Searching Algorithms & Hashing

Data Structures and Algorithms

Masoud Hamad

State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

1. Introduction to Searching
2. Linear Search
3. Binary Search
4. Search in Rotated Sorted Array
5. Introduction to Hashing
6. Choosing a Hash Function
7. Applications & Summary

Why Searching?

Searching is one of the most fundamental operations in computing:

- Finding a contact in your phone
- Looking up a word in a dictionary
- Searching for a file on your computer
- Database queries
- Web search engines

Key Question: How efficiently can we find an element in a collection?

Search Algorithms Overview

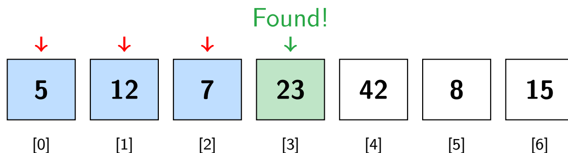
Algorithm	**Time**
Linear Search	$O(n)$
Binary Search	$O(\log n)$
Hash Table	$O(1)$ avg

Linear Search — The Simplest Approach

Idea

Check each element one by one from the beginning until the target is found or the end is reached.

Searching for 23 in:



- Checked indices 0, 1, 2, 3 $\rightarrow$ **4 comparisons**
- Works on **unsorted** arrays — no preprocessing needed

Linear Search — C Implementation

```c
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;        // Found at index i
        }
    }
    return -1;               // Not found
}

int main() {
    int arr[] = {5, 12, 7, 23, 42, 8, 15};
    int n = 7;
    int key = 23;

    int result = linearSearch(arr, n, key);
    if (result != -1)
        printf("Found %d at index %d\n", key, result);
    else
        printf("%d not found\n", key);
    return 0;
}
```

Linear Search — Complexity Analysis

Time Complexity

Case	Comparisons
Best Case	$O(1)$ (first element)
Average Case	$O(n/2) = O(n)$
Worst Case	$O(n)$ (last or not found)

Space Complexity: $O(1)$

When to Use Linear Search

- Small arrays
- Unsorted data
- Searching only once
- Linked lists (no random access)

Advantages

- Simple to implement
- No sorting required
- Works on any data structure

Binary Search — Divide and Conquer

Prerequisite

The array **must be sorted** before using binary search.

Idea

Repeatedly divide the search space in half:

1. Compare target with the **middle** element
2. If equal $\rightarrow$ found!
3. If target $<$ middle $\rightarrow$ search **left half**
4. If target $>$ middle $\rightarrow$ search **right half**

Like looking up a word in a dictionary — you open to the middle,
then go left or right depending on alphabetical order.

Binary Search — Visual Trace

Search for 23 in: [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

Step 1: low=0, high=9, mid=4

2	5	8	12	16	23	38	56	72	91
low				mid					high

$16 < 23$, go right

Step 2: low=5, high=9, mid=7

2	5	8	12	16	23	38	56	72	91
					low		mid		high

$56 > 23$, go left

Step 3: low=5, high=6, mid=5

2	5	8	12	16	23	38	56	72	91
					FOUND!				

$23 = 23$, found at index 5

Binary Search — Iterative Implementation

```c
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;    // Avoid overflow

        printf("low=%d, high=%d, mid=%d, arr[mid]=%d\n",
               low, high, mid, arr[mid]);

        if (arr[mid] == key)
            return mid;                 // Found!
        else if (arr[mid] < key)
            low = mid + 1;              // Search right half
        else
            high = mid - 1;             // Search left half
    }
    return -1;                          // Not found
}
```

Important: Overflow Prevention

Use `mid = low + (high - low) / 2` instead of `mid = (low + high) / 2`

Binary Search — Recursive Implementation

```
int binarySearchRec(int arr[], int low, int high, int key) {
    if (low > high)
        return -1;                    // Base case: not found

    int mid = low + (high - low) / 2;

    if (arr[mid] == key)
        return mid;                   // Base case: found!
    else if (arr[mid] < key)
        return binarySearchRec(arr, mid + 1, high, key);   // Right
    else
        return binarySearchRec(arr, low, mid - 1, key);    // Left
}

// Usage:
int result = binarySearchRec(arr, 0, n - 1, key);
```

Iterative

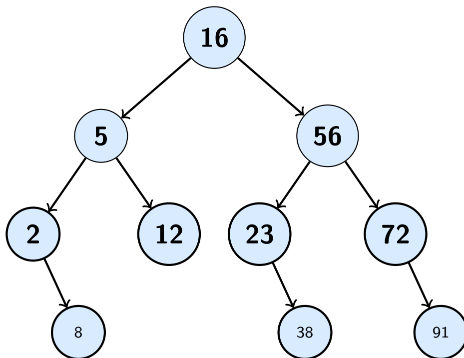
- Space: $O(1)$
- Slightly faster

Recursive

- Space: $O(\log n)$ stack
- More elegant

Binary Search — Decision Tree

Array: [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]



- Each node = one comparison
- **Height** of tree = maximum comparisons = $\lceil \log_2(n+1) \rceil$
- For $n = 10$: max $\lceil \log_2(11) \rceil = 4$ comparisons

Binary Search — Complexity Analysis

Time Complexity

Case	Time
Best Case	$O(1)$
Average Case	$O(\log n)$
Worst Case	$O(\log n)$

Why $O(\log n)$?
Each step eliminates **half** the remaining elements:

$$n \to \frac{n}{2} \to \frac{n}{4} \to \cdots \to 1$$

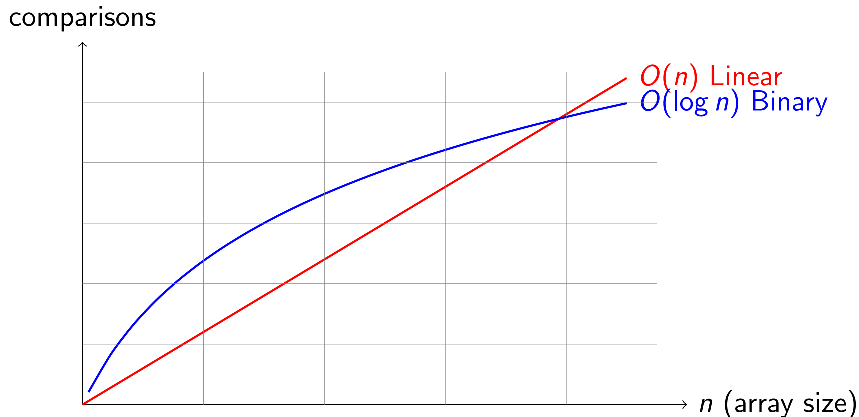
Steps: $\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

Power of $O(\log n)$

Array Size	Max Comparisons
10	4
100	7
1,000	10
1,000,000	20
1,000,000,000	30

Search 1 billion elements with only 30 comparisons!

Linear vs Binary Search — Comparison



Property	Linear Search	Binary Search
Requires sorted data?	No	**Yes**
Time complexity	$O(n)$	$O(\log n)$

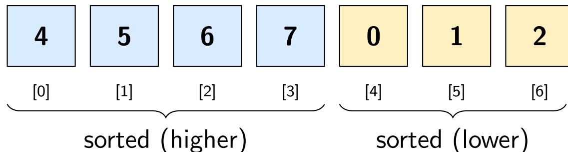
Rotated Sorted Array

What is a Rotated Sorted Array?

A sorted array that has been rotated at some unknown pivot point.

Example: Original sorted: [0, 1, 2, 4, 5, 6, 7]

Rotated at index 3:



Key Insight: At least one half is **always sorted**. We can use modified binary search to achieve $O(\log n)$.

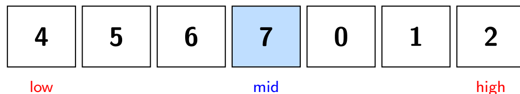
Search in Rotated Array — Implementation

```
int searchRotated(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key)
            return mid;

        // Left half is sorted
        if (arr[low] <= arr[mid]) {
            if (key >= arr[low] && key < arr[mid])
                high = mid - 1;      // Key is in left half
            else
                low = mid + 1;       // Key is in right half
        }
        // Right half is sorted
        else {
            if (key > arr[mid] && key <= arr[high])
                low = mid + 1;       // Key is in right half
            else
                high = mid - 1;      // Key is in left half
        }
```

Rotated Search — Trace Example

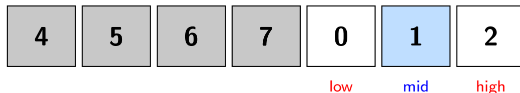
Search for 0 in [4, 5, 6, 7, 0, 1, 2]

Step 1: low=0, high=6, mid=3



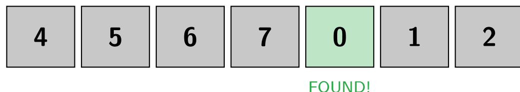
Left sorted [4..7], 0 not in [4,7] → go right

Step 2: low=4, high=6, mid=5



Right sorted [1..2], 0 not in [1,2] → go left

Step 3: low=4, high=4, mid=4



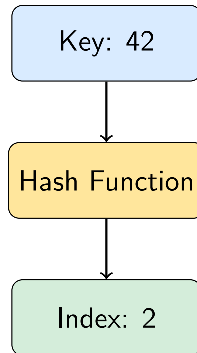
arr[4] = 0, found at index 4

The Need for Hashing

Problem: Can we do better than $O(\log n)$?

Method	**Search Time**
Linear Search	$O(n)$
Binary Search	$O(\log n)$
Hash Table	$O(1)$ **average**

Idea: Use a **hash function** to compute the *exact position* where an element should be stored.



Hash Function

Definition

A **hash function** maps a key to an index in a fixed-size array (hash table).

$$h(key) = key \mod \text{TABLE_SIZE}$$

Example: TABLE_SIZE $= 7$

Key	$h(key) = key \mod 7$	**Index**
15	$15 \mod 7 = 1$	1
22	$22 \mod 7 = 1$	1 (collision!)
35	$35 \mod 7 = 0$	0
42	$42 \mod 7 = 0$	0 (collision!)
10	$10 \mod 7 = 3$	3

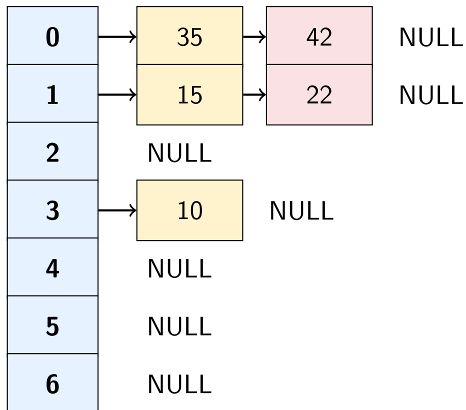
Collision

Collision Resolution: Chaining

Idea

Each slot in the hash table stores a **linked list** of all elements that hash to that index.

Insert: 35, 15, 22, 10, 42 with TABLE_SIZE $= 7$



Hash Table with Chaining — C Implementation

```c
#define TABLE_SIZE 7

struct HashNode {
    int key;
    struct HashNode *next;
};

struct HashNode *hashTable[TABLE_SIZE];

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

void hashInsert(int key) {
    int index = hashFunction(key);
    struct HashNode *newNode = malloc(sizeof(struct HashNode));
    newNode->key = key;
    newNode->next = hashTable[index];   // Insert at head
    hashTable[index] = newNode;
    printf("Inserted %d at index %d\n", key, index);
}
```

Hash Table — Search and Delete

```
int hashSearch(int key) {
    int index = hashFunction(key);
    struct HashNode *current = hashTable[index];
    while (current != NULL) {
        if (current->key == key)
            return 1;           // Found
        current = current->next;
    }
    return 0;                   // Not found
}

void hashDelete(int key) {
    int index = hashFunction(key);
    struct HashNode *current = hashTable[index];
    struct HashNode *prev = NULL;
    while (current != NULL) {
        if (current->key == key) {
            if (prev == NULL)
                hashTable[index] = current->next;
            else
                prev->next = current->next;
            free(current);
            printf("Deleted %d from index %d\n", key, index);
```

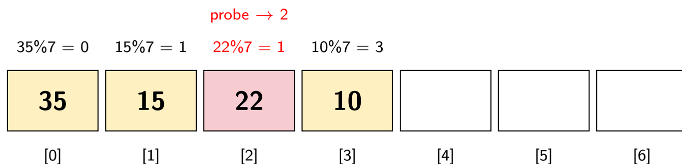
Collision Resolution: Linear Probing

Idea (Open Addressing)

If a slot is occupied, check the **next slot**, then the next, and so on (wrapping around).

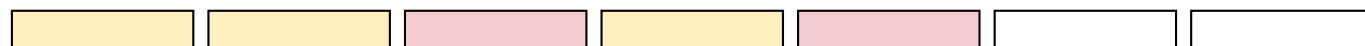
$$h(key, i) = (h(key) + i) \mod \text{TABLE_SIZE}, \quad i = 0, 1, 2, \ldots$$

Insert: 35, 15, 22, 10, 42 with TABLE_SIZE = 7



Insert 42:

42%7 = 0 → occupied → try 1 → occupied → try 2 → occupied → try 3 → occupied → **index 4**



Linear Probing — C Implementation

```c
#define TABLE_SIZE 7
#define EMPTY -1
#define DELETED -2

int hashTable[TABLE_SIZE];

void initTable() {
    for (int i = 0; i < TABLE_SIZE; i++)
        hashTable[i] = EMPTY;
}

void insertLP(int key) {
    int index = key % TABLE_SIZE;
    int i = 0;
    while (i < TABLE_SIZE) {
        int probe = (index + i) % TABLE_SIZE;
        if (hashTable[probe] == EMPTY || hashTable[probe] == DELETED) {
            hashTable[probe] = key;
            printf("Inserted %d at index %d\n", key, probe);
            return;
        }
        i++;
    }
```

Hash Table — Complexity Analysis

Chaining

Operation	Average
Insert	$O(1)$
Search	$O(1+\alpha)$
Delete	$O(1+\alpha)$

$\alpha = \frac{n}{\text{TABLE_SIZE}}$ (load factor)

Linear Probing

Operation	Average
Insert	$O\left(\frac{1}{1-\alpha}\right)$
Search	$O\left(\frac{1}{1-\alpha}\right)$
Delete	$O\left(\frac{1}{1-\alpha}\right)$

Degrades as table fills up

Load Factor α

- Keep $\alpha < 0.75$ for good performance
- If α too high $\rightarrow$ resize the table (rehashing)
- **Worst case** (all keys collide): $O(n)$ for both methods

Chaining vs Linear Probing

Property	Chaining	Linear Probing
Data structure	Linked lists	Array
Memory usage	Extra pointers	No extra memory
Clustering	No	Yes (primary clustering)
Load factor > 1	Possible	Not possible
Cache performance	Poor	Good
Deletion	Simple	Needs DELETED marker
Implementation	More complex	Simpler

Use Chaining When

- Unknown number of keys
- Frequent deletions
- Can't predict load factor

Use Linear Probing When

- Known maximum size
- Few deletions
- Cache performance matters

Properties of a Good Hash Function

1. **Deterministic:** Same key always produces same index
2. **Uniform Distribution:** Keys spread evenly across table
3. **Fast to Compute:** $O(1)$ computation
4. **Minimizes Collisions:** Different keys rarely map to same index

Common Hash Functions

Division Method: $h(k) = k \mod m$

Choose $m =$ prime number not close to power of 2

Multiplication Method: $h(k) = \lfloor m \cdot (k \cdot A \mod 1) \rfloor$

Where $0 < A < 1$ (Knuth suggests $A \approx 0.6180$)

For Strings:

$$h(s) = \left(\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i} \right) \mod m$$

Hashing Strings — C Implementation

```c
unsigned int hashString(char *str) {
    unsigned int hash = 0;
    while (*str) {
        hash = hash * 31 + *str;
        str++;
    }
    return hash % TABLE_SIZE;
}

// Example usage:
// hashString("hello")  ->  some index
// hashString("world")  ->  some index
// hashString("olleh")  ->  different index (order matters)
```

Why 31?

- 31 is prime $\rightarrow$ better distribution
- $31 = 2^5 - 1 \rightarrow$ compiler can optimize to shift and subtract
- Used by Java's `String.hashCode()`

Real-World Applications

Linear Search:

- Small datasets
- Unsorted or linked data
- Finding all occurrences

Binary Search:

- Dictionary lookup
- Database indexing (B-trees)
- Finding boundaries (lower/upper bound)
- Git bisect (bug finding)

Hash Tables:

- Symbol tables (compilers)
- Database indexing
- Caching (web browsers)
- Spell checkers
- Password storage
- Python dictionaries
- Counting frequencies

Summary — Searching Algorithms Comparison

Algorithm	Best	Average	Worst	Requirement
Linear Search	$O(1)$	$O(n)$	$O(n)$	None
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	Sorted array
Hash Table (chain)	$O(1)$	$O(1)$	$O(n)$	Hash function
Hash Table (probe)	$O(1)$	$O(1)$	$O(n)$	Hash function

Key Takeaways

- **Linear search** is simple but slow — use for small or unsorted data
- **Binary search** is efficient but requires sorted data — $O(\log n)$
- **Hash tables** offer $O(1)$ average but $O(n)$ worst case
- Choose based on: data size, sorted/unsorted, frequency of searches, memory constraints

Practice Problems

1. **[Easy]** Implement linear search that returns *all* indices where key appears
2. **[Easy]** Modify binary search to find the *first* occurrence of a duplicate
3. **[Medium]** Search in a rotated sorted array (LeetCode #33)
4. **[Medium]** Implement a hash table that resizes when $\alpha > 0.75$
5. **[Medium]** Two Sum problem using a hash table (LeetCode #1)
6. **[Hard]** Find median of two sorted arrays using binary search (LeetCode #4)

Questions?