fit@hcmus

DATA STRUCTURES & ALGORITHMS

Lecture 3: Searching

Lecturer: Dr. Nguyen Hai Minh

CONTENT

- Searching Problem - Introduction
- Searching Algorithms:
  - Sequential/Linear search
  - Binary search
- Exercises
- Conclusion

SEARCHING INTRODUCTION

Searching problem – Introduction

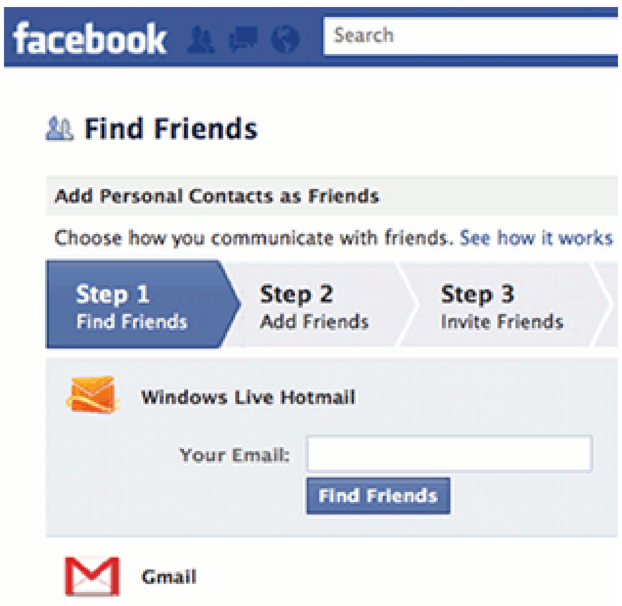
- Definition:
  - Finding an item with specified properties among a collection of items
  - Finding the position of an element with a specific value (key) among a collection of elements.
  - Finding a record in a database.

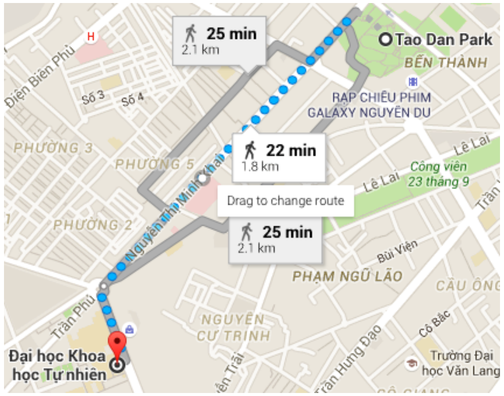
record = key + satellite data

We focus on searching numbers only

Searching problem – Introduction

- Why searching?
  - Searching is a very popular operation in computing.
  - The need of an application:

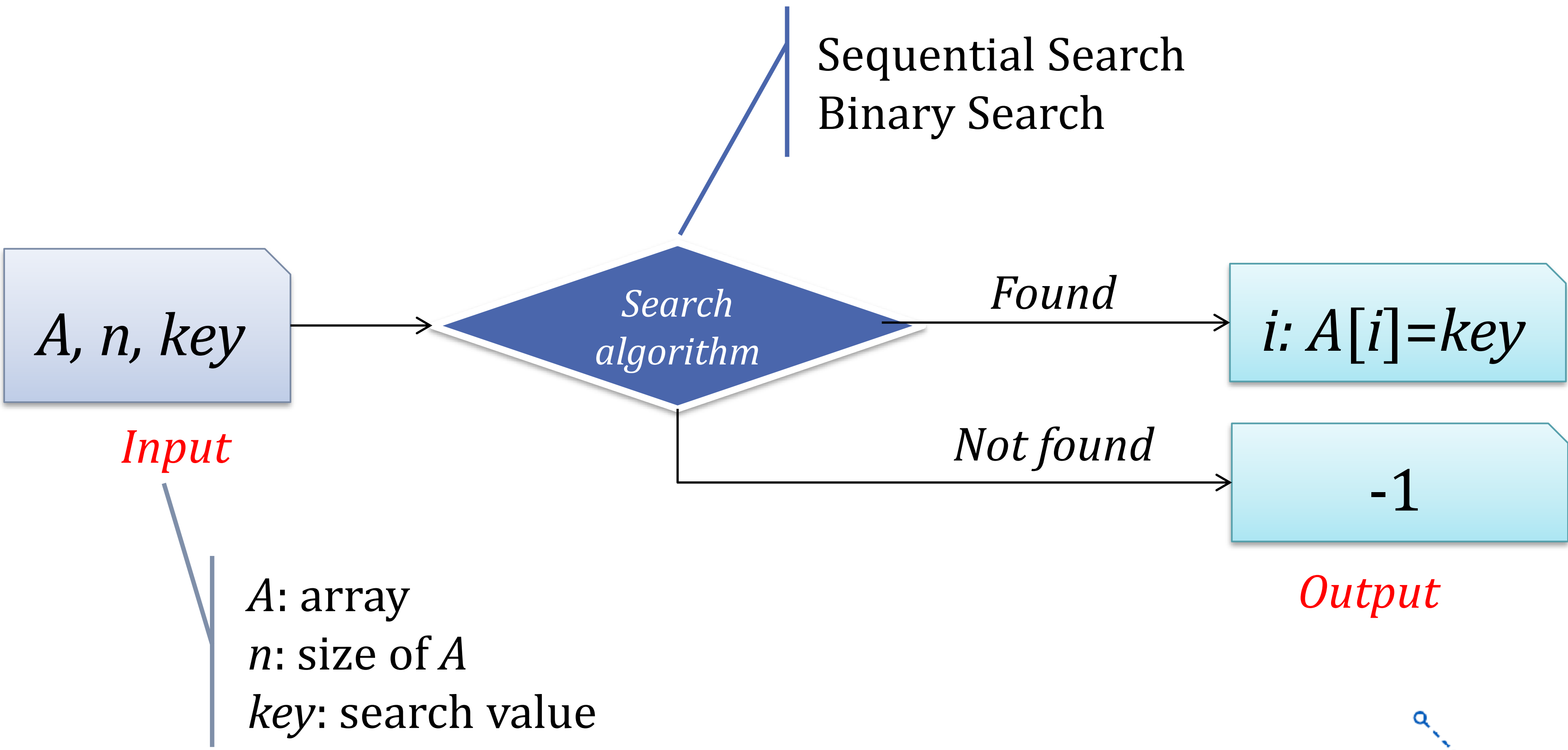




Searching problem – Introduction

- Big amount of data:
  - → The need of fast search algorithms.
  - → Faster in case the data has been sorted. (Example: dictionary, books in library, …)
- Local search algorithms (search in memory):
  - Sequential/ Linear Search
  - Binary Search

Searching problem – Introduction



SEQUENTIAL SEARCH

Sequential Search – Idea

- **Brute-force approach** (Exhaustive search)
  - Compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search)
- **Extra trick:** using a sentinel
  - Append the search key to the end of the list
  - → Eliminate the end of list check altogether.

Sequential Search – Algorithm

SEQUENTIAL-SEARCH(A[0…n-1],key)
//Input: An array A[0..n − 1] of integers and a key to search
//Output: position of key in array A or -1 if key is not in A
	Cost times

`for i ← 0 to n − 1 do`
`    if A[i] = key`
`        return i`
`return -1`
	$C(n)$

Sequential Search Analysis

1. Input size: n
2. Basic operation: key comparison `A[i] = key`
3. The number of key comparisons ***depends on the nature of the input.***
4. Sum of basic operations:
   - Best-case:
     - $A[0] = key \rightarrow C(n) = 1 \in O(1)$
   - Worst-case:
     - *key* is not in *A* $\rightarrow C(n) = n \in O(n)$
   - Average-case:
     - *key* is among *A*[0] and *A*[*n*-1] $\rightarrow C(n) = \frac{n+1}{2} \in O(n)$
5. Order of growth: $\boldsymbol{O(n)}$

Sequential Search – Idea

- Idea: Brute-force approach
  - Compare *key* with each element in *A* until we find *key* in *A* or reach the end of *A*.
  - Example: *A* = {1, 25, 6, 5, 2, 37, 40}, *key* = 6

key = 6

1	25	6	5	2	37	40

key = 6

1	**25**	6	5	2	37	40

key = 6

1	25	**6**	5	2	37	40

Return 2

Sequential Search with Sentinel

- In the exhaustive search, there is a comparison operation inside the iteration to detect the end of the array:

```
for (int i=0; i < n; i++)
```

- This comparison can be omitted using a "sentinel":

Sentinel value

- A special value that signals the end of a loop.
  - Sentinel loop: repeats until a sentinel value is seen.
- Example: A program that prompts the user for an integer input until the user types “0”, then output the total numbers inputted.
  - Enter a number (or 0 to exit): 12
  - Enter a number (or 0 to exit): 23
  - Enter a number (or 0 to exit): 0

  You have inputted 2 numbers.

  sentinel
- Also called *flag, trip, rogue, signal value* or *dummy data*.

Sentinel loop – Example

```
void IntegerInput()
{
    int num;
    int count = -1;
    do{
        cout << "Enter a number (or 0 to exit):");
         cin >> num;
         count++;
    }while(num != 0);
    cout << "The total inputted number is: " << count;
}
```

Sequential Search with Sentinel

- Sentinel in Sequential search:
  - An element that has the same value with key.
  - Put at the end of the array.
- Idea:
  - Continue search until key is found at $A[i]$
    - If $i < n$: the search key appear in *A*.
    - If $i = n$: the search key is not in *A*.

Sequential Search with Sentinel

- Example: $A = \{1, 25, 5, 2, 37\}$, $key = 6$ (n=5)

1	25	5	2	37

i=0

1	25	5	2	37	6

i=1

1	25	5	2	37	6

i=2

1	25	5	2	37	6

i=3

1	25	5	2	37	6

i=4

1	25	5	2	37	6

i=5

1	25	5	2	37	6

➡ **return -1;**

Sequential Search with Sentinel

SEQUENTIAL−SEARCH2(A[0…n−1],key)
//Input: An array A[0..n − 1] of integers and a key to search
//Output: position of key in array A or -1 if key is not in A
	Cost times

`A[n] ← key`	
`while A[i] ≠ key do`	$C(n)$
`    i ← i + 1`	
`if i < n`	
`    return i`	
`else return −1`	

- Worst-case:
  - $C(n) = n + 1 \in O(n)$

Sequential Search with Sentinel

- Experimental results show that with a sufficient large *n*, sequential search with a sentinel is faster than the original sequential search.
  - *n*=15000: 20% faster (0.22s vs 0.28s)

→ *Why?*

Sequential Search – Conclusion

- Sequential search provides an excellent illustration of the brute-force approach
  - **Pros:** Simple to implement
  - **Cons:** Slow!
- However, it is affected by the order of input elements

  → A general search approach for any kind of array.

BINARY SEARCH

Binary Search

- Is a sequential search on an already sorted array faster than an unsorted array?
- Binary search: take the advantage of the elements that are in order to narrow the search range.
  - Example: Look for "Harry" in the sorted contact list:

1	Andy			
2	Bobby			
3	Cathy			
4	David	Discard		
5	Ellen			
6	Fred			
7	George			
8	Harry	Harry	Harry	Found!
9	Helen	Helen	Helen	
10	James	James		
11	Kate	Kate		
12	Loren	Loren	Discard	
13	Will	Will		

Binary Search

- Look for "Harry" in the sorted contact list:
  - The search is dramatically reduced:
    - each time reduce ½ of search size

1	Andy			
2	Bobby			
3	Cathy			
4	David	Discard		
5	Ellen			
6	Fred			
7	George			
8	Harry	Harry	Harry	Found!
9	Helen	Helen	Helen	
10	James	James		
11	Kate	Kate	Discard	
12	Loren	Loren		
13	Will	Will		

Binary Search – Idea

- **Decrease-and-conquer approach:**
  → *Decrease by a constant factor*

Let *left* = 0, *right* = *n*-1, *mid* = (*left*+*right*)/2

1. If *left* > *right*: stop the search, *key* is not in the array *A*.
2. Compare *key* with *A*[*mid*].

   2.1 If *key* = *A*[*mid*]: return mid.

   2.2 Else if *key* < *A*[*mid*]: search for *key* on the left half array. (*right* = *mid*-1, go to step 1 again)

   2.3 Else: search for *key* on the right half array. (*left* = *mid*+1, go to step 1 again)

Binary Search – Algorithm

BINARY-SEARCH2(A[0…n−1],key)
//Input: An array A[0..n − 1] of sorted integers and a search key
//Output: position of key in array A or -1 if key is not in A
	Cost times

l ← 0
r ← n − 1
while l ≤ r **do**
    m ← (l + r)/2
    if A[m] = key
        return m
    else if key < A[m]
        r ← m − 1
    else l ← m + 1
return −1
	$C(n)$

Binary Search Analysis

- For simplicity, we count the so-called **three-way comparisons.**

→ *This assume that after one comparison of key with A[m], the algorithm can determine whether key is smaller, equal to, or larger than A[m]*

- Number of comparisons depends not only on n but also on the specifics of the problem.
- Worst case: $C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1.$
  - → Solving this recurrence for $n = 2^k$ gives:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil.$$

- Order of growth: $\boldsymbol{O(\log_2 n)}$

Binary Search – Analysis

- Each time searching is performed, the size of the array reduces ½.
- How many iterations are executed before *left > right*?
  - After 1st iteration: $n/2$ elements remaining.
  - After 2nd iteration: $n/4$ elements remaining.
  - After kth iteration: $n/2^k$ elements remaining.
  - Worst case: when $n/2^k \geq 1$ and $n/2^k+1 < 1$

  → $\log_2 n - 1 < k \leq \log_2 n$ → $O(\log_2 n)$

Binary Search – Analysis

- Best-case: $O(1)$
- Average-case: $O(\log_2 n)$
- Worst-case: $O(\log_2 n)$

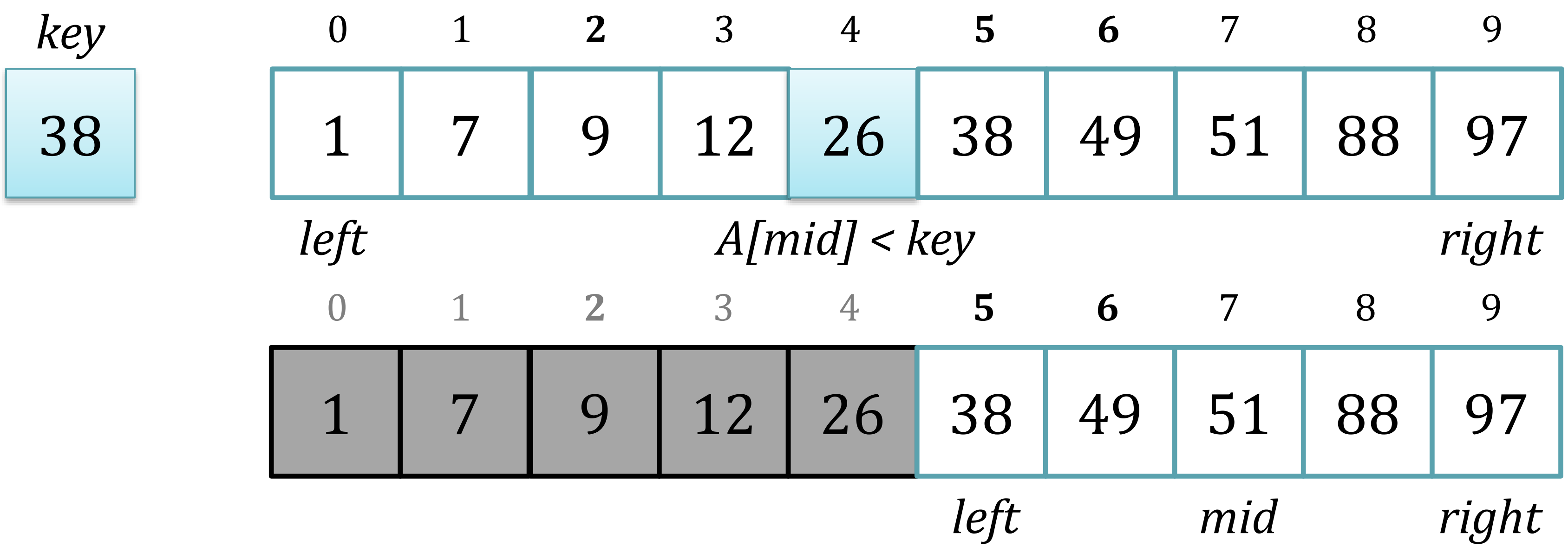
Binary Search – Example

key

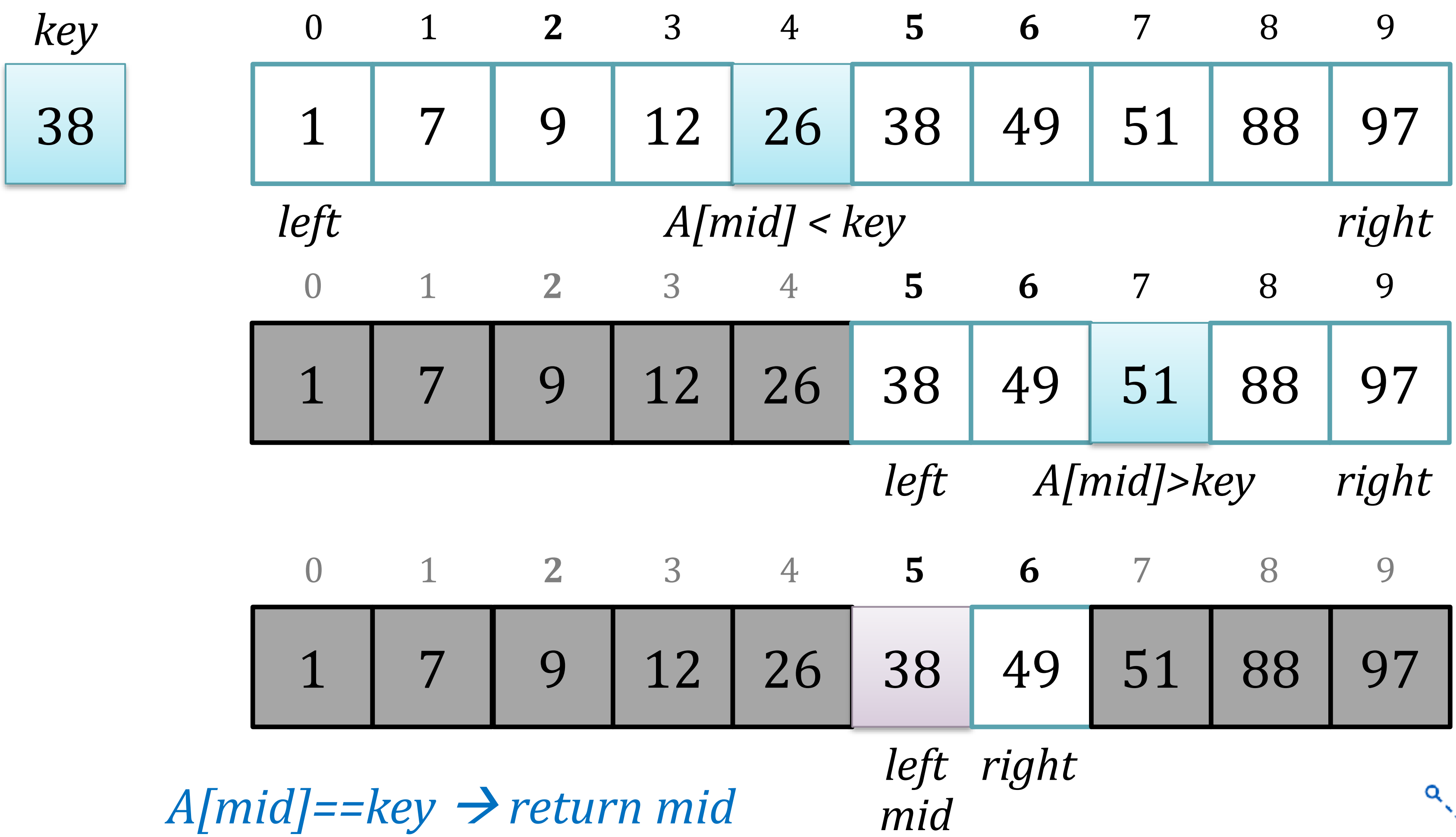
38

0	1	**2**	3	4	**5**	**6**	7	8	9
1	7	9	12	26	38	49	51	88	97
left				*mid*					*right*

Binary Search – Example



Binary Search – Example



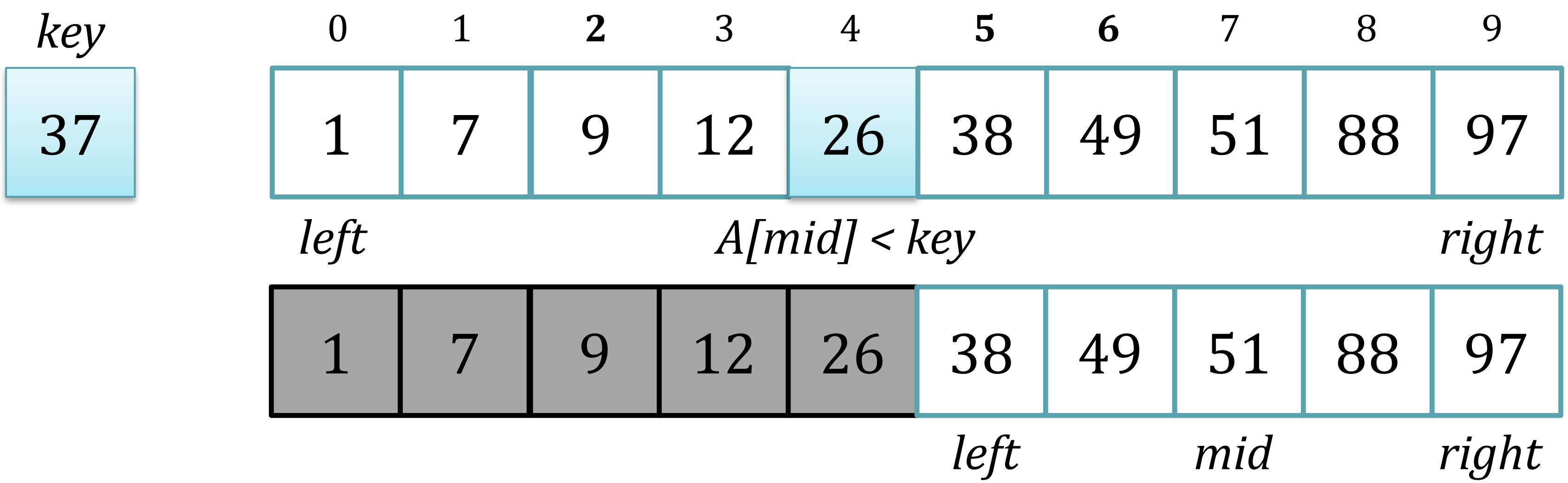
Binary Search – Example

key

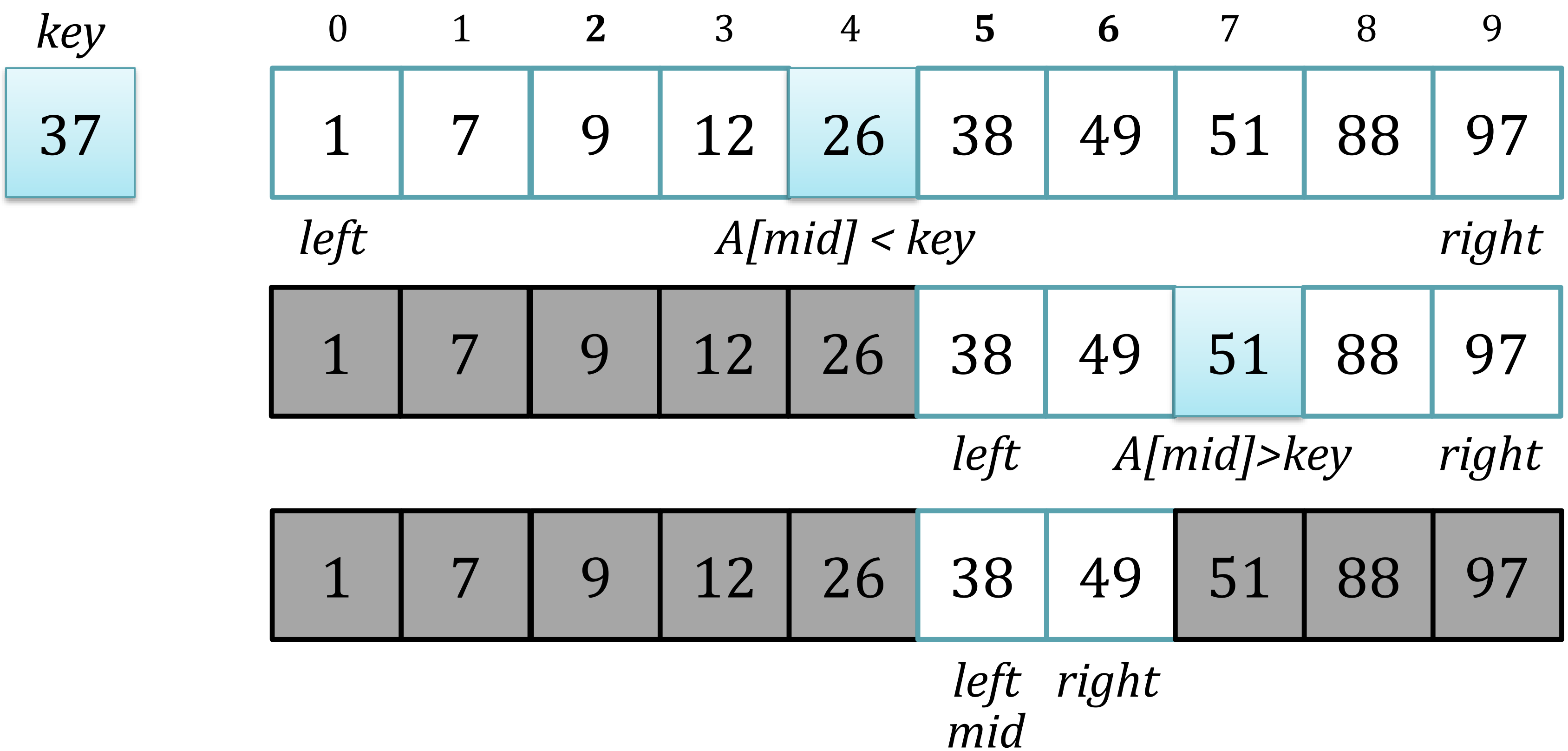
37

0	1	**2**	3	4	**5**	**6**	7	8	9
1	7	9	12	26	38	49	51	88	97
left				*mid*					*right*

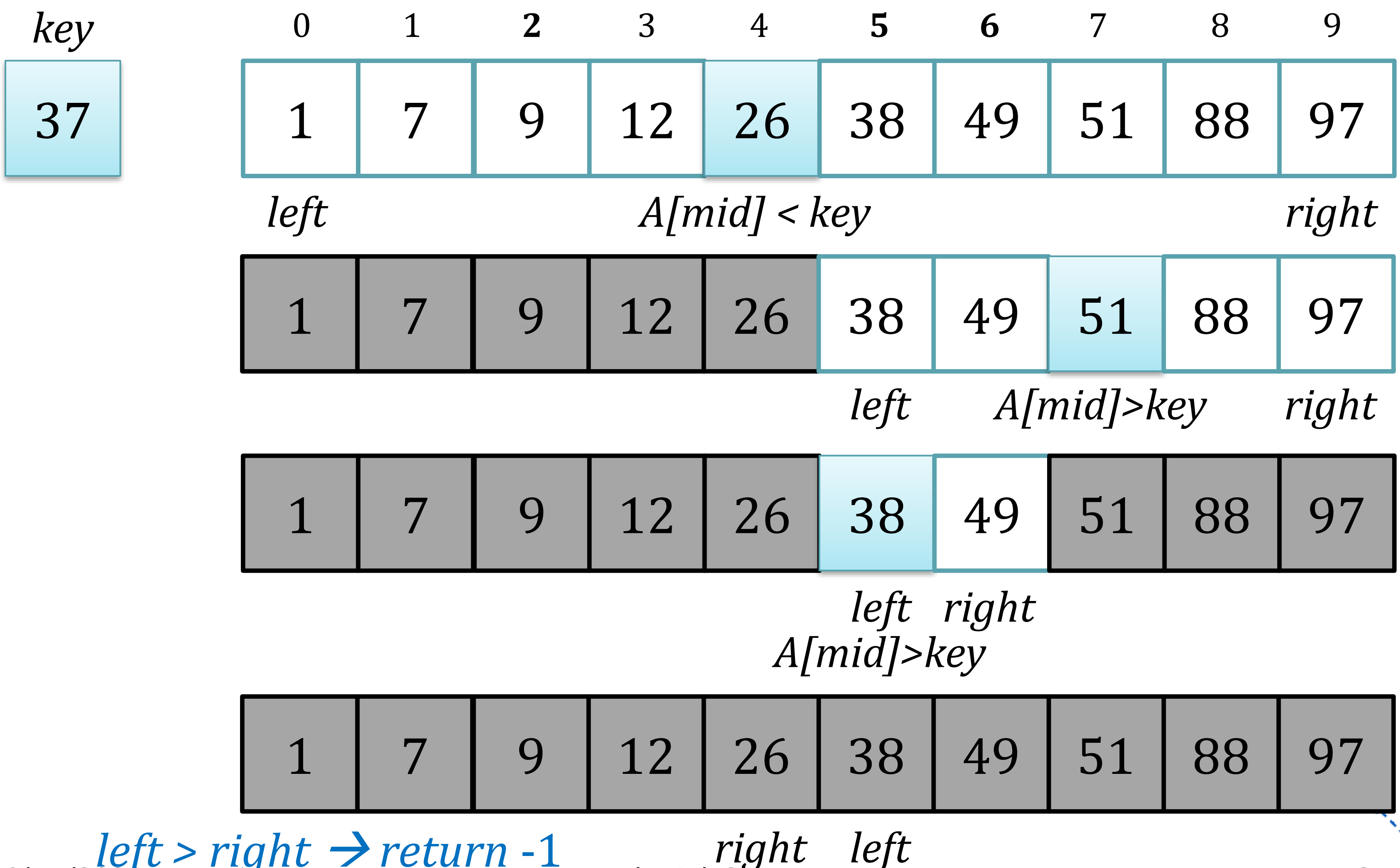
Binary Search – Example



Binary Search – Example



Binary Search – Example



Binary Search – Conclusion

- **Pros:**
  - Take use of the order of the elements in the array to reduce the search area.
  - Very fast!
- **Cons:**
  - The array must be sorted first.
  - Need to consider the sorting time.

Sequential Search vs Binary Search

- Worst-cases:

n	#Basic operation	
	Sequential search	Binary search
100.000	100.000	16
200.000	200.000	17
400.000	400.000	18
800.000	800.000	19
1.600.000	1.600.000	20

Conclusion

- When choosing between binary and sequential search, you must take into consideration the requirement that binary search needs a **sorted array** as input.
- If the program requires the data in the array to **change often**, you will need to sort the array again if its elements have changed prior to doing another search.

Conclusion

- Sequential search is good:
  - For arrays with a small number of elements
  - When you will not search the array many times
  - When the values in the array will be changed
- Binary search is good:
  - For arrays with a large number of elements
  - When the values in the array are less likely to change (cost of maintaining sorted order)

What's next?

- After today:
  - Read textbook 3 – 3.2 & 4.4
  - Do Homework 3
- Next week:
  - Individual Assignment 2 (topic: Sorting & Searching)
  - Lecture 4: Data Structures
    - Basic Concepts
    - Linked List, Stack, Queue Review
    - Hash Table

Q&A