Hashing

- Hashing
  - Enables access to items in time that is relatively constant and independent of the items
- Hash function
  - Maps the search key of an 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
- Collisions
  - Occur when the hash function maps more than one item into the same array location
- Collision-resolution schemes
  - Assign locations in the hash table to items with different search keys when the items 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
- 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

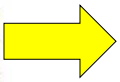
- Two approaches to collision resolution
  - Approach 1: Open addressing
    - A category of collision resolution schemes that probe for an empty, or open, location in the hash table
      - The sequence of locations that are examined is the probe sequence
    - Linear probing
      - Searches the hash 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
    - Possible problem
      - Secondary clustering
  - 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
  - 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: $\frac{1}{2}[1 + 1/(1- \alpha)^2]$
- Separate chaining
  - Insertion is O(1)
  - Retrievals and deletions
    - Successful search: $1 + (\alpha/2)$
    - Unsuccessful search: α

The Efficiency of Hashing

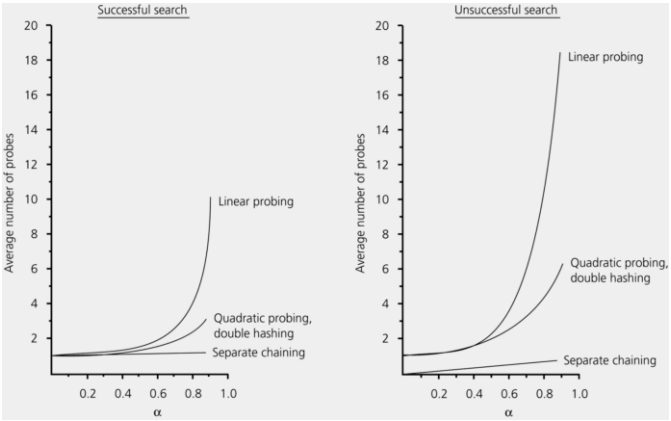


Figure 13-50

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 module arithmetic, the base should be prime

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

The JCF `Hashtable` Class

- JFC Hashtable implements a hash table
  - Maps keys to values
  - Large collection of methods

Example 1

- For a hash table of size 7 that uses h(x) = x mod 7 and linear probing to resolve collisions, what does the hash table look like after the following insertions occur: 8, 10, 24, 15, 32, 17? Assume that each item contains only a search key.

Example 2

The success of a hash-table is related to the choice of a good hash function. A good hash function is one that is easy to compute and will evenly distribute the possible data. Comment on the appropriateness of the following hash functions. What patterns would hash to the same location?

- table size 2048, search keys are English words, the hash function is h(key) = sum of positions in alphabet of key's letters) mod 2048.

Example 2 (Con't)

- Table size = 10000, keys are integers in the range 0 through 9999, the hash function is h(key) = (key * random) truncated to an integer.

  where random represents a random number generator that returns a real value between 0 and 1.