Hashing

Can do

Search —
Insert — } O(1) on average.
Delete —

* not useful for
  → finding closest value
  → sorted data
  → prefix searching - Trie data structure
  } ALL / red black tree

Application of Hashing

1. Dictionaries
2. Database Indexing
3. Cryptography
4. caches
5. symbol table in Compiler / Interpreter
6. Routers
7. searching data from databases
8. many more

Direct address table

Imagine a situation where you have 1000 keys with values from (0 to 999), how would you implement following in O(1) time

1) search
2) Insert
3) Delete

Problems of direct Address table
1) large values cannot be addressed
2) floating values " " "
3) strings " " "

How hash function works?
→ should always map a large key to same small key
→ should generate values from 0 to m-1
→ should be fast O(1) for integers and O(len) for string of length 'len'
→ should uniformly distribute large keys from hash table slots

Example hash function

1) h(large-key) = largekey % m
2) for strings, weighted sum
   str = "abcd"
   str[0]x^0 + str[1]x^1 + str[2]x^2 + str[3]x^3
3) universal hash

should chosen in way it is prime number away from 10 powers.

Collision Handling

If we know keys in advance, then we can perfect hashing

If we dont know keys then we use one of the following

- Chaining
- open addressing
  - → linear probing
  - → Quadratic probing
  - → Double hashing

Chaining

what we do is we will take array of linked lists
whenever there is collision we add element in the end of linked list.

Performance:

m = no of slots in hash table
n = no of keys to be inserted
load factor(α) = n/m
Expected chain length = α
Expected Time to search = $(1+\alpha)$
Expected time to Insert/Delete = $O(1+\alpha)$

Data structures for storing chain

- Linked list
- Dynamic size Arrays (vectors in C++, Arraylist in Java, list in python)
- Self balancing BST (AVL Tree, Red black tree)
  - O(log d)
    - search
    - Insert
    - delete