The Power of Tabulation Hashing

Mihai Pătraşcu and Mikkel Thorup

at&t

The Power of Tabulation Hashing

Mihai Pătraşcu and Mikkel Thorup

Thank you for inviting me to ThRaSH.

The Power of Tabulation Hashing

Mihai Pătraşcu and Mikkel Thorup

Thank you for inviting me to ThRaSH.

Some of this work can be found in Proc. STOC'11.

Target

- Safe and simple hashing.

Target

- Safe and simple hashing.
- Guarantees akin to those of truely random hashing, yet easy to implement.

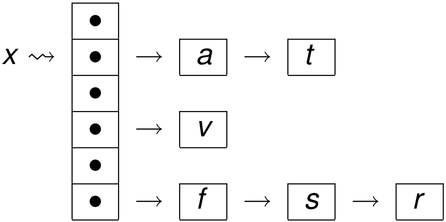
Target

- Safe and simple hashing.
- Guarantees akin to those of truely random hashing, yet easy to implement.
- Uniting theory and practice.

Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

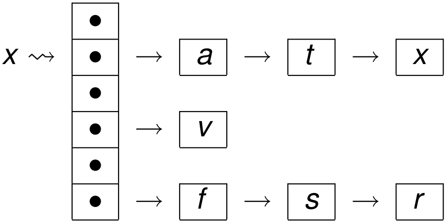
- chaining: follow pointers



Applications of Hashing ⇝

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

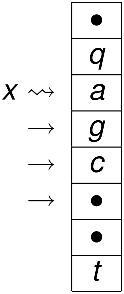
- chaining: follow pointers



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

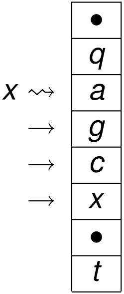
- chaining: follow pointers
- linear probing: sequential search in *one* array



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

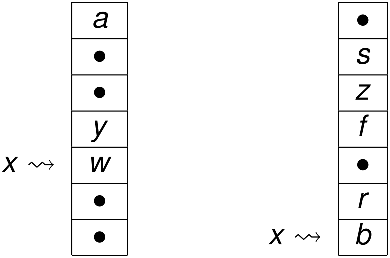
- chaining: follow pointers
- linear probing: sequential search in *one* array



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

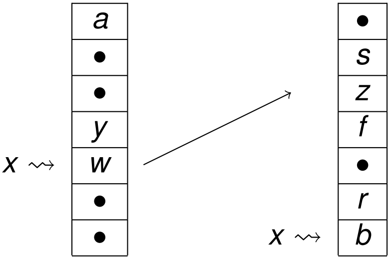
- chaining: follow pointers
- linear probing: sequential search in *one* array
- cuckoo hashing: search $\leq$ 2 locations, complex updates



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

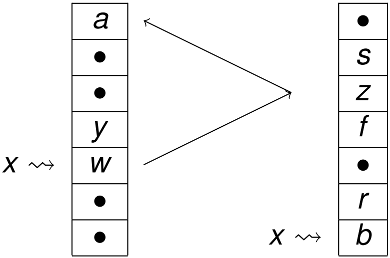
- chaining: follow pointers
- linear probing: sequential search in *one* array
- cuckoo hashing: search $\leq$ 2 locations, complex updates



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

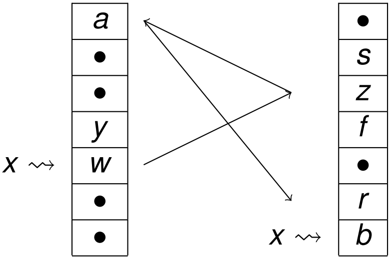
- chaining: follow pointers
- linear probing: sequential search in *one* array
- cuckoo hashing: search $\leq$ 2 locations, complex updates



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

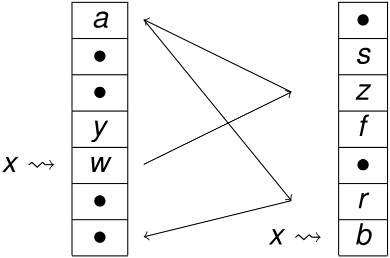
- chaining: follow pointers
- linear probing: sequential search in *one* array
- cuckoo hashing: search ≤ 2 locations, complex updates



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

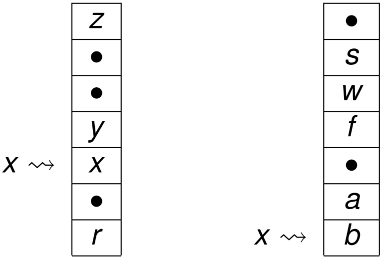
- chaining: follow pointers
- linear probing: sequential search in *one* array
- cuckoo hashing: search ≤ 2 locations, complex updates



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

- chaining: follow pointers
- linear probing: sequential search in *one* array
- cuckoo hashing: search ≤ 2 locations, complex updates



Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

- chaining: follow pointers.
- linear probing: sequential search in *one* array
- cuckoo hashing: search ≤ 2 locations, complex updates

Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

- chaining: follow pointers.
- linear probing: sequential search in *one* array
- cuckoo hashing: search ≤ 2 locations, complex updates

Sketching, streaming, and sampling:

- moment estimation: $F_2(\bar{x}) = \sum_i x_i^2$

Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

- chaining: follow pointers.
- linear probing: sequential search in *one* array
- cuckoo hashing: search ≤ 2 locations, complex updates

Sketching, streaming, and sampling:

- moment estimation: $F_2(\bar{x}) = \sum_i x_i^2$
- sketch A and B to later find $|A \cap B| / |A \cup B|$

$$|A \cap B| / |A \cup B| = \Pr_h[\min h(A) = \min h(B)]$$

We need h to be ε-minwise independent:

$$(\forall) x \notin S : \qquad \Pr[h(x) < \min h(S)] = \frac{1 \pm \varepsilon}{|S| + 1}$$

Applications of Hashing $\rightsquigarrow$

Hash tables (n keys and $2n$ hashes: expect 1/2 keys per hash)

- chaining: follow pointers.
- linear probing: sequential search in *one* array

Important outside theory. These simple practical hash tables often bottlenecks in the processing of data—substantial fraction of worlds computational resources spent here.

Carter & Wegman (1977)

We do not have space for truly random hash functions, but

A family $\mathcal{H} = \{h : [u] \rightarrow [b]\}$ is *k-independent* iff:

- $(\forall) x \in u$, $h(x)$ is uniform in $[b]$;
- $(\forall) x_1, \ldots, x_k \in [u]$, $h(x_1), \ldots, h(x_k)$ are independent.

Carter & Wegman (1977)

We do not have space for truly random hash functions, but

A family $\mathcal{H} = \{h : [u] \to [b]\}$ is k-independent iff:

- $(\forall) x \in u$, $h(x)$ is uniform in $[b]$;
- $(\forall) x_1, \ldots, x_k \in [u]$, $h(x_1), \ldots, h(x_k)$ are independent.

Prototypical example: degree $k - 1$ polynomial

- u prime;
- choose $a_0, a_1, \ldots, a_{k-1}$ randomly in $[u]$;
- $h(x) = \left(a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}\right) \bmod u$.

Carter & Wegman (1977)

We do not have space for truly random hash functions, but

A family $\mathcal{H} = \{h : [u] \to [b]\}$ is k-independent iff:

- $(\forall)x \in u$, $h(x)$ is uniform in $[b]$;
- $(\forall)x_1, \ldots, x_k \in [u]$, $h(x_1), \ldots, h(x_k)$ are independent.

Prototypical example: degree $k - 1$ polynomial

- u prime;
- choose $a_0, a_1, \ldots, a_{k-1}$ randomly in $[u]$;
- $h(x) = \left(a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}\right) \bmod u$.

Many solutions for k-independent hashing proposed, but generally slow for $k > 3$ and too slow for $k > 5$.

How much independence needed?

Chaining $\mathbf{E}[t] = O(1)$ $\mathbf{E}[t^k] = O(1)$ $t = O\left(\frac{\lg n}{\lg \lg n}\right)$ w.h.p.	2 $2k+1$ $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$	
Linear probing	≤ 5 [Pagh^2, Ružić'07]	≥ 5 **[PT ICALP'10]**
Cuckoo hashing	$O(\lg n)$	≥ 6 [Cohen, Kane'05]
F_2 estimation	4 [Alon, Mathias, Szegedy'99]	
ε-minwise indep.	$O\left(\lg \frac{1}{\varepsilon}\right)$ [Indyk'99]	$\Omega\left(\lg \frac{1}{\varepsilon}\right)$ **[PT ICALP'10]**

How much independence needed?

Chaining $\mathbf{E}[t] = O(1)$ $\mathbf{E}[t^k] = O(1)$ $t = O\left(\frac{\lg n}{\lg \lg n}\right)$ w.h.p.	2 $2k+1$ $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$	
Linear probing	≤ 5 [Pagh^2, Ružić'07]	≥ 5 **[PT ICALP'10]**
Cuckoo hashing	$O(\lg n)$	≥ 6 [Cohen, Kane'05]
F_2 estimation	4 [Alon, Mathias, Szegedy'99]	
ε-minwise indep.	$O\left(\lg \frac{1}{\varepsilon}\right)$ [Indyk'99]	$\Omega\left(\lg \frac{1}{\varepsilon}\right)$ **[PT ICALP'10]**

Independence has been the ruling measure for quality of hash functions for 30+ years, but is it right?

Simple tabulation

- Simple tabulation goes back to Carter and Wegman'77.

Simple tabulation

- Simple tabulation goes back to Carter and Wegman'77.
- Key x divided into $c = O(1)$ characters $x_1, ..., x_c$, e.g., 32-bit key as $4 \times$ 8-bit characters.

Simple tabulation

- Simple tabulation goes back to Carter and Wegman'77.
- Key x divided into $c = O(1)$ characters $x_1, ..., x_c$, e.g., 32-bit key as 4×8-bit characters.
- Hash value

$$h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]$$

where the R_i are independent random tables:
char $\rightarrow$ hash values (bit strings)

Simple tabulation

- Simple tabulation goes back to Carter and Wegman'77.
- Key x divided into $c = O(1)$ characters $x_1, ..., x_c$, e.g., 32-bit key as $4 \times$ 8-bit characters.
- Hash value

$$h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]$$

  where the R_i are independent random tables:
  char $\rightarrow$ hash values (bit strings)
- With 8-bit characters, each table R_i has 256 entries and fit in fast memory.

Simple tabulation

- Simple tabulation goes back to Carter and Wegman'77.
- Key x divided into $c = O(1)$ characters $x_1, ..., x_c$, e.g., 32-bit key as $4 \times$ 8-bit characters.
- Hash value

$$h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]$$

  where the R_i are independent random tables:
  char $\rightarrow$ hash values (bit strings)
- With 8-bit characters, each table R_i has 256 entries and fit in fast memory.
- Simple tabulation is the fastest 3-independent hashing scheme.

Simple tabulation

- Simple tabulation goes back to Carter and Wegman'77.
- Key x divided into $c = O(1)$ characters $x_1, ..., x_c$, e.g., 32-bit key as $4 \times$ 8-bit characters.
- Hash value

$$h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]$$

  where the R_i are independent random tables:
  char $\rightarrow$ hash values (bit strings)
- With 8-bit characters, each table R_i has 256 entries and fit in fast memory.
- Simple tabulation is the fastest 3-independent hashing scheme.
- Not 4-independent: $h(a_1 a_2) \oplus h(a_1 b_2) \oplus h(b_1 a_2) \oplus h(b_1 b_2)$

$$\begin{aligned} = \quad & (R_1[a_1] \oplus R_2[a_2]) \oplus (R_1[a_1] \oplus R_2[b_2]) \oplus \\ & (R_1[b_1] \oplus R_2[a_2]) \oplus (R_1[b_1] \oplus R_2[b_2]) \; = \; 0. \end{aligned}$$

How much independence needed? Wrong question

Chaining $\mathbf{E}[t] = O(1)$ $\mathbf{E}[t^k] = O(1)$ $t = O(\frac{\lg n}{\lg \lg n})$ w.h.p.	2 $2k+1$ $\Theta(\frac{\lg n}{\lg \lg n})$	
Linear probing	≤ 5 [Pagh^2, Ružić'07]	≥ 5 **[PT ICALP'10]**
Cuckoo hashing	$O(\lg n)$	≥ 6 [Cohen, Kane'05]
F_2 estimation	4 [Alon, Mathias, Szegedy'99]	
ε-minwise indep.	$O(\lg \frac{1}{\varepsilon})$ [Indyk'99]	$\Omega(\lg \frac{1}{\varepsilon})$ **[PT ICALP'10]**

How much independence needed? Wrong question

Chaining $\mathbf{E}[t] = O(1)$ $\mathbf{E}[t^k] = O(1)$ $t = O(\frac{\lg n}{\lg \lg n})$ w.h.p.	2 $2k+1$ $\Theta(\frac{\lg n}{\lg \lg n})$	
Linear probing	≤ 5 [Pagh[2], Ružić'07]	≥ 5 **[PT ICALP'10]**
Cuckoo hashing	$O(\lg n)$	≥ 6 [Cohen, Kane'05]
F_2 estimation	4 [Alon, Mathias, Szegedy'99]	
ε-minwise indep.	$O(\lg \frac{1}{\varepsilon})$ [Indyk'99]	$\Omega(\lg \frac{1}{\varepsilon})$ **[PT ICALP'10]**

New result: Despite its 4-dependence, simple tabulation suffices for all the above applications:

One simple and fast hashing scheme for almost all your needs.

How much independence needed? Wrong question

Chaining $\mathbf{E}[t] = O(1)$ $\mathbf{E}[t^k] = O(1)$ $t = O(\frac{\lg n}{\lg \lg n})$ w.h.p.	2 $2k+1$ $\Theta(\frac{\lg n}{\lg \lg n})$	
Linear probing	≤ 5 [Pagh², Ružić'07]	≥ 5 **[PT ICALP'10]**
Cuckoo hashing	$O(\lg n)$	≥ 6 [Cohen, Kane'05]
F_2 estimation	4 [Alon, Mathias, Szegedy'99]	
ε-minwise indep.	$O(\lg \frac{1}{\varepsilon})$ [Indyk'99]	$\Omega(\lg \frac{1}{\varepsilon})$ **[PT ICALP'10]**

New result: Despite its 4-dependence, simple tabulation suffices for all the above applications:

> *One simple and fast hashing scheme for almost all your needs.*

Knuth recommends simple tabulation but cites only 3-independence as mathematical quality.

How much independence needed? Wrong question

Chaining $\mathbf{E}[t] = O(1)$ $\mathbf{E}[t^k] = O(1)$ $t = O(\frac{\lg n}{\lg \lg n})$ w.h.p.	2 $2k+1$ $\Theta(\frac{\lg n}{\lg \lg n})$	
Linear probing	≤ 5 [Pagh[2], Ružić'07]	≥ 5 **[PT ICALP'10]**
Cuckoo hashing	$O(\lg n)$	≥ 6 [Cohen, Kane'05]
F_2 estimation	4 [Alon, Mathias, Szegedy'99]	
ε-minwise indep.	$O(\lg \frac{1}{\varepsilon})$ [Indyk'99]	$\Omega(\lg \frac{1}{\varepsilon})$ **[PT ICALP'10]**

New result: Despite its 4-dependence, simple tabulation suffices for all the above applications:

> *One simple and fast hashing scheme for almost all your needs.*

Knuth recommends simple tabulation but cites only 3-independence as mathematical quality.
We prove that dependence of simple tabulation is not harmful in any of the above applications.

Chaining/hashing into bins

Theorem Consider hashing n balls into $m \geq n^{1-1/(2c)}$ bins by simple tabulation. Let q be an additional *query ball*, and define X_q as the number of regular balls that hash into a bin chosen as a function of $h(q)$. Let $\mu = \mathbf{E}[X_q] = \frac{n}{m}$. The following probability bounds hold for any constant γ:

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\Omega(\mu)} + m^{-\gamma}$$

$$\Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\Omega(\mu)} + m^{-\gamma}$$

With $m \leq n$ bins, every bin gets

$$n/m \pm O\left(\sqrt{n/m}\log^c n\right).$$

keys with probability $1 - n^{-\gamma}$.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Nothing like this lemma holds if we instead of simple tabulation assumed k-independent hashing with $k = O(1)$.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.
Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.
Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.
Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.
Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.
Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently.

- Let i be character position where keys in T differ.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently.

- Let i be character position where keys in T differ.
- Let a be least common character in position i and pick $x \in T$ with $x_i = a$

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently.

- Let i be character position where keys in T differ.
- Let a be least common character in position i and pick $x \in T$ with $x_i = a$
- Reduce T to T' removing all keys y from T with $y_i = a$.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.
Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.
Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently.

- Let i be character position where keys in T differ.
- Let a be least common character in position i and pick $x \in T$ with $x_i = a$
- Reduce T to T' removing all keys y from T with $y_i = a$.
- The hash of x is independent of the hash of T' as only $h(x)$ depends on $R_i[a]$.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently.

- Let i be character position where keys in T differ.
- Let a be least common character in position i and pick $x \in T$ with $x_i = a$
- Reduce T to T' removing all keys y from T with $y_i = a$.
- The hash of x is independent of the hash of T' as only $h(x)$ depends on $R_i[a]$.
- Return $\{x\} \cup U'$ where U' independent subset of T'.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.
Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.
Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently—if $|T| \geq d$ then $|U| \geq (1+\gamma)/\varepsilon$. $\square$

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently—if $|T| \geq d$ then $|U| \geq (1+\gamma)/\varepsilon$. $\square$

Claim 2 The probability that there exists $u = (1+\gamma)/\varepsilon$ keys hashing independently to the same bin is $m^{-\gamma}$.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently—if $|T| \geq d$ then $|U| \geq (1+\gamma)/\varepsilon$. □

Claim 2 The probability that there exists $u = (1+\gamma)/\varepsilon$ keys hashing independently to the same bin is $m^{-\gamma}$.

- There are $\binom{n}{u} < n^u$ sets U of u keys to consider.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently—if $|T| \geq d$ then $|U| \geq (1+\gamma)/\varepsilon$. $\square$

Claim 2 The probability that there exists $u = (1+\gamma)/\varepsilon$ keys hashing independently to the same bin is $m^{-\gamma}$.

- There are $\binom{n}{u} < n^u$ sets U of u keys to consider.
- By independence, U hash to *one* bin with probability m^{u-1}.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.

Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.

Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently—if $|T| \geq d$ then $|U| \geq (1+\gamma)/\varepsilon$. $\square$

Claim 2 The probability that there exists $u = (1+\gamma)/\varepsilon$ keys hashing independently to the same bin is $m^{-\gamma}$.

- There are $\binom{n}{u} < n^u$ sets U of u keys to consider.
- By independence, U hash to *one* bin with probability m^{u-1}.
- Recall $n \leq m^{1-\varepsilon}$.

Hashing into many bins

Lemma If we hash n keys into $n^{1+\Omega(1)}$ bins, then all bins get $O(1)$ keys w.h.p.
Proof that for any positive constants ε, γ, if we hash n keys into m bins and $n \leq m^{1-\varepsilon}$, then all bins get less than $d = 2^{(1+\gamma)/\varepsilon}$ keys with probability $\geq 1 - m^{-\gamma}$.
Claim 1 Any set T contains a subset U of $\log_2 |T|$ keys that hash independently—if $|T| \geq d$ then $|U| \geq (1+\gamma)/\varepsilon$. □
Claim 2 The probability that there exists $u = (1+\gamma)/\varepsilon$ keys hashing independently to the same bin is $m^{-\gamma}$.□

Short of time?

Basic proof pattern with $m \geq n^{1-1/(2c)}$ bins

Basic proof pattern with $m \geq n^{1-1/(2c)}$ bins

- Deterministic partition key set S into groups G that are mutually "independent", each of size $\leq n^{1-1/c} \leq m^{1-\varepsilon}$.

Basic proof pattern with $m \geq n^{1-1/(2c)}$ bins

- Deterministic partition key set S into groups G that are mutually "independent", each of size $\leq n^{1-1/c} \leq m^{1-\varepsilon}$.
- By lemma, w.h.p., each G distributes with $\leq d$ in each bin.

Basic proof pattern with $m \geq n^{1-1/(2c)}$ bins

- Deterministic partition key set S into groups G that are mutually "independent", each of size $\leq n^{1-1/c} \leq m^{1-\varepsilon}$.
- By lemma, w.h.p., each G distributes with $\leq d$ in each bin.
- Let $X_G \leq d$ be contribution to fixed bin, and $X = \sum_G X_G$.

Basic proof pattern with $m \geq n^{1-1/(2c)}$ bins

- Deterministic partition key set S into groups G that are mutually "independent", each of size $\leq n^{1-1/c} \leq m^{1-\varepsilon}$.
- By lemma, w.h.p., each G distributes with $\leq d$ in each bin.
- Let $X_G \leq d$ be contribution to fixed bin, and $X = \sum_G X_G$.
- If the X_G were really independent, by Chernoff

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu/d}$$

$$\Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu/d}$$

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.

Recursive partition into “independent” groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$.

- For each position $i \in [c]$, we have $< n^{1/c}$ characters used by $> n^{1-1/c}$ keys.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$.

- For each position $i \in [c]$, we have $< n^{1/c}$ characters used by $> n^{1-1/c}$ keys.
- So claim false implies S in hypercube of size $< \left(n^{1/c}\right)^c = n$.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$

Recursive partition into “independent” groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a).

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$
- the hash of keys in $G_{(i,a)}$ except $R_i[a]$ which is a common "shift" moving bin h to $h \oplus R_i[a]$.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$
- the hash of keys in $G_{(i,a)}$ except $R_i[a]$ which is a common "shift" moving bin h to $h \oplus R_i[a]$.
- Particularly, it is fixed which keys from $G_{(i,a)}$ end in same bin. By Lemma, w.h.p., at most d in every bin.

Recursive partition into “independent” groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$
- the hash of keys in $G_{(i,a)}$ except $R_i[a]$ which is a common “shift” moving bin h to $h \oplus R_i[a]$.
- Particularly, it is fixed which keys from $G_{(i,a)}$ end in same bin. By Lemma, w.h.p., at most d in every bin.

Now we randomly pick $R_i[a]$ finalizing hashing of group $G_{(i,a)}$.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$
- the hash of keys in $G_{(i,a)}$ except $R_i[a]$ which is a common "shift" moving bin h to $h \oplus R_i[a]$.
- Particularly, it is fixed which keys from $G_{(i,a)}$ end in same bin. By Lemma, w.h.p., at most d in every bin.

Now we randomly pick $R_i[a]$ finalizing hashing of group $G_{(i,a)}$.

- The contribution $X_{G_{(i,a)}}$ to our bin is random variable.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$
- the hash of keys in $G_{(i,a)}$ except $R_i[a]$ which is a common "shift" moving bin h to $h \oplus R_i[a]$.
- Particularly, it is fixed which keys from $G_{(i,a)}$ end in same bin. By Lemma, w.h.p., at most d in every bin.

Now we randomly pick $R_i[a]$ finalizing hashing of group $G_{(i,a)}$.

- The contribution $X_{G_{(i,a)}}$ to our bin is random variable.
- The distribution of $X_{G_{(i,a)}}$ depends on previous fixings.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$
- the hash of keys in $G_{(i,a)}$ except $R_i[a]$ which is a common "shift" moving bin h to $h \oplus R_i[a]$.
- Particularly, it is fixed which keys from $G_{(i,a)}$ end in same bin. By Lemma, w.h.p., at most d in every bin.

Now we randomly pick $R_i[a]$ finalizing hashing of group $G_{(i,a)}$.

- The contribution $X_{G_{(i,a)}}$ to our bin is random variable.
- The distribution of $X_{G_{(i,a)}}$ depends on previous fixings.
- But always $\mathbf{E}[X_{G_{(i,a)}}] = |X_{G_{(i,a)}}|/m$. Moreover $X_{G_{(i,a)}} \leq d$.

Recursive partition into "independent" groups

Define position character (i, a) in key x iff $x_i = a$.
Let (i, a) be least common position character among keys in S and $G_{(i,a)} \subseteq S$ be the group of keys using it.
Claim $|G_{(i,a)}| \leq n^{1-1/c}$. $\square$
Recursively, we group $S \setminus G_{(i,a)}$ and hash all position characters in S excluding (i, a). This fixes

- the hash of all keys in $S \setminus G_{(i,a)}$
- the hash of keys in $G_{(i,a)}$ except $R_i[a]$ which is a common "shift" moving bin h to $h \oplus R_i[a]$.
- Particularly, it is fixed which keys from $G_{(i,a)}$ end in same bin. By Lemma, w.h.p., at most d in every bin.

Now we randomly pick $R_i[a]$ finalizing hashing of group $G_{(i,a)}$.

- The contribution $X_{G_{(i,a)}}$ to our bin is random variable.
- The distribution of $X_{G_{(i,a)}}$ depends on previous fixings.
- But always $\mathbf{E}[X_{G_{(i,a)}}] = |X_{G_{(i,a)}}|/m$. Moreover $X_{G_{(i,a)}} \leq d$.
- Good enough for Chernoff bounds.

Chernoff with $m \geq n^{1-1/(2c)}$ bins

W.h.p., the contribution X to given obeys Chernoff

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu/d}$$

$$\Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu/d}$$

Chernoff with $m \geq n^{1-1/(2c)}$ bins

W.h.p., the contribution X to given obeys Chernoff

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu/d}$$

$$\Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu/d}$$

Thus, from perspective of chaining, simple tabulation has same type of tail bounds as with truly random hash functions, modulo a constant factor loss and down to polynomially small probabilities.

Chernoff with $m \geq n^{1-1/(2c)}$ bins

W.h.p., the contribution X to given obeys Chernoff

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu/d}$$

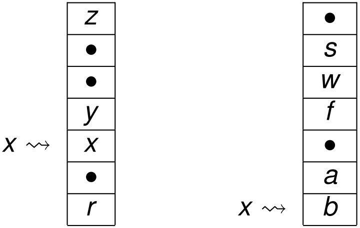
$$\Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu/d}$$

Thus, from perspective of chaining, simple tabulation has same type of tail bounds as with truly random hash functions, modulo a constant factor loss and down to polynomially small probabilities.

Similar story for linear probing.

Cuckoo hashing

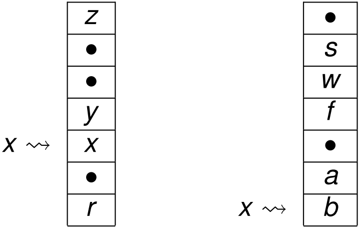
Each key placed in one of two hash locations.



Theorem With simple tabulation Cuckoo hashing works with probability $1 - \tilde{\Theta}(n^{-1/3})$.

Cuckoo hashing

Each key placed in one of two hash locations.

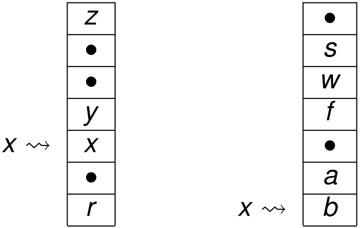


Theorem With simple tabulation Cuckoo hashing works with probability $1 - \tilde{\Theta}(n^{-1/3})$.

- For chaining and linear probing, we did not care about a constant loss, but obstructions to cuckoo hashing may be of just constant size, e.g., 3 keys sharing same two hash locations.

Cuckoo hashing

Each key placed in one of two hash locations.



Theorem With simple tabulation Cuckoo hashing works with probability $1 - \tilde{\Theta}(n^{-1/3})$.

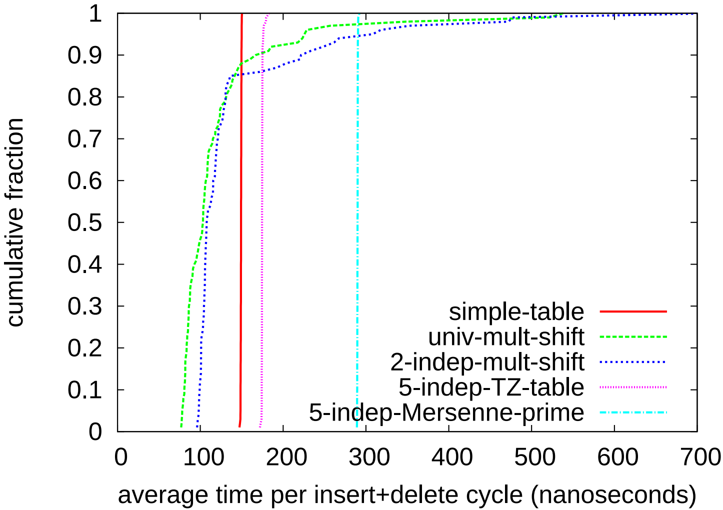
- For chaining and linear probing, we did not care about a constant loss, but obstructions to cuckoo hashing may be of just constant size, e.g., 3 keys sharing same two hash locations.
- Very delicate proof showing that obstruction can be used to code random tables R_i with few bits.

Speed

Hashing random keys		32-bit computer	64-bit computer
bits	hashing scheme	hashing time (ns)	
32	univ-mult-shift `(a*x)>>s`	1.87	2.33
32	2-indep-mult-shift	5.78	2.88
32	5-indep-Mersenne-prime	99.70	45.06
32	5-indep-TZ-table	10.12	12.66
32	simple-table	4.98	4.61
64	univ-mult-shift	7.05	3.14
64	2-indep-mult-shift	22.91	5.90
64	5-indep-Mersenne-prime	241.99	68.67
64	5-indep-TZ-table	75.81	59.84
64	simple-table	15.54	11.40

Experiments with help from Yin Zhang.

Robustness in linear probing for dense interval



Pitch for theory in case of linear probing

- Multiplicative hashing used in practice, but turns out to be very unreliable under typical denial-of-service (DoS) attacks based on consecutive IP addresses: systematic good performance 90% of the time, but systematic terrible performance 10% of the time [TZ'10].

Pitch for theory in case of linear probing

- Multiplicative hashing used in practice, but turns out to be very unreliable under typical denial-of-service (DoS) attacks based on consecutive IP addresses: systematic good performance 90% of the time, but systematic terrible performance 10% of the time [TZ'10].
- Problems in randomized algorithms like hashing hard to detect for practitioners. Hard for them to know if bad performance is from being unlucky, or because of systematic problems.

Pitch for theory in case of linear probing

- Multiplicative hashing used in practice, but turns out to be very unreliable under typical denial-of-service (DoS) attacks based on consecutive IP addresses: systematic good performance 90% of the time, but systematic terrible performance 10% of the time [TZ'10].
- Problems in randomized algorithms like hashing hard to detect for practitioners. Hard for them to know if bad performance is from being unlucky, or because of systematic problems.
- Linear probing had gotten a reputation for being fastest in practice, but sometimes unreliable needing special protection against bad cases.

Pitch for theory in case of linear probing

- Multiplicative hashing used in practice, but turns out to be very unreliable under typical denial-of-service (DoS) attacks based on consecutive IP addresses: systematic good performance 90% of the time, but systematic terrible performance 10% of the time [TZ'10].
- Problems in randomized algorithms like hashing hard to detect for practitioners. Hard for them to know if bad performance is from being unlucky, or because of systematic problems.
- Linear probing had gotten a reputation for being fastest in practice, but sometimes unreliable needing special protection against bad cases.
- Here we proved linear probing safe with good probabilistic performance for all input if we use simple tabulation.

Pitch for theory in case of linear probing

- Multiplicative hashing used in practice, but turns out to be very unreliable under typical denial-of-service (DoS) attacks based on consecutive IP addresses: systematic good performance 90% of the time, but systematic terrible performance 10% of the time [TZ'10].
- Problems in randomized algorithms like hashing hard to detect for practitioners. Hard for them to know if bad performance is from being unlucky, or because of systematic problems.
- Linear probing had gotten a reputation for being fastest in practice, but sometimes unreliable needing special protection against bad cases.
- Here we proved linear probing safe with good probabilistic performance for all input if we use simple tabulation.
- Simple tabulation also powerful for chaining, cuckoo hashing, and min-wise hashing:

  one simple and fast scheme for (almost) all your needs.

Work in progress: short range amortization with twisted tabulation

- With chaining and linear probing, each operation takes expected constant time, but out of $\sqrt{n}$ operations, some are expected to take $\tilde{\Omega}(\log n)$ time.

Work in progress: short range amortization with twisted tabulation

- With chaining and linear probing, each operation takes expected constant time, but out of $\sqrt{n}$ operations, some are expected to take $\tilde{\Omega}(\log n)$ time.
- With truly random hash function, we handle every window of $\log n$ operations in $O(\log n)$ time w.h.p.

Work in progress: short range amortization with twisted tabulation

- With chaining and linear probing, each operation takes expected constant time, but out of $\sqrt{n}$ operations, some are expected to take $\tilde{\Omega}(\log n)$ time.
- With truly random hash function, we handle every window of $\log n$ operations in $O(\log n)$ time w.h.p.
- Hence, with small buffer (as in Internet routers), we do get down to constant time per operation!

Work in progress: short range amortization with twisted tabulation

- With chaining and linear probing, each operation takes expected constant time, but out of $\sqrt{n}$ operations, some are expected to take $\tilde{\Omega}(\log n)$ time.
- With truly random hash function, we handle every window of $\log n$ operations in $O(\log n)$ time w.h.p.
- Hence, with small buffer (as in Internet routers), we do get down to constant time per operation!
- Simple tabulation does not achieve this: may often spend $\tilde{\Omega}(\log^2 n)$ time on $\log n$ consecutive operations, but can be made to work with small twist:

$$
\begin{aligned}
h &= R_1[x_1] \oplus \cdots \oplus R_{q-1}[x_{q-1}] \\
h(x) &= h \oplus R_q[((char)h) \oplus x_q]
\end{aligned}
$$

Work in progress: short range amortization with twisted tabulation

- With chaining and linear probing, each operation takes expected constant time, but out of $\sqrt{n}$ operations, some are expected to take $\tilde{\Omega}(\log n)$ time.
- With truly random hash function, we handle every window of $\log n$ operations in $O(\log n)$ time w.h.p.
- Hence, with small buffer (as in Internet routers), we do get down to constant time per operation!
- Simple tabulation does not achieve this: may often spend $\tilde{\Omega}(\log^2 n)$ time on $\log n$ consecutive operations, but can be made to work with small twist:

$$\begin{aligned} h &= R_1[x_1] \oplus \cdots \oplus R_{q-1}[x_{q-1}] \\ h(x) &= h \oplus R_q[((char)h) \oplus x_q] \end{aligned}$$

- Twisted tabulation also need for biased sampling of weighted items, and any other context where we really care about few among many keys.

Open problems

- Take any application using abstract fully-random hash function, and prove that simple/twisted tabulation works.

Open problems

- Take any application using abstract fully-random hash function, and prove that simple/twisted tabulation works.
- Could this be the first implementable hash function/RNG making classic quick sort work directly: using hash of i to generate index of ith pivot?

Open problems

- Take any application using abstract fully-random hash function, and prove that simple/twisted tabulation works.
- Could this be the first implementable hash function/RNG making classic quick sort work directly: using hash of i to generate index of ith pivot?
- Hash tables are used to look up keys in a dynamic set of stored keys. Can this be done without randomization?

Open problems

- Take any application using abstract fully-random hash function, and prove that simple/twisted tabulation works.
- Could this be the first implementable hash function/RNG making classic quick sort work directly: using hash of i to generate index of ith pivot?
- Hash tables are used to look up keys in a dynamic set of stored keys. Can this be done without randomization?
- Can we both insert and look up keys in constant deterministic time? (not just with high probability)

Open problems

- Take any application using abstract fully-random hash function, and prove that simple/twisted tabulation works.
- Could this be the first implementable hash function/RNG making classic quick sort work directly: using hash of i to generate index of ith pivot?
- Hash tables are used to look up keys in a dynamic set of stored keys. Can this be done without randomization?
- Can we both insert and look up keys in constant deterministic time? (not just with high probability)
- Currently, the best answer is that we can do both in $O(\sqrt{\log n / \log\log n})$ worst-case time [Andersson Thorup STOC'00] —tight for more general predecessor problem.

Open problems

- Take any application using abstract fully-random hash function, and prove that simple/twisted tabulation works.
- Could this be the first implementable hash function/RNG making classic quick sort work directly: using hash of i to generate index of ith pivot?
- Hash tables are used to look up keys in a dynamic set of stored keys. Can this be done without randomization?
- Can we both insert and look up keys in constant deterministic time? (not just with high probability)
- Currently, the best answer is that we can do both in $O(\sqrt{\log n / \log\log n})$ worst-case time [Andersson Thorup STOC'00] —tight for more general predecessor problem.
- Most people believe that deterministic constant time is not possible without randomization, but nobody can prove it.

Open problems

- Take any application using abstract fully-random hash function, and prove that simple/twisted tabulation works.
- Could this be the first implementable hash function/RNG making classic quick sort work directly: using hash of i to generate index of ith pivot?
- Hash tables are used to look up keys in a dynamic set of stored keys. Can this be done without randomization?
- Can we both insert and look up keys in constant deterministic time? (not just with high probability)
- Currently, the best answer is that we can do both in $O(\sqrt{\log n / \log\log n})$ worst-case time [Andersson Thorup STOC'00] —tight for more general predecessor problem.
- Most people believe that deterministic constant time is not possible without randomization, but nobody can prove it.
- So far, no technique is known that can make any such separation between deterministic and randomized solutions for *any* data structure problem.