HASHING IN DATA STRUCTURE

By: Rashmi Prabha

Hashing → Indexing

Chapter 5 → key value → Searching fast

accessing | retrieval

- Hashing is a technique or process of mapping keys, values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

hashing

→ key value pair

→ hash table

→ hash function ✓ efficiency

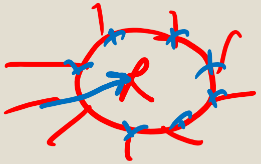
1 ✓ - Hash Table
2 ✓ - Methods of Hash table
  - ✓ Collision Handle
  - ✓ Chaining
  - ✓ Open Addressing
3 - Hash Function

Problem

collision ✓ → Bathroom

Resources

Synchronization



Timeslot

example

dbms

- Suppose we want to store Employee details:- Amazon

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

- Which data structure can be used for storing above values:-

1. Array of phone numbers and records. ✓
2. Linked List of phone numbers and records. ✓
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table. ✓

Areaswise
7666

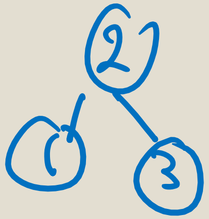
DS → Adv. | Disadv / App^n

linear DS

- For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

Non-linear

- With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.



Bank A/c No → key1

key2

SBI ICICI

SqL s

Phone no. (P.key)

↓

**

- *Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.*
- *Hash Function* - a hash function maps a big number or string to a small integer that can be used as index in hash table.
- A good hash function should have following properties
  1) Efficiently computable.
  2) Should uniformly distribute the keys (Each table position equally likely for each key)
- Hash Table - An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Array → index – indexing

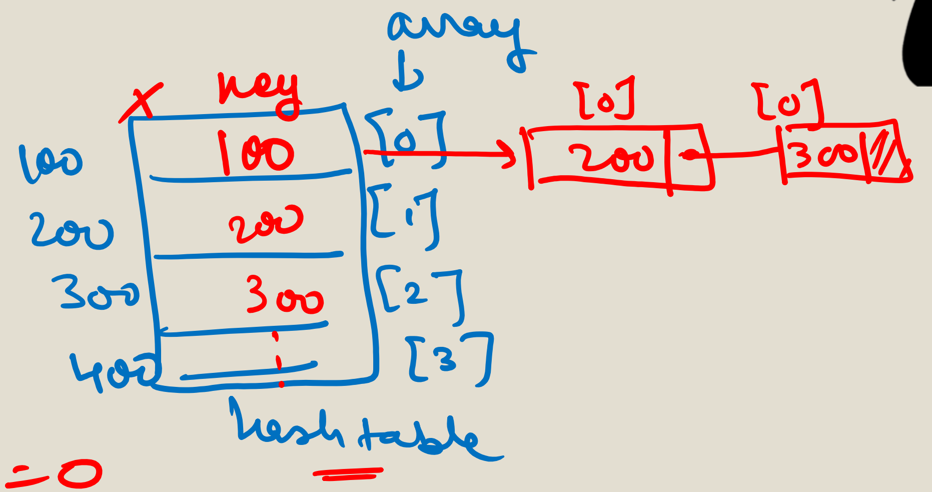
hash function: n%5 th index

n= 100 %5 = 0 ✓

n= 200 %5 = 0 ✓

collision

n=300%5=0



**

- What is collision? Problem
- two keys result in the same value
- How to handle collision?
- **Chaining: The** idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

hash function

next memory allocate

Example - Separate chaining

key

Q. Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Collision will occur or not ?

If yes → location ?

$\boxed{1, 3}$

Total Collision ⇒ 5 - 1 → 3

3 → 2

50 mod 7 = 1

700 mod 7 = 0

76 mod 7 = 6

85 mod 7 = 1

92 mod 7 = 1

73 mod 7 = 3

101 mod 7 = 3

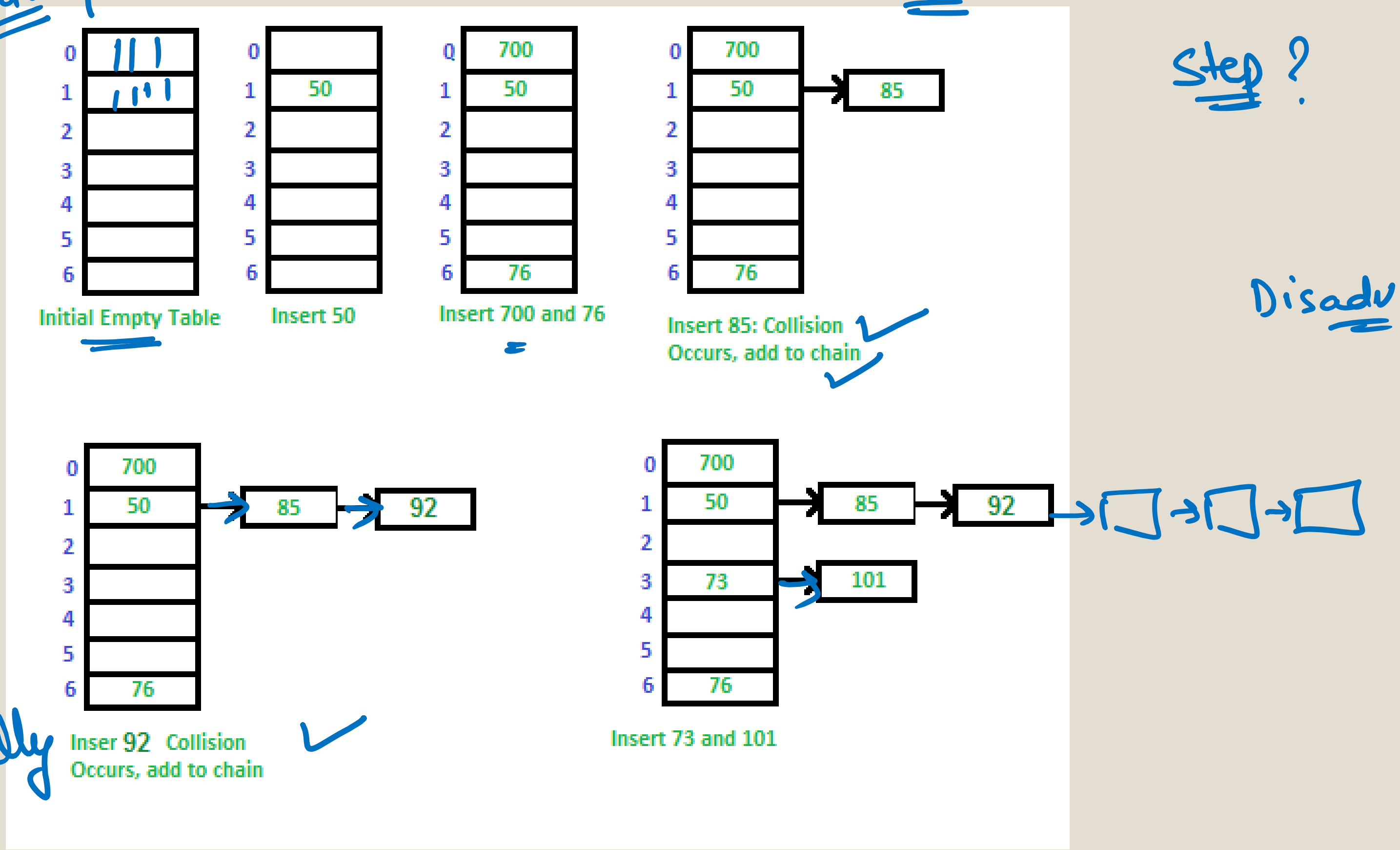
7)~~85~~(1

memory repn

array

numerical

Step ?



Disadv : extra space

① separate chaining
↓
linked list

dynamically create.

- **Advantages:** of separate chaining

1) Simple to implement.
2) Hash table never fills up, we can always add more elements to the chain.
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted

linked list → dynamic → size compile time ✗

runtime → memory allocate ✓

cache →

- **Disadvantages:** of separate chaining ②

  1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.

  2) Wastage of Space (Some Parts of hash table are never used)

  3) If the chain becomes long, then search time can become O(n) in the worst case.

  4) Uses extra space for links.

separate chaining or

- Time complexity of search insert and delete in Hash hunction (Linear probing) is O(1) if a is O(1) ✓

linked list → O(1)

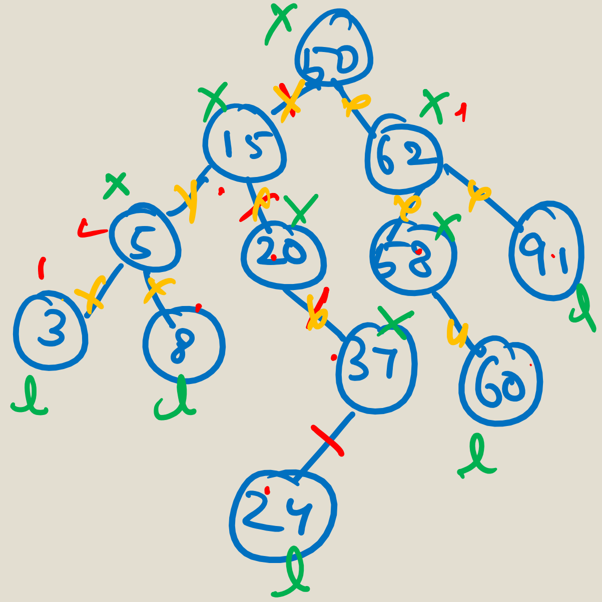
Q In BST insert inorder. Find no. of nodes in left and right subtree of root respectively?

→ 50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

7, 4 ✓

4, 7

Traverse



Root

left < right >

height of tree → max edge|path = 4

leaf nodes → 5

internal nodes → 7

total nodes → 5+7 = 12

total edges → 11

Q1 left, right

Gate

Q2 BST Postorder: ~~10~~, ~~9~~, ~~23~~, ~~22~~, ~~27~~, ~~25~~, ~~15~~, ~~50~~, ~~95~~, ~~60~~, ~~40~~, ~~29~~

L R Root

Key values

a) tree can be formed uniquely using only Postorder or Inorder or Preorder?

→ Postorder or Inorder both

→ Preorder or Inorder both

b) Inorder = ? L Root R increasing → 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95

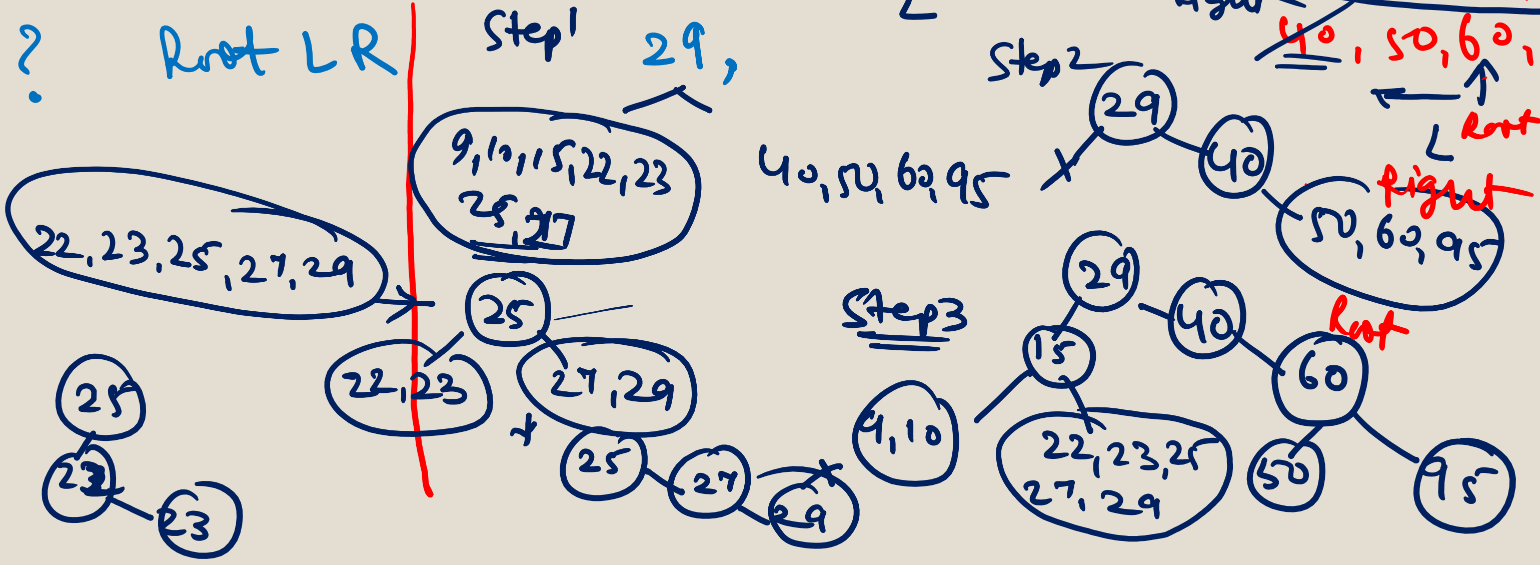
Root Right Right BST Right L Root R

Preorder = ? Root L R

Tree — yes

Pre-order

Test 5 M1 Program



split sorting merge

Root · key value · Root · Root · Root · Root · Root · Root · Root

Postorder → 10, 9, ~~23~~, ~~22~~, ~~27~~, ~~25~~, ~~15~~, ~~50~~, ~~95~~, ~~60~~, ~~40~~, ~~29~~

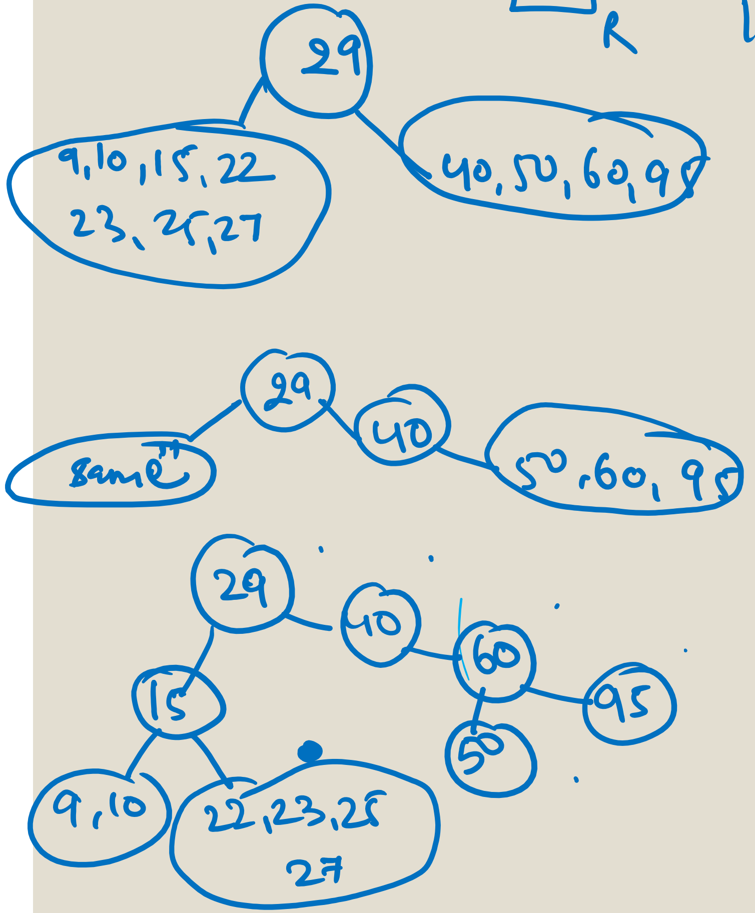
search · L · Right

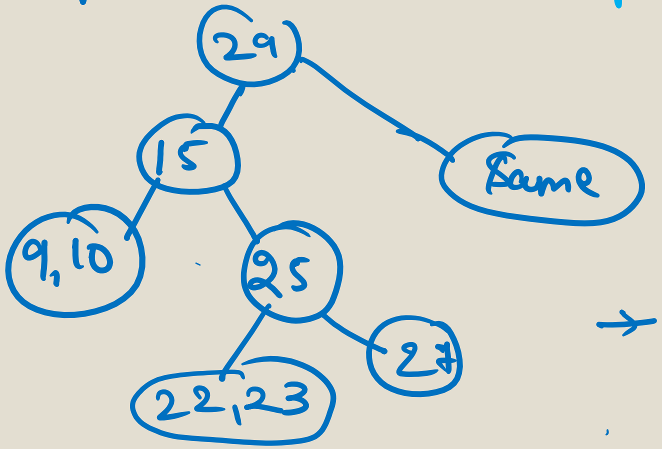
Inorder → ~~9~~, 10, ~~15~~, ~~22~~, ~~23~~, ~~25~~, ~~27~~, ~~29~~, ~~40~~, ~~50~~, ~~60~~, ~~95~~

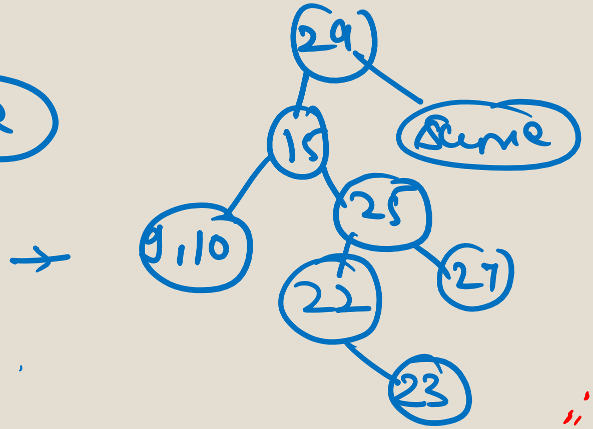
R · left · L · Root · R · Right · left · Right · Right

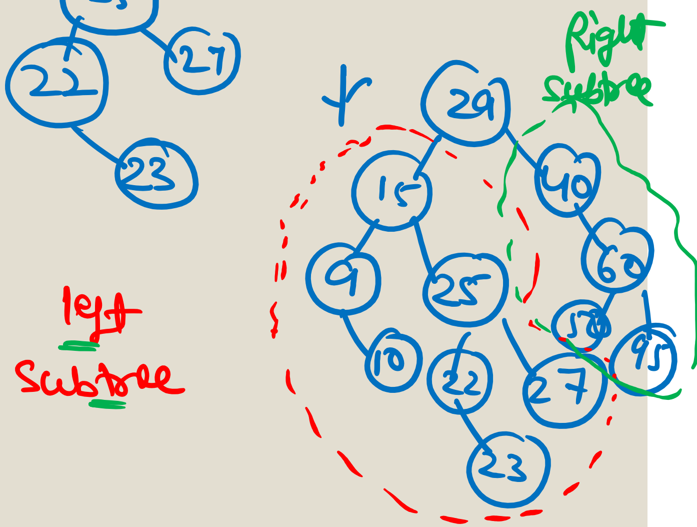
array

Postorder
↓
element(Root)
↓
match
Inorder









22 Sep 2021

Q1 Previous slide final tree → Preorder traversal → Root L R

Q2 Preorder = ?

Inorder = ? Increasing order

Q3 On the basic of Q2 → Form BST

Q4 previous slide Tree = new Tree

Step 1: Root node identify
on the basis of
pre or post order

Step2: Root inorder
Match, left ←
Right →

Step3: Immediate value inorder
& graph plot

traverse LR Root

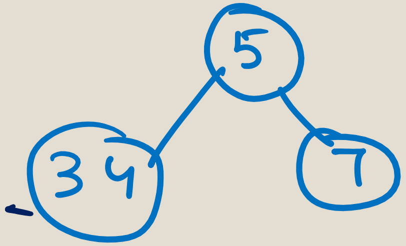
preorder / Postorder 3 ~~4~~ ~~7~~ ~~5~~ Root

Inorder :- 3 4 5 7
Left Root Rt
LRoot Right

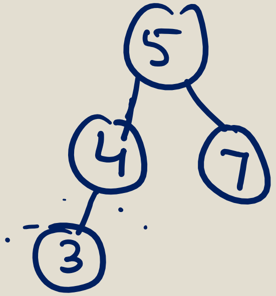
Dig

Step 1 →

Step2 →

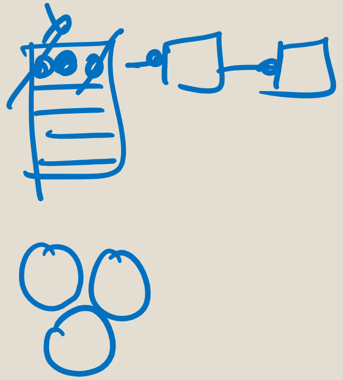


Step3 →



BST

Collision hash values



- **Challenges in Linear Probing :**

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.

2. **Secondary Clustering*:*** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

Data Structures For Storing Chains: **

Tree, Graph

- Linked lists Program
  - Search: O(l) where l = length of linked list
  - Delete: O(l)
  - Insert: O(l)
  - Not cache friendly

DSA

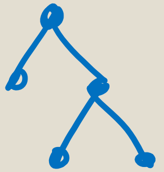
Constant O(1)

Dynamic / runtime

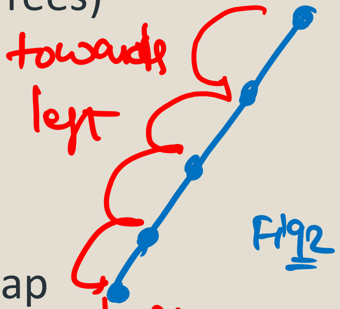
- Self Balancing BST (AVL Trees, Red Black Trees) ① ②
  - Search: O(log(l)) Properties
  - Delete: O(log(l))
  - Insert: O(l)
  - Not cache friendly
  - Java 8 onwards use this for HashMap

1.8

Fig1



Skewed towards left



loop.

Fig2

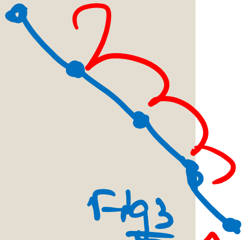


Fig3

Skewed towards Right

Technical

Java Ad Java ** Collection

- Dynamic Sized Arrays (Vectors in C++, Array List in Java, list in Python)
  - Search: O(l) where l = length of array
  - Delete: O(l)
  - Insert: O(l)
  - Cache friendly

2,3

~~[illegible]~~ Interview (Concept)

depth : Arraylist , Multithreading

Syntax OOPS

Program, Abstract, Interface

static, this

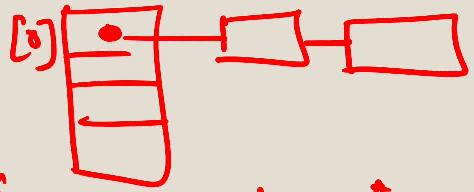
super

exception handling

② hashing

Open Addressing

Chaining



same index → list allocate

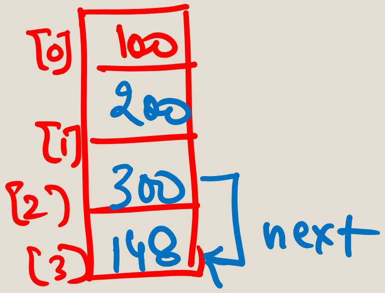
Adv. Di's

- In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

linear Probing → next vacant space allocate key value

- Quadratic Hashing
- Double Hashing - **(firstHash(key) + i * secondHash(key)) % sizeOfTable**



[0] - 200

148 % 7 = [2]

** diff' S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement. list	Open Addressing requires more computation. complex
2.	dynamic In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full. opposite
3.	Chaining is Less sensitive to the hash function or load factors. size	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
buffer area local memory 5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Double Hashing

Concept

- **Double hashing** is a collision resolving technique in **Open Addressed** Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

- *Double hashing can be done using :*
  (hash1(key) + i * hash2(key)) % TABLE_SIZE
  Here hash1() and hash2() are hash functions and TABLE_SIZE
  is size of hash table.

Thank you

Revise each concept properly

All the best

By: Rashmi Prabha