Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

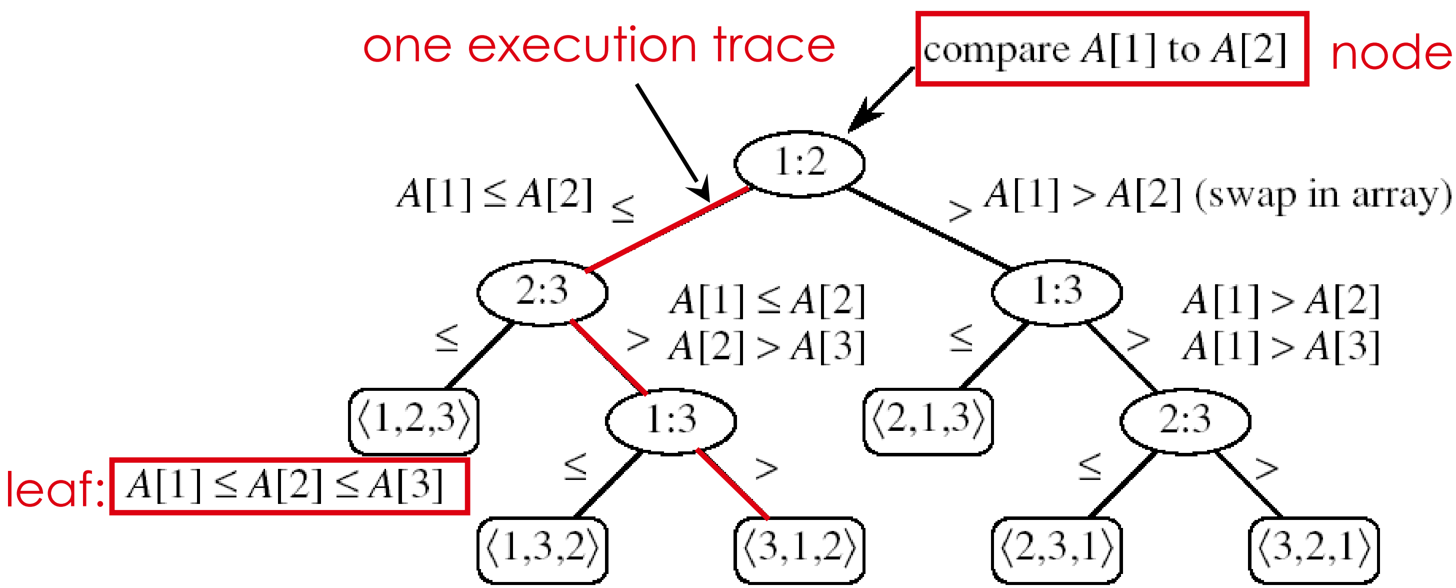
Lecture 11

How Fast Can We Sort?

- Insertion sort, Bubble Sort, Selection Sort $\Theta(n^2)$
- Merge sort $\Theta(n\lg n)$
- Quicksort $\Theta(n\lg n)$
- What is common to all these algorithms?
  - These algorithms sort by making comparisons between the input elements
- To sort n elements, comparison sorts must make $\Omega(n\lg n)$ comparisons in the worst case

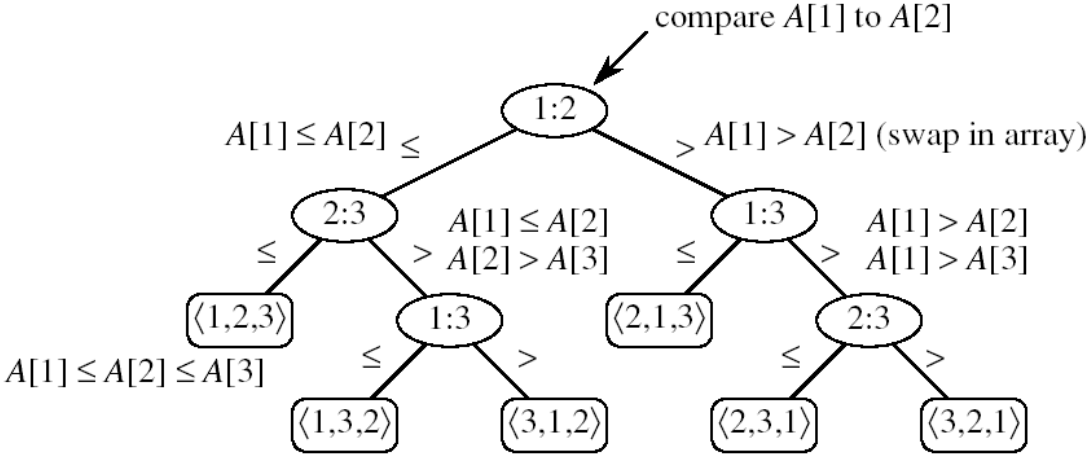
Decision Tree Model

- Represents the comparisons made by a sorting algorithm on an input of a given size: models all possible execution traces
- Control, data movement, other operations are ignored
- Count only the comparisons
- Decision tree for insertion sort on three elements:



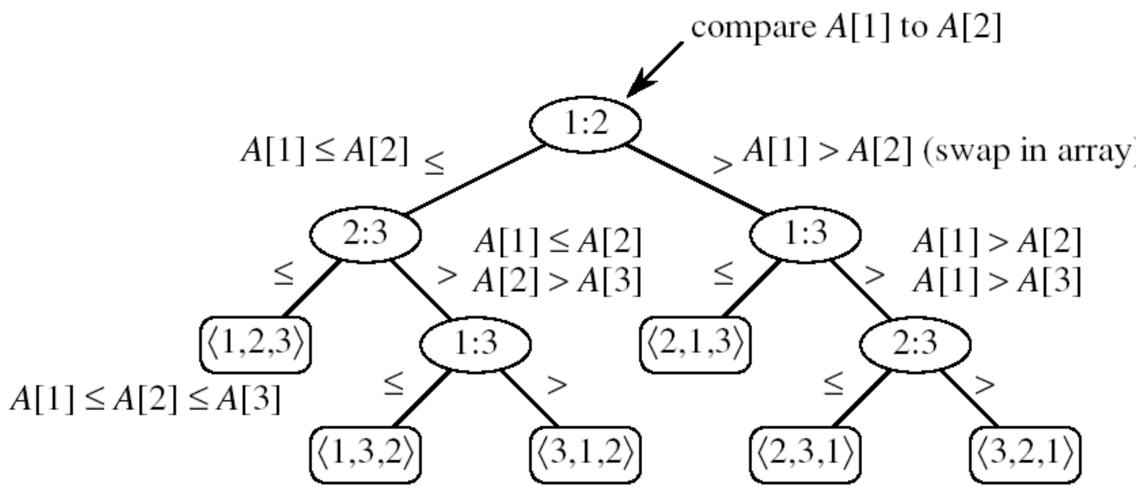
Decision Tree Model

- All permutations on n elements must appear as one of the leaves in the decision tree n! permutations
- Worst-case number of comparisons
  - the length of the longest path from the root to a leaf
  - the height of the decision tree



Decision Tree Model

- Goal: finding a lower bound on the running time on any comparison sort algorithm
  - find a lower bound on the heights of all decision trees for all algorithms



Lemma

- Any binary tree of height h has at most **2^h leaves**

Proof: induction on h

Basis: $h = 0 \Rightarrow$ tree has one node, which is a leaf

$2^h = 1$

Inductive step: assume true for $h-1$

- Extend the height of the tree with one more level
- Each leaf becomes parent to two new leaves

No. of leaves for tree of height h =

= $2 \times$(no. of leaves for tree of height $h-1$)

$\leq 2 \times 2^{h-1}$

$= 2^h$

Lower Bound for Comparison Sorts

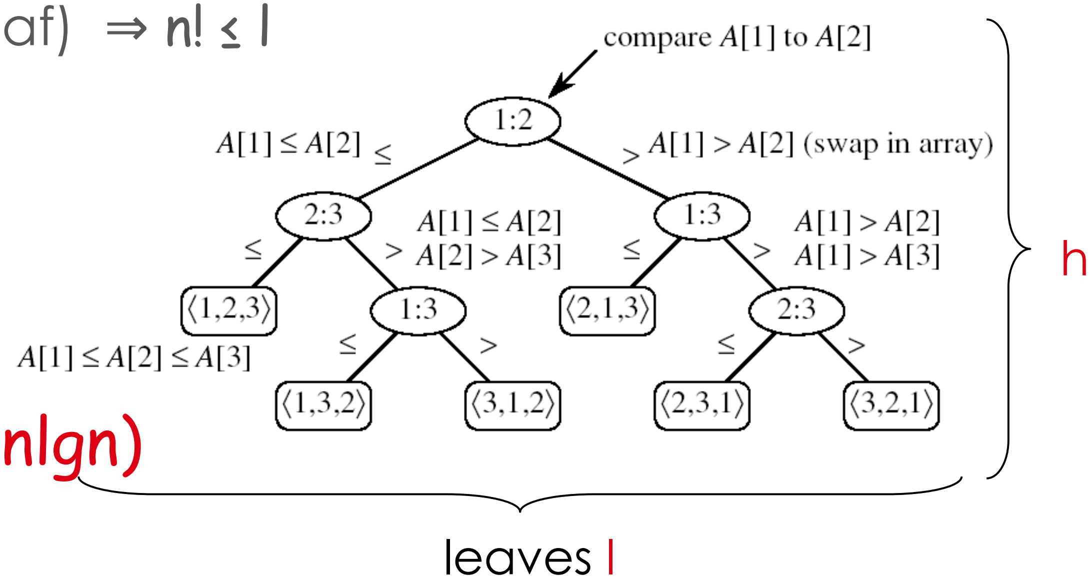
Theorem: Any comparison sort algorithm requires **Ω**(nlgn) comparisons in the worst case.

Proof: How many leaves does the tree have?

- At least n! (each of the **n!** permutations of the input appears as some leaf) $\Rightarrow n! \le l$
- At most 2^h leaves

$\Rightarrow \quad n! \le l \le 2^h$

$\Rightarrow \quad h \ge \lg(n!) = \Omega(n\lg n)$



We can beat the **Ω**(nlgn) running time if we use other operations than comparisons!

Counting Sort

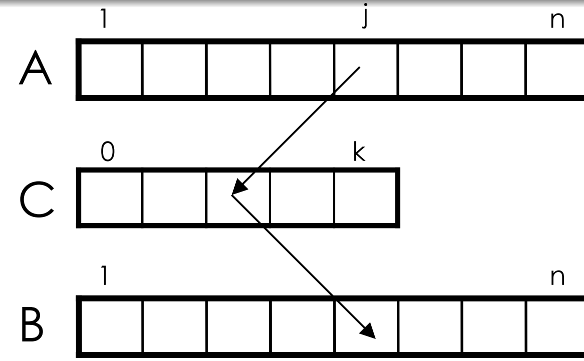
- Assumption:
  - The elements to be sorted are integers in the range **0** to **k**
- Idea:
  - Determine for each input element **x**, the number of elements smaller than **x**
  - Place element **x** into its correct position in the output array

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

COUNTING-SORT



Alg.: COUNTING-SORT(A, B, n, k)

```
for i ← 0 to k
    do C[ i ] ← 0
for j ← 1 to n
    do C[A[ j ]] ← C[A[ j ]] + 1
▷ C[i] contains the number of elements equal to i
for i ← 1 to k
    do C[ i ] ← C[ i ] + C[i -1]
▷ C[i] contains the number of elements ≤ i
for j ← n downto 1
    do B[C[A[ j ]]] ← A[ j ]
        C[A[ j ]] ← C[A[ j ]] - 1
```

Example

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

B

1	2	3	4	5	6	7	8
						3	

C

0	1	2	3	4	5
2	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0					3	

C

0	1	2	3	4	5
1	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0				3	3	

C

0	1	2	3	4	5
1	2	4	5	7	8

B

1	2	3	4	5	6	7	8
	0		2		3	3	

C

0	1	2	3	4	5
1	2	3	5	7	8

Example (cont.)

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0		2		3	3	

	0	1	2	3	4	5
C	0	2	3	5	7	8

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	5

	0	1	2	3	4	5
C	0	2	3	4	7	7

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	

	0	1	2	3	4	5
C	0	2	3	4	7	8

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Analysis of Counting Sort

Alg.: COUNTING-SORT(A, B, n, k)

```
for i ← 0 to k                              Θ(k)
    do C[ i ] ← 0
for j ← 1 to n                              Θ(n)
    do C[A[ j ]] ← C[A[ j ]] + 1
▷C[i] contains the number of elements equal to i
for i ← 1 to k                              Θ(k)
    do C[ i ] ← C[ i ] + C[i -1]
▷C[i] contains the number of elements ≤ i
for j ← n downto 1                          Θ(n)
    do B[C[A[ j ]]] ← A[ j ]
       C[A[ j ]] ← C[A[ j ]] - 1
```

Overall time: $\Theta(n + k)$

Analysis of Counting Sort

- Overall time: $\Theta(n + k)$
- In practice we use COUNTING sort when $k = O(n)$

  $\Rightarrow$ running time is $\Theta(n)$

- Counting sort is **stable**
  - Numbers with the same value appear in the same order in the output array
  - Important when additional data is carried around with the sorted keys

Radix Sort

- Considers keys as numbers in a base-k number
  - A **d**-digit number will occupy a field of **d** columns
- Sorting looks at one column at a time
  - For a **d** digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit, until all digits have been sorted
  - Requires only **d** passes through the list

326
453
608
835
751
435
704
690

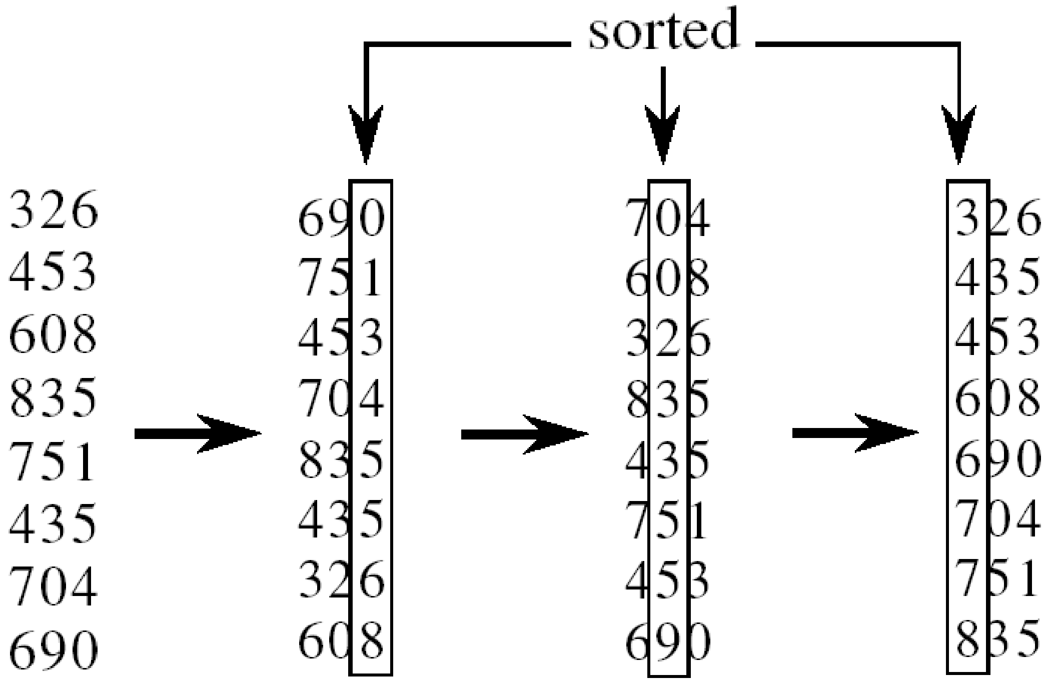
RADIX-SORT

Alg.: RADIX-SORT(*A*, d)

for *i* ← 1 **to** d

do use a stable sort to sort array *A* on digit *i*

- 1 is the lowest order digit, d is the highest-order digit

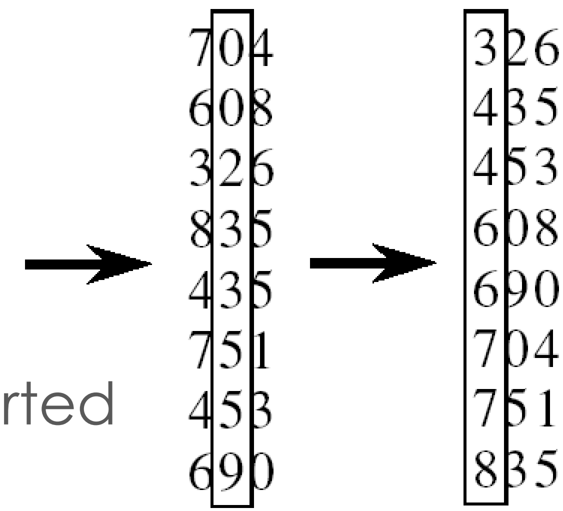


Analysis of Radix Sort

- Given **n** numbers of **d** digits each, where each digit may take up to **k** possible values, RADIX-SORT correctly sorts the numbers in **$\Theta(d(n+k))$**
  - One pass of sorting per digit takes **$\Theta(n+k)$** assuming that we use counting sort
  - There are **d** passes (for each digit)

Correctness of Radix sort

- We use induction on the number **d** of passes through the digits
- **Basis:** If **d** = 1, there's only one digit, trivial
- **Inductive step:** assume digits 1, 2, . . . , **d-1** are sorted
  - Now sort on the **d**-th digit
  - If $a_d < b_d$, sort will put **a** before **b**: correct
    a < **b** regardless of the low-order digits
  - If $a_d > b_d$, sort will put **a** after **b**: correct
    a > **b** regardless of the low-order digits
  - If $a_d = b_d$, sort will leave **a** and **b** in the same order and **a** and **b** are already sorted on the low-order **d-1** digits



Bucket Sort

- Assumption:
  - the input is generated by a random process that distributes elements uniformly over [0, 1)
- Idea:
  - Divide [0, 1) into n equal-sized buckets
  - Distribute the n input values into the buckets
  - Sort each bucket
  - Go through the buckets in order, listing elements in each one

- **Input:** $A[1 \,..\, n]$, where $0 \le A[i] < 1$ for all i
- **Output:** elements in A sorted
- **Auxiliary array:** $B[0 \,..\, n - 1]$ of linked lists, each list initially empty

BUCKET-SORT

Alg.: BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n

do insert A[i] into list $B[\lfloor nA[i] \rfloor]$

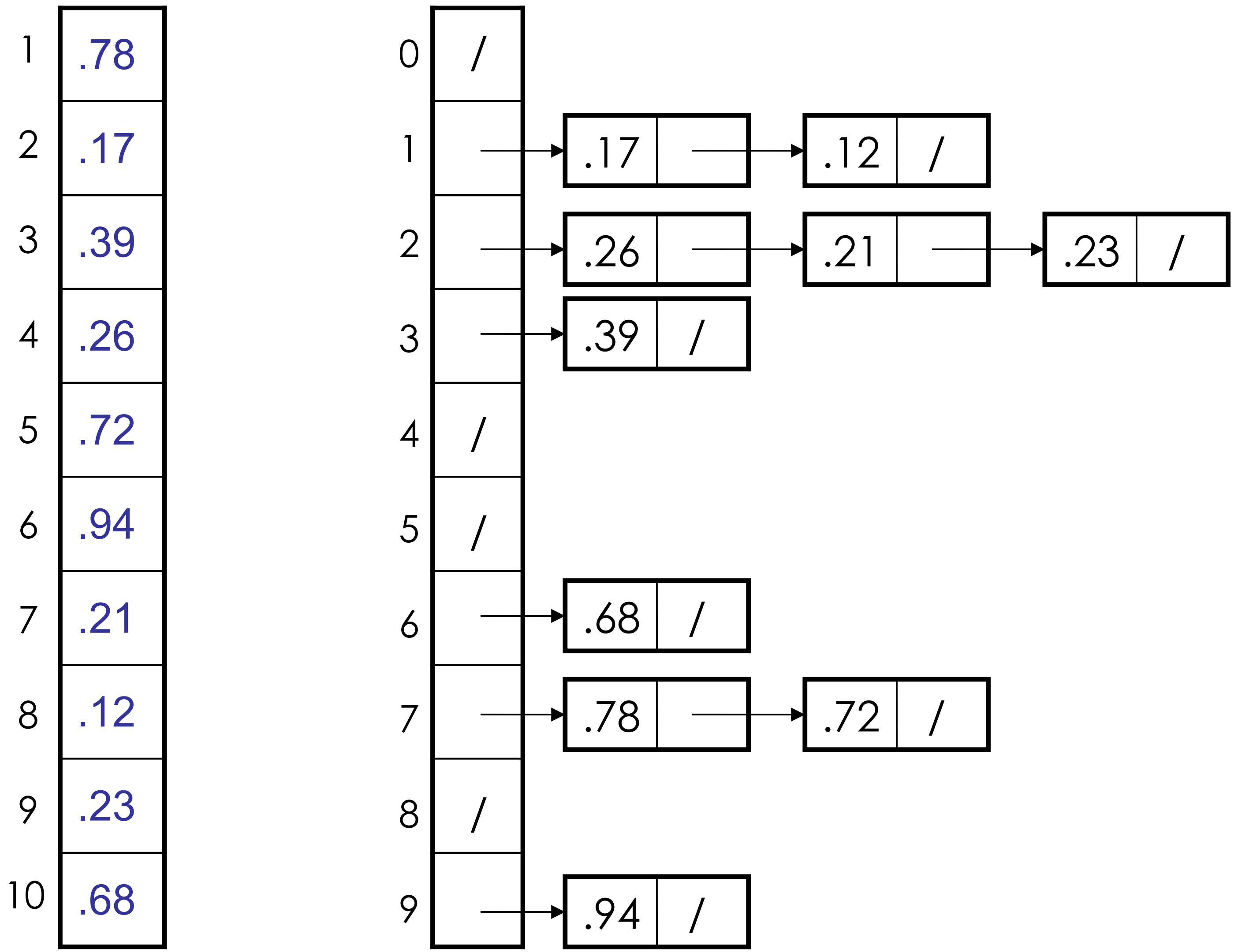
for $i \leftarrow 0$ **to** $n - 1$

do sort list $B[i]$ with insertion sort

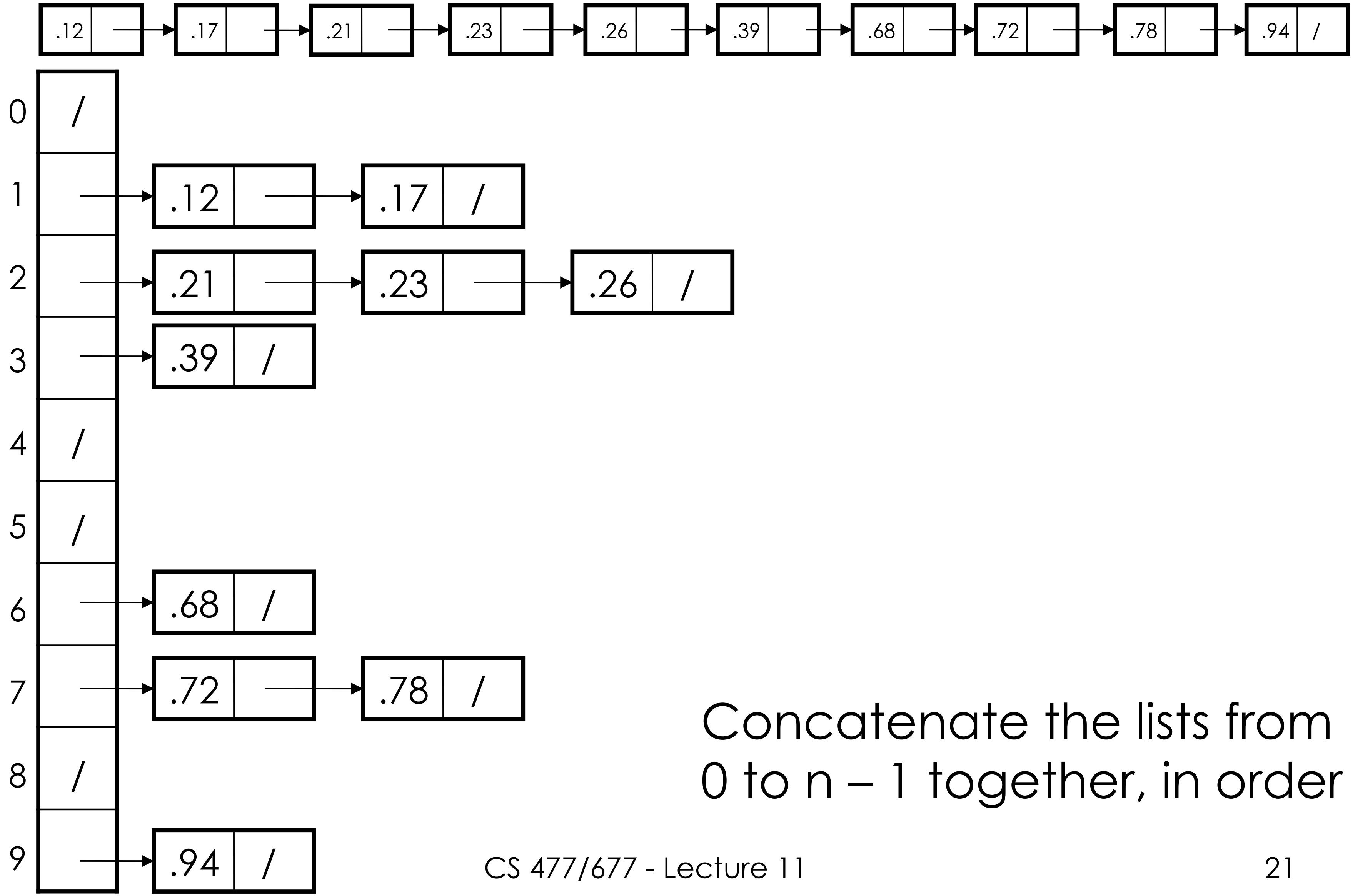
concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order

return the concatenated lists

Example - Bucket Sort



Example - Bucket Sort



Concatenate the lists from 0 to n – 1 together, in order

Correctness of Bucket Sort

- Consider two elements $A[i]$, $A[j]$
- Assume without loss of generality that $A[i] \le A[j]$
- Then $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$
  - $A[i]$ belongs to the same group as $A[j]$ or to a group with a lower index than that of $A[j]$
- If $A[i]$, $A[j]$ belong to the same bucket:
  - insertion sort puts them in the proper order
- If $A[i]$, $A[j]$ are put in different buckets:
  - concatenation of the lists puts them in the proper order

Analysis of Bucket Sort

Alg.: BUCKET-SORT(A, n)

for i ← 1 **to** n
  do insert A[i] into list B[⌊nA[i]⌋] **O**(n)

for i ← 0 **to** n - 1
  do sort list B[i] with insertion sort Θ(n)

concatenate lists B[0], B[1], . . . , B[n -1]
together in order **O**(n)

return the concatenated lists

Θ(n)

Conclusion

- Any comparison sort will take at least **nlgn** to sort an array of **n** numbers
- We can achieve a better running time for sorting if we can make certain assumptions on the input data:
  - **Counting sort:** each of the n input elements is an integer in the range **0** to **k**
  - **Radix sort:** the elements in the input are integers represented with **d** digits
  - **Bucket sort:** the numbers in the input are uniformly distributed over the interval **[0, 1)**

A Job Scheduling Application

- Job scheduling
  - The key is the priority of the jobs in the queue
  - The job with the highest priority needs to be executed next
- Operations
  - Insert, remove maximum
- Data structures
  - **Priority queues**
  - Ordered array/list, unordered array/list

PQ Implementations & Cost

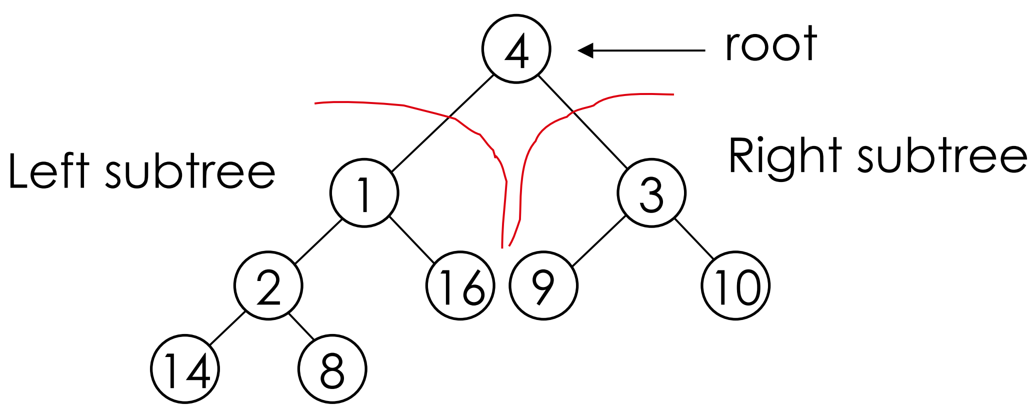
Worst-case asymptotic costs for a PQ with N items

	Insert	Remove max
ordered array	N	1
ordered list	N	1
unordered array	1	N
unordered list	1	N

Can we implement both operations efficiently?

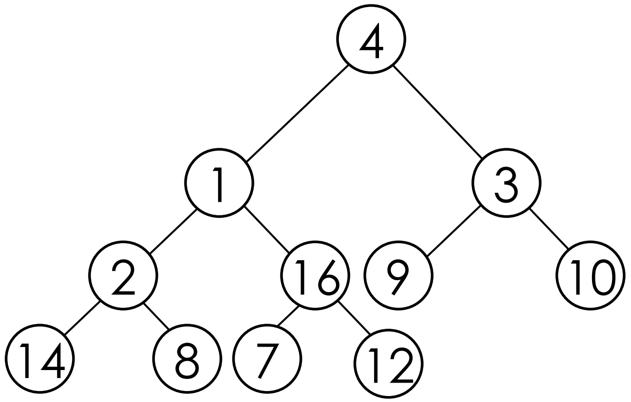
Background on Trees

- *Def:* Binary tree = structure composed of a finite set of nodes that either:
  - Contains no nodes, or
  - Is composed of three disjoint sets of nodes: a **root** node, a **left subtree** and a **right subtree**



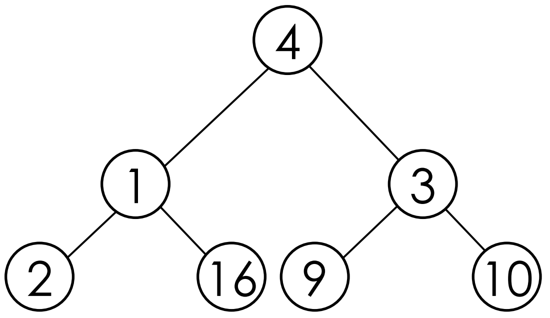
Special Types of Trees

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree (number of children) exactly 2.



Full binary tree

- *Def:* **Complete binary tree** = a binary tree in which all leaves have the same depth and all internal nodes have degree 2.

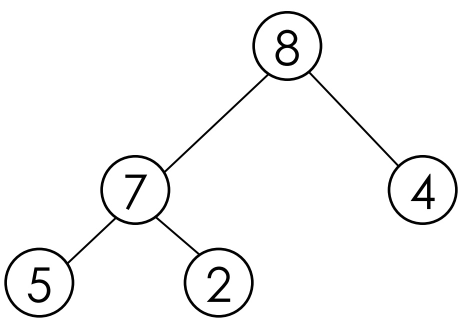


Complete binary tree

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

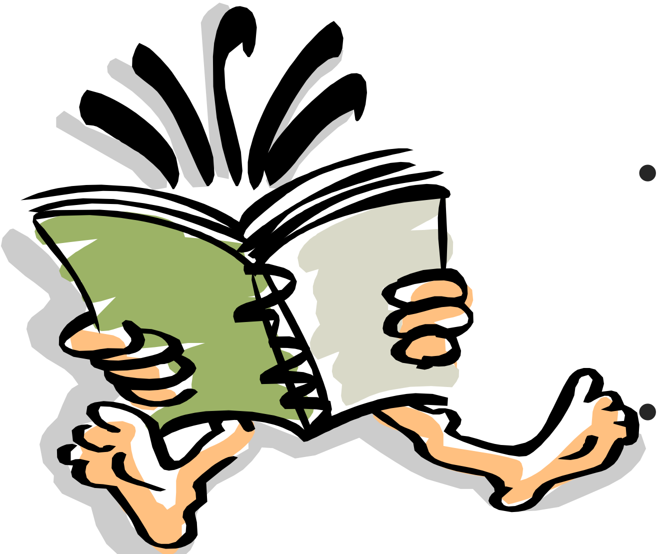
$$Parent(x) \geq x$$



Heap

It doesn't matter that 4 in level 1 is smaller than 5 in level 2

Readings



- For this lecture
  - Section 8.3, 8.4
  - Chapter 6
- Coming next
  - Chapter 13