Write a C program to demonstrate hashing. Also calculate the space and time complexity

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

Typedef struct Node {
   Int key;
   Int value;
   Struct Node* next;
} Node;

Node* createNode(int key, int value) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->key = key;
   newNode->value = value;
   newNode->next = NULL;
   return newNode;
}

Void insert(Node** hashTable, int key, int value) {
   Int index = key % TABLE_SIZE;
   If (hashTable[index] == NULL) {
      hashTable[index] = createNode(key, value);
   } else {
      Node* newNode = createNode(key, value);
      newNode->next = hashTable[index];
      hashTable[index] = newNode;
   }
}

Void display(Node** hashTable) {
    For (int I = 0; I < TABLE_SIZE; i++) {
        Printf("[%d] ->", i);
        Node* node = hashTable[i];
        While (node != NULL) {
            Printf(" %d:%d ->", node->key, node->value);
            Node = node->next;
        }
        Printf(" NULL\n");
    }
}

Int main() {
    Node* hashTable[TABLE_SIZE] = { NULL };

    Insert(hashTable, 10, 42);
    Insert(hashTable, 5, 23);
    Insert(hashTable, 20, 19);
    Insert(hashTable, 15, 33);
    Insert(hashTable, 25, 11);

    Display(hashTable);

    Return 0;
}
```

---

Write a C program to implement hash tables.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

Typedef struct {
    Int key;
    Int value;
} HashNode;

Typedef struct {
    HashNode* array[SIZE];
} HashTable;

HashTable* createHashTable() {
    HashTable* hashtable = (HashTable*)malloc(sizeof(HashTable));
    For (int I = 0; I < SIZE; i++) {
        Hashtable->array[i] = NULL;
    }
    Return hashtable;
}

Int hashFunction(int key) {
    Return key % SIZE;
}

Void insert(HashTable* hashtable, int key, int value) {
    Int index = hashFunction(key);
    HashNode* newNode = (HashNode*)malloc(sizeof(HashNode));
    newNode->key = key;
    newNode->value = value;

    if (hashtable->array[index] == NULL) {
        hashtable->array[index] = newNode;
    } else {
        Printf("Collision occurred at index %d. Resolving…\n", index);
        HashNode* currentNode = hashtable->array[index];
        While (currentNode->next != NULL) {
            currentNode = currentNode->next;
        }
        currentNode->next = newNode;
    }
    Printf("Key-value pair (%d, %d) inserted successfully.\n", key, value);
}

Int search(HashTable* hashtable, int key) {
    Int index = hashFunction(key);
    HashNode* currentNode = hashtable->array[index];
    While (currentNode != NULL) {
        If (currentNode->key == key) {
            Return currentNode->value;
        }
        currentNode = currentNode->next;
    }
    Return -1;
}

Void display(HashTable* hashtable) {
    For (int I = 0; I < SIZE; i++) {
        HashNode* currentNode = hashtable->array[i];
        Printf("Index %d: ", i);
        While (currentNode != NULL) {
            Printf("(%d, %d) ", currentNode->key, currentNode->value);.
            currentNode = currentNode->next;
        }
        Printf("\n");
    }
}

Void deleteHashTable(HashTable* hashtable) {
    For (int I = 0; I < SIZE; i++) {
        HashNode* currentNode = hashtable->array[i];
        While (currentNode != NULL) {
            HashNode* temp = currentNode;
            currentNode = currentNode->next;
            free(temp);
        }
    }
    Free(hashtable);
    Printf("Hash table deleted successfully.\n");
}

Int main() {
    HashTable* hashtable = createHashTable();

    Insert(hashtable, 1, 10);
    Insert(hashtable, 2, 20);
    Insert(hashtable, 11, 30);
    Insert(hashtable, 21, 40);
    Insert(hashtable, 3, 50);
    Insert(hashtable, 4, 60);
    Insert(hashtable, 12, 70);
    Insert(hashtable, 22, 80);
    Insert(hashtable, 5, 90);
    Insert(hashtable, 6, 100);

    Display(hashtable);

    Printf(“Value at key 11: %d\n”, search(hashtable, 11));
    Printf(“Value at key 6: %d\n”, searc#include <stdio.h>

#define TABLE_SIZE 10

// Division hash function
Int hash(int key) {
    Return key % TABLE_SIZE;
}

Int main() {
    Int key;
    Printf(“Enter a key: “);
    Scanf(“%d”, &key);

    Int hashValue = hash(key);
    Printf(“Hash value: %d\n”, hashValue);
    Return 0;
}
h(hashtable, 6));

    deleteHashTable(hashtable);

    return 0;
}
```

Write a C program to generate a hash value using division hash function

```
#include <stdio.h>

#define TABLE_SIZE 10

// Division hash function
int hash(int key) {
    return key % TABLE_SIZE;
}

int main() {
    int key;
    printf("Enter a key: ");
    scanf("%d", &key);

    int hashValue = hash(key);
    printf("Hash value: %d\n", hashValue);

    return 0;
}
```

Write a C program to generate a hash value using mid square hash function.

```
#include <stdio.h>
#include <math.h>

#define TABLE_SIZE 10

// Mid-square hash function
Int hash(int key) {
    Int square = key * key;
    Int numDigits = floor(log10(key)) + 1;

    Int start = numDigits / 2 – 1;
    Int end = start + 2;

    Int midSquare = 0;
    For (int I = start; I <= end; i++) {
        Int digit = (square / (int)pow(10, i)) % 10;
        midSquare = midSquare * 10 + digit;
    }

    Return midSquare % TABLE_SIZE;
}

Int main() {
    Int key;
    Printf(“Enter a key: “);
    Scanf(“%d”, &key);

    Int hashValue = hash(key);
    Printf(“Hash value: %d\n”, hashValue);

    Return 0;
}
```

---

Write a C program to generate a hash value using folding hash function

```
#include <stdio.h>

#define TABLE_SIZE 10

// Folding hash function
int hash(int key) {
    int sum = 0;

    while (key > 0) {
        sum += key % 1000;
        key /= 1000;
    }

    return sum % TABLE_SIZE;
}

int main() {
    int key;
    printf("Enter a key: ");
    scanf("%d", &key);

    int hashValue = hash(key);
    printf("Hash value: %d\n", hashValue);
```

```
   return 0;
}
```

---

Write a C program to generate a hash value using folding hash function

```
#include <stdio.h>


#define TABLE_SIZE 10


// Folding hash function
Int hash(int key) {
   Int sum = 0;


   While (key > 0) {
      Sum += key % 1000;
      Key /= 1000;
   }


   Return sum % TABLE_SIZE;
}


Int main() {
   Int key;
   Printf(“Enter a key: “);
   Scanf(“%d”, &key);


   Int hashValue = hash(key);
   Printf(“Hash value: %d\n”, hashValue);


   Return 0;
}
```

Write a C program to generate a hash value using multiplication hash function

```
#include <stdio.h>

#define TABLE_SIZE 10

// Multiplication hash function
Int hash(int key) {
   // Choose a constant value for the multiplication
   Float A = 0.6180339887;

   // Perform the multiplication
   Float product = key * A;

   // Extract the fractional part of the product
   Float fraction = product – (int)product;

   // Multiply the fraction by the table size
   Int hashValue = (int)(TABLE_SIZE * fraction);

   Return hashValue;
}

Int main() {
   Int key;
   Printf("Enter a key: ");
   Scanf("%d", &key);
   Int hashValue = hash(key);
   Printf("Hash value: %d\n", hashValue);

   Return 0;
}
```

Write a C program to generate a hash value using multiplication hash function

```
#include <stdio.h>

#define TABLE_SIZE 10

// Multiplication hash function
Int hash(int key) {
   // Choose a constant value for the multiplication
   Float A = 0.6180339887;

   // Perform the multiplication
   Float product = key * A;

   // Extract the fractional part of the product
   Float fraction = product – (int)product;

   // Multiply the fraction by the table size
   Int hashValue = (int)(TABLE_SIZE * fraction);

   Return hashValue;
}

Int main() {
   Int key;
```

```
    Printf(“Enter a key: “);
    Scanf(“%d”, &key);

    Int hashValue = hash(key);
    Printf(“Hash value: %d\n”, hashValue);

    Return 0;
}
```

---

Write a C program to implement hashing using linear probing as the collision Resolution strategy.

```
#include <stdio.h>

#define TABLE_SIZE 10

// Hash table
Int hashTable[TABLE_SIZE] = {0};

// Hash function
Int hash(int key) {
    Return key % TABLE_SIZE;
}

// Insert key into the hash table
Void insert(int key) {
    Int hashValue = hash(key);

    // Linear probing
    While (hashTable[hashValue] != 0) {
        hashValue = (hashValue + 1) % TABLE_SIZE;
    }

    hashTable[hashValue] = key;
    printf(“Inserted %d at index %d\n”, key, hashValue);
}

// Search for a key in the hash table
Int search(int key) {
    Int hashValue = hash(key);

    // Linear probing
    While (hashTable[hashValue] != 0) {
        If (hashTable[hashValue] == key) {
            Return hashValue;
        }
        hashValue = (hashValue + 1) % TABLE_SIZE;
    }

    Return -1; // Key not found
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);
    Int searchKey;
    Printf("Enter a key to search: ");
    Scanf("%d", &searchKey);

    Int index = search(searchKey);

    If (index != -1) {
        Printf("Key found at index %d\n", index);
    } else {
        Printf("Key not found\n");
    }

    Return 0;
} Resolution strategy.
#include <stdio.h>

#define TABLE_SIZE 10

// Hash table
Int hashTable[TABLE_SIZE] = {0};

// Hash function
Int hash(int key) {
    Return key % TABLE_SIZE;
}

// Insert key into the hash table
Void insert(int key) {
    Int hashValue = hash(key);
    // Linear probing
    While (hashTable[hashValue] != 0) {
        hashValue = (hashValue + 1) % TABLE_SIZE;
    }

    hashTable[hashValue] = key;
    printf(“Inserted %d at index %d\n”, key, hashValue);
}

// Search for a key in the hash table
Int search(int key) {
    Int hashValue = hash(key);

    // Linear probing
    While (hashTable[hashValue] != 0) {
        If (hashTable[hashValue] == key) {
            Return hashValue;
        }
        hashValue = (hashValue + 1) % TABLE_SIZE;
    }

    Return -1; // Key not found
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);

    Int searchKey;
    Printf("Enter a key to search: ");
    Scanf("%d", &searchKey);

    Int index = search(searchKey);

    If (index != -1) {
        Printf("Key found at index %d\n", index);
    } else {
        Printf("Key not found\n");
    }

    Return 0;
}
```

---

Write a C program to implement hashing using linear probing as the collision

Write a C program to implement hashing using chaining with replacement as the

Collision resolution strategy

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

// Linked list node
Struct Node {
    Int data;
    Struct Node* next;
};

// Hash table
Struct Node* hashTable[TABLE_SIZE] = {NULL};

// Hash function
Int hash(int key) {
    Return key % TABLE_SIZE;
}

// Insert key into the hash table
Void insert(int key) {
    Int hashValue = hash(key);

    // Create a new node
    Struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = key;
    newNode->next = NULL;

    // If the hash slot is empty, insert the node directly
    If (hashTable[hashValue] == NULL) {
        hashTable[hashValue] = newNode;
    } else {
        // Collision occurred, find the last node in the chain
        Struct Node* current = hashTable[hashValue];
        While (current->next != NULL) {
            Current = current->next;
        }

        // Replace the last node with the new node
        Current->next = newNode;
    }

    Printf("Inserted %d\n", key);
}

// Search for a key in the hash table
Int search(int key) {
    Int hashValue = hash(key);

    // Search the chain for the key
    Struct Node* current = hashTable[hashValue];
    While (current != NULL) {
        If (current->data == key) {
            Return hashValue; // Key found
        }
        Current = current->next;
    }

    Return -1; // Key not found
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);

    Int searchKey;
    Printf(“Enter a key to search: “);
    Scanf(“%d”, &searchKey);

    Int index = search(searchKey);

    If (index != -1) {
        Printf(“Key found at index %d\n”, index);
    } else {
        Printf(“Key not found\n”);
    }

    Return 0;
}
```

---

Write a C program to implement hashing using chaining without replacement as

The collision resolution strategy

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

// Linked list node
Struct Node {
    Int data;
    Struct Node* next;
```

```
};

// Hash table
Struct Node* hashTable[TABLE_SIZE] = {NULL};

// Hash function
Int hash(int key) {
    Return key % TABLE_SIZE;
}

// Insert key into the hash table
Void insert(int key) {
    Int hashValue = hash(key);

    // Create a new node
    Struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = key;
    newNode->next = NULL;

    // If the hash slot is empty, insert the node directly
    If (hashTable[hashValue] == NULL) {
        hashTable[hashValue] = newNode;
    } else {
        // Collision occurred, find the last node in the chain
        Struct Node* current = hashTable[hashValue];
        While (current->next != NULL) {
            Current = current->next;
        }
        // Insert the node at the end of the chain
        Current->next = newNode;
    }

    Printf("Inserted %d\n", key);
}

// Search for a key in the hash table
Int search(int key) {
    Int hashValue = hash(key);

    // Search the chain for the key
    Struct Node* current = hashTable[hashValue];
    While (current != NULL) {
        If (current->data == key) {
            Return hashValue; // Key found
        }
        Current = current->next;
    }

    Return -1; // Key not found
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);
    Int searchKey;
    Printf(“Enter a key to search: “);
    Scanf(“%d”, &searchKey);

    Int index = search(searchKey);

    If (index != -1) {
        Printf(“Key found at index %d\n”, index);
    } else {
        Printf(“Key not found\n”);
    }

    Return 0;
}
```

---

Write a C program to implement closed hashing.

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

// Hash table
Int hashTable[TABLE_SIZE] = {0};

// Hash function
Int hash(int key) {
    Return key % TABLE_SIZE;
}
```

```
// Insert key into the hash table
Void insert(int key) {
    Int hashValue = hash(key);
    Int initialHash = hashValue;

    // Linear probing
    While (hashTable[hashValue] != 0) {
        hashValue = (hashValue + 1) % TABLE_SIZE;
        // Check if we have traversed the entire table
        If (hashValue == initialHash) {
            Printf("Hash table is full. Unable to insert %d\n", key);
            Return;
        }
    }

    hashTable[hashValue] = key;
    printf("Inserted %d at index %d\n", key, hashValue);
}

// Search for a key in the hash table
Int search(int key) {
    Int hashValue = hash(key);
    Int initialHash = hashValue;

    // Linear probing
    While (hashTable[hashValue] != 0) {
        If (hashTable[hashValue] == key) {
            Return hashValue; // Key found
        }
        hashValue = (hashValue + 1) % TABLE_SIZE;
        // Check if we have traversed the entire table
        If (hashValue == initialHash) {
            Break;
        }
    }

    Return -1; // Key not found
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);

    Int searchKey;
    Printf(“Enter a key to search: “);
    Scanf(“%d”, &searchKey);

    Int index = search(searchKey);

    If (index != -1) {
        Printf(“Key found at index %d\n”, index);
    } else {
        Printf(“Key not found\n”);
    }
    Return 0;
}
```

---

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

// Hash table
Int hashTable[TABLE_SIZE] = {0};

// Hash function
Int hash(int key) {
    Return key % TABLE_SIZE;
}

// Insert key into the hash table
Void insert(int key) {
    Int hashValue = hash(key);

    // Linear probing
    While (hashTable[hashValue] != 0) {
        hashValue = (hashValue + 1) % TABLE_SIZE;
        // Check if we have traversed the entire table
        If (hashValue == hash(key)) {
            Printf("Hash table is full. Unable to insert %d\n", key);
            Return;
        }
    }

    hashTable[hashValue] = key;
    printf(“Inserted %d at index %d\n”, key, hashValue);
}

// Search for a key in the hash table
Int search(int key) {
    Int hashValue = hash(key);

    // Linear probing
    While (hashTable[hashValue] != 0) {
        If (hashTable[hashValue] == key) {
            Return hashValue; // Key found
        }
        hashValue = (hashValue + 1) % TABLE_SIZE;
        // Check if we have traversed the entire table
        If (hashValue == hash(key)) {
            Break;
        }
    }

    Return -1; // Key not found
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);

    Int searchKey;
    Printf(“Enter a key to search: “);
    Scanf(“%d”, &searchKey);

    Int index = search(searchKey);

    If (index != -1) {
        Printf(“Key found at index %d\n”, index);
    } else {
        Printf(“Key not found\n”);
    }

    Return 0;
}
```

---

2.Write a C Program to implement a Hash Table with Linear Probing.

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

// Hash table
Int hashTable[TABLE_SIZE] = {0};

// Hash function
Int hash(int key) {
   Return key % TABLE_SIZE;
}

// Insert key into the hash table
Void insert(int key) {
   Int hashValue = hash(key);

   // Linear probing
   While (hashTable[hashValue] != 0) {
      hashValue = (hashValue + 1) % TABLE_SIZE;
      // Check if we have traversed the entire table
      If (hashValue == hash(key)) {
         Printf("Hash table is full. Unable to insert %d\n", key);
         Return;
      }
   }

   hashTable[hashValue] = key;
   printf("Inserted %d at index %d\n", key, hashValue);
}

// Search for a key in the hash table
Int search(int key) {
   Int hashValue = hash(key);

   // Linear probing
   While (hashTable[hashValue] != 0) {
      If (hashTable[hashValue] == key) {
         Return hashValue; // Key found
```

```
        }
        hashValue = (hashValue + 1) % TABLE_SIZE;
        // Check if we have traversed the entire table
        If (hashValue == hash(key)) {
            Break;
        }
    }

    Return -1; // Key not found
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);

    Int searchKey;
    Printf("Enter a key to search: ");
    Scanf("%d", &searchKey);

    Int index = search(searchKey);

    If (index != -1) {
        Printf("Key found at index %d\n", index);
    } else {
        Printf("Key not found\n");
    }
```

```
   Return 0;
}
```

---

Write a program to insert a value in linear probing

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

// Hash table
Int hashTable[TABLE_SIZE] = {0};

// Hash function
Int hash(int key) {
   Return key % TABLE_SIZE;
}

// Insert key into the hash table
Void insert(int key) {
   Int hashValue = hash(key);

   // Linear probing
   While (hashTable[hashValue] != 0) {
      hashValue = (hashValue + 1) % TABLE_SIZE;
      // Check if we have traversed the entire table
      If (hashValue == hash(key)) {
         Printf("Hash table is full. Unable to insert %d\n", key);
         Return;
```

```
        }
    }

    hashTable[hashValue] = key;
    printf(“Inserted %d at index %d\n”, key, hashValue);
}

Int main() {
    Insert(12);
    Insert(25);
    Insert(38);
    Insert(14);
    Insert(19);

    Return 0;
}
```

4.Write a c program to search a value in linear probing

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

Typedef struct
{
    Int key;
    Int value;
    Int isActive;
} HashEntry;
```

```
HashEntry *hashTable[SIZE];

Int hashCode(int key)
{
   Return key % SIZE;
}

Void insert(int key, int value)
{
   Int hashIndex = hashCode(key);

   While (hashTable[hashIndex] != NULL && hashTable[hashIndex]->key != key)
   {
      hashIndex++;
      hashIndex %= SIZE;
   }

   HashEntry *entry = (HashEntry *)malloc(sizeof(HashEntry));
   Entry->key = key;
   Entry->value = value;
   Entry->isActive = 1;

   hashTable[hashIndex] = entry;
}

Int search(int key)
{
   Int hashIndex = hashCode(key);
```

```
    While (hashTable[hashIndex] != NULL)
    {
        If (hashTable[hashIndex]->key == key && hashTable[hashIndex]->isActive == 1)
        {
            Return hashTable[hashIndex]->value;
        }

        hashIndex++;
        hashIndex %= SIZE;
    }

    Return -1;
}

Int main()
{
    Int keys[] = {1, 2, 3, 4, 5};
    Int values[] = {10, 20, 30, 40, 50};
    Int size = sizeof(keys) / sizeof(keys[0]);

    For (int I = 0; I < SIZE; i++)
    {
        hashTable[i] = NULL;
    }

    For (int I = 0; I < size; i++)
    {
        Insert(keys[i], values[i]);
    }


    Int searchKey = 3;

    Int searchResult = search(searchKey);


    If (searchResult != -1)

    {

        Printf("Value found: %d\n", searchResult);

    }

    Else

    {

        Printf("Value not found.\n");

    }


    Return 0;

}
```

---

Write a c program,Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Insert (1, 5): Assign the pair {1, 5} at the index (1%20 =1) in the Hash Table.

```
#include <stdio.h>

#include <stdlib.h>


#define TABLE_SIZE 20


Typedef struct {

    Int key;

    Int value;

} Pair;
```

```
Void initializeHashTable(Pair *table, int size) {
   For (int I = 0; I < size; i++) {
      Table[i].key = -1; // -1 indicates an empty slot
      Table[i].value = -1;
   }
}

Void insertPair(Pair *table, int size, Pair pair) {
   Int index = pair.key % size; // Calculate the hash index

   // Linear probing to find an empty slot
   While (table[index].key != -1) {
      Index = (index + 1) % size;
   }

   // Insert the pair at the empty slot
   Table[index] = pair;
}

Void printHashTable(const Pair *table, int size) {
   Printf(“Hash Table:\n”);
   For (int I = 0; I < size; i++) {
      If (table[i].key != -1) {
         Printf(“[%d] -> {%d, %d}\n”, I, table[i].key, table[i].value);
      } else {
         Printf(“[%d] -> Empty\n”, i);
      }
   }
}

Int main() {
   Pair pairs[] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
   Int numPairs = sizeof(pairs) / sizeof(pairs[0]);

   Pair hashTable[TABLE_SIZE];
   initializeHashTable(hashTable, TABLE_SIZE);

   for (int I = 0; I < numPairs; i++) {
      insertPair(hashTable, TABLE_SIZE, pairs[i]);
   }

   printHashTable(hashTable, TABLE_SIZE);

   return 0;
}
```

---

Write a c program Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Insert(2, 15): Assign the pair {2, 15} at the index (2%20 =2) in the Hash Table.

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 20

Typedef struct {
   Int key;
   Int value;
```

```
} Pair;

Void initializeHashTable(Pair *table, int size) {
    For (int I = 0; I < size; i++) {
        Table[i].key = -1; // -1 indicates an empty slot
        Table[i].value = -1;
    }
}

Void insertPair(Pair *table, int size, Pair pair) {
    Int index = pair.key % size; // Calculate the hash index

    // Linear probing to find an empty slot
    While (table[index].key != -1) {
        Index = (index + 1) % size;
    }

    // Insert the pair at the empty slot
    Table[index] = pair;
}

Void printHashTable(const Pair *table, int size) {
    Printf(“Hash Table:\n”);
    For (int I = 0; I < size; i++) {
        If (table[i].key != -1) {
            Printf(“[%d] -> {%d, %d}\n”, I, table[i].key, table[i].value);
        } else {
            Printf(“[%d] -> Empty\n”, i);
        }
    }
}

Int main() {
    Pair pairs[] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
    Int numPairs = sizeof(pairs) / sizeof(pairs[0]);

    Pair hashTable[TABLE_SIZE];
    initializeHashTable(hashTable, TABLE_SIZE);

    for (int I = 0; I < numPairs; i++) {
        insertPair(hashTable, TABLE_SIZE, pairs[i]);
    }

    // Additional insert operation
    Pair additionalPair = {2, 15};
    insertPair(hashTable, TABLE_SIZE, additionalPair);

    printHashTable(hashTable, TABLE_SIZE);

    return 0;
}
```

---

Write a C program Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Insert(3, 20): Assign the pair {3, 20} at the index (3%20 =3) in the Hash Table

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define TABLE_SIZE 20

Typedef struct {
   Int key;
   Int value;
} Pair;

Void initializeHashTable(Pair *table, int size) {
   For (int I = 0; I < size; i++) {
      Table[i].key = -1; // -1 indicates an empty slot
      Table[i].value = -1;
   }
}

Void insertPair(Pair *table, int size, Pair pair) {
   Int index = pair.key % size; // Calculate the hash index

   // Linear probing to find an empty slot
   While (table[index].key != -1) {
      Index = (index + 1) % size;
   }

   // Insert the pair at the empty slot
   Table[index] = pair;
}

Void printHashTable(const Pair *table, int size) {
   Printf("Hash Table:\n");
```

```
    For (int I = 0; I < size; i++) {
        If (table[i].key != -1) {
            Printf(“[%d] -> {%d, %d}\n”, I, table[i].key, table[i].value);
        } else {
            Printf(“[%d] -> Empty\n”, i);
        }
    }
}

Int main() {
    Pair pairs[] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
    Int numPairs = sizeof(pairs) / sizeof(pairs[0]);

    Pair hashTable[TABLE_SIZE];
    initializeHashTable(hashTable, TABLE_SIZE);

    for (int I = 0; I < numPairs; i++) {
        insertPair(hashTable, TABLE_SIZE, pairs[i]);
    }

    // Additional insert operation
    Pair additionalPair = {3, 20};
    insertPair(hashTable, TABLE_SIZE, additionalPair);

    printHashTable(hashTable, TABLE_SIZE);

    return 0;
}
```

Write a C program Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Find(4): The key 4 is stored at the index (4%20 = 4). Therefore, print the 7 as it is the value of the key, 4, at index 4 of the Hash Table.

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 20

Typedef struct {
  Int key;
  Int value;
} Pair;

Void initializeHashTable(Pair *table, int size) {
  For (int I = 0; I < size; i++) {
    Table[i].key = -1; // -1 indicates an empty slot
    Table[i].value = -1;
  }
}

Void insertPair(Pair *table, int size, Pair pair) {
  Int index = pair.key % size; // Calculate the hash index

  // Linear probing to find an empty slot
  While (table[index].key != -1) {
    Index = (index + 1) % size;
  }

  // Insert the pair at the empty slot
```

```
    Table[index] = pair;
}

Int findValueByKey(const Pair *table, int size, int key) {
    Int index = key % size; // Calculate the hash index

    // Linear probing to find the pair with the given key
    While (table[index].key != key) {
        If (table[index].key == -1) {
            // Pair with the given key not found
            Return -1;
        }

        Index = (index + 1) % size;
    }

    // Return the value associated with the key
    Return table[index].value;
}

Void printHashTable(const Pair *table, int size) {
    Printf("Hash Table:\n");
    For (int I = 0; I < size; i++) {
        If (table[i].key != -1) {
            Printf("[%d] -> {%d, %d}\n", I, table[i].key, table[i].value);
        } else {
            Printf("[%d] -> Empty\n", i);
        }
    }
```

```
}

Int main() {
    Pair pairs[] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
    Int numPairs = sizeof(pairs) / sizeof(pairs[0]);

    Pair hashTable[TABLE_SIZE];
    initializeHashTable(hashTable, TABLE_SIZE);

    for (int I = 0; I < numPairs; i++) {
        insertPair(hashTable, TABLE_SIZE, pairs[i]);
    }

    // Find operation
    Int keyToFind = 4;
    Int value = findValueByKey(hashTable, TABLE_SIZE, keyToFind);

    If (value != -1) {
        Printf(“Value at key %d: %d\n”, keyToFind, value);
    } else {
        Printf(“Key %d not found in the hash table.\n”, keyToFind);
    }

    printHashTable(hashTable, TABLE_SIZE);

    return 0;
}
```

Write a C program Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Delete (4): The key 4 is stored at the index (4%20 = 4). After deleting Key 4, the Hash Table has keys {1, 2, 3}

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 20

Typedef struct {
    Int key;
    Int value;
} Pair;

Void initializeHashTable(Pair *table, int size) {
    For (int I = 0; I < size; i++) {
        Table[i].key = -1; // -1 indicates an empty slot
        Table[i].value = -1;
    }
}

Void insertPair(Pair *table, int size, Pair pair) {
    Int index = pair.key % size; // Calculate the hash index

    // Linear probing to find an empty slot
    While (table[index].key != -1) {
        Index = (index + 1) % size;
    }

    // Insert the pair at the empty slot
    Table[index] = pair;
}

Void deletePair(Pair *table, int size, int key) {
    Int index = key % size; // Calculate the hash index

    // Linear probing to find the pair with the given key
    While (table[index].key != key) {
        If (table[index].key == -1) {
            // Pair with the given key not found
            Return;
        }

        Index = (index + 1) % size;
    }

    // Delete the pair by marking it as empty (-1)
    Table[index].key = -1;
    Table[index].value = -1;
}

Void printHashTable(const Pair *table, int size) {
    Printf(“Hash Table:\n”);
    For (int I = 0; I < size; i++) {
        If (table[i].key != -1) {
            Printf(“[%d] -> {%d, %d}\n”, I, table[i].key, table[i].value);
        } else {
            Printf(“[%d] -> Empty\n”, i);
        }
    }
}

Int main() {
    Pair pairs[] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
    Int numPairs = sizeof(pairs) / sizeof(pairs[0]);

    Pair hashTable[TABLE_SIZE];
    initializeHashTable(hashTable, TABLE_SIZE);

    for (int I = 0; I < numPairs; i++) {
        insertPair(hashTable, TABLE_SIZE, pairs[i]);
    }

    // Delete operation
    Int keyToDelete = 4;
    deletePair(hashTable, TABLE_SIZE, keyToDelete);

    printf(“After deleting Key %d\n”, keyToDelete);
    printHashTable(hashTable, TABLE_SIZE);

    return 0;
}
```

---

Write a C program Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Find(4): Print -1, as the key 4 does not exist in the Hash Table.

```
#include <stdio.h>

#include <stdlib.h>
```

```
#define TABLE_SIZE 20

Typedef struct {
   Int key;
   Int value;
} Pair;

Void initializeHashTable(Pair *table, int size) {
   For (int I = 0; I < size; i++) {
      Table[i].key = -1; // -1 indicates an empty slot
      Table[i].value = -1;
   }
}

Void insertPair(Pair *table, int size, Pair pair) {
   Int index = pair.key % size; // Calculate the hash index

   // Linear probing to find an empty slot
   While (table[index].key != -1) {
      Index = (index + 1) % size;
   }

   // Insert the pair at the empty slot
   Table[index] = pair;
}

Int findValueByKey(const Pair *table, int size, int key) {
   Int index = key % size; // Calculate the hash index
```

```
  // Linear probing to find the pair with the given key
  While (table[index].key != key) {
    If (table[index].key == -1) {
      // Pair with the given key not found
      Return -1;
    }

    Index = (index + 1) % size;
  }

  // Return the value associated with the key
  Return table[index].value;
}

Void printHashTable(const Pair *table, int size) {
  Printf(“Hash Table:\n”);
  For (int I = 0; I < size; i++) {
    If (table[i].key != -1) {
      Printf(“[%d] -> {%d, %d}\n”, I, table[i].key, table[i].value);
    } else {
      Printf(“[%d] -> Empty\n”, i);
    }
  }
}

Int main() {
  Pair pairs[] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
  Int numPairs = sizeof(pairs) / sizeof(pairs[0]);
```

```
    Pair hashTable[TABLE_SIZE];
    initializeHashTable(hashTable, TABLE_SIZE);

    for (int I = 0; I < numPairs; i++) {
        insertPair(hashTable, TABLE_SIZE, pairs[i]);
    }

    // Find operation
    Int keyToFind = 4;
    Int value = findValueByKey(hashTable, TABLE_SIZE, keyToFind);

    If (value != -1) {
        Printf(“Value at key %d: %d\n”, keyToFind, value);
    } else {
        Printf(“Key %d not found in the hash table.\n”, keyToFind);
    }

    printHashTable(hashTable, TABLE_SIZE);

    return 0;
}
```