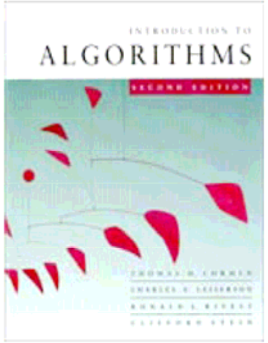
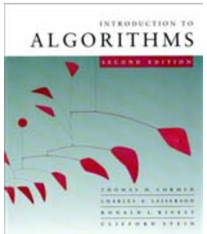
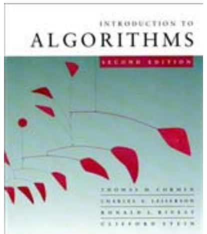


Hash Tables



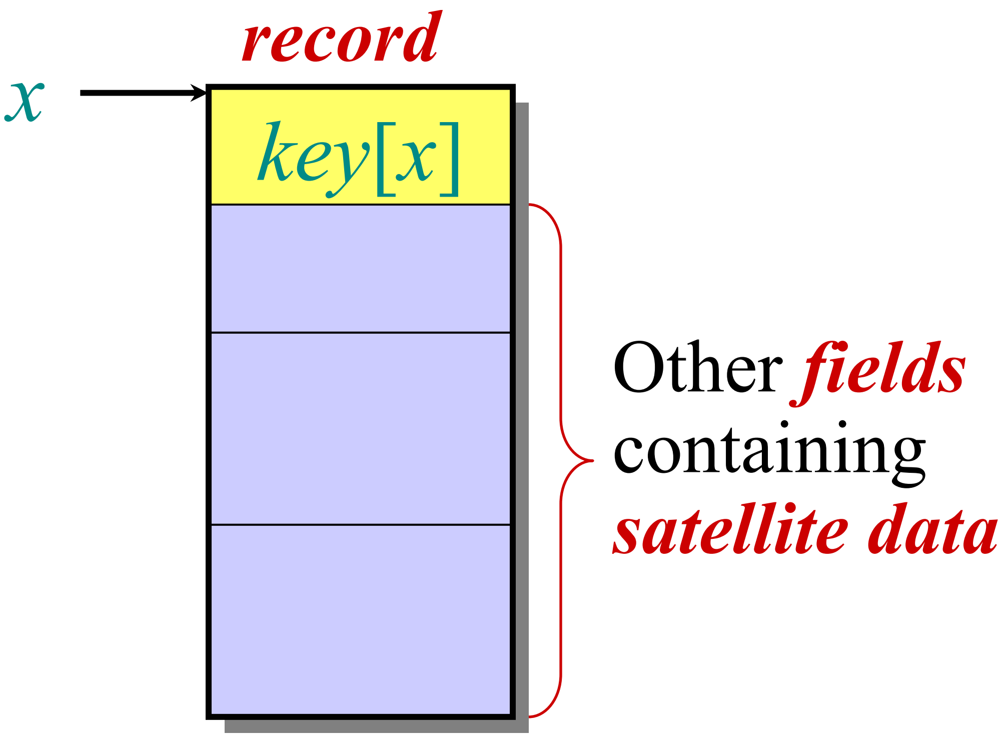
Data Structures

- Role of data structures:
  - Encapsulate data
  - Support certain operations (e.g., INSERT, DELETE, SEARCH)
- What data structures do we know already ?
- Yes, heap:
  - INSERT(x)
  - DELETE-MIN



Dictionary problem

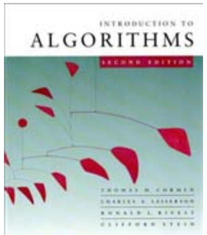
Dictionary T holding n ***records***:



Operations on T:

- INSERT(T, x)
- DELETE(T, x)
- SEARCH(T, k)

How should the data structure T be organized?

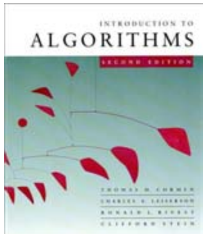


Assumptions

Assumptions:

- The set of keys is $K \subseteq U = \{0, 1, \ldots, u-1\}$
- Keys are distinct

What can we do ?



Direct access table

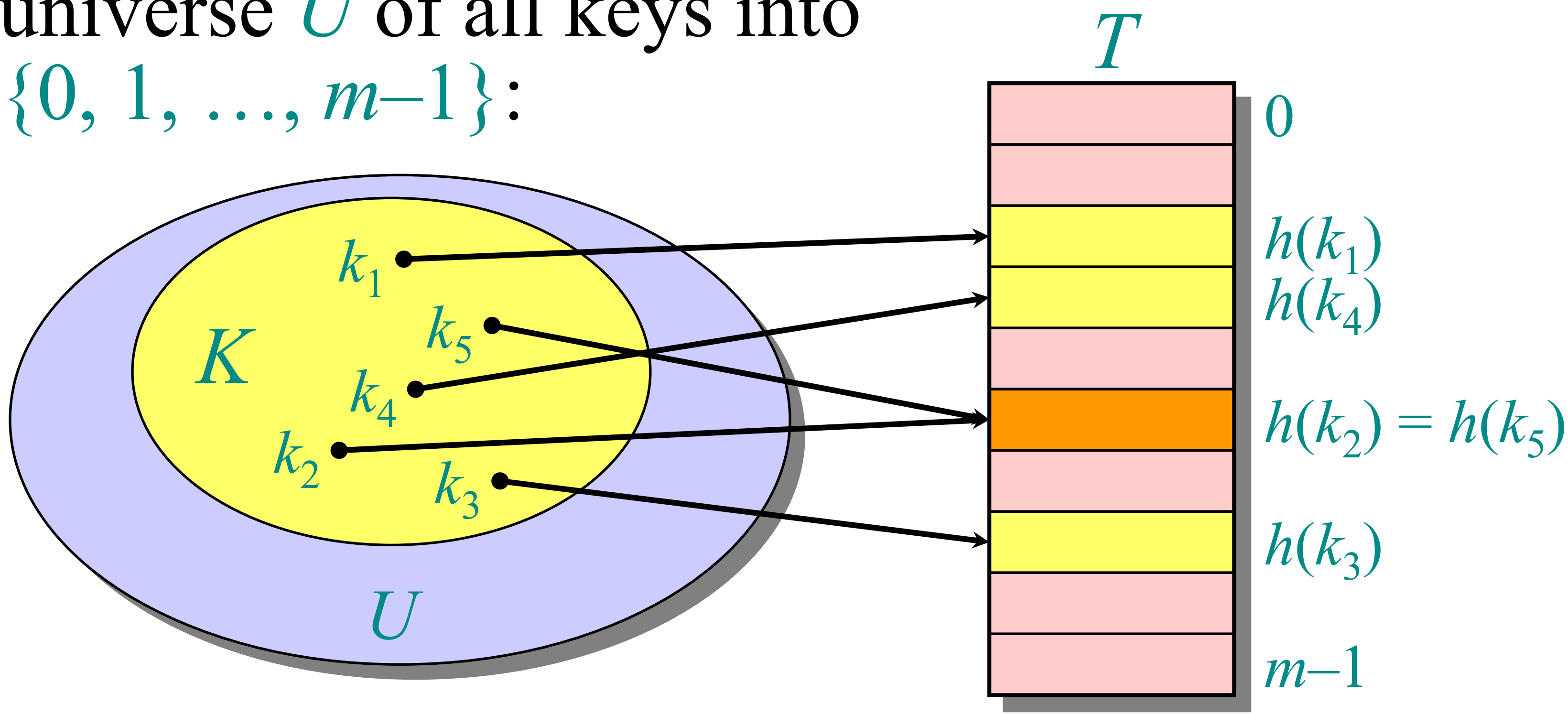
- Create a table $T[0 \ldots u\text{-}1]$:

$$T[k] = \begin{cases} x & \text{if } k \in K \text{ and } key[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

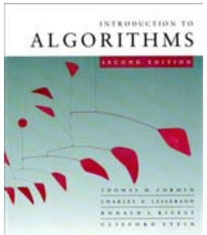
- Benefit:
  - Each operation takes constant time
- Drawbacks:
  - The range of keys can be large:
    - 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
    - character strings (even larger!)

Hash functions

Solution: Use a ***hash function*** h to map the universe U of all keys into $\{0, 1, \ldots, m-1\}$:

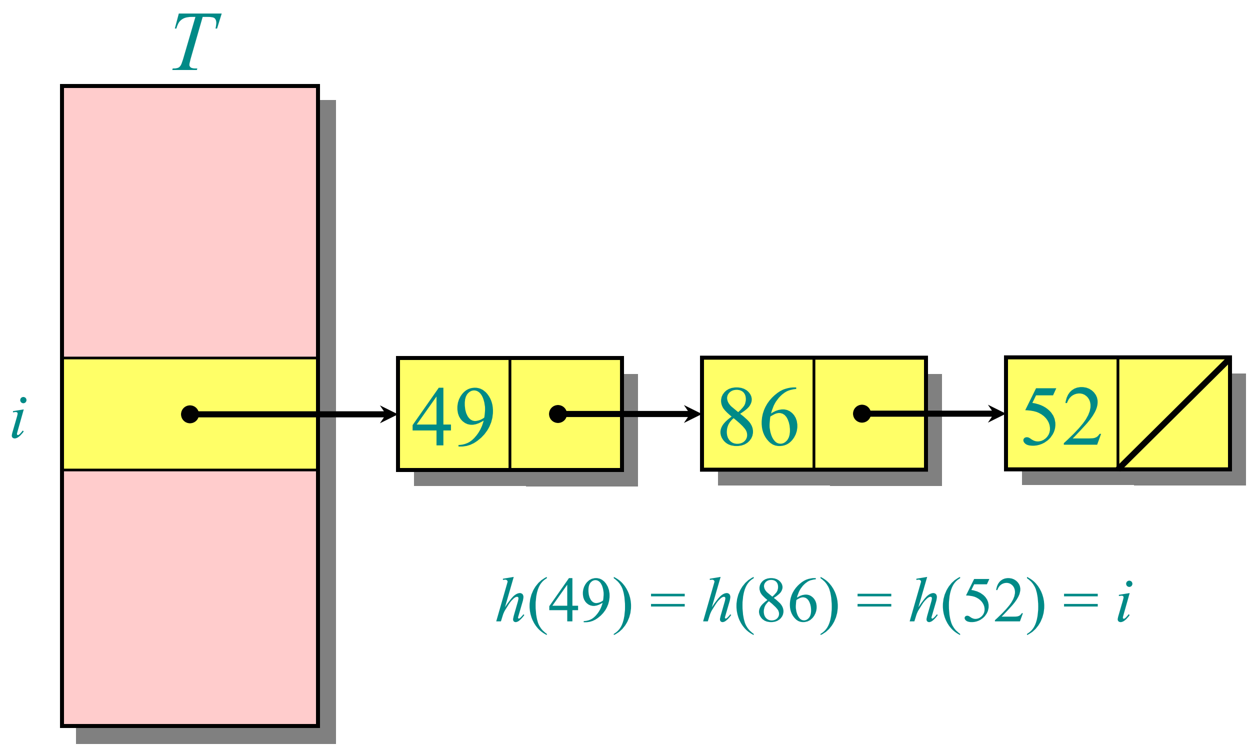


When a record to be inserted maps to an already occupied slot in T, a ***collision*** occurs.



Collisions resolution by chaining

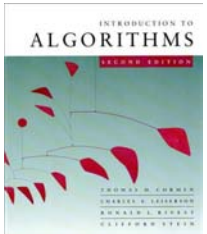
- Records in the same slot are linked into a list.



$$h(49) = h(86) = h(52) = i$$

Hash functions

- Designing good functions is quite non-trivial
- For now, we assume they exist. Namely, we assume ***simple uniform hashing:***
  - Each key $k \in K$ of keys is equally likely to be hashed to any slot of table T, independent of where other keys are hashed

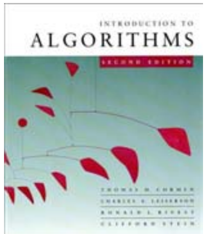


Analysis of chaining

Let n be the number of keys in the table, and let m be the number of slots.

Define the ***load factor*** of T to be

$$\begin{aligned}\alpha &= n/m \\ &= \text{average number of keys per slot.}\end{aligned}$$



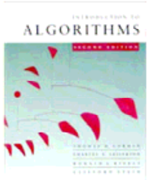
Search cost

Expected time to search for a record with a given key $= \Theta(1 + \alpha)$.

apply hash function and access slot

search the list

Expected search time $= \Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.



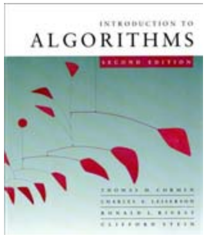
Other operations

- Insertion time ?
  - Constant: hash and add to the list

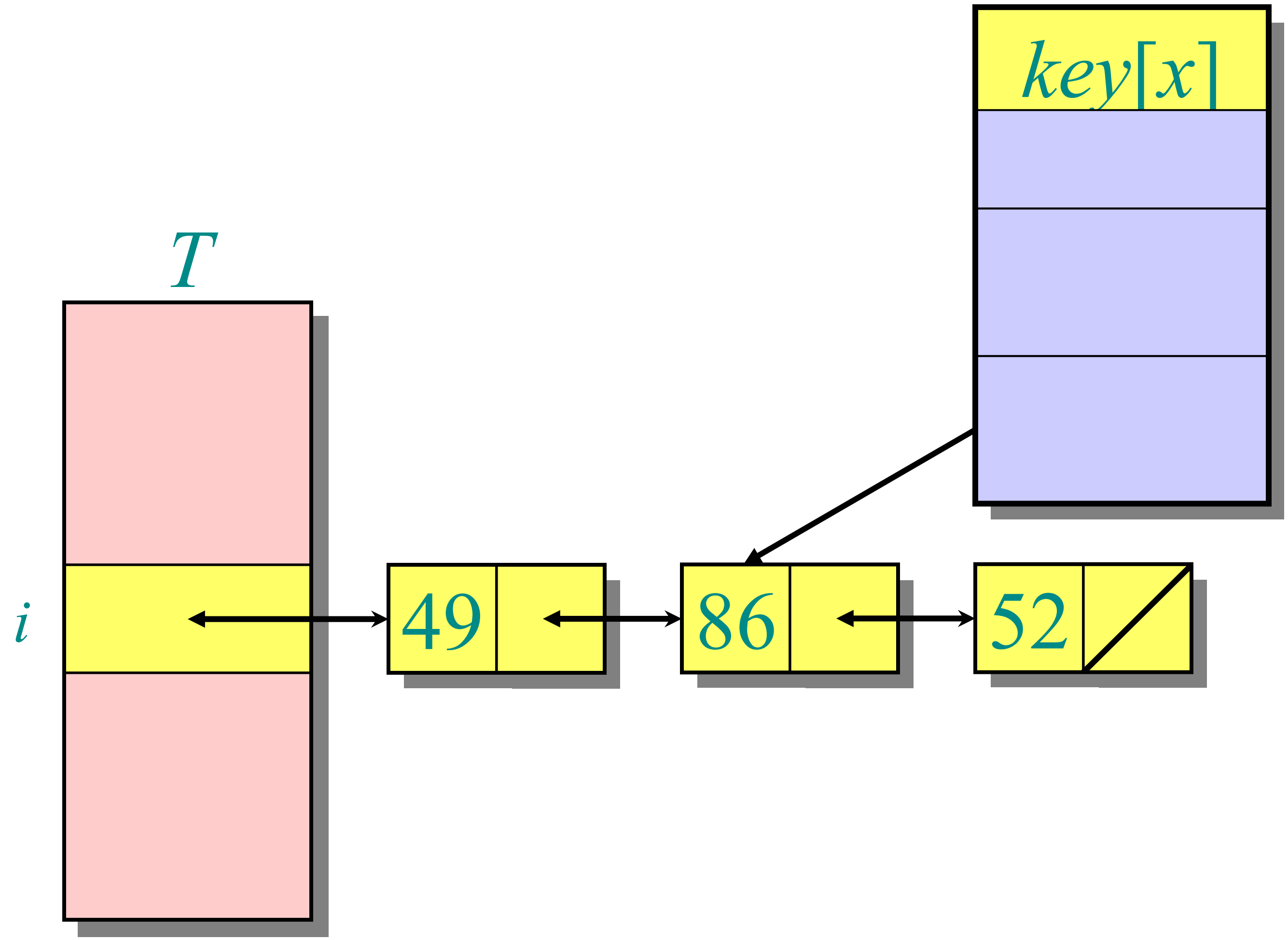
- Deletion time ? Recall that we defined

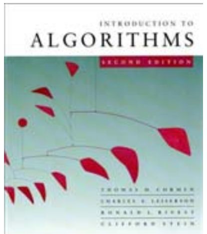
  DELETE(T, x)
  - Also constant, if $\alpha = O(1)$

    where, α is average number of keys per slot
  - Do SEARCH first



Delete



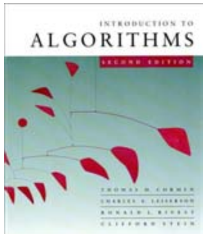


Dealing with wishful thinking

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution (e.g., arithmetic progression) should not affect this uniformity.



Division method

Define

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

Deficiency: Don't pick an m that has a small divisor d. A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: If $m = 2^r$, then the hash doesn't even depend on all the bits of k:

- If $k = 1011000111\underbrace{011010}_{h(k)}{}_2$ and $r = 6$, then $h(k) = 011010_2$.

Division method (continued)

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

Pick m to be a prime.

Annoyance:

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.

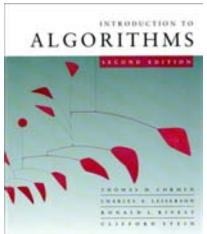
Multiplication method

Assume that all keys are integers, $m = 2^r$, and our computer has w-bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r),$$

where rsh is the "bit-wise right-shift" operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

- Don't pick A too close to 2^w.
- Multiplication modulo 2^w is fast.
- The rsh operator is fast.

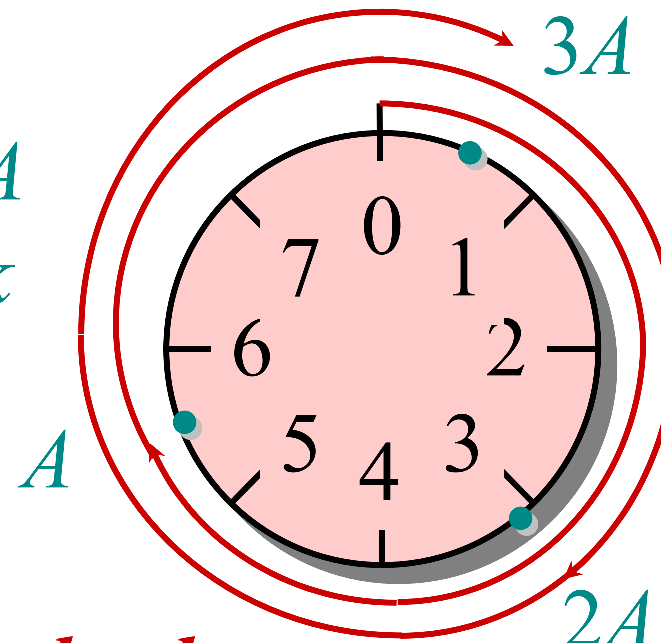


Multiplication method example

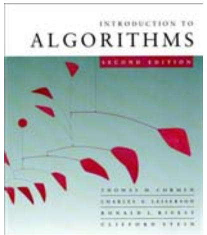
$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$-bit words:

$$
\begin{array}{rl}
1011001 & = A \\
\times \quad 1101011 & = k \\
\hline
1001010\underbrace{011}_{h(k)}0011 &
\end{array}
$$



Modular wheel



Resolving collisions by open addressing

No storage is used outside of the hash table itself.

- Insertion systematically probes the table until an empty slot is found.
- The hash function depends on both the key and probe number:

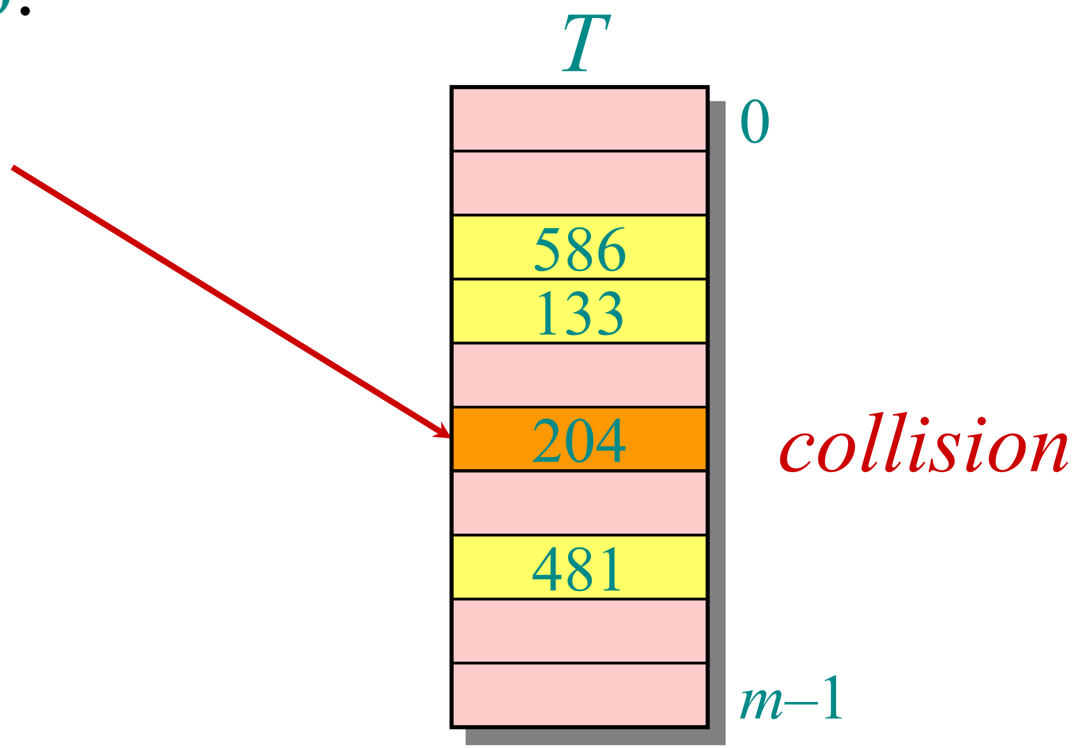
  $$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}.$$

- The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$.
- The table may fill up, and deletion is difficult (but not impossible).

Example of open addressing

Insert key $k = 496$:

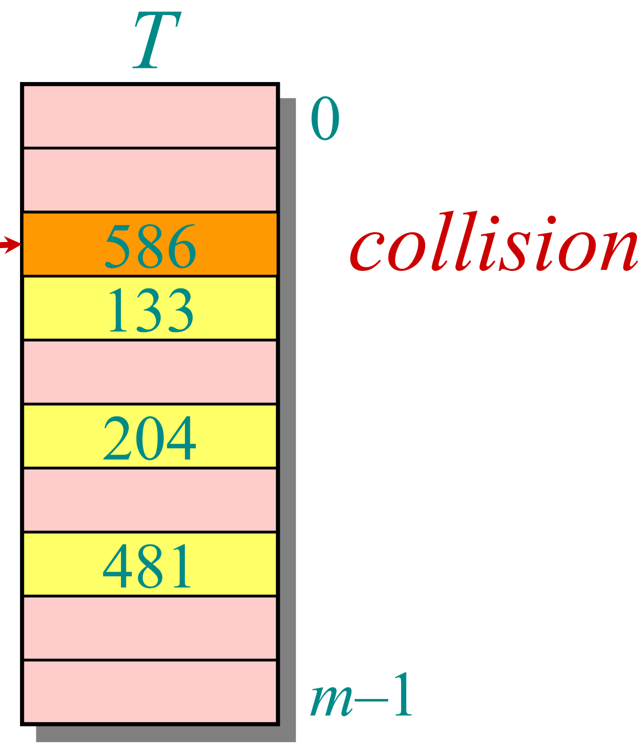
0. Probe $h(496,0)$

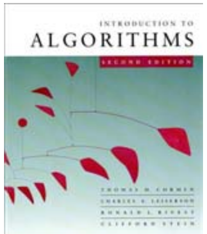


Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$

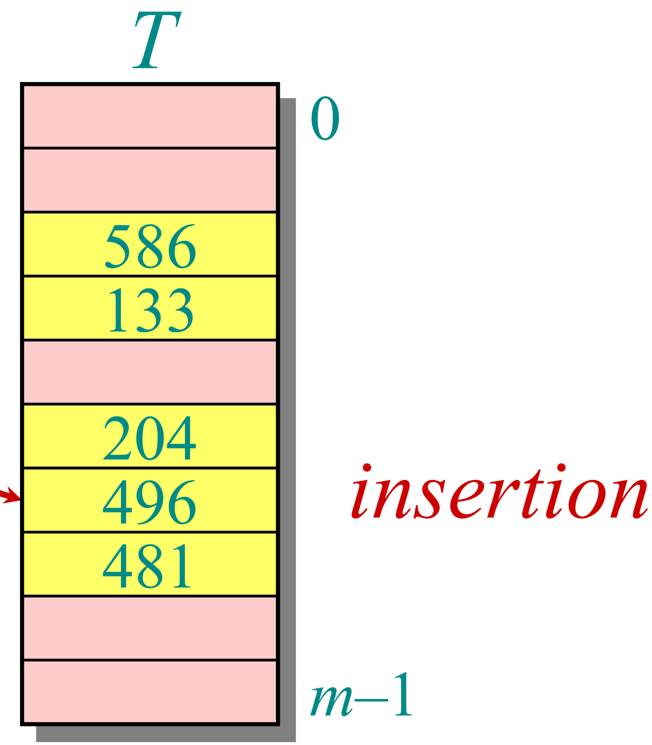


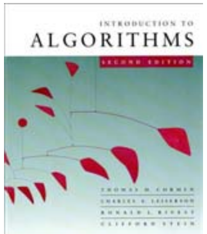


Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$

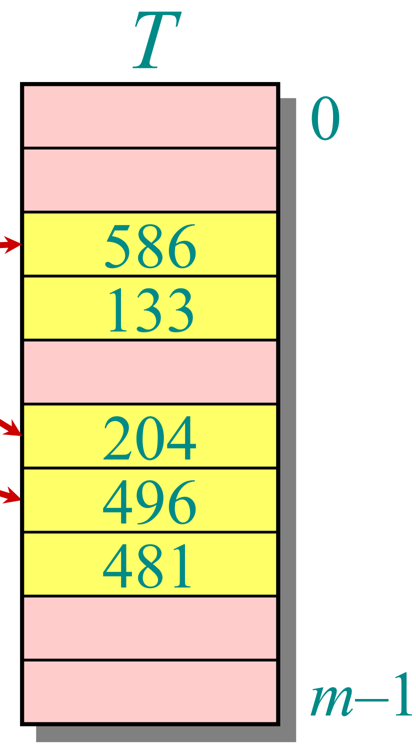




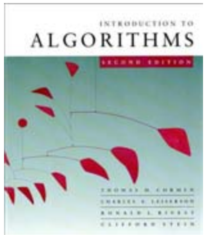
Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.



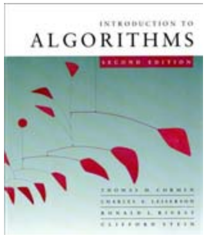
Probing strategies

Linear probing:

Given an ordinary hash function $h'(k)$, linear probing uses the hash function

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

This method, though simple, suffers from ***primary clustering***, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.



Probing strategies

Double hashing

Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

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

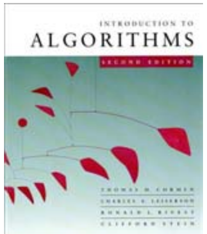
This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m. One way is to make m a power of 2 and design $h_2(k)$ to produce only odd numbers.

Analysis of open addressing

We make the assumption of ***uniform hashing:***

- Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.

Theorem. Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

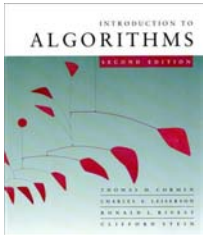


Proof of the theorem

Proof.

- At least one probe is always necessary.
- With probability n/m, the first probe hits an occupied slot, and a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, and a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, etc.

Observe that $\dfrac{n-i}{m-i} < \dfrac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$.



Proof (continued)

Therefore, the expected number of probes is

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$

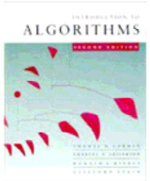
$$\leq 1+\alpha(1+\alpha(1+\alpha(\cdots(1+\alpha)\cdots)))$$

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

$$=\sum_{i=0}^{\infty}\alpha^i$$

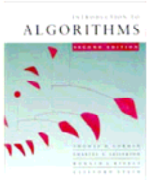
$$=\frac{1}{1-\alpha}. \quad \square$$

The textbook has a more rigorous proof.



Implications of the theorem

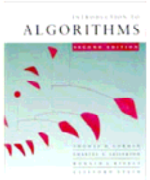
- If α is constant, then accessing the chaining hash table takes constant time.
- If the table is half full, then the expected number of probes is $1/(1–0.5) = 2$.
- If the table is 90% full, then the expected number of probes is $1/(1–0.9) = 10$.



Analysis of open addressing

Corollary. Inserting 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. Thus, the expected number of probes is at most $1/(1-\alpha)$.



Analysis of open addressing

Theorem

The expected number of probes in a successful search is at most $\dfrac{1}{\alpha}\ln\dfrac{1}{1-\alpha}$.

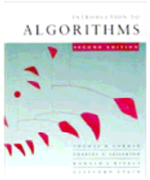
Proof A successful search for key k follows the same probe sequence as when key k was inserted.

By the previous corollary, if k was the $(i+1)$st key inserted, then α equaled i/m at the time. Thus, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$.

That was assuming that k was the $(i+1)$st key inserted. We need to average over all n keys:

$$\begin{aligned}\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) ,\end{aligned}$$

where $\boldsymbol{H_i}$ is the ith harmonic number: 1/1 + 1/2 + … + 1/i

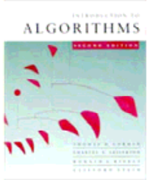


Analysis of open addressing

where $H_i = \sum_{j=1}^{i} 1/j$ is the ith harmonic number.

Simplify by using the technique of bounding a summation by an integral:

$$
\begin{aligned}
\frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} 1/k \\
&\leq \frac{1}{\alpha} \int_{m-n}^{m} (1/x)\, dx \quad \text{(inequality (A.12))} \\
&= \frac{1}{\alpha} \ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \qquad \blacksquare \text{ (theorem)}
\end{aligned}
$$



Analysis of open addressing

- If the table is half full, then the expected number of probes is 1.387.
- If the table is 90% full, then the expected number of probes is 2.559.