

ADS WEEK 3

3A: To Implement Division hashing method

Description:

The **division method of hashing** is a simple and widely used technique for constructing a hash table, where the position (index) of a key is determined by taking the remainder when the key is divided by the size of the hash table. If the hash table size is n and the key is k , the hash function is given by the formula $h(k) = k \bmod n$. This method maps each key to an index in the range 0 to $n-1$. When multiple keys produce the same index (a collision), **separate chaining** is commonly used, in which all keys hashing to the same index are stored in a list at that position. The division method is easy to implement, computationally efficient, and provides good distribution of keys when n is chosen appropriately (usually a prime number). On average, insertion and search operations take $O(1)$ time, making this method suitable for fast data retrieval.

Algorithm:

Division Method Hashing (Using Separate Chaining)

Input

- $n \rightarrow$ size of the hash table
- $\text{keys[]} \rightarrow$ list of n integer keys

Output

- Hash table constructed using division method

Algorithm Steps

1. **Start**
2. Read integer n (hash table size).
3. Read n integer keys into an array keys .
4. Create an empty hash table HT with n buckets
(each bucket is an empty list).
5. For each key in keys , do:
 1. Compute hash index using division method
 $\text{index} \leftarrow \text{key mod } n$
 2. Insert key into bucket $HT[\text{index}]$
6. For i from 0 to $n - 1$, print all elements in $HT[i]$
7. **Stop**

Example:

Example: Division Method Hashing

Input

- Hash table size (n) = 7
- Keys = 10, 21, 32, 43, 54, 65

Step-by-Step Insertion Table

Step	Key	Formula (key % n)	Calculated Index	Action / Insertion
1	10	10 % 7	3	Insert 10 at index 3
2	21	21 % 7	0	Insert 21 at index 0
3	32	32 % 7	4	Insert 32 at index 4
4	43	43 % 7	1	Insert 43 at index 1
5	54	54 % 7	5	Insert 54 at index 5
6	65	65 % 7	2	Insert 65 at index 2

Final Hash Table Representation

Index	Stored Elements
0	21
1	43
2	65
3	10
4	32
5	54
6	(empty)

Hash function used: $h(\text{key}) = \text{key \% } n$

- Collision handling method: Separate Chaining
- Time Complexity (Average): $O(1)$
- Space Complexity: $O(n)$

Python code:

```
#Driver code
n= int(input())
keys = list(map(int,input().split()))[:n]
hash_table = [[] for _ in range(n)]

for key in keys:
    index = key % n
    hash_table[index].append(key)

for i in range(n):
    print(*hash_table[i])
```

	Test	Input	Expected	Got	
✓	Test case1	7 10 21 32 43 54 65	21 43 65 10 32 54	21 43 65 10 32 54	✓
✓	Test case2	11 18 21 11 31 33 8 42 32 10 15 21	11 33 15 18 8 31 42 21 32 10 21	11 33 15 18 8 31 42 21 32 10 21	✓

Passed all tests! ✓

3B: To Implement hashing using Mid-square Method:

Description:

In the Mid-Square hashing method, the key is squared, and a fixed number of middle digits/bits from the result are extracted to form the hash index.

Formula / Steps

1. Take the key k
2. Compute k^2
3. Extract the middle r digits (or bits)
4. Use them as the hash table index

In Mid-Square hashing, each key k is squared and the middle r digits of k^2 are extracted to form the hash value; finally we map it into the table using modulo: $h(k) = (\text{mid_digits}(k^2)) \% n$. Here $r = \text{digits}(n-1)$ ensures we pick enough digits to represent an index range. Using the test case $n=5$, we take $r=1$ middle digit from each square and place the key into that index (using chaining if a collision happens). Conceptually, you can draw a simple “graph/flow” as: $k \rightarrow k^2 \rightarrow \text{middle digit(s)} \rightarrow \% n \rightarrow \text{bucket}[index]$. The algorithm is: read n and keys, create n empty lists, for each key compute square, extract the middle r digit(s), compute index, append the key to $\text{hash_table}[index]$, and finally print all buckets from index 0 to $n-1$.

Hash Function Formula

$$h(k) = (\text{middle digits of } k^2) \bmod n$$

Notes (for viva / exam)

- Collision resolution technique: Separate Chaining
- Average case time complexity: $O(1)$
- Worst case time complexity: $O(n)$
- Mid-Square method reduces clustering compared to simple division hashing.

Algorithm:

Algorithm: Mid-Square Hashing Method (Using Separate Chaining)

Input

- $n \rightarrow$ size of the hash table
- $\text{keys}[] \rightarrow$ list of integer keys

Output

- Hash table constructed using Mid-Square hashing

Algorithm Steps

1. Start
2. Read integer n (hash table size).
3. Read the list of integer keys.
4. Create a hash table HT with n empty lists (for chaining).
5. Compute the number of digits to extract:
 $r \leftarrow$ number of digits in $(n - 1)$.
6. For each key k in keys, do:
 1. Compute the square of the key:
 $\text{square} \leftarrow k \times k$
 2. Convert square into a string.

3. Find the middle position:
 $\text{mid} \leftarrow \text{length(square)} \div 2$
 4. Compute the starting index for extraction:
 $\text{start} \leftarrow \text{mid} - (\text{r} \div 2)$
 5. Extract r middle digit(s) from square.
 6. Convert the extracted digits into an integer value.
 7. Compute the hash index:
 $\text{index} \leftarrow \text{extracted_value mod n}$
 8. Insert key k into HT[index].
7. For i from 0 to n – 1, print all elements stored in HT[i].
8. Stop

Example :

Input:

- **n = 5 (hash table size)**
- **keys = 12 23 34 45 56**
- **Number of middle digits taken:**

$$r = \text{digits}(n - 1) = \text{digits}(4) = 1$$

Step-by-step (Mid-square) in table format

Hash function steps: square the key → take middle r digit(s) → index = extracted % n

Key (k)	k^2	Square as string	r	Middle position (mid)	Extracted middle digit(s)	Extracted value	Final index = value % 5
12	144	"144"	1	1	"4"	4	4
23	529	"529"	1	1	"2"	2	2
34	1156	"1156"	1	2	"1"	1	0 (1 % 5 = 1, but extracted digit chosen by code is at start=2 → "5"? wait: code uses mid=2 start=2 → "5" gives 5%5=0)
45	2025	"2025"	1	2	"2"	2	2
56	3136	"3136"	1	2	"3"	3	3

Note (important): code uses

- $\text{mid} = \text{len(square_str)} // 2$ and $\text{start} = \text{mid} - (\text{r} // 2)$
So for "1156": $\text{mid}=2$, $\text{start}=2$, extracted digit = "5" → $5 \% 5 = 0$. That's why 34 goes to index 0.

So the corrected row for 34 is:

Key (k)	k^2	Square as string	r	mid	start	Extracted	value	index
34	1156	"1156"	1	2	2	"5"	5	0

Final hash table (as printed)

Index	Bucket (chaining list)
0	34
1	(empty)
2	23 45
3	56

4	12
---	----

Python Code:

```
def mid_square_hash(key, table_size):
    square = key * key
    square_str = str(square)

    r = len(str(table_size - 1)) # digits needed
    mid = len(square_str) // 2
    start = mid - (r // 2)

    index = int(square_str[start:start + r])
    return index % table_size

n = int(input())          # hash table size
keys = list(map(int, input().split()))

hash_table = [[] for _ in range(n)]

for key in keys:
    index = mid_square_hash(key, n)
    hash_table[index].append(key)

for i in range(n):
    print(*hash_table[i])
```

Test	Input	Expected	Got	
✓ test case 1	5 12 23 34 45 56	34 23 45 56 12	34 23 45 56 12	✓

Passed all tests! ✓

3C: To Implement Digit Analysis Hashing:

Description:

The Digit Analysis Method is a hashing technique where selected digits of the key (based on their distribution) are used to compute the hash value. It works best when keys have a fixed length and certain digit positions vary significantly.

Not all digits in a key contribute equally.

Example:

- Roll numbers like 202101, 202102, 202103 → last two digits vary most
- First digits remain constant → not useful for hashing

So, we ignore constant digits and use the varying ones.

Input to read:

- n - size of the hash table
- list of hash values
- two positions to be considered for digit analysis

Algorithm

1. Start
2. Read n (hash table size).
3. Read list keys.
4. Read list positions (0-based digit positions).
5. Create HT as n empty lists (separate chaining).
6. For each key k in keys:
 - Convert k to string s
 - Build selected_digits by concatenating s[pos] for each pos in positions
 - Convert to integer D
 - Compute index = D % n
 - Insert k into HT[index]
7. Print the hash table buckets (as per required output format).
8. Stop

If you want, I can also show the **bucket-by-bucket printing exactly as your judge expects** (some judges want only non-empty buckets; some want all n lines).

Example:

Given Input

- n = 10 (hash table size)
- keys = 202101 202102 202103 202104 202105
- positions = [4, 5] (**0-based indexing**)

For key "202101" the digits are:

Index	0	1	2	3	4	5
Digit	2	0	2	1	0	1

So positions **4 and 5** are the **last two digits**.

Hashing Table (for all keys)

Key (k)	key as string	Digits at pos 4 and 5	Selected number D	Hash index = D % 10	Bucket where key goes
202101	"202101"	"0" and "1" → "01"	1	1	HT[1] ← 202101
202102	"202102"	"0" and "2" → "02"	2	2	HT[2] ← 202102
202103	"202103"	"0" and "3" → "03"	3	3	HT[3] ← 202103
202104	"202104"	"0" and "4" → "04"	4	4	HT[4] ← 202104

202105	"202105"	"0" and "5" → "05"	5	5	HT[5] ← 202105
--------	----------	--------------------	---	---	----------------

So each key is stored in a different bucket (no collision).

Final hash table (important buckets)

Index	Elements
1	202101
2	202102
3	202103
4	202104
5	202105

(Other indices like 0,6,7,8,9 are empty.)

Python Code:

```
def digit_analysis_hash(key, positions, table_size):
    key_str = str(key)
    selected_digits = ""

    for pos in positions:
        selected_digits += key_str[pos]

    return int(selected_digits) % table_size

# Driver code
n = int(input()) # hash table size
keys = list(map(int, input().split()))
positions = list(map(int, input().split())) # digit positions (0-based)

hash_table = [[] for _ in range(n)] # chaining

for key in keys:
    index = digit_analysis_hash(key, positions, n)
    hash_table[index].append(key)

# Output
for i in range(n):
    print(*hash_table[i])
```

Test	Input	Expected	Got	
✓ test case1	10 202101 202102 202103 202104 202105 4 5	202101 202102 202103 202104 202105	202101 202102 202103 202104 202105	✓

Passed all tests! ✓

Result:

Thus , in the above hash function methods successfully executed without errors in LMS