

Hash Code

$g("c_{k-1} c_{k-2} \dots c_2 c_1 c_0") \rightarrow \text{integer}$

$$g("c_{k-1} c_{k-2} \dots c_2 c_1 c_0") = \sum_{n=0}^{k-1} (\text{int}) c_n$$

$$(\text{int}) c_{k-1} + (\text{int}) c_{k-2} + \dots + (\text{int}) c_0$$

Casting ..

30,000



T

Polynomial Hash Code

* it only works for positive number.

$$\text{Polynomial: } p(x) = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \dots + a_1x^1 + a_0$$

$$g("c_{k-1} c_{k-2} \dots c_2 c_1 c_0") = \sum_{i=0}^{k-1} [\text{int}(c_i)] \cdot x^i$$

this value of x can be picked yourself.
and it have to be a prime number.

e.g. 33, 37, ...

these number produce less collision.

Also, make sure the output number would not be too large to store.

e.g. $i=51, x=37$.

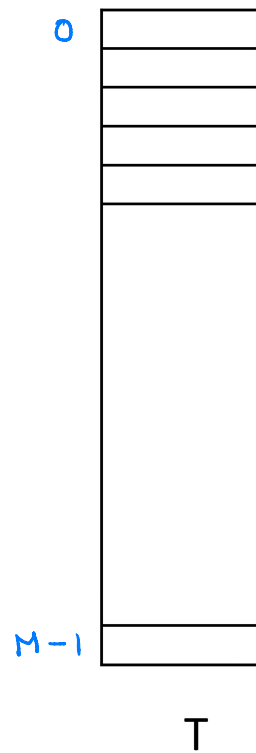
$\text{int}(c_{51}) \cdot 37^{51} \leftarrow \text{Too large!}$

But you can truncate this value so that it can be stored.

When doing truncation, there's risk that the interpreter might consider the number a negative number and the program crash.

No matter how good the hash function is, there's still possibility to have collisions, but we can improve the hash function to reduce the number of collision.

Compression Map



$$f(\text{integer}) \rightarrow \{0, 1, \dots, M-1\}$$

$f(i) = i \bmod M$ takes the remainder.

Hash Function with Polynomial Hash Code

$$h("c_{k-1} c_{k-2} \dots c_0") = (\dots (((((\text{int})c_{k-1})x + (\text{int})c_{k-2})x + (\text{int})c_{k-3})x + \dots + (\text{int})c_1)x + (\text{int})c_0) \bmod M$$

Handwritten annotations in blue ink below the equation:

- A bracket under $((\text{int})c_{k-1})x + (\text{int})c_{k-2}$ with the word "mod" written above it.
- A bracket under $((\text{int})c_{k-1})x + (\text{int})c_{k-2} + ((\text{int})c_{k-3})x + \dots + (\text{int})c_1$ with the word "mod" written below it.
- A bracket under the entire expression $((\text{int})c_{k-1})x + (\text{int})c_{k-2} + ((\text{int})c_{k-3})x + \dots + (\text{int})c_1 + (\text{int})c_0$ with the word "mod" written below it.
- A vertical ellipsis \vdots is written below the third bracket.

Hash Function with Polynomial Hash Code

$$h("c_{k-1} c_{k-2} \dots c_0") = (\dots (((\text{int})c_{k-1})x + (\text{int})c_{k-2})x + (\text{int})c_{k-3})x + \dots + (\text{int})c_1)x + (\text{int})c_0 \bmod M$$

\uparrow
k times of loop.

Algorithm polynomialHashFunction("c_{k-1}, c_{k-2}, ... c₀", x, M)

Input: String "c_{k-1}, c_{k-2}, ... c₀", value x, size M (M is a prime number) of the hash table

Output: value of the hash function for input string

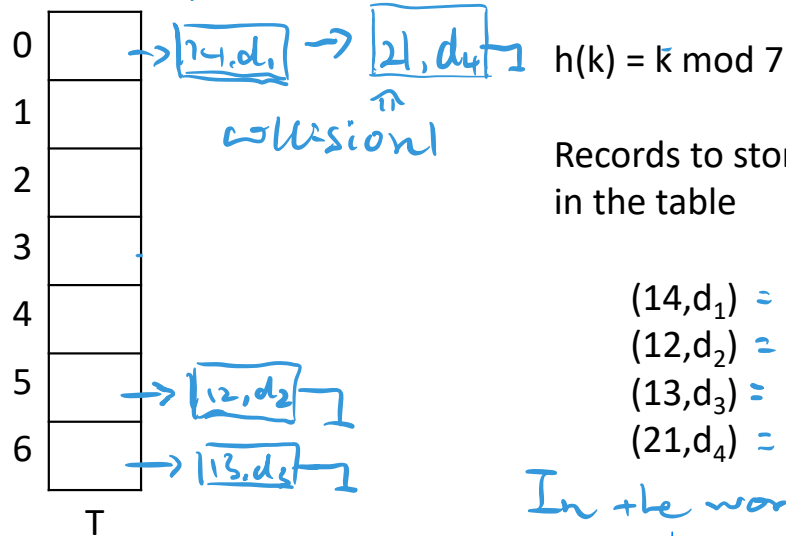
1- { v ← (int) c_{k-1}.
for i ← k-2 down to 0 => for (int i = k-2; i ≥ 0; i--) {
a1 v = (v · x + (int) c_i) mod M
L { return v.

$$\begin{aligned} \text{Complexity} &= a_1 + a_2 (k-1) \\ &= O(n). \end{aligned}$$

Drawback: Memory inefficient.

Create a new Collision Resolution: Separate Chaining

node to store
21, d₄ and make it a linkedlist.



In the worst case,
the linked list can
be infinitely long.
e.g. $h(0) = 0$.
A good hash function,
all nodes are separate
even in the entire
table.

$$\begin{aligned} & 90 + 5 \times 6 + 10 \times 3 + 20 \\ &= 90 + 30 + 30 + 20 \\ &= 170 \end{aligned}$$

↑ 2 30
5

Algorithm get(k)

Input: Key k

Output: Record with key k, or
null if no record has key k

```
pos ← h(k)
p ← T[pos]
while (p ≠ null) && (p.getKey() ≠ k) do {
    p ← p.getNext()
}
if p = null return null
else return p.getRecord()
```

Worst Case: P is not in the list

Complexity: $O(n)$.

Collision Resolution: Open Addressing

$$h(k) = k \bmod 7$$

Records to store
in the table

0	14, d ₁
1	21, d ₄
2	19, d ₅
3	2, d ₆
4	null
5	12, d ₂
6	13, d ₃

T

(14, d ₁)	0
(12, d ₂)	5
(13, d ₃)	6
(21, d ₄)	0
(19, d ₅)	5
(2, d ₆)	2
(5, d ₇)	5

remove (14)

← find the
nearest
available position

null cannot be a
valid data.

calculate hash function

⇒ when collision occur,

find the next available position
Initially every entry of T is null

Linear probing:

$$h(k), (h(k)+1) \bmod M, (h(k)+2) \bmod M, ((h(k)+3) \bmod M) \dots$$

stopped when 1) find the value 2) next position is null value.

Algorithm get(k)

Input: Key k

Output: Record with key k, or
null if no record has key k

```
pos ← h(k)
while (T[pos] ≠ null
      && T[pos].getKey() ≠ k) do {
    pos = (pos + 1) mod n // keep in array
    if (T[pos] == null) {
        return null;
    }
    else return T[pos];
}
```

⚠ this program is not finish, it needs to
add a counter to end the loop while
reach the end of the array.

Computer Memory

11010	11001	01100	10010	00011	10010	011010	11001	101100	010010	00011	110010	101100	010010	00011	110010	01110	11001	1101
11011	10001	00111	01101	10011	00101	101011	10001	100111	01101	10011	00101	100111	01101	10011	00101	11011	000111	11011
11101	11111	10000	001101	01101	001101	11101	11111	00000	001101	01101	10011	00000	001101	01101	10011	11001	10011	0011
101101	110010	101110	011111	11011	10001	101101	11001	11011	001111	11011	10001	11011	001111	11011	10001	10101	10101	01110
011010	11001	101100	010010	101101	011100	011010	11001	101100	010010	00011	110010	101100	010010	00011	110010	00011	010011	01110
101011	10001	100111	01101	10011	00101	101011	10001	100111	01101	10011	00101	100111	01101	10011	00101	11011	10011	10101
11101	11111	00000	001101	01101	10011	11101	11111	00000	001101	01101	10011	00000	001101	01101	10011	10110	011100	0111
101101	11001	11011	001111	11011	10001	101101	11001	11011	001111	11011	10001	11011	001111	11011	10001	10011	10111	0011
011010	11001	101100	010010	00011	110010	011010	11001	101100	010010	00011	110010	101100	010010	00011	110010	11100	001111	11101
101011	10001	100111	01101	10011	00101	101011	10001	100111	01101	10011	00101	100111	01101	10011	00101	10011	011101	10011
11101	11111	00000	001101	01101	10011	11101	11111	00000	001101	01101	10011	00000	001101	01101	10011	01110	11111	00111
101101	11001	11011	001111	11011	10001	101101	11001	11011	001111	11011	10001	11011	001111	11011	10001	11100	01010	01010
011010	11001	101100	010010	00011	110010	011010	11001	101100	010010	00011	110010	101100	010010	00011	110010	10111	0011	0111
101011	10001	100111	01101	10011	00101	101011	10001	100111	01101	10011	00101	100111	01101	10011	00101	10000	01110	10101
11101	11111	00000	001101	01101	10011	11101	11111	00000	001101	01101	10011	00000	001101	01101	10011	11100	0011	00110
101101	11001	11011	001111	11011	10001	101101	11001	11011	001111	11011	10001	11011	001111	11011	10001	10000	01101	10101
11101	11111	00000	001101	01101	10011	11101	11111	00000	001101	01101	10011	00000	001101	01101	10011	11101	11100	10101
101101	11001	11011	001111	11011	10001	101101	11001	11011	001111	11011	10001	11011	001111	11011	10001	10000	11111	00111

Linear Probing and Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

$$h(k) = k \bmod 11$$

Records to store
in the table

$(3, d_1)$
 $(14, d_2)$
 $(25, d_3)$
 $(5, d_4)$
 $(28, d_5)$
 $(91, d_6)$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Secondary hash function:
 $h'(k) = q - (k \bmod q)$
for some prime value q

$$h'(k) = 7 - (k \bmod 7)$$

Linear probing:

$h(k)$, $(h(k) + 1) \bmod M$, $(h(k) + 2) \bmod M$,
 $((h(k) + 3) \bmod M) \dots$

Double hashing:

$h(k)$, $(h(k) + h'(k)) \bmod M$, $(h(k) + 2h'(k)) \bmod M$,
 $((h(k) + 3h'(k)) \bmod M) \dots$

Double Hashing and Size of the Table

0	
1	
2	
3	
4	
5	
6	
7	

$$h(k) = k \bmod 8$$

Records to store
in the table

$(2, d_1)$

$(6, d_2)$

$(10, d_3)$

Secondary hash function:

$$h'(k) = q - (k \bmod q)$$

for some prime value q

$$h'(k) = 7 - (k \bmod 7)$$

Double hashing:

$$h(k), (h(k) + h'(k)) \bmod M, (h(k) + 2h'(k)) \bmod M, ((h(k) + 3h'(k)) \bmod M \dots$$

Open Addressing: put Method (linear probing)

Algorithm put (k,data, M)

In: record (k,data) to insert, size M of hash table

Out: {add record (k,data) to table, or ERROR if insertion not allowed}

pos \leftarrow h(k)

count \leftarrow 0

while (T[pos] \neq NULL) **and** (T[pos] \neq DELETED) **do** {

if T[pos].getKey() = k **then** ERROR

 pos \leftarrow (pos + 1) mod M

 count \leftarrow count + 1

if count = M **then** ERROR

the table is full.

}

T[pos] \leftarrow (k,data)

Open Addressing: put Method (double hashing)

Algorithm put (k,data, M)

In: record (k,data) to insert, size N of hash table

Out: {add record (k,data) to table, or ERROR if insertion not allowed}

pos \leftarrow h(k)

count \leftarrow 0

while (T[pos] \neq NULL) **and** (T[pos] \neq DELETED) **do** {

if T[pos].getKey() = k **then** *ERROR*

 pos \leftarrow (pos + $h'(k)$) **mod** M

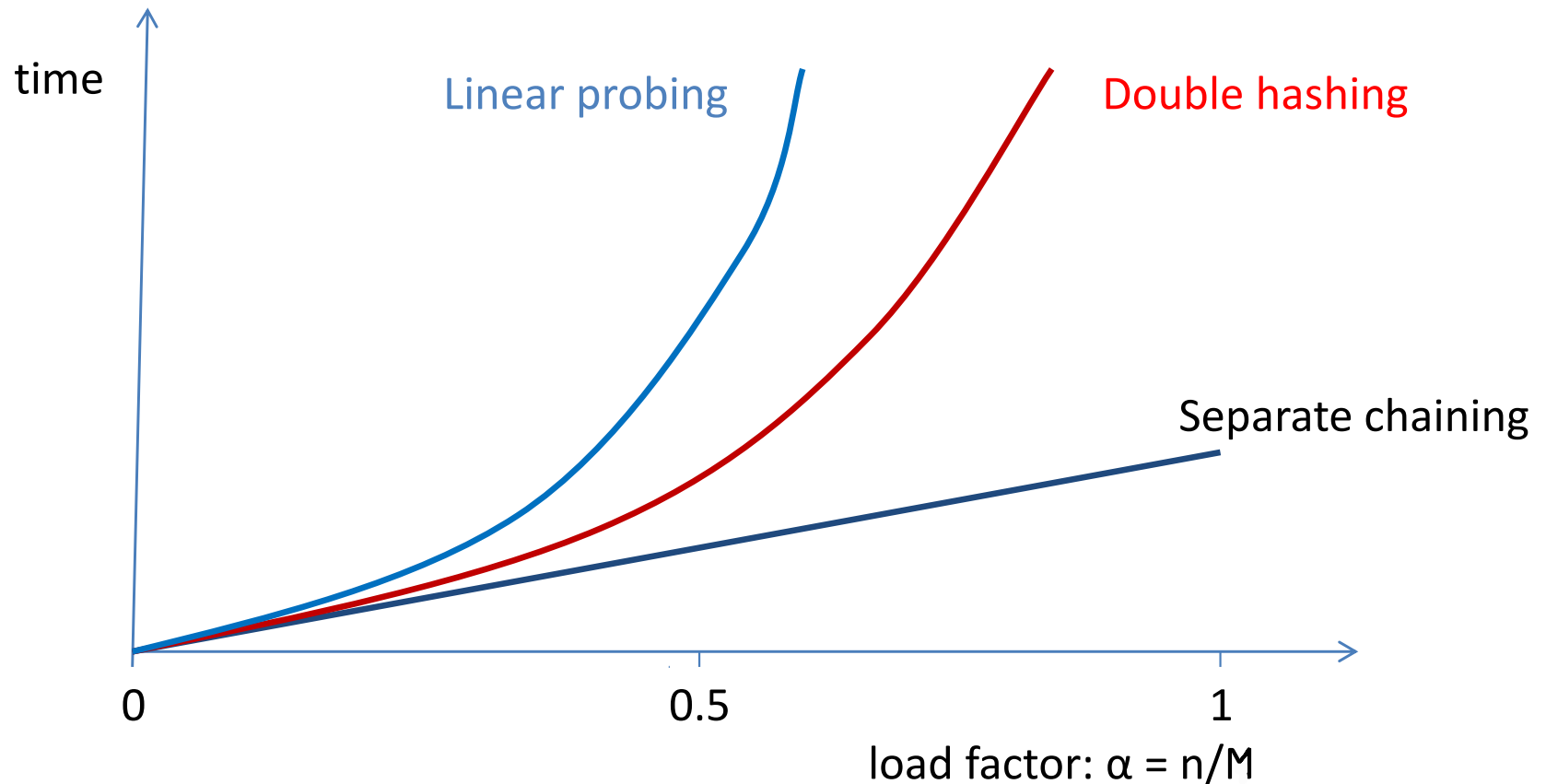
 count \leftarrow count + 1

if count = M **then** *ERROR*

}

T[pos] \leftarrow (k,data)

Average Time Complexity of **get** Operation



Average number of key
comparisons

Separate chaining

$$1 + \alpha$$

Linear Probing

$$\frac{1}{2} + \frac{1}{2(1 - \alpha)^2}$$

Double Hashing

$$\frac{1}{1 - \alpha}$$