Hashing (just the basics)

Thanks to
Prof. Charles E. Leiserson

Symbol-table problem

Symbol table $T$ holding $n$ records:

How should the data structure $T$ be organized?

Hash tables and hash functions

We always have a table (cubby). Each cell has an index. The index is a number between $0..m-1$.

A hash function $h$ computes for every possible key an index in a hash table $\{0, 1, \ldots, m-1\}$:

When a record to be inserted maps to an already occupied slot in $T$, a collision occurs.

Resolving collisions by chaining

Records in the same slot are linked into a list.
Analysis of chaining

Let $n$ be the number of keys in the table, and let $m$ be the number of slots.

Define the load factor of $T$ to be

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

= average number of keys per slot.

We will try to keep this value no larger than 1 (same number of keys and slots)

Search cost

Expected time to search for a record with a given key $= \Theta(1 + \alpha)$.

apply hash function and access slot

Expected search time $= \Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.

What to do if table too dense

Once $\alpha$ is too large

It does not effect corrections, but effects performances.

Once we have a chance, re-double the table (and compute a new hash function)

Example (credit GeeksforGeeks)

Start with a table of a small size
(Fig does not show the pointers to the linked list
When table too dense, double its size
sizes: 2, 4, 8, …

Total time: if it takes $O(1)$ to re-insert a key, the total time for inserting $n$ keys is $n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \leq 2n$

Choosing a hash function

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.
- Hope: if $k_1 \neq k_2$ in any bit, then there is a good chance $h(k_1) \neq h(k_2)$
- Functions that ignore some bits (e.g. $h(k) = k \mod 100$, $h(k) = k \mod 1024$) should be used only if know enough about the data distribution to think that this is not an issue.
**Division method**

Assume all keys are integers, and define 
\[ h(k) = k \mod m. \]

**Deficiency:** Don’t pick an \( m \) that has a small divisor \( d \). A preponderance of keys that are congruent modulo \( d \) can adversely affect uniformity.

**Extreme deficiency:** If \( m = 2^r \), then the hash doesn’t even depend on all the bits of \( k \):
- If \( k = 1011000111011010_2 \) and \( r = 6 \), then \( h(k) = 011010_2 \).

**Deficiency:** Don’t pick an \( m \) that has a small divisor \( d \). A preponderance of keys that are congruent modulo \( d \) can adversely affect uniformity.

**Annoyance:**
- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we’ll see is usually superior.

---

**Multiplication method**

Assume that all keys are integers. Pick a constant integer \( A \), and set 
\[ h(k) = (A \cdot k) \mod m \]

\( A \) is an odd integer

Other variant of the multiplication method:

Pick \( A \) as a non-integer number

\( A = 2.71828182846 \) or \( A = \sqrt{2} = 1.41421356237 \)

\[ h(k) = \left\lfloor m \left( (A \cdot k) - \lfloor A \cdot k \rfloor \right) \right\rfloor \]

Note - the part in the red parenthesis is a float in (0,1).

Multiply my \( m \) gives a float in (0,m). The second floor just makes it a legit index in the table \( T[0...m-1] \).

---

**Division method (continued)**

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

Pick \( m \) to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

**Annoyance:**
- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we’ll see is usually superior.

---

**Multiplication method example**

Variant 3: \( A \) is a large integer, but the value of \( h(k) \) is the number that several digits in the ‘middle’ of the \((A \cdot k)\).

\[
\begin{array}{c}
1 0 1 1 0 0 1 = A \\
\times 1 1 0 1 0 1 1 = k \\
\hline
1 0 0 1 0 1 0 0 1 1 0 0 1 1 \\
\hline
\end{array}
\]

Note - the part in the red parenthesis is a float in (0,1).
Multiple hash functions:

Applications to distributed database. We need to store a large number of records.

Let's think about a system with 8 disks. Or 8 users, each in a different locations.

- RAID of disks is a device that contains multiple disks (sometimes sharing parts)
- Each individual disk is prone to failures. Want to maintain robustness and load fairness.
  - Robustness to disk failures: We should still be able to access all our data, even if two disks crashed. So each record needs to be stored on multiple disks.
  - Load fairness: Each disk should store a small portion of the database, and these portions should be split fairly. So each disk should contain approximately \( \frac{n}{N} \) records.
- Efficiency: Search time should be small. When searching for a key, should not have to check each individual disk
  - Need to support: \( \text{Insert}(k)\), \( \text{delete}(k)\), \( \text{find}(k)\)
  - We don’t know the data in advance - changing dynamically.
  - So once a new record appears, we need to decide which 3 disks will store it. Once a query \( \text{find}(k) \) appears, need to be able to find these 3 disks.

Another way to think about dot product

- Obviously, our aim is to minimize collisions
- From now on, assume \( m \) (the table size) is a prime number.
- Assume all our keys \( K = \{k_1, k_2, \ldots, k_n\} \) are numbers, between \( 0..m-1 \). No key appears twice.
- Pick any integer \( \alpha \in \{1..m-1\} \). Let's consider the hash function \( h(x) = (\alpha x) \mod m \).

**Lemma 1**: For every \( Y \in \{0..m-1\} \), there is a unique \( t \in \{0..m-1\} \) such that \( \alpha t = Y \mod m \).

- Good news: The lemma guarantees that the hash function \( h(x) = (\alpha x) \mod m \) will map the keys of \( K \) to different cells of the hash table. No collisions at all.
- Bad news: This guarantee is waved if we don’t require that all keys of \( K \) are \( \leq m \). For example, let's play with \( h(x) = (3x) \mod 5 \). Then \( h(3) = h(8) \). So by itself, this is not very helpful. We will see next how to use it more efficiently.
- Before continuing, let's rewrite Lemma 1:

**Lemma 2**: For every fixed \( Y \in \{0..m-1\} \), and every fixed \( \alpha \in \{1..m-1\} \), there is exactly one value \( x \in \{0..m-1\} \) such that \( \alpha x = Y \mod m \).
Universal family of hash functions

We have a set of hash functions \( H = \{h_1(k) \ldots h_L(k)\} \)

We say that it is universal iff:

- for which, for every two keys \( k_i, k_j \), if we pick at random \( h(k) \in H \) the probability that it collides \( h(k_i) = h(k_j) \) is only 1/m.

That is, only \( L/m \) of the functions of \( H \) cause collisions.

If we think about all the possible hash functions

\[ h((x, y)) = ((ax + \beta y) \mod m) \]

When we change \( \alpha, \beta \), we create different members of the family.

We just saw that this facility is universal. It guaranties that the probability of collision is \( \leq \frac{1}{2m} \).

Dot-product method. Hashing large files.

In many applications, the key is too long to be considered a single number.
E.g. \( k = \text{“BDCZ”} \).

In general, we need a hash functions that could be used on very long keys, as text documents, books, images, DNA, geometric structures, malware, viruses...

Expressing the key as a single number is not useful.

**Idea:** Remember that if the key \( k \) is a small number, we could use the multiplication method, and set \( h(k) = \langle k \rangle \mod m \)

Now if the key is very large, let us break it into several small pieces, so instead of treat the key as a single number, let us think about it as a vector (or a list) consisting of several numbers.

Instead of a single constant \( A \), we compute multiple and different constants \( a_1, a_2, a_3, a_4 \ldots \)

We decompose the key into characters, multiply each by a different constant and sum (modulo \( m \)). Example:

\[ h(\text{BDCZ}) = (a_1 \cdot 66 + a_2 \cdot 68 + a_3 \cdot 67 + a_4 \cdot 90) \mod m \]

\( m \) is the size of the hash table. The ascii value of ‘B’ is 66 and of ‘Z’ is 90.

- Computing all constants \( a_i \) is very simple. Pick random integers between 1 and \( m-1 \).
- Excellent in practice, and theory
- Involve one pass of the file, in the case of very long keys.
Dot product method-cont.

Before any data item arrives, decide about the size $m$ of the hash table. $m$ should be prime, and $>2n$.

Let $m \approx 2^{20} = 1M$

Pick at random constants $\mathbf{a} = (a_0, a_2, \ldots, a_r)$. Each $a_i$ is picked individually at random uniformly $1 < a_i < m - 1$

Now the first key $k$ arrive. Let's break it into pairs of characters, and for each pair, compute its numeric value using base 256 (ASCII).

$k = \text{Ac}|\text{co}|\text{rd}|\text{in}|\text{g}|\text{t}_0 \cdot \text{Ac}|\text{co}|\text{rd}|\text{i}_1 \cdot \text{Ac}|\text{co}|\text{rd}|\text{t}_2 \cdots$ where $k_0 = 'A' \cdot 256 + 'c' = 65 \cdot 256 + 99$.

Finally $h_{\mathbf{a}}(k) = \left( \sum_{i=0}^{r} a_i \cdot k_i \right) \mod m$

Resolving collisions by open addressing

No storage is used outside of the hash table itself.

Each cell could contain at most one key.

The same key $k$ might be mapped by $h(k)$ to different locations in the table, depending on availability.

When either searching $k$ or searching for a place for $k$, we will check

- The first index that we search $k$. If fail
- The second index that we search $k$. If fail
- The third index that we search $k$. If fail etc

When should we give up? (will see in next slides)

How should we find these indexes?

$h(k, i)$ - a hash function that takes two parameters:

- Key $k$
- Trial number $i$ (first trial has index 0)

Example of Insertion

Hash function: $h(k, i) = (k + i) \mod 8$

$k$-key: $i$ is the attempt number (start at 0)

- insert(12). $h(12,0) = 4$
  
  Read: The first attempt ($i=0$) checks $T[h(12,0)]$. It is free

- insert(15). $h(15,0) = 7$

- insert(20). $h(20,0) = 4$ (collision)
  
  $h(20,1) = (20 + 1) \mod 8 = 5$
  

- insert(23). $h(23,0) = 7$ (collision)
  
  $h(23,1) = 0$

- insert(28). $h(28,0) = 4$ (collision)
  
  $h(28,1) = 5$ (collision); $h(28,2) = 6$
Searching a key. Example on the same table

Hash function: \( h(k, i) = (k + i) \mod 8 \)

Finding a key \( k \):
we check if \( T[h(k,0)] \neq k \). If not, if empty, stop. otherwise we check if \( T[h(k,1)] \neq k \). If not, if empty, stop... other etc.

Next, delete 20.

Now, lets search 28 again.
The search wrongly stops at the empty cell that used to contain 28. Error

Solution: Place a dummy to indicate that this cell used to contain a key, but this key was deleted. The 'search' treats this cell as 'nonempty' and continues the probing sequence.
The search stops only when reaching a cell that is "really" empty.

When inserting a new key, we can replace the dummy with a real key. Example - inserting 13 will override the dummy.

Maintenace
Scan the table from time to time, and get rid of all of all dummies.
Re-insert each key,
If the table needs to be expanded - good opportunity to use the dynamic table technique and re-hash.

Probing strategies

Linear probing:
Given an ordinary hash function \( h'(k) \), linear probing uses the hash function
\[
 h(k, i) = (h'(k) + i) \mod m.
\]

This method, though simple, suffers from **primary clustering**, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.
Probing strategies

Double hashing
Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

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

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to $m$. One way is to make $m$ a power of 2 and design $h_2(k)$ to produce only odd numbers.

Analysis of open addressing

**Theorem.** If the data is distributed well enough (detailed dropped), the expected number of probs for insert/delete/find is $1/(1-\alpha)$.

$$\alpha = \frac{\text{number of keys}}{\text{number of cells}}$$

Example: $\alpha = 0.99$. Need 100 probs on average.
Example: $\alpha = 0.5$. Need 2 probs on average.

Conclusion: Keep $m \geq 2n$. Use dynamic arrays if needed.