Hash tables

Textbook, Chapter 12, Sections 12.1, 12.2, 12.3

CSCE310: Data Structures and Algorithms

www.cse.unl.edu/~choueiry/S01-310/

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104

choueiry@cse.unl.edu, Tel: (402)472-5444

B. Y. Choueiry

March 9, 2001
Dynamic set (i.e., dictionary) operations required by many applications:

1. Insert
2. Search
3. Delete

→ A hash table is an effective data structure
   basic operations in $O(1)$ on average

**Worst case:** as bad as a linked list ($\Theta(n)$)

**In practice:** extremely competitive (nearly constant)
   basic operations in $O(1)$ on average
Hash table: A generalization of notion of an ordinary array

Array uses direct addressing, which

- allows access to an arbitrary position in $O(1)$
- requires one position for every possible key

Hash table

- does not use key as array index, but computes array index from key
- is advantageous when $\#\text{keys actually stored} \ll \#\text{keys possible}$
- uses an array of size proportional to $\#\text{of keys stored}$
- object can be stored in slot itself (instead of pointer)
Direct-access tables = array, $T[0, 1, \ldots, m - 1]$

Each element has a key drawn from $U = \{0, 1, \ldots, m - 1\}$
Assumption: no two elements have the same key

- Slot $k$ points to the element in set with key $k$
- When no element in set with key $= k$, $T[k] = \text{Nil}$
- Works well when $U$, universe of keys, is small
Dictionary operations

Direct-Address-Search \((T, k)\) return \(T[k]\)

Direct-Address-Insert \((T, x)\) return \(T[\text{key}][x] \leftarrow x\)

Direct-Address-Delete \((T, x)\) return \(T[\text{key}][x] \leftarrow \text{Nil}\)

\(O(1)\)
Direct-access table

- When $\mathcal{U}$ is large, storing $T$ of size $\mathcal{U}$ is impractical
- When $k << \mathcal{U}$, lots of space wasted

$\rightarrow$ do not use a direct-access table
$\rightarrow$ use a hash table

Hash table

- Storage requirement can be reduced to $\Theta(k)$
- Searching remains in $\Theta(1)$, however, *in average*
**Principle**

**Direct-access table:** stores element with key $k$ in slot $k$

**Hash table:** stores element with key $k$ in slot $h(k)$

$h$, *hashing function*, maps universe into slots

$h : U \rightarrow \{0, 1, \ldots, m - 1\}$

An element with key $k$ hashes to slot $h(k)$

**Collision:**

two keys may hash to the same slot (when $h$ not injective)
Avoiding collision

Make $h$ appear to be random: avoids or minimizes collisions

- $h$ must be deterministic: given $k$, $h(k)$ same
- Since $U > m$, no-collisions is impossible

Techniques

- Chaining
- Open addressing
Collision resolution by chaining

Chain elements that hash to the same slot in a linked list

Slot $j$ is a pointer to the head of the list of all elements that hash to $j$

When $\exists$ such elements, slot $j$ hashes to Nil
Dictionary operations in collision resolution by chaining

Chained-Hash-Search \((T, k)\)

search for an element with key \(k\) in list \(T[h(k)]\)

\textit{Worst case: proportional to length of list}

Chained-Hash-Insert \((T, x)\)

insert \(x\) at the head of list \(T[h(key[x])]\)

\textit{Worst case: }\(O(1)\)

Exercise 12.2-2

Chained-Hash-Delete \((T, x)\)

delete \(x\) from list \(T[h(key[x])]\)

\textit{Worst case: }\(O(1)\) if list is doubly linked

Singly-linked list: search for \(x\) predecessor to splice \(x\) out

Delete and search have same running time

To analyze hashing with chaining, examine search
Analysis of hashing with chaining

Given a hash table $T$ with $m$ slots storing $n$ elements

**Load factor** for $T$: average number of elements in a chain

$$\alpha = \frac{n}{m}$$

- $\alpha < 1$: on average, less than one element per slot
- $\alpha = 1$: on average, one element per slot
- $\alpha > 1$: on average, more than one element per slot
Analysis of hashing with chaining

Worst-case:
All $n$ keys hash to same slot, a list of length $n$
Time for searching: $\Theta(n)$, plus time to compute $h$
$\rightarrow$ $h$-table not attractive

Average case:
Depends on how $h$-function distributes set of keys among $m$ slots, on average

1. Division method
2. Multiplication method
3. Universal hashing

For now, assume **simple uniform hashing**
Simple uniform hashing

Assume:

- Compute \( h(k) \) and access slot is in \( O(1) \)
- Searching element of key \( k \) is linear in length of list \( T[h(k)] \)

Question: What is the number of elements considered by search? i.e., number of elements in list \( T[h(k)] \) checked to see if their key is equal to \( k \)

1. Search unsuccessful: no element in table has key \( k \)
2. Search successful: finds element in table with key \( k \)

Result: under assumption of uniform hashing, search is \( \Theta(1 + \alpha) \) on average

Theorems: 12.1 and 12.2
Theorem 12.1 In a hash table in which collisions are resolved by chaining, an unsuccessful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

Proof: Simple uniform hashing $\Rightarrow$ any key $k$ is equally likely to hash to any of the $m$ slots

The average time to search unsuccessfully for a key $k = \text{average time to search to the end of one of the } m \text{ lists}$
Average length of such a list is $\alpha = n/m$
$\Rightarrow$ expected number of elements examined is $\alpha$
$\Rightarrow$ Total time required $= \Theta(\alpha) + \text{time for computing } h(k)$
$\Rightarrow$ Total time required $= \Theta(1 + \alpha)$
Theorem 12.2 In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

Proof
Assumptions:

- Searched key equally likely to be in any of $n$ keys stored
- Chained-hash-Insert inserts new element at end of list
Theorem 12.2 . . . a **successful** search takes time
\( \Theta(1 + \alpha) \), on the average . . .

**Proof** (cont’)

The expected \#elements examined during a
successful search is 1 + number of elements
examined when sought for element was inserted.
Thus, we take the average, over the \( n \) items in
table, of: 1+ the expected length of the list to
which the \( i \) element is added

Expected length of list is \( (i - 1)/m \) \( \Rightarrow \) Expected
number of elements examined is:
\[
\frac{1}{n} \sum_{i=1}^{n} \left( 1 + \frac{i-1}{m} \right)
\]

\[
= 1 + \frac{1}{nm} \sum_{i=1}^{n} (i - 1) \quad (1)
\]

\[
= 1 + \left( \frac{1}{nm} \right) \left( \frac{(n-1)n}{2} \right) \quad (2)
\]

\[
= 1 + \frac{\alpha}{2} - \frac{1}{2m} \quad (3)
\]

Thus total time required for a successful search is
\( \Theta(2 + \frac{\alpha}{2} - \frac{1}{2m}) = \Theta(1 + \alpha) \)
**Interpretation** of Theorem 2.1 and 2.2

If the number of slots, $m$, is at least proportional to number of elements in table, $n$, we have $n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1)$

Thus searching takes constant time on average.

**Remember:**
Insertion is in $O(1)$, Deletion is in $O(1)$ (doubly-linked lists)

All dictionary operations can be supported in $O(1)$
Hash functions

A good hash function

- can be computed quickly
- satisfies (approx.tely) the simple uniform hashing assumption:
  each key is equally likely to hash to any of the $m$ slots

Formally:
Assume each key is drawn independently from $U$ with probability distribution $P$
$P(k)$ is the probability that $k$ is drawn
Simple uniform hashing $\Rightarrow$

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m}, \text{ for } j = 0, 1, \ldots, m - 1$$

However, $P$ is usually unknown
Hash functions

1. Division method
2. Multiplication method
3. Universal hashing

**Assumption:** keys are natural numbers \((\in \mathbb{N})\)

If keys \(\not\in \mathbb{N}\), find a way to express them as such

**Example:** strings can be interpreted by interpreting each character as an integer in notation radix-128, using the ASCII character set

**Illustration:** \(pt\) can be interpreted as \((112.128) + 116 = 14452\), since \(p = 112\) and \(t = 116\) in the ASCII character set
Division method

\[ h(k) = k \mod m \]

Example: \( m = 12 \) and \( k = 100 \Rightarrow h(k) = 4 \)

- Quick hash function
- Avoid \( m \) power of 2
  If \( m = 2^p \), then \( h(k) \) is just the \( p \) lowest order bits of \( k \)
  Unless \( P(k) \) makes all low-order \( p \)-bits patterns equally likely
- Avoid powers of 10 if keys are decimal numbers (as hash function does not depend on all the decimal digits of \( k \))
- Good values for \( m \) are primes not too close to exact powers of 2

Example: \( n = 2000 \) character strings, each character has 8 bits
If \( \alpha = 3 \) then choose \( m = 701 \) (prime not too close to a power of 2)
Multiplication method

Two steps:

1. Multiply $k$ by a constant ($0 < A < 1$), and extract the fractional part of $kA$

2. Then multiply value but $m$ and take floor of result

\[ h(k) = \lfloor m(kA \mod 1) \rfloor \]

where $kA \mod 1 = ka - \lfloor kA \rfloor$

- Value of $m$ is not critical
- We can choose $m$ to make h-function easy to implement
Universal hashing

Worst-case scenario:

- Malicious adversary chooses the keys to be hashed
- Bad choice of hashing, all $n$ keys hash to the same slot

Average retrieval time deteriorates: $\Theta(n)$

Any fixed hash function is vulnerable

Way out? Choose a $h$-function that is random, independent of keys to be stored

The scheme is called universal hashing
  yields good performance on average
  no matter what keys are chosen by adversary
Universal hashing: principal (I)

Select a hash function at random, at run time from a carefully designed class of functions

Randomization guarantees

- that no single input will evoke worst-case behavior
- good average-case performance, no matter what keys are provided as input
**Universal hashing:** principal (II)

Let $\mathcal{H}$ be a finite set of hash functions that map a universe $U$ of keys into the range $\{0, 1, 2, \ldots, m - 1\}$

$\mathcal{H}$ is universal if for every pair of distinct key $x, y, \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$

that is, with a hash function randomly chosen from $\mathcal{H}$, the chance of collision between $x$ and $y$ when $x \neq y$ is exactly $1/m$, which is exactly the chance of a collision if $h(x)$ and $h(y)$ are randomly chosen from the set $\{0, 1, 2, \ldots, m - 1\}$