DATA STRUCTURES & ALGORITHMS

UNIT NO. 6

HASHING

Topic Name

Hashing, Hash Table

By,
Prof. Dipika Birari

Content

Topics to cover:

- **Hashing Introduction**
- **Hash Table**
- **Applications of Hash Table**
- **Bucket**
- **Bucket Hashing**

Hashing: Introduction

Hashing:

Hashing is a search strategy based on the value of the key on which the search is based.

- In all searching techniques like
  - Linear Search
  - Binary Search
  - Search Trees

The number of elements get increased, the time required to search an element also increased linearly.

Hashing: Introduction

Hashing:

- Hashing is another approach in which time required to search an element doesn't depend on the number of elements.

> **“Hashing is the process of indexing and retrieving element in data structure to provide faster way of finding the element using hash key.”**

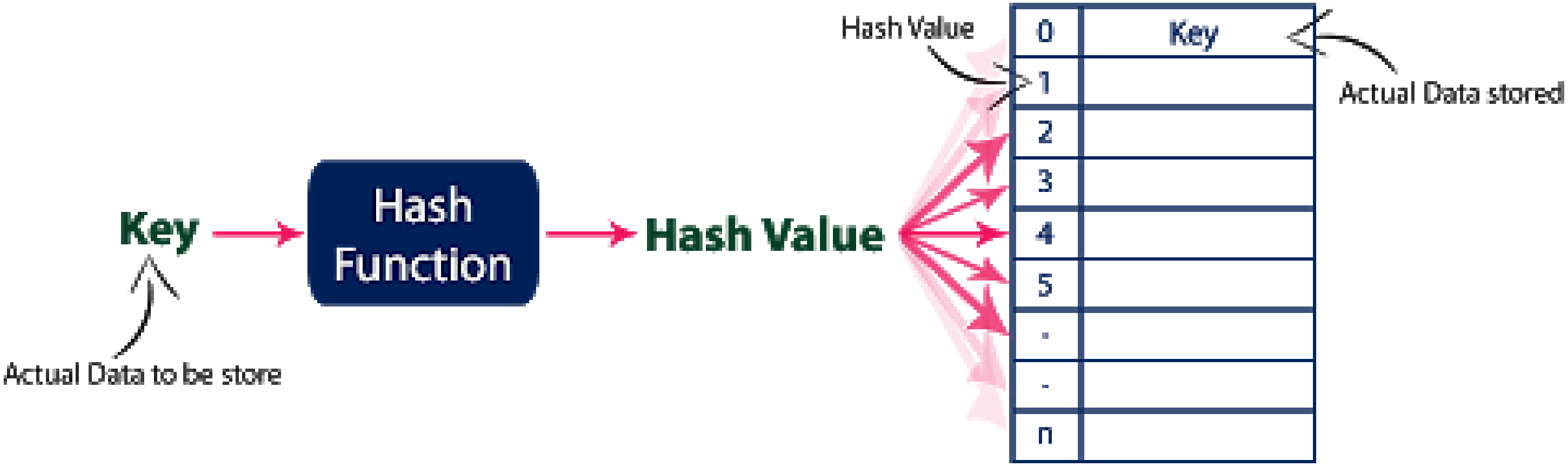
- Here,

Hash key: Value gives the index value where the actual data is likely to store in the data structure.

Hash Table

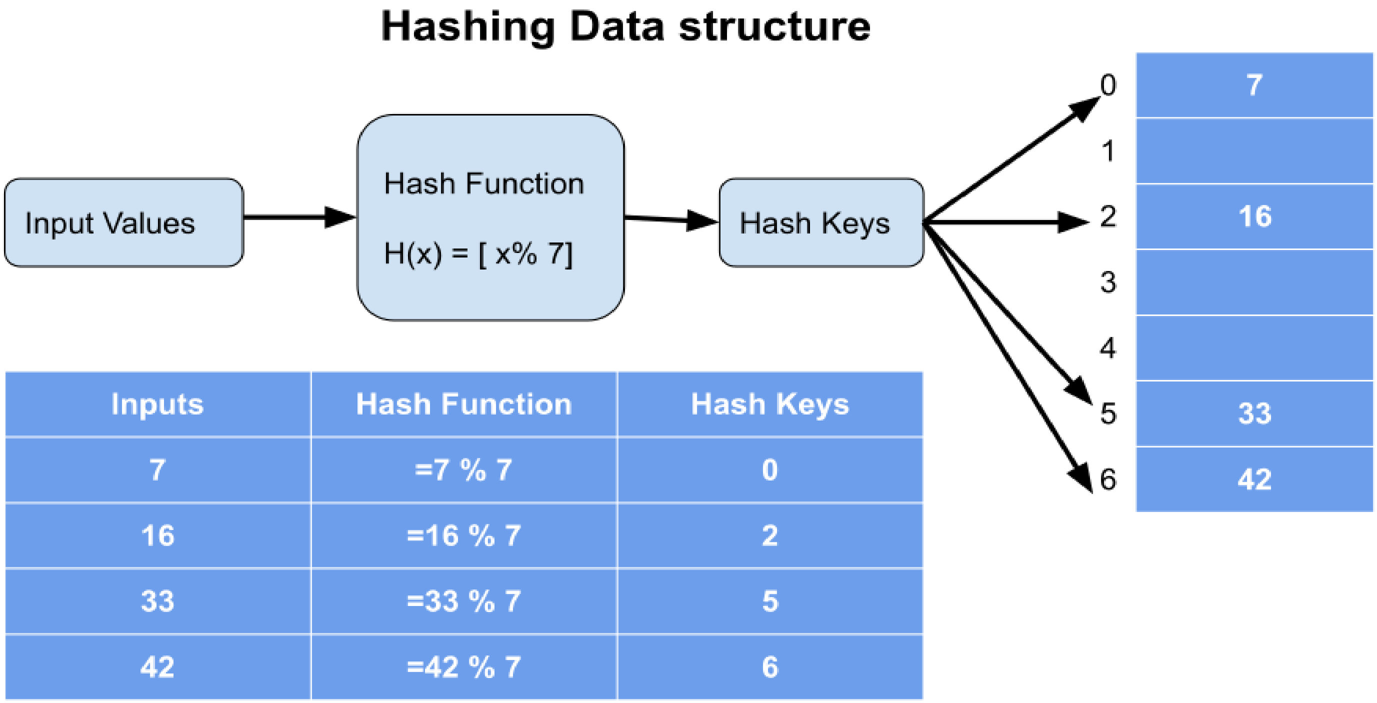
Hash Table:

- Hash table used to store data. All the data values are inserted into the hash table based on the hash key value.
- Hash key value is used to map the data with index in the hash table.
- And the hash key is generated for every data using a hash function.



Hash Table

> " Hash table is an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1))."



Hashing Data structure

Inputs	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

Hash Table

Applications of Hash Table

Applications of Hash Table:

- Database Systems
- Symbol Tables
- Data Dictionaries
- Network Processing Algorithm

Bucket

Consider the example of home address,
ABC live in XYZ Building

- So, a building or apartment = bucket,
  housing colony = hash table and
  house = entry.
- Technically, hash table stores entries(key, value) pairs in buckets.

Bucket = Block of records corresponding to one address in hash table.

- The hash function gives the Bucket address.

Bucket Hashing

- Hash Table slots into buckets. The hash function assigns each record to the first slot within one of the buckets.

- The M slots of the hash table are divided into B buckets, with each bucket consisting of M/B slots.

- If slot is already occupied
  - Then the bucket slot searched sequentially until an open slot is found.

- If bucket is entirely full
  - Then the record is stored in an "***Overflow Bucket"*** of infinite capacity at the end of the table.

- All buckets shares the same overflow bucket.

Bucket Hashing

Searching for a record:

- Hash the key to determine which bucket should contain the record. Then search records in bucket.

- If desired key value is not found & the bucket is still has free slot
  - Then the search is complete.

- If bucket is full, then it is possible that the desired record is stored in the " Overflow Bucket".
  - This bucket must be searched until the record is found or all records in the overflow bucket have been checked.

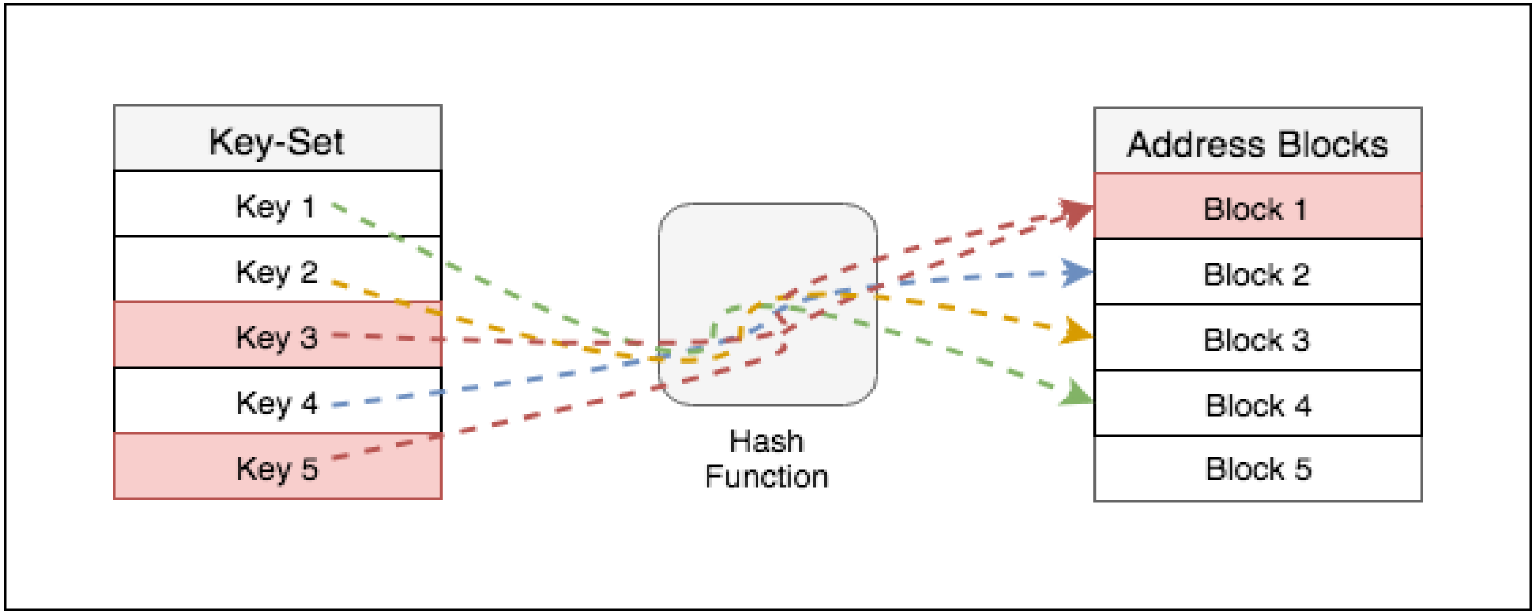
✓ Example

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BucketHash.html

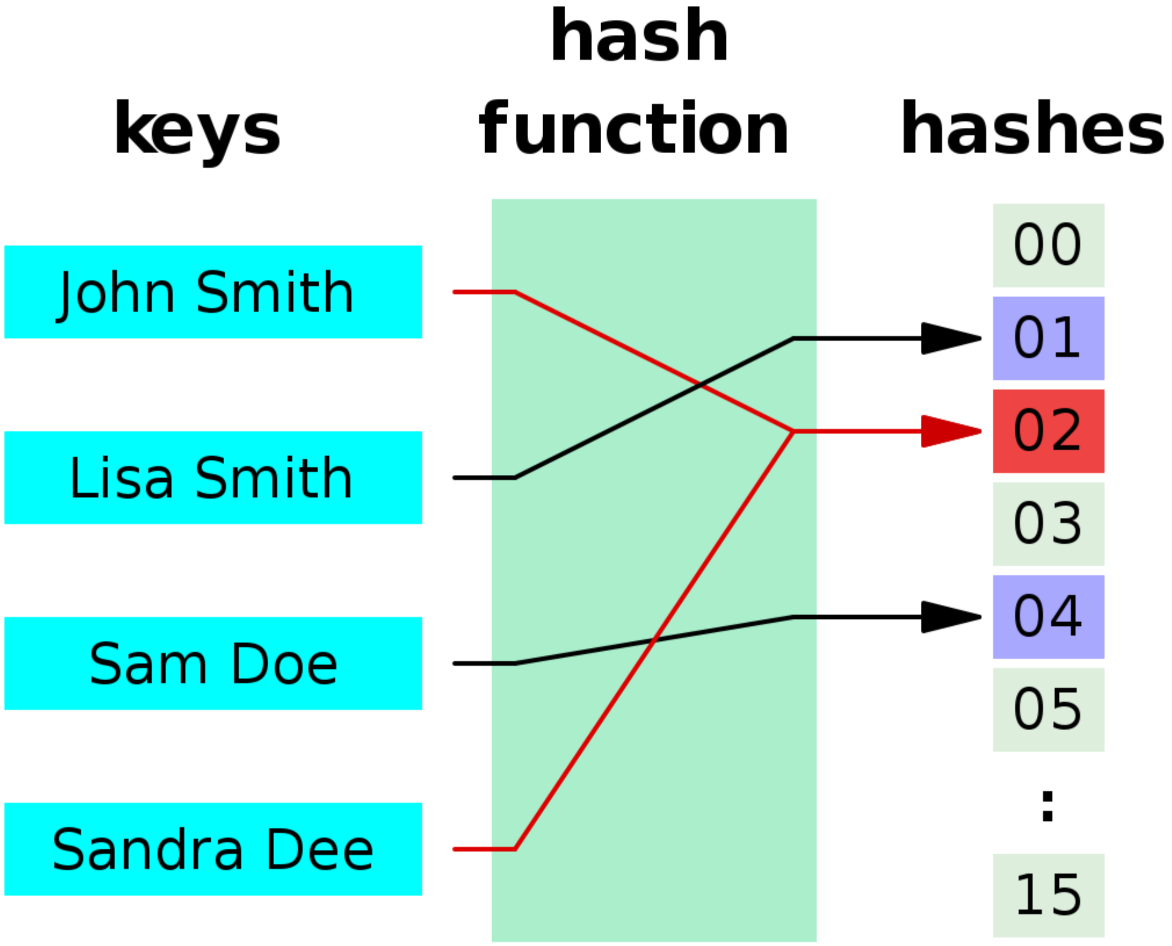
Collision

- A situation when the resultant hashes for two or more data elements in the data set U, maps to the same location in the hash table, is called a "***hash collision***".

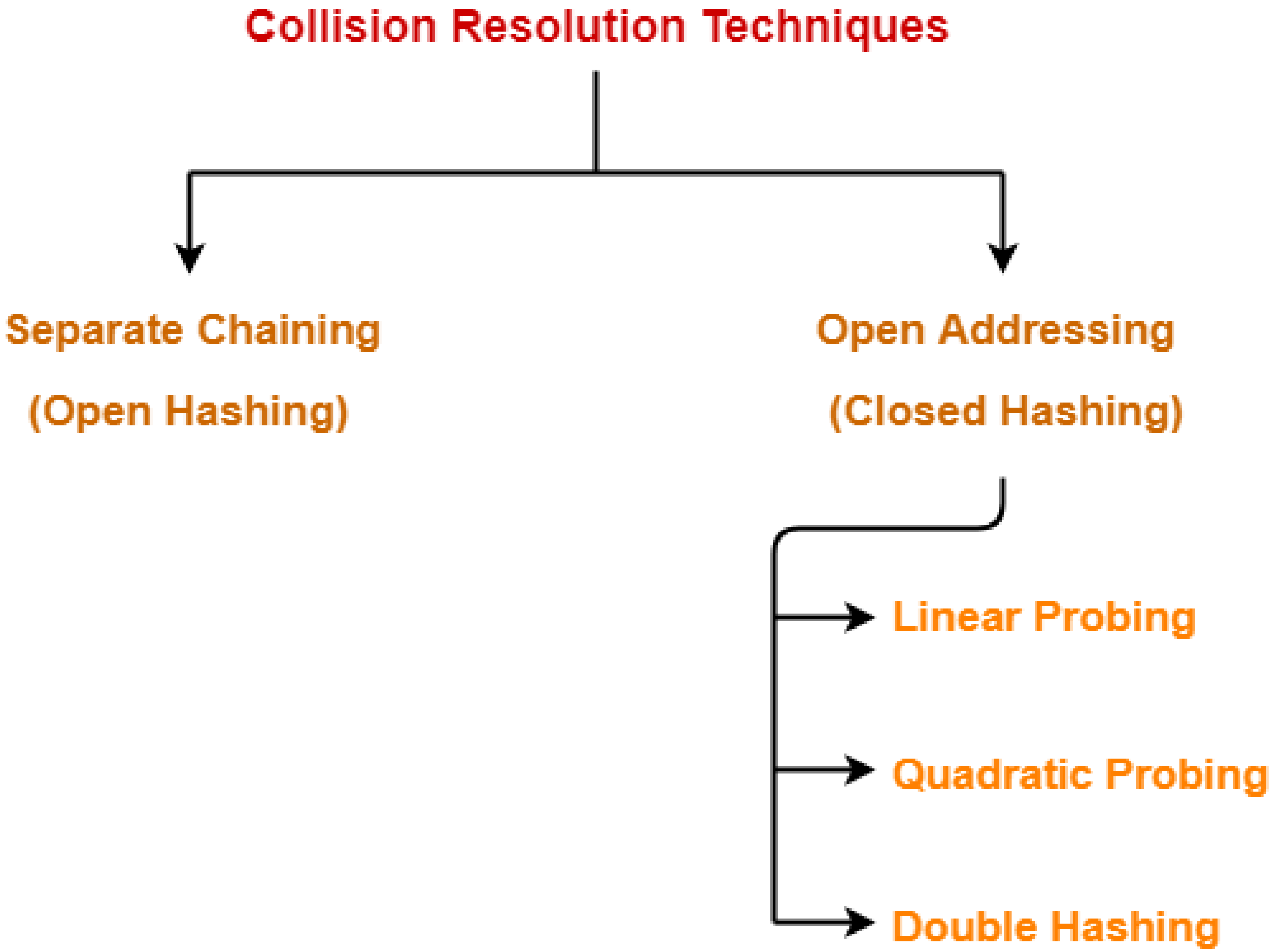
- In such situation two or more data elements would qualify to be stored or mapped to the same location in the hash table.



Collision


keys
hash function
hashes
John Smith
Lisa Smith
Sam Doe
Sandra Dee
00
01
02
03
04
05
:
15

Collision


Collision Resolution Techniques

Separate Chaining

(Open Hashing)

Open Addressing

(Closed Hashing)

Linear Probing

Quadratic Probing

Double Hashing

Overflow

Overflow:

An overflow occurs when the home bucket for a new pair (key, element) is full.

We may handle overflows by :

- □ Search the hash table in some systematic fashion for a bucket that is not full.
- □ Linear probing
- □ Quadratic probing.
- □ Random probing.

Perfect Hash Function

Hash Function:

A **hash function is any function that can be used to** map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.

The values returned by a hash function are called

- hash values,
- hash codes,
- hash sums, or
- simply hashes.

Perfect Hash Function

<u>Perfect Hash Function:</u>

Perfect hash function for a set S is a hash function that maps distinct elements in S to a set of integers, with no collisions.

Perfect hash functions may be used

- □ To implement a lookup table with constant worst-case access time.

- □ It has many of the same applications as other hash functions, but with the advantage that no collision resolution has to be implemented.

Load Density

The loading density or loading factor of a hash table is

$$\boxed{a = n/(sb)}$$

Where,

- s is the number of slots
- b is the number of buckets

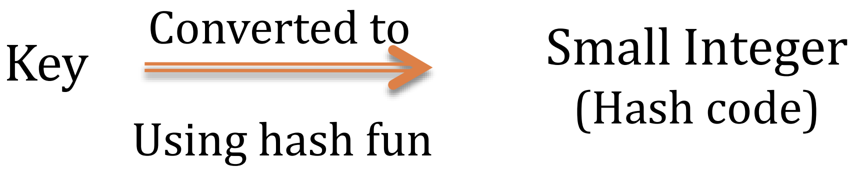
FULL TABLE

A table in which all the locations are filled with values and no any location is remained empty is known as Full Table.

Rehashing

Load Factor and Rehashing.

- For insertion of a key(K) – value(V) pair into a hash map, following steps are required:



Key

Converted to

Using hash fun

Small Integer

(Hash code)

- The hash code is used to find an index,

Index = hash code % array size

- The entire linked list at that index is first searched for the presence of the K already.
  - If found, it's value is updated and
  - Otherwise, the K-V pair is stored as a new node in the list.

Rehashing

Complexity and Load Factor:

- For the first step, time taken depends on the K and the hash function.
  - example, if the key is a string “abcdef”, then it’s hash function may depend on the length of the string.

- But for very large values of n,
  - the number of entries into the map,
  - length of the keys is almost negligible in comparison to n

So hash computation can be considered to take place in constant time,

i.e., O(1)

Rehashing

Complexity and Load Factor:

- For the second step, traversal of the list of K-V pairs present at that index needs to be done.

- The worst case = all the n entries are at the same index.

Time complexity = O(n).

- But hash functions uniformly distribute the keys in the array so this almost never happens.

Rehashing

Complexity and Load Factor:

- So, **on an average,**
  - if n = entries and
  - b = size of the array
  - n/b = entries on each index.

  This value n/b is called the **load factor**

- This Load Factor needs to be kept low, so that number of entries at one index is less and so is the complexity almost constant,

i.e., O(1)

Rehashing

"Rehashing means hashing again."

Basically,

□ when the load factor $\xrightarrow{\text{increases}}$ > its pre-defined value of load factor (default value =0.75),

Then complexity $\xrightarrow{\text{increases}}$

To overcome this, the size of the array is increased (doubled)

⇓

All the values are hashed again and

⇓

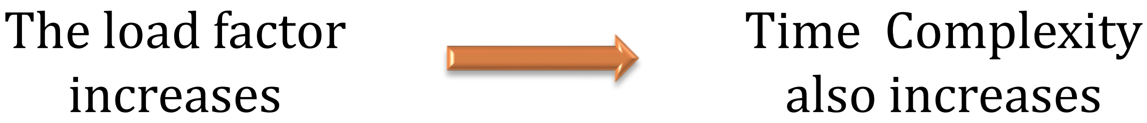
Stored in the new double sized array

⇓

To maintain a low load factor and low complexity.

Rehashing

whenever key value pairs are inserted into the map,



The load factor increases

Time Complexity also increases

This might not give the required time complexity of O(1).

Therefore, Rehashing is done

Rehashing

Rehashing can be done as follows:

- For each addition of a new entry to the map,
  - Check the load factor.
- If it’s greater than its pre-defined value (or default value = 0.75)
  - Then Rehash
- For Rehash,
  - make a new array of double the previous size and
  - make it the new bucket array.
- Then traverse to each element in the old bucket Array and call the insert() for each
  - So as to insert it into the new larger bucket array.

Issues in Hashing

- Hash tables are extremely useful data structures as lookups take expected O(1) time on average,
    - i.e. the amount of work that a hash table does to perform a lookup is at most some constant.

- Several data structure and algorithms problems can be very efficiently solved using hashing which otherwise have high time complexity.

Issues in Hashing

Here, few problems that can be solved in elegant fashion using hashing:

- □ Find pair with given sum in the array
- □ Shuffle a given array of elements
- □ Find majority element in an array
- □ Check if repeated subsequence is present in the string or not.
- □ Print all nodes of a given binary tree in specific order.
- □ Print left view of binary tree.
- □ Print Bottom View of Binary Tree.
- □ Print Top View of Binary Tree.
- □ Find Duplicate rows in a binary matrix.
- □ Remove duplicates from a linked list.

Collision Resolution Strategies

Collision

When two different keys produce the same address, there is a ***collision***.

Hash Collision

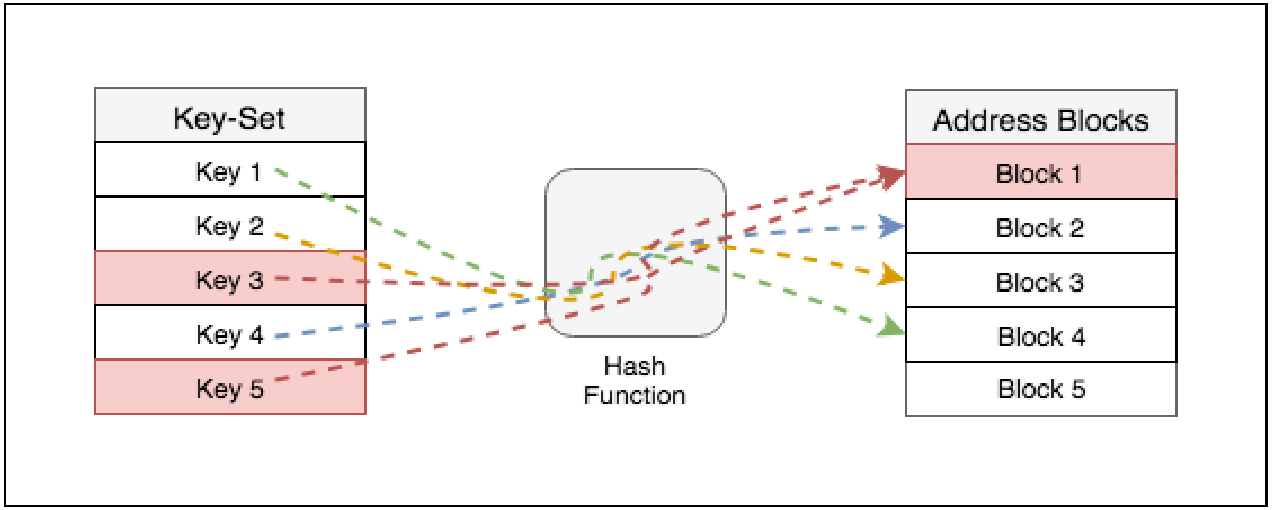
A situation when the resultant hashes for two or more data elements in the data set U, maps to the same location in the hash table, is called a ***hash collision***.

Collision Resolution Strategies

Hash Collision

- In such situation two or more data elements would qualify to be stored or mapped to the same location in the hash table.

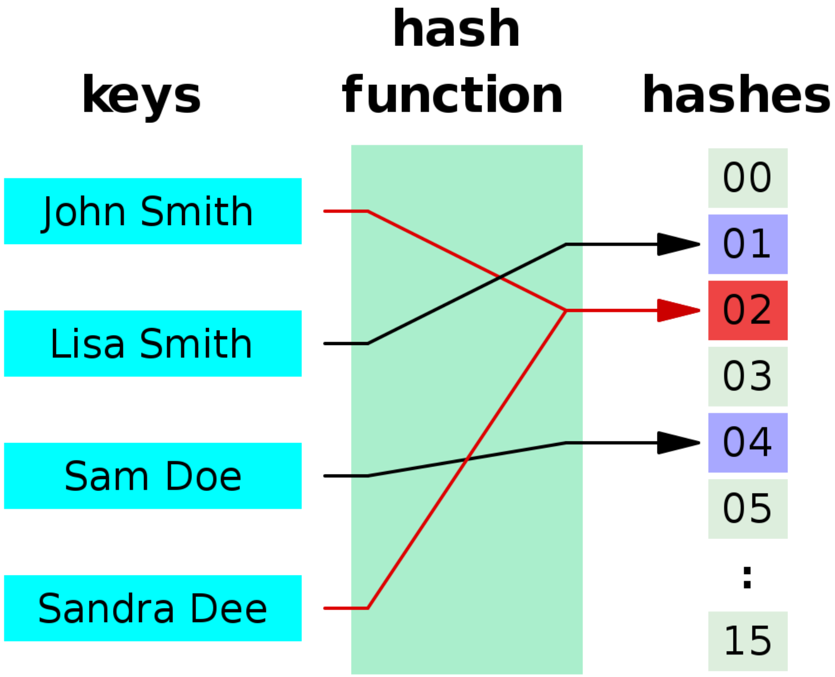
- The keys involved are called ***Synonyms.***

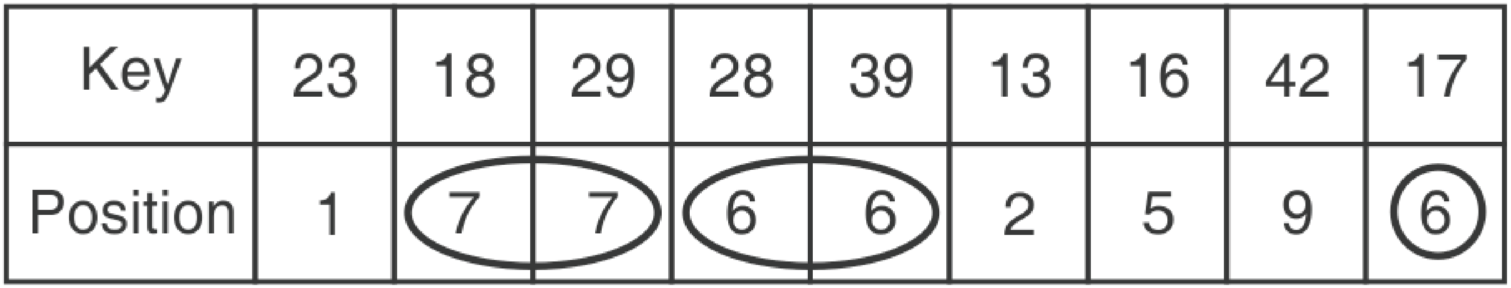


Collision Resolution Strategies

Hash Collision : Example

- Collision occurs when h(k1)= h(k2)
- Hash table size: 11
- Hash function: key mod hash size
- So, the new positions in the hash table are:





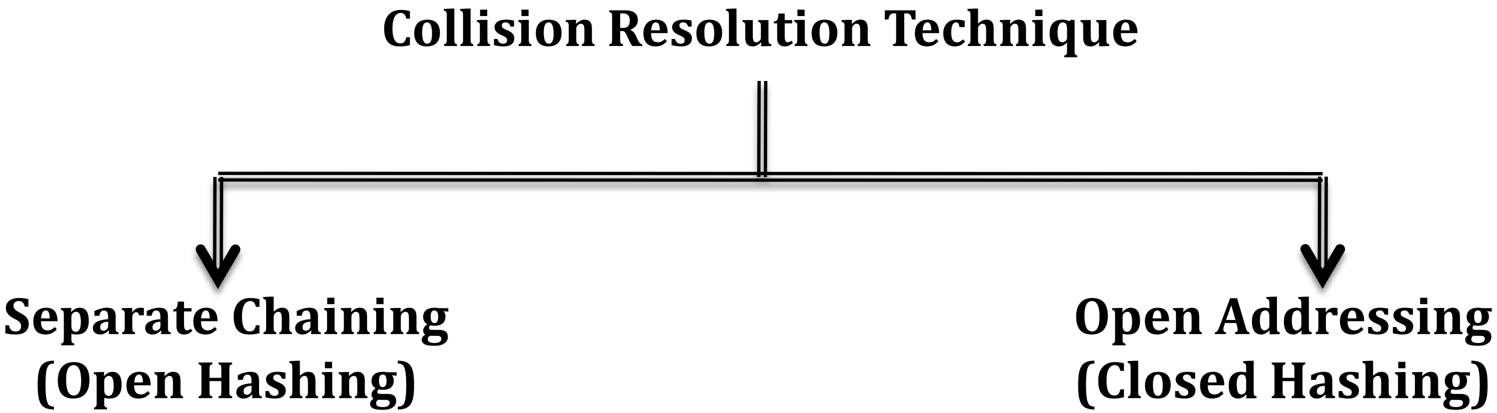
Key	23	18	29	28	39	13	16	42	17
Position	1	7	7	6	6	2	5	9	6

Collision Resolution Strategies

- A hashing function that avoids collision is extremely difficult. It is best to simply find ways to deal with them.

- The possible solution, can be :
  - Spread out the records
  - Use extra memory
  - Put more than one record at a single address.

Collision Resolution Strategies


Collision Resolution Technique

Separate Chaining
(Open Hashing)

Open Addressing
(Closed Hashing)

- **Chaining:** Stores colliding keys in Linked List at the same table index.
- **Open Addressing:** Store Colliding keys elsewhere in the table.

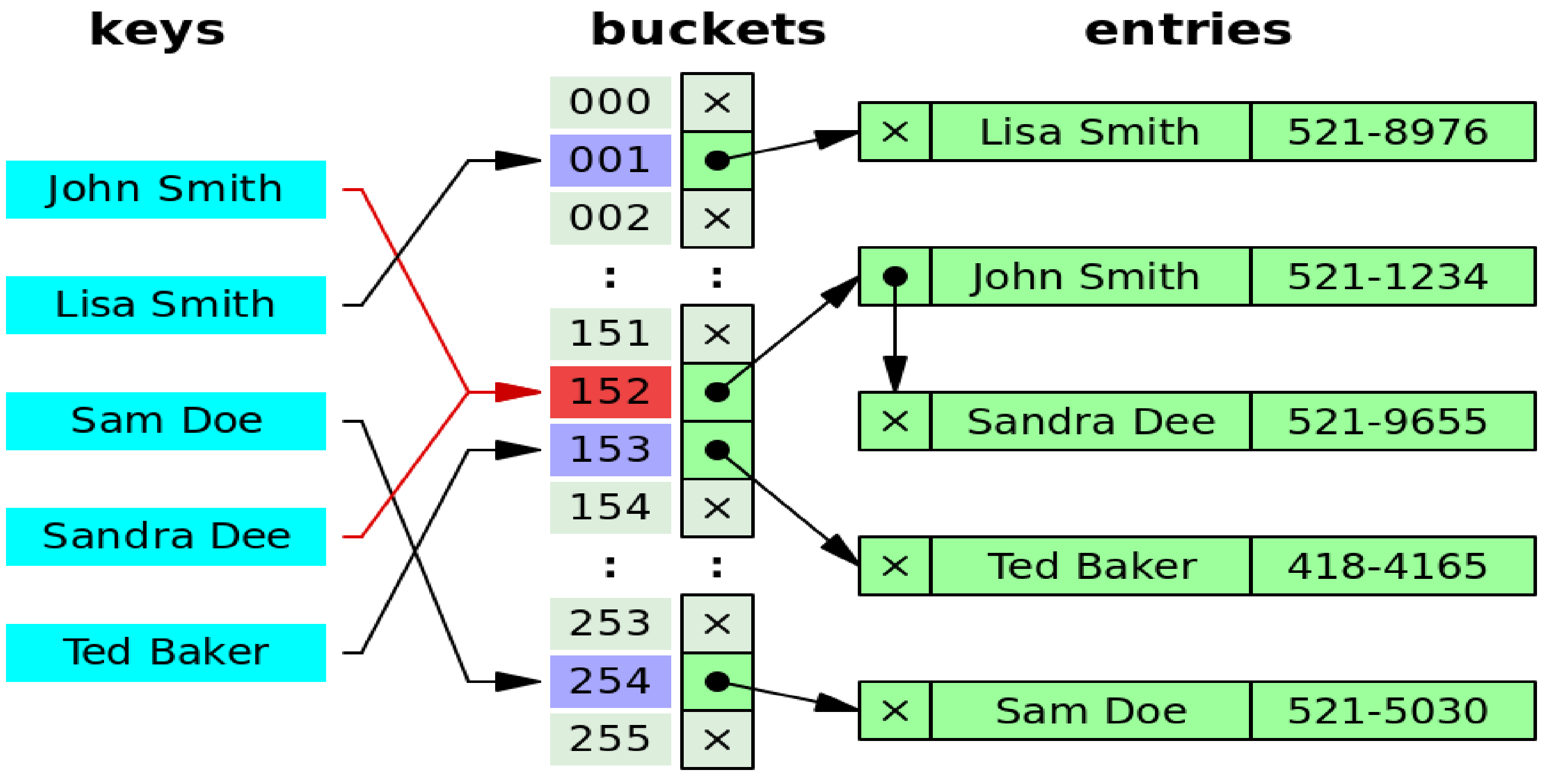
Open Hashing/Separate Chaining

Maintains a linked list at every hash index for collided elements.

- □ It is technique in which the data is not directly stored at the hash key index(k) of hash table.
- □ Rather the data at the key index (k) in the hash table is pointer to the head of the data structure where the data is actually stored.
- □ In the most simple & common implementations the data structure adopted for storing the element is a Linked List.

Open Hashing/Separate Chaining

In this, when data needs to be searched, it might becomes necessary to traverse all the nodes in the linked list to retrieve the data.



Open Hashing/Separate Chaining

Example:

Consider a simple hash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	
1	
2	
3	
4	
5	
6	

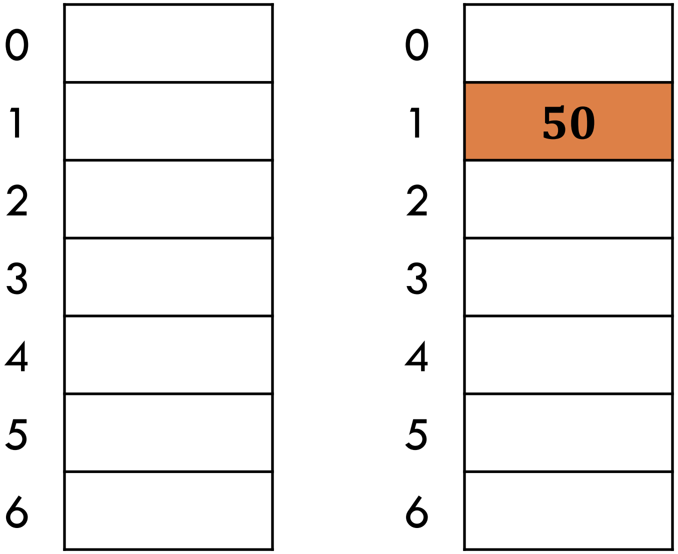
To insert 50,

Index = Key % 7
= 50 % 7
Index = 1

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	
1	**50**
2	
3	
4	
5	
6	

Insert 50

To insert 50,

Index = Key % 7
= 50 % 7
Index = 1

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a " Key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	
1	**50**
2	
3	
4	
5	
6	

To insert 700,

Index = Key % 7
= 700 % 7
Index = 0

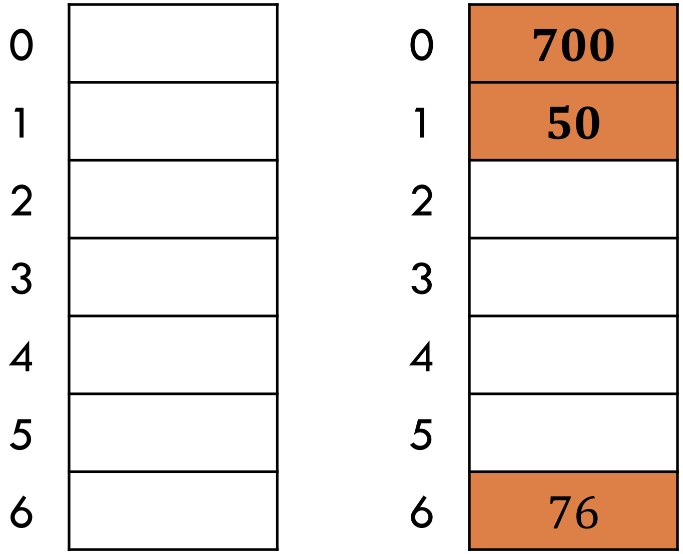
To insert 76,

Index = Key % 7
= 76 % 7
Index = 6

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a " Key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	**700**
1	**50**
2	
3	
4	
5	
6	76

Insert 700 & 76

To insert 700,

Index = Key % 7
= 700 % 7
Index = 0

To insert 76,

Index = Key % 7
= 76 % 7
Index = 6

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a " Key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	**700**
1	**50**
2	
3	
4	
5	
6	76

To insert 85,

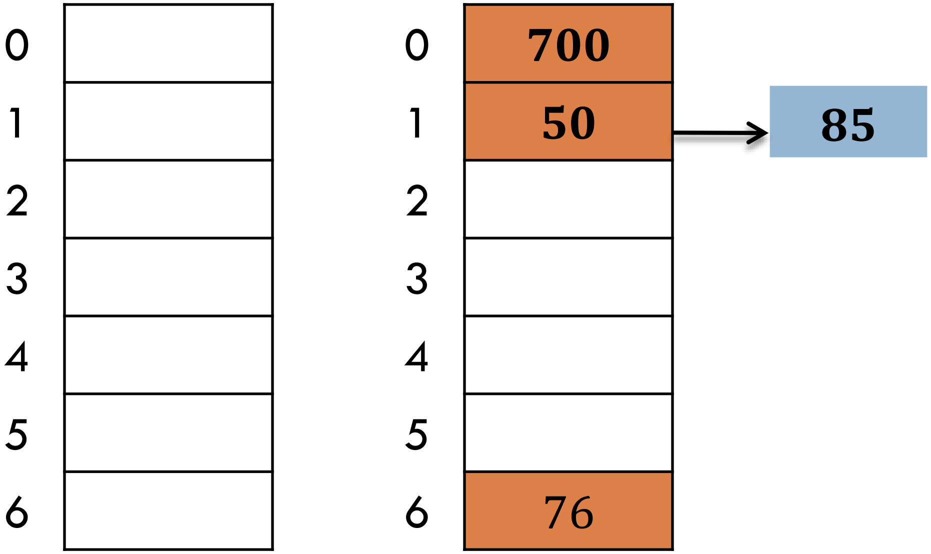
Index = Key % 7
= 85 % 7
Index = 1

Collision Occurs
Add to chain

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Initial Empty Table

To insert 85,

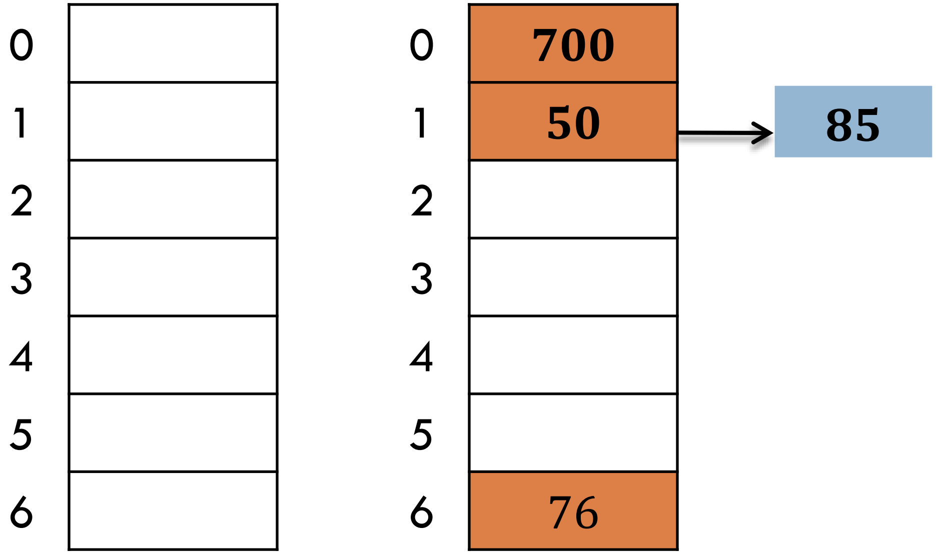
Index = Key % 7
= 85 % 7
Index = 1

Collision Occurs
Add to chain

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	**700**
1	**50** → **85**
2	
3	
4	
5	
6	76

Insert 85

To insert 92,

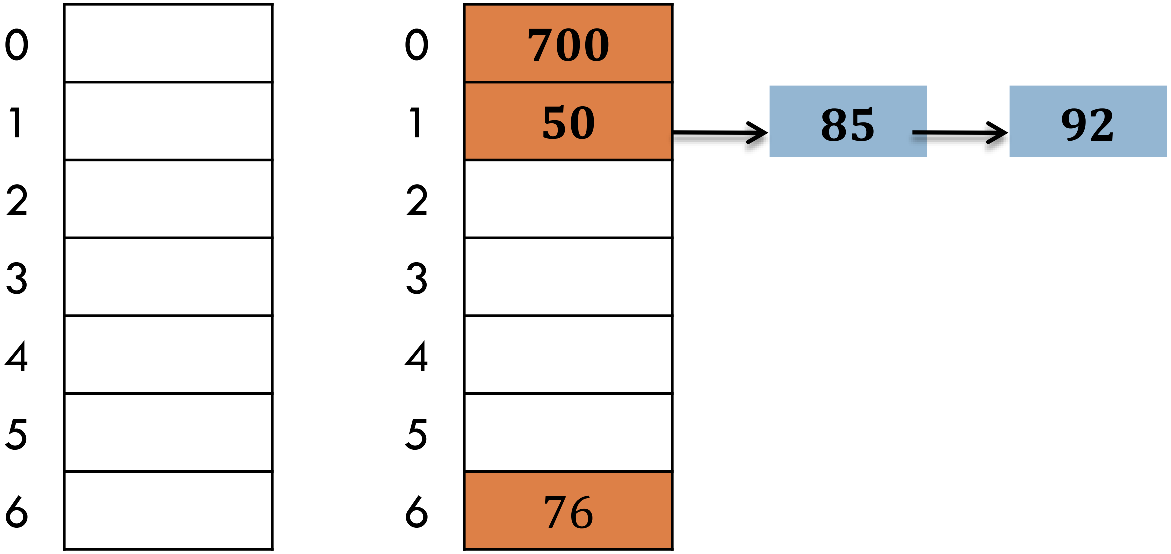
Index = Key % 7
= 92 % 7
Index = 1

Collision Occurs
Add to chain

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Initial Empty Table

Insert 92

To insert 92,

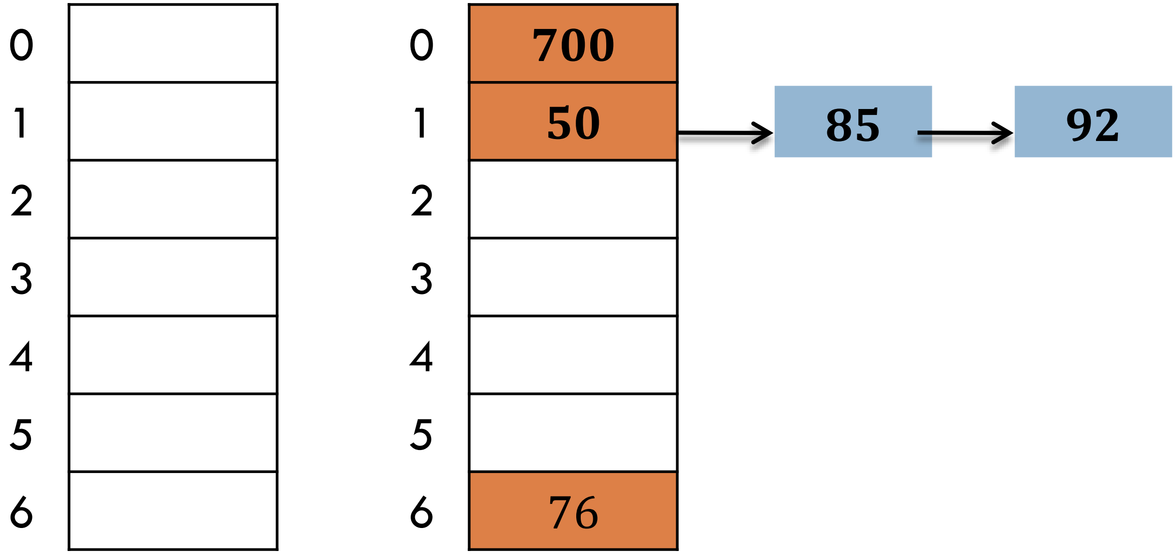
Index = Key % 7
= 92 % 7
Index = 1

Collision Occurs
Add to chain

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



To insert 73,

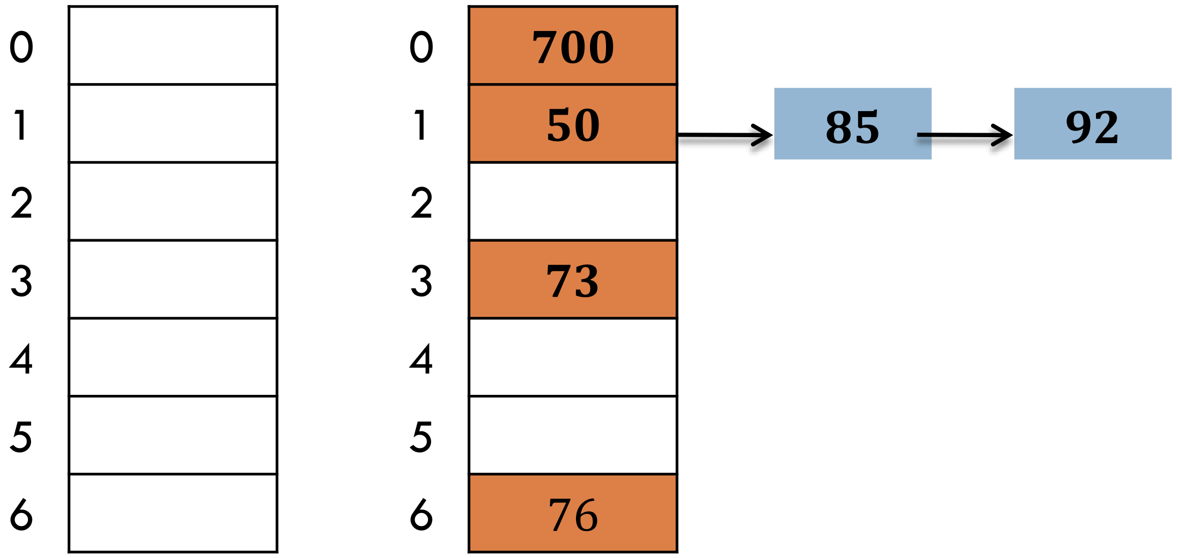
Index = Key % 7
= 73 % 7
Index = 3

Initial Empty Table

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Initial Empty Table

Insert 73

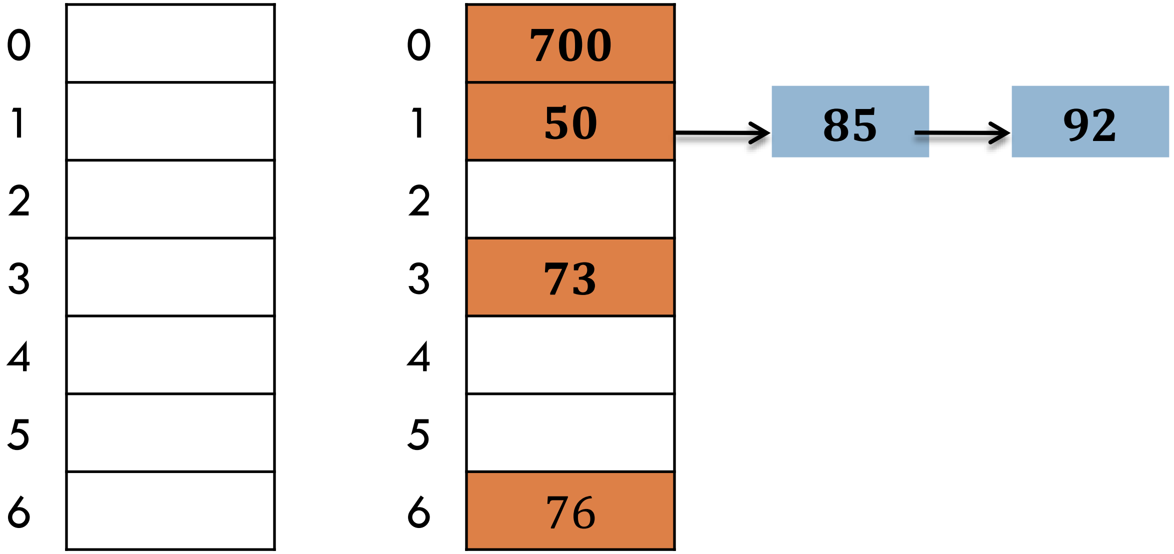
To insert 73,

Index = Key % 7
= 73 % 7
Index = 3

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a “ Key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Initial Empty Table

To insert 101,

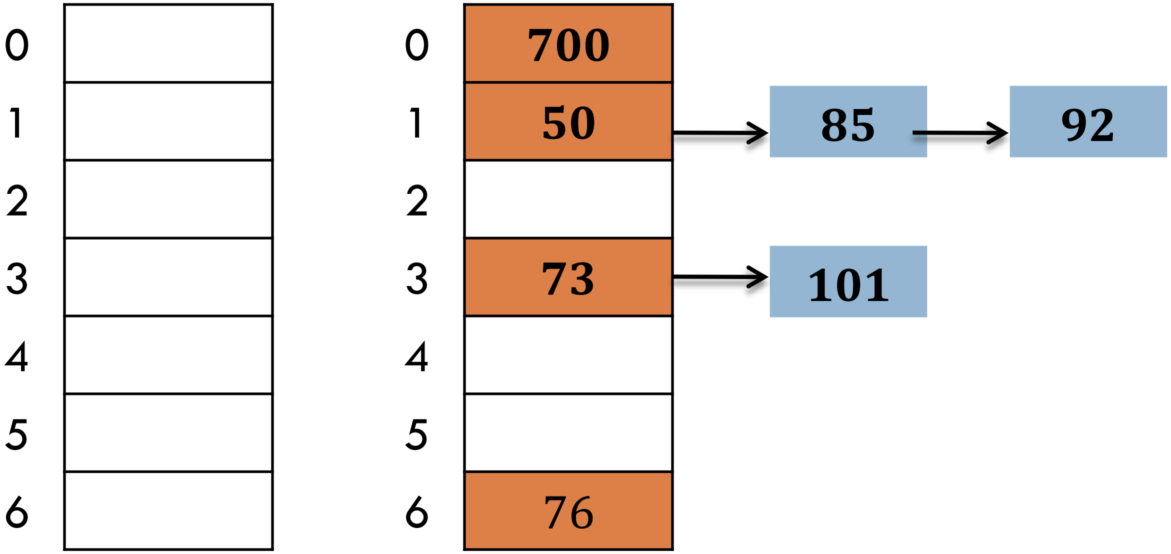
Index = Key % 7
= 101 % 7
Index = 3

Collision Occurs
Add to chain

Open Hashing/Separate Chaining

Example:

Consider a simple ash function a " Key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Initial Empty Table

Insert 101

To insert 101,

Index = Key % 7
= 101 % 7
Index = 3

Collision Occurs
Add to chain

Open Hashing/Separate Chaining

- **Advantages:**
  - Simple to implement
  - It is used it is unknown how many and how frequently key may be inserted and deleted.

- **Disadvantages:**
  - Wastage of space (Some part of hash table are never used.)
  - Use extra space for links.
  - Linked lists could get long. Longer linked lists could negatively impact performance
  - If the chain becomes long; then search time become 0(n) in worst case.

Closed Hashing/Open Addressing

- Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets).

- If the index given by the hash function is occupied, then there is need to find another bucket for the element to be stored. Then increment the table position by some number.

- Probing is just a way of resolving a collision when hashing values into bucket.

Closed Hashing/Open Addressing

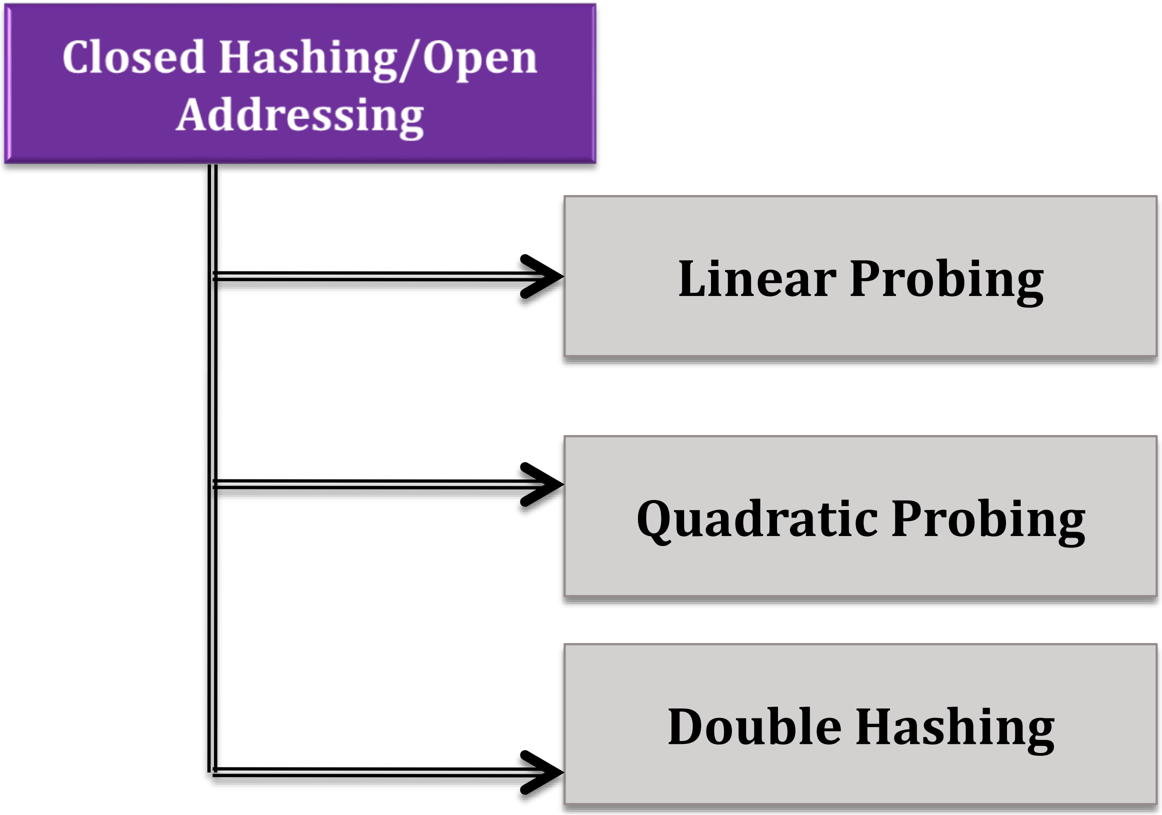
- In this, a has table wit predefined size is considered.

- All items are stored in the hash table itself.

- In addition to the data, each bucket also maintains 3 states.
  - EMPTY
  - OCCUPIED
  - DELETED

- While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found.

Closed Hashing/Open Addressing


Closed Hashing/Open Addressing

Linear Probing

Quadratic Probing

Double Hashing

Closed Hashing/Open Addressing

Linear Probing:

- It uses a technique "Probing".

- The next slot for the collided key is found in this method by using a technique called "Probing".

- Suppose that a key hashes into a position that has been already occupied.
  - The simplest strategy is to look for the next available position to place the item.

Closed Hashing/Open Addressing

Linear Probing:

- In linear probing, the rehashing process is linear.

- Location found at any step = n.
  - And n is occupied then the next attempt will be hash at position (n+1)

Rehashing (key) = (n+1) % Table_Size

Closed Hashing/Open Addressing

Linear Probing: Example

□ {89, 18, 49, 58, 9}

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Linear Probing: Example

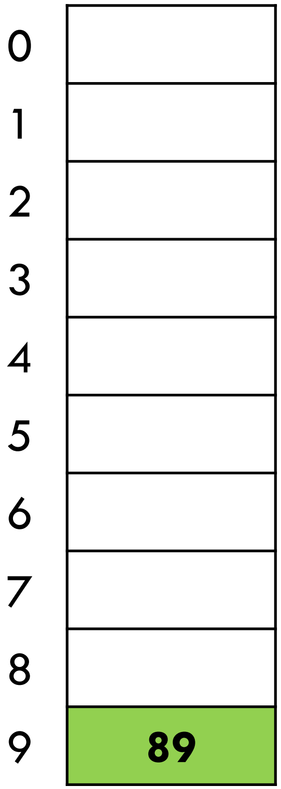
- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9



0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Insert 89

Closed Hashing/Open Addressing

Linear Probing: Example

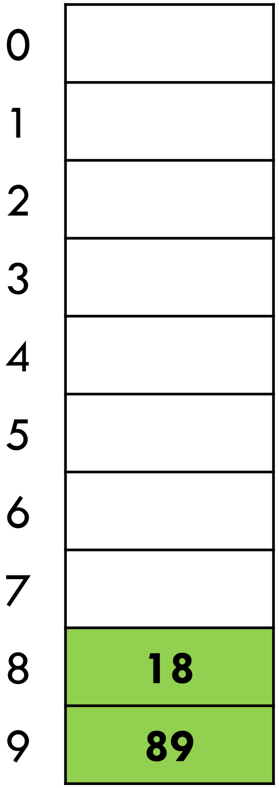
- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	**89**

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8



0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 18

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8
  - Hash(49,10) = 9

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8
  - Hash(49,10) = 9

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 49

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8
  - Hash(49,10) = 9
  - Hash(58,10) = 8

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	**18**
9	**89**

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8
  - Hash(49,10) = 9
  - Hash(58,10) = 8

0	49
1	58
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 58

Closed Hashing/Open Addressing

Linear Probing: Example

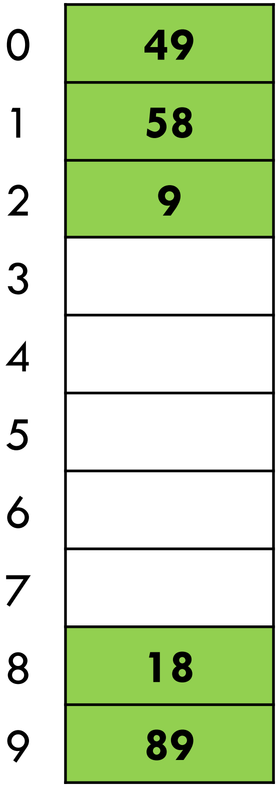
- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8
  - Hash(49,10) = 9
  - Hash(58,10) = 8
  - Hash(9,10) = 9

0	**49**
1	**58**
2	
3	
4	
5	
6	
7	
8	**18**
9	**89**

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}
  - Hash(89,10) = 9
  - Hash(18,10) = 8
  - Hash(49,10) = 9
  - Hash(58,10) = 8
  - Hash(9,10) = 9



0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

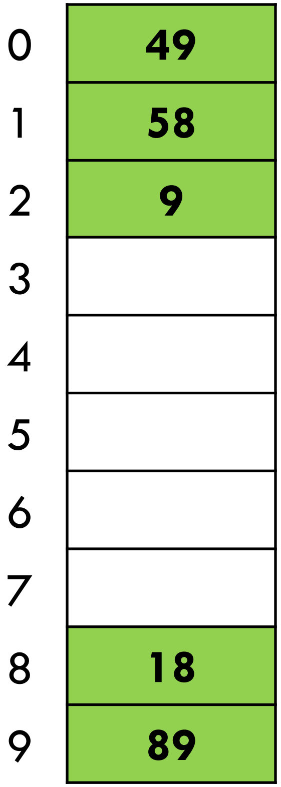
Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 49,

- Compute hash code=9
- Look in A[9]



0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 49,

  - Compute hash code=9
  - Look in A[9]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

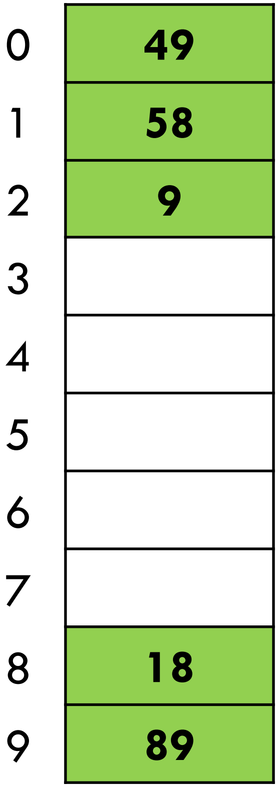
Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 49,

- Compute hash code=9
- Look in A[9]
- If not Found it there,
  - A[(9+1) % 10] =A[0]



0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

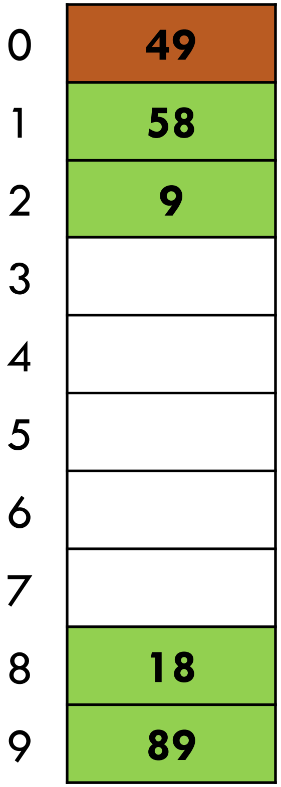
Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 49,

- Compute hash code=9
- Look in A[9]
- If not Found it there,
  - A[(9+1) % 10] =A[0]
  - Found and done.



0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 49,

- Compute hash code=9
- Look in A[9]
- If not Found it there,
  - A[(9+1) % 10] =A[0]
  - Found and done.

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

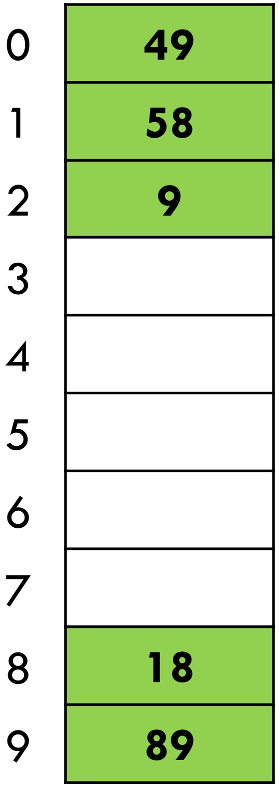
Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]



0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]
- Search at A[(9+2 %10]=A[1]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]
- Search at A[(9+2 %10]=A[1]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]
- Search at A[(9+2 %10]=A[1]
- Search at A[(9+3 %10]=A[2]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

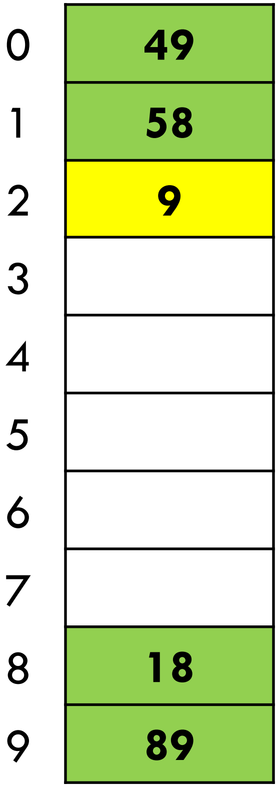
Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]
- Search at A[(9+2 %10]=A[1]
- Search at A[(9+3 %10]=A[2]



0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]
- Search at A[(9+2 %10]=A[1]
- Search at A[(9+3 %10]=A[2]
- Search at A[(9+4 %10]=A[3]

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

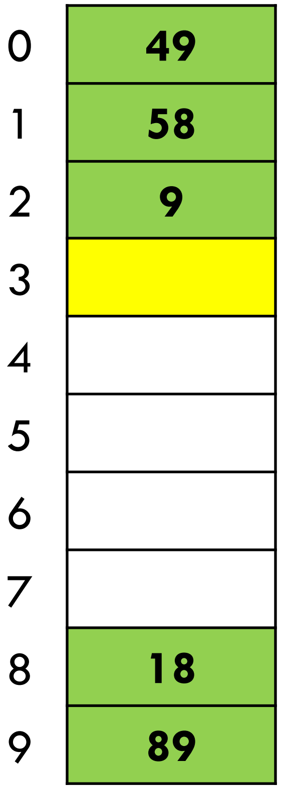
Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]
- Search at A[(9+2 %10]=A[1]
- Search at A[(9+3 %10]=A[2]
- Search at A[(9+4 %10]=A[3]



0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing: Example

- {89, 18, 49, 58, 9}

To find element 79,

- Compute hash code of 79 =9
- Search at A[9]
- Search at A[(9+1 %10]=A[0]
- Search at A[(9+2 %10]=A[1]
- Search at A[(9+3 %10]=A[2]
- Search at A[(9+4 %10]=A[3]
- And so on

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Insert 9

Closed Hashing/Open Addressing

Linear Probing with Replacement: Example

Construct hash table of size 10 using **linear probing with replacement strategy** for collision resolution. The hash function is h (x) = x % 10. Calculate total numbers of comparisons required for searching. Consider slot per bucket is 1.

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

- □ Table Size=10
- □ Slot per Bucket=1
- □ Total no. of Comparisons=???

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1										
2										
3										
4										
5	25									
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, **21**, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1										
2										
3		**3**								
4										
5	25	25								
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, **13**, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21							
2										
3		3	3							
4										
5	25	25	25							
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, **13**, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21						
2										
3		3	3	3						
4										
5	25	25	25	25						
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21						
2										
3		3	3	3						
4				**13**						
5	25	25	25	25						
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21					
2										
3		3	3	3	3					
4				13	13					
5	25	25	25	25	25					
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	**21**					
2					1					
3		3	3	3	3					
4				13	13					
5	25	25	25	25	25					
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21				
2					1	1				
3		3	3	3	3	3				
4				13	13	13				
5	25	25	25	25	25	25				
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21				
2					1	2				
3		3	3	3	3	3				
4				13	13	13				
5	25	25	25	25	25	25				
6										
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21				
2					1	2				
3		3	3	3	3	3				
4				13	13	13				
5	25	25	25	25	25	25				
6						1				
7										
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21			
2					1	2	2			
3		3	3	3	3	3	3			
4				13	13	13	13			
5	25	25	25	25	25	25	25			
6						1	1			
7							7			
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21	21		
2					1	2	2	2		
3		3	3	3	3	3	3	3		
4				13	13	13	13	13		
5	25	25	25	25	25	25	25	25		
6						1	1	1		
7							7	7		
8										
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21	21		
2					1	2	2	2		
3		3	3	3	3	3	3	3		
4				13	13	13	13	13		
5	25	25	25	25	25	25	25	25		
6						1	1	1		
7							7	7		
8								12		
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21	21	21	
2					1	2	2	2	2	
3		3	3	3	3	3	3	3	3	
4				13	13	13	13	13	13	
5	25	25	25	25	25	25	25	25	25	
6						1	1	1	1	
7							7	7	7	
8								**12**	**12**	
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21	21	21	
2					1	2	2	2	2	
3		3	3	3	3	3	3	3	3	
4				13	13	13	13	13	**4**	
5	25	25	25	25	25	25	25	25	25	
6						1	1	1	1	
7							7	7	7	
8								12	12	
9										

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21	21	21	
2					1	2	2	2	2	
3		3	3	3	3	3	3	3	3	
4				13	13	13	13	13	**4**	
5	25	25	25	25	25	25	25	25	25	
6						1	1	1	1	
7							7	7	7	
8								12	12	
9									13	

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	12
9									13	13

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	**8**
9									13	13

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										12
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	**8**
9									13	13

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										12
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	8
9									13	13
Compr	0	0	0							

Chaining with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										12
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	8
9									13	13
Compr	0	0	0	1	1					

Linear Probing with Replacement: Example
Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										12
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	8
9									13	13
Compr	0	0	0	1	1	4				

Linear Probing with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

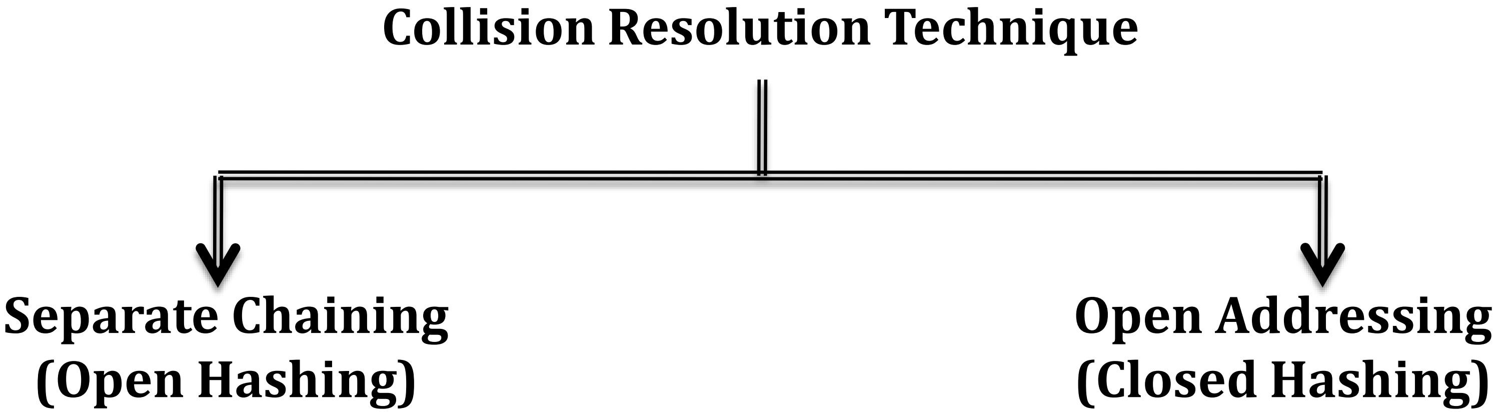
Bucket	25	3	21	13	1	2	7	12	4	8
0										12
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	8
9									13	13
Compr	0	0	0	1	1	4	0	6		

Linear Probing with Replacement: Example

Data: {25, 3, 21, 13, 1, 2, 7, 12, 4, 8}

Bucket	25	3	21	13	1	2	7	12	4	8
0										12
1			21	21	21	21	21	21	21	21
2					1	2	2	2	2	2
3		3	3	3	3	3	3	3	3	3
4				13	13	13	13	13	4	4
5	25	25	25	25	25	25	25	25	25	25
6						1	1	1	1	1
7							7	7	7	7
8								12	12	8
9									13	13
Compr	0	0	0	1	1	4	0	6	5	4

Collision Resolution Strategies



Collision Resolution Technique

Separate Chaining
(Open Hashing)

Open Addressing
(Closed Hashing)

- **Chaining:** Stores colliding keys in Linked List at the same table index.
- **Open Addressing:** Store Colliding keys elsewhere in the table.

Closed Hashing/Open Addressing

- Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets).

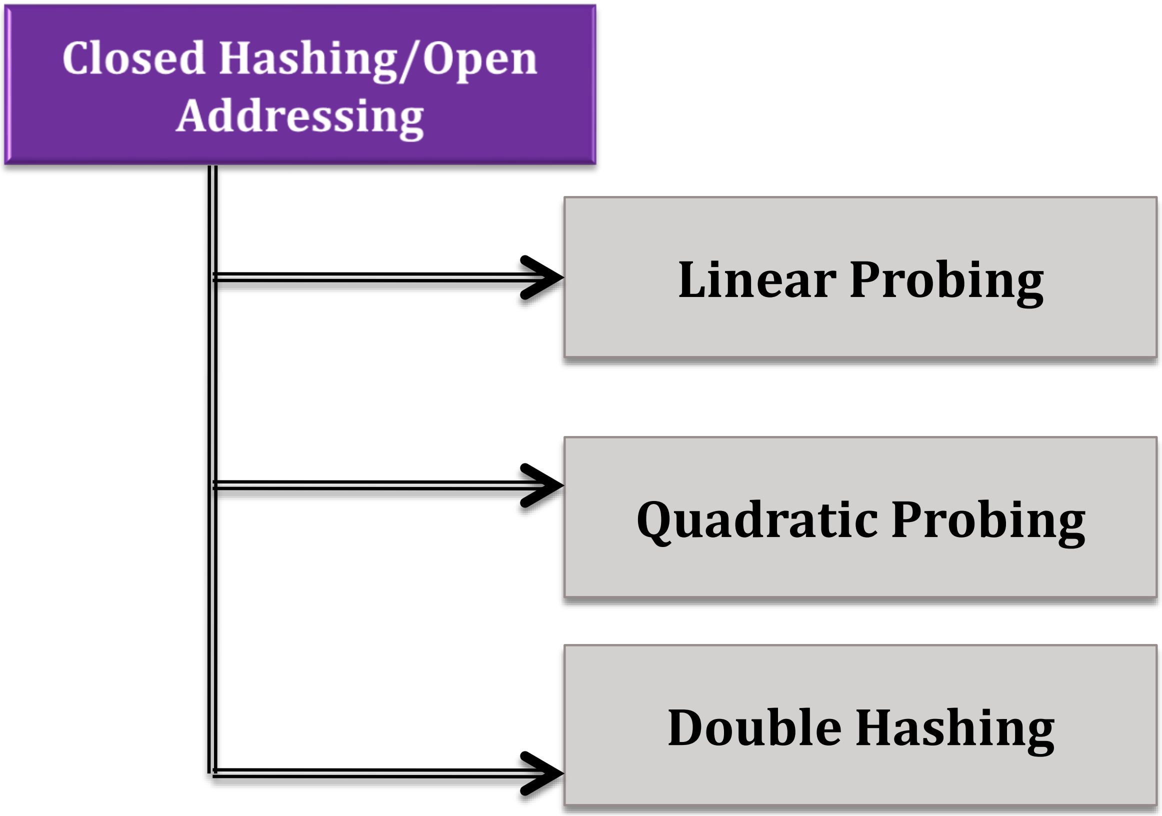
- If the index given by the hash function is occupied, then there is need to find another bucket for the element to be stored. Then increment the table position by some number.

- Probing is just a way of resolving a collision when hashing values into bucket.

Closed Hashing/Open Addressing

- In this, a has table wit predefined size is considered.
- All items are stored in the hash table itself.
- In addition to the data, each bucket also maintains 3 states.
  - EMPTY
  - OCCUPIED
  - DELETED
- While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found.

Closed Hashing/Open Addressing



Closed Hashing/Open Addressing

Linear Probing

Quadratic Probing

Double Hashing

Closed Hashing/Open Addressing

Linear Probing:

- It uses a technique “Probing”.
- The next slot for the collided key is found in this method by using a technique called "Probing".
- Suppose that a key hashes into a position that has been already occupied.
  - The simplest strategy is to look for the next available position to place the item.

Closed Hashing/Open Addressing

Linear Probing:

- In linear probing, the rehashing process is linear.

- Location found at any step = n.
  - And n is occupied then the next attempt will be hash at position (n+1)

Rehashing (key) = (n+1) % Table_Size

Closed Hashing/Open Addressing

LP with Replacement: Example

For the given set of values. 11, 33, 20, 88, 79, 98, 44, 68, 66, 22

Create a hash table with size 10 and resolve collision using Linear probing with replacement and without replacement. Use the modulus Hash function (key % size).

- Table Size=10

Chaining without Replacement: Example

Data: $\{11, 33, 20, 88, 79, 98, 44, 68, 66, 22\}$

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11									
2										
3										
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11	11								
2										
3										
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11	11								
2										
3		**33**								
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11	11	11							
2										
3		33	33							
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20							
1	11	11	11							
2										
3		33	33							
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20						
1	11	11	11	11						
2										
3		33	33	33						
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example
Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20						
1	11	11	11	11						
2										
3		33	33	33						
4										
5										
6										
7										
8				**88**						
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20					
1	11	11	11	11	11					
2										
3		33	33	33	33					
4										
5										
6										
7										
8				88	88					
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20					
1	11	11	11	11	11					
2										
3		33	33	33	33					
4										
5										
6										
7										
8				88	88					
9					**79**					

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, **98**, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20				
1	11	11	11	11	11	11				
2										
3		33	33	33	33	33				
4										
5										
6										
7										
8				88	88	88				
9					79	79				

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20				
1	11	11	11	11	11	11				
2										
3		33	33	33	33	33				
4										
5										
6										
7										
8				88	88	88				
9					79	79				

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, **98**, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20				
1	11	11	11	11	11	11				
2						**98**				
3		33	33	33	33	33				
4										
5										
6										
7										
8				88	88	88				
9					79	79				

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20			
1	11	11	11	11	11	11	11			
2						98	98			
3		33	33	33	33	33	33			
4										
5										
6										
7										
8				88	88	88	88			
9					79	79	79			

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, **44**, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20			
1	11	11	11	11	11	11	11			
2						98	98			
3		33	33	33	33	33	33			
4							44			
5										
6										
7										
8				88	88	88	88			
9					79	79	79			

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20		
1	11	11	11	11	11	11	11	11		
2						98	98	98		
3		33	33	33	33	33	33	33		
4							44	44		
5										
6										
7										
8				88	88	88	88	88		
9					79	79	79	79		

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20		
1	11	11	11	11	11	11	11	11		
2						98	98	98		
3		33	33	33	33	33	33	33		
4							44	44		
5										
6										
7										
8				88	88	88	88	88		
9					79	79	79	79		

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, **68**, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20		
1	11	11	11	11	11	11	11	11		
2						98	98	98		
3		33	33	33	33	33	33	33		
4							44	44		
5								**68**		
6										
7										
8				88	88	88	88	88		
9					79	79	79	79		

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	
1	11	11	11	11	11	11	11	11	11	
2						98	98	98	98	
3		33	33	33	33	33	33	33	33	
4							44	44	44	
5								68	68	
6										
7										
8				88	88	88	88	88	88	
9					79	79	79	79	79	

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	
1	11	11	11	11	11	11	11	11	11	
2						98	98	98	98	
3		33	33	33	33	33	33	33	33	
4							44	44	44	
5								68	68	
6									**66**	
7										
8				88	88	88	88	88	88	
9					79	79	79	79	79	

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	98
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	98
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	98
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										**22**
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Chaining with Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11									
2										
3										
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11	11								
2										
3										
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11	11								
2										
3		**33**								
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0										
1	11	11	11							
2										
3		33	33							
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, **20**, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			**20**							
1	11	11	11							
2										
3		33	33							
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20						
1	11	11	11	11						
2										
3		33	33	33						
4										
5										
6										
7										
8										
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20						
1	11	11	11	11						
2										
3		33	33	33						
4										
5										
6										
7										
8				**88**						
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20					
1	11	11	11	11	11					
2										
3		33	33	33	33					
4										
5										
6										
7										
8				88	88					
9										

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20					
1	11	11	11	11	11					
2										
3		33	33	33	33					
4										
5										
6										
7										
8				88	88					
9					**79**					

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, **98**, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20				
1	11	11	11	11	11	11				
2										
3		33	33	33	33	33				
4										
5										
6										
7										
8				88	88	88				
9					79	79				

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, **98**, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20				
1	11	11	11	11	11	11				
2										
3		33	33	33	33	33				
4										
5										
6										
7										
8				88	88	88				
9					79	79				

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, **98**, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20				
1	11	11	11	11	11	11				
2						**98**				
3		33	33	33	33	33				
4										
5										
6										
7										
8				88	88	88				
9					79	79				

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, **44**, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20			
1	11	11	11	11	11	11	11			
2						98	98			
3		33	33	33	33	33	33			
4										
5										
6										
7										
8				88	88	88	88			
9					79	79	79			

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20			
1	11	11	11	11	11	11	11			
2						98	98			
3		33	33	33	33	33	33			
4							44			
5										
6										
7										
8				88	88	88	88			
9					79	79	79			

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20		
1	11	11	11	11	11	11	11	11		
2						98	98	98		
3		33	33	33	33	33	33	33		
4							44	44		
5										
6										
7										
8				88	88	88	88	88		
9					79	79	79	79		

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, **68**, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20		
1	11	11	11	11	11	11	11	11		
2						98	98	98		
3		33	33	33	33	33	33	33		
4							44	44		
5										
6										
7										
8				88	88	88	88	88		
9					79	79	79	79		

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, **68**, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20		
1	11	11	11	11	11	11	11	11		
2						98	98	98		
3		33	33	33	33	33	33	33		
4							44	44		
5								**68**		
6										
7										
8				88	88	88	88	88		
9					79	79	79	79		

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	
1	11	11	11	11	11	11	11	11	11	
2						98	98	98	98	
3		33	33	33	33	33	33	33	33	
4							44	44	44	
5								68	68	
6										
7										
8				88	88	88	88	88	88	
9					79	79	79	79	79	

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, **66**, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	
1	11	11	11	11	11	11	11	11	11	
2						98	98	98	98	
3		33	33	33	33	33	33	33	33	
4							44	44	44	
5								68	68	
6									**66**	
7										
8				88	88	88	88	88	88	
9					79	79	79	79	79	

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	98
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	98
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, **22**}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	**22**
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	**22**
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										98
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Chaining without Replacement: Example

Data: {11, 33, 20, 88, 79, 98, 44, 68, 66, 22}

Bucket	11	33	20	88	79	98	44	68	66	22
0			20	20	20	20	20	20	20	20
1	11	11	11	11	11	11	11	11	11	11
2						98	98	98	98	22
3		33	33	33	33	33	33	33	33	33
4							44	44	44	44
5								68	68	68
6									66	66
7										98
8				88	88	88	88	88	88	88
9					79	79	79	79	79	79

Closed Hashing/Open Addressing

Quadratic Probing:

- Although linear probing is a simple process where it is easy to compute the next available location, linear probing also leads to some clustering when keys are computed to closer values.

- The problem of clustering can be avoided by quadratic probing.

- Therefore we define a new process of Quadratic probing that provides a better distribution of keys when collisions occur.

Closed Hashing/Open Addressing

Quadratic Probing:

- Therefore we define a new process of Quadratic probing that provides a better distribution of keys when collisions occur.

- Rehashing is applied as follows:

Rehashing(key) = (n + (k^2)) % Table_Size

where K= 1, 2, 3, ……

- We wrap around from last table location to first location if necessary.

Closed Hashing/Open Addressing

Quadratic Probing:

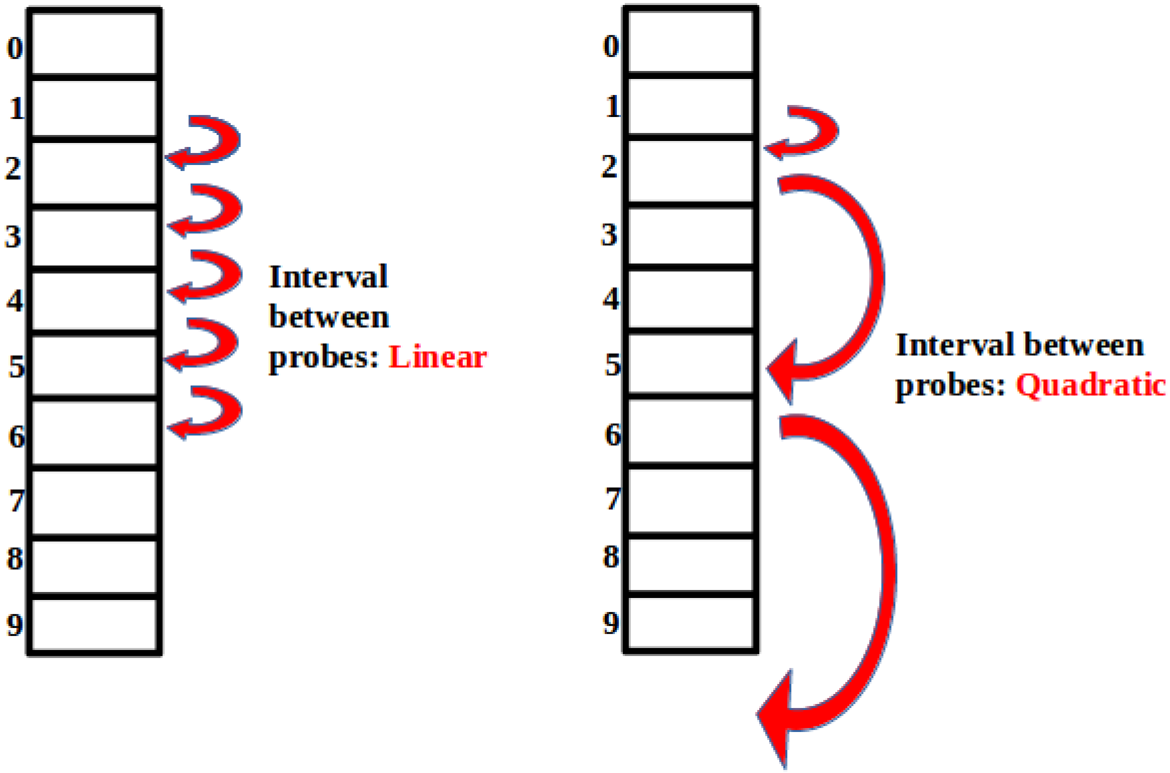
Rehashing(key) = (n + (k^2)) % Table_Size

where K= 1, 2, 3, ……

- If the hash value = k
- Then the next location = k+1
- Then the next location = k+4
- Then the next location = k+9 etc.

Closed Hashing/Open Addressing

Quadratic Probing:



Interval between probes: Linear

Interval between probes: Quadratic

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8
- Hash (49, 10) = 9

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8
- Hash (49, 10) = 9
  - For key = 9
    - Location 9 is occupied
  - Search for n+1=9+1=10 i.e.10 % 10= 0
    - Location 0 is empty, store at here.

Rehashing(key) = (n + (k^2)) % Table_Size

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8
- Hash (49, 10) = 9
  - For key =49
    - Location 9 is occupied
  - Search for n+1=9+1=10 i.e.10 % 10= 0
    - Location 0 is empty, store at here.

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Rehashing(key) = (n + (k^2)) % Table_Size

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8
- Hash (49, 10) = 9
- Hash (58, 10) = 8
  - For key =58
    - Location 8 is occupied
  - Search for n+1=8+1=9 i.e. 9 % 10= 9
    - Location 9 is occupied
  - Search for n+4=8+4=12 i.e. 12 % 10= 2
    - Location 2 is empty, store here

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Rehashing(key) = (n + (k^2)) % Table_Size

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8
- Hash (49, 10) = 9
- Hash (58, 10) = 8
  - For key =58
    - Location 8 is occupied
  - Search for n+1=8+1=9 i.e. 9 % 10= 9
    - Location 9 is occupied
  - Search for n+4=8+4=12 i.e. 12 % 10= 2
    - Location 2 is empty, store here

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

Rehashing(key) = (n + (k^2)) % Table_Size

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8
- Hash (49, 10) = 9
- Hash (58, 10) = 8
- Hash (9, 10) = 9
  - For key =9
    - Location 9 is occupied
  - Search for n+1=9+1=10 i.e. 10 % 10= 0
    - Location 0 is occupied
  - Search for n+4=9+4=13 i.e. 13 % 10= 3
    - Location 3 is empty, store here

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

Rehashing(key) = (n + (k^2)) % Table_Size

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 89, 18, 49, 58, 9} and Table_ Size=10

- Hash (89, 10) = 9
- Hash (18, 10) = 8
- Hash (49, 10) = 9
- Hash (58, 10) = 8
- Hash (9, 10) = 9
  - For key =9
    - Location 9 is occupied
  - Search for n+1=9+1=10 i.e. 10 % 10= 0
    - Location 0 is occupied
  - Search for n+4=9+4=13 i.e. 13 % 10= 3
    - Location 3 is empty, store here

0	49
1	
2	58
3	9
4	
5	
6	
7	
8	18
9	89

Rehashing(key) = (n + (k^2)) % Table_Size

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3

0	
1	
2	
3	123
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4

0	
1	
2	
3	123
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4

0	
1	
2	
3	123
4	124
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4
- Hash (333, 10) = 3
  - For key =3
    - Location 3 is occupied
  - Search for n+1=3+1=4   i.e. 4 % 10= 4
    - Location 4 is occupied
  - Search for n+4=3+4=7   i.e. 7 % 10= 7
    - Location 7 is empty, store here.

0	
1	
2	
3	123
4	124
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4
- Hash (333, 10) = 3
    - For key =3
        - Location 3 is occupied
    - Search for n+1=3+1=4 i.e. 4 % 10= 4
        - Location 4 is occupied
    - Search for n+4=3+4=7 i.e. 7 % 10= 7
        - Location 7 is empty, store here.

0	
1	
2	
3	123
4	124
5	
6	
7	333
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4
- Hash (333, 10) = 3= 3+4=7
- Hash (4679, 10) = 9

0	
1	
2	
3	123
4	124
5	
6	
7	333
8	
9	

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4
- Hash (333, 10) = 3 = 3+4=7
- Hash (4679, 10) = 9

0	
1	
2	
3	123
4	124
5	
6	
7	333
8	
9	4679

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4
- Hash (333, 10) = 3 = 3+4=7
- Hash (4679, 10) = 9
- Hash (983, 10) = 3
  - For key =3
    - Location 3 is occupied
  - Search for n+1=3+1=4 i.e. 4 % 10= 4
    - Location 4 is occupied
  - Search for n+4=3+4=7 i.e. 7 % 10= 7
    - Location 7 is occupied
  - Search for n+9=3+9=12 i.e. 12 % 10= 2
    - Location 2 is empty, store here.

0	
1	
2	
3	123
4	124
5	
6	
7	333
8	
9	4679

Closed Hashing/Open Addressing

Quadratic Probing: Example

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Hash (123, 10) = 3
- Hash (124, 10) = 4
- Hash (333, 10) = 3 = 3+4=7
- Hash (4679, 10) = 9
- Hash (983, 10) = 3
  - For key =3
    - Location 3 is occupied
  - Search for n+1=3+1=4 i.e. 4 % 10= 4
    - Location 4 is occupied
  - Search for n+4=3+4=7 i.e. 7 % 10= 7
    - Location 7 is occupied
  - Search for n+9=3+9=12 i.e. 12 % 10= 2
    - Location 2 is empty, store here.

0	
1	
2	983
3	123
4	124
5	
6	
7	333
8	
9	4679

Closed Hashing/Open Addressing

Double Hashing:

- It is best open addressing technique to overcome clustering chances.
- Here, we increment the probing length based on another hash function.
  - Primary hash function=h1
  - Secondary hash function=h2

f(key) = h1(key) + k * h2(key)
Where h1 != h2

  - Here h1 (key) = key % Table_Size

    h2 (key) = no. of digits in key

Closed Hashing/Open Addressing

Double Hashing:

f(key) = h1(key) + k * h2(key)
Where h1 != h2

- First we find h1(key)
- If its occupied, go for h1(key) + h2(key)
- If its still occupied, go for
  - h1(key) + 2 * h2 (key), so on…
- We can wrap up if necessary

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3

  123 % 10 = 3

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location ???

  123 % 10 = 3

0	
1	
2	
3	123
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location ???

  124 % 10 = 4

0	
1	
2	
3	123
4	
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4

124 % 10 = 4

0	
1	
2	
3	123
4	124
5	
6	
7	
8	
9	

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4
- Insert 333 at location ???

  333 % 10 = 3

  - Location 3 is already occupied
  - Next Location = 3 + 1. digits(333) =3+3 = 6
  - Insert 333 at location 6

0	
1	
2	
3	123
4	124
5	
6	
7	
8	
9	

f(key) = h1(key) + k * h2(key)

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4
- Insert 333 at location ???

  333 % 10 = 3

  - Location 3 is already occupied
  - Next Location = 3 + 1. digits(333) =3+3 = 6
  - Insert 333 at location 6

0	
1	
2	
3	123
4	124
5	
6	333
7	
8	
9	

f(key) = h1(key) + k * h2(key)

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4
- Insert 333 at location 6
- Insert 4679 at location ???

4679 % 10 = 9

0	
1	
2	
3	123
4	124
5	
6	333
7	
8	
9	

f(key) = h1(key) + k * h2(key)

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4
- Insert 333 at location 6
- Insert 4679 at location ???

4679 % 10 = 9

0	
1	
2	
3	123
4	124
5	
6	333
7	
8	
9	4679

f(key) = h1(key) + k * h2(key)

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4
- Insert 333 at location 6
- Insert 4679 at location 9
- Insert 983 at location ???

  983 % 10 = 3

  - Location 3 is already occupied
  - Next Location = 3 + 1*digits(983) =3+3 = 6 is already occupied
  - Next Location = 3 + 2* digits(983) =3+2*3 = 9 is already occupied
  - Next Location = 3 + 3 *digits(983) =3+3*3 = 12

    12 % 10 =2
  - Insert 983 at location 2

0	
1	
2	
3	123
4	124
5	
6	333
7	
8	
9	4679

f(key) = h1(key) + k * h2(key)

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4
- Insert 333 at location 6
- Insert 4679 at location 9
- Insert 983 at location ???

  983 % 10 = 3

  - Location 3 is already occupied
  - Next Location = 3 + 1*digits(983) =3+3 = 6 is already occupied
  - Next Location = 3 + 2* digits(983) =3+2*3 = 9 is already occupied
  - Next Location = 3 + 3 *digits(983) =3+3*3 = 12

    12 % 10 =2
  - Insert 983 at location 2

0	
1	
2	983
3	123
4	124
5	
6	333
7	
8	
9	4679

f(key) = h1(key) + k * h2(key)

Closed Hashing/Open Addressing

Double Hashing:

{ 123, 124, 333, 4679, 983} and Table_ Size=10

- Insert 123 at location 3
- Insert 124 at location 4
- Insert 333 at location 6
- Insert 4679 at location 9
- Insert 983 at location 2

0	
1	
2	983
3	123
4	124
5	
6	333
7	
8	
9	4679

Closed Hashing/Open Addressing

Double Hashing:

- Advantages
  - Clustering chances very less.

- Disadvantages
  - A new hashing function is overhead and take more time tan those two.