9/25/2018 Week 4 Hash table

Bloom filter

Hash table

⇒ array that is indexed by the value in it.

Purpose: Maintain a possibly evolving set of stuff.
(transactions, people +associated data, IP addresses)

Insert: Add new record
Delete: delete existing record
Lookup: Check for a particular record

All by using key.
run by O(1).

Used often in "dictionary" structure but it is not maintaining the ordering of the elements.

- Constant time can be ensured only when implemented correctly, and data is non-pathalogical..

Application ① De-duplication

⇒ When new object x arrives
- lookup x in hash table H.
- if not found, Insert x into H.

Application ② The Two-sum problem

Input : Unsorted array A of n integers. Target sum t.
Goal : Determine whether or not there are two numbers x, y in A with $x+y=t$.

Naive solution : $O(n^2)$ time via exhaustive search.

Better : ① Sort A ($n\log(n)$ time)
② for each x in A,
look for $t-x$ in A via binary search.
$O(n \log n) \rightarrow$ Repeating lookup

⇒ hash table allow to do it in $O(n)$.
Because the lookup in hash is constant $O(1)$ and faster than binary search ($\log n$).

Hash ① Insert elements of into hash table H
② for each x in A, loopup $t-x$ in H.

⇒ Used for Symbol table in compilers, ~~bla~~ blocking network traffic, searching algorithms.
- Use hash table to avoid exploring any configulation (Node) more than once.

High level idea

Setup: Universe U (All the possibilites of the data)
e.g. all IP address, all names etc.
[Really big]

Goal: Want to maintain evolving set $S \subseteq U$
[Reasonable size]

Naive Solution

1. Array-based solution [indexed by U]
   - $O(1)$ operations but $\theta(|U|)$ space.
2. list-based solution
   - $O(|S|)$ spase but $\theta(|S|)$ ~~loop~~ lookup.

Solution:

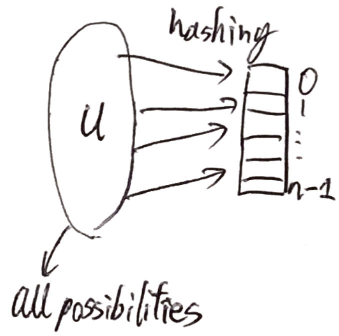
1. Pick n = # of "buckets" with $n \gtrsim |S|$.
   (for simplicity, assume $|S|$ doesn't vary too much)
   (In real number of elements chage dynamically.)
   (Usually it doubles the length of array)
2. Choose a hash function $h: U \rightarrow \{0, 1, 2, 3 \dots n-1\}$
   function to change key → position in array.
3. Use array A of length n, store x in $A[h(x)]$

⇒ Use element itself as a index

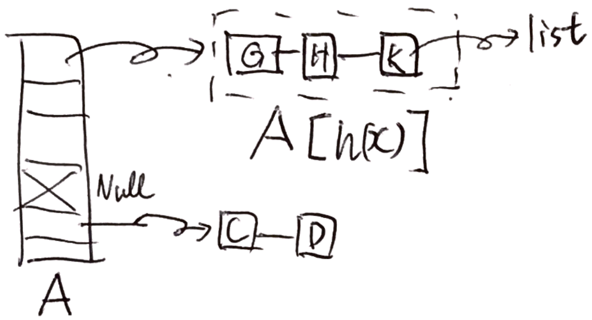
Issue (Collisions)

Collision: Distinct $x, y \in U$ such that $h(x) = h(y)$ ← hash function

Solution #1 (Separate) chaining



- Keep the linked list in each bucket.



- Given a key/object x, perform Insert/Delete/Loopup in the list in $A[h(x)]$.

Solution #2 : open addressing (Only one object per bucket)

- Try to find the open bucket if the bucket is already fall. Sequently probing the open address. (Probe sequence)
  $\{h_1(x), h_2(x), h_3(x) \cdots\}$
- e.g. linear probing (Check one after another)
  double hashing (Use two hashing)

⇒ Both right. Depend on the situation.
If space is premium: Open addressing.
If deletion is crucial function: Separate Chaining.

Running time of hash function in each case

- Hash table with chaining, Insert is $O(1)$. [Insert new object x at front of the list in A[h(x)]]
- Lookup is $O(\underbrace{\text{list length}}_{})$.

  → Could be anywhere from $\frac{m}{n}$ (Equal-length list; n → # of buckets) to m for m objects (all objects in same bucket).

⇒ The shorter is the better for list length.

⇒ Want to distribute the object equally in the buckets.
(Not only for Separate Chaining but for Open addressing too)

"Good" hash function

- Spread the data out for good performance.
  (Gold standard: Completely random hashing) ⇒ Impossible to actually implement.
- Run in constant time. (Hash function will be used everytime using either insertion/deletion/lookup.)
  ⇒ Should be easy to store/very fast to evaluate.

"Bad" hash function

e.g. keys = phone numbers (10-digits) $|U| = 10^{10}$

⇒ want to make buckets that size of $n = 10^3$. Have to "map" 10^{10} to 10^3.

- Terrible hash function: $h(x)$ = 1st 3 digits of x
- mediocre hash function: $h(x)$ = last 3 digits of x
  [Still vulnerable to patterns in last 3 digits].

Other bad example of hash function

e.g. keys = memory locations (will be multiples of a power of 2)

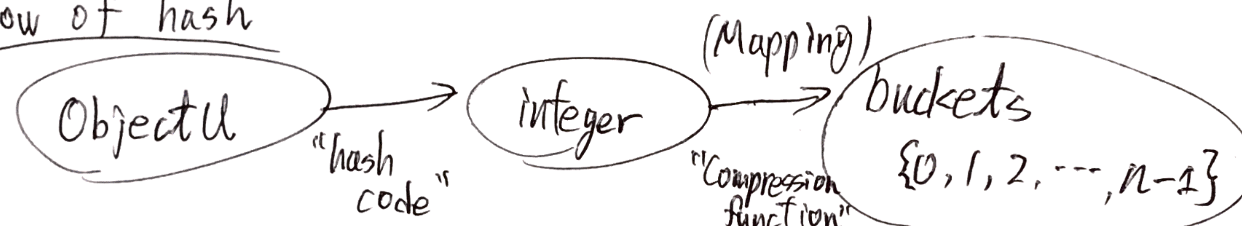
$n = 10^3$

- ~~Terrible~~ Terrible : $h(x) = x \% 1000$
  - ⇒ all odd buckets guaranteed to be empty. Waste of memory spaces.

⇒ Making good hash function is an art. Should be different in each application. New way of hashing is coming out every year.

Quick-and-Dirty hash function

Flow of hash



※ Sometimes (when the object is already int) 'hash code' step might be skipped.

- "hash code" : e.g. subroutine to convert string to integer. (Using ~~ASCII~~ ASCII code)
- "Compression func." : e.g. like mod n function (~~n % 1000~~) (% n function)

How to choose n = # of buckets? (Rule of thumb)

① Choose n to be a prime num. (within constant factor of # of object in table)
(If key is all divisible by 2, and hash function's modulous is also divisible by 2, it shares a common factor 2, so all the odd buckets remained empty)
⇒ By using "% prime" function, any bucket have chance to be filled.

② Not too close to power of 2

③ Not too close to power of 10

Universal hashing

load of the hash table

load factor $\alpha := \dfrac{\text{\# of objects in hash table}}{\text{\# of buckets of hash table}}$

1. $\alpha = 1$ or less to run operation in constant time. (Otherwise O(list length))
2. With open addressing, need to be $\alpha \ll 1$.

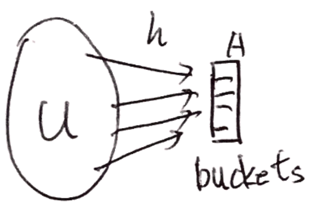
⇒ Controlling the load is important to keep the performance of hash function.

⇒ Usually increasing the buckets of hash table according to the # of objects in the hash table. (Double the length is typical implement

Pathological Data sets

⇒ There is no be-all-end-all hash function.

⇒ Every single hash function has a pathological data set!



⇒ Possible to make the data set that make all the data be stored in same bucket.
(e.g. Store all the data in A[31])

Hash's performance is not same in any input

Quick sort : $n \log n$
Merge sort : $n \log n$
DFS : n
BFS : n
hash function : ⇒ ?

⇒ Pathological data can be used for system attack.

how can we cope with the pathological data ?

① Use cryptographic hash function. (e.g. SHA-2)

⇒ Infeasible to reverse engineer a pathological data set.

② Use randomization.

- Design a family $\mathcal{H}$ of hash functions such that in avarage, "almost all" functions $h \in \mathcal{H}$ spread S out "pretty evenly"

⇒ Since randomization is real time, even reading the source code, cannot reverse engineer the pathological data.

→ Universal hashing (Randomized solution)

Part 1: Definition of "Good random hash function"

Part 2: Concreate example of simple + practical function

Part 3: Justification of definition of "Good random hash function" lead to "Good performance".

– Bloom Filter

Supported Operation

Purpose of using Bloom Filter

⇒ Fast insert and lookup. $\rightarrow$

Pros: ⇒ More space efficient than Hash table ↗

Cons: ⇒ ① Cannot store an associated object.
② No deletions
③ Small false positive probability.

– Application of Bloom Filter

Original: Early spell checker.
Cononical: List of forbidden passwords.
Modern: Network routers

} all doesn't care about the possibility of false positives.
⇒ If critical, use hash

– How it work?

Ingredients: ① Array of n bits. (So $\frac{n}{|S|}$ = # of bits per object in data set S. Let's say 8bits)
② k hash functions $h_1, h_2, \cdots h_k$ (k = Small constant like 3, 5)

Inserts: For $i = 1, 2, \cdots, k$
set $A[h_i(x)] = 1$ (No matter the existing bit in $A[h_i(x)]$)

Lookup: Return True ⟺ $A[h_i(x)] = 1$ for every $i = 1, 2, \cdots k$.

Note: No false negative. (If x has inserted, Lookup(x) guaranteed to succeed)
But: false positive if all k $h_i(x)$'s already get to 1 by other insertions.

How useful Bloom Filter is?

Intuition: Should be a trade-off between space and error prob.

⇒ Basically, the bigger the space Bloom Filter use, the smaller the false positive prob. is.

⇒ Way to set the right num. of k. (# of hash function)

b: Number of bits representing the object.

$$k \approx (\log 2) \cdot b$$

⇒ Prob. of error

$$\varepsilon \approx \left(\frac{1}{2}\right)^{(\ln 2) b} \quad \text{or} \quad b \gtrsim 1.44 \cdot \log_2 \frac{1}{\varepsilon}$$

e.g. with $b=8$, choose $k=5$ or 6, error prob. only $\approx 2\%$