Chapter 6
Heapsort

Chi-Yeh Chen
陳奇業
成功大學資訊工程學系

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

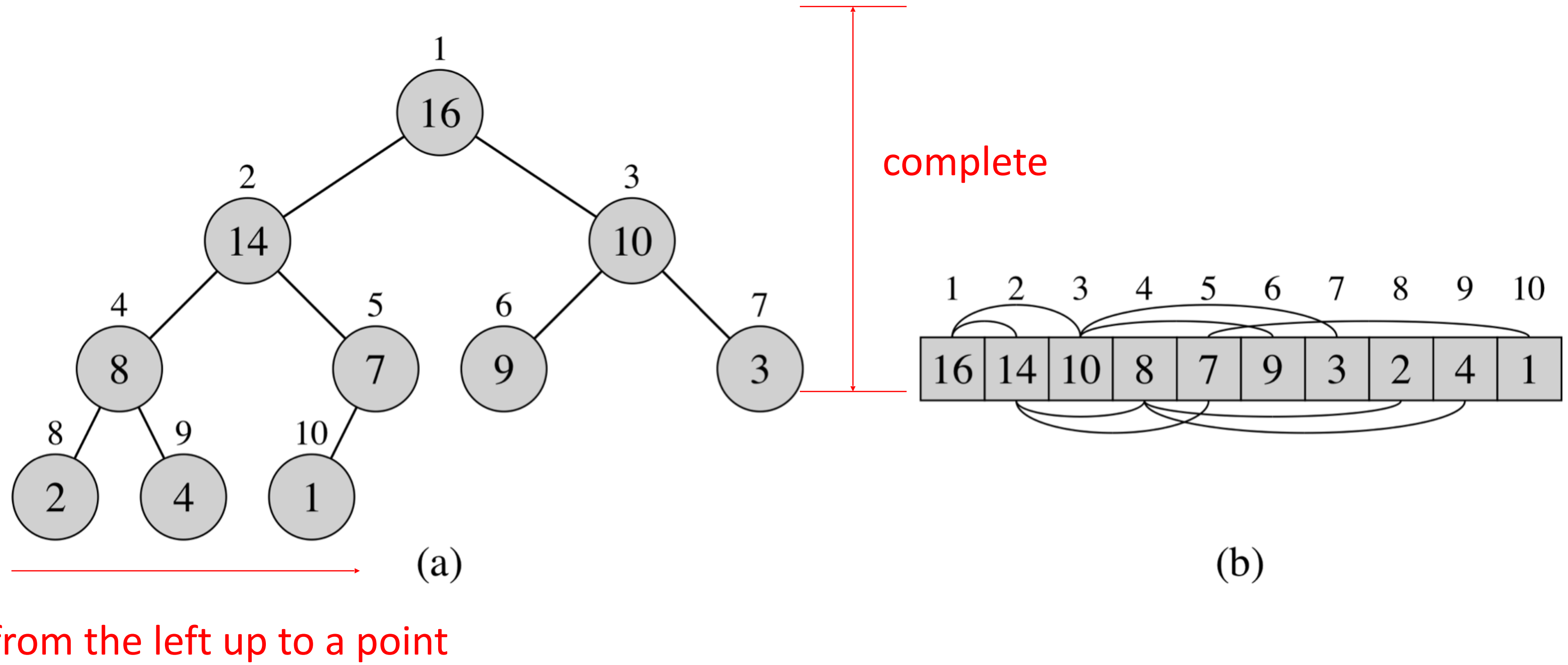
國立成功大學 九十週年
90th Anniversary of NCKU

Overview

- Like merge sort, but unlike insertion sort, heapsort's running time is $O(n\lg n)$.

- Like insertion sort, but unlike merge sort, heapsort sorts in place.

Heap



1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1
complete
1 2 3 4 5 6 7 8 9 10
16 14 10 8 7 9 3 2 4 1
from the left up to a point

(a) (b)

Heap

- Heap A is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\Theta(\lg n)$

Heap

- $A.length$, which gives the number of elements in the array
- $A.heap\_size$, which represents how many elements in the heap are stored within array A.
- Although $A[1 \ldots A.length]$ may contain numbers, only the elements in $A[1 \ldots A.heap\_size]$

Heap

- A heap can be stores as an array A
  - Root of trees is $A\left[1\right]$
  - Parent of $A[i] = A\left[\lfloor i/2 \rfloor\right]$
  - Left child of $A[i] = A\left[2i\right]$
  - Right child of$A[i] = A\left[2i + 1\right]$
  - Computing is fast with binary representation implementation

Heap

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

藏行顯光 成就共好 Achieve Securely Prosper Mutually

Heap property

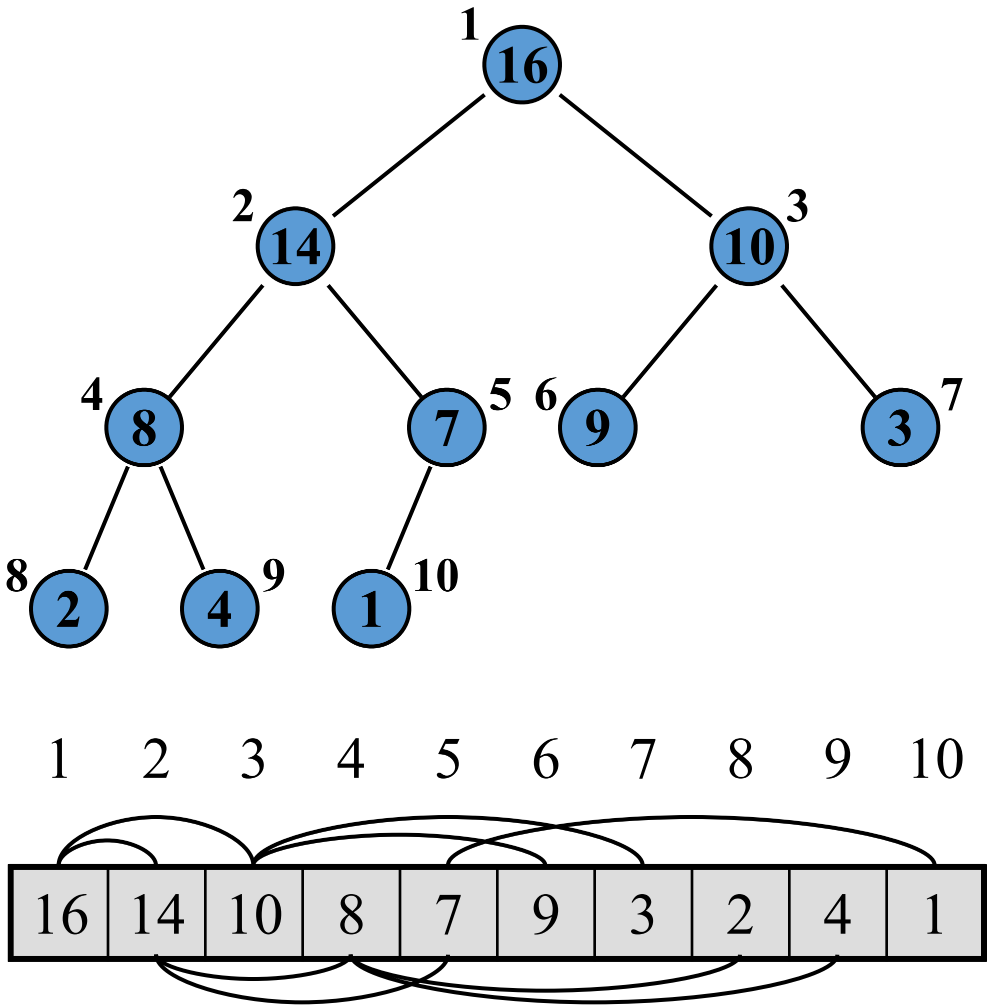
- Heap property
  - For max-heap (largest element at root), *max-heap property*: for all nodes i, excluding the root, $A\left[PARENT(i)\right] \geq A[i]$.
  - For min-heap (smallest element at root), *min-heap property*: for all nodes i, excluding the root, $A\left[PARENT(i)\right] \leq A[i]$.

- Maximum element of a max-heap is at the root.

- The heapsort algorithm we'll use max-heaps.

Example

- A max-heap



Heap property

- The basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time.
  - MAX-HEAPIFY: $O(\lg n)$
  - BUILD-MAX-HEAP: run in linear time $O(n)$.
  - HEAPSORT: $O(n \lg n)$
  - MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY and HEAP-MAXIMUM: $O(\lg n)$

- A heap with height h is an almost-complete binary tree (complete at all levels except possibly the lowest)
  - at most $2^{h+1} - 1$ elements (if it is complete)
  - at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

- Given an n-element heap of height h, we know that $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$. Thus, $h \leq \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor$.

Maintaining the heap property

- MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.
  - Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
  - Assume left and right subtrees of i are max-heaps.
  - After MAX-HEAPIFY, subtree rooted at i is a max-heap

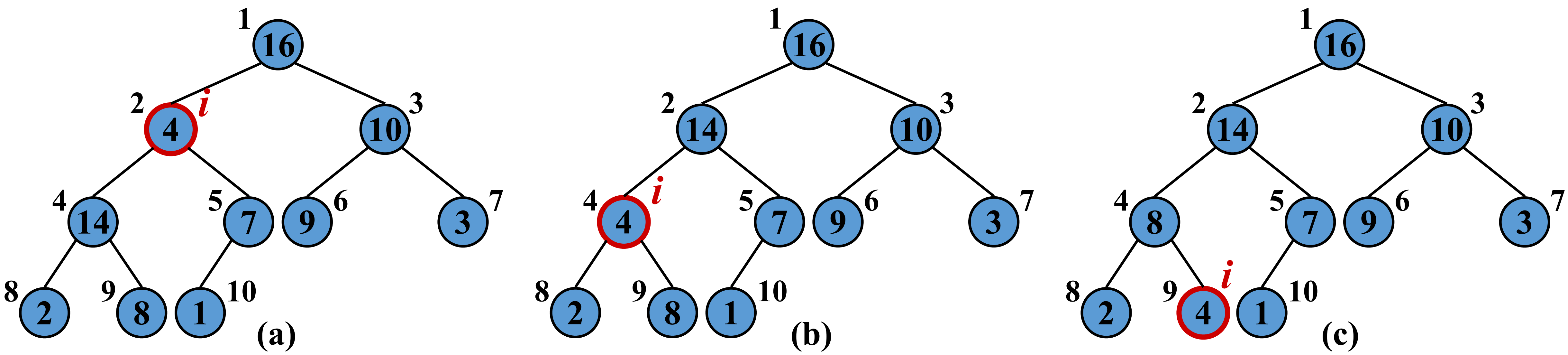
Pseudocode

MAXHEAPIFY(A, i, n)

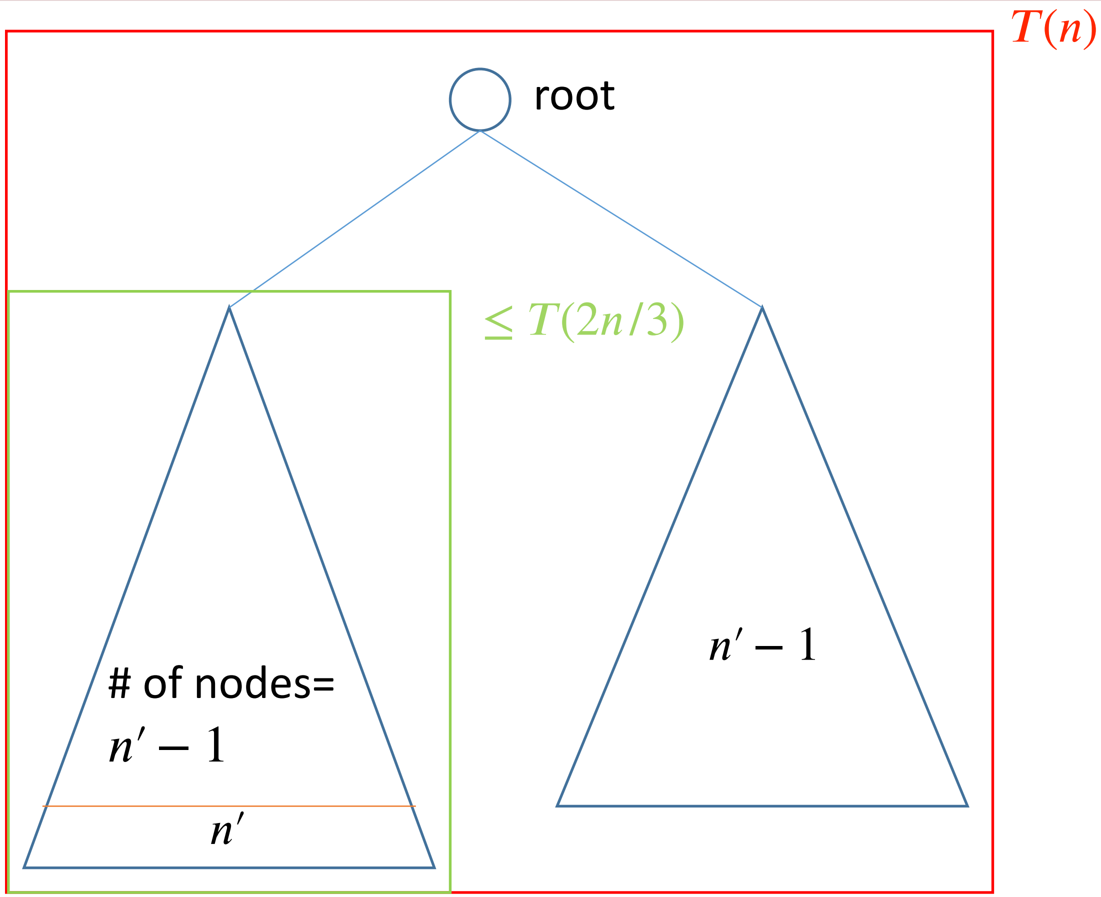
```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i] then
    largest ← l
else
    largest ← i
if r ≤ n and A[r] > A[largest] then
    largest ← r
if largest ≠ i then
    exchange A[i] ↔ A[largest]
    MAXHEAPIFY(A, largest, n)
```

Example

- Run MAX-HEAPIFY on the following heap example.



Analysis of MAX-HEAPIFY



The children's subtrees each have size at most $2n/3$

Analysis of MAX-HEAPIFY

- $T(n) \le T\left(\frac{2n}{3}\right) + \Theta(1)$
  By case 2 of the master theorem, the solution to this recurrence is $T(n) = O(\lg n)$.

- Case 2: If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

Analysis of MAX-HEAPIFY

- Time: $\mathrm{O}(\lg n)$

- Correctness: Heap is almost-complete binary tree, hence must process $\mathrm{O}(\lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

- The following procedure, given an unordered array, will produce a max-heap.

BUILD-MAX-HEAP(A, n)

---

for $i = \lfloor n/2 \rfloor$ ***downto*** 1 **do**
    MAX-HEAPIFY(A, i, n)

- $\lfloor n/2 \rfloor+1, \lfloor n/2 \rfloor+2, \ldots, n$.

    Assume $A[\lfloor n/2 \rfloor+1]$ is not a leaf, then LEFT($\lfloor n/2 \rfloor+1$)$\leq n$

    LEFT($\lfloor n/2 \rfloor+1$)$=2(\lfloor n/2 \rfloor+1)>2(n/2-1+1)=n$ →←

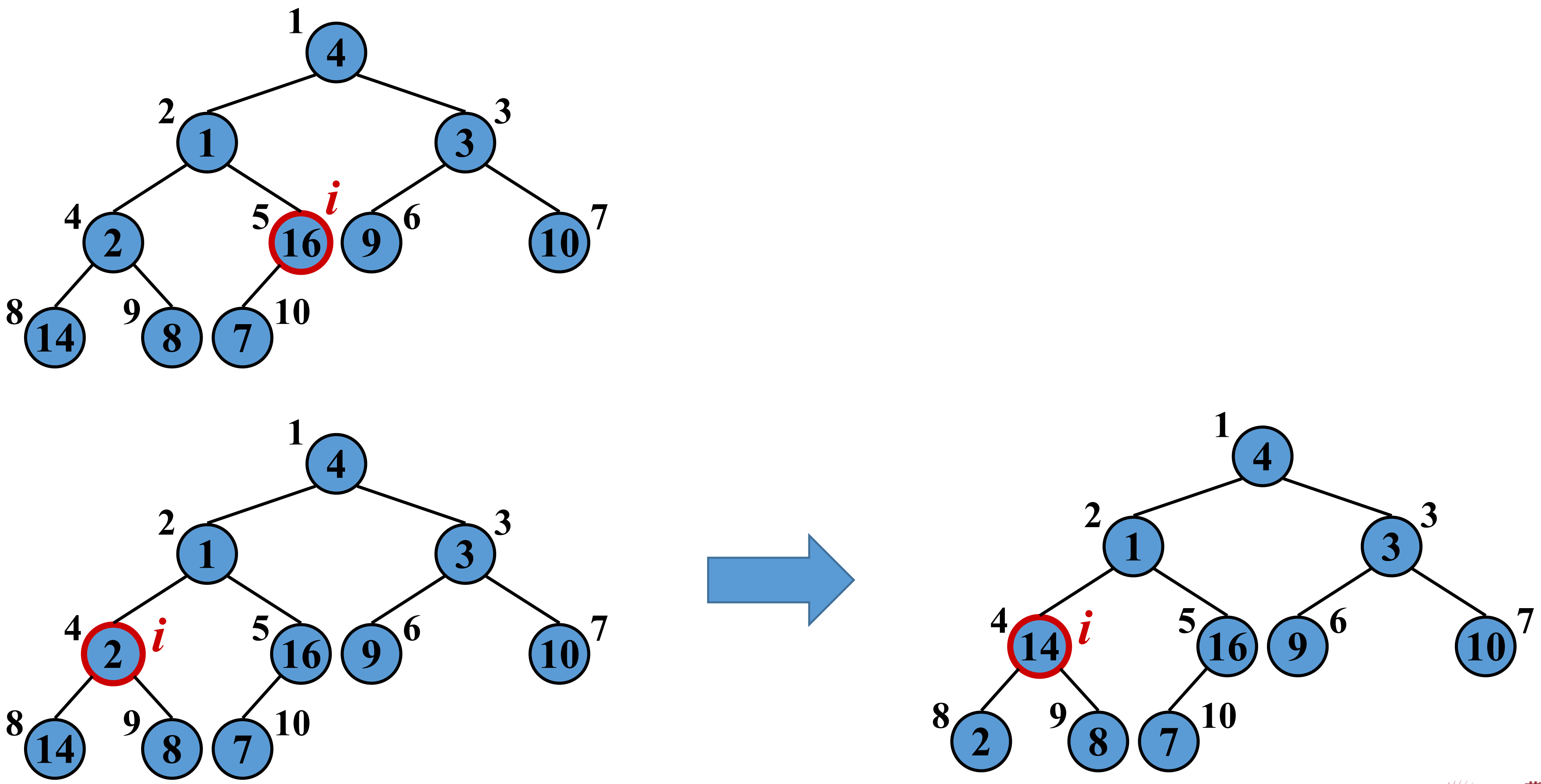
Example

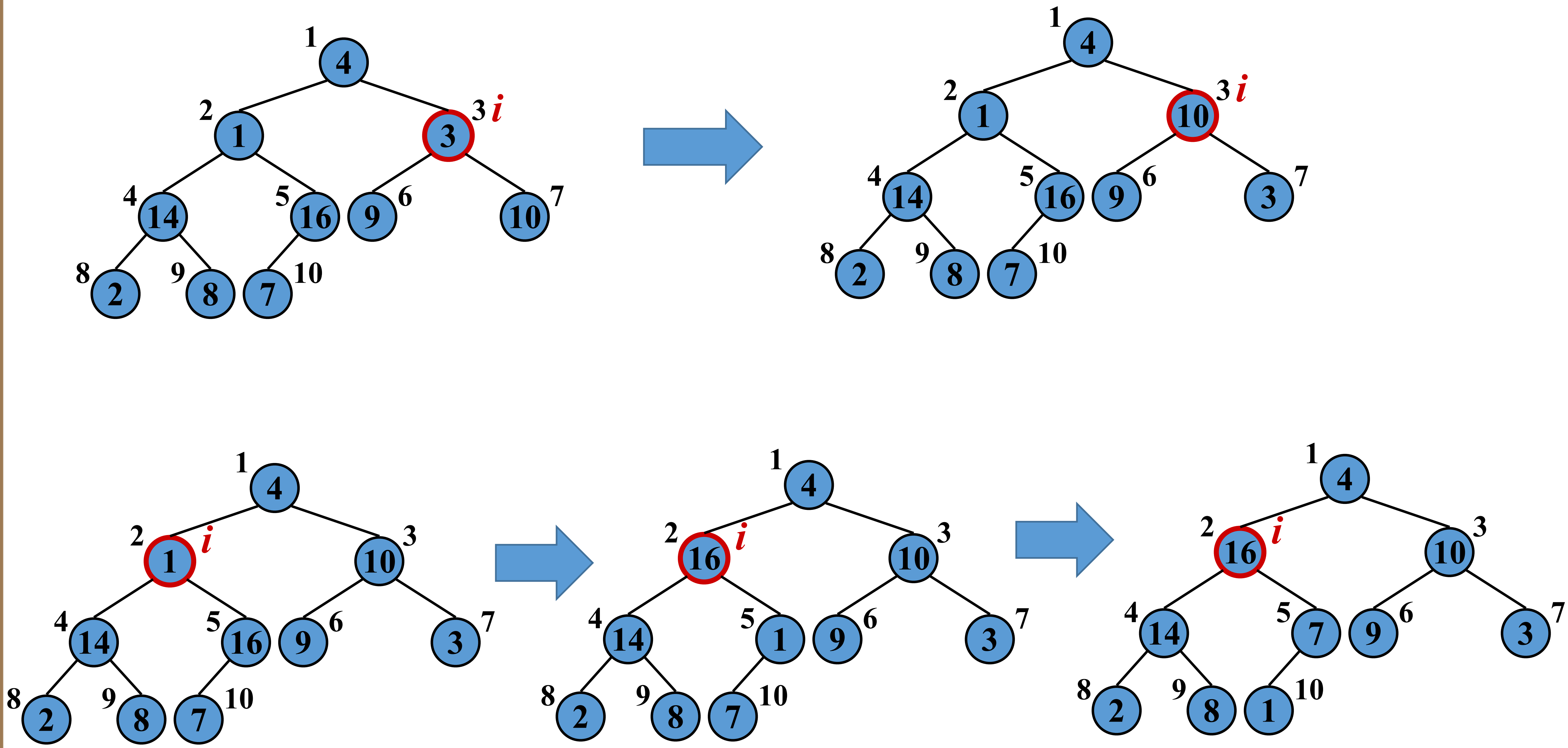
- Building a max-heap from the following unsorted array results in the first heap example.
  - i starts off as 5.
  - MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

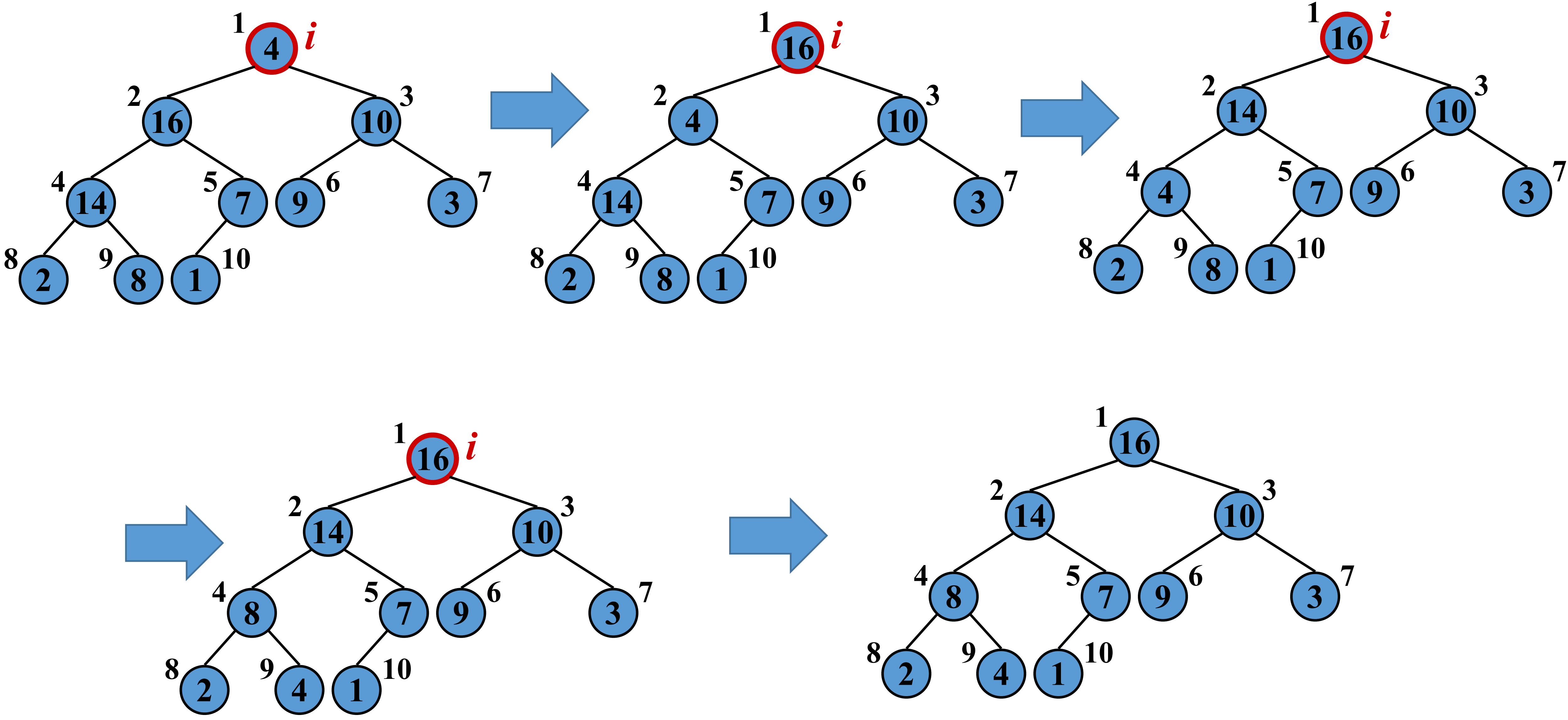
	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

藏行顯光 成就
共好 國立成功大學 九十週年
90th Anniversary of NCKU
Achieve Securely · Prosper Mutually

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7







Correctness

- A loop invariant is a property of a program loop that is true before (and after) each iteration.
- We must show three things about a loop invariant:
  - Initialization: It is true prior to the first iteration of the loop.
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Correctness

- Loop invariant: At start of every iteration of for loop, each node $i + 1, i + 2, \ldots, n$ is root of a max-heap.
  - Initialization: We know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the for loop, the invariant is initially true.
  - Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i + 1, i + 2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.
  - Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

- Be careful not to confuse the height of a node (longest distance from a leaf) with its depth (distance from the root).
- If the heap is not a complete binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same height. For example, although all nodes at depth H have height 0, nodes at depth $H-1$ can have either height 0 or height 1.

Analysis

- There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.
  - For a complete binary tree, it's easy to show that there are $\lceil n/2^{h+1} \rceil$ nodes of height h.
  - But the proof for an incomplete tree is tricky and is not derived from the proof for a complete tree.

藏行顯光 成就共好 國立成功大學九十週年 90th Anniversary of NCKU
Achieve Securely · Prosper Mutually

Proof. By induction on h. Let H be the height of the heap.

Basis: Show that it's true for $h = 0$ (i.e., that # of leaves $\leq \left\lceil n/2^{h+1} \right\rceil = \lceil n/2 \rceil$). In fact, we'll show that the # of leaves $= \lceil n/2 \rceil$.

The tree leaves (nodes at height 0) are at depths H and $H - 1$. They consist of

- all nodes at depth H, and
- the nodes at depth $H - 1$ that are not parents of depth-H nodes.

藏行顯光 成就共好 Achieve Securely Prosper Mutually

Let x be the number of nodes at depth H—that is, the number of nodes in the bottom (possibly incomplete) level.

Note that $n - x$ is odd, because the $n - x$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes. Thus if n is odd, x is even, and if n is even, x is odd.

- If n is odd, then x is even, so all nodes have siblings—i.e., all internal nodes have 2 children. Thus (see Exercise B.5-3), # of internal nodes = # of leaves-1.
  So, n= # of nodes = # of leaves+ # of internal nodes = 2 # of leaves-1.
  Thus, # of leaves = $\frac{(n+1)}{2} \leq \left\lceil \frac{n}{2} \right\rceil$. (The latter equality holds because n is odd.)

- If n is even, then x is odd, and some leaf doesn't have a sibling. If we gave it a sibling, we would have $n+1$ nodes, where $n+1$ is odd, so the case we analyzed above would apply. Observe that we would also increase the number of leaves by 1, since we added a node to a parent that already had a child. By the odd-node case above, # of leaves+1= $\frac{(n+1)}{2} \leq \left\lceil \frac{n}{2} \right\rceil + 1$. (The latter equality holds because n is even.)

- In either case, # of leaves $\leq \left\lceil \frac{n}{2} \right\rceil$.

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

Inductive step: Show that if it's true for height $h-1$, it's true for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \left\lceil \frac{n}{2} \right\rceil$, so

$$n' = n - n_0 \leq n - \left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor.$$

- Note that the nodes at height h in T would be at height $h-1$ in T'. Letting n'_{h-1} denote the number of nodes at height $h-1$ in T', we have
  $n_h = n'_{h-1}$

  By induction, we can bound n'_{h-1}:

$$n_h = n'_{h-1} \le \left\lceil \frac{n'}{2^h} \right\rceil \le \left\lceil \frac{\left\lfloor \frac{n}{2} \right\rfloor}{2^h} \right\rceil \le \left\lceil \frac{\frac{n}{2}}{2^h} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

$\blacksquare$

藏行顯光
成就共好
Achieve Securely
Prosper Mutually

Analysis

- Analysis of BUILD-MAX-HEAP
  - Simple bound: $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\rightarrow O(n \lg n)$.
  - Tighter analysis: Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h, and height of heap is $\lfloor \lg n \rfloor$.
    - The Time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is
      $$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$
    - Evaluate the last summation by substituting $x = 1/2$ in the formula $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$ for $|x| < 1$.
    - Thus, the running time of BUILD-MAX-HEAP is $O(n)$.
      $$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

藏行顯光 成就共好 國立成功大學 九十週年 90th Anniversary of NCKU Achieve Securely · Prosper Mutually

Building a min-heap

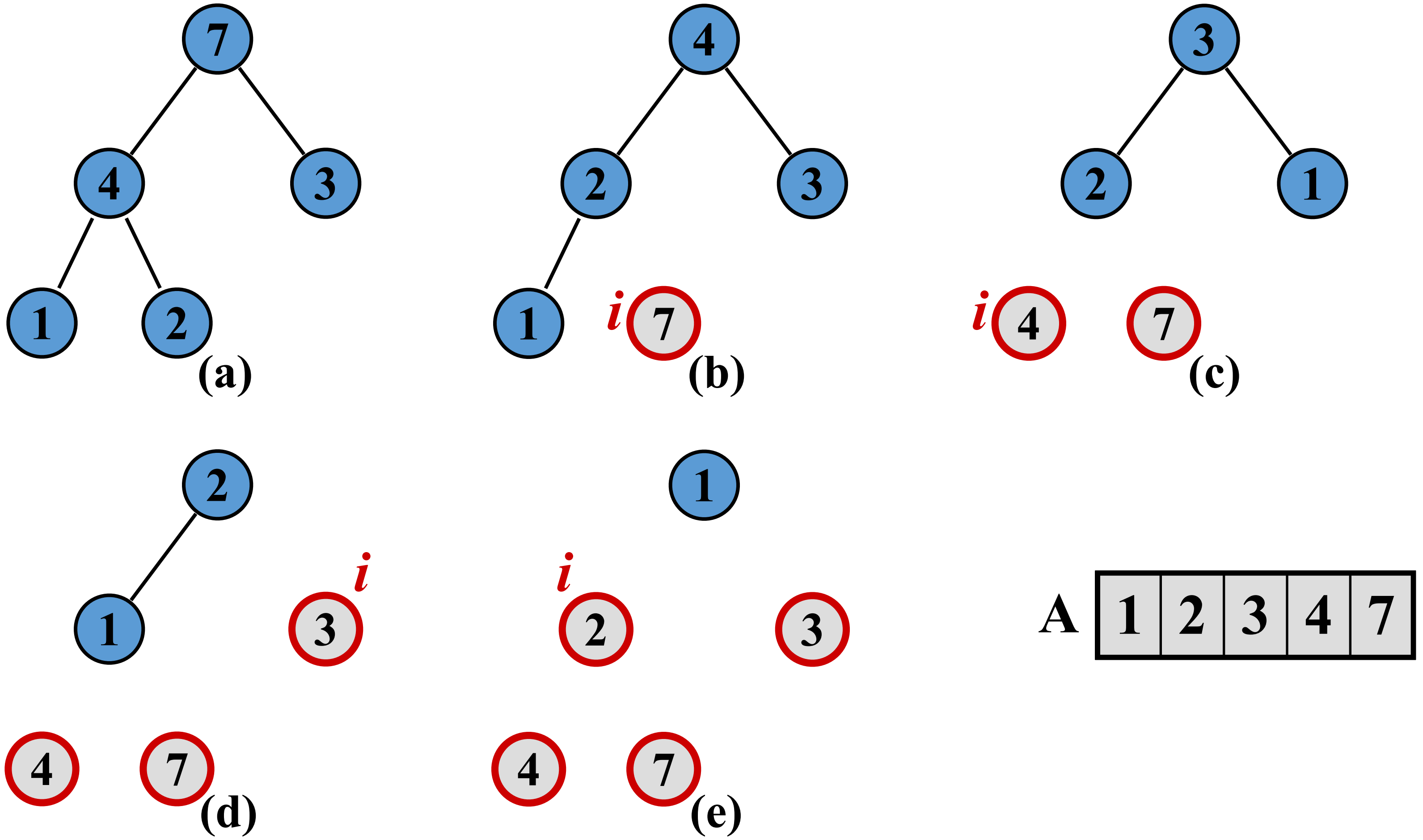
- Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.

The heapsort algorithm

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ ***downto*** 2 **do**
    exchange $A[1] \leftrightarrow A[i]$
    MAX-HEAPIFY(A, 1, $i - 1$)

Example: Heapsort Algorithm



Analysis

- Analysis of heapsort
  - BUILD-MAX-HEAP: $\mathrm{O}(n)$
  - for loop: $n-1$ times
  - Exchange elements: $\mathrm{O}(1)$
  - MAX-HEAPIFY: $\mathrm{O}(\lg n)$

  Total time: $\mathrm{O}(n \lg n)$

- The worst-case running time of HEAPSORT is $\Omega(n \lg n)$
    - Whenever we have an array that is already sorted, we take linear time to convert it to a max-heap and then $n \lg n$ time to sort it.
- When all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$
    - T. I. Fenner and A. M. Frieze,“On the Best Case of Heapsort”

Analysis

- Though heapsort is a great algorithm, it is usually not quite as fast as quicksort for large n.

- On the other hand, unlike quicksort, its performance is guaranteed.

Priority queue

- Heap implementation of priority queue
  - Max-priority queues are implemented with max-heaps. Min-priority queues are implemented with min-heaps similarly.
- Max Priority Queues
  - Maintains a dynamic set of S of elements.
  - Each with an associated value called a key.
  - Max-priority queue supports dynamic-set operations:
    - INSERT(S, x): inserts element x into set S.
    - MAXIMUM(S): returns elements of S with largest key.
    - EXTRACT-MAX(S): removes and returns element of S with largest key.
    - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume $k \geq x$'s current key value.

- Min-priority queue supports similar operation:
  - INSERT(S, x): inserts element x into set S.
  - MINIMUM(S): returns element of S with smallest key.
  - EXTRACT-MIN(S): removes and returns element of S with smallest key.
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k. Assume $k \leq x$'s current key value.

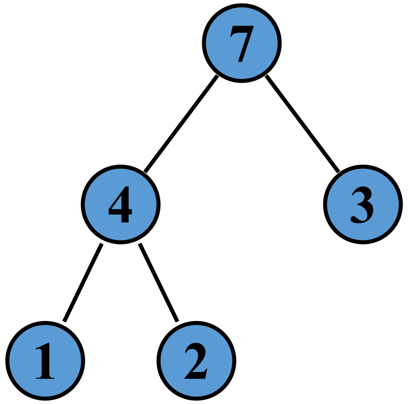
藏行顯光
成就共好
Achieve Securely
Prosper Mutually

- <u>Finding the maximum element</u>
  - Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)

1 **return** $A[1]$

- Time: $\Theta(1)$

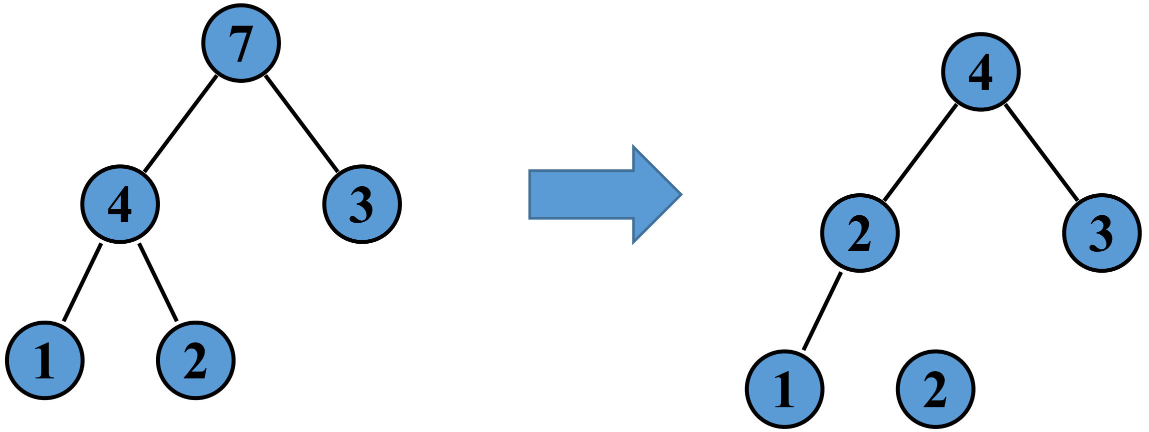


HEAP-MAXIMUM(A) returns 7

- Extracting max element
  - Given the array A:
    - Make sure heap is not empty
    - Make a copy of the maximum element (the root).
    - Make the last node in the tree the new root.
    - Re-heapify the heap, with one fewer node.
    - Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)

if $n < 1$ **then**
    error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
MAX-HEAPIFY(A, 1, $n-1$) ▶remakes heap
return max



$max = 7$

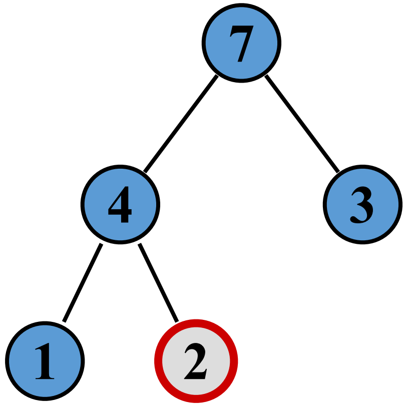
- Analysis: constant time assignments plus time for MAX-HEAPIFY.
- Time: $\mathrm{O}(\lg n)$

Increasing key value

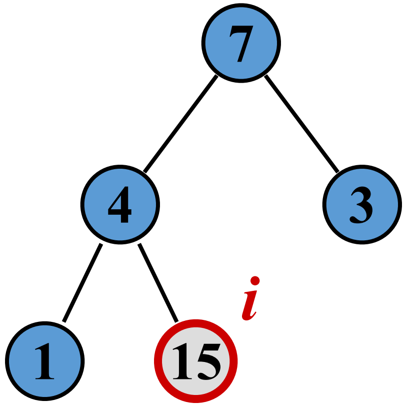
- Given set S, element x and new key value k:
  - Make sure $k \geq x$'s current key.
  - Update x's key value to k.
  - Traverse the tree upward comparing x to its parent and swapping keys if necessary, until x's key in smaller than parent's key.

HEAP-INCREASE-KEY(A, i, key)

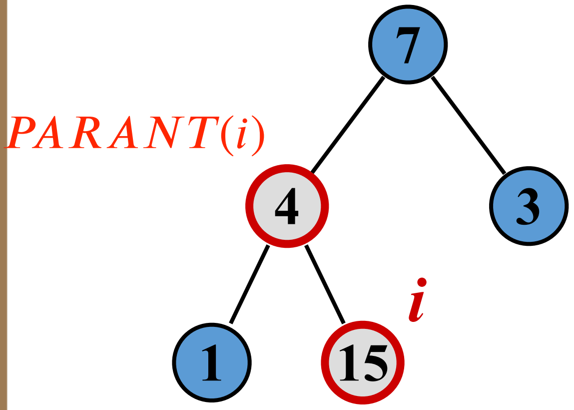
if $key < A[i]$ **then**
    error "new key is smaller than current key"
$A[i] \leftarrow key$
while $i > 1$ *and* $A[PARENT(i)] < A[i]$ **do**
    exchange $A[i] \leftrightarrow A[PARENT(i)]$
    $i \leftarrow PARENT(i)$



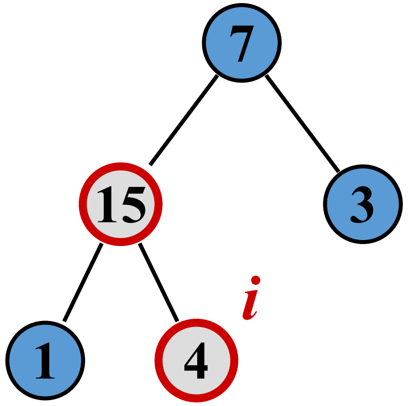
HEAP-INCREASE-KEY(A, 5, 15)



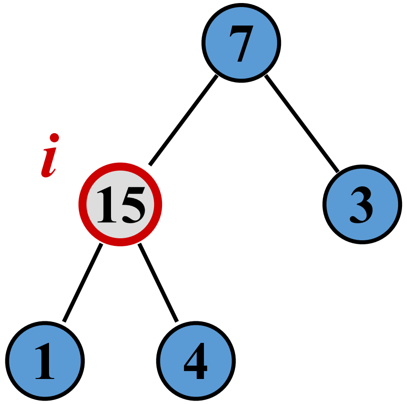
$A[i] \leftarrow key$



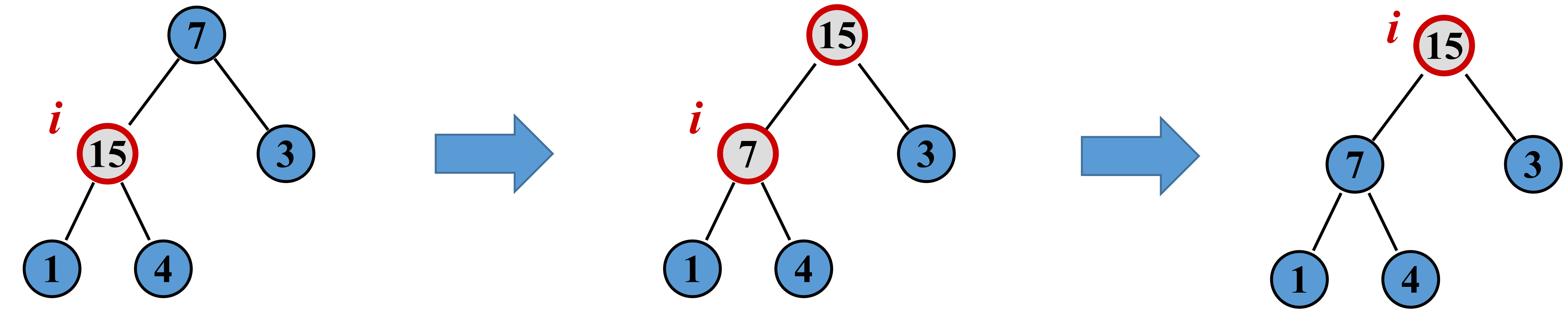
$A\left[PARANT(i)\right] < A[i]$



$A\left[PARANT(i)\right] \leftrightarrow A[i]$



$i \leftarrow PARANT(i)$



$A\left[PARANT(i)\right] < A[i]$ $\quad$ $A\left[PARANT(i)\right] \leftrightarrow A[i]$ $\quad$ $i \leftarrow PARANT(i)$

- Analysis: Upward path from node i has length $O(\lg n)$ in an n-element heap.
- Time: $O(\lg n)$

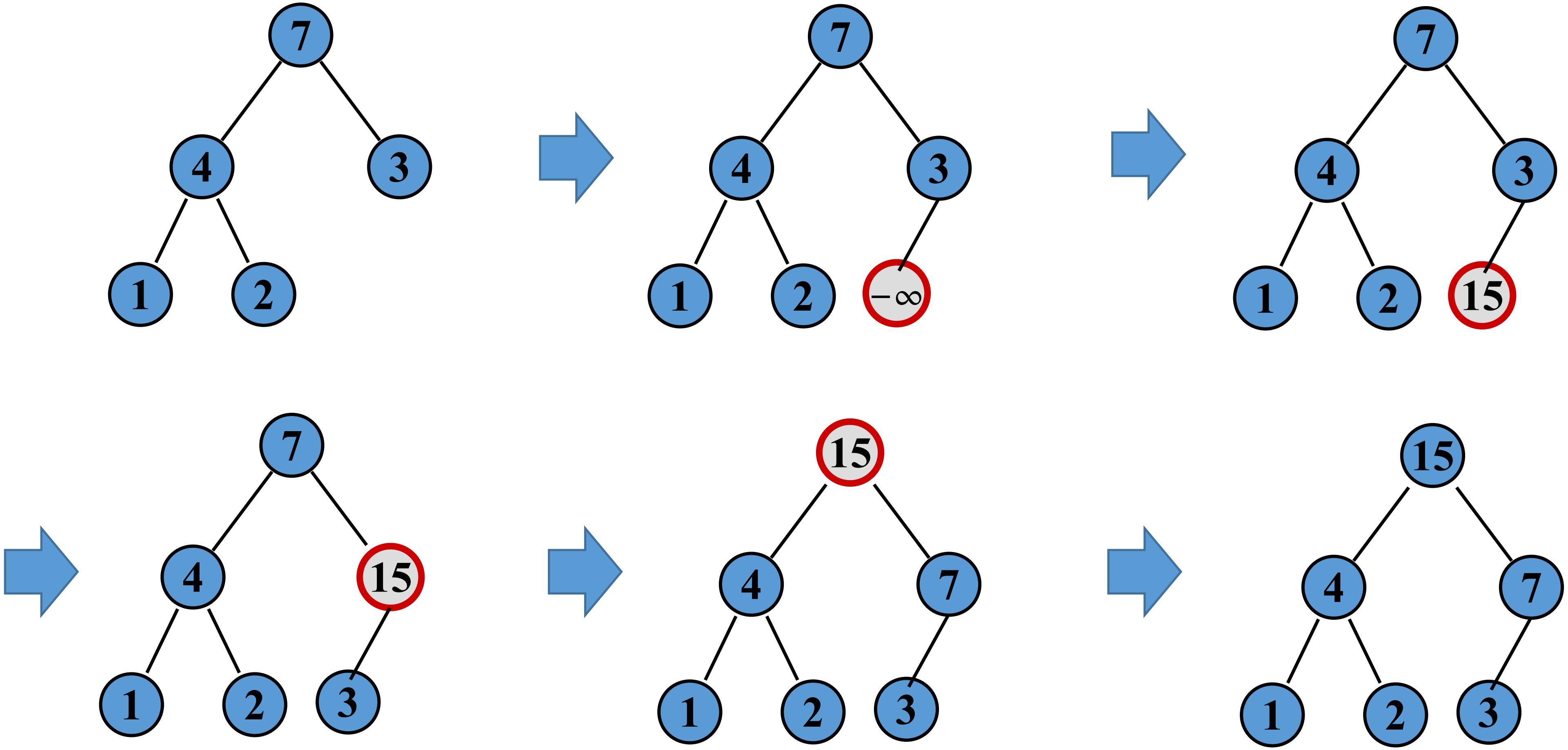
藏行顯光 成就共好
Achieve Securely Prosper Mutually

Inserting into the heap

- Given a key k to insert into the heap:
  - Insert a new node in the key last position in the tree with key $-\infty$
  - Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure defined above.

MAX-HEAP-INSERT(A, key, n)

1 $A[i+1] \leftarrow -\infty$
2 HEAP-INCREASE-KEY$(A, n+1, key)$



- Analysis: constant time assignments plus time for HEAP-INCREASE-KEY
- Time: $O(\lg n)$

- **Min-priority queue** operations are implemented similarly with min-heaps.

藏行顯光 成就共好
Achieve Securely Prosper Mutually

藏行顯光
成就共好

Achieve Securely
Prosper Mutually

國立成功大學 九十週年
90th Anniversary of NCKU

國立成功大學
National Cheng Kung University
1931