**HashQuestions**

**Linear Probing**

**1.You are implementing a simple hash table using linear probing for a library system. The hash table represents the availability of books in the library. Each book is identified by a unique integer key. The library has a fixed number of shelves, and each shelf corresponds to an index in the hash table.**

**Input Format:**

The first line contains an integer M representing the number of shelves in the library.

Subsequent lines contain operations to be performed on the hash table. Each operation is represented by a string ("INSERT" or "SEARCH") followed by an integer key.

**Output Format:**

For each "SEARCH" operation, output "true" if the book is available on a shelf, and "false" otherwise.

**CODE:**

import java.util.Scanner;

class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int M = scanner.nextInt();

int[] hashTable = new int[M];

while (scanner.hasNext()) {

String operation = scanner.next();

if (scanner.hasNextInt()) {

int x = scanner.nextInt();

if (operation.equals("INSERT")) {

int index = x % M;

while (hashTable[index] != 0) {

index = (index + 1) % M; // Linear probing

}

hashTable[index] = x;

} else if (operation.equals("SEARCH")) {

int index = x % M;

while (hashTable[index] != x && hashTable[index] != 0) {

index = (index + 1) % M; // Linear probing

}

System.out.println(hashTable[index] == x);

}

} else {

// Handle the case where the expected integer is not provided

System.out.println("Invalid input. Expected an integer.");

scanner.next(); // Consume the invalid input to avoid an infinite loop

}

}

}

}

TESTCASES

### Test Case 1:

\*\*Input:\*\*

4

INSERT 8

INSERT 4

SEARCH 12

INSERT 12

SEARCH 12

\*\*Output:\*\*

false

true

### Test Case 2:

\*\*Input:\*\*

6

INSERT 9

INSERT 3

INSERT 15

SEARCH 9

INSERT 21

SEARCH 21

\*\*Output:\*\*

true

true

### Test Case 3:

\*\*Input:\*\*

5

INSERT 5

INSERT 13

INSERT 21

SEARCH 5

SEARCH 21

SEARCH 16

\*\*Output:\*\*

true

true

false

### Test Case 4:

\*\*Input:\*\*

3

INSERT 7

INSERT 10

SEARCH 1

SEARCH 10

\*\*Output:\*\*

false

true

### Test Case 5:

\*\*Input:\*\*

7

INSERT 14

INSERT 21

SEARCH 7

SEARCH 14

INSERT 7

SEARCH 7

\*\*Output:\*\*

false

true

true

### Test Case 6:

\*\*Input:\*\*

8

INSERT 10

INSERT 15

SEARCH 10

SEARCH 12

\*\*Output:\*\*

true

false

**2. Alice works at a library and needs to efficiently catalog a list of books. Each book has a unique ID number. To quickly access information about a specific book, she decides to implement a hash table using linear probing. However, to prevent the library from having to constantly reallocate memory as the collection grows, the hash table's size is fixed at 20.**

**INPUT FORMAT :**

Help Alice insert the following book IDs into the hash table: 12, 17, 35, 28, 10, 5, 9, 14, 42, 31. Assume a simple modulo function h(key) = key % 20 for generating the initial index.

After each insertion, print the updated hash table state, highlighting the probing sequence if a collision occurs.

After all insertions, analyze the performance of the hash table based on the number of probing steps per operation.  
  
**Output Format:**

For each insertion, print:

The book ID being inserted.

The initial index calculated using the hash function.

If a collision occurs, print the complete probing sequence until an empty slot is found.

Finally, print the updated hash table with the inserted book in its final position.

After all insertions, calculate and print the average number of probing steps per operation.  
 **CODE:**

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

try {

System.out.print("Enter the size of the hash table: ");

int size = scanner.nextInt();

if (size <= 0) {

System.out.println("Invalid size. Size must be a positive integer.");

return;

}

int[] hashTable = new int[size];

System.out.println("Enter 10 integer keys:");

for (int i = 0; i < 10; i++) {

if (scanner.hasNextInt()) {

int key = scanner.nextInt();

linearProbingInsert(hashTable, key, size);

printHashTable(hashTable);

} else {

System.out.println("Invalid input. Please provide an integer.");

scanner.next(); // Consume the invalid input to avoid an infinite loop

}

}

} catch (Exception e) {

System.out.println("An error occurred: " + e.getMessage());

e.printStackTrace(); // Print the stack trace

} finally {

scanner.close();

}

}

private static void linearProbingInsert(int[] hashTable, int key, int size) {

int index = key % size;

while (hashTable[index] != 0) {

// Linear probing: move to the next slot if the current slot is occupied

index = (index + 1) % size;

}

// Insert the key into the found empty slot

hashTable[index] = key;

}

private static void printHashTable(int[] hashTable) {

System.out.print("Hash Table: [ ");

for (int i : hashTable) {

System.out.print(i + " ");

}

System.out.println("]");

}

}

Test Case 1:

Input:

10

12 17 35 28 10 5 9 14 42 31

Expected Output:

Hash Table: [ 12 0 0 0 0 17 0 0 0 0 ]

Hash Table: [ 12 0 0 0 0 17 35 0 0 0 ]

Hash Table: [ 12 0 0 0 0 17 35 28 0 0 ]

Hash Table: [ 12 0 0 0 0 17 35 28 0 10 ]

Hash Table: [ 12 0 0 0 0 17 35 28 0 10 ]

Hash Table: [ 12 0 0 0 0 17 35 28 5 10 ]

Hash Table: [ 12 9 0 0 0 17 35 28 5 10 ]

Hash Table: [ 12 9 0 0 14 17 35 28 5 10 ]

Hash Table: [ 12 9 0 0 14 17 35 28 5 10 ]

Hash Table: [ 12 9 0 0 14 17 35 28 42 10 ]

Test Case 2:

Input:

8

3 6 9 12 15 18 21 24

Expected Output:

Hash Table: [ 3 0 0 0 0 0 0 0 ]

Hash Table: [ 3 6 0 0 0 0 0 0 ]

Hash Table: [ 3 6 9 0 0 0 0 0 ]

Hash Table: [ 3 6 9 12 0 0 0 0 ]

Hash Table: [ 3 6 9 12 15 0 0 0 ]

Hash Table: [ 3 6 9 12 15 18 0 0 ]

Hash Table: [ 3 6 9 12 15 18 21 0 ]

Hash Table: [ 3 6 9 12 15 18 21 24 ]

Test Case 3:

Input:

6

5 10 15 20 25 30

Expected Output:

Hash Table: [ 5 0 0 0 0 0 ]

Hash Table: [ 5 10 0 0 0 0 ]

Hash Table: [ 5 10 15 0 0 0 ]

Hash Table: [ 5 10 15 20 0 0 ]

Hash Table: [ 5 10 15 20 25 0 ]

Hash Table: [ 5 10 15 20 25 30 ]

Test Case 4:

Input:

4

2 4 6 8

Expected Output:

Hash Table: [ 2 0 0 0 ]

Hash Table: [ 2 4 0 0 ]

Hash Table: [ 2 4 6 0 ]

Hash Table: [ 2 4 6 8 ]

**QUADRATIC HASHING**

**3. joy is developing a simple spell checker program using Quadratic Probing Hash in Java. She wants to efficiently check whether a given word is present in the dictionary or not. To implement the spell checker, she decides to use Quadratic Probing Hash table.**

Task:

Implement the spell checker program using Quadratic Probing Hash in Java.

Input Format:

The first line contains an integer N (1 ≤ N ≤ 1000), denoting the number of words in the dictionary.

The next N lines contain the words in the dictionary.

The next line contains an integer Q (1 ≤ Q ≤ 100), denoting the number of words to check.

The next Q lines contain the words to be checked for spelling.

Output Format:

For each word to be checked, output "Yes" if the word is found in the dictionary, otherwise output "No".

CODE

import java.util.Scanner;

public class QuadraticDemo1 {

private static final int TABLE\_SIZE = 1000;

private static String[] dictionary = new String[TABLE\_SIZE];

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input: Number of words in the dictionary and dictionary words

int n = 0;

if (scanner.hasNextInt()) {

n = scanner.nextInt();

scanner.nextLine(); // Consume newline

for (int i = 0; i < n; i++) {

dictionary[i] = scanner.nextLine();

}

} else {

System.out.println("Invalid input for number of dictionary words.");

return; // Exit the program

}

// Input: Number of words to check and words to check

int q = 0;

if (scanner.hasNextInt()) {

q = scanner.nextInt();

scanner.nextLine(); // Consume newline

for (int i = 0; i < q; i++) {

String word = scanner.nextLine();

System.out.println(checkSpelling(word) ? "Yes" : "No");

}

} else {

System.out.println("Invalid input for number of words to check.");

return; // Exit the program

}

}

// Quadratic probing hash function

private static int hashFunction(String word) {

int hash = 0;

for (int i = 0; i < word.length(); i++) {

hash = (hash \* 31 + word.charAt(i)) % TABLE\_SIZE;

}

return hash;

}

// Spell checker function using Quadratic Probing

private static boolean checkSpelling(String word) {

int hashIndex = hashFunction(word);

int i = 1;

int newIndex = hashIndex;

while (dictionary[newIndex] != null && !dictionary[newIndex].equals(word)) {

newIndex = (hashIndex + i \* i) % TABLE\_SIZE;

i++;

}

return dictionary[newIndex] != null && dictionary[newIndex].equals(word);

}

}

SampleInput

5

cat

dog

bird

fish

elephant

3

cat

lion

fish

SampleOutput

Yes

No

Yes

Test Cases:

Test Case 1:

**Input:**

5

apple

banana

orange

grape

peach

3

apple

kiwi

grape

**Output**

Yes

No

Yes

**TestCase2**

**Input**

3

red

green

blue

4

red

yellow

green

black

Output

Yes

No

Yes

No

Test Case 3:

Input:

4

sun

moon

stars

clouds

3

moon

sky

stars

Output:

Yes

No

Yes

Test Case 4:

Input:

bash

Copy code

2

dog

cat

2

puppy

kitten

Output:

No

No

**4. John is developing a spell checker using a hash table with quadratic probing. She needs to efficiently check whether a list of words is present in the dictionary. Implement a spell checker program using quadratic probing to achieve this.**

**Input Format:**

The first line contains an integer N, denoting the number of words in the dictionary.

The next N lines contain the words in the dictionary.

The following line contains an integer Q, denoting the number of words to check.

The next Q lines contain the words to check.

**Output Format:**

For each word to check, print "Yes" if the word is found in the dictionary, and "No" otherwise.

import java.util.Scanner;

public class SongSearchSystem {

private static final int TABLE\_SIZE = 1000;

private static final Song[] library = new Song[TABLE\_SIZE];

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input: Number of songs in the library and songs to add

int n = scanner.nextInt();

scanner.nextLine(); // Consume newline

for (int i = 0; i < n; i++) {

String songTitle = scanner.nextLine();

addToLibrary(songTitle);

}

// Input: Number of queries and songs to search for

int q = scanner.nextInt();

scanner.nextLine(); // Consume newline

for (int i = 0; i < q; i++) {

String query = scanner.nextLine();

System.out.println(searchSong(query) ? "Found" : "Not Found");

}

}

// Quadratic probing hash function

private static int hashFunction(String songTitle) {

int hash = 0;

for (int i = 0; i < songTitle.length(); i++) {

hash = (hash \* 31 + songTitle.charAt(i)) % TABLE\_SIZE;

}

return hash;

}

// Add song to the library using quadratic probing

private static void addToLibrary(String songTitle) {

int hashIndex = hashFunction(songTitle);

int i = 1;

int newIndex = hashIndex;

while (library[newIndex] != null && library[newIndex].isFilled) {

newIndex = (hashIndex + i \* i) % TABLE\_SIZE;

i++;

}

library[newIndex] = new Song(songTitle);

}

// Search for a song in the library using quadratic probing

private static boolean searchSong(String songTitle) {

int hashIndex = hashFunction(songTitle);

int i = 1;

int newIndex = hashIndex;

while (library[newIndex] != null) {

if (library[newIndex].title.equals(songTitle) && library[newIndex].isFilled) {

return true;

}

newIndex = (hashIndex + i \* i) % TABLE\_SIZE;

i++;

}

return false;

}

// Song class representing a song entry in the library

static class Song {

String title;

boolean isFilled;

public Song(String title) {

this.title = title;

this.isFilled = true;

}

}

}

**Test Case 1:**

**Input:**

3

song1

song2

song3

2

song1

song4

**Output:**

Found

Not Found

**Test Case 2:**

**Input:**

5

hello

world

java

programming

language

3

world

programming

python

**Output:**

Found

Found

Not Found

**Test Case 3:**

**Input:**

4

apple

banana

orange

grape

2

apple

pear

**Output:**

Found

Not Found

**Test Case 4:**

**Input:**

2

song1

song2

3

song1

song2

song3

**Output:**

Found

Found

Not Found

**DOUBLE HASHING**

**5.** **You are implementing a hash table using double hashing for storing integers. Write a Java program to perform the following operations:**

1.Initialize the hash table with a given size.

2.Insert a given list of integers into the hash table.

3.Search for a specific integer in the hash table.

Your program should implement the following functions:

initializeTable(int size): Initializes the hash table with the given size.

insert(int key): Inserts the given integer into the hash table using double hashing.

search(int key): Searches for the given integer in the hash table using double hashing.

Write a Java program that reads input from the standard input as follows:

The first line contains an integer n representing the size of the hash table.

The second line contains n integers separated by spaces, representing the list of integers to insert into the hash table.

The third line contains an integer m representing the number of queries.

The next m lines contain integers representing the keys to search in the hash table.

For each query, print "Found" if the key is present in the hash table, otherwise print "Not Found".

Input Format:

The first line contains an integer n (1 <= n <= 1000) representing the size of the hash table.

The second line contains n integers separated by spaces (-10^9 <= key <= 10^9) representing the list of integers to insert into the hash table.

The third line contains an integer m (1 <= m <= 1000) representing the number of queries.

The next m lines contain integers (-10^9 <= key <= 10^9) representing the keys to search in the hash table.

Output Format:

For each query, print "Found" if the key is present in the hash table, otherwise print "Not Found".

CODE

import java.util.Scanner;

public class DoubleHashingDemo {

private static int[] hashTable;

private static int tableSize;

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input: Size of the hash table

tableSize = scanner.nextInt();

hashTable = new int[tableSize];

// Input: Insert elements into the hash table

for (int i = 0; i < tableSize; i++) {

int key = scanner.nextInt();

insert(key);

}

// Input: Number of queries

int queries = scanner.nextInt();

// Input: Search for keys in the hash table

for (int i = 0; i < queries; i++) {

int key = scanner.nextInt();

System.out.println(search(key) ? "Found" : "Not Found");

}

}

// Double hashing function to get the hash index

private static int doubleHash(int key, int i) {

// First hash function

int hash1 = key % tableSize;

// Second hash function

int hash2 = 7 - (key % 7); // Make sure it's relatively prime to tableSize

return (hash1 + i \* hash2) % tableSize;

}

// Insert a key into the hash table

private static void insert(int key) {

int i = 0;

int hashIndex;

do {

hashIndex = doubleHash(key, i++);

} while (hashTable[hashIndex] != 0 && i < tableSize);

if (i < tableSize) {

hashTable[hashIndex] = key;

} else {

System.out.println("Table is full. Unable to insert key: " + key);

}

}

// Search for a key in the hash table

private static boolean search(int key) {

int i = 0;

int hashIndex;

do {

hashIndex = doubleHash(key, i++);

if (hashTable[hashIndex] == key) {

return true;

}

} while (hashTable[hashIndex] != 0 && i < tableSize);

return false;

}

}

Test Case 1:

Input:

5

12 25 37 48 60

3

25 37 48

Output:

Found

Found

Found

Test Case 2:

Input:

5

12 25 37 48 60

3

20 50 60

Output:

Not Found

Not Found

Found

Test Case 3:

Input:

7

10 20 30 40 50 60 70

4

10 20 40 70

Output:

Found

Found

Found

Found

Test Case 4:

Input:

4

15 30 45 60

3

15 45 75

Output:

Found

Found

Not Found

**6. You are implementing a hash table using double hashing. The hash table has a size of N. Implement a Java program to handle the following:**

1.Input N, the size of the hash table.

2.Input M keys to insert into the hash table.

3.Input Q, the number of queries to perform.

4.For each query, input a key to search in the hash table.

5.Output "1" if the key exists in the hash table, otherwise output "0".

Your program should handle collisions using double hashing.

Input Format:

The first line contains an integer N representing the size of the hash table.

The second line contains M space-separated integers representing the keys to insert into the hash table.

The third line contains an integer Q representing the number of queries.

The next Q lines contain a single integer each, representing a key to search.

Output Format:

For each query, output either "1" if the key exists in the hash table, or "0" otherwise.

import java.util.Scanner;

public class DoubleHashingDemo1 {

private static final int TABLE\_SIZE = 1000;

private static int[] hashTable = new int[TABLE\_SIZE];

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input: Size of the hash table

int tableSize = scanner.nextInt();

// Input: Insert elements into the hash table

for (int i = 0; i < tableSize; i++) {

int key = scanner.nextInt();

int index = doubleHash(key, TABLE\_SIZE);

hashTable[index] = key;

}

// Input: Number of queries

int queries = scanner.nextInt();

// Input: Search for keys in the hash table

for (int i = 0; i < queries; i++) {

int key = scanner.nextInt();

int index = doubleHash(key, TABLE\_SIZE);

System.out.println(hashTable[index] == key ? "1" : "0");

}

}

// Double hashing function to get the hash index

private static int doubleHash(int key, int tableSize) {

int hash1 = key % tableSize;

int hash2 = 7 - (key % 7); // Ensure it's relatively prime to tableSize

return (hash1 + hash2) % tableSize;

}

}

Test Case 1:

Input:

5

12 25 37 48 60

3

25 37 48

Output:

1

1

1

Test Case 2:

Input:

3

10 20 30

2

25 35

Output:

0

0

Test Case 3:

Input:

4

15 30 45 60

1

45

Output:

1

Test Case 4:

Input:

2

100 200

2

100 200

Output:

1

1

**HASH FUNCTION**

**7.You are developing a program that uses a simple hash function to map an integer key to an index in an array. The array is populated with user-input elements. Afterward, the program prompts the user to enter an integer key. If the entered key is present in the array, the program applies the hash function and prints the resulting index. Otherwise, it notifies the user to enter a key that is present in the array.**

**Input Format:**

The user is initially prompted to enter the size of the array (an integer).

The program then expects the user to input the array elements (integers) one by one.

Following that, the user is asked to enter an integer key.

**Output Format:**

If the entered key is present in the array, the program prints: "The key [key] is hashed to index [index]."

If the entered key is not in the array, the program prints: "The entered key is not in the array. Please enter a key present in the array."

CODE

import java.util.Scanner;

public class HashFunction1 {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

//System.out.print("Enter the size of the array: ");

int arraySize = scanner.nextInt();

// Create an array of the given size

int[] arr1 = new int[arraySize];

// Populate the array with user input

// System.out.println("Enter " + arraySize + " elements for the array:");

for (int i = 0; i < arraySize; i++) {

//System.out.print("Enter element at index " + i + ": ");

arr1[i] = scanner.nextInt();

}

int key;

while (true) {

// System.out.print("Enter an integer key that is present in the array: ");

key = scanner.nextInt();

// Check if the key is present in the array

boolean keyInArray = false;

for (int i = 0; i < arraySize; i++) {

if (arr1[i] == key) {

keyInArray = true;

break;

}

}

if (keyInArray) {

break;

} else {

System.out.println("The entered key is not in the array. Please enter a key present in the array.");

}

}

// Use the simple hash function to find the remainder when dividing the key by the array size

int index = hashFunction(key, arraySize);

System.out.println("The key " + key + " is hashed to index " + index);

scanner.close();

}

private static int hashFunction(int key, int arraySize) {

// Simple hash function: take the remainder when dividing the key by the array size

return key % arraySize;

}

\*\*Test Case 1:\*\*

Input:

4

15

27

42

10

42

Output:

The key 42 is hashed to index 2

\*\*Test Case 2:\*\*

Input:

3

8

16

24

10

Output:

The key 10 is hashed to index 1

\*\*Test Case 3:\*\*

Input:

2

5

10

7

Output:

The key 7 is hashed to index 1

\*\*Test Case 4:\*\*

Input:

6

30

45

60

75

90

105

Output:

The entered key is not in the array. Please enter a key present in the array.

\*\*Test Case 5:\*\*

Input:

5

2

4

8

16

32

64

Output:

The key 64 is hashed to index 4

\*\*Test Case 6:\*\*

Input:

1

7

7

Output:

The key 7 is hashed to index 0  
  
  
**HASHTABLE**

**8.** **Imagine you're developing a basic spell-checking feature for a text editor. You decide to use a hash table to store a set of correctly spelled words. The user can input words, and the program will check if they're present in the hash table. If not, they're considered potentially misspelled.  
  
Input Format:**

Hash table size: An integer representing the desired size of the hash table.

Words: A sequence of words, one per line, for the program to process. The user can type "exit" to stop entering words.

**Output Format:**

Hash table state: After each word is inserted, display the current state of the hash table, indicating the index and the word (or "empty" if the slot is empty)  
  
ANSWER:

import java.util.Scanner;

class HashTable {

private String[] table;

private int size;

public HashTable(int size) {

this.size = size;

this.table = new String[size];

}

public void insert(String word) {

int index = word.length() % size;

int i = 1;

while (table[(index + i) % size] != null) i++;

table[(index + i) % size] = word;

}

public void printTable() {

System.out.println("Hash Table:");

for (int i = 0; i < size; i++)

System.out.println(i + ": " + (table[i] == null ? "empty" : table[i]));

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// System.out.print("Enter hash table size: ");

HashTable spellChecker = new HashTable(scanner.nextInt());

// System.out.println("Enter words (type 'exit' to stop):");

while (true) {

String word = scanner.next();

if (word.equals("exit")) break;

spellChecker.insert(word);

spellChecker.printTable();

}

scanner.close();

}

}

\*\*Test Case 1:\*\*

Input:

3

apple

banana

exit

\*\*Expected Output:\*\*

Hash Table:

0: empty

1: empty

2: apple

Hash Table:

0: empty

1: banana

2: apple

\*\*Test Case 2:\*\*

Input:

5

apple

banana

orange

exit

\*\*Expected Output:\*\*

Hash Table:

0: empty

1: empty

2: empty

3: apple

4: empty

Hash Table:

0: banana

1: empty

2: empty

3: apple

4: empty

Hash Table:

0: banana

1: empty

2: empty

3: apple

4: orange

\*\*Test Case 3:\*\*

Input:

4

apple

banana

orange

exit

```

\*\*Expected Output:\*\*

```

Hash Table:

0: empty

1: empty

2: apple

3: empty

Hash Table:

0: banana

1: empty

2: apple

3: empty

Hash Table:

0: banana

1: orange

2: apple

3: empty

```

\*\*Test Case 4:\*\*

```

Input:

2

apple

banana

exit

```

\*\*Expected Output:\*\*

```

Hash Table:

0: empty

1: apple

Hash Table:

0: banana

1: apple

```

\*\*Test Case 5:\*\*

```

Input:

6

apple

banana

orange

grape

exit

```

\*\*Expected Output:\*\*

```

Hash Table:

0: empty

1: empty

2: empty

3: apple

4: empty

5: empty

Hash Table:

0: banana

1: empty

2: empty

3: apple

4: empty

5: empty

Hash Table:

0: banana

1: empty

2: grape

3: apple

4: empty

5: empty

Hash Table:

0: banana

1: empty

2: grape

3: apple

4: orange

5: empty

\*\*Test Case 6:\*

Input:

1

apple

banana

exit

\*\*Expected Output:\*\*

Hash Table:

0: banana

**9. Imagine you're developing a simple inventory management system for a small bookstore. You decide to use a Hashtable to store book information, where the keys are book ISBNs (integers) and the values are book titles (strings).**

**Input Format:**

Key-value pairs: A single line of text containing key-value pairs separated by spaces, where each key is an integer representing a book ISBN and each value is a string representing the corresponding book title.

**Output Format:**

Final Hashtable: The contents of the Hashtable after all key-value pairs have been added, displayed in a clear format that shows the ISBN-title associations.  
  
ANSWER:

import java.util.\*;

class Main{

public static void main(String args[]) {

Scanner scanner = new Scanner(System.in);

// Create a Hashtable to store book titles corresponding to ISBNs

Hashtable<Integer, String> inventory = new Hashtable<>();

// Input the values as key-value pairs

// System.out.println("Enter book ISBN-title pairs for the Hashtable (e.g., 123456789 Book1 987654321 Book2):");

String input = scanner.nextLine();

String[] keyValuePairs = input.split("\\s+");

for (int i = 0; i < keyValuePairs.length; i += 2) {

int isbn = Integer.parseInt(keyValuePairs[i]);

String title = keyValuePairs[i + 1];

inventory.put(isbn, title);

}

// Printing the final Hashtable

System.out.println("Final Inventory:");

for (Map.Entry<Integer, String> entry : inventory.entrySet()) {

System.out.println("ISBN: " + entry.getKey() + ", Title: " + entry.getValue());

}

scanner.close();

}

}

\*\*Test Case 1:\*\*

123456789 Book1 987654321 Book2 111111111 Book3

\*\*Output:\*\*

Final Inventory:

ISBN: 123456789, Title: Book1

ISBN: 987654321, Title: Book2

ISBN: 111111111, Title: Book3

```

\*\*Test Case 2:\*\*

555555555 JavaProgramming 888888888 DataStructures 999999999 Algorithms

```

\*\*Output:\*\*

```

Final Inventory:

ISBN: 555555555, Title: JavaProgramming

ISBN: 888888888, Title: DataStructures

ISBN: 999999999, Title: Algorithms

```

\*\*Test Case 3:\*\*

111111111 BookA 333333333 BookC 444444444 BookD

```

\*\*Output:\*\*

```

Final Inventory:

ISBN: 111111111, Title: BookA

ISBN: 333333333, Title: BookC

ISBN: 444444444, Title: BookD

```

\*\*Test Case 4:\*\*

123 Book1 456 Book2 789 Book3

```

\*\*Output:\*\*

```

Final Inventory:

ISBN: 123, Title: Book1

ISBN: 456, Title: Book2

ISBN: 789, Title: Book3

```

\*\*Test Case 5:\*\*

111 BookA 333 BookC 222 BookD

```

\*\*Output:\*\*

```

Final Inventory:

ISBN: 111, Title: BookA

ISBN: 333, Title: BookC

\*\*Test Case 6:\*\*

555 JavaProgramming 555 JavaProgramming 555 JavaProgramming

\*\*Output:\*\*

Final Inventory:

ISBN: 555, Title: JavaProgramming

**HASH FUNCTION**

**10. You are tasked with developing a student information system for a school. The system should efficiently store and retrieve student records based on their unique student IDs. Each student has a unique ID assigned by the system. for this question need to use hash function no collection let me update?**

**Input Format:**

The first line contains an integer n, the number of student records to be added.

Each of the next n lines contains three space-separated values: the student ID (an integer), the student's name (a string), and the student's age (an integer).

The next line contains an integer q, the number of queries.

Each of the next q lines contains a single integer, representing a student ID to query.

Output Format:

For each query, if the student ID exists in the system, print the student's information in the following format:

less

Copy code

Student ID: [studentID]

Name: [studentName]

Age: [studentAge]

CODE

import java.util.Scanner;

public class StudentInformationSystem {

private final int TABLE\_SIZE = 1000;

private StudentRecord[] studentRecords = new StudentRecord[TABLE\_SIZE];

public static void main(String[] args) {

StudentInformationSystem system = new StudentInformationSystem();

system.run();

}

public void run() {

Scanner scanner = new Scanner(System.in);

// Input: Number of students and their information

int n = scanner.nextInt();

for (int i = 0; i < n; i++) {

int studentID = scanner.nextInt();

String studentName = scanner.next();

int studentAge = scanner.nextInt();

insertStudentRecord(studentID, studentName, studentAge);

}

// Input: Query for student information based on ID

int q = scanner.nextInt();

for (int i = 0; i < q; i++) {

int queryID = scanner.nextInt();

getStudentRecord(queryID);

}

scanner.close();

}

private int hashFunction(int studentID) {

return studentID % TABLE\_SIZE;

}

private void insertStudentRecord(int studentID, String name, int age) {

int index = hashFunction(studentID);

studentRecords[index] = new StudentRecord(studentID, name, age);

}

private void getStudentRecord(int studentID) {

int index = hashFunction(studentID);

if (studentRecords[index] != null && studentRecords[index].studentID == studentID) {

System.out.println("Student ID: " + studentRecords[index].studentID);

System.out.println("Name: " + studentRecords[index].name);

System.out.println("Age: " + studentRecords[index].age);

} else {

System.out.println("Student with ID " + studentID + " not found.");

}

}

class StudentRecord {

int studentID;

String name;

int age;

public StudentRecord(int studentID, String name, int age) {

this.studentID = studentID;

this.name = name;

this.age = age;

}

}

}

Test Case 1: Basic Input

Input:

3

101 John 20

102 Alice 22

103 Bob 21

2

101

104

Output:

Student ID: 101

Name: John

Age: 20

Student with ID 104 not found.

Test Case 2: No Students

Input:

0

2

101

102

Output:

Student with ID 101 not found.

Student with ID 102 not found.

Test Case 3: Duplicate Student ID

Input:

2

101 John 20

101 Alice 22

1

101

Output:

Student ID: 101

Name: John

Age: 20

Test Case 4: Large Dataset

Input:

5

1001 John 20

1002 Alice 22

1003 Bob 21

1004 Carol 23

1005 Dave 24

3

1001

1005

1006

Output:

Student ID: 1001

Name: John

Age: 20

Student ID: 1005

Name: Dave

Age: 24

Student with ID 1006 not found.

**COLLISION RESOLUTION TECHNIQUES**

**11.You are developing a student registration system for a university. The system assigns each student a unique registration ID based on their personal information. The registration ID is an integer value generated using a hash function. However, due to the large number of students, collisions occur frequently.**

**System Requirements:**

Each student's personal information includes their name, date of birth, and program of study.

The system should efficiently store and retrieve student records based on their registration ID.

In case of collisions, implement separate chaining to handle multiple student records stored at the same index in the hash table.

Tasks:

Implement a hash function that generates registration IDs based on student information.

Develop a hash table data structure to store student records, handling collisions using separate chaining.

Allow users to add new student records to the system and retrieve student information by providing the registration ID.

Ensure that the system can handle a large number of student records efficiently, even in the presence of collisions.

Input Format:

User inputs include student personal information (name, date of birth, program of study) for adding new student records.

Queries for student information involve providing the registration ID.

Output Format:

Upon adding a new student record, the system should confirm successful insertion.

When querying student information, the system should display the student's personal details (name, date of birth, program of study) based on the provided registration ID.

import java.util.\*;

public class StudentRegistrationSystem {

// Define the StudentRecord class to store student information

static class StudentRecord {

String name;

String dob;

String program;

public StudentRecord(String name, String dob, String program) {

this.name = name;

this.dob = dob;

this.program = program;

}

@Override

public String toString() {

return name + ", " + dob + ", " + program;

}

}

// Define the hash table to store student records (using separate chaining for collision resolution)

private static final int TABLE\_SIZE = 1000;

private static LinkedList<StudentRecord>[] hashTable = new LinkedList[TABLE\_SIZE];

// Hash function to generate registration IDs based on student information

private static int hashFunction(int studentID) {

return studentID % TABLE\_SIZE;

}

// Method to add a new student record to the system

private static void addStudentRecord(int studentID, String name, String dob, String program) {

int index = hashFunction(studentID);

if (hashTable[index] == null) {

hashTable[index] = new LinkedList<>();

}

hashTable[index].add(new StudentRecord(name, dob, program));

System.out.println("Student Record Added: " + name + ", " + dob + ", " + program);

}

// Method to retrieve student information based on the registration ID

private static void queryStudentInformation(int studentID) {

int index = hashFunction(studentID);

if (hashTable[index] != null) {

for (StudentRecord record : hashTable[index]) {

System.out.println("Student Information (ID: " + studentID + "): " + record);

return;

}

}

System.out.println("Student Information (ID: " + studentID + "): No record found.");

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Process user input

while (true) {

System.out.println("Enter command (Add Student Record / Query Student Information / Exit):");

String command = scanner.nextLine();

if (command.equalsIgnoreCase("Exit")) {

break;

} else if (command.startsWith("Add Student Record")) {

// Parse input and add student record

String[] parts = command.split(":")[1].trim().split(",");

String name = parts[0].trim();

String dob = parts[1].trim();

String program = parts[2].trim();

int studentID = name.hashCode(); // Using name's hash code as a simplistic student ID

addStudentRecord(studentID, name, dob, program);

} else if (command.startsWith("Query Student Information")) {

// Parse input and query student information

int studentID = Integer.parseInt(command.split(":")[1].trim());

queryStudentInformation(studentID);

} else {

System.out.println("Invalid command.");

}

}

scanner.close();

}

}

**12.** **You are developing a system to manage employee records for a medium-sized company. The system stores employee information such as name, department, and employee ID. Each employee is assigned a unique ID, but due to the large number of employees, collisions may occur when hashing employee IDs to the hash table.**

**Your task is to implement collision resolution techniques to handle these collisions effectively. You decide to use chaining as the collision resolution method.**

Input Format:

The user inputs employee information, one employee per line.

Each line contains the employee ID (an integer), name (a string), and department (a string), separated by commas.

The user can enter "exit" to stop adding employees.

Output Format:

After each employee is added, the system displays the current state of the hash table, indicating the index and the employee information stored at that index.

If collisions occur, the system indicates which employees are stored at each index, separated by commas.

import java.util.\*;

public class EmployeeManagementSystem {

private static final int TABLE\_SIZE = 10; // Adjust the table size as needed

private static LinkedList<Employee>[] hashTable = new LinkedList[TABLE\_SIZE];

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Initialize the hash table

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTable[i] = new LinkedList<>();

}

// Input: Employee information

System.out.println("Enter employee information (ID, Name, Department), or type 'exit' to stop:");

while (true) {

String input = scanner.nextLine().trim();

if (input.equals("exit")) break;

String[] parts = input.split(",");

int employeeID = Integer.parseInt(parts[0]);

String name = parts[1];

String department = parts[2];

addEmployee(employeeID, name, department);

displayHashTable();

}

scanner.close();

}

// Add employee to the hash table using chaining

private static void addEmployee(int employeeID, String name, String department) {

int index = hash(employeeID);

if (hashTable[index] == null) {

hashTable[index] = new LinkedList<>();

}

hashTable[index].add(new Employee(employeeID, name, department));

}

// Display the current state of the hash table

private static void displayHashTable() {

for (int i = 0; i < TABLE\_SIZE; i++) {

System.out.print("Index " + i + ": ");

if (hashTable[i] == null || hashTable[i].isEmpty()) {

System.out.println("empty");

} else {

for (Employee employee : hashTable[i]) {

System.out.print(employee + " ");

}

System.out.println();

}

}

}

// Hash function to determine index

private static int hash(int key) {

return key % TABLE\_SIZE;

}

// Employee class to represent an employee record

static class Employee {

int employeeID;

String name;

String department;

public Employee(int employeeID, String name, String department) {

this.employeeID = employeeID;

this.name = name;

this.department = department;

}

@Override

public String toString() {

return employeeID + " " + name + " " + department;

}

}

}

Test Cases:

Test Case 1:

Input:

101,John Doe,HR

102,Alice Smith,Engineering

103,Bob Johnson,Marketing

exit

Output:

Index 0: empty

Index 1: 101 John Doe HR

Index 2: 102 Alice Smith Engineering

Index 3: 103 Bob Johnson Marketing

Index 4: empty

Index 5: empty

Index 6: empty

Index 7: empty

Index 8: empty

Index 9: empty

Test Case 2:

Input:

101,John Doe,HR

102,Alice Smith,Engineering

101,Daniel Brown,Finance

exit

Output:

Index 0: empty

Index 1: 101 John Doe HR 101 Daniel Brown Finance

Index 2: 102 Alice Smith Engineering

Index 3: empty

Index 4: empty

Index 5: empty

Index 6: empty

Index 7: empty

Index 8: empty

Index 9: empty

Test Case 3:

Input:

101,John Doe,HR

102,Alice Smith,Engineering

105,David Lee,HR

exit

Output:

Index 0: empty

Index 1: 101 John Doe HR

Index 2: 102 Alice Smith Engineering

Index 3: empty

Index 4: empty

Index 5: 105 David Lee HR

Index 6: empty

Index 7: empty

Index 8: empty

Index 9: empty

Test Case 4:

Input:

101,John Doe,HR

exit

Output:

Index 0: empty

Index 1: 101 John Doe HR

Index 2: empty

Index 3: empty

Index 4: empty

Index 5: empty

Index 6: empty

Index 7: empty

Index 8: empty

Index 9: empty