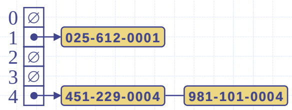
Hash Tables



Outline and Reading

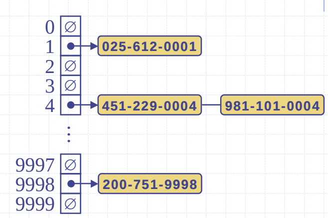
- Hash functions and hash tables (§2.5.2)
- Hash function details
  - Hash code map (§2.5.3)
  - Compression map (§2.5.4)
- Collision handling (§2.5.5)
  - Chaining
  - Linear probing
  - Double hashing

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:
  $h(x) = x \bmod N$
  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 $[0, N-1]$
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$
- A collision occurs when two keys in the dictionary have the same hash value
- Collision handing schemes:
  - Chaining: colliding items are stored in a sequence
  - Open addressing: the colliding item is placed in a different cell of the table

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 $N = 10{,}000$ and the hash function $h(x) =$ last four digits of x
- We use chaining to handle collisions



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code map:

  h_1: keys $\rightarrow$ integers

  Compression map:

  h_2: 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 we 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)
    $a_0\, a_1 \ldots a_{n-1}$
  - We evaluate the polynomial
    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$
    $\ldots + a_{n-1} z^{n-1}$
    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
    $p_0(z) = a_{n-1}$
    $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$
    $(i = 1, 2, \ldots, n-1)$
- We have $p(z) = p_{n-1}(z)$

Compression Maps

- Division:
  - $h_2(y) = y \bmod N$
  - 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):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

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
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Search with Linear Probing

- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm findElement(k)
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅
      return NO_SUCH_KEY
    else if c.key () = k
      return c.element()
    else
      i ← (i + 1) mod N
      p ← p + 1
  until p = N
  return NO_SUCH_KEY
```


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, o) is found, we replace it with the special item ***AVAILABLE*** and we return element o
  - Else, we return ***NO_SUCH_KEY***
- insert Item(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores ***AVAILABLE***, or
    - N cells have been unsuccessfully probed
  - We store item (k, o) 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
  $(i + jd(k)) \bmod N$
  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:
  $d_2(k) = q - k \bmod q$
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
  $1, 2, \ldots, q$

Example of Double Hashing

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

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

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
  $1 / (1 - \alpha)$
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches