Hash Table Data Structure

By Dr Shantanu Pathak

Hash Table Introduction

- Stores data in <Key,Value> format
- For efficient search on any size of data
- Insertion, deletion and Searching is very fast compared to array / other data structures

- How searching is made faster here ?
- By using Hashing function while inserting the elements
- Hashing function directly gives the location based on key
- So in single step one can find the element <key,value>

Hash Table Example

- There are 7 keys to be stored
- [23,56,21,45,78,99,77]
- Assume hash function **f = (key % 10)**
- So, hash value of every key is
- [3,6,1,5,8,9,7]

- **Find 77** key in Hash Table
- Calculate hash function f for 77 -> 7
- So, 77 will be found at index 7

- **Find 60** key in Hash Table
- Calculate hash function f for 60 -> 0 (Not found)

Index	Key
0	
1	**21**
2	
3	**23**
4	
5	**45**
6	**56**
7	**77**
8	**78**
9	**99**

Hash Table Performance

- Search -> O(1)
- Insert -> O(1)
- Delete -> O(1)

Performance depends on GOOD hashing Function

What is Good hashing Function?

fast

avoids same index for different keys (collision)

evenly spreads data across the whole table

Hash Table Performance

How following hash function will perform ?

1. Hash Table of size 30 , hash function is key % 10

2. Hash Table of Size 20, hash function is key % 40

Types of Hashing Functions

- 1. **Division method**
  - Mod with value n
- 2. **Multiplication Method**
  - Multiply with a constant
- 3. **Universal hashing**
  - Select a random hashing function from set of hashing functions every time

  - Java's HashMap Hashing function :
  - https://stackoverflow.com/questions/9364134/what-hashing-function-does-java-use-to-implement-hashtable-class
  - http://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/src/java.base/share/classes/java/util/HashMap.java
  - Good Hash Function Examples for Strings:
  - https://www.sparknotes.com/cs/searching/hashtables/section2/

Collision and Its Resolution Techniques

- What happens if hash function returns same index for multiple keys?
- Its called as **collision**

- Ex. F= (key % 10)
- then key 34 and 24 will have same index 4

- F = (key % 19)
- Will there be collision in this hash function

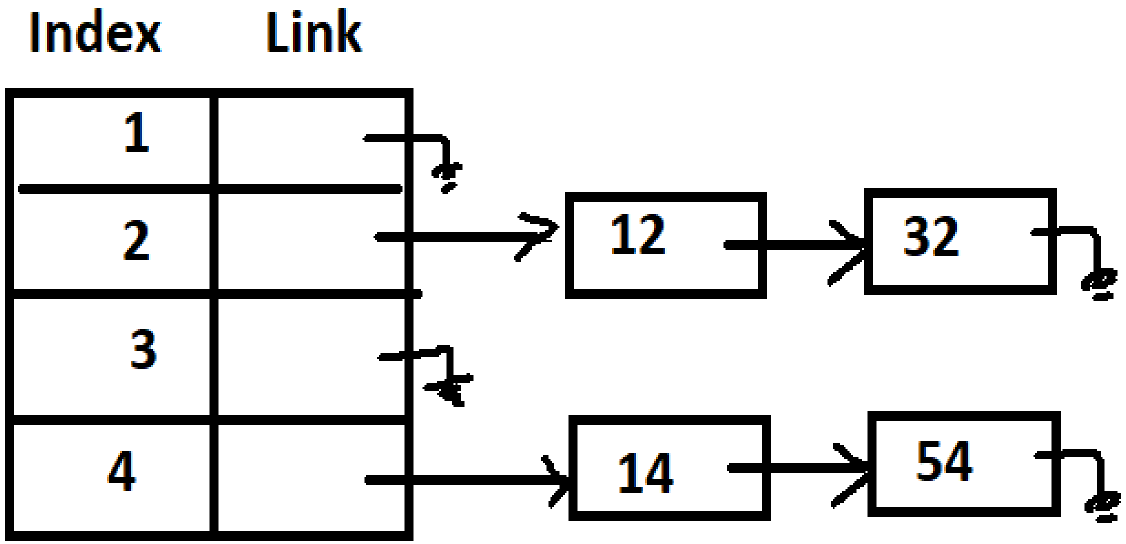
Collision Resolution Techniques

- Separate Chaining (Most popular and used)
- Linear Probing
- Quadratic Probing
- Double Hashing

Separate Chaining

- When collision
- Add node to the list of values for that index
- List can be in sorted order / not sorted

- So, In Finding element
  - First get index using hash
  - In that index do linear search
  - OR do binary search (if sorted list)
- Most popular and used approach
- **Dis-advantage:**
- Extra Linked Lists to be used



Linear Probing

- If collision then insert <key, value> at next free slot in circular manner (after end again search from beginning)

- For example , **we want to insert 51.**
- So index = hash(51) = 51 % 10 = 1
- But, already 21 is present
- So, search next free slot index 2 and put 51 in it
- **51 will be put at index 2**

- **We want to insert 105**
- So index = hash(105) = 105 % 10 = 5
- But, already 105 is present
- **So check 6, 7, 8, 9 and 0 insert 105 at index 0**

Index	Key
0	
1	21
2	
3	23
4	
5	45
6	56
7	77
8	78
9	99

Quadratic Probing

- If collision then insert <key, value>, find next free slot using **QUADRATIC approach** in circular manner
- **Formula :**
- next free slot = index + 1^1,
- OR index + 2^2,
- OR index + 3^2, etc

- Example
- Insert 105 -> index =5 , 5+1 , 5+2^2, 5+3^2
- So **Insert 105 at location 4** (5+3^2 = 14 % size = 4)

Index	Key
0	
1	21
2	
3	23
4	
5	45
6	56
7	77
8	78
9	99

Disadvantage of Linear & Quadratic Probing

- If many collision then searching time increases

- Elements may not be found at desired location

- All elements tend to cluster at one place in complete table

- Solution : Separate chaining

Double Hashing

- If collision use another hash function to calculate new location

- *hash1* != *hash2*

- Disadvantage :: Very Slow because of two times hashing