Hash tables
Dictionary

Definition

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 $U = \{0, \ldots, u - 1\}$. 

Hash tables
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($T, x$):** $T[x.key] \leftarrow$ NULL.

What is the problem with this structure?

![Hash tables diagram]
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
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 $T[h(x.key)]$.

**Search**($T, k$): Search the list $T[h(k)]$.

**Insert**($T, x$): Insert $x$ at the head of $T[h(x.key)]$.

**Delete**($T, x$): Delete $x$ from the list containing $x$.

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

$m = 5$, $h(x) = x \mod 5$
The worst case running times of the operations are:

- **Search**: $\Theta(n)$.
- **Insert**: $\Theta(1)$.
- **Delete**: $\Theta(n)$ (can be reduced to $\Theta(1)$ using doubly linked list).
Analysis

- **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 $|\{k \in U : h(k) = i\}| = u/m$ 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 $\Theta(1 + X)$. 
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 $\alpha = 1$,

\begin{align*}
\Pr[X = 0] &\approx 0.368 & \Pr[X = 4] &\approx 0.015 \\
\Pr[X = 1] &\approx 0.368 & \Pr[X = 5] &\approx 0.003 \\
\Pr[X = 2] &\approx 0.184 & \Pr[X = 6] &\approx 0.0005 \\
\Pr[X = 3] &\approx 0.061 & \Pr[X = 7] &\approx 0.00007
\end{align*}
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.

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

\[
h'(x) = x \mod 10
\]
Rehashing

- 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 \( \Theta(n) \).
In order to keep the space usage $\Theta(n)$, we need $n = \Omega(m)$.

We can keep $\alpha \in \left[\frac{1}{4}, 1\right]$. If an insert would make $\alpha > 1$, create a new table of size $2m$. If a delete would make $\alpha < \frac{1}{4}$, create a new table of size $m/2$. 
Universal hash functions

- **Random** keys + fixed hash function (e.g. mod) ⇒ The hashed keys are random numbers.
- A fixed set of keys + **random** hash function (selected from a universal set) ⇒ The hashed keys are semi-random numbers.
Universal hash functions

Definition

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

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

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

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

If \( k \notin S \), \( E[X] \leq n \cdot \frac{1}{m} \). Otherwise, \( E[X] \leq 1 + (n - 1) \frac{1}{m} \).
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.
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:

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

- 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. for \( i = d - 1 \) 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 $T = \text{BGUABC}$, $P = \text{GUAB}$, $T_1 = \text{BGUA}$, $T_2 = \text{GUAB}$.

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

$$h(T_2) = (71 \cdot 256^3 + 85 \cdot 256^2 + 65 \cdot 256 + 66) \mod p$$

$$= (h(T_1) - 66 \cdot 256^3) \cdot 256 + 66 \mod p$$
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 $h : U \mapsto \{1, \ldots, m\}$, 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 $h(f)$. 

![Diagram showing consistent hashing](image)
Distributed storage: Consistent hashing

- 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 $n/(m + 1)$.
Linear probing

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

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

```plaintext
evaluate insert(T, 20)
```

0 20
1
2
3
4 14
5
6
7
8
9
insert(T, 20)

Hash tables
Linear probing

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

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

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

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

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

---

**search**(T, 55)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>19</td>
</tr>
</tbody>
</table>
SEARCH($T$, $k$)

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

search($T, 25$)
Delete method 1: To delete element $k$, store in $T[h(k)]$ a special value DELETED.

Change line (3) in Insert to:

```java
if $T[j] = \text{NULL OR } T[j] = \text{DELETED}$
```

```
delete 34
```
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.

![Table with deletion highlighted]
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:
  \[
  \text{INSERT}(T, k) \\
  \begin{align*}
  (1) & \quad \text{for } i = 0 \text{ to } m - 1 \\
  (2) & \quad j \leftarrow h(k, i) \\
  (3) & \quad \text{if } T[j] = \text{NULL} \text{ OR } T[j] = \text{DELETED} \\
  (4) & \quad T[j] \leftarrow k \\
  (5) & \quad \text{return} \\
  (6) & \quad \text{error} \text{ “hash table overflow”}
  \end{align*}
  \]

- Deletion is done using delete method 1 defined above (using special value DELETED).
Double hashing

- 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,

\[
\begin{align*}
  h_1(k) &= k \mod m \\
  h_2(k) &= 1 + (k \mod m')
\end{align*}
\]

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 – Insert

Insert

- Chained Hashing
- Linear Probing

Clock Cycles

log n

Hash tables
Comparison of hash table methods – Delete

Hash tables
Bloom Filter
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).
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 $A[h(x)] \leftarrow 1$ 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.

$x_1 = 15$

$S = \{15, 32\}$
$h(y) = y \mod 10$
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.

$x_2 = 32$

$x_1 = 15$

$S = \{15, 32\}$

$h(y) = y \mod 10$
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.

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

$S = \{15, 32\} \quad h(y) = y \mod 10$
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 $y \not\in S$.
- If $A[h(y)] = 1$, either $y$ is in $S$ or not.

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

$S = \{15, 32\}$

$h(y) = y \mod 10$
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 $1 - p$.

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

\[ p = (1 - \frac{1}{10})^2 = 0.81 \quad 1 - p = 0.19 \]
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.

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

\( S = \{15, 32\} \)

\( k = 3 \)
Reducing the false positive probability

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

**Bloom Filter**

$x_1 = 15$

$S = \{15, 32\}$

$k = 3$
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.

\[ x_2 = 32 \]
\[ x_1 = 15 \]
\[ S = \{15, 32\} \]
\[ k = 3 \]
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 $y \not\in S$.
- 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 
$$p = (1 - 1/m)^{kn}.$$ 
For $y \notin S$, the probability that $A[h(y)] = 1$ is $(1 - p)^k$.

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

$$p = (1 - \frac{1}{10})^6 \approx 0.531$$

$$(1 - p)^3 \approx 0.103$$
Optimizing the probability of false positive

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 $\approx 0.6185m/n$.

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

Bloom Filter