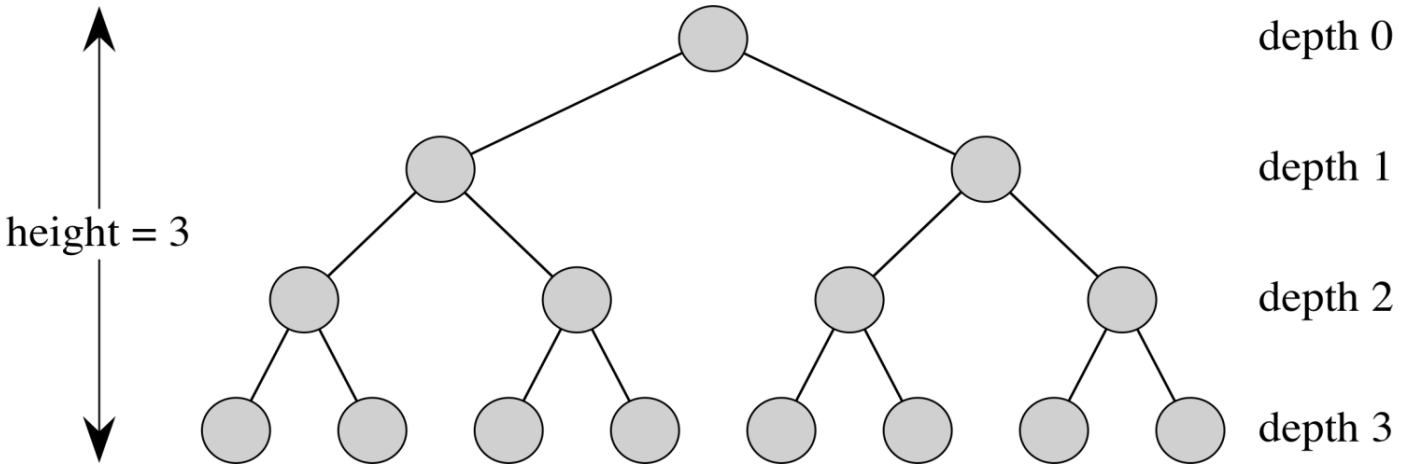
More sorting algorithms: Heap sort & Linear sorts

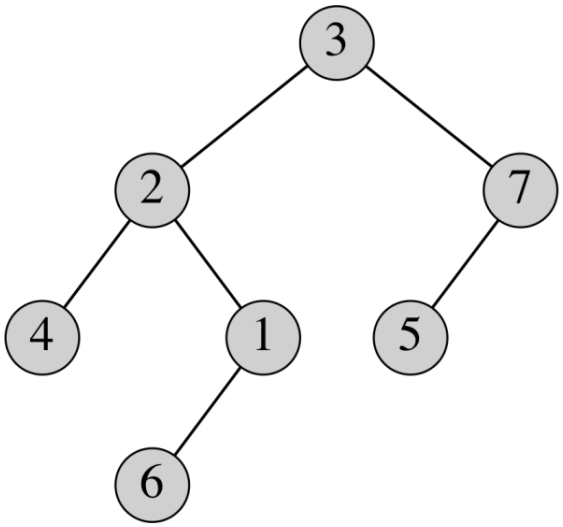
Heap Data Structure and Heap Sort

Basic Definition

- Depth of a node
  - the number of edges in the simple path from the node to the root of the tree.
- Depth of a tree T
  - the maximum depth of all nodes in T
- Height of a node
  - # of edges on a longest simple path from the node down to a leaf node.
- Height of a tree T
  - height of the root of T = depth of T

Depth of tree nodes





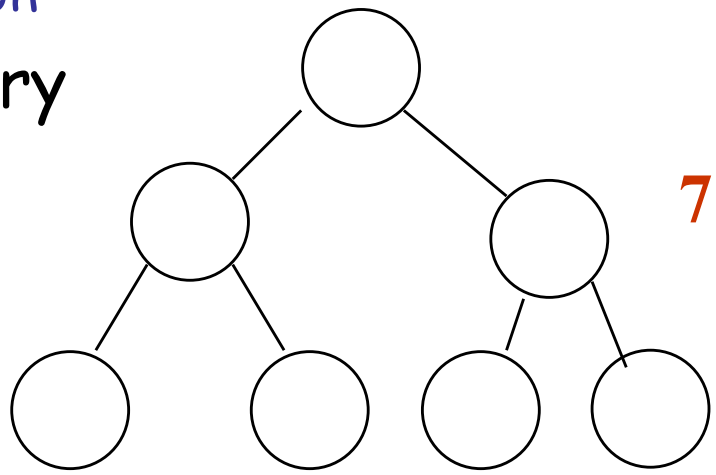
Depth of node 2 = 1
Depth of T = 3
Height of node 2 = 2
Height of T = 3

Terminologies

- Complete binary tree
  - Every internal node has two children
  - All leaves have depth d
- Essentially complete binary tree
  - It is a complete binary tree down to a depth of d-1
  - The nodes with depth d are as far to the left as possible

A complete binary tree

- A complete binary tree is a binary tree such that:
  - All internal nodes have 2 children
  - All leaves have the same depth d
- Number of nodes at level k = 2^k
- Total number of nodes in a complete binary tree with depth *d* is $n = 2^{d+1} - 1$
  - Exercise: Proof by induction
- Depth of a complete binary tree with n nodes is
- $d = \lg(n+1) - 1$

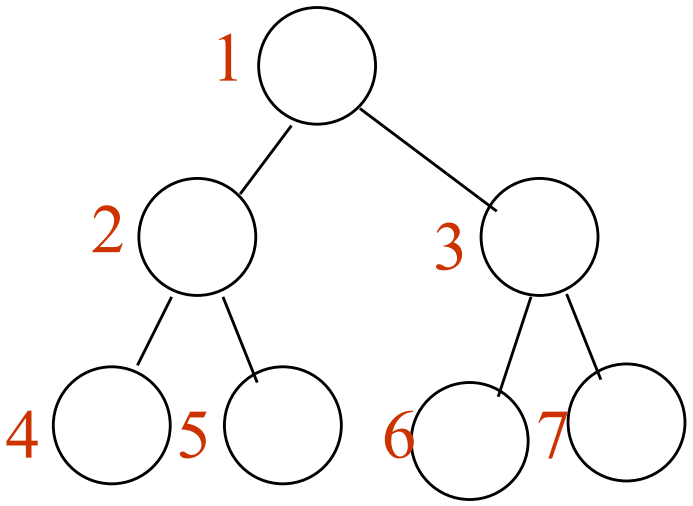


$7 = 2^{2+1} - 1$

A full binary tree of depth = height = 2

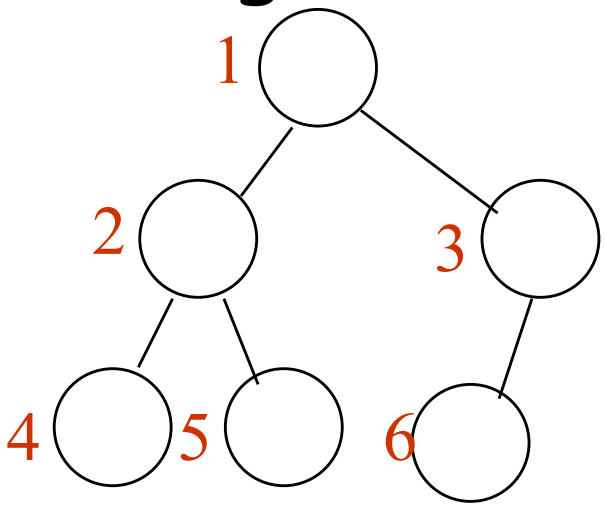
A complete binary tree (cont.)

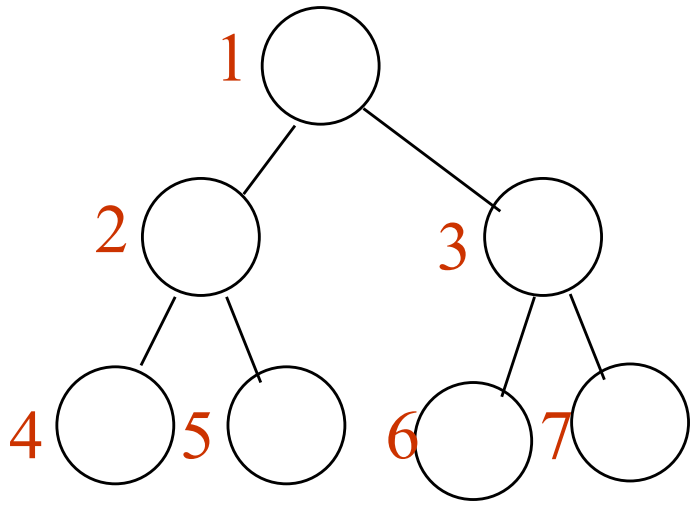
- Number of the nodes of a full (complete) binary tree of depth d:
  - root at depth 0 is numbered 1
  - The nodes at depth 1, …, d are numbered consecutively from left to right, in increasing depth
  - You can store the nodes in a 1D array in increasing order of node number



Essential complete binary tree

- An ***essential*** *complete binary tree* of depth *d* and *n* nodes is a binary tree such that its nodes would have the numbers 1, …, *n* in a binary tree of depth *d*.
- The number of nodes $2^d \le n \le 2^{d+1} - 1$
- $d = \lfloor \lg n \rfloor$ (See the next slide for proof)





Depth of an essential complete binary tree

- Number of nodes n satisfy:

$$2^d \leq n \leq 2^{d+1} - 1 \quad (1)$$

- By taking the log base 2, we get:

$$d \leq \lg n < d + 1 \quad (2)$$

- Since d is integer but $\lg n$ may not be an integer,

$$d = \lfloor \lg n \rfloor$$

- For a complete binary tree, $d = \lfloor \lg n \rfloor$ because (1) & (2) are satisfied for a complete binary tree too

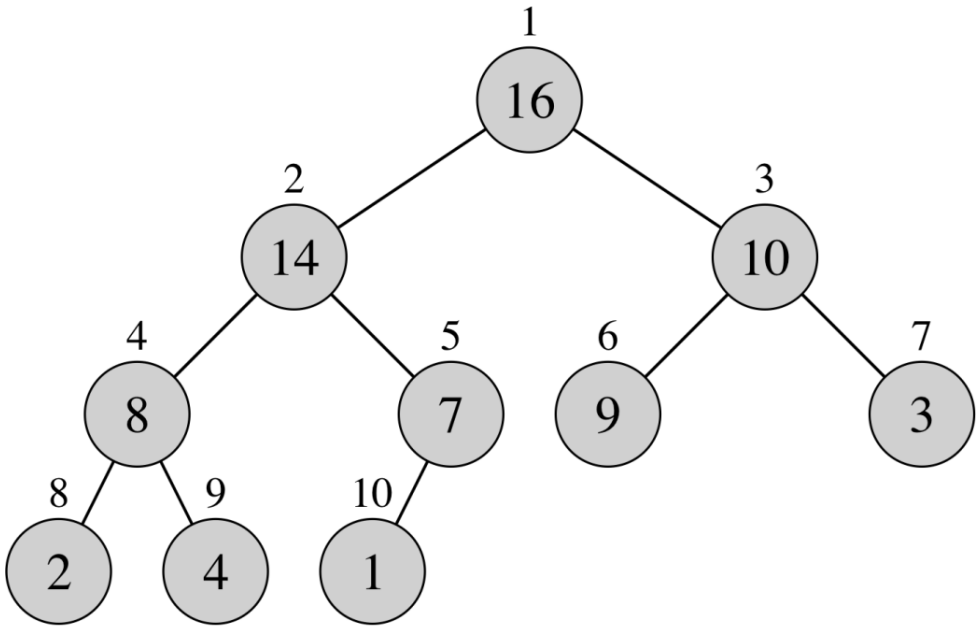
Heap Property

- Heap

  A heap is an essentially complete binary tree such that
  - The values stored at the nodes come from an ordered set
  - The value stored at each node is less/more than or equal to the values stored at its children → min-heap/max-heap
- Usage of heap
  - Heap sorting
  - Priority queue

Heap Property

- An example of a max-heap



- **What's the usually way to implement heap?**
- **Numbers outside the circles are index numbers of the nodes when the heap is stored in an array.**
- **Numbers inside the circles are the values or keys of each node.**

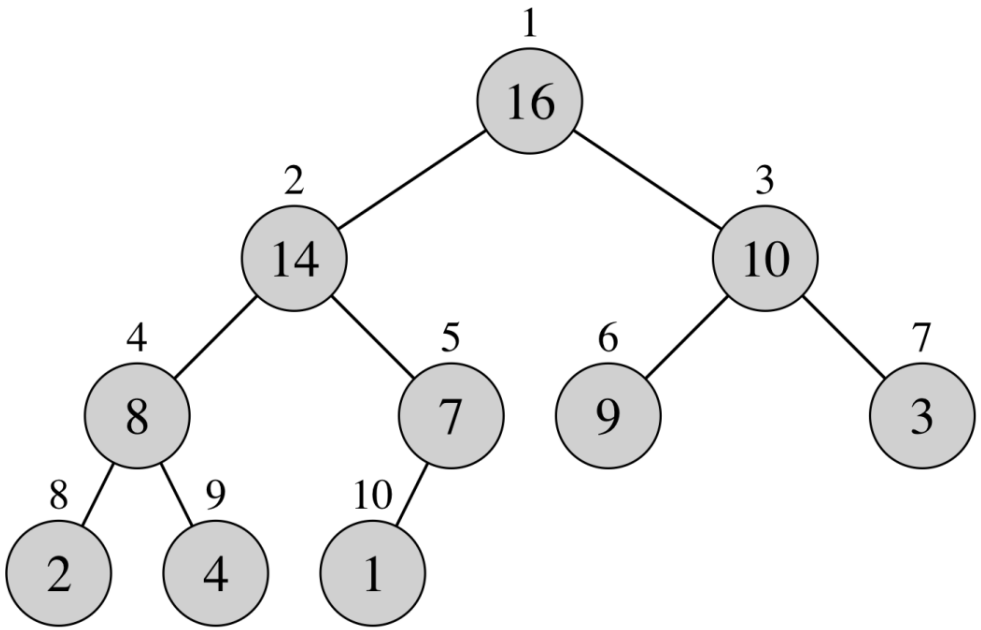
Array Implementation of Heap

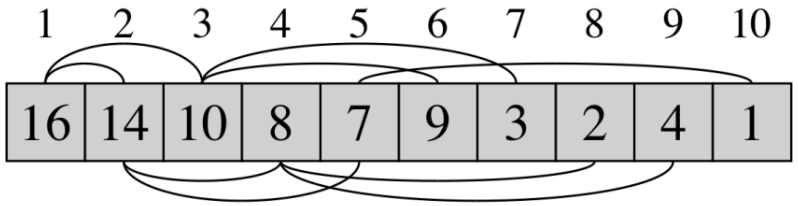
- **A heap can be stored as an array A.**
  - Root of tree is $A[1]$.
  - Left child of $A[i] = A[2i]$.
  - Right child of $A[i] = A[2i + 1]$.
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.

- **With this array implementation, basic operations such as finding the parent, the left child or the right child of a node can be performed very quickly.**

Array Implementation of Heap

- **An array implemented max-heap**





Arcs go between parents and children.

Basic Heap Procedures

- **Max-Heapify**: a key procedure for maintaining max-heap property; it runs in $O(\lg n)$ time.
- **Build-Max-Heap**: a procedure for building a max-heap from an unordered input array; it runs in $\Theta(n)$ time.
- **Heapsort**: it sorts an array in place with running time $O(n \lg n)$.
- **Max-Heap-Insert**, **Heap-Extract-Max**, **Heap-Increase-Key**, and **Heap-Maximum**: these procedures enable **priority queue** implementation using the heap data structure.

Maintaining Heap Property

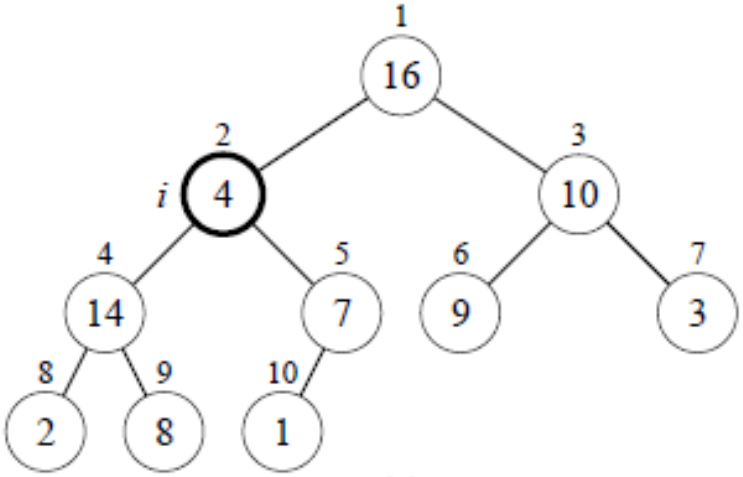
- *Max-Heapify* is used to maintain the max-heap property.
  - Before Max-Heapify: *A*[*i*], may be smaller than its child node(s).
  - Condition: The left and right subtrees of *i* are already max-heaps.
  - After Max-Heapify: the subtree rooted at *i* is a max-heap.
- Main Idea:
  - Compare A[i], A[Left(i)], and A[Right(i)].
  - If necessary, swap A[i] with the larger of the two child nodes.
  - Continue this process of comparing and swapping down the heap, until the subtree rooted at i is a max-heap.

Algorithm Max-Heapify

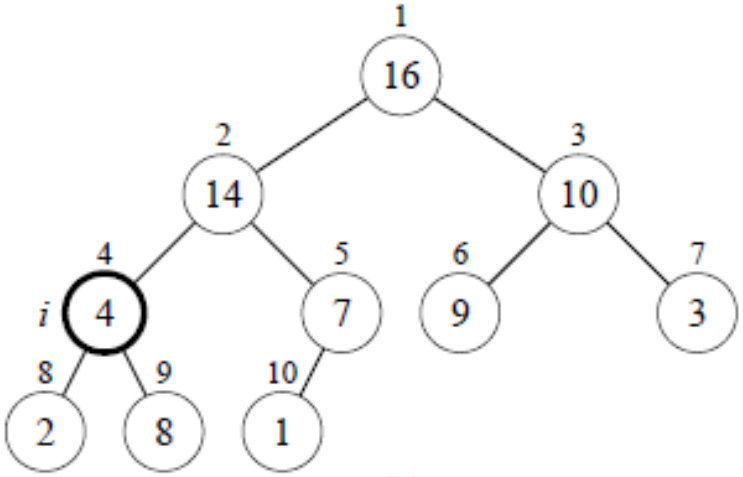
```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
       largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
       largest = r
  if largest ≠ i
       exchange A[i] with A[largest]
       MAX-HEAPIFY(A, largest, n)
```

- Running time: *O(lg n)*
  - The height of the tree is lg n and moving A[i] one level down takes a constant number of operations.

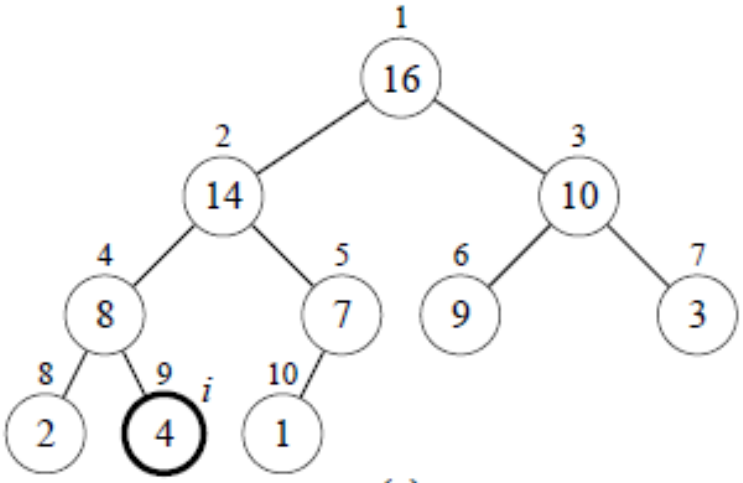
Illustrating Max-Heapify



(a)



(b)



(c)

- Node 2 violates the max-heap property.
- Compare node 2 with its child nodes, and then swap it with the larger of the two child nodes.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Building a Heap

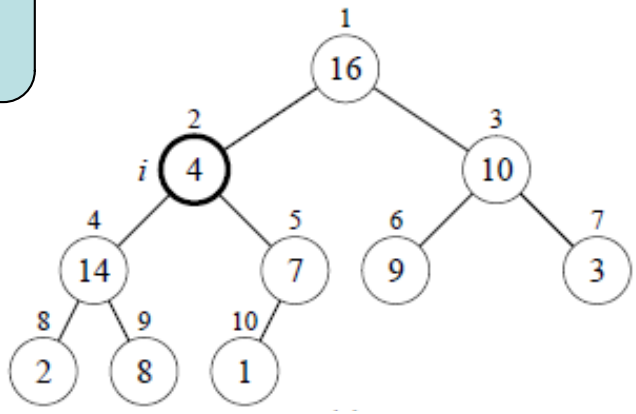
- The following bot…e, given an unordered array A, will proc…

Why start from this number but not n?

BUILD-MAX-HE…(A, n)

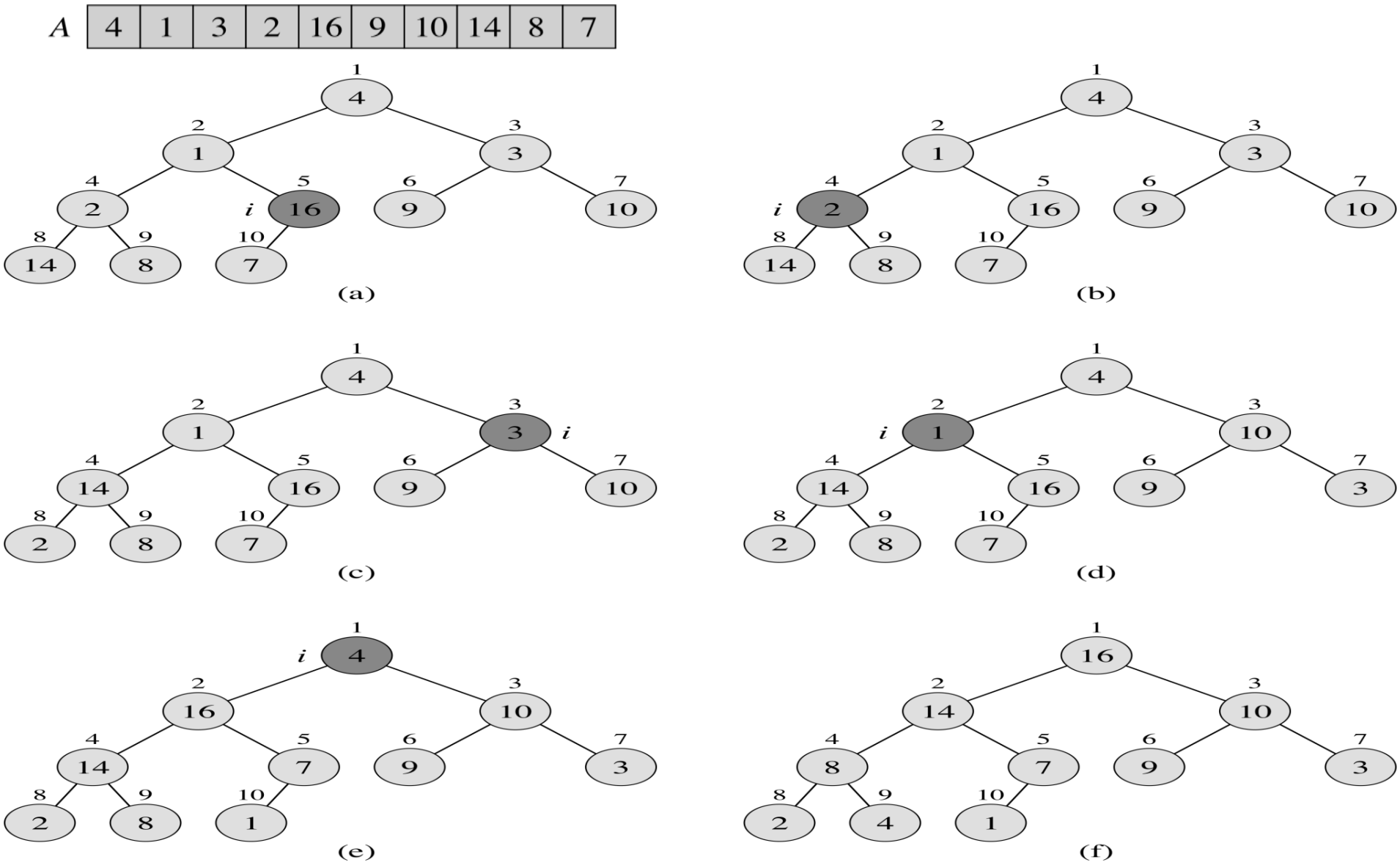
for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)



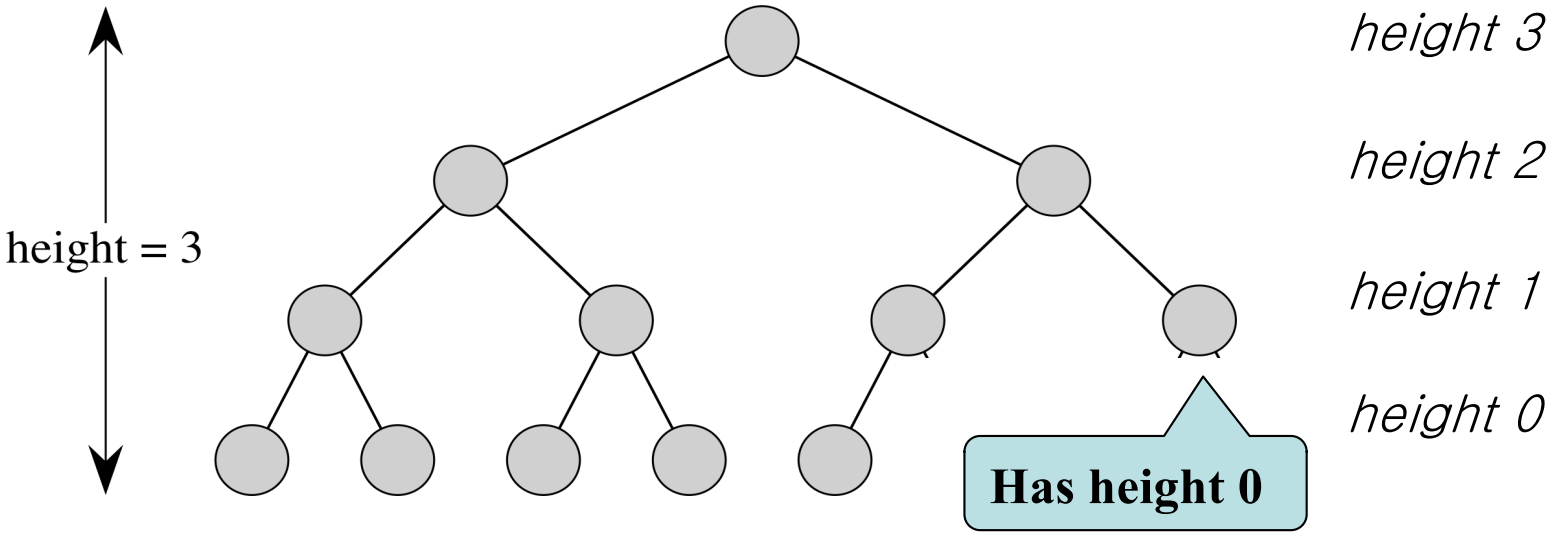
- During the heapification process, only non-leaf nodes need to be considered.
- All nodes corresponding to elements in subarray $A[\lfloor n/2 \rfloor + 1 .. n]$ are leaf nodes because $A[\lfloor n/2 \rfloor]$ is the element with the largest index to have child.
  - Note that the left child of $A[\lfloor n/2 \rfloor]$ is $A[2\lfloor n/2 \rfloor]$, which is $A[n]$ if n is an even number or $n - 1$ if n is an odd number.

Build-Max-Heap: Example



Build-Max-Heap: Analysis (1)

- *Simple bound*: $O(n)$ calls to Max-Heapify, each of which takes $O(\lg n)$ time ➔ Build-Max-Heap takes $O(n \lg n)$ time.
- Can we find a better bound?
- *Tighter bound*: Observation: Time to run Max-Heapify for a node is linear in the height of the node, and most nodes have small heights. Height of the heap is $\lg n$.
  - At most $\lceil n/2^{h+1} \rceil$ nodes are of height h.



Build-Max-Heap: Analysis (2)

- *Tighter bound* (continued):
- At most $\lceil n/2^{h+1} \rceil$ nodes are of height h (see previous slide).
- 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

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

- Since $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ for $|x| < 1$, by differentiating both sides

$$\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2} \Rightarrow \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \Rightarrow O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2} \right)^h \right) = O(2n)$$

- Thus, the running time of Build-Max-Heap is $O(n)$.

Heapsort Algorithm: Idea

Given an input array, the *Heapsort* algorithm acts as follows:

- Builds a max-heap from the array.
- Starting with the root (which has the maximum value), the algorithm places the maximum value into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is already in its correct place) by decreasing the heap size, and calling Max-Heapify on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest value) remains, and therefore is in the correct place in the array.

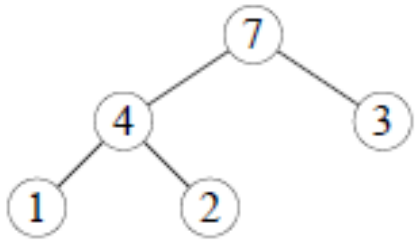
Heapsort Algorithm: Pseudocode

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

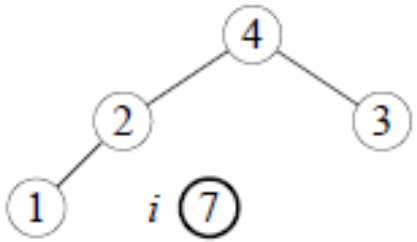
Heapsort Algorithm: Example

Initial array:

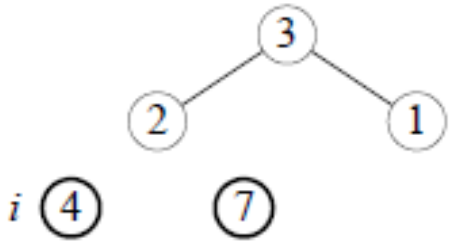
A	7	4	3	1	2



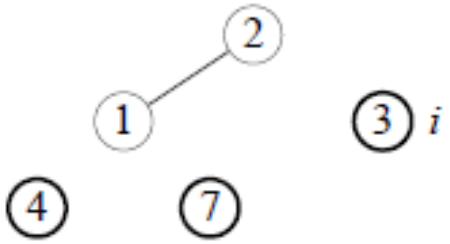
(a)



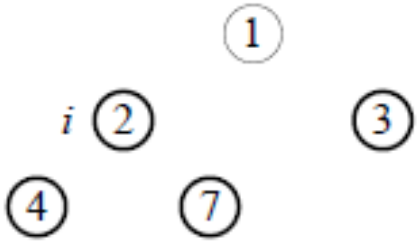
(b)



(c)



(d)



(e)

Sorted array:

A	1	2	3	4	7

Heapsort Algorithm: Analysis

```
HEAPSORT(A, n)
    BUILD-MAX-HEAP(A, n)
    for i = n downto 2
        exchange A[1] with A[i]
        MAX-HEAPIFY(A, 1, i − 1)
```

- Build-Max-Heap: $O(n)$
- for loop: $n - 1$ times
  - exchange values: $O(1)$
  - Max-Heapify: $O(\lg n)$
- *Total time*: $O(n \lg n)$
  - The same as Merge-Sort but it sorts in place.

Priority Queue

- A priority queue is a *collection* of zero or more items,
  - Each item is associated with a priority

- Operations:
  - Insert a new item
  - Find the item with the highest priority
  - Delete the highest priority item

Heap Implementation of Priority Queue

- All priority queue operations have running time O(lg n).
  - **findMax(*A*)**
    - $\Theta(1)$
  - **deleteMax(*A*) from a heap with n items**
    - $\Theta(\lg n)$
  - **insert(*A*,*x*) into a heap with n items**
    - $\Theta(\lg n)$
  - **IncreasePriority(*A*,*x*,*k*)**

Linear sorts

Sorting Algorithms So Far (1)

- How fast can we sort?
  - $\Omega(n \lg n)$ is already the best we can do for comparison-based sorting.
- Insertion-sort:
  - Easy to code
  - Sort in place
  - Efficient on already sorted or nearly-sorted inputs; best case running time: $O(n)$.
  - $O(n^2)$ worst case & average case running time

Sorting Algorithms So Far (2)

- Merge-sort:
  - Divide-and-conquer approach:
    - Divide input array in half
    - Recursively sort subarrays
    - Linear merge time
  - $\Theta(n \lg n)$ worst case running time
  - Doesn't sort in place

Sorting Algorithms So Far (3)

- Heapsort:
  - Use the very useful heap data structure
    - Nearly complete binary tree
    - Heap property: parent key > children's keys
  - $O(n \lg n)$ worst case running time
  - Sort in place

Sorting Algorithms So Far (4)

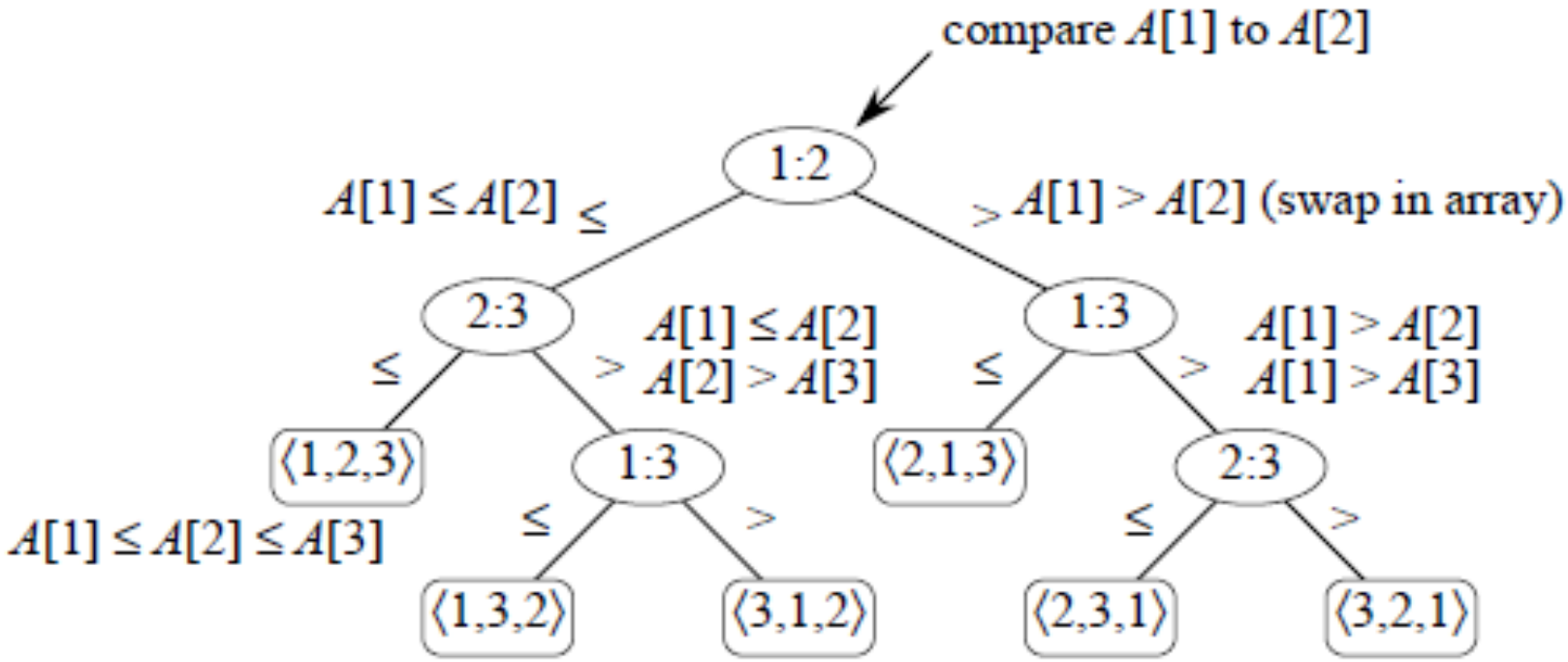
- Quicksort:
  - Divide-and-conquer approach:
    - Partition array into two subarrays
      - values in first subarray ≤ values in second subarray
    - Recursively sort subarrays
    - No merge step needed
  - $\Theta(n \lg n)$ average case running time
  - $\Theta(n^2)$ worst case running time
    - Naïve implementation: worst case on sorted input
    - Address this with randomized Quicksort
  - Fast in practice

How Fast Can We Sort?

- **We will provide a lower bound, then beat it**
  - **by playing a different game.**
- **First, an observation: all sorting algorithms we have introduced so far are *comparison sorting*.**
  - **The only operation used to gain ordering information is the pair-wise comparison of two elements.**
- **Lower bounds for sorting**
  - **$\Omega(n)$ to examine all the input.**
  - **All sorting algorithms seen so far are $\Omega(n \lg n)$ in the worst case.**
  - **We'll show that $\Omega(n \lg n)$ is a lower bound for comparison sorting.**

Lower Bound for Comparison Sorting: Decision Tree Example

- **A decision tree for Comparison sorting with an input of size 3.**



- ***Tree paths from root to leaves are all possible execution traces.***

Lower Bound for Sorting by Comparisons: Recap

- When a list of n integers is given as input, there are n! permutations
- Build a decision tree that has n! leaf nodes where each leaf could be a sorted permutation
- Height of the decision tree (h) indicates #total comparisons to reach a sorted permutation
  - $n! \leq 2^h$
  - $\lg(n!) \leq h$
  - $n! > (n/e)^n$
  - $h \geq \lg\left(\frac{n}{e}\right)^n = n(\lg n - \lg e) = \Omega(n \lg n)$

Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

Sorting in Linear Time

- **We introduce three linear, i.e., $\Theta(n)$ time, sorting algorithms:**
  - **Pigeonhole-sort**
  - **Counting-sort**
  - **Radix-sort**
  - **Bucket-sort**
- **Obviously these algorithms avoid doing pair-wise comparisons between input values.**

Pigeonhole Sort: Main Idea

- ***Problem*: Sort *n* keys (values) in ranges 1 .. *k*.**
- ***Main idea*:**
  - **Count the number of keys with value *i* .**
    - **Maintain count in an auxiliary array, *C*. *C*[*i*] saves the # of times value *i* appears in the input.**
  - **Use counts to overwrite input to produce sorted result.**

Example:

Input A

2	1	2	4	3	1	2

- **After counting**

C

1	2	3	4
2	3	1	1

- **After overwriting**

Output A

1	1	2	2	2	3	4

Pigeonhole Sort: Algorithm

```
Pigeonhole-Sort(A, k, n)          // A has n values
    for i = 1 to k                // initialize C
        C[i ] = 0
    for j = 1 to n
        C[A[ j ]] = C[A[ j ]] + 1   // Count keys: C[ i ] = m when
    q = 1                           // value i appears in A m times
    for  j = 1 to k       // rewrite A
        while C[ j ] > 0
            A[q] = j
            C[ j ] = C[ j ] – 1
            q = q + 1
```

Pigeonhole Sort: Analysis

- **Running time: $\Theta(2k + 2n)$**
  - **Note that for the last "for loop", we have $\Sigma\ C[j] = n$.**
- **When $k = O(n)$, $\Theta(2k + 2n) = \Theta(n)$.**

- **How can it achieve $\Theta(n)$ time and beat the lower bound of comparison sorting algorithms?**
- **No comparisons between input values!**
- ***But* it also depends on an assumption about the numbers to be sorted, that is, the values are in the range 0 .. k.**

Counting Sort: Main Idea

- ***Problem*: Sort n values stored in $A[1 .. n]$.**
- ***Assumption*: All values in A are in the range $0 .. k$.**
- ***Main idea*:**
  - **Count in auxiliary array $C[0 .. k]$ the number of times each value i appears, $i = 0, \ldots, k$.**
  - **Use counts in C to compute the offset in sorted array B of values $= i$ for $i = 0, \ldots, k$.**
  - **Copy A into sorted B using and updating (decrementing) the computed offsets.**
  - **To make the sort *stable* we start at last position of A.**
  - **The output is in $B[1 .. n]$.**
    - **Notice 1: not sorting in place**
    - **Notice 2: Pigeonhole-sort does not require B.**

Counting Sort: Computing Offset

- **Assume $C[0] = 3$: There are 3 values of 0 and they should be stored in positions 1, 2 and 3 in the sorted array B.**

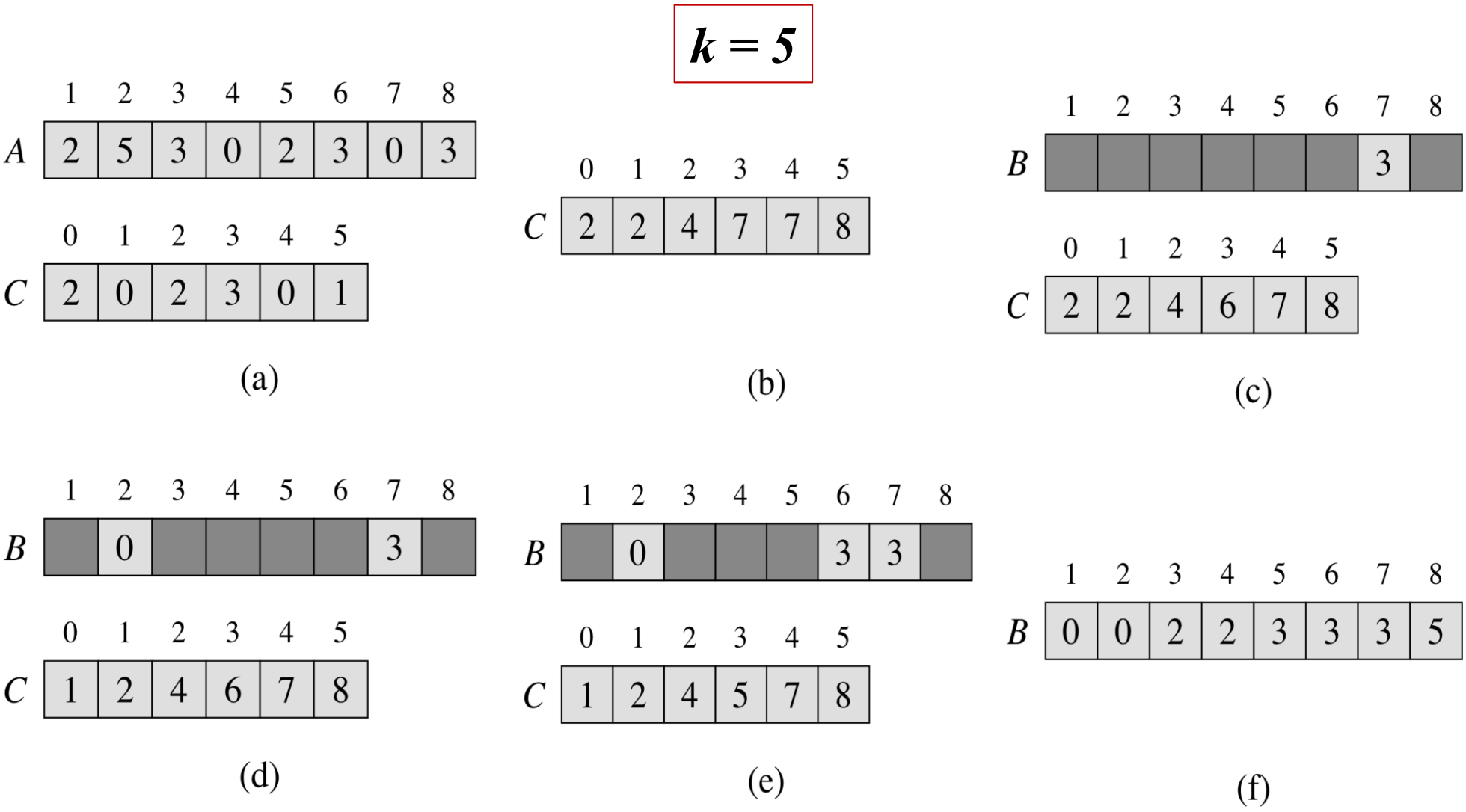
  ➔ We keep the offset for value 0 to be 3.
- **Let $C[1] = 2$. Then the 2 values of 1 should be in placed in positions 4 and 5 in B.**

  ➔ Keep the offset for value 1 to be 2.
- **We compute the offset for value 1 to be ($C[1]$ + offset for value 0) = 2 + 3 = 5.**
- **In general, the offset for value i is $C[i]$ + offset for $i - 1$.**

Counting Sort: Algorithm

```
COUNTING-SORT(A, B, n, k)
  let C[0..k] be a new array
  for i = 0 to k
      C[i] = 0                                  // Initialize C
  for j = 1 to n
      C[A[j]] = C[A[j]] + 1                     // Count different values
  for i = 1 to k
      C[i] = C[i] + C[i − 1]                    // Compute offset
  for j = n downto 1
      B[C[A[j]]] = A[j]                         offset = C[A[ j ]]
                                                B[offset] = A[ j ]
      C[A[j]] = C[A[j]] − 1                     // Update offset
```

Counting Sort: Example



Counting Sort: Stable Sorting (1)

***Definition*: A sorting algorithm is *stable* if it always preserves the original order of equal keys, i.e., keys with the same value appear in the same order in output as they did in input.**

- ***Example*: Consider input: (2, *a*), (3, *b*), (3, *c*), (1, *d*), where each element has two components and we want to sort the input by the first component.**
  - **A stable sorting algorithm will produce: (1, *d*), (2, *a*), (3, *b*), (3, *c*).**
    - **Notice how (3, *b*) and (3, *c*) are in the same order as they did in the input.**
  - **A non-stable sorting algorithm may produce: (1, *d*), (2, *a*), (3, *c*), (3, *b*).**
    - **Still correctly sorted, but the original order for (3, *b*) and (3, *c*) are not retained.**

Counting Sort: Stable Sorting (2)

- ***Question*: Is Counting-sort a stable sorting algorithm?**
- ***Answer*: Yes.**
  - **Notice how it processes the elements in A? From $A[n]$ to $A[1]$.**
  - **Notice also when a value appears multiple times, the last element with the value is always placed in the last position among those positions reserved for elements with this value.**
    - **Pay attention to how *offset* works.**
- ***Question*: Is Pigeonhole-sort stable?**
- ***Answer*: No.**
  - **It makes no effort in retaining the order of equal-valued elements.**
- ***Question*: Why is stable sorting important?**
- ***Answer*: We will find out soon …**

Counting Sort: Limitations

- ***Question*: Why don't we always use Counting-sort?**
- ***Answer*: Because it requires the values to appear in the range 0 .. *k*.**
- **If the values to be sorted are not in a "small" range (i.e., $k >> n$), this algorithm is neither computation efficient nor space efficient.**
  - **Recall the running time of this algorithm is $\Theta(n + k)$.**
    - **What if $k = \Theta(n^2)$ or larger?**
  - **Recall the algorithm requires array $C[0 .. k]$ for auxiliary storage**
- **Imagine that we use Counting-sort to sort a small number of 32-bit integers.**
  - **Not practical: *k* will be too large ($2^{32} = 4{,}294{,}967{,}296$)**

Radix sort

- When we know nothing about the keys to be sorted, we have no choice but sorting them by comparisons. So, sorting is $\Omega(nlgn)$.
- However, if we know something about data, we can take advantage of the knowledge to do sorting faster.
  - Example: Suppose we know that the keys are all nonnegative integers represented in base 10. Also, each key is at most d digits where d is a positive constant.

Radix sort

- Main idea
  - Break key into "digit" representation

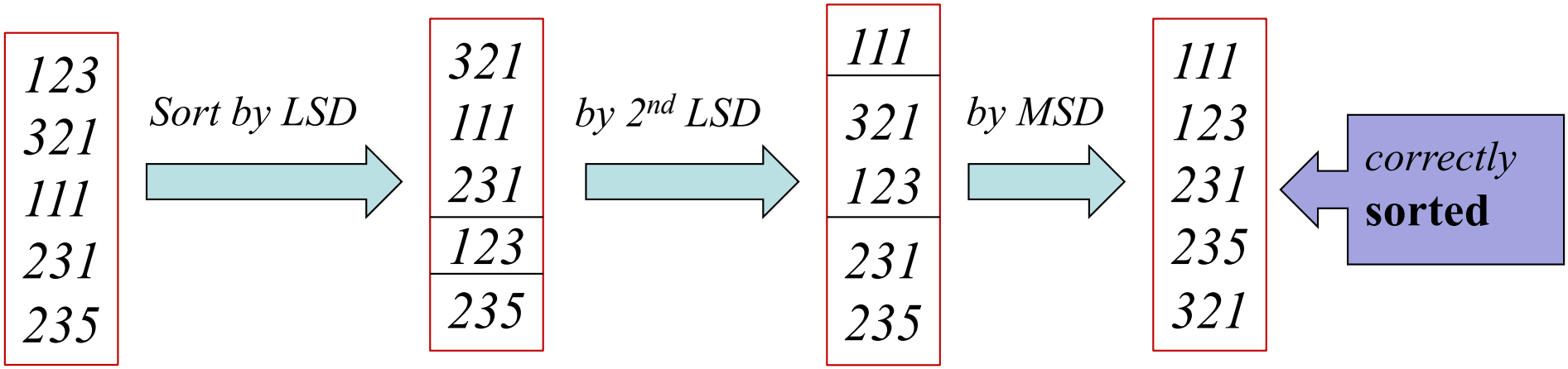
    key = $i_d, i_{d-1}, \ldots, i_2, i_1$
  - "digit" can be a number in any base, a character, etc.
- Radix sort:

  for i= 1 **to** d

    sort "digit" i using a stable sort
- Analysis : Θ(d * (stable sort time)) where d is the number of "digits"

Radix Sort: Main Idea

- ***Key idea of Radix-sort*: Sort the *least significant digit* first, then the second least significant digit, and so on.**



- **Note the importance of stable sorting at each step. Without it, correctness cannot be guaranteed.**

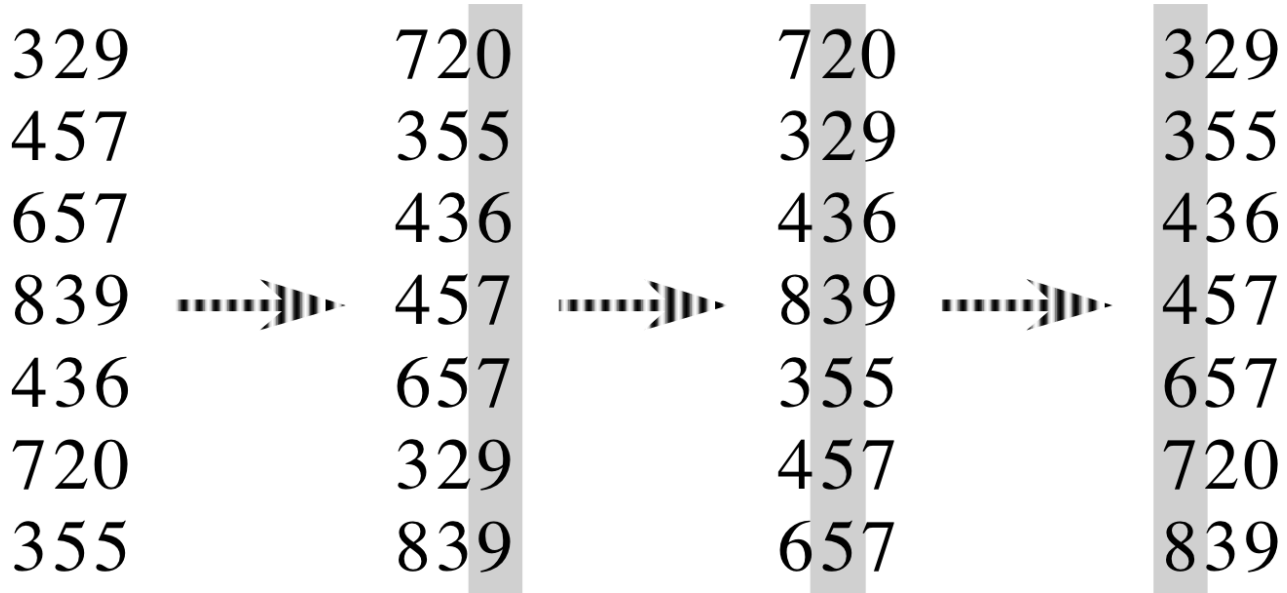
Radix Sort: Algorithm

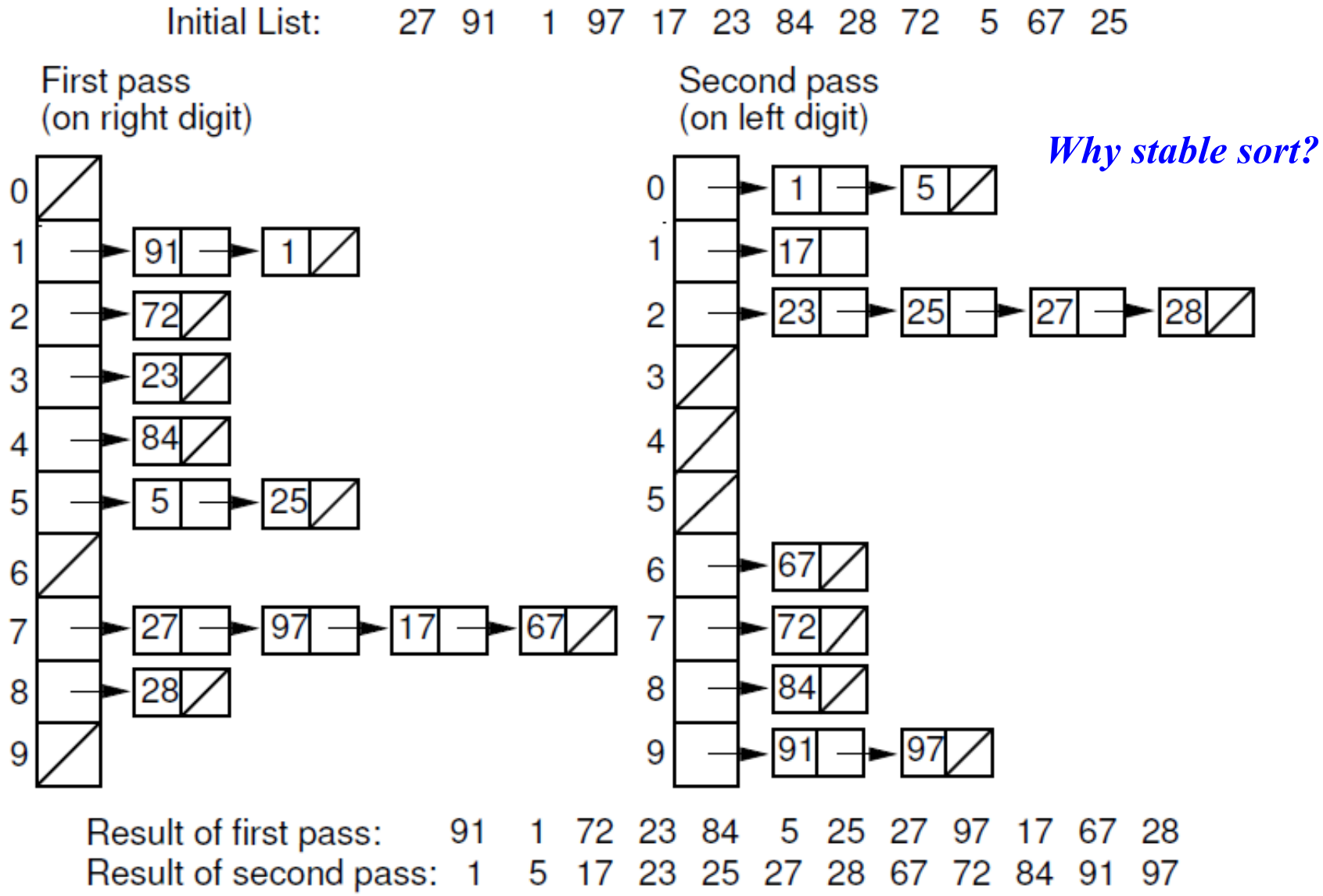
Let A be an array of numbers, each having d digits.

RADIX-SORT(A, d)

for $i = 1$ **to** d *// 1 corresponds to the LSD.*

use a stable sort to sort array A on digit i





Source: C.A. Shaffer, A Practical Introduction to Data Structures and Algorithm Analysis (freely available online)

Radix Sort: Stable Sorting

- ***Question*: What sorting algorithm to use to sort on individual digits?**
- ***Answer*: Counting-sort is an obvious choice:**
  - **It is stable.**
  - **Sort n numbers on digits that range from 0 .. k**
  - **Running time: $O(n + k)$**
- **Each pass over n numbers with d digits takes time $O(n + k)$ ➔ total time is $O(dn + dk)$.**
  - **When d is a constant and $k = O(n)$, Radix-sort takes $O(n)$ time.**

Radix Sort: Comparison (1)

- ***Problem*: Sort 1 million 32-bit numbers.**
- **If we use any comparison sorting algorithm, the cost is at least (with constant factor, usually larger than 1, ignored):**

  $n \lg n = 1000000 \lg 1000000 \approx 1000000 \lg 2^{20} = 20{,}000{,}000$

- **The maximum number of 32-bit is $2^{32} = 4{,}294{,}967{,}296$. That is, 32-bit numbers have at most 10 digits.**
- **Use Radix-sort: $n = 1000000$, $d = 10$,**

  $k = 10$, i.e., $0 .. k = \{0, 1, \ldots, 9\}$

  Use the original cost (keep constant):
  - **Counting-sort for each digit: $2n + 2k = 2{,}000{,}020$**
  - **There are 10 digits: $10 \times 2{,}000{,}020 = 20{,}000{,}200$**

  ➔ About the same as comparison sorting algorithms.
    - **Actually better because we ignored the constant factors in comparison sorting algorithms.**

Radix Sort: Comparison (2)

- **We can do much better than 20 million running time if we use Radix-sort more wisely.**
- ***Key idea*: Each "digit" in Radix-sort needs not be limited to be within {0, 1, …, 9}. We can have multi-digit "digits".**
- **Try this: let each digit in Radix-sort represent 8-bit numbers.**

  ➔ 32-bit numbers become 4-digit 8-bit numbers, i.e., $d = 4$.
- **For 8-bit numbers: $k = 256$, i.e., 0 .. k = {0, 1, …, 255}**
- **Now the cost for Radix-sort becomes:**
  - **Counting-sort for each digit: $2n + 2k = 2{,}000{,}512$**
  - **There are 4 digits: $4 \times 2{,}000{,}512 = 8{,}002{,}048$**

  ➔ Much better than the 20,000,200 we had earlier.

Radix Sort: Comparison (3)

- ***Question*: Can we do even better than 8,002,040?**
- ***Answer*: Yes, with smaller *d* and larger *k*.**
- **Try this: let each digit in Radix-sort represent 16-bit numbers.**

  ➔ 32-bit numbers become 2-digit 16-bit numbers, i.e., $d = 2$.

- **For 16-bit numbers: $k = 65536$, i.e., 0 .. $k = \{0, 1, \ldots, 65535\}$**
- **Now the cost for Radix-sort becomes:**
  - **Counting-sort for each digit: $2n + 2k = 2{,}130{,}072$**
  - **There are 2 digits: $2 \times 2{,}130{,}072 = 4{,}262{,}144$**

Radix Sort: Balancing k and d (1)

Let's look at the general problem:

- **Suppose we have n values to be sorted, each having b bits.**
- **Break the values into r-bit digits: $d = \lceil b / r \rceil$.**
- **Use Counting-sort: $k = 2^r - 1$.**

***Example*: 32-bit values, 8-bit digits. $b = 32$, $r = 8$, $d = \lceil b / r \rceil = 4$, $k = 2^8 - 1 = 255$.**

- **Running time =** $\Theta\left(\frac{b}{r}(n + 2^r)\right)$
- ***Question*: How to choose r to minimize the above cost?**

Radix Sort: Balancing k and d (2)

- ***Answer*: Let $r = \lg n$. Now we have**

$$\Theta\left(\frac{b}{r}(n+2^r)\right) = \Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta\left(\frac{bn}{\lg n}\right)$$

  - **If $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ is still $\Theta(n)$, not improved.**
  - **If $r > \lg n$, then $n + 2^r$ gets bigger, increasing the complexity.**
    For example: if $r = 2 \lg n$, then $2^r = 2^{2 \lg n} = (2^{\lg n})^2 = (n^{\lg 2})^2 = n^2$.

***Example*: For sorting $n = 1$ million 32-bit values,**

$r = \lg 1000000 \approx 20$, so $d = \lceil b / r \rceil = \lceil 32 / 20 \rceil = 2$.

$k = 2^r - 1 = 1{,}048{,}575$. ➔ *lead to a higher cost than for $r = 16$.*

- **The above analysis is approximate because it is based on Θ and $\lceil b / r \rceil \neq b / r$.**
- **After d is estimated, using $r = \lceil b / d \rceil$ is a good practice.**

Radix Sort: Limitations

- **The running time of Radix-sort is $\Theta(n)$ under some conditions ($k = O(n)$ & d is a constant), which is better than comparison sorting algorithms.**
- **Unlike Counting-sort, Radix-sort has much looser range constraint.**
- ***Question*: Why would we ever use anything but Radix-sort?**
- ***Answer*: There are several reasons:**
  - **For smaller n, a $\Theta(n)$ algorithm is not necessary faster than a $\Theta(n \lg n)$ algorithm because of the hidden constant factors.**
  - **Counting-sort does not sort in place ➔ Radix-sort needs significantly more memory than in-place sorting algorithms.**
  - **When input values have wide ranges (e.g., mixing a few very large values with mostly small values), Radix-sort is not efficient.**

Bucket Sort: Main Idea

- ***Assumption*: Values are distributed uniformly in interval [0, 1).**
  - **Numbers not in this interval can be mapped to the interval.**

  ***Example*: Numbers between 0 and 99 can be mapped to the interval by dividing each number by 100.**
- ***Main idea***
  - **Divide [0, 1) into *n* equal-sized *buckets* for a sorting problem of with *n* input values.**
  - **Distribute the *n* input values into the buckets.**
  - **Sort each bucket with Insertion-sort.**
  - **Concatenate the buckets in order, listing elements in each one.**

Bucket Sort: Algorithm

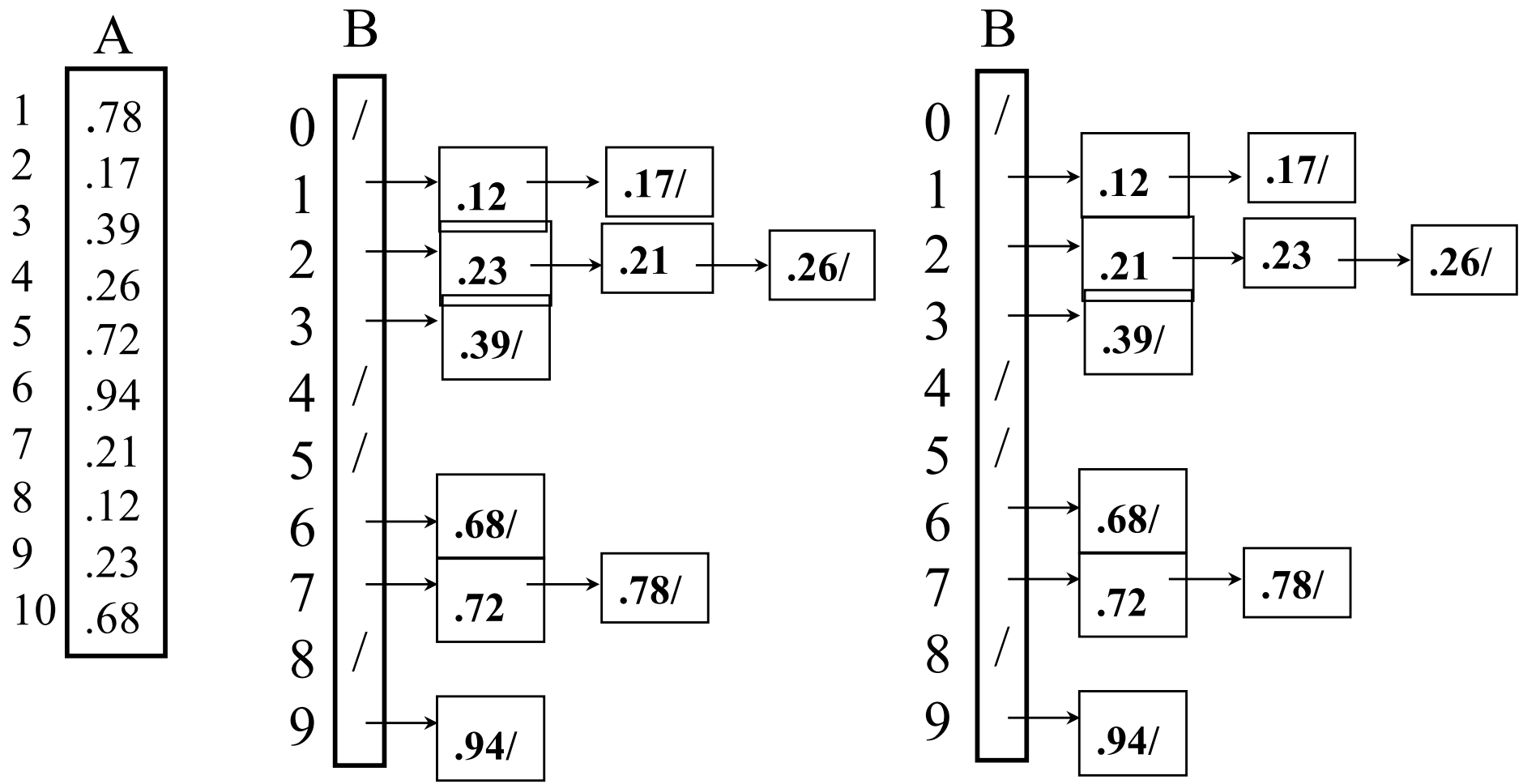
Input: A[1 .. n], where $0 \le A[i] < 1$ for all i .

Auxiliary array: B[0 .. n – 1] of linked lists, all initially empty.

BUCKET-SORT(A, n)

let $B[0..n-1]$ be a new array
for $i = 0$ **to** $n - 1$
    make $B[i]$ an empty list
for $i = 1$ **to** n
    insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$ **// Distribute the input values // to different buckets.**
for $i = 0$ **to** $n - 1$
    sort list $B[i]$ with insertion sort **// Sort each bucket.**
concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order
return the concatenated lists

Bucket Sort: Example



Step 1: distribute **Step 2: sort each bucket**

Step 3: concatenate ➔ .12 .17 .21 .23 .26 .39 .68 .72 .78 .94

Bucket Sort: Analysis

- **Relies on each bucket getting a small number of values.**
- **All lines of algorithm, except Insertion-sort, take $\Theta(n)$ altogether.**
- **If each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket ➔ $O(n)$ sort time for all buckets.**

BUCKET-SORT(A, n)

let $B[0\,..\,n-1]$ be a new array
for $i = 0$ **to** $n-1$
    make $B[i]$ an empty list
for $i = 1$ **to** n
    insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
for $i = 0$ **to** $n-1$
    sort list $B[i]$ with insertion sort
concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order
return the concatenated lists

- ***We "expect" each bucket to have few elements, since the average is 1 element per bucket.***
- ***But we need to do a careful analysis about the expected number of elements per bucket.***

Bucket Sort: Analysis

- **The expected running time of bucket sort is Θ(n).**
  - **Proof has been posted on blackboard.**
- ***Question*: What is the worst-case running time of bucket-sort?**
- ***Answer*: $O(n^2)$**
  - **It happens when all values appear in the same bucket.**
- ***Question*: Can we make the worst-case time to be $O(n \lg n)$?**
- ***Answer*: Yes – simply replace Insertion-sort by any $O(n \lg n)$ sorting algorithm.**