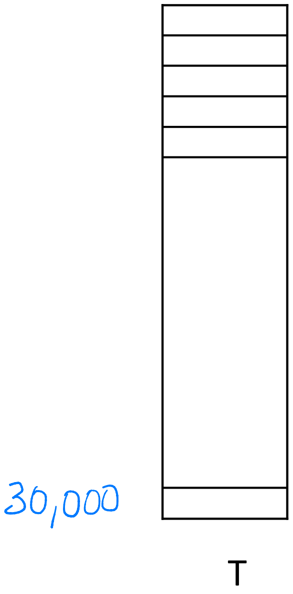
# Hash Code

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

$g(``c_{k-1}\ c_{k-2} \ldots c_2\ c_1\ c_0") = \sum_{n=0}^{k-1} (\text{int})\, C_n$

$(\text{int})\, C_{k-1} + (\text{int})\, C_{k-2} \cdots + (\text{int})\, C_0$

Casting



# Polynomial Hash Code

* it only works for positive number.

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

$g(\text{“}c_{k-1}\, c_{k-2} \ldots c_2\, c_1\, c_0\text{”}) = \sum_{i=0}^{k-1} \left[ (\text{int})\, C_i \right] \cdot x^i$

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.

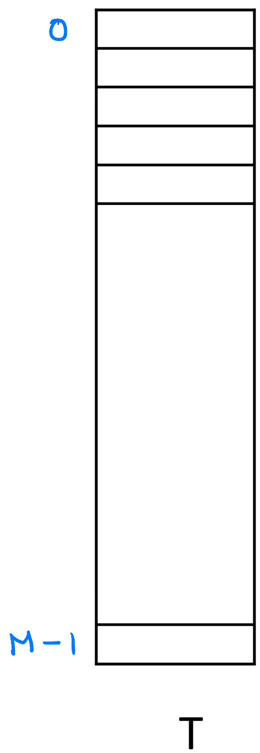
e.g. i=51, x=37.

$(\text{int})\, C_{57} \cdot 37^{57}$ ⇐ Too large!

But you can <u>truncate</u> 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




$f(\text{integer}) \longrightarrow \{0, 1, \ldots, M-1\}$

$f(i) = i \underbrace{\bmod}_{\%} M$ takes the remainder.

# Hash Function with Polynomial Hash Code

$$h(\text{“}c_{k-1}\ c_{k-2} \ldots c_0\text{”}) = (\ldots \underbrace{\underbrace{(((\underbrace{(\text{int})c_{k-1}}_{\text{mod}})x + (\text{int})c_{k-2})}_{\text{mod}}x + (\text{int})\ c_{k-3})}_{\text{mod}}x + \ldots + (\text{int})c_1)x + (\text{int})c_0)\ \text{mod } M$$

# Hash Function with Polynomial Hash Code

$h(\text{“}c_{k-1}\ c_{k-2}\ \ldots\ c_0\text{”}) = (\ldots ((((\text{int})c_{k-1})x + (\text{int})c_{k-2})x + (\text{int})\ c_{k-3})x + \ldots + (\text{int})c_1)x + (\text{int})c_0)$ mod M

k times of loop.

**Algorithm** polynomialHashFunction(“$c_{k-1}, c_{k-2}, \ldots\ c_0$”,x,M)
**Input**: String “$c_{k-1}, c_{k-2}, \ldots\ c_0$”, value x, size M (M is a prime number) of the hash table
**Output**: value of the hash function for input string

a1: v ← (int) $c_{k-1}$.

for i ← k-2 down to 0 => for(int i=k-2, i>-1, i--){ — a2.

v = (v · x + (int) (i)) mod M

return v.

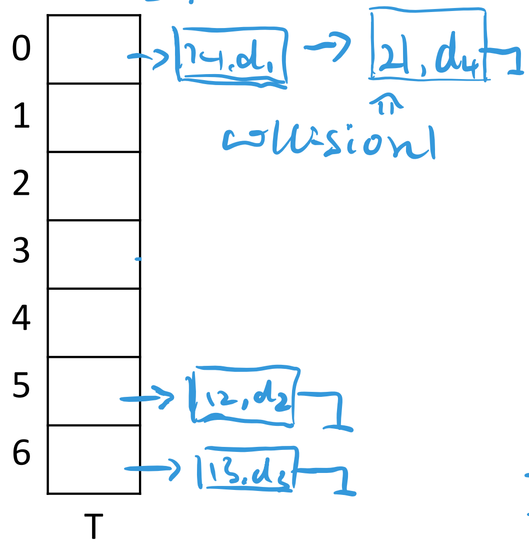
Complexity $= a_1 + a_2(k-1)$

$= O(n)$.

Drawback: Memory inefficient.

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

# Collision Resolution: Separate Chaining




h(k) = k mod 7

Records to store in the table

$(14,d_1)$ = 0
$(12,d_2)$ = 5
$(13,d_3)$ = 6
$(21,d_4)$ = 0

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

$80+5\times6+10\times3+20$
$=80+30+30+20$
$\boxed{=170}$.

100

12 30

5

**Algorithm** get(k)
**Input**: Key k
**Output**: Record with key k, or null if no record has key k

c1
```
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()
```
c2

Worst case: p is not in the list

Complexity: O(n).

# Collision Resolution: Open Addressing

| | T |
|---|---|
| 0 | 14, $d_1$ |
| 1 | 21, $d_4$ |
| 2 | 19, $d_5$ |
| 3 | 2, $d_6$ |
| 4 | null. |
| 5 | 12, $d_2$ |
| 6 | 13, $d_3$ |

← find the nearest available position.

null cannot be a valid data.

$h(k) = k \bmod 7$

Records to store in the table

| | |
|---|---|
| (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)

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) mod M, (h(k) + 2) mod M, ((h(k) + 3) mod M ..

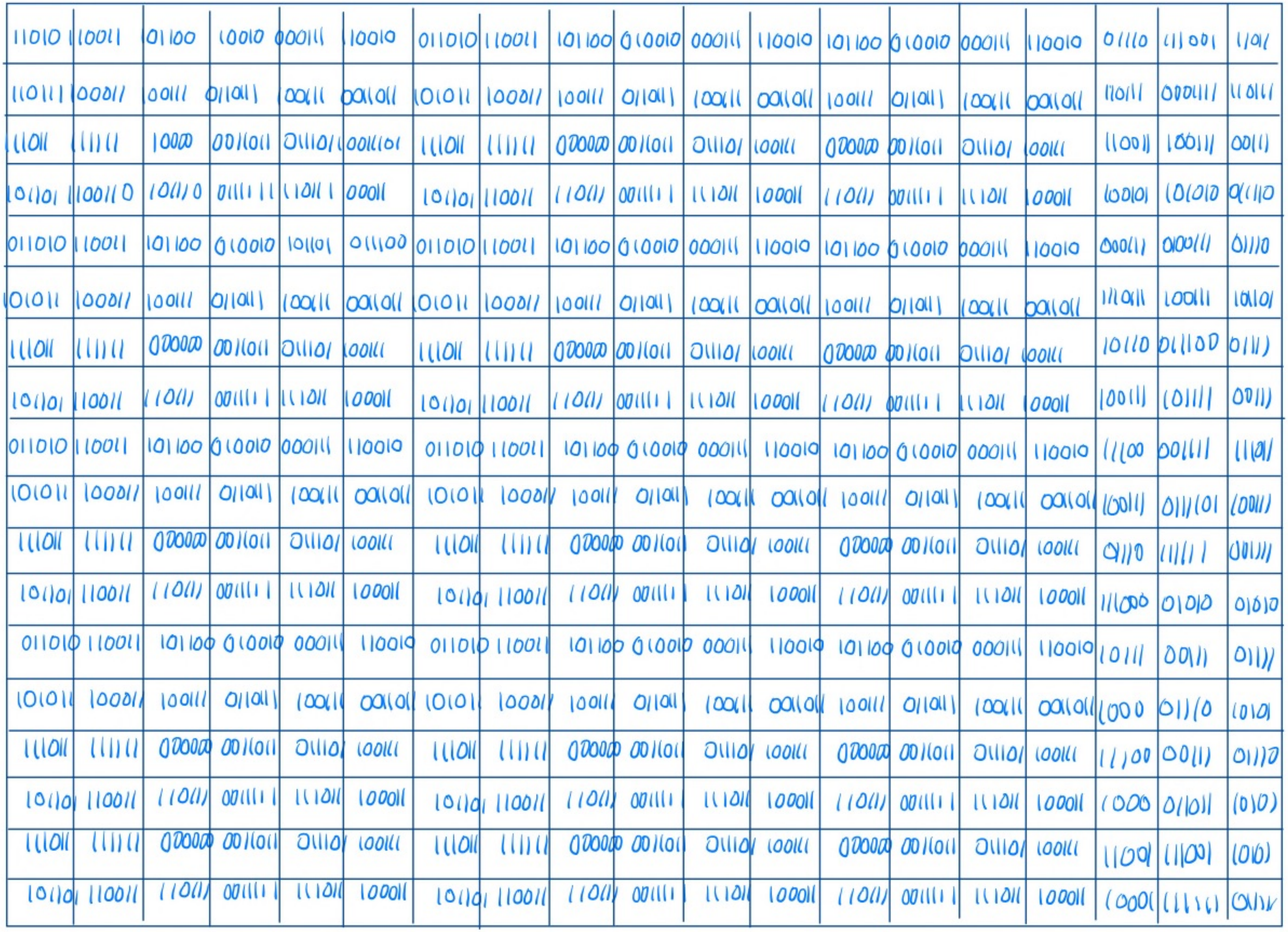
stopped when 1) find the value 2) next position is null value.

**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



# Linear Probing and Double Hashing

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |

h(k) = k mod 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 | |

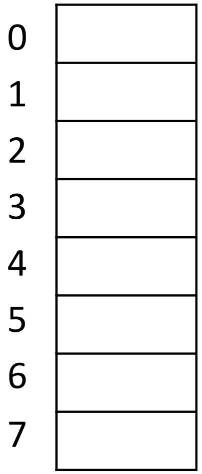
Secondary hash function:
h’(k) = q – (k mod q)
for some prime value q

h’(k) = 7 – (k mod 7)

Linear probing:
h(k), (h(k)+1) mod M, (h(k) + 2) mod M,
((h(k) + 3) mod M …

Double hashing:
h(k), (h(k)+h’(k)) mod M, (h(k) + 2h’(k)) mod M,
((h(k) + 3h’(k)) mod M …

# Double Hashing and Size of the Table




$h(k) = k \bmod 8$

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:

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

# 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 ← h(k)

count ← 0

**while** (T[pos] != NULL) **and** (T[pos] != DELETED) **do** {

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

    pos ← (pos + 1) **mod** M

    count ← count + 1

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

the table is full.

}

T[pos] ← (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 ← h(k)

count ← 0

**while** (T[pos] != NULL) **and** (T[pos] != DELETED) **do** {

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

  pos ← (pos + h'(k)) **mod** M

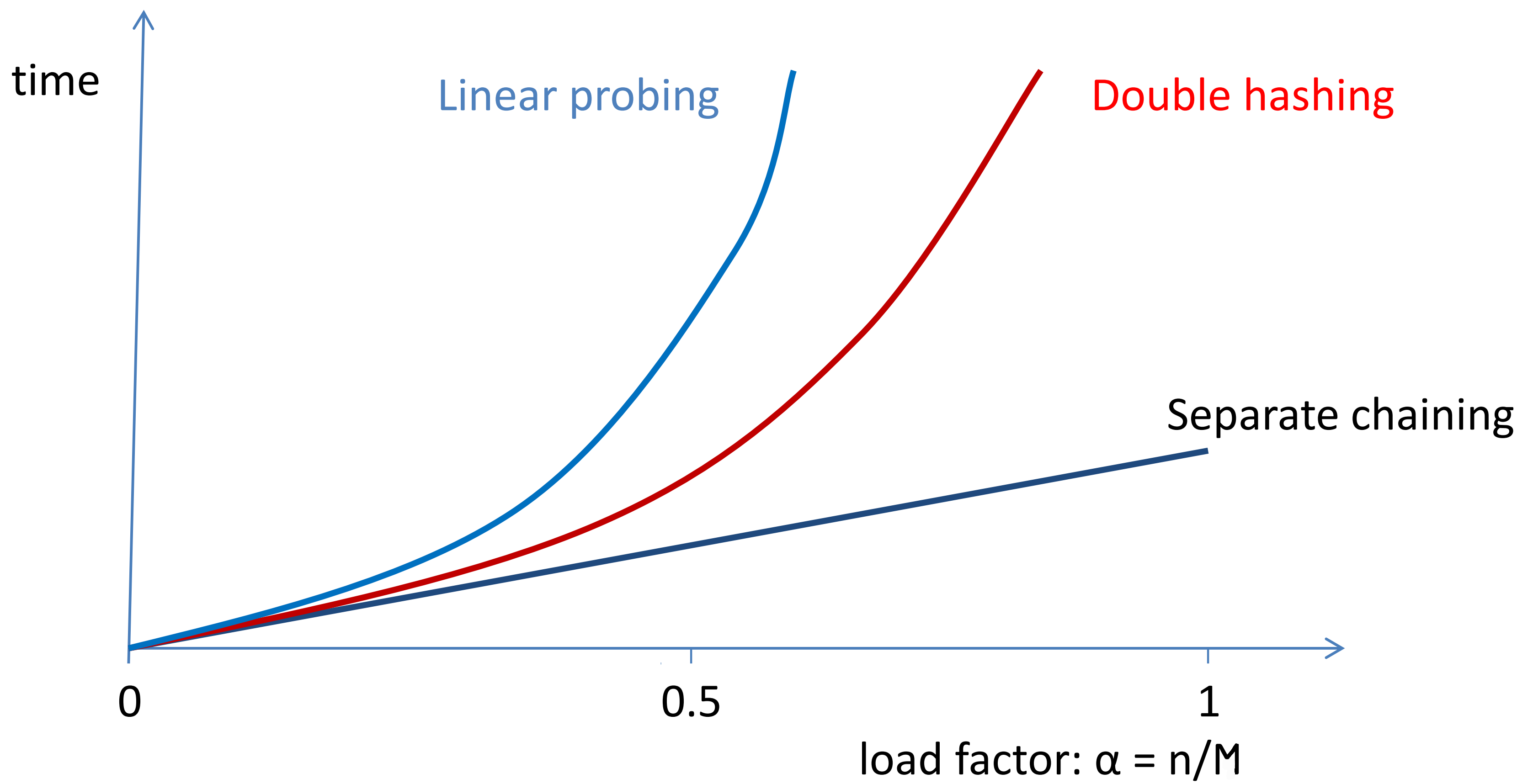
  count ← count + 1

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

}

T[pos] ← (k,data)

# Average Time Complexity of get Operation




| | Average number of key comparisons |
|---|---|
| Separate chaining | $1 + \alpha$ |
| Linear Probing | $\frac{1}{2} + 1/(2(1 - \alpha)^2)$ |
| Double Hashing | $1/(1 - \alpha)$ |