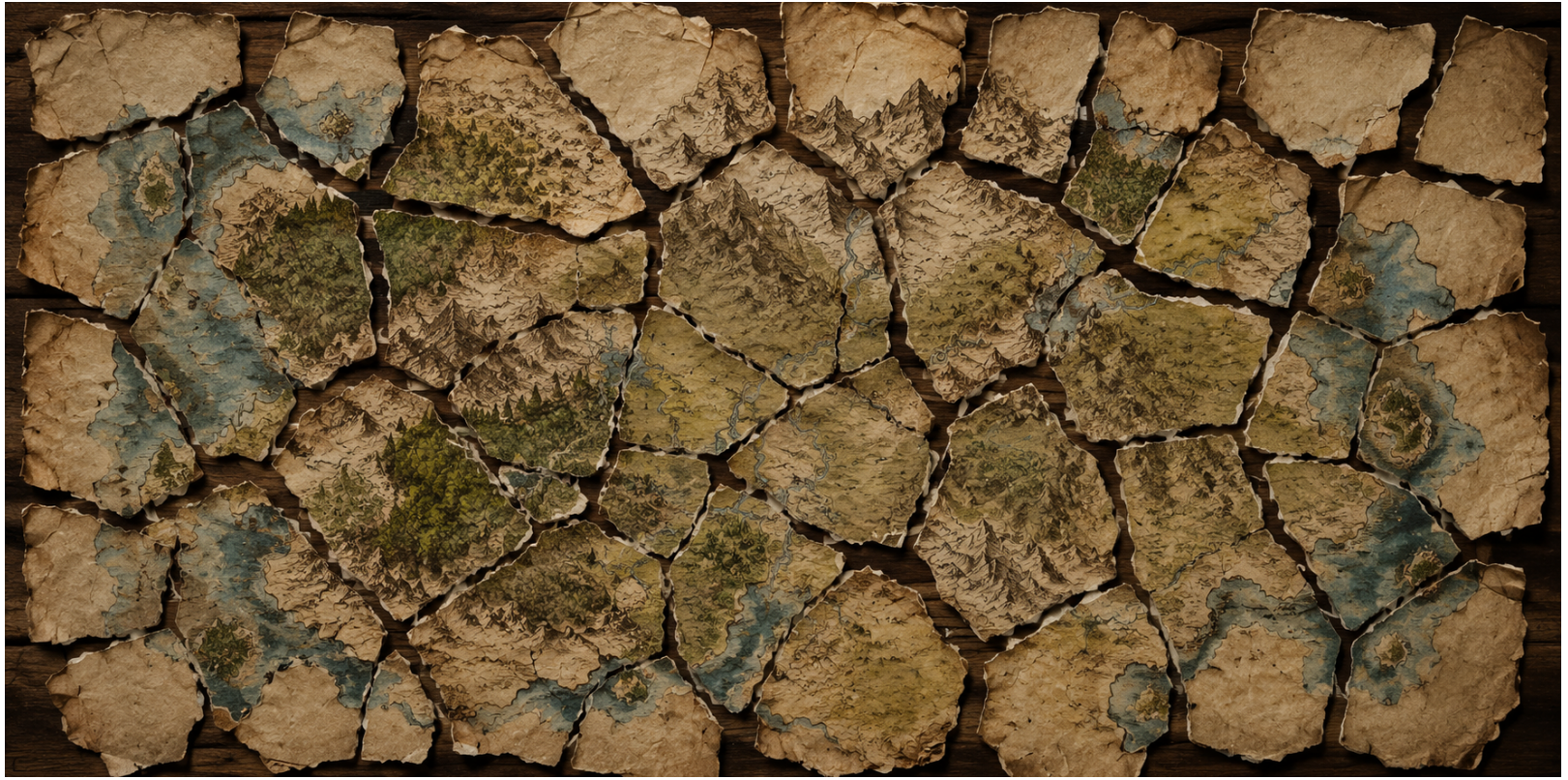
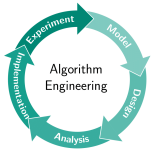


Hash Tables IV

Stefan Walzer, Ragnar Groot Koerkamp, Stefan Hermann | compiled on 27. Juli 2026



Hash table alternatives

Hash table are all-rounders

Hash tables support many operations.
In particular, S may change dynamically.

	Hash Table
Construct on $f : S \to \{0,1\}^r$	✓
Query an existing key	✓
Update the value of an existing key	✓
insert, delete, contains	✓
Space lower bound	$\log_2 \binom{\lvert D \rvert}{n} + nr$

Techniques on n keys $S \subseteq D$ mapping to r bit values.

x	$f(x)$
Bob	00
Alice	01
Carol	11
Dave	11
Peter	10

Example $r = 2$
(e.g. blood type)

Hash table alternatives

Hash table are all-rounders

Hash tables support many operations.
In particular, S may change dynamically.

Specialized data structures

For static S we can use a data structure that is simpler, more space efficient and often faster.

	Hash Table
Construct on $f : S \to \{0,1\}^r$	✓
Query an existing key	✓
Update the value of an existing key	✓
insert, delete, contains	✓
Space lower bound	$\log_2 \binom{\lvert D \rvert}{n} + nr$

Techniques on n keys $S \subseteq D$ mapping to r bit values.

x	$f(x)$
Bob	00
Alice	01
Carol	11
Dave	11
Peter	10

Example $r = 2$
(e.g. blood type)

Hash table alternatives

Hash table are all-rounders

Hash tables support many operations.
In particular, S may change dynamically.

Specialized data structures

For static S we can use a data structure that is simpler, more space efficient and often faster.

	Hash Table	**Updatable Retrieval**
Construct on $f : S \to \{0,1\}^r$	✓	✓
Query an existing key	✓	✓
Update the value of an existing key	✓	✓
insert, delete, contains	✓	✗
Space lower bound	$\log_2 \binom{\lvert D\rvert}{n} + nr$	$n \log_2(e) + nr$

Techniques on n keys $S \subseteq D$ mapping to r bit values.

x	$f(x)$
Bob	00
Alice	01
Carol	11
Dave	11
Peter	10

Example $r = 2$
(e.g. blood type)

Hash table alternatives

Hash table are all-rounders

Hash tables support many operations.
In particular, S may change dynamically.

Specialized data structures

For static S we can use a data structure that is simpler, more space efficient and often faster.

	Hash Table	**Updatable Retrieval**	**Static Retrieval**
Construct on $f : S \to \{0,1\}^r$	✓	✓	✓
Query an existing key	✓	✓	✓
Update the value of an existing key	✓	✓	✗
insert, delete, contains	✓	✗	✗
Space lower bound	$\log_2 \binom{\lvert D\rvert}{n} + nr$	$n \log_2(e) + nr$	nr

Techniques on n keys $S \subseteq D$ mapping to r bit values.

x	$f(x)$
Bob	00
Alice	01
Carol	11
Dave	11
Peter	10

Example $r = 2$
(e.g. blood type)

The (Updatable) Retrieval Problem

The retrieval data type

construct(f):

input: function $f : S \to \{0,1\}^r$
where $S \subseteq D$ has size $n = |S|$

output: data structure R.

The (Updatable) Retrieval Problem

The retrieval data type

construct(f):
input: function $f : S \to \{0,1\}^r$
where $S \subseteq D$ has size $n = |S|$
output: data structure R.
query$_R(x)$:
input: $x \in D$
output: some value in $\{0,1\}^r$
requirement: **query**$_R(x) = f(x)$ for all $x \in S$

The (Updatable) Retrieval Problem

The retrieval data type

construct(f):
input: function $f : S \to \{0,1\}^r$
where $S \subseteq D$ has size $n = |S|$
output: data structure R.
query$_R(x)$:
input: $x \in D$
output: some value in $\{0,1\}^r$
requirement: **query**$_R(x) = f(x)$ for all $x \in S$

The **updatable** retrieval data type

Also supports
update$_R(x, v)$:
input: $x \in S, v \in \{0,1\}^r$ // $x \in D \setminus S$ is undefined
output: data structure R' with

$$f'(k) = \begin{cases} v, & \text{if } k = x \\ f(k), & \text{otherwise.} \end{cases}$$

The (Updatable) Retrieval Problem

The retrieval data type

construct(f):
input: function $f : S \to \{0,1\}^r$ where $S \subseteq D$ has size $n = |S|$
output: data structure R.
query$_R(x)$:
input: $x \in D$
output: some value in $\{0,1\}^r$
requirement: **query**$_R(x) = f(x)$ for all $x \in S$

The **updatable** retrieval data type

Also supports
update$_R(x, v)$:
input: $x \in S, v \in \{0,1\}^r$ // $x \in D \setminus S$ is undefined
output: data structure R' with

$$f'(k) = \begin{cases} v, & \text{if } k = x \\ f(k), & \text{otherwise.} \end{cases}$$

The price to pay

- R cannot be used to decide "is $x \in S$?"
- **query**$_R(x)$ is *unspecified* if $x \notin S$.

Motivation For Retrieval: Router Forwarding Table

- The forwarding table in a router maps IP addresses to ports.

Motivation For Retrieval: Router Forwarding Table

- The forwarding table in a router maps IP addresses to ports.
- We want GB/s internet, so this needs to have very fast queries.

Motivation For Retrieval: Router Forwarding Table

- The forwarding table in a router maps IP addresses to ports.
- We want GB/s internet, so this needs to have very fast queries.
- Therefore, expensive SRAM hardware is used - space efficiency becomes crucial.

Motivation For Retrieval: Router Forwarding Table

- The forwarding table in a router maps IP addresses to ports.
- We want GB/s internet, so this needs to have very fast queries.
- Therefore, expensive SRAM hardware is used - space efficiency becomes crucial.
- An IP address needs 128 bit but a port only $\approx$ 10 bits.

Motivation For Retrieval: Router Forwarding Table

- The forwarding table in a router maps IP addresses to ports.
- We want GB/s internet, so this needs to have very fast queries.
- Therefore, expensive SRAM hardware is used - space efficiency becomes crucial.
- An IP address needs 128 bit but a port only $\approx$ 10 bits.
- We do not care about IPs that are not in the forwarding table (handled otherwise).

Motivation For Retrieval: Router Forwarding Table

- The forwarding table in a router maps IP addresses to ports.
- We want GB/s internet, so this needs to have very fast queries.
- Therefore, expensive SRAM hardware is used - space efficiency becomes crucial.
- An IP address needs 128 bit but a port only $\approx$ 10 bits.
- We do not care about IPs that are not in the forwarding table (handled otherwise).
- A retrieval data structure is the perfect fit!

Updatable Retrieval

Possible design

- The lower bound says that at least $n \log_2(e) + nr$ bits are needed.
- The nr bits hint that maybe we can just use an array of length n containing r bit values.

Updatable Retrieval

Possible design

- The lower bound says that at least $n \log_2(e) + nr$ bits are needed.
- The nr bits hint that maybe we can just use an array of length n containing r bit values.
- How can we encode which key maps to which array entry?
- Can it be done in just $n \log_2(e)$?

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively).

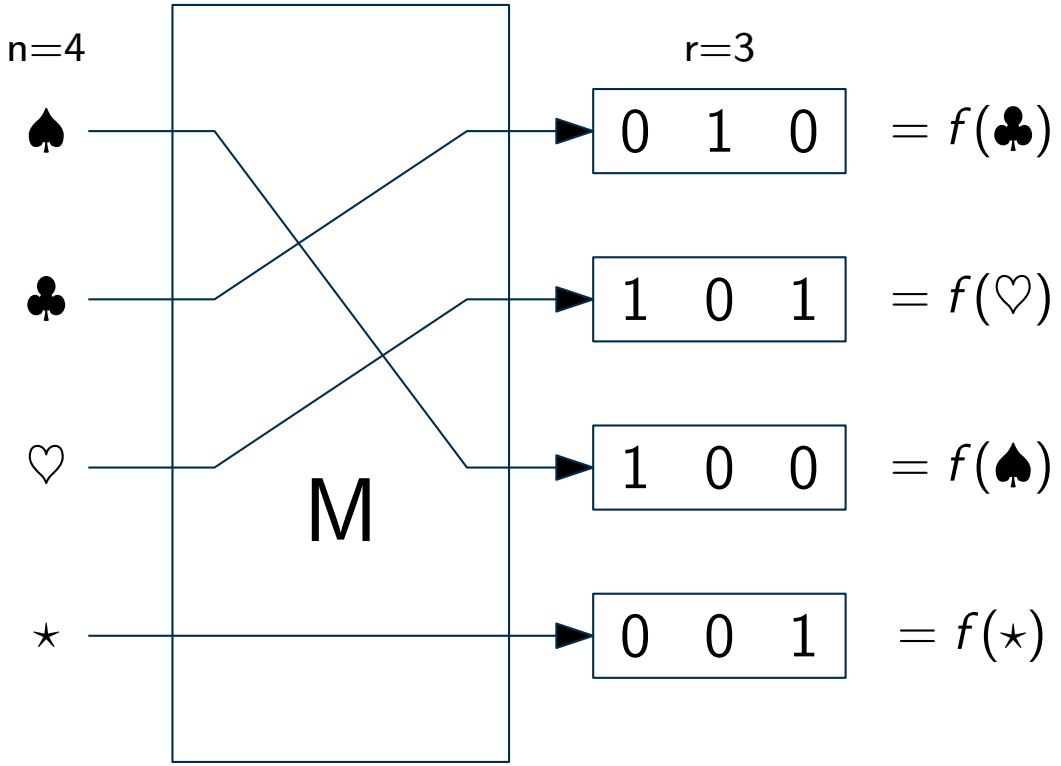
Updatable Retrieval

Possible design

- The lower bound says that at least $n \log_2(e) + nr$ bits are needed.
- The nr bits hint that maybe we can just use an array of length n containing r bit values.
- How can we encode which key maps to which array entry?
- Can it be done in just $n \log_2(e)$?

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively).



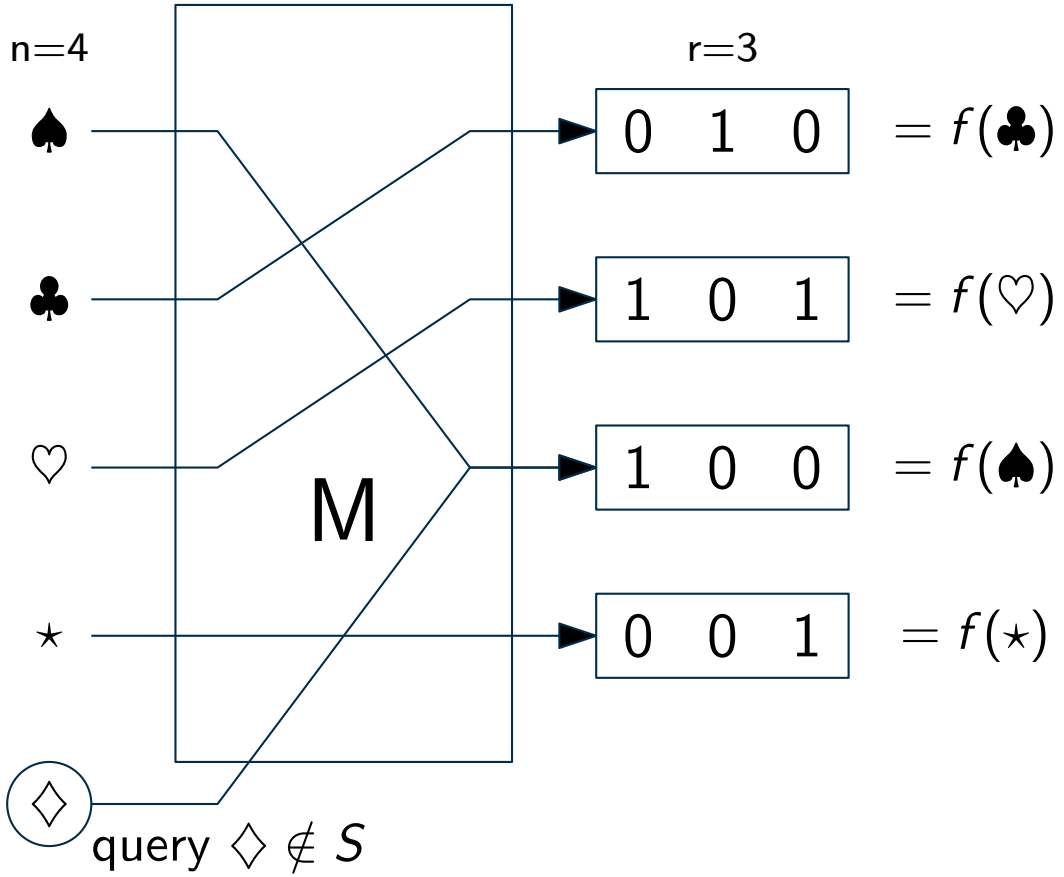
Updatable Retrieval

Possible design

- The lower bound says that at least $n \log_2(e) + nr$ bits are needed.
- The nr bits hint that maybe we can just use an array of length n containing r bit values.
- How can we encode which key maps to which array entry?
- Can it be done in just $n \log_2(e)$?

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively).

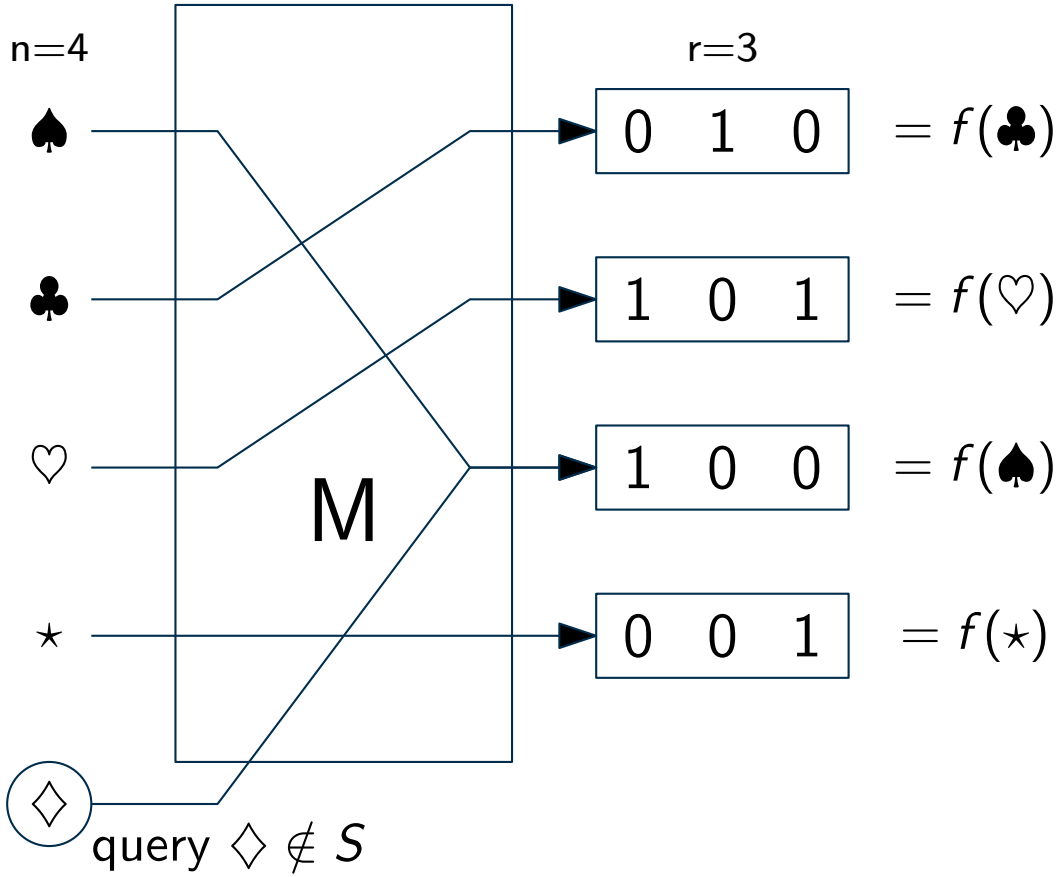


Updatable Retrieval

Algorithm construct($f : S \to \{0,1\}^r$)**:**
- construct M for S
- allocate A, length n, r bit values
- **for** $x \in S$ **do**
  - $A[M.query(x)] = f(x)$

Algorithm query($x \in D$)**:**
- **return** $A[M.query(x)]$

Algorithm update($x \in S, v \in \{0,1\}^r$)**:**
- $A[M.query(x)] = v$



Updatable Retrieval

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively). Ideally, the space should be close to $n \log_2(e)$.

A space inefficient construction

Just assign each key a unique value in $[n]$ and store the mapping in a hash table. However, just storing the values would already need $\mathcal{O}(n \log n)$ bits!

Updatable Retrieval

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively). Ideally, the space should be close to $n \log_2(e)$.

A space inefficient construction

Just assign each key a unique value in $[n]$ and store the mapping in a hash table. However, just storing the values would already need $\mathcal{O}(n \log n)$ bits!

A slow but space efficient construction

Search for a seed of a hash function:

Algorithm construct(S)**:**

```
for s = 0, 1, ... do
    if h_s is bijection from S to [n] then
        return s
```

Updatable Retrieval

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively). Ideally, the space should be close to $n \log_2(e)$.

A space inefficient construction

Just assign each key a unique value in $[n]$ and store the mapping in a hash table. However, just storing the values would already need $\mathcal{O}(n \log n)$ bits!

A slow but space efficient construction

Search for a seed of a hash function:

Algorithm construct(S)**:**

```
for s = 0, 1, ... do
    if h_s is bijection from S to [n] then
        return s
```

Algorithm query(x)**:**

```
return h_s(x)
```

Updatable Retrieval

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively). Ideally, the space should be close to $n \log_2(e)$.

A space inefficient construction

Just assign each key a unique value in $[n]$ and store the mapping in a hash table. However, just storing the values would already need $\mathcal{O}(n \log n)$ bits!

A slow but space efficient construction

Search for a seed of a hash function:

Algorithm construct(S)**:**

```
for s = 0, 1, ... do
    if h_s is bijection from S to [n] then
        return s
```

Algorithm query(x)**:**

```
return h_s(x)
```

The probability that h_s is bijective on S is $\frac{n!}{n^n}$, because we draw one of n^n functions and only $n!$ of them are bijective on S. Hence, we need to test $\frac{n^n}{n!}$ seeds in expectation. Storing the seed requires $\log_2 \frac{n^n}{n!} + \mathcal{O}(1)$ bits [a]. Simplifying using Stirling's approximation gives $n \log_2(e) + \mathcal{O}(\log n)$ bits.

[a] using encoding techniques that are not covered here (e.g. Golomb)

Updatable Retrieval

What we need

We need a data structure M that maps a fixed set of n keys to $[n]$ without collisions (aka bijectively). Ideally, the space should be close to $n \log_2(e)$.

A space inefficient construction

Just assign each key a unique value in $[n]$ and store the mapping in a hash table. However, just storing the values would already need $\mathcal{O}(n \log n)$ bits!

Recent research at ITI Sanders

A minimal perfect hash function (this is what M is called in literature) constructed in linear time with near optimal space.

A slow but space efficient construction

Search for a seed of a hash function:

Algorithm construct(S)**:**

```
for s = 0, 1, ... do
    if h_s is bijection from S to [n] then
        return s
```

Algorithm query(x)**:**

```
return h_s(x)
```

The probability that h_s is bijective on S is $\frac{n!}{n^n}$, because we draw one of n^n functions and only $n!$ of them are bijective on S. Hence, we need to test $\frac{n^n}{n!}$ seeds in expectation. Storing the seed requires $\log_2 \frac{n^n}{n!} + \mathcal{O}(1)$ bits [a]. Simplifying using Stirling's approximation gives $n \log_2(e) + \mathcal{O}(\log n)$ bits.

[a] using encoding techniques that are not covered here (e.g. Golomb)

Hash table alternatives

Hash table are all-rounders

Hash tables support many operations.
In particular, S may change dynamically.

Specialized data structures

For static S we can use a data structure that is simpler, more space efficient and often faster.

	Hash Table	**Updatable Retrieval**	**Static Retrieval**
Construct on $f : S \to \{0,1\}^r$	✓	✓	✓
Query an existing key	✓	✓	✓
Update the value of an existing key	✓	✓	✗
insert, delete, contains	✓	✗	✗
Space lower bound	$\log_2 \binom{\lvert D\rvert}{n} + nr$	$n \log_2(e) + nr$	nr

Techniques on n keys $S \subseteq D$ mapping to r bit values.

x	$f(x)$
Bob	00
Alice	01
Carol	11
Dave	11
Peter	10

Example $r = 2$
(e.g. blood type)

Static Retrieval using Linear Algebra over the field $\mathbb{F}_2 = \{0, 1\}$

Static retrieval for $f : S \to \mathbb{F}_2^r$ with $|S| = n$

Pick $m \geq n$. Data structure is pair $R = (h : D \to \mathbb{F}_2^m, \vec{z} \in \mathbb{F}_2^{m \times r})$ such that $h(x)^T \cdot \vec{z} = f(x)$ for all $x \in S$.

$r = 3$
$m = 7$
$n = 5$

$$\underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}}_{\overset{h(x)}{=}} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{pmatrix} \overset{!}{=} f(x)$$

Static Retrieval using Linear Algebra over the field $\mathbb{F}_2 = \{0, 1\}$

Static retrieval for $f : S \to \mathbb{F}_2^r$ with $|S| = n$

Pick $m \geq n$. Data structure is pair $R = (h : D \to \mathbb{F}_2^m, \vec{z} \in \mathbb{F}_2^{m \times r})$ such that $h(x)^T \cdot \vec{z} = f(x)$ for all $x \in S$.

$r = 3$
$m = 7$
$n = 5$

$$\underset{h(x)^T}{\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}} \begin{pmatrix} 011 \\ \mathbf{010} \\ 000 \\ 001 \\ \mathbf{000} \\ \mathbf{110} \\ 101 \end{pmatrix} \overset{!}{=} 100$$

Static Retrieval using Linear Algebra over the field $\mathbb{F}_2 = \{0, 1\}$

Static retrieval for $f : S \to \mathbb{F}_2^r$ with $|S| = n$

Pick $m \geq n$. Data structure is pair $R = (h : D \to \mathbb{F}_2^m, \vec{z} \in \mathbb{F}_2^{m \times r})$ such that $h(x)^T \cdot \vec{z} = f(x)$ for all $x \in S$.

$r = 3$
$m = 7$
$n = 5$

$$\begin{pmatrix} \text{———} h(x_1) \text{———} \\ \text{———} h(x_2) \text{———} \\ \text{———} h(x_3) \text{———} \\ \text{———} h(x_4) \text{———} \\ \text{———} h(x_5) \text{———} \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ f(x_5) \end{pmatrix}$$

Static Retrieval using Linear Algebra over the field $\mathbb{F}_2 = \{0, 1\}$

Static retrieval for $f : S \to \mathbb{F}_2^r$ with $|S| = n$

Pick $m \geq n$. Data structure is pair $R = (h : D \to \mathbb{F}_2^m, \vec{z} \in \mathbb{F}_2^{m \times r})$ such that $h(x)^T \cdot \vec{z} = f(x)$ for all $x \in S$.

$r = 3$
$m = 7$
$n = 5$

$$\begin{pmatrix} & & & & & & \\ \text{---} & h(x_1) & \text{---} & & & & \\ \text{---} & h(x_2) & \text{---} & & & & \\ \text{---} & h(x_3) & \text{---} & & & & \\ \text{---} & h(x_4) & \text{---} & & & & \\ \text{---} & h(x_5) & \text{---} & & & & \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ f(x_5) \end{pmatrix}$$

A simple instantitation

Use $m = n(1 + \epsilon)$ and $h \sim \mathcal{U}\left(\mathbb{F}_2^{r\,D}\right)$ then the system is solvable whp.

Space	rn
Construction	$\mathcal{O}(n^3)$
Query	$\mathcal{O}(n)$

Static Retrieval using Linear Algebra over the field $\mathbb{F}_2 = \{0, 1\}$

Static retrieval for $f : S \to \mathbb{F}_2^r$ with $|S| = n$

Pick $m \geq n$. Data structure is pair $R = (h : D \to \mathbb{F}_2^m, \vec{z} \in \mathbb{F}_2^{m \times r})$ such that $h(x)^T \cdot \vec{z} = f(x)$ for all $x \in S$.

$r = 3$
$m = 7$
$n = 5$

$$\begin{pmatrix} & & \\ \text{——}h(x_1)\text{——} & & \\ \text{——}h(x_2)\text{——} & & \\ \text{——}h(x_3)\text{——} & & \\ \text{——}h(x_4)\text{——} & & \\ \text{——}h(x_5)\text{——} & & \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ f(x_5) \end{pmatrix}$$

A simple instantitation

Use $m = n(1 + \epsilon)$ and $h \sim \mathcal{U}\left(\mathbb{F}_2^{rD}\right)$ then the system is solvable whp.

Space	rn
Construction	$\mathcal{O}(n^3)$
Query	$\mathcal{O}(n)$

Current Research at ITI Sanders

Choosing h involves a trade-off and is non-trivial. How can we achieve fast construction, fast queries and space efficiency simultaneously?

Possible Exam Questions

- What is a (updatable) retrieval data structure?
- What are the respective advantages compared to a hash table? What are the disadvantages?
- How can we construct (updatable) retrieval data structures?