

Stacks and queues

Abstract data types and data structures

- An **abstract data type**
 - Specifies supported operations
- A **data structure** is an implementation of an **abstract data type**
 - Specifies data layout in memory
 - Specifies algorithms for operations

Lists

- support storing multiple elements together
and optionally append, insert, delete, random access, ...
- implementations:
 - **dynamic arrays**
everything $O(1)$ in practice except insert/delete $O(n)$
 - **linked lists**
everything $O(1)$ (but slower than arrays) except random access $O(n)$

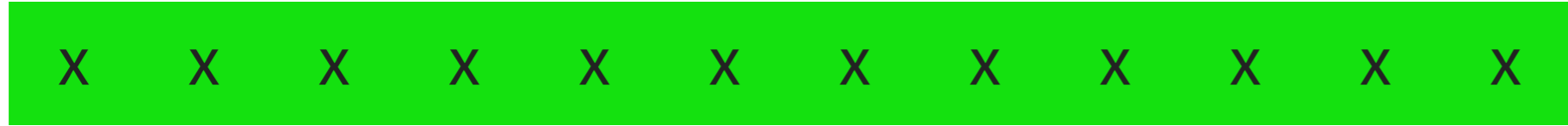
Stacks / LIFO

- A **stack** is an ordered collection of elements
- supports two operations:
 - “push”: add an element
 - “pop”: retrieve-and-remove the last-added element

⇒ **last in, first out (LIFO)**

Static array implementation of a **stack**

- Useful **only when there is a hard limit** on the number of elements
- We maintain a **static array**
- and a **stack pointer** (or **top index**)
- this is how “the” stack is implemented
(for storing function arguments, local variables and return addresses)



↑ stack pointer

- push A
- push B
- push C
- pop
- pop
- push E
- push F
- push G
- pop
- pop
- pop
- pop
- .



↑ stack pointer

push A

→ push B

push C

pop

pop

push E

push F

push G

pop

pop

pop

pop

.



↑ stack pointer

push A

push B

→ push C

pop

pop

push E

push F

push G

pop

pop

pop

pop

.



push A

push B

push C

→ pop

pop

push E

push F

push G

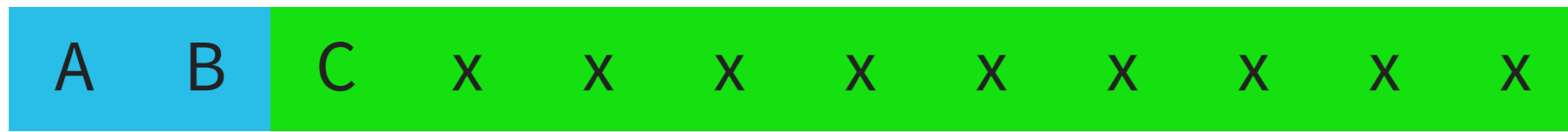
pop

pop

pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

→ pop

push E

push F

push G

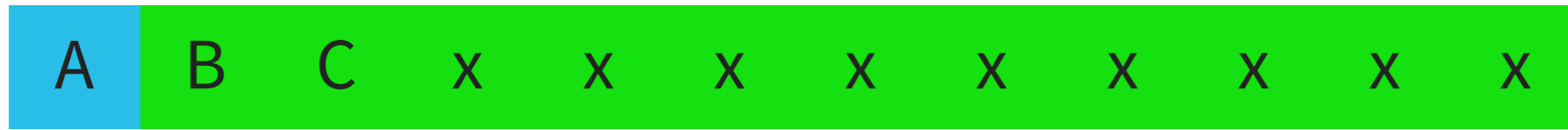
pop

pop

pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

pop → B

→ push E

push F

push G

pop

pop

pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

pop → B

push E

→ push F

push G

pop

pop

pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

pop → B

push E

push F

→ push G

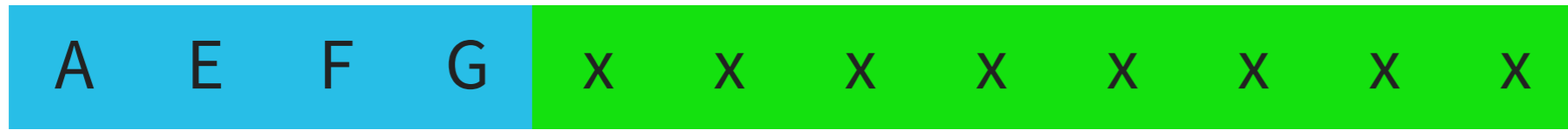
pop

pop

pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

pop → B

push E

push F

push G

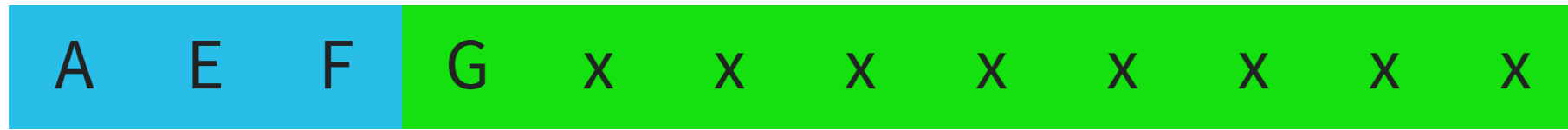
→ pop

pop

pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

pop → B

push E

push F

push G

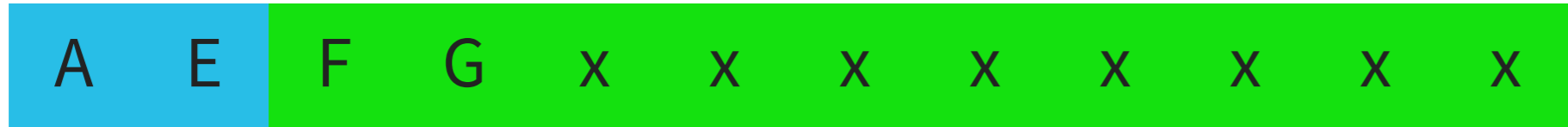
pop → G

→ pop

pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

pop → B

push E

push F

push G

pop → G

pop → F

→ pop

pop

.



↑ stack pointer

push A

push B

push C

pop → C

pop → B

push E

push F

push G

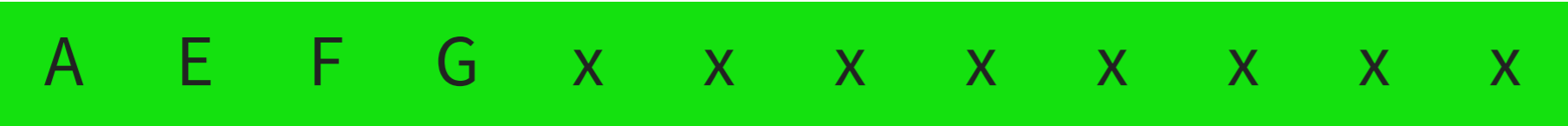
pop → G

pop → F

pop → E

→ pop

.



↑ stack pointer

push A
push B
push C
pop → C
pop → B
push E
push F
push G
pop → G
pop → F
pop → E
pop → A

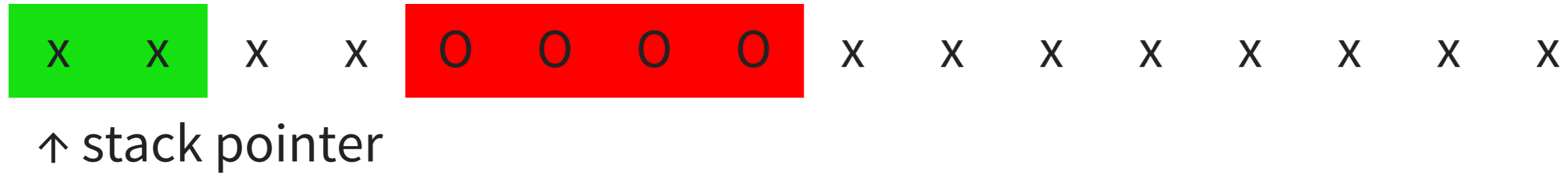
→ .

Linked list implementation of a **stack**

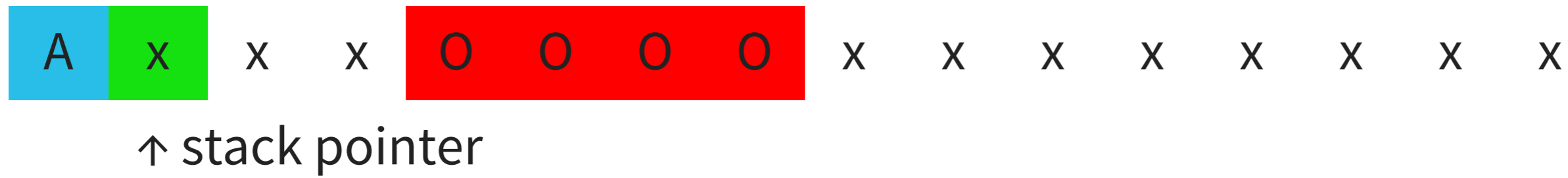
- **Pro:** No hard limit on number of elements
- **Con:**
 - Memory allocation for every **push**
 - Memory freed for every **pop**

Dynamic array implementation of a stack

- Pros:
 - No hard limit on number of elements
 - Memory management overhead is small
- Con:
 - No pointer stability



- push A
- push B
- push C
- p = address of C
- push D
- push E
- change C into C' using p
- .



push A

→ push B

push C

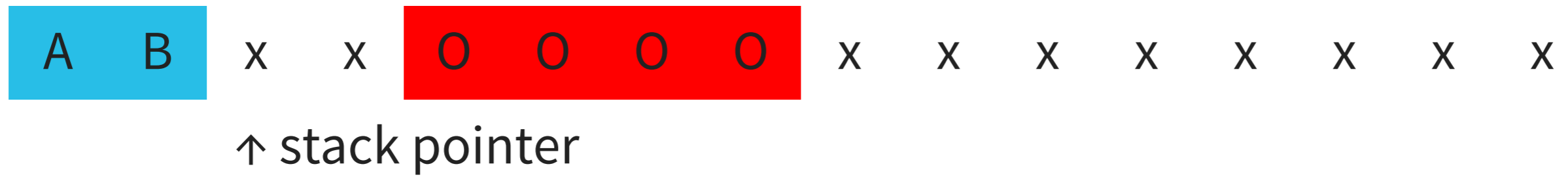
p = address of C

push D

push E

change C into C' using p

.



push A

push B

→ push C

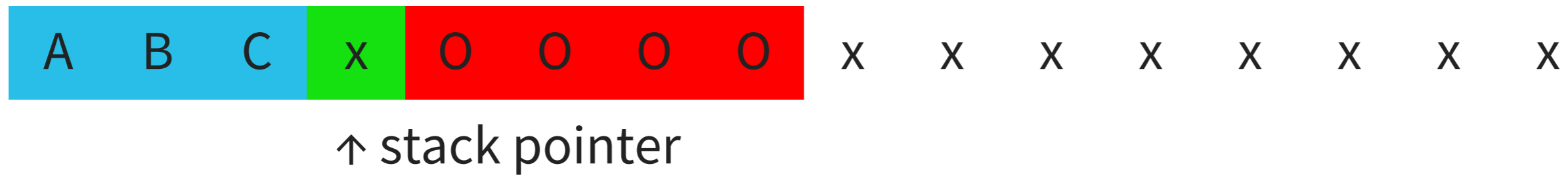
p = address of C

push D

push E

change C into C' using p

.



push A

push B

push C

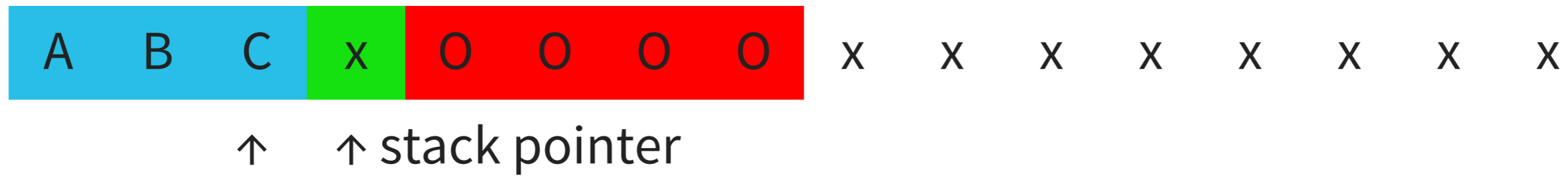
→ p = address of C

push D

push E

change C into C' using p

.



push A

push B

push C

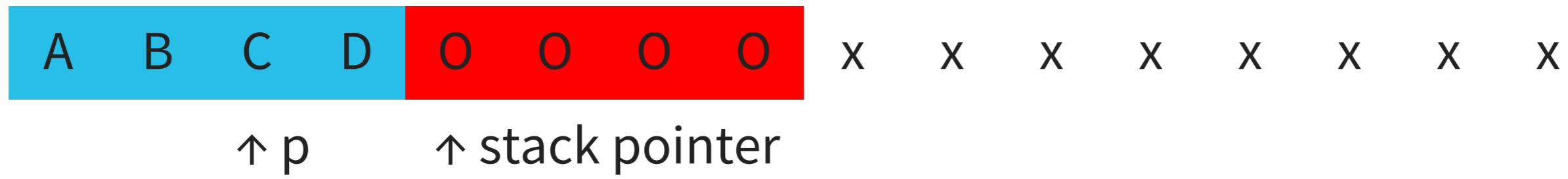
p = address of C

→ push D

push E

change C into C' using p

.



push A

push B

push C

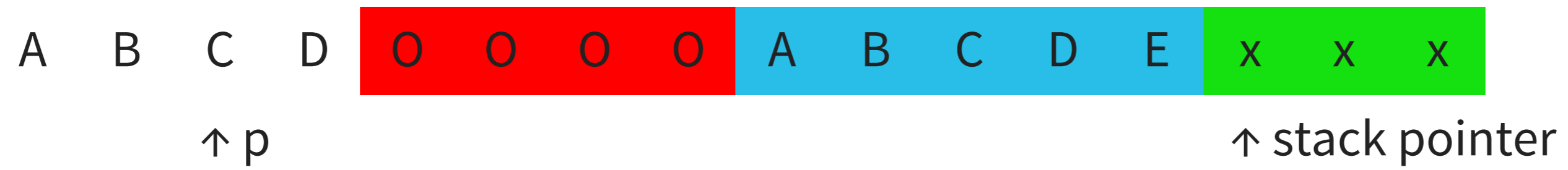
p = address of C

push D

→ push E

change C into C' using p

.



push A

push B

push C

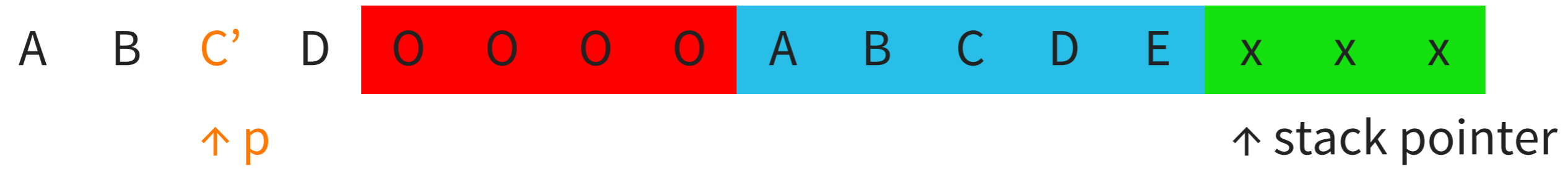
p = address of C

push D

push E

→ change C into C' using p

.



push A

push B

push C

p = address of C

push D

push E

change C into C' using p

→ .

Stack implementations

Implementation	Size	Requires allocations	Pointer stability
static array	constant	no	yes
dynamic array	can grow	when growing	no
linked list	can grow	every push	yes

Arena

- Known as arena allocator, region-based allocator, zone-based allocator, obstack
- Implemented as a list of static array stacks



↑ stack pointer

↑ block 0

- push A
- push B
- push C
- p = address of C
- push D
- push E
- change C into C' using p
- push F
- pop
- pop
- pop
- pop
- .

list of regions: block 0



↑ stack pointer

↑ block 0

push A

list of regions: block 0

→ push B

push C

p = address of C

push D

push E

change C into C' using p

push F

pop

pop

pop

pop

.



↑ stack pointer

↑ block 0

push A

list of regions: block 0

push B

→ push C

p = address of C

push D

push E

change C into C' using p

push F

pop

pop

pop

pop

.



↑ stack pointer

↑ block 0

push A

list of regions: block 0

push B

push C

→ p = address of C

push D

push E

change C into C' using p

push F

pop

pop

pop

pop

.



↑ ↑ stack pointer

↑ block 0

push A

list of regions: block 0

push B

push C

p = address of C

→ push D

push E

change C into C' using p

push F

pop

pop

pop

pop

.



push A list of regions: block 0

push B

push C

p = address of C

push D

→ push E

change C into C' using p

push F

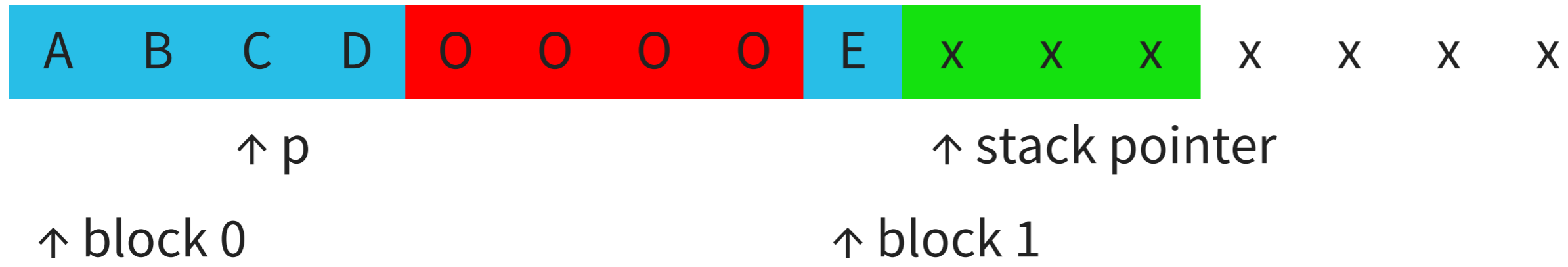
pop

pop

pop

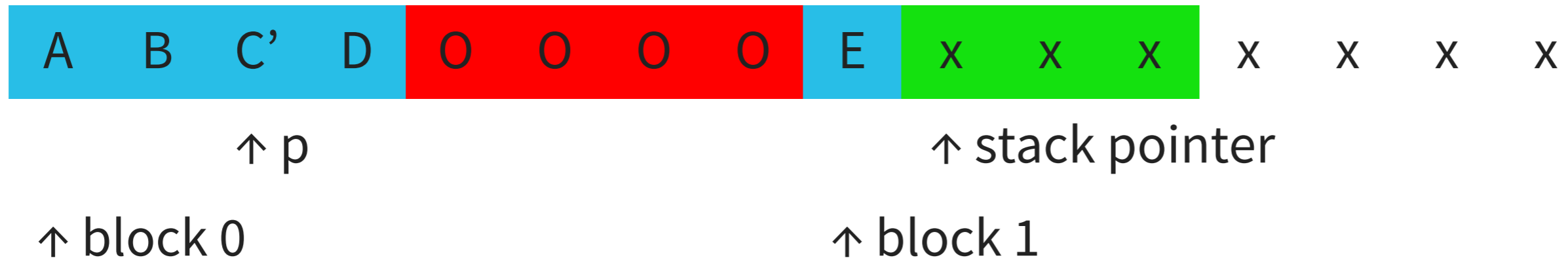
pop

.



push A
 push B
 push C
 p = address of C
 push D
 push E
 → change C into C' using p
 push F
 pop
 pop
 pop
 pop
 .

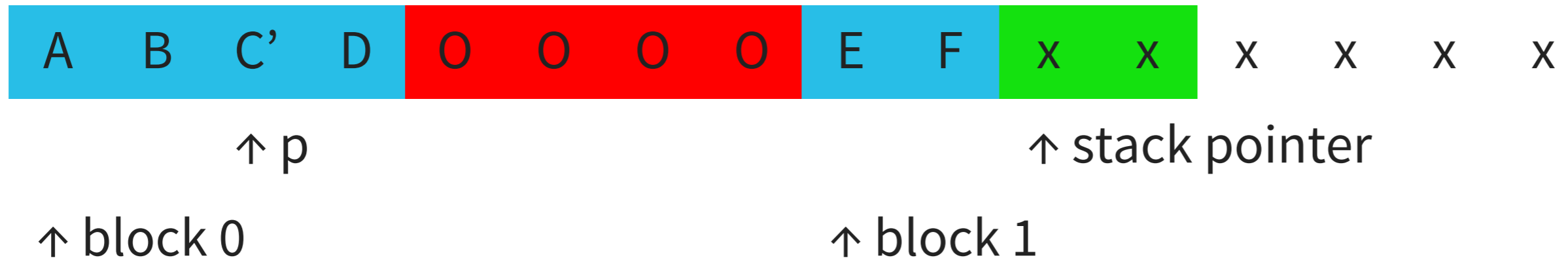
list of regions: block 0
block 1



push A
 push B
 push C
 p = address of C
 push D
 push E
 change C into C' using p

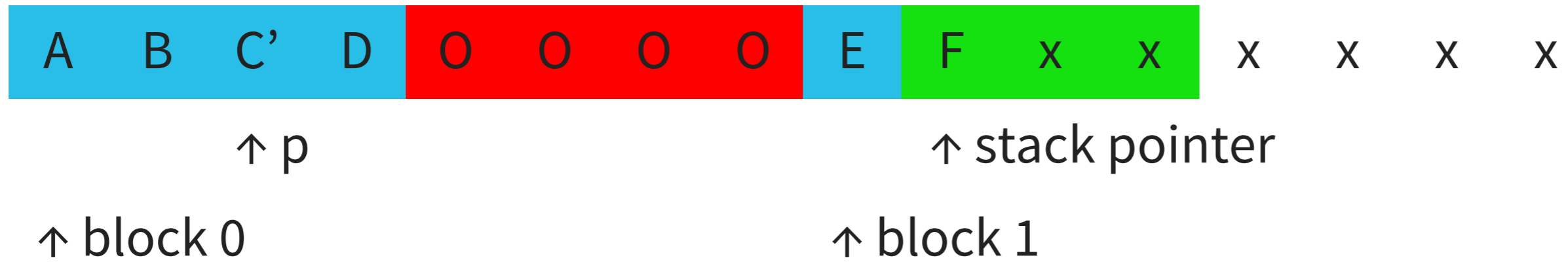
list of regions: block 0
block 1

→ push F
 pop
 pop
 pop
 pop
 .

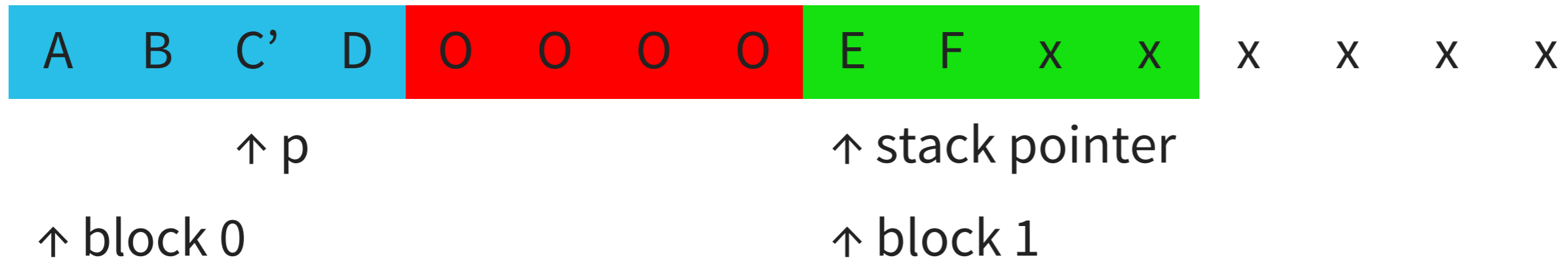


push A
 push B
 push C
 p = address of C
 push D
 push E
 change C into C' using p
 push F
 → pop
 pop
 pop
 pop
 .

list of regions: block 0
block 1



push A list of regions: block 0
 push B block 1
 push C
 p = address of C
 push D
 push E
 change C into C' using p
 push F
 pop → F
 → pop
 pop
 pop
 .



push A list of regions: block 0
 push B block 1
 push C
 p = address of C
 push D
 push E
 change C into C' using p
 push F
 pop → F
 pop → E
 → pop
 pop
 .



↑ ↑ stack pointer

↑ block 0

push A

list of regions: block 0

push B

push C

p = address of C

push D

push E

change C into C' using p

push F

pop → F

pop → E

pop → D

→ pop

.



↑ stack pointer

↑ block 0

push A

list of regions: block 0

push B

push C

p = address of C

push D

push E

change C into C' using p

push F

pop → F

pop → E

pop → D

pop → C'

→ .

Stack implementations

Implementation	Size	Requires allocations	Pointer stability
static array	constant	no	yes
dynamic array	can grow	when growing	no
linked list	can grow	every push	yes
arena	can grow	when growing	yes

More options

Combination of other data structures and indirection can be used, depending on desired properties.

Queues / FIFO

- A **queue** is an ordered collection of elements
 - supports two operations:
 - “enqueue”: add an element
 - “dequeue”: retrieve-and-remove the earliest-added element
- ⇒ **first in, first out (FIFO)**

Ring buffer implementation of a queue

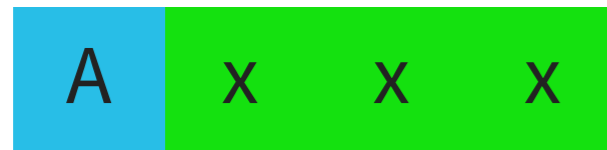
- Useful **only when there is a hard limit** on the number of elements
- We maintain a **static array**
- and two pointers/indices: **head** and **tail**

x x x x

↑ head

↑ tail

- enqueue A
- enqueue B
- dequeue
- dequeue
- enqueue C
- enqueue D
- enqueue E
- enqueue F
- dequeue
- dequeue
- dequeue
- enqueue G
- dequeue
- dequeue
- .



↑ head

↑ tail

enqueue A

→ enqueue B

dequeue

dequeue

enqueue C

enqueue D

enqueue E

enqueue F

dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

→ dequeue

dequeue

enqueue C

enqueue D

enqueue E

enqueue F

dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

→ dequeue

enqueue C

enqueue D

enqueue E

enqueue F

dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.

A B x x

↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

→ enqueue C

enqueue D

enqueue E

enqueue F

dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

→ enqueue D

enqueue E

enqueue F

dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

→ enqueue E

enqueue F

dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

→ enqueue F

dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.

E F C D

↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

enqueue F

→ dequeue

dequeue

dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

enqueue F

dequeue → C

→ dequeue

dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

enqueue F

dequeue → C

dequeue → D

→ dequeue

enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

enqueue F

dequeue → C

dequeue → D

dequeue → E

→ enqueue G

dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

enqueue F

dequeue → C

dequeue → D

dequeue → E

enqueue G

→ dequeue

dequeue

.



↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

enqueue F

dequeue → C

dequeue → D

dequeue → E

enqueue G

dequeue → F

→ dequeue

.

E F G D

↑ head

↑ tail

enqueue A

enqueue B

dequeue → A

dequeue → B

enqueue C

enqueue D

enqueue E

enqueue F

dequeue → C

dequeue → D

dequeue → E

enqueue G

dequeue → F

dequeue → G

→ .

Implementation detail

When `head == tail`:

the queue is empty?

E F G D

↑ head

↑ tail

or the queue is full?

E F C D

↑ head

↑ tail

... Solutions:

- maintain a variable with the number of elements currently in the queue
- or keep incrementing head and tail, and index the static array as
`array[head % size]` and
`array[tail % size]`

Applications of ring buffers

- Audio playback/recording devices
- Video capture devices
- Special case (double-buffering, i.e. size = 2) for computer graphics
- Network devices (routers, switches)

More options

- use dynamic arrays
- use linked lists
- use indirection
- ...

... depending on specific needs

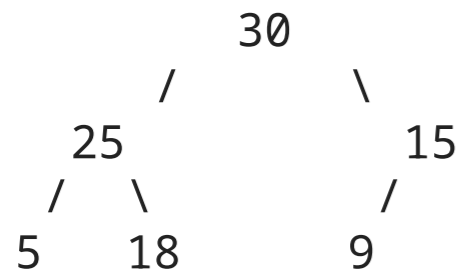
Priority queues

- A **priority queue** is a collection of elements, each with an associated **priority**
- supports two operations:
 - “push”: add an element-priority tuple
 - “pop”: retrieve-and-remove the highest-priority element

Implementation of a priority queue

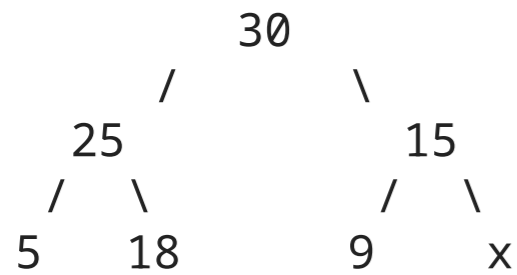
- Store element-priority tuples in an [array](#) or in a [linked list](#)
- “push”: $O(1)$ of the underlying data structure
- “pop”: scan all elements, find max priority, $O(n)$

Binary heap implementation of a **priority queue**



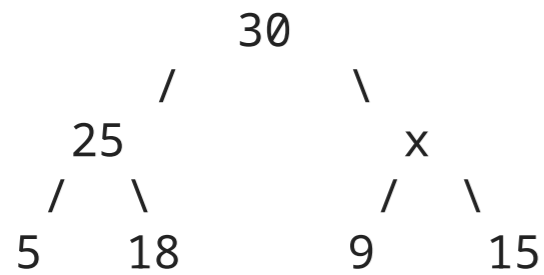
- **Binary heaps** represent a **priority queue** as
 - a binary tree (every node has at most two children)
 - that is complete (every level full, except possibly the deepest)
- Every node is labeled by the corresponding element's priority
- Tree has the **heap property**:
 - Priority of any node \geq priority of its children
 - \Leftrightarrow Priority of any node \geq priority of all its descendants

Binary heap push



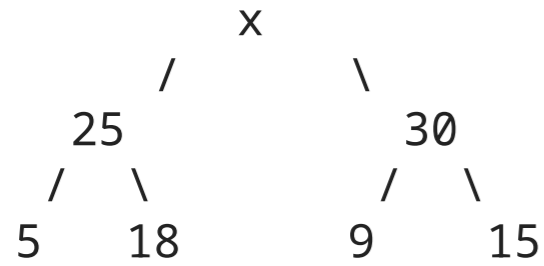
- Step 0: Add new element at the first free slot on the deepest level
- Step 1:
 - If its priority is **not higher** than its parent's,
 - the heap property is **satisfied**, we are done
 - If its priority is **higher** than its parent's,
 - swap them,
 - go back to Step 1, looking at the pushed element's new position

Binary heap push



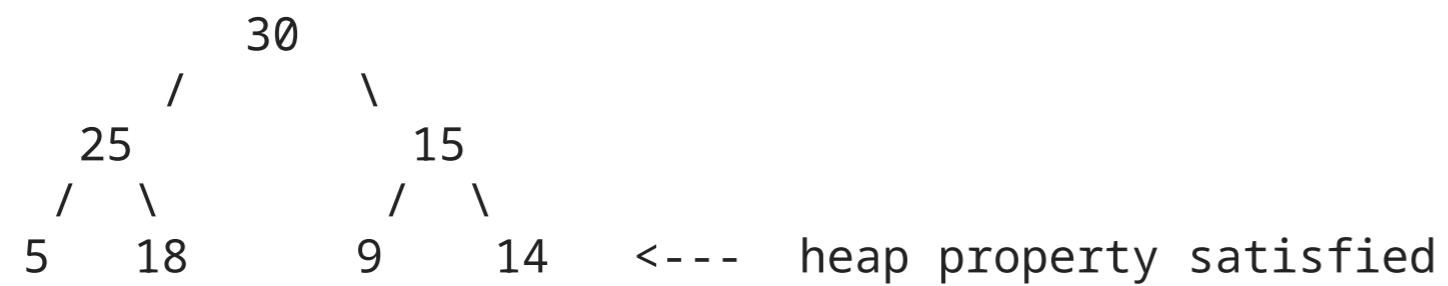
- Step 0: Add new element at the first free slot on the deepest level
- Step 1:
 - If its priority is **not higher** than its parent's,
 - the heap property is **satisfied**, we are done
 - If its priority is **higher** than its parent's,
 - swap them,
 - go back to Step 1, looking at the pushed element's new position

Binary heap push



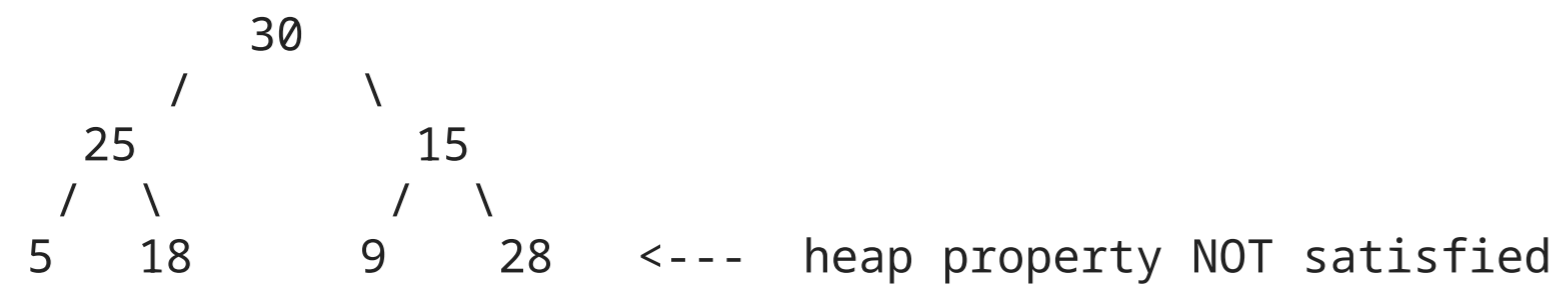
- Step 0: Add new element at the first free slot on the deepest level
- Step 1:
 - If its priority is **not higher** than its parent's,
 - the heap property is **satisfied**, we are done
 - If its priority is **higher** than its parent's,
 - swap them,
 - go back to Step 1, looking at the pushed element's new position

Binary heap push: 14



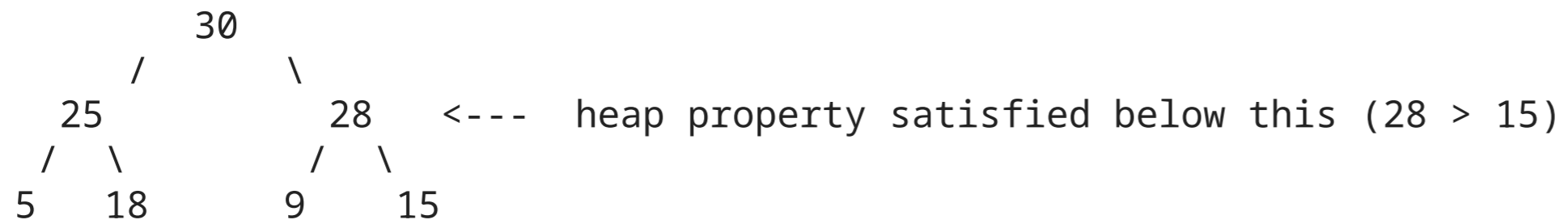
- Add new element at the first free slot on the deepest level
- If it is **not higher** than its parent's,
 - the heap property is **satisfied**, we are done

Binary heap push: 28



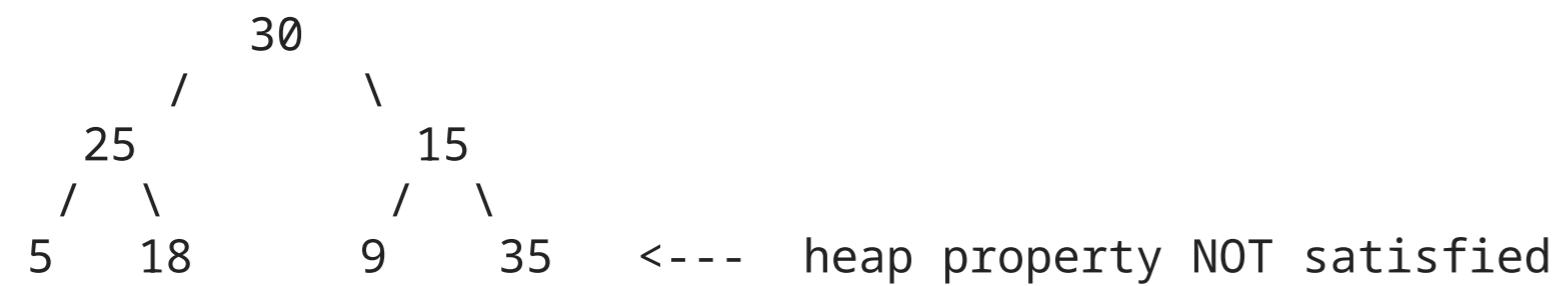
- Add new element at the first free slot on the deepest level
- If its priority is **higher** than its parent's,
 - swap them

Binary heap push: 28



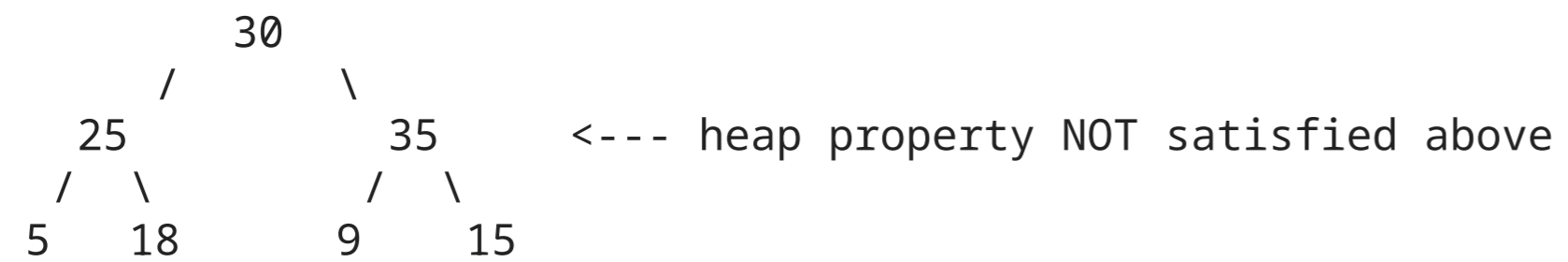
- Heap property **satisfied** below the pushed element's new position
- If its priority is **not higher** than its new parent's,
 - the heap property is **satisfied**, we are done

Binary heap push: 35



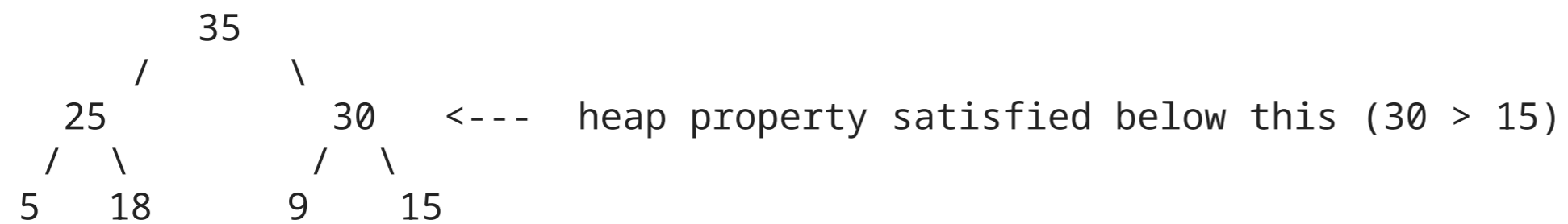
- Add new element at the first free slot on the deepest level
- If its priority is **higher** than its parent's,
 - swap them

Binary heap push: 35



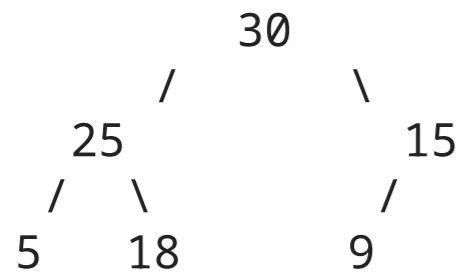
- Heap property **satisfied** below the pushed element's new position
- If its priority is **still higher** than its new parent's,
 - swap them

Binary heap push: 35



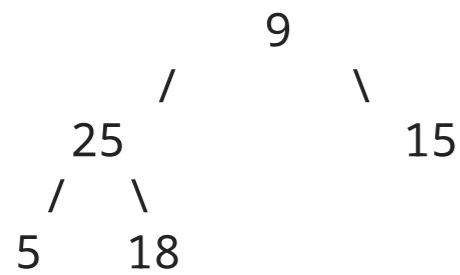
- Heap property **satisfied** below the pushed element's new position
 - new child was an ancestor of its direct children
- Continue until heap property is **satisfied**

Binary heap pop



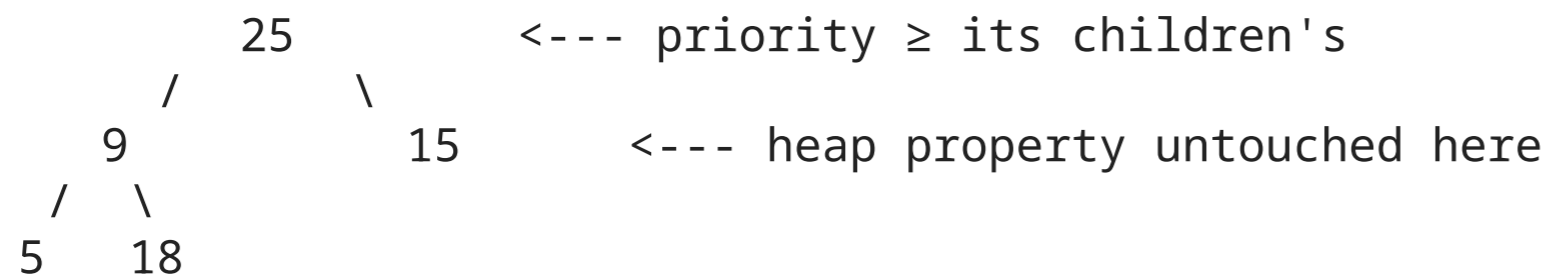
- Step 0: Replace root with last element (on deepest level)
- Step 1:
 - If its priority is **not lower** than its children's,
 - the heap property is **satisfied**, we are done
 - If its priority is **lower** than one of its children's,
 - swap with the highest-priority child,
 - go back to Step 1, looking at the pushed element's new position

Binary heap pop



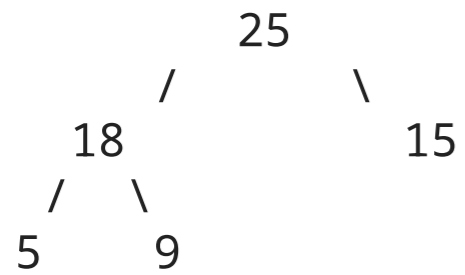
- Step 0: Replace root with last element (on deepest level)
- Step 1:
 - If its priority is **not lower** than its children's,
 - the heap property is **satisfied**, we are done
 - If its priority is **lower** than one of its children's,
 - swap with the highest-priority child,
 - go back to Step 1, looking at the pushed element's new position

Binary heap pop



- Step 0: Replace root with last element (on deepest level)
- Step 1:
 - If its priority is **not lower** than its children's,
 - the heap property is **satisfied**, we are done
 - If its priority is **lower** than one of its children's,
 - swap with the highest-priority child,
 - go back to Step 1, looking at the pushed element's new position

Binary heap pop



18 was a descendant of 25

- Step 0: Replace root with last element (on deepest level)
- Step 1:
 - If its priority is **not lower** than its children's,
 - the heap property is **satisfied**, we are done
 - If its priority is **lower** than one of its children's,
 - swap with the highest-priority child,
 - go back to Step 1, looking at the pushed element's new position

Binary heap operations

- Push: $O(\log_2(n))$
- Find max: $O(1)$
- Pop: $O(\log_2(n))$

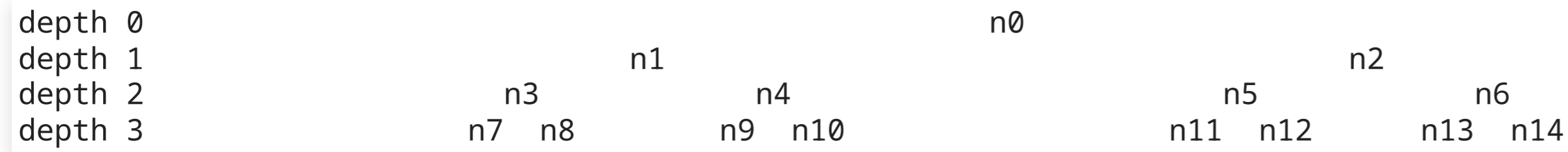
Representation of binary heaps

- Binary heaps are complete binary trees
- We can avoid allocation for every “push” by storing nodes in an array
- Depth ℓ of the tree has at most 2^ℓ nodes, $\forall \ell$
- Depth ℓ of the tree has exactly 2^ℓ nodes, except for the deepest level

depth	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- There are exactly $(2^\ell - 1)$ nodes of with depth $< \ell$

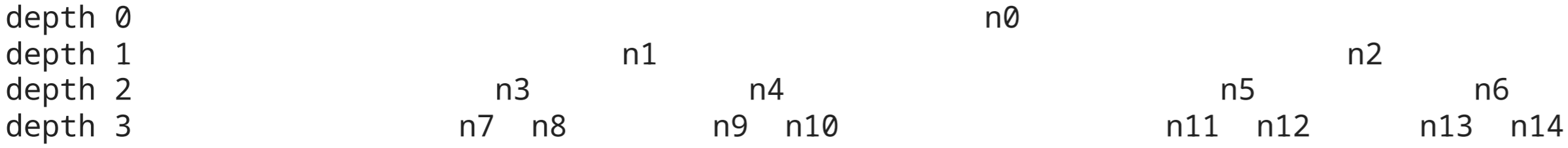
Storage scheme



depth	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- If a node has index j
- its children are stored at indices $2j + 1$ and $2j + 2$
- its parent is stored at index $\lfloor (j - 1) / 2 \rfloor$

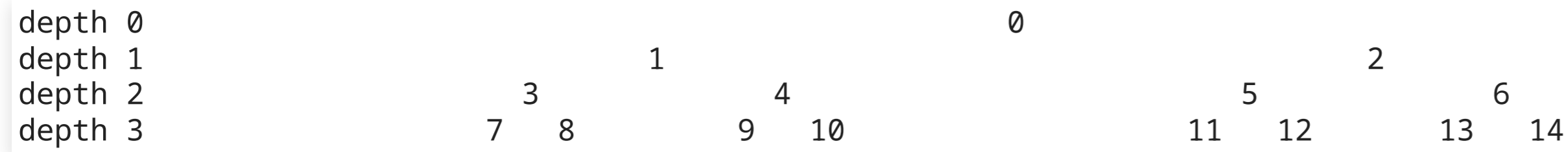
Storage scheme



depth	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- If a node has index j
- its children are stored at indices $2j + 1$ and $2j + 2$
- its parent is stored at index $\lfloor (j - 1) / 2 \rfloor$

Storage scheme



depth	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- If a node has index j
- its **children** are stored at indices $2j + 1$ and $2j + 2$
- its **parent** is stored at index $\lfloor (j - 1) / 2 \rfloor$

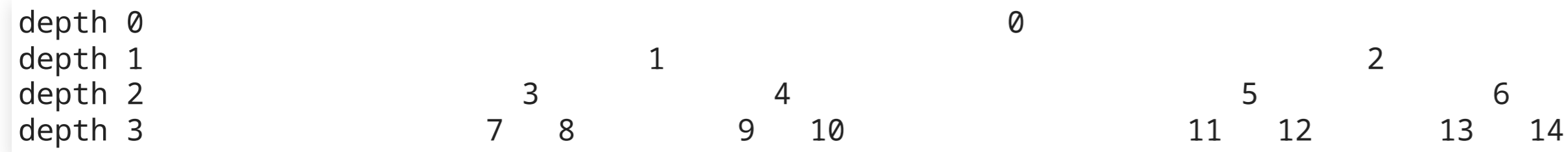
Storage scheme

depth 0					0									
depth 1			1					2						
depth 2		3		4			5		6					
depth 3	7	8		9	10		11	12		13	14			

depth	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- If a node has index j
- its **children** are stored at indices $2j + 1$ and $2j + 2$
- its **parent** is stored at index $\lfloor (j - 1) / 2 \rfloor$

Storage scheme



depth	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- If a node has index j
- its **children** are stored at indices $2j + 1$ and $2j + 2$
- its **parent** is stored at index $\lfloor (j - 1) / 2 \rfloor$

Binary heap with array storage

- Superior to *in-memory tree* (with pointers)
 - We avoid allocation for every “push”
 - We avoid data dependencies (load node data to get pointer to parent/children)
- Still,
 - Push and pop operations are tough for branch predictor
 - Jumps to indices $(2j + 1)$, $(2j + 2)$ or $\lfloor (j - 1)/2 \rfloor$
not cache-friendly for large j

Priority queue: special case

- Assume that
 - priorities are distinct integers $p \in \{0, \dots, P - 1\}$
 - we always push at a priority \leq last popped priority (initially ∞)
- Then,
 - we allocate a **static array** of size P
 - Push: store in **array** at index p $O(1)$
 - Pop: sweep **array** backwards $O(P/n)$ amortized

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element															
max															↑

→ push A, 12
push B, 5
push C, 9
pop
push E, 3
push F, 10
pop
push G, 4
pop
pop
pop
push H, 1
pop
pop
.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element													A		
max															↑

push A, 12

→ push B, 5

push C, 9

pop

push E, 3

push F, 10

pop

push G, 4

pop

pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element						B							A		
max															↑

- push A, 12
- push B, 5
- push C, 9
- pop
- push E, 3
- push F, 10
- pop
- push G, 4
- pop
- pop
- pop
- push H, 1
- pop
- pop
- .

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element						B				C			A		
max															↑

push A, 12

push B, 5

push C, 9

→ pop

push E, 3

push F, 10

pop

push G, 4

pop

pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element						B				C			A		
max													↑		

push A, 12

push B, 5

push C, 9

pop → A

→ push E, 3

push F, 10

pop

push G, 4

pop

pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element				E		B				C			A		
max													↑		

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

→ push F, 10

pop

push G, 4

pop

pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element				E		B				C	F		A		
max													↑		

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

→ pop

push G, 4

pop

pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element				E		B				C	F		A		
max											↑				

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

→ push G, 4

pop

pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element				E	G	B				C	F		A		
max											↑				

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

push G, 4

→ pop

pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element				E	G	B				C	F		A		
max										↑					

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

push G, 4

pop → C

→ pop

pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element				E	G	B				C	F		A		
max						↑									

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

push G, 4

pop → C

pop → B

→ pop

push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element				E	G	B				C	F		A		
max					↑										

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

push G, 4

pop → C

pop → B

pop → G

→ push H, 1

pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element		H		E	G	B				C	F		A		
max					↑										

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

push G, 4

pop → C

pop → B

pop → G

push H, 1

→ pop

pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element		H		E	G	B				C	F		A		
max				↑											

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

push G, 4

pop → C

pop → B

pop → G

push H, 1

pop → E

→ pop

.

priority	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
element		H		E	G	B				C	F		A		
max		↑													

push A, 12

push B, 5

push C, 9

pop → A

push E, 3

push F, 10

pop → F

push G, 4

pop → C

pop → B

pop → G

push H, 1

pop → E

pop → H

→ .

Implementation details

- good for branch predictor
- great for caches
- we can store, additionally, an array of P bits (“bitmap”)
 - bit p set to one if there is an element with priority p
 - makes “pop” operations essentially 64x faster

Applications of **bitmap priority queues**

- Linux kernel: scheduling parallel tasks
- Linear algebra: sparse matrices

Sort operations

Heap sort

- Push n elements to heap: $O(n \log n)$
- Pop n elements one by one: $O(n \log n)$

$\Rightarrow O(n \log n)$ worst case

Comparison sort methods

Method	Average	Worst case	Additional storage	Combines with insert. sort
Quicksort	$O(n \log(n))$	$O(n^2)$	none	yes
Merge sort	$O(n \log(n))$	$O(n \log(n))$	n	yes
Heap sort	$O(n \log(n))$	$O(n \log(n))$	none	no

Special case 1

- Assume that
 - we sort n elements with priorities $S \subseteq \{0, \dots, P - 1\}$
 - no two elements have the same priority (hence $P \geq n$)
- Then,
 - we represent the elements as a **bitmap priority queue**
 - Push: $O(n)$
 - Pop: $O(P)$

$$\Rightarrow O(n + P)$$

Special case 2: counting sort

- Assume that
 - we sort n elements with priorities $\mathcal{S} \subseteq \{0, \dots, P - 1\}$
 - $P \leq n$ (we may have duplicates)
- Then,
 - we allocate a **static array** count of size P
 - we allocate a **static array** result of size n
 - we count the number of occurrences of each priority: $O(n)$
 - we sweep count backwards to determine offsets: $O(P) = O(n)$
 - we construct the sorted result list: $O(n)$

$\Rightarrow O(n)$

Count

priority	3	6	1	3	3	7	5	6	1	6
↑										
count index	0	1	2	3	4	5	6	7		
count value	0	0	0	0	0	0	0	0		
↑										
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
	↑									
count index	0	1	2	3	4	5	6	7		
count value	0	0	0	1	0	0	0	0		
				↑						
result index	0	1	2	3	4	5	6	7	8	9
result										
	↑									

Count

priority	3	6	1	3	3	7	5	6	1	6
		↑								
count index	0	1	2	3	4	5	6	7		
count value	0	0	0	1	0	0	1	0		
							↑			
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
			↑							
count index	0	1	2	3	4	5	6	7		
count value	0	1	0	1	0	0	1	0		
		↑								
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
				↑						
count index	0	1	2	3	4	5	6	7		
count value	0	1	0	2	0	0	1	0		
				↑						
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
					↑					
count index	0	1	2	3	4	5	6	7		
count value	0	1	0	3	0	0	1	0		
				↑						
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
						↑				
count index	0	1	2	3	4	5	6	7		
count value	0	1	0	3	0	0	1	1		
								↑		
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
							↑			
count index	0	1	2	3	4	5	6	7		
count value	0	1	0	3	0	1	1	1		
						↑				
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
								↑		
count index	0	1	2	3	4	5	6	7		
count value	0	1	0	3	0	1	2	1		
							↑			
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Count

priority	3	6	1	3	3	7	5	6	1	6
									↑	
count index	0	1	2	3	4	5	6	7		
count value	0	2	0	3	0	1	2	1		
		↑								
result index	0	1	2	3	4	5	6	7	8	9
result										
		↑								

Count

priority	3	6	1	3	3	7	5	6	1	6
										↑
count index	0	1	2	3	4	5	6	7		
count value	0	2	0	3	0	1	3	1		
							↑			
result index	0	1	2	3	4	5	6	7	8	9
result										
										↑

Determine offsets

priority	3	6	1	3	3	7	5	6	1	6	
											↑
count index	0	1	2	3	4	5	6	7			
count value	0	2	0	3	0	1	3	1			
								↑			
result index	0	1	2	3	4	5	6	7	8	9	
result											
											↑

Determine offsets

priority	3	6	1	3	3	7	5	6	1	6
----------	---	---	---	---	---	---	---	---	---	---

↑

count index	0	1	2	3	4	5	6	7
-------------	---	---	---	---	---	---	---	---

count value	0	2	0	3	0	1	4	1
-------------	---	---	---	---	---	---	---	---

↑

result index	0	1	2	3	4	5	6	7	8	9
--------------	---	---	---	---	---	---	---	---	---	---

result

↑

Determine offsets

priority 3 6 1 3 3 7 5 6 1 6

↑

count index 0 1 2 3 4 5 6 7

count value 0 2 0 3 0 5 4 1

↑

result index 0 1 2 3 4 5 6 7 8 9

result

↑

Determine offsets

priority	3	6	1	3	3	7	5	6	1	6
----------	---	---	---	---	---	---	---	---	---	---

↑

count index	0	1	2	3	4	5	6	7
-------------	---	---	---	---	---	---	---	---

count value	0	2	0	3	5	5	4	1
-------------	---	---	---	---	---	---	---	---

↑

result index	0	1	2	3	4	5	6	7	8	9
--------------	---	---	---	---	---	---	---	---	---	---

result

↑

Determine offsets

priority	3	6	1	3	3	7	5	6	1	6
----------	---	---	---	---	---	---	---	---	---	---

↑

count index	0	1	2	3	4	5	6	7
-------------	---	---	---	---	---	---	---	---

count value	0	2	0	8	5	5	4	1
-------------	---	---	---	---	---	---	---	---

↑

result index	0	1	2	3	4	5	6	7	8	9
--------------	---	---	---	---	---	---	---	---	---	---

result

↑

Determine offsets

priority	3	6	1	3	3	7	5	6	1	6
----------	---	---	---	---	---	---	---	---	---	---

↑

count index	0	1	2	3	4	5	6	7
-------------	---	---	---	---	---	---	---	---

count value	0	2	8	8	5	5	4	1
-------------	---	---	---	---	---	---	---	---

↑

result index	0	1	2	3	4	5	6	7	8	9
--------------	---	---	---	---	---	---	---	---	---	---

result

↑

Determine offsets

priority	3	6	1	3	3	7	5	6	1	6	
											↑
count index	0	1	2	3	4	5	6	7			
count value	0	10	8	8	5	5	4	1			
		↑									
result index	0	1	2	3	4	5	6	7	8	9	
result											
											↑

Determine offsets

priority	3	6	1	3	3	7	5	6	1	6
----------	---	---	---	---	---	---	---	---	---	---

↑

count index	0	1	2	3	4	5	6	7
-------------	---	---	---	---	---	---	---	---

count value	10	10	8	8	5	5	4	1
-------------	----	----	---	---	---	---	---	---

↑

result index	0	1	2	3	4	5	6	7	8	9
--------------	---	---	---	---	---	---	---	---	---	---

result

↑

Construct result

priority	3	6	1	3	3	7	5	6	1	6
↑										
count index	0	1	2	3	4	5	6	7		
count value	10	10	8	8	5	5	4	1		
↑										
result index	0	1	2	3	4	5	6	7	8	9
result										
↑										

Construct result

priority	3	6	1	3	3	7	5	6	1	6
	↑									
count index	0	1	2	3	4	5	6	7		
count value	10	10	8	7	5	5	4	1		
				↑						
result index	0	1	2	3	4	5	6	7	8	9
result								3		
								↑		

Construct result

priority	3	6	1	3	3	7	5	6	1	6
		↑								
count index	0	1	2	3	4	5	6	7		
count value	10	10	8	7	5	5	3	1		
							↑			
result index	0	1	2	3	4	5	6	7	8	9
result				6				3		
			↑							

Construct result

priority	3	6	1	3	3	7	5	6	1	6
			↑							
count index	0	1	2	3	4	5	6	7		
count value	10	9	8	7	5	5	3	1		
		↑								
result index	0	1	2	3	4	5	6	7	8	9
result				6				3		1
										↑

Construct result

priority	3	6	1	3	3	7	5	6	1	6
				↑						
count index	0	1	2	3	4	5	6	7		
count value	10	9	8	6	5	5	3	1		
				↑						
result index	0	1	2	3	4	5	6	7	8	9
result				6			3	3		1
							↑			

Construct result

priority	3	6	1	3	3	7	5	6	1	6
					↑					
count index	0	1	2	3	4	5	6	7		
count value	10	9	8	5	5	5	3	1		
				↑						
result index	0	1	2	3	4	5	6	7	8	9
result				6		3	3	3		1
						↑				

Construct result

priority	3	6	1	3	3	7	5	6	1	6
						↑				
count index	0	1	2	3	4	5	6	7		
count value	10	9	8	5	5	5	3	0		
								↑		
result index	0	1	2	3	4	5	6	7	8	9
result	7			6		3	3	3		1
	↑									

Construct result

priority	3	6	1	3	3	7	5	6	1	6
							↑			
count index	0	1	2	3	4	5	6	7		
count value	10	9	8	5	5	4	3	0		
						↑				
result index	0	1	2	3	4	5	6	7	8	9
result	7			6	5	3	3	3		1
				↑						

Construct result

priority	3	6	1	3	3	7	5	6	1	6
								↑		
count index	0	1	2	3	4	5	6	7		
count value	10	9	8	5	5	4	2	0		
							↑			
result index	0	1	2	3	4	5	6	7	8	9
result	7		6	6	5	3	3	3		1
			↑							

Construct result

priority	3	6	1	3	3	7	5	6	1	6
									↑	
count index	0	1	2	3	4	5	6	7		
count value	10	8	8	5	5	4	2	0		
		↑								
result index	0	1	2	3	4	5	6	7	8	9
result	7		6	6	5	3	3	3	1	1
									↑	

Construct result

priority	3	6	1	3	3	7	5	6	1	6
										↑
count index	0	1	2	3	4	5	6	7		
count value	10	8	8	5	5	4	1	0		
							↑			
result index	0	1	2	3	4	5	6	7	8	9
result	7	6	6	6	5	3	3	3	1	1
		↑								

Result

priority 3 6 1 3 3 7 5 6 1 6

↑

count index 0 1 2 3 4 5 6 7

count value 10 8 8 5 5 4 1 0

↑

result index 0 1 2 3 4 5 6 7 8 9

result 7 6 6 6 5 3 3 3 1 1

↑

