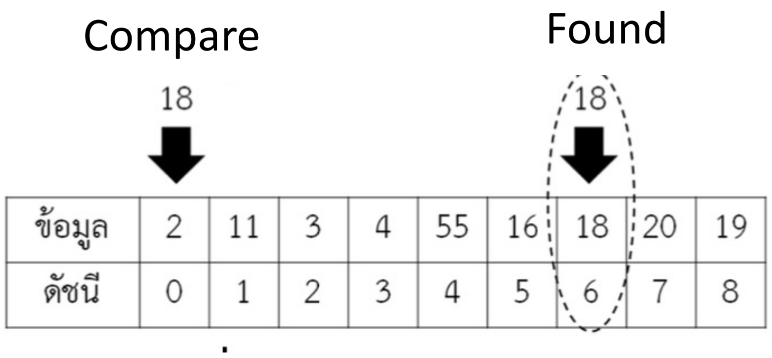
Hash Table

Outline

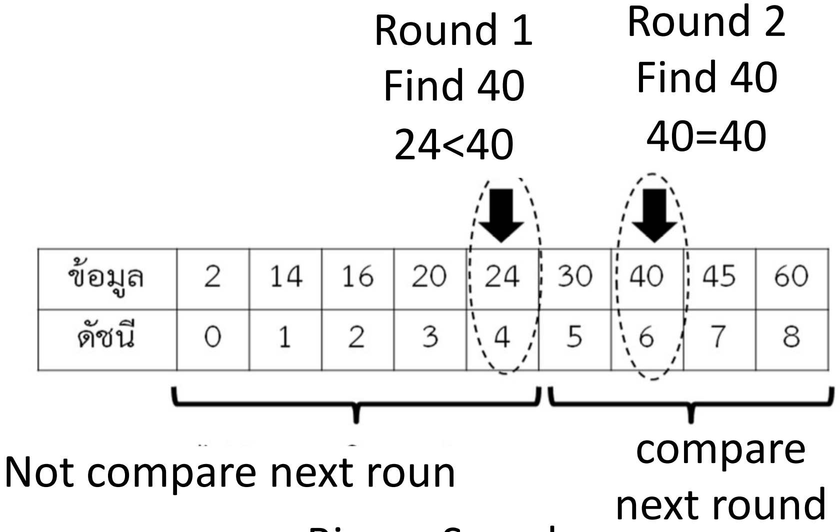
- Searching
- Hash Table
- Hash Function
- Example

Searhing

- There are 2 types to search data in data structure
    - Linear Search: search data in linear data structure
        - Sequential Search
        - Binary Search
        - Index Sequential Search
        - Hash Search



Sequential Search



Binary Search

Sequential Search

Compare

18

Found

18

ข้อมูล	2	11	3	4	55	16	18	20	19
ดัชนี	0	1	2	3	4	5	6	7	8

Binary Search

Round 1
Find 40
24<40
at $\left(\frac{8+0}{2}\right) = 4$

Round 2
Find 40
40=40
at $\left(\frac{5+8}{2}\right) = 6$

ข้อมูล	2	14	16	20	24	30	40	45	60
ดัชนี	0	1	2	3	4	5	6	7	8

Not compare next round

compare next round

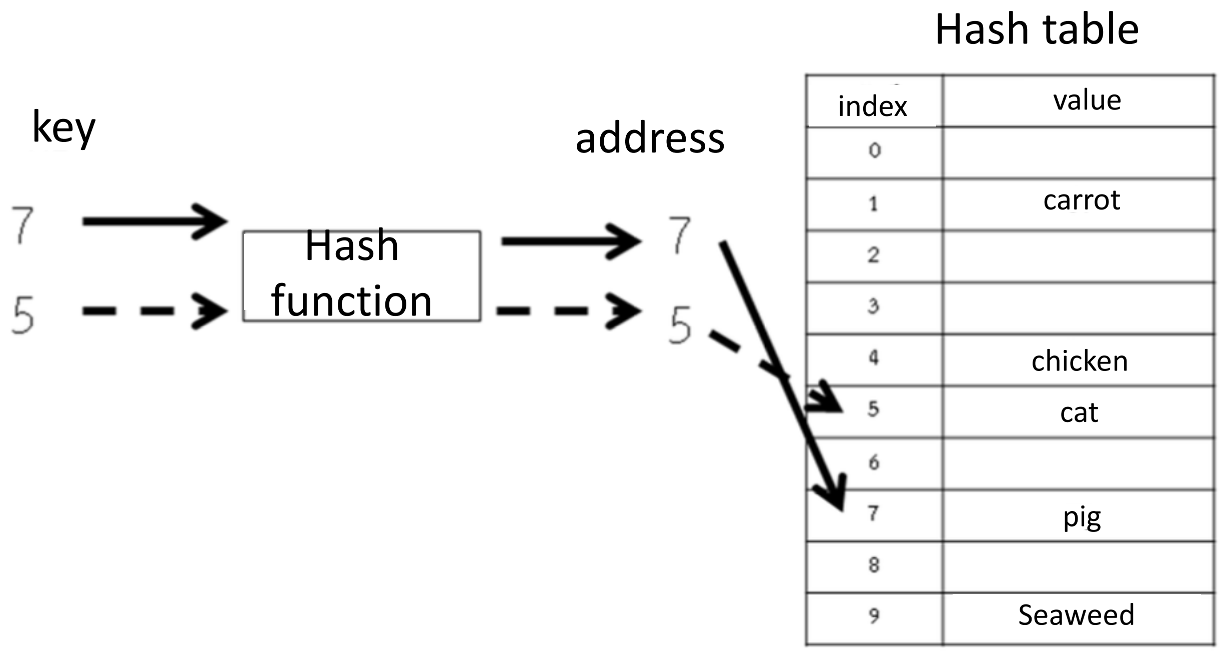
Index Sequential Search

Find Persian
cat
Search in Topic
"Cat"
Page 1-3

Page	Topic
1-3	Cat
4-50	Pig
51-60	Fish

Page	Topic
1	Siamese Cat
2	Persian Cat
3	British Shorthair
4	Pig
...	...
50	Boar
51	Shark
...	...
60	Catfish

Hash Search



Direct Hash

Hash Table

- A **hash table**, also known as **hash map**, is a data structure that implements an associative array or dictionary. It is an abstract data type that maps keys to values.
- A hash table uses a hash function to compute an *index*, also called a *hash code*, into an array of *buckets* or *slots*, from which the desired value can be found.
- During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

Key	index	0	1	2	3	4	5	6	7	8	9
Value	data		cat			dog	ant		pig		fish

Hash Table

Hashing

- Each key is mapped into some number in the range 0 to Table size-1 and placed in the appropriate cell.
- This mapping is called hash function.
- The only remaining problems deal with choosing a function, deciding what to do when two keys hash to the same value (collision), and deciding on the table size.

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

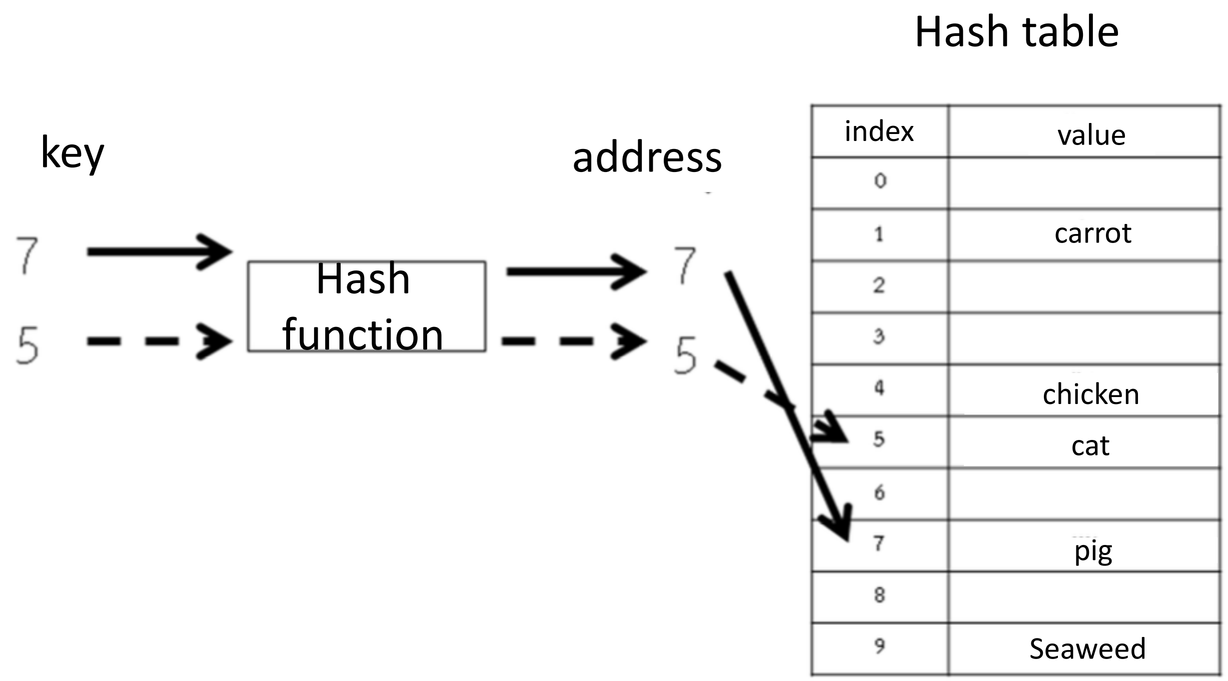
An ideal hash table

Hash Function

- Hash function
    - Direct Hashing
    - Subtract Hashing
    - Digit-Extraction Hashing
    - Mid Square Hashing
    - Fold Shift Hashing
    - Fold Boundary Hashing
    - Modulo-Division Hashing

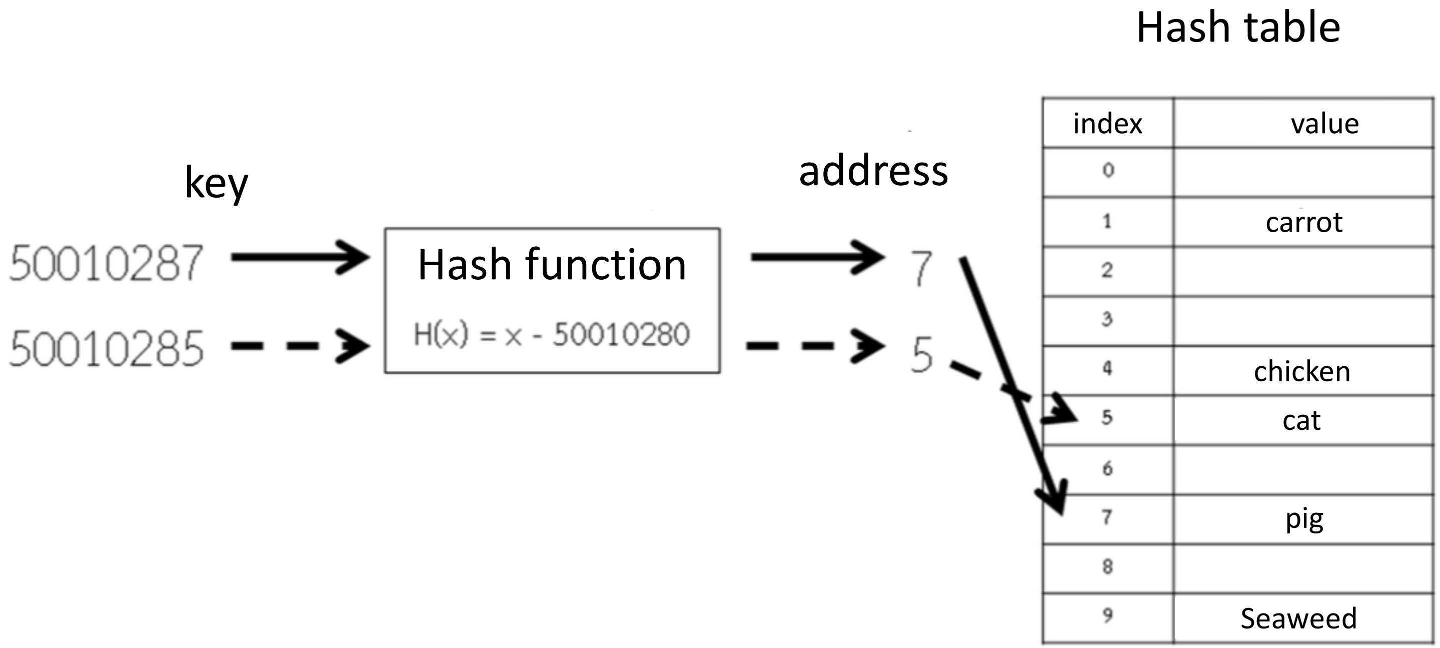
Direct hashing

- Direct Hashing
  - Key is at the same address of hash table.



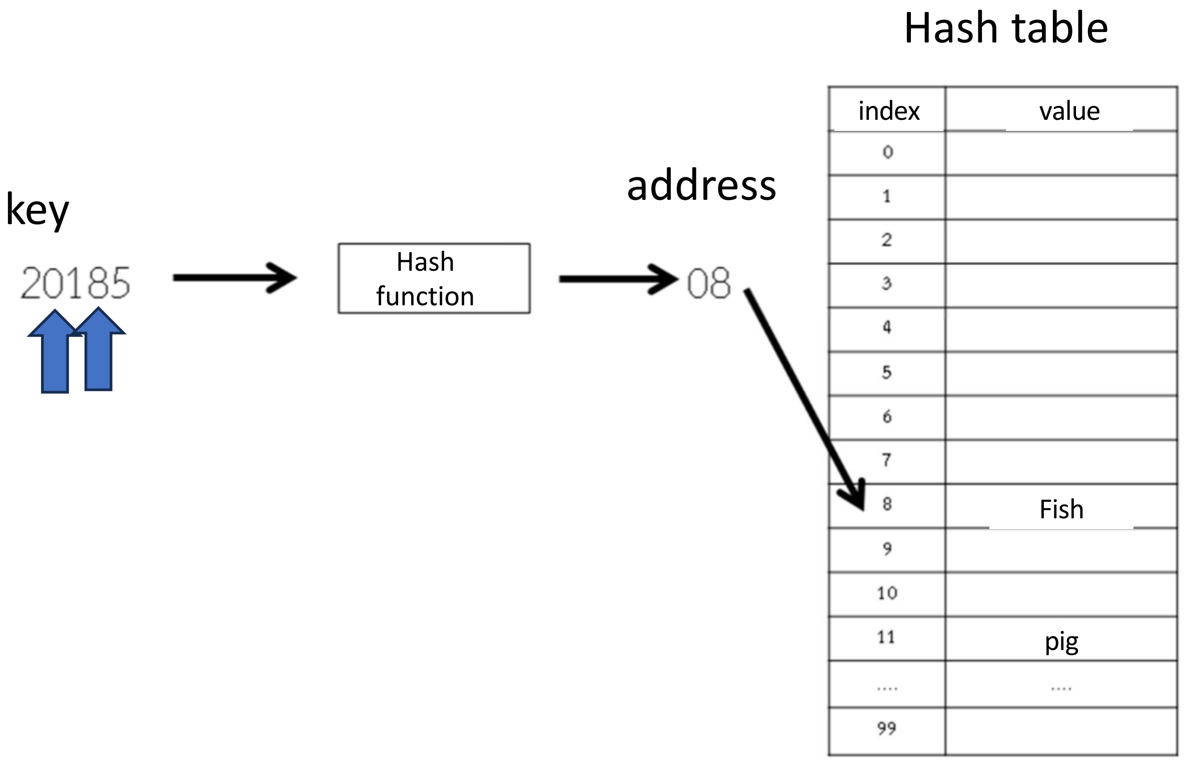
Subtraction hashing

- Subtraction Hashing
  - It is similar to direct hashing.
  - However, key is subtracted by constant value.



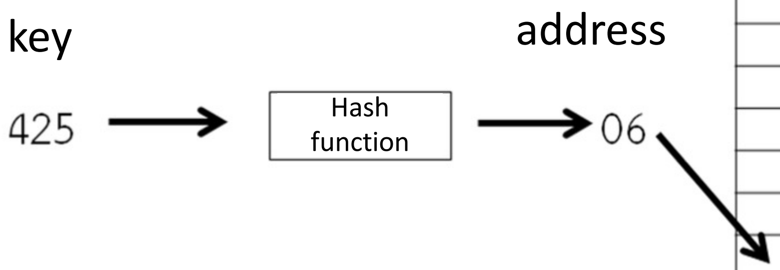
Digit-Extraction hashing

- Digit-Extraction Hashing
  - Choose some position from key to use as address of hash table.
  - Choose the position which is not duplicate with another to reduce the collision.



Mid Square hashing

- Mid Square Hashing
  - Calculate address from $(key)^2$
  - Then, select middle position of the result as address of hash table.



Hash table

index	value
0	
1	
2	
3	
4	
5	
6	pig
7	
8	
9	
10	Fish
11	
....	....
99	

value	key	$(key)^2$	Result of $(key)^2$	index
pig	425	425^2	180625	06
chick	124	124^2	015376	53
Fish	145	145^2	021025	10

รูปที่ 7.10 วิธีการทำแฮชโดยค่ากลางยกกำลังสอง

Fold Shift Hashing

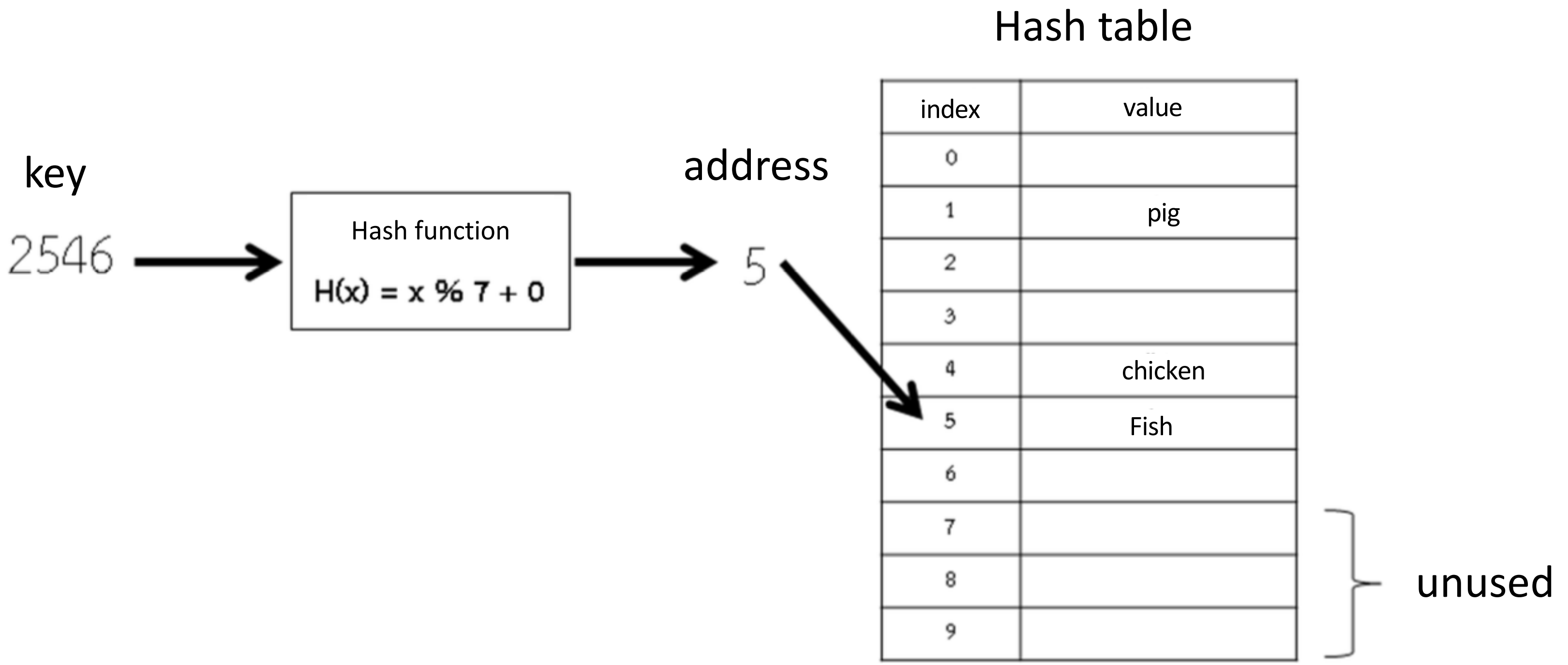
- Fold Shift Hashing
    - Divides key into 3 parts (a,b,c)
    - Then, sum those parts (a+b+c)
    - Calculates (a+b+c) Mod M, where M is the table size.

Fold Boundary Hashing

- Fold Shift Hashing
  - Divides key into 3 parts (a,b,c)
  - Reverses left and right keys
  - Then, sum those parts (a+b+c)
  - Calculates (a+b+c) Mod M, where M is the table size.

Modulo-Division Hashing

- Modulo Division Hashing
  - Calculates address using key(x) mod table size(N).
  - H(x) = (x%N)

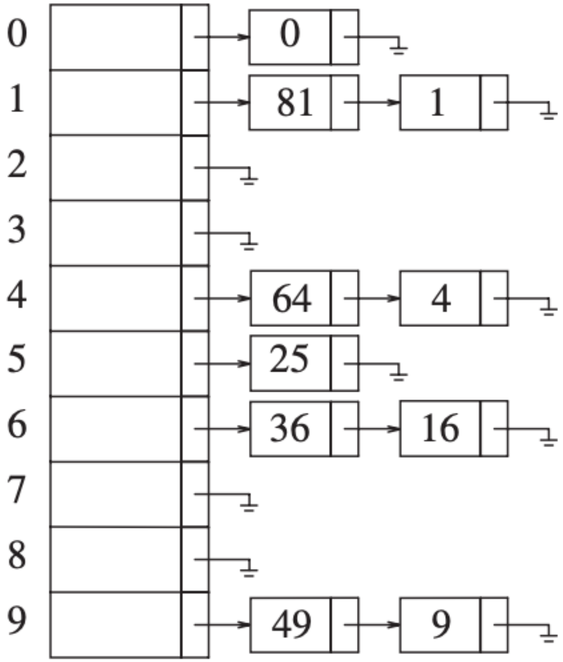


Collision

- Collision
    - Hash collision is when 2 pieces of data in a hash table share the same value
    - for example
        - 17%7 = 3 and 24%7 = 3
    - There are 2 solutions for hash collision
        - Separate Chaining
        - Open Addressing

Separate Chaining

- Separate Chaining or Open Hashing is the strategy allows more than one records to be chained in hash value.
- Linked List can be used to solve this strategy.



Open Addressing

- Open Addressing or Closed Hashing is the strategy which uses 2 hash functions.
- Key are calculated with the first function.
  - If it can insert into hash table, insert it to hash table.
  - If there is collision, calculate the second function until it can be inserted to hash table.
- There are 3 types of function
  - Linear Probing function
  - Quadratic Probing function
  - Double hashing function

Linear Probing function

- When there is collision, the function will be increased until there is empty address.
- First hashing function: $H(x) = x\%N$
- Second hashing function: $h_i(x) = (x+i)\%N$
- While i is number of collision.

Linear Probing function

- Example: data-> 156, 85, 42, 54, 189, 125, 34, 99, 112
- 156%13 = 0
- 85%13 = 7
- 42%13 = 3
- 54%13 = 2
- 189%13 = 7 collision!!!
    - (189+1)%13 = 8

Address	Data
0	156
1	
2	54
3	42
4	
5	
6	
7	85
8	189
9	
10	
11	
12	

Linear Probing function

- Example: data-> 156, 85, 42, 54, 189, 125, 34, 99, 112
- 125%13 = 8 collision
  - (125+1)%13 = 9
- 34%13 = 8 collision
  - (34+1)%13 = 10
- 99%13 = 8 collision
  - (99+1)%13 = 9 collision
  - (99+2)%13 = 10 collision
  - (99+3)%13 = 11
- 112%13 = 8 collision
  - (112+1)%13 = 9 collision
  - (112+2)%13 = 10 collision
  - (112+3)%13 = 11 collision
  - (112+4)%13 = 12

Address	Data
0	156
1	
2	54
3	42
4	
5	
6	
7	85
8	189
9	125
10	34
11	99
12	112

Linear Probing function

- We use Lazy deletion solution to delete data by set the symbol at the address which is deleted.

Lazy deletion

Address	Data
0	156
1	
2	54
3	42
4	
5	
6	
7	85
8	189
9	-
10	34
11	99
12	112

Quadratic Probing function

- When there is collision, the function will be increased by i^2 until there is empty address.
- First hashing function: $H(x) = x\%N$
- Second hashing function: $h_i(x) = (x+i^2)\%N$
- While i is number of collision.

Quadratic Probing function

- Example: data-> 85, 55, 42, 96, 132, 140
- 85%11 = 8
- 55%11 = 0
- 42%11 = 9
- 96%11 = 8 collision!!!
  - $(96+1^2)\%11 = 9$ collision!!!
  - $(96+2^2)\%11 = 1$
- 132%11 = 0 collision!!!
  - $(132+1^2)\%11 = 1$ collision!!!
  - $(132+2^2)\%11 = 4$

Address	Data
0	55
1	96
2	
3	
4	132
5	
6	
7	
8	85
9	42
10	

Quadratic Probing function

- Example: data-> 85, 55, 42, 96, 132, 140
- 140%11 = 8 collision!!!
  - $(140+1^2)$ % 11 = 9 collision!!!
  - $(140+2^2)$ % 11 = 1 collision!!!
  - $(140+3^2)$ % 11 = 6

Address	Data
0	55
1	96
2	
3	
4	132
5	
6	140
7	
8	85
9	42
10	

Double Hashing function

- When there is collision, this function contains 2 hash functions as follows
  - First function: $H(x) = x\%N$
  - Second function: $f(x) = (R – (x\%R))$ and $h_i(x) = (x + i*f(x)) \% N$

Double Hashing function

- Example: data-> 75, 152, 171, 38, 211, 18, 189 [R=7]
- 75%11 = 9
- 152%11 = 9 collision!!!
  - f(152) = 7 - (152%7) = 2
  - h_1(152) = (152 + (1*2)) %11 = 0
- 171 % 11 = 6
- 38 % 11 = 5
- 211 % 11 = 2
- 18 % 11 = 7

Address	Data
0	152
1	
2	211
3	
4	
5	38
6	171
7	18
8	
9	75
10	

Double Hashing function

- Example: data-> 75, 152, 171, 38, 211, 18, 189
- 189 % 11 = 2 collision!!!
  - f(189) = 7 - (189%7) = 7
  - $h_1(189) = (189 + (1*7)) \% 11 = 9$ collision!!!
  - $h_2(189) = (189 + (2*7)) \% 11 = 5$ collision!!!
  - $h_3(189) = (189 + (3*7)) \% 11 = 1$

Address	Data
0	152
1	189
2	211
3	
4	
5	38
6	171
7	18
8	
9	75
10	

Rehashing

- Rehashing is the process of increasing the size of a hashmap and redistributing the elements to new buckets based on their new hash values.
- It is done to improve the performance of the hashmap and to prevent collisions caused by a high load factor.

Rehashing

- Example: data-> 4371, 1323, 6173, 4199, 4344, 9679,1989 [R=7]
- 4371%9 = 6
- 1323%9 = 0
- 6173%9 = 8
- 4199%9 = 5
- 4344%9 = 6 collision!!!
  - f(4344 = 7-(4344 %7) = 3
  - h1(4344 = (4344+(1*3)) %9 = 0 collision!!!
  - h2(4344 = (4344+(2*3)) %9 = 3

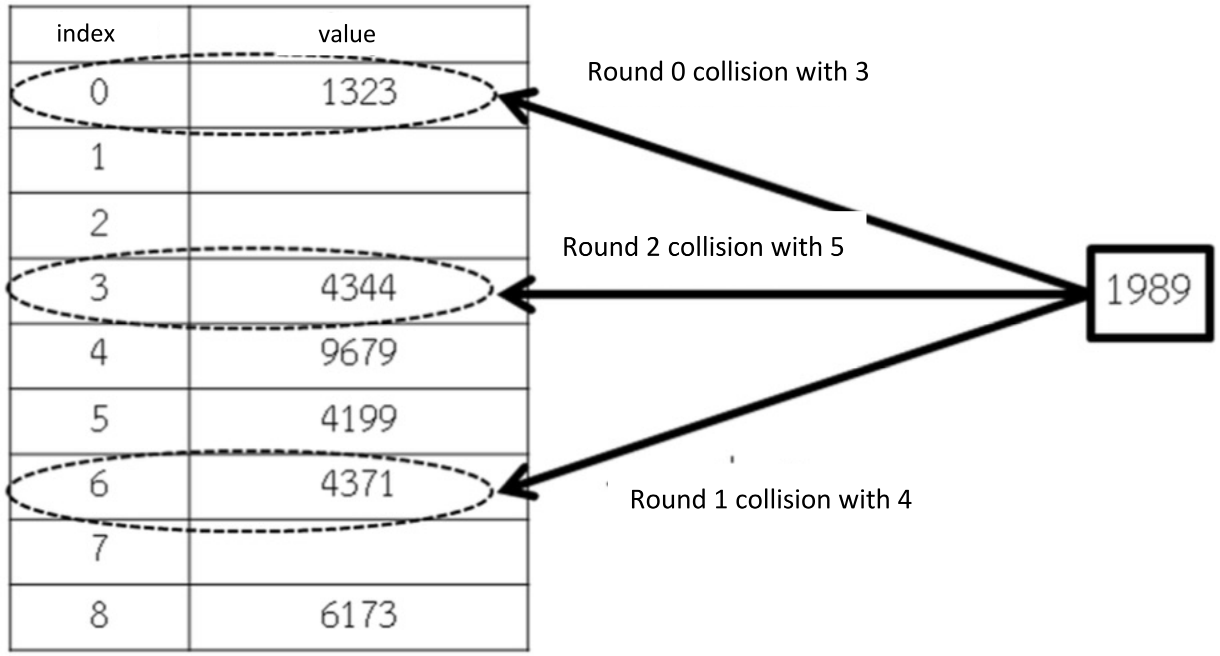
Address	Data
0	1323
1	
2	
3	4344
4	
5	4199
6	4371
7	
8	6173

Rehashing

- Example: data-> 4371, 1323, 6173, 4199, 4344, 9679,1989 [R=7]
- 9679%9=4
- 1989%9=0 collision
  - f(1989) = 7-(1989%7) = 6
  - h1(1989) = (1989 + (1*6)) %9 = 6 collision!!!
  - h2(1989) = (1989 + (2*6)) %9 = 3 collision!!!
  - h3(1989) = (1989 + (3*6)) %9 = 0 collision!!!
  - h4(1989) = (1989 + (4*6)) %9 = 6 collision!!!
  - h5(1989) = (1989 + (5*6)) %9 = 3 collision!!!
  - …

Address	Data
0	1323
1	
2	
3	4344
4	9679
5	4199
6	4371
7	
8	6173

Rehashing



index
value
0
1323
1
2
3
4344
4
9679
5
4199
6
4371
7
8
6173
Round 0 collision with 3
Round 2 collision with 5
Round 1 collision with 4
1989

Rehashing – increase Table size to 19

- Example: data-> 4371, 1323, 6173, 4199, 4344, 9679,1989 [R=13]
- 4371 % 19 = 1
- 1323 % 19 = 12
- 6173 % 19 = 17
- 4199 % 19 = 0
- 4344 % 19 = 12 collision!!!
  - f(4344) = 19 – (4344 % 13) = 17
  - h1(4344) = (4344 + (1*17)) % 19 = 10

Address	Data
0	4199
1	4371
2	
3	
4	
5	
6	
7	
8	
9	
10	4344
11	
12	1323
13	
14	
15	
16	
17	6173
18	

Rehashing – increase Table size to 19

- Example: data-> 4371, 1323, 6173, 4199, 4344, 9679,1989 [R=13]
- 9679 % 19 = 8
- 1989 % 19 = 13

Address	Data
0	4199
1	4371
2	
3	
4	
5	
6	
7	
8	9679
9	
10	4344
11	
12	1323
13	1989
14	
15	
16	
17	6173
18	

```
#include<bits/stdc++.h>
using namespace std;

class open_addressing
{
    public:
    int      hash_key[100];
    string   hash_table[100];
    int      n  = 0;
    int      r  = 0;

    open_addressing(int p_n, int p_r)
    {
        n = p_n;
        r = p_r;
        for(int i=0 ; i < n ; i++)
        {
            hash_key[i] = -1;
            hash_table[i] = "-";
        }
    }
    void add_modulo_division (int key, string data)
    {
        hash_key[key%n]      = key;
        hash_table[key%n]    = data;
    }
    string search_modulo_division (int key)
    {
        if( key == hash_key[key%n] )
        {
            return hash_table[key%n];
        }
        else
        {
            return "-";
        }
    }
    void add_linear_probing (int key, string data)
    {
        for(int i=0 ; i < n ; i++)
        {
            int j = (key + i) % n;
            if( hash_key[j] == -1 )
            {
                hash_key[j]    = key;
                hash_table[j]     = data;
                break;
            }
        }
    }
```

```
string search_linear_probing (int key)
{
    for(int i=0 ; i < n ; i++)
    {
        int j = (key + i) % n;
        if( hash_key[j] == key )
        {
            return hash_table[j];
        }
        if( hash_key[j] == -1 )
        {
            return "-";
        }
    }
}
void add_quadratic_probing (int key,string data)
{
    for(int i=0 ; i < n ; i++)
    {
        int j = (key + (i*i)) % n;
        if( hash_key[j] == -1 )
        {
            hash_key[j]    = key;
            hash_table[j]     = data;
            break;
        }
    }
}
```

```
string search_quadratic_probing (int key)
{
    for(int i=0 ; i < n ; i++)
    {
        int j = (key + (i*i)) % n;
        if( hash_key[j] == key )
        {
            return hash_table[j];
        }
        if( hash_key[j] == -1 )
        {
            return "-";
        }
    }
}
void add_double_hashing (int key,string data)
{
    int hash2 = r - (key % r);
    for(int i=0 ; i < n ; i++)
    {
        int j = (key + (i*hash2)) % n;
        if( hash_key[j] == -1 )
        {
            hash_key[j]    = key;
            hash_table[j]     = data;
            break;
        }
    }
}
```

```
    string search_double_hashing (int key)
    {
        int hash2 = r - (key % r);
        for(int i=0 ; i < n ; i++)
        {
            int j = (key + (i*hash2)) % n;
            if( hash_key[j] == key )
            {
                return hash_table[j];
            }
            if( hash_key[j] == -1 )
            {
                return "-";
            }
        }
    }
    void print()
    {
        for(int i=0 ; i < n ; i++)
        {
            cout<<"("<<hash_key[i]<<","<<hash_table[i]<<") ";
        }
        cout<<endl;
    }
};
```

```
class node
{
    public:
    int key;
    string data;
    node *next;
    node()
    {
        key = -1;
        data = "-";
        next = NULL;
    }
    node(string s, int k)
    {
        key = k;
        data = s;
        next = NULL;
    }
};
class separate_chaining
{
    public:
    int n;
    node hash_table[100];
    separate_chaining(int p_n)
    {
        n = p_n;
    }
```

```
    void add(int key, string data)
    {
        int j  = key % n;
        if( hash_table[j].next == NULL )
        {
            hash_table[j].data  = data;
            hash_table[j].key   = key;
            hash_table[j].next  = new node();
        }
        else
        {
            node *t_node = hash_table[j].next;
            while(t_node->next != NULL)
            {
                t_node = t_node->next;
            }
            t_node->data    = data;
            t_node->key = key;
            t_node->next    = new node();
        }
    }
    string search(int key)
    {
        int j  = key%n;
        if( hash_table[j].key == key )
        {
            return hash_table[j].data;
        }
        else
        {
            node *t_node = hash_table[j].next;
            while(t_node->next != NULL)
            {
                if( key == t_node->key )
                {
                    return t_node->data;
                }
                t_node = t_node->next;
            }
        }
        return "-";
    }
    void print()
    {
        for(int i=0 ; i < n ; i++)
        {
            cout<<"("<<hash_table[i].data<<","<<hash_table[i].key<<") ";
            if( hash_table[i].next != NULL )
            {
                node *t_node = hash_table[i].next;
                while( t_node->next != NULL )
                {
                    cout<<"("<<t_node->data<<","<<t_node->key<<") ";
                    t_node = t_node->next;
                }
            }
            cout<<"|";
        }
```

```
        cout<<endl;
    }
};
int main()
{
    open_addressing h1(13,7);
    h1.add_modulo_division(62,"cat");
    h1.add_modulo_division(77,"bird");
    h1.add_modulo_division(56,"ant");
    h1.add_modulo_division(55,"dog");
    h1.add_modulo_division(132,"fish");
    h1.print();
    cout<<h1.search_modulo_division(62)<<endl;
    cout<<h1.search_modulo_division(3)<<endl;
    open_addressing h2(11,5);
    h2.add_linear_probing(62,"cat");
    h2.add_linear_probing(77,"bird");
    h2.add_linear_probing(56,"ant");
    h2.add_linear_probing(55,"dog");
    h2.add_linear_probing(132,"fish");
    h2.print();
    cout<<h2.search_linear_probing(132)<<endl;
    cout<<h2.search_linear_probing(35)<<endl;
    open_addressing h3(11,5);
    h3.add_quadratic_probing(62,"cat");
    h3.add_quadratic_probing(77,"bird");
    h3.add_quadratic_probing(56,"ant");
    h3.add_quadratic_probing(55,"dog");
    h3.add_quadratic_probing(132,"fish");
    h3.print();
    cout<<h3.search_quadratic_probing(55)<<endl;
    cout<<h3.search_quadratic_probing(165)<<endl;
    open_addressing h4(11,5);
    h4.add_double_hashing(62,"cat");
    h4.add_double_hashing(77,"bird");
    h4.add_double_hashing(56,"ant");
    h4.add_double_hashing(55,"dog");
    h4.add_double_hashing(132,"fish");
    h4.print();
    cout<<h4.search_double_hashing(77)<<endl;
    cout<<h4.search_double_hashing(143)<<endl;
    separate_chaining h5(11);
    h5.add(62,"cat");
    h5.add(77,"bird");
    h5.add(56,"ant");
    h5.add(55,"dog");
    h5.add(132,"fish");
    h5.print();
    cout<<h5.search(132)<<endl;
    cout<<h5.search(187)<<endl;
    return 0;
}
```

Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Mehlhorn, Kurt; Sanders, Peter (2008), "4 Hash Tables and Associative Arrays", Algorithms and Data Structures: The Basic Toolbox (PDF), Springer, pp. 81–98

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

Stallings, W., & Paul, G. K. (2012). Operating systems: internals and design principles (Vol. 9). New York: Pearson.

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/

https://en.wikipedia.org/wiki/Hash_table