Data Structures

Hashing

Uri Zwick

January 2014

Dictionaries

D $\leftarrow$ Dictionary() – Create an empty dictionary

Insert(D,x) – Insert item x into D

Find(D,k) – Find an item with key k in D

Delete(D,k) – Delete item with key k from D

(Predecessors and successors, etc., not supported)

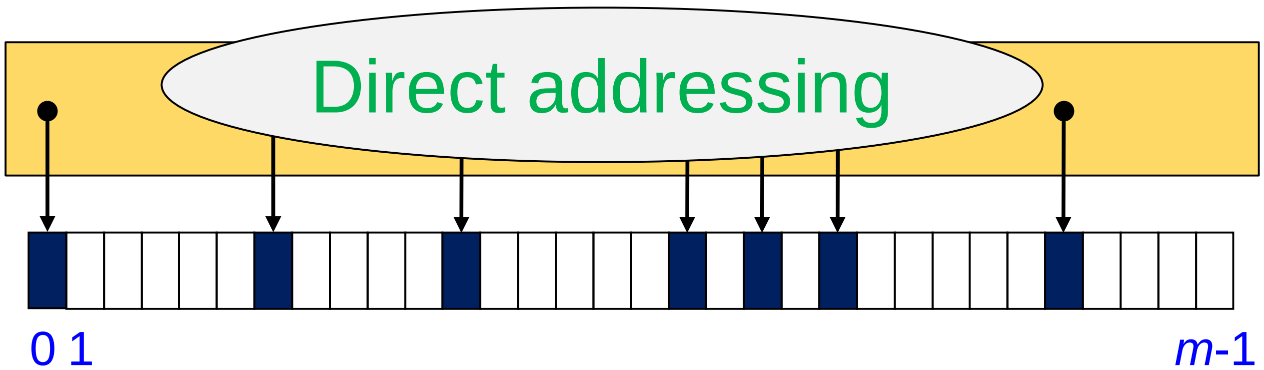
Can use balanced search trees
O(log n) time per operation

Can we do better? YES !!!

Dictionaries with "small keys"

Suppose all keys are in $[m] = \{0,1,\ldots,m-1\}$, where $m = O(n)$

Can implement a dictionary using an array D of length m.



$$\texttt{Insert}(D,x)\ :\ D[x.key] \leftarrow x$$
$$\texttt{Find}(D,k)\ :\ \textbf{return}\ D[k]$$
$$\texttt{Delete}(D,k)\ :\ D[k] \leftarrow null$$

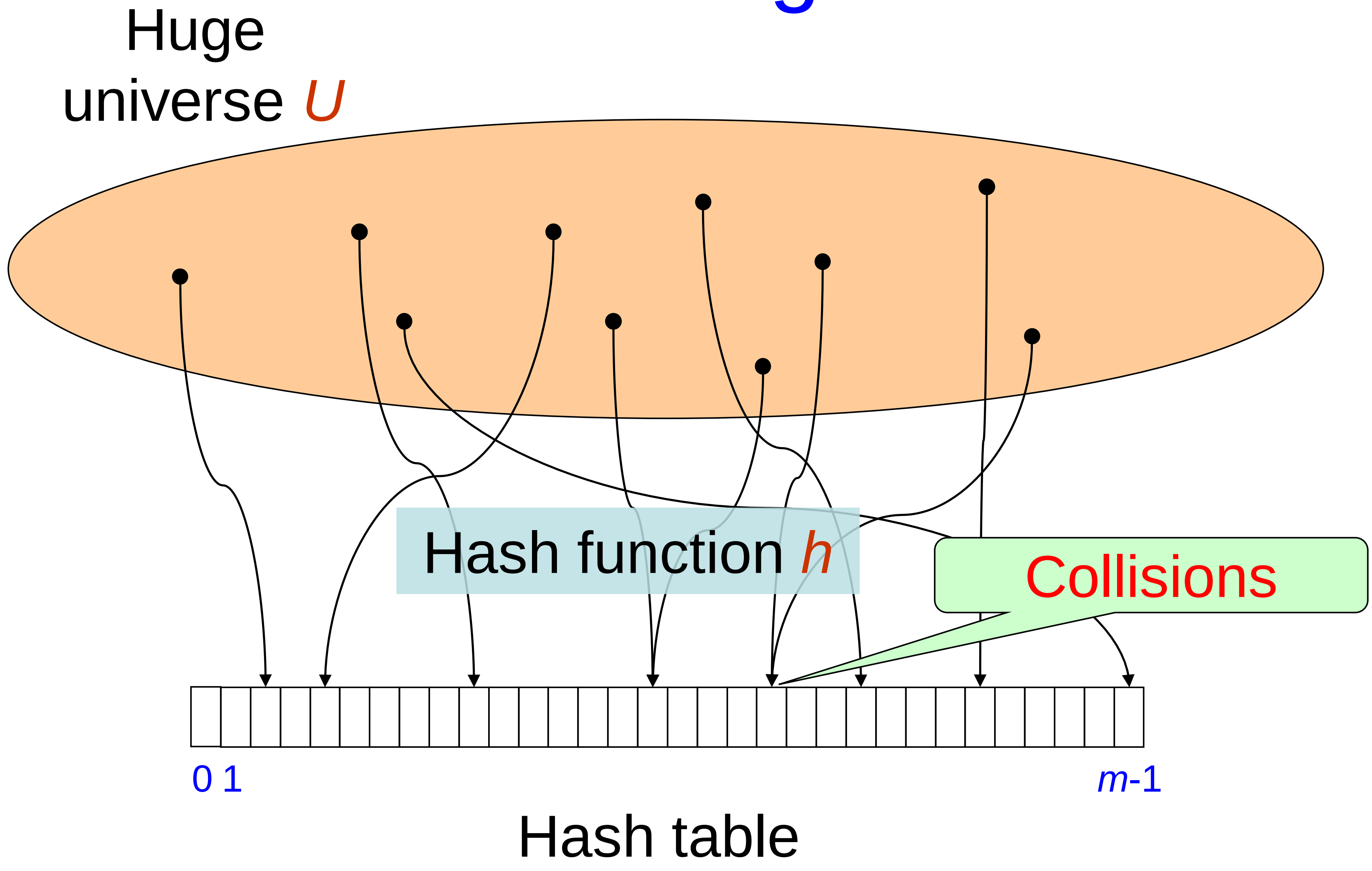
Special case: Sets
D is a bit vector

$O(1)$ time per operation (after initialization)

(Assume different items have different keys.)

What if $m >> n$? Use a hash function

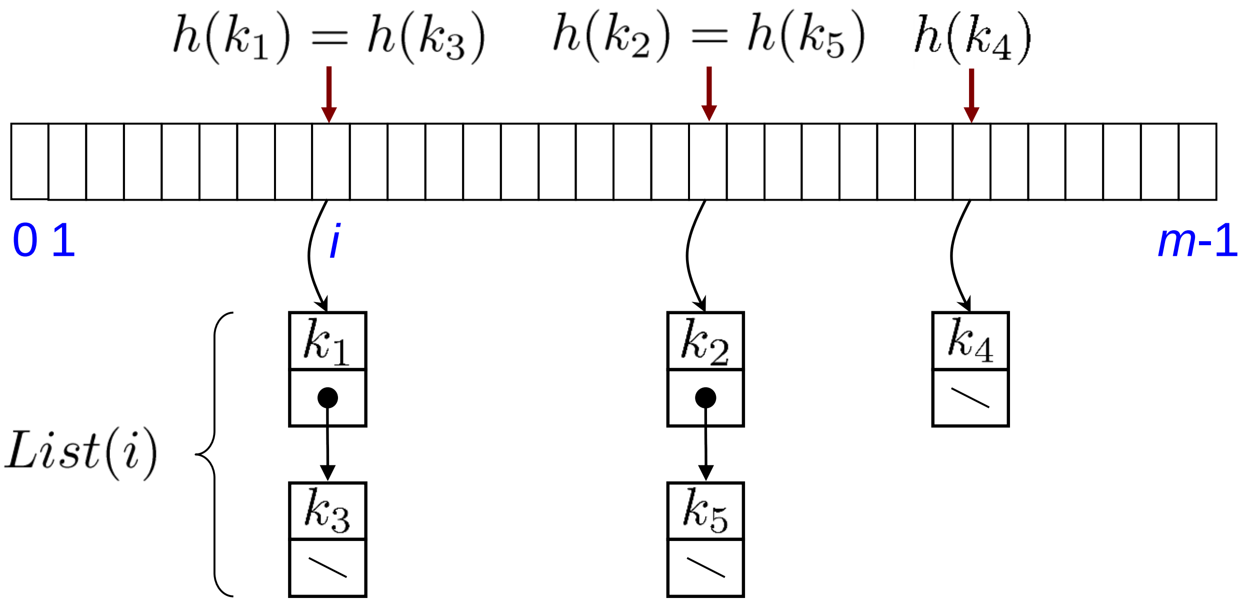
Hashing



Hashing with chaining

[Luhn (1953)] [Dumey (1956)]

Each cell points to a linked list of items



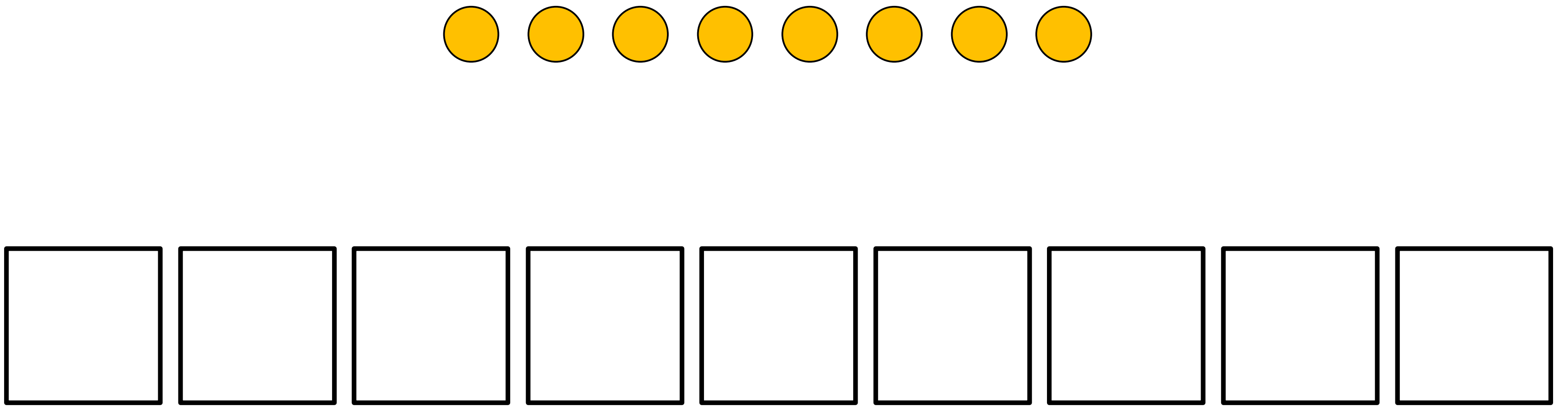
Hashing with chaining
with a random hash function

Balls in Bins

Throw n balls randomly into m bins

Balls in Bins

Throw n balls randomly into m bins



All throws are uniform and independent

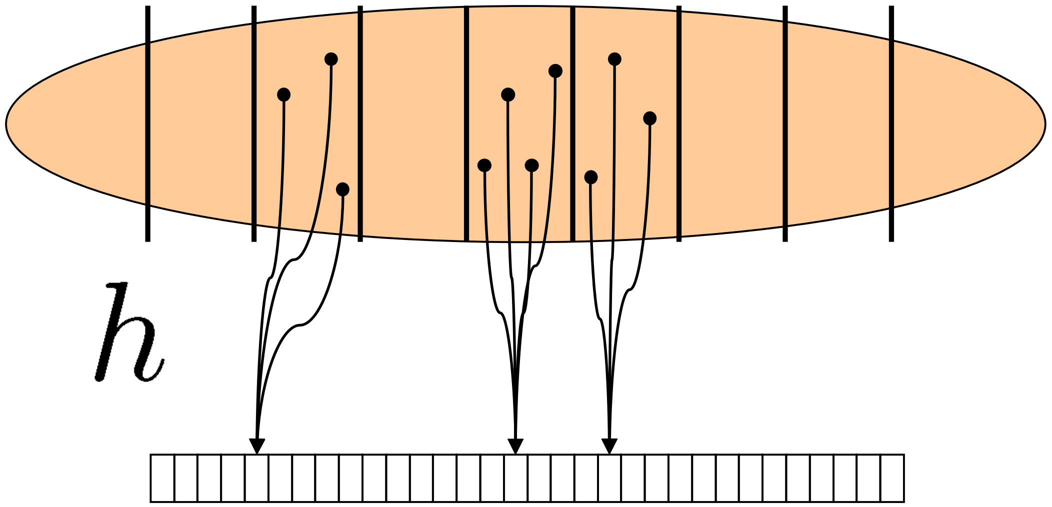
Balls in Bins

Throw n balls randomly into m bins

Expected number of balls in each bin is n/m

When $n=\Theta(m)$, with probability of at least $1-1/n$, all bins contain at most $O(\log n/(\log \log n))$ balls

What makes a hash function good?



Behaves like a "random function"

Has a succinct representation

Easy to compute

A single hash function cannot satisfy the first condition

Families of hash functions

We cannot choose a "truly random" hash function

Using a fixed hash function is usually not a good idea

Compromise:

Choose a random hash function h from a carefully chosen family H of hash functions

Each function h from H should have a succinct representation and should be easy to compute

Goal:

For every sequence of operations, the running time of the data structure, when a random hash function h from H is chosen, is expected to be small

Modular hash functions

[Carter-Wegman (1979)]

$$h(x) \;=\; x \bmod m$$

$$x \bmod 2^k \;=\; x \text{ and } (2^k - 1)$$

$$h_{a,b}(x) \;=\; ((ax+b) \bmod p) \bmod m$$

$$h_{a,b} : U = [p] \to [m]$$

p – prime number

Form a “Universal Family” (see below)

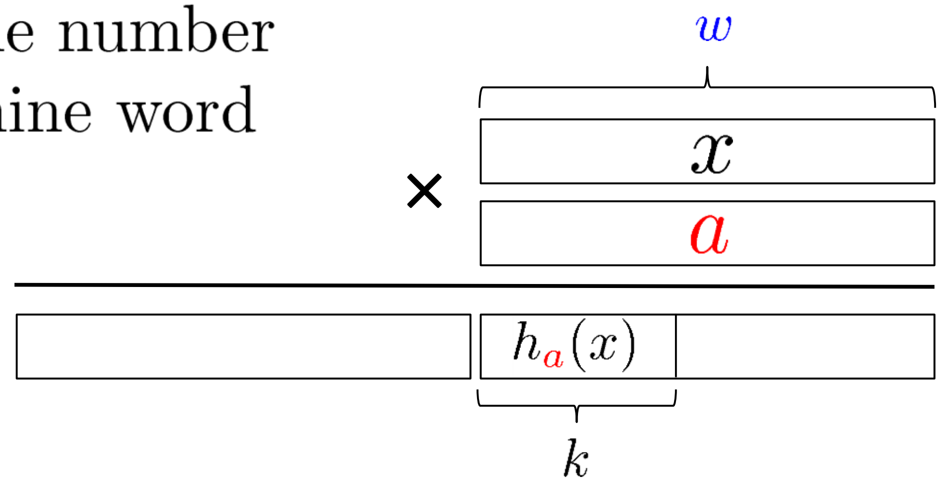
Require (slow) divisions

Multiplicative hash functions

[Dietzfelbinger-Hagerup-Katajainen-Penttonen (1997)]

$$h_a : U = [2^w] \to [2^k] \qquad h_a(x) = \left\lfloor \frac{ax \bmod 2^w}{2^{w-k}} \right\rfloor$$

Typically, w is the number of bits in a machine word



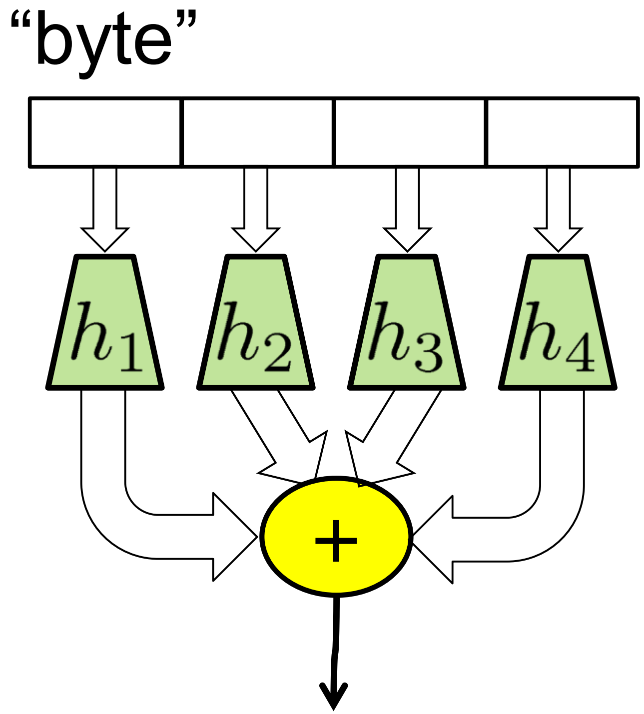
$$h_a(x) = (\texttt{a} * \texttt{x}) >> (\texttt{w}-\texttt{k})$$

Form an "almost-universal" family (see below)

Extremely fast in practice!

Tabulation based hash functions

[Patrascu-Thorup (2012)]



A variant can also be used to hash strings

h_i can be stored in a small table

Very efficient in practice

Very good theoretical properties

Universal families of hash functions

[Carter-Wegman (1979)]

A family H of hash functions from U to $[m]$ is said to be universal if and only if

$$\text{for every } k_1 \neq k_2 \in U \text{ we have}$$
$$\Pr_{h \in H}[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

A family H of hash functions from U to $[m]$ is said to be almost universal if and only if

$$\text{for every } k_1 \neq k_2 \in U \text{ we have}$$
$$\Pr_{h \in H}[h(k_1) = h(k_2)] \leq \frac{2}{m}$$

k-independent families of hash functions

A family H of hash functions from U to $[m]$
is said to be k-independent if and only if

$$\text{for every distinct } x_1, x_2, \ldots, x_k \in U \text{ and } y_1, y_2, \ldots, y_k \in [m]$$
$$\Pr_{h \in H}[h(x_1) = y_1, h(x_2) = y_2, \ldots, h(x_k) = y_k] \;=\; \frac{1}{m^k}$$

A family H of hash functions from U to $[m]$
is said to be almost k-independent if and only if

$$\text{for every distinct } x_1, x_2, \ldots, x_k \in U \text{ and } y_1, y_2, \ldots, y_k \in [m]$$
$$\Pr_{h \in H}[h(x_1) = y_1, h(x_2) = y_2, \ldots, h(x_k) = y_k] \;\leq\; \frac{2}{m^k}$$

A simple universal family

[Carter-Wegman (1979)]

$$U = [p] = \{0, 1, \ldots, p-1\}, \text{ where } p \text{ is prime}$$

$$H_{p,m} \;=\; \{h_{a,b} \mid 1 \le a < p \,,\, 0 \le b < p\}$$

$$h_{a,b}(x) \;=\; ((ax+b) \bmod p) \bmod m$$

$$h_{a,b} : [p] \to [m]$$

Theorem: $H_{p,m}$ is a universal family

To represent a function from the family
we only need two numbers, *a* and *b*.

The size *m* of the hash table can be arbitrary.

A simple universal family

[Carter-Wegman (1979)]

$$U = [p] = \{0, 1, \ldots, p-1\}, \text{ where } p \text{ is prime}$$

$$H_{p,m} \;=\; \{h_{a,b} \mid 1 \le a < p\,,\, 0 \le b < p\}$$

$$h_{a,b}(x) \;=\; ((ax+b) \bmod p) \bmod m$$

$$h_{a,b} : [p] \to [m]$$

Theorem: $H_{p,m}$ is a universal family

Let $x_1 \neq x_2 \in [p]$. For every $y_1 \neq y_2 \in [p]$ there are unique $a, b \in [p]$, $a \neq 0$, such that $y_1 \equiv_p ax_1 + b$ and $y_2 \equiv_p ax_2 + b$.

$$\text{Thus,} \quad \mathbb{P}[y_1 \equiv_m y_2] \;\le\; \frac{\lceil \frac{p}{m} \rceil - 1}{p-1} \;\le\; \frac{\frac{p+m-1}{m} - 1}{p-1} \;=\; \frac{1}{m}$$

Probabilistic analysis of chaining

n – number of elements in dictionary D
m – size of hash table
$\alpha = n/m$ – load factor

Assume that h is randomly chosen from a universal family H

If $k \notin D$, then
$E[|List(h(k))|] \leq \frac{n}{m} = \alpha$

If $k \in D$, then
$E[|List(h(k))|] \leq 1 + \frac{n-1}{m} \leq 1 + \alpha$

	Expected	**Worst-case**
Successful Search **Delete**	$1 + \frac{\alpha}{2}$	n
Unsuccessful Search **(Verified) Insert**	$1 + \alpha$	n

Chaining: pros and cons

Pros:

Simple to implement (and analyze)

Constant time per operation ($O(1+\alpha)$)

Fairly insensitive to table size

Simple hash functions suffice

Cons:

Space wasted on pointers

Dynamic allocations required

Many cache misses

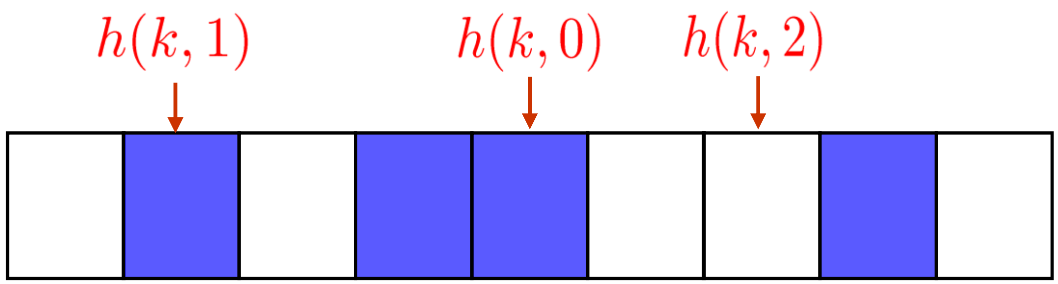
Hashing with open addressing

Hashing without pointers

Assume that $h : U \times [m] \to [m]$

Insert key k in the first free position among

$$\underbrace{h(k,0)\ ,\ h(k,1)\ ,\ h(k,2)\ ,\ \ldots\ ,\ h(k,m-1)}_{\text{Assumed to be a permutation}}$$



No room found → Table is full

To search, follow the same order

Hashing with open addressing

Function `Hash-Insert`(T, k)

for $i \leftarrow 0$ **to** $m-1$ **do**
  $j \leftarrow h(k,i)$
  if $T[j] = null$ **then**
    $T[j] \leftarrow k$
    return j
throw 'hash table full'

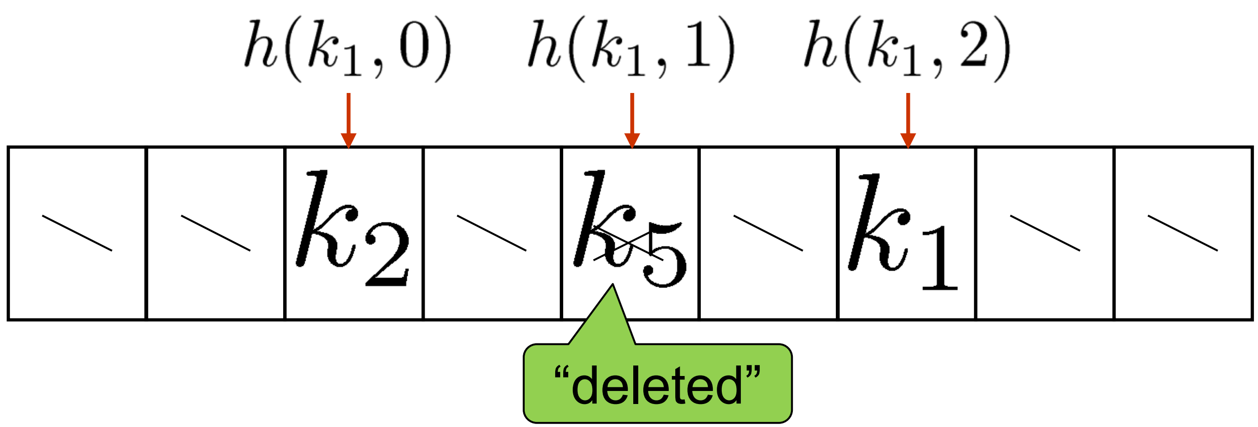
Function `Hash-Search`(T, k)

for $i \leftarrow 0$ **to** $m-1$ **do**
  $j \leftarrow h(k,i)$
  if $T[j] = null$ **then**
    return $null$
  else if $T[j] = k$ **then**
    return j
return $null$

How do we delete elements?

Caution: When we delete elements, do **not** set the corresponding cells to *null*!

$$\cancel{T[j] \leftarrow null} \qquad T[j] \leftarrow deleted$$



Problematic solution…

Probabilistic analysis of open addressing

n – number of elements in dictionary D

m – size of hash table

$\alpha=n/m$ – load factor (Note: $\alpha\le1$)

Uniform probing: Assume that for every k, $h(k,0),\ldots,h(k,m-1)$ is random permutation

Expected time for unsuccessful search

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

Expected time for successful search

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

Probabilistic analysis of open addressing

Claim: Expected no. of probes for an unsuccessful search is at most: $\frac{1}{1-\alpha}$

If we probe a random cell in the table, the probability that it is full is α.

The probability that the first i cells probed are all occupied is at most α^i.

$$1 + \alpha + \alpha^2 + \ldots = \frac{1}{1-\alpha}$$

Open addressing variants

How do we define *h(k,i)* ?

Linear probing:

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

Quadratic probing:

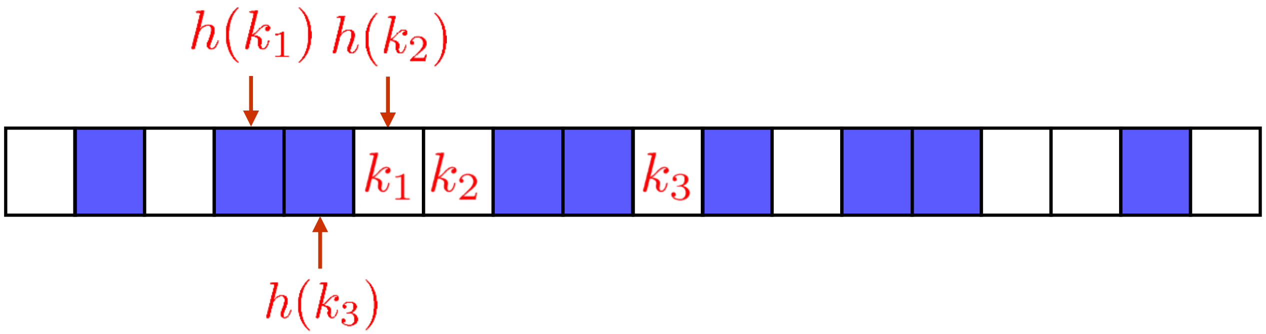
$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Double hashing:

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

Linear probing

"The most important hashing technique"



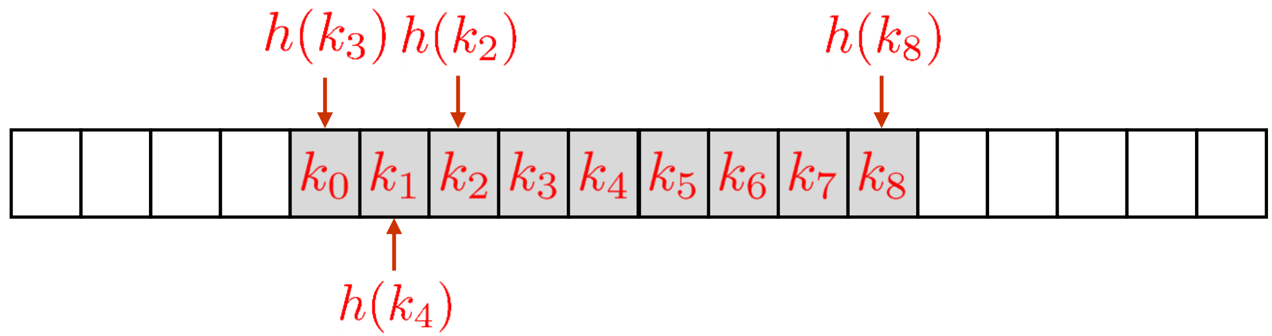
More probes than uniform probing, as probe sequences "merge"

But, much less cache misses

Extremely efficient in practice

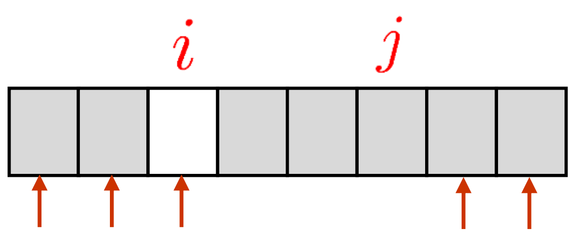
More complicated analysis
(Requires 5-independence or tabulation hashing)

Linear probing – Deletions

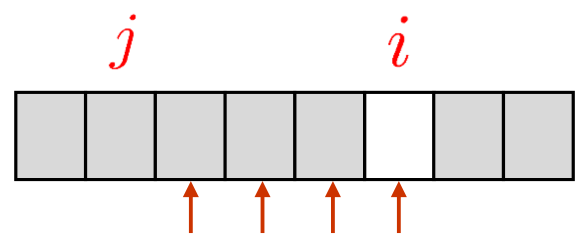


Can the key in cell *j* be moved to cell *i*?

$$h(T[j]) \in \text{“}[j+1, i]\text{”}$$

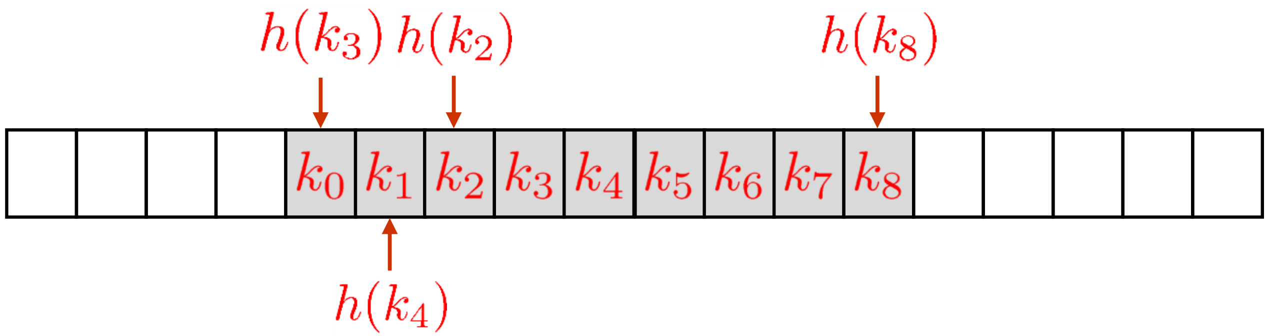


$$h(T[j]) \le i \text{ or } h(T[j]) > j$$



$$j < h(T[j]) \le i$$

Linear probing – Deletions



When an item is deleted, the hash table is in exactly the state it would have been if the item were not inserted!

Expected number of probes

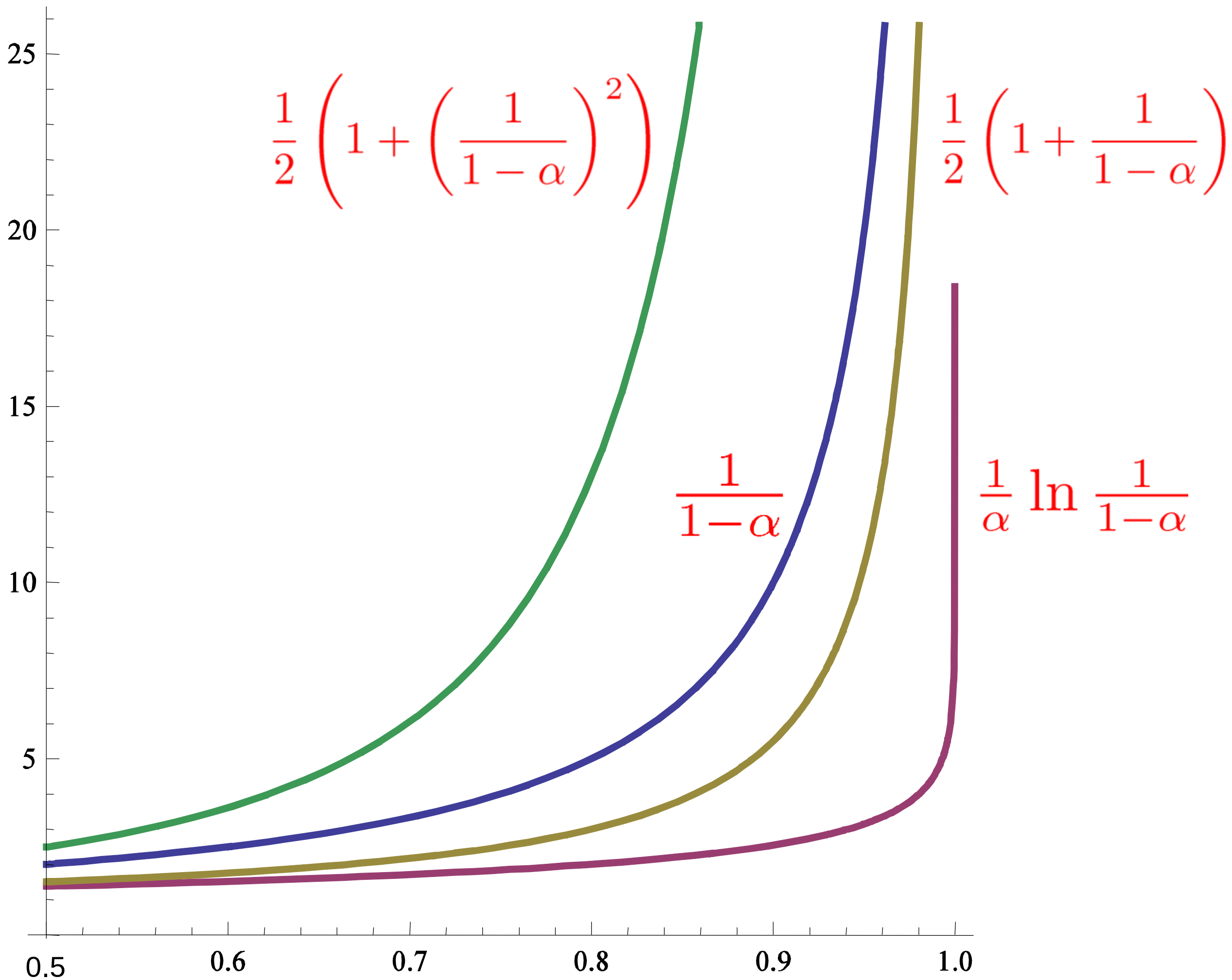
Assuming random hash functions

	Unsuccessful Search	Successful Search
Uniform Probing	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$
Linear Probing	$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

[Knuth (1962)]

When, say, $\alpha \leq 0.6$, all small constants

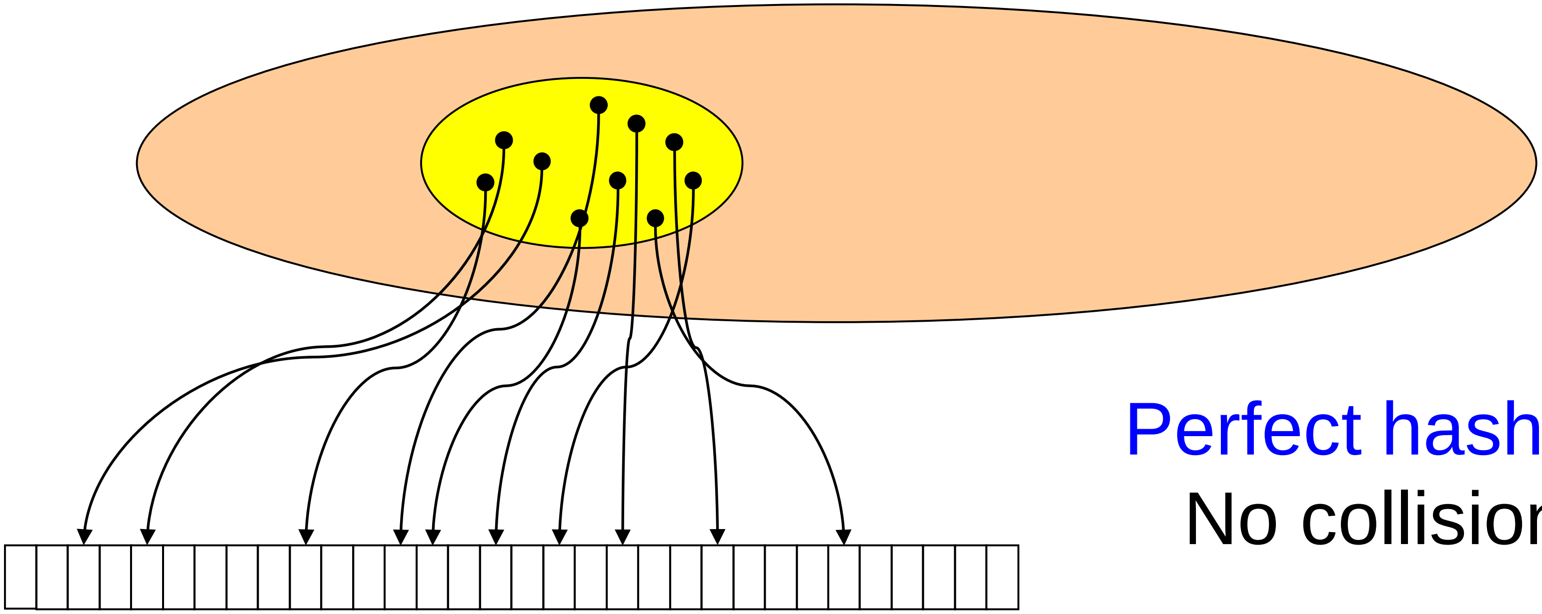
Expected number of probes



Perfect hashing

Suppose that *D* is static.

We want to implement Find is O(1) worst case time.



Perfect hashing:
No collisions

Can we achieve it?

Expected no. of collisions

Suppose that $|D| = n$ and that

h is randomly chosen from a universal family

Collisions:

$$Col \;=\; \{\{k_1, k_2\} \subseteq D \mid k_1 \neq k_2, h(k_1) = h(k_2)\}$$

$$E[|Col|] \;=\; \sum_{\substack{\{k_1,k_2\}\subseteq D \\ k_1 \neq k_2}} \Pr[h(k_1) = h(k_2)] \;\leq\; \frac{\binom{n}{2}}{m}$$

Corollary 1: If $m = n$, then $E[|Col|] \;<\; \frac{n}{2}$

Corollary 2: If $m = n^2$, then $E[|Col|] \;<\; \frac{1}{2}$

Expected no. of collisions

Markov's inequality: $\Pr[X \leq 2E[X]] \geq \frac{1}{2}$

Corollary 1: If $m = n$, then $E[|Col|] \ < \ \frac{n}{2}$

Corollary 1': If $m = n$, then $\Pr[|Col| < n] \ \geq \ \frac{1}{2}$

Corollary 2: If $m = n^2$, then $E[|Col|] \ < \ \frac{1}{2}$

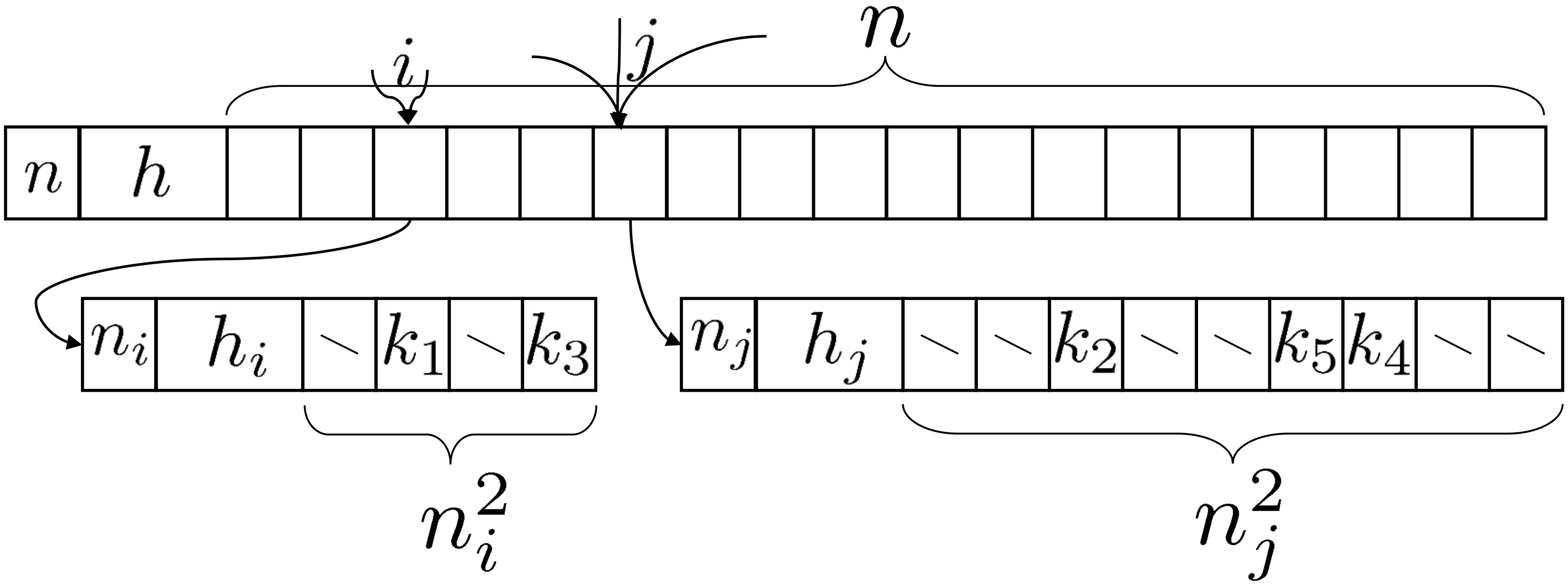
Corollary 2': If $m = n^2$, then $\Pr[|Col| < 1] \ \geq \ \frac{1}{2}$

If we are willing to use *m*=*n*^{2}, then any universal family contains a perfect hash function.

No collisions!

Two level hashing

[Fredman, Komlós, Szemerédi (1984)]

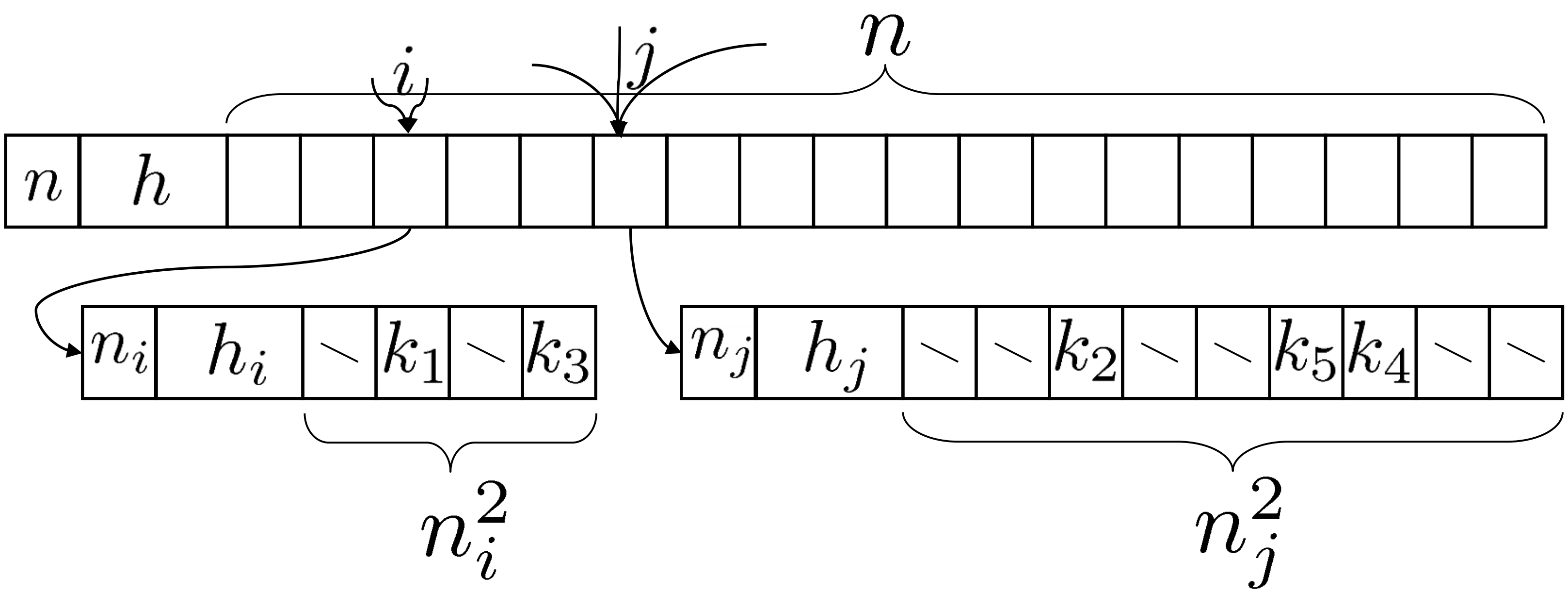


$$n_i = |h^{-1}(i)| = |\{k \in D \mid h(k) = i\}|$$

$$\sum_{i=0}^{n-1} n_i = n \qquad \sum_{i=0}^{n-1} \binom{n_i}{2} = |Col|$$

Two level hashing

[Fredman, Komlós, Szemerédi (1984)]

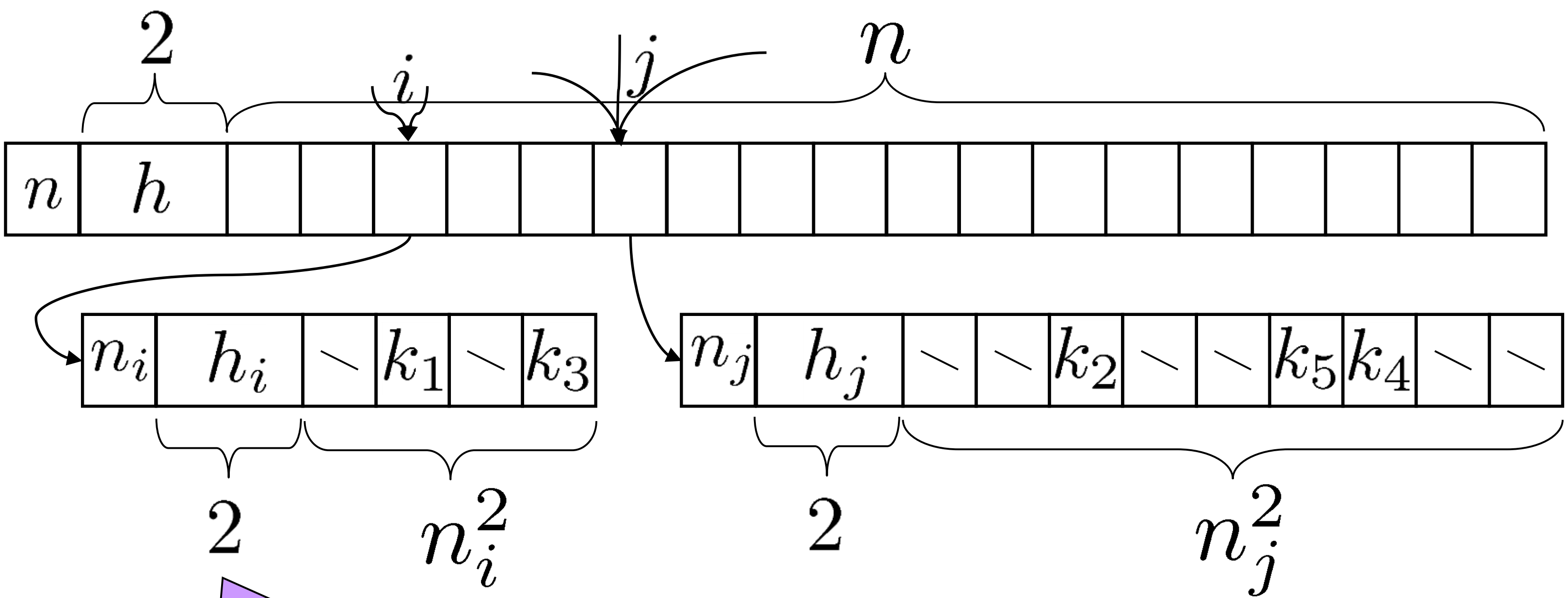


Choose $m = n$ and h such that $|Col| < n$

Store the n_i elements hashed to i
in a small hash table of size n_i^2
using a *perfect* hash function h_i

Two level hashing

[Fredman, Komlós, Szemerédi (1984)]



Assume that each h_i can be represented using 2 words

Total size:

$$
\begin{aligned}
& 3+n+3n+\sum_i n_i^2 \\
= \quad & 4n+3+\sum_i \left(2\binom{n_i}{2}+n_i\right) \\
= \quad & 5n+3+2|Col| \\
\leq \quad & 7n+3
\end{aligned}
$$

A randomized algorithm for constructing a perfect two level hash table:

Choose a random h from $H(\mathrm{p},n)$ and compute the number of collisions. If there are more than n collisions, repeat.

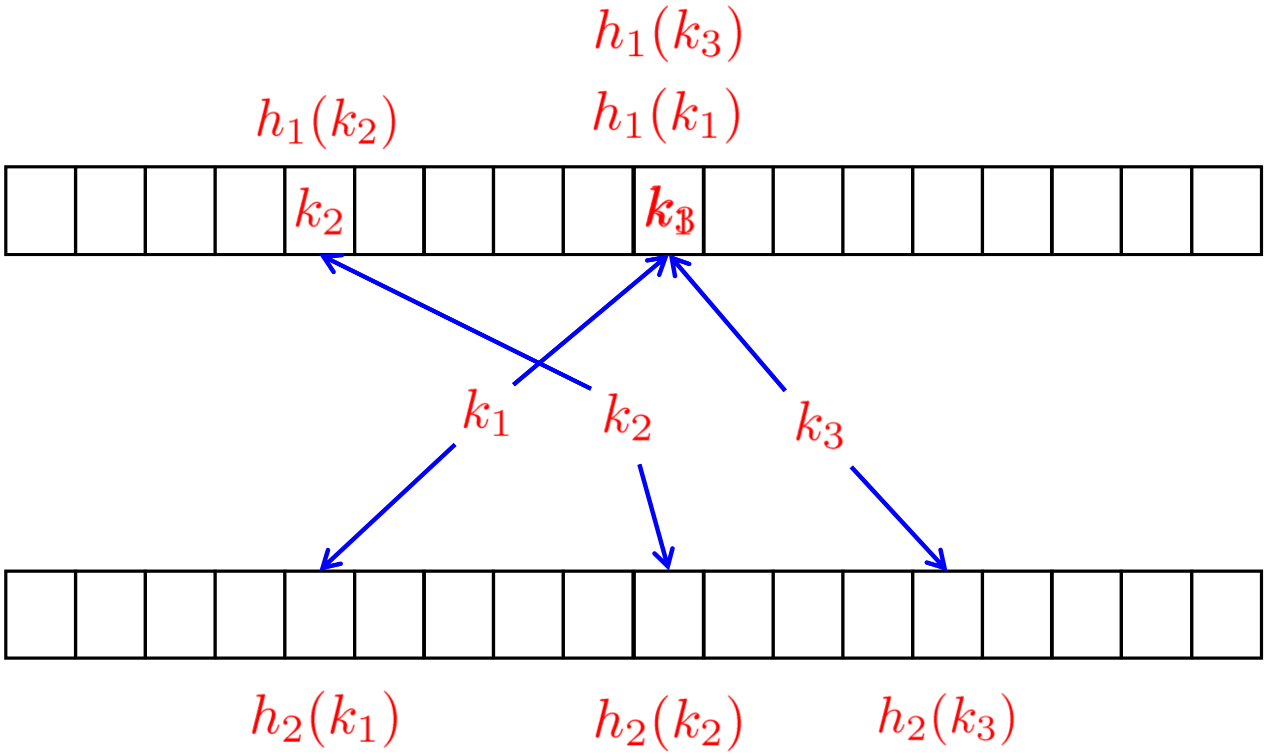
For each cell i,if $n_i>1$, choose a random hash function h_i from $H(\mathrm{p},n_i^2)$. If there are any collisions, repeat.

Expected construction time – $O(n)$

Worst case search time – $O(1)$

Cuckoo Hashing

[Pagh-Rodler (2004)]



Cuckoo Hashing

[Pagh-Rodler (2004)]

Function `Cuckoo-Search`(T, k)

$i_1 \leftarrow h_1(k)$
if $T_1[i_1].key = k$ **then return** $T_1[i_1]$

$i_2 \leftarrow h_2(k)$
if $T_2[i_2].key = k$ **then return** $T_2[i_2]$

return $null$

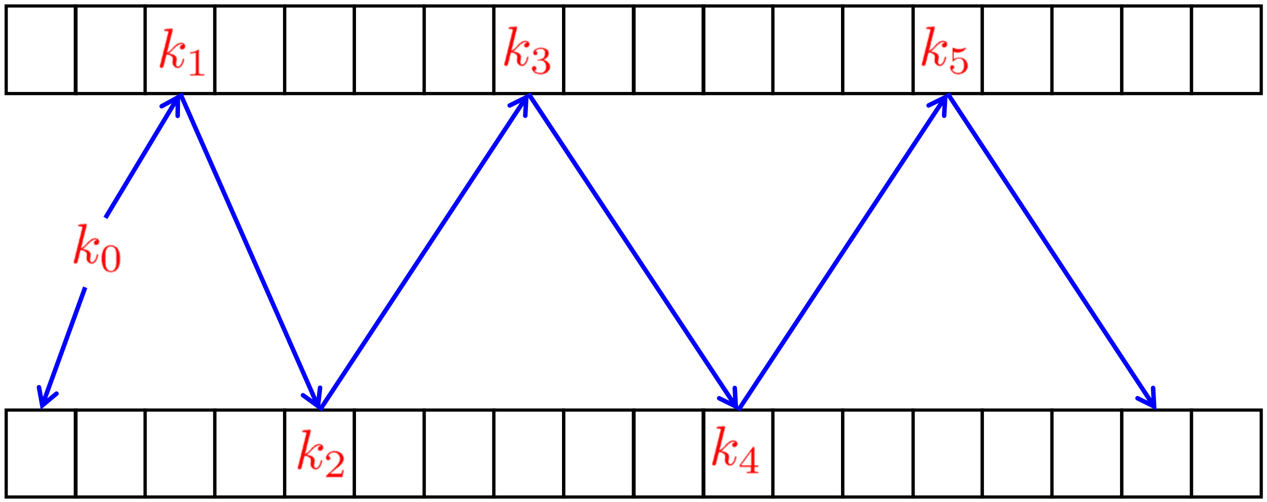
O(1) worst case search time!

What is the (expected) insert time?

Cuckoo Hashing

[Pagh-Rodler (2004)]

Difficult insertion

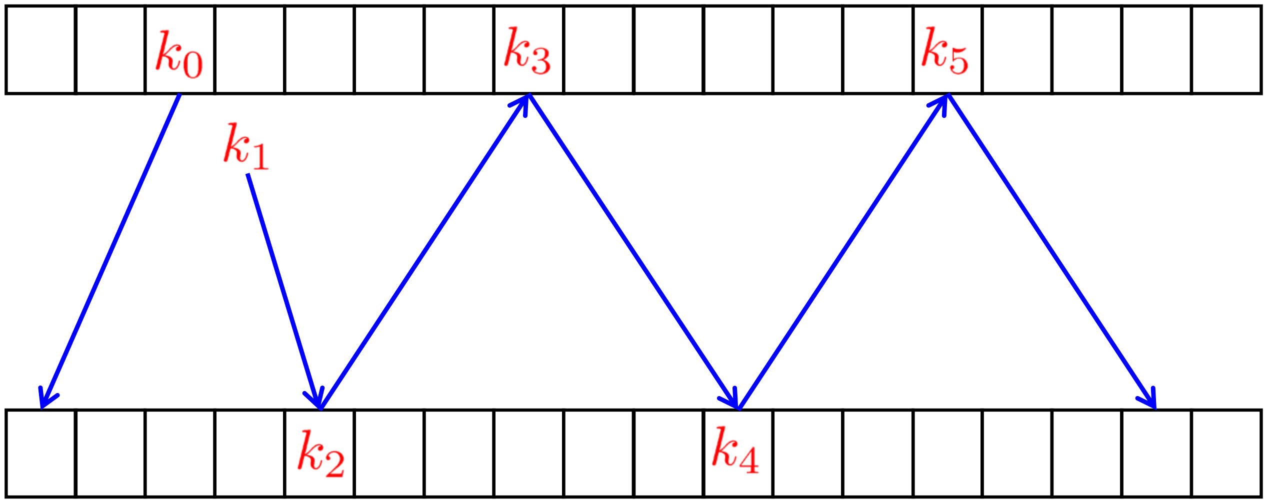


How likely are difficult insertion?

Cuckoo Hashing

[Pagh-Rodler (2004)]

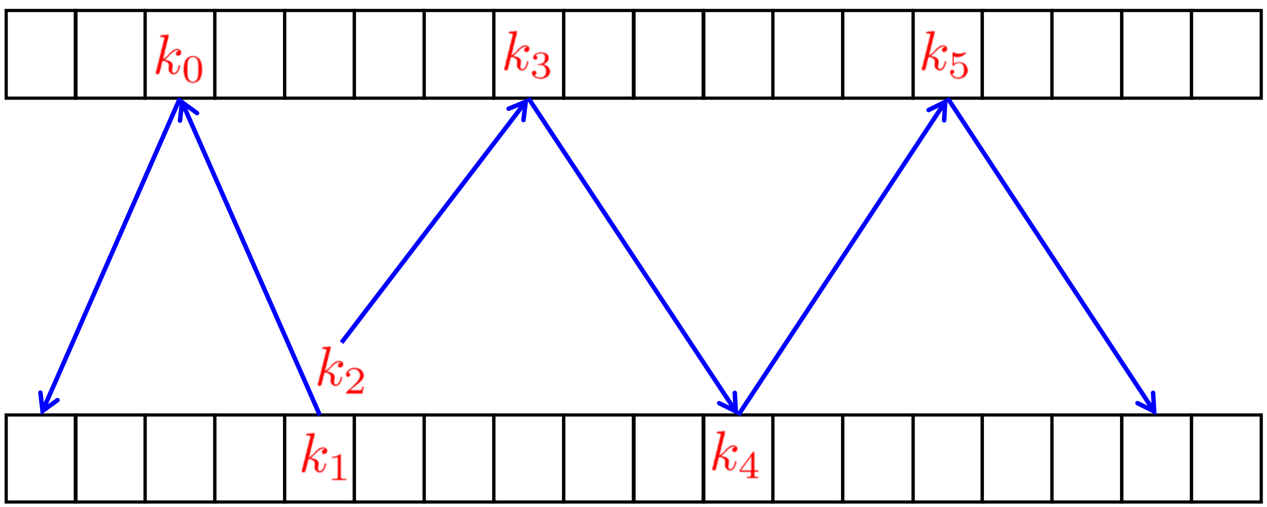
Difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

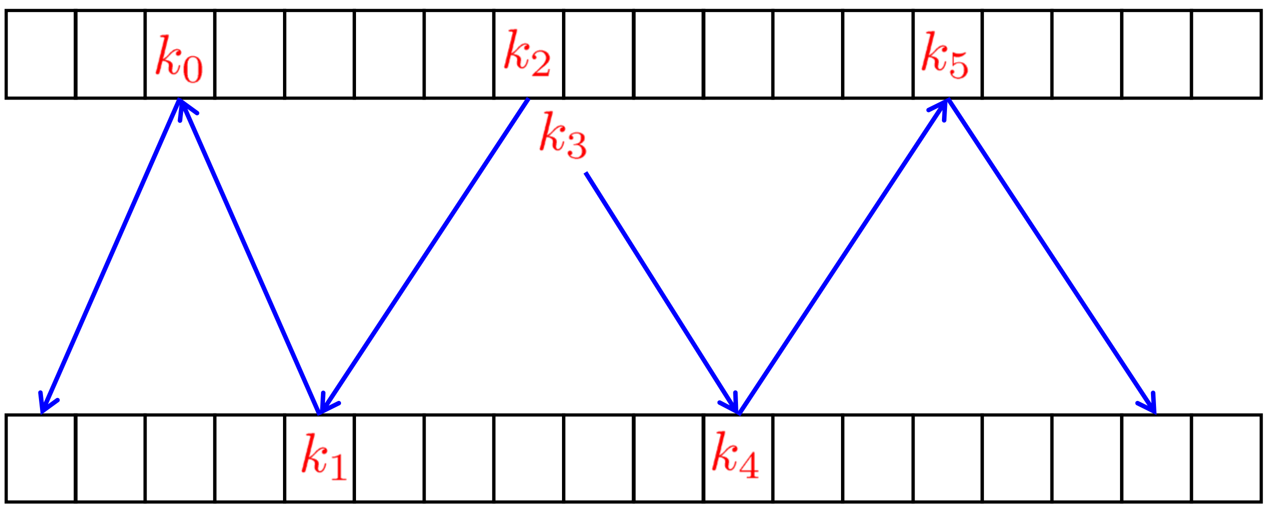
Difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

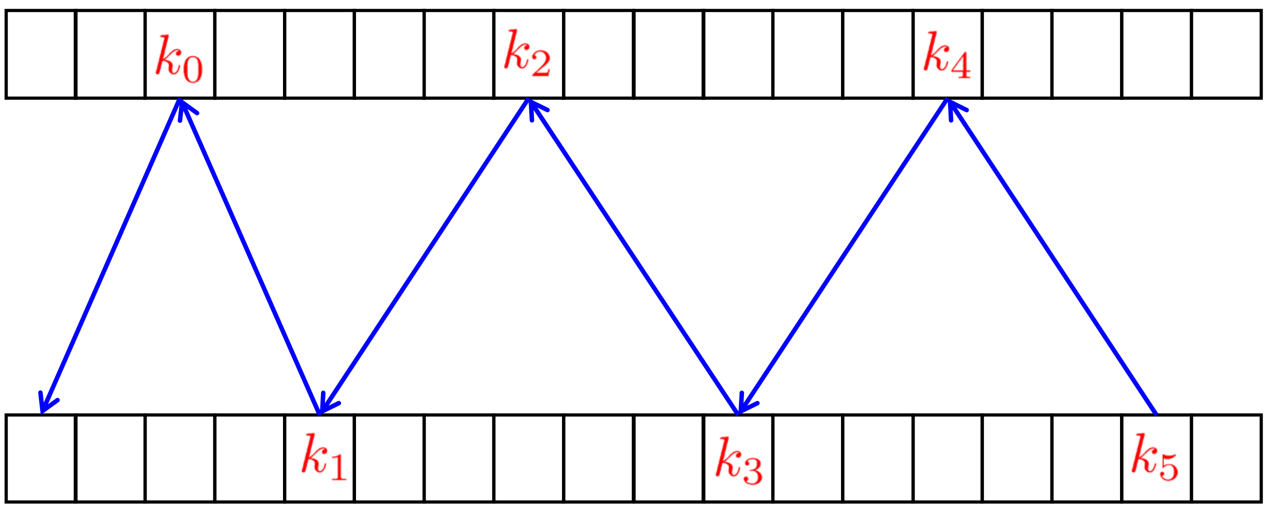
Difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

Difficult insertion

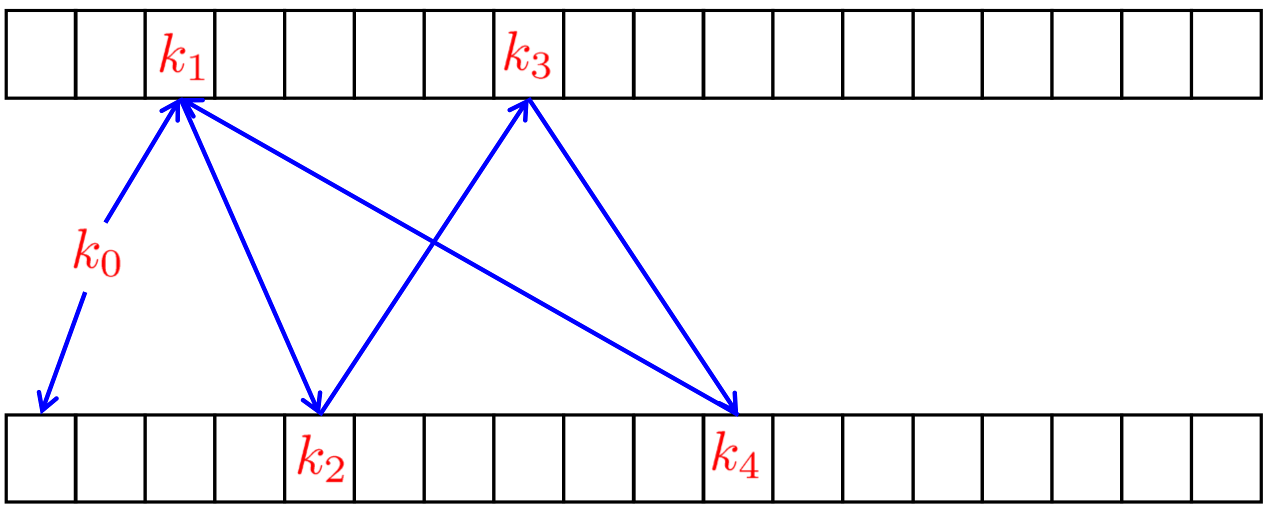


How likely are difficult insertion?

Cuckoo Hashing

[Pagh-Rodler (2004)]

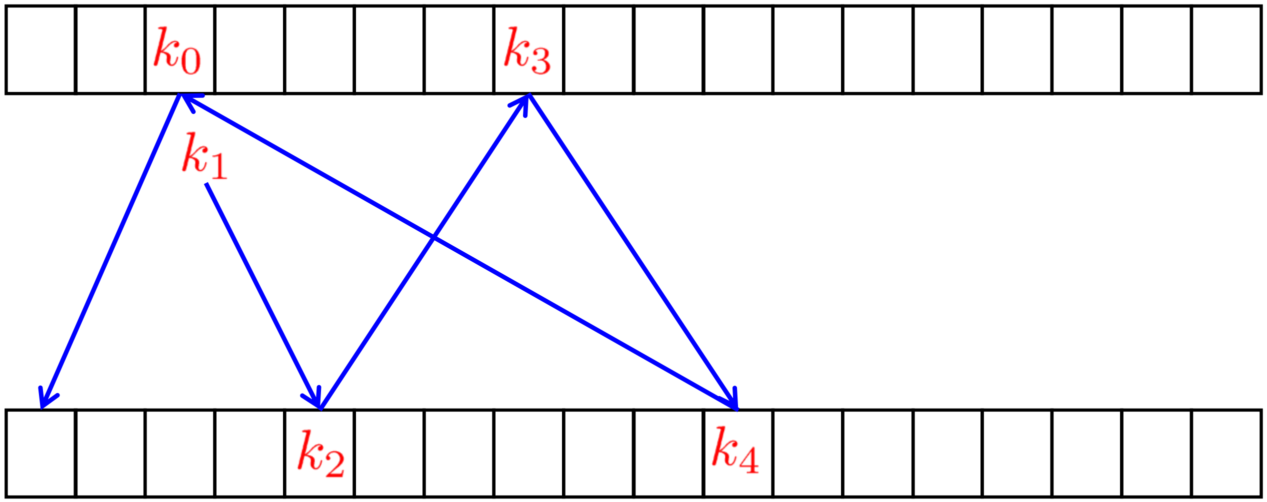
A more difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

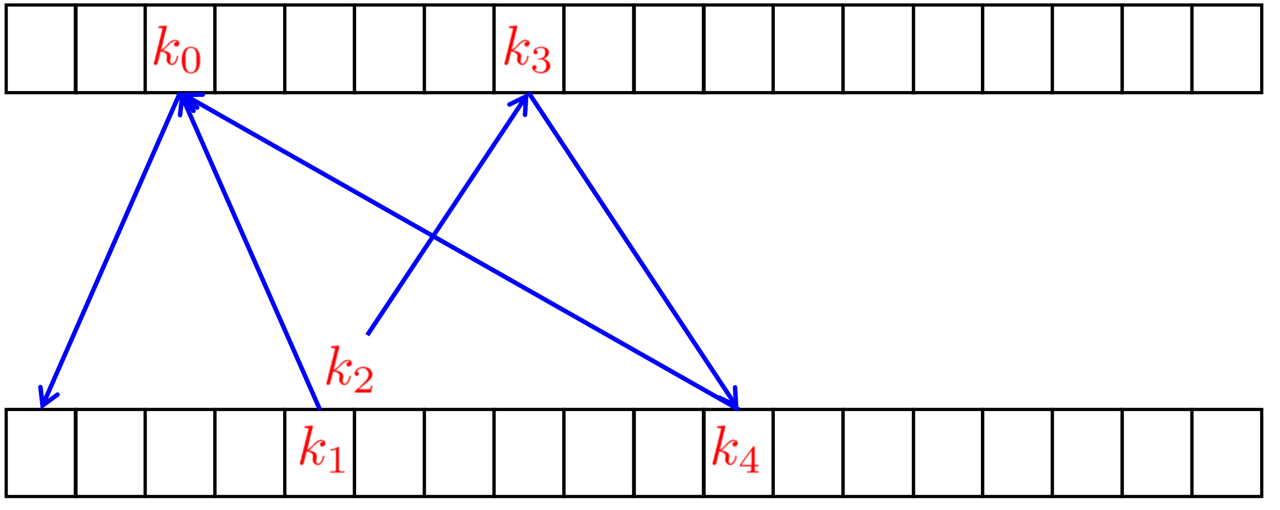
A more difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

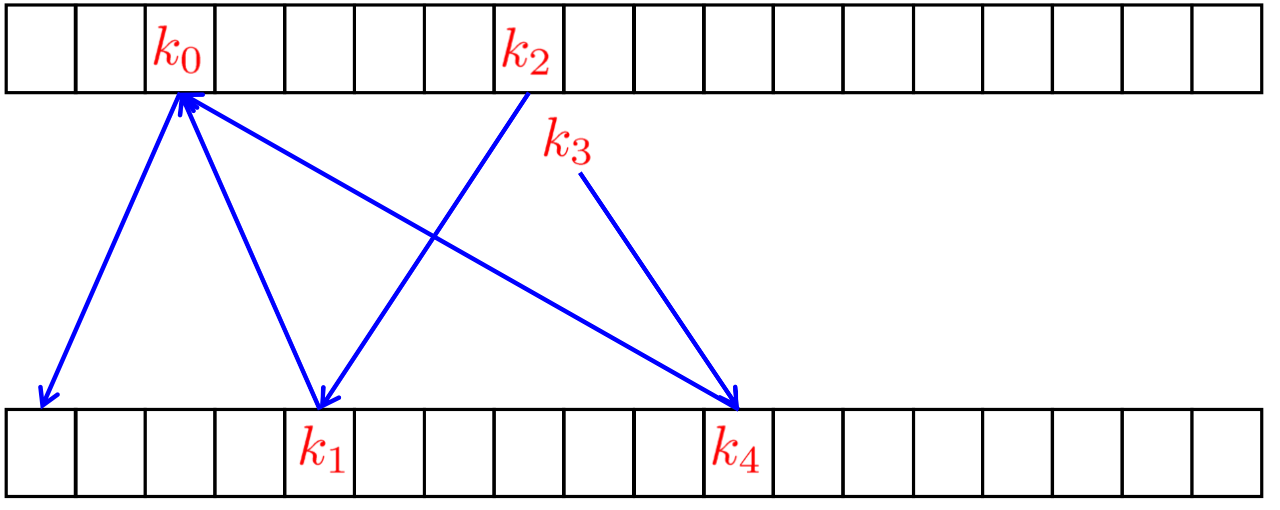
A more difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

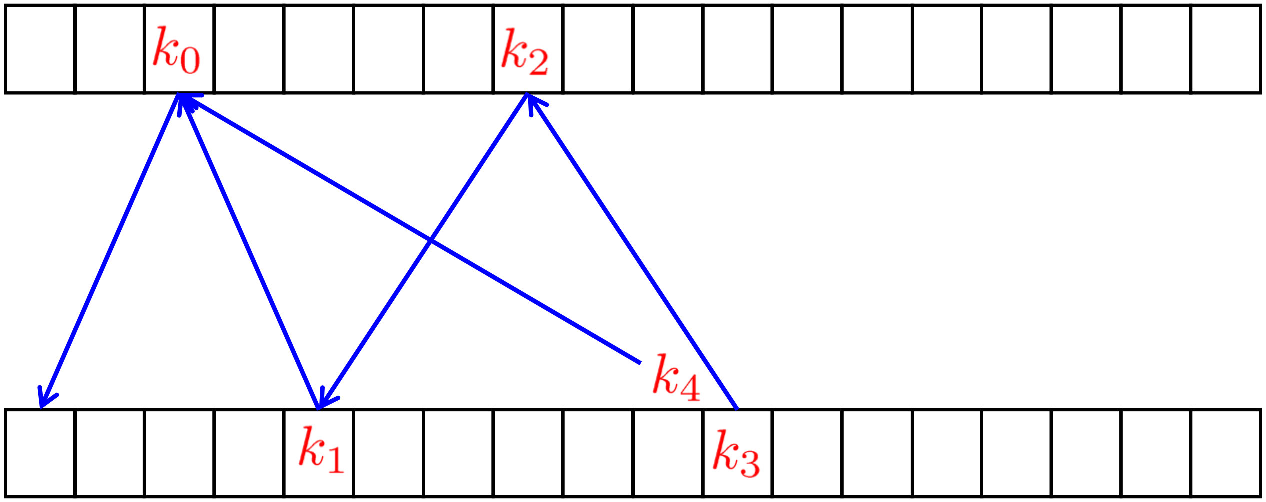
A more difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

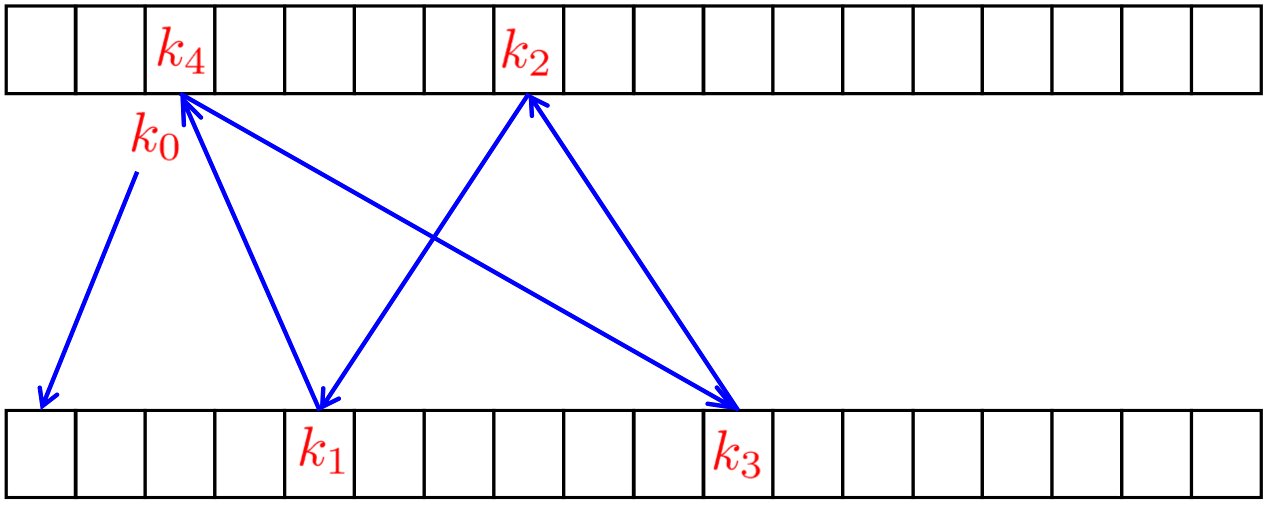
A more difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

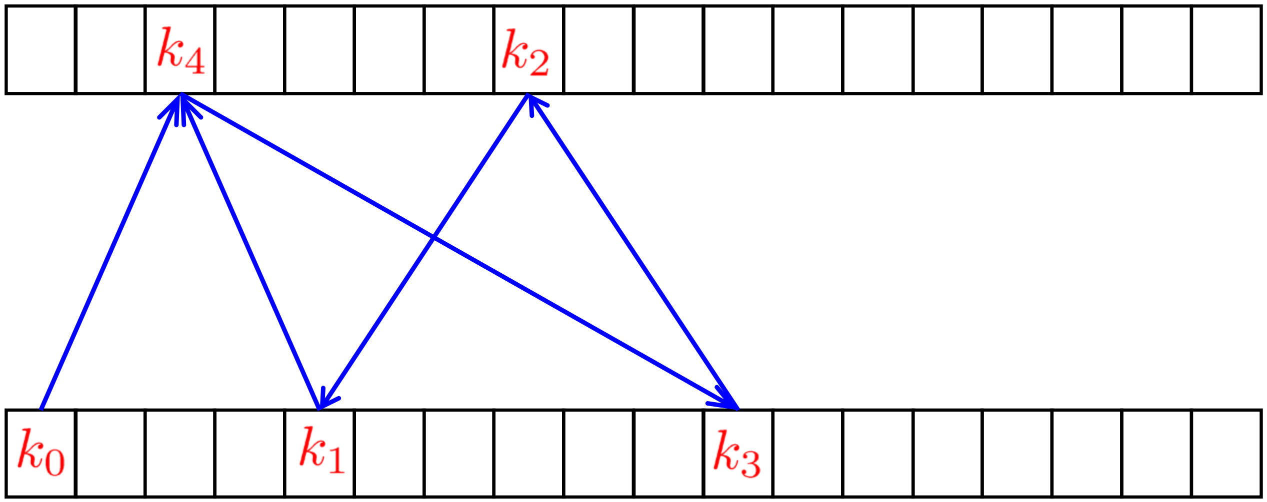
A more difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

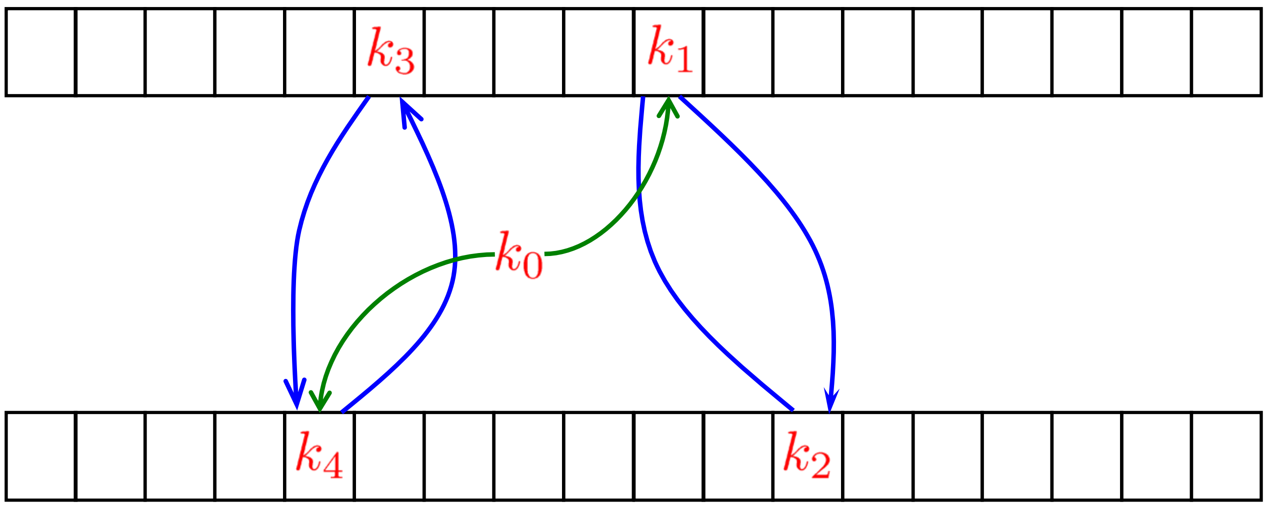
A more difficult insertion



Cuckoo Hashing

[Pagh-Rodler (2004)]

A failed insertion



If Insertion takes more than MAX steps, rehash

Cuckoo Hashing

[Pagh-Rodler (2004)]

Function `Cuckoo-Insert`(T, x)

---

for $i \leftarrow 1$ **to** MAX **do**
  $x \leftrightarrow T_1[h_1(x.key)]$
  if $x = null$ **then return**
  $x \leftrightarrow T_2[h_2(x.key)]$
  if $x = null$ **then return**

`Rehash(T)`

`Cucko-Insert`(T, x)

With hash functions chosen at random from an appropriate family of hash functions, the amortized expected insert time is O(1)