Hashing with Linear Probing

A Mini Project Report Submitted to

Jawaharlal Nehru Technological University Anantapur, Ananthapuramu

BACHELOR OF TECHNOLOGY
IN
INFORMATION TECHNOLOGY

Submitted by

Yekkaluri Susmitha	**21121A12B9**
Y.Upendra Yadav	**21121A12C0**
Mukthapuram Prasanthi	**21121A12C1**
Kondreddy Gari Archana	**22125A1204**
Maddilineni Sravani	**22125A1205**

Under the Supervision of

Ms. R. Suruthi Sutharsana, *M.E.*
Assistant Professor
Department of Information Technology

Department of Information Technology

SREE VIDYANIKETHAN ENGINEERING COLLEGE

(AUTONOMOUS)

(Affiliated to JNTUA, Ananthapuramu, Approved by AICTE, Accredited by NBA & NAAC)

Sree Sainath Nagar, Tirupati – 517 102, A.P., INDIA

2022-2023

SREE VIDYANIKETHAN ENGINEERING COLLEGE

(AUTONOMOUS)

(Affiliated to JNTUA, Ananthapuramu, Approved by AICTE, Accredited by NBA & NAAC)
Sree Sainath Nagar, Tirupati – 517 102, A.P., INDIA

Department of Information Technology

Certificate

This is to certify that, the Mini Project work entitled
"House Prices Prediction using Machine Learning"
is the bonafide work done by

Yekkaluri Susmitha	21121A12B9
YUpendra Yadav	21121A12C0
Mukthapuram Prasanthi	21121A12C1
Kodredy Gari Archana	22125A1204
Maddilineni Sravani	22125A1205

In the Department of **Information Technology**, **Sree Vidyanikethan Engineering College (Autonomous), Sree Sainath Nagar, Tirupati** and is submitted to **Jawaharlal Nehru Technological University Anantapur, Ananthapuramu** during the academic year 2022-2023.

Supervisor:

Ms. R. Suruthi Sutharsana, *M.E.*
Assistant Professor
Dept. of Information Technology
Sree Vidyanikethan Engineering College
Sree Sainath Nagar, Tirupati – 517 102

Head of the Dept.:

Dr. K. Ramani, *M.Tech., Ph.D.*
Professor
Dept. of Information Technology
Sree Vidyanikethan Engineering College
Sree Sainath Nagar, Tirupati – 517 102

SIGNATURE **SIGNATURE**

Table of content

Abstract:

We present the first exact analysis of a linear probing hashing its definitions and history of probing. From the generating function for the Robin Hood heuristic we obtain exact expressions for the cost of successful searches. For a full table, with the help of Singularity Analysis, we find the asymptotic expansion of this cost up to $O((BM)^{-1})$. We conclude with a new approach to study certain recurrences that involve truncated exponentials. A new family of numbers that satisfies a recurrence resembling that of the Bernoulli numbers is introduced. These numbers may prove helpful in studying recurrences involving truncated generating functions.

1. Definitions and a little bit of history

Since the invention of the telephone by German professor Johann Philipp Reis in 1861, yes that is right, it seems it was not Graham Bell, people need a way to keep an evidence of their phone number agenda. To do so, they would have need for a name and a number associate with it, i.e. a key and a value, called in modern terms a hash table.

After approximately a century, linear probing was invented in 1954 by Gene Amdahl, Elaine M. McGraw, and Arthur Samuel (and first analyzed in 1963 by Donald Knuth) , for resolving collisions in hash tables, and that happened long time before Dennis Ritchie, in 1972, at Bell Labs will invent the C language, that later on will be extended by Bjorn Stroustrup, still at Bell Labs, in 1985.

A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values, for example a phone book.

A hash collision is when two keys in a hash table share the same value, for example two persons share the same phone number. The probability of occurrence is given by the most common algorithms .With varying levels of collision risk, CRC-32, MD5, and SHA-1. Since hash collisions are inevitable, hash tables have mechanisms of dealing with them, known as collision resolutions. Two of the most common strategies are open addressing and separate chaining.

Methods	Description
clone()	Creates a shallow copy of this hash table.
compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
computeIfPresent(K key, BiFunction<? super K,? super V,?extends V>remappingFunction)	If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
contains(Object value)	Tests if some key maps into the specified value in this hash table.
containsKey(Object key)	Tests if the specified object is a key in this hash table.
containsValue(Object value)	Returns true if this hash table maps one or more keys to this value.
elements()	Returns an enumeration of the values in this hash table.
entrySet()	Returns a Set view of the mappings contained in this map.

equals(Object o)	Compares the specified Object with this Map for equality, as per the definition in the Map interface.
get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
hashCode()	Returns the hash code value for this Map as per the definition in the Map interface.
isEmpty()	Tests if this hash table maps no keys to values.
keys()	Returns an enumeration of the keys in this hash table.
keySet()	Returns a Set view of the keys contained in this map.
merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
put(K key, V value)	Maps the specified key to the specified value in this hash table.
putAll(Map<? extends K,? extends V> t)	Copies all of the mappings from the specified map to this hash table.
rehash()	Increases the capacity of and internally reorganizes this hash table, in order to accommodate and access its entries more efficiently.
remove?(Object key)	Removes the key (and its corresponding value) from this hash table.

size()	Returns the number of keys in this hash table.
toString()	Returns a string representation of this Hash table object in the form of a set of entries, enclosed in braces and separated by the ASCII characters “, ” (comma and space).
values()	Returns a Collection view of the values contained in this map.
clear()	Clears this hash table so that it contains no keys.

2. Linear Probing

In this article we are going to refer at the Linear Probing which together with Double Hashing and Quadratic Probing forms the open addressing strategy.

- **Core Idea:**

Cells in the hash table are assigned to one of the three states - occupied, empty, or deleted. If a hash collision occurs, the table will be probed to move the record to an alternate cell that is stated as empty.

- **Insertion in Hash Table with Linear Probing**

```
if <- hash(key)
loop
   if array[i] is empty then
       array[i] <- key
   else
       i <- (i + 1) mod size_of_array
end loop
```

- **Searching in Hash Table with Linear Probing**

```
i <- hash(key)loop
   if array[i] = key or array[i] is empty then
       return
   else
       i <- (i + 1) mod size_of_array
end loop
```

- **Removal in Hash Table with Linear Probing**

After an element is removed, records in same cluster with a higher index than the removed one has to be recalculated. Otherwise the empty element left after deletion will cause valid searches to fail.

```
i <- hash(key)
if array[i] = key then
   array[i] <- nil
for ( k<-i ; array[k] is not empty ; k++ )
   j <- hash(array[k])
   if i < j
       array[i] <-> array[j]
```

Time and Space Complexity:

For **Insertion operation**, the complexity analysis is as follows:

Best Case Time Complexity: O(1)

Worst Case Time Complexity: O(N). This happens when all elements have collided and we need to insert the last element by checking free space one by one.

Average Case Time Complexity: O(1) for good hash function; O(N) for bad hash function

Assuming the hash function uniformly distributes the elements, then the average case time complexity will be constant O(1). In the case where hash function work poorly, then the average case time complexity will degrade to O(N) time complexity.

Space Complexity: O(1) for Insertion operation

For **Deletion operation**, the complexity analysis is as follows:

- Best Case Time Complexity: O(1)
- Worst Case Time Complexity: O(N)
- Average Case Time Complexity: O(1) for good hash function; O(N) for bad hash function
- Space Complexity: O(1) for deletion operation

The ideas are similar to insertion operation.

Similarly for Search operation, the complexity analysis is as follows:

- Best Case Time Complexity: O(1)
- Worst Case Time Complexity: O(N)

- Average Case Time Complexity: O(1) for good hash function; O(N) for bad hash function
- Space Complexity: O(1) for search operation

3. A way of implementation

We can start by implementing a solution for a phone book case study.   In the first step we create a Phone Record class that stores information about the person name and the telephone number. These two can be associated with the key and value from our definitions. After that, we can declare as many individual persons, or objects, as we know but there will be no logic in addressing a person, each object is independent and does not affect the other ones as we might add for ex the "Per's 1" with two telephone numbers, which means the key will not be unique and that will result in a wrong addressing.

With the open addressing technique, a hash collision is resolved by probing, or searching through alternative locations in the array (the probe sequence) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.

The principle of this idea is to look at the hash Key implementation which should compute an index for the given key. Since our domain is a string and co domain is an int we must cast the string to int. The simplest way is to sum up each char of the string. But this will not solve our problem. For example "Per's 1" and "1 per’s" are made up by the same characters, and simply sum up the corresponding integer values will result in having the same value.

What we need to do next is to implement a computation for *hashing strings*, meaning to convert the string key into a unique number that is different regardless combinations of characters that compose it. And here are where collisions appear, given two different strings, the result of the hash key might be the same in some exceptional cases.

The pseudo code for this implementation is given by the next function:

```
hash(String x, int a, int p)
    h <- INITIAL_VALUE
    for ( i <- 0 ; i < x.length ; i <-i+1 )
        h <- ((h*a) + x[i]) mod p
    return h
```

where
* x represents our key
* INITIAL_VALUE represents a random number from a universal family mapping integer domain
* a represents the prime number multiplication
* p represents the dimension of our array

Step by step example:

phone book index	key	value	step 1: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0							
1				2	2	Pers 1	Tel 1
2							

phone book index	key	value	step 2: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0							
1				0	0	Pers 2	Tel 2
2	Pers 1	Tel 1					

phone book index	key	value	step 3: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 2	Tel 2					
1				1	1	Pers 3	Tel 3
2	Pers 1	Tel 1					

phone book index	key	value	step 4: remove	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 2	Tel 2					
1	Pers 3	Tel 3		0	0	Pers 2	
2	Pers 1	Tel 1					

phone book index	key	value	step 5: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0							
1	Pers 3	Tel 3		0	2	Pers 4	Tel 4
2	Pers 1	Tel 1					

phone book index	key	value	step 6: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 4	←				
1	Pers 3	Tel 3		0	2	Pers 4	Tel 2
2	Pers 1	Tel 1					

phone book index	key	value	step 7: lookup	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 2	←				
1	Pers 3	Tel 3		0	0	Pers 2	
2	Pers 1	Tel 1					

phone book index	key	value	step 8: remove	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 2					
1	Pers 3	Tel 3		2	2	Pers 1	
2	Pers 1	Tel 1	→				

phone book index	key	value	step 9: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 2					
1	Pers 3	Tel 3		2	0	Pers 5	Tel 1
2			←				

phone book index	key	value	step 10: remove	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 2					
1	Pers 3	Tel 3		2	2	Pers 1	
2	Pers 5	Tel 1	→				

phone book index	key	value	step 11: remove	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 2					
1	Pers 3	Tel 3	→	1	1	Pers 3	
2	Pers 5	Tel 1					

phone book index	key	value	step 12: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 2					
1			←	1	1	Pers 6	Tel 6
2	Pers 5	Tel 1					

phone book index	key	value	step 13: set	genereated index by the ***find_slot*** function	genereated index by the ***hashKey*** function	key	value
0	Pers 4	Tel 2					
1	Pers 6	Tel 6		2	2	Pers 7	Tel 7
2	Pers 5	Tel 1					

At the final step "Per's 7" does not have an empty slot, so collision appears with the "Per's 5" which will share the same cell.

In this situation, there are two possibilities:

1. replace the existing key with the new oneincrease dimension of the table.
2. usually this is recommended to be done when table is 80% full of capacity, and reallocate all the existing keys to the new table.

4. Comparison with other probing methods

Linear probing - the interval between probes is fixed — often set to 1.

Quadratic probing - the interval between probes increases quadratic ally (hence, the indices are described by a quadratic function).

Double hashing - the interval between probes is fixed for each record but is computed by another hash function.

Some open addressing methods, such as Hopscotch hashing, Robin Hood hashing, last-come-first-served hashing and cuckoo hashing move existing keys around in the array to make room for the new key. This gives better maximum search times than the methods based on probing.

IMPLEMENTATION

Source Code:

```
import java.io.*;
import java.util.*;
// Importing Scanner class as in do-while
// inputs are entered at run-time when
// menu is popped to user to perform desired action
import java.util.Scanner;

// Helper class - LinearProbingHashTable
class LinearProbingHashTable {
    // Member variables of this class
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    // Constructor of this class
    public LinearProbingHashTable(int capacity)
    {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    // Method 1
    // Function to clear hash table
    public void makeEmpty()
```

```
{
    currentSize = 0;
    keys = new String[maxSize];
    vals = new String[maxSize];
}

// Method 2
// Function to get size of hash table
public int getSize() { return currentSize; }

// Method 3
// Function to check if hash table is full
public boolean isFull()
{
    return currentSize == maxSize;
}

// Method 4
// Function to check if hash table is empty
public boolean isEmpty() { return getSize() == 0; }

// Method 5
// Function to check if hash table contains a key
public boolean contains(String key)
{
    return get(key) != null;
}
```

```
// Method 6
// Function to get hash code of a given key
private int hash(String key)
{
    return key.hashCode() % maxSize;
}

// Method 7
// Function to insert key-value pair
public void insert(String key, String val)
{
    int tmp = hash(key);
    int i = tmp;

    // Do-while loop
    // Do part for performing actions
    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }

        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
        i = (i + 1) % maxSize;

    }

    // Do-while loop
    // while part for condition check
    while (i != tmp);
}

// Method 8
// Function to get value for a given key
public String get(String key)
{
    int i = hash(key);
    while (keys[i] != null) {
        if (keys[i].equals(key))
            return vals[i];
        i = (i + 1) % maxSize;
    }
    return null;
}

// Method 9
// Function to remove key and its value
public void remove(String key)
{
    if (!contains(key))
```

```
        return;

    // Find position key and delete
    int i = hash(key);
    while (!key.equals(keys[i]))
        i = (i + 1) % maxSize;
    keys[i] = vals[i] = null;

    // rehash all keys
    for (i = (i + 1) % maxSize; keys[i] != null;
         i = (i + 1) % maxSize) {
        String tmp1 = keys[i], tmp2 = vals[i];
        keys[i] = vals[i] = null;
        currentSize--;
        insert(tmp1, tmp2);
    }
    currentSize--;
}

// Method 10
// Function to print HashTable
public void printHashTable()
{
    System.out.println("\nHash Table: ");
    for (int i = 0; i < maxSize; i++)
        if (keys[i] != null)
            System.out.println(keys[i] + " " + vals[i]);
    System.out.println();
    }
}

// Main testing class
// Main Class for LinearProbingHashTableTest
public class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Creating a scanner object
        // to take input from user
        Scanner scan = new Scanner(System.in);

        // Display messages
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");

        // maxSizeake object of LinearProbingHashTable
        LinearProbingHashTable lpht
            = new LinearProbingHashTable(scan.nextInt());

        char ch;

        // Do-while loop
        // Do part for performing actions
        do
        // Menu is displayed
        // LinearProbingHashTable operations performed as
// per keys Users enter 'y' to continue 'n' if
// entered by user , the program terminates

{
    // Menu
    // Display messages
    System.out.println("\nHash Table Operations\n");
    System.out.println("1. insert ");
    System.out.println("2. remove");
    System.out.println("3. get");
    System.out.println("4. clear");
    System.out.println("5. size");

    // Reading integer using nextInt()
    int choice = scan.nextInt();

    // Switch case
    switch (choice) {

    // Case 1
    case 1:

        // Display message
        System.out.println("Enter key and value");
        lpht.insert(scan.next(), scan.next());
        // Break statement to terminate a case
        break;
```

```
// Case 2
case 2:

    // Display message
    System.out.println("Enter key");
    lpht.remove(scan.next());
    // Break statement to terminate a case
    break;

// Case 3
case 3:

    // Print statements
    System.out.println("Enter key");
    System.out.println("Value = "
                    + lpht.get(scan.next()));
    // Break statement to terminate a case
    break;

// Case 4
case 4:

    lpht.makeEmpty();
    // Print statement
    System.out.println("Hash Table Cleared\n");
    // Break statement to terminate a case
    break;
```

```
// Case 5
case 5:


    // Print statement
    System.out.println("Size = "
                   + lpht.getSize());
    break;


// Default case
// Executed when mentioned switch cases are not
// matched
default:
    // Print statement
    System.out.println("Wrong Entry \n ");
    // Break statement
    break;
}


// Display hash table
lpht.printHashTable();


// Display message asking the user whether
// he/she wants to continue
System.out.println(
    "\nDo you want to continue (Type y or n) \n");


// Reading character using charAt() method to
// fetch
```

```
            ch = scan.next().charAt(0);
        } while (ch == 'Y' || ch == 'y');
    }
}
```

Output

```
Hash Table Test

Enter size

5

Hash Table Operations

1. insert

2. remove

3. get

4. clear

5. size

1

Enter key and value

Na sodium

Hash Table:

Na sodium

Do you want to continue (Type y or n)

y

Hash Table Operations

1. insert

2. remove

3. get

4. clear

5. size

1

Enter key and value

K potassium

Hash Table:

Na sodium
```

```
K potassium
Do you want to continue (Type y or n)
 y
 Hash Table Operations
 1. insert
2. remove
3. get
4. clear
5. size
1
Enter key and value
Fe iron
 Hash Table:
Na sodium
K potassium
Fe iron
 Do you want to continue (Type y or n)
y
 Hash Table Operations
1. insert
2. remove
3. get
4. clear
5. size
1
Enter key and value
H hydrogen
Hash Table:
```

```
Na sodium
K potassium
Fe iron
H hydrogen
 Do you want to continue (Type y or n)
y
 Hash Table Operations
 1. insert
2. remove
3. get
4. clear
5. size
1
Enter key and value
He helium
 Hash Table:
Na sodium
K potassium
Fe iron
H hydrogen
He helium
 Do you want to continue (Type y or n)
y
 Hash Table Operations
1. insert
2. remove
3. get
4. clear
```

```
5. size
1
Enter key and value
Ag silver
Hash Table:
Na sodium
K potassium
Fe iron
H hydrogen
He helium
Do you want to continue (Type y or n)
 y
 Hash Table Operations
 1. insert
2. remove
3. get
4. clear
5. size
3
Enter key
Fe
Value = iron
 Hash Table:
Na sodium
K potassium
Fe iron
H hydrogen
He helium
```

```
Do you want to continue (Type y or n)
y
 Hash Table Operations
 1. insert
2. remove
3. get
4. clear
5. size
2
Enter key
H
 Hash Table:
Na sodium
K potassium
Fe iron
He helium
 Do you want to continue (Type y or n)
y
 Hash Table Operations
1. insert
2. remove
3. get
4. clear
5. size
3
Enter key
H
Value = null
```

```
Hash Table:
Na sodium
K potassium
Fe iron
He helium
 Do you want to continue (Type y or n)
y
 Hash Table Operations
 1. insert
2. remove
3. get
4. clear
5. size
1
Enter key and value
Au gold
 Hash Table:
Na sodium
K potassium
Fe iron
He helium
Au gold
 Do you want to continue (Type y or n)
y
 Hash Table Operations
 1. insert
2. remove
3. get
```

```
4. clear
5. size
5
Size = 5
 Hash Table:
Na sodium
K potassium
Fe iron
He helium
Au gold
 Do you want to continue (Type y or n)
y
 Hash Table Operations
 1. insert
2. remove
3. get
4. clear
5. size
4
Hash Table Cleared
 Hash Table:
 Do you want to continue (Type y or n)
 y
Hash Table Operations
1. insert
2. remove
3. get
4. clear
```

```
5. size
5
Size = 0
 Hash Table:
Do you want to continue (Type y or n)
 n
```

CONCLUSION