Hashing

- Hashing
  - Enables access to table items in time that is relatively constant regardless of their locations
- 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
  - Possible if all the search keys are known
- Collision
  - Occurs when the hash function maps two or more items—all having different search keys—into the same array location
- Collision-resolution scheme
  - Assigns distinct locations in the hash table to items involved in a collision

Hashing

- Requirements for a hash function
  - Is easy and fast to compute
  - Places items evenly throughout the hash table
  - Involves the entire search key
  - Uses a prime base, if it uses modulo arithmetic

Hash Functions

- Simple hash functions
  - Selecting digits
    - Does not distribute items evenly
  - Folding
  - Modulo arithmetic
    - The table size should be prime
  - Converting a character string to an integer
    - Use the integer in the hash function instead of the string search key

Resolving Collisions

- Approach 1: Open addressing
  - A category of collision resolution schemes that probe for an empty, or open, location in the hash table
  - As the hash table fills, collisions increase
    - The size of the hash table must be increased
    - Need to hash the items again

Resolving Collisions

- Approach 1: Open addressing
  - Linear probing
    - Searches the hash table sequentially, starting from the original location specified by the hash function
  - Quadratic probing
    - Searches the hash table beginning at the original location specified by the hash function and continuing at increments of 1^2, 2^2, 3^2, and so on
  - Double hashing
    - Uses two hash functions
    - One specifies the first location, the other gives the step size

Resolving Collisions

- Approach 2: Restructuring the hash table
  - Allows the hash table to accommodate more than one item in the same location
  - Buckets
    - Each location in the hash table is itself an array
  - Separate chaining
    - Each hash table location is a linked list
    - Successfully resolves collisions
    - The size of the hash table is dynamic

Resolving Collisions

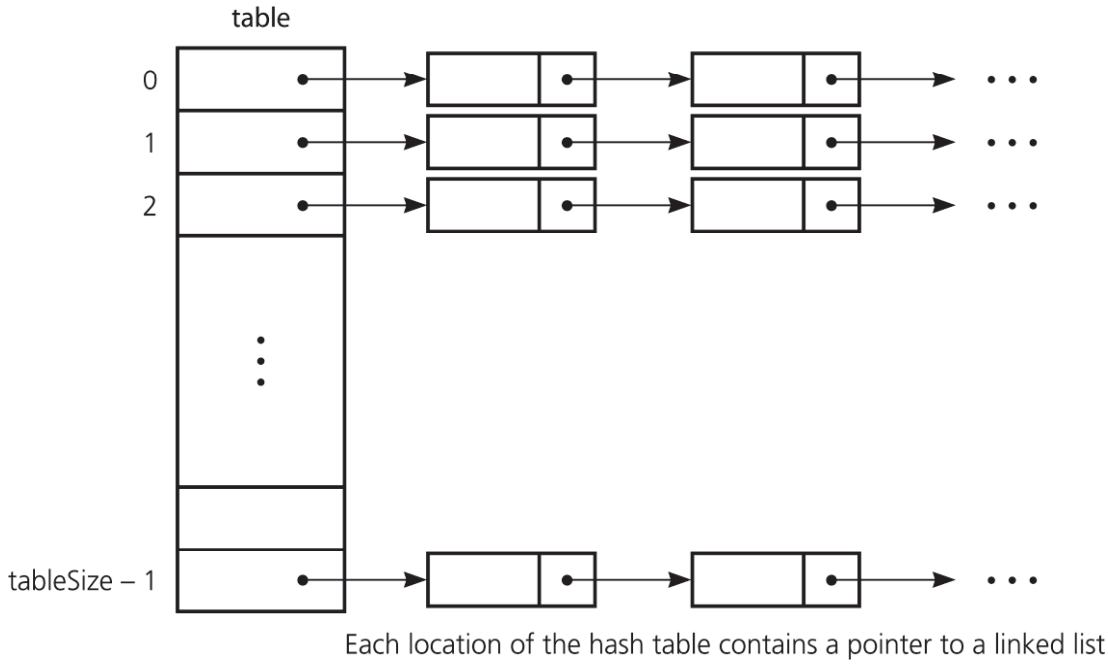

Figure 12-49 Separate chaining

The Efficiency of Hashing

- An analysis of the average-case efficiency
  - 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

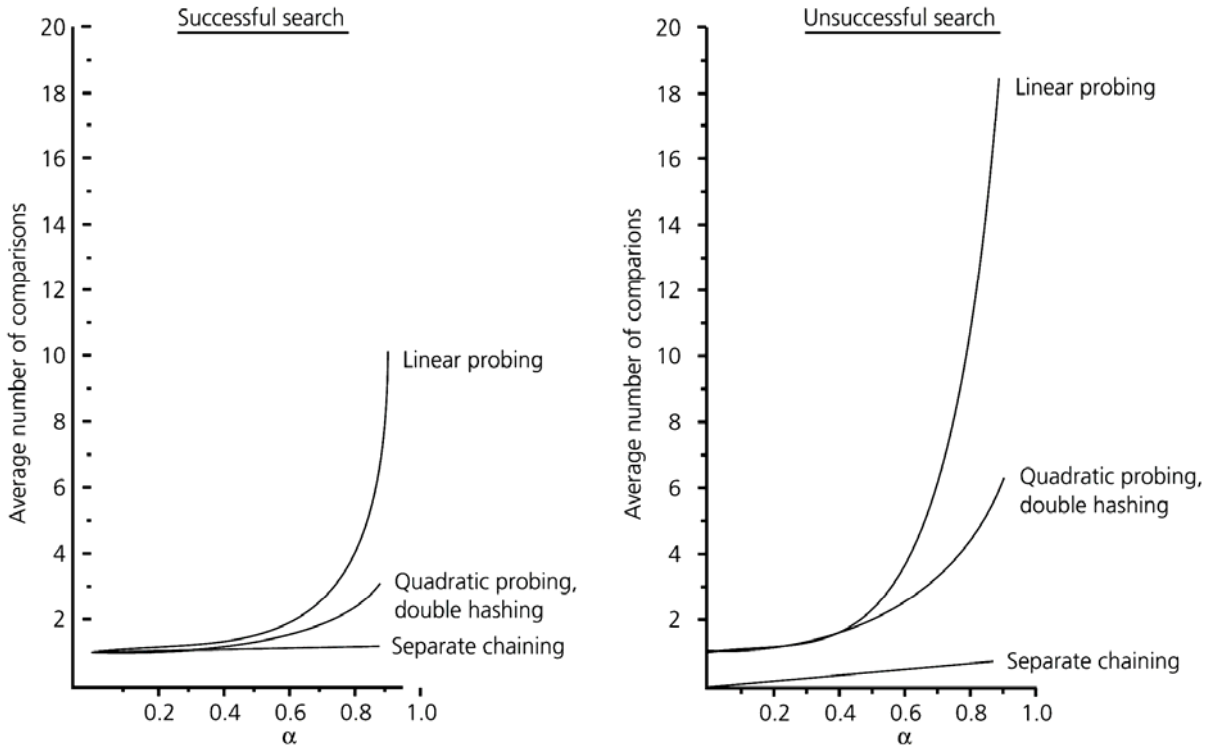

Figure 12-50 The relative efficiency of four collision-resolution methods

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 a poor choice for certain operations
    - Traversal in sorted order
    - Finding the item that has the smallest or largest search key
    - Range query

Implementing a HashMap Class Using the STL

- A template class *HashMap* can be derived from existing STL containers
- An implementation using separate chaining
  - A vector holds the dynamic hash buckets
  - Each bucket is a map that holds elements having the same hash value
  - The hash function is supplied as a template parameter

```
template <typename Key, typename T, typename Hash>
class HashMap : private vector<map<Key, T> >
```

Data With Multiple Organizations

- Many applications require a data organization that simultaneously supports several different data-management tasks
  1. Several independent data structures
     - Do not support all operations efficiently
     - Waste space
  2. Interdependent data structures
     - Provide a better way to support a multiple organization of data

Data With Multiple Organizations

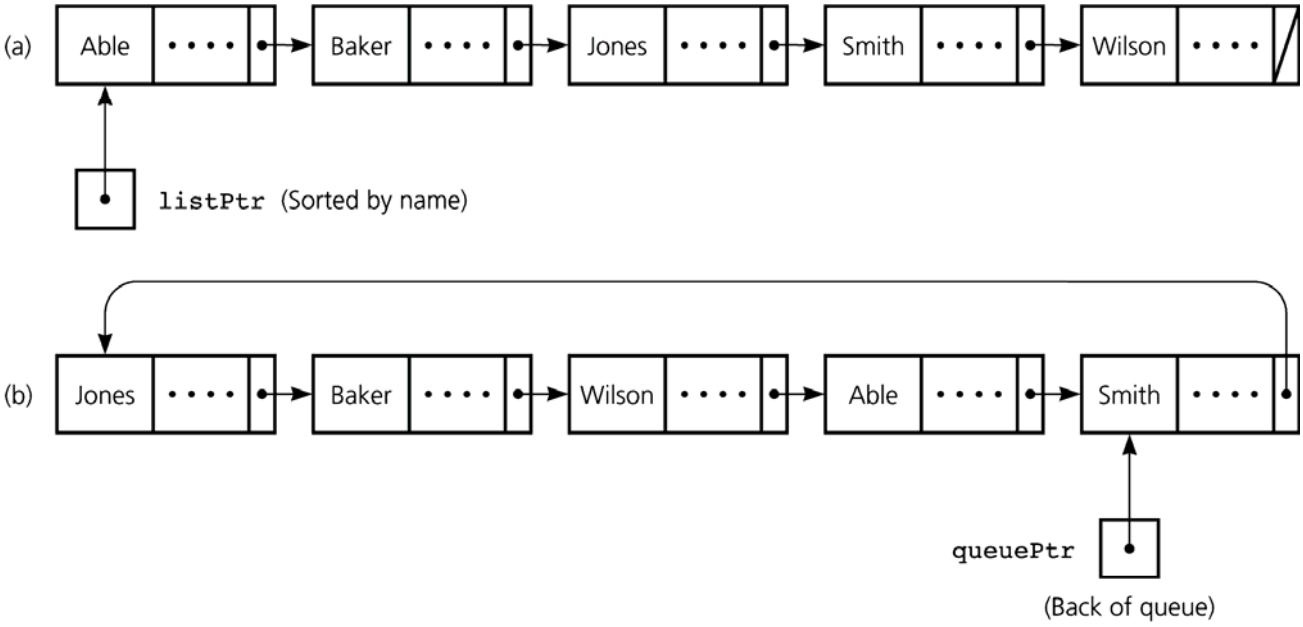

Figure 12-51 Independent data structures: (a) a sorted linked list; (b) a pointer-based queue

Data With Multiple Organizations

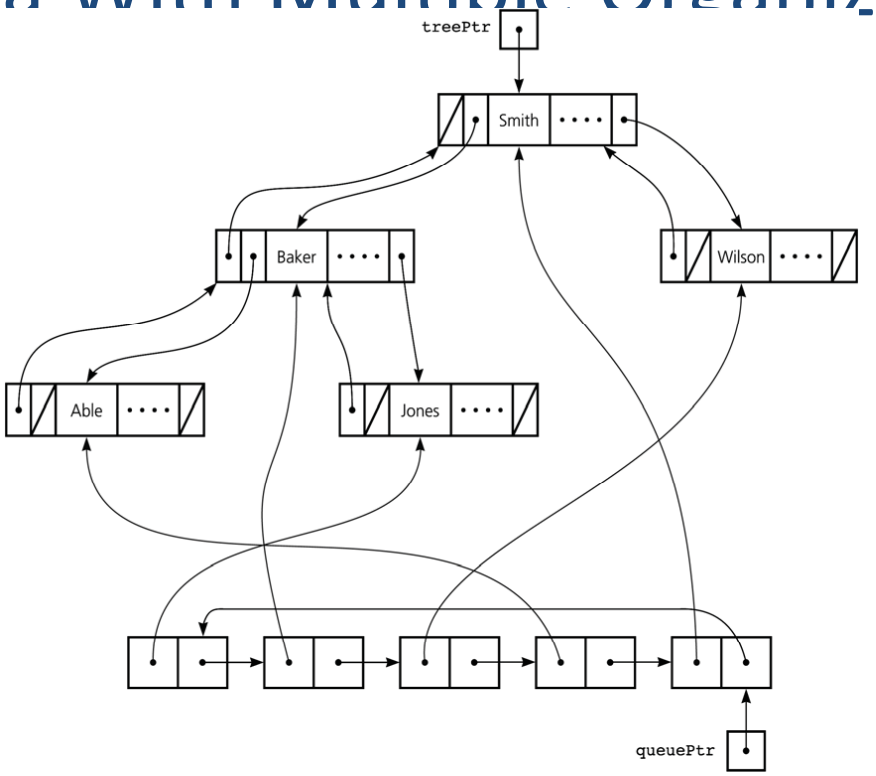

Figure 12-54 A queue pointing into a doubly linked binary search tree

Summary

- 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
- Probing and chaining are two ways to resolve a collision

Summary

- Hashing as a table implementation
  - Does not efficiently support operations that require the table items to be ordered
  - Is simpler and faster than balanced search tree implementations when
    - Traversals are not important
    - The maximum number of table items is known
    - Ample storage is available

Summary

- Some applications require data organized in more than one way
  - Several independent data structures can be inefficient and waste space
  - Several data structures linked together can be a good choice