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

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 $h(key) = key \mod table_size$. 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 $h(key) = \lfloor table_size * (key * A \mod 1) \rfloor$.

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 $h(key) = ((a * key + b) \mod p) \mod table_size$, 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 $h(key, i) = (h'(key) + i) \mod table\_size$, 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 $h(key, i) = (h'(key) + c_1 \cdot i + c_2 \cdot i^2) \mod table\_size$, 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 $h(key, i) = (h_1(key) + i \cdot h_2(key)) \mod table\_size$, 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 $h(key) = key \mod table\_size$.

- **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 $h(key, i) = (h_1(key) + i \cdot h_2(key)) \mod table\_size$, 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

```
void HashTable::insert_without_replacement(int key)
{
    int index = hash(key);
    if (table[index] == -1)
        table[index] = key;
    else
    {
        int i = 1;
        while (table[(index + i) % SIZE] != -1)
            i++;
        table[(index + i) % SIZE] = key;
    }
}
```

7.2 Linear Probing without Replacement

```
void HashTable::insert_with_replacement(int key)
    {
        int index = hash(key);

        // there is no value there.
        if (table[index] == -1)
            table[index] = key;
        // the value that is already there, belongs there, then check, and then find another empty slot and insert there.
        else if (hash(table[index]) == index)
        {
            int i = 1;
            while (table[(index + i) % SIZE] != -1)
                i++;
            table[(index + i) % SIZE] = key;
```

```
        }
        // the value that is already there, does not belong there, then replace it
    with the new value, and push the existing value down.
        else
        {
            // find empty slot
            int i = 1;
            while (table[(index + i) % SIZE] != -1)
                i++;

            int temp = table[index]; // current value that doesnt belong there.
            table[index] = key;
            table[(index + i) % SIZE] = temp;
        }
    }
```

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

```
// Program for linear probing with and without replacement.

#include <iostream>
#include <fstream>
using namespace std;

const int SIZE = 10;

class Employee
{

public:
    string name;
    int id;
    int age;
    Employee()
    {
        name = "";
        id = 0;
        age = 0;
```

```
    }
    Employee(string name, int id, int age)
    {
        this->name = name;
        this->id = id;
        this->age = age;
    }
    void accept_data()
    {
        cout << "Enter name: " << endl;
        cin >> name;
        cout << "Enter id: " << endl;
        cin >> id;
        cout << "Enter age: " << endl;
        cin >> age;
    }

    void print_data()
    {
        cout << "Name: " << name << endl;
        cout << "Id: " << id << endl;
        cout << "Age: " << age << endl;
    }
};

class HashTable
{
public:
    HashTable();
    int insert_without_replacement(int key);
    int insert_with_replacement(int key);
    void print();
    int table[SIZE];
    void read_data_from_file()
    {
        // read the records from the file
        cout << "Reading data from the file. " << endl;
        Employee temp;
        fstream file("data.txt", ios::binary | ios::in);
        for (int i = 0; i < SIZE; i++)
        {
            file.read((char *)&temp, sizeof(Employee));
            cout << "Employee " << i + 1 << ": " << endl;
            temp.print_data();
            cout << endl;
        }
        // file.close();
    }

    void write_data_to_file(Employee hashed_employees[])
    {
        // Create a file in binary write mode.
        fstream file("data.txt", ios::binary | ios::out);

        // Write all the employee objects into the file in the order of the hash employee array.
        for (int i = 0; i < SIZE; i++)
        {
            cout << "Writing employee " << i + 1 << endl;
```

```cpp
            cout << "Employee " << i + 1 << ": " << endl;
            hashed_employees[i].print_data();
            file.write((char *)&hashed_employees[i], sizeof(Employee));
        }
        // Close the file.
        file.close();
    }

private:
    int hash(int key);
};

HashTable::HashTable()
{
    for (int i = 0; i < SIZE; i++)
        table[i] = -1;
}

int HashTable::hash(int key)
{
    return key % SIZE;
}

int HashTable::insert_without_replacement(int key)
{
    int index = hash(key);
    if (table[index] == -1)
        table[index] = key;
    else
    {
        int i = 1;
        while (table[(index + i) % SIZE] != -1)
        {
            i++;
            if (i == SIZE)
            {
                cout << "Table is full. " << endl;
                return 1;
            }
        }
        table[(index + i) % SIZE] = key;
    }
    return 0;
}

int HashTable::insert_with_replacement(int key)
{
    int index = hash(key);

    // there is no value there.
    if (table[index] == -1)
        table[index] = key;
    // the value that is already there, belongs there, then check, and then find another empty slot and insert there.
    else if (hash(table[index]) == index)
    {
        int i = 1;
        while (table[(index + i) % SIZE] != -1)
        {
```

```
            i++;
            if (i == SIZE)
            {
                cout << "Table is full. " << endl;
                return 1;
            }
        }
        table[(index + i) % SIZE] = key;
    }
    // the value that is already there, does not belong there, then replace it with the new value, and push the existing value down.
    else
    {
        // find empty slot
        int i = 1;
        while (table[(index + i) % SIZE] != -1)
        {
            i++;
            if (i == SIZE)
            {
                cout << "Table is full. " << endl;
                return 1;
            }
        }

        int temp = table[index]; // current value that doesnt belong there.
        table[index] = key;
        table[(index + i) % SIZE] = temp;
    }
    return 0;
}

void HashTable::print()
{
    for (int i = 0; i < SIZE; i++)
        cout << i << ":" << table[i] << " " << endl;
    cout << endl;
}

int main()
{
    int size;
    cout << "Enter size of table: " << endl;
    cin >> size;
    Employee employees[size];
    Employee hashed_employees[SIZE];
    HashTable hash_table;
    // Accept data for all employees
    for (int i = 0; i < size; i++)
    {
        cout << "Enter data for employee " << i + 1 << endl;
        employees[i].accept_data();
        cout << endl
             << endl;
        cout << "Employee " << i + 1 << ": " << endl;
        employees[i].print_data();
        cout << endl;
    }

```

```
    // Provide choice to users if they wanna insert with or without replacement
    int choice;
    cout << "Enter 1 to insert with replacement, 2 to insert without replacement:
    " << endl;
    cin >> choice;
    if (choice == 1)
    {
        // Insert data into hash table with replacement
        for (int i = 0; i < size; i++)
        {
            hash_table.insert_with_replacement(employees[i].id);
        }
    }
    else
    {
        // Insert data into hash table without replacement
        for (int i = 0; i < size; i++)
        {
            hash_table.insert_without_replacement(employees[i].id);
        }
    }

    cout << "Hash table now looks like this. " << endl;
    hash_table.print();

    cout << "Inserting data into the file. " << endl;

    // depending on the order of id in hash table, write the hashed_employee array
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (employees[j].id == hash_table.table[i])
            {
                hashed_employees[i].age = employees[j].age;
                hashed_employees[i].id = employees[j].id;
                hashed_employees[i].name = employees[j].name;
            }
        }
    }
    hash_table.write_data_to_file(hashed_employees);
    hash_table.read_data_from_file();

    return 0;
}
```

```
Enter size of table:
4
Enter data for employee 1
Enter name:
Krish
Enter id:
12
Enter age:
21


Employee 1:
Name: Krish
```

```
Id: 12
Age: 21

Enter data for employee 2
Enter name:
Part
Enter id:
42
Enter age:
22


Employee 2:
Name: Part
Id: 42
Age: 22

Enter data for employee 3
Enter name:
Ram
Enter id:
23
Enter age:
32


Employee 3:
Name: Ram
Id: 23
Age: 32

Enter data for employee 4
Enter name:
Ramesh
Enter id:
24
Enter age:
21


Employee 4:
Name: Ramesh
Id: 24
Age: 21

Enter 1 to insert with replacement, 2 to insert without replacement:
1
Hash table now looks like this.
0:-1
1:-1
2:12
3:23
4:24
5:42
6:-1
7:-1
8:-1
9:-1

```

```
Inserting data into the file.
Writing employee 1
Employee 1:
Name:
Id: 0
Age: 0
Writing employee 2
Employee 2:
Name:
Id: 0
Age: 0
Writing employee 3
Employee 3:
Name: Krish
Id: 12
Age: 21
Writing employee 4
Employee 4:
Name: Ram
Id: 23
Age: 32
Writing employee 5
Employee 5:
Name: Ramesh
Id: 24
Age: 21
Writing employee 6
Employee 6:
Name: Part
Id: 42
Age: 22
Writing employee 7
Employee 7:
Name:
Id: 0
Age: 0
Writing employee 8
Employee 8:
Name:
Id: 0
Age: 0
Writing employee 9
Employee 9:
Name:
Id: 0
Age: 0
Writing employee 10
Employee 10:
Name:
Id: 0
Age: 0
Reading data from the file.
Employee 1:
Name:
Id: 0
Age: 0

Employee 2:
Name:
```

```
Id: 0
Age: 0

Employee 3:
Name: Krish
Id: 12
Age: 21

Employee 4:
Name: Ram
Id: 23
Age: 32

Employee 5:
Name: Ramesh
Id: 24
Age: 21

Employee 6:
Name: Part
Id: 42
Age: 22

Employee 7:
Name:
Id: 0
Age: 0

Employee 8:
Name:
Id: 0
Age: 0

Employee 9:
Name:
Id: 0
Age: 0

Employee 10:
Name:
Id: 0
Age: 0

Enter size of table:
2
Enter data for employee 1
Enter name:
krish
Enter id:
124
Enter age:
21


Employee 1:
Name: krish
Id: 124
Age: 21

```

```
Enter data for employee 2
Enter name:
Tony
Enter id:
4
Enter age:
23


Employee 2:
Name: Tony
Id: 4
Age: 23

Enter 1 to insert with replacement, 2 to insert without replacement:
2
Hash table now looks like this.
0:-1
1:-1
2:-1
3:-1
4:124
5:4
6:-1
7:-1
8:-1
9:-1

Inserting data into the file.
Writing employee 1
Employee 1:
Name:
Id: 0
Age: 0
Writing employee 2
Employee 2:
Name:
Id: 0
Age: 0
Writing employee 3
Employee 3:
Name:
Id: 0
Age: 0
Writing employee 4
Employee 4:
Name:
Id: 0
Age: 0
Writing employee 5
Employee 5:
Name: krish
Id: 124
Age: 21
Writing employee 6
Employee 6:
Name: Tony
Id: 4
Age: 23
```

```
Writing employee 7
Employee 7:
Name:
Id: 0
Age: 0
Writing employee 8
Employee 8:
Name:
Id: 0
Age: 0
Writing employee 9
Employee 9:
Name:
Id: 0
Age: 0
Writing employee 10
Employee 10:
Name:
Id: 0
Age: 0
Reading data from the file.
Employee 1:
Name:
Id: 0
Age: 0

Employee 2:
Name:
Id: 0
Age: 0

Employee 3:
Name:
Id: 0
Age: 0

Employee 4:
Name:
Id: 0
Age: 0

Employee 5:
Name: krish
Id: 124
Age: 21

Employee 6:
Name: Tony
Id: 4
Age: 23

Employee 7:
Name:
Id: 0
Age: 0

Employee 8:
Name:
Id: 0
```

```
Age: 0

Employee 9:
Name:
Id: 0
Age: 0

Employee 10:
Name:
Id: 0
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" → slot 3*
   - *key "banana" → slot 1*
   - *key "cherry" → slot 3*
   - *key "date" → 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.