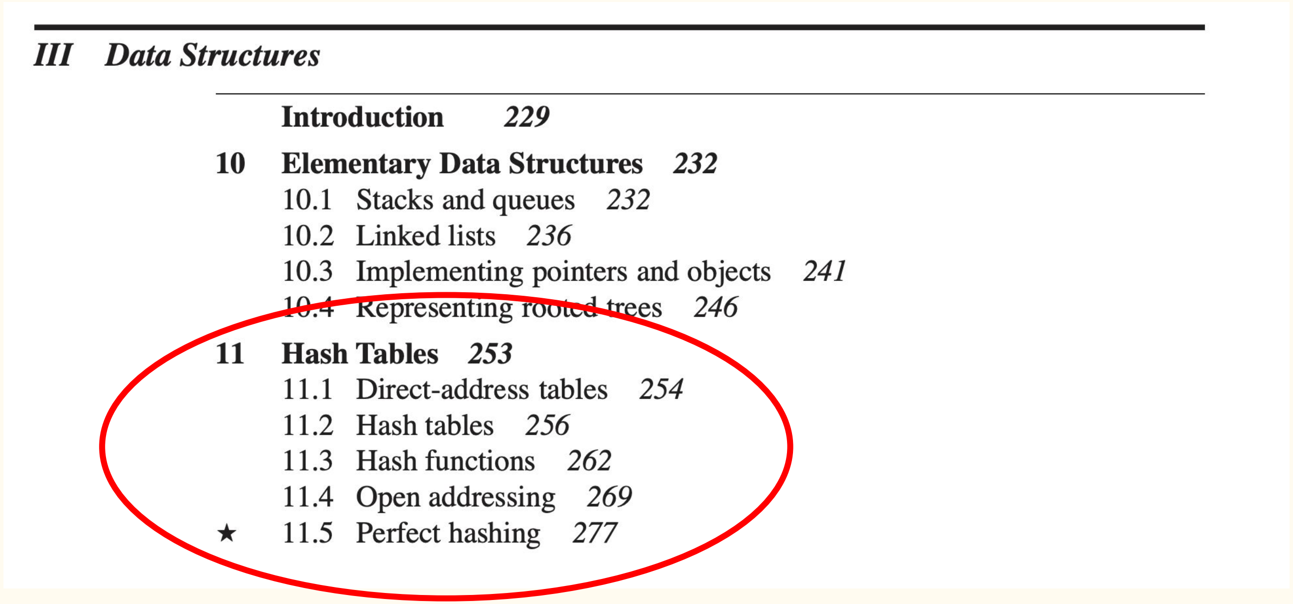
Data Structures and Algorithms

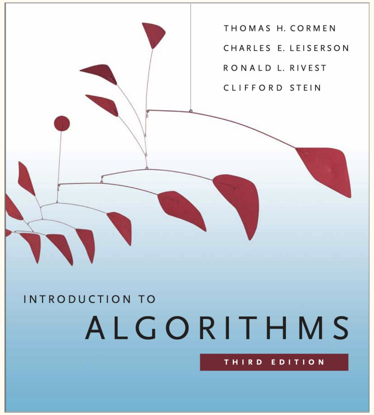
Tutorial 3. Analysis of runtime in hashing

Today's topic is covered in detail in

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. **Introduction to Algorithms.** The MIT Press 2009.



III Data Structures



Objectives

- Recap: Hashing & Hashtables
- Separate chaining & its analysis
- Open addressing & its analysis

Recap: Hashtables

1. Implementation of a Map/Dictionary ADT:
    - **`get`** value by key
    - **`insert`** a key-value pair
    - **`delete`** a key with an associated value

Recap: Hashtables

1. Implementation of a Map/Dictionary ADT:
    - **`get`** value by key
    - **`insert`** a key-value pair
    - **`delete`** a key with an associated value

2. Uses array to store key-value pairs
    - Directly (using open addressing) or
    - Storing a list of key-value pairs (separate chaining)

Recap: Hashtables

1. Implementation of a Map/Dictionary ADT:
    - **`get`** value by key
    - **`insert`** a key-value pair
    - **`delete`** a key with an associated value

2. Uses array to store key-value pairs
    - Directly (using open addressing) or
    - Storing a list of key-value pairs (separate chaining)

3. Relies on a **hash function** to convert key to array index

Recap: Ideal* hash function

1. Avoid collisions entirely

* Ideal for general purpose HashTable implementation. Not necessarily for other purposes, such as cryptography, document integrity checks, unique identifier generators, etc.

Recap: Ideal* hash function

1. Each key is equally likely to hash to any of the slots, independently of where any other key has hashed to (simple uniform distribution)

* Ideal for general purpose HashTable implementation. Not necessarily for other purposes, such as cryptography, document integrity checks, unique identifier generators, etc.

Recap: Ideal* hash function

1. Each key is equally likely to hash to any of the slots, independently of where any other key has hashed to (simple uniform distribution)
2. Same keys hash to same slots (determinism)

* Ideal for general purpose HashTable implementation. Not necessarily for other purposes, such as cryptography, document integrity checks, unique identifier generators, etc.

Recap: Ideal* hash function

1. Each key is equally likely to hash to any of the slots, independently of where any other key has hashed to (simple uniform distribution)
2. Same keys hash to same slots (determinism)
3. Works for arbitrary data types

* Ideal for general purpose HashTable implementation. Not necessarily for other purposes, such as cryptography, document integrity checks, unique identifier generators, etc.

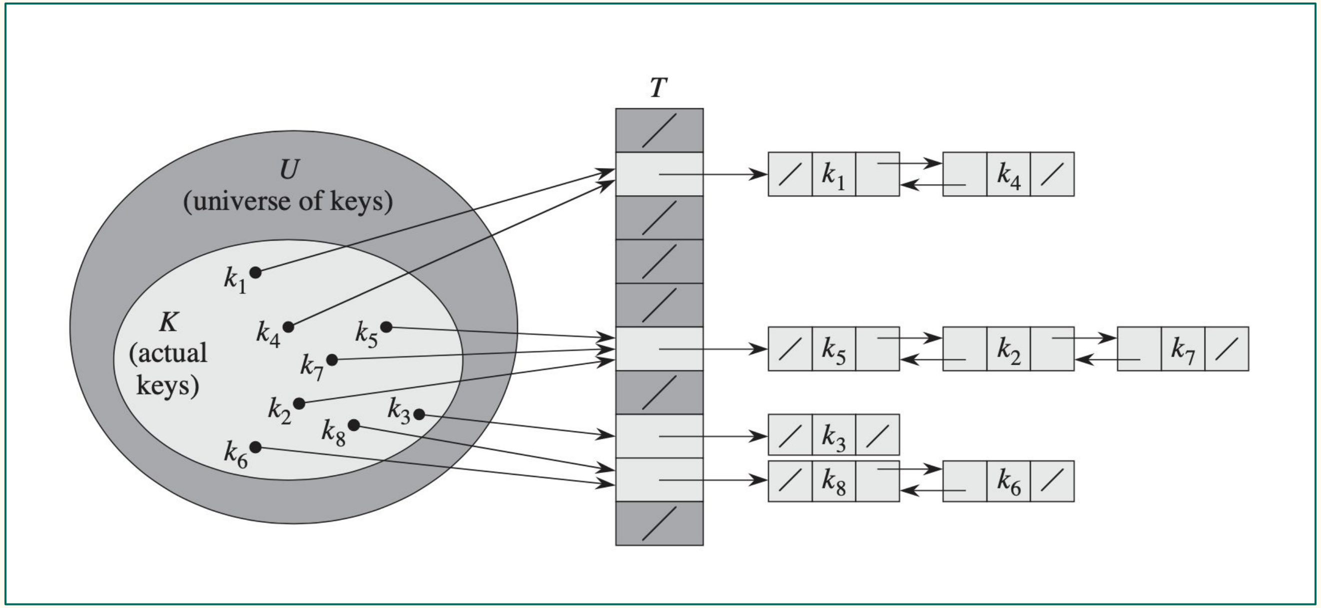
Recap: Ideal* hash function

1. Each key is equally likely to hash to any of the slots, independently of where any other key has hashed to (simple uniform distribution)
2. Same keys hash to same slots (determinism)
3. Works for arbitrary data types
4. [Cryptography] Tiny change to input changes output significantly (avalanche effect)

* Ideal for general purpose HashTable implementation. Not necessarily for other purposes, such as cryptography, document integrity checks, unique identifier generators, etc.

Separate chaining

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$



Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$

- To **`get(k)`** we search for an element with key **k** in the list **T[h(k)]**

Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$
- To **`get(k)`** we search for an element with key **k** in the list **T[h(k)]**
- To **`insert(k, v)`** we add **(k, v)** pair at the head of the list **T[h(k)]**

Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that h(k_i) = j
- To **`get(k)`** we search for an element with key **k** in the list **T[h(k)]**
- To **`insert(k, v)`** we add **(k, v)** pair at the head of the list **T[h(k)]**
- To **`delete(k)`** we delete pair with key **k** from the list **T[h(k)]**

Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$

- To `get(k)` we search for an element with key **k** in the list **T[h(k)]**
  O(length of T[h(k)]) — we will look closely in a moment

- To `insert(k, v)` we add **(k, v)** pair at the head of the list **T[h(k)]**

- To `delete(k)` we delete pair with key **k** from the list **T[h(k)]**

Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$
- To **`get(k)`** we search for an element with key **k** in the list **T[h(k)]**
  O(length of T[h(k)]) — we will look closely in a moment
- To **`insert(k, v)`** we add **(k, v)** pair at the head of the list **T[h(k)]**
  O(1)
- To **`delete(k)`** we delete pair with key **k** from the list **T[h(k)]**

Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$

- To **`get(k)`** we search for an element with key **k** in the list **T[h(k)]**
  O(length of T[h(k)]) — we will look closely in a moment

- To **`insert(k, v)`** we add **(k, v)** pair at the head of the list **T[h(k)]**
  O(1) or O(length of T[h(k)]) to also check if key is already present

- To **`delete(k)`** we delete pair with key **k** from the list **T[h(k)]**

Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$

- To `get(k)` we search for an element with key **k** in the list **T[h(k)]**
  O(length of T[h(k)]) — we will look closely in a moment

- To `insert(k, v)` we add **(k, v)** pair at the head of the list **T[h(k)]**
  O(1) or O(length of T[h(k)]) to also check if key is already present

- To `delete(k)` we delete pair with key **k** from the list **T[h(k)]**
  O(length of T[h(k)]) if we scan the list

Separate chaining: operations

- T[j] contains a list of key-value pairs (k_i, v_i) such that $h(k_i) = j$

- To **`get(k)`** we search for an element with key **k** in the list **T[h(k)]**
  O(length of T[h(k)]) — we will look closely in a moment

- To **`insert(k, v)`** we add **(k, v)** pair at the head of the list **T[h(k)]**
  O(1) or O(length of T[h(k)]) to also check if key is already present

- To **`delete(item)`** we delete pair **item** from the list **T[h(k)]**
  O(1) if we use doubly linked lists (and delete by reference to the node)

Exercise: Separate Chaining

Exercise 3.1. Demonstrate what happens when we insert the keys

$$5;\ 28;\ 19;\ 15;\ 20;\ 33;\ 12;\ 17;\ 10$$

into a hash table with collisions resolved by chaining.
Let the table have 9 slots, and let the hash function be $\mathbf{h(k) = k \bmod 9}$.

Analysis of separate chaining: assumptions

- Hash table T has **m** slots, and stores **n** key-value pairs

Analysis of separate chaining: assumptions

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** of T is defined as **n / m**.

Analysis of separate chaining: assumptions

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** of T is defined as **n / m**.
- Worst case for chaining — all keys hash to the same slot — O(n).

Analysis of separate chaining: assumptions

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** of T is defined as **n / m**.
- Worst case for chaining — all keys hash to the same slot — O(n).
- So instead we consider **average case!**

Analysis of separate chaining: assumptions

- Hash table T has **m** slots, and stores **n** key-value pairs

- Load factor **α** of T is defined as **n / m**.

- Worst case for chaining — all keys hash to the same slot — O(n).

- So instead we consider **average case**!

- We assume **simple uniform hashing**:
  «Any given element is equally likely to hash into any of the m slots.»

Analysis of separate chaining: assumptions

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** of T is defined as **n / m**.
- Worst case for chaining — all keys hash to the same slot — O(n).
- So instead we consider **average case!**
- We assume **simple uniform hashing**:
  «Any given element is equally likely to hash into any of the m slots.»
- We also assume it suffices to compute the hash value in O(1) time.

Analysis of separate chaining (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** of T is defined as **n / m**.

Theorem. With separate chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.

Analysis of separate chaining (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** of T is defined as **n / m**.

Theorem. With separate chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.

Proof. Under simple uniform hashing, any key **k** not already stored in the table is equally likely to hash to any of the **m** slots. The expected time to search unsuccessfully for a key **k** is the expected length of **T[h(k)]**, which is exactly **α**.
Thus, the total time required (including time to compute h(k)) is $\Theta(1+\alpha)$.

Analysis of separate chaining (successful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** of T is defined as **n / m**.

Theorem. With separate chaining, a successful search takes average-case time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.

Proof. We assume that the key being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined is equal to the position of the element in the corresponding chain. Intuitively, the expected position is approximately $\alpha/2$, so overall we get $\Theta(1+\alpha)$.
See precise proof in Theorem 11.2 of Cormen et al.

Exercise: Analysis of Separate Chaining

Exercise 3.2. Consider keeping each chain in the hashtable as a sorted list. How would that affect running time for (un)successful searches, inserts, and deletes?

Attendance

https://baam.duckdns.org

Open addressing

- All elements (key-value pairs) occupy the hash table itself
- Corollary: load factor **α** cannot be more that 1.
- To insert we successively **probe** the hash table,
  until we find an empty slot
- The **probe sequence** depends on the key
- Essentially, we now have a hash function with two arguments:
  h(k, 0), h(k, 1), h(k, 2), ...
- In our analysis we assume uniform hashing:
  «The probe sequence of each key is equally likely to be
  any of the **m!** permutations of $\boldsymbol{<0, 1, \ldots, m-1>}$.»

Open addressing: `insert`

1. Let $\mathbf{i = 0}$

2. Try index $\mathbf{h(k, i)}$
    a. If $\mathbf{T[h(k, i)]}$ is empty, use it
    b. Otherwise, increment $\mathbf{i}$ and repeat until $\mathbf{i \geqslant m}$

3. If there was no empty slot, then hash table is full

Open addressing: `search`

- Let **i = 0**

- Try index **h(k, i)**
  - If **T[h(k, i)]** has key **k**, then return associated value and exit;
  - If **T[h(k, i)]** is empty, then exit: there is no key **k** in **T**
  - Otherwise, increment **i** and repeat until $\mathbf{i} \geqslant \mathbf{m}$

- At this point we know that hashtable is full and there is no key **k** in **T**

Open addressing: `delete`

- Very difficult (if we want it efficient).

Open addressing: `delete`

- Very difficult (if we want it efficient).
- **Why difficult?**

Open addressing: `delete`

- Very difficult (if we want it efficient).

- **Why difficult?**
  If we remove (mark as EMPTY) an element in the middle of a probing sequence, then some elements will become invisible for search!

Open addressing: `delete`

- Very difficult (if we want it efficient).

- **Why difficult?**
  If we remove (mark as EMPTY) an element in the middle of a probing sequence, then some elements will become invisible for search! And if we introduce DELETED mark, then running time of search/insert no longer depends on the load factor.

Open addressing: `delete`

- Very difficult (if we want it efficient).

- **Why difficult?**
  If we remove (mark as EMPTY) an element in the middle of a probing sequence, then some elements will become invisible for search! And if we introduce DELETED mark, then running time of search/insert no longer depends on the load factor.

- So typically, if many deletes are expected, separate chaining is used.

Open addressing: strategies

- **Linear probing**

  $h(k, i) = (h'(k) + i) \bmod m$

  $h(k),\ h(k)+1,\ h(k)+2,\ h(k)+3,\ h(k)+4,\ h(k)+5, \ldots$

Open addressing: strategies

- **Linear probing**

  Primary clustering

  $h(k, i) = (h'(k) + i) \bmod m$

  $h(k),\ h(k)+1,\ h(k)+2,\ h(k)+3,\ h(k)+4,\ h(k)+5, \ldots$

Open addressing: strategies

- **Linear probing**
  $h(k, i) = (h'(k) + i) \bmod m$
  $h(k),\ h(k)+1,\ h(k)+2,\ h(k)+3,\ h(k)+4,\ h(k)+5, \ldots$

Primary clustering

- **Quadratic probing**
  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  $h(k),\ h(k)+1,\ h(k)+4,\ h(k)+9,\ h(k)+25,\ h(k)+36, \ldots$

Open addressing: strategies

- **Linear probing**
  $h(k, i) = (h'(k) + i) \bmod m$
  $h(k),\ h(k)+1,\ h(k)+2,\ h(k)+3,\ h(k)+4,\ h(k)+5, \ldots$

Primary clustering

- **Quadratic probing**
  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  $h(k),\ h(k)+1,\ h(k)+4,\ h(k)+9,\ h(k)+25,\ h(k)+36, \ldots$

Secondary clustering

Open addressing: strategies

- **Linear probing**

  $h(k, i) = (h'(k) + i) \bmod m$

  $h(k),\ h(k)+1,\ h(k)+2,\ h(k)+3,\ h(k)+4,\ h(k)+5, \ldots$

Primary clustering

- **Quadratic probing**

  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

  $h(k),\ h(k)+1,\ h(k)+4,\ h(k)+9,\ h(k)+25,\ h(k)+36, \ldots$

Secondary clustering

- **Double hashing**

  $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

Open addressing: strategies

- **Linear probing**
  $h(k, i) = (h'(k) + i) \bmod m$
  $h(k),\ h(k)+1,\ h(k)+2,\ h(k)+3,\ h(k)+4,\ h(k)+5, \ldots$

Primary clustering

- **Quadratic probing**
  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  $h(k),\ h(k)+1,\ h(k)+4,\ h(k)+9,\ h(k)+25,\ h(k)+36, \ldots$

Secondary clustering

- **Double hashing**
  $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

Typically has nice random-like properties

Exercise: Open addressing

Exercise 3.4. Consider inserting the keys

$$10;\ 22;\ 31;\ 4;\ 15;\ 28;\ 17;\ 88;\ 59$$

into a hash table of length $\mathbf{m = 11}$ using open addressing with the auxiliary hash function $\mathbf{h'(k) = k}$. Illustrate the result of inserting these keys

1. using linear probing,
2. using quadratic probing with $\mathbf{c_1 = 1}$ and $\mathbf{c_2 = 3}$, and
3. using double hashing with $\mathbf{h_1(k) = k}$ and $\mathbf{h_2(k) = 1 + (k \bmod (m - 1))}$.

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} / \mathbf{m} < \mathbf{1}$.

Theorem. With open addressing, an unsuccessful search takes average-case time $O(1/(1-\alpha))$ under the assumption of simple uniform hashing.

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} / \mathbf{m} < \mathbf{1}$.

Theorem. With open addressing, an unsuccessful search takes average-case time $O(1/(1-\alpha))$ under the assumption of simple uniform hashing.

Proof. $\underset{\uparrow \text{ We need at least one probe}}{\#\text{ of probes} = 1 +}$

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} / \mathbf{m} < \mathbf{1}$.

Theorem. With open addressing, an unsuccessful search takes average-case time $O(1/(1-\alpha))$ under the assumption of simple uniform hashing.

Proof. $$\#\text{ of probes} = 1 + \underset{\uparrow}{\frac{n}{m}}\left(1 + \frac{n-1}{m-1}(\ldots)\right)$$

Probability of hitting an occupied slot for the 1st probe

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} \,/\, \mathbf{m} < \mathbf{1}$.

Theorem. With open addressing, an unsuccessful search takes average-case time O(1/(1–α)) under the assumption of simple uniform hashing.

Proof. $$\#\text{ of probes} = 1 + \frac{n}{m}\Big(1 + \underset{\Big\uparrow}{\frac{n-1}{m-1}}(\ldots)\Big)$$

Probability of hitting an occupied slot for the 2nd probe

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} / \mathbf{m} < \mathbf{1}$.

Theorem. With open addressing, an unsuccessful search takes average-case time O(1/(1–α)) under the assumption of simple uniform hashing.

Proof.
$$\begin{aligned}\#\text{ of probes} &= 1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}(\ldots)\right) \\ &\leq 1 + \alpha(1 + \alpha(\ldots))\end{aligned}$$

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} / \mathbf{m} < \mathbf{1}$.

Theorem. With open addressing, an unsuccessful search takes average-case time O(1/(1−α)) under the assumption of simple uniform hashing.

Proof.

$$\begin{aligned} \#\text{ of probes} &= 1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}(\ldots)\right) \\ &\leq 1 + \alpha(1 + \alpha(\ldots)) \\ &= 1 + \alpha + \alpha^2 + \ldots \end{aligned}$$

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α = n / m < 1**.

Theorem. With open addressing, an unsuccessful search takes average-case time O(1/(1–α)) under the assumption of simple uniform hashing.

Proof.

$$\begin{aligned}\#\text{ of probes} &= 1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}(\ldots)\right) \\ &\leq 1 + \alpha(1 + \alpha(\ldots)) \\ &= 1 + \alpha + \alpha^2 + \ldots \\ &= \sum_{k=1}^{\infty} \alpha^k\end{aligned}$$

Analysis of open addressing (unsuccessful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor **α** = **n / m < 1**.

Theorem. With open addressing, an unsuccessful search takes average-case time O(1/(1–α)) under the assumption of simple uniform hashing.

Proof.

$$\begin{aligned}\#\text{ of probes} &= 1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}(\dots)\right) \\ &\leq 1 + \alpha(1 + \alpha(\dots)) \\ &= 1 + \alpha + \alpha^2 + \dots \\ &= \sum_{k=1}^{\infty} \alpha^k = \frac{1}{1-\alpha}\end{aligned}$$

Analysis of open addressing (insertion)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} / \mathbf{m} < \mathbf{1}$.

Corollary. With open addressing, an insertion takes average-case time $O(1/(1-\alpha))$ under the assumption of simple uniform hashing.

Analysis of open addressing (successful search)

- Hash table T has **m** slots, and stores **n** key-value pairs
- Load factor $\boldsymbol{\alpha} = \mathbf{n} / \mathbf{m} < \mathbf{1}$.

Theorem. With open addressing, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \log \frac{1}{1-\alpha}$$

Proof. See Theorem 11.8 in Cormen et al.

Exercise: Analysis of open addressing

Exercise 3.4. Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is $3/4$ and when it is $7/8$.

Remark on open addressing

1. More difficult to implement (compared to separate chaining)

2. Typically requires more fine-tuning

3. Works best if you know the frequency and distribution of keys, as a somewhat specialized solution

There exist other, more complicated open addressing approaches, such as the Cuckoo Hashing, which introduce worst-case O(1) search and delete operations, and amortized O(1) insert.
However, we will not cover those in this course.

A note on terminology

- «Open Hashing» = «Separate chaining»
- «Closed Hashing» = «Open addressing»

Summary

- Hash tables & Hashing
- Separate chaining supports $\Theta(1)$ and/or $\Theta(1+\alpha)$ operations
- Open addressing supports $O(1/(1-\alpha))$ search

Summary

- Hash tables & Hashing
- Separate chaining supports $\Theta(1)$ and/or $\Theta(1+\alpha)$ operations
- Open addressing supports $O(1/(1-\alpha))$ search

See you next week!