Chapter 12: Maps and Hash Tables

Data Structures

The map ADT | Direct addressing | Hash functions | Collisions | Separate chaining | Open addressing | Load factor | Resizing

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/data_structures/slides/Chapter_12_Maps_and_Hash_Tables.html

INTRODUCTION

Learning Outcomes

The map model

Explain key–value association and how a hash table works.

Collisions

Resolve collisions with chaining and with open addressing.

Load factor

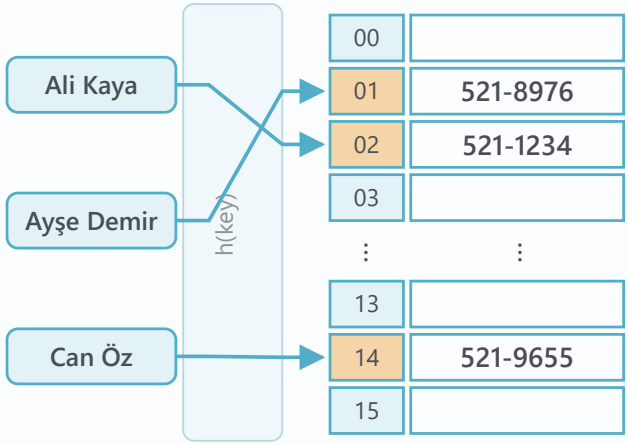
Relate load factor, resizing, and the spread of keys.

Correctness and cost

Honor the `equals` / `hashCode` contract and compare average- and worst-case costs.

Maps

- A map stores data as **key–value** pairs.
- Each key identifies its entry **uniquely**; the value is associated with that key.
- We reach a value through its key: "What is Ayşe Demir's phone number?"
- Implemented with a hash table, operations take **O(1) on average**.



A map is an **abstract data type** (it says what to do); a hash table is the most common **data structure** that implements it (it says how).

Core Operations of the Map ADT

Operation	java.util.Map	Description
Insertion	`put(k, v)`	Adds the pair to the table
Search	`get(k)`	Returns the value for key k, or `null` if absent
Deletion	`remove(k)`	Removes the entry with key k and returns its value
Update	`put(k, v')`	If k is present, replaces its value with v' (returns the old one)
Other	`size()`, `isEmpty()`, `containsKey(k)`, `keySet()`, `values()`	

```
Map<Integer, String> m = new HashMap<>();
m.put(105, "Murat");   m.put(21, "Leyla");
m.put(21, "Ali");      // update: "Leyla" → "Ali"
m.get(21);             // "Ali"
m.remove(105);         // "Murat"
m.get(88);             // null: no such key
```

Searching an Array

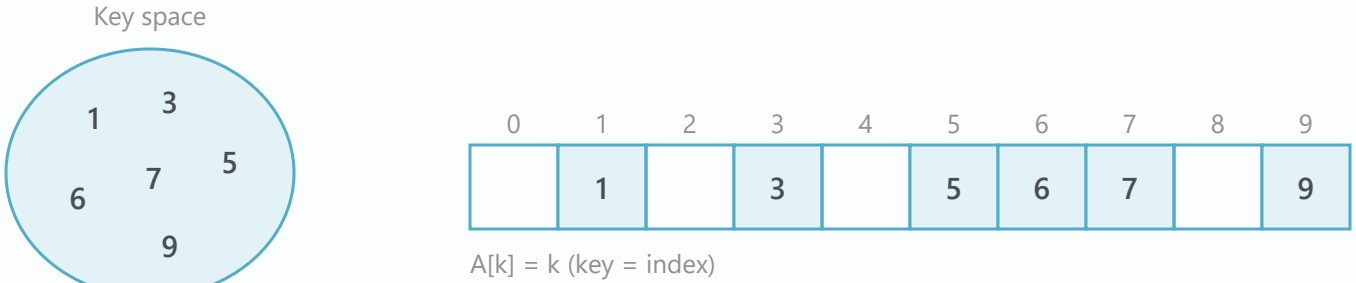
- Can we search an array in O(1) time?
- Not by comparing elements: linear search is O(n), and binary search on a sorted array is O(log n).
- But if we **know the element's position (its index)**, access takes one step:

```
int[] sortedArr = {1, 3, 5, 7, 9};
int idx = 2;                    // index of the element we want
int item = sortedArr[idx];      // O(1): item = 5
```

The idea: if we can turn a key directly into an array index, we reach the right cell in one step, with no comparisons.

Indexing: Key = Index

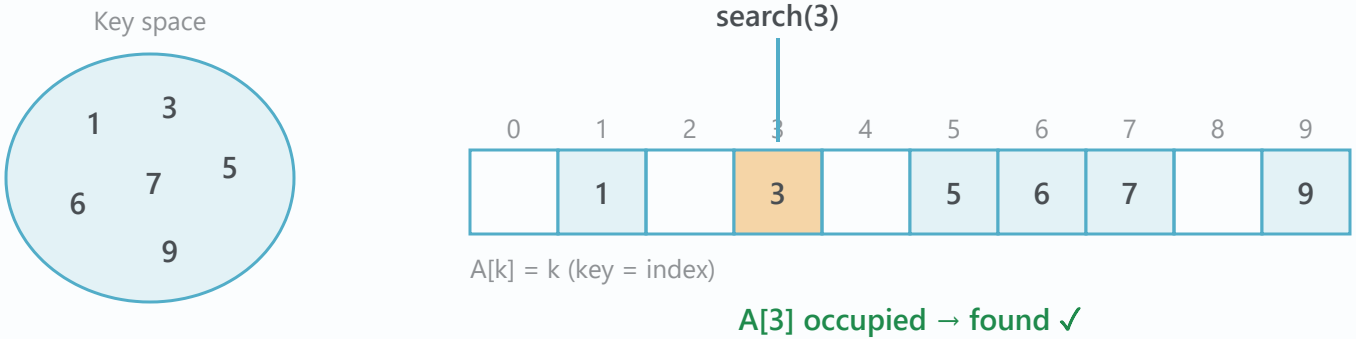
If the keys are integers from 0 to 9, we put key k in cell k of the array, and a search takes one step.



Each key k in the key space goes into cell A[k]; unused cells stay empty.

Indexing: Key = Index

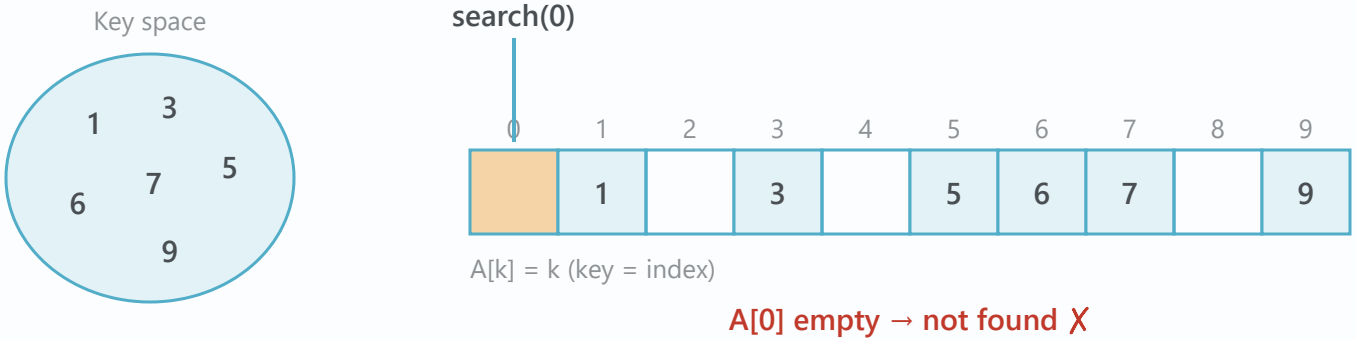
If the keys are integers from 0 to 9, we put key k in cell k of the array, and a search takes one step.



search(3): look straight at A[3] → occupied → **found**, O(1).

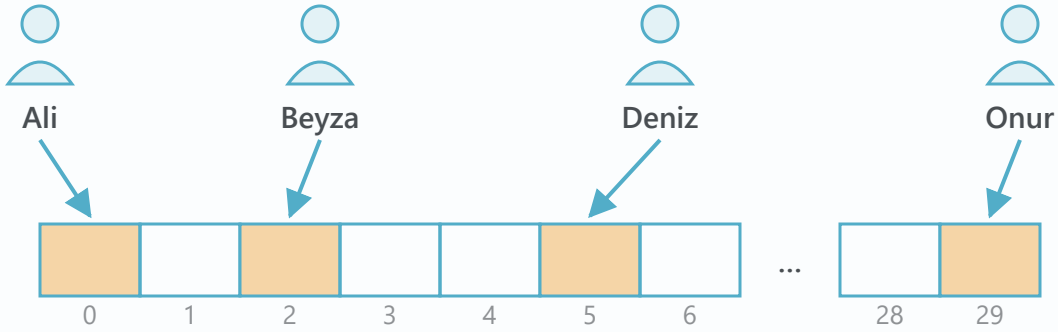
Indexing: Key = Index

If the keys are integers from 0 to 9, we put key k in cell k of the array, and a search takes one step.



search(0): look at A[0] → empty → **not found**, still O(1).

Direct Addressing and Its Limits

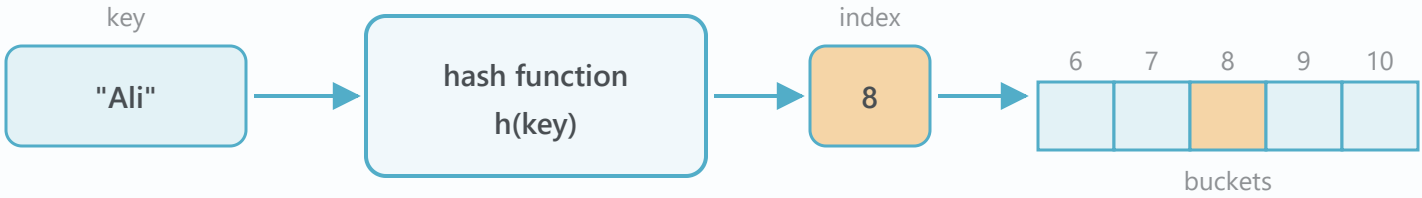


- Reserve one cell for every possible key: search, insert and delete are all O(1).
- A large key space wastes huge amounts of memory: 9-digit student IDs need 10^9 cells (≈ 4 GB as an int array) for a department of a few hundred students.
- Keys that aren't integers ("Ali", "Beyza") can't serve as indices at all.

The fix: a function that squeezes the large key space into a small array → a **hash function**.

Hashing

- Hashing is a technique for storing and retrieving entries.
- It quickly finds a location (a **bucket**) for each entry.
- Once the location is known, inserting, removing and updating in an array are easy.
- The **hash function** does the work: it turns a key into an integer index in the range `0 … N−1`.



"Ali".hashCode() = 65918 → 65918 mod 13 = 8 (a table with N = 13 buckets)

Hash Functions

- At the heart of a hash table is its hash function: it takes a key and produces the entry's position (index) in the table.

Deterministic

The same key must always give the same index, or the entry can never be found again.

Fast

It should run in O(1) (treating key length as constant).

Uniform

It should spread keys evenly over the buckets, which keeps collisions down.

Uses the whole key

A function that looks only at the first letter sends "Ali", "Ayşe" and "Ahmet" to the same bucket.

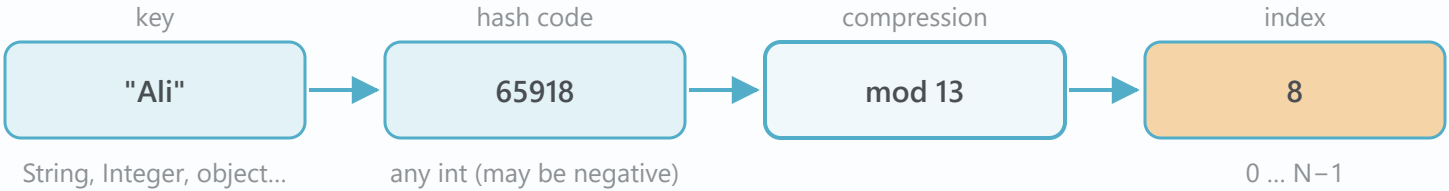
A bad function

– If every entry hashes to the same slot, the table degenerates into a linked list: search is O(n).

Choosing the capacity N

+ Too large wastes memory; too small causes more collisions.

Hash Code + Compression



Division method

`h(k) = k mod N`

Choosing a **prime** N keeps regular patterns in the keys (such as multiples of 10) from piling into the same buckets.

MAD (multiply-add-divide)

`h(k) = ((a·k + b) mod p) mod N`

p > N is prime; a and b are random (a ≠ 0).

k	k mod 10	k mod 11
10	0	10
20	0	9
30	0	8
40	0	7
50	0	6

N = 10: all in one bucket. N = 11 (prime): each in a different bucket.

HASH FUNCTIONS

The `equals` and `hashCode` Contract

Equal objects

If `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` must hold; otherwise equal keys land in different buckets.

Same hash code

Two objects with the same hash code are not necessarily equal. Inside the bucket, the hash table still compares with `equals`.

Override both

If you override `equals`, override `hashCode` too, using the same fields.

Mutable keys

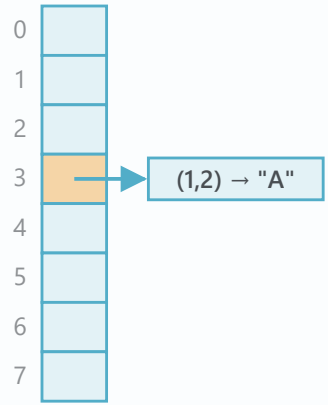
If the fields behind equals/hashCode change after insertion, the entry is looked for in the wrong bucket.

```
record Student(int id, String name) {}
Map<Student, Double> grades = new HashMap<>();
grades.put(new Student(42, "Ada"), 87.5); // record: equals/hashCode come free (Java 16+)
```

In a hash table, `hashCode` picks the bucket and `equals` decides whether two keys are really the same.

Common Mistake: `equals` Without `hashCode`

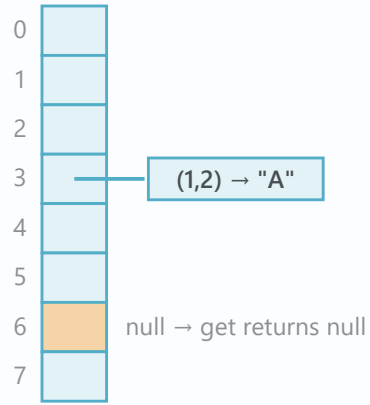
```
class Point {
  int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
  public boolean equals(Object o) {
    return o instanceof Point p
        && x == p.x && y == p.y;
  }
  // no hashCode() -> Object's is used
}
Map<Point, String> m = new HashMap<>();
m.put(new Point(1, 2), "A");
m.get(new Point(1, 2));  // most likely null (!)
```



`put(p1, "A")` : `p1.hashCode()` picks the bucket. Point doesn't override it, so **Object.hashCode()** (object identity) is used; say, bucket 3.

Common Mistake: `equals` Without `hashCode`

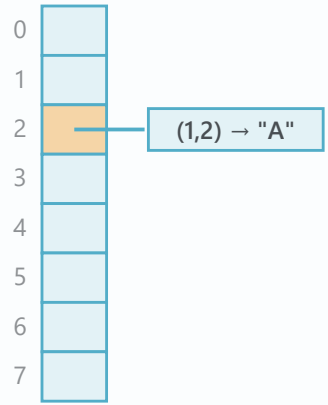
```
class Point {
  int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
  public boolean equals(Object o) {
    return o instanceof Point p
        && x == p.x && y == p.y;
  }
  // no hashCode() -> Object's is used
}
Map<Point, String> m = new HashMap<>();
m.put(new Point(1, 2), "A");
m.get(new Point(1, 2));  // most likely null (!)
```



`get(p2)` : p2 is **equal** to p1 (equals is true) but a different object, so its identity hash (almost always) differs → we look in, say, bucket 6, find it empty → **null**. equals is never even called!

Common Mistake: `equals` Without `hashCode`

```
class Point {
  int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
  public boolean equals(Object o) {
    return o instanceof Point p
        && x == p.x && y == p.y;
  }
  // no hashCode() -> Object's is used
}
Map<Point, String> m = new HashMap<>();
m.put(new Point(1, 2), "A");
m.get(new Point(1, 2));  // most likely null (!)
```



The fix: override `hashCode()` too:

`return Objects.hash(x, y);`

Equal points land in the same bucket, where equals finds the match → **"A"**.

Sample Hash Functions

Modulus (division remainder)

```
int modulusHash(int key, int arraySize) {
  return Math.floorMod(key, arraySize);
}
```

`floorMod` gives a safe index in 0 … arraySize−1 even for negative keys.

CRC32

```
long crc32Hash(String key) {
  CRC32 crc = new CRC32();  // java.util.zip
  crc.update(key.getBytes());
  return crc.getValue();
}
```

DJB2 (strings)

```
int djb2Hash(String s) {
  int h = 5381;
  for (int i = 0; i < s.length(); i++)
    h = (h * 33) ^ s.charAt(i);
  return h;
}
```

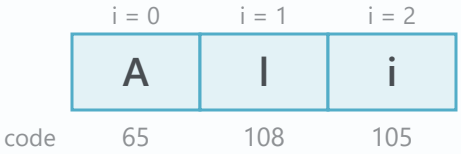
Classic DJB2 uses `h*33 + c`; this is the XOR variant. The result can be negative, so it needs compressing before use as an index.

Java: String.hashCode()

```
// Σ s[i] · 31^(n−1−i)
"abc".hashCode()  // 96354
```

$97 \cdot 31^2 + 98 \cdot 31 + 99 = 96354$

Code Trace: `"Ali".hashCode()`



`hash("Ali")` is called; each character is processed by its Unicode code (A = 65, l = 108, i = 105).

Step 1 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
4    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
▶ int code = hash("Ali");                  // 65918
11 int index = Math.floorMod(code, 13);    // 8
```

VARIABLES

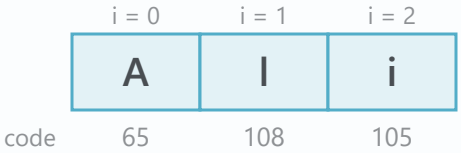
`s = "Ali"`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



h = 0

Start with **h = 0**.

Step 2 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
▶    int h = 0;
4    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

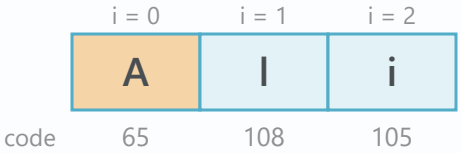
`s = "Ali"` `h = 0`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



h = 0

0 < 3 → the loop continues; the character is **s.charAt(0) = 'A'** (65).

Step 3 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
▶    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

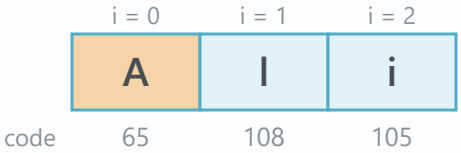
`s = "Ali"` `h = 0` `i = 0`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



$h = 31 \cdot 0 + 65$

$= 65$

h = 31 · 0 + 65 = 65. The running total is multiplied by 31, so the character's **position** matters too.

Step 4 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
4    for (int i = 0; i < s.length(); i++)
▶      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

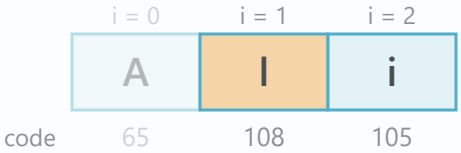
VARIABLES

`s = "Ali"` `h = 65` `i = 0`

CALL STACK

- `hash("Ali")`
- `main()`

Code Trace: `"Ali".hashCode()`



h = 65

1 < 3 → the loop continues; the character is **s.charAt(1) = 'l'** (108).

Step 5 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
▶    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

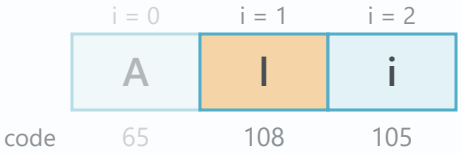
`s = "Ali"` `h = 65` `i = 1`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



h = 31 · 65 + 108

= 2123

h = 31 · 65 + 108 = 2123. The running total is multiplied by 31, so the character's **position** matters too.

Step 6 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
4    for (int i = 0; i < s.length(); i++)
▶      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

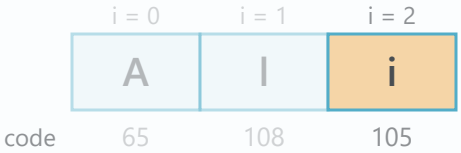
`s = "Ali"` `h = 2123` `i = 1`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



h = 2123

2 < 3 → the loop continues; the character is **s.charAt(2) = 'i'** (105).

Step 7 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
▶    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

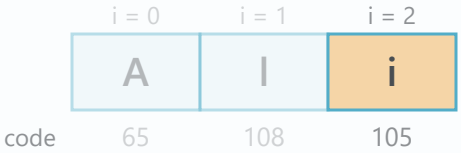
`s = "Ali"` `h = 2123` `i = 2`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



h = 31 · 2123 + 105

= 65918

h = 31 · 2123 + 105 = 65918. The running total is multiplied by 31, so the character's **position** matters too.

Step 8 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
4    for (int i = 0; i < s.length(); i++)
▶      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

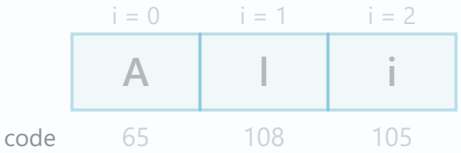
`s = "Ali"` `h = 65918` `i = 2`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



h = 65918

3 < 3 is false → the loop ends.


```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
▶    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

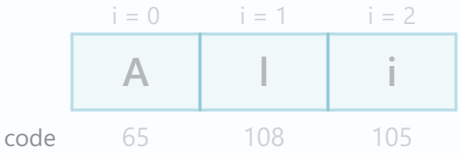
`s = "Ali"` `h = 65918` `i = 3`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



$65 \cdot 31^2 + 108 \cdot 31 + 105$

$= 65918$

`return 65918` $= 65 \cdot 31^2 + 108 \cdot 31 + 105$.

Step 10 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
4    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
▶    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                // 65918
11 int index = Math.floorMod(code, 13);   // 8
```

VARIABLES

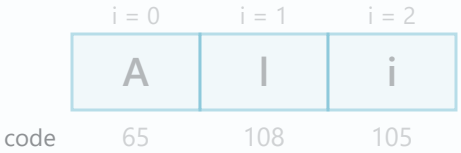
`s = "Ali"` `h = 65918`

CALL STACK

`hash("Ali")`

`main()`

Code Trace: `"Ali".hashCode()`



code = 65918

code = 65918. For long strings the product overflows int and the result can be **negative**.

Step 11 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
4    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
▶ int code = hash("Ali");                  // 65918
11 int index = Math.floorMod(code, 13);    // 8
```

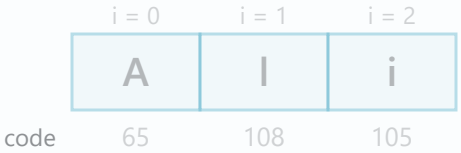
VARIABLES

`code = 65918`

CALL STACK

`main()`

Code Trace: `"Ali".hashCode()`



65918 mod 13

= 8

Compression: floorMod(65918, 13) = **8** → bucket 8 in a 13-bucket table.

Step 12 / 12

```
1  // same computation as String.hashCode():
2  int hash(String s) {
3    int h = 0;
4    for (int i = 0; i < s.length(); i++)
5      h = 31 * h + s.charAt(i);
6    return h;
7  }
8
9  // main:
10 int code = hash("Ali");                  // 65918
▶  int index = Math.floorMod(code, 13);     // 8
```

VARIABLES

`code = 65918` `index = 8`

CALL STACK

`main()`

Code Trace: `modulusHash()`



An empty 10-cell table; every cell is `null`.

Step 1 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
▶  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

VARIABLES

`table.length = 10`

CALL STACK

`main()`

Code Trace: `modulusHash()`



Next key: **k = 5**.

Step 2 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
▶ for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

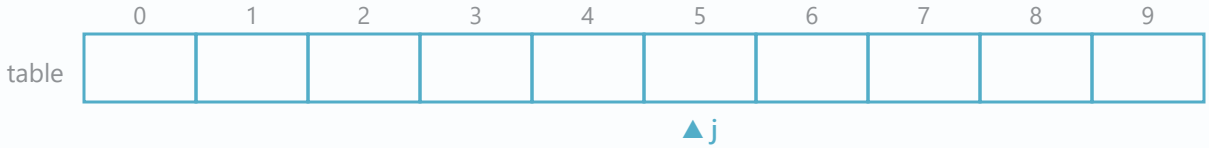
VARIABLES

k = 5

CALL STACK

main()

Code Trace: `modulusHash()`



`modulusHash(5, 10)` : 5 mod 10 = **5** (the remainder).

Step 3 / 16

```
1  int modulusHash(int key, int arraySize) {
▶    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

VARIABLES

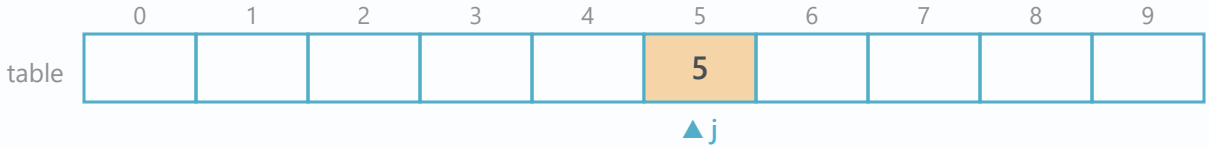
`key = 5` `arraySize = 10`

CALL STACK

`modulusHash(5, 10)`

`main()`

Code Trace: `modulusHash()`



`table[5] = 5` .

Step 4 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
▶    table[j] = k;
10 }
```

VARIABLES

`k = 5` `j = 5`

CALL STACK

`main()`

Code Trace: `modulusHash()`



Next key: **k = 1**.

Step 5 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
▶ for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

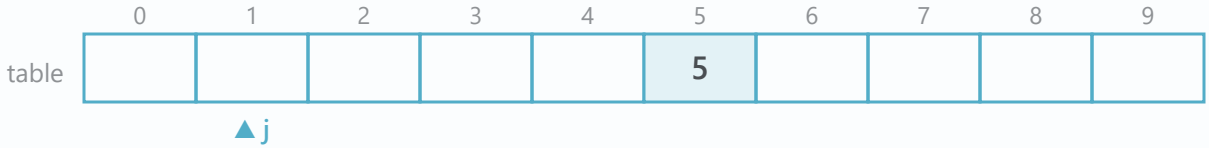
VARIABLES

k = 1

CALL STACK

main()

Code Trace: `modulusHash()`



`modulusHash(1, 10)` : 1 mod 10 = **1** (the remainder).

Step 6 / 16

```
1  int modulusHash(int key, int arraySize) {
▶    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

VARIABLES

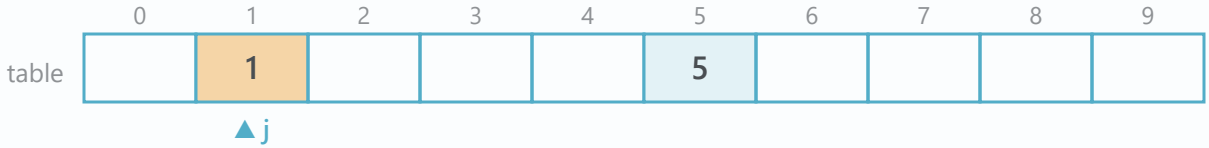
`key = 1` `arraySize = 10`

CALL STACK

`modulusHash(1, 10)`

`main()`

Code Trace: `modulusHash()`



`table[1] = 1` .

Step 7 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
▶    table[j] = k;
10 }
```

VARIABLES

`k = 1` `j = 1`

CALL STACK

`main()`

Code Trace: `modulusHash()`



Next key: **k = 10**.

Step 8 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
▶ for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

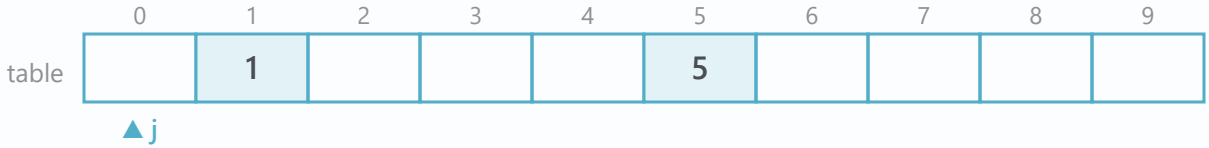
VARIABLES

`k = 10`

CALL STACK

`main()`

Code Trace: `modulusHash()`



`modulusHash(10, 10)` : 10 mod 10 = **0** (the remainder).

Step 9 / 16

```
1  int modulusHash(int key, int arraySize) {
▶    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

VARIABLES

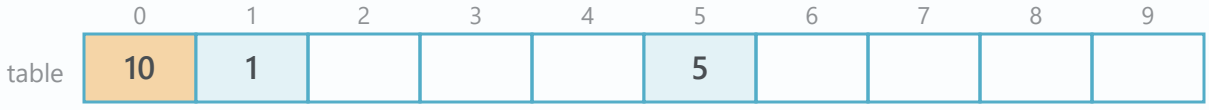
`key = 10` `arraySize = 10`

CALL STACK

`modulusHash(10, 10)`

`main()`

Code Trace: `modulusHash()`



`table[0] = 10` .

Step 10 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
▶    table[j] = k;
10 }
```

VARIABLES

`k = 10` `j = 0`

CALL STACK

`main()`

Code Trace: `modulusHash()`



Next key: **k = 26**.

Step 11 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
▶ for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

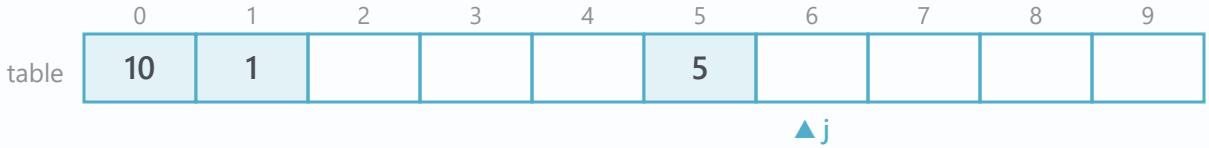
VARIABLES

k = 26

CALL STACK

main()

Code Trace: `modulusHash()`



`modulusHash(26, 10)` : 26 mod 10 = **6** (the remainder).

Step 12 / 16

```
1  int modulusHash(int key, int arraySize) {
▶    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

VARIABLES

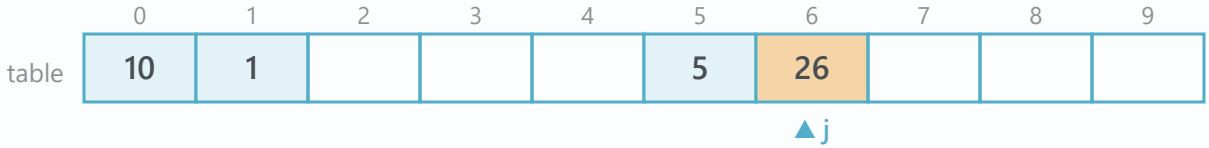
`key = 26` `arraySize = 10`

CALL STACK

`modulusHash(26, 10)`

`main()`

Code Trace: `modulusHash()`



`table[6] = 26` .

Step 13 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
▶    table[j] = k;
10 }
```

VARIABLES

`k = 26` `j = 6`

CALL STACK

`main()`

Code Trace: `modulusHash()`



Next key: **k = 99**.

Step 14 / 16

```
1  int modulusHash(int key, int arraySize) {
2    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
▶ for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

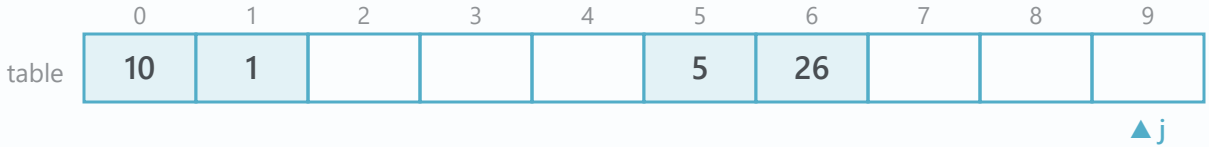
VARIABLES

k = 99

CALL STACK

main()

Code Trace: `modulusHash()`



`modulusHash(99, 10)` : 99 mod 10 = **9** (the remainder).

Step 15 / 16

```
1  int modulusHash(int key, int arraySize) {
▶    return Math.floorMod(key, arraySize);
3  }
4
5  // main:
6  Integer[] table = new Integer[10];
7  for (int k : new int[]{5, 1, 10, 26, 99}) {
8    int j = modulusHash(k, table.length);
9    table[j] = k;
10 }
```

VARIABLES

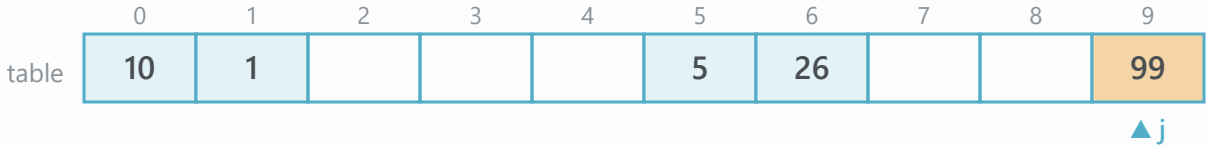
`key = 99` `arraySize = 10`

CALL STACK

`modulusHash(99, 10)`

`main()`

Code Trace: `modulusHash()`



`table[9] = 99`. Five keys, five different cells: no collisions. A 15, though, would collide with 5.

Step 16 / 16

```
1 int modulusHash(int key, int arraySize) {
2   return Math.floorMod(key, arraySize);
3 }
4
5 // main:
6 Integer[] table = new Integer[10];
7 for (int k : new int[]{5, 1, 10, 26, 99}) {
8   int j = modulusHash(k, table.length);
▶   table[j] = k;
10 }
```

VARIABLES

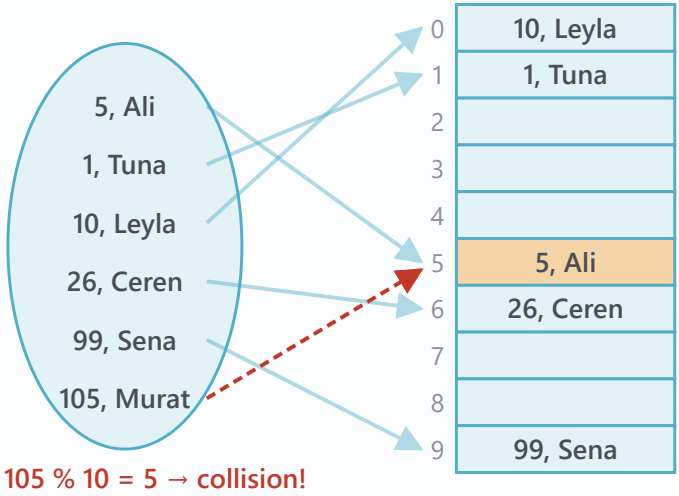
`k = 99` `j = 9`

CALL STACK

`main()`

Collisions

- A collision happens when different keys produce the same index: h(5) = h(105) = 5.
- Two entries cannot share a cell directly.
- Because the key space is larger than the table, collisions are **inevitable**, so we need a strategy for them:



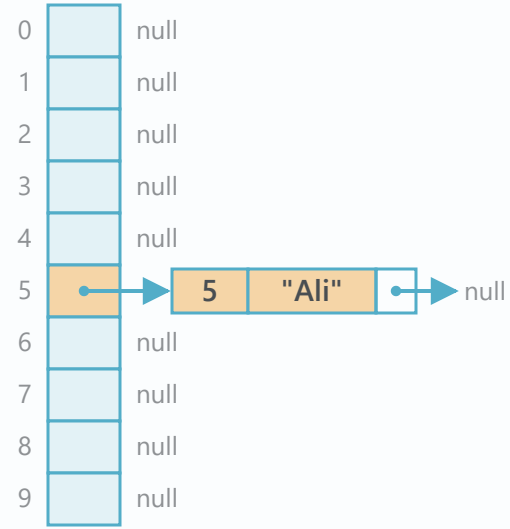
Separate chaining

Each bucket holds a linked list of the entries that hash to it.

Open addressing

An entry that hits an occupied slot looks for another empty slot in the array.

Resolving Collisions: Separate Chaining

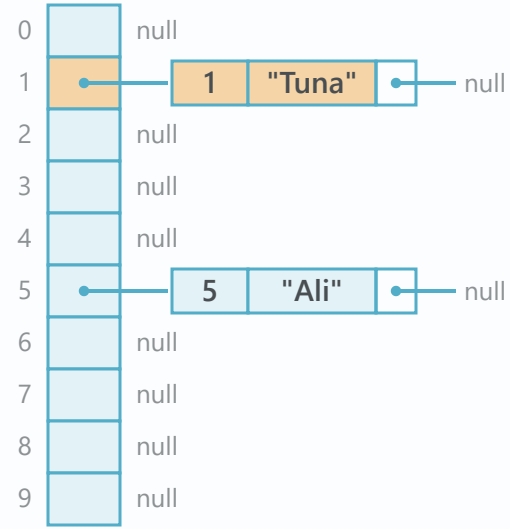


h(k) = k % 10

5 % 10 = 5

h(5) = 5 % 10 = **5** → bucket 5 is empty; node (5, \"Ali\") is linked in.

Resolving Collisions: Separate Chaining



HashNode[] buckets

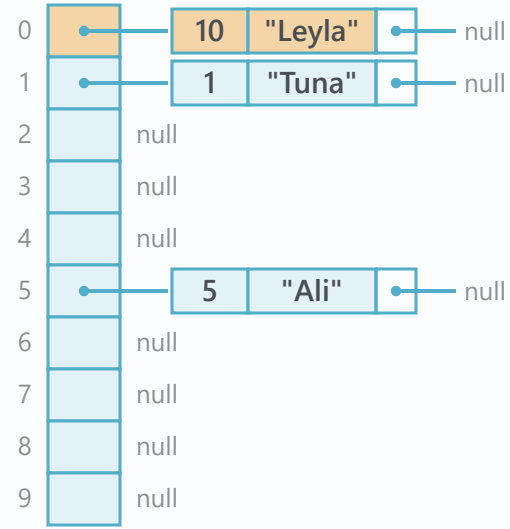
h(k) = k % 10

5 % 10 = 5

1 % 10 = 1

h(1) = 1 % 10 = **1** → bucket 1 is empty; node (1, \"Tuna\") is linked in.

Resolving Collisions: Separate Chaining



HashNode[] buckets

h(k) = k % 10

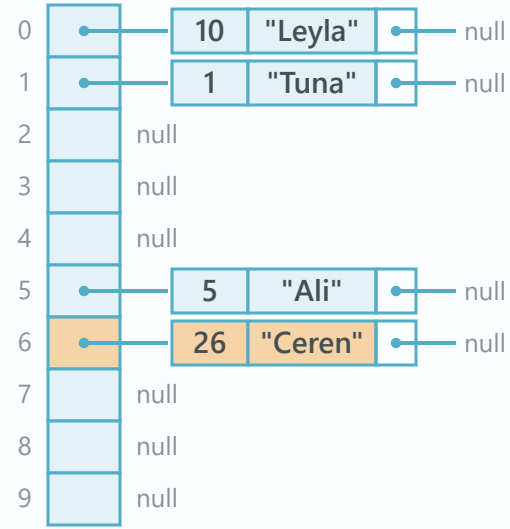
5 % 10 = 5

1 % 10 = 1

10 % 10 = 0

h(10) = 10 % 10 = **0** → bucket 0 is empty; node (10, \"Leyla\") is linked in.

Resolving Collisions: Separate Chaining



h(k) = k % 10

5 % 10 = 5

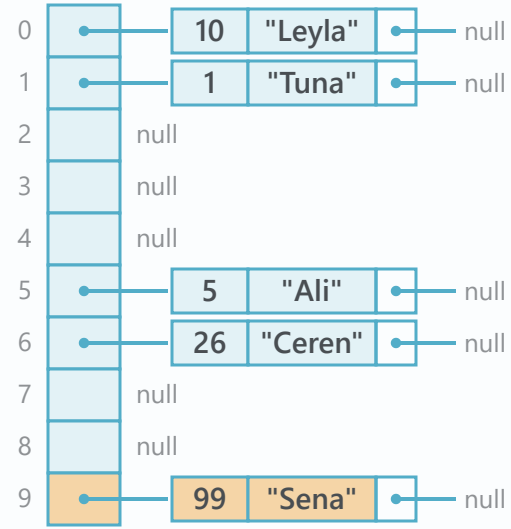
1 % 10 = 1

10 % 10 = 0

26 % 10 = 6

h(26) = 26 % 10 = **6** → bucket 6 is empty; node (26, \"Ceren\") is linked in.

Resolving Collisions: Separate Chaining



h(k) = k % 10

5 % 10 = 5

1 % 10 = 1

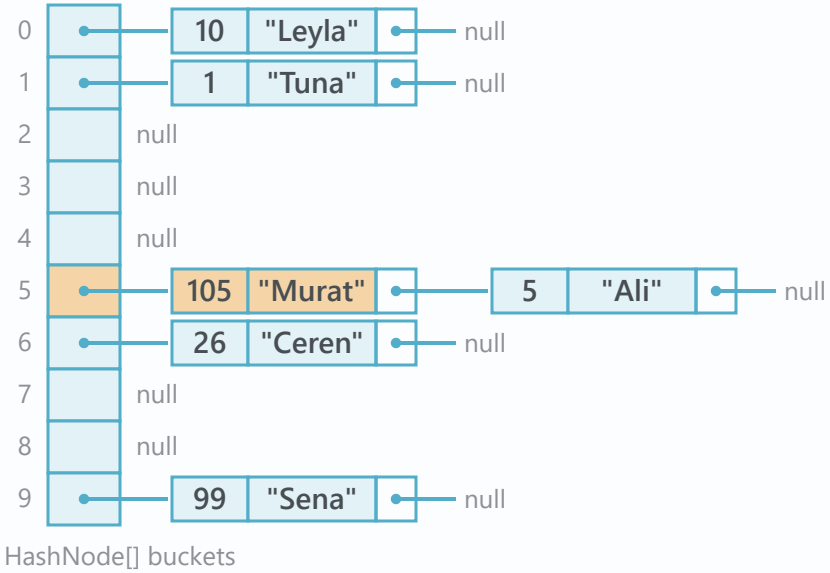
10 % 10 = 0

26 % 10 = 6

99 % 10 = 9

h(99) = 99 % 10 = **9** → bucket 9 is empty; node (99, \"Sena\") is linked in.

Resolving Collisions: Separate Chaining



h(k) = k % 10

5 % 10 = 5

1 % 10 = 1

10 % 10 = 0

26 % 10 = 6

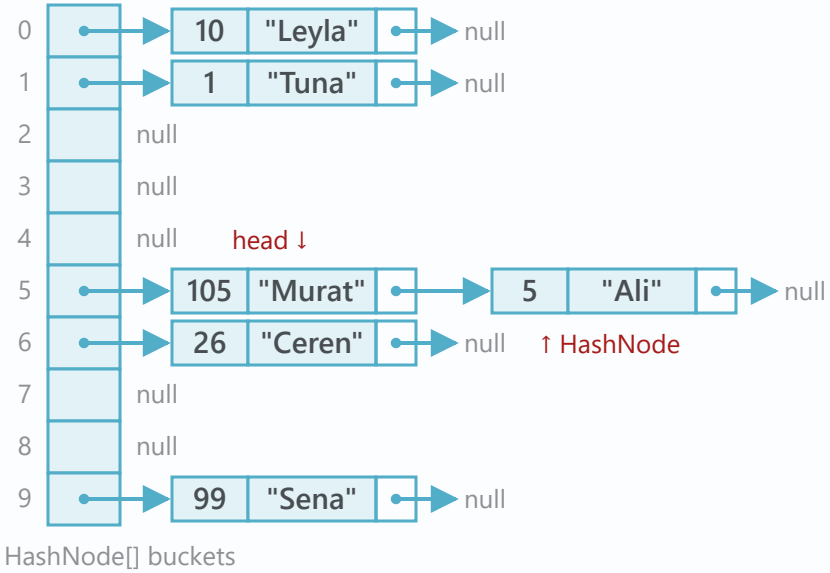
99 % 10 = 9

105 % 10 = 5

h(105) = 105 % 10 = **5** → bucket 5 already holds (5, "Ali"): a **collision**. The new node goes at the **head** of the chain.

SEPARATE CHAINING

Hash Table Terminology



buckets

HashNode[] array

bucket

one cell of the array

numBuckets = 10

array length = capacity

size = 6

number of stored pairs

chain

nodes in the same bucket

head

first node of the chain

Implementing a Hash Table: The Class

```
public class HashTable {
  private HashNode[] buckets;   // heads of the chains
  private int numBuckets, size;

  public HashTable(int capacity) {
    this.numBuckets = capacity;
    this.buckets = new HashNode[capacity];
    this.size = 0;
  }
  private class HashNode {
    private Integer key;
    private String value;
    private HashNode next;
    public HashNode(Integer key, String value) {
      this.key = key; this.value = value;
    }
  }
}
```

buckets

Each cell holds the first node of a chain.

HashNode

key + value + next: a singly linked list node.

new HashTable(10)

Creates 10 buckets, all `null`; size = 0.

Bucket Index: getBucketIndex()

```java
// Naive: key % buckets.length → may be negative!
private int getBucketIndex(Integer key) {
  int h = key.hashCode();  // = key
  return Math.floorMod(h, buckets.length);
}
```

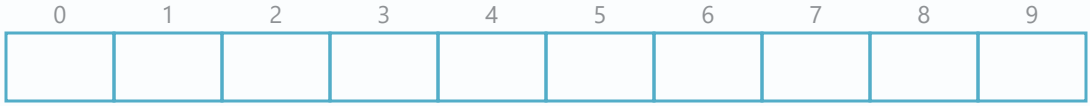
key	key % 10	Math.floorMod(key, 10)	Result
105	5	5	bucket 5
21	1	1	bucket 1
41	1	1	bucket 1 (collides with 21)
−7	−7	3	with `%`, `buckets[-7]` → ArrayIndexOutOfBoundsException

In Java, the result of `%` takes the sign of the dividend. `Integer.hashCode()` is just the number, and for keys such as strings `hashCode()` is often negative. `Math.abs(h) % N` isn't safe either: `Math.abs(Integer.MIN_VALUE)` is negative.

Code Trace: `getBucketIndex(41)`, `(-7)`

key

41



`getBucketIndex(41)` is called; the table has 10 buckets.

Step 1 / 14

```
▶ private int getBucketIndex(Integer key) {
2   int h = key.hashCode();  // = key
3   return Math.floorMod(h, buckets.length);
4 }
5
6 // what floorMod does for n > 0:
7 static int floorMod(int x, int n) {
8   int r = x % n;        // sign comes from x
9   if (r < 0) r += n;    // if negative, add n
10  return r;
11 }
```

VARIABLES

`key = 41` `buckets.length = 10`

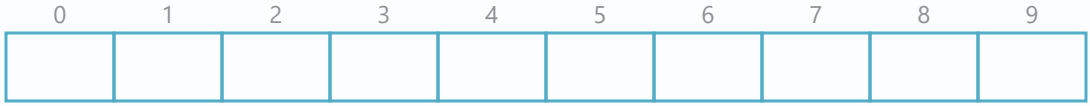
CALL STACK

`getBucketIndex(41)`

Code Trace: `getBucketIndex(41)`, `(-7)`

key

41



`Integer.hashCode()` is just the number: **h = 41**.

Step 2 / 14

```
1  private int getBucketIndex(Integer key) {
▶    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

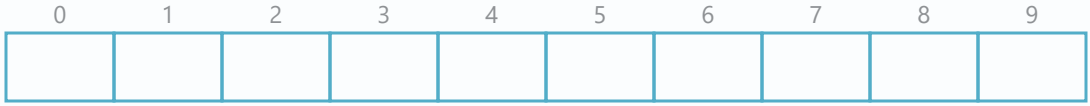
key = **41** h = **41**

CALL STACK

`getBucketIndex(41)`

Code Trace: `getBucketIndex(41)`, `(-7)`

key
41



`floorMod(41, 10)` is called.

Step 3 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
▶    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

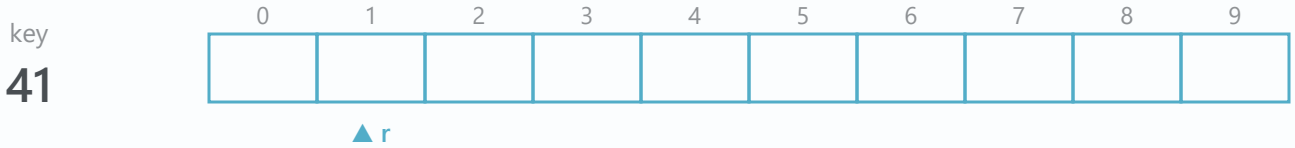
`key = 41` `h = 41`

CALL STACK

`floorMod(41, 10)`

`getBucketIndex(41)`

Code Trace: `getBucketIndex(41)`, `(-7)`



In Java, **41 % 10 = 1**.

Step 4 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
▶    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

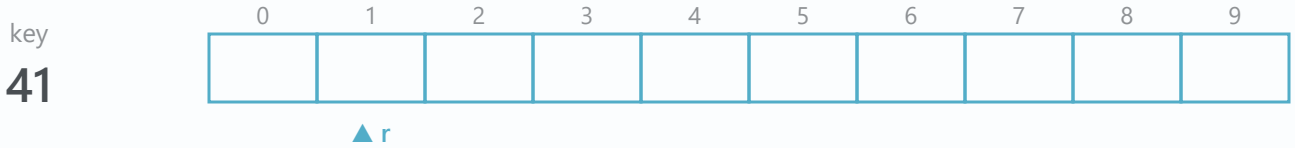
`x = 41` `n = 10` `r = 1`

CALL STACK

`floorMod(41, 10)`

`getBucketIndex(41)`

Code Trace: `getBucketIndex(41)`, `(-7)`



r = 1 ≥ 0 → no correction needed.

Step 5 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
▶    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

`x = 41` `n = 10` `r = 1`

CALL STACK

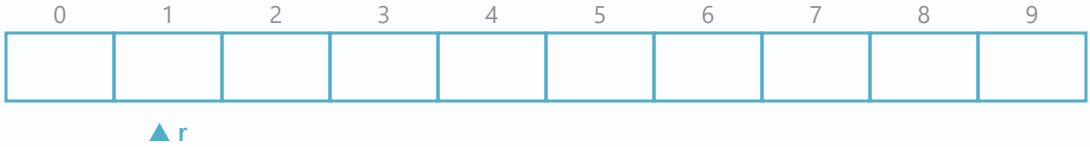
`floorMod(41, 10)`

`getBucketIndex(41)`

Code Trace: `getBucketIndex(41)`, `(-7)`

key

41



`return 1` .

Step 6 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
▶    return r;
11 }
```

VARIABLES

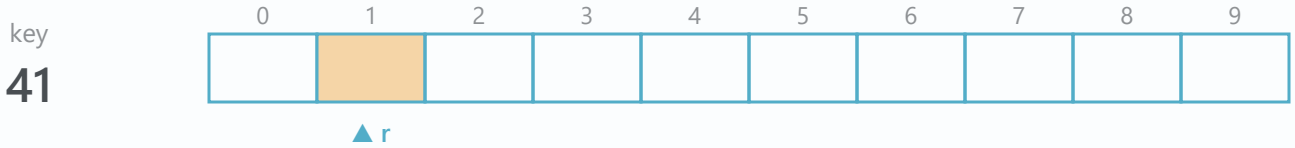
`x = 41` `n = 10` `r = 1`

CALL STACK

`floorMod(41, 10)`

`getBucketIndex(41)`

Code Trace: `getBucketIndex(41)`, `(-7)`



Bucket index **1**. (21 also lands in bucket 1: a collision.)

Step 7 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
▶    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

`key = 41` `h = 41`

CALL STACK

`getBucketIndex(41)`

Code Trace: `getBucketIndex(41)`, `(-7)`



`getBucketIndex(-7)` is called; the table has 10 buckets.

Step 8 / 14

```
▶ private int getBucketIndex(Integer key) {
2   int h = key.hashCode();  // = key
3   return Math.floorMod(h, buckets.length);
4 }
5
6 // what floorMod does for n > 0:
7 static int floorMod(int x, int n) {
8   int r = x % n;        // sign comes from x
9   if (r < 0) r += n;    // if negative, add n
10  return r;
11 }
```

VARIABLES

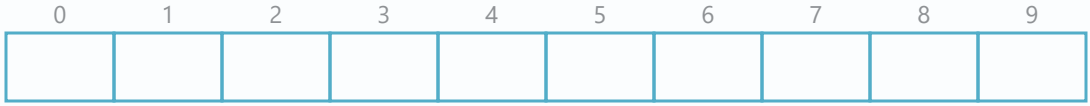
`key = -7` `buckets.length = 10`

CALL STACK

`getBucketIndex(-7)`

Code Trace: `getBucketIndex(41), (-7)`

key
-7



`Integer.hashCode()` is just the number: **h = -7**.

Step 9 / 14

```
1  private int getBucketIndex(Integer key) {
▶    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

key = **-7** h = **-7**

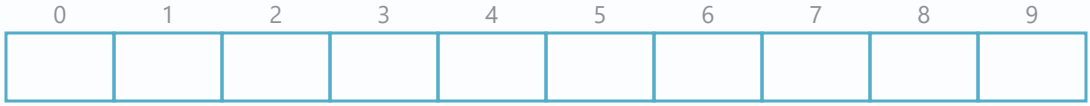
CALL STACK

getBucketIndex(-7)

Code Trace: `getBucketIndex(41)`, `(-7)`

key

-7



`floorMod(-7, 10)` is called.

Step 10 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
▶    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

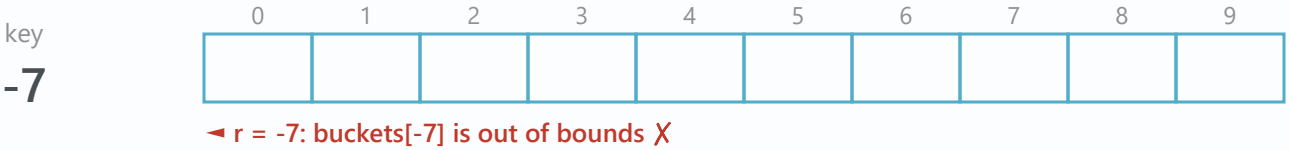
`key = -7` `h = -7`

CALL STACK

`floorMod(-7, 10)`

`getBucketIndex(-7)`

Code Trace: `getBucketIndex(41), (-7)`



In Java, **-7 % 10 = -7**: the result takes the sign of the dividend.

Step 11 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
▶    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

`x = -7` `n = 10` `r = -7`

CALL STACK

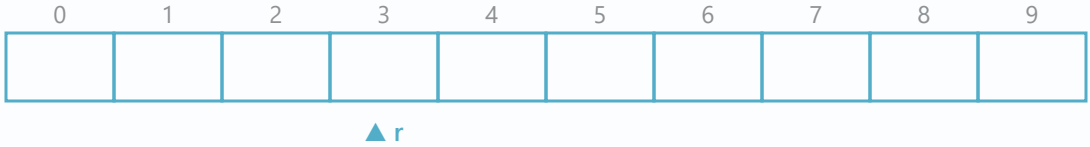
`floorMod(-7, 10)`

`getBucketIndex(-7)`

Code Trace: `getBucketIndex(41), (-7)`

key

-7



r < 0 → **r += 10** → r = 3, now in the range 0 … 9.

Step 12 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
▶    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

`x = -7` `n = 10` `r = 3`

CALL STACK

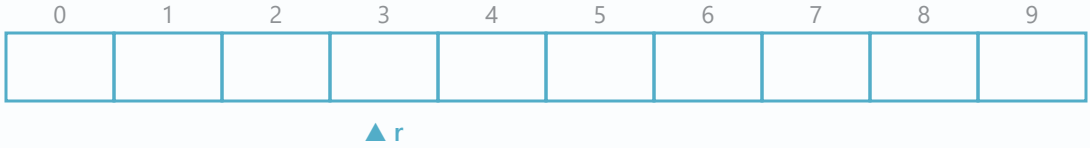
`floorMod(-7, 10)`

`getBucketIndex(-7)`

Code Trace: `getBucketIndex(41), (-7)`

key

-7



`return 3` .

Step 13 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
3    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
▶    return r;
11 }
```

VARIABLES

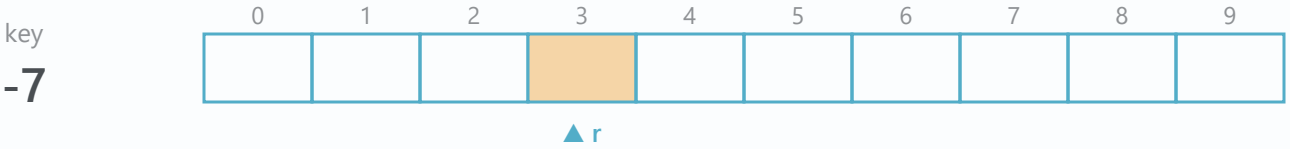
`x = -7` `n = 10` `r = 3`

CALL STACK

`floorMod(-7, 10)`

`getBucketIndex(-7)`

Code Trace: `getBucketIndex(41)`, `(-7)`



Bucket index **3**. With `%` we would get `buckets[-7]` → an exception.

Step 14 / 14

```
1  private int getBucketIndex(Integer key) {
2    int h = key.hashCode();  // = key
▶    return Math.floorMod(h, buckets.length);
4  }
5
6  // what floorMod does for n > 0:
7  static int floorMod(int x, int n) {
8    int r = x % n;        // sign comes from x
9    if (r < 0) r += n;    // if negative, add n
10   return r;
11 }
```

VARIABLES

`key = -7` `h = -7`

CALL STACK

`getBucketIndex(-7)`

Inserting a Key–Value Pair

```
public void put(Integer key,
                      String value) {
  int bucketIndex = getBucketIndex(key);
  HashNode head = buckets[bucketIndex];
  while (head != null) {    // key present?
    if (head.key.equals(key)) {
      head.value = value;   // yes: update
      return;
    }
    head = head.next;
  }
  size++;
  HashNode node =
      new HashNode(key, value);
  node.next = buckets[bucketIndex];
  buckets[bucketIndex] = node;   // insert at head
}
```

1. Find the bucket

O(1) with `h(key)`.

2. Scan the chain

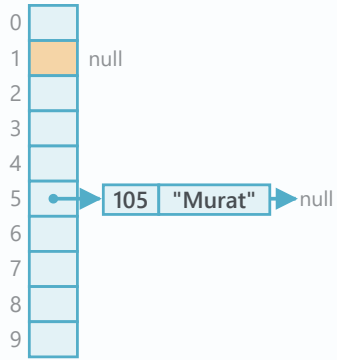
If the key is there, only the value changes; size stays the same.

3. Insert at the head

Otherwise a new node goes at the head of the chain: O(1).

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`put(21, \"Leyla\")` : h(21) = 21 mod 10 = **1**
→ only bucket 1 is examined.

Step 1 / 17

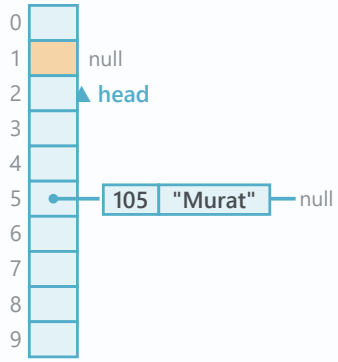
VARIABLES

`key = 21` `value = "Leyla"`

`bucketIndex = 1` `head = –` `size = 1`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



Bucket 1 is empty: `head = null`.

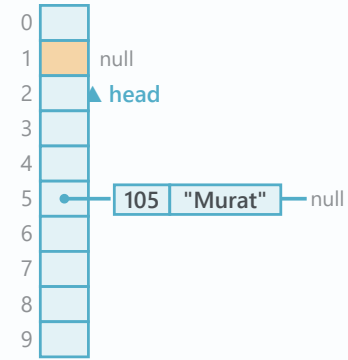
Step 2 / 17

VARIABLES

`key = 21` `value = "Leyla"` `bucketIndex = 1` `head = null` `size = 1`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
▶    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head == null` → the loop ends: 21 is **not** in the table; we need a new node.

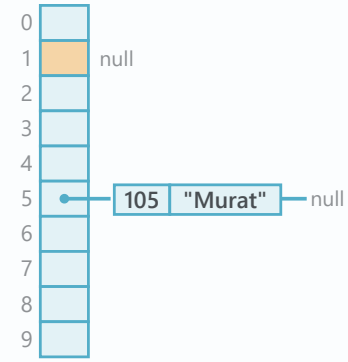
Step 3 / 17

VARIABLES

`key = 21` `value = "Leyla"` `bucketIndex = 1` `head = null` `size = 1`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
▶  size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



New pair: **size = 2**.

Step 4 / 17

VARIABLES

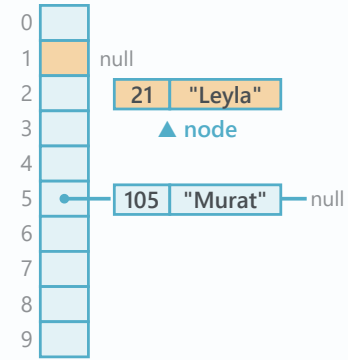
`key = 21` `value = "Leyla"`

`bucketIndex = 1` `head = null`

`size = 2`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
▶    HashNode node =
▶        new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



A new node (21, \"Leyla\") is created; its `next` is still null, so it isn't in the table yet.

Step 5 / 17

VARIABLES

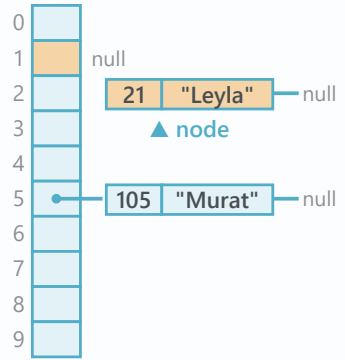
`key = 21` `value = "Leyla"`

`bucketIndex = 1` `head = null`

`size = 2` `node = node(21)`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
▶  node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`node.next = buckets[1]` = null (the bucket was empty).

Step 6 / 17

VARIABLES

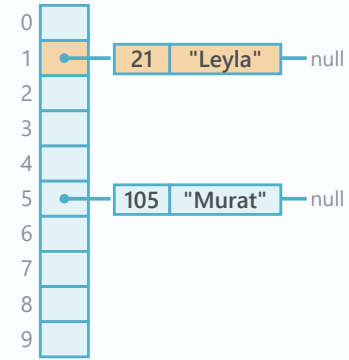
`key = 21` `value = "Leyla"`

`bucketIndex = 1` `head = null`

`size = 2` `node = node(21)`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
▶    buckets[bucketIndex] = node;   // insert at head
17 }
```



`buckets[1] = node` : the new node is at the **head** of the chain. No need to walk to the end → O(1).

Step 7 / 17

VARIABLES

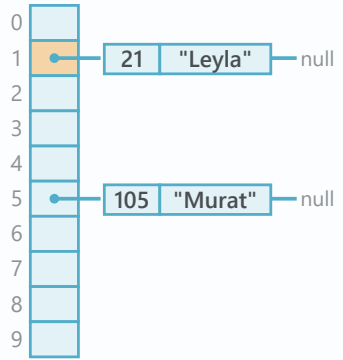
`key = 21` `value = "Leyla"`

`bucketIndex = 1` `head = null`

`size = 2` `node = node(21)`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`put(41, \"Sena\")` : h(41) = 41 mod 10 = **1**
→ only bucket 1 is examined.

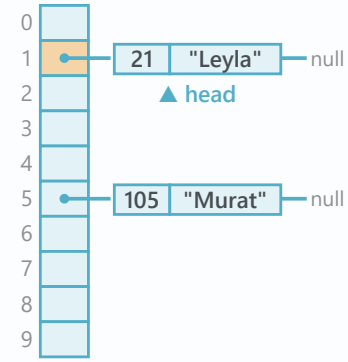
Step 8 / 17

VARIABLES

`key = 41` `value = "Sena"` `bucketIndex = 1` `head = –` `size = 2`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head` points to the first node of the chain.

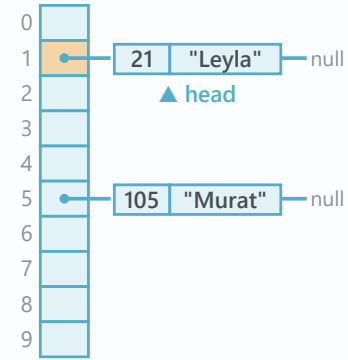
Step 9 / 17

VARIABLES

`key = 41` `value = "Sena"` `bucketIndex = 1` `head = node(21)` `size = 2`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
▶    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head != null` → enter the loop to examine this node.

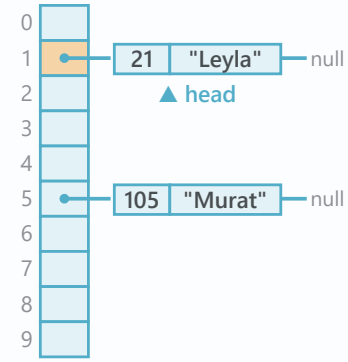
Step 10 / 17

VARIABLES

`key = 41` `value = "Sena"` `bucketIndex = 1` `head = node(21)` `size = 2`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
▶      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



21 ≠ 41 → no match; move on.

Step 11 / 17

VARIABLES

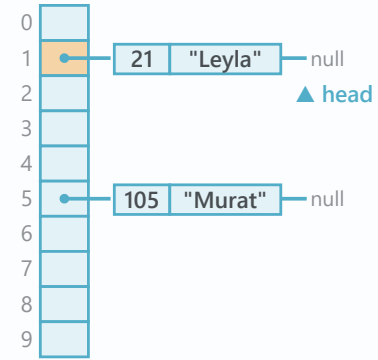
`key = 41` `value = "Sena"`

`bucketIndex = 1` `head = node(21)`

`size = 2`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
▶      head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head = head.next` → null (end of the chain).

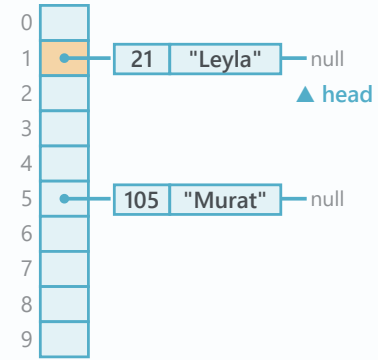
Step 12 / 17

VARIABLES

`key = 41` `value = "Sena"` `bucketIndex = 1` `head = null` `size = 2`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
▶    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head == null` → the loop ends: 41 is **not** in the table; we need a new node.

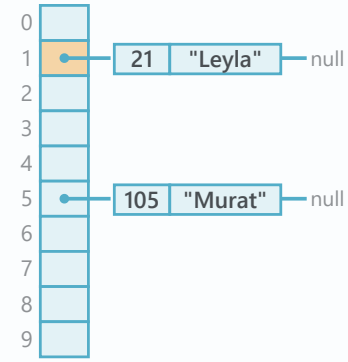
Step 13 / 17

VARIABLES

`key = 41` `value = "Sena"` `bucketIndex = 1` `head = null` `size = 2`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
▶  size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



New pair: **size = 3**.

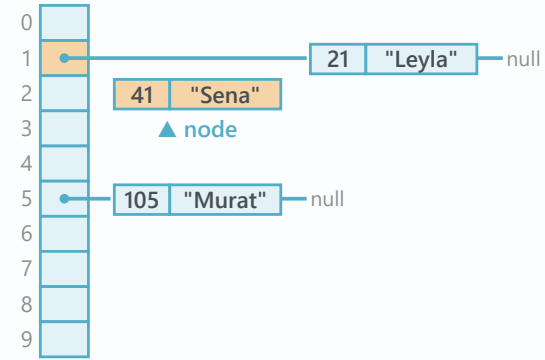
Step 14 / 17

VARIABLES

`key = 41` `value = "Sena"` `bucketIndex = 1` `head = null` `size = 3`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
▶    HashNode node =
▶        new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



A new node (41, \"Sena\") is created; its `next` is still null, so it isn't in the table yet.

Step 15 / 17

VARIABLES

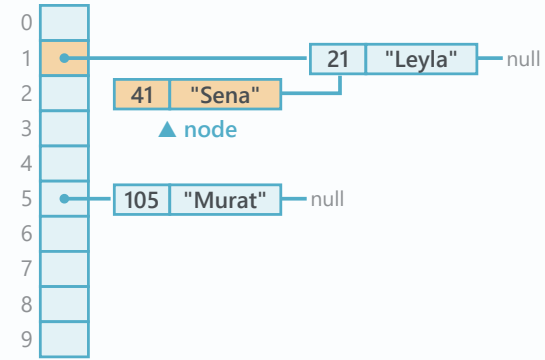
`key = 41` `value = "Sena"`

`bucketIndex = 1` `head = null`

`size = 3` `node = node(41)`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
▶  node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`node.next` points to the old head (21), so the existing chain is kept.

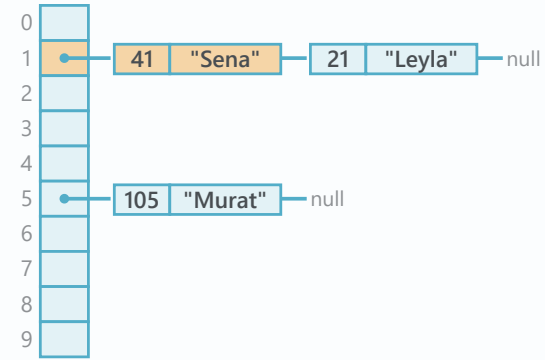
Step 16 / 17

VARIABLES

`key = 41` `value = "Sena"` `bucketIndex = 1` `head = null` `size = 3` `node = node(41)`

Code Trace: `put(21)`, `put(41)`

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
▶    buckets[bucketIndex] = node;   // insert at head
17 }
```



`buckets[1] = node` : the new node is at the **head** of the chain. No need to walk to the end → O(1).

Step 17 / 17

VARIABLES

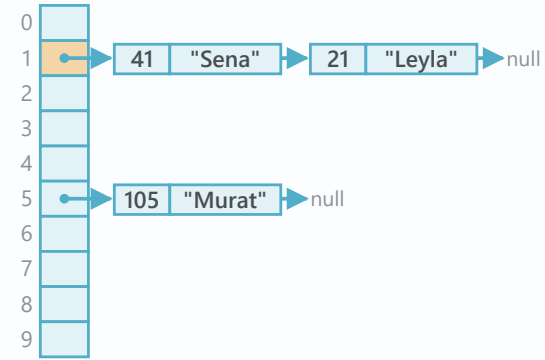
`key = 41` `value = "Sena"`

`bucketIndex = 1` `head = null`

`size = 3` `node = node(41)`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`put(21, \"Ali\")` : h(21) = 21 mod 10 = **1** → only bucket 1 is examined.

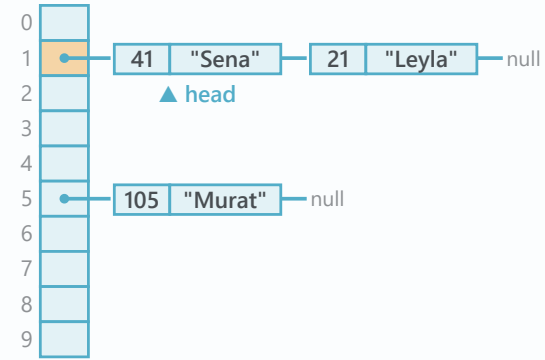
Step 1 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = –` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head` points to the first node of the chain.

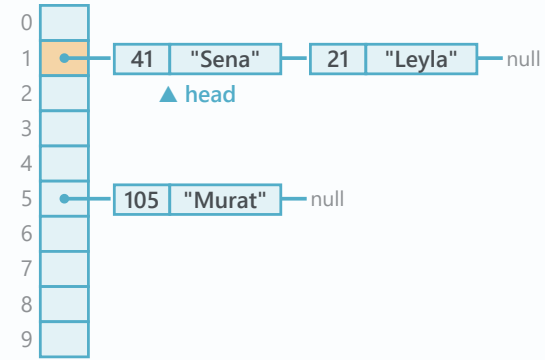
Step 2 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(41)` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
▶   while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head != null` → enter the loop to examine this node.

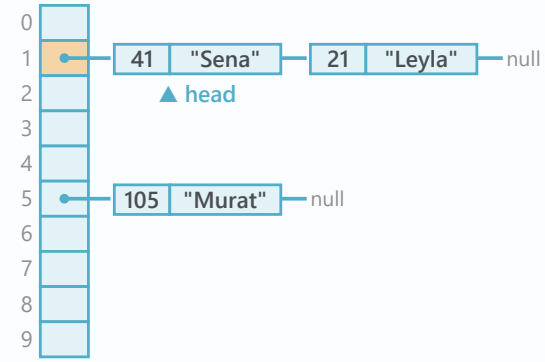
Step 3 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(41)` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
▶      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



41 ≠ 21 → no match; move on.

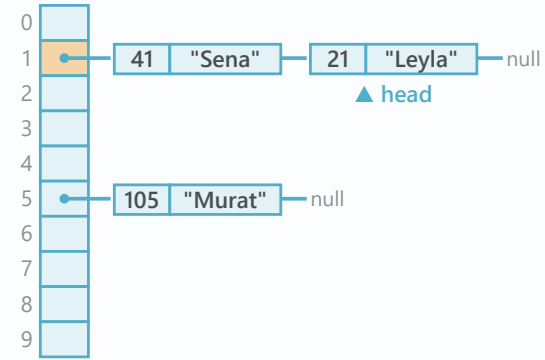
Step 4 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(41)` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
▶      head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head = head.next` → node 21.

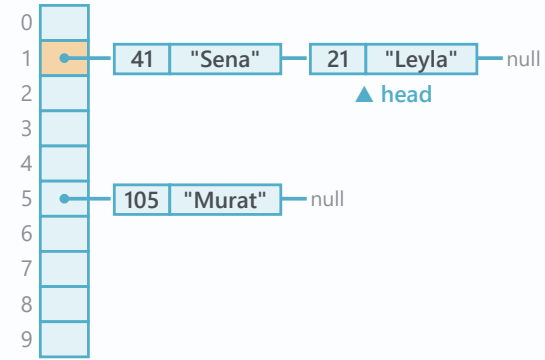
Step 5 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(21)` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
▶   while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`head != null` → enter the loop to examine this node.

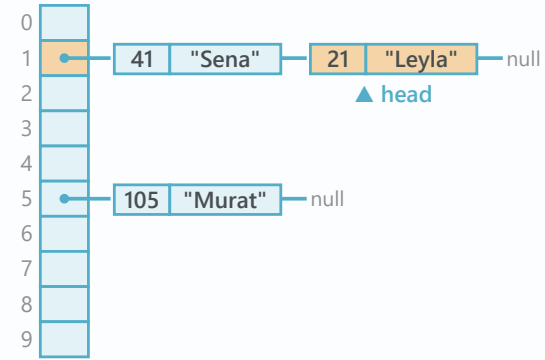
Step 6 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(21)` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
▶      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`21.equals(21)` → **equal**: the key is already in the table.

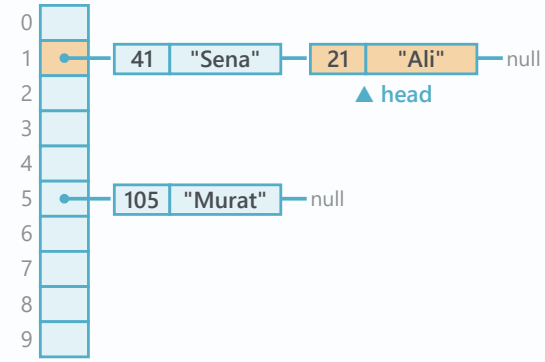
Step 7 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(21)` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
▶        head.value = value;   // yes: update
8        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



The value is updated: "Leyla" → **"Ali"**. No new node is added.

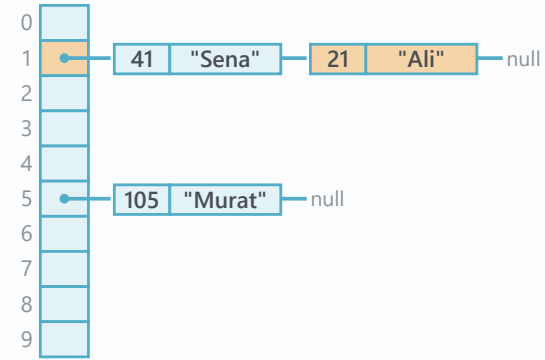
Step 8 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(21)` `size = 3`

Code Trace: `put(21, "Ali")` (update)

```
1  public void put(Integer key,
2                  String value) {
3    int bucketIndex = getBucketIndex(key);
4    HashNode head = buckets[bucketIndex];
5    while (head != null) {    // key present?
6      if (head.key.equals(key)) {
7        head.value = value;   // yes: update
▶        return;
9      }
10     head = head.next;
11   }
12   size++;
13   HashNode node =
14       new HashNode(key, value);
15   node.next = buckets[bucketIndex];
16   buckets[bucketIndex] = node;   // insert at head
17 }
```



`return` : **size = 3** is unchanged (same number of pairs).

Step 9 / 9

VARIABLES

`key = 21` `value = "Ali"` `bucketIndex = 1` `head = node(21)` `size = 3`

Looking Up a Value by Key

```
public String get(Integer key) {
  int bucketIndex = getBucketIndex(key);
  HashNode head = buckets[bucketIndex];
  while (head != null) {
    if (head.key.equals(key))
      return head.value;          // found
    head = head.next;
  }
  return null;                    // no such key
}
```

Just one bucket

No other bucket is ever examined.

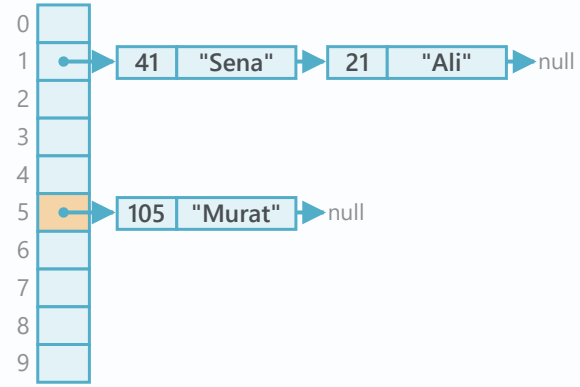
Cost

O(1 + chain length). The average chain length is n/N.

Call	Bucket	Nodes examined	Returns
`table.get(105)`	105 % 10 = 5	105	"Murat"
`table.get(21)`	21 % 10 = 1	41, 21	"Ali"
`table.get(88)`	88 % 10 = 8	— (empty bucket)	`null`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`get(105)`: h(105) = 105 mod 10 = **5**. No other bucket is examined.

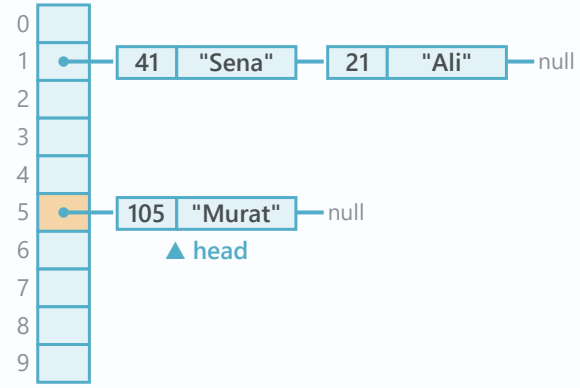
Step 1 / 17

VARIABLES

`key = 105` `bucketIndex = 5` `head = –` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`head` points to the bucket's first node.

Step 2 / 17

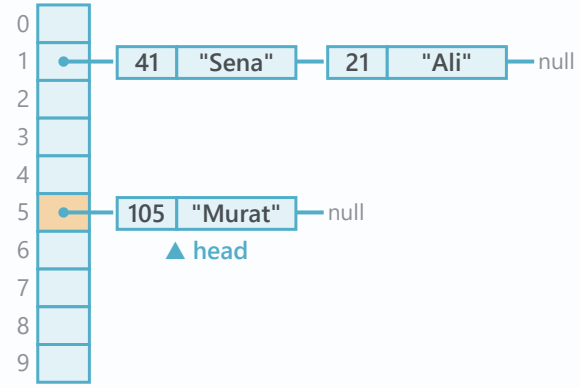
VARIABLES

`key = 105` `bucketIndex = 5`

`head = node(105)` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
▶    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`head != null` → enter the loop body.

Step 3 / 17

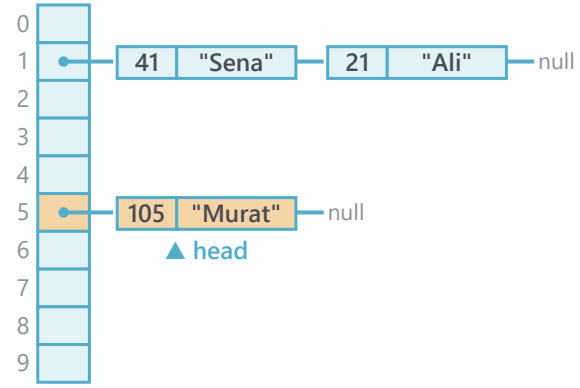
VARIABLES

`key = 105` `bucketIndex = 5`

`head = node(105)` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
▶      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



105.equals(105) → **equal**.

Step 4 / 17

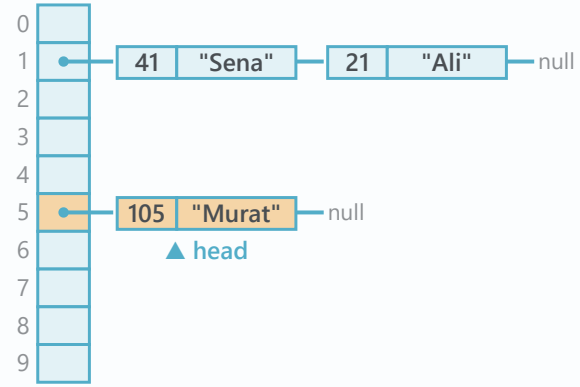
VARIABLES

`key = 105` `bucketIndex = 5`

`head = node(105)` `comparisons = 1`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`return "Murat"` : one comparison was enough.

Step 5 / 17

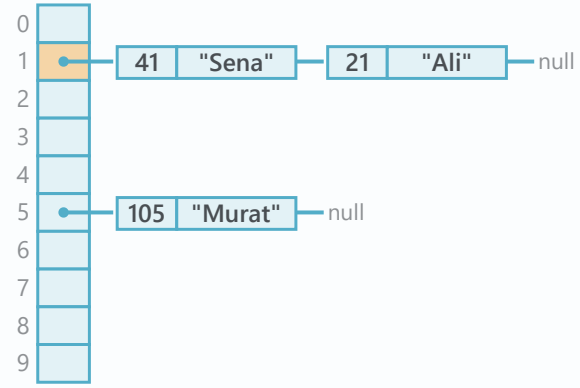
VARIABLES

`key = 105` `bucketIndex = 5`

`head = node(105)` `comparisons = 1`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`get(21)`: h(21) = 21 mod 10 = **1**. No other bucket is examined.

Step 6 / 17

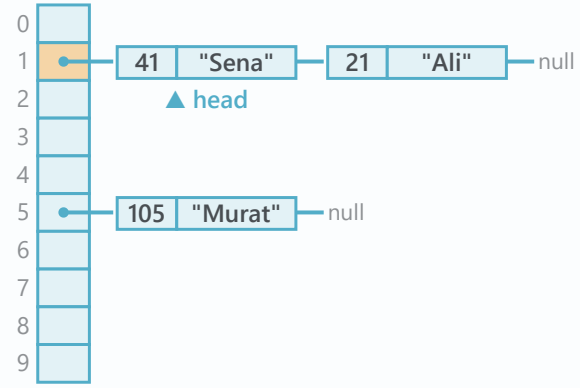
VARIABLES

`key = 21` `bucketIndex = 1`

`head = –` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`head` points to the bucket's first node.

Step 7 / 17

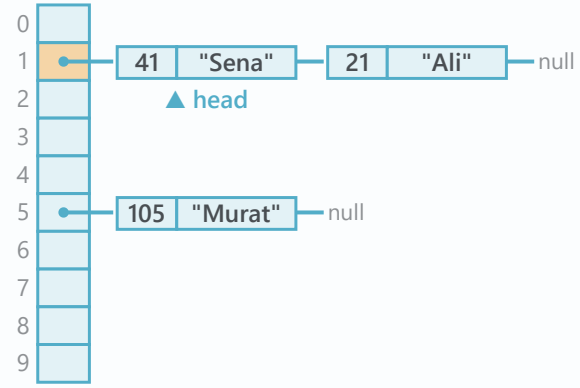
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(41)` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
▶    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`head != null` → enter the loop body.

Step 8 / 17

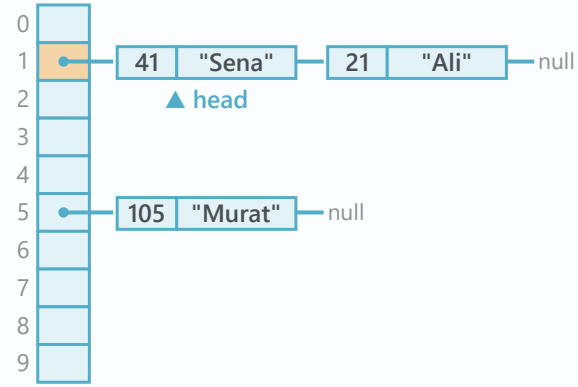
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(41)` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
▶      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



41 ≠ 21 → another key that landed in the same bucket (a collision).

Step 9 / 17

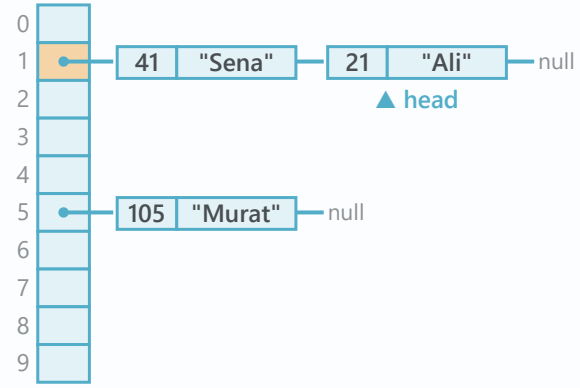
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(41)` `comparisons = 1`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
▶      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`head = head.next` → node 21.

Step 10 / 17

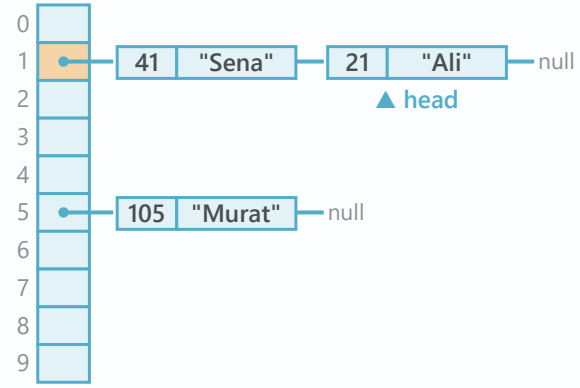
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(21)` `comparisons = 1`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
▶    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`head != null` → enter the loop body.

Step 11 / 17

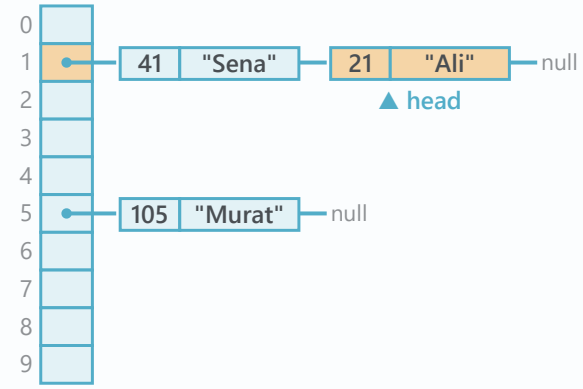
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(21)` `comparisons = 1`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
▶      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



21.equals(21) → **equal**.

Step 12 / 17

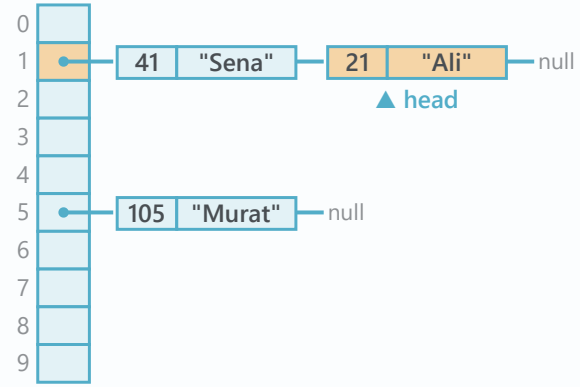
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(21)` `comparisons = 2`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
▶        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`return "Ali"` : two comparisons were enough.

Step 13 / 17

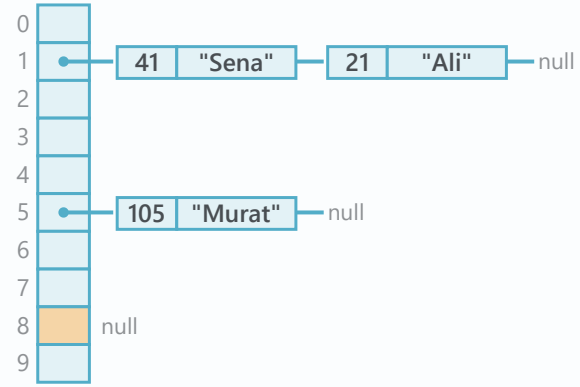
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(21)` `comparisons = 2`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`get(88)`: h(88) = 88 mod 10 = **8**. No other bucket is examined.

Step 14 / 17

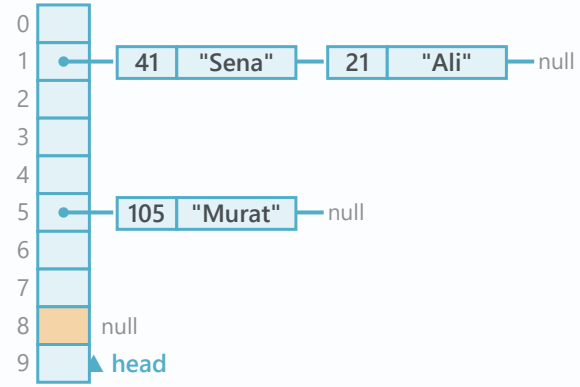
VARIABLES

`key = 88` `bucketIndex = 8`

`head = –` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



Bucket 8 is empty: `head = null`.

Step 15 / 17

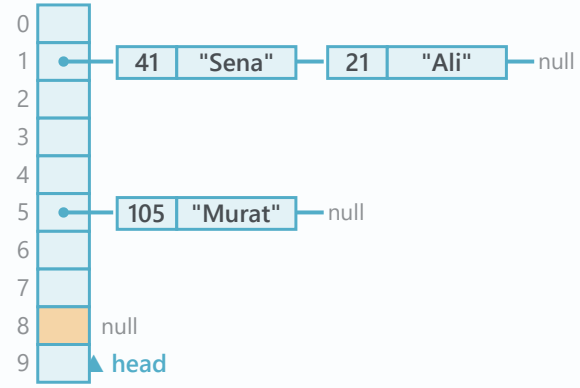
VARIABLES

`key = 88` `bucketIndex = 8`

`head = null` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
▶    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
9    return null;                    // no such key
10 }
```



`head == null` → the loop body never runs.

Step 16 / 17

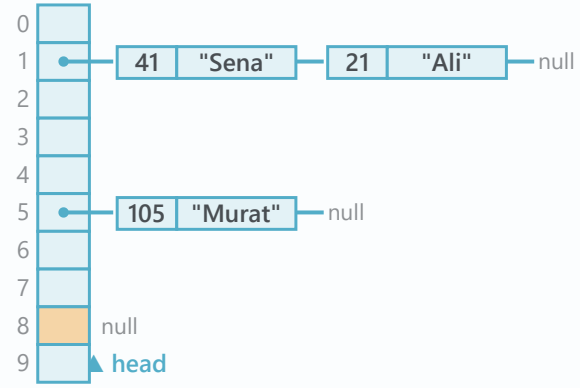
VARIABLES

`key = 88` `bucketIndex = 8`

`head = null` `comparisons = 0`

Code Trace: `get(105)`, `get(21)`, `get(88)`

```
1  public String get(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    while (head != null) {
5      if (head.key.equals(key))
6        return head.value;          // found
7      head = head.next;
8    }
▶    return null;                    // no such key
10 }
```



`return null` : 88 is not in the table. An empty bucket answers in one step.

Step 17 / 17

VARIABLES

`key = 88` `bucketIndex = 8`

`head = null` `comparisons = 0`

Removing a Key from a Hash Table

```java
public String remove(Integer key) {
  int bucketIndex = getBucketIndex(key);
  HashNode head = buckets[bucketIndex];
  HashNode prev = null;
  while (head != null &&
         !head.key.equals(key)) {
    prev = head;
    head = head.next;
  }
  if (head == null) return null;   // absent
  size--;
  if (prev != null)        // middle/end
    prev.next = head.next;
  else                          // head
    buckets[bucketIndex] = head.next;
  return head.value;
}
```

prev

Unlinking a node from a singly linked list requires the node before it.

Two cases

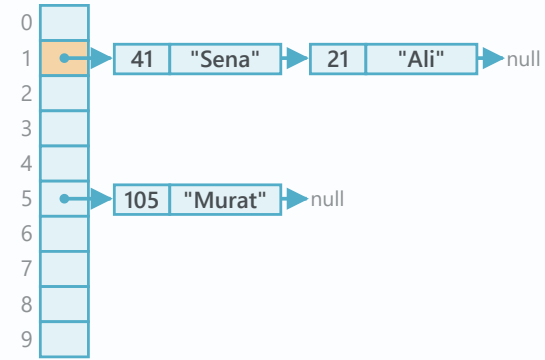
If the node is the head, update the bucket; otherwise update `prev.next`.

Cost

Same as lookup: O(1 + chain length).

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
▶    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`remove(21)` : h(21) = 21 mod 10 = **1**.

Step 1 / 12

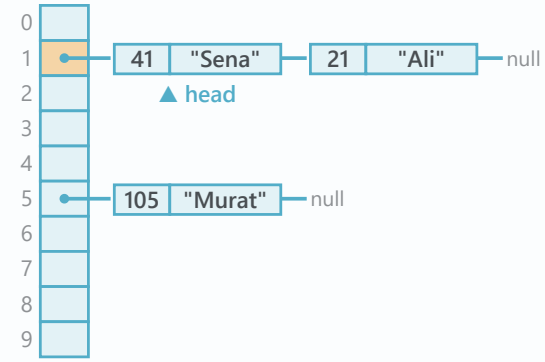
VARIABLES

`key = 21` `bucketIndex = 1` `head = –`

`prev = –` `size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`head` points to the first node.

Step 2 / 12

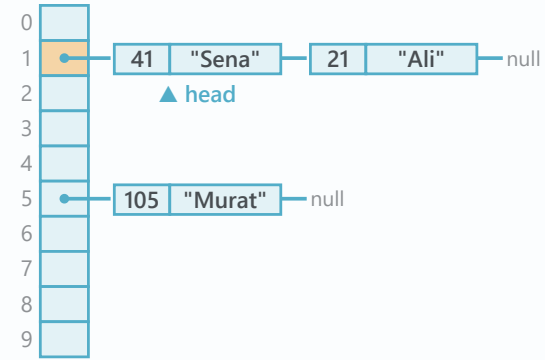
VARIABLES

`key = 21` `bucketIndex = 1`

`head = node(41)` `prev = –` `size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
▶    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`prev = null` : to unlink a node from a singly linked list we need the node before it.

Step 3 / 12

VARIABLES

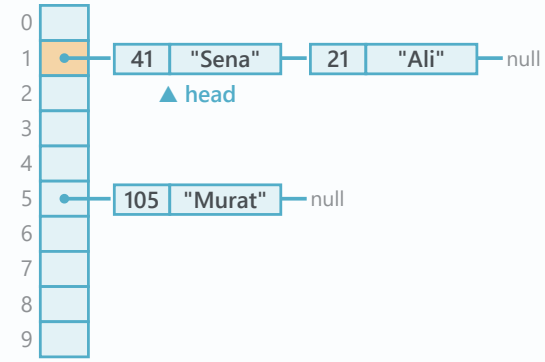
`key = 21` `bucketIndex = 1`

`head = node(41)` `prev = null`

`size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
▶    while (head != null &&
▶           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



41 ≠ 21 → the condition holds; enter the loop body.

Step 4 / 12

VARIABLES

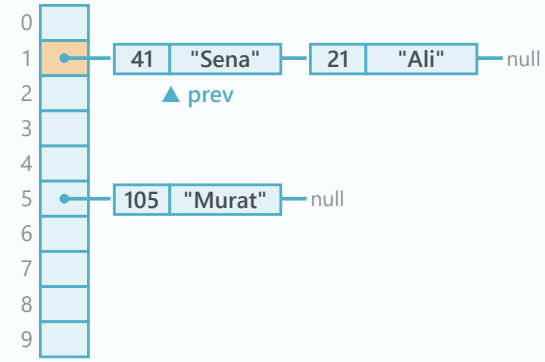
`key = 21` `bucketIndex = 1`

`head = node(41)` `prev = null`

`size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
▶      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`prev = head` → node 41.

Step 5 / 12

VARIABLES

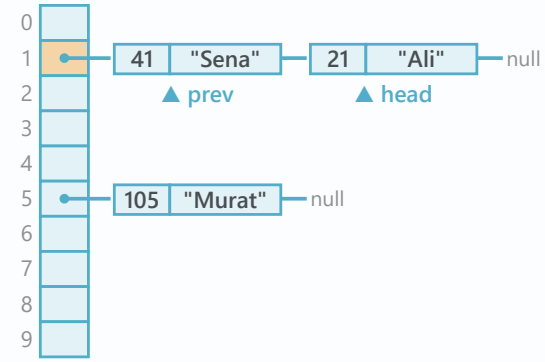
`key = 21` `bucketIndex = 1`

`head = node(41)` `prev = node(41)`

`size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
▶      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`head = head.next` → node 21.

Step 6 / 12

VARIABLES

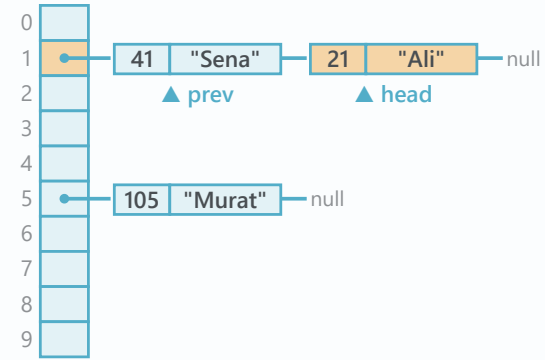
`key = 21` `bucketIndex = 1`

`head = node(21)` `prev = node(41)`

`size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
▶    while (head != null &&
▶           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



21.equals(21) → equal, so the loop ends: we've found the node to remove.

Step 7 / 12

VARIABLES

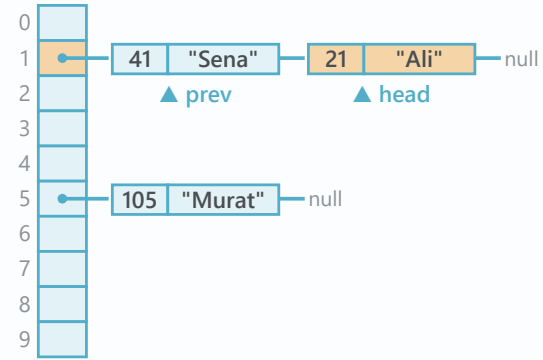
`key = 21` `bucketIndex = 1`

`head = node(21)` `prev = node(41)`

`size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
▶   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`head != null` → the key is present; remove it.

VARIABLES

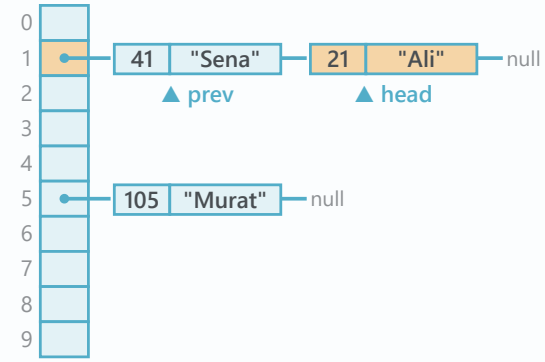
`key = 21` `bucketIndex = 1`

`head = node(21)` `prev = node(41)`

`size = 3`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
▶    size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



size = 2.

Step 9 / 12

VARIABLES

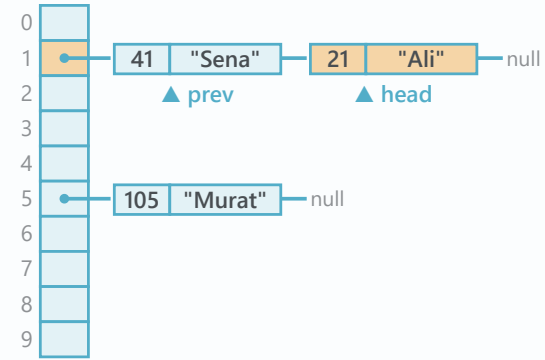
`key = 21` `bucketIndex = 1`

`head = node(21)` `prev = node(41)`

`size = 2`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
▶ 12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                        // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`prev != null` → the node is in the middle or at the end: bypass it through `prev`.

Step 10 / 12

VARIABLES

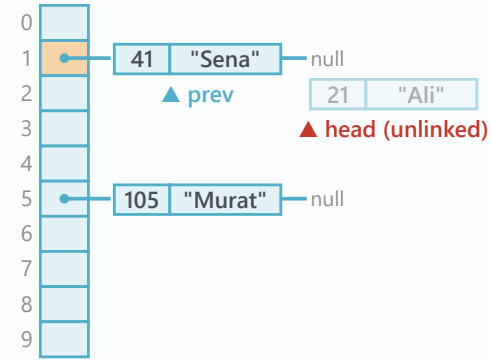
`key = 21` `bucketIndex = 1`

`head = node(21)` `prev = node(41)`

`size = 2`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
▶      prev.next = head.next;
14   else                        // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`prev.next = head.next` (null): 41 now skips 21, which is unlinked from the chain.

Step 11 / 12

VARIABLES

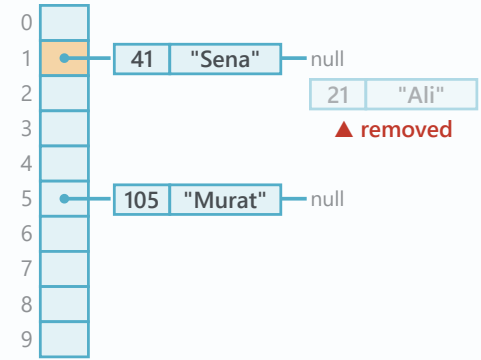
`key = 21` `bucketIndex = 1`

`head = node(21)` `prev = node(41)`

`size = 2`

Code Trace: `remove(21)` (from the middle of a chain)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)        // middle/end
13     prev.next = head.next;
14   else                       // head
15     buckets[bucketIndex] = head.next;
▶    return head.value;
17 }
```



`return \"Ali\"`. The garbage collector reclaims the unlinked node.

Step 12 / 12

VARIABLES

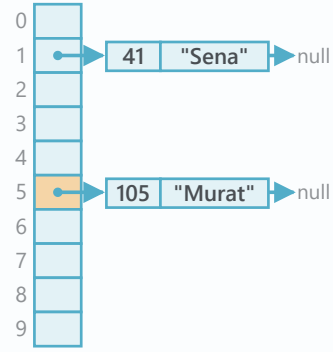
`key = 21` `bucketIndex = 1`

`head = node(21)` `prev = node(41)`

`size = 2`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                     // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`remove(105)` : h(105) = 105 mod 10 = **5**.

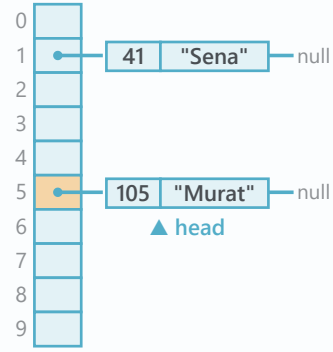
Step 1 / 14

VARIABLES

`key = 105` `bucketIndex = 5` `head = –` `prev = –` `size = 2`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`head` points to the first node.

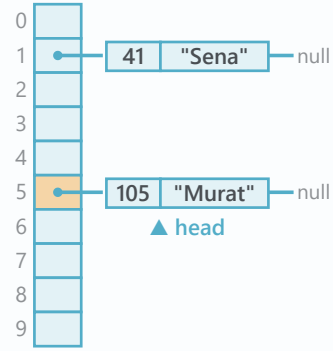
Step 2 / 14

VARIABLES

`key = 105` `bucketIndex = 5` `head = node(105)` `prev = –` `size = 2`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
▶    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                     // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`prev = null` : to unlink a node from a singly linked list we need the node before it.

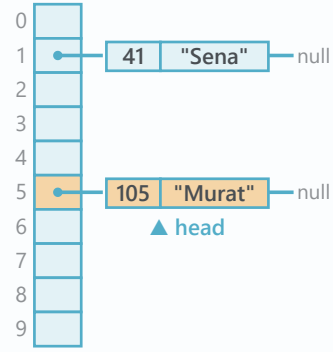
Step 3 / 14

VARIABLES

`key = 105` `bucketIndex = 5` `head = node(105)` `prev = null` `size = 2`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
▶    while (head != null &&
▶           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



105.equals(105) → equal, so the loop ends: we've found the node to remove.

Step 4 / 14

VARIABLES

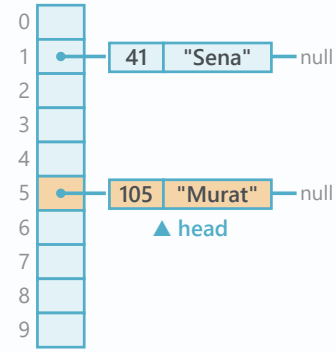
`key = 105` `bucketIndex = 5`

`head = node(105)` `prev = null`

`size = 2`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
▶    if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                     // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`head != null` → the key is present; remove it.

Step 5 / 14

VARIABLES

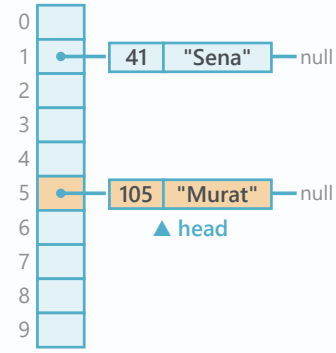
`key = 105` `bucketIndex = 5`

`head = node(105)` `prev = null`

`size = 2`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
▶    size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                     // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



size = 1.

Step 6 / 14

VARIABLES

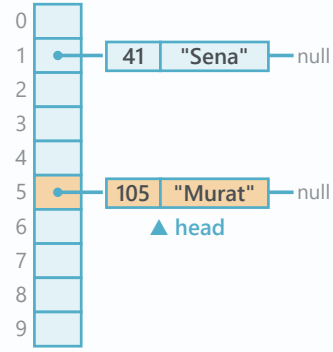
`key = 105` `bucketIndex = 5`

`head = node(105)` `prev = null`

`size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
▶    if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`prev == null` → the node to remove is the head of the chain: take the `else` branch.

Step 7 / 14

VARIABLES

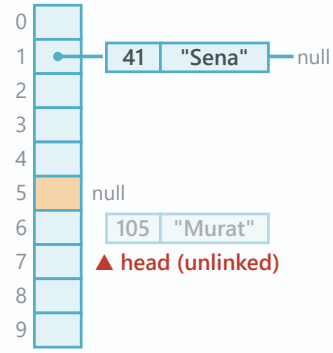
`key = 105` `bucketIndex = 5`

`head = node(105)` `prev = null`

`size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
▶      buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`buckets[5] = head.next` (null): the bucket no longer points to 105.

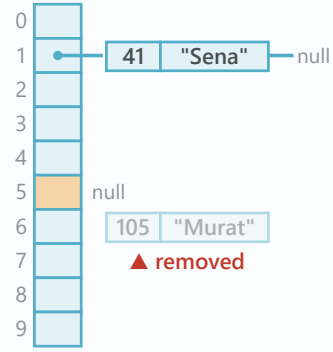
Step 8 / 14

VARIABLES

`key = 105` `bucketIndex = 5` `head = node(105)` `prev = null` `size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
▶    return head.value;
17 }
```



`return \"Murat\"`. The garbage collector reclaims the unlinked node.

Step 9 / 14

VARIABLES

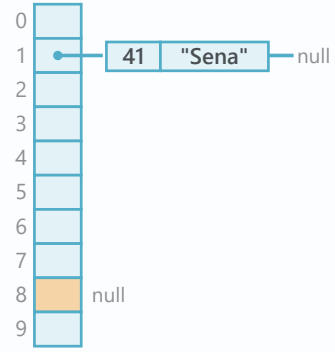
`key = 105` `bucketIndex = 5`

`head = node(105)` `prev = null`

`size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
▶    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`remove(88)` : h(88) = 88 mod 10 = **8**.

Step 10 / 14

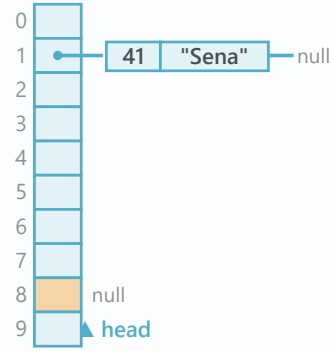
VARIABLES

`key = 88` `bucketIndex = 8` `head = –`

`prev = –` `size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
▶    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                      // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



Bucket 8 is empty: `head = null`.

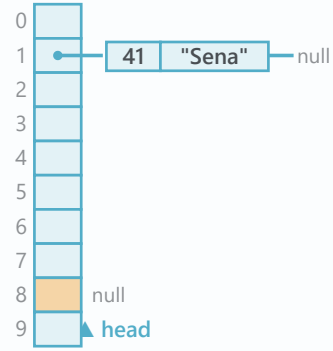
Step 11 / 14

VARIABLES

`key = 88` `bucketIndex = 8` `head = null` `prev = –` `size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
▶    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
10   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                     // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`prev = null` : to unlink a node from a singly linked list we need the node before it.

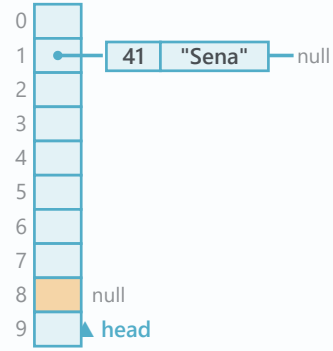
Step 12 / 14

VARIABLES

`key = 88` `bucketIndex = 8` `head = null` `prev = null` `size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1 public String remove(Integer key) {
2   int bucketIndex = getBucketIndex(key);
3   HashNode head = buckets[bucketIndex];
4   HashNode prev = null;
▶   while (head != null &&
▶          !head.key.equals(key)) {
7     prev = head;
8     head = head.next;
9   }
10  if (head == null) return null;   // absent
11  size--;
12  if (prev != null)       // middle/end
13    prev.next = head.next;
14  else                     // head
15    buckets[bucketIndex] = head.next;
16  return head.value;
17 }
```



`head == null` → the loop ends immediately.

Step 13 / 14

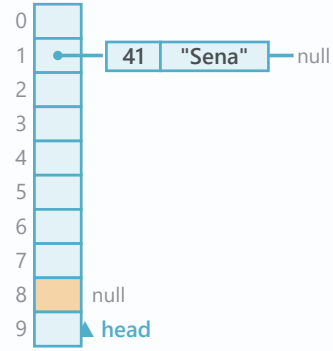
VARIABLES

`key = 88` `bucketIndex = 8`

`head = null` `prev = null` `size = 1`

Code Trace: `remove(105)` (head), `remove(88)` (absent)

```
1  public String remove(Integer key) {
2    int bucketIndex = getBucketIndex(key);
3    HashNode head = buckets[bucketIndex];
4    HashNode prev = null;
5    while (head != null &&
6           !head.key.equals(key)) {
7      prev = head;
8      head = head.next;
9    }
▶   if (head == null) return null;   // absent
11   size--;
12   if (prev != null)       // middle/end
13     prev.next = head.next;
14   else                     // head
15     buckets[bucketIndex] = head.next;
16   return head.value;
17 }
```



`return null` : 88 is absent; **size** is unchanged.

Step 14 / 14

VARIABLES

`key = 88` `bucketIndex = 8`

`head = null` `prev = null` `size = 1`

Open Addressing

- All pairs live in the array itself, at most one per slot ($n \le N$).
- On a collision, a **probe sequence** searches for an empty slot:
  `h(k, i) = (h(k) + f(i)) mod N`, $i = 0, 1, 2, \ldots$

Method	f(i)	Probe sequence (h(k) = 5, N = 13)	Weakness
Linear probing	i	5, 6, 7, 8, 9, …	Primary clustering
Quadratic probing	i^2	5, 6, 9, 14→1, 21→8, …	Secondary clustering; may not try every slot
Double hashing	$i \cdot h_2(k)$	5, 5+d, 5+2d, … (d = $h_2(k)$)	Needs $h_2(k) \neq 0$ and the step coprime to N

Compared with chaining: no extra nodes or references, and memory is contiguous (cache-friendly). **But:** the load factor must stay low, and deletion needs special care. If N is prime, every step from 1 to N−1 is coprime to N.

Linear Probing

order: 18 41 22 **44** 59 32 31 73



18 → 5, 41 → 2, 22 → 9: all three find an empty slot on the first try. Next up: 44.

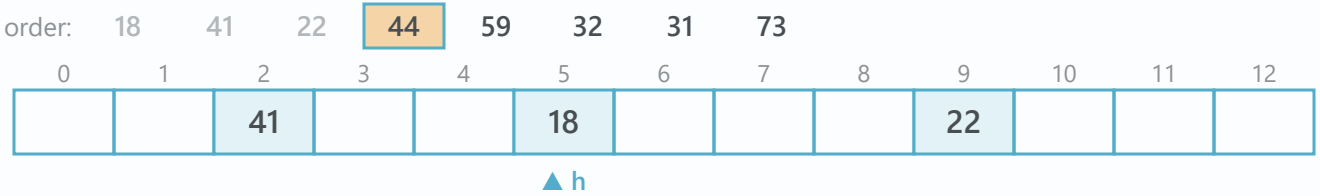
Step 1 / 13

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`N = 13`

Linear Probing



Next key 44: **h = 44 mod 13 = 5** (home slot).

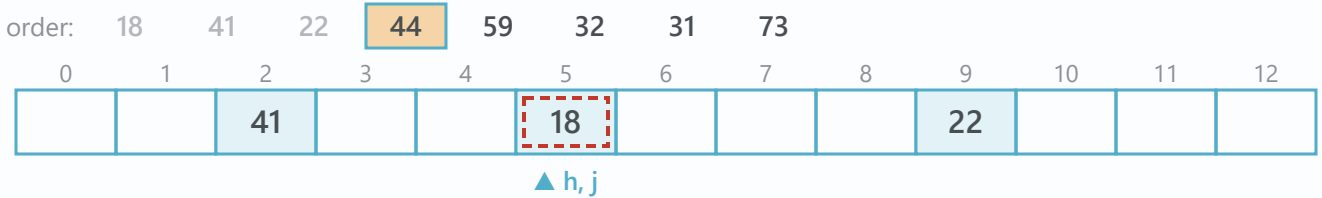
Step 2 / 13

```
1 // put(k): linear probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 44` `h = 5`

Linear Probing



i = 0: j = (5 + 0) mod 13 = **5** → occupied (18); next i.

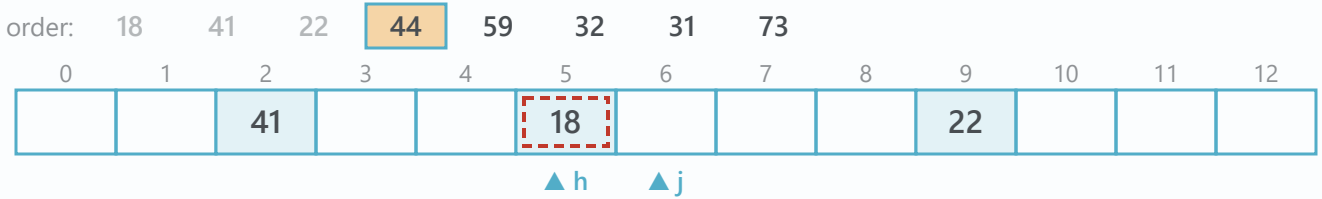
Step 3 / 13

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 44` `h = 5` `i = 0` `j = 5` `table[j] = 18`

Linear Probing



i = 1: j = (5 + 1) mod 13 = **6** → **empty**.

Step 4 / 13

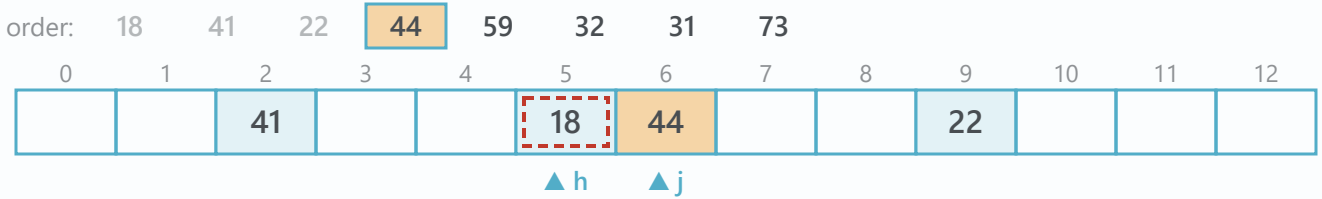
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 44` `h = 5` `i = 1` `j = 6`

`table[j] = null`

Linear Probing



`table[6] = 44`, `break` (2 probes).

Step 5 / 13

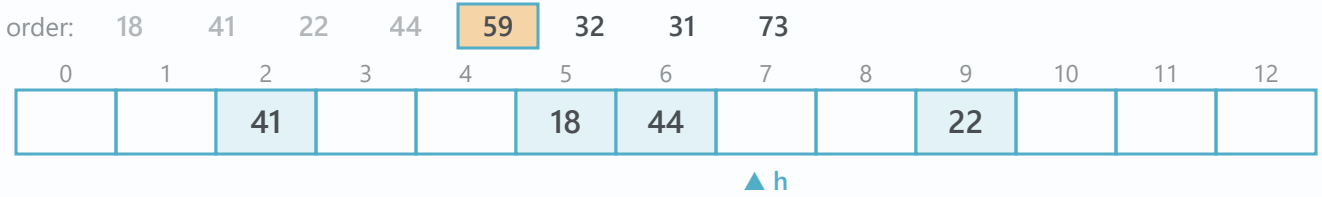
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 44` `h = 5` `i = 1` `j = 6`

`table[j] = 44`

Linear Probing



Next key 59: **h = 59 mod 13 = 7** (home slot).

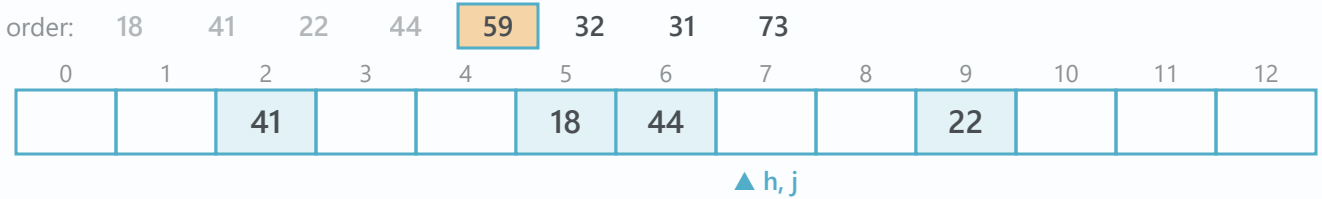
Step 6 / 13

```
1 // put(k): linear probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 59` `h = 7`

Linear Probing



i = 0: j = (7 + 0) mod 13 = **7** → **empty**.

Step 7 / 13

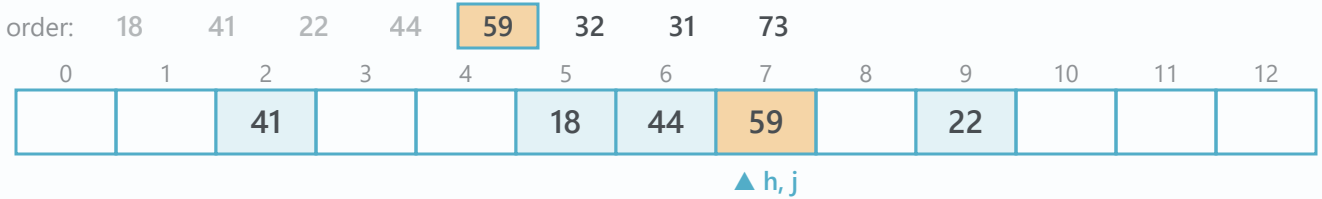
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

k = 59 | h = 7 | i = 0 | j = 7

table[j] = null

Linear Probing



`table[7] = 59` , `break` (1 probe).

Step 8 / 13

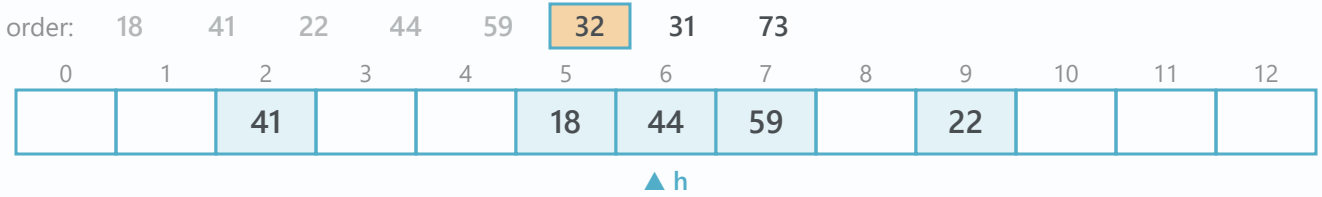
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 59` `h = 7` `i = 0` `j = 7`

`table[j] = 59`

Linear Probing



Next key 32: **h = 32 mod 13 = 6** (home slot).

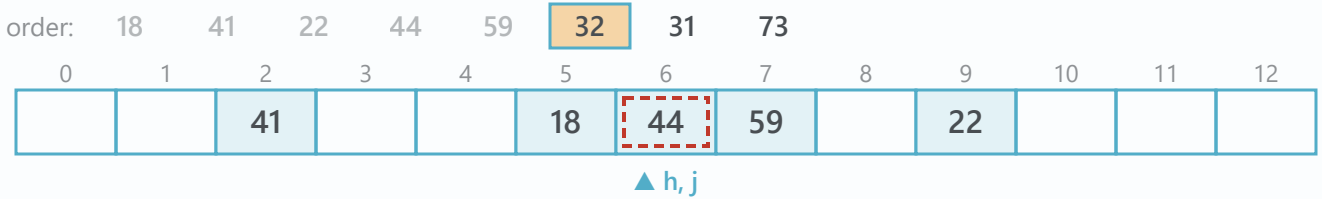
Step 9 / 13

```
1 // put(k): linear probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6`

Linear Probing



i = 0: j = (6 + 0) mod 13 = **6** → occupied (44); next i.

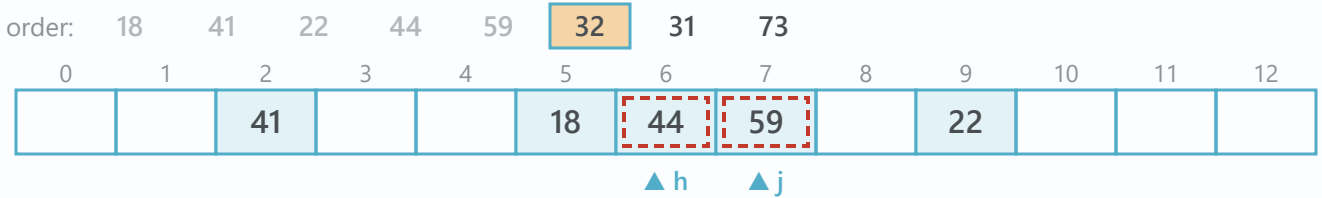
Step 10 / 13

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6` `i = 0` `j = 6` `table[j] = 44`

Linear Probing



i = 1: j = (6 + 1) mod 13 = **7** → occupied (59); next i.

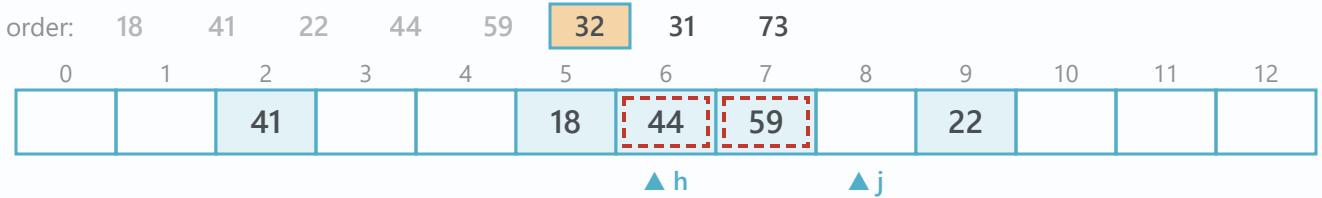
Step 11 / 13

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6` `i = 1` `j = 7` `table[j] = 59`

Linear Probing



i = 2: j = (6 + 2) mod 13 = **8** → **empty**.

Step 12 / 13

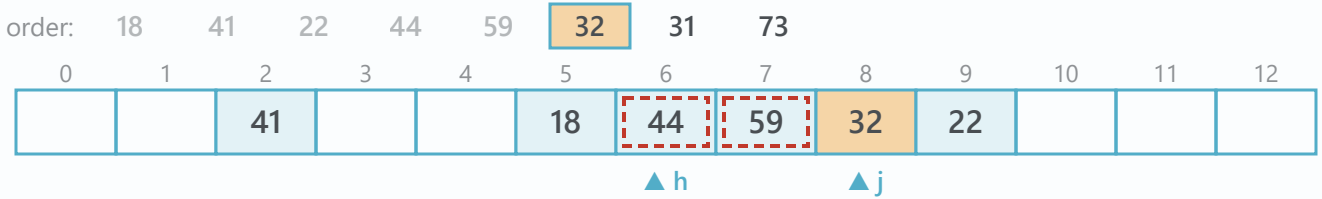
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6` `i = 2` `j = 8`

`table[j] = null`

Linear Probing



`table[8] = 32` , `break` (3 probes).

Step 13 / 13

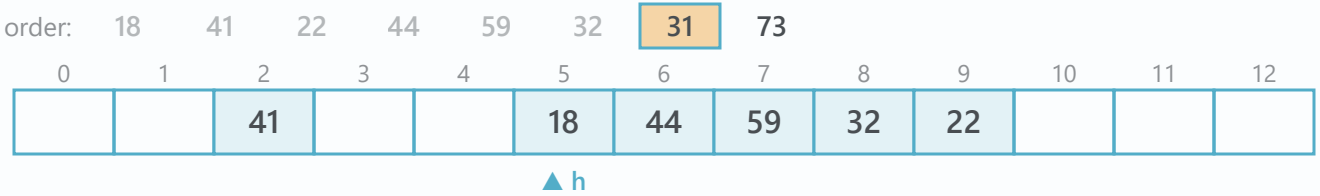
```
1  // put(k): linear probing, N = 13
2  int h = k % 13;                  // home slot
3  for (int i = 0; i < 13; i++) {
4    int j = (h + i) % 13;            // h, h+1, h+2…
5    if (table[j] == null) {        // empty?
▶      table[j] = k;
▶      break;
8    }                              // occupied: i++
9  }
```

VARIABLES

`k = 32` `h = 6` `i = 2` `j = 8`

`table[j] = 32`

Linear Probing (cont.): 31 and 73



Next key 31: **h = 31 mod 13 = 5** (home slot).

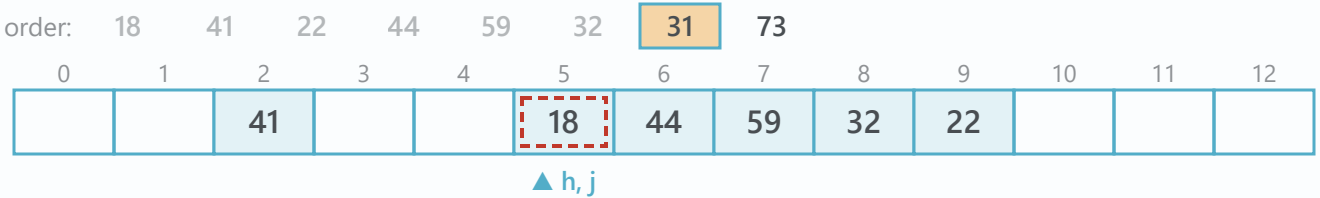
Step 1 / 14

```
1 // put(k): linear probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5`

Linear Probing (cont.): 31 and 73



i = 0: j = (5 + 0) mod 13 = **5** → occupied (18); next i.

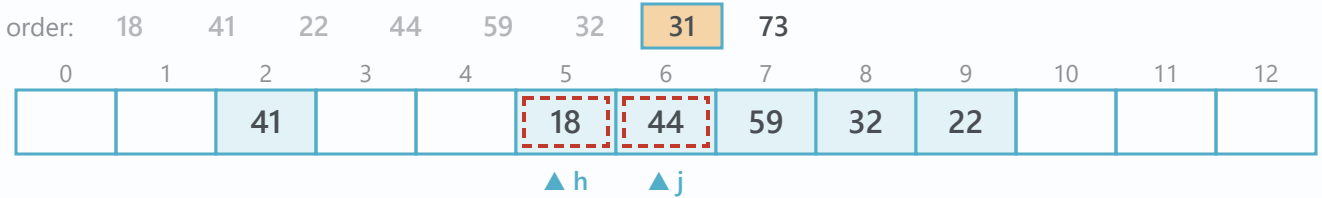
Step 2 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 0` `j = 5` `table[j] = 18`

Linear Probing (cont.): 31 and 73



i = 1: j = (5 + 1) mod 13 = **6** → occupied (44); next i.

Step 3 / 14

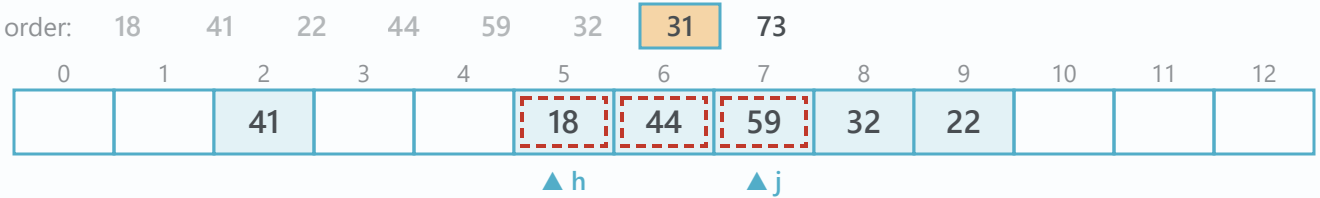
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 1` `j = 6`

`table[j] = 44`

Linear Probing (cont.): 31 and 73



i = 2: j = (5 + 2) mod 13 = **7** → occupied (59); next i.

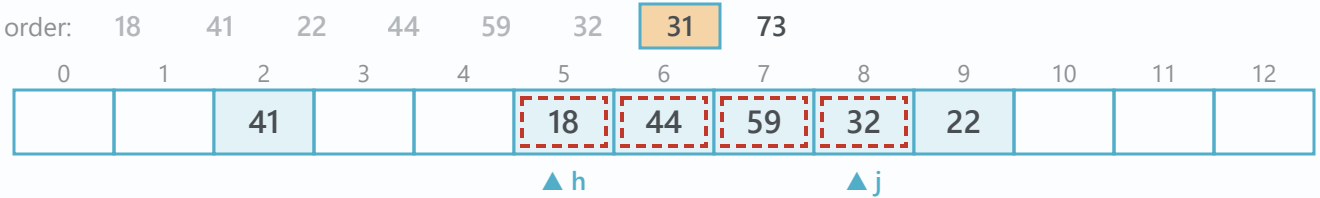
Step 4 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 2` `j = 7` `table[j] = 59`

Linear Probing (cont.): 31 and 73



i = 3: j = (5 + 3) mod 13 = **8** → occupied (32); next i.

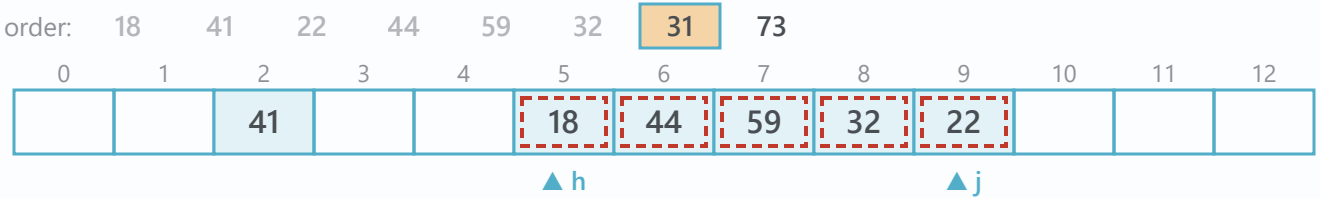
Step 5 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 3` `j = 8` `table[j] = 32`

Linear Probing (cont.): 31 and 73



i = 4: j = (5 + 4) mod 13 = **9** → occupied (22); next i.

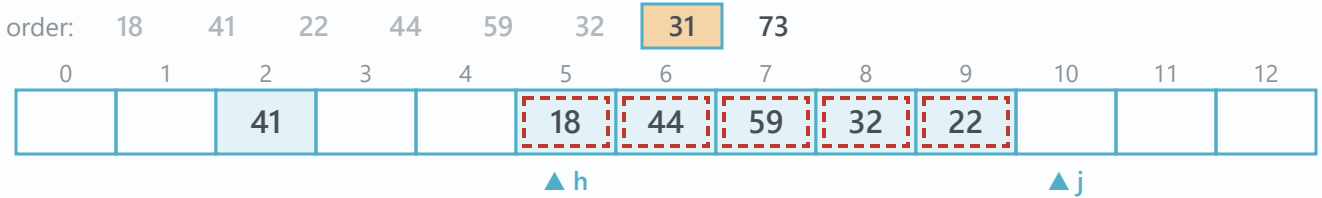
Step 6 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 4` `j = 9` `table[j] = 22`

Linear Probing (cont.): 31 and 73



i = 5: j = (5 + 5) mod 13 = **10** → **empty**.

Step 7 / 14

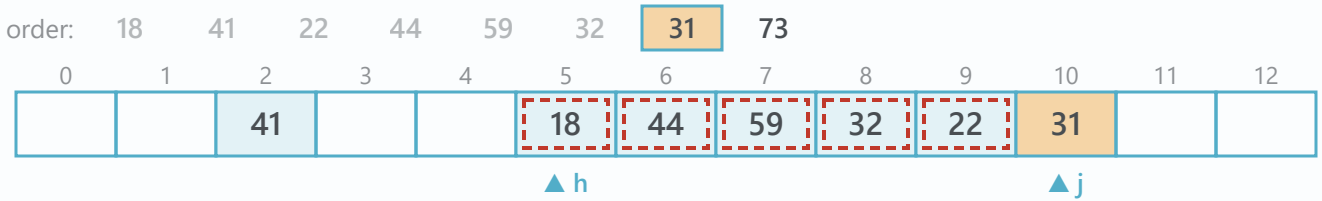
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 5` `j = 10`

`table[j] = null`

Linear Probing (cont.): 31 and 73



`table[10] = 31`, `break` (6 probes).

Step 8 / 14

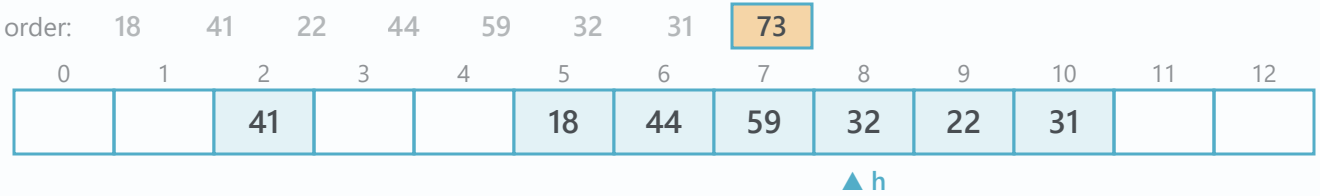
```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 5` `j = 10`

`table[j] = 31`

Linear Probing (cont.): 31 and 73



Next key 73: **h = 73 mod 13 = 8** (home slot).

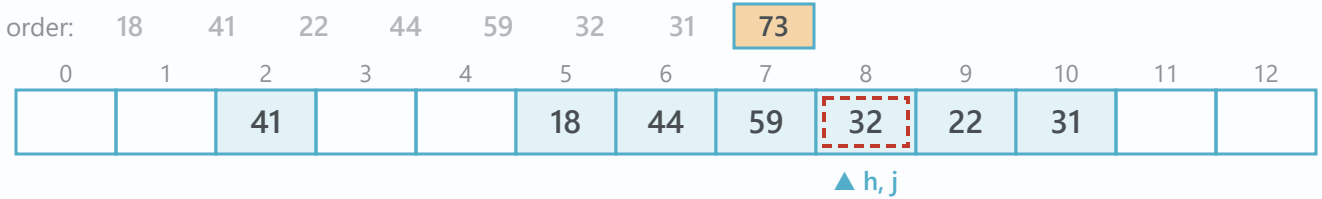
Step 9 / 14

```
1 // put(k): linear probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

k = 73 h = 8

Linear Probing (cont.): 31 and 73



i = 0: j = (8 + 0) mod 13 = **8** → occupied (32); next i.

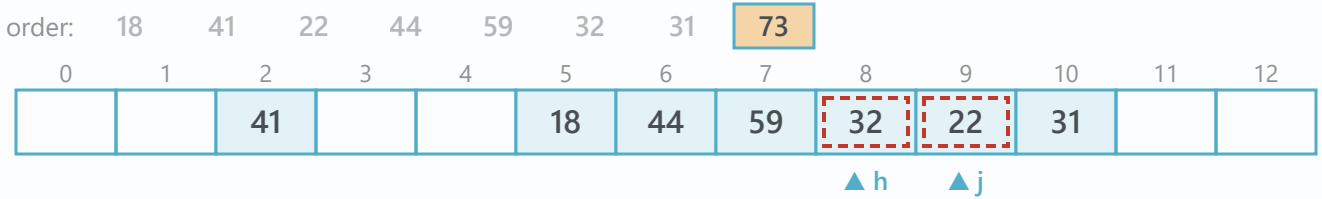
Step 10 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 73` `h = 8` `i = 0` `j = 8` `table[j] = 32`

Linear Probing (cont.): 31 and 73



i = 1: j = (8 + 1) mod 13 = **9** → occupied (22); next i.

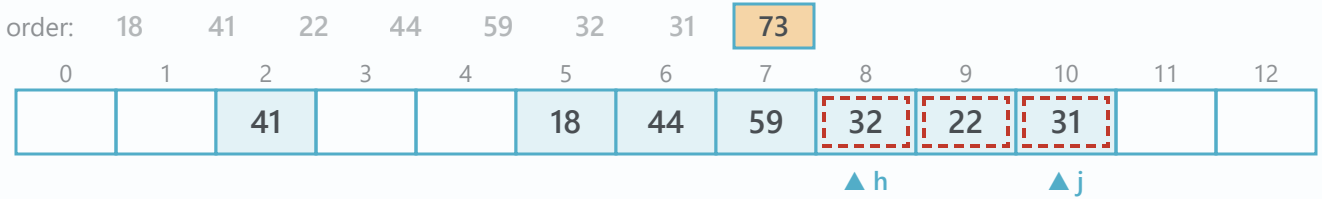
Step 11 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 73` `h = 8` `i = 1` `j = 9` `table[j] = 22`

Linear Probing (cont.): 31 and 73



i = 2: j = (8 + 2) mod 13 = **10** → occupied (31); next i.

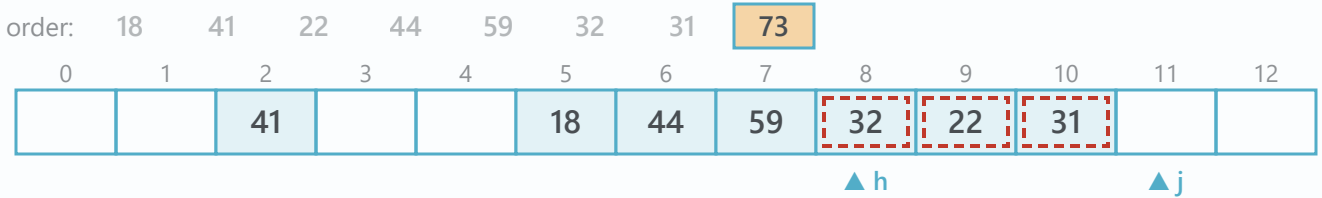
Step 12 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i) % 13;            // h, h+1, h+2…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 73` `h = 8` `i = 2` `j = 10` `table[j] = 31`

Linear Probing (cont.): 31 and 73



i = 3: j = (8 + 3) mod 13 = **11** → **empty**.

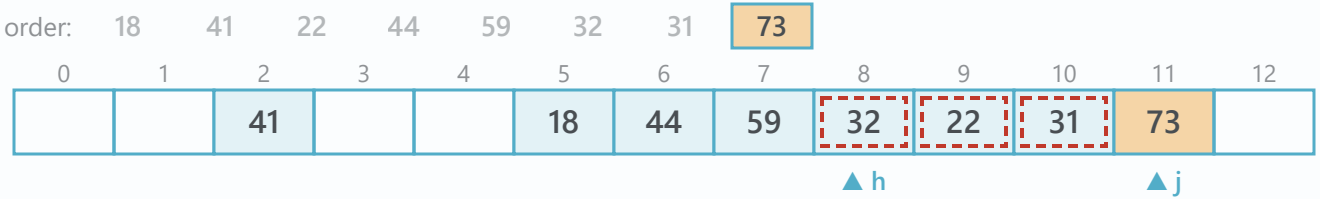
Step 13 / 14

```
1  // put(k): linear probing, N = 13
2  int h = k % 13;                  // home slot
3  for (int i = 0; i < 13; i++) {
▶    int j = (h + i) % 13;            // h, h+1, h+2…
▶    if (table[j] == null) {          // empty?
6      table[j] = k;
7      break;
8    }                                // occupied: i++
9  }
```

VARIABLES

`k = 73` `h = 8` `i = 3` `j = 11` `table[j] = null`

Linear Probing (cont.): 31 and 73



`table[11] = 73`, `break` (4 probes). Slots 5–11 form one big cluster: **primary clustering**.

Step 14 / 14

```
1 // put(k): linear probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i) % 13;            // h, h+1, h+2…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 73` `h = 8` `i = 3` `j = 11`

`table[j] = 73`

Quadratic Probing

order: 18 41 22 **44** 59 32 31 73



18 → 5, 41 → 2, 22 → 9: all three find an empty slot on the first try. Next up: 44.

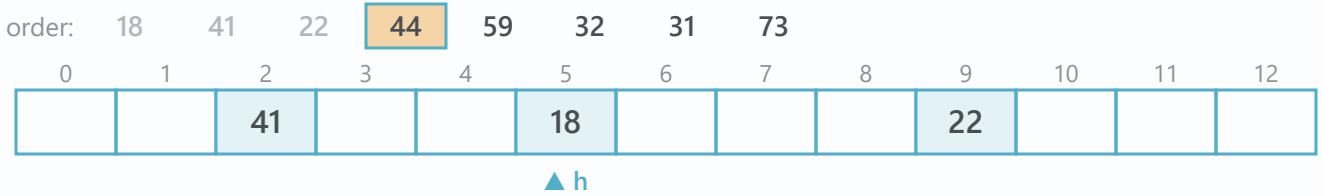
Step 1 / 13

```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;        // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

N = **13**

Quadratic Probing



Next key 44: **h = 44 mod 13 = 5** (home slot).

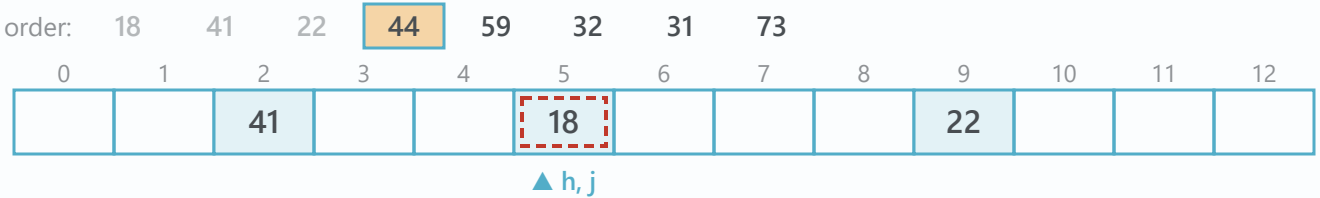
Step 2 / 13

```
1 // put(k): quadratic probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;        // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 44` `h = 5`

Quadratic Probing



i = 0: j = (5 + 0^2) mod 13 = **5** → occupied (18); next i.

Step 3 / 13

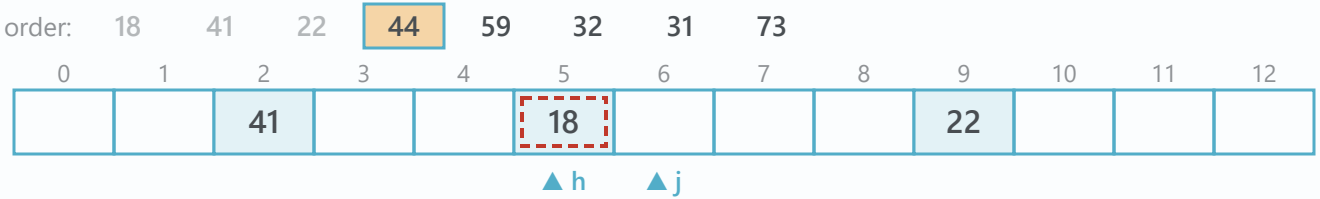
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

k = 44 | h = 5 | i = 0 | j = 5

table[j] = 18

Quadratic Probing



i = 1: j = (5 + 1^2) mod 13 = **6** → **empty**.

Step 4 / 13

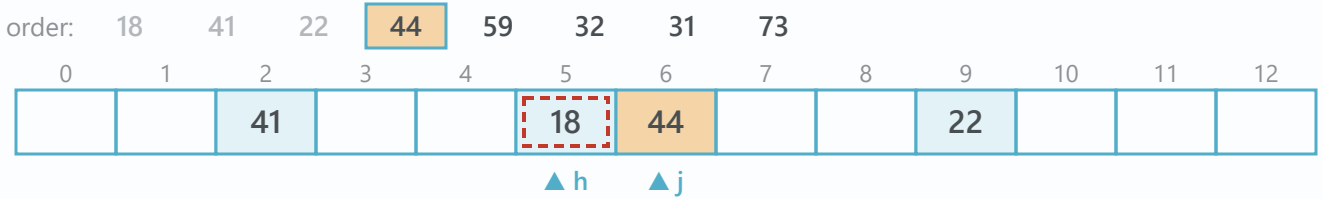
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 44` `h = 5` `i = 1` `j = 6`

`table[j] = null`

Quadratic Probing



`table[6] = 44`, `break` (2 probes).

Step 5 / 13

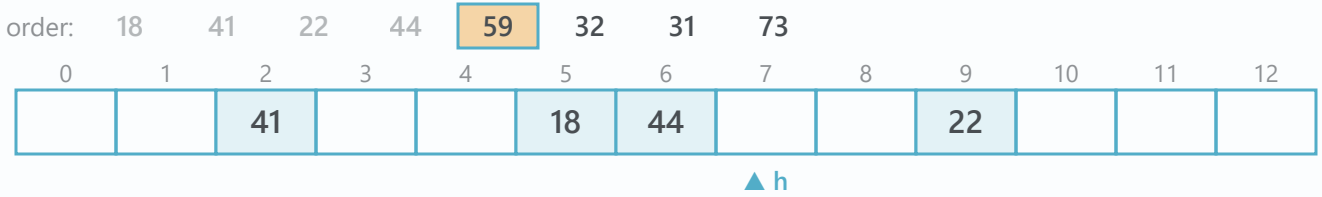
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;       // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 44` `h = 5` `i = 1` `j = 6`

`table[j] = 44`

Quadratic Probing



Next key 59: **h = 59 mod 13 = 7** (home slot).

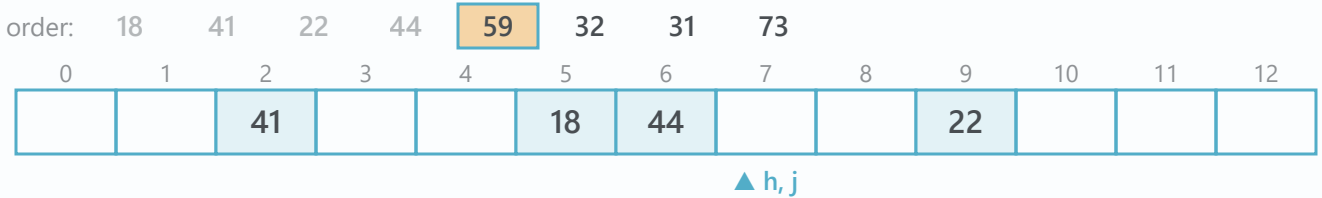
Step 6 / 13

```
1 // put(k): quadratic probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;       // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 59` `h = 7`

Quadratic Probing



i = 0: j = (7 + 0^2) mod 13 = **7** → **empty**.

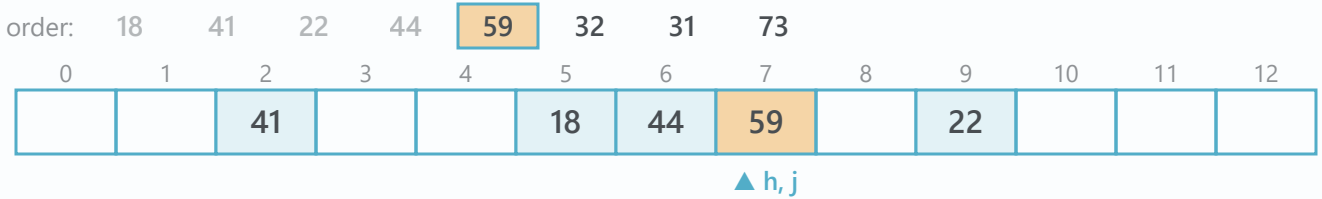
Step 7 / 13

```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 59` `h = 7` `i = 0` `j = 7` `table[j] = null`

Quadratic Probing



`table[7] = 59` , `break` (1 probe).

Step 8 / 13

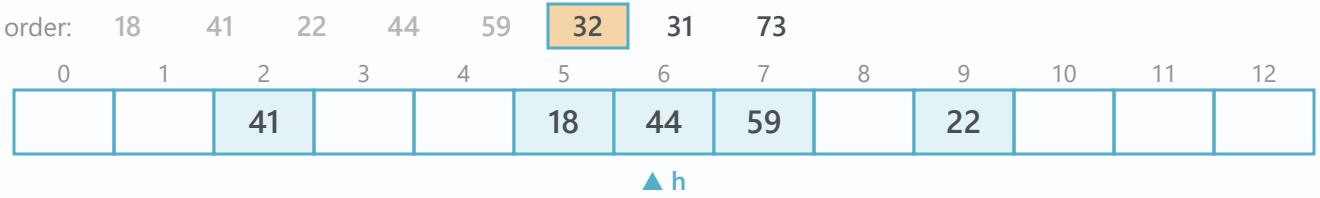
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;       // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 59` `h = 7` `i = 0` `j = 7`

`table[j] = 59`

Quadratic Probing



Next key 32: **h = 32 mod 13 = 6** (home slot).

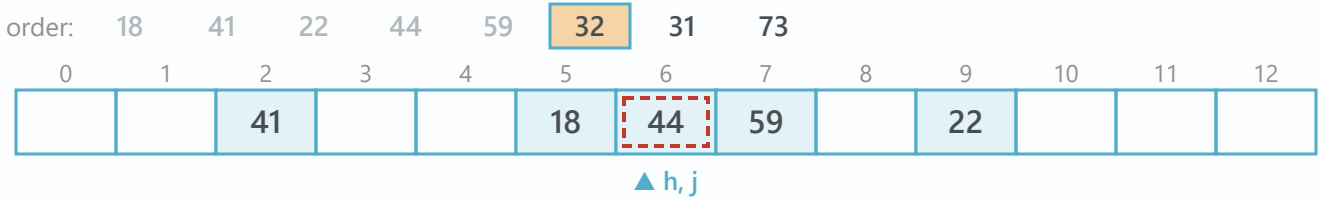
Step 9 / 13

```
1 // put(k): quadratic probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;        // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6`

Quadratic Probing



i = 0: j = (6 + 0^2) mod 13 = **6** → occupied (44); next i.

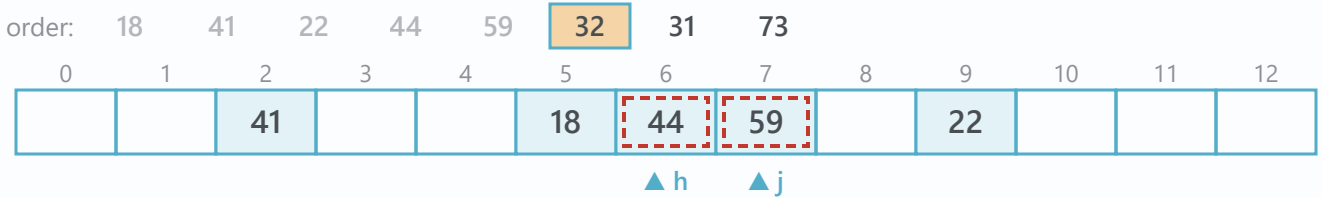
Step 10 / 13

```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6` `i = 0` `j = 6` `table[j] = 44`

Quadratic Probing



i = 1: j = (6 + 1^2) mod 13 = **7** → occupied (59); next i.

Step 11 / 13

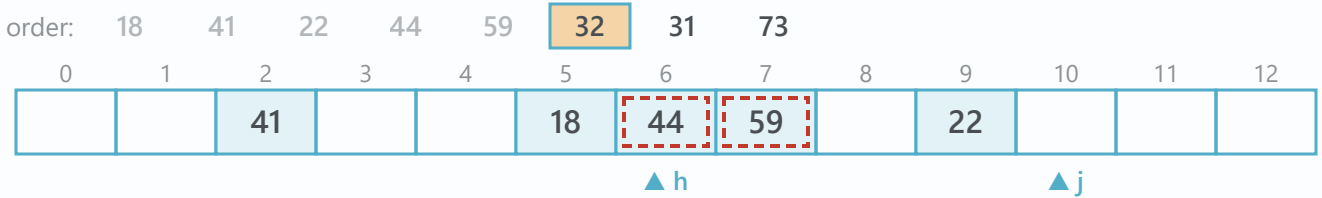
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

k = **32** h = **6** i = **1** j = **7**

table[j] = **59**

Quadratic Probing



i = 2: j = (6 + 2^2) mod 13 = **10** → **empty**.

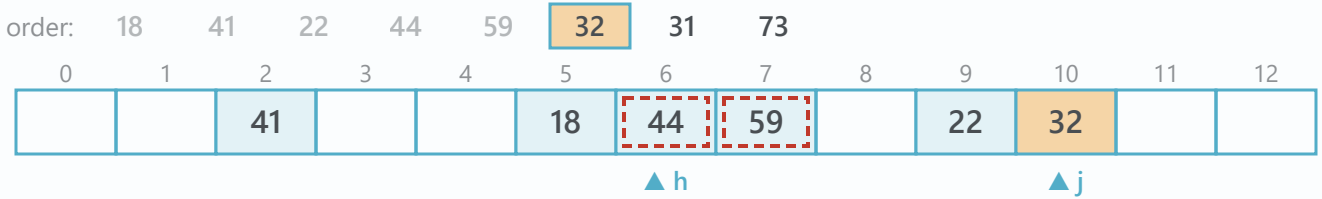
Step 12 / 13

```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6` `i = 2` `j = 10` `table[j] = null`

Quadratic Probing



`table[10] = 32`, `break` (3 probes).

Step 13 / 13

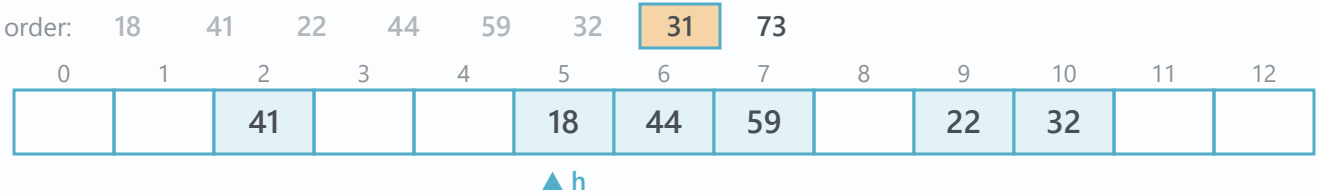
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;       // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 32` `h = 6` `i = 2` `j = 10`

`table[j] = 32`

Quadratic Probing (cont.): 31 and 73



Next key 31: **h = 31 mod 13 = 5** (home slot).

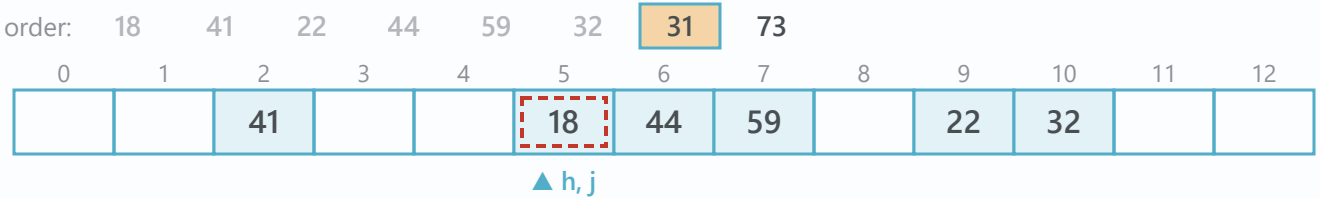
Step 1 / 9

```
1 // put(k): quadratic probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;        // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5`

Quadratic Probing (cont.): 31 and 73



i = 0: j = (5 + 0^2) mod 13 = **5** → occupied (18); next i.

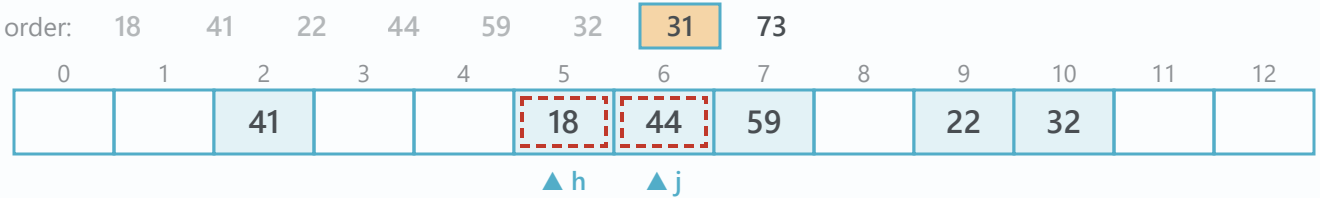
Step 2 / 9

```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 0` `j = 5` `table[j] = 18`

Quadratic Probing (cont.): 31 and 73



i = 1: j = (5 + 1^2) mod 13 = **6** → occupied (44); next i.

Step 3 / 9

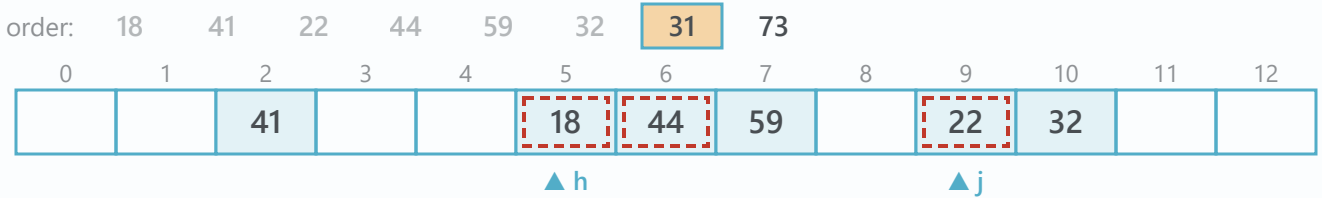
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

k = **31** h = **5** i = **1** j = **6**

table[j] = **44**

Quadratic Probing (cont.): 31 and 73



i = 2: j = (5 + 2^2) mod 13 = **9** → occupied (22); next i.

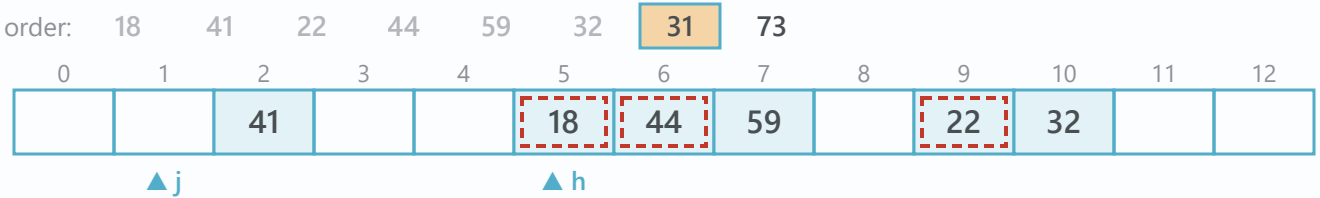
Step 4 / 9

```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 2` `j = 9` `table[j] = 22`

Quadratic Probing (cont.): 31 and 73



i = 3: j = (5 + 3^2) mod 13 = **1** → **empty**.

Step 5 / 9

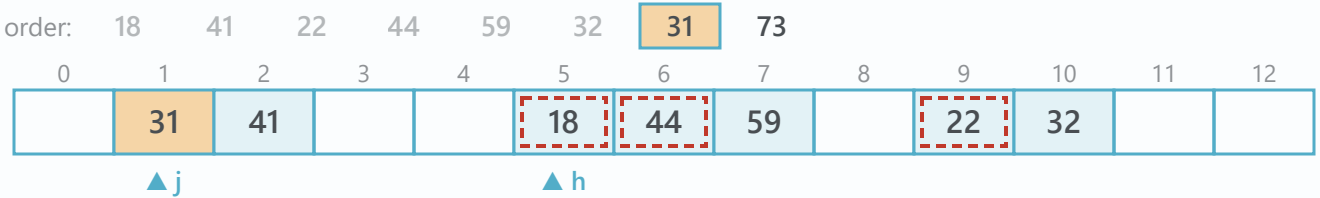
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 3` `j = 1`

`table[j] = null`

Quadratic Probing (cont.): 31 and 73



`table[1] = 31`, `break` (4 probes). The jumps grow, so the probe escapes the cluster quickly.

Step 6 / 9

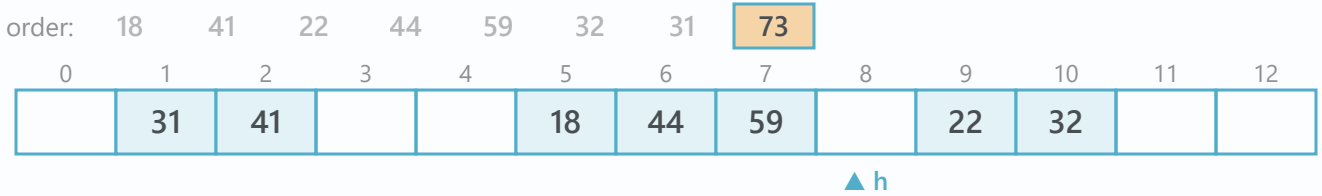
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;       // h, h+1, h+4…
5   if (table[j] == null) {         // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 31` `h = 5` `i = 3` `j = 1`

`table[j] = 31`

Quadratic Probing (cont.): 31 and 73



Next key 73: **h = 73 mod 13 = 8** (home slot).

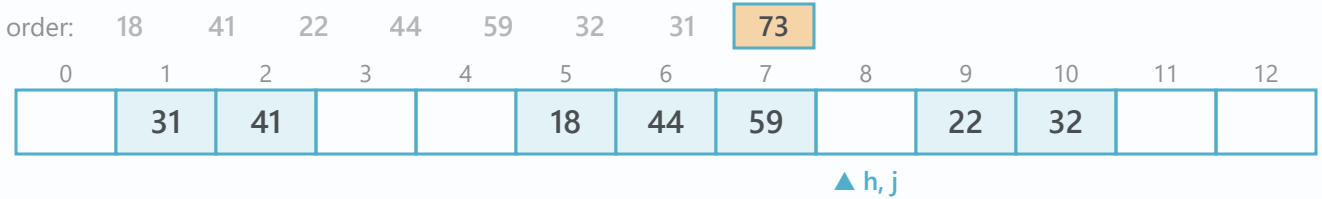
Step 7 / 9

```
1 // put(k): quadratic probing, N = 13
▶ int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;        // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 73` `h = 8`

Quadratic Probing (cont.): 31 and 73



i = 0: j = (8 + 0^2) mod 13 = **8** → **empty**.

Step 8 / 9

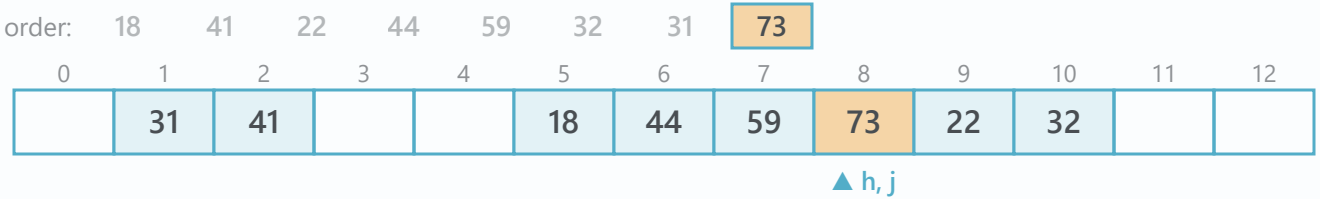
```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
▶   int j = (h + i * i) % 13;       // h, h+1, h+4…
▶   if (table[j] == null) {        // empty?
6     table[j] = k;
7     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

k = **73** h = **8** i = **0** j = **8**

table[j] = **null**

Quadratic Probing (cont.): 31 and 73



`table[8] = 73` , `break` (1 probe).

Step 9 / 9

```
1 // put(k): quadratic probing, N = 13
2 int h = k % 13;                  // home slot
3 for (int i = 0; i < 13; i++) {
4   int j = (h + i * i) % 13;       // h, h+1, h+4…
5   if (table[j] == null) {        // empty?
▶     table[j] = k;
▶     break;
8   }                              // occupied: i++
9 }
```

VARIABLES

`k = 73` `h = 8` `i = 0` `j = 8`

`table[j] = 73`

OPEN ADDRESSING

Double Hashing

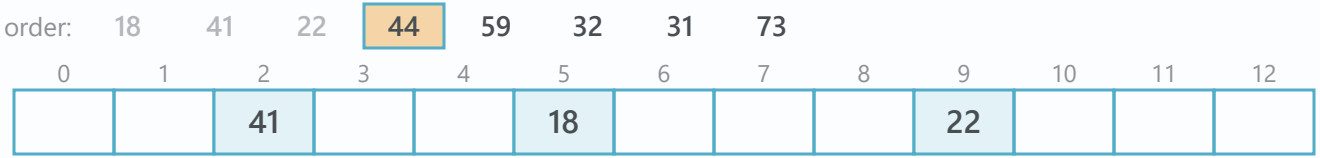
`h(k, i) = (h(k) + i · h₂(k)) mod 13`, `h₂(k) = 7 − (k mod 7)` → the step is 1…7, never 0.

k	18	41	22	44	59	32	31	73
h(k) = k mod 13	5	2	9	5	7	6	5	8
$h_2(k)$ (on collision)	—	—	—	7−2 = 5	—	—	7−3 = 4	—
Slots probed	5	2	9	5, 10	7	6	5, 9, 0	8



44: 5 occupied → 5+5 = **10**. 31: 5 occupied → 5+4 = 9 occupied → 5+8 = 13 mod 13 = **0**. Keys with the same home slot advance by **different steps**, so clustering drops.

Code Trace: Double Hashing



18 → 5, 41 → 2, 22 → 9: all three find an empty slot on the first try. Next up: 44.

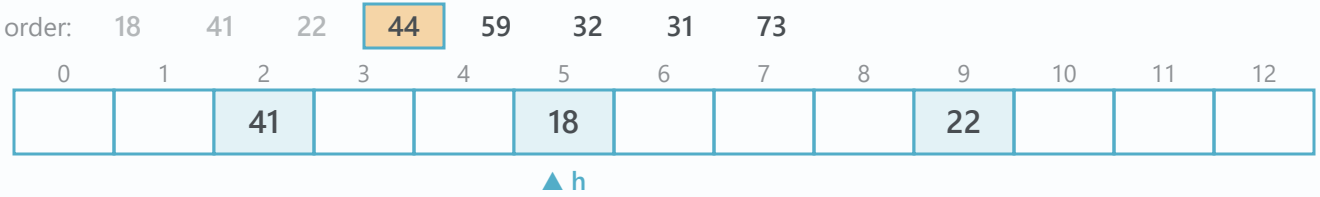
Step 1 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`N = 13`

Code Trace: Double Hashing



h = 44 mod 13 = 5 (home slot), **d = 7 − 2 = 5** (step size, never 0).

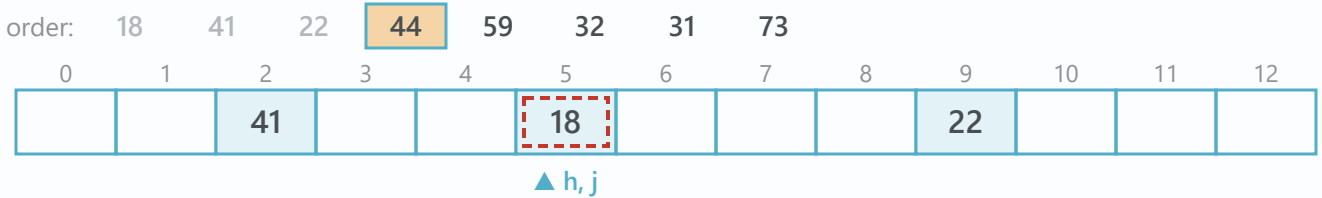
Step 2 / 19

```
1  // put(k): double hashing, N = 13
▶ int h = k % 13;
▶ int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {      // empty?
7      table[j] = k;
8      break;
9    }                            // occupied: i++
10 }
```

VARIABLES

k = 44 | h = 5 | d = 5

Code Trace: Double Hashing



i = 0: j = (5 + 0·5) mod 13 = **5** → occupied (18); next i.

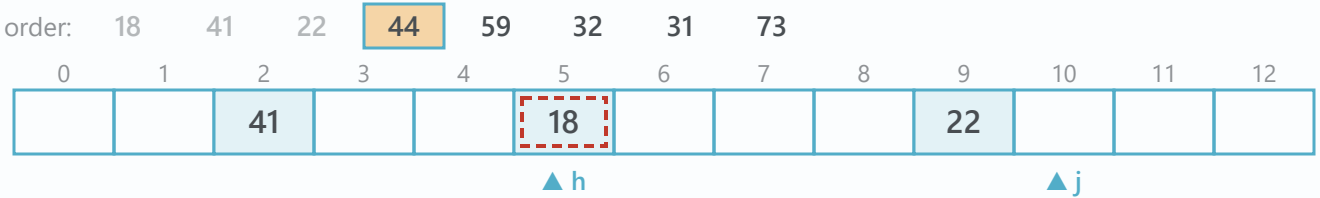
Step 3 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 44` `h = 5` `d = 5` `i = 0` `j = 5` `table[j] = 18`

Code Trace: Double Hashing



i = 1: j = (5 + 1·5) mod 13 = **10** → **empty**.

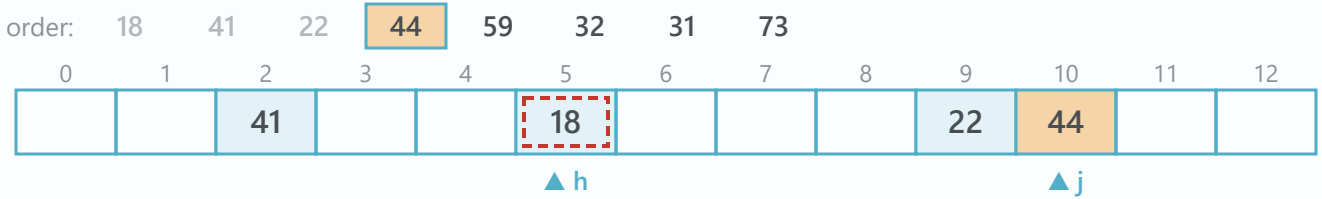
Step 4 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 44` `h = 5` `d = 5` `i = 1` `j = 10` `table[j] = null`

Code Trace: Double Hashing



`table[10] = 44`, `break` (2 probes).

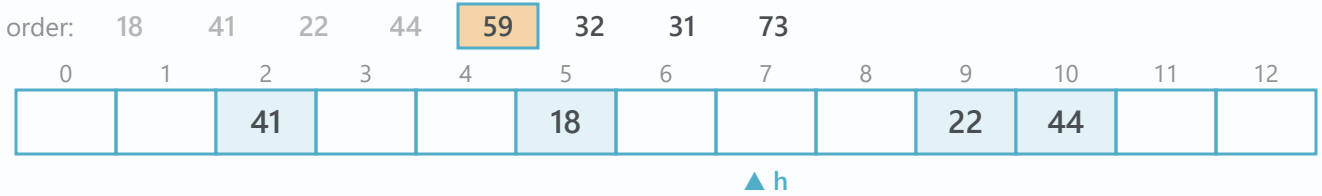
Step 5 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {       // empty?
▶      table[j] = k;
▶      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 44` `h = 5` `d = 5` `i = 1` `j = 10` `table[j] = 44`

Code Trace: Double Hashing



h = 59 mod 13 = 7 (home slot), **d = 7 − 3 = 4** (step size, never 0).

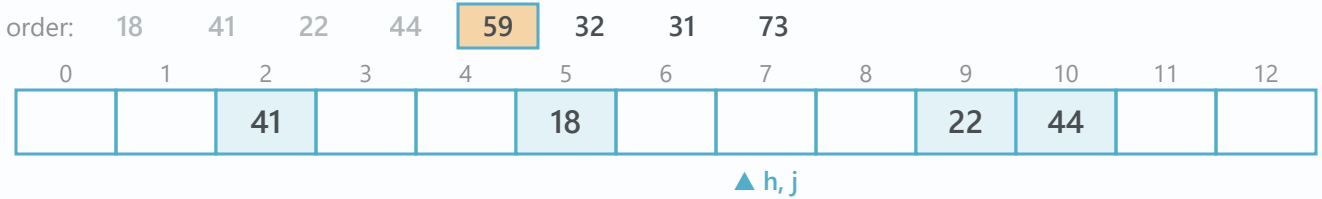
Step 6 / 19

```
1  // put(k): double hashing, N = 13
▶ int h = k % 13;
▶ int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {      // empty?
7      table[j] = k;
8      break;
9    }                            // occupied: i++
10 }
```

VARIABLES

k = 59 | h = 7 | d = 4

Code Trace: Double Hashing



i = 0: j = (7 + 0·4) mod 13 = **7** → **empty**.

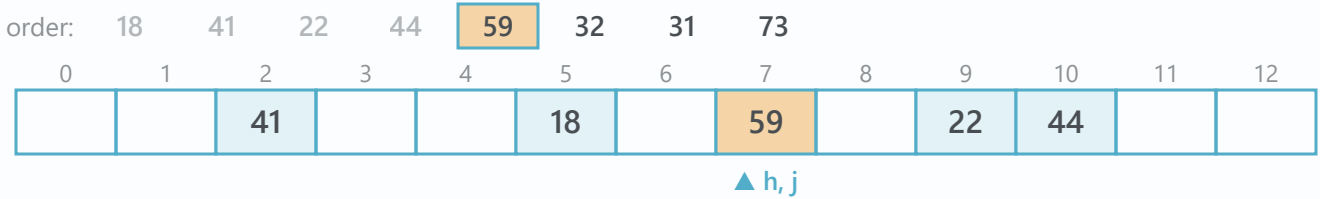
Step 7 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 59` `h = 7` `d = 4` `i = 0` `j = 7` `table[j] = null`

Code Trace: Double Hashing



`table[7] = 59` , `break` (1 probe).

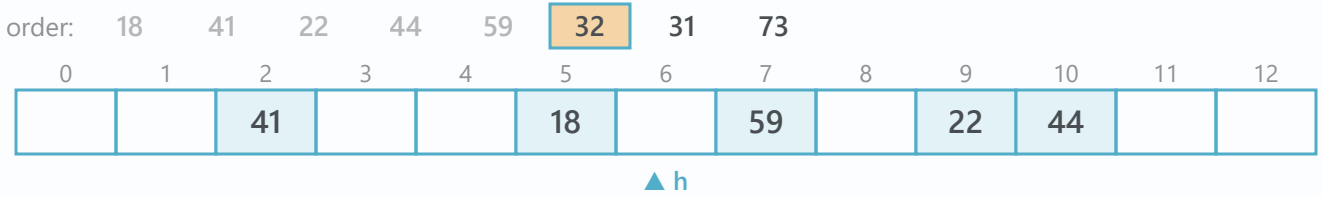
Step 8 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {       // empty?
▶      table[j] = k;
▶      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 59` `h = 7` `d = 4` `i = 0` `j = 7` `table[j] = 59`

Code Trace: Double Hashing



h = 32 mod 13 = 6 (home slot), **d = 7 − 4 = 3** (step size, never 0).

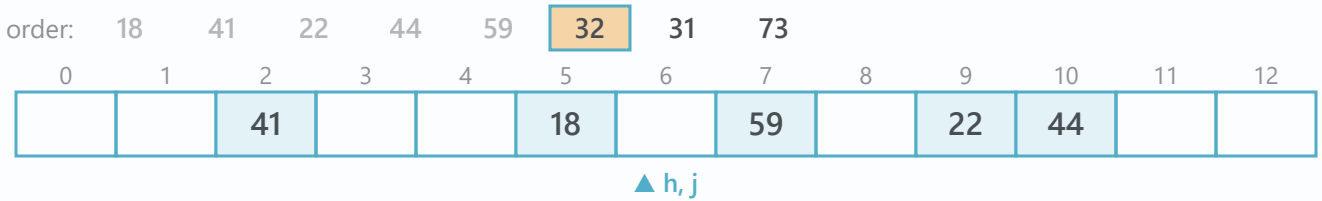
Step 9 / 19

```
1  // put(k): double hashing, N = 13
▶ int h = k % 13;
▶ int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {      // empty?
7      table[j] = k;
8      break;
9    }                            // occupied: i++
10 }
```

VARIABLES

k = 32 h = 6 d = 3

Code Trace: Double Hashing



i = 0: j = (6 + 0·3) mod 13 = **6** → **empty**.

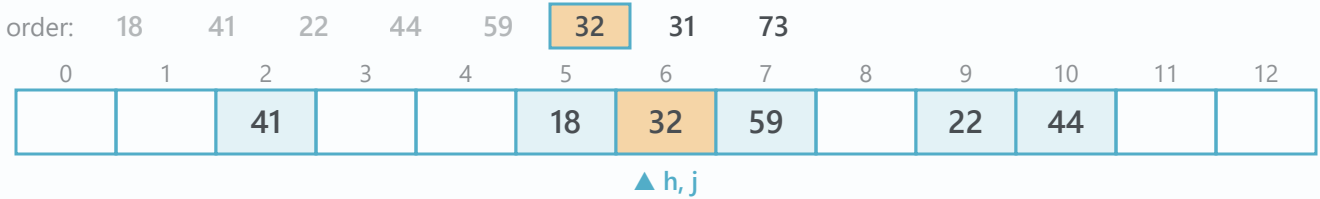
Step 10 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 32` `h = 6` `d = 3` `i = 0` `j = 6` `table[j] = null`

Code Trace: Double Hashing



`table[6] = 32` , `break` (1 probe).

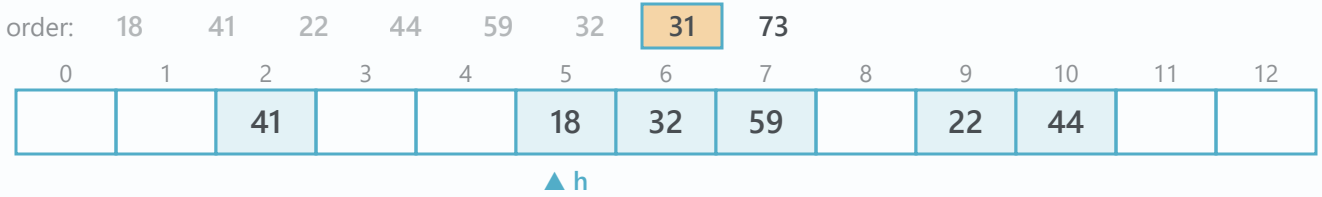
Step 11 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {       // empty?
▶      table[j] = k;
▶      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 32` `h = 6` `d = 3` `i = 0` `j = 6` `table[j] = 32`

Code Trace: Double Hashing



h = 31 mod 13 = 5 (home slot), **d = 7 − 3 = 4** (step size, never 0).

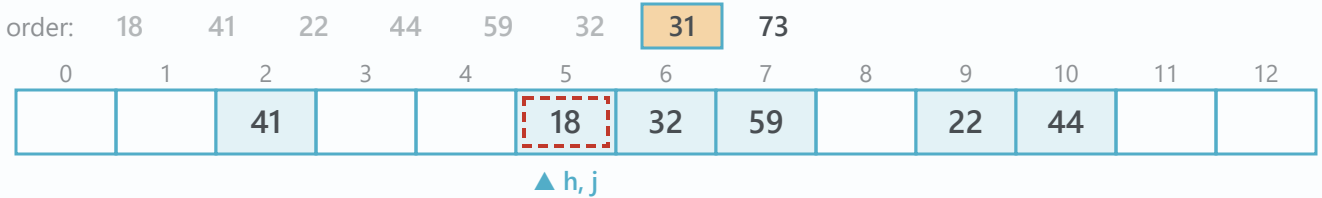
Step 12 / 19

```
1  // put(k): double hashing, N = 13
▶ int h = k % 13;
▶ int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {      // empty?
7      table[j] = k;
8      break;
9    }                            // occupied: i++
10 }
```

VARIABLES

k = 31 h = 5 d = 4

Code Trace: Double Hashing



i = 0: j = (5 + 0·4) mod 13 = **5** → occupied (18); next i.

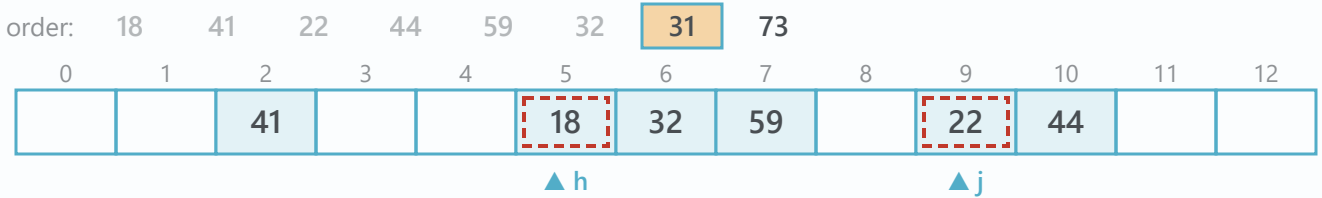
Step 13 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 31` `h = 5` `d = 4` `i = 0` `j = 5` `table[j] = 18`

Code Trace: Double Hashing



i = 1: j = (5 + 1·4) mod 13 = **9** → occupied (22); next i.

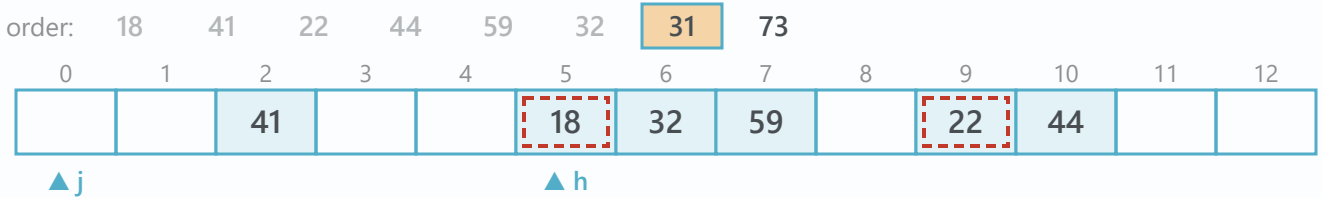
Step 14 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 31` `h = 5` `d = 4` `i = 1` `j = 9` `table[j] = 22`

Code Trace: Double Hashing



i = 2: j = (5 + 2·4) mod 13 = **0** → **empty**.

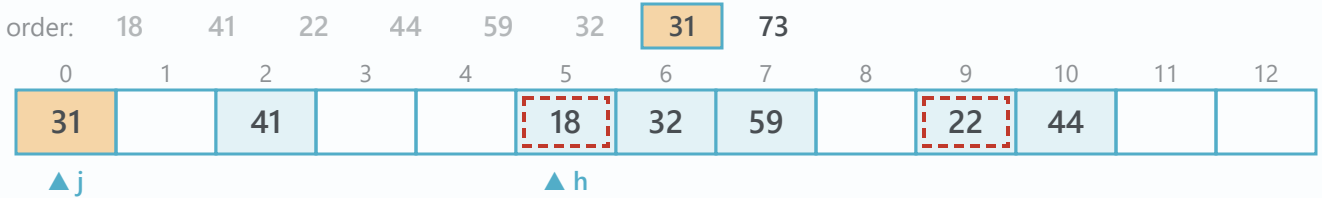
Step 15 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 31` `h = 5` `d = 4` `i = 2` `j = 0` `table[j] = null`

Code Trace: Double Hashing



`table[0] = 31`, `break` (3 probes).

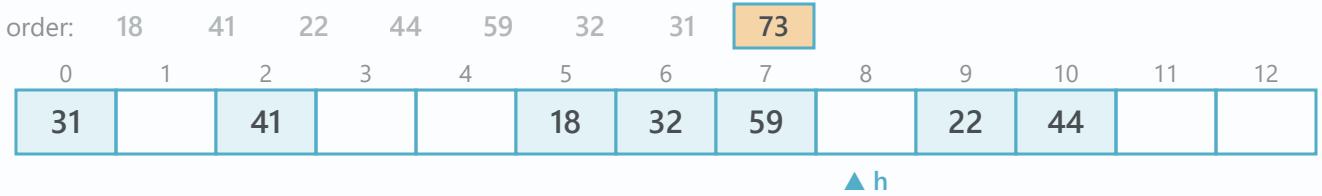
Step 16 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {       // empty?
▶      table[j] = k;
▶      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 31` `h = 5` `d = 4` `i = 2` `j = 0` `table[j] = 31`

Code Trace: Double Hashing



h = 73 mod 13 = 8 (home slot), **d = 7 − 3 = 4** (step size, never 0).

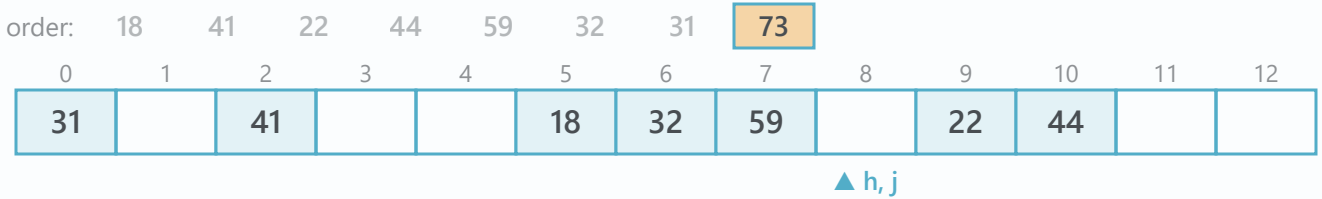
Step 17 / 19

```
1  // put(k): double hashing, N = 13
▶ int h = k % 13;
▶ int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {      // empty?
7      table[j] = k;
8      break;
9    }                            // occupied: i++
10 }
```

VARIABLES

k = 73 h = 8 d = 4

Code Trace: Double Hashing



i = 0: j = (8 + 0·4) mod 13 = **8** → **empty**.

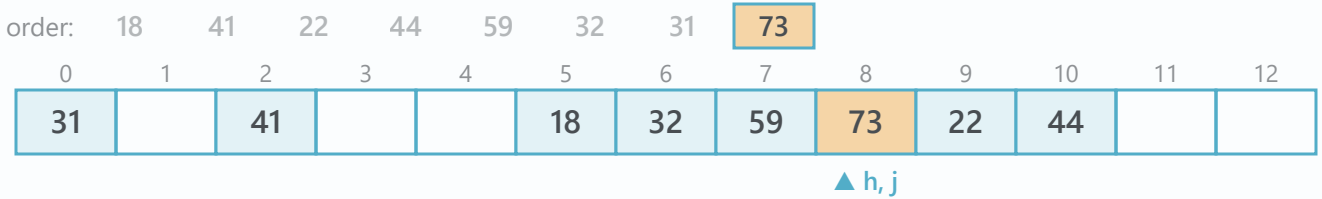
Step 18 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
▶    int j = (h + i * d) % 13;
▶    if (table[j] == null) {       // empty?
7      table[j] = k;
8      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 73` `h = 8` `d = 4` `i = 0` `j = 8` `table[j] = null`

Code Trace: Double Hashing



`table[8] = 73` , `break` (1 probe).

Step 19 / 19

```
1  // put(k): double hashing, N = 13
2  int h = k % 13;
3  int d = 7 - k % 7;              // step: 1…7
4  for (int i = 0; i < 13; i++) {
5    int j = (h + i * d) % 13;
6    if (table[j] == null) {       // empty?
▶      table[j] = k;
▶      break;
9    }                             // occupied: i++
10 }
```

VARIABLES

`k = 73` `h = 8` `d = 4` `i = 0` `j = 8` `table[j] = 73`

OPEN ADDRESSING

Deleting Under Open Addressing: The "Deleted" Marker

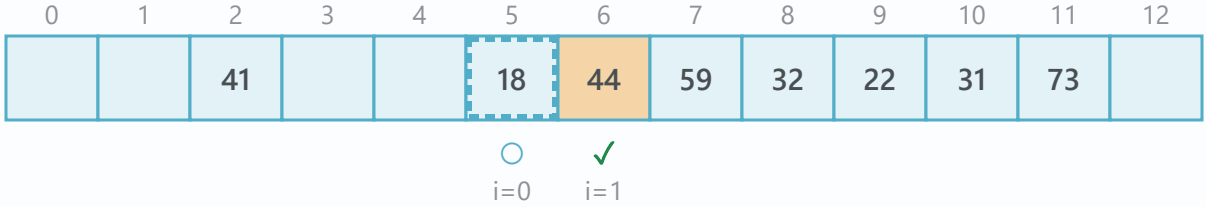


The table after linear probing. We want to delete 44.

Step 1 / 6

✗ = deleted (DEFUNCT). A search skips this slot as if occupied; an insert may reuse it as if empty. ○ = probed slot, ✓ = found.

Deleting Under Open Addressing: The "Deleted" Marker

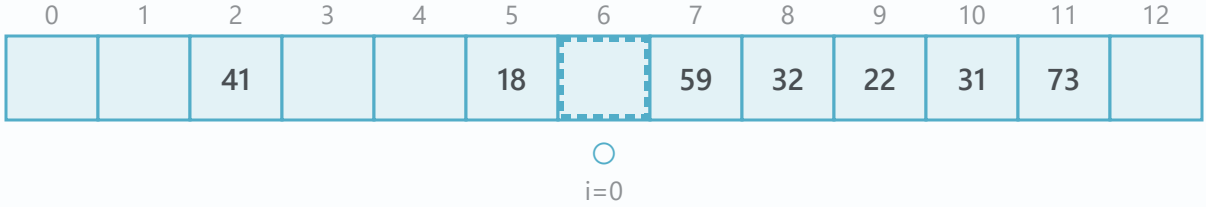


remove(44): h(44) = **5** → 18 ≠ 44 → slot 6: found 44. What if we just set the slot to `null`?

Step 2 / 6

X = deleted (DEFUNCT). A search skips this slot as if occupied; an insert may reuse it as if empty. ○ = probed slot, ✓ = found.

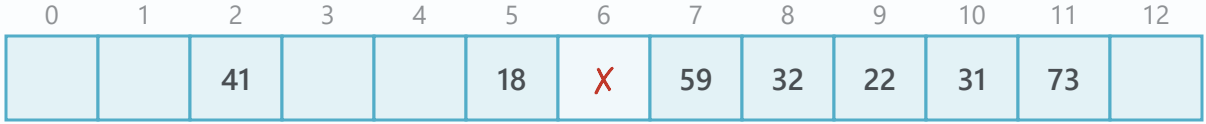
Deleting Under Open Addressing: The "Deleted" Marker



Wrong: slot 6 is emptied. get(32): h(32) = **6** → empty slot → reports "absent". But 32 is sitting in slot 8!

X = deleted (DEFUNCT). A search skips this slot as if occupied; an insert may reuse it as if empty. ○ = probed slot, ✓ = found.

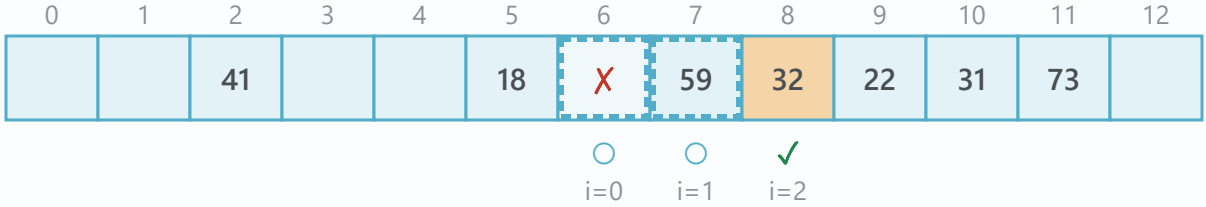
Deleting Under Open Addressing: The "Deleted" Marker



Right: mark the slot with a special **deleted** marker (DEFUNCT, a tombstone ╳).

╳ = deleted (DEFUNCT). A search skips this slot as if occupied; an insert may reuse it as if empty. ○ = probed slot, ✓ = found.

Deleting Under Open Addressing: The "Deleted" Marker

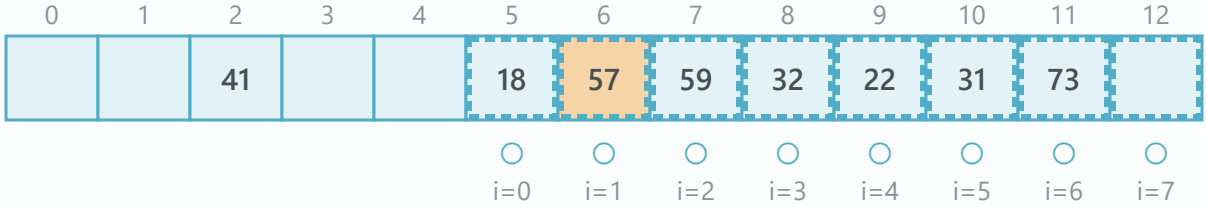


get(32): h = **6** → X deleted, keep searching → 7: 59 ≠ 32 → 8: **found 32**.

Step 5 / 6

X = deleted (DEFUNCT). A search skips this slot as if occupied; an insert may reuse it as if empty. ○ = probed slot, ✓ = found.

Deleting Under Open Addressing: The "Deleted" Marker



put(57): h = **5** → 5, 6(X), 7 … 11 occupied, 12 empty → 57 is absent. It goes into the first available slot, **6**.

Step 6 / 6

X = deleted (DEFUNCT). A search skips this slot as if occupied; an insert may reuse it as if empty. ○ = probed slot, ✓ = found.

Finding a Slot: The Step Search and Insert Share

```
private int findSlot(int k) {       // slot index
  int h = Math.floorMod(k, N), avail = -1;
  for (int i = 0; i < N; i++) {
    int j = (h + i) % N;           // linear
    if (table[j] == null ||
        table[j] == DEFUNCT) {
      if (avail == -1) avail = j;
      if (table[j] == null) break;
    } else if (table[j].key == k)
      return j;                    // found
  }
  return -(avail + 1);         // absent: encoded slot
}
```

get(k)

`j = findSlot(k)`; if j ≥ 0, return `table[j].value`, otherwise `null`.

put(k, v)

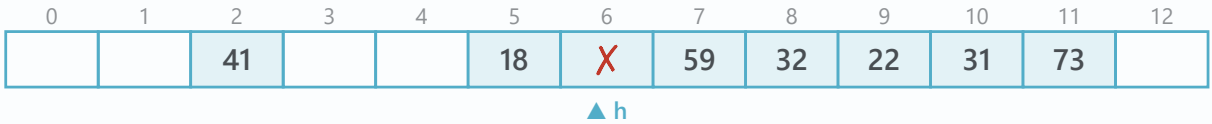
If j ≥ 0, update; otherwise store the new pair in slot `table[-(j+1)]`.

Other probes

Quadratic: `(h + i*i) % N`
Double hashing: `(h + i*h2) % N`

The table never fills up completely (λ is kept below 1 and the table grows when needed), so `avail` is always found.

Code Trace: `findSlot(32)`, with 44 deleted (X)



`findSlot(32)`: h = 32 mod 13 = **6**; no available slot yet (−1).

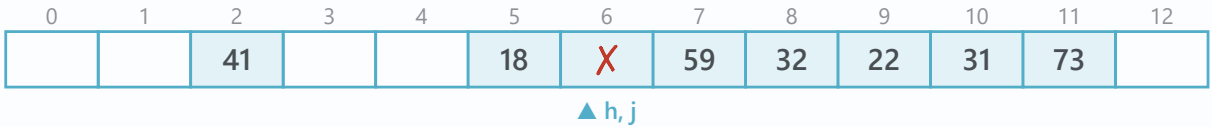
Step 1 / 7

```
1  private int findSlot(int k) {        // slot index
▶    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                      // found
11   }
12   return -(avail + 1);          // absent: encoded slot
13 }
```

VARIABLES

`k = 32` `h = 6` `avail = -1`

Code Trace: `findSlot(32)`, with 44 deleted (X)



i = 0: j = **6** → the slot is **DEFUNCT** (*X*).

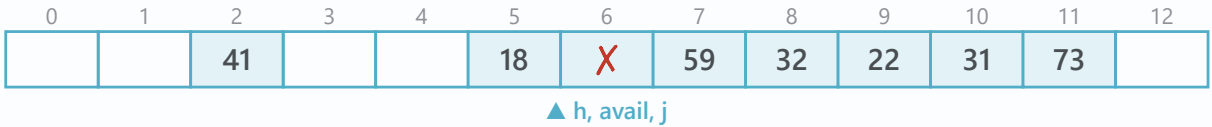
Step 2 / 7

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
▶          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 32` `h = 6` `avail = -1` `i = 0` `j = 6` `table[j] = DEFUNCT`

Code Trace: `findSlot(32)`, with 44 deleted (✗)



First empty or deleted slot: **avail = 6** (an insert would go here).

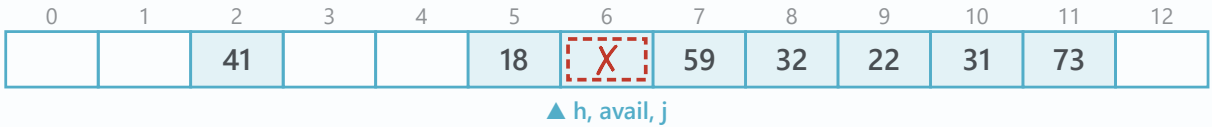
Step 3 / 7

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
▶        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 32` `h = 6` `avail = 6` `i = 0` `j = 6` `table[j] = DEFUNCT`

Code Trace: `findSlot(32)`, with 44 deleted (X)



Deleted ≠ empty: **keep searching** (32 may lie beyond).

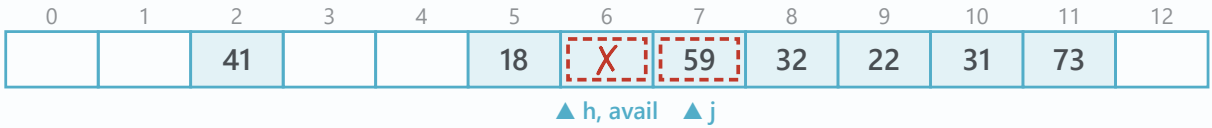
Step 4 / 7

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
▶        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 32` `h = 6` `avail = 6` `i = 0` `j = 6` `table[j] = DEFUNCT`

Code Trace: `findSlot(32)`, with 44 deleted (X)



i = 1: j = **7** → occupied (59 ≠ 32); next slot.

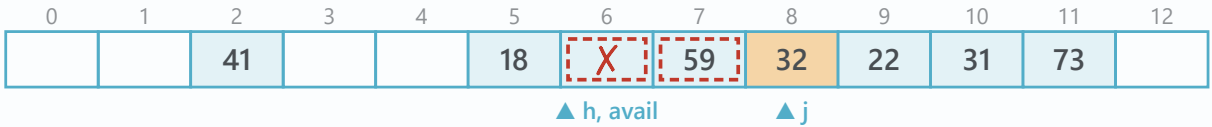
Step 5 / 7

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;             // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                      // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 32` `h = 6` `avail = 6` `i = 1` `j = 7` `table[j] = 59`

Code Trace: `findSlot(32)`, with 44 deleted (X)



i = 2: j = **8** → occupied, and table[8].key = 32 == 32.

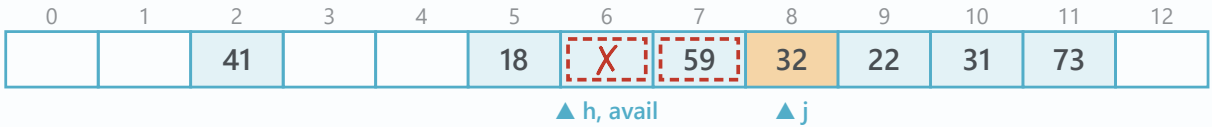
Step 6 / 7

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;             // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                      // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 32` `h = 6` `avail = 6` `i = 2` `j = 8` `table[j] = 32`

Code Trace: `findSlot(32)`, with 44 deleted (X)



`return 8` (≥ 0 means found). `get` returns table[8].value; the X did not break the probe sequence.

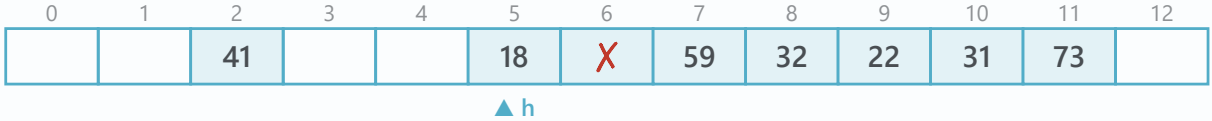
Step 7 / 7

```
1  private int findSlot(int k) {       // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;            // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
▶        return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 32` `h = 6` `avail = 6` `i = 2` `j = 8` `table[j] = 32`

Code Trace: `findSlot(57)` and Insertion



`findSlot(57)` : h = 57 mod 13 = **5**; no available slot yet (−1).

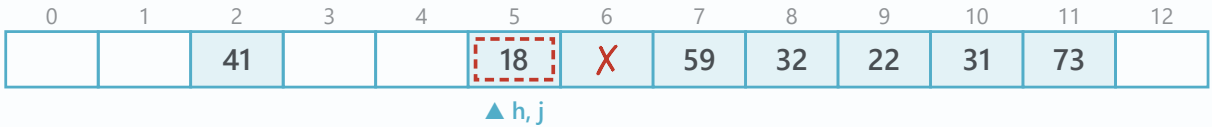
Step 1 / 15

```
1  private int findSlot(int k) {       // slot index
▶    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;           // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                   // found
11   }
12   return -(avail + 1);        // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = −1`

Code Trace: `findSlot(57)` and Insertion



i = 0: j = **5** → occupied (18 ≠ 57); next slot.

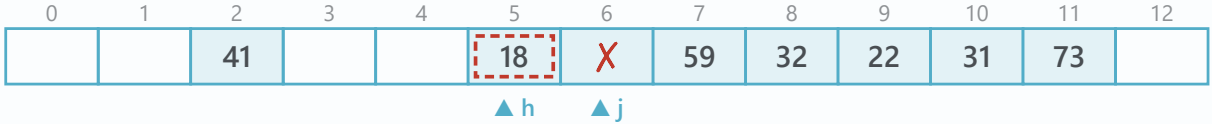
Step 2 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = -1` `i = 0` `j = 5` `table[j] = 18`

Code Trace: `findSlot(57)` and Insertion



i = 1: j = **6** → the slot is **DEFUNCT** (*X*).

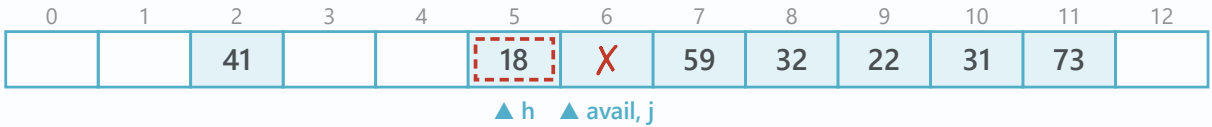
Step 3 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
▶          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = −1` `i = 1` `j = 6` `table[j] = DEFUNCT`

Code Trace: `findSlot(57)` and Insertion



First empty or deleted slot: **avail = 6** (an insert would go here).

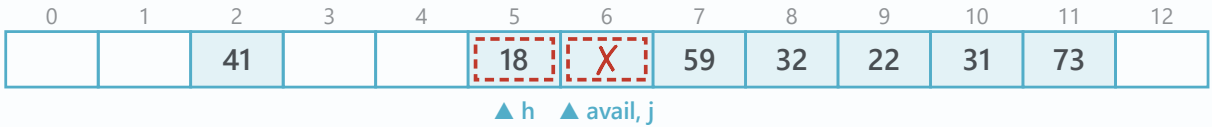
Step 4 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
▶        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                      // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 1` `j = 6` `table[j] = DEFUNCT`

Code Trace: `findSlot(57)` and Insertion



Deleted ≠ empty: **keep searching** (57 may lie beyond).

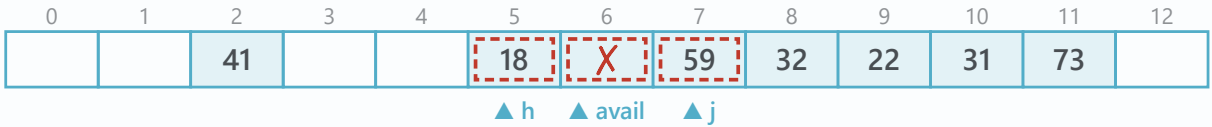
Step 5 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
▶        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                      // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 1` `j = 6` `table[j] = DEFUNCT`

Code Trace: `findSlot(57)` and Insertion



i = 2: j = **7** → occupied (59 ≠ 57); next slot.

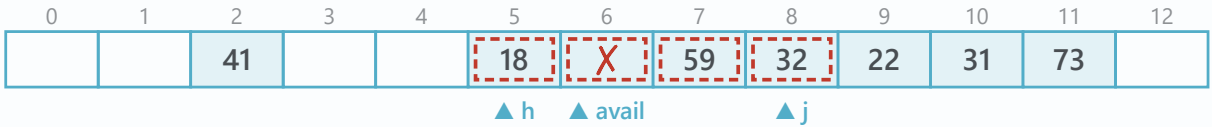
Step 6 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 2` `j = 7` `table[j] = 59`

Code Trace: `findSlot(57)` and Insertion



i = 3: j = **8** → occupied (32 ≠ 57); next slot.

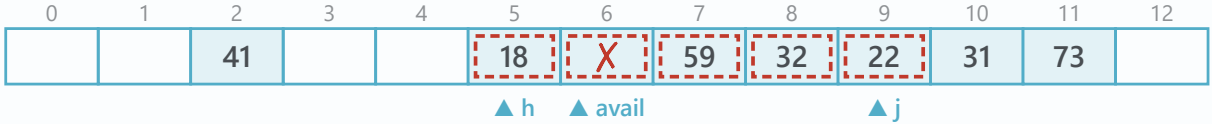
Step 7 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 3` `j = 8` `table[j] = 32`

Code Trace: `findSlot(57)` and Insertion



i = 4: j = **9** → occupied (22 ≠ 57); next slot.

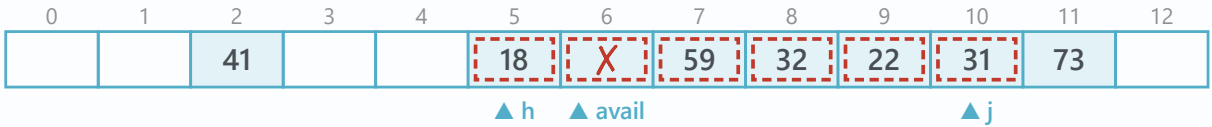
Step 8 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 4` `j = 9` `table[j] = 22`

Code Trace: `findSlot(57)` and Insertion



i = 5: j = **10** → occupied (31 ≠ 57); next slot.

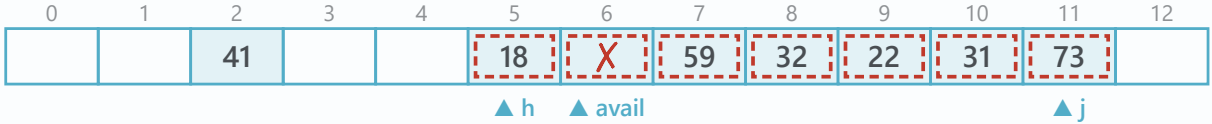
Step 9 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 5` `j = 10` `table[j] = 31`

Code Trace: `findSlot(57)` and Insertion



i = 6: j = **11** → occupied (73 ≠ 57); next slot.

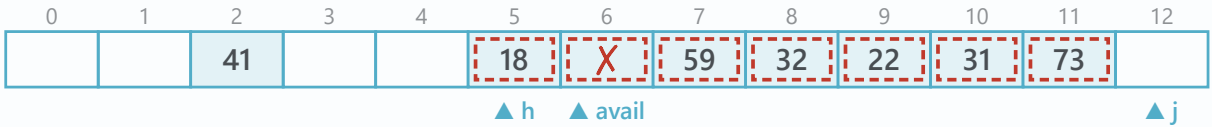
Step 10 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
▶      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 6` `j = 11` `table[j] = 73`

Code Trace: `findSlot(57)` and Insertion



i = 7: j = **12** → the slot is empty.

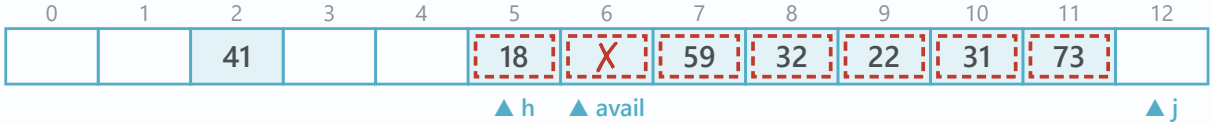
Step 11 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
▶      int j = (h + i) % N;            // linear
▶      if (table[j] == null ||
▶          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                     // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 7` `j = 12` `table[j] = null`

Code Trace: `findSlot(57)` and Insertion



avail is already 6 and stays (we keep the first available slot).

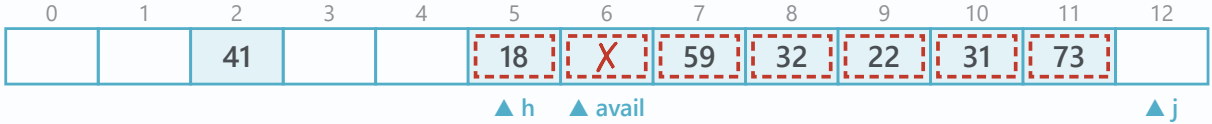
Step 12 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
▶        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                      // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 7` `j = 12` `table[j] = null`

Code Trace: `findSlot(57)` and Insertion



A truly **empty** slot → 57 cannot lie beyond it: `break`.

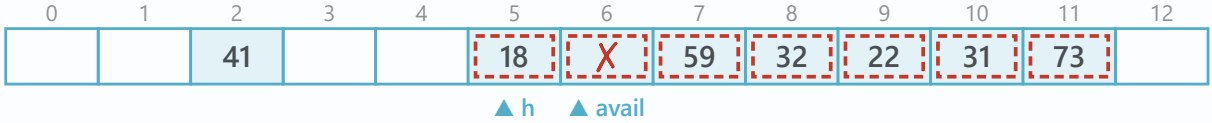
Step 13 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
▶        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                      // found
11   }
12   return -(avail + 1);         // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6` `i = 7` `j = 12` `table[j] = null`

Code Trace: `findSlot(57)` and Insertion



57 is absent: `return -(6 + 1) = -7` (negative means "absent" and encodes where to insert).

Step 14 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                      // found
11   }
▶    return -(avail + 1);          // absent: encoded slot
13 }
```

VARIABLES

`k = 57` `h = 5` `avail = 6`

Code Trace: `findSlot(57)` and Insertion



`put` : j = -7 < 0 → `table[-(j + 1)] = table[6] = 57`. The deleted slot is reused.

Step 15 / 15

```
1  private int findSlot(int k) {        // slot index
2    int h = Math.floorMod(k, N), avail = -1;
3    for (int i = 0; i < N; i++) {
4      int j = (h + i) % N;             // linear
5      if (table[j] == null ||
6          table[j] == DEFUNCT) {
7        if (avail == -1) avail = j;
8        if (table[j] == null) break;
9      } else if (table[j].key == k)
10       return j;                      // found
11   }
▶    return -(avail + 1);         // absent: encoded slot
13 }
```

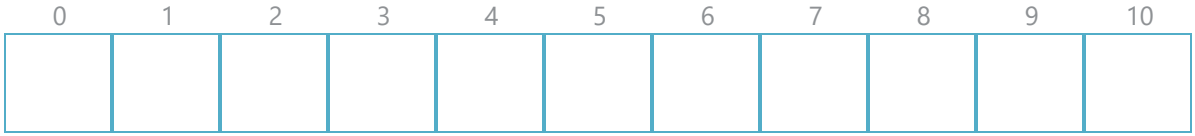
VARIABLES

`k = 57` `h = 5` `avail = 6`

Try It: Open Addressing (N = 11)

Method linear: h + i Key 22

insert search delete fill sample reset

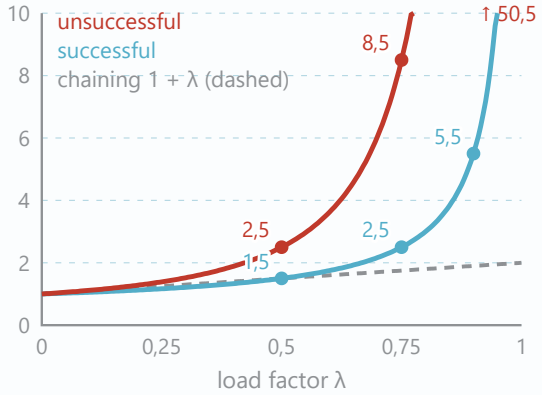


Enter a key and press **insert**, or start with **fill sample**.

h(k) = k mod 11. Dashed outline: occupied slot probed · orange: result slot · X: deleted (DEFUNCT). Enter = insert.

Load Factor

- **λ = n / N** (n: number of pairs = size, N: number of buckets).
- With chaining, λ is the average chain length; λ > 1 is allowed, but operations slow down.
- With open addressing, λ ≤ 1 is mandatory, and the number of probes explodes as λ → 1.
- Practical thresholds: λ ≈ 0.75 for chaining (Java's `HashMap`), λ ≤ 0.5 for open addressing.
- Once the threshold is crossed, the table roughly doubles.



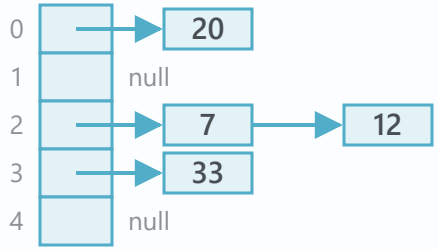
Expected probes under linear probing (Knuth):

successful $\tfrac{1}{2}(1 + 1/(1-\lambda))$ ·

unsuccessful $\tfrac{1}{2}(1 + 1/(1-\lambda)^2)$

With chaining the expected cost is **O(1 + λ)**: keep λ constant (grow the table along with n) and insert, search and delete are **O(1) on average**.

Resizing and Rehashing



N = 5, h(k) = k mod 5

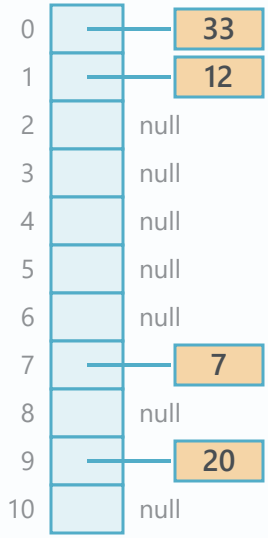
12 → 2, 7 → 2, 20 → 0, 33 → 3

λ = 4 / 5 = 0.80 > 0.75

→ grow the table

12, 7, 20 and 33 are inserted into a table with N = 5 buckets. λ = 0.80 exceeds the 0.75 threshold → time to resize.

Resizing and Rehashing



New N = 11 (first prime above 2·5)

20 mod 11 = 9

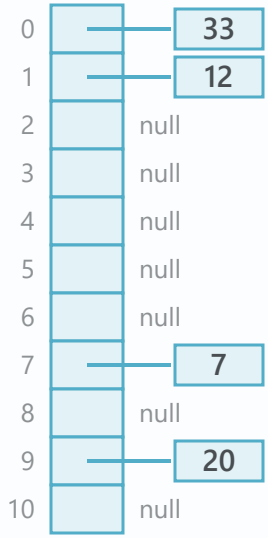
7 mod 11 = 7

12 mod 11 = 1

33 mod 11 = 0

A new array is allocated and **every key is rehashed**: indices change, so the old positions are meaningless.

Resizing and Rehashing



$\lambda = 4 / 11 \approx 0{,}36$

no collisions left

cost: O(n), once

amortized: O(1) per insert

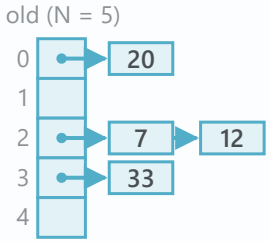
A single resize costs O(n), which is expensive; but because the capacity roughly doubles each time, the cost per insertion is **amortized O(1)**.

Code Trace: `resize()`

```
1  private void resize() {
▶    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {   // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)`



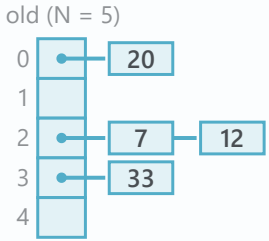
Called because λ = 4/5 = 0.8 > 0.75. `old` holds the old array.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
▶    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {   // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11`



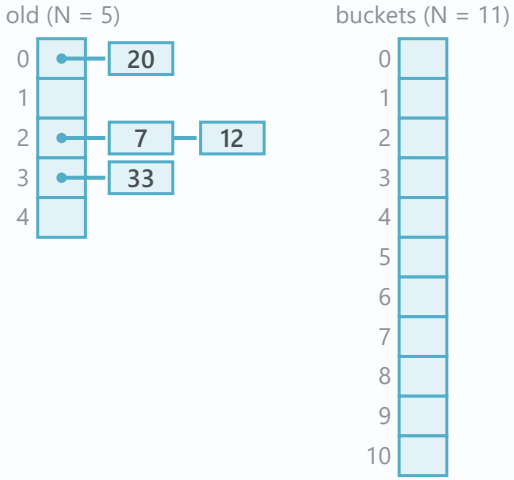
newN = 2·5 + 1 = 11: about twice as large, and odd (prime, in this case).

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
▶    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11`



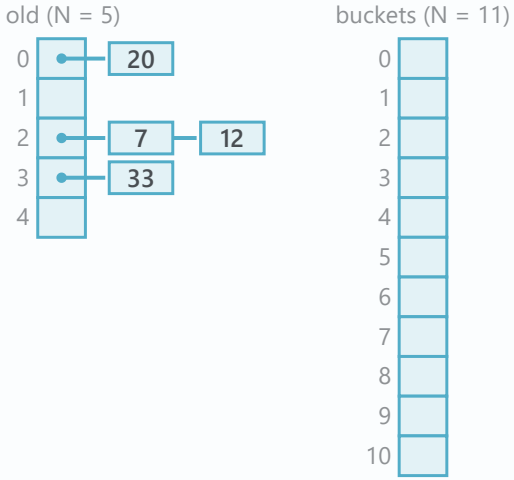
`buckets` now points to the new array of 11 empty buckets.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
▶    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 0`



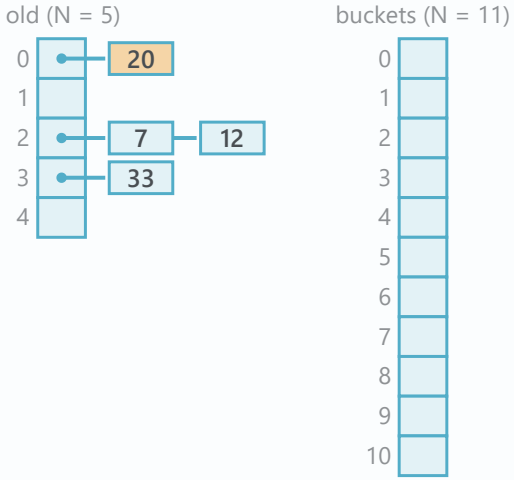
size = 0: `put` will count every pair again.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {   // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 0` `head = node(20)`



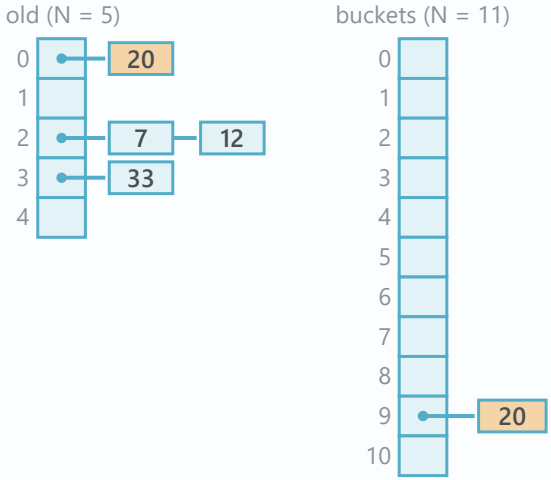
Old bucket 0: `head` = node 20.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
▶        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 1` `head = node(20)`



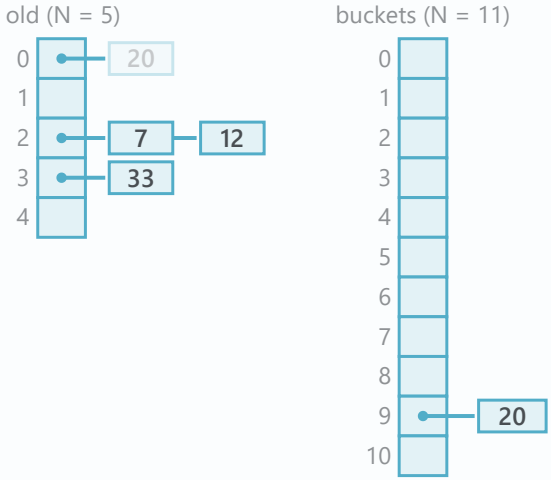
`put(20)`: it was 20 mod 5 = 0; now **20 mod 11 = 9** → new bucket 9.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
8        put(head.key, head.value);
▶        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 1` `head = null`



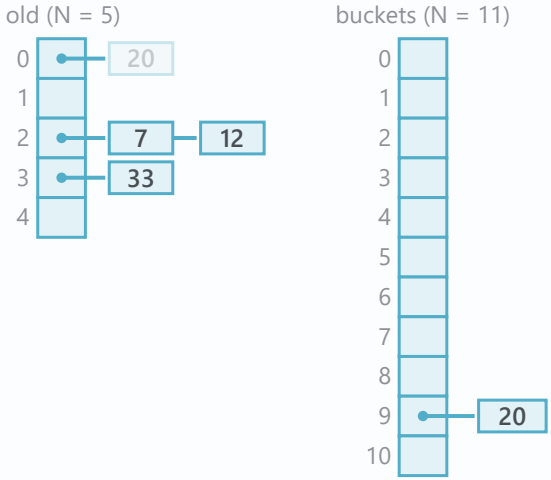
`head = null` : this chain is done.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
▶    for (HashNode head : old) {   // each bucket
▶      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 1` `head = null`



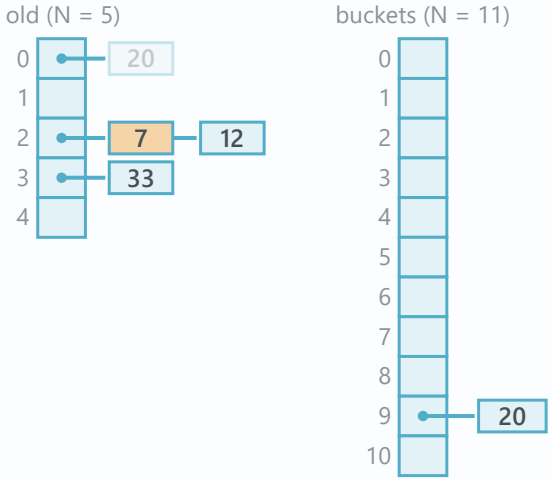
Old bucket 1 is empty: `head = null`, so the while loop is skipped.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
▶  for (HashNode head : old) {   // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 1` `head = node(7)`



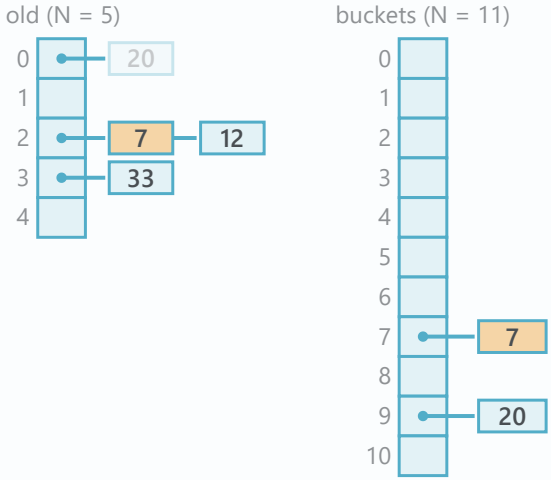
Old bucket 2: `head` = node 7.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
▶        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 2` `head = node(7)`



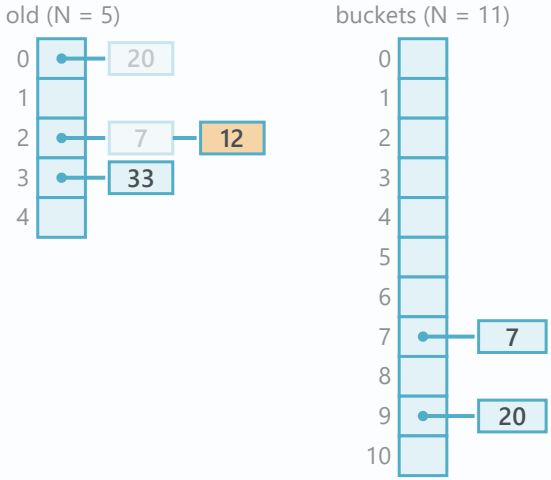
`put(7)`: it was 7 mod 5 = 2; now **7 mod 11 = 7** → new bucket 7.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
8        put(head.key, head.value);
▶        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 2` `head = node(12)`



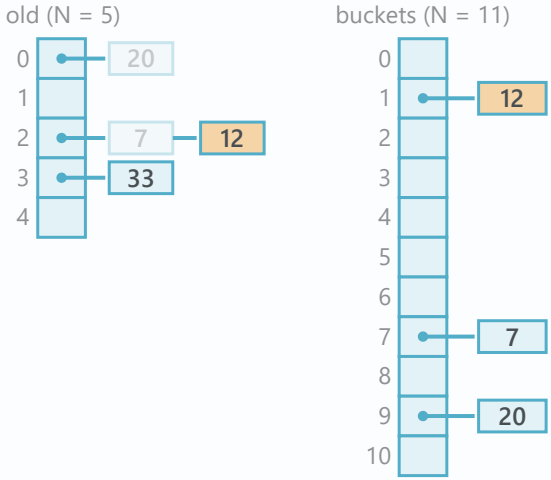
`head = head.next` → node 12 (same old chain).

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
▶        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 3` `head = node(12)`



`put(12)`: it was 12 mod 5 = 2; now **12 mod 11 = 1** → new bucket 1.

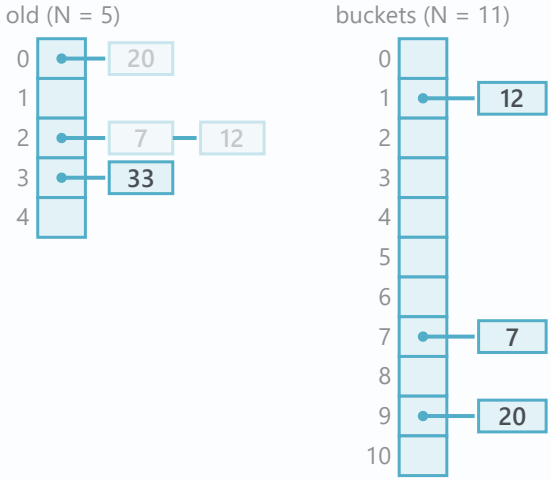
Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
8        put(head.key, head.value);
▶        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 3`

`head = null`



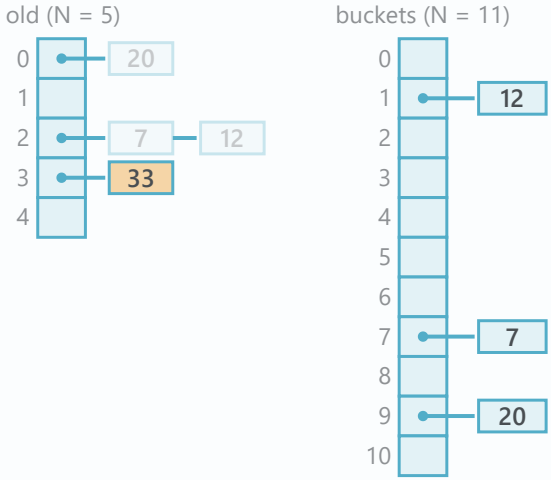
`head = null` : this chain is done.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {   // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 3` `head = node(33)`



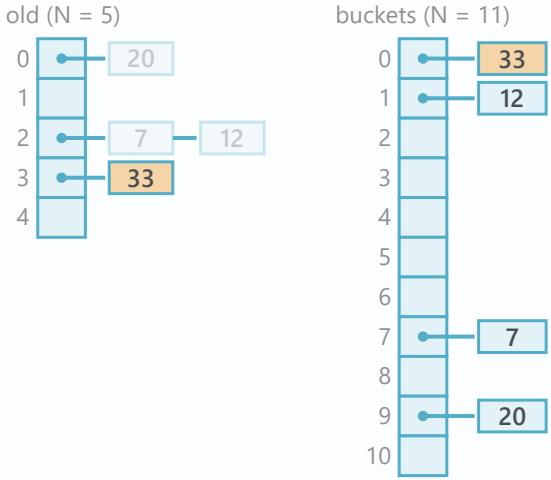
Old bucket 3: `head` = node 33.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
▶        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 4` `head = node(33)`



`put(33)`: it was 33 mod 5 = 3; now **33 mod 11 = 0** → new bucket 0.

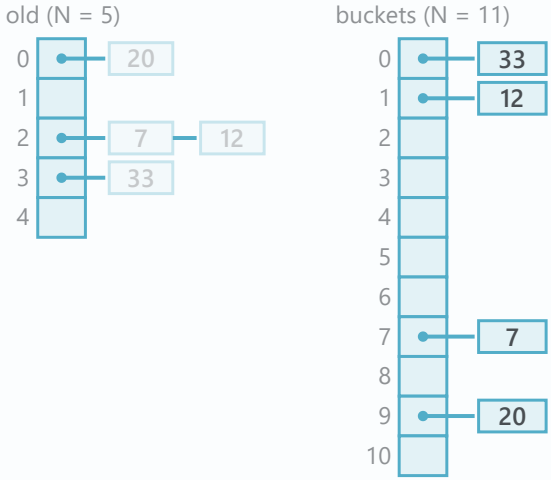
Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
8        put(head.key, head.value);
▶        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 4`

`head = null`



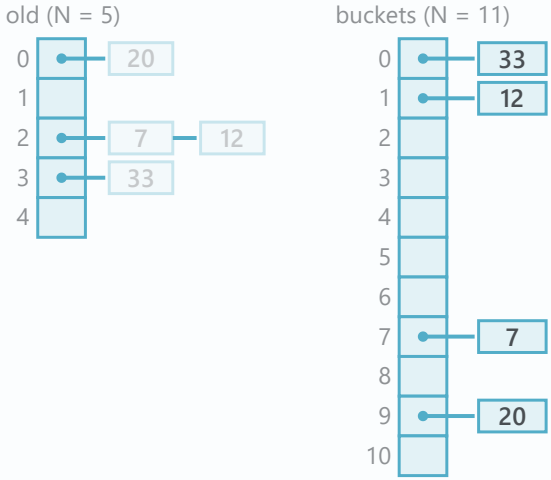
`head = null` : this chain is done.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
▶    for (HashNode head : old) {   // each bucket
▶      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
12 }
```

VARIABLES

`old = buckets(5)` `newN = 11` `size = 4` `head = null`



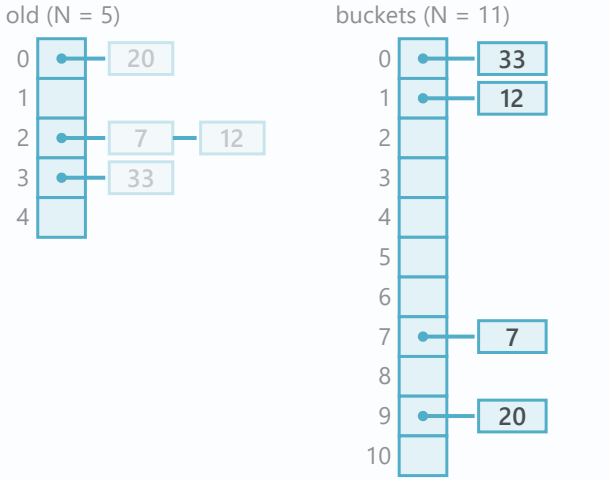
Old bucket 4 is empty: `head = null`, so the while loop is skipped.

Code Trace: `resize()`

```
1  private void resize() {
2    HashNode[] old = buckets;
3    int newN = 2 * old.length + 1;
4    buckets = new HashNode[newN];
5    size = 0;  numBuckets = newN;
6    for (HashNode head : old) {    // each bucket
7      while (head != null) {
8        put(head.key, head.value);
9        head = head.next;
10     }
11   }
▶ }
```

VARIABLES

`newN = 11` `size = 4`



Done: 7 and 12 are now in different buckets. λ = 4/11 ≈ 0.36. Every pair was moved, so this is **O(n)**.

Comparing Complexities

Map implementation	get	put	remove
Unsorted array / list	O(n)	O(n)*	O(n)
Sorted array (binary search)	O(log n)	O(n)	O(n)
Hash table: average	**O(1)**	**O(1)**	**O(1)**
Hash table: worst case	O(n)	O(n)	O(n)

* The whole list must be scanned to check whether the key is already present.

Worst case

If every key lands in the same bucket, we get one long chain (or cluster): O(n).

Space

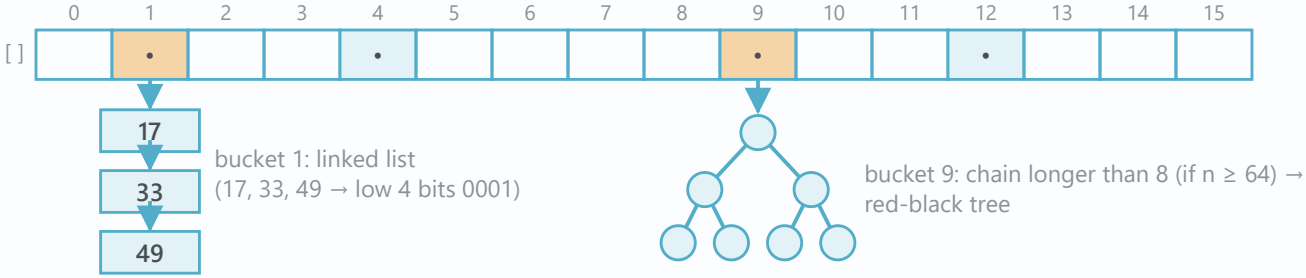
O(n + N): the bucket array plus the pairs. Chaining adds one reference per node.

No order

Finding the smallest key or iterating in order costs O(n); for ordered access, use search trees.

PERFORMANCE AND APPLICATIONS

How Java's `HashMap` Works Inside



Separate chaining, n = 2^k

The bucket count is always a power of 2 (16 by default), so a fast `& (n − 1)` mask replaces mod.

Bit mixing

The mask sees only the low bits; `h ^ (h >>> 16)` folds the high bits into the index too.

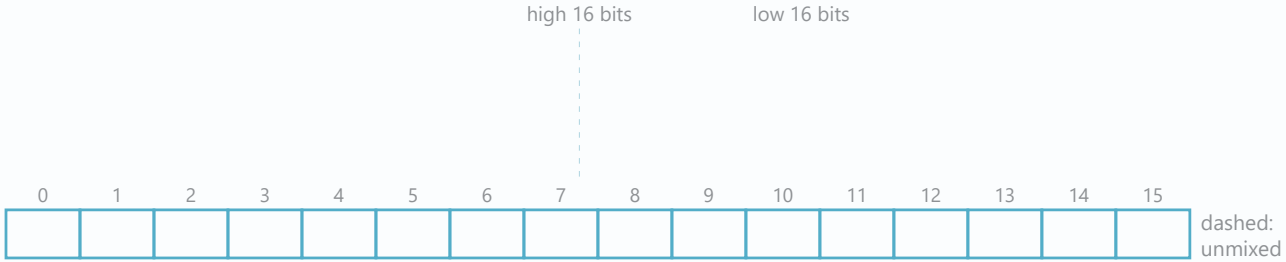
The λ = 0.75 threshold

Once the entry count exceeds 0.75·n, the array **doubles** and the nodes are redistributed.

Treeification (Java 8+)

A chain longer than 8 nodes becomes a tree (if n ≥ 64; otherwise the table grows): worst case O(log n).

Code Trace: `index(k, 16)`: Mix, Then Mask



A table with 16 buckets (n = 2^4). The three keys share their **low 16 bits** (...0001) and differ only in the high bits. The mask `& 15` sees only the last 4 bits.

Step 1 / 11

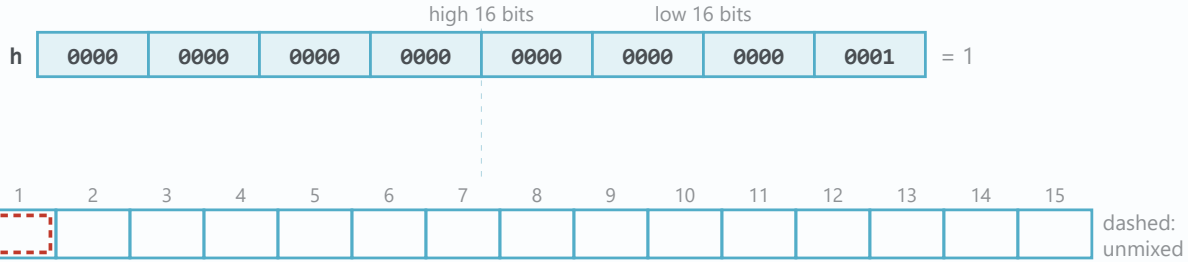
```
1 static int index(Object key, int n) { // n = 2^k
2   int h = key.hashCode();
3   int m = h ^ (h >>> 16); // mix in the high bits
4   return m & (n - 1); // low k bits
5 }
6 // main:
▶ for (int k : new int[]{1, 65537, 131073})
8   System.out.println(k + " -> " + index(k, 16));
```

VARIABLES

`n = 16`

OUTPUT

Code Trace: `index(k, 16)`: Mix, Then Mask



`Integer.hashCode()` is the value itself: **h = 1** = 0x00000001. Without mixing, the bucket would be h & 15 = **1** (dashed).

Step 2 / 11

```
1  static int index(Object key, int n) { // n = 2^k
▶    int h = key.hashCode();
3    int m = h ^ (h >>> 16); // mix in the high bits
4    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
8    System.out.println(k + " -> " + index(k, 16));
```

VARIABLES

`k = 1` `key = 1` `n = 16` `h = 1`

OUTPUT

Code Trace: `index(k, 16)`: Mix, Then Mask



`h >>> 16` = 0: the high half shifts down. XOR gives **m = 1**; the low 4 bits are now 0001.

Step 3 / 11

```
1  static int index(Object key, int n) { // n = 2^k
2    int h = key.hashCode();
▶    int m = h ^ (h >>> 16); // mix in the high bits
4    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
8    System.out.println(k + " -> " + index(k, 16));
```

VARIABLES

`k = 1` `key = 1` `n = 16` `h = 1` `m = 1`

OUTPUT

Code Trace: `index(k, 16)`: Mix, Then Mask



`m & 15` = **1** → key 1 goes to bucket 1; print it.

Step 4 / 11

```
1  static int index(Object key, int n) { // n = 2^k
2    int h = key.hashCode();
3    int m = h ^ (h >>> 16); // mix in the high bits
▶    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
▶    System.out.println(k + " -> " + index(k, 16));
```

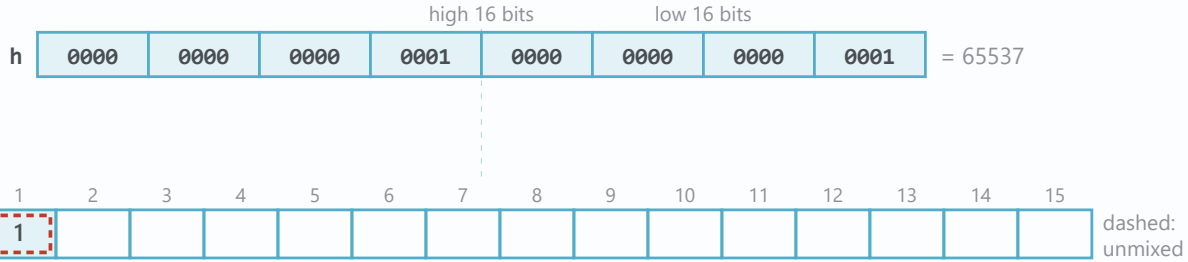
VARIABLES

`k = 1` `key = 1` `n = 16` `h = 1` `m = 1`

OUTPUT

```
1 -> 1
```

Code Trace: `index(k, 16)`: Mix, Then Mask



`Integer.hashCode()` is the value itself: **h = 65537** = 0x00010001. Without mixing, the bucket would be h & 15 = **1** (dashed).

Step 5 / 11

```
1  static int index(Object key, int n) { // n = 2^k
▶    int h = key.hashCode();
3    int m = h ^ (h >>> 16); // mix in the high bits
4    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
8    System.out.println(k + " -> " + index(k, 16));
```

VARIABLES

`k = 65537` `key = 65537` `n = 16` `h = 65537`

OUTPUT

```
1 -> 1
```

Code Trace: `index(k, 16)`: Mix, Then Mask



`h >>> 16` = 1: the high half shifts down. XOR gives **m = 65536**; the low 4 bits are now 0000.

Step 6 / 11

```
1 static int index(Object key, int n) { // n = 2^k
2   int h = key.hashCode();
▶   int m = h ^ (h >>> 16); // mix in the high bits
4   return m & (n - 1); // low k bits
5 }
6 // main:
7 for (int k : new int[]{1, 65537, 131073})
8   System.out.println(k + " -> " + index(k, 16));
```

VARIABLES

`k = 65537` `key = 65537` `n = 16` `h = 65537` `m = 65536`

OUTPUT

```
1 -> 1
```

Code Trace: `index(k, 16)`: Mix, Then Mask



`m & 15` = **0** → key 65537 goes to bucket 0; print it.

Step 7 / 11

```
1  static int index(Object key, int n) { // n = 2^k
2    int h = key.hashCode();
3    int m = h ^ (h >>> 16); // mix in the high bits
▶    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
▶    System.out.println(k + " -> " + index(k, 16));
```

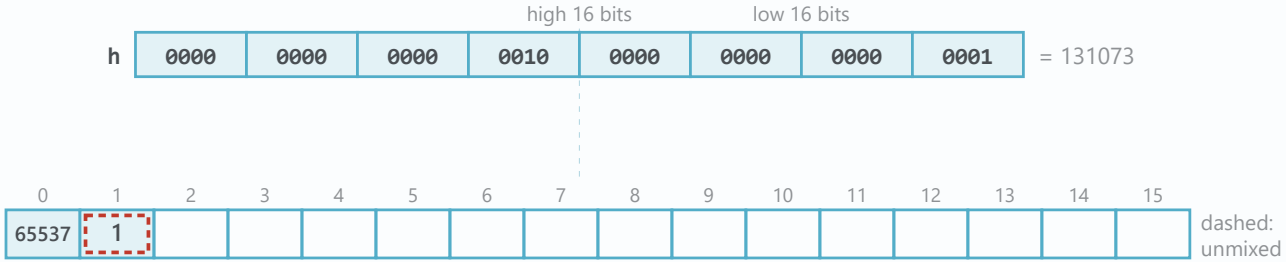
VARIABLES

`k` = **65537** `key` = **65537** `n` = **16**
`h` = **65537** `m` = **65536**

OUTPUT

```
1 -> 1
65537 -> 0
```

Code Trace: `index(k, 16)`: Mix, Then Mask



`Integer.hashCode()` is the value itself: **h = 131073** = 0x00020001. Without mixing, the bucket would be h & 15 = **1** (dashed).

Step 8 / 11

```
1  static int index(Object key, int n) { // n = 2^k
▶    int h = key.hashCode();
3    int m = h ^ (h >>> 16); // mix in the high bits
4    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
8    System.out.println(k + " -> " + index(k, 16));
```

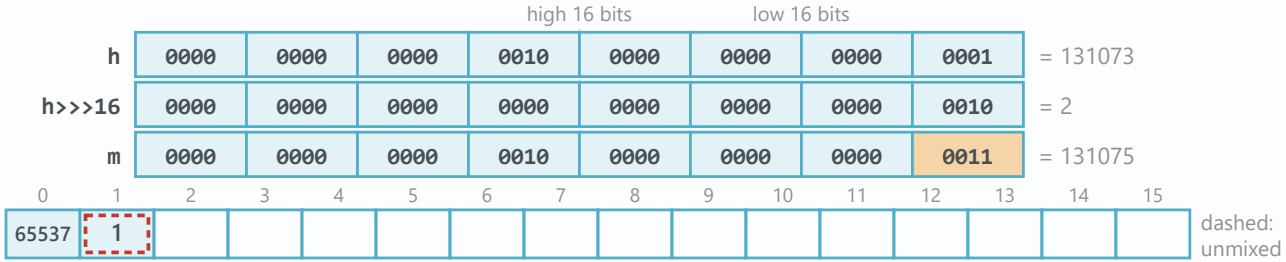
VARIABLES

`k = 131073` `key = 131073` `n = 16` `h = 131073`

OUTPUT

```
1 -> 1
65537 -> 0
```

Code Trace: `index(k, 16)`: Mix, Then Mask



`h >>> 16` = 2: the high half shifts down. XOR gives **m = 131075**; the low 4 bits are now 0011.

Step 9 / 11

```
1  static int index(Object key, int n) { // n = 2^k
2    int h = key.hashCode();
▶    int m = h ^ (h >>> 16); // mix in the high bits
4    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
8    System.out.println(k + " -> " + index(k, 16));
```

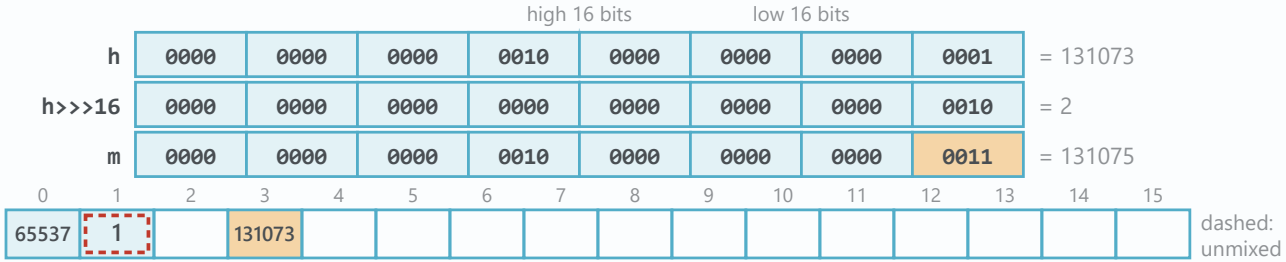
VARIABLES

`k = 131073` `key = 131073` `n = 16` `h = 131073` `m = 131075`

OUTPUT

```
1 -> 1
65537 -> 0
```

Code Trace: `index(k, 16)`: Mix, Then Mask



`m & 15` = **3** → key 131073 goes to bucket 3; print it.

Step 10 / 11

```
1  static int index(Object key, int n) { // n = 2^k
2    int h = key.hashCode();
3    int m = h ^ (h >>> 16); // mix in the high bits
▶    return m & (n - 1); // low k bits
5  }
6  // main:
7  for (int k : new int[]{1, 65537, 131073})
▶    System.out.println(k + " -> " + index(k, 16));
```

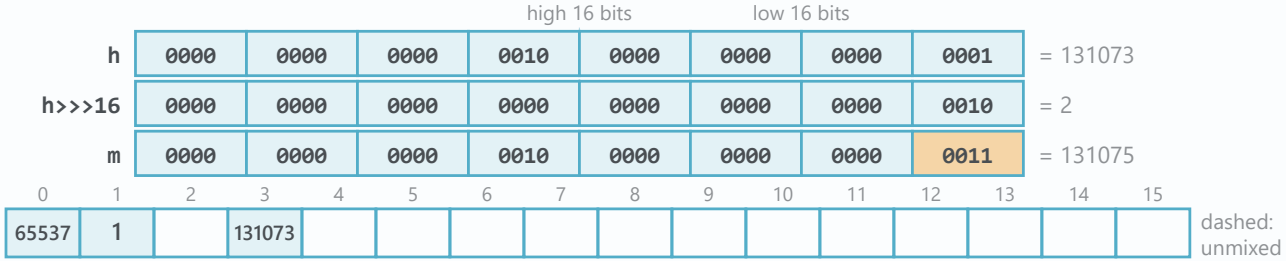
VARIABLES

`k = 131073` `key = 131073` `n = 16` `h = 131073` `m = 131075`

OUTPUT

```
1 -> 1
65537 -> 0
131073 -> 3
```

Code Trace: `index(k, 16)`: Mix, Then Mask



Without mixing, all three would collide in bucket 1; with mixing they spread to buckets 1, 0 and 3. The cost: one shift and one XOR.

Step 11 / 11

```
1 static int index(Object key, int n) { // n = 2^k
2   int h = key.hashCode();
3   int m = h ^ (h >>> 16); // mix in the high bits
4   return m & (n - 1); // low k bits
5 }
6 // main:
▶ for (int k : new int[]{1, 65537, 131073})
8   System.out.println(k + " -> " + index(k, 16));
```

VARIABLES

`n = 16`

OUTPUT

```
1 -> 1
65537 -> 0
131073 -> 3
```

PERFORMANCE AND APPLICATIONS

Where Hash Tables Are Used

Dictionaries

Word → meaning; in spell checking, "is this a word?"

Databases

Fast access by key through hash indexes and hash joins.

In-memory caches

URL or query → precomputed result (e.g., Redis, memcached).

Browser history

Visited address → time of last visit; "have I opened this before?"

Cookie management

Cookie name → value; session ID → user data.

Compilers

Symbol table: variable name → type, address. In Java: `HashMap`, `HashSet`.

Application: `twoSum([8, 3, 11, 5, 7], 12)`

a — target = 12

0	1	2	3	4
8	3	11	5	7

seen (value → index)

{ } empty

We want two indices whose values sum to 12. Brute force tries every pair: $O(n^2)$. Here a **map** remembers each value seen so far, with its index.

Step 1 / 11

```
1  static int[] twoSum(int[] a, int target) {
▶    Map<Integer, Integer> seen = new HashMap<>();
3    for (int i = 0; i < a.length; i++) {
4      int need = target - a[i]; // partner needed
5      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
7      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

`a = [8, 3, 11, 5, 7]` `target = 12` `seen = {}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



i = 0: a[0] = 8, so the partner is **need = 4**. `containsKey(4)` → false: no 4 seen yet.

Step 2 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
▶    for (int i = 0; i < a.length; i++) {
▶      int need = target - a[i]; // partner needed
▶      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
7      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

`a = [8, 3, 11, 5, 7]` `target = 12` `i = 0` `need = 4` `seen = {}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`

a

target = 12

0	1	2	3	4
8	3	11	5	7

▲ i

seen (value → index)

8→0

`seen.put(8, 0)` : later elements can find 8 as a partner.

Step 3 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
3    for (int i = 0; i < a.length; i++) {
4      int need = target - a[i]; // partner needed
5      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
▶      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

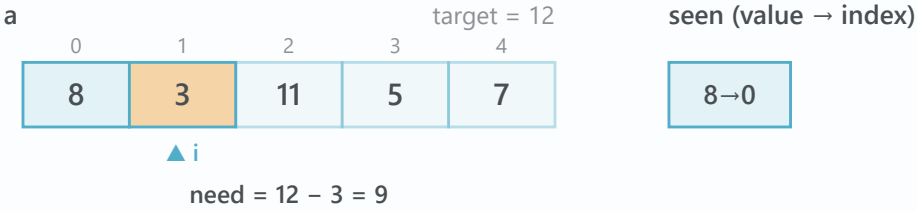
`a = [8, 3, 11, 5, 7]` `target = 12` `i = 0` `need = 4` `seen = {8=0}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



i = 1: a[1] = 3, so the partner is **need = 9**. `containsKey(9)` → false: no 9 seen yet.

Step 4 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
▶    for (int i = 0; i < a.length; i++) {
▶      int need = target - a[i]; // partner needed
▶      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
7      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

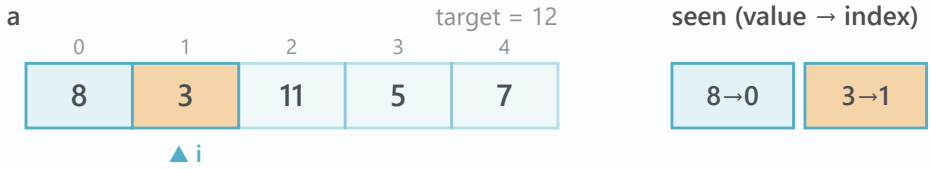
`a = [8, 3, 11, 5, 7]` `target = 12` `i = 1` `need = 9` `seen = {8=0}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



`seen.put(3, 1)` : later elements can find 3 as a partner.

Step 5 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
3    for (int i = 0; i < a.length; i++) {
4      int need = target - a[i]; // partner needed
5      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
▶      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

`a = [8, 3, 11, 5, 7]` `target = 12`

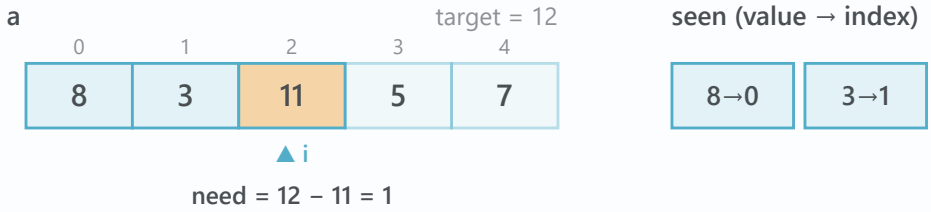
`i = 1` `need = 9` `seen = {8=0, 3=1}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



i = 2: a[2] = 11, so the partner is **need = 1**. `containsKey(1)` → false: no 1 seen yet.

Step 6 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
▶    for (int i = 0; i < a.length; i++) {
▶      int need = target - a[i]; // partner needed
▶      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
7      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

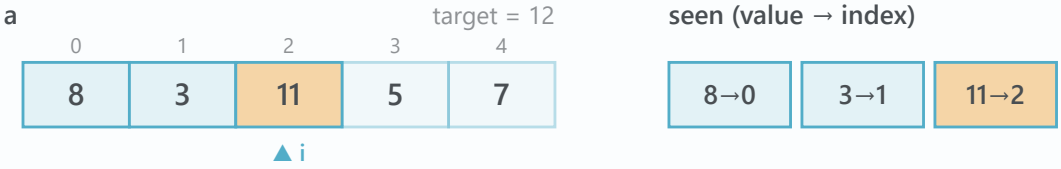
`a = [8, 3, 11, 5, 7]` `target = 12` `i = 2` `need = 1` `seen = {8=0, 3=1}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



`seen.put(11, 2)` : later elements can find 11 as a partner.

Step 7 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
3    for (int i = 0; i < a.length; i++) {
4      int need = target - a[i]; // partner needed
5      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
▶      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

`a = [8, 3, 11, 5, 7]` `target = 12`

`i = 2` `need = 1`

`seen = {8=0, 3=1, 11=2}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



i = 3: a[3] = 5, so the partner is **need = 7**. `containsKey(7)` → false: no 7 seen yet.

Step 8 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
▶    for (int i = 0; i < a.length; i++) {
▶      int need = target - a[i]; // partner needed
▶      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
7      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

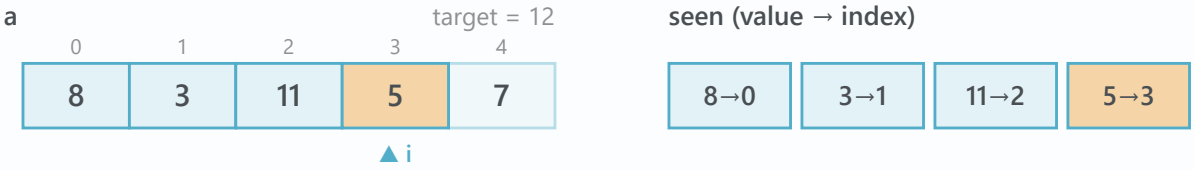
`a = [8, 3, 11, 5, 7]` `target = 12` `i = 3` `need = 7` `seen = {8=0, 3=1, 11=2}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



`seen.put(5, 3)` : later elements can find 5 as a partner.

Step 9 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
3    for (int i = 0; i < a.length; i++) {
4      int need = target - a[i]; // partner needed
5      if (seen.containsKey(need))
6        return new int[]{seen.get(need), i};
▶      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

`a = [8, 3, 11, 5, 7]` `target = 12`

`i = 3` `need = 7`

`seen = {8=0, 3=1, 11=2, 5=3}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



i = 4: need = 5, and `containsKey(5)` is **true** (O(1) on average) → a[3] + a[4] = 5 + 7 = 12. **return [3, 4]**.

Step 10 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
3    for (int i = 0; i < a.length; i++) {
▶      int need = target - a[i]; // partner needed
▶      if (seen.containsKey(need))
▶        return new int[]{seen.get(need), i};
7      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

`a = [8, 3, 11, 5, 7]` `target = 12`

`i = 4` `need = 5`

`seen = {8=0, 3=1, 11=2, 5=3}`

CALL STACK

`twoSum(a, 12)`

`main()`

Application: `twoSum([8, 3, 11, 5, 7], 12)`



One pass, with an average O(1) lookup and insert per step → **O(n)** time in total, plus O(n) extra memory.

Step 11 / 11

```
1  static int[] twoSum(int[] a, int target) {
2    Map<Integer, Integer> seen = new HashMap<>();
3    for (int i = 0; i < a.length; i++) {
4      int need = target - a[i]; // partner needed
5      if (seen.containsKey(need))
▶        return new int[]{seen.get(need), i};
7      seen.put(a[i], i); // value -> index
8    }
9    return null; // no pair
10 }
```

VARIABLES

`a = [8, 3, 11, 5, 7]` `target = 12`

`i = 4` `need = 5`

`seen = {8=0, 3=1, 11=2, 5=3}`

CALL STACK

`main()`

WORKED EXAMPLE

A Hash Function for Student IDs

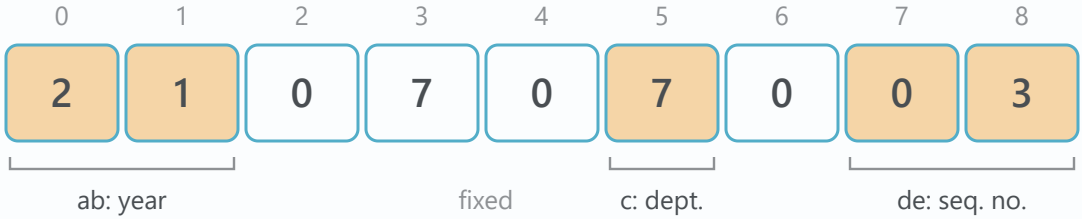
Problem

Spread student IDs evenly over a 1000-element array.

Solution

Design a hash function that exploits the structure of the ID, cutting collisions.

ID format: `ab070c0de` · example: `210707003`



- **ab**: enrollment year (20–24) · **c**: department (6 or 7) · **de**: sequence number (00–99, two digits)
- Goal: map this format to an array index in the range 0–999.

WORKED EXAMPLE

The Student ID Hash Function

$$\text{hash}(ab070c0de) = ((ab - 20) \times 100 + (c - 6) \times 500 + de) \bmod 1000$$

ID	ab	c	de	Computation	Index
`210706003`	21	6	03	(1×100 + 0×500 + 3) mod 1000	**103**
`240707050`	24	7	50	(4×100 + 1×500 + 50) mod 1000	**950**
`230707042`	23	7	42	(3×100 + 1×500 + 42) mod 1000	**842**

Year
$(ab - 20) \in 0 \ldots 4$ selects a block of 100.

Department
$(c - 6) \in 0, 1$ shifts by 500.

Sequence
$de \in 0 \ldots 99$ is added directly.

5 years × 2 departments × 100 sequence numbers = **1000** possible IDs, mapped to 1000 different indices: no collisions at all. This is a **perfect hash**, possible only because we know the structure of the keys.

WORKED EXAMPLE

The calculateHash Function

```
// s: student ID of the form "ab070c0de"
public int calculateHash(String s) {
  int ab = Integer.parseInt(s.substring(0, 2));
  int c  = Integer.parseInt(s.substring(5, 6));
  int de = Integer.parseInt(s.substring(7, 9));
  int year = (ab - 20) * 100;     // year block
  int dept = (c - 6) * 500;       // dept: 0/500
  return (year + dept + de) % 1000;
}
```

substring(begin, end)

end is excluded: `substring(5, 6)` gives only character 5 (c); `substring(7, 9)` gives characters 7 and 8 (de).

Example

`calculateHash("240707050")`

ab = 24, c = 7, de = 50

400 + 500 + 50 = **950**

Input that breaks the format (a year outside 20–24, c ∉ {6, 7}) can collide or even go negative; real code must validate its input.

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`

0	1	2	3	4	5	6	7	8
2	4	0	7	0	7	0	5	0

`calculateHash(\"240707050\")` : indices 0 … 8.

Step 1 / 14

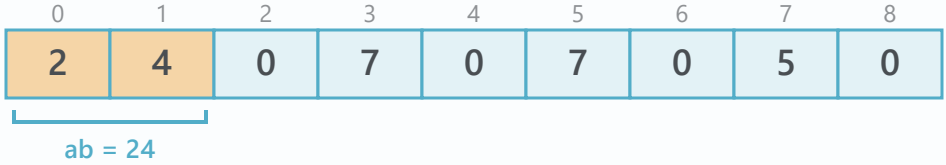
```
1 // s: student ID of the form "ab070c0de"
▶ public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "240707050"`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



`substring(0, 2)` = \"24\" (end excluded) → **ab = 24** (enrollment year).

Step 2 / 14

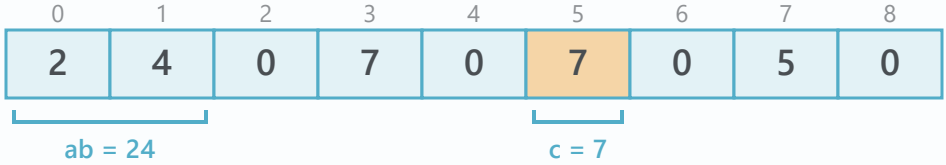
```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
▶   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "240707050"` `ab = 24`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



`substring(5, 6)` = \"7\" → **c = 7** (department).

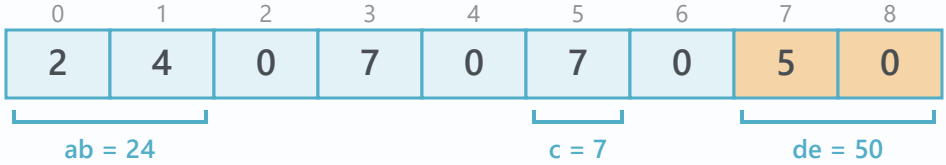
Step 3 / 14

```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
▶   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "240707050"` `ab = 24` `c = 7`

Code Trace: `calculateHash()`



`substring(7, 9)` = \"50\" → **de = 50** (sequence number).

Step 4 / 14

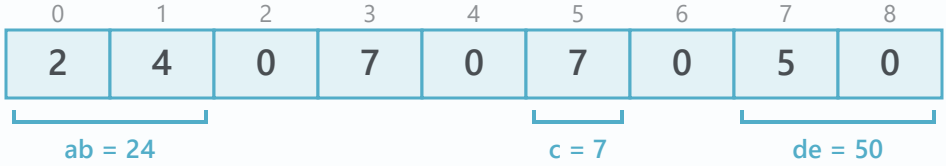
```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
▶   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "240707050"` `ab = 24` `c = 7` `de = 50`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



year = (24 − 20) · 100 = **400**: each year gets a block of 100.

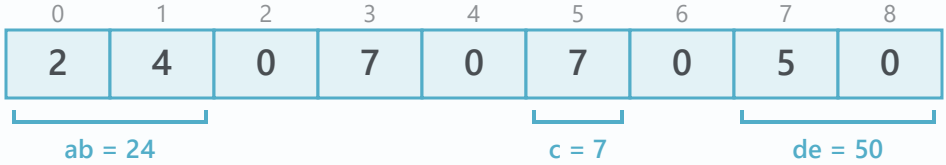
Step 5 / 14

```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
▶   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "240707050"` `ab = 24` `c = 7` `de = 50` `year = 400`

Code Trace: `calculateHash()`



dept = (7 − 6) · 500 = **500**: department 7 shifts by 500.

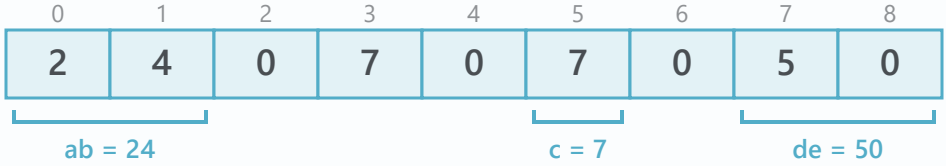
Step 6 / 14

```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
▶   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "240707050"` `ab = 24` `c = 7` `de = 50` `year = 400` `dept = 500`

Code Trace: `calculateHash()`



`return (400 + 500 + 50) % 1000` = **950**.

Step 7 / 14

```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
▶   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "240707050"` `ab = 24` `c = 7` `de = 50` `year = 400` `dept = 500`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`

0	1	2	3	4	5	6	7	8
2	1	0	7	0	6	0	0	3

`calculateHash(\"210706003\")` : indices 0 … 8.

Step 8 / 14

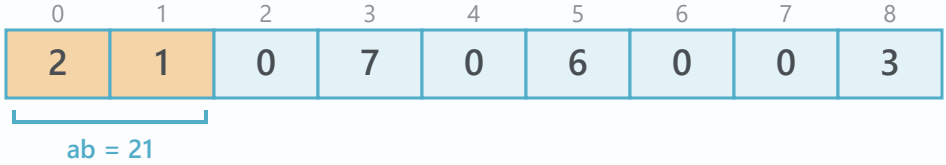
```
1 // s: student ID of the form "ab070c0de"
▶ public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "210706003"`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



`substring(0, 2)` = \"21\" (end excluded) → **ab = 21** (enrollment year).

Step 9 / 14

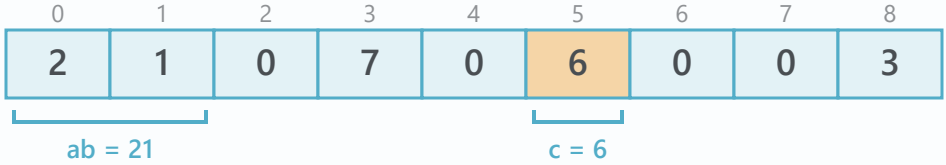
```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
▶   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "210706003"` `ab = 21`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



`substring(5, 6)` = \"6\" → **c = 6** (department).

Step 10 / 14

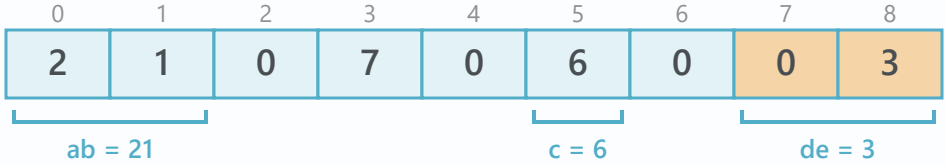
```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
▶   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "210706003"` `ab = 21` `c = 6`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



`substring(7, 9)` = \"03\" → **de = 3** (sequence number).

Step 11 / 14

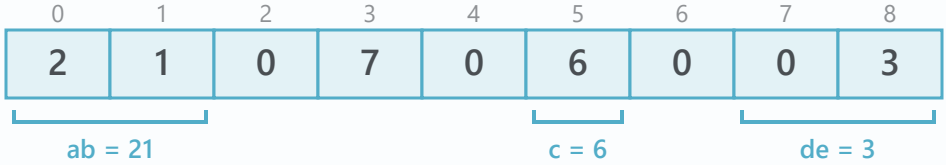
```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
▶   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "210706003"` `ab = 21` `c = 6` `de = 3`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



year = (21 − 20) · 100 = **100**: each year gets a block of 100.

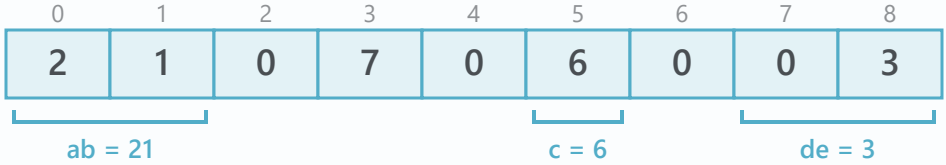
Step 12 / 14

```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
▶   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "210706003"` `ab = 21` `c = 6` `de = 3` `year = 100`

Code Trace: `calculateHash()`



dept = (6 − 6) · 500 = **0**: department 6 is not shifted.

Step 13 / 14

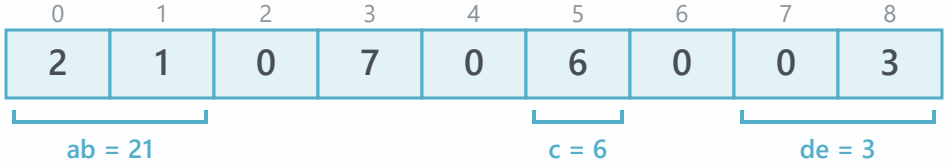
```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
▶   int dept = (c - 6) * 500;        // dept: 0/500
8   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "210706003"` `ab = 21` `c = 6` `de = 3` `year = 100` `dept = 0`

WORKED EXAMPLE · STEP THROUGH THE CODE

Code Trace: `calculateHash()`



`return (100 + 0 + 3) % 1000` = **103**.

Step 14 / 14

```
1 // s: student ID of the form "ab070c0de"
2 public int calculateHash(String s) {
3   int ab = Integer.parseInt(s.substring(0, 2));
4   int c  = Integer.parseInt(s.substring(5, 6));
5   int de = Integer.parseInt(s.substring(7, 9));
6   int year = (ab - 20) * 100;      // year block
7   int dept = (c - 6) * 500;        // dept: 0/500
▶   return (year + dept + de) % 1000;
9 }
```

VARIABLES

`s = "210706003"` `ab = 21` `c = 6` `de = 3` `year = 100` `dept = 0`

SUMMARY

Summary

The map ADT

Unique key → value. `put`, `get`, `remove`; a put with an existing key is an update.

Hash functions

Hash code + compression (`mod N`, N prime). Deterministic, fast, uniform.

Separate chaining

Each bucket is a linked list. Insert at the head; cost O(1 + λ). λ > 1 is allowed.

Open addressing

Linear (primary clustering), quadratic (secondary clustering), double hashing. Deletion uses a "deleted" marker.

Load factor λ = n/N

Past the threshold (≈ 0.5–0.75) the table roughly doubles and every key is rehashed.

Correctness and complexity

Equal keys must have equal hash codes. O(1) on average, O(n) in the worst case; resizing is amortized O(1).

SUMMARY

Interactive Simulators

Hash Table Simulator →

Separate chaining, step-by-step hash computation, visualized collisions, and automatic resizing.

Map Simulator →

set, get, has and delete; iterating over keys and tracking insertion order.

Open Addressing Simulator →

Linear and quadratic probing and double hashing; probe sequences step by step, the deleted marker (*X*), and the load factor.

SUMMARY

Common Mistakes

`Math.abs(h) % N`

`Math.abs(Integer.MIN_VALUE)` is negative → an index-out-of-bounds error. Use `Math.floorMod(h, N)`.

Overriding only `equals`

Equal keys land in different buckets and `get` returns null. Override both.

Mutable keys

A key whose hashed fields change after insertion is stranded in the wrong bucket and can't be found.

`==` on `Integer`

Boxed values outside −128…127 are separate objects, so `==` gives false; use `equals`.

Deleting with `null` under open addressing

It breaks probe sequences; you need a "deleted" (DEFUNCT) marker.

Expecting order

`HashMap` does not iterate in insertion order; use `LinkedHashMap` or `TreeMap`.

SUMMARY

Sample Problems

1. With N = 7, h(k) = k mod 7 and **linear probing**, insert 10, 17, 24, 3, 5 in order. What is the final table?
2. Insert the same keys with **quadratic probing**.
3. A chained table with N = 10 buckets holds 25 pairs. What is λ? How many nodes does an unsuccessful search examine on average?
4. Under open addressing, why can't we simply set a deleted slot to `null`?
5. Compute hash(ab070c0de) for the ID 230706015.
6. Which buckets do the keys 0, 12, 24, 36 land in under `k mod N` with N = 12 and with N = 11?

▶ **Show answers**

SUMMARY

Exercises

1. With N = 11, h(k) = k mod 11 and $h_2(k) = 7 - (k \bmod 7)$, use **double hashing**: which slots do 14, 25, 36, 3 end up in?
2. A chained table with N = 5 holds the keys 12, 7, 20, 33. Which buckets do they move to when it grows to N = 11?
3. Under linear probing with λ = 0.8, how many probes does an unsuccessful search make on average?
4. Write a loop that uses a `HashMap` to count how often each word occurs in an array of words.
5. How does the table behave if `hashCode()` returns 42 for every object?
6. What are `index(32, 16)` and `index(65568, 16)`? And without mixing?

▶ **Answers and hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- Oracle, Java SE 17 API documentation for `Map`, `HashMap`, and `HashSet`.

The End

Chapter 12: Maps and Hash Tables

Next deck: Chapter 13: Sets and Disjoint Sets · Study questions · Simulators: Hash Table, Map, Open Addressing · Glossary