

Data Structures in Java

Session 20

Instructor: Bert Huang

<http://www.cs.columbia.edu/~bert/courses/3134>

Announcements

- Homework 5 due 11/24
- Homework 4 solutions posted

Review

- Review Disjoint Set ADT
- Start Discussion of Sorting
 - Lower bound
 - Breaking the lower bound
 - Radix Sort (Trie, Counting Sort)

Today's Plan

- Radix Sort specifics
- Comparison sorting algorithm characteristics
- Algorithms: Selection Sort, Insertion Sort, Shellsort, Heapsort, Mergesort, Quicksort

Radix Sort with Least Significant Digit

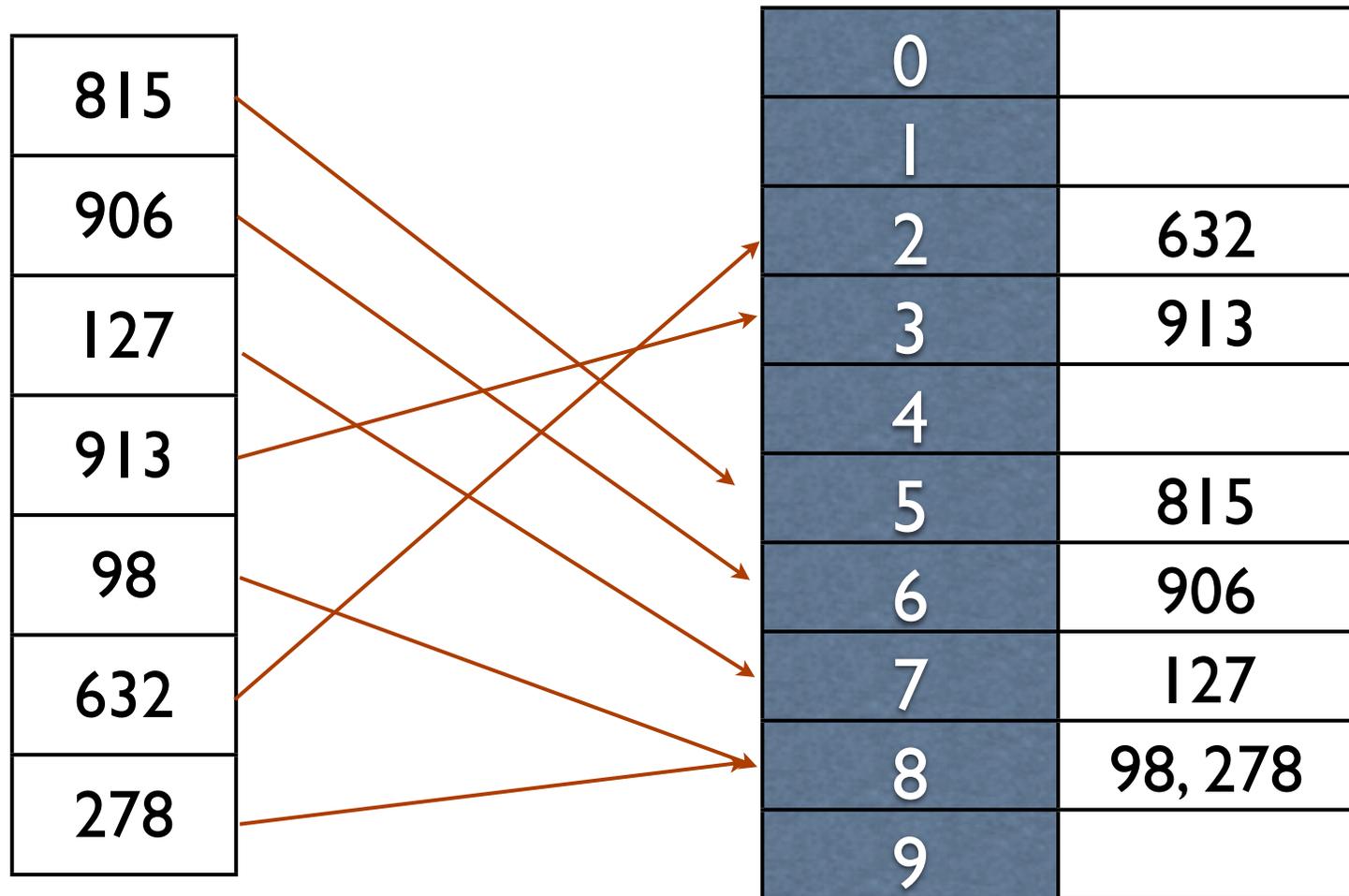
- CountingSort according to the least significant digit
- Repeat: CountingSort according to the next least significant digit
- Each step must be **stable**
- Running time: **$O(Nk)$** for maximum of **k** digits
- Space: **$O(N+b)$** for base- **b** number system*

Radix Sort Example

815
906
127
913
98
632
278

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Radix Sort Example

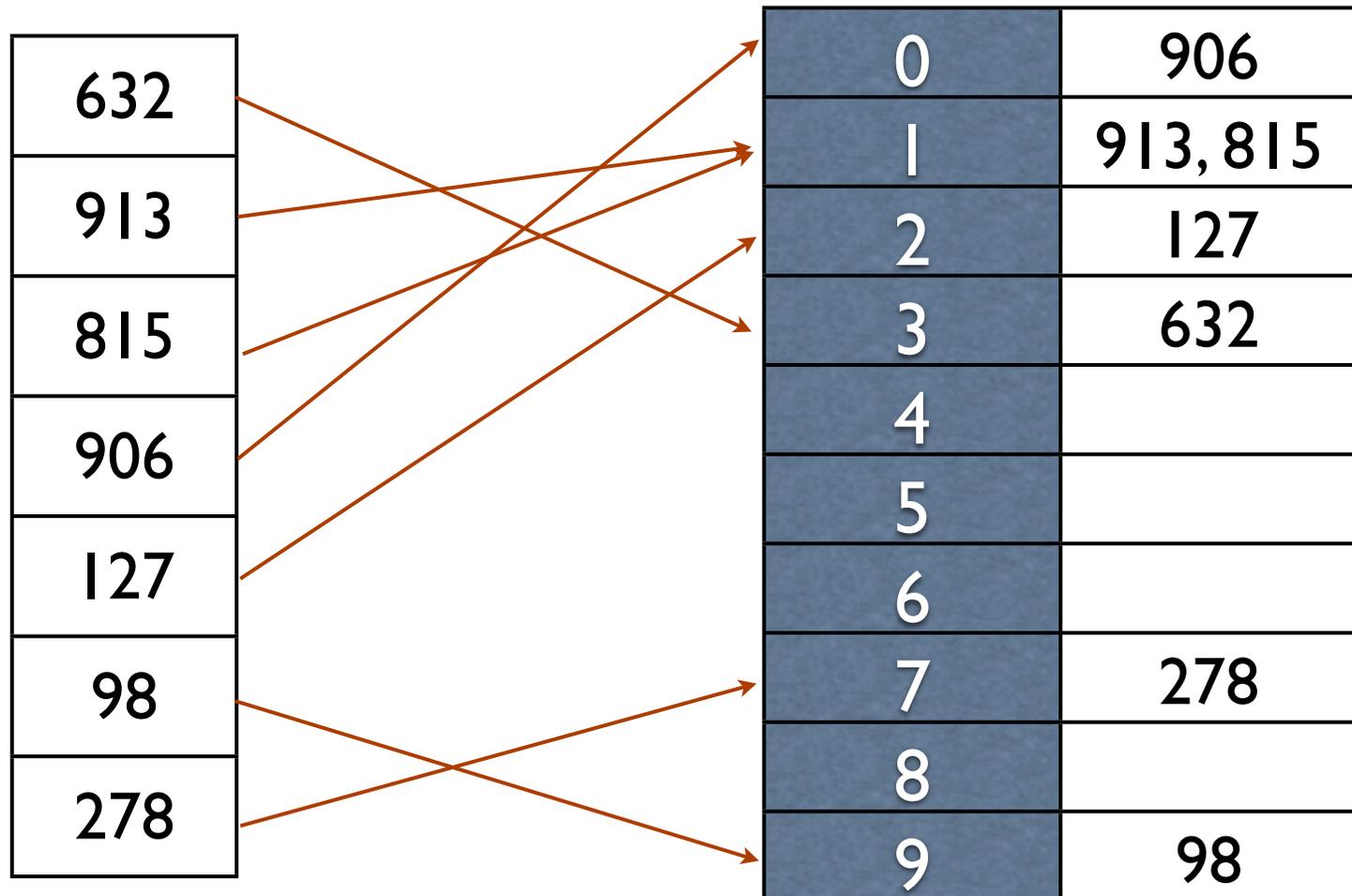


Radix Sort Example

632
913
815
906
127
98
278

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Radix Sort Example

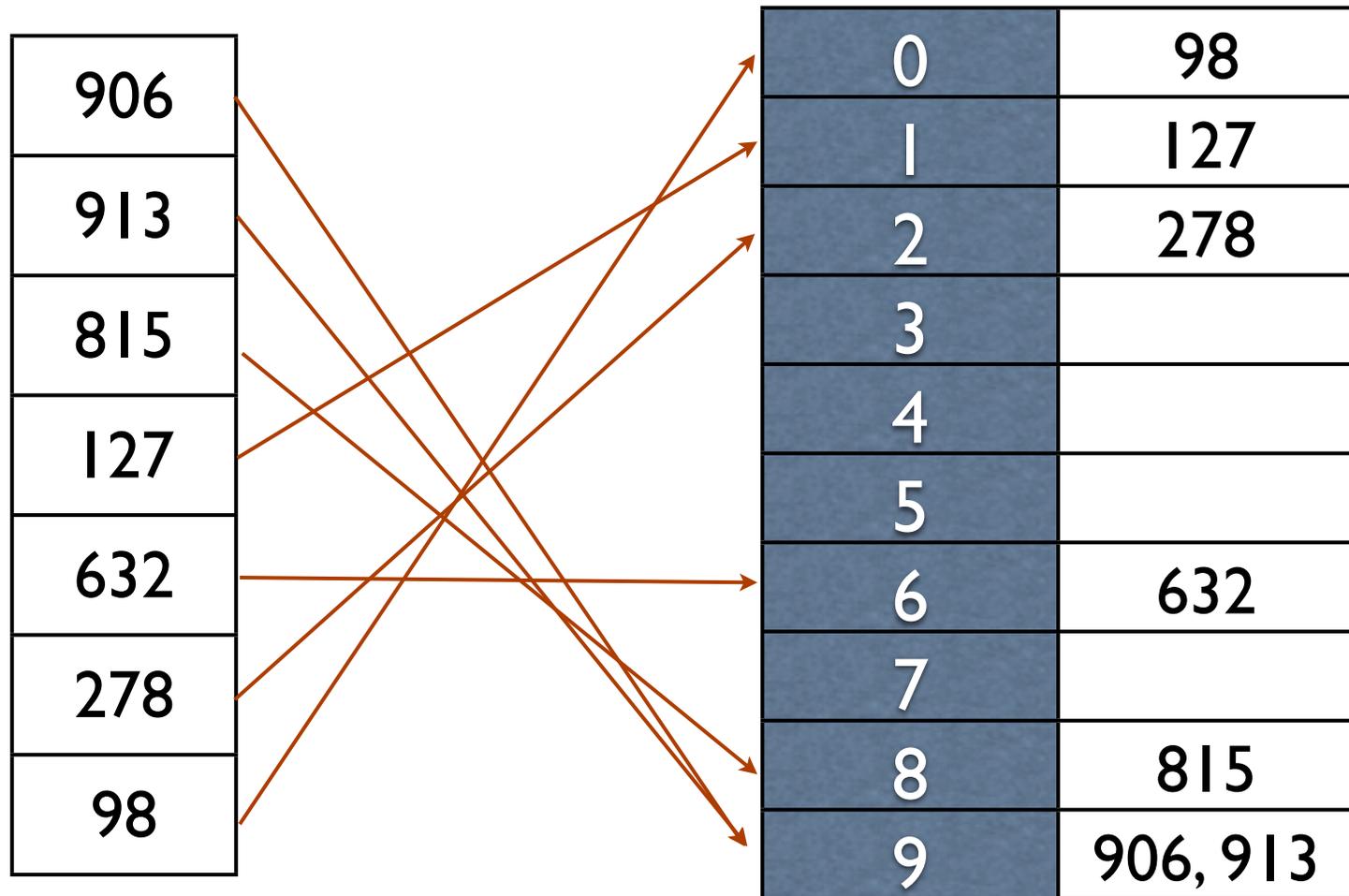


Radix Sort Example

906
913
815
127
632
278
98

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Radix Sort Example



Analysis

- For maximum of **k** digits (in whatever base), we need **k** passes through the array, **$O(Nk)$**
- For base-**b** number system, we need **b** queues, which will end up containing **N** elements total, so **$O(N+b)$** space
- Stable because if elements are the same, they keep being enqueued and dequeued in the same order

Comparison Sorts

- Of course, Radix Sort only works well for sorting keys representable as digital numbers
- In general, we must often use comparison sorts
- We have proven a $\Omega(N \log N)$ lower bound for running time
- But algorithms also have other desirable characteristics

Sorting Algorithm Characteristics

- Worst case running time
- Worst case space usage (can it run in place?)
- Stability
- Average running time/space
- (simplicity)
- (Best case running time/space usage)

Preview

	Worst Case Time	Average Time	Space	Stable?
Selection	$O(N^2)$	$O(N^2)$	$O(1)$	No
Insertion	$O(N^2)$	$O(N^2)$	$O(1)$	Yes
Shell	$O(N^{3/2})$?	$O(1)$	No
Heap	$O(N \log N)$	$O(N \log N)$	$O(1)$	No
Merge	$O(N \log N)$	$O(N \log N)$	$O(N)/O(1)$	Yes/No
Quick	$O(N^2)$	$O(N \log N)$	$O(\log N)$	No

Selection Sort

- Swap least unsorted element with first unsorted element
- Unstable if in place
- Running time $O(N^2)$
- In place $O(1)$ space
- Algorithm Animation

Insertion Sort

- Assume first **p** elements are sorted. Insert **(p+1)**'th element into appropriate location.
 - Save **A[p+1]** in temporary variable **t**, shift sorted elements greater than **t**, and insert **t**
- Stable
- Running time $O(N^2)$
- In place **O(1)** space

Insertion Sort Analysis

- When the sorted segment is i elements, we may need up to i shifts to insert the next element

$$\sum_{i=2}^N i = N(N-1)/2 - 1 = O(N^2)$$

- Stable because elements are visited in order and equal elements are inserted after its equals
- Algorithm Animation

Shellsort

- Essentially splits the array into subarrays and runs Insertion Sort on the subarrays
- Uses an increasing sequence, h_1, \dots, h_t , such that $h_1 = 1$.
- At phase **k**, all elements h_k apart are sorted; the array is called h_k -sorted
- for every **i**, $A[i] \leq A[i + h_k]$

Shell Sort

Correctness

- Efficiency of algorithm depends on that elements sorted at earlier stages remain sorted in later stages
- Unstable. Example: 2-sort the following: [5 5 1]

Increment Sequences

- Shell suggested the sequence $h_t = \lfloor N/2 \rfloor$ and $h_k = \lfloor h_{k+1}/2 \rfloor$, which was suboptimal
- A better sequence is $h_k = 2^k - 1$
- Using better sequence sorts in $\Theta(N^{3/2})$
- Often used for its simplicity and sub-quadratic time, even though **$O(N \log N)$** algorithms exist
- Animation

Heapsort

- Build a **max** heap from the array: **$O(N)$**
- call deleteMax **N** times: **$O(N \log N)$**
- **$O(1)$** space
- Simple if we abstract heaps
- Unstable
- Animation

Mergesort

- Quintessential divide-and-conquer example
- Mergesort each half of the array, merge the results
- Merge by iterating through both halves, compare the current elements, copy lesser of the two into output array
- Animation

Mergesort

Recurrence

- Merge operation is costs $O(N)$
- $T(N) = 2 T(N/2) + N$
- A few ways to solve this recurrence, i.e., visualizing equation as a tree

$$= \sum_{i=0}^{\log N} 2^i c \frac{N}{2^i}$$

$$= \sum_{i=0}^{\log N} cN = cN \log N$$

Quicksort

- Choose an element as the **pivot**
- Partition the array into elements greater than pivot and elements less than pivot
- Quicksort each partition
- Animation

Choosing a Pivot

- The worst case for Quicksort is when the partitions are of size zero and **$N-1$**
- Ideally, the pivot is the median, so each partition is about half
- If your input is random, you can choose the first element, but this is very bad for presorted input!
- Choosing randomly works, but a better method is...

Median-of-Three

- Choose three entries, use the median as pivot
- If we choose randomly, $2/N$ probability of worst case pivots
- Median-of-three gives **0** probability of worst case, tiny probability of 2nd-worst case. (Approx. $2/N^3$)
- Randomness less important, so choosing (first, middle, last) works reasonably well

Partitioning the Array

- Once pivot is chosen, swap pivot to end of array. Start counters $i=1$ and $j=N-1$
- Intuition: i will look at less-than partition, j will look at greater-than partition
- Increment i and decrement j until we find elements that don't belong ($A[i] > \text{pivot}$ or $A[j] < \text{pivot}$)
- Swap ($A[i]$, $A[j]$), continue increment/decrements
- When i and j touch, swap pivot with $A[j]$

Quicksort Worst Case

- Running time recurrence includes the cost of partitioning, then the cost of 2 quicksorts
- We don't know the size of the partitions, so let i be the size of the first partition
- $T(N) = T(i) + T(N-i-1) + N$
- Worst case is $T(N) = T(N-1) + N$

Quicksort Properties

- Unstable
- Average time $O(N \log N)$
- Worst case time $O(N^2)$

Summary

	Worst Case Time	Average Time	Space	Stable?
Selection	$O(N^2)$	$O(N^2)$	$O(1)$	No
Insertion	$O(N^2)$	$O(N^2)$	$O(1)$	Yes
Shell	$O(N^{3/2})$?	$O(1)$	No
Heap	$O(N \log N)$	$O(N \log N)$	$O(1)$	No
Merge	$O(N \log N)$	$O(N \log N)$	$O(N)/O(1)$	Yes/No
Quick	$O(N^2)$	$O(N \log N)$	$O(\log N)$	No

Reading

- <http://www.sorting-algorithms.com/>
- Weiss Chapter 7
 - Skim 7.4.1 (proof of Shell Sort)