Overview

⇒ Hash Tables

• Chaining

• Hash Functions Chapter 11.3

• Open Addressing (Chapter 11.5)
Dictionary Operations

- Dictionary operations: insert, search, delete
  - Search means ...
    + Was the key inserted into the table?
      + Set membership
      + To find the value of satellite data associated with the key
  - Should be efficient. Hopefully $O(1)$
Direct-Address Tables

- When $|U| \sim |K|$

**DIRECT-ADDRESS-SEARCH**($T, k$)
1. return $T[k]

**DIRECT-ADDRESS-INSERT**($T, x$)
1. $T[x.key] = x$

**DIRECT-ADDRESS-DELETE**($T, x$)
1. $T[x.key] = \text{NIL}$

Each of these operations takes only $O(1)$ time.
Hash-Tables

- When \(|U| \gg |K|\)
  + Reduce storage requirements but still maintain \(O(1)\) access time
  + Terminology: \(k\) hashes to \(h(k)\)

\[ T \]

\(0\)

\(h(k_1)\)

\(h(k_4)\)

\(h(k_2) = h(k_5)\)

\(h(k_3)\)

\(m-1\)

\(U\) (universe of keys)

\(K\) (actual keys)

\(k_1\)

\(k_2\)

\(k_3\)

\(k_4\)

\(k_5\)
Collisions

• Two keys might hash to the same value
  - Collision
  - Can happen since size of universe $\gg$ size of hash table

• Try to avoid collisions as much as possible
  - Hash function is deterministic: $h(k)$ is always same value
  - Will hopefully map keys randomly across the hash table
Overview

• Hash Tables
⇒ Chaining
• Hash Functions Chapter 11.3
• Open Addressing (Chapter 11.5)
Collision Resolution by Chaining
Efficiency of Dictionary Operations

- **Insert(T,x)**
  - Insert x at the head of list $T[h(x.key)]$
  - If we do not check if x.key is already in list: $O(1)$

- **Search(T,k)**
  - Search for an element with key $k$ in list $T[h(k)]$
  - Worst case: proportional to length of list

- **Delete(T,x)**
  - Delete x from list $T[h(x.key)]$
  - $O(1)$ time since we already have pointer to element and if doubly-linked
Analysis of Hashing with Chaining

- Load factor $\alpha$: $n/m$ where $n$ is keys stored, and $m$ is size of table
- Worse-case for search is $\Theta(n)$
  - Same as using one linked list for all elements
- Average-case performance of hashing depends on how well $h$ distributes keys among $m$ slots on average
  - Let $n_j$ be length of list $T[j]$ for $0 \leq j \leq m$
  - So $\sum_{i=0}^{m-1} n_i = n$
  - So $E[n_j] = \alpha = n/m$
  - But what is the expected value for hash values that are used?
    + If keys not distributed randomly, could still be all in one hash value
      so search takes $\Theta(n)$ time
Analysis of Search

• Assume hash of key is independent of keys already inserted
  - Referred to as simple uniform hashing assumption

**Theorem 11.1**
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1 + \alpha)$, under simple uniform hashing assumption.

**Proof:**
Under assumption of simply uniform hashing, any key $k$ not already stored in table is equally likely to hash to any of the $m$ slots.

Expected time to search unsuccessfully for $k$ is expected time to search to end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$.

Thus, expected number of elements examined in an unsuccessful search is $\alpha$, and total time including computing $h(k)$ is $\Theta(1 + \alpha)$.
Successful Search

**Theorem 11.2**
In a hash table in which collisions are resolved by chaining, a **successful** search takes average-case time $\Theta(1 + \alpha)$, under simple uniform hashing.

**Proof:**
We assume element being searched $x$ for is equally likely to be any of the $n$ elements stored in the table.

How many other elements are in the same list?

How far $x$ is from front of list?

Textbook gives a big derivation, but under simple uniform hashing works out to be $\Theta(1 + \alpha)$
Implications

• Load factor $\alpha$: $n/m$ where $n$ is keys stored, and $m$ is size of table
• Search time depends on load factor
  - If number of keys used is proportional to size of table
    + Search is $O(1)$ time
Overview

• Hash Tables
• Chaining
⇒ Hash Functions Chapter 11.3
• Open Addressing (Chapter 11.5)
What Makes a Good Hash Function?

• Should satisfy assumption of simple uniform hashing
  - Each key is equally likely to hash to any of the $m$ slots regardless of what the other keys have hashed to
  - But, rarely know the probability distribution from which keys are drawn

• Domain knowledge might help in designing hash function
  - If we know keys are random real numbers $k$ independently and uniformly distributed in range $0 \leq k < l$
    + Good hash function?
  - Hashing English words: be careful of ‘hat’ and ‘hats’
    + Don’t use suffix ... or prefix

• Good approach
  - Make hash function independent of any patterns that might exist in data
Division Method

• Assume keys are in the set of natural numbers
  - Otherwise find way to map them to numbers
• Take remainder of $k$ modulo $m$: $h(k) = k \mod m$
  - Allows you to map onto all of the slots in the table
  - Seems that we pick size of table so that it works well for hashing
• Do not use $m$ as a power of 2
  - Otherwise just using the lowest-order bits of $k$
  - Make it depend on all of the bits of the key
  - Even using $m = 2^p - 1$ is problematic (see textbooks)
  - Prime not to close to a power of 2 seems to work out well
  - If $n = 2000$, and $\alpha = 3$ seems reasonable, can pick $m = 701$ since it is a prime near $2000/3$ but not near any power of 2
Multiplication Method

• $h(k) = \lfloor m(kA \mod 1) \rfloor$
  - First multiply key $k$ by a constant $A$ in the range $0 < A < 1$ and extract the fraction part of $kA$
  - Then multiply it by $m$ and take the floor of the result

• Advantage
  - Reduces dependency on $m$
Multiplication Method: Typical approach

- $h(k) = \lfloor m(kA \mod 1) \rfloor$
  - A constant: $0 < A < 1$  \[ m \text{ is size of table} \]

- Choose $m$ be power of 2 ($m = 2^p$)
  - Suppose word size of machine is $w$ bits and $k$ fits into a single word
  - Restrict $A = s/2^w$ where $s$ is integer $0 < s < 2^w$ (so $s = A \times 2^w$)
  - First multiply $k$ by $w$-bit integer $s$
  - Result is $2w$ bits long with value $r_12^w + r_0$
  - Hash value is $p$ most significant bits of $r_0$
Example

• Some values of $A$ work better than others
  - Knuth suggests $A \approx (\sqrt{5} - 1)/2 = 0.6180339887$

• Example
  \[
  k = 123456 \\
  p = 14 \\
  m = 2^{14} = 16384 \\
  w = 32 \\
  \]
  set $A = s/2^{32}$ closes to Knuth’s suggestion: $A = 2654435769/2^{32}$
  \[
  k \times s = 327706022297664 = 76300 \times 2^{32} + 17612864 \\
  \]
  so $r_1 = 76300$ and $r_0 = 17612864$
  most 14 significant bits of $r_0$ yield $h(k) = 67$
Overview

- Hash Tables
- Chaining
- Hash Functions Chapter 11.3

⇒ Open Addressing (Chapter 11.5)
Open Addressing

• Do not use chaining to a linked-list for collisions
• Each table entry contains either an element of dynamic set or Nil
• When searching, systematically examine table slots until
  - find desired element
  - ascertain element is not in table
• Hash table can fill up
  - Load factor $\alpha$ cannot exceed 1
• Successively **probe** table until you find an empty slot to put the key

• Rather than probe starting at 0 (would require $\Theta(n)$)
  - Sequence of probes depends on key being inserted
  - Function takes inputs key and probe number
    $h: U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$
  - Probe sequence for each key should be permutation of $\langle 0, 1, \ldots, m - 1 \rangle$

```plaintext
FUNCTION Hash-Insert(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j] == NIL
            T[j] = k
            return j
        else
            i = i + 1
    until i == m
    error "hash table overflow"
```

© P. Heeman, 2017
Deletion

- Deletion from an open address table is difficult
  - If you delete a key from slot $i$ by changing its entry to Nil
    Won’t be able to find any key $k$ during whose insertion we had probed
    slot $i$ and found it occupied

- Can add a special value ‘Deleted’
  - When searching, viewed as having a value
  - When inserting, viewed as nil

- Search times no longer depend on load factor $\alpha$
  - Open addressing not commonly used when deletion is needed

- Any advantage of chaining with a linked-list?
Uniform Hashing

- Uniform Hashing
  - Generalizes simple uniform hashing
  - Probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m - 1 \rangle$
  - Difficult to implement, usually approximated
    - Do guarantee that each table entry is included
Linear Probing

• Let $h' : U \rightarrow \{0, 1, \ldots, m - 1\}$ be an ordinary hash function
  - Referred to as auxiliary hash function
  - Hash function: $h(k, i) = (h'(k) + i) \mod m$
  - Initial probe determines sequence: only $m$ distinct prob sequences

• Primary Clustering
  - If there are $i$ slots filled in a role, odds are $i/m$ that hash function will
do initial hash to it, and cause cluster to grow by one
  - Will increase search times
Quadratic Probing

\[ h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \]

where \( h' \) is an auxiliary hash function
\( c_1 \) and \( c_2 \) are positive auxiliary constants

- Later positions are offset by amounts that depend on a quadratic (not linear) manner on the probe number \( i \)
- Much better performance than linear probing
  - To make full use of hash table, values for \( c_1, c_2, m \) are constrained
  - If initial probe is the same, so are all subsequent ones
    - Can lead to secondary clustering

- Still only \( m \) different probe sequences
Double Hashing

\[ h(k, i) = (h_1(k) + ih_2(k)) \mod m \]

• Initial probe depends on \( h_1 \)
• Successive probes are offset by \( h_2 \)
  - Now keys with same initial probe will not follow same probe sequence
• \( h_2(k) \) must be relatively prime with the hash-table size \( m \) for entire hash table to be search
  - Let \( m \) be a power of 2 and make \( h_2 \) always return odd numbers
  - Let \( m \) be prime and \( h_2 \) return a positive number less than \( m \)
  - Either approach gives \( \Theta(m^2) \) probe sequences
    + We can use \( \Theta, O \) and \( \Omega \) for any asymptotic analysis
Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$ (and no deletions) and uniform hashing assumption, expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$.

**Intuition**
Always make a first probe: 1
Make a second probe if first probe is unsuccessful $\alpha$
Make a third probe? $\alpha \times \alpha$
Make a fourth probe? $\alpha^3$
$\Sigma_{i=0}^{\infty} \alpha^i = 1/(1 - \alpha)$
If load factor is .9, number of probes is 10.
For chaining, $1 + \alpha$