Hashing

• **Data structures we have considered**
  – Use comparisons to find items
  – Good behavior is $O(\log n)$ for search and insert
  – $\log(100)=6.6$ and $\log(100,000)=16.6$

• **Hash tables**
  – Support only *search*, *insert* and *delete*
  – Designed for $O(1)$ behavior on each operation.
  – Efficient use of space: roughly the number of items handled.
Dictionary ADT

• \( U \) = Universe of possible keys

• \( K \subseteq U \) - Set of keys actually handled

Operations:
   - Search(key)
   - Insert(x)
   - Delete(x)

Possible implementations: Linked list, BST, Array, Hash table
Terminology

- **U** is the *Universe* of all possible keys
- **m** is the number of *slots* in the data structure
- **K ⊆ U** - the set of *objects* that are actually inserted into the dictionary
- **n = |K|** is the # of objects in the dictionary
- The load factor is **α = n / m**
Linked List implementation

- **Insert**\((x)\) - add \(x\) at head of list
  
  **running time:** \(O(1)\)

- **Search**\((k)\) - start at head and scan list
  
  **worst-case:** \(n\) operations
  **average-case:** \(n/2\) operations
  **best-case:** \(1\) operation

- **Delete**\((x)\) - start at head, scan list and delete
  
  **running time:** same as **Search**

**Space usage:** \(O(n)\) - very efficient
**Array implementation**

**Data structure:** Array $T[0..m-1]$, $m = |U|$

- Search($T$, $k$) – return $T[k]$
- Insert($T$, $x$) – $T[\text{key}[x]] \leftarrow x$
- Delete($T$, $x$) – $T[\text{key}[x]] \leftarrow \text{NIL}$

**Running time:** $O(1)$ for each operation

**Space usage:** $O(m)$ always!
- good if $n = \Theta(m)$
- bad if $n \ll m$ (e.g., if keys are long strings)
Array implementation - picture
Hash Tables

**Hash Table:** an array \( T[0..m-1] \), \( m << |U| \)

**Hash function:** \( h: U \rightarrow \{0,1,...,m-1\} \)

- \textbf{Search}(k) - return \( T[h(k)] \)
- \textbf{Insert}(x) - \( T[h(\text{key}[x])] \leftarrow x \)
- \textbf{Delete}(x) - \( T[h(\text{key}[x])] \leftarrow \text{NIL} \)

Using hash tables, instead of searching we “go directly to the key” location in the table.
Array implementation

- **Search(k)** – return $T[k]
- **Insert(x)** – $T[\text{key}[x]] \leftarrow x$
- **Delete(x)** – $T[\text{key}[x]] \leftarrow \text{NIL}$
Hashing - picture
Collisions

Hash Table Size $\ll$ Universe Size

\[\downarrow\]

For many pairs $k_1 \neq k_2$ we have $h(k_1) = h(k_2)$

- If we try to insert $k_2$ to a table containing $k_1$, we get a collision because $T[h(k_2)] = T[h(k_1)]$

- The hash table operations need to be modified to handle collisions correctly.
Choosing a Hash Function

The actual key values matter

• They often are a particular subset of $U$:
  – English words
  – I.d. numbers from certain years
  – Variable names or keywords
  – Phone numbers

A Hash Function should:

• Be easy to compute
• Create few collisions:
  – inverse should be evenly distributed
  – not be vulnerable to non-random patterns
Perfect Hashing

If all keys are known in advance, can sometimes design a simple one-to-one hash function.

Example:

\[ hash(s) = s \mod 10 \]

<table>
<thead>
<tr>
<th>s</th>
<th>120</th>
<th>331</th>
<th>912</th>
<th>74</th>
<th>665</th>
<th>47</th>
<th>888</th>
<th>219</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(s)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Division Method - the \texttt{mod} Hash function

A common type of hash function has the form

\[ h(key) = key \mod size \]

(remainder of \texttt{key} divided by \texttt{size})

where \texttt{size} is the table size \( m \)

- What if \texttt{size}=12 and all keys are \( 12k+2 \) (2, 26, 50,\ldots)
  - All keys hash to same index (lots of collisions!)
  - We cannot store more than one key in a cell
  - Must pick table size carefully: a prime number is usually a good choice for \texttt{size}
Hashing Strings

We often use hashing to store strings. We want to consider a number associated with a string $c_0c_1...c_t$:

- One option is to use $key = c_0 + c_1 + ... + c_t$ where the $c_i$ used are ASCII values of the characters
  - If strings are short – keys are concentrated

- More appropriate – consider the string as a number in base 128 or 256
  - Can use Horner’s rule to Hash:
    
    \[
    r \leftarrow 0; \\
    \text{for } i=1 \text{ to } t \text{ do} \\
    r \leftarrow (c_i + 256r) \mod size
    \]
Resolving Collisions

Chaining:
Elements are stored outside the table
(Table locations are pointers to "buckets")
Load factor $\alpha$ is allowed to be $> 1$

Open Addressing:
On collision, look for another place, in the table
Load factor $\alpha$ must be $< 1$
Chaining - picture

h(a) = h(b) = 1
h(d) = h(g) = 4

There is usually no advantage to using sorted lists or BSTs
Chaining

- Linked list at each table location
- The table entries are pointers to list heads

- Search(T, k) - search in the list at T[h(k)]
- Insert(T, x) - insert x at head of list at T[h(key[x])]
- Delete(T, x) - delete x from the list at T[h(key[x])]

Running Time:

Insert - \( O(1) \)

Search/Delete - proportional to list length (avg. 1+\( \alpha \))
worst-case: time to compute \( h(k) + \Theta(n) \) BAD!
Open Addressing (Probing)

- All elements are in the table itself, no pointers
- Must have $\alpha < 1$
- Basic idea in probing: Cell full? Keep looking...
- Insert - probe hash table, until find empty slot
- Search - probe table slots, until find key, or determine key is not there
- Delete - must be “lazy”
Probing Schemes

A hash function has the form:

\[ h(key, i) = (h_1(key) + F(i)) \mod \text{size} \]

\( F \) is the collision resolution function.
Typically \( F(0) = 0 \)

In Linear Probing \( F(i) = i \)
In Quadratic Probing \( F(i) = i^2 \)
In Double Hashing \( F(i) = ih_2(key) \)
Linear Probing

Probe $i$: $h(k, i) = (h_1(k) + i) \mod \text{size}$

- If the table is sparse – search likely to be short
- Once table starts filling – clustering occurs
  - Cluster: a sequence of adjacent filled cells
- When table is full – insert produces an infinite loop
- Primary clustering: items that hash to different cells probe the same alternative cells
- Clusters become targets for multiple collisions
- Clusters merge to form larger clusters
Quadratic Probing

Probe $i$: $h(k,i) = (h_1(k) + i^2) \mod \text{size}$

- Clustering is still possible
- But it is secondary – it happens only if $h(k,0) = h(k',0)$
Double Hashing

\[ h(k, i) = (h_1(k) + ih_2(k)) \mod \text{size} \]

**Probe Sequences:**
- \( h_1(k) \)
- \( h_1(k) + h_2(k) \mod \text{size} \)
- \( h_1(k) + 2h_2(k) \mod \text{size} \)
- \( h_1(k) + 3h_2(k) \mod \text{size} \)

Need to be careful about \( h_2(k) \):
- Not 0 and not a divisor of \( \text{size} \)
Probe Sequence

Consider the probe sequence:

\[ <h(k, 0), h(k, 1), \ldots, h(k, m-1)> \]

- What happens if it is not a permutation of \( <0, 1, \ldots, m-1> \)?
- How many different probe sequences are there with
  - Linear probing?
  - Quadratic probing?
  - Double hashing?
Rules of Thumb

- Separate chaining is simple but wastes space
- Linear Probing is fast when tables are sparse
- Double hashing is fast but needs careful implementation.

<table>
<thead>
<tr>
<th>load factor $\alpha$</th>
<th>linear probing</th>
<th>double hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>66%</td>
<td>2.0</td>
<td>1.6</td>
</tr>
<tr>
<td>75%</td>
<td>3.0</td>
<td>1.8</td>
</tr>
<tr>
<td>90%</td>
<td>5.5</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Number of probes
Universal Hashing

A given hash function may behave badly on a particular set of keys. Idea: Select a hash function at run-time, from a set of functions.

- A set $H = \{h_1, h_2, \ldots, h_k\}$ of hash functions $h_i: U \rightarrow \{0, \ldots, m-1\}$ is universal if for all $x, y \in U$:

$$\left| \{h \in H : h(x) = h(y)\} \right| = \frac{H}{m}$$

- So, if $h$ is chosen from $H$ at random, the chances of $x$ and $y$ colliding is $1/m$
Performance of Universal hashing

• **Theorem:** If \( h \in H \) (a universal set of hash functions), and \( n < m \), the average number of collisions involving a particular key \( x \) is less than 1.

• **Proof:**

\[
C_{xy} \equiv \begin{cases} 
1 & h(x) = h(y) \\
0 & \text{otherwise}
\end{cases}
\]

indicates whether \( x \) and \( y \) collide under hash function \( h \).

\[
C_x \equiv \left| \left\{ y \in U \mid y \neq x \land h(x) = h(y) \right\} \right| \text{ total number of collisions involving } x
\]

\[
\text{Average}(C_x) = \left( \frac{1}{m} \right)(n-1) = \frac{n-1}{m}
\]

\( T \) - the hash table

Since \( n < m \), we have that the average number of collisions with \( x \) is less than 1!
Handling load by Rehashing

• Build a bigger hash table of approximately twice the size when $\alpha$ exceeds a particular value
  – Go through old hash table, ignoring items marked deleted
  – Re-compute hash value for each non-deleted key and put the item in new position in new table (can’t just copy old table - why?)

• Running time is $O(n)$ but this happens infrequently (what is $n$ here?)
  – Not good for real-time safety-critical applications
Rehashing Example

- Open addressing, change from
  \[ h(k) = k \mod 5 \]
  \[ h'(k) = k \mod 11 \]
Rehashing Intuition

- Starting with table of size 2, double when load factor exceeds 1:
Amortized Analysis of Rehashing

• Cost of inserting $n$ keys is $< 3n$
• Choose $k$ such that $2^k + 1 \leq n \leq 2^{k+1}$
  - Cost of hashes = $n$
  - Cost of rehashes = $2+2^2+\ldots+2^k = 2^{k+1}−2$
  - Total Cost = $n+2^{k+1}−2 < 3n$