Hash Tables

Inserting, Finding and Deleting in $O(1)$

Hash Tables

We want to support an efficient “dictionary” – this means supporting the operations of INSERT, SEARCH and DELETE in as little time as possible (i.e., $O(1)$) and, if possible, independently of the size of the set.

Idea: an object is identified by a key, and the key is used to compute a (unique?) integer. We can use this integer (or the key) to index into an array. If the computation from the key to the array position is efficient, we should be done.

Can we do it? The usual answer applies – and it is?

Yes, kind of, but...
Hash Tables

Direct-Address Tables

We make some assumptions:
1. The universe of keys $U$, is fairly small, say $U = \{1, \ldots, m-1\}$.
2. No two elements have the same key.
3. We represent the dynamic set via an array (direct-address table) $T[0, \ldots, m-1]$.
4. Each position in the array corresponds to an element with key $k$.
5. If the set contains no element with key $k$, then $T[k] = NIL$.

The dictionary operations are trivial:

\begin{verbatim}
DIRECT-ADDRESS-SEARCH($T, k$)
1 return $T[k]$

DIRECT-ADDRESS-INSERT($T, x$)
1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE($T, x$)
1 $T[x.key] = NIL$
\end{verbatim}

Each of these operations takes only $O(1)$ time.
What happens if the universe is large — and the set being stored is actually much smaller than the universe?

• $T$ would have to be very large with most of its slots empty.

**Solution:** use an array large enough to hold all the items (keys) and use the keys to compute array indices. The computation should be quick but the map $h: key \rightarrow index$ is not 1-1.

We call this function $h$ a **Hash Function**:

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

A key $k$ hashes to a slot $h(k)$. 
Hash Tables

• How do we deal with collisions?
First of all, the “birthday paradox” we saw earlier in the course tells us that the probability of a collision goes above 50% with with about 6% of the table filled (why?), at least under uniformity and independence assumptions on the keys and their images under $h$.

• So we have to deal with collisions. **Methods:**
  1. Collision resolution by chaining
  2. Collision resolution by Open Addressing (linear probing, quadratic probing, double hashing).
  3. Perfect hash functions (for sets where all keys are known – and you have time to hand-craft a function).
**Hash Tables**

**Collision Resolution by Chaining**

The Operations

1. **CHAINED-HASH-INSERT**(*T*, *x*) \( \mathcal{O}(1) \)
   1. Insert *x* at the head of the list \( T[h(x\text{.key})] \)

2. **CHAINED-HASH-SEARCH**(*T*, *k*) \( \mathcal{O}(?) \)
   1. Search for an element with key *k* in list \( T[h(k)] \)

3. **CHAINED-HASH-DELETE**(*T*, *x*) \( \mathcal{O}(?) \)
   1. Delete *x* from the list \( T[h(x\text{.key})] \)

**Figure 11.3** Collision resolution by chaining. Each hash table slot \( T[j] \) contains a linked list of all the keys whose hash value is *j*. For example, \( h(k_1) = h(k_4) \) and \( h(k_3) = h(k_7) \). The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.
Hash Tables

Analysis of Hashing with Chaining

• **Load factor:** if a hash table $s$ has $m$ slots and $n$ elements stored, we say $\alpha = n/m$ is the table load factor. $\alpha$ can be $>1$...

• **Worst Case** for Chained-Hash-Search: $\Theta(n)$, since we could hash all elements to the same index.

• What is the **average case** performance?

• **Assumptions:** any given element is equally likely to hash into any of the $m$ slots, independently of where any other element has hashed to (**simple uniform hashing**).

• **Notation:** for $j = 0, 1, \ldots, m-1$, let $n_j$ denote the length of the list $T[j]$. $n = n_0 + n_1 + \ldots + n_{m-1}$.

• **The expected value of $n_j$:** $E[n_j] = \alpha = n/m$.

• More **Assumptions:** computing $h(k)$ take time $O(1)$.

Theorem 11.1. In a hash table that resolves collisions by chaining, and under the assumption of simple uniform hashing, an unsuccessful search takes average-case time $\Theta(1+\alpha)$.

**Proof.** Any key not in the table is equally likely to hash to any slot. The expected time to search unsuccessfully for a key $k$ is the expected time to search to the end of $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$. The expected number of elements examined is $\alpha$, to which we add the time to compute $h(k)$. Total: $\Theta(1+\alpha)$.
Hash Tables

Analysis of Hashing with Chaining

**Theorem 11.2.** Under simple uniform hashing, and in a hash table where collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$.

**Proof.**

• Assume the element being searched for is equally likely to be any of the $n$ in the table.
• The number of elements examined during a successful search for $x$ is one more than than the number of elements that appear before $x$ in $x$’s list (newly inserted elements are always at the front of the list).

• The expected number of element examined is given by the average, taken over the $n$ elements in the table, of 1 plus the expected number of elements added to $x$’s list after adding $x$.
• Let $x_i$ be the $i_{th}$ element (of $n$) inserted into the table, and let $k_j = x_j.key$.
• Define the indicator random variable $X_{ij} = I\{h(k_i) = h(k_j)\}$.
• Simple uniform hashing implies that $\Pr\{h(k_i) = h(k_j)\} = 1/m$ and, consequently, that $E[X_{ij}] = 1/m$.
• We now compute the expected number of elements examined in a successful search:
Hash Tables

Analysis of Hashing with Chaining

\[ E\left[ \frac{1}{n} \sum_{j=1}^{n} \left( 1 + \frac{1}{j} \right) \right] = \frac{1}{n} \sum_{j=1}^{n} \left( 1 + \frac{1}{j} \right) = 1 + \frac{1}{nm} \sum_{i=1}^{n} n - i \]

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

\[ = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}. \]

And the time required for a successful search is \( \Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha). \)

In other words: if the number of slots is proportional to the number of elements \( (n = O(m)) \), so that \( \alpha = n/m = O(m)/m = O(1) \).

The search, on average, takes constant time.
Hash Tables

Choosing a Hash Function
Can we satisfy the assumption of simple uniform hashing?
In general, not quite, but we have to try...
Keys can be anything that can be transformed into an index, and they usually have patterns – our job is to find a way to use the patterns to compute indices that satisfy, to the best of our ability, the simple uniform hashing property.

Interpreting keys as natural numbers: if keys are strings of characters, we can take the ASCII code values as digits for an integer in base 256 – converting to decimal or binary as needed.

Hash Tables

Choosing a Hash Function
The Division Method. Assuming that we have an easy (and generally 1-to-1) way of moving from key to integer, and with a table with \( m \) slots, where \( m \) is much smaller than the integers associated with the keys, we define
\[
h(k) = k \mod m.
\]

How do we choose \( m \)? This is crucial and often not obvious.
Avoid: if \( m = 2^p \), for some \( p > 0 \), \( k \mod h(k) \) is just the last \( p \) bits of the key – so avoid powers of 2, unless you know the low order bit are enough.
Avoid: if \( m = 2^{p-1} \), for some \( p > 0 \), and \( k \) is a character string interpreted in radix \( 2^p \) – permuting the characters of \( k \) does not change its hash value (Ex. 11.3-3).
Hash Tables

Choosing a Hash Function

Use: it turns out that choices of $m$ as a prime not close to a power of 2 seems to work well. Ex.: $n = 2000$, $m = 701$ would give expected lengths of 3 for chains.

The Multiplication Method. Pick $0 < A < 1$, multiply by $k$ and extract the fractional part of $kA$ (i.e. $kA \mod 1 = kA - \text{floor}(kA)$), multiply the result by $m$ and take the floor: $h(k) = \text{floor}(m \,(k \, A \mod 1))$.

In this case we can choose $m = 2^p$, for some $p > 0$.

Use: experiment – a good choice appears to be $A = (\sqrt{5} - 1)/2$.

Look at the data for a guess.

---

**Figure 11.4** The multiplication method of hashing. The $w$-bit representation of the key $k$ is multiplied by the $w$-bit value $s = A \cdot 2^w$. The $p$ highest-order bits of the lower $w$-bit half of the product form the desired hash value $h(k)$.
Hash Tables

Open Addressing

Instead of chaining, the items occupy actual slots in the table.

The first pass is the same: compute the index from the key and, if the position is not yet occupied, copy the item into the table.

The first problem arises when we have a collision: how do we resolve collisions, since there are no chains?

Answer: we have a repeatable sequence of “probes”, starting at the collision point, and providing a permutation of the set \{0, \ldots, m\} so that we can find an empty slot if one exists.

<table>
<thead>
<tr>
<th>HASH-INSERT(T, k)</th>
<th>HASH-SEARCH(T, k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( i = 0 )</td>
<td>1 ( i = 0 )</td>
</tr>
<tr>
<td>2 repeat</td>
<td>2 repeat</td>
</tr>
<tr>
<td>3 ( j = h(k, i) )</td>
<td>3 ( j = h(k, i) )</td>
</tr>
<tr>
<td>4 if ( T[j] == ) NIL</td>
<td>4 if ( T[j] == k )</td>
</tr>
<tr>
<td>5 ( T[j] = k )</td>
<td>5 return ( j )</td>
</tr>
<tr>
<td>6 return ( j )</td>
<td>6 ( i = i + 1 )</td>
</tr>
<tr>
<td>7 else ( i = i + 1 )</td>
<td>7 until ( T[j] == ) NIL or ( i == m )</td>
</tr>
<tr>
<td>8 until ( i == m )</td>
<td>8 return NIL</td>
</tr>
<tr>
<td>9 error “hash table overflow”</td>
<td></td>
</tr>
</tbody>
</table>
Hash Tables

Open Addressing

Problem: how do we delete?

• With chaining, the answer was easy: just remove the item from the chain.

• With open addressing, removal results in the question what kind of information is being stored in the array position? We can’t have NIL, since that would stop a search along a probe sequence even when we want to continue.

• Solution: use a new marker - DELETED: this allows for the position to be re-used in an insertion, and to be skipped (without stopping) in a search. Note that search times and load factors can become decoupled if the table changes substantially over time. Note further that there may be no NIL (stopping criterion?).

---

Hash Tables

Open Addressing

Linear Probing. We are given an auxiliary hash function \( h' : U \rightarrow \{0, \ldots, m-1\} \). We construct a hash function \( h(k, i) = (h'(k) + i) \mod m \).

We first probe \( T[h'(k)] \); if needed, we probe \( T[h'(k)+1] \); and so on to \( T[h'(k)-1] \). This provides us with \( m \) probe sequences.

Problem: primary clustering. Long runs of occupied slots build up, increasing the average search time.

Why? An empty slot preceded by \( i \) full slots gets filled next with probability \((i + 1)/m\), so long filled runs are extended with higher probability than short ones.
Hash Tables

Open Addressing

**Quadratic Probing.** To try to avoid the problems of primary clustering, we introduce a hash function

\[ h(k, i) = (h'(k) + c_1i + c_2i^2) \mod m, \]

where \( c_1 \) and \( c_2 \) are positive constants.

This spreads successive probes farther and farther. Since the sequence is decided by the first probe (we still have just \( m \) sequences), we still have some clustering: **secondary clustering**.

---

**Double Hashing.** We move one step further in our attempted approximation of uniform hashing.

Assume we have two auxiliary hash functions, \( h_1 \) and \( h_2 \). We construct

\[ h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m. \]

This means that the probe sequence depends in two ways on the key: we should be able to guarantee \( \Theta(m^2) \) probe sequences.

**Careful:** in order for each sequence to result in a permutation of \( \{0, \ldots, m-1\} \) we need all values \( h_2(k) \) to be relatively prime to the size \( m \) of the table. This can be obtained in several ways: a) choose \( m \) a power of 2 and design \( h_2 \) so it always returns an odd number. b) choose \( m \) prime and design \( h_2 \) to return a positive integer less than \( m \).
Hash Tables

Open Addressing

Analysis:
• **Assume** the load factor $\alpha = n/m \leq 1$. This is obvious, because we don’t have chains.

• **Assume uniform hashing**: each probe sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ (of the possible $m!$ ones) is equally likely.

**Theorem 11.6.** Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1 – \alpha)$.

Proof. In an unsuccessful search, every probe but the last accesses an occupied slot with the wrong key – the last has NIL.

• Define the random variable $X$ to denote the number of probes made in an unsuccessful search.

• Define the event $A_i, i = 1, 2, \ldots$, to be the event that an $i^{th}$ probe occurs and it is to an occupied slot.

• The event $\{X \geq i\}$ is the intersection $A_1 \cap A_2 \cap \ldots \cap A_i$, so $\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \ldots \cap A_i\}$. We now bound...

$\Pr\{A_1 \cap A_2 \cap \ldots \cap A_i\} = \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \cdot \ldots \cdot \Pr\{A_{i-1} | A_1 \cap A_2 \cap \ldots \cap A_{i-2}\}$
Hash Tables

Open Addressing

- Pr\{A_j\} = n/m, since this is the probability that the first probe is to an occupied slot.
- The probability that there is a \( j \)th probe and it is to an occupied slot, given that the first \( j-1 \) probes were to occupied slots, is \( (n-j+1)/(m-j+1) \) – we have \( n-j+1 \) elements unaccounted for and \( m-j+1 \) slots unaccounted for, and uniform hashing give the probability as the quotient. Putting this together:
  - Pr\{X \geq i\} = (n/m)((n-1)/(m-1))\cdots((n-i+2)/(m-i+2)) \leq (n/m)^{i-1} = \alpha^{i-1}.
  - \( E[X] = \sum_{i=0}^{\alpha} k \Pr\{X \geq i\} \leq \sum_{i=0}^{\alpha} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}. \)

Corollary 11.7. Inserting an element into an open-address hash table with load factor \( \alpha \) requires at most \( \frac{1}{1-\alpha} \) probes on average, assuming uniform hashing.

Proof. An element is inserted only if \( \alpha < 1 \). Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. The theorem gives that the expected number of probes is \( \frac{1}{1-\alpha} \).

The case of a successful search follows.
Hash Tables

**Open Addressing**

**Theorem 11.8.** Given an open-address hash table, $\alpha < 1$, the expected number of probes in a successful search is $(1/\alpha) \ln (1/(1 - \alpha))$.

**Proof.** A search for a key $k$ reproduces the probe sequence of the insertion of $k$. If $k$ was the $(i+1)^{th}$ key inserted, the expected number of probes made in a search for $k$ is at most $1/(1 - i/m) = m/(m - i)$.

Averaging over all $n$ keys in the table:

$$
\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m - i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m - i} = \frac{1}{\alpha} \sum_{j=m-n+1}^{m} \frac{1}{x} \leq \frac{1}{\alpha} \int_{m-n}^{m} \frac{1}{x} \, dx = \frac{1}{\alpha} \ln \frac{m}{m-n}
$$

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