CS 234

Module 7

September 5, 2018

Refinements of ADT Dictionary

Observations:

- Searching is used for LookUp, Add, and Delete.
- Worst-case cost of search: $\Theta(\log n)$ for AVL trees and (2,3)-trees.
- Creating a dictionary out of n data items can be accomplished by using Add on each one, total cost $\Theta(n \log n)$.
- A sorted array can be formed by sorting n data items in time $\Theta(n \log n)$.

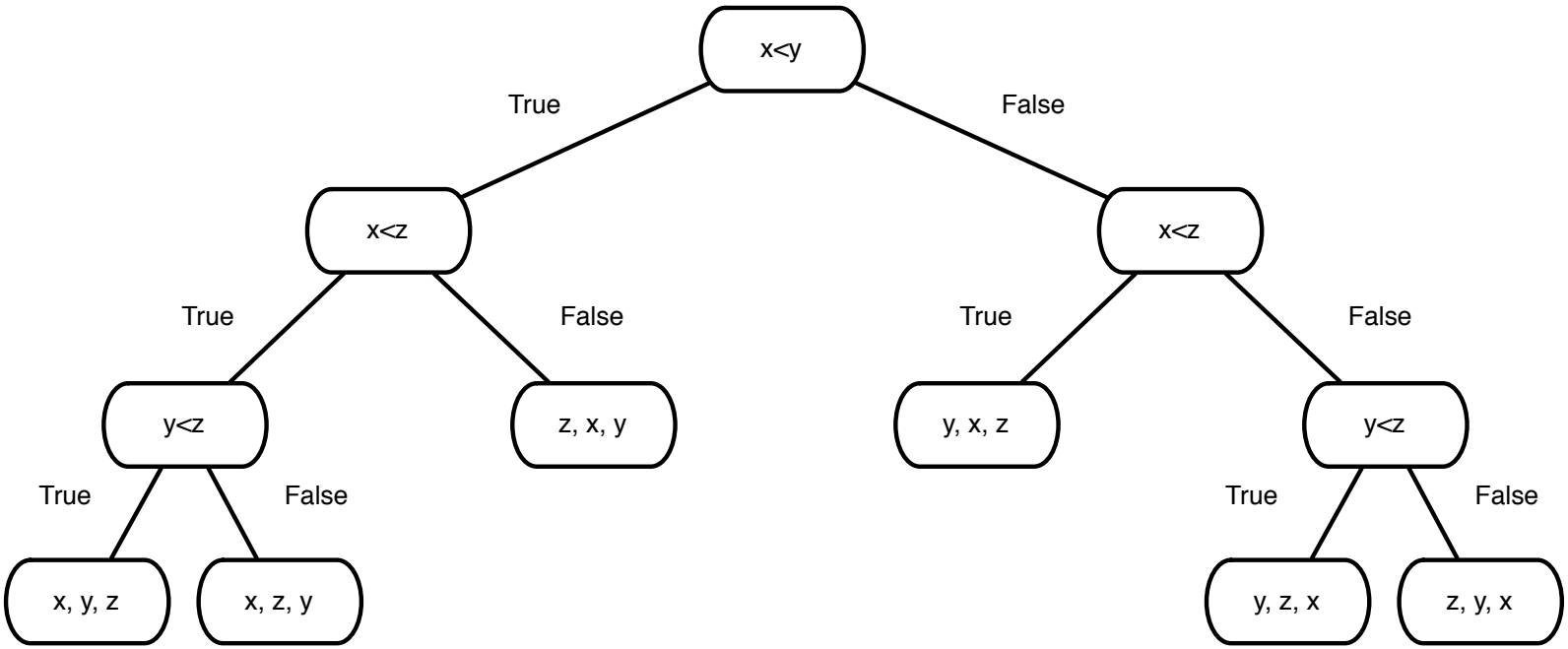
Questions:

- Is it possible to search or sort faster in the data structures studied?
- Are there special cases in which faster algorithms are possible?
- Can we use knowledge about probabilities of data items being searched to obtain good average case results?

Comparison-based algorithms

A **comparison-based** algorithm can be represented as a decision tree:

- Internal nodes represent comparisons.
- Children of a node represent the results of a comparison.
- Leaves are solutions.
- The number of internal nodes from the root to a leaf is the number of comparisons needed to determine that solution.



Decision tree lower bound

The cost of a comparison-based algorithm is at least the cost of all the necessary comparisons.

The number of comparisons needed to reach a particular solution (leaf) is the number of nodes on the path from the root of the decision tree to the leaf.

The worst-case cost of an algorithm is at least the number of comparisons needed to reach the leaf farthest from the root.

Fact: A binary tree with L leaves has height at least $\lfloor \log L \rfloor$.

Any comparison-based algorithm for a problem with L possible solutions will require time in $\Omega(\log L)$.

Lower bounds for searching and sorting

Searching:

- For **any** comparison-based search algorithm, there are n solutions, and hence worst-case running time is in $\Omega(\log n)$.
- Binary search is a comparison-based search algorithm with worst-case running time in $\Theta(\log n)$.

Sorting:

- For **any** comparison-based sorting algorithm, there are $n!$ solutions, and hence worst-case running time is in $\Omega(n \log n)$.
- Mergesort is a comparison-based search algorithm with worst-case running time in $\Theta(n \log n)$.

Using extra information

Faster average case for searching:

- Use information about distribution of data items.
- Use information about probability of accessing data items.
- Use non-comparison-based algorithms.

Faster worst case for sorting:

- Use information about range of data items.
- Use non-comparison-based algorithms.

Using information about distribution of data items

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

Examples:

- Even: 10, 20, 30, 40, 50, 60, 70, 80, 90
- Not even: 1, 2, 3, 12, 29, 30, 90

Idea:

- Real-life: instead of linear or binary search, in dictionary first flip to area letter likely to be.
- Suppose you are searching for t and have found s and w.
- Instead of using the half-way point between s and w, assume an even distribution; t should be $1/5$ of the way

Give example of searching for 30 in above examples, $1/3$ of way.

Interpolation search

Interpolation search:

- Use the outcome of a comparison to reduce the size of the interval being searched (like binary search).
- When searching between `low` and `high`, compare the search value to a value `mid` that is in the range from `low` to `high`.
- 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`.

Unlike binary search:

- Choice of `mid` depends on how close the search value is to each of `low` and `high`.
- Instead of the halfway point between `low` and `high`, determine the point by dividing the difference between the search value and `low` by the difference between the values at indices `high` and `low`.
- Worst case: $\Theta(n)$
- Average case, for uniformly distributed data: $O(\log \log n)$

Average case analysis

The **average case** is the sum over all possible events of the product the cost of an event and probability of the event.

A **probability distribution** is an assignment of probabilities to events, so that the sum of all probabilities is 1.

In a **uniform distribution** on n items, each has probability $1/n$.

Using linear search in an unordered array:

- Searching for item at index i: cost is i, probability is p_i, where $\Sigma_{i=0}^{n-1} p_i = 1$.
- Average case for successful search: $\Sigma_{i=0}^{n-1}(i+1) \cdot p_i$
- For a uniform distribution, average case is $\Sigma_{i=0}^{n-1}(i+1) \cdot \frac{1}{n} = \frac{1}{n}\Sigma_{i=0}^{n-1}(i+1) = \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{n+1}{2}$

Using information about probability of accessing data items

- Unsuccessful search: search all n items in all cases.
- Successful search: search all n items the worst case.

How should items be arranged for best average-case successful search?

Example:

- ape 1/8
- cat 1/4
- dog 1/8
- pig 1/2

Show that if in this order, then
$1/8 + 2 \times 1/4 + 3 \times 1/8 + 4 \times 1/2 = 1/8 + 4/8 + 4/8 + 16/8 = 25/8.$

Now show that if pig, cat, ape, dog, then
$1/2 + 2 \times 1/4 + 3 \times 1/8 + 4 \times 1/8 = 4/8 + 4/8 + 3/8 + 4/8 = 15/8.$

Conclude: Put in decreasing order of probability.

Guessing probability of accessing data items

Idea 1: Guess probabilities and then put in decreasing order of probability.

Problem:
Only as good as the guesses.

Idea 2:

- Keep count of number of accesses so far.
- Change order based on changing count.

Problems:

- Approximation may not be good.
- Extra space may be needed.

Self-organizing heuristics

A **heuristic** is a way of trying to solve a problem.

Self-organizing heuristics

Move to front heuristic:

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

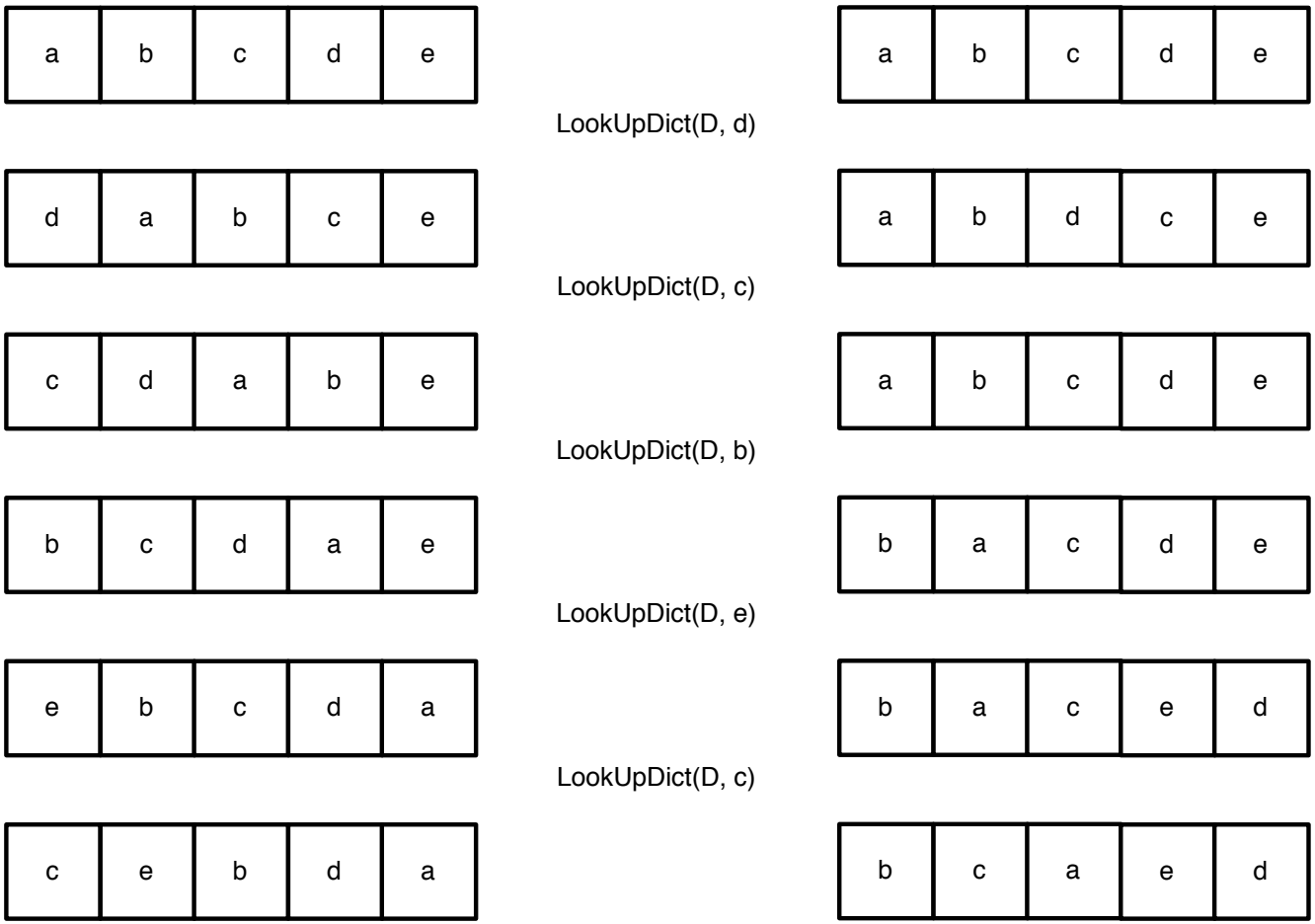
Self-organizing heuristics

Transpose heuristic:

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

Move to Front and Transpose

Note: Best implemented as linked list.



Using non-comparison-based algorithms

What if we don't use comparisons?
Constant time access in an array through random access, using the index.
Is there a natural "index" elements in a dictionary?

Idea: If we know the range of keys, such as all are integers in the range 0-9, we can use the key as the index.

Complications:
What if the keys aren't integers? Use a function to map keys to integers.
What if we don't know the range? Use a function that maps any key to an integer in some chosen range.

Special case: bit vector

If relatively dense in values chosen from a fixed range with an easy indexing function, can use a bit vector.
This requires:

- Simple-to-compute function mapping an element to an index in the range 0 to $R-1$.
- An array of R bits (0 or 1).

For LookUp(D, key), Add(D, key, element), Delete(D, key), can modify so that instead of a single bit, store either 0 or the element.
Problems:

- Need to know that most of the entries will be used, or large cost of initializing all the values.
- Need simple-to-compute function.

Extending the bit vector idea

What if more than one key is mapped to the same integer?
View each "bucket" (elements whose keys map to the same integer) as a dictionary.

What if the buckets are really imbalanced in size?
Goals:

- Find a (cheap) way to (evenly) assign values to buckets
- Find a way to handle multiple keys being assigned ot the same bucket.

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:

- Hash function distributes keys evenly among buckets.
- Collision resolution minimizes the average number of **probes** (keys checked) to find a search key.

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 **load factor** (the number of values stored divided by the number of buckets).
- Use average case. (Details omitted here.)

Hash functions

Goals:

- Evenly distributed
- Quick to compute

Idea:

- 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.

A little bit of math

The number of different binary numbers of length k is 2^k.

- Intuition: there are two choices for the first position, times two choices for the second position, and so on.
- Since $\log_2 2^k$ is k, we can generate the values from 0 to $N-1$ by using $\lceil \log_2 N \rceil$ bits.

In **modular arithmetic**, a mod N is the remainder when dividing a by N.

- The possible remainders when dividing by N are 0 to $N-1$.
- Numbers behave as if they were "wrapping around", like the minutes on a clock.

Two numbers are **relatively prime** if their only common divisor is 1.

- Recall that a number is **prime** if its only divisors are itself and 1.
- If p is prime and q is not equal to p, then p and q are relatively prime.

Possible hash functions

Attempt 1: $\log N$ leading or trailing bits in k represented as a binary number

- If the key is a string, this might be equivalent to using the first or last letter or letters, which may not be very evenly distributed.

Attempt 2: $f(k) = k \bmod N$

- If N is a power of 2, then this is equivalent to using the trailing bits.

Attempt 3: $f(k) = (ak + b) \bmod N$ for random a and b (a and b selected once and then reused)

- The randomness of a and b might help to break up any patterns in the data.

Separate chaining

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

- Use hash function to determine bucket.
- Implement each bucket as a linked list.
- Add O(1) (insert at front of list)
- Delete and LookUp worst case $O(n)$.

Pros:

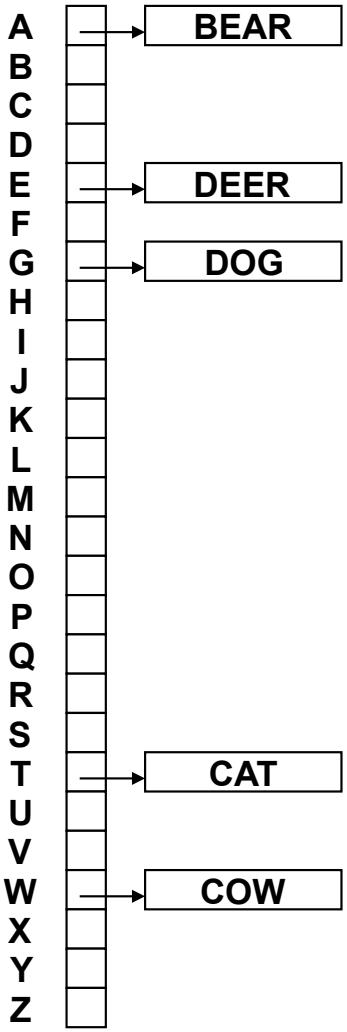
- Flexibility in size of buckets (assume linked implementation of sequence)
- Fast insertion.

Cons:

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

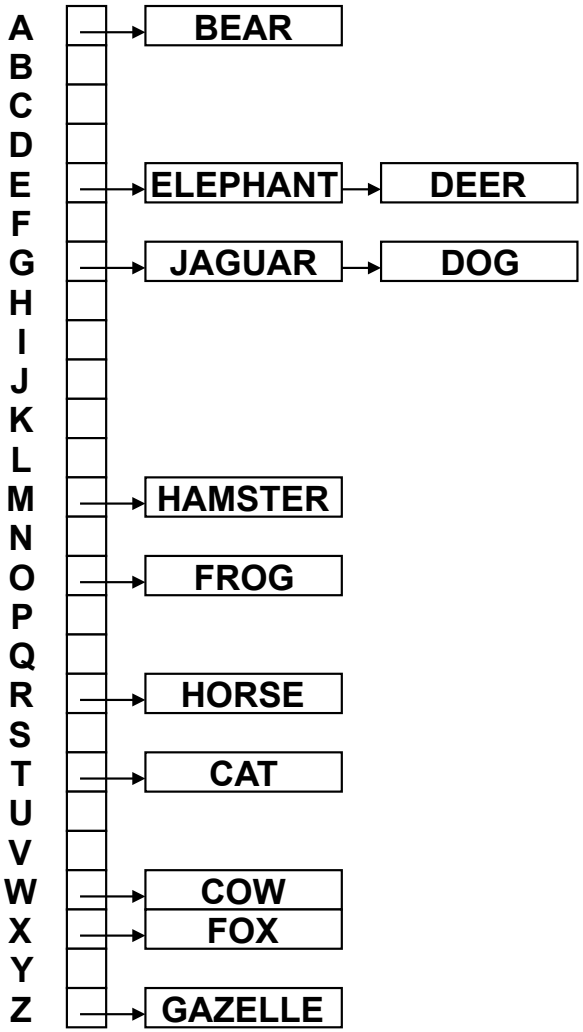
Separating chaining example

f(x)=third letter (bad choice, but easy to understand)



1. BEAR
2. CAT
3. COW
4. DEER
5. DOG

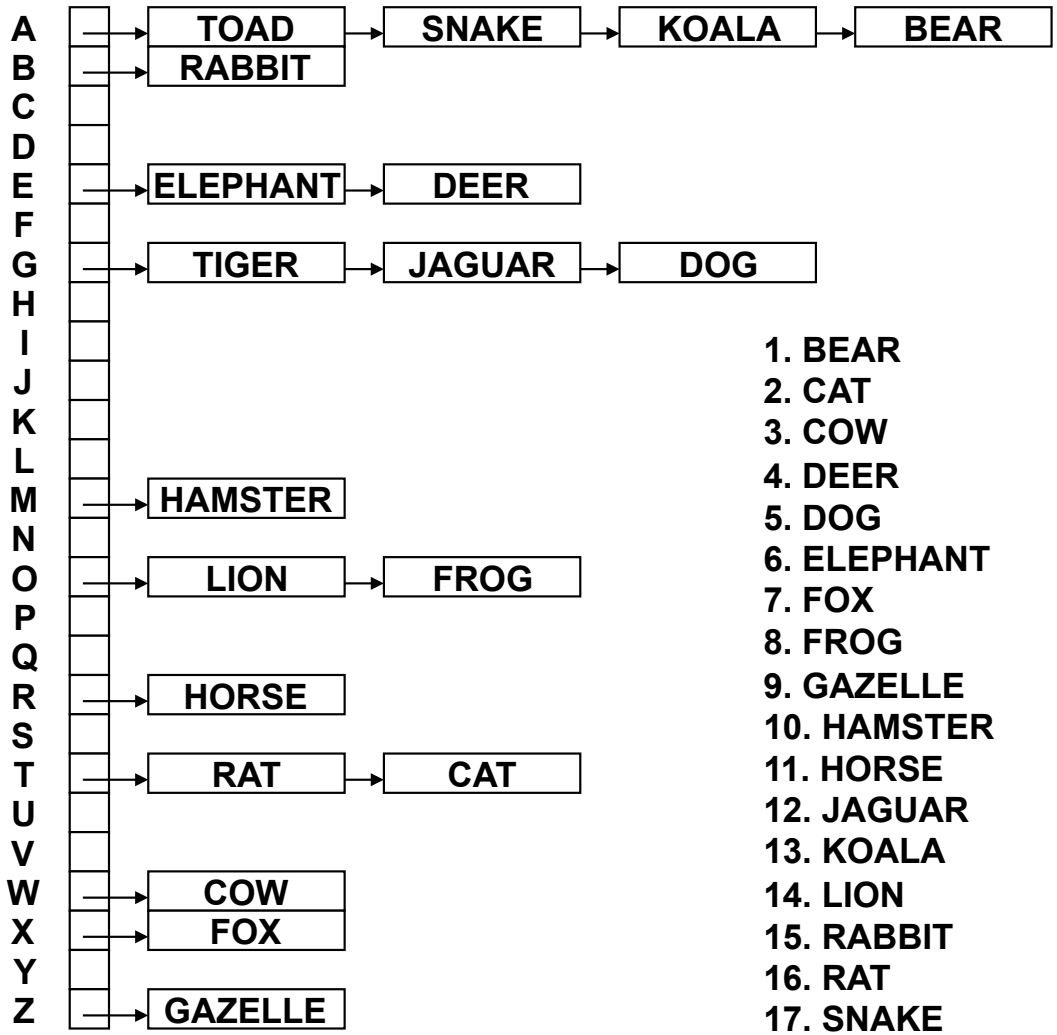
Separating chaining example



1. BEAR
2. CAT
3. COW
4. DEER
5. DOG
6. ELEPHANT
7. FOX
8. FROG
9. GAZELLE
10. HAMSTER
11. HORSE
12. JAGUAR

Separating chaining example

average successful search: 2.63 probes



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

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 knowing there was an item there before.
- Can improve search time by keeping items in order (find something larger, takes it place and then reinsert the larger item).

Pros and cons of open addressing:

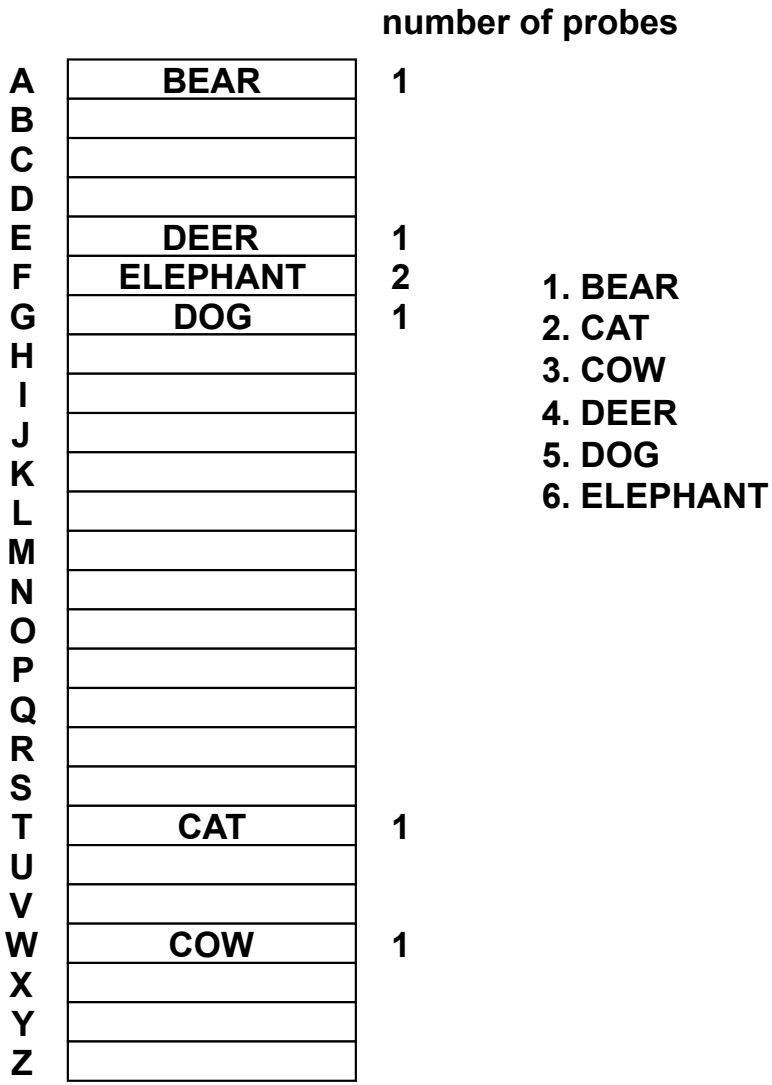
- Eliminates need for auxiliary data structure.
- Requires that number of items stored no greater than size of the table

Probe sequences

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

- The 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



Linear probing example

		number of probes
A	BEAR	1
B	KOALA	2
C		
D		
E	DEER	1
F	ELEPHANT	2
G	DOG	1
H	JAGUAR	2
I		
J		
K		
L		
M	HAMSTER	1
N		
O	FROG	1
P		
Q		
R	HORSE	1
S		
T	CAT	1
U		
V		
W	COW	1
X	FOX	1
Y		
Z	GAZELLE	1

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

Linear probing example

		number of probes
A	BEAR	1
B	KOALA	2
C	RABBIT	2
D		
E	DEER	1
F	ELEPHANT	2
G	DOG	1
H	JAGUAR	2
I		
J		
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

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

Linear probing example

		number of probes
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

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

Linear probing assessment

Options for deletion (like for array):

- mark as deleted and ignore from then on (waste space)
- mark as deleted, use for later insertion
- shift items down (costly!)

Problem: clustering

- A cluster of items becomes bigger and bigger, making search times long.
- TOAD takes 10 probes

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 $g(k)$.
- The 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 $d \neq 1$.
- 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.

Double hashing example

		number of probes
A	BEAR	1
B		
C		
D		
E	DEER	1
F		
G	DOG	1
H		
I		
J	ELEPHANT	2
K		
L		
M		
N		
O		
P		
Q		
R		
S		
T	CAT	1
U		
V		
W	COW	1
X		
Y		
Z		
α		
β		
γ		

1. BEAR	A,C,E,G,I,K,...
2. CAT	T,W,Z,γ,C,F,..
3. COW	W, Z,γ,C,F,I, ..
4. DEER	E,I,M,Q,U,...
5. DOG	G,K,O,S,W,...
6. ELEPHANT	E,J,O,T,Y,A,...

Double hashing example

		number of probes
A	BEAR	1
B		
C		
D		
E	DEER	1
F		
G	DOG	1
H		
I		
J	ELEPHANT	2
K		
L	KOALA	2
M	HAMSTER	1
N		
O	FROG	1
P		
Q	JAGUAR	2
R	HORSE	1
S		
T	CAT	1
U		
V		
W	COW	1
X	FOX	1
Y		
Z	GAZELLE	1
α		
β		
γ		

1. BEAR	A,C,E,G,I,K,...
2. CAT	T,W,Z,γ,C,F,..
3. COW	W, Z,γ,C,F,I, ..
4. DEER	E,I,M,Q,U,...
5. DOG	G,K,O,S,W,...
6. ELEPHANT	E,J,O,T,Y,A,...
7. FOX	X,A,G,M,S,...
8. FROG	O,U, α,D,J,...
9. GAZELLE	Z,D,J,P,...
10. HAMSTER	M,U, γ,H,…
11. HORSE	R,Z,E,M,…
12. JAGUAR	G,Q, α,H,...
13. KOALA	A,L,W,E,P, α,…

Double hashing example

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

1. BEAR	A,C,E,G,I,K,...
2. CAT	T,W,Z,γ,C,F,..
3. COW	W, Z,γ,C,F,I, ..
4. DEER	E,I,M,Q,U,...
5. DOG	G,K,O,S,W,...
6. ELEPHANT	E,J,O,T,Y,A,...
7. FOX	X,A,G,M,S,...
8. FROG	O,U, α,D,J,...
9. GAZELLE	Z,D,J,P,...
10. HAMSTER	M,U, γ,H,…
11. HORSE	R,Z,E,M,…
12. JAGUAR	G,Q, α,H,...
13. KOALA	A,L,W,E,P, α,…
14. LION	O, α,J,V,…
15. RABBIT	B,T,I, α,…
16. RAT	T,I, α,P,…
17. SNAKE	A,T,J, γ,…

Double hashing example

		number of probes
A	BEAR	1
B	RABBIT	1
C		
D		
E	DEER	1
F		
G	DOG	1
H		
I	RAT	2
J	ELEPHANT	2
K	TIGER	6
L	KOALA	2
M	HAMSTER	1
N		
O	FROG	1
P		
Q	JAGUAR	2
R	HORSE	1
S		
T	CAT	1
U	TOAD	2
V		
W	COW	1
X	FOX	1
Y		
Z	GAZELLE	1
α	LION	2
β		
γ	SNAKE	4

1. BEAR	A,C,E,G,I,K,...
2. CAT	T,W,Z,γ,C,F,..
3. COW	W, Z,γ,C,F,I, ..
4. DEER	E,I,M,Q,U,...
5. DOG	G,K,O,S,W,...
6. ELEPHANT	E,J,O,T,Y,A,...
7. FOX	X,A,G,M,S,...
8. FROG	O,U, α,D,J,...
9. GAZELLE	Z,D,J,P,...
10. HAMSTER	M,U, γ,H,…
11. HORSE	R,Z,E,M,…
12. JAGUAR	G,Q, α,H,...
13. KOALA	A,L,W,E,P, α,…
14. LION	O, α,J,V,…
15. RABBIT	B,T,I, α,…
16. RAT	T,I, α,P,…
17. SNAKE	A,T,J, γ,…
18. TIGER	G, α,R,I, γ,T,K,...
19. TOAD	A,U,L,C,...

Bucket sort

Idea:

- Array of N buckets
- Each bucket is an ADT that supports AddToBucket and ReturnContents
- Sort by adding value i to bucket i

Algorithm:

- For value with index i, AddToBucket(B_i, value)
- For each bucket in order, ReturnContents

Bucket sort analysis

- Process each bucket ($\Theta(N)$)
- Process each element ($\Theta(n)$)
- $\Theta(N+n)$ in total

When is this any good?

- Consider $N = n^2$ (time), $N = n \log n$ (space), $N = n$
- Best when N is no bigger than $O(n \log n)$, depending on space/time tradeoff

How can it be made stable?

- Recall: stable keeps order same in input and output for elements with same key
- Buckets should behave like ADT Queue: add at end, extract from beginning to end.

Practicalities

If N is really large as a function of n, then the number of bits per key, $\log N$, is very large, and indexing shouldn't necessarily be seen as a constant time operation.

Radix sort

Idea:

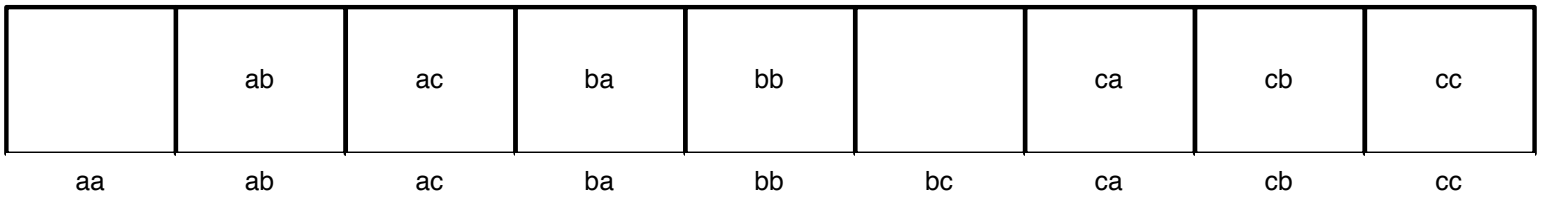
- Generalize bucket sort idea to use in situations when N is too large.
- Use idea iteratively on fragments of keys (must be stable!).
- Consider sorting a spreadsheet by month and then year.
- Consider putting words in buckets for each letter, starting with the letters closer to the end of the words.

Algorithm:

- Divide key into chunks of bits
- Starting with chunk of smallest bits up to chunk of biggest bits
- Use bucket sort on set of bits

Comparison of bucket sort and radix sort

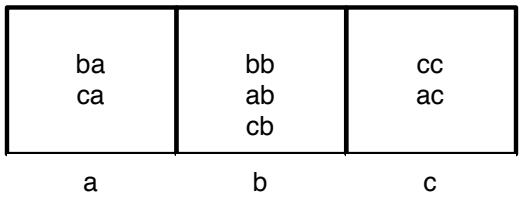
Keys: cb ba ac ca bb ab cc



Order after extraction: ab ac ba bb ca cb cc

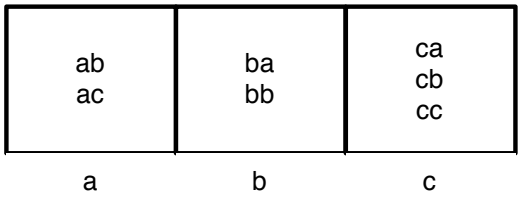
Keys: bb ba cc ca ab cb ac

Pass 1: sort using second letter in each word



Order after pass 1: ba ca bb ab cb cc ac

Pass 2: sort using first letter in each word



Order after pass 2: ab ac ba bb ca cb cc

Radix sort in general

- n items, b bits each
- Use bucket sort repeatedly, for N buckets
- Number of bits needed for N values: $\log N$
- Number of chunks of bits of size $\log N$: $b/\log N$.

Two examples for $n = 7$ and $b = 6$:

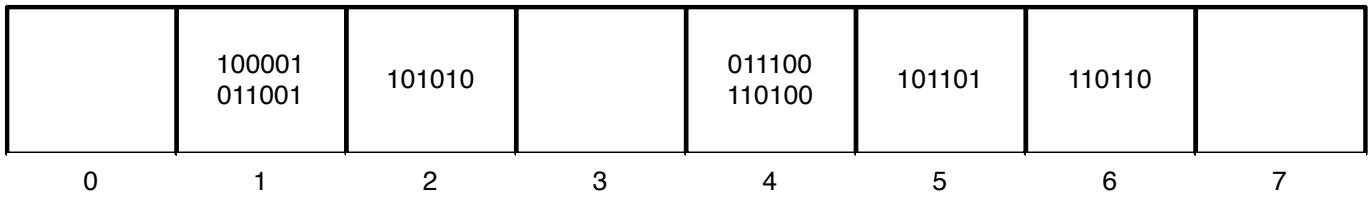
- Option 1: $N = 8$, use 2 chunks of size $\log 8 = 3$.
- Option 2: $N = 4$, use 3 chunks of size $\log 4 = 2$.

Example of radix sort, option 1

$N = 8$, use 2 chunks of size $\log 8 = 3$.

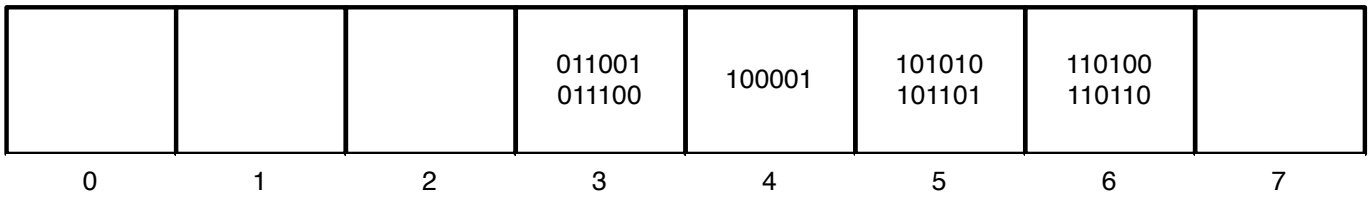
Keys: 100001 101101 110110 011100 110100 101010 011001

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

$N = 4$, use 3 chunks of size $\log 4 = 2$.

Keys: 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

- n items, b bits each
- Use bucket sort repeatedly, for N buckets
- Number of bits needed for N values: $\log N$
- Number of chunks of bits of size $\log N$: $b/\log N$.

Cost of radix sort:

- Bucket sort with n items and N buckets: $O(n+N)$
- Number of passes of bucket sort = number of chunks
- Radix sort cost is $O(b/\log N)(n+N)$

Comparing radix sort to comparison-based sort

Cost of comparison-based sort: $O(n \log n)$.

Cost of radix sort: $O(b/\log N)(n+N)$

Goal:

- Choose N such that $n+N \in O(n)$ and $b/\log N$ is as small as possible.
- Choose N to be n.

- If b is smaller than $\Theta(\log^2 n)$, radix sort is better than comparison-based.
- If b is in $\Theta(\log^2 n)$, radix sort may or may not be better than comparison-based.
- If b is bigger than $\Theta(\log^2 n)$, radix sort is worse than comparison-based.