Introduction to Hashing

Want to maintain a collection (dictionary) $D$ of items $(k, e)$ to support

- **findElement**$(k)$: If $D$ contains an item with key equal to $k$, then return the element. Otherwise, return a sentinel **NO_SUCH_KEY**

- **insertItem**$(k, e)$: Duplicates may or may not be allowed.

- **removeElement**$(k)$

Hashing is the method of choice (expected time $O(1)$) where there are $n$ elements in $D$) especially when **removeElement**() is not needed. Main “defects”

- Ordering is not taken into consideration. Cannot be used to find the minimum.

- Performs poorly for near matches (e.g. **findElement**$(k')$, $k' \approx k$)

- Non-productive space utilization.

Popular uses: Search engines, symbol tables in compiler, spell checkers
Hashing Concepts

Main idea: Use an array of size $m$ and the key $k$ as address of the array. A hash function $h$ is used to map keys to $[0..m - 1]$.

Key technical issues

- What is a good $h$? A good function avoids (but does not eliminate) collisions, and is quick to compute.
- How do we resolving collisions? Retrieval time is a function of collisions.
- What if we run out of space in the table?
- Can we rearranging keys upon an insertion?

Main memory versus disk based storage
About Hashing Functions

Convert keys to hash codes (numbers), Convert hash codes to legal array indices. Use every bit of key (e.g. temp01, temp02,..., temp10)

• Converting strings. \( h(\text{cmsc 420}) = \ldots \) ?
  View string as a \( k \)-tuple \((x_0, x_1, \ldots, x_{k-1})\).
  1. Compute a number: \( \Sigma_i x_i \). Works poorly for \text{cmsc 420} and \text{cmsc 402}
  2. Compute a number \( \Sigma_i a^i x_i \). Can be computed in linear time using Horner’s rule. Good values for \( a \) are 33, 37, 39, 41.

• Handling overflows (assuming keys are integers)
  1. Division method: \( h(k) = k \mod m \). Works poorly, for example, if keys are \{200, 205, 210, \ldots, 600\} and \( m = 100 \) (lots of unused locations, and four collisions per index). Choose \( m \) to be prime.
  2. MAD \( h(k) = ak + b \mod m \). (\( a \) and \( b \) are non-negative integers randomly chosen, \( a \mod m \neq 0 \)...)
Sample Hash Function

Direct version based on Horner’s rule:

```java
public static int hash (String key, int tableSize) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++)
        hashVal = (37 * hashVal + key.charAt(i)) % tableSize;
    return hashVal;
}
```

Faster version (computes modulo outside the loop)

```java
public static int hash (String key, int tableSize) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++)
        hashVal = 37 * hashVal + key.charAt(i);
    hashVal %= tableSize;
    if (hashVal < 0) hashVal += tableSize;
    return hashVal;
}
```
Handling Collisions: Separate Chaining

- Each of the $m$ locations serve as head pointers to $m$ linked lists
- Inserting, removing, and searching: Simple to implement
- Example: Last digit as hash code

Analysis: Define the load factor $\lambda$ to be the fraction of the table that is full ($\lambda = n/m$). Assume that key $k$ is hashed to location $i$ with probability $1/m$ independent of $i$. We expect all linked lists to be equally long.

- Insertion: The expected length of a list is $\lambda$. If we insert at the end (for instance to check duplicates), probe time is $(1 + \lambda) \in O(1)$
- Unsuccessful search also takes roughly the same amount of time.
- Successful search takes roughly $1 + \frac{\lambda}{2}

Problems: Additional memory for pointers, memory allocation
Handling Collisions: Open addressing

Avoid using external memory (use cells in the table itself)
Use a special key value **EMPTY** to determine which entries have keys and which do not
Use secondary search function $f$: If $h(k)$ is occupied, try $(h(k) + f(1)) \mod m$, $(h(k) + f(2)) \mod m$, $(h(k) + f(3)) \mod m$, and so on. The sequence is called a probe sequence
Linear probing uses $f(i) = i$.

- Insertion and find are similar
- As the table becomes full, it becomes harder and harder to find empty spaces (aka *primary clustering*).
Deletion is non-trivial

Consider linear probing

- Insert key $k$ at position $i$, insert colliding key $k'$ in position $(i + 1) \mod m$
- Delete key $k$ making table entry $i$ empty
- Search for key $k'$ finding empty slot on the way?

Solution: Add a lazy deletion flag

- Search proceeds over any entry with deleted bit
- Insertion occurs at the first position that is empty or has deletion bit set
How Good is Linear Probing?

Naive analysis assumes large hash table, and each probe independent of the previous probe.

- Some experiments can be categorized into two: **success** and **failure**. Say probability of success is \( p \).
  - Example: Toss a coin. Success if head. \( (p = 0.5) \).
  - How many trials occur, on the average, before we get success? \( 1/p \)
- Since the probability of a cell being empty is \( 1 - \lambda \), expected probes for insertion is \( I = 1/(1 - \lambda) \)
- The number of probes needed for a successful search \( S \) depends on the probes needed when the item was inserted. The average value is

\[
\frac{1}{\lambda} \int_{0}^{\lambda} I(x) \, dx = - \frac{\log(1 - \lambda)}{\lambda} \tag{1}
\]

Better Analysis: \( I = \frac{1}{2}(1 + 1/(1 - \lambda)^2) \) and \( S = \frac{1}{2}(1 + 1/(1 - \lambda)) \).
Handling Collisions: Quadratic probing

Primary clustering makes the average probe sequence longer, and also makes a long probe sequence more likely. Quadratic probing \( f(i) = i^2 \) scatters probes around more effectively. Probe sequence is \( h(k), h(k) + 1, h(k) + 4, h(k) + 9, h(k) + 16, \ldots \)

```c
find (int key, int T[]) {
    int i = 0; c = hash(key);
    while (T[c] != EMPTY) {
        c += 2 * (++i) - 1;
        c = c % m;
    }
    return c;
}
```

Quadratic probing is efficient:
\[
H^{(i)}(k) - H^{(i-1)}(k) = i^2 - (i - 1)^2 = 2i - 1
\]
Quadratic Probing Guarantees Finding Empty Cell

If table size \( m \) is prime, load factor is less than 0.5, a new item can always be inserted. No cell is probed twice.

- If the first \( \lceil \frac{m}{2} \rceil \) probes are distinct, then we must get an empty cell.
- Consider two probes \( h + i^2 \mod m \) and \( h + j^2 \mod m \) where \( 0 \leq i, j \leq \lfloor \frac{m}{2} \rfloor \). Suppose, by way of contradiction, \( i \neq j \) but the probe values are identical. This requires (modulo \( m \))
  
  \[- i^2 = j^2 \]
  
  \[- i^2 - j^2 = 0 \]
  
  \[- (i - j)(i + j) = 0 \]
  
  - Either \( i - j \) is divisible by \( m \)
  - Or \( i + j \) is divisible by \( m \)
- A contradiction results in either case
Handling Collisions: Double Hashing

Use another hashing function to scatter things around (that is, probe sequence is chosen independent of primary position): \( H(k) = h(k) + ig(k) \)

Performance as predicted by Equation 1.

Incremental formula: \( H(k, 0) = h(k), \ H(k, p + 1) = H(k, p) + g(k) \mod m \)

To ensure that the probe sequence visits all entries, \( g(k) \) must be relatively prime to \( m \)

- If \( d > 1 \) were a common divisor, \( \left( \frac{m}{d} g(k) \right) \mod m = \left( m \frac{g(k)}{d} \right) \mod m = 0 \)
- The offset is zero. The zeroth probe location \( h(k) \) is the same as the \( \frac{m}{d} \)th probe location.
Self Organizing Double Hashing

During the insertion process, rearrange keys so that collision lists are shorter. Insertion becomes more expensive. Subsequent searchers are faster (good, when searches are more common than insertion).

Let $p_0, p_1, \ldots, p_t$ be the sequence of addresses generated by a double hashing function for some key $k$ before encountering an empty table at position $p_t$. Where do we place $k$?

- If we insert $k$ at $p_t$, then any subsequent search for $k$ will require $t + 1$ probes.

- If we place $k$ in $p_r$, $0 \leq r < t$, then $k$ will require less probes (savings of $t - r$). However, we must move key $k'$ currently sitting at $p_r$ to some place, possibly at the end of its collision chain.
Brent Double Hashing: Example

• Inserting Rudy at the end of the chain (location $p_4$) costs 4 useless probes for subsequent search of Rudy.

• Replacing Alan (location $p_1$) with Rudy, and then inserting Alan after Ruth would save 3 probes for Rudy (4-1), but cost 2 extra probes for Alan.

• Net saving of 1 if we reorganize. Notice that we have conceptually organized the data as a two dimensional grid.

<table>
<thead>
<tr>
<th></th>
<th>Tim</th>
<th>Alan</th>
<th>Jay</th>
<th>Katy</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joan</td>
<td>-</td>
<td>Ruth</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ron</td>
<td>Rita</td>
<td>-</td>
<td>-</td>
<td>Alex</td>
<td>-</td>
</tr>
<tr>
<td>Rita</td>
<td>-</td>
<td>-</td>
<td>Bob</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Each column represents a collision chain. A “-” signifies an empty location.

We attempt to insert Rudy which takes us along the probe sequence Tim, Alan, Jay, and Katy.
Brent Double Hashing: Algorithm

Think of (but don’t allocate!) a two dimensional grid

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0$</td>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$p_4$</td>
</tr>
<tr>
<td>$p_0 + c_0$</td>
<td>$p_1 + c_1$</td>
<td>$p_2 + c_2$</td>
<td>$p_3 + c_3$</td>
<td>$p_4 + c_4$</td>
</tr>
<tr>
<td>$p_0 + 2c_0$</td>
<td>$p_1 + 2c_1$</td>
<td>$p_2 + 2c_2$</td>
<td>$p_3 + 2c_3$</td>
<td>$p_4 + 2c_4$</td>
</tr>
<tr>
<td>$p_0 + 3c_0$</td>
<td>$p_1 + 3c_1$</td>
<td>$p_2 + 3c_2$</td>
<td>$p_3 + 3c_3$</td>
<td>$p_4 + 3c_4$</td>
</tr>
</tbody>
</table>

The $i$th column represents a collision chain. Columns may have different lengths.
The $j$th row contains the address of the $j$th element on each collision chain (or is empty)
Try to insert $k$ as close to $p_0$ as possible
Traverse the array in a zig-zag order ($p_0$, $p_1$, $p_0 + c_0$, $p_0 + 2c_0$, $p_1 + c_1$, $p_2$, $p_3$, $\ldots$) looking for an empty slot.
Gonnet Munro Double Hashing: Example

Brent’s algorithm only moves keys on the probe sequence of the insertion key to the end of their probe sequences.
Gonnet-Munro will consider moving keys that are on these probe sequences, and on those keys’ probe sequences also.

- Net saving of 1 with Brent
- Replacing Tim with Rudy (saving 4 probes), replacing Joan with Tim (costing 1), and moving Joan one position down (costing 1) beats the Brent algorithm
Gonnet-Munro Search Tree

Search is conducted using a “binary tree”

- Leftmost branch: Original probe sequence $p_0, p_1, \ldots, p_t$
- Right branches: Probe sequences starting from these positions
- Root: $h(Tim) + qg(Tim)$
- Alan: $h(Rudy) + g(Rudy)$
- Joan: $h(Tim) + (q+1)g(Tim)$
Gonnet-Munro Data Structure

Each node in the binary tree has fields for

- String: A key (corresponding to LOC, see below).
- Integer (LOC): The location in the hash table at which this key value is stored
- Integer (INC): The secondary hash function \((g)\) value

The root of the tree: Key, the original key at position LOC; LOC = \(h(k)\), where \(k\) is the key to be inserted; INC = \(g(k)\);
The left child for any node \(P\) in the tree has LOC = LOC(P) + INC(P); INC = INC(P);
The right child for any node \(P\) in the tree has LOC = LOC(P) + g(Key(P)); INC = g(Key(P))
Key = HashTable[LOC] in either case
Gonnet-Munro Algorithm

Generate probe tree in breadth-first manner (top to bottom, and left to right) until encountering the first null position
Keys will be moved along the path from the root to this empty position using the following recursive rule
Algorithm is at node N holding key $K_N$ trying to insert key $K$. If $K_N$ is empty, insert. Otherwise if the path mandates moving to

- Left child: Do nothing. Algorithm is now at node $\text{left}(N)$ holding key $K_{\text{left}(N)}$ trying to insert key $K$

- Right child: Substitute $K$ at location currently occupied by $K_N$. Algorithm is now at node $\text{right}(N)$ holding key $K_{\text{right}(N)}$ trying to insert key $K_N$

Applying the algorithm in the example results in

1. Path generated: Right, Right
2. Substitute Tim by Rudy, Joan by Tim, insert Joan at free spot