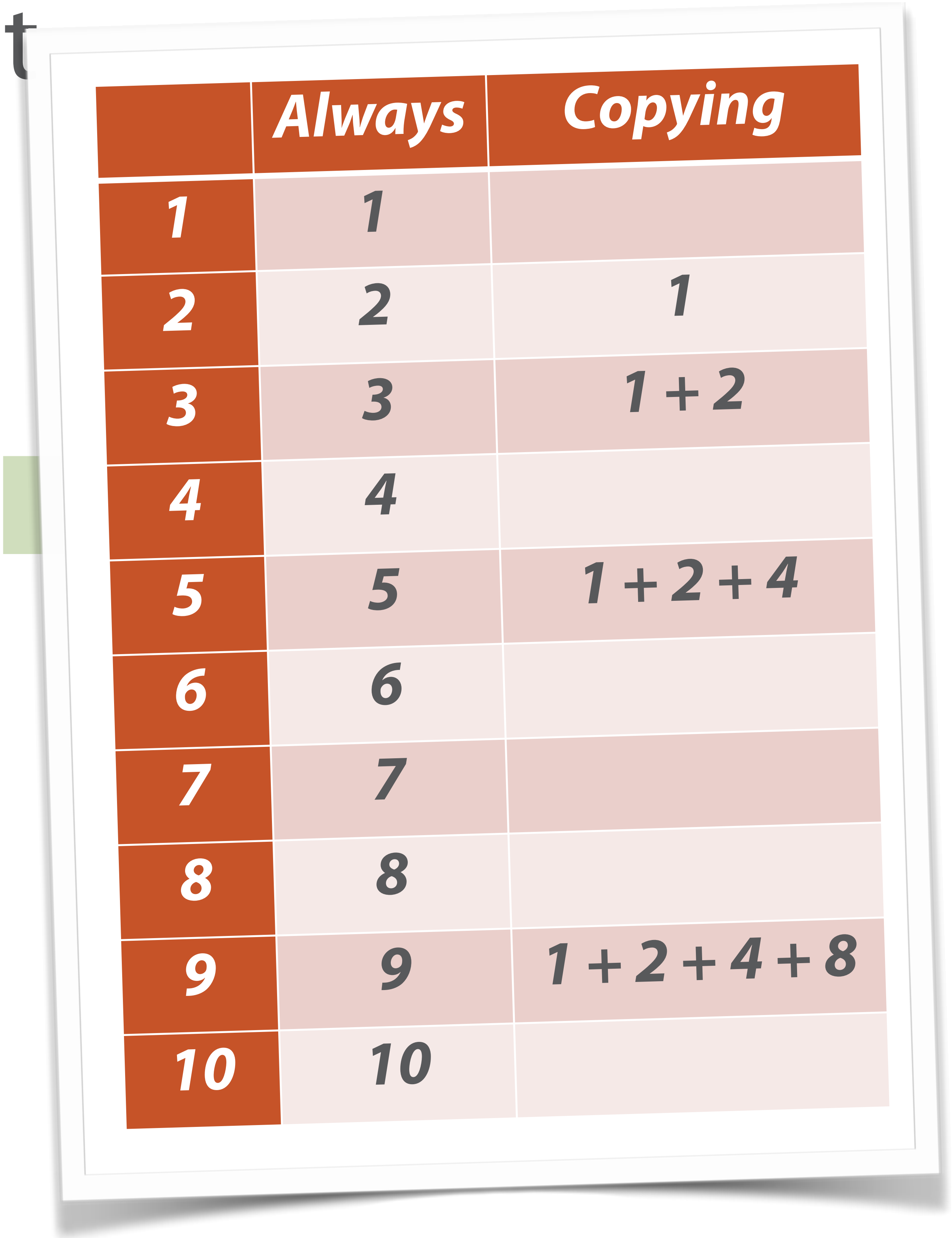
Add Element



Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?



	Always	*Copying*
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

Size: 8 | 4 | 2 | 6 | 9 | 2 | 1 | 6 | 3

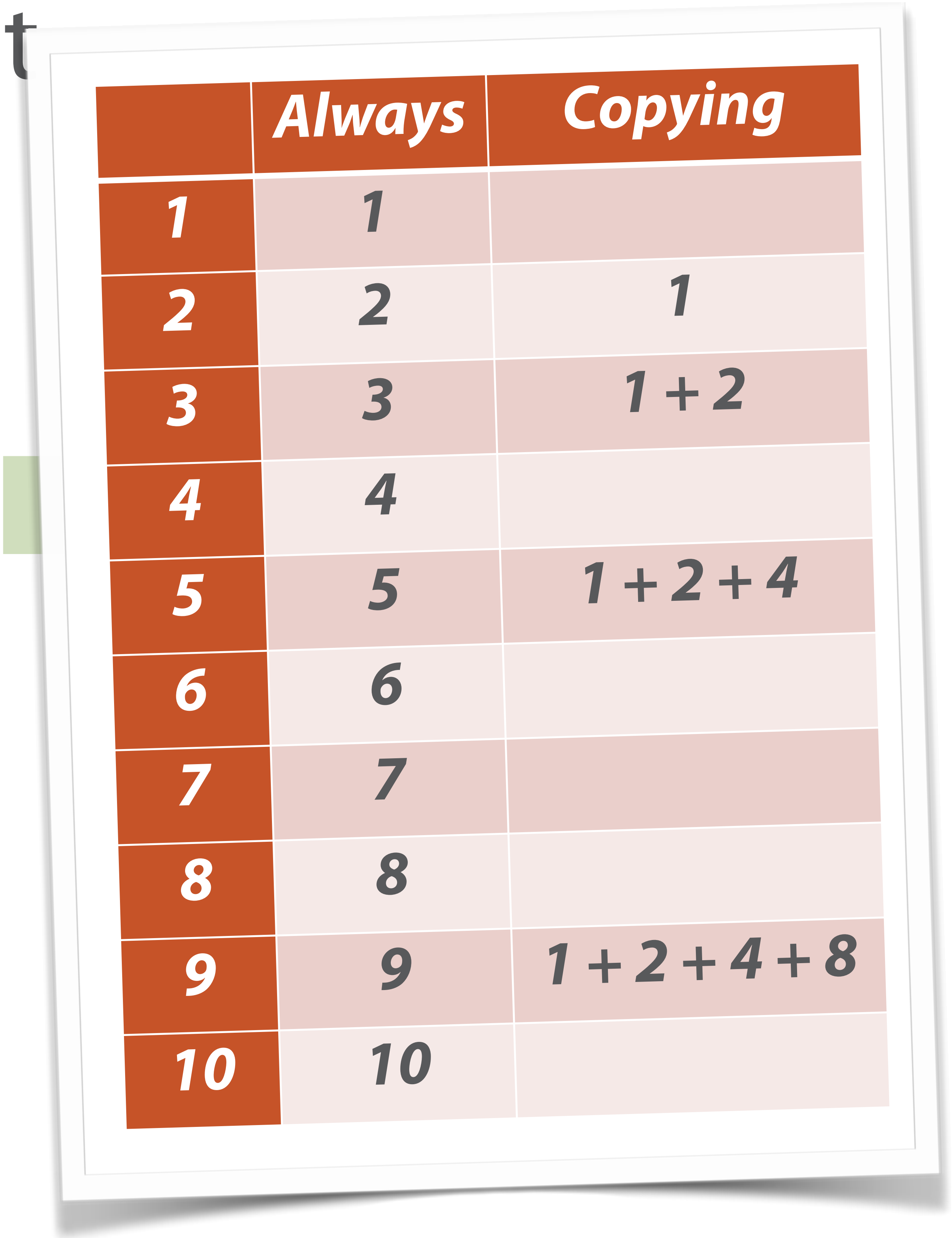
Size: 16 | 4 | 2 | 6 | 9 | 2 | 1 | 6 | 3 | 5

$9_{10} = 1001_2$

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

Size: 8 | 4 | 2 | 6 | 9 | 2 | 1 | 6 | 3

Size: 16 | 4 | 2 | 6 | 9 | 2 | 1 | 6 | 3 | 5 | |

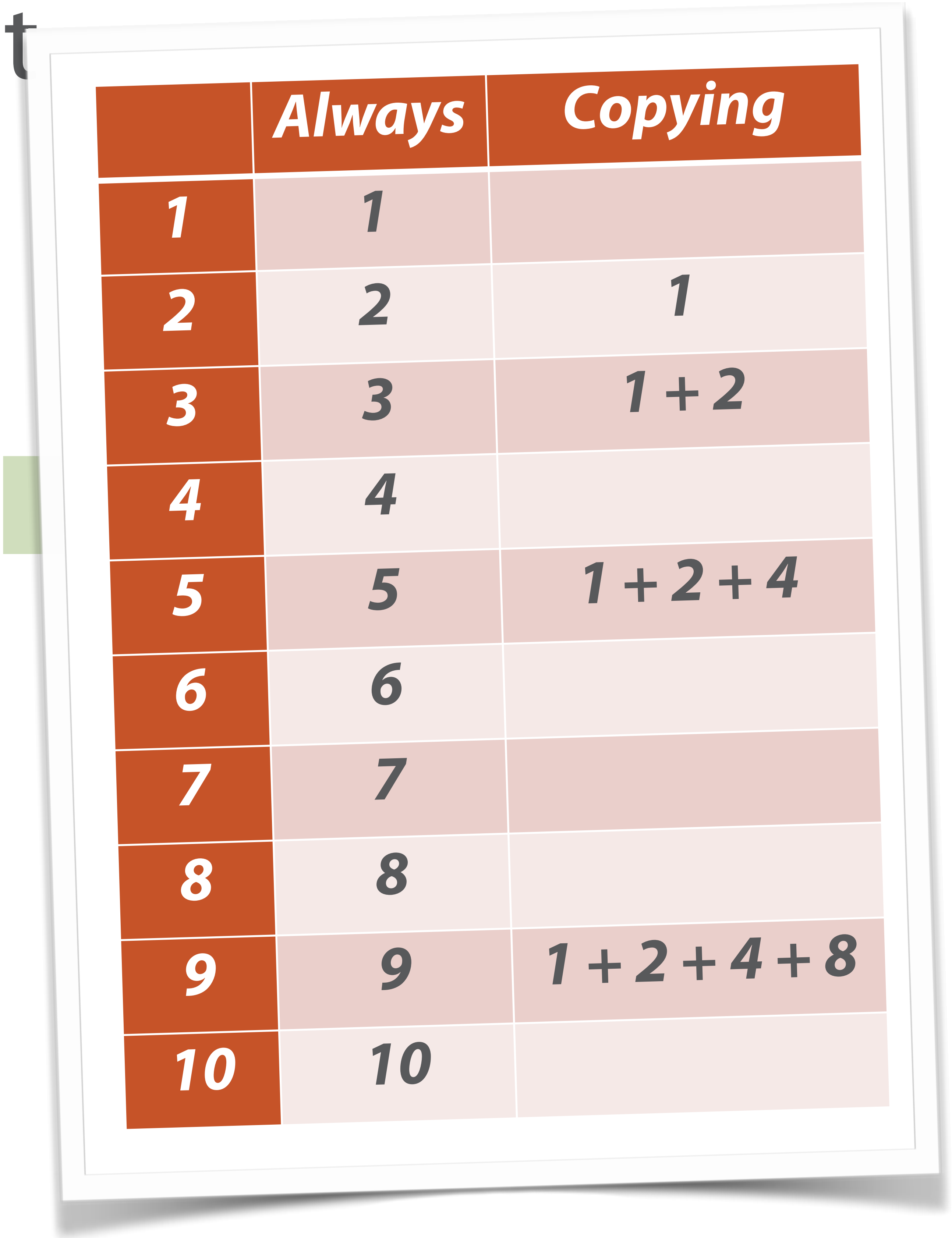
Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?

$9_{10} = 1001_2$

1111_2



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

Size: 8	4	2	6	9	2	1	6	3

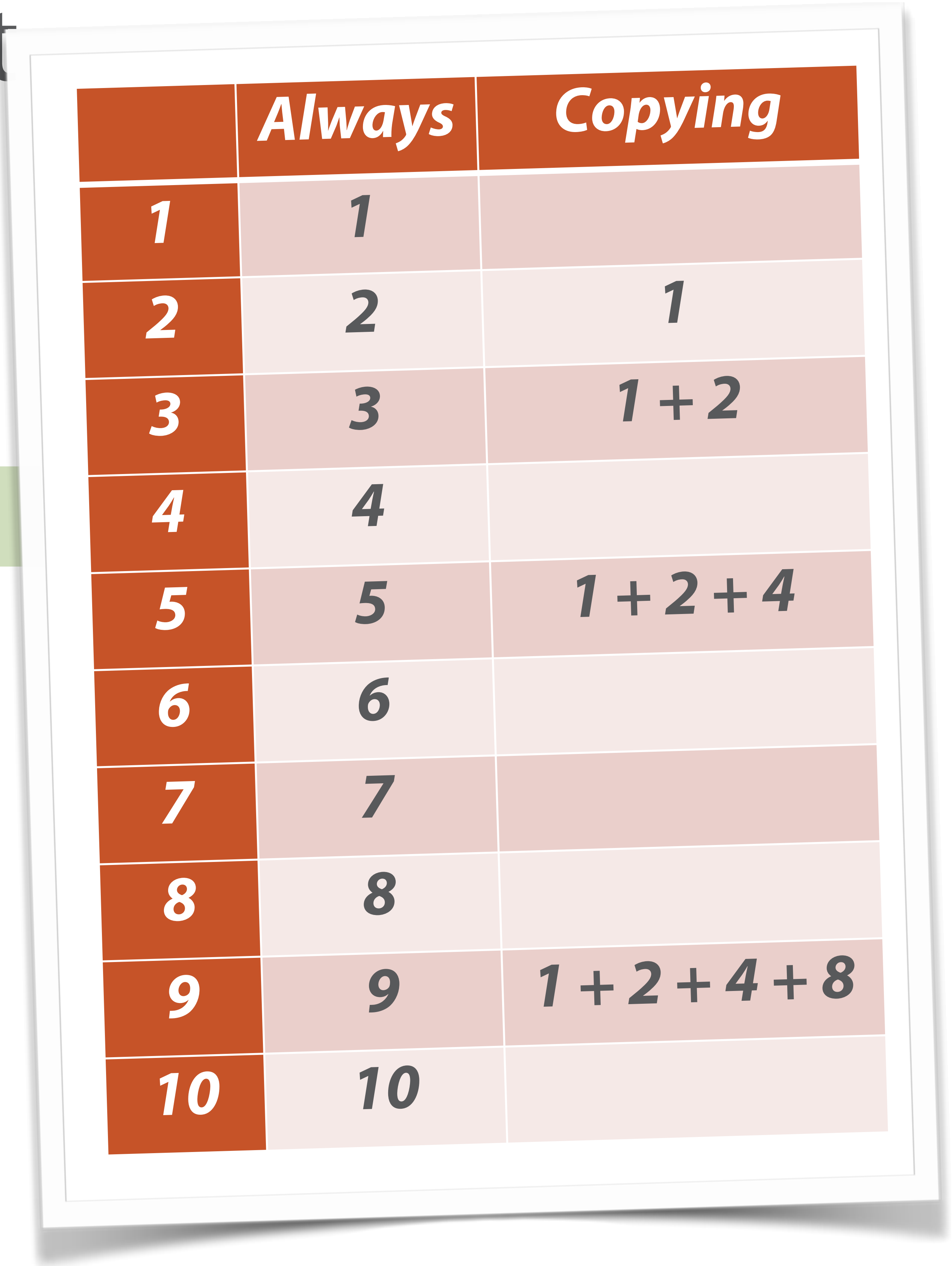
Size: 16	4	2	6	9	2	1	6	3	5		

Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?

$$\begin{array}{r} 9_{10} = 1001_2 \\ 1111_2 \\ \hline 11000_2 \end{array}$$



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

Size: 8								
	4	2	6	9	2	1	6	3

Size: 16											
	4	2	6	9	2	1	6	3	5		

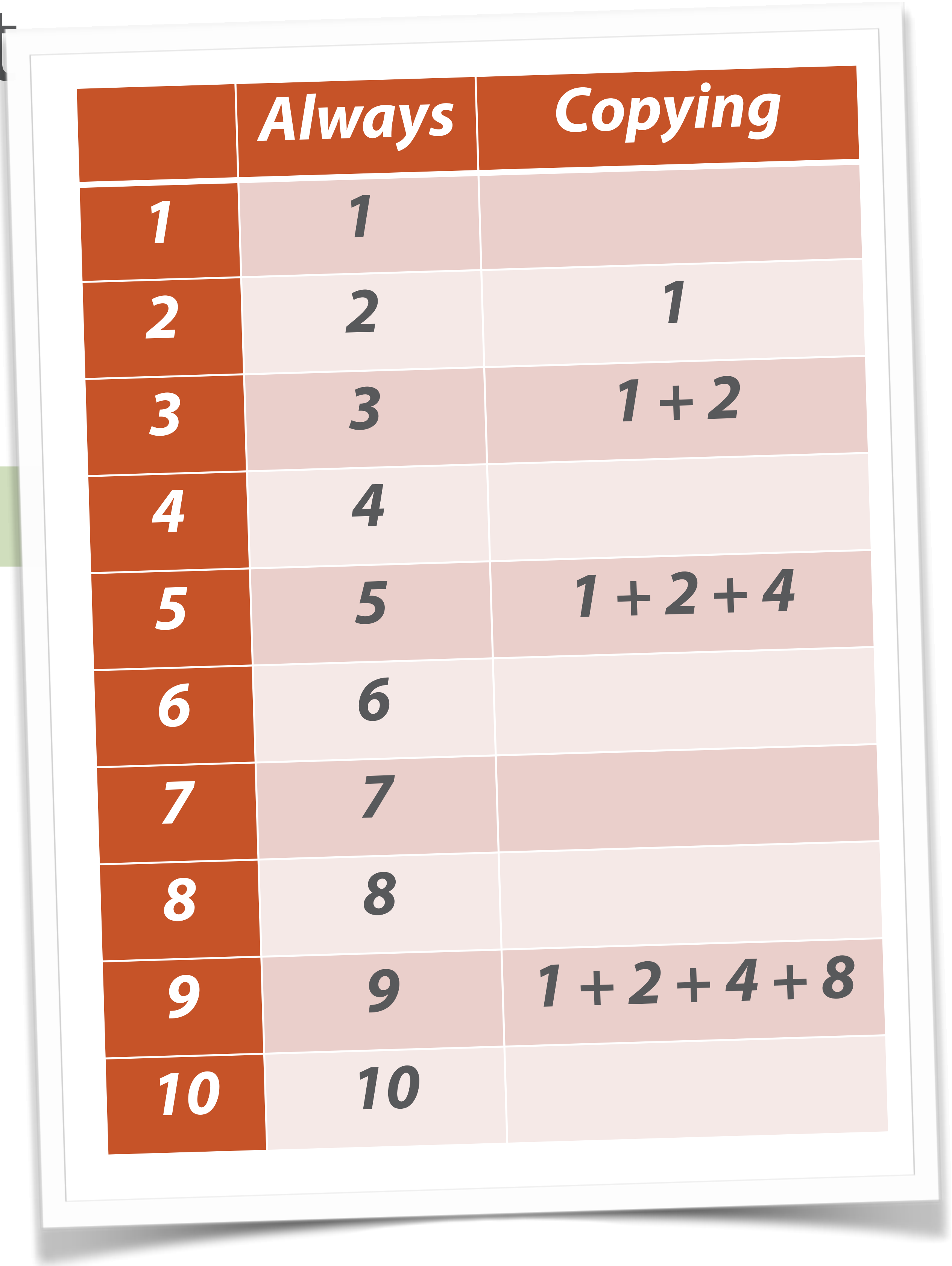
Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?

$$\begin{array}{r} 9_{10} = 1001_2 \\ 1111_2 \\ \hline 11000_2 \end{array}$$

$$17_{10} = 10001_2$$



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

Size: 8	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*

Size: 16	*4*	*2*	*6*	*9*	*2*	*1*	*6*	*3*	*5*		

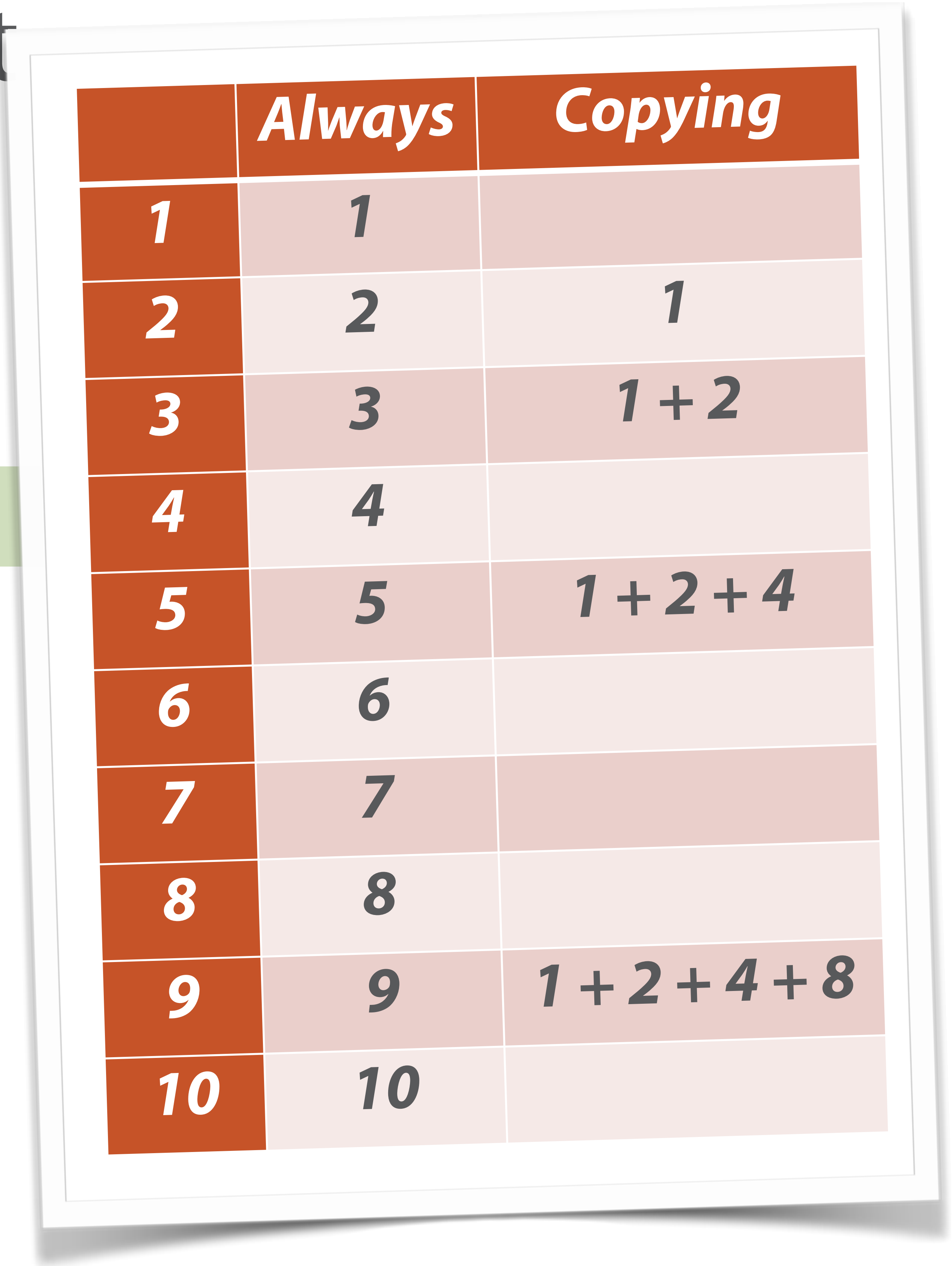
Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?

$$\begin{array}{r} 9_{10} = 1001_2 \\ 1111_2 \\ \hline 11000_2 \end{array}$$

$$\begin{array}{r} 17_{10} = 10001_2 \\ 11111_2 \end{array}$$



	Always	*Copying*
1	*1*	
2	*2*	*1*
3	*3*	*1 + 2*
4	*4*	
5	*5*	*1 + 2 + 4*
6	*6*	
7	*7*	
8	*8*	
9	*9*	*1 + 2 + 4 + 8*
10	*10*	

Add Element

Size: 8								
	4	2	6	9	2	1	6	3

Size: 16										
	4	2	6	9	2	1	6	3	5	

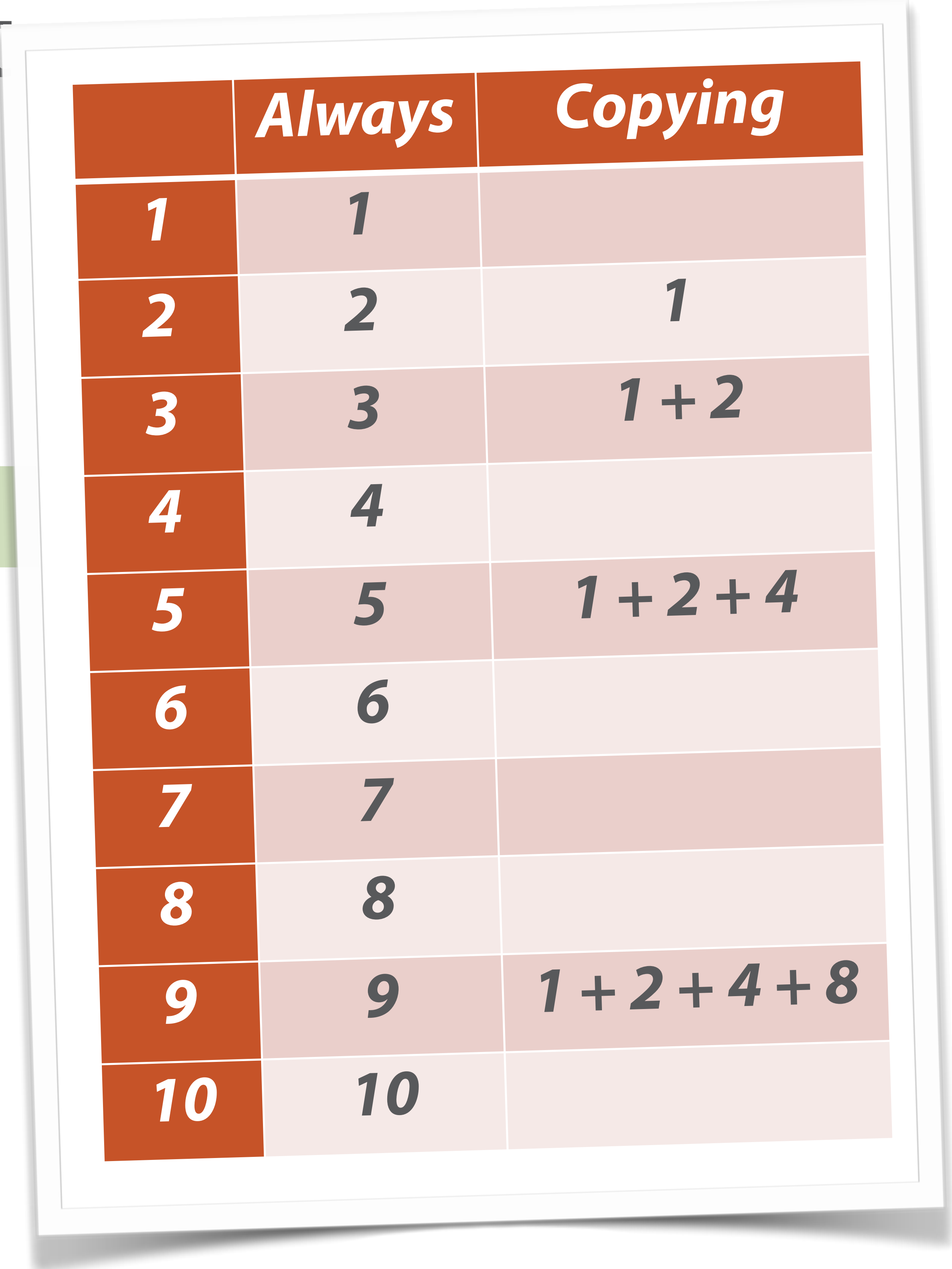
Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: $O(N^2)$?

$$\begin{array}{r} 9_{10} = 1001_2 \\ 1111_2 \\ \hline 11000_2 \end{array}$$

$$\begin{array}{r} 17_{10} = 10001_2 \\ 11111_2 \\ \hline 110000_2 \end{array}$$



	Always	Copying
1	1	
2	2	1
3	3	1 + 2
4	4	
5	5	1 + 2 + 4
6	6	
7	7	
8	8	
9	9	1 + 2 + 4 + 8
10	10	

Add Element

Size: 8

4	*2*	*6*	*9*	*2*	*1*	*6*	*3*

Size: 16

4	*2*	*6*	*9*	*2*	*1*	*6*	*3*	*5*		

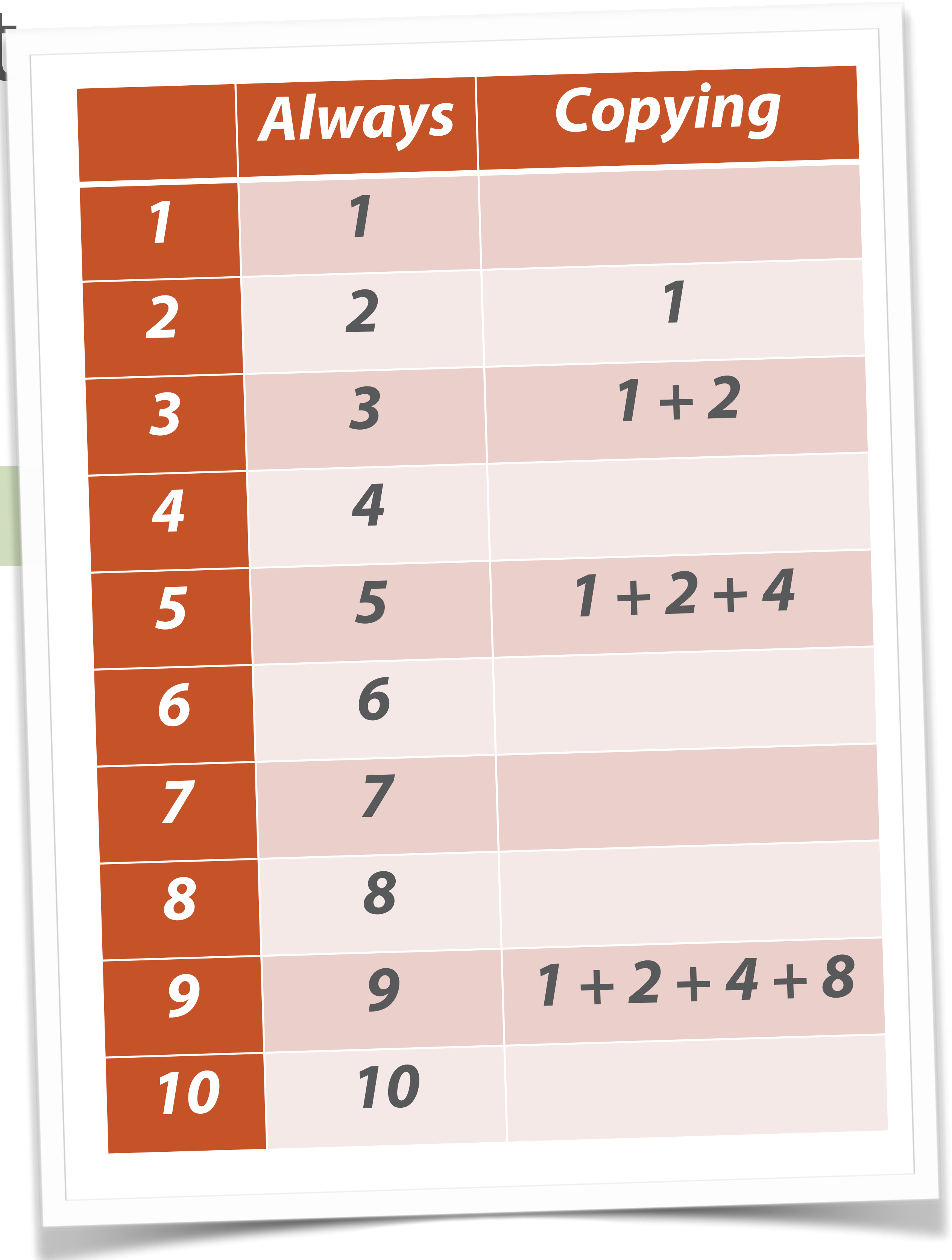
Common case: $O(1)$

Worst case: $O(N)$ (array size: N)

Amortized: ~~$O(N^2)$~~? $O(N)$

$$\begin{array}{r} 9_{10} = 1001_2 \\ 1111_2 \\ \hline 11000_2 \end{array}$$

$$\begin{array}{r} 17_{10} = 10001_2 \\ 11111_2 \\ \hline 110000_2 \end{array}$$

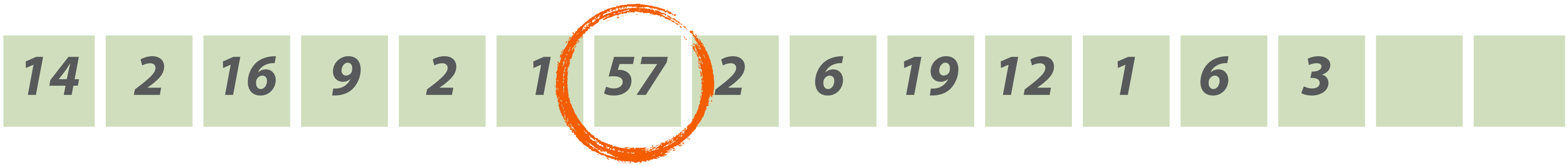


	Always	***Copying***
1	*1*	
2	*2*	*1*
3	*3*	*1 + 2*
4	*4*	
5	*5*	*1 + 2 + 4*
6	*6*	
7	*7*	
8	*8*	
9	*9*	*1 + 2 + 4 + 8*
10	*10*	

Remove Element

14	*2*	*16*	*9*	*2*	*1*	*57*	*2*	*6*	*19*	*12*	*1*	*6*	*3*		

Remove Element

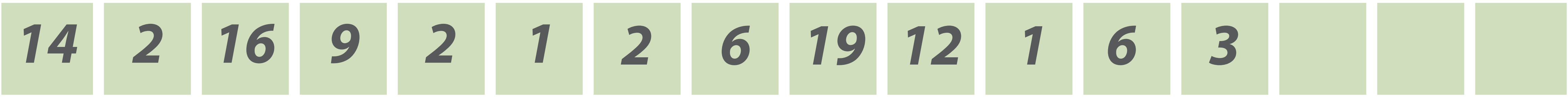


14	2	16	9	2	1	57	2	6	19	12	1	6	3		

Remove Element

14	*2*	*16*	*9*	*2*	*1*	*2*	*6*	*19*	*12*	*1*	*6*	*3*			

Remove Element



14	*2*	*16*	*9*	*2*	*1*	*2*	*6*	*19*	*12*	*1*	*6*	*3*			

Array size: *N*

Remove Element

14	*2*	*16*	*9*	*2*	*1*	*2*	*6*	*19*	*12*	*1*	*6*	*3*			

Array size: *N*

Worst case: *O*(*N*)

Find Element

14	2	16	9	2	1	57	3	2	19	12	3	1	0	75	13

Find Element

Element to find: 891

14	*2*	*16*	*9*	*2*	*1*	*57*	*3*	*2*	*19*	*12*	*3*	*1*	*0*	*75*	*13*

Find Element

Element to find: 891

14	*2*	*16*	*9*	*2*	*1*	*57*	*3*	*2*	*19*	*12*	*3*	*1*	*0*	*75*	*13*

Find Element

Element to find: 891

14	*2*	*16*	*9*	*2*	*1*	*57*	*3*	*2*	*19*	*12*	*3*	*1*	*0*	*75*	*13*

Array size: *N*

Worst case: *O*(*N*)

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

C#

LinkedList

std::list

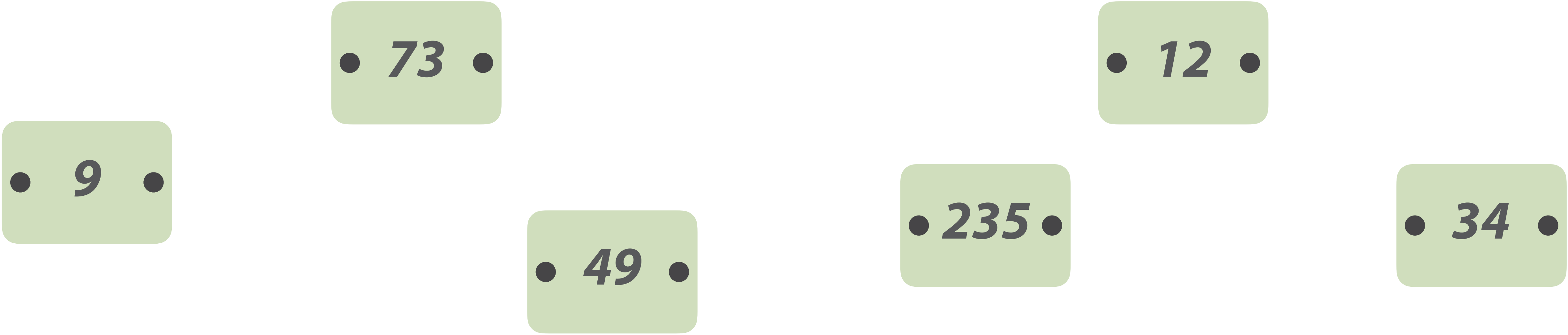
python

collections.deque

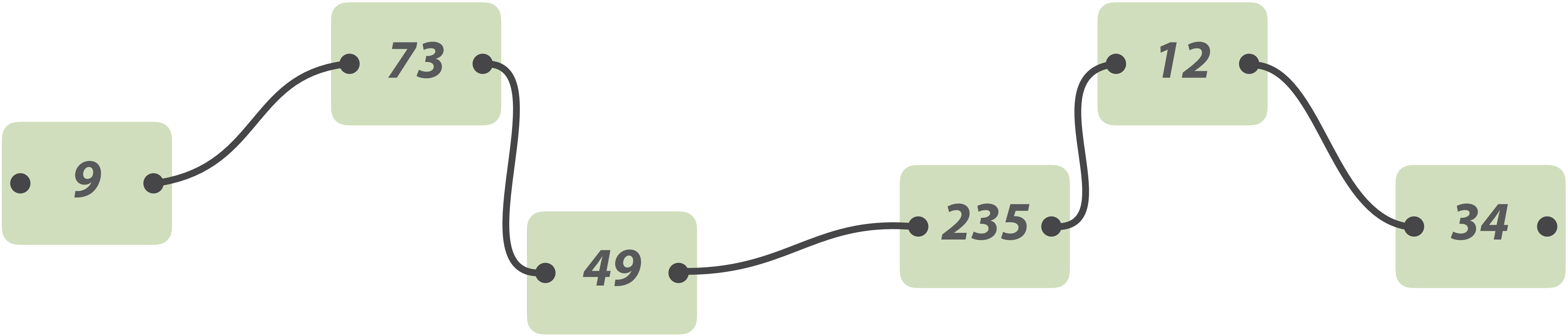
(Doubly) Linked List

14	*2*	*16*	*9*	*2*	*1*	*57*	*2*	*6*	*19*	*12*	*1*	*6*	*3*		

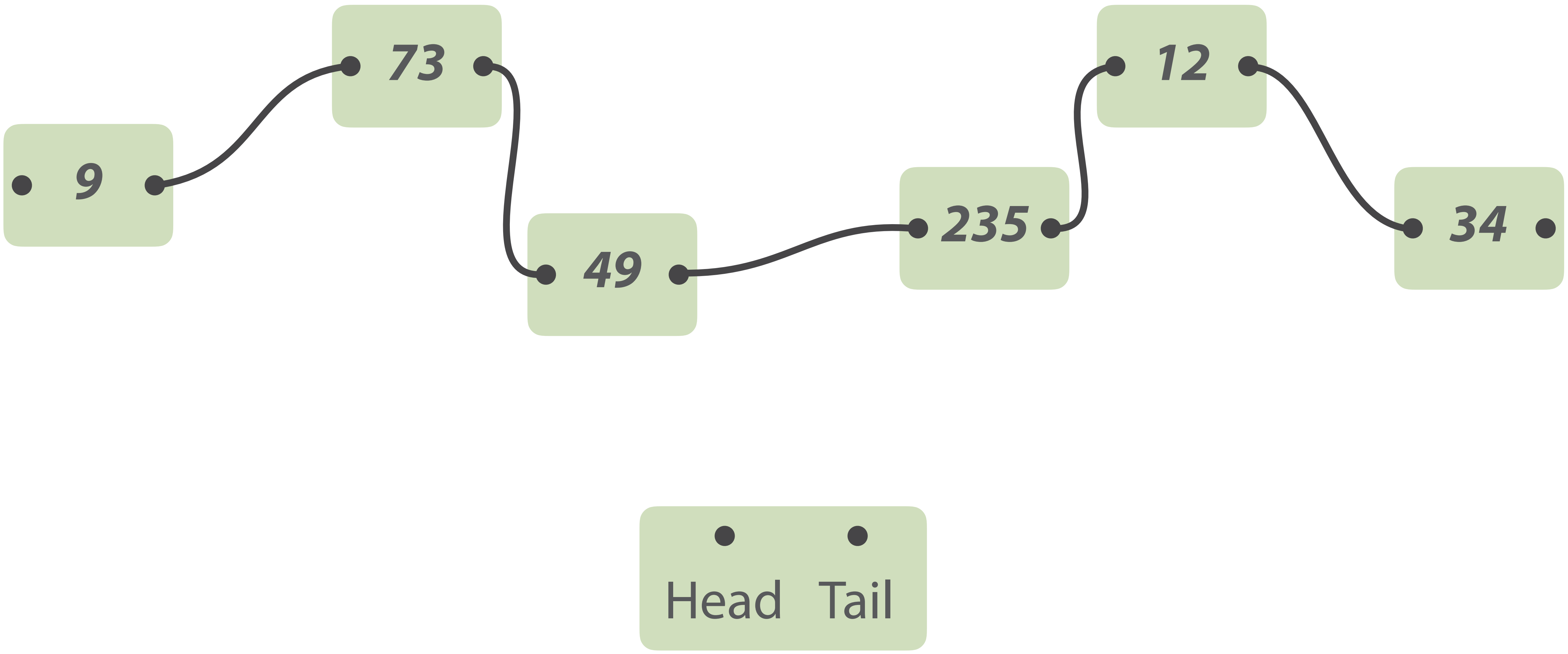
(Doubly) Linked List



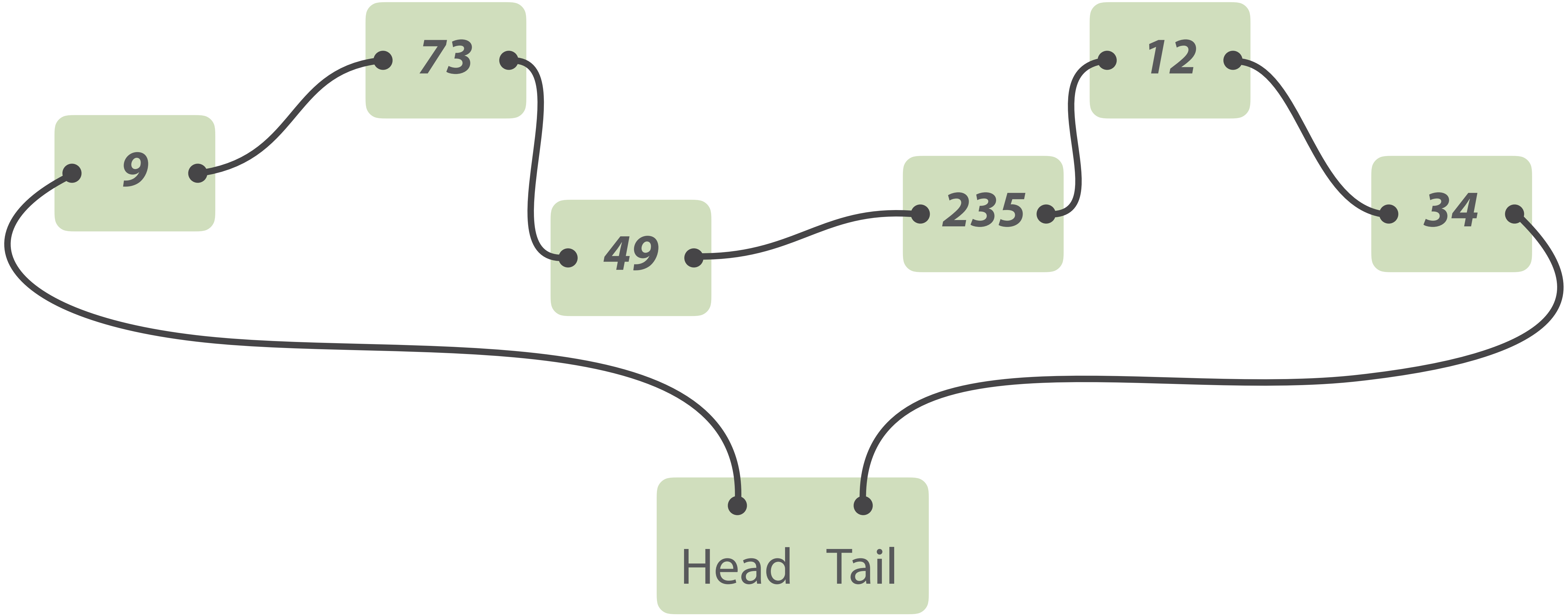
(Doubly) Linked List



(Doubly) Linked List

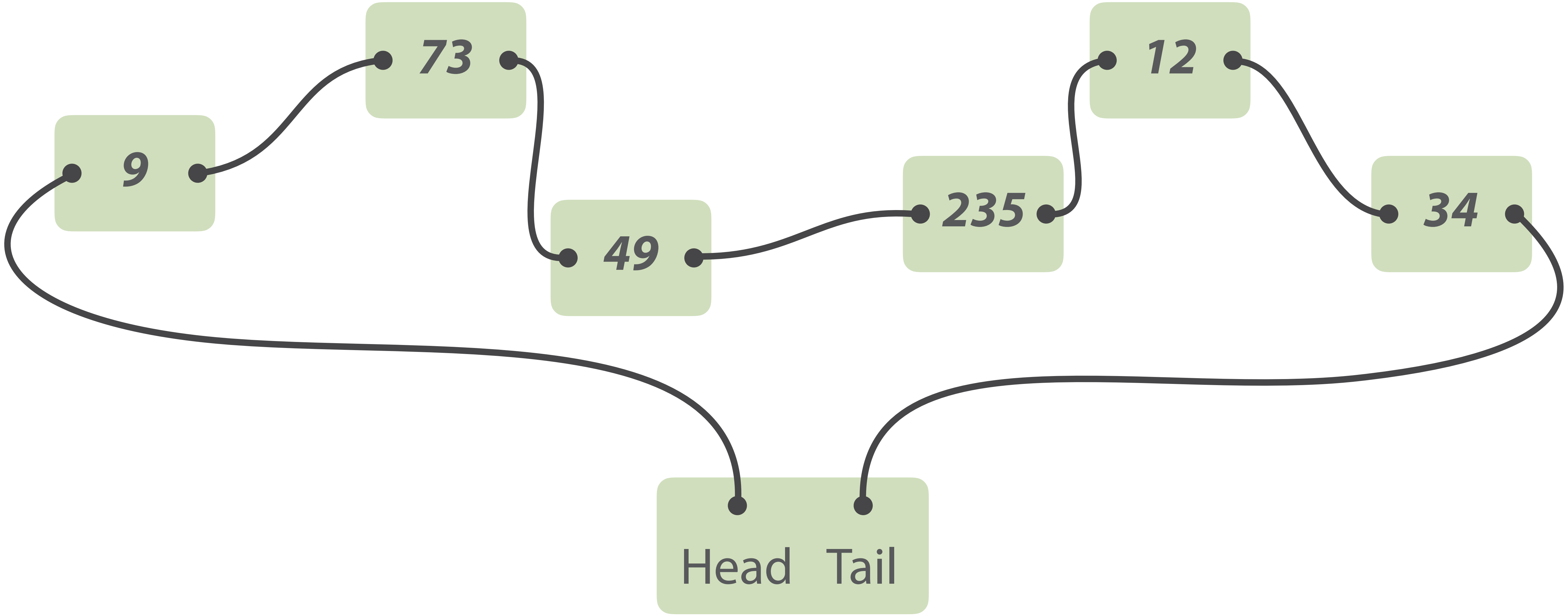


(Doubly) Linked List



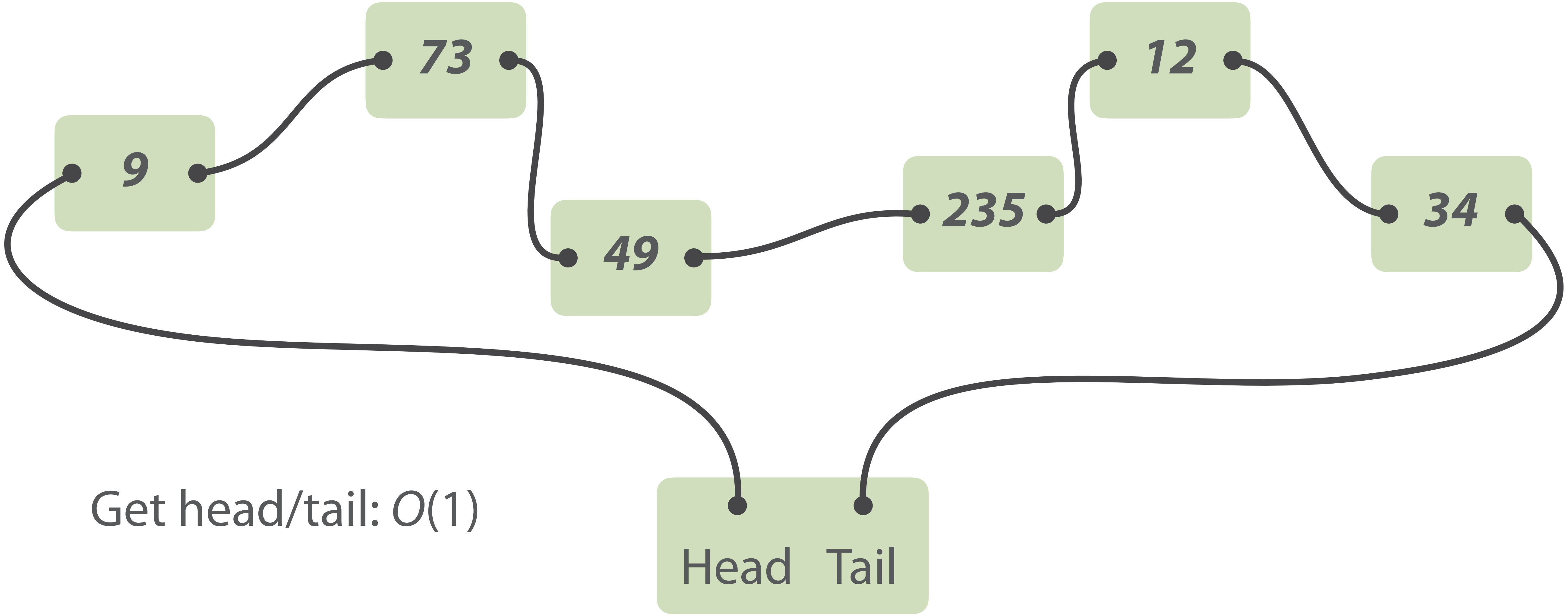
(Doubly) Linked List

Assume *N* elements in chain



(Doubly) Linked List

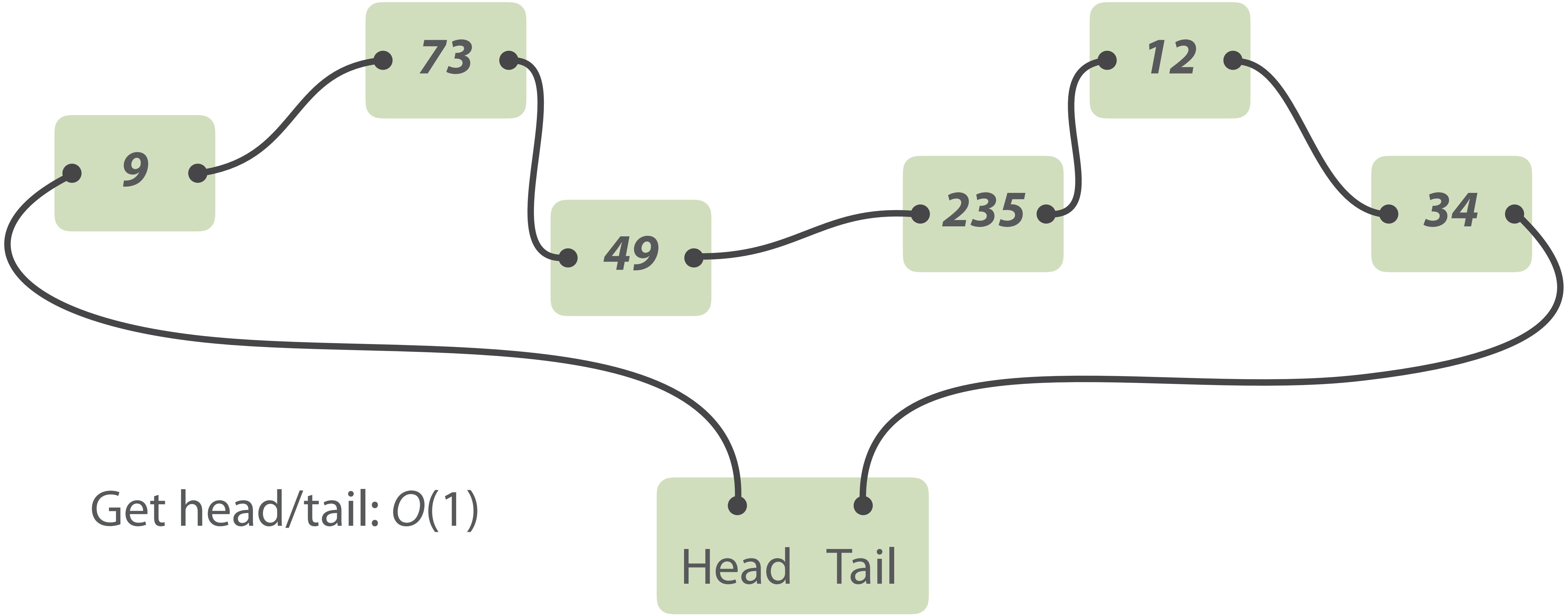
Assume *N* elements in chain



Get head/tail: *O*(1)

(Doubly) Linked List

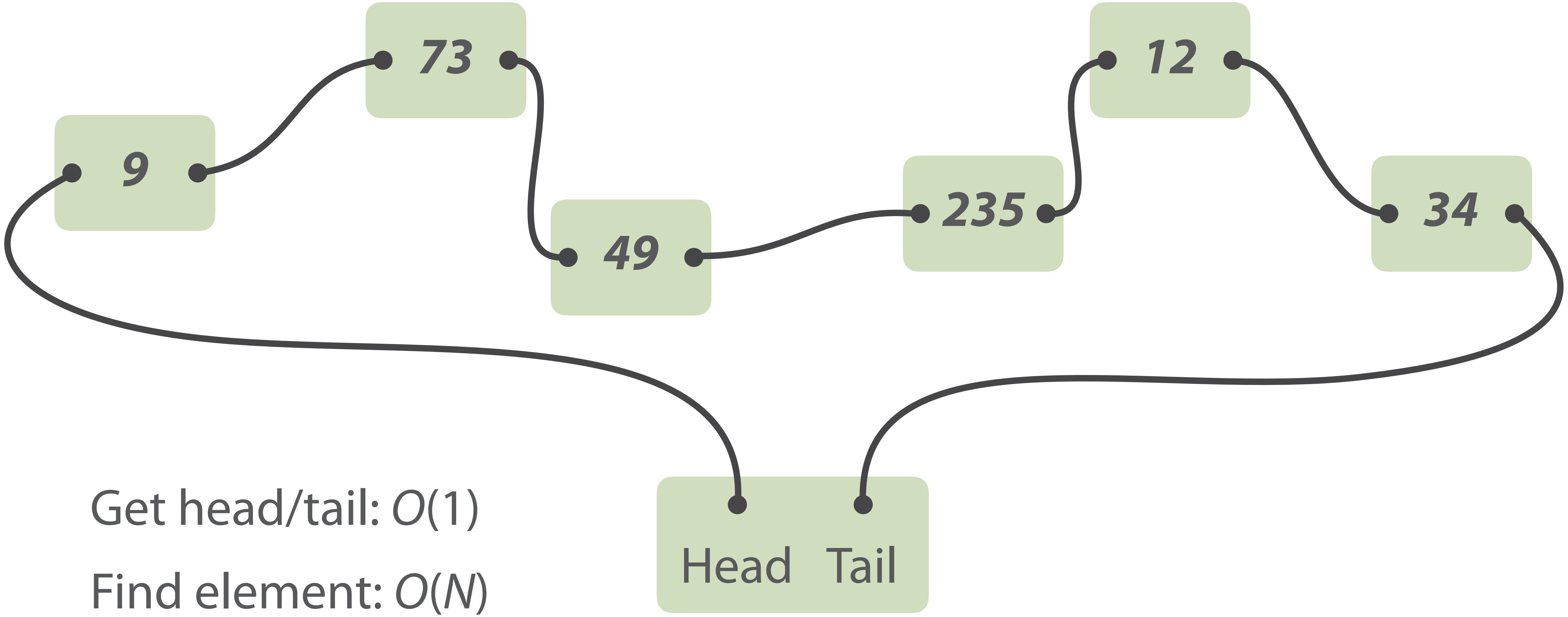
Assume *N* elements in chain



Get head/tail: *O*(1)

(Doubly) Linked List

Assume *N* elements in chain

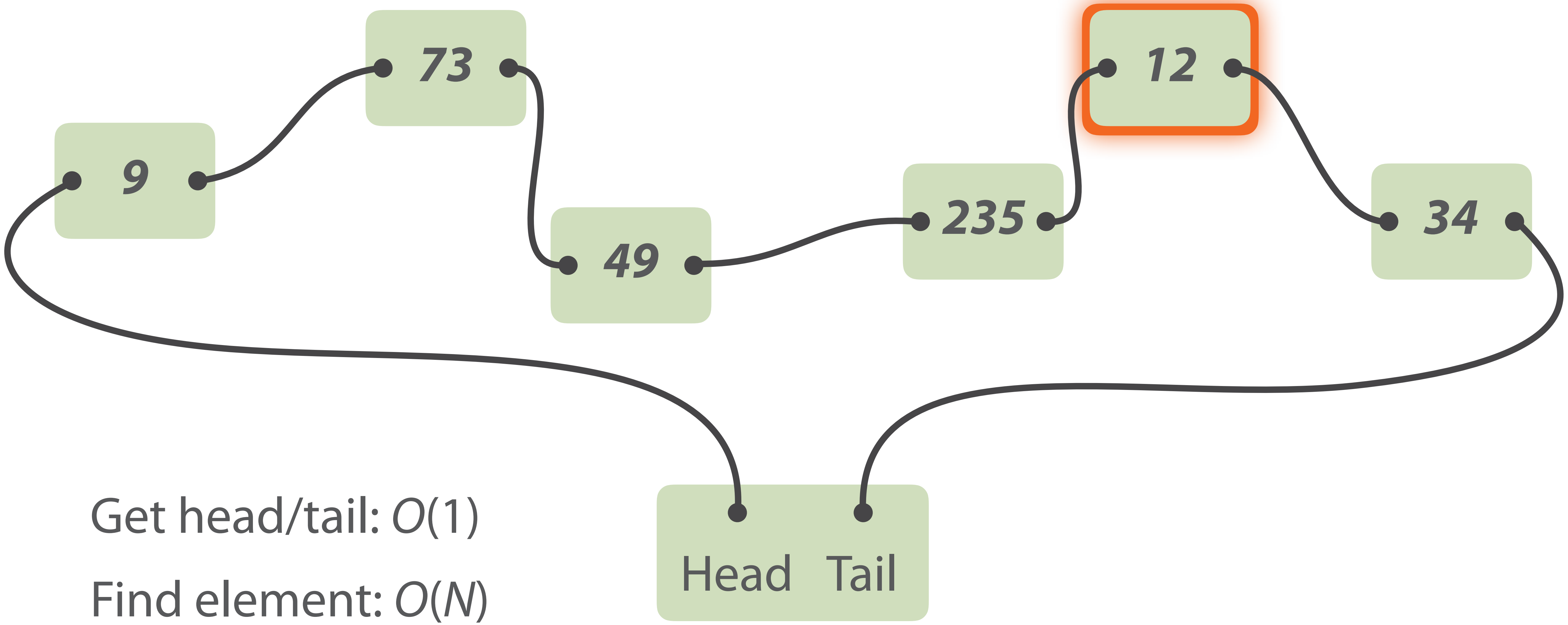


Get head/tail: *O*(1)

Find element: *O*(*N*)

(Doubly) Linked List

Assume *N* elements in chain

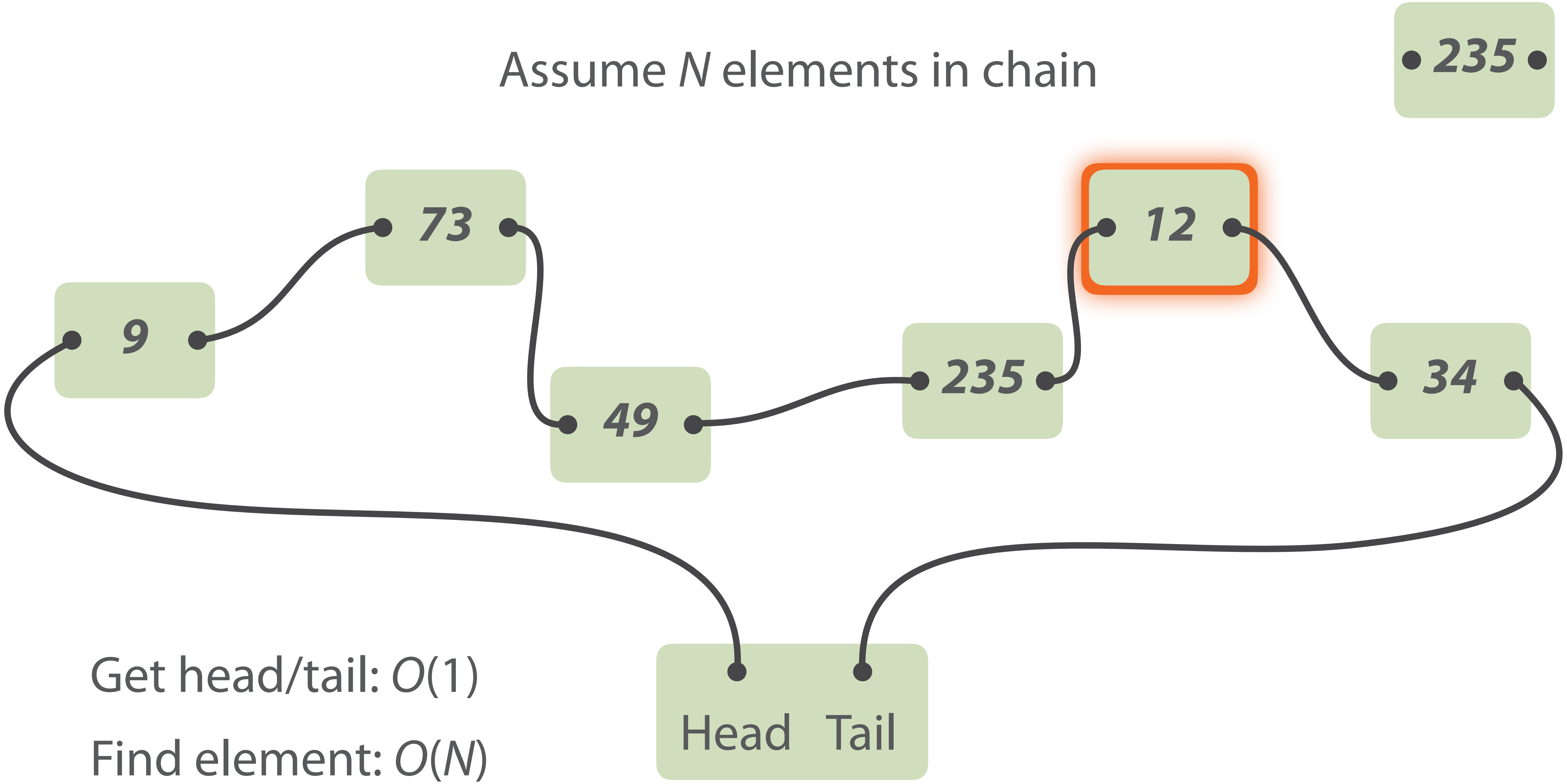


Get head/tail: *O*(1)

Find element: *O*(*N*)

(Doubly) Linked List

Assume *N* elements in chain

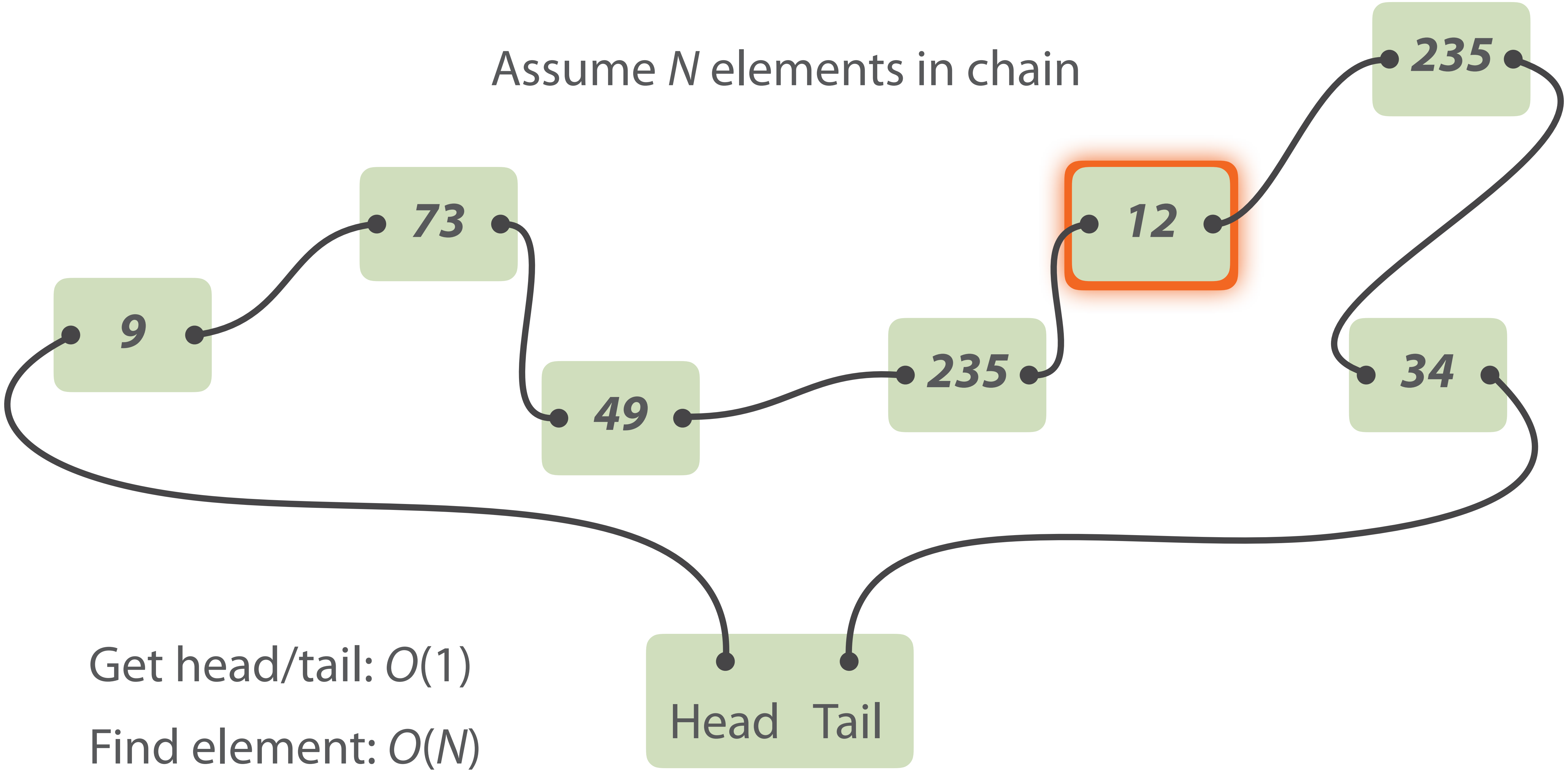


Get head/tail: *O*(1)

Find element: *O*(*N*)

(Doubly) Linked List

Assume *N* elements in chain

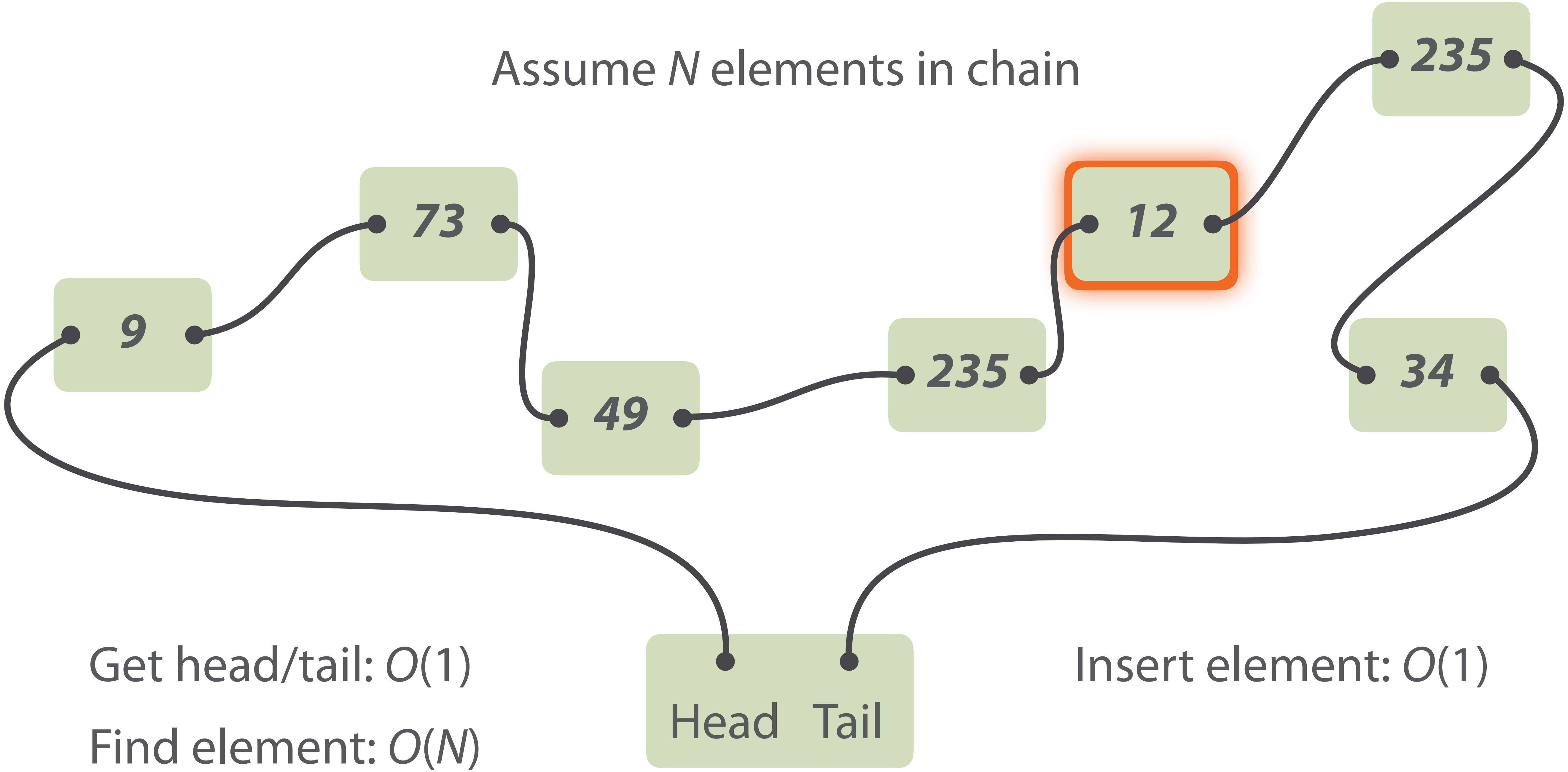


Get head/tail: *O*(1)

Find element: *O*(*N*)

(Doubly) Linked List

Assume *N* elements in chain



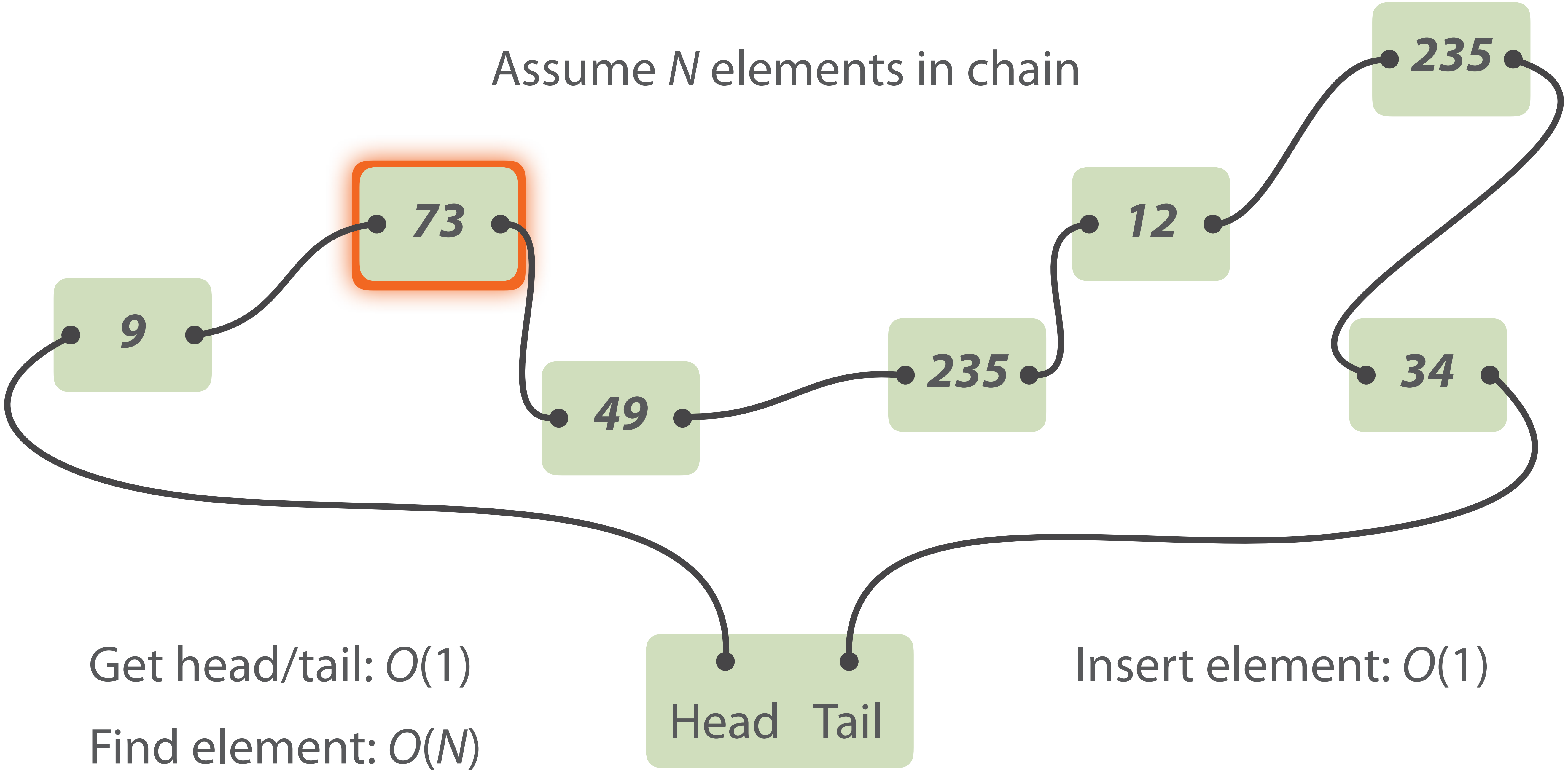
Get head/tail: *O*(1)

Find element: *O*(*N*)

Insert element: *O*(1)

(Doubly) Linked List

Assume *N* elements in chain



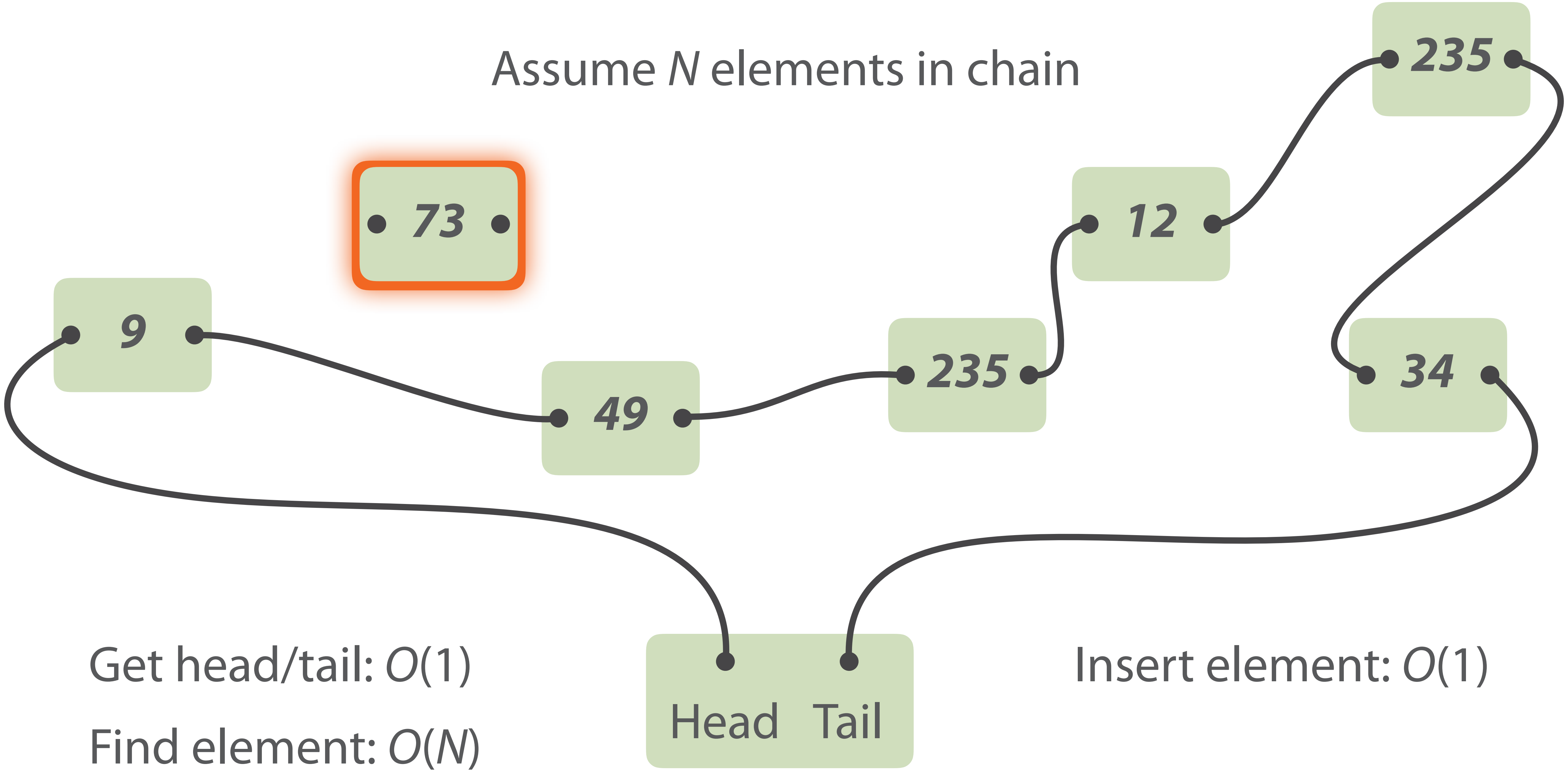
Get head/tail: *O*(1)

Find element: *O*(*N*)

Insert element: *O*(1)

(Doubly) Linked List

Assume *N* elements in chain



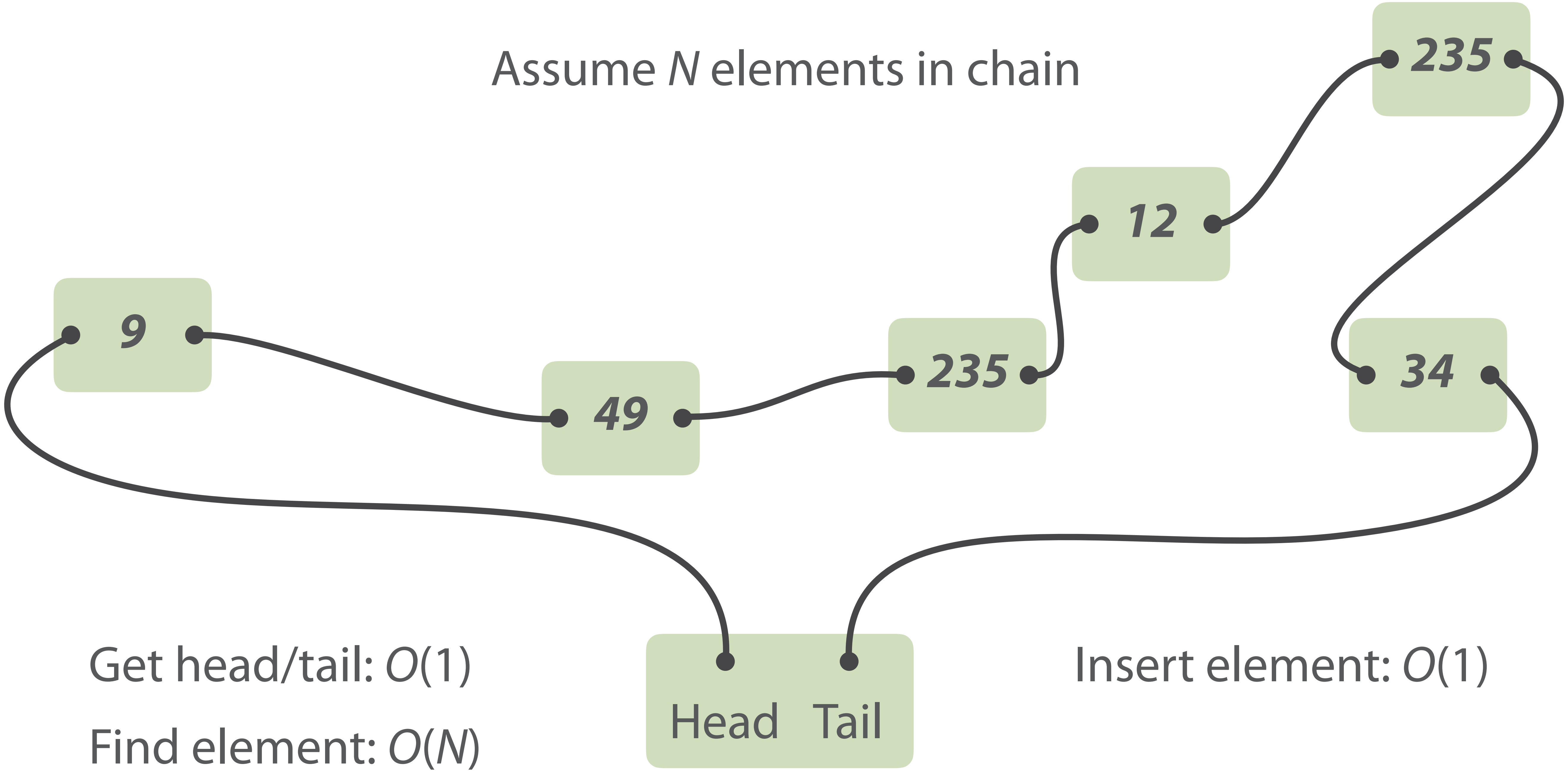
Get head/tail: *O*(1)

Find element: *O*(*N*)

Insert element: *O*(1)

(Doubly) Linked List

Assume *N* elements in chain



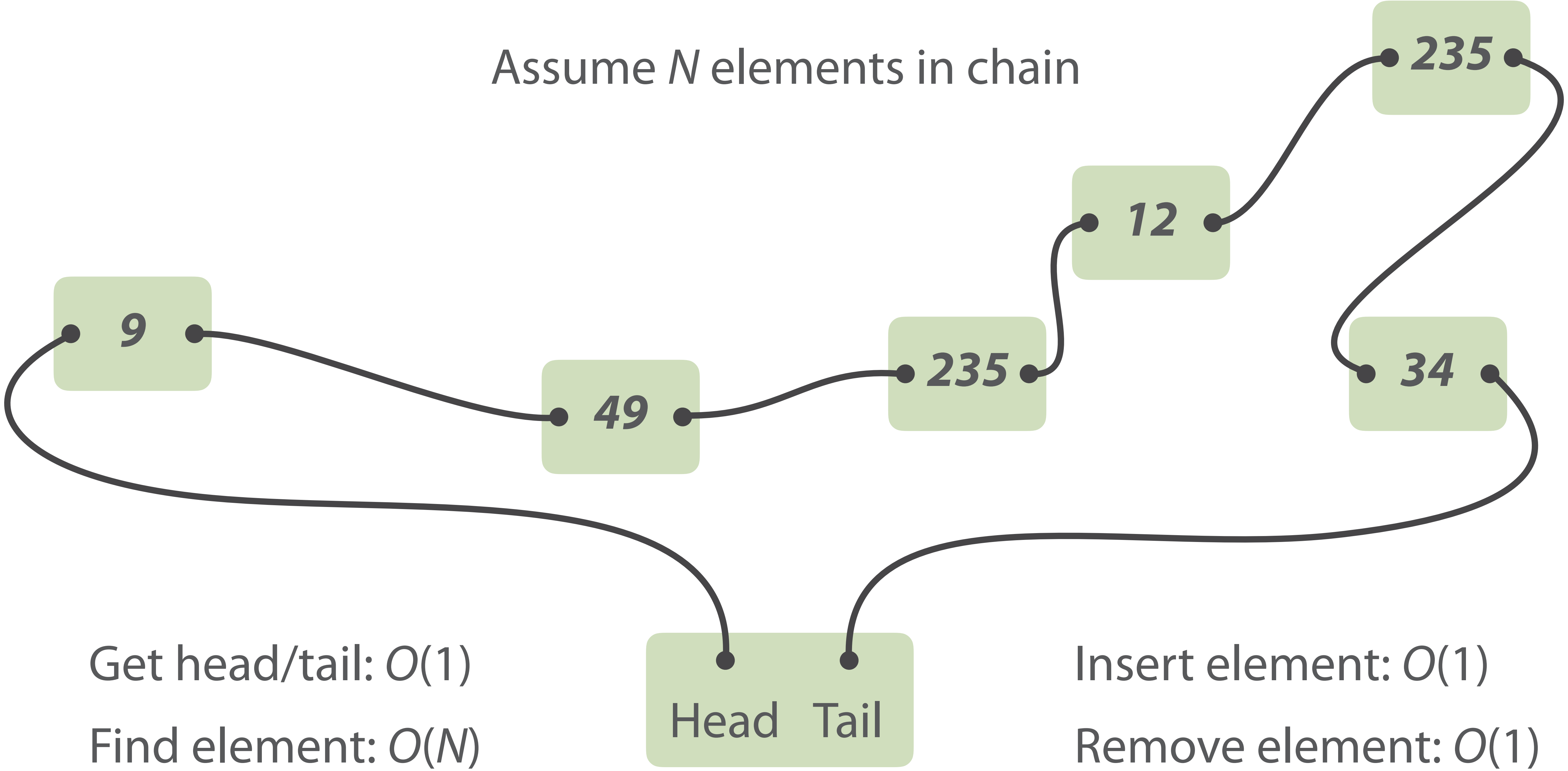
Get head/tail: *O*(1)

Find element: *O*(*N*)

Insert element: *O*(1)

(Doubly) Linked List

Assume *N* elements in chain



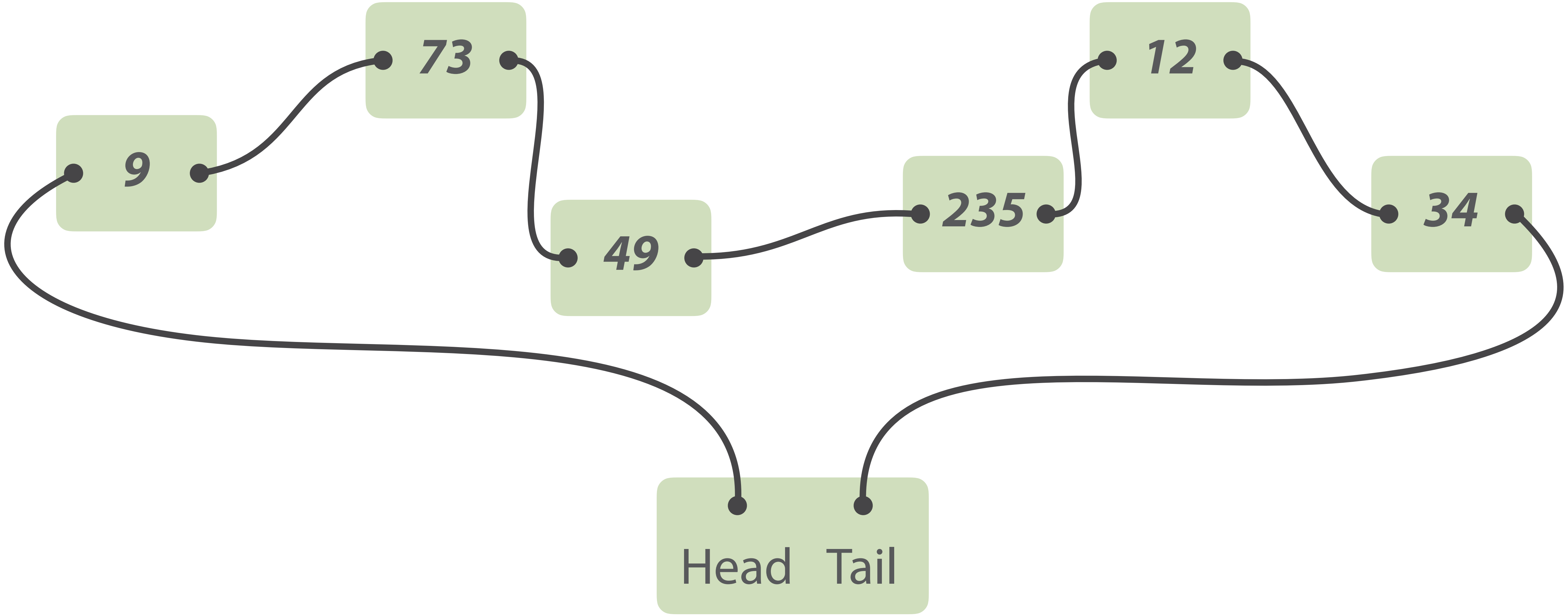
Get head/tail: *O*(1)

Find element: *O*(*N*)

Insert element: *O*(1)

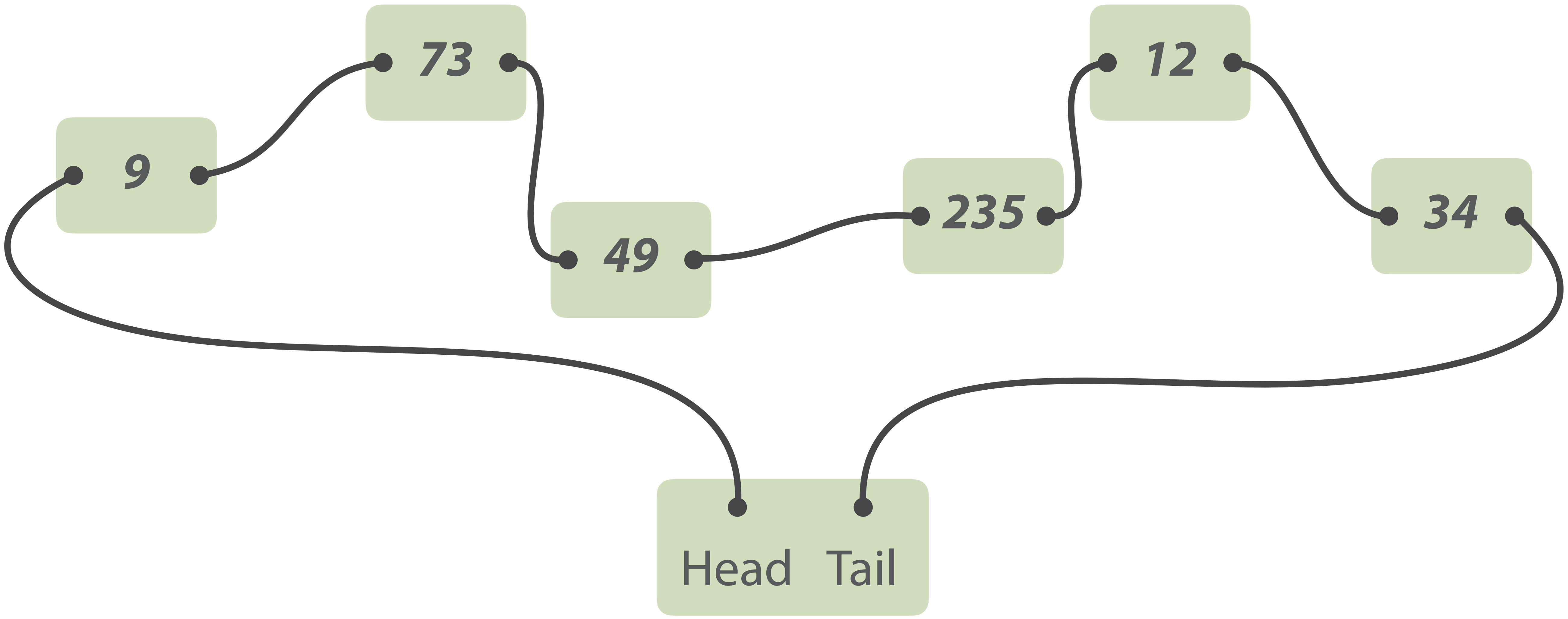
Remove element: *O*(1)

Linked Lists in Practice



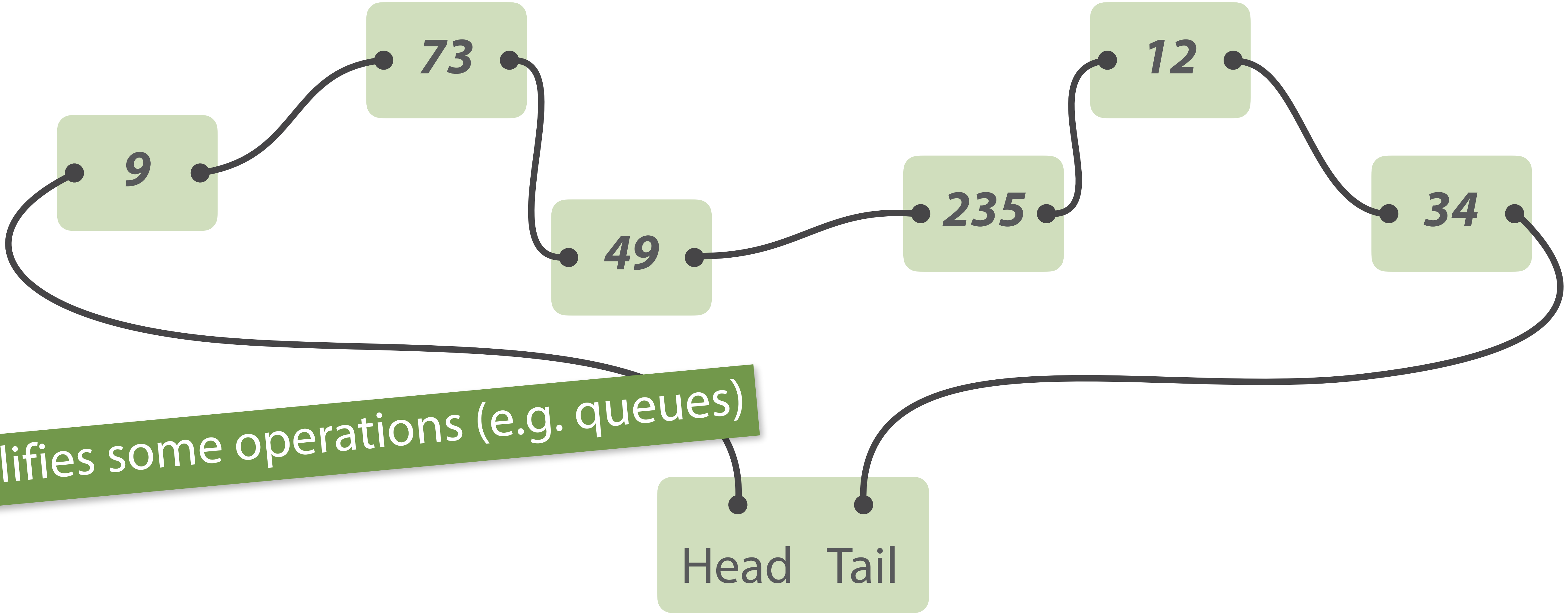
Linked Lists in Practice

Append two lists in $O(1)$ time



Linked Lists in Practice

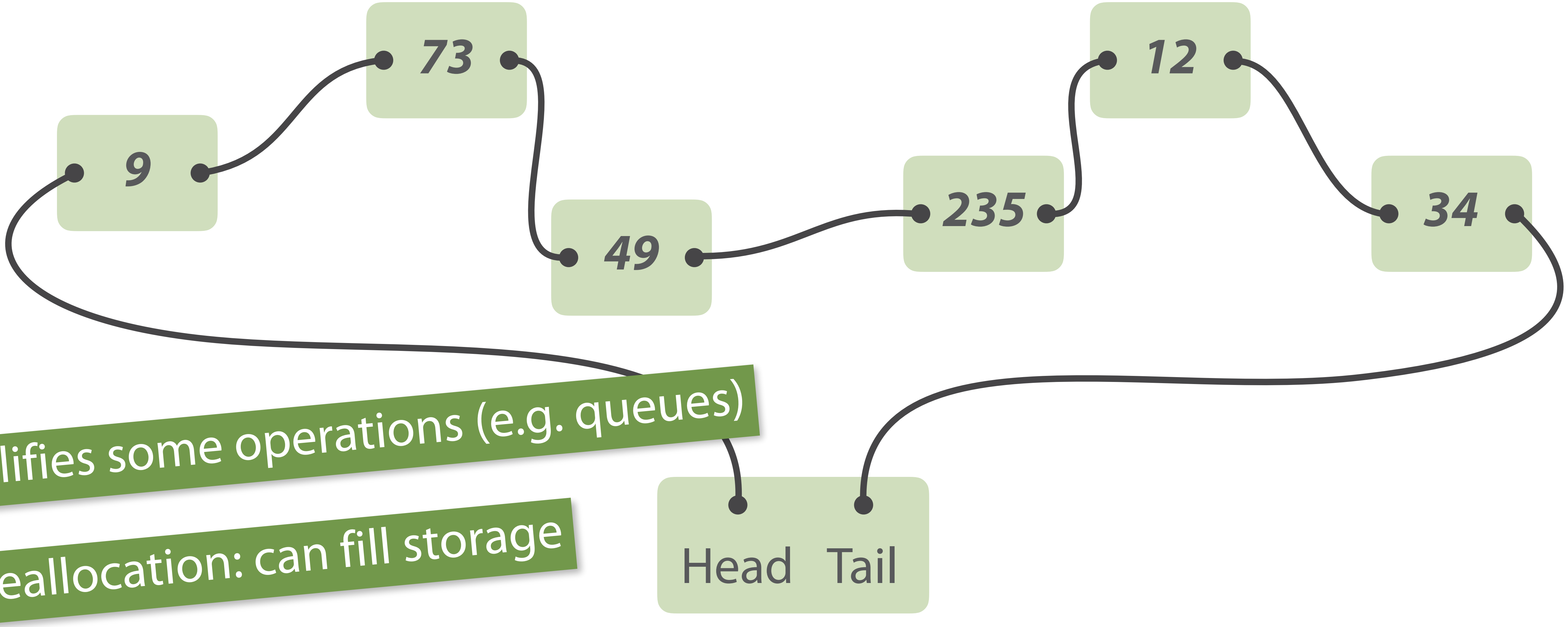
Append two lists in $O(1)$ time



Simplifies some operations (e.g. queues)

Linked Lists in Practice

Append two lists in *O*(1) time



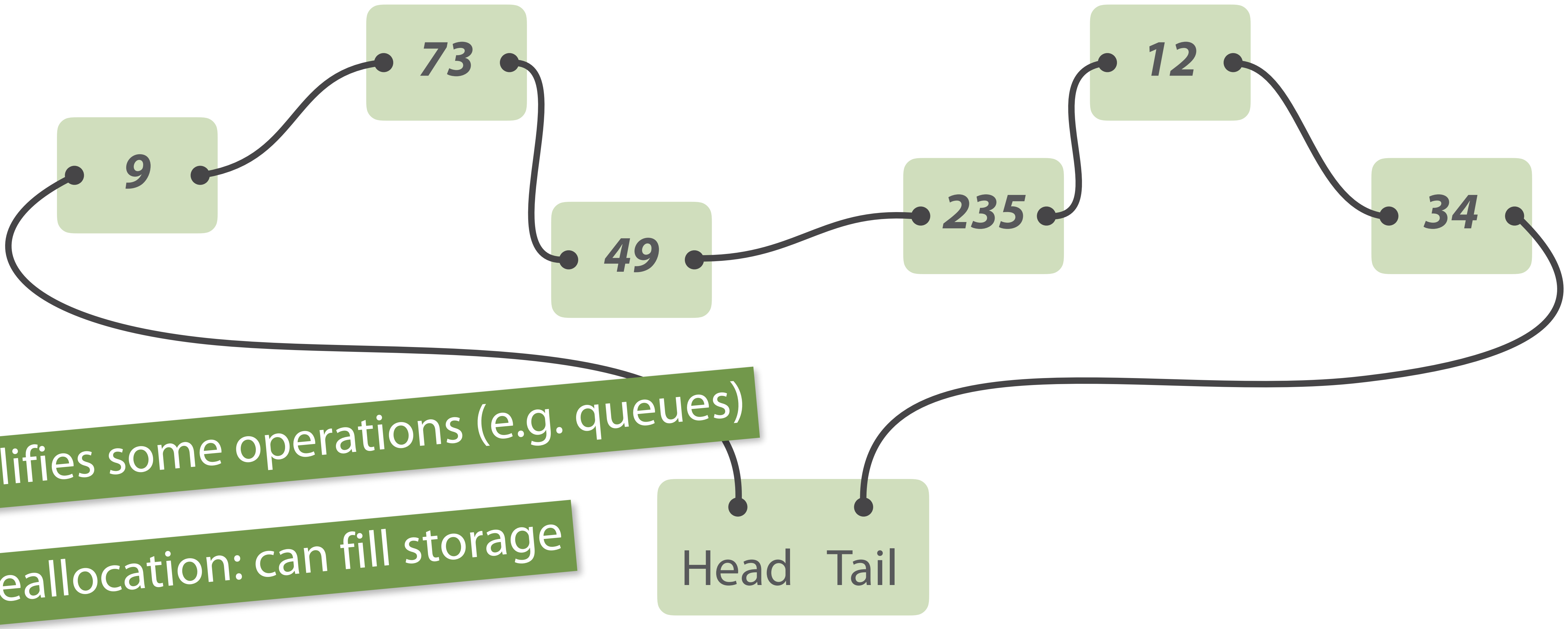
Simplifies some operations (e.g. queues)

No preallocation: can fill storage

Linked Lists in Practice

Append two lists in $O(1)$ time

Increased element access time



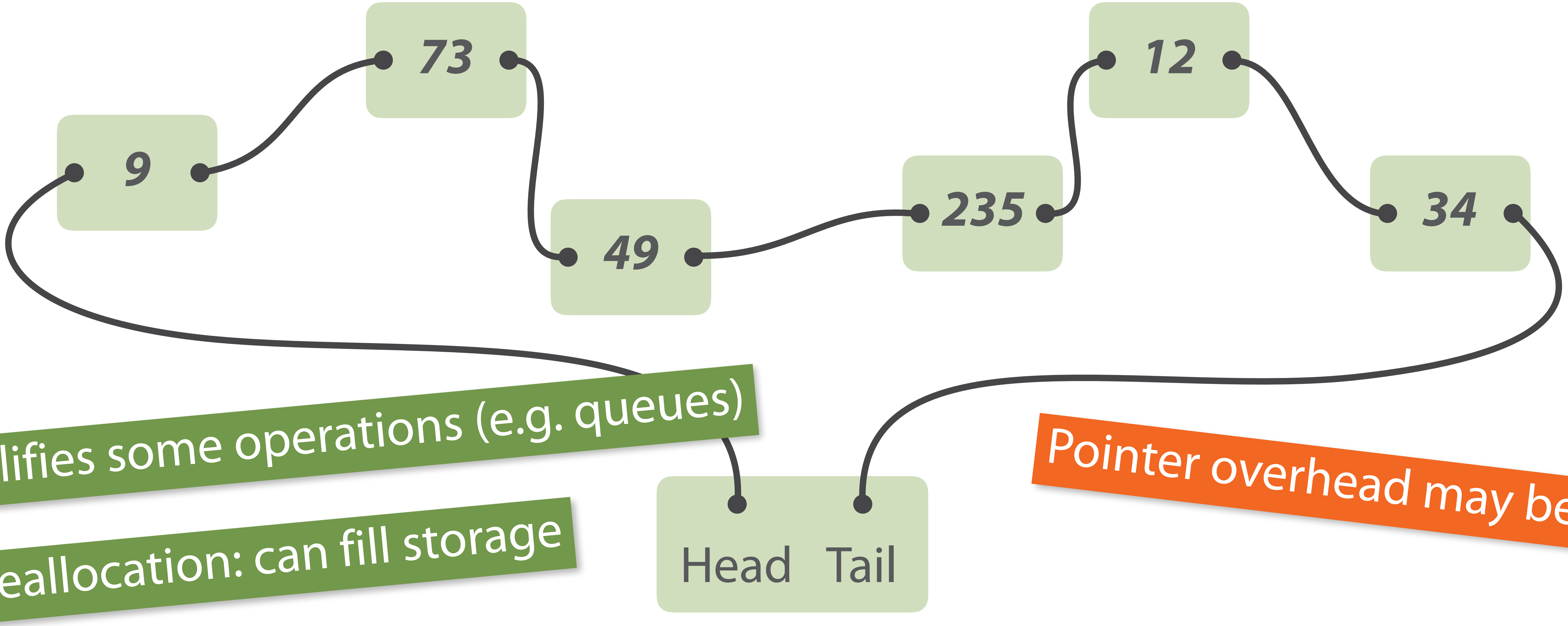
Simplifies some operations (e.g. queues)

No preallocation: can fill storage

Linked Lists in Practice

Append two lists in *O*(1) time

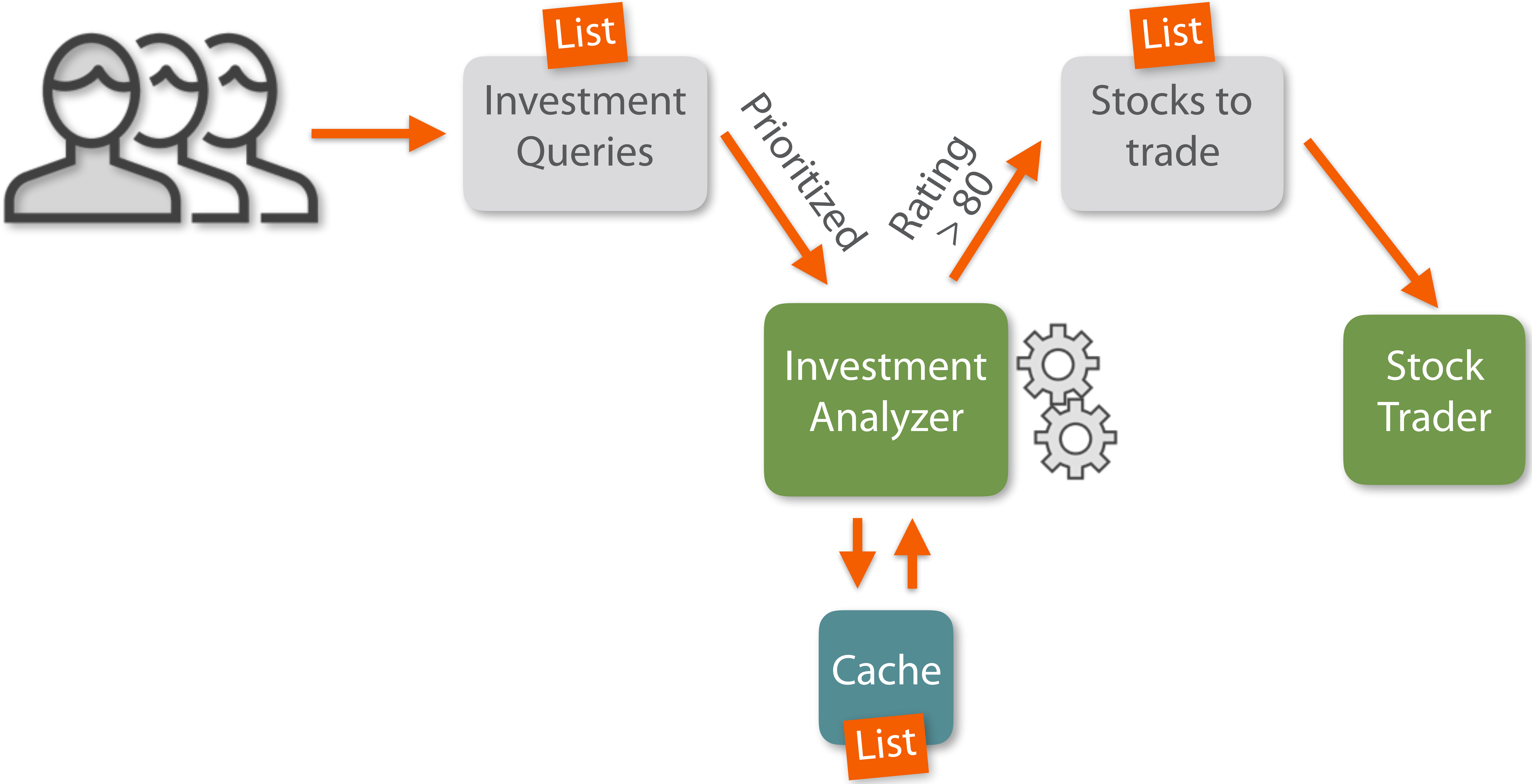
Increased element access time

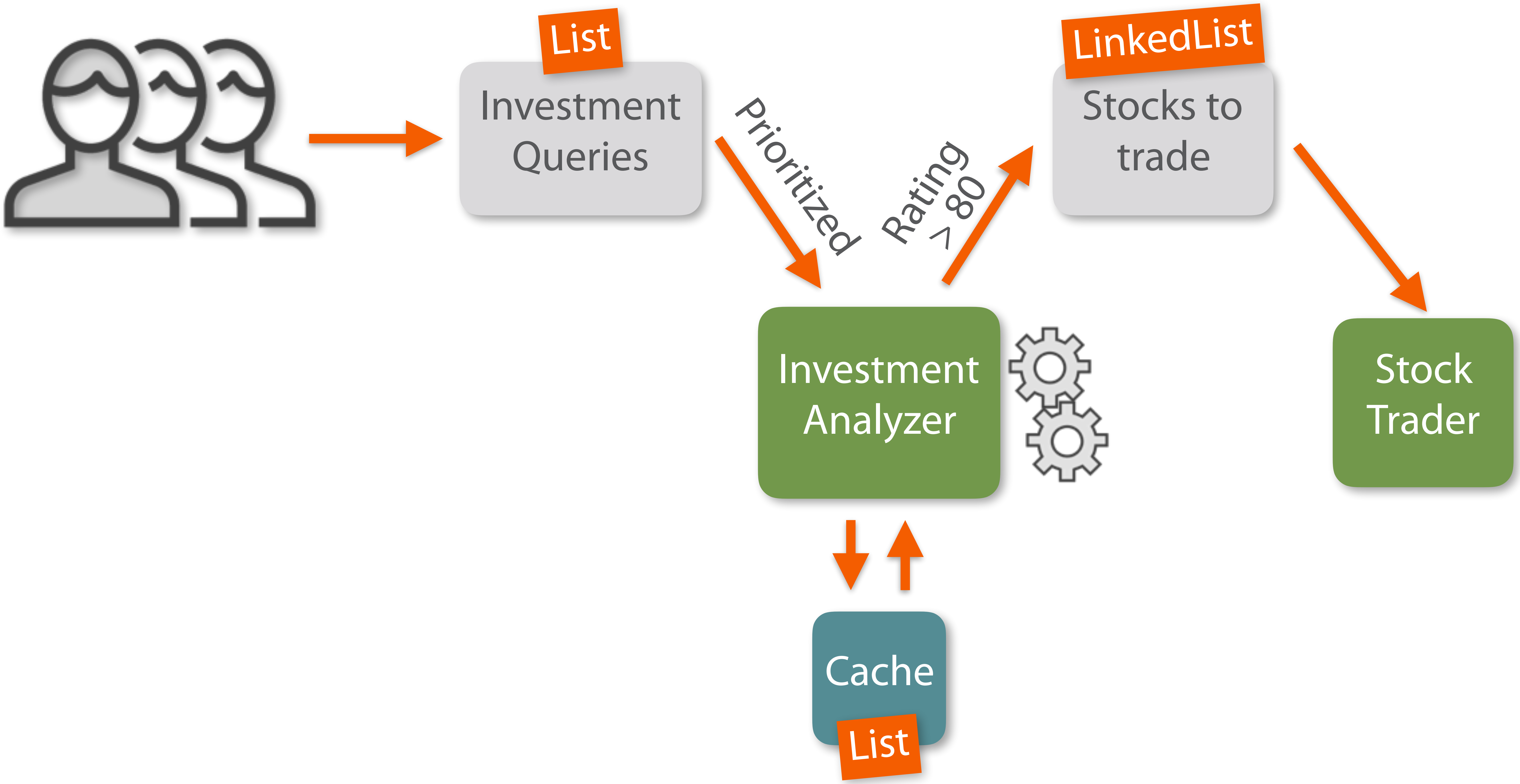


Simplifies some operations (e.g. queues)

No preallocation: can fill storage

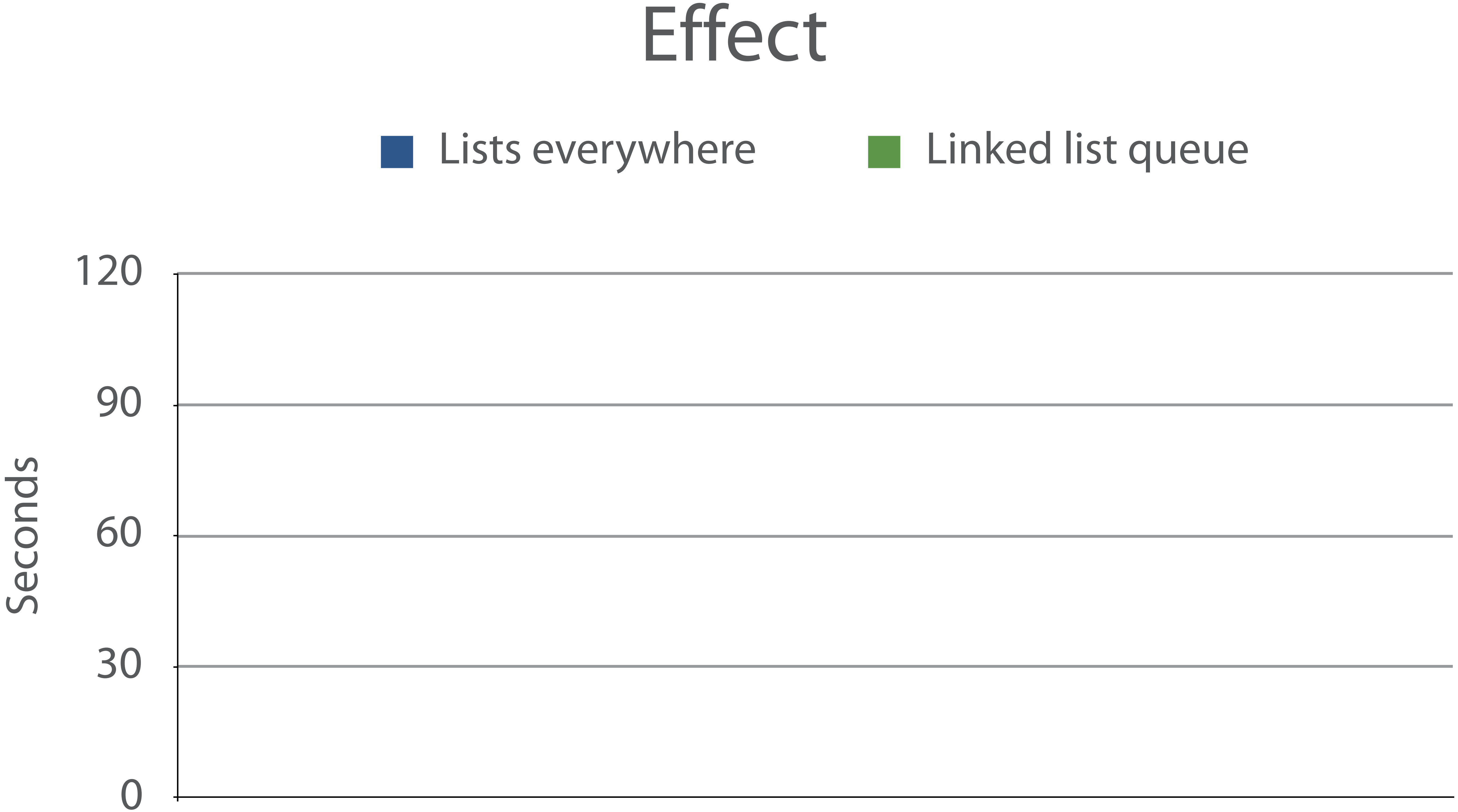
Pointer overhead may be large


List
Investment Queries
Prioritized
Rating > 80
List
Stocks to trade
Investment Analyzer
Stock Trader
Cache
List

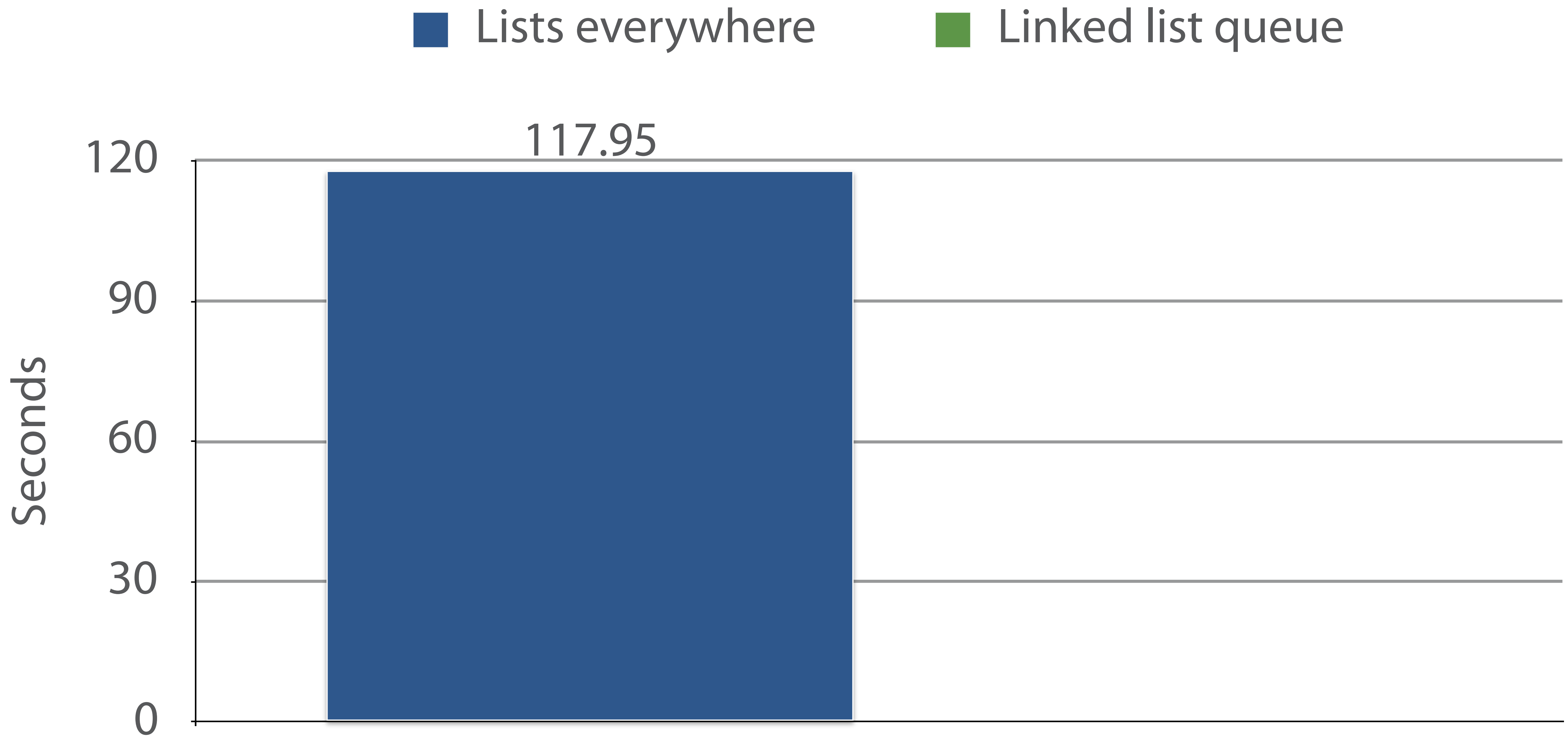

List
Investment Queries
Prioritized
Rating > 80
LinkedList
Stocks to trade
Investment Analyzer
Stock Trader
Cache
List

Investment Analyzer

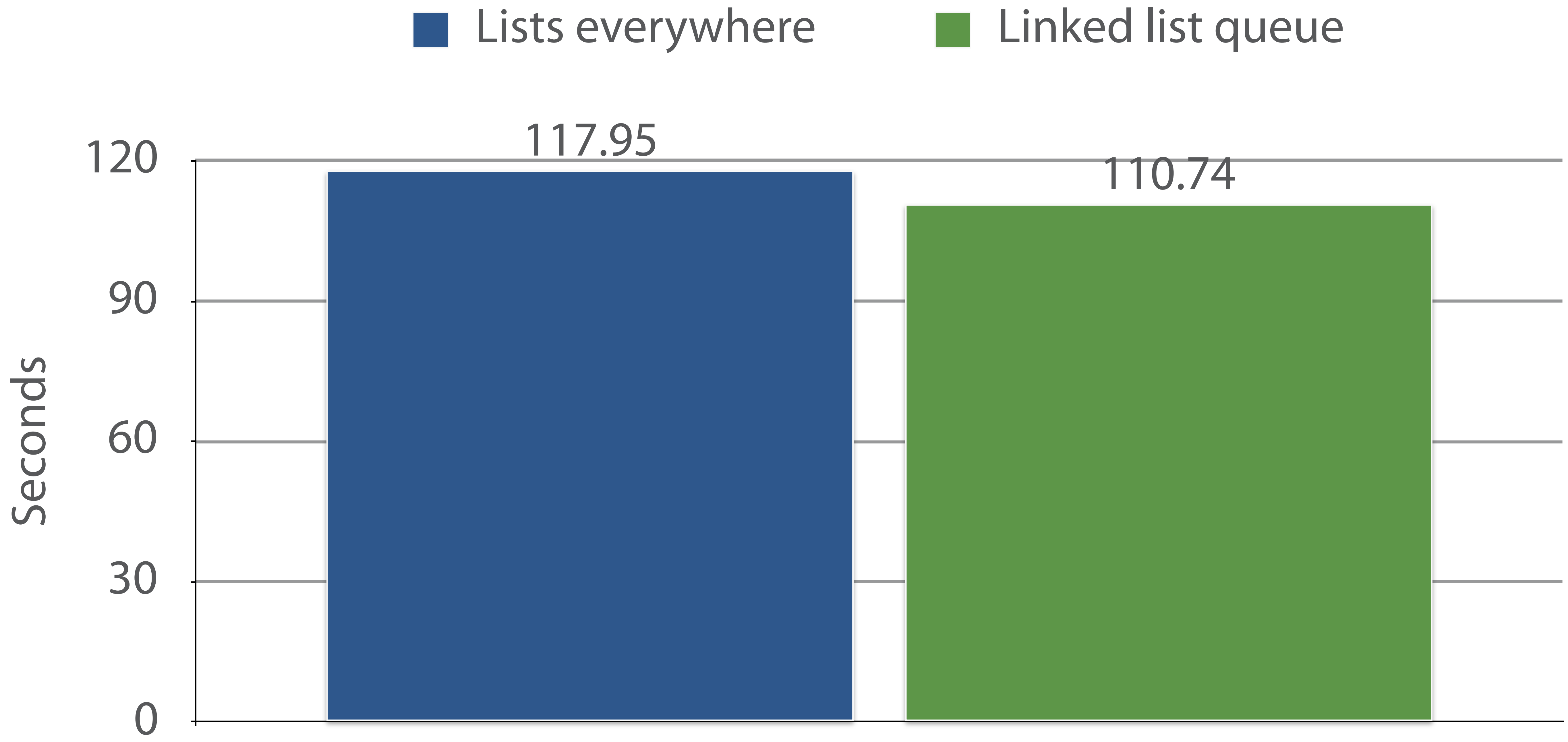
Using a LinkedList as queue


Effect
🟦 Lists everywhere
🟩 Linked list queue
Seconds
120
90
60
30
0

Effect


Lists everywhere
Linked list queue
117.95
Seconds
120
90
60
30
0

Effect


■ Lists everywhere
■ Linked list queue
117.95
110.74
Seconds
120
90
60
30
0

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

Data Structures

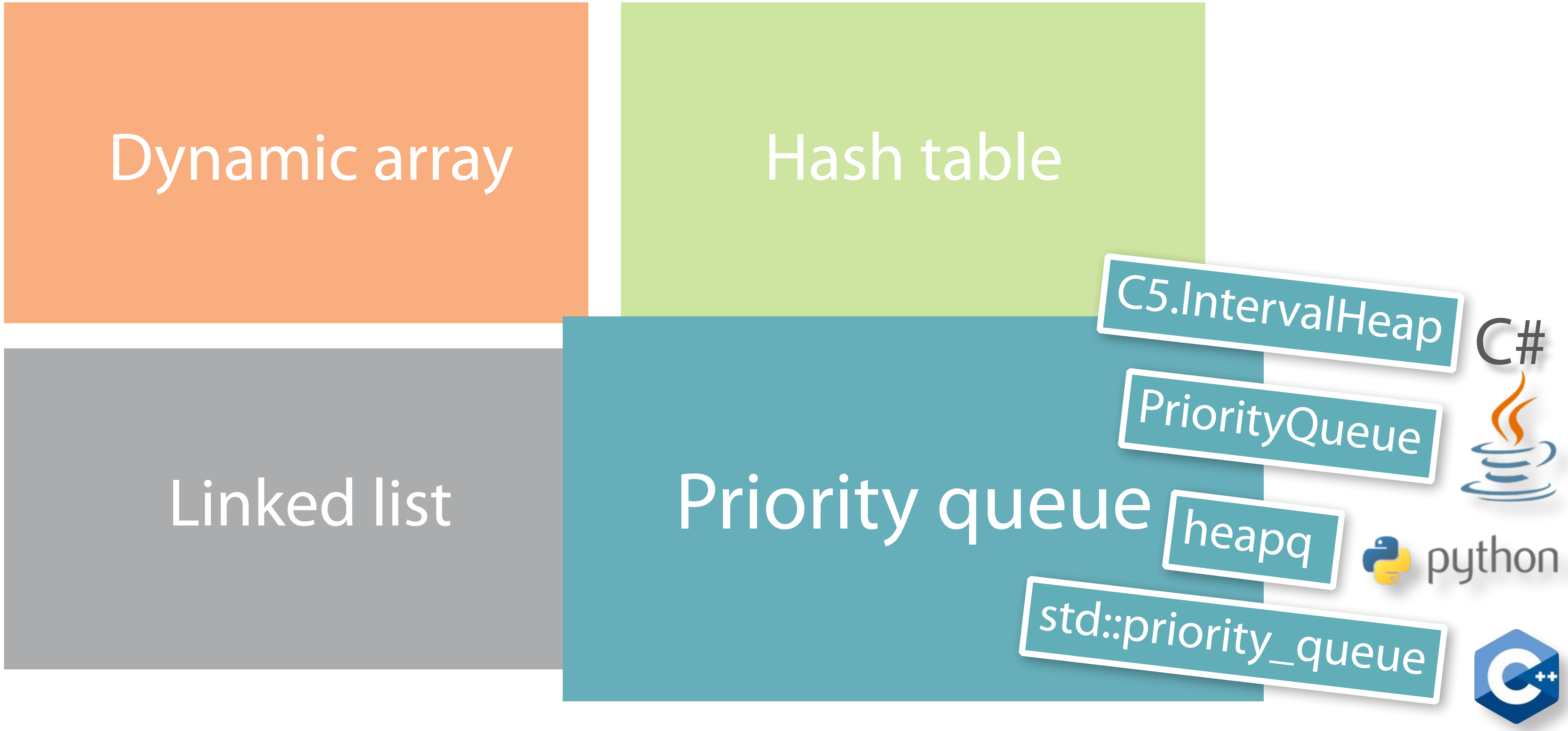
Dynamic array

Hash table

Linked list

Priority queue

Data Structures



Dynamic array

Hash table

Linked list

Priority queue

C5.IntervalHeap

C#

PriorityQueue

heapq

python

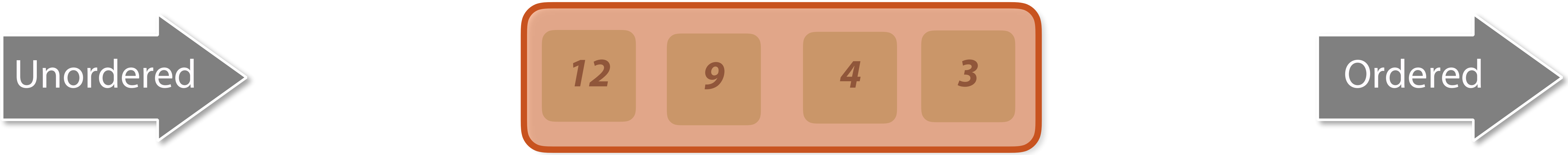
std::priority_queue

(Double-ended) Priority Queues

Unordered

Ordered

(Double-ended) Priority Queues

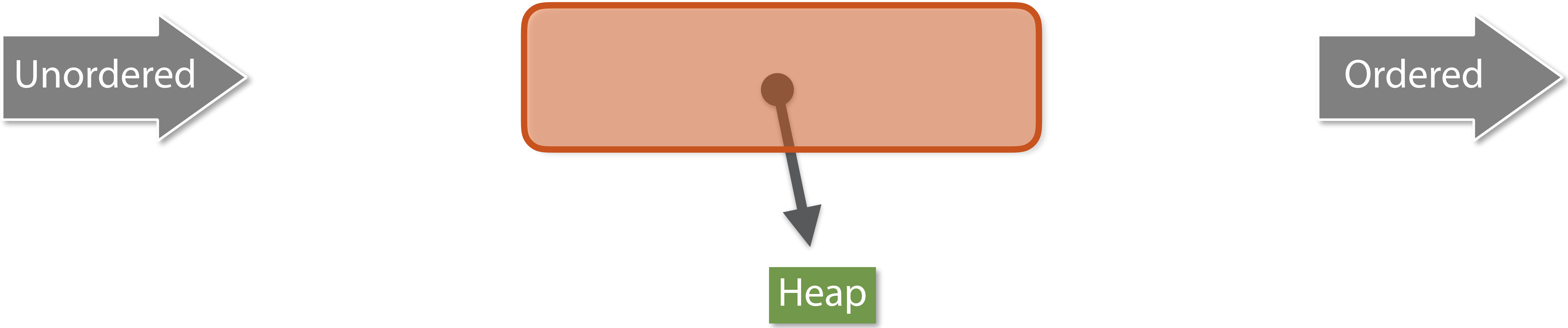


(Double-ended) Priority Queues

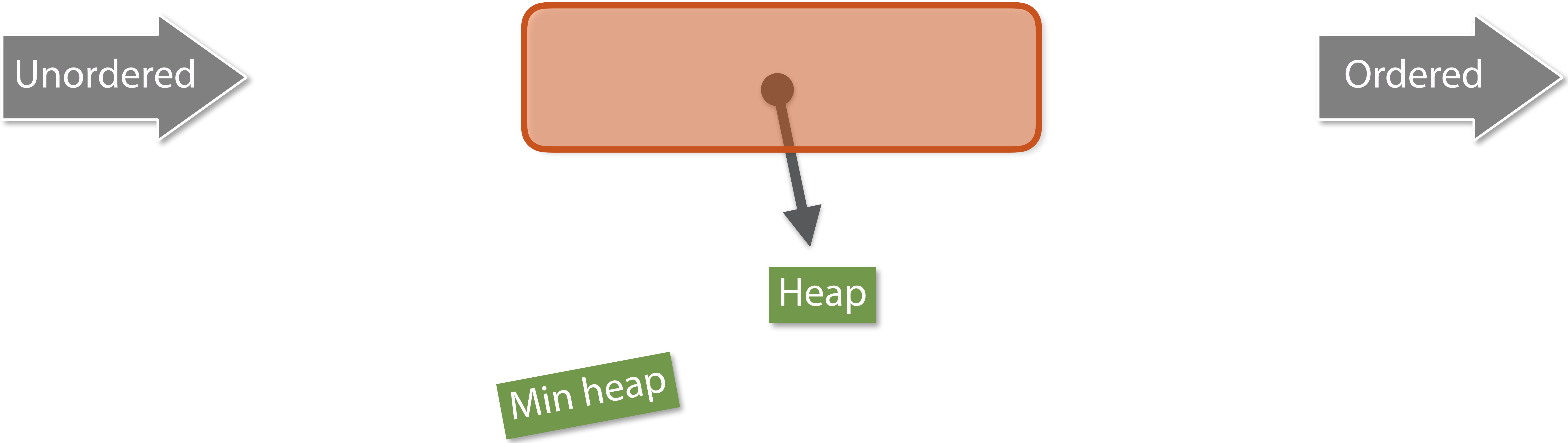
Unordered

Ordered

(Double-ended) Priority Queues



(Double-ended) Priority Queues



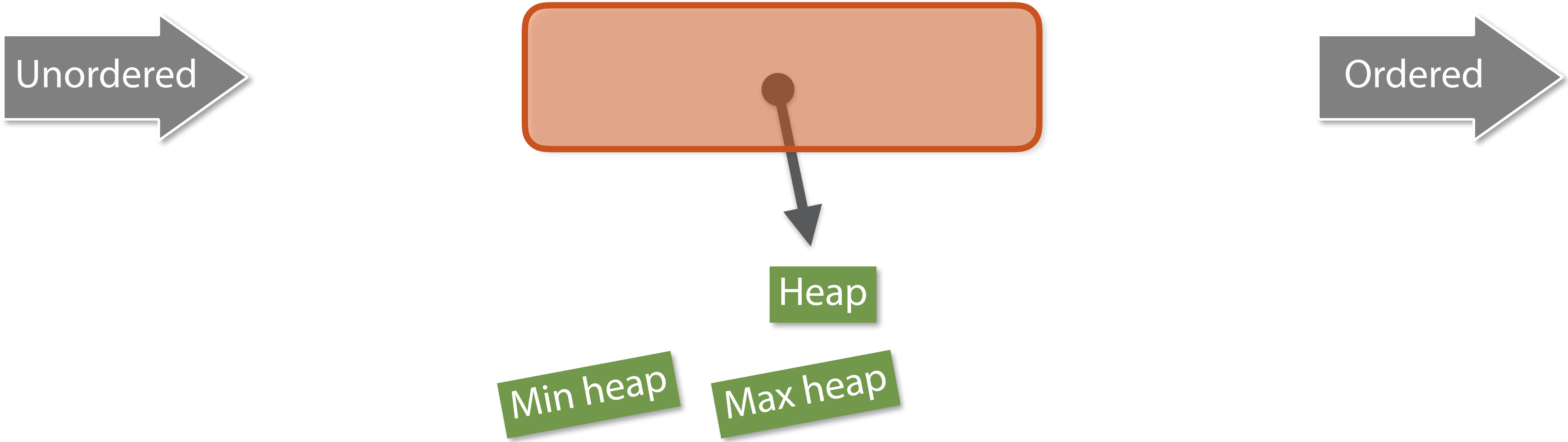
Unordered

Ordered

Heap

Min heap

(Double-ended) Priority Queues



Unordered

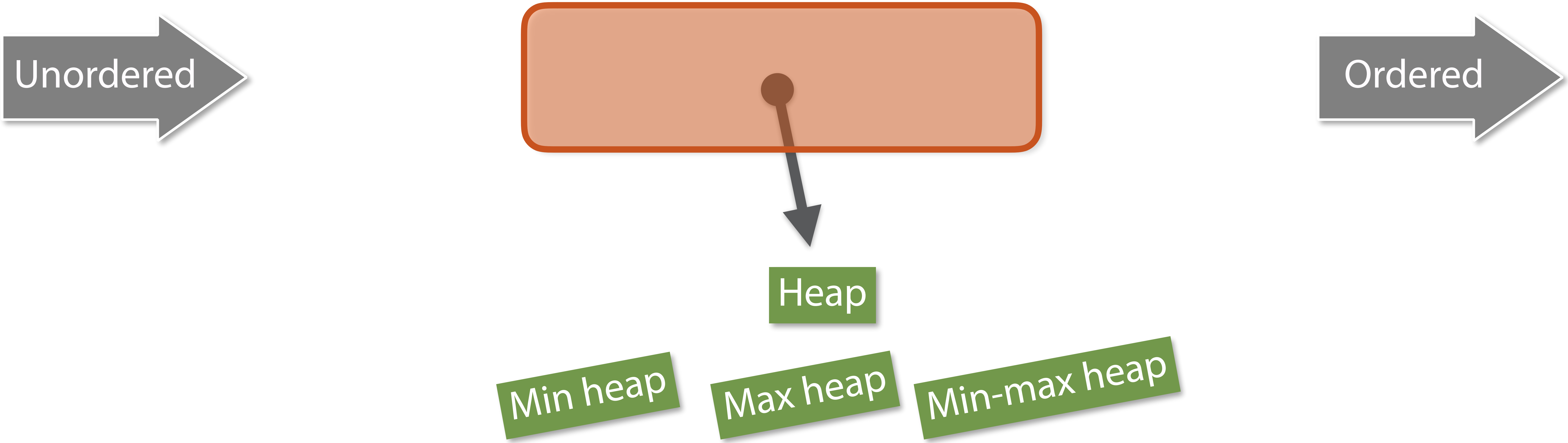
Ordered

Heap

Min heap

Max heap

(Double-ended) Priority Queues



Unordered

Ordered

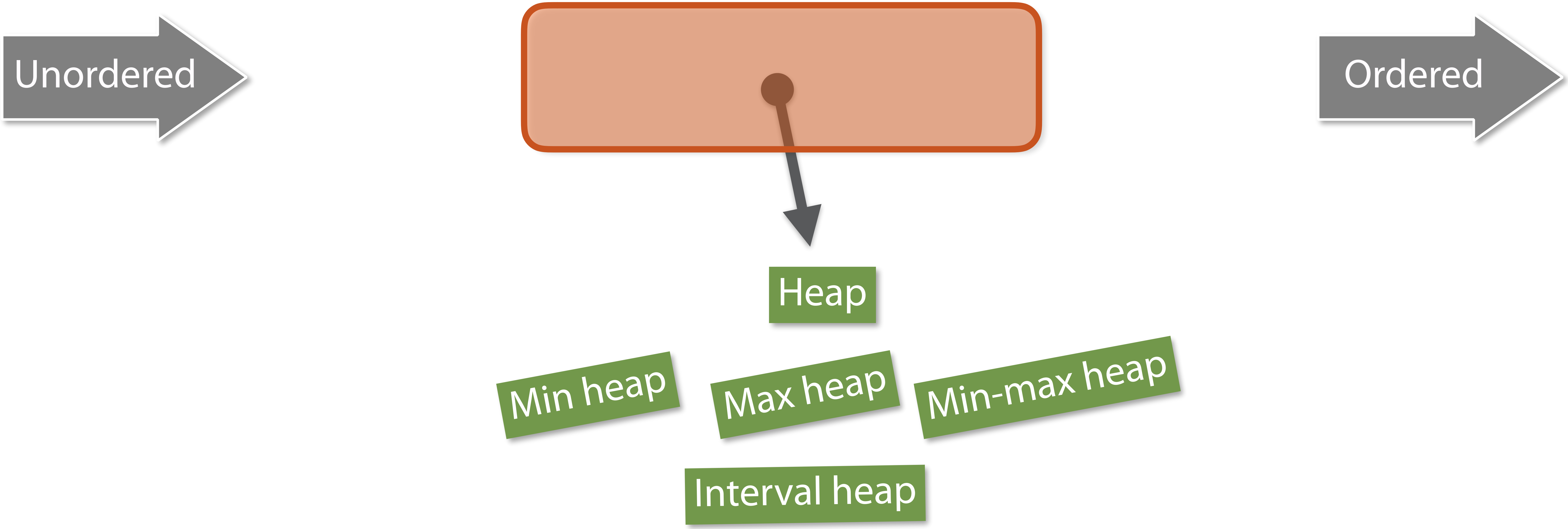
Heap

Min heap

Max heap

Min-max heap

(Double-ended) Priority Queues



Unordered

Ordered

Heap

Min heap

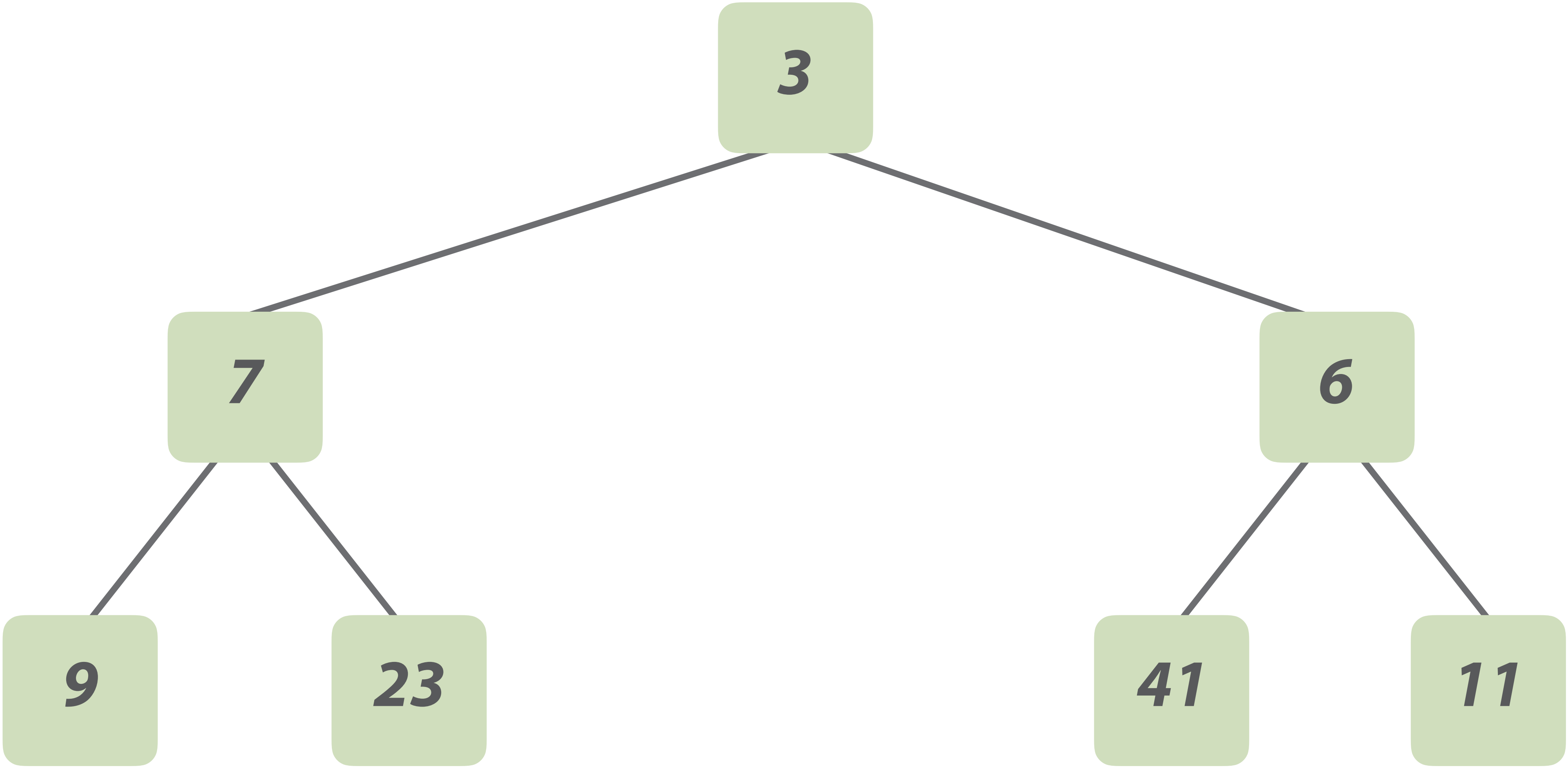
Max heap

Min-max heap

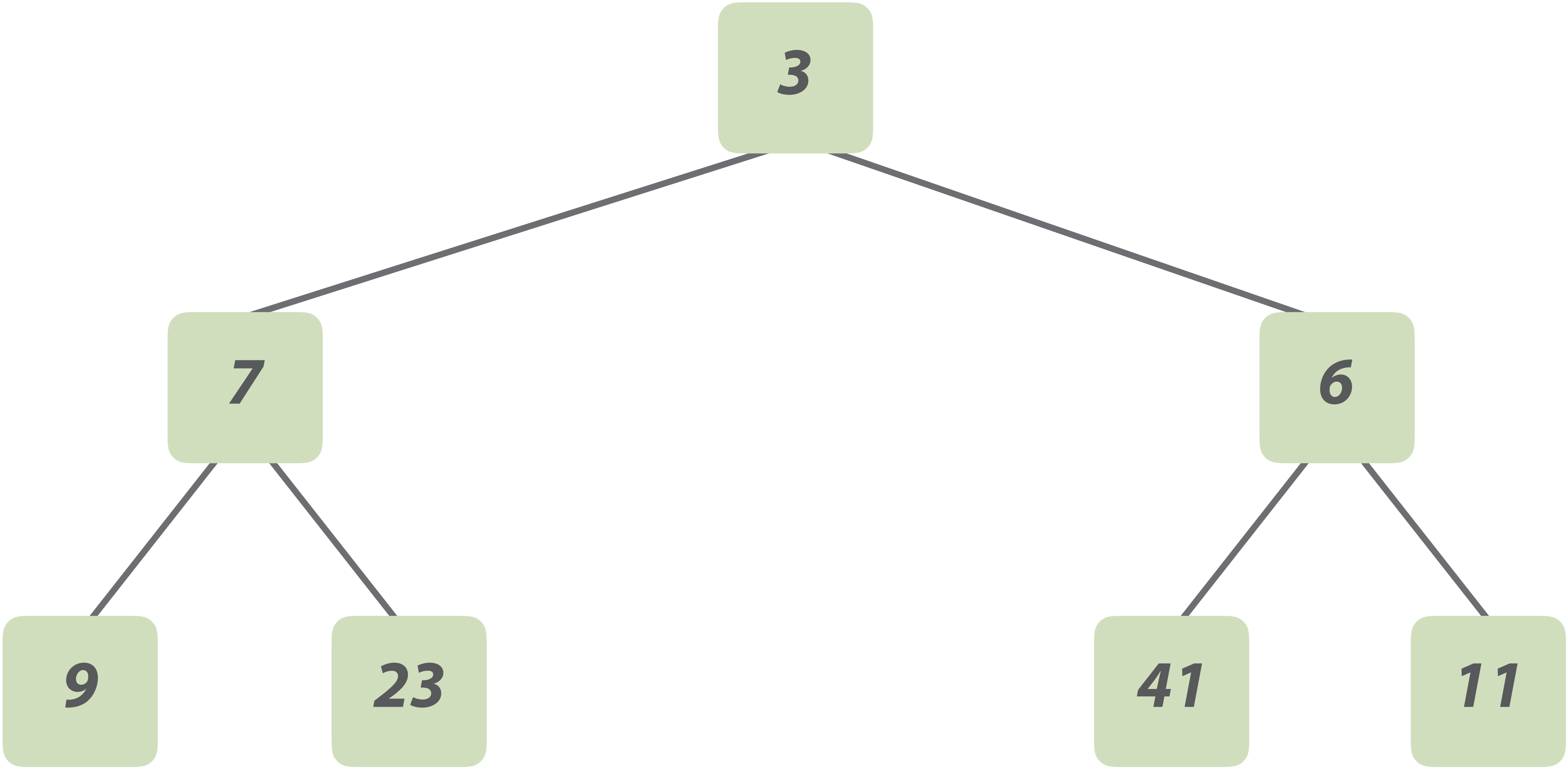
Interval heap

Min-heaps

Min-heaps

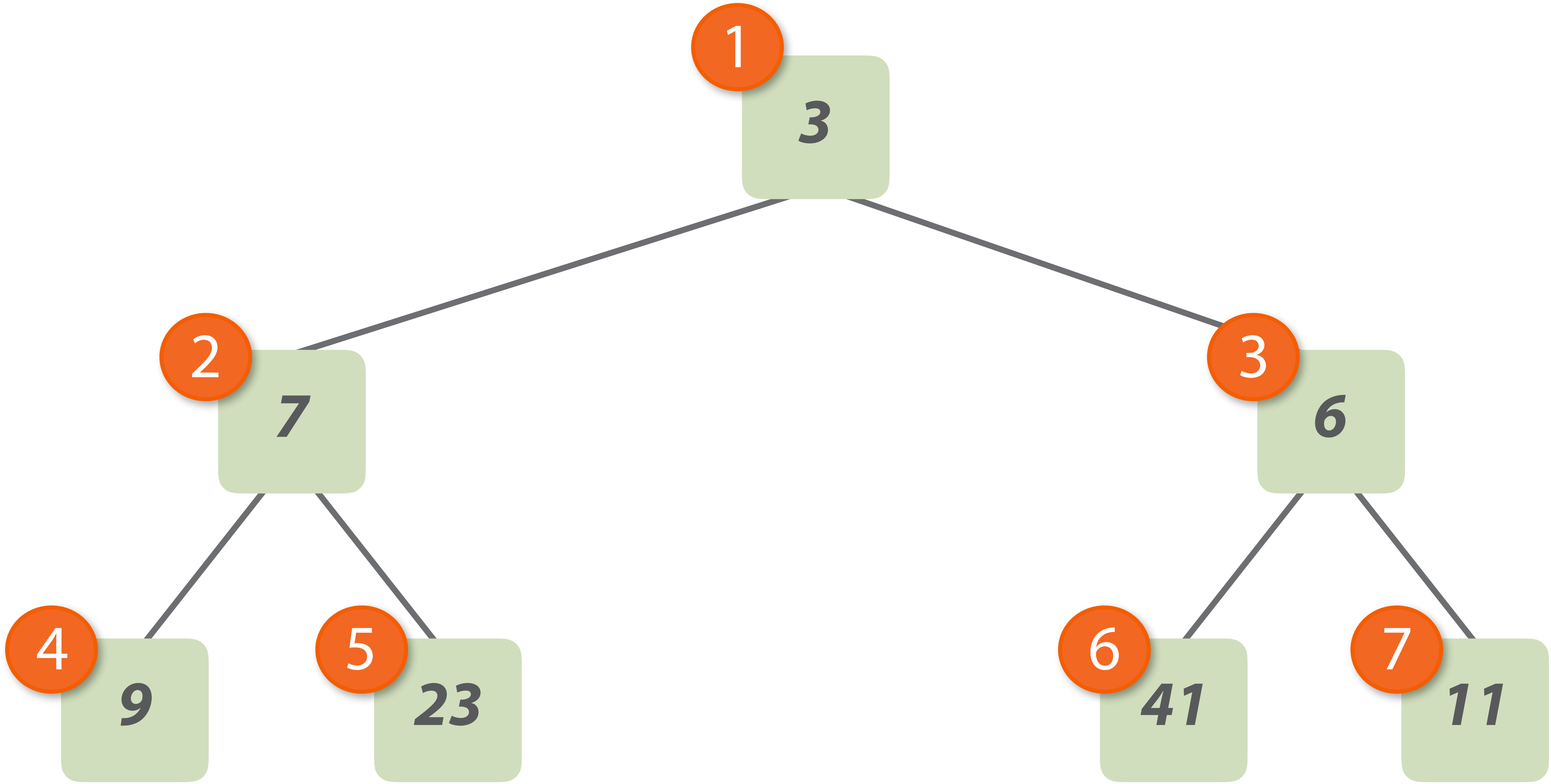


Min-heaps



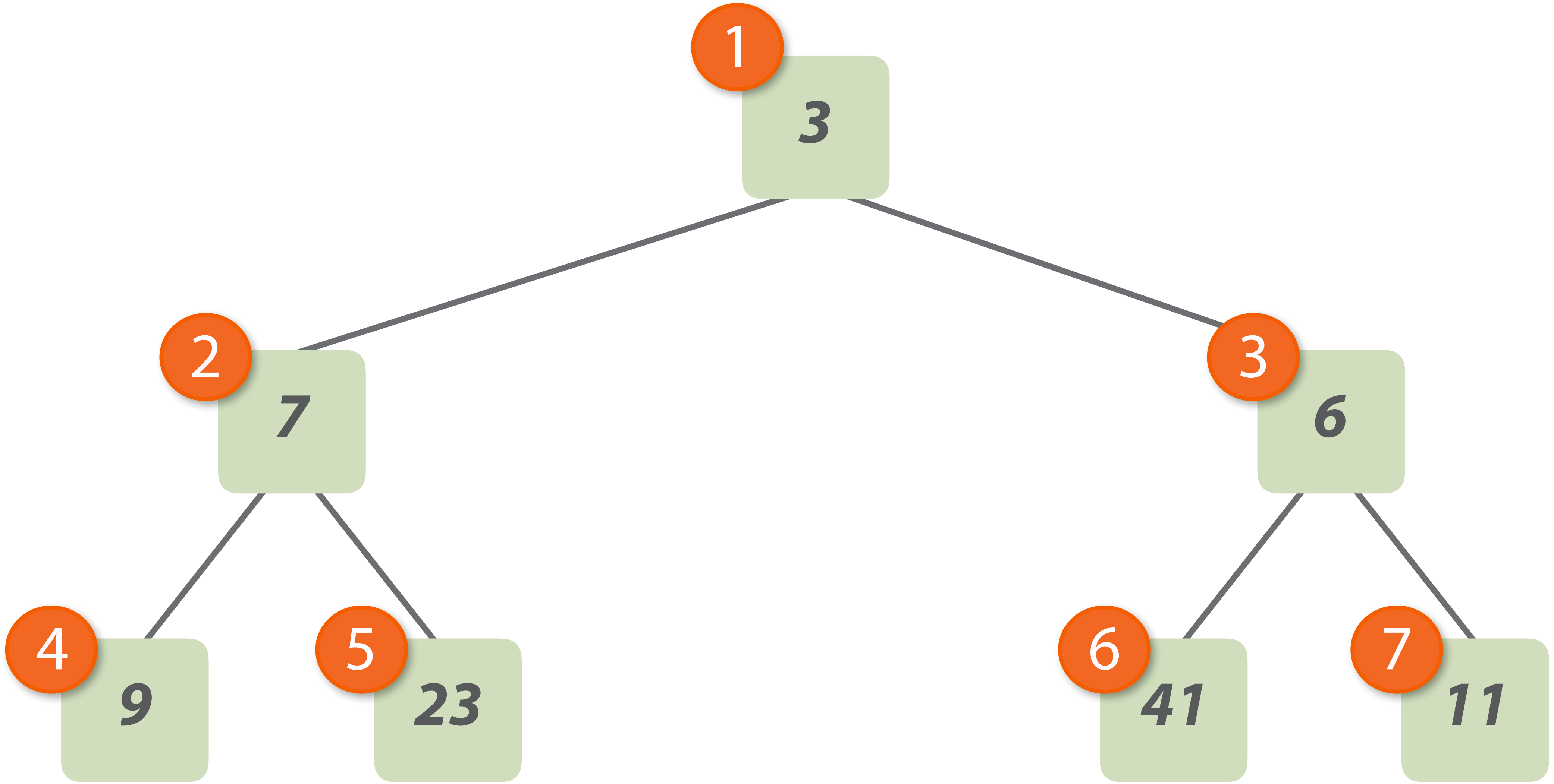
Rule: each element is *smaller* than its children

Min-heaps



Rule: each element is *smaller* than its children

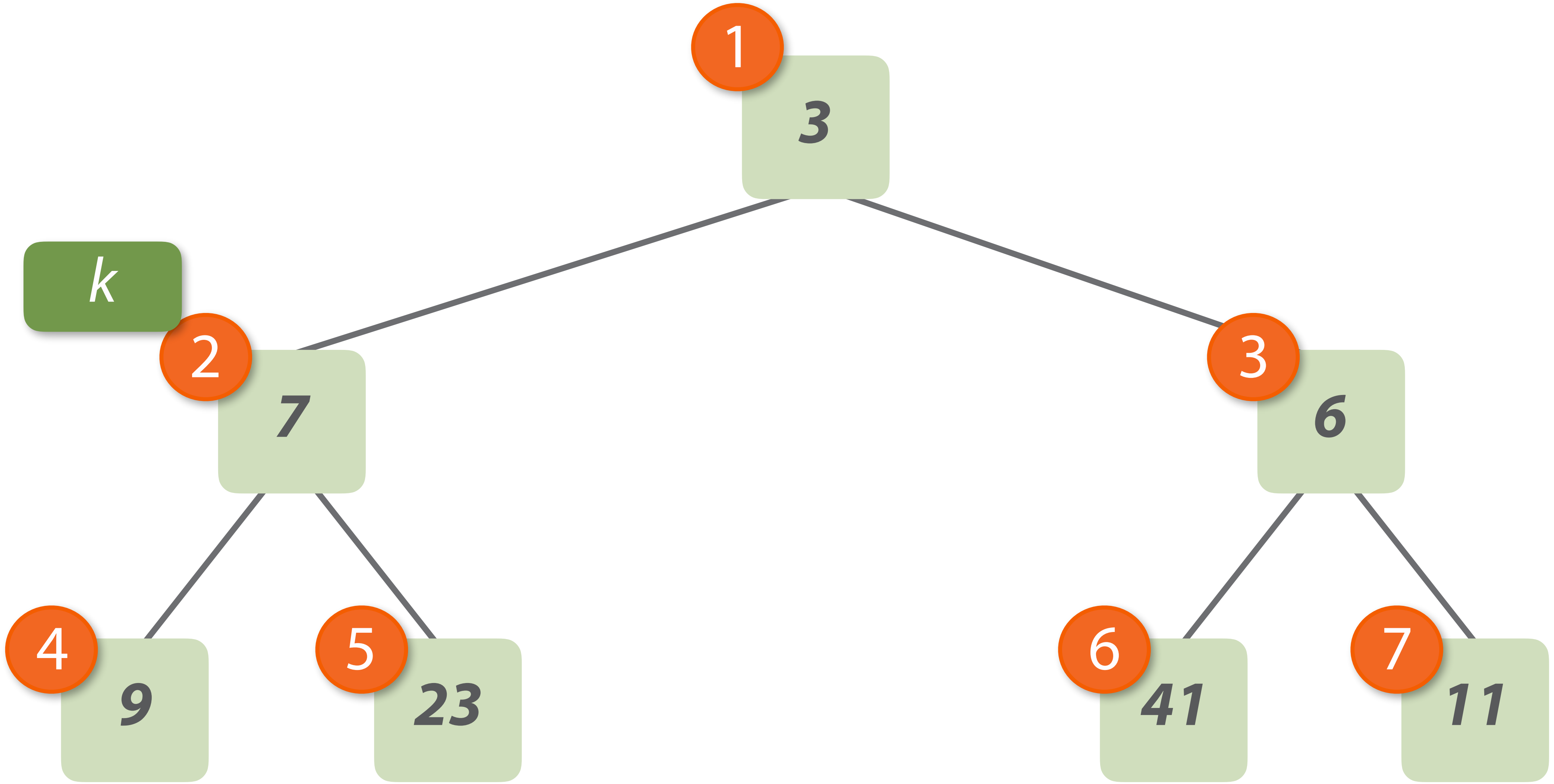
Min-heaps



Rule: each element is *smaller* than its children

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	***7***	***6***	***9***	***23***	***41***	***11***								

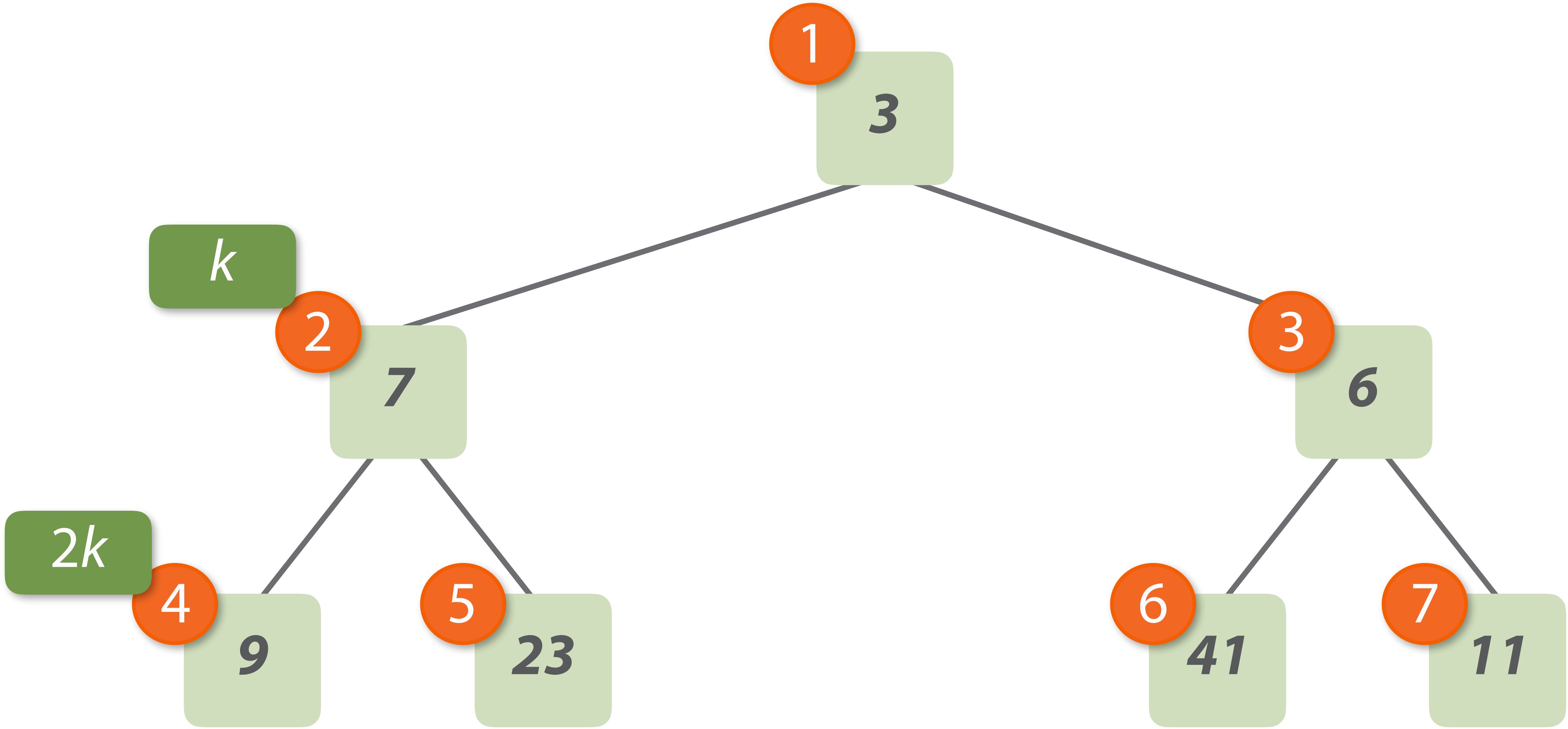
Min-heaps



Rule: each element is *smaller* than its children

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	*7*	*6*	*9*	*23*	*41*	*11*								

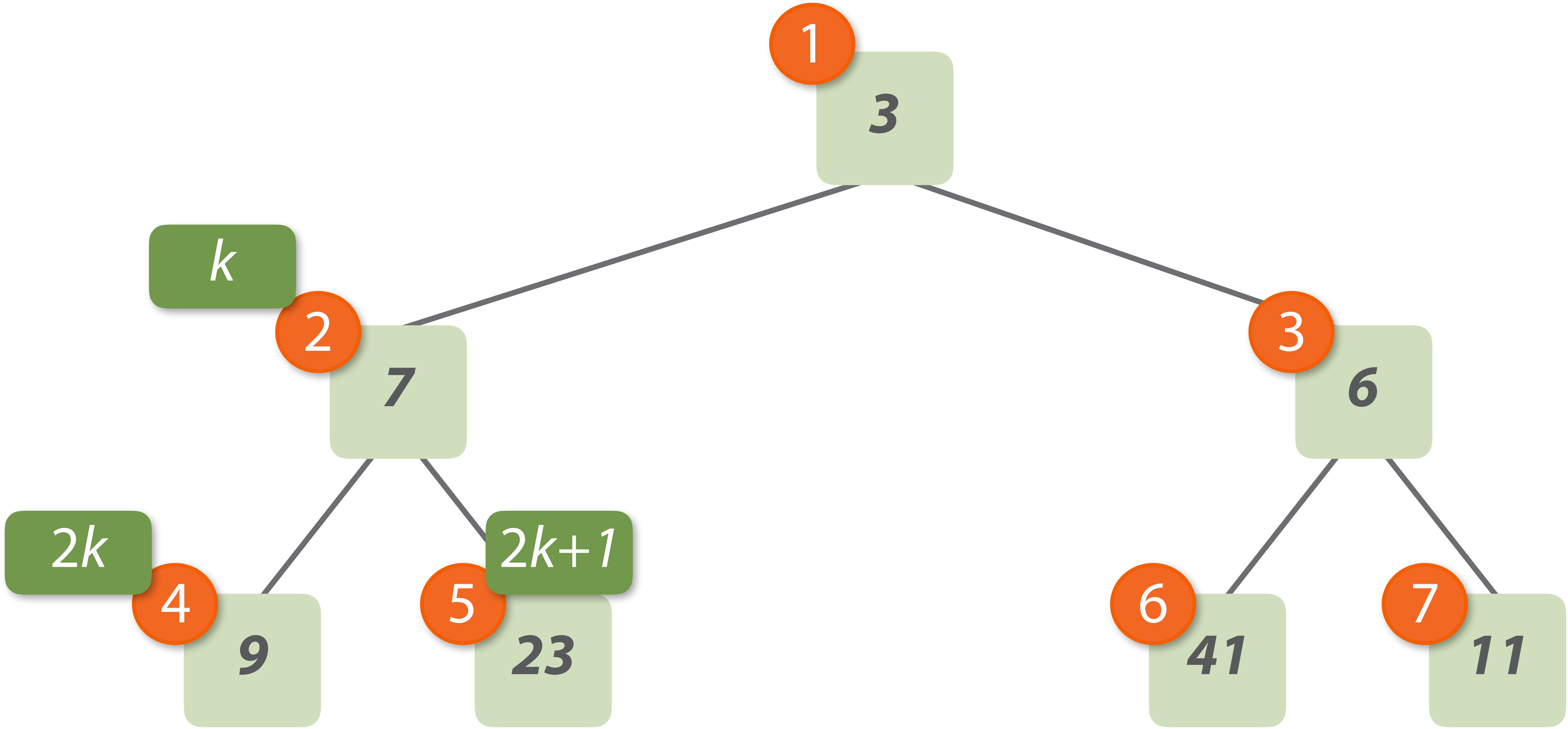
Min-heaps



Rule: each element is *smaller* than its children

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	7	6	9	23	41	11								

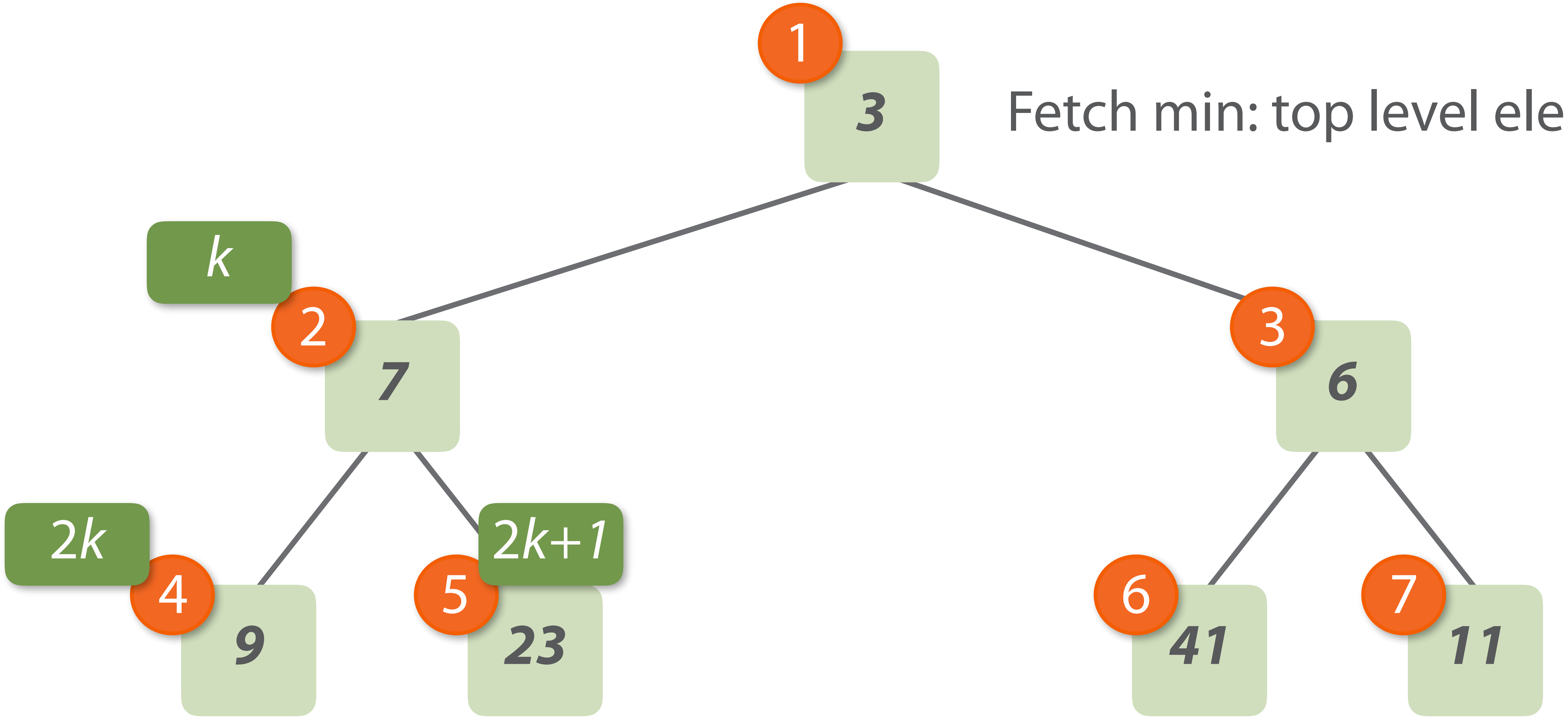
Min-heaps



Rule: each element is *smaller* than its children

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	*7*	*6*	*9*	*23*	*41*	*11*								

Min-heaps

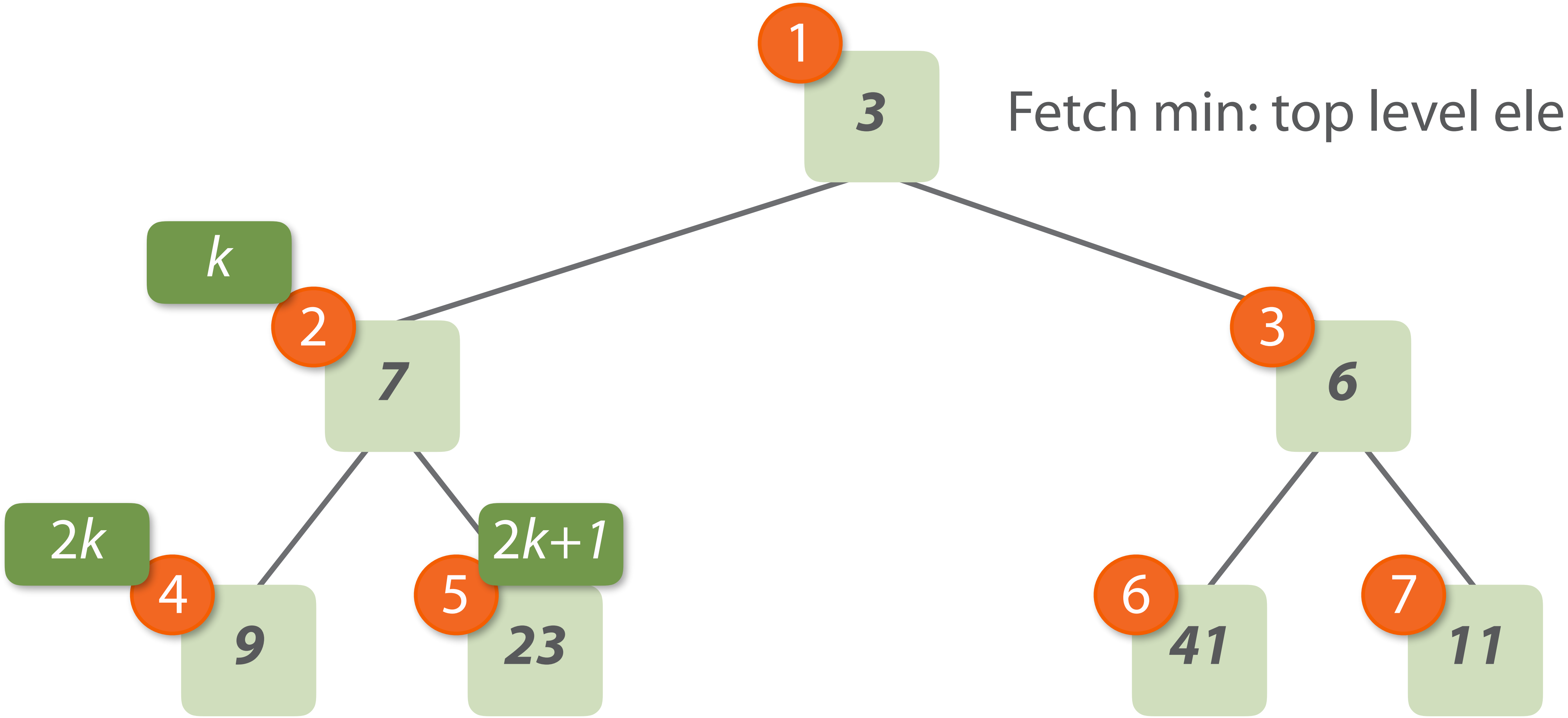


Fetch min: top level element

Rule: each element is *smaller* than its children

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	7	6	9	23	41	11								

Min-heaps

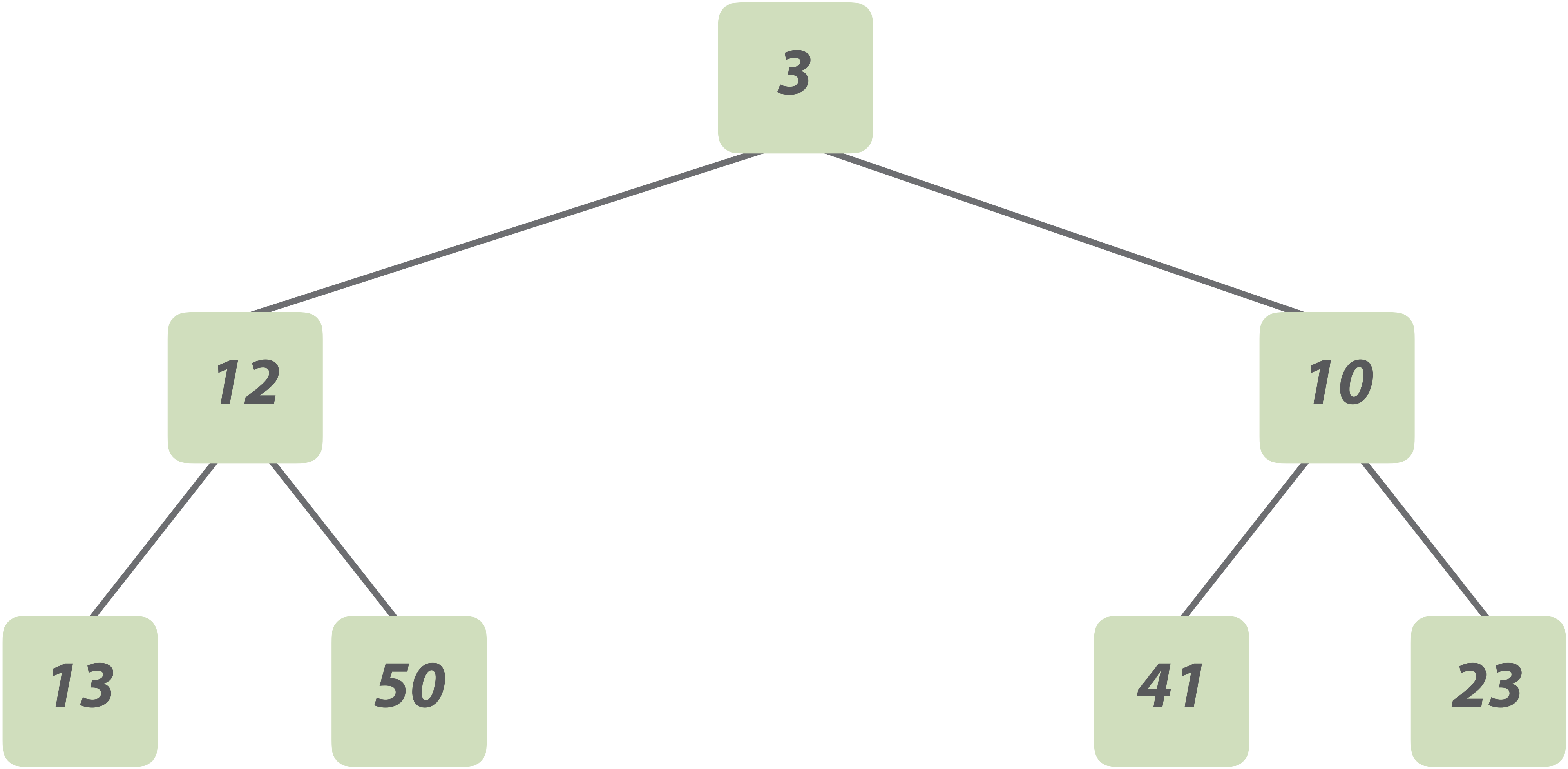


Fetch min: top level element O(1)

Rule: each element is *smaller* than its children

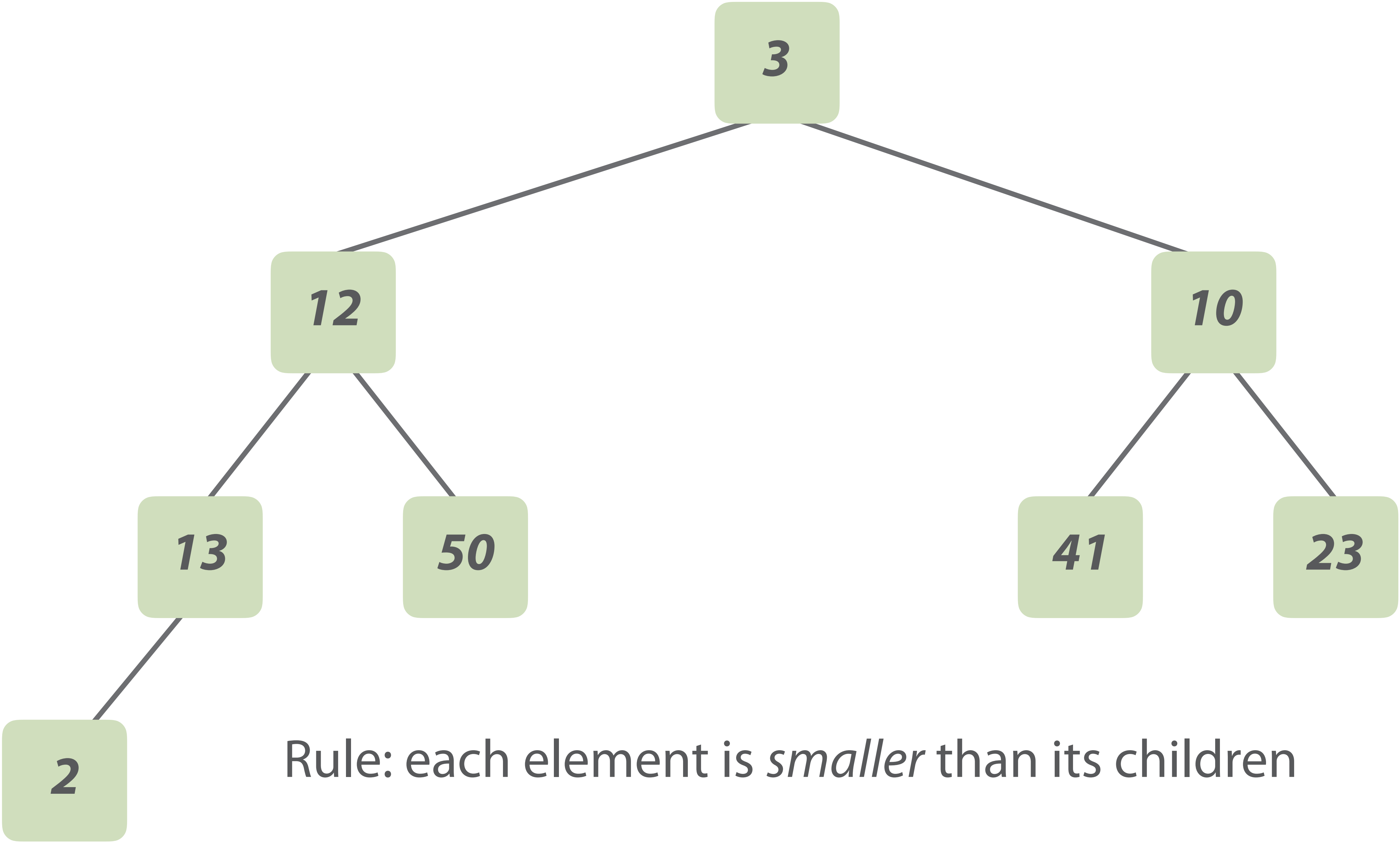
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	7	6	9	23	41	11								

Insertion



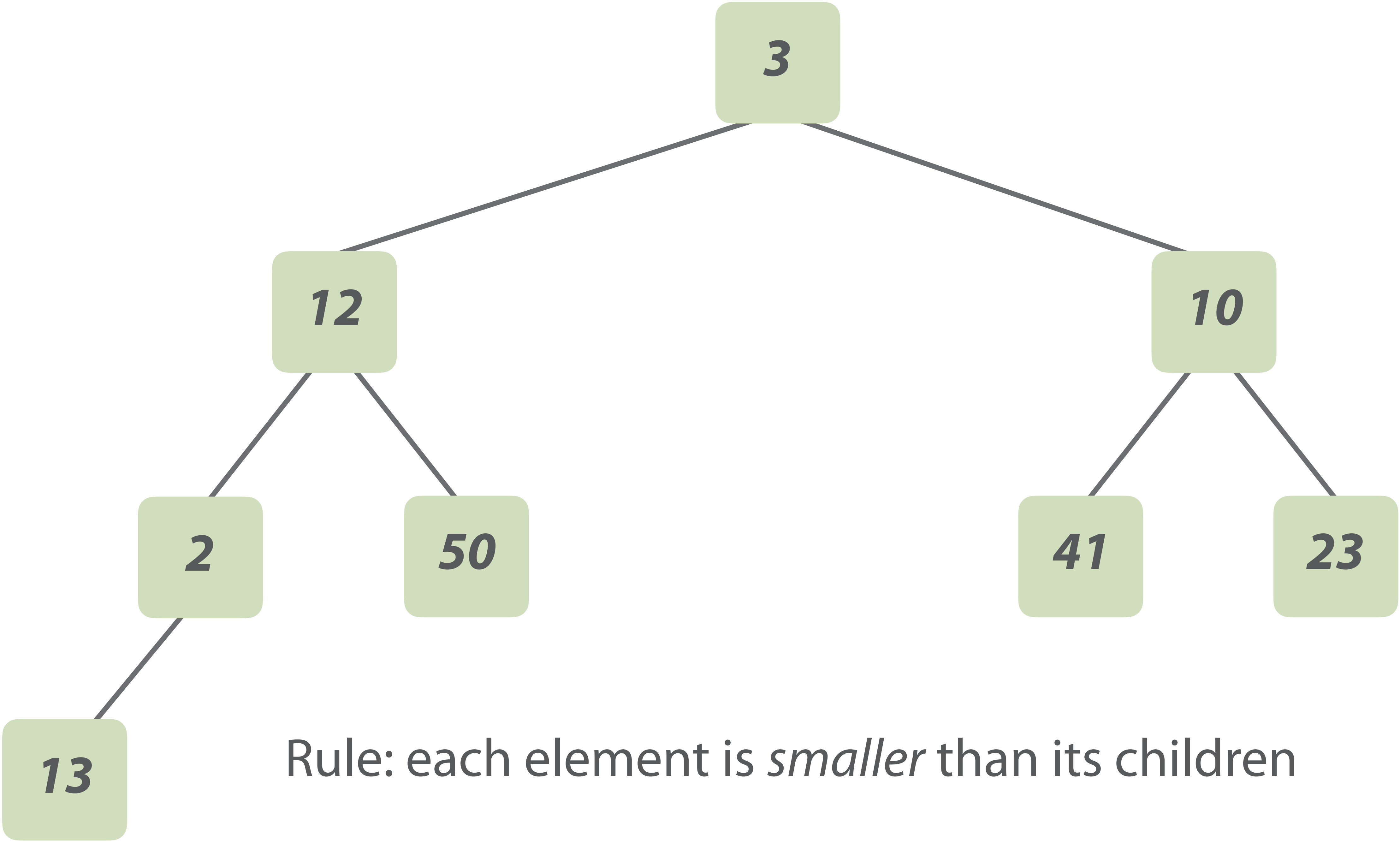
Rule: each element is *smaller* than its children

Insertion



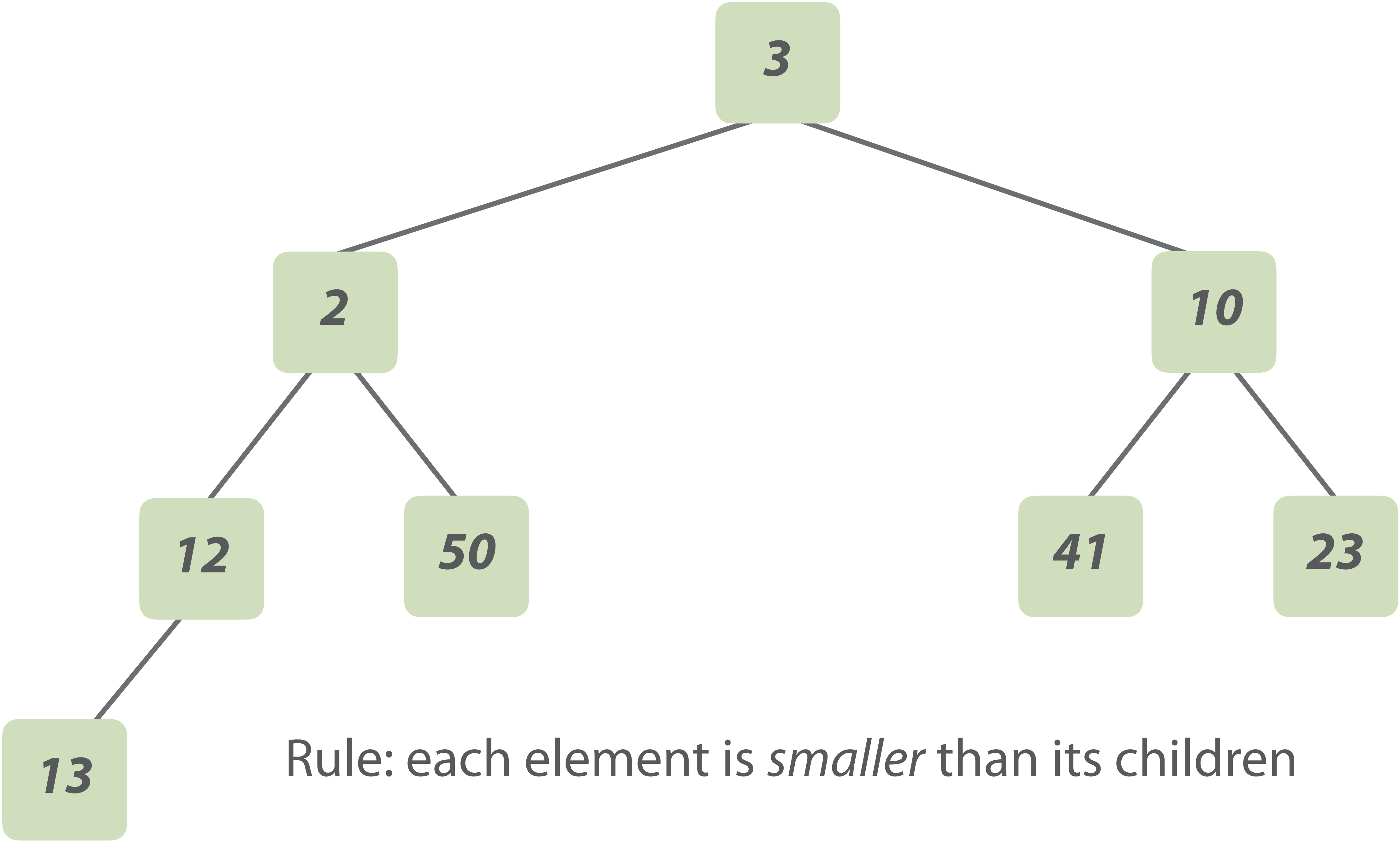
Rule: each element is *smaller* than its children

Insertion



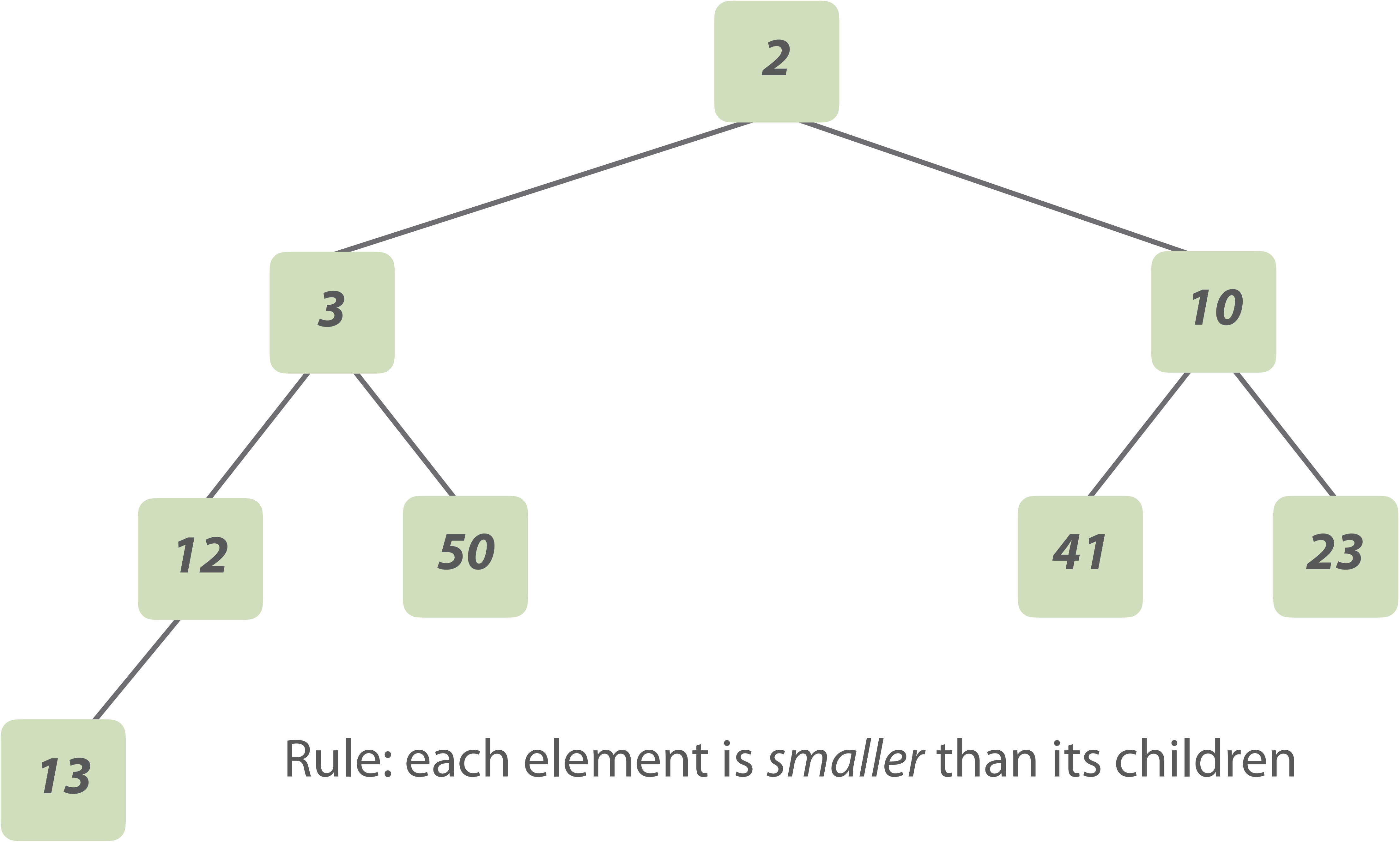
Rule: each element is *smaller* than its children

Insertion



Rule: each element is *smaller* than its children

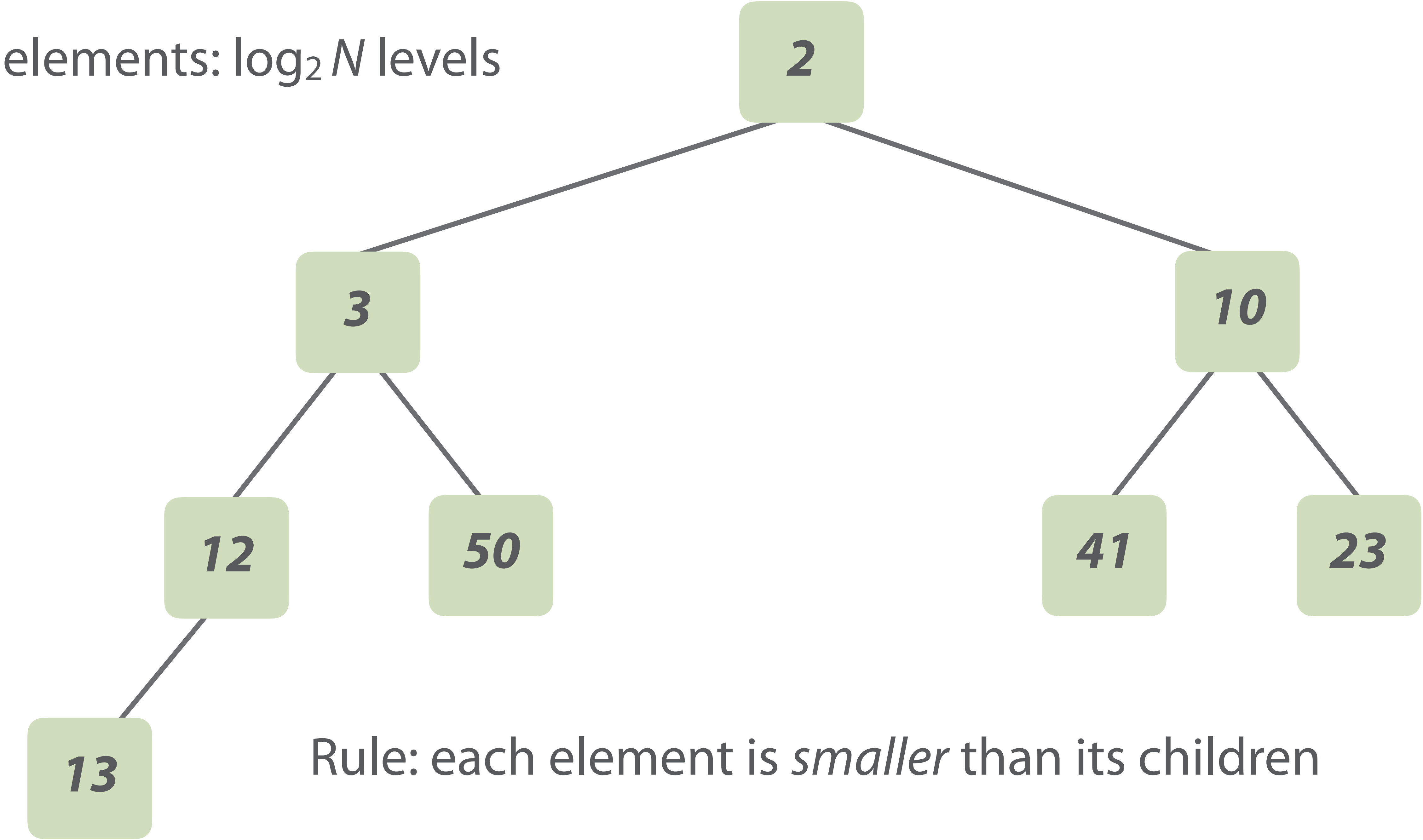
Insertion



Rule: each element is *smaller* than its children

Insertion

For *N* elements: $\log_2 N$ levels

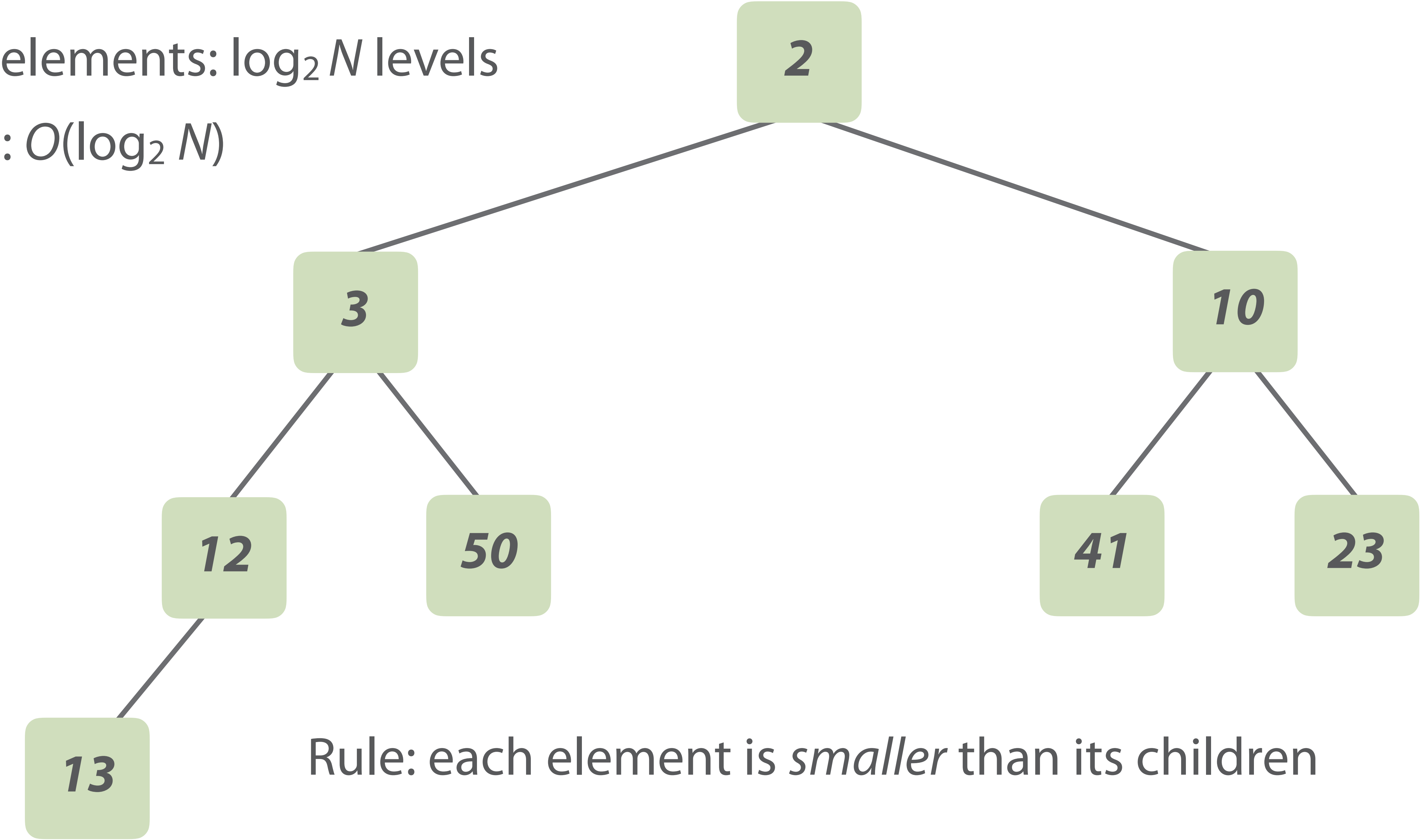


Rule: each element is *smaller* than its children

Insertion

For N elements: $\log_2 N$ levels

Insert: $O(\log_2 N)$



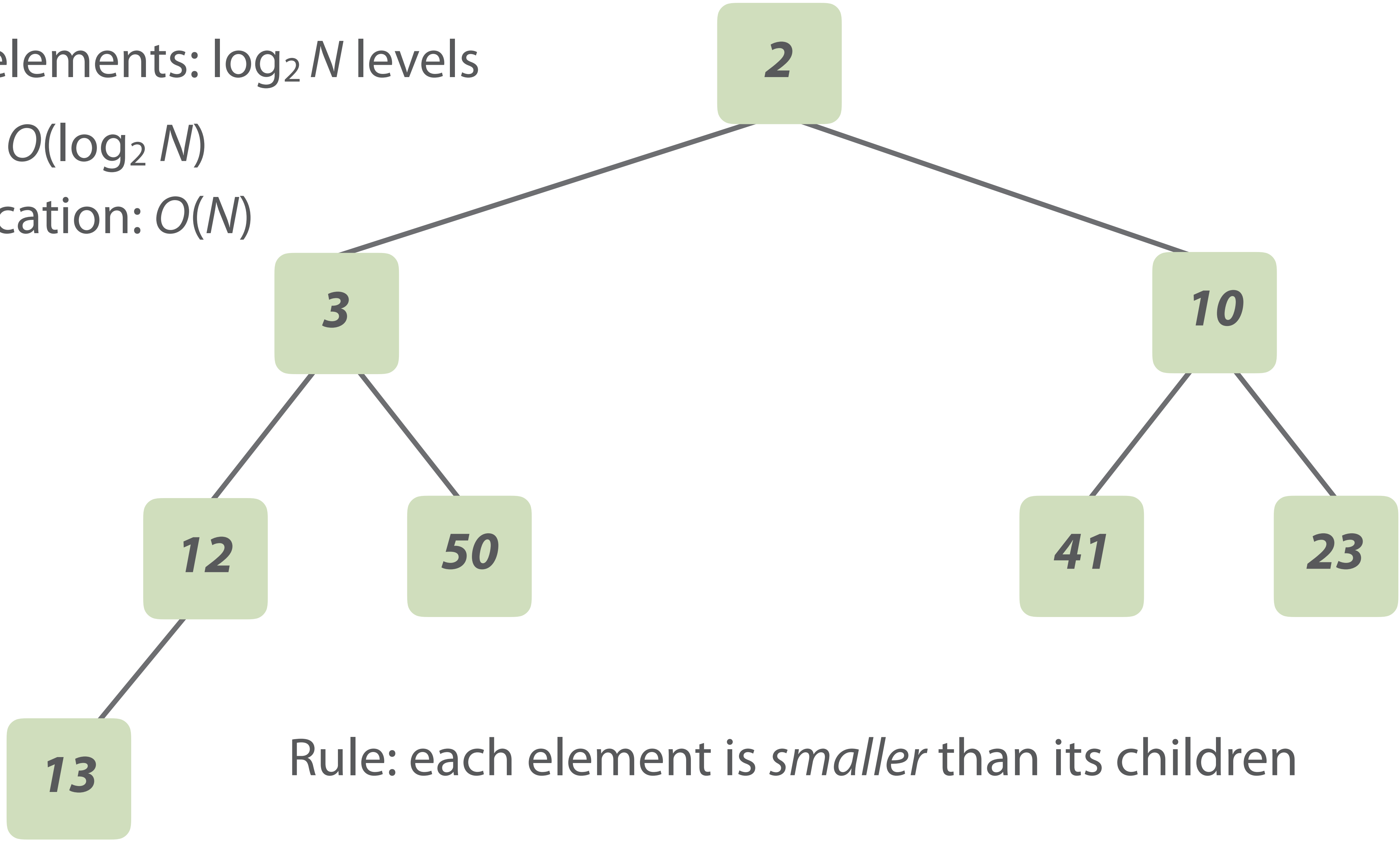
Rule: each element is *smaller* than its children

Insertion

For N elements: $\log_2 N$ levels

Insert: $O(\log_2 N)$

Reallocation: $O(N)$



Rule: each element is *smaller* than its children

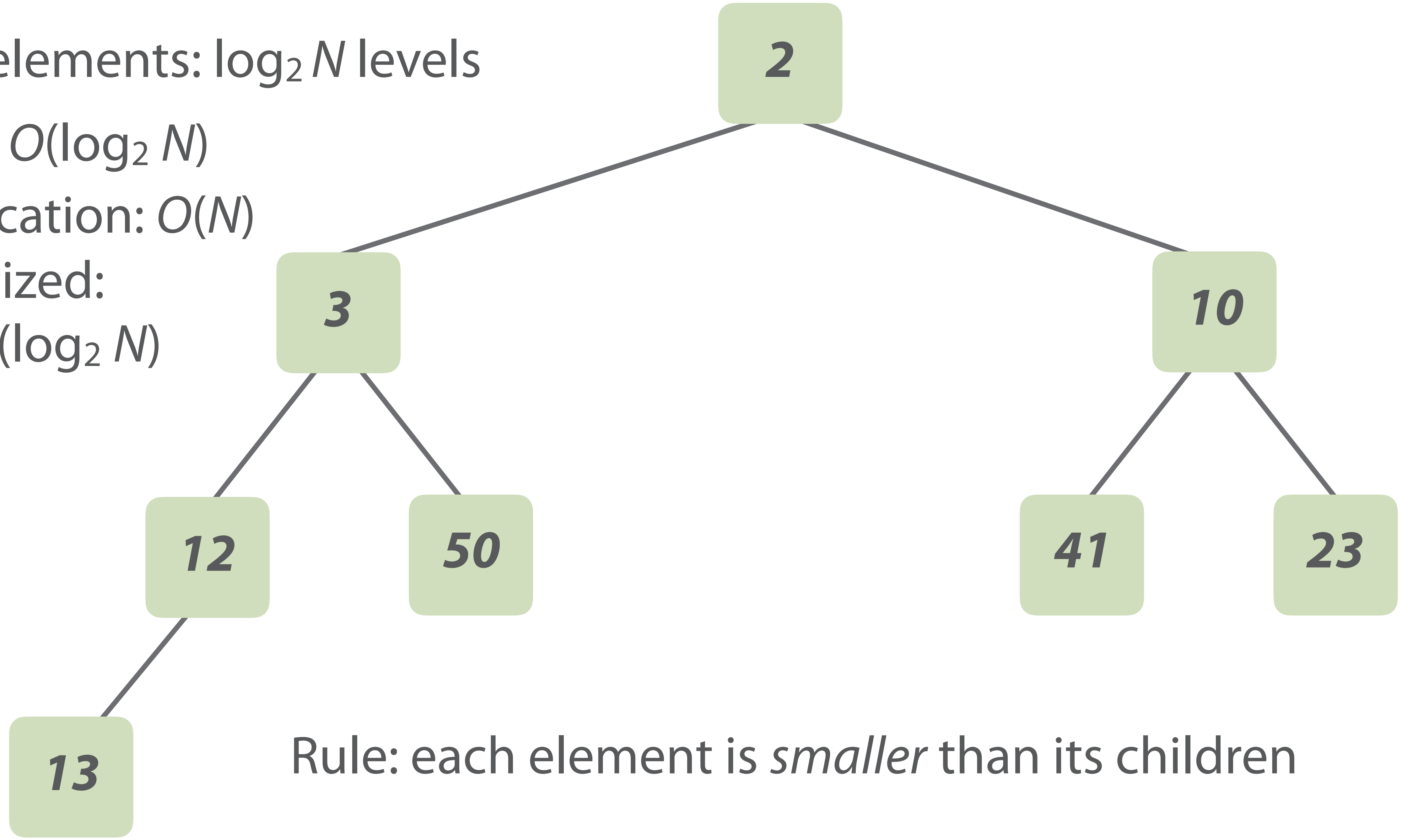
Insertion

For N elements: $\log_2 N$ levels

Insert: $O(\log_2 N)$

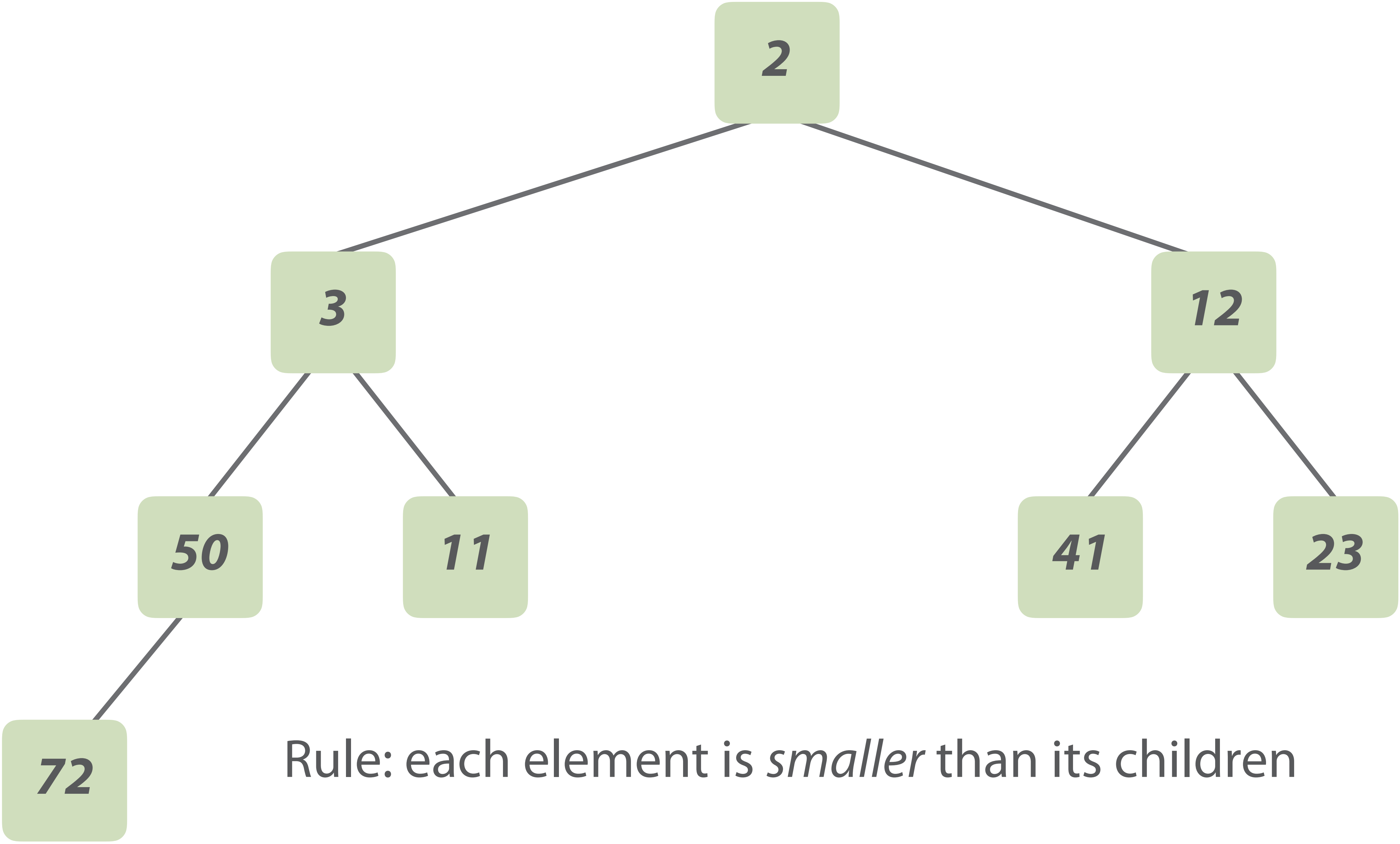
Reallocation: $O(N)$

Amortized:
$O(\log_2 N)$



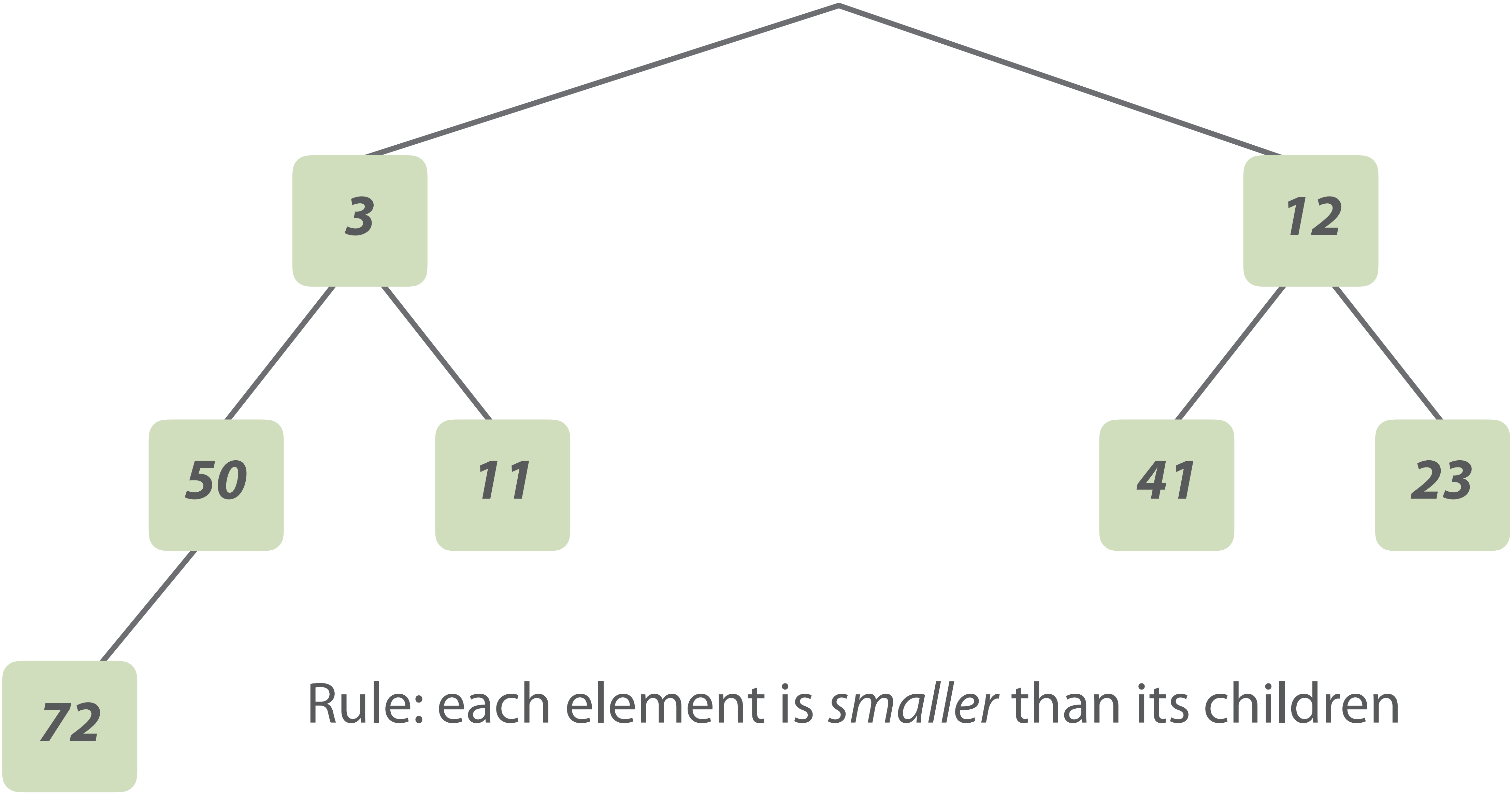
Rule: each element is *smaller* than its children

Removal



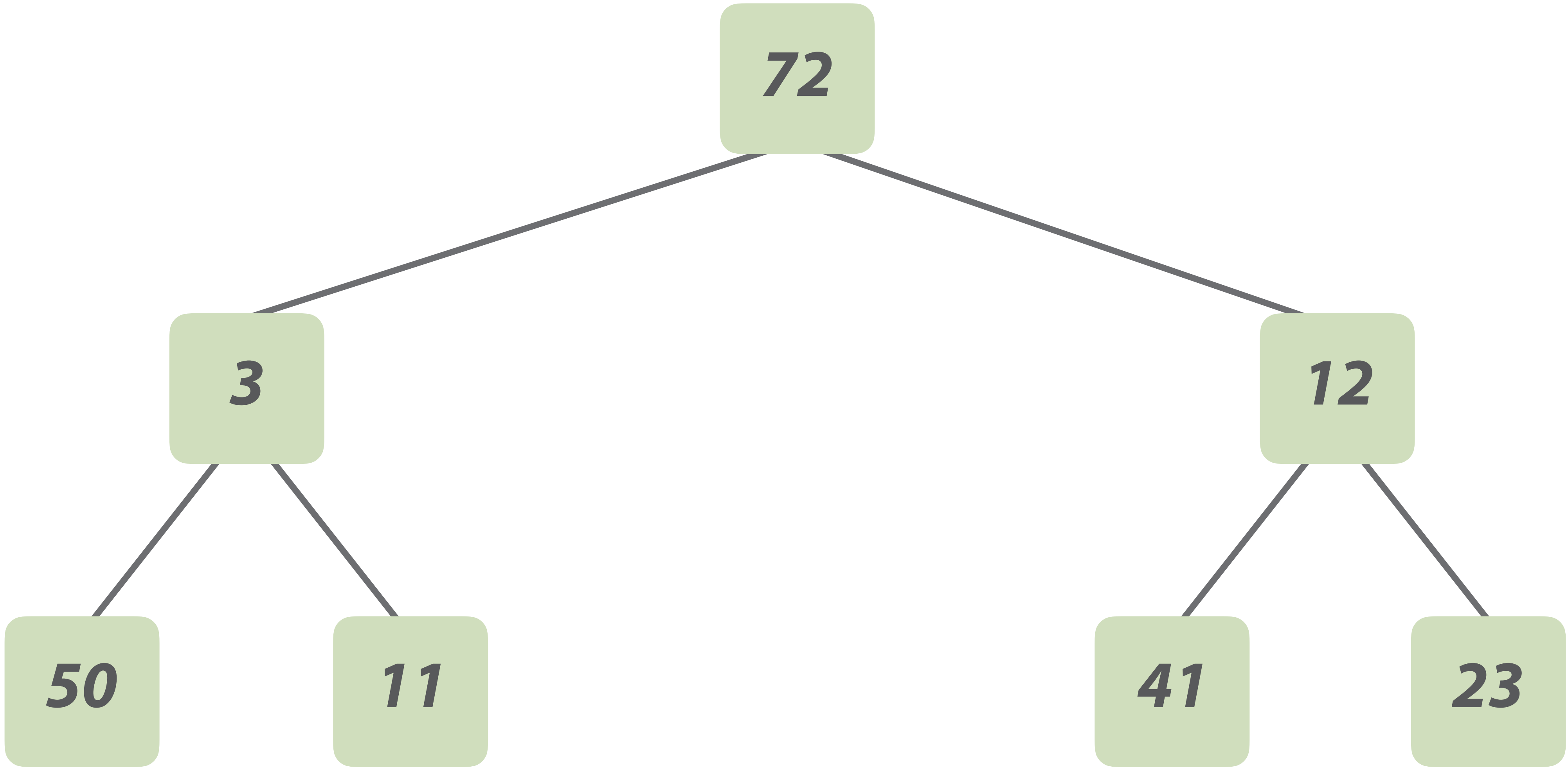
Rule: each element is *smaller* than its children

Removal



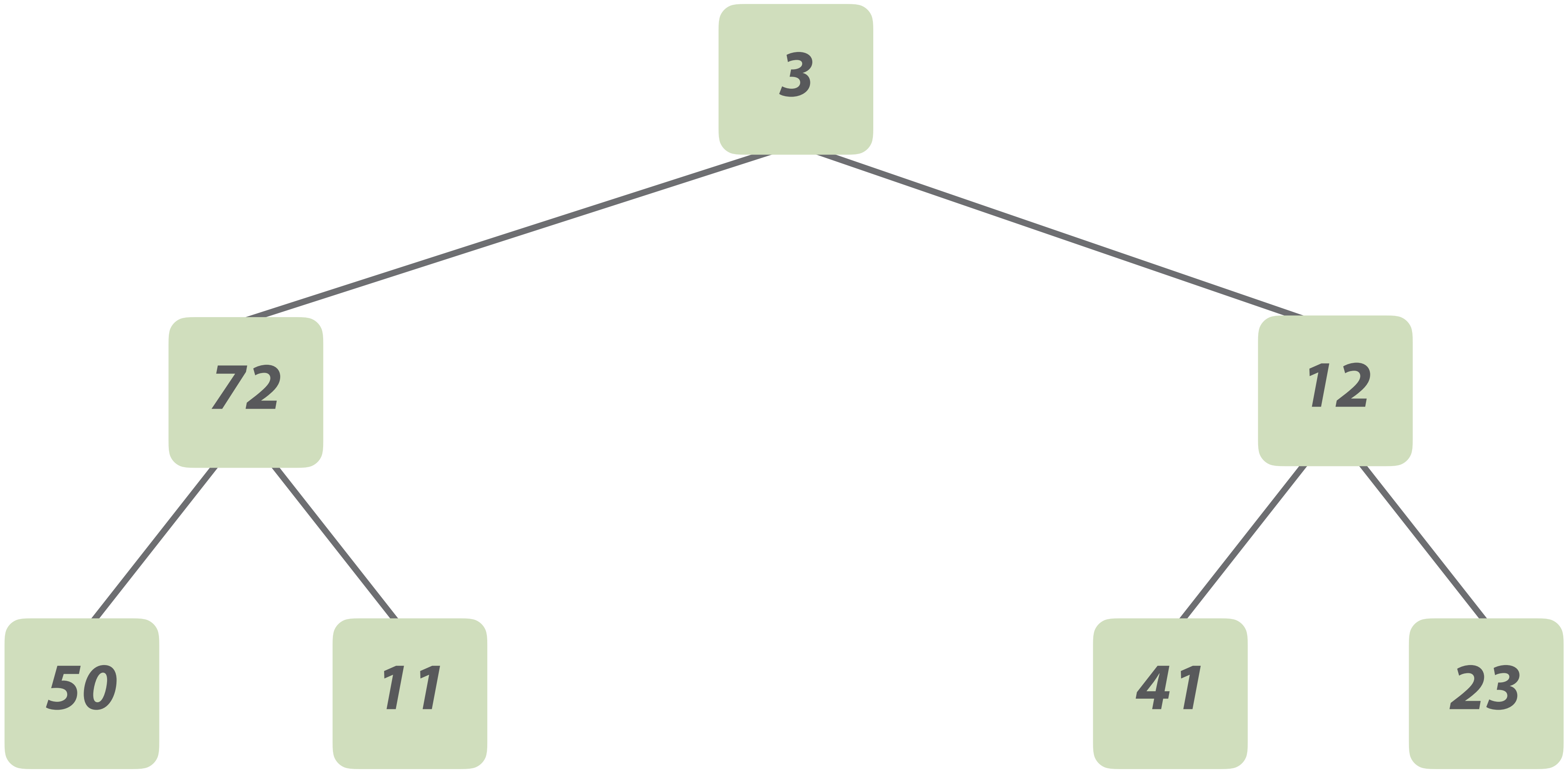
Rule: each element is *smaller* than its children

Removal



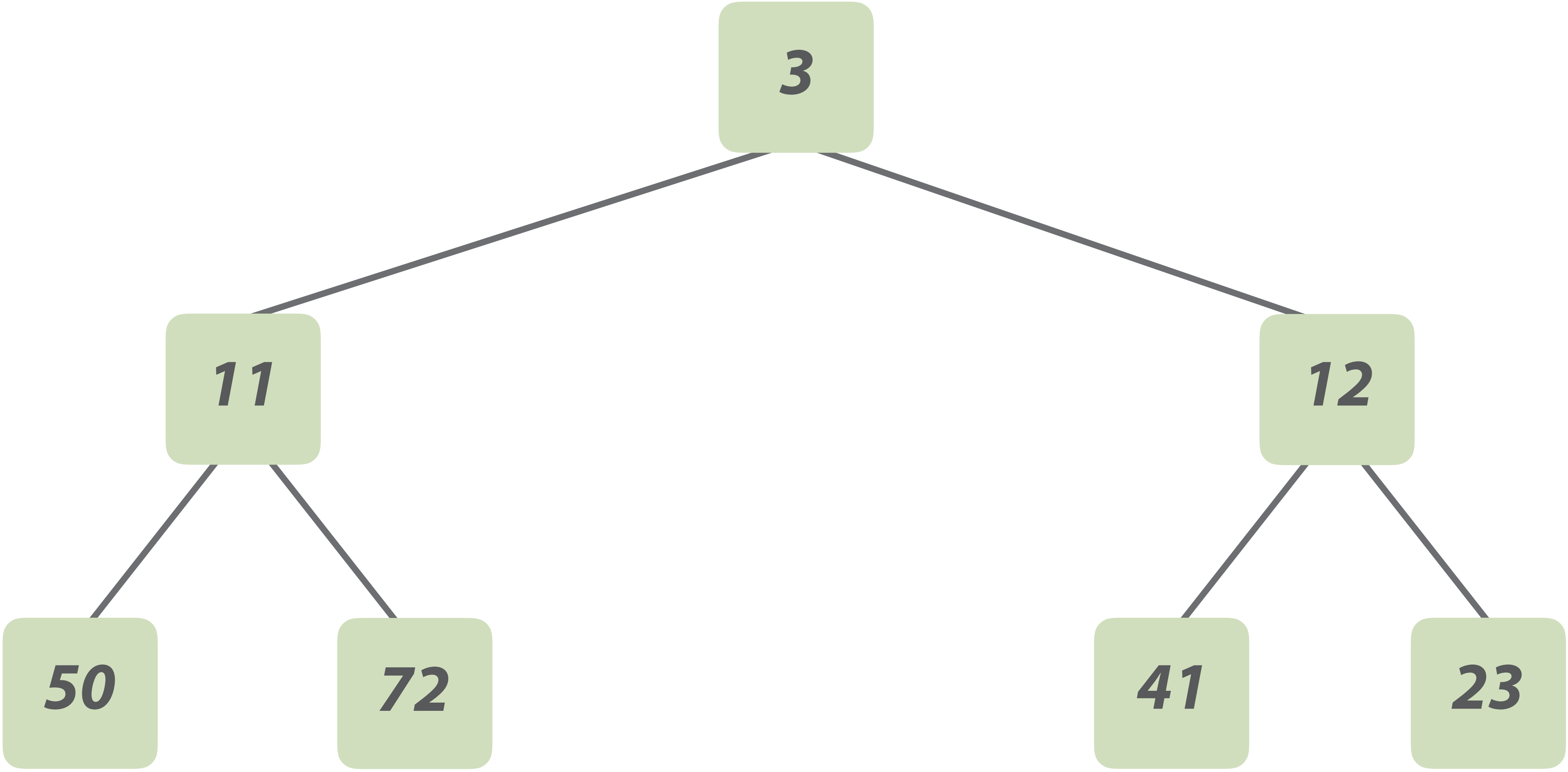
Rule: each element is *smaller* than its children

Removal



Rule: each element is *smaller* than its children

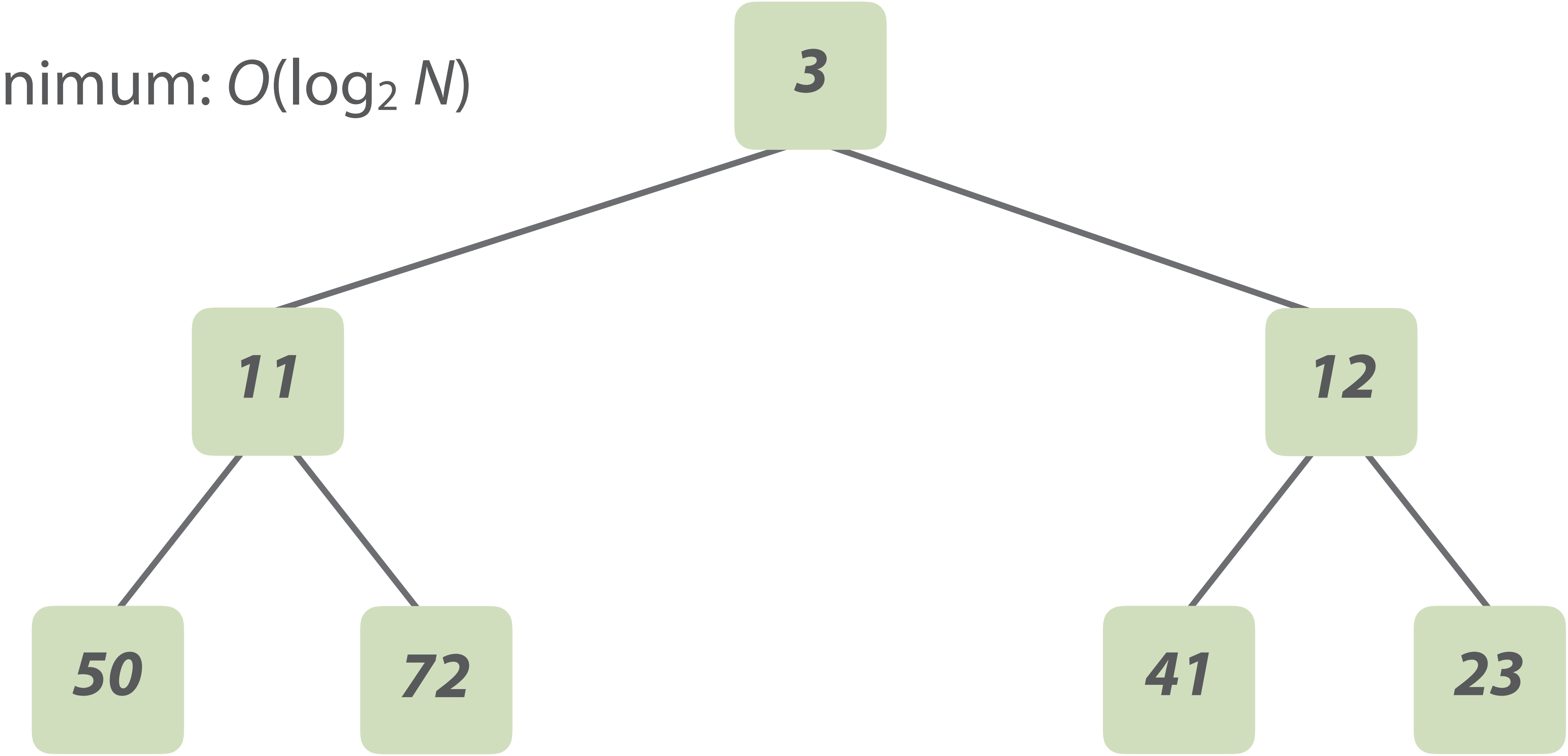
Removal



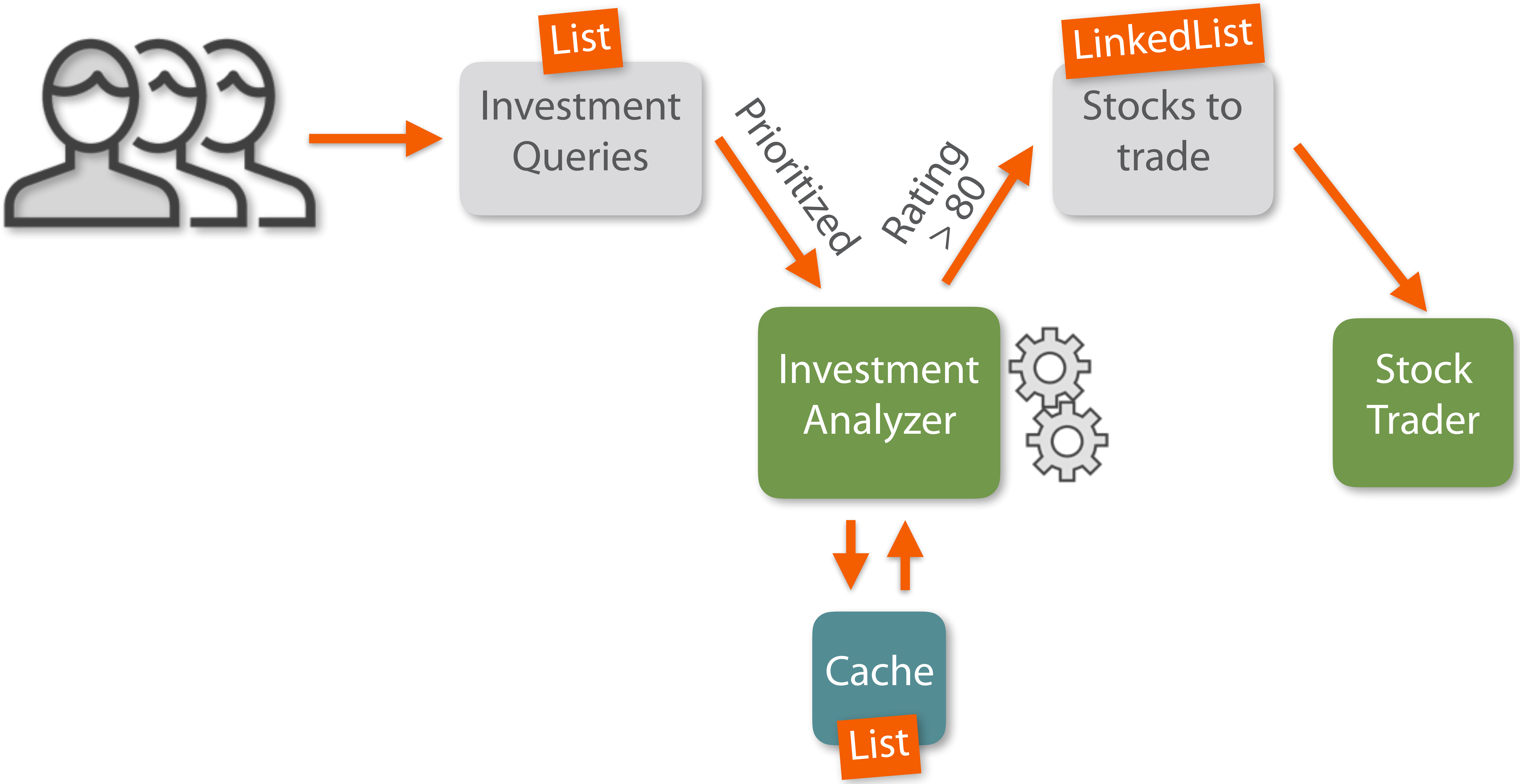
Rule: each element is *smaller* than its children

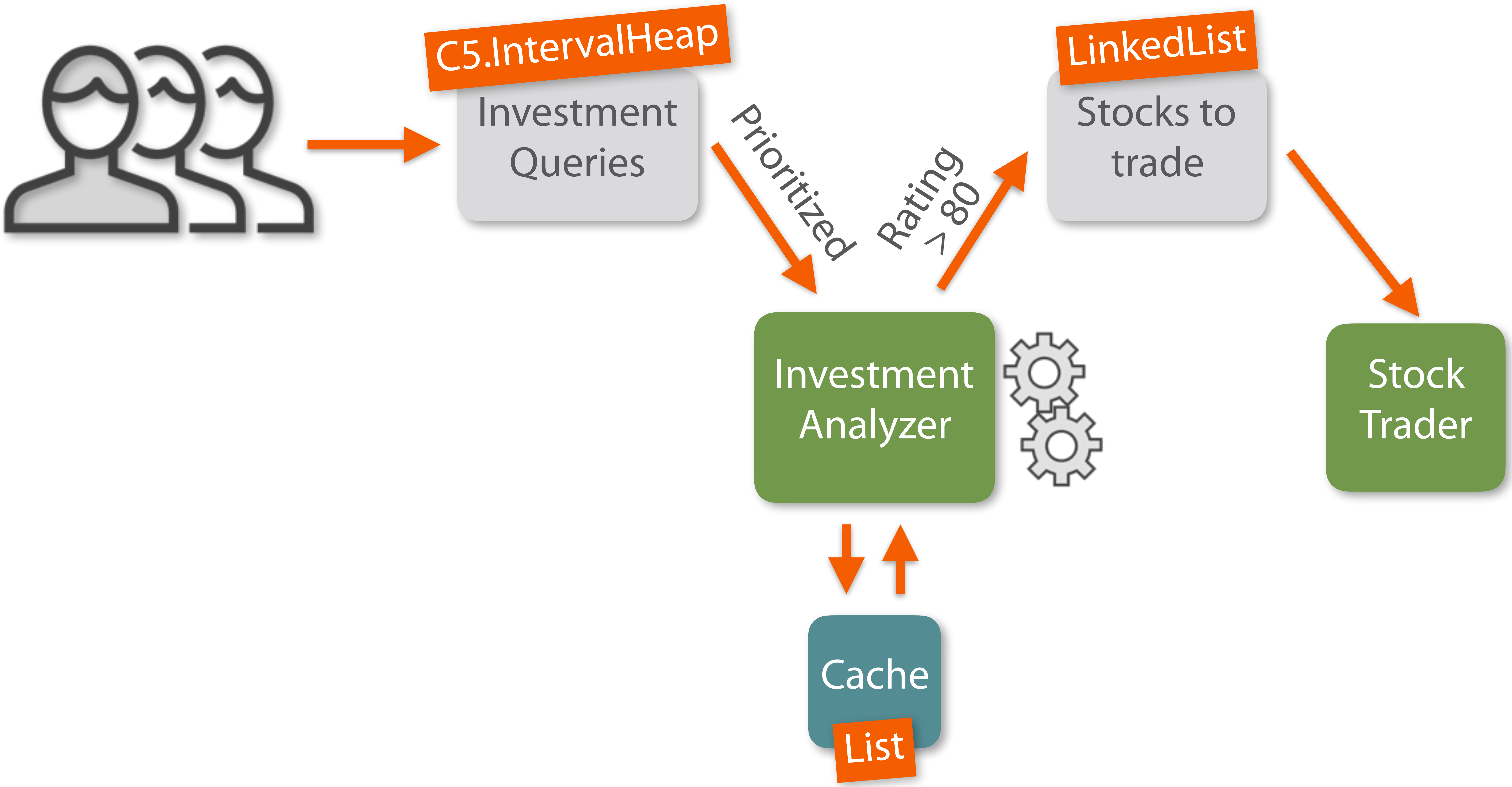
Removal

Remove minimum: $O(\log_2 N)$



Rule: each element is *smaller* than its children

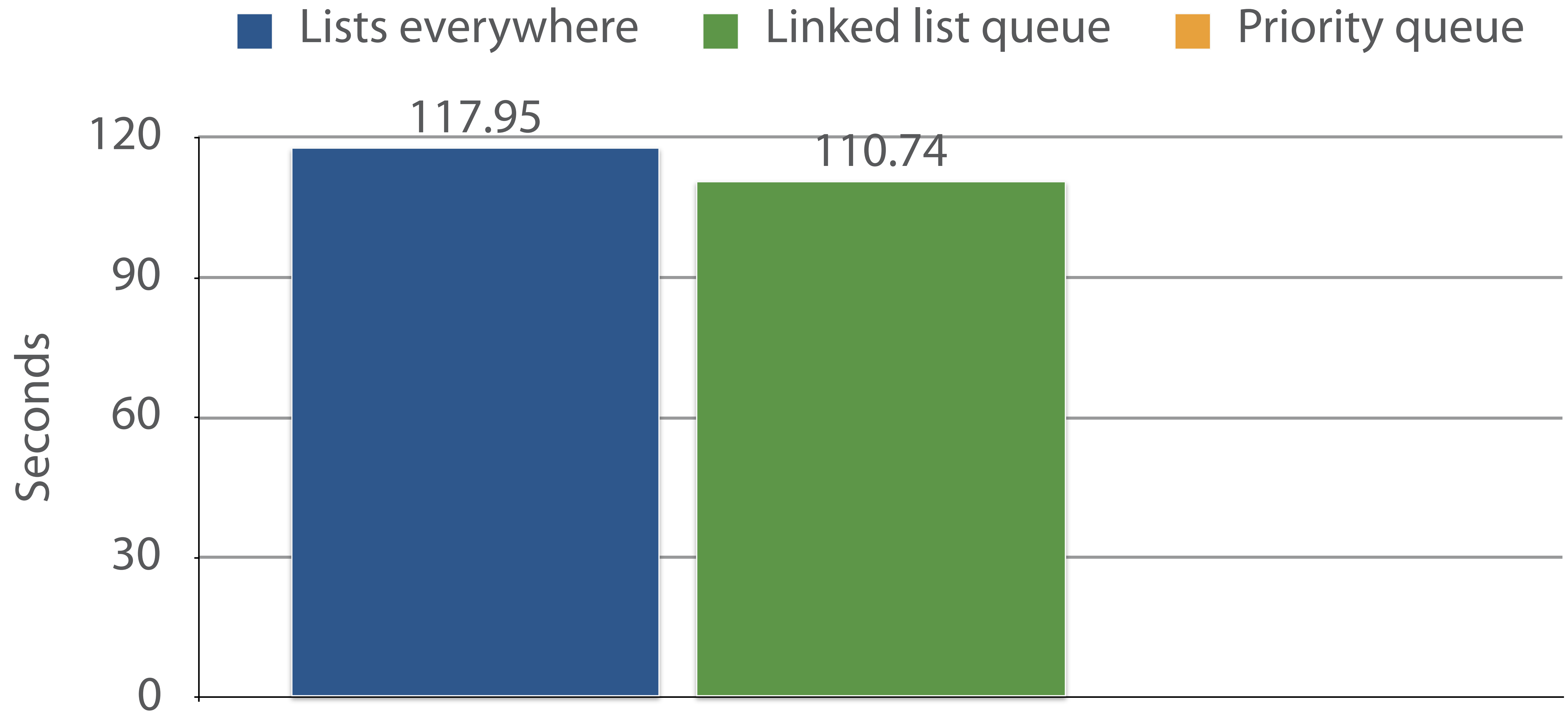

List
Investment Queries
Prioritized
Rating > 80
LinkedList
Stocks to trade
Investment Analyzer
Stock Trader
Cache
List


C5.IntervalHeap
Investment Queries
Prioritized
Rating > 80
LinkedList
Stocks to trade
Investment Analyzer
Stock Trader
Cache
List

Investment Analyzer

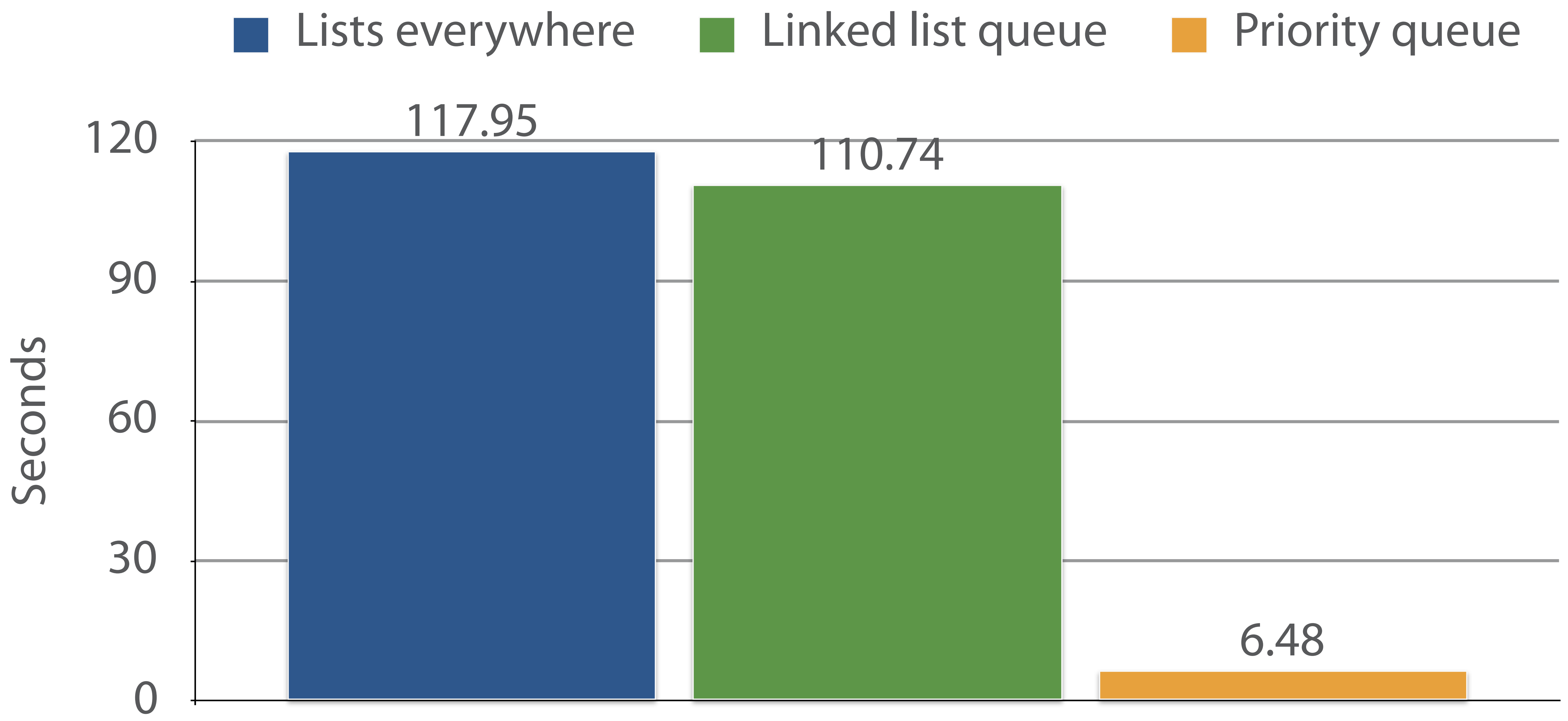
Using a C5.IntervalHeap as priority queue

Effect



🟦 Lists everywhere
🟩 Linked list queue
🟧 Priority queue
Seconds
120
90
60
30
0
117.95
110.74

Effect


Lists everywhere
Linked list queue
Priority queue
Seconds
120
90
60
30
0
117.95
110.74
6.48

Data Structures

Dynamic array

Hash table

Linked list

Priority queue

Data Structures

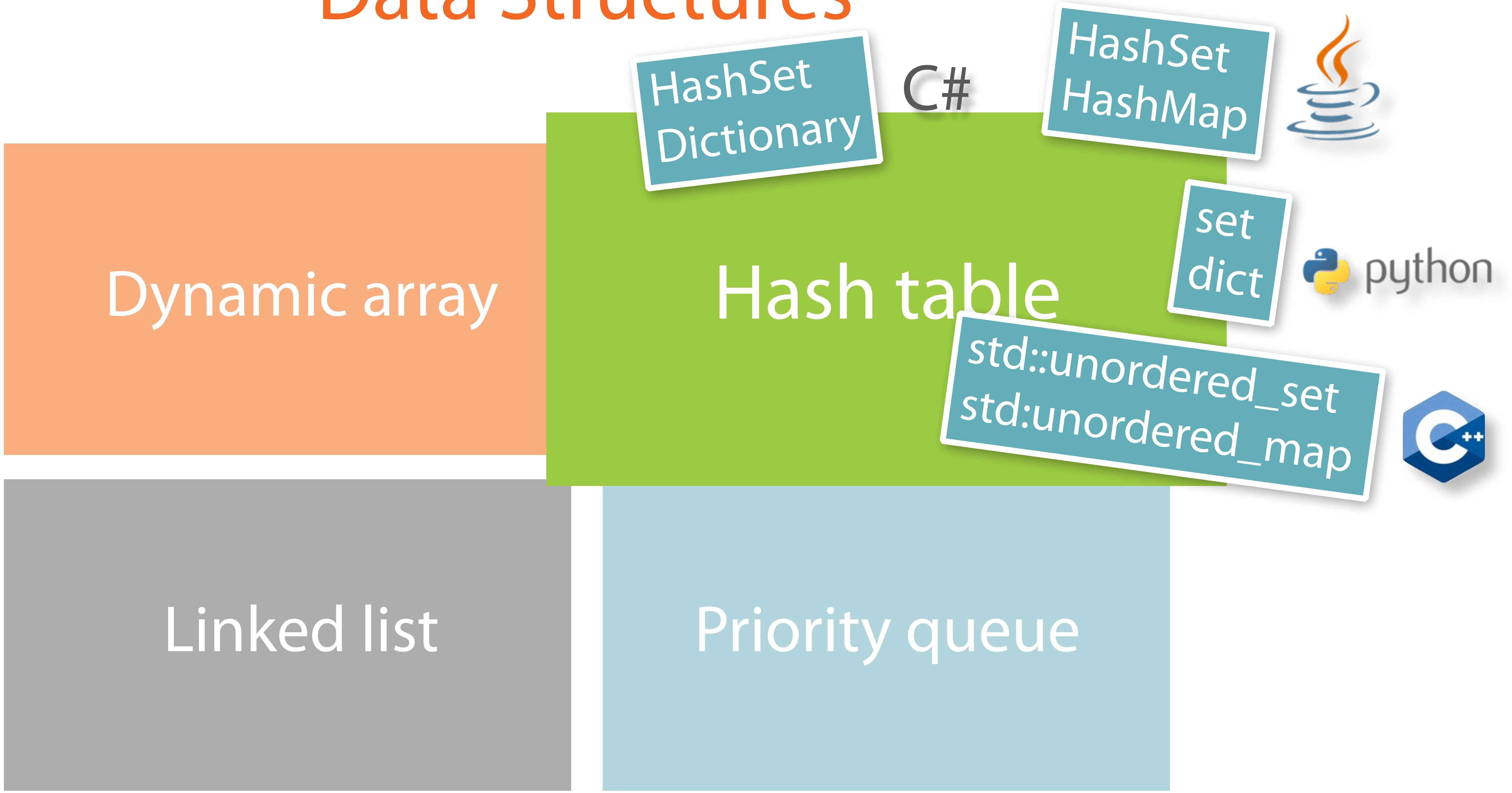
Dynamic array

Hash table

Linked list

Priority queue

Data Structures



Dynamic array

Hash table

HashSet
Dictionary

C#

HashSet
HashMap

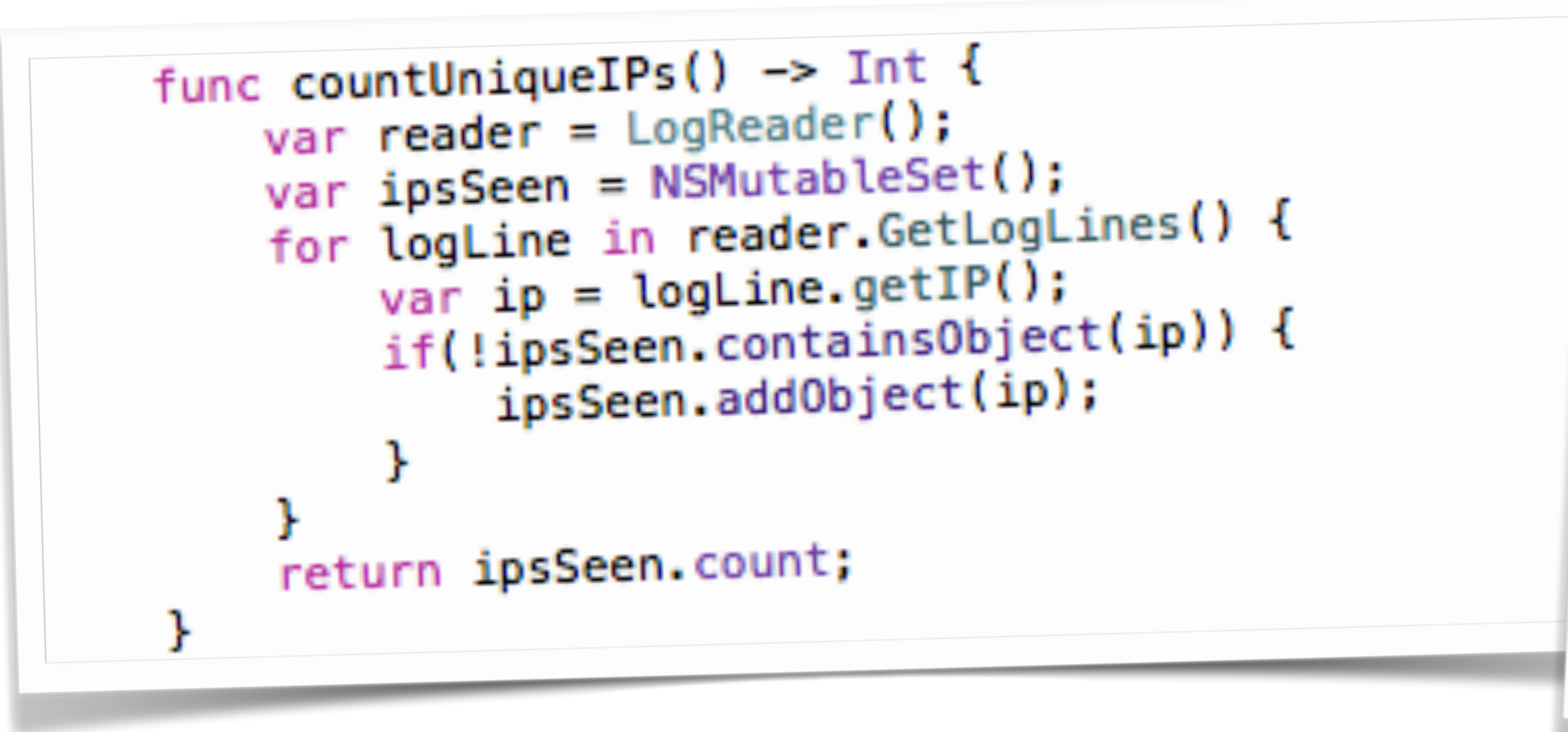
set
dict

python

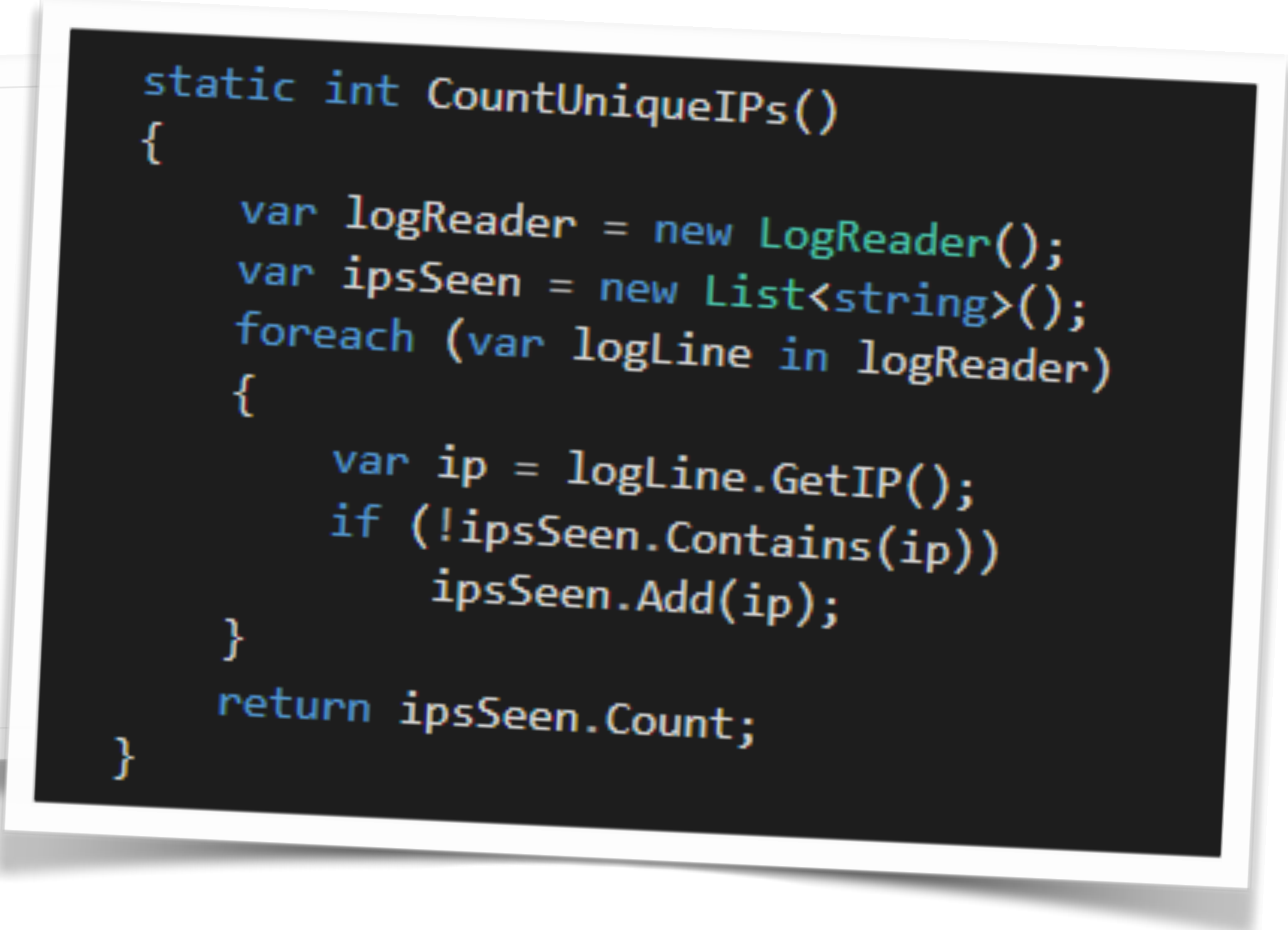
std::unordered_set
std:unordered_map

Linked list

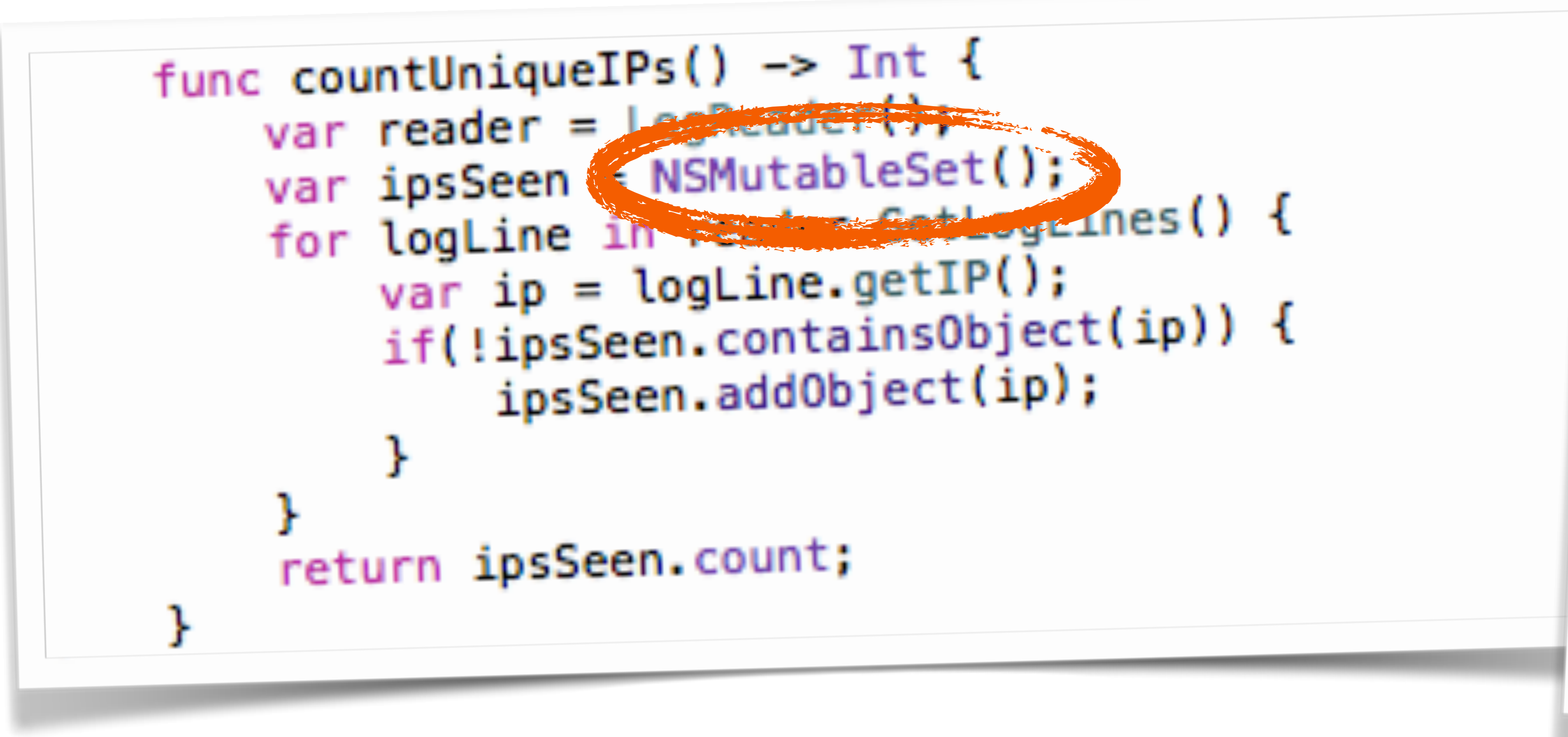
Priority queue



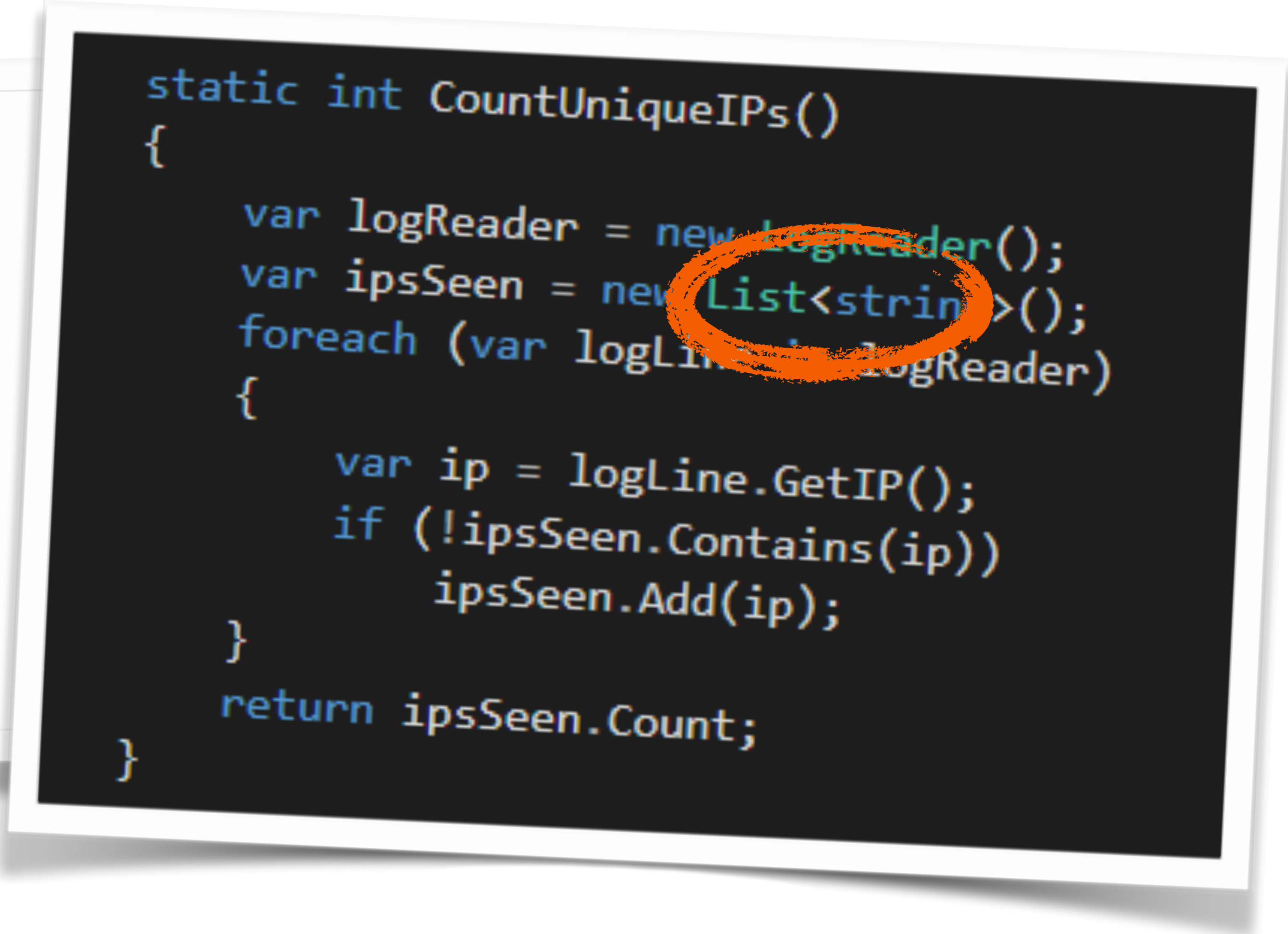
```
func countUniqueIPs() -> Int {
    var reader = LogReader();
    var ipsSeen = NSMutableSet();
    for logLine in reader.GetLogLines() {
        var ip = logLine.getIP();
        if(!ipsSeen.containsObject(ip)) {
            ipsSeen.addObject(ip);
        }
    }
    return ipsSeen.count;
}
```



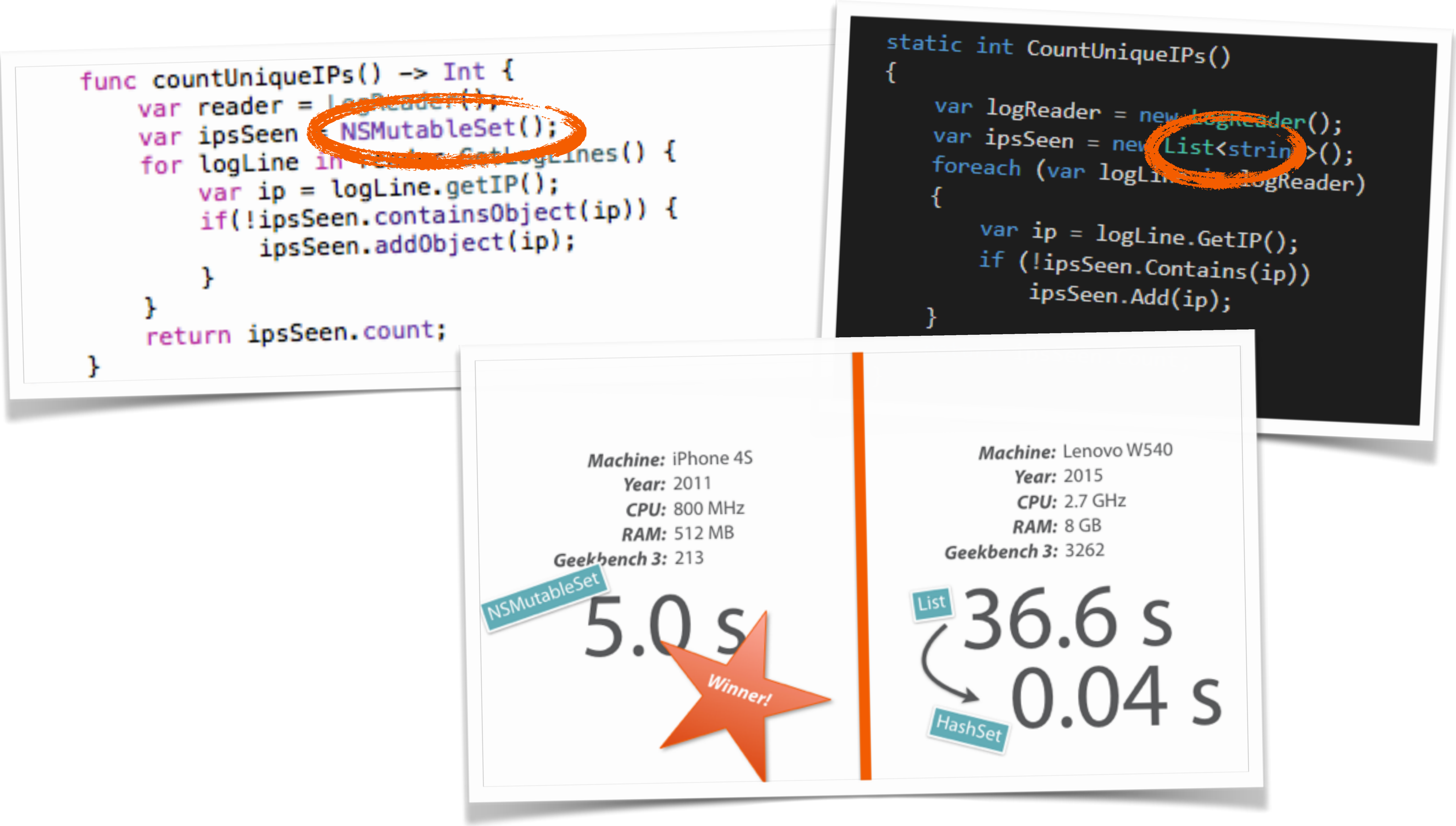
```
static int CountUniqueIPs()
{
    var logReader = new LogReader();
    var ipsSeen = new List<string>();
    foreach (var logLine in logReader)
    {
        var ip = logLine.GetIP();
        if (!ipsSeen.Contains(ip))
            ipsSeen.Add(ip);
    }
    return ipsSeen.Count;
}
```



```
func countUniqueIPs() -> Int {
    var reader = LogReader();
    var ipsSeen = NSMutableSet();
    for logLine in reader.GetLogLines() {
        var ip = logLine.getIP();
        if(!ipsSeen.containsObject(ip)) {
            ipsSeen.addObject(ip);
        }
    }
    return ipsSeen.count;
}
```



```
static int CountUniqueIPs()
{
    var logReader = new LogReader();
    var ipsSeen = new List<string>();
    foreach (var logLine in logReader)
    {
        var ip = logLine.GetIP();
        if (!ipsSeen.Contains(ip))
            ipsSeen.Add(ip);
    }
    return ipsSeen.Count;
}
```



```
func countUniqueIPs() -> Int {
    var reader = LogReader();
    var ipsSeen = NSMutableSet();
    for logLine in reader.GetLogLines() {
        var ip = logLine.getIP();
        if(!ipsSeen.containsObject(ip)) {
            ipsSeen.addObject(ip);
        }
    }
    return ipsSeen.count;
}
```

```
static int CountUniqueIPs()
{
    var logReader = new LogReader();
    var ipsSeen = new List<string>();
    foreach (var logLine in logReader)
    {
        var ip = logLine.GetIP();
        if (!ipsSeen.Contains(ip))
            ipsSeen.Add(ip);
    }
```

Machine: iPhone 4S
Year: 2011
CPU: 800 MHz
RAM: 512 MB
Geekbench 3: 213

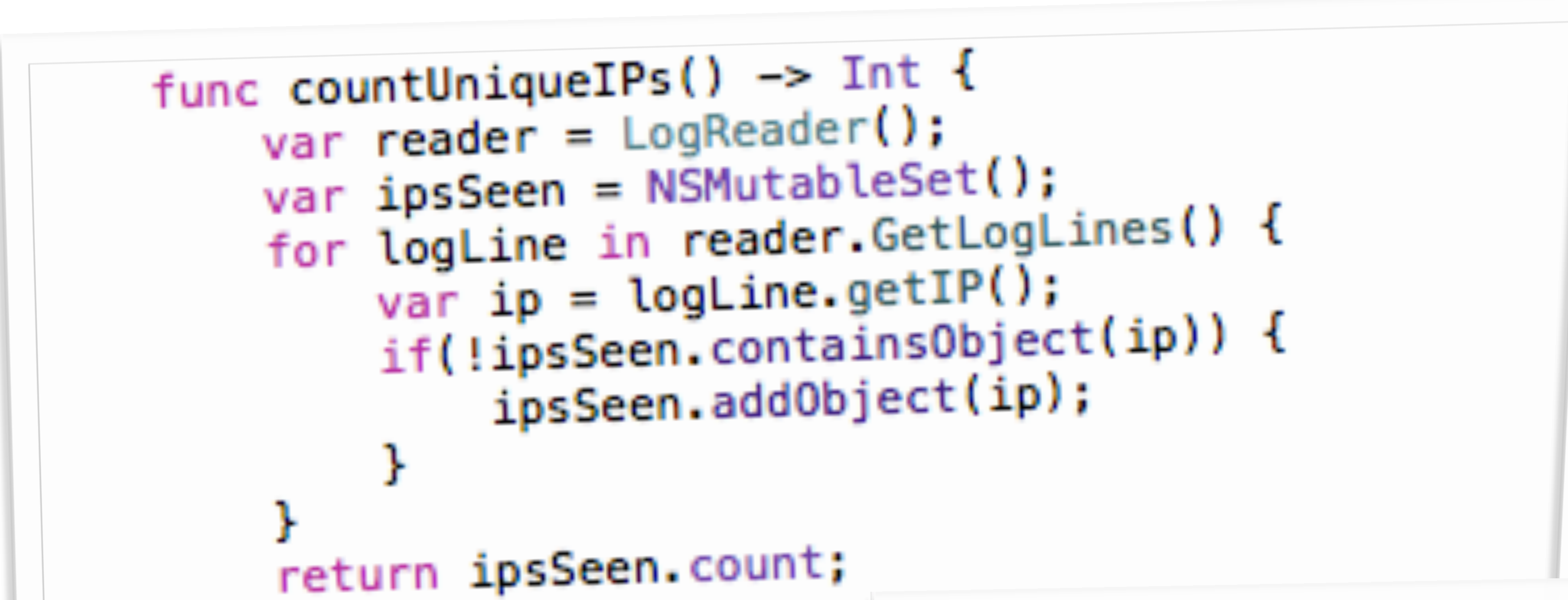
NSMutableSet 5.0 s

Winner!

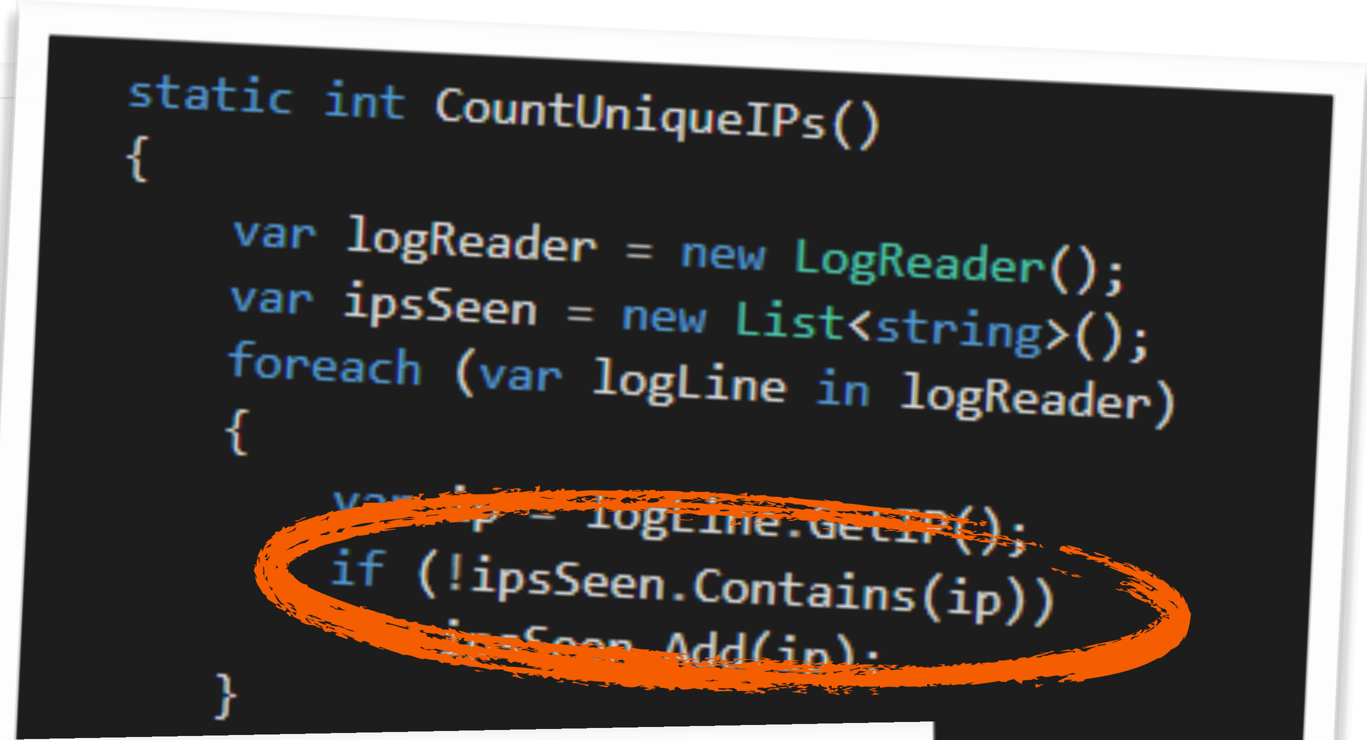
Machine: Lenovo W540
Year: 2015
CPU: 2.7 GHz
RAM: 8 GB
Geekbench 3: 3262

List 36.6 s

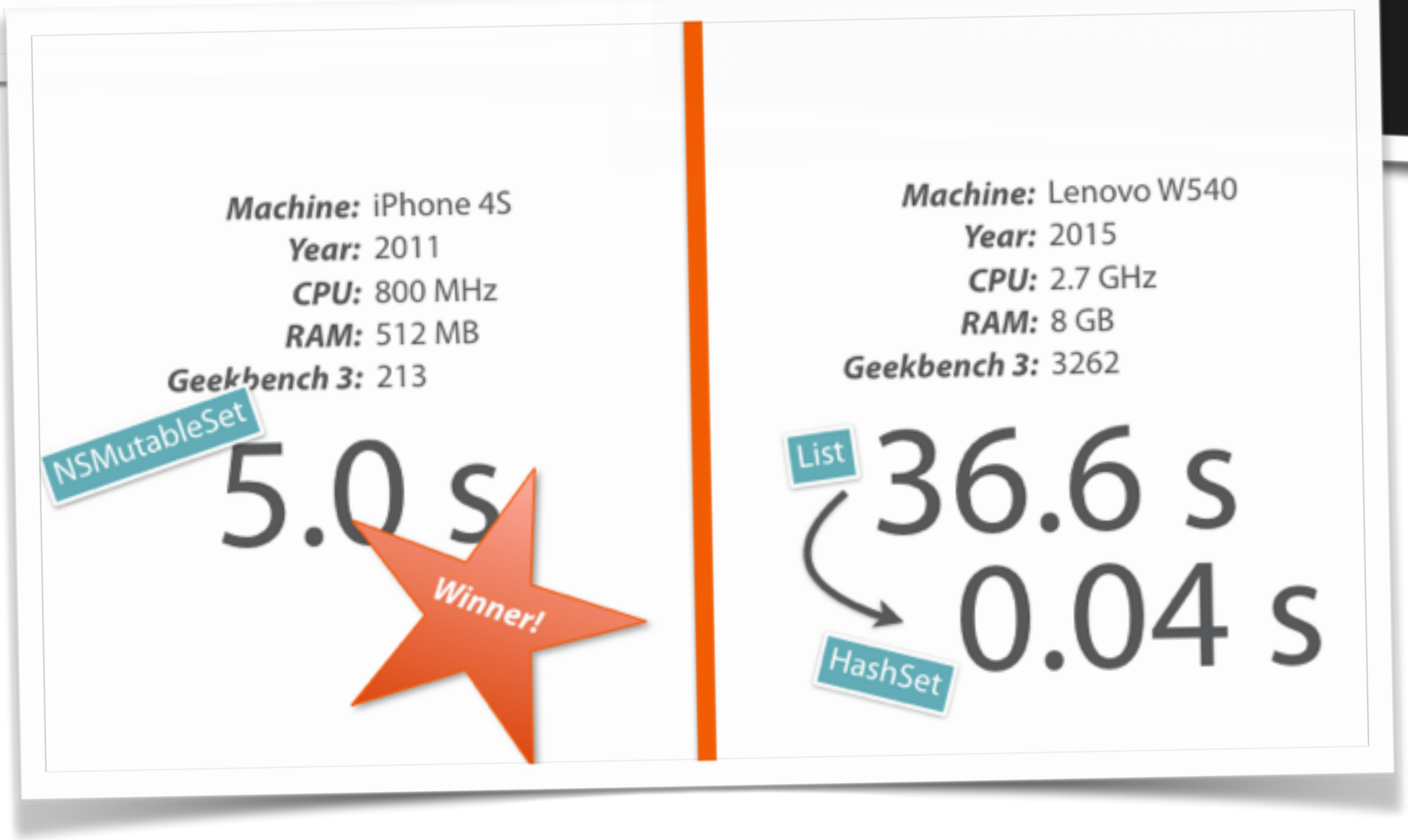
HashSet 0.04 s



```
func countUniqueIPs() -> Int {
    var reader = LogReader();
    var ipsSeen = NSMutableSet();
    for logLine in reader.GetLogLines() {
        var ip = logLine.getIP();
        if(!ipsSeen.containsObject(ip)) {
            ipsSeen.addObject(ip);
        }
    }
    return ipsSeen.count;
}
```



```
static int CountUniqueIPs()
{
    var logReader = new LogReader();
    var ipsSeen = new List<string>();
    foreach (var logLine in logReader)
    {
        var ip = logLine.GetIP();
        if (!ipsSeen.Contains(ip))
            ipsSeen.Add(ip);
    }
```



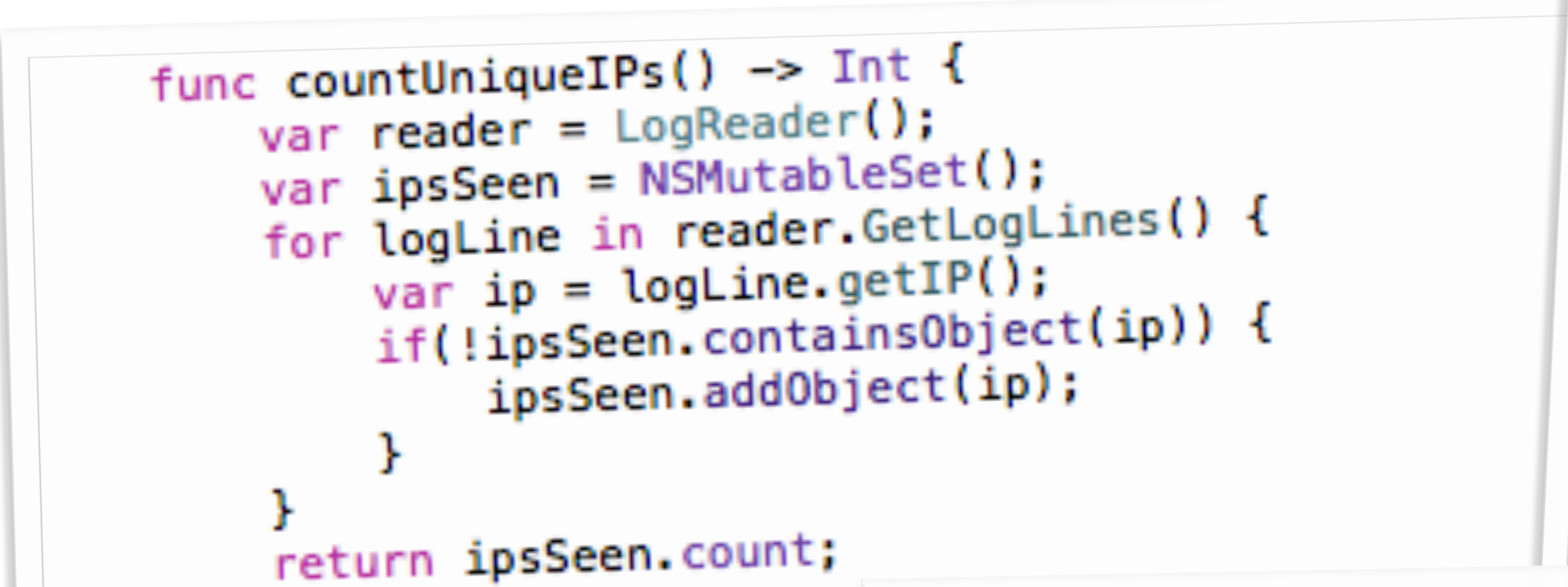
Machine: iPhone 4S
Year: 2011
CPU: 800 MHz
RAM: 512 MB
Geekbench 3: 213

5.0 s

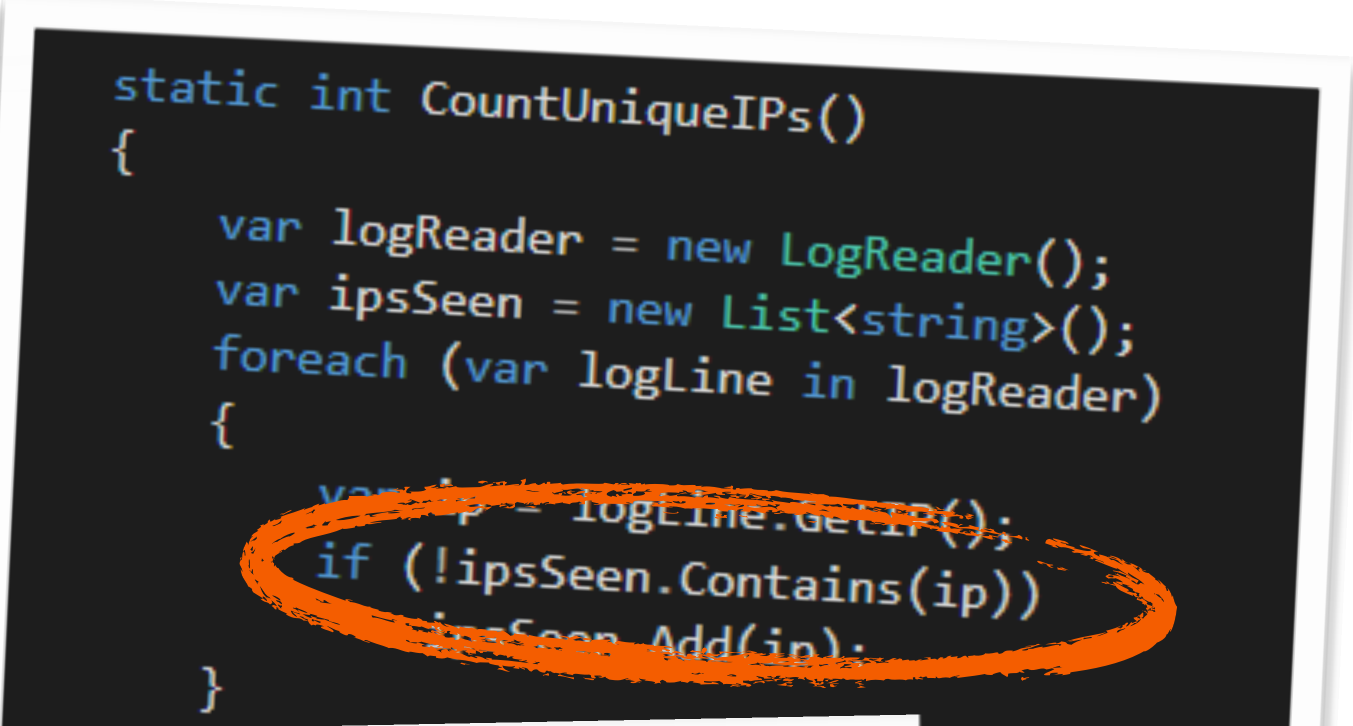
Machine: Lenovo W540
Year: 2015
CPU: 2.7 GHz
RAM: 8 GB
Geekbench 3: 3262

36.6 s

0.04 s

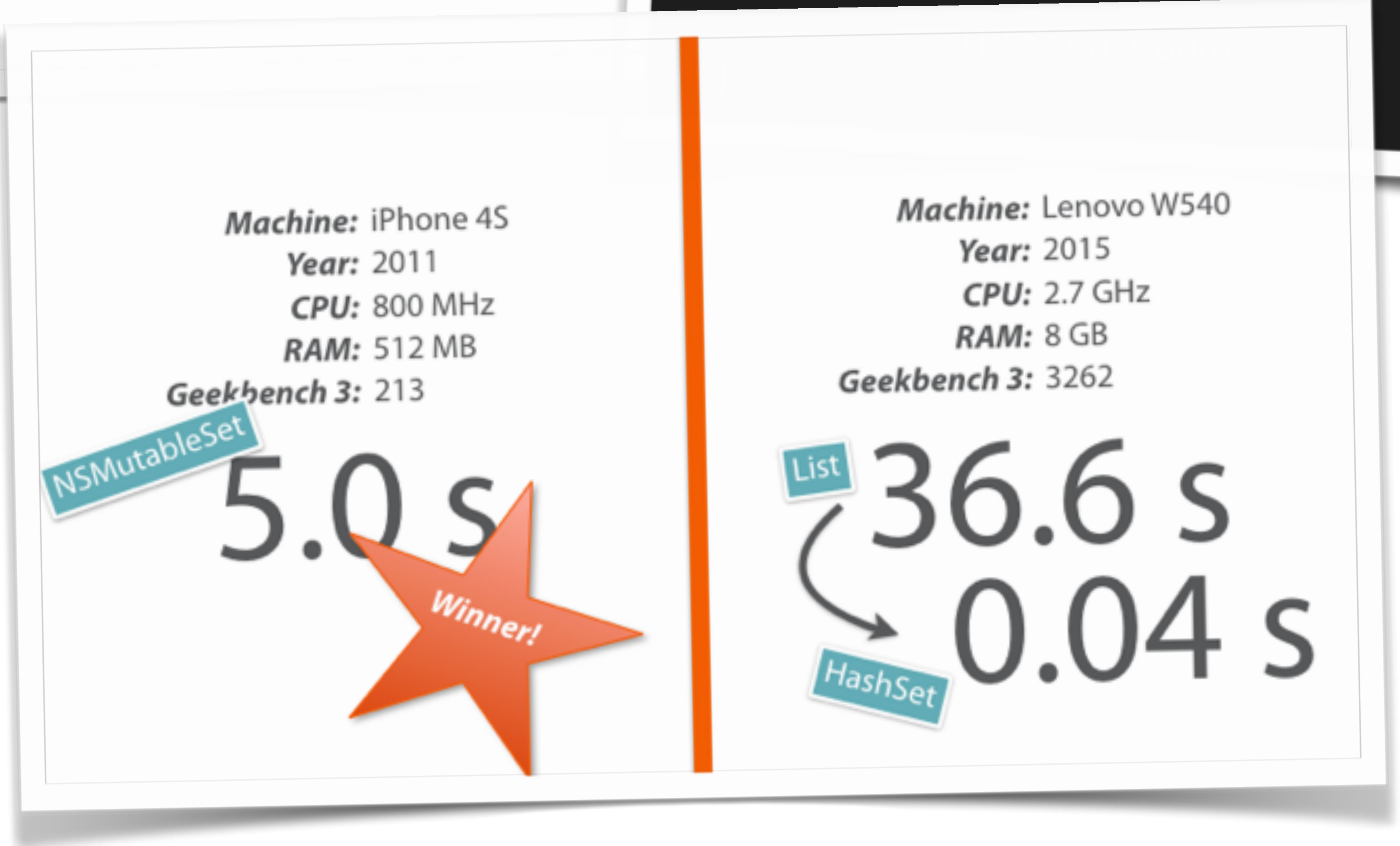


```
func countUniqueIPs() -> Int {
    var reader = LogReader();
    var ipsSeen = NSMutableSet();
    for logLine in reader.GetLogLines() {
        var ip = logLine.getIP();
        if(!ipsSeen.containsObject(ip)) {
            ipsSeen.addObject(ip);
        }
    }
    return ipsSeen.count;
}
```



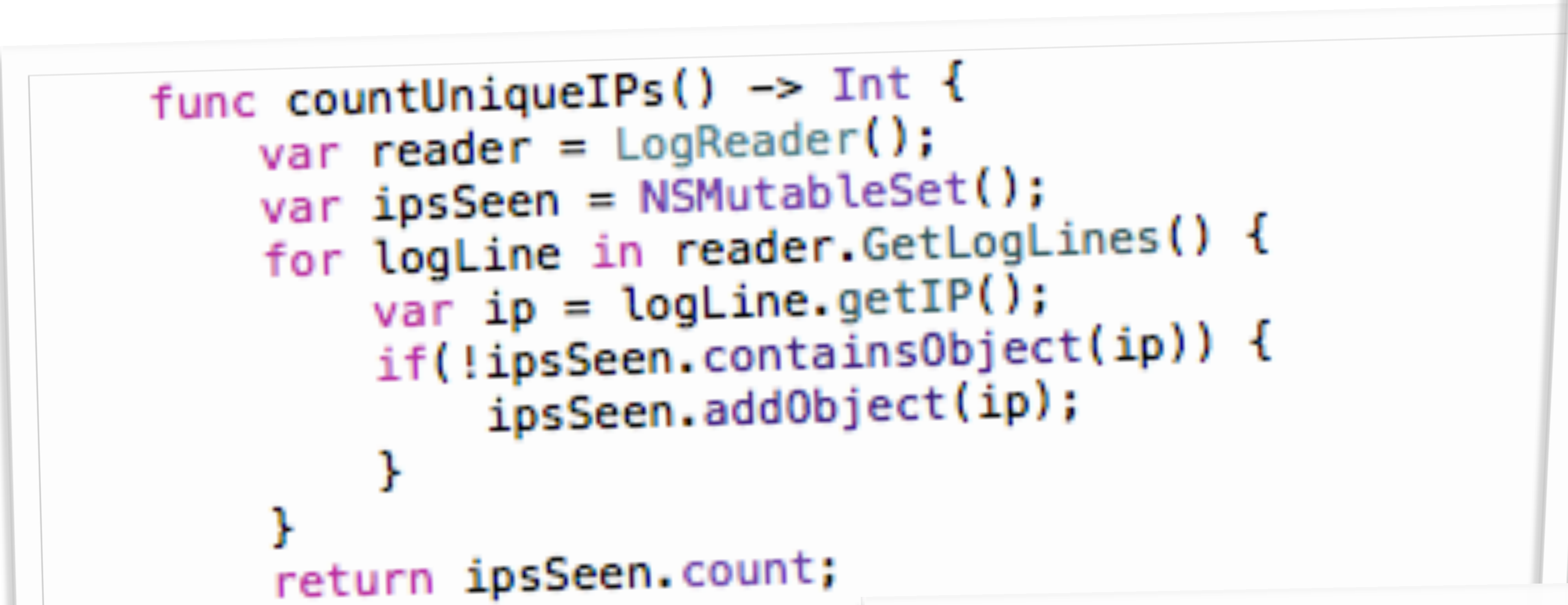
```
static int CountUniqueIPs()
{
    var logReader = new LogReader();
    var ipsSeen = new List<string>();
    foreach (var logLine in logReader)
    {
        var ip = logLine.GetIP();
        if (!ipsSeen.Contains(ip))
            ipsSeen.Add(ip);
    }
```

List.Contains(…): *O*(*N*)

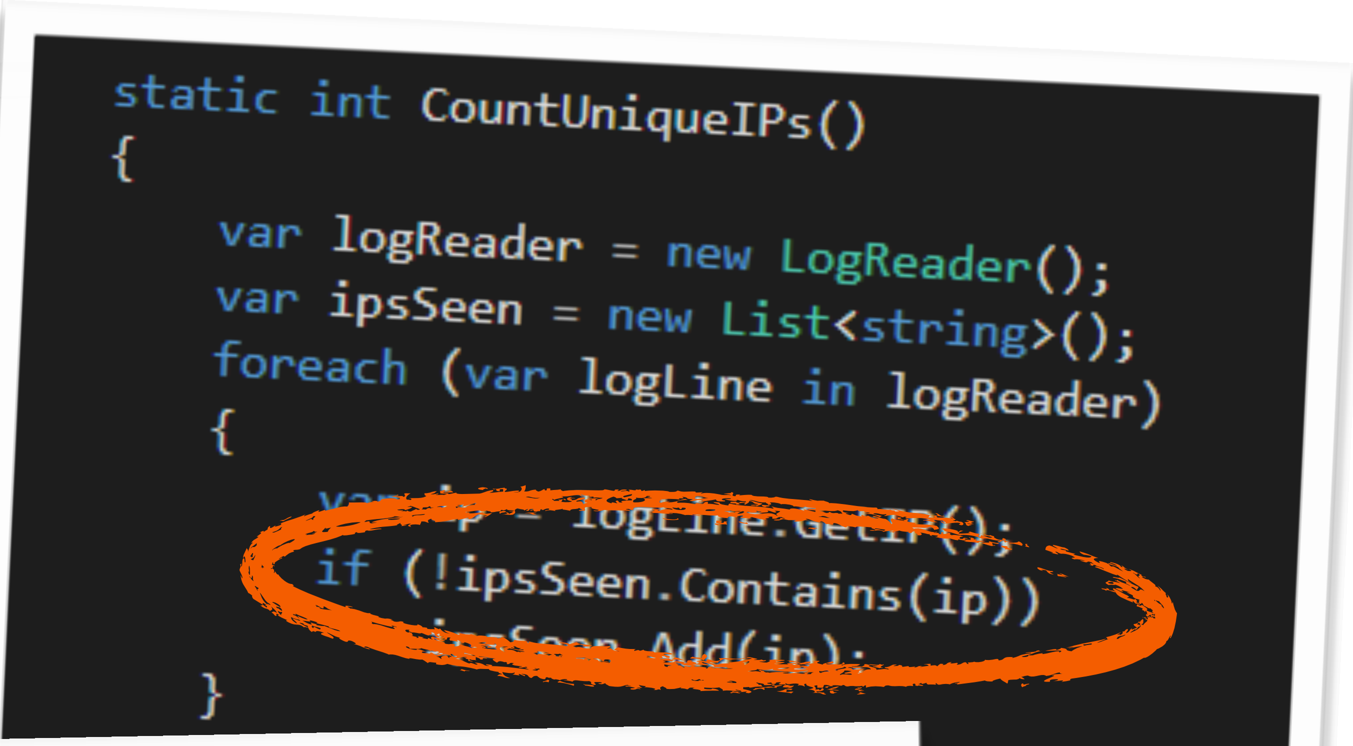


Machine: iPhone 4S
Year: 2011
CPU: 800 MHz
RAM: 512 MB
Geekbench 3: 213

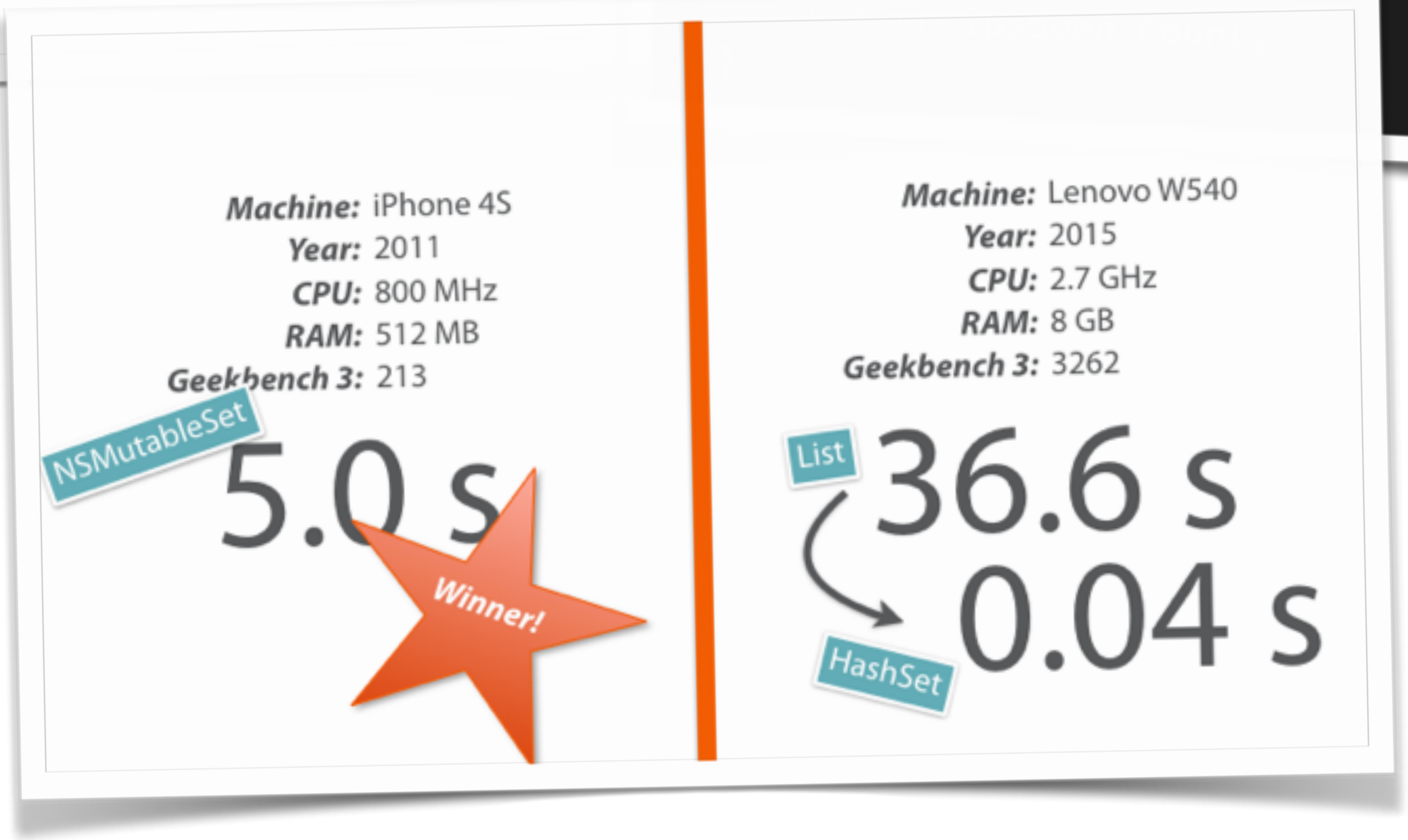
Machine: Lenovo W540
Year: 2015
CPU: 2.7 GHz
RAM: 8 GB
Geekbench 3: 3262



```
func countUniqueIPs() -> Int {
    var reader = LogReader();
    var ipsSeen = NSMutableSet();
    for logLine in reader.GetLogLines() {
        var ip = logLine.getIP();
        if(!ipsSeen.containsObject(ip)) {
            ipsSeen.addObject(ip);
        }
    }
    return ipsSeen.count;
}
```



```
static int CountUniqueIPs()
{
    var logReader = new LogReader();
    var ipsSeen = new List<string>();
    foreach (var logLine in logReader)
    {
        var ip = logLine.GetIP();
        if (!ipsSeen.Contains(ip))
            ipsSeen.Add(ip);
    }
```



Machine: iPhone 4S
Year: 2011
CPU: 800 MHz
RAM: 512 MB
Geekbench 3: 213

5.0 s

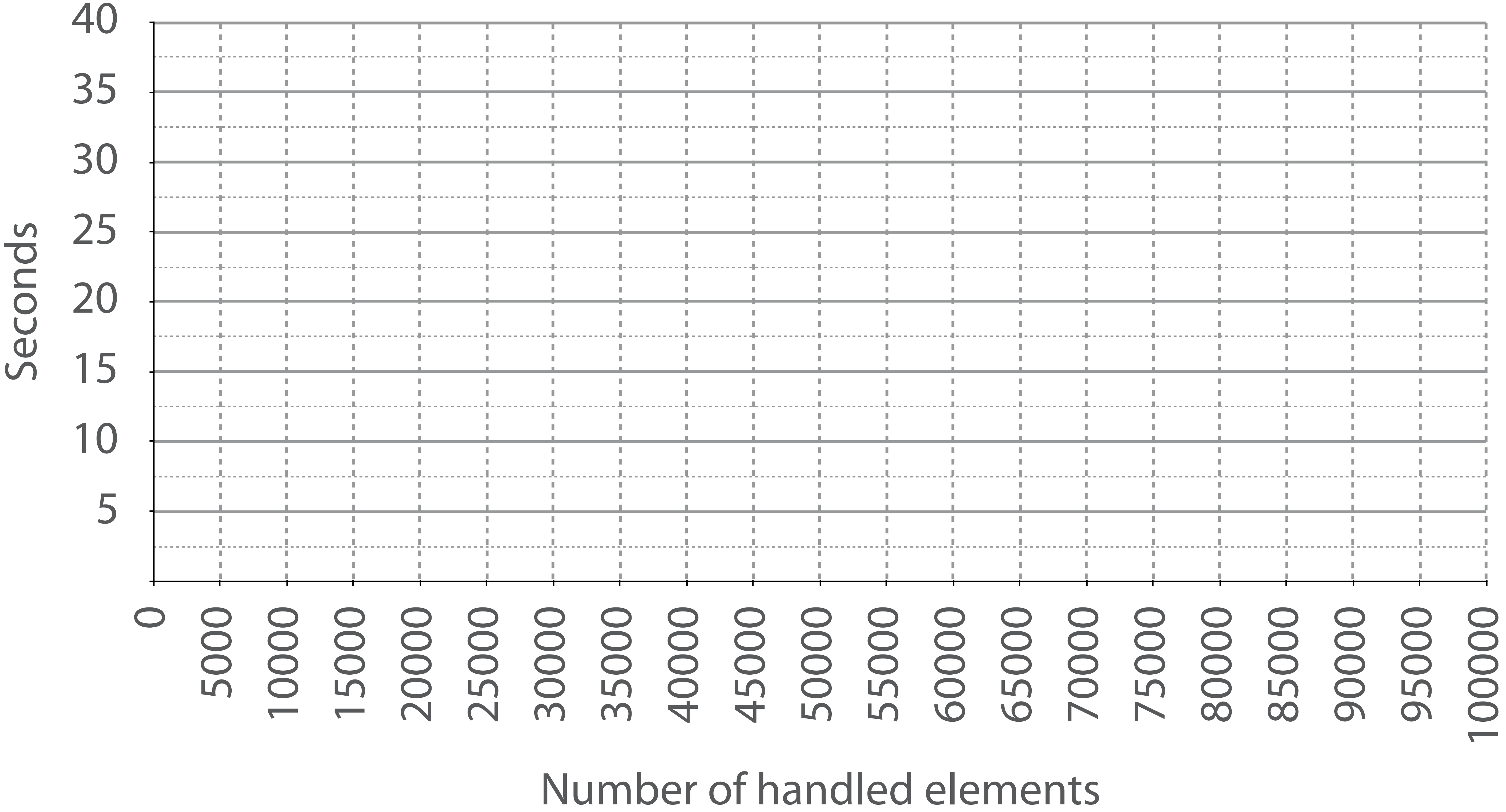
Machine: Lenovo W540
Year: 2015
CPU: 2.7 GHz
RAM: 8 GB
Geekbench 3: 3262

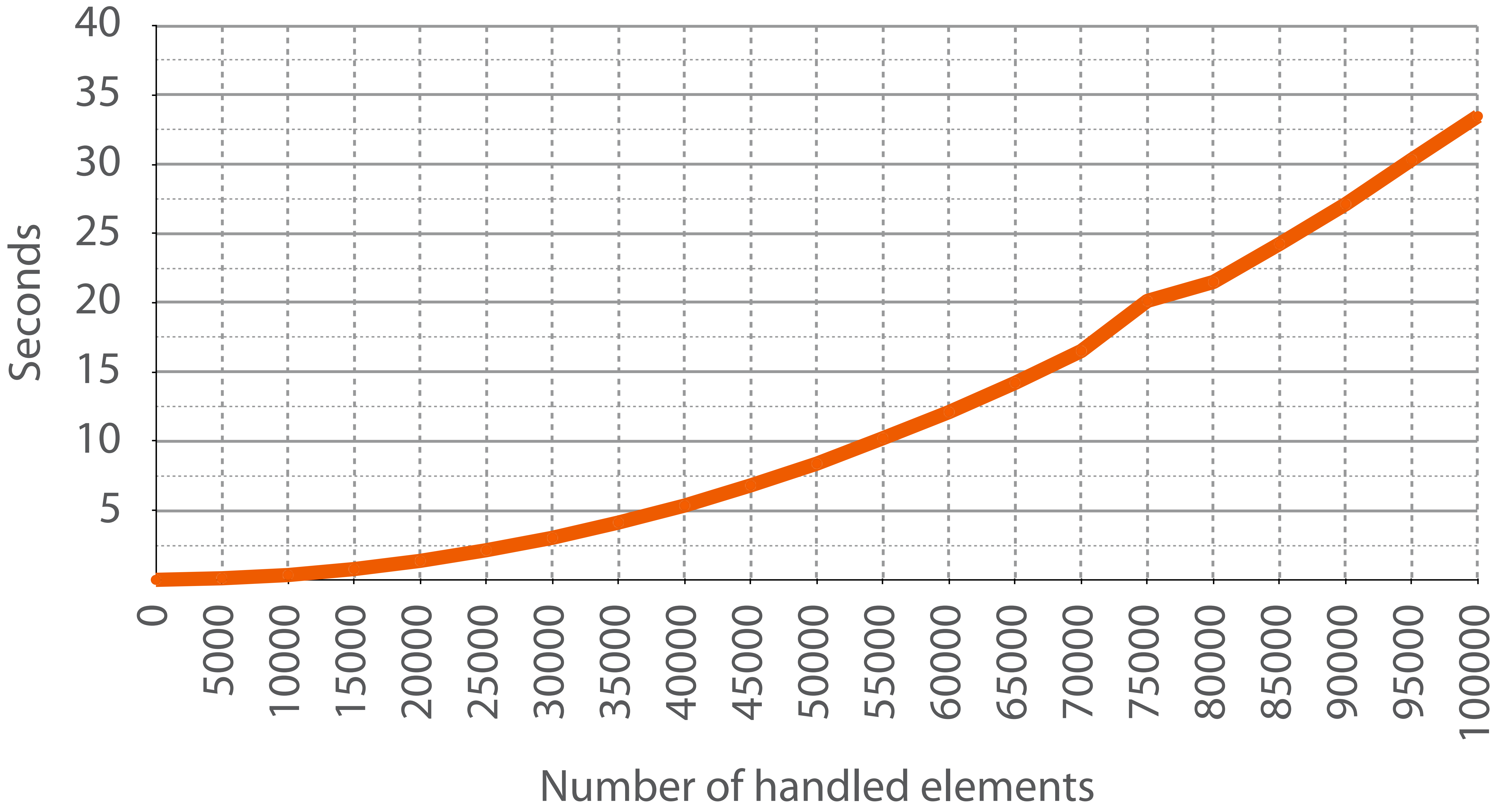
36.6 s

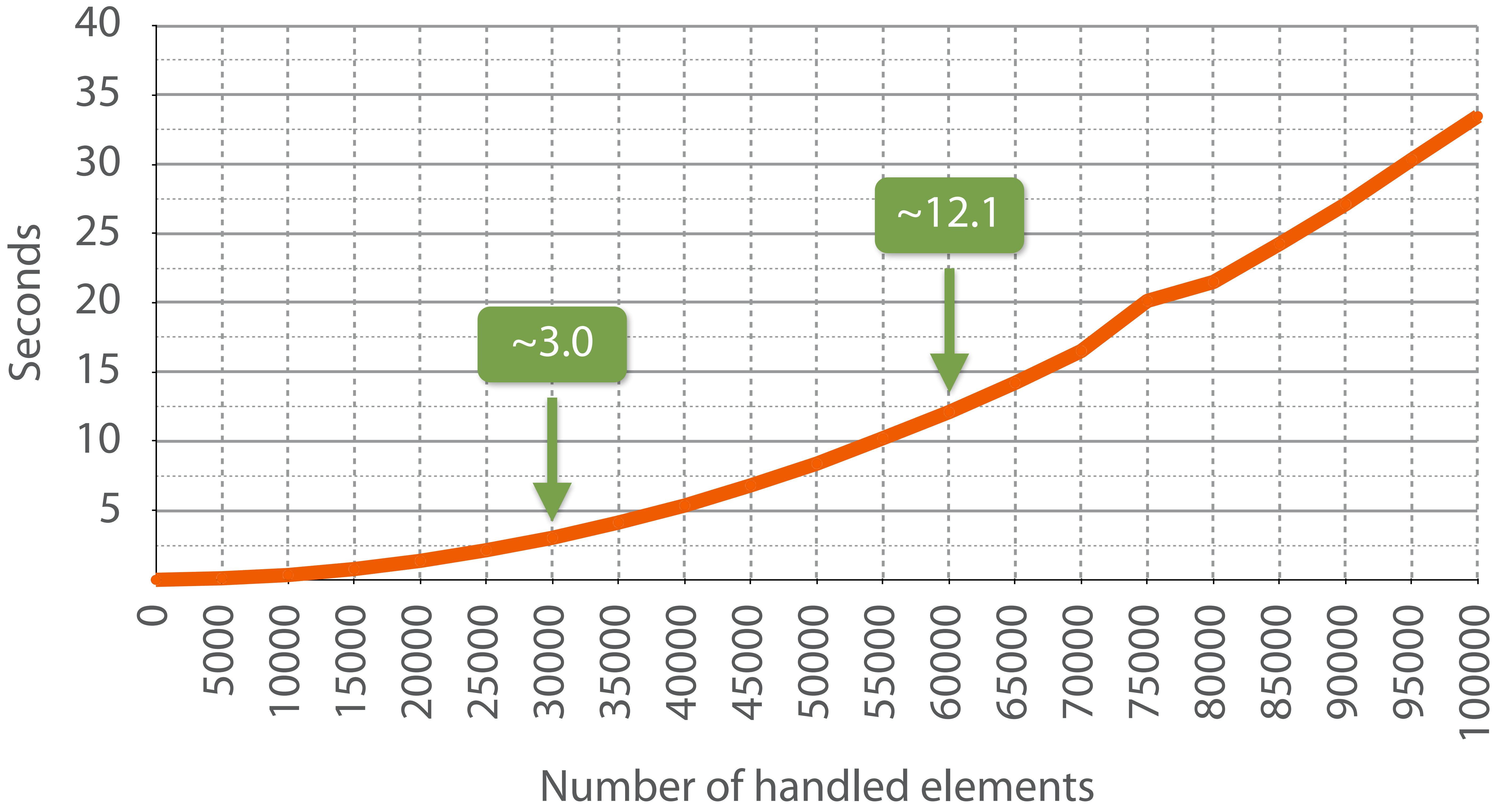
0.04 s

List.Contains(…): *O*(*N*)

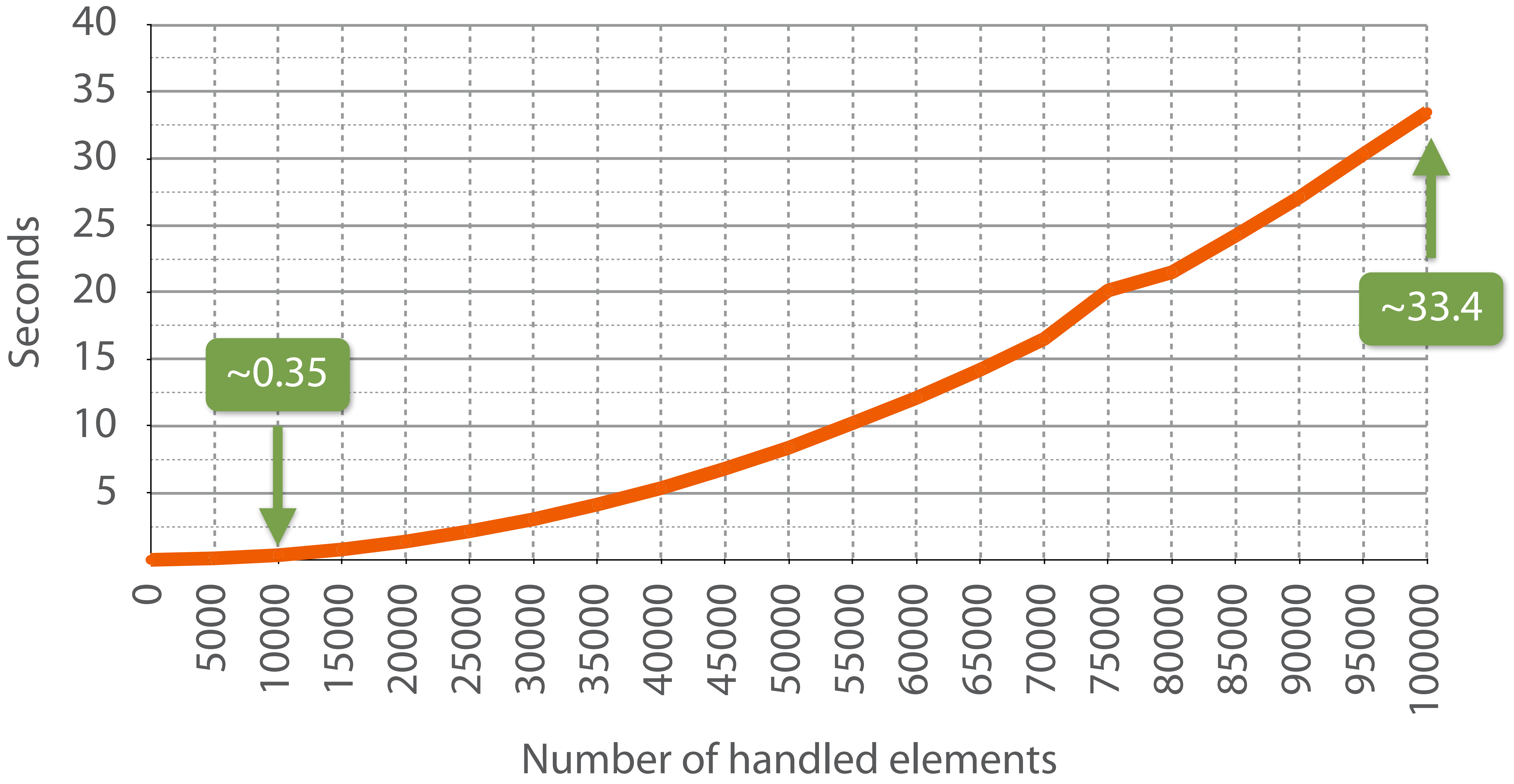
N times: *O*(N^2)


Seconds
40
35
30
25
20
15
10
5
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
500000
550000
600000
650000
700000
750000
800000
850000
900000
950000
1000000
Number of handled elements


40
35
30
25
20
15
10
5
Seconds
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
500000
550000
600000
650000
700000
750000
800000
850000
900000
950000
1000000
Number of handled elements



Seconds
40
35
30
25
20
15
10
5
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
500000
550000
600000
650000
700000
750000
800000
850000
900000
950000
1000000
Number of handled elements
~3.0
~12.1

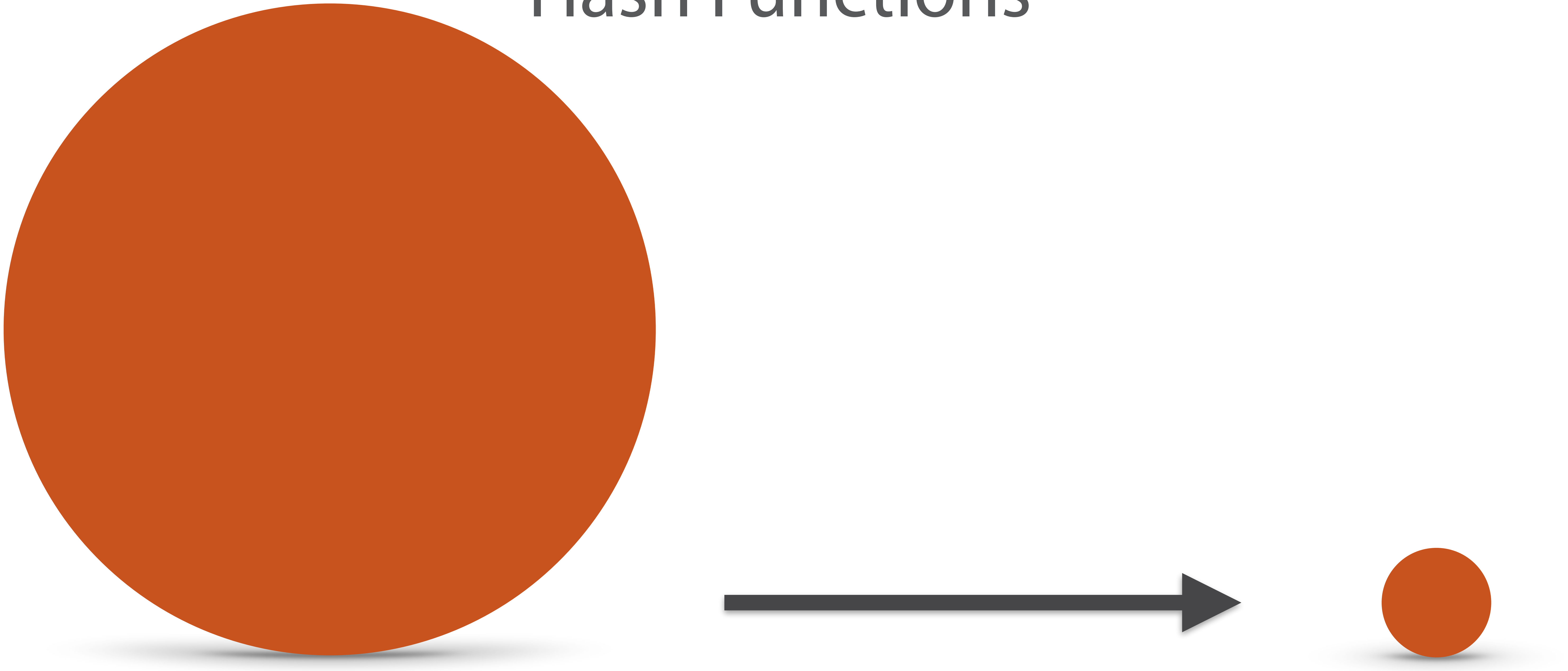

Seconds
40
35
30
25
20
15
10
5
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
500000
550000
600000
650000
700000
750000
800000
850000
900000
950000
1000000
Number of handled elements
~0.35
~33.4

Hash tables to the rescue!

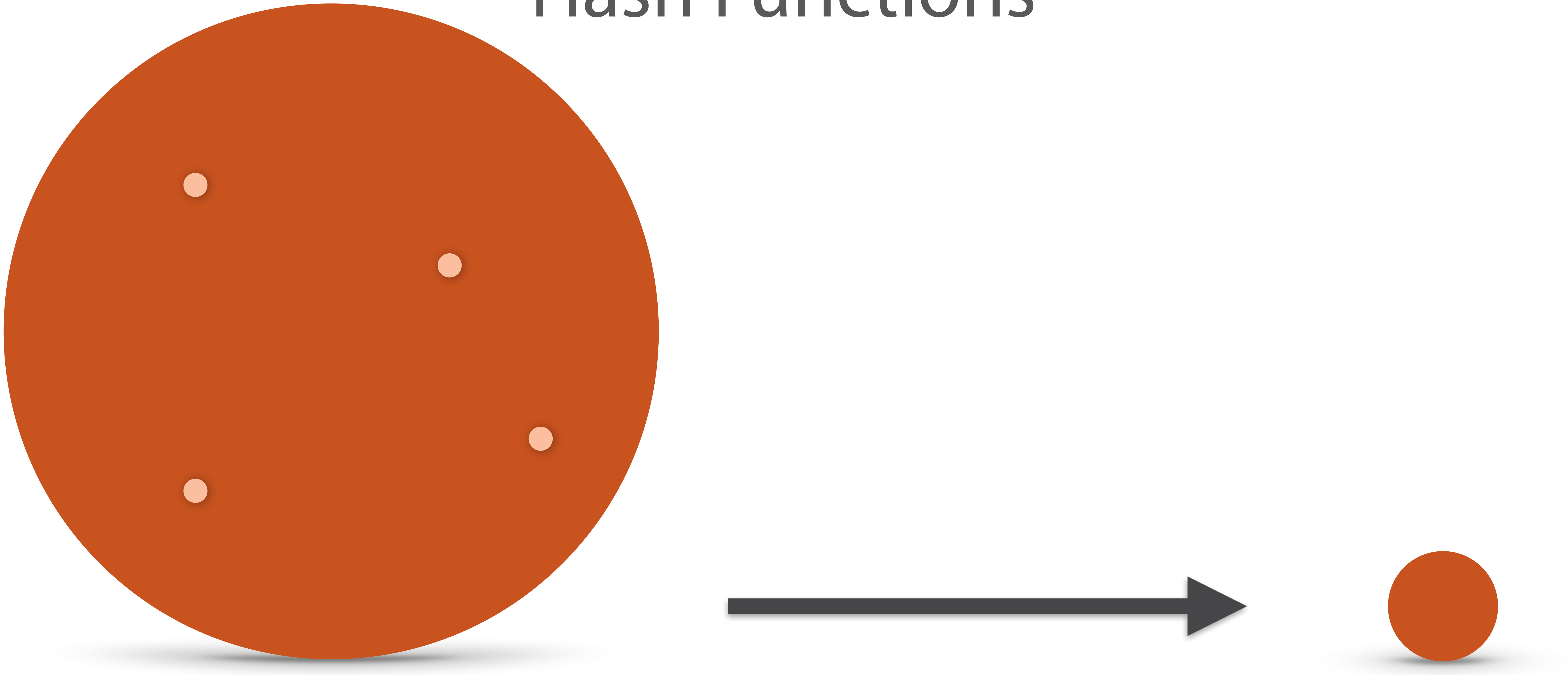
Hash Functions

Hash Functions

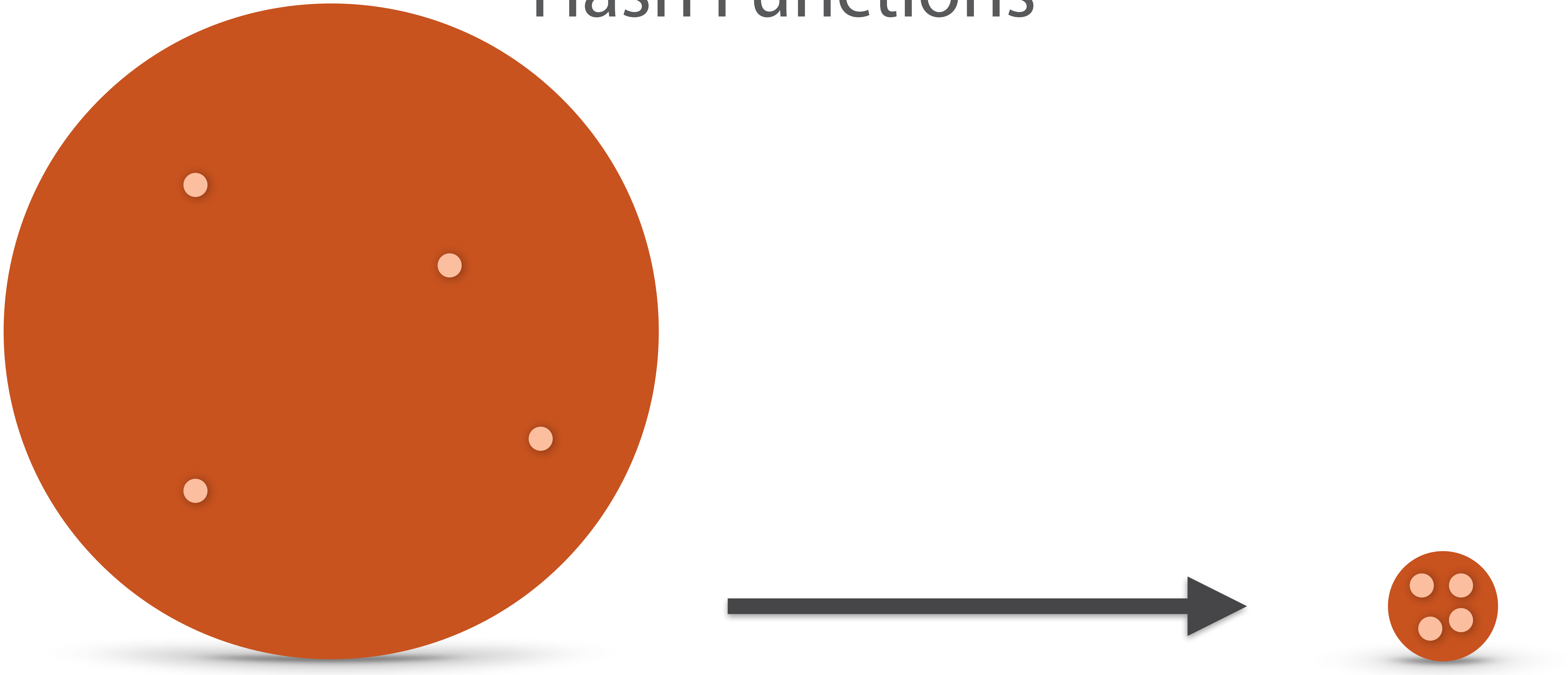
Hash Functions



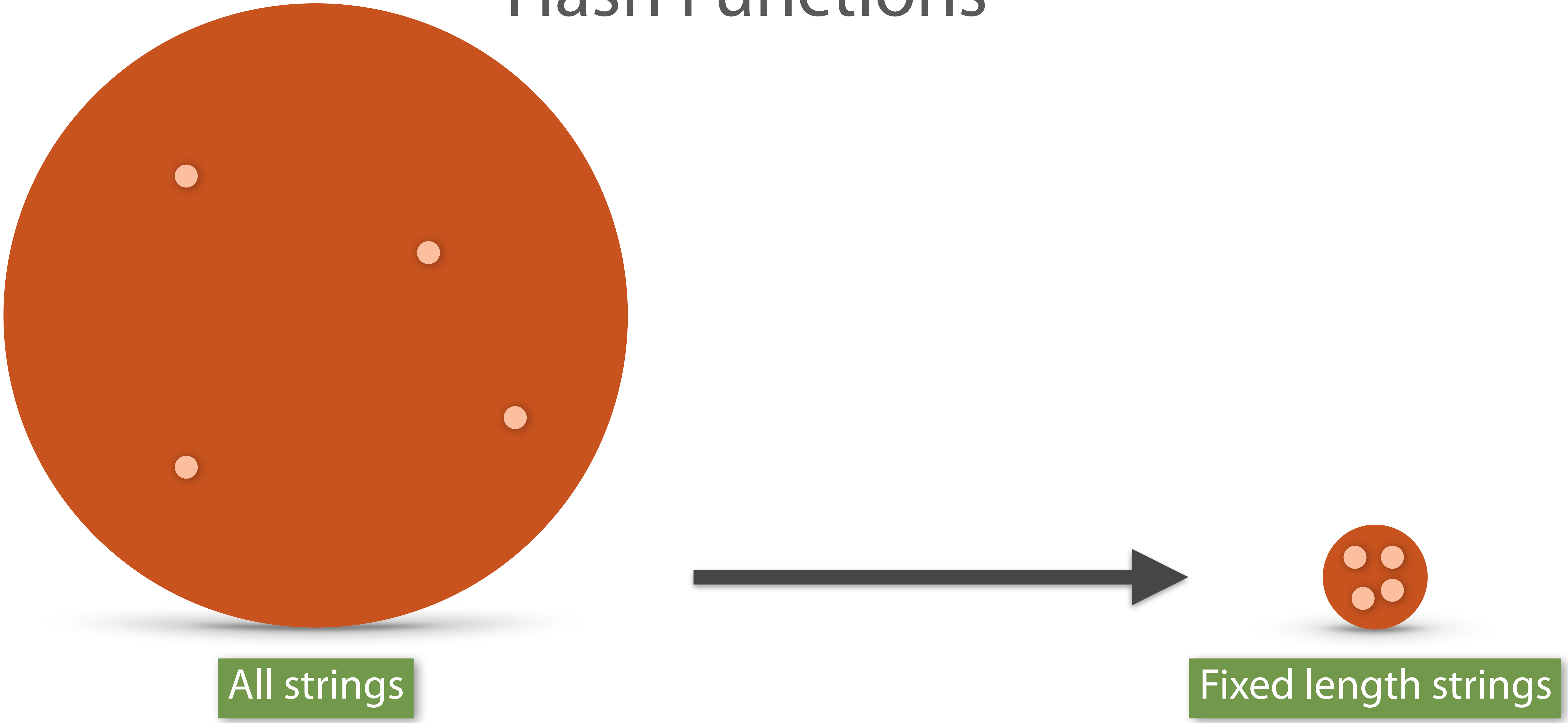
Hash Functions



Hash Functions

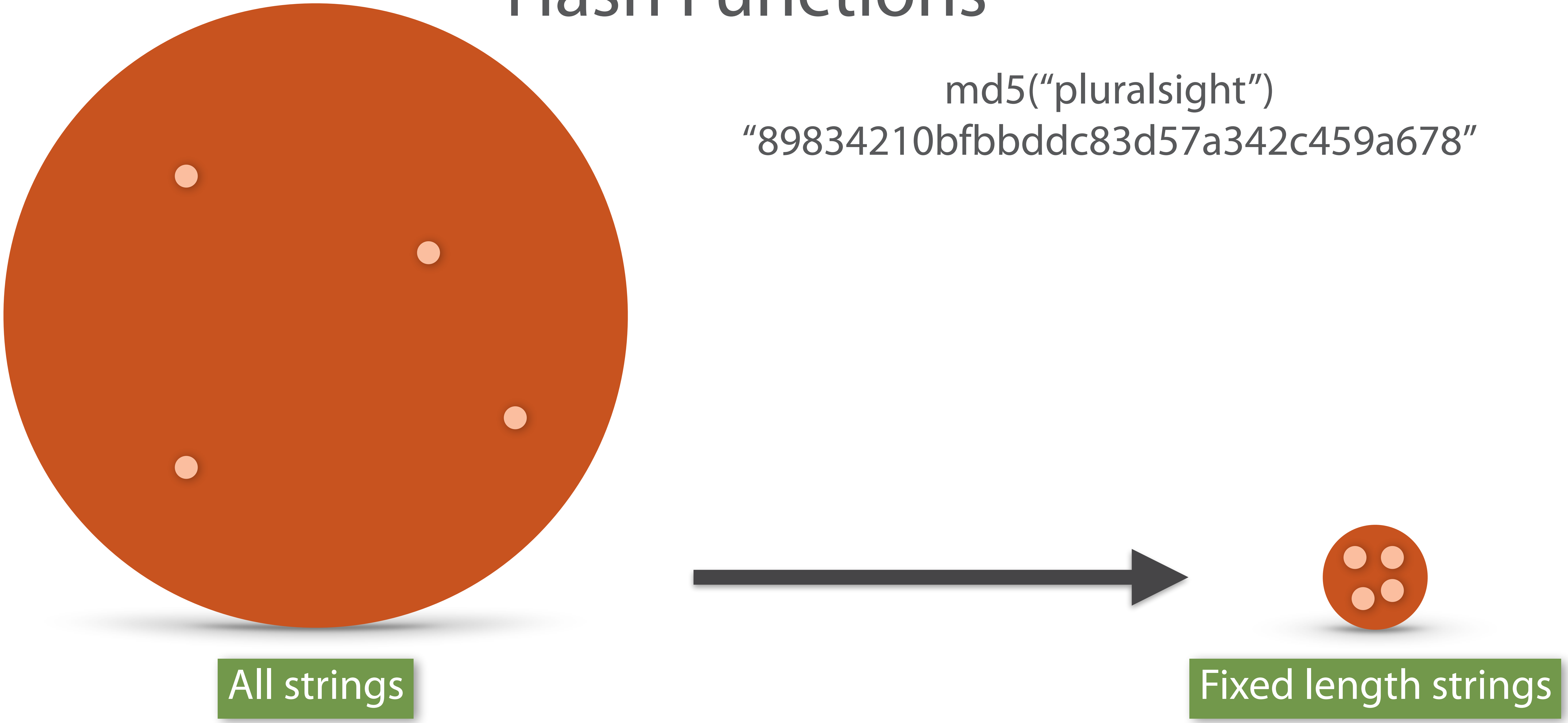


Hash Functions



Hash Functions

md5(“pluralsight”)
“89834210bfbbddc83d57a342c459a678”



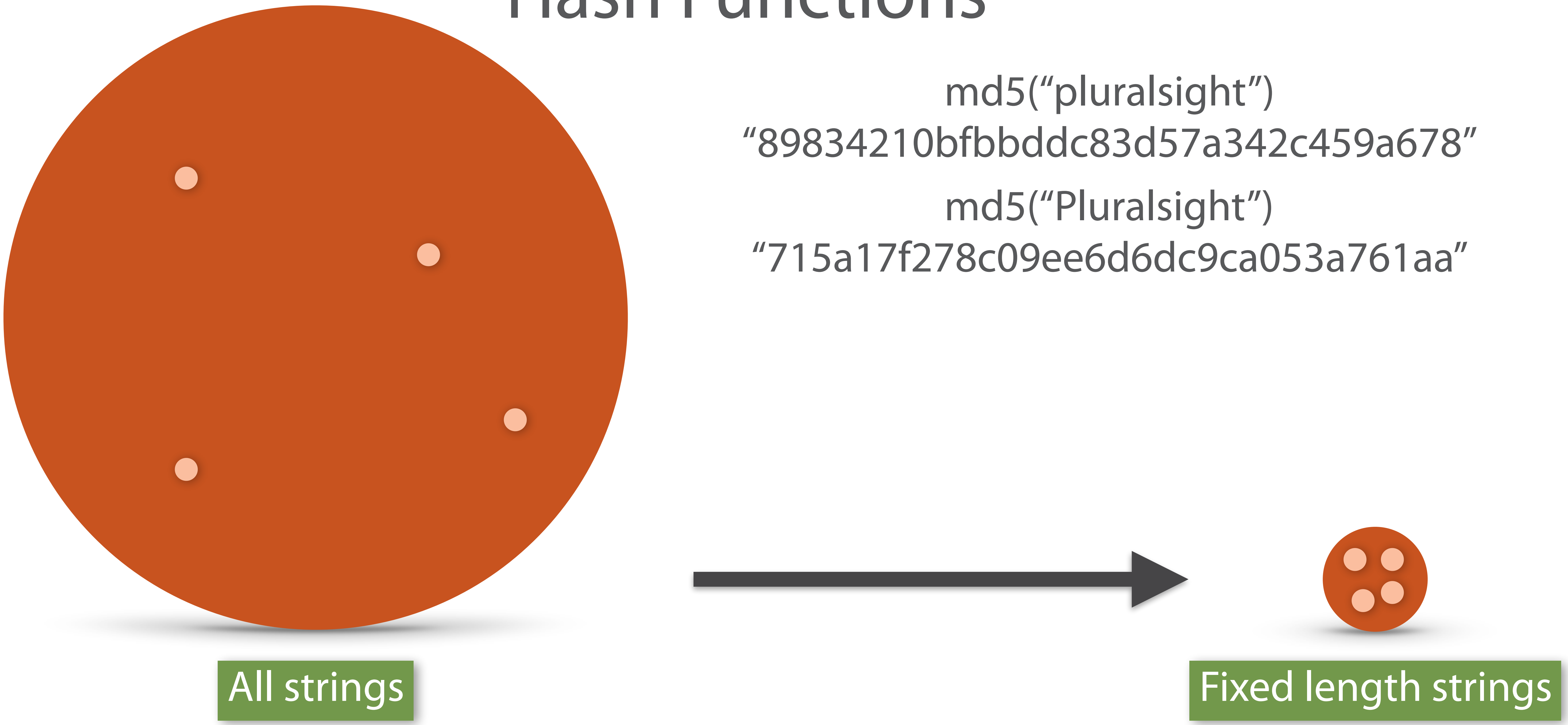
All strings

Fixed length strings

Hash Functions

md5(“pluralsight”)
“89834210bfbbddc83d57a342c459a678”

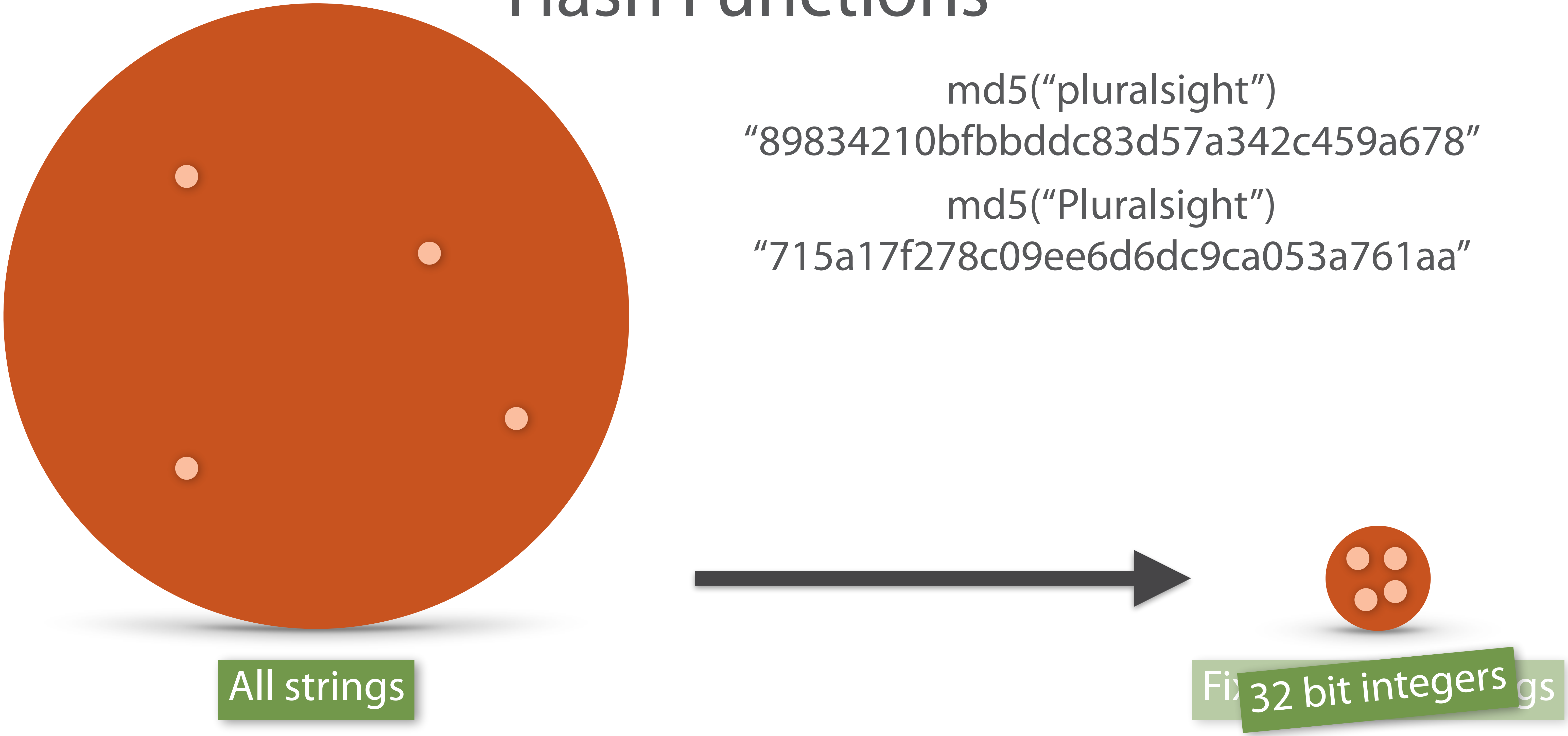
md5(“Pluralsight”)
“715a17f278c09ee6d6dc9ca053a761aa”



Hash Functions

md5(“pluralsight”)
“89834210bfbbddc83d57a342c459a678”

md5(“Pluralsight”)
“715a17f278c09ee6d6dc9ca053a761aa”

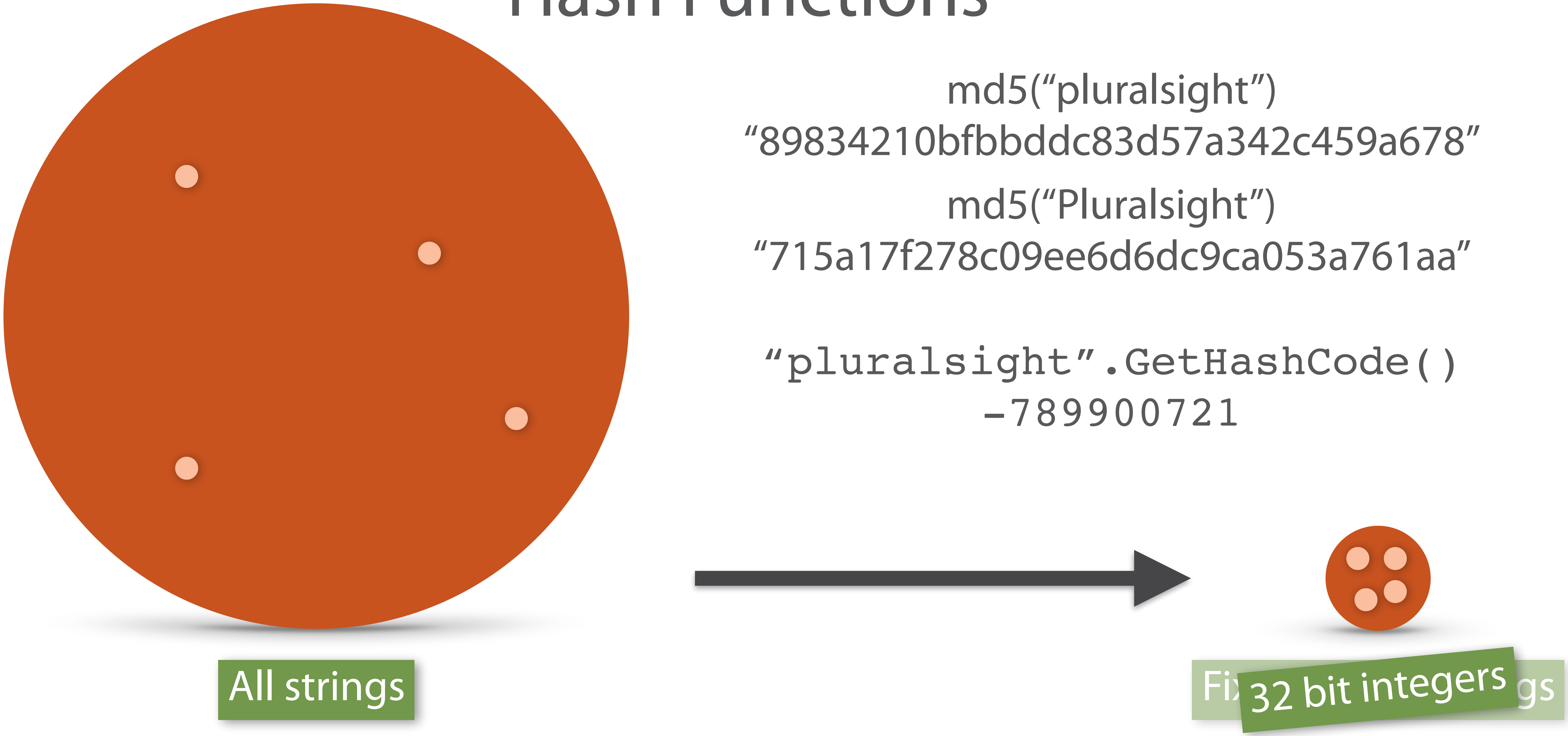


Hash Functions

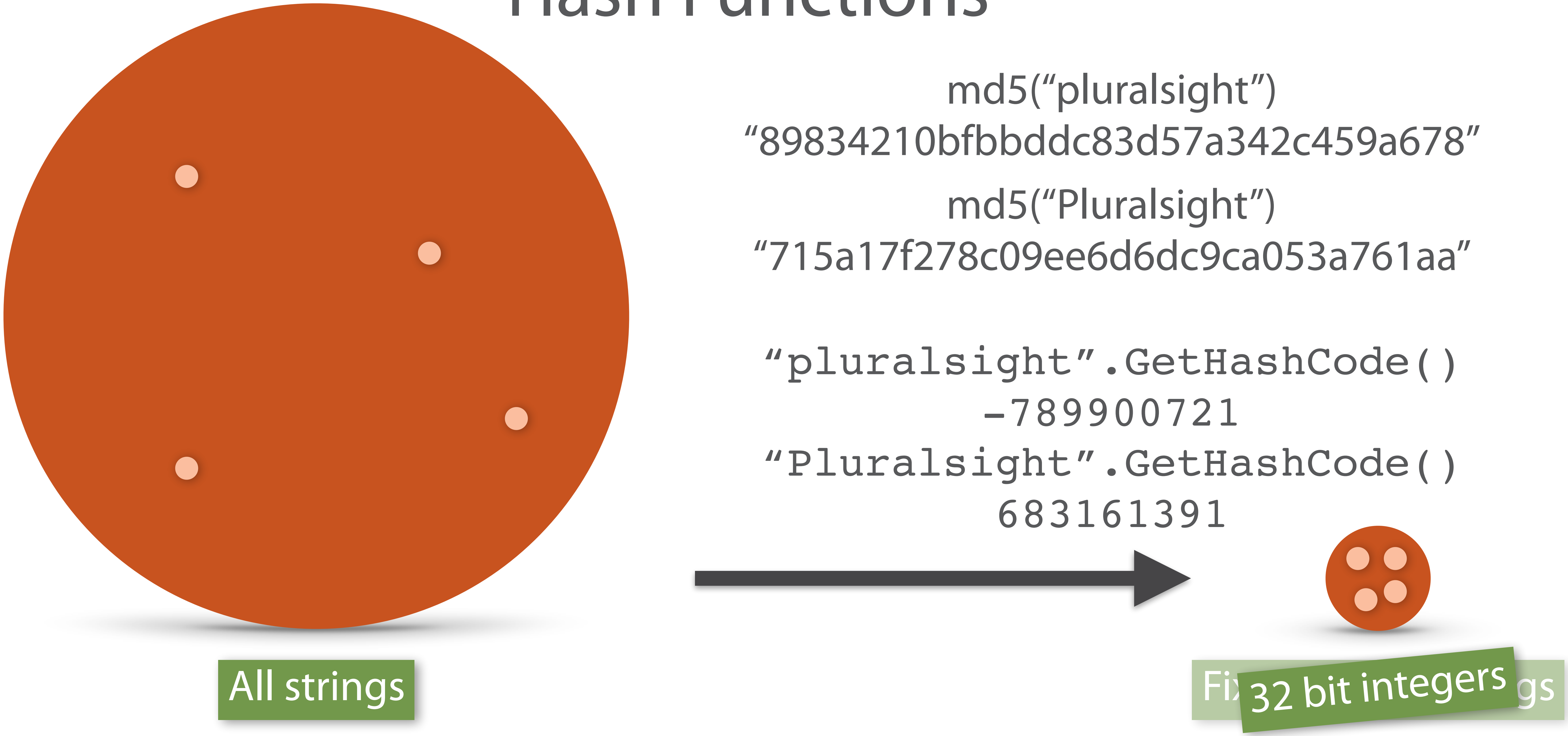
md5(“pluralsight”)
“89834210bfbbddc83d57a342c459a678”

md5(“Pluralsight”)
“715a17f278c09ee6d6dc9ca053a761aa”

```
“pluralsight”.GetHashCode()
-789900721
```



Hash Functions

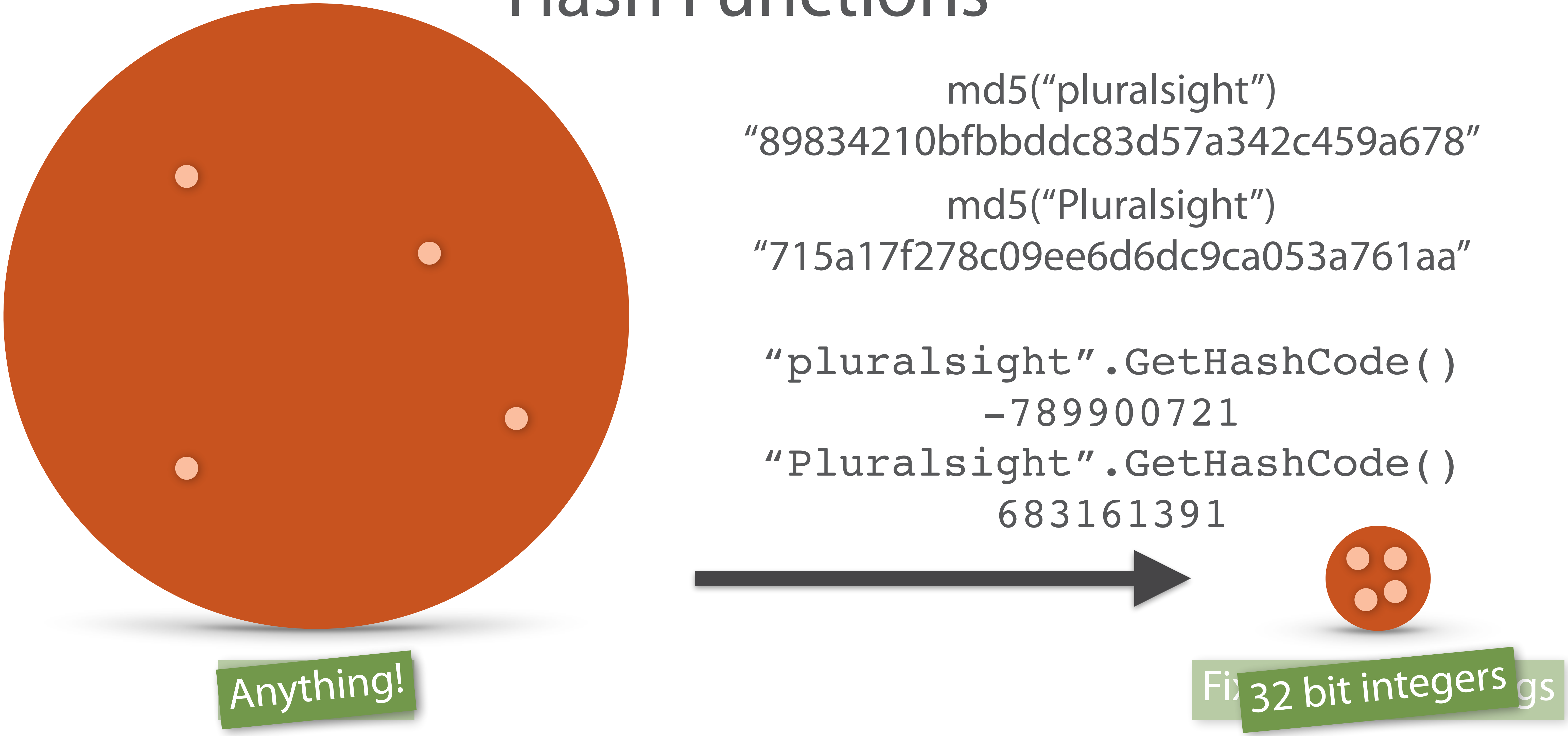


md5(“pluralsight”)
“89834210bfbbddc83d57a342c459a678”

md5(“Pluralsight”)
“715a17f278c09ee6d6dc9ca053a761aa”

```
“pluralsight”.GetHashCode()
-789900721
“Pluralsight”.GetHashCode()
683161391
```

Hash Functions



md5("pluralsight")
"89834210bfbbddc83d57a342c459a678"

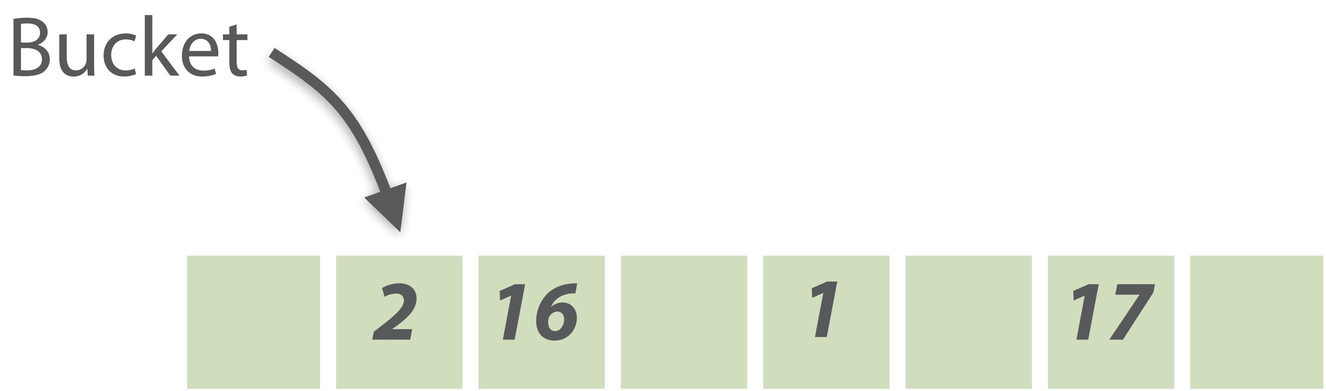
md5("Pluralsight")
"715a17f278c09ee6d6dc9ca053a761aa"

```
"pluralsight".GetHashCode()
-789900721
"Pluralsight".GetHashCode()
683161391
```

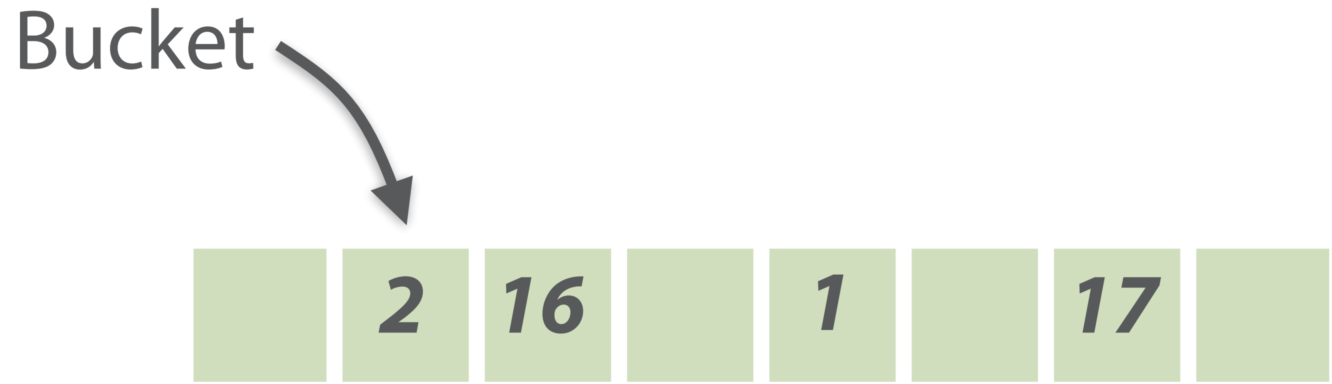
Hash Tables

	2	*16*		*1*		*17*	

Hash Tables



Hash Tables

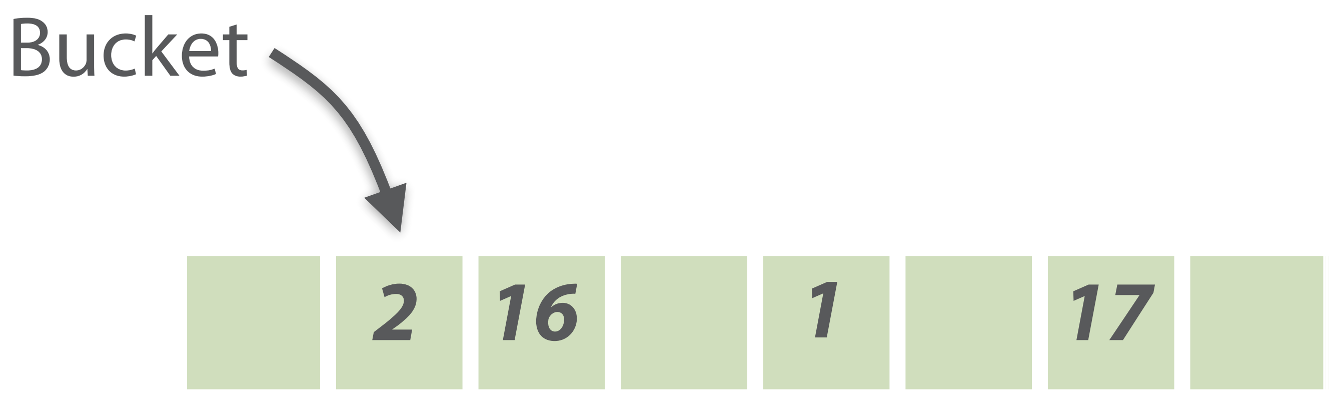


Bucket

	2	*16*		*1*		*17*	

Element to insert: ***25***

Hash Tables



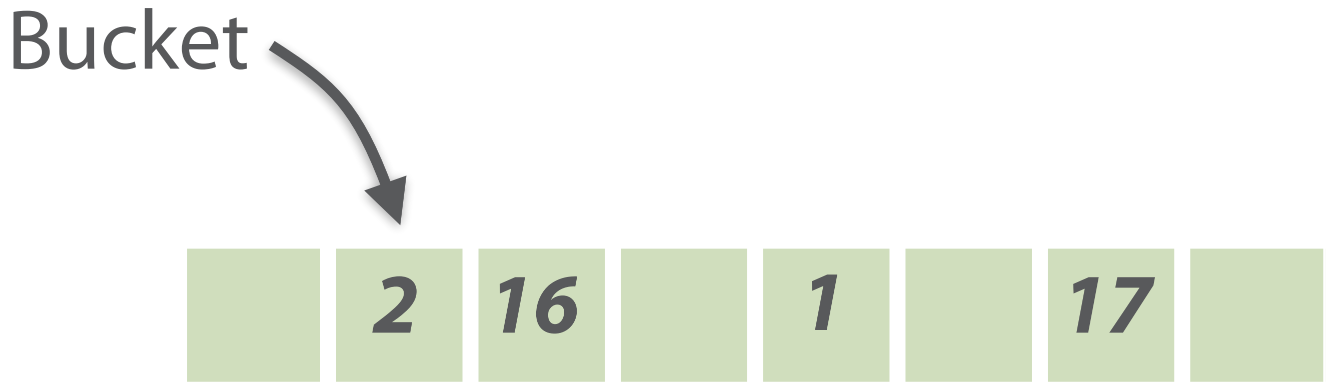
Bucket

	2	***16***		***1***		***17***	

Element to insert: ***25***

h(25) =

Hash Tables

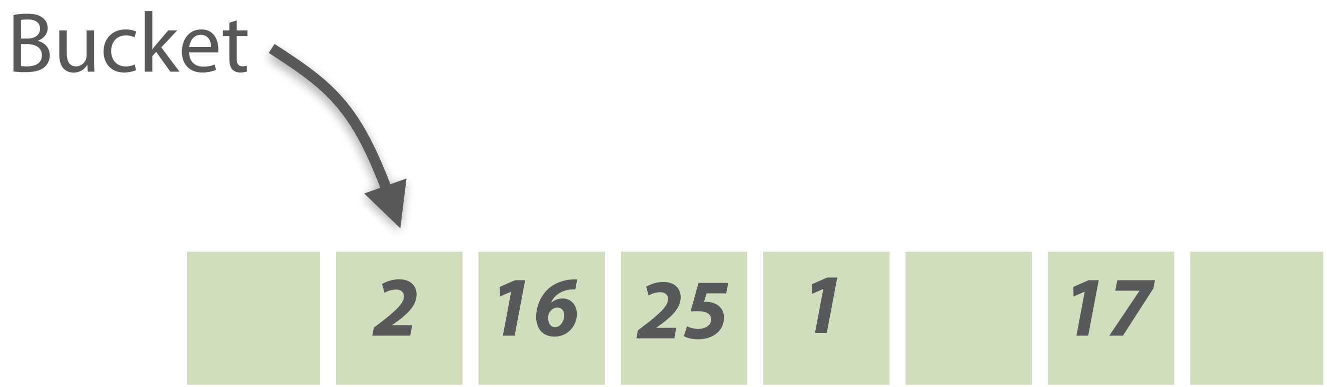


	2	*16*		*1*		*17*	

Element to insert: ***25***

h(25) = 3

Hash Tables



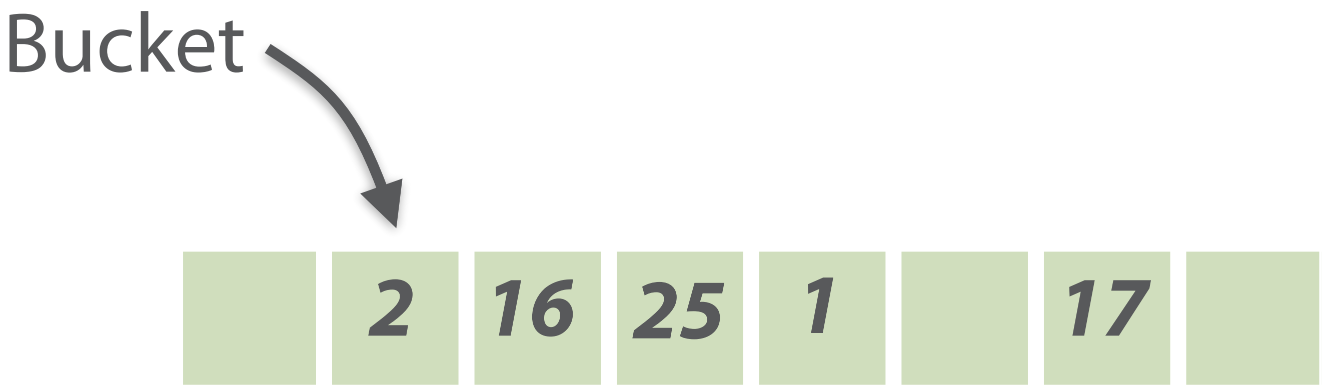
Bucket

	2	16	25	1		17	

Element to insert:

h(25) = 3

Hash Tables

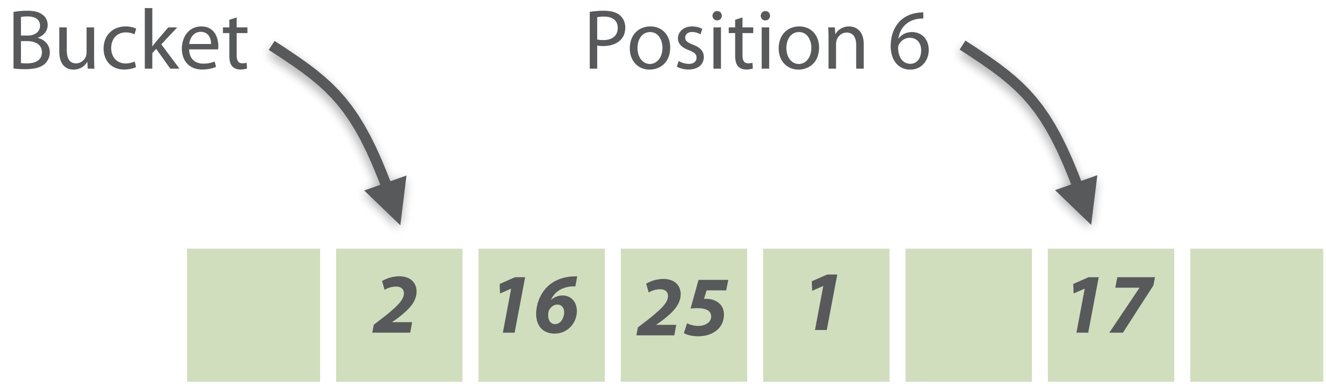


Element to insert:

h(25) = 3

h(23) = 6

Hash Tables



Bucket

Position 6

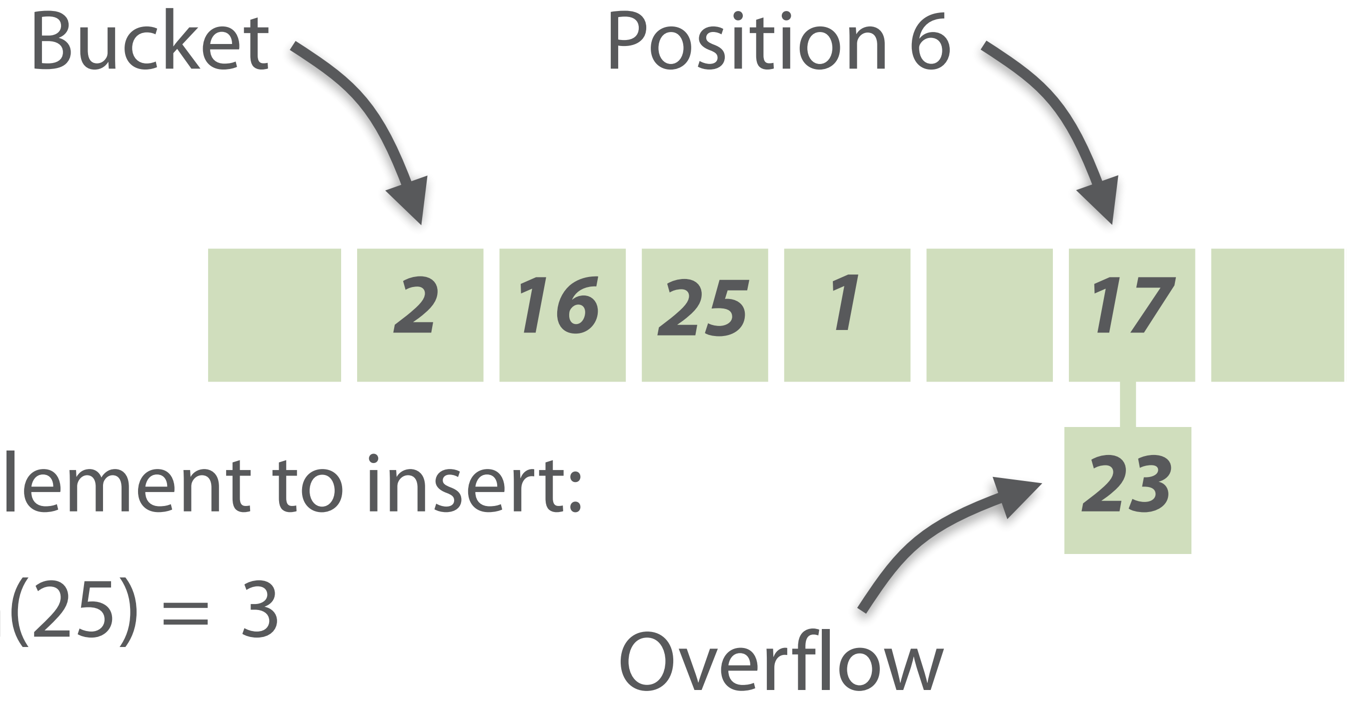
	2	*16*	*25*	*1*		*17*	

Element to insert:

h(25) = 3

h(23) = 6

Hash Tables



Bucket

Position 6

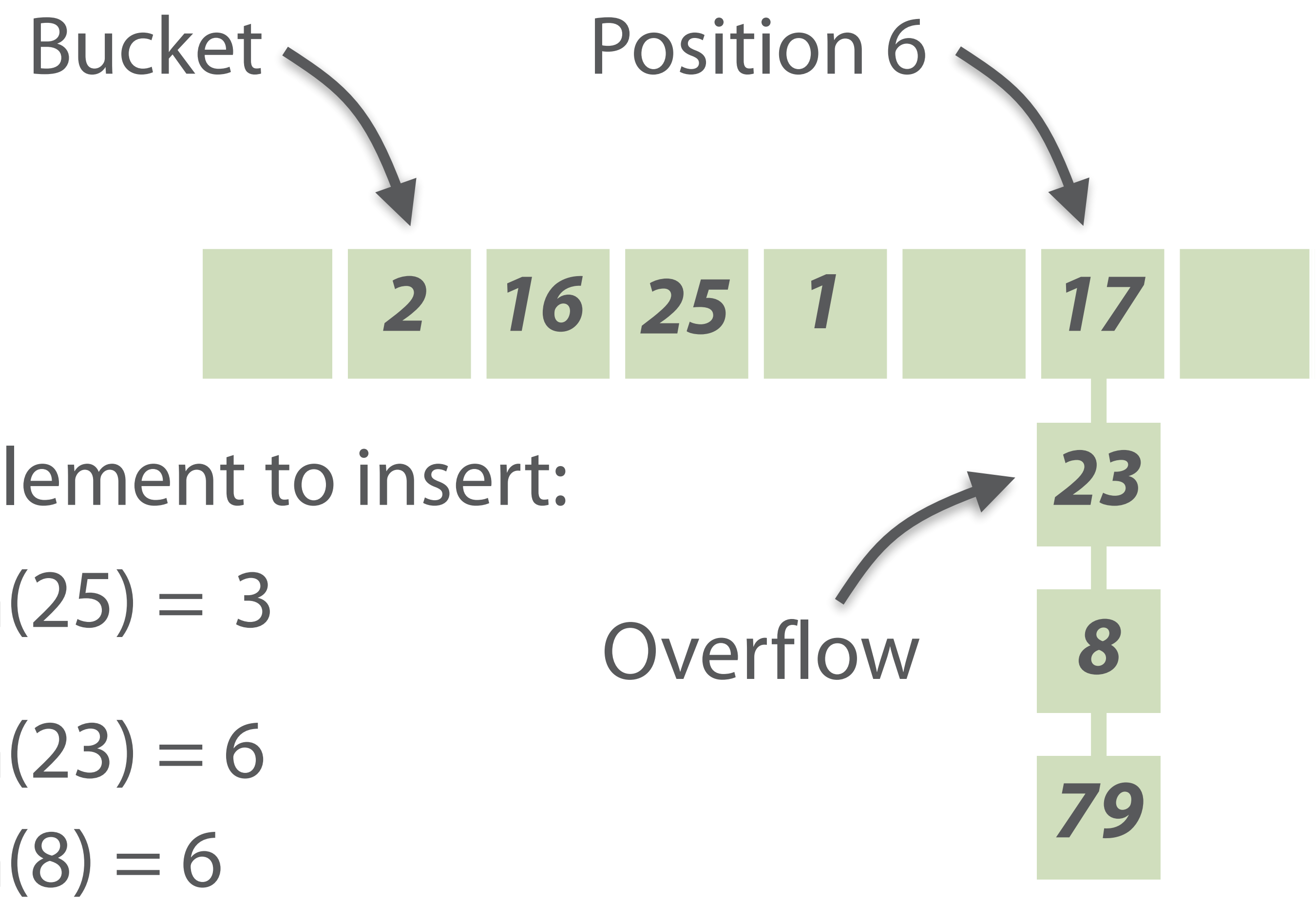
Overflow

Element to insert:

h(25) = 3

h(23) = 6

Hash Tables



Element to insert:

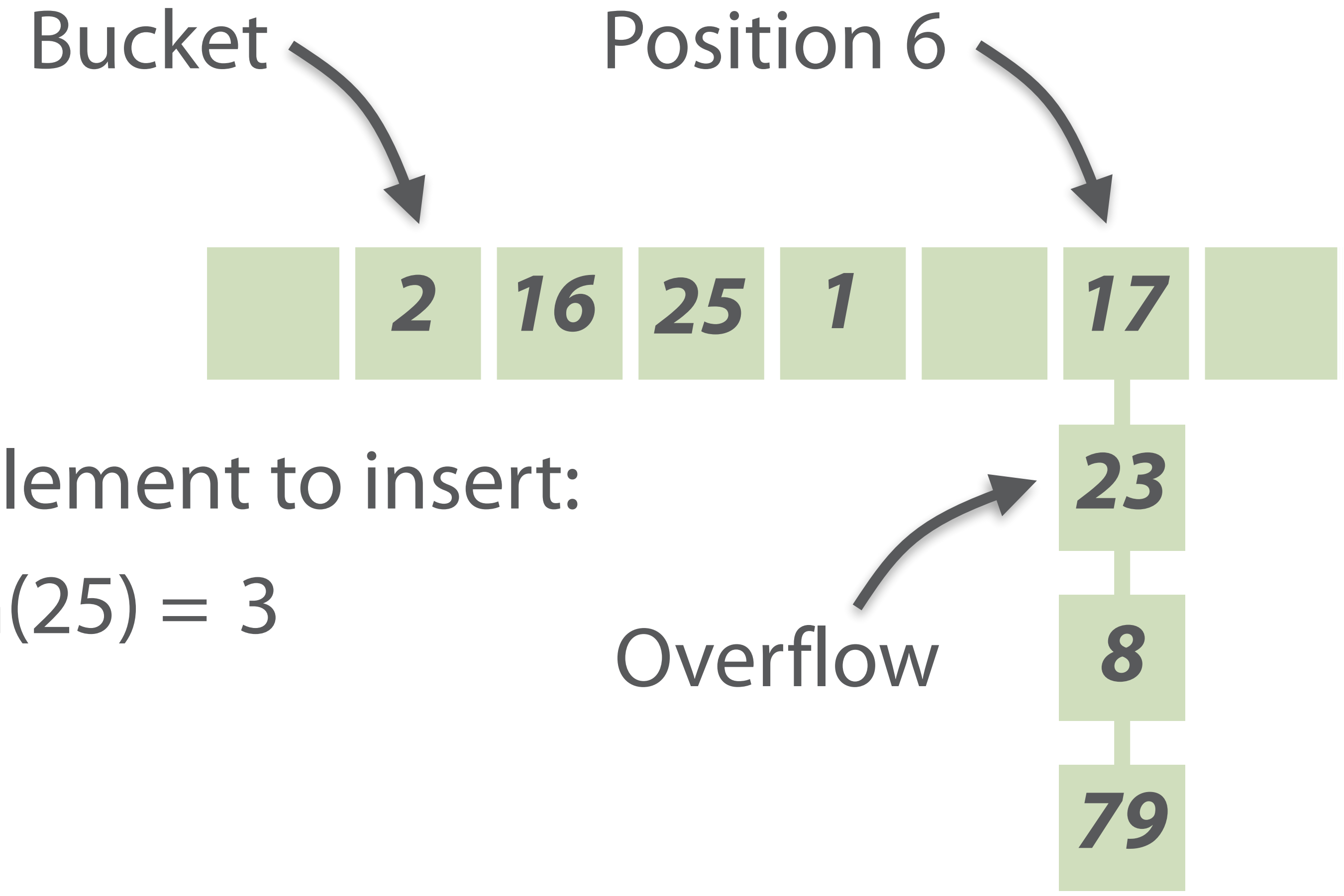
h(25) = 3

h(23) = 6

h(8) = 6

h(79) = 6

Hash Tables


Bucket

Position 6

2 | 16 | 25 | 1 | | 17 |

23

8

79

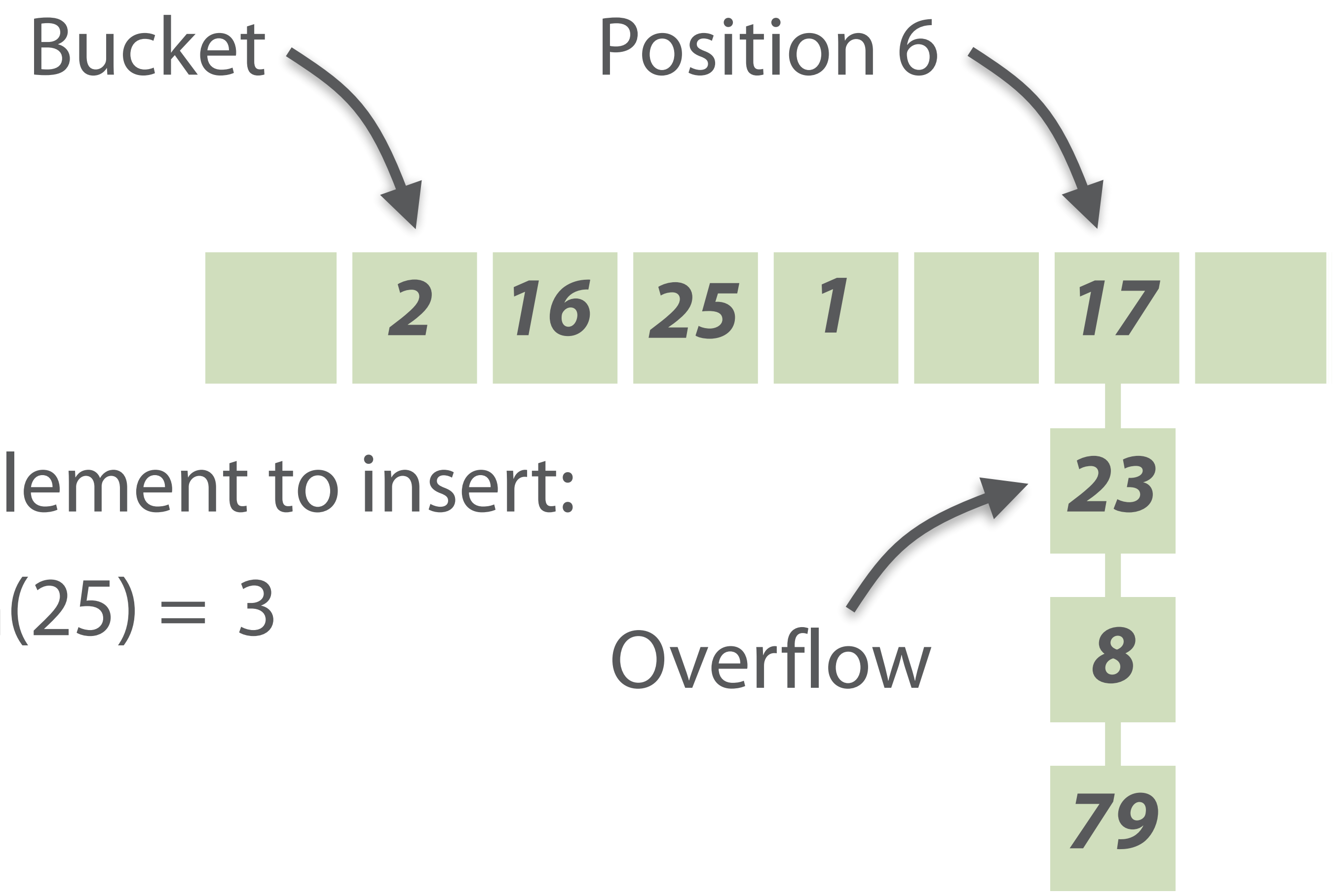
Overflow

Element to insert:

h(25) = 3

More than, say 75% full:

Hash Tables

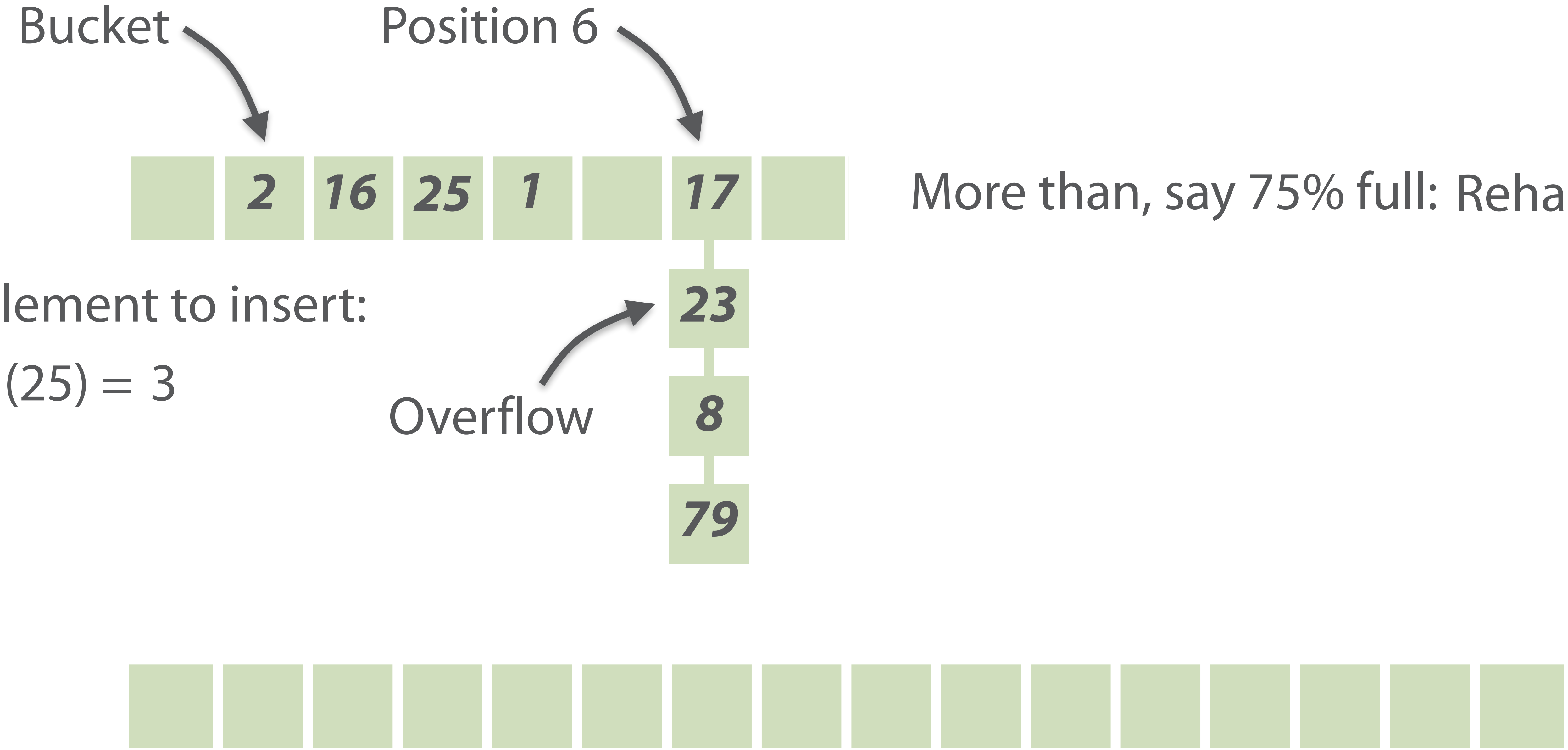


Element to insert:

h(25) = 3

More than, say 75% full: Rehash!

Hash Tables

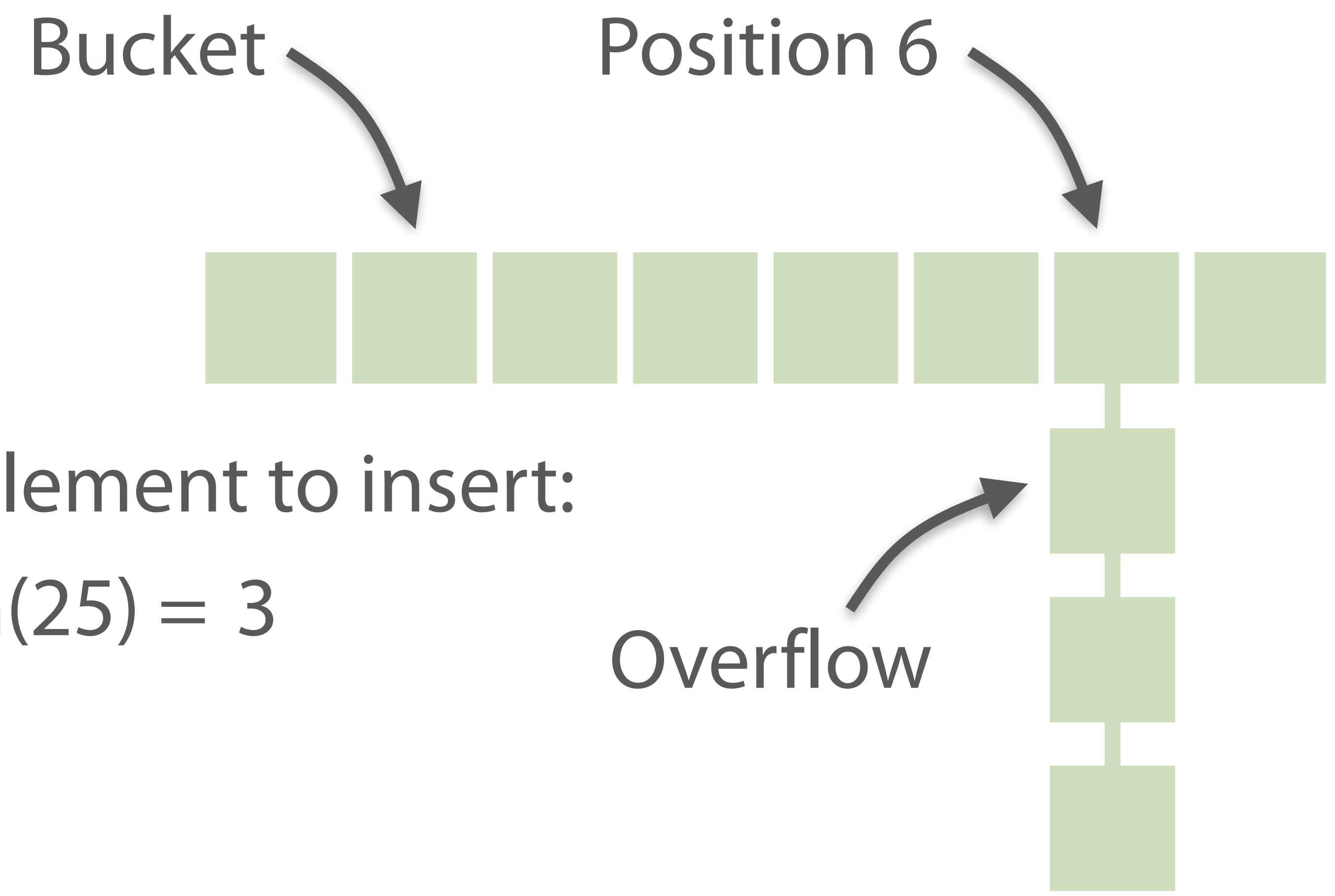


Element to insert:

h(25) = 3

More than, say 75% full: Rehash!

Hash Tables



Bucket

Position 6

Overflow

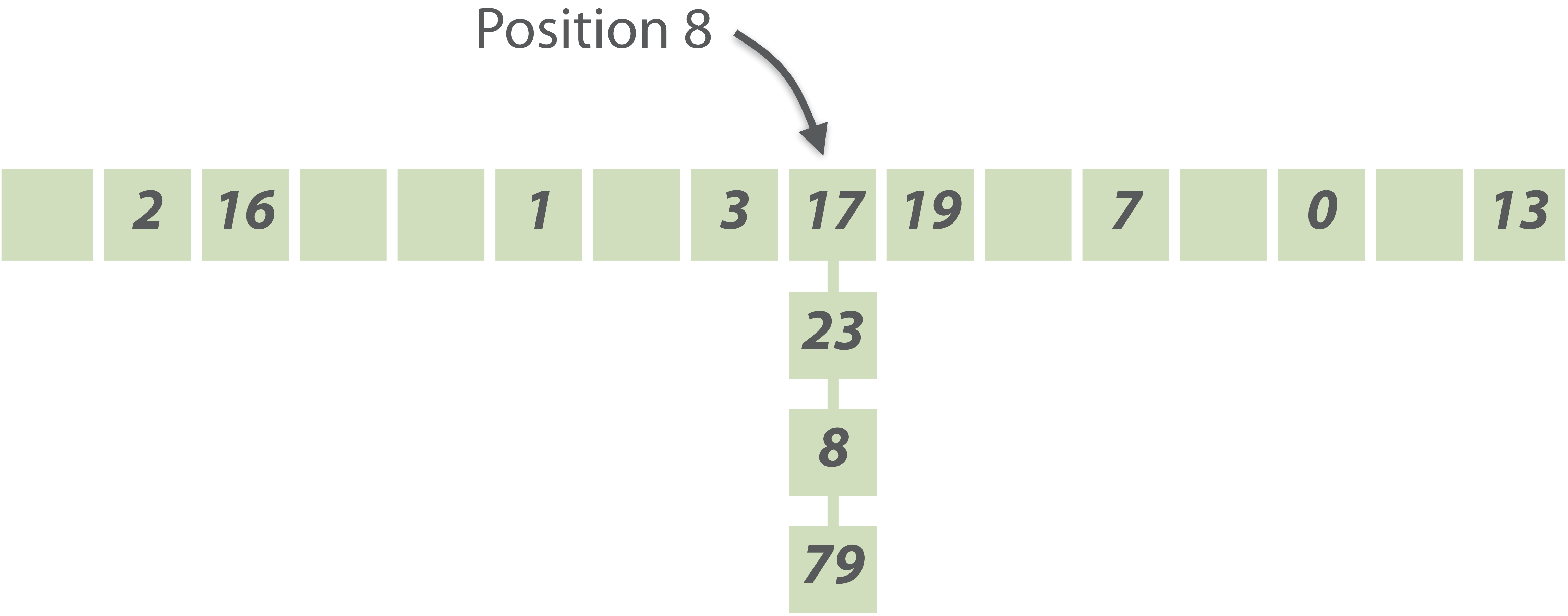
More than, say 75% full: Rehash!

Element to insert:

h(25) = 3

16		*25*		*23*			*2*		*1*	*79*				*17*	*8*

Complexity Analysis



Position 8

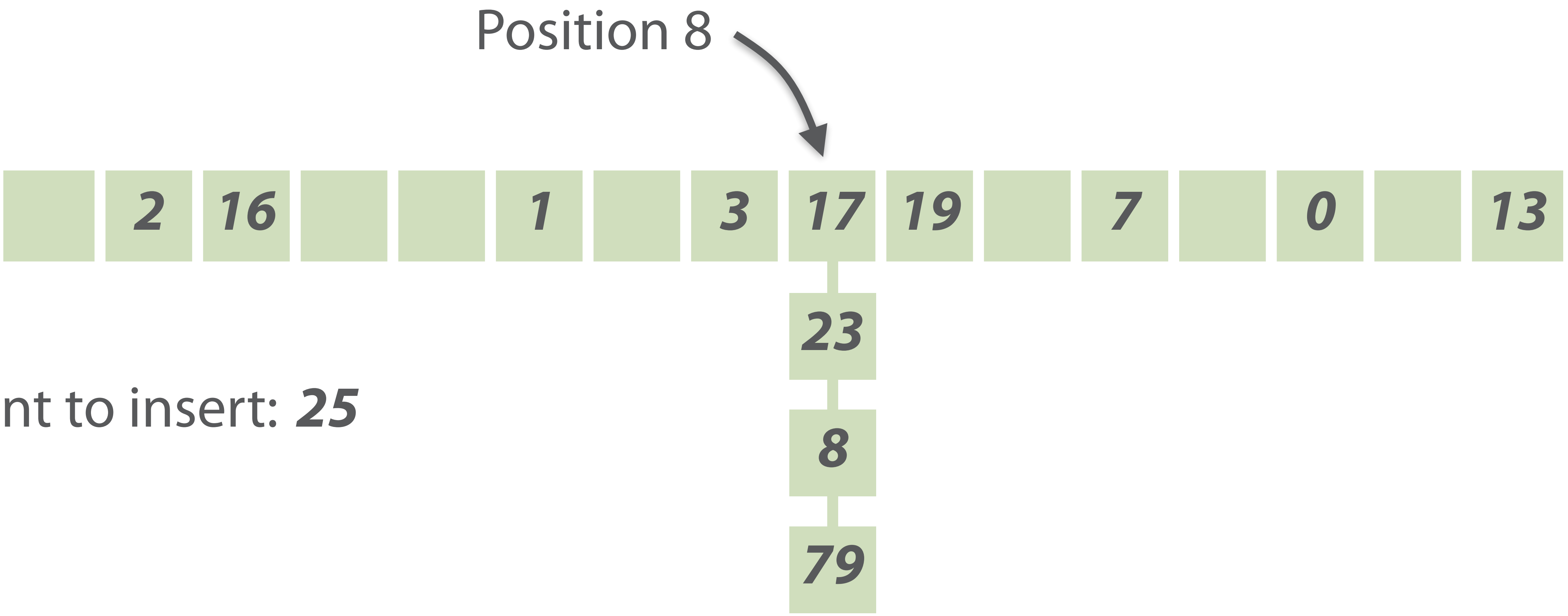
	2	16			1		3	17	19		7		0		13

23

8

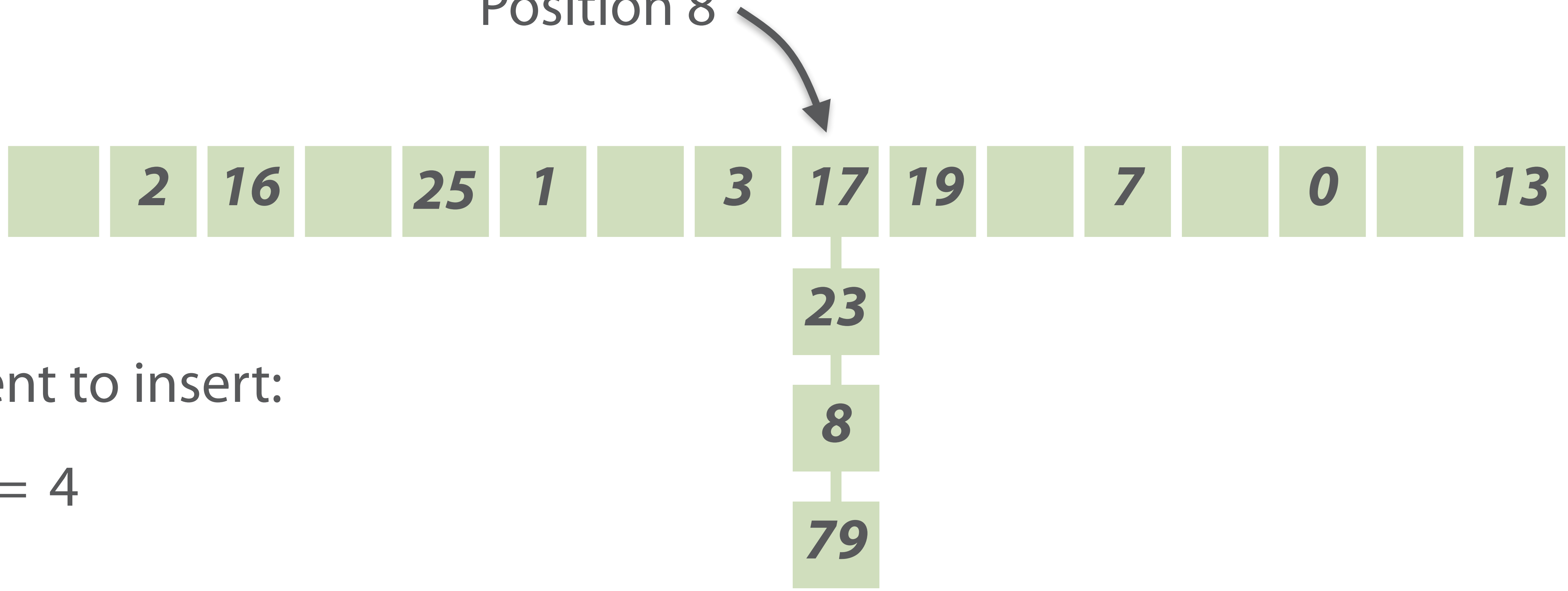
79

Complexity Analysis



Element to insert: ***25***

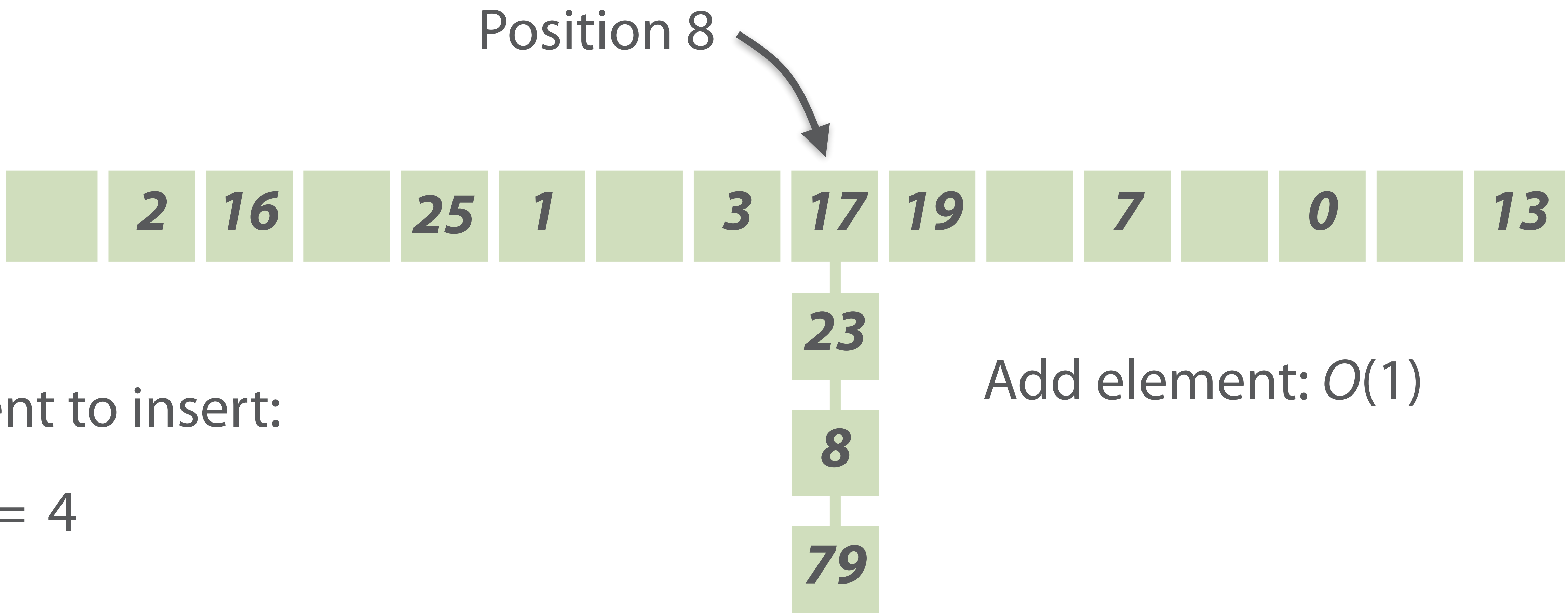
Complexity Analysis


Position 8

Element to insert:

h(25) = 4

Complexity Analysis

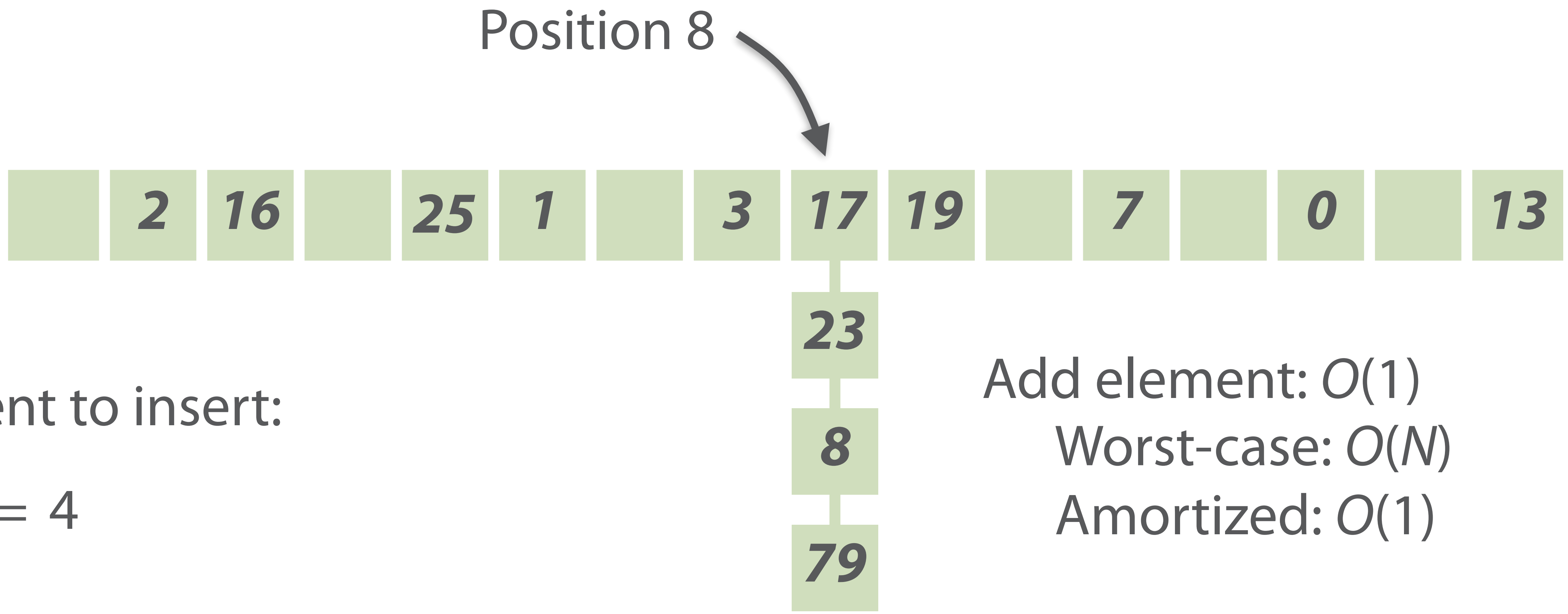


Add element: $O(1)$

Element to insert:

h(25) = 4

Complexity Analysis



Element to insert:

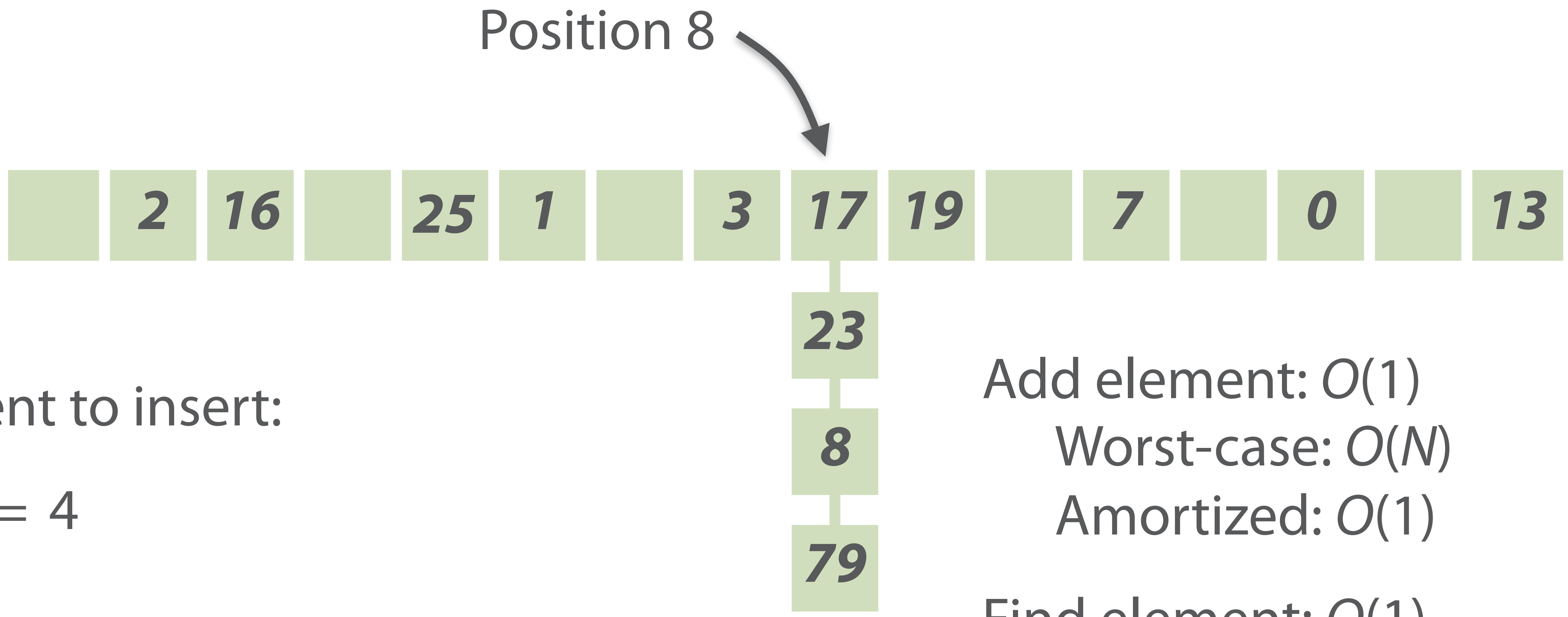
h(25) = 4

Add element: $O(1)$

Worst-case: $O(N)$

Amortized: $O(1)$

Complexity Analysis



Element to insert:

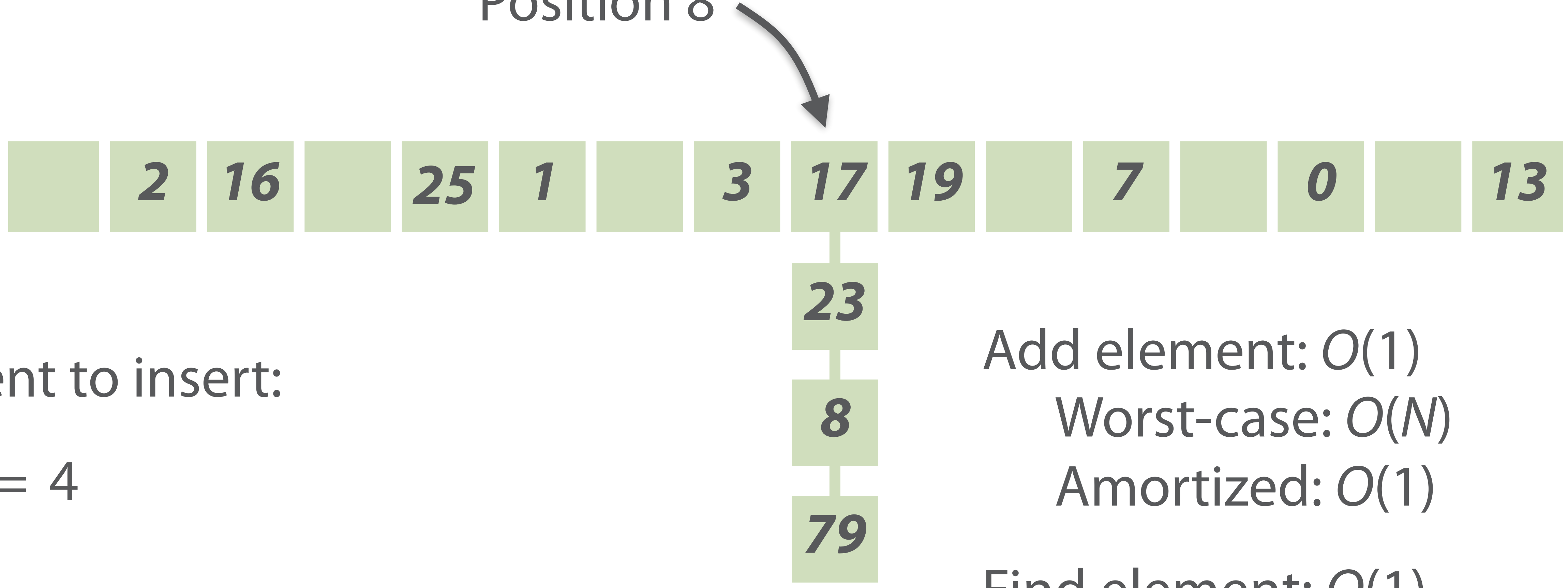
h(25) = 4

Add element: *O*(1)
  Worst-case: *O*(*N*)
  Amortized: *O*(1)

Find element: *O*(1)

Complexity Analysis

Position 8



Element to insert:

h(25) = 4

Add element: $O(1)$

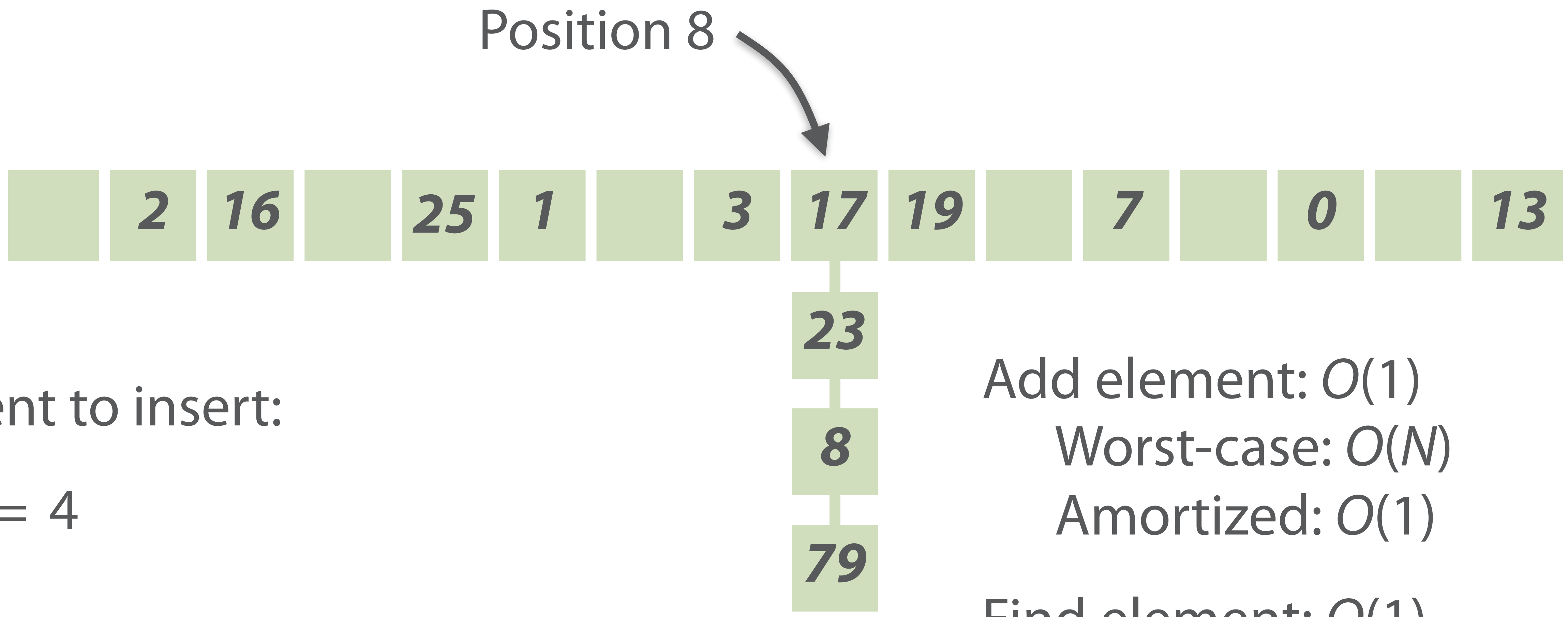
Worst-case: $O(N)$

Amortized: $O(1)$

Find element: $O(1)$

Theoretical worst-case: $O(N)$

Complexity Analysis



Element to insert:

h(25) = 4

Add element: $O(1)$

Worst-case: $O(N)$

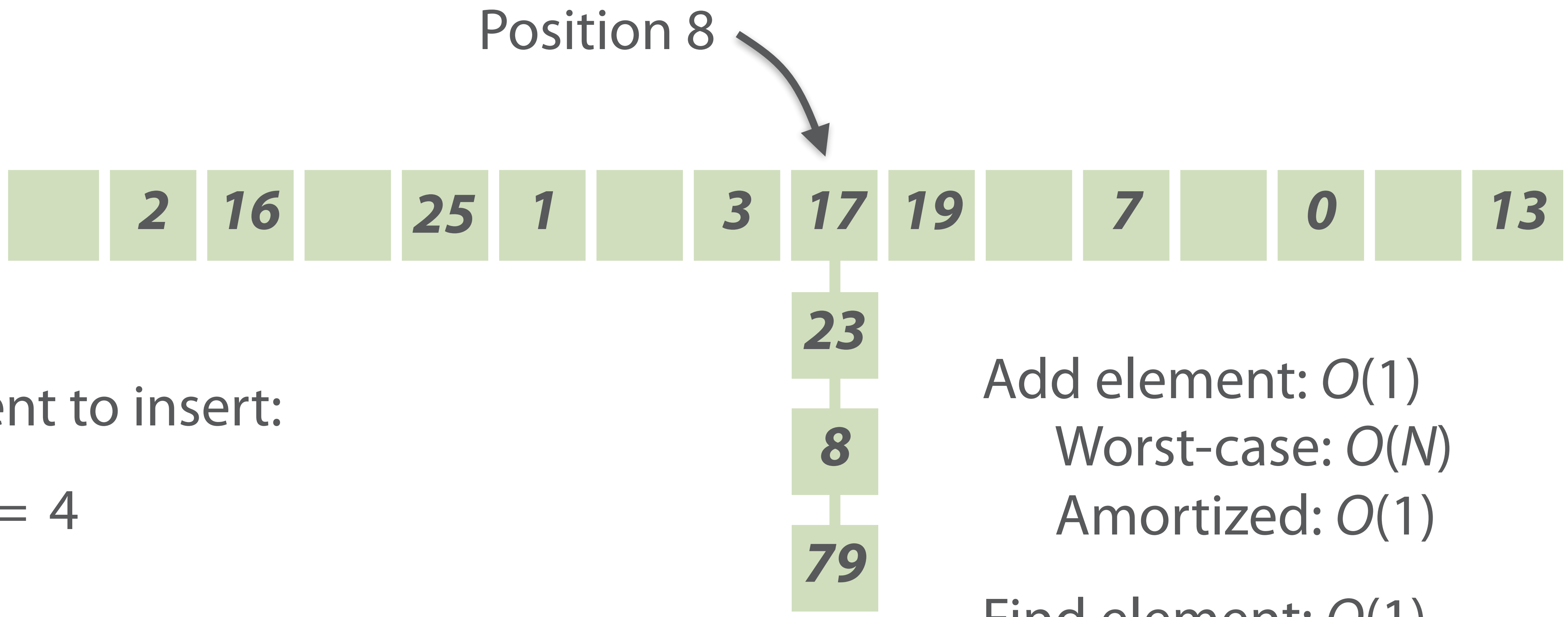
Amortized: $O(1)$

Find element: $O(1)$

Theoretical worst-case: $O(N)$

Remove element: $O(1)$

Complexity Analysis



Element to insert:

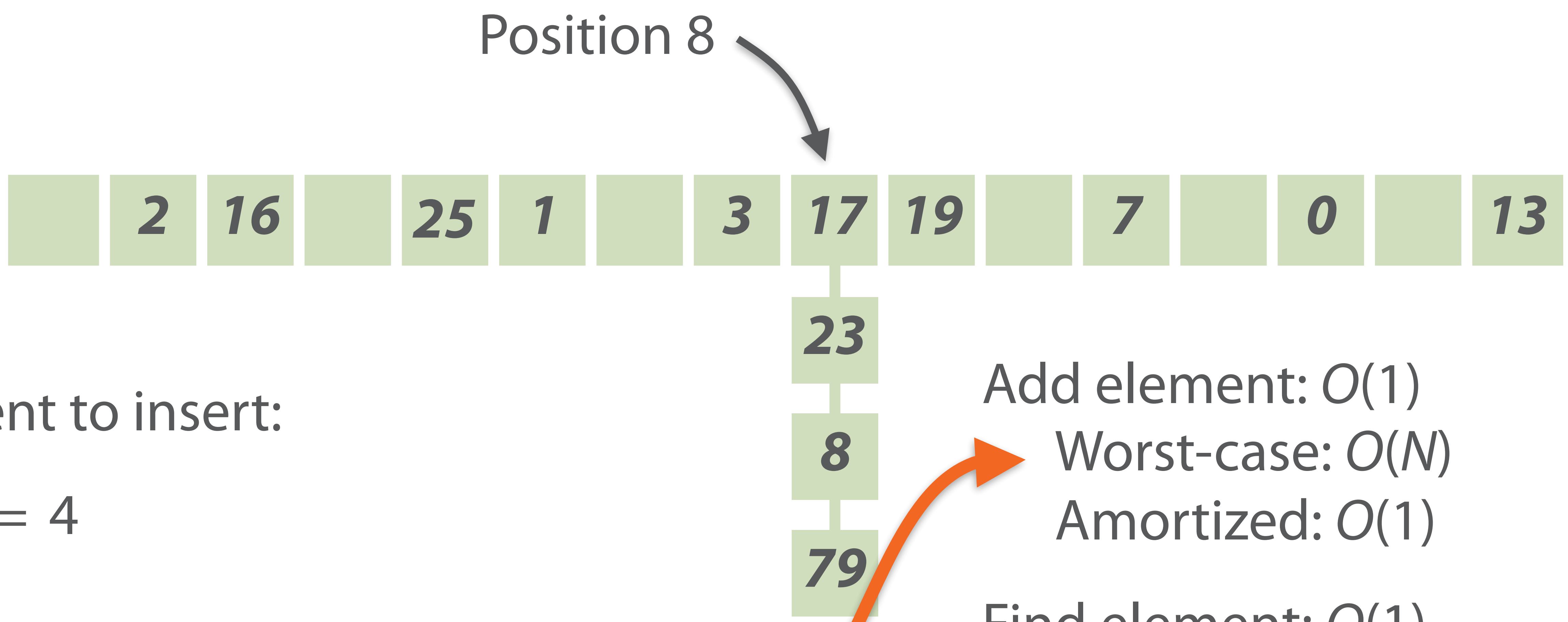
h(25) = 4

Add element: $O(1)$
  Worst-case: $O(N)$
  Amortized: $O(1)$

Find element: $O(1)$
  Theoretical worst-case: $O(N)$

Remove element: $O(1)$
  Theoretical worst-case: $O(N)$

Complexity Analysis



Element to insert:

h(25) = 4

Add element: *O*(1)

Worst-case: *O*(*N*)

Amortized: *O*(1)

Find element: *O*(1)

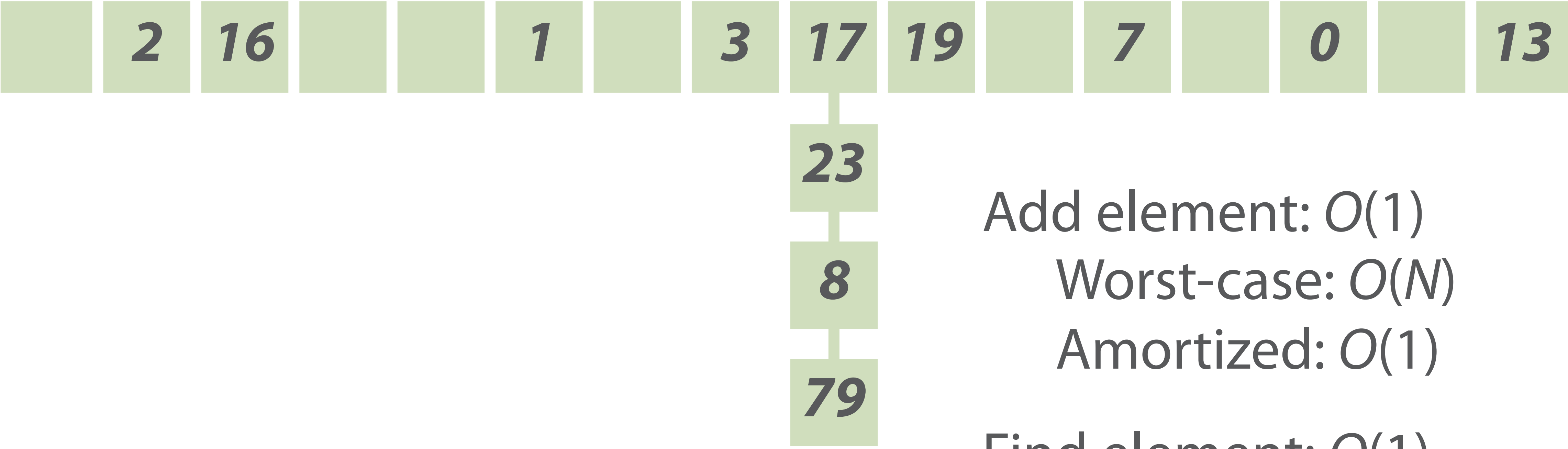
Theoretical worst-case: *O*(*N*)

Remove element: *O*(1)

Theoretical worst-case: *O*(*N*)

Really low probability!

Dictionaries and Maps



Add element: $O(1)$

Worst-case: $O(N)$

Amortized: $O(1)$

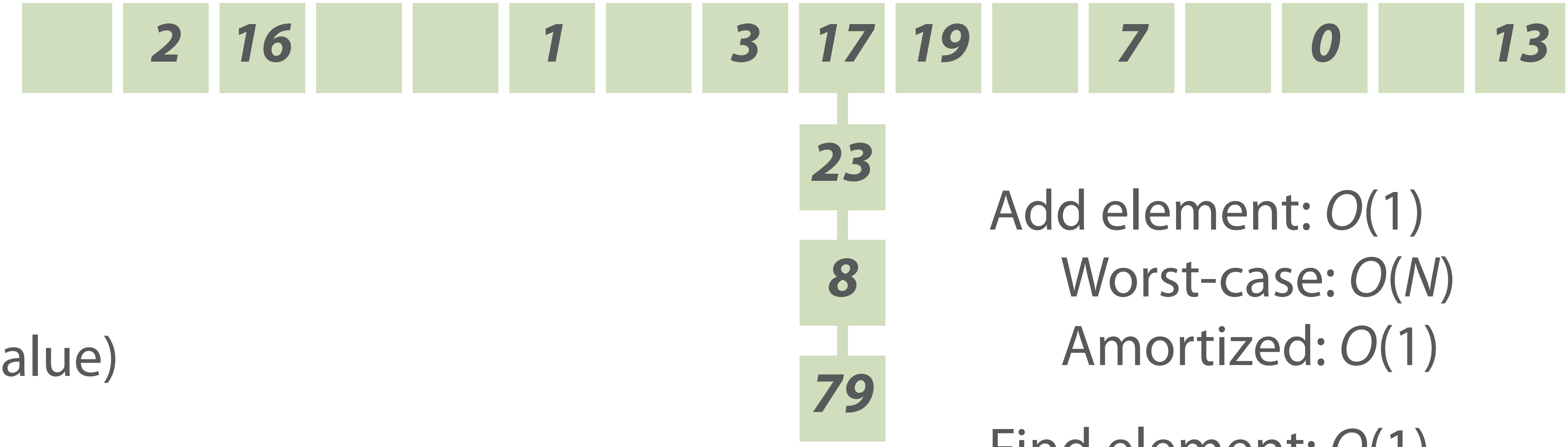
Find element: $O(1)$

Theoretical worst-case: $O(N)$

Remove element: $O(1)$

Theoretical worst-case: $O(N)$

Dictionaries and Maps



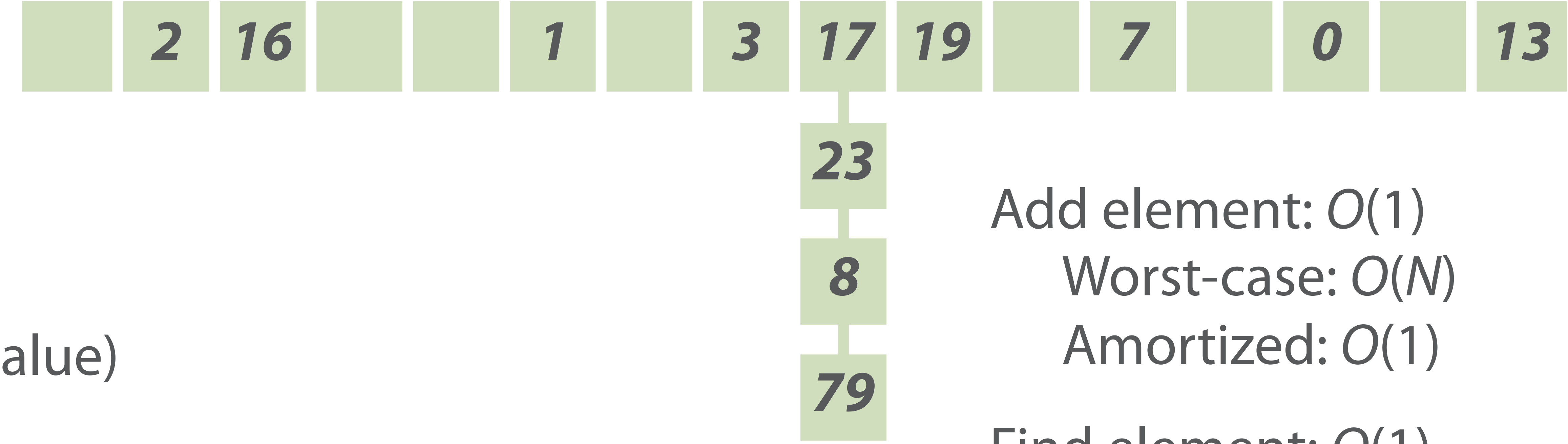
(key, value)

Add element: $O(1)$
Worst-case: $O(N)$
Amortized: $O(1)$

Find element: $O(1)$
Theoretical worst-case: $O(N)$

Remove element: $O(1)$
Theoretical worst-case: $O(N)$

Dictionaries and Maps



(key, value)

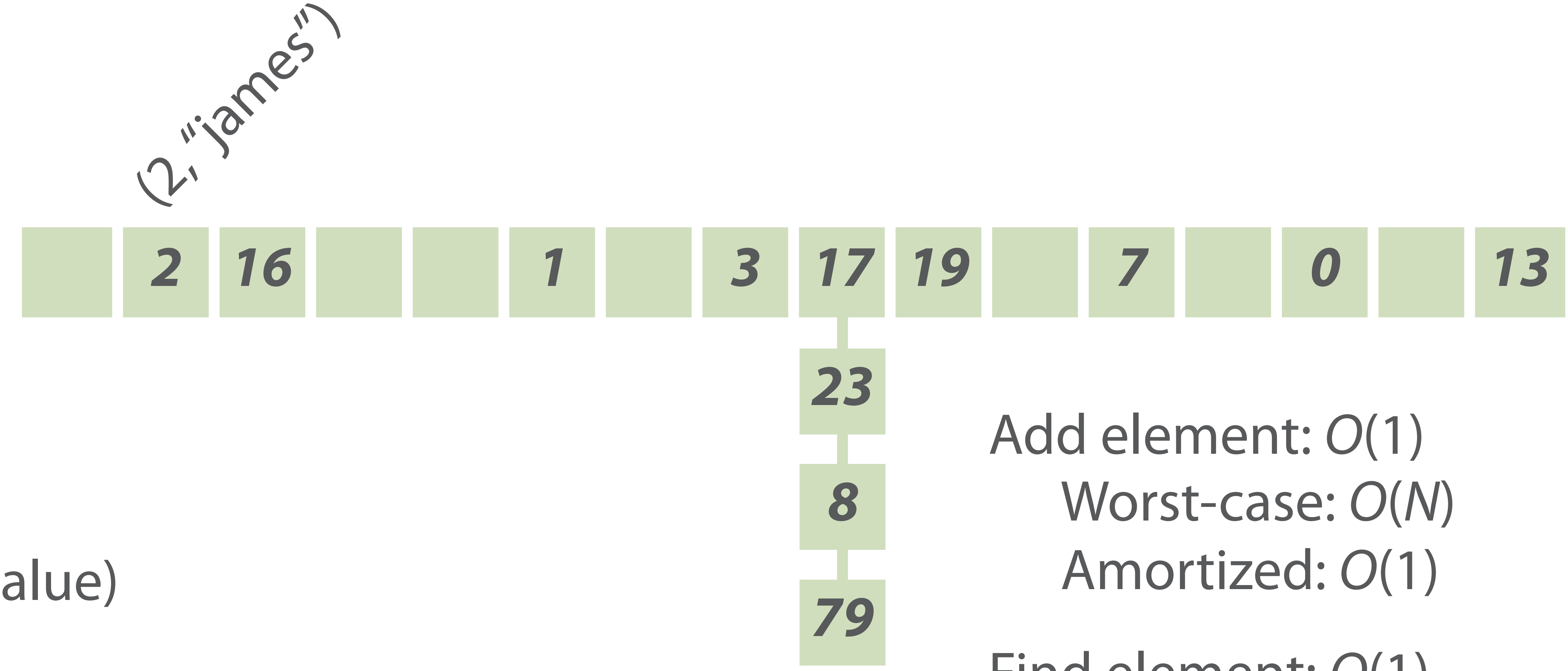
Position: h(key)

Add element: *O*(1)
    Worst-case: *O*(*N*)
    Amortized: *O*(1)

Find element: *O*(1)
  Theoretical worst-case: *O*(*N*)

Remove element: *O*(1)
  Theoretical worst-case: *O*(*N*)

Dictionaries and Maps



(key, value)

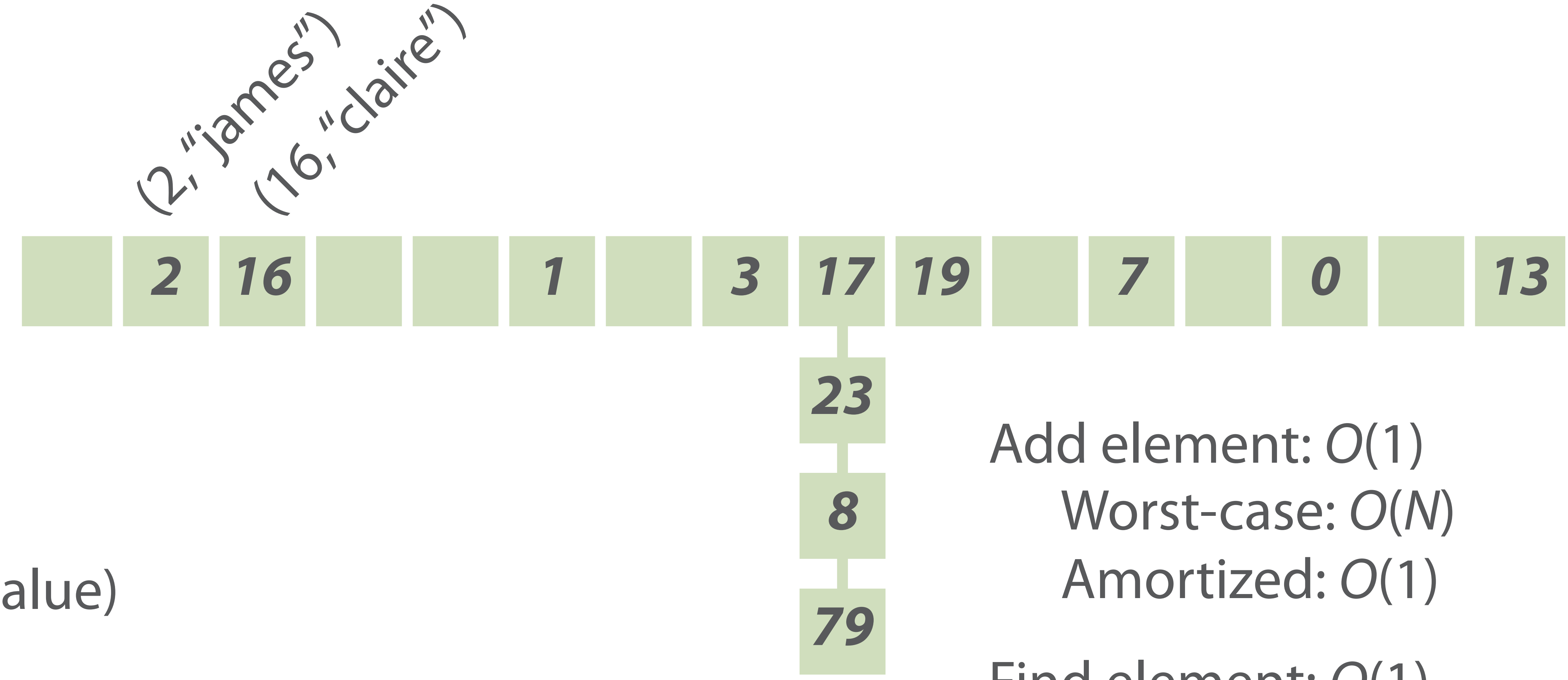
Position: h(key)

Add element: $O(1)$
Worst-case: $O(N)$
Amortized: $O(1)$

Find element: $O(1)$
Theoretical worst-case: $O(N)$

Remove element: $O(1)$
Theoretical worst-case: $O(N)$

Dictionaries and Maps



(key, value)

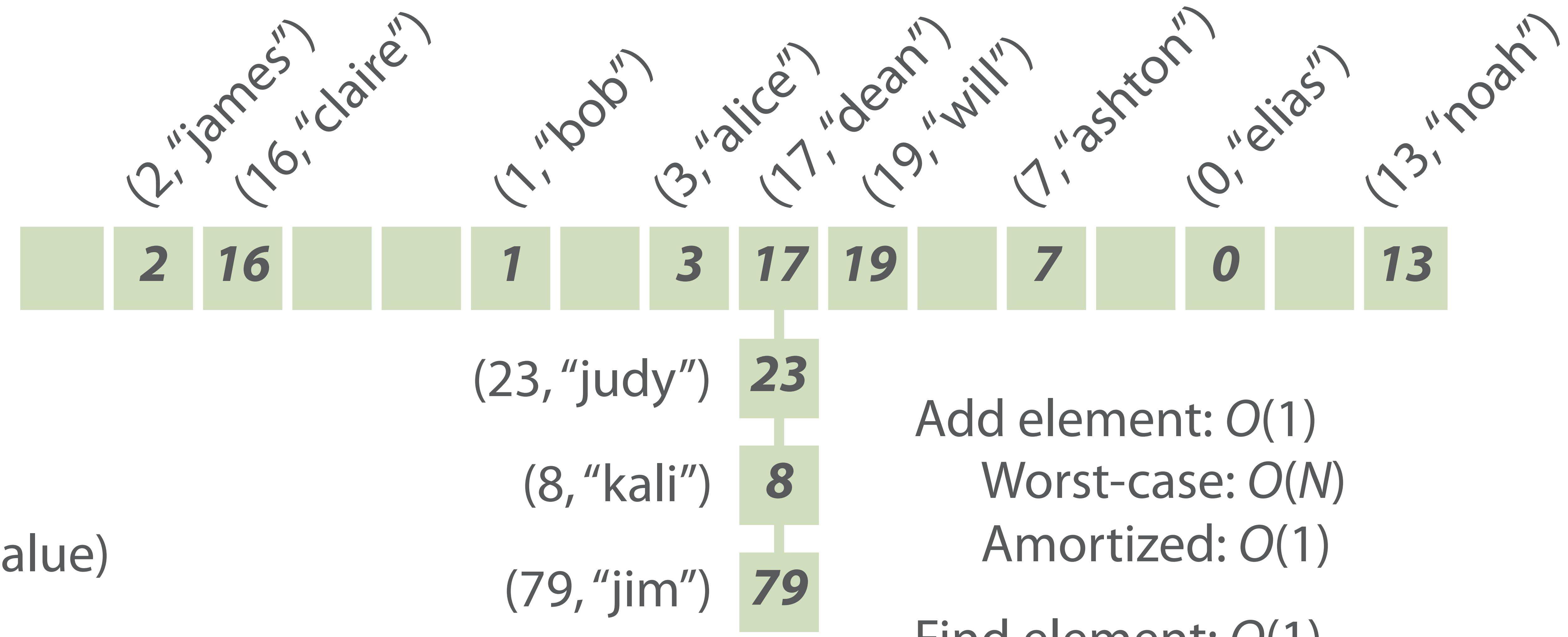
Position: h(key)

Add element: $O(1)$
Worst-case: $O(N)$
Amortized: $O(1)$

Find element: $O(1)$
Theoretical worst-case: $O(N)$

Remove element: $O(1)$
Theoretical worst-case: $O(N)$

Dictionaries and Maps



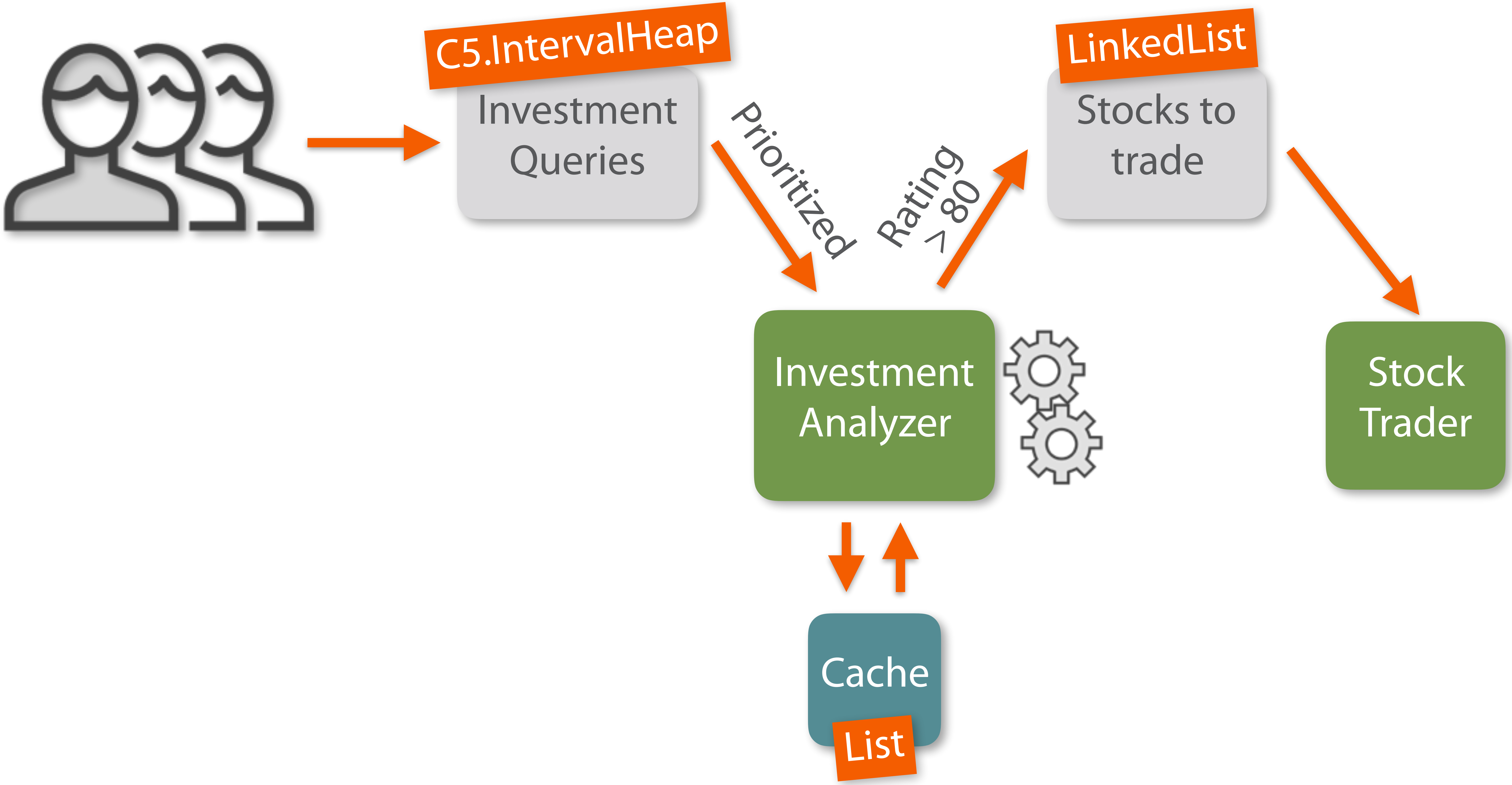
(key, value)

Position: h(key)

Add element: *O*(1)
Worst-case: *O*(*N*)
Amortized: *O*(1)

Find element: *O*(1)
Theoretical worst-case: *O*(*N*)

Remove element: *O*(1)
Theoretical worst-case: *O*(*N*)



C5.IntervalHeap

Investment Queries

Prioritized

Investment Analyzer

Cache

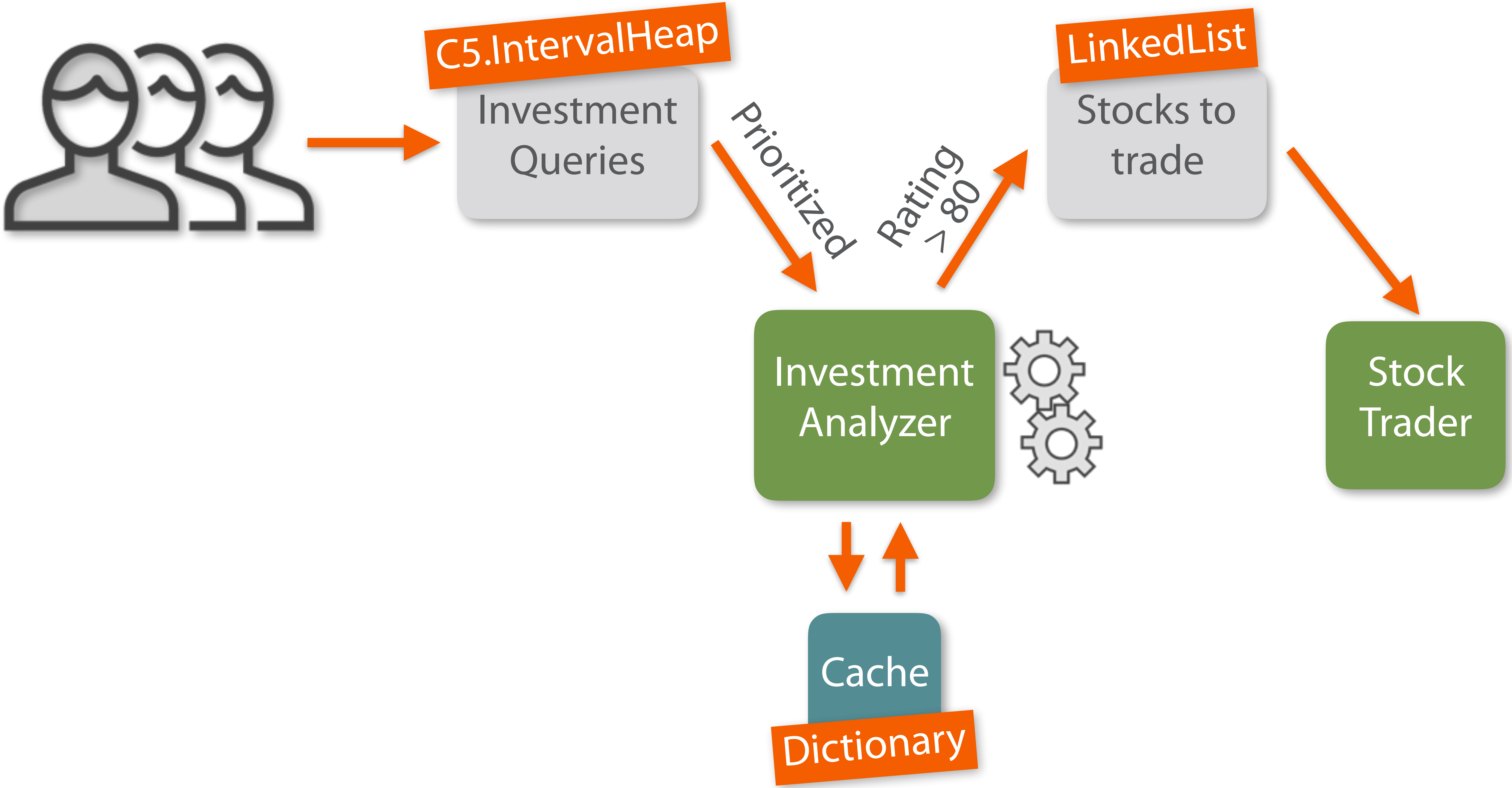
List

Rating > 80

LinkedList

Stocks to trade

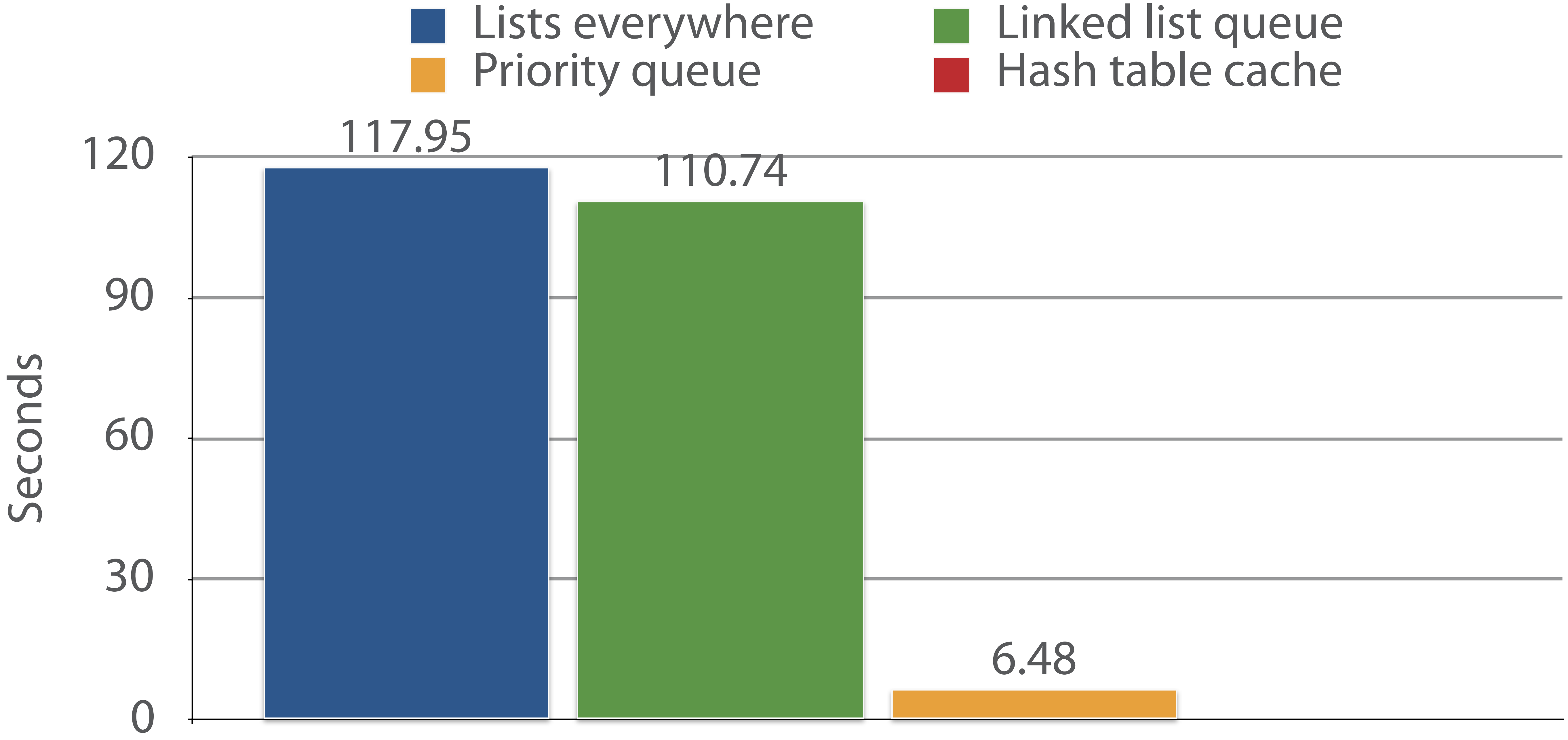
Stock Trader


C5.IntervalHeap
Investment Queries
Prioritized
Rating > 80
LinkedList
Stocks to trade
Investment Analyzer
Stock Trader
Cache
Dictionary

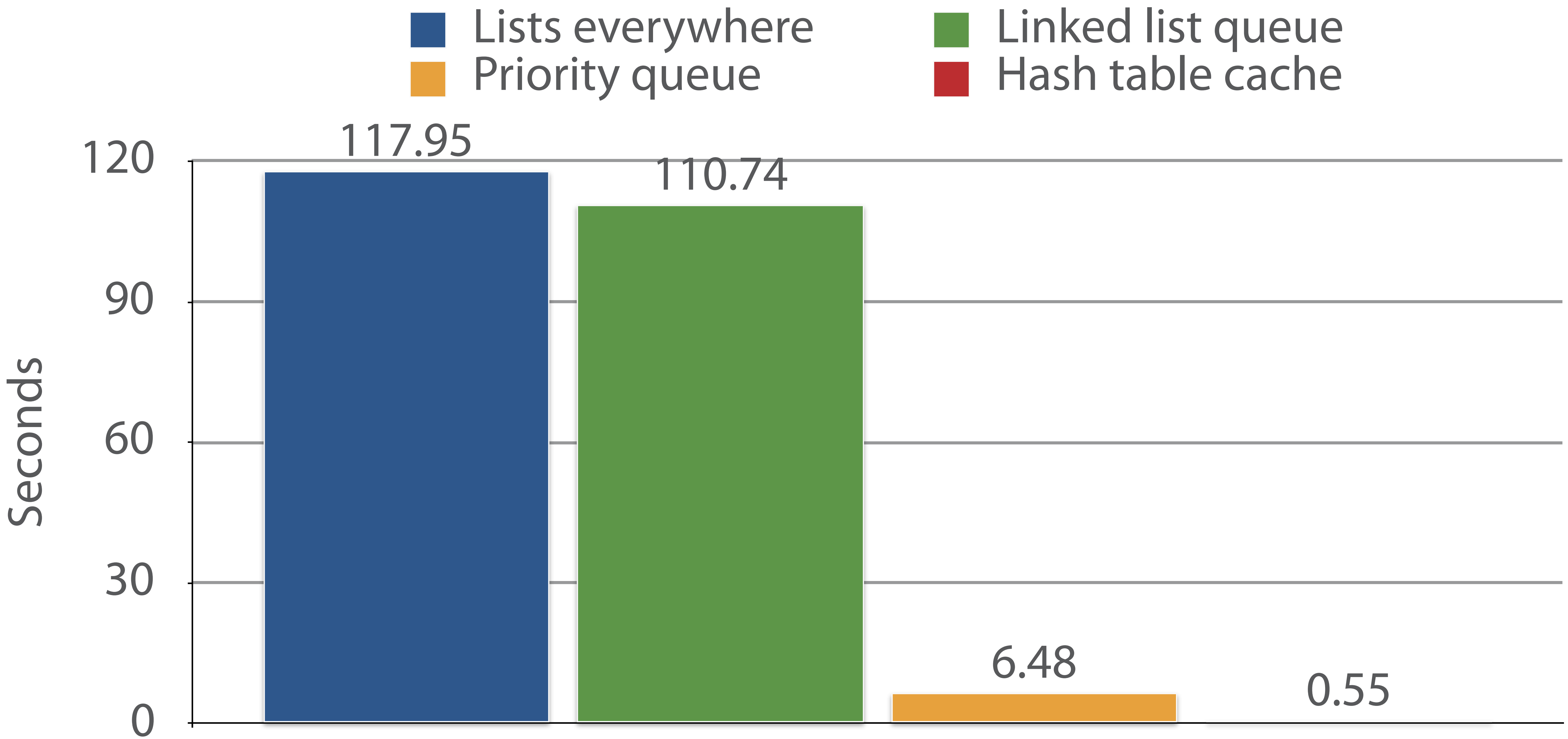
Investment Analyzer

Using a Dictionary as cache

Effect


Lists everywhere
Linked list queue
Priority queue
Hash table cache
Seconds
120
90
60
30
0
117.95
110.74
6.48

Effect


Lists everywhere
Linked list queue
Priority queue
Hash table cache
Seconds
120
90
60
30
0
117.95
110.74
6.48
0.55

Lessons Learned

Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned

Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned

Order preserving

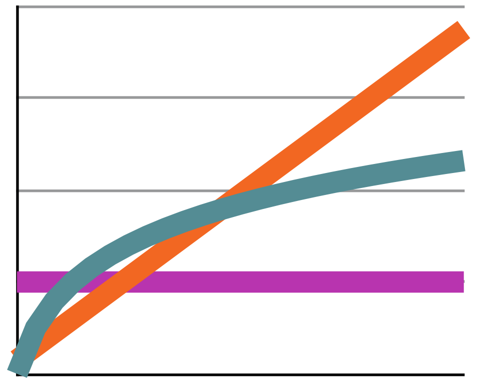
Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned



Order preserving

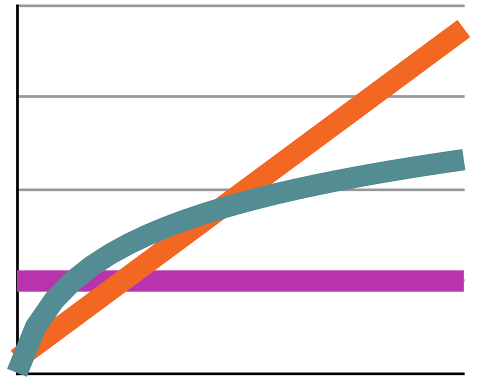
Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned



Add (amortized): $O(1)$

Order preserving

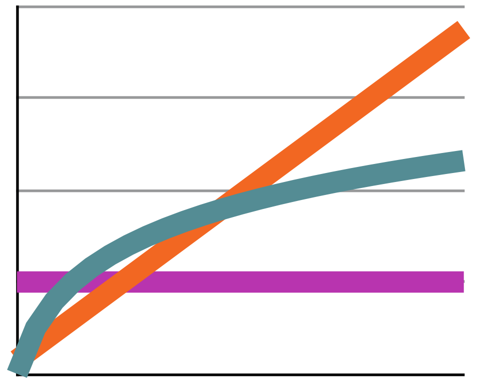
Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Order preserving

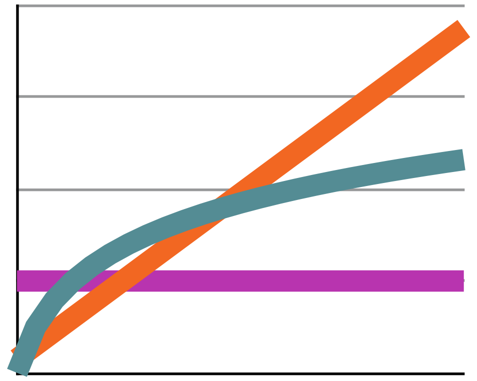
Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Order preserving

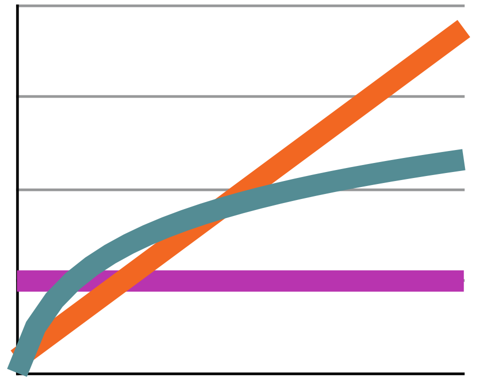
Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

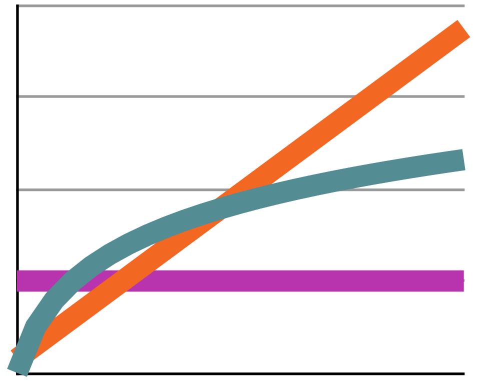
Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

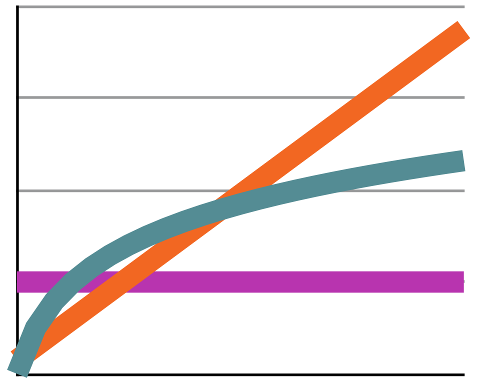
Dynamic array

Hash table

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

Dynamic array

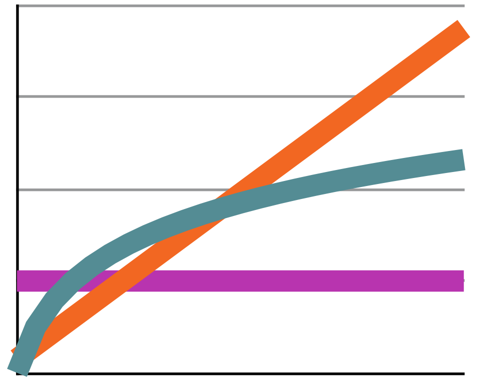
Hash table

Order preserving

Linked list

Priority queue

Lessons Learned



Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Order preserving

Dynamic array

Hash table

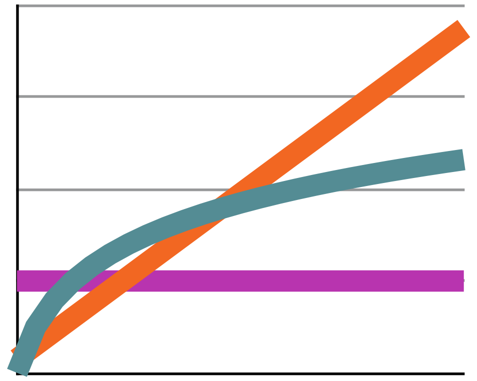
No reorganization

Order preserving

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

Dynamic array

Hash table

No reorganization

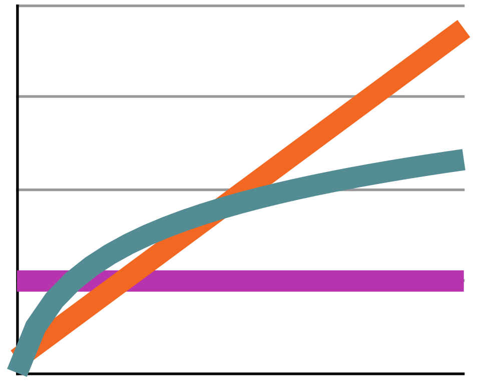
Get first/last: *O*(1)

Order preserving

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

Dynamic array

Hash table

No reorganization

Get first/last: *O*(1)

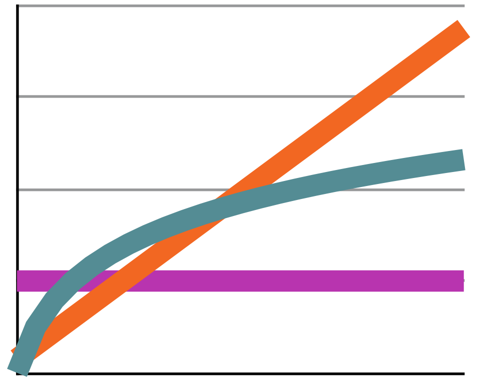
Go to index: *O*(*N*)

Order preserving

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

Dynamic array

Hash table

No reorganization

Get first/last: *O*(1)

Go to index: *O*(*N*)

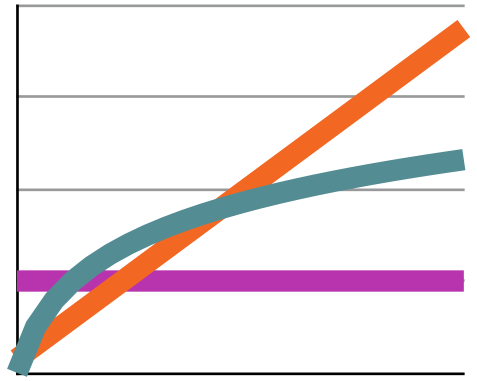
Add/remove/merge: *O*(1)

Order preserving

Linked list

Priority queue

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

Dynamic array

Hash table

No reorganization

Get first/last: *O*(1)

Go to index: *O*(*N*)

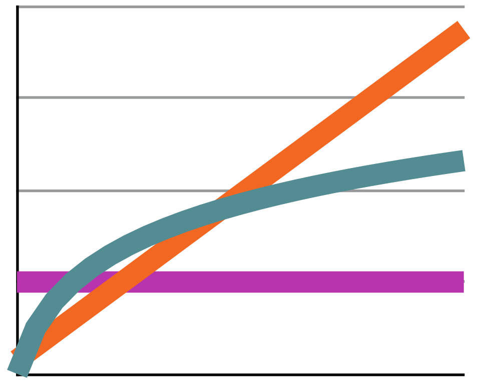
Add/remove/merge: *O*(1)

Order preserving

Linked list

Priority queue

Lessons Learned



Dynamic array

Order preserving

Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Hash table

Linked list

Order preserving

No reorganization

Get first/last: $O(1)$

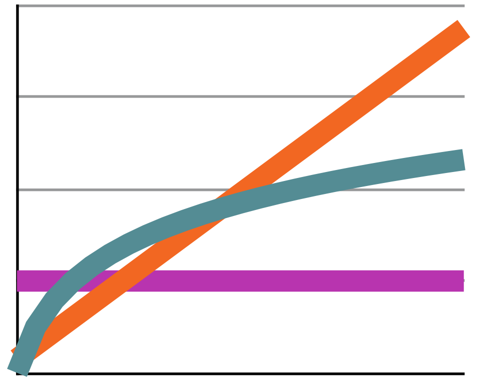
Go to index: $O(N)$

Add/remove/merge: $O(1)$

Priority queue

Order enforcing

Lessons Learned



Add (amortized): *O*(1)

Remove: *O*(*N*)

Go to index: *O*(1)

Find: *O*(*N*)

Order preserving

Dynamic array

Hash table

No reorganization

Get first/last: *O*(1)

Go to index: *O*(*N*)

Add/remove/merge: *O*(1)

Order preserving

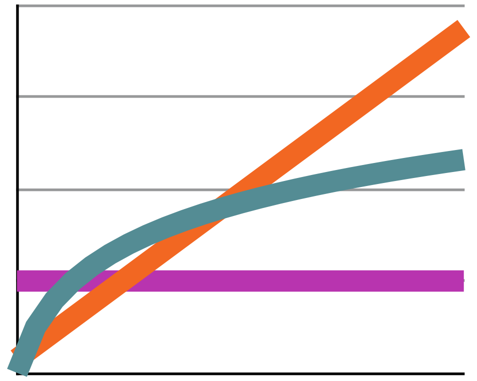
Linked list

Order enforcing

Get smallest/largest: *O*(1)

Priority queue

Lessons Learned



Dynamic array

Order preserving

Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Hash table

Linked list

Order preserving

No reorganization

Get first/last: $O(1)$

Go to index: $O(N)$

Add/remove/merge: $O(1)$

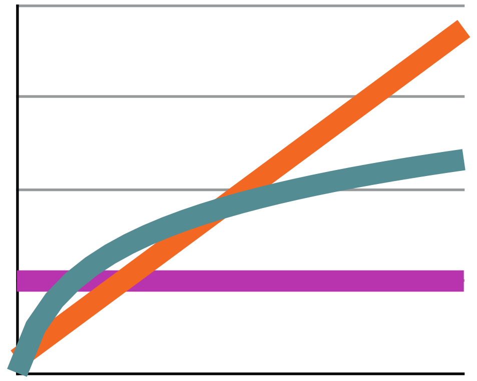
Priority queue

Order enforcing

Get smallest/largest: $O(1)$

Add: $O(\log_2 N)$

Lessons Learned



Dynamic array

Order preserving

Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Hash table

Linked list

Order preserving

No reorganization

Get first/last: $O(1)$

Go to index: $O(N)$

Add/remove/merge: $O(1)$

Priority queue

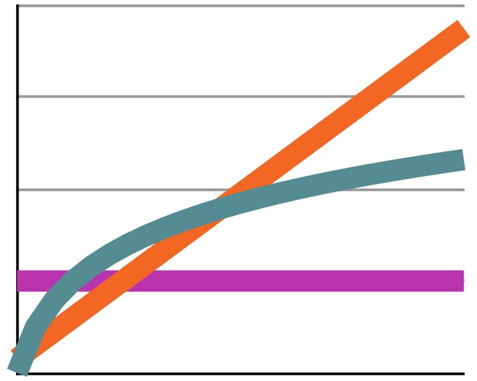
Order enforcing

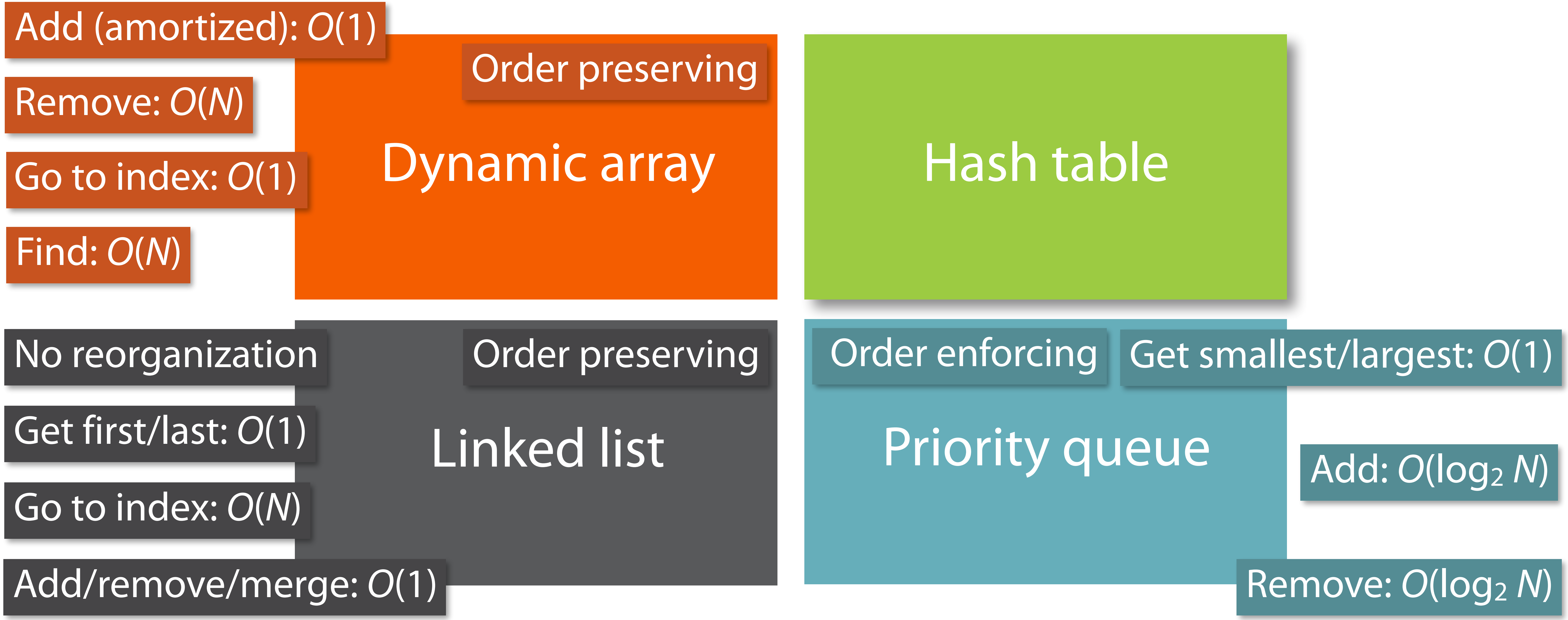
Get smallest/largest: $O(1)$

Add: $O(\log_2 N)$

Remove: $O(\log_2 N)$

Lessons Learned





Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Order preserving

Dynamic array

Hash table

No reorganization

Get first/last: $O(1)$

Go to index: $O(N)$

Add/remove/merge: $O(1)$

Order preserving

Linked list

Order enforcing

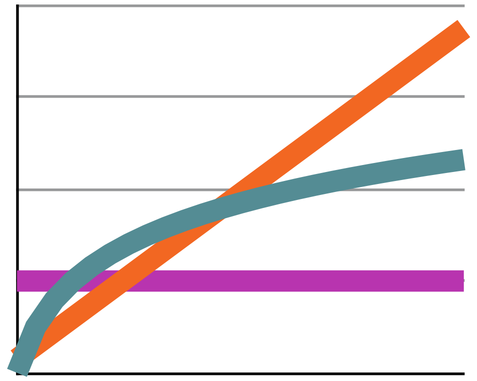
Get smallest/largest: $O(1)$

Priority queue

Add: $O(\log_2 N)$

Remove: $O(\log_2 N)$

Lessons Learned



Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Order preserving

Dynamic array

Unordered

Hash table

No reorganization

Get first/last: $O(1)$

Go to index: $O(N)$

Add/remove/merge: $O(1)$

Order preserving

Linked list

Order enforcing

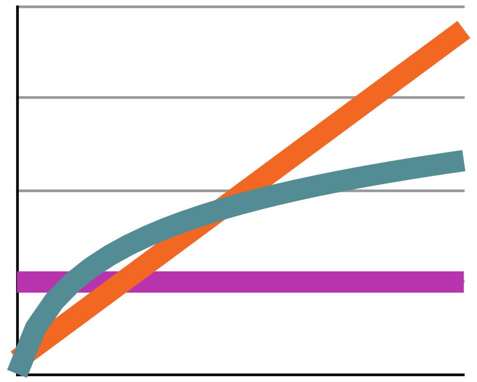
Get smallest/largest: $O(1)$

Priority queue

Add: $O(\log_2 N)$

Remove: $O(\log_2 N)$

Lessons Learned



Dynamic array

Order preserving

- Add (amortized): $O(1)$
- Remove: $O(N)$
- Go to index: $O(1)$
- Find: $O(N)$

Hash table

Unordered

- Add (amortized): $O(1)$

Linked list

Order preserving

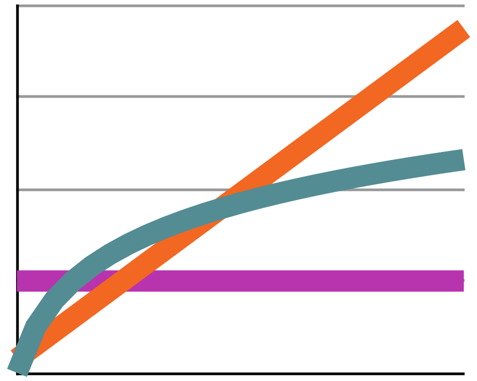
- No reorganization
- Get first/last: $O(1)$
- Go to index: $O(N)$
- Add/remove/merge: $O(1)$

Priority queue

Order enforcing

- Get smallest/largest: $O(1)$
- Add: $O(\log_2 N)$
- Remove: $O(\log_2 N)$

Lessons Learned



Dynamic array

Order preserving

Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Hash table

Unordered

Add (amortized): $O(1)$

Remove: $O(1)$

Linked list

Order preserving

No reorganization

Get first/last: $O(1)$

Go to index: $O(N)$

Add/remove/merge: $O(1)$

Priority queue

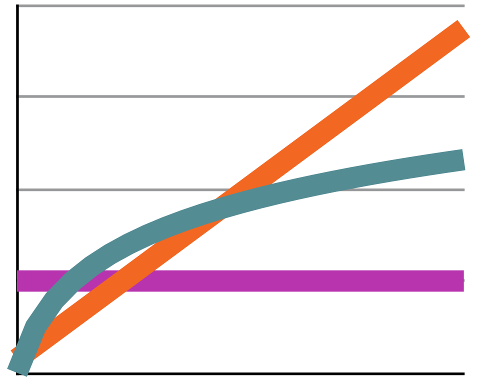
Order enforcing

Get smallest/largest: $O(1)$

Add: $O(\log_2 N)$

Remove: $O(\log_2 N)$

Lessons Learned



Dynamic array

Order preserving

- Add (amortized): $O(1)$
- Remove: $O(N)$
- Go to index: $O(1)$
- Find: $O(N)$

Hash table

Unordered

- Add (amortized): $O(1)$
- Remove: $O(1)$
- Find: $O(1)$

Linked list

Order preserving

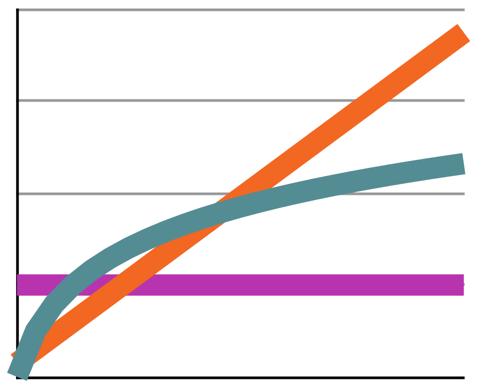
- No reorganization
- Get first/last: $O(1)$
- Go to index: $O(N)$
- Add/remove/merge: $O(1)$

Priority queue

Order enforcing

- Get smallest/largest: $O(1)$
- Add: $O(\log_2 N)$
- Remove: $O(\log_2 N)$

Lessons Learned



Dynamic array

Order preserving

Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Hash table

Unordered

Add (amortized): $O(1)$

Remove: $O(1)$

Find: $O(1)$

“Uniquify”, cache/dictionary

Linked list

Order preserving

No reorganization

Get first/last: $O(1)$

Go to index: $O(N)$

Add/remove/merge: $O(1)$

Priority queue

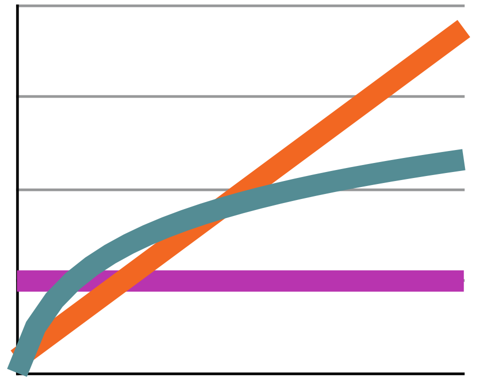
Order enforcing

Get smallest/largest: $O(1)$

Add: $O(\log_2 N)$

Remove: $O(\log_2 N)$

Lessons Learned



Dynamic array

Order preserving

Add (amortized): $O(1)$

Remove: $O(N)$

Go to index: $O(1)$

Find: $O(N)$

Hash table

Unordered

Add (amortized): $O(1)$

Remove: $O(1)$

Find: $O(1)$

"Uniquify", cache/dictionary

Linked list

Order preserving

No reorganization

Get first/last: $O(1)$

Go to index: $O(N)$

Add/remove/merge: $O(1)$

Priority queue

Order enforcing

Get smallest/largest: $O(1)$

Add: $O(\log_2 N)$

Remove: $O(\log_2 N)$