Hashing

- We can do better than O(log n) to find elements.
- If we have a function to convert a search key into a position indicator (e.g. array index) we can find values in constant time O(1).
- Hashing is incredibly useful, many modern languages include hashtables as built-in data types.

Basic ideas of Hashing

- If the search key was a number from 1 to 1000
  - You could have an array of 1000 elements (called a hashtable).
  - The element with search key n would be at table[n – 1]
- If we don't have the same number of elements as search keys or if we don't know all the possible search keys then we have to use a hash function to convert the search key into an appropriate value.

Saving space

- What if we only expect to have 50 elements in our array. We have a lot of wasted space.
- We can take the bottom two digits of the search key as the index into an array of 100 elements.
- 145 is in table[45]
- 709 is in table[9]
- What about 545?

Example hash function

- If the key is a string
  - take the ASCII value of each character
  - add the values together
  - get the modulo of this value when divided by the size of the hash table (array)
  - e.g. If the hash table is 101 elements long and the key is "hello" the hash value is (104+101+108+108+111)%101 = 27
  - this is not a particularly good hash function but it shows the idea

Hashing

- Hashing
  - Enables access to table items in time that is relatively constant and independent of the items
- Hash function
  - Maps the search key of a table item into a location that will contain the item
- Hash table
  - An array that contains the table items, as assigned by a hash function

Hashing

- A perfect hash function
  - Maps each search key into a unique location of the hash table
  - Possible if all the search keys are known
- Collisions
  - Occur when the hash function maps more than one item into the same array location
- Collision-resolution schemes
  - Assign other locations in the hash table to items which are involved in a collision
- Requirements for a hash function
  - Be easy and fast to compute
  - Place items evenly throughout the hash table

Hash Functions

- It is sufficient for hash functions to operate on integers
- Simple hash functions that operate on positive integers
  - Selecting digits
  - Folding
  - Modulo arithmetic
- Converting a character string to an integer
  - If the search key is a character string, it can be converted into an integer before the hash function is applied

Resolving Collisions

- Two approaches to collision resolution
  - Approach 1: Open addressing
    - A category of collision resolution schemes that probe for an empty, or open, location in the hash table
      - The sequence of locations that are examined is the probe sequence
    - Linear probing
      - Searches the hash table sequentially, starting from the original location specified by the hash function
      - Possible problem
        - » Primary clustering

Resolving Collisions

- Approach 1: Open addressing (Continued)
  - Quadratic probing
    - Searches the hash table beginning with the original location that the hash function specifies and continues at increments of 1^2, 2^2, 3^2, and so on
  - Double hashing
    - Uses two hash functions
    - Searches the hash table starting from the location that one hash function determines and considers every n^{th} location, where n is determined from a second hash function
- Increasing the size of the hash table
  - The hash function must be applied to every item in the old hash table before the item is placed into the new hash table

Resolving Collisions

- Approach 2: Restructuring the hash table
  - Changes the structure of the hash table so that it can accommodate more than one item in the same location
  - Buckets
    - Each location in the hash table is itself an array called a bucket
  - Separate chaining
    - Each hash table location is a linked list

The Efficiency of Hashing

- An analysis of the average-case efficiency of hashing involves the load factor
  - Load factor α
    - Ratio of the current number of items in the table to the maximum size of the array `table`
    - Measures how full a hash table is
    - Should not exceed 2/3 (Java uses 0.75 as the default before growing the table)
  - Hashing efficiency for a particular search also depends on whether the search is successful
    - Unsuccessful searches generally require more time than successful searches

The Efficiency of Hashing

- Linear probing
  - Successful search: $\frac{1}{2}[1 + 1/(1-\alpha)]$
  - Unsuccessful search: $\frac{1}{2}[1 + 1/(1- \alpha)^2]$
- Quadratic probing and double hashing
  - Successful search: $-\log_e(1- \alpha)/ \alpha$
  - Unsuccessful search: $1/(1- \alpha)$
- Separate chaining
  - Insertion is O(1)
  - Retrievals and deletions
    - Successful search: $1 + (\alpha/2)$
    - Unsuccessful search: α

The Efficiency of Hashing

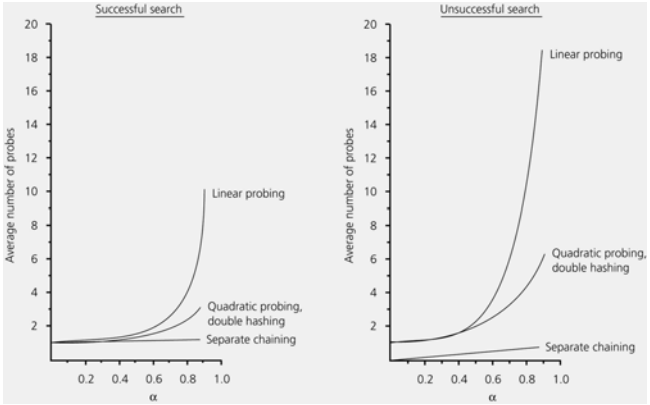

Figure 13-50

The relative efficiency of four collision-resolution methods

What Constitutes a Good Hash Function?

- A good hash function should
  - Be easy and fast to compute
  - Scatter the data evenly throughout the hash table
- Issues to consider with regard to how evenly a hash function scatters the search keys
  - How well does the hash function scatter random data?
  - How well does the hash function scatter nonrandom data?
- General requirements of a hash function
  - The calculation of the hash function should involve the entire search key
  - If a hash function uses modulo arithmetic, the base should be prime

JCF – HashMap example

```
import java.util.*;

public class HashExample {

    public static void main(String[] args) {
        Map<String, String> dictionary = new HashMap<String,
          String>();
        dictionary.put("ROTFL", "rolling on the floor
          laughing");
        dictionary.put("IIRC", "if I recall correctly");
        dictionary.put("IMHO", "in my humble opinion");
        dictionary.put("AFAIK", "as far as I know");
        dictionary.put("BTW", "by the way");

        System.out.println(dictionary.get("IMHO"));
        dictionary.put("IMHO", "in my honest opinion");
        System.out.println(dictionary.get("IMHO"));
    }
}
```


Table Traversal: An Inefficient Operation Under Hashing

- Hashing as an implementation of the ADT table
  - For many applications, hashing provides the most efficient implementation
  - Hashing is not efficient for
    - Traversal in sorted order
    - Finding the item with the smallest or largest value in its search key
    - Range query
- In external storage, you can simultaneously use
  - A hashing implementation of the `tableRetrieve` operation
  - A search-tree implementation of the ordered operations

Summary

- Hashing as a table implementation calculates where the data item should be rather than search for it
- A hash function should be extremely easy to compute and should scatter the search keys evenly throughout the hash table
- A collision occurs when two different search keys hash into the same array location
- Hashing does not efficiently support operations that require the table items to be ordered
- Hashing as a table implementation is simpler and faster than balanced search tree implementations when table operations such as traversal are not important to a particular application
- Several independent organizations can be imposed on a given set of data

Exam questions

1	Exceptions		5
2	Multi-dimensional arrays		10
3	Recursion		10
4	Files		10
5	Sorting		10
6	Abstract Data Types		15
7	Referenced-based lists		10
8	Balanced trees		9
9	Queues		12
10	Binary Search Trees		9