Hashing

19.10.2023

Dictionary (Map) ADT

Definition

- A dictionary is an ADT composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.
- **Idea:** use the key as index information to reach the value - key and value mapping

key	value
hello	hola
red	roja
blue	azul

domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

Dictionary (Map) ADT

Where do we use it?

- Symbol tables for a compiler: name of the variable and its value
- Customer records (access by name)
- Games (positions, configurations)
- Spell checkers, etc.

Operations

- Three basic operations:
  - find: Look up the existence of key value pair.
  - insert: Insert key value pair
  - remove: Delete key value pair
- Note that there are no operations that require ordering information.

How to Implement a Dictionary?

We can use:

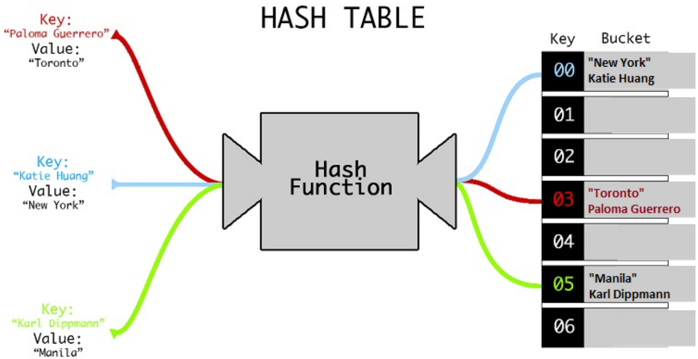
- Lists
- Binary Search Trees
- Hash tables

Hash Table

- An important and widely used technique for implementing dictionaries
- A fixed size array containing the items.
- Operations are performed based on the value of the key, i.e. use the key value as an index.
- Constant time per operation (on the average): If you know the index of an element, you can access it in a step (shorter than N or $logN$)
- instead of using the key as an array index directly, the array index is computed from the key.

Basic Idea

- Use the hash function to map keys into positions (or indexes) in a hash table.
    - If element e has key k and h is the hash function, then e is stored in position $h(k)$ of the table
    - To search for e, compute $h(k)$ to locate the position. If no element has key k, dictionary does not contain e.

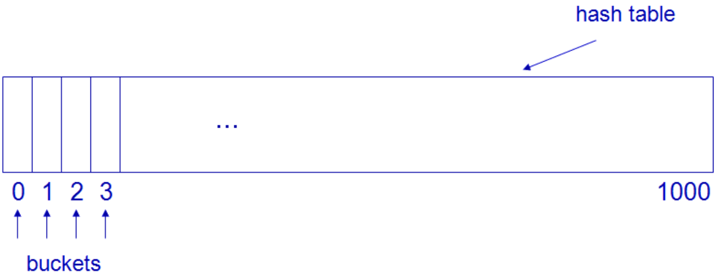


SHA Hash Functions

```
SHA224("The quick brown fox jumps over the lazy dog")
0x 730e109bd7a8a32b1cb9d9a09aa2325d2430587ddbc0c38bad911525
SHA224("The quick brown fox jumps over the lazy dog.")
0x 619cba8e8e05826e9b8c519c0a5c68f4fb653e8a3d8aa04bb2c8cd4c
```

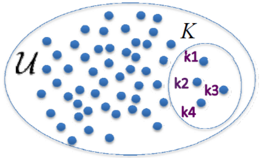
An Ideal Example

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 1001 students
  - Hash function: $h(k) = k - 951000$ maps ID into distinct table positions 0-1000
  - `array table[1001]`



An Ideal Example

- $O(1)$ time to perform `insert, remove, findItem`
- Such an ideal case is not realistic since many applications have key ranges that are too large to have one-to-one mapping between table cells and keys
  - Huge universe U of possible keys, but only N keys actually present (26 letters in English but not all letter combinations make up a word).
  - Hashing exploits sparsity of space
- or finding a suitable hash function is not possible



$\mathcal{U}$: *universe of all possible keys; huge set*
K : *actual keys; small set but not known in advance*

Decisions

- Choosing a hash function.
  - An ideal hash function should be simple to compute, and should ensure that any two distinct keys get different cells.
  - Since generally there is inexhaustible supply of keys, two distinct keys may map to same cell, hence we seek a hash function that distributes the keys evenly among the cells (consider telephone numbers as keys: using first 3 digits, or last 3 digits?)
- Deciding on the table size.
- Deciding what to do when two keys hash to the same value (*collision*).

Example

- Table size is 10.
- Assume *john* hashes to 3, *phil* hashes to 4, *dave* hashes to 6 and *mary* hashes to 7.
- What to do if jack also hashes to 4?

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

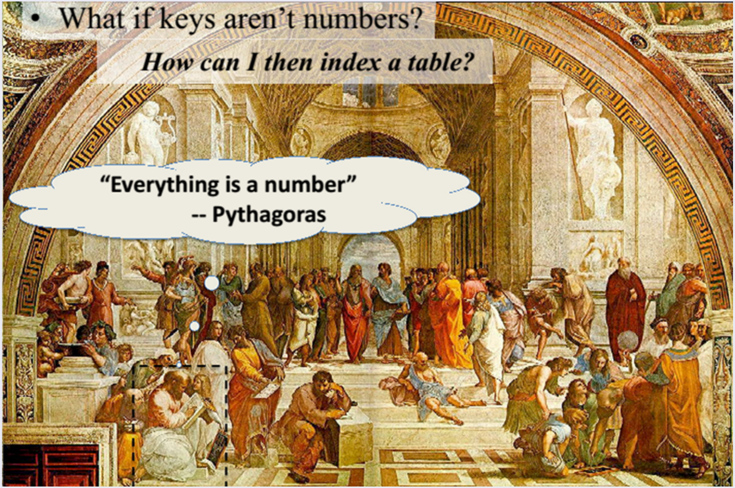
Hash Functions

- For keys that are integer a simple hash function is
  $h(k) = k \mod \mathit{TableSize}$,
  where *TableSize* is the size of the hash table array.
- Example: hash table with size 11
  $h(k) = k \mod 11$
  $80 \rightarrow 3 \;\; (80\%11 = 3)$,
  $40 \rightarrow 7$,
  $65 \rightarrow 10$
  $58 \rightarrow 3$ collision!
- It is often good idea to select a prime *TableSize*. Why?

Distribution of Keys

- Mostly even or multiple of some k
- If k is a factor of *TableSize*, then only $(\mathit{TableSize}/k)$ slots will be used

Hash Functions



Hash Functions

- What if the key is string?
- One option can be to add up the ASCII values of the characters in the string.
- Assume table size is 10007 and all keys are eight or less character strings. (127 x 8 = 1016)
- Solution:
  - Base 128 (Use Horner's Method for efficient evaluation)
  - Base 32 (Use Horner's Method for efficient evaluation)

- What if the key is <name, birthdate> ?
- What if the key is a Class?

Strings

Java library implementation

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

ith character of s

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

Ex.

```
String s = "call";
int code = s.hashCode();
```

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$
$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$
(Horner's method)

Desired Properties of Hash Function

Recall

- Distributing keys as evenly as possible in the hash table
- Simple to evaluate

Collision Resolution Policies

Definition

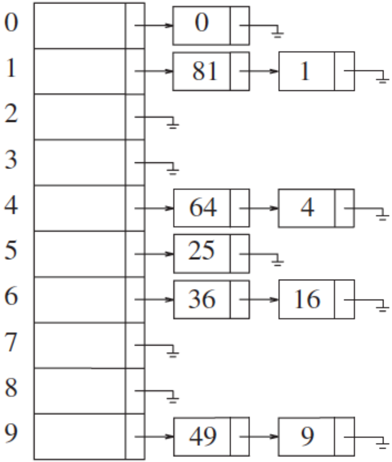
A collision occurs when two different keys hash to the same value.

- Two simple approaches:
  - Separate Chaining (Open hashing)
  - Open Addressing (Closed hashing, probing hash tables)
- The difference
  - Open Hashing: collisions are stored outside the table
  - Closed Hashing: collisions are stored at another slot in the table

Separate Chaining (Open Hashing)

- Each cell in the hash table is the head of a linked list
- All elements that hash to a particular value are placed on that cell's linked list
- Records within a list can be ordered in several ways such as
  - by order of insertion,
  - by key value order, or
  - by frequency of access order

Open Hashing Data Organization



Analysis

- Open hashing is most appropriate when the hash table is kept in the main memory, implemented with a standard in-memory linked list
- We hope that the number of elements per cell is roughly equal in size, so that the lists will be short
- If there are n elements in the set, then each cell will have roughly $\lambda = n/TableSize$ members. λ is called the load factor of the hash table.
- If we can estimate n, then we can choose *TableSize*, so that the lists will have only 1-3 members on average.

Analysis (Cont.)

- Average time per dictionary operation:
  - On average $n/TableSize$ elements per list
  - `insert`, `find`, `remove` operations take $O(1 + n/TableSize)$ time each
  - If we can choose $TableSize$ to be about n, running time will be constant
  - Assuming each element is likely to be hashed to any cell, running time will be constant and independent of n

Closed Hashing (Probing Hash Tables)

Fixed size data storage

- An alternative way to resolve collisions is to try alternative cells until an empty cell is found.
- More formally, cells $h_0(x), h_1(x), h_2(x)...$ are tried in succession, where $h_i(x) = (hash(x) + f(i)) \mod TableSize$ with $f(0) = 0$.
- The function f, is the collusion resolution strategy.
- Three common collusion resolution strategies;
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

Linear Probing

- The simplest strategy is
  called linear probing where $f(i)$ is a linear function of i, typically $f(i) = i$

$$h_i(x) = (hash(x) + f(i)) \mod TableSize$$

$$h_i(x) = (hash(x) + i) \mod TableSize$$

Linear Probing - Example

- $TableSize = 8$, keys a, b, c, d have hash values
  $h(a) = 3, h(b) = 0, h(c) = 4,$
  $h(d) = 3$
- Where do we insert d? 3 is already filled!
- Probe sequence using linear hashing:
  $h_1(d) = (h(d) + 1)\%8 = 4\%8 = 4$
  $h_2(d) = (h(d) + 2)\%8 = 5\%8 = 5$*
  $h_3(d) = (h(d) + 3)\%8 = 6\%8 = 6$
  etc.
  $7, 0, 1, 2$
- Wraps around the beginning of the table!

0	b
1	
2	
3	a
4	c
5	**d**
6	
7	

Linear Probing - Example

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

Primary Clustering

- As long as the table is big enough, a free cell can always be found
- The time to insert can get quite large even if the table is relatively empty since blocks of occupied cells are forming.
- This effect is known as primary clustering, meaning any key that hashes into the cluster -even if the keys map to different values- will require several attempts to resolve collusion and then it will be added to the cluster.
- As in other hash tables, worst case running time for `find` and `insert` is $O(n)$, where n is the number of elements in the table.

Example

1. What if the next element has home cell 0? $\rightarrow$ will be inserted to cell 3
   Same for elements with home cell 1 or 2!
   $\Rightarrow$ With probability 4/11, next record will go to cell 3
2. Similarly, records hashing to 7,8,9 will end up in 10
3. Only records hashing to 4 will end up in 4 (p=1/11); same for 5 and 6

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	

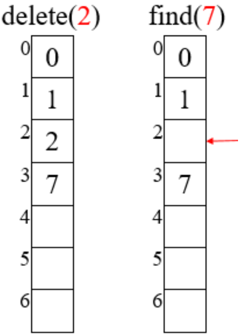
Example (Cont.)

- insert 1052 (home cell: 7)
- next element will end up in cell 3 with $p = 8/11$

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	1052

Operations Using Linear Probing

- Test for membership: `find`
  - Examine $h(k), h_1(k), h_2(k), \ldots,$ until we find k or an empty cell or home cell
- If no deletions are possible, the strategy works!
- What if there are deletions?
  - If we reach empty cell, we cannot be sure that k is not somewhere else and the empty cell was occupied when k was inserted
  - Need a special placeholder `deleted`, to distinguish the cell that was never used from the one that once held a value (Lazy Deletion)
  - May need to reorganize the table after many deletions



Quadratic Probing

- The collusion function is quadratic $f(i) = i^2$

$$h_i(x) = (hash(x) + f(i)) \mod TableSize$$

$$h_i(x) = (hash(x) + i^2) \mod TableSize$$

Quadratic Probing - Example

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

Quadratic Probing

- Eliminates the primary clustering.
- No guarantee that an empty cell will be found if table is more than half full, or even before that if table size is not prime.
- Elements that hash to the same position will probe to the same alternative cells, which is known as secondary clustering.

Double Hashing

- The collusion function includes another hash function.
  One popular option is $f(i) = i * hash_2(x)$

$$h_i(x) = (hash(x) + f(i)) \mod TableSize$$

$$h_i(x) = (hash(x) + i * hash_2(x)) \mod TableSize$$

Double Hashing - Example

- $hash_2(x) = R - (x \mod R)$ where R is a prime smaller than *TableSize*. For $R = 7$

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

Rehashing

Problem

- When table gets too full, the running time for operations will start taking too long.
- Insertion may fail for open address hashing (i.e., closed hashing) with quadratic probing.

Solution

- Building a new table that is twice big and insert all the items to the new table.
- This process is called rehashing
- Running time of rehashing is $O(n)$, but happens rarely. For any system that is time critical, this should be considered.

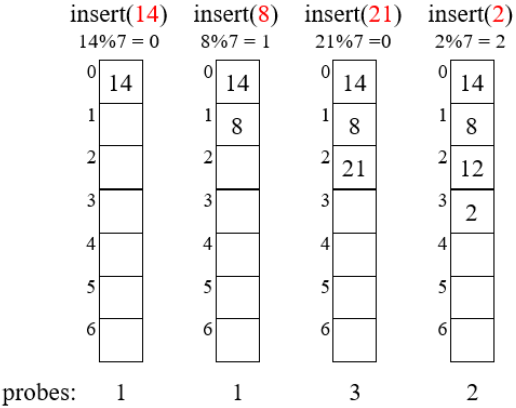
Rehashing - Example

- Linear probing with $h(x) = x \mod \mathit{TableSize}$. Inserting 13, 15, 6, 24, 23.

0	6
1	15
2	23
3	24
4	
5	
6	13

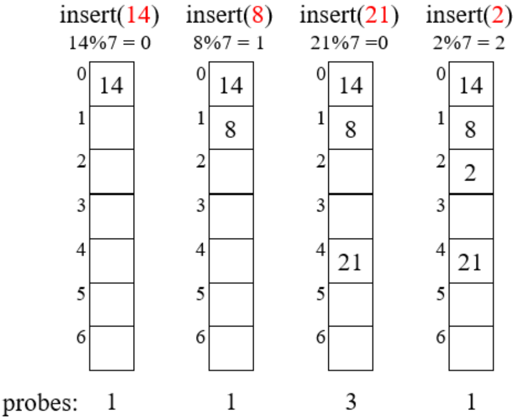
0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Closed Hashing; Linear Probing Example



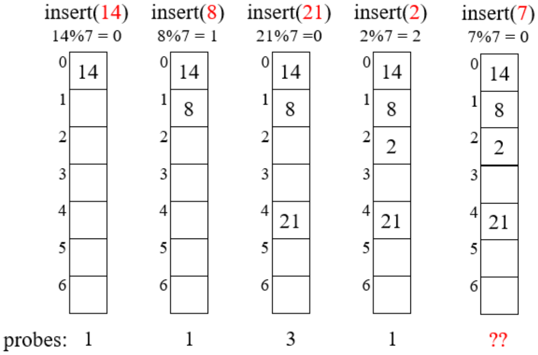
Collisions are happening, even though the hash function isn't producing lots of collisions. (Primary Clustering)

Closed Hashing; Quadratic Probing Example (1)



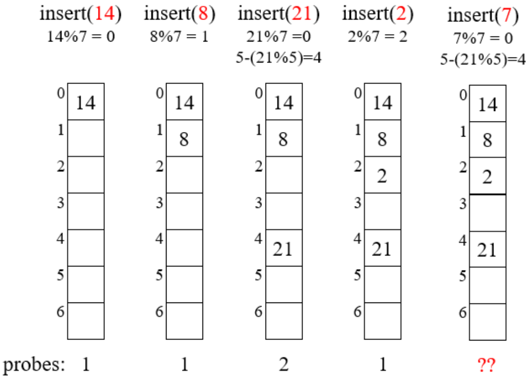
Collisions are happening for the keys that are mapping to same value. (Secondary Clustering)

Closed Hashing; Quadratic Probing Example (2)



If *TableSize* is prime and $\lambda < 0.5$, quadratic probing will find an empty slot; for greater λ, might not

Closed Hashing; Double Hash Probing Example



Load Factor λ

- Insert using Closed Hashing cannot work with $\lambda = 1$
- Quadratic probing can fail if $\lambda > 0.5$
- Linear probing and double hashing are slow if $\lambda > 0.5$
- Open Hashing becomes slow once $\lambda > 2$

Solution for Increasing Collisions

Rehashing