Hash Tables

交通大學 資訊工程系

Introduction

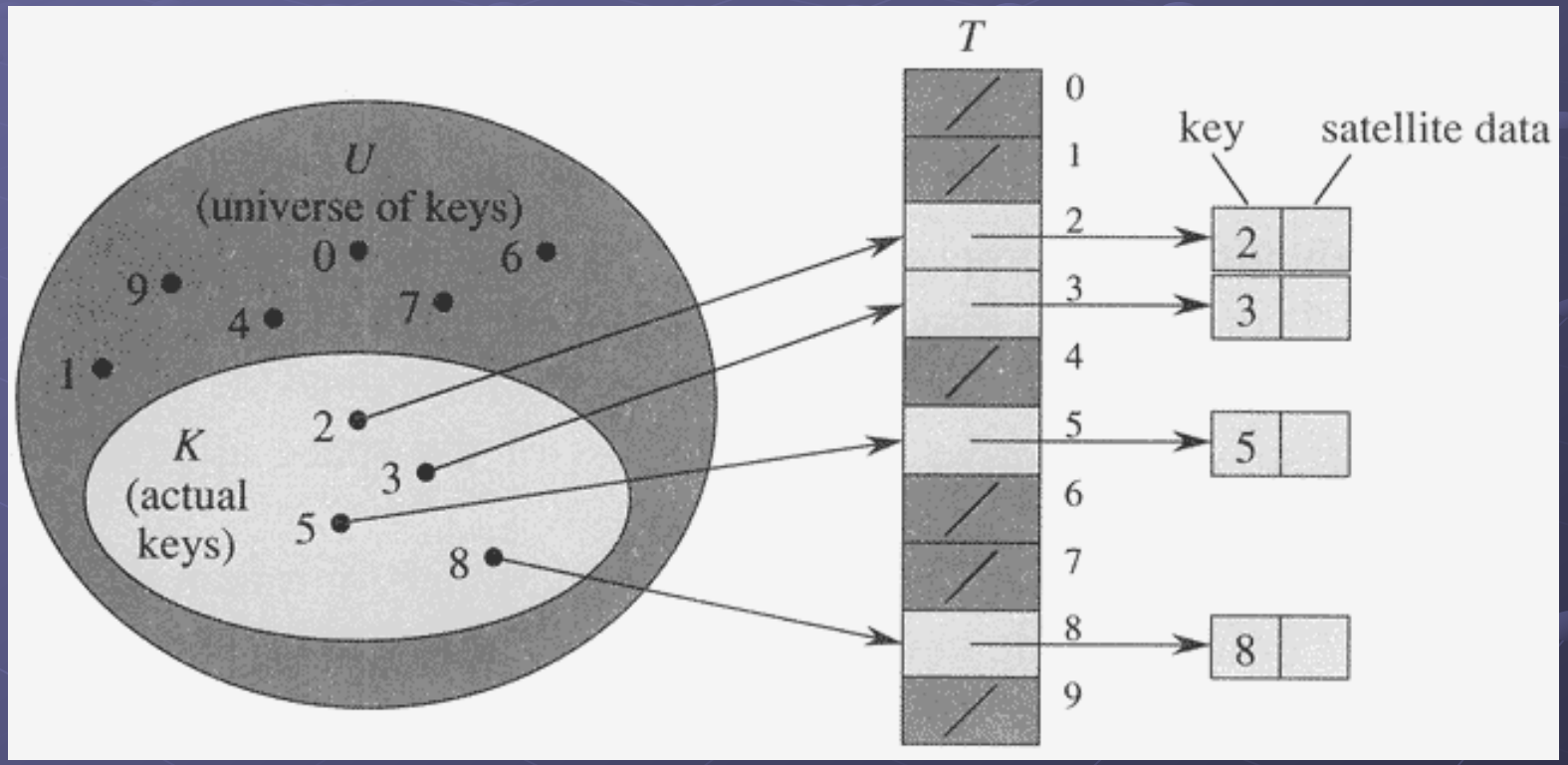
- **Hash tables**
  - **Many applications require a dynamic set that supports only the dictionary operations: INSERT, SEARCH, and DELETE.**
  - **For example: A compiler maintains a *symbol table*.**
  - **A hash table is an efficient data structure for implementing dictionaries.**
  - **Searching: $\Theta(n)$ time in the worst case; expected time is O(1).**
  - ***Direct addressing***
  - ***Chaining***
  - ***Open addressing***

Direct-Address Tables[1]

- **Assumption**
  - The universe U of keys is reasonably small, $U = \{0, 1, \ldots, m - 1\}$.
  - Use a linear array, ***direct-address table***, to represent the dynamic set, $T[0 \ldots m - 1]$.
- **Dictionary operations**

```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]
DIRECT-ADDRESS-INSERT(T, x)
    T[x.key] = x
DIRECT-ADDRESS-DELETE(T, x)
    T[x.key] = NIL
```

Direct-Address Tables[2]



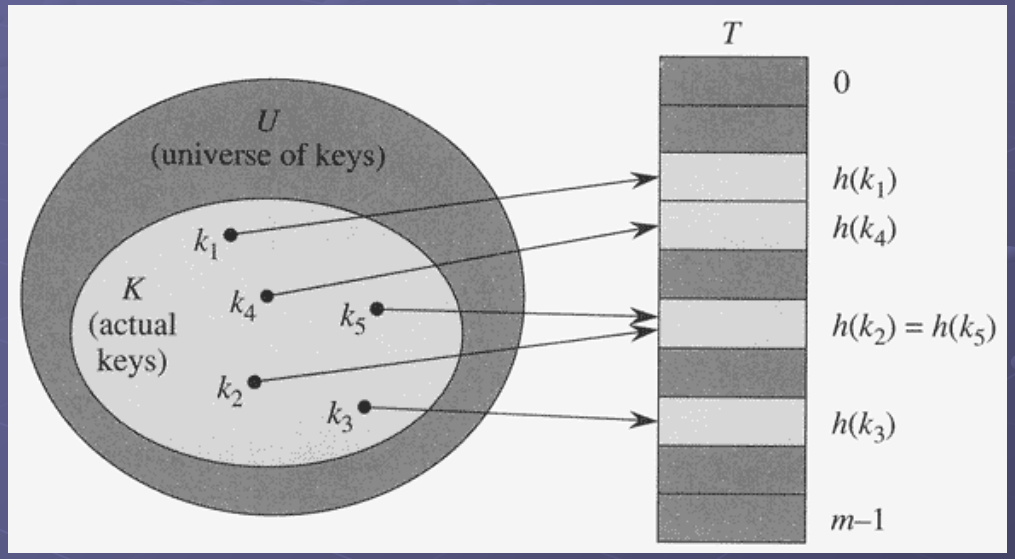
Hash Tables[1]

- **The difficulty of direct addressing:**
  - **If the universe U is large, storing a table T of size $|U|$ may be impractical , or even impossible.**
- ***Hash function h***
  - **An element with key k is stored in slot $h(k)$.**
  - **h maps the universe U of keys into the slots of a hash table $T[0 \,..\, m-1]$.**

    $$h : U \rightarrow \{0, 1, \ldots, m-1\}.$$

  - **The point of the hash function is to reduce the range of array indices that need to be handled.**
  - **h must be *deterministic*.**

Hash Tables[2]



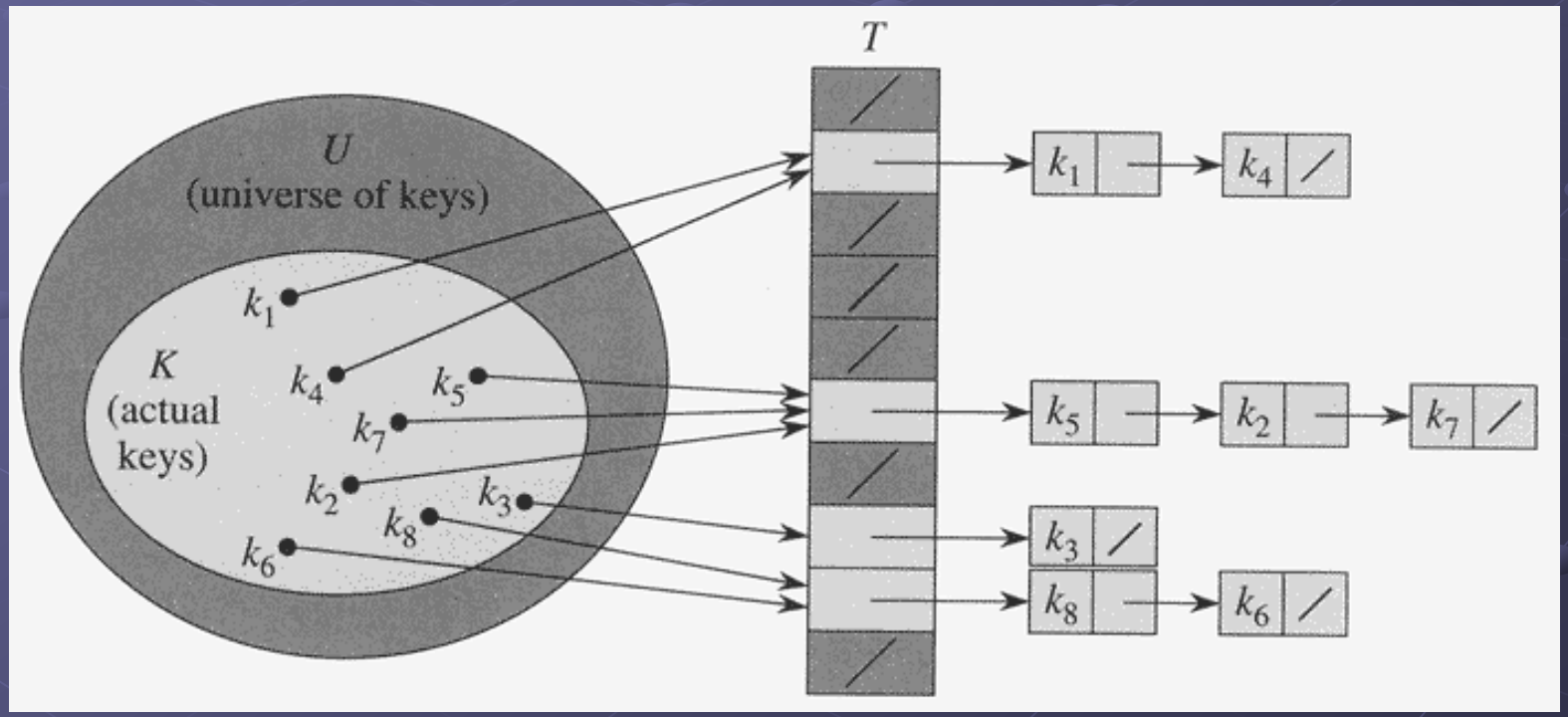
☞ *Collision*

- Two keys hash to the same slot.
- Ideal solution: Avoid collisions altogether.
- Choosing suitable hash function *h*: Make *h* appear to be "*random*".

Collision Resolution by Chaining[1]

☞ Chaining

- Put all the elements that hash to the same slot in a linked list.



Collision Resolution by Chaining[2]

- **Dictionary operations**

DIRECT-ADDRESS-INSERT (*T*, *x*)
insert *x* at the head of list *T*[*h*(*x.key*)]

DIRECT-ADDRESS- SEARCH (*T*, *k*)
search for an element with key *k* in list *T*[*h*(*k*)]

DIRECT-ADDRESS-DELETE(*T*, *x*)
delete x from the list *T*[*h*(*x.key*)]

Collision Resolution by Chaining[3]

- **The worst-case running time**
  - **Insertion: O(1).**
  - **Searching: Proportional to the length of the list.**
  - **Deletion: O(1) time if the lists are doubly linked.**
- **Analysis of hashing with chaining**
  - **Load factor α for T: n/m.**
    - **The average number of elements stored in a chain.**
  - **Worst-case behavior: all n keys hash to the same slot.**
- **Average performance of hashing: *simple uniform hashing*.**

Collision Resolution by Chaining[4]

☞ Simple uniform hashing

- Any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.

For $j = 0, 1, \ldots, m-1$, denote the length of $T[j]$ by n_j, so that

$$n = n_0 + n_1 + \ldots + n_{m-1}.$$

The average value of n_j is $E[n_j] = \alpha = n/m$.

Collision Resolution by Chaining[5]

- *Theorem 11.1*: In a hash table in which collision are resolved by chaining, an unsuccessful search takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.
- Proof:
  - The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$.
  - The expected number of elements examined in an unsuccessful search is α.
  - The total time required (including the time for computing $h(k)$) is $\Theta(1+\alpha)$.

Collision Resolution by Chaining[6]

- ***Theorem 11.2*:** If a hash table in which collision are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.
- **Proof:**
  - The number of elements examined during a successful search for an element x is one more than the number of elements that appear before x in x's list.
  - We take the average, over the n elements x in the table, of one plus the expected number of elements added to x's list after x was added to the list.

Collision Resolution by Chaining[7]

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

Collision Resolution by Chaining[8]

$$
\begin{aligned}
&= 1 + \frac{1}{nm}\left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right) \\
&= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}\,.
\end{aligned}
$$

$$\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$$

Hash Functions[1]

- What makes a good hash function?
  - Satisfies the assumption of ***simple uniform hashing***.
    - Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.
  - Unfortunately, it is not possible to check this condition.
  - Occasionally we do know the distribution.
    - If we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \le k < 1$, then

      $h(k) = \lfloor km \rfloor$

      satisfies the condition of simple hash function.

Hash Functions[2]

- Use heuristic techniques to create hash function
  - Qualitative information about distribution of keys may be useful in this design process.
    - For example: Compiler's symbol table.
- Derive the hash value in a way independent of any patterns that might exist in the data.
  - For example: Divided by a specified prime number.
- Stronger properties than simple uniform hashing.
  - "Closer" keys yield hash values that are far apart.
- Interpreting keys as natural numbers.
  - ASCII code: pt ⇒ (112, 116), (112×128)+116=14452.

Hash Functions[3]

- ***Division method***
  - Map a key k into one of m slots by taking the remainder of k divided by m.

    $h(k) = k \bmod m$.
- Avoid certain values of m.
  - ***m should not be a power of 2***, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k.
- It is better to make the hash function depend on all the bits of the key.
  - A prime not too close to an exact power of 2 is often a good choice for m. Consider $n = 2000$, choose $m = 701$.

    $h(k) = k \bmod 701$.

Hash Functions[4]

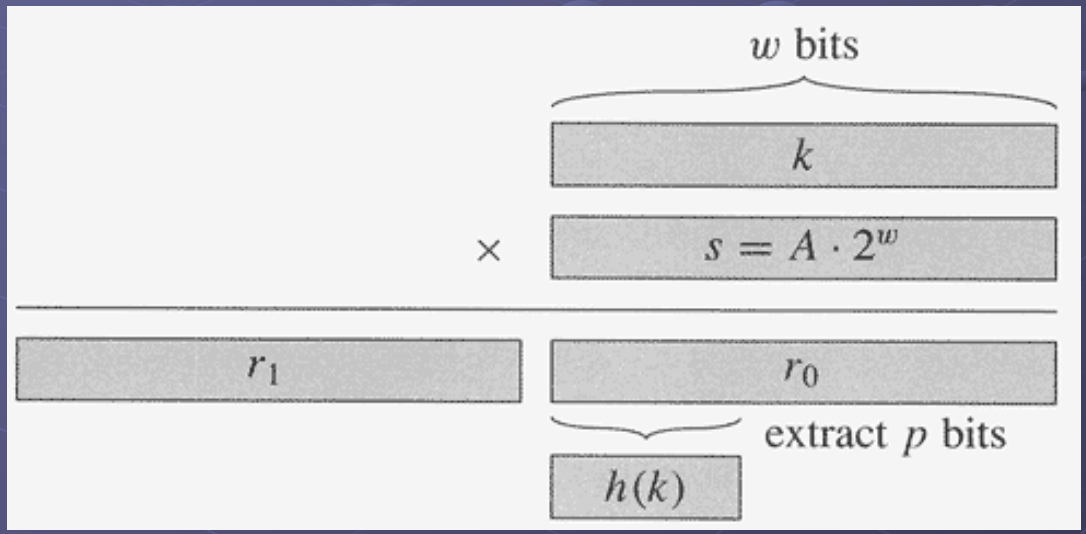
☞ Multiplication method

- Multiply the key k by a constant A, $0 < A < 1$, and extract the fractional part of kA.
- Then multiply kA by m and take the floor of the result.

  $h(k) = \lfloor m(kA \bmod 1) \rfloor$.
- Advantage: The value of m is not critical.
- Implementation on computer
  - $m = 2^p$ for some integer p.
  - A is a fraction of the form $s/2^w$, $0 < s < 2^w$.
  - First multiply k by the w-bit integer $s = A \cdot 2^w$, the result is a $2w$-bit value $r_1 2^w + r_0$.

Hash Functions[5]

- The desired p-bit hash value consists of the p most significant bits of r_0.



- Knuth suggests that

$$A \approx \left(\sqrt{5}-1\right)/2 = 0.6180339887\cdots$$

Open Addressing[1]

- ***Open addressing***
  - All elements occupy the hash table itself.
  - When searching for an element, examine table slots until the desired element is found, or not in the table.
  - Instead of following pointers, we compute the sequence of slots to be examined.
- **Insertion using open addressing**
  - ***Probe***
  - The sequence of positions probed depends upon the key being inserted.

Open Addressing[2]

- Extend the hash function to include probe number

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

  - For every key k, the *probe sequence* $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```

Open Addressing[3]

- Searching
  - The search can terminate (unsuccessfully) when it finds an empty slot.

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Open Addressing[4]

- Deletion from open addressing hash table is difficult.
  - Mark the slot by storing in it the special value DELETED instead of NIL.
  - If store a NIL, we might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupies.
- True uniform hashing is difficult to implement
  - Three commonly used techniques to compute the probe sequences: *linear probing*, *quadratic probing*, and *double hashing*.
  - None of them is capable of generating more than m^2 different probe sequences.

Linear Probing

- Given an ordinary hash function $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, referred as ***auxiliary hash function***,

  $h(k, i) = (h'(k) + i) \bmod m$, for $i = 0, 1, \ldots, m-1$.

  - The first slot probed is $T[h'(k)]$, the next is $T[h'(k) + 1]$, …
  - When up to $T[m-1]$, wrap around to slots $T[0]$, $T[1]$, …
- There are only m distinct probe sequences.
- Suffers from a problem: ***Primary clustering***
  - Long runs of occupied slots build up, increasing the average search time.
  - Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i + 1)/m$.

Quadratic Probing

- Quadratic probing hash function

  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, for $i = 0, 1, \ldots, m - 1$.

  - Auxiliary constants c_1 and c_2 are positive.
  - Initial probe is $T[h'(k)]$.
  - To make full use of hash table, the values of c_1, c_2 and m are constrained.

- *Secondary clustering*

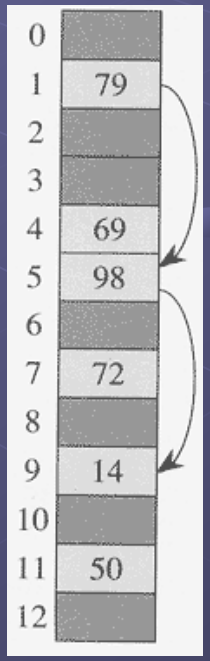
  - If two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$.
  - Only m distinct probe sequences are used.

Double Hashing[1]

- Double hashing function

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

  - Initial probe is $T[h_1(k)]$; successive probes are offset from previous positions by the amount $h_2(k)$, modulo m.



Insert 14:

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

Double Hashing^2

- The value $h_2(k)$ must be ***relatively prime*** to the hash-table size m for the entire hash table to be searched.
  - Let m be a power of 2 and to design h_2 so that it always produces an odd number.
  - Let m be prime and design h_2 so that it always returns a positive integer less than m. For example:

    $h_1(k) = k \bmod m$
    $h_2(k) = 1 + (k \bmod m')$

  - m' is chosen to be slightly less than m.

☞ $\Theta(m^2)$ probe sequences are used.

Analysis of Open-Address Hashing[1]

☞ *Theorem 11.6*

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

  Let A_i be the event that an ith probe occurs and it is an occupied slot. Then the event $\{X \geq i\}$ is the intersection of $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$.

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} .$$

Analysis of Open-Address Hashing[2]

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} .\end{aligned}$$

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} .\end{aligned}$$

$$= 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

Analysis of Open-Address Hashing[3]

☞ *Corollary 11.7*

- **Inserting an element into an open-address hash table with load factor α requires at most $1/(1 - \alpha)$ probes on average, assuming uniform hashing.**
- **Proof:**
  - **Inserting a key requires an unsuccessful search followed by placement of the key in the first empty slot found.**

Analysis of Open-Address Hashing[4]

☞ ***Theorem 11.8***

- **Given an open-address hash table with load factor $\alpha < 1$, the expected number of successful search is at most**

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

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Analysis of Open-Address Hashing[5]

- **Proof:**
  - A search for a key k follows the same probe sequence as was followed when the element with key k was inserted.
  - If k is the $(i+1)$st key inserted in the hash table, the expected number of probes made in a search for k is at most $1/(1 - i/m) = m/(m - i)$.
  - Averaging over all n key in the hash table gives us the average number of probes in a successful search:

Analysis of Open-Address Hashing[6]

$$
\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \quad \text{(by inequality (A.12))} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\,.
\end{aligned}
$$