Hash Tables – Outline

• Definition
• Hash functions
• Open hashing
• Closed hashing
  – collision resolution techniques
• Efficiency
Overview

• Implementation style for the Table ADT that is good in a wide range of situations is the hash table
  – efficient Insert, Delete, and Search operations
  – difficult Sorted Traversal
  – efficient unsorted traversal

• Good approach as long as sorted output comparatively rare in the total set of hash table operations
Definition

• Hash table is defined by:
  – set of records \( R = \{ r_1, r_2, \ldots, r_n \} \) stored by the table
  – set of input keys \( K = \{ k_1, k_2, \ldots, k_n \} \), \( n \geq 0 \) that can be associated with records \((k_x, r_y)\)
• Array of buckets \( B[0 \ldots m-1] \): each array element is capable of holding one or more \((k_x, r_y)\) pairs
• Hash Function \( H: K \rightarrow \{0, 1, \ldots, m-1\} \)
  – for any given \((k_x, r_y)\), \( B[H(k_x)] \) is the designated storage location for \((k_x, r_y)\)
• Collision resolution scheme
  – when \((k_x, r_y)\) and \((k_a, r_b)\) map to the same bucket under \( H \), this scheme determines where the second record is stored
Definitions

• An Array of buckets $B[0 \ldots m-1]$ holds all data managed by the hash table

• Open or External Hashing
  – bucket locations store pointers (references) to record pairs $(k_x, r_y)$
  – colliding records stored in a linked list

• Closed or Internal Hashing
  – buckets store actual objects
  – colliding records stored in other bucket locations

• Note that the associated keys may be implicit rather than explicitly stored
Hash Functions

• $H(i) = i$
  – reduces the hash table to an array

• Selecting digits
  – choose some subset of digits in a large number
    • specific slice or positions

• Folding
  – take digits or slices of a number and add them together with roll-over

• $H(i) = i \mod m$ – where $m$ is Hash Table size
  – choosing $m$ as a prime number is popular for an “even distribution of keys”
Hash Function – 2

• Strings are a common search key in many cases
  – convert string to an integer
  – \( H(\text{string}) \rightarrow \text{integer} \)

• Approaches
  – add characters or slices of characters together as n-bit unsigned numbers with the sum rolling over within x-bits
    • bit shifting to form numbers possible
    • x-bits chose for table size or x modulo m
  – several other options possible
Open Hashing

• Example: take a hash table size of 7 (prime) and a hash function \( h(x) = x \mod 7 \)
  
  – insert 64, 26, 56, 72, 8, 36, 42

• If data set is large compared to hash table size, or the hash function clusters data, then length of the list holding the bucket contents can be significant
  
  – sorted list will reduce the average failure time
    • can identify failure before the end of the list
  – use binary search tree instead of list
    • why not a BST for the whole data set?
  – use second Hash table
Open Hashing – 2

• Advantages of Open Hashing with chaining
  – simple in concept and implementation
  – insertion is always possible

• Disadvantages of hashing with chaining
  – unbalanced distribution decreases efficiency
    • $O(n)$ for a linked list, $O(\log n)$ for a BST
  – greater memory overhead
  – higher execution overhead of stepping through pointers
Closed Hashing

• Closed hashing with Open addressing
  – storing all data items within single hash table, but “open” up the address assigned to item on collision
• Hash table of size m can hold at most m items
• Only a “perfect” hash function will distribute m items to m different table elements
  – collisions will generally occur before table is full
• Collision resolution is thus crucial to efficient use of closed hash tables
Closed Hashing – Collision Resolution

• Create a sequence of collision resolution functions
  – \( h_0(x) \) is base hash function
  – \( h_1(x) \) used to find first alternate storage location after a collision
  – \( h_2(x) \) used to find the next alternate if first alternate is occupied
• Each \( h_i(x) \) must be guaranteed to choose different table locations
• Hash function series should ideally check all table locations
Collision Resolution – Linear Probing

• Search hash table sequentially starting from the original location specified by the hash function
  \[ h_i(x) = (h_0(x) + i) \mod m, \forall i > 0 \]

• Insert 64, 26, 56, 72, 8, 36, 42 in an empty table of size 7

• Fragile – causes primary clusters by occupying adjacent table locations
  – similar to long chains in open hashing
Collision Resolution – Quadratic Probing

• Spread probed locations across the table
  \[ h_i(x) = (h_0(x) + i^2) \mod m, \forall i > 0 \]
• Example: Insert 64, 26, 56, 72, 8, 36, 42
• Series of probed locations is not guaranteed to cover the whole table without duplication
• Closed hashing schemes can fail even though the table is not full
  – and secondary clusters may form
  – if the probing scheme will not visit all table locations and distribute probes “evenly” over 0..m
Collision Resolution –
Linear Probing with Fixed Increment

• \( h_i(x) = (h_0(x) + (i \times FI)) \mod m, \forall i > 0 \)
  – FI is relatively prime to m
  – linear probing will visit all table locations without repeats

• X is relatively prime to Y iff GCD(X,Y) = 1
Collision Resolution – Double Hashing

• Use a second hash function ($h'(x)$) to generate the probe sequence used after a collision
  
  $h_i(x) = (h_0(x) + (ih'(x))) mod m, \forall i > 0$
  
  – Use $h'(x)=R - (x mod R)$, where $R < m$ is prime

• Example: $m=7$, $R=5$, insert 64,26,56,72,8,36,42
Closed Hashing -- Deletions

- Example: Insert 64, 56, 72, 8 using linear probing
  - delete 64; delete 8
- Deletion along the probing path from A → B creates a problem because the empty cell could be there for two reasons
  - no further elements exist along this probing sequence
  - deletion of an item along the sequence took place
- Two types of empty buckets
  - bucket has always been empty (AE) (flag 0)
  - bucket emptied by deletion (ED) (flag 1)
Closed Hashing -- Deletions

• During a probing sequence,
  – if an AE bucket is found, searching can stop
  – if an ED bucket is found, searching must continue

• Closed Hashing is thus subject to a form of “fatigue”
  – as cells are deleted, probing sequences generally lengthen as the probability of encountering ED cells increases
  – failed searches get more expensive because they cannot terminate until
    • an AE cell is found
    • all cells of the table can be visited
Closed Hashing

• **Advantages of Closed Hashing with Open Addressing**
  – lower execution overhead as addresses are calculated rather than read from pointers in memory
  – lower memory overhead as pointers are not stored

• **Disadvantages**
  – more complex than chaining
  – can degenerate into linear search due to primary or secondary clustering
  – Delete and Find operations are more complex
  – Insert is not always possible even though the table is not full
  – Delete can increase probe sequence length by making search termination conditions ambiguous
The Efficiency of Hashing

• An analysis of the average-case efficiency
  – Load factor $\alpha$
    • 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

Successful search

Unsuccessful search

Average number of comparisons

\[ \alpha \]

Linear probing

Quadratic probing, double hashing

Separate chaining

Linear probing

Quadratic probing, double hashing

Separate chaining
Summary

• Hash Tables are useful and efficient data structures in a wide range of applications
• Open hashing with chaining is simple, easy to implement, and usually efficient
  – length of the chains is key to performance
• Closed hashing with various approaches to generating a probe sequence can also be efficient
  – lower space and computation overhead
  – more complex implementation
  – performance is sensitive to probe sequence
• Monitoring load factor and other hash-table behavior parameters is important in maintaining performance