Universal Hashing

COMPCSI 753: Algorithms for Massive Data

Ninh Pham

University of Auckland

Parts of this material are modifications of the lecture slides of Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/)
Designed for the textbook **Algorithm Design**
by Jon Kleinberg and Eva Tardos.

Auckland, Aug 3, 2020

Dictionary data type

- Dictionary problem:
  - Given a universe **U** of possible elements, maintain a subset $\mathbf{S} \subseteq \mathbf{U}$ supporting fast searching, inserting, deleting in **S** (in constant time).

- Main dictionary operations:
  - Insert(x): insert element $\mathbf{x} \in \mathbf{U}$ to **S**.
  - Delete(x): remove element **x** from **S** if $\mathbf{x} \in \mathbf{S}$.
  - Search(x): return true if $\mathbf{x} \in \mathbf{S}$ and false otherwise.

- Applications:
  - File systems, web caching, databases, Google, checksum P2P network,…

C++ unordered_set

cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::unordered_set

std::unordered_set

Defined in header <unordered_set>

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

(1) (since C++11)

```
namespace pmr {
    template <class Key,
              class Hash = std::hash<Key>,
              class Pred = std::equal_to<Key>>
    using unordered_set = std::unordered_set<Key, Hash, Pred,
                                    std::pmr::polymorphic_allocator<Key>>;
}
```

(2) (since C++17)

Unordered set is an associative container that contains a set of unique objects of type Key. Search, insertion, and removal have average constant-time complexity.

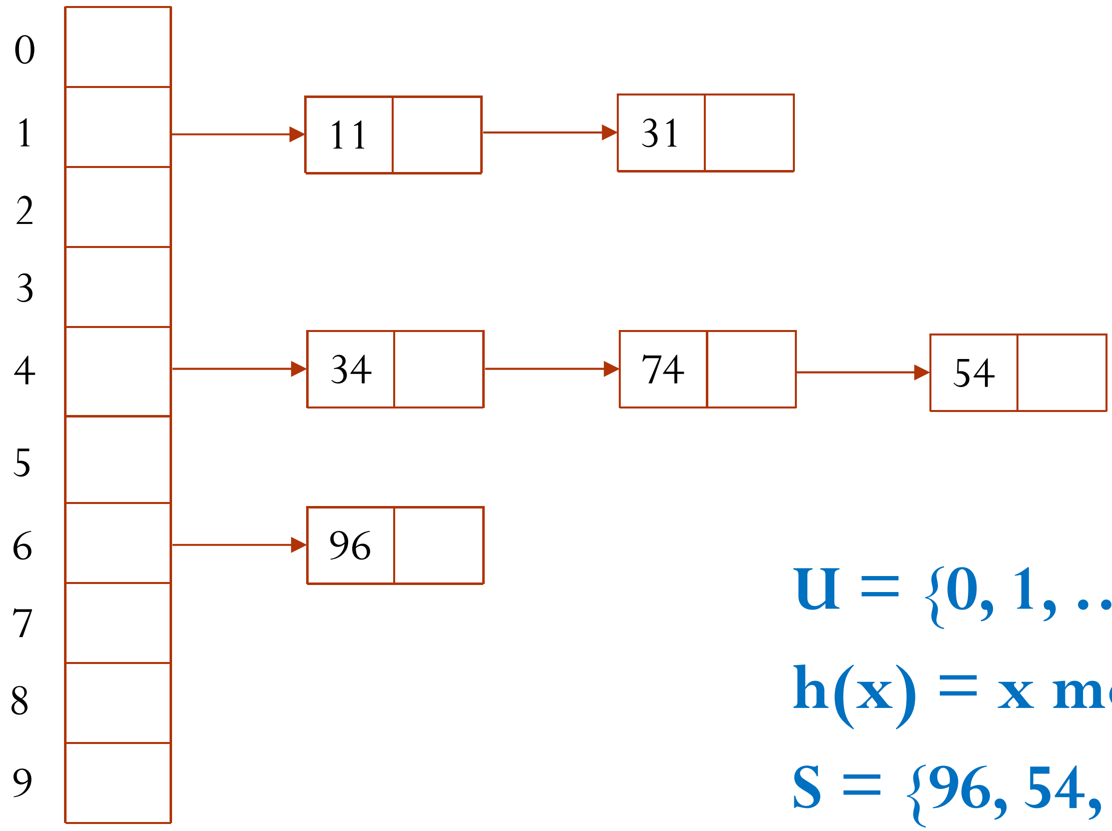
Dictionary problem

- Challenge:
  - The size of universe **U** is too large to store in an array of size |**U**|.
- Can we use the following data structures?
  - Bitvector
  - Linked list
  - Binary search tree
  - Hash table

Hashing

- Hash function:
  - $\mathbf{h: U \rightarrow \{0, 1, \ldots, n\text{-}1\}}$
- Hashing:
  - Create an array **A** of length **n**. For any operation on element $\mathbf{x} \in \mathbf{S}$, operate on the array position **A[h(x)]**
- Collision:
  - Two different elements **x** and **y** have the same hash value: **h(x) = h(y)**.
  - Chaining: **A[i]** is a linked list containing all elements **x** where **h(x) = i**.

Example of chain hashing



U = {0, 1, … , 99}

h(x) = x mod 10

S = {96, 54, 31, 11, 74, 34}

Dictionary operations

- Search(x):
  - Compute **h(x)**, then scan through the list **A[h(x)]**. Return true if finding **x**, and false otherwise.
- Insert(x):
  - Compute **h(x)**, then scan through the list **A[h(x)]** to find **x**. If **x** is not in the list, add **x** to the front of the list. Otherwise, do nothing.
- Delete(x):
  - Compute **h(x)**, then scan through the list **A[h(x)]** to find **x**. If **x** is in the list, remove **x** from the list. Otherwise, do nothing.

- Time complexity: **O**(**1** + length of **A[h(x)]**)

Deterministic hash function

- For any choice of **h**, we can find a set whose elements have the same hash values (map to one slot).
- We end up with a single linked list containing all elements. Hence, it takes **O(n)** time for the searching operation.
- Solutions:
  - Assume the input **S** is random.
  - Ideal hash function: Map **m** elements uniformly at random to **n** slots.

Hashing performance

- Ideal hash function:
  - Any **m** elements are uniformly distributed to **n** slots.
  - Average length of chain = **m/n**.
  - Choose **m ~ n**, we expect **O(1)** cost for insert, delete, search operations.

- We cannot have an ideal hash function for all possible sets of elements (true randomness might not exist).

- Approach: Choose the hash function **h** at random from a hash family **H**.

Universal hashing

- A universal family of hash functions **H** is a set of hash functions **h** mapping the universe **U** to a set $\{\mathbf{0, 1, \ldots, n\text{-}1}\}$ such that
  - For any different pair $\mathbf{x \neq y}$: $\mathbf{Pr_h[h(x) = h(y)] \leq 1/n}$, i.e. collision probability is very tiny.
  - Can select a random **h** efficiently
  - Can compute **h(x)** efficiently (in constant time)

h is chosen at random.

Example

- Universal property: If $\mathbf{x \neq y}$, $\mathbf{Pr_h[h(x) = h(y)] \leq 1/n}$.

Ex. $U = \{ a, b, c, d, e, f \}$, $n = 2$.

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1

$H = \{h_1, h_2\}$

$\Pr_{h \in H}[h(a) = h(b)] = 1/2$

$\Pr_{h \in H}[h(a) = h(c)] = 1$

$\Pr_{h \in H}[h(a) = h(d)] = 0$

...

not universal

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1
$h_3(x)$	0	0	1	0	1	1
$h_4(x)$	1	0	0	1	1	0

$H = \{h_1, h_2, h_3, h_4\}$

$\Pr_{h \in H}[h(a) = h(b)] = 1/2$

$\Pr_{h \in H}[h(a) = h(c)] = 1/2$

$\Pr_{h \in H}[h(a) = h(d)] = 1/2$

$\Pr_{h \in H}[h(a) = h(e)] = 1/2$

$\Pr_{h \in H}[h(a) = h(f)] = 0$

...

universal

Universal hash family for integer

- We construct a universal hash function **h** that hashes an integer $\mathbf{x} \in \mathbf{U}$ to a set of $\{\mathbf{0, 1, \ldots, n\text{-}1}\}$ as follows:
  - Pick a large prime number $\mathbf{p} \geq \mathbf{n}$
  - Pick a random pair of integers $\mathbf{0} \leq \mathbf{a, b} < \mathbf{p}$
  - $\mathbf{h_{a,b}(x) = ((ax + b) \bmod p) \bmod n}$

- Universal hash family: $\mathbf{H} = \{\mathbf{h_{a,b}: 0} \leq \mathbf{a, b} < \mathbf{p}\}$

- Proof of the universal property:
  - The 13.6 section of Algorithm Design textbook.
  - Carter & Wegman: "Universal Classes of Hash Functions". STOC'77.

Universal hash for a set of integers

- We construct a universal hash function **h** that hashes a set of integers $\mathbf{X} = \{\mathbf{x_1, \ldots, x_d}\}$ to a set $\{\mathbf{0, 1, \ldots, n\text{-}1}\}$ as follows:
  - Choose independently at random **d** hash functions $\mathbf{h_1, \ldots, h_d}$ from a universal hash family **H**.
  - $\mathbf{h(X) = h_1(x_1) + \ldots + h_d(x_d) \bmod n}$.
- Proof of universal property:
  - Carter & Wegman: "Universal Classes of Hash Functions". STOC'77.
- Practical hash function: For a large prime **p** and $\mathbf{0 \leq a_i < p}$,
  - $\mathbf{h_{a0,\ldots,ad}(X) = ((a_0 + a_1 x_1 + a_2 x_2 + \ldots + a_d x_d) \bmod p) \bmod n}$

Analysis of dictionary operations

- Time complexity: **O**(1 + length of **A[h(x)]**)
- Assumption: **|S|** $\leq$ **n** at all time, hence our dictionary uses **O(n)** space.
- The expectation of the length of the linked list **A[h(x)]**:
  - $$\mathbf{E}\left[\text{length of } A[h(\boldsymbol{x})]\right] = 1 + \mathbf{E}\left[|\boldsymbol{y} \in S \backslash \{\boldsymbol{x}\} \mid h(\boldsymbol{y}) = h(\boldsymbol{x})|\right]$$
    $$\leq 1 + (|S| - 1) * \frac{1}{n} \leq 2.$$
  - Expected time complexity of dictionary operations is **O(1)**.

Conclusion

- Given a random choice of a hash function **h** from the universal family such that **h:** $\mathbf{U} \rightarrow \{\mathbf{0, 1, \ldots, n\text{-}1}\}$**.**

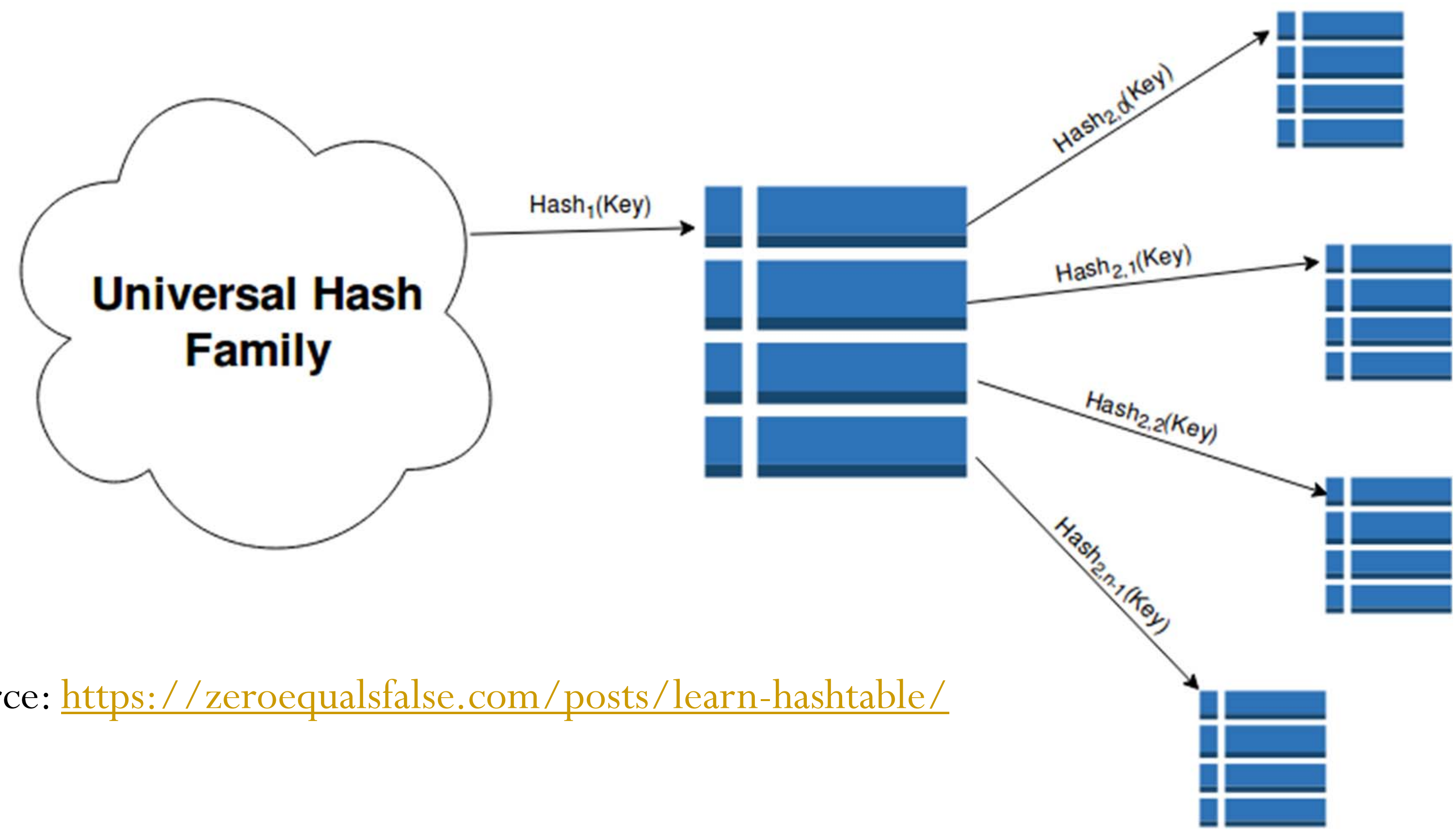
- Our dictionary needs **O(n)** space and **O(1)** expected time per operation (search, insert, delete).

- Expectation is over the random choice of hash function.

- Independent of input set.

Homework

- Write Python script to implement the universal hash family for a set of integers



Source: https://zeroequalsfalse.com/posts/learn-hashtable/