EECS 2011M: Fundamentals of Data Structures

Instructor: Suprakash Datta

Office : LAS 3043

Course page: http://www.eecs.yorku.ca/course/2011M

Also on Moodle

Note: Some slides in this lecture are adopted from James Elder' s and the authors' slides.

Linear Time Sorting

1. We can do better than the lower bound if the algorithm is not comparison-based
2. We can sort using information other than comparisons between data items if we restrict the scope of the problem
3. For some restricted scenarios, we can sort in worse-case linear time

Bucket Sort

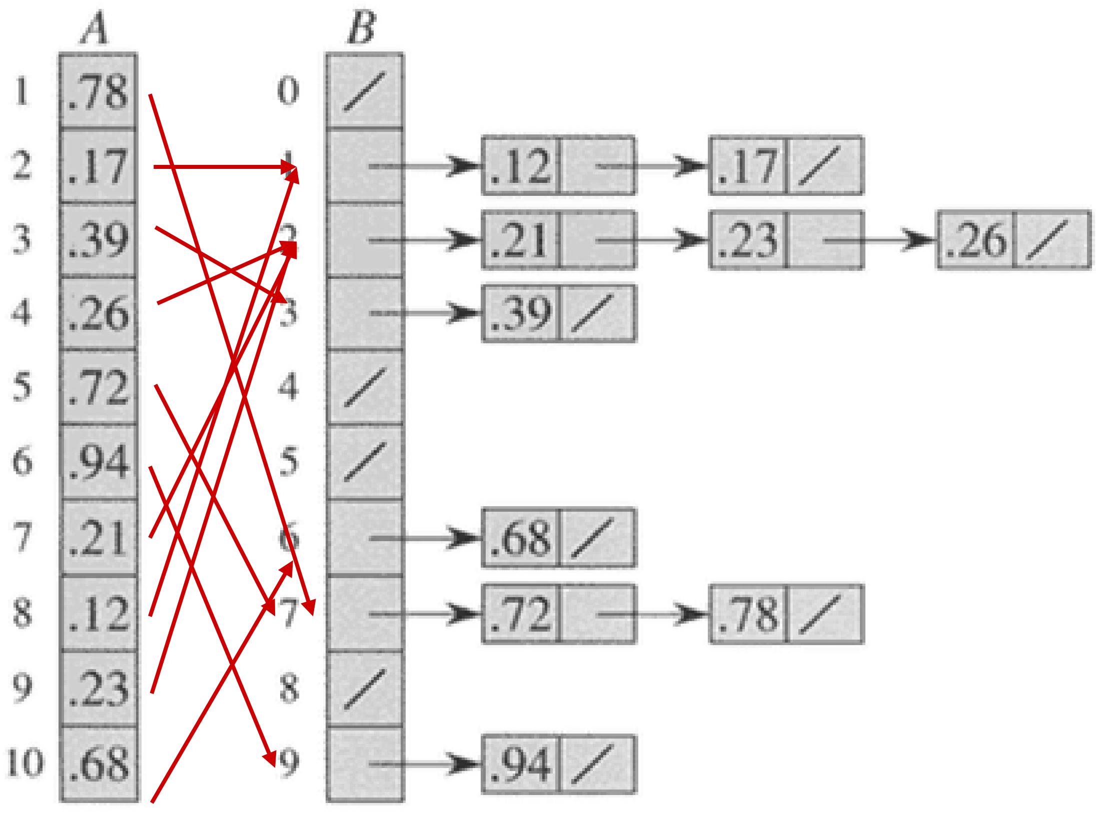
1. Suppose all keys come from a finite interval, say [0,1)
2. We can define buckets for ranges, e.g. [0,0.1),[0.1,0.2),…,[0.9,1)
3. Insert keys in appropriate bucket
4. If input is random and uniformly distributed, **expected** run time is Θ(n).

Bucket Sort - Illustration

Given A[1..n]:

Create new table B of length n

Insert A[i] into $B\left[\left\lfloor nA[i] \right\rfloor\right]$



Bucket Sort - Pseudocode

$\text{BUCKET-SORT}(A, n)$

<u>Expected</u> Running Time

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

do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$ $\longleftarrow \Theta(1) \times n$

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

do sort list $B[i]$ with insertion sort $\longleftarrow \Theta(1) \times n$

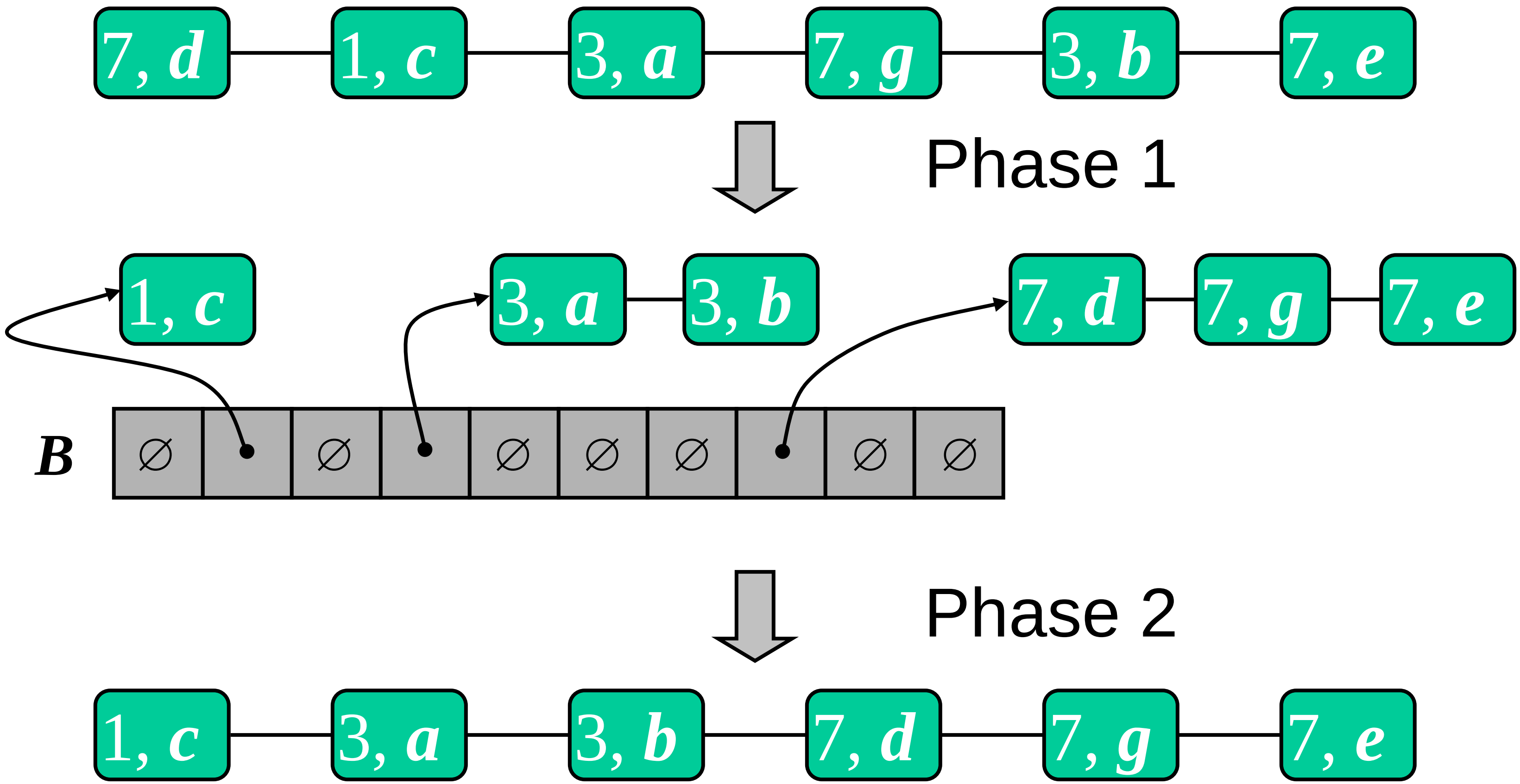
concatenate lists $B[0], B[1], \ldots, B[n-1]$ $\longleftarrow \Theta(n)$

return the concatenated lists

$\Theta(n)$

Bucket Sort – Example

Key range [0, 9]



Bucket Sort – Properties and Extensions

1. Stable Sort
2. Keys must be numbers -- since they are used to generate array indices
3. Extension: Set of fixed keys like the set of names of 50 US states – Sort the keys and give each key its unique bucket. Insert each item into the bracket corresponding to its key
4. What if input numbers are NOT uniformly distributed?
5. What if the distribution is not known a priori?

Towards Worst-case Linear Time Sorting

1. Counting Sort
2. Radix Sort

Like Counting Sort, these are also not comparison-based

First step: Counting Sort

1. applies when the keys come from a finite (and preferably small) set, e.g., are integers in the range [0…k-1], for some fixed integer k.
2. We can then create an array V[0…k-1] and use it to count the number of elements with each key [0…k-1].
3. Then each input element can be placed in exactly the right place in the output array in constant time.

Counting Sort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	3	3	3

Input: N records with integer keys from [0…3].

Output: Stable sorted keys.

Algorithm:

- Count frequency of each key value to determine transition locations
- Go through the records in order putting them where they go.

Counting Sort

- Assumption: n input numbers are integers in the range [0,k], $k=O(n)$.
- Idea:
  - Determine the number of elements less than x, for each input x.
  - Place x directly in its position.

Counting Sort - pseudocode

Counting-Sort(A,B,k)

- **for** $i \leftarrow 0$ **to** k
- do C[i] $\leftarrow$0
- **for** $j \leftarrow$1 **to** length[A]
- **do** C[A[j]] $\leftarrow$C[A[j]]+1
- // C[i] contains number of elements equal to i.
- **for** $i \leftarrow$1 **to** k
- **do** C[i]=C[i]+C[i-1]
- // C[i] contains number of elements $\leq i$.
- **for** $j \leftarrow$length[A] **downto** 1
- **do** B[C[A[j]]] $\leftarrow$A[j]
- C[A[j]] $\leftarrow$C[A[j]]-1

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:																			
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit v:	5	9	3	3
# of records with digit < v:	0	5	14	17

N records, k different values. Time to count? $\theta(k)$

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit < v:	0	5	14	17

= location of first record with digit v.

Counting Sort - analysis

for $i \leftarrow 0$ **to** k	$\Theta(k)$
do $C[i] \leftarrow 0$	$\Theta(1)$
for $j \leftarrow 1$ **to** length[A]	$\Theta(n)$
do $C[A[j]] \leftarrow C[A[j]]+1$	$\Theta(1)$ $(\Theta(1)\ \Theta(n) = \Theta(n))$
// $C[i]$ contains number of elements equal to i.	$\Theta(0)$
for $i \leftarrow 1$ **to** k	$\Theta(k)$
do $C[i]=C[i]+C[i-1]$	$\Theta(1)$ $(\Theta(1)\ \Theta(n) = \Theta(n))$
// $C[i]$ contains number of elements $\leq i$.	$\Theta(0)$
for $j \leftarrow$ length[A] **downto** 1	$\Theta(n)$
do $B[C[A[j]]] \leftarrow A[j]$	$\Theta(1)$ $(\Theta(1)\ \Theta(n) = \Theta(n))$
$C[A[j]] \leftarrow C[A[j]]-1$	$\Theta(1)$ $(\Theta(1)\ \Theta(n) = \Theta(n))$

Total cost is $\Theta(k+n)$, suppose $k=O(n)$, then total cost is $\Theta(n)$.

So, it beats the Ω(n log n) lower bound!

Stability

- Counting sort is stable.

Crucial question: can counting sort be used to sort large integers efficiently?

Radix Sort

Input:

- An array of *N* numbers.
- Each number contains *d* digits.
- Each digit between *[0…k-1]*

Output:

- Sorted numbers.

Each digit (column) can be sorted (e.g., using Counting Sort).

Which digit to start from?

RadixSort

344	125	125
125	134	224
333	143	225
134	224	325
224	225	134
334	243	333
143	344	334
225	333	143
325	334	243
243	325	344

All meaning in first sort lost.

Radix Sort

1. Start from the least significant digit, sort
2. Sort by the next least significant digit
3. Are the last 2 columns sorted?
4. Generalize: after j iterations, the last j columns are sorted
5. Loop invariant: Before iteration i, the keys have been correctly stable-sorted with respect to the *i-1* least-significant digits.

Radix sort

Radix-Sort(A,d)

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

Analysis:

Given n d-digit numbers where each digit takes on up to k values, Radix-Sort sorts these numbers correctly in $\Theta(d(n+k))$ time.

Radix sort – example

1019	2231	1019	1019	1019	
3075	3075	2225	3075	2225	**Sorted!**
2225	2225	2231	2225	2231	
2231	1019	3075	2231	3075	

1019	1019	
3075	2231	
2231	2225	**Not sorted!**
2225	3075	

Radix sort – example (binary)

Sorting a sequence of 4-bit integers