Hash Tables

Outline and Reading

- Hash functions and hash tables (§2.5.2)
- Hash function details
  - Hash code map (§2.5.3)
  - Compression map (§2.5.4)
- Collision handling (§2.5.5)
  - Chaining
  - Linear probing
  - Double hashing

Hash Functions and Hash Tables

- A hash function \( h \) maps keys of a given type to integers in a fixed interval \([0, N - 1]\).
- Example: \( h(x) = x \mod N \) is a hash function for integer keys.
- The integer \( h(x) \) is called the hash value of key \( x \).
- The goal of a hash function is to uniformly disperse keys in the range \([0, N - 1]\).

Hash Functions

- 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

- Component sum:
  - We interpret 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).
  - 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).

Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer.
- Our hash table uses an array of size \( N = 10,000 \).
- We use chaining to handle collisions.
- Example:
  - We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer.
  - Our hash table uses an array of size \( N = 10,000 \).
  - We use chaining to handle collisions.

Hash Code Maps

- 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 Functions

- 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 (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 $p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n$ 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 $p(z)$ can be evaluated in $O(n)$ time using Horner’s rule:**
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
  
  
  

**Compression Maps**

- **Division:**
  - $h_1(y) = y \mod N$
  - The size $N$ of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

- **Multiply, Add and Divide (MAD):**
  - $h_2(y) = (qy + b) \mod N$
  - $a$ and $b$ are nonnegative integers such that $a \mod N \neq 0$
  - Otherwise, every integer would map to the same value

**Search with Linear Probing**

- **Consider a hash table $A$ that uses linear probing**

  **findElement$(k)$**
  - We start at cell $A[i]$ and probe the consecutive locations until one of the following occurs
    - An item with key $i$ is found, or
    - An empty cell is found, or
    - $N$ cells have been unsuccessfully probed
  - **Algorithm findElement$(k)$**
    
    ```
    i <- b(k)  
p <- 0  
repeat  
  c <- A[i]  
  if c.key == k  
    return c.element  
  else if c.key != k  
    return NO_SUCH_KEY  
    i <- (i + 1) \mod N  
  else  
    p <- p + 1  
    i <- (i + p) \mod N  
until p = N
    return NO_SUCH_KEY
    ```

**Linear Probing**

- **Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell**
- **Each table cell inspected is referred to as a “probe”**
- **Colliding items lump together, causing future collisions to cause a longer sequence of probes**

**Updates with Linear Probing**

- **To handle insertions and deletions, we introduce a special object, called AVAILABLE, which replaces deleted elements**
  - **removeElement$(k)$**
    - We search for an item with key $k$
    - If such an item $(k, a)$ is found, we replace it with the special item AVAILABLE
      - and we return element $a$
    - Else, we return NO_SUCH_KEY

- **insertItem$(k, a)$**
  - We throw an exception if the table is full
  - We start at cell $A[i]$
    - We probe consecutive cells until one of the following occurs
      - A cell is found that is either empty or stores AVAILABLE
      - $N$ cells have been unsuccessfully probed
    - We store item $(k, a)$ in cell $i$

**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**

- **Common choice of compression map for the secondary hash function:**
  - $d_1(k) = q - k \mod q$
  - Where $q < N$
  - $q$ is a prime
  - The possible values for $d_1(k)$ are $1, 2, \ldots, q$
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.

Example of Double Hashing

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take \( O(n) \) time.
- The worst case occurs when all the keys inserted into the dictionary collide.
- The load factor \( \alpha = n/N \) 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 \( 1/(1 - \alpha) \).
- In practice, hashing is very fast provided the load factor is not close to 100%.
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

The expected running time of all the dictionary ADT operations in a hash table is \( O(1) \).