CS630 Graduate Algorithms

October 29, 2024

by Dora Erdos and Jeffrey Considine

- Hash tables

Expected Value

Let X be some random variable. Before performing the trial what do we expect the outcome to be?

For discrete random variables the expected value or mean is the weighted average of the range of X.

$$E[X] = \sum_{a \in R_X} a \cdot P(X = a)$$

Linearity of expectation: let X and Y be two discrete random variables and a and b be constants. Then

$$E[aX + b] = aE[X] + b$$

$$E[X + Y] = E[X] + E[Y]$$

exercise: prove this by writing out the definition.

Hash Tables

What is a hash table?

Hash Tables

Interface:

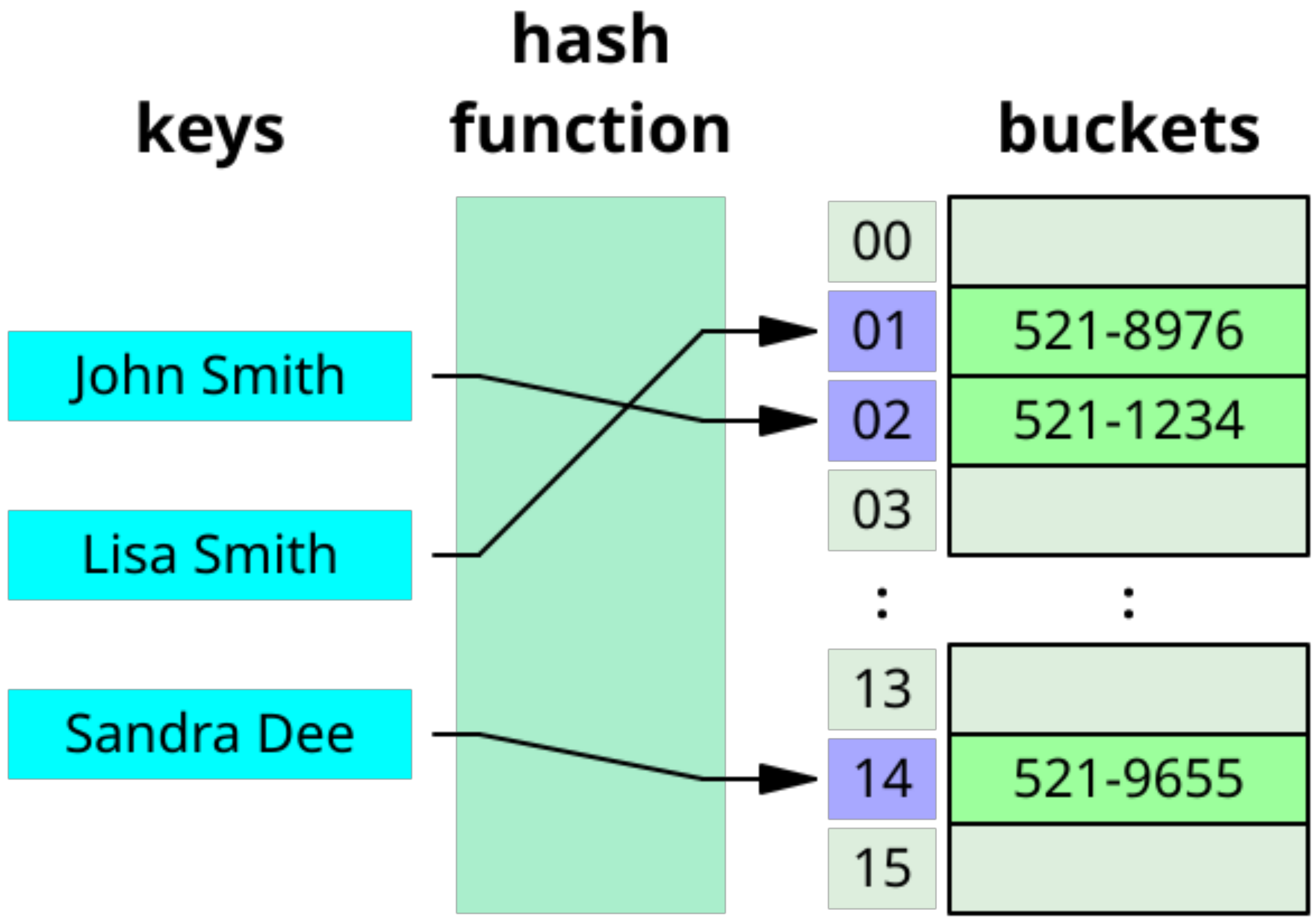
- A hash table is a data structure implementing a key-value mapping.
  - Insert
  - Lookup / search
  - Delete
- A common variant: skip values to implement a set interface.

Internals:

- Uses a “hash” function to evenly map keys across an array…
- Array entries are often called buckets.
- Bucket implementation details vary a lot to get different behaviors.

What are some examples of hash tables that you’ve used when programming?

Hash Tables



Values in hash table are phone numbers.

Image source: https://en.wikipedia.org/wiki/Hash_table

What is a hash function?

A hash function is a function used to map a key to an index in a hash table.

- Ideally maps keys uniformly at random to different indexes in the hash table.
  - Most of our analysis will assume this is the case.

What is a hash function?

A hash function is a function used to map a key to an index in a hash table.

- Ideally maps keys uniformly at random to different indexes in the hash table.
  - Most of our analysis will assume this is the case.

- In practice, we use simple functions for speed.
  - e.g. $h(x) = (17x + 39) \mod 64$
  - Cryptographic hash functions appear close to the ideal behavior, but too slow.

SHA-256 main loop: just repeat 64 times!

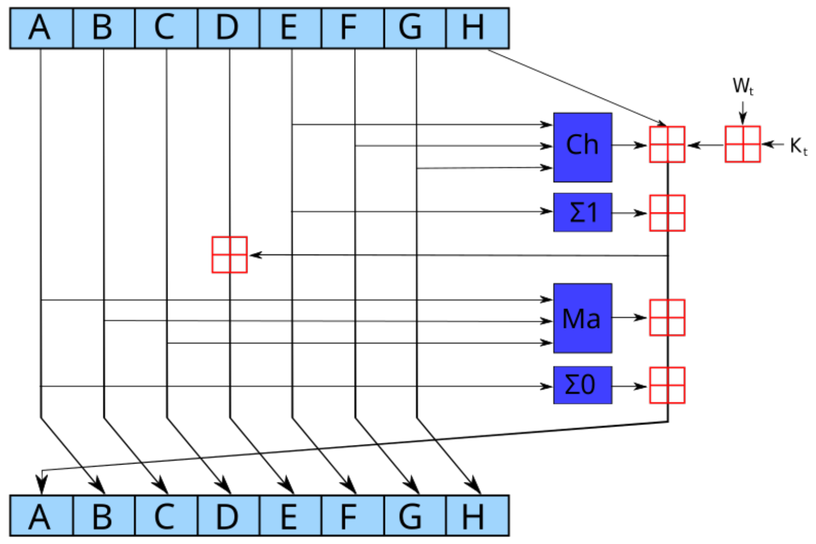


Image source: https://en.wikipedia.org/wiki/SHA-2

Hash Tables and Randomized Load Balancing

Random assignment of items uniformly at random should sound familiar…

How did randomized load balancing work last week?

Top Hat

Given an idealized hash table with n buckets or randomized load balancing with n servers, which of the following are true?

1. Items are assigned uniformly at random to hash table buckets and servers.

2. If n items are assigned, then at least $0.9n$ buckets or servers have at least one item assigned with probability at least $1 - 1/n$.

3. If n items are assigned, the maximum load is $\Theta(\log n / \log \log n)$ with probability at least $1 - 1/n$.

Handling Collisions

With randomized load balancing, a server assigned multiple items or requests might be slow…

How does a hash table handle many items being assigned to the same bucket?

Aside: Perfect Hashing

For a particular set of keys, it is possible to construct a “perfect” hash function that maps each key to a different bucket in the hash table.

- Generally take linear space to construct.
- Niche usage since updates are more expensive.
  - The hash function needs to be adjusted with each key change.
  - O(1) evaluations and updates are possible.
- Space per item is higher, and usually we prefer leaner hash tables.

We will be focused on imperfect hash functions, with idealized randomness instead.

- Perfect hashing may be useful if you have data that rarely (never) changes and you need O(1) guarantee.

Hashing with Separate Chaining

Handle collisions with a linked list from each bucket.

- Usually analysis assumes adding to the end.
- Easiest variant to analyze
- Less space efficient from linked list overhead and memory management

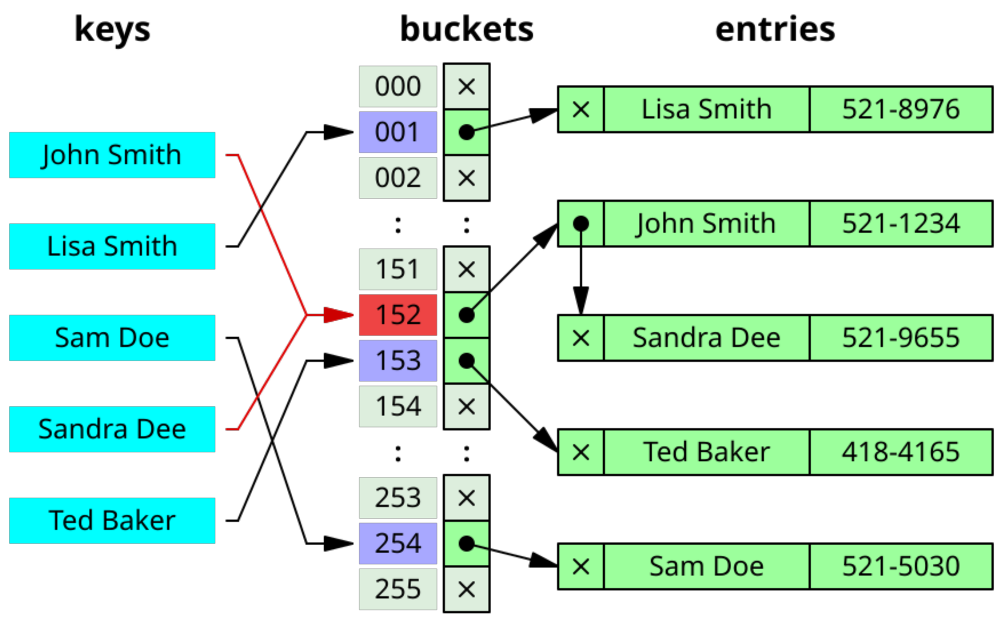


Image source: https://en.wikipedia.org/wiki/Hash_table

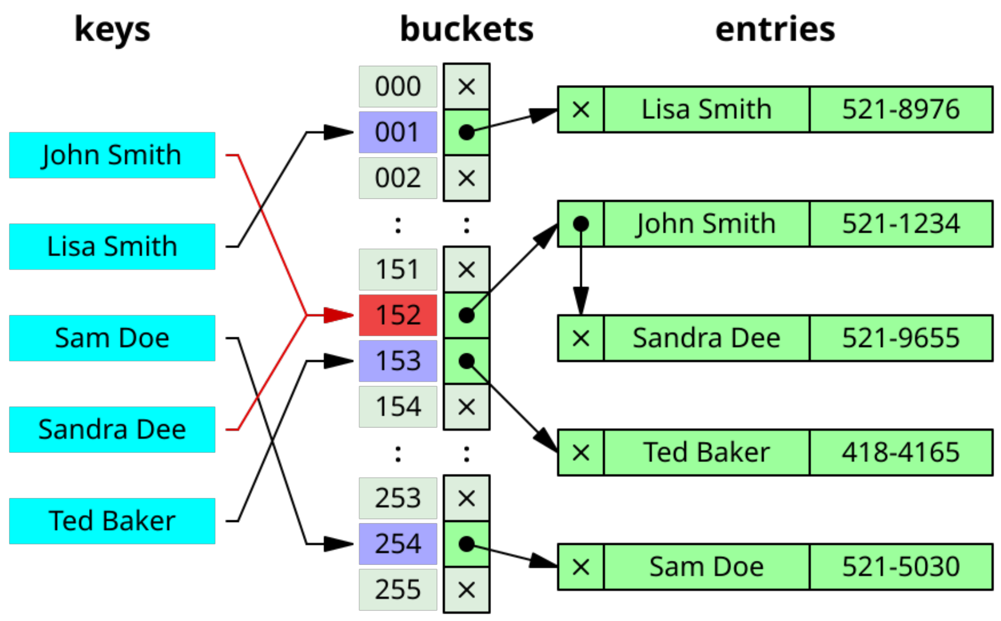
Hashing with Separate Chaining

Handle collisions with a linked list from each bucket.

- Usually analysis assumes adding to the end.
- Easiest variant to analyze
- Less space efficient from linked list overhead and memory management

Two sub variants of note -

- Put first key/value pair in the bucket, then link to collisions.
- Use a sorted tree for faster processing of big buckets.



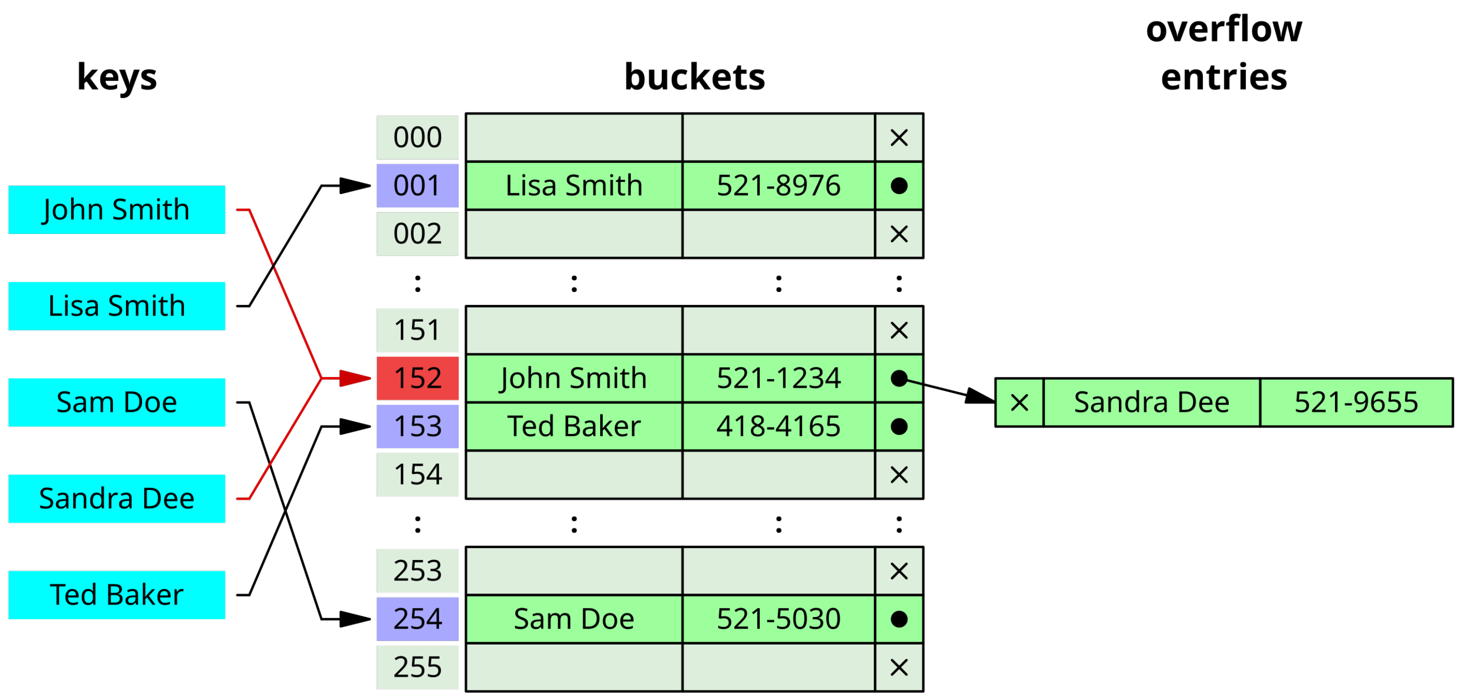


Image source: https://en.wikipedia.org/wiki/Hash_table

How to Analyze? Load Factors

A hash table with m items inserted into n buckets has load factor $\alpha = m/n$.

So the average number of items per table entry is?

How to Analyze? Load Factors

A hash table with m items inserted into n buckets has load factor $\alpha = m/n$.

So the average number of items per table entry is α.

And the cost to scan a uniform random table entry is?

How to Analyze? Load Factors

A hash table with m items inserted into n buckets has load factor $\alpha = m/n$.

So the average number of items per table entry is α.

And the cost to scan a uniform random table entry is $1 + \alpha$.

Average Case Performance of Separate Chaining

Search:

- Given any key k, how long do does it take to find k in a hash table or determine that it is not in the hash table?

Average Case Performance of Separate Chaining

Search:

- Given any key k, how long do does it take to find k in a hash table or determine that it is not in the hash table?

- Let l_i be the current length of bucket i.
- Then the average time to search is $O(S(m, n))$ where

$$\begin{aligned} S(m,n) &= E_i[1 + l_i] \\ &= \sum_i \frac{1}{n}(1 + l_i) \\ &= \frac{1}{n}\sum_i (1 + l_i) \\ &= \frac{1}{n}(n + m) \\ &= 1 + \alpha \end{aligned}$$

Average Case Performance of Separate Chaining

Insertion:

- Given any key k and value v to insert in a hash table, how long does it take to insert them into the hash table?

Average Case Performance of Separate Chaining

Insertion:

- Given any key k and value v to insert in a hash table, how long does it take to insert them into the hash table?

Do we care about duplicates?

- Probably…
- But if we know there are no duplicates, just insert at the head of the list for $O(1)$ worst case.

Average Case Performance of Hashing with Chaining

Insertion:

- Given any key k and value v to insert in a hash table, how long does it take to insert them into the hash table?
- Then the average time to search is bounded by $O(I(m,n))$ where

$$\begin{aligned} I(m,n) &= E_i[1+l_i] \\ &= \sum_i \frac{1}{n}(1+l_i) \\ &= \frac{1}{n}\sum_i (1+l_i) \\ &= \frac{1}{n}(n+m) \\ &= 1+\alpha \end{aligned}$$

-

- Does it matter if k was already in the hash table?

Average Case Performance of Hashing with Chaining

Deletion:

- Same as insertion.

Average Case Performance of m Hash Table Operations

How long does it take to perform any m hash table operations?

Average Case Performance of m Hash Table Operations

How long does it take to perform any m hash table operations?

$O(m(1 + m/n)) = O(m + m^2/n)$?

Average Case Performance of m Hash Table Operations

How long does it take to perform any m hash table operations?

$O(m(1 + m/n)) = O(m + m^2/n)$?

Does every operation take $O(1 + m/n)$ time?

Average Case Performance of m Hash Table Operations

How long does it take to perform any m hash table operations?

$O(m(1+m/n)) = O(m+m^2/n)$?

Does every operation take $O(1+m/n)$ time?

No, just the later ones.

But the last $m/2$ operations will take $\Theta(m(1+m/n))$ expected time.

Average Case Performance of m Hash Table Operations

How long does it take to perform any m hash table operations?

$O(m(1 + m/n)) = O(m + m^2/n)$?

Does every operation take $O(1 + m/n)$ time?

No, just the later ones.

But the last $m/2$ operations will take $\Theta(m(1 + m/n))$ expected time.

What does this mean for practical use of hash tables?

Pros and Cons of Hashing with Separate Chaining

Pros:

- Simple use of basic data structures
- Easy to analyze

Cons:

- Overhead for linked lists
- Poor memory locality when traversing linked lists?
- Worst case costs are pretty bad even if we prove they are rare.
- Even without worst cases, actual times vary a lot.

Space-Sensitive Applications

Space efficiency is sometimes really important

- Does everything fit in memory?
- Critical for cache efficiency!
  - Less space per entry

    → more entries

    → more cache hits

    → more performance

Relative importance does vary based on key vs value vs pointer sizes.

Critical for

- Processor caches
- API/database cache

Not critical for

- File cache (operating system)

Worst Case Performance of Separate Chaining

What is the absolute worst case scenario for search or insertion?

Attacks on Hash Tables

If you know what kinds of data a web server caches, and what hash function it uses,

- You can construct a lot of requests that hash to the same bucket.
  - This is expensive, but can be done offline.
  - 32 bit hashes easy enough to brute force.
  - Easier if you know the current table size.

https://static.usenix.org/event/sec03/tech/full_papers/crosby/crosby_html/

Attacks were actually known much earlier, and had been actually carried out in 1990s.

Hash Table Defenses

Basic ideas:

- Don’t use a fixed hash function. Pick hash function parameters randomly on startup.
- Use a better family of hash function. Some families are still easy to attack when rotating.
- Rebuild the hash table with a new hash function if worst case observed.

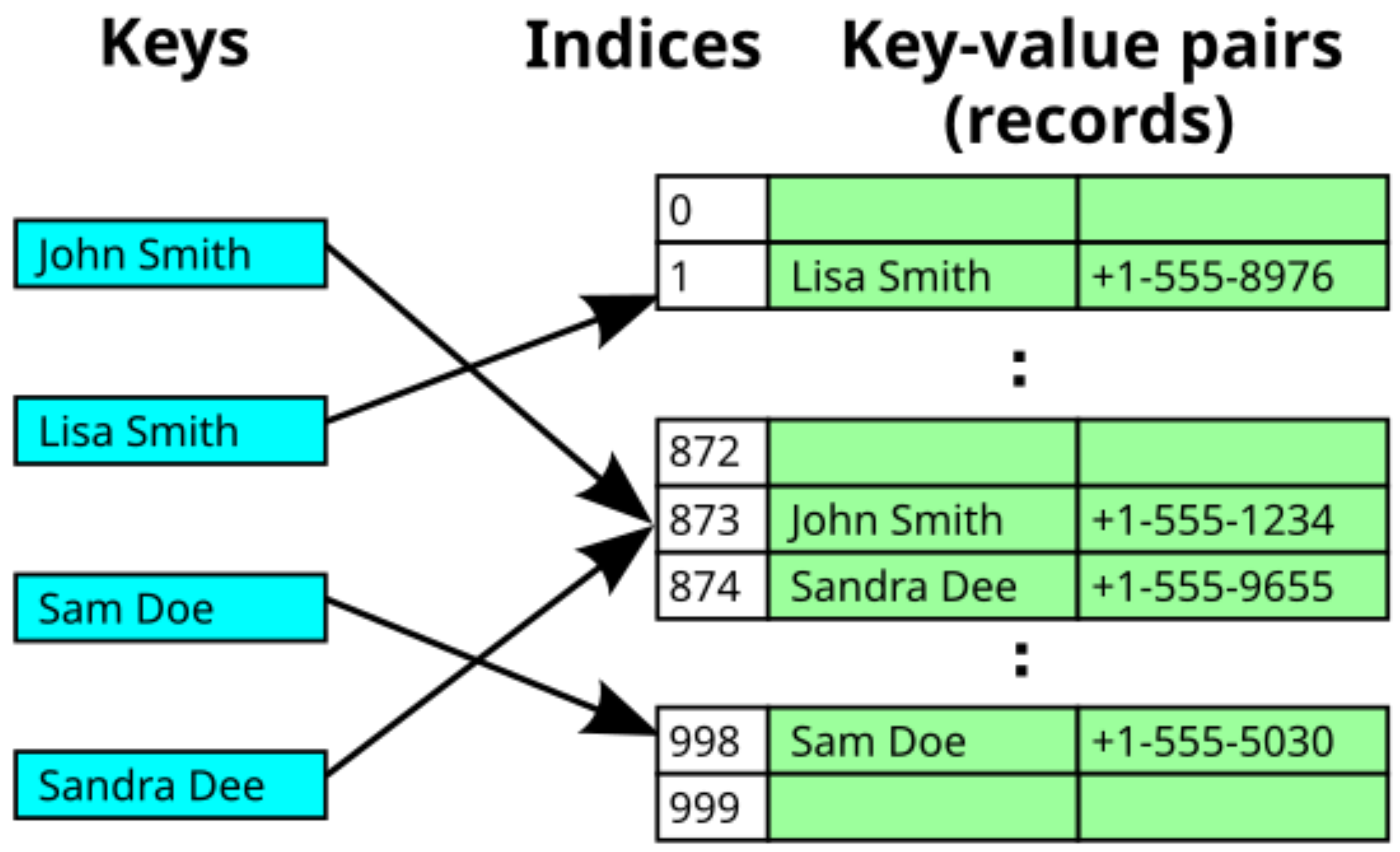
Other ideas:

- Resize hash table if possible. But probably already memory limits.
- Use fancier bucket data structures, but those can exacerbate space usage.

Open Addressing or Closed Hashing

Alternative to separate chaining

- If there is a collision when inserting, pick another bucket.
- When searching, if another key is found, repeat the "pick another bucket" process until you find your key or an empty bucket.



Linear Probing

Insertion:

- If the bucket picked by the hash function is already in use, keep looking at the next bucket until an empty bucket is found.
  - Memory locality is great.

Linear Probing

Insertion:

- If collision on $h(x)$, try $h(x) + 1,\ h(x) + 2,\ h(x) + 3,\ \ldots$ until an empty bucket is found.
  - Memory locality is great.
- What happens if the hash table is full?

Linear Probing

Insertion:

- If collision on $h(x)$, try $h(x)+1,\ h(x)+2,\ h(x)+3,\ \ldots$ until an empty bucket is found.
    - Memory locality is great.
- What happens if the hash table is full?
    - If the hash table is full, then this will loop forever.
    - Will get really slow when almost full.

Linear Probing

Insertion:

- If collision on $h(x)$, try $h(x) + 1,\ h(x) + 2,\ h(x) + 3,\ \ldots$ until an empty bucket is found.
  - Memory locality is great.
- What happens if the hash table is full?
  - If the hash table is full, then this will loop forever.
  - Will get really slow when almost full.
- Primary clustering problem
  - Collisions create short runs of full buckets.
  - But if those short runs catch up to another run of full buckets, they combine.

Linear Probing

Insertion:

- If collision on $h(x)$, try $h(x)+1,\ h(x)+2,\ h(x)+3,\ \ldots$ until an empty bucket is found.
  - Memory locality is great.
- What happens if the hash table is full?
  - If the hash table is full, then this will loop forever.
  - Will get really slow when almost full.
- Primary clustering problem
  - Collisions create short runs of full buckets.
  - But if those short runs catch up to another run of full buckets, they combine.
- Leads to insertion times of $\Theta\left(1+\frac{1}{(1-\alpha)^2}\right)$
- Searches for items not in hash table take the same time.

Linear Probing

Insertion:

- If collision on $h(x)$, try $h(x)+1,\ h(x)+2,\ h(x)+3,\ \ldots$ until an empty bucket is found.
  - Memory locality is great.
- What happens if the hash table is full?
  - If the hash table is full, then this will loop forever.
  - Will get really slow when almost full.
- Primary clustering problem
  - Collisions create short runs of full buckets.
  - But if those short runs catch up to another run of full buckets, they combine.
- Leads to insertion times of $\Theta\left(1+\frac{1}{(1-\alpha)^2}\right)$
- Searches for items not in hash table take the same time.
- Searches for items in hash table somewhat faster $\Theta\left(1+\frac{1}{1-\alpha}\right)$

Linear Probing

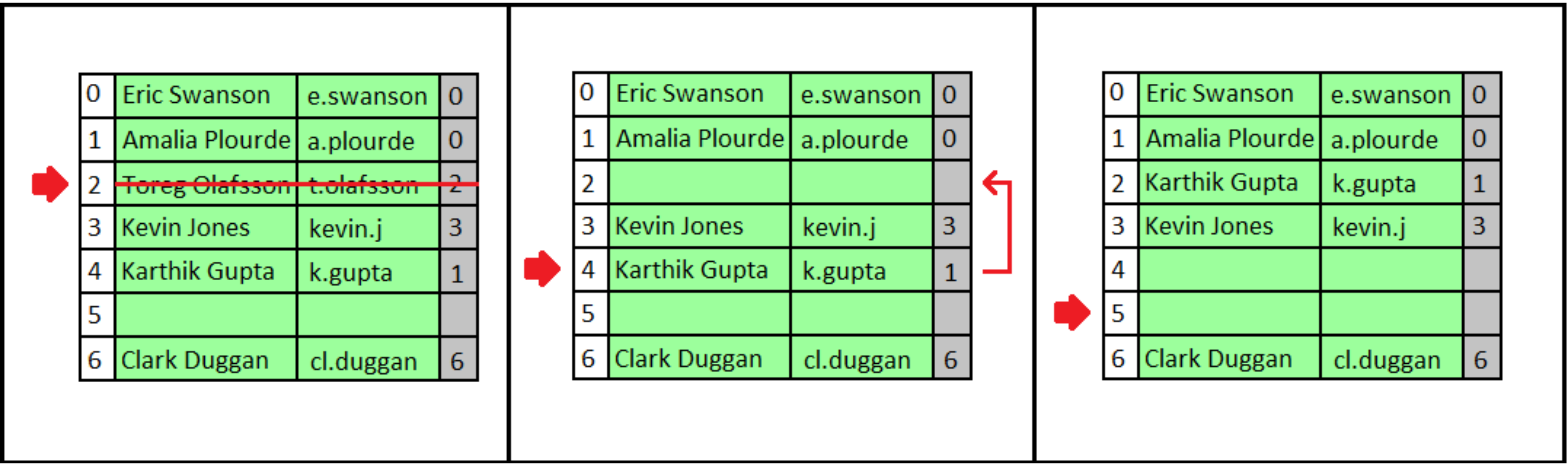
Deletion:

- How does this work?

Linear Probing

Deletion:

- How does this work?
- Cannot simply clear bucket
  - searches for items that previously moved past will fail.



0	Eric Swanson	e.swanson	0
1	Amalia Plourde	a.plourde	0
2	~~Toreg Olafsson~~	~~t.olafsson~~	~~2~~
3	Kevin Jones	kevin.j	3
4	Karthik Gupta	k.gupta	1
5			
6	Clark Duggan	cl.duggan	6

0	Eric Swanson	e.swanson	0
1	Amalia Plourde	a.plourde	0
2			
3	Kevin Jones	kevin.j	3
4	Karthik Gupta	k.gupta	1
5			
6	Clark Duggan	cl.duggan	6

0	Eric Swanson	e.swanson	0
1	Amalia Plourde	a.plourde	0
2	Karthik Gupta	k.gupta	1
3	Kevin Jones	kevin.j	3
4			
5			
6	Clark Duggan	cl.duggan	6

  - And moving one item might trigger more moves…

Image source: https://en.wikipedia.org/wiki/Linear_probing

Linear Probing

Deletion:

- How does this work?
- Cannot simply clear bucket
  - searches for items that previously moved past will fail.
- Tombstones
  - Mark buckets as having been previously used but currently empty.
  - Search continues past tombstones.
  - Later inserts can replace tombstones.

Linear Probing

Deletion:

- How does this work?
- Cannot simply clear bucket
  - searches for items that previously moved past will fail.
- Tombstones
  - Mark buckets as having been previously used but currently empty.
  - Search continues past tombstones.
  - Later inserts can replace tombstones.
- If this sounds like accumulating inefficiency, that is correct.
  - Rebuild hash table periodically?
  - Amortized cost is constant, but leads to pauses or complications interleaving.

Linear Probing

Deletion:

- How does this work?
- Cannot simply clear bucket
  - searches for items that previously moved past will fail.
- Tombstones
  - Mark buckets as having been previously used but currently empty.
  - Search continues past tombstones.
  - Later inserts can replace tombstones.
- If this sounds like accumulating inefficiency, that is correct.
  - Rebuild hash table periodically?
  - Amortized cost is constant, but leads to pauses or complications interleaving.

- This is the standard deletion handling for open addressing.

Quadratic Probing

Insertion

- If collision on $h(x)$, try $h(x) + 1^2,\ h(x) + 2^2,\ h(x) + 3^2,\ \ldots$ until an empty bucket is found.

Quadratic Probing

Insertion

- If collision on $h(x)$, try $h(x) + 1^2,\ h(x) + 2^2,\ h(x) + 3^2,\ \ldots$ until an empty bucket is found.
- Avoid primary clustering problem of linear probing.

Quadratic Probing

Insertion

- If collision on $h(x)$, try $h(x) + 1^2,\ h(x) + 2^2,\ h(x) + 3^2,\ \ldots$ until an empty bucket is found.
- Avoid primary clustering problem of linear probing.
- If alternating signs used, this even gives a permutation through all buckets.

Quadratic Probing

Insertion

- If collision on $h(x)$, try $h(x) + 1^2,\ h(x) + 2^2,\ h(x) + 3^2,\ \ldots$ until an empty bucket is found.
- Avoid primary clustering problem of linear probing.
- If alternating signs used, this even gives a permutation through all buckets.
- Memory locality not as good as linear probing because of increasing gaps.

Double Hashing

Insertion

- If collision on $h(x)$, try $h(x) + h_2(x),\ h(x) + 2h_2(x),\ h(x) + 3h_2(x),\ \ldots$ until an empty bucket is found.
- Avoids primary clustering problem with one extra hash function evaluation.
- Both searches and insertions take

$\Theta\left(1 + \dfrac{1}{1-\alpha}\right)$ expected time

Load Factors

Linear probing performs a lot worse as load factor increases.

Successful search: $\Theta\left(1 + \frac{1}{1-\alpha}\right)$

Insertion or failed search: $\Theta\left(1 + \frac{1}{(1-\alpha)^2}\right)$

$\alpha = 0.9$ looks really expensive. Usual advice is to keep $\alpha \leq 2\ldots$

Resizing Hash Tables

Usual answer to high load factors is resizing the hash table.

- Works if not too memory constrained.
- And can wait for rebuild.
- Or can handle memory overhead of building a second copy in parallel.
- Better if you can get the size right the first time.

Cuckoo Hashing

Rough idea:

- Item can be inserted at $h_1(x)$ or $h_2(x)$.

Cuckoo Hashing

Rough idea:

- Item can be inserted at $h_1(x)$ or $h_2(x)$.
- If both are full, pick either location and push out previous occupant.
  - Previous occupant moves to its other location.
  - Possibly pushes out previous occupant again…

Cuckoo Hashing

Rough idea:

- Item can be inserted at $h_1(x)$ or $h_2(x)$.
- If both are full, pick either location and push out previous occupant.
  - Previous occupant moves to its other location.
  - Possibly pushes out previous occupant again…
- How fast is search?

Cuckoo Hashing

Rough idea:

- Item can be inserted at $h_1(x)$ or $h_2(x)$.
- If both are full, pick either location and push out previous occupant.
  - Previous occupant moves to its other location.
  - Possibly pushes out previous occupant again…
- How fast is search?
  - $O(1)$ worst case
- How fast is insertion?

Cuckoo Hashing

Rough idea:

- Item can be inserted at $h_1(x)$ or $h_2(x)$.
- If both are full, pick either location and push out previous occupant.
  - Previous occupant moves to its other location.
  - Possibly pushes out previous occupant again…
- How fast is search?
  - $O(1)$ worst case
- How fast is insertion?
  - $O(1)$ amortized $\leftarrow$ what is this word that I keep using?
  - Assumes rebuilds to maintain load factor
  - Analysis is complicated, and usually a side structure for edge cases (cycles)
- Insertion speed drops if load factor allowed to grow past $1/2$…