05. Hash Tables

CPSC 535 ∼ Spring 2019

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

February 25, 2019

Big Idea in Algorithm Design

Recall that radix sort takes $\Theta(n)$ time

- linear time is extremely fast
- same asymptotically as merely observing all the elements
- under the assumptions of radix sort, sorting is "free"; no significant computation time

Natural questions:

- How far can we take this?
- What is the most powerful way we can organize data, and still only spend $O(1)$ time per element?
- What are the trade-offs (assumptions) and are they worth it?

The Story So Far

- *offline:* all operations performed at once; e.g. BUILD-HEAP
- *online:* data structure valid after each individual operation; e.g. DELETE-MIN
- computing order offline = sorting problem = $\Omega(n \log n)$ lower bound = $\Omega(\log n)$/element
- maintaining order online = search tree = $\Theta(\log n)$/element
- sorting $\underline{\text{integers}}$ = radix sort = $\Theta(n)$ total = $\Theta(1)$/element; subjectively more complex
- Question: what about maintaining order $\underline{\text{of integers}}$ online, $\Theta(1)$ per element?

Introducing Hash Tables

- (later) van Emde Boas trees can maintain order of integers in $\Theta(\log \log n)$ per element (huh?)
- (today) hash tables can't maintain order, but can support dictionary operations INSERT/SEARCH/DELETE, in $\Theta(1)$/element **expected** time
- hash tables match runtime of arrays, but (again) are more complicated to understand and implement
- success story for algorithm design
- mainstream; taken for granted in CS systems design
- GZIP, DNS, Java, Python, JSON, . . .

Dictionary Operations

CREATE(T): intialize T as a valid dictionary

INSERT(T, k, v): associate key k and value v in dictionary T; k must not already be in T

SEARCH(T, k): return the value v associated with key k in dictionary T; or return NIL if k is absent from T

DELETE(T, k): remove key k and its associated value from dictionary T; k must already be in T

(Practical implementations are often more flexible about re-insertions and inneffectual deletions.)

Direct Address Table

Suppose the *universe* of keys is $k \in U = \{0, 1, \ldots, m-1\}$ for fixed m; create m-element array; $T[k] = v$ (or NIL if k is absent)

```
1: function DA-CREATE(T)
2:     $T[0 \ldots m-1] = $ NIL
3: end function
1: function DA-INSERT(T, k, v)
2:     $T[k] = v$
3: end function
```

```
1: function DA-SEARCH(T, k)
2:     return $T[k]$
3: end function
1: function DA-DELETE(T, k)
2:     $T[k] = $ NIL
3: end function
```

$\Theta(m)$ space; CREATE is $\Theta(m)$ time; rest are $\Theta(1)$ time; good when $m \in O(n)$ but m could be exponential in n

Hash Functions

- U = set of keys, m = size of array
- $m = |U|$ impractical in general
- introduce *hash function* $h : U \mapsto \{0, 1, \ldots, m-1\}$
- h “compresses” large space of keys U into denser space of table indices $\leq m$
- *collisions* exist, i.e. $k_1, k_2 \in U$, $k_1 \neq k_2$, such that

  $$h(k_1) = h(k_2)$$

  (pigeonhole principle)
- hard part is designing a concrete h, and collision resolution strategy

Collision Resolution by Chaining

Chaining: each $T[i]$ is a linked list of (colliding) key-value entries

```
1: function CHAINED-CREATE(T)
2:     $T[0 \ldots m-1] =$ empty list
3: end function
```

```
1: function CHAINED-INSERT(T, k, v)
2:     insert $(k, v)$ at front of $T[h(k)]$
3: end function
```

```
1: function CHAINED-SEARCH(T, k)
2:     search for $k$ in $T[h(k)]$
3: end function
```

```
1: function DA-DELETE(T, k)
2:     remove $k$ from $T[h(k)]$
3: end function
```

CREATE is $O(m)$; INSERT is $O(1)$; SEARCH and DELETE are $O(|T[h(k)]|)$

Chaining Analysis

SEARCH, DELETE take $\Theta(n)$ time in worst case

Let $\alpha = n/m =$ average keys/table element $=$ **load factor**

simple uniform hashing assumption: any element is equally likely to hash to each index, independent of other elements

Under this assumption, expected chain length is $O(1+\alpha)$
$\implies$ SEARCH, DELETE take $O(1+\alpha)$ expected time each

To achieve $O(1+\alpha) = O(1)$ exp. time, choose $m \in \Omega(n)$

e.g. choose $m \geq 2n$ so

$$\alpha = \frac{n}{m} \leq \frac{n}{(2n)} = \frac{1}{2}$$

Deterministic Hash Functions

First: map keys to natural numbers $\{0, 1, \ldots\}$;
henceforth assume $k \in \mathbb{N}$

Division method: $h(k) = k \mod m$,
for appropriate m
e.g. m is a prime "not too close to" a power of 2

Multiplication method: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
for carefully-chosen $0 < A < 1$
e.g. $A \approx (\sqrt{5} - 1)/2$

Each is fast, but deterministic, so an adversary could craft a set of keys that force $\Theta(n)$/element $\implies \Theta(n^2)$ to insert n elements

Hash Collision Exploits

Denial-of-service attacks based on hash collisions are a real thing

- CVE-2011-4885
- working exploit

Universal Hashing

Goal: prevent hash collision exploit

Hash function not compiled in; picked randomly at runtime (once per boot, execution, or table creation); unknowable to adversary

A set $\mathcal{H}$ of hash functions is **universal** when

- for any distinct keys $k, l \in U$,
- the number of hash functions for which k and l collide, is at most
$$\frac{|\mathcal{H}|}{m}$$
- i.e. if h is picked randomly from $\mathcal{H}$, then for **any** keys k, l,

$$Pr[h(k) = h(l)] = Pr[\text{two random table indices match}] = \frac{1}{m}$$

Universal Hashing Analysis

Theorem: If hash function h is chosen randomly from a universal family; T is a chained hash table using h, with m table elements, containing n keys, and with load factor $\alpha = n/m$; k is a query key that hashes to list ℓ; then the expected length of ℓ is α if $k \in T$, or $(1+\alpha)$ if $k \notin T$.

Note: the hash function type and collision resolution algorithm need to be analyzed together.

Corollary: In a hash table using a universal hash function and chaining collision resolution, INSERT takes $O(1)$ time, while SEARCH and DELETE each take $O(1)$ expected time and $\Theta(n)$ worst-case time.

An Example Universal Hash Family

Fix p as a prime number greater than every possible key k

Let a, b be int. parameters with $1 \leq a < p$ and $0 \leq b < p$; define

$$h_{ab}(k) \equiv ((ak + b) \mod p) \mod m.$$

Number theory shows this is a universal family.

Conveniences

- can hardcode p as a prime near $2^W - 1$
- generating h_{ab} only involves choosing two random int's $< p$
- evaluating h_{ab} only involves one integer multiply, one add, and two modulo operations
- can recalibrate a fixed h_{ab} to any m

Open Addressing

Concern with chaining: linked lists have poor *locality of reference*, require expensive allocate/free operations, and are conceptually complicated.

open addressing: all key-value entries are stored direclty in the table T itself; there are no chains

Modest assumptions

- each $T[i]$ can store either an active (k, v) entry; or NIL (meaning never used); or TOMBSTONE (meaning was once active, but is now empty)
- $n < m$, i.e. $\alpha < 1$

Open Addressing Delete

DELETE:

- let $i = h(k)$; **probe** (examine) $T[i]$
- if k is found, $T[i] =$ TOMBSTONE, done
- else if $T[i]$ is NIL, conclude $k \notin T$, done
- else ($T[i]$ is TOMBSTONE or key other than k), probe a different index
- after m probes, conclude $k \notin T$, stop (otherwise infinite loop)

probe sequence: sequence of indices probed; different approaches

$O(|\text{probe sequence}|)$ time

Open Addressing Search and Insert

SEARCH:

- probe in the same order as DELETE
- keep searching past TOMBSTONE elements

INSERT:

- probe in the same order as DELETE
- overwrite the first $T[i]$ containing NIL or TOMBSTONE

Probe Sequence Functions; Linear Probing

Hash function: $h : U \mapsto \{0, 1, \ldots, m-1\}$
Probe sequence function:
$h : U \times \{0, 1, \ldots, m-1\} \mapsto \{0, 1, \ldots, m-1\}$, i.e.

$$h(k, i) = i\text{th index to probe for key } k$$

Linear probing: probe in sequential order, so

$$ps(k, i) = (h(k) + i) \mod m$$

Simple; excellent locality of reference; but develops long **clusters** of occupied elements; an empty slot after ℓ occupied slots will be filled with probability $(\ell + 1)/m$

Quadratic Probing

Quadratic probing:

$$ps(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m$$

for constant parameters c_1, c_2 (with some constraints)

Much less clustering than linear probing; but still some, because if k_1, k_2 collide because $h(k_1) = h(k_2)$, then their entire probe sequences collide.

$\Longrightarrow$ partially mitigates the clustering problem; but the *ps* function is more expensive, and locality of reference is not as good

Double Hashing

Double hashing:

$$ps(k, k) = (h_1(k) + i \cdot h_2(k)) \mod m$$

where h_1, h_2 are two distinct hash functions

Since h_1, h_2 are different, even if k_1, k_2 collide at first with $h_1(k_1) = h_1(k_2)$, their probe sequences are almost certainly different
$\implies$ clustering problem essentially solved; but more expensive than linear/quadratic programming because we evaluate two hash functions, and locality of reference is poor (random), though still entirely within T

Open Addressing Analysis

Universal hashing assumption: the probability of every possible probe sequence $\langle p_1, p_2, \langle, p_m \rangle$ with each $p_i \in \{0, 1, \ldots, m_1\}$ is equally likely.

Theorem: Under the universal hashing assumption, an open-address hash table with load factor $\alpha < 1$ performing a search makes at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

probes when the search succeeds, or at most

$$\frac{1}{1-\alpha}$$

probes when the search fails.

Corollary: Inserting requires at most $\frac{1}{1-\alpha}$ probes on average.

Corollary: If $\alpha \in O(1)$ then the INSERT, SEARCH, and DELETE operations on an open-addressing hash table each take $O(1)$

Perfect hashing

Static hashing: set of keys is known statically (algorithm design-time or program compile-time)

- CREATE, INSERT happen offline (all-at-once) at compile-time
- SEARCH happens online at run-time
- DELETE not supported

Perfect hashing: static hashing where SEARCH $O(1)$ worst case time (**not** $O(1)$ expected time)
Applies when key set is constant, or evolves very slowly: top-level domains, programming language keywords, state abbreviations (CA), etc.

Two-Level Hashing

Hash-table-of-hash-tables:

- *outer* hash table w/ function h
- slot $T[j]$ holds an *inner* table using function h_j
- all hash functions h, h_j are universal
- if $T[j]$ holds n_j colliding keys, then its size m_j must be $\Omega(n_j^2)$
- non-obvious: large n_j^2 size prevents collisions in the inner tables but the whole table still only takes $O(n)$ space

Two-Level Collisions

Theorem: if inner hash table $T[j]$ uses a universal hash function h_j and has $m_j = n_j^2$, then the probability of a collision in $T[j]$ is less than $\frac{1}{2}$
Sketch: there are $\binom{n_j}{2}$ pairs of keys; due to universal h_j, each collides with probability $\frac{1}{m}$; so

$$E[\#\text{ collisions}] = \binom{n_j}{2} \cdot \frac{1}{m_j} = \binom{n_j}{2} \cdot (\frac{1}{n_j^2}) = (\frac{n_j^2 - n}{2}) \cdot \frac{1}{n_j^2} < \frac{1}{2}.$$

To pick an h_j, keep retrying until no collisions $\implies O(1)$ expected trials

Two-Level Analysis

Theorem: total space used by a two-level hash is $O(n)$
Sketch:

- outer hash uses $O(n)$ space
- each $T[j]$ has n_j elements and uses $\Theta(n_j^2)$; space
- $O(1)$ collisions in $T[j] \implies n_j \in O(1) \implies n_j^2 \in O(1)$

Takeaway: two-level perfect hash table takes $O(n)$ expected time to create statically; a SEARCH takes $O(1)$ deterministic worst-case time at runtime.