Notes on Hash Tables

Geoffrey Matthews

May 15, 2018

Dictionary Operations

- INSERT
- SEARCH
- DELETE

Hash table implementation of Dictionary

- Expected search time: $O(1)$
- Worst case search: $O(n)$

Hash table is generalization of an ordinary array

- With array, the key k is the position k in the array.
- Given a key k, we find the element with key k by **direct addressing**.
- Direct addressing only applicable when we can afford to allocate an array with one position for every key.

Use hash table when we don't have one position for each key

- Number of keys stored is small relative to the number of possible keys.
- Hash table is an array with size proportional to the number of keys stored, not the number of possible keys.
- Given a key k, don't use k to index the array.
- Instead, compute a function of k and use that to index the array.
- This function is called a **hash function**.
- Have to solve issue of what to do when hash function maps multiple keys to same table entry.
    - chaining
    - open addressing

Direct-address tables

- Scenario:
    - Maintain a dynamic set
    - Each element has a key drawn from a universe
      $U = \{0, 1, \ldots, m-1\}$ where m isn't too large.
    - No two elements have the same key.
- Represent by a **direct-address table**, or array, $T[0 \ldots m-1]$:
    - Each *slot*, or position, corresponds to a key in U.
    - If there's an element x with key k, then $T[k]$ contains a pointer to x.
    - Otherwise, $T[k]$ is empty, represented by NIL.

Direct-address table

T

key   satellite data

U
(universe of keys)

K
(actual keys)
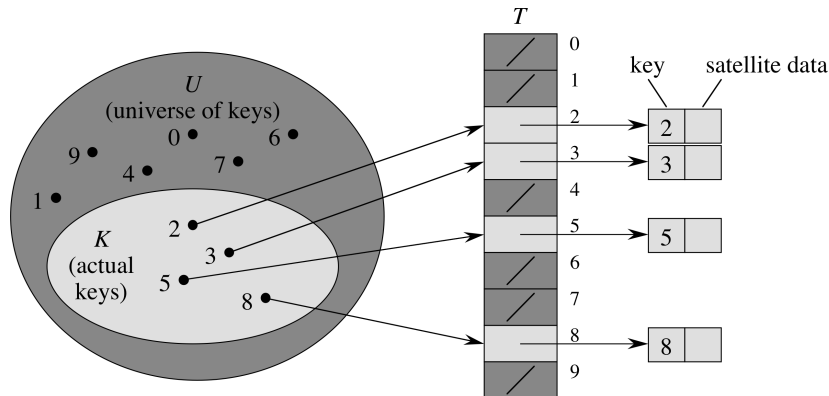
Direct-Address-Search(T,k)

1   **return** $T[k]$

Direct-Address-Insert(T,k)

1    $T[key[x]] = x$

Direct-Address-Delete(T,k)

1    $T[key[x]] = $ NIL

All operations $O(1)$.

Hash tables

- If U is large, storing a table of size $|U|$ is impractical.
- Often the set K of keys actually used is small compared to U.
  - Most of the space in a direct-access table is wasted.
- When K is much smaller than U, a hash table requires much less space than a direct-address table.
- Can reduce storage requirements to $\Theta(|K|)$
- Can still get $O(1)$ search time on *average*, but not *worst* case.

Hash table idea

- Instead of storing an element with key k in slot k, use a function h and store the element in slot $h(k)$.
- h is called a **hash function**
- $h : U \rightarrow \{0, 1, \ldots, m-1\}$
- $m \ll |U|$
- $h(k)$ is a legal slot number in T
- We say k *hashes* to $h(k)$

Collisions
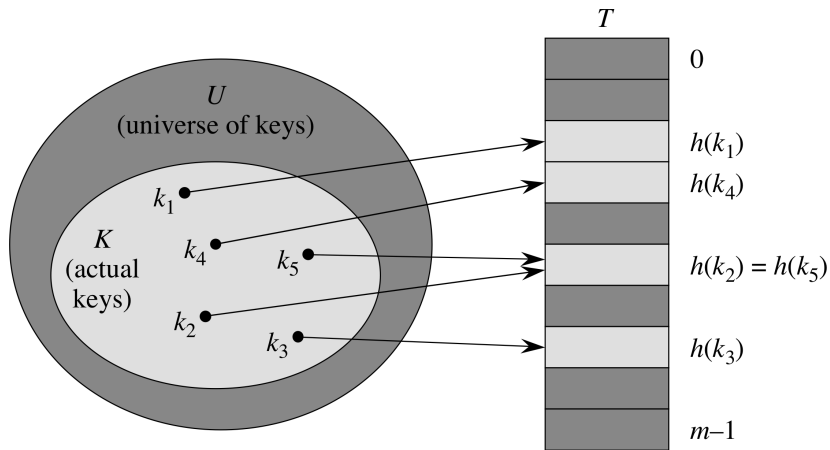
Collisions

- When two or more keys hash to the same slot.
- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen.
- Definitely happens when $|K| > m$.
- Must be prepared to handle collisions in all cases.
- Two methods:
    - chaining
    - open addressing
- Chaining is usually better.

Collision resolution by chaining

- Put all elements that hash to the same slot into a linked list.
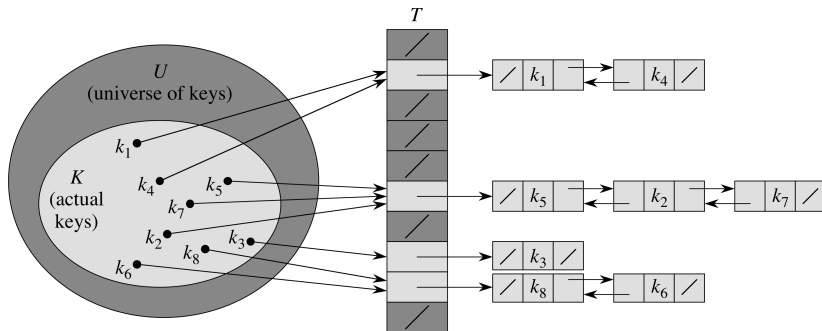
- Doubly linked list allows easy deletion.
- Note that elements may include keys, linked-list pointers, and other satellite data.

Implementation of hash table with chaining

Chained-Hash-Insert(T, x)

1   insert x at the head of list $T[h(key[x])]$

- Worst case $O(1)$
- Assumes element inserted not already in list.
- Would take an additional search to see if it was already inserted.

Implementation of hash table with chaining

Chained-Hash-Search(T, k)

1   search for element with key k in list $T[h(k)]$

- ▶ Running time proportional to length of list in slot $h(k)$

Implementation of hash table with chaining

Chained-Hash-Delete(T, x)

1   delete x from the list $T[h(key[x])]$

- ▶ Given pointer x to the element to delete, so no search is needed to find this element.
- ▶ Worst case $O(1)$ if lists are doubly linked.
- ▶ If lists are singly linked, deletion takes as long as search, because we must find x's predecessor.

Analysis of hashing with chaining

- Given a key, how long does it take to find an element with that key, or determine that there is no element with that key?
- Analysis is in terms of the **load factor** $\alpha = n/m$
- $n = \#$ elements in the table
- $m = \#$ slots in the table
- Load factor is average number of elements per linked list.
- Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$
- Worst case is when all n keys hash to the same slot:
  - a single list of length n
  - worst case is $\Theta(n)$ plus time to compute h
- Average case depends on how well the hash function distributes keys among slots.

Average-case analysis of hashing with chaining

- Assume **simple uniform hashing**: any given element is equally likely to hash to any of the m slots.
- For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j.
- $n = n_0 + n_1 + \cdots + n_{m-1}$
- Average value of n_j is $E[n_j] = \alpha = n/m$
- Assume we can compute h in $O(1)$ time, so that the time required to search for k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.
- Two cases:
  - Unsuccessful search: hash table has no element with key k
  - Successful search: hash table contains an element with key k

Unsuccessful search

Theorem

An unsuccessful search takes expected time $\Theta(1 + \alpha)$.

Proof

- Simple uniform hashing means any key not already in the table is equally likely to hash to any of the m slots.
- To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$.
- This list has expected length α.
- Adding the time to compute the hash function gives $\Theta(1 + \alpha)$.

Successful search

- The expected time for a successful search is also $\Theta(1 + \alpha)$.
- The probability that each list is searched is proportional to the length of the list.

Successful search

Theorem

A successful search takes expected time $\Theta(1 + \alpha)$.

Proof

- Assume the element x is equally likely to be any of the n elements stored in the table.

- The number examined during the search for x is 1 more than the number of elements that appear before x in x's list.

- These are the elements inserted *after* x was inserted.

- We need to find the average, over n elements, of how many elements were inserted into x's list after x was inserted.

- Let x_i be the ith element inserted, and let $k_i = key[x_i]$.

- For all i and j, let $X_{ij} = I\{h(k_i) = h(k_j)\}$

- Simple uniform hashing means

$$Pr\{h(k_k) = h(k_j)\} = 1/m = E[X_{ij}]$$

Expected number of elements examined, successful search

$$
\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}\sum_{i=0}^{n-1}i \\
&= 1 + \frac{1}{nm}\frac{n(n-1)}{2} \\
&= 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1+\alpha)
\end{aligned}
$$

Alternative analysis

- $X_{ij\ell} = I\{$the search is for x_i, $h(k_i) = h(k_j) = \ell\}$
- Simple uniform hashing means
  $Pr\{h(k_i) = \ell\} = Pr\{h(k_j) = \ell\} = 1/m$
- $Pr\{$the search is for $x_i\} = 1/n$
- All these are independent: $Pr\{X_{ij\ell} = 1\} = E[X_{ij\ell}] = 1/nm^2$

$$Y_j = I\{x_j \text{ appears in a list prior to the } x_i\}$$
$$= \sum_{i=1}^{j-1} \sum_{\ell=0}^{m-1} X_{ij\ell}$$

Alternative analysis, continued

$$
\begin{aligned}
E\left[1 + \sum_{j=1}^{n} Y_j\right] &= 1 + E\left[\sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{\ell=0}^{m-1} X_{ij\ell}\right] \\
&= 1 + \sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{\ell=0}^{m-1} E[X_{ij\ell}] \\
&= 1 + \sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{\ell=0}^{m-1} \frac{1}{nm^2} \\
&= 1 + \binom{n}{2} \cdot m \cdot \frac{1}{nm^2} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1 + \alpha)
\end{aligned}
$$

Interpretation

- If $n = O(m)$ then $\alpha = n/m = O(1)$, which means searching takes constant time on average.
- Since insertion and deletion take $O(1)$ worst case time, all dictionary operations take average time $O(1)$.

Hash functions

- Ideally, satisfies the assumption of simple uniform hashing.
- In practice, impossible since we don't know the distribution of input keys.
- Often use heuristics, based on the domain of the keys, to create hash functions that work well.

Keys as natural numbers

- Hash functions usually assume keys are natural numbers.
- Can interpret any computer data as natural number.
- Interpret as radix 2^p number.
- Strings, for example: CLRS
  - ASCII: 67, 76, 82, 83
  - There are 128 ASCII values, use radix 2^7:
  - $h(\mathrm{CLRS}) = 67(128^3) + 76(128^2) + 82(128^1) + 83(128^0) = 141,764,947$

Division method for hash functions

$$h(k) = k \bmod m$$

- Example: $m = 20$ and $k = 91 \Rightarrow h(k) = 11$
- Fast: requires only one division.
- Bad ideas:
    - Powers of 2 are bad values for m:
      just uses least significant bits.
    - If k is a character string interpreted as radix 2^p number, then
      $m = 2^p - 1$ is bad: permuting characters does not change hash
      value.
- Good choice for m:
  Prime number not too close to a power of 2.

Division method example

- Store $n \approx 2000$ character strings.
- Don't mind searching 3 strings per unsuccessful search.
- Choose $m = 701$.
- This is a prime near $2000/3$ but not near any power of 2.

Multiplication method for hash functions

1. Choose $0 < A < 1$
2. Multiply k by A
3. Extract the fractional part
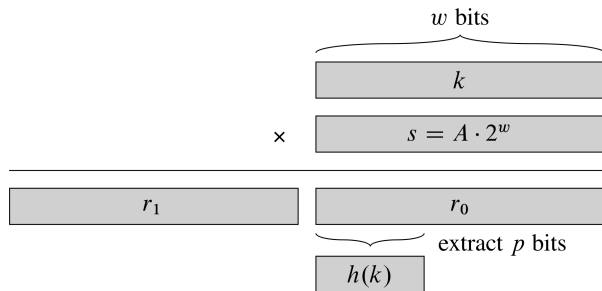4. Multiply by m
5. Take the floor.

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$
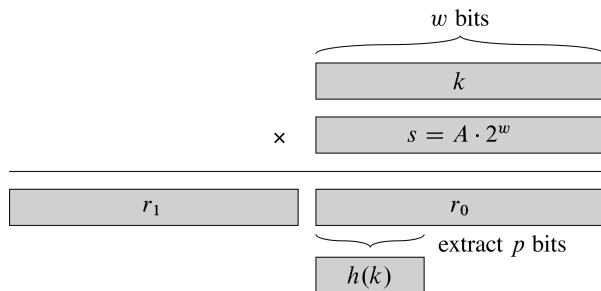
where

$$kA \quad \bmod 1 = kA - \lfloor kA \rfloor$$

gives the fractional part.

Easy implementation of: $h(k) = \lfloor m(kA \bmod 1) \rfloor$

- Choose $m = 2^p$.
- Let word size be w bits.
- Assume k fits in a single word.
- Let s be an integer in the range $0 < s < 2^w$.
- Let A be $s/2^w$.

# Easy implementation of: $h(k) = \lfloor m(kA \bmod 1) \rfloor$

- Multiply k by s.
- Result is $2w$ bits.
- We can ignore r_1, since r_0 is the fractional part.
- Multiplying by $m = 2^p$ just shifts r_0 left p places.
- Instead, just take the p most significant bits of r_0.

Example computation of: $h(k) = \lfloor m(kA \bmod 1) \rfloor$

- Choose $m = 2^3 = 8$, $w = 5$, $k = 21$.
- Choose $0 < s < 2^5$, s $= 13$, therefore $A = 13/32$.

- $kA = 21(13/32) = 273/32 = 8\frac{17}{32}$
- $kA \bmod 1 = 17/32$
- $m(kA \bmod 1) = 8(17/32) = 17/4 = 4\frac{1}{4}$
- $\lfloor m(kA \bmod 1) \rfloor = 4 = h(k)$

- $ks = 21(13) = 273 = 8(2^5) + 17$
- $r_1 = 8$, $r_0 = 17 = 10001_b$.
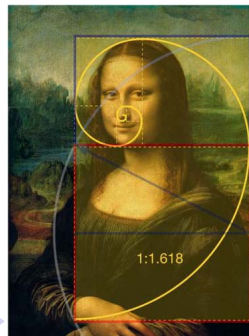- $p = 3$ most significant bits is $100_b = 4$.
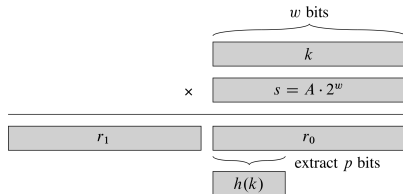
Multiplication method for hash functions

$$h(k) = \lfloor m((kA) \bmod 1) \rfloor$$

- ▶ Disadvantage: slower than division method.
- ▶ Advantage: value of m is not critical, 2^p a good choice.
- ▶ Knuth suggests a value for A:

$$\frac{\sqrt{5} - 1}{2} = 0.6180339887...$$

- ▶ Look up the Golden Ratio.

1:1.618

Universal hashing—randomized hashing

- For any hash function, the world *could* give us keys that all hash to the same spot. If the world was very, very mean.
- To randomize this, choose a hash function randomly from a collection of hash functions, each time the program starts.
- A collection of hash functions, $\mathcal{H}$ that map a universe U into keys $0 \leq k < m$ is called **universal** if for each pair, $k, \ell \in U$, $k \neq \ell$,

$$\Pr_{h \in \mathcal{H}}\{h(k) = h(\ell)\} \leq \frac{1}{m}$$

- In other words, the chance of a collision between k and ℓ is no more than $1/m$, when h is chosen at random from $\mathcal{H}$.
- Such collections of hash functions are easy to design.

Universal hashing expected chain lengths

Using chaining and universal hashing on key k:

- If k is not in the table,

$$E[n_{h(k)}] \leq \alpha$$

- If k is in the table,

$$E[n_{h(k)}] \leq 1 + \alpha$$

- So the expected time for SEARCH is $O(1)$.

Open addressing

- Instead of chaining, store all keys in hash table.
- Must have $\alpha \leq 1$.
- Use $h(key) + i \mod m, i = 0, 1, 2, ...m - 1$
- Example: $h(n) = n \mod 10$

Index:	0	1	2	3	4	5	6	7	8	9
Insert 12:	x	x	12	x	x	x	x	x	x	x
Insert 14:	x	x	12	x	14	x	x	x	x	x
Insert 32:	x	x	12	32	14	x	x	x	x	x
Insert 92:	x	x	12	32	14	92	x	x	x	x
Insert 53:	x	x	12	32	14	92	53	x	x	x

- Linear probing like this leads to **clustering**.

Open addressing

- More generally, use a hash function that takes both a key and a position and returns a position:

$$h : U \times \{0, 1, ..., m - 1\} \to \{0, 1, ..., m - 1\}$$

- Then use probe sequence

$$h(k, 0), h(k, 1), ..., h(k, m - 1)$$

- We require that for every key, k, the probe sequence be a permutation of

$$(0, 1, ..., m - 1)$$

so that all possible probes are examined in m probes.

- Linear probing satisfies this requirement, but has bad clustering.

Hash insertion and search, open addressing

HASH-INSERT(T, k)

  $i = 0$
  repeat
     $j = h(k, i)$
     if $T[j]$ == NIL
       $T[j] = k$
       return j
     else $i = i + 1$
  until i == m
  error "hash table overflow"

HASH-SEARCH(T, k)

  $i = 0$
  repeat
     $j = h(k, i)$
     if $T[j]$ == k
       return j
     $i = i + 1$
  until $T[j]$ == NIL **or** $i = m$
  return NIL

Hash deletion, open addressing

- We cannot simply replace a deleted element with NIL.
- This might make search halt prematurely if clustering has occurred.
- Instead we insert a special DELETED value.
- Insert will treat DELETED as available.
- Search will treat DELETED as full.
- Search time no longer depends on α alone.

Uniform hashing with open addressing

- In our analysis, we assume **uniform hashing**:
  - The probe sequence for a key k is equally likely to be any of the $m!$ possible sequences.
- Open addressing has to generalize the notion of uniform hashing to a function that generates an entire sequence of probes.
- True open uniform hashing is difficult.
- In practice approximations are used.

Linear probing

- Given an ordinary hash function $h' : U \rightarrow \{0, ..., m-1\}$, called an **auxiliary hash function**, use

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

- Only generates m of the $m!$ possible permutations of $\{0, ..., m-1\}$

- Extremely susceptible to **primary clustering**.

- Any slot preceded by i full slots gets filled with probability

$$\frac{i+1}{m}$$

and hence long chains get longer.

Linear probing

- Given an ordinary hash function $h' : U \to \{0, ..., m-1\}$, called an **auxiliary hash function**, use

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

- Only generates m of the $m!$ possible permutations of $\{0, ..., m-1\}$

- Extremely susceptible to **primary clustering**.

- Any slot preceded by i full slots gets filled with probability

$$\frac{i + 1}{m}$$

and hence long chains get longer.

- Note: it does not matter if we use

$$h(k, i) = (h'(k) + ai + b) \mod m$$

. Why not?

Quadratic probing

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

- c_1, c_2, m must be carefully selected.
- Primary clustering is eliminated.
- If two keys have the same initial probe, their entire sequence is the same.
- This is called **secondary clustering** and is not as serious.
- Only m of the $m!$ possible permutations of $\{0, ..., m-1\}$ are used.

Double hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$

- Needs two auxiliary hash functions, h_1, h_2.
- $h_2(k)$ must be relatively prime to m:
  - Let $m = 2^p$ and make $h_2(k)$ odd.
  - Let m be prime and make $h_2(k) < m$:

$$h_1(k) = k \mod m$$
$$h_2(k) = 1 + (k \mod (m - 1))$$

- $\Theta(m^2)$ probe sequences used, instead of $\Theta(m)$.
- Performance is very close to ideal uniform hashing.

Double hashing

$$h_1(k) = k \mod 13$$
$$h_2(k) = 1 + (k \mod 11)$$

$$h_1(14) = 1$$
$$h_2(14) = 4$$

Analysis of open-address hashing, unsuccessful search

Theorem 11.6

Assuming uniform hashing and an open-address hash table with load factor $\alpha = n/m < 1$ the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$.

Proof

$$X = \text{number of probes in unsuccessful search}$$

$$A_k = \{k\text{th probe is to an occupied slot}\}$$

$$\{X \geq i\} = \bigcap_{k=1}^{i-1} A_k$$

$$\Pr\{X \geq i\} = \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\}$$

$$\cdots$$

$$\cdot \Pr\{A_{i-1} | A_1 \cap A_2 \cap ... \cap A_{i-2}\}$$

Probability that i probes find occupied slot

$$X = \text{number of probes in unsuccessful search}$$

$$A_k = \{k\text{th probe is to an occupied slot}\}$$

$$\{X \geq i\} = \bigcap_{k=1}^{i-1} A_k$$

$$\Pr\{X \geq i\} = \Pr\{A_1\} \cdot \Pr\{A_2|A_1\} \cdot \Pr\{A_3|A_1 \cap A_2\}$$

$$\cdots$$

$$\cdot \Pr\{A_{i-1}|A_1 \cap A_2 \cap ... \cap A_{i-2}\}$$

$$= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}$$

$$\leq \left(\frac{n}{m}\right)^{i-1}$$

$$= \alpha^{i-1}$$

Expected number of probes in unsuccessful search

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$
$$\leq \sum_{i=1}^{\infty} \alpha^{i-1}$$
$$= \sum_{i=0}^{\infty} \alpha^{i}$$
$$= \frac{1}{1-\alpha}$$
$$= 1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \cdots$$

▶ The last expression gives us an intuitive picture.

Theorem 11.6

Assuming uniform hashing and an open-address hash table with load factor $\alpha = n/m < 1$ the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$.

- If $\alpha = 0.5$, then we expect less than 2 probes on average.
- If $\alpha = 0.9$, then we expect less than 10 probes on average.

Expected number of probes for HASH-INSERT

- An element is inserted after finding an open slot.
- This is the same procedure followed by an unsuccessful search.
- Therefore the expected number of probes is at most

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

Analysis of open address hashing, successful search

Theorem 11.8

Assuming uniform hashing and every key is equally likely, in an open-address hash table with load factor $\alpha < 1$ the expected number of probes in a successful search is at most

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

Proof

- A successful search does the same as when the k was inserted.
- If k was the ith key inserted, then $\alpha = i/m$ when inserted.
- Therefore there were at most $1/(1 - i/m) = m/(m - i)$ probes
- The average over all n keys is then

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i}$$

Averaging over the *n* keys in the table:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i}$$

$$= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} \frac{1}{k}$$

$$\leq \frac{1}{\alpha} \int_{m-n}^{m} (1/x)dx$$

$$= \frac{1}{\alpha} \ln \frac{m}{m-n}$$

$$= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Theorem 11.8

Assuming uniform hashing and every key is equally likely, in an open-address hash table with load factor $\alpha < 1$ the expected number of probes in a successful search is at most

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

- If $\alpha = 0.5$, then we expect less than 1.387 probes on average.
- If $\alpha = 0.9$, then we expect less than 2.559 probes on average.

Perfect hashing

- In some applications the keys are static:
  - Reserved words in a programming language.
  - File names on a write-once CD-rom.
- In this case we can guarantee **worst-case** $O(1)$.
- Use a double hashing scheme.
- Choose a good primary h from a universal hash, $\mathcal{H}$.
- For each slot j, choose a h_j into a secondary hash table, S_j.
- If size of S_j is proportional to n_j^2, we can find h_j with no collisions.
- If we choose h carefully, expected size of all secondary hash tables is still $O(n)$.

Perfect hashing
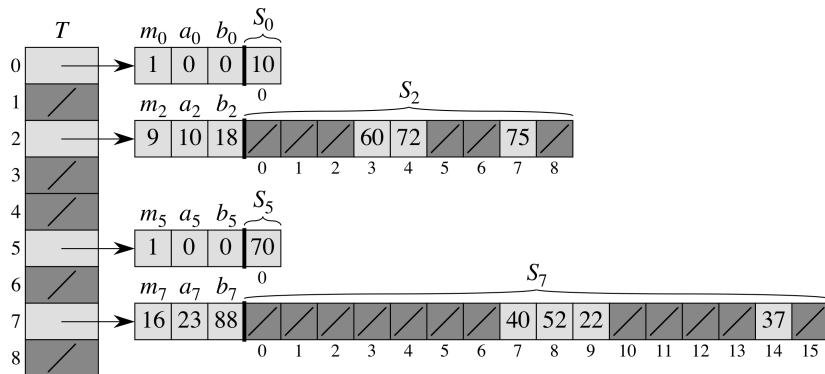
- m_i, a_i, b_i are parameters to choose a hash function from a set of universal hash functions:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m$$