

Searching in simple structures (Linear search, logarithmic search, hash tables, min-max search, and linear-time selection)

Data Structures and algorithms

Attila Házy

2026

Learning goals

After this lecture, students should be able to:

- explain linear search and the sentinel optimization;
- describe logarithmic search methods in sorted arrays;
- define hash tables and distinguish direct addressing, chaining, and open addressing;
- choose and evaluate basic hash functions;
- compare linear, quadratic, and double-hashing probing;
- find minimum and maximum efficiently;
- explain the linear-time selection algorithm based on median of medians.

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection

Searching in an unsorted array

Situation

The elements are stored in an array, but no ordering of the keys is assumed.

Consequences:

- the array positions are directly accessible;
- nevertheless, direct access does not help if we do not know where the key may be;
- every element may have to be tested;
- the worst-case running time is linear.

$$T(n) = \Theta(n).$$

Standard linear search

Idea

Start at the first element and move right until the key is found or the array ends.

```
LINEAR_SEARCH(A, k, x)
1  x ← head[A]
2  IF length[A] > 0 THEN
3      WHILE x ≤ end[A] AND key[A[x]] ≠ k DO
4          INC(x)
5      IF x > end[A] THEN
6          x ← NIL
7  RETURN(x)
```

Worst case: the searched key is not present, so all n elements are examined.

Why the lower bound is linear

For an unsorted array, no comparison can be skipped safely.

Adversarial viewpoint

Until an element has been examined, it may still be the searched key.

Therefore, in the worst case:

- all keys must be compared with k ;
- the time complexity is $\Theta(n)$;
- improvement is possible only in constant factors, not in asymptotic order.

A constant-factor improvement

The standard loop checks two things repeatedly:

- ① whether the current index is still inside the array;
- ② whether the current key equals the searched key.

A common trick is to append a temporary copy of the searched key to the end of the array.

Sentinel idea

The temporary key guarantees that the search will eventually stop because a match is always found.

Linear search with sentinel

```
SENTINEL_LINEAR_SEARCH(A, k, x)
```

```
1  x ← head[A]
2  INC(end[A])
3  key[A[end[A]]] ← k
4  WHILE key[A[x]] != k DO
5      INC(x)
6  DEC(end[A])
7  IF x > end[A] THEN
8      x ← NIL
9  RETURN(x)
```

- The loop no longer tests the end condition.
- The running time remains $\Theta(n)$.
- The hidden constant is smaller in many implementations.

Standard search vs. sentinel search

Property	Standard search	Sentinel search
Array order required	no	no
Worst-case order	$\Theta(n)$	$\Theta(n)$
Loop tests	boundary + key	key only
Extra space	none	one temporary slot
Main advantage	simple	lower constant factor

Outline

- 1 Linear search
- 2 Logarithmic search**
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection

Searching in a sorted array

If the array is sorted by increasing keys, the search can use order information.

Binary search principle

Compare the searched key with the middle element and discard one half of the remaining interval.

Each step reduces the search interval by a factor of approximately 2:

$$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots$$

Therefore:

$$T(n) = \Theta(\log n).$$

Fibonacci search

Motivation

Fibonacci search is a logarithmic search method for sorted arrays. It divides the remaining interval using Fibonacci numbers instead of exact halves.

Assume for the basic version that:

$$n + 1 = F_{k+1}.$$

Initial values:

$$i = F_k, \quad p = F_{k-1}, \quad q = F_{k-2}.$$

The method compares k with $\text{key}[A[i]]$ and then moves left or right according to Fibonacci offsets.

Fibonacci search steps

- 1 Compare the target key with $\text{key}[A[i]]$.
- 2 If the target is smaller, decrease the index by a Fibonacci offset.
- 3 If the target is larger, increase the index by a Fibonacci offset.
- 4 If the key is equal, the search succeeds.
- 5 If the remaining Fibonacci offset becomes zero, the search fails.

Complexity

Fibonacci search also has logarithmic running time:

$$T(n) = \Theta(\log n).$$

Linear vs. logarithmic search

Method	Required condition	Worst-case time
Linear search	no ordering	$\Theta(n)$
Sentinel search	no ordering, extra slot	$\Theta(n)$
Binary search	sorted array	$\Theta(\log n)$
Fibonacci search	sorted array	$\Theta(\log n)$

The price of logarithmic search is maintaining sorted order, which may make insertion and deletion expensive.

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables**
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection

Why hash tables?

Problem

The universe of possible keys may be enormous, while only a small subset is actually stored.

Example:

- at most 5000 employee records;
- a 9-digit identifier gives up to 10^9 possible keys;
- allocating one position for every possible key is wasteful.

A hash table stores the records in a much smaller array and computes positions using a **hash function**.

Hash table terminology

- The array is called a **hash table**.
- Its positions are often called **slots**.
- The table is indexed from 0 to $m - 1$.
- A **hash function** maps keys to slots:

$$h : U \rightarrow \{0, 1, \dots, m - 1\}.$$

- A **collision** occurs when two distinct keys map to the same slot.

Operations supported efficiently

Hash tables are designed mainly for:

- search;
- insertion;
- deletion;
- modification of records.

Important note

If the key field of a record is changed, the record must usually be deleted first and then inserted again according to the new key.

Operations such as finding the minimum key are not the main strength of hash tables.

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining**
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection

Direct address tables

Definition

A **direct address table** can be used when the key universe is small enough:

$$U = \{0, 1, \dots, M - 1\}.$$

If $m = M$, the key itself can serve as the array index.

Empty positions contain NIL.

search, insert, delete: $\Theta(1)$.

Direct addressing algorithms

DIRECT_ADDRESS_SEARCH(T, k, x)

1 $x \leftarrow T[k]$

2 RETURN(x)

DIRECT_ADDRESS_INSERT(T, x)

1 $T[\text{key}[x]] \leftarrow x$

2 RETURN

DIRECT_ADDRESS_DELETE(T, x)

1 $T[\text{key}[x]] \leftarrow \text{NIL}$

2 RETURN

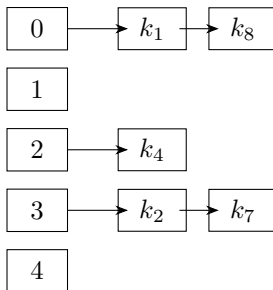
These algorithms are simple because no collisions are possible when every key has its own position.

Chained hash tables

When the key universe is much larger than the table, collisions must be handled.

Chaining

Each table slot stores the head pointer of a linked list. All records whose keys hash to the same slot are stored in that list.



Load factor

Definition

The **load factor** of a hash table is

$$\alpha = \frac{n}{m},$$

where n is the number of stored records and m is the number of slots.

For chained hashing:

- α is the average list length;
- α may be larger than 1;
- average running times depend on α .

Under simple uniform hashing, search time is usually written as:

$$\Theta(1 + \alpha).$$

Chained hashing algorithms

CHAINED_HASH_SEARCH(T , k , x)

- 1 search for key k in list $T[h(k)]$
- 2 return pointer x or NIL

CHAINED_HASH_INSERT(T , x)

- 1 insert x at the beginning of list $T[h(\text{key}[x])]$
- 2 RETURN

CHAINED_HASH_DELETE(T , x)

- 1 delete x from list $T[h(\text{key}[x])]$
- 2 RETURN

Insertion at the beginning of the list is especially simple and fast.

Coalesced chaining

The notes also mention a variant where records and links are stored inside the table itself.

Main idea

Each slot contains at least a key field and a pointer to the next slot in a list. Lists belonging to different hash positions may grow into each other.

Additional information:

- each slot has an occupancy state;
- an attribute r helps manage free positions;
- in an empty table, $r = m$ and all next pointers are NIL.

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions**
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection

What makes a good hash function?

A good hash function should satisfy two practical requirements.

Uniformity

Every key should be mapped to each slot with approximately equal probability. This helps reduce collisions.

Efficiency

The value $h(k)$ should be very fast to compute, because hashing is used in every operation.

Converting keys to numbers

Keys are often not originally numeric.

Examples:

- characters can be converted using ASCII or Unicode codes;
- uppercase letters can be mapped as $A \mapsto 0, B \mapsto 1, \dots, Z \mapsto 25$;
- a word can be interpreted as a number in a suitable base.

Example

The word ZABA can be interpreted as a base-26 number using letter positions.

Formula

$$h(k) = k \bmod m.$$

Choosing m requires care:

- if m is even, parity patterns may be preserved;
- if m is a power of 2, only the last bits of k matter;
- in practice, a prime table size is often a good choice.

Multiplication method

Formula

$$h(k) = \lfloor m \cdot \{kA\} \rfloor, \quad 0 < A < 1.$$

Here $\{kA\}$ denotes the fractional part of kA .

A classical recommended value is related to the golden ratio:

$$A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887.$$

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing**
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection

Open addressing

In open addressing, all records are stored directly in the table.

Collision handling

If the computed slot is occupied, the algorithm probes other slots according to a probing sequence.

The hash function is extended to two arguments:

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

For a fixed key k :

$$h(k, 0), h(k, 1), \dots, h(k, m - 1)$$

should be a permutation of all table positions.

Open addressing: search

```
OPEN_ADDRESS_HASH_SEARCH(T, k, x)
1  i ← 0
2  REPEAT
3      j ← h(k, i)
4      IF key[T[j]] = k AND status[T[j]] = occupied THEN
5          x ← j
6          RETURN(x)
7      INC(i)
8  UNTIL T[j] = NIL OR i = m
9  x ← NIL
10 RETURN(x)
```

The search follows the same probing sequence that insertion would have used.

Open addressing: insertion

```
OPEN_ADDRESS_HASH_INSERT(T, k, error)
1  i <- 0
2  REPEAT
3      j <- h(k, i)
4      IF status[T[j]] = free OR status[T[j]] = deleted THEN
5          key[T[j]] <- k
6          error <- "successful insertion"
7          RETURN(error)
8      ELSE
9          INC(i)
10 UNTIL i = m
11 error <- "table full"
12 RETURN(error)
```

Deletion in open addressing

When deleting a key from an open-addressed hash table, writing NIL is not safe.

Reason

A NIL slot may incorrectly signal the end of a probing sequence, even though later positions may contain keys inserted after collisions.

Therefore, the slot is marked as:

DELETED.

After many insertions and deletions, many slots may become either occupied or deleted. Then the table should often be rebuilt by rehashing its active keys.

Expected time in open addressing

Let

$$\alpha = \frac{n}{m} < 1$$

be the load factor.

Under the uniform hashing assumption:

$$\begin{aligned}\text{unsuccessful search} &\leq \frac{1}{1 - \alpha}, \\ \text{successful search} &\leq \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}.\end{aligned}$$

As α approaches 1, the table becomes crowded and the expected number of probes grows rapidly.

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods**
- 8 Minimum and maximum
- 9 Linear-time selection

Formula

$$h(k, i) = (h'(k) + ci) \bmod m.$$

Usually $c = 1$, or more generally c is chosen relatively prime to m .

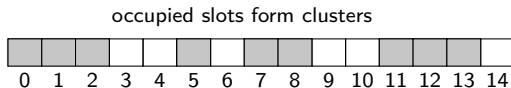
- If a slot is occupied, move to the next slot cyclically.
- It is simple and cache-friendly.
- It suffers from **primary clustering**.

Primary clustering

Definition

A **cluster** is a consecutive sequence of occupied slots along the probing sequence.

Large clusters are harmful because new keys must scan through them before finding a free slot.



Quadratic probing

Formula

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m.$$

Properties:

- the probe positions are not simply consecutive;
- primary clustering is reduced;
- keys with the same initial slot still follow the same probe sequence;
- this remaining problem is called **secondary clustering**.

A triangular-step variant

One possible quadratic-style correction uses cumulative sums:

$$0, 1, 1 + 2, 1 + 2 + 3, \dots$$

Thus the correction term becomes:

$$\frac{i(i+1)}{2}.$$

For suitable table sizes, for example powers of 2, this can visit all slots.

Practical point

The constants and table size must be chosen carefully so that the probing sequence can actually reach every slot.

Double hashing

Formula

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m.$$

Advantages:

- many more probing sequences are possible;
- clustering is reduced compared with linear probing;
- the method often performs very well in practice.

Requirement:

$h_2(k)$ must be relatively prime to m .

Double hashing examples

If m is a power of 2, then $h_2(k)$ should always be odd.

If m is prime, a common choice is:

$$h_2(k) = 1 + (k \bmod (m - 1)).$$

Example with $m = 701$:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod 701.$$

The probe sequence depends strongly on the key, which helps avoid clustering.

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum**
- 9 Linear-time selection

Minimum search

Given n comparable keys in an array, the minimum can be found by scanning once.

Number of comparisons

Finding only the minimum requires exactly $n - 1$ comparisons in the worst case.

```
MINIMUM_SEARCH(A, min)
1  min ← A[1]
2  FOR i ← 2 TO length[A] DO
3      IF min > A[i] THEN
4          min ← A[i]
5  RETURN(min)
```

Finding both minimum and maximum

A simple approach:

- find the minimum using $n - 1$ comparisons;
- then find the maximum among the remaining elements using $n - 2$ comparisons.

Total:

$$(n - 1) + (n - 2) = 2n - 3.$$

This is still linear, but the constant factor is close to 2.

Pairwise min-max method

Compare elements in pairs.

For each pair:

- 1 compare the two elements to each other;
- 2 compare the smaller one with the current minimum;
- 3 compare the larger one with the current maximum.

Comparison counts:

$$\begin{cases} \frac{3n}{2} - 2, & \text{if } n \text{ is even,} \\ \frac{3(n-1)}{2}, & \text{if } n \text{ is odd.} \end{cases}$$

The asymptotic constant is reduced from 2 to $3/2$.

Outline

- 1 Linear search
- 2 Logarithmic search
- 3 Hash tables
- 4 Direct addressing and chaining
- 5 Hash functions
- 6 Open addressing
- 7 Probing methods
- 8 Minimum and maximum
- 9 Linear-time selection**

The selection problem

Definition

Given a set A of n distinct numbers and an index i with $1 \leq i \leq n$, find the element x such that exactly $i - 1$ elements of A are smaller than x .

Special cases:

- $i = 1$: minimum;
- $i = n$: maximum;
- $i = \lceil n/2 \rceil$: median.

Sorting would solve the problem, but it costs $\Theta(n \log n)$ time. Selection can be solved in linear time.

Definition

The **median** is an element occupying the middle position in the sorted order.

If $n = 2k - 1$ is odd, the median has sorted index k .

If $n = 2k$ is even, there are two middle elements:

- lower median: index k ;
- upper median: index $k + 1$.

Unless stated otherwise, the lower median is often used.

Partitioning around a chosen value

The selection algorithm uses a partitioning subroutine.

Partition goal

Given a value x , rearrange the subarray so that:

- elements not greater than x go to the left part;
- elements not smaller than x go to the right part.

This is the same central idea used in quicksort partitioning.

Partition pseudocode

```
PARTITION(A, p, r, x, q)
1  i ← p - 1
2  j ← r + 1
3  WHILE TRUE DO
4      REPEAT j ← j - 1 UNTIL A[j] ≤ x
5      REPEAT i ← i + 1 UNTIL A[i] ≥ x
6      IF i < j THEN
7          swap A[i] and A[j]
8      ELSE
9          q ← j
10     RETURN(A, q)
```

After termination, $A[p], \dots, A[q]$ are not greater than x , and $A[q + 1], \dots, A[r]$ are not smaller than x .

Median of medians: high-level idea

Goal

Choose a pivot that is guaranteed to be reasonably good, not merely good on average.

The algorithm:

- ① split the array into groups of five elements;
- ② find the median of each group;
- ③ recursively find the median of these medians;
- ④ use that value as the pivot;
- ⑤ recurse only into the side that contains the desired order statistic.

Linear-time selection algorithm

SELECT(A, i, x)

- 1 If $n = 1$, then x is the only element; RETURN(x)
- 2 Divide A into groups of 5 elements
- 3 Find the median of each group
- 4 Recursively find the median of the medians: medmed
- 5 Partition A around medmed
- 6 Let k be the number of elements in the lower part
- 7 If $i \leq k$, recursively select the i -th element in the lower part
- 8 Otherwise, recursively select the $(i-k)$ -th element in the upper part
- 9 RETURN(x)

Why groups of five work

The median of medians guarantees that many elements are definitely smaller and many are definitely larger than the pivot.

In each group of five:

- the group median has two elements below it and two above it;
- at least half of the group medians are at least as large as the median of medians;
- this gives a constant fraction of elements on both sides of the pivot.

Therefore the recursive call after partitioning has size at most about:

$$\frac{7n}{10} + O(1).$$

Running time recurrence

The work consists of:

- grouping and finding group medians: $O(n)$;
- recursively selecting the median of $n/5$ medians: $T(n/5)$;
- partitioning: $O(n)$;
- one recursive call on at most $7n/10 + O(1)$ elements.

Thus:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + O(1)\right) + O(n).$$

This recurrence solves to:

$$T(n) = \Theta(n).$$

Summary

- Linear search is optimal for unsorted arrays in the worst case.
- Sentinel search improves the constant factor but not the asymptotic order.
- Sorted arrays allow logarithmic searching.
- Hash tables trade ordering for efficient search, insertion, and deletion.
- Collision handling can use chaining or open addressing.
- Probing method choice strongly affects performance.
- Min-max search can be improved by pairwise comparisons.
- Selection can be solved in worst-case linear time using median of medians.

Complexity overview

Problem / structure	Assumption	Time
Linear search	unsorted array	$\Theta(n)$
Binary / Fibonacci search	sorted array	$\Theta(\log n)$
Direct addressing	small key universe	$\Theta(1)$
Chained hashing	average, load factor α	$\Theta(1 + \alpha)$
Open addressing	uniform hashing, $\alpha < 1$	depends on α
Minimum only	comparable keys	$n - 1$ comparisons
Min and max pairwise	comparable keys	about $3n/2$ comparisons
Selection	median of medians	$\Theta(n)$