CS 234: Data Types and Structures

Naomi Nishimura

Module 9

Date of this version: November 19, 2019

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

Improving implementations

Importance of searching and sorting:

- Typically, costs of implementations of addition and deletion operations depend on the cost of searching.
- For implementations on orderable data, the cost of keeping data organized depends on the cost of sorting.

Limitations on searching and sorting

On n items of orderable data, any searching algorithm will take $\Omega(\log n)$ time in the worst case and any sorting algorithm will take $\Omega(n \log n)$ time in the worst case.

Details of this result are discussed in CS 231.

Ways to improve searching and sorting

Ideas:

- Avoid worst-case: Use average-case analysis or a **heuristic**.
- Avoid orderable data: Use digital data.

A **heuristic** is a method that may not guarantee bounds on running time or on correctness.

We will consider further options in Module 10.

Outline of the rest of the module:

- Average-case analysis explained
- Searching: Average-case analysis
- Searching: Heuristics
- Searching: Digital data
- Sorting: Digital data

Average-case analysis explained

To calculate the average-case cost, we need to know:

- the possible events,
- the probability of each event, and
- the cost of each event.

A **probability distribution** gives the probabilities of all events, where each event is assigned a number (a **probability**) and the sum of all probabilities is 1.

Average-case cost = $\sum_e Pr[e] \cdot Cost[e]$ where $\sum_e Pr[e] = 1$.

Average-case analysis for running time

Recall from Module 2: The value of $f(n)$ is the sum over all inputs I of size n of the probability of I multiplied by the running time of the algorithm on input I.

Here the "events" are the inputs and the cost of each "event" is the running time on that input.

Average-case cost = $\sum_I Pr[I] \cdot Cost[I]$

Average-case analysis for searching

We will also consider probability distributions on data items, where the probability of an item is the probability that it will be searched.

Here the "events" are the data items and the cost of each "event" is the cost of searching for that item.

In a **uniform distribution**, all probabilities are equal. If there are n items, each has a probability of $\frac{1}{n}$ (totalling 1, as required).

Searching: Average-case analysis for evenly-spaced data

Special case: Data is orderable and evenly spaced throughout the range.

Examples:

- Evenly spaced: 10, 20, 30, 40, 50, 60, 70, 80, 90
- Not evenly spaced: 1, 2, 3, 12, 29, 30, 90

Interpolation search

How interpolation search is similar to binary search:

- At each phase, the interval being searched is from position *Low* to position *High*.
- Compare the value sought with the value in position *Mid*.
- In the next phase, either *Low* is set to *Mid* + 1 (if the search value is greater than *Mid*) or *High* is set to *Mid* - 1 (if the search value is less than *Mid*).

How interpolation search differs from binary search:

- The choice of *Mid* depends not only on *Low* and *High* but also on the values stored in those positions.
- The choice of *Mid* also depends on the value being sought.

Details of interpolation search

- The value of *Mid* depends on the values stored in positions *Low* and *High* and the value that is sought.
- *Mid* is set to *Low* plus the "weight" multiplied by the size of the interval.
- The size of the interval is *High* - *Low*.
- The "weight" is the difference between the value in position *Low* and the value being sought divided by the total range of values stored in positions *Low* through *High*.
- The total range of values is calculated as the difference between the values stored in positions *High* and *Low*.

Worst case: $\Theta(n)$
Average case, for evenly spaced data: $O(\log \log n)$
You are not expected to understand the details of the average case analysis.

Searching: Average-case analysis for static cases

Special cases:

- When the data structure is **static**, there are no rearrangements made to the data that is stored.
- When the data is **static**, there are no additions or deletions.

Distinguishing outcomes (linear search in a linked list):

- For successful search, in the worst case all items are accessed.
- For unsuccessful search, in **all** cases all items are accessed.

Results to follow (for successful searches):

- Using known probabilities to determine average-case cost of linear search in a static unordered array
- Using heuristics for static data

Successful search in a static unordered array

Average-case cost = $\sum_e Pr[e] \cdot Cost[e]$

We need to know

- the possible events,
- the probability of each event, and
- the cost of each event.

In a list of length n, the n events are the item sought being in position $0, 1, \ldots, n-1$.

In a uniform distribution, the probability of each event is $\frac{1}{n}$.

The cost of finding an item in position i is the number of items that need to be read, or $i+1$.

The average-case cost of successful search is $\sum_{i=0}^{n-1}(i+1) \cdot \frac{1}{n} = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{n+1}{2}$.

What if the probability distribution is not uniform?

Self-organizing heuristics

Idea: Assuming that an item that has been accessed is more likely to be accessed in the future, rearrange the data based on guesses of probabilities of access.

Move to front heuristic:

- After a successful search, move item to front.
- (If it is already first, do nothing.)

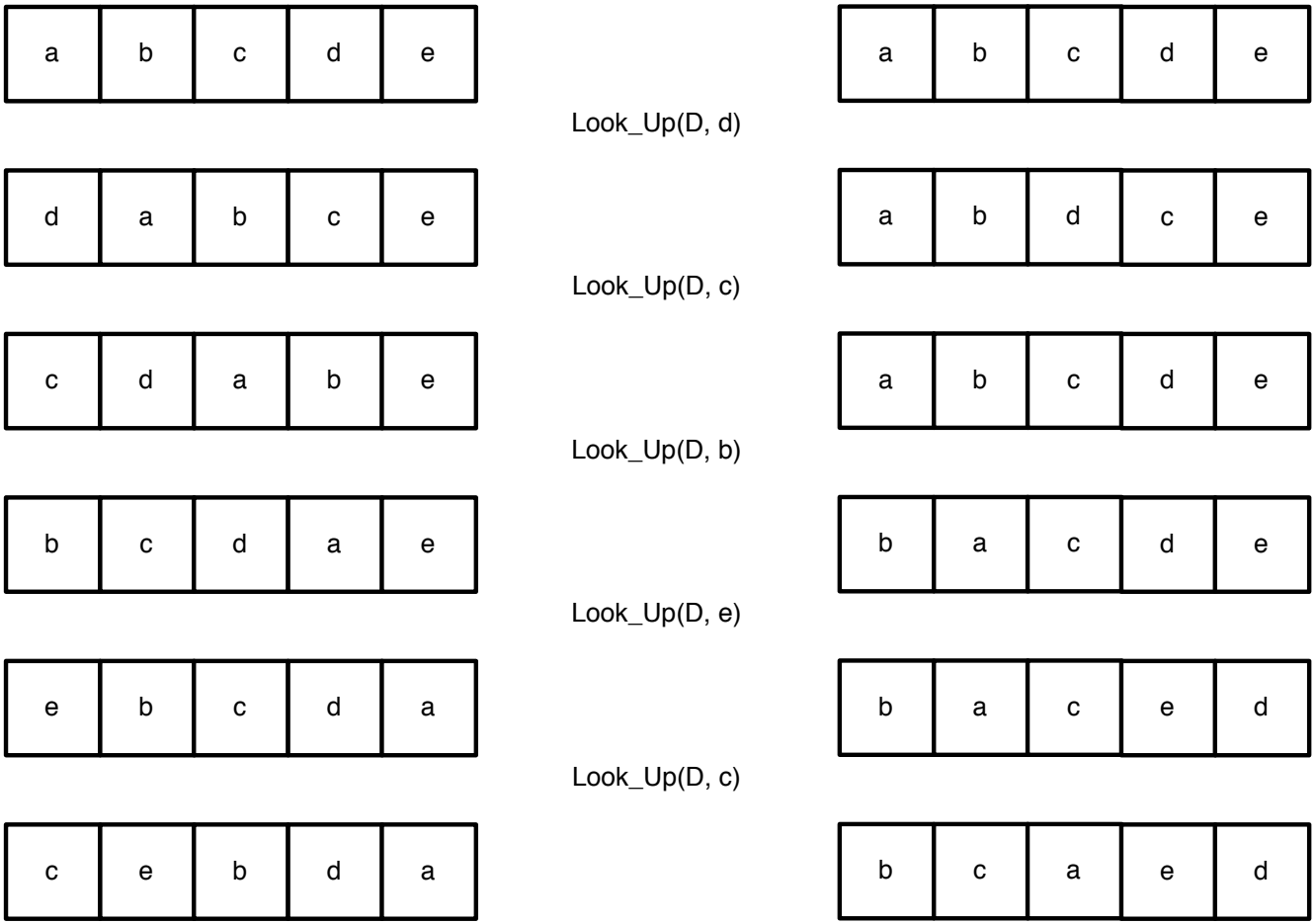
Transpose heuristic:

- After successful search, exchange with item preceding it in sequence (if any).

Average case results (analysis omitted):

- Move to front is no worse than twice optimal cost of a static data structure.
- Transpose does better than move to front unless there are only two items or all items are equally probable to be sought.

Move to Front and Transpose



Searching: Digital data

Idea: Search in a bit vector takes constant time due to random access.

Problem: A bit vector can be used only when the data consists of integers in a fixed and known range.

Issues to consider:

- What if the data items are not integers?
- What if the range of values is not known?

Goal: Use these ideas to implement ADT Dictionary.

Extending the idea to more general data

Use a function that will map any key to an integer in some chosen range (and hence an index in an array).

Differences from a bit vector:

- More than one key may map to the same integer.
- The slot at an index should store a (key, element) pair instead of just a 0 or a 1.

Possible problems:

- We need a simple-to-compute function.
- We need a small enough range that most slots will be used (otherwise we incur a large cost to initialize the array).
- We need a big enough range that not all data items map to the same slot (otherwise we incur a large cost to search among the items in a slot).

Hashing

Hashing distributes keys into buckets by specifying

- a **hash function** f, which maps keys to values in the range from 0 to $N-1$, where N is the number of buckets, and
- a method for **collision resolution**, where **collision** occurs for keys k_1 and k_2 such that $k_1 \neq k_2$ but $f(k_1) = f(k_2)$.

Goals:

- The hash function distributes keys evenly among buckets.
- Collision resolution minimizes the average number of **probes** (keys checked) to find the key being sought.

Types of hashing and assessment

Basic types of conflict resolution:

- **Separate chaining**: Store all items k with $f(k) = i$ in bucket i.
- **Open addressing**: Store at most one item in each bucket.

Assessment:

- Compare the average number of probes with the **load factor** (the number of data items stored divided by the number of buckets)
- Use average case (Details omitted here)

Note: In most of our discussion we omit details and analysis of deletion, focusing instead on search costs.

Hash functions

Goals:

- Evenly spaced
- Quick to compute

Ideas:

- View keys as strings of 0's and 1's (**bits**).
- Map each key to an integer from 0 to $N-1$, where N is the number of buckets.

For mapping ideas (see Appendix B.4 of the mini-textbook for details):

- The number of different binary numbers of length k is 2^k.
- In **modular arithmetic**, a mod N is the remainder when dividing a by N, so the possible remainders when dividing by N are 0 to $N-1$.

Separate chaining

Store all items k with $f(k) = i$ in bucket i.

Ideas:

- Use hash function to determine bucket
- Implement each bucket as a linked list
- *ADD* worst case $\Theta(1)$ (insert at front of list)
- *DELETE* and *LOOK_UP* worst case $\Theta(n)$

Pros:

- Flexibility in size of buckets (given a linked implementation)
- Fast insertion

Cons:

- In the worst case all keys may end up in the same bucket
- Extra space is required

Data for examples

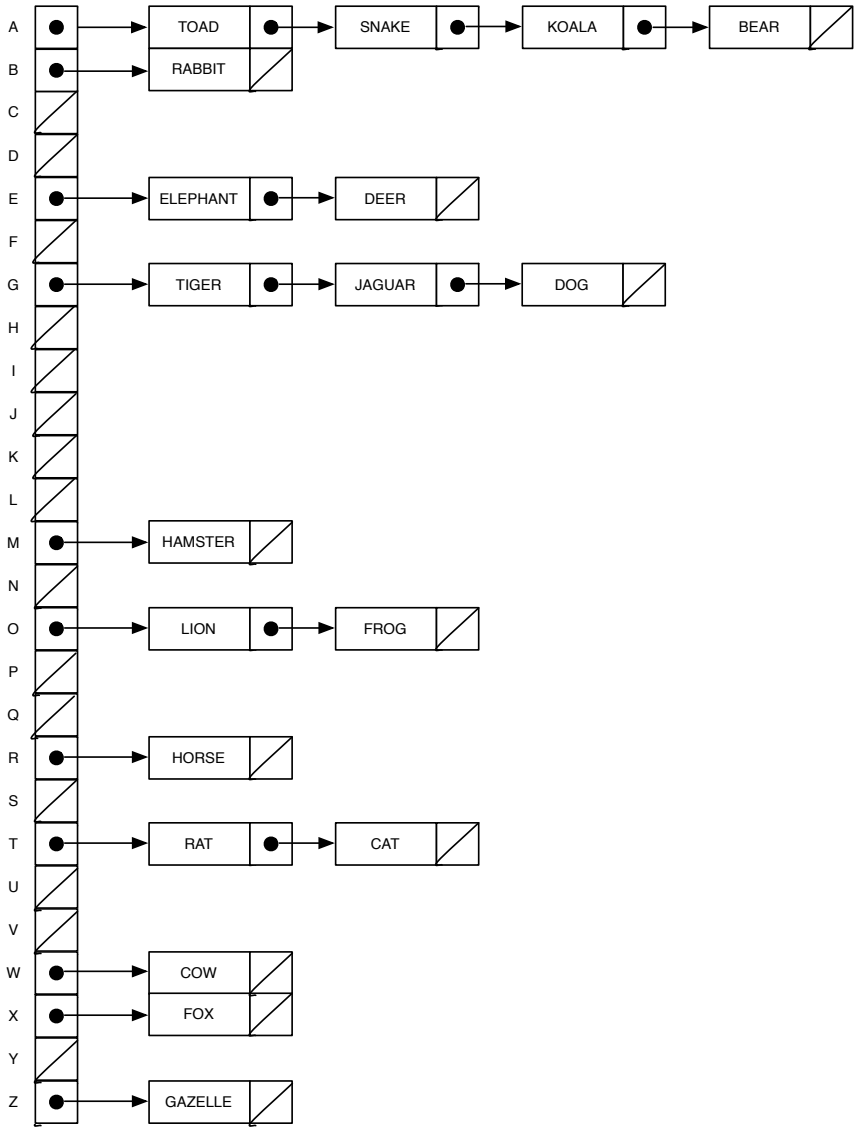
$f(x)$ is the third letter

Order of data to add:

1. BEAR
2. CAT
3. COW
4. DEER
5. DOG
6. ELEPHANT
7. FOX
8. FROG
9. GAZELLE
10. HAMSTER
11. HORSE
12. JAGUAR
13. KOALA
14. LION
15. RABBIT
16. RAT
17. SNAKE
18. TIGER
19. TOAD

Separating chaining example

f(x)=third letter; average successful search: 2.63 probes



Open addressing

- Store at most one item in each bucket.
- In searching for a free bucket in which to store an item, check buckets in a particular order.
- Use the same order in which to search for an item.

The **probe sequence** for an item is the order of buckets searched to find the item or to find a free bucket in which to store the item.

Modifications:

- Can implement deletion by adding "deleted" bit so searches can continue being able to detect that there was an item there before
- Can improve search time by keeping items in order (find something larger, take its place and then reinsert the larger item)

Pros and cons of open addressing:

- Eliminates need for an auxiliary data structure
- Requires that the number of items stored is no greater than size of the table

Probe sequences

Clustering is the phenomenon in which once a collision occurs, more collisions tend to pile up in the same place.

Linear probing:

- Probe sequence for k is $f(k)$, $f(k)+1 \bmod N$, $f(k)+2 \bmod N$, and so on
- "Linear" since like linear search through array starting at $f(k)$

Linear probing example

A	BEAR	1
B	KOALA	2
C	RABBIT	2
D	SNAKE	4
E	DEER	1
F	ELEPHANT	2
G	DOG	1
H	JAGUAR	2
I	TIGER	3
J	TOAD	10
K		
L		
M	HAMSTER	1
N		
O	FROG	1
P	LION	2
Q		
R	HORSE	1
S		
T	CAT	1
U	RAT	2
V		
W	COW	1
X	FOX	1
Y		
Z	GAZELLE	1

Trying to avoid clustering

Quadratic probing uses probe sequence $f(k), f(k)+1^2 \bmod N$, $f(k)+2^2 \bmod N, f(k)+3^2 \bmod N$, and so on.

Idea: Find a nice way of selecting the next location which is:

- easy to compute, and
- differs from other probe sequences as much as possible.

Double hashing

Idea:

- Use a secondary hash function $g(k)$
- Probe sequence will be $f(k)$, $f(k)+g(k) \bmod N$, $f(k)+2g(k) \bmod N$ and so on

Choosing $g(k)$:

- If $g(k)$ and N are not **relatively prime**, then $g(k)$ and N have a common divisor $d \neq 1$. (See Appendix B.4 of the mini-textbook.)
- This means that $(f(k)+(N/d)g(k)) \bmod N = (f(k)+N(g(k)/d)) \bmod N = f(k)$.
- Our probe sequence starts to repeat after only N/d iterations, so we don't reach all the buckets.

Solution: Make N prime.

In our example, we let $g(k)$ be the position in the alphabet of the first letter in k.

Data for double hashing example

$f(k)$ is the position in the alphabet of the third letter $g(k)$ is the position in the alphabet of the first letter

k	$f(k)$	$g(k)$	Part of sequence
BEAR	1	2	$1, 3, 5, 7, 9, 11, \ldots$
CAT	20	3	$20, 23, 26, 29, 3, 6, \ldots$
COW	23	3	$23, 26, 29, 3, 6, 9, \ldots$
DEER	5	4	$5, 9, 13, 17, 21, 25, \ldots$
DOG	7	4	$7, 11, 15, 19, 23, 27, \ldots$
ELEPHANT	5	5	$5, 10, 15, 20, 25, 1, \ldots$
FOX	24	6	$24, 1, 7, 13, 19, 25, \ldots$
FROG	15	6	$15, 21, 27, 4, 10, 16, \ldots$
GAZELLE	26	7	$26, 4, 11, 18, 25, 3, \ldots$

Notes:

- Values of $g(k) > 0$ so non-zero offset each time
- Choose 29 instead of 26 to have the size be a prime number

Rest of data

k	$f(k)$	$g(k)$	Part of sequence
HAMSTER	13	8	$13, 21, 29, 8, 16, 24, \ldots$
HORSE	18	8	$18, 26, 5, 13, 21, 29, \ldots$
JAGUAR	7	10	$7, 17, 27, 8, 18, 28, \ldots$
KOALA	1	11	$1, 12, 23, 5, 16, 27, \ldots$
LION	15	12	$15, 27, 10, 22, 5, 17, \ldots$
RABBIT	2	18	$2, 20, 9, 27, 16, 5, \ldots$
RAT	20	18	$20, 9, 27, 16, 5, \ldots$
SNAKE	1	19	$1, 20, 10, 29, 19, 9, \ldots$
TIGER	7	20	$7, 27, 18, 9, 29, 20, 11, \ldots$
TOAD	1	20	$1, 21, 12, 3, 23, 14, \ldots$

Double hashing example

1	BEAR	1
2	RABBIT	1
3		
4		
5	DEER	1
6		
7	DOG	1
8		
9	RAT	2
10	ELEPHANT	2
11	TIGER	7
12	KOALA	2
13	HAMSTER	1
14		
15	FROG	1

16		
17	JAGUAR	2
18	HORSE	1
19		
20	CAT	1
21	TOAD	2
22		
23	COW	1
24	FOX	1
26		
26	GAZELLE	1
27	LION	2
28		
29	SNAKE	4

Sorting: Digital data

Idea: If the data consists of distinct integers in a fixed and known range, use a bit vector for sorting.

Algorithm:

- Initialize each entry in the bit vector to 0.
- For each data item i, enter a 1 in index i.
- Create a sorted list by linear search through the slots, adding each value i such that slot i contains a 1.

Analysis for n items and range r:

- $\Theta(r)$
- $\Theta(n)$
- $\Theta(r)$

Total cost: $\Theta(n+r)$

Extending the idea

Modifications:

- To allow integers that are not necessarily distinct, store the number of occurrences instead of just 0 or 1.
- For (key, element) pairs in ADT Dictionary such that keys are integers in the range from 0 to $r - 1$, use an array to store elements instead of bits.
- For (key, element) pairs where keys need not be distinct, use each slot in the array either to store or point to an ADT Bucket that can store multiple elements.

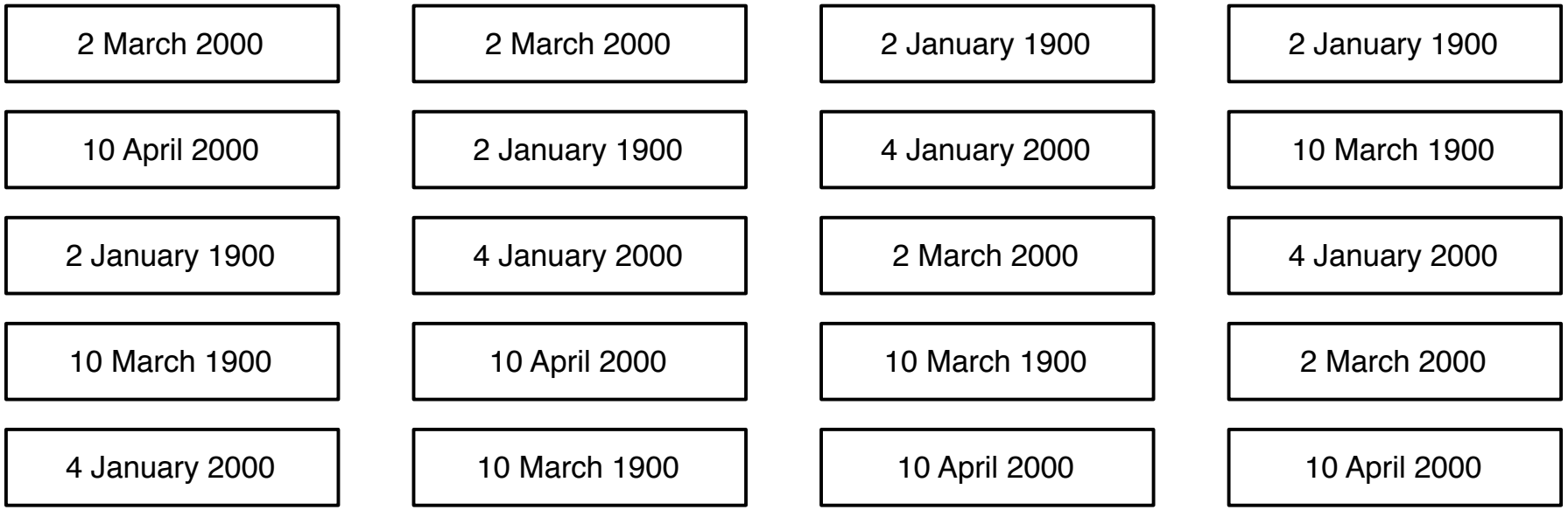
Bucket sort

- Array of r buckets, B_0 through B_{r-1}
- Each bucket is an ADT that supports *ADD_TO_BUCKET* and *RETURN_CONTENTS*
- Sort by adding value i to bucket B_i

Algorithm:

- Initialize each entry in an array of length r to empty.
- For each value $(i, Value)$, execute *ADD_TO_BUCKET*(B_i, *Value*).
- For each bucket in order, concatenate the results of RETURN_CONTENTS.

Sorting by date



Radix sort

Algorithm:

- Divide each value into chunks of bits
- For each chunk from rightmost to leftmost, use a stable bucket sort of all values based on the current chunk

Relating chunk size and number of buckets:

- Suppose we have n items, each of length ℓ bits
- To use 2^b buckets with the range 0 to $2^b - 1$, we need b bits to represent each index
- To use chunks of size b, we have a total of $\lceil \ell / b \rceil$ chunks

Two examples for $\ell = 6$:

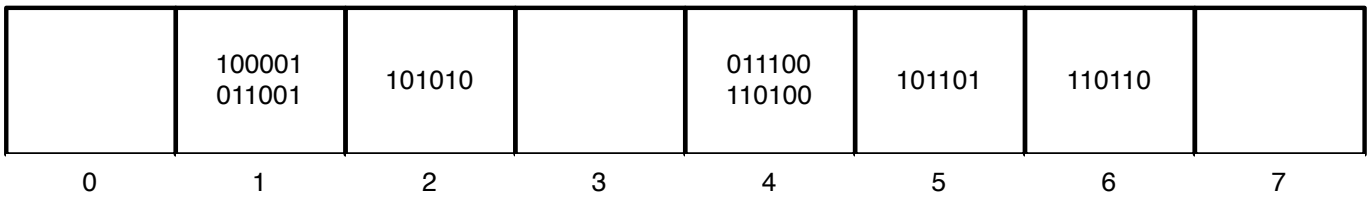
- Option 1: To use $2^3 = 8$ buckets with range 0 to 7, we need 3 bits for each chunk, and use $6/3 = 2$ chunks of size 3 bits each.
- Option 2: To use $2^2 = 4$ buckets with range 0 to 3, we need 2 bits for each chunk, and use $6/2 = 3$ chunks of size 2 bits each.

Example of radix sort, option 1

$2^3 = 8$ buckets with range 0 to 7, $6/3 = 2$ chunks of size 3 bits each.

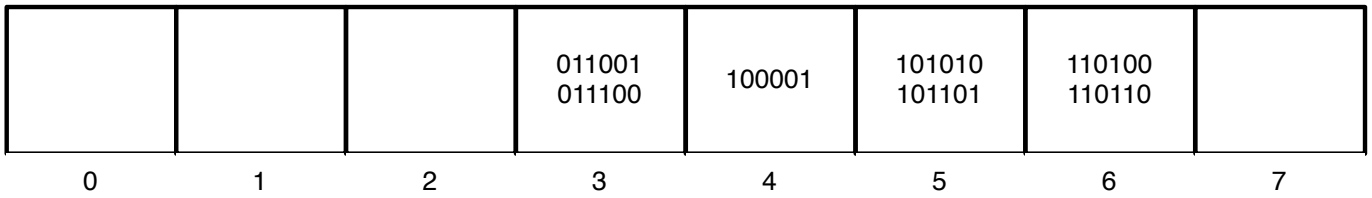
Values: 100001 101101 110110 011100 110100 101010 011001

Pass 1: interpret last three bits as a binary number



Order after pass 1: 100001 011001 101010 011100 110100 101101 110110

Pass 2: interpret first three bits as a binary number



Order after pass 2: 011001 011100 100001 101010 101101 110100 110110

Example of radix sort, option 2

$2^2 = 4$ buckets with range 0 to 3, $6/2 = 3$ chunks of size 2 bits each

Values: 100001 101101 110110 011100 110100 101010 011001

Pass 1: interpret last two bits as a binary number

0	1	2	3
011100 110100	100001 101101 011001	110110 101010	

Order after pass 1: 011100 110100 100001 101101 011001 110110 101010

Pass 2: interpret middle two bits as a binary number

0	1	2	3
100001	110100 110110	011001 101010	011100 101101

Order after pass 2: 100001 110100 110110 011001 101010 011100 101101

Pass 3: interpret first two bits as a binary number

0	1	2	3
	011001 011100	100001 101010 101101	110100 110110

Order after pass 3: 011001 011100 100001 101010 101101 110100 110110

Radix sort analysis

Algorithm:

- Divide each value into chunks of bits
- For each chunk from rightmost to leftmost, use a stable bucket sort of all values based on the current chunk

Assume n items, ℓ bits each, $\lceil \ell/b \rceil$ chunks of size b

Cost of radix sort:

- Bucket sort with n items and 2^b buckets: $O(n + 2^b)$ or $O(n)$ for appropriate choice of b
- Number of passes of bucket sort is number of chunks: $O(\ell/b)$
- Cost of all passes: $O((\ell/b)(n))$
- Total cost: $O(\ell n)$

Module summary

Topics covered:

- Improving implementations
- Average-case analysis
- Interpolation search
- Self-organizing heuristics
- Hashing
- Separate chaining
- Open addressing
- Linear probing
- Quadratic probing
- Double hashing
- Bucket sort
- Radix sort