Chapter 14: Hash Tables

Data Structures and Algorithms with C

Direct addressing · Hash functions · Collisions · Linear / quadratic probing · Double hashing · Chaining · Load factor · Tombstones

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_14_Hash_Tables.html

INTRODUCTION

Learning Outcomes

Explain the idea

Define the hash table, which computes a position from the key and searches in O(1) on average.

Choose a function

Compute division, multiplication, folding and string hashes; list what makes a function good.

Resolve collisions

Apply linear/quadratic probing, double hashing and chaining step by step.

Write it in C

Write insert, search and delete with `malloc` / `free` and the empty cases.

Delete correctly

Show why open addressing needs a deleted marker (tombstone).

Analyze

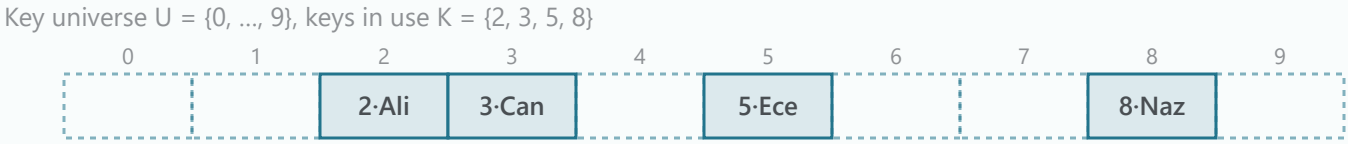
Explain how the load factor affects cost, and the amortized cost of resizing.

Bringing Search Cost Down to a Constant

Structure	Search	Insert	Delete	Note
Unsorted array	O(n)	O(1)	O(n)	append at the end, scan from the front
Sorted array	O(lg n)	O(n)	O(n)	binary search + shifting
Balanced search tree	O(lg n)	O(lg n)	O(lg n)	also gives sorted traversal
Hash table	O(1)*	O(1)*	O(1)*	* average; worst case O(n)

A hash table does not compare keys: it **computes the position** from the key. The price: no sorted traversal, min/max or range queries; the worst case is O(n).

The Simplest Case First: A Direct-Address Table



T[k] looks directly at k: search, insert, delete O(1) — but the table must be as large as |U|.

```c
typedef struct { int used; char name[16]; } rec;
rec T[10];                  /* key = index     */

rec *da_search(int k) {
  return T[k].used ? &T[k] : NULL;
}
```

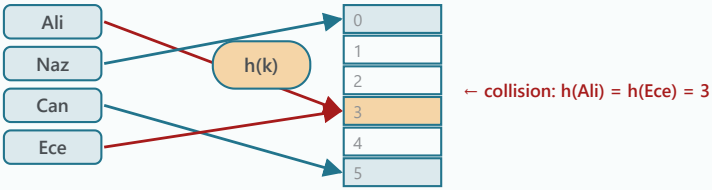
The problem

For 9-digit keys such as student numbers, $|U| = 10^9$: the table does not fit, and most of it stays empty.

The solution

A **hash function** that maps keys to a small range, h: U → {0, …, m−1}.

Key → Hash → bucket



Term	Meaning
bucket / slot	a position in the table, 0 … m−1
home address	h(k): the first place probed for the key
collision	$k_1 \neq k_2$ but $h(k_1) = h(k_2)$
synonyms	keys with the same home address
load factor α	n / m (how full the table is)

Since m < |U| (the pigeonhole principle), collisions are **unavoidable**. Two questions remain: how to choose a good h, and how to resolve collisions?

Abstract Data Type: The Dictionary

```
typedef struct { int key; /* + value */ } item;

bool hash_insert(item x);   /* update if found */
item *hash_search(int key); /* NULL if absent  */
bool hash_delete(int key); /* false if absent */
```

Dictionary

A key → value mapping; each key appears at most once.

Uses

Compiler symbol tables (Horowitz's motivating example), caches, duplicate detection, set implementations.

The C standard library has no ready-made hash table (POSIX has a limited `hsearch`), so a C programmer must know how to write one.

What Makes a Hash Function Good

Fast

O(key length) to compute; a few arithmetic operations.

Spreads evenly

Keys should spread over the m buckets as evenly as possible (the simple uniform hashing assumption).

Uses all the bits

Functions that look only at the last digits cluster on patterned data.

Deterministic

The same key always goes to the same bucket; randomness may enter only when the function is chosen.

Division Method: h(k) = k mod m

k	10	20	30	40	50	60	17	27
k mod 10	0	0	0	0	0	0	7	7
k mod 11	10	9	8	7	6	5	6	5

- m = 10: everything but the last digit is ignored; 10, 20, …, 60 land in the same bucket.
- A good rule is to choose m as a **prime** not close to a power of 2 (CLRS).
- If $m = 2^p$, h(k) is just the lowest p bits of k: avoid it.

```
unsigned h_div(unsigned k, unsigned m) { return k % m; }
```

In C, `%` can give a negative result for a negative number: keep the key `unsigned` or use `((k % m) + m) % m`.

Multiplication Method: $h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$

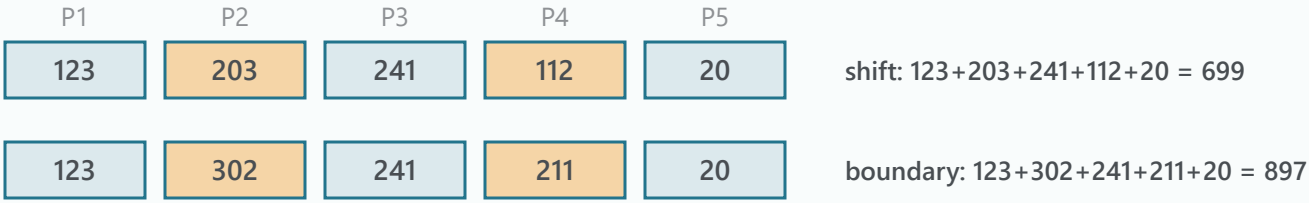
- A is a constant with $0 < A < 1$; the **fractional part** of $k \cdot A$ is multiplied by m.
- The choice of m is not critical; $m = 2^p$ is fine.
- Knuth suggests $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$
- Example (CLRS): k = 123456, p = 14 (m = 16384) → **h(k) = 67**.

```
/* w = 32 bit, m = 2^p; s = A * 2^32 */
uint32_t h_mul(uint32_t k, int p) {
  const uint32_t s = 2654435769u;
  return (uint32_t)(k * s) >> (32 - p);
}
```

In the integer version the product $k \cdot s$ overflows modulo 2^{32}; the overflow is discarded (same as taking the fractional part). The top p bits of the remaining 32 bits are h(k). The floating-point version gives the same result: $\lfloor 16384 \cdot 0.\ldots \rfloor = 67$. (Needs `<stdint.h>`.)

Folding and Mid-Square Methods

The key `x = 123203241112220` is split into three-digit parts that are added up (Horowitz's example):



Shift folding

The parts are added as they are: **699**.

Folding at the boundaries

Every other part is reversed (like folding paper): **897**.

Mid-square

Compute k^2 and take the middle r bits: $m = 2^r$. The middle bits depend on all the bits of k.

If the result exceeds the table size, also take `mod m`.

String Keys: Polynomial (Horner) Hashing

```c
unsigned str_hash(const char *s, unsigned m) {
  unsigned h = 0;
  for (; *s != '\0'; s++)
    h = (h * 31 + (unsigned char)*s) % m;
  return h;
}
```

```c
unsigned long djb2(const char *s) {  /* Bernstein */
  unsigned long h = 5381;
  int c;
  while ((c = (unsigned char)*s++) != 0)
    h = h * 33 + c;
  return h;       /* then: h % m */
}
```

Summing is bad

Adding character codes sends anagrams such as `"abc"` and `"cba"` to the same bucket.

Polynomial

$h = c_0 \cdot 31^{n-1} + \ldots + c_{n-1}$: one multiplication per character with Horner's rule. Position information enters the hash.

Overflow

Taking `% m` at each step keeps values small; `unsigned` overflow is well defined in C (mod 2^w).

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = ?

m = 11

`str_hash("kedi", 11)` is called.

Step 1 / 12

```
▶ unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "kedi"` `m = 11`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = 0 m = 11

Start with **h = 0**.

Step 2 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
▶   unsigned h = 0;
3   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "kedi"` `m = 11` `h = 0`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = 0

m = 11

`*s` = 'k' ≠ '\0' → the loop continues.

Step 3 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "kedi"` `m = 11` `h = 0`

`*s = 'k' (107)`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = 8 m = 11

h = (0·31 + 107) % 11 = 107 % 11 = **8**.

Step 4 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "kedi"` `m = 11` `h = 8` `*s = 'k' (107)`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = 8 m = 11

`*s` = 'e' ≠ '\0' → the loop continues.

Step 5 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "edi"` `m = 11` `h = 8`

`*s = 'e' (101)`

CALL STACK

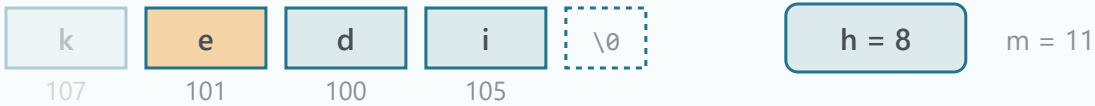
`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

Code Trace: `str_hash("kedi", 11)`

characters and codes



h = (8·31 + 101) % 11 = 349 % 11 = **8**.

Step 6 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "edi"` `m = 11` `h = 8` `*s = 'e' (101)`

CALL STACK

`str_hash("kedi", 11)`

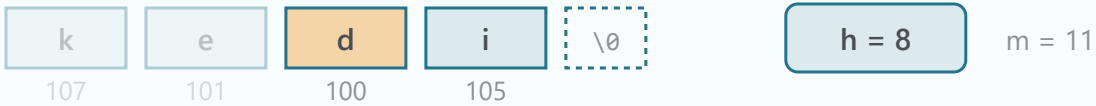
`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes



`*s` = 'd' ≠ '\0' → the loop continues.

Step 7 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "di"` `m = 11` `h = 8` `*s = 'd' (100)`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = 7 m = 11

h = (8·31 + 100) % 11 = 348 % 11 = **7**.

Step 8 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "di"` `m = 11` `h = 7` `*s = 'd' (100)`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = 7 m = 11

`*s` = 'i' ≠ '\0' → the loop continues.

Step 9 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "i"` `m = 11` `h = 7` `*s = 'i' (105)`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

Code Trace: `str_hash("kedi", 11)`

characters and codes



h = (7·31 + 105) % 11 = 322 % 11 = **3**.

Step 10 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = "i"` `m = 11` `h = 3` `*s = 'i' (105)`

CALL STACK

`str_hash("kedi", 11)`

`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes

k	e	d	i	\0
107	101	100	105	

h = 3

m = 11

`*s == '\0'` : end of string.

Step 11 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

VARIABLES

`s = ""` `m = 11` `h = 3` `*s = '\0'`

CALL STACK

`str_hash("kedi", 11)`

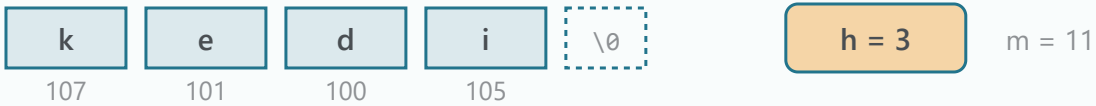
`main()`

Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

STRING HASHING · RUN THE CODE STEP BY STEP

Code Trace: `str_hash("kedi", 11)`

characters and codes



return 3: the home address of the key "kedi" is 3. The cost is linear in the string length.

Step 12 / 12

```
1 unsigned str_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
▶   return h;
6 }
```

VARIABLES

`m = 11` `h = 3`

CALL STACK

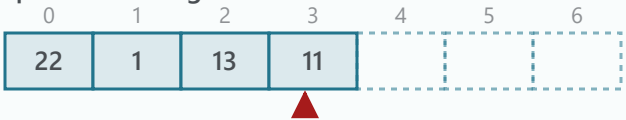
`str_hash("kedi", 11)`

`main()`

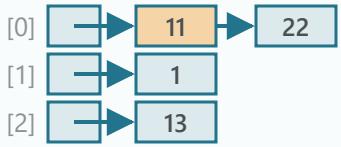
Check: the full polynomial is 3287903, and 3287903 mod 11 = 3; taking the mod at each step does not change the result.

Collisions: Two Basic Remedies

Open addressing: look for another slot in the table



Chaining: each bucket keeps a list



The birthday paradox

In a class of 23, the chance that two people share a birthday is ≈ 50.7%. 23 keys in m = 365 slots: collisions come surprisingly early.

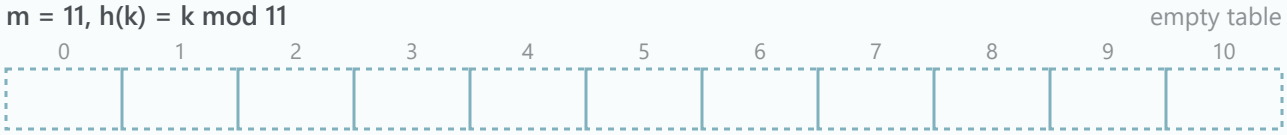
Open addressing

All items live inside the table; $\alpha \leq 1$. An empty slot is found by following a probe sequence.

Chaining

Each bucket heads a linked list; $\alpha > 1$ is possible. Insertion is O(1).

Linear Probing: h(k, i) = (h(k) + i) mod m

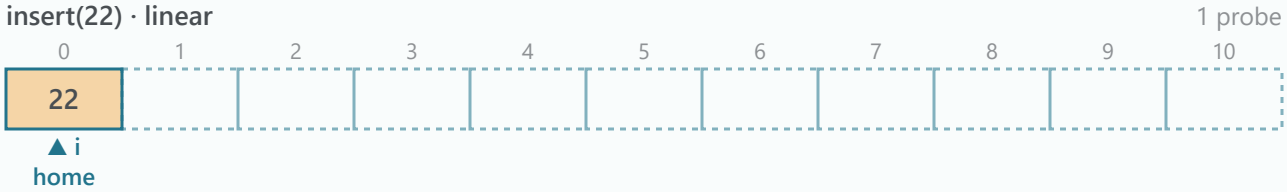


Insert **22, 1, 13, 11, 24, 33** in order. Home addresses: 0, 1, 2, 0, 2, 0.

Step 1 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

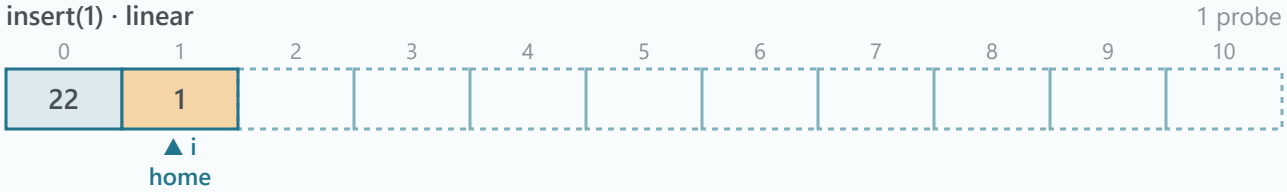
Linear Probing: h(k, i) = (h(k) + i) mod m



22 → home 0: **slot 0**.

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

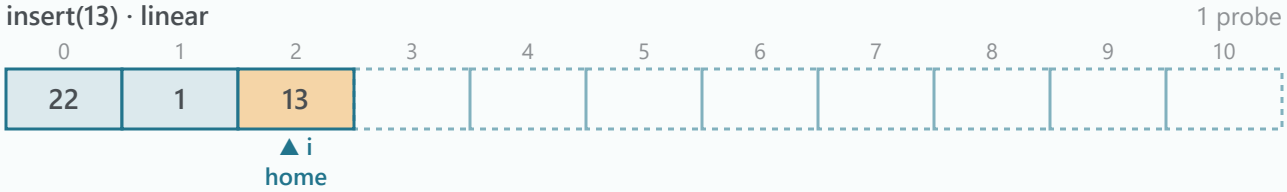


1 → home 1: **slot 1**.

Step 3 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

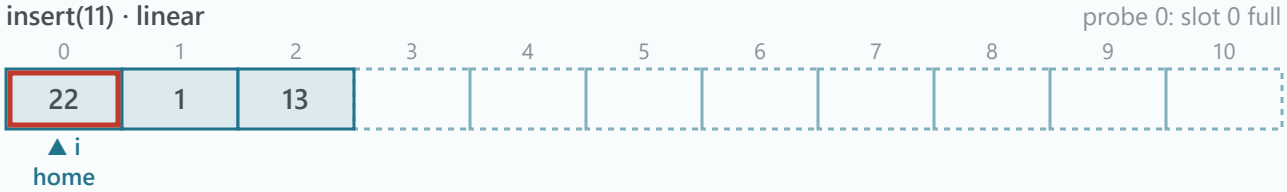


13 → home 2: **slot 2**.

Step 4 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

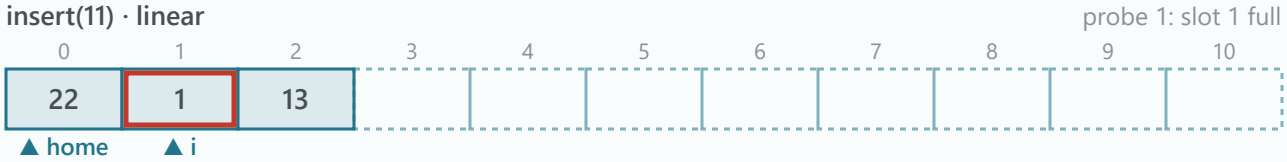
Linear Probing: h(k, i) = (h(k) + i) mod m



11: slot **0** is full (22) → try the next slot.

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

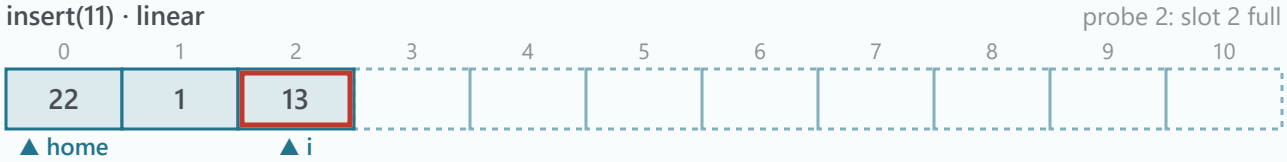
Linear Probing: h(k, i) = (h(k) + i) mod m



11: slot **1** is full (1) → try the next slot.

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

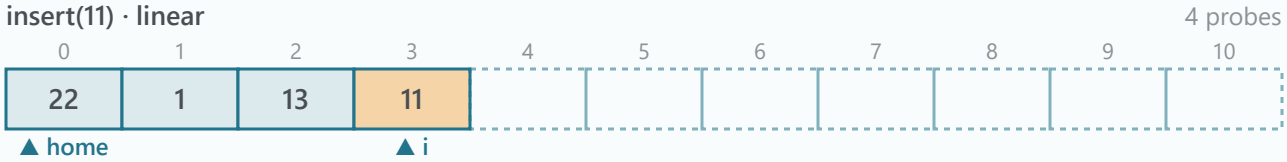
Linear Probing: h(k, i) = (h(k) + i) mod m



11: slot **2** is full (13) → try the next slot.

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

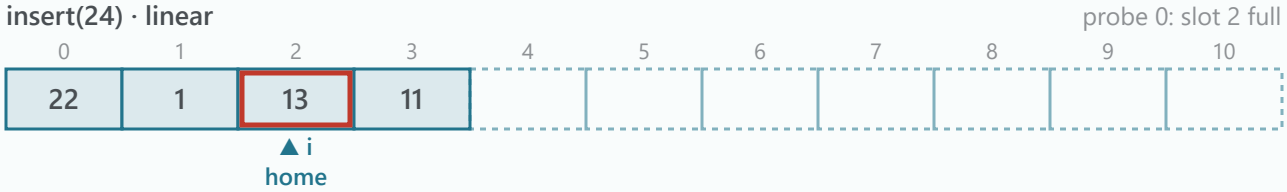


11 → home 0, probe sequence 0, 1, 2, 3: **slot 3**.

Step 8 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

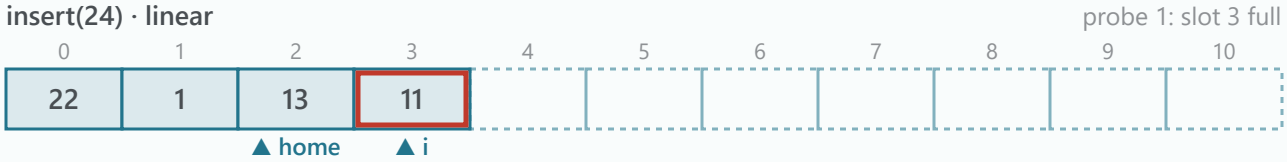


24: slot **2** is full (13) → try the next slot.

Step 9 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

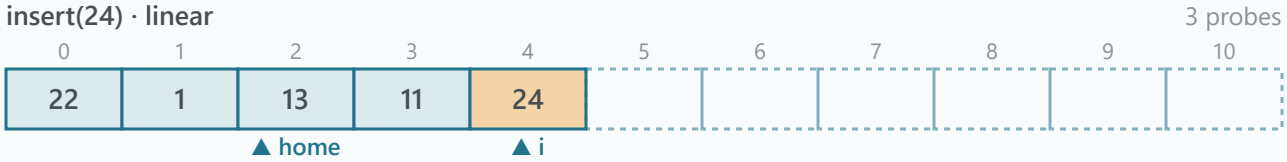


24: slot **3** is full (11) → try the next slot.

Step 10 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

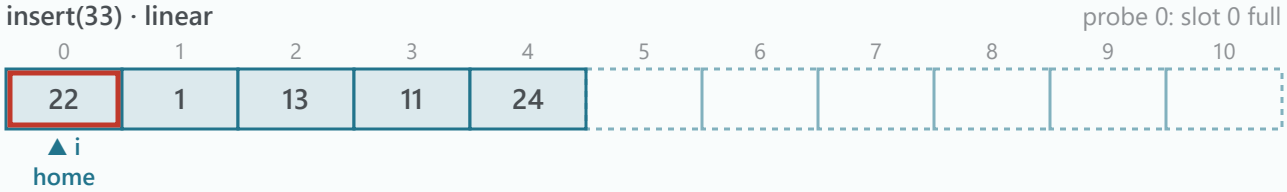


24 → home 2, probe sequence 2, 3, 4: **slot 4**.

Step 11 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

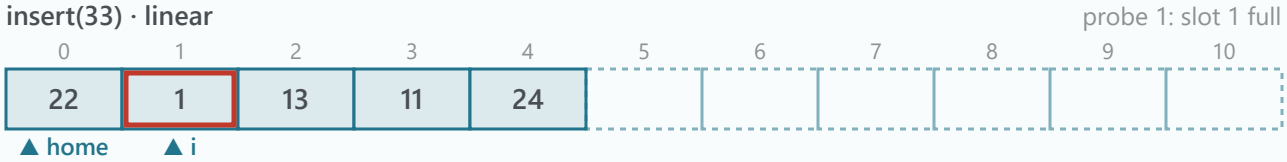


33: slot **0** is full (22) → try the next slot.

Step 12 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

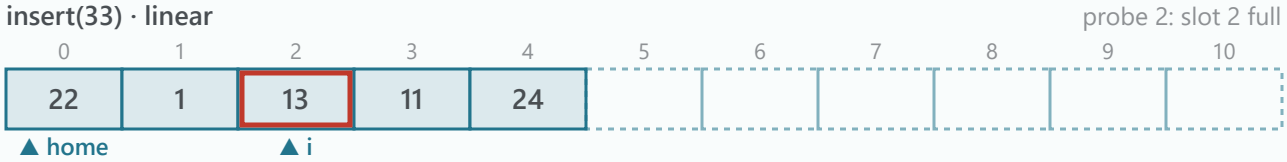


33: slot **1** is full (1) → try the next slot.

Step 13 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

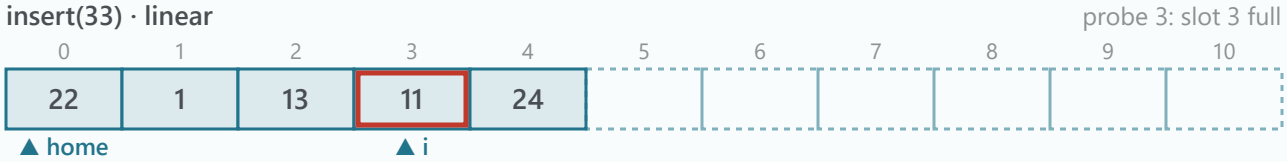


33: slot **2** is full (13) → try the next slot.

Step 14 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

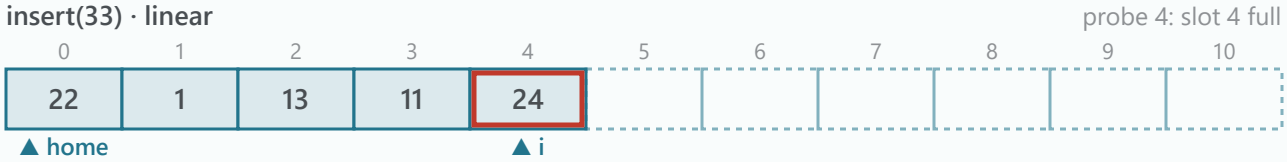


33: slot **3** is full (11) → try the next slot.

Step 15 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m

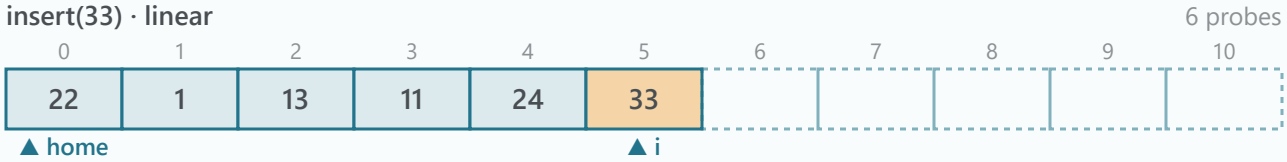


33: slot **4** is full (24) → try the next slot.

Step 16 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Linear Probing: h(k, i) = (h(k) + i) mod m



33 → home 0, probe sequence 0, 1, 2, 3, 4, 5: **slot 5**.

Step 17 / 17

Primary clustering: occupied slots form a contiguous block, and every new key that lands in it makes it bigger. 33 needed 6 probes.

Open Addressing in C: Slot State and Insertion

```c
#define M 11
typedef enum { EMPTY, USED, DELETED } state_t;
typedef struct {
  int key;          /* assumes key >= 0 */
  state_t state;
} slot;
slot table[M];          /* global: all 0 = EMPTY */
```

Initial state

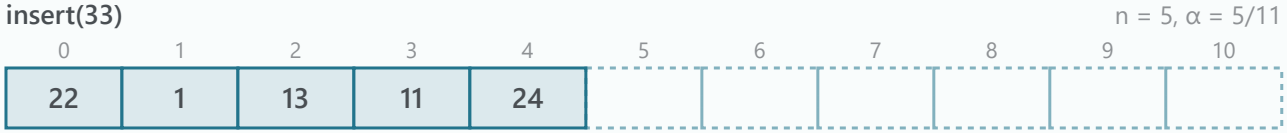
A global array starts zeroed; since `EMPTY` = 0, every slot is empty.

Insertion

Start at the home address and write into the first slot that is not `USED`; at most m probes.

A slot is in one of three states: **EMPTY** (never used), **USED**, **DELETED** (tombstone). Insertion may reuse a DELETED slot, but it must first check with `search` that the key is not already in the table.

Code Trace: `insert(33)` — Linear Probing



insert(33): the table holds 22, 1, 13, 11, 24 (slots 0–4, one full block).

Step 1 / 16

```
▶ int insert(int k) {
2   int i = k % M;                    /* home addr */
3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
4     if (table[i].state != USED) { /* free slot */
5       table[i].key = k;
6       table[i].state = USED;
7       return i;
8     }
9   }
10  return -1;                        /* table full */
11 }
```

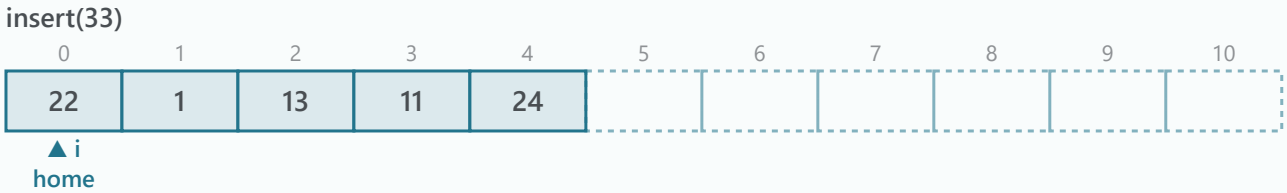
VARIABLES

`k = 33`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



Home address **i = 33 % 11 = 0**.

Step 2 / 16

```
1  int insert(int k) {
▶    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

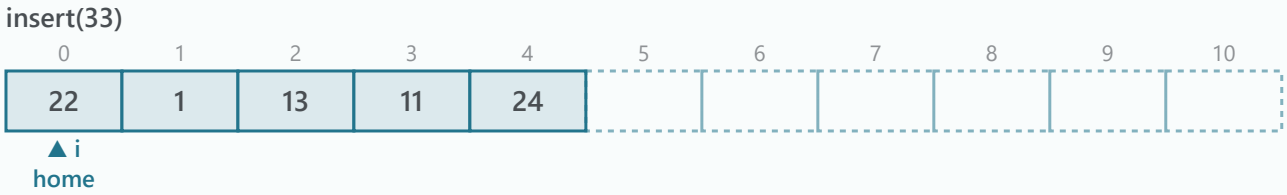
VARIABLES

`k = 33` `i = 0`

CALL STACK

- **insert(33)**
- main()

Code Trace: `insert(33)` — Linear Probing



d = 0 < 11: probe 0, slot 0.

Step 3 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

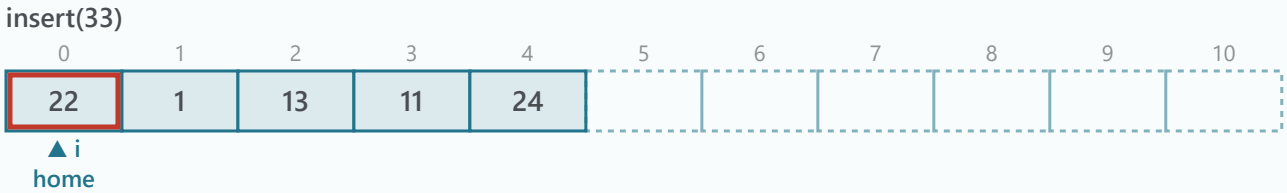
VARIABLES

`k = 33` `i = 0` `d = 0`

CALL STACK

- **insert(33)**
- main()

Code Trace: `insert(33)` — Linear Probing



Slot 0 is USED (22) → next: i = (0+1) % 11.

Step 4 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

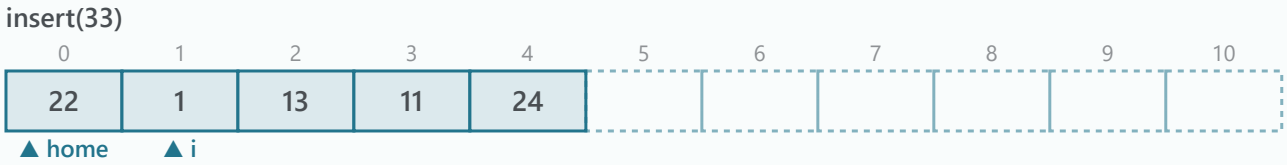
`k = 33` `i = 0` `d = 0`

`table[i].state = USED`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



d = 1 < 11: probe 1, slot 1.

Step 5 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

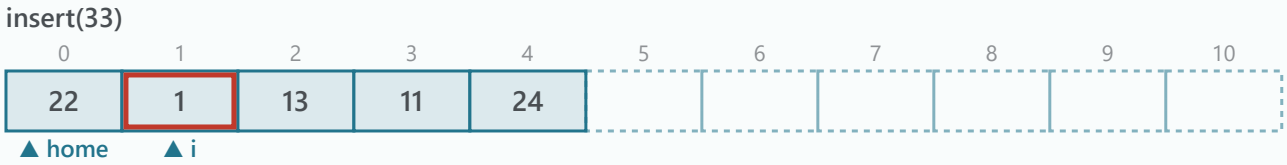
VARIABLES

`k = 33` `i = 1` `d = 1`

CALL STACK

- **insert(33)**
- main()

Code Trace: `insert(33)` — Linear Probing



Slot 1 is USED (1) → next: i = (1+1) % 11.

Step 6 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

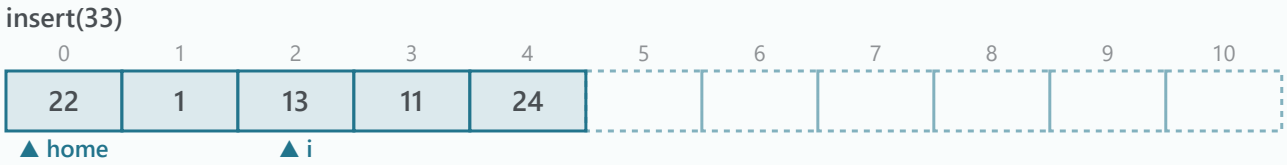
`k = 33` `i = 1` `d = 1`

`table[i].state = USED`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



d = 2 < 11: probe 2, slot 2.

Step 7 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

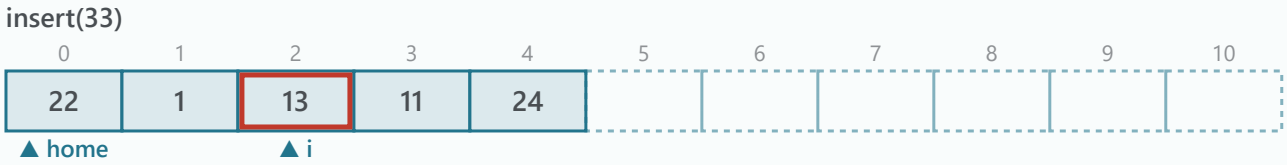
VARIABLES

`k = 33` `i = 2` `d = 2`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



Slot 2 is USED (13) → next: i = (2+1) % 11.

Step 8 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

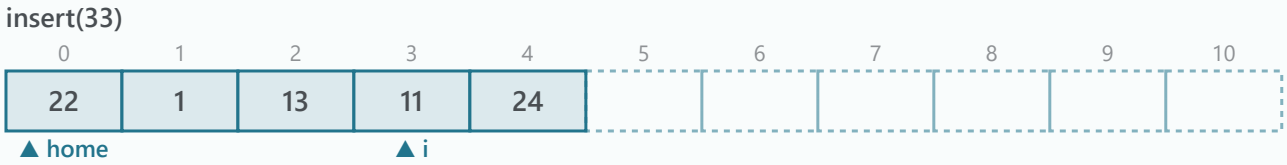
`k = 33` `i = 2` `d = 2`

`table[i].state = USED`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



d = 3 < 11: probe 3, slot 3.

Step 9 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

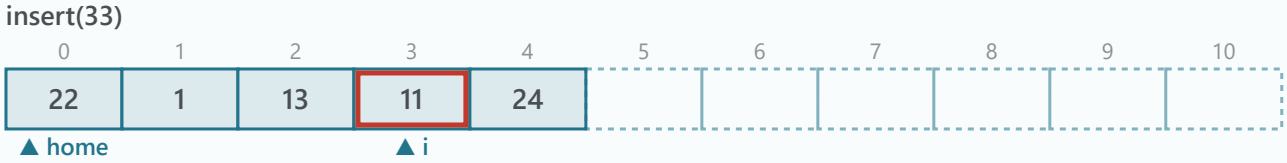
VARIABLES

`k = 33` `i = 3` `d = 3`

CALL STACK

- **insert(33)**
- main()

Code Trace: `insert(33)` — Linear Probing



Slot 3 is USED (11) → next: i = (3+1) % 11.

Step 10 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

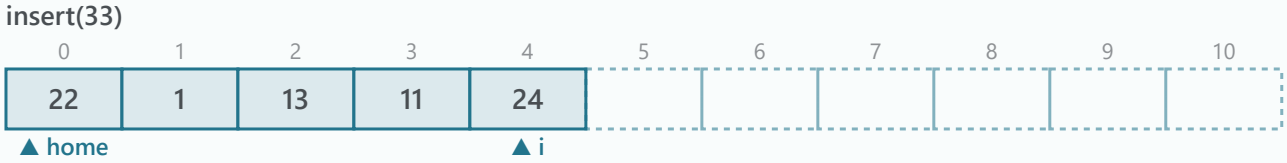
`k = 33` `i = 3` `d = 3`

`table[i].state = USED`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



d = 4 < 11: probe 4, slot 4.

Step 11 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

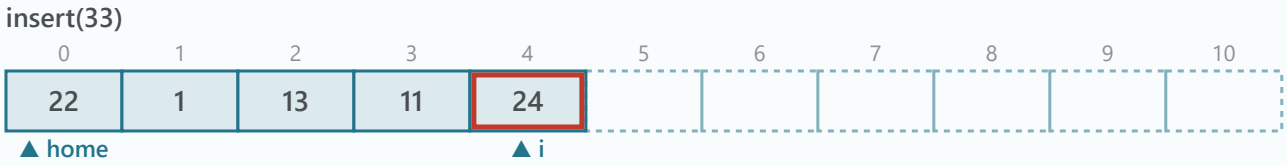
VARIABLES

`k = 33` `i = 4` `d = 4`

CALL STACK

- **insert(33)**
- main()

Code Trace: `insert(33)` — Linear Probing



Slot 4 is USED (24) → next: i = (4+1) % 11.

Step 12 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

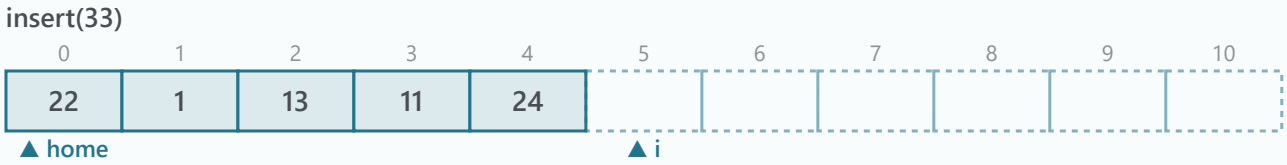
`k = 33` `i = 4` `d = 4`

`table[i].state = USED`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



d = 5 < 11: probe 5, slot 5.

Step 13 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

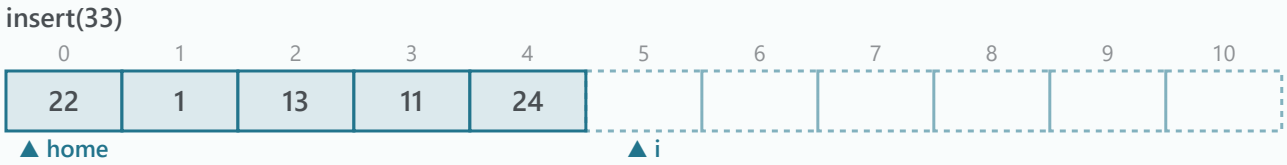
VARIABLES

`k = 33` `i = 5` `d = 5`

CALL STACK

- **insert(33)**
- main()

Code Trace: `insert(33)` — Linear Probing



Slot 5 is **EMPTY** → write here.

Step 14 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

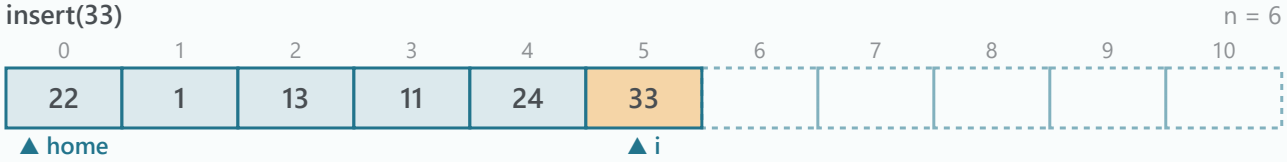
`k = 33` `i = 5` `d = 5`

`table[i].state = EMPTY`

CALL STACK

- `insert(33)`
- `main()`

Code Trace: `insert(33)` — Linear Probing



`table[5] = {33, USED}` .

Step 15 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
▶        table[i].key = k;
▶        table[i].state = USED;
7        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

`k = 33` `i = 5` `d = 5`

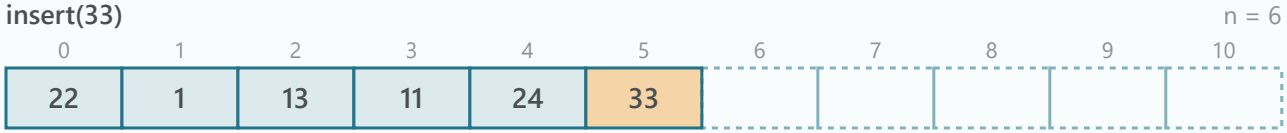
`table[i].state = USED`

CALL STACK

- `insert(33)`
- `main()`

OPEN ADDRESSING · RUN THE CODE STEP BY STEP

Code Trace: `insert(33)` — Linear Probing



return 5: 6 probes. Primary clustering pushed it 5 slots away from its home address.

Step 16 / 16

```
1  int insert(int k) {
2    int i = k % M;                    /* home addr */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state != USED) { /* free slot */
5        table[i].key = k;
6        table[i].state = USED;
▶        return i;
8      }
9    }
10   return -1;                        /* table full */
11 }
```

VARIABLES

`k = 33` `i = 5` `d = 5`

CALL STACK

- **insert(33)**
- main()

Code Trace: `search(24)` and `search(44)`



search(24) is called.

Step 1 / 14

```
▶ int search(int k) {
2   int i = k % M;
3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
4     if (table[i].state == EMPTY) return -1;
5     if (table[i].state == USED &&
6         table[i].key == k) return i;
7   }
8   return -1;
9 }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

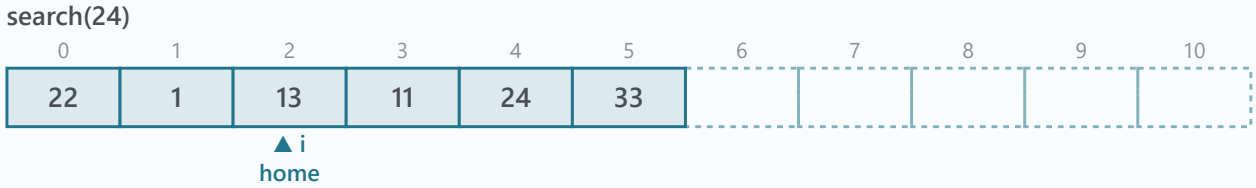
k = 24

CALL STACK

search(24)

main()

Code Trace: `search(24)` and `search(44)`



Home address i = 24 % 11 = **2**.

Step 2 / 14

```
1  int search(int k) {
▶    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state == EMPTY) return -1;
5      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

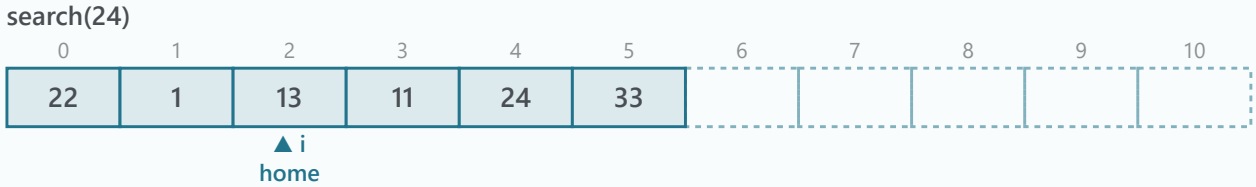
`k = 24` `i = 2`

CALL STACK

`search(24)`

`main()`

Code Trace: `search(24)` and `search(44)`



Slot 2: another key (13) → i = 3.

Step 3 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

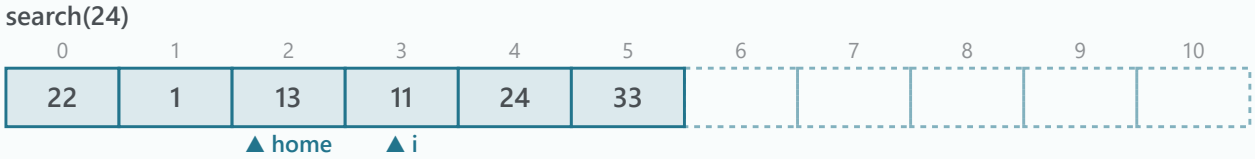
`k = 24` `i = 2` `d = 0`

`state = USED`

CALL STACK

- `search(24)`
- `main()`

Code Trace: `search(24)` and `search(44)`



Slot 3: another key (11) → i = 4.

Step 4 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

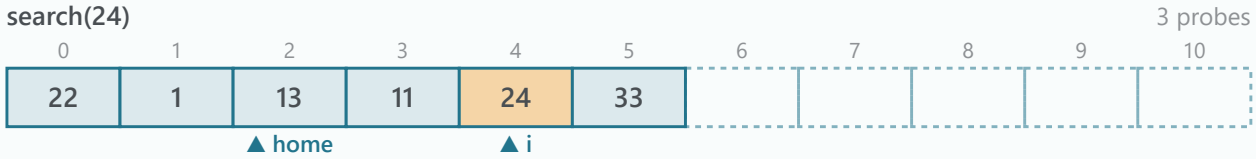
`k = 24` `i = 3` `d = 1`

`state = USED`

CALL STACK

- `search(24)`
- `main()`

Code Trace: `search(24)` and `search(44)`



Slot 4 is USED and holds 24 → **return 4** (3 probes).

Step 5 / 14

```
 1 int search(int k) {
 2   int i = k % M;
 3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
 ▶     if (table[i].state == EMPTY) return -1;
 ▶     if (table[i].state == USED &&
 ▶         table[i].key == k) return i;
 7   }
 8   return -1;
 9 }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

`k = 24` `i = 4` `d = 2` `state = USED`

CALL STACK

- `search(24)`
- `main()`

Code Trace: `search(24)` and `search(44)`



search(44) is called.

Step 6 / 14

```
▶ int search(int k) {
2   int i = k % M;
3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
4     if (table[i].state == EMPTY) return -1;
5     if (table[i].state == USED &&
6         table[i].key == k) return i;
7   }
8   return -1;
9 }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

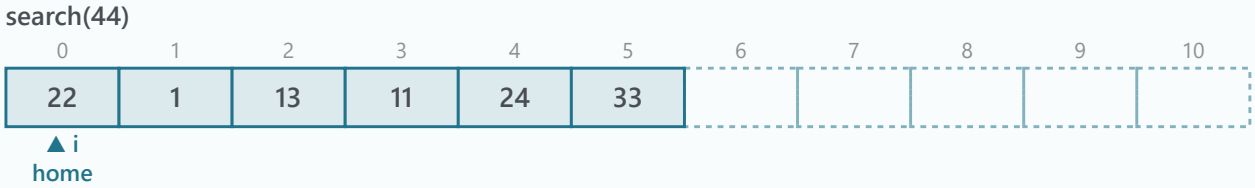
k = 44

CALL STACK

search(44)

main()

Code Trace: `search(24)` and `search(44)`



Home address i = 44 % 11 = **0**.

Step 7 / 14

```
1  int search(int k) {
▶    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state == EMPTY) return -1;
5      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

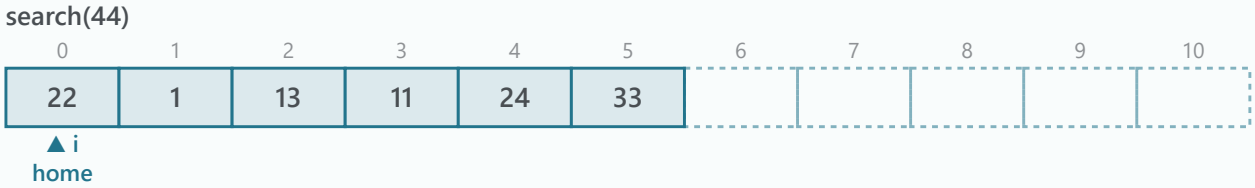
`k = 44` `i = 0`

CALL STACK

`search(44)`

`main()`

Code Trace: `search(24)` and `search(44)`



Slot 0: another key (22) → i = 1.

Step 8 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

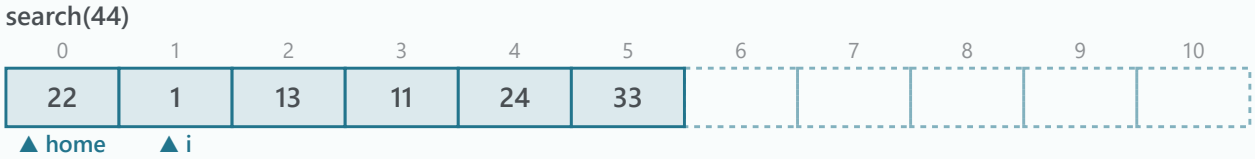
VARIABLES

`k = 44` `i = 0` `d = 0` `state = USED`

CALL STACK

- `search(44)`
- `main()`

Code Trace: `search(24)` and `search(44)`



Slot 1: another key (1) → i = 2.

Step 9 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

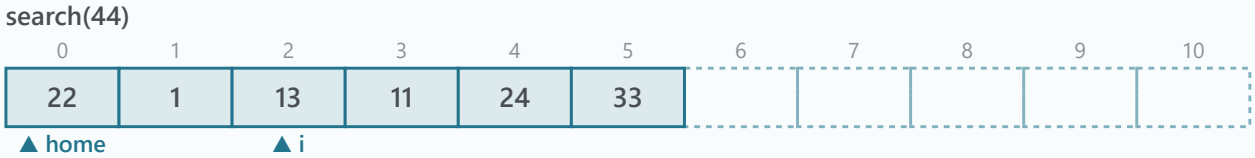
VARIABLES

`k = 44` `i = 1` `d = 1` `state = USED`

CALL STACK

- `search(44)`
- `main()`

Code Trace: `search(24)` and `search(44)`



Slot 2: another key (13) → i = 3.

Step 10 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

`k = 44` `i = 2` `d = 2`

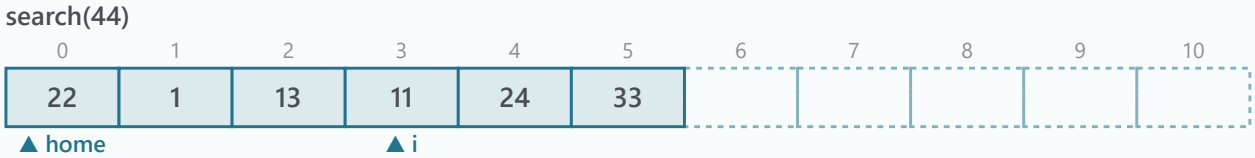
`state = USED`

CALL STACK

`search(44)`

`main()`

Code Trace: `search(24)` and `search(44)`



Slot 3: another key (11) → i = 4.

Step 11 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

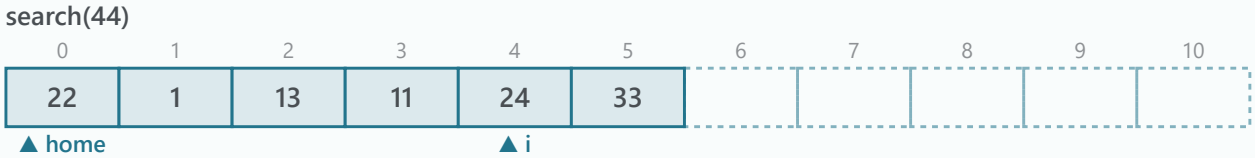
`k = 44` `i = 3` `d = 3`

`state = USED`

CALL STACK

- `search(44)`
- `main()`

Code Trace: `search(24)` and `search(44)`



Slot 4: another key (24) → i = 5.

Step 12 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

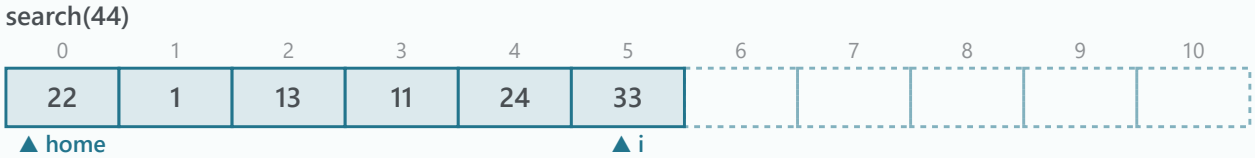
VARIABLES

`k = 44` `i = 4` `d = 4` `state = USED`

CALL STACK

- `search(44)`
- `main()`

Code Trace: `search(24)` and `search(44)`



Slot 5: another key (33) → i = 6.

Step 13 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

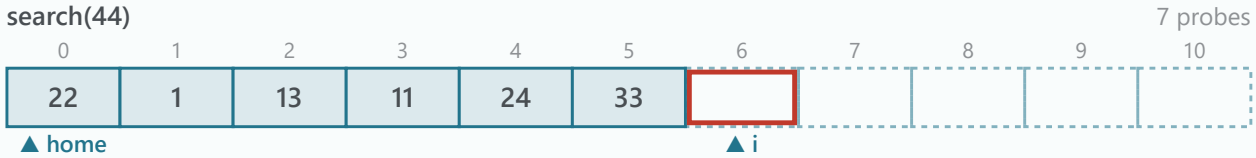
VARIABLES

`k = 44` `i = 5` `d = 5` `state = USED`

CALL STACK

- `search(44)`
- `main()`

Code Trace: `search(24)` and `search(44)`



Slot 6 is **EMPTY**: had 44 been inserted, it would be here at the latest → **return −1** (absent).

Step 14 / 14

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
5      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

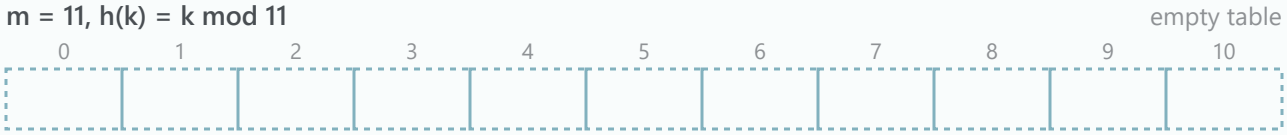
VARIABLES

`k = 44` `i = 6` `d = 6` `state = EMPTY`

CALL STACK

- `search(44)`
- `main()`

Quadratic Probing: $h(k, i) = (h(k) + I^2) \bmod m$

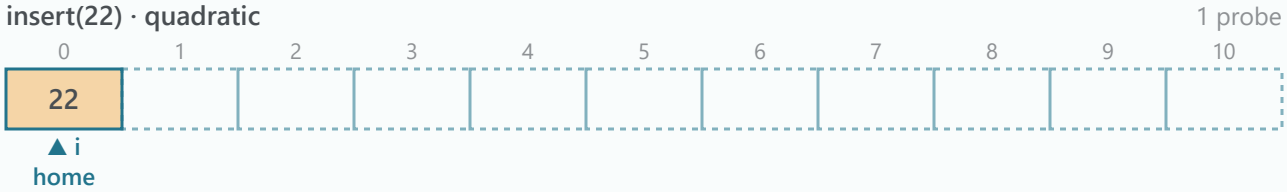


Same keys: 22, 1, 13, 11, 24, 33. Probes jump 1, 4, 9, 16… away from the home address.

Step 1 / 13

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

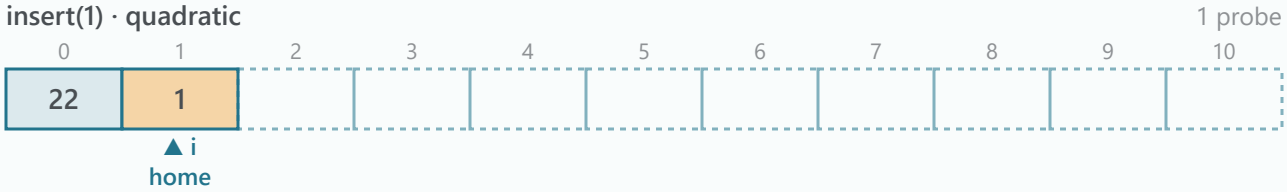
Quadratic Probing: h(k, i) = (h(k) + I^2) mod m



22 → home 0: **slot 0**.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and α ≤ 1/2, an empty slot is guaranteed to be found.

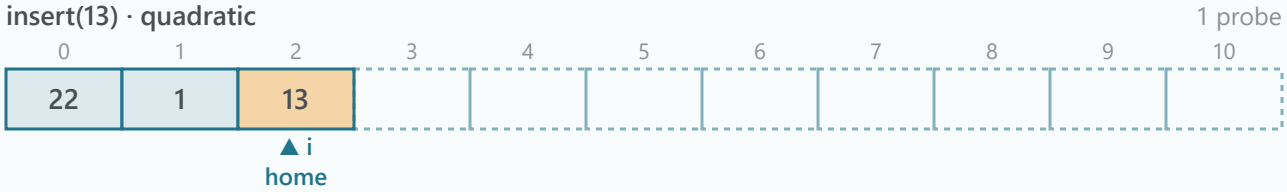
Quadratic Probing: $h(k, i) = (h(k) + I^2) \bmod m$



1 → home 1: **slot 1**.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

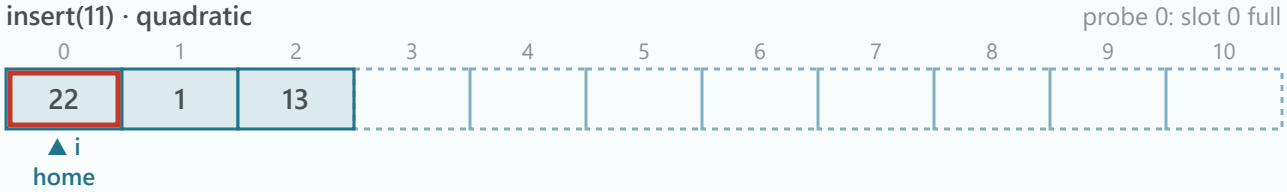
Quadratic Probing: h(k, i) = (h(k) + I^2) mod m



13 → home 2: **slot 2**.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and α ≤ 1/2, an empty slot is guaranteed to be found.

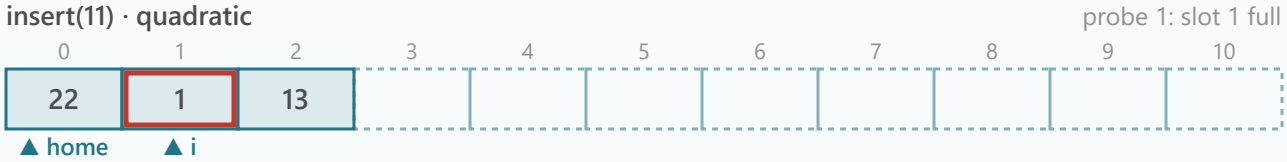
Quadratic Probing: $h(k, i) = (h(k) + i^2) \bmod m$



11: slot **0** is full (22) → try the next slot.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

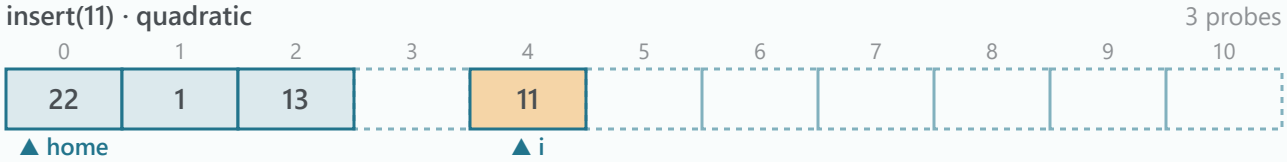
Quadratic Probing: $h(k, i) = (h(k) + I^2) \bmod m$



11: slot **1** is full (1) → try the next slot.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

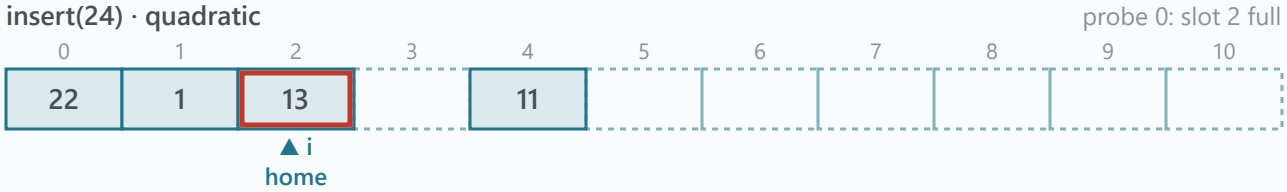
Quadratic Probing: h(k, i) = (h(k) + I^2) mod m



11 → home 0, probe sequence 0, 1, 4: **slot 4**.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and α ≤ 1/2, an empty slot is guaranteed to be found.

Quadratic Probing: $h(k, i) = (h(k) + i^2) \bmod m$

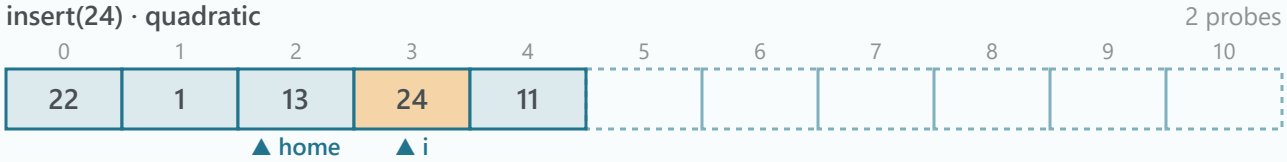


24: slot **2** is full (13) → try the next slot.

Step 8 / 13

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

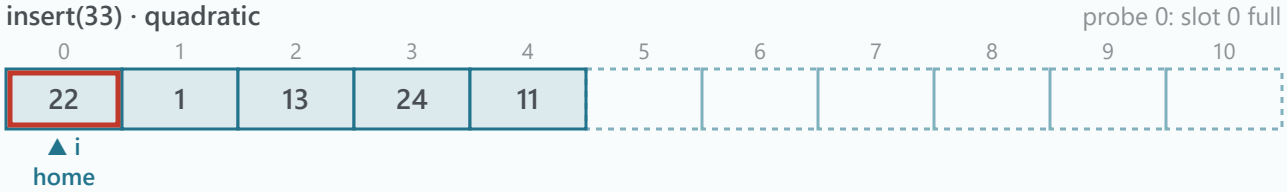
Quadratic Probing: $h(k, i) = (h(k) + i^2) \bmod m$



24 → home 2, probe sequence 2, 3: **slot 3**.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

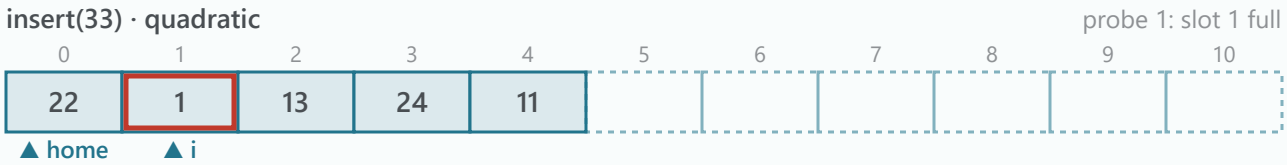
Quadratic Probing: $h(k, i) = (h(k) + i^2) \bmod m$



33: slot **0** is full (22) → try the next slot.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

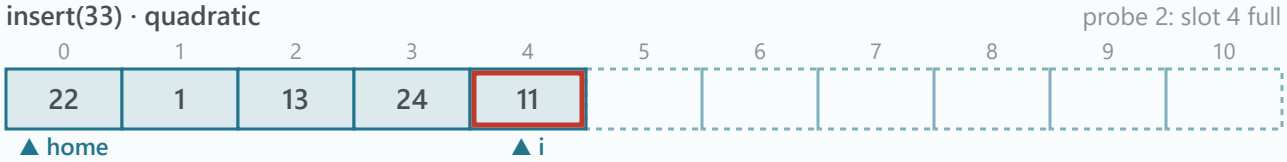
Quadratic Probing: $h(k, i) = (h(k) + I^2) \bmod m$



33: slot **1** is full (1) → try the next slot.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

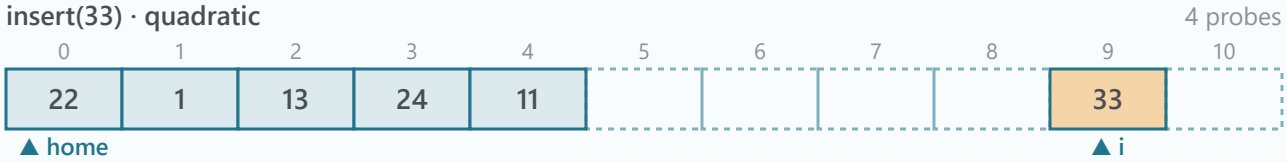
Quadratic Probing: $h(k, i) = (h(k) + I^2) \bmod m$



33: slot **4** is full (11) → try the next slot.

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \leq 1/2$, an empty slot is guaranteed to be found.

Quadratic Probing: $h(k, i) = (h(k) + i^2) \bmod m$

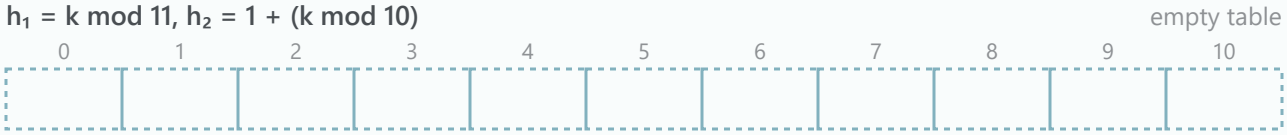


33 → home 0, probe sequence 0, 1, 4, 9: **slot 9**.

Step 13 / 13

Primary clustering goes away; keys with the same home address still follow the same sequence (**secondary clustering**). If m is prime and $\alpha \le 1/2$, an empty slot is guaranteed to be found.

Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

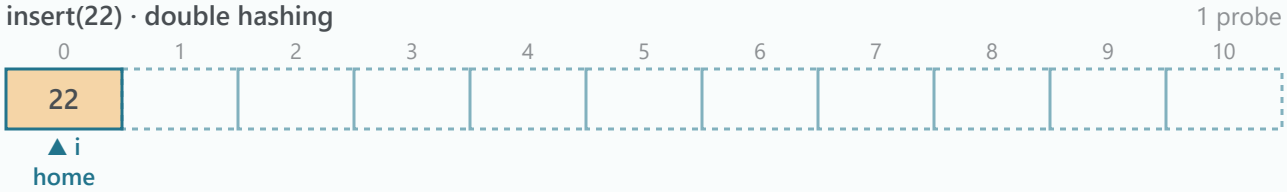


The step size depends on the key: 11 → $h_2 = 2$, 24 → $h_2 = 5$, 33 → $h_2 = 4$.

Step 1 / 12

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

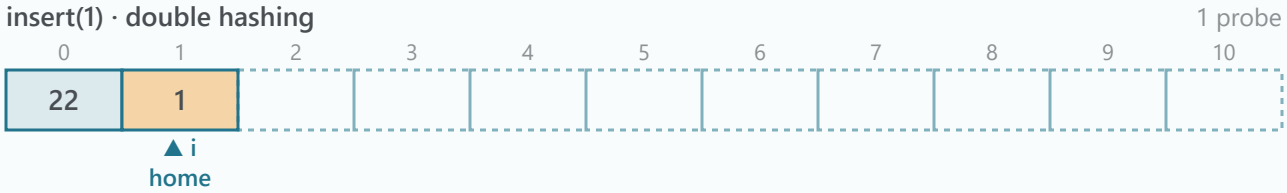


22 → home 0: **slot 0**.

Step 2 / 12

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

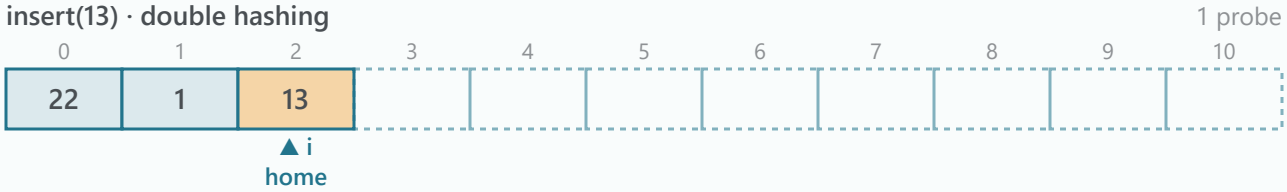
Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



1 → home 1: **slot 1**.

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

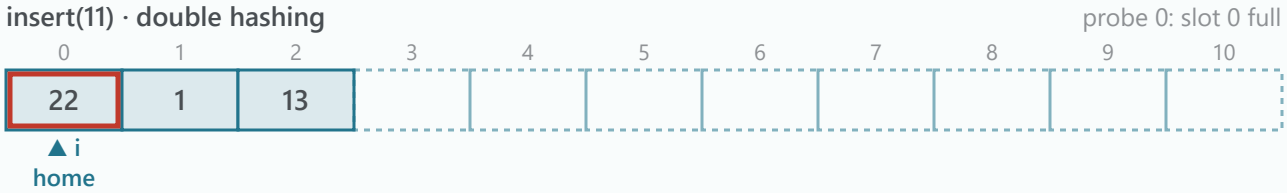


13 → home 2: **slot 2**.

Step 4 / 12

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

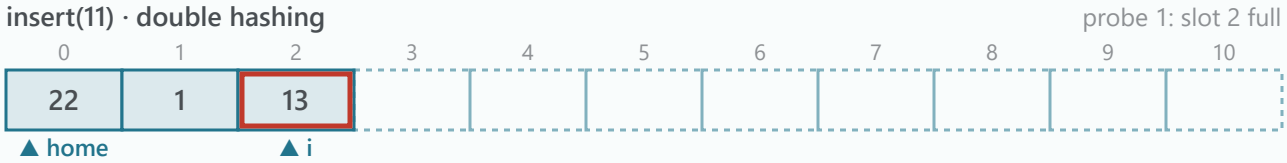
Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



11: slot **0** is full (22) → try the next slot.

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

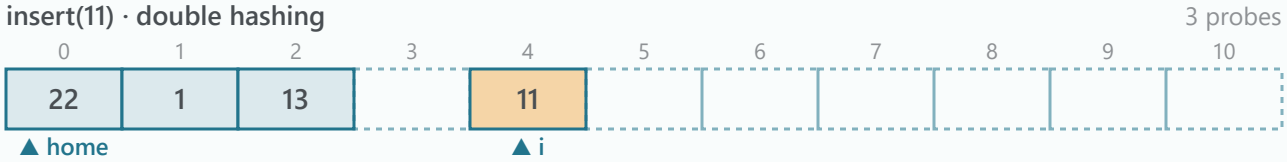
Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



11: slot **2** is full (13) → try the next slot.

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

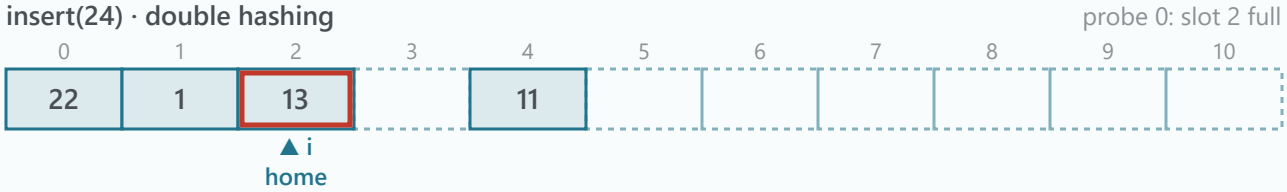
Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



11 → home 0, probe sequence 0, 2, 4: **slot 4**.

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

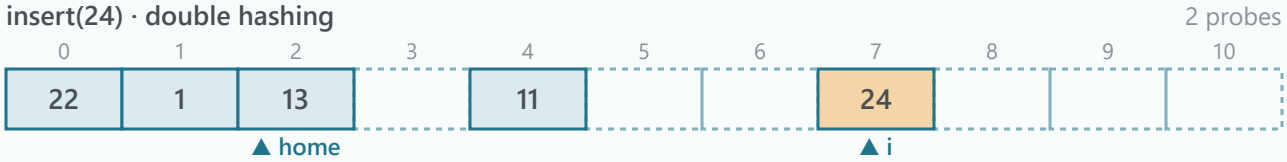


24: slot **2** is full (13) → try the next slot.

Step 8 / 12

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

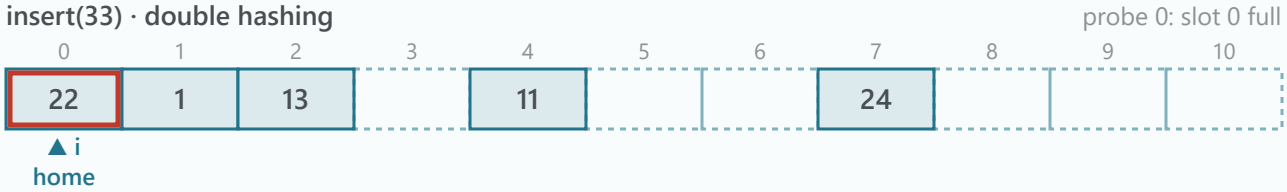
Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



24 → home 2, probe sequence 2, 7: **slot 7**.

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

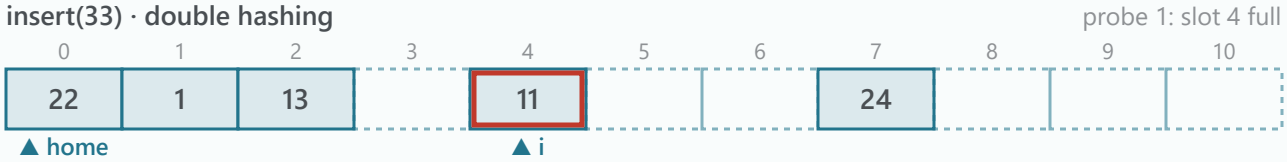
Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



33: slot **0** is full (22) → try the next slot.

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

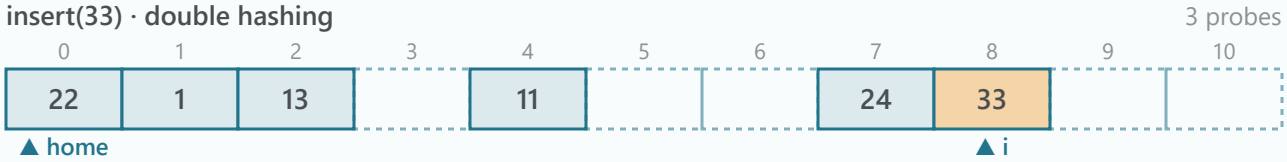
Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



33: slot **4** is full (11) → try the next slot.

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



33 → home 0, probe sequence 0, 4, 8: **slot 8**.

Step 12 / 12

$h_2(k)$ must never be 0 and must be coprime to m (automatic if m is prime). The probe sequence is specific to the key, so clustering is minimal.

One Table, Three Probing Methods: Where Does 33 Land?

Method	probe i	probe sequence of 33	Probes
Linear	$h + i$	0, 1, 2, 3, 4, 5	6
Quadratic	$h + i^2$	0, 1, 4, 9	4
Double hashing	$h_1 + i \cdot h_2$	0, 4, 8	3

Method	Clustering	Distinct probe sequences
Linear	primary (blocks)	m
Quadratic	secondary (same home address)	m
Double hashing	minimal	$\approx m^2$

Linear probing is cache-friendly and very fast while α is small; as α grows, clustering makes the cost climb quickly.

Separate Chaining: One List per bucket

```c
#define MZ 5
typedef struct node {
  int key;
  struct node *next;
} node;
node *bucket[MZ];           /* all NULL */
```

```c
bool chain_insert(int k) {
  int i = k % MZ;
  node *np = malloc(sizeof *np);
  if (np == NULL) return false;  /* no memory */
  np->key = k;
  np->next = bucket[i];          /* at the head */
  bucket[i] = np;
  return true;
}
```

Insert at the head

The order in the list does not matter: inserting at the head is O(1). (If duplicates are not allowed, search first.)

Search

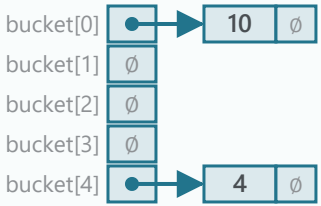
Only the list of its own bucket is scanned: 1 + α/2 nodes on average.

Easy deletion

Deleting from a linked list is enough; no tombstone is needed.

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(12)



chain_insert(12).

Step 1 / 15

```
▶ bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 12`

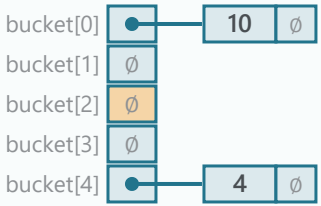
CALL STACK

`chain_insert(12)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(12)



i = 12 % 5 = **2**.

Step 2 / 15

```
1 bool chain_insert(int k) {
▶   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 12` `i = 2`

CALL STACK

`chain_insert(12)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(12)



Memory for a new node is allocated; it is not `NULL`, so continue.

Step 3 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
▶   node *np = malloc(sizeof *np);
▶   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];         /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 12` `i = 2` `np = &node(?)`

CALL STACK

`chain_insert(12)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)



key = 12; **next = bucket[2]** (NULL): the new node points to the old head.

Step 4 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
▶   np->key = k;
▶   np->next = bucket[i];         /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 12` `i = 2` `np = &node(12)`

`np→next = NULL`

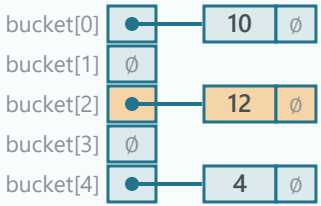
CALL STACK

`chain_insert(12)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(12)



bucket[2] = np: 12 is at the head of the list. O(1).

Step 5 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
▶   bucket[i] = np;
▶   return true;
9 }
```

VARIABLES

`k = 12` `i = 2` `np = &node(12)`

`bucket[i] = &node(12)`

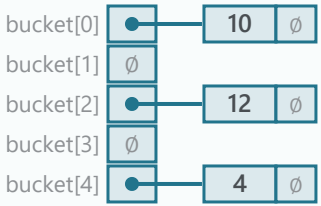
CALL STACK

`chain_insert(12)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(7)



chain_insert(7).

Step 6 / 15

```
▶ bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 7`

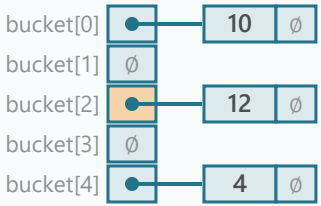
CALL STACK

`chain_insert(7)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(7)



i = 7 % 5 = **2**.

Step 7 / 15

```
1 bool chain_insert(int k) {
▶   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 7` `i = 2`

CALL STACK

`chain_insert(7)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(7)



Memory for a new node is allocated; it is not `NULL`, so continue.

Step 8 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
▶   node *np = malloc(sizeof *np);
▶   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];         /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 7` `i = 2` `np = &node(?)`

CALL STACK

- `chain_insert(7)`
- `main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(7)



key = 7; **next = bucket[2]** (&node(12)): the new node points to the old head.

Step 9 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
▶   np->key = k;
▶   np->next = bucket[i];         /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 7` `i = 2` `np = &node(7)`

`np→next = &node(12)`

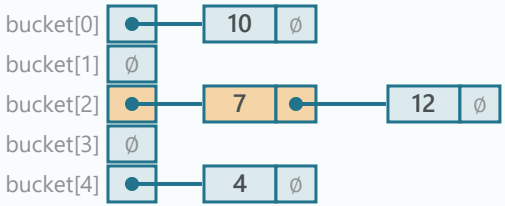
CALL STACK

`chain_insert(7)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(7)



bucket[2] = np: 7 is at the head of the list. O(1).

Step 10 / 15

```
1  bool chain_insert(int k) {
2    int i = k % MZ;
3    node *np = malloc(sizeof *np);
4    if (np == NULL) return false;  /* no memory */
5    np->key = k;
6    np->next = bucket[i];          /* at the head */
▶    bucket[i] = np;
▶    return true;
9  }
```

VARIABLES

`k = 7` `i = 2` `np = &node(7)`

`bucket[i] = &node(7)`

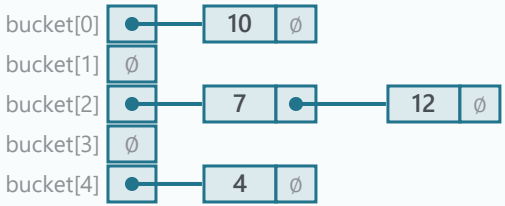
CALL STACK

`chain_insert(7)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(22)



chain_insert(22).

Step 11 / 15

```
▶ bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 22`

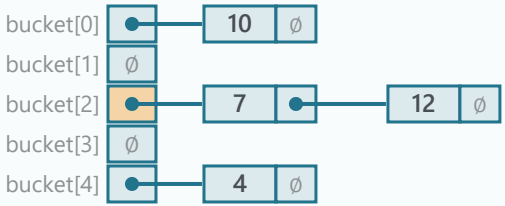
CALL STACK

`chain_insert(22)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(22)



i = 22 % 5 = **2**.

Step 12 / 15

```
1 bool chain_insert(int k) {
▶   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 22` `i = 2`

CALL STACK

`chain_insert(22)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)



Memory for a new node is allocated; it is not `NULL`, so continue.

Step 13 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
▶   node *np = malloc(sizeof *np);
▶   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];         /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 22` `i = 2` `np = &node(?)`

CALL STACK

`chain_insert(22)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)



key = 22; **next = bucket[2]** (&node(7)): the new node points to the old head.

Step 14 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
▶   np->key = k;
▶   np->next = bucket[i];         /* at the head */
7   bucket[i] = np;
8   return true;
9 }
```

VARIABLES

`k = 22` `i = 2` `np = &node(22)` `np→next = &node(7)`

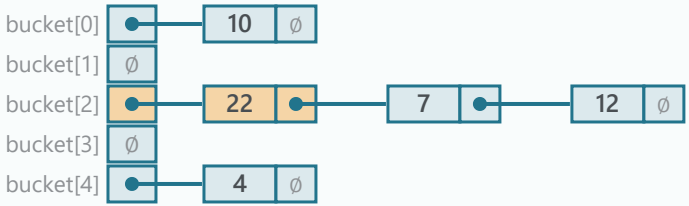
CALL STACK

`chain_insert(22)`

`main()`

Code Trace: `chain_insert` 12, 7, 22 (All in bucket 2)

insert(22)



bucket[2] = np: 22 is at the head of the list. O(1).

Step 15 / 15

```
1 bool chain_insert(int k) {
2   int i = k % MZ;
3   node *np = malloc(sizeof *np);
4   if (np == NULL) return false;  /* no memory */
5   np->key = k;
6   np->next = bucket[i];          /* at the head */
▶   bucket[i] = np;
▶   return true;
9 }
```

VARIABLES

`k = 22` `i = 2` `np = &node(22)`

`bucket[i] = &node(22)`

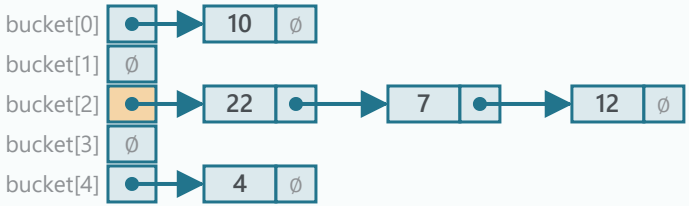
CALL STACK

`chain_insert(22)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



chain_search(12): bucket 12 % 5 = 2.

Step 1 / 18

```
▶ node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 12`

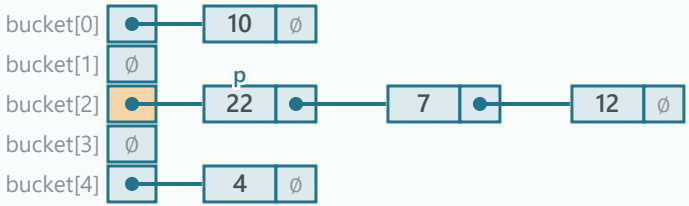
CALL STACK

`chain_search(12)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



p = bucket[2]: the head of the list (22).

Step 2 / 18

```
1 node *chain_search(int k) {
▶   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 12` `p = &node(22)`

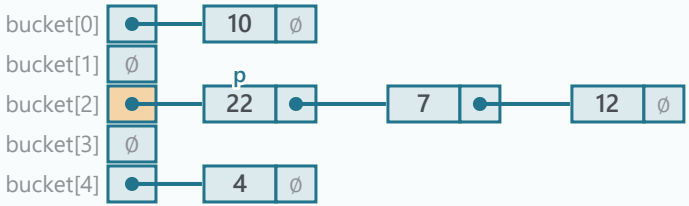
CALL STACK

`chain_search(12)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



p->key = 22 ≠ 12 → move on.

Step 3 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
▶   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 12` `p = &node(22)`

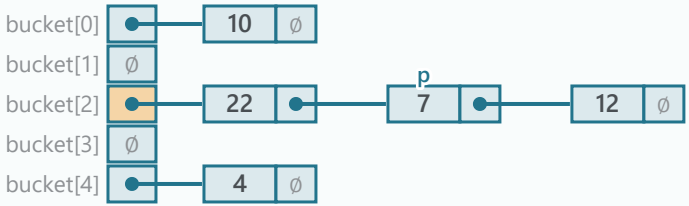
CALL STACK

`chain_search(12)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



p = p->next → node(7).

Step 4 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
      p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 12` `p = &node(7)`

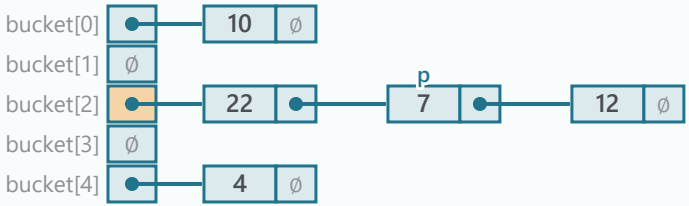
CALL STACK

`chain_search(12)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



p->key = 7 ≠ 12 → move on.

Step 5 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
▶   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 12` `p = &node(7)`

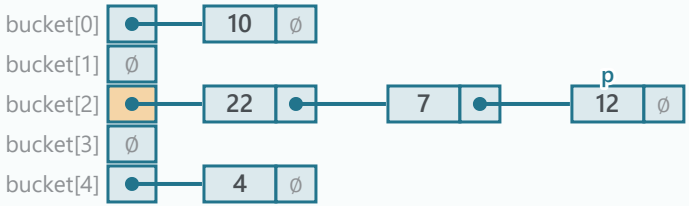
CALL STACK

`chain_search(12)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



p = p->next → node(12).

Step 6 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
▶     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 12` `p = &node(12)`

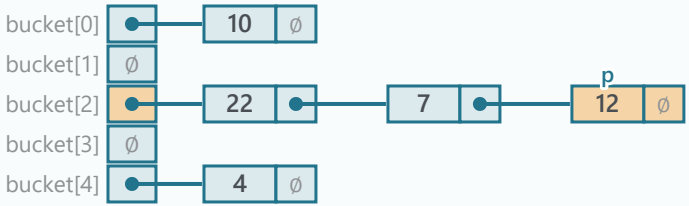
CALL STACK

`chain_search(12)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



p->key == 12: the loop condition is false; stop.

Step 7 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
▶   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

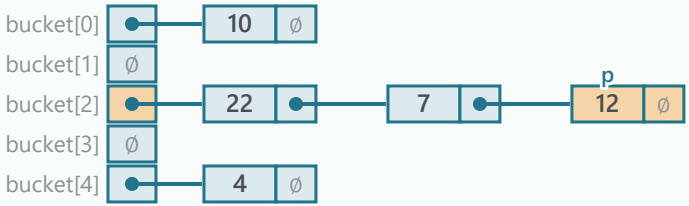
`k = 12` `p = &node(12)`

CALL STACK

- `chain_search(12)`
- `main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(12)



return p: found (3 nodes visited).

Step 8 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
4     p = p->next;
▶   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 12` `p = &node(12)`

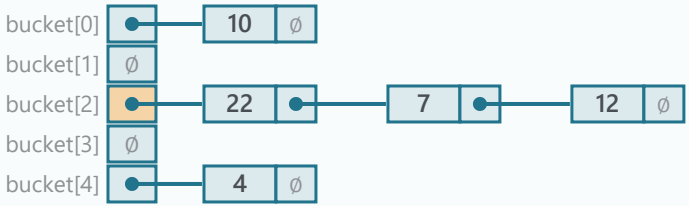
CALL STACK

`chain_search(12)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



chain_search(17): bucket 17 % 5 = 2.

Step 9 / 18

```
▶ node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 17`

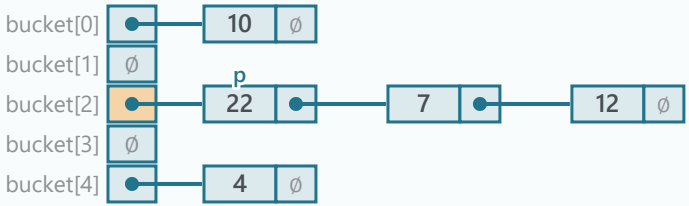
CALL STACK

`chain_search(17)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p = bucket[2]: the head of the list (22).

Step 10 / 18

```
1 node *chain_search(int k) {
▶   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 17` `p = &node(22)`

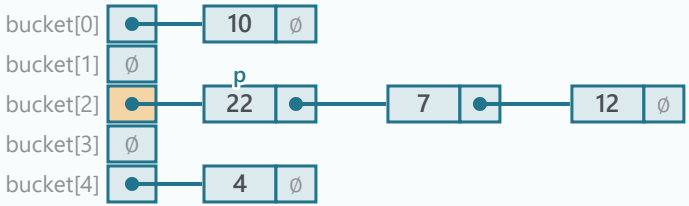
CALL STACK

`chain_search(17)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p->key = 22 ≠ 17 → move on.

Step 11 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
▶   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

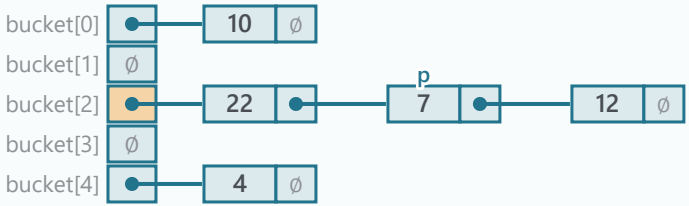
`k = 17` `p = &node(22)`

CALL STACK

- `chain_search(17)`
- `main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p = p->next → node(7).

Step 12 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
▶     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 17` `p = &node(7)`

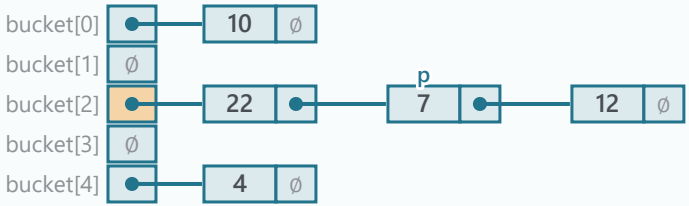
CALL STACK

`chain_search(17)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p->key = 7 ≠ 17 → move on.

Step 13 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
▶   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

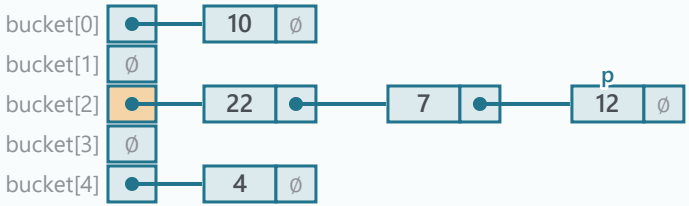
`k = 17` `p = &node(7)`

CALL STACK

- `chain_search(17)`
- `main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p = p->next → node(12).

Step 14 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
      p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 17` `p = &node(12)`

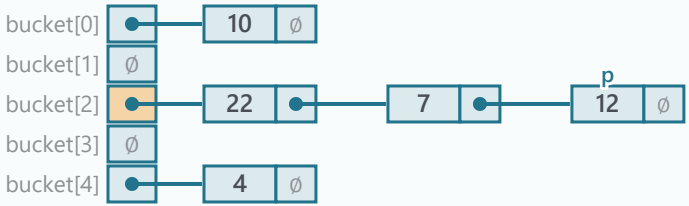
CALL STACK

`chain_search(17)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p->key = 12 ≠ 17 → move on.

Step 15 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
▶   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

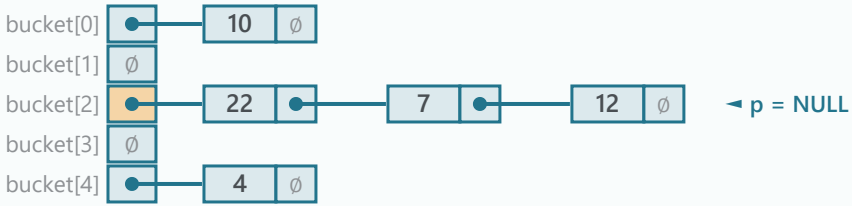
`k = 17` `p = &node(12)`

CALL STACK

- `chain_search(17)`
- `main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p = p->next → NULL.

Step 16 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
      p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 17` `p = NULL`

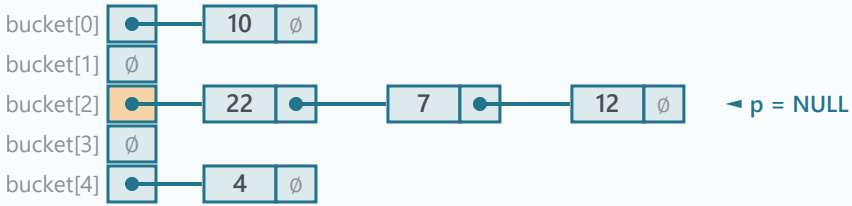
CALL STACK

`chain_search(17)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



p == NULL: end of the list; leave the loop.

Step 17 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
▶   while (p != NULL && p->key != k)
4     p = p->next;
5   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 17` `p = NULL`

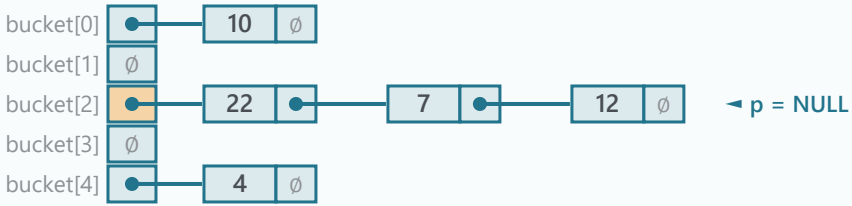
CALL STACK

`chain_search(17)`

`main()`

Code Trace: `chain_search(12)` and `chain_search(17)`

search(17)



return NULL: 17 is not in the table. The whole chain (3 nodes) was scanned.

Step 18 / 18

```
1 node *chain_search(int k) {
2   node *p = bucket[k % MZ];
3   while (p != NULL && p->key != k)
4     p = p->next;
▶   return p;                  /* or NULL */
6 }
```

VARIABLES

`k = 17` `p = NULL`

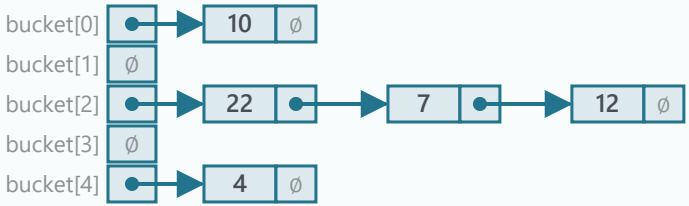
CALL STACK

`chain_search(17)`

`main()`

Code Trace: `chain_erase(7)`

erase(7)



chain_erase(7): 7 is in the middle of the chain (22 → 7 → 12).

Step 1 / 10

```
▶ bool chain_erase(int k) {
2   int i = k % MZ;
3   node *p = bucket[i], *prev = NULL;
4   while (p != NULL && p->key != k) {
5     prev = p;
6     p = p->next;
7   }
8   if (p == NULL) return false;        /* absent */
9   if (prev == NULL) bucket[i] = p->next;
10  else prev->next = p->next;
11  free(p);
12  return true;
13 }
```

VARIABLES

`k = 7`

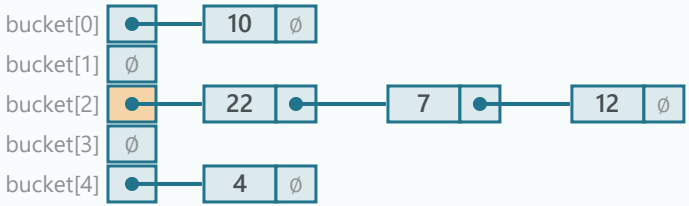
CALL STACK

`chain_erase(7)`

`main()`

Code Trace: `chain_erase(7)`

erase(7)



i = 7 % 5 = **2**.

Step 2 / 10

```
 1 bool chain_erase(int k) {
 2   int i = k % MZ;
 3   node *p = bucket[i], *prev = NULL;
 4   while (p != NULL && p->key != k) {
 5     prev = p;
 6     p = p->next;
 7   }
 8   if (p == NULL) return false;        /* absent */
 9   if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
11   free(p);
12   return true;
13 }
```

VARIABLES

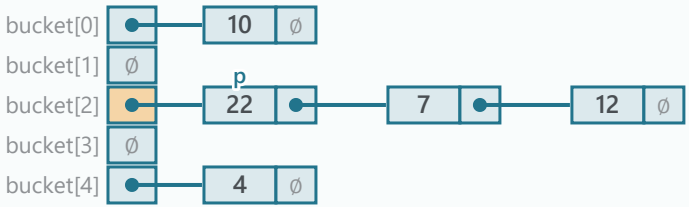
`k = 7` `i = 2`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`

erase(7)



p is at the head of the list; prev = NULL.

Step 3 / 10

```
1  bool chain_erase(int k) {
2    int i = k % MZ;
▶    node *p = bucket[i], *prev = NULL;
4    while (p != NULL && p->key != k) {
5      prev = p;
6      p = p->next;
7    }
8    if (p == NULL) return false;        /* absent */
9    if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
11   free(p);
12   return true;
13 }
```

VARIABLES

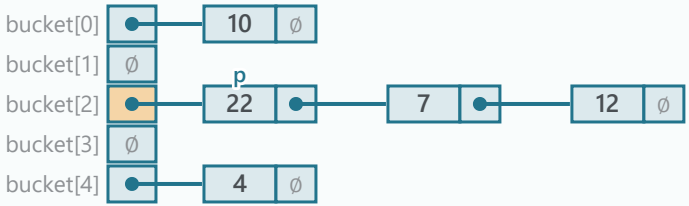
`k = 7` `i = 2` `p = &node(22)` `prev = NULL`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`

erase(7)



22 ≠ 7 → enter the loop.

Step 4 / 10

```
1  bool chain_erase(int k) {
2    int i = k % MZ;
3    node *p = bucket[i], *prev = NULL;
▶    while (p != NULL && p->key != k) {
5      prev = p;
6      p = p->next;
7    }
8    if (p == NULL) return false;        /* absent */
9    if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
11   free(p);
12   return true;
13 }
```

VARIABLES

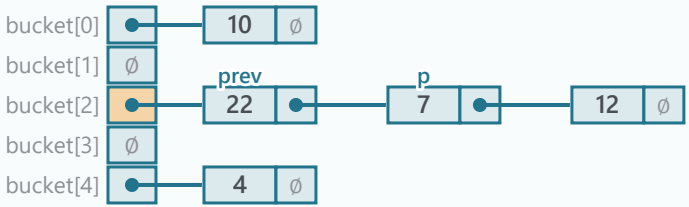
`k = 7` `i = 2` `p = &node(22)` `prev = NULL`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`

erase(7)



prev = 22's node, p = 7's node.

Step 5 / 10

```
1  bool chain_erase(int k) {
2    int i = k % MZ;
3    node *p = bucket[i], *prev = NULL;
4    while (p != NULL && p->key != k) {
▶      prev = p;
▶      p = p->next;
7    }
8    if (p == NULL) return false;        /* absent */
9    if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
11   free(p);
12   return true;
13 }
```

VARIABLES

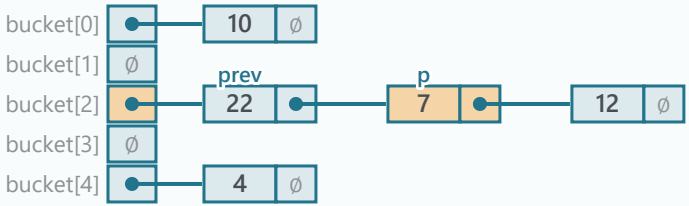
`k = 7` `i = 2` `p = &node(7)` `prev = &node(22)`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`

erase(7)



p->key == 7 → the loop ends.

Step 6 / 10

```
1  bool chain_erase(int k) {
2    int i = k % MZ;
3    node *p = bucket[i], *prev = NULL;
▶    while (p != NULL && p->key != k) {
5      prev = p;
6      p = p->next;
7    }
8    if (p == NULL) return false;        /* absent */
9    if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
11   free(p);
12   return true;
13 }
```

VARIABLES

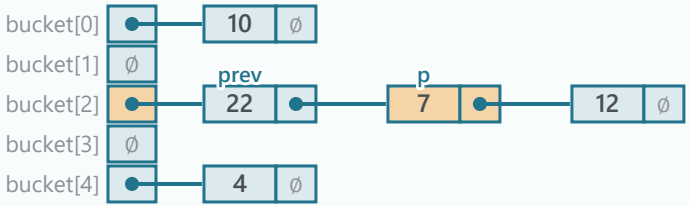
`k = 7` `i = 2` `p = &node(7)` `prev = &node(22)`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`

erase(7)



p is not NULL: the key was found.

Step 7 / 10

```
1  bool chain_erase(int k) {
2    int i = k % MZ;
3    node *p = bucket[i], *prev = NULL;
4    while (p != NULL && p->key != k) {
5      prev = p;
6      p = p->next;
7    }
▶    if (p == NULL) return false;        /* absent */
9    if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
11   free(p);
12   return true;
13 }
```

VARIABLES

`k = 7` `i = 2` `p = &node(7)` `prev = &node(22)`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`



prev is not NULL → **prev->next = p->next**: 22 now points straight to 12; 7 is out of the chain.


```
1  bool chain_erase(int k) {
2    int i = k % MZ;
3    node *p = bucket[i], *prev = NULL;
4    while (p != NULL && p->key != k) {
5      prev = p;
6      p = p->next;
7    }
8    if (p == NULL) return false;        /* absent */
▶    if (prev == NULL) bucket[i] = p->next;
▶    else prev->next = p->next;
11   free(p);
12   return true;
13 }
```

VARIABLES

`k = 7` `i = 2` `p = &node(7)`

`prev = &node(22)`

`prev→next = &node(12)`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`



free(p): the node's memory is released. p must not be used after this (dangling pointer).

Step 9 / 10

```
1  bool chain_erase(int k) {
2    int i = k % MZ;
3    node *p = bucket[i], *prev = NULL;
4    while (p != NULL && p->key != k) {
5      prev = p;
6      p = p->next;
7    }
8    if (p == NULL) return false;        /* absent */
9    if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
▶    free(p);
12   return true;
13 }
```

VARIABLES

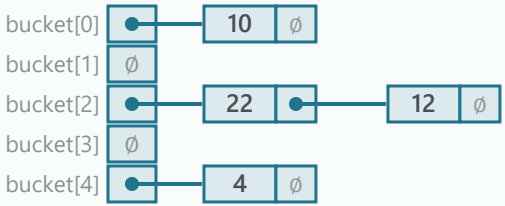
`k = 7` `i = 2` `p = (freed)` `prev = &node(22)`

CALL STACK

- `chain_erase(7)`
- `main()`

Code Trace: `chain_erase(7)`

erase(7)



return true. Had the head node been deleted (prev == NULL), `bucket[i]` would have been updated.

Step 10 / 10

```
1  bool chain_erase(int k) {
2    int i = k % MZ;
3    node *p = bucket[i], *prev = NULL;
4    while (p != NULL && p->key != k) {
5      prev = p;
6      p = p->next;
7    }
8    if (p == NULL) return false;        /* absent */
9    if (prev == NULL) bucket[i] = p->next;
10   else prev->next = p->next;
11   free(p);
▶    return true;
13 }
```

VARIABLES

`k = 7` `i = 2`

CALL STACK

- `chain_erase(7)`
- `main()`

The Deletion Trap: Emptying a Slot Cuts Searches Short

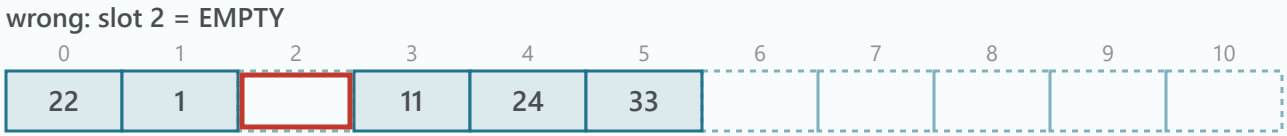


We will delete 13. 24's home address is 2; since slots 2 and 3 were full, 24 went to 4.

Step 1 / 5

In tables with many deletions tombstones pile up and searches get longer: rehash periodically (cleanup). Chaining does not have this problem.

The Deletion Trap: Emptying a Slot Cuts Searches Short

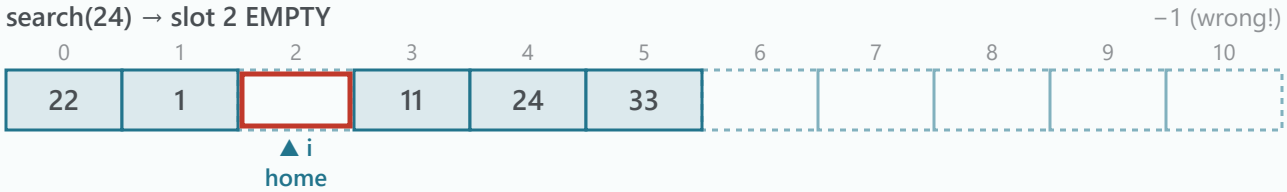


If we make the slot **EMPTY**...

Step 2 / 5

In tables with many deletions tombstones pile up and searches get longer: rehash periodically (cleanup). Chaining does not have this problem.

The Deletion Trap: Emptying a Slot Cuts Searches Short

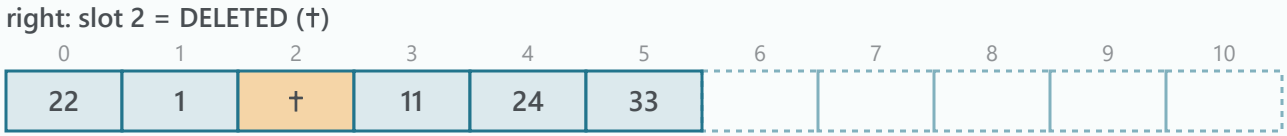


… `search(24)` sees an empty slot at the home address and reports “absent”. Yet 24 is in the table!

Step 3 / 5

In tables with many deletions tombstones pile up and searches get longer: rehash periodically (cleanup). Chaining does not have this problem.

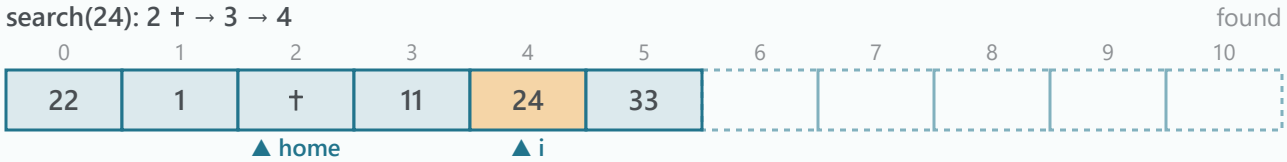
The Deletion Trap: Emptying a Slot Cuts Searches Short



The right way is to mark the slot with a **tombstone** (DELETED).

In tables with many deletions tombstones pile up and searches get longer: rehash periodically (cleanup). Chaining does not have this problem.

The Deletion Trap: Emptying a Slot Cuts Searches Short



Search does not treat a tombstone as empty and keeps going; insertion, however, may reuse a tombstone.

Step 5 / 5

In tables with many deletions tombstones pile up and searches get longer: rehash periodically (cleanup). Chaining does not have this problem.

Code Trace: `erase(13)`, Then `search(24)`



erase(13) is called; first its slot must be found.

Step 1 / 10

```
 1 int search(int k) {
 2   int i = k % M;
 3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
 4     if (table[i].state == EMPTY) return -1;
 5     if (table[i].state == USED &&
 6         table[i].key == k) return i;
 7   }
 8   return -1;
 9 }
▶ bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

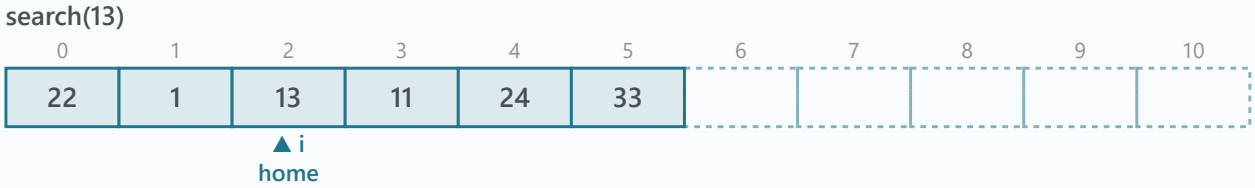
VARIABLES

`k = 13`

CALL STACK

- `erase(13)`
- `main()`

Code Trace: `erase(13)`, Then `search(24)`



Home address i = 13 % 11 = **2**.

Step 2 / 10

```
1  int search(int k) {
▶    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state == EMPTY) return -1;
5      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

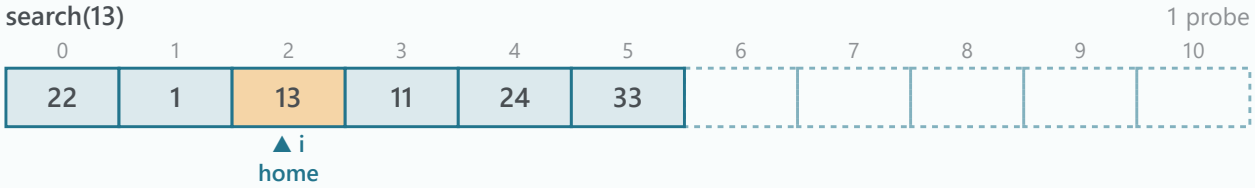
VARIABLES

`k = 13` `i = 2`

CALL STACK

- `search(13)`
- `erase(13)`
- `main()`

Code Trace: `erase(13)`, Then `search(24)`



Slot 2 is USED and holds 13 → **return 2** (1 probe).

Step 3 / 10

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
▶          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

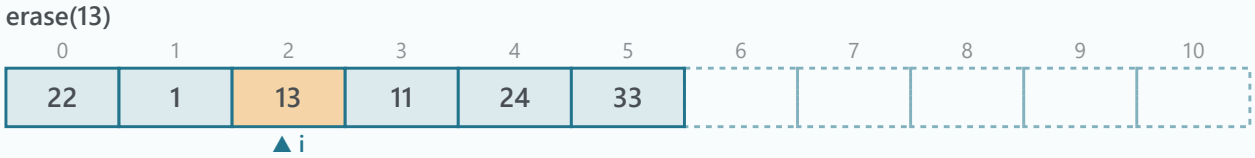
`k = 13` `i = 2` `d = 0`

`state = USED`

CALL STACK

- `search(13)`
- `erase(13)`
- `main()`

Code Trace: `erase(13)`, Then `search(24)`



`search` returned 2; i ≥ 0, so there is an item to delete.

Step 4 / 10

```
 1 int search(int k) {
 2   int i = k % M;
 3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
 4     if (table[i].state == EMPTY) return -1;
 5     if (table[i].state == USED &&
 6         table[i].key == k) return i;
 7   }
 8   return -1;
 9 }
10 bool erase(int k) {
▶  int i = search(k);
▶  if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

`k = 13` `i = 2`

CALL STACK

- **erase(13)**
- main()

Code Trace: `erase(13)`, Then `search(24)`



table[2].state = DELETED: a tombstone.

Step 5 / 10

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state == EMPTY) return -1;
5      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
▶    table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

`k = 13` `i = 2`

`table[i].state = DELETED`

CALL STACK

- **erase(13)**
- main()

Code Trace: `erase(13)`, Then `search(24)`



return true.

Step 6 / 10

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state == EMPTY) return -1;
5      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
▶    return true;
15 }
```

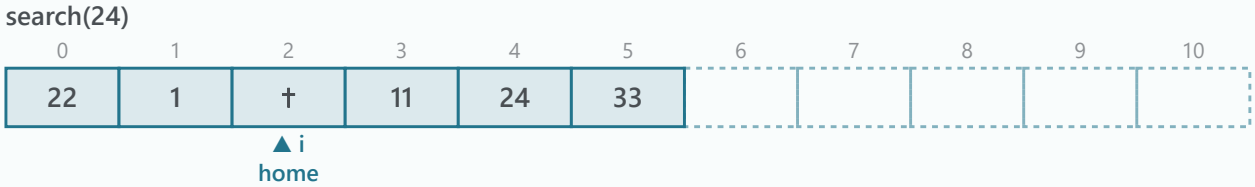
VARIABLES

`k = 13` `i = 2`

CALL STACK

- **erase(13)**
- main()

Code Trace: `erase(13)`, Then `search(24)`



Home address i = 24 % 11 = **2**.

Step 7 / 10

```
1  int search(int k) {
▶    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (table[i].state == EMPTY) return -1;
5      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

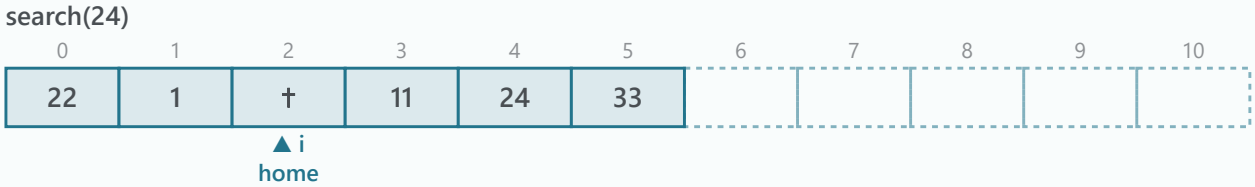
VARIABLES

`k = 24` `i = 2`

CALL STACK

- `search(24)`
- `main()`

Code Trace: `erase(13)`, Then `search(24)`



Slot 2 holds a tombstone: deleted, keep searching → i = 3.

Step 8 / 10

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

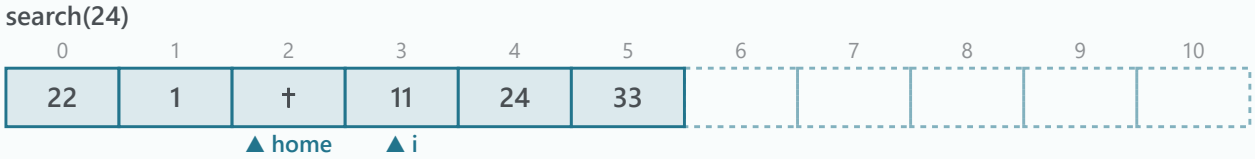
`k = 24` `i = 2` `d = 0`

`state = DELETED`

CALL STACK

- `search(24)`
- `main()`

Code Trace: `erase(13)`, Then `search(24)`



Slot 3: another key (11) → i = 4.

Step 9 / 10

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
6          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

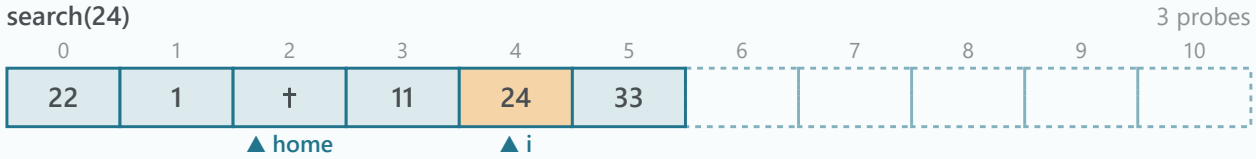
VARIABLES

`k = 24` `i = 3` `d = 1` `state = USED`

CALL STACK

- `search(24)`
- `main()`

Code Trace: `erase(13)`, Then `search(24)`



Slot 4 is USED and holds 24 → **return 4** (3 probes).

Step 10 / 10

```
1  int search(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (table[i].state == EMPTY) return -1;
▶      if (table[i].state == USED &&
▶          table[i].key == k) return i;
7    }
8    return -1;
9  }
10 bool erase(int k) {
11   int i = search(k);
12   if (i < 0) return false;
13   table[i].state = DELETED;   /* tombstone */
14   return true;
15 }
```

VARIABLES

`k = 24` `i = 4` `d = 2` `state = USED`

CALL STACK

- `search(24)`
- `main()`

Load Factor α = n/m and the Expected Number of Probes

α	Chain ✓	Chain ✗	Linear ✓	Linear ✗	Uniform* ✓	Uniform* ✗
0,25	1,13	1,25	1,2	1,4	1,15	1,3
0,5	1,25	1,50	1,5	2,5	1,39	2,0
0,75	1,38	1,75	2,5	8,5	1,85	4,0
0,9	1,45	1,90	5,5	50,5	2,56	10,0

✓ successful, ✗ unsuccessful search. Formulas: chaining $1 + \alpha/2$ and $1 + \alpha$ ($\Theta(1 + \alpha)$); linear probing (Knuth) $\frac{1}{2}(1 + 1/(1-\alpha))$ and $\frac{1}{2}(1 + 1/(1-\alpha)^2)$; * open addressing under uniform hashing $(1/\alpha)\cdot\ln(1/(1-\alpha))$ and $1/(1-\alpha)$ (CLRS).

If α is held constant, every operation is **O(1)** on average. Practical thresholds: α ≤ 0.5–0.7 for open addressing, α ≤ about 1 for chaining. Past the threshold the table grows.

Resizing (Rehashing)

```c
typedef struct {
  slot *a;      /* dynamic array of m slots */
  int m, n;     /* size, used slots */
} Table;
/* after inserting: */
if (2 * t->n > t->m)             /* α > 1/2 */
  resize(t, next_prime(2 * t->m));
```

Why rehash?

h(k) = k mod m depends on m: items land in different slots in the new table; copying is not enough.

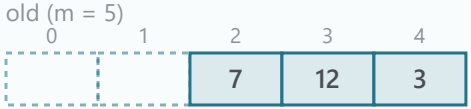
Amortized O(1)

If the size doubles each time, n insertions copy at most 2n items in total: constant per insertion.

Tombstones

Rehashing also clears out the tombstones.

Code Trace: `resize(t, 11)`



α = 0,6

n = 3, m = 5 → α = 0.6 > 0.5: the table grows to 11 slots (≈ 2m, prime).

Step 1 / 11

```
▶ bool resize(Table *t, int new_m) {
2   slot *old = t->a;
3   int old_m = t->m;
4   slot *a = calloc(new_m, sizeof *a); /* EMPTY */
5   if (a == NULL) return false;
6   t->a = a; t->m = new_m; t->n = 0;
7   for (int i = 0; i < old_m; i++)
8     if (old[i].state == USED)
9       table_insert(t, old[i].key);
10  free(old);
11  return true;
12 }
```

VARIABLES

`t→m = 5` `t→n = 3` `new_m = 11`

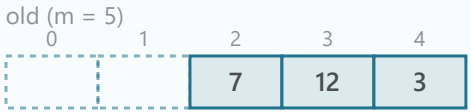
CALL STACK

`resize(t, 11)`

`main()`

RESIZING · RUN THE CODE STEP BY STEP

Code Trace: `resize(t, 11)`



The old array's address and size are saved.

Step 2 / 11

```
 1 bool resize(Table *t, int new_m) {
 2   slot *old = t->a;
 3   int old_m = t->m;
 4   slot *a = calloc(new_m, sizeof *a); /* EMPTY */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = new_m; t->n = 0;
 7   for (int i = 0; i < old_m; i++)
 8     if (old[i].state == USED)
 9       table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

VARIABLES

`old = &slot[5]` `old_m = 5` `new_m = 11`

CALL STACK

`resize(t, 11)`

`main()`

Code Trace: `resize(t, 11)`



`calloc` returns 11 zeroed slots: all EMPTY (= 0).

Step 3 / 11

```
1  bool resize(Table *t, int new_m) {
2    slot *old = t->a;
3    int old_m = t->m;
▶    slot *a = calloc(new_m, sizeof *a); /* EMPTY */
5    if (a == NULL) return false;
6    t->a = a; t->m = new_m; t->n = 0;
7    for (int i = 0; i < old_m; i++)
8      if (old[i].state == USED)
9        table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

VARIABLES

`old = &slot[5]` `old_m = 5`

`a = &slot[11]`

CALL STACK

`resize(t, 11)`

`main()`

Code Trace: `resize(t, 11)`



Memory allocated; the table is linked to the new array, n = 0.

Step 4 / 11

```
1  bool resize(Table *t, int new_m) {
2    slot *old = t->a;
3    int old_m = t->m;
4    slot *a = calloc(new_m, sizeof *a); /* EMPTY */
▶    if (a == NULL) return false;
▶    t->a = a; t->m = new_m; t->n = 0;
7    for (int i = 0; i < old_m; i++)
8      if (old[i].state == USED)
9        table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

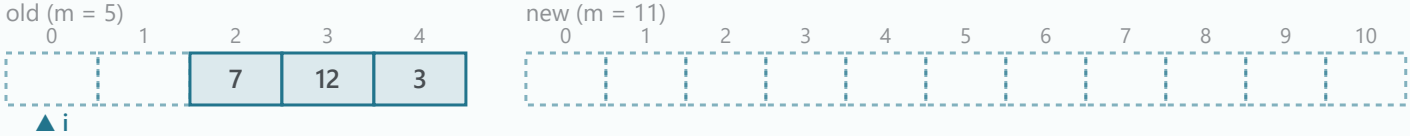
VARIABLES

`old_m = 5` `t→m = 11` `t→n = 0`

CALL STACK

- `resize(t, 11)`
- `main()`

Code Trace: `resize(t, 11)`



old[0] is EMPTY: skipped.

Step 5 / 11

```
1  bool resize(Table *t, int new_m) {
2    slot *old = t->a;
3    int old_m = t->m;
4    slot *a = calloc(new_m, sizeof *a); /* EMPTY */
5    if (a == NULL) return false;
6    t->a = a; t->m = new_m; t->n = 0;
▶    for (int i = 0; i < old_m; i++)
▶      if (old[i].state == USED)
9        table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

VARIABLES

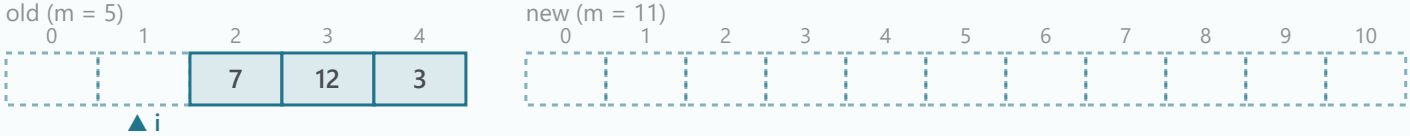
`i = 0` `old[i].state = EMPTY` `t->n = 0`

CALL STACK

`resize(t, 11)`

`main()`

Code Trace: `resize(t, 11)`



old[1] is EMPTY: skipped.

Step 6 / 11

```
1  bool resize(Table *t, int new_m) {
2    slot *old = t->a;
3    int old_m = t->m;
4    slot *a = calloc(new_m, sizeof *a); /* EMPTY */
5    if (a == NULL) return false;
6    t->a = a; t->m = new_m; t->n = 0;
▶    for (int i = 0; i < old_m; i++)
▶      if (old[i].state == USED)
9        table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

VARIABLES

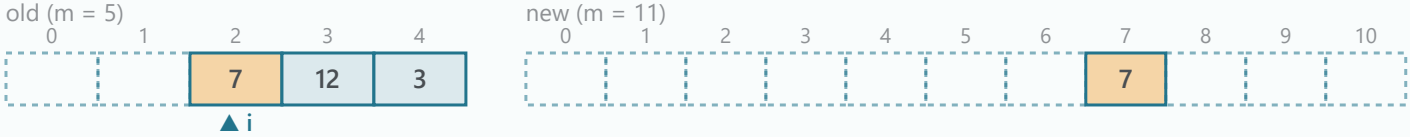
`i = 1` `old[i].state = EMPTY` `t->n = 0`

CALL STACK

`resize(t, 11)`

`main()`

Code Trace: `resize(t, 11)`



old[2] = 7 USED → rehash: 7 % 11 = 7 → slot **7**. (Its old home address was 7 % 5 = 2.)

Step 7 / 11

```
1  bool resize(Table *t, int new_m) {
2    slot *old = t->a;
3    int old_m = t->m;
4    slot *a = calloc(new_m, sizeof *a); /* EMPTY */
5    if (a == NULL) return false;
6    t->a = a; t->m = new_m; t->n = 0;
7    for (int i = 0; i < old_m; i++)
8      if (old[i].state == USED)
▶        table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

VARIABLES

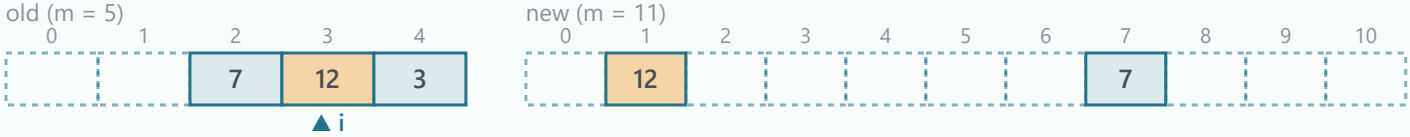
`i = 2` `old[i].key = 7` `t→n = 1`

CALL STACK

`resize(t, 11)`

`main()`

Code Trace: `resize(t, 11)`



old[3] = 12 USED → rehash: 12 % 11 = 1 → slot **1**. (Its old home address was 12 % 5 = 2.)

Step 8 / 11

```
1  bool resize(Table *t, int new_m) {
2    slot *old = t->a;
3    int old_m = t->m;
4    slot *a = calloc(new_m, sizeof *a); /* EMPTY */
5    if (a == NULL) return false;
6    t->a = a; t->m = new_m; t->n = 0;
7    for (int i = 0; i < old_m; i++)
8      if (old[i].state == USED)
▶        table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

VARIABLES

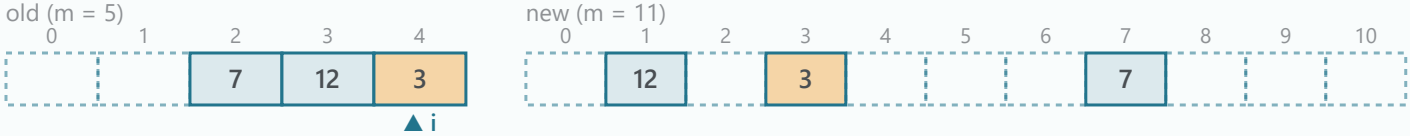
`i = 3` `old[i].key = 12` `t→n = 2`

CALL STACK

`resize(t, 11)`

`main()`

Code Trace: `resize(t, 11)`



old[4] = 3 USED → rehash: 3 % 11 = 3 → slot **3**. (Its old home address was 3 % 5 = 3.)

Step 9 / 11

```
 1 bool resize(Table *t, int new_m) {
 2   slot *old = t->a;
 3   int old_m = t->m;
 4   slot *a = calloc(new_m, sizeof *a); /* EMPTY */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = new_m; t->n = 0;
 7   for (int i = 0; i < old_m; i++)
 8     if (old[i].state == USED)
 ▶       table_insert(t, old[i].key);
10   free(old);
11   return true;
12 }
```

VARIABLES

`i = 4` `old[i].key = 3` `t->n = 3`

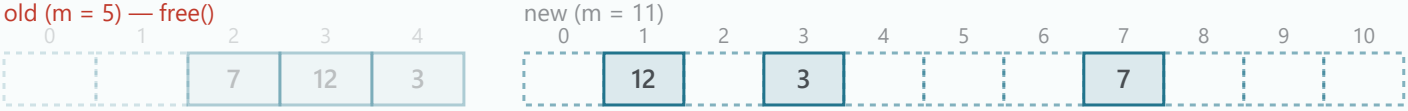
CALL STACK

`resize(t, 11)`

`main()`

RESIZING · RUN THE CODE STEP BY STEP

Code Trace: `resize(t, 11)`



The old array is **free**d. New α = 3/11 ≈ 0.27.

Step 10 / 11

```
1  bool resize(Table *t, int new_m) {
2    slot *old = t->a;
3    int old_m = t->m;
4    slot *a = calloc(new_m, sizeof *a); /* EMPTY */
5    if (a == NULL) return false;
6    t->a = a; t->m = new_m; t->n = 0;
7    for (int i = 0; i < old_m; i++)
8      if (old[i].state == USED)
9        table_insert(t, old[i].key);
▶    free(old);
11   return true;
12 }
```

VARIABLES

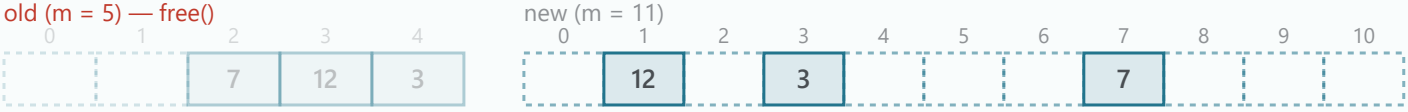
`t→m = 11` `t→n = 3`

CALL STACK

`resize(t, 11)`

`main()`

Code Trace: `resize(t, 11)`



α ≈ 0,27

return true. Every item was reinserted: O(n + m).

Step 11 / 11

```
 1 bool resize(Table *t, int new_m) {
 2   slot *old = t->a;
 3   int old_m = t->m;
 4   slot *a = calloc(new_m, sizeof *a); /* EMPTY */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = new_m; t->n = 0;
 7   for (int i = 0; i < old_m; i++)
 8     if (old[i].state == USED)
 9       table_insert(t, old[i].key);
10   free(old);
▶    return true;
12 }
```

VARIABLES

`t→m = 11` `t→n = 3`

CALL STACK

- **`resize(t, 11)`**
- `main()`

Lab: Open Addressing

22, 1, 13, 11, 24, 33 and searches

Input (number = insert, find 24, delete 13) `22, 1, 13, 11, 24, 33, find 24, find 44` Apply

CODE: insert

```
 1 int insert(int k) {
 2   int i = k % M; /* home addr */
 3   for (int d = 0; d < M;
 4        d++, i = (i + 1) % M) {
 5     if (table[i].state != USED) {
 6       table[i].key = k;
 7       table[i].state = USED;
 8       return i;
 9     }
10   }
11   return -1; /* table full */
12 }
```



`n 0`

OUTPUT (none yet)

`M = 11` , `h(k) = k % 11` . If a slot is occupied, the next slot is tried (linear probing). † is a tombstone (deleted mark): a search does not stop there, but an insertion may reuse the slot.

SUMMARY

Chaining or Open Addressing?

Chaining

+ $\alpha > 1$ is possible; it never fills up, it just slows down.
+ Deletion is simple, no tombstones.
+ More robust against a poor hash.
+ A pointer plus `malloc` cost per node; not cache-friendly.

Open addressing

– No extra pointers; contiguous memory, cache-friendly.
– $\alpha \leq 1$; the cost explodes as $\alpha \to 1$.
– Deletion needs tombstones.
– Needs good probing (double hashing) against clustering.

Rule of thumb: if the item count is unpredictable and deletions are frequent, chain; if memory and speed are critical, deletions are rare and α can be kept low, use open addressing.

SUMMARY

Complexity Summary

Operation	Average (α constant)	Worst	Note
search	O(1)	O(n)	if every key lands in the same bucket
insert	O(1)	O(n)	inserting at the head of a chain is always O(1)
delete	O(1)	O(n)	tombstones in open addressing
resize	O(n + m)	O(n + m)	amortized O(1) per insertion
sorted traversal / min / max	O(m + n lg n)	O(m + n lg n)	hashing keeps no order; min/max only by scanning: O(n + m)

The worst case can be triggered by malicious input (hash flooding). The defense: choose the function at run time from a random family — **universal hashing** (CLRS 11.3.3).

SUMMARY

Common Mistakes

1. A negative key makes `k % m` negative, writing outside the array.
2. Choosing m as 10 or a power of 2, so only the last digits/bits matter.
3. Marking a deleted slot EMPTY in open addressing: searches stop halfway.
4. Looping forever on a full table: limit the number of probes to m.
5. Not checking the result of `malloc`; using a node after `free`.
6. Copying items at their old indices when resizing (without rehashing).

SUMMARY

Exercises

1. m = 7, h(k) = k mod 7, linear probing: what does the table look like after inserting 10, 17, 24, 3 in order?
2. Place the same keys with chaining (insert at the head); how long is the longest chain?
3. Write the probe sequence 0, 1, 4, 9, 16, 25 (mod 11) for quadratic probing with m = 11.
4. How many probes does an unsuccessful search expect in a linear-probing table with α = 0.9?
5. Compute the additive hash and the base-31 polynomial hash (mod 11) of `"ab"` and `"ba"`.
6. In open addressing, why must `insert` call `search` before writing into a DELETED slot?

▶ **Show answers**

SUMMARY

Chapter Summary

The idea

Compute the position from the key: O(1) search/insert/delete on average.

Hash function

Division (m prime), multiplication (A ≈ 0.618), folding, polynomial string hash.

Open addressing

Linear, quadratic, double hashing; clustering and tombstones.

Chaining

A linked list per bucket; O(1) insertion at the head, easy deletion.

Load factor

Keep $\alpha = n/m$ small; resize past the threshold.

Limits

No order information; worst case O(n); universal hashing as protection.

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Chapter 8 (Hashing).
- T. H. Cormen et al., *Introduction to Algorithms* (CLRS), Chapter 11 (Hash Tables).
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 14 (Hashing).
- D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Section 6.4.

← Chapter 13: Priority Queues and Heaps

Course page

Chapter 15: Graphs

The End

Chapter 14: Hash Tables

Next chapter: Chapter 15: Graphs · Course page