Recitation 7

Problem 1

2-SUM

Problem definition

- Given an array of A = [x1, x2, …, xn] and x
- Find a pair (i,j) such that A[i] + A[j] = x

$30	$8	$15	$18	$23	$20	$25	$1
#1	#2	#3	#4	#5	#6	#7	#8

Naive solution

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true
return false
```

Naive solution

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true
return false
```

Runtime?

Naive solution

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true
return false
```

Runtime?
O(n^2)

Naive solution

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true
return false
```

Runtime?
O(n^2)

What is this trying to do?

Naive solution

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true
return false
```

Runtime?
O(n^2)

What is this trying to do?

Given a[i], find some a[j] matching x.

Naive solution

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true
return false
```

Runtime?
O(n^2)

What is this trying to do?

~~Given a[i], find some a[j] matching x.~~

Given a[i], check if x - a[i] exists.

Specify what, not how.

1B O(log n) extra space

Speed up this query:

Given a[i], check if x - a[i] exists.

How fast can this be done with given restrictions?

1B) Your solution must only incur O(logn) extra space. What is its time complexity?

1B O(log n) extra space

Speed up this query:

Currently O(n)

Given a[i], check if x - a[i] exists.

How fast can this be done with given restrictions?

1B) Your solution must only incur O(logn) **extra space**. What is its time complexity?

Can reuse current space.

Any ideas?

1B O(log n) extra space

Speed up this query:

Currently O(n)

Given a[i], check if x - a[j] exists. **Can do O(log n) if array a is sorted.**

How fast can this be done with given restrictions?

1B) Your solution must only incur O(logn) **extra space**. What is its time complexity?

Can reuse current space.

Any ideas?

1B O(log n) extra space

Speed up this query:

Given a[i], check if x - a[j] exists.

Currently O(n)

Can do O(log n) if array a is sorted.

How fast can this be done with given restrictions?

Runtime?

Algorithm outline:

```
Sort(A)
for each i in [1,…, n]:
    find(x-A[i], A) by binary search.
```

1B O(log n) extra space

Speed up this query:

Given a[i], check if x - a[j] exists.

Currently O(n)

Can do O(log n) if array a is sorted.

How fast can this be done with given restrictions?

Algorithm outline:

```
Sort(A)
for each i in [1,…, n]:
    find(x-A[i], A) by binary search.
```

Runtime?

O(n log n) sort
n x O(log n)

Total:
O(n log n)

1B O(log n) extra space

Note: There's a faster 2 pointer solution, but still bounded by sorting.

Speed up this query:

Currently O(n)

Given a[i], check if x - a[j] exists. Can do O(log n) if array a is sorted.

How fast can this be done with given restrictions?

Algorithm outline:

```
Sort(A)
for each i in [1,…, n]:
    find(x-A[i], A) by binary search.
```

Runtime?

O(n log n) sort
n x O(log n)

Total:
O(n log n)

1C O(n) extra space

Speed up this query:

Given a[i], check if x - a[j] exists.

How fast can this be done with given restrictions?

1B) Your solution must only incur O(n) extra space. What is its time complexity?

Any ideas?

1C O(n) extra space

Speed up this query:

Given a[i], check if x - a[j] exists.

How fast can this be done with given restrictions?

1B) Your solution must only incur O(n) extra space. What is its time complexity?

Any ideas?

Create lookup table for O(1) search

1C O(n) extra space

Speed up this query:

Given a[i], check if x - a[j] exists.

How fast can this be done with given restrictions?

Algorithm outline: **Runtime?**

```
insert all A into hash table H
for each i in [1,…, n]:
    find(x-A[i], H)
```

1C O(n) extra space

Speed up this query:

Given a[i], check if x - a[j] exists.

How fast can this be done with given restrictions?

Algorithm outline:

```
insert all A into hash table H
for each i in [1,…, n]:
    find(x-A[i], H)
```

Runtime?

O(n) insert into hash table
n x O(1)

Total:
O(n)

1D Outputting all pairs.

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true
return false
```

Minimal modifications?

1D Outputting all pairs.

Output = []

```
for each i in [1,…, n]:
    for each j in [1,…, n]:
        if A[i] + A[j] == x:
            return true append (i, j) to output
return false output
```

Similar modifications

Sort(A)
for each i in [1,…, n]:
    find(x-A[i], A) by binary search.

Insert all A into hash table H
for each i in [1,…, n]:
    find(x-A[i], H)

Similar modifications

If you want a version without duplicate indices, filter the list.

```
output = []
Sort(A)
for each i in [1,…, n]:
    if find(x-A[i], A) by binary search:
        append (i, find(x-A[i], H)) to output
return output
```

```
output = []
Insert all A into hash table H
for each i in [1,…, n]:
    if find(x-A[i], H):
        append (i, find(x-A[i], H)) to output
return output
```

1E 3-SUM

- Given an array of A = [x1, x2, …, xn] and x
- Find a tuple (i,j, k) such that A[i] + A[j] + A[k] = x

1E 3-SUM

- Given an array of A = [x1, x2, …, xn] and x
- Find a tuple (i,j, k) such that A[i] + A[j] + A[k] = x

Approach?

1E 3-SUM

- Given an array of A = [x1, x2, …, xn] and x
- Find a tuple (i,j, k) such that A[i] + A[j] + A[k] = x

Approach?

Insert all (i, j) pairs to H[A[i] + A[j]]
for each k:
    look for x - A[k] in H

Runtime?

1E 3-SUM

- Given an array of A = [x1, x2, …, xn] and x
- Find a tuple (i,j, k) such that A[i] + A[j] + A[k] = x

Approach?

Insert all (i, j) pairs to H[A[i] + A[j]]
for each k:
    look for x - A[k] in H

Runtime?
O(n^2) insertion into H
n x O(1) search

HOMEWORK: 4-SUM?

How would you adapt this to 4-SUM? What would be the runtime?

1Extra 4-SUM?

How would you adapt this to 4-SUM? What would be the runtime?

Approach?

Insert all (i, j) pairs to H[A[i] + A[j]]
for each (k,l):
    look for x - A[k]- A[l] in H

Runtime?
O(n^2) insertion into H
n^2 x O(1) search
= O(n^2)

Problem 2

Cuckoo hashing

- Instead of having a single hash table
- Use two! With their own hash keys.

Cuckoo hashing

- Instead of having a single hash table
- Use two! With their own hash keys.
- When inserting:
  - ….< see other slides> …

Cuckoo Hashing (Introduction)

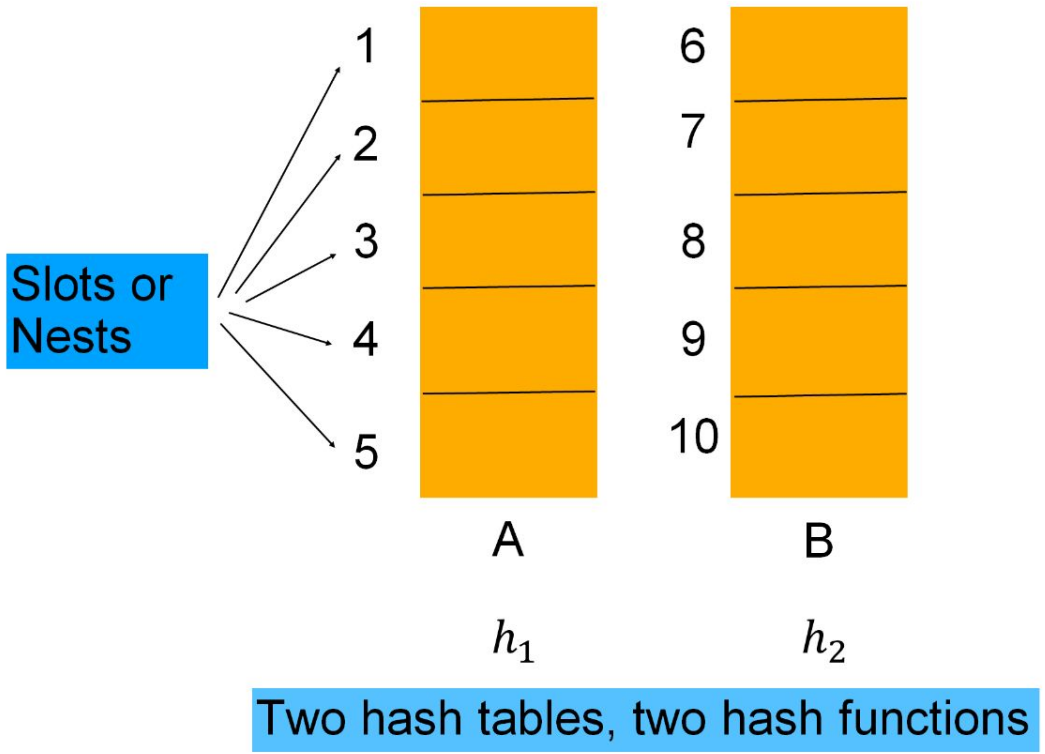
Birds or Keys = {a, b, c, d}

$h_1(a) = 3$ $h_2(a) = 9$

$h_1(b) = 3$ $h_2(b) = 9$

$h_1(c) = 3$ $h_2(c) = 7$

$h_1(d) = 3$ $h_2(d) = 9$



Two hash tables, two hash functions

Slides from Kushagra

Cuckoo Hashing (Insertion)

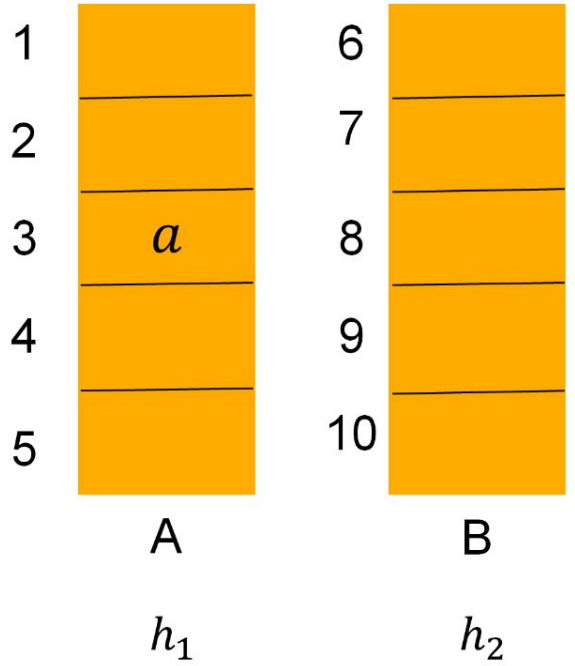
Birds or Keys = {a, b, c, d}

$h_1(a) = 3$ $h_2(a) = 9$

$h_1(b) = 3$ $h_2(b) = 9$

$h_1(c) = 3$ $h_2(c) = 7$

$h_1(d) = 3$ $h_2(d) = 9$



Insert a. We prefer to insert in table A more than table B

Slides from Kushagra

Cuckoo Hashing (Insertion)

Birds or Keys = {a, b, c, d}

$h_1(a) = 3$ $h_2(a) = 9$

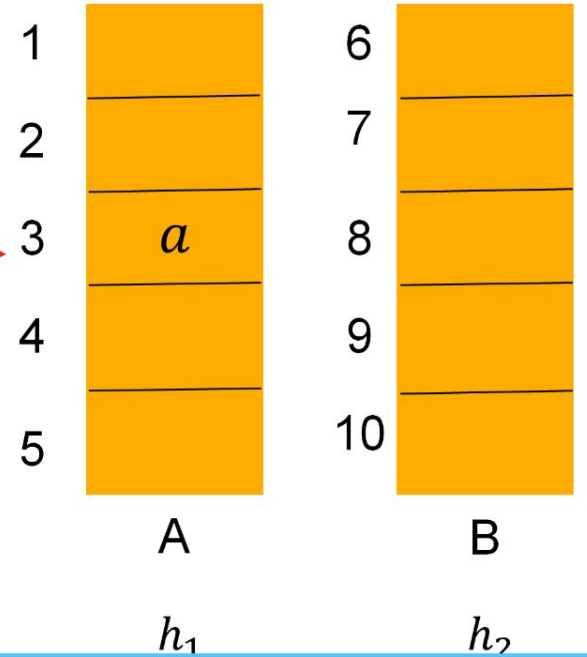
$h_1(b) = 3$ $h_2(b) = 9$

$h_1(c) = 3$ $h_2(c) = 7$

$h_1(d) = 3$ $h_2(d) = 9$

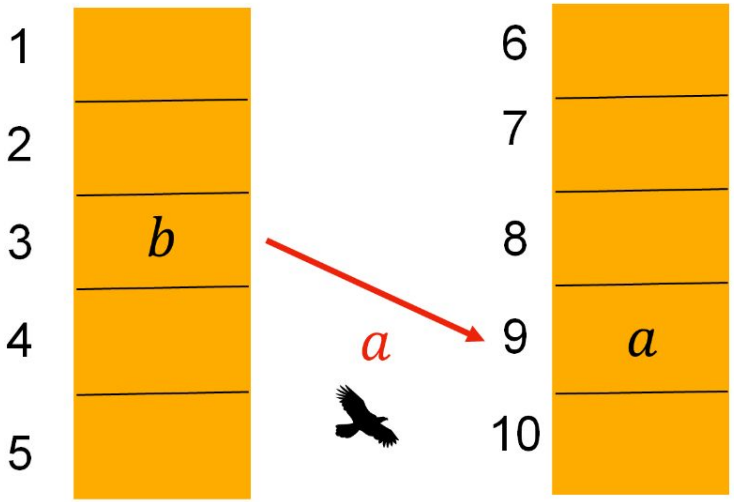
b

Nestless cuckoo bird b looking for a nest



Insert b, but nest 3 is not empty in Table A. What should we do now (any guesses)?

Slides from Kushagra



Kicks out the present resident of the nest

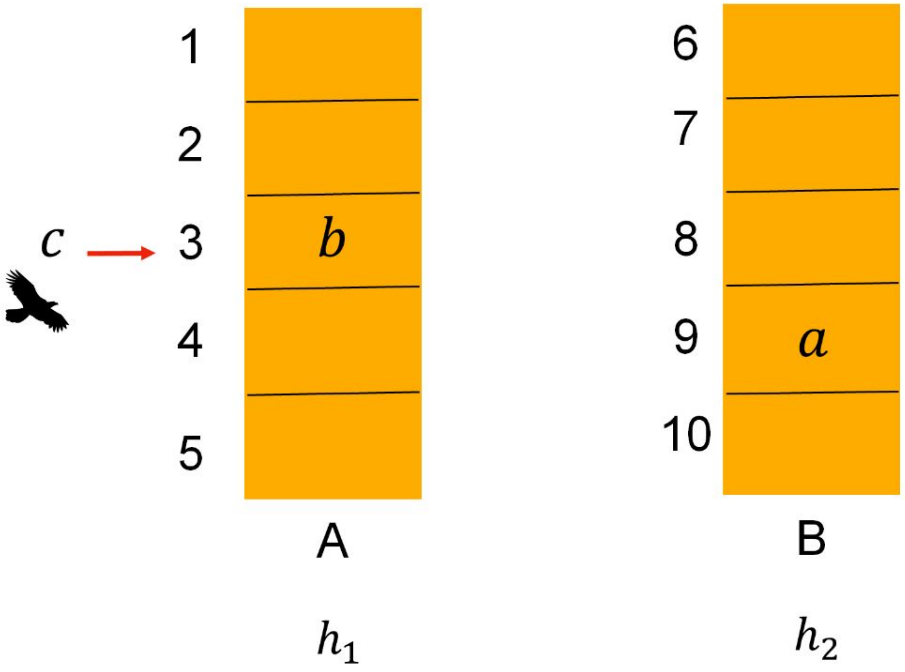
Nestless bird a flies to its other available nest in table B

Nestless Sequence: { b, a }

Slides from Kushagra $h_2(a) = 9$

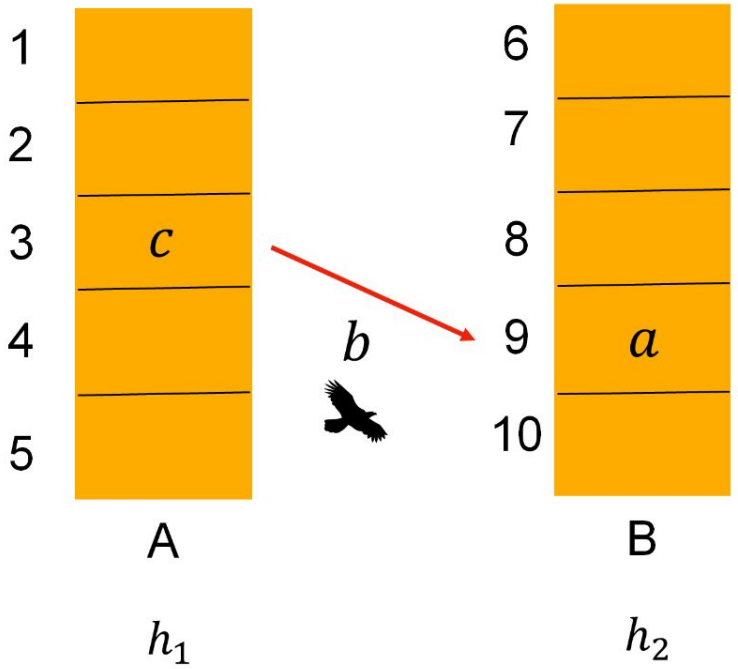
(Answer to Problem 2(a) (1))

Slides from Kushagra



Insert c

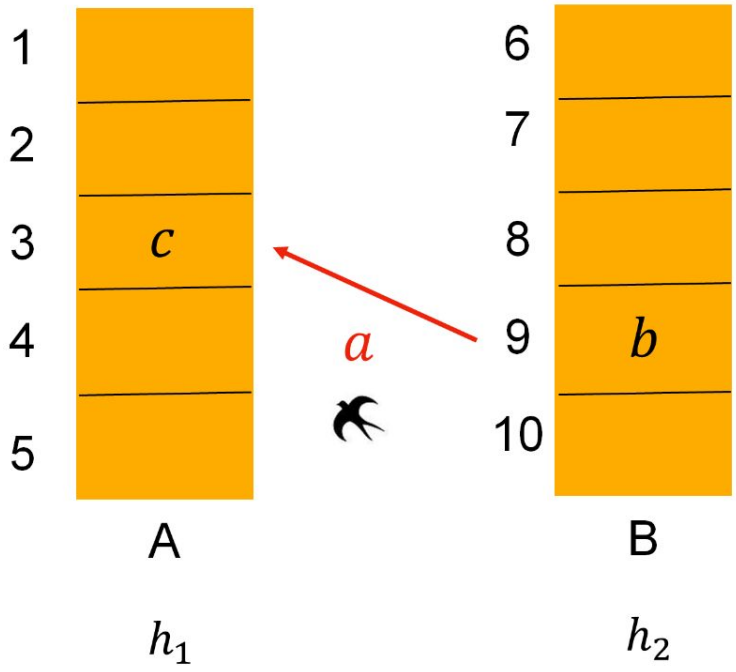
$h_1(c) = 3$



$h_2(b) = 9$

Nestless Sequence: {c, b, ..

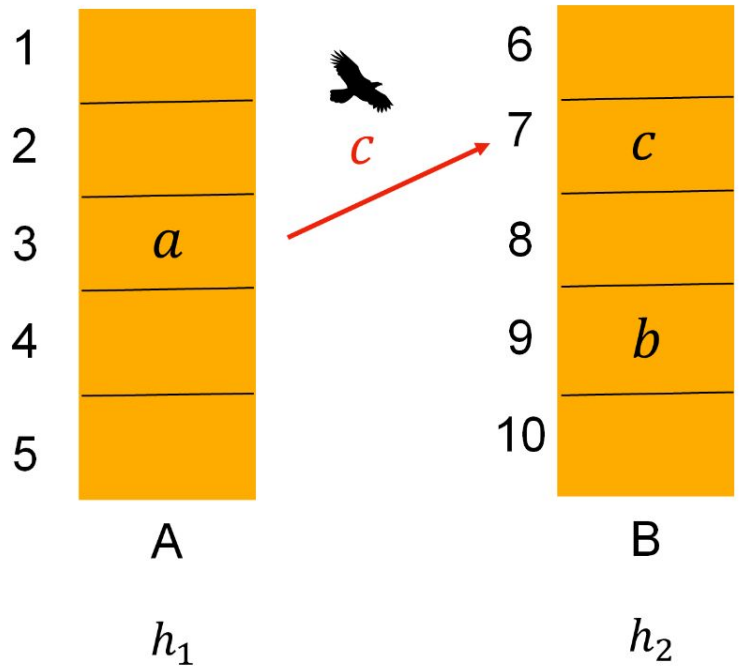
Slides from Kushagra



$h_1(a) = 3$

Nestless Sequence: {c, b, a, ..

Our insertion process terminates



$h_2(c) = 7$

Nestless Sequence: {c, b, a, c}

(Answer to Problem 2(a) (2))

2C Benefits, if this works.

```
insert(item x, Table T, hashfunction h)
      slot = h(x)
      z = T[slot]
      T[slot] = x
      if (z != null)
            if (T == A)
                  insert(z, B, g)
            else if (T == B)
                  insert(z, A, f)


search(x)
      if (A[f(x)] == x) then return A[f(x)];
      if (B[g(x)] == x) then return B[g(x)];
      else return NOT_IN_TABLE;
```

There's one obvious benefit:

2C Benefits, if this works.

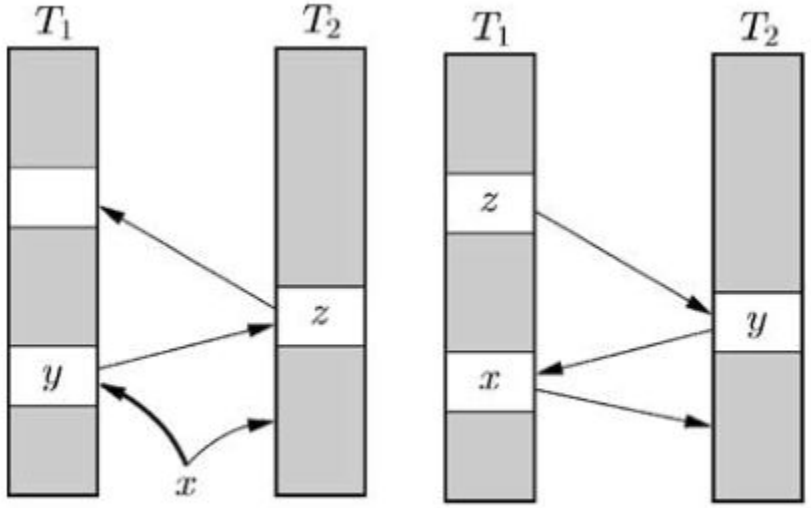
```
insert(item x, Table T, hashfunction h)
      slot = h(x)
      z = T[slot]
      T[slot] = x
      if (z != null)
            if (T == A)
                  insert(z, B, g)
            else if (T == B)
                  insert(z, A, f)


search(x)
      if (A[f(x)] == x) then return A[f(x)];
      if (B[g(x)] == x) then return B[g(x)];
      else return NOT_IN_TABLE;
```

There's one obvious benefit:

O(1) search/delete!

Each element only has 2 possible spots!

Example - Case 1



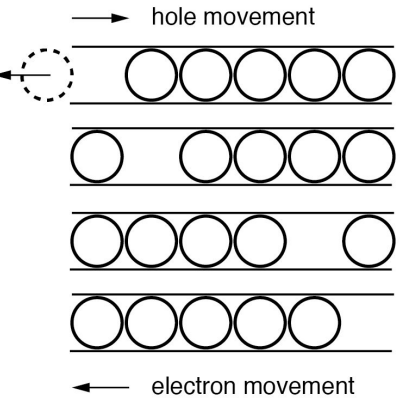
Nestless keys:

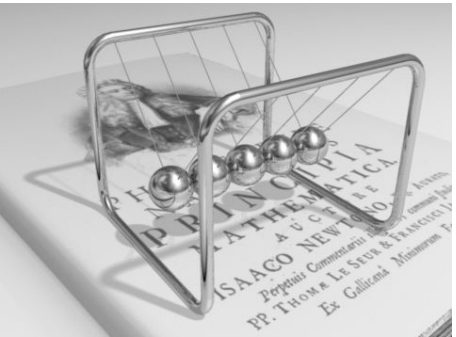
- Items being kicked.
- In this case no cycle:
  - Keys are distinct.

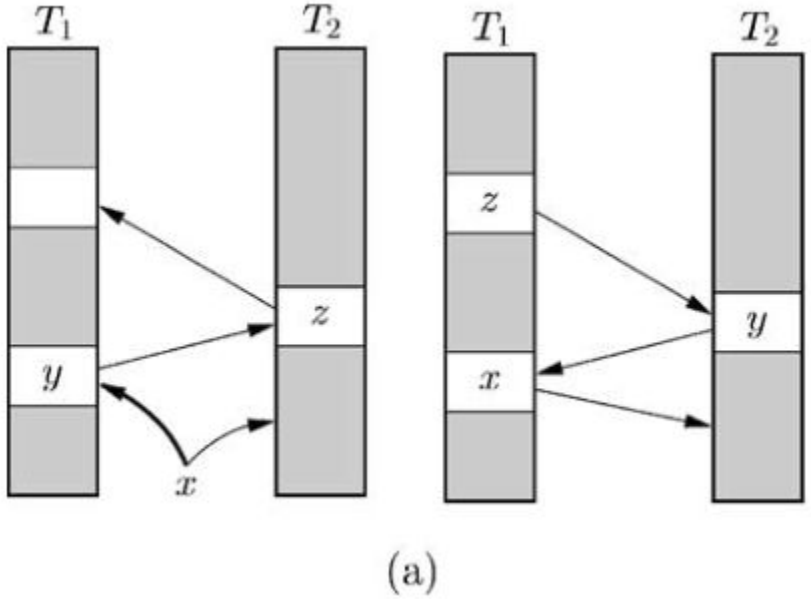
Note the change in direction of arrows!

Each element only has 2 possible spots!

Example - Case 1







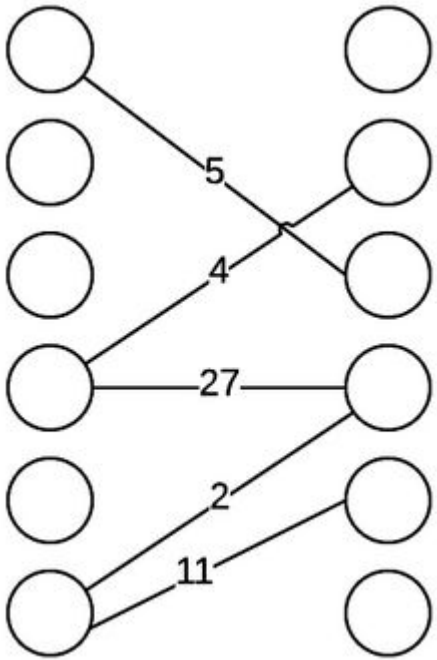
(a)

Nestless keys:

- Items being kicked.
- In this case no cycle:
  - Keys are distinct.

Note the change in direction of arrows!

Cuckoo graph



A cuckoo graph is a bipartite graph.

- Every item inserted can be at 2 positions

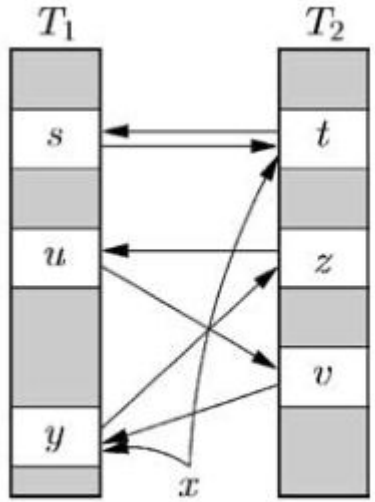
Failure case

Obvious:

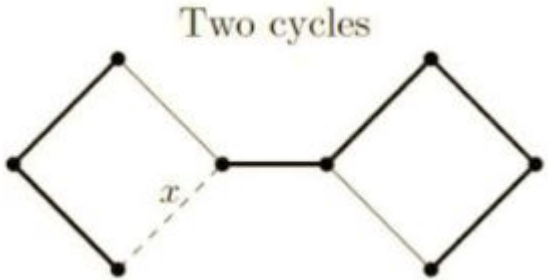
- Infinite loop

Less obvious:

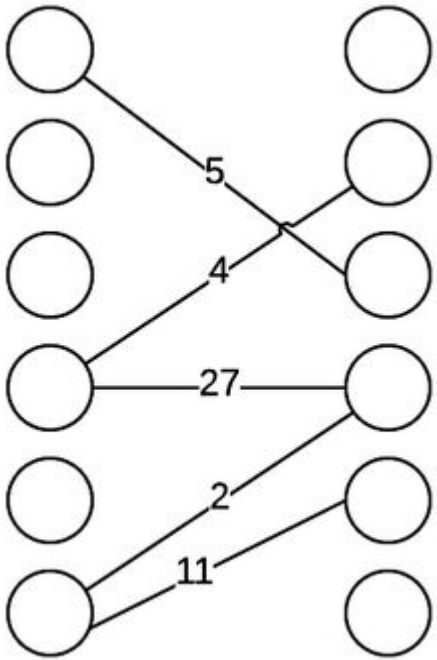
- If x was inserted into A and kicked, it will try inserting into B.
- Nestless keys not distinct
- Terminates if one cycle (case 2)
  - Infinite loop if two cycles (case 3)



(b)



Cuckoo graph



A cuckoo graph is a bipartite graph.

- Every item inserted can be at 2 positions

Student Joel observed:

- 1 cycle: n edges n nodes, life is good
- 2 cycles: n+1 edges, n nodes, not enough space.

Note: 1 edge = 1 item, 1 node = 1 position.

Pigeonhole principle.

Analysis slide

T(n) = O(chain length) ← Success (and also failure)

\+ O(explosion) * P(explosion) ← Failure

We think about what could happen during insertion.

Analysis slide

T(n) = O(chain length) ← Success (and also failure)

\+ O(explosion) * P(explosion) ← Failure

We want to show this is not too expensive.
I.e. competitive with other hashing like linear probing.

Analysis slide

T(n) = O(chain length) ← Success (and also failure)

\+ O(explosion) * P(explosion) ← Failure

We want to show this is not too expensive.
I.e. competitive with other hashing like linear probing.

- O(1) expected
- O(log n) expected worst case

Analysis slide

E[T(n)] = E[O(chain length)] ← Success (and also failure)

\+ E[O(explosion) * P(explosion)] ← Failure

= O(1)

We want to show this is not too expensive.
I.e. competitive with other hashing like linear probing.

- **O(1) expected**
- O(log n) expected worst case

Analysis slide

E[T(n)] = E[O(chain length)] ← Success (and also failure)

\+ E[O(explosion) * P(explosion)] ← Failure

= O(1)

Conclude:

- E[O(chain length)] = O(1)
- E[O(explosion) * P(explosion)] = O(1)

Analysis

While search / delete are O(1)

Insertion has issues:

- Insertion might cause multiple evictions:
  - Maximum Cost: O([redacted])
  - Expected Cost: O(E[redacted])
- Insertion might cause explosion
  - Cost of explosion: O([redacted])
  - Probability of explosion: ????

Intuition?

Analysis

While search / delete are O(1)

Insertion has issues:

- Insertion might cause multiple evictions:
  - Maximum Cost: O(Chain length)
  - Expected Cost: O(E[Chain length])
- Insertion might cause explosion
  - Cost of explosion: O(n)
  - Probability of explosion: ???? ← What do we require?

Analysis

Interesting fact:
In the original paper, they just defined a maximum loop count at O(log n)

While search / delete are O(1)

Insertion has issues:

- Insertion might cause multiple evictions:
  - Maximum Cost: O(Chain length)
  - Expected Cost: O(E[Chain length])
- Insertion might cause explosion
  - Cost of explosion: O(n)
  - Probability of explosion: O(1/n) ← What do we require?

$$T(n) = O(\text{chain length}) + \text{Cost}(\text{explosion}) * P(\text{explosion})$$

$$E[T(n)] = O(E[\text{chain length}]) + E[\text{Cost}(\text{explosion}) * P(\text{explosion})]$$

Analysis

Interesting fact:
In the original paper, they just defined a maximum loop count at O(log n)

While search / delete are O(1)

Insertion has issues:

- Insertion might cause multiple evictions:
  - Maximum Cost: O(Chain length)
  - Expected Cost: O(E[Chain length])
- Insertion might cause explosion
  - Cost of explosion: O(n)
  - Probability of explosion: O(1/n) ← What do we require?

Can anyone spot the flaw?

$$T(n) = O(\text{chain length}) + \text{Cost}(\text{explosion}) * P(\text{explosion})$$

$$E[T(n)] = O(E[\text{chain length}]) + E[\text{Cost}(\text{explosion}) * P(\text{explosion})]$$

Analysis

Interesting fact:
In the original paper, they just defined a maximum loop count at O(log n)

While search / delete are O(1)

Insertion has issues:

- Insertion might cause multiple evictions:
  - Maximum Cost: O(Chain length)
  - Expected Cost: O(E[Chain length])
- Insertion might cause explosion
  - Cost of explosion: O(n)
  - Probability of explosion: O(1/n) ← What do we require?

Can anyone spot the flaw?

Rehashing may trigger rehashing

$$T(n) = O(\text{chain length}) + \text{Cost(explosion)} * P(\text{explosion})$$

$$E[T(n)] = O(E[\text{chain length}]) + E[\text{Cost(explosion)} * P(\text{explosion})]$$

Rehashing

T(n) = O(chain length) + Cost(explosion) * P(explosion)

E[T(n)] = O(E[chain length]) + E[Cost(explosion) * P(explosion)]
E[T(n)] = O(E[chain length]) + E[(T(1)+T(2)+...+T(n)) * P(explosion)]
<= O(E[chain length]) + E[n*T(n)] * P(explosion)]

Hence need P(explosion) -> O(1/n^2)

Suspect the above statement is sufficient, not necessary.

Rehashing (discussion)

T(n) = O(chain length) + Cost(explosion) * P(explosion)

E[T(n)] = O(E[chain length]) + E[Cost(explosion) * P(explosion)]
E[T(n)] = O(E[chain length]) + E[(T(1)+T(2)+...+T(n)) * P(explosion)]
<= O(E[chain length]) + E[n*T(n)] * P(explosion)]

Hence need P(explosion) -> O(1/n^2)

Intuitively:
Rehashing has n inserts, if the probability of each failing is O(1/n), quite likely that one will fail.
If it's O(1/n^2) each, then the entire rehashing has O(1/n) failure probability.

Key lesson

You can estimate what costs need to be for an algorithm to work without actually writing the algorithm

- Do this by breaking it into parts then looking at the performance of each part.

I use this in my research to rapidly reject many standard solutions!

- E.g. my algorithm needs to be O(d) per step (for d dimensions)
  - (d is large for machine learning)
- Means i cannot use anything which finds d basis vectors, SVD, etc.

Goals

Produce intuition that Cuckoo hashing is competitive with Linear Probing.

Introduce mathematics for the following:

- Show the “expected worst case”
- Show the derivation of E[chain length]

Linear probing revision

- If the table is sufficiently empty(< 1/4 full) then:
  - O(1) expected runtime
  - O(log n) expected worst case (from cluster sizes)
- Worst case is still O(n)

Length of nestless key chain

Intuition:

- Make an argument that this is highly similar to another hashing algorithm
- Use the result from there to upper bound the chain length.

Length of nestless key chain

Three bad arguments:

- To give you some intuition that Cuckoo hashing is “better than” linear probing

Proof by eminent authority →

Length of nestless key chain

Bad argument 1:

Cuckoo hashing

Rasmus Pagh [a,*,1] and Flemming Friche Rodler [b,2]

[a] *IT University of Copenhagen, Rued Langgaardsvej 7, 2300 København S, Denmark*
[b] *ON-AIR A/S, Digtervejen 9, 9200 Aalborg SV, Denmark*

Received 23 January 2002

Abstract

We present a simple dictionary with worst case constant lookup time, equaling the theoretical performance of the classic dynamic perfect hashing scheme of Dietzfelbinger et al. [SIAM J. Comput. 23 (4) (1994) 738–761]. The space usage is similar to that of binary search trees. Besides being conceptually much simpler than previous dynamic dictionaries with worst case constant lookup time, our data structure is interesting in that it does not use perfect hashing, but rather a variant of open addressing where keys can be moved back in their probe sequences. An implementation inspired by our algorithm, but using weaker hash functions, is found to be quite practical. It is competitive with the best known dictionaries having an average case (but no nontrivial worst case) guarantee on lookup time.

Unfortunately, this isn't explained further

Length of nestless key chain

Bad argument 2:

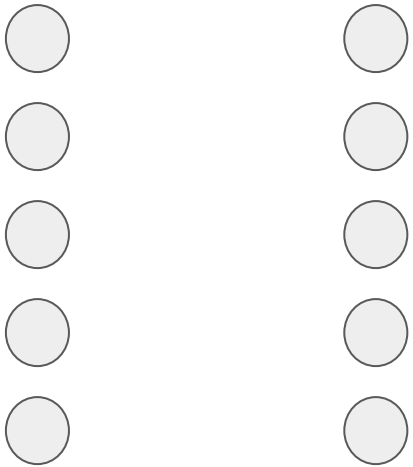
Reduce cuckoo hashing to linear probing by supplying a bad hash.

- Let f(x) = g(x) +- 1.

This generates two independent graphs

Notice that chains form if each edge is occupied (vs in linear probing, each slot)

(LP requires < ¼ full)



This argument is bad because probe sequence is much shorter than linear probing.

Length of nestless key chain

Bad argument 2:

Reduce cuckoo hashing to linear probing by supplying a bad hash.

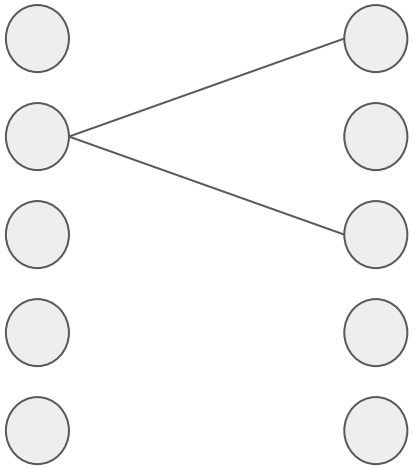
- Let f(x) = g(x) +- 1.

This generates two independent graphs

Notice that chains form if each edge is occupied (vs in linear probing, each slot)

(LP requires < ¼ full)

Correlated hash:
- f(x) = g(x) +- 1.



This argument is bad because probe sequence is much shorter than linear probing.

Length of nestless key chain

Bad argument 2:

Reduce cuckoo hashing to linear probing by supplying a bad hash.

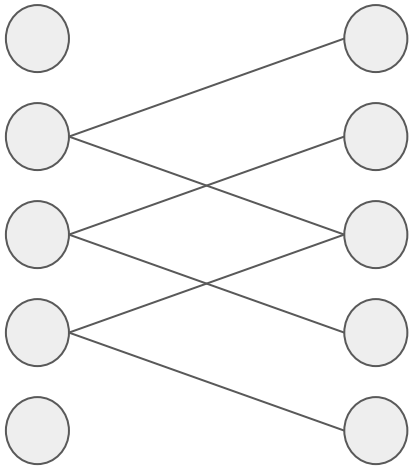
- Let f(x) = g(x) +- 1.

This generates two independent graphs

Notice that chains form if each edge is occupied (vs in linear probing, each slot)

(LP requires < ¼ full)

Correlated hash:
- f(x) = g(x) +- 1.



This argument is bad because probe sequence is much shorter than linear probing.

Length of nestless key chain

Bad argument 2:

Reduce cuckoo hashing to linear probing by supplying a bad hash.

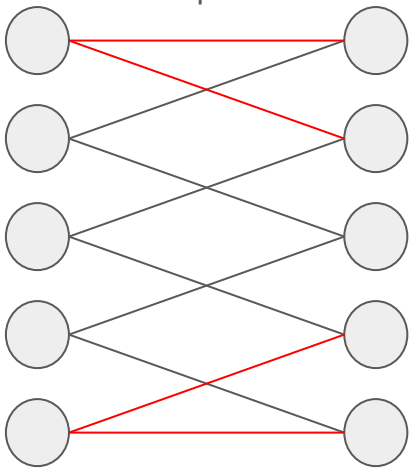
- Let f(x) = g(x) +- 1.

This generates two independent graphs

Notice that chains form if each edge is occupied (vs in linear probing, each slot)

(LP requires < ¼ full)

Correlated hash:
- f(x) = g(x) +- 1.
- Except end



Observe that this forms a ring, similar to linear probing

This argument is bad because probe sequence is much shorter than linear probing.

Length of nestless key chain

Bad argument 3:

Let p1, p2, …, pk be the probe sequence.

Then the probability that each of the buckets is filled is similar to linear probing.

- In particular, if you started part way, the probing order will still be the same.

This argument is bad because probe sequence depends on item probed.
But not that bad, the good one uses this idea :D

Good argument?

2. The insertion procedure has not yet entered a closed loop, as in Case 2 of Section 5.3.1. We will bound this probability by $2(1+\epsilon)^{-\frac{k-1}{3}+1}$, where ϵ is as in our table size invariant $r \geq (1+\epsilon)n$.

Analysis of case 2 When the insertion procedure has not entered a closed loop, Lemma 5.1 provides that for $v = \lceil k/3 \rceil$, there is a sequence of consecutive distinct nestless keys $b_1 = x, b_2, \cdots, b_v$ among the nestless keys in our insertion procedure. Since consecutive nestless keys in our insertion procedure by definition have hash functions equal, alternating among the two hash functions for each consecutive pair, for either $(\beta_1, \beta_2) = (1, 2)$ or $(\beta_2, \beta_2) = (2, 1)$,

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \cdots . \tag{5.2}$$

We wish to bound the probability this case occurs for each of the two choices of (β_1, β_2). Given the choice of $b_1 = x$, there are less than n^{v-1} possible ways to choose possible sequences of v distinct keys and further, since we assume that h_1 and h_2 are random functions from the key space to $\{0, 1, \cdots, r-1\}$, the equalities in 5.2 hold with probability $r^{-(v-1)}$. Together, this implies that the probability that there exists $b_1, \cdots, b_v$ as above is at most

$$2n^{v-1}r^{-(v-1)} = 2(r/n)^{-(v-1)} < 2(1+\epsilon)^{-\lceil k/3 \rceil + 1} \leq 2(1+\epsilon)^{-k/3+1}, \tag{5.3}$$

where we used our invariant $r/n < 1+\epsilon$. Since the probability of this case is bounded above by the probability that $b_1, \cdots, b_v$ as above exist, this case has probability bounded by 5.3.

Appendix:

- k is chain length
- v = k/3 is DISTINCT chain length
- r is the hash table capacity

This algorithm uses (1+epsilon) for the size of the hash table relative to number of items.

We simply use 2 in the argument.

Good argument explained

Suppose there's a distinct*, compact nestless chain of length k

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \cdots .$$

p p p

$$\begin{aligned}
\mathrm{P}(\text{one chain} \mid \text{n items inserted}) &= p^{k-1} \\
\mathrm{P}(\text{any chain} \mid \text{n items inserted}) &\leq n \cdot p^{k-1} \\
&\leq \frac{n}{2^{k-1}} \\
&= \frac{n}{2^{3log_2 n+1-1}} \\
&= \frac{n}{n^3} = \frac{1}{n^2}
\end{aligned}$$

has length = k

p = probability other slot is filled $= \frac{n}{m} \leq \frac{1}{2}$

Picking $k = 3log_2 n + 1$

(This suggests that log length chains are incredibly unlikely)

For large n, probability tends to 0.

Good argument explained

All the slots are already occupied with independent probability p.

Suppose there's a distinct*, compact nestless chain of length k

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \cdots .$$

p p p

$$\begin{aligned}
\mathrm{P}(\text{one chain} \mid \text{n items inserted}) &= p^{k-1} \\
\mathrm{P}(\text{any chain} \mid \text{n items inserted}) &\leq n \cdot p^{k-1} \\
&\leq \frac{n}{2^{k-1}} \\
&= \frac{n}{2^{3log_2 n+1-1}} \\
&= \frac{n}{n^3} = \frac{1}{n^2}
\end{aligned}$$

has length = k

p = probability other slot is filled $= \frac{n}{m} \leq \frac{1}{2}$

Picking $k = 3log_2 n + 1$

(This suggests that log length chains are incredibly unlikely)

For large n, probability tends to 0.

Good argument explained

All the slots are already occupied with independent probability p.

Suppose there's a distinct*, compact nestless chain of length k

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \cdots .$$

p p p

$$\begin{aligned}
\mathrm{P}(\text{one chain} \mid \text{n items inserted}) &= p^{k-1} \\
\mathrm{P}(\text{any chain} \mid \text{n items inserted}) &\leq n \cdot p^{k-1} \\
&\leq \frac{n}{2^{k-1}} \\
&= \frac{n}{2^{3log_2 n+1-1}} \\
&= \frac{n}{n^3} = \frac{1}{n^2}
\end{aligned}$$

k-1 slots are occupied to produce length k chain

has length = k

p = probability other slot is filled $= \frac{n}{m} \leq \frac{1}{2}$

Picking $k = 3log_2 n + 1$

(This suggests that log length chains are incredibly unlikely)

For large n, probability tends to 0.

Good argument explained

All the slots are already occupied with independent probability p.

Suppose there's a distinct*, compact nestless chain of length k

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \cdots .$$

p p p

$$\begin{aligned}
\mathrm{P}(\text{one chain} \mid \text{n items inserted}) &= p^{k-1} \\
\mathrm{P}(\text{any chain} \mid \text{n items inserted}) &\leq n \cdot p^{k-1} \\
&\leq \frac{n}{2^{k-1}} \\
&= \frac{n}{2^{3log_2 n+1-1}} \\
&= \frac{n}{n^3} = \frac{1}{n^2}
\end{aligned}$$

k-1 slots are occupied to produce length k chain

has length = k

p = probability other slot is filled $= \frac{n}{m} \leq \frac{1}{2}$

Picking $k = 3log_2 n + 1$

(This suggests that log length chains are incredibly unlikely)

For large n, probability tends to 0.

Want a low probability event: 1/n^2 for n = 1000 = 1 in 1 million

Good argument explained

All the slots are already occupied with independent probability p.

Suppose there's a distinct*, compact nestless chain of length k

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \cdots .$$

p p p

$$\begin{aligned}
\mathrm{P}(\text{one chain} \mid \text{n items inserted}) &= p^{k-1} \\
\mathrm{P}(\text{any chain} \mid \text{n items inserted}) &\leq n \cdot p^{k-1} \\
&\leq \frac{n}{2^{k-1}} \\
&= \frac{n}{2^{3log_2 n+1-1}} \\
&= \frac{n}{n^3} = \frac{1}{n^2}
\end{aligned}$$

k-1 slots are occupied to produce length k chain

has length = k

p = probability other slot is filled $= \frac{n}{m} \leq \frac{1}{2}$

Select table fullness and chain length to get probability required

Picking $k = 3log_2 n + 1$

(This suggests that log length chains are incredibly unlikely)

Want a low probability event: 1/n^2 for n = 1000 = 1 in 1 million

For large n, probability tends to 0.

Explanation of explanation

There's a lemma which states that any nestless sequence of length L have a distinct subsequence of length k=L/3.

- Distinct sequence is important, that gives us independent probabilities.
- Repeating sequences have correlation.

Case 2a. Nestless keys repeat, i.e. the nestless key x_i is evicted for the second time at position x_j (here and in the next case, for some (i, j), $x_i = x_j$, and we assume (i, j) is the lexicographically smallest ordered pair for which this condition holds), but the insertion process terminates when a nestless key moves to an empty bucket.

5.3.2 Insertion loop lemma

We will use the following lemma in our analysis of the expected number of iterations in the insertion loop.

Lemma 5.1. *Suppose at some step p, the insertion procedure for x produces a sequence $x_1 = x, x_2, \cdots, x_p$ of nestless keys, where no closed loop has yet been formed. Then, in $x_1, \cdots, x_p$, there exists a consecutive subsequence of length $l \geq p/3$ $x_q, x_{q+1}, \cdots, x_{q+l-1}$ of distinct nestless keys for which $x_q = x_1 = x$.*

Proof. Suppose all nestless keys $x_1, \cdots, x_p$ are distinct: then, $x_1, \cdots, x_p$ is such a sequence, so the lemma trivially holds. Now, if $p < i + j$, we have that the first $j - 1$ keys $x_1, \cdots, x_{j-1}$ are distinct, so since $j > i$, $j - 1 \geq (i + j - 1)/2 \geq p/2$, so $x_1, \cdots, x_{j-1}$ is the desired sequence. Now, if $p \geq i + j$, consider the sequences of distinct keys $x_1, \cdots, x_{j-1}$ and $x_{i+j-1}, \cdots, x_p$: we claim one of these must be of length at least $p/3$. These two sequences have $j - 1$ and $p - i - j + 2$ keys, respectively. Note that $p = (j - 1) + (i - 1) + (p - i - j + 2)$, and we know $j - 1 > i - 1$. If $j - 1 > p - i - j + 2$, then we know $3(j - 1) > p$, and otherwise, $3(p - i - j + 2) \geq p$. In either case, there is a sequence with properties as above. □

Implied: nothing else repeats. It's not obvious why nothing repeats.
Intuitively it's because if something is repeated it goes to the other table.

Explanation of explanation of explanation

Subgoal: Prove that any chain of length L with no closed loop has a distinct sub-chain of k = L/3

- A chain is infinite if you see the item 3x.
- Therefore the most you can see something is twice.

Given a chain of length L, if you cut it twice, the longest piece must be at least length L/3.

Case 2a. Nestless keys repeat, i.e. the nestless key x_i is evicted for the second time at position x_j (here and in the next case, for some (i, j), $x_i = x_j$, and we assume (i, j) is the lexicographically smallest ordered pair for which this condition holds), but the insertion process terminates when a nestless key moves to an empty bucket.

5.3.2 Insertion loop lemma

We will use the following lemma in our analysis of the expected number of iterations in the insertion loop.

Lemma 5.1. *Suppose at some step p, the insertion procedure for x produces a sequence $x_1 = x, x_2, \cdots, x_p$ of nestless keys, where no closed loop has yet been formed. Then, in $x_1, \cdots, x_p$, there exists a consecutive subsequence of length $l \geq p/3$ $x_q, x_{q+1}, \cdots, x_{q+l-1}$ of distinct nestless keys for which $x_q = x_1 = x$.*

Proof. Suppose all nestless keys $x_1, \cdots, x_p$ are distinct: then, $x_1, \cdots, x_p$ is such a sequence, so the lemma trivially holds. Now, if $p < i + j$, we have that the first $j - 1$ keys $x_1, \cdots, x_{j-1}$ are distinct, so since $j > i$, $j - 1 \geq (i + j - 1)/2 \geq p/2$, so $x_1, \cdots, x_{j-1}$ is the desired sequence. Now, if $p \geq i + j$, consider the sequences of distinct keys $x_1, \cdots, x_{j-1}$ and $x_{i+j-1}, \cdots, x_p$: we claim one of these must be of length at least $p/3$. These two sequences have $j - 1$ and $p - i - j + 2$ keys, respectively. Note that $p = (j - 1) + (i - 1) + (p - i - j + 2)$, and we know $j - 1 > i - 1$. If $j - 1 > p - i - j + 2$, then we know $3(j - 1) > p$, and otherwise, $3(p - i - j + 2) \geq p$. In either case, there is a sequence with properties as above. □

You can see why i hate integer math xD

Expected length

has length = k

$$\mathrm{P}(\text{one chain} \mid \text{n items inserted}) = p^{k-1}$$

E[chain length | n items inserted] $= E[p^{k-1}]$

has length = k

$= \sum_{k=1}^{\infty} k p^{k-1}$ p = probability other slot is filled $= \frac{n}{m} \leq \frac{1}{2}$

$\leq \sum_{k=1}^{\infty} \frac{k}{2^{k-1}}$

$= O(1)$ Sequence converges

2A.1/2 Conclusion

If it terminates:

- Expected cycle length is O(1)
  - Argument from linear probing
  - Or you can sum the last part.
- Expected Worst case O(log n)
  - Argument from linear probing.
  - Or you can take the probability of the last part.

Rehashing

If the insertion fails, rebuild whole table.

We can detect this by checking for O(n) iterations (as long as it’s rare enough).

- Original paper just uses ~3 log n

Rehashing

If the insertion fails, rebuild whole table.

Rehashing costs O(n) on average.

- We already know how to solve this:
  - Doubling the hash table works.

“ The basic idea behind Cuckoo Hashing is that we use two tables A and B of size m = 2n to store n items”

Except: Rehashing could fail.

Rehashing failure probability

http://web.stanford.edu/class/archive/cs/cs166/cs166.1196/lectures/13/Slides13.pdf

The probability that a single insertion fails is $O(1 / m^2)$ if $m = (1+\varepsilon)n$.

Question 1: What is the probability at least one insert fails if we do n total insertions?

$$\begin{aligned}
&\Pr[\text{some insert fails}] \\
\leq\ &\sum_{k=1}^{n} \Pr[\text{the } k\text{th insert fails}] \\
=\ &\sum_{k=1}^{n} O\left(\frac{1}{m^2}\right) \qquad \leftarrow \textbf{This one is out of syllabus} \\
=\ &O\left(\frac{n}{m^2}\right) \\
=\ &\mathbf{O\left(\frac{1}{m}\right)}
\end{aligned}$$

Self declared by me.

This is why it's out of syllabus

http://web.stanford.edu/class/archive/cs/cs166/cs166.1196/lectures/13/Slides13.pdf

$$\sum_{k=1}^{n}\left(\frac{(k+1)^4\, m^{k-1}\, n^k}{m^{2k}\, m}\right) = \sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{k-1-2k-1}\right)$$

$$= \sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{k-2}\right)$$

$$= \frac{1}{m^2}\sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{-k}\right)$$

$$= \frac{1}{m^2}\sum_{k=1}^{n}(k+1)^4\left(\frac{n}{m}\right)^k$$

$$= \frac{1}{m^2}\sum_{k=1}^{n}\frac{(k+1)^4}{(1+\varepsilon)^k}$$

$$= \frac{1}{m^2}\cdot O(1)$$

$$= \mathbf{O\left(\frac{1}{m^2}\right)}$$

Analysis of case 3 Here, we wish to bound the probability that our sequence of nestless keys $x_1, \cdots, x_k$ has entered a closed loop. Let $v \leq k$ be the number of distinct nestless keys. We can choose the keys other than $x_1 = x$ in less than n^{v-1} ways and among the keys, we can assign them into buckets in r^{v-1} ways. In a closed loop, as in Section 5.3.1, we defined x_i and x_j to be such that (i, j) is the lexicographically first ordered pair for which $x_i = x_j$, $i \neq j$, and we defined x_l to be the first nestless key $l \geq i + j$ for which x_l has been encountered previously as $x_l = x_o$ for $1 \leq o \leq i + j - 1$. Here, as a crude approximation, we have at most v^3 possible values for i, j and l. Since each we assume our hash functions h_1 and h_2 are random, the probability for each configuration described above is r^{-2v}, since each of the v nestless keys y must in each configuration have set values of $h_1(y)$ and $h_2(y)$, each of which occurs with probability $1/r$. Putting this together and summing over possible v, we get that the probability of this case is at most

$$\sum_{v=3}^{l} n^{v-1} r^{v-1} v^3 r^{-2v} \leq \frac{1}{nr}\sum_{v=3}^{\infty} v^3 (n/r)^v < \frac{1}{nr}\sum_{v=3}^{\infty} v^3 (1+\epsilon)^{-v} = \frac{1}{nr} O(1) = O(1/n^2), \quad (5.4)$$

Rehashing failure probability

http://web.stanford.edu/class/archive/cs/cs166/cs166.1196/lectures/13/Slides13.pdf

Question 2: On expectation, how many rehashes are needed per insertion?

Let X be a random variable counting the number of rehashes assuming at least one rehash occurs.

X is geometrically distributed with success probability $1 - O(1 / m)$.

Some insertion failure in rehash operation

$$E[X] = \frac{1}{1 - O(1/m)} = \mathbf{O(1)}$$

In practice

Linear probing is still faster:

- Caching >> small big O gains

Cuckoo hashing is sensitive to hash function.