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

Applications

- Cryptography
  - Integrity and authentication
- Error Corrections
  - CRC
  - Reed-Solomon Codes
- Audio Identification
  - MD5
- String Search Algorithms
  - Rabin-Karp string search algorithm, \(O(n)\)
  - Bloom Filter
    - Check whether an element is in a set (false positive are possible!)

HashMap diagram

How does Hashing work?

- The key is put through a hash function, which converts the key to an integer, which is then used to index the array
- If there is nothing at this location, the element is stored there
- If there is something at this location, we have to resolve the collision.
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, i.e. complexity O(1)
  - Place items evenly throughout the hash table

Hash Function Properties

- Assume hash table has \( m \) elements
- Hash function maps a key to an integer number between 0 and \( m-1 \)
- If the distribution of keys is uniform, a good hash function has the property
  \[ |H(i) - H(j)| \leq 1 \]
  where
  \[ H(i) = \{ \text{key} | \text{hash(key)} = i \} \]
  so the number of items mapped to the same hash value is about the same.

Examples of Hash Functions

- Integers
  \[ h(x) = x \mod m \]
- Strings
  ```
  int h(String s)
  {
      int h=0;
      for (i=0; i<s.length; i++) {
          h = ((h<<5)^(h>>27))^s.charAt(i);
      }
      h = h % m;
      return h;
  }
  ```

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 non-random 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 (why?)
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 nth location, where n is determined from a second hash function
    - Possible problem: 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 (Continued)

- 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: \( \frac{m}{n} \)
    - Ratio of the current number of items in the table to the maximum number of items it can accommodate
  - Measures how full a hash table is
  - Should not exceed 2/3 in general (see later)
  - Hashing efficiency for a particular search also depends on whether the search is successful
    - Unsuccessful searches generally require more time than successful searches (why?)

The Efficiency of Hashing

- Linear probing
  - Successful search: \( \frac{1}{2} + \frac{1}{1 + \alpha} \)
  - Unsuccessful search: \( \frac{1}{2} + \frac{1}{1 + (1 - \alpha)^2} \)

- Quadratic probing and double hashing
  - Successful search: \( \frac{1}{1 - \alpha} \)
  - Unsuccessful search: \( \frac{1}{1 - \alpha} \)

- Separate chaining
  - Insertion is \( O(1) \)
  - Retrievals and deletions
    - Successful search: \( 1 + (\alpha/2) \)
    - Unsuccessful search: \( \alpha \)
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 `tableRetrieves` operation
  - A search-tree implementation of the ordered operations

Open Addressing: Unsuccessful Insert/Retrieval

- Assume a probability \( p(i) \) of \( 1/m \) for storing \( h(key) \) at location \( i \).
- Assume \( T(i) \) is the length of the list at index \( i \).
- Average complexity of an unsuccessful retrieval is then

\[
A(n) = \sum_{i=0}^{m-1} p(i) T(i) = \frac{1}{m} \sum_{i=0}^{m-1} T(i) = \frac{n}{m}
\]

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

Terminology

- \( G = (V, E) \)
- A graph \( G \) consists of two sets
  - A set \( V \) of vertices, or nodes
  - A set \( E \) of edges
- A subgraph
  - Consists of a subset of a graph’s vertices and a subset of its edges
- Adjacent vertices
  - Two vertices that are joined by an edge

Graphs

Terminology

- Figure 14-2
  a) A campus map as a graph; b) a subgraph
Terminology

- A path between two vertices
  - A sequence of edges that begins at one vertex and ends at another vertex
  - May pass through the same vertex more than once
- A simple path
  - A path that passes through a vertex only once
- A cycle
  - A path that begins and ends at the same vertex
- A simple cycle
  - A cycle that does not pass through a vertex more than once

Terminology

- A connected graph
  - A graph that has a path between each pair of distinct vertices
- A disconnected graph
  - A graph that has at least one pair of vertices without a path between them
- A complete graph
  - A graph that has an edge between each pair of distinct vertices

Terminology

- Multigraph
  - Not a graph
  - Allows multiple edges between vertices

Terminology

- Weighted graph
  - A graph whose edges have numeric labels

Terminology

- Undirected graph
  - Edges do not indicate a direction
- Directed graph, or digraph
  - Each edge is a directed edge
**Terminology**

- Directed graph
  - Can have two edges between a pair of vertices, one in each direction
- Directed path
  - A sequence of directed edges between two vertices
  - Vertex $y$ is adjacent to vertex $x$ if
    - There is a directed edge from $x$ to $y$

**Graphs as ADTs**

- Graphs can be used as abstract data types
- Two options for defining graphs
  - Vertices contain values
  - Vertices do not contain values
- Operations of the ADT graph
  - Create an empty graph
  - Determine whether a graph is empty
  - Determine the number of vertices in a graph
- Operations of the ADT graph (Continued)
  - Determine whether an edge exists between two given vertices; for weighted graphs, return weight value
  - Insert a vertex in a graph whose vertices have distinct search keys that differ from the new vertex’s search key
  - Insert an edge between two given vertices in a graph
  - Delete a particular vertex from a graph and any edges between the vertex and other vertices
  - Delete the edge between two given vertices in a graph
  - Retrieve from a graph the vertex that contains a given search key

**Implementing Graphs**

- Most common implementations of a graph
  - Adjacency matrix
  - Adjacency list
- Adjacency matrix
  - Adjacency matrix for a graph with $n$ vertices numbered $0, 1, \ldots, n-1$
    - An $n$ by $n$ array matrix such that $matrix[i][j]$ is
      - The weight that labels the edge from vertex $i$ to vertex $j$ if there is an edge from $i$ to $j$
      - 0 (or false) if there is no edge from vertex $i$ to vertex $j$

**Implementing Graphs**

- Adjacency matrix for a weighted graph with $n$ vertices numbered $0, 1, \ldots, n-1$
  - An $n$ by $n$ array matrix such that $matrix[i][j]$ is
    - The weight that labels the edge from vertex $i$ to vertex $j$ if there is an edge from $i$ to $j$
    - 0 (or false) if there is no edge from vertex $i$ to vertex $j$

---

**Figure 14-6**

(a) A directed graph and (b) its adjacency matrix

**Figure 14-7**

(a) A weighted undirected graph and (b) its adjacency matrix
Implementing Graphs

- Adjacency list
  - An adjacency list for a graph with \( n \) vertices numbered 0, 1, …, \( n - 1 \)
  - Consists of \( n \) linked lists
  - The \( i \)th linked list has a node for vertex \( j \) if and only if the graph contains an edge from vertex \( i \) to vertex \( j \)
    - This node can contain either
      - Vertex \( j \)’s value, if any
      - An indication of vertex \( j \)’s identity

Figure 14-8

a) A directed graph and
b) its adjacency list

Implementing Graphs

- Adjacency list for an undirected graph
  - Treats each edge as if it were two directed edges in opposite directions

Figure 14-9

a) A weighted undirected graph and b) its adjacency list

Implementing a Graph Class Using the JCF

- ADT graph not part of JCF
- Can implement a graph using an adjacency list consisting of a vector of maps
- Implementation presented uses TreeSet class

Graph Traversals

- A graph-traversal algorithm
  - Visits all the vertices that it can reach
  - Visits all vertices of the graph if and only if the graph is connected
    - A connected component
      - The subset of vertices visited during a traversal that begins at a given vertex
  - Can loop indefinitely if a graph contains a loop
    - To prevent this, the algorithm must
      - Mark each vertex during a visit, and
      - Never visit a vertex more than once
Graph Traversals

Figure 14-10
Visitation order for a) a depth-first search; b) a breadth-first search.