# Data Structures: Lecture 14

*Herbert H. Chang, Ph.D.*
*張恆華*
*Eng Sci Ocean Eng*
*National Taiwan University*

*Linear-Time Sorting*



## Bucket Sort

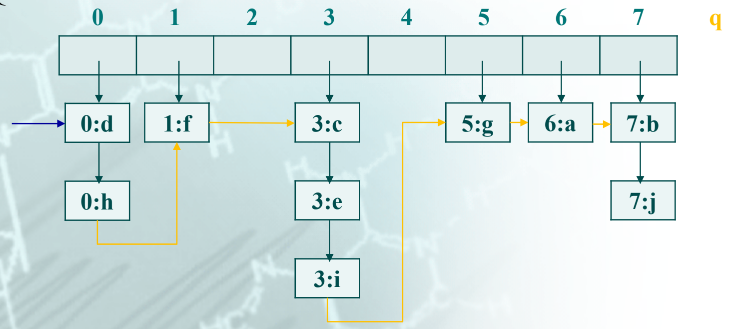
- Works well when keys are in small range, e.g., from 0 to q-1 and the number of items n is larger than, or nearly as large as, q. → i.e. when $q \in O(n)$.
- Allocate an array of q queues, numbered from 0 to q-1. The queues are called "buckets".
- Walk through the list of input items and enqueue each item: an item with key i goes into queue i.
- Input: each item has a numerical key and an associated value.

# Illustration

- Input: each item has a numerical key and an associated value.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 6:a | 7:b | 3:c | 0:d | 3:e | 1:f | 5:g | 0:h | 3:i | 7:j | n |

- Queues:




- Concatenate queues together in order.(→)
- This data structure is exactly like a hash table, no compression fcn.




# Running Time

- Runs in Θ(q+n) time (best- and worst-case).
  - Θ(q) time to initialize the buckets in the beginning and to concatenate them together in the end.
  - Θ(n) time to put all items in their buckets.
- If q∈O(n), that is, the number of possible keys isn't much larger than the number of items we're sorting, then total is Θ(n) time.

## Stable Sorting

- Bucket sort is said to be stable.
- A sort is stable if items with equal keys come out in the same order they went in.
- Insertion, selection, mergesort are easily made stable.
- Linked list quicksort can too; but array version is not.
- Heapsort is never stable.
- Bucket sort is ONLY appropriate when keys are distributed in a small range; i.e. q is in O(n).




## Counting Sort

- If the items are naked keys with no associated values, bucket sort can be simplified to become "counting sort".
- In counting sort, use no queues.
- Merely keep a count of how many copies of each key.
- Suppose we sort 6 7 3 0 3 1 5 0 3 7:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| counts | 2 | 1 | 0 | 3 | 0 | 1 | 1 | 2 |

- When finished, output: 0 0 1 3 3 3 5 6 7 7

# Counting Sort with Complete Items

- Having complete items (key plus associated value).
- The trick is to use the counts to find the right index to move each item to.
- Let x be an input array of objects with keys.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| x | 6 | 7 | 3 | 0 | 3 | 1 | 5 | 0 | 3 | 7 |
| | value | … | | | | | | | | |

7



# Counting Sort with Complete Items

- Begin by counting the keys in x

```
int lengthx = sizeof(x)/sizeof(x[0]);
for (i=0; i<lengthx; i++) {
      counts[x[i].key]++;
}
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| counts | 2 | 1 | 0 | 3 | 0 | 1 | 1 | 2 |

8

## Counting Sort with Complete Items

- Next, do a scan: so that counts[i] contains the number of keys "less than" i.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| counts | 0 | 2 | 3 | 3 | 6 | 6 | 7 | 8 |

```
total = 0;
lengthc = sizeof(counts)/sizeof(counts[0]);
for (j=0; j<lengthc; j++) {
      c = counts[j];
      counts[j] = total;
      total = total + c;
}
```




## Counting Sort with Complete Items

- Let y be the output array to put the sorted objects.
- counts[i] tells us the first index of y to put items with key i.
- Walk through the array x and copy each item to its final position in y.
- When you copy an item with key k, increment counts[k] to make sure that the next item with key k goes into the next slot.

# Counting Sort with Complete Items

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| y | | | | | | | | | | |
| | 0 | 0 | 1 | 3 | 3 | 3 | 5 | 6<br>value | 7 | 7 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| counts | 0 | 2 | 3 | 3 | 6 | 6 | 7 | 8 |
| updated | 1 | 3 | | 4 | | 7 | 8 | 9 |
| index | 2 | | | 5<br>6 | | | | 10 |

```
for (i=0; i<lengthx; i++) {
      y[counts[x[i].key]] = x[i];
      counts[x[i].key]++;
}
```

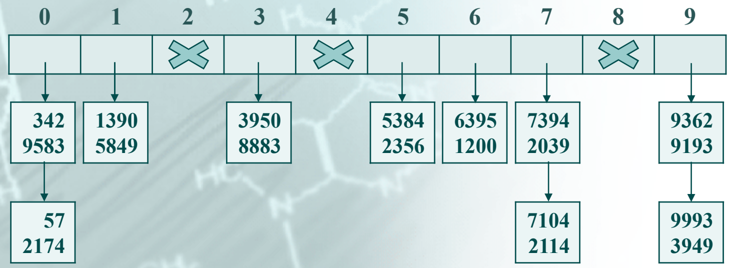



# Running Time

- Bucket sort and counting sort both take O(q+n) time. If $q \in O(n)$, they take O(n) time.
- If you're sorting an array, counting sort takes less memory than bucket sort.
- If you're sorting a linked list, bucket sort more natural.
- What if q >> n?

# Radix Sort

- Suppose we want to sort 1,000 items in the range from 0 to 99,999,999.
- Instead of providing 100 million buckets, let's provide q = 10 buckets and sort on the first digit only.
- We use bucket sort or counting sort, treating each item as if its key is the first digit of its true key.







# Radix Sort

- We could sort queue recursively on second digit, then on sorted third digit, and so on.
- Unfortunately, this tends to break the set of input items into too many small subsets, each of which will be sorted relatively inefficiently.
- Clever idea: we'll keep all numbers in one pile throughout sort, sort on <u>last</u> digit first, then next-to-last, up to the most significant digit.
- Why this works? Because bucket sort and counting sort are <u>stable</u>. Once we sort on the last digit, 55,555,552 and 55,555,58 remain sorted.

## Example with three-digit numbers

- Three-digit numbers:

Sort on 1s: 720 450 771 822 332 504 925 5 955 825 777 858 28 829
Sort on 10s: 504 5 720 822 925 825 28 829 332 450 955 858 771 777
Sort on 100s: 5 28 332 450 504 720 771 777 822 825 829 858 925 955

- It would likely be faster if we sort on 2 digits at a time (a radix of q=100) or even 3 (a radix of q=1,000).
- On computers, more natural to choose power-of-two radix like q=256. (1-byte at a time)
- Note that q = # of buckets = radix of digit we use as a sort key during one pass of bucket or counting sort.





## Number of passes

- How many passes must we perform?
- Each pass inspects $\log_2 q$ bits of each key.
- b=32-bit int: q=256


$\rightarrow$ 4 passes

- If all the keys can be represented in b bits, the number of passes is $\left\lceil \frac{b}{\log_2 q} \right\rceil$.
- Running time of radix sort is $O((n+q)\left\lceil \frac{b}{\log_2 q} \right\rceil)$.

# Running time

- How to choose q?
  Choose $q \in O(n)$, so each pass takes linear time.
  Make q large enough → # of passes small.
  Let's choose $q \approx n$ →
  Radix sort takes $O(n + \frac{b}{\log n} n)$ time.
- For keys = ints: b is a constant; radix sort takes linear time.
- Practical choice: make q be n rounded down to the next power of two.
- To keep memory use low: make $q \approx \sqrt{n}$, rounded to the nearest power of two.

17