Module 7: Hashing

CS 240 - Data Structures and Data Management

Sajed Haque Veronika Irvine Taylor Smith
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2017

Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing n items. Is this the best possible?

Theorem: In the comparison model (on the keys),
$\Omega(\log n)$ comparisons are required to search a size-n dictionary.

Proof:

□

Direct Addressing

Requirement: For a given $M \in \mathbb{N}$,
every key k is an integer with $0 \leq k < M$.

Data structure : An array of *values* A with size M

search(k) : Check whether $A[k]$ is empty

insert(k, v) : $A[k] \leftarrow v$

delete(k) : $A[k] \leftarrow$ *empty*

Each operation is $\Theta(1)$.
Total storage is $\Theta(M)$.

What sorting algorithm does this remind you of?

Hashing

Direct addressing isn't possible if keys are not integers.
And the storage is very wasteful if $n \ll M$.

Say keys come from some *universe* U.
Use a *hash function* $h : U \to \{0, 1, \ldots, M-1\}$.
Generally, h is not injective, so many keys can map to the same integer.

Hash table Dictionary: Array T of size M (the *hash table*).
An item with key k is stored in $T[h(k)]$.
search, *insert*, and *delete* should all cost $O(1)$.

Challenges:

- Choosing a good hash function (later)
- Dealing with *collisions* (when $h(k_1) = h(k_2)$)

Collision Resolution

Even the best hash function may have *collisions*:
when we want to insert (k, v) into the table,
but $T[h(k)]$ is already occupied.

Two basic strategies:

- Allow multiple items at each table location (buckets)
- Allow each item to go into multiple locations (open addressing)

We will examine the average cost of *search*, *insert*, *delete*,
in terms of n, M, and/or the *load factor* $\alpha = n/M$.

We probably want to rebuild the whole hash table and change
the value of M when the load factor gets too large or too small.
This is called *rehashing*, and should cost roughly $\Theta(M + n)$.

Chaining

Each table entry is a *bucket* containing 0 or more KVPs.
This could be implemented by any dictionary (even another hash table!).

The simplest approach is to use an unsorted linked list in each bucket.
This is called collision resolution by *chaining*.

- *search*(k): Look for key k in the list at $T[h(k)]$.
- *insert*(k, v): Add (k, v) to the front of the list at $T[h(k)]$.
- *delete*(k): Perform a search, then delete from the linked list.

Chaining example

$M = 11$, $h(k) = k \bmod 11$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Complexity of chaining

Recall the load balance $\alpha = n/M$.

Assuming uniform hashing, average bucket size is exactly α.

Analysis of operations:

search $\Theta(1 + \alpha)$ average-case, $\Theta(n)$ worst-case

insert $O(1)$ worst-case, since we always insert in front.

delete Same cost as *search*: $\Theta(1 + \alpha)$ average, $\Theta(n)$ worst-case

If we maintain $M \in \Theta(n)$, then average costs are all $O(1)$.
This is typically accomplished by rehashing whenever $n < c_1 M$ or $n > c_2 M$, for some constants c_1, c_2 with $0 < c_1 < c_2$.

Open addressing

Main idea: Each hash table entry holds only one item,
but any key k can go in multiple locations.

search and *insert* follow a *probe sequence* of possible locations for key k:
$\langle h(k, 0), h(k, 1), h(k, 2), \ldots \rangle$.

delete becomes problematic; we must distinguish between
empty and *deleted* locations.

Simplest idea: *linear probing*
$h(k, i) = (h(k) + i) \bmod M$, for some hash function h.

Linear probing example

$M = 11, \qquad h(k) = k \bmod 11$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Double Hashing

Say we have **two** hash functions h_1, h_2 that are **independent**.

So, under uniform hashing, we assume the probability that a key k has $h_1(k) = a$ and $h_2(k) = b$, for any particular a and b, is

$$\frac{1}{M^2}.$$

For *double hashing*, define $h(k, i) = h_1(k) + i \cdot h_2(k) \bmod M$.

search, *insert*, *delete* work just like for linear probing, but with this different probe sequence.

Double hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor k/2 \rfloor \bmod 11$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Cuckoo hashing

This is a relatively new idea from Pagh and Rodler in 2001.

Again, we use two independent hash functions h_1, h_2.
The idea is to *always* insert a new item into $h_1(k)$.
This might "kick out" another item, which we then attempt to re-insert into its alternate position.

Insertion might not be possible if there is a loop.
In this case, we have to rehash with a larger M.

The big advantage is that an element with key k can only be in $T[h_1(k)]$ or $T[h_2(k)]$.

Cuckoo hashing insertion

```
cuckoo-insert(T,x)
T: hash table,  x: new item to insert
1.   y ← x,  i ← h1(x.key)
2.   do at most n times:
3.       swap(y, T[i])
4.       if y is "empty" then return "success"
5.       if i = h1(y.key) then i ← h2(y.key)
6.       else i ← h1(y.key)
7.   return "failure"
```


Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

0	44
1	
2	
3	
4	26
5	
6	
7	
8	
9	92
10	

Choosing a good hash function

Uniform Hashing Assumption: Each hash function value is equally likely.

Proving is usually impossible, as it requires knowledge of the input distribution and the hash function distribution.

We can get good performance by following a few rules.

A good hash function should:

- be very efficient to compute
- be unrelated to any possible patterns in the data
- depend on all parts of the key

Basic hash functions

If all keys are integers (or can be mapped to integers), the following two approaches tend to work well:

Division method: $h(k) = k \bmod M$.
We should choose M to be a prime.

Multiplication method: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$,
for some constant floating-point number A with $0 < A < 1$.
Knuth suggests $A = \varphi = \dfrac{\sqrt{5}-1}{2} \approx 0.618$.

Multi-dimensional Data

What if the keys are multi-dimensional, such as strings?

Standard approach is to *flatten* string w to integer $f(w) \in \mathbb{N}$, e.g.

$$\begin{aligned} A \cdot P \cdot P \cdot L \cdot E &\rightarrow 65 \cdot 80 \cdot 80 \cdot 76 \cdot 69 \quad \text{(ASCII)} \\ &\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R^1 + 68R^0 \\ &\qquad \text{(for some radix } R\text{, e.g. } R = 255) \end{aligned}$$

We combine this with a standard hash function
$h : \mathbb{N} \rightarrow \{0, 1, 2, \ldots, M-1\}$.

With $h(f(k))$ as the hash values, we then use any standard hash table.

Note: computing each $h(f(k))$ takes Ω(length of w) time.

Hashing vs. Balanced Search Trees

Advantages of Balanced Search Trees

- $O(\log n)$ worst-case operation cost
- Does not require any assumptions, special functions, or known properties of input distribution
- No wasted space
- Never need to rebuild the entire structure

Advantages of Hash Tables

- $O(1)$ cost, but only on average
- Flexible load factor parameters
- Cuckoo hashing achieves $O(1)$ worst-case for search & delete