

Data Structures and Algorithm

Xiaoqing Zheng
zhengxq@fudan.edu.cn



Investment problem

- *Suppose that you try to in the stock market.*

Deterministic algorithm

STOCK-INVESTMENT(n)

1. $best = 0$
2. **for** $i \leftarrow 1$ **to** n
3. **do** investigate candidate i
4. **if** candidate i is better than candidate $best$
5. **then** $best \leftarrow i$
6. buy candidate i

Total cost: $O(nc_i + mc_b)$

Worst case: $O(nc_b)$

Indicator random variable

Indicator random variable $I\{A\}$ associated with event A is defined as

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur.} \end{cases}$$

Analysis of investment problem

Let X be the random variable whose value equals the numbers of times we buy a new stock.

$$X_i = I\{\text{candidate } i \text{ is better}\} = \begin{cases} 1 & \text{if candidate } i \text{ is better,} \\ 0 & \text{if candidate } i \text{ is not better.} \end{cases}$$

and

$$X = X_1 + X_2 + \dots + X_n$$

Analysis of investment problem

Now we can compute $E[X]$:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^n X_i\right] \\ &= \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n 1/i \\ &= \ln n + O(1) \end{aligned}$$

Average cost: $O(\ln n c_b)$

Randomized algorithm

RANDOMIZED-STOCK-INVESTMENT(n)

1. Randomly permute the list of candidates
2. $best = 0$
3. **for** $i \leftarrow 1$ **to** n
4. **do** investigate candidate i
5. **if** candidate i is better than candidate $best$
6. **then** $best \leftarrow i$
7. buy candidate i

Divide and conquer

Quicksort an n -element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.



2. **Conquer:** Recursively sort the two subarrays.
3. **Combine:** Trivial.

Key: Linear-time partitioning subroutine.

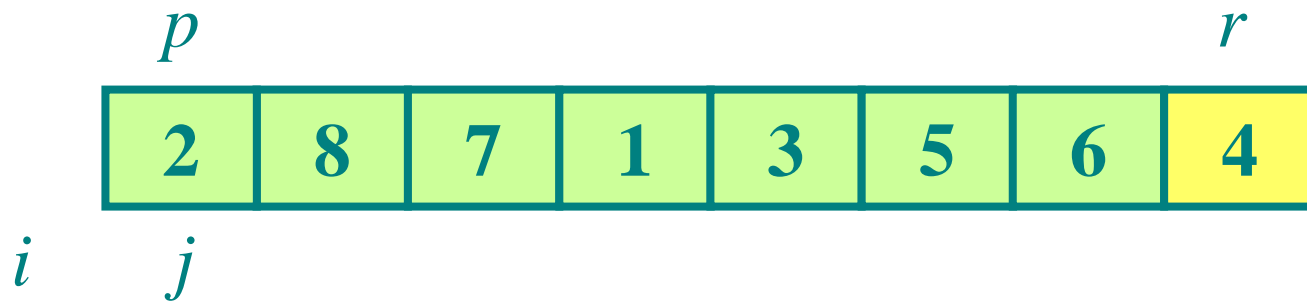
Pseudocode for quicksort

QUICKSORT(A, p, r)

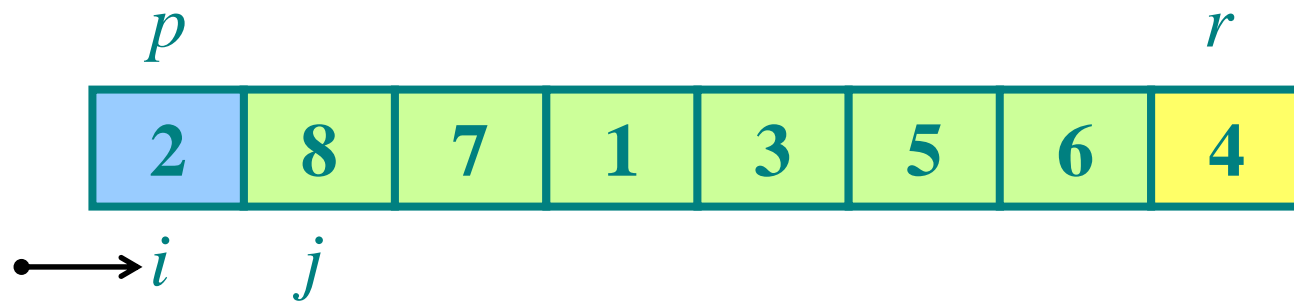
1. **if** $p < r$
2. **then** $q \leftarrow \text{PARTITION}(A, p, r)$
3. **QUICKSORT**($A, p, q - 1$)
4. **QUICKSORT**($A, q + 1, r$)

Initial call: **QUICKSORT**($A, 1, n$)

Example of partitioning

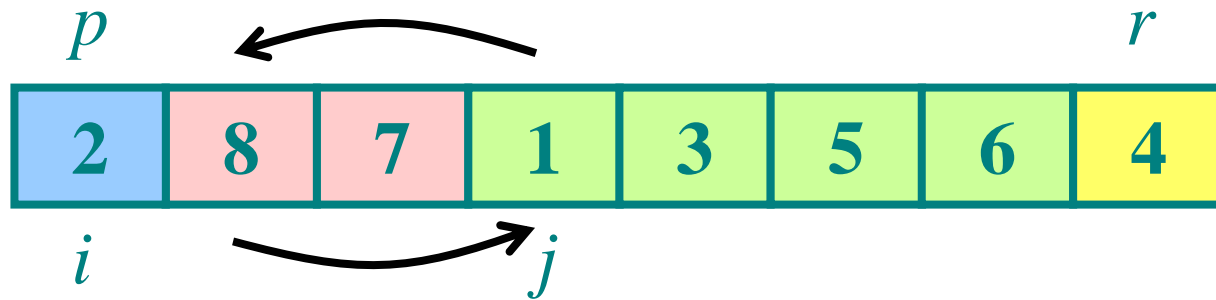


Example of partitioning



Example of partitioning

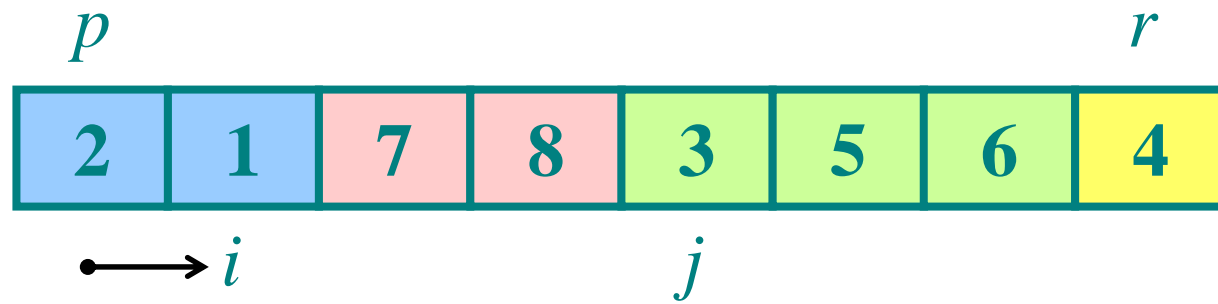
Example of partitioning



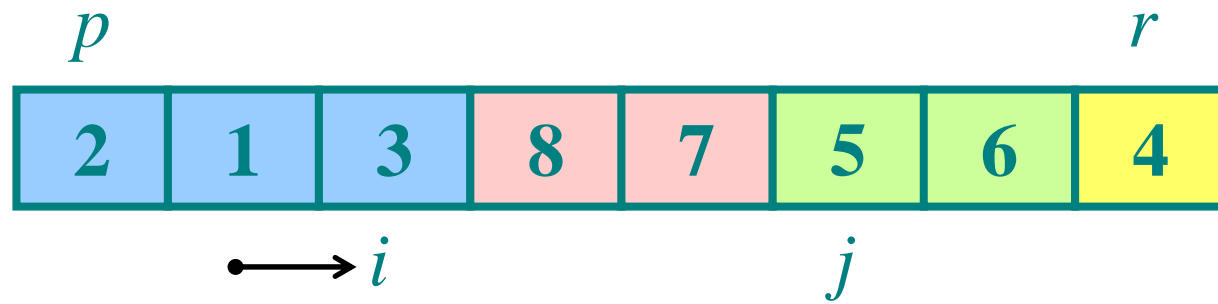
now: $A[j] \leq x$

- 4. **do if** $A[j] \leq x$
- 5. **then** $i \leftarrow i + 1$
- 6. exchange $A[i] \leftrightarrow A[j]$

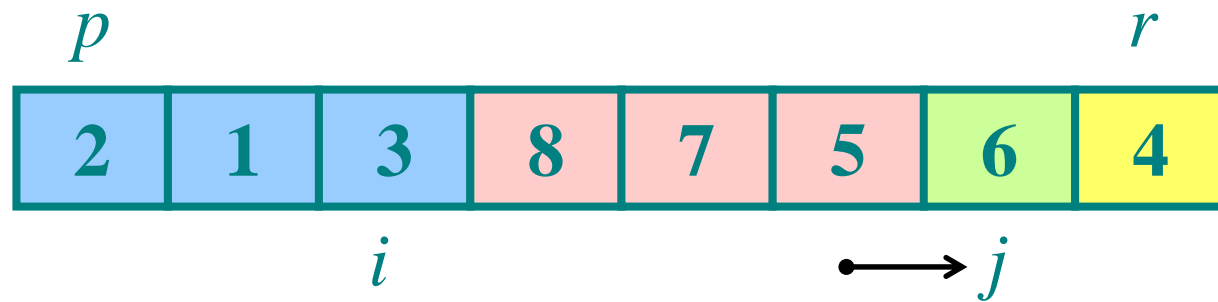
Example of partitioning



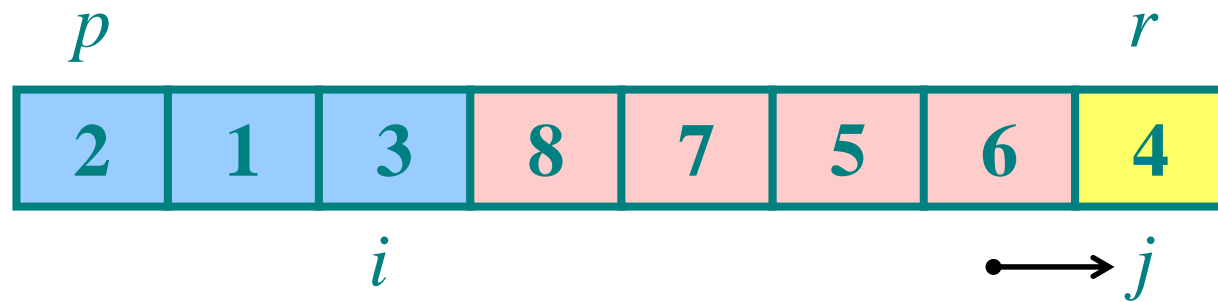
Example of partitioning



Example of partitioning

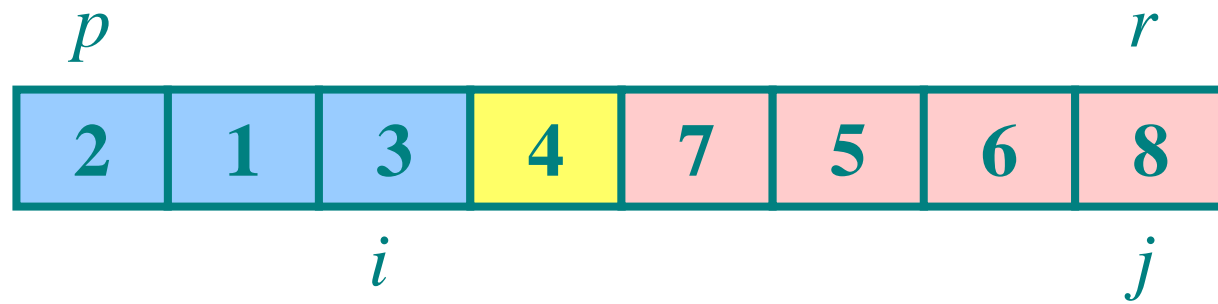


Example of partitioning



7. exchange $A[r] \leftrightarrow A[i + 1]$

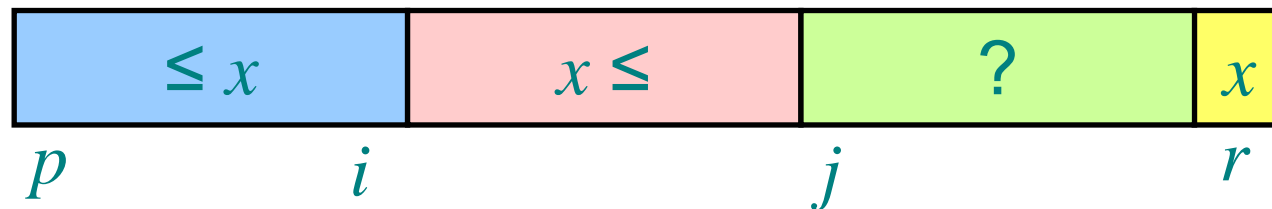
Example of partitioning



Partitioning subroutine

PARTITION(A, p, r) // $A[p \dots r]$

1. $x \leftarrow A[r]$ // pivot = $A[p]$
2. $i \leftarrow p - 1$
3. **for** $j \leftarrow p$ **to** $r - 1$
4. **do if** $A[j] \leq x$
5. **then** $i \leftarrow i + 1$
6. exchange $A[i] \leftrightarrow A[j]$
7. exchange $A[r] \leftrightarrow A[i + 1]$
8. **return** $i + 1$



Worst-case of quicksort

- ❑ Input sorted or reverse sorted.
- ❑ Partition around min or max element.
- ❑ One side of partition always has no elements.

$$\begin{aligned}T(n) &= T(0) + T(n - 1) + \Theta(n) \\&= \Theta(1) + T(n - 1) + \Theta(n) \\&= T(n - 1) + \Theta(n) \\&= \Theta(n^2) \text{ (*arithmetic series*)}\end{aligned}$$

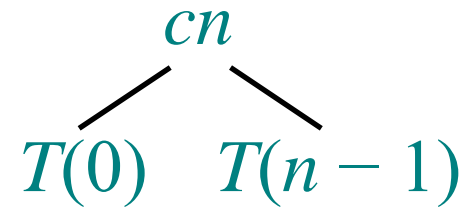
Worst-case recursion tree

$$T(n) = T(0) + T(n - 1) + cn$$

$$T(n)$$

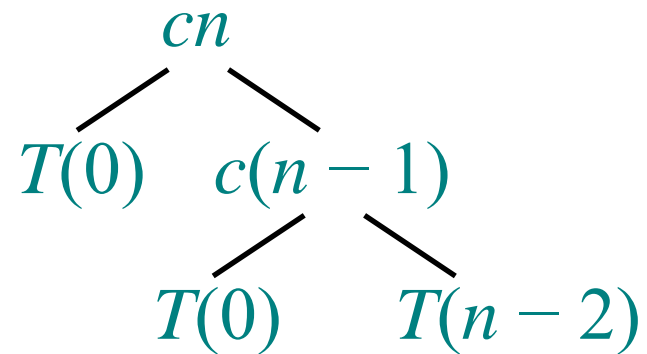
Worst-case recursion tree

$$T(n) = T(0) + T(n - 1) + cn$$



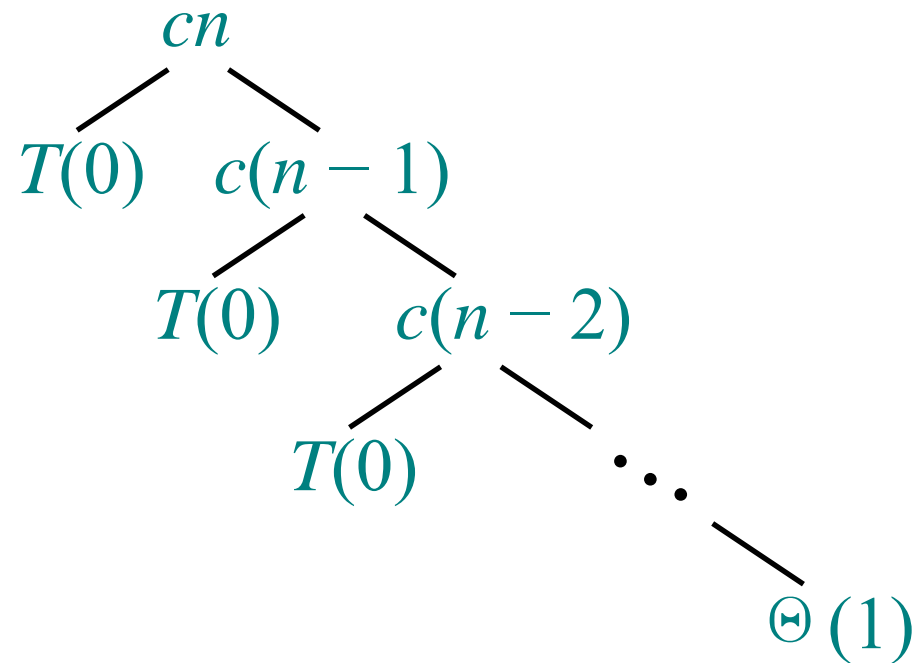
Worst-case recursion tree

$$T(n) = T(0) + T(n - 1) + cn$$



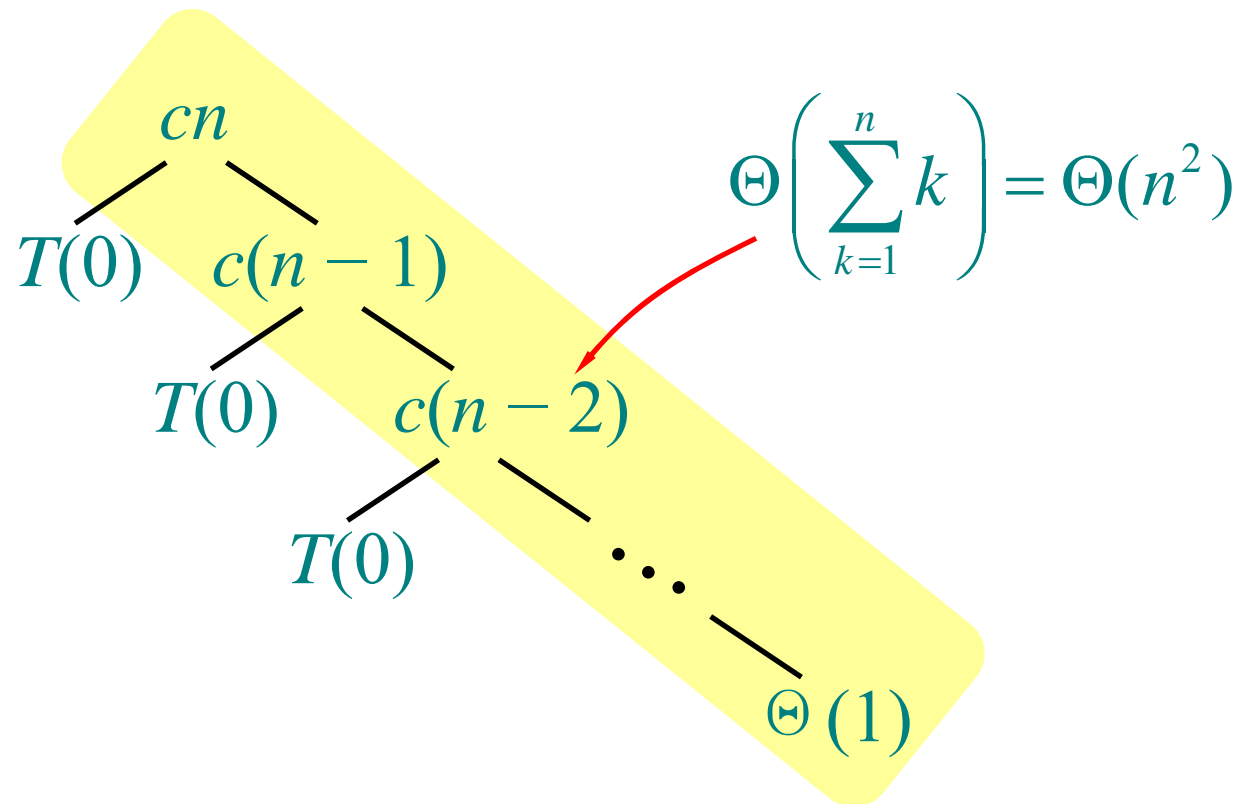
Worst-case recursion tree

$$T(n) = T(0) + T(n - 1) + cn$$



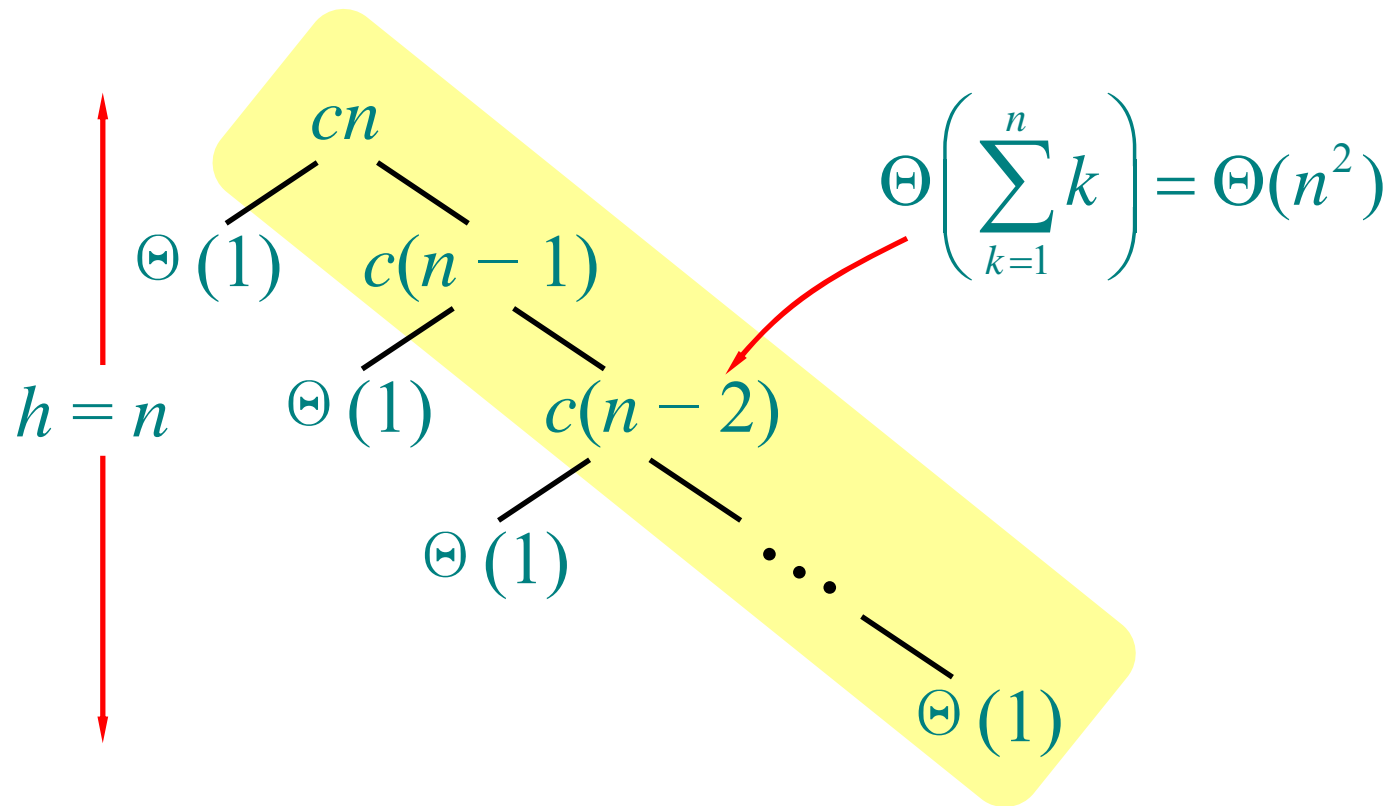
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



Nice-case analysis

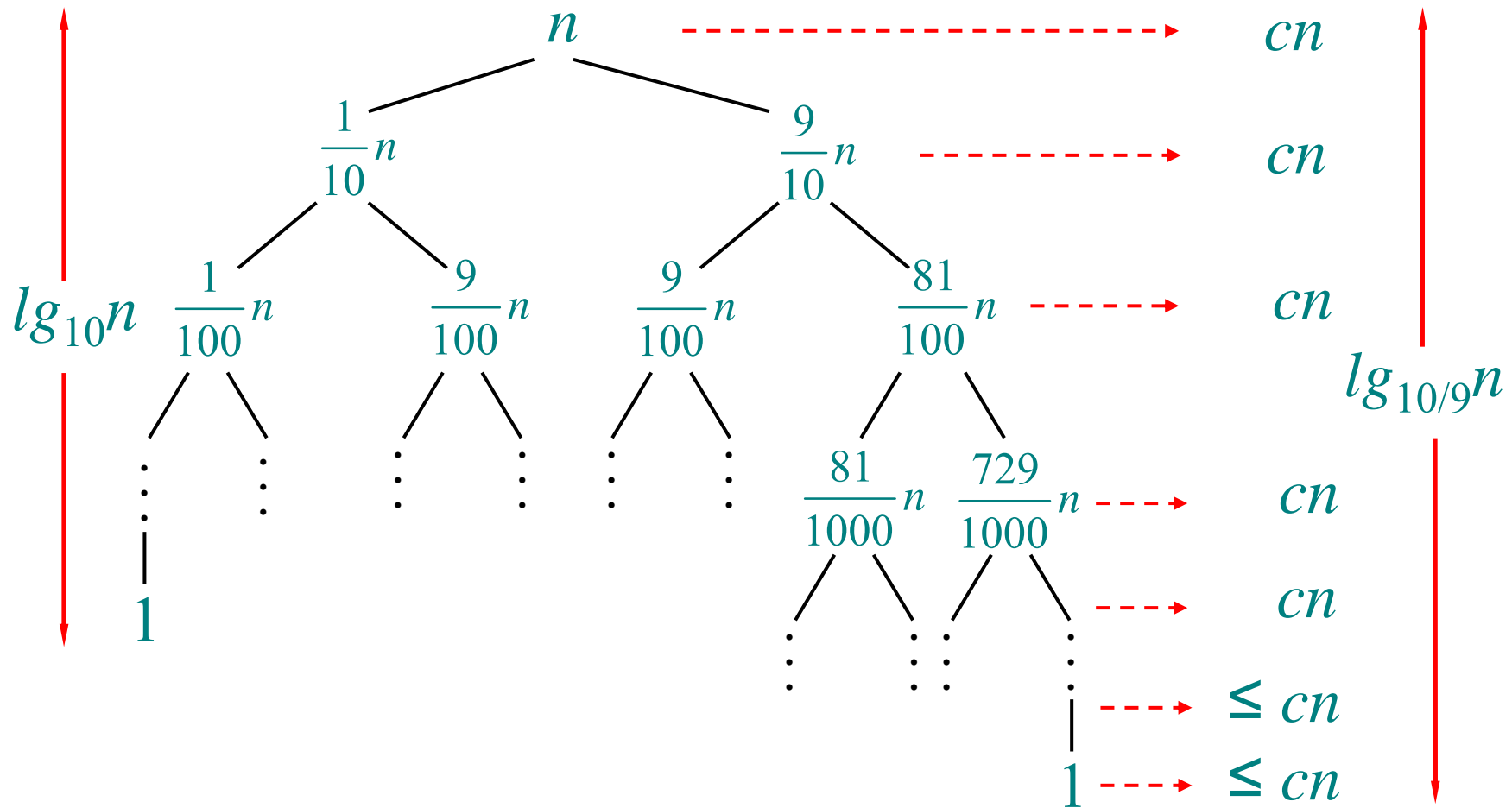
If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \quad (\text{same as merge sort}) \end{aligned}$$

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

Analysis of nice case



Total: $O(n \lg n)$

Randomized quicksort

Partition around a *random* element around $A[t]$, where t chosen uniformly at random from $\{p \dots r\}$

RANDOMIZED-PARTITION(A, p, r)

1. $i \leftarrow \text{RANDOM}(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. **return** PARTITION(A, p, r)

We will show that the expected time is $O(n \lg n)$

Analysis of quicksort

Running time of **QUICKSORT** is $O(n + X)$

- n be the number of call to PARTITION
- X be the number of comparisons performed in line 4 of PARTITION

```
PARTITION( $A, p, r$ )  //  $A[p \dots r]$ 
1.  $x \leftarrow A[r]$   // pivot =  $A[p]$ 
2.  $i \leftarrow p - 1$ 
3. for  $j \leftarrow p$  to  $r - 1$ 
4.   do if  $A[j] \leq x$ 
5.     then  $i \leftarrow i + 1$ 
6.         exchange  $A[i] \leftrightarrow A[j]$ 
7. exchange  $A[r] \leftrightarrow A[i + 1]$ 
8. return  $i + 1$ 
```

Running time and comparisons

$Z_{ij} = \{ z_i, z_{i+1}, \dots, z_j \}$ to be the set of elements between z_i and z_j , inclusive.

$X_{ij} = I \{ z_i \text{ is compared to } z_j \}$

Total number of comparisons performed by the algorithm

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

Number of comparisons

{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }



{ 1, 2, 3, 4, 5, 6 } 7 { 8, 9, 10 }

The pivot element 7 is compared to all other elements, but no number from the first set (e.g. 2) is or ever will be compared to any number from the second set (e.g. 9).

Expected running time

$$\begin{aligned} E(X) &= E \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr \{ z_i \text{ is compared to } z_j \} \end{aligned}$$

$$\begin{aligned} \Pr \{ z_i \text{ is compared to } z_j \} &= \Pr \{ z_i \text{ is first pivot chosen from } Z_{ij} \} \\ &\quad + \Pr \{ z_j \text{ is first pivot chosen from } Z_{ij} \} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \end{aligned}$$

Expected running time

$$\begin{aligned} E(X) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \quad \text{note: } k = j - i \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(\lg n) \quad \sum_{k=1}^n \frac{1}{k} = \ln n + O(1) \\ &= O(n \lg n) \quad \square \end{aligned}$$

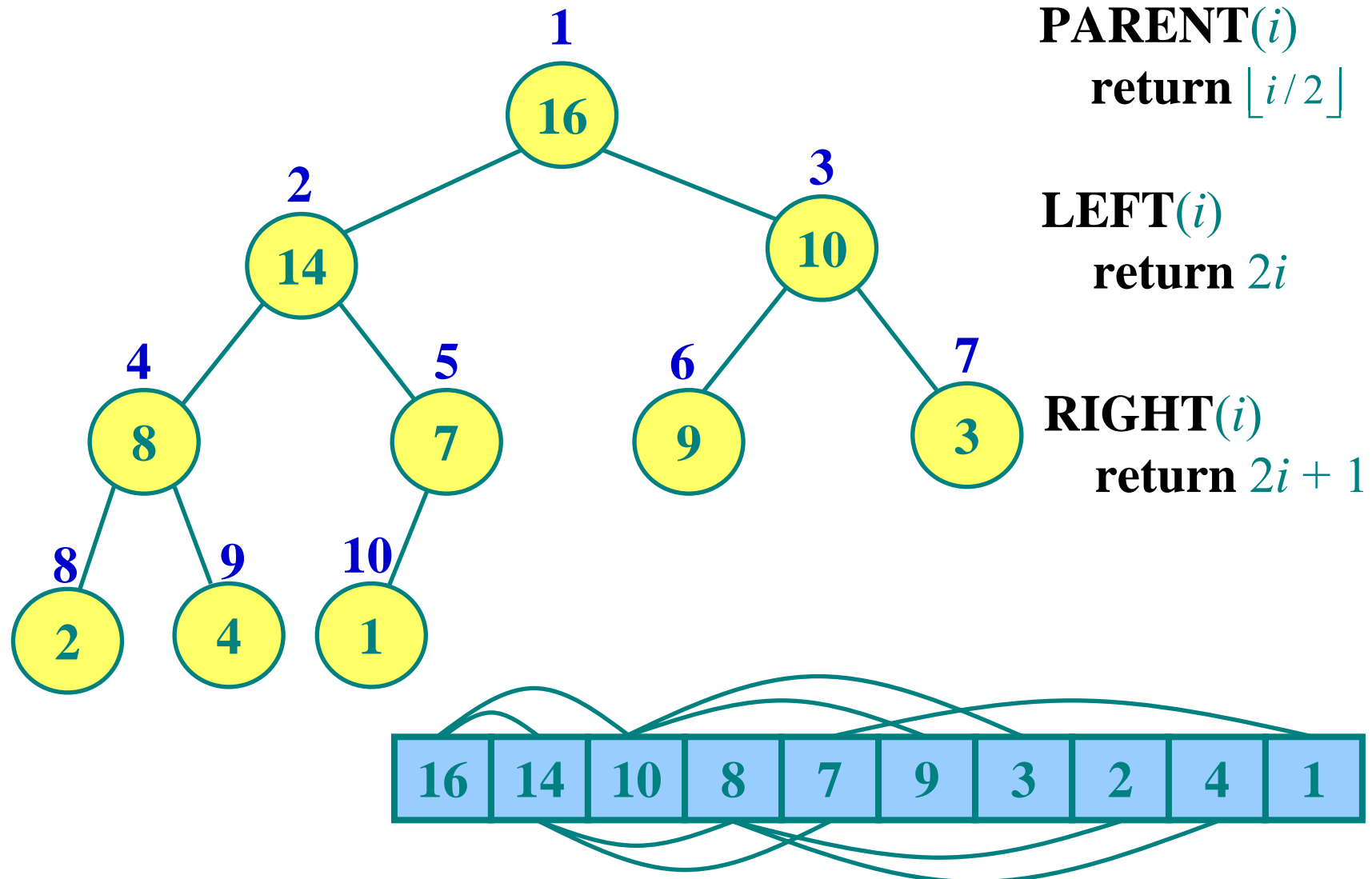
Quicksort in practice

- ❑ Quicksort is a *general-purpose* sorting algorithm.
- ❑ Quicksort is typically over *twice* as fast as merge sort.
- ❑ Quicksort behaves well even with *caching and virtual memory*.
- ❑ Quicksort is *great*.

Randomized algorithms

- ❑ Algorithms that make decisions based on random.
- ❑ Can "fool" the adversary.
- ❑ The running time (or even correctness) is a random variable; we measure the *expected* running time.
- ❑ We assume all random choices are *independent*.
- ❑ This is *not* the average case!

Max heap



Max-heaps and min-heaps

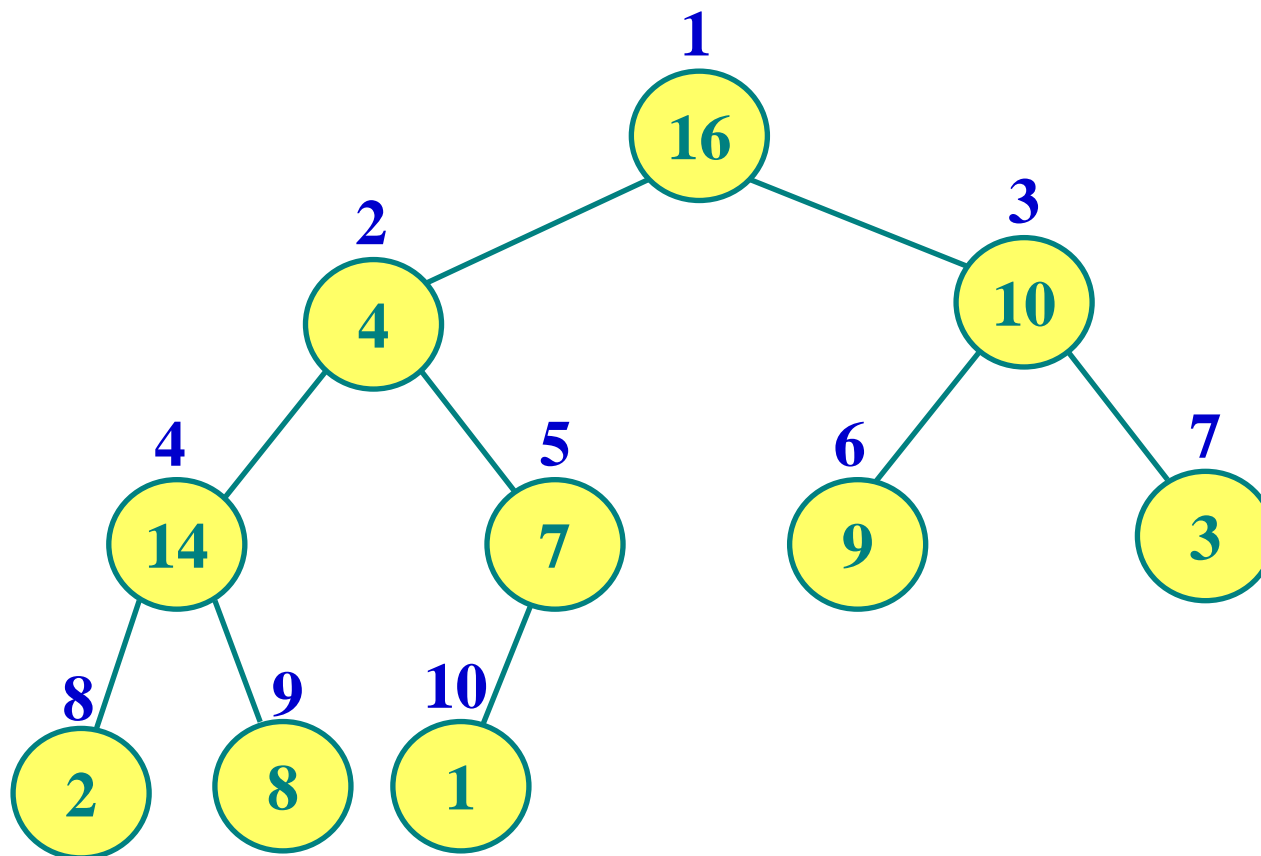
Max-heap property is that for every node i other than the root

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

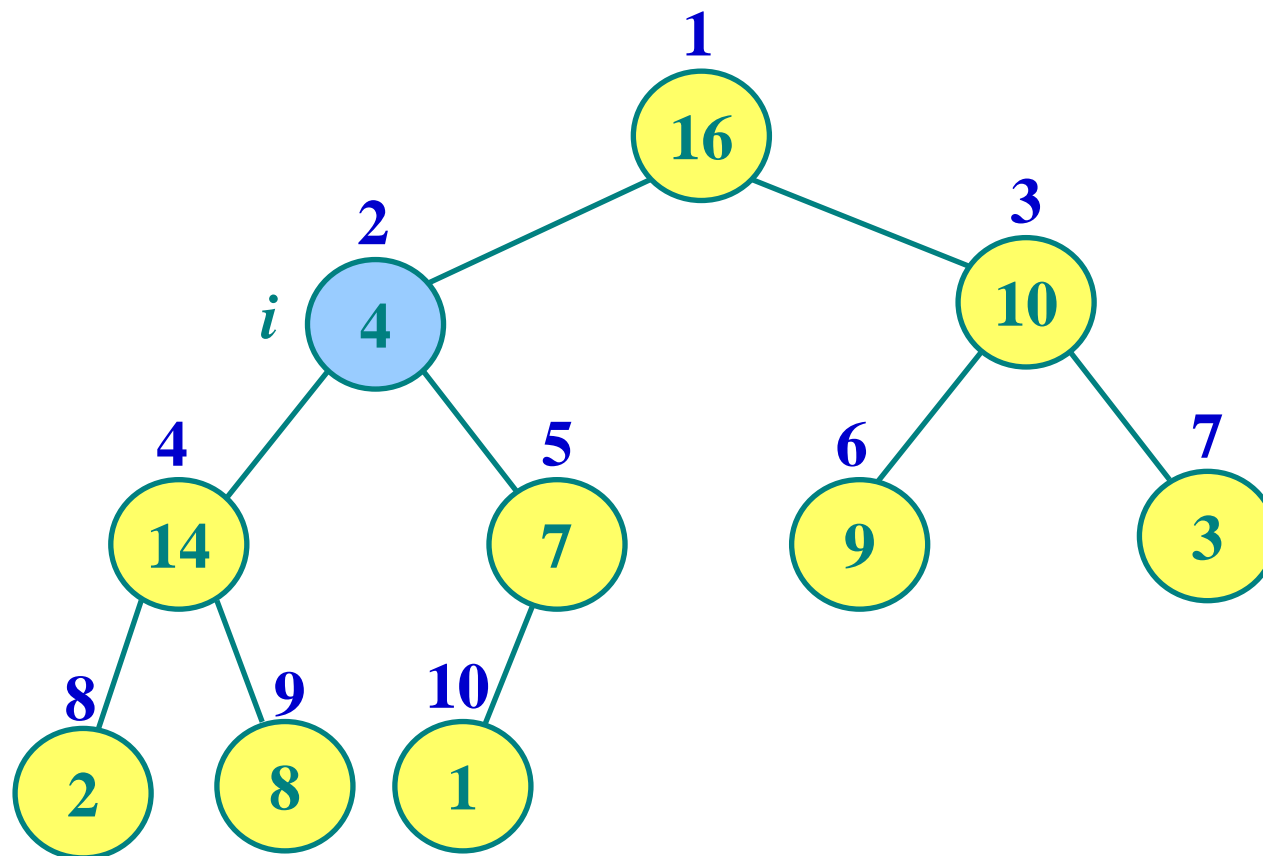
Min-heap property is that for every node i other than the root

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

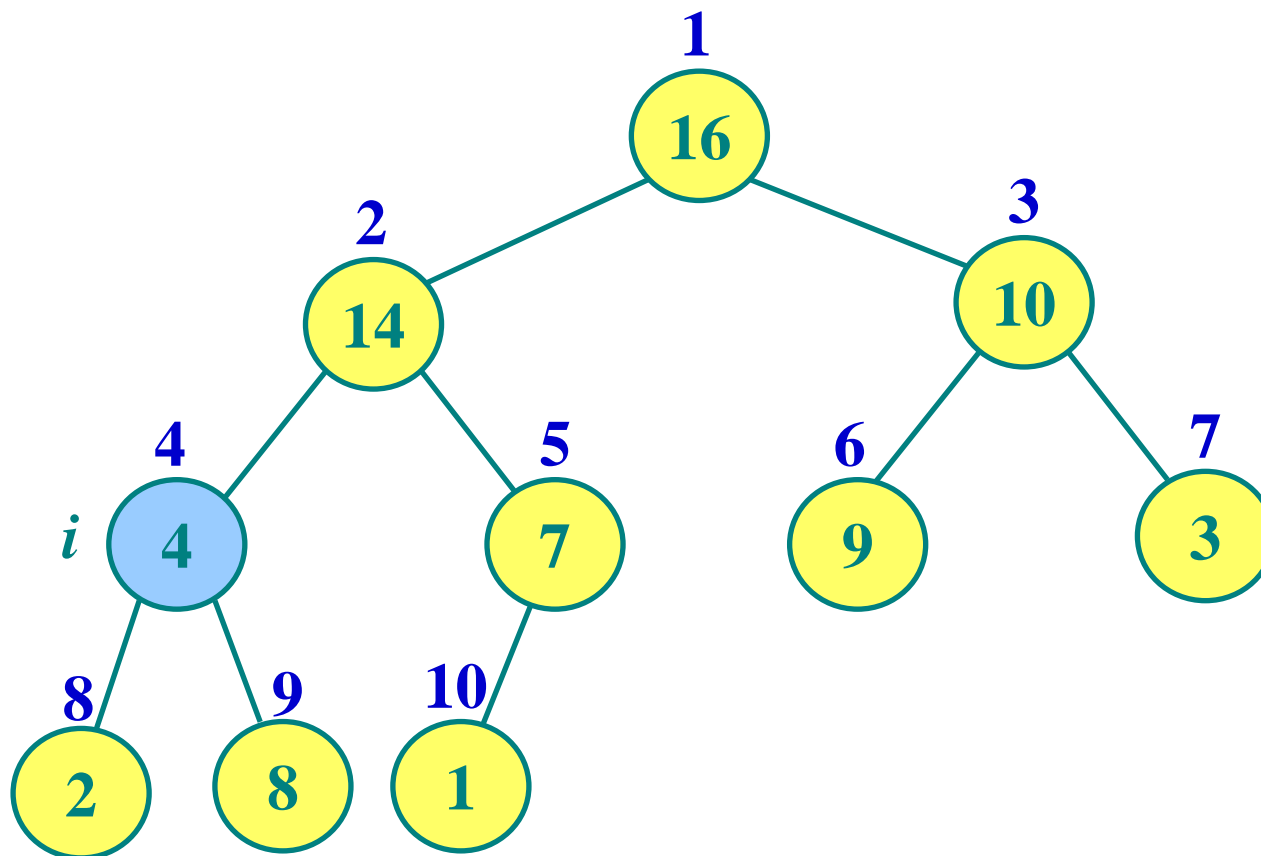
Max-heapify



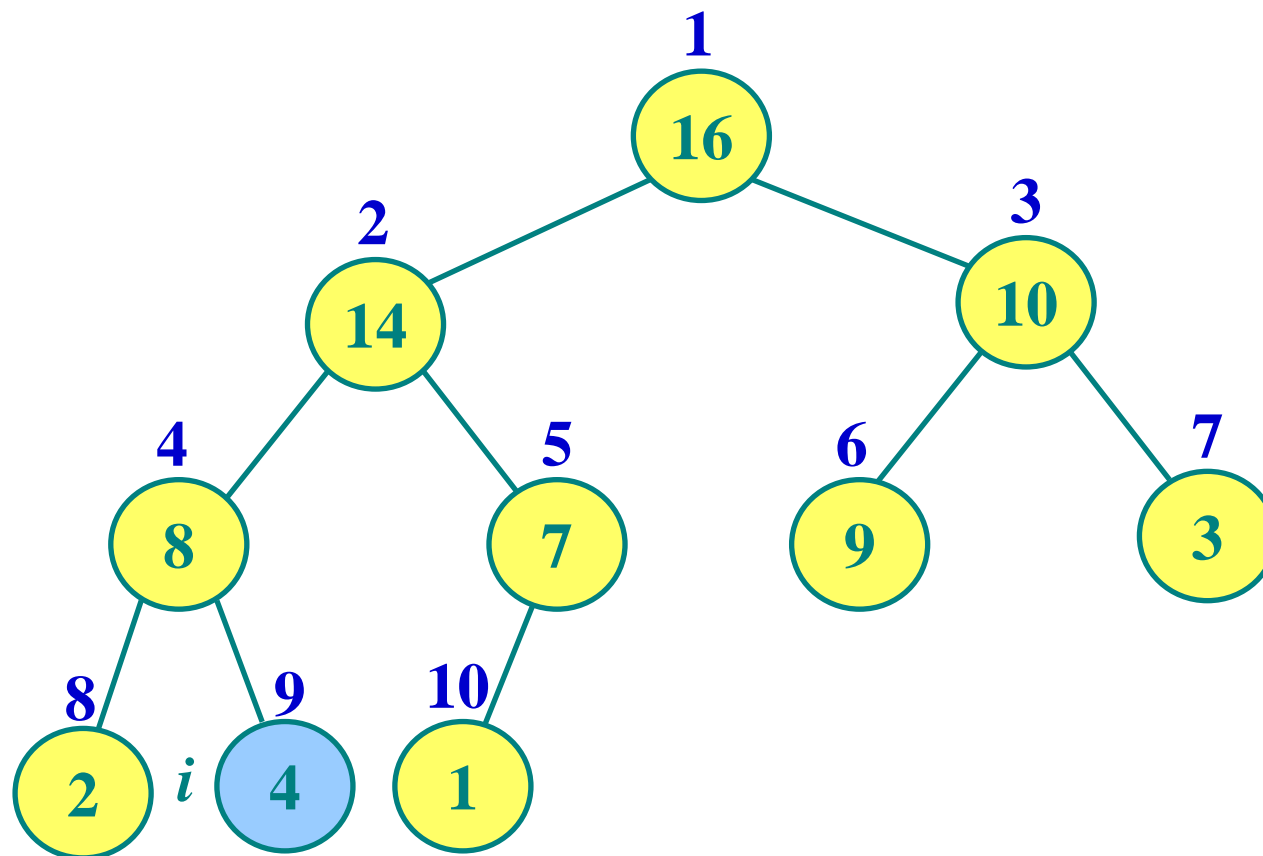
Max-heapify



Max-heapify



Max-heapify



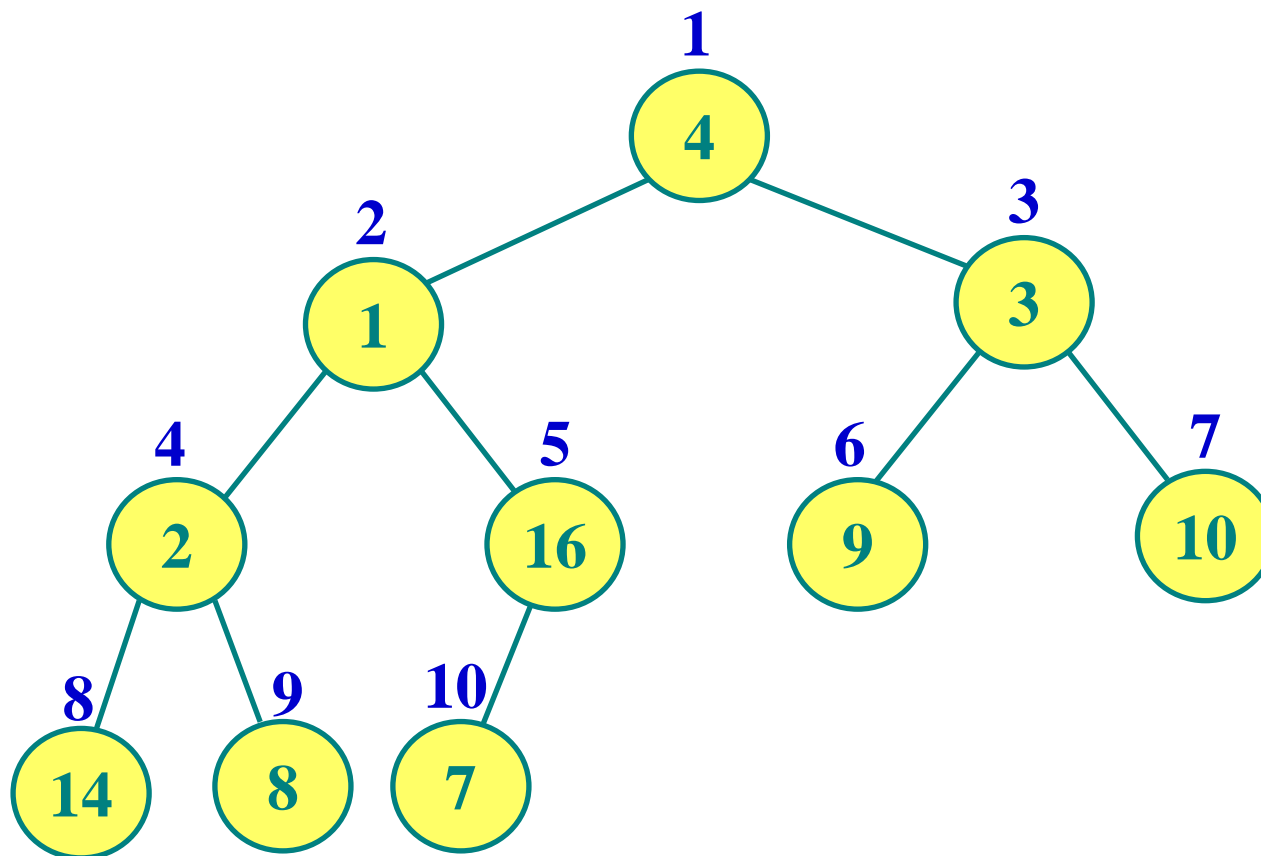
$$O(h) = O(\lg n)$$

Max-heapify

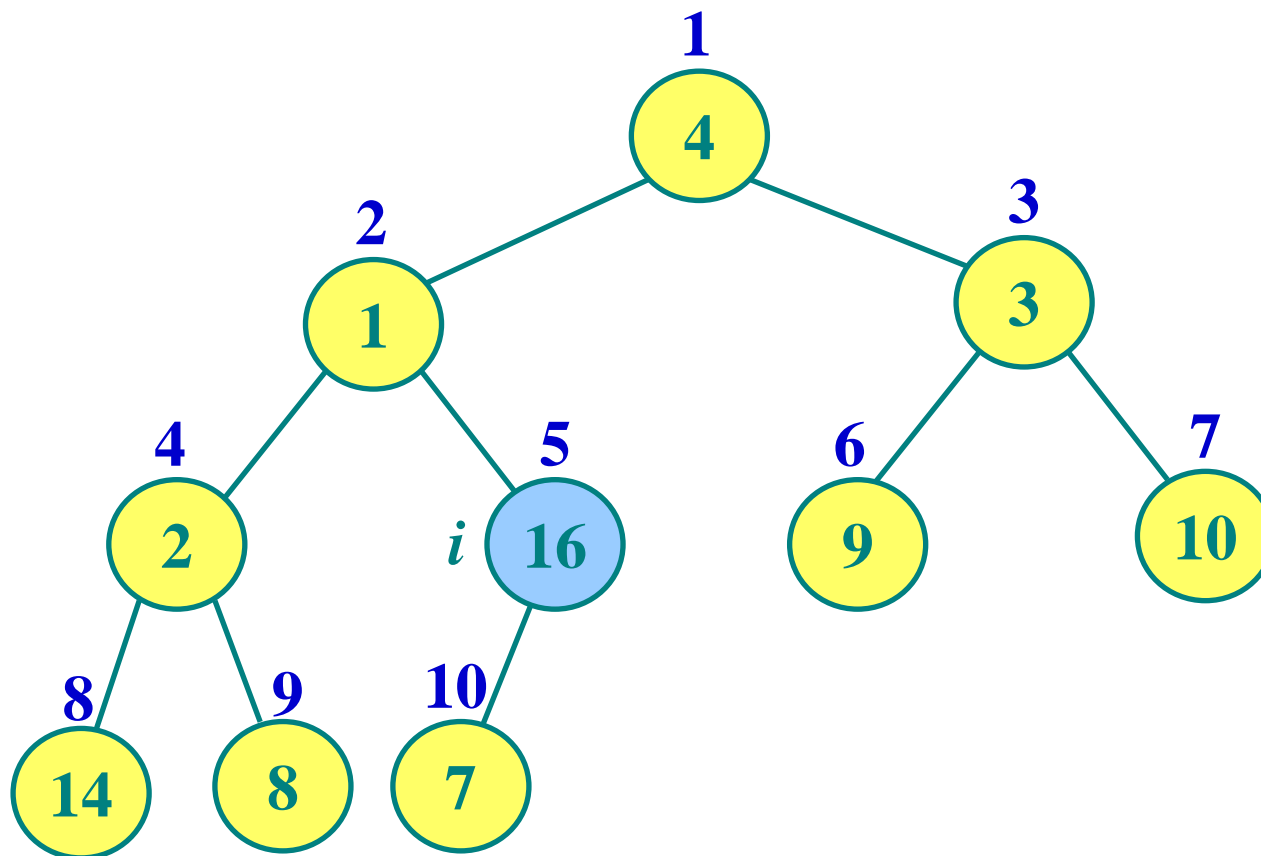
MAX-HEAPIFY(A, i)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq \text{heap-size}[A]$ **and** $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq \text{heap-size}[A]$ **and** $A[r] > A[\text{largest}]$
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\text{largest}]$
10. **MAX-HEAPIFY**($A, \text{largest}$)

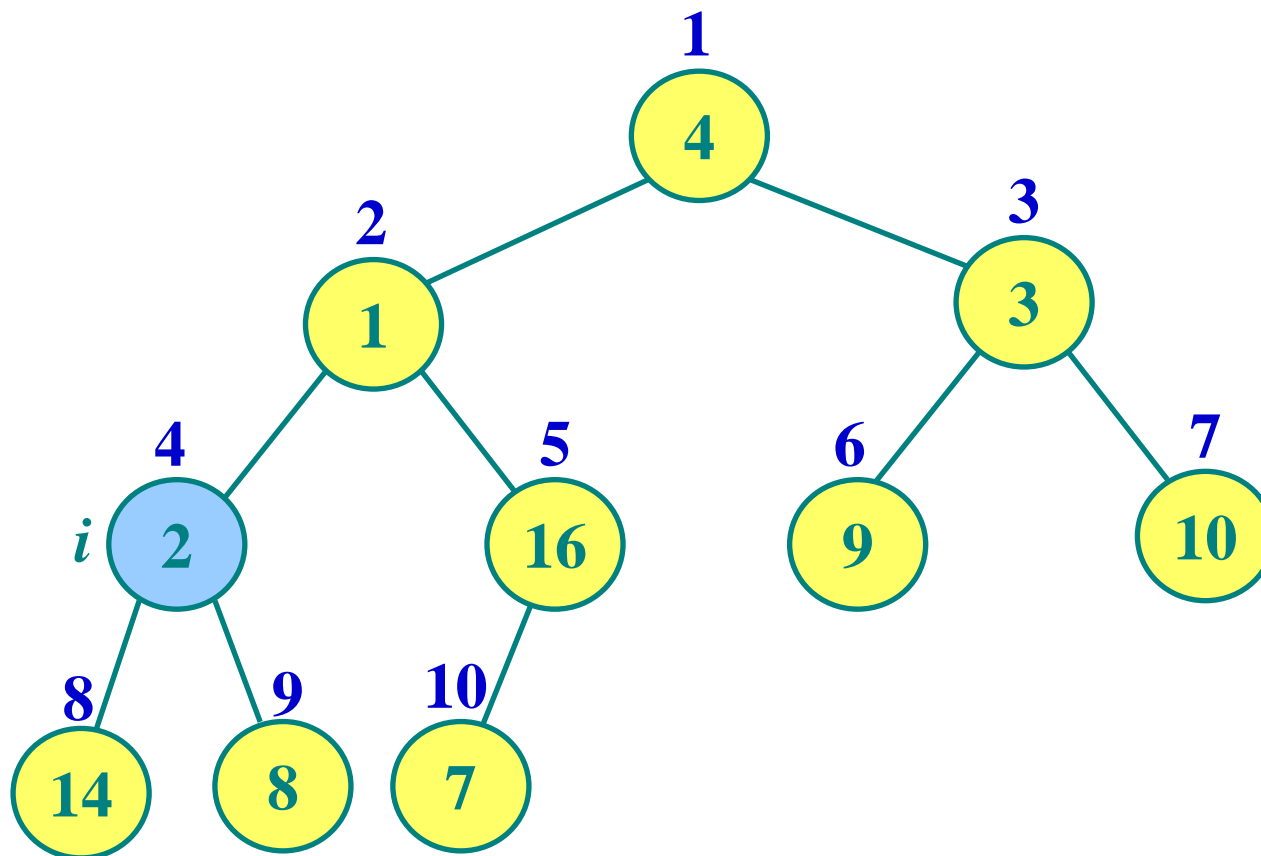
Max-heapify



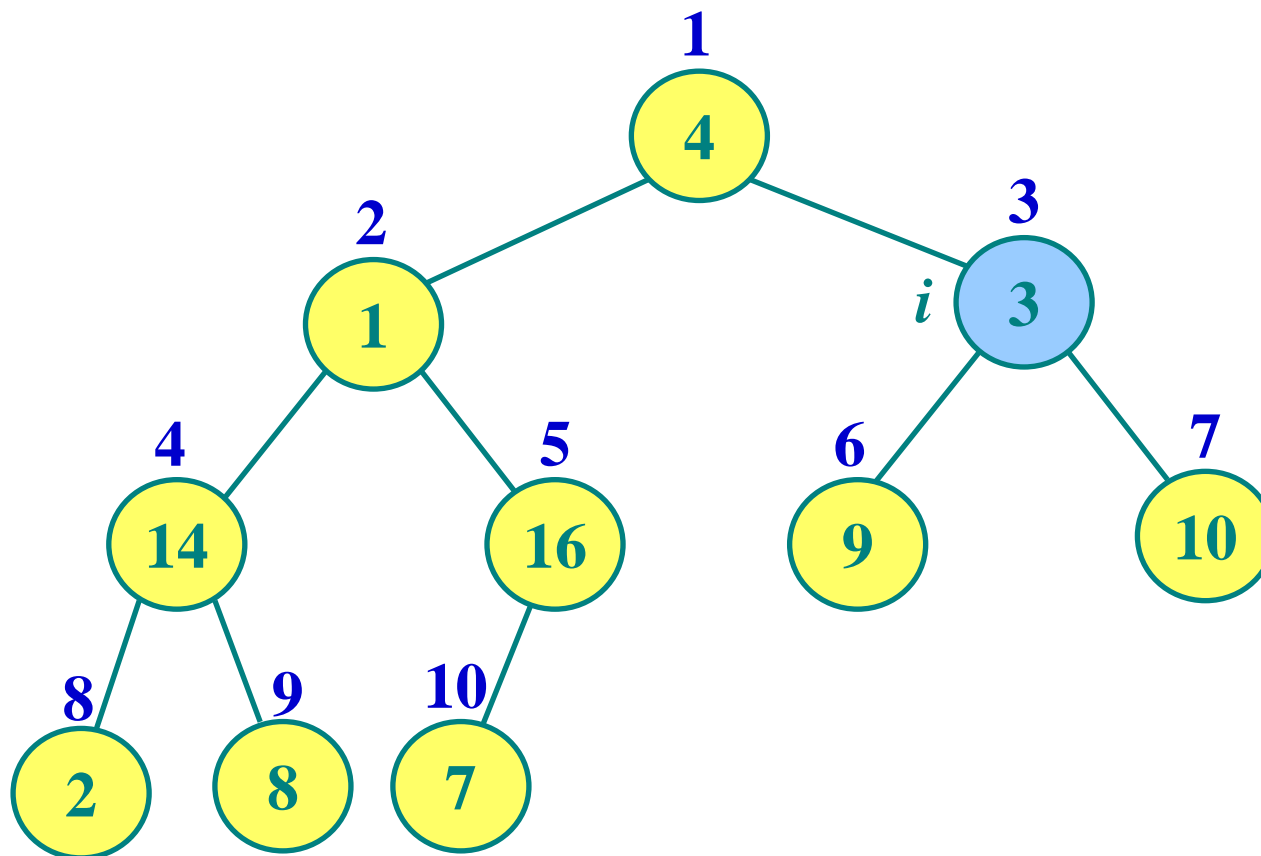
Max-heapify



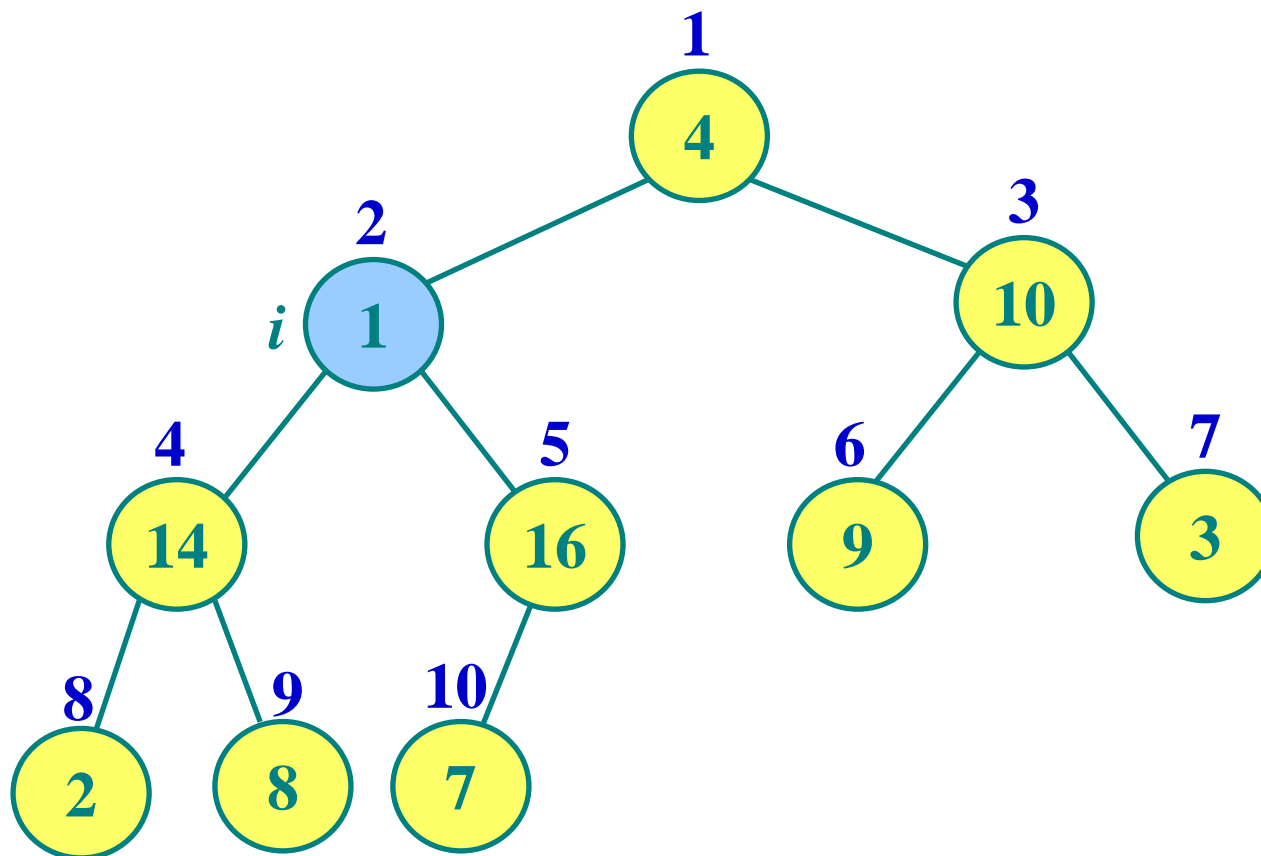
Max-heapify



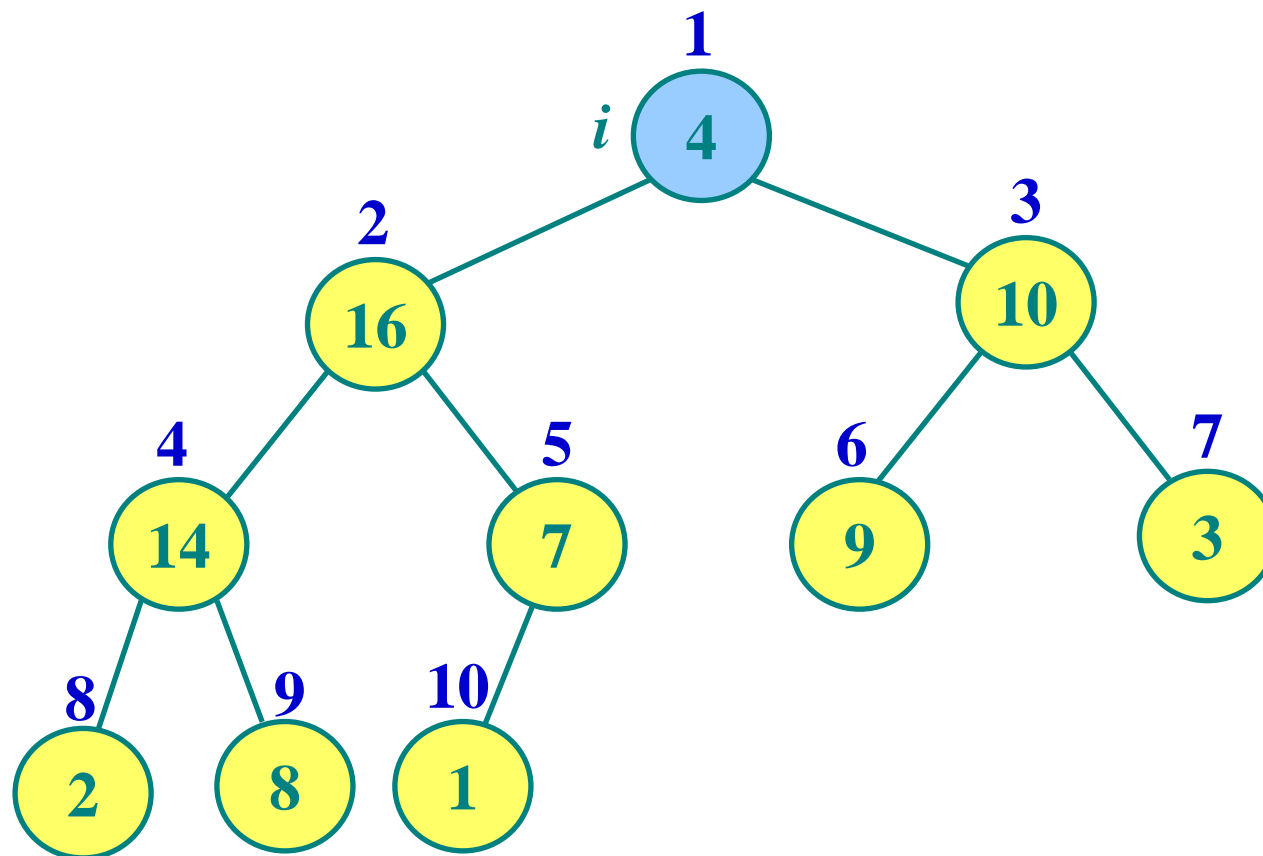
Max-heapify



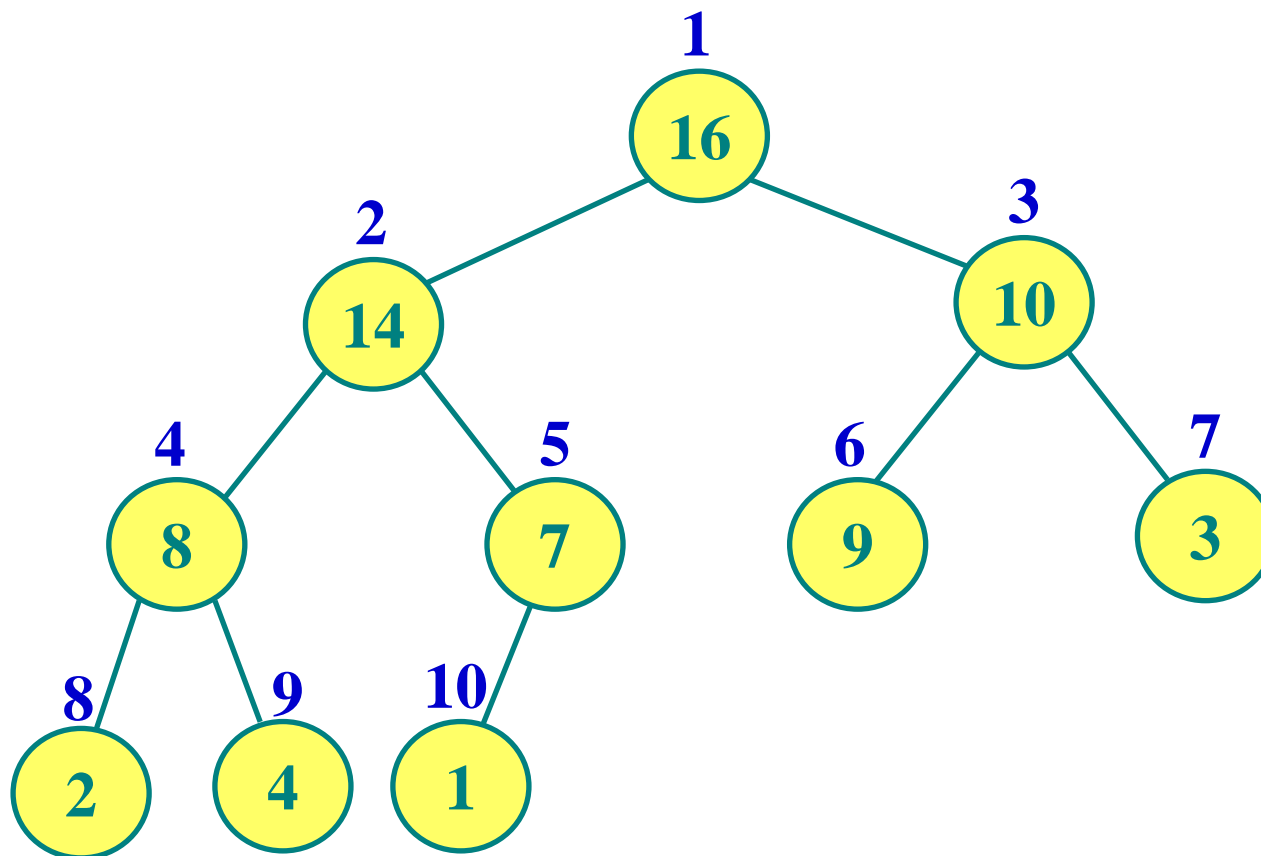
Max-heapify



Max-heapify



Max-heapify



done

Build max heap

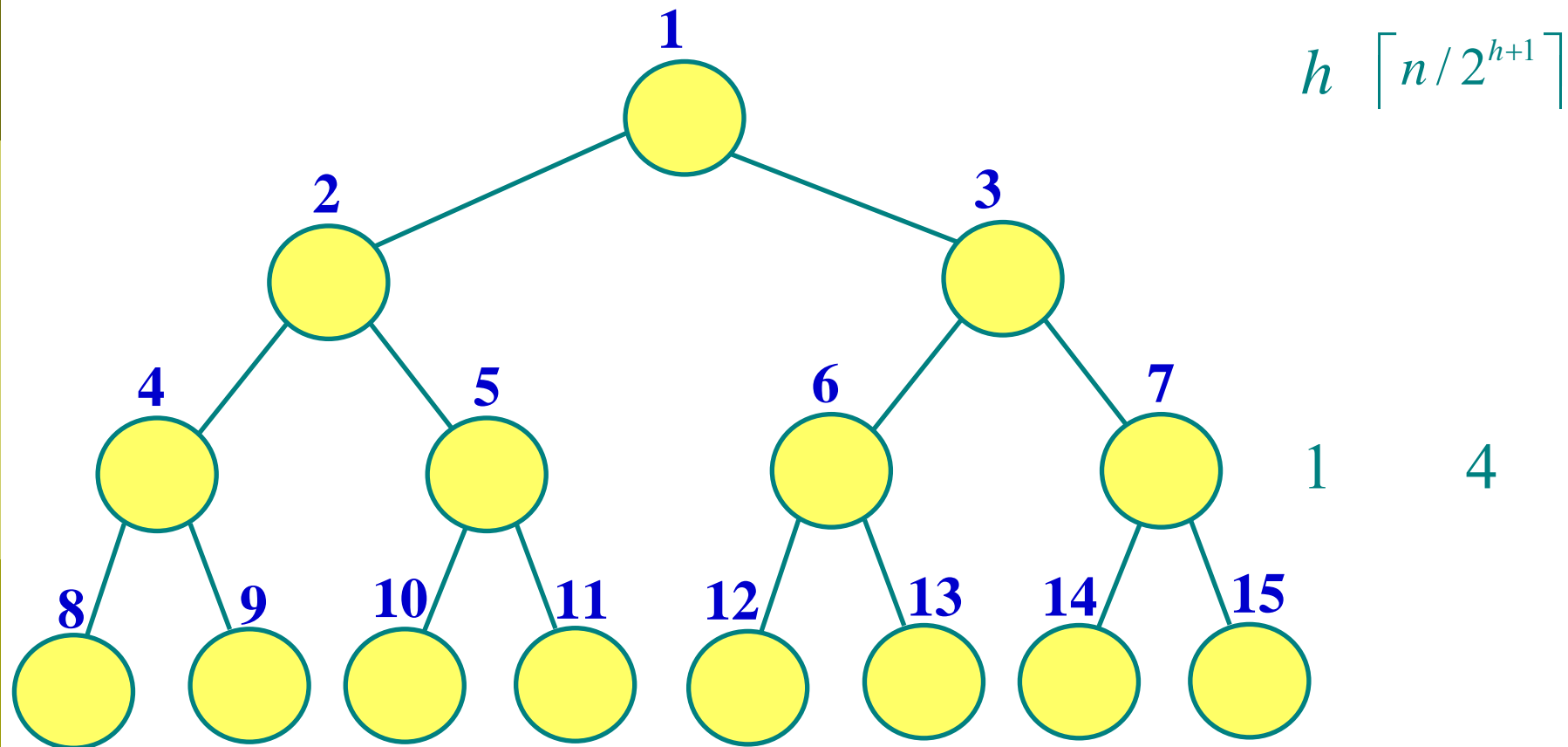
BUILD-MAX-HEAP(*A*)

1. *heap-size*[*A*] \leftarrow *length*[*A*]
2. **for** *i* $\leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(*A*, *i*)

$$O(n \lg n)$$

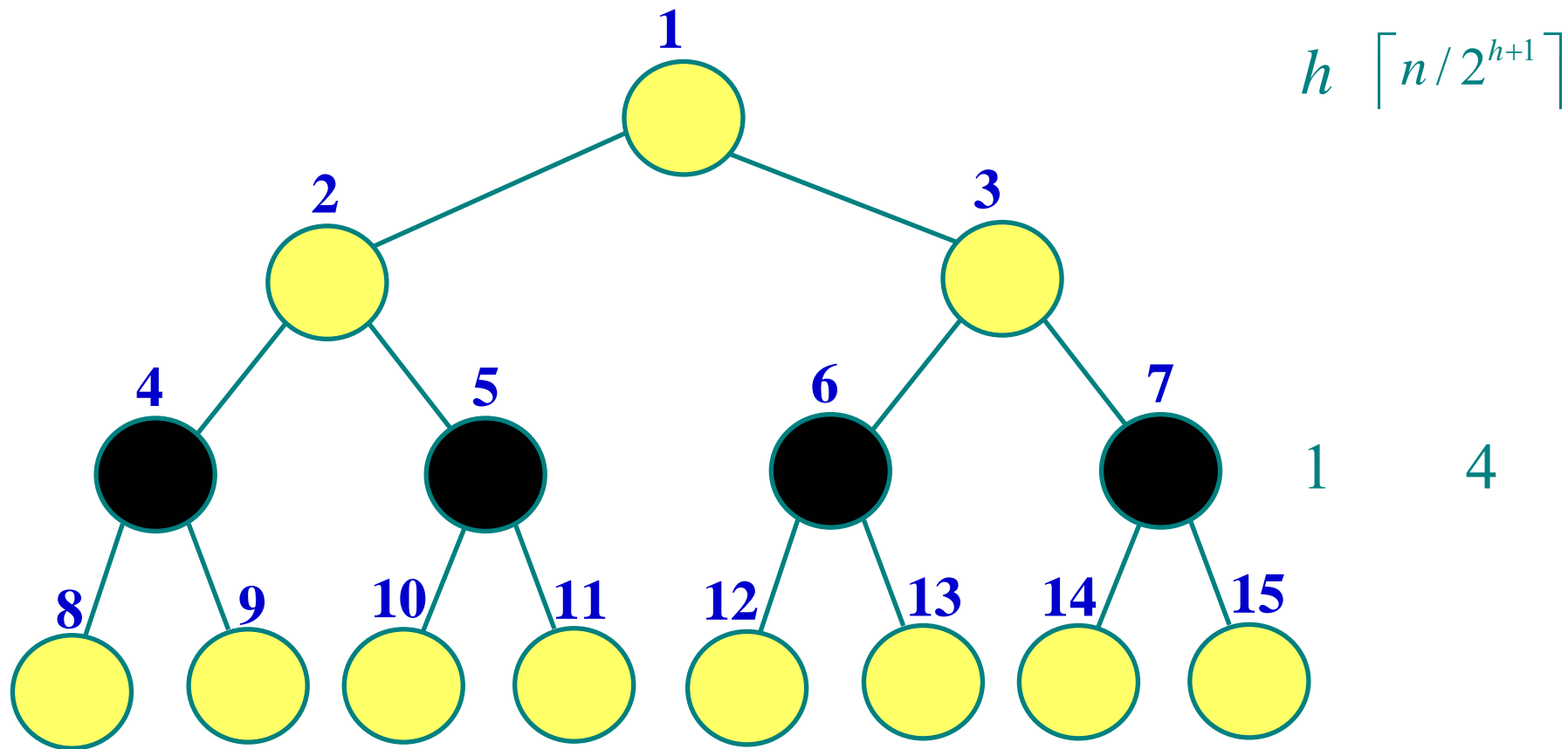
This is *upper bound* and is not asymptotically *tight*.

Nodes of any height



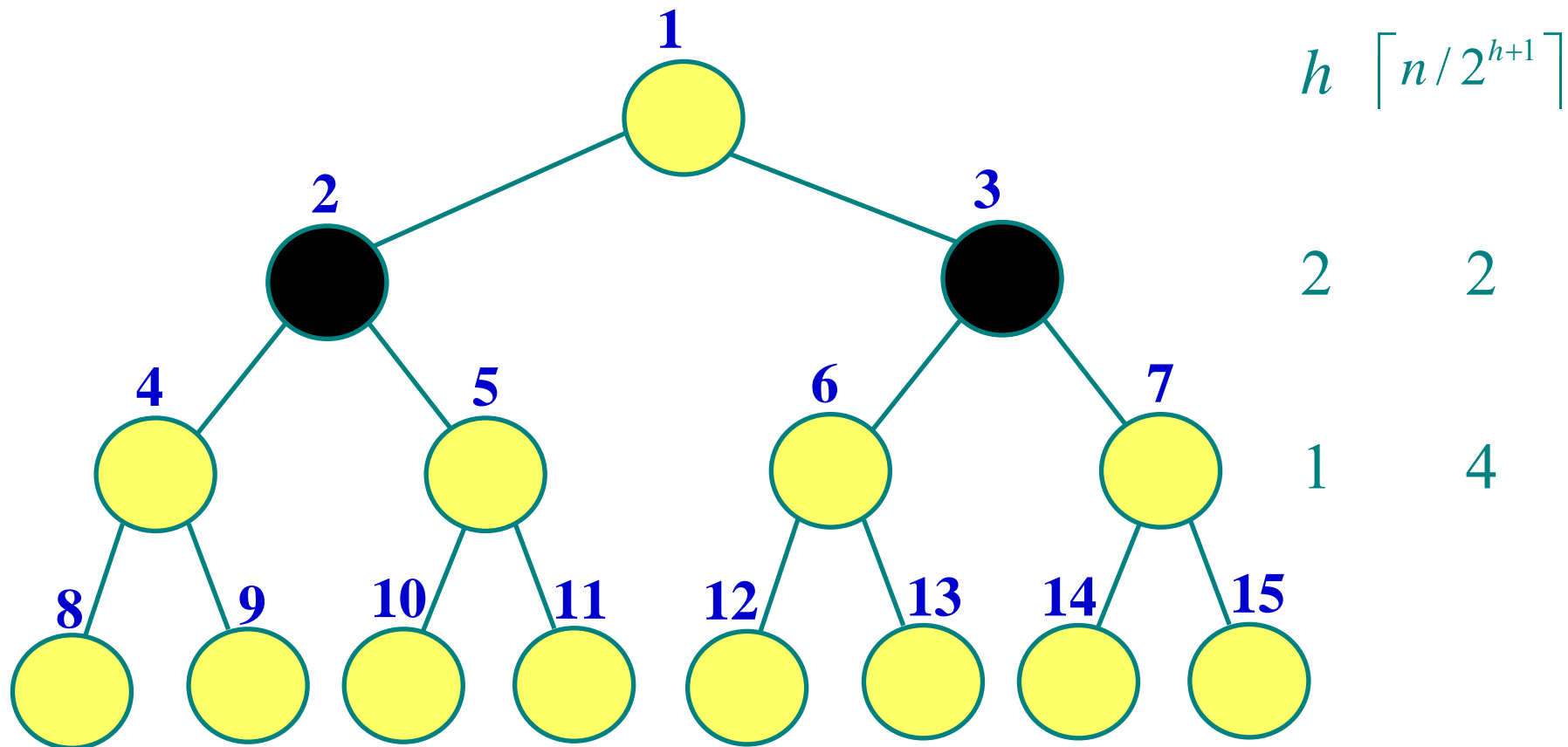
n -element heap has height $\lfloor \lg n \rfloor$ and at most $\left\lceil n / 2^{h+1} \right\rceil$ nodes of any height h

Nodes of any height



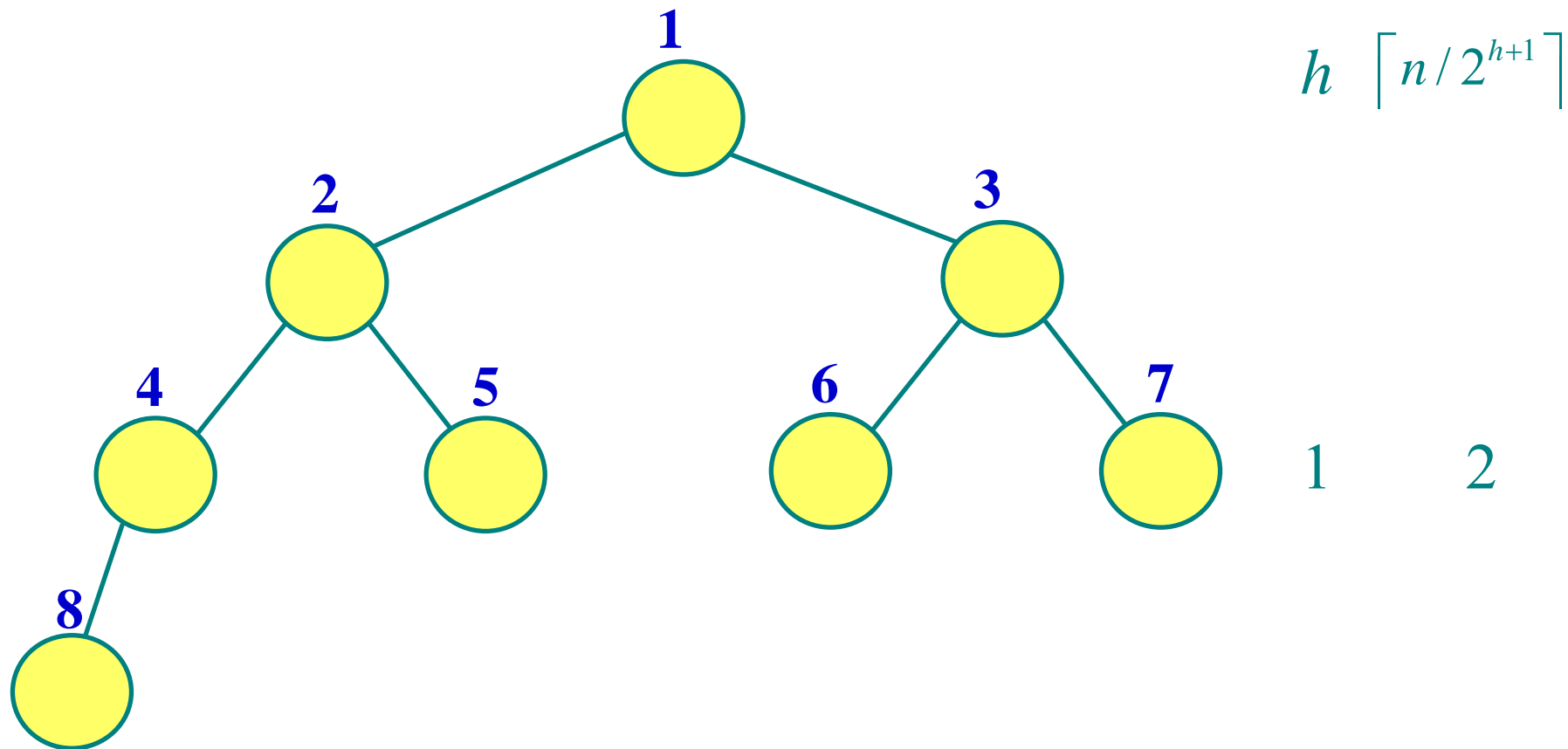
n -element heap has height $\lfloor \lg n \rfloor$ and at most $\left\lceil n / 2^{h+1} \right\rceil$ nodes of any height h

Nodes of any height



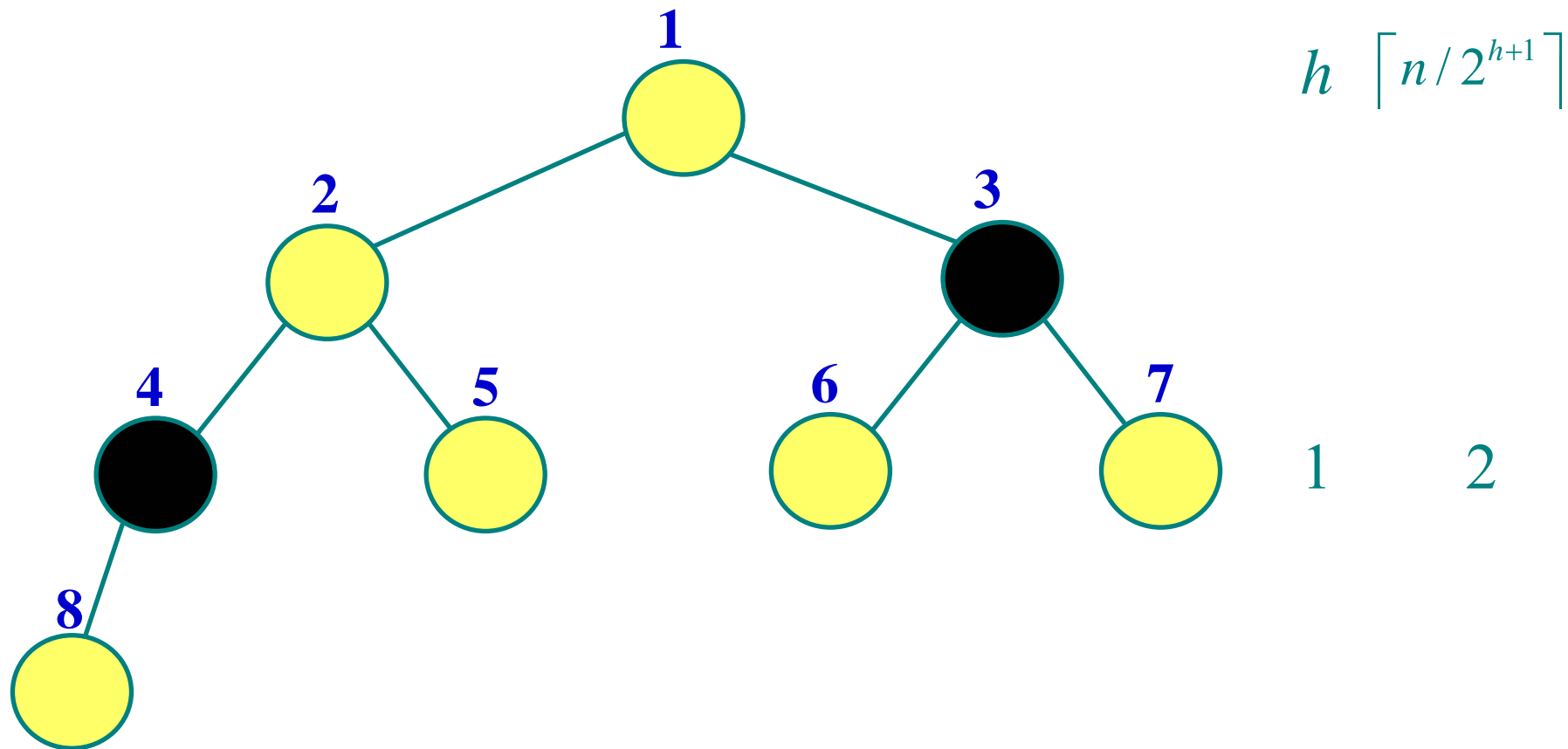
n -element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h

Nodes of any height



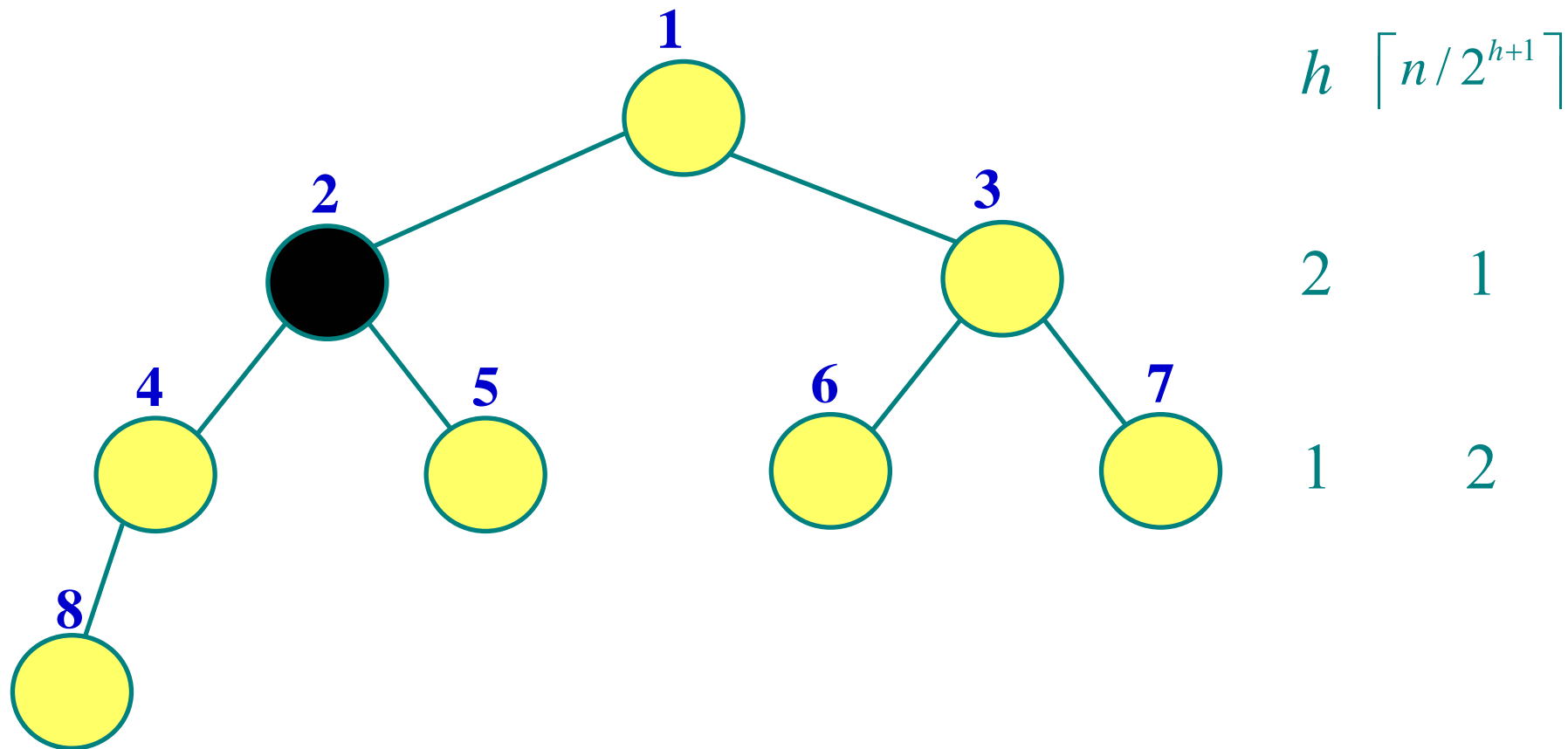
n -element heap has height $\lfloor \lg n \rfloor$ and at most $\left\lceil n / 2^{h+1} \right\rceil$ nodes of any height h

Nodes of any height



n -element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h

Nodes of any height



n -element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h

Analysis of building a heap

n -element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h

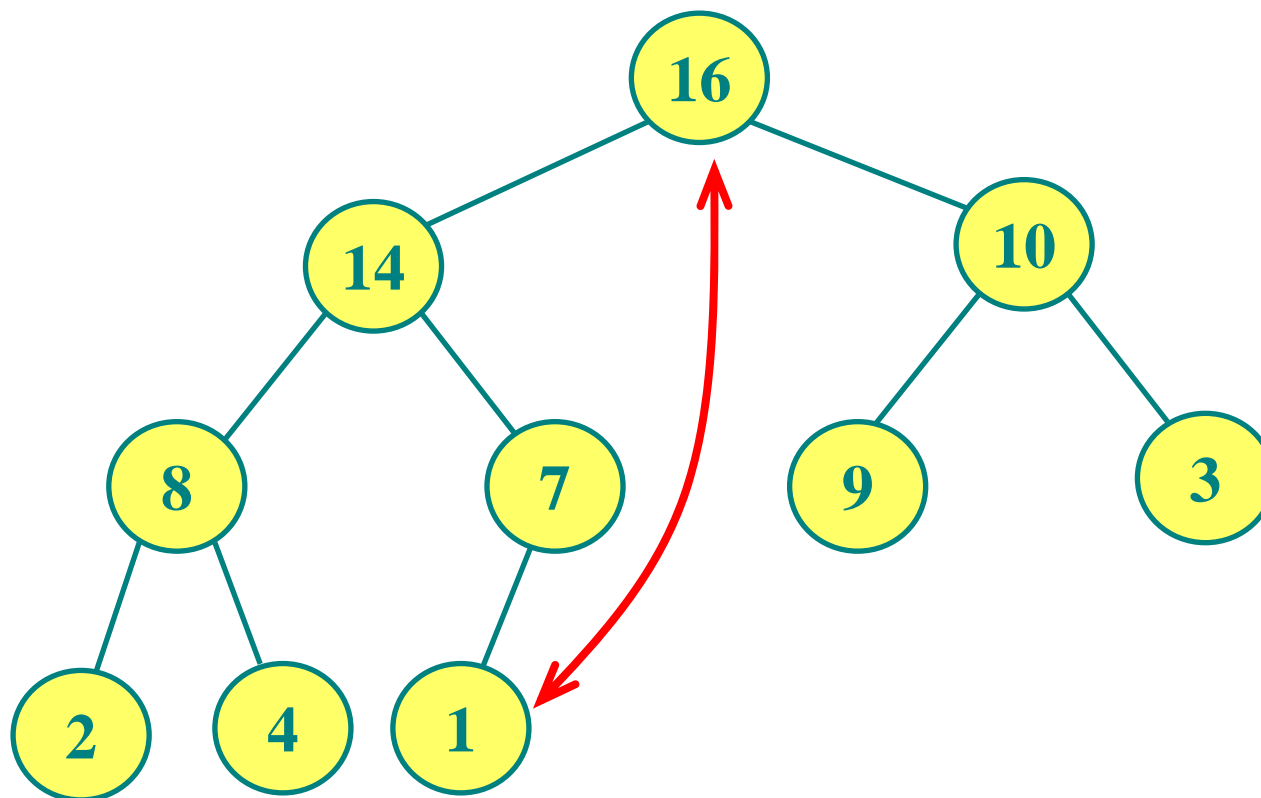
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$< O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \quad \left(\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2\right)$$

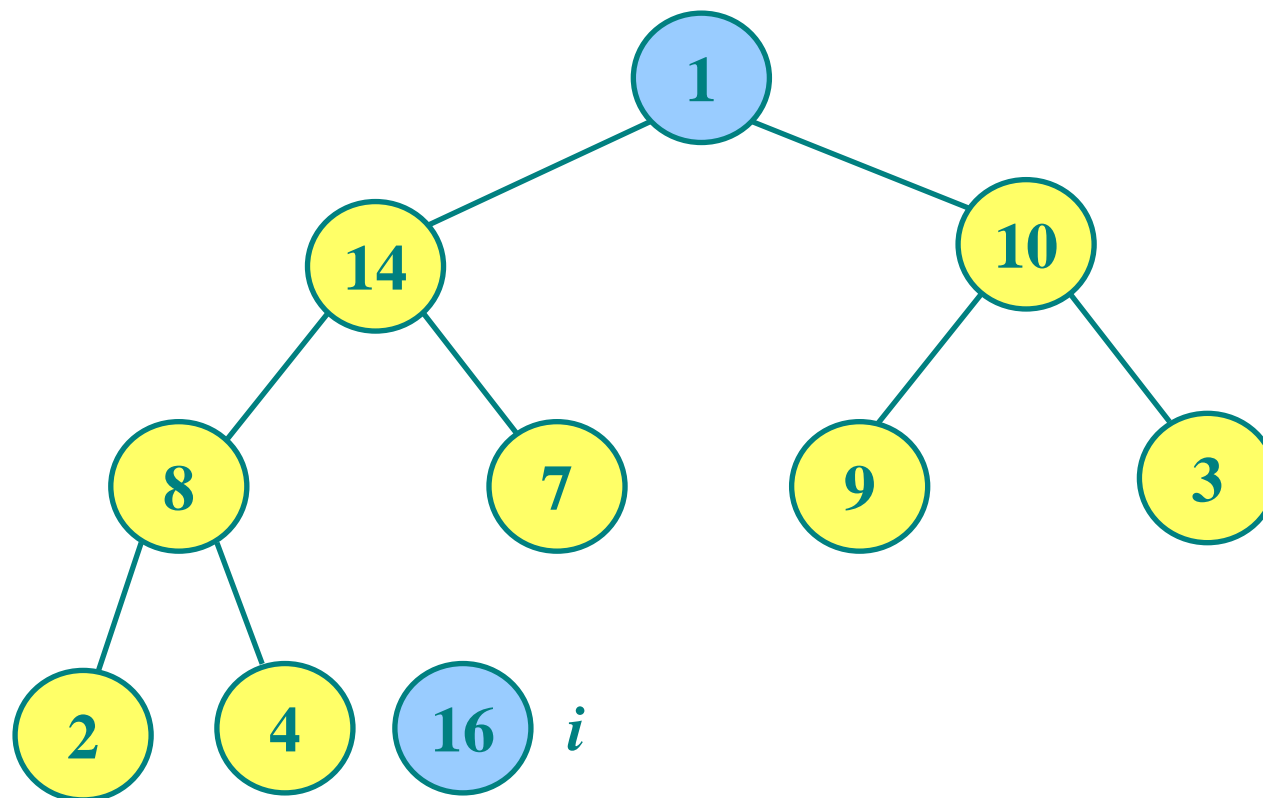
$$= O(n)$$



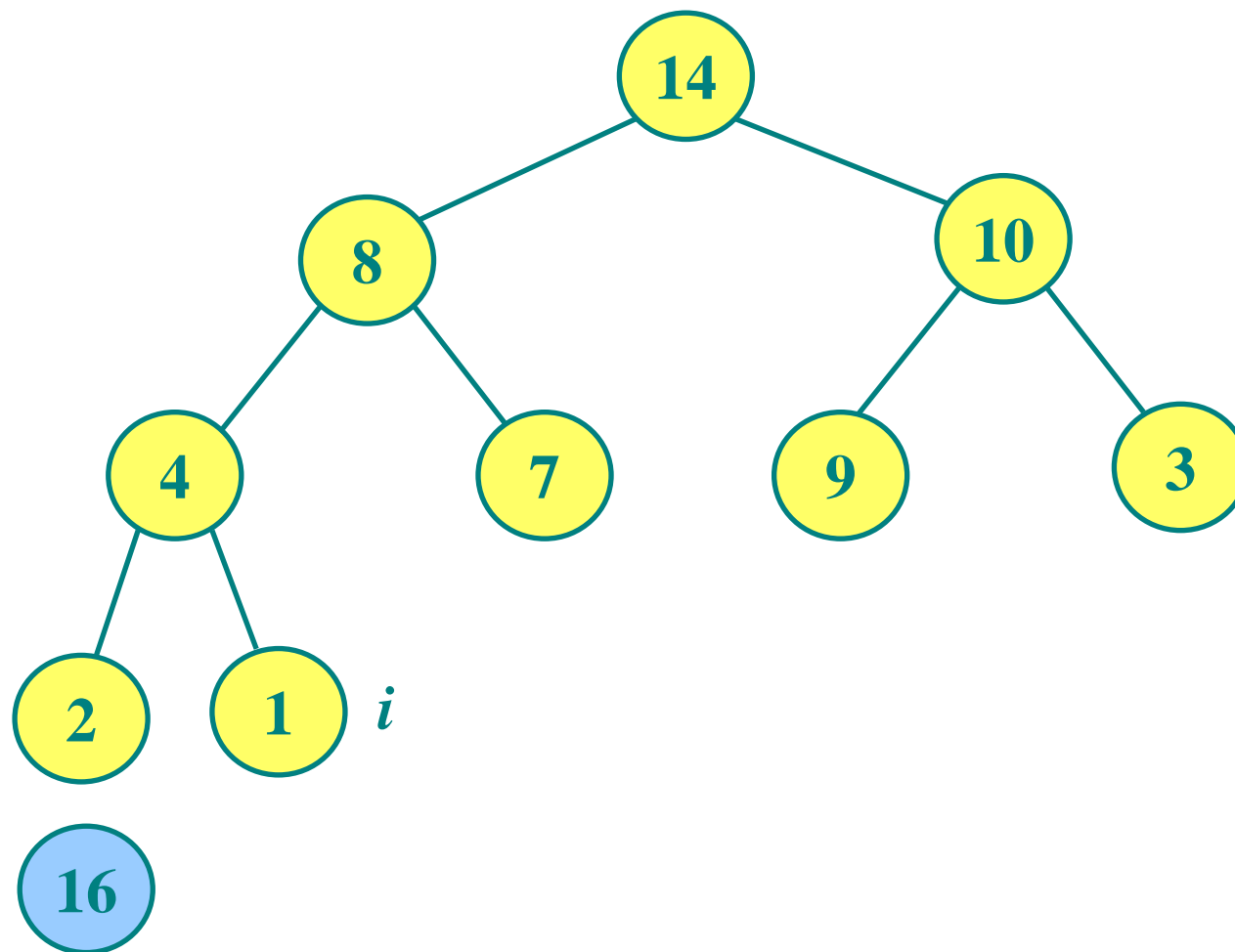
Heap sort



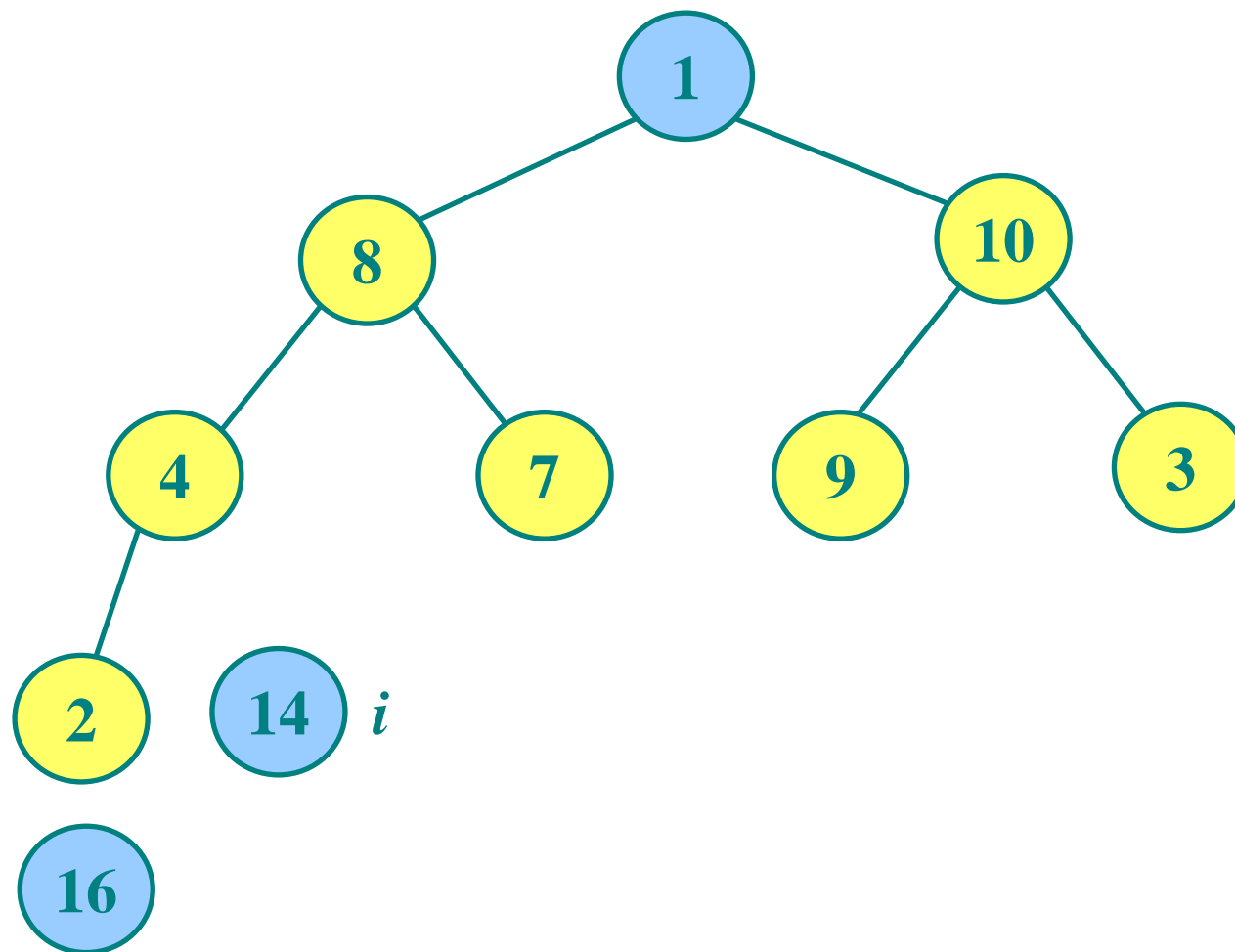
Heap sort



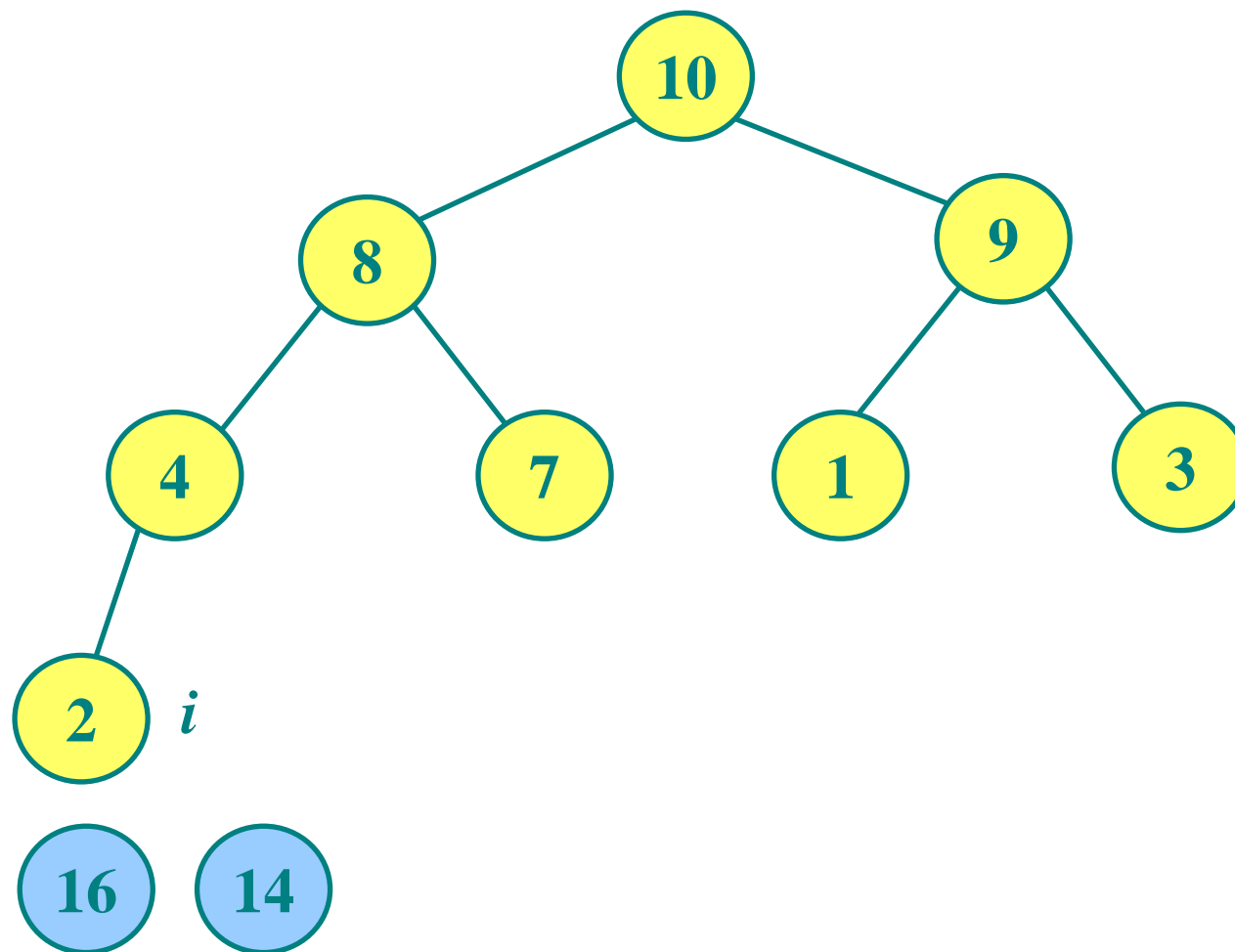
Heap sort



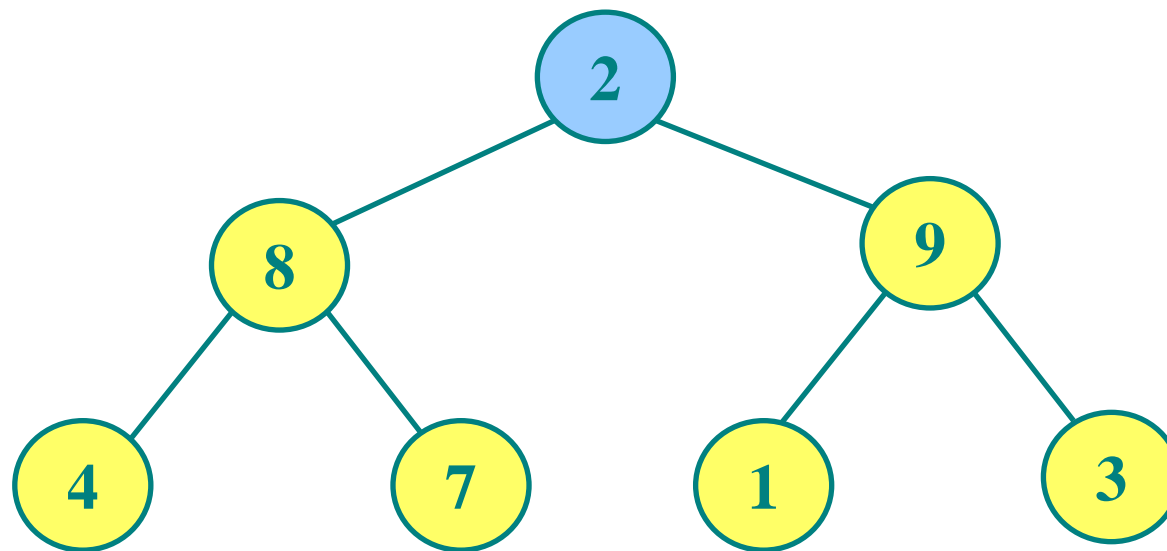
Heap sort



Heap sort



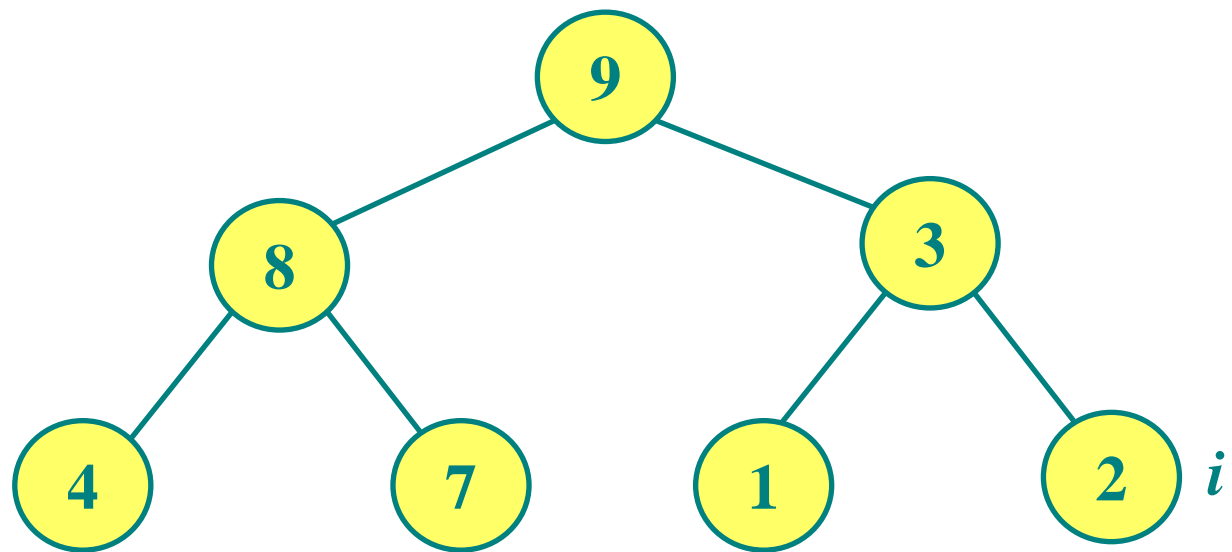
Heap sort



10 i

16 14

Heap sort



Heap sort

HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. MAX-HEAPIFY(*A*, 1)

$O(n \lg n)$

Priority queues

*A **priority queue** is a data structure for maintaining a set of S of elements, each with an associated value called a **key**.*

A max-priority queue supports the following operations.

- $\text{INSERT}(S, x)$ inserts the element x into the set S .
- $\text{MAXIMUM}(S)$ returns the element S with the largest key.
- $\text{EXTRACT-MAX}(S)$ removes and returns the element S with the largest key.
- $\text{INCREASE-KEY}(S, x, k)$ increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

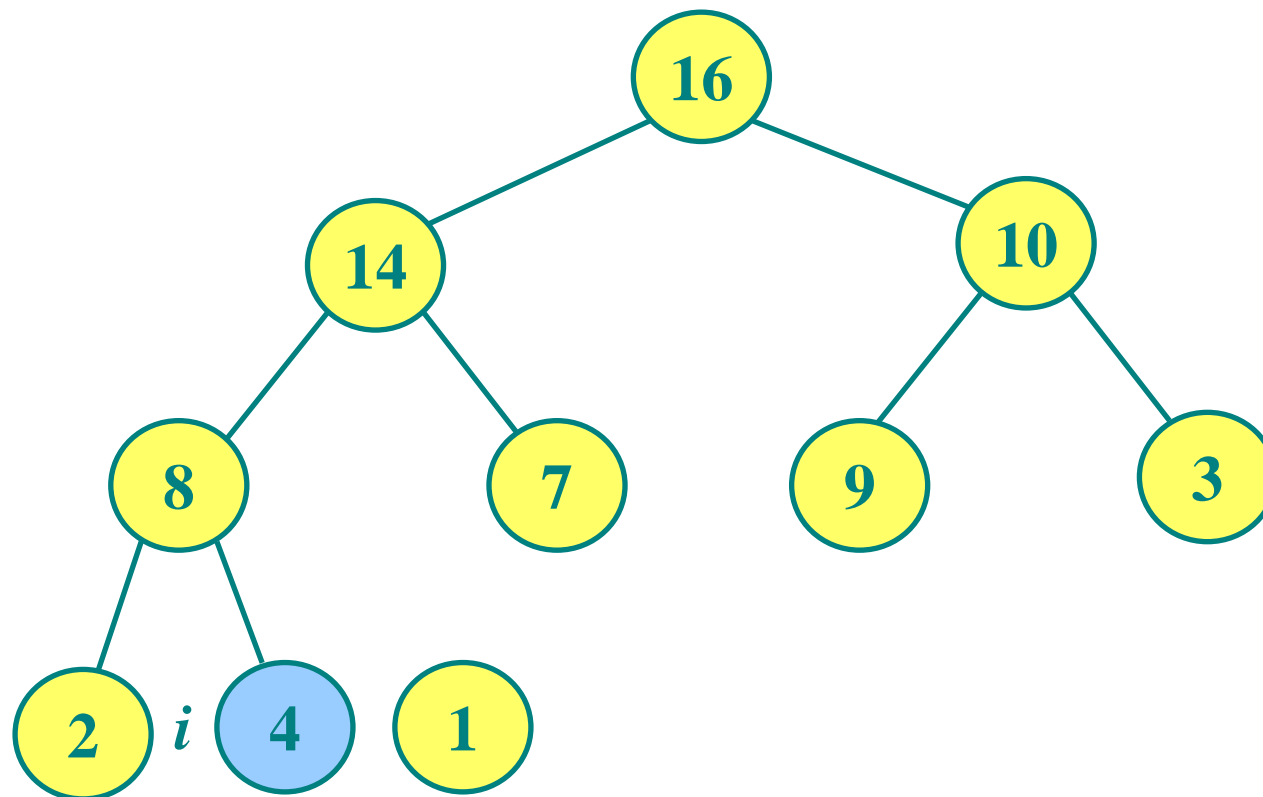
Operation of priority queues

HEAP-EXTRACT-MAX(A)

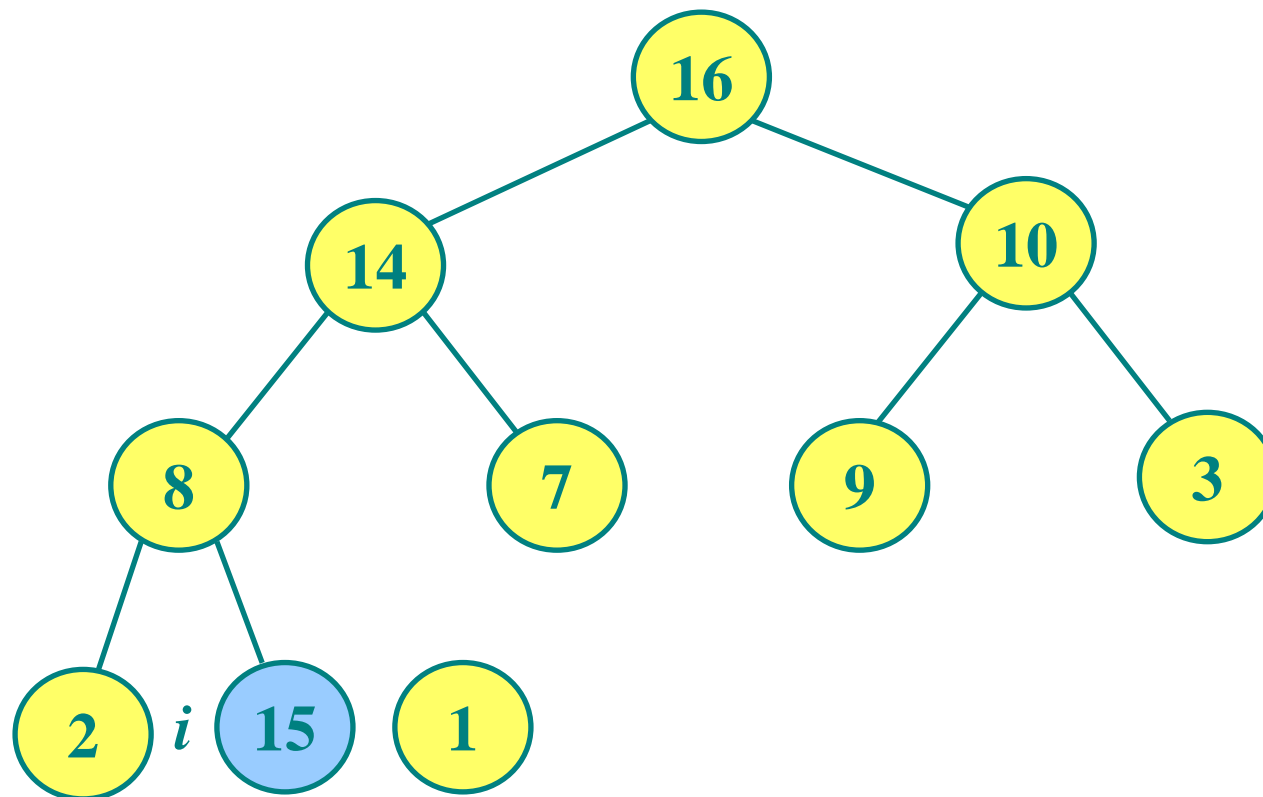
1. **if** $heap\text{-}size[A] < 1$
2. **then error** "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[heap\text{-}size[A]]$
5. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
6. MAX-HEAPIFY($A, 1$)
7. **return** max

$O(\lg n)$

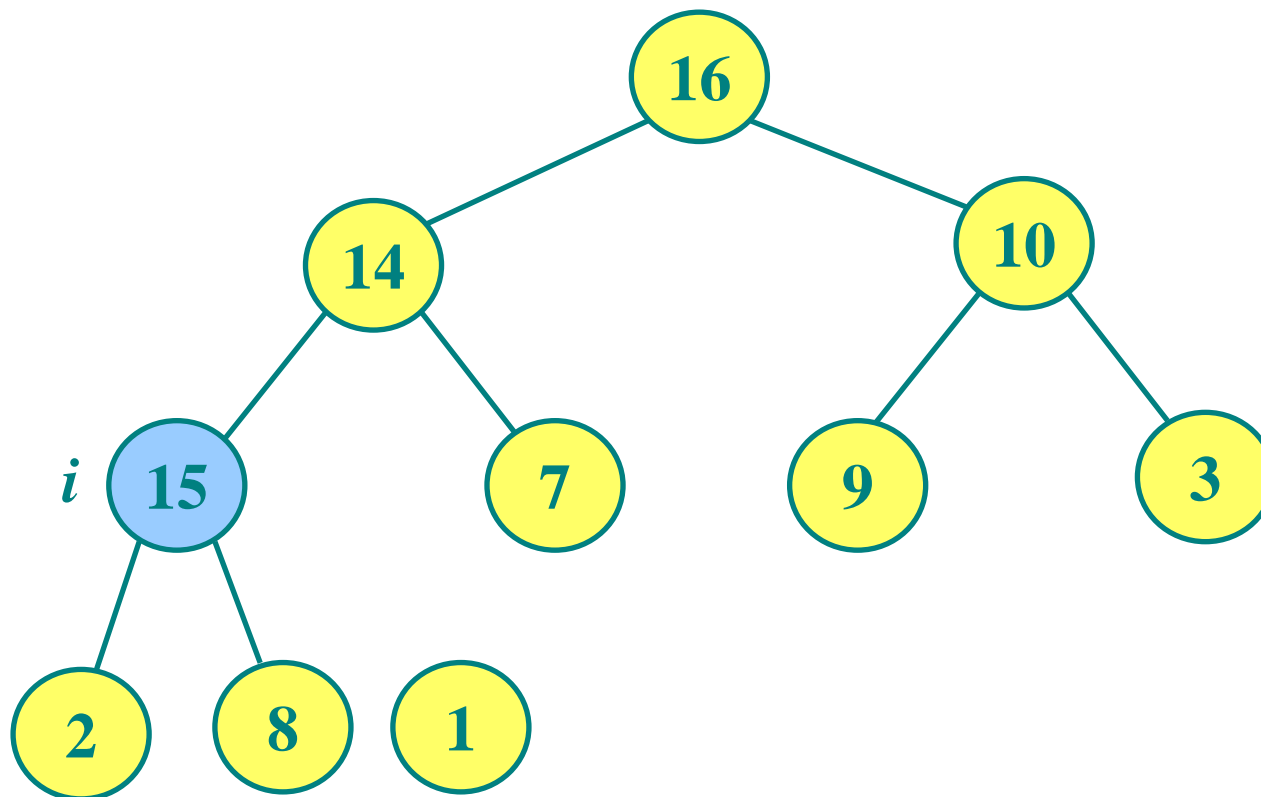
Heap-increase-key



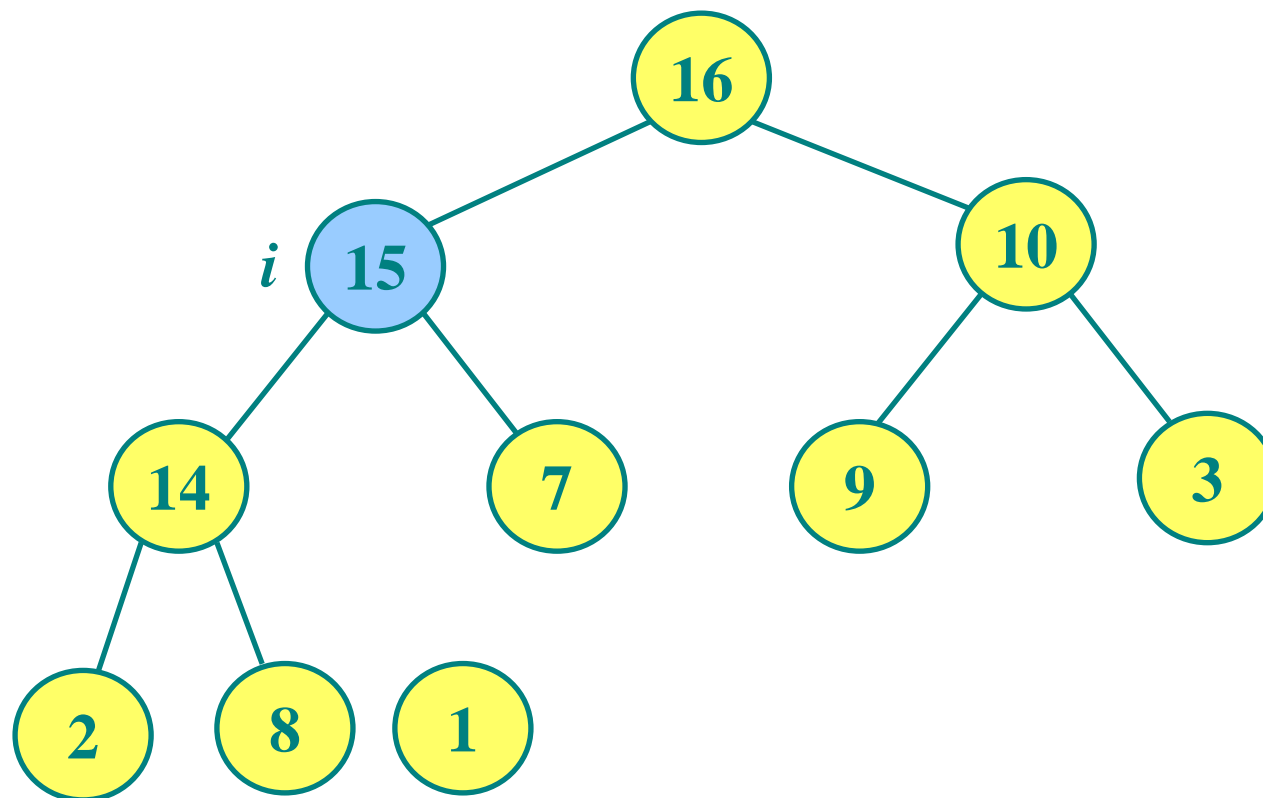
Heap-increase-key



Heap-increase-key



Heap-increase-key



done

Operation of priority queues

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

1. **if** $key < A[i]$
2. **then error** "new key is smaller than current key"
3. $A[i] \leftarrow key$
4. **while** $i > 1$ **and** $A[\text{PARENT}(i)] < A[i]$
5. do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
6. $i \leftarrow \text{PARENT}(i)$

$O(\lg n)$

Operation of priority queues

MAX-HEAP-INSERT(A, key)

1. $heap-size[A] \leftarrow heap-size[A] + 1$
2. $A[heap-size[A]] \leftarrow -\infty$
3. **HEAP-INCREASE-KEY**($A, heap-size[A], key$)

$O(\lg n)$

Sort algorithm

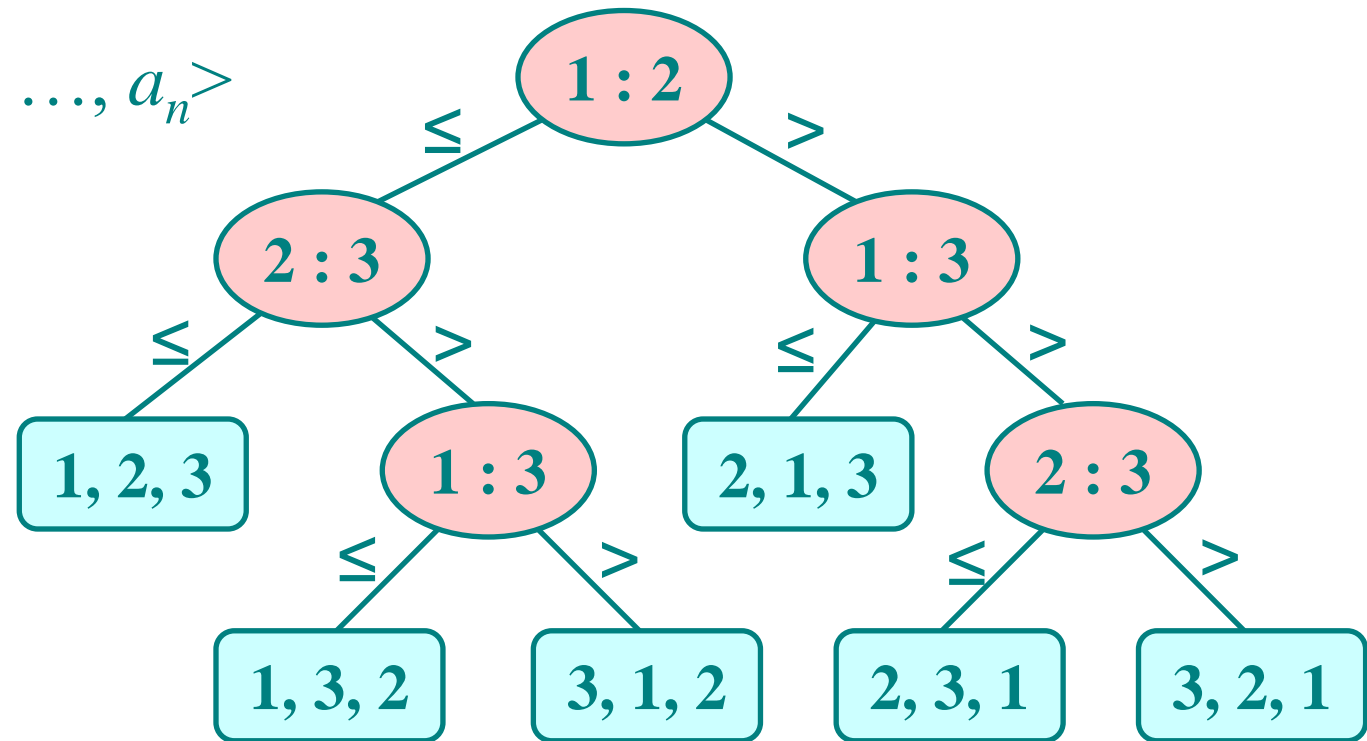
| Running time | Worst-case | Average-case | In place |
|----------------|------------|--------------|----------|
| Heap sort | $n \lg n$ | $n \lg n$ | Yes |
| Quick sort | n^2 | $n \lg n$ | Yes |
| Insertion sort | n^2 | n^2 | Yes |
| Merge sort | $n \lg n$ | $n \lg n$ | No |

Comparison sort

- All of our algorithms used *comparisons*.
- All of our algorithms have the running time $\Omega(n \lg n)$.
- Is it the best that we can do using just comparisons?

Decision-tree example

Sort $\langle a_1, a_2, \dots, a_n \rangle$
($n = 3$)

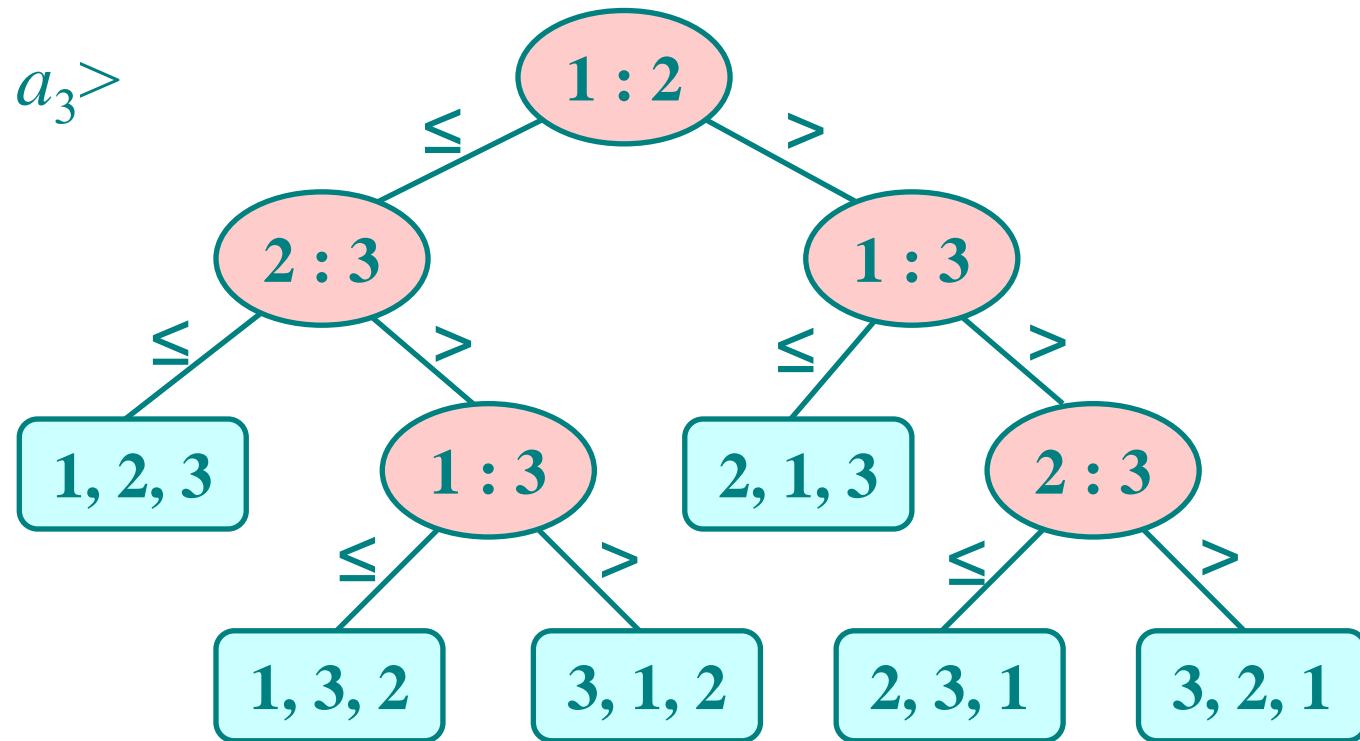


Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 6, 8, 5 \rangle$

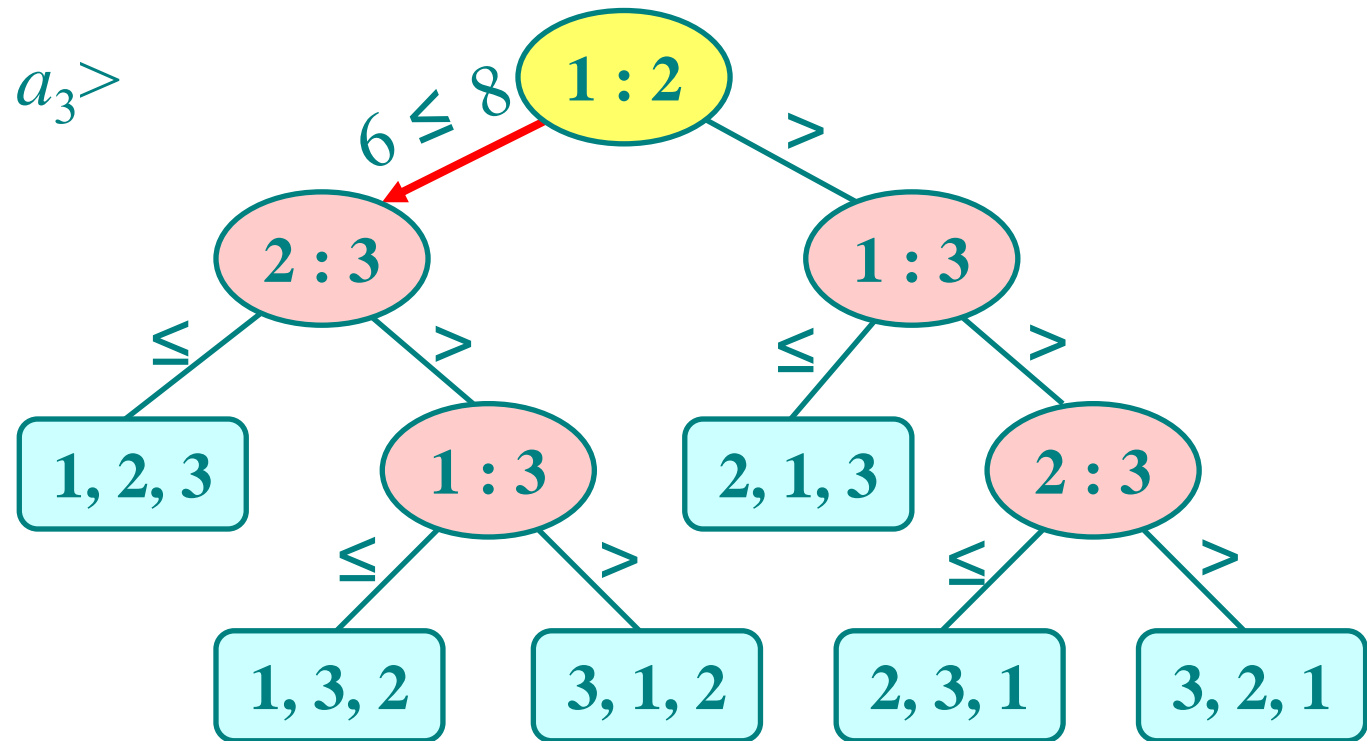


Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 6, 8, 5 \rangle$

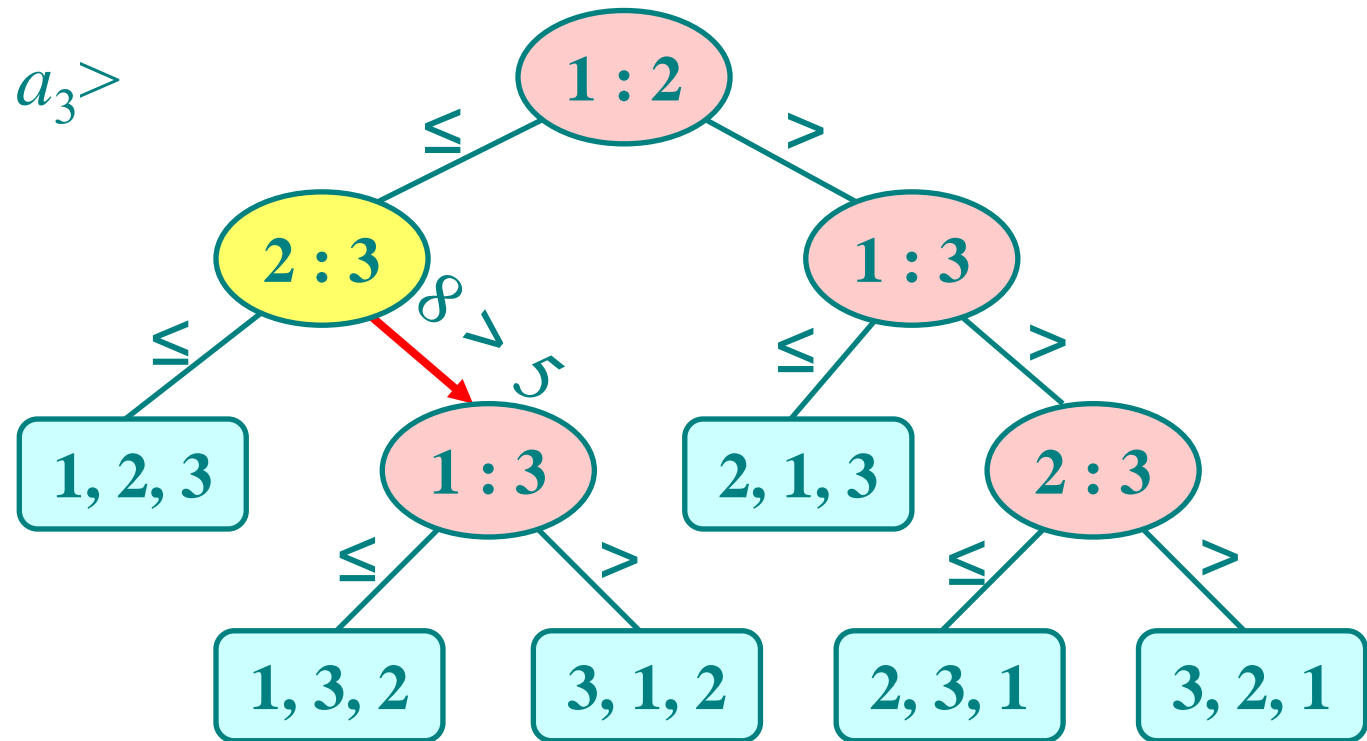


Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 6, 8, 5 \rangle$

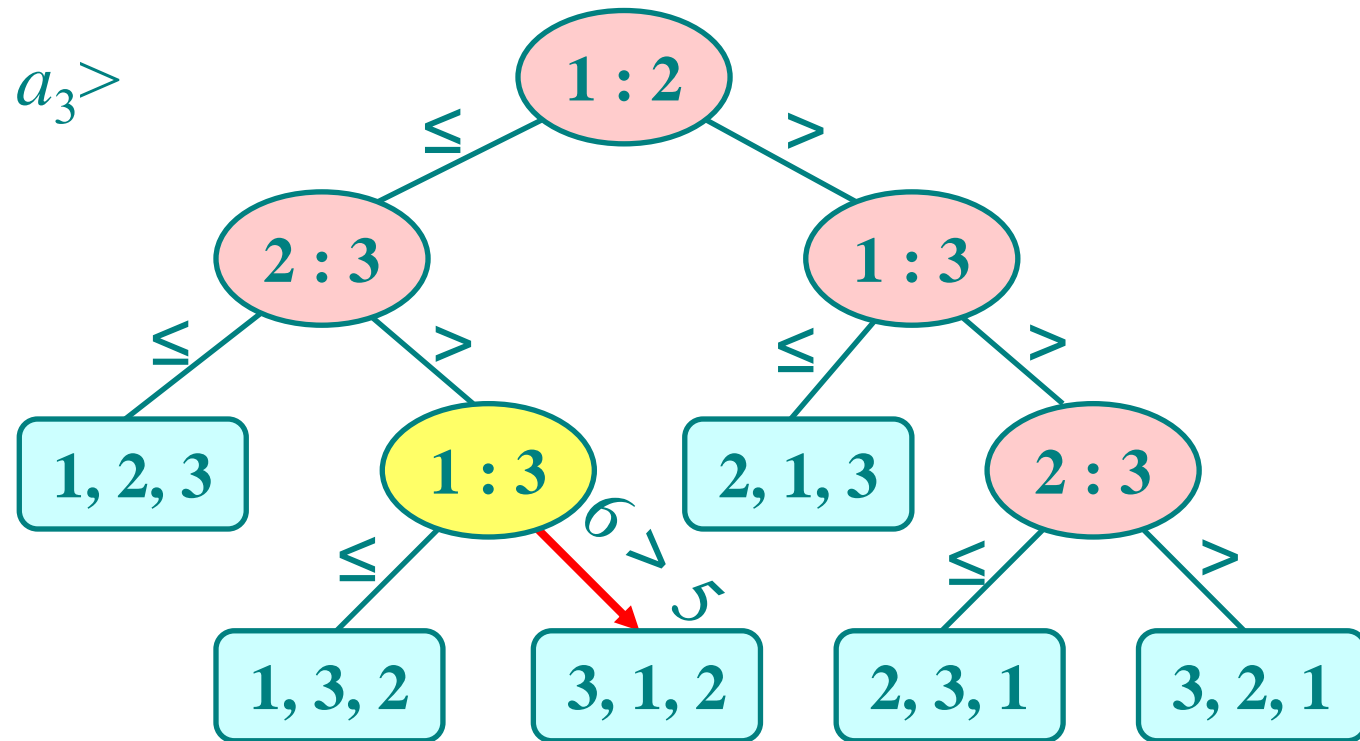


Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 6, 8, 5 \rangle$

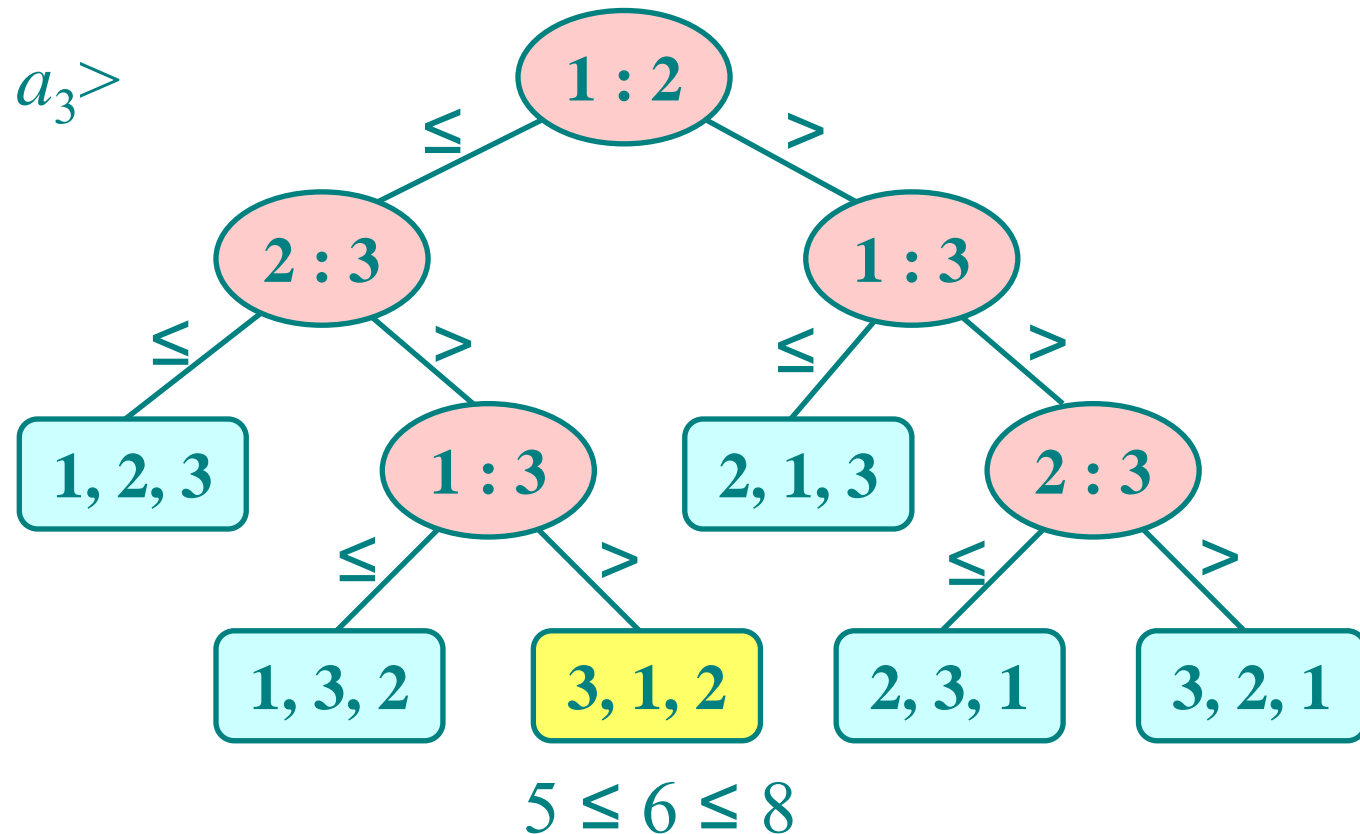


Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 6, 8, 5 \rangle$



Each leaf contains a permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ to indicate that the ordering $\langle a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)} \rangle$ has been established.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n .
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The number of comparisons done by the algorithm on a given input = the length of the path taken.
- Worst-case number of comparisons = max path length = height of tree.

Lower bound for decision tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Corollary. Any comparison sorting algorithm has worst-case running time $\Omega(n \lg n)$.

Corollary 2. Merge sort and Heap sort are asymptotically optimal comparison sorting algorithms.

Lower bound for decision tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof.

- The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations
- A height- h binary tree has $\leq 2^h$ leaves
- Thus, $2^h \geq \text{number of leaves} \geq n!$, or $h \geq \lg(n!)$

Proof

$$\begin{aligned}h &\geq \lg(n!) \\&= \lg(n(n-1)(n-2)\cdots(2)(1)) \\&= \lg n + \lg(n-1) + \lg(n-2) + \cdots + \lg 2 + \lg 1 \\&\geq \lg n + \lg(n-1) + \lg(n-2) + \cdots + \lg(n/2) \\&\geq \frac{n}{2} \lg \frac{n}{2} \\&= \frac{n}{2} (\lg n - \lg 2) \\&= \Omega(n \lg n) \quad \square\end{aligned}$$

Example: sorting 3 elements

Recall $h \geq \lg(n!)$

- $n = 3$
- $n! = 6$
- $\lg 6 = 2.58$
- Sorting 3 elements requires ≥ 3 comparisons in the worst case

Thinking

Is any sort algorithm the run in linear time?

Answer: Yes!

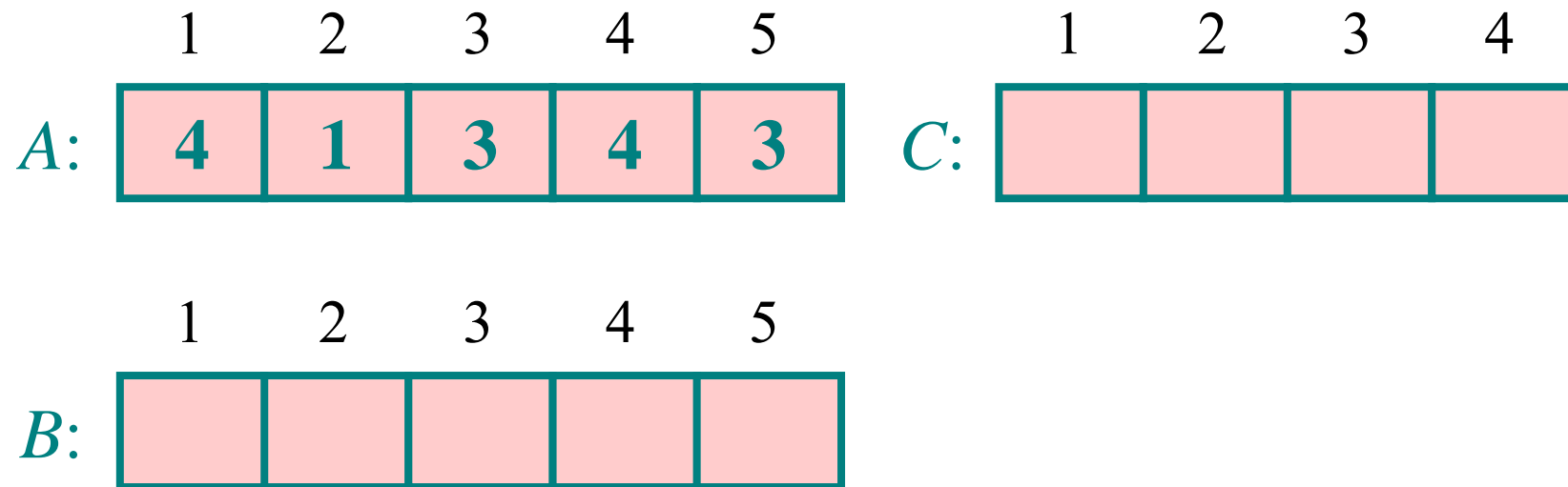
Needless to say, these algorithms use operations other than comparisons to determine the sorted order.

Sorting in linear time

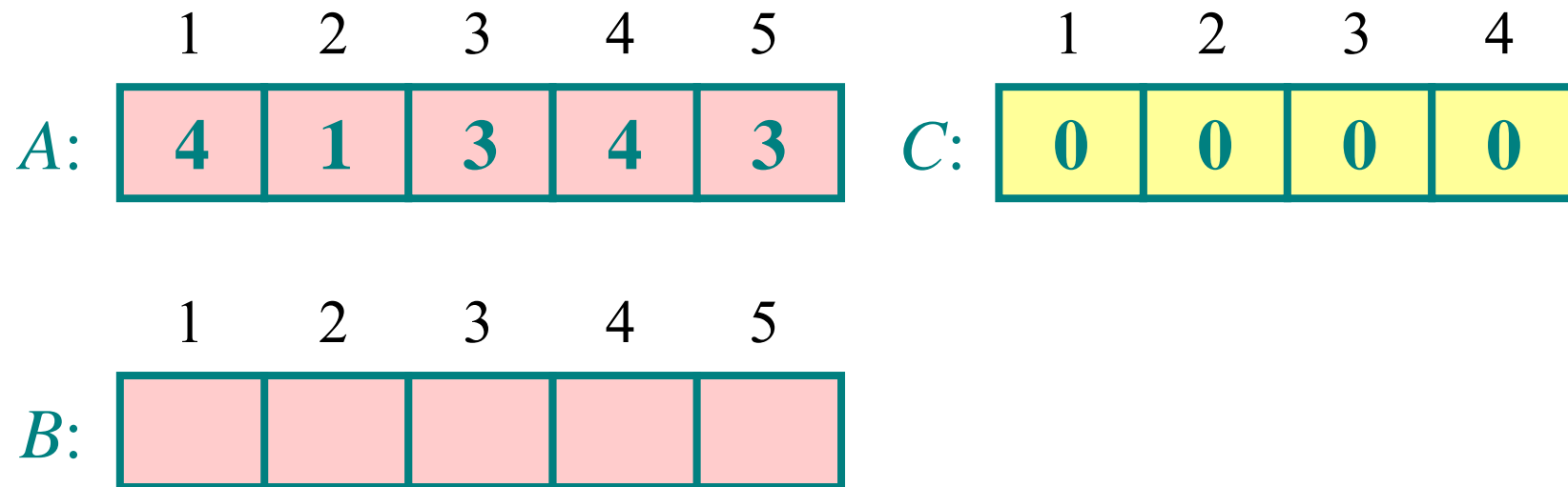
Counting sort: No comparisons between elements.

- **Input:** $A[1 \dots n]$, where $A[j] \in \{1, 2, \dots, k\}$.
- **Output:** $B[1 \dots n]$, sorted.
- **Auxiliary storage:** $C[1 \dots k]$.

Counting-sort example



Loop 1



1. **for** $i \leftarrow 1$ **to** k
2. **do** $C[i] \leftarrow 0$

Loop 2

| | | | | | | | | | | |
|------------|---|---|---|---|---|------------|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| <i>A</i> : | 4 | 1 | 3 | 4 | 3 | <i>C</i> : | 0 | 0 | 0 | 1 |
| | 1 | 2 | 3 | 4 | 5 | | | | | |
| <i>B</i> : | | | | | | | | | | |

3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 2

| | | | | | | | | | | |
|----|---|---|---|---|---|----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 0 | 1 |
| B: | | | | | | | | | | |

3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 2

| | | | | | | | | | | |
|----|---|---|---|---|---|----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 1 | 1 |
| B: | | | | | | | | | | |

3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 2

| | | | | | | | | | | |
|----|---|---|---|---|---|----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 1 | 2 |
| B: | | | | | | | | | | |

3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 2

| | | | | | | | | | | |
|----|---|---|---|---|---|----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 2 | 2 |
| B: | | | | | | | | | | |

3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 3

| | | | | | | | | | | |
|----|---|---|---|---|---|-----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 2 | 2 |
| B: | | | | | | C': | 1 | 0 | 2 | 2 |

5. **for** $i \leftarrow 2$ **to** k

6. **do** $C[i] \leftarrow C[i] + C[i-1]$

Loop 3

| | | | | | | | | | | |
|----|---|---|---|---|---|-----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 2 | 2 |
| B: | | | | | | C': | 1 | 1 | 2 | 2 |

5. **for** $i \leftarrow 2$ **to** k

6. **do** $C[i] \leftarrow C[i] + C[i-1]$

Loop 3

| | | | | | | | | | | |
|----|---|---|---|---|---|-----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 2 | 2 |
| B: | | | | | | C': | 1 | 1 | 3 | 2 |

5. **for** $i \leftarrow 2$ **to** k

6. **do** $C[i] \leftarrow C[i] + C[i-1]$

Loop 3

| | | | | | | | | | | |
|----|---|---|---|---|---|-----|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 |
| A: | 4 | 1 | 3 | 4 | 3 | C: | 1 | 0 | 2 | 2 |
| B: | | | | | | C': | 1 | 1 | 3 | 5 |

5. **for** $i \leftarrow 2$ **to** k

6. **do** $C[i] \leftarrow C[i] + C[i-1]$

Loop 4

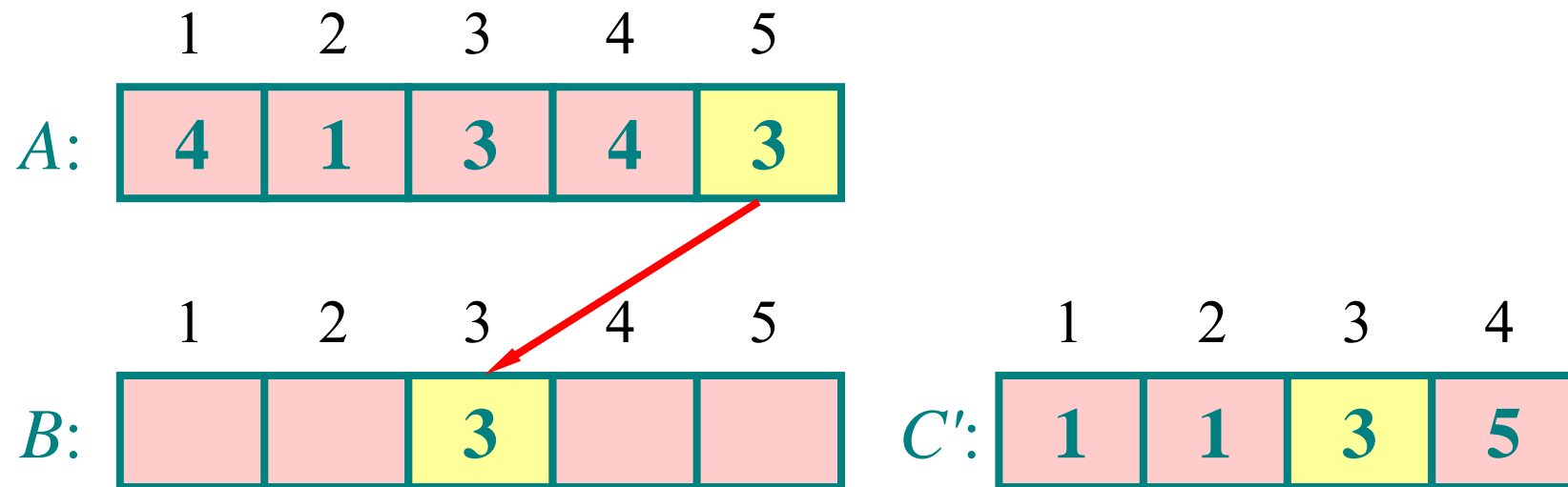
| | | | | | |
|----|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| A: | 4 | 1 | 3 | 4 | 3 |

| | | | | | |
|----|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| B: | | | | | |

| | | | | |
|-----|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| C': | 1 | 1 | 3 | 5 |

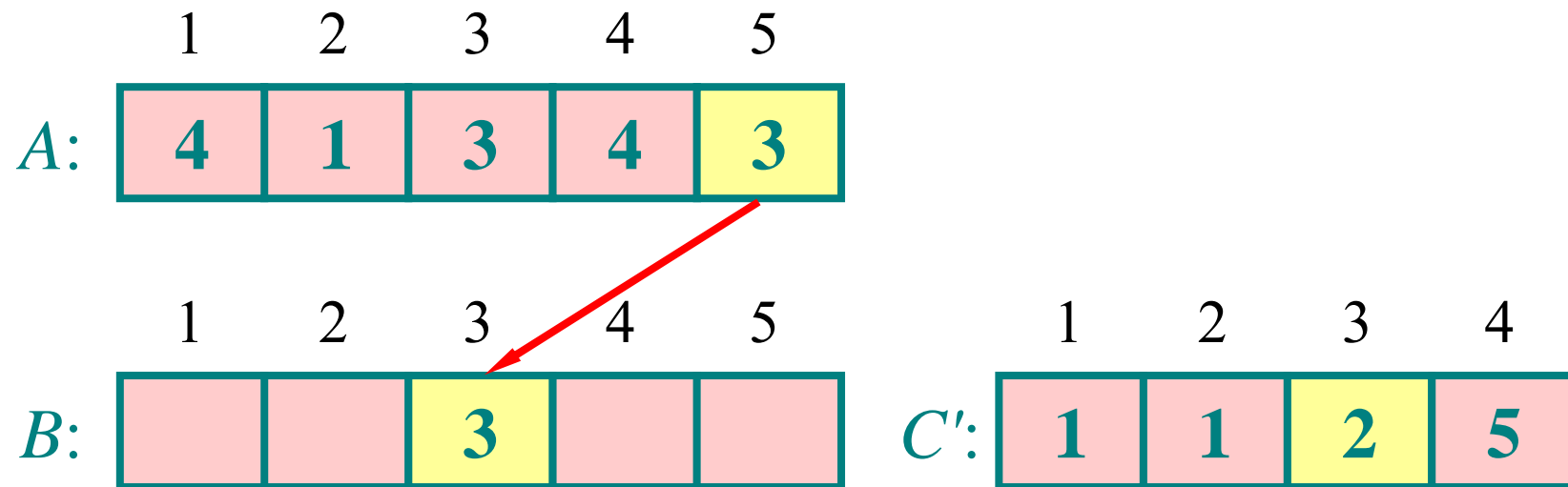
7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



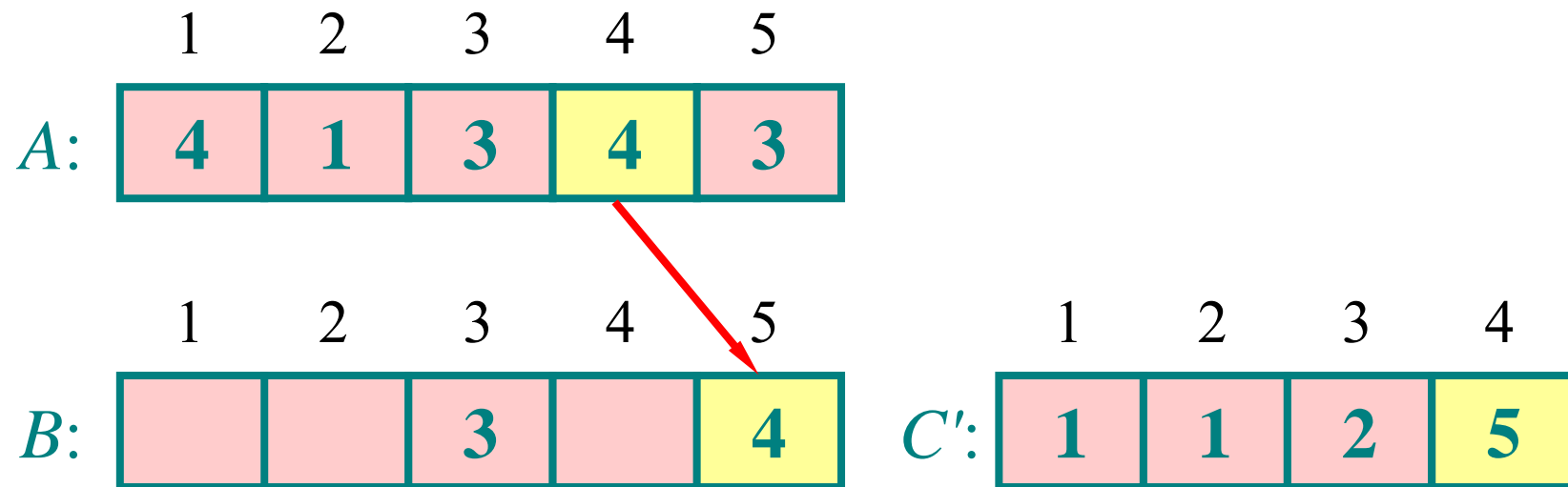
7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



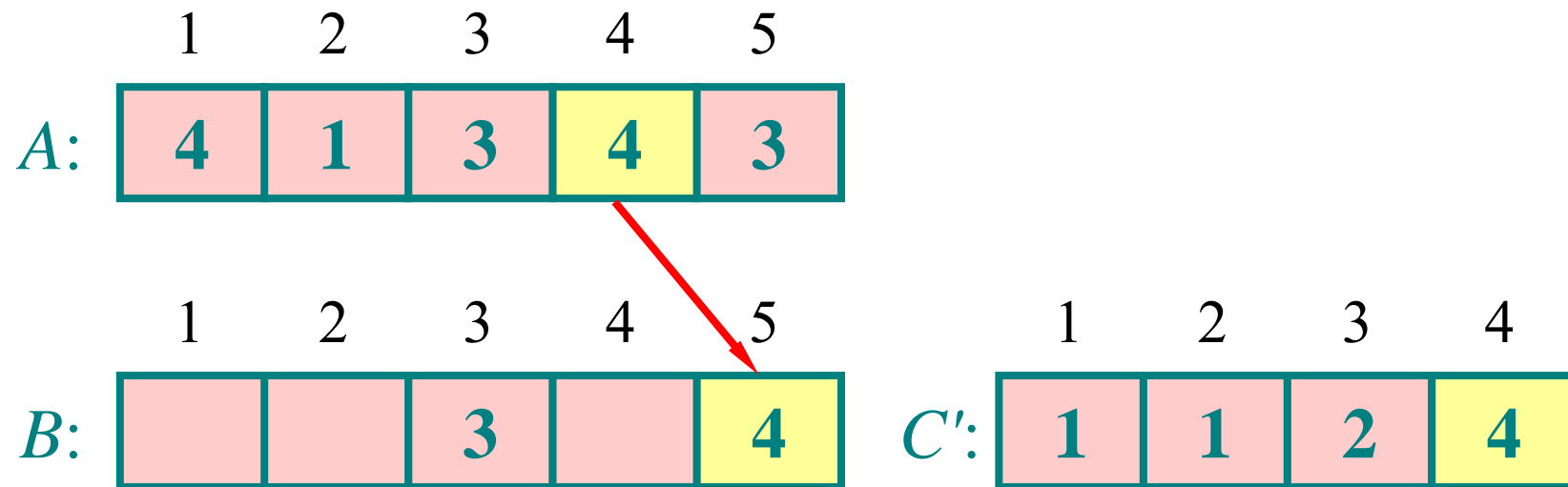
```
7. for  $j \leftarrow \text{length}[A]$  downto 1
8.   do  $B[C[A[j]]] \leftarrow A[j]$ 
9.      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```


Loop 4



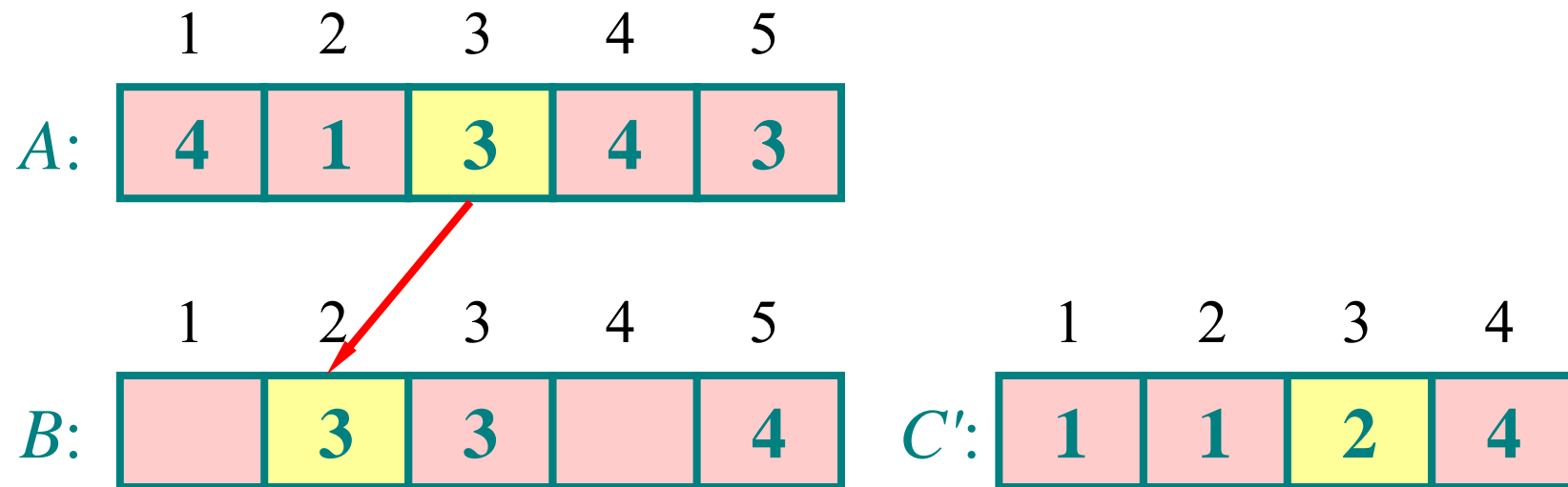
7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



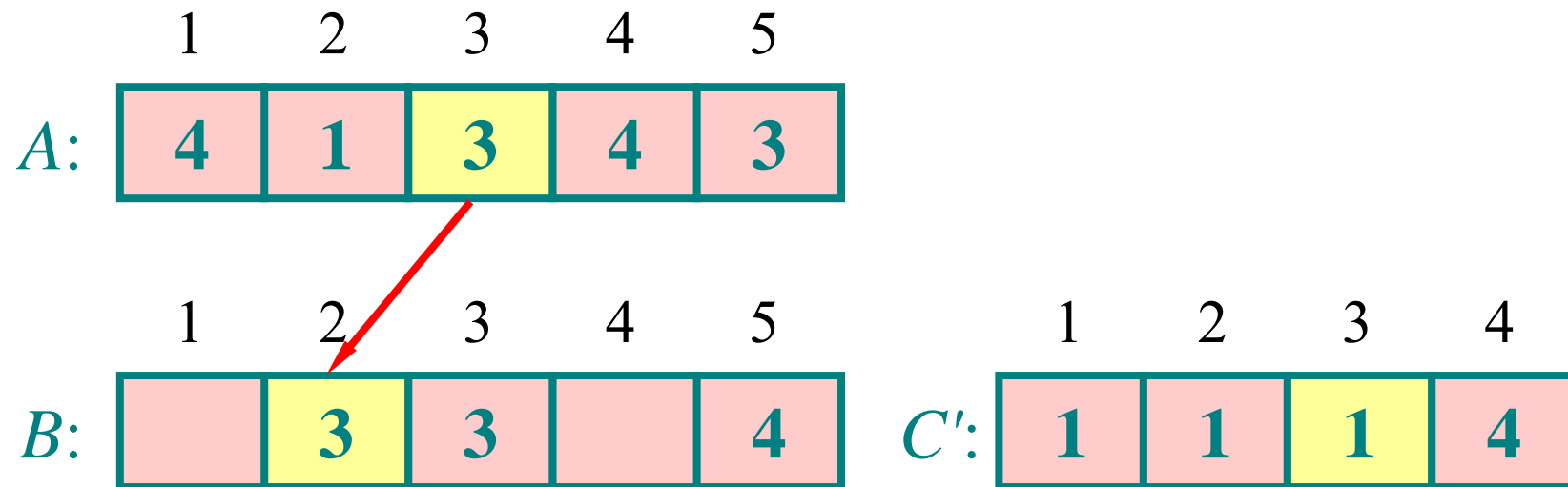
```
7. for  $j \leftarrow \text{length}[A]$  downto 1
8.   do  $B[C[A[j]]] \leftarrow A[j]$ 
9.      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

Loop 4



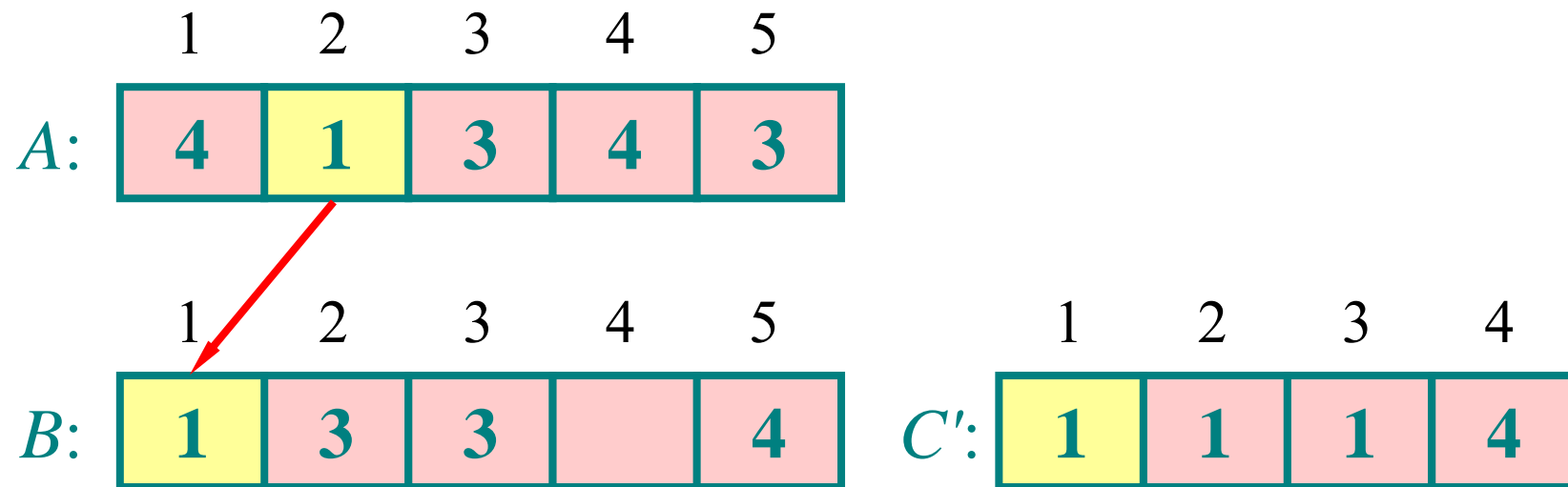
7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



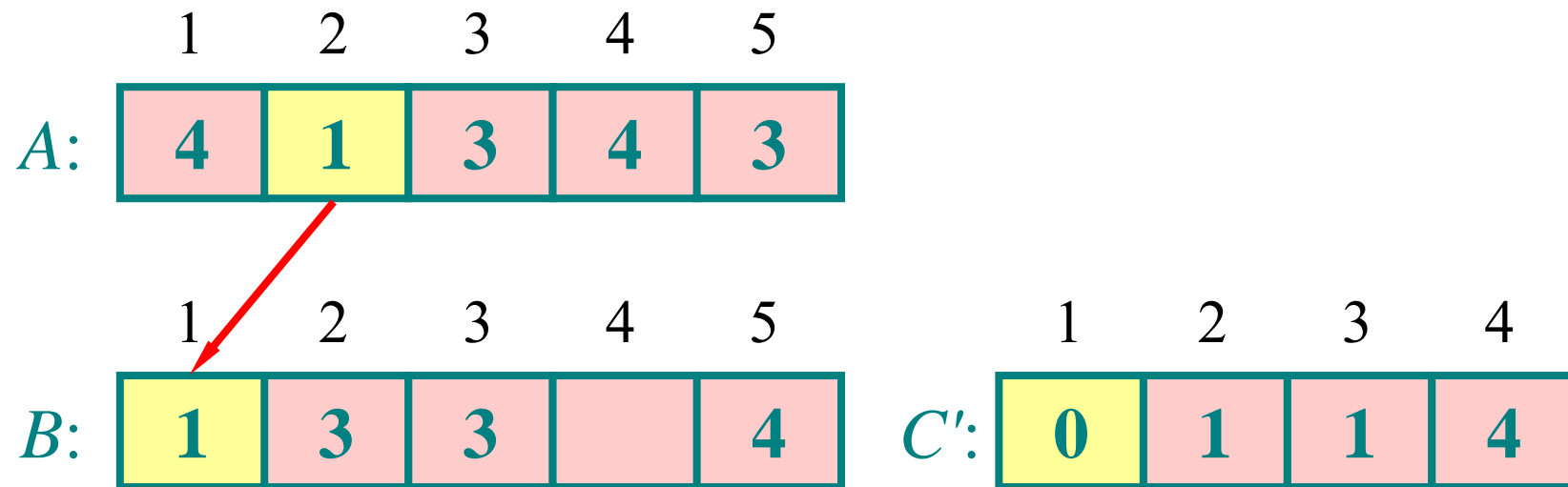
7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



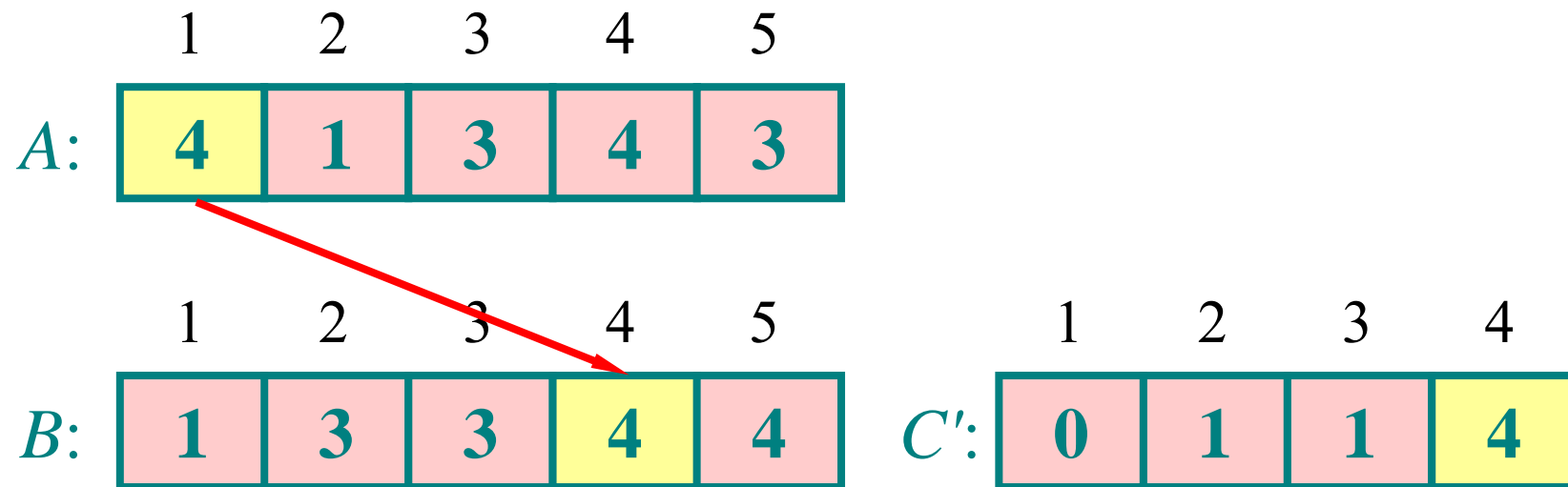
7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



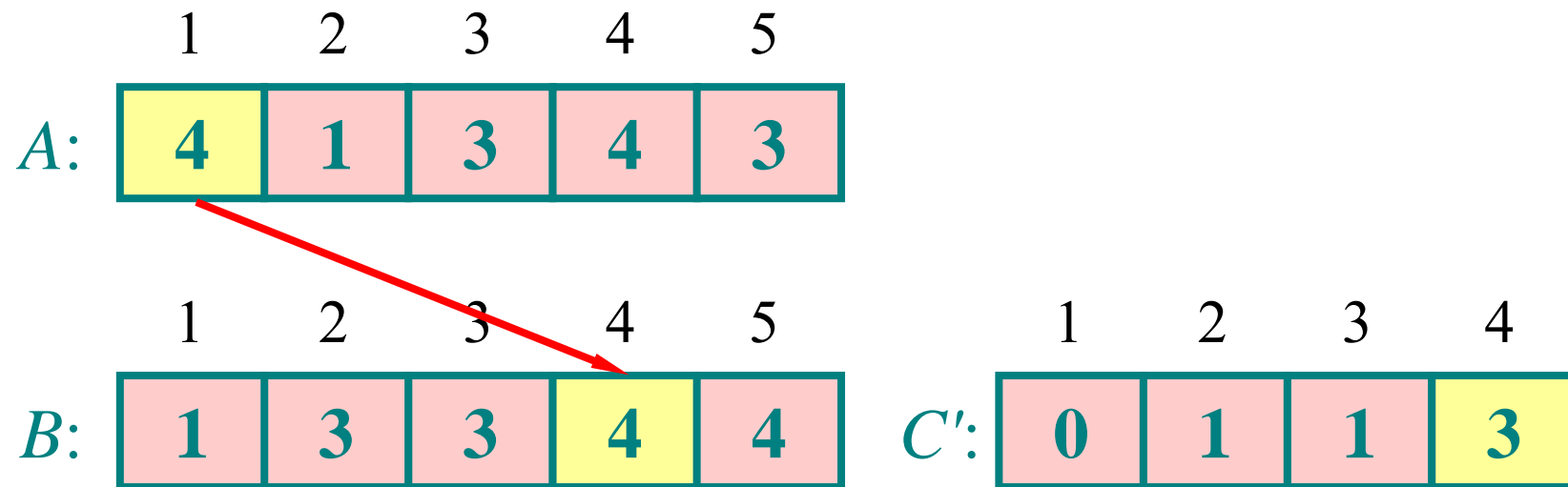
7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



7. **for** $j \leftarrow \text{length}[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

| | | | | | |
|----|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| A: | 4 | 1 | 3 | 4 | 3 |

| | | | | | |
|----|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| B: | 1 | 3 | 3 | 4 | 4 |

done

Counting sort

COUNTING-SORT(A, B, k)

1. **for** $i \leftarrow 1$ **to** k
 2. **do** $C[i] \leftarrow 0$
 3. **for** $j \leftarrow 1$ **to** $length[A]$
 4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$
 5. **for** $i \leftarrow 2$ **to** k
 6. **do** $C[i] \leftarrow C[i] + C[i-1]$
 7. **for** $j \leftarrow length[A]$ **downto** 1
 8. **do** $B[C[A[j]]] \leftarrow A[j]$
 9. $C[A[j]] \leftarrow C[A[j]] - 1$
- }

$\Theta(k)$

}

$\Theta(n)$

}

$\Theta(k)$

}

$\Theta(n)$

}

$\Theta(n + k)$

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

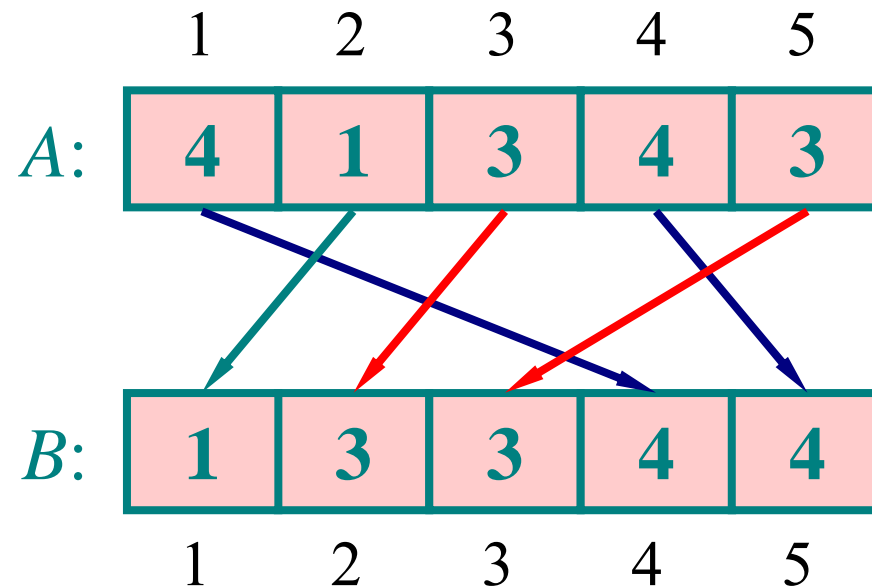
- But, sorting takes $\Omega(n \lg n)$ time!
- Why?

Answer:

- *Comparison sorting* takes $\Omega(n \lg n)$ time.
- Counting sort is not a *comparison sort*.
- In fact, not a single comparison between elements occurs!

Stable sorting

Counting sort is a *stable* sort: it preserves the input order among equal elements.



Radix sort

- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on *least-significant digit first* with auxiliary *stable* sort.

Operation of radix sort

3 2 9

4 5 7

6 5 7

8 3 9


4 3 6

7 2 0

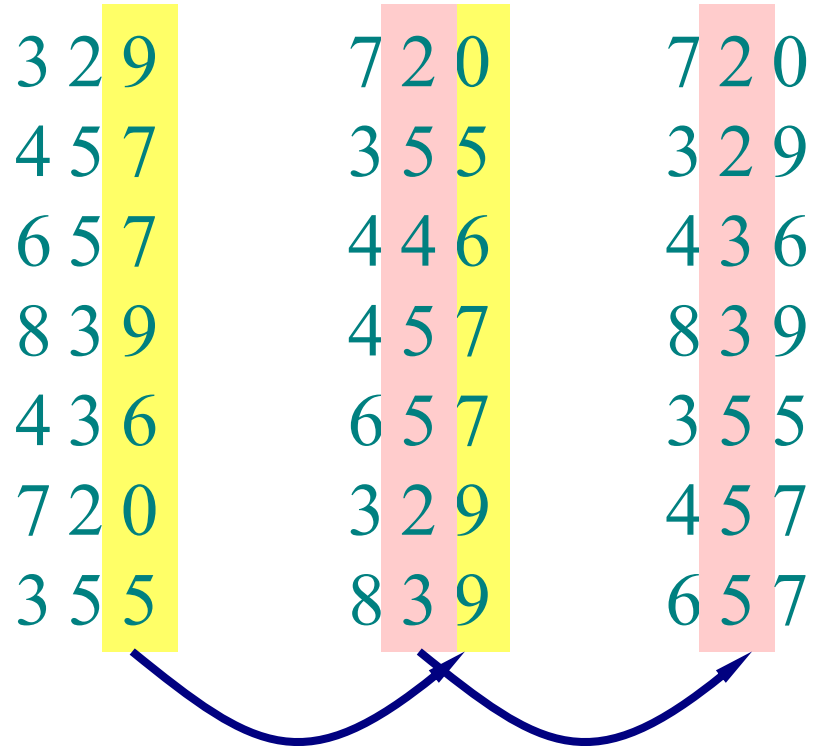
3 5 5

Operation of radix sort

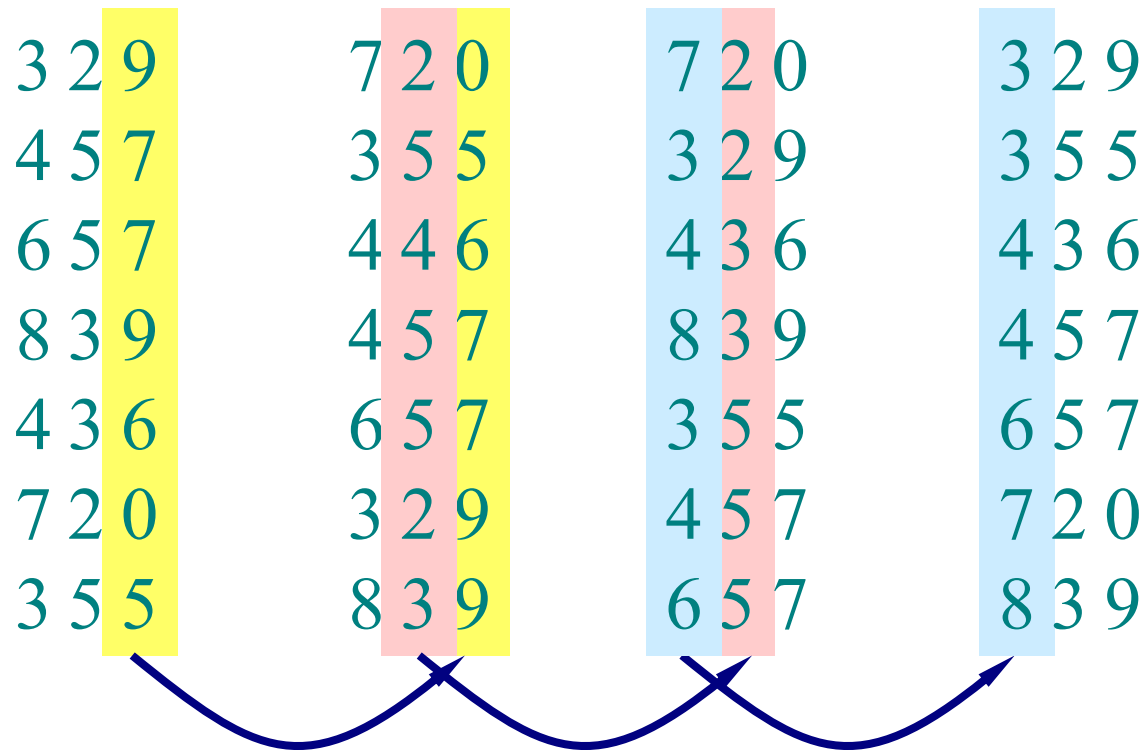
| | |
|-------|-------|
| 3 2 9 | 7 2 0 |
| 4 5 7 | 3 5 5 |
| 6 5 7 | 4 4 6 |
| 8 3 9 | 4 5 7 |
| 4 3 6 | 6 5 7 |
| 7 2 0 | 3 2 9 |
| 3 5 5 | 8 3 9 |



Operation of radix sort



Operation of radix sort



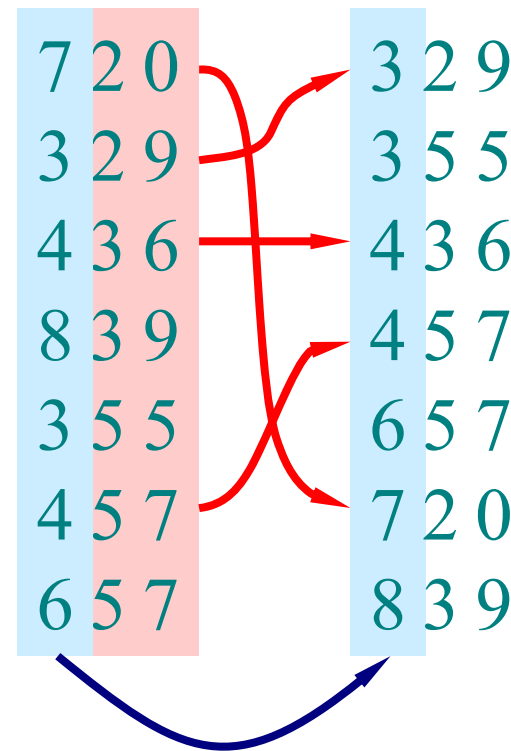
Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t .

Two numbers that differ in digit t are correctly sorted.

Two numbers equal in digit t are put in the same order as the input \Rightarrow correct order.



Radix sort

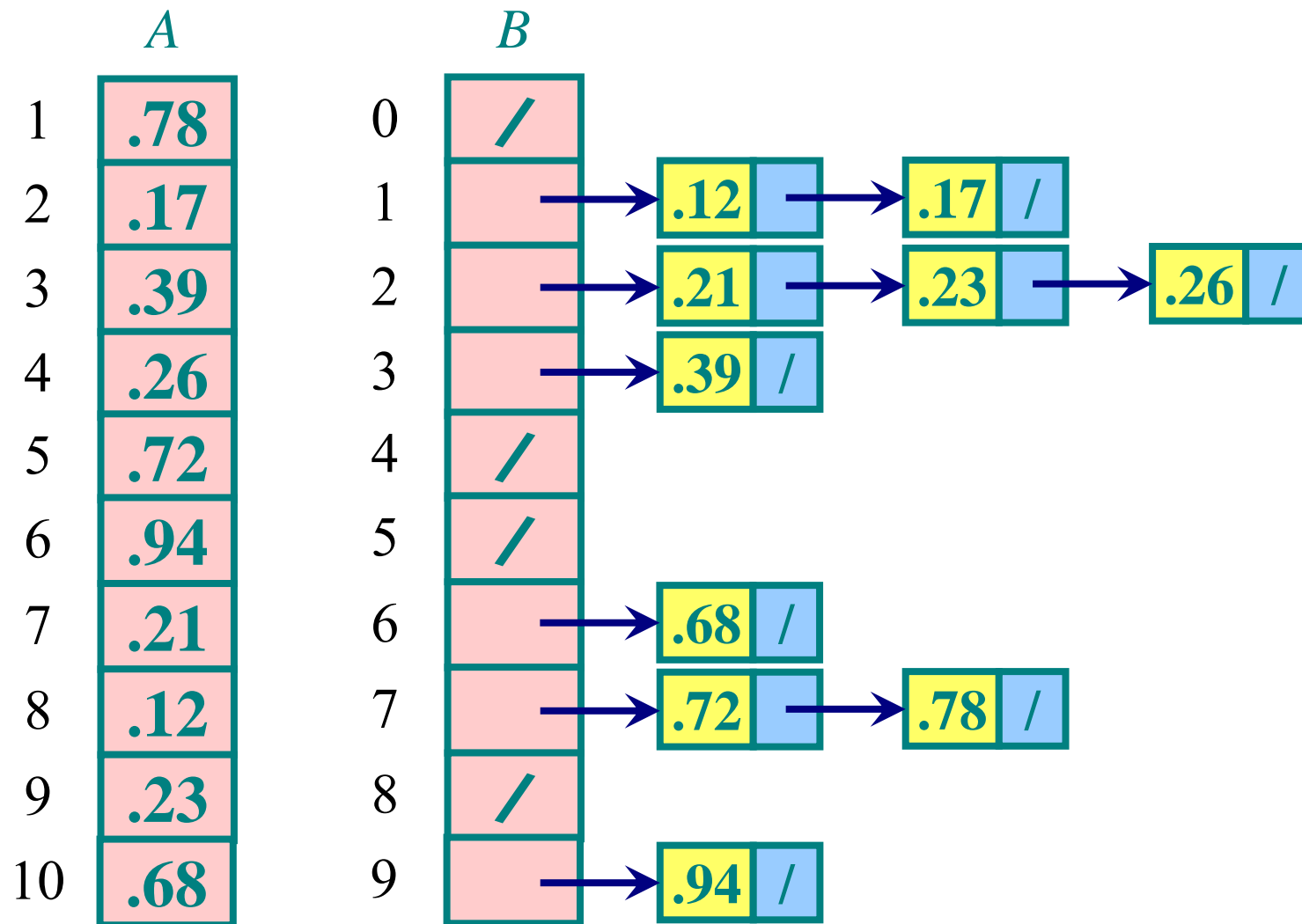
RADIX-SORT(A, d)

1. **for** $i \leftarrow 1$ **to** d
2. **do** use a stable sort to sort array A on digit i

$\Theta(d(n + k))$ Each digit can take on up to k
possible values

when d is constant and $k = O(n) \implies \Theta(n)$

Operation of bucket sort



Bucket sort

- ❑ Bucket sort runs in linear time when the input is drawn from a *uniform distribution*.
- ❑ Bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the *interval* $[0, 1)$.
- ❑ The idea of bucket sort is to divide the interval $[0, 1)$ into n equal-sized subintervals, or *buckets*, and then distribute the n input numbers into buckets.

Bucket sort

BUCKET-SORT(A)

1. $n \leftarrow \text{length}[A]$
2. **for** $j \leftarrow 1$ **to** n
3. **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i \leftarrow 0$ **to** $n - 1$
5. **do** sort list $B[i]$ with insertion sort
6. concatenate the list $B[1], B[2], \dots, B[n]$ together in order

Analysis of bucket sort

Let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time, the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Analysis of bucket sort

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E\left[O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right) \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\ &= \Theta(n) + n \cdot O(2 - 1/n) \\ &= \Theta(n) \quad \square \end{aligned}$$

Analysis of bucket sort

$X_{ij} = I \{A[j] \text{ falls in bucket } i\}$

for $i = 0, 1, \dots, n - 1$ and $j = 1, 2, \dots, n$. Thus,

$$n_i = \sum_{j=1}^n X_{ij}$$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right]$$

Analysis of bucket sort

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik}\right] \\ &= E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} X_{ij} X_{ik}\right] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij} X_{ik}] \end{aligned}$$

Analysis of bucket sort

$$\begin{aligned} E[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} \end{aligned}$$

$$\begin{aligned} E[X_{ij} X_{ik}] &= E[X_{ij}] E[X_{ik}] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \left(\frac{1}{n}\right)^2 \end{aligned}$$

Analysis of bucket sort

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij} X_{ik}] \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \left(\frac{1}{n}\right)^2 \\ &= 1 + \frac{n-1}{n} \\ &= 2 - \frac{1}{n} \quad \square \end{aligned}$$

Any question?



Xiaoqing Zheng
Fudan University