Hash-tables - or how a computer looks things up

Mikkel Thorup
Keys and data

The Danish word for a computer is a data machine, highlighting significance of how data are organized and accessed.
Keys and data

The Danish word for a computer is a data machine, highlighting significance of how data are organized and accessed. One of the most basic functionalities is that we want to associate data with keys.
Keys and data

The Danish word for a computer is a data machine, highlighting significance of how data are organized and accessed. One of the most basic functionalities is that we want to associate data with keys.

- In phone book, key is a name and data is address and phone number (perhaps multiple if more than one person with the name).
Keys and data

The Danish word for a computer is a data machine, highlighting significance of how data are organized and accessed. One of the most basic functionalities is that we want to associate data with keys.

- In phone book, key is a name and data is address and phone number (perhaps multiple if more than one person with the name).
- In dictionary, the key is a word, and the data are the possible meanings of the word. A computer dictionary may also have common misspellings of words with associated possible correct spellings.
Keys and data

The Danish word for a computer is a data machine, highlighting significance of how data are organized and accessed. One of the most basic functionalities is that we want to associate data with keys.

- In phone book, key is a **name** and data is **address and phone number** (perhaps multiple if more than one person with the name).
- In dictionary, the key is a **word**, and the data are the **possible meanings** of the word. A computer dictionary may also have **common misspellings of words** with associated **possible correct spellings**.
- With google, key could be a **word or a common combination of words** and the data could be **links to relevant web-pages**.
Keys and data

The Danish word for a computer is a data machine, highlighting significance of how data are organized and accessed. One of the most basic functionalities is that we want to associate data with keys.

- In phone book, key is a name and data is address and phone number (perhaps multiple if more than one person with the name).
- In dictionary, the key is a word, and the data are the possible meanings of the word. A computer dictionary may also have common misspellings of words with associated possible correct spellings.
- With google, key could be a word or a common combination of words and the data could be links to relevant web-pages.
- When you log into a computer, the key is your username, and the data account information such as password that you have to match for access.
Keys and data

So far only top of the iceberg: the part we see directly as users.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>src IP</td>
<td>11.2.5.3</td>
</tr>
<tr>
<td>src port</td>
<td>80</td>
</tr>
<tr>
<td>dest IP</td>
<td>165.11.4.88</td>
</tr>
<tr>
<td>dest port</td>
<td>48811</td>
</tr>
<tr>
<td>protocol</td>
<td>17</td>
</tr>
<tr>
<td>bytes</td>
<td>1340</td>
</tr>
</tbody>
</table>

To find out which packets belong together, we group them based on having the same key: vector of red values. For download size, add byte counts for packets with same key.
Keys and data

So far only top of the iceberg: the part we see directly as users. Associating data with keys is used internally in many kinds of data processing.
Keys and data

So far only top of the iceberg: the part we see directly as users. Associating data with keys is used internally in many kinds of data processing. When you download a file from Internet, it is split into packets, each with a header containing information like:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>src IP</td>
<td>11.2.5.3</td>
</tr>
<tr>
<td>src port</td>
<td>80</td>
</tr>
<tr>
<td>dest IP</td>
<td>165.11.4.88</td>
</tr>
<tr>
<td>dest port</td>
<td>48811</td>
</tr>
<tr>
<td>protocol</td>
<td>17</td>
</tr>
<tr>
<td>bytes</td>
<td>1340</td>
</tr>
</tbody>
</table>
Keys and data

So far only top of the iceberg: the part we see directly as users. Associating data with keys is used internally in many kinds of data processing.
When you download a file from Internet, it is split into packets, each with a header containing information like:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>src IP</td>
<td>11.2.5.3</td>
</tr>
<tr>
<td>src port</td>
<td>80</td>
</tr>
<tr>
<td>dest IP</td>
<td>165.11.4.88</td>
</tr>
<tr>
<td>dest port</td>
<td>48811</td>
</tr>
<tr>
<td>protocol</td>
<td>17</td>
</tr>
<tr>
<td>bytes</td>
<td>1340</td>
</tr>
</tbody>
</table>

To find out which packets belong together, we group them based on having the same key: vector of red values.
Keys and data

So far only top of the iceberg: the part we see directly as users. Associating data with keys is used internally in many kinds of data processing.

When you download a file from Internet, it is split into packets, each with a header containing information like:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>src IP</td>
<td>11.2.5.3</td>
</tr>
<tr>
<td>src port</td>
<td>80</td>
</tr>
<tr>
<td>dest IP</td>
<td>165.11.4.88</td>
</tr>
<tr>
<td>dest port</td>
<td>48811</td>
</tr>
<tr>
<td>protocol</td>
<td>17</td>
</tr>
<tr>
<td>bytes</td>
<td>1340</td>
</tr>
</tbody>
</table>

To find out which packets belong together, we group them based on having the same key: vector of red values. For download size, add byte counts for packets with same key.
Hash table

- The association between keys and data is done via hash tables. Hash tables also known as dictionaries, indexing in data bases, and associative arrays.
Hash table

- The association between keys and data is done via hash tables. Hash tables also known as dictionaries, indexing in data bases, and associative arrays.
- Hash tables are often bottlenecks in data processing, so better dictionaries have impact wherever computers are used.
Hash table

- The association between keys and data is done via hash tables. Hash tables also known as dictionaries, indexing in data bases, and associative arrays.

- Hash tables are often bottlenecks in data processing, so better dictionaries have impact wherever computers are used.

- The basic problem is that computers can only associate data with with a limited number of indices 0, 1, 2, 3..., m, not general keys.
Hash table

- The association between keys and data is done via hash tables. Hash tables also known as dictionaries, indexing in data bases, and associative arrays.
- Hash tables are often bottlenecks in data processing, so better dictionaries have impact wherever computers are used.
- The basic problem is that computers can only associate data with a limited number of indices 0, 1, 2, 3..., m, not general keys.
- Use hash function mapping any possible key to an index.
Hash table

- The association between keys and data is done via hash tables. Hash tables also known as dictionaries, indexing in data bases, and associative arrays.
- Hash tables are often bottlenecks in data processing, so better dictionaries have impact wherever computers are used.
- The basic problem is that computers can only associate data with with a limited number of indices $0, 1, 2, 3..., m$, not general keys.
- Use hash function mapping any possible key to an index.
- For each index, we can have a chain with all keys mapping to that index.
From name to phone number?

Hash name based on first letter: A $\mapsto$ 0, B $\mapsto$ 1, .., K $\mapsto$ 10, ...

<table>
<thead>
<tr>
<th>Hash Code</th>
<th>Name</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Anders</td>
<td>2799 1478</td>
</tr>
<tr>
<td>1</td>
<td>Bjarke</td>
<td>8526 6739</td>
</tr>
<tr>
<td>2</td>
<td>And</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Aksel</td>
<td>4758 9977</td>
</tr>
<tr>
<td>4</td>
<td>Emma</td>
<td>8771 8845</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Kasper</td>
<td>2883 4817</td>
</tr>
<tr>
<td>11</td>
<td>Line</td>
<td>3358 8393</td>
</tr>
<tr>
<td>12</td>
<td>Mikkel</td>
<td>2297 4422</td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Rasmus</td>
<td>3317 8020</td>
</tr>
<tr>
<td>18</td>
<td>Sophie</td>
<td>8020 7957</td>
</tr>
<tr>
<td>19</td>
<td>Trine</td>
<td>9253 2542</td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Â</td>
<td></td>
</tr>
</tbody>
</table>
From name to phone number? Kirsten?

Hash name based on first letter: A ⇔ 0, B ⇔ 1, .., K ⇔ 10, ...

0 (A) → Anders 2799 1478 → Annika 4166 3804 → Andy 2246 7019
1 (B) → Bjarke 8526 6739 → ☒
2 (C) → ☒
3 (D) → ☒
4 (E) → Emma 8771 8845 → ☒
5 (F) → ☒
...
10 (K) → Kasper 2883 4817 → Kirsten 2029 5179 → ☒
11 (L) → Line 3358 8393 → Leo 7348 9225 → ☒
12 (M) → Mikkel 2297 4422 → ☒
...
17 (R) → Rasmus 3317 8020 → ☒
18 (S) → Sophie 8020 7957 → ☒
19 (T) → Trine 9253 2542 → ☒
...
27 (˚A) → ☒
From name to phone number? Kirsten?

Hash name based on first letter: A ↦ 0, B ↦ 1,.., K ↦ 10,...

0 (A) → Anders 2799 1478 → Annika 4166 3804 → Andy 2246 7019
1 (B) → Bjarke 8526 6739 → ⃝
2 (C) → ⃝
3 (D) → ⃝
4 (E) → Emma 8771 8845 → ⃝
5 (F) → ⃝

... 10 (K) → Kasper 2883 4817 → Kirsten 2029 5179 → ⃝
11 (L) → Line 3358 8393 → Leo 7348 9225 → ⃝
12 (M) → Mikkel 2297 4422 → ⃝

... 17 (R) → Rasmus 3317 8020 → ⃝
18 (S) → Sophie 8020 7957 → ⃝
19 (T) → Trine 9253 2542 → ⃝

... 27 (˚A) → ⃝
From name to phone number? _Kirsten?

Hash name based on first letter: A $\mapsto$ 0, B $\mapsto$ 1,.., K $\mapsto$ 10,...

0 (A) $\mapsto$ Anders 2799 1478 $\mapsto$ Annika 4166 3804 $\mapsto$ Andy 2246 7019
1 (B) $\mapsto$ Bjarke 8526 6739 $\mapsto$ Aksel 4758 9977
2 (C) $\mapsto$ Emma 8771 8845 $\mapsto$ Leo 7348 9225
3 (D) $\mapsto$ Mikkel 2297 4422 $\mapsto$ Mikkel 2297 4422
4 (E) $\mapsto$ Emma 8771 8845 $\mapsto$ Emma 8771 8845
5 (F) $\mapsto$ Mikkel 2297 4422 $\mapsto$ Mikkel 2297 4422

...:

10 (K) $\mapsto$ Kasper 2883 4817 $\mapsto$ Kirsten 2029 5179 $\mapsto$ Kirsten 2029 5179
11 (L) $\mapsto$ Line 3358 8393 $\mapsto$ Leo 7348 9225 $\mapsto$ Leo 7348 9225
12 (M) $\mapsto$ Mikkel 2297 4422 $\mapsto$ Mikkel 2297 4422

...:

17 (R) $\mapsto$ Rasmus 3317 8020 $\mapsto$ Sophie 8020 7957 $\mapsto$ Sophie 8020 7957
18 (S) $\mapsto$ Sophie 8020 7957 $\mapsto$ Sophie 8020 7957
19 (T) $\mapsto$ Trine 9253 2542 $\mapsto$ Trine 9253 2542

...:

27 (˚A) $\mapsto$
From name to phone number? Kirsten?

Hash name based on first letter: $A \leftrightarrow 0, B \leftrightarrow 1, \ldots, K \leftrightarrow 10, \ldots$

- $0$ (A) $\rightarrow$ Anders $2799$ $1478$ $\rightarrow$ Annika $4166$ $3804$ $\rightarrow$ Andy $2246$ $7019$
- $1$ (B) $\rightarrow$ Bjarke $8526$ $6739$ $\rightarrow$ $\times$
- $2$ (C) $\rightarrow$ $\times$
- $3$ (D) $\rightarrow$ $\times$
- $4$ (E) $\rightarrow$ Emma $8771$ $8845$ $\rightarrow$ $\times$
- $5$ (F) $\rightarrow$ $\times$
- $10$ (K) $\rightarrow$ Kasper $2883$ $4817$ $\rightarrow$ Kirsten $2029$ $5179$ $\rightarrow$ $\times$
- $11$ (L) $\rightarrow$ Line $3358$ $8393$ $\rightarrow$ Leo $7348$ $9225$ $\rightarrow$ $\times$
- $12$ (M) $\rightarrow$ Mikkel $2297$ $4422$ $\rightarrow$ $\times$
- $17$ (R) $\rightarrow$ Rasmus $3317$ $8020$ $\rightarrow$ $\times$
- $18$ (S) $\rightarrow$ Sophie $8020$ $7957$ $\rightarrow$ $\times$
- $19$ (T) $\rightarrow$ Trine $9253$ $2542$ $\rightarrow$ $\times$
- $27$ (˚A) $\rightarrow$ $\times$
From name to phone number? Kirsten? 2029 5179

Hash name based on first letter: A → 0, B → 1,.., K → 10,...

0 (A) → Anders 2799 1478 → Annika 4166 3804 → Andy 2246 7019
1 (B) → Bjarke 8526 6739 → ⊗
2 (C) → ⊗
3 (D) → ⊗
4 (E) → Emma 8771 8845 → ⊗
5 (F) → ⊗
...
10 (K) → Kasper 2883 4817 → Kirsten 2029 5179 → ⊗
11 (L) → Line 3358 8393 → Leo 7348 9225 → ⊗
12 (M) → Mikkel 2297 4422 → ⊗
...
17 (R) → Rasmus 3317 8020 → ⊗
18 (S) → Sophie 8020 7957 → ⊗
19 (T) → Trine 9253 2542 → ⊗
...
27 (˚Å) → ⊗
From name to phone number? Alberta?

Hash name based on first letter: (A ↦ 0, B ↦ 1, ..., K ↦ 10, ...)

0 (A) → Anders 2799 1478 → Annika 4166 3804 → Andy 2246 7019
1 (B) → Bjarke 8526 6739 → ☠
2 (C) → ☠
3 (D) → ☠
4 (E) → Emma 8771 8845 → ☠
5 (F) → ☠

... 
10 (K) → Kasper 2883 4817 → Kirsten 2029 5179 → ☠
11 (L) → Line 3358 8393 → Leo 7348 9225 → ☠
12 (M) → Mikkel 2297 4422 → ☠

... 
17 (R) → Rasmus 3317 8020 → ☠
18 (S) → Sophie 8020 7957 → ☠
19 (T) → Trine 9253 2542 → ☠

... 
27 (˚A) → ☠
From name to phone number? Albert?

Hash name based on first letter: (A↦0, B↦1,..,K↦10,...

0 (A) → Anders  2799 1478 → Annika 4166 3804 → Andy 2246 7019
1 (B) → Bjarke  8526 6739 → X
2 (C) → X
3 (D) → X
4 (E) → Emma  8771 8845 → X
5 (F) → X

...  
10 (K) → Kasper  2883 4817 → Kirsten 2029 5179 → X
11 (L) → Line  3358 8393 → Leo  7348 9225 → X
12 (M) → Mikkel  2297 4422 → X

...  
17 (R) → Rasmus  3317 8020 → X
18 (S) → Sophie  8020 7957 → X
19 (T) → Trine  9253 2542 → X

...  
27 (˚A) → X
<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Name</th>
<th>Phone Number 1</th>
<th>Phone Number 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (A)</td>
<td>Anders</td>
<td>2799 1478</td>
<td></td>
</tr>
<tr>
<td>1 (B)</td>
<td>Bjarke</td>
<td>8526 6739</td>
<td></td>
</tr>
<tr>
<td>2 (C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (D)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (E)</td>
<td>Emma</td>
<td>8771 8845</td>
<td></td>
</tr>
<tr>
<td>5 (F)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 (K)</td>
<td>Kasper</td>
<td>2883 4817</td>
<td></td>
</tr>
<tr>
<td>11 (L)</td>
<td>Line</td>
<td>3358 8393</td>
<td></td>
</tr>
<tr>
<td>12 (M)</td>
<td>Mikkel</td>
<td>2297 4422</td>
<td></td>
</tr>
<tr>
<td>13 (N)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 (O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 (P)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 (Q)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 (R)</td>
<td>Rasmus</td>
<td>3317 8020</td>
<td></td>
</tr>
<tr>
<td>18 (S)</td>
<td>Sophie</td>
<td>8020 7957</td>
<td></td>
</tr>
<tr>
<td>19 (T)</td>
<td>Trine</td>
<td>9253 2542</td>
<td></td>
</tr>
<tr>
<td>20 (U)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21 (V)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22 (W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23 (X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 (Y)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 (Z)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26 (A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27 (Å)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
From name to phone number? Albert?

Hash name based on first letter: \((A \mapsto 0, B \mapsto 1, \ldots, K \mapsto 10, \ldots)\)

0 (A) → Anders 2799 1478 → Annika 4166 3804 → Andy 2246 7019
1 (B) → Bjarke 8526 6739 →
2 (C) →
3 (D) →
4 (E) → Emma 8771 8845 →
5 (F) →
   ...
10 (K) → Kasper 2883 4817 → Kirsten 2029 5179 →
11 (L) → Line 3358 8393 → Leo 7348 9225 →
12 (M) → Mikkel 2297 4422 →
   ...
17 (R) → Rasmus 3317 8020 →
18 (S) → Sophie 8020 7957 →
19 (T) → Trine 9253 2542 →
   ...
27 (Å) →
From name to phone number? Albert?

Hash name based on first letter: (A → 0, B → 1, ..., K → 10, ...)

<table>
<thead>
<tr>
<th>Hash Letter</th>
<th>Name</th>
<th>Phone Number</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (A)</td>
<td>Anders</td>
<td>2799 1478</td>
<td>Annika 4166 3804 → Andy 2246 7019</td>
</tr>
<tr>
<td>1 (B)</td>
<td>Bjarke</td>
<td>8526 6739</td>
<td></td>
</tr>
<tr>
<td>2 (C)</td>
<td></td>
<td></td>
<td>Aksel 4758 9977</td>
</tr>
<tr>
<td>3 (D)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (E)</td>
<td>Emma</td>
<td>8771 8845</td>
<td></td>
</tr>
<tr>
<td>5 (F)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 (K)</td>
<td>Kasper</td>
<td>2883 4817</td>
<td>Kirsten 2029 5179 →</td>
</tr>
<tr>
<td>11 (L)</td>
<td>Line</td>
<td>3358 8393</td>
<td>Leo 7348 9225 →</td>
</tr>
<tr>
<td>12 (M)</td>
<td>Mikkel</td>
<td>2297 4422</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 (R)</td>
<td>Rasmus</td>
<td>3317 8020</td>
<td></td>
</tr>
<tr>
<td>18 (S)</td>
<td>Sophie</td>
<td>8020 7957</td>
<td></td>
</tr>
<tr>
<td>19 (T)</td>
<td>Trine</td>
<td>9253 2542</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27 (Å)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
From name to phone number? _Albert?

Hash name based on first letter: (A ↦ 0, B ↦ 1, .., K ↦ 10, ...)

0 (A) → Anders 2799 1478 → Annika 4166 3804 → Andy 2246 7019
1 (B) → Bjarke 8526 6739 →          ↓
2 (C) →         ↓
3 (D) →
4 (E) → Emma 8771 8845 →          ↓
5 (F) →
...
10 (K) → Kasper 2883 4817 → Kirsten 2029 5179 →          ↓
11 (L) → Line 3358 8393 → Leo 7348 9225 →          ↓
12 (M) → Mikkel 2297 4422 →          ↓
...
17 (R) → Rasmus 3317 8020 →          ↓
18 (S) → Sophie 8020 7957 →          ↓
19 (T) → Trine 9253 2542 →          ↓
...
27 (˚A) →          ↓
From name to phone number? Albert? absent

Hash name based on first letter: (A\rightarrow 0, B\rightarrow 1, \ldots, K\rightarrow 10, \ldots)

0 (A) \rightarrow Anders 2799 1478 \rightarrow Annika 4166 3804 \rightarrow Andy 2246 7019
1 (B) \rightarrow Bjarke 8526 6739 \rightarrow \bigotimes
2 (C) \rightarrow \bigotimes
3 (D) \rightarrow \bigotimes
4 (E) \rightarrow Emma 8771 8845 \rightarrow \bigotimes
5 (F) \rightarrow \bigotimes
\vdots
10 (K) \rightarrow Kasper 2883 4817 \rightarrow Kirsten 2029 5179 \rightarrow \bigotimes
11 (L) \rightarrow Line 3358 8393 \rightarrow Leo 7348 9225 \rightarrow \bigotimes
12 (M) \rightarrow Mikkel 2297 4422 \rightarrow \bigotimes
\vdots
17 (˚A) \rightarrow Rasmus 3317 8020 \rightarrow \bigotimes
18 (S) \rightarrow Sophie 8020 7957 \rightarrow \bigotimes
19 (T) \rightarrow Trine 9253 2542 \rightarrow \bigotimes
\vdots
27 (˚Å) \rightarrow \bigotimes
From name to phone number?

Start with surname?
From name to phone number?

Start with surname?

1 (A) →  ☒

... 

9 (I) →  Inoue, Leo  7348 9225 →  ☒

... 

12 (L) →  Lützen, Trine  9253 2542 →  ☒

... 

19 (T) →  Thorup, Anders  2799 1478 →  Thorup-Lützen, Annika  4166 3804 →  Thøgersen, Anders  2246 7019

   Thøgersen, Bjarke  8526 6739 ←  Thøgersen, Aksel  4758 9977 ←  Thorup, Emma  8771 8845

   ↓

   Thorup, Kasper  2883 4817 →  Thorup, Kirsten  2029 5179 →  Thorup, Mikkel  2297 4422

   ☒

   ←  Thorup, Sophie  8020 7957 ←  Thorup-Lützen, Rasmus  3317 8020

... 

28 (˚A) →  ☒
From name to phone number?

Start with surname?

1 (A) → ☒

... 

9 (I) → Inoue, Leo 7348 9225 → ☒

... 

12 (L) → Lützen, Trine 9253 2542 → ☒

... 

19 (T) → Thorup, Anders 2799 1478 → Thorup-Lützen, Annika 4166 3804 → Thøgersen, Anders 2246 7019

↓

Thøgersen, Bjarke 8526 6739 ← Thøgersen, Aksel 4758 9977 ← Thorup, Emma 8771 8845

↓

Thorup, Kasper 2883 4817 → Thorup, Kirsten 2029 5179 → Thorup, Mikkel 2297 4422

↓

,Thorup, Sophie 8020 7957 ← Thorup-Lützen, Rasmus 3317 8020

... 

28 (˚A) → ☒

Long T-chain in my family phone book: slow look-ups for surnames starting with T.
Hash tables with chains

Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.

We have to pay memory for every chain, even empty ones.

Typically, we aim for $m \approx 2^n$ chains.

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

Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

Fundamental problem If hash function $h$ known, bad guy may provide set $X$ of keys that all have same hash, and then we get one long chain.

No "bad guy" gets to pick names for my phone book, but there are lots of bad guys who try to take down the Internet with vicious attacks.

As in surname example, a bad case may not be malicious, just a bad choice of hash function for the input.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- We have to pay memory for every chain, even empty ones.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- We have to pay memory for every chain, even empty ones.
- Typically, we aim for $m \approx 2n$ chains.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- We have to pay memory for every chain, even empty ones.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$. 
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- We have to pay memory for every chain, even empty ones.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, ..., m - 1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- We have to pay memory for every chain, even empty ones.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

Fundamental problem If hash function $h$ known, bad guy may provide set $X$ of keys that all have same hash, and then we get one long chain.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- We have to pay memory for every chain, even empty ones.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

**Fundamental problem** If hash function $h$ known, bad guy may provide set $X$ of keys that all have same hash, and then we get one long chain.

- No “bad guy” gets to pick names for my phone book, but there are lots of bad guys who try to take down the Internet with vicious attacks.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- We have to pay memory for every chain, even empty ones.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

**Fundamental problem** If hash function $h$ known, **bad guy** may provide set $X$ of keys that all have same hash, and then we get one long chain.

- No “bad guy” gets to pick names for my phone book, but there are lots of bad guys who try to take down the Internet with vicious attacks.
- As in surname example, a bad case may not be malicious, just a bad choice of hash function for the input.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

**Fundamental problem** If hash function $h$ known, bad guy may provide set $X$ of keys that all have the same hash, and then we get one long chain.
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, ..., m - 1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

**Fundamental problem** If hash function $h$ known, bad guy may provide set $X$ of keys that all have the same hash, and then we get one long chain.

**Solution idea** Independent of $X$, secretly fix $h$ at random, assigning every possible key an independent random hash.

The expected number of keys in chain with any given key is $(n - 1)/m \approx 1/2$.

With very high probability, e.g., $1 - 1/n^{10}$, the maximal chain length is $O((\log n)/(\log \log n))$. 
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

**Fundamental problem** If hash function $h$ known, bad guy may provide set $X$ of keys that all have the same hash, and then we get one long chain.

**Solution idea** Independent of $X$, secretly fix $h$ at random, assigning every possible key an independent random hash.
  - The expected number of keys in chain with any given key is $(n-1)/m \approx 1/2$. 
Hash tables with chains

- Set $X$ of $n$ keys (14 names above) from a much much larger universe $\mathcal{U}$ of possible keys.
- Typically, we aim for $m \approx 2n$ chains.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$.
- Keys from $X$ with same hash end in the same chain, and long chains mean slow access.

**Fundamental problem** If hash function $h$ known, bad guy may provide set $X$ of keys that all have the same hash, and then we get one long chain.

**Solution idea** Independent of $X$, secretly fix $h$ at random, assigning every possible key an independent random hash.
- The expected number of keys in chain with any given key is $(n - 1)/m \approx 1/2$.
- With very high probability, e.g., $1 - 1/n^{10}$, the maximal chain length is $O((\log n)/(\log \log n))$. 
Phone book with truly random hash function

0 → Sophie 8020 7957 → ❌
1 → ❌
2 → Trine 9253 2542 → ❌
3 → ❌
4 → ❌
5 → ❌
6 → Line 3358 8393 → Rasmus 3317 8020 → ❌
7 → Anni 4166 3804 → ❌
8 → ❌
9 → Aksel 4758 9977 → Leo 7348 9225 → ❌
10 → ❌
11 → Andy 2246 7019 → ❌
12 → ❌
13 → ❌
14 → Emma 8771 8845 → ❌
15 → ❌
16 → ❌
17 → ❌
18 → ❌
19 → Bjarke 8526 6739 → ❌
20 → Anders 2799 1478 → ❌
21 → ❌
22 → Kirsten 2029 5179 → ❌
23 → ❌
24 → ❌
25 → Mikkel 2297 4422 → ❌
26 → ❌
27 → Kasper 2883 4817 → ❌
Phone book with truly random hash function

0 → Sophie 8020 7957 → ☒
1 → ☒
2 → Trine 9253 2542 → ☒
3 → ☒
4 → ☒
5 → ☒
6 → Line 3358 8393 → Rasmus 3317 8020 → ☒
7 → Annika 4166 3804 → ☒
8 → ☒
9 → Aksel 4758 9977 → Leo 7348 9225 → ☒
10 → ☒
11 → Andy 2246 7019 → ☒
12 → ☒
13 → ☒
14 → Emma 8771 8845 → ☒
15 → ☒
16 → ☒
17 → ☒
18 → ☒
19 → Bjarke 8526 6739 → ☒
20 → Anders 2799 1478 → ☒
21 → ☒
22 → Kirsten 2029 5179 → ☒
23 → ☒
24 → ☒
25 → Mikkel 2297 4422 → ☒
26 → ☒
27 → Kasper 2883 4817 → ☒

▶ Names nicely spread and with at most 2 in each chain.
Phone book with truly random hash function

0 → Sophie 8020 7957
1 →  
2 → Trine 9253 2542
3 →  
4 →  
5 →  
6 → Line 3358 8393 → Rasmus 3317 8020
7 → Annika 4166 3804
8 →  
9 → Aksel 4758 9977 → Leo 7348 9225
10 →  
11 → Andy 2246 7019
12 →  
13 →  
14 → Emma 8771 8845
15 →  
16 →  
17 →  
18 →  
19 → Bjarke 8526 6739
20 → Anders 2799 1478
21 →  
22 → Kirsten 2029 5179
23 →  
24 →  
25 → Mikkel 2297 4422
26 →  
27 → Kasper 2883 4817

Split
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Names nicely spread and with at most 2 in each chain.

In what chains should I look for Kirsten or Albert?
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use **pseudo-random hash function**:

- Suppose $\mathcal{U}$ are integers bounded by some prime $\mathcal{P}$—on computer, everything can be viewed as integers.
- Independent of $X$, pick one secret uniformly random number $R \in \mathbb{Z}_\mathcal{P}$, and define $h_R(x) = ((x \times R) \mod \mathcal{P}) \mod m$.
- For two different $x, y \in \mathbb{Z}_\mathcal{P}$, $\Pr_{R \in \mathbb{Z}_\mathcal{P}}[h_R(x) = h_R(y)] < \frac{2}{m}$.
- By linearity of expectation, the expected number of keys in chain with any given key is $< \frac{2(n - 1)}{m} \approx 1$.

OK random

Unfortunately there exists bad sets $X$ such that the expected maximal chain length is $\sqrt{n} \gg (\log n) / (\log \log n)$.
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use pseudo-random hash function:

- Suppose $\mathcal{U}$ are integers bounded by some prime $p$—on computer, everything can be viewed as integers.

Unfortunately there exists bad sets $X$ such that the expected maximal chain length is $\sqrt{n} \gg \frac{(\log n)}{(\log \log n)}$. 
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \rightarrow \{0, ..., m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use pseudo-random hash function:

- Suppose $\mathcal{U}$ are integers bounded by some prime $\wp$—on computer, everything can be viewed as integers.
- Independent of $X$, pick one secret uniformly random number $R \in \mathbb{Z}_\wp$, and define

$$h_R(x) = ((x \times R) \mod \wp) \mod m$$
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use pseudo-random hash function:

- Suppose $\mathcal{U}$ are integers bounded by some prime $\wp$ —on computer, everything can be viewed as integers.
- Independent of $X$, pick one secret uniformly random number $R \in \mathbb{Z}_\wp$, and define

$$h_R(x) = (((x \times R) \mod \wp) \mod m)

- For two different $x, y \in \mathbb{Z}_\wp$, $\Pr_{R \in \mathbb{Z}_\wp} [h_R(x) = h_R(y)] < 2/m.$
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \rightarrow \{0, ..., m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use pseudo-random hash function:

- Suppose $\mathcal{U}$ are integers bounded by some prime $\wp$ —on computer, everything can be viewed as integers.
- Independent of $X$, pick one secret uniformly random number $R \in \mathbb{Z}_\wp$, and define
  \[
  h_R(x) = ((x \times R) \mod \wp) \mod m
  \]

- For two different $x, y \in \mathbb{Z}_\wp$, $\Pr_{R \in \mathbb{Z}_\wp} [h_R(x) = h_R(y)] < 2/m$.
- By linearity of expectation, the expected number of keys in chain with any given key is $< 2(n - 1)/m \approx 1$. 
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use pseudo-random hash function:

- Suppose $\mathcal{U}$ are integers bounded by some prime $\wp$ —on computer, everything can be viewed as integers.
- Independent of $X$, pick one secret uniformly random number $R \in \mathbb{Z}_\wp$, and define

$$h_R(x) = ((x \times R) \mod \wp) \mod m$$

- For two different $x, y \in \mathbb{Z}_\wp$, $\Pr_{R \in \mathbb{Z}_\wp}[h_R(x) = h_R(y)] < 2/m$.
- By linearity of expectation, the expected number of keys in chain with any given key is $< 2(n - 1)/m \approx 1$. OK random
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \longrightarrow \{0, \ldots, m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use pseudo-random hash function:

- Suppose $\mathcal{U}$ are integers bounded by some prime $\wp$—on computer, everything can be viewed as integers.
- Independent of $X$, pick one secret uniformly random number $R \in \mathbb{Z}_{\wp}$, and define

\[ h_R(x) = ((x \times R) \mod \wp) \mod m \]

- For two different $x, y \in \mathbb{Z}_{\wp}$, $\Pr_{R \in \mathbb{Z}_{\wp}}[h_R(x) = h_R(y)] < 2/m$.
- By linearity of expectation, the expected number of keys in chain with any given key is $< 2(n - 1)/m \approx 1$. OK random

- Unfortunately there exists bad sets $X$ such that the expected maximal chain length is $\sqrt{n}$.
Pseudo-random hash function

- Set $X$ of $n$ keys from universe $\mathcal{U}$ of possible keys.
- Hash function $h : \mathcal{U} \rightarrow \{0, ..., m - 1\}$, $m \approx 2n$.

Not feasible to represent truly random $h$, storing hash for every possible key, but we can use pseudo-random hash function:

- Suppose $\mathcal{U}$ are integers bounded by some prime $\wp$—on computer, everything can be viewed as integers.
- Independent of $X$, pick one secret uniformly random number $R \in \mathbb{Z}_\wp$, and define
  \[
  h_R(x) = (((x \times R) \mod \wp) \mod m
  \]
- For two different $x, y \in \mathbb{Z}_\wp$, $\Pr_{R \in \mathbb{Z}_\wp}[h_R(x) = h_R(y)] < 2/m$.
- By linearity of expectation, the expected number of keys in chain with any given key is $< 2(n - 1)/m \approx 1$. OK random
- Unfortunately there exists bad sets $X$ such that the expected maximal chain length is $\sqrt{n} \gg (\log n)/(\log \log n)$. 
Target

- Simple and reliable pseudo-random hashing.
Target

- Simple and reliable pseudo-random hashing.
- Providing important probabilistic guarantees akin to those of truly random hashing, yet easy to implement.
Target

- Simple and reliable pseudo-random hashing.
- Providing **important** probabilistic guarantees akin to those of truly random hashing, yet easy to implement.
- Bridging theory (assuming truly random hashing) with practice (needing something implementable).
The Power of Tabulation Hashing

Joint work with Mihai Pătraşcu
Applications of Hashing

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

- **chaining**: follow pointers

$$x \rightsquigarrow$$

- •
- •
- •
- •
- •

- $\rightarrow a \rightarrow t$
- $\rightarrow v$
- $\rightarrow f \rightarrow s \rightarrow r$
Applications of Hashing

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

- chaining: follow pointers

$x \rightsquigarrow$

- $\bullet$ → $a$ → $t$ → $x$
- $\bullet$ → $v$
- $\bullet$ → $f$ → $s$ → $r$
Applications of Hashing

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

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

\[
\begin{array}{c}
\bullet \\
q \\
a \\
g \\
c \\
\bullet \\
\bullet \\
t \\
\end{array}
\]
Applications of Hashing

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

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

\[
\begin{array}{c|c|c}
    a & \bullet & \bullet \\
    \bullet & \bullet & \bullet \\
    y & z & f \\
    w & s & \bullet \\
    \bullet & r & \bullet \\
    \bullet & \bullet & b \\
\end{array}
\]
Applications of Hashing

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

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

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

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

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

![Diagram of hash table entries]
Applications of Hashing

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

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

Sketching, streaming, and sampling:
- second moment estimation: $F_2(\bar{x}) = \sum_i x_i^2$
Applications of Hashing

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

Sketching, streaming, and sampling:

- second 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 $\varepsilon$-minwise independent:

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

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

Family $\mathcal{H} = \{ h : [u] \to [b] \}$ \textit{k-independent} iff for random $h \in \mathcal{H}$:

- $(\forall) x \in [u], h(x)$ 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

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

Many solutions for $k$-independent hashing proposed, but generally slow for $k \geq 3$ and too slow for $k \geq 5$. 
Carter & Wegman (1977)

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

Family $\mathcal{H} = \{ h : [u] \to [b] \}$ \textit{k-independent} iff for random $h \in \mathcal{H}$:

$\begin{align*}
\forall x \in [u], h(x) & \text{ is uniform in } [b]; \\
\forall x_1, \ldots, x_k \in [u], h(x_1), \ldots, h(x_k) & \text{ are independent.}
\end{align*}$

Prototypical example: degree $k - 1$ polynomial

$\begin{align*}
\forall u = b \text{ prime; } \\
\text{choose } a_0, a_1, \ldots, a_{k-1} \text{ randomly in } [u]; \\
h(x) = (a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}) \mod u.
\end{align*}$
Carter & Wegman (1977)

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

Family $\mathcal{H} = \{ h : [u] \rightarrow [b] \}$ \textit{k-independent} iff for random $h \in \mathcal{H}$:

$\quad (\forall) x \in [u], h(x) \text{ is uniform in } [b]$;

$\quad (\forall) x_1, \ldots, x_k \in [u], h(x_1), \ldots, h(x_k) \text{ are independent}.$

Prototypical example: degree $k - 1$ polynomial

$\quad u = b \text{ prime}$;

$\quad \text{choose } a_0, a_1, \ldots, a_{k-1} \text{ randomly in } [u]$;

$\quad h(x) = (a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}) \mod u.$

Many solutions for $k$-independent hashing proposed, but generally slow for $k > 3$ and too slow for $k > 5$. 
How much independence needed?

<table>
<thead>
<tr>
<th>Method</th>
<th>Independence needed</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chaining</td>
<td>$E[t] = O(1)$</td>
<td>$E[t^k] = O(1)$</td>
</tr>
<tr>
<td></td>
<td>$t = O\left(\frac{\lg n}{\lg \lg n}\right)$ w.h.p.</td>
<td>$2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2k + 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Theta\left(\frac{\lg n}{\lg \lg n}\right)$</td>
</tr>
<tr>
<td>Linear probing</td>
<td>$\leq 5$</td>
<td>[Pagh², Ružić’07]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\geq 5$</td>
</tr>
<tr>
<td>Cuckoo hashing</td>
<td>$O(\lg n)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\geq 6$</td>
</tr>
<tr>
<td>$F_2$ estimation</td>
<td>$4$</td>
<td>[Alon, Mathias, Szegedy’99]</td>
</tr>
<tr>
<td>$\varepsilon$-minwise indep.</td>
<td>$O\left(\lg \frac{1}{\varepsilon}\right)$</td>
<td>[Indyk’99]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Omega\left(\lg \frac{1}{\varepsilon}\right)$</td>
</tr>
</tbody>
</table>
How much independence needed?

<table>
<thead>
<tr>
<th>Method</th>
<th>Independence Needed</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chaining</td>
<td>$E[t] = O(1)$</td>
<td>$2k + 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$E[t^k] = O(1)$</td>
<td>$O(\frac{\lg n}{\lg \lg n})$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t = O\left(\frac{\lg n}{\lg \lg n}\right)$ w.h.p.</td>
<td>$\Theta\left(\frac{\lg n}{\lg \lg n}\right)$</td>
<td></td>
</tr>
<tr>
<td>Linear probing</td>
<td>$\leq 5$</td>
<td></td>
<td>$\geq 5$</td>
</tr>
<tr>
<td>Cuckoo hashing</td>
<td>$O(\lg n)$</td>
<td></td>
<td>$\geq 6$</td>
</tr>
<tr>
<td>$F_2$ estimation</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\varepsilon$-minwise indep.</td>
<td>$O\left(\lg \frac{1}{\varepsilon}\right)$</td>
<td></td>
<td>$\Omega\left(\lg \frac{1}{\varepsilon}\right)$</td>
</tr>
</tbody>
</table>

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 $x_1, \ldots, x_c$, 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 $4 \times 8$-bit characters.
- For $i = 1, ..., c$, we have truly random hash table:
  \[ R_i : \text{char} \rightarrow \text{hash values (bit strings)} \]
Simple tabulation

- **Simple tabulation** goes back to Carter and Wegman ’77.
- Key $x$ divided into $c = O(1)$ characters $x_1, \ldots, x_c$, e.g., 32-bit key as $4 \times 8$-bit characters.
- For $i = 1, \ldots, c$, we have truly random hash table:
  \[ R_i : \text{char} \rightarrow \text{hash values (bit strings)} \]
- Hash value
  \[
  h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]
  \]
Simple tabulation

- **Simple tabulation** goes back to Carter and Wegman’77.
- Key $x$ divided into $c = O(1)$ characters $x_1, \ldots, x_c$, e.g., 32-bit key as $4 \times 8$-bit characters.
- For $i = 1, \ldots, c$, we have truly random hash table:
  $$R_i : \text{char} \rightarrow \text{hash values (bit strings)}$$
- Hash value
  $$h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]$$
- 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 $x_1, \ldots, x_c$, e.g., 32-bit key as $4 \times 8$-bit characters.
- For $i = 1, \ldots, c$, we have truly random hash table:
  $$R_i : \text{char} \rightarrow \text{hash values (bit strings)}$$
- Hash value
  $$h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]$$
- 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 $x_1, \ldots, x_c$, e.g., 32-bit key as $4 \times 8$-bit characters.
- For $i = 1, \ldots, c$, we have truly random hash table:
  $R_i : \text{char} \rightarrow \text{hash values (bit strings)}$
- Hash value
  $$h(x) = R_1[x_1] \oplus \cdots \oplus R_c[x_c]$$
- 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: $h(a_1 a_2) \oplus h(a_1 b_2) \oplus h(b_1 a_2) \oplus h(b_1 b_2)$
  $$= (R_1[a_1] \oplus R_2[a_2]) \oplus (R_1[a_1] \oplus R_2[b_2]) \oplus (R_1[b_1] \oplus R_2[a_2]) \oplus (R_1[b_1] \oplus R_2[b_2]) = 0.$$
**How much independence needed? Wrong question**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>2k + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chaining</strong></td>
<td>( \mathbb{E}[t] = O(1) )</td>
<td>( \mathbb{E}[t^k] = O(1) )</td>
</tr>
<tr>
<td></td>
<td>( t = O(\frac{\log n}{\log \log n}) ) w.h.p.</td>
<td>( \Theta(\frac{\log n}{\log \log n}) )</td>
</tr>
</tbody>
</table>

- **Linear probing**: \( \leq 5 \) [Pagh², Ružić’07] \( \geq 5 \) [PT ICALP’10]
- **Cuckoo hashing**: \( O(\log n) \) \( \geq 6 \) [Cohen, Kane’05]
- **\( F_2 \) estimation**: \( 4 \) [Alon, Mathias, Szegedy’99]
- **\( \varepsilon \)-minwise indep.**: \( O(\log \frac{1}{\varepsilon}) \) [Indyk’99] \( \Omega(\log \frac{1}{\varepsilon}) \) [PT ICALP’10]
How much independence needed? Wrong question

<table>
<thead>
<tr>
<th>Chaining $\mathbb{E}[t] = O(1)$</th>
<th>$2$</th>
<th>(2k + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{E}[t^k] = O(1)$</td>
<td>$\Theta\left(\frac{\lg n}{\lg \lg n}\right)$</td>
<td>(\Theta\left(\frac{\lg n}{\lg \lg n}\right))</td>
</tr>
<tr>
<td>$t = O\left(\frac{\lg n}{\lg \lg n}\right)$ w.h.p.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linear probing</th>
<th>$\leq 5$</th>
<th>$\geq 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq 5$</td>
<td>[Pagh², Ružić'07]</td>
<td>[PT ICALP’10]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cuckoo hashing</th>
<th>$O(\lg n)$</th>
<th>$\geq 6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq 6$</td>
<td>[Cohen, Kane'05]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$F_2$ estimation</th>
<th>$4$</th>
<th>[Alon, Mathias, Szegedy'99]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$\varepsilon$-minwise indep.</th>
<th>$O(\lg \frac{1}{\varepsilon})$</th>
<th>$\Omega(\lg \frac{1}{\varepsilon})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq 6$</td>
<td>[Indyk'99]</td>
<td>[PT ICALP’10]</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Chaining $E[t] = O(1)$</th>
<th>$2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E[t^k] = O(1)$</td>
<td>$2k + 1$</td>
</tr>
<tr>
<td>$t = O(\frac{\lg n}{\lg \lg n})$ w.h.p.</td>
<td>$\Theta(\frac{\lg n}{\lg \lg n})$</td>
</tr>
<tr>
<td>Linear probing</td>
<td>$\leq 5$ [Pagh², Ružić’07]</td>
</tr>
<tr>
<td></td>
<td>$\geq 5$ [PT ICALP’10]</td>
</tr>
<tr>
<td>Cuckoo hashing</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td></td>
<td>$\geq 6$ [Cohen, Kane’05]</td>
</tr>
<tr>
<td>$F_2$ estimation</td>
<td>$4$ [Alon, Mathias, Szegedy’99]</td>
</tr>
<tr>
<td>$\varepsilon$-minwise indep.</td>
<td>$O(\lg \frac{1}{\varepsilon})$ [Indyk’99]</td>
</tr>
<tr>
<td></td>
<td>$\Omega(\lg \frac{1}{\varepsilon})$ [PT ICALP’10]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Independence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chaining $E[t] = O(1)$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$2k + 1$</td>
</tr>
<tr>
<td>$E[t^k] = O(1)$</td>
<td></td>
</tr>
<tr>
<td>$t = O\left(\frac{\lg n}{\lg \lg n}\right)$ w.h.p.</td>
<td>$\Theta\left(\frac{\lg n}{\lg \lg n}\right)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technique</th>
<th>Independence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear probing</td>
<td>$\leq 5$</td>
</tr>
<tr>
<td></td>
<td>[Pagh², Ružič'07]</td>
</tr>
<tr>
<td></td>
<td>$\geq 5$</td>
</tr>
<tr>
<td></td>
<td>[PT ICALP'10]</td>
</tr>
<tr>
<td>Cuckoo hashing</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td></td>
<td>$\geq 6$</td>
</tr>
<tr>
<td></td>
<td>[Cohen, Kane'05]</td>
</tr>
<tr>
<td>$F_2$ estimation</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>[Alon, Mathias, Szegedy'99]</td>
</tr>
<tr>
<td>$\varepsilon$-minwise indep.</td>
<td>$O(\lg \frac{1}{\varepsilon})$</td>
</tr>
<tr>
<td></td>
<td>$\Omega(\lg \frac{1}{\varepsilon})$</td>
</tr>
</tbody>
</table>

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 $\mu = \mathbb{E}[X_q] = \frac{n}{m}$. The following probability bounds hold for any constant $\gamma$:

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

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

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

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

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 \( \varepsilon, \gamma \), 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 $\varepsilon, \gamma$, 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 $\varepsilon, \gamma$, 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 \( \varepsilon, \gamma \), 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 \( \varepsilon, \gamma \), 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 \( \varepsilon, \gamma \), 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 \( \varepsilon, \gamma \), 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 $\varepsilon, \gamma$, 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 $|U| \geq (1 + \gamma)/\varepsilon$. □
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 $\varepsilon, \gamma$, 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 $(n u) < n u$ sets $U$ of $u$ keys to consider.

▶ Each such $U$ hash to one bin with probability $1/m^{u-1}$.

▶ Probability bound over all $U$ is $n u m^{u-1} \leq m (1 - \varepsilon) u + 1 - u = m^{1-\varepsilon} u = m^{-\gamma}$. 
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 $\varepsilon, \gamma$, 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 \( \varepsilon, \gamma \), 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.
- Each such \( U \) hash to one bin with probability \( 1/m^{u-1} \).
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 $\varepsilon$, $\gamma$, 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.
- Each such $U$ hash to one bin with probability $1/m^{u-1}$.
- Probability bound over all $U$ is

$$n^u m^{u-1} \leq m^{(1-\varepsilon)u+1-u} = m^{1-\varepsilon u} = m^{-\gamma}.$$
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 $\varepsilon, \gamma$, 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 $m^{-\gamma}$. □
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 \(\geq n^{1-1/c}\) 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 \(< (n^{1/c})^c = n.\)
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 $|G(i, a)| \leq n^{1-1/c}$. □
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 \(|G_{(i,a)}| \leq n^{1-1/c}\). □

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 $|G_{(i,a)}| \leq n^{1-1/c}$. □

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 $|G(i,a)| \leq n^{1-1/c}$. □

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 \(|G_{(i,a)}| \leq n^{1-1/c}\). □

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 \(|G_{(i,a)}| \leq n^{1-1/c}\). □

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 \(|G_{(i,a)}| \leq n^{1-1/c}.\) □

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 \(|G(i,a)| \leq n^{1-1/c}\). □

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 \(\mathbb{E}[X_{G(i,a)}] = |X_{G(i,a)}|/m\). 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 \(|G_{(i,a)}| \leq n^{1-1/c}\).

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 \(\mathbb{E}[X_{G_{(i,a)}}] = |X_{G_{(i,a)}}|/m\). 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

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

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

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.
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.

\[ x \mapsto \begin{array}{c}
\text{z} \\
\bullet \\
\bullet \\
y \\
x \\
\bullet \\
r
\end{array} \quad x \mapsto \begin{array}{c}
\bullet \\
\text{s} \\
\text{w} \\
f \\
\bullet \\
\text{a} \\
\text{b}
\end{array} \]

**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

<table>
<thead>
<tr>
<th>Hashing random keys</th>
<th>32-bit computer</th>
<th>64-bit computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>hashing scheme</td>
<td>hashing time (ns)</td>
</tr>
<tr>
<td>32</td>
<td>univ-mult-shift $(a \times x) &gt;&gt; s$</td>
<td>1.87</td>
</tr>
<tr>
<td>32</td>
<td>2-indep-mult-shift</td>
<td>5.78</td>
</tr>
<tr>
<td>32</td>
<td>5-indep-Mersenne-prime</td>
<td>99.70</td>
</tr>
<tr>
<td>32</td>
<td>5-indep-TZ-table</td>
<td>10.12</td>
</tr>
<tr>
<td>32</td>
<td>simple-table</td>
<td>4.98</td>
</tr>
<tr>
<td>64</td>
<td>univ-mult-shift</td>
<td>7.05</td>
</tr>
<tr>
<td>64</td>
<td>2-indep-mult-shift</td>
<td>22.91</td>
</tr>
<tr>
<td>64</td>
<td>5-indep-Mersenne-prime</td>
<td>241.99</td>
</tr>
<tr>
<td>64</td>
<td>5-indep-TZ-table</td>
<td>75.81</td>
</tr>
<tr>
<td>64</td>
<td>simple-table</td>
<td>15.54</td>
</tr>
</tbody>
</table>

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 95% of the time, but systematic terrible performance 5% 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 95% of the time, but systematic terrible performance 5% 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 95% of the time, but systematic terrible performance 5% 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 95% of the time, but systematic terrible performance 5% 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 95% of the time, but systematic terrible performance 5% 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: 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: $h = R_1[x_1] \oplus \cdots \oplus R_{c-1}[x_{c-1}]; h(x) = h \oplus R_c[(\text{char } h) \oplus x_c]$

- Twisted tabulation also implements Chernoff bounds: $0$-$1$ variables $X_i$ where $X_i = 1$ with probability $p_i$. Hashes uniformly in $[0, 1]$, set $X_i = 1$ if $h(i) < p_i$.

- First realistic scheme with exponential concentration of $\sum X_i$ around mean $\sum p_i$. 
Work in progress: 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: 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: 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:

$$h = R_1[x_1] \oplus \cdots \oplus R_{c-1}[x_{c-1}]; \quad h(x) = h \oplus R_c[((\text{char})h) \oplus x_c]$$
Work in progress: 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:

$$h = R_1[x_1] \oplus \cdots \oplus R_{c-1}[x_{c-1}]; h(x) = h \oplus R_c[((\text{char})h) \oplus x_c]$$

- Twisted tabulation also implements Chernoff bounds: 0-1 variables $X_i$ where $X_i = 1$ with probability $p_i$. Hashes uniformly in $[0, 1]$, set $X_i = 1$ if $h(i) < p_i$. First realistic scheme with exponential concentration of $\sum_i X_i$ around mean $\sum_i p_i$. 
Open problems

- Take any application using abstract truly 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 $i$th 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.
Open problems

- Take any application using abstract truly 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 $i$th 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.
Open problems

- Take any application using abstract truly 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 \( i \)th 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 truly 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 $i$th 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 truly 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 $i$th 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 truly 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 $i$th 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 truly 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 $i$th 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.