VE281

Data Structures and Algorithms

Dictionary and Hashing

Announcement

- I have to go for a U.S. visa interview at 12:30 this Thursday. We will not have the 10-minute break for the class on this Thursday.

- The office hour of this Thursday is cancelled.

Review

- Queues
  - Methods: **enqueue**, **dequeue**, **size**, etc.
  - Implementations by linked lists
  - Implementation by arrays: circular array
  - Application: wire routing - lee’s algorithm
  - Deque: combination of stack and queue

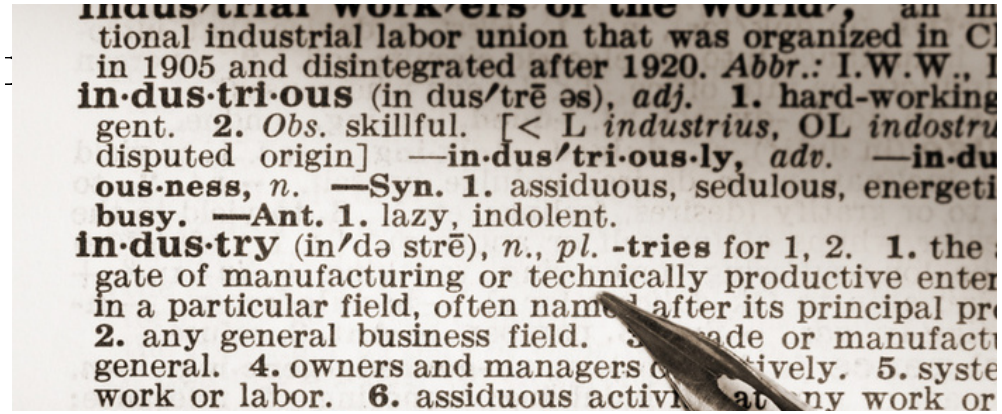
Outline

- Dictionary
- Basics of Hashing
- Hash Function

Dictionary

- How do you use a dictionary?
  - Look up a “word” and find its meaning.
- We also have an ADT of dictionary.
  - It is a collection of pairs, each containing a key and an element

    (key, element)
  - Differ



Dictionary

- Key space is usually more regular/structured than value space, so easier to search.

- Dictionary is optimized to quickly add **(key, element)** pair and retrieve element by key.

Methods

- **Element find(Key k)**: Return the element whose key is **k**. Return **Null** if none.
- **void insert(Key k, Element e)**: Insert a pair **(k, e)** into the dictionary. If the pair with key as **k** already exists, update its element.
- **Element remove(Key k)**: Remove the pair with key as **k** from the dictionary and return its element. Return **Null** if none.
- **int size()**: return number of pairs in the dictionary.

Example

- Collection of student records in the class
  - (key, element) = (student name, linear list of assignment and exam scores)
  - All keys are distinct

- Operations
  - Get the element whose key is John Adams.
  - Insert a record for the student whose name is Diana Ross.

Representation as a Linear List

$$\mathbf{L = (e_1, e_2, \ldots, e_N)}$$

- Each $\mathbf{e_i}$ is a pair **(key, element)**.

- 5-pair dictionary D = (a, b, c, d, e).
  - a = (aKey, aElement), b = (bKey, bElement), etc.

- Implementation using arrays or linked lists.

Runtime for Array Implementation

`Pair Array[MAXSIZE]:`

a	b	c	d			

- Unsorted array
  - find() $O(n)$
  - insert() $O(n)$: $O(n)$ to verify duplicate, $O(1)$ to put at the end
  - remove() $O(n)$: $O(n)$ to verify existence, $O(1)$ to exchange the "hole" with the last element
- Sorted array
  - find() $O(\log n)$: binary search
  - insert() $O(n)$: $O(\log n)$ to verify duplicate, $O(n)$ to insert
  - remove() $O(n)$: $O(\log n)$ to verify existence, $O(n)$ to remove

Can we do **`find`**, **`insert`**, and **`remove`** in $O(1)$ time?

Outline

- Dictionary
- Basics of Hashing
- Hash Function

Hashing and Hash Table

- Access table items by their keys in time that is relatively **constant regardless of their locations**.

- Main idea: use arithmetic operations, known as **hash function**, to transform keys into table locations.
  - The same key is always hashed to the **same** location.
  - Thus, **insert()** and **find()** are both directed to the same location in $O(1)$ time.

- **Hash table**: An array of **buckets**, where each bucket contains items as assigned by a hash

Ideal Hashing

- Uses a 1D array (or table) **`table[0:M-1]`**.
  - Each position of this array is a bucket.
  - A bucket can normally hold only one item.

- Uses a hash function **`h`** that converts each key **`k`** into an index in the range **`[0, M-1]`**.
  - **`h(k)`** is the home bucket for key **`k`**.

- Every item with **`key`** is stored in its home bucket **`table[h[key]]`**.

Ideal Hashing Example

- Pairs are: (22, a), (33, b), (3, c), (73, d), (85, e).
- Hash table is **table[0:7]** and table size is **M = 8**.
- Hash function is **key/11**.

(3, c)		(22, a)	(33, b)			(73, d)	(85, e)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Question: What is the time complexity for **find()**, **insert()**, and **remove()**?

O(1)

What Can Go Wrong?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3, c)		(22, a)	(33, b)			(73, d)	(85, e)

- Where does (35, g) go?
- Keys that have the same **home bucket** are **synonyms**.
  - 33 and 35 are synonyms with respect to the hash function **h(key)=key/11**.
- Problem: The home bucket for (35, g) is already occupied!
  - This is a "**collision**".

Collision and Collision Resolution

- Collision occurs when the hash function maps two or more items—all having different search keys—into the same bucket.

- What to do when there is a collision?

- Collision-resolution scheme: assigns distinct locations in the hash table to items involved in a collision.
- Two major schemes:
  - Separate chaining
  - Open addressing

Hash Table Issues

- Choice of the hash function.
- Collision resolution scheme.
- Size of the hash table and rehashing.

Outline

- Dictionary
- Basics of Hashing
- Hash Function

Hash Functions

- Hash function (**h(key)**) maps key to buckets in two steps:

1. Convert key into an integer in case the key is not an integer.
   - A function **t(key)** which returns an integer value, known as hash code.
2. Compression map: Map an integer (hash code) into a home bucket.
   - A function **c(hashcode)** which gives an integer in the range [0, M-1], where M is the number of buckets in the table.

- In summary, **h(key) = c(t(key))**, which gives an index in the table.

Hash Function Design Criteria

- Must compute a hash for every key.
- Must compute the same hash for the same key.
- Should be easy and quick to compute.
- Involves the entire search key.
- Scatters keys that differ slightly.
- Minimizes collision
  - Distributes keys evenly in hash table

- Good hash function = avoiding worst case
  - We cannot guarantee this, but can improve statistics by ensuring that buckets are used equally.

Map Non-integers into Hash Code

- String: use the ASCII (or UTF-8) encoding of each char and then perform arithmetic on them.

- Floating-point number: treat it as a string of bits.

- Images, viral code snippets, malicious Web site URLs: in general, treat the representation as a bit-string, using all of it or extracting parts of it (i.e., www.abc.com.cn).

Strings to Integers

- Simple scheme: adds up all the ASCII codes for all the chars in the string.
  - Example: t(“He”) = 72 + 101 = 173.

- Not good. Why?
  - Consider English words “post”, “pots”, “spot”, “stop”, “tops”.

Strings to Integers

- A better strategy: Polynomial hash code taking **positional** info into account.

$$t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$$

where a is a constant.

- If $a = 33$, the hash codes for "post" and "stop" are

$$t(\text{post}) = 112 \cdot 33^3 + 111 \cdot 33^2 + 115 \cdot 33 + 116 = 4149734$$
$$t(\text{stop}) = 115 \cdot 33^3 + 116 \cdot 33^2 + 111 \cdot 33 + 112 = 4262854$$

- This operation is also known as **folding**: **partition** the key into several parts and combine them in a **convenient** way

Strings to Integers

- $t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$
- Good choice of a for English words: 31, 33, 37, 39, 41
  - What does it mean for a to be a **good** choice? Why are these particular values **good**?
  - Answer: according to statistics on 50,000 English words, each of these constants will produce less than 7 collisions.
- In Java, its **`string`** class has a built-in **`hashCode()`** function. It takes $a = 31$. Why?
  - Multiplication by 31 can be replaced by a shift and a subtraction for **better performance**: **`31*i == (i << 5) - i`**

Hash function criteria: Should be easy and quick t

Compression Map

- Map an integer (hash code) into a home bucket.
- The most common method is by **modulo arithmetic**.

    homeBucket = c(hashcode) = hashcode % M

    where M is the number of buckets in the ha

	(22, a)	(79, e)	(3, c)		(33, b)	(55, d)
[0]	[1]	[2]	[3]	[4]	[5]	[6]

- Example: Pairs are (22,a), (33,b), (3,c), (55,d), (79,e). Hash table size is 7.

Uniform Hash Function

- Let **`keySpace`** be the set of all possible keys. A **uniform hash function** maps the keys in **`keySpace`** into buckets such that approximately the same number of keys get mapped into each bucket.

- Equivalently, the probability that a randomly selected key has bucket i as its home bucket is $1/M$, $0 \leq i < M$.

- A uniform hash function minimizes the likelihood of an collision when keys are selected at random.

Hashing by Modulo

- In practice, keys tend to be correlated.
- Because of this correlation, applications tend to have a bias towards keys that map into a specific set of integers.
  - For example, the keys of an application may more likely be mapped into odd (or even) integers.
- The choice of the divisor M will affect the distribution of home buckets.

Hashing by Modulo

- Suppose the keys of an application are more likely to be mapped into odd (or even) integers.

- When the divisor M is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - 20%14 = 6, 30%14 = 2, 8%14 = 8
  - 15%14 = 1, 3%14 = 3, 23%14 = 9
- The bias in the keys results in a bias toward either the odd or even home buckets.
  - The distribution of home buckets is not uniform!

Hashing by Modulo

- However, when the divisor is an odd number, odd (even) integers may hash into any home buckets.
  - 20%15 = 5, 30%15 = 0, 8%15 = 8
  - 15%15 = 0, 3%15 = 3, 23%15 = 8
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
  - Better chance of uniformly distributed home buckets.
- So do not use an even divisor.

Hashing by Modulo

- Similar **biased** distribution of home buckets is seen, in practice, when the divisor M is a multiple of prime numbers such as 3, 5, 7 ...
- The effect of each prime divisor p of M **decreases** as p gets **larger**.

- Ideally, choose M as a **large prime number**.
- Alternatively, choose M so that it has no prime factor smaller than 20.