Lecture 4. Hash.

Dmitri Shmelkin

Math Modeling Competence Center at Moscow Research Center of Huawei
Shmelkin.Dmitri@huawei.com

March 3, 2023

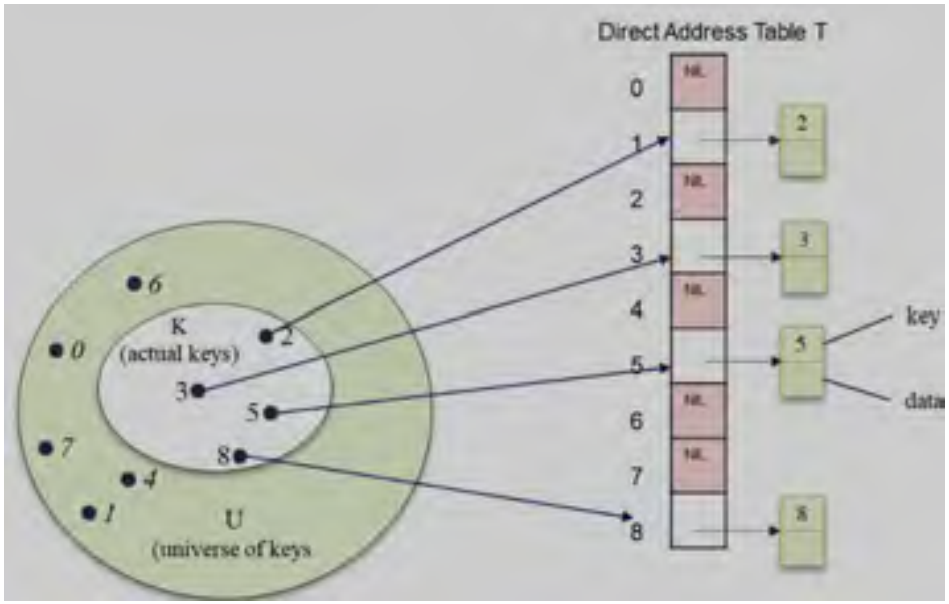
Overview

Recollection and the goal

• In our exploration of data structures to store data, we mentioned an **Array** that allows for logarithmic element query complexity, if data is sorted. The drawback is the linear penalty for insertion and deletion, which is critical for **dynamic** sets.
• We then introduced **Heap**, a less obvious structure, having logarithmic complexity for insertion and deletion, and constant compexity for query, but only if the maximum is queried.
• Despite the above query has this limitation, the Heap is useful in many cases, such as Dijkstra algorithm for shortest paths in graph.
• In this lecture we will consider another type of data structures, univerally called **Hash Tables**, with features close to ideal, if talking about general data.
• The examples of such structures occur every time, when one needs to deal with large dynamic set and is ready to access elements by some **key**.
• Everyone remembers, language **dictionary**, a heavy book that you had to use when reading a text, to search for a word.
• In fact, language dictionaries are just an array (by first letter) of ordered lists with words in their lexicographic order.
• How about a Wizard able to point to a string in $O(1)$?
• This wizard is called **Hash function**

Direct Address Table

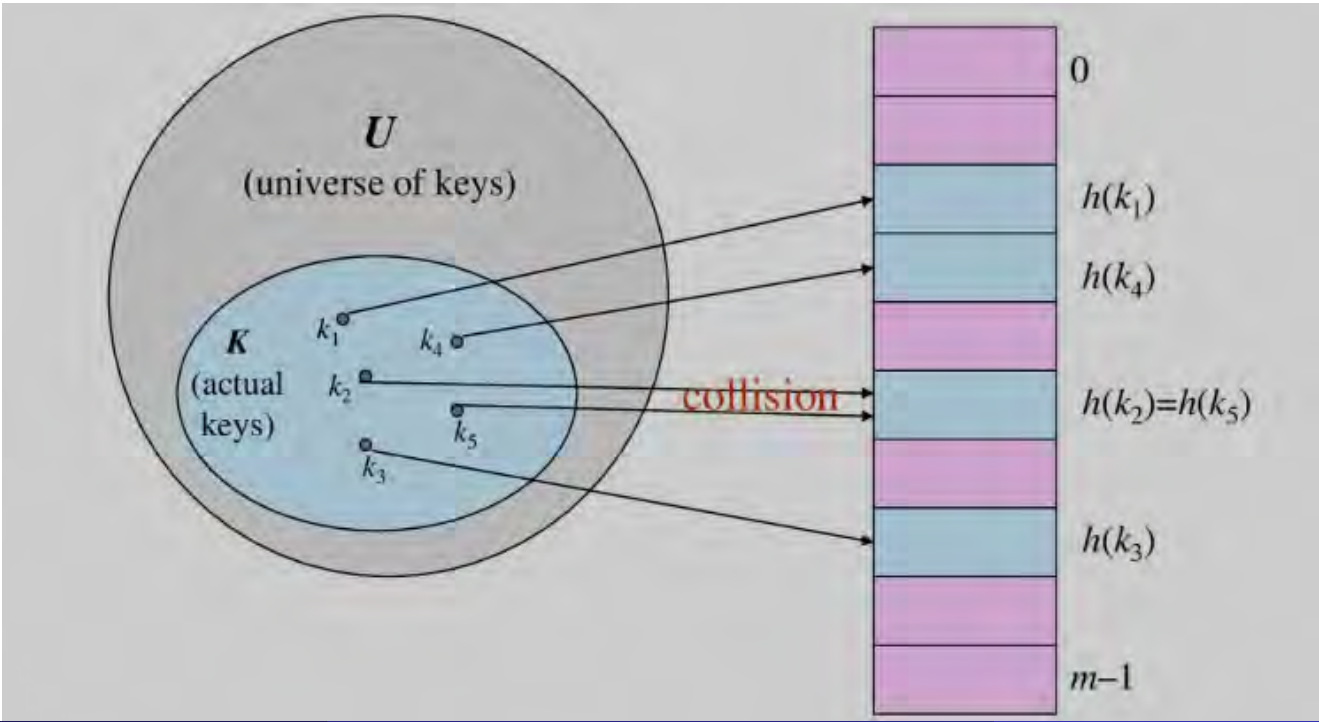
The simplest hash function is a tautological one. It is useful in the cases, where a dynamic set can be enabled with a key such that the set U of all key values is small enough



So the number of Table entries is that of the universe of keys and only those entries, which correspond to the actual keys will be used.

General Hash

• The disadvantage of Direct Address is obvious: the set of all key values is in many cases huge, and only a small part of keys is used.
• In such a general case, dynamic set is injectively mapped to a set, U, of all keys, then a hash function, $h : U \to [0, \ldots, m-1]$, maps the keys to an array.
• An element with key $k \in U$ is *hashed* to the position or *slot* $h(k)$ in the array.
• Reserve notation K for the image of dynamic set in U, that of actual keys.
• Hash functions are typically not injective, this is guranteed, if $|U| > m$. Cases when keys $k_1, k_2 \in U$ have the same hash, $h(k_1) = h(k_2)$ are called *collisions*.



Heuristics hash function

• It is clear that to be a good hash, a function should generate some homogeneous map, and avoid concentration of collisions in few slots.
• In many cases, the set U may be represented as a set of numbers. so that arithmetic operation can be used to define hash.
Division method $h(k) = k \bmod(m)$. This method will be homogeneous or not depending on the distribition of keys and selection of m.
• Taking $m = 2^q$ yields a very fast hash function: h just takes q last bits of k. But hence, h does not depend on all bits, which may lead to more collisions for specific key sets. Heuristics suggest to use m prime, and not close to 2^q numbers.
Multiplication method $h(k) = \lfloor m(\beta k \bmod(1)) \rfloor$.
• The idea is to, first map keys to $[0, 1)$ by taking the fractional part of the product βk, for some pre-selected non-integer number β. Then, using the fraction to get a corresponding part of the maximal hash, m.
• The advantage of this method is that we may use any m, including convenient $m = 2^q$. For the value of β people select what they like, including (inverse to) Golden Section $\beta = \frac{\sqrt{5}-1}{2}$, recommended by D.Knut.

Example: packet forwarding

• Internet is a global web of autonomous networks, in which IP packets flow from source to destination, passing across many IP routers.
• Each router forwards the packet arrived from some port to another port, according to the desitnation IP address and to the *Routing Table*
• In the simplest case, the set of all IP adress, $U, |U| = 2^{32} \sim 4 \cdot 10^9$ but even the most powerful routers support $10^5 - 10^6$ IP adresses, so do not use Direct Address Tables.
• Typically, the IP adress is represented as 8-bit decimal numbers delimited with point, e.g., 192.168.1.0 and this yields an idea of hash function:

$$h(x_1.\, x_2.\, x_3.\, x_4) = (a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4) \bmod(p), U \to \mathbb{Z}/p\mathbb{Z}, \qquad (1)$$

where $p > 255$ is a prime number and $a_1, \ldots, a_4 \in \{0, \ldots, p-1\}$.

IP addresses hash function property

Theorem 1

Let $x_1.x_2.x_3.x_4$ and $y_1.y_2.y_3.y_4$ be two different IP addresses. If the coefficients $a_1, \ldots, a_4$ are independent random variables, homogenously distributed in $\{0, \ldots, p-1\}$, then holds $Pr(h(x_1.x_2.x_3.x_4) = h(y_1.y_2.y_3.y_4)) = 1/p$.

Proof: Assume that $x_4 \neq y_4$. The event E, $h(x_1.x_2.x_3.x_4) = h(y_1.y_2.y_3.y_4)$ is equivalent to $\sum_{i=1}^{4} a_i x_i \equiv \sum_{i=1}^{4} a_i y_i \bmod(p)$. Let's compute $Pr(E|(a_1, a_2, a_3))$. Holds: $-a_4(y_4 - x_4) \equiv \sum_{i=1}^{3} a_i(y_i - x_i) \bmod(p)$. Since $y_4 - x_4 \neq 0$ and $|y_4 - x_4| \leq 255$ and $p > 255$ is prime, there is exactly 1 value of a_4, out of p possible, that solves the above equation. So we have, $Pr(E|(a_1, a_2, a_3)) = 1/p$ and the claim follows from the assumption of independence for a_1, a_2, a_3, a_4. □

- The property from Theorem 1 can be generalized and imply nice features of hash function, if selected at random.

Definition 2

For n keys hashed to a Table of size m, the ratio $\alpha = n/m$ is called *load*.

Universal Hash Families

Definition 3

A family $\mathcal{H}$ of function $U \to \{0, \ldots, m-1\}$ is called *universal* if for any $k_1 \neq k_2$ the number of functions $h \in \mathcal{H}$ such that $h(k_1) = h(k_2)$ is not more than $|\mathcal{H}|/m$.

Corollary 4

The family of hash functions on IP addresses introduced in (1) is universal.

Theorem 5

Let a hash function is selected at random in a universal family $\mathcal{H}$ to hash n keys to a table T of size m. For $k \in U$ let $C(k)$ be the random number of $j \in K$ such that $h(j) = h(k)$. If $k \notin K$, then $E(C(k)) \leq \alpha$. If $k \in K$, then $E(C(k)) < 1 + \alpha$.

Proof: For $k, j \in U$ let I_{kj} be the random value taking 1 iff $h(k) = h(j)$. By universality, $E(I_{kj}) \leq 1/m$. If $k \notin K$, then $C(k) = \sum_{j \in K} I_{kj} \Rightarrow E(C(k)) \leq n/m$. If $k \in K$, then $C(k) = 1 + \sum_{j \in K, j \neq k} I_{kj}$, $E(C(k)) \leq 1 + (n-1)/m < 1 + \alpha$. □

Builidng Universal Hash Families

Theorem 6

Let p be a prime number, $m < p$. For any $a, b \in \mathbb{Z}/p\mathbb{Z}, a \neq 0$, consider a map $\varphi_{a,b} : \mathbb{Z}/p\mathbb{Z} \to \mathbb{Z}/p\mathbb{Z}, q \to aq + b \bmod(p)$. Consider a map $\rho : \mathbb{Z}/p\mathbb{Z} \to \mathbb{Z}/m\mathbb{Z} : q \to q \bmod(m)$. Then the family $\mathcal{H} = \{h = \rho \circ \varphi_{a,b}\}$ is a universal family.

Proof: Each map $\varphi_{a,b}$ is invertible because $a \neq 0$ and p is prime. So it is a bijection and moreover $\Phi = \{\varphi_{a,b}\}$ is a group that acts transitively on the set $\{(k, l) \in (\mathbb{Z}/p\mathbb{Z})^2 | k \neq l\}$: for any $r \neq s$ and $k \neq l \in \mathbb{Z}/p\mathbb{Z}$, the system:

$$r \equiv ak + b \bmod(p);\ s \equiv al + b \bmod(p), \tag{2}$$

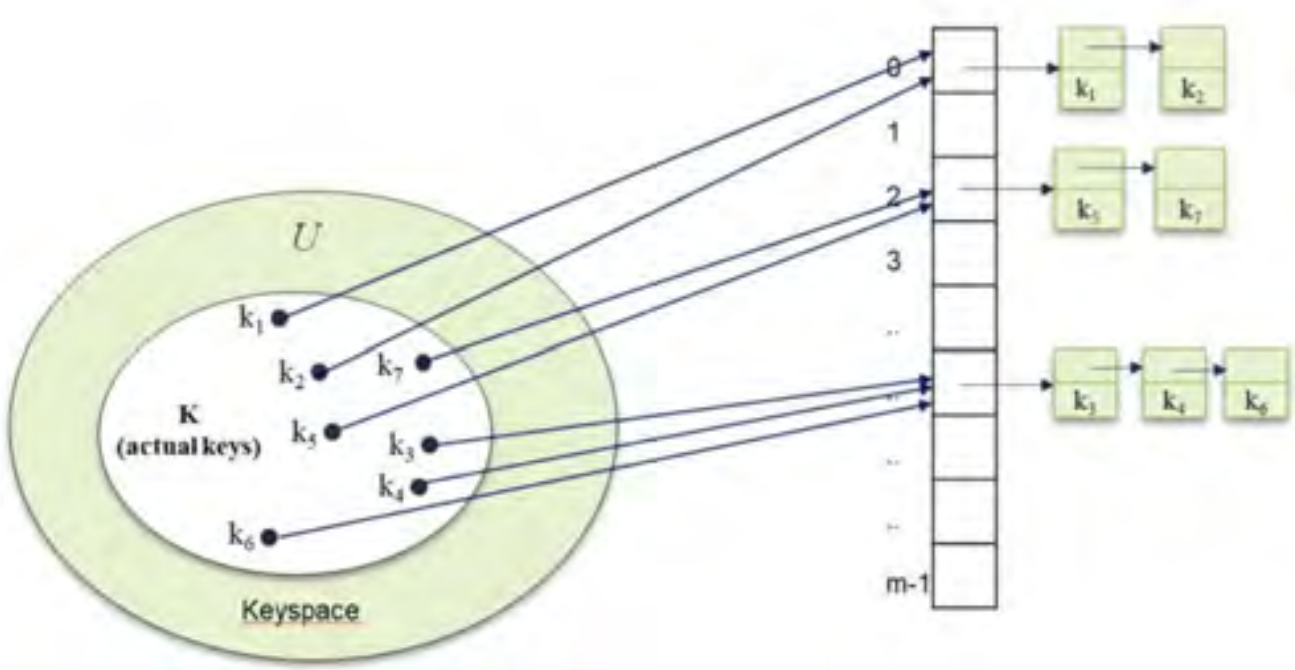
has a unique solution: $a = (r - s)(k - l)^{-1}$, $b = r - ak$.
Fix $k, l \in \mathbb{Z}/p\mathbb{Z}, k \neq l$. By the above transitivity, the pairs (a, b) such that $h(k) = h(l)$ correspond to the pairs (r, s) such that $r \neq s$ and $r \equiv s \bmod(m)$. For each of p possible values for r there are at most

$$\lceil p/m \rceil - 1 \leq (p + m - 1)/m - 1 = (p - 1)/m \tag{3}$$

values of s. Totally $\leq p(p - 1)/m$ pairs of r, s yield a collision of k and l.

Supporting collisions by chains

- Assume that a hash function is selected. How to support collisions in the hash?
- The first straightforward solution is to think of the hash slot as a starting position of a *chain* of keys having all the same hash value.
- The chain $T(h)$ can be implemented either as a singly or doubly linked list (to be introduced in Lecture 5) or as a dynamic array.



Chained hash methods and complexity

The implementation of ChainedHash methods is straighforward:
ChainedHashInsert(T, x) : $T(h(x.key)).pushback(x)$
ChainedHashSearch(T, k) : Run through $T(h(k))$ until $x.key = k$; return x;
If k is not found, return *NULL*;
ChainedHashDelete(T, k) : Run through $T(h(k))$ until $x.key = k$;
Delete x from $T(h(k))$; return x;
If k is not found, return *NULL*;

Theorem 7

Let a hash function is selected at random in a universal family, to hash n keys to Chained Hash with m slots, with load $\alpha = n/m$. Then Search has average complexity $O(\alpha)$ if key is not found and $O(1 + \alpha)$, if found. Delete has average complexity $O(\alpha)$. Insert has complexity $O(1)$.

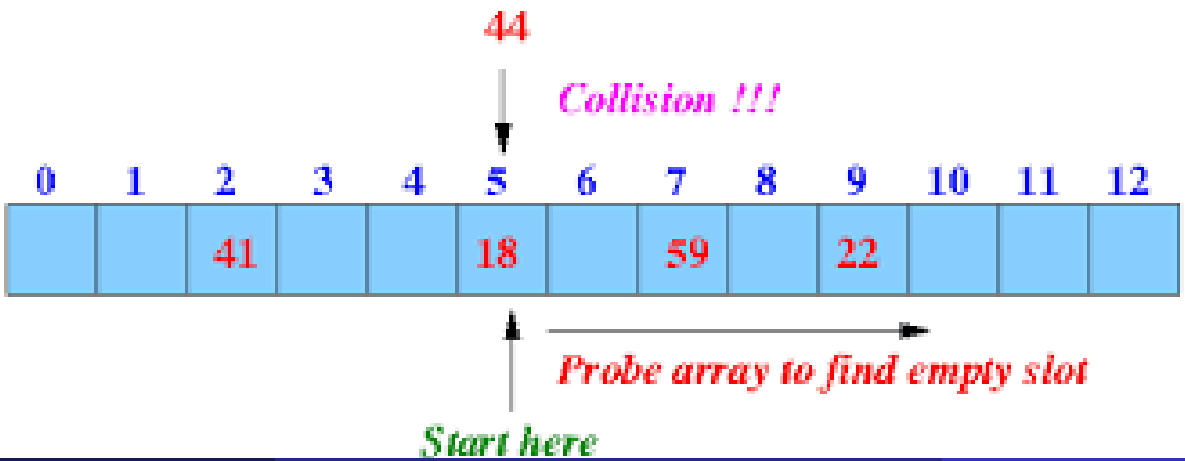
Proof: Follows from Theorem 5.

Closed Hashing

• Chained hash is natural. But it is not so elegant and has some drawbacks:
(1) some slots are empty and some others store data of different size.
(2) the size of table includes the auxillary fields like pointers for linked lists.
• The idea of *Closed Hashing* that is often called *Open Addressing* is to deploy all collided items in different slots, using some extended hash function:

$$h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}, \tag{4}$$

where the second parameter is the step of **probing** that happens if the previous step showed that the slot is occupied and, if this is *Search* algorithm, the key of previous slot is not as required.
• A requirement to h from (4) is that for every $k \in U$ the map $h(k, *)$ is a permutation of $\{0, \ldots, m-1\}$ so that every slot may be visited in probing.



Closed Hashing Methods

- Since every key occupies a separate slot the load of closed hash table, $\alpha \leq 1$.
- Hence, an overload needs to be prevented in *Insert* function.
- After a key has been deleted, and if the key was deployed with $step > 0$, after deletion, we need to put to the slot a special data *DEL* different from another special data, *NULL* used to put to empty slots, including when in initialization.

ClosedHashInsert: Input: a data x with key, $x.k$.

Output: the slot, where x is deployed or *ERROR* message, if overfull:

1. *for* $step = 0 : m - 1$
2. $j := h(x.k, step)$;
3. *if* $(T[j] = NULL$ *or* $T[j] = DEL)$
4. $T[j] := x$; *return* j
5. *endfor*
6. *return ERROR*//all slots are occupied

Remark 1. Note that we reoccupy the slots, where previously data was deleted.

2. If the table is almost full, we have a high probability to probe **every** slot.

Closed Hashing Methods: Search and Delete

ClosedHashSearch: Input: a key, k.
Output: data x with $x.key = k$, if found, $NULL$, otherwise:

1. *for* $step = 0 : m - 1$
2. $j := h(k, step)$;
3. *if* $(T[j] = NULL)$ *return* $NULL$
4. *if* $(T[j] = DEL)$ *continue*;
5. *if* $(T[j].key = k)$ *return* $T[j]$
6. *endfor*

ClosedHashDelete: Input: a slot, j, we need to delete: $T[j] = DEL$.

Remark 1. We might input a key to *Delete*, but anyway, we will start from searching the data with that key, and if found, delete. So the recommended sequence is, first get the slot by *Search*, then, *Delete* this slot.

2. What we delete is the record about data in the Table. If the object itself needs to be deleted, it might be deleted after getting by preliminary *Search*.

Probing methods for Closed Hashing

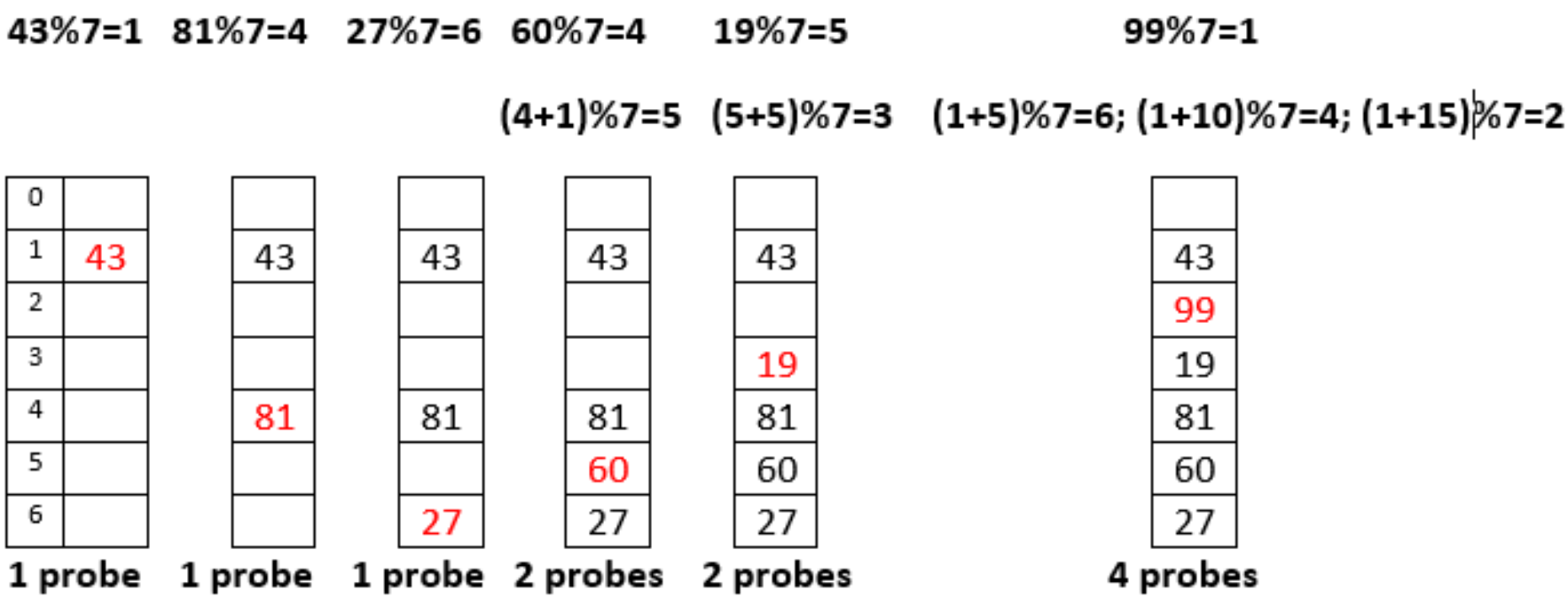
• We see that the concept of Closed Hashing works well, if an extended hash function as in (4) defined. Below we consider available approaches to this.
Linear Probing: $h(k, step) = h_1(k) + step \bmod(m)$, where h_1 is a hash function
• The linear probing obviously visits every slot. The issue is that linear probing tries to occupy consecutive slots, which is very far from homogeneous distribution.
• In particular, if $h(k_1), h(k_2)$ are close, a mutually negative impact occurs.
Double Hashing: $h(k, step) = h_1(k) + step \cdot h_2(k) \bmod(m)$
• Double hashing visits every slot iff every value of h_2 is coprime with m.
• The latter can be ensured by having either of 2 easily controlled features:
(1) m is prime, $0 < h_2(k) < m$ or **(2)** $m = 2^q$, $h_2(k)$ is odd.
Example of (1): $m = p, q < p$. $h_1(k) = k \bmod(p)$, $h_2(k) = 1 + (k \bmod(q))$

Example of Double Hashing

Consider the extened hash function to a table with $m = 7$ slots:

$$h(k, step) = [k + step \cdot (1 + k \bmod(5))] \bmod(7)$$

Let's apply hash function to the sequence of keys: 43, 81, 27, 60, 19, 99:



We see how the number of probes grows as load tends to 1.

Operation Complexity for Closed Hashing: Insert

- When studying Closed Hashing, the function $h(k, step)$ is assumed *homogenous*: *The permutation* $h(k, *)$ *takes every of* $n!$ *permutations with equal probability*
- The homogeneity assumption is obviously wrong for linear probing.

Theorem 8

For a homogeneous closed hashing and a table with load $n/m = \alpha$*, let* X *be the random number of probe steps that Insert needs. Then* $E(X) \leq 1/(1-\alpha)$*.*

Proof: *Insert* does probing $h(k, step)$, $step = 0, 1, \ldots$ until *NULL* or *DEL* is met. Consider the event $A_i, i = 0, 1, \ldots$ that happens when step i was done and got no *NULL*. Homogeneity implies:

$$Pr(A_0) = \frac{n}{m};\ Pr(A_1|A_0) = \frac{n-1}{m-1}; \ldots, Pr(A_t|A_0A_1 \ldots A_{t-1}) = \frac{n-t+1}{m-t+1} \quad (5)$$

$$Pr(X \geq t) = Pr(A_0A_1 \ldots A_{t-1}) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-t+2}{m-t+2} \leq (\frac{n}{m})^{t-1} = \alpha^{t-1}.$$

Then $E(X) = \sum_{t=0}^{\infty} tPr(X = t) = \sum_{t=1}^{\infty} Pr(X \geq t) \leq \sum_{i=0}^{\infty} \alpha^t = \frac{1}{1-\alpha}$ □

Search Complexity for Closed Hashing: Failed Search

Corollary 9

For a homogeneous closed hashing and a table with load n/m and d slots marked as DEL, set $\beta = (n+d)/m$ and let Y be the random number of probe steps leading to **failed** *Search. Then $E(Y) \leq 1/(1-\beta)$.*

Proof: Let Z be the random number of probe steps needed for *Insert* assuming for a while that *DEL* slots aren't available to occupy. By Theorem 8, $E(Z) \leq 1/(1-\beta)$. On the other hand, a failed search does the same steps or stops earlier, if the key is found. So $E(Y) \leq E(Z)$. □

Search Complexity for Closed Hashing: Successful Search

Theorem 10

For a homogeneous closed hashing and a table with load $n/m = \alpha$, and **no deletion happened**, *let X be a random value of probe steps before finding data. Then $E(X) \leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.*

Proof: By Corollary 8, and because no deletion happened, the insertion of $i+1$-th key took $\leq 1/(1-i/m) = m/(m-i)$ steps, in average. The same number of steps is needed to find this element. So $E(X)$ is less than or equal to the average:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{t=m-n+1}^{m}\frac{1}{t} \leq \frac{1}{\alpha}\int_{t=m-n}^{m}\frac{dt}{t} = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}. \quad (6)$$

Dynamic Updates of Closed Hash Table

• Theorems 7,8,9,10 imply that all operations with Hash Tables (well, under some ideality assumptions) have $O(1)$ complexity in average, if the load, α is also $O(1)$, that is limited by a constant.
• To have load limited for a dynamic set of arbitrary size n, a heuristics may be used, named **Dynamic Hash Table**, similar to that for dynamic arrays:
1. Initialize a Hash Table of size $m_0 = 2^{q_0}$ and use your favorite hashing, Chained or Closed, e.g., Double Hashing with feature **(2)** (see above).
2. Keep adding, searching, and removing until the load reaches $\alpha = 0.5$.
3. When $\alpha = 0.5$ re-hash data to a table of size $2m$ and release previous table.

Theorem 11

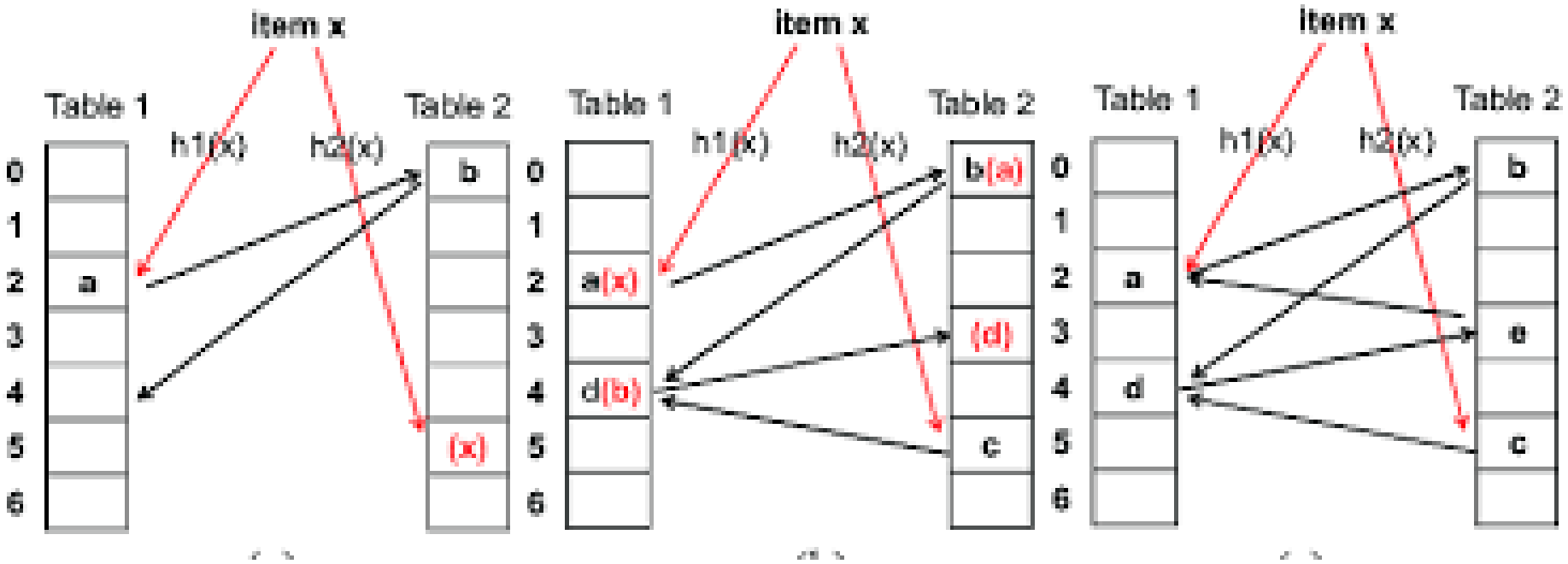
The above Dynamic Hash Table has $O(1)$ compexity for Search, Insert, and Delete operations, in average.

Proof is analogous to that for Dynamic Arrays.

Cuckoo Hashing

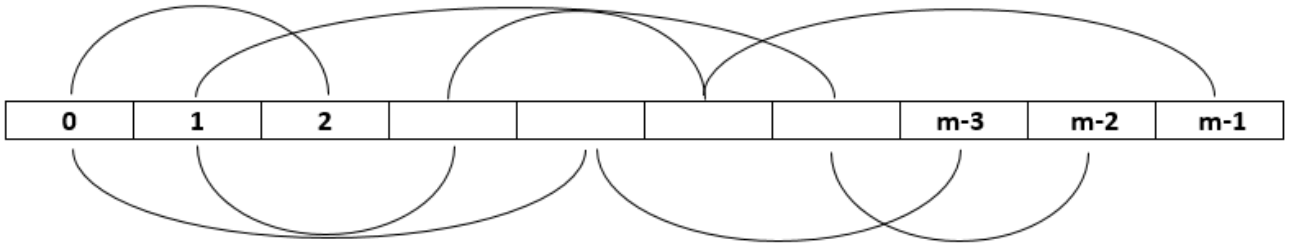
• Since 1950-s, when Hash Tables were invented, the progress didn't stop.
• Particularly known is so-called **Bloom filter** invented in [Bloom 1970] as a probabilistic data structure that supports the query of whether a key is in the Table and may get false positive but not false negative answers.
• The progress didn't stop in 21th century and in [Pagh-Rodler 2001] a surprising new idea has been proposed that leads to a worst-case $O(1)$ query time while having $O(1)$ in average *Insert* time, based on **Cuckoo** principle.
• As in Double Hashing, one uses two Hash functions, $h_0, h_1 : U \to \{0, \ldots, m-1\}$ and every key, k, is to be placed either to $h_0(k)$ or to $h_1(k)$. If able to guarantee this, *Search* checks exactly 1 or 2 slots, so $O(1)$, and same for *Delete*.
• When doing *Insert*(x), several variations of the Cuckoo principle apply. For example to check $h_0(x_0), h_1(x_0), x_0 = x$ and if both are occupied, remove x_1 such that $h_e(x_1) = h_0(x_0)$, then move x_1 to $h_{1-e}(x_1)$ and then apply the same cuckoo principle to $x_2, \ldots$ until either of two results obtained:
1. $x_0, x_1, \ldots$ are successfully placed in different slots or
2. the process commits an infinite loop
If case **2** happens, the pair (h_0, h_1) is to be replaced with another pair, chosen at random, and all stored elements rehashed, as explained above.

Cuckoo Hashing Example



Cuckoo Hashing: reasons to work

• For a pair of hash functions, (h_0, h_1), and a set $K \subseteq U$, let Γ_K be the graph with nodes $\{0, \ldots, m-1\}$ and $|K|$ arcs connecting $h_0(k)$ with $h_1(k)$:



Theorem 12

1. *If each $k \in K$ is placed in different slots such that every key is at either $h_0(k)$ or $h_1(k)$, then every connected component of Γ_K has no more arcs than nodes.*
2. *When adding a key, k, an infinite loop occurs if and only if at least one of connected components of $\Gamma_{K\cup\{k\}}$ has more arcs than nodes.*

Sketch of the Proof: Part **1** and "if" of part **2** are obvious. Let Δ be the connected comonent of $\Gamma_{K\cup\{k\}}$ containing both $h_0(k)$ and $h_1(k)$. Assume that Δ has q nodes and either $q-1$ or q arcs. In both cases the q nodes either

Cuckoo Hashing: reasons to work

constitute a subtree (cycle free connected component) in Γ_K or two connected components out of which one is a subtree and another is either a subtree or has as many arcs as nodes. In all cases the propagation of the cuckoo process over a subtree provably leads to getting a distinct slot for every key. □

- Theorem 12 implies that supporting connected components of Γ_K would be enough to prevent infinite loops and timely call rehashing. In practice, it is simpler just to hard code the upper bound for the cuckoo loops before rehashing.
- [Pagh-Rodler 2001] showed that *Insert* requires $O(1)$ complexity in average.
- The idea of rehashing as allowing to ensure Γ_K has the necessary features of connected components is supported in [Tromp 2014].

Theoretical support of cuckoo hashing and other heuristics is still in progress.

References


Bloom, Burton H., *Space/Time Trade-offs in Hash Coding with Allowable Errors*, *Communications of the ACM*, **13** (7), 1970: 422–426


Pagh, Rasmus and Rodler, Flemming F: *Cuckoo Hashing* in "Algorithms — ESA 2001". Lecture Notes in Computer Science. **2161**, 2001.


John Tromp: *Cuckoo Cycle: a memory bound graph-theoretic proof-of-work*, https://eprint.iacr.org/2014/059.pdf