CSE 100: Algorithm Design and Analysis
Chapter 11: Hashing Tables

Sungjin Im

University of California, Merced

Last Update: 03-09-2023

There is no great genius without some touch of madness.
Aristotle

Outline

- Understanding hashing/hash table.
  - Advantages over direct-access tables.
- A method resolving collisions: chaining, open addressing.
- Universal Hashing Family. Science behind hashing

Hash Tables

- Dictionary operations: Insert, Search, Delete.
- Hash table is effective for implementing a dictionary;
  - Each operation takes $O(1)$ time in expectation under some 'reasonable' assumptions.
  - Each operation provably takes $O(1)$ time in expectation using universal hash family.
- Hash table is a generalization of an ordinary array (direct addressing).

Hash Tables

Direct-address tables

Scenario:

- Each element has a key drawn from a universe $U = \{0, 1, 2, ..., m-1\}$ where m is VERY large.
- No two elements have the same key.

Direct-address table $T[0 \cdots m-1]$:

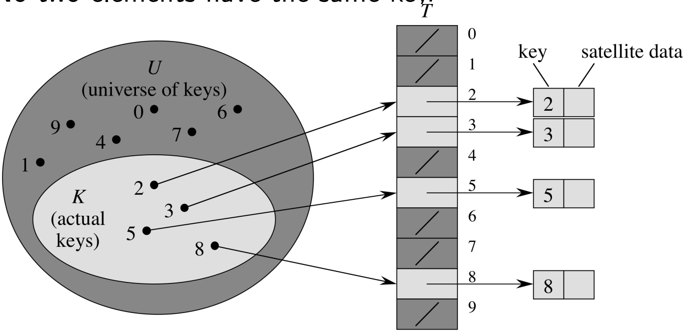
- If there's an element x with key k, then $T[k]$ contains a pointer to x.
- Otherwise, $T[k] = NIL$.

Hash Tables

Direct-address tables

Scenario:

- Each element has a key drawn from a universe $U = \{0, 1, 2, \ldots, m-1\}$ where m is VERY large.
- No two elements have the same key.



Hash Tables

Direct-address tables

```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]
```

```
DIRECT-ADDRESS-INSERT(T, x)
    T[x.key] = x
```

```
DIRECT-ADDRESS-DELETE(T, x)
    T[x.key] = NIL
```

Each of these operations takes $O(1)$ time.

Hash Tables

Direct-address tables vs. Hash tables

Direct-address tables:
Simple and Fast.

Hash Tables

Direct-address tables vs. Hash tables

Direct-address tables:
Simple and Fast. But a lot of space will be wasted if the *universe* U is too large compared to the set K of keys *actually stored*.

Hash Tables

Direct-address tables vs. Hash tables

Direct-address tables:
Simple and Fast. But a lot of space will be wasted if the *universe* U is too large compared to the set K of keys *actually stored*.

Hashing table:
Can reduce storage requirements to $\Theta(|K|)$.

Hash Tables

Direct-address tables vs. Hash tables

Direct-address tables:
Simple and Fast. But a lot of space will be wasted if the *universe* U is too large compared to the set K of keys *actually stored*.

Hashing table:
Can reduce storage requirements to $\Theta(|K|)$.
Can still get $O(1)$ time, but only in expectation.

Hash Tables

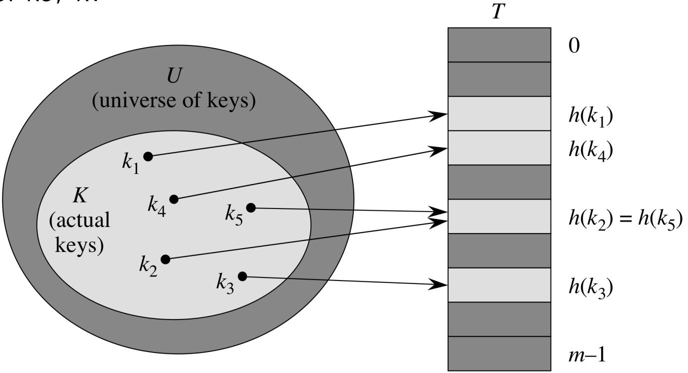
Main idea:
Use a hash function h and store an element of key k in slot $h(k)$.

- $h : U \rightarrow \{0, 1, \cdots, t-1\}$; hash table has t slots/buckets
- We say that k *hashes* to slot $h(k)$, or $h(k)$ is the hash value of key k.

Hash Tables

Main idea:
Use a hash function h and store an element of key k in slot $h(k)$.

- $h : U \to \{0, 1, \cdots, t-1\}$; hash table has t slots/buckets
- We say that k *hashes* to slot $h(k)$, or $h(k)$ is the hash value of key k.



Hash Tables

Collisions

However, two or more keys may hash to the same slot.

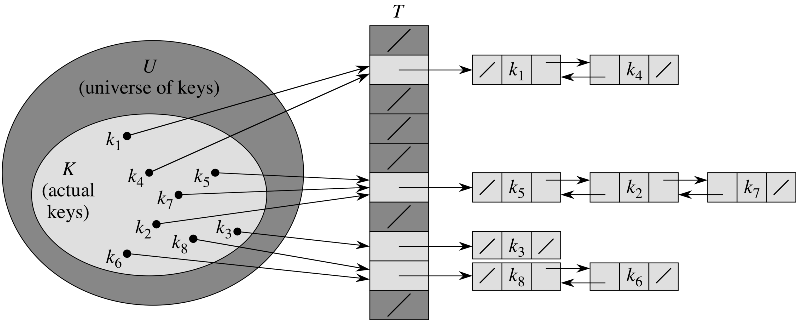
- If $|K| > t$, then a collision must occur.
- If $|K| \leq t$, then a collision may or may not occur.

We will learn one method to resolve collisions: chaining.

Hash Tables

Collision resolution by chaining

Put all elements that hash to the same slot into a linked list.



Slot j contains a pointer to the head of the list of all stored elements that hash to j;if there are no such elements, slot j contains NIL.

* Use doubly linked lists if you want to support delete operations.

Hash Tables

Chaining

Chained-Hash-Insert(T, x)
insert x at the head of list $T[h(x.key)]$.

Worst-case running time is

Hash Tables

Chaining

Chained-Hash-Insert(T, x)
insert x at the head of list $T[h(x.key)]$.

Worst-case running time is $O(1)$

Hash Tables

Chaining

Chained-Hash-Insert(T, x)
insert x at the head of list $T[h(x.key)]$.

Worst-case running time is $O(1)$ assuming that x isn't already in the list. If not, need to search the list first.

Hash Tables

Chaining

Chained-Hash-Insert(T, x)
insert x at the head of list $T[h(x.key)]$.

Worst-case running time is $O(1)$ assuming that x isn't already in the list. If not, need to search the list first. So the asymptotic run time will be equal to Search

Hash Tables
Chaining

Chained-Hash-Search(T, k)
search for an element with key k in list $T[h(k)]$.

Running time is

Chained-Hash-Search(T, k)
search for an element with key k in list $T[h(k)]$.

Running time is proportional to the list length of slot $h(k)$.

Chained-Hash-Delete(T, x)
delete x from the list $T[h(x.key)]$.

We assume that pointer x is given.

Chained-Hash-Delete(T, x)
delete x from the list $T[h(x.key)]$.

We assume that pointer x is given. So no need to search for it.
Worst case running time:

- if the list is doubly linked: $O(1)$.
- if the list is singly linked: could be as long as search.

(Without the pointer, we need to search it first.)

Hash Table

Chaining: running time of search

In the worst case

Hash Table

Chaining: running time of search

In the worst case when all elements hash to the same slot,

Hash Table

Chaining: running time of search

In the worst case when all elements hash to the same slot, it could be $\Theta(n)$.

In the average case:

Hash Table

Chaining: running time of search

In the worst case when all elements hash to the same slot, it could be $\Theta(n)$.

In the average case: Load factor α for table T is defined as n/t

- $|K| = n$: # of elements stored
- t: # of slots.

i.e. $\alpha =$ the average number of elements stored in a chain under the simple uniform assumption that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.
The running time is

Hash Table

Chaining: running time of search

In the worst case when all elements hash to the same slot, it could be $\Theta(n)$.

In the average case: Load factor α for table T is defined as n/t

- $|K| = n$: # of elements stored
- t: # of slots.

i.e. $\alpha =$ the average number of elements stored in a chain under the simple uniform assumption that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.
The running time is $\Theta(1 + \alpha)$.

Hash Table

Chaining: running time of search

Theorem

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1 + \alpha)$ *under the assumption of simple uniform hashing.*

Theorem

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1 + \alpha)$ *under the assumption of simple uniform hashing.*

Science behind hashing (with no assumptions)

Goal

Recall: load factor $\alpha := n/t$.

Goal: for *any* input (any sequence of insert, delete, search operations), we want to have $O(\alpha + 1)$ run time per each operation in expectation. This is different from "for some good inputs."

Science behind hashing (with no assumptions)

Goal

Recall: load factor $\alpha := n/t$.

Goal: for *any* input (any sequence of insert, delete, search operations), we want to have $O(\alpha + 1)$ run time per each operation in expectation. This is different from "for some good inputs."

More Formal Goal: The hypothetical adversary chooses an arbitrary instance. We choose a hash function $h : U \to \{0, 1, 2, ..., t - 1\}$ at random. The expected running time of each operation in the input is $O(\alpha + 1)$.

Science behind hashing (with no assumptions)

Goal

Recall: load factor $\alpha := n/t$.

Goal: for *any* input (any sequence of insert, delete, search operations), we want to have $O(\alpha + 1)$ run time per each operation in expectation. This is different from "for some good inputs."

More Formal Goal: The hypothetical adversary chooses an arbitrary instance. We choose a hash function $h : U \to \{0, 1, 2, ..., t-1\}$ at random. The expected running time of each operation in the input is $O(\alpha + 1)$.

Why equivalent?

Science behind hashing (with no assumptions)

Goal

Recall: load factor $\alpha := n/t$.

Goal: for *any* input (any sequence of insert, delete, search operations), we want to have $O(\alpha + 1)$ run time per each operation in expectation. This is different from "for some good inputs."

More Formal Goal: The hypothetical adversary chooses an arbitrary instance. We choose a hash function $h : U \to \{0, 1, 2, ..., t - 1\}$ at random. The expected running time of each operation in the input is $O(\alpha + 1)$.

Why equivalent?
The user typically doesn't know where we store each key value. She is provided only the interface "insert(key), search(key), delete(key)." So, it is equivalent to creating an instance without seeing the hash function we will choose.

What if the adversary knows the hash function we chose? Can we still have $O(\alpha + 1)$ expected running time?

Science behind hashing

Oblivious vs non-oblivious adversary

What if the adversary knows the hash function we chose? Can we still have $O(\alpha + 1)$ expected running time?
No. Then the adversary can try to insert keys that hash to the same slot/bucket

Science behind hashing

Setup

- m: the universe size.
- n: number of keys that actually occur.
- t: the number of slots in the hash table.

Assume that t is more or less equal to n if it helps you.

Science behind hashing

Key question

From which hash family should we choose a hash function at random?

Science behind hashing

Key question

From which hash family should we choose a hash function at random? Key criteria:

- compactness
- universal hash family (similar to pairwise independence)

(If we don't use randomness, we can't get $O(\alpha + 1)$ running time for some inputs. Do you see why?)
(Choosing a random function can be viewed as a choosing one from a family of functions.)

Science behind hashing

Why compactness matters

What goes wrong if we choose a function from a big family $\mathcal{H}$ of hash functions?

Science behind hashing

Why compactness matters

What goes wrong if we choose a function from a big family $\mathcal{H}$ of hash functions?

We need to store the hash function. That is, once we have chosen h, then $h(k)$ should have the same value all the time. Otherwise, we may not be able to find a key we have inserted! To remember which function we have chosen, say we index all functions in the family. Then, the index can be as large as $\mathcal{H}$, so, we will need at least $\log_2 |\mathcal{H}|$ bits. This means we would need at least $\log_2 |\mathcal{H}|$ bits to store the hash function. If this is large, there's no point of using hashing.

Science behind hashing

Universal Hash Family

If for any *pair* of two distinct keys x and y, $\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] =$ [1] $1/t$, then we say $\mathcal{H}$ is a universal hash family. In other words, for any two different key values, if we choose h from $\mathcal{H}$ uniformly at random, they hash to the same slot with probability equal to the inverse of the table size.

[1] or $\leq$

Science behind hashing

Universal Hash Family

If for any *pair* of two distinct keys x and y, $\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] =$ [2] $1/n$, then we say $\mathcal{H}$ is a universal hash family. In other words, for any two different key values, if we choose h from $\mathcal{H}$ uniformly at random, they hash to the same slot with probability equal to the inverse of the number of elements. (definition is slightly different from the textbook.)

[2] or $\leq$

Science behind hashing

Universal Hash Family: why is it good?

If for any *pair* of two distinct keys u and v, $\Pr_{h \sim \mathcal{H}}[h(u) = h(v)] = 1/t$, then we say $\mathcal{H}$ is a universal hash family.

- For an arbitrary key u, we want to know the expected number of other keys v that collide with u.
- Let $C(u, v)$ denote the 0-1 random variable that has value 1 iff u and v collide.
- We want to bound $E[\sum_{v \neq u} C(u, v)]$.
- $E[C(u, v)] = Pr[h(u) = h(v)] = 1/t$ thanks to the property of universal hashing.
- $E[\sum_{v \neq u} C(u, v)] = \sum_{v \neq u} E[C(u, v)] \leq (n - 1)/t \leq \alpha$.
- Thus, the u's bucket/slot has at most $\alpha + 1$ keys in expectation.

Science behind hashing

Suppose we choose a function at random from all possible functions from U to $\{0, 1, ..., t-1\}$, what goes wrong?

Science behind hashing

Suppose we choose a function at random from all possible functions from U to $\{0, 1, ..., t-1\}$, what goes wrong?
$\mathcal{H}$ is too big. There are t^m possible functions. So we need $m \log t$ bits to store the hash function we've chosen, as discussed before.
But, this is a universal hash family.

Science behind hashing

good or bad?

Suppose we sample a function from
$\mathcal{H} := \{h(u) = 0 \forall u, h(u) = 1 \forall u, ..., h(u) = t - 1 \forall u\}$. What is wrong?

Science behind hashing

good or bad?

Suppose we sample a function from $\mathcal{H} := \{h(u) = 0 \forall u, h(u) = 1 \forall u, ..., h(u) = t - 1 \forall u\}$. What is wrong?
It's compact. We only need $\lg t$ bigs to store the hash function.
But, this is terrible, since for any input, all keys hash to the same slot. So, the running time per each operation will be $O(n)$. And what is $\Pr_{h \sim \mathcal{H}}[h(u) = h(v)]$ for $u \neq v$?

Science behind hashing

Universal Hash Family Example

Say p is a prime number that is greater than m such that $p = \Theta(m)$.
$\mathcal{H} := \{a \in \{1, 2, ..., p-1\}, b \in \{0, 1, 2, ..., p-1\} \mid (ax + b \text{ mod p}) \text{ mod t})\}$.

Science behind hashing

Universal Hash Family Example

Say p is a prime number that is greater than m such that $p = \Theta(m)$.
$\mathcal{H} := \{a \in \{1, 2, ..., p-1\}, b \in \{0, 1, 2, ..., p-1\} \mid (ax + b \text{ mod p}) \text{ mod t})\}$.

And it is also compact! Uses only $O(\log m)$ bits.

Science behind hashing

Universal Hash Family Example

Say p is a prime number that is greater than m such that $p = \Theta(m)$.
$\mathcal{H} := \{a \in \{1, 2, ..., p-1\}, b \in \{0, 1, 2, ..., p-1\} \mid (ax + b \bmod p) \bmod t)\}$.

Science behind hashing

Universal Hash Family Example

Say p is a prime number that is greater than m such that $p = \Theta(m)$.
$\mathcal{H} := \{a \in \{1, 2, ..., p-1\}, b \in \{0, 1, 2, ..., p-1\} \mid (ax + b \bmod p) \bmod t)\}$.

In other words, choose a and b from $\{1, 2, ..., p-1\}$ and $\{0, 1, 2, ..., p-1\}$ uniformly at random, respectively. Then, let $h_{a,b}(x) := (ax + b \bmod p) \bmod t)$. Key u hashes to slot $h_{a,b}(u)$.