

MIT WORLD PEACE UNIVERSITY

Advanced Data Structures  
Second Year B. Tech, Semester 4

---

---

LINEAR PROBING WITH AND WITHOUT  
REPLACEMENT FOR HASHING

---

---

ASSIGNMENT NO. 8

Prepared By

Krishnaraj Thadesar  
Cyber Security and Forensics  
Batch A1, PA 20

April 15, 2023

# Contents

<b>1 Objectives</b>	<b>1</b>
<b>2 Problem Statement</b>	<b>1</b>
<b>3 Theory</b>	<b>1</b>
3.1 What is Hashing? . . . . .	1
3.1.1 Hashing in Comparison with other Searching Techniques . . . . .	1
3.2 Hash Functions . . . . .	2
3.2.1 Hash Function . . . . .	2
3.3 Collision Resolution Techniques . . . . .	2
3.3.1 Without Replacement . . . . .	3
3.3.2 With Replacement . . . . .	3
<b>4 Platform</b>	<b>4</b>
<b>5 Test Conditions</b>	<b>4</b>
<b>6 Input and Output</b>	<b>4</b>
<b>7 Pseudo Code</b>	<b>4</b>
7.1 Linear Probing with Replacement . . . . .	4
7.2 Linear Probing without Replacement . . . . .	4
<b>8 Time Complexity</b>	<b>5</b>
8.1 Linear Probing . . . . .	5
<b>9 Code</b>	<b>5</b>
9.1 Program . . . . .	5
<b>10 Conclusion</b>	<b>15</b>
<b>11 FAQ</b>	<b>16</b>

## 1 Objectives

1. To study hashing techniques
2. To implement different hashing techniques
3. To study and implement linear probing with and without replacement
4. To study how hashing can be used to model real world problems

## 2 Problem Statement

Implement Direct access file using hashing (linear probing with and without replacement) perform following operations on it:

1. Create Database
2. Display Database
3. Add a record
4. Search a record
5. Modify a record

## 3 Theory

### 3.1 What is Hashing?

Hashing is a technique to convert a range of key values into a range of indexes of an array. The idea is to use the key itself as an index into an array that is why it is also called as direct access table.

#### 3.1.1 Hashing in Comparison with other Searching Techniques

- **Sequential Search:**

- The worst case time complexity of the sequential search is  $O(n)$
- The worst case time complexity of the hashing is  $O(1)$
- The sequential search is not suitable for large data sets
- The hashing is suitable for large data sets

- **Binary Search:**

- The worst case time complexity of the binary search is  $O(\log n)$
- The worst case time complexity of the hashing is  $O(1)$
- The binary search is only suitable for sorted data sets
- The hashing is suitable for unsorted data sets

- **Binary Tree:**

- The worst case time complexity of the binary tree search is  $O(\log n)$
- The worst case time complexity of the hashing is  $O(1)$
- The binary tree search is only suitable for sorted data sets
- The hashing is suitable for unsorted data sets

### 3.2 Hash Functions

#### 3.2.1 Hash Function

A hash function is a function that maps a given key to a location in the hash table. The hash function is used to calculate the index of the array where the data is to be stored or retrieved from.

Different types of Hash functions are:

##### 1. Division Method

The division method is one of the simplest hashing methods. It works by computing the remainder of the key when divided by the table size, using a hash function of the form  $h(key) = key \bmod table\_size$ . The result is the index of the slot in the hash table where the key-value pair should be stored.

##### 2. Multiplication Method

The multiplication method is another common hashing method. It works by multiplying the key by a constant  $A$  in the range  $(0, 1)$  and then extracting the fractional part of the product. The result is then multiplied by the table size and rounded down to obtain the index of the slot in the hash table where the key-value pair should be stored. The hash function has the form  $h(key) = \lfloor table\_size * (key * A \bmod 1) \rfloor$ .

##### 3. Universal Hashing

Universal hashing is a family of hashing methods that use a randomly chosen hash function from a set of functions to minimize collisions. The set of functions is chosen to be large enough that the probability of two different keys having the same hash value is small, and the function is chosen randomly each time a new hash table is created. The hash function has the form  $h(key) = ((a * key + b) \bmod p) \bmod table\_size$ , where  $a$  and  $b$  are randomly chosen integers and  $p$  is a large prime number.

##### 4. Perfect Hashing

Perfect hashing is a technique that is used when the keys are known in advance and fixed. It works by creating a hash function that maps each key to a unique index in the hash table, without any collisions. This is achieved by using two levels of hashing: the first level maps the keys to a set of buckets, and the second level uses a different hash function to map each key within a bucket to a unique index in the hash table. This approach guarantees that there are no collisions, but requires more memory and computation than other hashing methods.

### 3.3 Collision Resolution Techniques

The main types of Collision Resolution techniques are:

- **Open Addressing:**

Open addressing is a family of hashing methods that use the hash table itself to resolve collisions, by storing each key-value pair in the next available slot in the table. There are several methods of open addressing, including:

- **Linear Probing**

Linear probing is an open addressing method where, when a collision occurs, the algorithm searches for the next available slot in the table, by linearly checking each slot in the table until an empty slot is found. The hash function has the form  $h(key, i) = (h'(key) + i) \bmod table\_size$ , where  $h'(key)$  is the primary hash function and  $i$  is the number of the probe.

- **Quadratic Probing**

Quadratic probing is similar to linear probing, but uses a quadratic function to search for the next available slot. The hash function has the form  $h(key, i) = (h'(key) + c_1 \cdot i + c_2 \cdot i^2) \bmod table\_size$ , where  $c_1$  and  $c_2$  are constants that depend on the hash table size.

- **Double Hashing**

Double hashing is another open addressing method that uses two hash functions to determine the next available slot in the table. The hash function has the form  $h(key, i) = (h_1(key) + i \cdot h_2(key)) \bmod table\_size$ , where  $h_1$  and  $h_2$  are two different hash functions.

- **Separate Chaining**

Separate chaining is a hashing method that uses linked lists to store the key-value pairs that hash to the same slot in the table. When a collision occurs, the key-value pair is added to the linked list at the appropriate slot. The hash function has the form  $h(key) = key \bmod table\_size$ .

- **Double Hashing**

Double hashing is both an open addressing method and a separate chaining method. It uses two hash functions to determine the slot in the table, and if there is a collision, it uses a linked list to store the key-value pairs that hash to the same slot. The hash function has the form  $h(key, i) = (h_1(key) + i \cdot h_2(key)) \bmod table\_size$ , where  $h_1$  and  $h_2$  are two different hash functions.

Most of these techniques can be implemented in 2 ways

### 3.3.1 Without Replacement

[Without Replacement] In this technique, when a collision occurs, the new element is simply discarded.

### 3.3.2 With Replacement

[With Replacement] In this technique, when a collision occurs, the old element is replaced by the new element.

## 4 Platform

**Operating System:** Arch Linux x86-64

**IDEs or Text Editors Used:** Visual Studio Code

**Compilers :** g++ and gcc on linux for C++

## 5 Test Conditions

1. Input at least 10 nodes.
2. Display collision with replacement and without replacement.

## 6 Input and Output

1. The minimum cost of the spanning tree.

## 7 Pseudo Code

### 7.1 Linear Probing with Replacement

```
1 void HashTable::insert_without_replacement(int key)
2 {
3     int index = hash(key);
4     if (table[index] == -1)
5         table[index] = key;
6     else
7     {
8         int i = 1;
9         while (table[(index + i) % SIZE] != -1)
10             i++;
11         table[(index + i) % SIZE] = key;
12     }
13 }
```

### 7.2 Linear Probing without Replacement

```
1 void HashTable::insert_with_replacement(int key)
2 {
3     int index = hash(key);
4
5     // there is no value there.
6     if (table[index] == -1)
7         table[index] = key;
8     // the value that is already there, belongs there, then check, and then
9     // find another empty slot and insert there.
10    else if (hash(table[index]) == index)
11    {
12        int i = 1;
13        while (table[(index + i) % SIZE] != -1)
14            i++;
15        table[(index + i) % SIZE] = key;
16    }
```

```
15     }
16     // the value that is already there, does not belong there, then replace it
    with the new value, and push the existing value down.
17     else
18     {
19         // find empty slot
20         int i = 1;
21         while (table[(index + i) % SIZE] != -1)
22             i++;
23
24         int temp = table[index]; // current value that doesnt belong there.
25         table[index] = key;
26         table[(index + i) % SIZE] = temp;
27     }
28 }
```

## 8 Time Complexity

### 8.1 Linear Probing

- **Time Complexity:**

$$O(n)$$

It would be 1 if the it was the best case, and n for worst case, where all the slots would be filled, and we would have to keep probing over all the elements.

- **Space Complexity:**

$$O(n)$$

For storing the Table, as there is no additional array or table required to store and hash.

## 9 Code

### 9.1 Program

```
1 // Program for linear probing with and without replacement.
2
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
7 const int SIZE = 10;
8
9 class Employee
10 {
11
12 public:
13     string name;
14     int id;
15     int age;
16     Employee()
17     {
18         name = "";
19         id = 0;
20         age = 0;
```

```
21     }
22     Employee(string name, int id, int age)
23     {
24         this->name = name;
25         this->id = id;
26         this->age = age;
27     }
28     void accept_data()
29     {
30         cout << "Enter name: " << endl;
31         cin >> name;
32         cout << "Enter id: " << endl;
33         cin >> id;
34         cout << "Enter age: " << endl;
35         cin >> age;
36     }
37
38     void print_data()
39     {
40         cout << "Name: " << name << endl;
41         cout << "Id: " << id << endl;
42         cout << "Age: " << age << endl;
43     }
44 };
45
46 class HashTable
47 {
48 public:
49     HashTable();
50     int insert_without_replacement(int key);
51     int insert_with_replacement(int key);
52     void print();
53     int table[SIZE];
54     void read_data_from_file()
55     {
56         // read the records from the file
57         cout << "Reading data from the file. " << endl;
58         Employee temp;
59         fstream file("data.txt", ios::binary | ios::in);
60         for (int i = 0; i < SIZE; i++)
61         {
62             file.read((char *)&temp, sizeof(Employee));
63             cout << "Employee " << i + 1 << ": " << endl;
64             temp.print_data();
65             cout << endl;
66         }
67         // file.close();
68     }
69
70     void write_data_to_file(Employee hashed_employees[])
71     {
72         // Create a file in binary write mode.
73         fstream file("data.txt", ios::binary | ios::out);
74
75         // Write all the employee objects into the file in the order of the hash
76         // employee array.
77         for (int i = 0; i < SIZE; i++)
78         {
79             cout << "Writing employee " << i + 1 << endl;
```



```
79         cout << "Employee " << i + 1 << ": " << endl;
80         hashed_employees[i].print_data();
81         file.write((char *)&hashed_employees[i], sizeof(Employee));
82     }
83     // Close the file.
84     file.close();
85 }
86
87 private:
88     int hash(int key);
89 };
90
91 HashTable::HashTable()
92 {
93     for (int i = 0; i < SIZE; i++)
94         table[i] = -1;
95 }
96
97 int HashTable::hash(int key)
98 {
99     return key % SIZE;
100 }
101
102 int HashTable::insert_without_replacement(int key)
103 {
104     int index = hash(key);
105     if (table[index] == -1)
106         table[index] = key;
107     else
108     {
109         int i = 1;
110         while (table[(index + i) % SIZE] != -1)
111         {
112             i++;
113             if (i == SIZE)
114             {
115                 cout << "Table is full. " << endl;
116                 return 1;
117             }
118         }
119         table[(index + i) % SIZE] = key;
120     }
121     return 0;
122 }
123
124 int HashTable::insert_with_replacement(int key)
125 {
126     int index = hash(key);
127
128     // there is no value there.
129     if (table[index] == -1)
130         table[index] = key;
131     // the value that is already there, belongs there, then check, and then find
132     // another empty slot and insert there.
133     else if (hash(table[index]) == index)
134     {
135         int i = 1;
136         while (table[(index + i) % SIZE] != -1)
```

```
137         i++;
138         if (i == SIZE)
139         {
140             cout << "Table is full. " << endl;
141             return 1;
142         }
143     }
144     table[(index + i) % SIZE] = key;
145 }
146 // the value that is already there, does not belong there, then replace it
147 // with the new value, and push the existing value down.
148 else
149 {
150     // find empty slot
151     int i = 1;
152     while (table[(index + i) % SIZE] != -1)
153     {
154         i++;
155         if (i == SIZE)
156         {
157             cout << "Table is full. " << endl;
158             return 1;
159         }
160     }
161     int temp = table[index]; // current value that doesnt belong there.
162     table[index] = key;
163     table[(index + i) % SIZE] = temp;
164 }
165 return 0;
166 }
167
168 void HashTable::print()
169 {
170     for (int i = 0; i < SIZE; i++)
171         cout << i << ":" << table[i] << " " << endl;
172     cout << endl;
173 }
174
175 int main()
176 {
177     int size;
178     cout << "Enter size of table: " << endl;
179     cin >> size;
180     Employee employees[size];
181     Employee hashed_employees[SIZE];
182     HashTable hash_table;
183     // Accept data for all employees
184     for (int i = 0; i < size; i++)
185     {
186         cout << "Enter data for employee " << i + 1 << endl;
187         employees[i].accept_data();
188         cout << endl
189             << endl;
190         cout << "Employee " << i + 1 << ": " << endl;
191         employees[i].print_data();
192         cout << endl;
193     }
194 }
```

## Advanced Data Structures - Assignment 8

---

```
195 // Provide choice to users if they wanna insert with or without replacement
196 int choice;
197 cout << "Enter 1 to insert with replacement, 2 to insert without replacement:
" << endl;
198 cin >> choice;
199 if (choice == 1)
200 {
201     // Insert data into hash table with replacement
202     for (int i = 0; i < size; i++)
203     {
204         hash_table.insert_with_replacement(employees[i].id);
205     }
206 }
207 else
208 {
209     // Insert data into hash table without replacement
210     for (int i = 0; i < size; i++)
211     {
212         hash_table.insert_without_replacement(employees[i].id);
213     }
214 }
215
216 cout << "Hash table now looks like this. " << endl;
217 hash_table.print();
218
219 cout << "Inserting data into the file. " << endl;
220
221 // depending on the order of id in hash table, write the hashed_employee array
222 for (int i = 0; i < SIZE; i++)
223 {
224     for (int j = 0; j < size; j++)
225     {
226         if (employees[j].id == hash_table.table[i])
227         {
228             hashed_employees[i].age = employees[j].age;
229             hashed_employees[i].id = employees[j].id;
230             hashed_employees[i].name = employees[j].name;
231         }
232     }
233 }
234 hash_table.write_data_to_file(hashed_employees);
235 hash_table.read_data_from_file();
236
237 return 0;
238 }
```

```
1 Enter size of table:
2 4
3 Enter data for employee 1
4 Enter name:
5 Krish
6 Enter id:
7 12
8 Enter age:
9 21
10
11
12 Employee 1:
13 Name: Krish
```

## Advanced Data Structures - Assignment 8

---

```
14 Id: 12
15 Age: 21
16
17 Enter data for employee 2
18 Enter name:
19 Part
20 Enter id:
21 42
22 Enter age:
23 22
24
25
26 Employee 2:
27 Name: Part
28 Id: 42
29 Age: 22
30
31 Enter data for employee 3
32 Enter name:
33 Ram
34 Enter id:
35 23
36 Enter age:
37 32
38
39
40 Employee 3:
41 Name: Ram
42 Id: 23
43 Age: 32
44
45 Enter data for employee 4
46 Enter name:
47 Ramesh
48 Enter id:
49 24
50 Enter age:
51 21
52
53
54 Employee 4:
55 Name: Ramesh
56 Id: 24
57 Age: 21
58
59 Enter 1 to insert with replacement, 2 to insert without replacement:
60 1
61 Hash table now looks like this.
62 0:-1
63 1:-1
64 2:12
65 3:23
66 4:24
67 5:42
68 6:-1
69 7:-1
70 8:-1
71 9:-1
72
```

```
73 Inserting data into the file.
74 Writing employee 1
75 Employee 1:
76 Name:
77 Id: 0
78 Age: 0
79 Writing employee 2
80 Employee 2:
81 Name:
82 Id: 0
83 Age: 0
84 Writing employee 3
85 Employee 3:
86 Name: Krish
87 Id: 12
88 Age: 21
89 Writing employee 4
90 Employee 4:
91 Name: Ram
92 Id: 23
93 Age: 32
94 Writing employee 5
95 Employee 5:
96 Name: Ramesh
97 Id: 24
98 Age: 21
99 Writing employee 6
100 Employee 6:
101 Name: Part
102 Id: 42
103 Age: 22
104 Writing employee 7
105 Employee 7:
106 Name:
107 Id: 0
108 Age: 0
109 Writing employee 8
110 Employee 8:
111 Name:
112 Id: 0
113 Age: 0
114 Writing employee 9
115 Employee 9:
116 Name:
117 Id: 0
118 Age: 0
119 Writing employee 10
120 Employee 10:
121 Name:
122 Id: 0
123 Age: 0
124 Reading data from the file.
125 Employee 1:
126 Name:
127 Id: 0
128 Age: 0
129
130 Employee 2:
131 Name:
```

```
132 Id: 0
133 Age: 0
134
135 Employee 3:
136 Name: Krish
137 Id: 12
138 Age: 21
139
140 Employee 4:
141 Name: Ram
142 Id: 23
143 Age: 32
144
145 Employee 5:
146 Name: Ramesh
147 Id: 24
148 Age: 21
149
150 Employee 6:
151 Name: Part
152 Id: 42
153 Age: 22
154
155 Employee 7:
156 Name:
157 Id: 0
158 Age: 0
159
160 Employee 8:
161 Name:
162 Id: 0
163 Age: 0
164
165 Employee 9:
166 Name:
167 Id: 0
168 Age: 0
169
170 Employee 10:
171 Name:
172 Id: 0
173 Age: 0
174
175 Enter size of table:
176 2
177 Enter data for employee 1
178 Enter name:
179 krish
180 Enter id:
181 124
182 Enter age:
183 21
184
185
186 Employee 1:
187 Name: krish
188 Id: 124
189 Age: 21
190
```

## Advanced Data Structures - Assignment 8

---

```
191 Enter data for employee 2
192 Enter name:
193 Tony
194 Enter id:
195 4
196 Enter age:
197 23
198
199
200 Employee 2:
201 Name: Tony
202 Id: 4
203 Age: 23
204
205 Enter 1 to insert with replacement, 2 to insert without replacement:
206 2
207 Hash table now looks like this.
208 0:-1
209 1:-1
210 2:-1
211 3:-1
212 4:124
213 5:4
214 6:-1
215 7:-1
216 8:-1
217 9:-1
218
219 Inserting data into the file.
220 Writing employee 1
221 Employee 1:
222 Name:
223 Id: 0
224 Age: 0
225 Writing employee 2
226 Employee 2:
227 Name:
228 Id: 0
229 Age: 0
230 Writing employee 3
231 Employee 3:
232 Name:
233 Id: 0
234 Age: 0
235 Writing employee 4
236 Employee 4:
237 Name:
238 Id: 0
239 Age: 0
240 Writing employee 5
241 Employee 5:
242 Name: krish
243 Id: 124
244 Age: 21
245 Writing employee 6
246 Employee 6:
247 Name: Tony
248 Id: 4
249 Age: 23
```

## *Advanced Data Structures - Assignment 8*

---

```
250 Writing employee 7
251 Employee 7:
252 Name:
253 Id: 0
254 Age: 0
255 Writing employee 8
256 Employee 8:
257 Name:
258 Id: 0
259 Age: 0
260 Writing employee 9
261 Employee 9:
262 Name:
263 Id: 0
264 Age: 0
265 Writing employee 10
266 Employee 10:
267 Name:
268 Id: 0
269 Age: 0
270 Reading data from the file.
271 Employee 1:
272 Name:
273 Id: 0
274 Age: 0
275
276 Employee 2:
277 Name:
278 Id: 0
279 Age: 0
280
281 Employee 3:
282 Name:
283 Id: 0
284 Age: 0
285
286 Employee 4:
287 Name:
288 Id: 0
289 Age: 0
290
291 Employee 5:
292 Name: krish
293 Id: 124
294 Age: 21
295
296 Employee 6:
297 Name: Tony
298 Id: 4
299 Age: 23
300
301 Employee 7:
302 Name:
303 Id: 0
304 Age: 0
305
306 Employee 8:
307 Name:
308 Id: 0
```



```
309 Age: 0
310
311 Employee 9:
312 Name:
313 Id: 0
314 Age: 0
315
316 Employee 10:
317 Name:
318 Id: 0
319 Age: 0
```

## **10 Conclusion**

Thus, we have implemented linear probing with and without replacement.

## 11 FAQ

### 1. Write different types of hash functions.

There are several types of Hashing Functions. Here are a few:

- **Division Method:** This method is the simplest of all hash functions. It simply divides the key by the table size and uses the remainder as the hash value. The hash function is:

$$h(k) = k \mod m$$

- **Multiplication Method:** In this method, the hash value is obtained by multiplying the key with a constant A and then taking the fractional part.
- **Universal Hashing:** In this method, the hash function is obtained by using a universal hash function.
- **Mid Square Method:** In this method, the key is first squared and the middle digits are then taken as the hash value.
- **Random Number Method:** In this method, a random number is generated and then multiplied with the key to obtain the hash value.
- **Folding Method:** In this method, the key is divided into equal parts and then added to obtain the hash value.
- **Exponential Method:** In this method, the key is multiplied by a constant A and then the fractional part is taken as the hash value.
- **Truncation Method:** In this method, the key is divided into equal parts and then the first part is taken as the hash value.

### 2. Explain chaining with and without replacement with example.

*Chaining is a collision resolution technique used in hash tables to resolve collisions by storing multiple keys in the same slot in the table, with each slot containing a linked list of key-value pairs.*

- (a) **With Replacement:** Chaining with replacement involves replacing the old value with the new one when a collision occurs, while chaining without replacement involves inserting the new value into the linked list without replacing the old one.

Here is an example of chaining with replacement:

Suppose we have a hash table of size 5, and the hash function maps keys to slots as follows:

- key "apple"  $\rightarrow$  slot 3
- key "banana"  $\rightarrow$  slot 1
- key "cherry"  $\rightarrow$  slot 3
- key "date"  $\rightarrow$  slot 0

When we try to insert the key "cherry", we find that slot 3 is already occupied by the key "apple". In chaining with replacement, we replace the old value ("apple") with the new one ("cherry"), resulting in the linked list at slot 3 containing only the key-value pair ("cherry", value). Similarly, when we try to insert the key "orange", we find that slot 1 is already occupied by the key "banana". We replace the old value ("banana") with the new one ("orange"), resulting in the linked list at slot 1 containing only the key-value pair ("orange", value). The resulting hash table looks like this:

- slot 0: ("date", value)
- slot 1: ("orange", value)
- slot 2: empty
- slot 3: ("cherry", value)

(b) Chaining without replacement:

Suppose we have the same hash table and keys as in the previous example.

When we try to insert the key "cherry", we find that slot 3 is already occupied by the key "apple". In chaining without replacement, we add the key-value pair ("cherry", value) to the linked list at slot 3, resulting in the linked list containing both key-value pairs ("apple", value) and ("cherry", value).

Similarly, when we try to insert the key "orange", we find that slot 1 is already occupied by the key "banana". We add the key-value pair ("orange", value) to the linked list at slot 1, resulting in the linked list containing both key-value pairs ("banana", value) and ("orange", value).

The resulting hash table looks like this:

- slot 0: ("date", value)
- slot 1: ("banana", value) → ("orange", value)
- slot 2: empty
- slot 3: ("apple", value) → ("cherry", value)

In chaining without replacement, multiple key-value pairs can be stored in the same slot, without replacing any existing values.

### 3. Explain quadratic probing with example

**Quadratic Probing** *Quadratic probing is a technique used to resolve collisions in hash tables. When a collision occurs, meaning that two or more keys are mapped to the same slot, quadratic probing searches for the next available slot by adding a quadratic sequence of values to the original hash value until an empty slot is found.*

To illustrate how quadratic probing works, consider the following example. We have a hash table with 10 slots, and the following keys are inserted using a hash function:

- Slot 0: ("date", value)
- Slot 1: empty
- Slot 2: empty
- Slot 3: ("apple", value)
- Slot 4: empty
- Slot 5: empty
- Slot 6: empty
- Slot 7: ("banana", value)
- Slot 8: empty
- Slot 9: ("cherry", value)

Suppose we want to insert the key "fig" into the hash table. The hash function maps "fig" to slot 9, but we find that slot 9 is already occupied by the key "cherry".

To resolve this collision using quadratic probing, we start at slot 9 and search for the next available slot by adding a quadratic sequence of values to the original hash value.

Here's how we can find the next available slot using quadratic probing:

- (a) Starting from slot 9, we add 1 to get slot 0. But slot 0 is already occupied by "date".
- (b) We add 4 to the original hash value to get slot 13. We need to wrap around to the beginning of the hash table since the hash table only has 10 slots. So slot 13 becomes slot 3. But slot 3 is already occupied by "apple".
- (c) We add 9 to the original hash value to get slot 18. We need to wrap around again to the beginning of the hash table. So slot 18 becomes slot 8. But slot 8 is empty, so we can insert "fig" into slot 8.

We insert the key-value pair ("fig", value) into slot 8, and the resulting hash table looks like this:

- Slot 0: ("date", value)
- Slot 1: empty
- Slot 2: empty
- Slot 3: ("apple", value)
- Slot 4: empty
- Slot 5: empty
- Slot 6: empty
- Slot 7: ("banana", value)
- Slot 8: ("fig", value)
- Slot 9: ("cherry", value)

Here, quadratic probing allowed us to find the next available slot by searching through a quadratic sequence of values until an empty slot was found.