

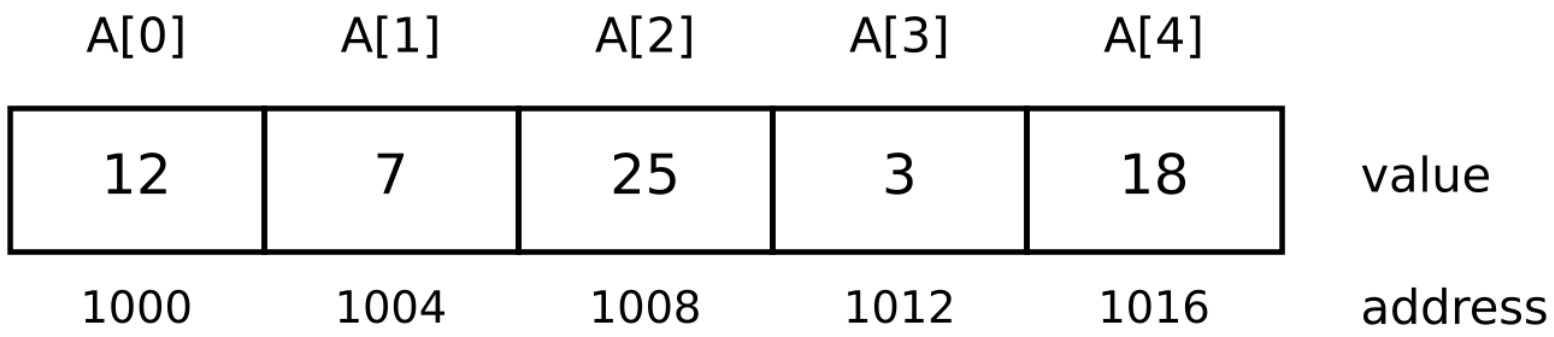
Course Algorithms and Data Structures · Lecture 4

Arrays. Sequential and Binary Search

Textbook: A. Levitin, Sections 1.4, 3.1, 4.4

Arrays

An **array** A[0..n – 1] stores n elements of the same type **next to each other** in memory. Each element is reached by its index.



Why access is instant

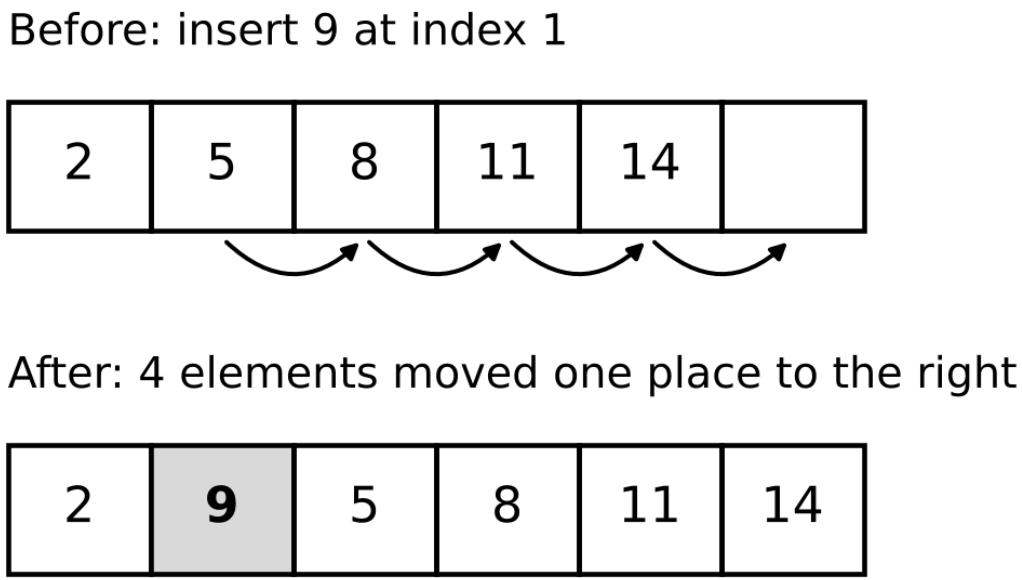
$$\text{address}(A[i]) = \text{start} + i \cdot s$$

start: address of A[0]; s: size of one element

In the picture: address(A[3]) = 1000 + 3 · 4 = 1012.

One multiplication and one addition, for 5 elements or 5 million: **O(1)**. This is called **random access**.

Why insertion is slow



In the worst case (insert at the front) all n elements move: **O(n)**. Deletion is the same. Inserting at the end is O(1).

Static and dynamic arrays

Static array

fixed size, chosen when it is created: int a[100] in C++

Dynamic array

grows when needed: vector in C++, list in Python. When full, it copies everything into a block twice as big

Copying is rare, so adding at the end is still O(1) on average.

Cost of array operations

Operation	Cost	Why
Read or change A[i]	O(1)	address is calculated
Search, unsorted array	O(n)	may check every element
Search, sorted array	O(log n)	binary search
Insert / delete at the end	O(1)	nothing to move
Insert / delete in the middle	O(n)	elements are shifted

Sequential search

```
ALGORITHM SequentialSearch(A[0..n − 1], K)
    i ← 0
    while i < n and A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```

Best case 1 comparison, worst case n: **O(n)**. Works on any array, sorted or not.

A trick: the sentinel

```
ALGORITHM SequentialSearch2(A[0..n], K)
    A[n] ← K              //the sentinel
    i ← 0
    while A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```

K at the end guarantees that the loop stops, so the check i < n disappears. Still O(n), but each step is cheaper.

Binary search: the idea

Compare K with the **middle** element of a **sorted** array:

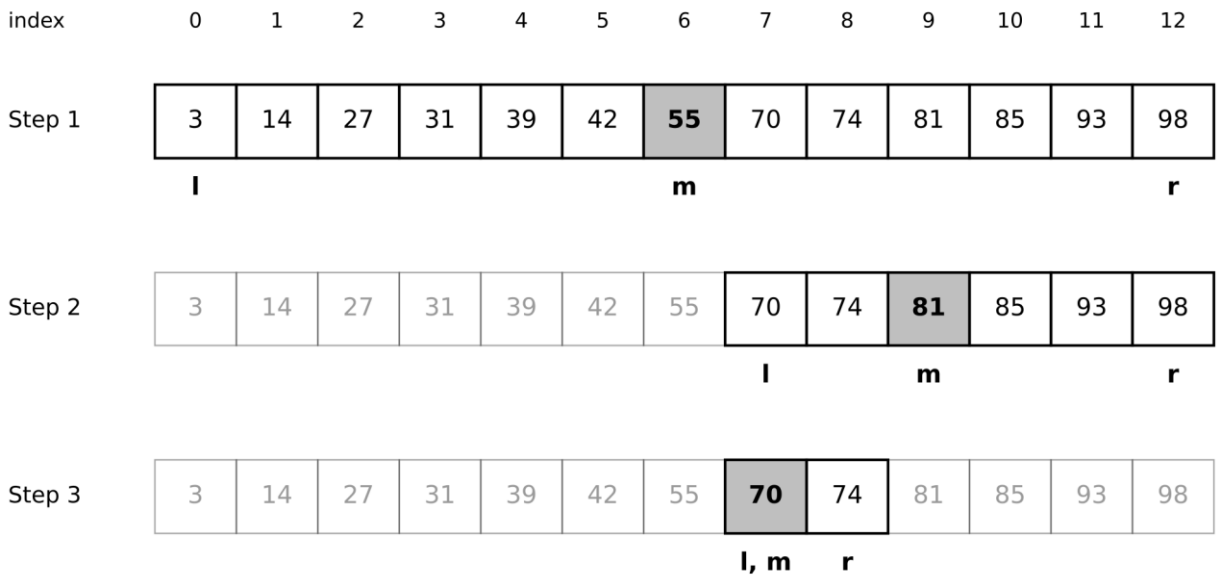
- equal: found
- K is smaller: keep only the **left** half
- K is bigger: keep only the **right** half

> Indices l and r mark the part of the array where K can still be.

Binary search (iterative)

```
ALGORITHM BinarySearch(A[0..n − 1], K)
    //Input: A sorted in ascending order, a key K
    l ← 0;  r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r) / 2⌋
        if K = A[m] return m
        else if K < A[m] r ← m − 1
        else l ← m + 1
    return −1
```

Searching for K = 70



Step	l	r	m	A[m]	Decision
1	0	12	6	55	70 > 55: l ← 7
2	7	12	9	81	70 < 81: r ← 8
3	7	8	7	70	found

Searching for K = 10 (not in the array)

Step	l	r	m	A[m]	Decision
1	0	12	6	55	10 < 55: r ← 5
2	0	5	2	27	10 < 27: r ← 1
3	0	1	0	3	10 > 3: l ← 1
4	1	1	1	14	10 < 14: r ← 0
–	1	0			l > r: return −1

When l > r, the part where K could be is empty.

How fast is binary search?

$$C_worst(n) = \lfloor \log_2 n \rfloor + 1$$

n	Sequential (worst)	Binary (worst)
100	100	7
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

Where people make mistakes

- The loop condition is **$l \le r$**, not $l < r$
- Move past the middle: **$r \leftarrow m - 1$**, **$l \leftarrow m + 1$**; with $r \leftarrow m$ the loop may never end
- In C++ or Java, $l + r$ can overflow: use $m \leftarrow l + \lfloor (r - l)/2 \rfloor$
- Binary search on **unsorted** data gives wrong answers silently

Which search to use?

	Sequential	**Binary**
Needs sorted data?	no	yes
Worst case	O(n)	O(log n)
Works on a linked list?	yes	no
Best for	small or unsorted data, one search	large sorted data, many searches

Many searches in the same data: sort once, then search in O(log n).

Summary

- An array stores elements next to each other: A[i] in O(1), but inserting in the middle costs O(n).
- Sequential search works on any array and takes O(n) in the worst case.
- Binary search needs a sorted array, halves the range at every step and takes O(log n).
- A million elements: a million comparisons against twenty.

Review questions

1. Why can we reach any element of an array in one step? Why is inserting into the middle slow?
2. Trace binary search for K = 31 in 3, 14, 27, 31, 39, 42, 55, 70, 74, 81, 85, 93, 98. Write l, r, m for every step.
3. When would you use sequential search instead of binary search? Give two reasons.

Reading: Levitin, Sections 1.4, 3.1, 4.4; Bhargava, Chapters 1–2.