Algorithm Design and Analysis

算法设计与分析

David N. JANSEN, Bohua ZHAN
名 姓

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

Recap

- What is an algorithm?

- 什么是算法?

Recap

- Algorithm := sequence of instructions that transform input into output
  把输入转换成输出的计算步骤的序列

- Big-O Notation: describe asymptotic rate of growth of functions
  大O记号：描写函数的渐近的增长速度

- Divide and Conquer: a method to construct algorithms
  divide a problem into smaller problems and solve every one recursively

- Recurrence 递归式: describe runtime of a divide-and-conquer algorithm

Simple Sort Algorithms

简单的排序算法

Simple Sort Algorithms

- Insertion Sort:
  The first part of the array is already sorted.
  To extend that part, insert an additional element in the correct place.

- Selection Sort:
  The first part of the array already contains the smallest elements in order.
  To extend that part,
  find the smallest remaining element
  and append it to the sorted elements.

简单的排序算法

- 插入排序:
  数组的第一部分已经排序的。
  为了延长这个部分,
  在正确的位置插入一个附加元素。

- 选择排序:
  数组的第一部分已包含
  最小的元素按顺序。
  为了延长这个部分,
  找到剩余的最小元素,
  并将其附加到已排序的元素中。

Insertion Sort

INSERTION-SORT(array A)

for j := 2 **to** $A.length$
  $key = A[j]$
  $i = j - 1$
  while $i > 0$ **and** $A[i] > key$
    $A[i+1] = A[i]$
    $i = i - 1$
  $A[i + 1] = key$

$j{-}1 \leq A.length \wedge$
$A[1 \dots j{-}1]$ is sorted

$j \leq A.length \wedge$
$A[1 \dots j]$ is sorted

Insertion Sort

INSERTION-SORT(array A)

```
j = 2
while j ≤ A.length
    key = A[j]
    i = j – 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i – 1
    A[i + 1] = key
    j = j + 1
```

$j–1 \le A.length \wedge$
$A[1 \ldots j–1]$ is sorted

$0 \le i \le j–1 \le A.length \wedge$
$A[1 \ldots i]$ is sorted $\wedge$
$\langle key, A[i+2 \ldots j–1]\rangle$ is sorted

$j \le A.length \wedge$
$A[1 \ldots j]$ is sorted

$j–1 \le A.length \wedge$
$A[1 \ldots j–1]$ is sorted

Loop Invariant

- Initialisation: Need to prove: When the loop starts, the loop invariant holds.

- Maintenance: Need to prove: If the loop invariant (and the loop condition) hold at the beginning of the loop iteration, then the invariant holds at the end of the loop iteration.

- Termination: Use the loop invariant (and the negation of loop condition) to prove any property required after the loop.

循环不变式

- 初始化：需要证明：
循环开始的时候，不变式为真。

- 保持：需要证明：
如果某次迭代之前不变式（和循环条件）为真，
那么这次迭代之后它仍为真。

- 终止：使用循环不变式（和循环条件的否定）
证明任何循环后的要求。

Insertion Sort: Loop Invariant

- Loop invariant for insertion sort: At the beginning of an iteration,
  $j–1 \leq A.length \wedge A[1 \ldots j–1]$ is sorted.

  - Initialisation:
    When the loop starts ($j = 2$), $1 \leq A.length \wedge A[1 \ldots 1]$ is sorted.

  - Maintenance:
    If at beginning of an iteration, $j–1 \leq A.length \wedge A[1 \ldots j–1]$ is sorted,
    then at end of the iteration $j \leq A.length \wedge A[1 \ldots j]$ is sorted (before $j = j+1$).

  - Termination: At the end of the last iteration ($j = n$), $A[1 \ldots n]$ is sorted.

Selection Sort

SELECTION-SORT(array A)

for j := 1 **to** $A.length-1$
    $min = j$
    for $i = j+1$ **to** $A.length$
        if $A[i] < A[min]$
            $min = i$
    Swap $A[j]$ and $A[min]$

$A[1 \ldots j-1]$ is sorted $\wedge$ no element of $A[j \ldots n]$ is smaller

$A[1 \ldots j-1]$ is sorted $\wedge$ no element of $A[j \ldots n]$ is smaller $\wedge$ $A[min]$ is minimal in $A[j \ldots i-1]$

$A[1 \ldots j]$ is sorted $\wedge$ no element of $A[j+1 \ldots n]$ is smaller

Runtime analysis of selection sort

- $n = A.length$
- Outer loop runs n–1 times.
- Inner loop runs n–1, n–2, ..., 2, 1 times = $n(n-1)/2 = (n^2 - n)/2$ times total.
- Therefore, the whole algorithm is in $\Theta(n^2)$.
- Even if the array is already sorted, the loops are not left early.
Slower than insertion sort!

Heapsort 堆排序

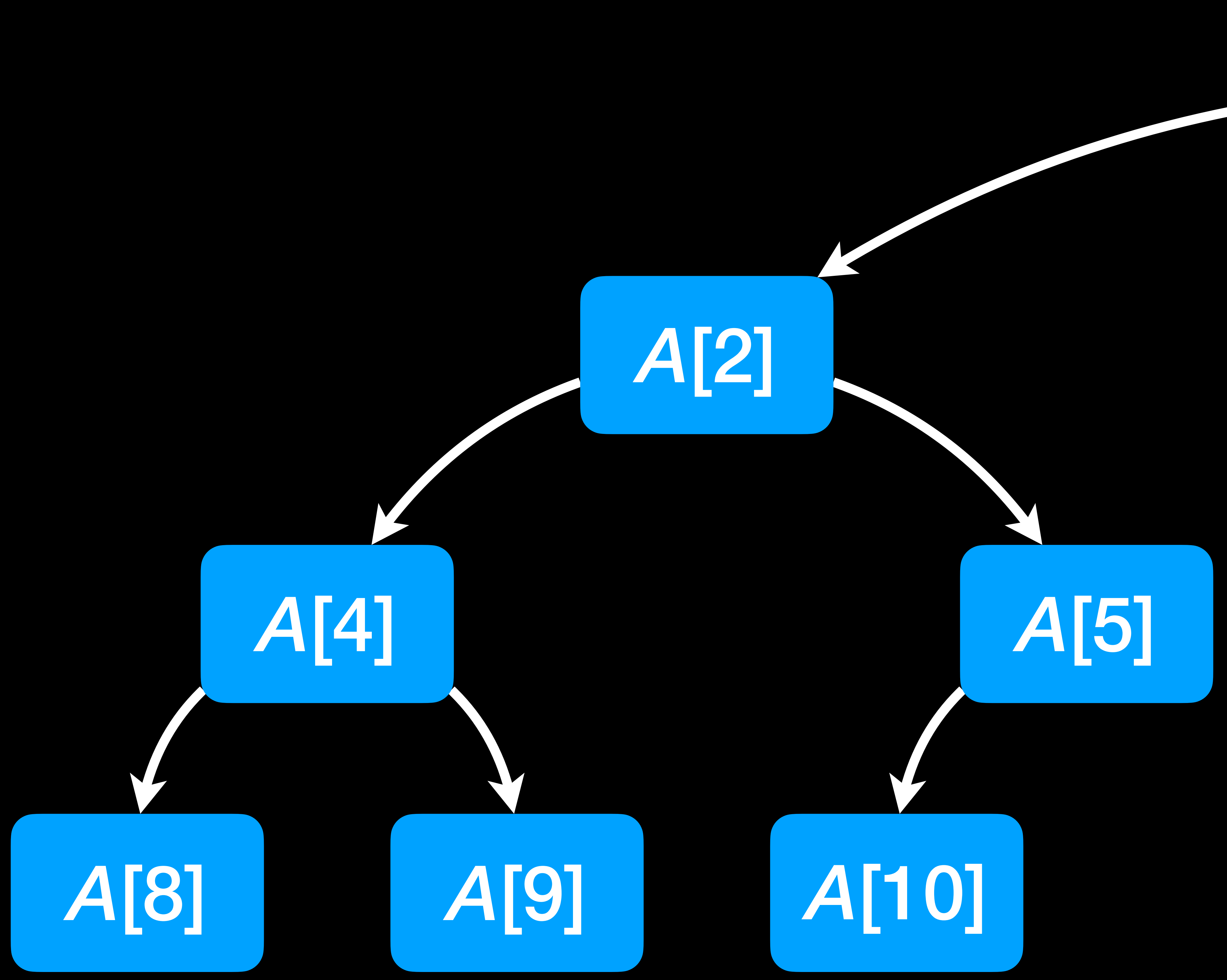
Heap

- Array that can be viewed as a nearly complete binary tree
- root node is $A[1]$
- children of $A[i]$ are $A[2i]$ and $A[2i+1]$
(if $2i$ and $2i+1 \le A.\textit{heap-size}$, resp.)
- parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$ (if $i > 1$)
- $A.\textit{length}$ = allocated size of array
$A.\textit{heap-size}$ = number of elements in heap

堆

- 数组，
可以看成一个近似的完全二叉树
- 根结点是 $A[1]$
- $A[i]$ 的孩子结点是 $A[2i]$ 和 $A[2i+1]$
（如果 $2i$ 和 $2i+1 \le A.\textit{heap-size}$）
- $A[i]$ 的父结点 $A[\lfloor i/2 \rfloor]$（$i > 1$ 的话）
- $A.\textit{length}$ = 数组的分配大小
$A.\textit{heap-size}$ = 堆中的元素数

Heap 堆



A.heap-size = 10

Max-Heap, Min-Heap

- use max-heaps for sorting,
  they satisfy the max-heap property:

  $A[\text{PARENT}(i)] \geq A[i] \quad \text{if } i > 1$

- (Root of contains the largest node.)

- use min-heaps for min-priority queues,
  they satisfy the min-heap property:

  $A[\text{PARENT}(i)] \geq A[i] \quad \text{if } i > 1$

最大堆，最小堆

- 使用最大堆可以排序，
  最大堆满足最大堆性质：

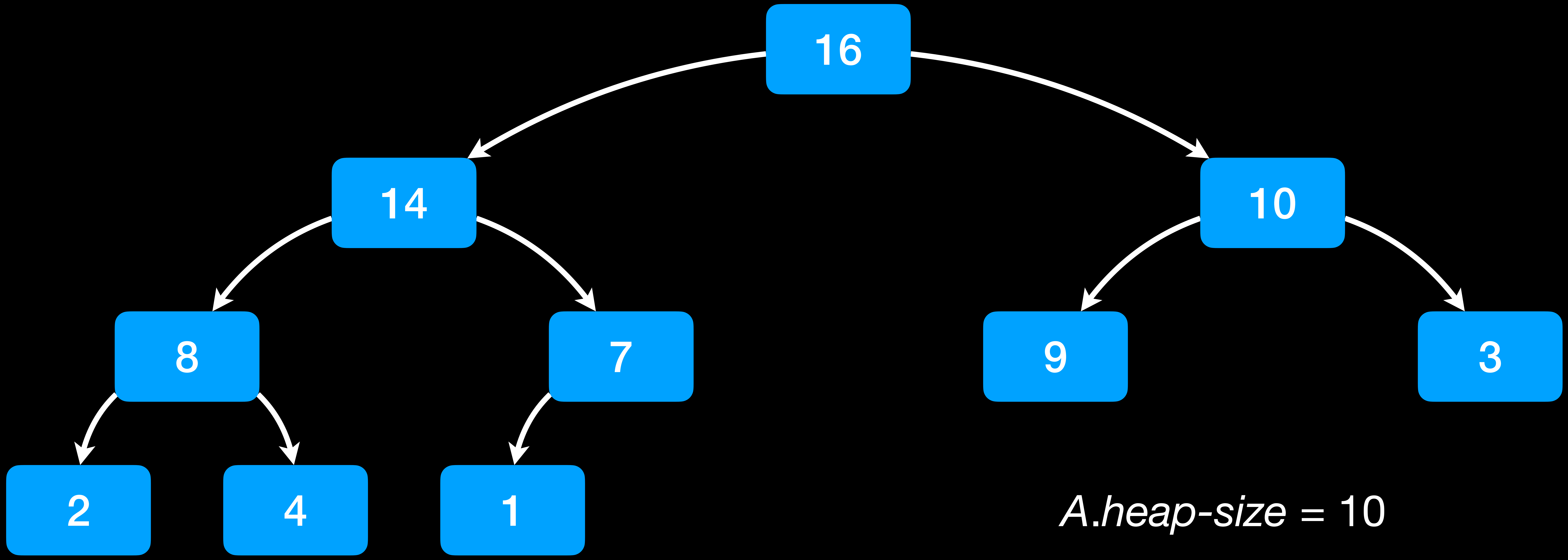
  $A[\text{PARENT}(i)] \geq A[i] \quad$ 如果 $i > 1$

- (最大堆的根保存最大的结点。)

- 使用最小堆可以实现最小优先队列，
  最小堆满足最小堆性质：

  $A[\text{PARENT}(i)] \leq A[i] \quad$ 如果 $i > 1$

Max-Heap 最大堆



A.heap-size = 10

Operations on Max-Heaps

- MAX-HEAPIFY:
  correct one error in a max-heap
  Running time: $O(\log n)$

- BUILD-MAX-HEAP:
  build a heap from an unordered array
  Running time: $O(n)$

- HEAPSORT:
  sort an array using a heap
  Running time: $O(n \log n)$

最大堆的操作

- MAX-HEAPIFY:
  更正最大堆中的一个错误
  运行时间：$O(\log n)$

- BUILD-MAX-HEAP:
  从无序数组构造最大堆
  运行时间：$O(n)$

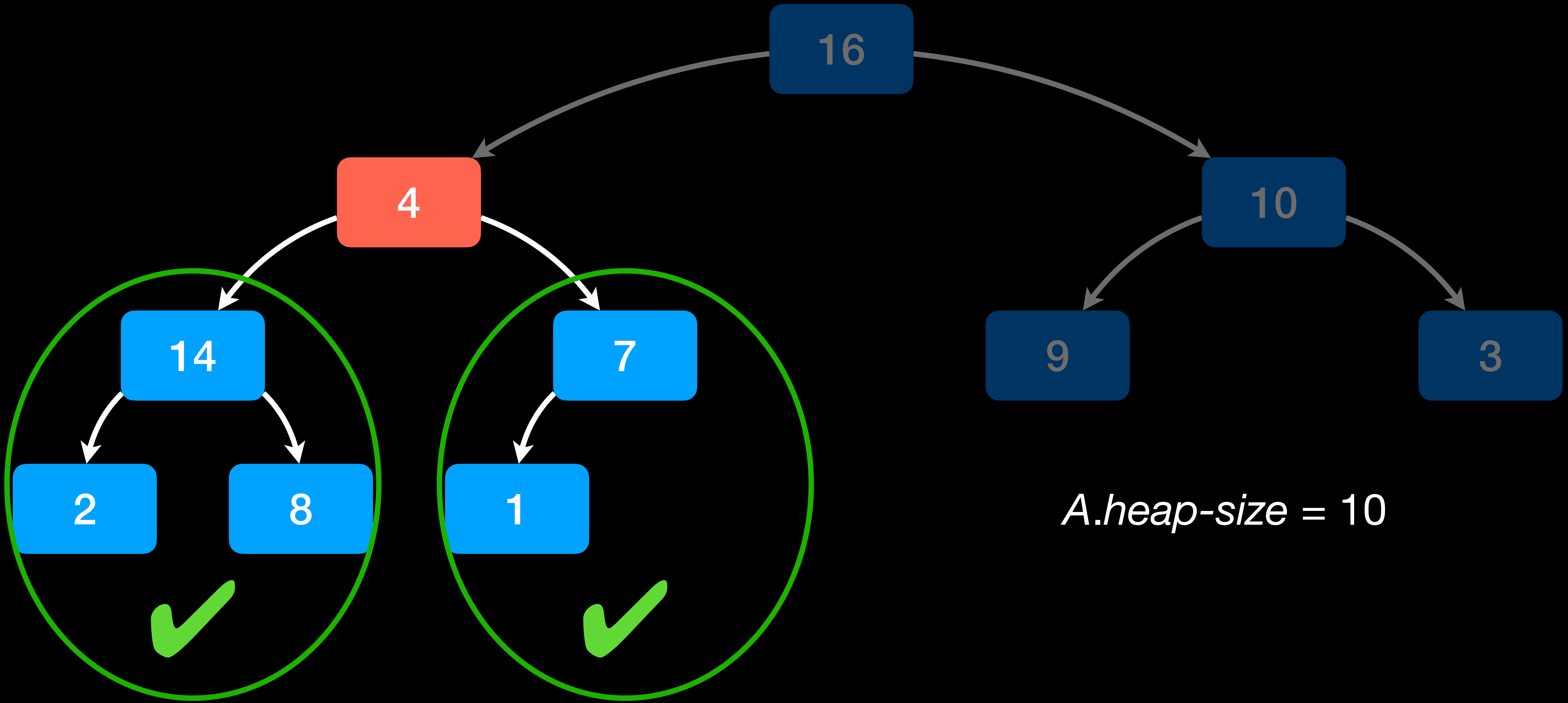
- HEAPSORT:
  使用最大堆对数组进行排序
  运行时间：$O(n \log n)$

MAX-HEAPIFY

- correct an error at $A[i]$
- Precondition: children $A[2i]$ and $A[2i+1]$ are correct max-heaps
(i.e. all their descendants satisfy the max-heap property),
$A[i]$ may be too small.
- Postcondition: $A[i]$ is a correct max-heap.

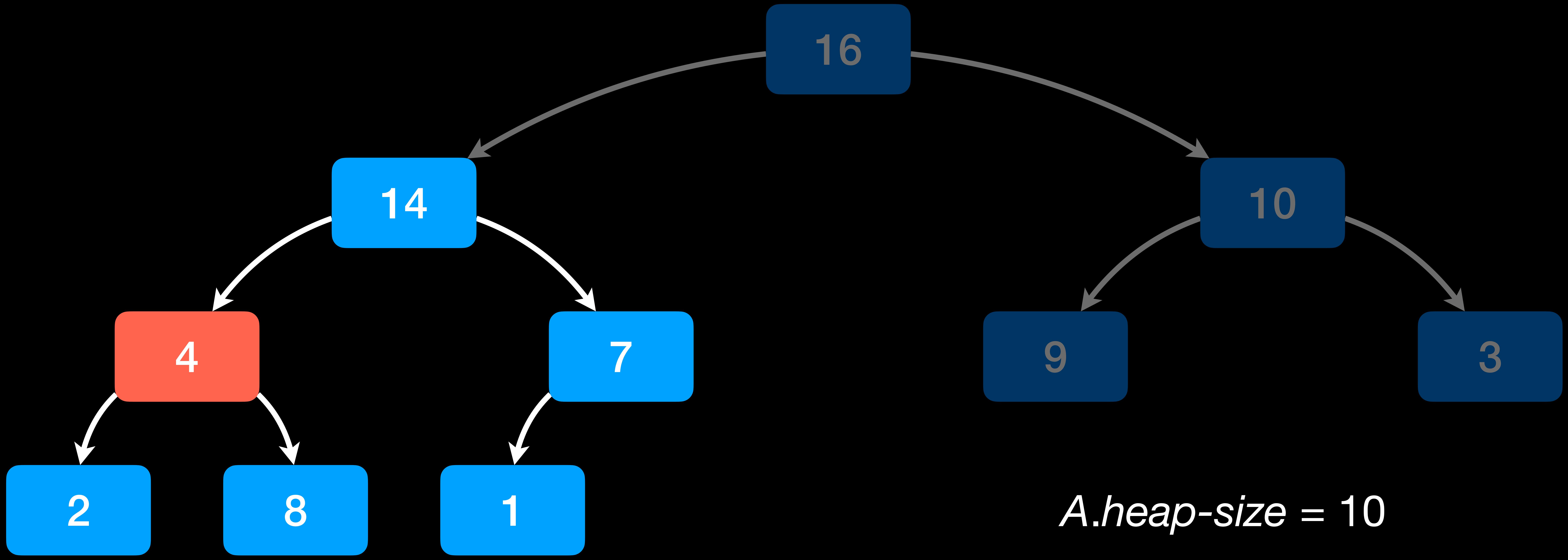
- 更正最大堆中 $A[i]$ 的错误
- 先条件：孩子结点 $A[2i]$ 和 $A[2i+1]$ 是正确的最大堆
（他们的子孙结点都满足最大堆性质）
但是可能 $A[i]$ 小于他的孩子结点。
- 后条件：$A[i]$ 是正确的最大堆。

MAX-HEAPIFY



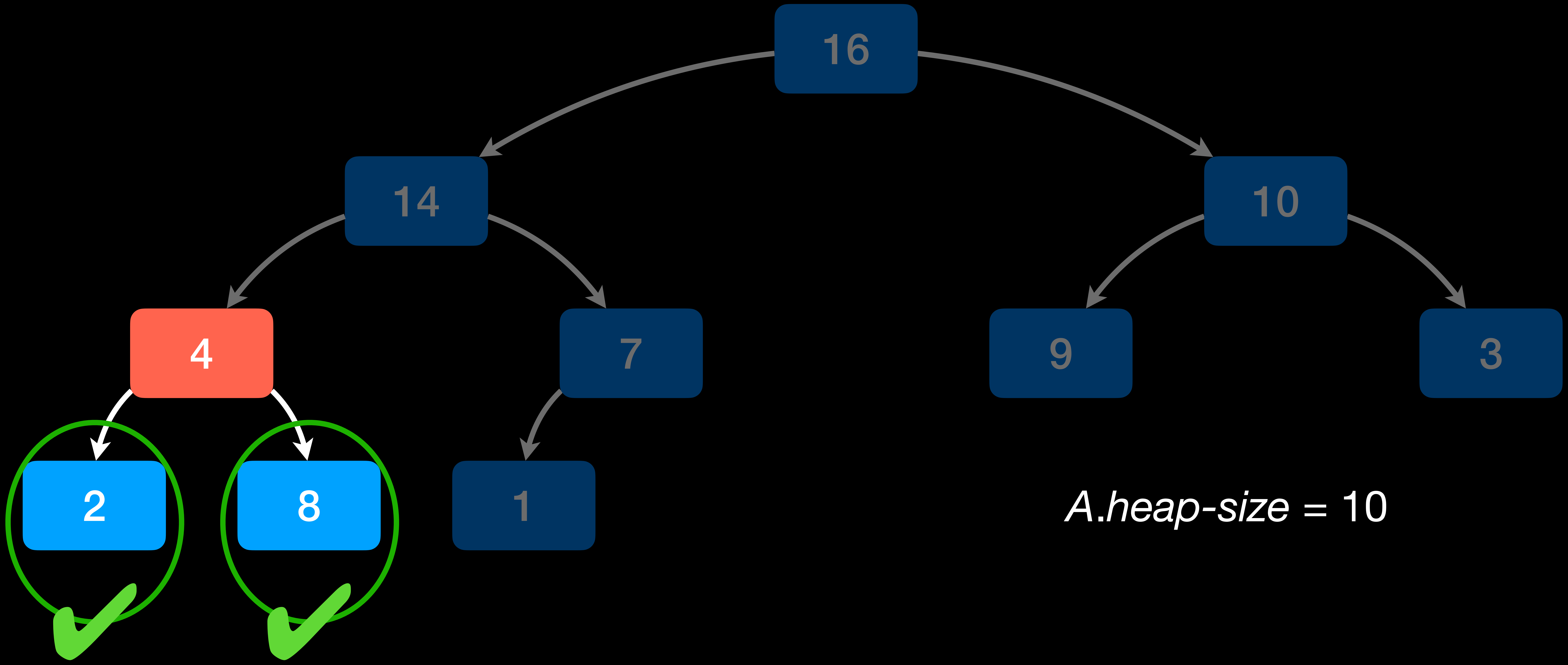
A.heap-size = 10

Max-Heapify



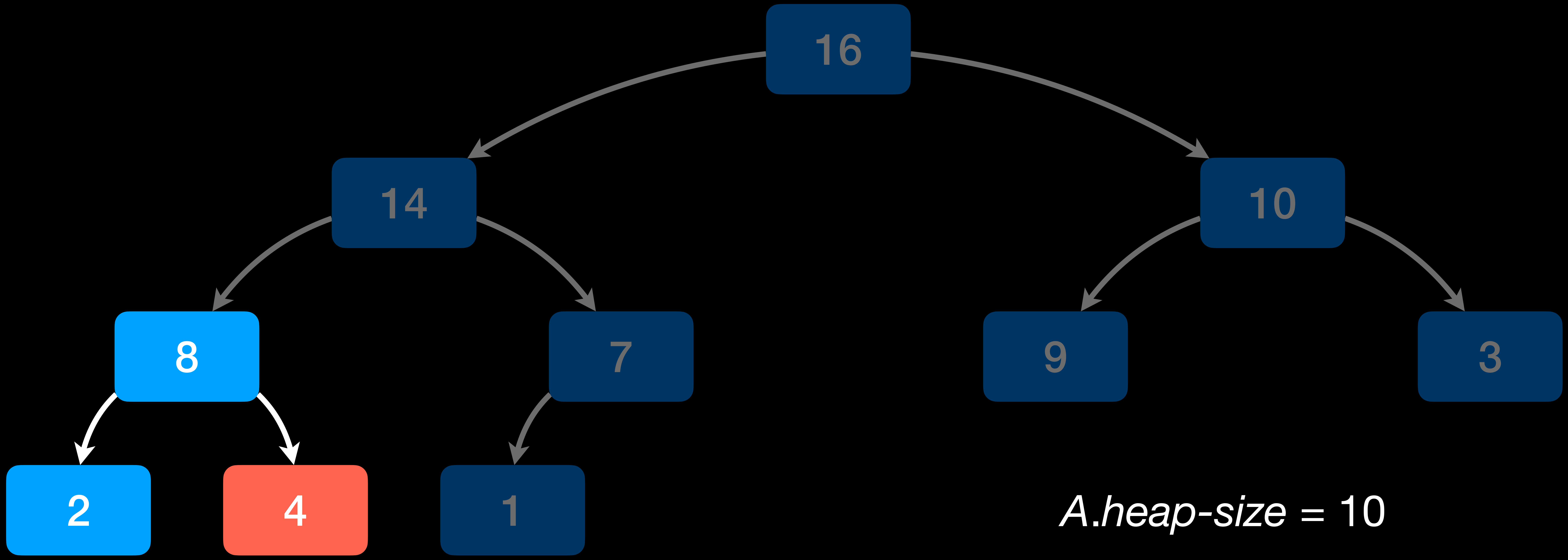
A.heap-size = 10

MAX-HEAPIFY



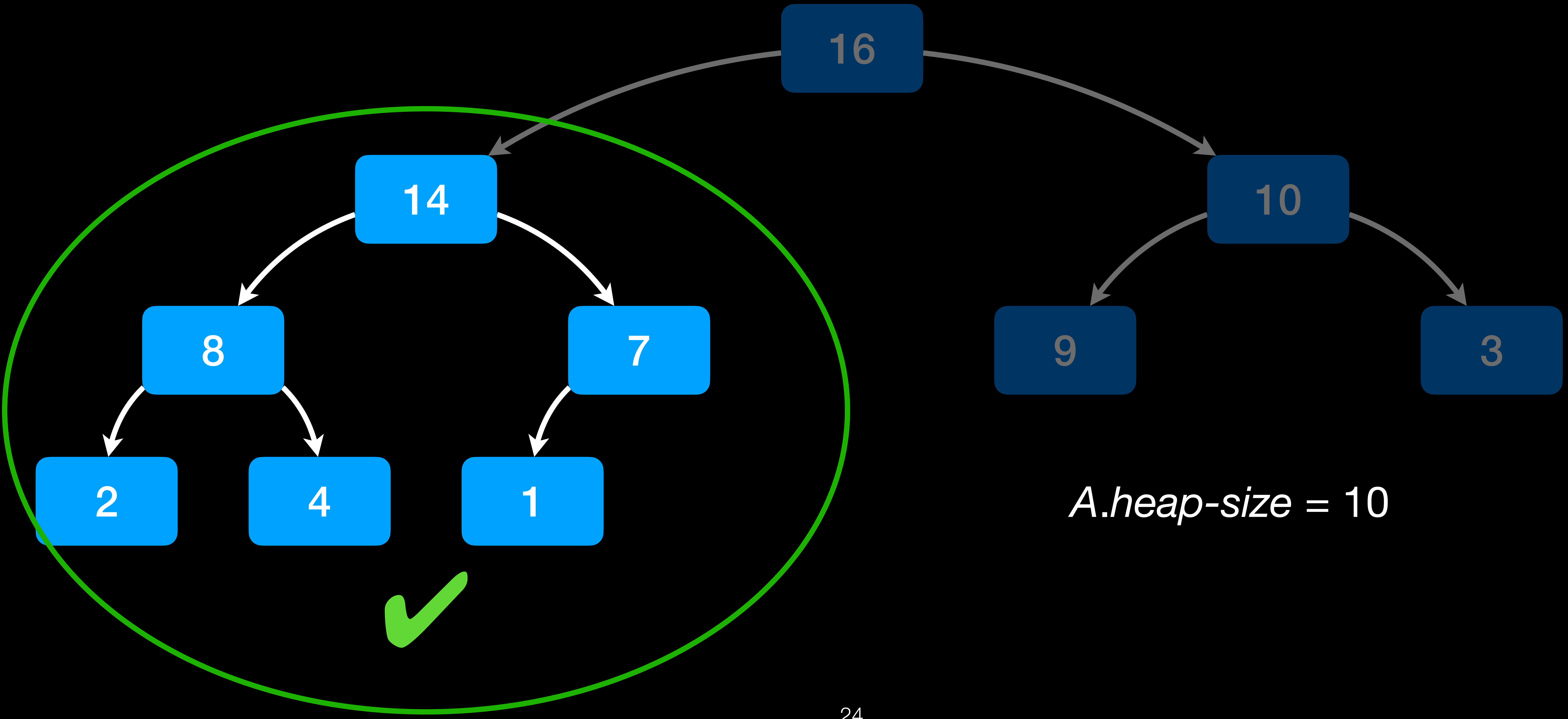
A.heap-size = 10

MAX-HEAPIFY



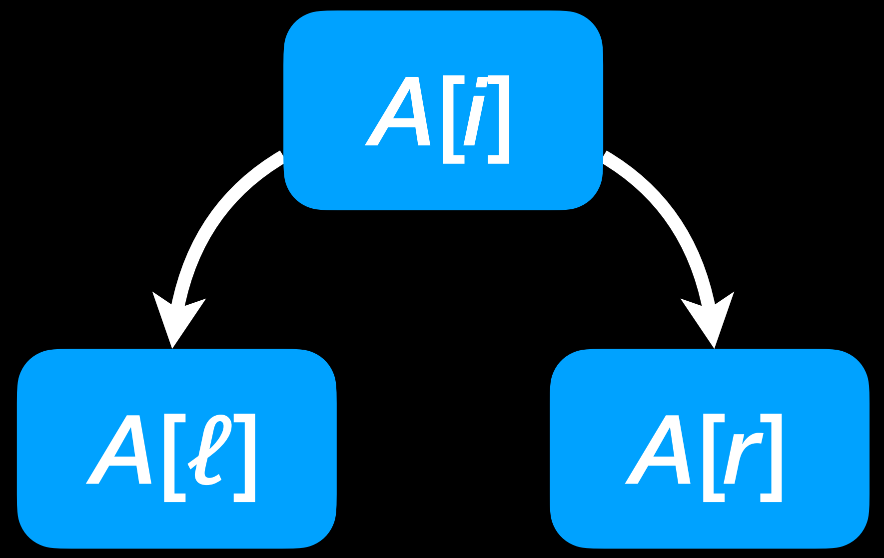
A.heap-size = 10

Max-Heapify



$A.heap\text{-}size = 10$

MAX-HEAPIFY



Find the largest of $A[i]$, $A[\ell]$, $A[r]$

MAX-HEAPIFY(A, i)
ℓ = LEFT(i) // $\ell = 2i$
r = RIGHT(i) // $r = 2i + 1$
if $l \le A.heap\text{-}size$ **and** $A[\ell] > A[i]$
    $largest = \ell$
else $largest = i$
if $r \le A.heap\text{-}size$ **and** $A[r] > A[largest]$
    $largest = r$
if $largest \ne i$
    Exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY(A, $largest$)

$A[i]$、$A[\ell]$、$A[r]$中找到最大的元素

BUILD-MAX-HEAP

- build a heap from an unordered array
- Idea:
  - 1-element trees are correct heaps
  - use MAX-HEAPIFY to construct heaps from smaller correct ones
  - repeat constructing heaps until *A*[1] is root of a correct heap

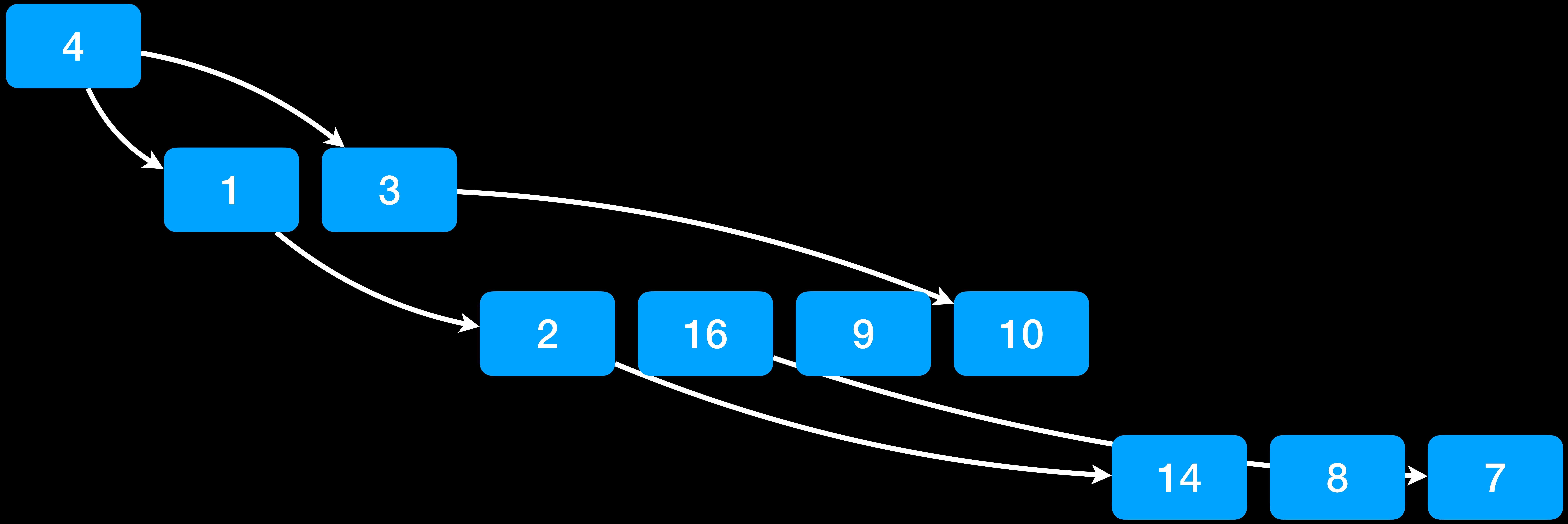
- 从无序数组构造最大堆
- 主意：
  - 1个元素的树是正确的堆
  - 使用 MAX-HEAPIFY 从较小的正确堆构造堆
  - 重复建造堆直到 *A*[1]

BUILD-MAX-HEAP

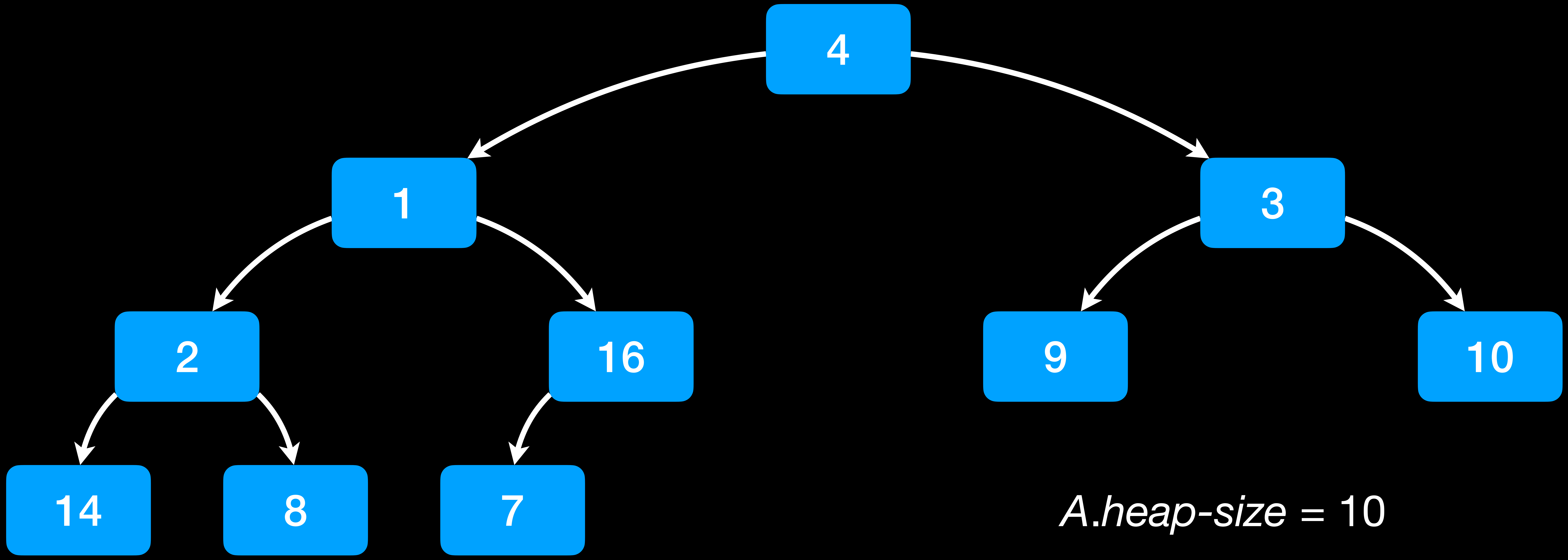
4	1	3	2	16	9	10	14	8	7

$A.length = 10$

BUILD-MAX-HEAP

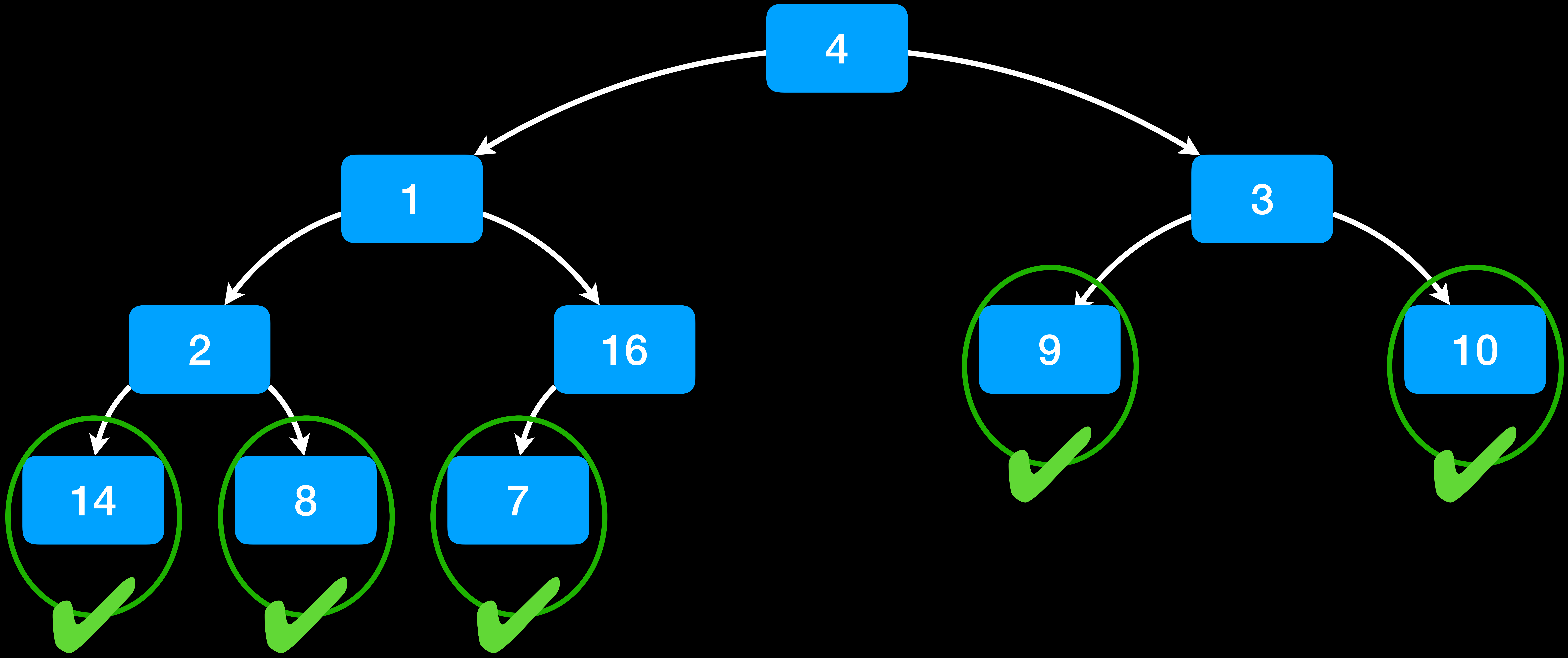


BUILD-MAX-HEAP

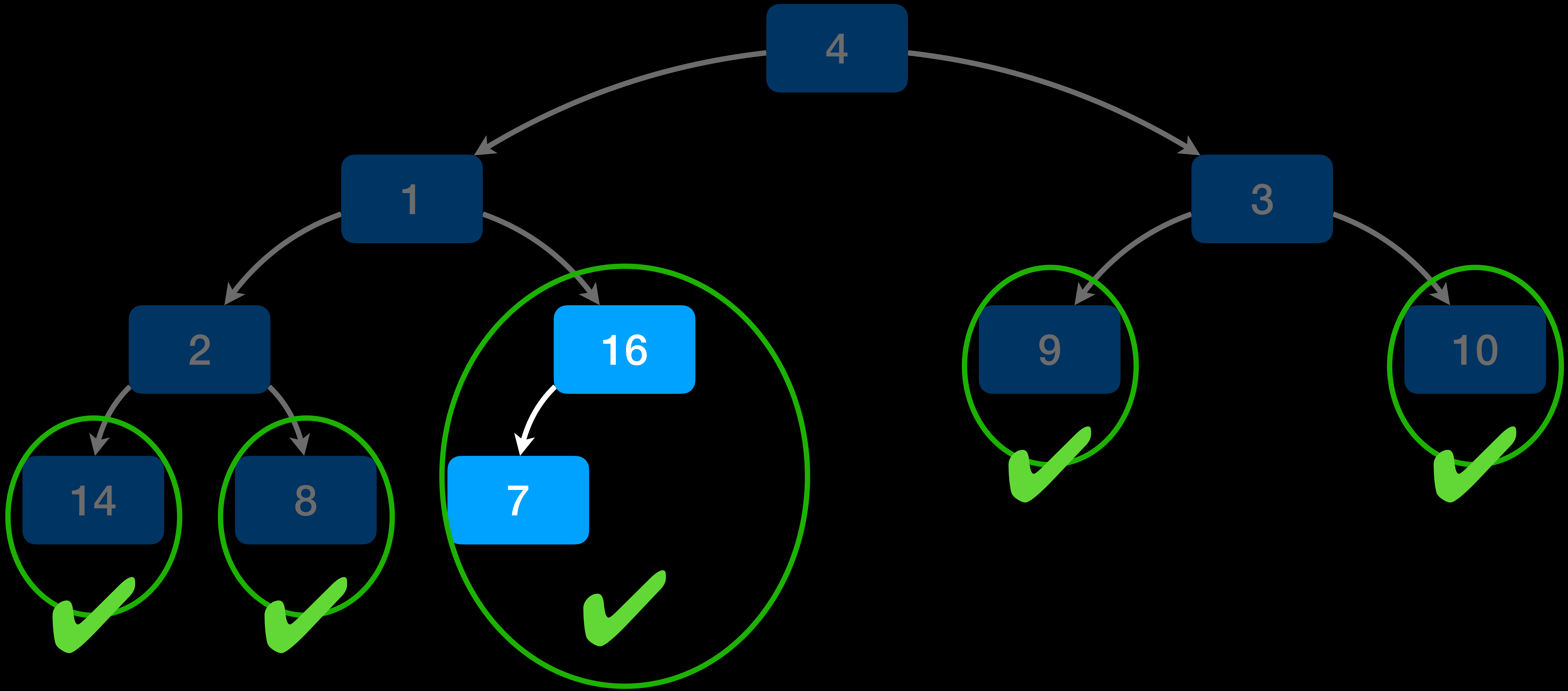


A.heap-size = 10

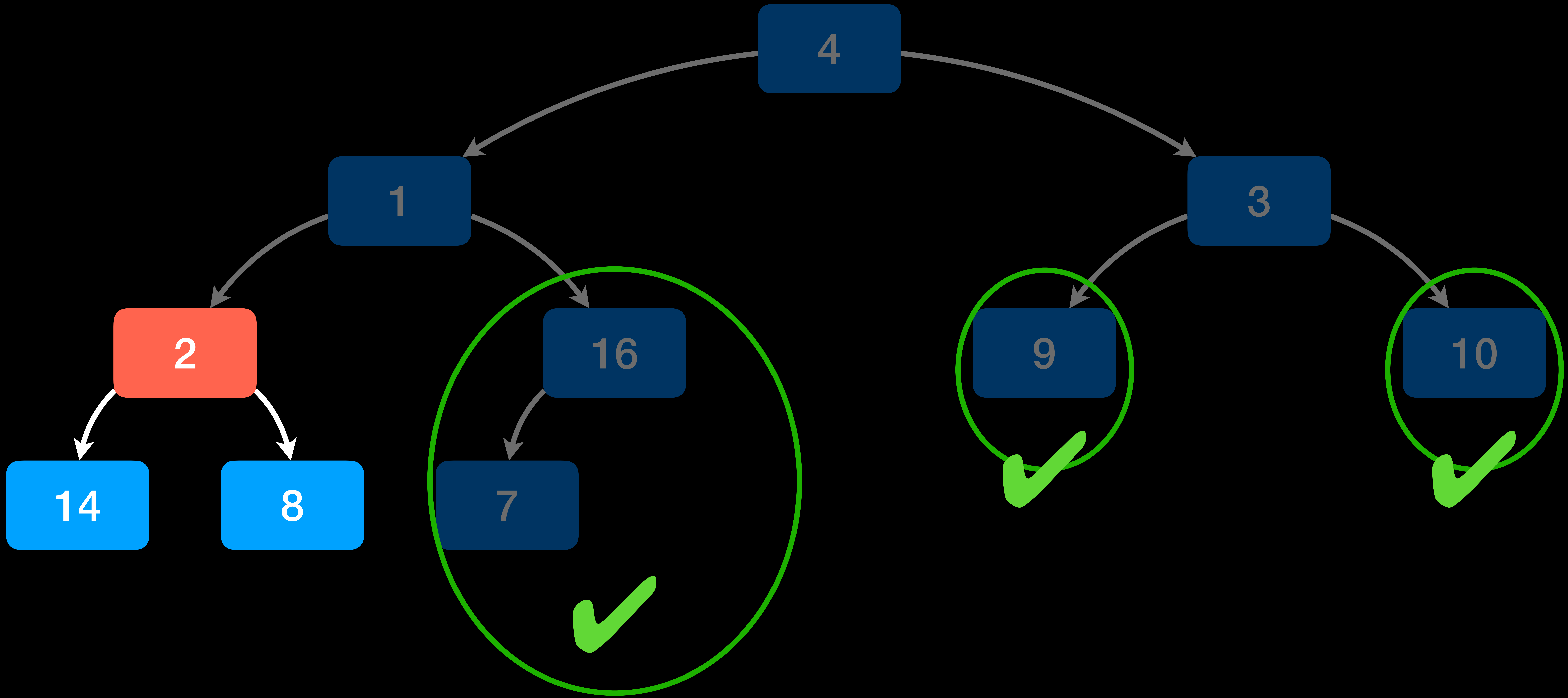
BUILD-MAX-HEAP



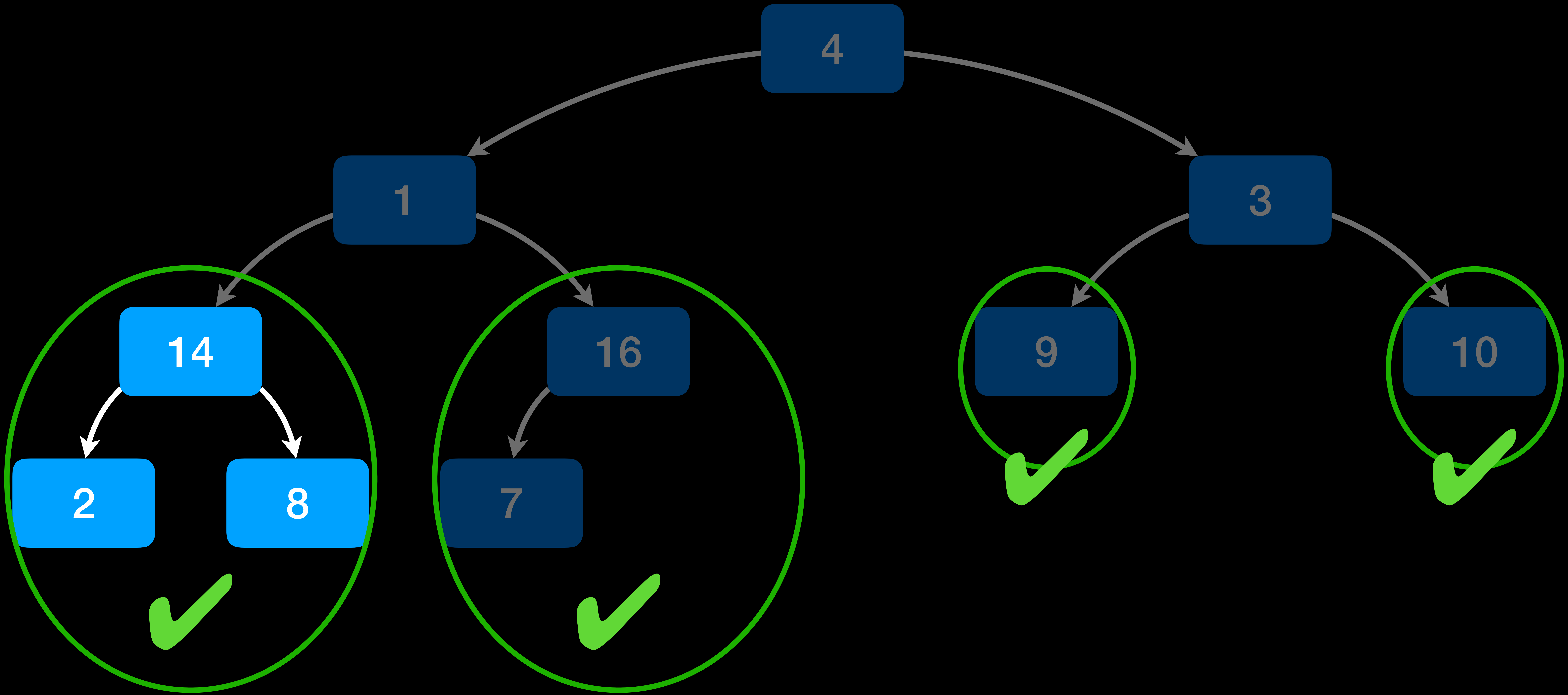
BUILD-MAX-HEAP



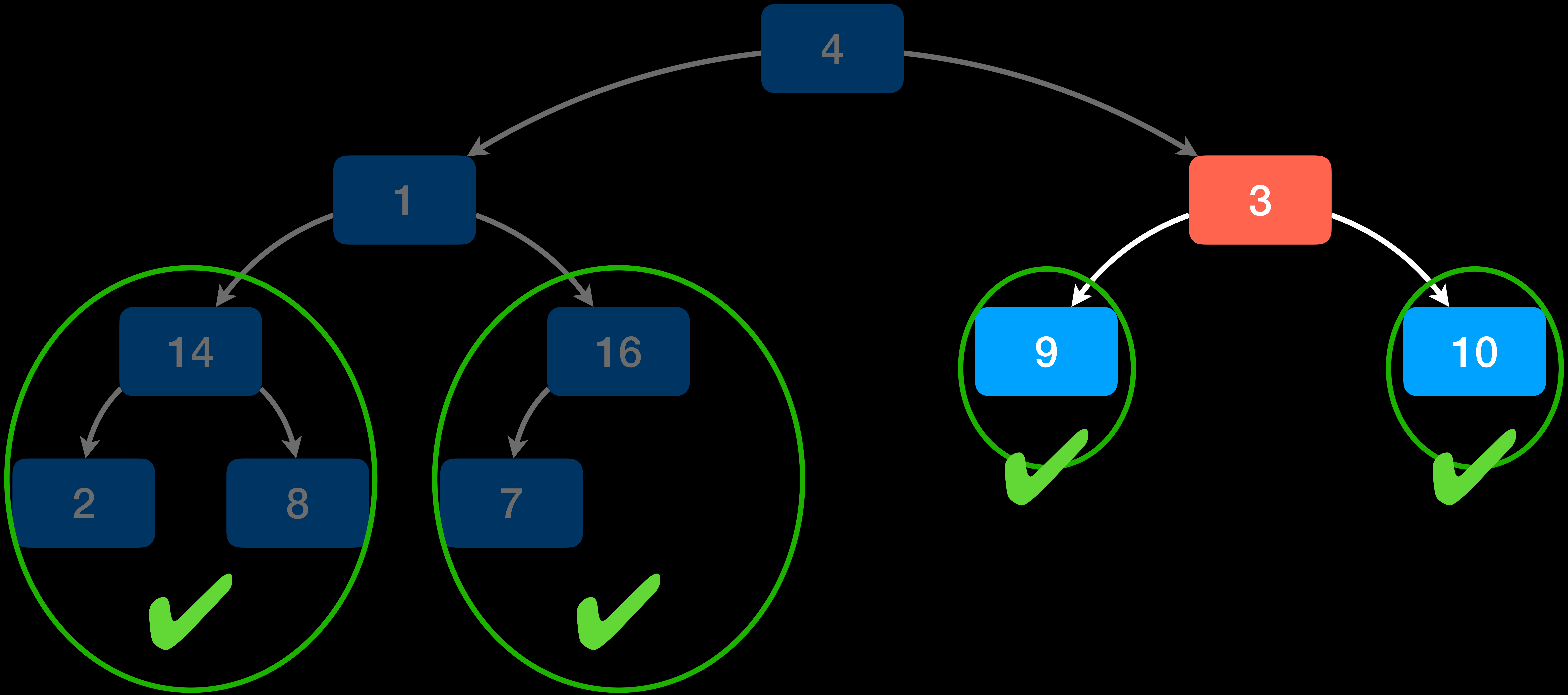
BUILD-MAX-HEAP



BUILD-MAX-HEAP



BUILD-MAX-HEAP



BUILD-MAX-HEAP

BUILD-MAX-HEAP

Build-Max-Heap

BUILD-MAX-HEAP

BUILD-MAX-HEAP

Build-Max-Heap

BUILD-MAX-HEAP

BUILD-MAX-HEAP

BUILD-MAX-HEAP

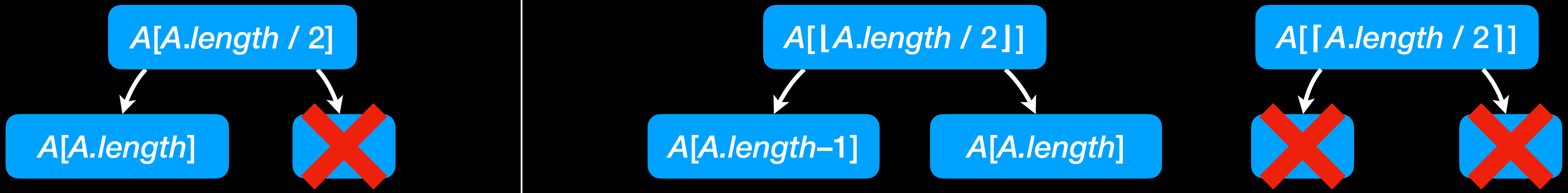
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = PARENT(A.length)          // ⌊A.length / 2⌋
                    downto 1
     MAX-HEAPIFY(A, i)
```

$A[i+1], \ldots, A[A.length]$ are roots of correct max-heaps.

$A[i+1], \ldots, A[A.length]$ 都是正确的最大堆的根。

A.length is even 偶数

A.length is odd 奇数



BUILD-MAX-HEAP

- Loop invariant: $A[i+1], ..., A[A.length]$ are roots of correct max-heaps.

- Initialisation: The loop invariant holds because the nodes $A[\lfloor A.length / 2\rfloor+1], ..., A[A.length]$ have no children.

- Maintenance: MAX-HEAPIFY(A, i) ensures that $A[i]$ becomes a correct max-heap.

- Termination: $A[1]$ is a root of a correct max-heap. ➡ The whole heap is correct.

- 循环不变式：$A[i+1], ..., A[A.length]$ 都是正确的最大堆的根。

- 初始化：循环不变式为真因为结点 $A[\lfloor A.length / 2\rfloor+1], ...,$ $A[A.length]$ 没有孩子结点。

- 保特：MAX-HEAPIFY(A, i) 确保 $A[i]$ 成为正确的最大堆。

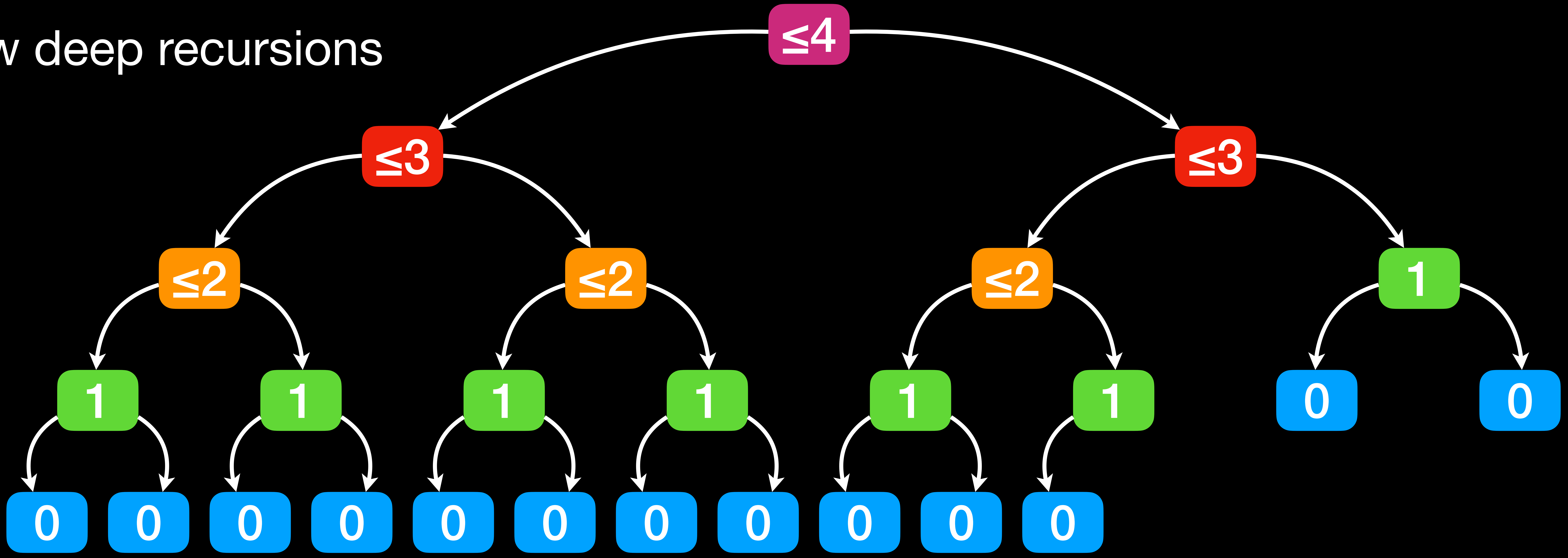
- 终止：$A[1]$ 是正确的最大堆的根。➥ 整个最大堆是正确的。

Running Time 运行时间

- How many calls to MAX-HEAPIFY including recursive calls?
  - $i = \lfloor n/4 \rfloor + 1, \ldots, \lfloor n/2 \rfloor$: no recursive calls
  - $i = \lfloor n/8 \rfloor + 1, \ldots, \lfloor n/4 \rfloor$: at most 1 recursive call
- (MAX-HEAPIFY contains no loops, so time is in O(calls to MAX-HEAPIFY).)

Running Time 运行时间

- How many calls to MAX-HEAPIFY including recursive calls?
- Few deep recursions



Running Time 运行时间

- How many calls to MAX-HEAPIFY including recursive calls?
  - overall upper bound:

$$\sum_{h=1}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \, h = O(n \, ½ \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n)$$

Note: $\sum_{h=1}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \le \sum_{h=1}^{\infty} \frac{h}{2^h} =: p$, then $p = ½ \sum_{h=1}^{\infty} \frac{h-1}{2^{h-1}} + \frac{1}{2^{h-1}} = ½p + \frac{2}{2} ➨ p = 2$

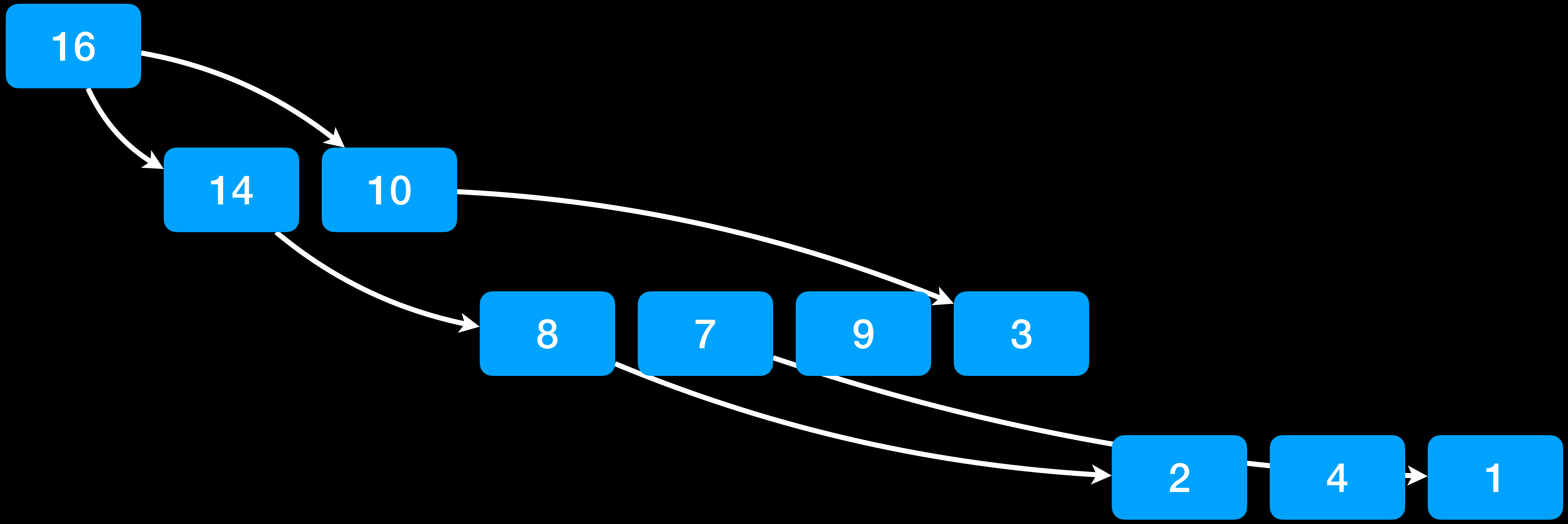
Heapsort

- sort an array using a heap
- Idea:
  1. use Build-Max-Heap to construct a heap
  2. repeatedly remove the largest element (= the root of the heap) and correct the heap with Max-Heapify

- 使用最大堆对数组进行排序

Result of BUILD-MAX-HEAP

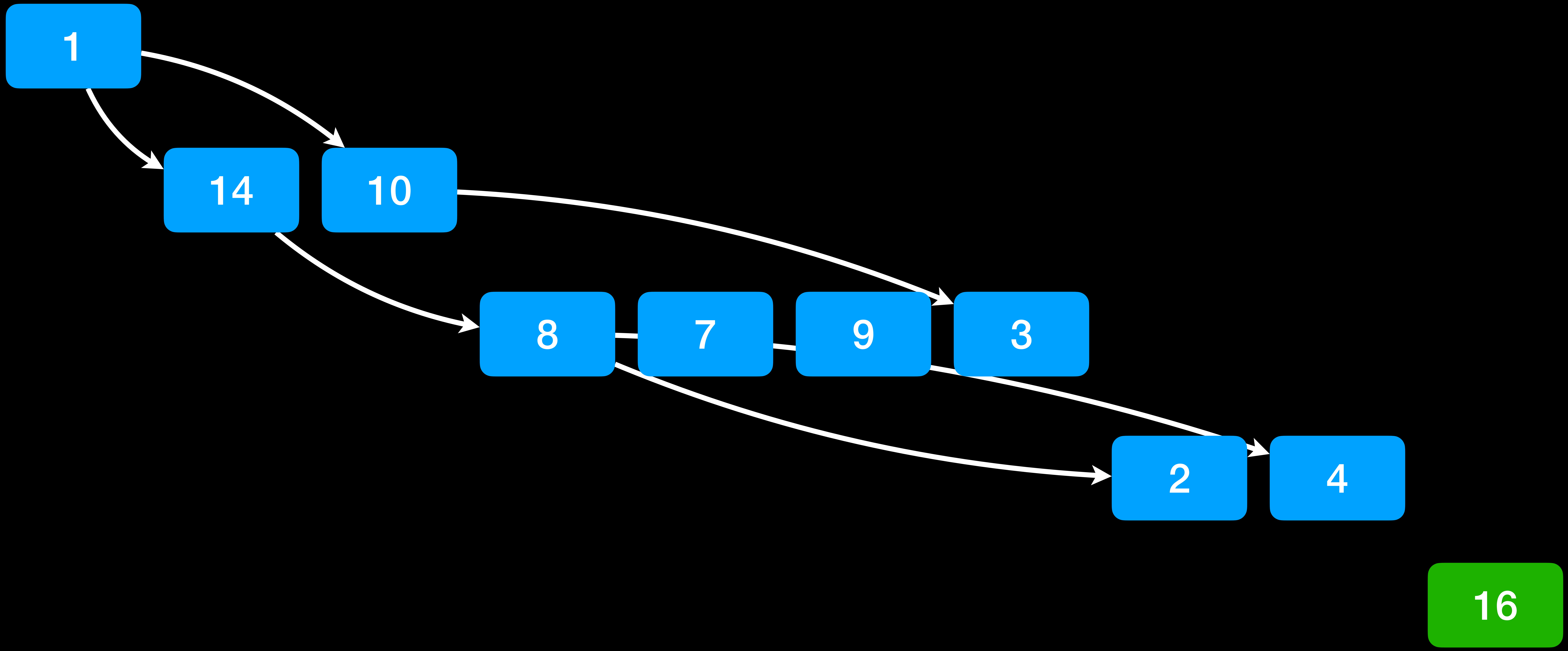
Result of BUILD-MAX-HEAP



Remove largest element 移开最大的元素

16	14	10	8	7	9	3	2	4	1

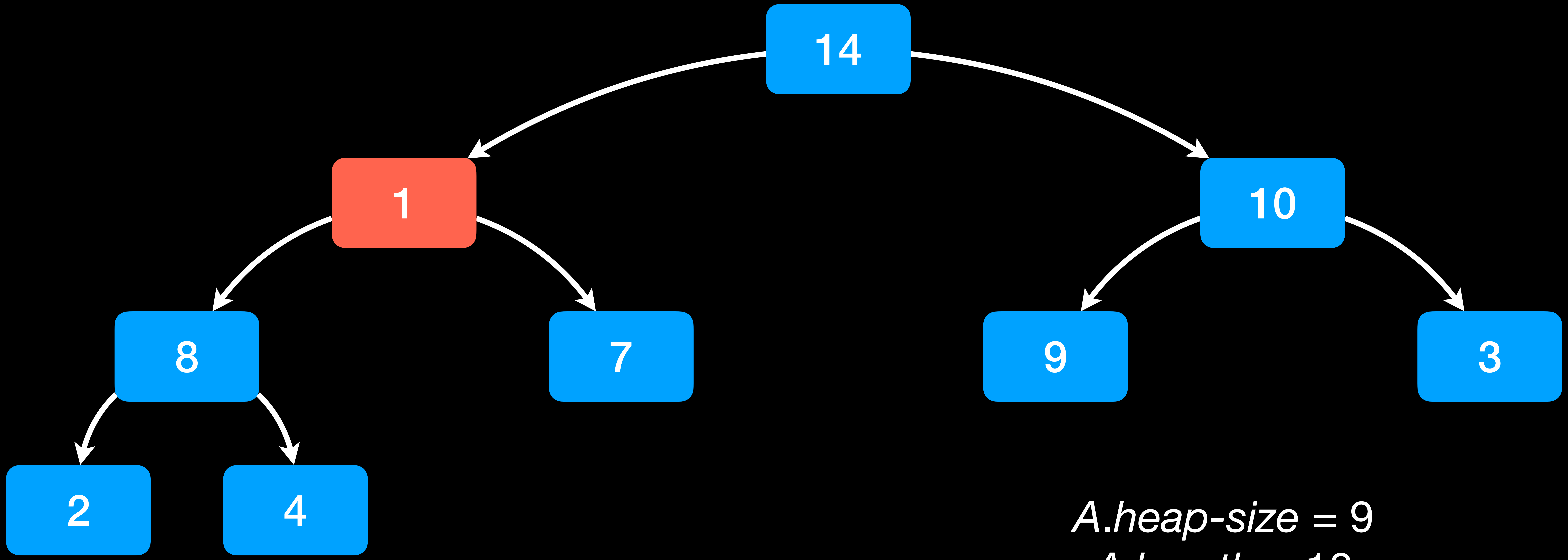
Remove largest element 移开最大的元素



Correct error in heap 纠正堆内的错误

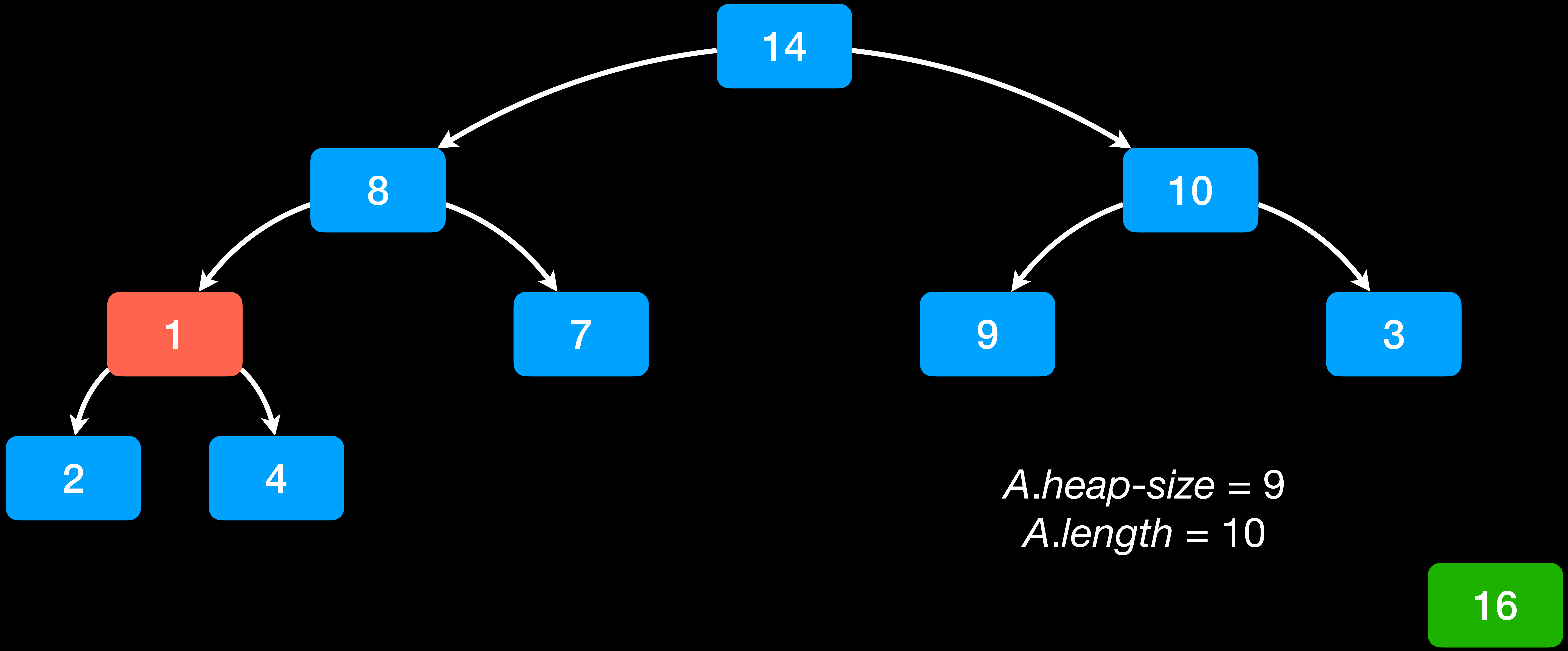
A.heap-size = 9
A.length = 10

Correct error in heap 纠正堆内的错误



A.heap-size = 9
A.length = 10

Correct error in heap 纠正堆内的错误



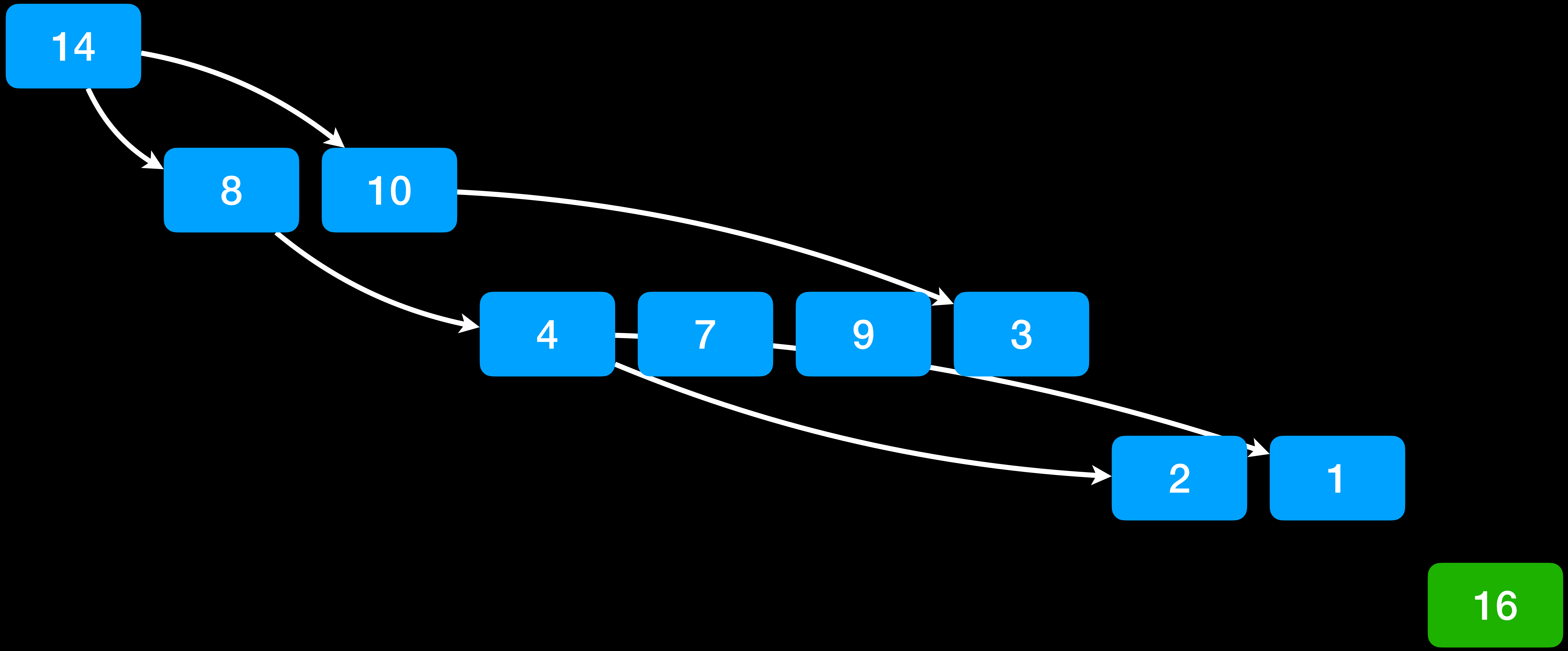
A.heap-size = 9
A.length = 10

Heap is correct 堆正确的

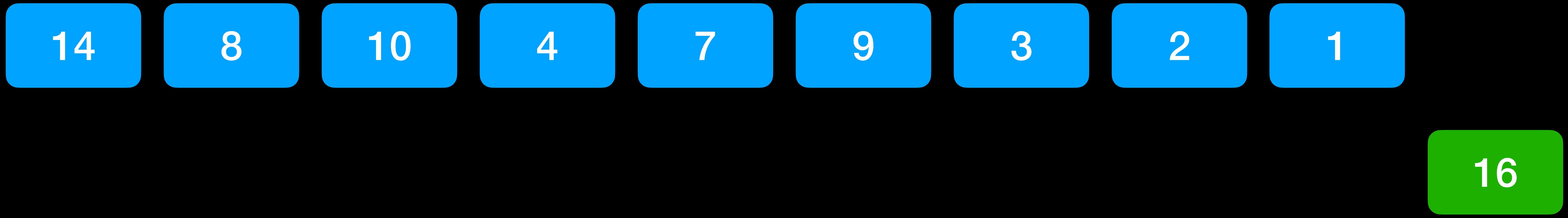
A.heap-size = 9
A.length = 10

✔

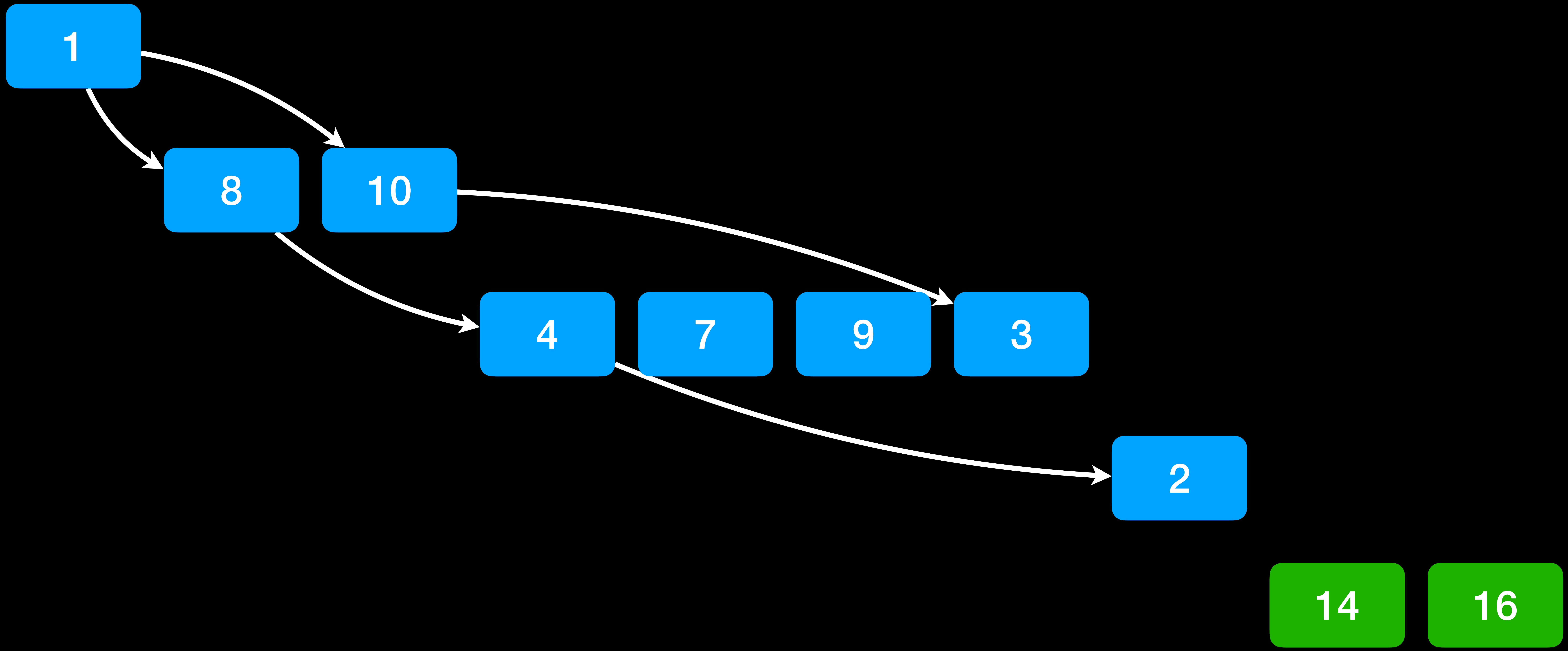
Remove largest element 移开最大的元素



Remove largest element 移开最大的元素

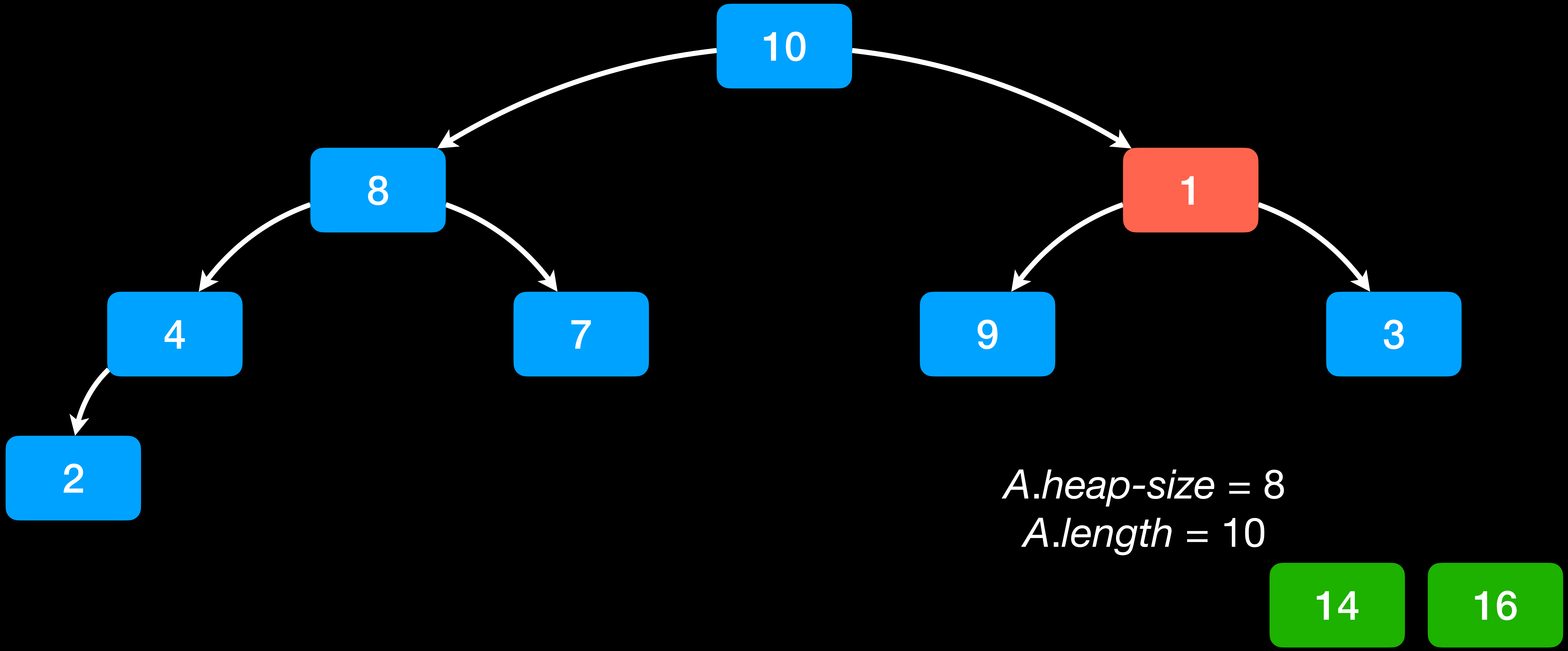


Remove largest element 移开最大的元素



Correct error in heap 纠正堆内的错误

Correct error in heap 纠正堆内的错误



Heap is correct　堆正确的

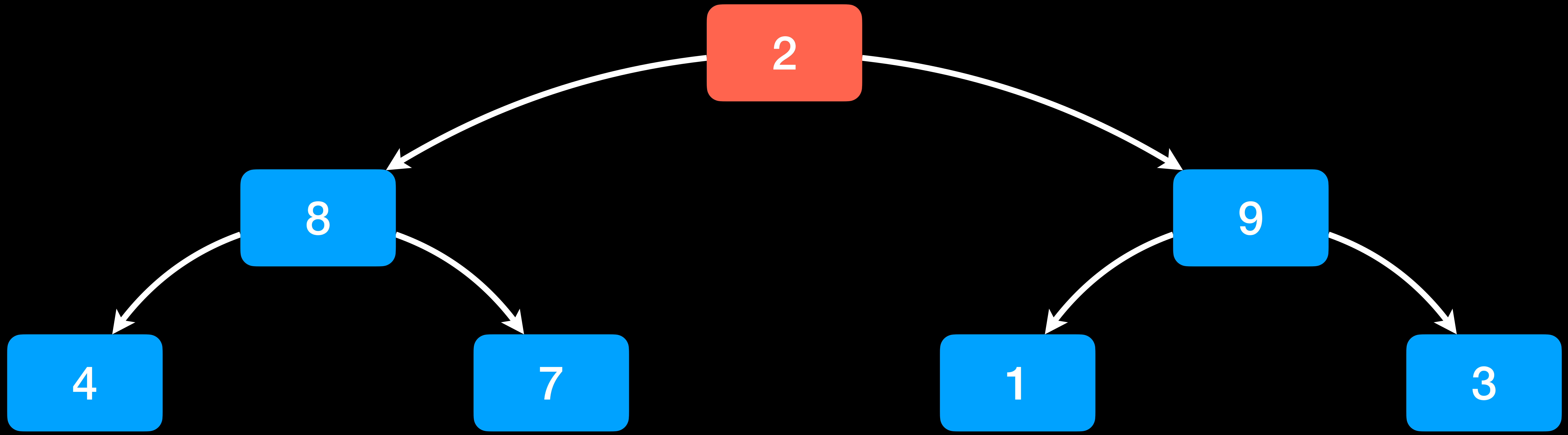
$A.heap\text{-}size = 8$
$A.length = 10$

Remove largest element 移开最大的元素

A.heap-size = 8
A.length = 10

✔

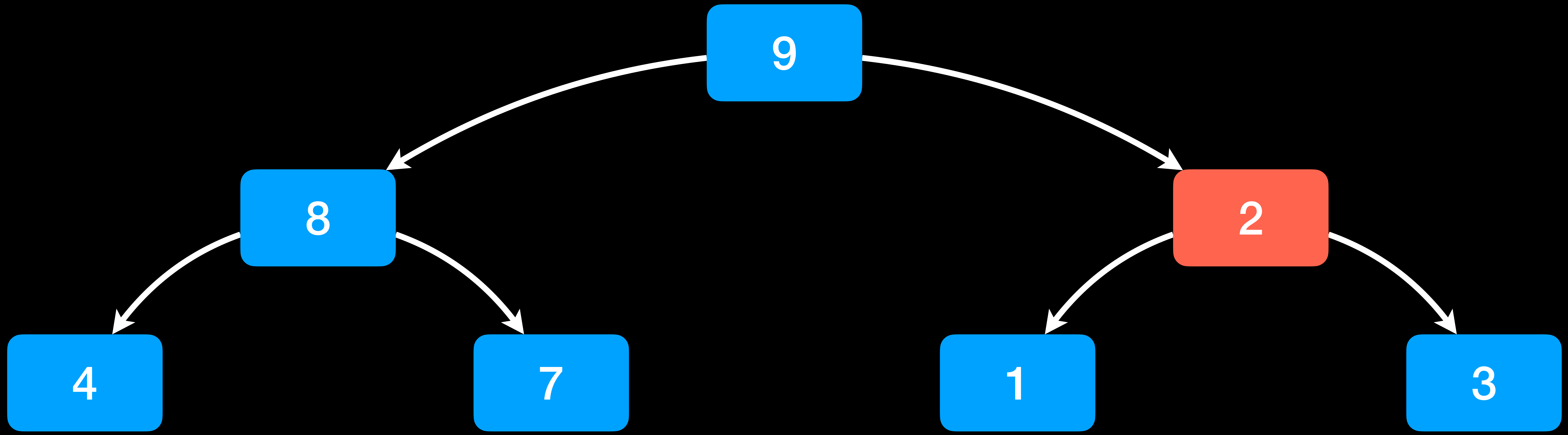
Remove largest element 移开最大的元素



A.heap-size = 7

A.length = 10

Correct error in heap 纠正堆内的错误

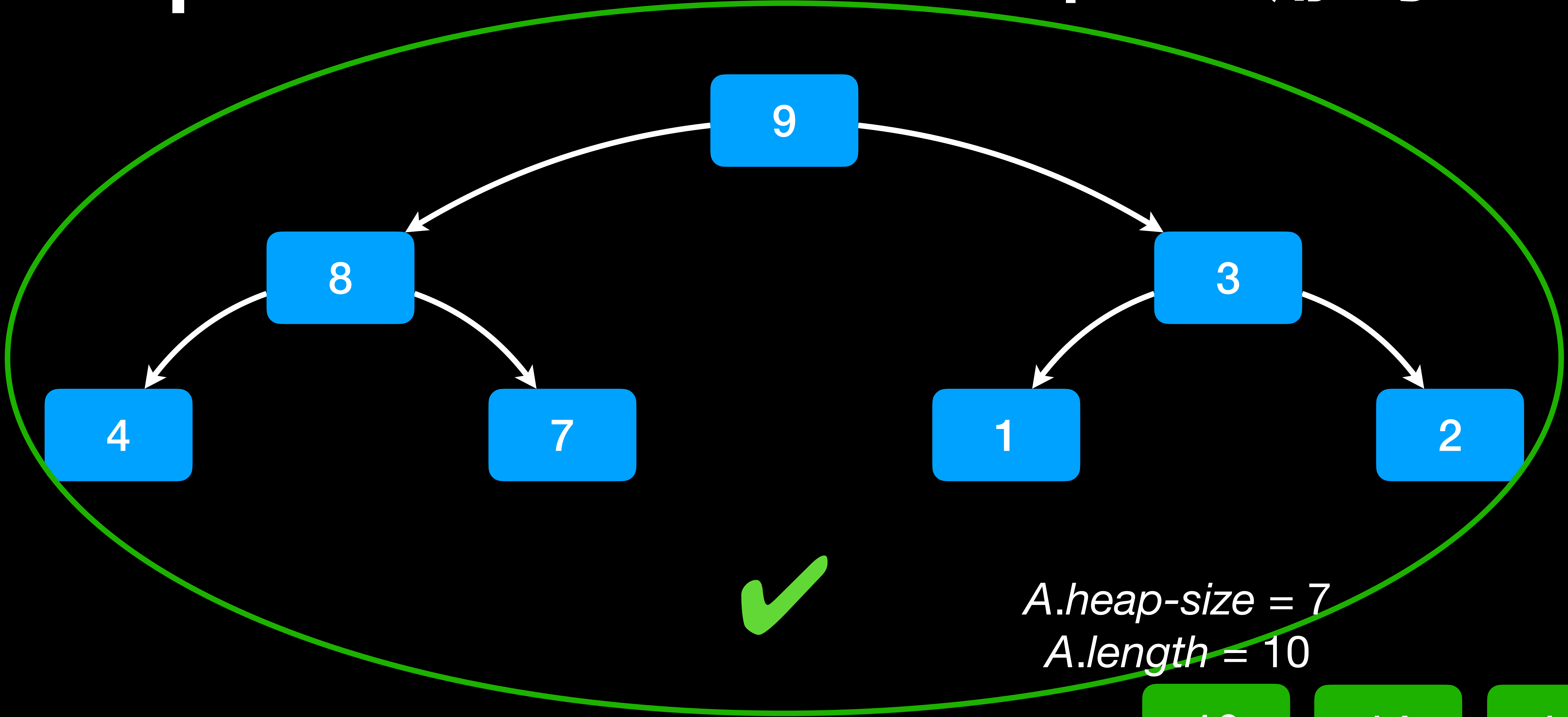


$A.heap\text{-}size = 7$

$A.length = 10$

10 14 16

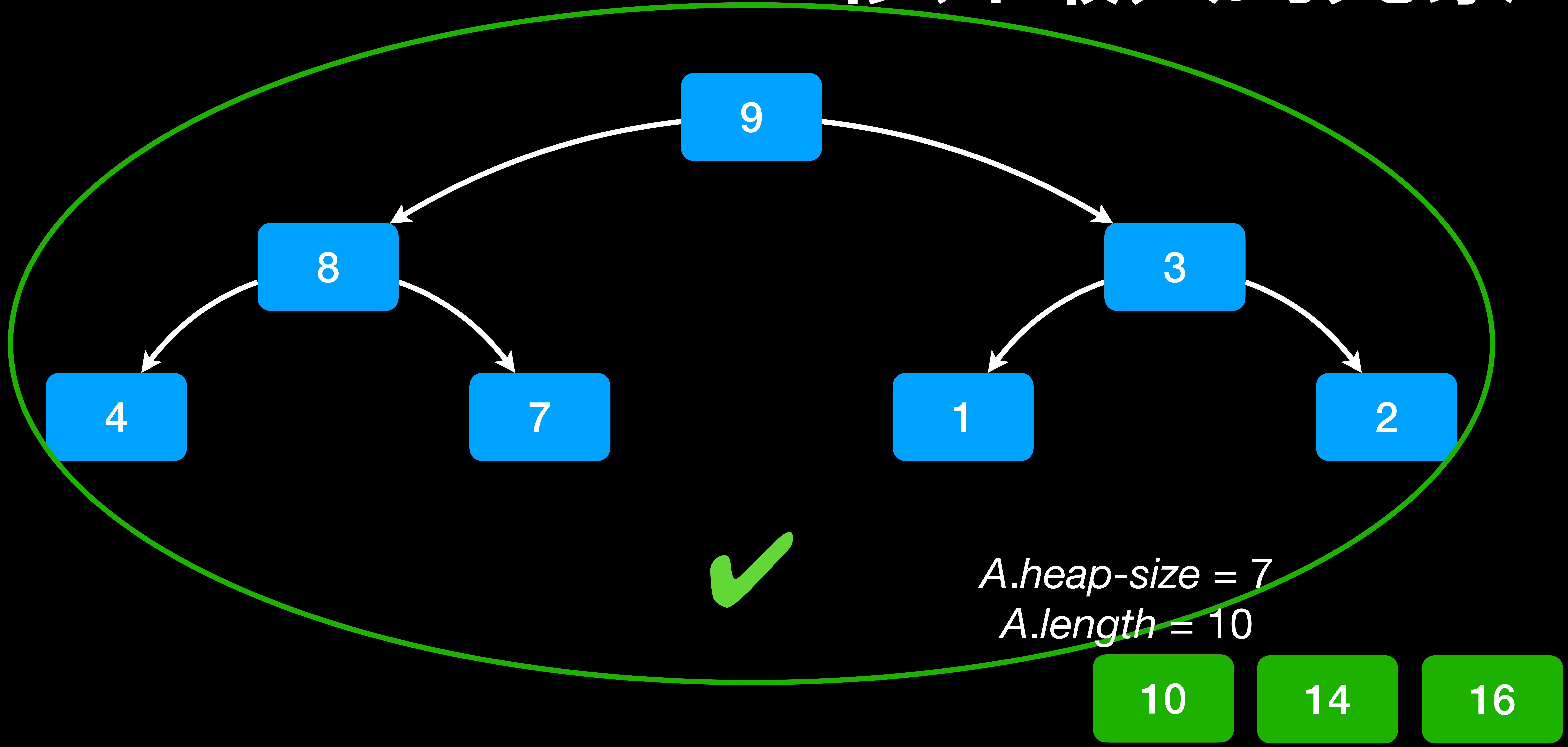
Heap is correct 堆正确的



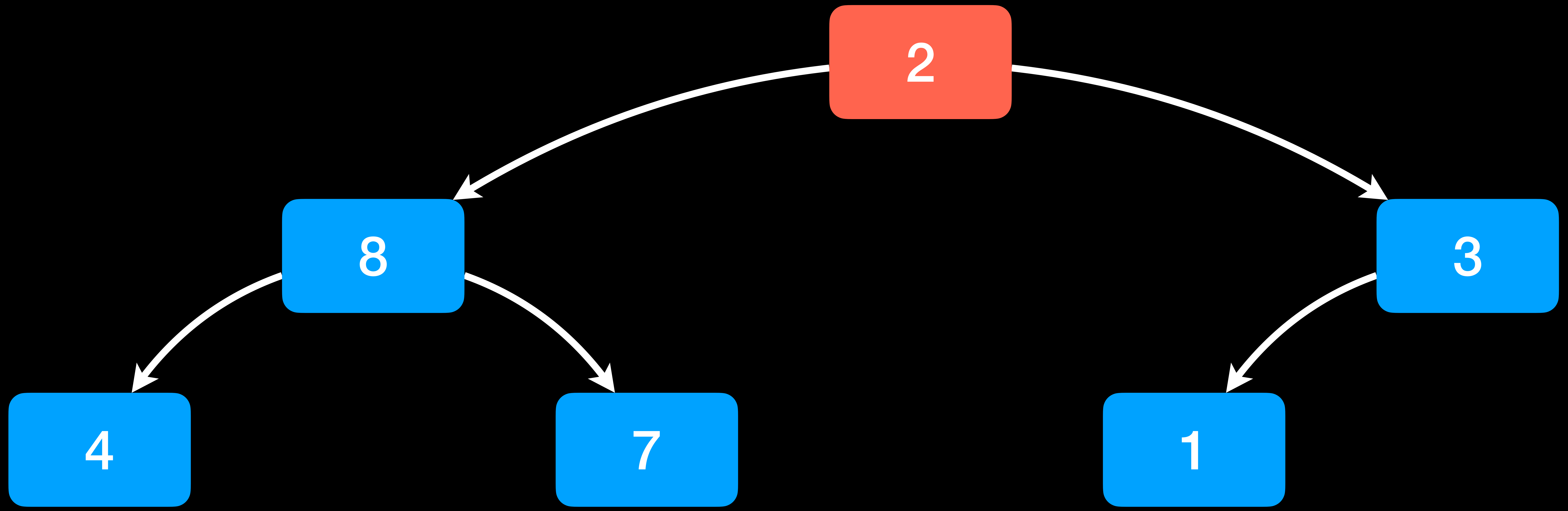
A.heap-size = 7
A.length = 10

10	14	16

Remove largest element 移开最大的元素

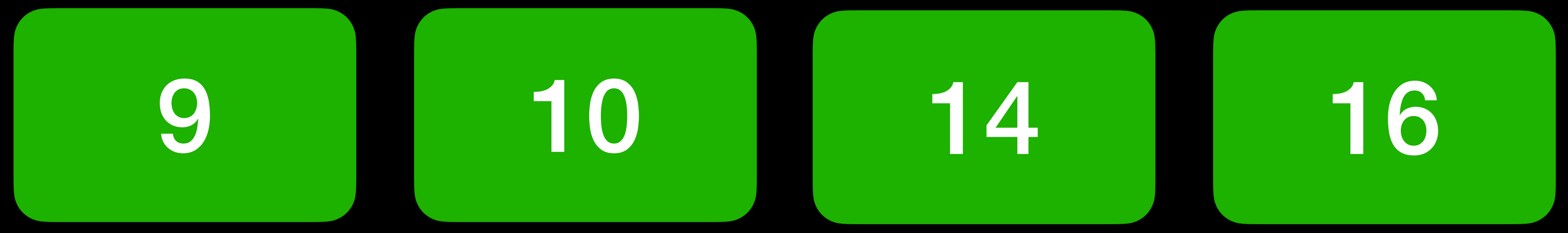


Remove largest element 移开最大的元素

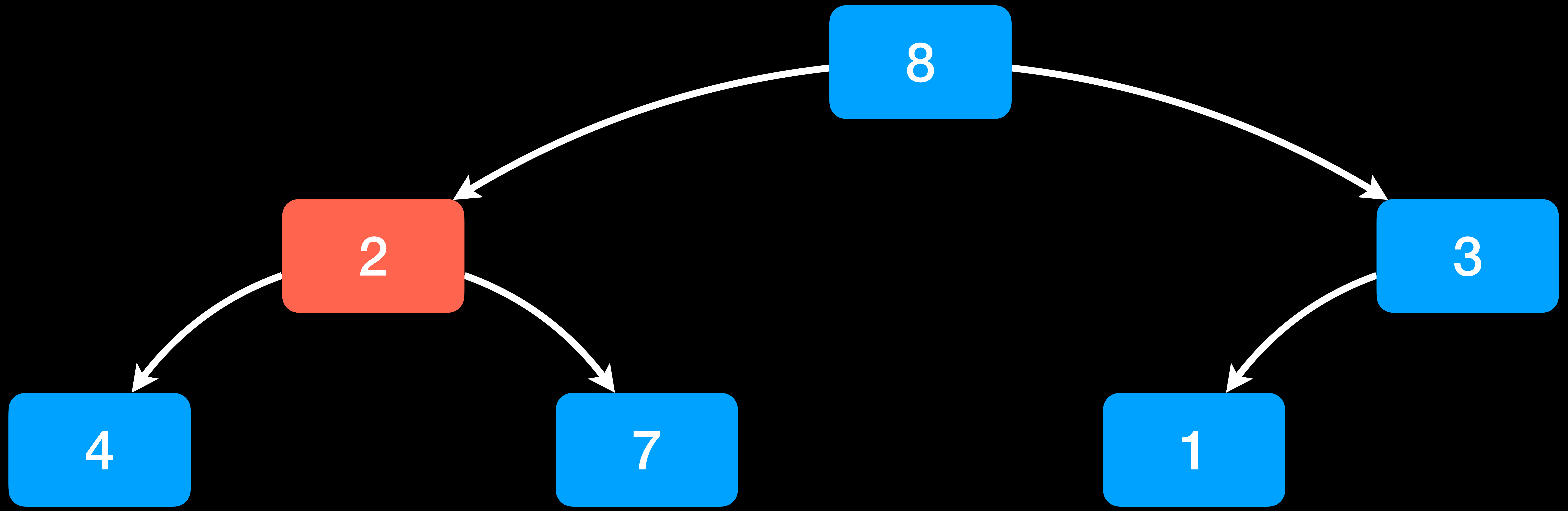


$A.heap\text{-}size = 6$

$A.length = 10$

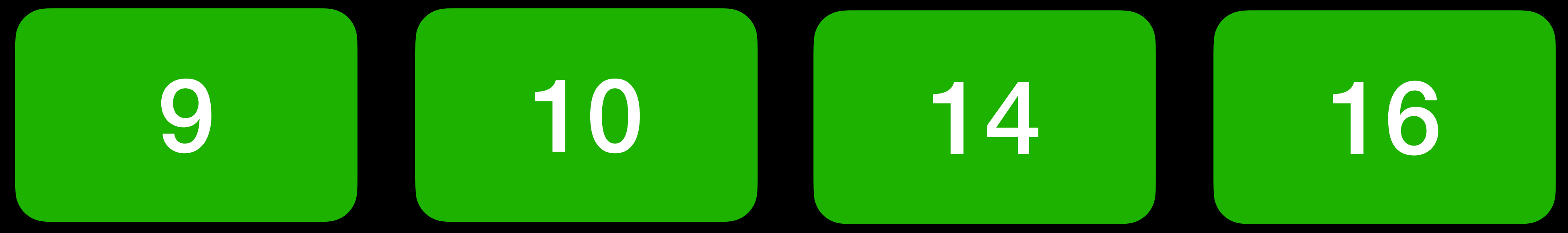


Correct error in heap 纠正堆内的错误

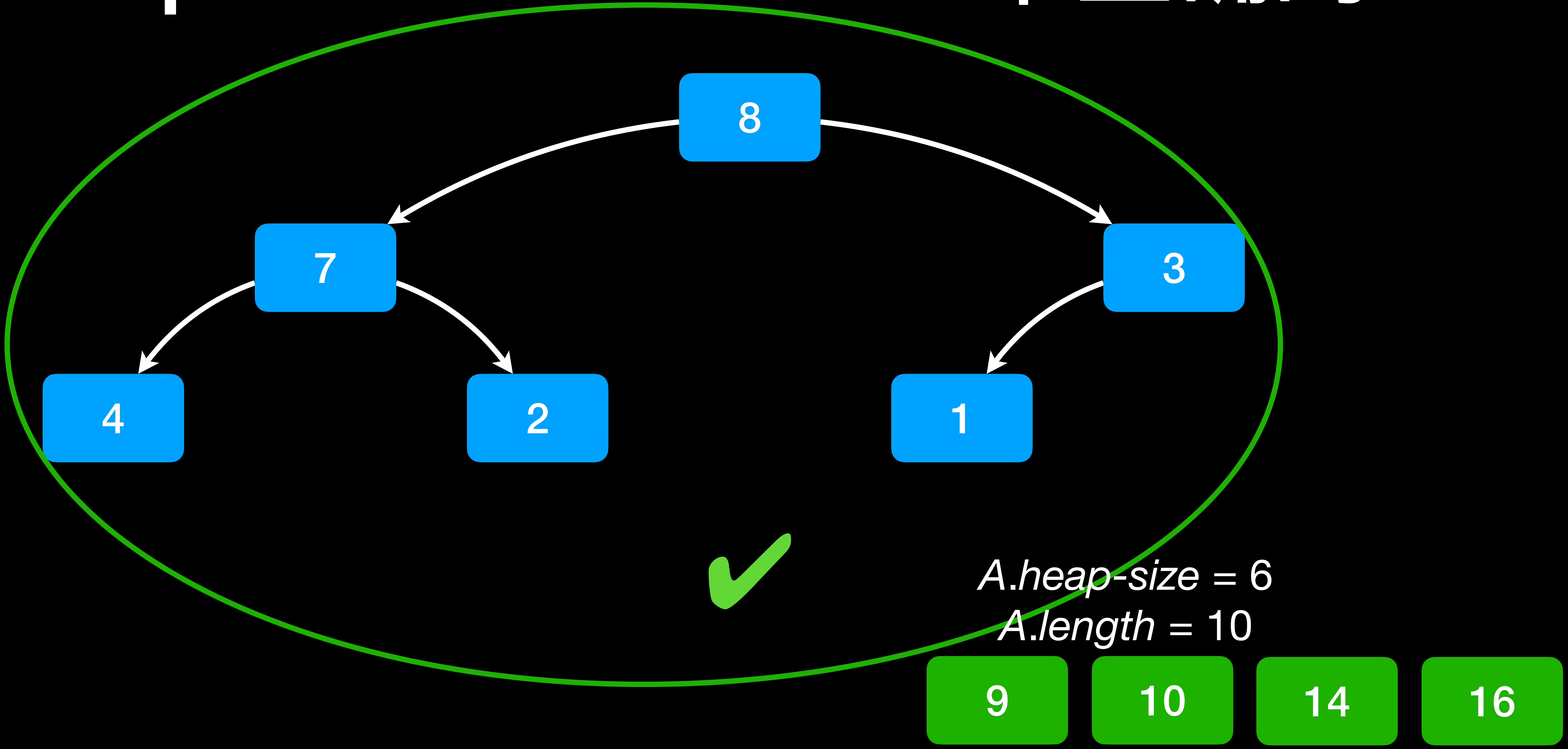


A.heap-size = 6

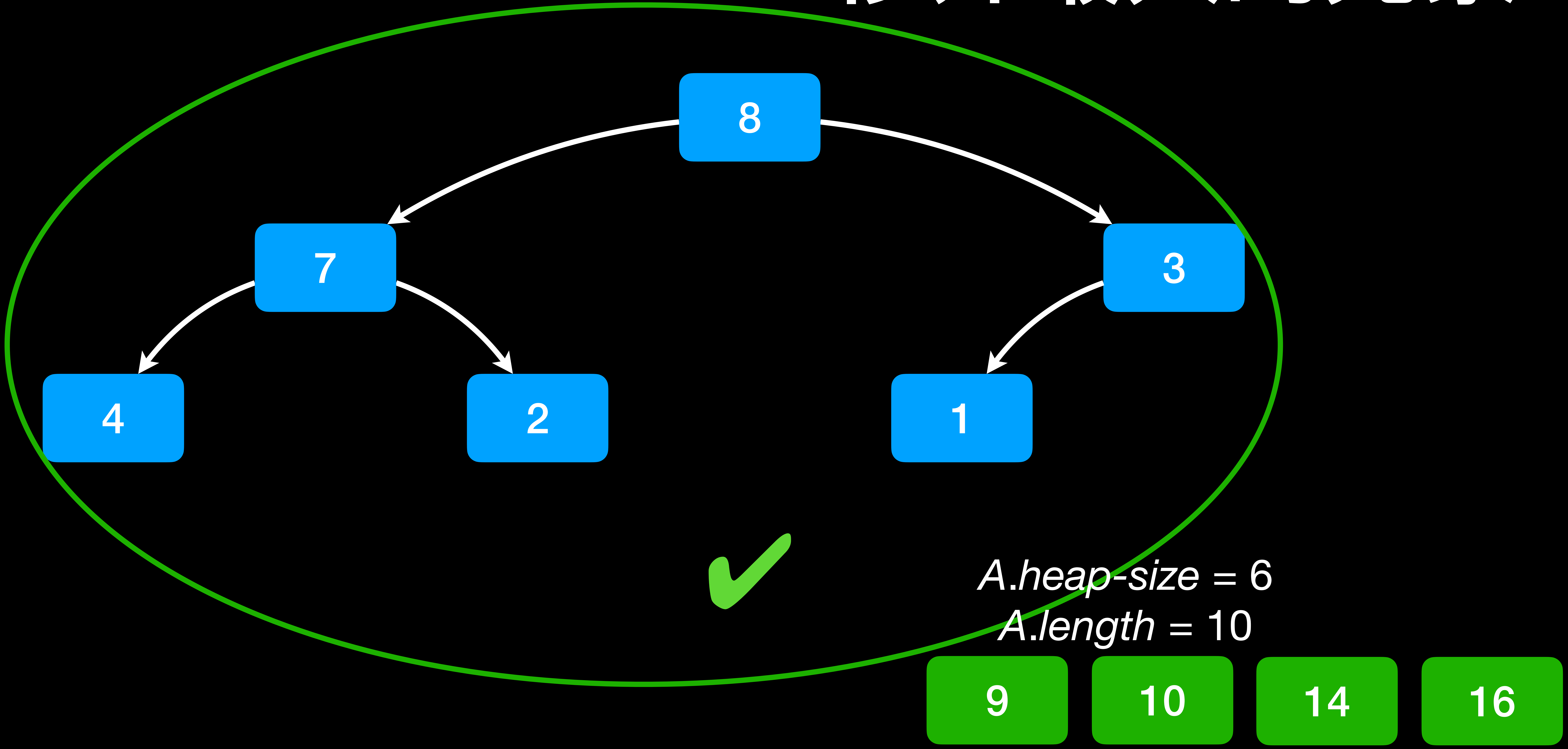
A.length = 10



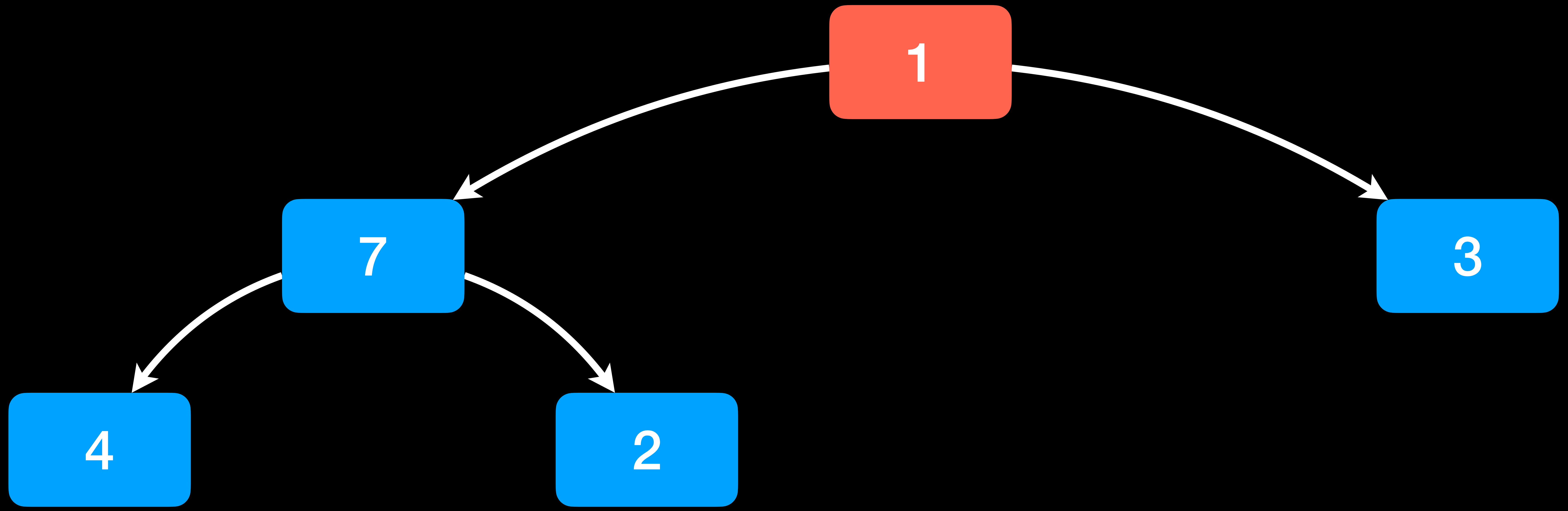
Heap is correct 堆正确的



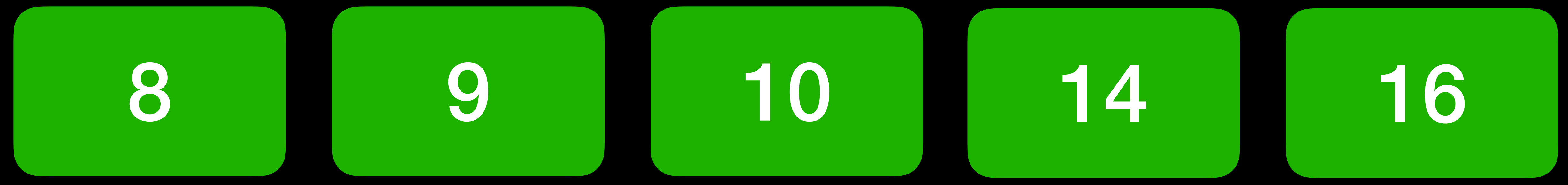
Remove largest element 移开最大的元素



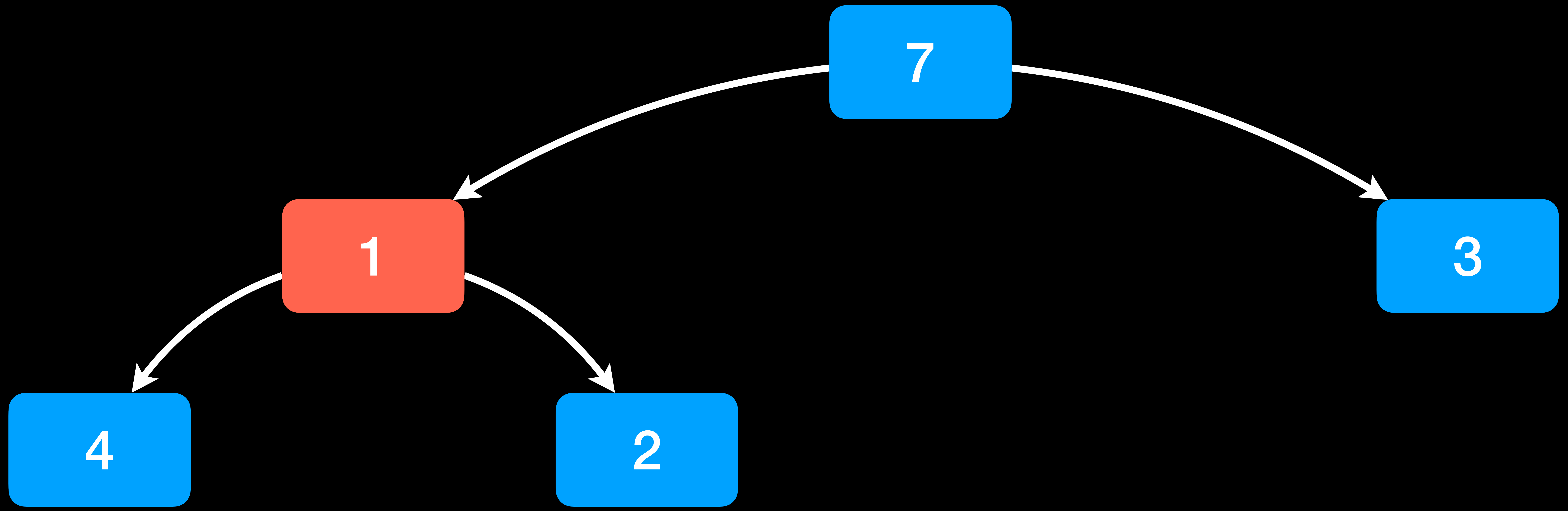
Remove largest element 移开最大的元素



A.heap-size = 5
A.length = 10

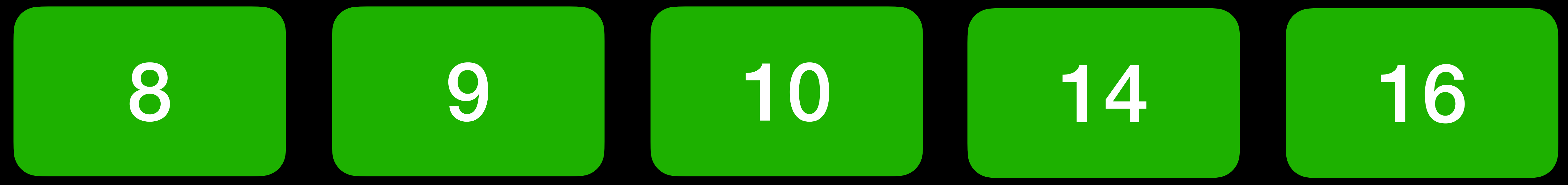


Correct error in heap 纠正堆内的错误

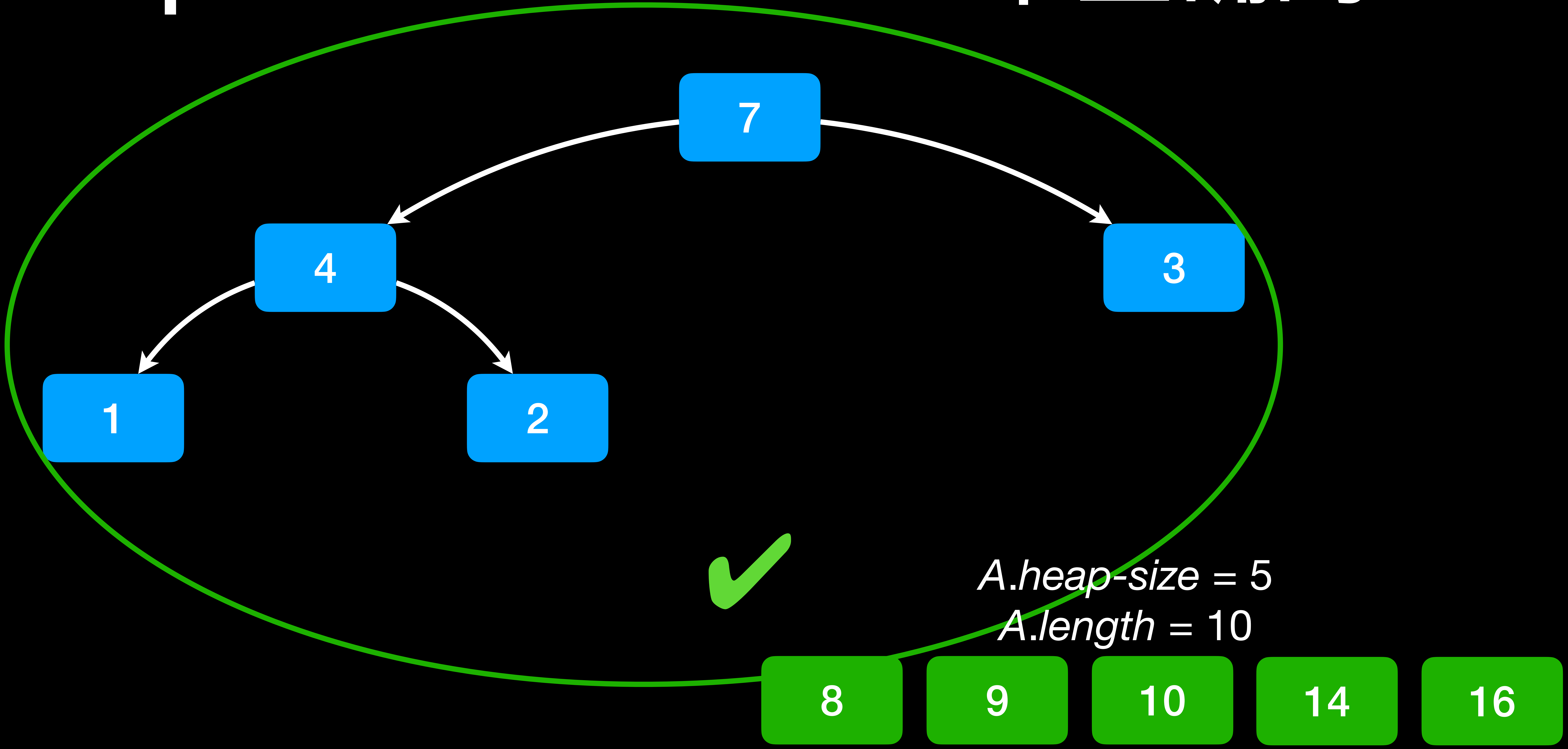


$A.heap\text{-}size = 5$

$A.length = 10$

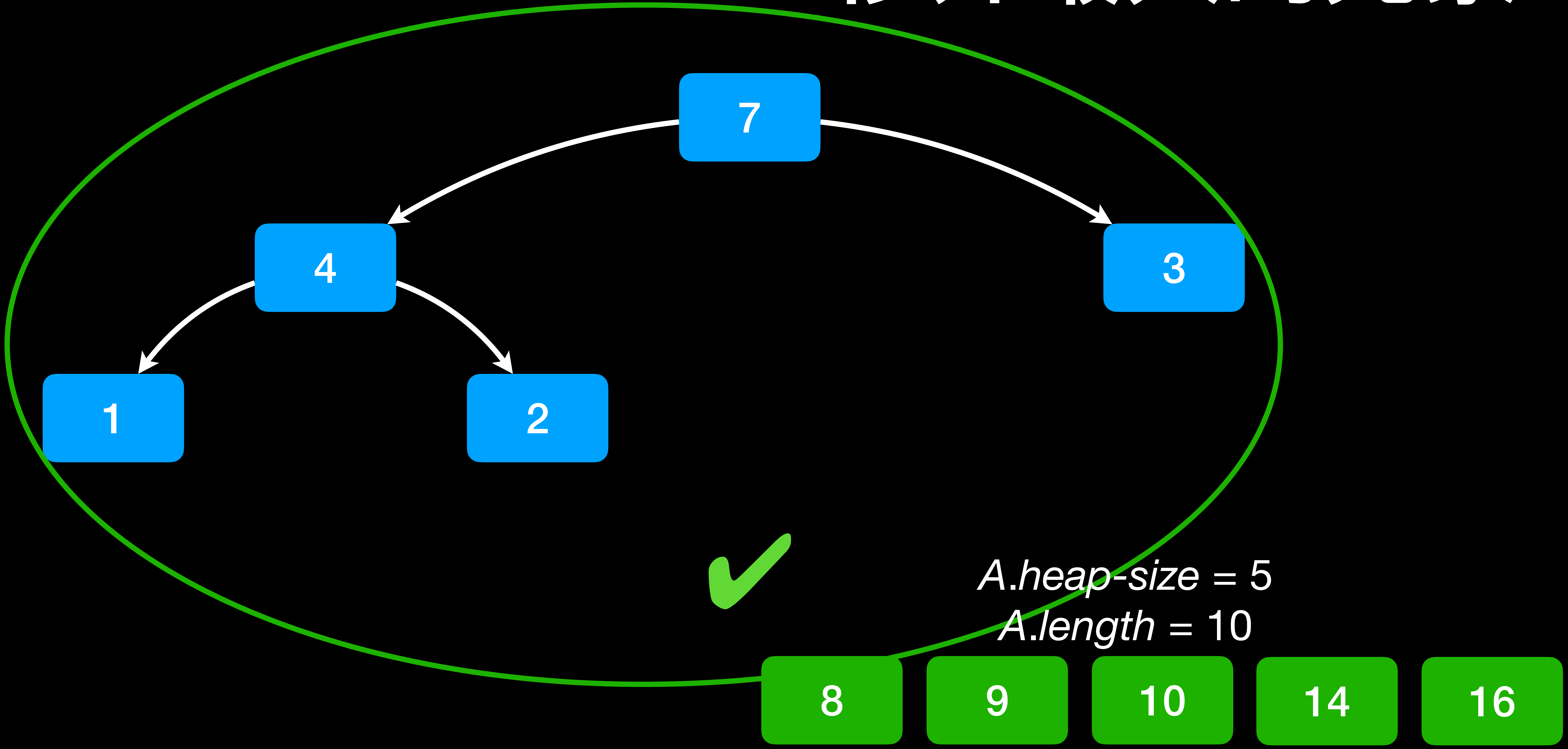


Heap is correct 堆正确的



A.heap-size = 5
A.length = 10

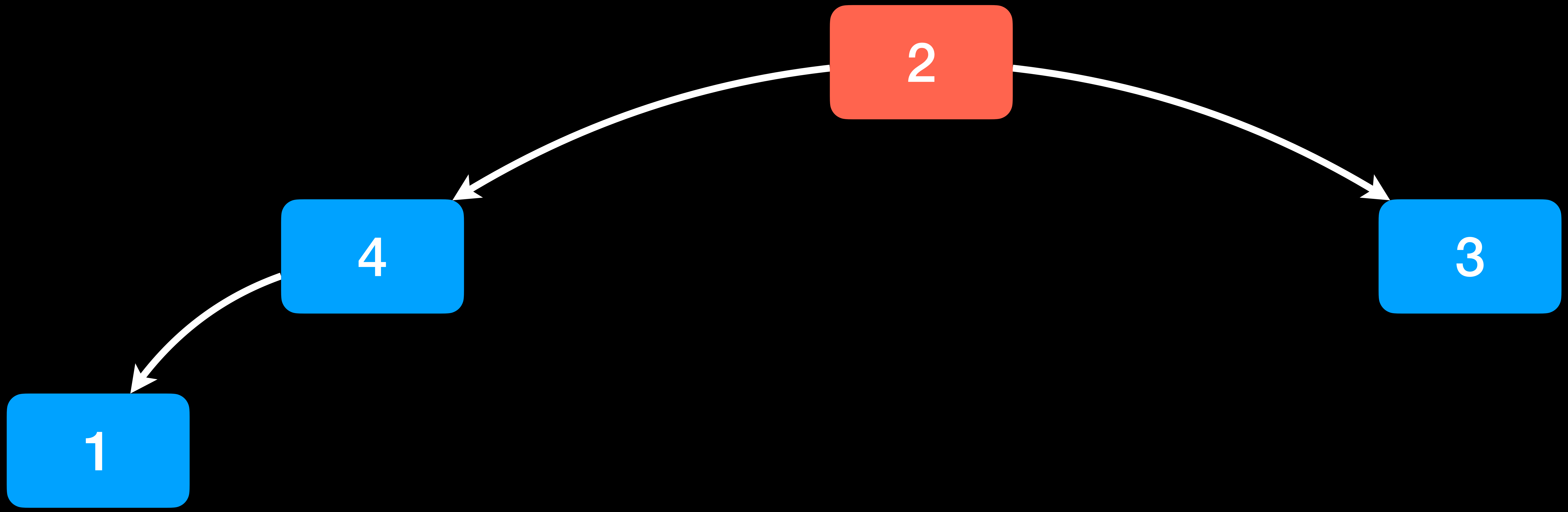
Remove largest element 移开最大的元素



A.heap-size = 5

A.length = 10

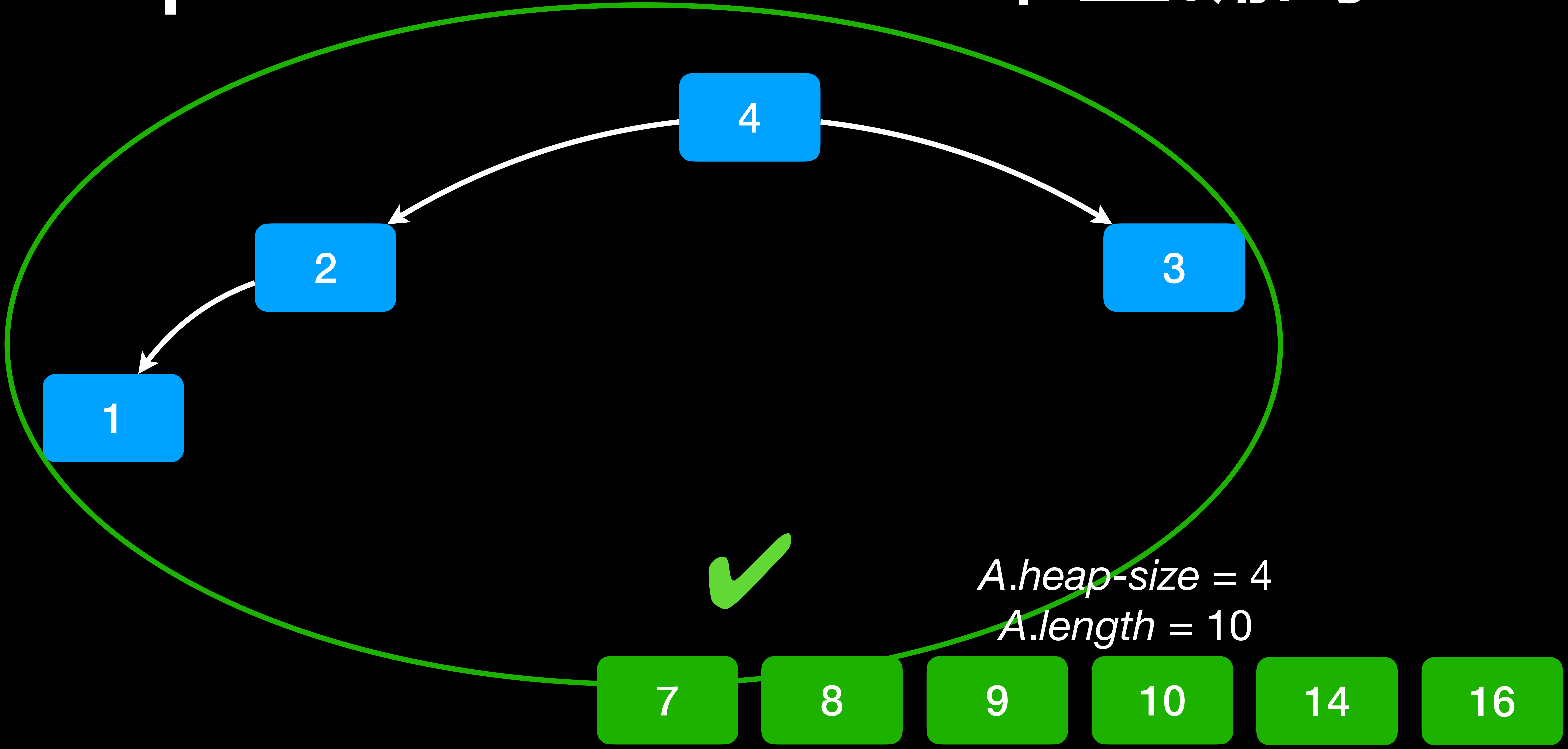
Remove largest element 移开最大的元素



$A.heap\text{-}size = 4$
$A.length = 10$

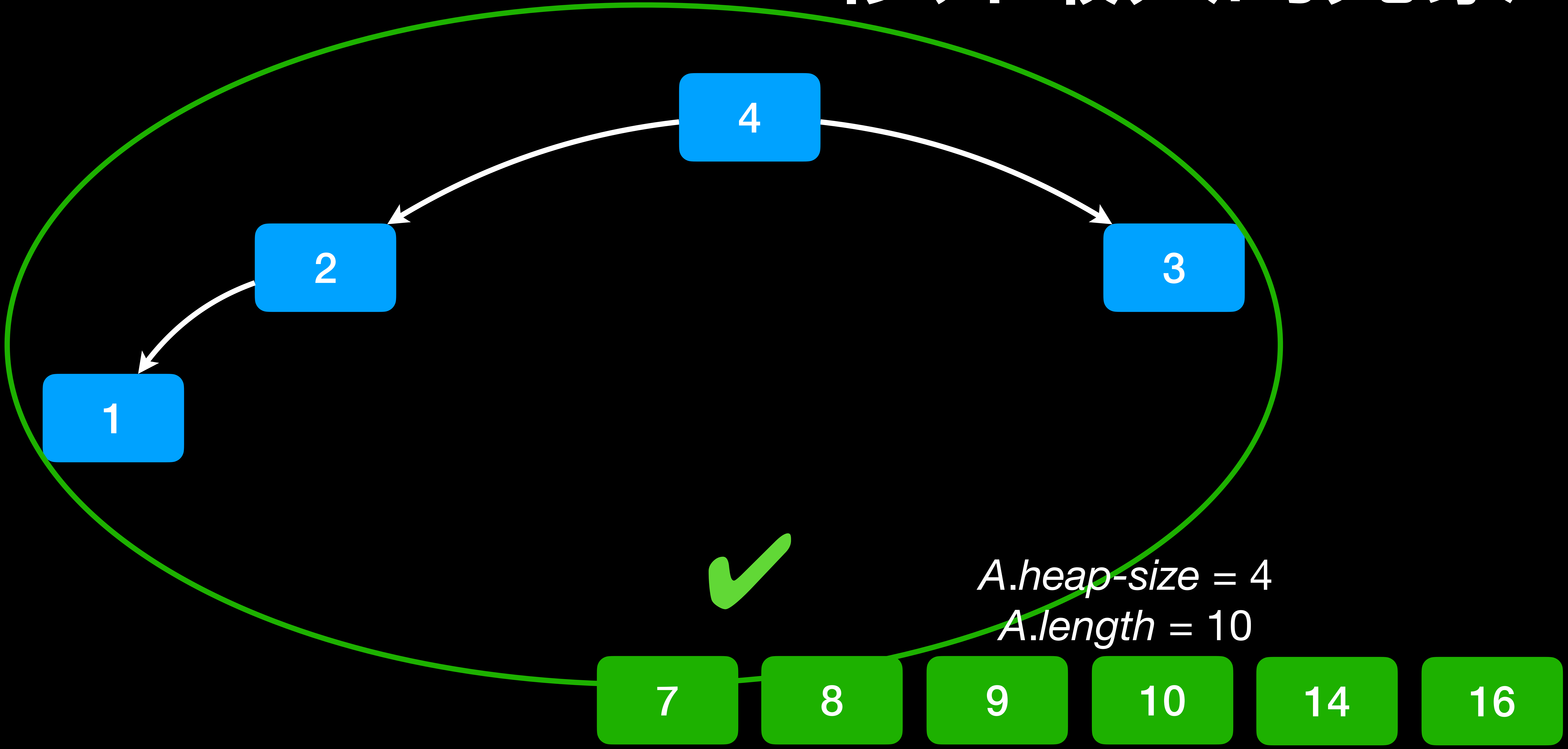
7	8	9	10	14	16

Heap is correct 堆正确的

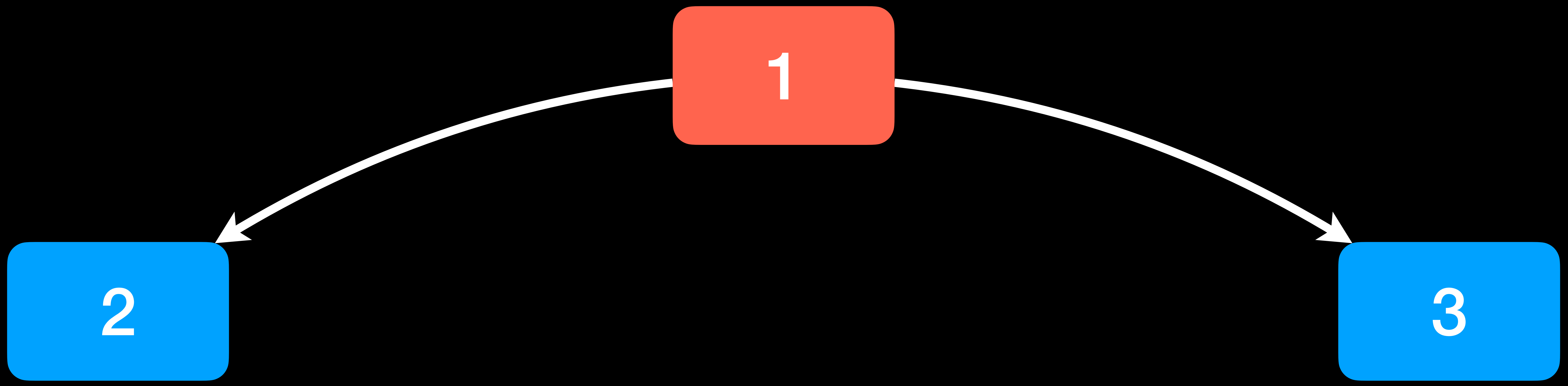


A.heap-size = 4
A.length = 10

Remove largest element 移开最大的元素



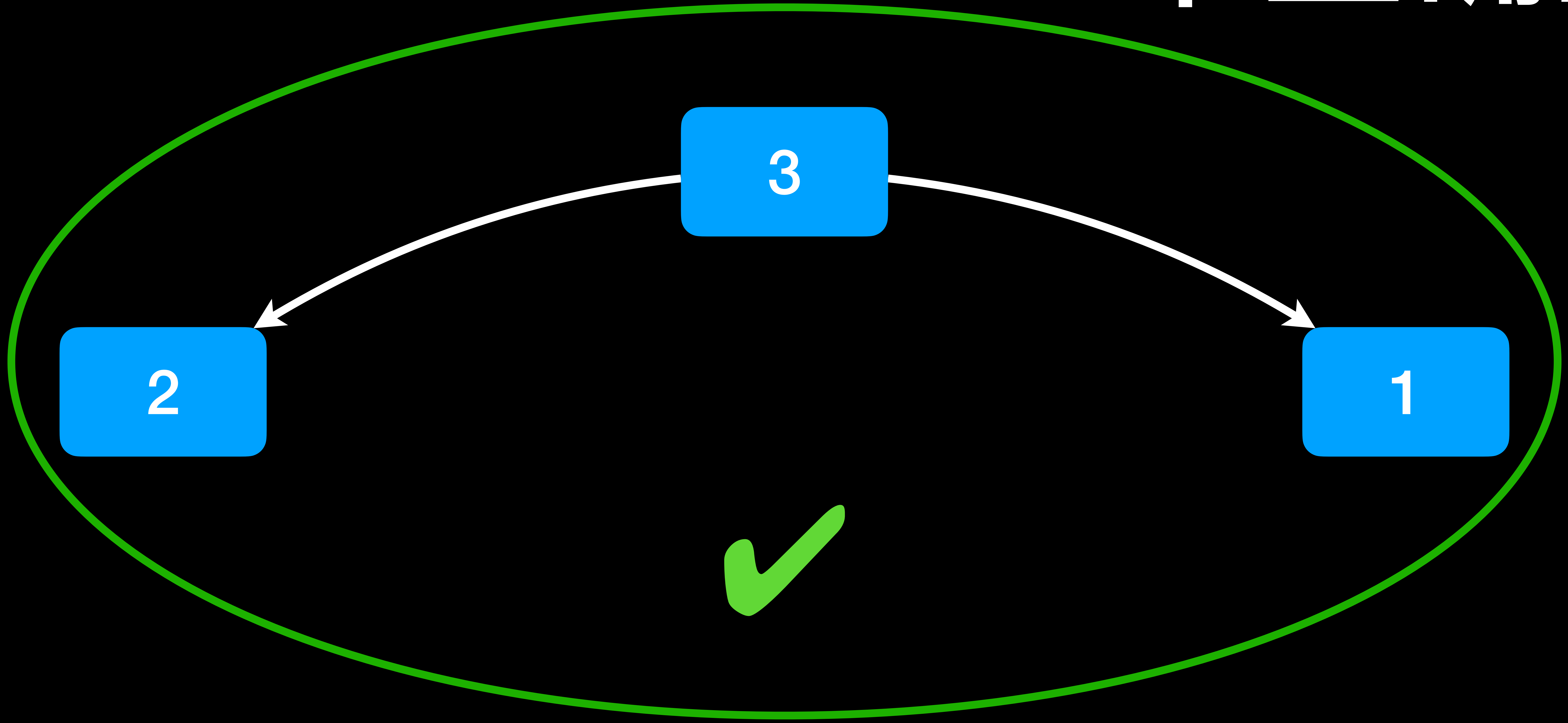
Remove largest element 移开最大的元素



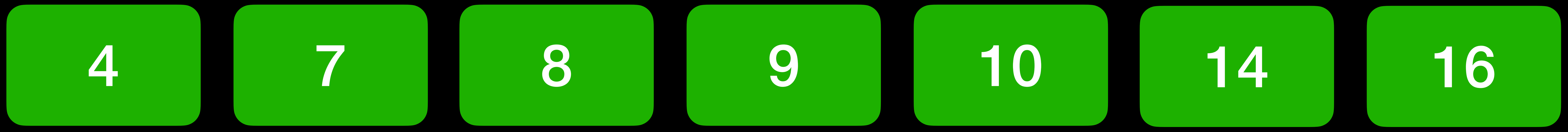
$A.heap\text{-}size = 3$
$A.length = 10$

4	7	8	9	10	14	16

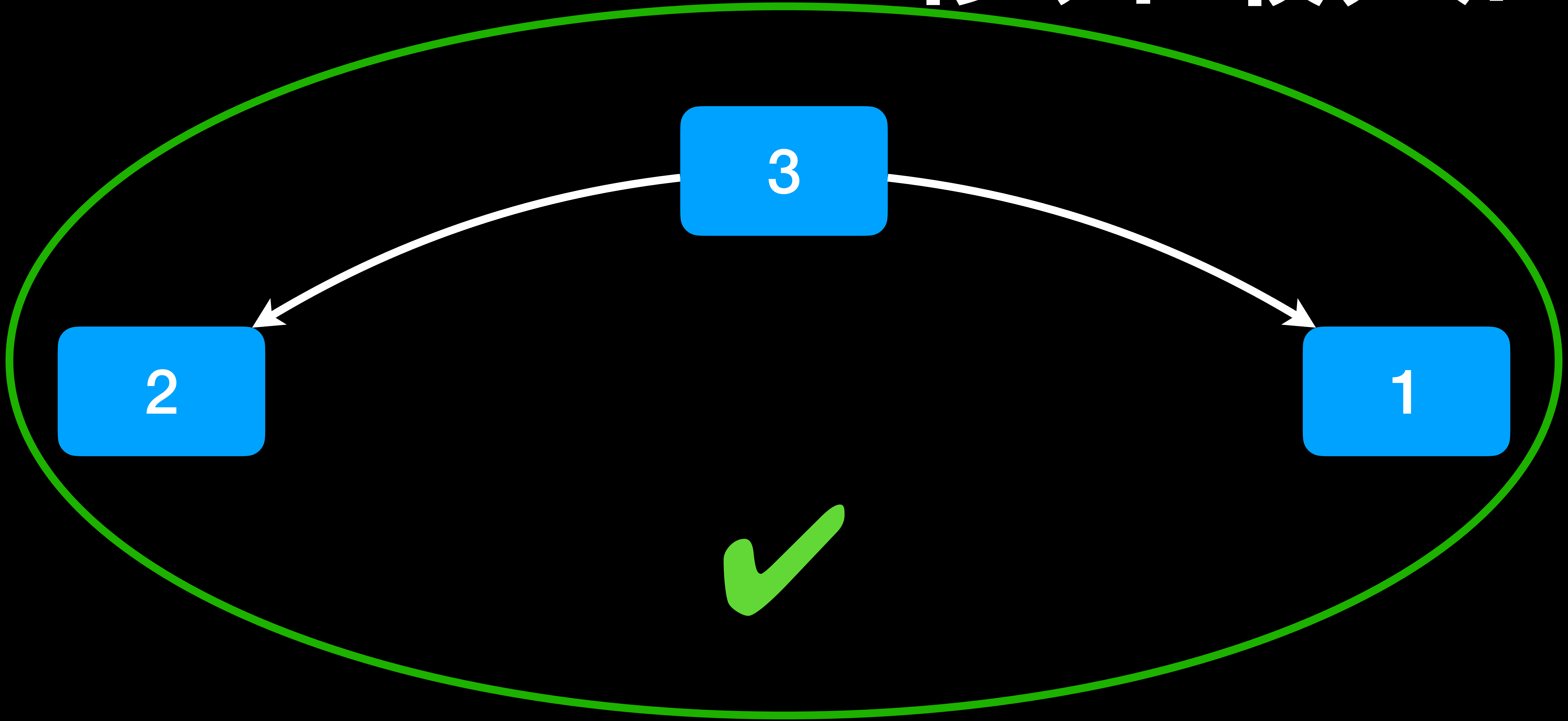
Heap is correct 堆正确的



$A.heap\text{-}size = 3$
$A.length = 10$



Remove largest element 移开最大的元素

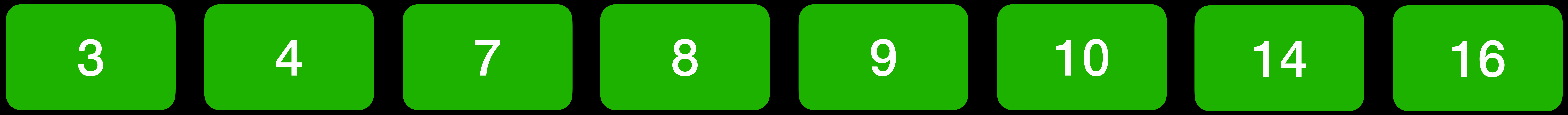


$A.heap\text{-}size = 3$
$A.length = 10$

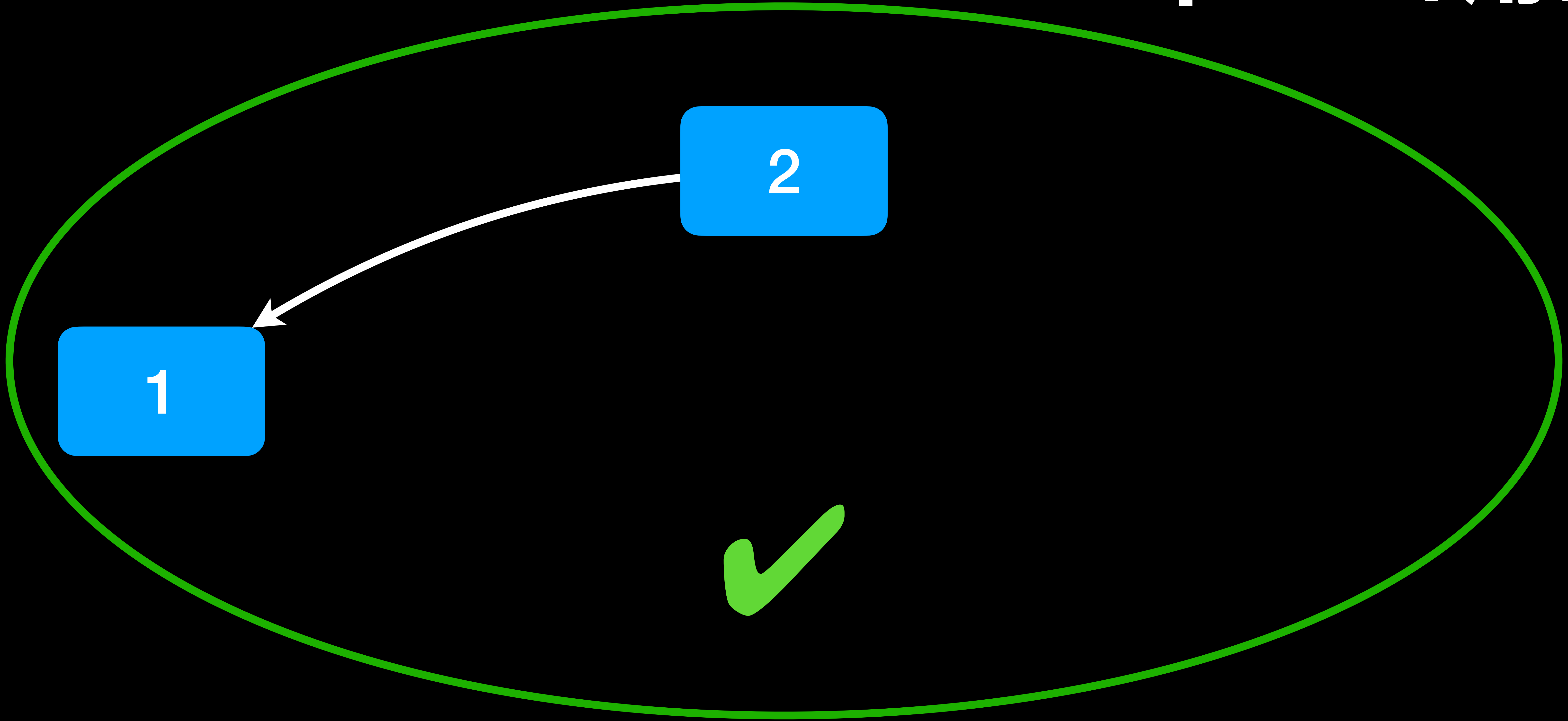
4	7	8	9	10	14	16

Remove largest element 移开最大的元素

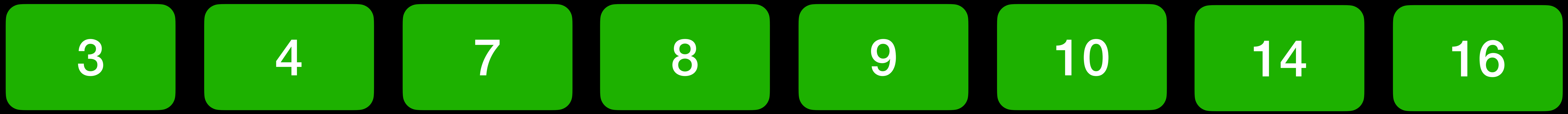
A.heap-size = 2
A.length = 10



Heap is correct 堆正确的



$A.heap\text{-}size = 2$
$A.length = 10$



Remove largest element 移开最大的元素

1

A.heap-size = 1
A.length = 10

2	3	4	7	8	9	10	14	16

Finished! 结束了！

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

$A.\mathit{length} = 10$

1	2	3	4	7	8	9	10	14	16

HEAPSORT

HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = $A.length$ **downto** 2
    Exchange A[1] with $A[i]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    MAX-HEAPIFY(A, 1)

$A[1], \ldots, A[i]$ is a correct max-heap.
$A[i+1], \ldots, A[A.length]$ contains the largest elements of A in order.
$i = A.heap\text{-}size$.

$A[1], \ldots, A[i]$ 是正确的最大堆。
$A[i+1], \ldots, A[A.length]$ 按顺序保存 A 的最大的元素。
$i = A.heap\text{-}size$。

Running Time 运行时间

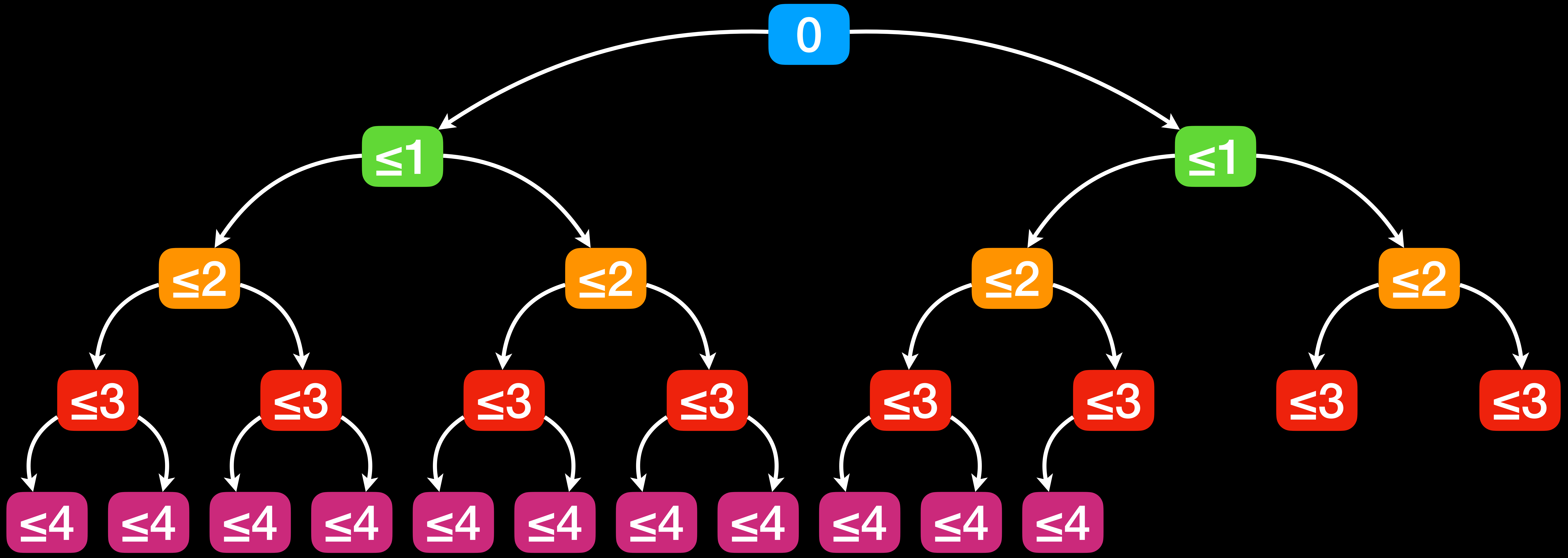
- n = *A.length*

- HEAPSORT first calls BUILD-MAX-HEAP, which takes time $O(n)$.

- Then it calls n–1 times MAX-HEAPIFY; these calls take time $O(\log(n-1) + \log(n-2) + ... + \log 1) = O(\log(n-1)!) = O(n \log n)$.

Running Time 运行时间

- The call to MAX-HEAPIFY after removing this node from the heap takes time ...:



Priority Queues 优先队列

Priority Queue 运行时间

- Examples:
  - scheduling of printer jobs
  - work with deadlines:
    need to handle urgent business
    before less urgent one
- Need queue where one job can overtake another
- priority: high priority = handle first

Priority Queue 优先队列

- use heap data structure to store priority queue
- need additional operations:
  - HEAP-EXTRACT-MAX: find the job with the highest priority and remove it from the queue.
  - HEAP-INCREASE-KEY: increase the priority of some waiting job
  - MAX-HEAP-INSERT: add a new job to the priority queue

Extract Maximum

- In a heap, the maximum is always the root ➨ easy to find
- To remove the maximum, one needs to replace it by some other entry (and correct the heap property)
➨ similar to one step in HEAPSORT

HEAP-EXTRACT-MAX

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
        error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size – 1
MAX-HEAPIFY(A, 1)
return max
```

Extract Maximum

- In a heap, the maximum is always the root ➨ easy to find
- To remove the maximum, one needs to replace it by some other entry (and correct the heap property)
  ➨ similar to one step in HEAPSORT
- Running time: $O(1 + \log n) = O(\log n)$

Increase Key

- Sometimes priorities change.
- A max-heap priority queue can increase the priority.
- Idea: if the new priority is too large (violates the max-heap property), exchange the node with its parent.
- (MAX-HEAPIFY can be used to decrease priority.)

HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, i, *key*)   // changes the priority of $A[i]$ to *key*

if *key* $< A[i]$

    error “new key is smaller than current key”

$A[i]$ = *key*

while $i > 1$ **and** $A[\text{PARENT}(i)] < A[i]$

    Exchange $A[i]$ with $A[\text{PARENT}(i)]$

    i = PARENT(i)

Increase Key

- Sometimes priorities change.
- A max-heap priority queue can increase the priority.
- Idea: if the new priority is too large (violates the max-heap property), exchange the node with its parent.
- Running time: $O(\log n)$ because the loop may go through all ancestors of $A[i]$.

Insert a new job

- Idea to insert a new job:
  First insert the job with priority $-\infty$,
  then increase priority to actual value
  using HEAP-INCREASE-KEY.

MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, key)   // adds a new entry with priority key

if $A.length \leq A.heap\text{-}size$

    error "The heap storage is full"

$A.heap\text{-}size = A.heap\text{-}size + 1$

$A[A.heap\text{-}size] = -\infty$

HEAP-INCREASE-KEY(A, $A.heap\text{-}size$, key)

Insert a new job

- Idea to insert a new job:
  First insert the job with priority $-\infty$,
  then increase priority to actual value
  using HEAP-INCREASE-KEY.

- Running time: only few operations in
  addition to HEAP-INCREASE-KEY,
  so the runtime is in $O(\log n)$.

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。

4.3-9

- 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。

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。

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

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.

急诊室队列

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。