Chapter 4: Searching Algorithms

Algorithms

The search problem | Linear search | Binary search | Hashing | Interpolation search | Jump search | Exponential search | Comparison

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/algorithms/slides/Chapter_04_Searching_Algorithms.html

Learning Outcomes

Basic searches

Implement linear and binary search (binary both iteratively and recursively) in Java, trace them on an array, and work out their best, average and worst cases.

Hashing

Explain hash functions, buckets, collisions, linear probing and chaining; explain how the load factor affects cost.

Advanced searches on sorted arrays

Justify the idea, code and complexity of interpolation, jump and exponential search: O(log log n), O(√n), O(log i).

Choosing

Choose a search algorithm based on whether the data is sorted, its distribution and its size; avoid common implementation mistakes.

The Search Problem

- Search algorithms find a **desired value** in a collection of data.
- They scan the data by a **search criterion** (usually a **key**) and try to find a matching item.

Input

An array `arr` of n items and a `target` value.

Output

An **index** i with `arr[i] == target`; **−1** if the target is absent (unsuccessful search).

Cost measure: search time is usually measured by how many times the target is **compared** with array items (the number of probes). Every complexity in this chapter counts this number.

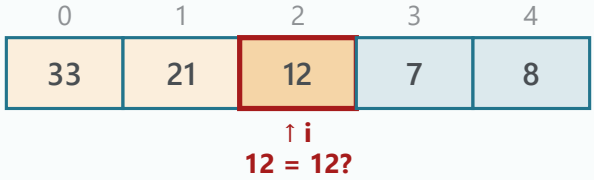
Kinds of Search Algorithms

Search algorithms differ in how they work and in what they require.

Algorithm	Requires	How it works
Linear search	nothing	Compares the items one by one.
Binary search	sorted array	Halves the search range at each step.
Hash search	hash table	Items are stored in a hash table; the search looks in the key's bucket.
Interpolation search	sorted, uniform	Estimates the position from the ratio of values.
Jump search	sorted array	Jumps ahead in blocks of $\sqrt{n}$, then searches linearly inside a block.
Exponential search	sorted array	Grows a bound as 1, 2, 4, 8, …, then runs binary search.

Linear Search

- Searches by checking **every item in turn**.
- Like checking the items on a shopping list in order, or searching a room **systematically** for something you lost.
- The target is compared with each item, starting from the **beginning** of the list:
  - if found, its **position** (index) is returned;
  - if absent, the failure value **−1** is returned.
- Sometimes looking one by one is the **best** way to find something: if the array is unsorted there is no other option.



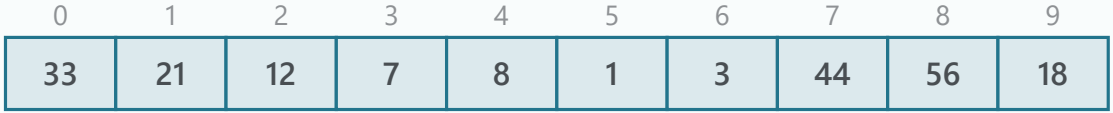
The Linear Search Algorithm

```
int linearSearch(int[] arr, int n,
                 int target) {
  int i = 0;
  while (i < n && arr[i] != target)
    i++;
  if (i < n)
    return i;       // found: index
  else
    return -1;      // not found
}
```

- **Parameters:** `arr` the items to search, `n` the array size, `target` the item to find.
- **Returns:** the index of the target; −1 if not found.
- `i` walks over the array; the `while` loop continues while **i < n** and **arr[i] ≠ target**, incrementing i each time.
- After the loop, the `if` checks whether the item was found: if **i < n** it returns i (the item's index), otherwise −1.

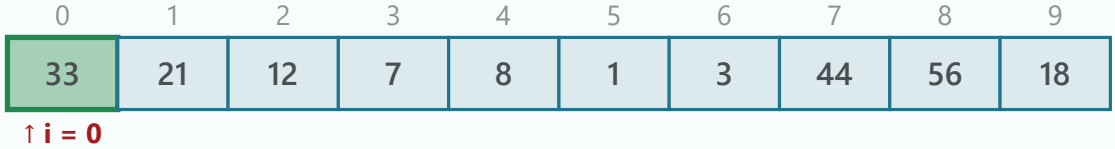
The order of the conditions matters: `i < n` is tested first, and thanks to Java's short-circuit `&&`, `arr[n]` is never accessed when `i == n`.

Linear Search: Best and Worst Case



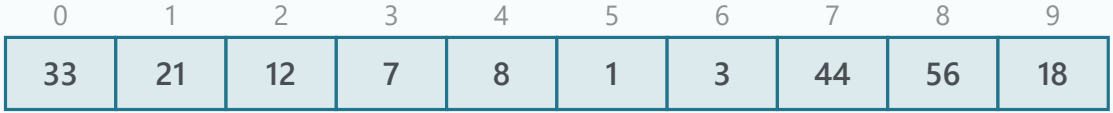
Best case: search for 33. The array is unsorted: 33, 21, 12, 7, 8, 1, 3, 44, 56, 18 (n = 10).

Linear Search: Best and Worst Case



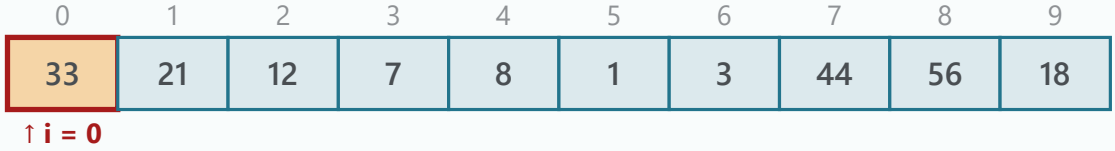
arr[0] = 33 = target → the loop ends at once; **returns 0**. Only **1 comparison**: best case O(1).

Linear Search: Best and Worst Case



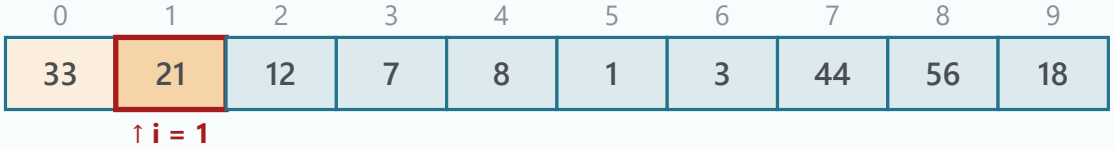
Worst case: search for 17. 17 is not in the array; every item must be checked.

Linear Search: Best and Worst Case



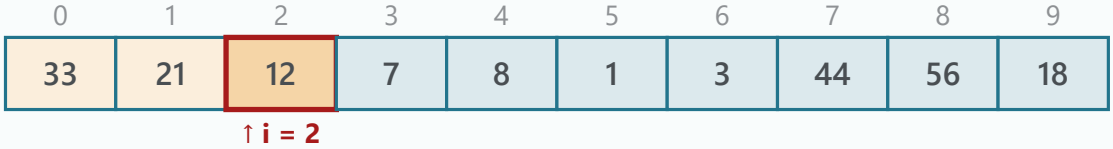
i = 0: arr[0] = 33 ≠ 17 → i = 1.

Linear Search: Best and Worst Case



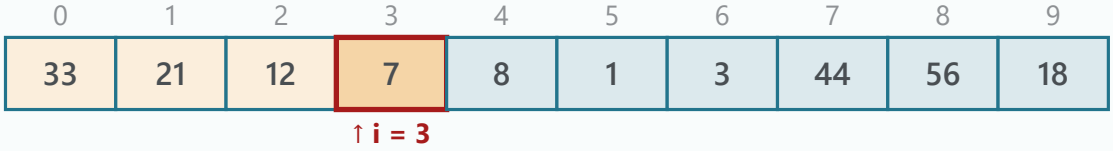
i = 1: arr[1] = 21 ≠ 17 → i = 2.

Linear Search: Best and Worst Case



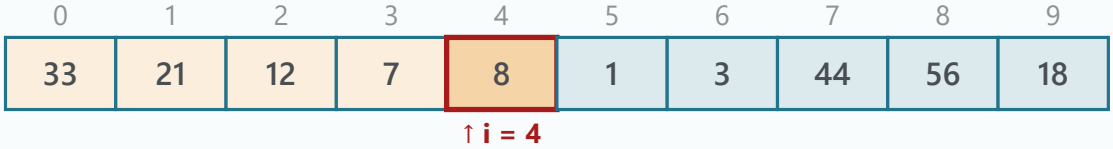
i = 2: arr[2] = 12 ≠ 17 → i = 3.

Linear Search: Best and Worst Case



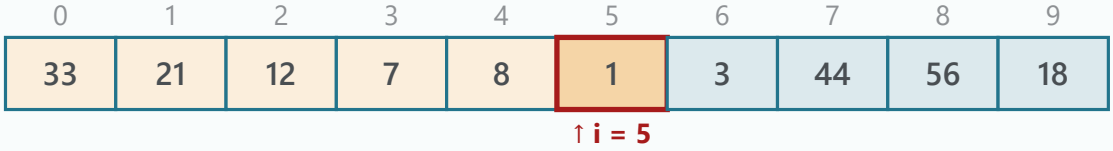
i = 3: arr[3] = 7 ≠ 17 → i = 4.

Linear Search: Best and Worst Case



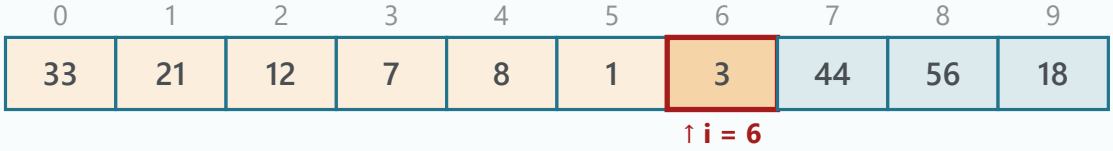
i = 4: arr[4] = 8 ≠ 17 → i = 5.

Linear Search: Best and Worst Case



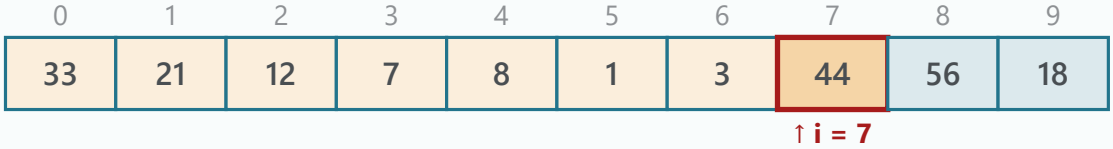
i = 5: arr[5] = 1 ≠ 17 → i = 6.

Linear Search: Best and Worst Case



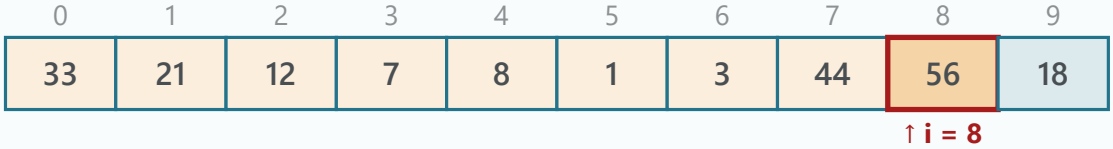
i = 6: arr[6] = 3 ≠ 17 → i = 7.

Linear Search: Best and Worst Case



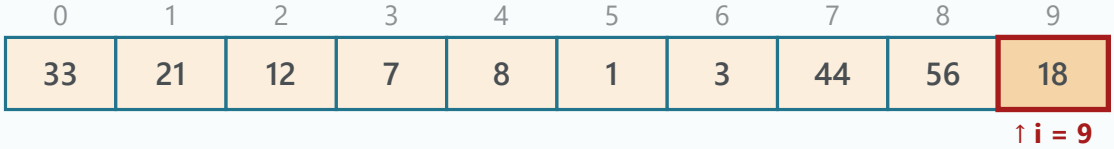
i = 7: arr[7] = 44 ≠ 17 → i = 8.

Linear Search: Best and Worst Case



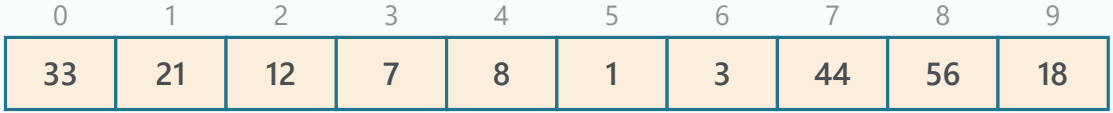
i = 8: arr[8] = 56 ≠ 17 → i = 9.

Linear Search: Best and Worst Case



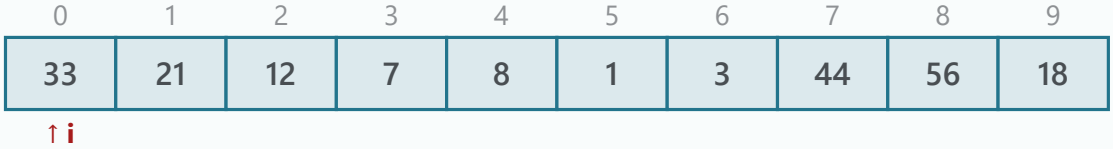
i = 9: arr[9] = 18 ≠ 17 → i = 10.

Linear Search: Best and Worst Case



i = 10 = n → the loop ends; `i < n` is false, so it **returns −1 (not found)**. **n = 10 comparisons**: worst case O(n).

linearSearch(arr, 10, 1): Tracing the Code



Start with `i` = 0. Target: 1 (an **average case** example).

Step 1 / 13

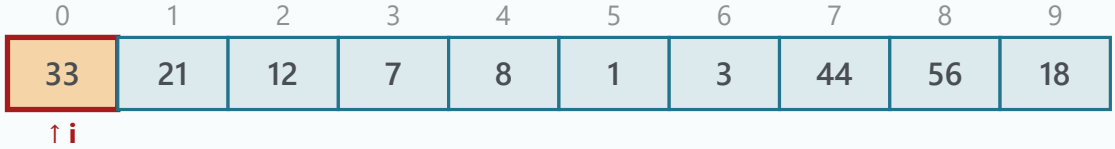
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
▶    int i = 0;
4    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 0`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



i < n and arr[0] = 33 ≠ 1 → the loop body runs.

Step 2 / 13

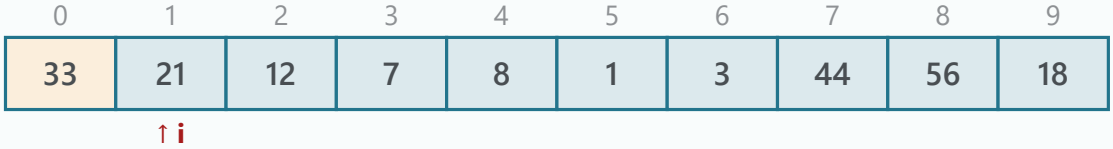
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 0`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



`i++` → i = 1.

Step 3 / 13

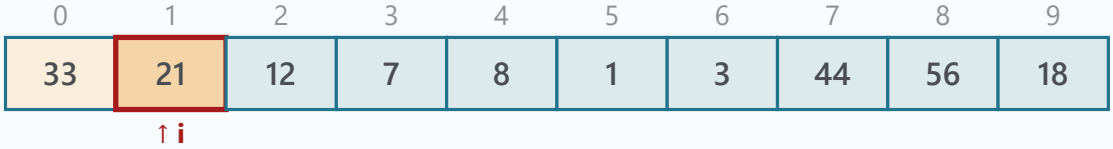
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 1`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



i < n and arr[1] = 21 ≠ 1 → the loop body runs.

Step 4 / 13

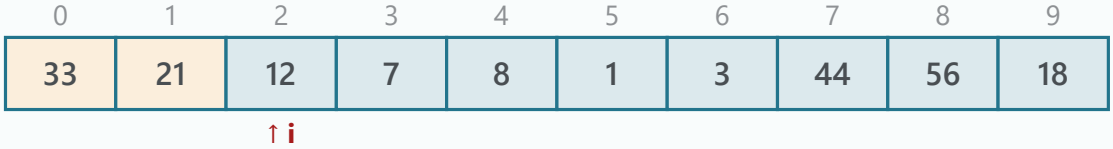
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 1`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



`i++` → i = 2.

Step 5 / 13

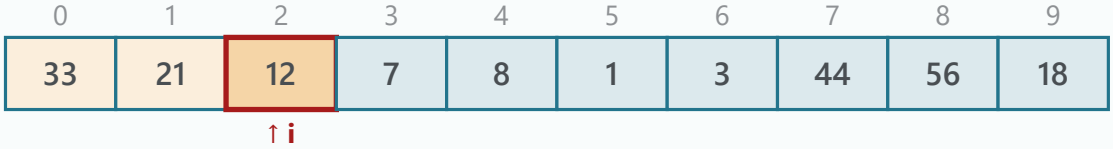
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 2`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



i < n and arr[2] = 12 ≠ 1 → the loop body runs.

Step 6 / 13

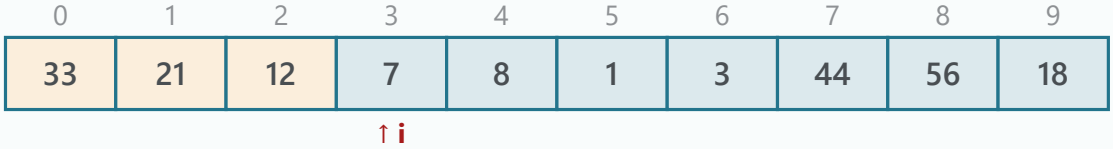
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
▶    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 2`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



`i++` → i = 3.

Step 7 / 13

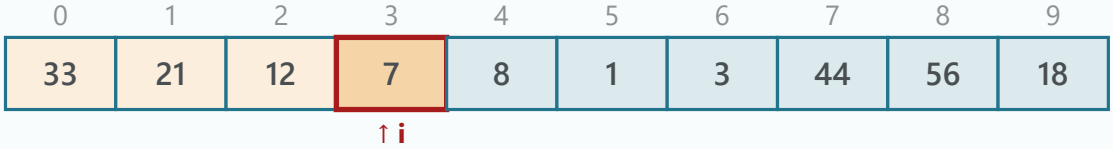
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 3`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



i < n and arr[3] = 7 ≠ 1 → the loop body runs.

Step 8 / 13

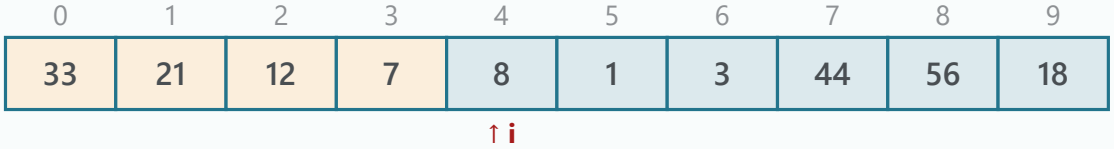
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

n = 10 target = 1 i = 3

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



`i++` → i = 4.

Step 9 / 13

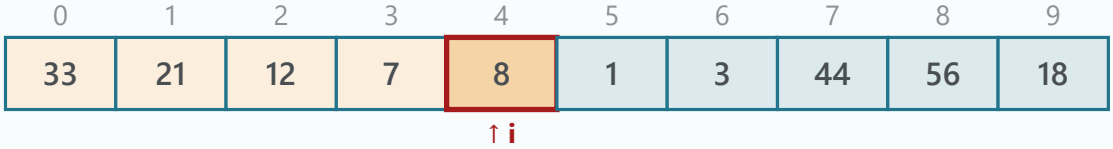
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 4`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



i < n and arr[4] = 8 ≠ 1 → the loop body runs.

Step 10 / 13

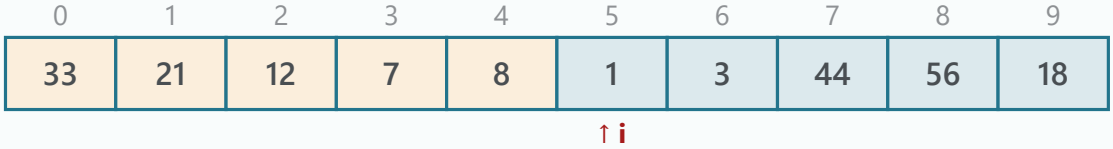
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
▶    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 4`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



`i++` → i = 5.

Step 11 / 13

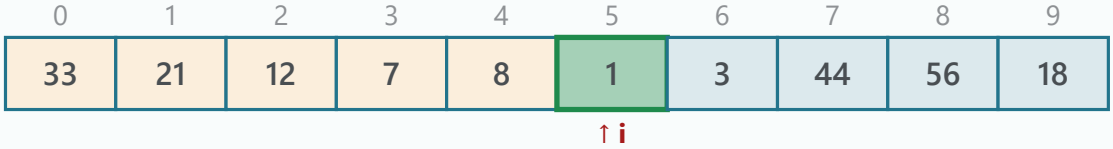
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 5`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



arr[5] = 1 = target → the `while` condition is false and the loop exits.

Step 12 / 13

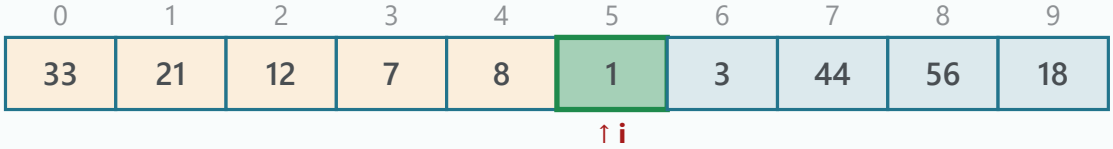
```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 5`

OUTPUT

linearSearch(arr, 10, 1): Tracing the Code



i = 5 < n → **returns 5**. 6 comparisons were made.

Step 13 / 13

```
1  int linearSearch(int[] arr, int n,
2                   int target) {
3    int i = 0;
4    while (i < n && arr[i] != target)
5      i++;
▶    if (i < n)
▶      return i;
8    else
9      return -1;
10 }
```

VARIABLES

`n = 10` `target = 1` `i = 5`

OUTPUT

```
linearSearch(arr, 10, 1) → 5
```

Linear Search: Complexity and Assessment

Case	Comparisons	Order	Example (n = 10)
Best (first item)	1	O(1)	33 → 1 comparison
Average (successful, equally likely)	(1 + 2 + … + n) / n = (n + 1)/2	O(n)	average 5.5
Worst / unsuccessful	n	O(n)	17 → 10 comparisons

Advantages

+ Easy to write and understand; needs no complex data structure.
+ **No preprocessing**: the array need not be sorted.
+ Ideal for small data sets (e.g. a number in a contact list).
+ Also works on sequential-access structures such as linked lists.

Disadvantages

– Slow on large data: 500,000 comparisons on average for 10^6 items.
– Even if the array is sorted, it **does not exploit** that.
– With many queries, sorting once and using binary search is cheaper.

Linear Search with a Sentinel

```
int sentinelSearch(int[] arr, int n,
                   int target) {
  if (n == 0) return -1;
  int last = arr[n - 1];
  arr[n - 1] = target;     // sentinel
  int i = 0;
  while (arr[i] != target)
    i++;
  arr[n - 1] = last;  // restore
  if (i < n - 1 || last == target)
    return i;
  return -1;
}
```

- The classic loop makes **two** comparisons per iteration: `i < n` and `arr[i] != target`.
- If the target is written into the last cell as a **sentinel**, the loop is guaranteed to stop; the `i < n` test is no longer needed.
- The comparison count drops from ≈ 2n to ≈ n + 2; the order is still **O(n)**.

Since the last item is changed temporarily, the array must not be **read-only** or read by another thread at the same time.

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
▶    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
7    while (arr[i] != target)
8      i++;
9    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 0

arr = [7, 3, 9, 4, 6], target = 5 (absent). The last item, 6, is saved in `last`.

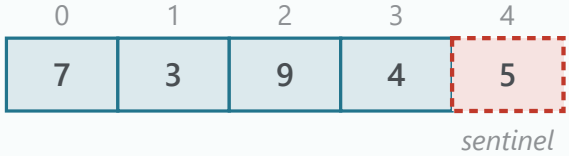
Step 1 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = –`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
▶    arr[n - 1] = target;      // sentinel
▶    int i = 0;
7    while (arr[i] != target)
8      i++;
9    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 0

The target 5 is written into the last cell as a **sentinel**: the loop now stops here at the latest. `i = 0` .

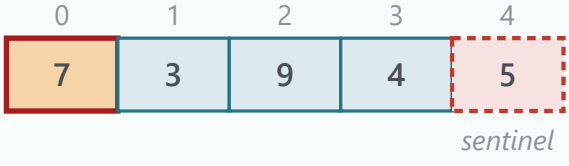
Step 2 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 0`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
▶    while (arr[i] != target)
▶      i++;
9    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 1

arr[0] = 7 ≠ 5 → `i++`. Only **one** comparison; no `i < n` test.

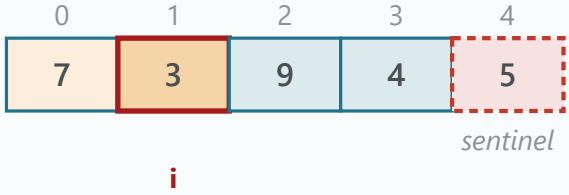
Step 3 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 0`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
▶    while (arr[i] != target)
▶      i++;
9    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 2

arr[1] = 3 ≠ 5 → `i++`. Only **one** comparison; no `i < n` test.

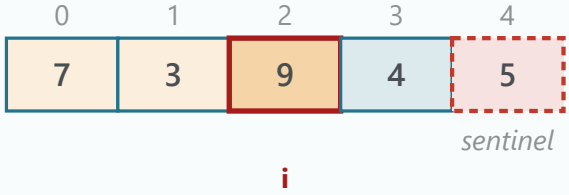
Step 4 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 1`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
▶    while (arr[i] != target)
▶      i++;
9    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 3

arr[2] = 9 ≠ 5 → `i++`. Only **one** comparison; no `i < n` test.

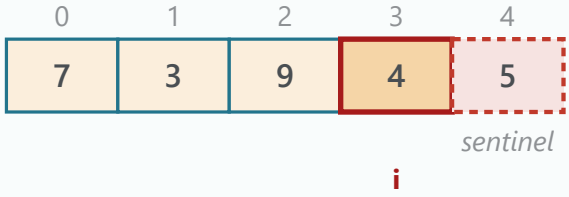
Step 5 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 2`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
▶    while (arr[i] != target)
▶      i++;
9    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 4

arr[3] = 4 ≠ 5 → `i++`. Only **one** comparison; no `i < n` test.

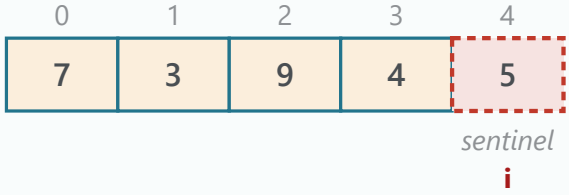
Step 6 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 3`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
▶    while (arr[i] != target)
8      i++;
9    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 5

arr[4] = 5 = target: the loop stops. But what it found is the sentinel, not a real item.

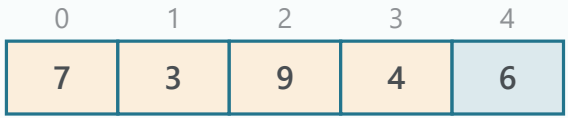
Step 7 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 4`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
7    while (arr[i] != target)
8      i++;
▶    arr[n - 1] = last;  // restore
10   if (i < n - 1 || last == target)
11     return i;
12   return -1;
13 }
```



last = 6 **target = 5**

comparisons: 5

The last item is restored: arr is [7, 3, 9, 4, 6] again.

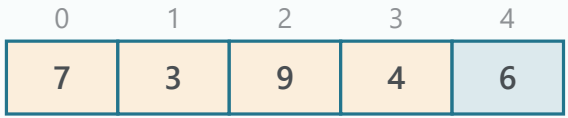
Step 8 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 4`

sentinelSearch(arr, 5, 5): Tracing the Code

```
1  int sentinelSearch(int[] arr, int n,
2                     int target) {
3    if (n == 0) return -1;
4    int last = arr[n - 1];
5    arr[n - 1] = target;      // sentinel
6    int i = 0;
7    while (arr[i] != target)
8      i++;
9    arr[n - 1] = last;  // restore
▶    if (i < n - 1 || last == target)
11     return i;
▶    return -1;
13 }
```



last = 6 target = 5

comparisons: 7

i < n − 1? 4 < 4 is false; last == 5? 6 ≠ 5 → `return -1`. **7** comparisons in total = n + 2; the classic loop would make 2n = 10.

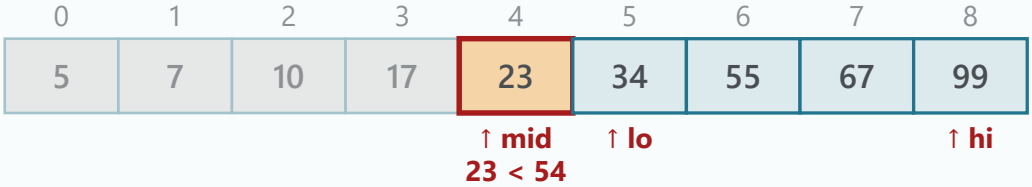
Step 9 / 9

VARIABLES

`n = 5` `target = 5` `last = 6` `i = 4` `result = -1`

Binary Search

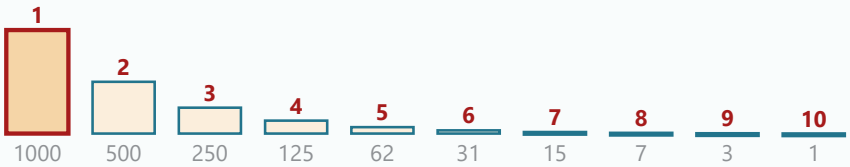
- Like **looking for a book in a library** whose books are shelved alphabetically by the author's surname.
- Instead of searching every shelf, go to the **middle** first.
  - If the book is **to the left** of the middle, continue on the left shelves;
  - if it is **to the right**, continue on the right shelves.
- Each comparison **halves** the part left to search.
- Precondition: the array must be **sorted**, and items must be accessible by index in O(1).



Why Is Binary Search Fast?

- It is fast on sorted lists: the remaining part **halves** at every step.
- On a list of 1000 items, linear search makes **500 checks on average**, while binary search finishes in at most **10 steps**.
- 1000 → 500 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1: one comparison per range size, 10 in total.
- Ideal for searching large, sorted data sets.

n	Linear (worst)	Binary (worst)
10	10	4
1,000	1,000	10
10^6	10^6	20
10^9	10^9	30



Binary search makes at most $\lfloor \log_2 n \rfloor + 1$ comparisons. Growing the data 1000-fold adds only ≈ 10 steps.

The Binary Search Algorithm

```
int binarySearch(int[] arr, int n, int target) {
  int lo = 0, hi = n - 1;
  while (lo <= hi) {
    int mid = (lo + hi) / 2;
    if (target < arr[mid])
      hi = mid - 1;      // left half
    else if (target > arr[mid])
      lo = mid + 1;      // right half
    else
      return mid;         // found
  }
  return -1;
}
```

- `lo` and `hi` bound the range being searched.
- The `while` loop continues as long as `lo ≤ hi` (the range is not empty).
- `mid` is the index of the middle item of the range at each step.
- If the target is **smaller** than the middle, continue in the lower half; if **larger**, in the upper half; if **equal**, return `mid`.
- If the range becomes empty (`lo > hi`), return −1.

Recursive Binary Search

```
int search(int[] arr, int lo,
        int hi, int target) {
  if (lo > hi) return -1;  // empty range
  int mid = (lo + hi) / 2;
  if (arr[mid] == target) return mid;
  if (arr[mid] < target)           // right half
    return search(arr, mid + 1, hi, target);
  return search(arr, lo, mid - 1, target);
}
```

- The initial call is `search(arr, 0, n - 1, target)`; the call replaces the loop's update of `lo` / `hi`.
- Viewed as divide and conquer: **divide** at the middle item, **conquer** by continuing in only **one** half, no **combine**.
- Since it has a single branch, it is also called **decrease and conquer**: T(n) = T(n/2) + 1.
- Extra memory: an O(log n) call stack (O(1) in the iterative version).

Binary Search = Balanced Binary Search Tree



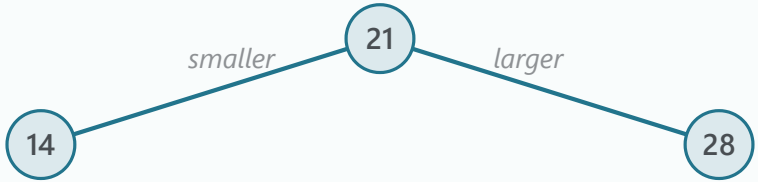
A sorted array of 15 items. Let us build the **best (balanced) binary search tree** from it.

Binary Search = Balanced Binary Search Tree



The middle item **21** becomes the root: it is always the **first** comparison of binary search.

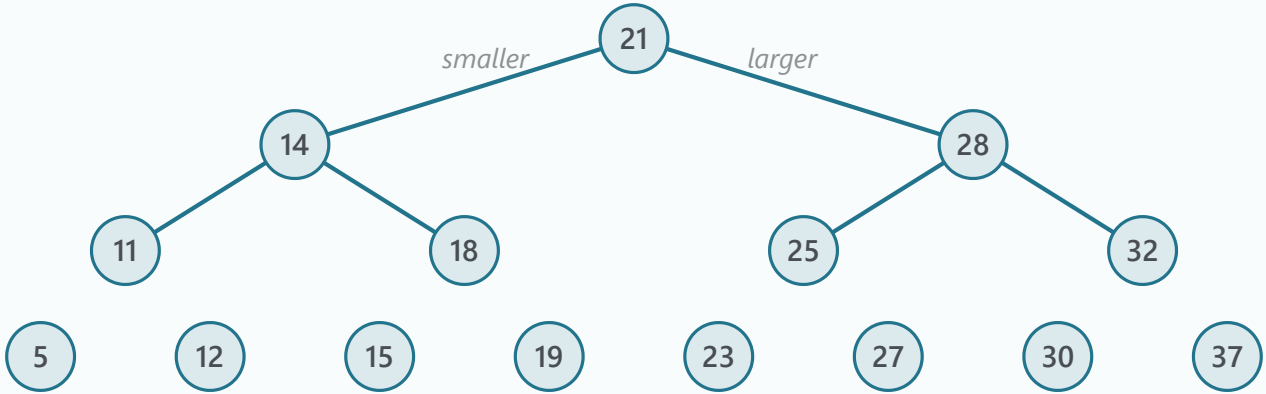
Binary Search = Balanced Binary Search Tree



5 11 12 15 18 19 23 25 27 30 32 37

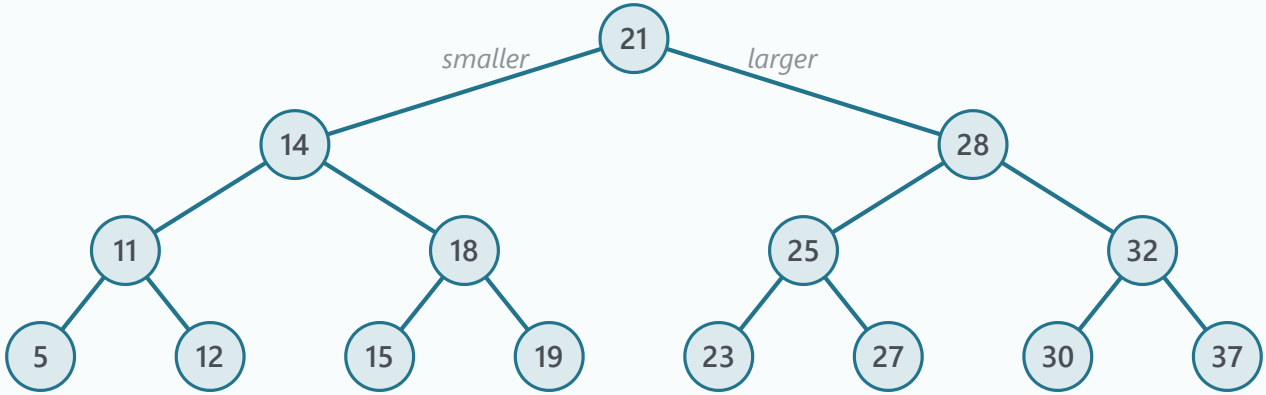
The middle of the left half is 14, of the right half 28: items **smaller** than 21 go left, **larger** ones right.

Binary Search = Balanced Binary Search Tree



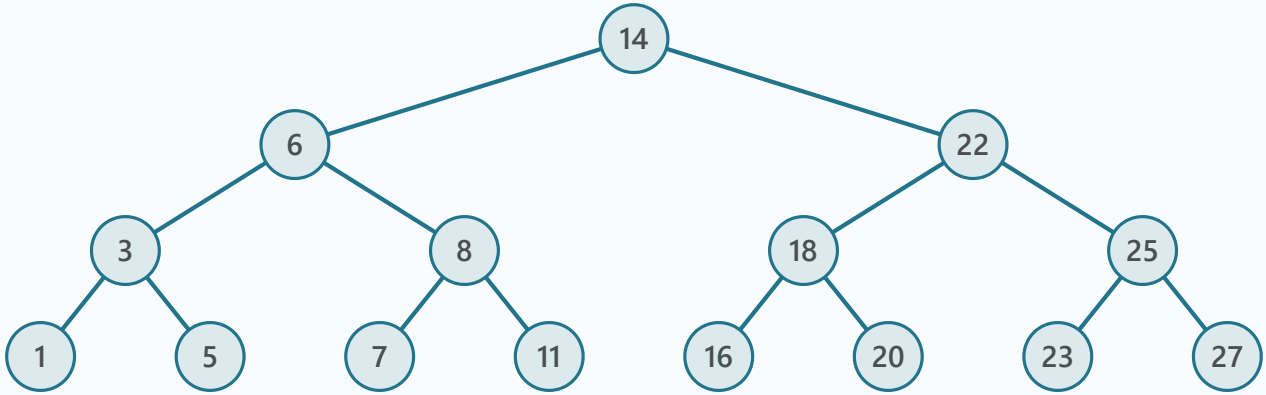
The middle of each subrange forms the next level: 11, 18, 25, 32.

Binary Search = Balanced Binary Search Tree



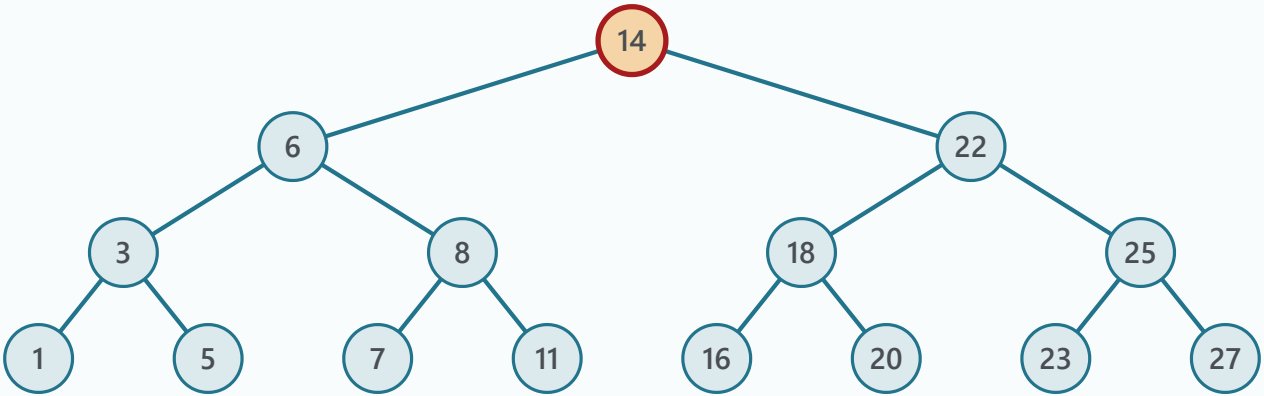
The remaining items are leaves. The tree has height 3 and 4 levels = $\lfloor \log_2 15 \rfloor + 1$: every search makes **at most 4** comparisons.

A Sorted Array Is an Implicit Tree



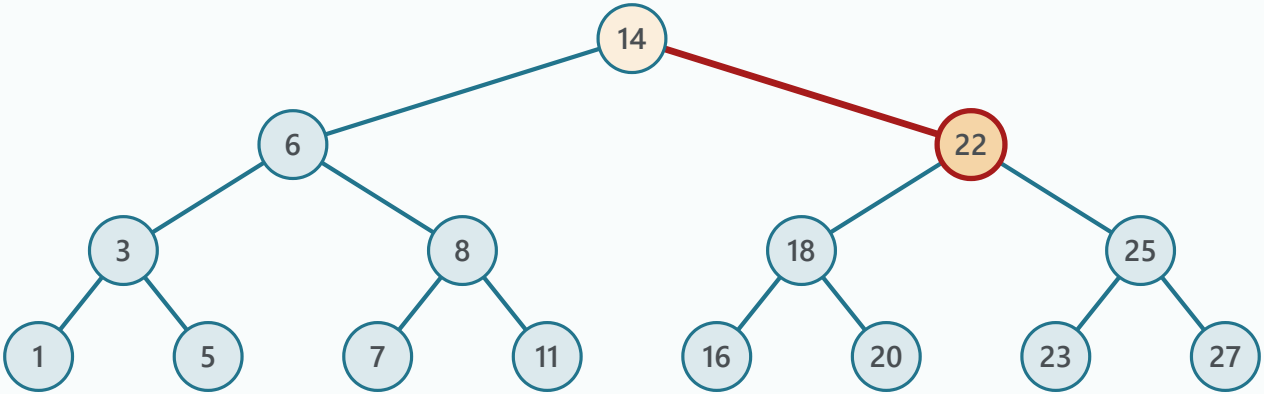
A second sorted array (15 items: 1, 3, 5, 6, …, 25, 27) can be viewed as a tree in the same way. The tree is **never built** in memory: the `mid` computation walks it implicitly. Let us search for 20.

A Sorted Array Is an Implicit Tree



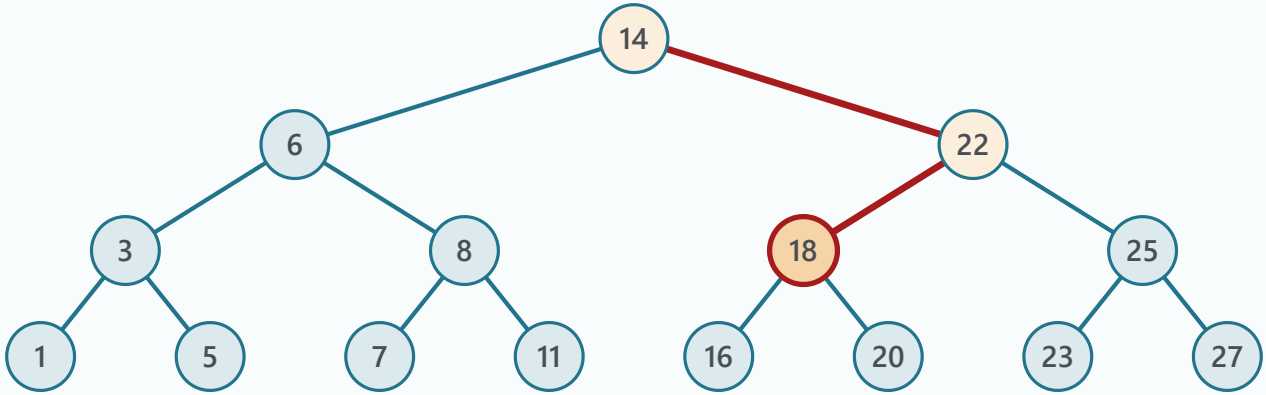
mid = 7 → arr[7] = 14; 20 > 14 → go right.

A Sorted Array Is an Implicit Tree



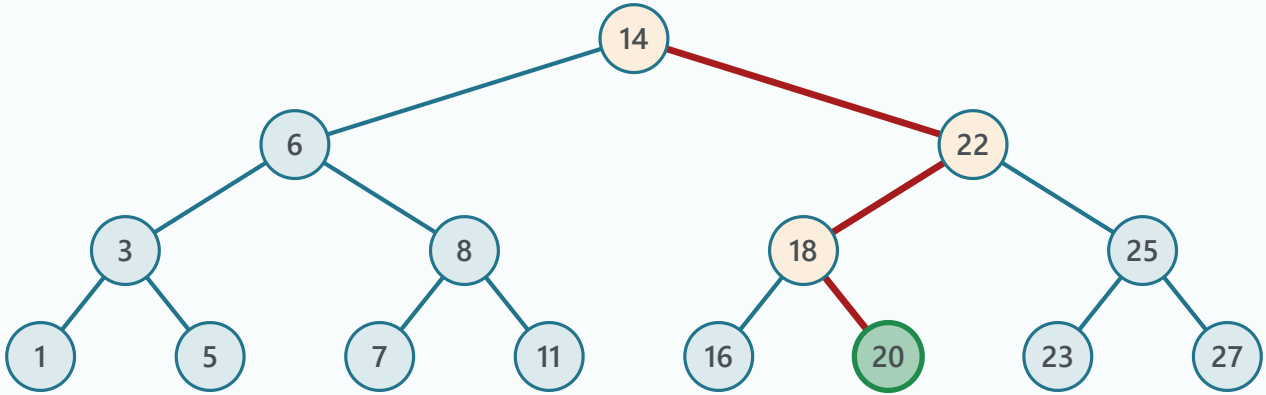
mid = 11 → arr[11] = 22; 20 < 22 → go left.

A Sorted Array Is an Implicit Tree



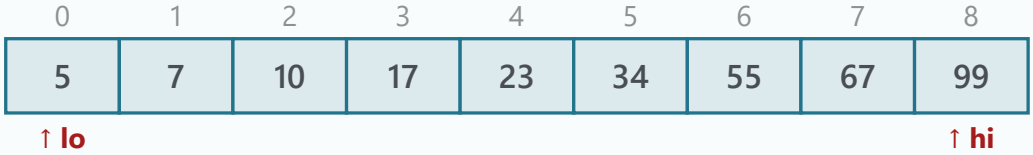
mid = 9 → arr[9] = 18; 20 > 18 → go right.

A Sorted Array Is an Implicit Tree



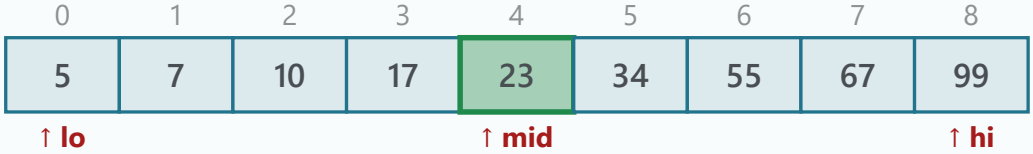
mid = 10 → arr[10] = 20: **found**. 4 comparisons from root to leaf; path 14 → 22 → 18 → 20.

Binary Search: Best and Worst Case



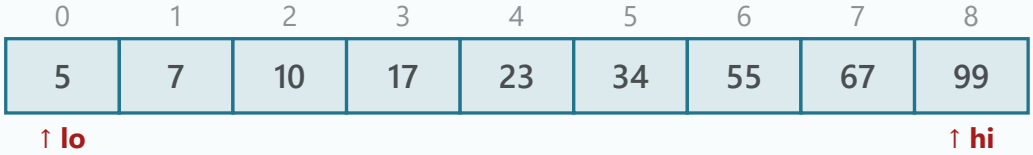
Best case: search for 23. lo = 0, hi = 8.

Binary Search: Best and Worst Case



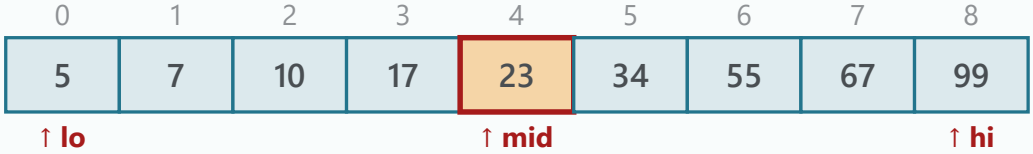
mid = (0 + 8) / 2 = 4 → arr[4] = 23 = target: **found in 1 step**, returns 4.

Binary Search: Best and Worst Case



Worst case: search for 54 (absent).

Binary Search: Best and Worst Case



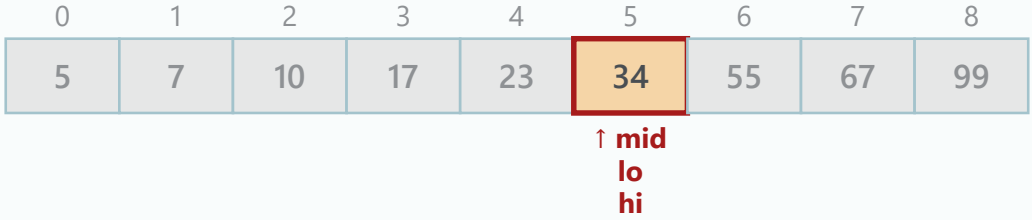
Step 1: mid = 4 → 23 < 54 → `lo = 5`; the left half is discarded.

Binary Search: Best and Worst Case



Step 2: mid = (5 + 8) / 2 = 6 → 55 > 54 → `hi = 5`.

Binary Search: Best and Worst Case



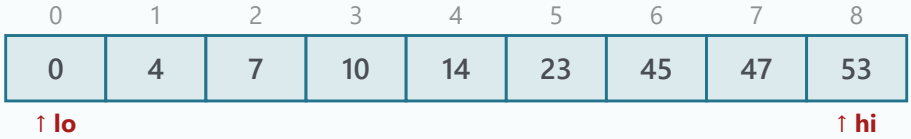
Step 3: mid = (5 + 5) / 2 = 5 → 34 < 54 → `lo = 6`.

Binary Search: Best and Worst Case



lo = 6 > hi = 5: the range is empty → **not found in 3 steps**, returns −1. If 54 were present it would lie between 34 and 55 (at index 6).

binarySearch(arr, 9, 47): Tracing the Code



Target 47; `lo` = 0, `hi` = 8.

Step 1 / 7

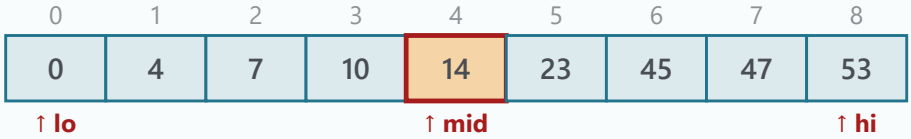
```
1  int binarySearch(int[] arr, int n, int target) {
▶    int lo = 0, hi = n - 1;
3    while (lo <= hi) {
4      int mid = (lo + hi) / 2;
5      if (target < arr[mid])
6        hi = mid - 1;      // left half
7      else if (target > arr[mid])
8        lo = mid + 1;      // right half
9      else
10       return mid;         // found
11   }
12   return -1;
13 }
```

VARIABLES

`target = 47` `lo = 0` `hi = 8` `mid = –`

OUTPUT

binarySearch(arr, 9, 47): Tracing the Code



lo ≤ hi → mid = (0 + 8) / 2 = 4; arr[4] = 14.

Step 2 / 7

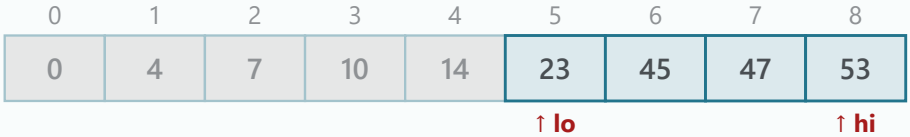
```
1  int binarySearch(int[] arr, int n, int target) {
2    int lo = 0, hi = n - 1;
▶    while (lo <= hi) {
▶      int mid = (lo + hi) / 2;
5      if (target < arr[mid])
6        hi = mid - 1;      // left half
7      else if (target > arr[mid])
8        lo = mid + 1;      // right half
9      else
10       return mid;         // found
11   }
12   return -1;
13 }
```

VARIABLES

`target = 47` `lo = 0` `hi = 8` `mid = 4`

OUTPUT

binarySearch(arr, 9, 47): Tracing the Code



47 > 14 → `lo = 5` ; the left part is discarded.

Step 3 / 7

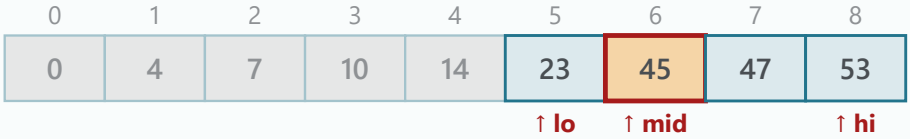
```
1  int binarySearch(int[] arr, int n, int target) {
2    int lo = 0, hi = n - 1;
3    while (lo <= hi) {
4      int mid = (lo + hi) / 2;
5      if (target < arr[mid])
6        hi = mid - 1;      // left half
▶      else if (target > arr[mid])
▶        lo = mid + 1;      // right half
9      else
10       return mid;         // found
11   }
12   return -1;
13 }
```

VARIABLES

target = 47 | lo = 5 | hi = 8 | mid = 4

OUTPUT

binarySearch(arr, 9, 47): Tracing the Code



lo ≤ hi → mid = (5 + 8) / 2 = 6; arr[6] = 45.

Step 4 / 7

```
1  int binarySearch(int[] arr, int n, int target) {
2    int lo = 0, hi = n - 1;
▶    while (lo <= hi) {
▶      int mid = (lo + hi) / 2;
5      if (target < arr[mid])
6        hi = mid - 1;      // left half
7      else if (target > arr[mid])
8        lo = mid + 1;      // right half
9      else
10       return mid;        // found
11   }
12   return -1;
13 }
```

VARIABLES

target = 47 | lo = 5 | hi = 8 | mid = 6

OUTPUT

binarySearch(arr, 9, 47): Tracing the Code



47 > 45 → `lo = 7`; the left part is discarded.

Step 5 / 7

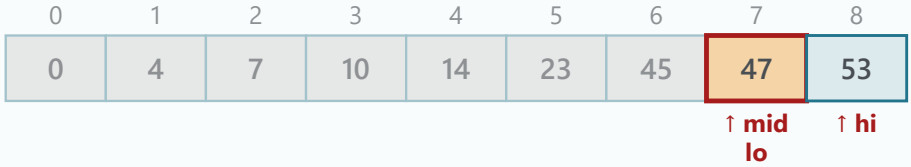
```
1  int binarySearch(int[] arr, int n, int target) {
2    int lo = 0, hi = n - 1;
3    while (lo <= hi) {
4      int mid = (lo + hi) / 2;
5      if (target < arr[mid])
6        hi = mid - 1;       // left half
▶      else if (target > arr[mid])
▶        lo = mid + 1;       // right half
9      else
10       return mid;          // found
11   }
12   return -1;
13 }
```

VARIABLES

`target = 47` `lo = 7` `hi = 8` `mid = 6`

OUTPUT

binarySearch(arr, 9, 47): Tracing the Code



lo ≤ hi → mid = (7 + 8) / 2 = 7; arr[7] = 47.

Step 6 / 7

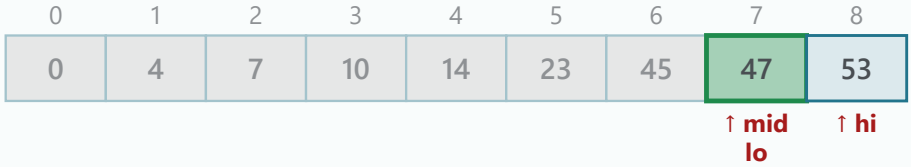
```
1  int binarySearch(int[] arr, int n, int target) {
2    int lo = 0, hi = n - 1;
▶    while (lo <= hi) {
▶      int mid = (lo + hi) / 2;
5      if (target < arr[mid])
6        hi = mid - 1;      // left half
7      else if (target > arr[mid])
8        lo = mid + 1;      // right half
9      else
10       return mid;         // found
11   }
12   return -1;
13 }
```

VARIABLES

target = 47 | lo = 7 | hi = 8 | mid = 7

OUTPUT

binarySearch(arr, 9, 47): Tracing the Code



47 == arr[7] → **returns 7**. 3 comparisons on 9 items (at most $\lfloor \log_2 9 \rfloor + 1 = 4$).

Step 7 / 7

```
1  int binarySearch(int[] arr, int n, int target) {
2    int lo = 0, hi = n - 1;
3    while (lo <= hi) {
4      int mid = (lo + hi) / 2;
5      if (target < arr[mid])
6        hi = mid - 1;       // left half
7      else if (target > arr[mid])
8        lo = mid + 1;       // right half
▶      else
▶        return mid;         // found
11   }
12   return -1;
13 }
```

VARIABLES

`target = 47` `lo = 7` `hi = 8` `mid = 7`

OUTPUT

```
binarySearch(arr, 9, 47) → 7
```

Binary Search: Complexity

$T(n) = T(n/2) + 1 \Rightarrow T(n) = O(\log n)$

- The range at least halves on every iteration: $n \to n/2 \to n/4 \to \dots \to 1$.
- After k iterations $n/2^k \le 1$: $k \approx \log_2 n$.
- Extra memory: **O(1)** for the iterative version, an O(log n) stack for the recursive one.

Case	Order
Best (first mid matches)	O(1)
Average	O(log n)
Worst / unsuccessful	O(log n)
Preprocessing (sorting)	O(n log n), once

When is it worth it? When the array is sorted once and searched many times: for k queries, linear is $O(k \cdot n)$, sort + binary is $O(n \log n + k \log n)$. The gap widens as k grows.

Binary Search: Implementation Details

Overflow

`(lo + hi) / 2` goes negative when lo + hi > $2^{31} - 1$. Safe forms: `lo + (hi - lo) / 2` or `(lo + hi) >>> 1`.

Updating the bounds

Writing `hi = mid` or `lo = mid` may not shrink the range → **infinite loop**. The middle item has already been checked, so the ±1 is required.

Duplicates

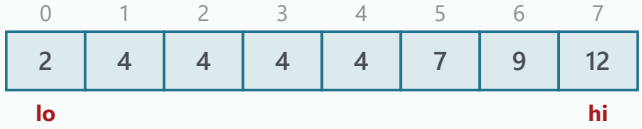
Returning on equality gives **some** match. For the first position, record the result on equality and continue left with `hi = mid - 1`.

Java library

`Arrays.binarySearch(arr, x)`: if not found it returns −(insertion point) − 1, which also tells where x can be inserted without breaking the order.

First Position Among Duplicates: firstIndex(arr, 4)

```
1  int firstIndex(int[] arr, int target) {
▶    int lo = 0, hi = arr.length - 1, result =
   -1;
3    while (lo <= hi) {
4      int mid = lo + (hi - lo) / 2;
5      if (arr[mid] == target) {
6        result = mid;      // record,
7        hi = mid - 1;    // go left
8      } else if (arr[mid] < target)
9        lo = mid + 1;
10     else
11       hi = mid - 1;
12   }
13   return result;
14 }
```



result = -1

arr = [2, 4, 4, 4, 4, 7, 9, 12], target = 4: it occurs four times. The goal is the index of the **first** 4. Start with result = −1.

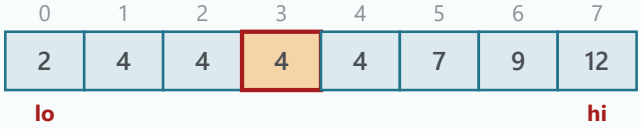
Step 1 / 5

VARIABLES

`lo = 0` `hi = 7` `mid = –` `result = -1`

First Position Among Duplicates: firstIndex(arr, 4)

```
1  int firstIndex(int[] arr, int target) {
2    int lo = 0, hi = arr.length - 1, result =
   -1;
3    while (lo <= hi) {
▶      int mid = lo + (hi - lo) / 2;
▶      if (arr[mid] == target) {
▶        result = mid;      // record,
▶        hi = mid - 1;    // go left
8      } else if (arr[mid] < target)
9        lo = mid + 1;
10     else
11       hi = mid - 1;
12   }
13   return result;
14 }
```



result = 3

mid = 0 + (7 − 0) / 2 = 3: arr[3] = 4 = target. Instead of returning, record `result = 3` and look for an earlier 4 **on the left** with `hi = 2`.

Step 2 / 5

VARIABLES

`lo = 0` `hi = 2` `mid = 3` `result = 3`

First Position Among Duplicates: firstIndex(arr, 4)

```
1  int firstIndex(int[] arr, int target) {
2    int lo = 0, hi = arr.length - 1, result =
   -1;
3    while (lo <= hi) {
▶      int mid = lo + (hi - lo) / 2;
▶      if (arr[mid] == target) {
▶        result = mid;      // record,
▶        hi = mid - 1;    // go left
8      } else if (arr[mid] < target)
9        lo = mid + 1;
10     else
11       hi = mid - 1;
12   }
13   return result;
14 }
```

0	1	2	3	4	5	6	7
2	4	4	4	4	7	9	12
lo		hi					

result = 1

mid = 0 + (2 − 0) / 2 = 1: arr[1] = 4 = target. Instead of returning, record `result = 1` and look for an earlier 4 **on the left** with `hi = 0`.

Step 3 / 5

VARIABLES

`lo = 0` `hi = 0` `mid = 1` `result = 1`

First Position Among Duplicates: firstIndex(arr, 4)

```
1  int firstIndex(int[] arr, int target) {
2    int lo = 0, hi = arr.length - 1, result =
   -1;
3    while (lo <= hi) {
▶      int mid = lo + (hi - lo) / 2;
▶      if (arr[mid] == target) {
6        result = mid;      // record,
7        hi = mid - 1;    // go left
▶      } else if (arr[mid] < target)
▶        lo = mid + 1;
10     else
11       hi = mid - 1;
12   }
13   return result;
14 }
```

0	1	2	3	4	5	6	7
2	4	4	4	4	7	9	12

lo/hi

result = 1

mid = 0: arr[0] = 2 < 4 → `lo = 1`.

Step 4 / 5

VARIABLES

`lo = 1` `hi = 0` `mid = 0` `result = 1`

First Position Among Duplicates: firstIndex(arr, 4)

```
1  int firstIndex(int[] arr, int target) {
2    int lo = 0, hi = arr.length - 1, result =
   -1;
▶    while (lo <= hi) {
4      int mid = lo + (hi - lo) / 2;
5      if (arr[mid] == target) {
6        result = mid;       // record,
7        hi = mid - 1;     // go left
8      } else if (arr[mid] < target)
9        lo = mid + 1;
10     else
11       hi = mid - 1;
12   }
▶    return result;
14 }
```

0	1	2	3	4	5	6	7
2	4	4	4	4	7	9	12
	hi				lo		

result = 1

lo = 1 > hi = 0: the range is empty and the loop ends. `return 1` : the first 4 is at index 1. Still O(log n): 3 probes.

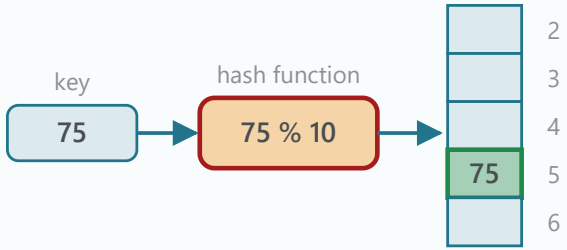
Step 5 / 5

VARIABLES

`lo = 1` `hi = 0` `mid = –` `result = 1`

Hash Table Search

- Like a **key that fits its lock**.
- The data consists of **keys** and their associated **values**.
- A **hash function** turns the key into a table index (the hash value), which points to the data's place in the table.
- The search looks in the slot for the key's hash value (the **bucket**).
- If the bucket is not empty, the key is compared with the keys in it; a match ends the search.



Other searches find a position by **comparing** items; hashing **computes** the position. So the average time does not depend on n: O(1).

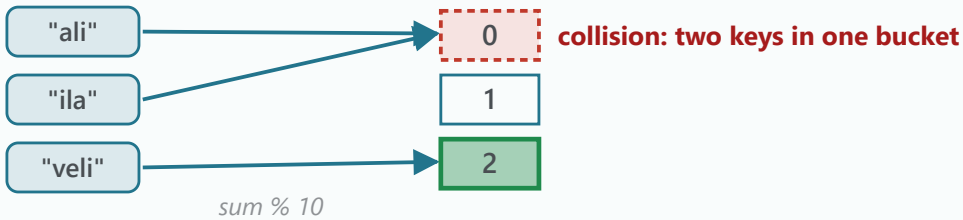
Hashing: Pros and Cons

Advantages

+ Very fast search on average: **O(1)**.
+ The data **need not be sorted** in advance.
+ Ideal for searching large data sets.
+ With a well-chosen hash function, the target is accessed directly.

Disadvantages

– The hash function can cause **collisions**: two different keys with the same hash value.
– Collisions need extra steps; in the worst case **O(n)**.
– Needs extra memory (empty buckets); no ordered traversal, minimum or range queries.



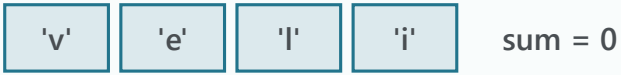
The Hash Function

```
int hash(String key,
                  int m) {
  int sum = 0;
  for (char k : key.toCharArray())
    sum += k;          // Unicode code
  return sum % m;      // m: table size
}
```

key	character codes	sum % 10
"ali"	97 + 108 + 105 = 310	0
"ila"	105 + 108 + 97 = 310	0
"veli"	118 + 101 + 108 + 105 = 432	2

- This function adds up the Unicode codes of the characters (the numeric value of a Java `char`) and takes the result **mod** the table size.
- Simple but weak: words with the same letters (**"ali"** and **"ila"**) always **collide**.
- A good hash function is fast to compute and spreads keys **evenly** over the buckets; Java's `String.hashCode()` weights each position: $\Sigma\, s[i] \cdot 31^{n-1-i}$.

hash("veli", 10): Tracing the Code



key = "veli", m = 10 buckets. `sum = 0` .

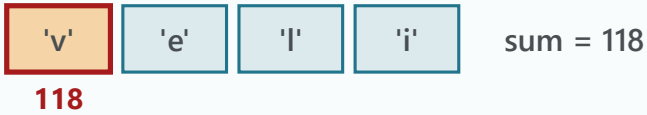
Step 1 / 6

```
1 int hash(String key,
2                   int m) {
▶   int sum = 0;
4   for (char k : key.toCharArray())
5     sum += k;         // Unicode code
6   return sum % m;     // m: table size
7 }
```

VARIABLES

key = "veli" | m = 10 | k = – | sum = 0

hash("veli", 10): Tracing the Code



k = 'v': add its Unicode code **118** → sum = 118.

Step 2 / 6

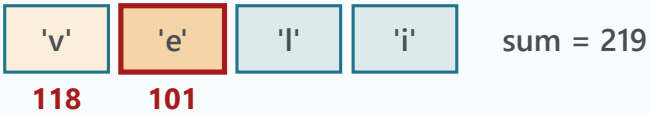
```
1 int hash(String key,
2                   int m) {
3   int sum = 0;
▶   for (char k : key.toCharArray())
▶     sum += k;          // Unicode code
6   return sum % m;     // m: table size
7 }
```

VARIABLES

key = "veli" m = 10

k = 'v' sum = 118

hash("veli", 10): Tracing the Code



k = 'e': add its Unicode code **101** → sum = 219.

Step 3 / 6

```
1 int hash(String key,
2                   int m) {
3   int sum = 0;
▶   for (char k : key.toCharArray())
▶     sum += k;          // Unicode code
6   return sum % m;     // m: table size
7 }
```

VARIABLES

key = "veli" m = 10 k = 'e' sum = 219

hash("veli", 10): Tracing the Code



k = 'l': add its Unicode code **108** → sum = 327.

Step 4 / 6

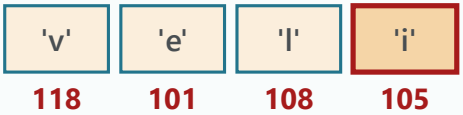
```
1 int hash(String key,
2                   int m) {
3   int sum = 0;
▶   for (char k : key.toCharArray())
▶     sum += k;         // Unicode code
6   return sum % m;     // m: table size
7 }
```

VARIABLES

key = "veli" m = 10

k = 'l' sum = 327

hash("veli", 10): Tracing the Code



k = 'i': add its Unicode code **105** → sum = 432.

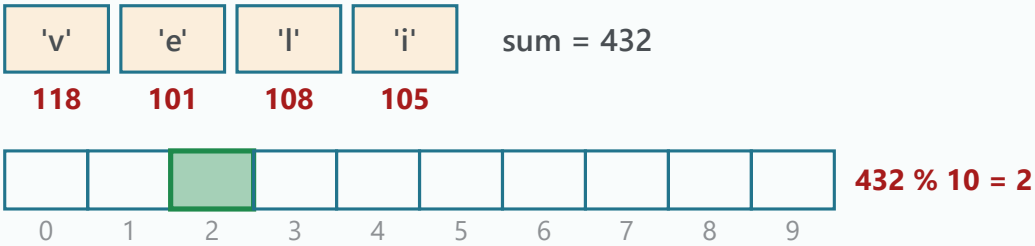
Step 5 / 6

```
1 int hash(String key,
2                   int m) {
3   int sum = 0;
▶   for (char k : key.toCharArray())
▶     sum += k;          // Unicode code
6   return sum % m;     // m: table size
7 }
```

VARIABLES

key = "veli" m = 10 k = 'i' sum = 432

hash("veli", 10): Tracing the Code



432 % 10 = **2**: "veli" lands in bucket 2. "ali" and "ila" have the same letters, so both sum to 310 and collide in bucket 0.

Step 6 / 6

```
1 int hash(String key,
2                   int m) {
3   int sum = 0;
4   for (char k : key.toCharArray())
5     sum += k;         // Unicode code
▶   return sum % m;     // m: table size
7 }
```

VARIABLES

`key = "veli"` `m = 10` `k = –` `sum = 432` `result = 2`

Hash Table Search (Linear Probing)

```
int hashSearch(Integer[] table, int key) {
  int m = table.length;     // m > 0
  int idx = Math.floorMod(key, m);   // hash
  while (table[idx] != null) {
    if (table[idx] == key)
      return idx;           // found
    idx = (idx + 1) % m; // next bucket
  }
  return -1;                 // empty: absent
}
```

- **Data structures:** `table` stores the keys (key–value pairs in general); the hash function turns a key into an index.
- The `while` loop runs until `table[idx]` is empty (null).
- Each iteration compares the key in the bucket with the target; on a match it returns the position (the value, in a key–value table).
- `floorMod` : the index stays in 0..m−1 even for negative keys.
- With no match, the **next** bucket is checked; `% m` wraps around from the end to the start (circular search).
- Reaching an empty bucket means the key is absent.

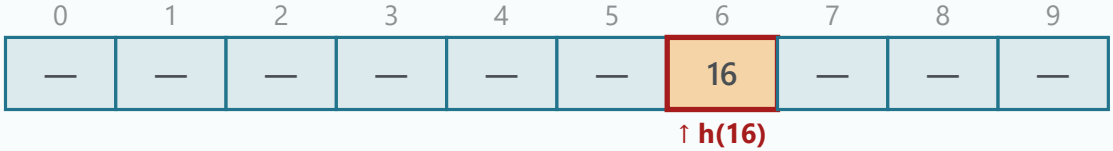
This collision strategy is called **open addressing** (linear probing). In the examples the keys are integers and h(x) = x % 10.

Insertion with Linear Probing



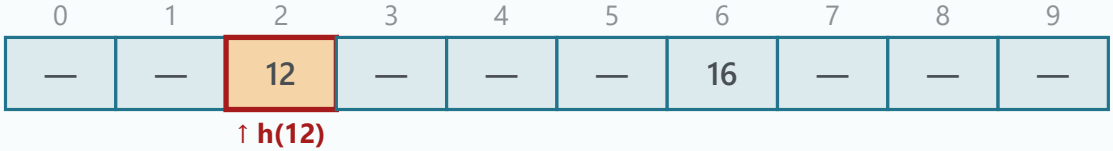
Empty table (m = 10). Keys are inserted in order: 16, 12, 45, 29, 6, 122, 5, 58, 75; h(x) = x % 10.

Insertion with Linear Probing



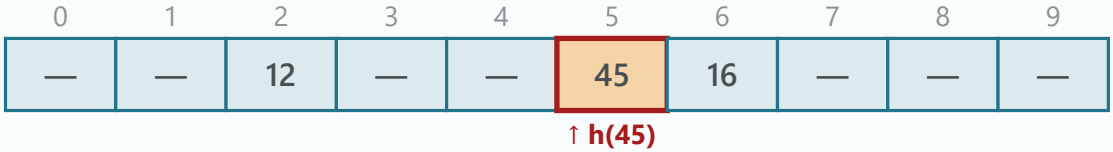
h(16) = 6: bucket empty → 16 goes here.

Insertion with Linear Probing



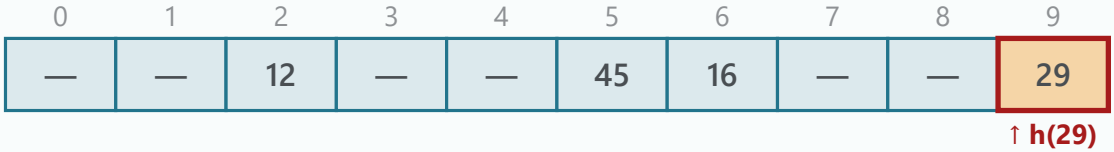
h(12) = 2: bucket empty → 12 goes here.

Insertion with Linear Probing



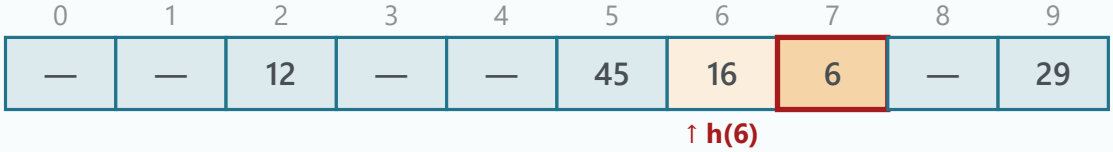
h(45) = 5: bucket empty → 45 goes here.

Insertion with Linear Probing



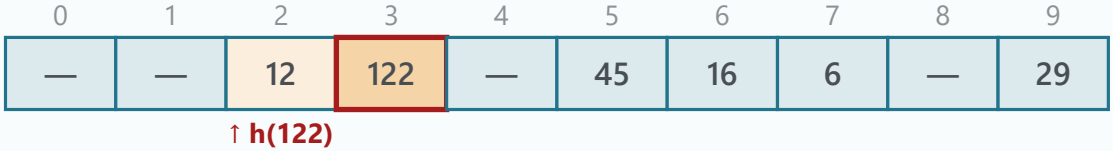
h(29) = 9: bucket empty → 29 goes here.

Insertion with Linear Probing



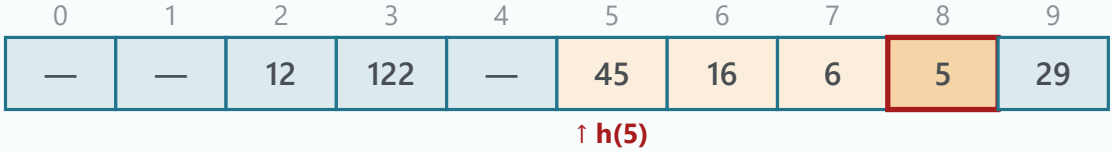
h(6) = 6 is full → probe 7; first empty bucket **7**. (2 probes)

Insertion with Linear Probing



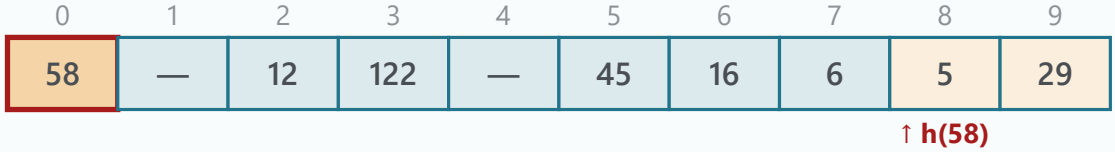
h(122) = 2 is full → probe 3; first empty bucket **3**. (2 probes)

Insertion with Linear Probing



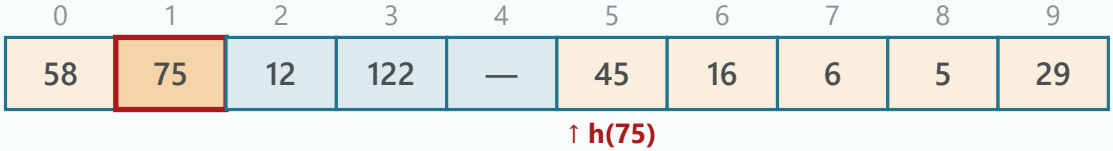
h(5) = 5 is full → probe 6, 7, 8; first empty bucket **8**. (4 probes)

Insertion with Linear Probing



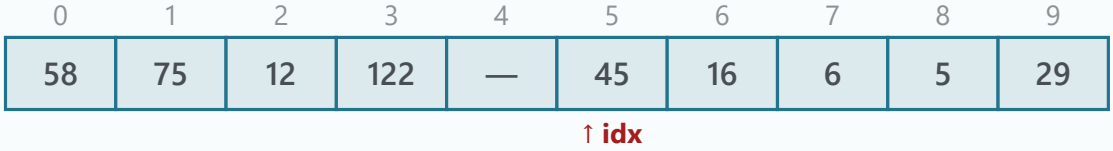
h(58) = 8 is full → probe 9, 0; first empty bucket **0**. (3 probes)

Insertion with Linear Probing



h(75) = 5 is full → probe 6, 7, 8, 9, 0, 1; first empty bucket **1**. (7 probes) The full buckets form a contiguous **cluster** (5, 6, …, 9, 0, …, 3; only bucket 4 is empty): primary clustering.

hashSearch(table, 75): Tracing the Code



m = 10, `idx` = 75 % 10 = 5.

Step 1 / 8

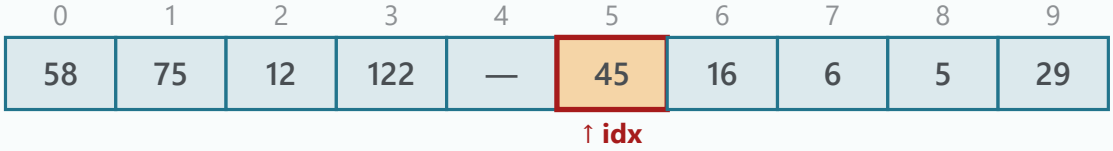
```
1 int hashSearch(Integer[] table, int key) {
▶   int m = table.length;     // m > 0
▶   int idx = Math.floorMod(key, m);   // hash
4   while (table[idx] != null) {
5     if (table[idx] == key)
6       return idx;           // found
7     idx = (idx + 1) % m; // next bucket
8   }
9   return -1;                 // empty: absent
10 }
```

VARIABLES

`key = 75` `m = 10` `idx = 5`

OUTPUT

hashSearch(table, 75): Tracing the Code



table[5] = 45 ≠ 75 → idx = (5 + 1) % 10 = 6.

Step 2 / 8

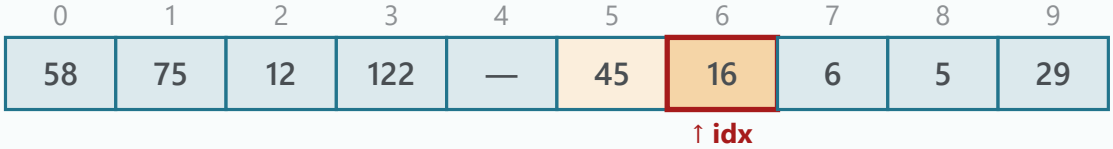
```
1  int hashSearch(Integer[] table, int key) {
2    int m = table.length;     // m > 0
3    int idx = Math.floorMod(key, m);   // hash
▶    while (table[idx] != null) {
▶      if (table[idx] == key)
6        return idx;           // found
▶      idx = (idx + 1) % m; // next bucket
8    }
9    return -1;                 // empty: absent
10 }
```

VARIABLES

`key = 75` `m = 10` `idx = 5`

OUTPUT

hashSearch(table, 75): Tracing the Code



table[6] = 16 ≠ 75 → idx = (6 + 1) % 10 = 7.

Step 3 / 8

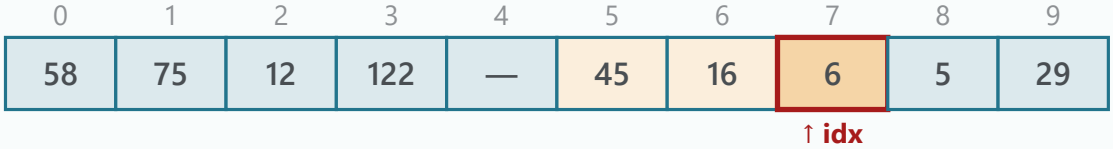
```
1  int hashSearch(Integer[] table, int key) {
2    int m = table.length;      // m > 0
3    int idx = Math.floorMod(key, m);    // hash
▶    while (table[idx] != null) {
▶      if (table[idx] == key)
6        return idx;           // found
▶      idx = (idx + 1) % m; // next bucket
8    }
9    return -1;                  // empty: absent
10 }
```

VARIABLES

key = 75 | m = 10 | idx = 6

OUTPUT

hashSearch(table, 75): Tracing the Code



table[7] = 6 ≠ 75 → idx = (7 + 1) % 10 = 8.

Step 4 / 8

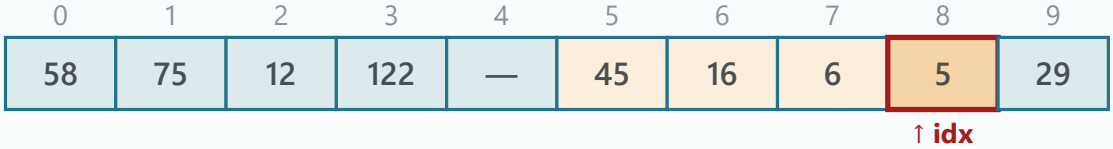
```
1  int hashSearch(Integer[] table, int key) {
2    int m = table.length;     // m > 0
3    int idx = Math.floorMod(key, m);   // hash
▶    while (table[idx] != null) {
▶      if (table[idx] == key)
6        return idx;           // found
▶      idx = (idx + 1) % m; // next bucket
8    }
9    return -1;                 // empty: absent
10 }
```

VARIABLES

`key = 75` `m = 10` `idx = 7`

OUTPUT

hashSearch(table, 75): Tracing the Code



table[8] = 5 ≠ 75 → idx = (8 + 1) % 10 = 9.

Step 5 / 8

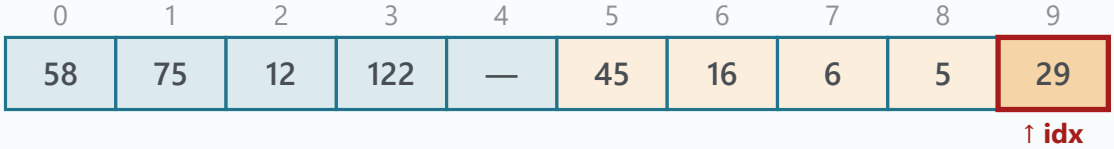
```
1  int hashSearch(Integer[] table, int key) {
2    int m = table.length;      // m > 0
3    int idx = Math.floorMod(key, m);    // hash
▶    while (table[idx] != null) {
▶      if (table[idx] == key)
6        return idx;           // found
▶      idx = (idx + 1) % m; // next bucket
8    }
9    return -1;                  // empty: absent
10 }
```

VARIABLES

key = 75 | m = 10 | idx = 8

OUTPUT

hashSearch(table, 75): Tracing the Code



table[9] = 29 ≠ 75 → idx = (9 + 1) % 10 = 0. Wrapped around to the start.

Step 6 / 8

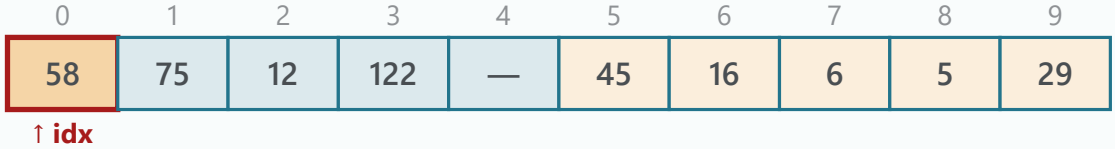
```
1  int hashSearch(Integer[] table, int key) {
2    int m = table.length;     // m > 0
3    int idx = Math.floorMod(key, m);   // hash
▶    while (table[idx] != null) {
▶      if (table[idx] == key)
6        return idx;           // found
▶      idx = (idx + 1) % m; // next bucket
8    }
9    return -1;                 // empty: absent
10 }
```

VARIABLES

`key = 75` `m = 10` `idx = 9`

OUTPUT

hashSearch(table, 75): Tracing the Code



table[0] = 58 ≠ 75 → idx = (0 + 1) % 10 = 1.

Step 7 / 8

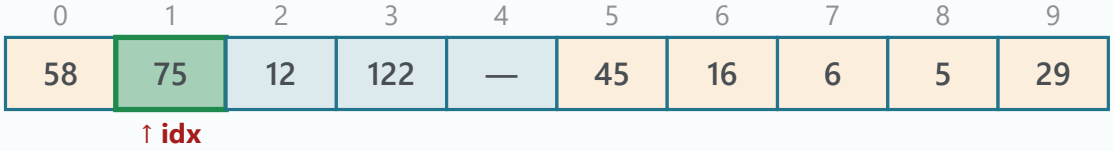
```
1  int hashSearch(Integer[] table, int key) {
2    int m = table.length;     // m > 0
3    int idx = Math.floorMod(key, m);   // hash
▶    while (table[idx] != null) {
▶      if (table[idx] == key)
6        return idx;           // found
▶      idx = (idx + 1) % m; // next bucket
8    }
9    return -1;                 // empty: absent
10 }
```

VARIABLES

key = 75 | m = 10 | idx = 0

OUTPUT

hashSearch(table, 75): Tracing the Code



table[1] = 75 → **returns 1**. 7 probes: 75 was reached at the end of the cluster.

Step 8 / 8

```
1  int hashSearch(Integer[] table, int key) {
2    int m = table.length;     // m > 0
3    int idx = Math.floorMod(key, m);   // hash
▶    while (table[idx] != null) {
▶      if (table[idx] == key)
▶        return idx;           // found
7      idx = (idx + 1) % m; // next bucket
8    }
9    return -1;                 // empty: absent
10 }
```

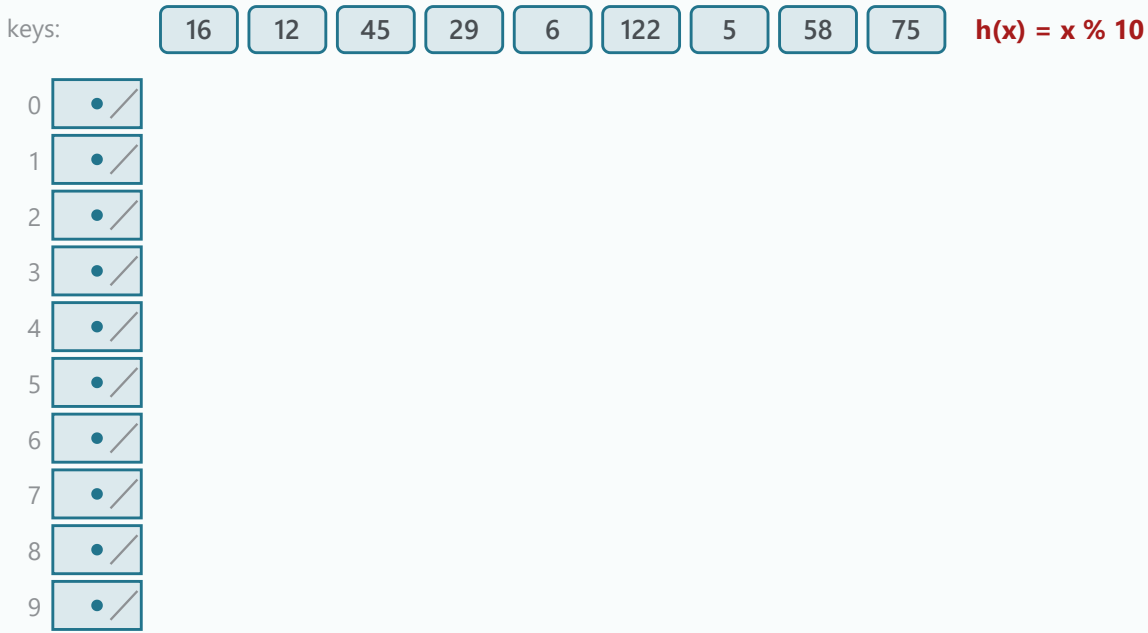
VARIABLES

`key = 75` `m = 10` `idx = 1`

OUTPUT

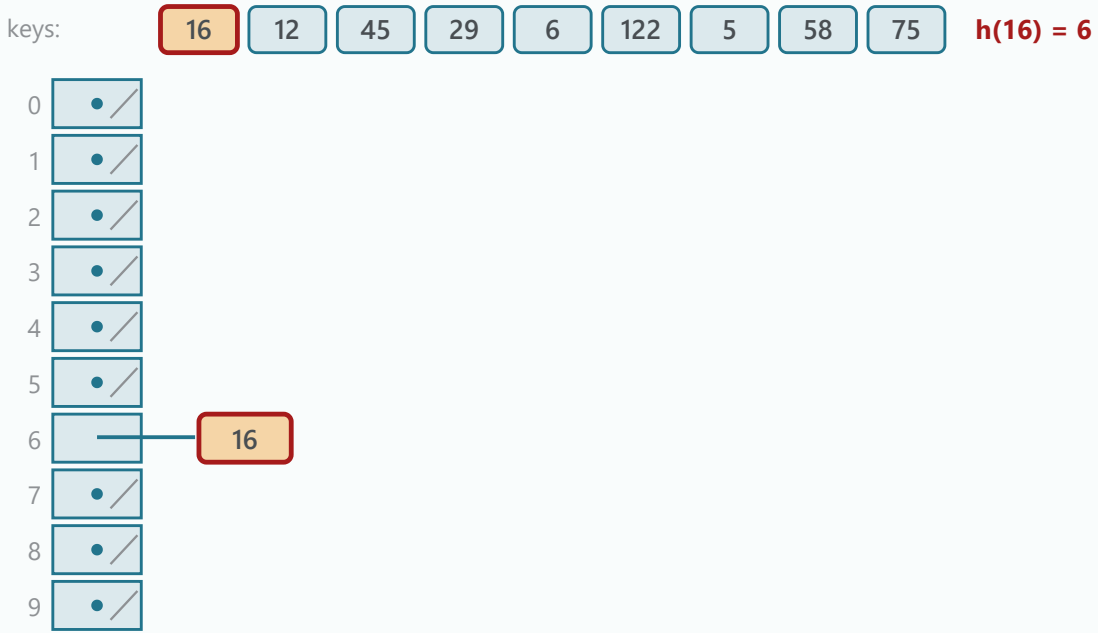
```
hashSearch(table, 75) → 1
```

Insertion and Search with Chaining



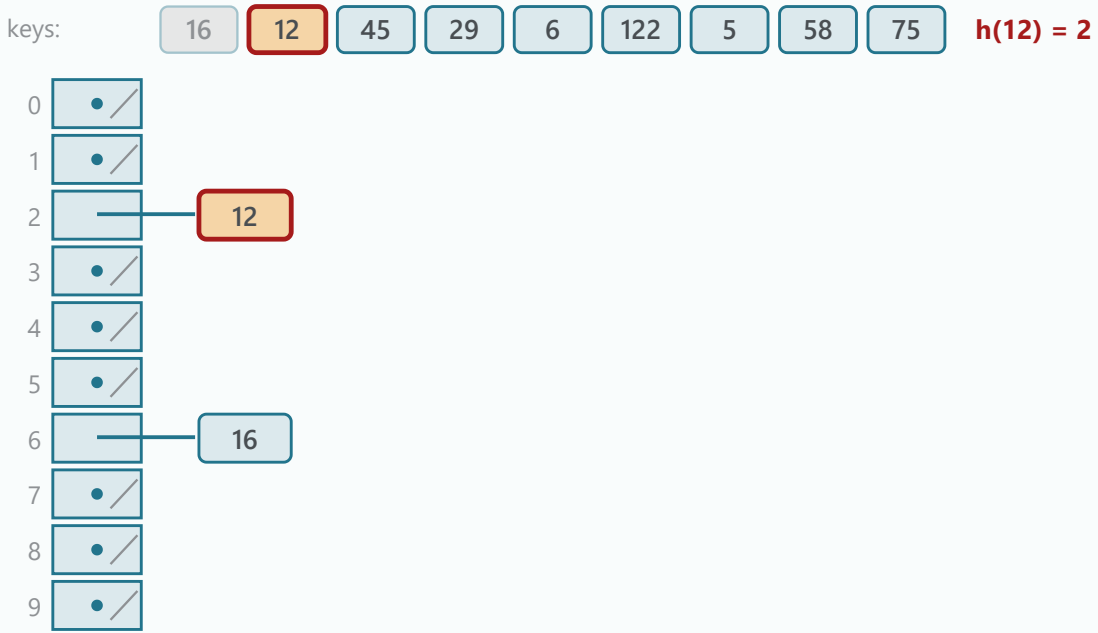
Separate chaining: each bucket holds the head of a linked list. Same keys, h(x) = x % 10.

Insertion and Search with Chaining



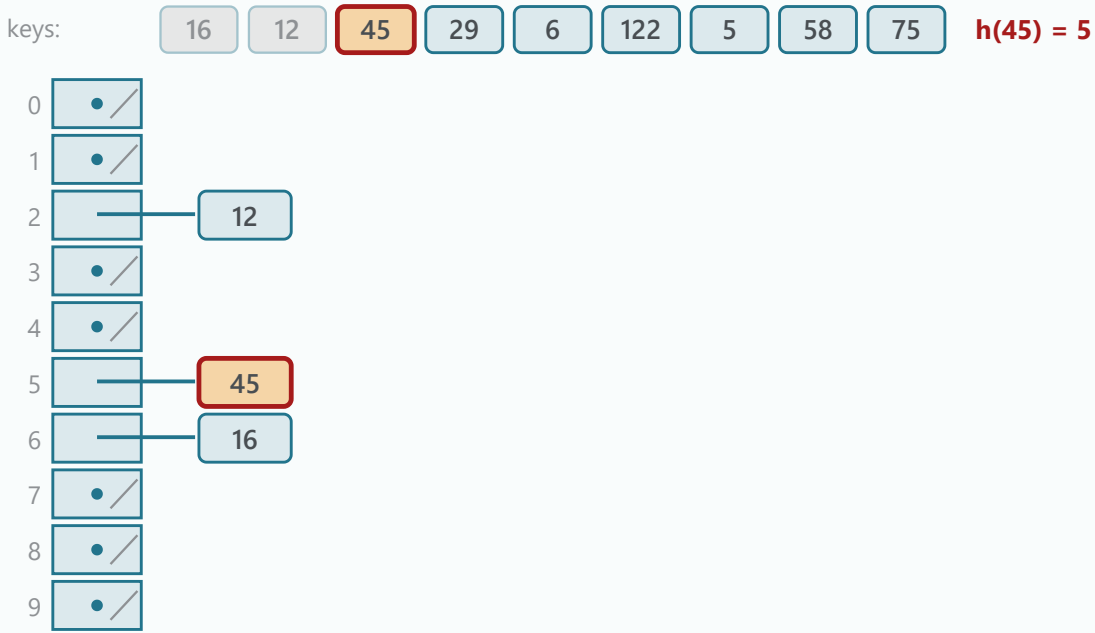
h(16) = 6 → bucket 6 is empty; 16 is appended to the **end** of the list.

Insertion and Search with Chaining



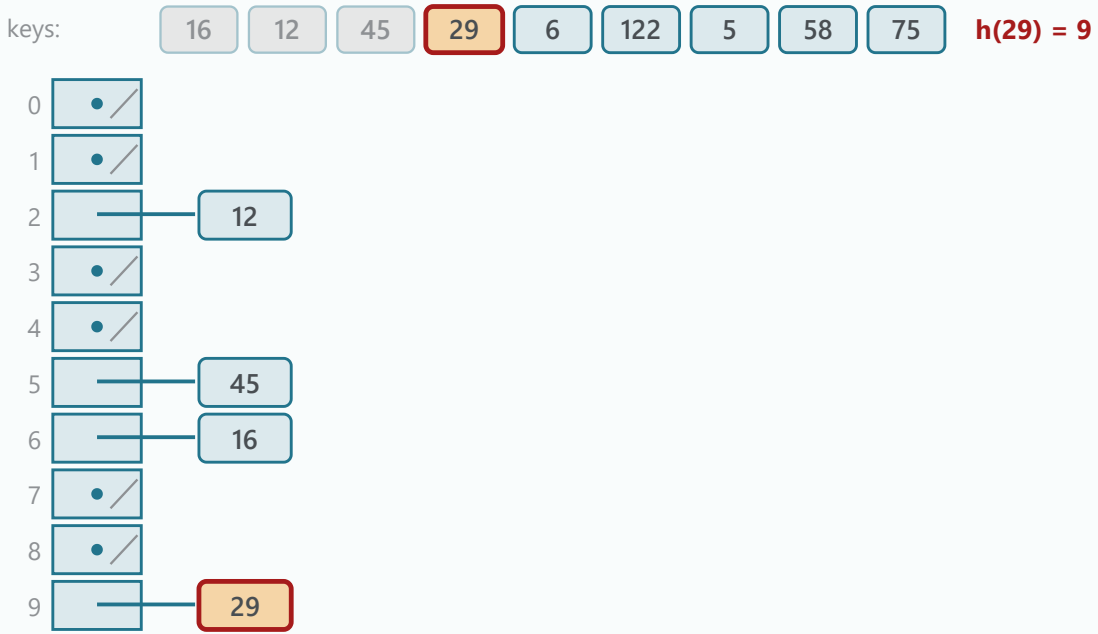
h(12) = 2 → bucket 2 is empty; 12 is appended to the **end** of the list.

Insertion and Search with Chaining



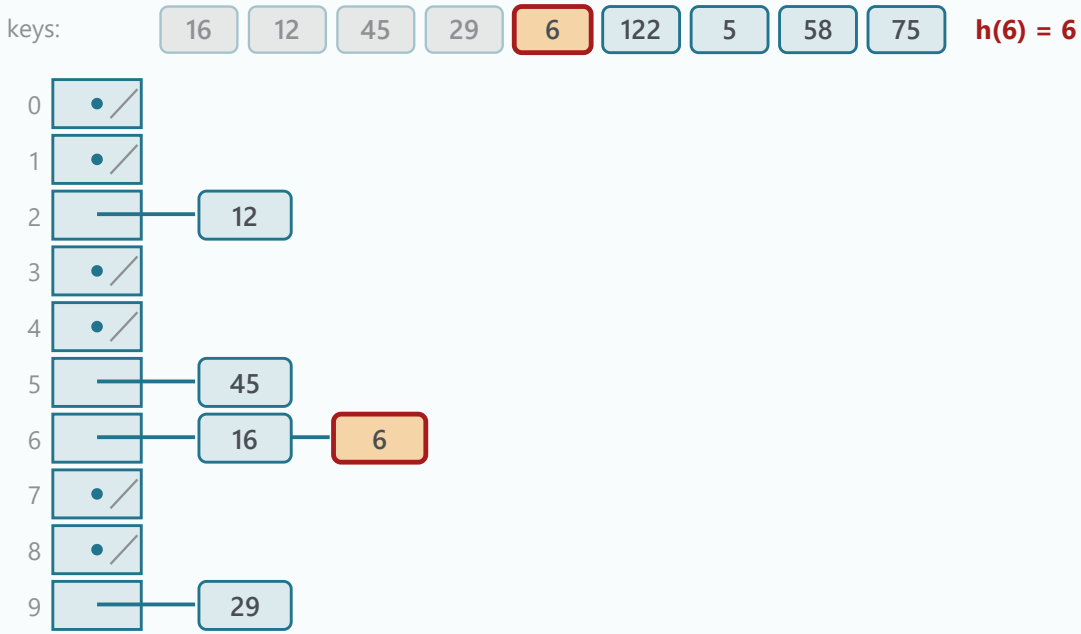
h(45) = 5 → bucket 5 is empty; 45 is appended to the **end** of the list.

Insertion and Search with Chaining



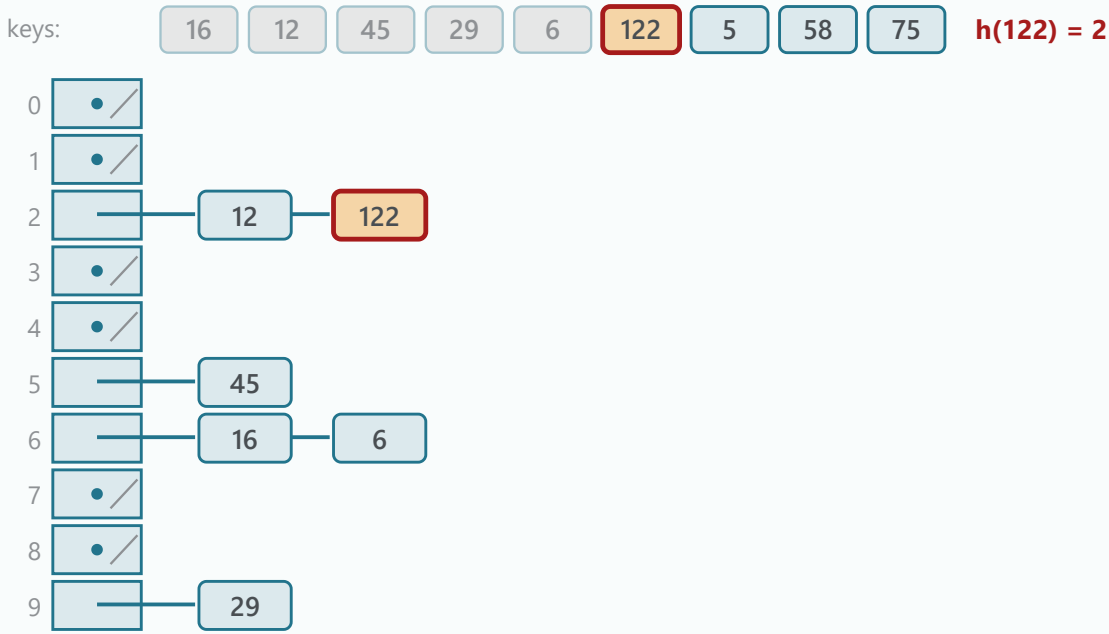
h(29) = 9 → bucket 9 is empty; 29 is appended to the **end** of the list.

Insertion and Search with Chaining



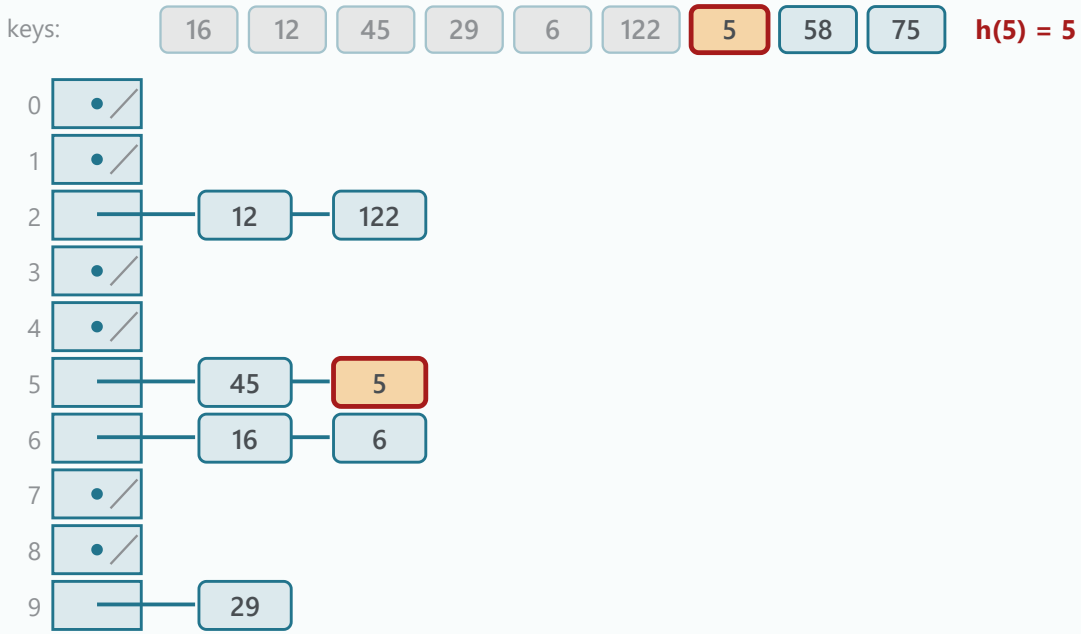
h(6) = 6 → bucket 6 is occupied (collision); 6 is appended to the **end** of the list.

Insertion and Search with Chaining



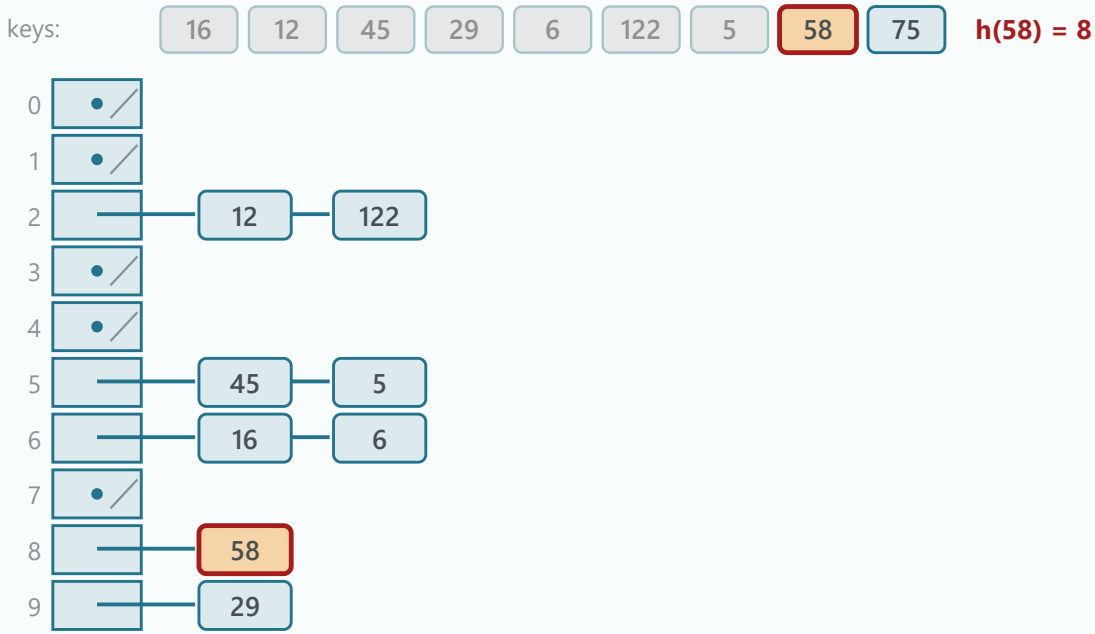
h(122) = 2 → bucket 2 is occupied (collision); 122 is appended to the **end** of the list.

Insertion and Search with Chaining



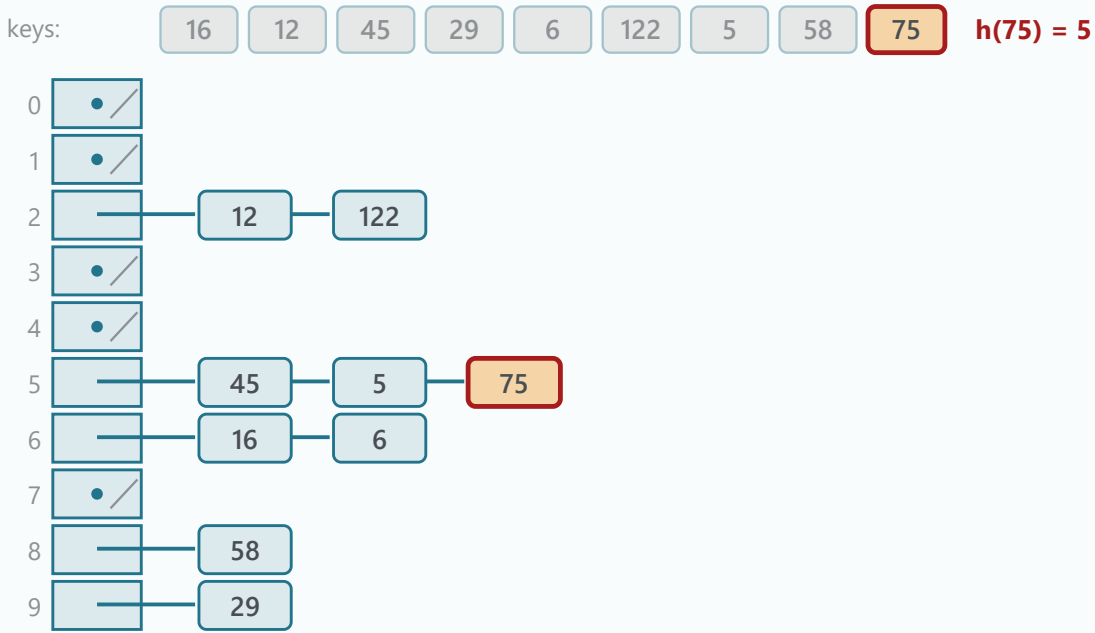
h(5) = 5 → bucket 5 is occupied (collision); 5 is appended to the **end** of the list.

Insertion and Search with Chaining



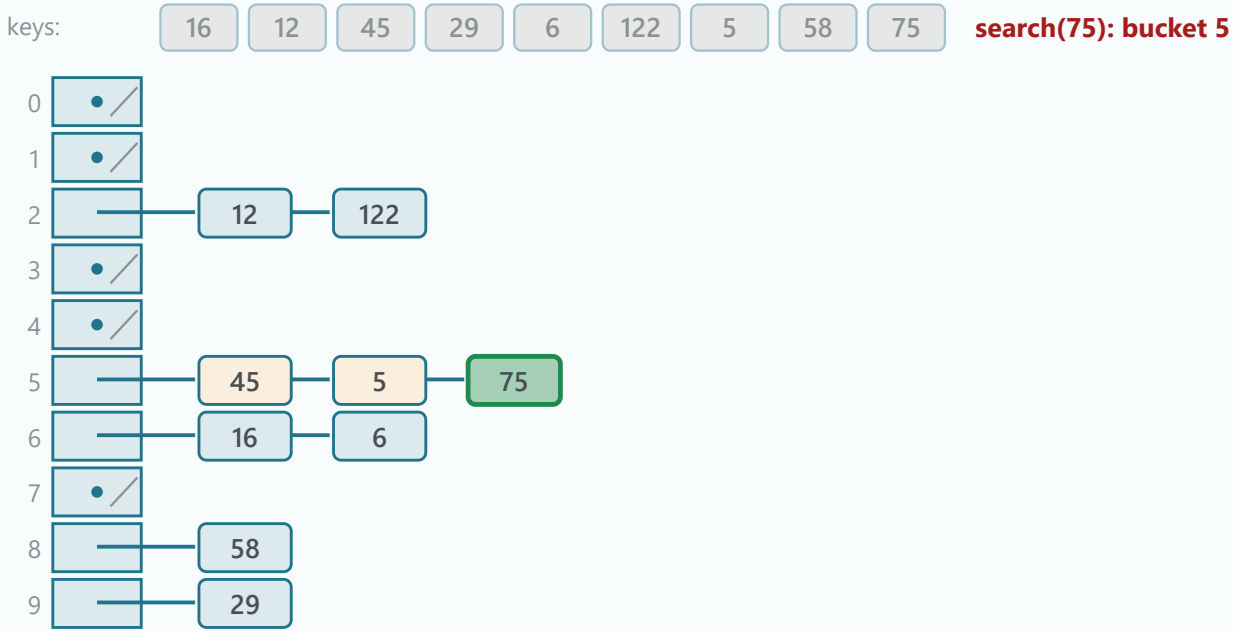
h(58) = 8 → bucket 8 is empty; 58 is appended to the **end** of the list.

Insertion and Search with Chaining



h(75) = 5 → bucket 5 is occupied (collision); 75 is appended to the **end** of the list.

Insertion and Search with Chaining



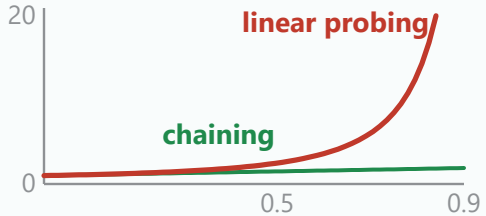
Search for 75: walk the list of bucket 5: 45 ≠ 75, 5 ≠ 75, 75 = 75 → found (3 comparisons). Fewer than the 7 probes of linear probing.

Load Factor and Complexity

`load factor α = n / m  (n: number of keys, m: number of buckets)`

Method	Successful search (avg.)	Unsuccessful search (avg.)	α = 0.5	α = 0.9
Chaining	≈ 1 + α/2	≈ 1 + α	1.25 / 1.5	1.45 / 1.9
Linear probing	≈ ½ (1 + 1/(1 − α))	≈ ½ (1 + 1/(1 − α)^2)	1.5 / 2.5	5.5 / 50.5

- If α is kept bounded (e.g. doubling the table and rehashing once α exceeds 0.75), the average search is **O(1)**.
- Worst case **O(n)**: if every key lands in the same bucket (a bad hash function), the search degenerates into linear search.
- With linear probing, clustering drives the cost up fast as α → 1; if the table is completely full, an unsuccessful search **never terminates**.



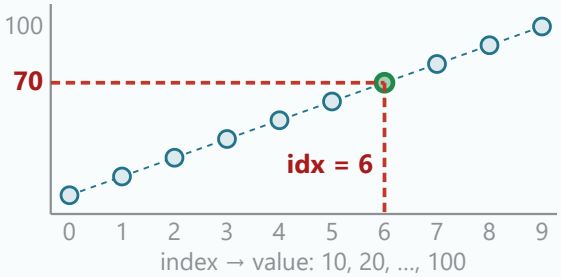
The formulas are Knuth's approximations under the uniform hashing assumption.

Interpolation Search

- Needs **sorted** data.
- It assumes the values are spread **evenly** over the list: after the first value, consecutive items differ by about the same amount.
- Under this assumption, the position of the target is **estimated** from the **smallest and largest** items.
- The estimated position is checked; if wrong, the search continues in a narrower range (to the right or left).
- Like opening a phone book near the end, not the middle, to find "Yılmaz".

```
idx = lo + (target - arr[lo]) · (hi - lo) / (arr[hi] - arr[lo])
```

Example: 70 in 10, 20, ..., 100: idx = 0 + (70 − 10) · 9 / 90 = 6 → arr[6] = 70. With a uniform distribution it is found in **a single probe**.



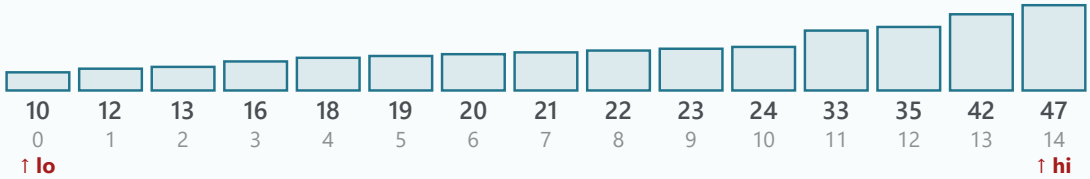
Binary search always looks at the middle (ratio ½); interpolation search computes the ratio from the values.

The Interpolation Search Algorithm

```
int interpolationSearch(int[] arr, int n,
                        int target) {
  int lo = 0, hi = n - 1;
  while (lo <= hi && target >= arr[lo]
                  && target <= arr[hi]) {
    if (arr[hi] == arr[lo]) return lo; // avoid /0
    long num = (long) (target - arr[lo]) * (hi - lo);
    int idx = lo + (int) (num / (arr[hi] - arr[lo]));
    if (arr[idx] == target) return idx;
    if (arr[idx] < target) lo = idx + 1;
    else hi = idx - 1;
  }
  return -1;
}
```

- `lo` and `hi` bound the search range; if `arr[hi] == arr[lo]`, the target equals that value (avoiding division by 0).
- If the target is **smaller** or **larger** than the range's end values, the loop is never entered and −1 is returned.
- The estimated `idx` is computed and checked:
  - if equal, return idx;
  - if `arr[idx]` is smaller, continue from `idx + 1`,
  - if larger, continue up to `idx - 1`.
- If the range becomes empty, return −1.

interpolationSearch(arr, 15, 24): Tracing the Code



Target 24. Bar heights show the values: the distribution is **roughly** uniform.

Step 1 / 10

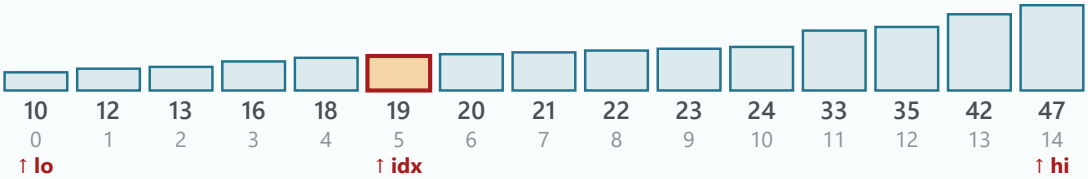
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
▶   int lo = 0, hi = n - 1;
4    while (lo <= hi && target >= arr[lo]
5                        && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
7      long num = (long) (target - arr[lo]) * (hi - lo);
8      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
10     if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

target = 24 | lo = 0 | hi = 14 | idx = –

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



idx = 0 + (24 − 10)·14 / (47 − 10) = 0 + 196/37 → **5**; arr[5] = 19.

Step 2 / 10

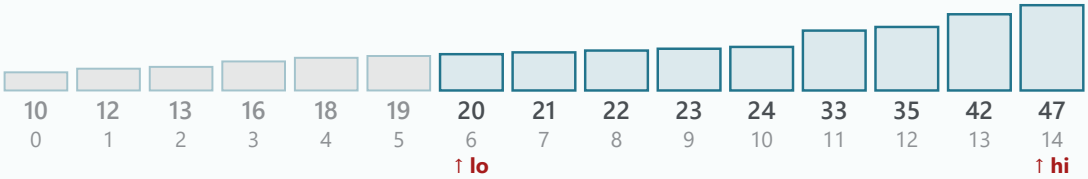
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
▶    while (lo <= hi && target >= arr[lo]
▶                    && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
▶      long num = (long) (target - arr[lo]) * (hi - lo);
▶      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
10     if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 0` `hi = 14` `idx = 5`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



19 < 24 → `lo = 6`.

Step 3 / 10

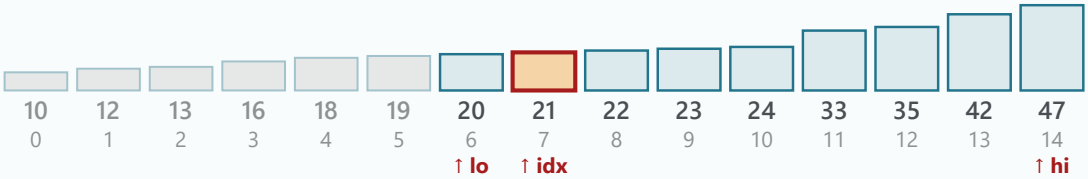
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
4    while (lo <= hi && target >= arr[lo]
5                        && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
7      long num = (long) (target - arr[lo]) * (hi - lo);
8      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
▶      if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 6` `hi = 14` `idx = 5`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



idx = 6 + (24 − 20)·8 / (47 − 20) = 6 + 32/27 → **7**; arr[7] = 21.

Step 4 / 10

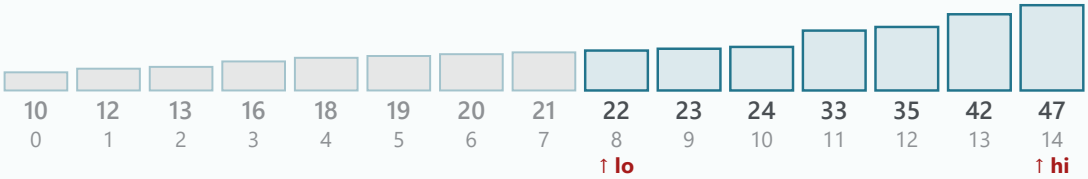
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
▶    while (lo <= hi && target >= arr[lo]
▶                    && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
▶      long num = (long) (target - arr[lo]) * (hi - lo);
▶      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
10     if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 6` `hi = 14` `idx = 7`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



21 < 24 → `lo = 8`.

Step 5 / 10

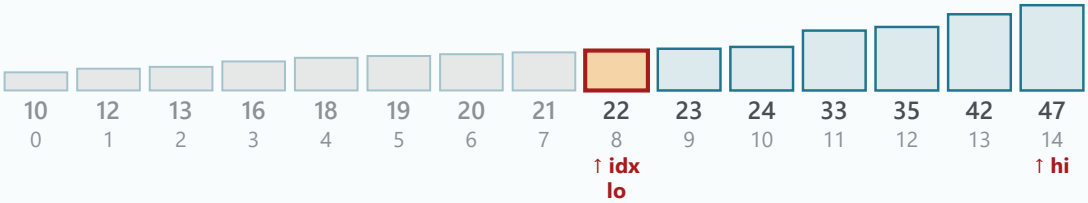
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
4    while (lo <= hi && target >= arr[lo]
5                        && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
7      long num = (long) (target - arr[lo]) * (hi - lo);
8      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
▶      if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 8` `hi = 14` `idx = 7`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



idx = 8 + (24 − 22)·6 / (47 − 22) = 8 + 12/25 → **8**; arr[8] = 22.

Step 6 / 10

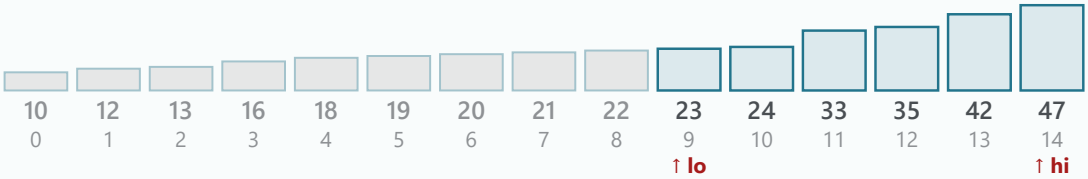
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
▶    while (lo <= hi && target >= arr[lo]
▶                    && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
▶      long num = (long) (target - arr[lo]) * (hi - lo);
▶      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
10     if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 8` `hi = 14` `idx = 8`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



22 < 24 → `lo = 9`.

Step 7 / 10

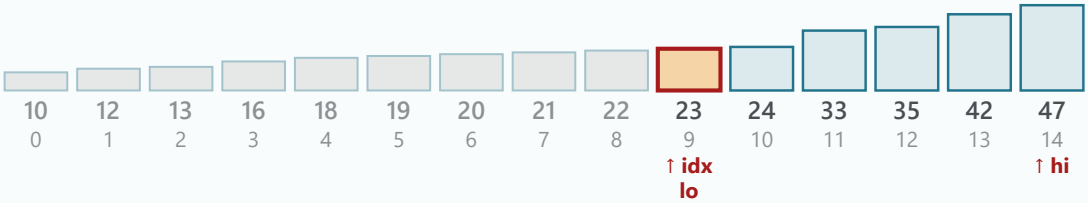
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
4    while (lo <= hi && target >= arr[lo]
5                        && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
7      long num = (long) (target - arr[lo]) * (hi - lo);
8      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
▶      if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 9` `hi = 14` `idx = 8`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



idx = 9 + (24 − 23)·5 / (47 − 23) = 9 + 5/24 → **9**; arr[9] = 23.

Step 8 / 10

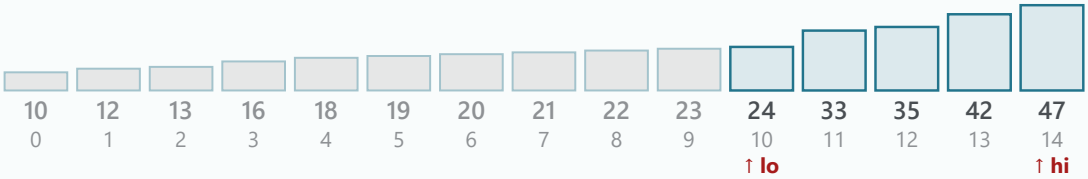
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
▶    while (lo <= hi && target >= arr[lo]
▶                    && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
▶      long num = (long) (target - arr[lo]) * (hi - lo);
▶      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
10     if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 9` `hi = 14` `idx = 9`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



23 < 24 → `lo = 10`.

Step 9 / 10

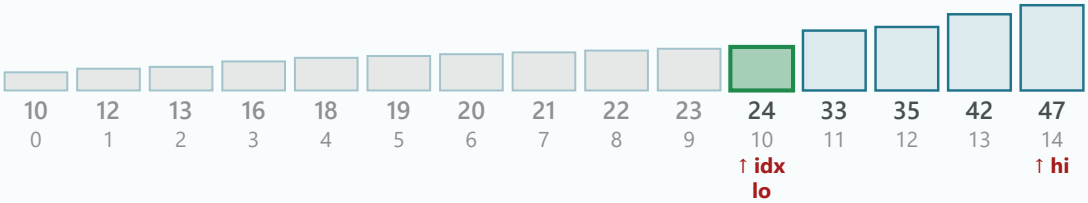
```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
4    while (lo <= hi && target >= arr[lo]
5                        && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
7      long num = (long) (target - arr[lo]) * (hi - lo);
8      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
9      if (arr[idx] == target) return idx;
▶      if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 10` `hi = 14` `idx = 9`

OUTPUT

interpolationSearch(arr, 15, 24): Tracing the Code



idx = 10 + (24 − 24)·4 / (47 − 24) = 10 + 0/23 → **10**; arr[10] = 24. Equal → **returns 10** (5 probes).

Step 10 / 10

```
1  int interpolationSearch(int[] arr, int n,
2                          int target) {
3    int lo = 0, hi = n - 1;
4    while (lo <= hi && target >= arr[lo]
5                        && target <= arr[hi]) {
6      if (arr[hi] == arr[lo]) return lo; // avoid /0
7      long num = (long) (target - arr[lo]) * (hi - lo);
8      int idx = lo + (int) (num / (arr[hi] - arr[lo]));
▶      if (arr[idx] == target) return idx;
10     if (arr[idx] < target) lo = idx + 1;
11     else hi = idx - 1;
12   }
13   return -1;
14 }
```

VARIABLES

`target = 24` `lo = 10` `hi = 14` `idx = 10`

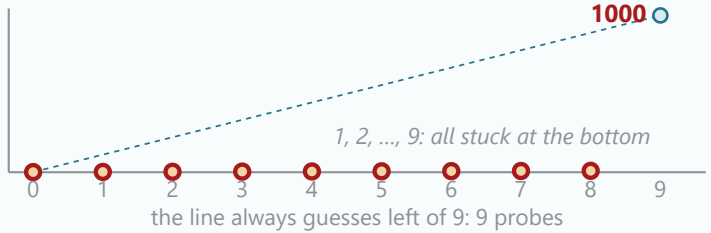
OUTPUT

```
interpolationSearch(arr, 15, 24)
→ 10
```

Interpolation Search: Complexity and the Bad Case

Case	Order
Best	O(1)
Average (uniform distribution)	O(log log n)
Worst (skewed distribution)	O(n)
Extra memory	O(1)

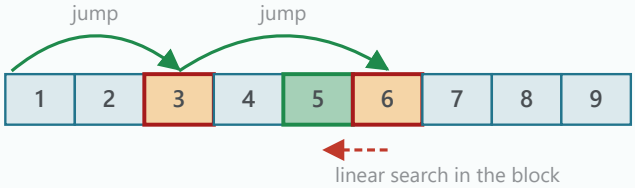
- For n = 10^6, $\log_2 \log_2 n \approx 4.3$; binary search needs ≈ 20 probes.
- Each probe costs more than in binary search (a multiplication and a division).
- The intermediate product `(target - arr[lo]) · (hi - lo)` can overflow an `int` for large values, so the code computes it as a `long`.



A bad case: arr = 1, 2, ..., 9, 1000 and target 9. The extreme value 1000 always pulls the estimate left: indices 0, 1, 2, 3, 4, 5, 6, 7, 8 → **9 probes**. Binary search finishes the same search in **3 probes** at indices 4, 7, 8.

Jump Search

- Works like a **hurdler** clearing hurdles along the track; the array must be **sorted**.
- The list is divided into **blocks** of a fixed size (m = √n).
- Jump one block ahead and check the item there (the last item of the block).
- If the target is there, the search ends; if the target is larger, keep jumping.
- If the value lies in the block just jumped over, **go back and search that range linearly**.



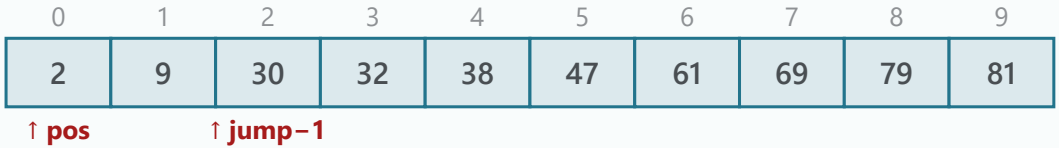
Going back needs no big leaps as in binary search: the data is read **sequentially**. So it is preferred where moving backward is costly (magnetic tape, disks read block by block).

The Jump Search Algorithm

```
int jumpSearch(int[] arr, int target,
                    int n) { // n ≤ arr.length
  if (n == 0) return -1;
  int block = (int) Math.sqrt(n);
  int jump = block, pos = 0;
  while (arr[Math.min(jump, n) - 1] < target) {
    pos = jump; jump += block; // jump
    if (pos >= n) return -1;
  }
  while (arr[pos] < target) {     // linear
    pos++;
    if (pos == Math.min(jump, n)) return -1;
  }
  if (arr[pos] == target) return pos;
  return -1;
}
```

- The jump size is computed with `Math.sqrt(n)`; `pos` starts at 0.
- The first loop looks at the **last item** of each block, jumping a block as long as it is smaller than the target.
- When the loop ends, the target can only be in the range `[pos, jump)`.
- The second loop scans this range **linearly**.
- Returns the index if found, otherwise −1.

jumpSearch(arr, 69, 10): Tracing the Code



Target 69; n = 10 → block = ⌊√10⌋ = 3; jump = 3, pos = 0.

Step 1 / 6

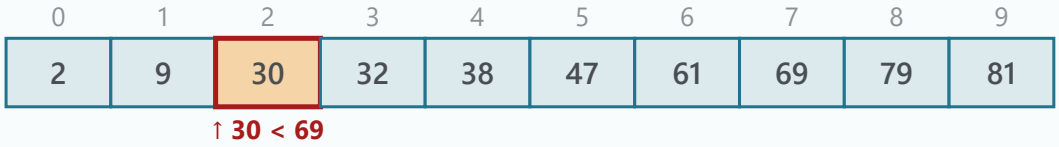
```
1  int jumpSearch(int[] arr, int target,
2                 int n) { // n ≤ arr.length
3    if (n == 0) return -1;
▶    int block = (int) Math.sqrt(n);
▶    int jump = block, pos = 0;
6    while (arr[Math.min(jump, n) - 1] < target) {
7      pos = jump; jump += block; // jump
8      if (pos >= n) return -1;
9    }
10   while (arr[pos] < target) {     // linear
11     pos++;
12     if (pos == Math.min(jump, n)) return -1;
13   }
14   if (arr[pos] == target) return pos;
15   return -1;
16 }
```

VARIABLES

target = 69 | block = 3 | jump = 3 | pos = 0

OUTPUT

jumpSearch(arr, 69, 10): Tracing the Code



arr[min(3, 10) − 1] = arr[2] = 30 < 69 → jump a block: pos = 3, jump = 6.

Step 2 / 6

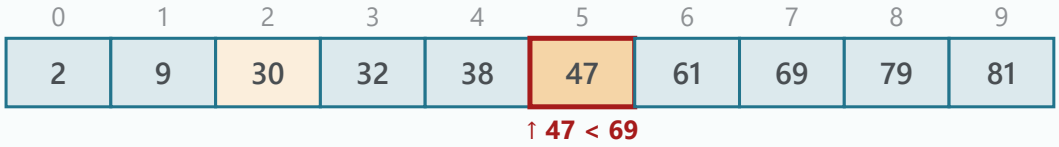
```
1  int jumpSearch(int[] arr, int target,
2                 int n) { // n ≤ arr.length
3    if (n == 0) return -1;
4    int block = (int) Math.sqrt(n);
5    int jump = block, pos = 0;
▶    while (arr[Math.min(jump, n) - 1] < target) {
▶      pos = jump; jump += block; // jump
8      if (pos >= n) return -1;
9    }
10   while (arr[pos] < target) {     // linear
11     pos++;
12     if (pos == Math.min(jump, n)) return -1;
13   }
14   if (arr[pos] == target) return pos;
15   return -1;
16 }
```

VARIABLES

target = 69 | block = 3 | jump = 6 | pos = 3

OUTPUT

jumpSearch(arr, 69, 10): Tracing the Code



arr[min(6, 10) − 1] = arr[5] = 47 < 69 → jump a block: pos = 6, jump = 9.

Step 3 / 6

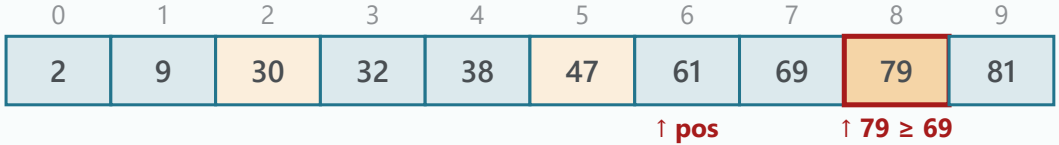
```
1  int jumpSearch(int[] arr, int target,
2                 int n) { // n ≤ arr.length
3    if (n == 0) return -1;
4    int block = (int) Math.sqrt(n);
5    int jump = block, pos = 0;
▶    while (arr[Math.min(jump, n) - 1] < target) {
▶      pos = jump; jump += block; // jump
8      if (pos >= n) return -1;
9    }
10   while (arr[pos] < target) {     // linear
11     pos++;
12     if (pos == Math.min(jump, n)) return -1;
13   }
14   if (arr[pos] == target) return pos;
15   return -1;
16 }
```

VARIABLES

target = 69 | block = 3 | jump = 9 | pos = 6

OUTPUT

jumpSearch(arr, 69, 10): Tracing the Code



arr[8] = 79 ≥ 69 → jumping stops. The target can only be in block [6, 9).

Step 4 / 6

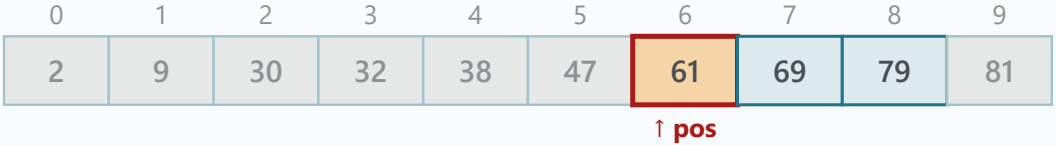
```
1  int jumpSearch(int[] arr, int target,
2                 int n) { // n ≤ arr.length
3    if (n == 0) return -1;
4    int block = (int) Math.sqrt(n);
5    int jump = block, pos = 0;
▶  while (arr[Math.min(jump, n) - 1] < target) {
7      pos = jump; jump += block; // jump
8      if (pos >= n) return -1;
9    }
10   while (arr[pos] < target) {     // linear
11     pos++;
12     if (pos == Math.min(jump, n)) return -1;
13   }
14   if (arr[pos] == target) return pos;
15   return -1;
16 }
```

VARIABLES

target = 69 | block = 3 | jump = 9 | pos = 6

OUTPUT

jumpSearch(arr, 69, 10): Tracing the Code



Linear search: arr[6] = 61 < 69 → pos = 7.

Step 5 / 6

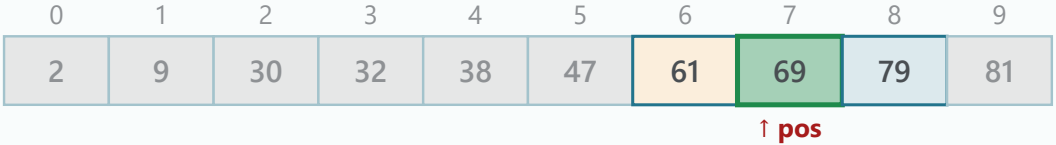
```
1  int jumpSearch(int[] arr, int target,
2                 int n) { // n ≤ arr.length
3    if (n == 0) return -1;
4    int block = (int) Math.sqrt(n);
5    int jump = block, pos = 0;
6    while (arr[Math.min(jump, n) - 1] < target) {
7      pos = jump; jump += block; // jump
8      if (pos >= n) return -1;
9    }
▶    while (arr[pos] < target) {     // linear
▶      pos++;
▶      if (pos == Math.min(jump, n)) return -1;
13   }
14   if (arr[pos] == target) return pos;
15   return -1;
16 }
```

VARIABLES

target = 69 | block = 3 | jump = 9 | pos = 7

OUTPUT

jumpSearch(arr, 69, 10): Tracing the Code



arr[7] = 69 → **returns 7**. Probed: 2, 5, 8, 6, 7.

Step 6 / 6

```
1  int jumpSearch(int[] arr, int target,
2                 int n) { // n ≤ arr.length
3    if (n == 0) return -1;
4    int block = (int) Math.sqrt(n);
5    int jump = block, pos = 0;
6    while (arr[Math.min(jump, n) - 1] < target) {
7      pos = jump; jump += block; // jump
8      if (pos >= n) return -1;
9    }
10   while (arr[pos] < target) {     // linear
11     pos++;
12     if (pos == Math.min(jump, n)) return -1;
13   }
▶    if (arr[pos] == target) return pos;
15   return -1;
16 }
```

VARIABLES

`target = 69` `block = 3` `jump = 9` `pos = 7`

OUTPUT

```
jumpSearch(arr, 69, 10) → 7
```

Jump Search: Why √n?

```
worst-case probes ≈ n/m + (m − 1)
```

- n/m: the maximum number of jumps; m − 1: the linear search inside a block.
- Derivative: $-n/m^2 + 1 = 0 \Rightarrow$ **m = √n**; total ≈ 2√n − 1.
- Complexity **O(√n)**: between linear O(n) and binary O(log n).
- Extra memory O(1); the array must be sorted.

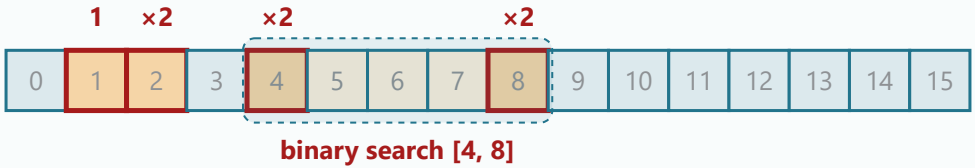
n	m = √n	≈ 2√n − 1	binary
100	10	19	7
10,000	100	199	14
10^6	1,000	1,999	20

Once the block is found, it can be scanned backward from its end (8 → 7) or forward from its start (6 → 7) as in `jumpSearch`. Either way it takes at most m − 1 extra probes.

Improvement: once the block is found, search inside it by jumping again (in steps of √m) or by binary search. Given binary search, jump search only makes sense where **moving backward is expensive**.

Exponential Search

- Splits the sorted list into **exponentially growing** pieces: the bound **doubles** as 1, 2, 4, 8, 16, ….
- Stop once `arr[bound]` exceeds the target: the target lies in **[bound/2, bound]**.
- Run **binary search** on this range (Bentley and Yao, 1976).
- For example, on a list of 1024 items with the target near index 600, the bounds are 1, 2, …, 512, 1024, and binary search continues between 512 and 1023.



Why? Ideal when the array's length is unknown (unbounded or streaming data) or the target is near the front: the cost depends not on n but on the target's **position** (i).

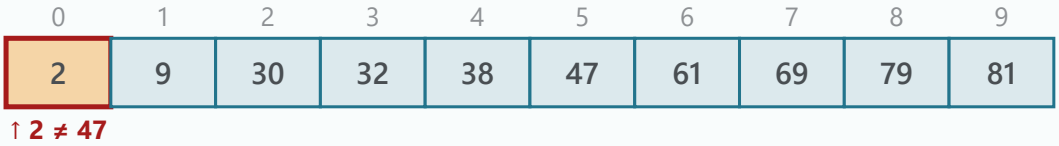
The Exponential Search Algorithm

```
int exponentialSearch(int[] arr, int target, int n) {
  if (n == 0) return -1;
  if (arr[0] == target) return 0;
  int bound = 1;
  while (bound < n && arr[bound] <= target)
    bound *= 2;               // 1, 2, 4, 8, ...
  int lo = bound / 2, hi = Math.min(bound, n - 1);
  while (lo <= hi) {        // binary search
    int mid = (lo + hi) / 2;
    if (arr[mid] == target) return mid;
    if (arr[mid] < target) lo = mid + 1;
    else hi = mid - 1;
  }
  return -1;
}
```

- `bound` starts at 1 and is doubled as long as `arr[bound] ≤ target`.
- When the loop ends, `bound` marks the **end** of the range that can contain the target; `bound / 2` is its start.
- `Math.min(bound, n − 1)`: the bound must not run past the array.
- The remaining part is searched with binary search.

If the final phase used a **linear** scan (from `bound / 2` onward) instead of binary search, the cost would be O(i), and an absent target would scan to the end of the array. Binary search brings the cost down to O(log i).

exponentialSearch(arr, 47, 10): Tracing the Code



Target 47. First, arr[0] = 2 ≠ 47.

Step 1 / 8

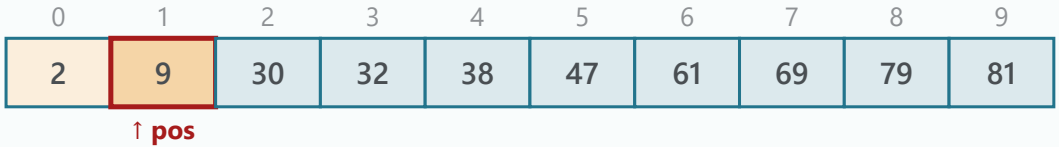
```
1  int exponentialSearch(int[] arr, int target, int n) {
2    if (n == 0) return -1;
▶    if (arr[0] == target) return 0;
▶    int bound = 1;
5    while (bound < n && arr[bound] <= target)
6      bound *= 2;                // 1, 2, 4, 8, ...
7    int lo = bound / 2, hi = Math.min(bound, n - 1);
8    while (lo <= hi) {          // binary search
9      int mid = (lo + hi) / 2;
10     if (arr[mid] == target) return mid;
11     if (arr[mid] < target) lo = mid + 1;
12     else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

target = 47 | bound = 1 | lo = – | hi = – | mid = –

OUTPUT

exponentialSearch(arr, 47, 10): Tracing the Code



arr[1] = 9 ≤ 47 → bound = 2.

Step 2 / 8

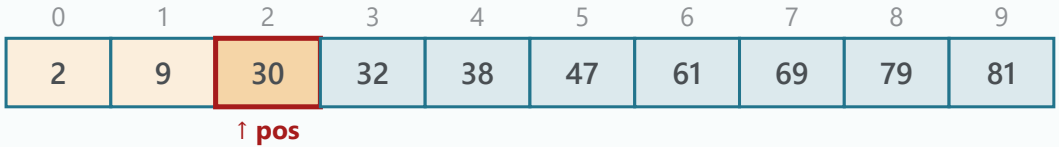
```
1  int exponentialSearch(int[] arr, int target, int n) {
2    if (n == 0) return -1;
3    if (arr[0] == target) return 0;
4    int bound = 1;
▶    while (bound < n && arr[bound] <= target)
▶      bound *= 2;               // 1, 2, 4, 8, ...
7    int lo = bound / 2, hi = Math.min(bound, n - 1);
8    while (lo <= hi) {         // binary search
9      int mid = (lo + hi) / 2;
10     if (arr[mid] == target) return mid;
11     if (arr[mid] < target) lo = mid + 1;
12     else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

target = 47 | bound = 2 | lo = – | hi = – | mid = –

OUTPUT

exponentialSearch(arr, 47, 10): Tracing the Code



arr[2] = 30 ≤ 47 → bound = 4.

Step 3 / 8

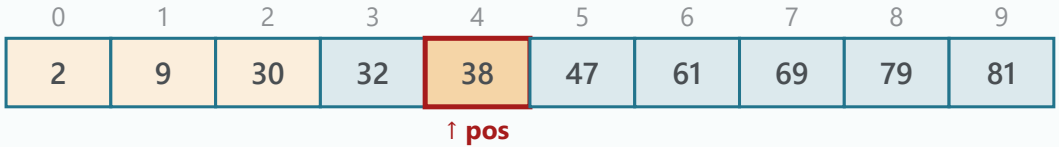
```
1  int exponentialSearch(int[] arr, int target, int n) {
2    if (n == 0) return -1;
3    if (arr[0] == target) return 0;
4    int bound = 1;
▶    while (bound < n && arr[bound] <= target)
▶      bound *= 2;               // 1, 2, 4, 8, ...
7    int lo = bound / 2, hi = Math.min(bound, n - 1);
8    while (lo <= hi) {         // binary search
9      int mid = (lo + hi) / 2;
10     if (arr[mid] == target) return mid;
11     if (arr[mid] < target) lo = mid + 1;
12     else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

target = 47 | bound = 4 | lo = – | hi = – | mid = –

OUTPUT

exponentialSearch(arr, 47, 10): Tracing the Code



arr[4] = 38 ≤ 47 → bound = 8.

Step 4 / 8

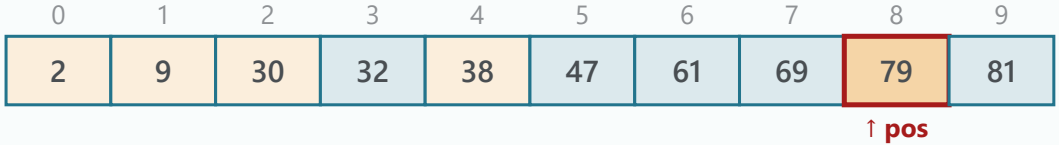
```
1  int exponentialSearch(int[] arr, int target, int n) {
2    if (n == 0) return -1;
3    if (arr[0] == target) return 0;
4    int bound = 1;
▶    while (bound < n && arr[bound] <= target)
▶      bound *= 2;               // 1, 2, 4, 8, ...
7    int lo = bound / 2, hi = Math.min(bound, n - 1);
8    while (lo <= hi) {         // binary search
9      int mid = (lo + hi) / 2;
10     if (arr[mid] == target) return mid;
11     if (arr[mid] < target) lo = mid + 1;
12     else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

target = 47 | bound = 8 | lo = – | hi = – | mid = –

OUTPUT

exponentialSearch(arr, 47, 10): Tracing the Code



arr[8] = 79 > 47 → doubling stops. The target lies in [8/2, min(8, 9)] = [4, 8].

Step 5 / 8

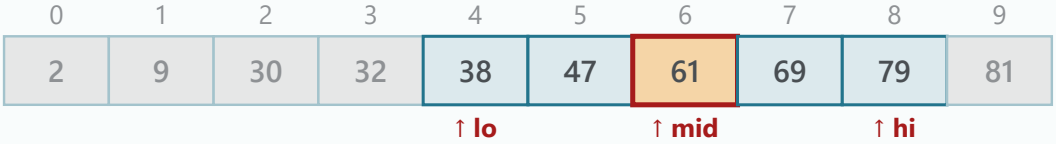
```
1  int exponentialSearch(int[] arr, int target, int n) {
2    if (n == 0) return -1;
3    if (arr[0] == target) return 0;
4    int bound = 1;
▶    while (bound < n && arr[bound] <= target)
6      bound *= 2;               // 1, 2, 4, 8, ...
▶    int lo = bound / 2, hi = Math.min(bound, n - 1);
8    while (lo <= hi) {         // binary search
9      int mid = (lo + hi) / 2;
10     if (arr[mid] == target) return mid;
11     if (arr[mid] < target) lo = mid + 1;
12     else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

target = 47 | bound = 8 | lo = 4 | hi = 8 | mid = –

OUTPUT

exponentialSearch(arr, 47, 10): Tracing the Code



Binary search: mid = (4 + 8) / 2 = 6 → arr[6] = 61 > 47 → hi = 5.

Step 6 / 8

```
1  int exponentialSearch(int[] arr, int target, int n) {
2    if (n == 0) return -1;
3    if (arr[0] == target) return 0;
4    int bound = 1;
5    while (bound < n && arr[bound] <= target)
6      bound *= 2;               // 1, 2, 4, 8, ...
7    int lo = bound / 2, hi = Math.min(bound, n - 1);
8    while (lo <= hi) {          // binary search
▶      int mid = (lo + hi) / 2;
▶      if (arr[mid] == target) return mid;
▶      if (arr[mid] < target) lo = mid + 1;
▶      else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

`target = 47` `bound = 8` `lo = 4` `hi = 8` `mid = 6`

OUTPUT

exponentialSearch(arr, 47, 10): Tracing the Code



Binary search: mid = (4 + 5) / 2 = 4 → arr[4] = 38 < 47 → lo = 5.

Step 7 / 8

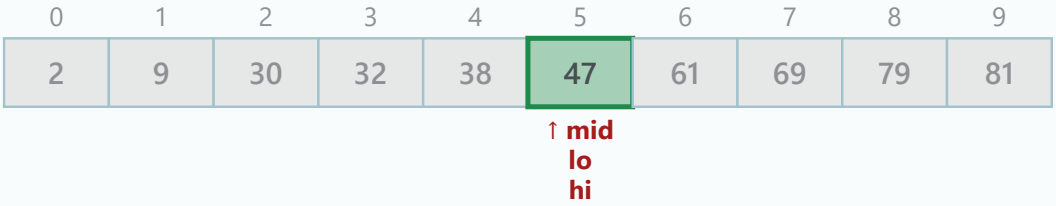
```
1  int exponentialSearch(int[] arr, int target, int n) {
2    if (n == 0) return -1;
3    if (arr[0] == target) return 0;
4    int bound = 1;
5    while (bound < n && arr[bound] <= target)
6      bound *= 2;                // 1, 2, 4, 8, ...
7    int lo = bound / 2, hi = Math.min(bound, n - 1);
8    while (lo <= hi) {          // binary search
▶      int mid = (lo + hi) / 2;
▶      if (arr[mid] == target) return mid;
▶      if (arr[mid] < target) lo = mid + 1;
12     else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

target = 47 | bound = 8 | lo = 4 | hi = 5 | mid = 4

OUTPUT

exponentialSearch(arr, 47, 10): Tracing the Code



Binary search: mid = (5 + 5) / 2 = 5 → arr[5] = 47 = 47 → **returns 5**.

Step 8 / 8

```
 1 int exponentialSearch(int[] arr, int target, int n) {
 2   if (n == 0) return -1;
 3   if (arr[0] == target) return 0;
 4   int bound = 1;
 5   while (bound < n && arr[bound] <= target)
 6     bound *= 2;               // 1, 2, 4, 8, ...
 7   int lo = bound / 2, hi = Math.min(bound, n - 1);
 8   while (lo <= hi) {        // binary search
▶      int mid = (lo + hi) / 2;
▶      if (arr[mid] == target) return mid;
11     if (arr[mid] < target) lo = mid + 1;
12     else hi = mid - 1;
13   }
14   return -1;
15 }
```

VARIABLES

target = **47** bound = **8**

lo = **5** hi = **5** mid = **5**

OUTPUT

```
exponentialSearch(arr, 47, 10) → 5
```

Exponential Search: Complexity

- If the target is at index i, the doubling phase takes $\lfloor \log_2 i \rfloor + 1$ steps.
- The final range [i/2, i] has at most i/2 + 1 items → binary search takes ≈ $\log_2$ i steps.
- Total: **O(log i)**; worst case (i ≈ n) **O(log n)**, best case O(1). Extra memory O(1).
- Faster than binary search when the target is near the front; ≈ 2× slower when it is near the end.

i (n = 10^6)	exponential (≈ 2 $\log_2$ i)	binary (≈ $\log_2$ n)
10	≈ 7	20
1,000	≈ 20	20
500,000	≈ 38	20

Uses: unbounded (unknown-length) sorted lists, records near the start of a large file, and "galloping" search in sorted streams; Timsort, for instance, uses this idea when merging.

COMPARISONS

Comparing the Search Algorithms

Algorithm	Requires	Best	Average	Worst	Extra memory
Linear	nothing	O(1)	O(n)	O(n)	O(1)
Binary	sorted	O(1)	O(log n)	O(log n)	O(1)
Hash (table)	hash table	O(1)	O(1)	O(n)	O(m)
Interpolation	sorted, uniform	O(1)	O(log log n)	O(n)	O(1)
Jump	sorted	O(1)	O(√n)	O(√n)	O(1)
Exponential	sorted	O(1)	O(log i)	O(log n)	O(1)

i: index of the target, m: number of buckets. The O(1) average of a hash table relies on keeping the load factor bounded. If sorting is needed, add O(n log n) once.

COMPARISONS

Probe Counts on the Same Array

Array: **2, 9, 30, 32, 38, 47, 61, 69, 79, 81** (the array from the jump and exponential search examples).
Cell: number of probes (indices probed).

target	Linear	Binary	Jump	Interpolation	Exponential	result
69	8 (0,1,2,3,4,5,6,7)	2 (4,7)	5 (2,5,8,6,7)	1 (7)	7 (0,1,2,4,8,6,7)	7
47	6 (0,1,2,3,4,5)	3 (4,7,5)	5 (2,5,3,4,5)	1 (5)	8 (0,1,2,4,8,6,4,5)	5
2	1 (0)	3 (4,1,0)	2 (2,0)	1 (0)	1 (0)	0
50	10 (0,1,2,3,4,5,6,7,8,9)	4 (4,7,5,6)	4 (2,5,8,6)	1 (5)	8 (0,1,2,4,8,6,4,5)	-1

Since the array is nearly uniform, interpolation search makes **a single probe** for every query. Searching for 50, it probes 47 and then returns −1 at once because the target falls outside the range. Exponential search probes index 4 twice (in the doubling and the binary phase).

COMPARISONS

Which Algorithm When?

Linear
Unsorted or small data; linked lists; one-off searches.

Binary
Sorted array, many queries; the general-purpose default.

Hash
Exact-match queries only, with many insertions / searches; no ordering needed.

Interpolation
Sorted, uniformly distributed numeric keys (e.g. sequential ID numbers).

Jump
Sequential media where going back is expensive (tape, block reads).

Exponential
A sorted list of unknown length, or a target near the front.

Order of questions: **Is the data sorted?** No → linear search, or build a hash table. Yes → binary search; interpolation if the distribution is uniform, exponential if the length is unknown, jump if going back is expensive.

COMPARISONS

Search Race: Five Algorithms on One Array

Target 65 linear binary jump interpolation exponential uniform array skewed array

3	8	15	21	27	34	40	46	52	59	65	71	78	84	90	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Enter a value and pick an algorithm (Enter = binary).

The number above a cell is the probe order; green: found. The table compares the probe counts of the five algorithms for the last target. Interpolation search slows down on the skewed array.

Summary

Topic	Key points
Linear	compare one by one from start to end; no precondition; average $(n + 1)/2$, worst n → $O(n)$
Binary	sorted array; compare with the middle, discard half; at most $\lfloor \log_2 n \rfloor + 1$ → $O(\log n)$; the recursive version uses an $O(\log n)$ stack
Hash	compute the position: h(key); collisions → linear probing or chaining; avg. $O(1)$, worst $O(n)$
Interpolation	estimate the position from the value ratio; $O(\log \log n)$ on uniform data, $O(n)$ on skewed data
Jump	jump in blocks of $\sqrt{n}$, then search linearly inside the block → $O(\sqrt{n})$
Exponential	double the bound 1, 2, 4, …, then binary search in [bound/2, bound] → $O(\log i)$

SUMMARY

Common Mistakes

Binary search on an unsorted array

No error, but it can return a wrong result (−1). Binary, interpolation, jump and exponential search all need a **sorted** array.

Off-by-one at the bounds

`while (lo < hi)` skips a one-item range; `hi = mid` can cause an infinite loop.

Overflow computing mid

On very large arrays `(lo + hi) / 2` overflows; `lo + (hi − lo) / 2` does not.

Division in interpolation

If `arr[hi] == arr[lo]` it divides by zero; if the target is outside the end values the index leaves the array. Check the range first.

A full hash table

With linear probing and no empty bucket left, the search loop never ends. Bound the load factor or stop after m probes.

The bound in exponential search

Forgetting `Math.min(bound, n − 1)` gives an `ArrayIndexOutOfBoundsException`; a linear scan in the last phase ruins O(log i).

In linear probing a deleted bucket is not set to `null` but marked DELETED; the search skips over it and continues.

SUMMARY

Sample Problems

Array D = **3, 8, 15, 21, 27, 34, 40, 46, 52, 59, 65, 71, 78, 84, 90, 97** (n = 16, indices from 0).

1. Which indices does binary search probe for 65?
2. Which indices does jump search probe for 65?
3. In how many probes does interpolation search settle 65 and 30?
4. Which indices does exponential search probe for 65?
5. m = 7, h(x) = x % 7, linear probing: insert 10, 24, 17, 31, 5. What is the final table? How many buckets are probed when searching for 12?

▶ **Show answers**

SUMMARY

Exercises

1. Does ternary search, which splits the range into **three** parts instead of two (looking at two middle items), make fewer comparisons than binary search?
2. Write a binary search that finds the **first** position of the target in a sorted array with duplicates.
3. Show that the average number of comparisons in a successful linear search is $(n + 1)/2$.
4. Show that the best block size for jump search is $\sqrt{n}$; what is the worst-case probe count for $n = 10{,}000$?
5. In how many probes does interpolation search find 70 in 10, 20, …, 100? And 9 in 1, 2, …, 9, 1000?
6. A chained hash table ($m = 10$) holds 16, 12, 45, 29, 6, 122, 5, 58, 75. How many keys are compared when searching for 35?

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Sections 2.3, 11).
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011 (Section 3.1 Binary search, 3.4 Hash tables).
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., 1998.
- J. L. Bentley and A. C. Yao, "An Almost Optimal Algorithm for Unbounded Searching", *Information Processing Letters* 5(3), 1976.
- Y. Perl, A. Itai and H. Avni, "Interpolation Search — A Log Log N Search", *Communications of the ACM* 21(7), 1978.
- B. Shneiderman, "Jump Searching: A Fast Sequential Search Technique", *Communications of the ACM* 21(10), 1978.

The End

Chapter 4: Searching Algorithms

Previous: Chapter 3: Sorting Algorithms · Next: Chapter 5: Graph Traversal