PART A:

1. What is a hash table and why is collision resolution necessary?

A hash table is an efficient data structure for storing data so that you can quickly find it by its key. It uses a hash function to convert a key into an index in an array. The problem comes when two keys get hashed to the same index this is called a collision. Collision resolution is necessary to ensure both keys get stored and can be retrieved correctly, even if they hash to the same spot.

2. Differences between open hashing (chaining) and closed hashing (open addressing).

Open Hashing (Chaining): In this method, each spot in the table holds a linked list of keys that all hash to the same index. Collisions are handled by adding the new key to the list.

Closed Hashing (Open Addressing): The keys are stored directly in the table, and if a collision happens, we search for another open spot using a method like linear probing (next spot), quadratic probing (check spots in a square), or double hashing (use a second hash function).

3. Collision Resolution Techniques:

A.

Linear Probing: If a key collides, you just check the next index. It's simple but can lead to clustering, where groups of keys form together and make it slower to find new keys.

Pros: Easy to implement.

Cons: Can cause clustering, which makes lookups slower.

B.

Quadratic Probing: Instead of just moving one slot over, you move by squares (like squares 4, 9, etc.). This spreads the keys out a little better.

Pros: Less clustering than linear probing.

Cons: Still don’t completely avoid it, and you might not be able to reach every slot.

C.

Double Hashing: Uses a second hash function to decide where to go next after a collision. This method spreads out the keys well.

Pros: Reduces clustering.

Cons: More complicated to implement

4. Which collision resolution technique can handle more values than table slots?

Open hashing (chaining) can handle more keys than slots because each slot is just a bucket (like a linked list), and it can grow as needed. In contrast, closed hashing (open addressing) has a fixed number of slots, so it can't handle more than what's available.

5. Worst performance (Big O) for each resolution technique?

Open Hashing (Chaining): The worst case is O(n) if all keys hash to the same index.

Linear Probing: The worst case is O(n) when the table is almost full, and you have to probe a lot of spots.

Quadratic Probing: The worst case is also O(n), though it happens less often than with linear probing.

Double Hashing: The worst case is O(n), but this is unlikely with a good second hash function.

6. How does the choice of table size affect the distribution of keys?

If the table size is too small or not prime, keys will hash to the same slots more often, causing more collisions. A prime table size often leads to better distribution, as it reduces the chance of patterns that lead to clustering.

7. Pitfalls of poor table sizes:

Using a power of 2 or round number (like 10) as the table size can cause patterns in key distribution, leading to more collisions and inefficient lookups. A prime number table size helps spread out the keys more evenly and reduces collisions.

PART B:

Exercise 1: Open Hashing (Separate Chaining)

5, 22, 17, 18, 35, 101, 16, 0, 8. h(k) = k mod 10.

5 mod 10 = 5

22 mod 10 = 2

17 mod 10 = 7

18 mod 10 = 8

35 mod 10 = 5 (Collision at index 5, so we add to the link list at index 5)

101 mod 10 = 1

16 mod 10 = 6

0 mod 10 = 0

8 mod 10 = 8 (Collision at index 8, so we add to the link list at index 8)

Index	Key
0	0
1	101
2	22
3	
4	
5	5 ->35
6	16
7	17
8	18 -> 8
9	

Exercise 2: Closed Hashing (Linear Probing)

5, 22, 17, 18, 35, 101, 16, 0, 8. h(k) = k mod 10.

5 mod 10 = 5, Insert at index 5

22 mod 10 = 2, Insert at index 2

17 mod 10 = 7, Insert at index 7

18 mod 10 = 8, Insert at index 8

35 mod 10 = 5, Collision at index 5, move to the next available slot: index 6

101 mod 10 = 1, Insert at index 1

16 mod 10 = 6, Collision at index 6, move to next available slot: index 7, Collision at index 7, move to next available slot: index 8, Collision at index 8, move to next available slot: index 9

0 mod 10 = 0, Insert at index 0

8 mod 10 = 8, Collision at index 8, move to next available slot: index 9, Collision at index 9, move to next available slot: index 10 (is 0), but index 0 is already filled, checking until it finds a free spot. Index 3 is free, so insert 8.

Index	Key
0	0
1	101
2	22
3	8
4	
5	5
6	35
7	17
8	18
9	16

Exercise 3:

A.

Keys to Insert: 5, 10, 15, 20, 25, 30, 35, 40.

Table Size = 10.

h(k) = k mod 10

5 mod 10 = 5, Insert at index 5

10 mod 10 = 0, Insert at index 0

15 mod 10 = 5, Collision at index 5, so insert at next available index (index 6)

20 mod 10 = 0, Collision at index 0, so insert at next available index (index 1)

25 mod 10 = 5, Collision at index 5, move to index 6 , Collision at index 6, move to index 7

30 mod 10 = 0 , Collision at index 0, move to index 1 , Collision at index 1, move to index 2

35 mod 10 = 5, Collision at index 5, move to index 6 , Collision at index 6, move to index 7, Collision at index 7, move to index 8

40 mod 10 = 0 , Collision at index 0, move to index 1 , Collision at index 1, move to index 2 , Collision at index 2, move to index 3

Index	Key
0	10
1	20
2	30
3	40
4	
5	5
6	15
7	25

8	35
9	

B.

Table Size = 11 (Prime Number)

h(k) = k mod 11

5 mod 11 = 5: Insert at index 5

10 mod 11 = 10: Insert at index 10

15 mod 11 = 4: Insert at index 4

20 mod 11 = 9: Insert at index 9

25 mod 11 = 3: Insert at index 3

30 mod 11 = 8: Insert at index 8

35 mod 11 = 2: Insert at index 2

40 mod 11 = 7: Insert at index 7

Index	Key
0	
1	
2	35
3	25
4	15
5	5
6	
7	40
8	30
9	20

10	10

The table size of 11 (a prime number) offers a more uniform distribution of keys, reducing collisions and clustering. The table size of 10 causes more collisions and leads to a less efficient hash table, especially when the table is getting filled. Using a prime number for the table size helps spread out the keys more evenly, leading to better performance in both open and closed hashing methods.

Extra Credit:

12, 23, 34, 45, 56, 67, 78, 89

h(k, i)=(h(k)+i^2)mod m

if m=10

h(k) = k mod 10.

1. Key 12:12 mod 10 = 2: Insert at index 2

2. Key 23:23 mod 10 = 3: Insert at index 3

3. Key 34:34 mod 10 = 4: Insert at index 4

4. Key 45:45 mod 10 = 5: Insert at index 5

5. Key 56:56 mod 10 = 6: Insert at index 6

6. Key 67:67 mod 10 = 7: Insert at index 7

7. Key 78:78 mod 10 = 8: Insert at index 8

8. Key 89:89 mod 10 = 9: Insert at index 9

Index	Key
0	
1	
2	12

3	23
4	34
5	45
6	56
7	67
8	78
9	89

No collisions happened, so no quadratic probing was needed.

Benefits:

Less Clustering: It reduces clustering (when all the keys group up in one area), so it’s generally more efficient than linear probing.

Efficient Use of Space: The keys are more evenly spread out, so there’s less chance of wasted space on the table.

Good for Moderate Load Factors: If the table isn’t too full, quadratic probing works smoothly.

Drawbacks:

Secondary Clustering: Even though it reduces primary clustering, there’s still something called secondary clustering (where certain keys keep trying the same spots). This can still cause issues.

Performance Degrades When Full: As the table fills up, quadratic probing can slow down. If the table's nearly full, it takes longer to find an open spot.

Prime Number Table Size Helps: For quadratic probing to work best, the table size should usually be a prime number. If the size isn’t prime, it can cause some keys to just never find an open slot.

PART C:

1.Why does choosing a poor hash table size cause problems?

More Collisions: When you choose a poor table size, you'll end up with more collisions (two products trying to go into the same spot). This means the hash table has to keep looking for other places to put things, which takes more time.

Clustering: If your table size is a power of 2 or some round number, you'll get something called clustering, where all your products bunch up in the same area.

Resizing Costs: If the table gets too full and you don’t resize it, performance drops because it has to search harder for empty spots. If the table's too full, you'll even have to resize the whole thing, which can be complex.

2. How do open and closed hashing handle collisions differently in high-load situations?

Open Hashing (Separate Chaining): As the table fills up, more collisions happen, so the linked lists can get longer. If they get too long, finding the right product might take a little more time, but overall, it’s flexible. However, once the linked list is long enough, searching becomes slower.

Closed Hashing (Open Addressing): As the table fills up, collisions increase, and the search for empty slots can become a pain. If you're not careful, things can get clustered together, which becomes more complex.

3. If you were to design a hash table for a high-frequency trading system where every millisecond counts, which collision resolution strategy might you choose and why?

For a high-frequency trading system where every millisecond counts, I will go with open addressing, specifically double hashing, or quadratic probing. Here's why:

Faster Lookups: Open addressing keeps everything in the table itself, so lookups are really fast (constant time, O (1) in most cases).

No Extra Memory: No need for linked list likes in separate chaining, which can cause unpredictable delays and extra memory usage.

Reduced Clustering: Double hashing or quadratic probing minimizes collisions and clustering, ensuring keys are evenly spread out.

4. How might you combine the benefits of chaining and open addressing to design a hybrid hash table? (Brief discussion)

A hybrid hash table mixes open addressing and chaining to get the best of both:

Start with Open Addressing: When the table has lots of empty space, use open addressing for quick lookups and less memory use.

Switch to Chaining: If too many collisions happen and a bucket gets too full, switch to chaining (linked lists) to handle the collisions better.

Switch When Needed: The table can change between methods based on how full it is, making sure it works well in different situations.