Name : Deshmukh Mehmood Rehan

MIS No. : 612303050

SY Computer Science – Div 1

What is Hashing?

Hashing is a technique used to map data of arbitrary size to fixed-size values. It is a common method used in computer science to quickly locate a data record given its search key. Hash functions are used to generate these fixed-size values, which are typically used as indices in data structures like hash tables. Hashing is widely used in various applications, including databases, cryptography, and data retrieval systems.

Hash Functions

Hash functions are at the core of hashing techniques. They take an input (or key) and produce a fixed-size output (hash value) that represents the input data. A good hash function should have the following properties:

1. Deterministic: For a given input, the hash function should always produce the same output.

2. Fast: The hash function should be computationally efficient to calculate the hash value quickly.

3. Uniformity: The hash function should distribute the hash values uniformly across the output space to minimize collisions.

4. Avalanche Effect: A small change in the input should result in a significantly different hash value.

5. Non-invertible: It should be computationally infeasible to reverse-engineer the input from the hash value.

There are several types of hash functions, each with its own characteristics. Some common types include:

1. Division Method: This simple hash function divides the key by a fixed number (the table size) and uses the remainder as the hash value. The division method is easy to implement but may lead to clustering and collisions if not carefully chosen.

2. Multiplication Method: This hash function multiplies the key by a constant factor, extracts a portion of the product, and uses it as the hash value. The multiplication method is known for its good distribution properties and is widely used in practice.

3. Universal Hashing: Universal hashing uses a family of hash functions, allowing for random selection of a hash function at runtime. This technique helps reduce the likelihood of collisions and is commonly used in scenarios where security is a concern.

Basic Hash Function Implementation along with a simple Hash Table

To understand the concept of hashing better, I implemented a simple hash function using the division method. The hash function takes an integer key as input, divides it by a fixed table size, and returns the remainder as the hash value. I then created a small program that hashes a series of integers using this function and observed how the data is distributed across an array of a fixed size.

Here is the implementation of the hash function in C:

Header.h

```c
#define SIZE 10

typedef struct Data {
    int key;
    int value;
} Data;


typedef struct HashTable{
    int size;
    Data* array;
} HashTable;


void init_hash_table(HashTable* ht, int size);
int hash(int key, int size);
void insert(HashTable* ht, int key, int value);
int search(HashTable* ht, int key);
int remove_node(HashTable* ht, int key);
void display(HashTable ht);
```

Logic.c

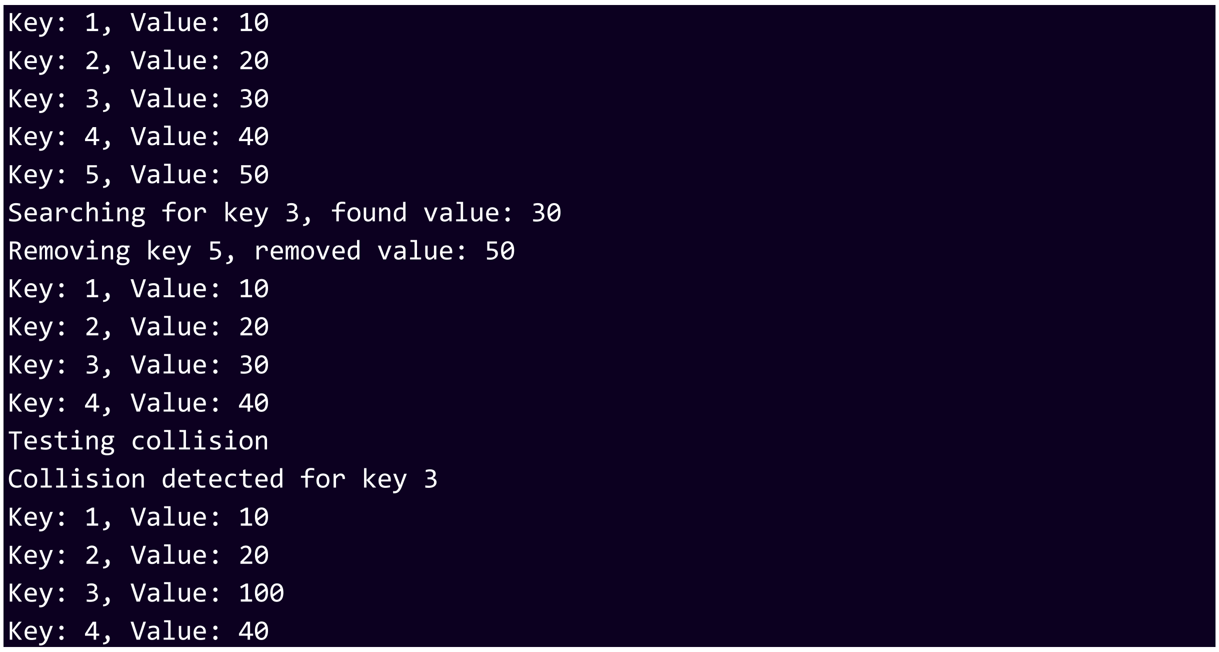
```c
#include "header.h"
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

/* Initialize the hash table */
void init_hash_table(HashTable* ht, int size) {
    ht->size = size;
    ht->array = (Data*)malloc(ht->size * sizeof(Data));

    for (int i = 0; i < ht->size; i++) {
        ht->array[i] = (Data){-1, -1};
    }
}

/* Hash function to map values to key */
int hash(int key, int size) {
    return key % size;
}

/* Insert key-value pair into hash table */
void insert(HashTable* ht, int key, int value) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == -1) {
        ht->array[index] = (Data){key, value};
    } else {
        printf("Collision detected for key %d\n", key);
    }

}

/* Search for a key in the hash table */
int search(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        return ht->array[index].value;
    }

    return -1;
}

/* Remove a key from the hash table */
int remove_node(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        ht->array[index] = (Data){-1, -1};
        return key;
    }

    return -1;
}

/* Display the contents of the hash table */
void display(HashTable ht) {
    for (int i = 0; i < ht.size; i++) {
        if(ht.array[i].key != -1)
            printf("Index: %d, Key: %d, Value: %d\n", i, ht.array[i].key,
ht.array[i].value);
    }
}
```

main.c

```c
#include <stdio.h>
#include "./header.h"

int main() {
    HashTable ht;
    init_hash_table(&ht, SIZE);

    insert(&ht, 1, 10);
    insert(&ht, 2, 20);
    insert(&ht, 3, 30);
    insert(&ht, 4, 40);
    insert(&ht, 5, 50);

    display(ht);

    int key = 3;
    printf("Searching for key %d, found value: %d\n", key, search(&ht, key));

    key = 5;
    printf("Removing key %d, removed value: %d\n", key, remove_node(&ht,
key));

    display(ht);

    printf("Testing collision\n");
    insert(&ht, 3, 100);
    display(ht);

    return 0;
}
```

Output



```
Key: 1, Value: 10
Key: 2, Value: 20
Key: 3, Value: 30
Key: 4, Value: 40
Key: 5, Value: 50
Searching for key 3, found value: 30
Removing key 5, removed value: 50
Key: 1, Value: 10
Key: 2, Value: 20
Key: 3, Value: 30
Key: 4, Value: 40
Testing collision
Collision detected for key 3
Key: 1, Value: 10
Key: 2, Value: 20
Key: 3, Value: 100
Key: 4, Value: 40
```

Observations and Insights

From the output of the program, we can see that the hash function using the division method successfully distributes the data across the hash table. The keys are hashed to their respective indices, and the values are stored at those locations. When a collision occurs (e.g., inserting key 3 again), the collision is detected, and the value is not overwritten.

This simple hash table implementation demonstrates the basic concept of hashing and how it can be used to store and retrieve key-value pairs efficiently. However, this implementation does not handle collisions effectively, as it does not provide a mechanism to resolve them. In real-world scenarios, we need collision handling techniques like chaining or open addressing to handle collisions and ensure efficient data retrieval.

In the next sections of this assignment, we will explore different collision handling techniques, including chaining, overflow handling without chaining, and open addressing methods like linear probing and quadratic probing. We will implement and analyze these techniques to understand their impact on data distribution, probe counts, and efficiency in handling collisions.

Hashing and rehashing

Hashing and rehashing are techniques used to handle overflows in hash tables without using chaining. When a collision occurs, instead of storing the data in a linked list (as in chaining), we can use a secondary hash function to find a new position for the data. This process is known as double hashing, where the secondary hash function is used to calculate the next available slot for the data.

Here is an example implementation of double hashing in C:

Logic.c

```c
#include "header.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

/* Initialize the hash table */
void init_hash_table(HashTable* ht, int size) {
    ht->size = size;
    ht->array = (Data*)malloc(ht->size * sizeof(Data));

    for (int i = 0; i < ht->size; i++) {
        ht->array[i] = (Data){-1, -1};
    }
}

/* Hash function to map values to key */
int hash(int key, int size) {
    return key % size;
}

/* Rehash function to handle collisions */
int rehash(int key, int size) {
    return 7 - (key % 7);
}

/* Insert key-value pair into hash table */
void insert(HashTable* ht, int key, int value) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == -1) {
        ht->array[index] = (Data){key, value};
    } else {
        printf("Collision detected for key %d\n", key);
        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i * rehash(key, ht->size)) % ht->size;
            if (ht->array[new_index].key == -1) {
                ht->array[new_index] = (Data){key, value};
                return;
            }
        }
        printf("Unable to insert key %d\n", key);
    }
}

/* Search for a key in the hash table */
int search(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        return ht->array[index].value;
    }else{
        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i * rehash(key, ht->size)) % ht->size;
            if (ht->array[new_index].key == key) {
                return ht->array[new_index].value;
            }
        }
    }

    return -1;
}

/* Remove a key from the hash table */
int remove_node(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        ht->array[index] = (Data){-1, -1};
        return key;
    }else{
        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i * rehash(key, ht->size)) % ht->size;
            if (ht->array[new_index].key == key) {
                ht->array[new_index] = (Data){-1, -1};
                return key;
            }
        }
    }

    return -1;
}

/* Display the contents of the hash table */
void display(HashTable ht) {
    for (int i = 0; i < ht.size; i++) {
        if(ht.array[i].key != -1)
            printf("Index: %d, Key: %d, Value: %d\n", i, ht.array[i].key,
ht.array[i].value);
    }
```

```c

}
```

Output

```
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Index: 5, Key: 5, Value: 50
Searching for key 3, found value: 30
Removing key 5, removed value: 5
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Testing collision
Collision detected for key 1
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Index: 7, Key: 1, Value: 100
```

Observations and Insights

In this implementation, we used double hashing to handle collisions in the hash table. When a collision occurs, we calculate a new index using a secondary hash function and find the next available slot for the data. This technique allows us to resolve collisions without using chaining and ensures that the data is stored efficiently in the hash table.

Hashing with Chaining

Chaining is a common technique used to handle collisions in hash tables. Instead of storing multiple values at the same hash index, we use linked lists to store the values. When a collision occurs, we append the new value to the linked list at that index. Chaining is an effective way to handle collisions and ensures that data is stored and retrieved efficiently.

Here is an example implementation of chaining in C:

Header.h

```c
#define SIZE 10

/* Data structure to store key-value pairs */
```

```c
typedef struct Node {
    int key;
    int value;
    struct Node* next;
} Node;

/* Data structure to store hash table */
typedef struct HashTable {
    struct Node** array;
    int size;
} HashTable;

/* function prototypes */
Node* create_node(int key, int value);
void init_hash_table(HashTable* ht, int size);
int hash(int key, int size);
void insert(HashTable* ht, int key, int value);
int search(HashTable* ht, int key);
int remove_node(HashTable* ht, int key);
void display(HashTable ht);
```

logic.c

```c
#include "header.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>


/*  Create a new node */
Node* create_node(int key, int value) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->key = key;
    new_node->value = value;
    new_node->next = NULL;

    return new_node;
}

/* Initialize the hash table */
void init_hash_table(HashTable* ht, int size) {
    ht->size = size;
    ht->array = (Node**)malloc(ht->size * sizeof(Node*));

    for (int i = 0; i < ht->size; i++) {
        ht->array[i] = NULL;
    }
}
```

```
/* Hash function to map values to key */
int hash(int key, int size) {
    return key % size;
}

/* Insert key-value pair into hash table */
void insert(HashTable* ht, int key, int value) {
    int index = hash(key, ht->size);
    Node* new_node = create_node(key, value);
    /* use chaining */
    if (ht->array[index] != NULL) {
        Node* temp = ht->array[index];
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    } else {
        ht->array[index] = new_node;
    }
}

/* Search for a key in the hash table */
int search(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index] != NULL) {
        Node* temp = ht->array[index];
        while (temp != NULL) {
            if (temp->key == key) {
                return temp->value;
            }
            temp = temp->next;
        }
    }

    return -1;
}

/* Remove a key from the hash table */
int remove_node(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index] != NULL) {
        Node* temp = ht->array[index];
        Node* prev = NULL;
        while (temp != NULL) {
            if (temp->key == key) {
```

```c
                if (prev == NULL) {
                    ht->array[index] = temp->next;
                } else {
                    prev->next = temp->next;
                }
                int value = temp->value;
                free(temp);
                return value;
            }
            prev = temp;
            temp = temp->next;
        }
    }

    return -1;
}

/* Display the contents of the hash table */
void display(HashTable ht) {
    for (int i = 0; i < ht.size; i++) {
        if(ht.array[i] != NULL) {
            Node* temp = ht.array[i];
            printf("Index %d: ", i);
            while (temp != NULL) {
                printf("(%d, %d) -> ", temp->key, temp->value);
                temp = temp->next;
            }
            printf("NULL\n");
        }
    }
}
```

Output

```
Index 1: (1, 10) -> NULL
Index 2: (2, 20) -> NULL
Index 3: (3, 30) -> NULL
Index 4: (4, 40) -> NULL
Index 5: (5, 50) -> NULL
Searching for key 3, found value: 30
Removing key 5, removed value: 50
Index 1: (1, 10) -> NULL
Index 2: (2, 20) -> NULL
Index 3: (3, 30) -> NULL
Index 4: (4, 40) -> NULL
Testing collision
Index 1: (1, 10) -> NULL
Index 2: (2, 20) -> NULL
Index 3: (3, 30) -> (3, 100) -> NULL
Index 4: (4, 40) -> NULL
```

Observations and Insights

In this implementation, we used chaining to handle collisions in the hash table. When a collision occurs, we store the values in a linked list at the same hash index. This technique allows us to efficiently store and retrieve data even when collisions occur. Chaining is a simple and effective way to handle collisions and ensures that data is distributed uniformly across the hash table.

Hashing with Open Addressing (Linear and Quadratic Probing)

Open addressing is another technique used to handle collisions in hash tables. Instead of using separate chains or linked lists, open addressing methods like linear probing and quadratic probing move to the next available slot when a collision occurs. Linear probing moves to the next slot in a linear sequence, while quadratic probing adjusts the probing sequence using quadratic increments.

Linear Probing Implementation

Here is an example implementation of linear probing in C:

Logic.c

```c
#include "header.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

/* Initialize the hash table */
void init_hash_table(HashTable* ht, int size) {
    ht->size = size;
    ht->array = (Data*)malloc(ht->size * sizeof(Data));

    for (int i = 0; i < ht->size; i++) {
        ht->array[i] = (Data){-1, -1};
    }
}

/* Hash function to map values to key */
int hash(int key, int size) {
    return key % size;
}

/* Insert key-value pair into hash table */
void insert(HashTable* ht, int key, int value) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == -1) {
        ht->array[index] = (Data){key, value};
    } else {
        /* linear probing */
        printf("Collision detected for key %d\n", key);

        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i) % ht->size;
            if (ht->array[new_index].key == -1) {
                ht->array[new_index] = (Data){key, value};
                return;
            }
        }

        printf("Unable to insert key %d\n", key);
    }

}

/* Search for a key in the hash table */
int search(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        return ht->array[index].value;
    }else{
        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i) % ht->size;
            if (ht->array[new_index].key == key) {
                return ht->array[new_index].value;
            }
        }
    }

    return -1;
}

/* Remove a key from the hash table */
int remove_node(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        ht->array[index] = (Data){-1, -1};
        return key;
    }else{
        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i) % ht->size;
            if (ht->array[new_index].key == key) {
                ht->array[new_index] = (Data){-1, -1};
                return key;
            }
        }
    }

    return -1;
}

/* Display the contents of the hash table */
void display(HashTable ht) {
    for (int i = 0; i < ht.size; i++) {
        if(ht.array[i].key != -1)
            printf("Index: %d, Key: %d, Value: %d\n", i, ht.array[i].key,
ht.array[i].value);
    }
}
```

Output

```
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Index: 5, Key: 5, Value: 50
Searching for key 3, found value: 30
Removing key 5, removed value: 5
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Testing collision
Collision detected for key 3
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Index: 5, Key: 3, Value: 100
```

Observations and Insights

In this implementation, we used linear probing to handle collisions in the hash table. When a collision occurs, we move to the next available slot in a linear sequence until an empty slot is found. This technique ensures that data is stored efficiently in the hash table and minimizes clustering. Linear probing is a simple and effective way to handle collisions and ensures that data is distributed uniformly across the hash table.

Quadratic Probing Implementation

Here is an example implementation of quadratic probing in C:

Logic.c

```c
#include "header.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

/* Initialize the hash table */
void init_hash_table(HashTable* ht, int size) {
    ht->size = size;
    ht->array = (Data*)malloc(ht->size * sizeof(Data));

    for (int i = 0; i < ht->size; i++) {
        ht->array[i] = (Data){-1, -1};
    }
}

/* Hash function to map values to key */
int hash(int key, int size) {
    return key % size;
}

/* Insert key-value pair into hash table */
void insert(HashTable* ht, int key, int value) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == -1) {
        ht->array[index] = (Data){key, value};
    } else {
        /* quadratic probing */
        printf("Collision detected for key %d\n", key);

        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i * i) % ht->size;
            if (ht->array[new_index].key == -1) {
                ht->array[new_index] = (Data){key, value};
                return;
            }
        }

        printf("Unable to insert key %d\n", key);
    }
```

```c
}

/* Search for a key in the hash table */
int search(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        return ht->array[index].value;
    }else{
        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i * i) % ht->size;
            if (ht->array[new_index].key == key) {
                return ht->array[new_index].value;
            }

        }
    }

    return -1;
}

/* Remove a key from the hash table */
int remove_node(HashTable* ht, int key) {
    int index = hash(key, ht->size);

    if (ht->array[index].key == key) {
        ht->array[index] = (Data){-1, -1};
        return key;
    }else{
        for(int i = 1; i < ht->size; i++) {
            int new_index = (index + i * i) % ht->size;
            if (ht->array[new_index].key == key) {
                ht->array[new_index] = (Data){-1, -1};
                return key;
            }
        }
    }

    return -1;
}

/* Display the contents of the hash table */
void display(HashTable ht) {
    for (int i = 0; i < ht.size; i++) {
        if(ht.array[i].key != -1)
            printf("Index: %d, Key: %d, Value: %d\n", i, ht.array[i].key,
ht.array[i].value);
```

```
    }
}
```

Output

```
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Index: 5, Key: 5, Value: 50
Searching for key 3, found value: 30
Removing key 5, removed value: 5
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Testing collision
Collision detected for key 3
Index: 1, Key: 1, Value: 10
Index: 2, Key: 2, Value: 20
Index: 3, Key: 3, Value: 30
Index: 4, Key: 4, Value: 40
Index: 7, Key: 3, Value: 100
```

Observations and Insights

In this implementation, we used quadratic probing to handle collisions in the hash table. When a collision occurs, we adjust the probing sequence using quadratic increments until an empty slot is found. This technique helps reduce clustering and ensures that data is stored efficiently in the hash table. Quadratic probing is a simple and effective way to handle collisions and ensures that data is distributed uniformly across the hash table.

Comparison of Collision Handling Techniques

In this assignment, we explored different collision handling techniques, including chaining, overflow handling without chaining, linear probing, and quadratic probing. Each technique has its own advantages and disadvantages, and the choice of technique depends on the specific requirements of the application.

Here is a comparison of the collision handling techniques based on our experiments:

1. Chaining:

- Advantages: Chaining is simple to implement and allows for efficient handling of collisions. It ensures that data is stored and retrieved efficiently, even when collisions occur.

- **Disadvantages:** Chaining may lead to increased memory overhead due to the use of linked lists. It may also result in slower performance for large hash tables.

2. Overflow Handling Without Chaining:

- **Advantages:** Overflow handling without chaining, such as double hashing, provides an alternative way to handle collisions. It allows for efficient storage and retrieval of data without the need for linked lists.

- **Disadvantages**: Double hashing may require additional computation to find new positions for data when collisions occur. It may also result in clustering if the secondary hash function is not carefully chosen.

3. Open Addressing (Linear and Quadratic Probing):

- **Advantages:** Open addressing methods like linear probing and quadratic probing provide efficient ways to handle collisions without using separate chains. They ensure that data is stored uniformly across the hash table and minimize clustering.

- **Disadvantages:** Linear probing may result in clustering if consecutive slots are filled, leading to slower performance. Quadratic probing may require additional computation to find the next available slot, especially for large hash tables.

In conclusion, each collision handling technique has its own trade-offs in terms of memory usage, performance, and efficiency. The choice of technique depends on the specific requirements of the application and the characteristics of the data being stored. By experimenting with different techniques and analyzing their performance, we can determine the most suitable collision handling method for a given scenario.