Data Structure

Syllabus

1	Introduction to Data Structure: Fundamentals of Data Structure, Operations of Data Structure: Traversing, Inserting and Deleting, Arrays as Data Structure, Searching, Sorting Bubble, Insertion, Selection).
2	Stacks: Introduction and Definition, Representation, Operations on Stacks, Applications of Stacks, Representation of Arithmetic Expressions: Infix, Postfix, Prefix.
3	Queues: Introduction and Definition, Representation, Operation on Queues, Types of Queues, Dequeue, Circular Queue, Priority Queue, Applications of Queue.
4	Linked List: Definition of Linked List, Dynamic Memory Management, Representation of Linked List, Operations on Linked List, Inserting, Removing, Searching, Sorting, Merging Nodes , Double Linked List
5	Trees : Definition of Tree, Binary Tree and their types, Representation of Binary Tree, Operations on Binary Tree, Binary Search Tree (BST), Traversal of Binary Tree, Preorder Traversal, In-order Traversal, Post-order Traversal, Introduction of Threaded Binary Tree, AVL Tree and B-Tree.
6	Graphs: Definition of Graph, Basic Concepts of Graph, Representation of Graph, Adjacency Matrix, Adjacency List, Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS)

Reference Books

- Data Structures Using C 2E: Reema Thareja, Oxford Higher Education
- Data Structures Using C and C++ : Yedidyah Langsam, Moshe Augenstein, Aaron Tenenbaum, PHI,2nd Ed.
- Magnifying Data Structures : Arpita Gopal, PHI Learning Pvt. Ltd
- The Ultimate C with Data Structures: R. Nageswara Rao, Dreamtech publications
- Data Structure through C Y.P. Kanetkar, BPB,2nd Ed.
- Data Structures with C: Schaums Outlines Hubbard John
- Data Structures Using C and C++ (Tenenbaum) Tenenbaum, Pearson Pub.
- Fundamental of Data Structure in C Horowitz Sahani, Galgotia pub.

- **Introduction to Data Structure**
- **Fundamentals of Data Structure**
- **Operations of Data Structure: Traversing, Inserting and Deleting Arrays as Data Structure**
- **Searching**
- **Sorting Bubble, Insertion, Selection**

Understanding Data Structures

- Imagine you are the librarian of a large public library with over 10,000 books.
- Your task is to organize these books efficiently.
- How would you organize this library so that you can quickly find, add, remove, and categorize books?

Understanding Data Structures

Form small groups for this activity. Each group will be assigned a different library task,

- **Group 1:** Managing a book return counter .
- **Group 2:** Organizing a checkout line.
- **Group 3:** Creating a digital catalog .
- **Group 4:** Recommending similar books based on connections.

Each group has to discuss and present how their assigned structure helps in managing the library efficiently.

Exploring different data structures

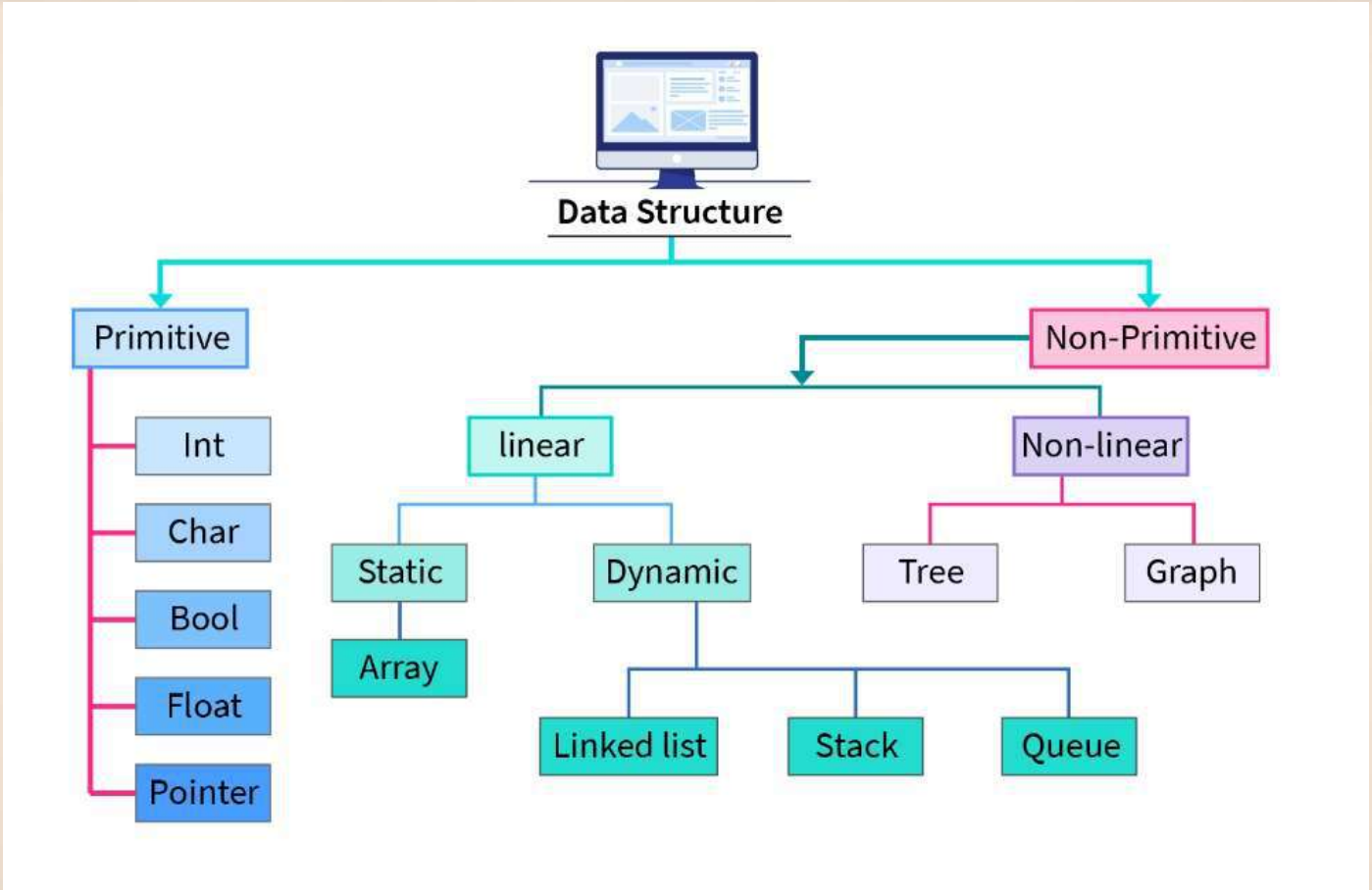
- Shelves arranged in fixed rows, each with a specific capacity (fixed size) -> **Array Analogy**
- A trail of books, where each book points to the next one, allowing flexible addition but slower searching -> **Linked List Analogy**
- A pile of returned books, where the last book returned is the first to be processed -> **Stack Analogy**
- A line of people waiting to check out books – first person in is the first person served -> **Queue Analogy**
- A hierarchical organization, like categorizing books by genre, then by author, then by title-> **Tree Analogy.**
- A recommendation network, where books are connected by similar themes or shared readers-> **Graph Analogy**

Why use Data Structures?

- **Efficient Data Management**: Organizes data for quick access and modification.
- **Optimized Performance**: Reduces time complexity for operations like searching, sorting, and traversal.
- **Memory Efficiency**: Minimizes the memory footprint by storing only what is necessary.
- **Data Integrity**: Maintains relationships between data elements effectively.
- **Scalability**: Handles large volumes of data without significant performance loss.

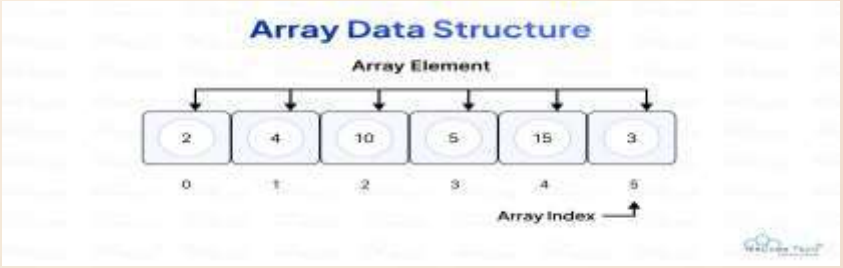
Basic Terminology: Elementary Data Organization:

- Data: Data are simply values or sets of values.
- Information is organized or classified data, which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based.
- Data items: Data items refers to a single unit of values.
- Data items that are divided into sub-items are called Group items. Ex: An Employee Name may be divided into three subitems- first name, middle name, and last name.
- Data items that are not able to divide into sub-items are called Elementary items. Ex: SSN
- Entity: An entity is something that has certain attributes or properties which may be
- assigned values. The values may be either numeric or non-numeric.
- Ex: Attributes- Names, Age, Sex, SSN
- Values- Rohland Gail, 34, F, 134-34-5533


Data Structure
Primitive
Non-Primitive
Int
Char
Bool
Float
Pointer
linear
Non-linear
Static
Dynamic
Tree
Graph
Array
Linked list
Stack
Queue

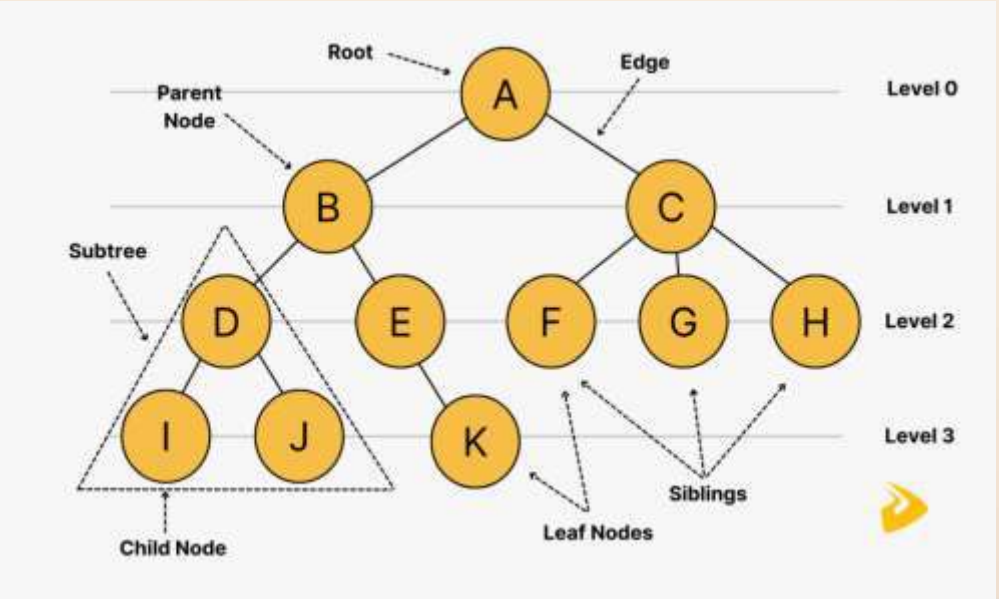
Arrays:

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number n of similar data referenced respectively by a set of n consecutive numbers, usually 1, 2, 3 . . . . . n. if A is chosen the name for the array, then the elements of A are denoted by subscript notation a1, a2, a3..... an
by the bracket notation A [1], A [2], A [3] . . . . . A [n]



Trees:

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or a tree



1. Stack: A stack, also called a fast-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the top.

This structure is similar in its operation to a stack of dishes on a spring system as shown in fig. Note that new 4 dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the Stack

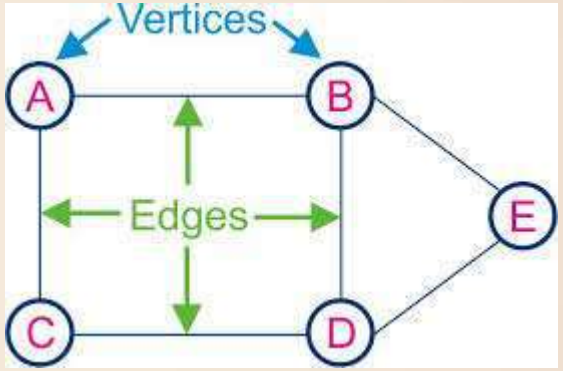
2. Queue: A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "from'' of the list, and insertions can take place only at the other end of the list, the “rear” of the list.

This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. the first person in line is the first person to board the bus. Another analogy is with automobiles waiting to pass through an intersection the first car in line is the first car through.

3. Graph: Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines in Fig. The data structure which reflects this type of relationship is called a graph.

A pile of plates

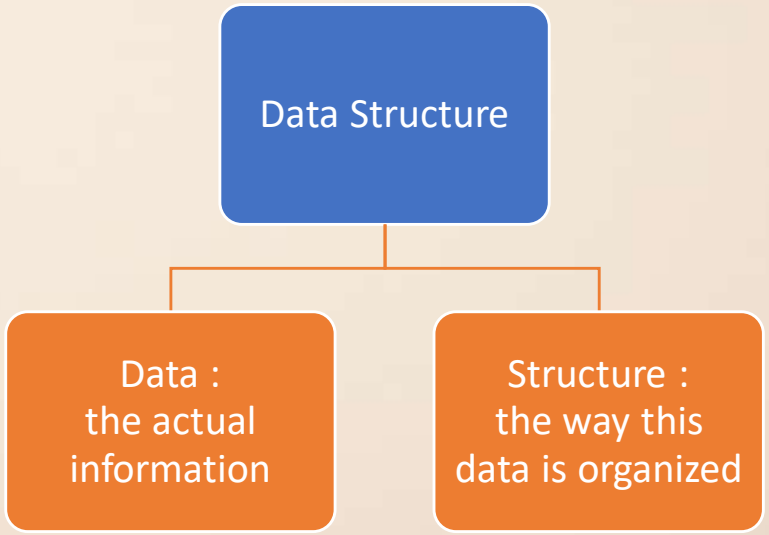
Queue in Busstand



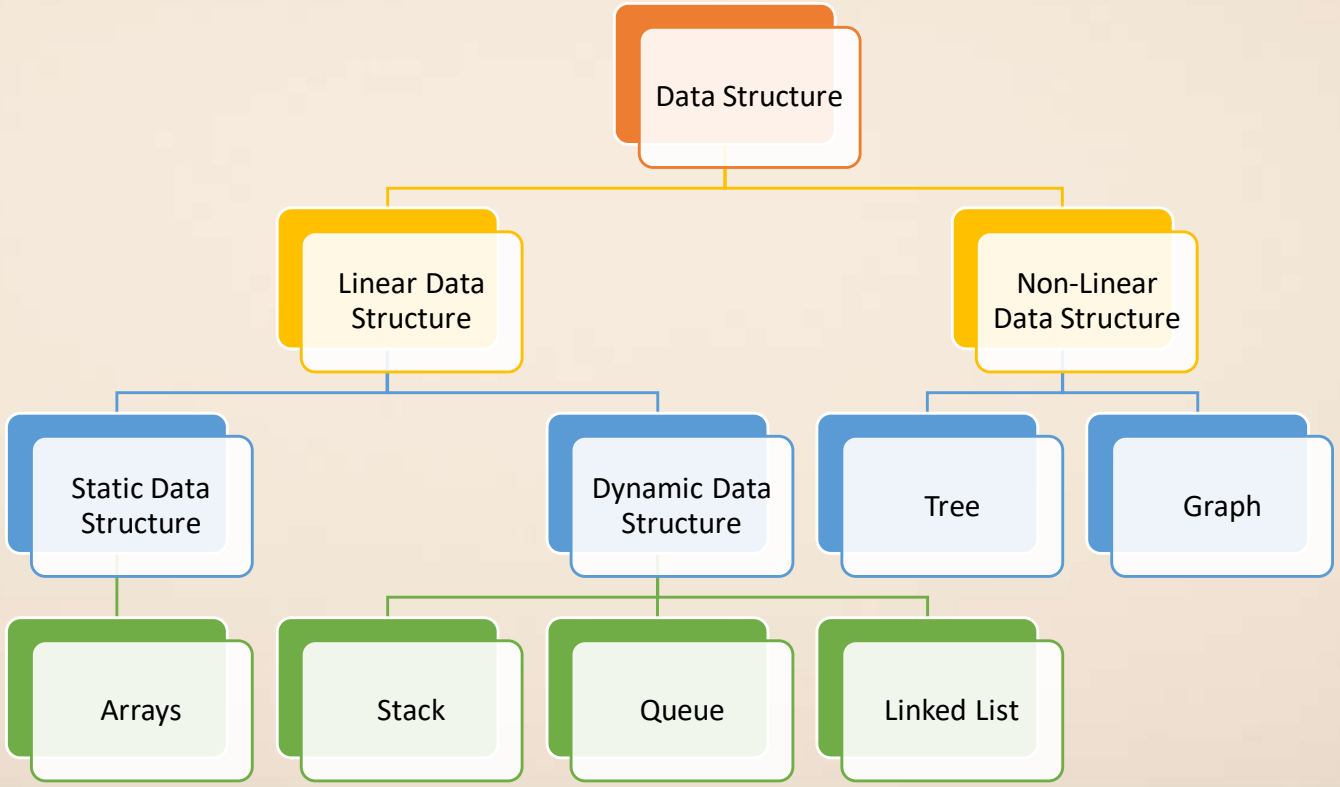
Graph

What is Data Structure?

Data Structure is essentially a combination of data (the actual information) and structure (the way this data is organized). It defines how data is stored, organized, and accessed in a computer's memory.



Types of Data Structures



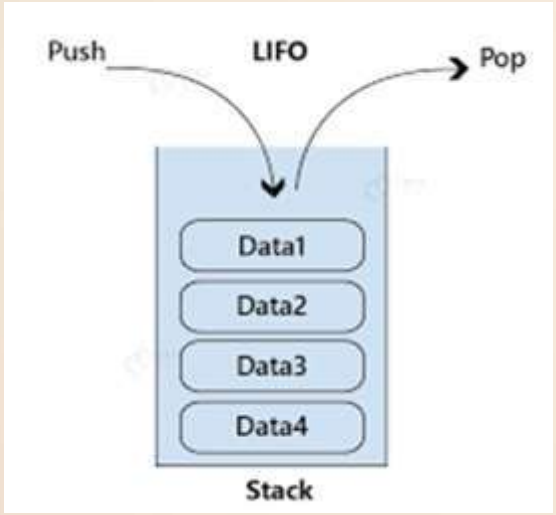
Linear Data Structures

- Elements are arranged in a sequential order.
- Memory is typically allocated in a contiguous manner (except linked lists).
- Simple to implement and understand.
- Examples: Arrays, Stacks, Queues, Linked Lists.
- Advantages: Easy traversal, predictable performance.
- Disadvantages: Fixed size (arrays), higher memory overhead (linked lists).
- Common Use Cases: Task scheduling, undo/redo features, sequential data storage.

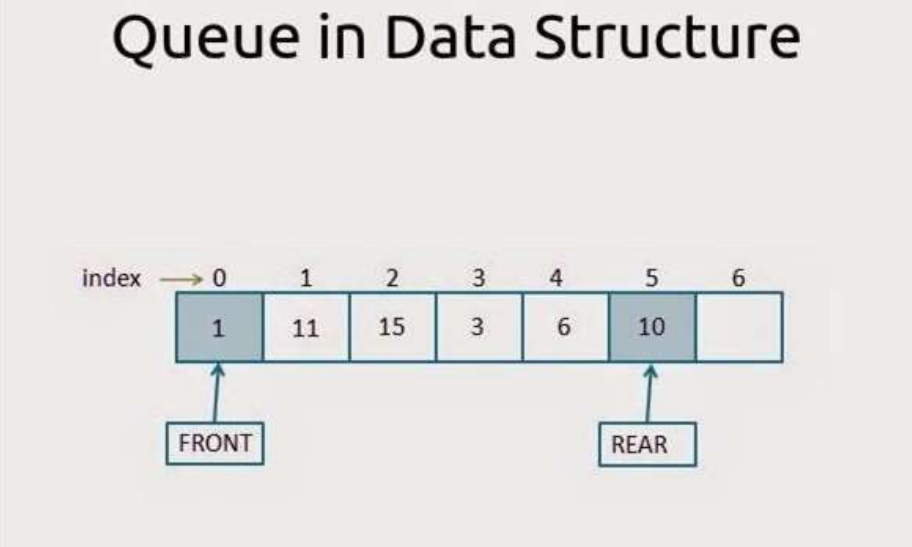
Arrays Data Structure

An Array is a collection of elements, each identified by an index or a key.

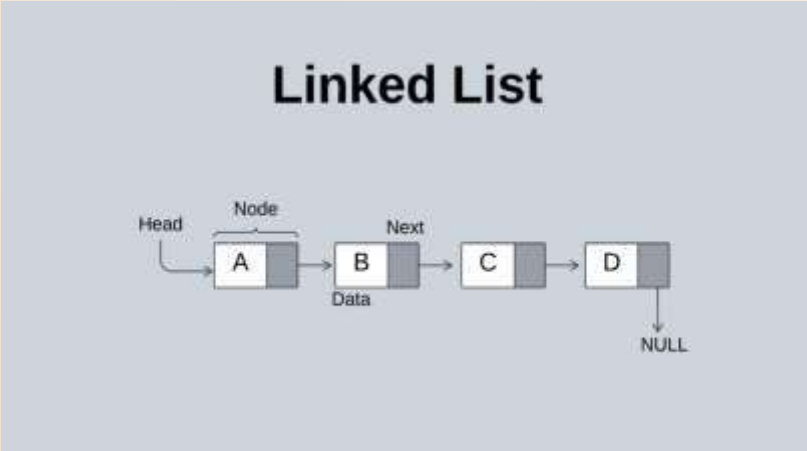
Stack Data Structure

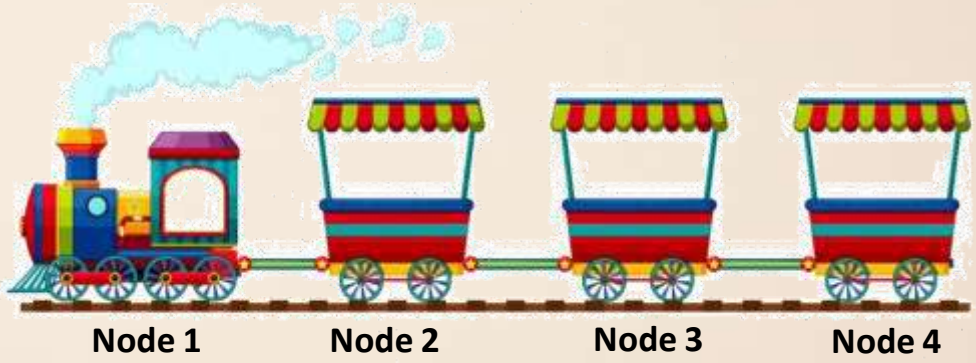


Queue Data Structure



Linked List Data Structure

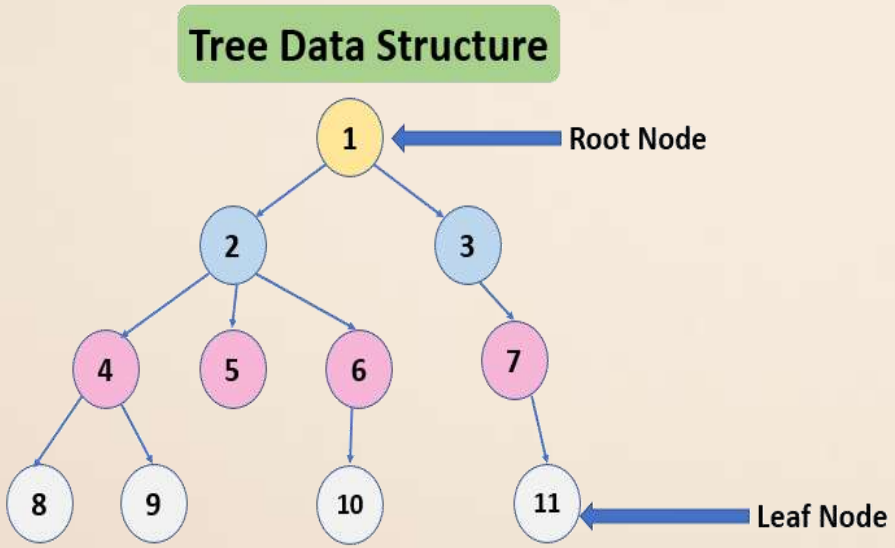


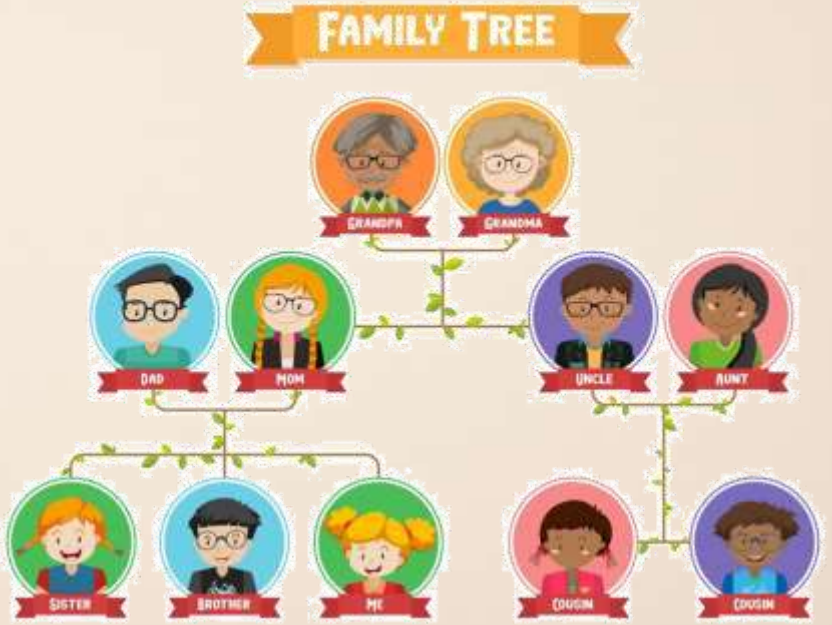


Non-Linear Data Structures

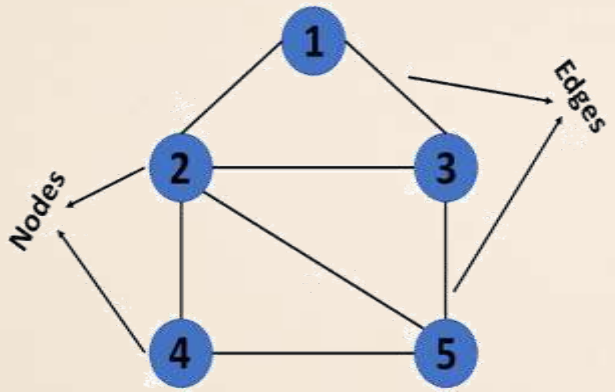
- Elements are not arranged in a sequential manner, forming a hierarchical or interconnected pattern.
- Can represent complex relationships.
- Examples: Trees, Graphs, Heaps.
- Advantages: Efficient searching, dynamic relationships.
- Disadvantages: More complex to implement and traverse.
- Common Use Cases: File systems, network routing, AI pathfinding.

Tree Data Structure





Graph Data Structure



Graph Data Structures

Social Media Graph

Array Operations

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

Traversing an array

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:      Apply Process to A[I]
Step 4:      SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

```
#include <stdio.h>
#include <conio.h>
int main()
{
int i, n, arr[20];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d",&arr[i]);
}
printf("\n The array elements are ");
for(i=0;i<n;i++)
 printf("\t %d", arr[i]);
return 0;
}
```

Inserting element in array

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Algorithm to append a new element to an existing array

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:          SET A[I + 1] = A[I]
Step 4:          SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Algorithm to insert an element in the middle of an array.

```
int main()
 {
 int i, n, num, pos, arr[10];
 clrscr();
 printf("\n Enter the number of
elements in the array : ");
 scanf("%d", &n);
 for(i=0;i<n;i++)
 {
  printf("\n arr[%d] = ", i);
  scanf("%d", &arr[i]);
 }
printf("\n Enter the number to be inserted : ");
 scanf("%d", &num);
 printf("\n Enter the position at which the
number has to be added : ");
 scanf("%d", &pos);
 for(i=n–1;i>=pos;i—)
  arr[i+1] = arr[i];
 arr[pos] = num;
 n = n+1;
 printf("\n The array after insertion of %d is : ",
num);
 for(i=0;i<n;i++)
 printf("\n arr[%d] = %d", i, arr[i]);
 getch();
 return 0;
 }
```

Delete Element in an array

```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

Algorithm to delete the last element of an array

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:          SET A[I] = A[I + 1]
Step 4:          SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Algorithm to delete an element from the middle of an array

```
int main()
{
}
Output
int i, n, pos, arr[10];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d", &arr[i]);
}
printf("\nEnter the position from which the number has to be deleted : ");
 scanf("%d", &pos);
 for(i=pos; i<n–1;i++)
 arr[i] = arr[i+1];
 n– –;
 printf("\n The array after deletion is : ");
 for(i=0;i<n;i++)
 printf("\n arr[%d] = %d", i, arr[i]);
 getch();
 return 0;
```

Linear Search

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:              IF A[I] = VAL
                             SET POS = I
                             PRINT POS
                             Go to Step 6
                     [END OF IF]
                      SET I = I + 1
              [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Linear Search Algorithm

- Linear search executes in O(n) time where n is the number of elements in the array.
- Best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.
- Worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made.
- Performance of the linear search algorithm can be improved by using a sorted array.

Binary search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Binary Search Algorithm

The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array.

The complexity of the algorithm is calculated depending on the number of comparisons that are made.

In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half.

The total number of comparisons that will be made is given as $2f(n) > n$ or $f(n) = \log_2 n$

```
#include <stdio.h>
int main() {
    int A[100], n, x, i = 1, found = 0;
    printf("Enter number of elements (n): ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for (int j = 0; j < n; j++) {
        scanf("%d", &A[j]);
    }
    printf("Enter value to search (x): ");
    scanf("%d", &x);
    while (i <= n) {
        if (A[i - 1] == x) {
            printf("Element %d found at index %d\n", x, i);
            found = 1;
            break;
        }
        i++;
    }
    if (!found) {
        printf("Element %d not found\n", x);
    }
    return 0;
}
```

Sorting

BUBBLE_SORT(A, N)

Step 1: Repeat Step2For1= to N-1

Step 2:

Repeat ForJ= toN-I

Step 3:

IF A[J]>A[J + 1]

SWAP A[J] and A[J+1]

[END OF INNER LOOP]

[END OF OUTER LOOP]

Step 4: EXIT

Complexity of Bubble Sort

In the first pass, N–1 comparisons are made to place the highest element in its correct position.

In Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$f(n) = (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2 + 1$

$f(n) = n (n – 1)/2$

$f(n) = n2/2 + O(n) = O(n2)$

Therefore, the complexity of bubble sort algorithm is O(n2).

Example

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.

(b) Compare 52 and 29. Since 52 > 29, swapping is done.

30, **29**, **52**, 87, 63, 27, 19, 54

(c) Compare 52 and 87. Since 52 < 87, no swapping is done.

(d) Compare 87 and 63. Since 87 > 63, swapping is done.

30, 29, 52, **63**, **87**, 27, 19, 54

(e) Compare 87 and 27. Since 87 > 27, swapping is done.

30, 29, 52, 63, **27**, **87**, 19, 54

(f) Compare 87 and 19. Since 87 > 19, swapping is done.

30, 29, 52, 63, 27, **19**, **87**, 54

(g) Compare 87 and 54. Since 87 > 54, swapping is done.

30, 29, 52, 63, 27, 19, **54**, **87**

Pass 2:

(a) Compare 30 and 29. Since 30 > 29, swapping is done.

29, 30, 52, 63, 27, 19, 54, 87

(b) Compare 30 and 52. Since 30 < 52, no swapping is done.

(c) Compare 52 and 63. Since 52 < 63, no swapping is done.

(d) Compare 63 and 27. Since 63 > 27, swapping is done.

29, 30, 52, 27, 63, 19, 54, 87

(e) Compare 63 and 19. Since 63 > 19, swapping is done.

29, 30, 52, 27, 19, 63, 54, 87

(f) Compare 63 and 54. Since 63 > 54, swapping is done.

29, 30, 52, 27, 19, 54, 63, 87

Pass 3:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.

(b) Compare 30 and 52. Since 30 < 52, no swapping is done.

(c) Compare 52 and 27. Since 52 > 27, swapping is done.

29, 30, 27, 52, 19, 54, 63, 87

(d) Compare 52 and 19. Since 52 > 19, swapping is done.

29, 30, 27, 19, 52, 54, 63, 87

(e) Compare 52 and 54. Since 52 < 54, no swapping is done

Pass 4:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.

(b) Compare 30 and 27. Since 30 > 27, swapping is done.

29, 27, 30, 19, 52, 54, 63, 87

(c) Compare 30 and 19. Since 30 > 19, swapping is done.

29, 27, 19, 30, 52, 54, 63, 87

(d) Compare 30 and 52. Since 30 < 52, no swapping is done.

Pass 5:

(a) Compare 29 and 27. Since 29 > 27, swapping is done. 27, 29, 19, 30, 52, 54, 63, 87

(b) Compare 29 and 19. Since 29 > 19, swapping is done. 27, 19, 29, 30, 52, 54, 63, 87

(c) Compare 29 and 30. Since 29 < 30, no swapping is done

Pass 6:

(a) Compare 27 and 19. Since 27 > 19, swapping is done.

19, 27, 29, 30, 52, 54, 63, 87

(b) Compare 27 and 29. Since 27 < 29, no swapping is done

Pass 7: (a) Compare 19 and 27. Since 19 < 27, no swapping is done

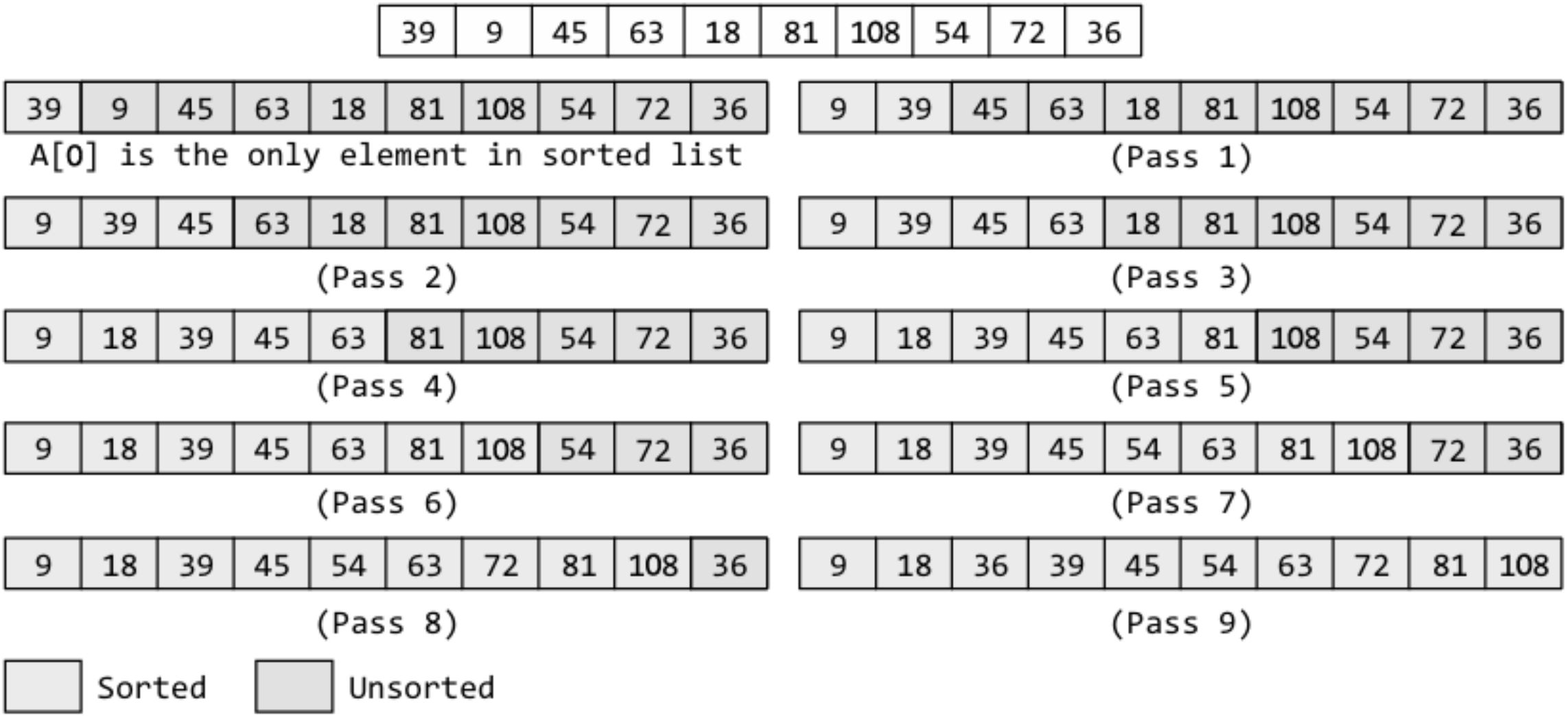
Case	Time Complexity	Description
Best Case	**O(n)**	Array is already sorted. Optimized version detects no swaps and exits early.
Average Case	**$O(n^2)$**	Elements are randomly ordered. Each pass involves multiple comparisons and swaps.
Worst Case	**$O(n^2)$**	Array is sorted in reverse. Every element must be compared and swapped in each pass.

Insertion sort

- Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time.
- Ordering a deck of cards while playing bridge.
- The main idea behind insertion sort is that it inserts each item into its proper place in the final list.
- To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.
- Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and

INSERTION-SORT (ARR, N)

```
Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:      SET TEMP = ARR[K]
Step 3:      SET J = K - 1
Step 4:      Repeat while TEMP <= ARR[J]
                  SET ARR[J + 1] = ARR[J]
                  SET J = J - 1
             [END OF INNER LOOP]
Step 5:      SET ARR[J + 1] = TEMP
         [END OF LOOP]
Step 6: EXIT
```


39 9 45 63 18 81 108 54 72 36
39 9 45 63 18 81 108 54 72 36
A[0] is the only element in sorted list
9 39 45 63 18 81 108 54 72 36
(Pass 1)
9 39 45 63 18 81 108 54 72 36
(Pass 2)
9 39 45 63 18 81 108 54 72 36
(Pass 3)
9 18 39 45 63 81 108 54 72 36
(Pass 4)
9 18 39 45 63 81 108 54 72 36
(Pass 5)
9 18 39 45 63 81 108 54 72 36
(Pass 6)
9 18 39 45 54 63 81 108 72 36
(Pass 7)
9 18 39 45 54 63 72 81 108 36
(Pass 8)
9 18 36 39 45 54 63 72 81 108
(Pass 9)
Sorted
Unsorted

Case	Time Complexity	Description
Best Case	**O(n)**	When the array is already sorted. Only one comparison per element.
Average Case	**$O(n^2)$**	Elements are randomly ordered. Each element may be compared with half of the sorted part.
Worst Case	**$O(n^2)$**	When the array is sorted in reverse. Every new element must shift all previous elements.

Selection sort

1. Start from the first element.
2. Find the **smallest element** in the unsorted portion.
3. Swap it with the first unsorted element.
4. Move the boundary of the sorted portion one step forward.
5. Repeat until the entire array is sorted

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

TimeComplexity

Case	Time Complexity	Description
Best Case	$O(n^2)$	Still compares all pairs
Average Case	$O(n^2)$	Typical performance
Worst Case	$O(n^2)$	Reverse sorted or random order

Advantages of Insertion Sort

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantiallysorted.
- It performs betterthan algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- it requires less memory space (only O(1) of additional memory space).
- it is said to be online, as it can sort a list as and when it receives new elements.

Criteria	Bubble Sort	Insertion Sort	Selection Sort
Time Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$
Best Case	O(n) (optimized)	O(n)	$O(n^2)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Space Complexity	O(1)	O(1)	O(1)
Stability	Yes	Yes	No
Adaptiveness	No	Yes	No
Number of Swaps	High	Low	Lowest
Ease of Implementation	Very Easy	Easy	Easy
Real-world Use	Rare	Insertion for small sets	Rare

Applications of Array

- Arrays are frequently used in C, as they have a number of useful applications. These applications are
- Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.
- Many databases include one-dimensional arrays whose elements are records.
- Arrays are also used to implement other data structures such as strings, stacks, queues, heaps, and hash tables. We will read about these data structures in the subsequent chapters.
- Arrays can be used for sorting elements in ascending or descending order.

```c
#include <stdio.h>
int main() {
    int arr[] = {7, 2, 9, 6, 4};
    int length = sizeof(arr) / sizeof(arr[0]); // Calculate array length
    printf("Array elements:\n");
    for (int i = 0; i < length; i++) {
        printf("Element at index %d: %d\n", i, arr[i]);
    }
    return 0;
}
```