Empirical data based on 900 table entries:

<table>
<thead>
<tr>
<th>Linear</th>
<th>Quadratic</th>
<th>Nonchaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.01</td>
<td>0.05</td>
<td>1.09</td>
</tr>
<tr>
<td>0.19</td>
<td>0.04</td>
<td>1.00</td>
</tr>
<tr>
<td>1.05</td>
<td>0.16</td>
<td>1.08</td>
</tr>
<tr>
<td>1.04</td>
<td>0.05</td>
<td>1.07</td>
</tr>
<tr>
<td>1.05</td>
<td>0.06</td>
<td>1.06</td>
</tr>
<tr>
<td>1.04</td>
<td>0.15</td>
<td>1.05</td>
</tr>
<tr>
<td>1.05</td>
<td>0.10</td>
<td>1.04</td>
</tr>
<tr>
<td>1.05</td>
<td>0.10</td>
<td>1.03</td>
</tr>
<tr>
<td>1.05</td>
<td>0.15</td>
<td>1.02</td>
</tr>
<tr>
<td>1.05</td>
<td>0.10</td>
<td>1.01</td>
</tr>
<tr>
<td>1.05</td>
<td>0.15</td>
<td>1.00</td>
</tr>
<tr>
<td>1.05</td>
<td>0.10</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Load Factor: 0.10 0.50 0.80 0.90
Introduction to Hash Tables

Graph:

Average Time:

\[ \text{AVG} \approx (1/n)(1 + \log_2 n) \]

Average Insertion Time:

\[ \frac{1}{n} \approx \frac{u}{p + 1} \]

Average Search Time to Find Item:

\[ E = \frac{p}{p + 1} \]

Conclusions for Closed (unbuckled) hashing:

and Rehashing functions being truly uniform.

But note, this analysis relies upon the hashing
constant time.

Space is always free; then hashing works in
so we can be confident that some percentage of
Thus, if there is enough room in a hash table,

then order of operations is constant.

Suppose load factor \( \frac{n}{b} \) bounded away from 1

What is \( O(\text{insertion}) \) or \( O(\text{search}) \)?
If \( b \) is about twice \( n \), then \( E \) is about 2.

Example

If \( b \) large enough, then this number is small.

Increases up to its limit of \( b^{-1} \).

Grows very slowly from \( 1 \) to \( b^{-1} \) as \( n \)

Notes:

\[
\frac{q + l}{q + l} = E
\]

After some (actually a lot of) algebra,

\[
q \quad b \quad \ldots \quad q \quad b \quad \ldots \quad q \quad b \quad \ldots
\]

\[
\frac{q}{q} + \frac{b}{b} + \frac{1}{1} + \frac{0}{0} + \frac{n}{n} \quad \ldots \quad \frac{n}{n} + \frac{1}{1} + \frac{0}{0} + \frac{n}{n} = \frac{p}{p} + \frac{p}{p} + \frac{p}{p}
\]

\[
\prod \text{Prob (at least 3 steps)}
\]

\[
\prod \text{Prob (at least 2 steps)}
\]

\[
\prod \text{Prob (at least 1 step)}
\]

\[
E = \prod \text{Prob (at least 1 step)}
\]

Expected number of steps for insertion is

Average using algebra, this average is

\[
\text{Average} = \frac{q}{q} \log_b (q - c)
\]

\[
\text{To get average work per insertion, divide by m}
\]

\[
\frac{m + 1 - m}{q + l} = \frac{n}{n} \quad \text{Total work} = \frac{n}{n} \quad \text{Total work should be}
\]

To insert \( m \) items, total work should be

\[
\text{Start with } n = 0, \text{ then consider } n = 1, \text{ etc.}
\]

Now consider average work for insertion.

where \( c = m / q \) is the loading factor.
2. If collision, then next position is computed as:

\[ \frac{n \cdot (n-1) \cdot (n-2) \ldots (n-i+1)}{b^i} \]

3. Similarly, likelihood of at least 1 collision is:

\[ \frac{b \cdot (b-1) \ldots (b-k+1)}{n \cdot (n-1) \ldots (n-k+1)} \]

Thus, if a collision occurs at first position, then the second position is (n-1) / (b-1) (n-1) / (b-1)

Thus, assuming the first collision is the last occurring, the b-1 remaining buckets will be n-1 remaining elements in the b buckets stored. As before, suppose the hash function is truly uniform.

Theoretical analysis:

Closed (unkeyed) Hashing

Thus, likelihood that an item already is at

so likelihood of specific array entry being filled is also not.

Fraction of array filled is n/b.

Consider algorithm

Analyses for individual insertion or search

only 1 data entry.

Suppose further that each bucket may contain

hash function h is truly uniform

\[ \text{Distance actually stored} \]

\[ \text{Distance (unkeyed) Hashing} \]
Then work for all operations is constant.

b is about n/2, so n/b is about 2
b is about n, so n/b is about 1

Examples:

- Enough so n/b is bounded.
- Suppose number of lists is chosen to be large.
- O(1 + n/b)

Work for insertion, deletion, searching is

n/b operations (maximum) to search
1 operation to determine which bucket

Total work required is:

- Should be close to this average.
- If hash function is truly uniform, then all lists

Elements per list.

Then there should be an average of n/b

Stored

Suppose b buckets and n elements actually

Theoretical analysis - Open (bucketed) Hashing

Much depends upon actual hash functions chosen.

Actually used:

n / sp, which gives the fraction of the array
Thus, work often given in terms of load factor.

Data likely

If array is almost full, large clusters of
should have small isolated clusters of data
- If array contains much unused space,

Amount of room in the actual array.

In practice, results clearly depend upon the

- Cost of unsuccessful searches
- Cost of successful searches

Useful to distinguish two types of results

- Actual data
- Use experiments to get comparisons about

Use theoretical results

Use probability and statistics about data

Can proceed in at least two ways

Comparison of methods of collision resolution
Introduction to Hash Tables

deleted
Need separate markers for empty and
defined
How do you know when something is not in
deleted

Reaching
one array, are all places eventually
in closed or unbucketed hashing, using only

chaining
-- probing (linear, quadratic, rehash function)
Overflows are handled by:

bucket is full, then overflow occurs.
next part of a bucket until the

If s < 1, storage can proceed in the
If s = 1, an overflow has occurred
a collision has occurred

x, y are synonyms
If h(x) = h(y), then

between 0 and b-1. i.e., for any name x,
acts on symbols or names and gives values
A hashing function is a function h which

If h(x) = h(y), then

Let $T$ be the number of all possible pairs.

Altogether, $H_T$ can store $s$ pairs.

$H_T(0), H_T(1), \ldots, H_T(p-1)$ are the $p$ buckets in $H_T$.

Consider the following picture:

More generally, it is not always possible to store $s$ pairs in $H_T$.

Above, each array element held $t$ pair <name, value>. Each bucket can hold $s$ pairs <name, value>.

$H_T$ is partitioned into $b$ buckets.

$H_T$: Hash Table.

Rehashing

$h_2\cdot h_3 \cdot h_4$, etc. are called rehash functions.

After $(h_1 \bmod p)$, try...

Use several hash functions $h_1, h_2, h_3$.

Rehashing

$\ldots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$
Introduction to Hash Tables

- Searching must continue until item found
- Groups tend to grow together
  - Strings with same hash value become
    - With this approach, strings tend to coalesce
      - a collision.

  - This approach is called linear probing after
    - until symbol or blank found

    ... h(symbol) + 2
    h(symbol) + 1

- If symbol not found, consider
  - h(symbol) indicates to start searching

- More precisely,
  - in example, we just moved down the array.

- Where to look next,
  - following a collision, we had to decide

- Collision has occurred.
  - When this situation arises, we say a
    - h(Jan) = h(Jun) = h(Jul)
  - e.g., h(May) = h(May)

- h maps several symbols to the same place

Difficulties:

Simple Example Using Months

Use array of 26 elements with

h(string) = first letter of string
Introduction to Hash Tables

Well possible hash functions perform best to run some experiments to see how
H & S also note that in many cases, it may

For more details on this and other

Now add

Folding at boundaries

0100
1110
0001
1110
0001
1110
0001
1110

Alternatively, reverse alternate pieces

Now add

Shift Folding

0100
1111
0000
1110
0100
0100

(except last piece may be smaller)

Divide data into pieces of some size

0100 1110 0010 1110 1100

Take bit pattern for case

3. Shifting and Folding

Start searching for specific symbol
Use hash function to determine where to
Put all data into one large array

Basic Ideas:

Closed Hashing of Unbucketed Hash Tables
2. Consider the string as a bit pattern of specified number of bits from the middle.

Now square this number and take some 0's and 1's.

Consider the string as a bit pattern of specified number of bits from the middle.

Now square this number and take some 0's and 1's.

H & S mention it. However, this function is not uniform. The letters of the alphabet are not equally likely; the function is not suitable for this purpose.

h(sympbol) = letter of alphabet

Since the first letters of words in English are a uniform hashing function.

If h satisfies this property, then h is called a uniform hashing function.

If h satisfies this property, then h is called a uniform hashing function.

h(sympbol) = letter of alphabet

Then each one should be equally likely.

If buckets are numbered 0, 1, 2, ..., b-1, want to fill up buckets equally.

(h(sympbol) must be defined for all symbols)

For each symbol, must get bucket.
2. Use standard search techniques within the bucket.

1. Use h to determine which bucket to process a symbol to

h is called a hashing function

Provide a function for indexing into the buckets

More generally,

...

trees
lists
arrays

For each list we can use
For each list we can be short
Which list to use
For each string, must be able to determine
Strings divided over 25 lists

Important properties of this approach

For months, consider the following:

First letter of alphabet

Place string in structure depending upon

Create (list) structure for each letter

Suppose symbols involve character strings

Simple example of Open Hashing
Introduction to Hash Tables

- More fun (Walker)
  - Scatter storage within a vector (Elsion)
  - Hashing or double hashing (Hale, Easton)
  - Hashing (Horowitz, Sahni)
  - Unbucketed hash tables (Garland)
  - Closed hashing (Aho, Hopcroft, Ullman)

- Tuning (Walker)
  - External linking (Elsion)
  - Method of chaining (Hale, Easton)
  - Hash chaining (Horowitz, Sahni)
  - Bucketed hash tables (Garland)
  - Open hashing (Aho, Hopcroft, Ullman)

Two variations

Then store and retrieve within the groups

- Divide objects into smaller groups
- Reduce n

New approach

- Arrange all objects in a structure
- Share with n objects

Basic approach so far

Hash Tables

The question: Of course, otherwise I would not have asked

Answer:

Efficiency?

Can we find even better ways to improve?

Question:

- Log n
- Binary search tree
- Log n
- List
- Log n
- Ordered array

Structure

Approaches so far:

- Efficient
- Arranged pairs to make searching most
- Goal:

- Want to retrieve value based on symbol
- Want to store <symbol, value> pairs

Familiar problem: