

CS 2009

Design and Analysis of Algorithms

Waheed Ahmed

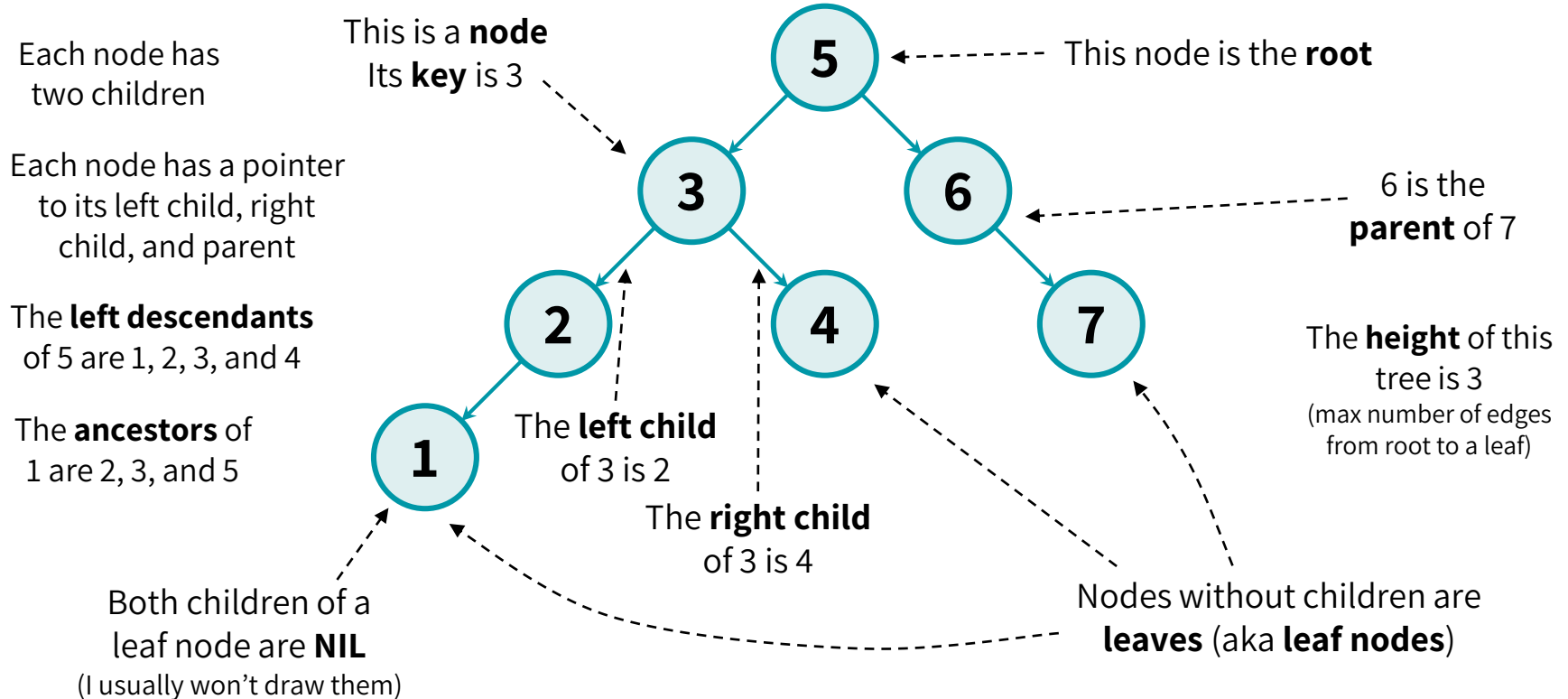
Farrukh Salim Shaikh

Lecture 15 & 16 & 17:

Heap Sort & Linear Time Sorting (Counting Sort, Radix Sort, Bucket Sort)

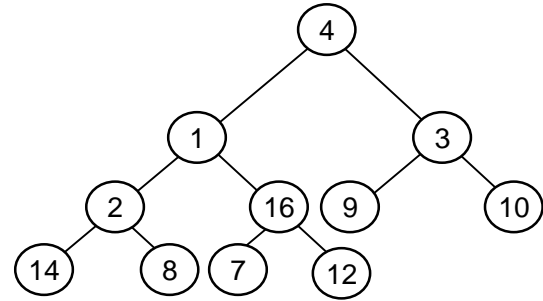
HEAP SORT

BINARY TREE TERMINOLOGY

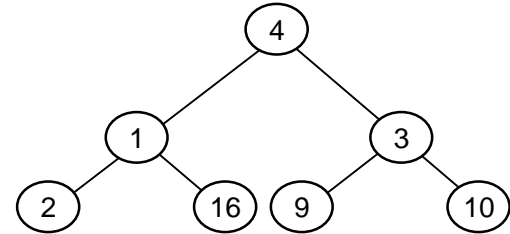


Special Types of Trees

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree exactly 2.
- *Def:* **Complete binary tree** = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Full binary tree

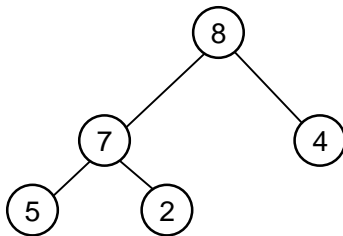


Complete binary tree

The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
 - **Structural property:** all levels are full, except possibly the last one, which is filled from **left to right**
 - **Order (heap) property:** for any node x

$$\text{Parent}(x) \geq x$$



Heap

From the heap property, it follows that:

“The root is the maximum element of the heap!”

A heap is a binary tree that is filled in order

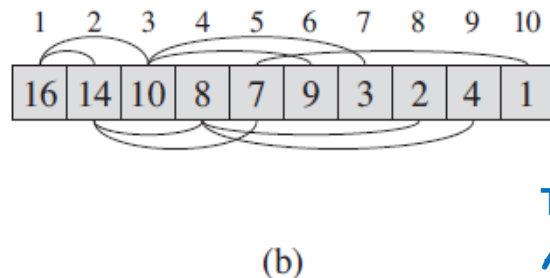
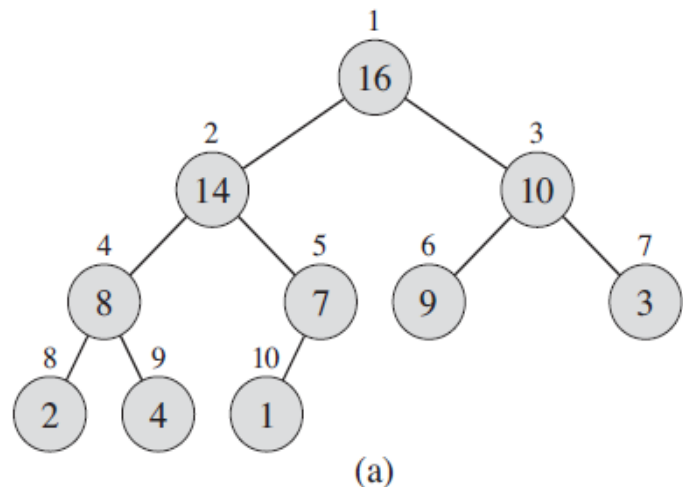
Array Representation of Heaps

Root of tree is $A[1]$

Left child of $A[i] = A[2i]$

Right child of $A[i] = A[2i + 1]$

Parent of $A[i] = A[\lfloor i/2 \rfloor]$

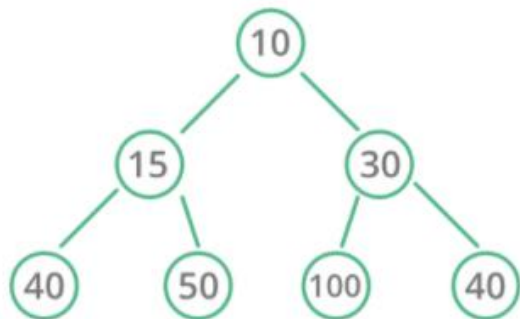


The elements in the subarray $A[\lfloor n/2 \rfloor + 1] \dots n$ are leaves

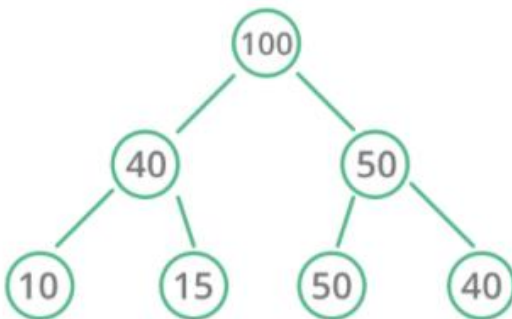
Figure 6.1 A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

Heap Types

- **Max-heaps** (largest element at root), have the max-heap property:
 - for all nodes i , excluding the root: $A[\text{PARENT}(i)] \geq A[i]$
- **Min-heaps** (smallest element at root), have the min-heap property:
 - for all nodes i , excluding the root: $A[\text{PARENT}(i)] \leq A[i]$
- For heap sort algorithm, we use **max-heaps**.
- **Min-heaps** are commonly used for implementing priority queues.



Min Heap



Max Heap

Adding/Deleting Nodes

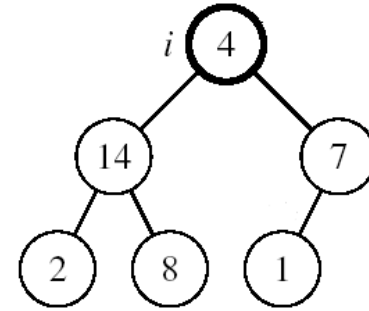
- New nodes are always **inserted** at the bottom level (**left to right**)
- Nodes are **removed** from the bottom level (**right to left**)

Operations on Heaps

- Maintain/Restore the max-heap property
 - MAX-HEAPIFY
- Create a max-heap from an unordered array
 - BUILD-MAX-HEAP
- Sort an array in place
 - HEAPSORT
- Priority queues

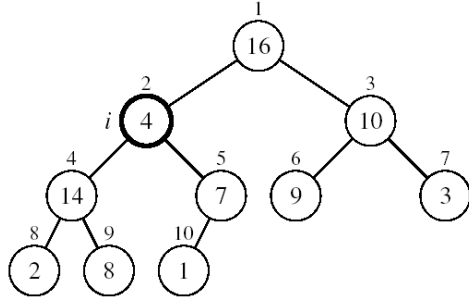
Maintaining the Heap Property

- Suppose a node is smaller than a child
 - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
 - Exchange with larger child
 - Move down the tree
 - Continue until node is not smaller than children

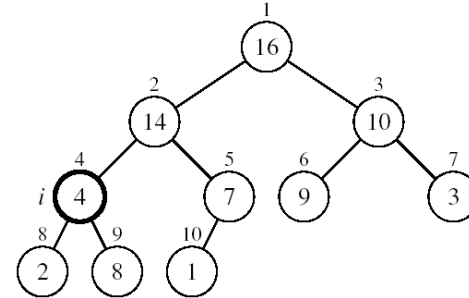


Example

MAX-HEAPIFY(A, 2, 10)



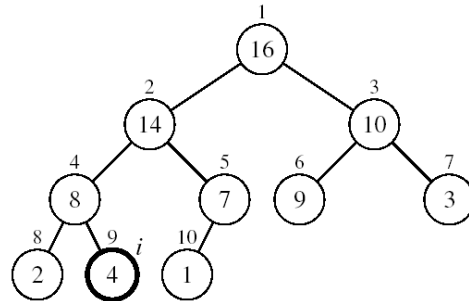
$A[2] \leftrightarrow A[4]$



A[2] violates the heap property

A[4] violates the heap property

$A[4] \leftrightarrow A[9]$

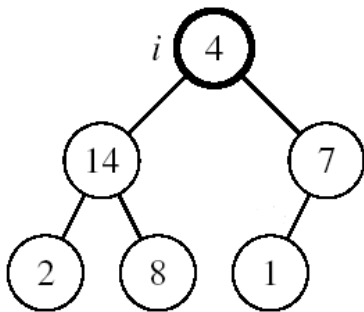


Heap property restored

Maintaining the Heap Property

- Assumptions:

- Left and Right subtrees of i are max-heaps
- $A[i]$ may be smaller than its children



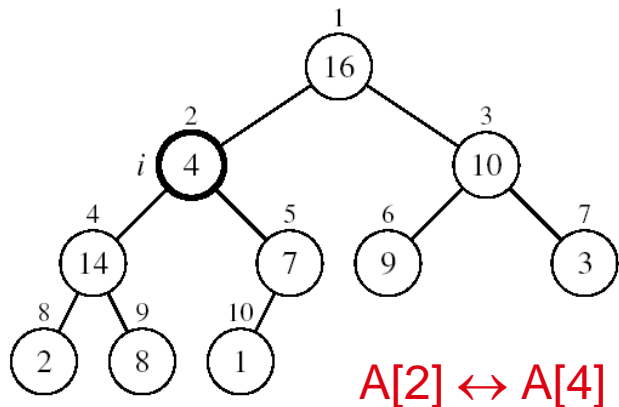
Alg: MAX-HEAPIFY(A, i, n)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq n$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq n$ 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**($A, \text{largest}, n$)

Maintaining the Heap Property

- Assumptions:

- Left and Right subtrees of i are max-heaps
- $A[i]$ may be smaller than its children



MAX-HEAPIFY(A, 2, 10)

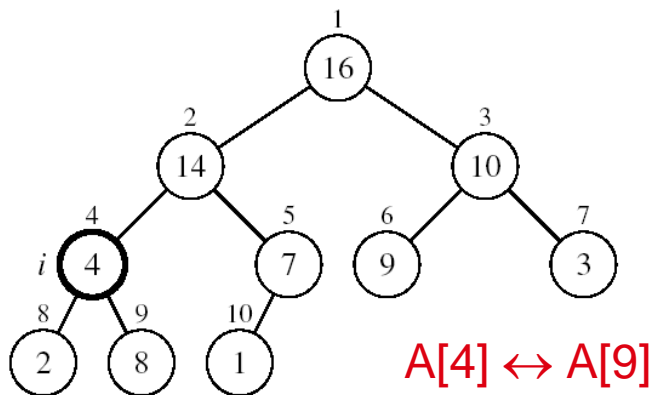
Alg: MAX-HEAPIFY(A, i, n)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq n$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq n$ 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, largest, n)

Maintaining the Heap Property

- Assumptions:

- Left and Right subtrees of i are max-heaps
- $A[i]$ may be smaller than its children



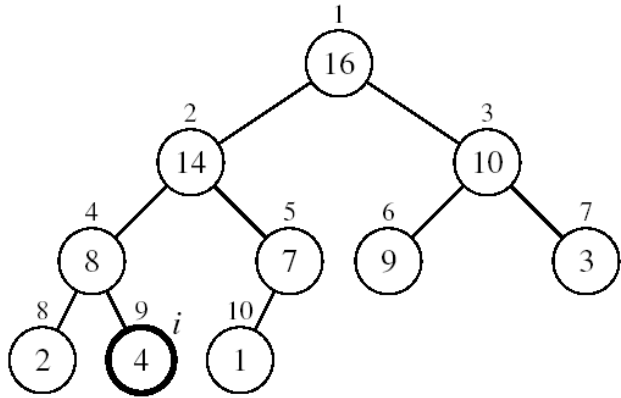
Alg: MAX-HEAPIFY(A, i, n)

- $l \leftarrow \text{LEFT}(i)$
- $r \leftarrow \text{RIGHT}(i)$
- if** $l \leq n$ and $A[l] > A[i]$
- then** $\text{largest} \leftarrow l$
- else** $\text{largest} \leftarrow i$
- if** $r \leq n$ and $A[r] > A[\text{largest}]$
- then** $\text{largest} \leftarrow r$
- if** $\text{largest} \neq i$
- then** exchange $A[i] \leftrightarrow A[\text{largest}]$
- $\text{MAX-HEAPIFY}(A, \text{largest}, n)$

Maintaining the Heap Property

- Assumptions:

- Left and Right subtrees of i are max-heaps
- $A[i]$ may be smaller than its children



Alg: MAX-HEAPIFY(A, i, n)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq n$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq n$ 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($A, \text{largest}, n$)

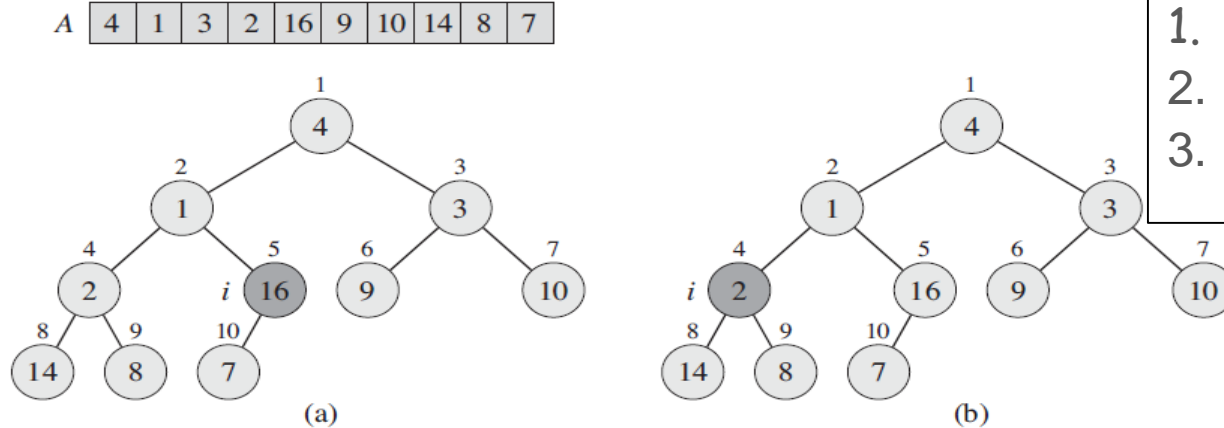
MAX-HEAPIFY Running Time

- Intuitively:
 - It traces a path from the root to a leaf (longest path length: h)
 - AT each level, it makes exactly 2 comparisons
 - Total number of comparisons are $2h$
 - Running time is $O(h)$ or $O(\log n)$
- Running time of MAX-HEAPIFY is $O(\lg n)$
- Can be written in terms of the height of the heap, as being $O(h)$
 - Since the height of the heap is $\lfloor \lg n \rfloor$

Build Max Heap Procedure

- Convert an array $A[1 \dots n]$ into a max-heap ($n = \text{length}[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Figure 6.3 :

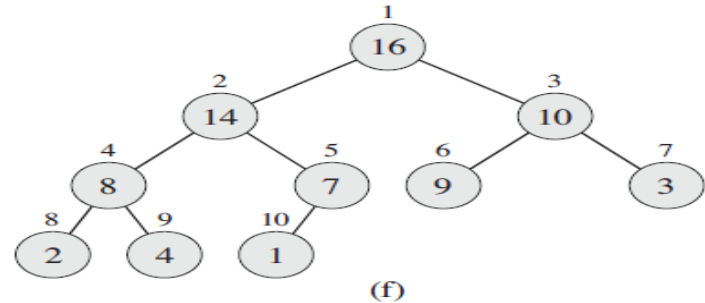
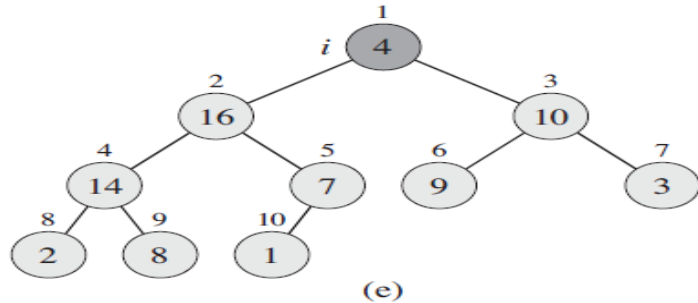
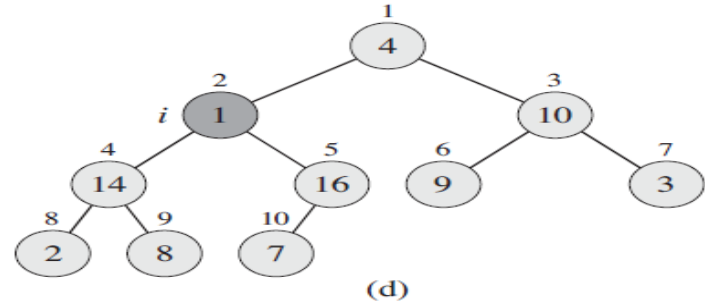
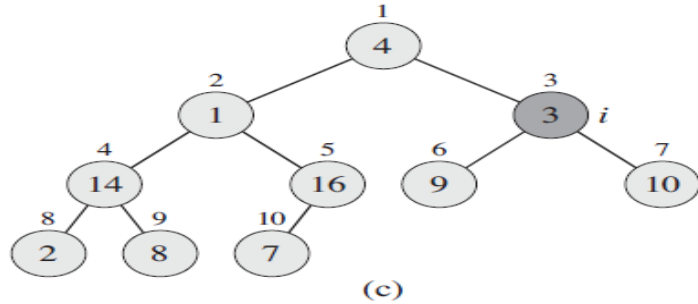


Alg: BUILD-MAX-HEAP(A)

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

Build Max Heap Procedure

- Figure 6.3 :



Heapsort

- It has time complexity of divide and conquer i-e “ $n \log n$ ” but it does not behave like divide and conquer because it splits data into sorted and unsorted sections. It is not divide and conquer algorithm
- It is sorting in place algorithm i-e does not require extra space like merge sort.
- Combines the better attributes of two sorting algorithms. Time complexity like mergeSort i-e “ $n \log n$ ” and space complexity like insertion sort i-e “ $O(1)$ ”
- Heap was used as “garbage collected storage” in languages like java, lisp etc.
- But here in heap sort, it will be used as a data structure and not the garbage collected storage.

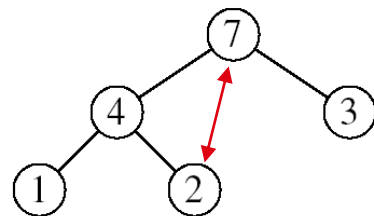
Heapsort

- **Goal:**

- Sort an array using heap representations

- **Idea:**

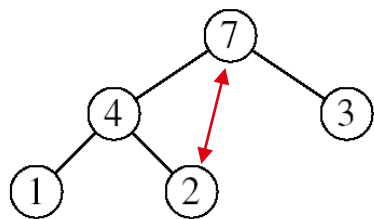
- Build a **max-heap** from the array
- Swap the root (the maximum element) with the last element in the array
- “Discard” this last node by decreasing the heap size
- Call MAX-HEAPIFY on the new root
- Repeat this process until only one node remains



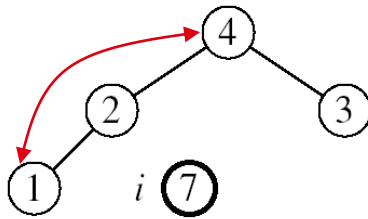
MAX-HEAPIFY(A, 1, 4)

Example:

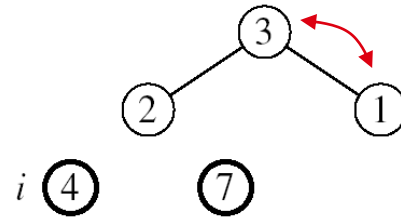
$A=[7, 4, 3, 1, 2]$



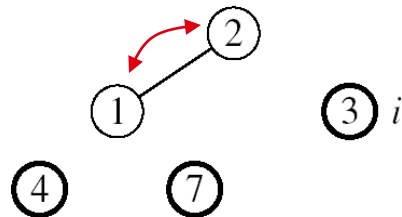
MAX-HEAPIFY(A, 1, 4)



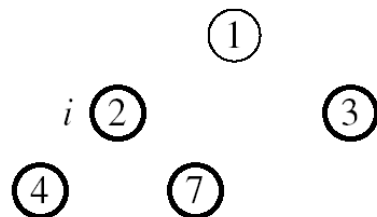
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)



A

1	2	3	4	7
---	---	---	---	---

Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. **for** *i* \leftarrow length[*A*] **downto** 2
3. **do** exchange *A*[1] \leftrightarrow *A*[*i*]
4. MAX-HEAPIFY(*A*, 1, *i* - 1)

$O(\lg n)$ } $n-1$ times

- Running time: $O(n \lg n)$ --- Can be shown to be $\Theta(n \lg n)$

Priority Queue

- Queue – only access element in front
- Queue elements sorted by order of importance
- Implement as a heap where nodes store priority values

Extract Max

- Remove root
- Swap with last node
- Re-heapify

Comparison-based Sorting

- **You want to sort an array of items**
- **You can't access the items' values directly: you can only *compare* two items and find out which is bigger or smaller.**
- Examples: Insertion Sort, MergeSort, QuickSort

“Comparison-based sorting algorithms” are general-purpose.

The worst case complexity of comparison-based sorting can not be reduced more than “ $n \cdot \log n$ ” (Proof in textbook)

Linear-time Sorting

Beyond comparison-based sorting algorithms!

A New Model Of Computation

The elements we're working with have meaningful values.

Before:

arbitrary elements whose values
we could never directly access,
process, or take advantage of
(i.e. we could only interact with
them via comparisons)



Now (examples):

9	18	27	4	9	18	27
---	----	----	---	---	----	----

not-too-large integers

Dec	Feb	Oct	May
-----	-----	-----	-----

months in a year

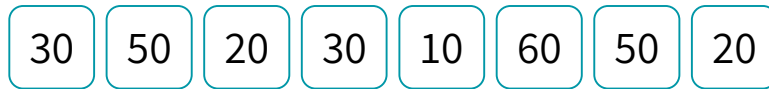
- The worst-case complexity can be reduced further from “ $n \cdot \log n$ ” without making comparisons, called linear sorting. Counting, Radix and Bucket sort are three examples.
- However, it is possible only under restrictive circumstances, for example sorting small integers (exam score), characters etc.

Counting Sort

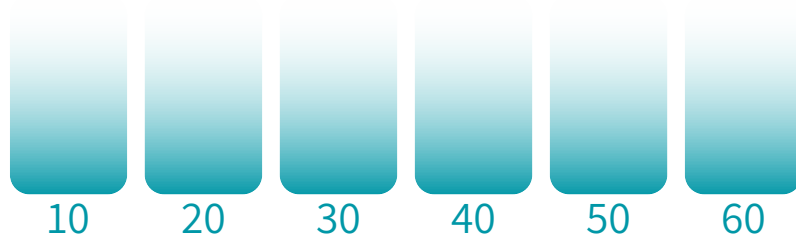
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in $\{10, 20, 30, 40, 50, 60\}$

Input:



Buckets:

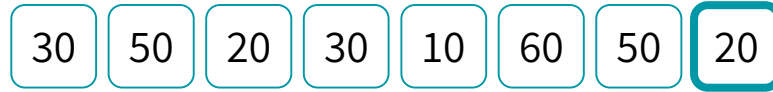


Counting Sort

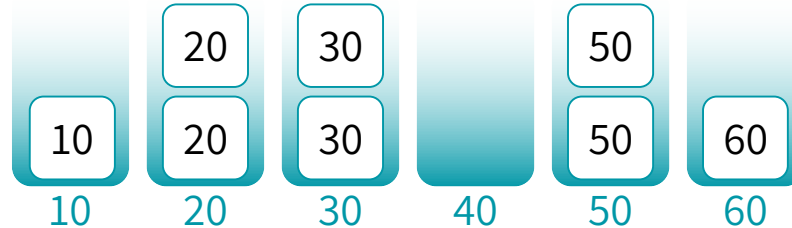
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in $\{10, 20, 30, 40, 50, 60\}$

Input:



Buckets:

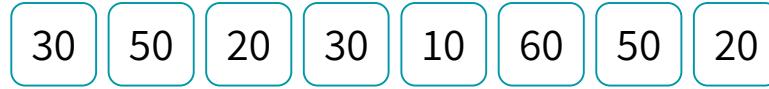


Counting Sort

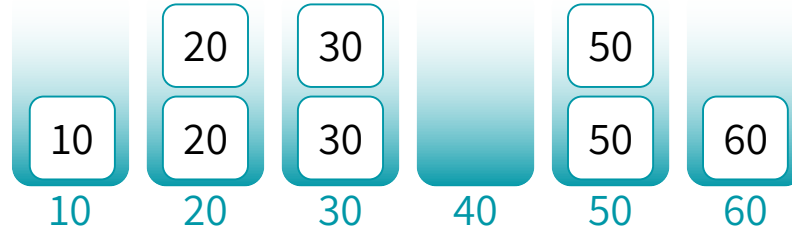
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in $\{10, 20, 30, 40, 50, 60\}$

Input:



Buckets:



Output:



Sorted in time:
 $O(n)$

Counting Sort

- Input: array $A[1, \dots, n]$; k (elements in A have values from 1 to k)
- Output: sorted array A

Algorithm:

1. Create a counter array $C[1, \dots, k]$
2. Create an auxiliary array $B[1, \dots, n]$
3. Scan A once, record element frequency in C
4. Calculate prefix sum in C
5. Scan A in the reverse order, copy each element to B at the correct position according to C .
6. Copy B to A

Counting Sort: Pseudocode

COUNTING-SORT(A, B, k):

1. let $C[1..k]$ be a new array
2. **for** $i = 1$ to k
3. $C[i] = 0$
4. **for** $j = 1$ to $A.length$
5. $C[A[j]] = C[A[j]] + 1$

1. **for** $i = 2$ to k
2. $C[i] = C[i] + C[i - 1]$

3. **for** $j = A.length$ to 1
4. $B[C[A[j]]] = A[j]$
5. $C[A[j]] = C[A[j]] - 1$

1. Create a counter array $C[1, \dots, k]$
2. Create an auxiliary array $B[1, \dots, n]$
3. Scan A once, record element frequency in C
4. Calculate prefix sum in C
5. Scan A in the reverse order, copy each element to B at the correct position according to C.
6. Copy B to A

Analysis of Counting Sort

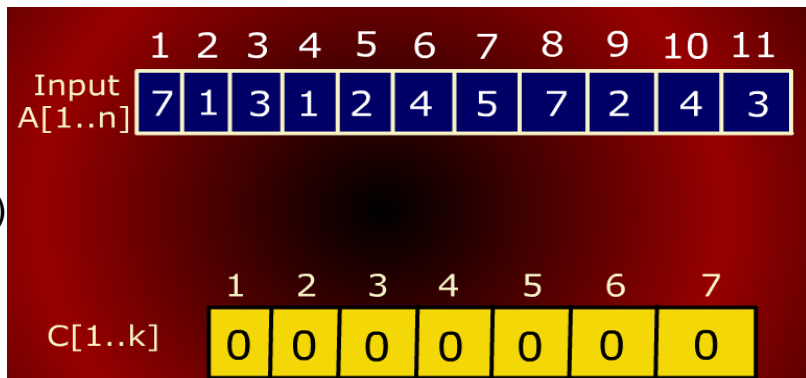
- Input: array $A[1, \dots, n]$; k (elements in A have values from 1 to k)
- Output: sorted array A

Algorithm:

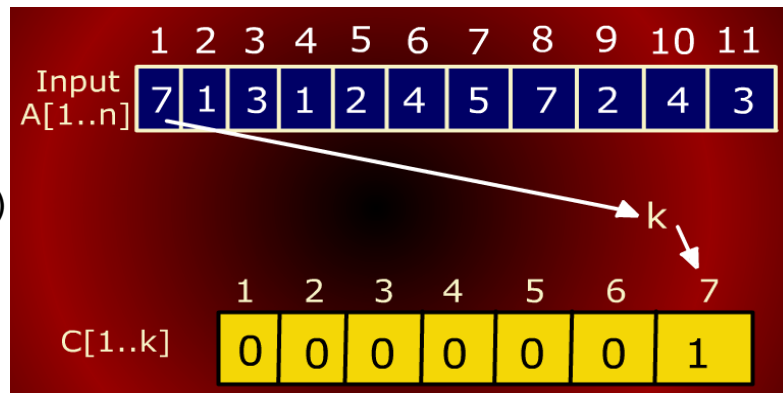
	Time	Space
1. Create a counter array $C[1, \dots, k]$		$O(k)$
2. Create an auxiliary array $B[1, \dots, n]$		$O(n)$
3. Scan A once, record element frequency in C	$O(n)$	
4. Calculate prefix sum in C	$O(k)$	
5. Scan A in the reverse order, copy each element to B at the correct position according to C .	$O(n)$	
6. Copy B to A	$O(n)$	
	$O(n+k)=O(n)$ (if $k=O(n)$)	$O(n+k)=O(n)$ (if $k=O(n)$)

Counting sort

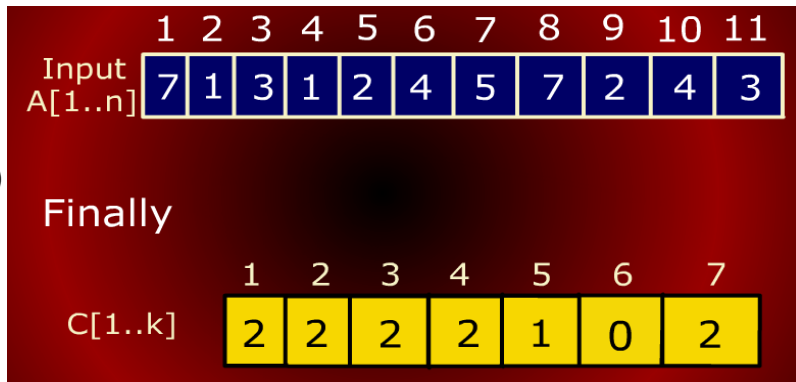
(1)



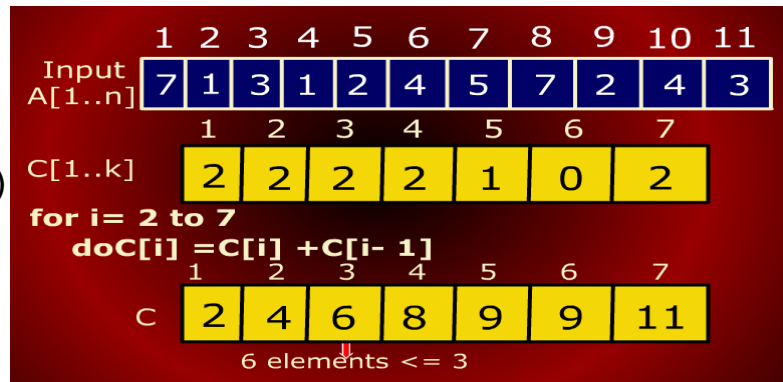
(2)



(3)



(4)

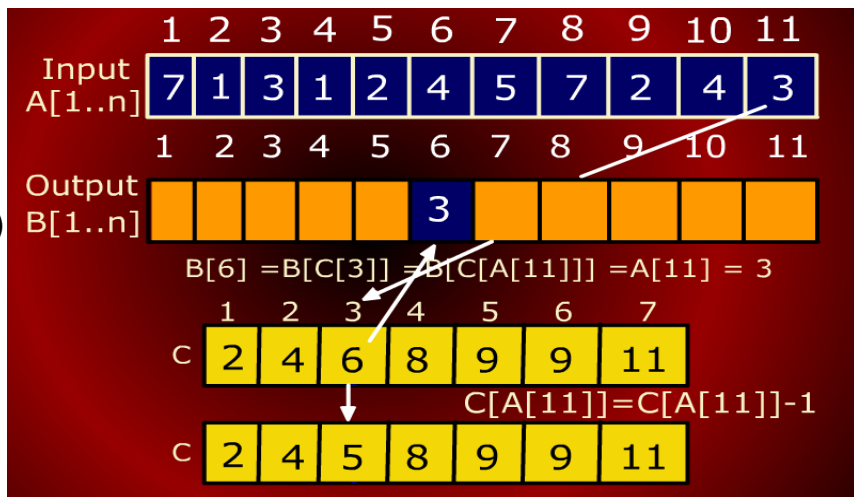


Counting sort

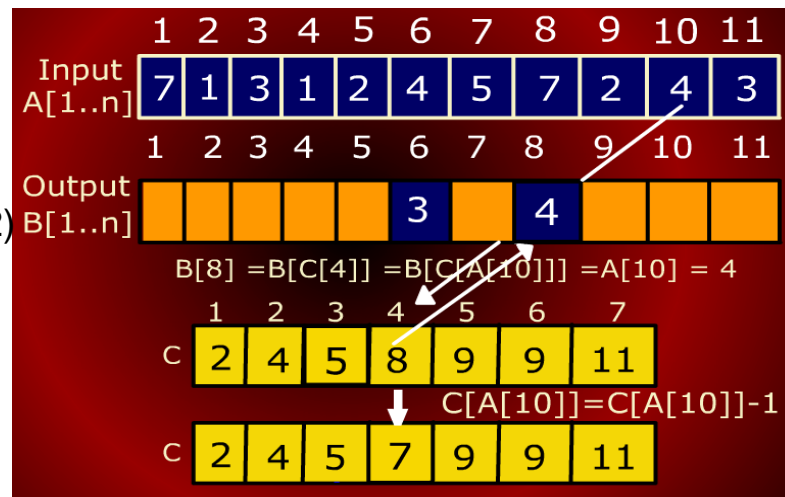
COUNTING-SORT(A, B, k):

1. ...
8. **for** $j = A.length$ to 1
9. $B[C[A[j]]] = A[j]$
10. $C[A[j]] = C[A[j]] - 1$

(1)



(2)

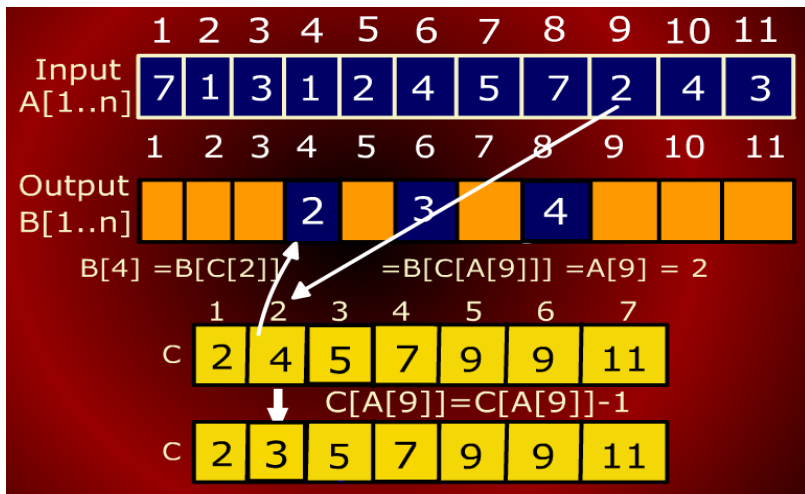


Counting sort

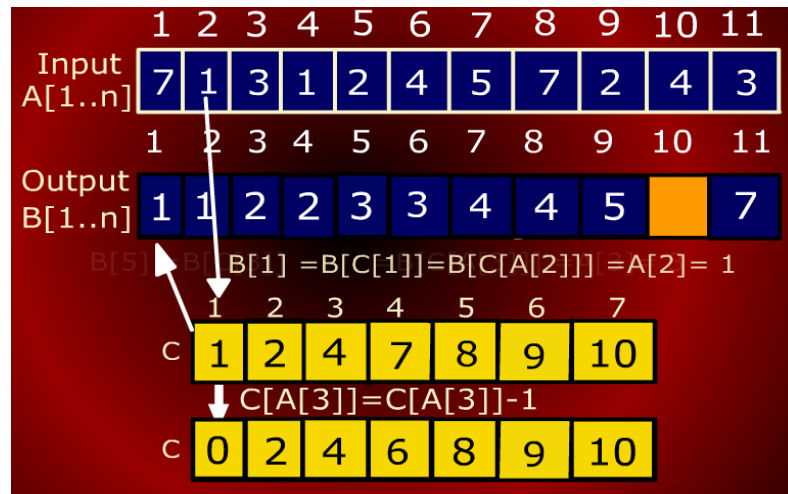
COUNTING-SORT(A, B, k):

1. ...
8. **for** $j = A.length$ to 1
9. $B[C[A[j]]] = A[j]$
10. $C[A[j]] = C[A[j]] - 1$

(3)



(4)



RADIX SORT

A sorting algorithm for integers up to size M
(or more generally, for sorting strings)

RADIX SORT

For sorting integers where the maximum value of any integer is M .
(This can be generalized to lexicographically sorting strings as well)

IDEA:

Perform CountingSort on the least-significant digit first,
then perform CountingSort on the next least-significant, and so on...

Instead of a bucket per possible value, **we just need to maintain a bucket per possible value that a single digit (or character) can take on!**

e.g. 10 buckets labeled 0, 1, ..., 9

RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:



Buckets:



RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:

21 345 13 101 50 234 1

Buckets:



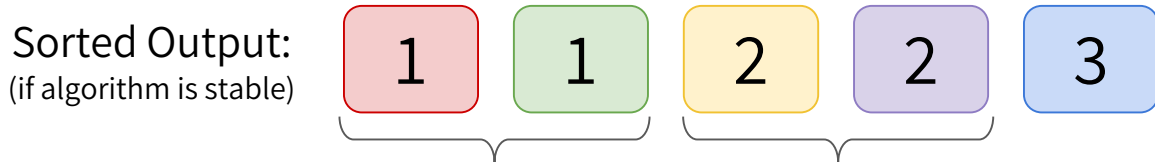
Output:

50 21 101 1 13 234 345

When creating the output list, make sure bucket items exit in FIFO order
(i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

QUICK ASIDE: STABLE SORTING

We say a sorting algorithm is **STABLE** if two objects with equal values appear in the same order in the sorted output as they appear in the input.



The red 1 appeared before the green 1 in the input, so they have to also appear in this order in the output!

The yellow 2 appeared before the purple 2 in the input, so they have to also appear in this order in the output!

RADIX SORT

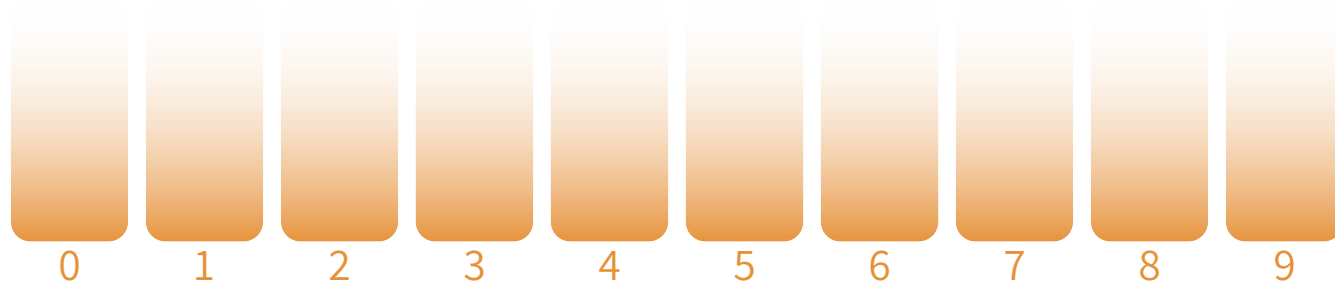
STEP 2: CountingSort on the 2nd least significant digit

Input:

(output from STEP 1)

50 21 101 1 13 234 345

Buckets:



RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50 21 101 01 13 234 345

Buckets:



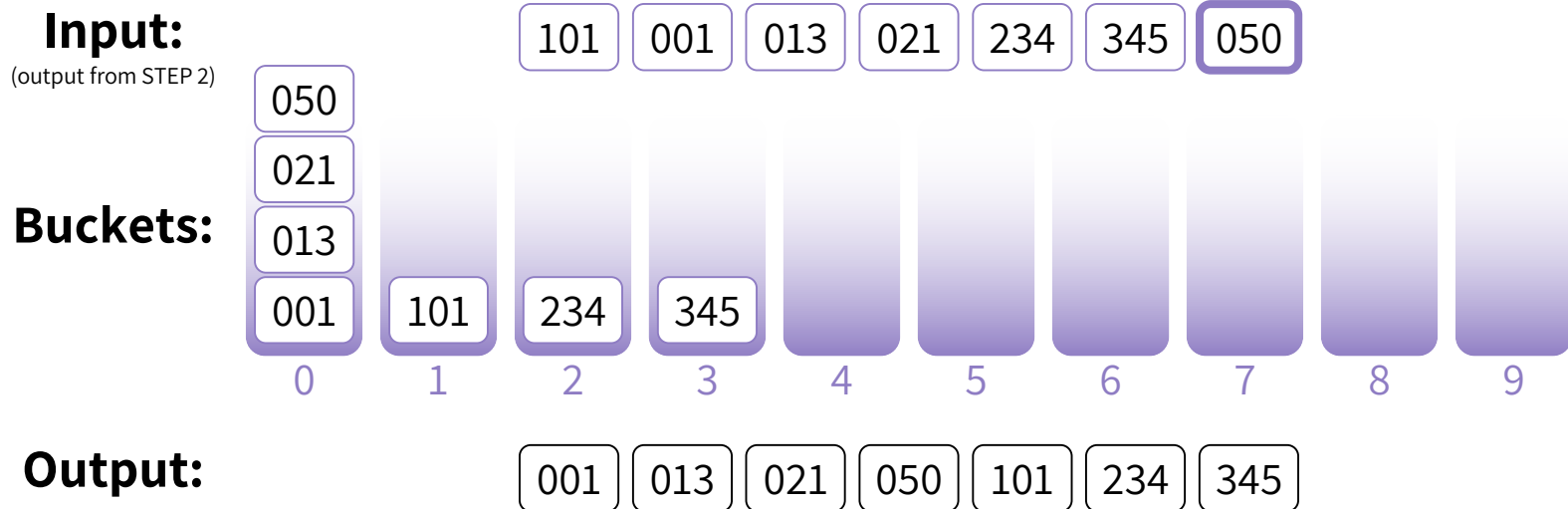
Output:

101 01 13 21 234 345 50

When creating the output list, make sure bucket items exit in FIFO order
(i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit



It worked! But why does it work???

RADIX SORT RUNTIME

Suppose we are sorting n (up-to-) d -digit numbers in base 10 (e.g. $n = 7$, $d = 3$):

21	345	13	101	50	234	1
----	-----	----	-----	----	-----	---

How many iterations are there?

d iterations

How long does each iteration take?

Initialize 10 buckets + put n numbers in 10 buckets \Rightarrow **$O(n)$**

What is the total running time?

$O(nd)$

Comparison of Sorting Algorithms

Algorithm	Worst Time	Extra Memory	Stable
Insertion sort	$O(n^2)$	$O(1)$ (in place)	Yes
Merge sort	$O(n \lg n)$	$O(n)$	Yes
Quick sort	$O(n^2)$	$O(1)$ (in place)	Yes
Heap sort	$O(n \lg n)$	$O(1)$ (in place)	No
Counting sort	$O(n + k)$	$O(n + k)$	Yes