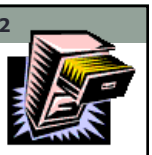
HASH TABLES (PART 1)

EECS 2011

1

The Map ADT



- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

2

3

Hash Tables

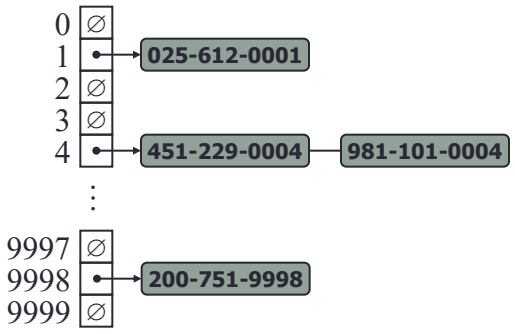
- Balanced BST (e.g., AVL trees): $O(\log N)$ for insertion, deletions and searches.
- Hashing is a technique used for performing insertions, deletions and searches in *constant average* time (i.e., $O(1)$ expected time)
- A hash table data structure consists of:
  - Hash function ***h***
  - Array of size ***N*** (bucket array)

3

4

Example

- We design a hash table for a dictionary storing items (SIN, Name), where SIN (social insurance number) is a ten-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) = x \bmod N$
- We use chaining to handle collisions



4

5

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]
- A hash table for a given key type consists of
  - Hash function ***h***
  - Array of size ***N***
- 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

5

6

Design Issues

- Hash functions
  - Converting a key to an index in the hash table
- Collision handling
  - Separate chaining
  - Probing (open addressing)
    - Linear probing
    - Quadratic probing
    - Double hashing
- Table size (should be a prime number)

6

Hash Functions

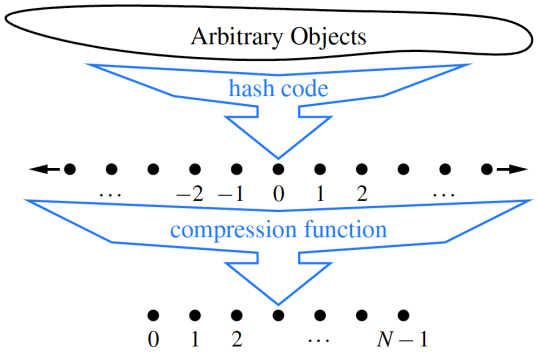


Figure 10.5: Two parts of a hash function: a hash code and a compression function.

7

Hash Functions

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

  h_2: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function 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.

8

Hash Codes

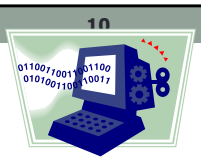
- To "transform" an arbitrary key (e.g., words in an English dictionary) to an integer.

  h_1: keys $\rightarrow$ integers

- Should avoid collisions as much as possible

9

Hash Codes



- Treating the bit representation as an integer
  - 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)
    - Ex: Float.floatToIntBits(x)
- 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), or XOR the components.
  - 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)

10

11

Hash Codes for Keys That Are Strings

- We need to convert a string to an integer before hashing.
- One option is to add up the ASCII values of the characters in the string.
  - Is this a good strategy?
- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits).

    $x_0 x_1 \ldots x_{n-1}$
  - We evaluate the polynomial

    $p(z) = x_0 + x_1 z + x_2 z^2 + \ldots + x_{n-1} z^{n-1}$

    at a fixed value z, *ignoring overflows*.

11

12

Polynomial Accumulation

- 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

    $p_0(z) = x_{n-1}$

    $p_i(z) = x_{n-i-1} + z p_{i-1}(z)$

    $(i = 1, 2, \ldots, n-1)$
  - We have $p(z) = p_{n-1}(z)$
  - https://www.math10.com/en/algebra/horner.html
- Good z values: 33, 37, 39, 41.
  - Especially suitable for strings
    - $z = 33$ gives at most 6 collisions on a set of 50,000 English words.

12

13

Hash Functions

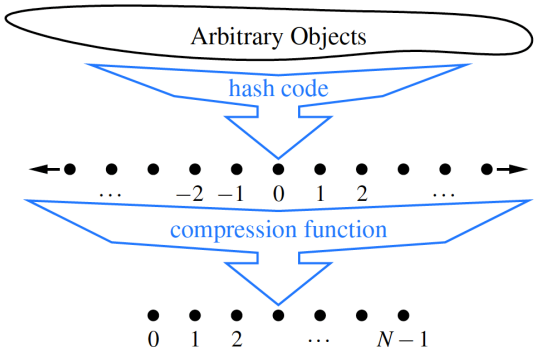


Figure 10.5: Two parts of a hash function: a hash code and a compression function.

13

14

Compression Function: Division

- $h_2(y) = y \bmod N$
- The size N of the hash table is usually chosen to be a prime number to minimize the number of collisions.
- Example: inserting {200, 205, 210, 215, 220, . . . ,600} into a bucket array of size 100: each hash code collides with three others.
- Bucket array of size 101: no collision.
- Good compression function: probability of two different keys getting hashed to the same bucket is 1/N.

14

15

Compression Function: MAD

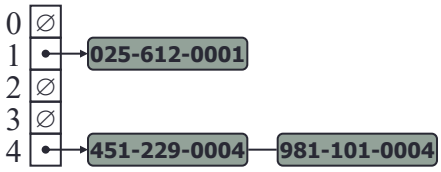
- Multiply, Add and Divide (MAD):
  - $h_2(y) = [(ay + b) \bmod p] \bmod N$
  - N: size of the bucket array
  - p is a prime number larger than N
  - a and b: integers chosen at random interval $[0, p-1]$, with $a > 0$
- Helps eliminate repeated patterns in a set of integer keys

15

16

Collision Handling

- Collisions occur when different elements are mapped to the same cell.
- **Separate Chaining**: let each cell in the table point to a linked list of entries that map there

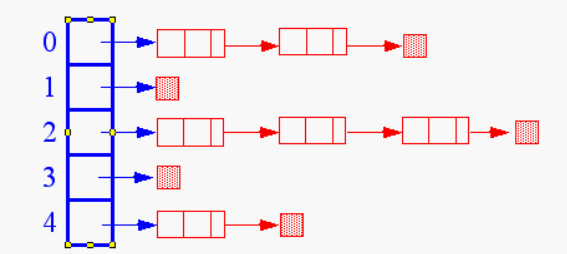


- Separate chaining is simple, but requires additional memory outside the table

16

17

Separate Chaining



- Use ***chaining*** to set up ***lists*** of items with same index
- The ***expected*** search/insertion/removal time is O(n/N), provided that the indices are uniformly distributed
  - N = hash table size
  - n = number of elements in the table
- If n/N is *O*(1), the expected running time is *O*(1)

17

18

Load Factor – Separate Chaining

- Define the load factor $\lambda = n/N$
  - n = number of elements in the hash table
  - N = hash table size (prime number)
- To obtain best performance with separate chaining, ensure $\lambda < 0.9$.
- As long as λ is *O*(1), insert, delete and search operations run in *O*(1) expected time.
- As we add more elements to the hash table, λ goes up $\Rightarrow$ rehashing (allocate a bigger table, define a new compression function, and put the elements into the new array).

18

Next lecture: Collision Handling

- Separate chaining
- Probing (open addressing): the colliding item is placed in a different cell of the table
  - Linear probing
  - Quadratic probing
  - Double hashing