Hash Tables

(Version of 15th March 2010)

(Partially based on original slides by John Hamer)

The m-slot array has an unbeatable property: It takes $\Theta(1)$ time to find, insert, or delete an element, given its index (or key). (Finding an element of given satellite data can be done in at worst $O(m)$ time in an unsorted array and in at worst $O(\lg m)$ time in a sorted array.)

Alas, only small ranges of integers can be used as array indices.

Would it not also be great to index an array with strings, say?

$$\mathit{OfficePierre} \leftarrow \mathit{office}[\text{“}\mathit{Pierre}\text{”}]$$
$$\mathit{office}[\text{“}\mathit{Pierre}\text{”}] \leftarrow 1336$$

Hashing is a method that allows us to do just that, using just about any kind of data (strings, reals, records, . . .) as the index (or key).

But Strings Are Numbers!

They may not look like it, but strings are numbers, too.
Each character in a string can take one of 256 different values.
For example, the character `J` has the value 74 in the ASCII code,
while `o` is 111, `h` is 104, and `n` is 110.

We can think of the string `John` as a number in base 256. Just as the number 123 in base 10 represents $1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$, the string `John` can represent $74 \cdot 256^3 + 111 \cdot 256^2 + 104 \cdot 256^1 + 110 \cdot 256^0$, which we write as $1, 248, 815, 214$ in base 10.

But What Is the Point of That?

If we had an array with at least $1,248,815,214$ slots, we could index into it with the string `John`!

But that would be a terrible waste of space, since most of the $1,248,815,214$ or more slots in the array would be empty.
If we want to store elements for n keys, then we would like to use a data structure of size $\Theta(n)$.

Hashing is a technique for storing elements for a subset K of a universe U of keys in $\Theta(|K|)$ space, rather than in $\Theta(|U|)$ space, while still retaining the $\Theta(1)$ search, insertion, and deletion times.

Computing an Index

Assume we want to store $n = 1000$ elements in a hash table (which is just an array) with $m = 1000$ slots. We cannot use the keys (seen as numbers) to index into the array, but we could try computing a suitable index from a key and use that index instead. In other words, we want a function

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

that returns a suitable index for each key of type U.

Given such a function, we could then write:

$$OfficePierre \leftarrow office[hash(\text{“}Pierre\text{”})]$$
$$office[hash(\text{“}Pierre\text{”})] \leftarrow 1336$$

This is called perfect hashing.

Problems

The function $hash$ has a tough job. It can be passed any key, and must try to find a different index for each key. Further, $hash$ has to run in $\Theta(1)$ time.

Theorem: Such a perfect $hash$ function cannot be written.

"Proof:" Assume $hash$ has filled up 999 of the $m = 1000$ slots in the array. It must return the one remaining index for the next key it is given. But that key could be anything. If $hash$ succeeds, then try again with a different key. One way or the other, $hash$ loses.

Such a perfect $hash$ function can only succeed if it is written with perfect advance knowledge of the keys that are to be passed in.
That is asking too much.

A Statistical Solution

Forget about perfect hashing. We want a solution that works in $\Theta(1)$ time, but that gives us some room to move.

Assumption (until page 15 inclusive) of simple uniform hashing: All array slots are equally likely to be hashed to. If the n keys we pass to $hash$ are 'random' (that is, when considered as numbers, they come from a uniformly distributed random variable), then consider

$$hash(key) = intRepr(key) \bmod m$$

where $intRepr(key)$ is the integer representation of key, and m is the number of array slots (usually a prime not too close to some 2^i).

But it looks like we have replaced one problem with another:
There is no longer any guarantee that $hash$ will return a unique index for each key, so there can be collisions!

Approach 1: Collision Resolution by Chaining

We can solve this problem by making each array slot hold a list of values, called a chain. A slot is then also called a bin. Then:

- The search operation can search the list.
- The insert operation can insert the new value into the list.
- The delete operation can delete the value from the list.

This works just fine, provided the lists do not get too long.
They do not. Here follows the statistical analysis.

The probability of having a chain of length k after $n = m$ elements have been added to a hash table of m slots is:

$$\left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{m-k} \binom{m}{k} \approx \frac{1}{ek!}$$

Example: Chain Lengths

For $n = 1000 = m$ elements and bins, we have:

k	probability of k collisions
0	37.0%
1	37.0%
2	18.0%
3	6.0%
4	1.5%
5	0.3%

If we place $n = 1000$ random elements into $m = 1000$ bins, about 37% of the bins will be empty, 37% will have exactly one element, 18% will contain exactly two elements, and it would be unlucky if there was a single bin with 6 or more elements.

Load Factor

About 37% empty bins and a 'maximum' chain length of 5 suggests that we can get away with rather fewer bins if we wish.
Or, to put it another way, there is no reason to stop adding elements once we reach $n = 1000 = m$:

The average chain length will grow in proportion,
but the maximum chain length will grow much more slowly.

Definition: When a hash table contains m bins and n elements, then its load factor is

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

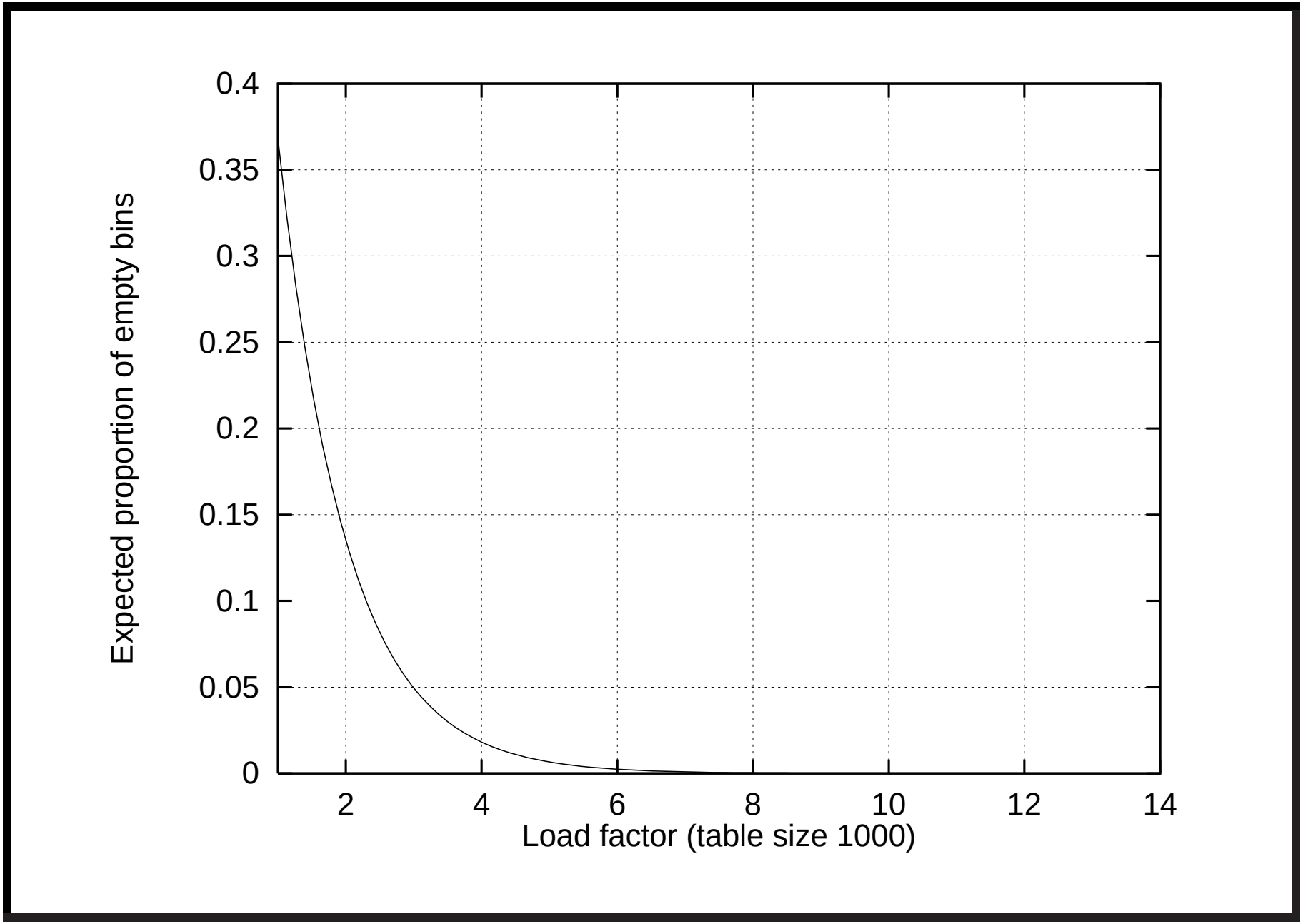
Example: On page 8, it was the case that $\alpha = \frac{1000}{1000} = 1$.

Behaviour as Load Factor Increases

The first things to go as the load factor increases are some of the empty bins. The probability that any particular bin will be empty ($k = 0$) after n elements have been added to a table of m bins is

$$\begin{aligned} \left(1 - \frac{1}{n}\right)^m &= \left(1 - \frac{\alpha}{m}\right)^m \\ &\approx e^{-\alpha} \end{aligned}$$

When $\alpha = 1$, about 37% of the bins are empty.
When $\alpha = 2$ (a normal load factor), about 13% of the bins are empty.
When $\alpha = 5$ (still not high), about 0.7% of the bins are empty.


Expected proportion of empty bins
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
2
4
6
8
10
12
14
Load factor (table size 1000)

Chain Lengths under Load

The probability of having a chain of length k after n elements have been added to a hash table of m slots is

$$\left(\frac{1}{m}\right)^k \left(1-\frac{1}{m}\right)^{n-k} \binom{n}{k} \approx \frac{\alpha^k}{e^\alpha k!}$$

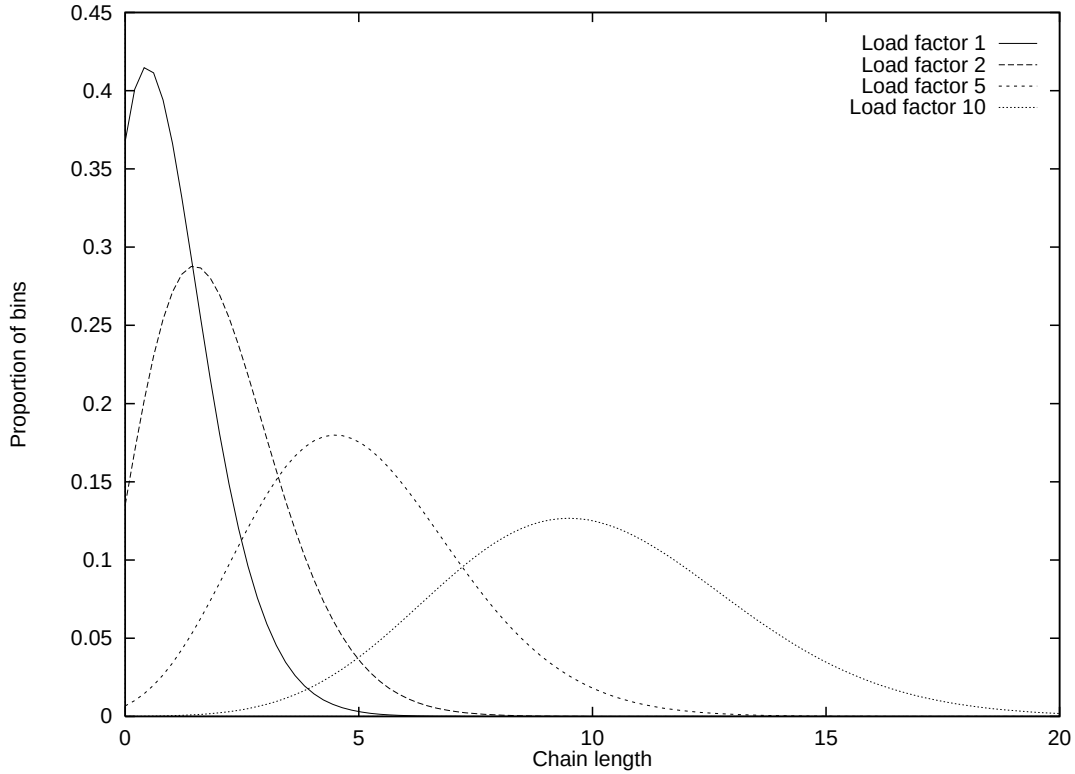
Compare with the $\frac{1}{ek!}$ approximation for $\alpha = 1$ on page 7.

Compare also with the $e^{-\alpha}$ approximation for $k = 0$ on page 10.

Example: Chain Lengths under Load

For n elements in $m = 1000$ bins under load factor α, we have:

k	probability of k collisions	
	$\alpha = 2$ & $n = 2000$	$\alpha = 1$ & $n = 1000$
0	13.0%	37.0%
1	27.0%	37.0%
2	27.0%	18.0%
3	18.0%	6.0%
4	9.0%	1.5%
5	3.0%	0.3%
6	1.0%	
7	0.3%	



This graph shows how the expected distribution of chain lengths changes with the load factor. Observe that (as announced on page 9) while the average chain length is equal to the load factor α, the tail of the distribution moves much more slowly, growing only four-fold from around 5 to around 20 as the load factor increases ten-fold.

Expected Probe Length

Upon simple uniform hashing, for a given load factor, α, the expected number of nodes examined in a search is $\Theta(1+\alpha)$, whether it is successful or not. If $n = O(m)$, then we have $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$, hence search, insertion, and deletion take $O(1)$ time on average!

Use this to predict the behaviour and guide the design of hash tables.

Example: If we expect about $n = 2000$ keys and allocate a hash table with $m = 400$ bins, then the load factor is $\alpha = 5$. The expected performance is around 6 probes (for successful or unsuccessful search) and around 3 empty hash-table bins (less than 1%).

If many more (or fewer) elements are actually stored and the load factor increases beyond (or falls below) a certain threshold, then we can rehash (move the data into a new table of more convenient size).

What If the Keys Are Not Random?

Keys are usually not randomly distributed, and the method described so far collapses then.

Example: People's names are sequences of letters taken from an alphabet. Under the Swedish alphabet, that yields only 29 possibilities for each letter, not 256. For names of 10 letters, only 29^{10} of the 256^{10} possible character strings can actually appear. That is about 0.0000003%. No distribution that is restricted to such a tiny range can possibly be considered random.

With a skewed distribution, chain lengths shoot right through the roof.
Does this mean hashing is a lost cause?
Can we create a random distribution from a highly non-random source?

Mathematics to the Rescue

Consider a deck of cards. Being finite, a deck holds only a finite amount of information. This information is the sequence of cards within the deck. There are 52 cards and therefore $52! \approx 8 \cdot 10^{67} \approx 2^{226}$ different sequences that the deck can have.

Shuffling a deck adds 'noise' to it, or randomises it. A perfect cut-and-shuffle creates one of 2^{52} different equally likely sequences; this is much less information than the deck can hold. So, after a single cut-and-shuffle, much of the original input sequence remains: a perfect cut-and-shuffle injects at most 52 'bits' worth of 'noise' into the deck. But after about five perfect cuts-and-shuffles, more noise has been added to the deck than the deck can hold, and none of the original sequence can be said to remain.

What Has That Got To Do With Hashing?

In a hash function, the output value has a finite capacity to hold information, just like a deck of cards. So, one can ‘shuffle’ the hash input sufficiently to saturate the information capacity of the output.

How to do this is another question . . .

The Magic of Exclusive-Or

The exclusive-or (*xor*) operation is the difference between two bits:

a	b	a xor b
0	0	0
0	1	1
1	0	1
1	1	0

We talk of bits since any data can be expressed as a sequence of bits.

Property: When *xor*-ing a random bit stream and a non-random bit stream, the result is a random bit stream! (A similar phenomenon is observed when a radio programme is superimposed with white noise of equal intensity: the result is white noise.)

Aside: Using Exclusive-Or for Encryption

The *xor* operation can also be used to make an unbreakable method of encryption. Given a message, m, to encrypt, we choose a random key, k, of the same length and combine them using *xor*. The result, $m \text{ xor } k$, will be left with no vestige of the original message. However, the original message, m, can be retrieved from the encryption, $m \text{ xor } k$, by *xor*-ing it with the same key, k:

a	b	$a \text{ xor } b$	$(a \text{ xor } b) \text{ xor } b$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Approach 2: Collision Resolution by Open Addressing

Store all the n elements in the m-slot hash table itself, thereby saving the extra memory necessary for the chaining. Hence we always have that $n \leq m$ and that the load factor $\alpha = \frac{n}{m}$ satisfies $\alpha \leq 1$.

Instead of walking through a list, we compute a sequence of slots to be probed for a given key. We extend $hash : U \rightarrow \{0, 1, \ldots, m-1\}$ to

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

where the number i in $hash(key, i)$ is called the probe number, and

$$\langle hash(key, 0), hash(key, 1), \ldots, hash(key, m-1) \rangle$$

is called the probe sequence, which must be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, so that every slot in the hash table is eventually considered.

Probing Functions

Given an ordinary hash function $hash' : U \to \{0, 1, \ldots, m-1\}$, define:

$$hash(key, i) = \big(hash'(key) + f(i)\big) \bmod m$$

where i runs from 0 to $m-1$, and f is called the probing function:

- Linear probing: $f(i) = i$. Hence the probe sequence is $hash'(key), \ldots, (m-1), 0, 1, \ldots, hash'(key) - 1$.
  Problem: primary clustering: long runs of occupied slots.
- Quadratic probing: $f(i) = c_1 \cdot i + c_2 \cdot i^2$, where $c_2 \neq 0$.
  Much better (for instance, for m prime, $c_1 = 0$, $c_2 = 1$, $\alpha \leq 0.5$), but secondary clustering when $hash'(key_1) = hash'(key_2)$.
- Double hashing: $f(i) = i \cdot hash''(i)$, where $hash''$ is another ordinary hash function. Even better (for well-chosen m & $hash''$).

Basic Operations

Initialise every slot of the hash table to some value $\perp$ denoting that the slot was never used. Then, assuming that duplicates are allowed:

- **Insert** v: Probe until a $\perp$ (or Δ, see below) slot is found, replace it by v, and raise an overflow exception if $i = m$, or rehash if $i = m$ or if α exceeds some threshold (usually 0.50).
- **Search** v: Probe until v is found, and fail if a $\perp$ slot is found or if $i = m$.
- **Delete** v: Probe until v is found, replace it not by $\perp$ but by Δ, denoting that the slot value was deleted, and fail if a $\perp$ slot is found or if $i = m$. Better: Use collision resolution by chaining.

Example of rehashing: if $\alpha > 0.50$, then move the data into a new hash table whose size is the smallest prime number that is larger than $2 \cdot m$.

Example 1: Inserting with Linear Probing

Insert into a hash table t of $m = 10$ slots the following $n = 10$ integers

$$100, 121, 144, 169, 196, 225, 256, 289, 324, 361$$

using $hash'(key) = key \bmod 10$ and $f(i) = i$.

After the first six insertions, t is:

0	1	2	3	4	5	6	7	8	9
100	121	$\perp$	$\perp$	144	225	196	$\perp$	$\perp$	169

Next, 256 collides with 196 for $i = 0$, but $t[7] = \perp$ for $i = 1$, so:

0	1	2	3	4	5	6	7	8	9
100	121	$\perp$	$\perp$	144	225	196	256	$\perp$	169

Next, 289 collides with 169 for $i = 0$, with 100 for $i = 1$, and with 121 for $i = 2$, but $t[2] = \perp$ for $i = 3$, so:

0	1	2	3	4	5	6	7	8	9
100	121	289	$\perp$	144	225	196	256	$\perp$	169

Next, 324 collides with 144 for $i = 0$, with 225 for $i = 1$, with 196 for $i = 2$, and with 256 for $i = 3$, but $t[8] = \perp$ for $i = 4$, so:

0	1	2	3	4	5	6	7	8	9
100	121	289	$\perp$	144	225	196	256	324	169

Finally, 361 collides with 121 for $i = 0$, and with 289 for $i = 1$, but $t[3] = \perp$ for $i = 2$, so:

0	1	2	3	4	5	6	7	8	9
100	121	289	361	144	225	196	256	324	169

Example 2: Deleting with Linear Probing

Delete 121 from t using $hash'(key) = key \bmod 10$:

0	1	2	3	4	5	6	7	8	9
100	121	289	361	144	225	196	256	324	169

If we set $t[1] = \perp$, then search for 361 would be unsuccessful! Hence:

0	1	2	3	4	5	6	7	8	9
100	Δ	289	361	144	225	196	256	324	169

Now insert 521: for $i = 0$, we find an available slot $t[1] \in \{\perp, \Delta\}$, so:

0	1	2	3	4	5	6	7	8	9
100	521	289	361	144	225	196	256	324	169

assuming that duplicates are allowed. (Insert 361 instead of 521!)

Example 3: Inserting with Quadratic Probing

Insert into a hash table t of $m = 10$ slots the following $n = 10$ integers

$$100, 121, 144, 169, 196, 225, 256, 289, 324, 361$$

using $hash'(key) = key \bmod 10$ and $f(i) = i^2$.

After the first six insertions, t is:

0	1	2	3	4	5	6	7	8	9
100	121	$\perp$	$\perp$	144	225	196	$\perp$	$\perp$	169

Next, 256 collides with 196 for $i = 0$, but $t[7] = \perp$ for $i = 1$, so:

0	1	2	3	4	5	6	7	8	9
100	121	$\perp$	$\perp$	144	225	196	256	$\perp$	169

289 collides with 169 and 100 for $i = 0$ and 1, but $t[3] = \perp$ for $i = 2$:

0	1	2	3	4	5	6	7	8	9
100	121	$\perp$	289	144	225	196	256	$\perp$	169

324 collides with 144 and 225 for $i = 0$ and 1, but $t[8] = \perp$ for $i = 2$:

0	1	2	3	4	5	6	7	8	9
100	121	$\perp$	289	144	225	196	256	324	169

Finally, 361 collides with 121 for $i = 0$, but $t[2] = \perp$ for $i = 1$, so:

0	1	2	3	4	5	6	7	8	9
100	121	361	289	144	225	196	256	324	169

The number of probes shrunk by three compared to linear probing.

However, we were lucky to find $\perp$ slots, at least since the moment when $\alpha \geq 0.5$ (insert 200 instead of 256!): we should have rehashed then!

Expected Probe Length

Assumption of uniform hashing: Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence.
(Linear and quadratic probing only have m distinct probe sequences, while double hashing only has m^2 distinct probe sequences.)

For a given load factor, α, with $\alpha < 1$, the expected number of slots probed is at most $\frac{1}{1-\alpha}$ in an insertion or unsuccessful search,
and at most $\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}$ in a successful search or deletion.
Hence search, insertion, and deletion take $O(1)$ time on average!

Use this to predict the behaviour and guide the design of hash tables.

Example: If $\alpha = 50\%$ (respectively $\alpha = 90\%$), then an unsuccessful search probes 2 (respectively 10) cells, while a successful search probes 1.387 (respectively 2.559) cells.

Requirements for a General Hash Function

Recall that we are trying to write a function whose output appears to be a uniformly distributed random variable, when given an arbitrary, non-random input distribution (such as a set of people's names).

In other words, the $hash$ function must ensure that:

- Every bit of the key affects every bit of the hash value.
- Each bit of the hash value is equally likely.

Further:

- Any two keys that compare equal must hash to the same value.

For example, if you use a case-insensitive string comparison, then the $hash$ function must be case-insensitive as well.

Worst-Case Hashing

Although hashing takes $O(1)$ time on average, in the worst case it takes $O(n)$ time. The worst case arises when all keys hash to the same value, in which case, under the chaining approach, the hash table contains $m - 1$ empty bins and one bin containing a chain of length n: a successful probe reduces to linear search in a list of n elements, which indeed takes $O(n)$ time at worst.

If a good hash function is applied to n keys, it uniformly hashes them into the m available bins. The probability that all keys fall into any particular bin is

$$\frac{1}{m^n}$$

Even for $n = 1000 = m$, this is an extremely low probability.