FREE EDITION · NOTES + 3 SAMPLE MCQS

CUET · COMPUTER SCIENCE · CLASS XII · CODE 308

Searching

CUET unit: Searching

By **UniDrill** · NCERT-grounded study material

WWW.UNIDRILL.IN

📌 Snapshot

- Three core searching techniques exist — Linear Search, Binary Search, and Search by Hashing — each with distinct preconditions, algorithms, and efficiency characteristics.
- Linear search works on any list (sorted or unsorted); binary search requires a sorted list; hashing achieves near-constant-time search by computing an index directly.
- Searching algorithms appear in algorithm analysis questions, Python program tracing questions, and complexity/comparison-count questions, so CUET tests them heavily.
- Fully worked Python implementations (Programs 6-1, 6-2, 6-3) make this fertile ground for code-output and dry-run MCQs.
- Understanding collision and collision resolution in hashing is a key conceptual distinction that NTA frequently probes with distractor-heavy options.

📖 Detailed Notes

2.1 Core concepts

- **Searching defined:** Searching means locating a particular element (called the key) in a collection of elements. The search result determines whether the key is present or not; if present, it also finds the key's position. (NCERT §6.1, p. 81)
- **Linear Search overview:** Linear search (also called sequential search or serial search) is the most fundamental and simplest method. Every element of the list is compared with the key one by one, starting from the first element and moving towards the last. If a match is found the search is successful; if the entire list is traversed without a match, the search is unsuccessful. It works on both sorted and unsorted lists and is useful for small, unordered collections. (NCERT §6.2, p. 82)
- **Linear Search algorithm (Algorithm 6.1):** LinearSearch(numList, key, n) — initialise index = 0; while index < n, compare numList[index] with key; if equal, print position (index+1) and stop; else increment index; after loop, print "Search unsuccessful". (NCERT §6.2, p. 82)
- **Linear Search comparisons — best case:** If the key is the first element, only 1 comparison is needed. This is the minimum work. (NCERT §6.2, p. 83)

- **Linear Search comparisons — worst case:** If the key is the last element or not present, the algorithm makes n comparisons (where n is the number of elements). This is the maximum work. (NCERT §6.2, p. 83)
- **Program 6-1 (Python — Linear Search):** The `linearSearch(list, key)` function iterates using `range(0, len(list))`, returns `index+1` when the key is found, and returns `None` if not found. The calling code checks `if position is None` to decide output. (NCERT §6.2, p. 84)
- **Binary Search overview:** Binary search makes use of the ordering (sorted order) of elements in the list to search efficiently. The key is compared with the middle element of the sorted list, yielding three possibilities: (i) middle element equals key — search successful; (ii) middle element is greater than key — search the first half; (iii) middle element is smaller than key — search the second half. The list size is halved at each step, hence the name. (NCERT §6.3, p. 85)
- **Binary Search prerequisite:** The list must be sorted (ascending or descending for numeric data; alphabetically for textual data) before binary search can be applied. (NCERT §6.3, p. 85)
- **Mid-point calculation:** For a list with an even number of elements, mid = (first + last) // 2 using floor division. For a 10-element list, mid = 10 // 2 = 5, meaning the sixth element (index 5) is the middle. (NCERT §6.3, p. 86)
- **Binary Search algorithm (Algorithm 6.2):** BinarySearch(numList, key) — SET first = 0, last = n-1; calculate mid = (first+last)//2; WHILE first <= last: if numList[mid] == key, print position (mid+1) and STOP; elif numList[mid] > key, set last = mid-1; else set first = mid+1; after loop, print "Search unsuccessful". (NCERT §6.3, p. 87)
- **Binary Search best case:** Only 1 iteration is needed if the key is the middle element of the list. (NCERT §6.3, p. 87)
- **Binary Search worst case:** Maximum iterations are required when the key is the first element or not in the list; for a 15-element list, 4 iterations were required (list reduced: 15 → 7 → 3 → 1 elements). (NCERT §6.3, p. 88)
- **Program 6-2 (Python — Binary Search):** The `binarySearch(list, key)` function uses `while first <= last`, computes `mid = (first + last)//2`, returns `mid` on match, sets `first = mid + 1` if key > list[mid], sets `last = mid - 1` if key < list[mid], and returns `-1` if not found. (NCERT §6.3, p. 89)
- **Applications of Binary Search (§6.3.1):** Binary search is used for searching a dictionary or telephone directory, finding minimum/maximum in a sorted list; modified binary search techniques are used for indexing in databases, implementing routing tables in routers, and data compression code. (NCERT §6.3.1, p. 90)
- **Search by Hashing overview:** Hashing is a technique that can determine the presence of a key in a list in just one step (one comparison). A formula called the hash function generates an index value for each element and places it in a structure called the hash table. (NCERT §6.4, p. 90)

- **Hash function — remainder method:** h(element) = element % size(hash table). Each element is divided by the hash table size; the remainder is the hash value (index position). (NCERT §6.4, p. 90)
- **Hash table structure:** A hash table is implemented as a Python list with a fixed number of positions (e.g., 10), initialised to None. Each index can hold only one item; positions are indexed from 0. The hash table size can be larger than the original list. (NCERT §6.4, p. 90)
- **Hashing search operation:** To search for a key, compute its hash value and check the element at that index in the hash table. Only one comparison is needed regardless of the list size. (NCERT §6.4, p. 91)
- **Collision:** When two or more elements produce the same hash value (and hence compete for the same index), the situation is called collision. Example: with h(element) = element % 10, both 16 and 26 give hash value 6 — a collision. (NCERT §6.4.1, p. 92)
- **Collision resolution:** The process of finding an alternative slot in the hash table for elements that collide is called collision resolution. Collision resolution methods (e.g., chaining, open addressing) are beyond the scope of this book. (NCERT §6.4.1, p. 92)
- **Perfect hash function:** If every element of the list maps to a unique index in the hash table, the hash function is called a perfect hash function. If the hash function is perfect, collision will never occur. (NCERT §6.4.1, p. 93)
- **Other hash function techniques:** Apart from the remainder method, hash functions may use integer division, shift folding, boundary folding, mid-square function, extraction, radix transformation, etc. (NCERT §6.4.1, p. 93)
- **Time complexity of hashing:** The time taken to compute the hash value is independent of the number of items in the search list. However, the cost of computing the hash function must be small enough to make hashing more efficient than other methods. (NCERT §6.4.1, p. 93)

2.2 Definitions to memorise

Term	Definition	Page
Searching	Locating a particular element (key) in a collection; determines presence/absence and position of the key.	81
Key	The particular element being searched for in a collection.	81
Linear Search	An exhaustive search method that compares each element of the list with the key one by one from first to last; also called sequential or serial search.	82
Binary Search	A search technique that uses the sorted order of a list, repeatedly halving the search area by comparing the key with the middle element.	85
Hashing		90

Term	Definition	Page
	A search technique that uses a hash function to compute the index of an element in a hash table, enabling search in one comparison.	
Hash Function	A formula that takes an element and generates an index value (hash value) indicating its position in the hash table.	90
Hash Table	A list structure where elements are stored at positions determined by the hash function.	90
Remainder Method	A hash function where h(element) = element % size(hash table); the remainder is used as the hash value/index.	90
Collision	The situation when two or more elements produce the same hash value and must occupy the same index in the hash table.	92
Collision Resolution	The process of finding an alternative slot in the hash table for elements that produce the same hash value.	92
Perfect Hash Function	A hash function that maps every input key to a unique index in the hash table, ensuring no collisions ever occur.	93
Sequential search	Synonym for linear search	82
Serial search	Synonym for linear search	82
Best case (search)	Minimum comparisons needed; linear: 1; binary: 1 (key is middle); hashing: 1	83, 87, 91
Worst case (search)	Maximum comparisons; linear: n; binary: $\log_2 n$; hashing: 1 if no collision	83, 88
Mid index	`(first + last) // 2` — used by binary search to split the list	86
Hash value	Output of the hash function — the index in the hash table	90
Integer division `//`	Floor-division operator used to compute mid in binary search	86
Hash table size	Capacity of the hash table — must be at least the number of keys	90
Open addressing	A collision-resolution family (not covered in detail in NCERT)	92
Chaining	Another collision-resolution family (not covered in detail in NCERT)	92
Database indexing	Real-world application of modified binary search	90
Routing tables	Real-world application of modified binary search in routers	90

2.3 Diagrams / processes to remember

- **Table 6.1 & 6.2 (p. 82–83):** Step-by-step trace of linear search for key = 17 in numList [8, -4, 7, 17, 0, 2, 19] showing index, condition check, comparison, and increment at each step. Key is found after 4 comparisons.

- **Table 6.5 & 6.6 (p. 87):** Binary search trace for key = 17 in a 15-element sorted list [2, 3, 5, 7, 10, 11, 12, 17, 19, 23, 29, 31, 37, 41, 43]; shows first, last, mid values and result. Key found in 1 iteration since 17 is the middle element.
- **Table 6.7 (p. 88):** Binary search trace for key = 2 in the same 15-element list; takes 4 iterations, halving the list each time (15 → 7 → 3 → 1 elements).
- **Table 6.8 & 6.9 & 6.10 (p. 90–91):** Hash table construction — empty 10-position table, hash values computed for [34, 16, 2, 93, 80, 77, 51] using element % 10, and final populated hash table showing each element at its hash index.
- **Algorithm 6.1 (p. 82):** Pseudocode for LinearSearch — a WHILE loop with index starting at 0, incrementing until match or exhaustion.
- **Algorithm 6.2 (p. 87):** Pseudocode for BinarySearch — WHILE first <= last, compute mid, branch on three comparison outcomes.

2.4 Common confusions / NTA trap points

- **Binary search requires sorted input:** Students often apply binary search to an unsorted list in MCQs — this is wrong. Binary search is ONLY valid when the list is already sorted. NTA may present an unsorted list and ask which algorithm can be applied.
- **Position vs. index:** The NCERT programs return `index + 1` as the "position" (1-based), but the algorithm internally uses 0-based indices. NTA questions may ask for "position" or "index value" — read carefully. In Program 6-2, the function returns the raw mid index (0-based), while Program 6-1 returns `index + 1`.
- **Mid calculation for even-length lists:** When the list has an even number of elements (say 10), mid = 10 // 2 = 5, which is the 6th element (index 5, not index 4). Students confuse index 5 with "5th element".
- **Hashing comparison count vs. linear/binary:** Hashing requires exactly ONE comparison to search regardless of list size, but only when no collision occurs. NTA may ask about best-case comparisons for all three methods — linear: 1, binary: 1, hashing: 1 — but the conditions differ.
- **Collision vs. collision resolution (NCERT §6.4.1, p. 92).** Collision is the problem; collision resolution is the solution process.
- **Linear search returns 1-based position (NCERT Program 6-1, p. 84).** NCERT returns `index + 1`. NTA may ask for the 0-based index — read carefully.
- **Binary search worst case is $\log_2 n$ iterations (NCERT §6.3, p. 88).** For 15 elements: 4 iterations. For 1024 elements: 10 iterations.
- **Hash function cost matters (NCERT §6.4.1, p. 93).** A complex hash function that takes more time than the search itself defeats the purpose.
- **Perfect hash ≠ realistic (NCERT §6.4.1, p. 93).** In practice, collisions occur; perfect hash is an ideal.

- **Modulus operator gives non-negative remainder for positive ints (NCERT §6.4, p. 90).** `34 % 10 = 4` — always in `[0, size-1]`.
- **Empty list searches (NCERT §6.2-6.4).** All three algorithms return "not found" on empty input.

🎯 Practice MCQs

Q1. Which of the following statements correctly describes Linear Search?

A. It requires the list to be sorted before searching begins.

B. It compares each element of the list with the key one by one, starting from the first element.

C. It divides the list into two halves and searches only the relevant half.

D. It uses a hash function to compute the index of the key.

Q2. What is the maximum number of comparisons linear search will make while searching for a key in a list of n elements?

A. 1

B. n/2

C. $\log_2 n$

D. n

Q3. Consider the following Python function: ```python def linearSearch(list, key): for index in range(0, len(list)): if list[index] == key: return index + 1 return None ``` What does the function return when the key is not present in the list?

A. 0

B. -1

C. None

D. False

PYQ Alignment

Searching techniques appear in CUET Computer Science papers primarily through algorithm-trace questions (counting comparisons in linear or binary search on a given list) and conceptual distinction questions contrasting the three search methods, with hashing collision questions appearing as moderate-difficulty items in recent years. See PYQ archive for Computer Science.