

Dictionaries



Dictionary ADT

- Models a searchable collection of key-element items called **entries**
- Main operations: find, insert, remove
 - `findElement(k)`, `insertItem(k, o)`, `removeElement(k)`
 - `size()`, `isEmpty()`
 - `keys()`, `elements()`
- Applications:
 - address book
 - word-definition pairs
 - mapping host names to internet addresses (e.g., `www.cs16.net` to `128.148.34.101`)

Log File

- A **log file** is a dictionary implemented by means of storing items in an **unsorted sequence**
 - **insertItem** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
 - **findElement** and **removeElement** take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- Effective only for dictionaries of
 - small size or
 - when insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

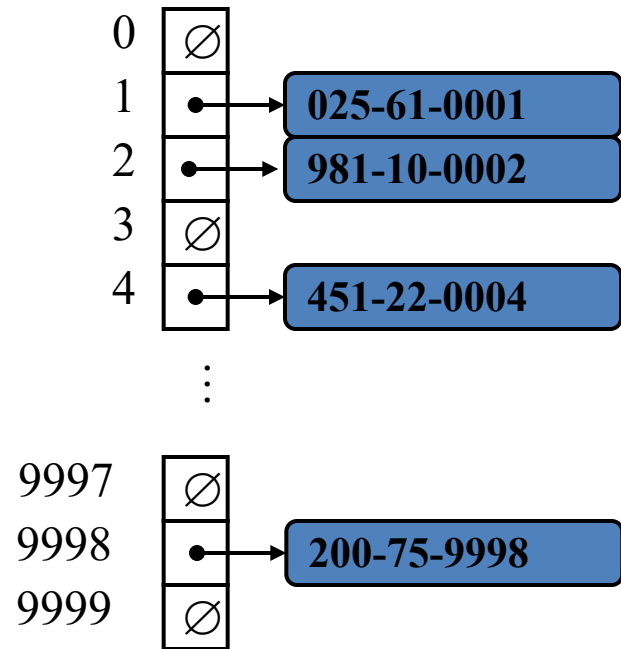
Hash Table -based Dictionaries

Hash Functions and Hash Tables

- A **hash table** for a given key type consists of
 - Array (called table) of size N
 - Hash function h
- A **hash function** h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
 - Ex: $h(x) = x \bmod N$ is a hash function for integer keys
 - The integer $h(x)$ is called the **hash value** of key x
- When implementing a dictionary with a hash table, the goal is to **store** item (k, o) at index $i = h(k)$

Example

- We design a hash table for a dictionary storing items (social security number, name)
- Our hash table uses an array of size $N = 10,000$ and the hash function $h(x) = \text{last four digits of } x$



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

$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to “disperse” the keys in an apparently random way

Hash Code Maps: keys \rightarrow integers

Memory address

- reinterpret the memory address of the key object as an integer
- default hash code of Java objects
- disadvantage: two key objects with equal value have different hash codes

Integer cast

- reinterpret bits of the key as an integer
- suitable for smaller keys (when number of bits in the key is at most the number of bits in an integer)

Hash Code Maps: keys \rightarrow integers

Component sum

- suitable for larger keys
- partition bits of the key into components of fixed length and sum the components
- disadvantage: many strings will have the same sum

$$h_I(k) = a_0 + a_1 + a_2 + \dots + a_{n-1}$$

Polynomial accumulation

- good for strings
- partition bits of the key into components of fixed length and evaluate the polynomial

$$h_I(k) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$$

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

- A **good hash function** guarantees the probability that two different keys have the same hash is $1/N$.
- The size N of the hash table is usually chosen to be a **prime**.
 - The reason involves number theory and is beyond the scope of this course

Division

- $h_2(y) = y \bmod N$
- disadvantage: repeated keys of the form $iN + j$ cause collisions

Multiply, Add and Divide (MAD)

- $h_2(y) = ((ay + b) \bmod p) \bmod N$
- This is a “good” hash function (continued next slide...)

Universal Hashing

- Recall that a good hash function guarantees the probability that two different keys have the same hash is $1/N$.
- A family of hash functions is **universal** if for any $0 \leq j, k \leq M-1$,
$$\Pr(h(j)=h(k)) \leq 1/N$$

Theorem: The set of all functions, h , as defined below, is universal.

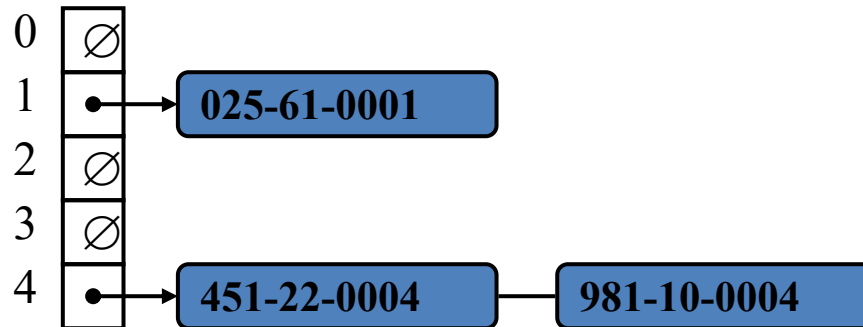
- Choose p as a prime between M and $2M$
- Randomly select $0 < a < p$ and $0 \leq b < p$
- a and b are nonnegative integers such that $a \bmod N \neq 0$
(otherwise, every integer would map to the same value b)
- Define $h(k)=((ak+b) \bmod p) \bmod N$

Collision Handling

Collisions occur when different elements are mapped to the same cell

Chaining

- each cell in the table points to a linked list of elements that map there
- simple, but requires additional memory outside the table



Open Addressing

- the colliding item is placed in a different cell of the table
- no additional memory, but complicates searching/removing
- common types: **linear probing**, quadratic probing, **double hashing**

Open Addressing: Linear Probing

- Placing the colliding item in the next (circularly) available table cell
try $A[(h(k) + i) \bmod N]$ for $i = 0, 1, 2, \dots$
- Colliding items cluster together, causing future collisions to cause a longer sequence of probes (searches for next available cell)

- 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

$$h(18) = 18 \bmod 13 = 5$$

$$41 \bmod 13 = 2$$

$$22 \bmod 13 = 9$$

$$44 \bmod 13 = 5$$

$$59 \bmod 13 = 7$$

$$32 \bmod 13 = 6$$

$$31 \bmod 13 = 5$$

$$73 \bmod 13 = 8$$

Search with Linear Probing

Consider a hash table A that uses linear probing

findElement(k)

- Start at cell $h(k)$
- Check 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)*

$i \leftarrow h(k)$

$p \leftarrow 0$

repeat

$c \leftarrow A[i]$

if $c = \emptyset$

return *NO_SUCH_KEY*

else if $c.key() = k$

return *c.element()*

else

$i \leftarrow (i + 1) \bmod N$

$p \leftarrow p + 1$

until $p = N$

return *NO_SUCH_KEY*

Updates with Linear Probing

A special object, called *AVAILABLE*, replaces deleted elements

- `removeElement(k)`
 - Search for an item with key k
 - If it is found, replace it with item *AVAILABLE* and return element
 - Else, return *NO_SUCH_KEY*
- `insertItem(k, o)`
 - Throw an exception if the table is full
 - Start at cell $h(k)$
 - Search consecutive cells until a cell i is found that is either empty or stores *AVAILABLE*
 - Store item (k, o) in cell i

Open Addressing: Double Hashing

- Use a secondary hash function $d(k)$ to place items in first available cell
try $A[(h(k) + id(k)) \bmod N]$ for $i = 0, 1, 2, \dots$
- $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

Example of Double Hashing

Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 1 + (k \bmod 7)$

Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	5	5		
41	2	7	2		
22	9	2	9		
44	5	3	5	8	
59	7	4	7		
32	6	5	6		
31	5	4	5	9	0
73	8	4	8	12	

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



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

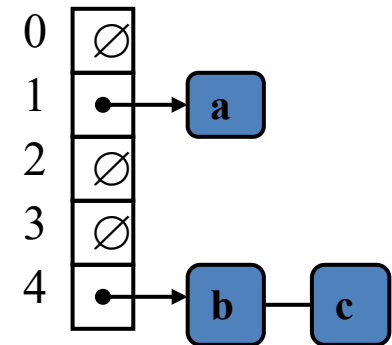
Performance of Hashing

- In the **worst case**, searches, insertions and removals on a hash table take $O(n)$ time
 - occurs when all inserted keys 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 number of probes for an insertion with **chaining** is $O(1 + \alpha)$
- 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%

Chaining vs. Open Addressing

Chaining

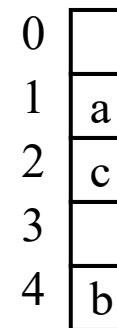
- Less sensitive to hash functions and load factor
- Supports $\alpha > 100\%$



Open Addressing

- Requires careful selection of hash function to avoid clustering
- Degrades past $\alpha > 70\%$
- Can't support $\alpha > 100\%$
- Better memory usage

$$h(a) = 1 \quad h(b) = 4 \quad h(c) = 4$$



Other

- You are given an array A of integers. Determine the integer that occurs most frequently in A .