Hash tables
A **dictionary** is a data-structure that stores a set of elements where each element has a unique **key**, and supports the following operations:

- **Search**\((S, k)\)  Return the element whose key is \(k\).
- **Insert**\((S, x)\)  Add \(x\) to \(S\).
- **Delete**\((S, x)\)  Remove \(x\) from \(S\).

- Assume that the keys are from \(U = \{0, \ldots, u - 1\}\).
Direct addressing

- $S$ is stored in an array $T[0..u - 1]$. The entry $T[k]$ contains a pointer to the element with key $k$, if such element exists, and NULL otherwise.

- **SEARCH($T$, $k$):** return $T[k]$.

- **INSERT($T$, $x$):** $T[x.key] \leftarrow x$.

- **DELETE($T$, $x$):** $T[x.key] \leftarrow \text{NULL}$.

- What is the problem with this structure?
In order to reduce the space complexity of direct addressing, map the keys to a smaller range \( \{0, \ldots, m - 1\} \) using a hash function.

There is a problem of collisions (two or more keys that are mapped to the same value).

There are several methods to handle collisions.
- Chaining
- Open addressing
Hash table with chaining

- Let $h : U \rightarrow \{0, \ldots, m - 1\}$ be a hash function ($m < u$).
- $S$ is stored in a table $T[0..m-1]$ of linked lists. The element $x \in S$ is stored in the list $T[h(x\text{.key})]$.
- **Search**($T$, $k$): Search the list $T[h(k)]$.
- **Insert**($T$, $x$): Insert $x$ at the head of $T[h(x\text{.key})]$.
- **Delete**($T$, $x$): Delete $x$ from the list containing $x$.

$S = \{6, 9, 19, 26, 30\}$

$m = 5$, $h(x) = x \mod 5$
• Assumption of simple uniform hashing: any element is equally likely to hash into any of the \( m \) slots, independently of where other elements have hashed into.

• The above assumption is true when the keys are chosen uniformly and independently at random (with repetitions), and the hash function satisfies
\[
|\{k \in U : h(k) = i\}| = u/m \text{ for every } i \in \{0, \ldots, m - 1\}.
\]

• We want to analyze the performance of hashing under the assumption of simple uniform hashing. This is the balls into bins problem.

• Suppose we randomly place \( n \) balls into \( m \) bins. Let \( X \) be the number of balls in bin 1.

• The time complexity of a random search in a hash table is \( \Theta(1 + X) \).
The expectation of $X$

- Each ball has probability $1/m$ to be in bin 1.
- The random variable $X$ has binomial distribution with parameters $n$ and $1/m$.
- Therefore, $E[X] = n/m$. 
The distribution of $X$

**Claim**

$$\Pr[X = r] \approx \frac{e^{-\alpha} \alpha^r}{r!},$$

where $\alpha = n/m$ (i.e., $X$ has approximately Poisson distribution).

**Example**

If $\alpha = 1$,

- $\Pr[X = 0] \approx 0.368$
- $\Pr[X = 1] \approx 0.368$
- $\Pr[X = 2] \approx 0.184$
- $\Pr[X = 3] \approx 0.061$
- $\Pr[X = 4] \approx 0.015$
- $\Pr[X = 5] \approx 0.003$
- $\Pr[X = 6] \approx 0.0005$
- $\Pr[X = 7] \approx 0.00007$

Hash tables
The distribution of $X$

Claim

$\Pr[X = r] \approx \frac{e^{-\alpha} \alpha^r}{r!}$, where $\alpha = n/m$ (i.e., $X$ has approximately Poisson distribution).

Proof.

$$\Pr[X = r] = \binom{n}{r} \left(\frac{1}{m}\right)^r \left(1 - \frac{1}{m}\right)^{n-r}$$

$$= \frac{n(n-1) \cdots (n-r+1)}{r!} \frac{1}{m^r} \left(1 - \frac{1}{m}\right)^{n-r}$$

If $m$ and $n$ are large, $n(n-1) \cdots (n-r+1) \approx n^r$ and $(1 - \frac{1}{m})^{n-r} \approx e^{-n/m}$. Thus, $\Pr[X = r] \approx \frac{e^{-n/m}(n/m)^r}{r!}$. 

Hash tables
Suppose that $n = m = 10^6$. Most bins contain 0–3 balls.

The probability that a specific bin contains at least 8 balls is $\approx 0.000009$.

However, it is very likely that some bin will contain 8 balls.

**Theorem**

If $m = \Theta(n)$ then the size of the largest bin is $\Theta(\log n / \log \log n)$ with probability at least $1 - 1/n^{\Theta(1)}$. 
We wish to maintain \( n = O(m) \) in order to have \( \Theta(1) \) search time. This can be achieved by rehashing. Suppose we want \( n \leq m \). When the table has \( m \) elements and a new element is inserted, create a new table of size \( 2m \) and copy all elements into the new table.

\[
h(x) = x \mod 5
\]

\[
h'(x) = x \mod 10
\]
We wish to maintain \( n = O(m) \) in order to have \( \Theta(1) \) search time.

This can be achieved by rehashing. Suppose we want \( n \leq m \). When the table has \( m \) elements and a new element is inserted, create a new table of size \( 2m \) and copy all elements into the new table.

The cost of rehashing is \( \Theta(n) \).
Universal hash functions

- **Random** keys + fixed hash function (e.g. mod)
  \[ \Rightarrow \text{The hashed keys are random numbers.} \]

- A fixed set of keys + **random** hash function (selected from a universal set)
  \[ \Rightarrow \text{The hashed keys are semi-random numbers.} \]
Definition

A collection $\mathcal{H}$ of hash functions is a universal if for every pair of distinct keys $x, y \in U$, $\Pr_{h \in H}[h(x) = h(y)] \approx \frac{1}{m}$.

Example

Let $p$ be a prime number larger than $u$.

$f_{a, b}(x) = ((ax + b) \mod p) \mod m$

$\mathcal{H}_{p, m} = \{ f_{a, b} | a \in \{1, 2, \ldots, p - 1\}, b \in \{0, 1, \ldots, p - 1\} \}$
Universal hash functions

**Theorem**

Suppose that $\mathcal{H}$ is a universal collection of hash functions. If a hash table for $S$ is built using a randomly chosen $h \in \mathcal{H}$, then for every $k \in U$, the expected time of $\text{Search}(S, k)$ is $\Theta(1 + n/m)$.

**Proof.**

Let $X = \text{length of } T[h(k)]$.

$X = \sum_{y \in S} I_y$ where $I_y = 1$ if $h(y.\text{key}) = h(k)$ and $I_y = 0$ otherwise.

$$E[X] = E \left[ \sum_{y \in S} I_y \right] = \sum_{y \in S} E[I_y] = \sum_{y \in S} \Pr_{h \in \mathcal{H}} [h(y.\text{key}) = h(k)]$$

$$\approx 1 + n \cdot \frac{1}{m}.$$
Under the assumption of simple uniform hashing, the expected time of a search is $\Theta(1 + \alpha)$ time.

If $\alpha = \Theta(1)$, and under the assumption of simple uniform hashing, the worst case time of a search is $\Theta(\log n / \log \log n)$, with probability at least $1 - 1/n^{\Theta(1)}$.

If the hash function is chosen from a universal collection at random, the expected time of a search is $\Theta(1 + \alpha)$.

The worst case time of insert is $\Theta(1)$ if there is no rehashing.
Interpreting keys as natural numbers

- How can we convert floats or ASCII strings to natural numbers?
- An ASCII string can be interpreted as a number in base 256.

**Example**

For the string CLRS, the ASCII values of the characters are C = 67, L = 76, R = 82, S = 83. So CLRS is 
\[(67 \cdot 256^3) + (76 \cdot 256^2) + (82 \cdot 256^1) + (83 \cdot 256^0) = 1129075283.\]
Horner’s rule

- Horner’s rule:
  \[ a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0 = \]
  \[ (\cdots ((a_d x + a_{d-1}) x + a_{d-2}) x + \cdots )x + a_0. \]

- Example:
  \[ 67 \cdot 256^3 + 76 \cdot 256^2 + 82 \cdot 256^1 + 83 \cdot 256^0 = \]
  \[ ((67 \cdot 256 + 76) \cdot 256 + 82) \cdot 256 + 83 \]

- If \( d \) is large the value of \( y \) is too big.
- Solution: evaluate the polynomial modulo \( p \)
  
  1. \( y \leftarrow a_d \)
  2. \( \text{for } i = d - 1 \text{ to } 0 \)
  3. \( y \leftarrow a_i + xy \mod p \)
Rabin-Karp pattern matching algorithm

- Suppose we are given strings $P$ and $T$, and we want to find all occurrences of $P$ in $T$.
- The Rabin-Karp algorithm is as follows:
  
  Compute $h(P)$ for every substring $T'$ of $T$ of length $|P|$.
  
  if $h(T') = h(P)$ check whether $T' = P$.
- The values $h(T')$ for all $T'$ can be computed in $\Theta(|T|)$ time using rolling hash.

Example

Let $T = \text{BGUACC}$, $P = \text{GUAB}$, $T_1 = \text{BGUA}$, $T_2 = \text{GUAB}$.

$h(T_1) = (66 \cdot 256^3 + 71 \cdot 256^2 + 85 \cdot 256 + 65) \mod p$

$h(T_2) = (71 \cdot 256^3 + 85 \cdot 256^2 + 65 \cdot 256 + 67) \mod p$

$= (h(T_1) - 66 \cdot 256^3) \cdot 256 + 67 \mod p$
Applications

- **Data deduplication**: Suppose that we have many files, and some files have duplicates. In order to save storage space, we want to store only one instance of each distinct file.

- **Distributed storage**: Suppose we have many files, and we want to store them on several servers.
Let $m$ denote the number of servers.

The simple solution is to use a hash function $h : U \rightarrow \{1, \ldots, m\}$, and assign file $x$ to server $h(x)$.

The problem with this solution is that if we add a server, we need to do rehashing which will move most files between servers.
Suppose that the server have identifiers $s_1, \ldots, s_m$.

Let $h : U \rightarrow [0, 1]$ be a hash function.

For each server $i$ associate a point $h(s_i)$ on the unit circle.

For each file $f$, assign $f$ to the server whose point is the first point encountered when traversing the unit cycle anti-clockwise starting from $h(f)$.
Suppose that the server have identifiers \( s_1, \ldots, s_m \).

Let \( h : U \rightarrow [0, 1] \) be a hash function.

For each server \( i \) associate a point \( h(s_i) \) on the unit circle.

For each file \( f \), assign \( f \) to the server whose point is the first point encountered when traversing the unit cycle anti-clockwise starting from \( h(f) \).
When a new server $m + 1$ is added, let $i$ be the server whose point is the first server point after $h(s_{m+1})$.

We only need to reassign some of the files that were assigned to server $i$.

The expected number of files reassignments is $n/(m + 1)$. 
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert element $k$, try inserting to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

**INSERT**($T$, $k$)

1. $i = 0$ to $m - 1$
2. $j \leftarrow h(k) + i \mod m$
3. if $T[j] = \text{NULL}$ OR ... 
4. $T[j] \leftarrow k$
5. return
6. error “hash table overflow”
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert element $k$, try inserting to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

**INSERT**($T$, $k$)

1. **for** $i = 0$ **to** $m - 1$
2. $j \leftarrow h(k) + i \mod m$
3. **if** $T[j] = \text{NULL}$ **OR** ... 
4. $T[j] \leftarrow k$
5. **return**
6. **error** “hash table overflow”
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert element \( k \), try inserting to \( T[h(k)] \). If \( T[h(k)] \) is not empty, try \( T[h(k) + 1 \mod m] \), then try \( T[h(k) + 2 \mod m] \) etc.

**\text{INSERT}(T, k)**

(1) \( \text{for } i = 0 \text{ to } m - 1 \)
(2) \( j \leftarrow h(k) + i \mod m \)
(3) \( \text{if } T[j] = \text{NULL } \text{OR } \ldots \)
(4) \( T[j] \leftarrow k \)
(5) \( \text{return} \)
(6) \( \text{error} \) “hash table overflow”
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert element $k$, try inserting to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

**Insert**($T,k$)

1. for $i = 0$ to $m - 1$
2. $j \leftarrow h(k) + i \mod m$
3. if $T[j] = \text{NULL}$ OR …
4. $T[j] \leftarrow k$
5. return
6. error “hash table overflow”
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert element $k$, try inserting to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

**Insert** $(T, k)$

1. **for** $i = 0$ **to** $m - 1$
2. $j \leftarrow h(k) + i \mod m$
3. **if** $T[j] = \text{NULL}$ **or** ...
4. $T[j] \leftarrow k$
5. **return**
6. **error** “hash table overflow”

```markdown
insert(T, 19)
```
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert element $k$, try inserting to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

**Insert** ($T, k$

1. **for** $i = 0$ **to** $m - 1$
2. $j \leftarrow h(k) + i \mod m$
3. **if** $T[j] = \text{NULL}$ **or** ...
4. $T[j] \leftarrow k$
5. **return**
6. **error** “hash table overflow”
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert element \( k \), try inserting to \( T[h(k)] \). If \( T[h(k)] \) is not empty, try \( T[h(k) + 1 \mod m] \), then try \( T[h(k) + 2 \mod m] \) etc.

**INSERT**\((T, k)\)

1. for \( i = 0 \) to \( m - 1 \)
2. \( j \leftarrow h(k) + i \mod m \)
3. if \( T[j] = \text{NULL} \) OR ...
4. \( T[j] \leftarrow k \)
5. return
6. error “hash table overflow”

---

<table>
<thead>
<tr>
<th>0</th>
<th>20</th>
<th>1</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>55</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>9</td>
<td>19</td>
</tr>
</tbody>
</table>
**Search**

\[
\text{Search}(T, k)
\]

1. for \( i = 0 \) to \( m - 1 \)
2. \( j \leftarrow h(k) + i \mod m \)
3. if \( T[j] = k \)
4. return \( j \)
5. if \( T[j] = \text{NULL} \)
6. return \( \text{NULL} \)
7. return \( \text{NULL} \)

---

**search**

\[
\text{search}(T, 55)
\]
**Searching**

**SEARCH**(T, k)

(1) **for** i = 0 **to** m − 1
(2) \( j \leftarrow h(k) + i \mod m \)
(3) **if** T[j] = k
(4) \* return j 
(5) **if** T[j] = NULL
(6) \* return NULL 
(7) \* return NULL

**search**(T, 25)

Hash tables
Delete method 1: To delete element $k$, store in $T[h(k)]$ a special value DELETED.
Delete method 2: Erase $k$ from the table (replace it by NULL) and also erase the consecutive block of elements following $k$. Then, reinsert the latter elements to the table.
Delete method 2: Erase $k$ from the table (replace it by NULL) and also erase the consecutive block of elements following $k$. Then, reinsert the latter elements to the table.
Delete method 2: Erase $k$ from the table (replace it by NULL) and also erase the consecutive block of elements following $k$. Then, reinsert the latter elements to the table.
Open addressing is a generalization of linear probing.

Let
\[ h : U \times \{0, \ldots, m - 1\} \rightarrow \{0, \ldots, m - 1\} \]
be a hash function such that
\( \{h(k, 0), h(k, 1), \ldots h(k, m - 1)\} \) is a permutation of \( \{0, \ldots, m - 1\} \) for every \( k \in U \).

The slots examined during search/insert are \( h(k, 0) \), then \( h(k, 1), h(k, 2) \) etc.

In the example on the right,
\[ h(k, i) = (h_1(k) + ih_2(k)) \mod 13 \]
where
\[ h_1(k) = k \mod 13 \]
\[ h_2(k) = 1 + (k \mod 11) \]
Open addressing

- Insertion is the same as in linear probing:
  
  \[ \text{INSERT}(T, k) \]
  
  (1) \textbf{for} \ i = 0 \textbf{ to } m - 1
  
  (2) \ j \leftarrow h(k, i)
  
  (3) \textbf{if} \ T[j] = \text{NULL OR} \ T[j] = \text{DELETED}
  
  (4) \ T[j] \leftarrow k
  
  (5) \textbf{return}
  
  (6) \textbf{error} “hash table overflow”

- Deletion is done using delete method 1 defined above (using special value DELETED).
In the **double hashing** method,

\[ h(k, i) = (h_1(k) + ih_2(k)) \mod m \]

for some hash functions \( h_1 \) and \( h_2 \).

- The value \( h_2(k) \) must be relatively prime to \( m \) for the entire hash table to be searched. This can be ensured by either
  - Taking \( m \) to be a power of 2, and the image of \( h_2 \) contains only odd numbers.
  - Taking \( m \) to be a prime number, and the image of \( h_2 \) contains integers from \( \{1, \ldots, m-1\} \).

- For example,
  \[ h_1(k) = k \mod m \]
  \[ h_2(k) = 1 + (k \mod m') \]

where \( m \) is prime and \( m' < m \).
Assume **uniform hashing**: the probe sequence of each key is equally likely to be any of the \( m! \) permutations of \( \{0, \ldots m-1\} \).

Assuming uniform hashing and no deletions, the expected number of probes in a search is

- At most \( \frac{1}{1-\alpha} \) for unsuccessful search.
- At most \( \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \) for successful search.
Comparison of hash table methods – Search

Successful Lookup

- Chained Hashing
- Linear Probing

Clock Cycles vs. log n

Hash tables
Comparison of hash table methods – Search

Unsuccessful Lookup

- Cuckoo
- Two-Way Chaining
- Chained Hashing
- Linear Probing

Hash tables
Comparison of hash table methods – Insert

![Graph showing comparison of hash table methods: Cuckoo, Two-Way Chaining, Chained Hashing, Linear Probing. The x-axis represents \( \log n \) and the y-axis represents clock cycles. The graph shows the performance of each method as \( \log n \) increases.]
Comparison of hash table methods – Delete

Hash tables
Suppose we want to implement blocking of malicious web pages in a web browser.

Assume we have a list of 10,000,000 malicious pages.

We can store all pages in a hash table, but this requires a large amount of memory.

Bloom filter is an implementation of static dictionary that uses small amount of memory.

For a query key that is in the set, the Bloom filter always returns a positive answer.

For a query key that is not in the set, the Bloom filter can return either a negative answer, or a positive answer (false positive).
Blocking malicious web pages

- Suppose that the list of malicious pages is stored in a Bloom filter.
- For a query URL, if the answer is negative, we know that the URL is not a malicious page.
- If the answer is positive, we do not know the correct answer. In this case, we get the answer from a server.
- We want a small probability for a false positive.
• Suppose we want to store a set $S$.
• We use an array $A$ of size $m$ initialized to 0.
• We then choose a hash function $h$, and set $A[h(x)] \leftarrow 1$ for every $x \in S$.
• For a query $y$, if $A[h(y)] = 0$, we know that $y \not\in S$.
• If $A[h(y)] = 1$, either $y$ is in $S$ or not.

$S = \{15, 32\}$
$h(y) = y \mod 10$
Suppose we want to store a set $S$.

We use an array $A$ of size $m$ initialized to 0.

We then choose a hash function $h$, and set $A[h(x)] ← 1$ for every $x ∈ S$.

For a query $y$, if $A[h(y)] = 0$, we know that $y ∉ S$.

If $A[h(y)] = 1$, either $y$ is in $S$ or not.

$$x_1 = 15$$

$S = \{15, 32\}$

$$h(y) = y \mod 10$$
Bloom filter — simple version

- Suppose we want to store a set $S$.
- We use an array $A$ of size $m$ initialized to 0.
- We then choose a hash function $h$, and set $A[h(x)] ← 1$ for every $x ∈ S$.
- For a query $y$, if $A[h(y)] = 0$, we know that $y ∉ S$.
- If $A[h(y)] = 1$, either $y$ is in $S$ or not.

$$x_2 = 32 \quad x_1 = 15$$

$S = \{15, 32\}$

$$h(y) = y \mod 10$$
Suppose we want to store a set $S$.

We use an array $A$ of size $m$ initialized to 0.

We then choose a hash function $h$, and set $A[h(x)] \leftarrow 1$ for every $x \in S$.

For a query $y$, if $A[h(y)] = 0$, we know that $y \notin S$.

If $A[h(y)] = 1$, either $y$ is in $S$ or not.

$x_2 = 32 \quad y = 27 \quad x_1 = 15$

$S = \{15, 32\}$

$h(y) = y \mod 10$
Bloom filter — simple version

- Suppose we want to store a set $S$.
- We use an array $A$ of size $m$ initialized to 0.
- We then choose a hash function $h$, and set $A[h(x)] \leftarrow 1$ for every $x \in S$.
- For a query $y$, if $A[h(y)] = 0$, we know that $y \notin S$.
- If $A[h(y)] = 1$, either $y$ is in $S$ or not.

\[ x_2 = 32 \quad y = 35 \quad x_1 = 15 \]

\[
\begin{array}{cccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

$S = \{15, 32\}$

$h(y) = y \mod 10$
Probability of false positive

- Assume simple uniform hashing: any element is equally likely to hash into any of the $m$ slots, independently of where other elements have hashed into.
- For fixed $i$, the probability that $A[i] = 0$ is $p = (1 - 1/m)^n$.
- For $y \notin S$, the probability that $A[h(y)] = 1$ is $1 - p$.

$x_2 = 32 \quad y = 35 \quad x_1 = 15$

```
0 0 1 0 0 1 0 0 0 0
```

$p = (1 - \frac{1}{10})^2 = 0.81$

$1 - p = 0.19$
Reducing the false positive probability

- We choose $k$ hash functions $h_1, h_2, \ldots, h_k$, and set $A[h_j(x)] \leftarrow 1$ for $j = 1, 2, \ldots, k$, for every $x \in S$.
- For a query $y$, if $A[h_j(y)] = 0$ for some $j$, we know that $y \not\in S$.
- If $A[h_j(y)] = 1$ for all $j$, either $y$ is in $S$ or not.

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

$S = \{15, 32\}$

$k = 3$
Reducing the false positive probability

- We choose $k$ hash functions $h_1, h_2, \ldots, h_k$, and set $A[h_j(x)] \leftarrow 1$ for $j = 1, 2, \ldots, k$, for every $x \in S$.
- For a query $y$, if $A[h_j(y)] = 0$ for some $j$, we know that $y \not\in S$.
- If $A[h_j(y)] = 1$ for all $j$, either $y$ is in $S$ or not.

$S = \{15, 32\}$

$k = 3$
Reducing the false positive probability

- We choose $k$ hash functions $h_1, h_2, \ldots, h_k$, and set $A[h_j(x)] \leftarrow 1$ for $j = 1, 2, \ldots, k$, for every $x \in S$.
- For a query $y$, if $A[h_j(y)] = 0$ for some $j$, we know that $y \notin S$.
- If $A[h_j(y)] = 1$ for all $j$, either $y$ is in $S$ or not.

$x_2 = 32$

$h_3$

$h_1$ $h_2$

$h_2$ $h_1$ $h_3$

$x_1 = 15$

$S = \{15, 32\}$

$k = 3$
Reducing the false positive probability

- We choose $k$ hash functions $h_1, h_2, \ldots, h_k$, and set $A[h_j(x)] \leftarrow 1$ for $j = 1, 2, \ldots, k$, for every $x \in S$.
- For a query $y$, if $A[h_j(y)] = 0$ for some $j$, we know that $y \notin S$.
- If $A[h_j(y)] = 1$ for all $j$, either $y$ is in $S$ or not.

$$x_2 = 32 \quad y = 27 \quad x_1 = 15$$

$S = \{15, 32\}$

$k = 3$
Reducing the false positive probability

- We choose $k$ hash functions $h_1, h_2, \ldots, h_k$, and set $A[h_j(x)] \leftarrow 1$ for $j = 1, 2, \ldots, k$, for every $x \in S$.
- For a query $y$, if $A[h_j(y)] = 0$ for some $j$, we know that $y \notin S$.
- If $A[h_j(y)] = 1$ for all $j$, either $y$ is in $S$ or not.

$x_2 = 32 \quad y = 35 \quad x_1 = 15$

$S = \{15, 32\}$
$k = 3$
For fixed $i$, the probability that $A[i] = 0$ is 
$$p = \left(1 - \frac{1}{m}\right)^{kn}.$$ 
For $y \notin S$, the probability that $A[h(y)] = 1$ is $(1 - p)^k$.

$x_2 = 32$  $y = 35$  $x_1 = 15$

$$p = \left(1 - \frac{1}{10}\right)^6 \approx 0.531$$  
$$(1 - p)^3 \approx 0.103$$
For the previous example \((n = 2, m = 10)\):

- For \(k = 1\), \(p = 0.81\), \((1 - p)^1 \approx 0.19\).
- For \(k = 2\), \(p \approx 0.656\), \((1 - p)^2 \approx 0.118\).
- For \(k = 3\), \(p \approx 0.531\), \((1 - p)^3 \approx 0.103\).
- For \(k = 4\), \(p \approx 0.430\), \((1 - p)^4 \approx 0.105\).
- For \(k = 5\), \(p \approx 0.349\), \((1 - p)^5 \approx 0.117\).
Optimizing the probability of false positive

Let $f(k)$ be the probability of false positive.

$$p = (1 - 1/m)^{kn} \approx e^{-kn/m}$$

$$f(k) = (1 - p)^k \approx (1 - e^{-kn/m})^k = e^{k \cdot \ln(1 - e^{-kn/m})}$$

Minimizing $f(k)$ is equivalent to minimizing $g(k) = k \cdot \ln(1 - e^{-kn/m})$.

$$\frac{dg}{dk} = \ln(1 - e^{-kn/m}) + \frac{kn}{m} \frac{e^{-kn/m}}{1 - e^{-kn/m}}.$$ 

The minimum of $g(k)$ is when $dg/dk = 0 \implies k = \frac{m}{n} \ln 2$.

The minimum of $f(k)$ is $\approx 0.6185^{m/n}$.

Example: For $m = 8n$, $\frac{m}{n} \ln 2 \approx 5.55$. $f(5) \approx 0.0217$, $f(6) \approx 0.0216$. 

Bloom Filter