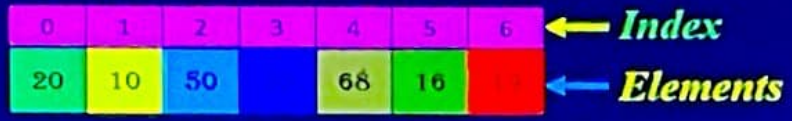
Searching

Searching: Searching is the process of finding the location or presence of a specific element (called key) within a collection of data.

Example: DATA[7]



Why Do We Need Searching?

Purpose: To determine whether an element exists and where it is located.

Uses:

- Database queries
- File systems
- Data analytics
- Online searching (e.g., product or name lookups)
- Compiler symbol tables and hash maps

Searching

Types of Searching Algorithm					
Type	**Description**	**Example**	**Best Case**	**Average Case**	**Worst Case**
Linear Search	Sequentially compares each element with the target.	Unsorted lists	O(1)	O(log n)	O(n)
Binary Search	Divides search interval in half each step (sorted list only).	Sorted arrays	O(1)	O(log n)	O(log n)
Interpolation Search	Improves Binary Search using estimated position.	Uniformly distributed data	O(1)	O(log log n)	O(n)
Exponential Search	Finds range using powers of two before Binary Search.	Infinite or unbounded lists	--	--	--
Hash Search	Uses key-to-address mapping for constant-time lookup.	Hash tables	O(1)	O(1)	O(n)

Searching

Linear Search: Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

Syntax:

```
LinearSearch(arr, n, key)
for i ← 0 to n-1
    if arr[i] == key
        return i   // Element found
return -1          // Element not found
```

Steps

1. Start
2. Input array and key
3. Compare each element
4. If found → Print position
5. Else → Continue
6. Stop

Example: Search for an element 68

$E \neq 68$

$E \neq 68$

$E \neq 68$

$E \neq 68$

$E = 68$

Results : Elements found at index 4

Time Complexity: O(5)

Searching

Binary Search is a searching algorithm that efficiently finds the position of an element in a sorted array by repeatedly dividing the search interval in half and checking the middle element in each step.

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

Binary Search Algorithm implemented in two ways 1. **Iterative Method** and 2. **Recursive Method**

- The recursive method follows the **divide and conquer** approach.

Steps

1. Divide the search space into two halves by finding the middle element.
2. If the middle element matches the key, stop the search.
3. If the key is smaller, continue searching in the left half; if larger, search in the right half.
4. Repeat until the key is found or the search space is exhausted.

Sorting

Sorting: *Sorting is the process of arranging data in a particular order (ascending or descending).*

Purpose:

- *Simplifies searching and retrieval operations*
- *Helps in data analysis and presentation*
- *Data organization and visualization*

Example:

Real-world applications: Schools, E-commerce, Libraries, Databases, Finance

- *Organize student names alphabetically for easy attendance tracking.*
- *Arrange products by price, popularity, or rating to help customers find what they need quickly.*
- *Books are sorted by title or author name to make it easy to locate a specific book.*
- *Sorting data in databases makes searching for information faster and more efficient.*

Sorting

Types of Sorting

1. **Internal Sorting:** All data fits into main memory. Example: Quick Sort, Merge Sort.
2. **External Sorting:** Data is too large for memory; uses disk storage. Example: External Merge Sort.

Classification of Sorting Algorithms

1. **By Comparison:**
   - **Comparison-based Sorting:** These algorithms sort data by comparing elements.
     Examples: *Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort.*
   - **Non-comparison-based Sorting:** These algorithms sort data without comparing elements directly.
     Examples: *Counting Sort, Radix Sort, and Bucket Sort.*
2. **By Stability:**
   - **Stable Sorting Algorithms:** Stable sort algorithms maintain the relative order of equal elements. Examples: Bubble Sort, Merge Sort, and Insertion Sort.
   - **Unstable Sorting Algorithms:** Unstable sort algorithms do not guarantee the relative order of equal elements. Examples: Quick Sort, Heap Sort, and Selection Sort.

Sorting

Characteristics of Sorting Algorithms

1. **Stability**
   - **Stable:** Maintains the relative order of equal elements (e.g., Bubble Sort and Insertion Sort)
   - **Unstable:** May change the relative order of equal elements (e.g., Quick Sort and Heap Sort)
2. **Recursive vs. Iterative**
   - **Recursive:** Uses recursive calls (e.g., Merge Sort, Quick Sort).
   - **Iterative:** Uses loops (e.g., Bubble Sort, Selection Sort).
3. **Adaptive vs. Non-Adaptive**
   - **Adaptive:** Performs better on partially sorted data (e.g., Insertion Sort).
   - **Non-Adaptive:** Does not take advantage of pre-existing order (e.g., Selection Sort).
4. **Internal vs. External**
   - **Internal:** All data fits into memory (e.g., most common sorting algorithms).
   - **External:** Used for large datasets that don't fit into memory (e.g., External Merge Sort).

Sorting

Types of Sorting Algorithm

Sorting Algorithm	Time Complexity (B)	Time Complexity(W)	Avg. Time Complexity	Space Complexity	Best When / Use Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	When the list is *almost sorted* and simplicity is needed
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	When *memory writes* are costly (fewer swaps)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	For *small* or *nearly sorted* datasets
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	When *stability* is needed and data is large
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	O(log n)	When *average-case speed* is preferred and recursion depth is manageable
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(max)	When data are *integers in a limited range*
Radix Sort	O(n + k)	O(n + k)	O(n + k)	O(max)	When sorting *numbers* or *strings* of fixed length
Bucket Sort	O(n + k)	$O(n^2)$	O(n)	O(n + k)	When data are *uniformly distributed* across a range
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	When *memory usage must be minimal*
Shell Sort	O(n log n)	$O(n^2)$	O(n log n)	O(1)	For *medium-sized arrays* where partial ordering helps

Sorting

Bubble Sort is a simple sorting algorithm that repeatedly traverses the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is completely sorted.

Why It's Called Bubble Sort?

- Smaller elements gradually "bubble" to the top (beginning) of the list.
- Larger elements sink to the bottom with each pass.

Steps

1. Compare the first two adjacent elements.
2. Swap them if they are in the wrong order.
3. Move to the next pair and repeat.
4. Continue this process until the end of the list.
5. Repeat for all elements until no swaps are needed.

Syntax

```
for(i = 0; i < n - 1; i++) {
    for(j = 0; j < n - i - 1; j++) {
        if(arr[j] > arr[j + 1]) {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
```

Advantages:

- Easy to understand and implement.
- Works well for small datasets.

Sorting

Bubble Sort Example:

Unsorted Array

STEP 1:

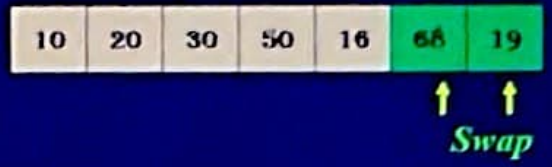
STEP 2:

STEP 3:

STEP 4:

STEP 5:

STEP 6:



After Pass 1: 10 20 30 50 16 19 68 ← *Largest element 68 is now at the end*

After Pass 2: 10 20 30 16 19 50 68 ← *Second Largest element 50 is now at the second end*

After Pass 3: 10 20 16 19 30 50 68 ← *Third Largest element 30 is now at the third end*

After Pass 4: 10 16 19 20 30 50 68 ← *Fourth Largest element 20 is now at the fourth end*

After Pass 5: 10 16 19 20 30 50 68 ← *No swaps in Pass 5 → array is sorted; algorithm terminates.*

Passes performed: 5 (stopped early because a pass had no swaps), Total swaps: 10

Sorting

Selection Sort is a simple comparison-based algorithm that divides the array into a **sorted** and an **unsorted** part, repeatedly selecting the smallest element from the unsorted section and placing it at the beginning to grow the sorted portion.

Steps

1. Start from the first element (assume it as the smallest).
2. Compare this element with the rest of the array.
3. If any element is smaller, update the smallest index.
4. After scanning, swap the smallest element with the first element.
5. Move the boundary of the sorted and unsorted parts one step forward.
6. Repeat until the entire array is sorted..

Syntax

```
for (i = 0; i < n - 1; i++) {
   min = i;
   for (j = i + 1; j < n; j++) {
      if (arr[j] < arr[min])
         min = j;
   }
   // Swap smallest with first element
   temp = arr[min];
   arr[min] = arr[i];
   arr[i] = temp;
}
```

Advantages:

- Easy to understand and implement.
- Performs well on small datasets.
- Requires minimal swaps (n-1 max).

Disadvantages:

- Inefficient for large datasets.
- Time complexity is always $O(n^2)$.
- Not adaptive (doesn't stop early if already sorted).

Sorting

Why Selection Sort When Bubble Sort Exists ?

1. **Fewer Swaps:**
   - Selection Sort performs only one swap per pass, totaling (n – 1) swaps.
   - Bubble Sort may perform many swaps per pass, increasing data movement.
   - Better choice when swapping elements is expensive (e.g., large records or files).
2. **Predictable Performance:**
   - Selection Sort always performs the same number of comparisons, even if the array is already sorted.
   - Bubble Sort performs fewer operations on nearly sorted data because it can stop early when no swaps are needed.
   - Selection Sort = consistent; Bubble Sort = adaptive.
3. **Simplicity of Logic:**
   - Selection Sort's logic of finding the smallest (or largest) element and placing it in order is easier to understand.

Important: *Selection Sort is preferred when the cost of swapping is high, while Bubble Sort is preferred when early termination (adaptive sorting) can save time on nearly sorted data.*

C2 4B XW

Sorting

Selection Sort Example:

20	10	50	30	68	16	19

Pass 1:

STEP 1:

20	10	50	30	68	16	19

Min = 20

STEP 2:

20	10	50	30	68	16	19

Min = 10, Swap(20,10)

STEP 3:

10	20	50	30	68	16	19

Min = 10

STEP 4:

10	20	50	30	68	16	19

Min = 10

STEP 5:

10	20	50	30	68	16	19

Min = 10

STEP 6:

10	20	50	30	68	16	19

Min = 10

STEP 7:

10	20	50	30	68	16	19

Min = 10

After Pass 1:	10	20	50	30	68	16	19	← *Smallest element 10 is now at the first position*
After Pass 2:	10	16	50	30	68	20	19	← *Second smallest element 16 is now at the second position*
After Pass 3:	10	16	19	30	68	20	50	← *Third smallest element 19 is now at the third position*
After Pass 4:	10	16	19	20	68	30	50	← *Fourth smallest element 20 is now at the fourth position*
After Pass 5:	10	16	19	20	30	68	50	← *Fifth smallest element 30 is now at the fifth position.*
After Pass 6:	10	16	19	20	30	50	68	← *Sixth smallest element 50 is now at the sixth position.*

Sorting

Insertion Sort is a simple algorithm that builds a sorted array one element at a time by repeatedly inserting each item from the unsorted part into its correct position in the sorted part.

Steps

1. Start from the second element (first element is assumed sorted).
2. Compare the current element with the elements before it.
3. Shift all larger elements one position to the right.
4. Insert the current element into its correct position.
5. Repeat for all remaining elements.

Syntax

```
for (i = 1; i < n; i++) {
   key = arr[i];
   j = i - 1;
   // Move elements greater than key one position ahead
   while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j = j - 1;
   }
   arr[j + 1] = key;
}
```

Advantages:

- Simple and easy to implement.
- Efficient for small or nearly sorted datasets.
- In-place sorting (no extra memory needed).

Disadvantages:

- Inefficient for large datasets.
- $O(n^2)$ time complexity in worst case.

Insertion Sort is efficient for small or nearly sorted datasets because it reduces unnecessary comparisons and shifts, adapting dynamically to the data's existing order — unlike Bubble or Selection Sort, which always perform a fixed number of passes.

Sorting

Insertion Sort Example: 20 | 10 | 50 | 30 | 68 | 16 | 19 — First element is assumed sorted i.e., 20

	Key Element	Comparison / Shifting	Result After Pass
Pass 1:	20 | 10 … (Key = 10)	20 > 10 → shift 20	10 | 20 …
Pass 2:	10 | 20 | 50 … (Key = 50)	50 > 20 → No shift	10 | 20 | 50 …
Pass 3:	10 | 20 | 50 | 30 … (Key = 30)	50 > 30 → shift 50	10 | 20 | 30 | 50 …
Pass 4:	10 | 20 | 30 | 50 | 68 … (Key = 68)	68 > 50 → no shift	10 | 20 | 30 | 50 | 68 …
Pass 5:	10 | 20 | 30 | 50 | 68 | 16 … (Key = 16)	68, 50, 30, 20 > 16 → shift all	10 | 16 | 20 | 30 | 50 | 68 …
Pass 6:	10 | 20 | 30 | 50 | 68 | 16 | 19 (Key = 19)	68, 50, 30, 20 > 19 → shift all	10 | 16 | 19 | 20 | 30 | 50 | 68

Passes performed: 6 (Each pass inserts one element into its correct position in the sorted part)

Total shifts/swaps: Around 10 (depends on how many elements need to be moved during each insertion)

Sorting

Quick Sort Quick Sort is one of the most efficient and widely used sorting algorithms that follows the *divide and conquer approach*. It works by selecting a pivot element, partitioning the array into two subarrays based on whether elements are less than or greater than the pivot, and then recursively sorting the subarrays until the entire array is sorted.

Steps

1. Choose a pivot element from the array/list.
2. Partition the array into two subarrays:
   - Left subarray ⟶ elements less than pivot
   - Right subarray ⟶ elements greater than pivot
3. Recursively apply the same process to both subarrays.
4. Combine the subarrays ⟶ the array becomes sorted.

Syntax

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);  // Left subarray
        quickSort(arr, pivotIndex + 1, high); // Right subarray
    }
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

Advantages:

- Very efficient for large datasets.
- In-place sorting (no extra memory required).
- Better average performance than Merge Sort.

Disadvantages:

- Worst case $O(n^2)$ occurs if pivot is poorly chosen (e.g., smallest or largest element every time).
- Not stable (order of equal elements not guaranteed).

Sorting

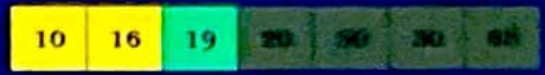
Quick Sort Example:

Step 1: Choose Pivot = 19

Array after Partition →

Step 2: Sort Left Subarray [10, 16]

Already Sorted →



Step 3: Sort Right Subarray [20, 50, 30, 68]

Choose Pivot = 30

Not Sorted →

Left and Right Already Sorted

Step 4: Combine the subarrays(Final Sorted Array) →

Sorting

Q49. Write a program to sort an array of integers using Bubble Sort.

Q50. Write a program to sort elements in ascending and descending order using Bubble Sort.

Q51. Write a program to count the number of swaps and comparisons made during Bubble Sort.

Q52. Write a program to implement Selection Sort on an array of numbers.

Q53. Write a program to implement Insertion Sort.

Q54. Write a program to implement Quick Sort.