Day 8—Hash Tables
Universal Hashing

Neil Rhodes
CSE 202
UC San Diego
Dictionary

Data structure storing elements containing keys (and possibly related data)

Operations:
- Insert(e): Insert element e
- Delete(key): Delete element with matching key
- Search(key): Returns element with matching key, or NIL

For simplicity, assume no elements with duplicate keys in the dictionary
Hash Table Implementation

U: universe of possible keys
m: indices 0..m-1 into an array T

Hash function: any function, h, maps from U to {0..m-1}

Collision: when two keys, k≠l, map to the same index (h(k)=h(l))

Collision resolution:
- Chaining (linked list of elements that collide)
- Open Addressing (store colliding elements in the table)
  - Linear probing: Sequence 1..i of indices: (h(k) + i) mod m
  - Quadratic probing: sequence 1..i of indices: (h(k) + c1i + c2i2) mod m
  - Double-hashing: sequence: (h1(k) + ih2(k)) mod m

Load factor α: number of elements in table / m
Hash Table with Chaining

U: 1-100

m = 5

\( h(x) = x \mod m \) (maps U to \{0..m-1\})

\( \alpha = \frac{5}{5} = 1 \)

\begin{align*}
T & \rightarrow 25 \rightarrow 10 \\
& \rightarrow 96 \\
& \rightarrow 54 \rightarrow 74
\end{align*}
Expected time

Given sequence of n requests, \( r_1 \ldots r_n \), on an empty dictionary

- Each request is Insert, Search, or Delete

Let \( k_i \) be the key involved in \( r_i \) and \( b_i = h(k_i) \)

- \( T(request_i) \leq d(1 + \text{number of duplicates of } b_i \text{ in the table}) \)
  \( \leq c(\text{number of } r_j \text{ such that } b_j = b_i \text{ and } j < i) \)
  \( \leq c(\text{number of } r_j \text{ such that } b_i = b_j) \)
  - Overcounts when \( j \) isn’t an insert
  - Overcounts when element \( i \) is deleted before \( r_i \)

- Define indicator random variable \( X_{ij} = 1 \) if \( b_i = b_j \) and \( 0 \) otherwise

- \( T(r_i) \leq c \sum_j X_{ij} \)

- \( E[T(r_i)] \leq E[c \sum_j X_{ij}] = c \sum_j E[X_{ij}] \)

- If we know \( E[X_{ij}] = 1/n \), then \( E[T(r_i)] = O(1) \)

Size of hash table

- \( m \gg n \) wastes lots of space
- \( m \ll n \) makes lots of collisions
- Conclusions, let \( m = n \) (approximate number of items in the table)
When is $E[X_{ij}] = 1/n$?

**Assume keys are uniformly distributed**

- If we ensure that $h$ maps the same number of keys to each index, then the indices will also be uniformly distributed
  - mod, for example
- However, this is not always a reasonable assumption
When is $E[X_{ij}] = 1/n$?

Assume indices are uniformly distributed

- Assume the hash function acts like a random number generator
- For example, use a multiplicative hash:
  - $0 < A < 1$
  - $h(k) = \text{floor}(m ((kA) \mod 1))$
    - Multiply fractional part of $kA$ by $m$
  - Knuth suggests $A = 0.6180339887$ (inverse of the golden ratio)

- Will work well for most inputs, but not all
Randomized algorithm (Universal hashing)

A universal set of hash functions: set $H$ mapping from $U$ to $\{0..m-1\}$ such that, for all $x \neq y$ in $U$:

- the fraction of $H$ such that $h(x) = h(y) \leq 1/m$

Exactly what we need to get probabilistic time $O(1)$ for Insert, Delete, and Search

CLRS describes the following universal set:

- $p =$ prime such that all keys are in $\{0..p-1\}$
- $\mathbb{Z}_p = \{0, 1, .. p-1\}$, $\mathbb{Z}_p^* = \{1, .., p-1\}$
- $h_{a,b}(k) = ((ak + b) \mod p) \mod m$ where $a$ in $\mathbb{Z}_p^*$ and $b$ in $\mathbb{Z}_p$

- $p(p-1)$ different hash functions
Randomized algorithm (Universal hashing)

Claim: Given $k \neq l$ in $\mathbb{Z}_p$, each different hash function produces a different pair $(ak+b \mod p, al+b \mod p)$.

- Proof: Given any $(q, r)$ in $\mathbb{Z}_p \times \mathbb{Z}_p$ ($q \neq r$), there exist an $(a, b)$ (such that
  - $q = (ak+b) \mod p$
  - $r = (al+b) \mod p$
  - $b = (al-r) \mod p$
  - $q = (ak+al-r) \mod p = (a(k-l) - r) \mod p$
  - $q-r = a(k-l) \mod p$ \hspace{1cm} ($k \neq l$, $q \neq r$, $a \neq 0$)
  - $1(k-l), 2(k-l), 3(k-l)$ take on all different values of $1..p-1$
  - Given $a$, can solve for $b$

Same number of hash functions as pairs, so given $k, l$ and random $a, b$, $(ak+b \mod p, al+b \mod p)$ is randomly chosen from distinct pairs of values $\mod p$.

$Pr\{\text{random distinct pairs of values } \mod p \text{ are equal } \mod m\} = 1/m.$
Hashing summary

Hash tables with chaining have average-time $O(1)$, worst-case $O(n)$ time to handle: Insert, Delete, Find

Universal hashing yields:
- Each time you create a hash table, after the input has been specified, you choose a random function from the universal set.
- Then, the expected time for each operation is $O(1)$
- There are no bad inputs (since the random function foils adversaries)
- However, in practice, the hash function is chosen before the input has been seen
  - An adversary who “knows” the hash function can choose input causing expected time $> O(1)$

- See Denial of Service via Algorithmic Complexity Attacks <http://www.cs.rice.edu/~scrosby/hash/> for good explanation of why universal hashing is necessary
Amortized analysis
Chapter 17
Amortized Analysis

**Amortized analysis:** Produces an average cost of a sequence of operations $1..i$

- Bounds worst-case total cost of a sequence of operations
- Not like average-case analysis, which uses probability

**Three major approaches**

- **Aggregate analysis** (upper-bound $T(n)$ for a sequence of $n$ operations)
- **Accounting method** (amortized cost for each operation)
  - Overcharges for some operations, storing credits in data structure objects
  - Uses credits to help pay for expensive operations
- **Potential method**
  - Define potential of the data structure as a whole. Each operation is charged real cost + change in potential

**Can't run a deficit**

- Never allowed to get behind. For all $j$, $T_{\text{real}}(\text{first } j \text{ operations}) \leq T_{\text{amortized}}(\text{first } j \text{ operations})$
Amortized analysis

Example

- Stack with push(e), pop() and multipop(k) (pops k elements at once)
- Aggregate analysis
  - Any multipop(k) requires k previous pushes. For n operations, the worst-case is n-1 pushes O(n) followed by a single multipop(n-1) O(n). Total time is O(n). Per-operation = O(n)/n
- Accounting method
  - Push puts a credit on the item pushed. Amortized cost = 1 + 1 = 2
  - Pop uses the credit already on the item. Amortized cost=0
  - Multipop uses the credit on each item. Amortized cost = 0
  - Note that total credit is never negative (otherwise, we’d have run a deficit)
- Potential method
  - Define \( P(sta\text{ck}_i) \) = number of items on stack at time \( i \)
  - \( P(sta\text{ck}_0) = 0 \) (Important. Also, \( P \) never negative)
  - Push: amortized cost = cost - \( P(sta\text{ck}_{i-1}) + P(sta\text{ck}_i) = 2 \)
  - Pop, Multipop: amortized cost = cost - \( P(sta\text{ck}_{i-1}) + P(sta\text{ck}_i) = 0 \)
Hash table: What if you don’t know the number of elements?

Normally, set m=approximate number of elements

What to do if you don’t know ahead of time?

- Resize the table when it gets full.
- How?
  - Allocate new table twice as big: $O(n)$
  - Put all elements from old table in new table $O(n)$

- Cost of some inserts is $O(n)$, most are still $O(1)$. What is the amortized cost? Let’s use accounting method
  - Insert costs 3. One pays for inserting the item. One is placed on the item itself to pay for a future resize. One is placed on an item already in the table that needs a credit
  - When an insert takes place causing a split, every item in the table will have a credit and we’ll have a new table half-full with no credits. Subsequent inserts will add credits to each of those elements.
  - Shrink table when load factor goes below .25
    - Allocate new table half as big and copy elements. $O(n)$
Resizing hash table

Delete

- Accounting: Charge delete 2 credits. One pays for the delete. One is placed on one of the $n/4..n/2$ elements remaining in the table. By the time we’re down to $n/4$ elements, each has a credit on them, which can be used to pay for the resize.
Perfect Hashing
Perfect hashing

**Definition:** Worst-case search: $O(1)$

**Context:** Fixed set of keys, known in advance

**Idea:** Very similar to bucket sort
- Choose pretty-good function $h$, from universal set of hash functions.
- For each set, $S_j$, of colliding keys hashing to index $j$ create a secondary hash table of size $|S_j|^2$. Choose a perfect function, $h_j$, from universal set of hash functions that has no collisions.

**Notes:**
- Size is still $O(n)$ because $h$ will, like bucket sort, cause the sum of the sizes to be $O(n)$.
- Probability of $h$, chosen at random, satisfying requirement, is $\geq 1/2$.
- Probability of $h_j$, chosen at random, satisfying its requirement, is $>1/2$. 
Perfect hashing

If we have a hash table of size $n^2$ and $n$ elements, with $h$ chosen randomly from a universal set of hash functions, $\Pr\{\text{collisions}\} < 1/2$

- $n(n-1)/2$ pairs that may collide. $\Pr\{\text{pair collides}\} = 1/n^2$
- $E[\text{number collisions}] = n(n-1)/2 \times 1/n^2 < 1/2$
- $\Pr\{\text{number collisions} \geq 1\} \leq E[\text{number collisions}]/1$ (by C.29)
  < 1/2

Given $h$ chosen randomly from a universal set of hash functions, $\Pr\{\text{sum of } |S_j|^2 \geq 4n\} \leq 1/2$

- $E[|S_j|^2] = 2 - 1/n$ (refer back to bucket sort slides)
  $E[\text{Sum of } |S_j|^2] < 2n$
  $\Pr\{\text{Sum of } |S_j|^2 > 4n\} \leq 2n/4n = 1/2$