Algorithm Design and Analysis

David N. JANSEN, Bohua ZHAN
名 姓

算法设计与分析

詹博华，杨大卫

This week's content

- Today Wednesday:
  - Recap Simple Sort Algorithms
  - Chapter 6: Heapsort and
    Priority Queues
  - Exercises
- Tomorrow Thursday:
  - Exercise solutions
  - Chapter 7: Quicksort
  - Chapter 8: Linear sorting

这周的内容

- 今天周三：
  - 复习简单的排序算法
  - 第6章：堆排序，
    优先队列
  - 练习
- 明天周四：
  - 练习题解答
  - 第7章：快速排序
  - 第8章：线性时间排序

Exercises

练习

4.3-7

- Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/3) + n$ is $T(n) = \Theta(n^{\log_3 4})$. Show that a substitution proof with the assumption $T(n) \le cn^{\log_3 4}$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

- 使用4.5节中的主方法，可以证明 $T(n) = 4T(n/3) + n$ 的解为 $T(n) = \Theta(n^{\log_3 4})$。说明基于假设 $T(n) \le cn^{\log_3 4}$ 的代入法不能证明这一结论。然后说明如何通过减去一个低阶项完成代入法证明。

4.3-7

- Using $T(n) \leq cn^{\log_3 4}$ it is not possible to prove the upper bound of the recurrence solution:
  - We attempt a strong induction proof.
  - Induction base: $T(1) \leq 1$ holds if $c \geq 1$.
  - Induction step: Assume $T(m) \leq cm^{\log_3 4}$ for all $m \leq n$. We have to prove $T(n+1) \leq c(n+1)^{\log_3 4}$.
    But $T(n+1) = 4T(\lfloor (n+1)/3 \rfloor) + n+1 \leq 4c((n+1)/3)^{\log_3 4} + n+1$
    $= 4c(n+1)^{\log_3 4} / 3^{\log_3 4} + n+1$
    $= c(n+1)^{\log_3 4} + n+1.$
    We can only prove the required inequality if $n+1 \leq 0$, which is false.

4.3-7

- Using $T(n) \le cn^{\log_3 4}$ it is not possible to prove the upper bound of the recurrence solution:
  - We attempt a strong induction proof.
  - Induction base: $T(1) \le 1$ holds if $c \ge 1$.
  - Induction step: Assume $T(m) \le cm^{\log_3 4}$ for all $m \le n$. We have to prove $T(n+1) \le c(n+1)^{\log_3 4}$.
    But $T(n+1) = 4T(\lfloor (n+1)/3 \rfloor) + n+1 \le 4c((n+1)/3)^{\log_3 4} + n+1$
    $= 4c(n+1)^{\log_3 4} / 3^{\log_3 4} + n+1$
    $= c(n+1)^{\log_3 4} + n+1.$
    We can only prove the required inequality if $n+1 \le 0$, which is false.

4.3-7

- Using $T(n) \leq cn^{\log_3 4}$ it is not possible to prove the upper bound of the recurrence solution:
  - We need to prove $T(6) \leq c\ 6^{\log_3 4} = c\ (2 \times 3)^{\log_3 4} = c\ 2^{\log_3 4} \times 3^{\log_3 4} = c\ 2^{\log_3 4} \times 4$
  - But if we use $T(2) \leq c\ 2^{\log_3 4}$, then we can only prove $T(6) \leq 4T(2) + 6 \leq 4\ c\ 2^{\log_3 4} + 6$, which is not $\leq c\ 2^{\log_3 4} \times 4$.

4.3-7

- Now try using $T(n) \le cn^{\log_3 4} + dn$:
  - We prove the inequation by strong induction.
  - Induction base: $T(1) \le 1$ holds if $c + d \ge 1$.
  - Induction step: Assume $T(m) \le cm^{\log_3 4} + dm$ for all $m \le n$. We have to prove $T(n+1) \le c(n+1)^{\log_3 4} + d(n+1)$.
    But
$$
\begin{aligned}
T(n+1) = 4T(\lfloor (n+1)/3 \rfloor) + n+1 &\le 4c((n+1)/3)^{\log_3 4} + 4d\lfloor (n+1)/3 \rfloor + n+1 \\
&\le 4c(n+1)^{\log_3 4} / 3^{\log_3 4} + 4d(n+1)/3 + n+1 \\
&= c(n+1)^{\log_3 4} + 4d(n+1)/3 + n+1 \\
&= c(n+1)^{\log_3 4} + d(n+1) + \underbrace{d(n+1)/3 + n+1}_{\text{needs to be } \le 0}
\end{aligned}
$$

4.3-7

- It remains to be proven: $d(n+1)/3 + n+1 \leq 0$.
  iff $n+1 \leq -d(n+1)/3$
  iff $3(n+1) \leq -d(n+1)$
  iff $3 \leq -d$
  iff $-3 \geq d$

- To satisfy $c + d \geq 1$ and $d \leq -3$, we choose $c = 4$ and $d = -3$.

4.3-9

- Solve the recurrence

  $T(n) = 3T(\sqrt{n}) + \lg n$

  by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integral.

- 使用改变变量的方法求解归式

  $T(n) = 3T(\sqrt{n}) + \lg n$

  你的解应该是渐近紧确的。不必担心数值是否是整数。

4.3-9

- In $T(n) = 3T(\sqrt{n}) + \lg n$,
  try to find a substitution $T'(m) = T(g(m))$
  such that the recurrence becomes $T'(m) = aT'(m/b) + \ldots$ for some a and b.

- So we need a function g that satisfies: If $n = g(m)$, then $\sqrt{n} = g(m/b)$.
  Possible solution: $g(m) = 2^m$. Then $b = 2$ because $\sqrt{n} = \sqrt{g(m)} = \sqrt{2^m} = 2^{m/2}$.

- $T'(m) = 3T'(m/2) + m$

  The recurrence for T' can be solved normally; for example, by the master method, $a = 3$ and $b = 2$, $f(m) = m = O(m^{\log_b a - \varepsilon})$ for $\varepsilon = 0.58$, so the first case applies, and $T'(m) = O(m^{\lg 3})$.

  Therefore, $T(n) = T(2^m) = O(m^{\lg 3}) = O((\lg n)^{\lg 3}) = O(3^{\lg \lg n})$.

6.4-1

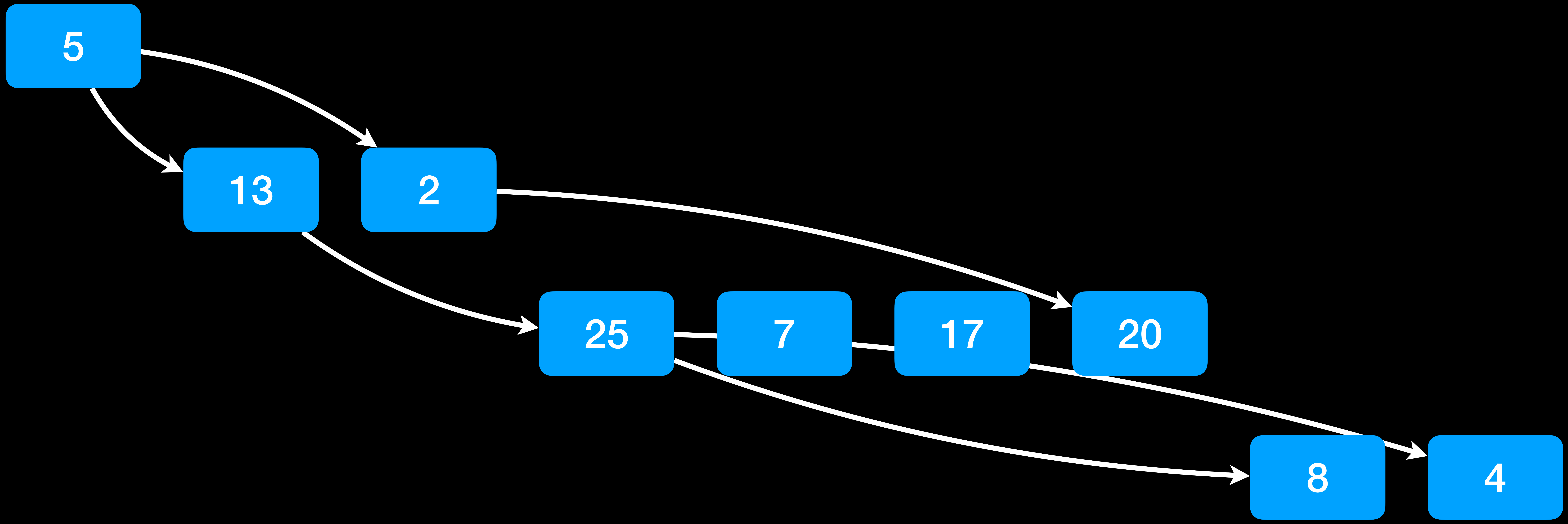
- Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

- 参照图6-4的方法，说明HEAPSORT在数组$A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$上的操作过程。

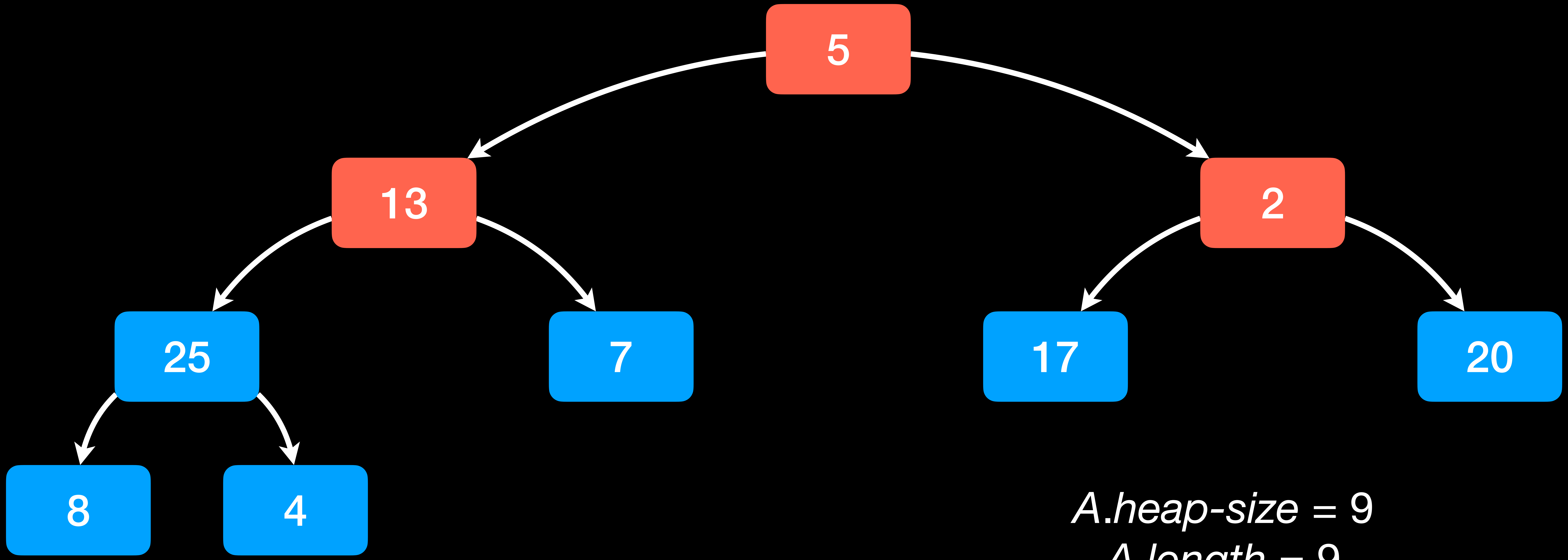
Initial array

5	13	2	25	7	17	20	8	4

Initial heap

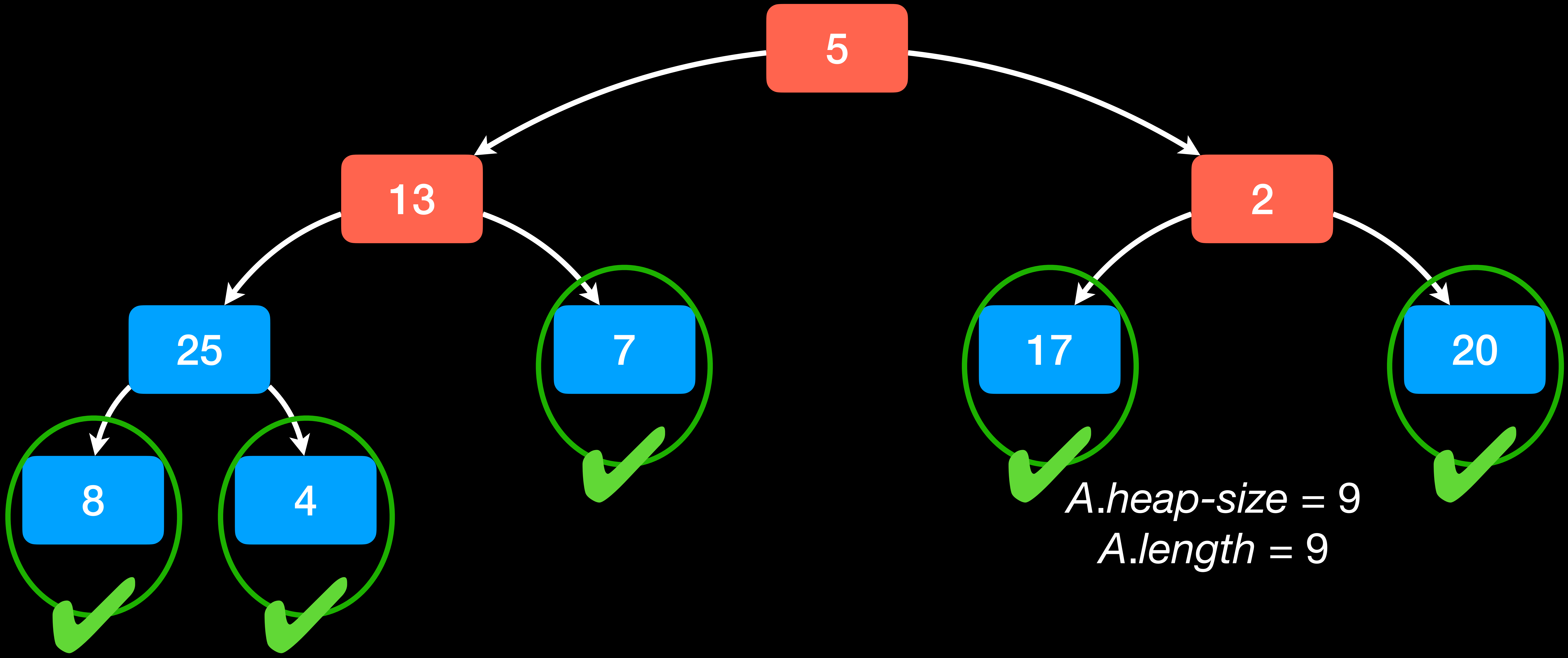


Initial heap



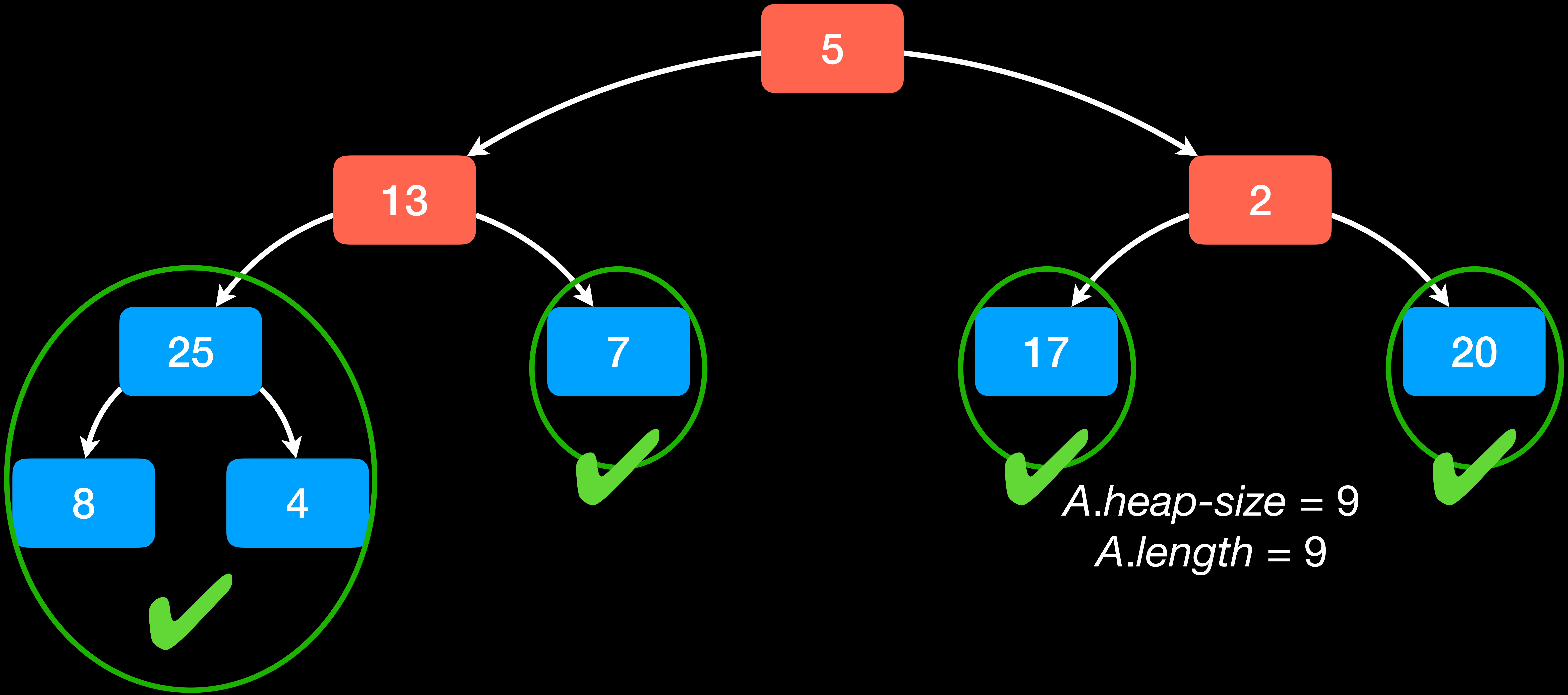
A.heap-size = 9
A.length = 9

BUILD-MAX-HEAP



A.heap-size = 9
A.length = 9

BUILD-MAX-HEAP

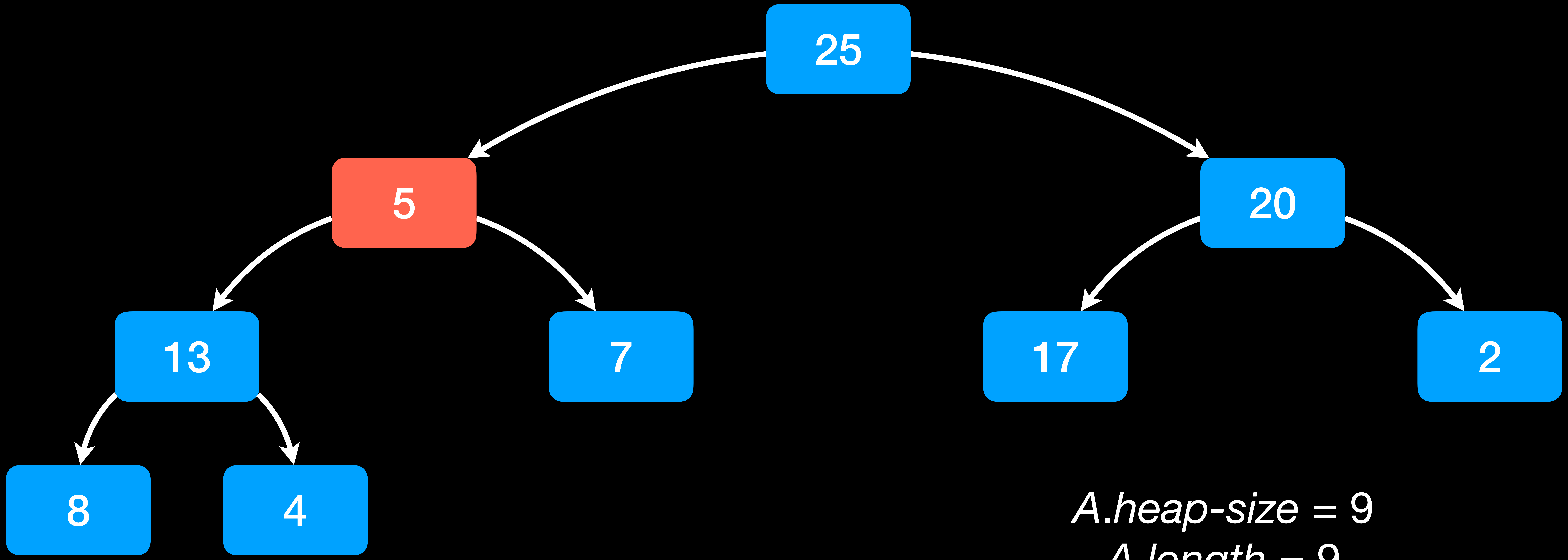


A.heap-size = 9
A.length = 9

Build-Max-Heap

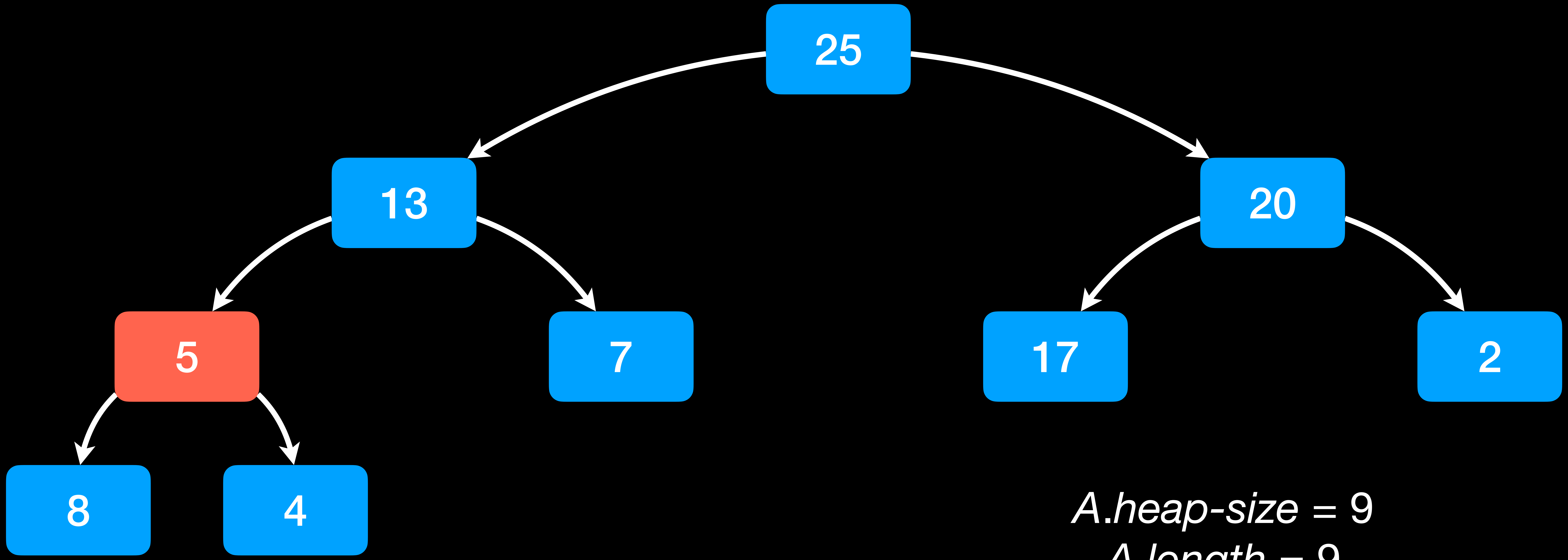
BUILD-MAX-HEAP

BUILD-MAX-HEAP



A.heap-size = 9
A.length = 9

BUILD-MAX-HEAP



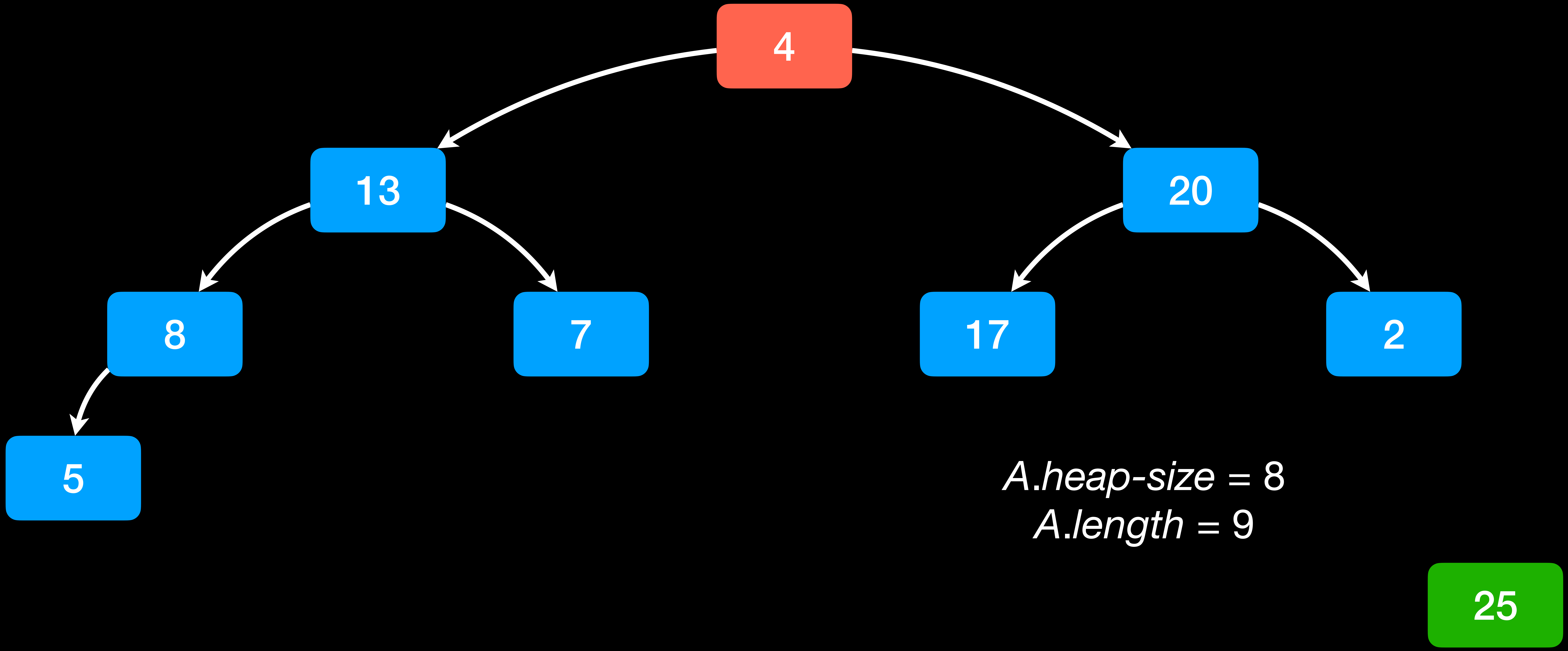
A.heap-size = 9
A.length = 9

BUILD-MAX-HEAP

A.heap-size = 9
A.length = 9

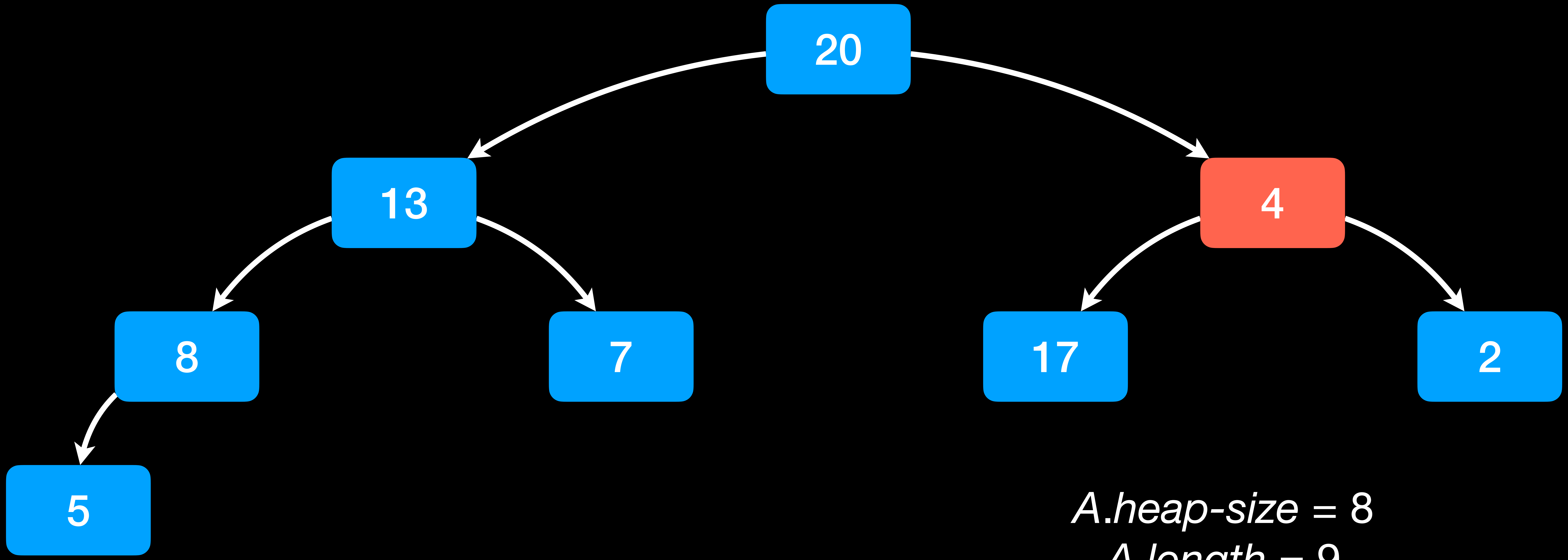
✔

Extract one element



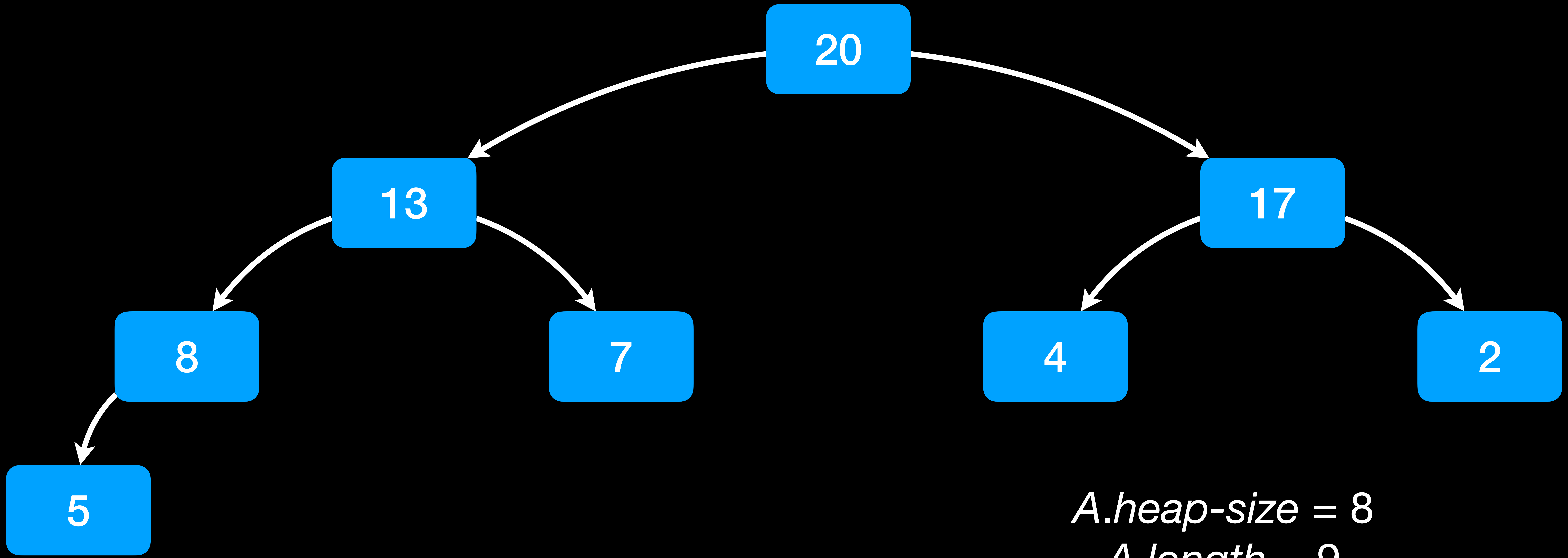
A.heap-size = 8
A.length = 9

MAX-HEAPIFY



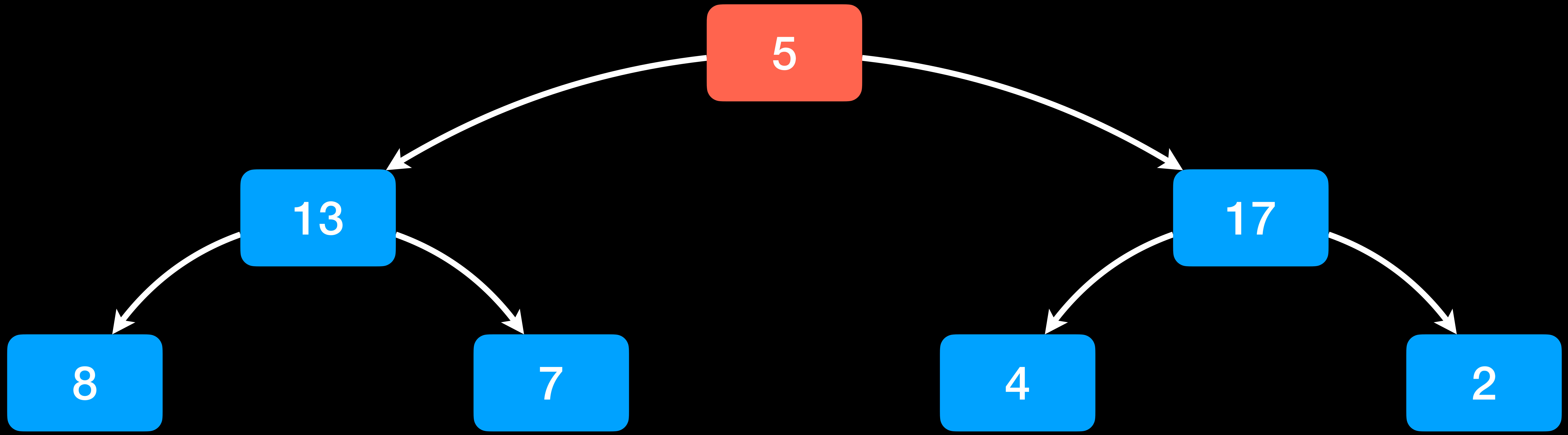
A.heap-size = 8
A.length = 9

MAX-HEAPIFY

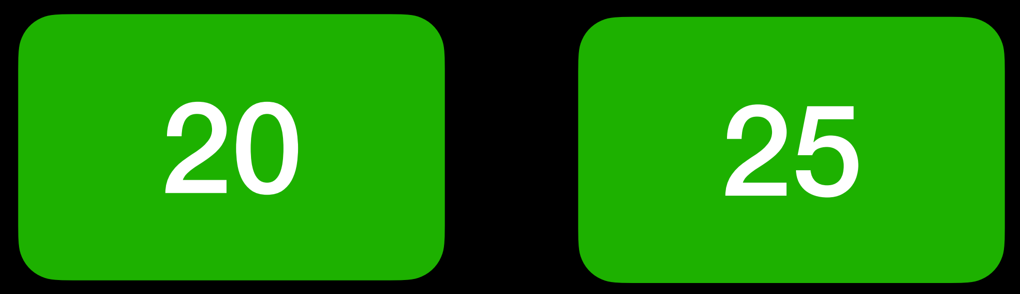


$A.heap\text{-}size = 8$
$A.length = 9$

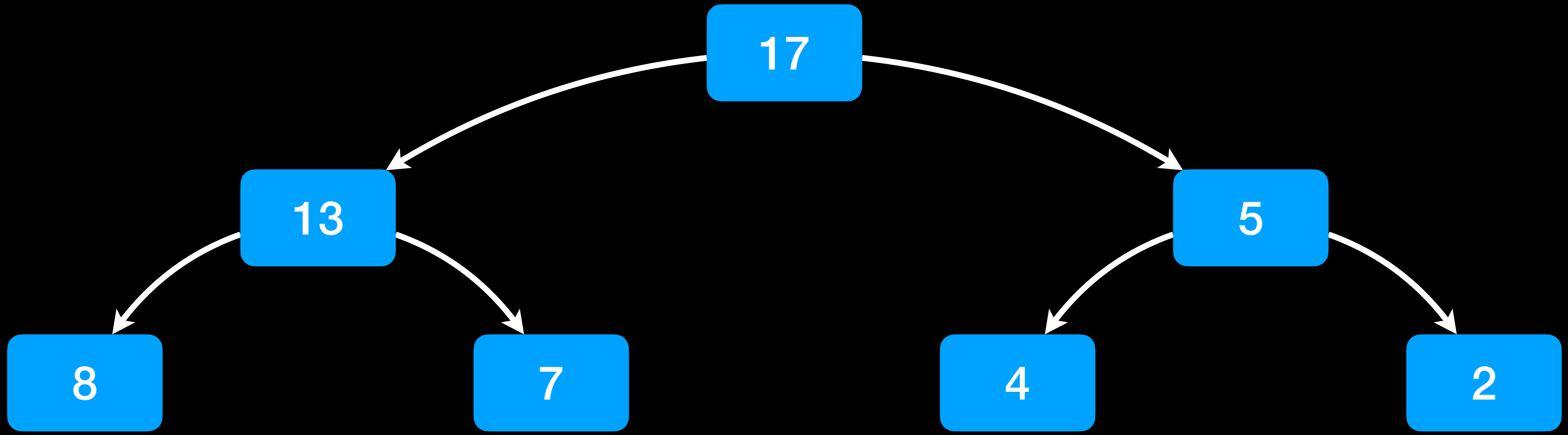
Extract one element



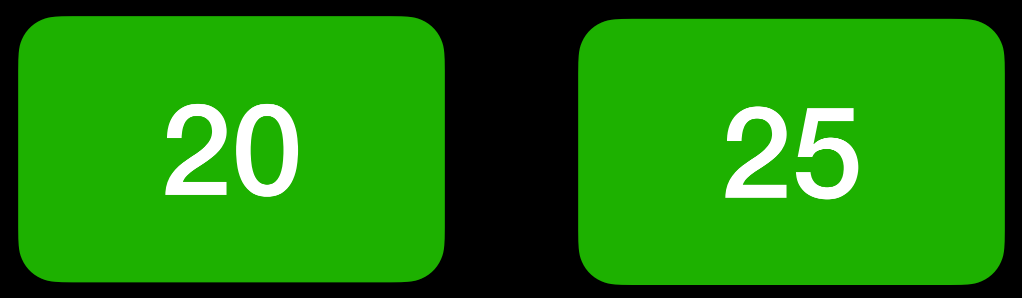
$A.heap\text{-}size = 7$
$A.length = 9$



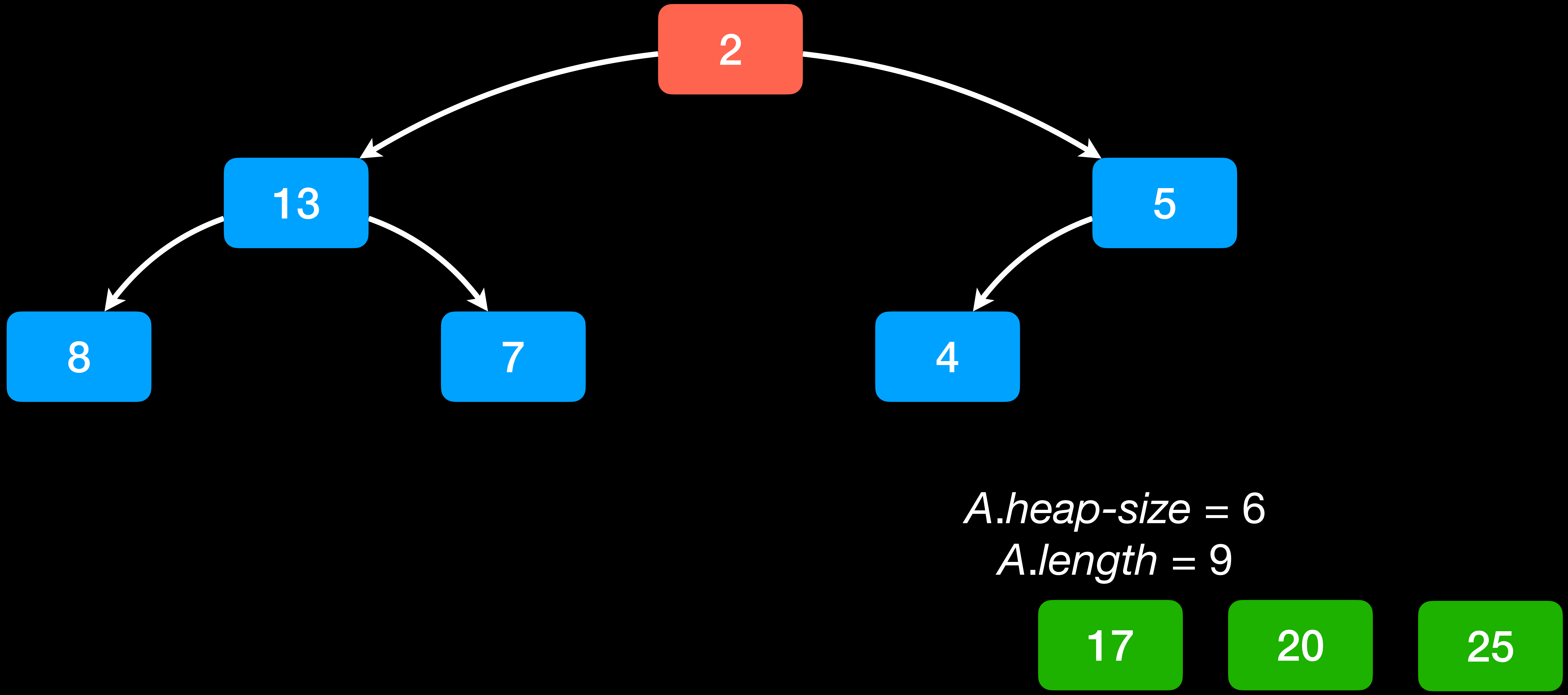
Max-Heapify



A.heap-size = 7
A.length = 9

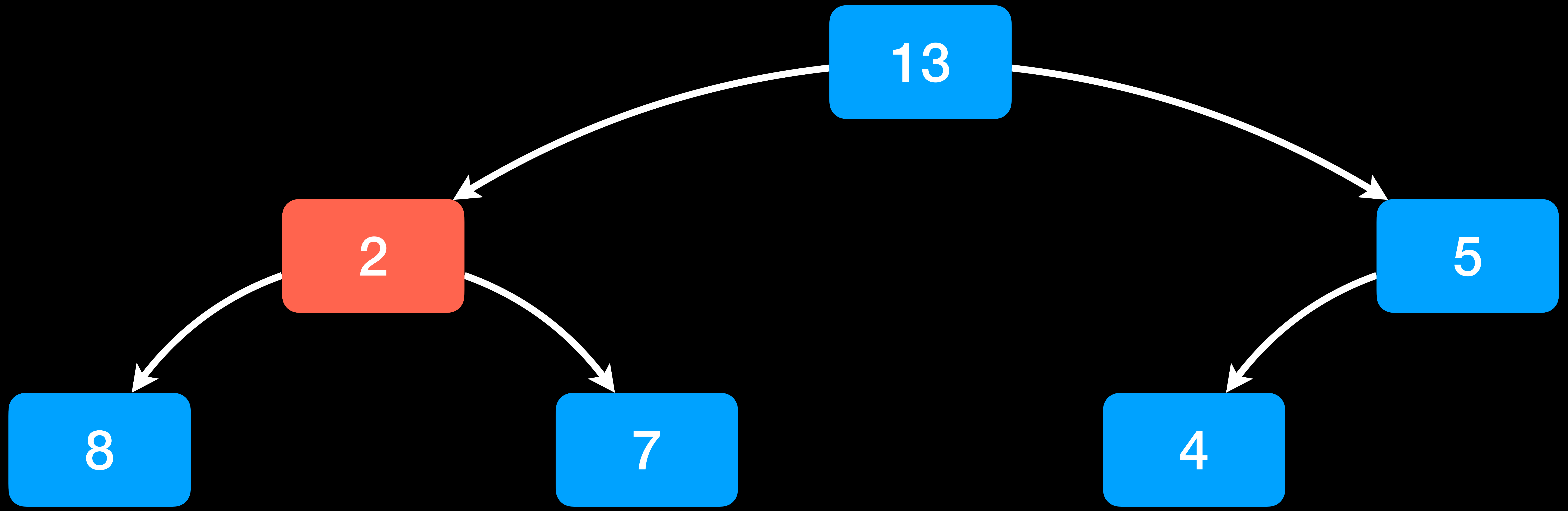


Extract one element



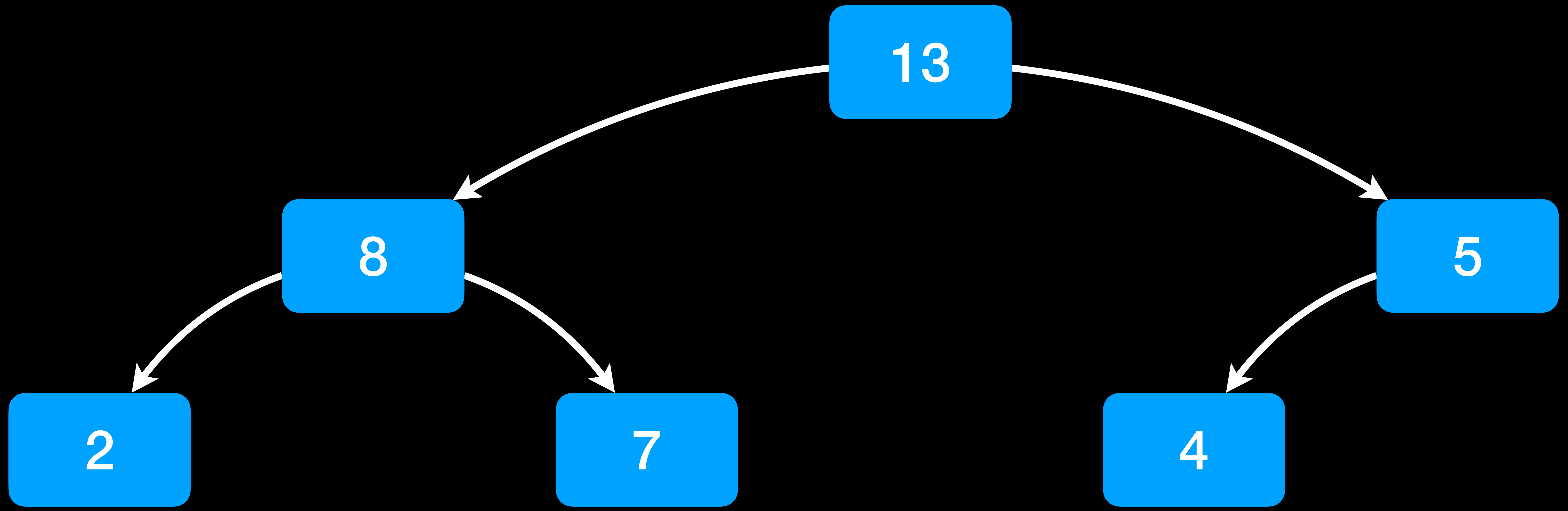
A.heap-size = 6
A.length = 9

MAX-HEAPIFY



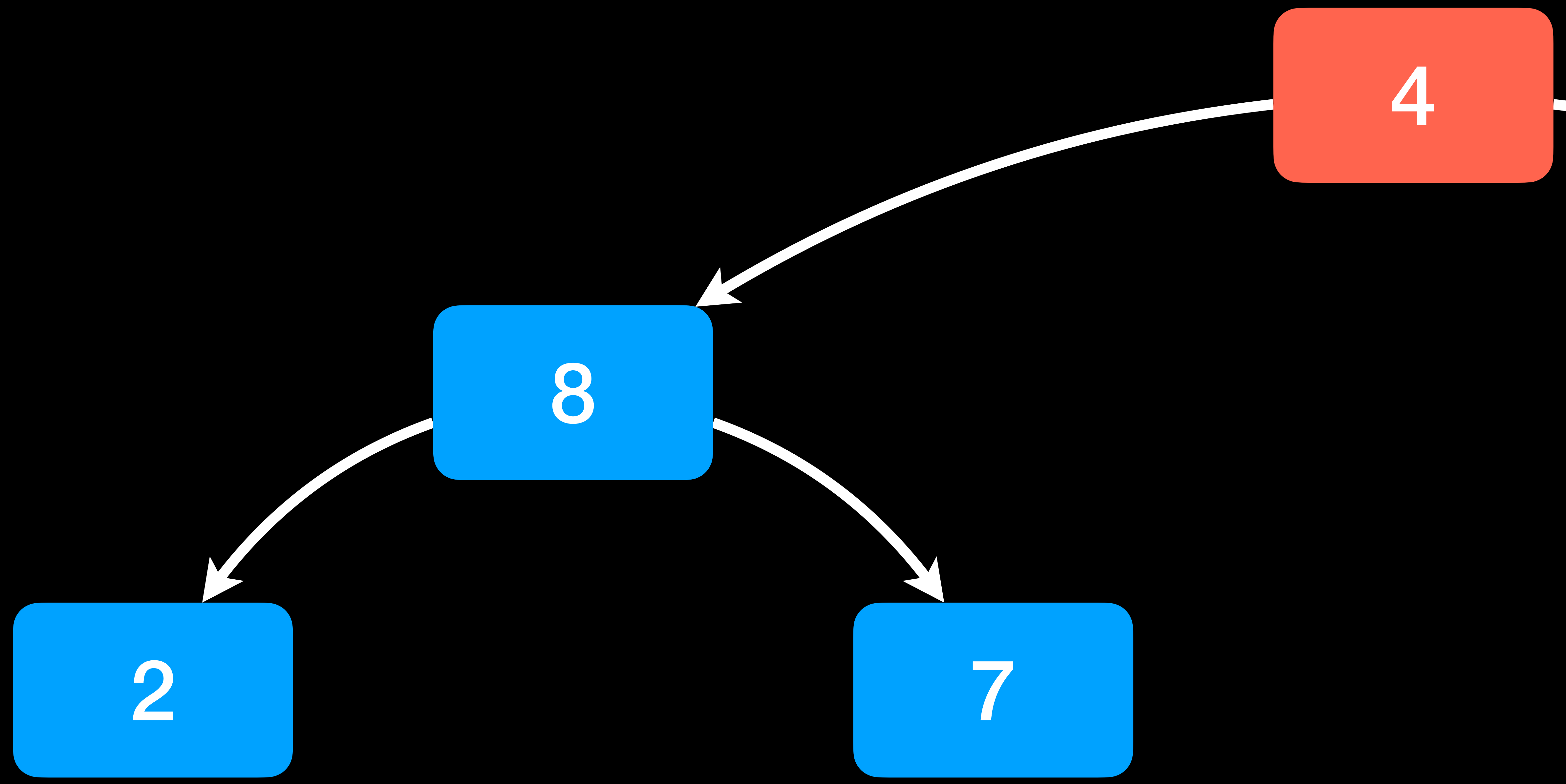
A.heap-size = 6
A.length = 9

MAX-HEAPIFY

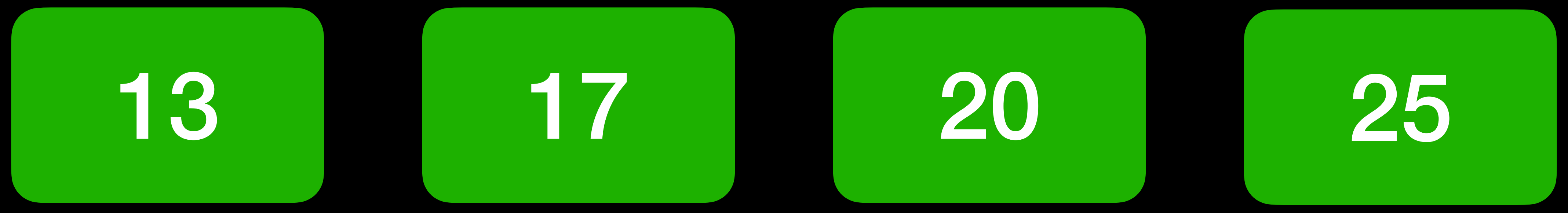


A.heap-size = 6
A.length = 9

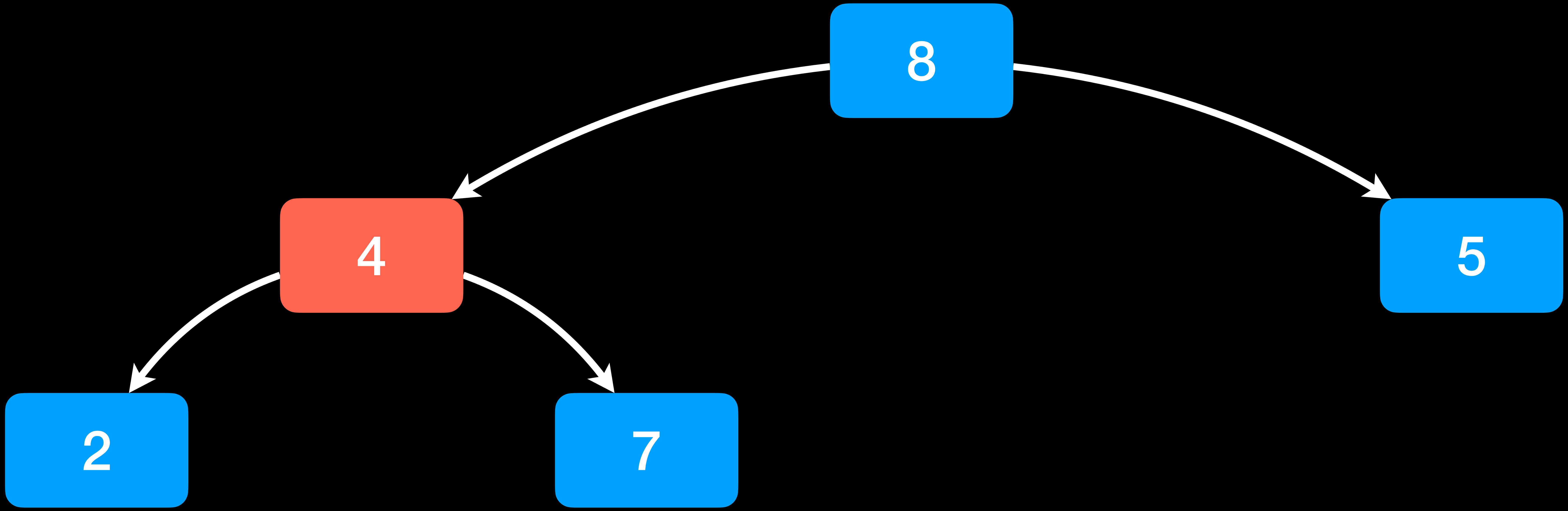
Extract one element



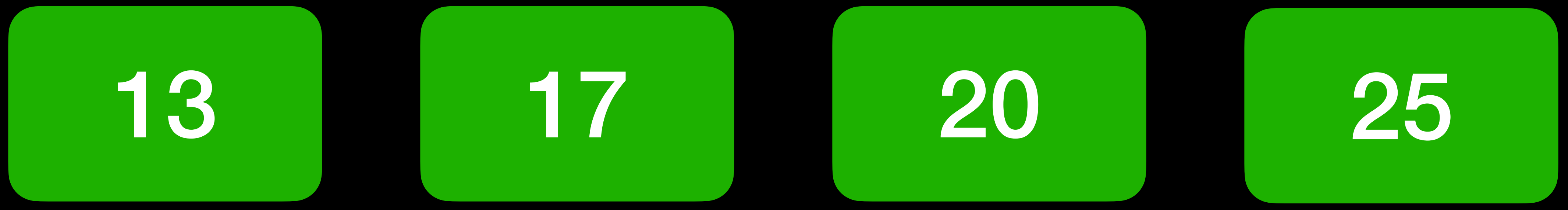
$A.heap\text{-}size = 5$
$A.length = 9$



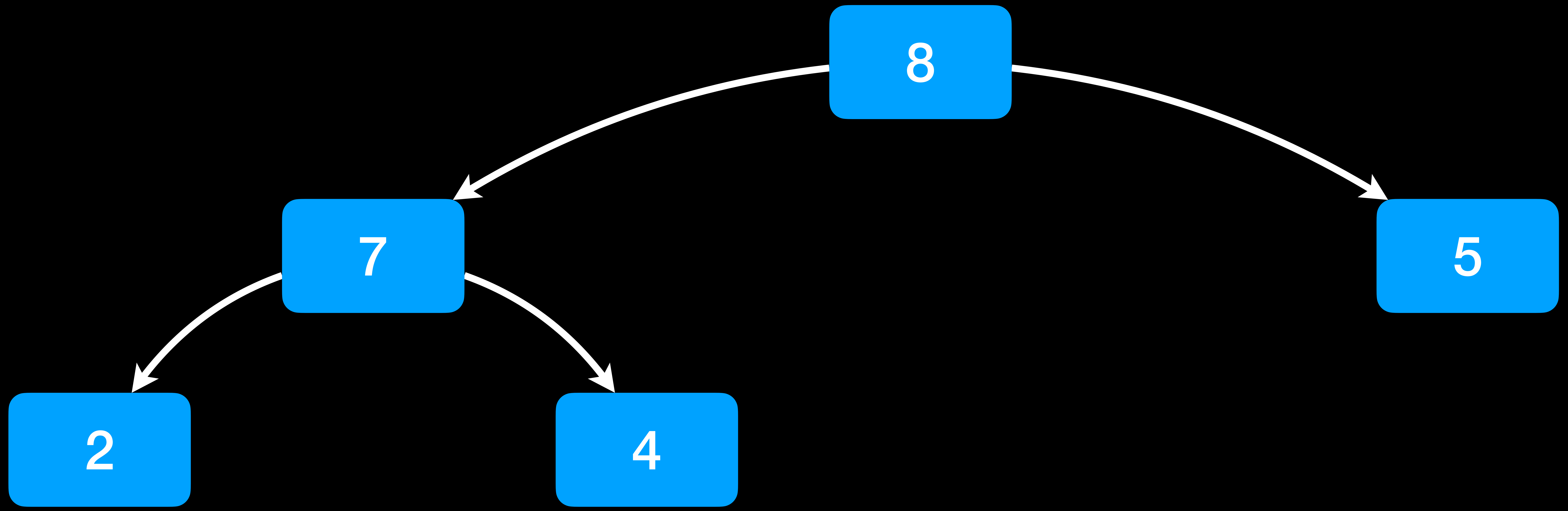
MAX-HEAPIFY



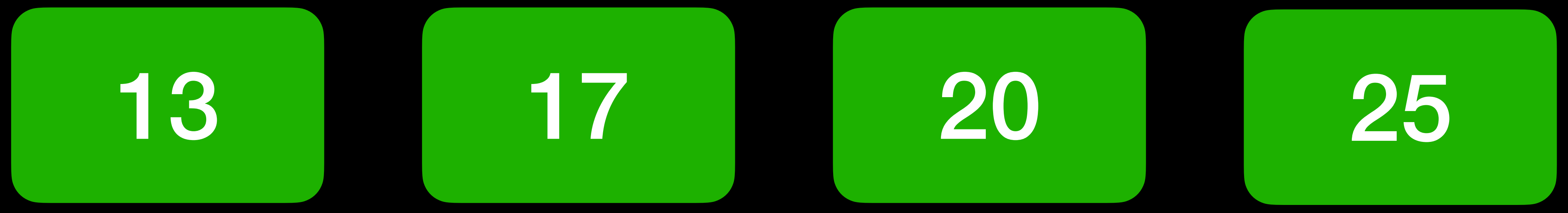
A.heap-size = 5
A.length = 9



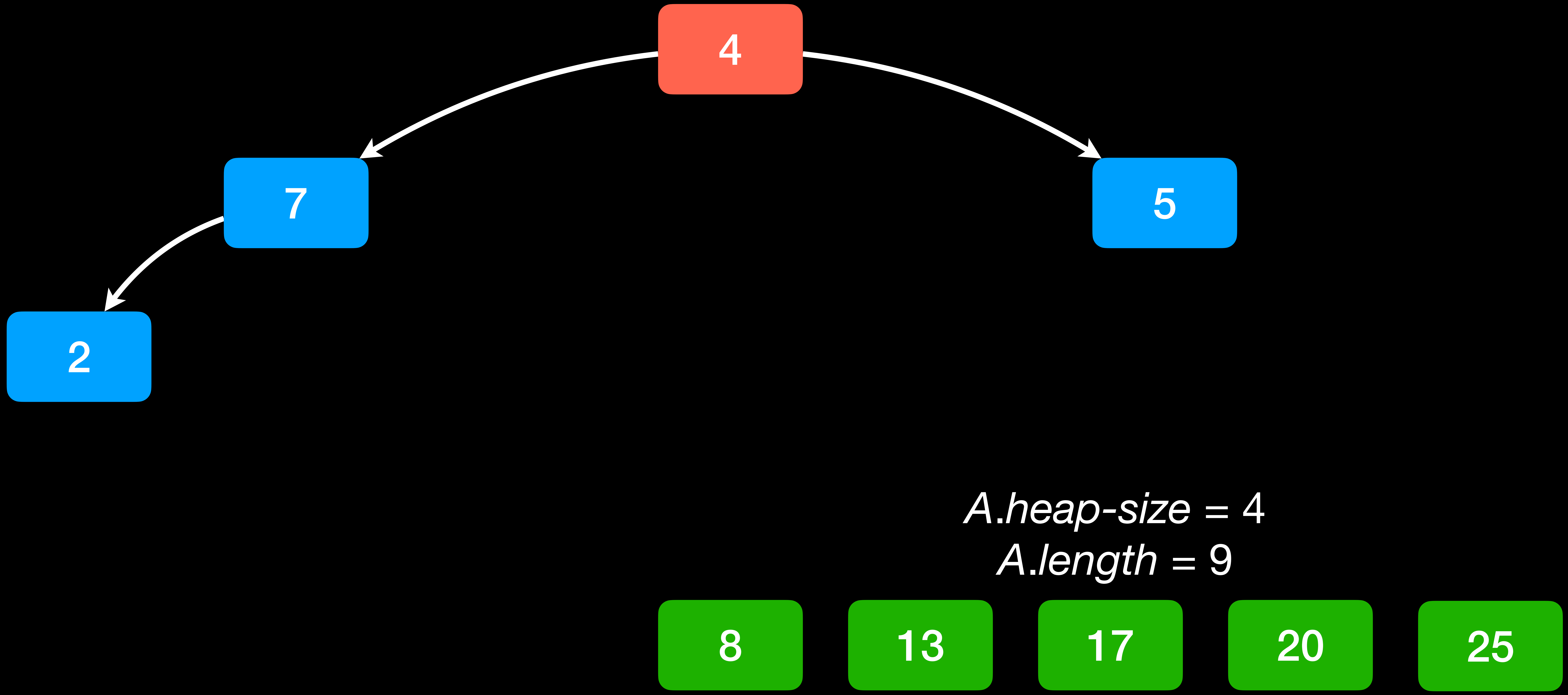
MAX-HEAPIFY



A.heap-size = 5
A.length = 9



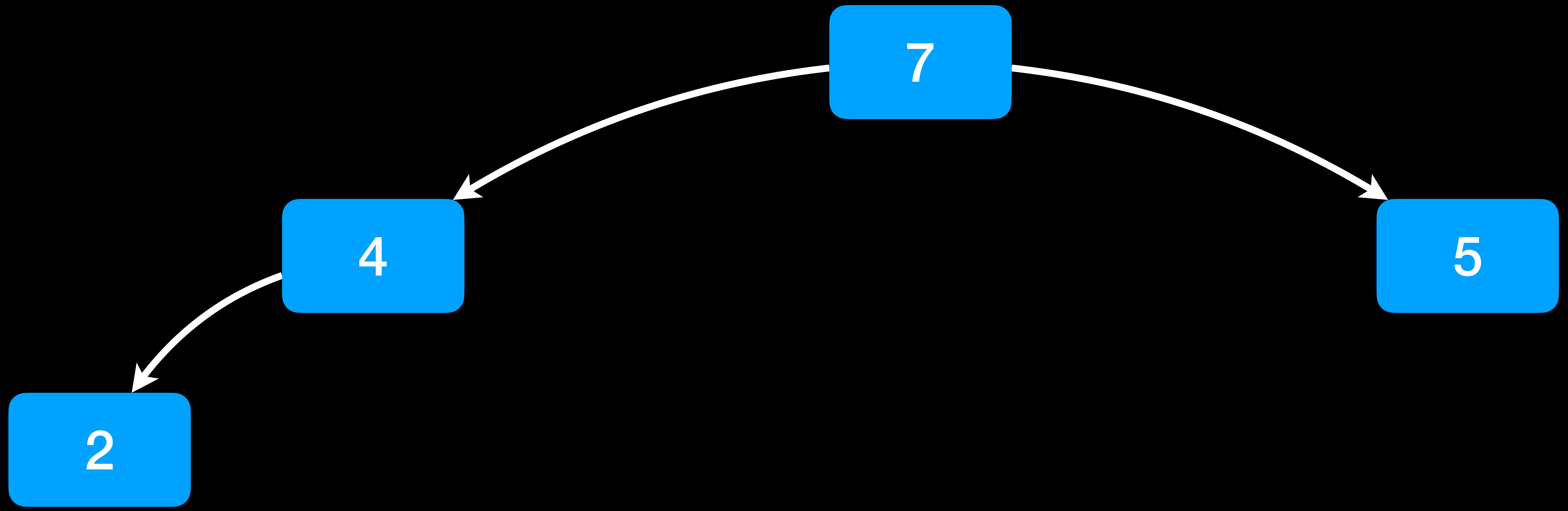
Extract one element



A.heap-size = 4

A.length = 9

MAX-HEAPIFY

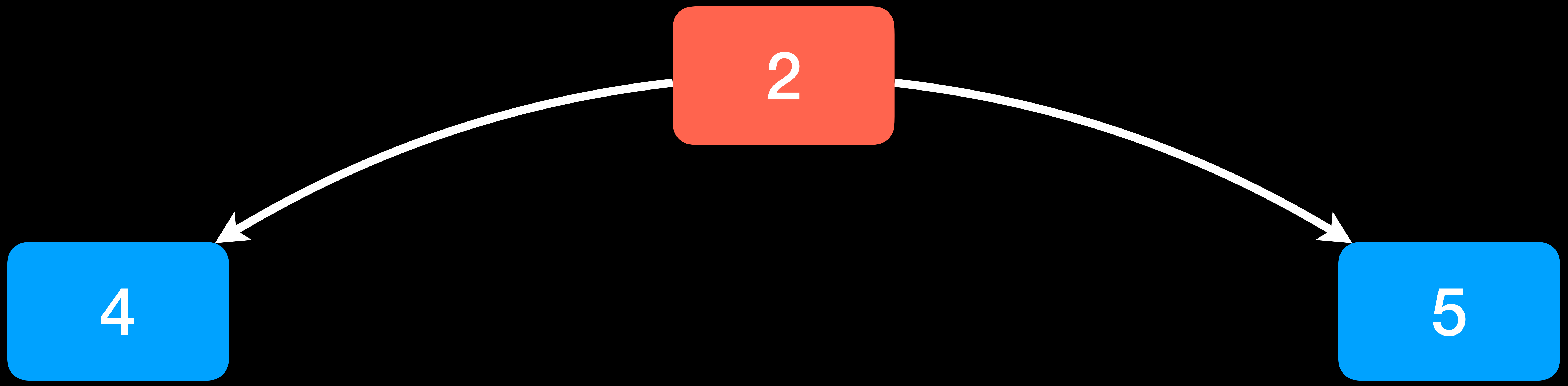


$A.heap\text{-}size = 4$

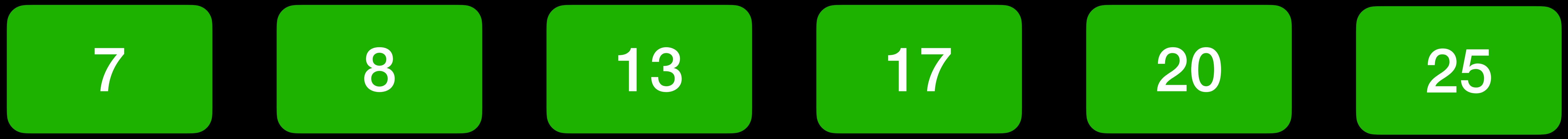
$A.length = 9$

8	13	17	20	25

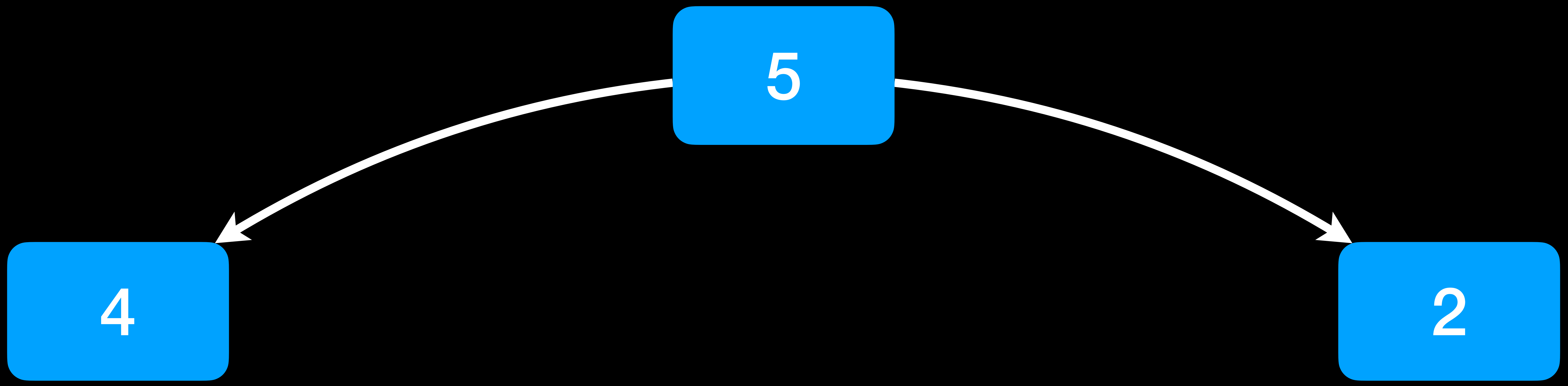
Extract one element



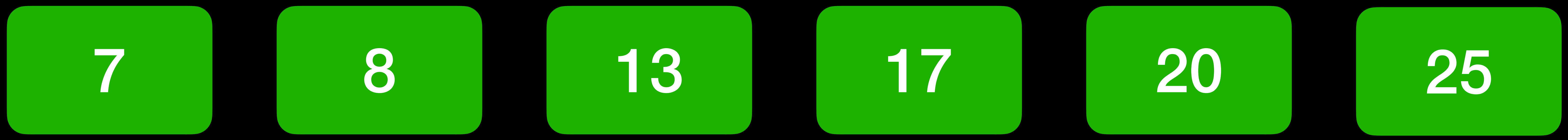
A.heap-size = 3
A.length = 9



MAX-HEAPIFY



A.heap-size = 3
A.length = 9



Extract one element

A.heap-size = 2
A.length = 9

MAX-HEAPIFY

A.heap-size = 2
A.length = 9

Extract one element

2

A.heap-size = 1
A.length = 9

4	5	7	8	13	17	20	25

Finished

A.*heap-size* = 0
A.*length* = 9

2	4	5	7	8	13	17	20	25

Emergency Queue

急诊室队列

- Assume that the priority queue in an hospital emergency ward is implemented using heaps. Draw the heap that results after each of the steps on the following slide.

- 假设医院急诊病房中的优先级队列是使用堆来实现的。绘制每个下一张幻灯片上步骤后的结果堆。

Emergency Queue

急诊室队列

1. Patient A arrives with urgency 7.
2. Patient B arrives with urgency 3.
3. Patient C arrives with urgency 5.
4. The doctor calls one patient for treatment.
5. Patient D arrives with urgency 8.
6. The doctor calls one patient for treatment.
7. Patient E arrives with urgency 4.
8. Patient B leaves the hospital without treatment.
9. The urgency of patient E changes to 6.
10. The doctor calls one patient for treatment.
11. The doctor calls one patient for treatment.

1. 病人 A 到达记者们，紧急度7。
2. 病人 B 到达记者们，紧急度3。
3. 病人 C 到达记者们，紧急度5。
4. 医生叫一个病人来治疗。
5. 病人 D 到达记者们，紧急度8。
6. 医生叫一个病人来治疗。
7. 病人 E 到达记者们，紧急度4。
8. 病人 B 未经治疗就出院了。
9. 病人 E 的紧急度增加到6。
10. 医生叫一个病人来治疗。
11. 医生叫一个病人来治疗。

Patient A arrives

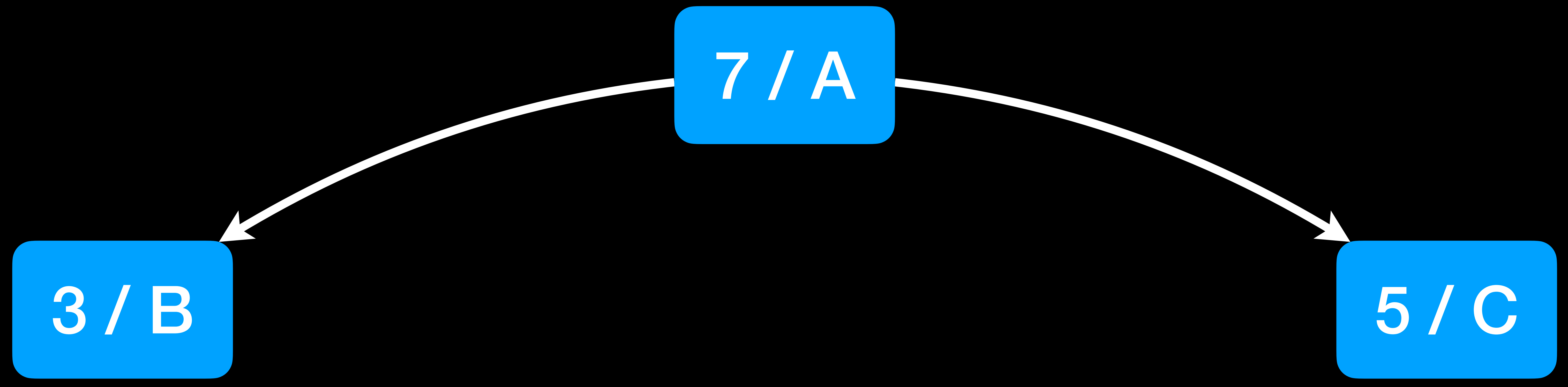
7 / A

A.heap-size = 1

Patient B arrives

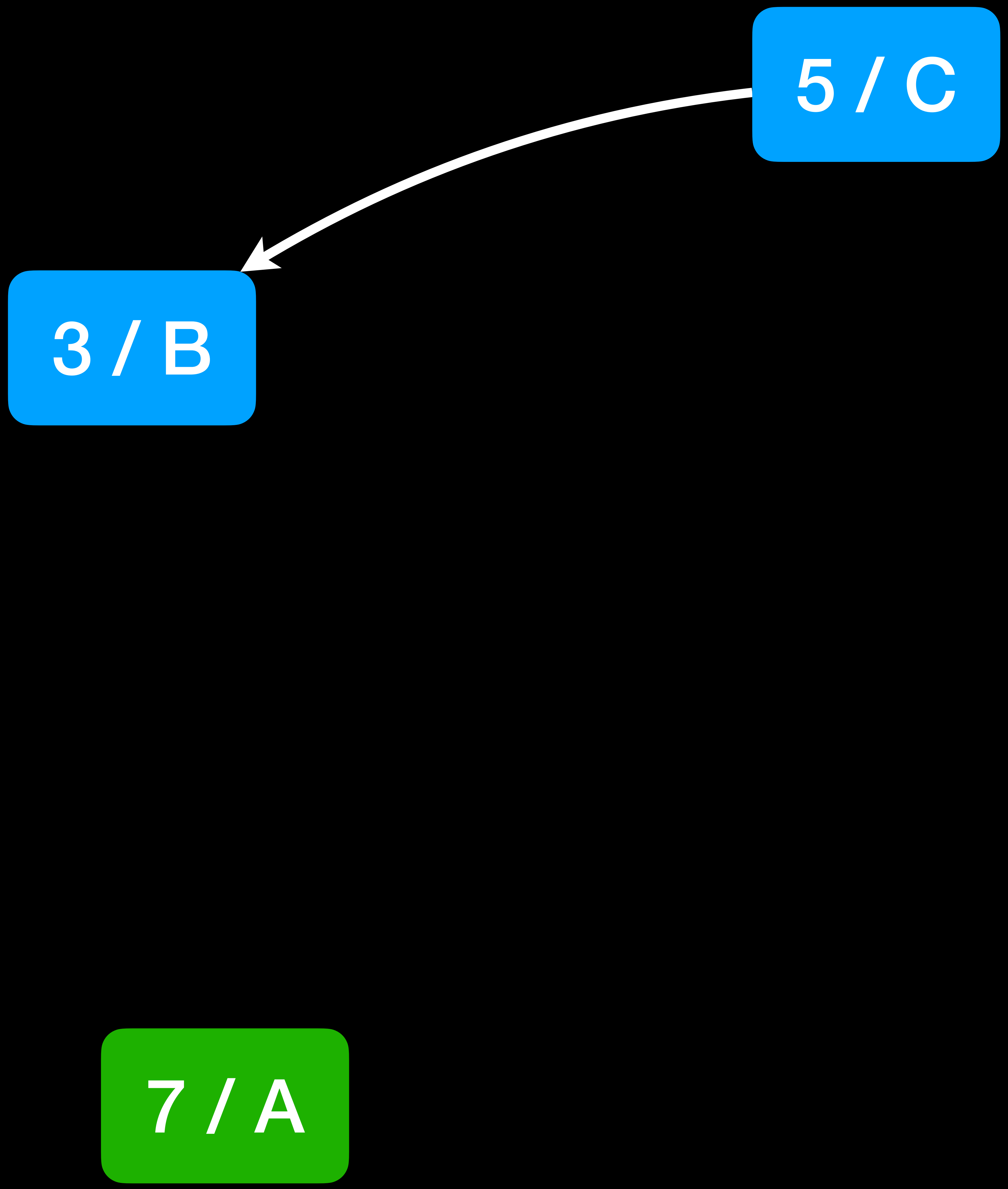
$A.\mathit{heap\text{-}size} = 2$

Patient C arrives



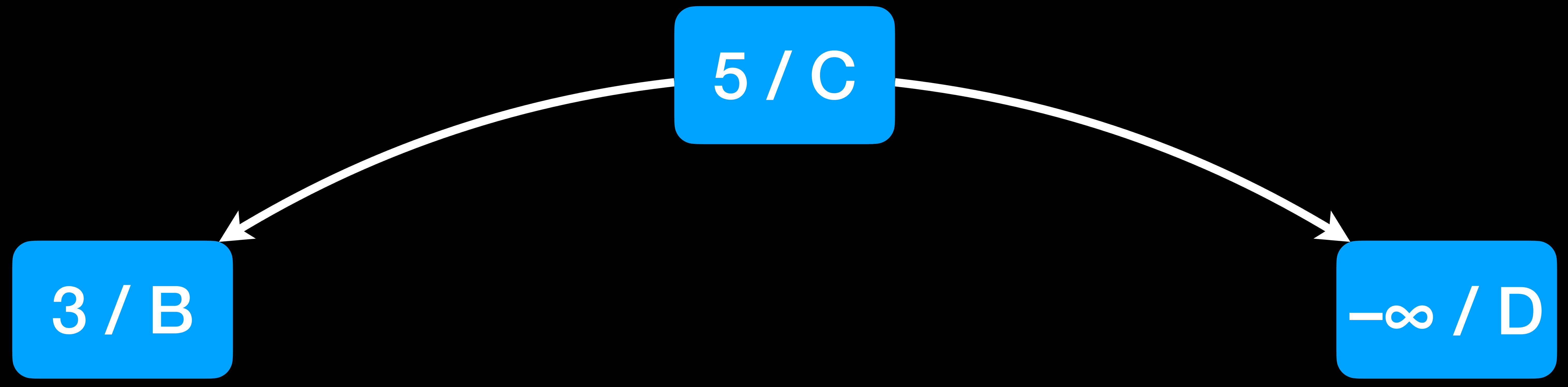
A.heap-size = 3

The doctor calls one patient

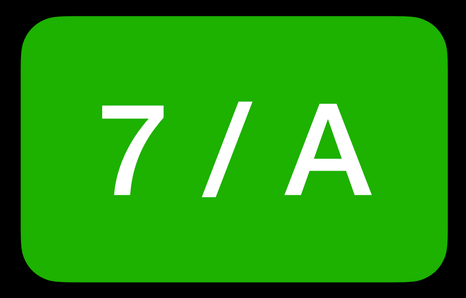


A.heap-size = 2

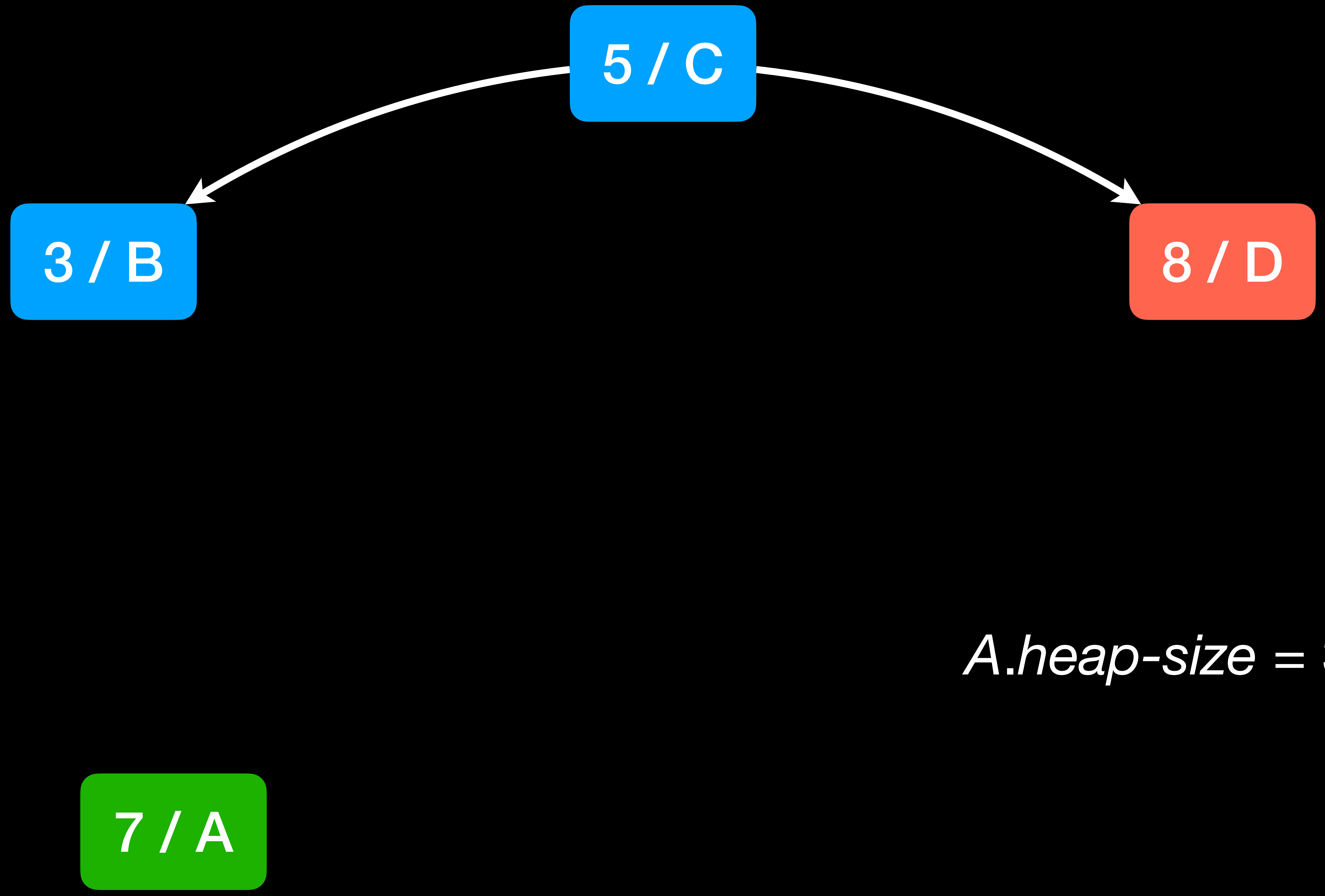
Patient D arrives



$A.heap\text{-}size = 3$

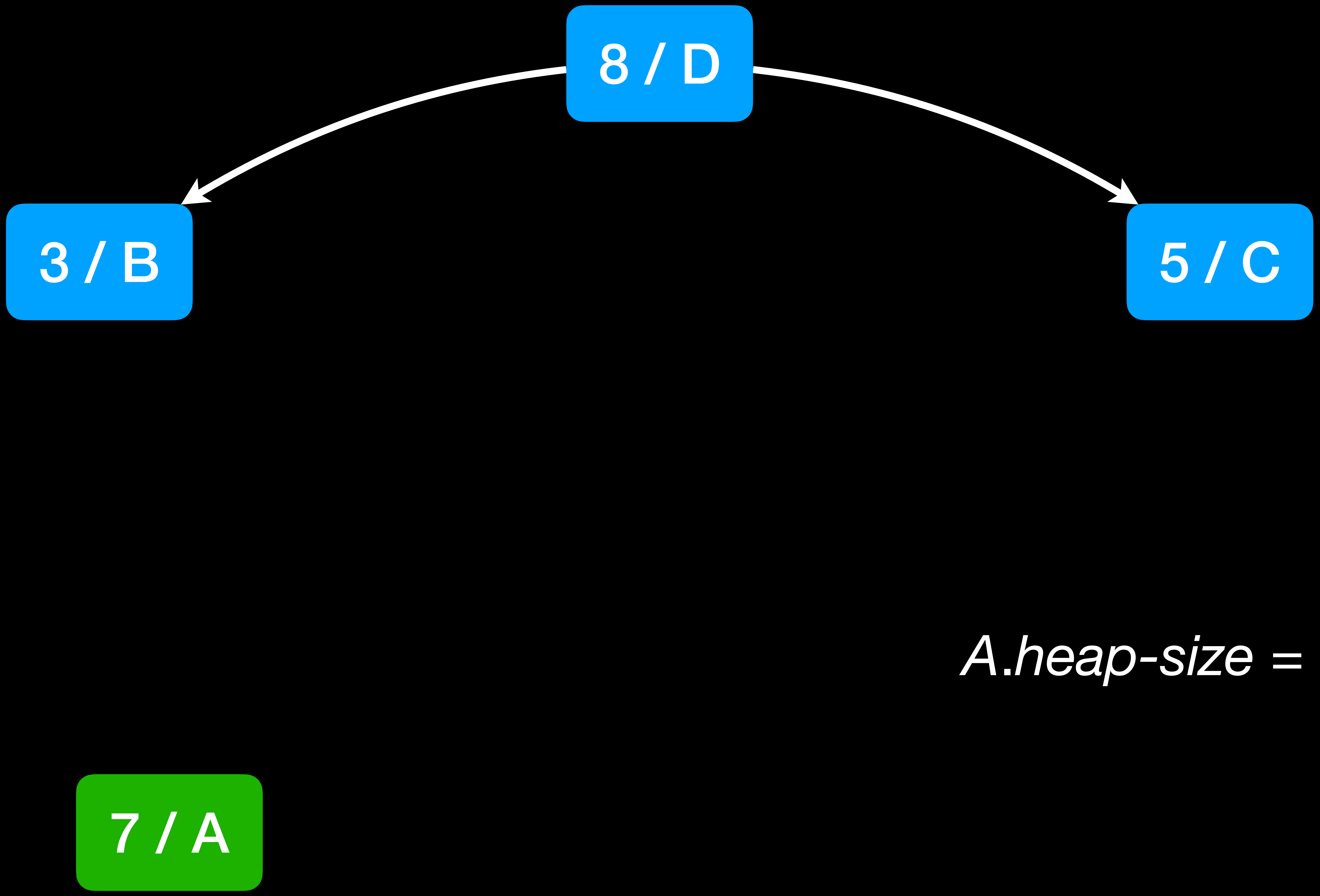


HEAP-INCREASE-KEY



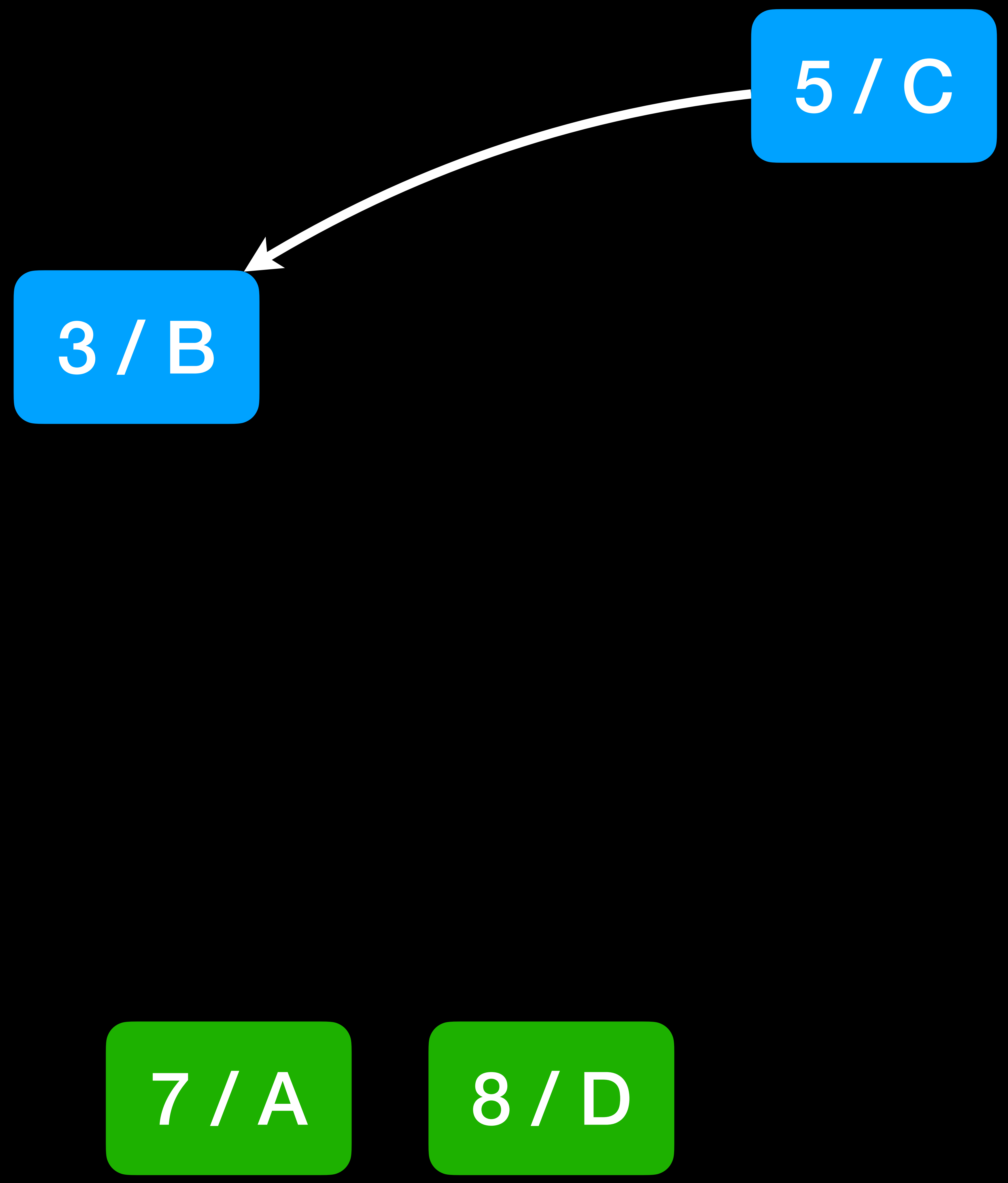
A.heap-size = 3

HEAP-INCREASE-KEY



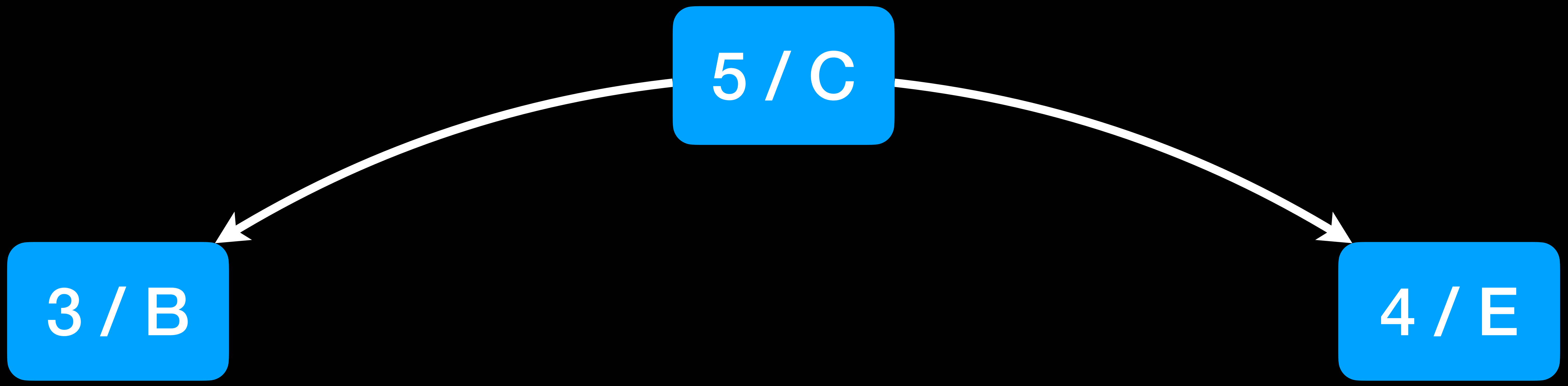
$A.heap\text{-}size = 3$

The doctor calls one patient

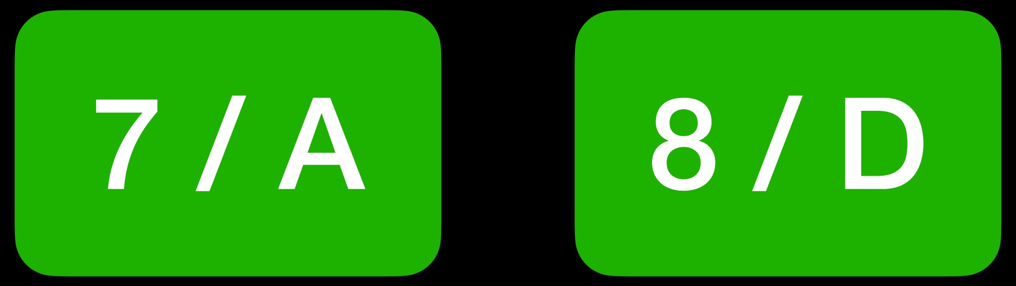


A.heap-size = 2

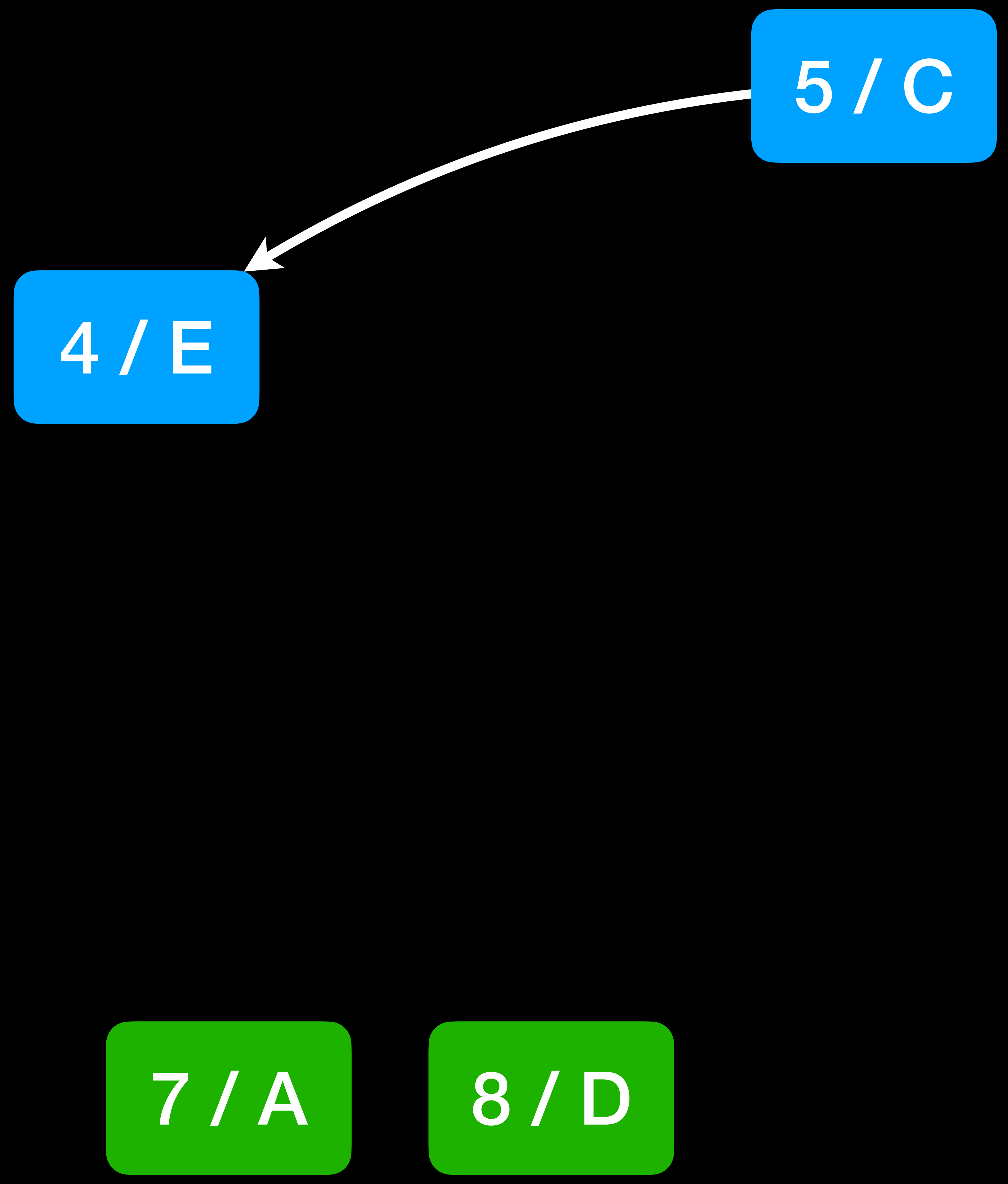
Patient E arrives



A.heap-size = 3

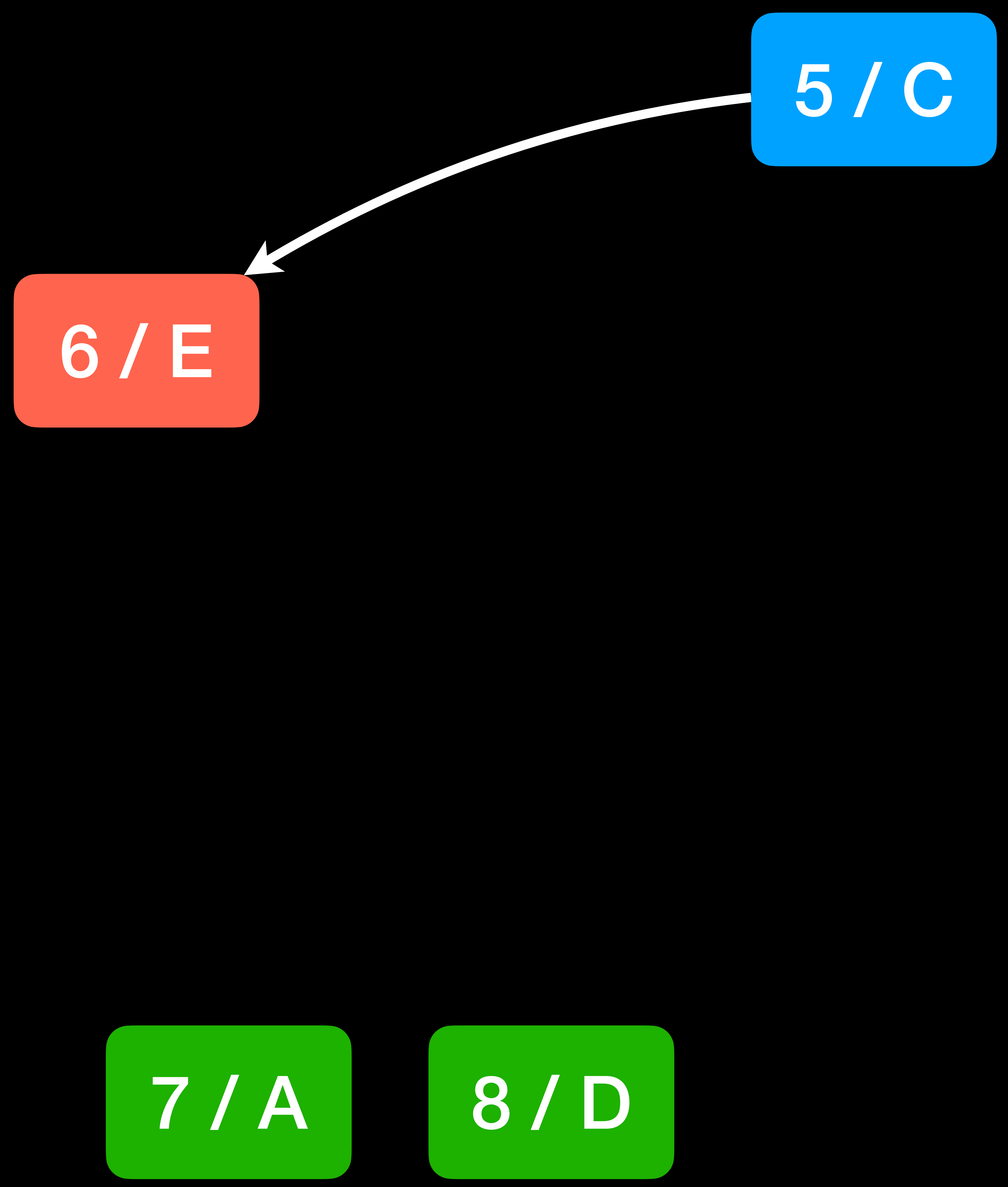


Patient B leaves



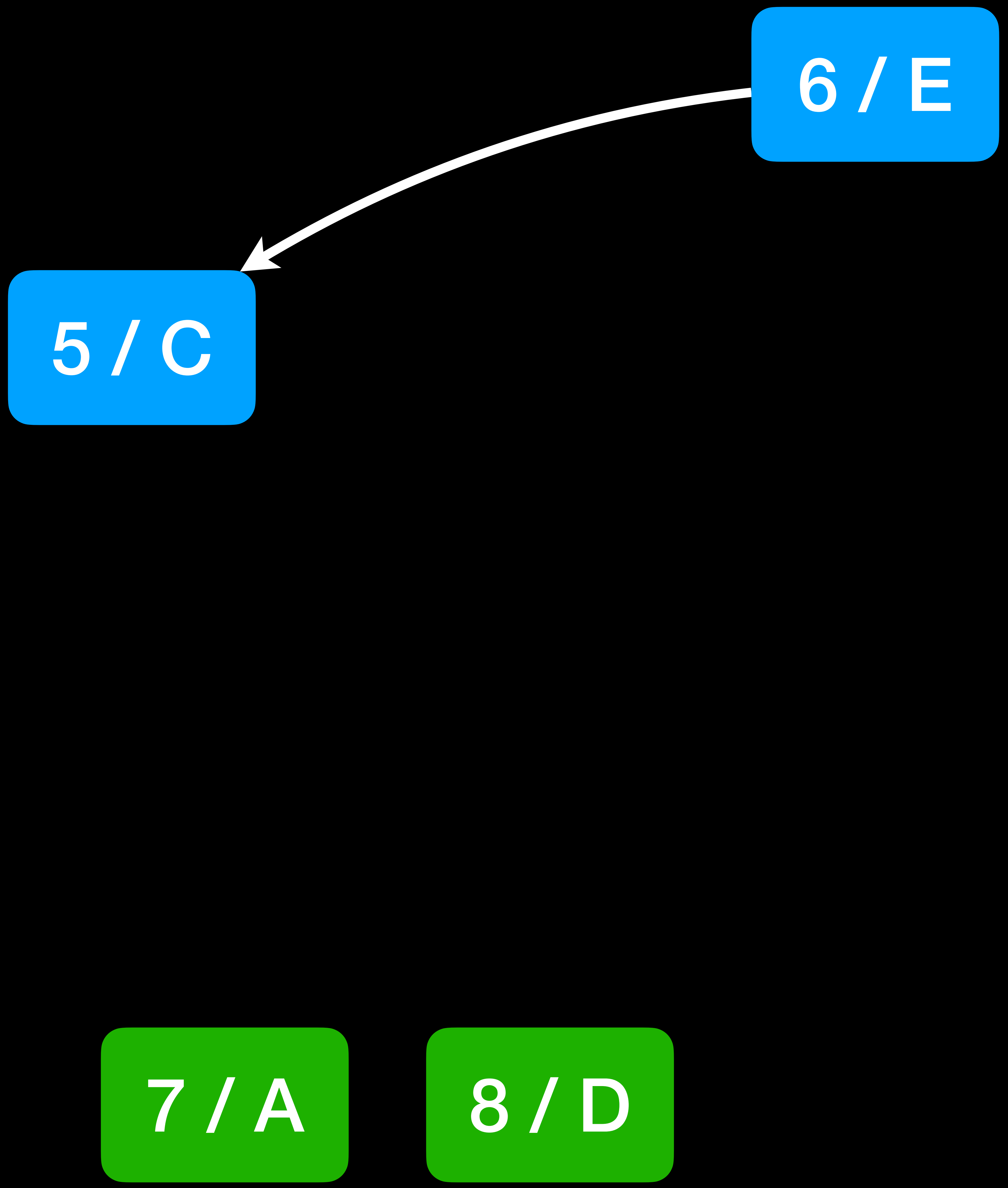
A.heap-size = 2

Patient E becomes more urgent



A.heap-size = 2

Heap-Increase-Key



A.heap-size = 2

The doctor calls one patient

5 / C

A.heap-size = 1

7 / A 8 / D 6 / E

The doctor calls one patient

$A.heap\text{-}size = 0$

7 / A   8 / D   6 / E   5 / C

Quicksort 快速排序

Quicksort

快速排序

- Divide-and-Conquer algorithm:
  1. partition the array into “small” and “large” elements
  2. sort “small” elements recursively
  3. sort “large” elements recursively
- What definition of “small” and “large” is general enough to apply to every array?
  ➥ smaller/larger than a sample from the array, called pivot

- 分析策略算法:
  1. 把数组划分
     找到“小的”和“大的”元素
  2. 递归排序“小的”元素
  3. 递归排序“大的”元素
- 什么“小”和“大”的定义是通用的,
  这样可以用在所有的数组?
  ➥ 小于/大于一个数组中的示例元素,
  成为主元(pivot = 枢轴)。

Partition example 划分的例子

2	9	12	1	4	11	8	6

Choose pivot 选主元

2	9	12	1	4	11	8	6

pivot/主元

$\underbrace{}_{\le \text{pivot}}$ $\underbrace{}_{\ge \text{pivot}}$

Find small element 找到小的元素

2		9	12	1	4	11	8		6

≤ pivot

≥ pivot

pivot/主元

Find large element 找到大的元素

Find large element 找到大的元素

2 | 9 | 12 | 1 | 4 | 11 | 8 | 6

$\underbrace{2}_{\leq \text{ pivot}}$ $\underbrace{9 \quad 12}_{\geq \text{ pivot}}$ 1 4 11 8 6

pivot/主元

Find small element 找到小的元素

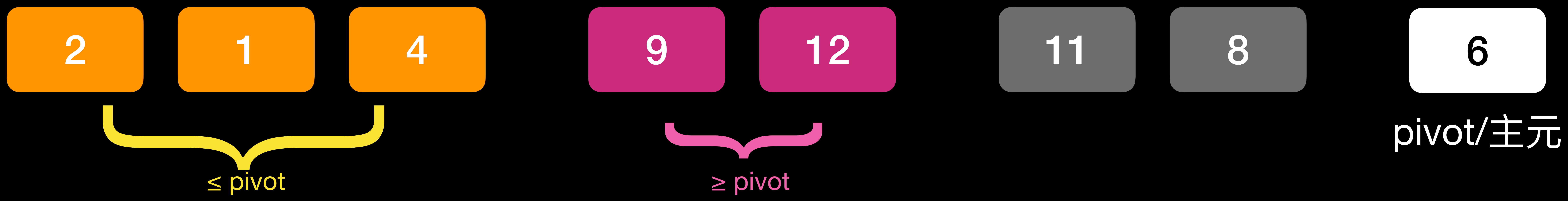
Find small element 找到小的元素

2	1	12	9	4	11	8	6

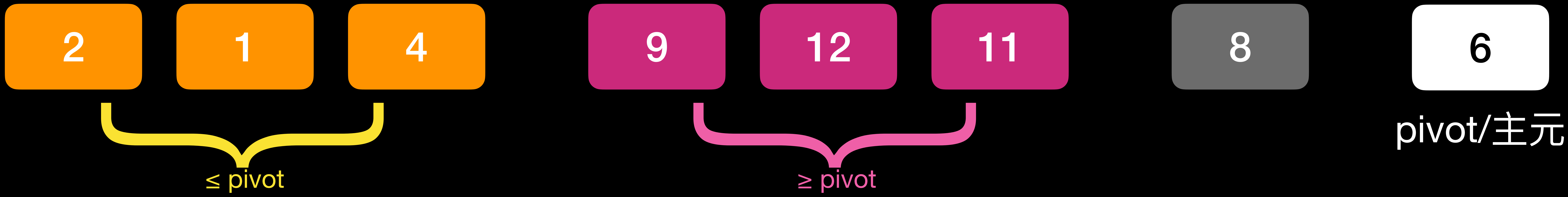
$\underbrace{2 \quad 1}_{\leq \text{ pivot}}$ $\underbrace{12 \quad 9}_{\geq \text{ pivot}}$ 4 11 8 $\underbrace{6}_{\text{pivot/主元}}$

Find small element　　找到小的元素

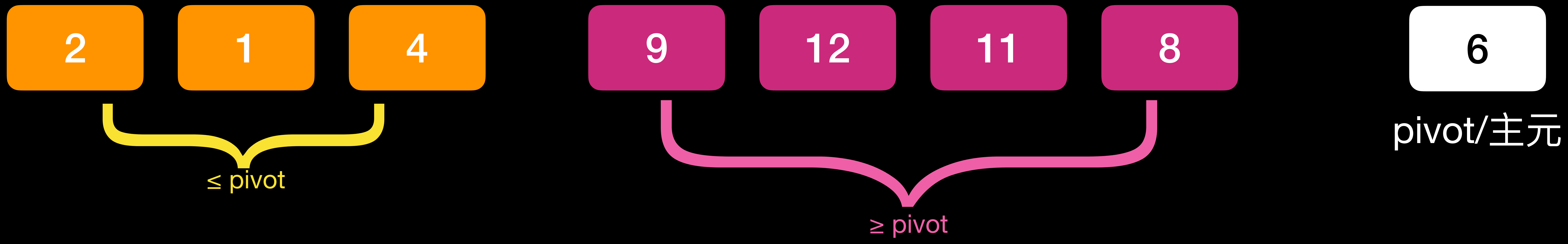
Find large element 找到大的元素



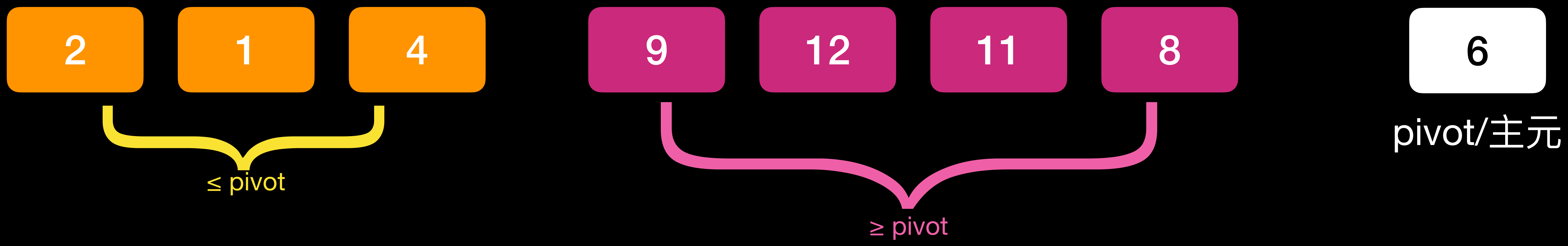
Find large element 找到大的元素



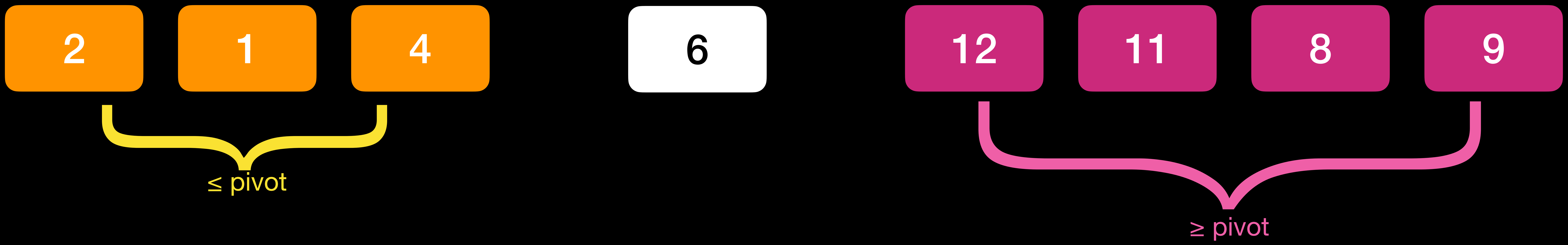
Find large element 找到大的元素



Place pivot correctly 校正主元的位置



Partition finished 划分结束



PARTITION

```
PARTITION(A, first, last)   // partition A[first] ... A[last]
pivot = A[last]
last-small = first – 1   // last of the small elements
for j = first to last – 1
        if A[j] ≤ pivot
                last-small = last-small + 1
                Exchange A[last-small] with A[j]
Exchange A[last-small + 1] with A[last]
return last-small + 1   // return new position of pivot
```

A[*first*] ... *A*[*last-small*] 都是 ≤ *pivot*.
A[*last-small*+1] ... *A*[*j*–1] 都是 ≥ *pivot*.
A[*last*] = *pivot*.

PARTITION

- Specification: PARTITION returns a value *mid* with the property: PARTITION permutes the elements of *A*[*first*] ... *A*[*last*] such that
  - *A*[*first*] ... *A*[*mid*–1] are all ≤ *A*[*mid*]
  - *A*[*mid*+1] ... *A*[*last*] are all ≥ *A*[*mid*]
- PARTITION runs in time *O*(*last*–*first*+1).

QUICKSORT

QUICKSORT(*A*, *first*, *last*)   // sort *A*[*first*] ... *A*[*last*]

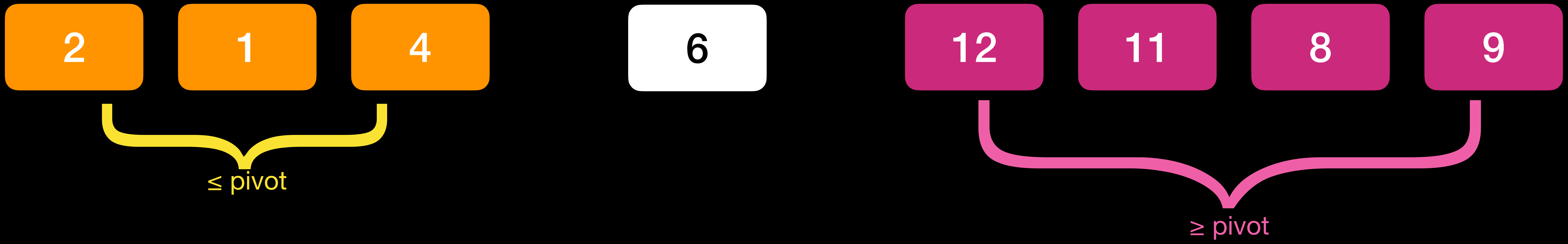
if *first* < *last*

    mid = PARTITION(*A*, *first*, *last*)

    QUICKSORT(*A*, *first*, *mid*–1)

    QUICKSORT(*A*, *mid*+1, *last*)

Partition finished 划分结束



Partition finished 划分结束

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
2	1	4	6	12	11	8	9

QUICKSORT(A, 1, 3)

QUICKSORT(A, 5, 8)

Quicksort left part 排序左边的部分

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
2	1	4	6	12	11	8	9

≤ pivot

QUICKSORT(A, 5, 8)

Quicksort left part　排序左边的部分

A[1]	*A*[2]	*A*[3]	*A*[4]	*A*[5]	*A*[6]	*A*[7]	*A*[8]
2	1	4	6	12	11	8	9

QUICKSORT(A, 1, 2)

QUICKSORT(A, 5, 8)

Quicksort left part 排序左边的部分

A[1]	*A*[2]	*A*[3]	*A*[4]	*A*[5]	*A*[6]	*A*[7]	*A*[8]
2	1	4	6	12	11	8	9

QUICKSORT(*A*, 1, 2)

QUICKSORT(*A*, 5, 8)

Quicksort left part 排序左边的部分

A[1]	*A*[2]	*A*[3]	*A*[4]	*A*[5]	*A*[6]	*A*[7]	*A*[8]
1	2	4	6	12	11	8	9
	≥ pivot						

QUICKSORT(*A*, 5, 8)

Quicksort left part 排序左边的部分

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
1	2	4	6	12	11	8	9

QUICKSORT(A, 2, 2)

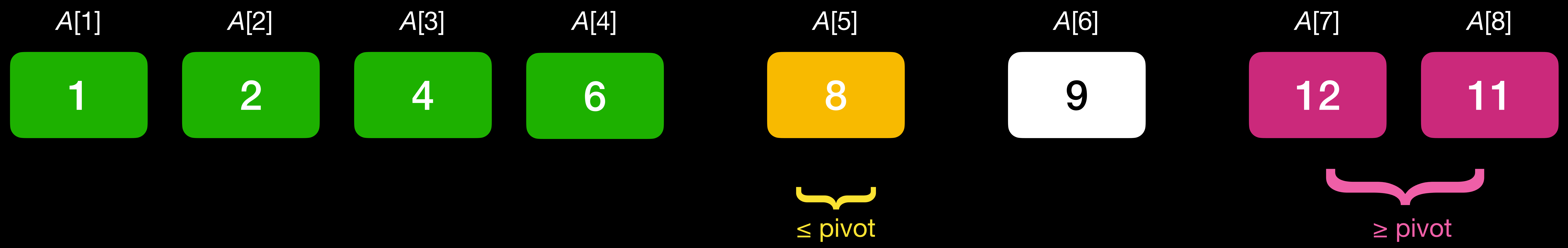
QUICKSORT(A, 5, 8)

Quicksort left part 排序左边的部分

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
1	2	4	6	12	11	8	9

QUICKSORT(A, 5, 8)

Quicksort right part 排序右边的部分



Quicksort right part 排序右边的部分

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
1	2	4	6	8	9	12	11

QUICKSORT(A, 5, 5)

QUICKSORT(A, 7, 8)

Quicksort right part 排序右边的部分

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
1	2	4	6	8	9	12	11

QUICKSORT(A, 7, 8)

Quicksort right part 排序右边的部分

A[1]	*A*[2]	*A*[3]	*A*[4]	*A*[5]	*A*[6]	*A*[7]	*A*[8]
1	2	4	6	8	9	11	12
							≥ pivot

Quicksort right part 排序右边的部分

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
1	2	4	6	8	9	11	12

$\text{QUICKSORT}(A, 8, 8)$

Quicksort right part 排序右边的部分

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$
1	2	4	6	8	9	11	12

Quicksort: timing 快速排序：运行时间

- Worst-case outcome of PARTITION:
  all elements ≤ pivot
  (or all elements ≥ pivot)

  $T(n) = T(n-1) + \Theta(n)$
  ➥ $T(n) = \Theta(n + n-1 + n-2 + ...) = \Theta(n^2)$

- Best-case outcome of Partition:
  50% ≤ pivot, 50% ≥ pivot

  $T(n) = 2T((n-1)/2) + \Theta(n)$
  ➥ $T(n) = \Theta(n \log n)$

- 最差的PARTITION的结果：
  所有的元素 ≤ 主元
  （或者所有的元素 ≥ 主元）

  $T(n) = T(n-1) + \Theta(n)$
  ➥ $T(n) = \Theta(n + n-1 + n-2 + ...) = \Theta(n^2)$

- 最优的PARTITION的结果：
  50% ≤ 主元, 50% ≥ 主元

  $T(n) = 2T((n-1)/2) + \Theta(n)$
  ➥ $T(n) = \Theta(n \log n)$

Quicksort: timing 快速排序：运行时间

- Expected time?
- To get a running time close to the worst case, almost all calls to PARTITION have to be close to the worst case.
- Probability that most calls to PARTITION are bad is very small.
- Therefore, the expected time is in $O(n \log n)$.
- Please read “Intuition for the average case” in Section 7.2.
  It is not necessary to read detailed Section 7.4.

- 平均运行时间?
- 为了使运行时间接近最坏的情况，
  几乎所有对PARTITION的调用都必须接近最坏情况。
- 大多数对PARTITION的调用都是坏的概率非常小。
- 因此，预期时间为 $O(n \log n)$。
- 请阅读第7.2节中的
  “对于平均情况的直观观察”。
  无需阅读第7.4节的详细内容。

Summary of Comparison Sort Algorithms

排序

Sort Algorithms 各种排序算法

Name 名称	Running time 运行时间	Stable? 稳定?	In-place? 原址?
Insertion Sort 插入排序	best-case: $\Theta(n)$ worst-case: $\Theta(n^2)$	Yes	是
Selection Sort 选择排序	$\Theta(n^2)$	否	Yes
Merge Sort 合并排序	$O(n \log n)$	是	No
Heapsort 堆排序	$O(n \log n)$	否	Yes
Quicksort 快速排序	worst-case: $\Theta(n^2)$ expected: $\Theta(n \log n)$	No	是*

Sort Algorithms

各种排序算法

- An algorithm is stable if it keeps the order of elements with equal keys. (important, for example, for multiple patients with the same urgency in the emergency ward)

- 排序算法称为稳定的：相同之的元素在输出数组中的相对次序与他们在输入中的次序相同。
（例如：急诊室里可能有多个同样紧急的病人）

- An algorithm is in-place if it only uses O(1) additional memory.

  * (Quicksort actually uses $O(\log n)$ memory for the recursive calls, but that is mostly ignored.)

- 排序算法称为原址的：除了输入数组以外仅需要O(1)大的存储。

  * （因为快速排序是递归算法，实际使用 $O(\log n)$ 大的存储，大部分选择忽略这个情况。）

Sort Algorithms 各种排序算法

- For small arrays, use insertion sort.

- In practice quicksort is mostly the fastest algorithm (for large arrays); heapsort takes about 2× the time of quicksort.

- If it is acceptable that (very seldomly) a sort operation takes a longer $O(n^2)$ time, I suggest to use quicksort.

- Heapsort can be used as a fallback if time $O(n \log n)$ must be guaranteed.

Comparison Sort

- What is the lowest possible bound for sorting algorithms?
- We assume that no additional information is given beforehand. The algorithm must allow every permutation as a possible result.
- decision tree

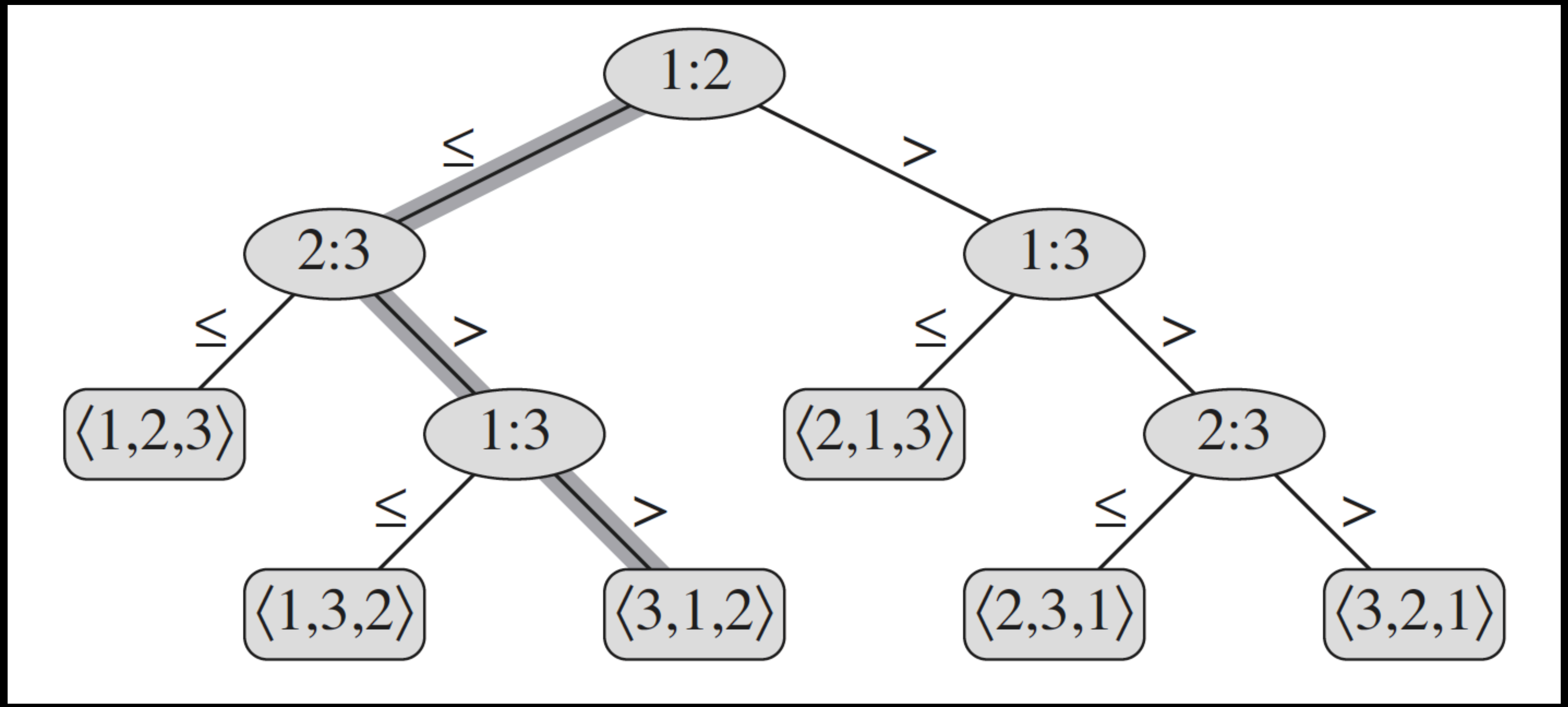
- 排序算法的最低可能界限是什么?
- 假设事先没有提供任何额外信息。

  算法必须允许每个排列作为可能的结果。

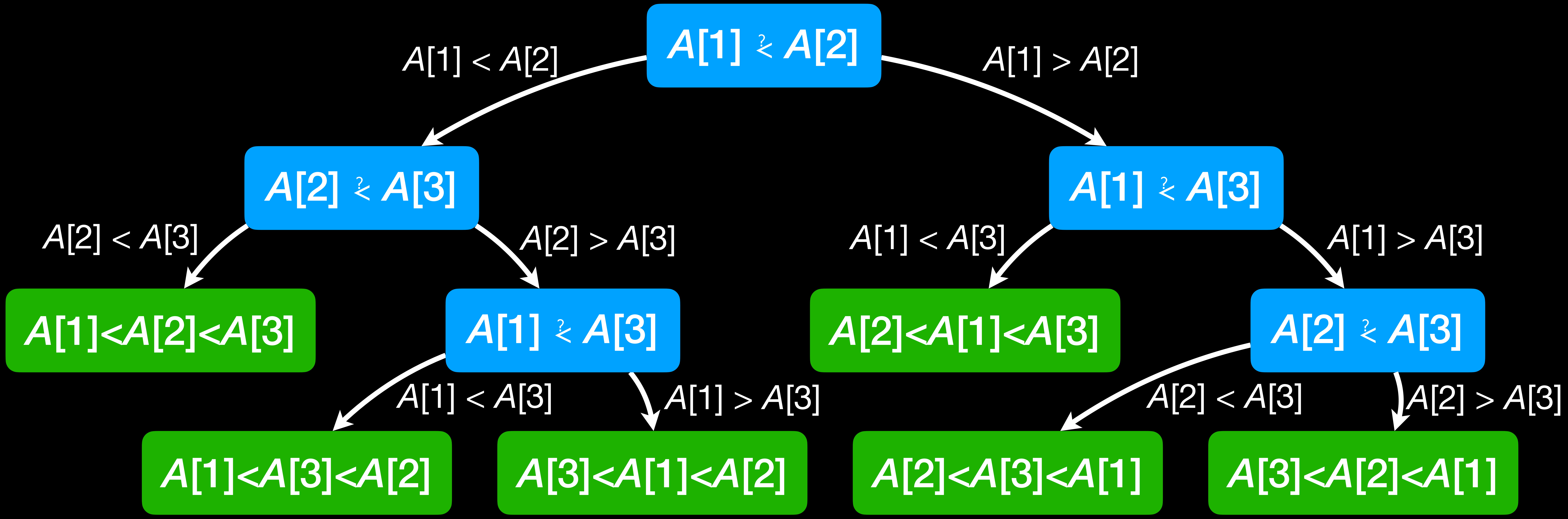
Decision tree

- Every comparison of $A[i]$ with $A[j]$ allows two outcomes: $A[i] < A[j]$ or $A[i] > A[j]$

- Decision tree := A (complete) binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

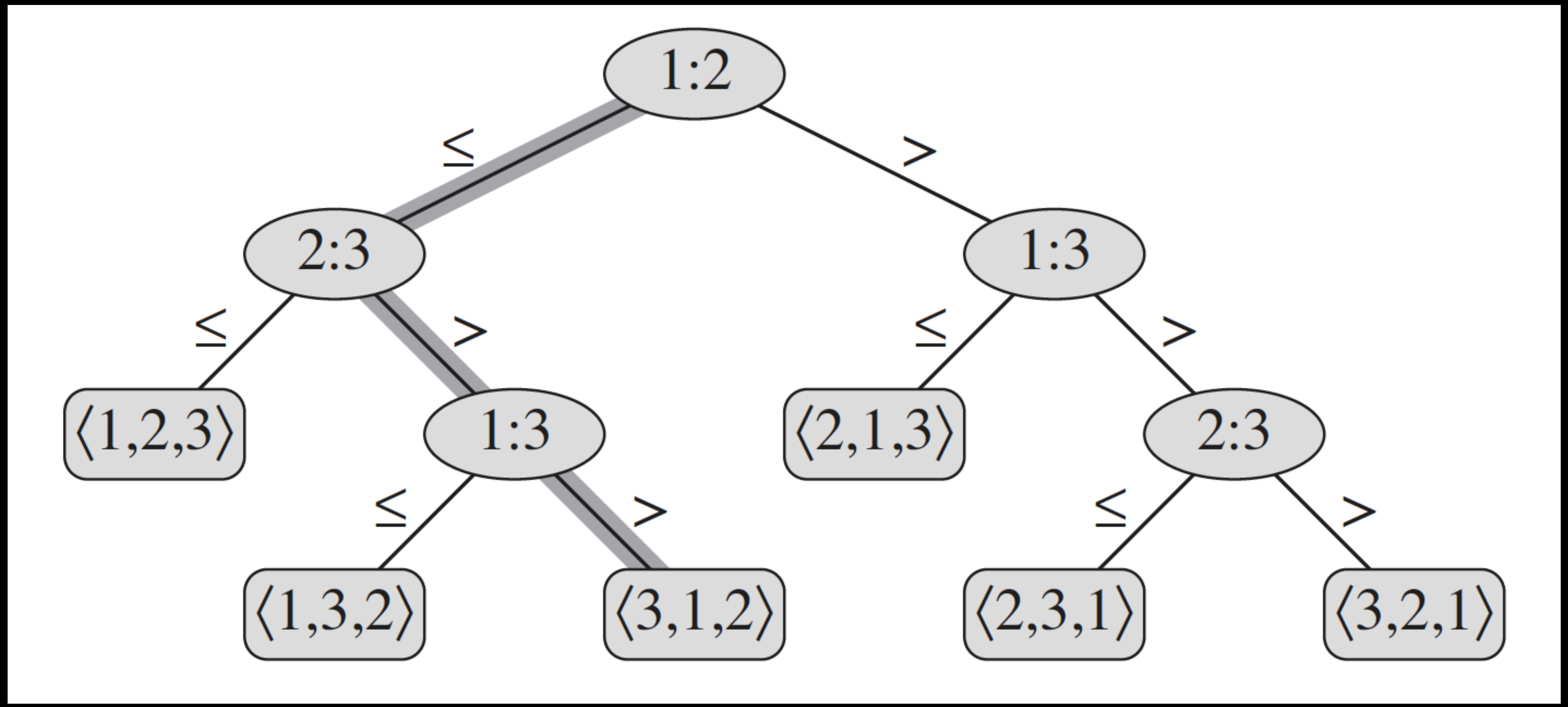


Decision tree



Decision tree

- A tree with height h has at most 2^h leaves. The height in the decision tree indicates the number of comparisons.
- n elements have $n!$ permutations. Every permutation must appear at least once. ➨ $2^h \geq n!$ ➨ $h \geq \lg n! = \Omega(n \log n)$.
- A sort algorithm must make at least $\Omega(n \log n)$ comparisons in the worst case.



Sorting in Linear Time

线性时间的排序

Use additional information!

- Counting sort:
  elements are in a small set $\{0, 1, ..., k\}$.

- Radix sort:
  elements are tuples from a small set
  $\{0, 1, ..., k\}^d$.

- ~~Bucket sort:~~
  ~~elements are uniformly distributed in~~
  ~~$[0,1]$.~~

Counting Sort

- Idea: Because the elements are in a small set {0, ..., k}, we can count how many elements have a certain value.

- If there are j elements ≤ $A[i]$, then $A[i]$ should be moved to $A[j]$
(with a small change if multiple elements have the same value).

Counting sort

Counting-Sort(*A*, *B*, *k*)

Let *C*[0 ... k] be a new array

Initialize every element of *C* to 0

for *j* = 1 **to** *A*.*length*

    C[A[j]] = C[A[j]] + 1

for *i* = 1 **to** *k*

    C[*i*] = *C*[*i*] + *C*[*i*–1]

for *j* = *A*.*length* **downto** 1

    B[C[A[j]]] = A[j]

    C[A[j]] = C[A[j]] – 1

- input: array A, containing elements in {0, ..., k}
- output: array *B*
- additional array *C* is used to count how many elements are ≤ a value.

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

C after first for loop:
C[*i*] = number of
elements with value *i*

	0	1	2	3	4	5
C	2	2	4	7	7	8

C after second for loop:
C[*i*] = number of
elements with value ≤ *i*

Some slides prepared by Prof. Zhilin WU

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B		0		2		3	3	

	0	1	2	3	4	5
C	1	2	3	5	7	8

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0		2		3	3	

	0	1	2	3	4	5
C	0	2	3	5	7	8

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	

	0	1	2	3	4	5
C	0	2	3	4	7	8

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	5

	0	1	2	3	4	5
C	0	2	3	4	7	7

Counting Sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

	0	1	2	3	4	5
C	0	2	2	4	7	7

Counting sort

- This variant of counting sort is stable.
- Running time:
    - initialize *C*: $\Theta(k)$
    - first for loop: $\Theta(n)$
    - second for loop: $\Theta(k)$
    - third for loop: $\Theta(n)$
    - total time: $\Theta(n + k)$
- Memory needed: $\Theta(n + k)$ — not in-place

Radix Sort 基数排序

- Sorting numbers with many digits: one can sort by one digit at a time
- Because there are few possible values for one digit, use counting sort for every digit.
- Which digit to sort first?

radix sort: 基数排序

Radix Sort

329
457
657
839
436
720
355

Some slides prepared by Prof. Zhilin WU

radix sort: 基数排序

Radix Sort

Sort on the most significant digit first ?

5 tiles

329
355
457
436
657
720
839

radix sort: 基数排序

Radix Sort

Sort on the most significant digit first ?

7 tiles

329
355
436
457
657
720
839

Radix Sort

Sort on the most significant digit first ?

7 tiles

329
355
436
457
657
720
839

If d digits, then
10^d tiles in the worst case,
and each tile needs to be
sorted independently

radix sort: 基数排序

Radix Sort

Sort on the least significant digit first

720
355
436
457
657
329
839

Radix Sort

Sort on the least significant digit first

720
355
436
457
657
329
839

radix sort: 基数排序

Radix Sort

Sort on the least significant digit first

720
329
436
839
355
457
657

radix sort: 基数排序

Radix Sort

Sort on the least significant digit first

329
355
436
457
657
720
839

Radix sort

基数排序

RADIX-SORT(A, d)
for i = 1 **to** d
    Call a stable sort to sort A on digit i

If $j < k$, then the digits i–1, ..., 1 of $A[j]$ are ≤ the digits of $A[k]$. If they are equal, then $A[j]$ and $A[k]$ are in the same order as in the original input.

- input: array A, containing elements in $\{0, ..., k\}^d$
- output: sorted array A

如果$j<k$，则 $A[j]$ 的数字 i–1, ..., 1 小于等于 $A[k]$的数字。如果它们相等，则$A[j]$和$A[k]$的顺序与原始输入中的顺序相同。

Radix sort 基数排序

- Suggest to use counting sort for the stable sort

- Running time:
  d × running time of the sort algorithm.

  If one uses counting sort, $O(d(n + k))$.

Summary

- Heap data structure: a binary tree that satisfies the max-heap property
  堆的数据结构：满足最大堆性质的二叉树

- Main use of heap: heapsort $O(n \log n)$, priority queue
  堆的主要用途：堆排序，优先级队列

- Quicksort: in practice quickest known general sort algorithm;
  may with a very low probability be $\Omega(n^2)$
  快速排序：在实践中已知的最快的通用排序算法（最叉运行时间$\Omega(n^2)$，可能性很低）

- Sorting faster than $O(n \log n)$ is only possible if one has additional information about the data.

Open office

开放时间

- I want to offer a time to ask questions every week.

- 每周有时间可以问我问题。