Lecture 7: February 28, 2002

Data structures:

- Hashing
Data structures

- Encapsulate data (e.g., a set of some elements)
- Supports various operations
  (e.g. INSERT, DELETE, SEARCH)
- Our focus: efficiency of the operations
  (linked list is not so great)
Dictionary data structure

- Maintain a dynamic set $T$ of elements

\[
\begin{array}{c}
\text{key}[x] \\
Satellite Data
\end{array}
\]

- Support dictionary operations:
  - $\text{INSERT}(T, x)$
  - $\text{SEARCH}(T, k)$
  - $\text{DELETE}(T, x)$

where $x$— > $(k, S)$ consists of key and satellite data.

- Examples:
  - Dictionary (word key to definition)
  - Compiler (symbol key to semantic data)
Direct-Address Table

- Idea:
  - Universe of keys is $U = \{0, 1, \ldots, m - 1\}$.
  - $K = \text{set of keys in use}$.
  - Define a direct-access table, $T[0..m - 1]$, where
    \[
    T[i] = \begin{cases} 
    x & \text{if } i \in K \text{ and } \text{key}[x] = i, \\
    \text{NIL} & \text{otherwise}.
    \end{cases}
    \]
Direct-Address Table Dictionary Operations

**DIRECT-ADDRESS-SEARCH**(\(T, k\))
\[
\text{return } T[k]
\]

**DIRECT-ADDRESS-INSERT**(\(T, x\))
\[
T[\text{key}[x]] \leftarrow x
\]

**DIRECT-ADDRESS-DELETE**(\(T, x\))
\[
T[\text{key}[x]] \leftarrow \text{NIL}
\]

- Only \(O(1)\) time required for each operation.
Direct-Address Table: Problems

- Range of keys usually large (e.g. ASCII strings).
- Space required for $T$ may be impractical.
- $|K|$ usually much smaller than $|U|$, so...

Wasteful to allocate space for every key in $U$. 
Hashing

- Fundamental in theory
- Crucial in practice (ask Chief Scientist of Yahoo!)
- Idea:
  - Use hash function $h$ to map $U$ into smaller set, $\{0, 1, \ldots, m - 1\}$.

  $$h : U \rightarrow \{0, 1, \ldots, m - 1\}$$

  - Can create hash table $T[0..m - 1]$, where

    $$T[i] = \begin{cases} 
    x & \text{if } key[x] \in K \text{ and } h(key[x]) = i, \\
    \text{NIL} & \text{otherwise}
    \end{cases}.$$
Hash Tables

\[ U \] (universe of keys)

\[ K \] (actual keys)

\[ k_1, k_2, k_3, k_4, k_5 \]

\[ T \]

\[ 0 \]

\[ h(k_1) \]

\[ h(k_4) \]

\[ h(k_2) = h(k_5) \]

\[ h(k_3) \]

\[ m - 1 \]
Collisions

- If some element already occupies slot to which an inserted element is mapped, a collision occurs.

- Must detect and resolve collisions!
**First method: Chaining**

- Each position in hash table is pointer to head of a linked list.
- To insert elements into the table, add to head of list.

\[ h(9) = h(52) = h(36) = i \]
Chaining Functions

• **Insertion**

\texttt{CHAINED-Hash-Insert}(T, x)

- insert \( x \) at the head of list \( T[h(key[x])] \)

Worst-case running time \( O(1) \).

• **Searching**

\texttt{CHAINED-Hash-Search}(T, k)

- search for an element with key \( k \) in list \( T[h(k)] \)

Worst-case running time proportional to length of list \( T[h(k)] \) (i.e., \( \Theta(n) \)).

• **Deletion**

\texttt{CHAINED-Hash-Delete}(T, x)

- delete \( x \) from the list \( T[h(key[x])] \)

Worst-case running time \( O(1) \) if doubly-linked lists used.
Analysis of Hashing with Chaining

• Assume each key equally likely to be hashed into any slot (simple uniform hashing)

• Given hash table $T$ with $m$ slots holding $n$ elements, define $T$’s load factor $\alpha$ as $n/m$

• Time for computing $h(k)$ is $\Theta(1)$.

• To find an element,
  
  – Look up its position in the table using $h$.
  – Search for element in linked list stored at slot.
Analysis Case 1: Unsuccessful Search

- Element for which we are searching is *not* in list.
- Must check each element in the list.
- Uniform hashing $\rightarrow$ average length of lists in $T = \alpha = n/m$.
- Expected number of elements examined $= \alpha$
- Running time: $\Theta(1 + \alpha)$. 
Case 2: Successful Search

- Assume CHAINED-HASH-INSERT adds new elements to the end of the list.
- Expected number of elements examined is at most 1 more than number of elements examined when sought-for element was inserted.
- Running time: $\Theta(1 + \alpha)$.
Method 2: Open Addressing

- All elements stored in hash table (i.e., no lists used).
- Each table entry contains either element or NIL.
- When searching for an element, systematically probe table slots

\[ h(k, 0), h(k, 1), h(k, 2) \ldots \]

until empty slot found, for

\[ h : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\} \]
Two views of hashing

- “Knuthology”: hash function fixed, random key set
- Universal hashing: hash function \textit{random}, worst-case key set [Carter-Wegman’79]
Knuthology 101

Ideally:

• Distribute keys uniformly into slots.
• Regularity in key distribution should not affect uniformity of hashing!
Division method

• Use hash function

\[ h(k) = k \mod m \]

• Must avoid certain values of \( m \)
  
  – Powers of 2. If \( m = 2^p \), \( h(k) \) is \( p \) lowest order bits of \( k \).
  
  – Powers of 10. If the keys are decimal numbers, hash function does not depend on all decimal digits of \( k \).

• Primes are usually good
Multiplication method

- Use hash function

\[ h(k) = \lfloor m (k \cdot A \mod 1) \rfloor \]

where \( A \) is a constant, \( 0 < A < 1 \).

- Value of \( m \) not critical; typically use \( m = 2^p \).

- Optimal choice of \( A \) depends on characteristics of data (Knuth says use \( A = \frac{\sqrt{5} - 1}{2} \))
Universal Hashing

• **Problem:** For any choice of hash function, there exists a bad set of identifiers—malicious adversary could force poor performance.

• **Solution:**
  
  – **RANDOMIZE!**
  
  – Choose hash function at random, 
    \emph{independent} of keys!
  
  – To do this, create a \emph{set} of hash functions, 
    \( \mathcal{H} \), from which \( h \) can be randomly selected!
Universal Hashing

- Let $\mathcal{H}$ be a collection of functions mapping $U$ to $\{0, 1, \ldots, m - 1\}$.
- **Definition:** $\mathcal{H}$ is universal if for all $x, y \in U$ ($x \neq y$),
  $$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}$$
Constructing a Universal Hash Function

One construction (CLRS, p. 251). Other constructions possible.

- Let $m$ be prime
- Assume $x \in \{0 \ldots m^{r+1} - 1\}$
- Decompose key $x$ into $r + 1$ digits, each with value $\{0, 1, \ldots, m - 1\}$, i.e.,
  \[ x = < x_0, x_1, \ldots, x_r >, \text{ where } 0 \leq x_i < m \]
- Pick $a = < a_0, a_1, \ldots, a_r >$, each $a_i \in \{0 \ldots m - 1\}$
- For each $a$ set
  \[ h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m \]

Deja-vu? Matrix product verification, via multiplication by random vector (PS2)
Universal Hashing Cont.

Theorem
\( \mathcal{H} \) is universal.

Proof

- Let \( x = \langle x_0, x_1, \ldots, x_r \rangle \) and \( y = \langle y_0, y_1, \ldots, y_r \rangle \) be distinct keys.
- \( x \) and \( y \) differ in at least one digit position.
- Without loss of generality, assume \( x_0 \neq y_0 \).
- Must show that probability that \( x, y \) collide is \( 1/m \)
Universal Hashing Cont.

• Show that for any choice of \(a_1, a_2, \ldots, a_r\) there is exactly one choice of \(a_0\) such that \(h_a(x) = h_a(y)\). I.e., there is a unique solution for \(a_0\) modulo \(m\):

\[
h_a(x) = h_a(y) \\
\sum_{i=0}^{r} a_i x_i = \sum_{i=0}^{r} a_i y_i \pmod{m} \\
a_0(x_0 - y_0) = \sum_{i=1}^{r} a_i (y_i - x_i) \pmod{m} \\
a_0 = (x_0 - y_0)^{-1} \sum_{i=1}^{r} a_i (y_i - x_i) \pmod{m}
\]

• \(a_0\) is uniformly chosen from \(\{0 \ldots m - 1\}\)

• probability of collision is \(1/m\).
Universal Hashing

Theorem
If $h$ is chosen randomly from $\mathcal{H}$ and used to hash $n$ keys into a table $T$ of size $m$, the expected number of collisions involving any particular key $x$ is less than $\alpha = n/m$.

Proof

- Let $C_x = \#$ of collisions of keys in $T$ with $x$
- Let

$$c_{yz} = \begin{cases} 1 & \text{if } h(y) = h(z) \\ 0 & \text{otherwise} \end{cases}$$

$$C_x = \sum_{y \in T \setminus \{x\}} c_{xy}$$
Universal Hashing

A single pair collides with probability $\frac{1}{m}$; 
That is, $E[c_{xy}] = 1/m$. Therefore,

$$E[C_x] = E \left[ \sum_{y \in T - \{x\}} c_{xy} \right]$$
$$= \sum_{y \in T - \{x\}} E[c_{xy}]$$
$$= \sum_{y \in T - \{x\}} \frac{1}{m}$$
$$= \frac{n - 1}{m}$$

$< \alpha$

So, the expected number of collisions with $x$ is $< \alpha$. 
A bonus “war story”

Once upon a time, when I was a Ph.D. student at Stanford, we were clustering web pages:

- web page → text summary → set of words
- web pages similar if their word sets have large intersection
- we were hashing words

Problem: the home page of colleague’s advisor got clustered with:

- assorted pornography
- web pages on alcohol abuse

Problem II: our algorithm was provably correct, i.e., probability of failure was one in a million (we calculated it exactly).
What happened?

- $x$ a word (really, word’s “signature”, but ignore that)
- We used hash function $h(x) = (ax \mod P) \mod 2^8$
  - $P = 2^{64} - 57$ (more or less)
  - $a$ randomly chosen
- For various reasons, $x$ divisible by 8 always (we were sampling 1 in 8 words)
- **Implementation bug:** forgot to use long long int $\Rightarrow$ $ax$ was computed modulo $2^{64}$ (rounding)
- mod $P$ had almost always no effect, since $P \approx 2^{64}$
- $x$ divisible by 8 $\Rightarrow$ $(ax)$ divisible by 8 $\Rightarrow$ $(ax) \mod 2^8$ divisible by 8
- 3 lowest bits of $h(x)$ were almost always 0, so the *actual* range size was $2^5$, not $2^8$
- Enough for unexpected word collisions to occur...

Moral: do your hashing well, or you might never graduate.