AVL Trees

- Rotations
  - Restore the balance of a tree
  - Two types
    - Single rotation
    - Double rotation

![Figure 13-38](a) An unbalanced binary search tree; b) a balanced tree after a single left rotation)

Advantage
- Height of an AVL tree with n nodes is always very close to the theoretical minimum

Disadvantage
- An AVL tree implementation of a table is more difficult than other implementations
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 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
- Simple hash functions that operate on positive integers
  - Selecting digits
  - Folding
  - Module 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
  - Quadratic probing
    - Searches the hash table beginning with the original location that the hash function specifies and continues at increments of 1, 2, 3, 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\textsuperscript{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 \( \alpha \)
    - Ratio of the current number of items in the table to the maximum size of the array
    - 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: \( \frac{1}{2} \left( 1 + (1-\alpha) \right) \)
  - Unsuccessful search: \( \frac{1}{2} \left( 1 + (1-\alpha)^2 \right) \)
- Quadratic probing and double hashing
  - Successful search: \( \log(1-\alpha)/\alpha \)
  - Unsuccessful search: \( 1/(1-\alpha) \)
- Separate chaining
  - Insertion is \( O(1) \)
  - Retrievals and deletions
    - Successful search: \( 1 + (\alpha/2) \)
    - Unsuccessful search: \( \alpha \)

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 and TreeMap Classes

- JCF Hashtable implements a hash table
  - Maps keys to values
  - Large collection of methods
- JCF TreeMap implements a red-black tree
  - Guarantees \( O(\log n) \) time for insert, retrieve, remove, and search
  - Large collection of methods

Data With Multiple Organizations

- Many applications require a data organization that simultaneously supports several different data-management tasks
  - Several independent data structures do not support all operations efficiently
  - Interdependent data structures provide a better way to support a multiple organization of data
Summary

- A 2-3 tree and a 2-3-4 tree are variants of a binary search tree in which the balanced is easily maintained
- The insertion and deletion algorithms for a 2-3-4 tree are more efficient than the corresponding algorithms for a 2-3 tree
- A red-black tree is a binary tree representation of a 2-3-4 tree that requires less storage than a 2-3-4 tree
- An AVL tree is a binary search tree that is guaranteed to remain balanced
- Hashing as a table implementation calculates where the data item should be rather than search for it

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
- 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