Chapter 13

Advanced Implementation of Tables
Balanced Search Trees

• The efficiency of the binary search tree implementation of the ADT table is related to the tree’s height
  – Height of a binary search tree of n items
    • Maximum: n
    • Minimum: $[\log_2(n + 1)]$

• Height of a binary search tree is sensitive to the order of insertions and deletions

• Variations of the binary search tree
  – Can retain their balance despite insertions and deletions
2-3 Trees

- A 2-3 tree
  - Has 2-nodes and 3-nodes
    - A 2-node
      - A node with one data item and two children
    - A 3-node
      - A node with two data items and three children
  - Is not a binary tree
  - Is never taller than a minimum-height binary tree
    - A 2-3 tree with $n$ nodes never has height greater than $\lceil \log_2(n + 1) \rceil$
2-3 Trees

- Rules for placing data items in the nodes of a 2-3 tree
  - A 2-node must contain a single data item whose search key is
    - Greater than the left child’s search key(s)
    - Less than the right child’s search key(s)
  - A 3-node must contain two data items whose search keys $S$ and $L$ satisfy the following
    - $S$ is
      - Greater than the left child’s search key(s)
      - Less than the middle child’s search key(s)
    - $L$ is
      - Greater than the middle child’s search key(s)
      - Less than the right child’s search key(s)
  - A leaf may contain either one or two data items
2-3 Trees

Figure 13-3
Nodes in a 2-3 tree a) a 2-node; b) a 3-node
2-3 Trees

• Traversing a 2-3 tree
  – To traverse a 2-3 tree
    • Perform the analogue of an inorder traversal

• Searching a 2-3 tree
  – Searching a 2-3 tree is as efficient as searching the shortest binary search tree
    • Searching a 2-3 tree is $O(\log_2 n)$
    • Number of comparisons required to search a 2-3 tree for a given item
      – Approximately equal to the number of comparisons required to search a binary search tree that is as balanced as possible
2-3 Trees

• Advantage of a 2-3 tree over a balanced binary search tree
  – Maintaining the balance of a binary search tree is difficult
  – Maintaining the balance of a 2-3 tree is relatively easy
2-3 Trees: Inserting Into a 2-3 Tree

- Insertion into a 2-node leaf is simple
- Insertion into a 3-node causes it to divide
2-3 Trees: The Insertion Algorithm

- To insert an item $I$ into a 2-3 tree
  - Locate the leaf at which the search for $I$ would terminate
  - Insert the new item $I$ into the leaf
  - If the leaf now contains only two items, you are done
  - If the leaf now contains three items, split the leaf into two nodes, $n_1$ and $n_2$
2-3 Trees: The Insertion Algorithm

- When an internal node contains three items
  - Split the node into two nodes
  - Accommodate the node’s children

Figure 13-13
Splitting an internal node in a 2-3 tree
2-3 Trees: The Insertion Algorithm

- When the root contains three items
  - Split the root into two nodes
  - Create a new root node

Figure 13-14
Splitting the root of a 2-3 tree
2-3 Trees: Deleting from a 2-3 Tree

• Deletion from a 2-3 tree
  – Does not affect the balance of the tree

• Deletion from a balanced binary search tree
  – May cause the tree to lose its balance
2-3 Trees: The Deletion Algorithm

Figure 13-19a and 13-19b
a) Redistributing values;
b) merging a leaf
2-3 Trees: The Deletion Algorithm

Figure 13-19c and 13-19d

(c) redistributing values and children; d) merging internal nodes
2-3 Trees: The Deletion Algorithm

Figure 13-19e

e) deleting the root
2-3 Trees: The Deletion Algorithm

• When analyzing the efficiency of the insertItem and deleteItem algorithms, it is sufficient to consider only the time required to locate the item.

• A 2-3 implementation of a table is $O(\log_2 n)$ for all table operations.

• A 2-3 tree is a compromise:
  – Searching a 2-3 tree is not quite as efficient as searching a binary search tree of minimum height.
  – A 2-3 tree is relatively simple to maintain.
2-3-4 Trees

- Rules for placing data items in the nodes of a 2-3-4 tree
  - A 2-node must contain a single data item whose search keys satisfy the relationships pictured in Figure 13-3a
  - A 3-node must contain two data items whose search keys satisfy the relationships pictured in Figure 13-3b
  - A 4-node must contain three data items whose search keys S, M, and L satisfy the relationship pictured in Figure 13-21
  - A leaf may contain either one, two, or three data items

Figure 13-21
A 4-node in a 2-3-4 tree
2-3-4 Trees: Searching and Traversing a 2-3-4 Tree

- Search and traversal algorithms for a 2-3-4 tree are simple extensions of the corresponding algorithms for a 2-3 tree
2-3-4 Trees: Inserting into a 2-3-4 Tree

• The insertion algorithm for a 2-3-4 tree
  – Splits a node by moving one of its items up to its parent node
  – Splits 4-nodes as soon as its encounters them on the way down the tree from the root to a leaf
• Result: when a 4-node is split and an item is moved up to the node’s parent, the parent cannot possibly be a 4-node and can accommodate another item
2-3-4 Trees: Splitting 4-nodes During Insertion

• A 4-node is split as soon as it is encountered during a search from the root to a leaf

• The 4-node that is split will
  – Be the root, or
  – Have a 2-node parent, or
  – Have a 3-node parent

Figure 13-28
Splitting a 4-node root during insertion
2-3-4 Trees: Splitting 4-nodes During Insertion

Figure 13-29
Splitting a 4-node whose parent is a 2-node during insertion
2-3-4 Trees: Splitting 4-nodes During Insertion

Figure 13-30
Splitting a 4-node whose parent is a 3-node during insertion
2-3-4 Trees: Deleting from a 2-3-4 Tree

- The deletion algorithm for a 2-3-4 tree
  - Locate the node \( n \) that contains the item \( \text{theItem} \)
  - Find \( \text{theItem} \)'s inorder successor and swap it with \( \text{theItem} \) (deletion will always be at a leaf)
  - If that leaf is a 3-node or a 4-node, remove \( \text{theItem} \)
  - To ensure that \( \text{theItem} \) does not occur in a 2-node
    - Transform each 2-node encountered into a 3-node or a 4-node
2-3-4 Trees: Concluding Remarks

• Advantage of 2-3 and 2-3-4 trees
  – Easy-to-maintain balance
• Insertion and deletion algorithms for a 2-3-4 tree require fewer steps than those for a 2-3 tree
• Allowing nodes with more than four children is counterproductive
Red-Black Trees

• A 2-3-4 tree
  – Advantages
    • It is balanced
    • Its insertion and deletion operations use only one pass from root to leaf
  – Disadvantage
    • Requires more storage than a binary search tree

• A red-black tree
  – A special binary search tree
  – Used to represent a 2-3-4 tree
  – Has the advantages of a 2-3-4 tree, without the storage overhead
Red-Black Trees

• Basic idea
  – Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree

• Red and black children references
  – Used to distinguish between 2-nodes that appeared in the original 2-3-4 tree and 2-nodes that are generated from 3-nodes and 4-nodes
    • Black references are used for child references in the original 2-3-4 tree
    • Red references are used to link the 2-nodes that result from the split 3-nodes and 4-nodes
Red-Black Trees

Figure 13-31
Red-black representation of a 4-node

Figure 13-32
Red-black representation of a 3-node
Red-Black Trees: Searching and Traversing a Red-Black Tree

- A red-black tree is a binary search tree
- The algorithms for a binary search tree can be used to search and traverse a red-black tree
Red-Black Trees: Inserting and Deleting From a Red-Black Tree

• Insertion algorithm
  – The 2-3-4 insertion algorithm can be adjusted to accommodate the red-black representation
  • The process of splitting 4-nodes that are encountered during a search must be reformulated in terms of the red-black representation
    – In a red-black tree, splitting the equivalent of a 4-node requires only simple color changes
    – Rotation: a reference change that results in a shorter tree

• Deletion algorithm
  – Derived from the 2-3-4 deletion algorithm
Red-Black Trees: Inserting and Deleting From a Red-Black Tree

Figure 13-34
Splitting a red-black representation of a 4-node that is the root
Red-Black Trees: Inserting and Deleting From a Red-Black Tree

**Figure 13-35**
Splitting a red-black representation of a 4-node whose parent is a 2-node
Red-Black Trees: Inserting and Deleting From a Red-Black Tree

Figure 13-36a
Splitting a red-black representation of a 4-node whose parent is a 3-node
Red-Black Trees: Inserting and Deleting From a Red-Black Tree

Figure 13-36b
Splitting a red-black representation of a 4-node whose parent is a 3-node
Red-Black Trees: Inserting and Deleting From a Red-Black Tree

Figure 13-36c
Splitting a red-black representation of a 4-node whose parent is a 3-node
AVL Trees

• An AVL tree
  – A balanced binary search tree
  – Can be searched almost as efficiently as a minimum-height binary search tree
  – Maintains a height close to the minimum
  – Requires far less work than would be necessary to keep the height exactly equal to the minimum

• Basic strategy of the AVL method
  – After each insertion or deletion
    • Check whether the tree is still balanced
    • If the tree is unbalanced, restore the balance
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
AVL Trees

Figure 13-42
a) Before; b) during; and c) after a double rotation
AVL Trees

• 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

- 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 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
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 $n^{\text{th}}$ 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 $\text{table}$
    • 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: $\frac{1}{2}[1 + 1(1 - \alpha)^2]$

- **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: $\alpha$
The Efficiency of Hashing

Figure 13-50
The relative efficiency of four collision-resolution methods

© 2011 Pearson Addison-Wesley. All rights reserved
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

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