Quadratic Probing

- Linear probing:
  - Insert item \( (k, e) \)
  - \( i = h(k) \)
  - \( A[i] \) is occupied
  - Try \( A[(i+1) \mod N] \): used
  - Try \( A[(i+2) \mod N] \) and so on until an empty bucket is found

- Quadratic probing
  - \( A[i] \) is occupied
  - Try \( A[(i+1) \mod N] \): used
  - Try \( A[(i+2^2) \mod N] \)
  - Try \( A[(i+3^2) \mod N] \) and so on

- May not be able to find an empty bucket if \( N \) is not prime, or the hash table is at least half full

Double Hashing

- Double hashing uses a secondary hash function \( d(k) \) and handles collisions by placing an item in the first available cell of the series \( (i + jd(k)) \mod N \) for \( j = 0, 1, \ldots, N-1 \)
- The secondary hash function \( d(k) \) cannot have zero values
- The table size \( N \) must be a prime to allow probing of all the cells

Double Hashing (cont.)

- \( d(k) \) should be chosen to minimize clustering
- Common choice of compression map for the secondary hash function:
  - \( d(k) = q - k \mod q \) where
    - \( q < N \)
    - \( q \) is a prime
  - The possible values for \( d(k) \) are \( 1, 2, \ldots, q \)

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - \( N = 13 \)
  - \( h(k) = k \mod 13 \)
  - \( d(k) = 7 - k \mod 7 \)
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

Comparing Collision Handling Schemes

- Separate chaining:
  - simple implementation
  - faster than open addressing in general
  - using more memory

- Linear probing: items are clustered into contiguous runs.

- Quadratic probing: secondary clustering.

- Open addressing:
  - using less memory
  - slower than chaining in general
  - more complex removals

- Double hashing: distributes keys more uniformly than linear probing does.
Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take \( \Theta(N) \) time.
- The worst case occurs when all the keys inserted into the map collide.
- The load factor \( \alpha = \frac{n}{N} \) affects the performance of a hash table.
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is \( 1 + \frac{1}{1 - \alpha} \).
- The expected running time of all the dictionary ADT operations in a hash table is \( \Theta(1) \).
- In practice, hashing is very fast provided the load factor is not close to 100%.
- Applications of hash tables:
  - Small databases
  - Compilers
  - Browser caches
  - Converting non-integer keys to integers

Load Factors and Rehashing

- Load factor: \( \alpha = \frac{n}{N} \)
- Experiments and average-case analyses suggest:
  - Separate chaining: \( \alpha < 0.9 \)
  - Open addressing: \( \alpha < 0.5 \)
- When \( \alpha \) goes above the specified threshold, rehash to get a new (smaller) \( \alpha \).
- Allocate a new array whose size is at least double the previous size.
- Define a new hash function (or new parameters).
- Re-insert all items into the new array using the new hash function.
- Analyzing the rehashing cost: similar to that of vector growth.

Keys That Are Not Integers

- When the keys are not integers, we need to convert them to integers before hashing.
- A hash function is usually specified as the composition of two functions:
  - Hash code map: \( h_1: \text{keys} \rightarrow \text{integers} \)
  - Compression map: \( h_2: \text{integers} \rightarrow [0, N - 1] \)
- The hash code map is applied first, and the compression map is applied next on the result, i.e., \( h(x) = h_2(h_1(x)) \).
- The goal of the hash function is to “disperse” the keys in an apparently random way.

Hash Code Maps

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects).
  - Good in general, except for numeric and string keys.
- Integer cast:
  - We reinterpret the bits of the key as an integer.
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java).
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and sum the components (ignoring overflows).
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java).

Hash Code Maps (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits).
  - We evaluate the polynomial \( p(x) = x_k + x_{k+1}z + x_{k+2}z^2 + \ldots + x_{n-1}z^k \) at a fixed value \( z \), ignoring overflows.
  - Especially suitable for strings (e.g., the choice \( z = 33 \) gives at most 6 collisions on a set of 50,000 English words).
- Polynomial \( h(x) \) can be evaluated in \( \Theta(1) \) time using Horner’s rule:
  - The following polynomials are successively computed, each from the previous one in \( \Theta(1) \) time:
    - \( p_0(z) = x_{n-1} \)
    - \( p_1(z) = x_{n-2} + zp_0(z) \)
    - \( p_2(z) = x_{n-3} + zp_1(z) \)
    - \( \ldots \) 
  - We have \( p(z) = p_{n-1}(z) \).
  - Good \( z \) values: 33, 37, 39, 41.

Summary

- Purpose of hash tables:
  - To obtain \( O(1) \) expected query time using \( O(n+N) \) space.
  - If the keys are not integers, map them to integers using a hash code map.
  - Map integer keys to the hash table entries using a compression map function.
- If collision occurs, use one of the collision handling schemes, taking into account available memory space.
- If the load factor \( \alpha = \frac{n}{N} \) approaches the specified threshold, rehash.