Motivation

- Suppose we have 16,000,000 data items. We would like to be able to find a particular item.
  - How long will it take to find the item?
- It depends on the data structure.

Motivation...

- If the data structure is a linked list and items are not sorted, the search time is $O(N)$.
- If the data structure is a linked list and items are sorted, the search time of a binary search is $O($           $)$.
- If the data structure is a binary search tree, the search time can be reduced to $O($           $)$.

$$\log_2 16,000,000 = 24$$

Motivation...

- Can we do even better than $O(\log_2 N)$?
- Yes we can. We can use “hash” functions.

Hashing

- The general idea behind hashing is to directly map each data item into a position in a table using some function $h$.
  - Complex data items must have a ‘key’ in order to perform the hashing. Then: $position = h(key)$.
- Generally we use an array of some fixed size to hold the data.

Example

An Array

<table>
<thead>
<tr>
<th>Data Item</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: Tom Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>University: NMT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Height: ……</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weight: ……</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insurance: ……</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etc…….</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h(\text{Smith}) = 1$.

Suppose we use a hashing function that uses the last name of a student and returns a 1. Then place the data record into slot 1 of the array. Note the array is of size $T$. 

$T=1$
Hash Functions: Division

- If the keys are integers then $key \% T$ is generally a good hash function.

An Example

<table>
<thead>
<tr>
<th>0</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>91</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
</tr>
<tr>
<td>9</td>
<td>49</td>
</tr>
</tbody>
</table>

Here is a hash table of size $T = 10$, where the entries 10, 25, 34, 49, 76, and 91 have been inserted. The hash function was $h(key) = key \% 10$. The empty table entries are omitted.

Hash Functions: Division (cont.)

- What happens if more than one key hash to the same position?
- The problem is called a "collision".
- Dealing with collisions is called "collision resolution".
- In general, to avoid situations like this, $T$ should be a prime number.

Hash Functions: Folding

- A key is divided into several parts, and use a simple operation to combine them in a certain way.
- Shift folding
- Boundary folding

Hash Functions: Folding (cont.)

- If the keys are strings the hash function is some function of the characters in the strings.
- One possibility is to simply add the ASCII values of the characters:

$$h(str) = \left( \sum_{i=0}^{\text{length-1}} \text{str}[i] \right) \% T$$

Hash Functions: Folding (cont.)

- Another possibility is to convert the string into some number in some arbitrary base $b$ ($b$ also might be a prime number):

$$h(str) = \left( \sum_{i=0}^{\text{length-1}} \text{str}[i] \times b^i \right) \% T$$

Example: $h(ABC) = (65b^0 + 66b^1 + 67b^2) \% T$
Rule of Thumb

- Hash functions should try to achieve uniform full coverage of the hash table, while minimizing collisions.
- Since this is usually impossible, and collisions will almost always occur, an important design consideration is how you deal with the collision resolution.

Open Addressing

- “Open addressing” resolves collisions by trying alternative slots in the hash table, until an empty cell is found.
- In general we try cells in the following order:
  \[ h_0(key), h_1(key), h_2(key), \ldots \]
  where \( h_i(key) = [h(key) + f(i)] \mod T \) and \( f(0) = 0 \).

Example

Here is a hash table of size \( T = 10 \), where the entries 0, 1, 4, 9, 16, 25, 36, 49, 64, and 81 have been inserted. The hash function was \( h(key) = key \mod 10 \) and \( f(i) = i \).

Chaining

- With “separate chaining” all data items that hash to the same number are kept in a list (often a linked list). So, each entry in the hash table can be a linked list of arbitrary length.
- To find a piece of data we use the hash function to find the correct list, and then we search the list.

Example

The three entries are now stored in a linked list.

Another Example

Here is a hash table of size \( T = 10 \), where the entries 0, 1, 4, 9, 16, 25, 36, 49, 64, and 81 have been inserted. The hash function was \( h(key) = key \mod 10 \).
Bucket Addressing

- Store colliding elements in the same position in the table
- When a bucket is full, the open addressing or chaining addressing can be used.