Hash Tables

Familiar Problem:
-- Want to store \(<symbol, value>\) pairs
-- Want to retrieve \(value\) based on \(symbol\)

Goal:
-- Arrange pairs to make searching most efficient

Approaches so far:

<table>
<thead>
<tr>
<th>Structure</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered array</td>
<td>(n)</td>
</tr>
<tr>
<td>Ordered array</td>
<td>(\log n)</td>
</tr>
<tr>
<td>List</td>
<td>(n)</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>(\log n)</td>
</tr>
</tbody>
</table>

Question:
Can we find even better ways to improve efficiency?

Answer:
Of course. (Otherwise, I would not have asked the question.)
Basic Approach So Far

-- Started with $n$ objects
-- Arranged all objects in a structure

New Approach

-- Reduce $n$
-- Divide objects into smaller groups
  Then store and retrieve within the groups

Two variations
(again, terminology varies)

-- open hashing (Aho, Hopcroft, Ullman)
  bucketed hash tables (Garland)
  hash chaining (Horowitz, Sahni)
  method of chaining (Hale, Easton)
  external linking (Elson)
  fun (Walker)

-- closed hashing (Aho, Hopcroft, Ullman)
  unbucketed hash tables (Garland)
  hashing (Horowitz, Sahni)
  hashing or double hashing (Hale, Easton)
  scatter storage within a vector (Elson)
  more fun (Walker)
**Simple Example of Open Hashing**

Suppose symbols involve character strings

-- Create (list) structure for each letter
-- place string in structure depending upon first letter of alphabet

For Months, consider the following:

<table>
<thead>
<tr>
<th>Array</th>
<th>List Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>A→</td>
<td>Apr → Aug</td>
</tr>
<tr>
<td>B→</td>
<td>Dec</td>
</tr>
<tr>
<td>C→</td>
<td>Feb</td>
</tr>
<tr>
<td>D→</td>
<td>Jan → Jun → Jul</td>
</tr>
<tr>
<td>E→</td>
<td>Mar → May</td>
</tr>
<tr>
<td>F→</td>
<td>Nov</td>
</tr>
<tr>
<td>G→</td>
<td>Oct</td>
</tr>
<tr>
<td>H→</td>
<td>Sept</td>
</tr>
</tbody>
</table>
Important properties of this approach

◊ strings divided over 26 lists
◊ for each string, must be able to determine which list to use
◊ each list can be short
◊ for each list can use arrays
   lists
   trees
   ...

More generally,
◊ Divide items into b pieces or buckets
◊ Provide a function
   h(symbol) --> bucket number
   h is called a hashing function
◊ To process a symbol
   1. use h to determine which bucket to consider
   2. use standard search techniques within the bucket
Properties desired of a Hash function $h$

◊ For each symbol, must get bucket. $(h(\text{symbol})$ must be defined for all symbols)

◊ Want to fill up buckets equally. If buckets are numbered $0, 1, 2, \ldots, b-1$, then each one should be equally likely. $(\text{prob}(h(\text{symbol})=i) \text{ should be } 1/b \text{ for all } i)$

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

Since the first letters of words in English are not all equally likely, the function $h(\text{symbol}) = \text{letter of alphabet}$ is not uniform.
Possible Hashing Functions

◊ Use internal representation of characters

1. Use sum of (ASCII) codes, mod b

   Example:
   
   string = 'cat'
   letter   code
   c        99
   a        97
   t        116
   h('cat') = 99 + 97 + 116 mod 26
            = 0
   and 'cat' would be in bucket 0

Here, choice of b is critical.
H & S suggest choosing b so that b
has no prime divisors less than 20.

2. Consider the string as a bit pattern of
   0's and 1's.
   e.g., 'cat' = 011000110110000101110100
   Now square this number and take some
   specified number of bits from the middle

   This approach is used fairly widely, so
   H & S mention it. However, this function
   has some poor statistical properties and is
   viewed with some skepticism by
   mathematicians & statisticians
3. Shifting and Folding
   Take bit pattern for 'cat'
   01100011011000010110100
   Divide data into pieces of same size
   (except last piece may be smaller)
   01100
   01101
   10000
   10111
   01000

   Shift folding
   Now add
   1100100

   Alternately, reverse alternate pieces
   01100
   10110 (reversed)
   10000
   11101 (reversed)
   01000

   Folding at boundaries
   Now add
   11100011

◇ For more details on this and other approaches, see H & S

◇ H & S also note that in many cases, it may be best to run some experiments to see how well possible hash functions perform
Closed Hashing or Unbucketed Hash Tables

Basic ideas:
◊ Put all data into one large array
◊ Use hash function to determine where to start searching for specific symbol
Simple Example Using Months

Use array of 26 elements with
h(string) = first letter of string

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

Apr
Aug
Dec
Feb
Jan
Jun
Jul
Mar
May
Oct
Nov
Sep

Introduction to Hash Tables
**Difficulties:**

◊ h maps several symbols to the same place e.g., \( h(\text{Mar}) = h(\text{May}) \) 
  \( h(\text{Jan}) = h(\text{Jun}) = h(\text{Jul}) \) 

When this situation arises, we say a *collision* has occurred.

◊ Following a collision, we had to decide where to look next. 
In example, we just moved down the array. More precisely, 
\[ h(\text{symbol}) \text{ indicates to start searching} \]
\[ \text{if symbol not found, consider} \]
\[ h(\text{symbol}) + 1 \]
\[ h(\text{symbol}) + 2 \]

... 
until symbol or blank found 
This approach is called *linear probing* after a collision.

◊ With this approach, strings tend to coalesce 
  -- Strings with same hash value become a group 
  -- Groups tend to grow together

◊ Searching must continue until item found or until blank found
Further Improvements

Linear probing specified that after searching h(symbol), one should try successive locations:

\[ h(\text{symbol}) + 1 \]
\[ h(\text{symbol}) + 2 \]
\[ \ldots \]

Some improvement is possible if subsequent locations are more scattered.

◊ Quadratic probing

After h(symbol), try

\[ h(\text{symbol}) + 1^2 \]
\[ h(\text{symbol}) + 2^2 \]
\[ h(\text{symbol}) + 3^2 \]
\[ h(\text{symbol}) + 4^2 \]
\[ \ldots \]
\[ h(\text{symbol}) + i^2 \]
\[ \ldots \]

◊ Rehashing

Use several hash functions \( h_1, h_2, h_3, \ldots \)

After \( h_1(\text{symbol}) \), try

\[ h_2(\text{symbol}) \]
\[ h_3(\text{symbol}) \]
\[ h_4(\text{symbol}) \]

\( h_2, h_3, \ldots \) are called rehash functions
Recap and Additional Terminology

◊ HT : Hash Table.

◊ HT is partitioned into $b$ buckets

◊ Each bucket can hold $s$ pairs <name, value>
  -- Above, each array element held 1 pair, and $s = 1$.
  -- More generally, if array used to store pairs, consider the following picture
    HT(0)..HT(b-1) are the $b$ buckets in HT

      |      |      |      |
      |------|------|------|
HT(0) |      |      |      |
      |------|------|------|
HT(1) |      |      |      |
      |------|------|------|
HT(2) |      |      |      |
      |------|------|------|
HT(3) |      |      |      |
      |------|------|------|
...   |      |      |      |
      |------|------|------|
HT(b-1)|      |      |      |
      |------|------|------|

Altogether, HT can store $sb$ pairs

◊ Let $T$ be the number of all possible <name, value> pairs and let $n$ be the number of pairs actually stored. Then
  \[
  \text{name density} = \frac{n}{T}
  \]
  \[
  \text{loading density or factor is } \alpha = \frac{n}{s^*b}
  \]
A hashing function is a function $h$ which acts on symbols or names and gives values between 0 and $b-1$. I.e., for any name $X$, $0 \leq h(X) \leq b-1$

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

-- $x,y$ are *synonyms*
-- a *collision* has occurred
-- if $s = 1$, an *overflow* has occurred
-- if $s > 1$, storage can proceed in the next part of a bucket until the bucket is full. Then *overflow* occurs.
◊ Overflows are handled by:
  -- probing (linear, quadratic, rehash func.)
  -- chaining

◊ In closed or unbucketed hashing, using only one array, are all places eventually reached?
  -- Yes, so no waste of space
  -- But eventually hashing can revert to linear searching.

◊ Deletion
  How do you know when something is not in the table?
  -- Need separate markers for empty and deleted
Comparison of methods of collision resolution

Can proceed in at least two ways

-- Use probability and statistics about data to get theoretical results

-- Use experiments to get comparisons about actual data

Useful to distinguish two types of results

-- Cost of successful searches
-- Cost of unsuccessful searches

In practice, results clearly depend upon the amount of room in the actual array.

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

Thus, work often given in terms of load factor, \( n / sb \), which gives the fraction of the array actually used.

Much depends upon actual hash functions chosen
Theoretical Analysis - Open (Bucketed) Hashing

Suppose b buckets and n elements actually stored

Then there should be an average of n/b elements per list.

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

Total work required is:

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

Work for insertion, deletion, searching is \( O(1 + n/b) \).

Suppose number of lists is chosen to be large enough so n/b is bounded.

Examples:

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

Then work for all operations is constant.
Theoretical Analysis -
Closed (Unbucketed) Hashing

As before, suppose
b buckets
n elements actually stored
hash function h is truly uniform

Suppose further that each bucket may contain
only 1 data entry.

Analysis for individual insertion or search
of entry E:

Consider algorithm
1. h(E) is computed

Fraction of array filled is n/b,
so likelihood of specific array entry being
filled is also n/b

Thus, likelihood that an item already is at
position h(E) is n/b .
2. If collision, then next position is computed.

Disregarding the initial position, there will be n-1 remaining elements in the b-1 remaining buckets

Thus, assuming the first collision, likelihood than a collision also occurs at the second position is \((n-1) / (b-1)\)

Likelihood of a collision both at first and second positions is

\[
\frac{n(n-1)}{b(b-1)}
\]

3. Similarly, likelihood of at least i collisions is

\[
\frac{n(n-1)(n-2) \ldots (n-i+1)}{b(b-1)(b-2) \ldots (b-i+1)}
\]
Expected number of steps for insertion is

\[ E = \sum i \cdot \text{prob}(i \text{ steps}) \]

\[ = \sum \text{prob(at least } j \text{ steps)} \]

\[ = \text{prob (at least 1 step)} + \text{prob (at least 2 steps)} + \text{prob (at least 3 steps)} + \ldots + \text{prob (at least } b \text{ steps)} \]

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

After some (actually a lot of) algebra,

\[ E = \frac{b + 1}{b + 1 - n} \]

**Notes:**

E grows very slowly from 1 to \( b-1 \) as \( n \) increases up to its limit of \( b-1 \).

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

**Example**

If \( b \) is about twice \( n \), then \( E \) is about 2
Now consider average work for insertion.
Start with \( n = 0 \), then consider \( n = 1 \), etc.

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

\[
\text{TotalW} = \sum_{n=0}^{m-1} \frac{b + 1}{b + 1 - n}
\]

To get average work per insertion, divide by \( m \)

Again, using algebra, this average is

\[
\text{AvgW} = \frac{b \log_e(b)}{m} - \frac{(1/\alpha) \log_e(1 - \alpha)}{b + 1 - m}
\]

where \( \alpha = m / b \) is the loading factor
Conclusions for closed (unbucketed) hashing:

Average search time to find item:

\[
E = \frac{b + 1}{b + 1 - n} \approx \frac{1}{1 - \alpha}
\]

Average insertion time:

\[
\text{AvgW} \approx -(1/\alpha) \log_e (1 - \alpha)
\]

Graph:

\[
\frac{1}{1 - \alpha} = \text{cost of insertion or unsuccessful search}
\]

\[-(1/\alpha) \log (1 - \alpha) = \text{cost of deletion or successful search}\]

\[\alpha = N/B = \text{fraction of table full}\]

Average operation cost.
What is $O(\text{insertion})$ or $O(\text{search})$?

Suppose load factor $\propto$ bounded away from 1 (e.g., $n/b \leq 0.5$ or $2n \leq b$) then order of operations is constant.

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

(But, note, this analysis relies upon the hashing and rehashing functions being truly uniform.)
### THEORETICAL vs EMPIRICAL COMPARISON

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>0.10</th>
<th>0.50</th>
<th>0.80</th>
<th>0.90</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Successful</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chaining</td>
<td>1.05</td>
<td>1.25</td>
<td>1.40</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>1.04</td>
<td>1.20</td>
<td>1.40</td>
<td>1.40</td>
</tr>
<tr>
<td>nonchaining</td>
<td>1.06</td>
<td>1.50</td>
<td>3.00</td>
<td>5.50</td>
</tr>
<tr>
<td>(quadratic)</td>
<td>1.04</td>
<td>1.50</td>
<td>2.10</td>
<td>2.70</td>
</tr>
<tr>
<td>(linear)</td>
<td>1.05</td>
<td>1.60</td>
<td>3.40</td>
<td>6.20</td>
</tr>
<tr>
<td><strong>Unsuccessful</strong></td>
<td>0.10</td>
<td>0.50</td>
<td>0.80</td>
<td>0.90</td>
</tr>
<tr>
<td>chaining</td>
<td>0.11</td>
<td>0.53</td>
<td>0.78</td>
<td>0.90</td>
</tr>
<tr>
<td>nonchaining</td>
<td>1.12</td>
<td>2.50</td>
<td>13.00</td>
<td>50.00</td>
</tr>
<tr>
<td>(quadratic)</td>
<td>1.13</td>
<td>2.20</td>
<td>5.20</td>
<td>11.90</td>
</tr>
<tr>
<td>(linear)</td>
<td>1.13</td>
<td>2.70</td>
<td>14.40</td>
<td>59.80</td>
</tr>
</tbody>
</table>

*Empirical data based on 900 table entries, produced by a pseudo-random number generator (Source: Robert L. Kruse)*