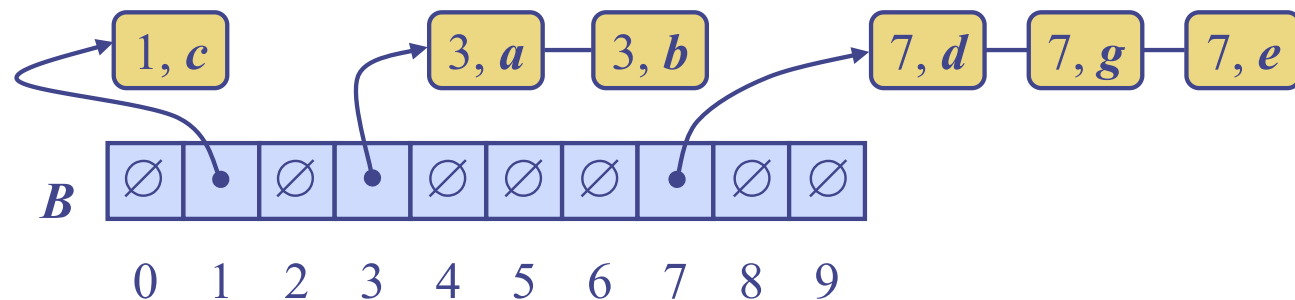


Linear Time Sorting: Bucket-sort, Radix-Sort




Definition:

Stable sorting algorithm

A sorting algorithm that preserves the order of items with identical key

key object



INPUT: (1,x),(6,m),(4,k),(5,t),(2,s),(3,b),(1,a),(6,f),(6,b)

Example of stable output:

OUTPUT: (1,x),(1,a),(2,s) (3,b),(4,k),(5,t),(6,m),(6,f),(6,b)

A Question...

Q: So far the best sorts that we have seen have been $O(n \log n)$, is it possible to do better than that?

A: No, in general there is a lower bound to the problem of $\Omega(n \log n)$

..... but **under certain circumstances**, $O(n)$ is possible!

Bucket-Sort

Imagine putting 200 student papers in alphabetical order according to the first letter of the last name. An insertion sort (or selection sort, or bubbleSort) would have us try to deal with the whole pile at once. Merge-sort would require spreading out all 200 papers and then comparing and piling them back up again in order.

Bucket-Sort has us place them into 26 piles by first letter and then stacking those piles together.

Bucket-Sort

- ◆ Let be S be a sequence of n (key, element) items with keys in the range $[0, N - 1]$
 - ◆ Bucket-sort uses 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 $i = 0, \dots, N - 1$, 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

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

$B \leftarrow$ array of N empty sequences

while $!S.isEmpty()$

$f \leftarrow S.first()$

$(k, o) \leftarrow S.remove(f)$

$B[k].insertLast((k, o))$

for $i \leftarrow 0$ to $N - 1$

while $!B[i].isEmpty()$

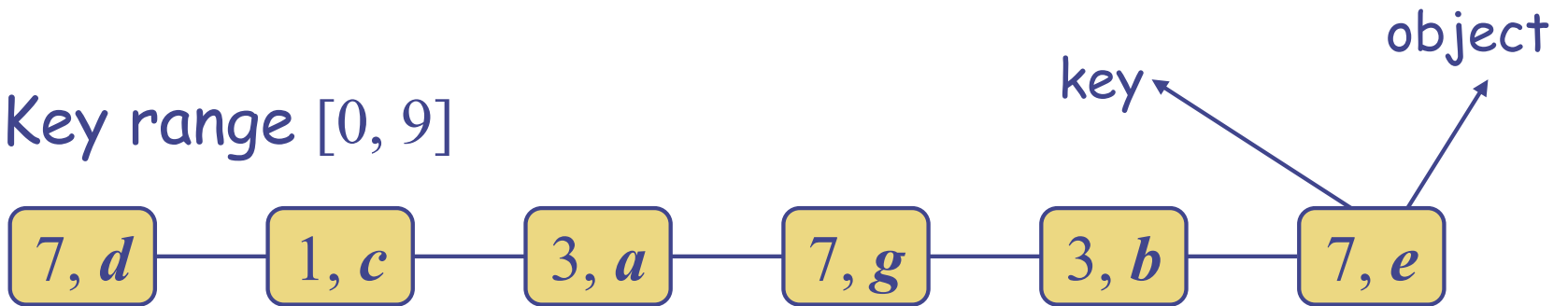
$f \leftarrow B[i].first()$

$(k, o) \leftarrow B[i].remove(f)$

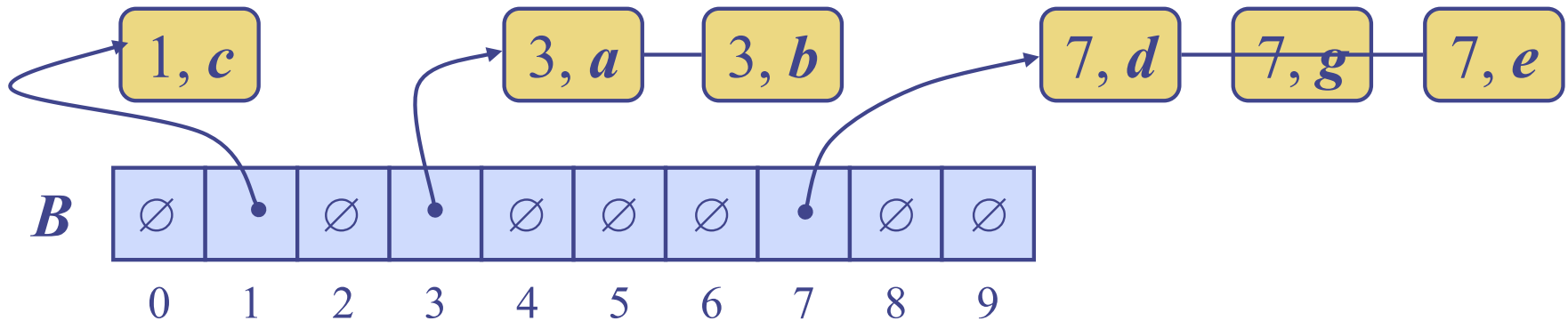
$S.insertLast((k, o))$

Example

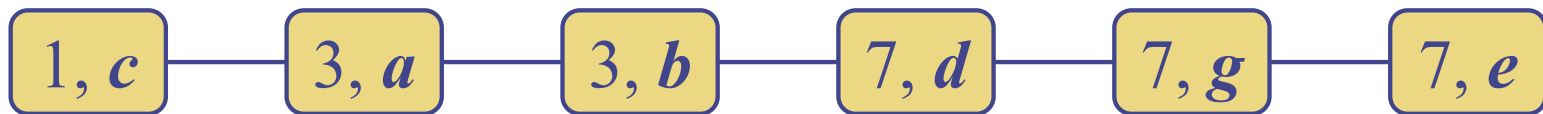
◆ Key range [0, 9]



Phase 1



Phase 2



Properties

Key-type Property

- The keys are used as indices into an array and cannot be arbitrary objects
- No external comparator

Stable Sort Property

- The relative order of any two items with the same key is preserved after the execution of the algorithm

Extensions

Extensions

- Integer keys in the range $[a, b]$
 - ◆ Put item (k, o) into bucket $B[k - a]$
- String keys from a set D of possible strings, where D has constant size
 - ◆ Sort D and compute the rank $r(k)$ of each string k of D in the sorted sequence
 - ◆ Put item (k, o) into bucket $B[r(k)]$

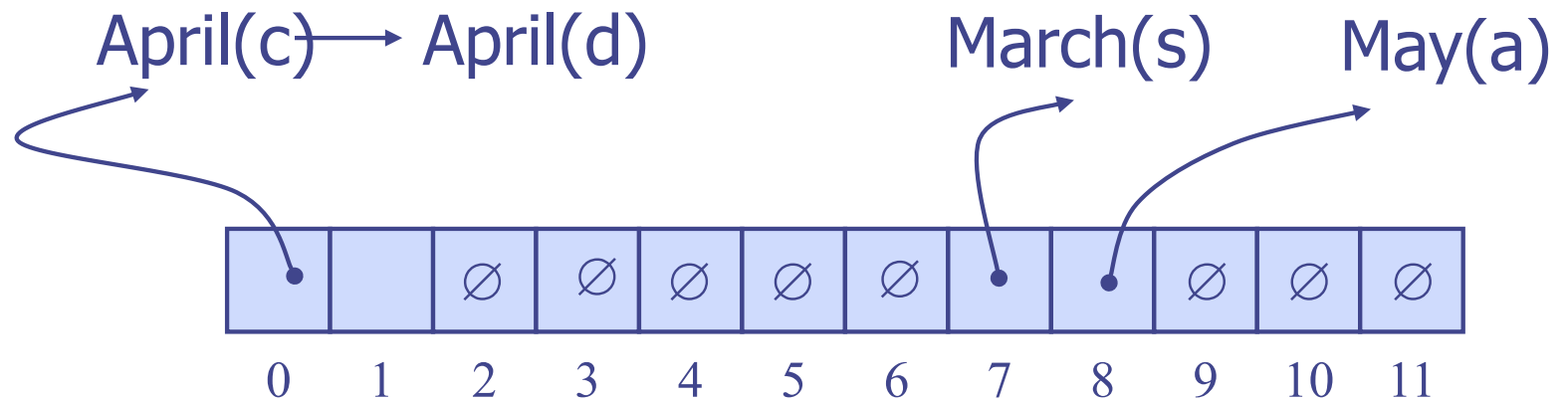
Ex: D = {January, February, March,..... December}

Sort D:

0.Apr
1.Aug
2.Dec
3.Feb
4.Jan
5.June
6.July
7.March
8.May
9.Nov
10.Oct
11.Sep

Sequence to sort:

{May(a), April(c), April(d), March(s)}



Lexicographic Order

- ◆ A d -tuple is a sequence of d keys (k_1, k_2, \dots, k_d) , where key k_i is said to be the i -th dimension of the tuple
- ◆ The lexicographic order of two d -tuples is recursively defined as follows

$$(x_1, x_2, \dots, x_d) < (y_1, y_2, \dots, y_d)$$



$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \dots, x_d) < (y_2, \dots, y_d)$$

i.e., the tuples are compared by the first dimension, then by the second dimension, etc.

$$1,2,3 < 2,5,3$$

$$3,2,1 > 3,1,5$$

Lexicographic-Sort

- ◆ Let C_i be the comparator that compares two tuples by their i -th dimension i.e. for C_2 ,
 $(x_1, x_2, x_3) \leq (y_1, y_2, y_3)$ if $x_2 \leq y_2$.
For C_3
 $(x_1, x_2, x_3) \leq (y_1, y_2, y_3)$ if $x_3 \leq y_3$.
- ◆ Let $stableSort(S, C)$ be
any stable sorting algorithm that uses comparator C
- ◆ Lexicographic-sort sorts a sequence of d -tuples in
lexicographic order by executing d times algorithm $stableSort$,
one per dimension

Example:

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

sort according to left-most dimension

(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)

Lexicographic-Sort

- ◆ Notice that the sorting goes from last to first. Why?

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)

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

Radix-Sort Variation 1

- ◆ uses bucket-sort as the stable sorting algorithm.
- ◆ applicable to tuples where the keys in each dimension i are integers in the range $[0, N - 1]$
- ◆ runs in $O(d(n + N))$

Algorithm *radishSort*(S, N)

Input sequence S of d -tuples such that $(0, \dots, 0) \leq (x_1, \dots, x_d)$ and $(x_1, \dots, x_d) \leq (N - 1, \dots, N - 1)$ for each tuple (x_1, \dots, x_d) in S

Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1

bucketSort(S, N, i)

Radix-Sort Variation 2

- ◆ 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 $N = 2$
- ◆ runs in $O(bn)$ time
- ◆ sorts Java ints (32-bits) in linear time

Algorithm *radicchioSort(S)*

Input sequence S of b -bit integers

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

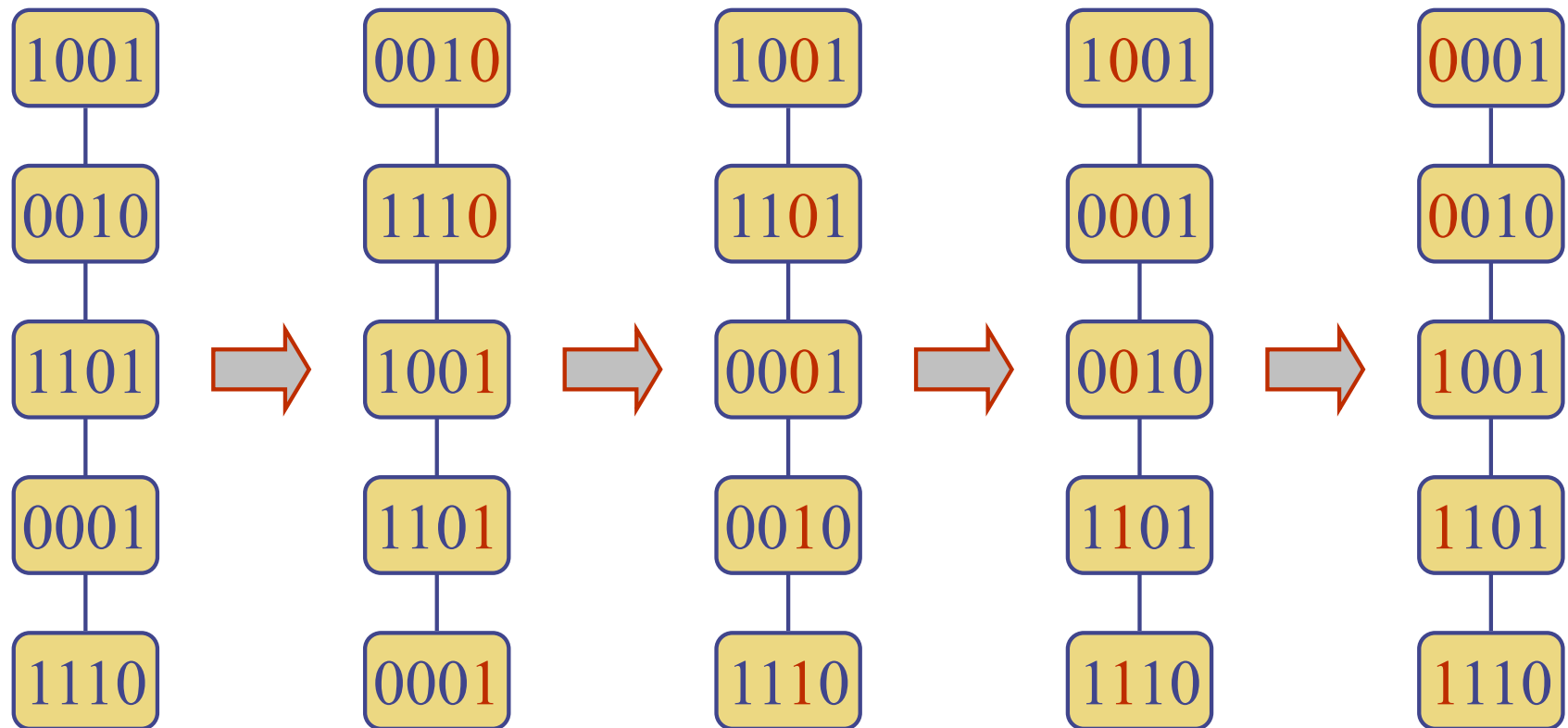
for $i \leftarrow 0$ **to** $b - 1$

use as the key k of each item (k, x) of S the bit x_i of x

bucketSort(S, 2, i)

Example

◆ Sorting a sequence of 4-bit integers



Extensions

◆ Radix-sort variation 3

- The keys are integers in the range $[0, N^2 - 1]$
- We represent a key as a 2-tuple of digits in the range $[0, N - 1]$ and apply radix-sort i.e. we write it in base N notation:
- Example ($N = 10$):
 - ◆ $75 \rightarrow (7, 5)$
- Example ($N = 8$):
 - ◆ $35 \rightarrow (4, 3)$
- The running time is $O(n + N)$
- Can be extended to integer keys in the range $[0, N^d - 1]$

■ Example ($N = 10$):

keys are integers in the range $[0, 99]$

75, 12, 54, 33, 14, 87, 45, 17

$75 = (7, 5)$, $12 = (1, 2)$, $54 = (5, 4)$, $33 = (3, 3)$, ...

$(7, 5)$ $(1, 2)$ $(5, 4)$ $(3, 3)$ $(1, 4)$ $(8, 7)$ $(4, 5)$ $(1, 7)$

$(1, 2)$ $(3, 3)$ $(5, 4)$ $(1, 4)$ $(7, 5)$ $(4, 5)$ $(8, 7)$ $(1, 7)$

$(1, 2)$ $(1, 4)$ $(1, 7)$ $(3, 3)$ $(4, 5)$ $(5, 4)$ $(7, 5)$ $(8, 7)$

Radix sort

- ◆ **radix sort** is a non-comparative **sorting** algorithm. It avoids comparison by creating and distributing elements into buckets according to their **radix**.
- ◆ For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, **radix sort** has also been called bucket sort and **digital sort**.
- ◆ Radix sort can be applied to data that can be sorted lexicographically, be they integers, words, punch cards, playing cards, or the mail.

Extensions

◆ Radix-sort string variation

- The keys are strings of d characters each
- We represent each key by a d -tuple of integers, where a_i is the ASCII (8-bit integer) or Unicode (16-bit integer) representation of the i -th character and apply radix-sort variation 1.

Example

Use the last variation to sort the following strings:
“cart ”, “core ”, “fore”, “cans”, “cars”, “bans”

Step 1: convert to 4-tuples

cart: (3,1,18,20)

core: (3,15,18,5)

fore: (6,15,18,5)

cans: (3,1,14,19)

cars: (3,1,18,19)

bans: (2,1,14,19)

Step 2: Starting from the last digit, bucket-sort the data.