Object-Oriented Programming and Data Structures

COMP2012: Hashing

Cecia Chan
Brian Mak
Dimitris Papadopoulos

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Motivation

- How would you find a student record given just the students name?
- How does an electronic dictionary look up for a word, say, “computer”?
- Each machine has an IP address in the Internet. How will an internet service look up an IPv6 address?

Part I

Hashing

= 79054025
255fb1a2
6e4bc422
aef54eb4

General Idea

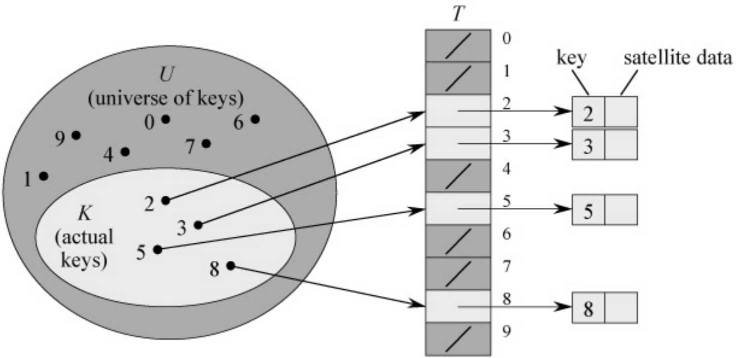
0	
1	
2	data item
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

- A hash table is an array of some fixed size, containing all the data items.
- Each item has a key; search is performed based on the keys.
- Each key is mapped into some position in the array in the range 0 to $m - 1$, where m is the array size.
- The mapping is called hash function.
- Example applications:
  - Compilers use hash tables, called symbol tables, to keep track of declared identifiers in a program.
  - On-line spell checkers: After hashing the entire dictionary, one can check each word in constant time and print out the mis-spelled words in order of their appearance in the document.

Hash Table

- Hash table is a data structure that supports: search, insertion, and deletion (deletion may be unnecessary in some applications).
- The implementation of hash tables is called hashing.
- Hashing is a technique which allows the executions of above operations in constant time on average.
- Unlike other data structures such as linked lists or binary trees, data items are generally not ordered in hash tables.
- As a consequence, hash tables don't support the following operations
  - find_min and find_max
  - finding successor and predecessor
  - reporting data within a given range
  - listing out the data in order

Unrealistic Solution: 1 Slot for 1 Key



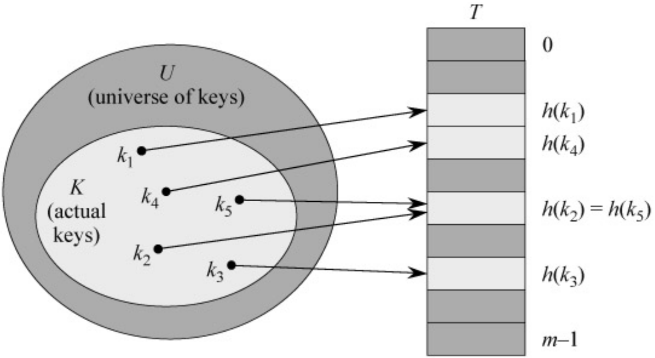
- Universe of keys U is the set of all possible values of the keys.
- Each position, also called a slot, in the hash table T corresponds to a key in U.
  - $T[k]$ corresponds to a data item with key k.
  - If the table contains no data with key k, then $T[k] = \text{nil}$.

Unrealistic Solution ..

- Insertion, deletion and search all take O(1) constant time.
- Problem: it wastes too much space if the universe of keys is large compared with the actual number of data to be stored.

E.g., in HKUST, student IDs are 8-digit integers. So the key universe has a size of 10^8, but we only have ∼7000 students (not counting the alumni)!

Hash Function



- Hash function, h maps the universe of keys U into the slots of a hash table $T[0, 1, ..., m-1]$.
- Several keys may be mapped to the same slot.

Hash Function ..

possible keys	hash function	hash table
$\{k_0, k_1, \ldots, k_{\|U\|-1}\}$	$\xrightarrow{h}$	$\{0, 1, \ldots, m-1\}$

- Usually, $m \ll |U|$.
- The keys k_i are assumed to be natural numbers.
- If they are not, they can always be converted or interpreted in natural numbers.
- $h(k_i)$ = an integer in $[0, \ldots, m-1]$ is called the hash value of k_i.
- With hashing, an item of key k is stored in $T[h(k)]$.

- Two keys may be hashed to the same slot.
- Question: Can we ensure that any two distinct keys are hashed to different slots?
- No! If $N > m$, where
  - m = size of the hash table, and
  - N = number of data

Solution:

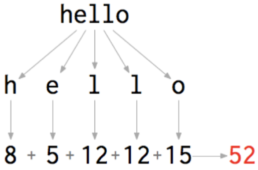
1. Design a good hash function that is
   - fast to compute, and
   - can minimize the number of collisions
2. Design a method to resolve the collisions when they occur.

Hash Function Design

- A simple and reasonable strategy: $h(k) = k \mod m$.
- e.g. $m = 12, k = 100, h(k) = 4$
- It requires only a single division operation (quite fast).
- Certain values of m should be avoided: e.g., if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k; thus, the hash function does not depend on all the bits.
- Similarly, if the keys are decimal numbers, m should not be set to be a power of 10.
- It's a good practice to set the table size m to a prime number.
- Good values for m: primes not too close to exact powers of 2
  - e.g., for a hash table to hold 2,000 numbers, if we don't mind an average of 3 numbers being hashed to the same slot, choose $m = 701$.

How to Deal with String Keys: Method 1

Add up the ASCII values of the characters in the string. (For simplicity, we only use their positions in the alphabets here.)



- Most hash functions assume that keys are natural numbers. If keys are not natural numbers, a way must be found to interpret them as natural numbers.
- Problems:
  - Different permutations of the same set of characters would have the same hash value.
  - If the table size is large, the keys do not distribute well.
  - e.g., $m = 10{,}007$ (a prime number) and all string keys have ≤ 8 characters. Since ASCII values ≤ 127, their numeric keys are in the range between 0 and $127 \times 8 = 1,016$.

How to Deal with String Keys: Method 2

$$h(key) = \big(key[0] + 27 \cdot key[1] + 27^2 \cdot key[2]\big) \mod m$$

where m is hash table size.

- If the first 3 characters are random and the table size m is 10,007, then it is a reasonably equitable distribution.
- Problems:
  - letters in an English word are not random;
  - according to some dictionary, there are only 2,851 different combinations for the first 3 letters of English words;
  - therefore, only at most 28% of the table can actually be hashed to (if $m = 10{,}007$).

How to Deal with String Keys: Method 3

$$h(key) = \left(\sum_{i=0}^{L-1} 37^{(L-1-i)} \cdot key[i] \right) \bmod m$$

$$= \left(37^{(L-1)} \cdot key[0] + 37^{(L-2)} \cdot key[1] + \ldots + key[L-1] \right) \bmod m$$

where L is the length of a key.

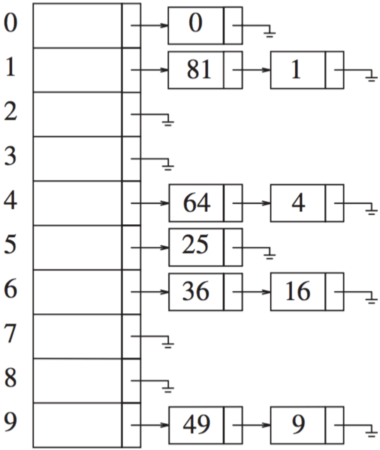
- This hash function involves all characters in the key and the hash values are expected to distribute well.

Part II

Collision Handling

Separate Chaining

- Keys: the set of squared numbers $\{1, 4, 9, 16, \ldots\}$.
- Hash function: $h(k) = k \mod 10$.
- Using the idea of equivalence classes.



- The hash table is more than a simple array, but a table of linked lists.
- Keys having the same hash values are chained on a separate linked list.

Separate Chaining Operations

To insert a key k:

- Compute $h(k)$.
- If $T[h(k)]$ contains a null pointer, the list (or chain) is empty. Initialize this table entry to point to a linked list with a single node containing k alone.
- If $T[h(k)]$ points to a non-empty list, add k to the beginning of the list.

To delete a key k

- Compute $h(k)$ to determine which list to traverse.
- Search for the key k in the list that $T[h(k)]$ points to.
- Delete the item with key k if it is found.

Separate Chaining Features

- If the hash function works well, the number of keys in each linked list will be a small constant.
- Therefore, we expect that each search, insertion, and deletion can be done in constant time.
- Disadvantage: Memory allocations and de-allocations in linked list manipulations slow down the operations.
- Advantage: deletion is easy.

Open Addressing

- Instead of putting keys of the same hash value into a chain, open addressing will relocate the key k to be inserted if it collides with an existing key.
- Open addressing needs to determine the sequence of slots to be examined for key relocation.
- Key k may be stored at an entry different from $T[h(k)]$.
- Two issues arise:
  - what is the relocation scheme?
  - how to search for k later?
- Three common methods for resolving collisions in open addressing
  1. linear probing
  2. quadratic probing
  3. double hashing

Basic Strategy of Open Addressing

- To insert a key k, compute $h_0(k)$.
- If $T[h_0(k)]$ is empty, insert it there.
- If collision occurs, probe alternative cell in the following order: $h_1(k), h_2(k), \ldots$, until an empty cell is found.
- $h_i(k) = (\mathrm{hash}(k) + f(i)) \mod m$, where the function f determines the collision resolution strategy and $f(0) = 0$.
- Different open addressing methods differ in the definition of the function $f(\,)$.

Linear Probing: Insertion

$$f(i) = i$$
$$h_i(k) = (\text{hash}(k) + i) \mod m$$

- Basic strategy: Table cells are probed sequentially (with wrap-around) until an empty slot is found.
- Again let m be the table size and N be the number of items.
- Let k be the new key to be inserted; compute $hash(k)$.
- For $i = 0$ to $m - 1$, compute $j = (\text{hash}(k) + i) \mod m$.
- If $T[j]$ is empty, then we put k there and stop.
- If no empty slot can be found to put k, the table is full; report an error.

Linear Probing: Example

- $\text{hash}(k) = k \mod 10$
- Insert the following keys: 89, 18, 49, 58, 69
- To insert 58, probe $T[8]$, $T[9]$, $T[0]$, $T[1]$
- To insert 69, probe $T[9]$, $T[0]$, $T[1]$, $T[2]$

Table Index	Insert 89	Insert 18	Insert 49	Insert 58	Insert 69
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					

Primary Clustering

- A block of contiguously occupied table entries is a cluster.
- On average, when we insert a new key k, we may hit the middle of a cluster. Therefore, the time to insert k would be proportional to half the size of a cluster.
  $\Rightarrow$ the larger the cluster, the slower the performance.
- Linear probing has the following disadvantages:
  - Once $h(k)$ falls into a cluster, this cluster will definitely grow in size by one. Thus, this may worsen the performance of insertions in the future.
  - If two clusters are separated by only one entry, they may be merged together by an insertion to that entry.
    $\Rightarrow$ cluster size can increase drastically by 1 insertion.
  - This means that the performance of insertion can deteriorate drastically after a single insertion.
  - Large clusters are easy targets for collisions.

Quadratic Probing

$$f(i) = i^2$$
$$h_i(k) = (\mathrm{hash}(k) + i^2) \mod m$$

Table Index	Insert 89	Insert 18	Insert 49	Insert 58	Insert 69
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					

Quadratic Probing: Example

- Example:
  - $\text{hash}(k) = k \mod 10$
  - Insert the following keys: 89, 18, 49, 58, 69
  - To insert 58, probe $T[8], T[9], T[(8+4) \mod 10]$
  - To insert 69, probe $T[9], T[(9+1) \mod 10]$, $T[(9+4) \mod 10]$
- Two keys with different home positions will have different probe sequences. E.g.,
  - $m = 101, h(k_1) = 30, h(k_2) = 29$
  - probe sequence for k_1: $30, 30+1, 30+4, 30+9$
  - probe sequence for k_2: $29, 29+1, 29+4, 29+9$
- If the table size is prime, then a new key can always be inserted if the table is at least half empty (see proof in the reference book by Weiss).

Secondary Clustering

- Keys that hashed to the same home position will probe the same alternative cells.
- Simulation results suggest that it generally causes less than an extra half probe per search.
- To avoid secondary clustering, the probe sequence need to be a function of the original key value, not the home position.

Double Hashing

- To alleviate the problem of clustering, the sequence of probes for a key should be independent of its primary home position.
- Thus, use two hash functions: hash( ) and hash2( ).

$$
\begin{aligned}
f(i) &= i \times \mathrm{hash}_2(k) \\
h_i(k) &= (\mathrm{hash}(k) + i \times \mathrm{hash}_2(k)) \mod m
\end{aligned}
$$

- e.g., $\mathrm{hash}_2(k) = R - (k \mod R)$, where $R < m$ and R is prime.

Double Hashing: Example

- $h_i(k) = (\text{hash}(k) + i \times \text{hash}_2(k)) \mod m$
- $m = 10, R = 7, \text{hash}(k) = k \mod 10, \text{hash}_2(k) = 7 - (k \mod 7)$
- Insert the following keys: 89, 18, 49, 58, 69
- 2nd probe for 49: $T[(9+7) \mod 10]$; 58: $T[(8+5) \mod 10]$;
- 2nd probe for 69: $T[(9+1) \mod 10]$

Table Index	Insert 89	Insert 18	Insert 49	Insert 58	Insert 69
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					

Double Hashing: Choice of $\mathrm{hash}_2(\,)$

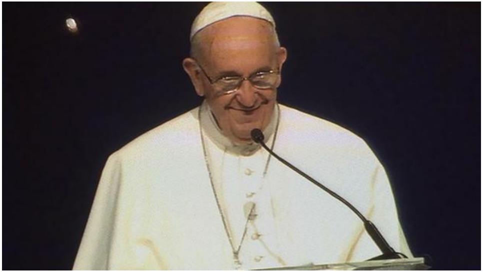
- $\mathrm{hash}_2(\,)$ must never evaluate to zero.
- For any key k, $\mathrm{hash}_2(k)$ must be relatively prime to the table size m. Otherwise, we will only be able to examine a fraction of the table entries.
- E.g., if $\mathrm{hash}(k) = 0$ and $\mathrm{hash}_2(k) = m/2$, then we can only examine the entries $T[0]$, $T[m/2]$, and nothing else!
- One solution is to make m prime, and choose R to be a prime number smaller than m, and set

$$\mathrm{hash}_2(k) = R - (k \bmod R)$$

- Quadratic probing, however, does not require the use of a second hash function, and thus is likely to be simpler and faster in practice.

Part III

Final Remarks



Deletion in Open Addressing

- Actual deletion cannot be performed in open addressing hash tables, otherwise the probing sequence will be broken.
- Solution: Add an extra field to each table entry, and mark it as
  - EMPTY
  - ACTIVE
  - DELETED
- It is also called lazy deletion.

Re-hashing

- Load factor $\alpha = N/m$, where N is the number of actually hashed items in the hash table.
- The operations in a hash table will become slower drastically when α becomes large.
- When α becomes large, (roughly) double the table size and re-hash all data items with a new hash function.
- Obviously, re-hashing is a very expensive operation. Fortunately, it usually happens infrequently in a well-designed hash table.