CS200: Tables, Priority Queues and Heaps

Prichard Ch. 12

Table Implementations

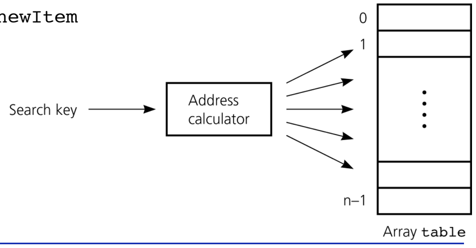
	Search	Add	Remove
Sorted array-based	O(log n)	O(n)	O(n)
Unsorted array-based	O(n)	O(1)	O(n)
Balanced Search Trees	O(log n)	O(log n)	O(log n)

Can we build a faster data structure?

Tables in O(1)

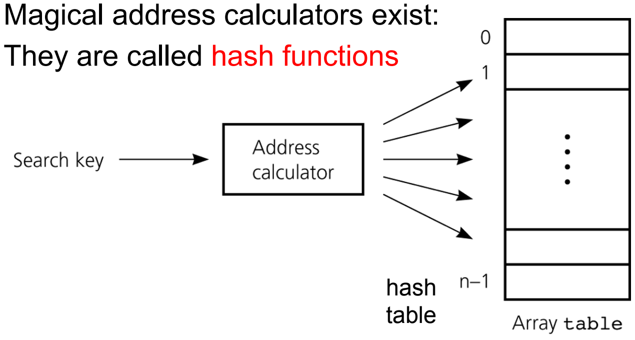
Suppose we have a magical address calculator…

```
tableInsert(in: newItem:TableItemType)
  i = index that the address calculator gives you
  for       newItem’s search key
  table[i] = newItem
```



Hash Functions and Hash Tables

Magical address calculators exist:
They are called hash functions



Hash Table: nearly-constant-time

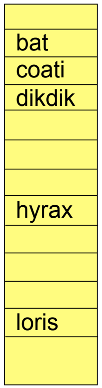
- A hash table is a dictionary in which the address of the data is determined directly from the key… which provides near constant time access!
- location of data determined from the key
  - implemented using vector / array
  - index computed from key using a **hash code**
- close to constant time access if nearly unique mapping from key to index
  - cost: extra space for unused slots

Hash Table: examples

- key is string of 3 letters
  - array of 17576 (26^3) entries, costly in space
  - hash code: letters are "radix 26" digits

    a/A -> 0, b/B -> 1, .. , z/Z -> 25,
  - Example: Joe -> 9*26*26+14*26+4
- key is student ID or social security #
  - how many likely entries?
  - what hash code?

Hash Table Issues



- Underlying data-structure
  - fixed length array, usually of prime length
  - each slot contains data
- Addressing
  - map key to slot index (hash code)
  - use a function of key
    - e.g., first letter of key
- What if we add 'capybara'?
  - collision with 'coati'

Simple Hash Functions

Credit card numbers

- 3: travel/entertainment cards (e.g. American Express and Diners Club)
  - Digits three and four are type and currency
  - Digits five through 11 are the account number
- 4: Visa
  - Digits two through six are the bank number
  - Digits seven through 12 or seven through 15 are the account number
- 5: Mastercard
  - Digits two and three, two through four, two through five or two through six are bank number
  - Till digits 15 are the account number
  - Digit 16 is a check digit

Hash Function Maps Key to Address

- Characteristics
    - uniform distribution, fast to compute
    - return an integer corresponding to slot index
        - within array size range
    - equivalent objects => equivalent hash codes
        - what is equivalent? Depends on the application, e.g. upper and lower case letters equivalent
          "Joe" == "joe"
- Perfect hash function: guarantees that every search key maps to unique address
    - takes enormous amount of space
    - cannot always be achieved (e.g., unbounded length strings)

Hash Function Computation

- Strategies:
    - Divide hash value by size of the array. (So table should be of prime length.)
    - Typical functions add together positions in key and weight their values.
- Functions on positive integers
    - Selecting digits (e.g., select a subset of digits)
    - Folding: add together digits or groups of digits
    - Modulo arithmetic: divide by table size

What is the hash function if selecting digits?

- h(001364825) = 35
- h(9783667) = 37

- h(225671) = ?

A. 39
B. 31
C. 61

Hash function: Selecting digits

- h(001364825) = 35
    - Select the fourth and last digits

- Simple and fast
    - Does not evenly distribute items

Hash function: Folding

- Suppose the search key is a 9-digit ID.
- Sum-of-digits:

  h(001364825) = 0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5

  satisfies: 0 <= h(key) <= 81
- Grouping digits: 001 + 364 + 825 = 1190

  0 <= h(search key) <=3*999=2997

What is the hash function using folding?

- h(001364825) = 29
- h(119239200) = 27

- h(336) = ?

A. 25

B. 12

C. 3

Hash Function: Folding

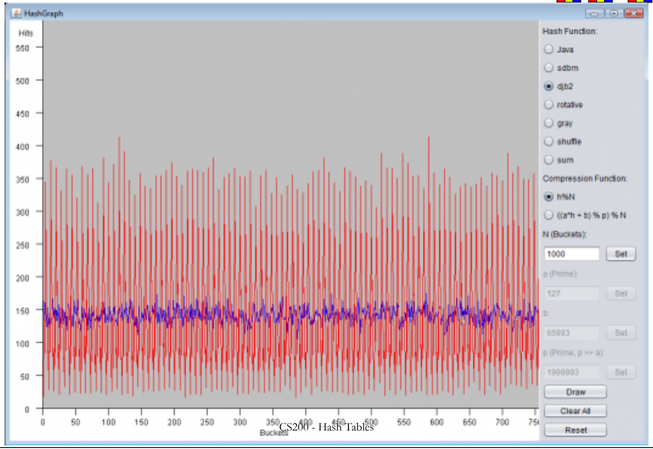
- Functions on Strings
  - Convert characters to integers, multiply by base (e.g., 32) raised to position and sum across letters using Horner's rule (e.g., "ABC" -> ((1 * 32 + 2) * 32 + 3))
  - Java provides a method for built-in objects:

    public int hashCode()
    - To use it to convert a word into an array of length hashSize,

      int code = word.toLowerCase().hashCode();

      code = Math.abs(code % hashSize);
    - hashSize should be a prime number

Impact of using prime number



Hash Function:
○ Java
○ sdbm
● djb2
○ rotative
○ gray
○ shuffle
○ sum

Compression Function:
● h%N
○ ((a*h + b) % p) % N

N (Buckets): 1000 Set
a (Prime): 127 Set
b: 65993 Set
p (Prime, p >> a): 1999993 Set

Hash function data distribution

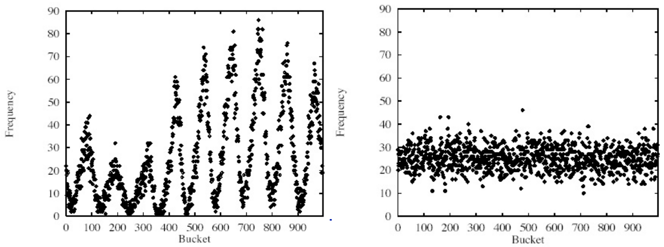
- pick a size; compute key to any integer using some hash code; index = (key -> integer) mod size
- key -> integer:

  Sum(i=0 to len-1)
  getNumericValue(string.charAt(i))*c^i
  - similar to Java built-in
- This does not work well for very long strings with large common subsets (URL), which needs hashing in a Web (Proxy) Cache.

Hash code example (cont.)

Using the Unix spelling dictionary, size = 997

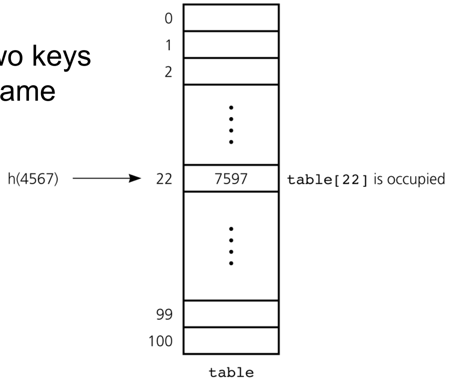
- c=1: BAD, periodic peaks
- c=31 (java) better



Collisions

Collision: two keys map to the same index

WHY?



The Birthday Problem

- What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than ½?
- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely
- p_n – the probability that all people have different birthdays

$$p_n = 1\frac{365}{366}\frac{364}{366}\cdots\frac{366-(n-1)}{366}$$

$$n = 23 \rightarrow 1 - p_n \approx 0.506$$

The Birthday Problem: Probabilities

N (# of people)	P n (Probability that at least two of the n persons have the same birthday)
10	11.7 %
20	41.1 %
23	50.7 %
30	70.6 %
50	97. 0 %
57	99.0%
100	99.99997%
200	99.999999999999999999999999999998%
366	100%

Probability of Collision

- How many items do you need to have in a hash table so that the probability of collision is greater than ½?
- For a table of size 1,000,000 you only need 1178 items for this to happen!

Methods for Handling Collisions

- Approach 1: Open addressing
  - probe for an empty slot in the hash table
- Approach 2: Restructuring the hash table
  - Change the structure of the array table

Open addressing

- A location in the hash table that is already occupied
  - Probe for some other empty, open, location in which to place the item.
  - Probe sequence
    - The sequence of locations that you examine

Open Addressing 1: Linear Probing

- Use first char. as hash function
  - Init: ale, bay, egg, home
- Where is
  - *egg* hash code 4
  - *ink* hash code 8
- Add
  - *gift* 6 empty
  - *age* 0 full, 1 full, 2 empty

Clicker Q: During the process of linear probing, if there is empty spot,
A. No item found
B. There is still a chance to find the item

ale
bay
age
egg
gift
home

Open addressing: Linear Probing

- Deletion: The empty positions created along a probe sequence could cause the retrieve method to stop, incorrectly indicating failure.
- Resolution: Each position can be in one of three states occupied, empty, or deleted. Retrieve then continue probing when encountering a deleted position. Insert into empty or deleted positions.

Linear Probing (cont.)

- remove
  - locate and then remove
  - *bay* 1
  - *age* 0

Clicker Q: Where does almond go?
A. 1
B. 2
C. 3

ale
reserved
reserved
egg
gift
home

Open Addressing 1: Linear Probing

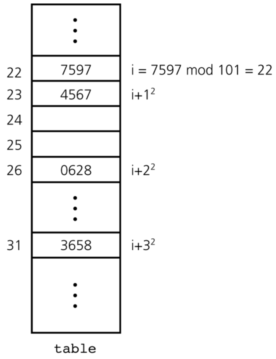
- Clustering problem
  - keys starting with ‘a’, ‘b’, ‘c’, ‘d’
  - fighting for same open slot (3)

ale
bay
age
egg
gift
home

Open Addressing: Quadratic Probing

- check

  `h(key) +` 1^2, `h(key) +` 2^2, `h(key) +` 3^2,…
- Eliminates the primary clustering phenomenon
- But.. Secondary clustering: two items that hash to the same location have the same probe sequence



Open Addressing: Double Hashing

Use two hash functions:

- h_1(key) – determines the position
- h_2(key) – determines the step size for probing
  - the secondary hash h_2 needs to satisfy:

    $h_2(key) \neq 0$

    $h_2 \neq h_1$ (why?)

- Rehashing

  Using more than one hash functions

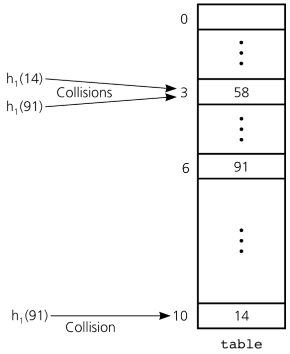
Double Hashing

Example:

$h_1(key) = key \bmod 11$

$h_2(key) = 7 - (key \bmod 7)$

Insert 58, 14, 91



Open Addressing: Increasing the table size

- Increasing the size of the table: as the table fills the likelihood of a collision increases.
  - Cannot simply increase the size of the table – need to run the hash function again

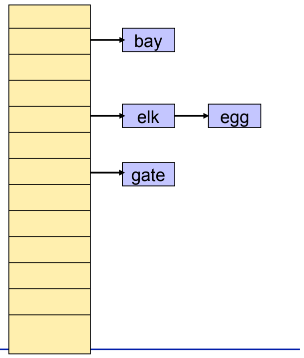
Restructuring the Hash Table: Hybrid Data Structures

- elements in hash table become collections
  - elements hashing to same slot grouped together in the collection
  - collection is a separate structure
    - e.g., ArrayList (bucket) or linked-list (separate chaining)
- a good hash function keeps a near uniform distribution, and hence the collections small
- does not need special case for removal as open addressing does

---

Separate Chaining Example

- Hash function
  - first char
- Locate
  - egg
  - gift
- Add
  - bee?
- Remove
  - bay?



The Efficiency of Hashing

- Consider a hash table with n items
  - Load factor $\alpha = n \,/\, tableSize$
  - n: current number of items in the table
  - tableSize: maximum size of array
  - α : a measure of how full the hash table is.
    - measures difficulty of finding empty slots
- Efficiency decreases as n increases

Size of Table

- Determining the size of Hash table
  - Estimate the largest possible n
  - Select the size of the table to get the load factor small.
  - Load factor should not exceed 2/3.

Hashing: Length of Probe Sequence

CS200

- Average number of comparisons that a search requires,
  - Linear Probing
    - successful $\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$
    - unsuccessful $\frac{1}{2}\left[1+\frac{1}{\left(1-\alpha^2\right)}\right]$
  - Quadratic Probing and Double Hashing
    - successful $\frac{-\log_e(1-\alpha)}{\alpha}$
    - unsuccessful $\frac{1}{1-\alpha}$

From D.E. Knuth, Searching and Sorting, Vol. 3 of The Art of C

Hashing: Length of Probe Sequence

CS200

- Chaining
  - successful: 1 + α/2
  - unsuccessful: α
  - Note that α can be > 1

Traversal of Hash Tables

CS200

- If you need to traverse your tables by the sorted order of keys – hash tables may not be the appropriate data structure.

Hash Tables in Java

CS200

From the JAVA API: “A map is an object that maps keys to values… The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.” Both provide methods to create and maintain a hash table data structure with key lookup.

```
public class Hashtable<K,V> extends Dictionary<K,V>
  implements Map<K,V>

public class HashMap<K,V> extends AbstractMap<K,V>
  implements Map<K,V>
  public HashMap(int initialCapacity, float loadFactor)
  public HashMap(int initialCapacity)  //default
  loadFactor: 0.75
```