Linear-time Sorting

Linear-time Sorting (integer sort)

Recall: Any comparison-based sorting algorithm runs in $\Omega(n\log n)$.

To achieve linear-time sorting of n elements:

- Assume **keys** are **integers** in the range **[0, *N*-1]**
- We can use other operations instead of comparisons
- We can sort in linear time when N is small enough

Bucket Sort

S is a sequence of n (key, element) items with keys in the range $[0, N-1]$

Use the keys as indices into an auxiliary array B of sequences (buckets)

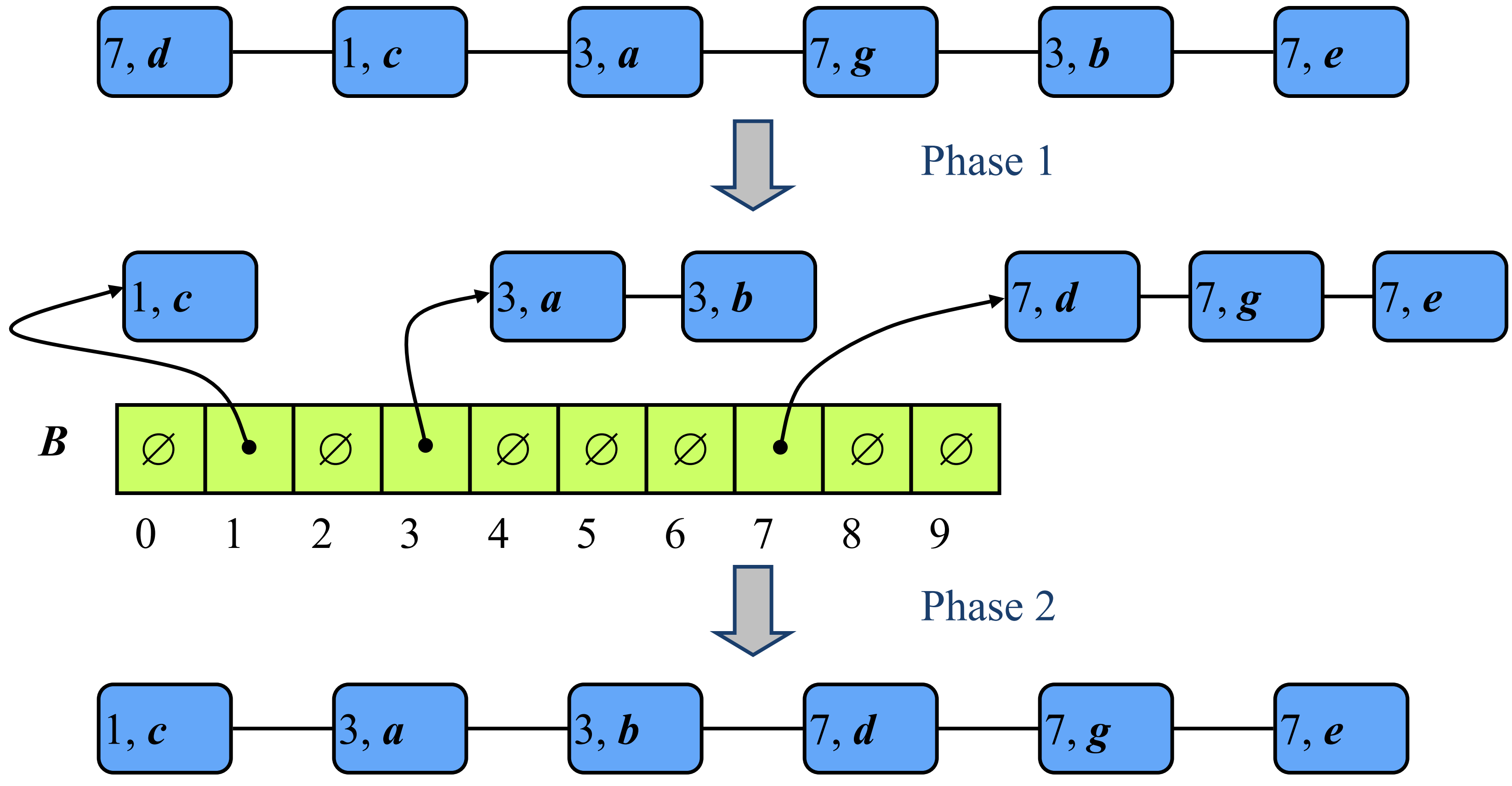
- Phase 1: Empty sequence S by moving each item (k, o) into its bucket $B[k]$
- Phase 2: For $i = 0, \ldots, N-1$, move the items of bucket $B[i]$ to the end of sequence S

Analysis:

- Phase 1 takes $O(n)$ time
- Phase 2 takes $O(n + N)$ time
- Bucket-sort takes $O(n + N)$ time.
- When is this linear time?

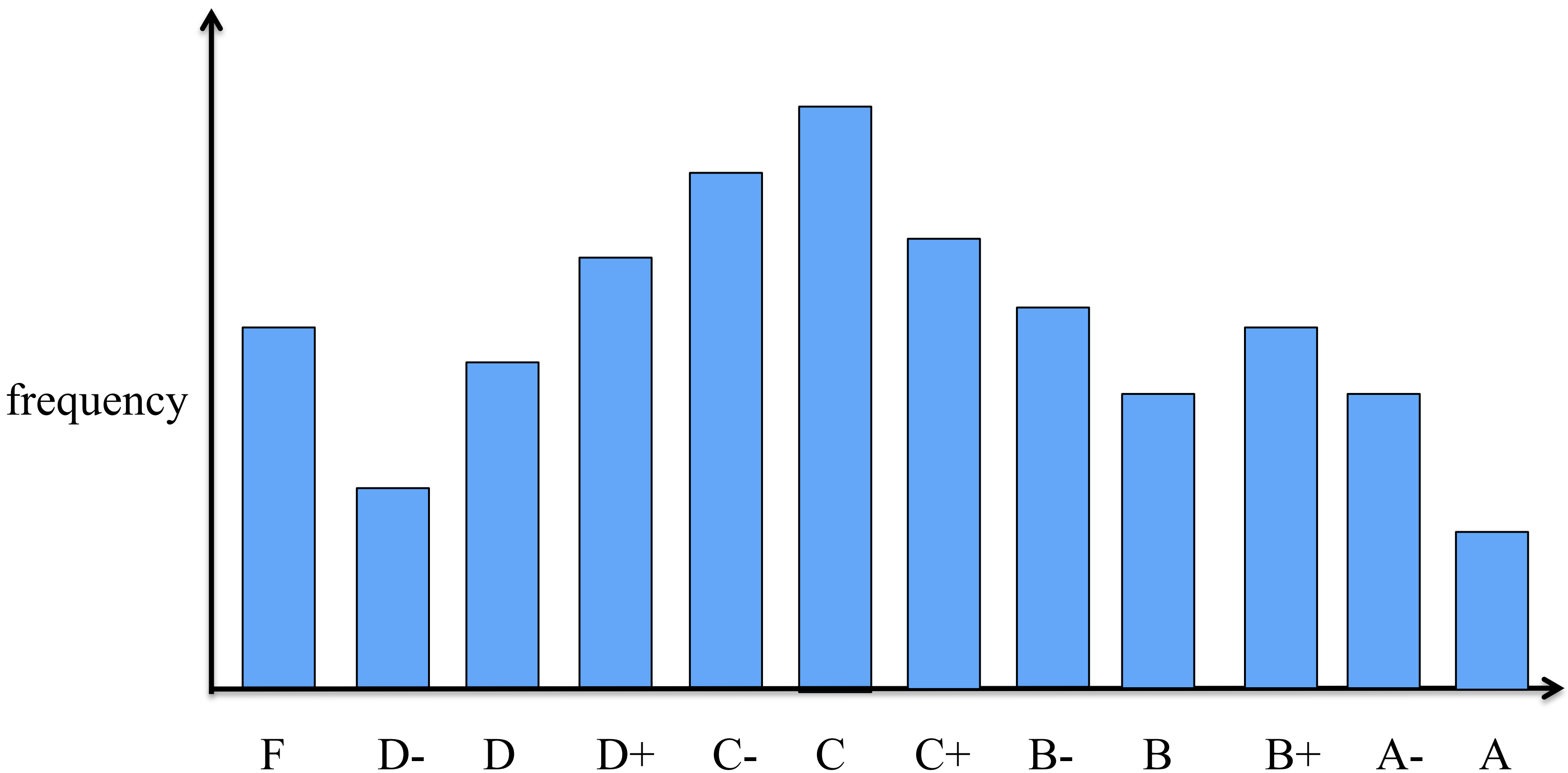
```
Algorithm bucketSort(S, N)
    Input sequence S of (key, element)
        items with keys in the range
        [0, N – 1]
    Output sequence S sorted by
        increasing keys
    B ← array of N empty sequences
    while ¬S.isEmpty()
        (k, o) ← S.remove(S.first())
        B[k].insertLast((k, o))
    for i ← 0 to N – 1
        while ¬B[i].isEmpty()
            (k, o) ← B[i].remove(B[i].first())
            S.insertLast((k, o))
```

Example: key range [0, 9]



Application: Create Histogram

- Use bucket sort and keep track of number of items in each bucket
- Example: histogram of student scores on an English exam



Properties and Extensions

Properties

- keys are used as indices into an array and cannot be arbitrary objects
- no external comparator
- stable sort

Extensions

- Integer keys in the range $[\boldsymbol{a}, \boldsymbol{b}]$
  - Put item $(\boldsymbol{k}, \boldsymbol{o})$ into bucket $\boldsymbol{B}[\boldsymbol{k} - \boldsymbol{a}]$
- String keys from a set $\boldsymbol{D}$ of possible strings, where $\boldsymbol{D}$ has constant size (e.g., names of the 50 U.S. states)
  - Sort $\boldsymbol{D}$ and compute the rank $\boldsymbol{r}(\boldsymbol{k})$ of each string $\boldsymbol{k}$ of $\boldsymbol{D}$ in the sorted sequence
  - Put item $(\boldsymbol{k}, \boldsymbol{o})$ into bucket $\boldsymbol{B}[\boldsymbol{r}(\boldsymbol{k})]$

Lexicographic Order

- A ***d***-tuple is a sequence of $\boldsymbol{d}$ keys $(\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_d)$, where key $\boldsymbol{k}_i$ is said to be the ***i***-th dimension of the tuple

- Ex: the Cartesian coordinates of a point in space are a 3-tuple

- The lexicographic order of two ***d***-tuples is recursively defined as follows

$$(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_d) < (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_d)$$
$$\Leftrightarrow$$
$$(\boldsymbol{x}_1 < \boldsymbol{y}_1) \vee (\boldsymbol{x}_1 = \boldsymbol{y}_1 \wedge (\boldsymbol{x}_2, \ldots, \boldsymbol{x}_d) < (\boldsymbol{y}_2, \ldots, \boldsymbol{y}_d))$$

that is, tuples are compared by the first dimension, then by the second, etc.

Lexicographic-Sort

Let ***stableSort***(S, C) be a stable sorting algorithm that uses comparator C

- C_i is the comparator that compares two tuples by their i-th dimension

Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm ***stableSort***, (one per dimension)

- runs in $O(dT(n))$ time, where $T(n)$ is the running time of ***stableSort***

Algorithm *lexicographicSort*(S)
    Input sequence S of d-tuples
    Output sequence S sorted in lexicographic order

    for $i \leftarrow d$ **downto** 1
        stableSort(S, C_i)

Example:

(7,4,6) (5,1,5) (2,4,6) (2,1,4) (3,2,4)

(2,1,4) (3,2,4) (5,1,5) (7,4,6) (2,4,6)

(2,1,4) (5,1,5) (3,2,4) (7,4,6) (2,4,6)

(2,1,4) (2,4,6) (3,2,4) (5,1,5) (7,4,6)

Radix Sort

- A specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension are integers in the range $[0, N - 1]$
- Radix-sort runs in time $O(d(n + N))$

Algorithm ***radixSort***(S, N)
    Input sequence S of d-tuples such that $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \leq (N - 1, \ldots, N - 1)$ for each tuple $(x_1, \ldots, x_d)$ in S
    Output sequence S sorted in lexicographic order
    for $i \leftarrow d$ **downto** 1
        bucketSort(S, N)

Radix Sort for Binary Numbers

- Consider a sequence of n b-bit integers
  $x = x_{b-1} \dots x_1 x_0$
- We represent each element as a b-tuple of integers in the range [0, 1] and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

Algorithm ***binaryRadixSort***(S)

  Input sequence S of b-bit integers
  Output sequence S sorted

  replace each element x of S with the item $(0, x)$

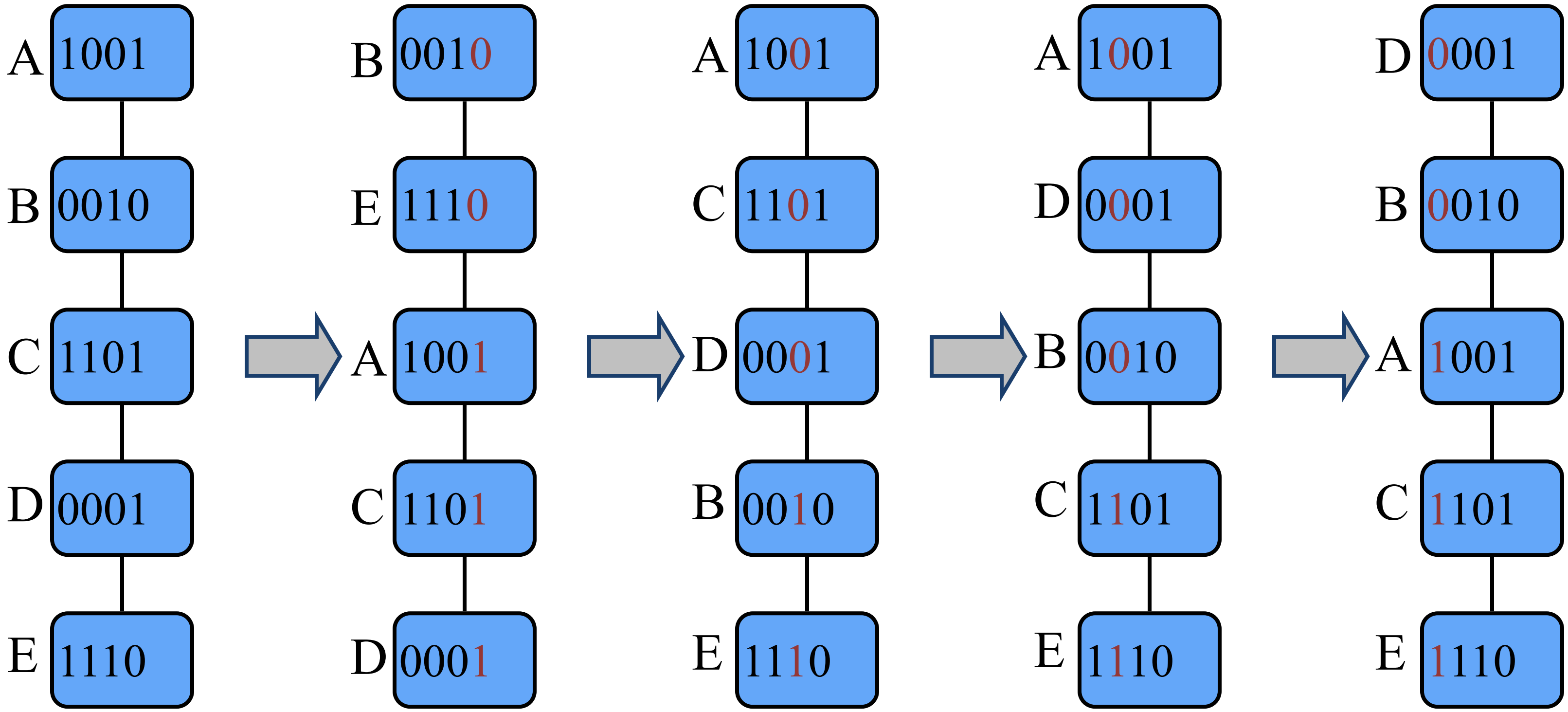
  for $i \leftarrow 0$ **to** $b - 1$

    replace the key k of
      each item (k, x) of S with bit x_i of x

    bucketSort(S, 2)

Example

Use radix sort to sort sequence of 4-bit integers



Other

Describe an efficient method to sort a sequence of n elements if…

1. … the keys fall into the range of $[n^2 - 5n, n^2 + 5n]$.

2. … the keys can be one of 26 possible characters.

3. … the keys fall into the range $[0, n^3 - 1]$.

xkcd #1185 – ineffective sorts