Hashing

Ming-Hwa Wang, Ph.D.
COEN 279/AMTH 377 Design and Analysis of Algorithms
Department of Computer Engineering
Santa Clara University

Direct-address Tables
When the universe $U$ of keys is reasonably small and no two elements have the same key, use an array or direct-address table that each position or slot corresponds to a key in the universe.

Hash Table ADT
- $O(1)$ for find, insert and delete
- ordering information among elements are not support efficiently
- an array of fixed size, keys and associated values
- load factor $\lambda$, the ratio of the number of elements in the table to the table size

Hashing function
- $H(key) = i$, where $0 \leq i < TableSize$
- TableSize better be prime number
- requirements
  - simple to compute, i.e., $O(1)$
    - use shift instead of multiply
    - use subtract instead of division/mod
    - use bitwise XOR instead of addition
  - simple uniform hashing: distribute the keys evenly
  - hash functions interpret each key as a natural number
  - division method: $H(key) = key \mod TableSize$
  - multiplication method: $H(key) = \lfloor TableSize(A \times key - \lfloor A \times key \rfloor) \rfloor$, where $A \approx (5^n - 1) / 2$, the TableSize can be power of 2
  - universal hashing: choose the hash function randomly from a collection of hash functions, each pair of distinct keys $x, y \in U$, the number of hash functions for which $H(x) = H(y)$ is precisely (number of hash functions) / TableSize

Collision
- key1 $\neq$ key2 and $H(key1) = H(key2) \Rightarrow$ collision
- Collision resolution
  - random collision resolution strategy
  - mean value of insertion time: $I(\lambda) = (1/\lambda) \ln(1/(1-\lambda))$

Open Hashing (Separate Chaining)
- keep a list of all elements that hash to the same value
  - stack is better (access locality)
  - can use list, binary search tree or another hash
  - search time = hashing time + traverse list time
  - $\lambda$ is the same as the average length of the list, close to 1 is better

Close Hashing (Open Addressing)
- cells $h_0(k), h_1(k), h_2(k), \ldots$ are tried in succession where $h_i(k) = (H(k) + F(i)) \mod TableSize$ with $F(0) = 0$.
- function $F$ is the collision resolution strategy
- $\lambda < 0.5$ is better
- can't support standard deletion, use lazy deletion
  - Linear Probing
    - $F(i) = i$, trying sequentially for empty cell
    - primary clustering
  - Quadratic Probing
    - $F(i) = i^2 = F(i - 1) + 2i - 1$
    - no guarantee of finding an empty cell if $\lambda > 0.5$ or TableSize is not prime
    - secondary clustering
    - Double Hashing $F(i) = h_2(k)$

Perfect hash functions
- a perfect hash function is a hash function $H$ such that $H(k_i) \neq H(k_j)$ for all distinct $i$ and $j$ (i.e., no collision).
- it is difficult to find a perfect hash function unless the set of keys is static and is frequently search, e.g., keywords in programming languages.
- minimal perfect hash function: perfect hash function for a set of $N$ keys in a table of only $N$ position.

Rehashing
- $O(n)$ - expensive
- however, there must have been $N/2$ inserts prior to the last rehash, so it essentially adds a constant to each insertion
- when to rehash for close hashing
  - to rehash as soon as the table is half full
  - to rehash when an insertion fails
  - to rehash when the table reach a certain load factor

Extendible Hashing
- directory $D$: the number of bits used by the root number of entries is $2^D$
- \( d_L \) is the number of leading bits that all the elements of some leaf \( L \) have in common
- when insert in a leaf which is already full, split the leaf into 2 leaves and increase the directory size.
- all of the leaves not involved in the split are now pointed to by 2 adjacent directory entries
- if more than \( M \) duplicates, then this doesn't work, where at most \( M \) records fit in one disk block
- expected number of leaves is \((N/M)\lceil e\rceil\)
- average leaf is in \(2 \sim 0.69\) full
- expected size of directory is \(O(N^{1+1}/M/M)\)

**Ordered Hash Tables**

- combining ordered keying with hashing. Given a file or table of data containing \( N \) distinct keys \( K_1, K_2, ..., K_n \), the search problem consists of taking a given argument \( K \) and determining whether or not \( K = K_i \) for some \( i \). If \( K \) is not in the table, we sometimes want to put it in (the insert problem). Let the hash address of \( K \) is \( h(K) \), the hash increment of \( K \) is \( i(K) \), \( 0 \leq h(K) < M, 1 \leq i(K) < M \), and \( i(K) \) is relative prime to \( M \).  

  **Searching algorithm:** searching in an ordered hash table.
  1. \( j \leftarrow h(K) \)
  2. if \( T_j = K \), return successfully
  3. if \( T_j < K \), return unsuccessfully
  4. \( j \leftarrow j - i(K), \) if \( j < 0, j \leftarrow j + M \)
  5. goto 2

- **Insertion algorithm:** insertion into an ordered hash table. Assume that \( K = T_j \) for \( 0 \leq j < M \), and that \( N \leq M - 2 \).
  1. \( j \leftarrow h(K) \)
  2. if \( T_j = 0, T_j \leftarrow K \) and terminate
  3. if \( T_j < K \), interchange \( T_j \leftarrow K \)
  4. \( j \leftarrow j - i(K), \) if \( j < 0, j \leftarrow j + M \)
  5. goto 2

**Brent Hashing and Passbit**

- when collision happens, either try insert the new key in next empty slot using double hashing or replace the existing key with the newly inserted key and try insert the existing key to its next empty slot using its own double hashing
- the decision of replacing or not is depends on the incremental cost
- add one extra bit (passbit) for each entry in the hash table, which is initialized to false and when collision happens, change the passbit to true, which can speed up future search
- divide entries into group and each group has a passbit to save memory

**Binary Tree Hashing**

- when a key to be added to the table hashes to a location already occupied, perform a breadth first search of the binary tree generated by these locations and subsequent rehashes of the keys encountered until an empty location is found (sampling with replacement)
- let each node shows tries for hashing with table index and key
  
  \[
  \begin{array}{ccc}
  1^\text{st} a & 10 b & \backslash \\
  2^\text{nd} & a & 7 d \backslash \\
  3^\text{rd} & a & 9 c \backslash
  \end{array}
  \]

  Then move key \( b \) to empty node indexed by 4, and put key \( a \) in node indexed by 10 at root

**Bloom Filter**

- space/time trade-off in hash coding with allowable errors. By allowing a small number of test messages to be falsely identified as members of the given set will permit a much smaller hash area to be used without increasing the reject time. Assume we are storing a set of \( N \) messages, each \( b \) bits long, and the allowable error probability is \( p (1 > p > 2^{-b}) \). One possible application is automatic hyphenation.

  There are two methods:
  - We organize the hash area into \( M \) cells with \( M > N \). Having decided on the size of cell, say \( c < b \), chosen so that the expected fraction of errors will be close to and smaller than \( p \), the hash area is organized into cells of \( c \) bits each. Then each message is coded into a \( c \)-bit code (not necessary unique), and those codes are stored and tested.
  - The hash area is considered as \( h \) (from 0 to \( h-1 \)) individual addressable bits instead of cells/buckets, all bits in the hash area are set to 0 initially. Next, each message in the set to be stored is hash coded into a number of distinct bit addresses, say \( a_1, a_2, ..., a_c \). Finally, all \( c \) bits addressed by \( a_1 \) through \( a_c \) are set to 1. To test a new message a sequence of \( c \) bit addresses is generated. If all \( c \) bits are 1, the new message is accepted. If any of these bits is zero, the message is rejected.

**Applications**

- compiler: symbol table
- graph theory: node with real names instead of numbers
- game-playing programs: transposition table
- on-line spelling checkers