

UNIT 9

Searching

(5 Hours) — *Data Structures and Algorithms*

Searching is the process of locating a particular record or key within a collection of data. It is one of the most frequently performed operations in computing, so the choice of search technique has a large impact on overall performance. This unit covers sequential and binary search, searching in trees, hashing with its hash functions and collision resolution techniques, and a comparison of the efficiency of all these methods.

9.1 Introduction to Search Techniques

A search takes a collection of records and a search key, and determines whether a record with that key exists. If it does, the search returns its location (index, pointer, or the record itself); if it does not, the search reports failure. A search that locates the key is called a successful search; one that does not is an unsuccessful search.

9.1.1 Essentials of Searching

Before choosing a search technique, several factors must be considered. These determine which algorithms are even applicable, and which will perform best.

Essential Factor	Why it matters
Is the data sorted?	Binary search requires sorted data; sequential search does not
Size of the data set (n)	For small n even a linear scan is fine; for large n the growth rate dominates
Data structure used	Arrays allow direct indexing; linked lists permit only sequential access
Storage location	Data in main memory vs. on disk changes what is expensive (comparisons vs. block reads)
Frequency of searching	Frequent searching justifies the cost of first sorting or building an index
Frequency of insertion / deletion	A structure that is cheap to search may be costly to update, and vice versa
Internal vs. external searching	Internal = all data fits in memory; external = data resides on secondary storage

9.1.2 Measuring Search Efficiency

The efficiency of a search algorithm is measured by the number of key comparisons it performs, expressed using big-O notation as a function of the number of elements n . Three cases are normally distinguished:

- Best case — the fewest comparisons possible (e.g. the key is the very first element examined).
- Average case — the expected number of comparisons over all possible inputs.
- Worst case — the maximum number of comparisons (typically when the key is absent).

Note: Because the worst case gives a guarantee that performance will never be worse than a stated bound, it is usually the most important measure when selecting an algorithm for a real system.

9.2 Sequential (Linear) Search

Sequential search, also called linear search, is the simplest search technique. It examines each element of the collection one after another, starting from the first, and compares it with the search key. The search stops as soon as a match is found, or when the end of the collection is reached.

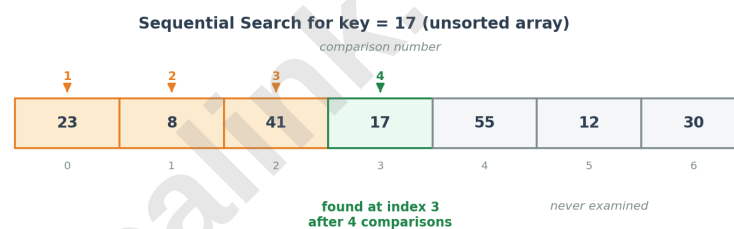


Fig 9.1 — Sequential search examines elements one by one until the key is found; elements after the match are never examined.

```
int sequentialSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i; /* found – return position */
    }
    return -1; /* key not present */
}
```

9.2.1 Analysis of Sequential Search

Case	Situation	Comparisons	Complexity
Best case	Key is the first element	1	$O(1)$
Average case	Key is somewhere in the middle	$(n + 1) / 2$	$O(n)$

Case	Situation	Comparisons	Complexity
Worst case	Key is the last element, or absent	n	O(n)

- Advantages — very simple to implement; works on unsorted data; works on any structure that supports sequential access, including linked lists; no preprocessing required.
- Disadvantages — very slow for large data sets, since cost grows linearly with n; ignores any ordering the data may already have.

Example 9.1 — Sequential search comparisons

An array holds 1000 unsorted elements. How many comparisons are needed in the best, average and worst cases?

Solution

Best case: the key is at index 0 → 1 comparison.

Average case: $(n + 1) / 2 = (1000 + 1) / 2 \approx 500$ comparisons.

Worst case: the key is last or absent → 1000 comparisons.

9.3 Binary Search

Binary search is a far more efficient technique, but it has one strict requirement: the data must already be sorted. It works on the divide-and-conquer principle — the key is compared with the middle element, and because the data is ordered, one half of the remaining range can be discarded after every single comparison.

- Compute the middle index: $\text{mid} = (\text{low} + \text{high}) / 2$.
- If $\text{arr}[\text{mid}]$ equals the key, the search succeeds.
- If $\text{arr}[\text{mid}]$ is less than the key, the key can only be in the right half, so set $\text{low} = \text{mid} + 1$.
- If $\text{arr}[\text{mid}]$ is greater than the key, the key can only be in the left half, so set $\text{high} = \text{mid} - 1$.
- Repeat until the key is found or low exceeds high (key absent).

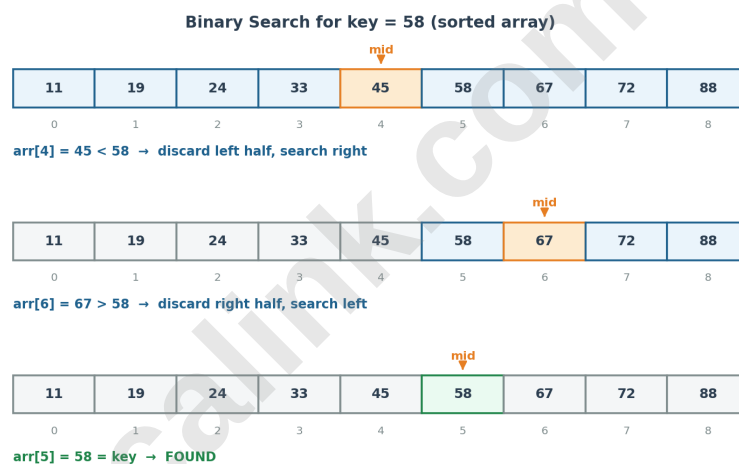


Fig 9.2 — Binary search for 58: each comparison eliminates half of the remaining elements, finding the key in 3 comparisons.

```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2; /* avoids overflow */
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1; /* search right half */
        else
            high = mid - 1; /* search left half */
    }
    return -1; /* key not present */
}
```

9.3.1 Analysis of Binary Search

Each comparison halves the search space. After k comparisons, at most $n / 2^k$ elements remain, so the search ends when $n / 2^k = 1$, giving $k = \log_2 n$. Binary search therefore runs in $O(\log n)$ time.

Case	Situation	Complexity
Best case	Key is the middle element on the first comparison	$O(1)$
Average case	Key is found after about $\log_2 n$ comparisons	$O(\log n)$
Worst case	Search narrows to a single element, or key is absent	$O(\log n)$

Note: The power of the logarithm is striking: searching one million sorted elements takes at most 20 comparisons with binary search, but up to one million with sequential search.

- Advantages — extremely fast, $O(\log n)$; simple to implement either iteratively or recursively.
- Disadvantages — requires sorted data, and sorting costs $O(n \log n)$ if the data is not already ordered; requires direct (random) access, so it cannot be applied to a linked list; insertions and deletions are expensive because the sorted order must be maintained.

Example 9.2 — Tracing a binary search

Search for key = 58 in the sorted array: 11, 19, 24, 33, 45, 58, 67, 72, 88 (indices 0–8).

Solution

Pass 1: low = 0, high = 8 $\rightarrow$ mid = 4, arr[4] = 45. Since $45 < 58$, set low = 5.

Pass 2: low = 5, high = 8 $\rightarrow$ mid = 6, arr[6] = 67. Since $67 > 58$, set high = 5.

Pass 3: low = 5, high = 5 $\rightarrow$ mid = 5, arr[5] = 58 = key $\Rightarrow$ found at index 5.

Total: 3 comparisons, compared with 6 for a sequential search.

9.4 Tree Search

Tree search applies the same halving idea as binary search, but to data stored in a Binary Search Tree (BST) rather than a sorted array. Recall the BST property: for every node, all keys in the left subtree are smaller and all keys in the right subtree are larger. This ordering lets the search discard an entire subtree at each step.

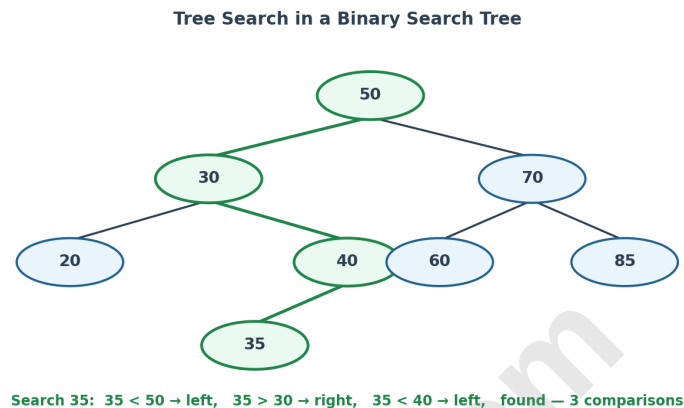


Fig 9.3 — Searching a BST: each comparison selects one subtree and discards the other entirely.

```
TNODEPTR treeSearch(TNODEPTR root, int key) {
    while (root != NULL) {
        if (key == root->key) return root; /* found */
        else if (key < root->key)
            root = root->left;
        else
            root = root->right;
    }
    return NULL; /* key not present */
}
```

The number of comparisons equals the number of levels traversed, so the cost of a tree search is proportional to the height of the tree. For a reasonably balanced BST the height is $O(\log n)$, giving $O(\log n)$ search. However, if keys are inserted in sorted order the tree becomes skewed and degenerates into a structure resembling a linked list, pushing the search cost to $O(n)$.

Tree shape	Height	Search complexity
Balanced (e.g. AVL, Red-Black)	$O(\log n)$	$O(\log n)$ guaranteed
Randomly built BST	About $1.39 \log_2 n$ on average	$O(\log n)$ on average
Completely skewed BST	$n - 1$	$O(n)$ — no better than sequential search

Note: This is exactly why self-balancing trees such as AVL trees matter: they keep the height at $O(\log n)$ no matter what order the keys arrive in, guaranteeing efficient searching in the worst case.

9.4.1 Tree Search vs. Binary Search

Both achieve $O(\log n)$, but they suit different situations. Binary search on an array has no pointer overhead and excellent cache behaviour, but insertions and deletions require shifting elements, costing $O(n)$. A BST supports insertion and deletion in $O(\log n)$, making it far better for data that changes frequently.

9.5 General Search Tree

A general search tree (also called a multiway search tree) relaxes the binary restriction: a node may hold several keys and have several children. A node containing k keys has $k + 1$ child pointers, and the keys inside the node partition the range of values among those children.

- Keys within each node are stored in ascending order.
- All keys in the subtree pointed to by the first child are smaller than the node's first key.
- All keys in the subtree between two consecutive keys lie between those two key values.
- All keys in the subtree pointed to by the last child are larger than the node's last key.

The best-known general search trees are the B-Tree and its variant the B+ Tree, together with the 2-3 tree and 2-3-4 tree. Because each node holds many keys, the tree becomes very shallow — a B-Tree indexing millions of records is typically only three or four levels deep.

Property	Binary Search Tree	General (Multiway) Search Tree
Keys per node	Exactly 1	Many ($m - 1$ for order m)
Children per node	At most 2	Many (up to m)
Height for n keys	$O(\log_2 n)$	$O(\log_m n)$ — much shallower
Nodes visited per search	Many (one per level)	Few, since the tree is shallow
Best suited for	Data held in main memory	Data held on disk / external storage
Typical use	In-memory dictionaries and sets	Database indexes, file systems

Note: Searching a general search tree costs $O(\log_m n)$ node accesses but requires a small search within each node. On disk this trade is very favourable, because reading a block is far slower than comparing keys already held in memory.

9.6 Hashing

All the techniques so far locate a key by comparing it with keys already stored, so their cost depends on n . Hashing takes a fundamentally different approach: it computes the storage location directly from the key itself, using a hash function. Ideally no searching is needed at all — the address is calculated in one step, giving $O(1)$ average-case access.

9.6.1 Hash Function and Hash Table

A hash table is an array of m positions called slots or buckets. A hash function $h(k)$ maps a key k to an index in the range 0 to $m - 1$, which is where that key's record is stored. The same function is applied when searching, so the record is located directly.

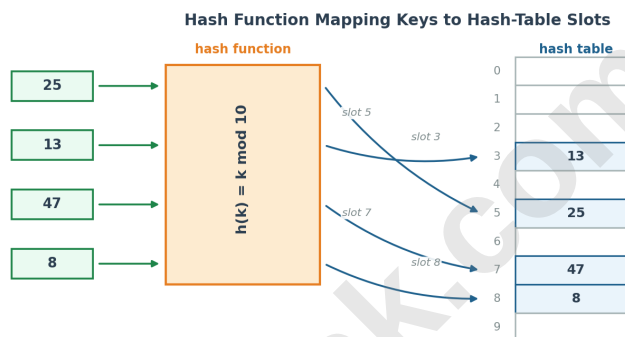


Fig 9.4 — A hash function maps each key to a slot in the hash table, allowing direct access without comparisons.

9.6.2 Common Hash Functions

Method	How it works	Example
Division method	$h(k) = k \bmod m$, where m is the table size (best chosen as a prime number)	$k = 47, m = 10 \rightarrow \text{slot } 7$
Mid-square method	Square the key and extract a fixed number of middle digits	$k = 34 \rightarrow 1156 \rightarrow \text{take '15'} \rightarrow \text{slot } 15$
Folding method	Split the key into equal-sized parts and add them together	$k = 123456 \rightarrow 12+34+56 = 102 \rightarrow \text{slot } 102 \bmod m$
Multiplication method	$h(k) = \text{floor}(m \times (k \times A \bmod 1))$ for a constant A between 0 and 1	Less sensitive to the choice of m
Digit extraction	Select specific digit positions from the key	$k = 942617 \rightarrow \text{take digits } 1,3,5 \rightarrow 921$

A good hash function should be quick to compute, should distribute keys as uniformly as possible across all slots to avoid clustering, and should use every part of the key so that similar keys do not map to the same slot.

9.6.3 Collisions and the Load Factor

Because the number of possible keys is usually far larger than the table size, two distinct keys may hash to the same slot. This is called a collision, and it is unavoidable in practice — so every hashing scheme needs a collision resolution technique.

The load factor is defined as $\alpha = n / m$, where n is the number of stored keys and m is the table size. It measures how full the table is, and directly controls how often collisions occur: as α approaches 1, collisions become frequent and performance degrades.

9.7 Collision Resolution Techniques

There are two broad families of collision resolution: separate chaining, which stores colliding keys outside the table, and open addressing, which finds another empty slot inside the table itself.

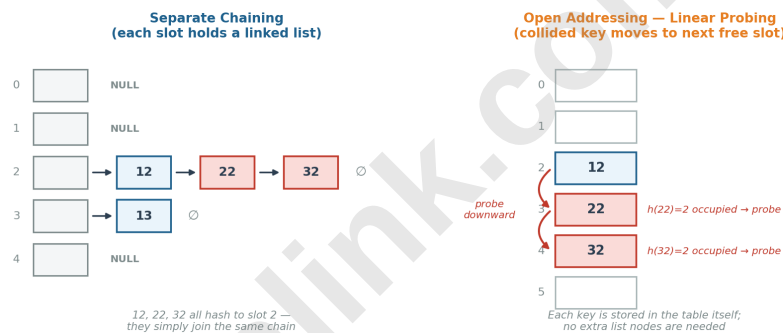


Fig 9.5 — Separate chaining keeps colliding keys in a linked list; open addressing places them in other free slots of the table.

9.7.1 Separate Chaining (Open Hashing)

Each slot of the hash table holds a pointer to a linked list of all keys that hash to that slot. Inserting simply appends to the list; searching computes the slot and then scans only that short list.

- Advantages — simple to implement; the table can never overflow, since chains grow as needed; deletion is straightforward; performs well even when the load factor exceeds 1.
- Disadvantages — extra memory is needed for the pointers; poor cache performance because list nodes are scattered in memory; if the hash function is bad, one chain can become very long, degrading to $O(n)$.

9.7.2 Open Addressing (Closed Hashing)

All keys are stored inside the table itself. When a collision occurs, the algorithm probes a sequence of alternative slots until an empty one is found. The three standard probing strategies differ in how the next slot is chosen.

Technique	Probe sequence for the i-th attempt	Characteristic
Linear probing	$h(k, i) = (h(k) + i) \bmod m$	Simple, but causes primary clustering — long runs of filled slots
Quadratic probing	$h(k, i) = (h(k) + c_1i + c_2i^2) \bmod m$	Avoids primary clustering, but suffers secondary clustering
Double hashing	$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$	Uses a second hash function; best distribution, minimal clustering

- Advantages — no pointer overhead, so memory use is lower; better cache performance because all data sits in one contiguous array.
- Disadvantages — the table can become full; performance degrades badly as the load factor approaches 1; deletion is awkward, since simply emptying a slot would break the probe sequences of other keys (a special “deleted” marker is normally used instead).

Note: For open addressing the load factor must be kept well below 1 — commonly under 0.7 — and the table is rehashed into a larger array when this threshold is exceeded.

Example 9.3 — Resolving collisions by linear probing

Insert the keys 12, 22, 32 into a hash table of size 10 using $h(k) = k \bmod 10$ with linear probing.

Solution

$h(12) = 12 \bmod 10 = 2 \rightarrow$ slot 2 is empty, so 12 is placed there.

$h(22) = 22 \bmod 10 = 2 \rightarrow$ slot 2 is occupied. Probe slot 3, which is free $\rightarrow$ 22 is placed at slot 3.

$h(32) = 32 \bmod 10 = 2 \rightarrow$ slots 2 and 3 are occupied. Probe slot 4, which is free $\rightarrow$ 32 is placed at slot 4.

Searching for 32 repeats the same probe sequence: check slot 2 (12, no), slot 3 (22, no), slot 4 (32 — found).

9.8 Efficiency Comparison of Search Techniques

The table below summarises the performance and requirements of every technique covered in this unit. No single method is best in all circumstances — the right choice depends on whether the data is sorted, how often it changes, and where it is stored.

Technique	Best	Average	Worst	Data must be sorted?	Extra space
Sequential search	$O(1)$	$O(n)$	$O(n)$	No	$O(1)$
Binary search	$O(1)$	$O(\log n)$	$O(\log n)$	Yes	$O(1)$
Tree search (balanced BST)	$O(1)$	$O(\log n)$	$O(\log n)$	Ordered structure	$O(n)$ for pointers
Tree search (skewed BST)	$O(1)$	$O(n)$	$O(n)$	Ordered structure	$O(n)$ for pointers
General search tree (B-Tree)	$O(1)$	$O(\log_m n)$	$O(\log_m n)$	Ordered structure	$O(n)$
Hashing (chaining)	$O(1)$	$O(1)$	$O(n)$	No	$O(n)$ for chains
Hashing (open addressing)	$O(1)$	$O(1)$	$O(n)$	No	$O(m)$ table

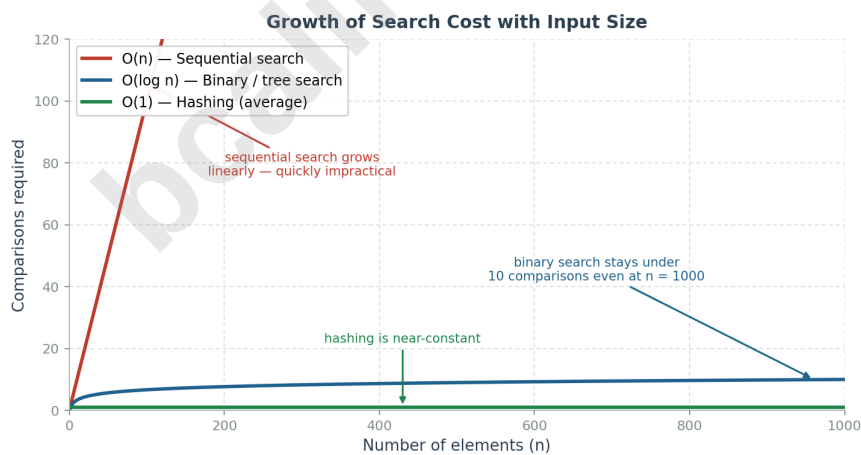


Fig 9.6 — Growth of the number of comparisons with input size for the three main complexity classes.

9.8.1 Choosing the Right Technique

Situation	Recommended technique
Small data set, or data is unsorted and searched rarely	Sequential search — simplest, no preprocessing
Large sorted array that changes rarely	Binary search — $O(\log n)$ with no extra memory

Situation	Recommended technique
Data in a linked list	Sequential search — binary search needs random access
Data changes frequently and must stay ordered	Balanced tree (AVL) — $O(\log n)$ search, insert and delete
Only exact-match lookups, order not needed	Hashing — $O(1)$ average access
Range queries or sorted traversal required	Tree search — hashing cannot answer range queries
Very large data held on disk	B-Tree / B+ Tree — minimises slow disk accesses

Note: Hashing gives $O(1)$ average access, but it cannot answer questions such as “find all keys between 20 and 40” or “list the keys in order”, because a hash function deliberately scatters related keys. When ordering matters, a search tree is the correct choice despite its $O(\log n)$ cost.

Key Takeaways

Unit 9 — Key Takeaways

- ✓ Searching locates a record by its key; the choice of technique depends on whether the data is sorted, its size, the structure holding it, and how often it changes.
- ✓ Sequential search checks elements one by one — $O(n)$, but it needs no sorting and works on any structure including linked lists.
- ✓ Binary search repeatedly halves a sorted array, achieving $O(\log n)$, but it requires sorted data and random access.
- ✓ Tree search on a BST is $O(\log n)$ when the tree is balanced, but degrades to $O(n)$ if the tree becomes skewed — which is why AVL and other balanced trees exist.
- ✓ A general (multiway) search tree stores many keys per node, keeping the tree very shallow and minimising the number of node accesses — ideal for disk-based data.
- ✓ Hashing computes an address directly from the key using a hash function, giving $O(1)$ average-case access instead of comparing keys.
- ✓ Collisions are unavoidable and are resolved either by separate chaining (linked lists per slot) or open addressing (linear probing, quadratic probing, double hashing).
- ✓ Hashing is fastest for exact-match lookups, but only a search tree can support ordered traversal and range queries.

— End of Unit 9 —