Hashing

Lecture 16

March 17, 2015

Part I

Hash Tables

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
    1. **Search**/**lookup**: given $x \in \mathcal{U}$ is $x \in S$?
    2. **Insert**: given $x \notin S$ add x to S.
    3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
   1. **Search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
   2. **Insert**: given $x \notin S$ add x to S.
   3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
    1. **Search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
    2. **Insert**: given $x \notin S$ add x to S.
    3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
   1. **Search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
   2. **Insert**: given $x \notin S$ add x to S.
   3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
   1. **Search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
   2. **Insert**: given $x \notin S$ add x to S.
   3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
   1. **Search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
   2. **Insert**: given $x \notin S$ add x to S.
   3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
   1. **Search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
   2. **Insert**: given $x \notin S$ add x to S.
   3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $S \subseteq \mathcal{U}$
3. **Operations:**
   1. **Search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
   2. **Insert**: given $x \notin S$ add x to S.
   3. **Delete**: given $x \in S$ delete x from S
4. **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store S as a *sorted* array
   2. **Lookup**: Binary search in $O(\log|S|)$ time (comparisons)
2. Dynamic:
   1. Store S in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $O(\log|S|)$ time (comparisons)

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store S as a *sorted* array
   2. **Lookup**: Binary search in $O(\log |S|)$ time (comparisons)
2. Dynamic:
   1. Store S in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $O(\log |S|)$ time (comparisons)

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store S as a *sorted* array
   2. **Lookup**: Binary search in $O(\log |S|)$ time (comparisons)
2. Dynamic:
   1. Store S in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $O(\log |S|)$ time (comparisons)

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store S as a *sorted* array
   2. **Lookup**: Binary search in $O(\log |S|)$ time (comparisons)
2. Dynamic:
   1. Store S in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $O(\log |S|)$ time (comparisons)

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store S as a *sorted* array
   2. **Lookup**: Binary search in $O(\log |S|)$ time (comparisons)
2. Dynamic:
   1. Store S in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $O(\log |S|)$ time (comparisons)

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store S as a *sorted* array
   2. **Lookup**: Binary search in $O(\log |S|)$ time (comparisons)
2. Dynamic:
   1. Store S in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $O(\log |S|)$ time (comparisons)

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store S as a *sorted* array
   2. **Lookup**: Binary search in $O(\log |S|)$ time (comparisons)
2. Dynamic:
   1. Store S in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $O(\log |S|)$ time (comparisons)

Dictionary Data Structures II

1. **Question:** "Should Tables be Sorted?"
(also title of famous paper by Turing award winner Andy Yao)
2. Hashing is a widely used & powerful technique for dictionaries.
3. **Motivation:**
    1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
    2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
    3. Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Dictionary Data Structures II

1. **Question:** "Should Tables be Sorted?"
   (also title of famous paper by Turing award winner Andy Yao)
2. Hashing is a widely used & powerful technique for dictionaries.
3. **Motivation:**
   1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
   2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
   3. Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Dictionary Data Structures II

1. **Question:** "Should Tables be Sorted?"
   (also title of famous paper by Turing award winner Andy Yao)
2. Hashing is a widely used & powerful technique for dictionaries.
3. **Motivation:**
   1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
   2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
   3. Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Dictionary Data Structures II

1. **Question:** "Should Tables be Sorted?"
   (also title of famous paper by Turing award winner Andy Yao)
2. Hashing is a widely used & powerful technique for dictionaries.
3. **Motivation:**
   1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
   2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
   3. Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Dictionary Data Structures II

1. **Question:** "Should Tables be Sorted?"
   (also title of famous paper by Turing award winner Andy Yao)
2. Hashing is a widely used & powerful technique for dictionaries.
3. **Motivation:**
   1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
   2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
   3. Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Dictionary Data Structures II

1. **Question:** "Should Tables be Sorted?"
   (also title of famous paper by Turing award winner Andy Yao)
2. Hashing is a widely used & powerful technique for dictionaries.
3. **Motivation:**
   1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
   2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
   3. Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Hashing and Hash Tables

1. Hash Table data structure:
   1. A (hash) table/array T of size m (the table **size**).
   2. A hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
   3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

1. Hash Table data structure:
   1. A (hash) table/array T of size m (the table **size**).
   2. A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
   3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

1. Hash Table data structure:
   1. A (hash) table/array T of size m (the table **size**).
   2. A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
   3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

1. Hash Table data structure:
   1. A (hash) table/array T of size m (the table **size**).
   2. A hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
   3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

1. Hash Table data structure:
   1. A (hash) table/array T of size m (the table **size**).
   2. A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
   3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

1. Hash Table data structure:
   1. A (hash) table/array T of size m (the table **size**).
   2. A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
   3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

1. Hash Table data structure:
    1. A (hash) table/array T of size m (the table **size**).
    2. A hash function $h : \mathcal{U} \to \{0, \ldots, m - 1\}$.
    3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

1. Hash Table data structure:
   1. A (hash) table/array T of size m (the table **size**).
   2. A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
   3. Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
2. Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
3. ...

Ideal situation:

1. Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
2. **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

4. Collisions unavoidable. Several different techniques to handle them.

Handling Collisions: Chaining

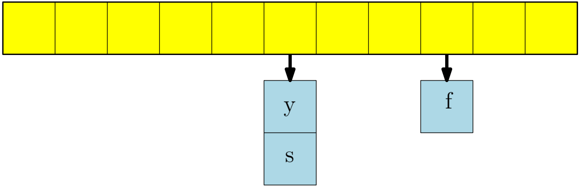
1. **Collision:** $h(x) = h(y)$ for some $x \neq y$.
2. **Chaining** to handle collisions:
   1. For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list
   2. **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.
3. This is also known as **Open hashing**.

Handling Collisions: Chaining

1. **Collision:** $h(x) = h(y)$ for some $x \neq y$.
2. **Chaining** to handle collisions:
   1. For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list
   2. **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.
3. This is also known as **Open hashing**.

Handling Collisions: Chaining

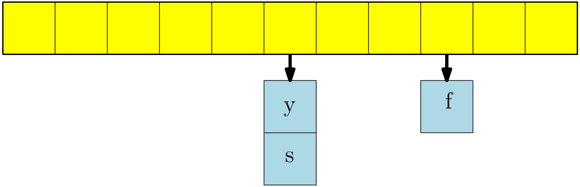
1. **Collision:** $h(x) = h(y)$ for some $x \neq y$.
2. **Chaining** to handle collisions:
   1. For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list
   2. **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.



This is also known as **Open hashing**.

Handling Collisions: Chaining

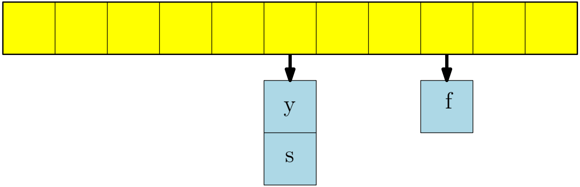
1. **Collision:** $h(x) = h(y)$ for some $x \neq y$.
2. **Chaining** to handle collisions:
   1. For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list
   2. **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.



This is also known as **Open hashing**.

Handling Collisions: Chaining

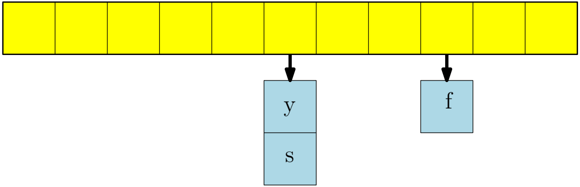
1. **Collision:** $h(x) = h(y)$ for some $x \neq y$.
2. **Chaining** to handle collisions:
   1. For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list
   2. **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.



This is also known as **Open hashing**.

Handling Collisions: Chaining

1. **Collision:** $h(x) = h(y)$ for some $x \neq y$.
2. **Chaining** to handle collisions:
   1. For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list
   2. **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.



This is also known as **Open hashing**.

Handling Collisions

Several other techniques:

1. Open addressing.
   Every element has a list of places it can be (in certain order). Check in this order.
2. ...
3. Cuckoo hashing.
   Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick stored value to its other location. Repeat till stable. if no stability then rebuild table.
4. Others.

Handling Collisions

Several other techniques:

1. Open addressing.
   Every element has a list of places it can be (in certain order). Check in this order.
2. . . .
3. Cuckoo hashing.
   Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick stored value to its other location. Repeat till stable. if no stability then rebuild table.
4. Others.

Handling Collisions

Several other techniques:

1. Open addressing.
   Every element has a list of places it can be (in certain order). Check in this order.
2. ...
3. Cuckoo hashing.
   Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick stored value to its other location. Repeat till stable. if no stability then rebuild table.
4. Others.

Handling Collisions

Several other techniques:

1. Open addressing.
   Every element has a list of places it can be (in certain order). Check in this order.
2. ...
3. Cuckoo hashing.
   Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick stored value to its other location. Repeat till stable. if no stability then rebuild table.
4. Others.

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
   1. Complexity of evaluating h on a given element?
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
   3. Size of table relative to size of S.
   4. Worst-case vs average-case vs randomized (expected) time?
   5. How do we choose h?

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
   1. Complexity of evaluating h on a given element?
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
   3. Size of table relative to size of S.
   4. Worst-case vs average-case vs randomized (expected) time?
   5. How do we choose h?

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
   1. Complexity of evaluating h on a given element?
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
   3. Size of table relative to size of S.
   4. Worst-case vs average-case vs randomized (expected) time?
   5. How do we choose h?

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
   1. Complexity of evaluating h on a given element?
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
   3. Size of table relative to size of S.
   4. Worst-case vs average-case vs randomized (expected) time?
   5. How do we choose h?

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
    1. Complexity of evaluating h on a given element?
    2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
    3. Size of table relative to size of S.
    4. Worst-case vs average-case vs randomized (expected) time?
    5. How do we choose h?

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
   1. Complexity of evaluating h on a given element?
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
   3. Size of table relative to size of S.
   4. Worst-case vs average-case vs randomized (expected) time?
   5. How do we choose h?

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
    1. Complexity of evaluating h on a given element?
    2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
    3. Size of table relative to size of S.
    4. Worst-case vs average-case vs randomized (expected) time?
    5. How do we choose h?

Understanding Hashing

1. Does hashing give $O(1)$ time per operation for dictionaries?
2. **Questions:**
    1. Complexity of evaluating h on a given element?
    2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
    3. Size of table relative to size of S.
    4. Worst-case vs average-case vs randomized (expected) time?
    5. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Understanding Hashing

1. Considerations:
   1. Complexity of evaluating h on a given element? Should be small.
   2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   3. Size of table relative to size of S. The **load factor** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
      Also known as the **fill factor**.
2. Main and interrelated questions:
   1. Worst-case vs average-case vs randomized (expected) time?
   2. How do we choose h?

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
3. Fix hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
4. N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
5. Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
3. Fix hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
4. N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
5. Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
3. Fix hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
4. N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
5. Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
3. Fix hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
4. N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
5. Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
3. Fix hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
4. N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
5. Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
3. Fix hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
4. N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
5. Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
3. Fix hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
4. N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
5. Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Picking a hash function

1. How to pick functions?
    1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
    2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
    1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
    2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
    3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function

1. How to pick functions?
   1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
   2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
   1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
   2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
   3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function

1. How to pick functions?
   1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
   2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
   1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
   2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
   3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function

1. How to pick functions?
   1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
   2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
   1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
   2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
   3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function

1. How to pick functions?
   1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
   2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
   1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
   2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
   3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function

1. How to pick functions?
   1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
   2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
   1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
   2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
   3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function

1. How to pick functions?
   1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
   2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
   1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
   2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
   3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function

1. How to pick functions?
    1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
    2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
2. Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
    1. $\mathcal{H}$ is a **family** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
    2. h is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
    3. Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

Picking a hash function II

1. **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
2. 1. Too many functions! A random function has high complexity!
   \# of functions: $M = m^{|\mathcal{U}|}$.
   Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
3. **Question:** Are there good and compact families $\mathcal{H}$?
   1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function II

1. **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
2.
   1. Too many functions! A random function has high complexity!
      \# of functions: $M = m^{|\mathcal{U}|}$.
      Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
3. **Question:** Are there good and compact families $\mathcal{H}$?
   1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function II

1. **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
2. 1. Too many functions! A random function has high complexity!
      \# of functions: $M = m^{|\mathcal{U}|}$.
      Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
3. **Question:** Are there good and compact families $\mathcal{H}$?
   1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function II

1. **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
2. 1. Too many functions! A random function has high complexity!
      \# of functions: $M = m^{|\mathcal{U}|}$.
      Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
3. **Question:** Are there good and compact families $\mathcal{H}$?
   1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function II

1. **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
2.
   1. Too many functions! A random function has high complexity!
      \# of functions: $M = m^{|\mathcal{U}|}$.
      Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
3. **Question:** Are there good and compact families $\mathcal{H}$?
   1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function II

1. **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
2.
    1. Too many functions! A random function has high complexity!
    # of functions: $M = m^{|\mathcal{U}|}$.
    Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
3. **Question:** Are there good and compact families $\mathcal{H}$?
    1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function II

1. **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
2. 1. Too many functions! A random function has high complexity!
      \# of functions: $M = m^{|\mathcal{U}|}$.
      Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
3. **Question:** Are there good and compact families $\mathcal{H}$?
    1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \leq i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \leq i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \leq i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \leq i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \leq i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \le i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \le i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \le i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Analyzing Uniform Hashing

1. T is hash table of size m.
2. $S \subseteq \mathcal{U}$ is a **fixed** set of size $\leq m$.
3. h is chosen randomly from a uniform hash family $\mathcal{H}$.
4. x is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $x \notin S$.

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

Analyzing Uniform Hashing

1. T is hash table of size m.
2. $S \subseteq \mathcal{U}$ is a **fixed** set of size $\leq m$.
3. h is chosen randomly from a uniform hash family $\mathcal{H}$.
4. x is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $x \notin S$.

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

Analyzing Uniform Hashing

1. T is hash table of size m.
2. $S \subseteq \mathcal{U}$ is a **fixed** set of size $\leq m$.
3. h is chosen randomly from a uniform hash family $\mathcal{H}$.
4. x is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $x \notin S$.

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

Analyzing Uniform Hashing

1. T is hash table of size m.
2. $S \subseteq \mathcal{U}$ is a **fixed** set of size $\leq m$.
3. h is chosen randomly from a uniform hash family $\mathcal{H}$.
4. x is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $x \notin S$.

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

Analyzing Uniform Hashing

1. T is hash table of size m.
2. $S \subseteq \mathcal{U}$ is a **fixed** set of size $\leq m$.
3. h is chosen randomly from a uniform hash family $\mathcal{H}$.
4. x is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $x \notin S$.

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

Analyzing Uniform Hashing

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

1. The time to look up x is the size of the list at $T[h(x)]$: same as the number of elements in S that collide with x under h.
2. Let $\ell(x)$ be this number. We want $\mathrm{E}[\ell(x)]$
3. For $y \in S$ let A_y be the even that x, y collide and D_y be the corresponding indicator variable.

Analyzing Uniform Hashing

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

1. The time to look up x is the size of the list at $T[h(x)]$: same as the number of elements in S that collide with x under h.
2. Let $\ell(x)$ be this number. We want $\mathrm{E}[\ell(x)]$
3. For $y \in S$ let A_y be the even that x, y collide and D_y be the corresponding indicator variable.

Analyzing Uniform Hashing

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

1. The time to look up x is the size of the list at $T[h(x)]$: same as the number of elements in S that collide with x under h.
2. Let $\ell(x)$ be this number. We want $\mathrm{E}[\ell(x)]$
3. For $y \in S$ let A_y be the even that x, y collide and D_y be the corresponding indicator variable.

Analyzing Uniform Hashing

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

1. The time to look up x is the size of the list at $T[h(x)]$: same as the number of elements in S that collide with x under h.
2. Let $\ell(x)$ be this number. We want $\mathrm{E}[\ell(x)]$
3. For $y \in S$ let A_y be the even that x, y collide and D_y be the corresponding indicator variable.

Analyzing Uniform Hashing

Continued...

Number of elements colliding with x: $\ell(x) = \sum_{y \in S} D_y$.

$$
\begin{aligned}
\Rightarrow \mathrm{E}[\ell(x)] &= \sum_{y \in S} \mathrm{E}[D_y] \qquad \text{linearity of expectation} \\
&= \sum_{y \in S} Pr[h(x) = h(y)] \\
&= \sum_{y \in S} \frac{1}{m} \qquad \text{since } \mathcal{H} \text{ is a uniform hash family} \\
&= |S|/m \\
&\leq 1 \quad \text{if } |S| \leq m
\end{aligned}
$$

Analyzing Uniform Hashing

1. **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
2. **Answer:** $O(n/m)$.
3. Comments:
   1. $O(1)$ expected time also holds for insertion.
   2. Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
   3. **Worst-case**: look up time can be large! How large? $\Omega(\log n / \log\log n)$
      [Lower bound holds even under stronger assumptions.]

Analyzing Uniform Hashing

1. **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
2. **Answer:** $O(n/m)$.
3. Comments:
    1. $O(1)$ expected time also holds for insertion.
    2. Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
    3. **Worst-case**: look up time can be large! How large? $\Omega(\log n / \log\log n)$
       [Lower bound holds even under stronger assumptions.]

Analyzing Uniform Hashing

1. **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
2. **Answer:** $O(n/m)$.
3. Comments:
   1. $O(1)$ expected time also holds for insertion.
   2. Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
   3. **Worst-case**: look up time can be large! How large? $\Omega(\log n / \log\log n)$
      [Lower bound holds even under stronger assumptions.]

Analyzing Uniform Hashing

1. **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
2. **Answer:** $O(n/m)$.
3. Comments:
    1. $O(1)$ expected time also holds for insertion.
    2. Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
    3. **Worst-case**: look up time can be large! How large? $\Omega(\log n / \log\log n)$
       [Lower bound holds even under stronger assumptions.]

Analyzing Uniform Hashing

1. **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
2. **Answer:** $O(n/m)$.
3. Comments:
   1. $O(1)$ expected time also holds for insertion.
   2. Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
   3. **Worst-case**: look up time can be large! How large? $\Omega(\log n / \log\log n)$
      [Lower bound holds even under stronger assumptions.]

Analyzing Uniform Hashing

1. **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
2. **Answer:** $O(n/m)$.
3. Comments:
   1. $O(1)$ expected time also holds for insertion.
   2. Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
   3. Worst-case: look up time can be large! How large? $\Omega(\log n / \log\log n)$ [Lower bound holds even under stronger assumptions.]

Analyzing Uniform Hashing

1. **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
2. **Answer:** $O(n/m)$.
3. Comments:
   1. $O(1)$ expected time also holds for insertion.
   2. Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
   3. **Worst-case**: look up time can be large! How large?
      $\Omega(\log n / \log \log n)$
      [Lower bound holds even under stronger assumptions.]

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

1. If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.
2. If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

1. If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.
2. If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

1. If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.
2. If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

1. If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.
2. If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

1. If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.
2. If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

1. If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.
2. If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

1. If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.
2. If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rebuilding the hash table

1. Start with table size m where m is some estimate of $|S|$ (can be some large constant).
2. If $|S|$ grows to more than twice current table size, build new hash table (choose a new random hash function) with double the current number of elements. Can also use similar trick if table size falls below quarter the size.
3. If $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant c (say $\mathbf{10}$), rebuild.

The amortize cost of rebuilding to previously performed operations. Rebuilding ensures $O(1)$ expected analysis holds even when S changes. Hence $O(1)$ expected look up/insert/delete time *dynamic* data dictionary data structure!

Rebuilding the hash table

1. Start with table size m where m is some estimate of $|S|$ (can be some large constant).
2. If $|S|$ grows to more than twice current table size, build new hash table (choose a new random hash function) with double the current number of elements. Can also use similar trick if table size falls below quarter the size.
3. If $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant c (say 10), rebuild.

The amortize cost of rebuilding to previously performed operations. Rebuilding ensures $O(1)$ expected analysis holds even when S changes. Hence $O(1)$ expected look up/insert/delete time *dynamic* data dictionary data structure!

Rebuilding the hash table

1. Start with table size m where m is some estimate of $|S|$ (can be some large constant).
2. If $|S|$ grows to more than twice current table size, build new hash table (choose a new random hash function) with double the current number of elements. Can also use similar trick if table size falls below quarter the size.
3. If $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant c (say 10), rebuild.

The amortize cost of rebuilding to previously performed operations. Rebuilding ensures $O(1)$ expected analysis holds even when S changes. Hence $O(1)$ expected look up/insert/delete time *dynamic* data dictionary data structure!

Rebuilding the hash table

1. Start with table size m where m is some estimate of $|S|$ (can be some large constant).
2. If $|S|$ grows to more than twice current table size, build new hash table (choose a new random hash function) with double the current number of elements. Can also use similar trick if table size falls below quarter the size.
3. If $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant c (say $\mathbf{10}$), rebuild.

The **amortize** cost of rebuilding to previously performed operations. Rebuilding ensures $O(\mathbf{1})$ expected analysis holds even when S changes. Hence $O(\mathbf{1})$ expected look up/insert/delete time *dynamic* data dictionary data structure!

Some math required...

Lemma

Let p be a prime number,
x: an integer number in $\{1, \ldots, p-1\}$.
$\implies$ *There exists a unique y s.t. $xy = 1 \mod p$.*

In other words: For every element there is a unique inverse.
$\implies$ $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ when working module p is a field.

Proof of lemma

Claim

Let p be a prime number. For any $\alpha, \beta, i \in \{1, \ldots, p-1\}$ s.t. $\alpha \neq \beta$, we have that $\alpha i \neq \beta i \mod p$.

Proof.

Assume for the sake of contradiction $\alpha i = \beta i \mod p$. Then

$$
\begin{aligned}
& i(\alpha - \beta) = 0 \mod p \\
\implies & \quad p \text{ divides } i(\alpha - \beta) \\
\implies & \quad p \text{ divides } \alpha - \beta \\
\implies & \quad \alpha - \beta = 0 \\
\implies & \quad \alpha = \beta.
\end{aligned}
$$

And that is a contradiction. □

Proof of lemma...

Uniqueness.

Lemma

Let p be a prime number,
x: an integer number in $\{1, \ldots, p-1\}$.
$\implies$ *There exists a unique y s.t. $xy = 1 \mod p$.*

Proof.

Assume the lemma is false and there are two distinct numbers $y, z \in \{1, \ldots, p-1\}$ such that

$$xy = 1 \mod p \quad \text{and} \quad xz = 1 \mod p.$$

But this contradicts the above claim (set $i = x$, $\alpha = y$ and $\beta = z$). □

Proof of lemma...

Existence

Proof.

By claim, for any $\alpha \in \{1, \dots, p-1\}$ we have that $\{\alpha * 1 \mod p, \alpha * 2 \mod p, \dots, \alpha * (p-1) \mod p\} = \{1, 2, \dots, p-1\}$.

$\implies$ There exists a number $y \in \{1, \dots, p-1\}$ such that $\alpha y = 1 \mod p$. $\square$

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

1. Choose a **prime** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
2. For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.
3. Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

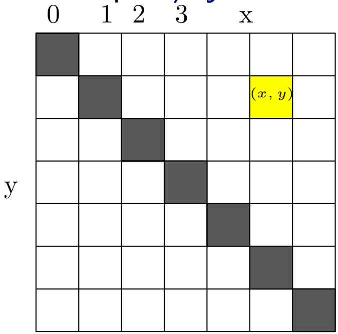
$\mathcal{H}$ is a 2-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function (a, b have to be stored) and evaluate it.

What the is going on?

$h_{a,b}(x) = ((ax + b) \mod p) \mod m$
First map $x \neq y$ to $r = h(x)$ and $s = h(y)$.

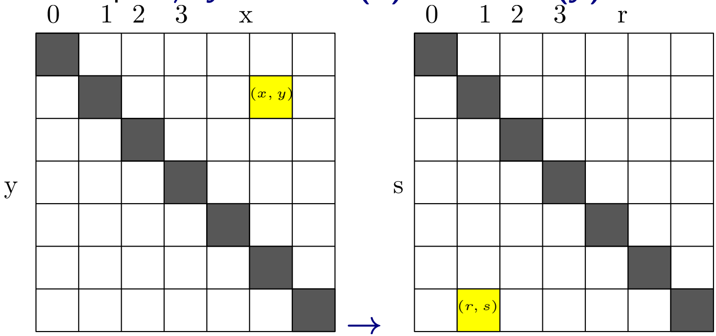


This is a random uniform mapping (choosing a and b) – every cell has the same probability to be the target, for fixed x and y.

What the is going on?

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

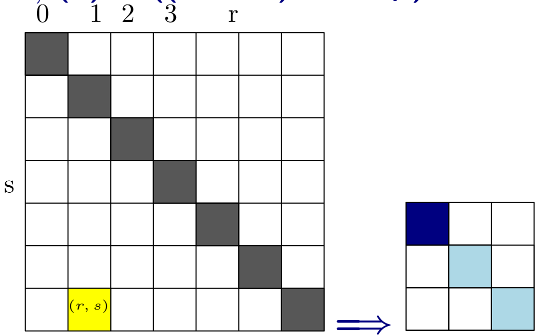
First map $x \neq y$ to $r = h(x)$ and $s = h(y)$.



This is a random uniform mapping (choosing a and b) – every cell has the same probability to be the target, for fixed x and y.

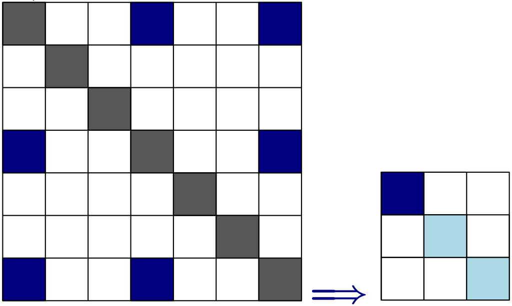
What the is going on?

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



What the is going on?

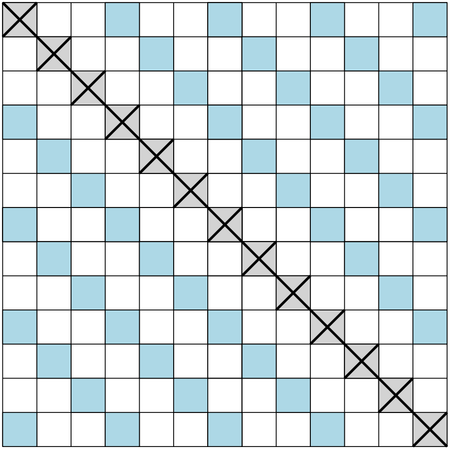
$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$



What the is going on?

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

1. First part of mapping maps (x, y) to a random location $(h_{a,b}(x), h_{a,b}(y))$ in the "matrix".
2. $(h_{a,b}(x), h_{a,b}(y))$ is not on main diagonal.
3. All blue locations are "bad" – map by $\mod m$ to a location of collusion.
4. But... at most $1/m$ fraction of allowable locations in the matrix are bad.



Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $x, y \in \mathcal{U}$. What is the probability they will collide if h is picked randomly from $\mathcal{H}$?

1. Let a, b be *bad* for x, y if $h_{a,b}(x) = h_{a,b}(y)$.
2. **Claim:** Number of bad pairs is at most $p(p-1)/m$.
3. Total number of hash functions is $p(p-1)$ and hence probability of a collision is $\leq 1/m$. □

Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $x, y \in \mathcal{U}$. What is the probability they will collide if h is picked randomly from $\mathcal{H}$?

1. Let a, b be *bad* for x, y if $h_{a,b}(x) = h_{a,b}(y)$.
2. **Claim:** Number of bad pairs is at most $p(p-1)/m$.
3. Total number of hash functions is $p(p-1)$ and hence probability of a collision is $\leq 1/m$. ∎

Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $x, y \in \mathcal{U}$. What is the probability they will collide if h is picked randomly from $\mathcal{H}$?

1. Let a, b be *bad* for x, y if $h_{a,b}(x) = h_{a,b}(y)$.
2. **Claim:** Number of bad pairs is at most $p(p-1)/m$.
3. Total number of hash functions is $p(p-1)$ and hence probability of a collision is $\leq 1/m$. □

Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $x, y \in \mathcal{U}$. What is the probability they will collide if h is picked randomly from $\mathcal{H}$?

1. Let a, b be *bad* for x, y if $h_{a,b}(x) = h_{a,b}(y)$.
2. **Claim:** Number of bad pairs is at most $p(p-1)/m$.
3. Total number of hash functions is $p(p-1)$ and hence probability of a collision is $\leq 1/m$. □

Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $x, y \in \mathcal{U}$. What is the probability they will collide if h is picked randomly from $\mathcal{H}$?

1. Let a, b be *bad* for x, y if $h_{a,b}(x) = h_{a,b}(y)$.
2. **Claim:** Number of bad pairs is at most $p(p-1)/m$.
3. Total number of hash functions is $p(p-1)$ and hence probability of a collision is $\leq 1/m$. $\square$

Some Lemmas

Lemma

If $x \neq y$ then for any $a, b \in \mathbb{Z}_p$ such that $a \neq 0$, we have

$$ax + b \mod p \neq ay + b \mod p.$$

Proof.

If $ax + b \mod p = ay + b \mod p$ then $a(x - y) \mod p = 0$ and $a \neq 0$ and $(x - y) \neq 0$. However, a and $(x - y)$ cannot divide p since p is prime and $a < p$ and $(x - y) < p$. $\square$

Some Lemmas

Lemma

If $x \neq y$ *then for each* (r, s) *such that* $r \neq s$ *and* $0 \leq r, s \leq p - 1$ *there is exactly* **one** a, b *such that*

$$ax + b \mod p = r \text{ and } ay + b \mod p = s.$$

Proof.

Solve the two equations:

$$ax + b = r \mod p \qquad \text{and} \qquad ay + b = s \mod p$$

We get $a = \frac{r-s}{x-y} \mod p$ and $b = r - ax \mod p$. □

Understanding the hashing

Once we fix a and b, and we are given a value x, we compute the hash value of x in two stages:

1. **Compute**: $r \leftarrow (ax + b) \mod p$.
2. **Fold**: $r' \leftarrow r \mod m$

Collision...

Given two values x and y they might collide because of either steps.

Lemma

$\#$ not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.

Understanding the hashing

Once we fix a and b, and we are given a value x, we compute the hash value of x in two stages:

1. **Compute**: $r \leftarrow (ax + b) \mod p$.
2. **Fold**: $r' \leftarrow r \mod m$

Collision...

Given two values x and y they might collide because of either steps.

Lemma

$\#$ not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.

Understanding the hashing

Once we fix a and b, and we are given a value x, we compute the hash value of x in two stages:

1. **Compute**: $r \leftarrow (ax + b) \mod p$.
2. **Fold**: $r' \leftarrow r \mod m$

Collision...

Given two values x and y they might collide because of either steps.

Lemma

 # not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.

Folding numbers

Lemma

 # not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.

Proof.

Consider a pair $(x, y) \in \{0, 1, \ldots, p-1\}^2$ s.t. $x \neq y$. Fix x:

1. There are $\lceil p/m \rceil$ values of y that fold into x. That is

$$x \mod m = y \mod m.$$

2. One of them is when $x = y$.
3. $\implies$ # of colliding pairs $(\lceil p/m \rceil - 1)p \leq (p-1)p/m$

Folding numbers

Lemma

 # not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.

Proof.

Consider a pair $(x, y) \in \{0, 1, \ldots, p-1\}^2$ s.t. $x \neq y$. Fix x:

1. There are $\lceil p/m \rceil$ values of y that fold into x. That is

$$x \mod m = y \mod m.$$

2. One of them is when $x = y$.
3. $\implies$ # of colliding pairs $(\lceil p/m \rceil - 1)p \leq (p-1)p/m$

Folding numbers

Lemma

 # not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ *that are folded to the same number is* $p(p-1)/m$.

Proof.

Consider a pair $(x, y) \in \{0, 1, \ldots, p-1\}^2$ s.t. $x \neq y$. Fix x:

1. There are $\lceil p/m \rceil$ values of y that fold into x. That is

$$x \mod m = y \mod m.$$

2. One of them is when $x = y$.
3. $\implies$ # of colliding pairs $(\lceil p/m \rceil - 1)p \leq (p-1)p/m$

Folding numbers

Lemma

$\#$ not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.

Proof.

Consider a pair $(x, y) \in \{0, 1, \ldots, p-1\}^2$ s.t. $x \neq y$. Fix x:

1. There are $\lceil p/m \rceil$ values of y that fold into x. That is

$$x \mod m = y \mod m.$$

2. One of them is when $x = y$.
3. $\implies \#$ of colliding pairs $(\lceil p/m \rceil - 1)p \leq (p-1)p/m$

Proof of Claim

of bad pairs is $\mathbf{p(p-1)/m}$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b \mod = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Proof of Claim

of bad pairs is $\mathbf{p(p-1)/m}$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b \mod = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Proof of Claim

\# of bad pairs is $\mathbf{p(p-1)/m}$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b \mod\ = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Proof of Claim

of bad pairs is $\mathbf{p(p-1)/m}$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b \mod = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Proof of Claim

of bad pairs is $\mathbf{p(p-1)/m}$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b \mod = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

□

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Proof of Claim

of bad pairs is $\mathbf{p(p-1)/m}$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b \mod = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

$\square$

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Perfect Hashing

1. **Question:** Can we make look up time $O(1)$ in worst case?
2. Yes, for static dictionaries but then space usage is $O(m)$ only in expectation.

Perfect Hashing

1. **Question:** Can we make look up time $O(1)$ in worst case?
2. Yes, for static dictionaries but then space usage is $O(m)$ only in expectation.

Perfect Hashing

1. **Question:** Can we make look up time $O(1)$ in worst case?
2. Yes, for static dictionaries but then space usage is $O(m)$ only in expectation.

Practical Issues

Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement for other objects need to map objects in some fashion to integers (via representation)
- Practical methods for various important cases such as vectors, strings are studied extensively. See http://en.wikipedia.org/wiki/Universal_hashing for some pointers.
- Recent important paper briding theory and practice of hashing. "The power of simple tabulation hashing" by Mikkel Thorup and Mihai Patrascu, 2011. See http://en.wikipedia.org/wiki/Tabulation_hashing

Practical Issues

Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement for other objects need to map objects in some fashion to integers (via representation)
- Practical methods for various important cases such as vectors, strings are studied extensively. See http://en.wikipedia.org/wiki/Universal_hashing for some pointers.
- Recent important paper briding theory and practice of hashing. "The power of simple tabulation hashing" by Mikkel Thorup and Mihai Patrascu, 2011. See http://en.wikipedia.org/wiki/Tabulation_hashing

Practical Issues

Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement for other objects need to map objects in some fashion to integers (via representation)
- Practical methods for various important cases such as vectors, strings are studied extensively. See http://en.wikipedia.org/wiki/Universal_hashing for some pointers.
- Recent important paper briding theory and practice of hashing. "The power of simple tabulation hashing" by Mikkel Thorup and Mihai Patrascu, 2011. See http://en.wikipedia.org/wiki/Tabulation_hashing

Practical Issues

Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement for other objects need to map objects in some fashion to integers (via representation)
- Practical methods for various important cases such as vectors, strings are studied extensively. See http://en.wikipedia.org/wiki/Universal_hashing for some pointers.
- Recent important paper briding theory and practice of hashing. "The power of simple tabulation hashing" by Mikkel Thorup and Mihai Patrascu, 2011. See http://en.wikipedia.org/wiki/Tabulation_hashing

Bloom Filters

1. **Hashing:**
    1. To insert x in dictionary store x in table in location $h(x)$
    2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
    1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
    2. To insert x in dictionary set *bit* to $\mathbf{1}$ in location $h(x)$ (initially all bits are set to $\mathbf{0}$)
    3. To lookup y if bit in location $h(y)$ is $\mathbf{1}$ say yes, else no.

Bloom Filters

1. **Hashing:**
   1. To insert x in dictionary store x in table in location $h(x)$
   2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
   1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
   2. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   3. To lookup y if bit in location $h(y)$ is 1 say yes, else no.

Bloom Filters

1. **Hashing:**
   1. To insert x in dictionary store x in table in location $h(x)$
   2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
   1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
   2. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   3. To lookup y if bit in location $h(y)$ is 1 say yes, else no.

Bloom Filters

1. **Hashing:**
   1. To insert x in dictionary store x in table in location $h(x)$
   2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
   1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
   2. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   3. To lookup y if bit in location $h(y)$ is 1 say yes, else no.

Bloom Filters

1. **Hashing:**
   1. To insert x in dictionary store x in table in location $h(x)$
   2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
   1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
   2. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   3. To lookup y if bit in location $h(y)$ is 1 say yes, else no.

Bloom Filters

1. **Hashing:**
   1. To insert x in dictionary store x in table in location $h(x)$
   2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
   1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
   2. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   3. To lookup y if bit in location $h(y)$ is 1 say yes, else no.

Bloom Filters

1. **Hashing:**
   1. To insert x in dictionary store x in table in location $h(x)$
   2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
   1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
   2. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   3. To lookup y if bit in location $h(y)$ is 1 say yes, else no.

Bloom Filters

1. **Hashing:**
   1. To insert x in dictionary store x in table in location $h(x)$
   2. To lookup y in dictionary check contents of location $h(y)$
2. **Bloom Filter:** tradeoff space for false positives
   1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
   2. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   3. To lookup y if bit in location $h(y)$ is 1 say yes, else no.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
    1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
    2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
    3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
    1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
    2. To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
    3. To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
    1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
    2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
    3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
    1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
    2. To insert x for $1 \le i \le k$ set bit in location $h_i(x)$ in table i to 1
    3. To lookup y compute $h_i(y)$ for $1 \le i \le k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
   1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
   3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
   1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
   2. To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
   3. To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
   1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
   3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
   1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
   2. To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
   3. To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
    1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
    2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
    3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
    1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
    2. To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
    3. To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
   1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
   3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
   1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
   2. To insert x for $1 \le i \le k$ set bit in location $h_i(x)$ in table i to 1
   3. To lookup y compute $h_i(y)$ for $1 \le i \le k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
   1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
   3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
   1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
   2. To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
   3. To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
   1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
   2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
   3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
   1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
   2. To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
   3. To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Bloom Filters

1. **Bloom Filter:** tradeoff space for false positives
    1. To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
    2. To lookup y if bit in location $h(y)$ is 1 say yes, else no
    3. No false negatives but false positives possible due to collisions
2. Reducing false positives:
    1. Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
    2. To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
    3. To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

Take away points

1. Hashing is a powerful and important technique for dictionaries. Many practical applications.
2. Randomization fundamental to understanding hashing.
3. Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
4. Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
5. Many applications in practice.

Take away points

1. Hashing is a powerful and important technique for dictionaries. Many practical applications.
2. Randomization fundamental to understanding hashing.
3. Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
4. Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
5. Many applications in practice.

Take away points

1. Hashing is a powerful and important technique for dictionaries. Many practical applications.
2. Randomization fundamental to understanding hashing.
3. Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
4. Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
5. Many applications in practice.

Take away points

1. Hashing is a powerful and important technique for dictionaries. Many practical applications.
2. Randomization fundamental to understanding hashing.
3. Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
4. Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
5. Many applications in practice.

Take away points

1. Hashing is a powerful and important technique for dictionaries. Many practical applications.
2. Randomization fundamental to understanding hashing.
3. Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
4. Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
5. Many applications in practice.

Take away points

1. Hashing is a powerful and important technique for dictionaries. Many practical applications.
2. Randomization fundamental to understanding hashing.
3. Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
4. Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
5. Many applications in practice.

Notes

Notes

Notes

Notes