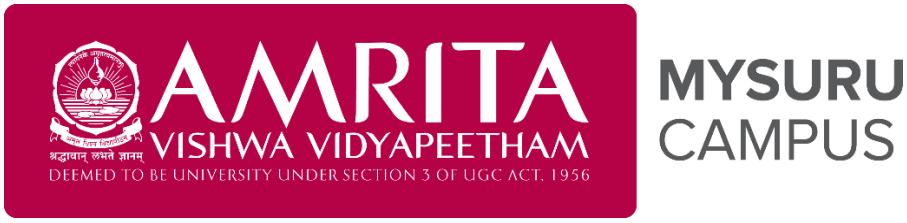

AMRITA SCHOOL OF COMPUTING, MYSURU CAMPUS

School of Computing

24CSA502: Data Structures

A Project Report on

DATA STRUCTURE INTERNAL COMPONENT

Submitted by:

Darshan Suresh
MY.AC.P2MCA25104
MCA 'D', Semester I

Submitted To:
MS. Priyanka
Master of Computer Applications (MCA)
Amrita Vishwa Vidyapeetham, Mysuru Campus
Academic Year: 2025 – 2026

INDEX

PART A — LAB PROGRAMS INDEX

1. ARRAYS

2. LINKED LISTS

3. STACK

3.9 Simulate Recursive Factorial

4. QUEUE

4.1 Queue Using Array
4.2 Circular Queue
4.3 Priority Queue (Linked List Based)
4.4 Deque (Double Ended Queue)

5. GRAPHS

5.1 Breadth First Search (BFS)
5.2 Depth First Search (DFS – Adjacency Matrix)
5.3 DFS (Adjacency List)

6. TREE

6.1 Tree Traversals (Inorder, Preorder, Postorder)
6.2 DFS on Trees
6.3 Binary Search Tree (Insertion & Traversal)
6.4 Heap Tree

PART B — PROGRAMS

ARRAY

Q01 Frequency Counter
Q02 Array Compression (Remove Duplicates)
Q03 Missing and Repeated Number
Q04 Print All Subarrays with Sum S
Q05 Array Rotation by K
Q06 Majority Element
Q07 Maximum Product of Three Numbers
Q08 Second Largest Distinct Number
Q09 Wave Rearrangement

STACK

Q10 Undo Operation Simulator
Q11 Min Stack
Q12 Two Stacks in One Array
Q13 Reverse a Stack

Q14 Next Greater Element
Q15 Evaluate Prefix Expression
Q16 Recursive Factorial Using Stack

QUEUE

Q17 Queue Using One Array
Q18 Reverse First K Elements of Queue
Q19 Job Queue with Time Slicing
Q20 Check if Queue is Palindrome
Q21 First Non-Repeating Character in Stream
Q22 Queue Rotation by K
Q23 Inter-Queue Swap
Q24 Queue Sorting Using One Additional Queue

LINKED LIST

Q25 Intersection Point of Two Linked Lists
Q26 Sort Elements in Linked List
Q27 Palindrome Check (Without Extra Array)
Q28 Remove Duplicates from Unsorted Linked List
Q29 Merge Two Sorted Linked Lists (No New Nodes)
Q30 Rotate Linked List Right by K

SCENARIO-BASED

Q35 Number of Ways to Climb Stairs
Q36 Count Books Before Reaching Target
Q37 Remove Duplicate Marbles
Q38 Sort Book Titles (Merge Sort)
Q39 Reverse List and Swap Nodes A & B
Q40 Balanced Parentheses Checker
Q41 Reverse First K Elements of Queue
Q42 Count Total Web Pages and Hyperlinks
Q43 Heap Sort

PART A — LAB PROGRAMS

1. ARRAYS

1.1 Array Initialization

Aim
To initialize an array with predefined values and display its elements.

Algorithm

1. Declare an integer array with a fixed size.
2. Initialize the array with predefined values.
3. Traverse the array using a loop.
4. Print each element.

Program

```
#include <stdio.h>

int main() {
   int numbers[5] = {10, 20, 30, 40, 50};
   int i;

   printf("Array elements are:\n");

   for (i = 0; i < 5; i++) {
      printf("Element at index %d = %d\n", i, numbers[i]);
   }

   return 0;
}
```

Output

```
Array elements are:
Element at index 0 = 10
Element at index 1 = 20
Element at index 2 = 30
Element at index 3 = 40
Element at index 4 = 50
```

Key Notes
• Array size is fixed at compile time.
• Elements are stored in contiguous memory locations.
• Indexing starts from 0.
• Direct initialization avoids runtime input overhead.

1.2 Linear Search

Aim
To search for a given element in an array using the linear search technique.

Algorithm

1. Read the number of elements and the array elements.
2. Read the element to be searched.
3. Compare the search element with each array element one by one.
4. If a match is found, display the position.
5. If no match is found after full traversal, display a not found message.

Program

```
#include <stdio.h>

int main() {
   int arr[50], n, key, i, found = 0;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   printf("Enter element to search: ");
   scanf("%d", &key);

   for (i = 0; i < n; i++) {
      if (arr[i] == key) {
         printf("Element %d found at position %d\n", key, i + 1);
         found = 1;
         break;
      }
   }

   if (found == 0) {
      printf("Element %d not found in the array\n", key);
   }

   return 0;
```

```
}
```

Output

```
Enter number of elements: 5
Enter 5 elements:
10 20 30 40 50
Enter element to search: 30
Element 30 found at position 3
```

Key Notes

- Linear search checks each element sequentially.
- Works on both sorted and unsorted arrays.
- Simple to implement but inefficient for large datasets.
- Best case time complexity is O(1), worst case is O(n).

1.3 Binary Search

Aim

To search for a given element in a sorted array using the binary search technique.

Algorithm

1. Read the number of elements and the sorted array elements.
2. Set low = 0 and high = n − 1.
3. Find mid = (low + high) / 2.
4. If the middle element matches the key, display its position.
5. If the key is smaller, search the left subarray.
6. If the key is larger, search the right subarray.
7. Repeat until the element is found or the search space becomes empty.

Program

```
#include <stdio.h>

int main() {
   int arr[50], n, key;
   int low, high, mid, i, found = 0;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter %d sorted elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   printf("Enter element to search: ");
   scanf("%d", &key);

   low = 0;
   high = n - 1;
```

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

        if (arr[mid] == key) {
            printf("Element %d found at position %d\n", key, mid + 1);
            found = 1;
            break;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    if (found == 0) {
        printf("Element %d not found in the array\n", key);
    }

    return 0;
}
```

Output

```
Enter number of elements: 5
Enter 5 sorted elements:
10 20 30 40 50
Enter element to search: 40
Element 40 found at position 4
```

Key Notes

- Binary search works only on **sorted arrays**.
- The search space is reduced to half in each step.
- Faster than linear search for large datasets.
- Time complexity is O(log n).

1.4 Binary Search (Recursion)

Aim

To search for a given element in a sorted array using recursive binary search.

Algorithm

1. Read the number of elements and the sorted array.
2. Define a recursive function with parameters: array, low, high, and key.
3. Find the middle element of the array.
4. If the middle element matches the key, return its position.
5. If the key is smaller, recursively search the left subarray.
6. If the key is larger, recursively search the right subarray.
7. If low becomes greater than high, the element is not found.

Program

```
#include <stdio.h>
```

```
int binarySearch(int arr[], int low, int high, int key) {
    int mid;

    if (low <= high) {
        mid = (low + high) / 2;

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

        if (arr[mid] > key)
            return binarySearch(arr, low, mid - 1, key);

        return binarySearch(arr, mid + 1, high, key);
    }

    return -1;
}

int main() {
    int arr[50], n, key, i, result;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d sorted elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter element to search: ");
    scanf("%d", &key);

    result = binarySearch(arr, 0, n - 1, key);

    if (result != -1)
        printf("Element %d found at position %d\n", key, result + 1);
    else
        printf("Element %d not found in the array\n", key);

    return 0;
}
```

Output

```
Enter number of elements: 5
Enter 5 sorted elements:
10 20 30 40 50
Enter element to search: 20
Element 20 found at position 2
```

Key Notes

- Uses recursion to divide the search space.
- Works only on sorted arrays.
- Each recursive call reduces the array size by half.
- Time complexity is O(log n), space complexity is O(log n).

1.5 Row Major Order

Aim
To display the elements of a matrix in row major order.

Algorithm

1. Read the number of rows and columns.
2. Read the matrix elements.
3. Traverse the matrix row by row.
4. Print each element in row major order.

Program

```
#include <stdio.h>

int main() {
   int matrix[10][10];
   int rows, cols, i, j;

   printf("Enter number of rows: ");
   scanf("%d", &rows);

   printf("Enter number of columns: ");
   scanf("%d", &cols);

   printf("Enter matrix elements:\n");
   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) {
         scanf("%d", &matrix[i][j]);
      }
   }

   printf("Matrix elements in Row Major Order:\n");
   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) {
         printf("%d ", matrix[i][j]);
      }
   }

   return 0;
}
```

Output

```
Enter number of rows: 2
Enter number of columns: 3
Enter matrix elements:
1 2 3
4 5 6
Matrix elements in Row Major Order:
1 2 3 4 5 6
```

Key Notes
• Row major order stores elements row by row.
• Consecutive elements of a row are stored in contiguous memory locations.
• This is the default storage order in C for 2D arrays.
• Traversal is done using row loop first, then column loop.

1.6 Column Major Order

Aim
To display the elements of a matrix in column major order.

Algorithm

1. Read the number of rows and columns.
2. Read the matrix elements.
3. Traverse the matrix column by column.
4. Print each element in column major order.

Program

```
#include <stdio.h>

int main() {
    int matrix[10][10];
    int rows, cols, i, j;

    printf("Enter number of rows: ");
    scanf("%d", &rows);

    printf("Enter number of columns: ");
    scanf("%d", &cols);

    printf("Enter matrix elements:\n");
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    printf("Matrix elements in Column Major Order:\n");
    for (j = 0; j < cols; j++) {
        for (i = 0; i < rows; i++) {
            printf("%d ", matrix[i][j]);
        }
    }

    return 0;
}
```

Output

```
Enter number of rows: 2
Enter number of columns: 3
Enter matrix elements:
1 2 3
4 5 6
Matrix elements in Column Major Order:
1 4 2 5 3 6
```

Key Notes
• Column major order accesses elements column by column.
• Consecutive elements of a column are accessed first.

• C does not store arrays in column major order by default.
• Useful for understanding memory access patterns.

1.7 Bubble Sort

Aim

To sort the elements of an array in ascending order using the bubble sort technique.

Algorithm

1. Read the number of elements and the array elements.
2. Compare adjacent elements of the array.
3. Swap the elements if they are in the wrong order.
4. Repeat the process for all elements until the array is sorted.

Program

```
#include <stdio.h>

int main() {
   int arr[50], n, i, j, temp;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   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;
         }
      }
   }

   printf("Sorted array:\n");
   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }

   return 0;
}
```

Output

```
Enter number of elements: 5
Enter 5 elements:
5 1 4 2 8
Sorted array:
```

1 2 4 5 8

Key Notes
• Bubble sort repeatedly compares adjacent elements.
• Larger elements “bubble” to the end of the array.
• Simple but inefficient for large data sets.
• Time complexity is $O(n^2)$.

1.8 Insertion Sort

Aim
To sort the elements of an array in ascending order using the insertion sort technique.

Algorithm

1. Read the number of elements and the array elements.
2. Assume the first element is already sorted.
3. Pick the next element and insert it into the correct position in the sorted part.
4. Repeat the process until all elements are sorted.

Program

```
#include <stdio.h>

int main() {
   int arr[50], n, i, j, key;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;

      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = key;
   }

   printf("Sorted array:\n");
   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }

   return 0;
}
```

Output

```
Enter number of elements: 5
Enter 5 elements:
9 5 1 4 3
Sorted array:
1 3 4 5 9
```

Key Notes

- Insertion sort builds the sorted array one element at a time.
- Efficient for small datasets and nearly sorted arrays.
- Stable sorting algorithm.
- Time complexity is $O(n^2)$.

1.9 Selection Sort

Aim

To sort the elements of an array in ascending order using the selection sort technique.

Algorithm

1. Read the number of elements and the array elements.
2. Select the smallest element from the unsorted part of the array.
3. Swap it with the first element of the unsorted part.
4. Move the boundary of the sorted part one position forward.
5. Repeat until the array is completely sorted.

Program

```
#include <stdio.h>

int main() {
    int arr[50], n, i, j, minIndex, temp;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

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

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
```

```
    return 0;
}
```

Output

```
Enter number of elements: 5
Enter 5 elements:
64 25 12 22 11
Sorted array:
11 12 22 25 64
```

Key Notes

- Selection sort selects the minimum element in each pass.
- Number of swaps is less compared to bubble sort.
- Not suitable for large datasets.
- Time complexity is $O(n^2)$.

1.10 Shell Sort

Aim

To sort the elements of an array in ascending order using the shell sort technique.

Algorithm

1. Read the number of elements and the array elements.
2. Initialize the gap as half of the array size.
3. Compare elements that are gap positions apart and sort them.
4. Reduce the gap and repeat the process.
5. Continue until the gap becomes 0.

Program

```
#include <stdio.h>

int main() {
    int arr[50], n, i, j, gap, temp;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (gap = n / 2; gap > 0; gap /= 2) {
        for (i = gap; i < n; i++) {
            temp = arr[i];
            j = i;
            while (j >= gap && arr[j - gap] > temp) {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = temp;
```

```
        }
    }

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 6
Enter 6 elements:
23 12 1 8 34 54
Sorted array:
1 8 12 23 34 54
```

Key Notes

- Shell sort is an improvement over insertion sort.
- Uses gap-based comparison to reduce swaps.
- Faster than simple quadratic sorts for large arrays.
- Average time complexity is better than $O(n^2)$.

1.11 Merge Sort

Aim

To sort the elements of an array in ascending order using the merge sort technique.

Algorithm

1. Read the number of elements and the array elements.
2. Divide the array into two halves.
3. Recursively sort each half.
4. Merge the two sorted halves into a single sorted array.
5. Repeat until the entire array is sorted.

Program

```
#include <stdio.h>

void merge(int arr[], int low, int mid, int high) {
    int temp[50];
    int i = low, j = mid + 1, k = low;

    while (i <= mid && j <= high) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while (i <= mid)
        temp[k++] = arr[i++];
```

```
    while (j <= high)
        temp[k++] = arr[j++];

    for (i = low; i <= high; i++)
        arr[i] = temp[i];
}

void mergeSort(int arr[], int low, int high) {
    int mid;
    if (low < high) {
        mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

int main() {
    int arr[50], n, i;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 5
Enter 5 elements:
38 27 43 3 9
Sorted array:
3 9 27 38 43
```

Key Notes

- Merge sort follows the divide and conquer technique.
- The array is repeatedly divided into smaller subarrays.
- Requires additional memory for merging.
- Time complexity is O(n log n).

1.12 Quick Sort

Aim

To sort the elements of an array in ascending order using the quick sort technique.

Algorithm

1. Read the number of elements and the array elements.
2. Choose a pivot element from the array.
3. Partition the array such that elements smaller than the pivot are on the left and larger elements are on the right.
4. Recursively apply quick sort on the left and right subarrays.
5. Continue until the array is completely sorted.

Program

```
#include <stdio.h>

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1, j, temp;

    for (j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    int pi;
    if (low < high) {
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int arr[50], n, i;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    quickSort(arr, 0, n - 1);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
```

```
    return 0;
}
```

Output

```
Enter number of elements: 5
Enter 5 elements:
10 7 8 9 1
Sorted array:
1 7 8 9 10
```

Key Notes

- Quick sort uses the divide and conquer approach.
- Sorting is done in-place without extra memory.
- Very efficient for large datasets.
- Average time complexity is O(n log n).

1.13 Magic Matrix

Aim

To generate and display a magic matrix of odd order.

Algorithm

1. Read an odd number n.
2. Initialize an n × n matrix with zeros.
3. Place number 1 in the middle column of the first row.
4. Move one position up and one position right for the next number.
5. If the position is already filled, move one position down instead.
6. Repeat until all numbers from 1 to n^2 are placed.

Program

```
#include <stdio.h>

int main() {
    int n, i, j, num;
    int row, col;

    printf("Enter an odd number: ");
    scanf("%d", &n);

    int magic[10][10] = {0};

    row = 0;
    col = n / 2;

    for (num = 1; num <= n * n; num++) {
        magic[row][col] = num;
        i = (row - 1 + n) % n;
        j = (col + 1) % n;

        if (magic[i][j] != 0)
            row = (row + 1) % n;
```

```
        else {
            row = i;
            col = j;
        }
    }

    printf("Magic Matrix:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%4d", magic[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output

```
Enter an odd number: 3
Magic Matrix:
 8  1  6
 3  5  7
 4  9  2
```

Key Notes

- Magic matrix works only for odd-sized matrices.
- Sum of every row, column, and diagonal is equal.
- Uses modular arithmetic for wrapping positions.
- Total elements are n^2.

1.14 Shell Sort (Version–2)

Aim

To sort the elements of an array in ascending order using an improved version of shell sort.

Algorithm

1. Read the number of elements and the array elements.
2. Initialize the gap value as n/2.
3. Perform insertion sort for elements separated by the gap.
4. Reduce the gap value and repeat the process.
5. Continue until the gap becomes 0.

Program

```
#include <stdio.h>

void shellSort(int arr[], int n) {
    int gap, i, j, temp;

    for (gap = n / 2; gap > 0; gap /= 2) {
        for (i = gap; i < n; i++) {
            temp = arr[i];
            j = i;
```

```
            while (j >= gap && arr[j - gap] > temp) {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = temp;
        }
    }
}

int main() {
    int arr[50], n, i;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    shellSort(arr, n);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 6
Enter 6 elements:
19 2 31 45 6 11
Sorted array:
2 6 11 19 31 45
```

Key Notes

- This version separates sorting logic into a function.
- Uses gap-based insertion sorting.
- Reduces total number of shifts compared to insertion sort.
- More efficient than simple quadratic sorting methods.

1.15 Bubble Sort (Row-wise)

Aim

To sort the elements of each row of a 2D array in ascending order using bubble sort.

Algorithm

1. Read the number of rows and columns.
2. Read the matrix elements.
3. For each row, apply bubble sort on its elements.
4. Display the row-wise sorted matrix.

Program

```
#include <stdio.h>

int main() {
    int mat[10][10];
    int r, c, i, j, k, temp;

    printf("Enter number of rows: ");
    scanf("%d", &r);

    printf("Enter number of columns: ");
    scanf("%d", &c);

    printf("Enter matrix elements:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &mat[i][j]);
        }
    }

    for (i = 0; i < r; i++) {
        for (j = 0; j < c - 1; j++) {
            for (k = 0; k < c - j - 1; k++) {
                if (mat[i][k] > mat[i][k + 1]) {
                    temp = mat[i][k];
                    mat[i][k] = mat[i][k + 1];
                    mat[i][k + 1] = temp;
                }
            }
        }
    }

    printf("Row-wise sorted matrix:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output

```
Enter number of rows: 2
Enter number of columns: 3
Enter matrix elements:
3 1 2
6 5 4
Row-wise sorted matrix:
1 2 3
4 5 6
```

Key Notes

- Each row is sorted independently.
- Bubble sort is applied row by row.

• Does not change column-wise order across rows.
• Time complexity is $O(r \times c^2)$.

1.16 Bubble Sort (Column-wise)

Aim
To sort the elements of each column of a 2D array in ascending order using bubble sort.

Algorithm

1. Read the number of rows and columns.
2. Read the matrix elements.
3. For each column, apply bubble sort on its elements.
4. Display the column-wise sorted matrix.

Program

```
#include <stdio.h>

int main() {
    int mat[10][10];
    int r, c, i, j, k, temp;

    printf("Enter number of rows: ");
    scanf("%d", &r);

    printf("Enter number of columns: ");
    scanf("%d", &c);

    printf("Enter matrix elements:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &mat[i][j]);
        }
    }

    for (j = 0; j < c; j++) {
        for (i = 0; i < r - 1; i++) {
            for (k = 0; k < r - i - 1; k++) {
                if (mat[k][j] > mat[k + 1][j]) {
                    temp = mat[k][j];
                    mat[k][j] = mat[k + 1][j];
                    mat[k + 1][j] = temp;
                }
            }
        }
    }

    printf("Column-wise sorted matrix:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }

    return 0;
```

```
}
```

Output

```
Enter number of rows: 3
Enter number of columns: 2
Enter matrix elements:
9 3
5 1
7 4
Column-wise sorted matrix:
5 1
7 3
9 4
```

Key Notes

- Each column is sorted independently.
- Bubble sort is applied column by column.
- Row order changes after sorting columns.
- Time complexity is $O(c \times r^2)$.

1.17 Bubble Sort (2D Array)

Aim

To sort the elements of a 2D array using bubble sort.

Algorithm

1. Read the number of rows and columns.
2. Read the matrix elements.
3. Apply bubble sort on each row of the matrix.
4. Display the sorted 2D array.

Program

```
#include <stdio.h>

int main() {
    int mat[10][10];
    int r, c, i, j, k, temp;

    printf("Enter number of rows: ");
    scanf("%d", &r);

    printf("Enter number of columns: ");
    scanf("%d", &c);

    printf("Enter matrix elements:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &mat[i][j]);
        }
    }

    for (i = 0; i < r; i++) {
```

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

    printf("Sorted 2D array:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output

```
Enter number of rows: 2
Enter number of columns: 3
Enter matrix elements:
6 2 4
5 1 3
Sorted 2D array:
2 4 6
1 3 5
```

Key Notes

- Bubble sort is applied row-wise on the 2D array.
- Each row is sorted independently.
- Simple to implement for matrix sorting.
- Time complexity is $O(r \times c^2)$.

1.18 Row Major Address Calculation

Aim

To calculate and display the memory addresses of elements in a matrix using row major order.

Algorithm

1. Read the number of rows and columns.
2. Read the base address and size of each element.
3. Use the row major address formula to calculate the address of each element.
4. Display the address for every matrix element.

Program

```
#include <stdio.h>
```

```
int main() {
   int r, c, i, j;
   int base, size;
   int mat[10][10];

   printf("Enter number of rows: ");
   scanf("%d", &r);

   printf("Enter number of columns: ");
   scanf("%d", &c);

   printf("Enter base address: ");
   scanf("%d", &base);

   printf("Enter size of each element: ");
   scanf("%d", &size);

   printf("Row Major Address Calculation:\n");
   for (i = 0; i < r; i++) {
      for (j = 0; j < c; j++) {
         printf("Address of A[%d][%d] = %d\n",
               i, j, base + ((i * c) + j) * size);
      }
   }

   return 0;
}
```

Output

```
Enter number of rows: 2
Enter number of columns: 3
Enter base address: 1000
Enter size of each element: 4
Row Major Address Calculation:
Address of A[0][0] = 1000
Address of A[0][1] = 1004
Address of A[0][2] = 1008
Address of A[1][0] = 1012
Address of A[1][1] = 1016
Address of A[1][2] = 1020
```

Key Notes

- Row major order stores rows contiguously in memory.
- Address formula: Base + ((i × columns) + j) × size.
- Used internally by C for 2D arrays.
- Efficient for row-wise traversal.

1.19 Column Major Address Calculation

Aim

To calculate and display the memory addresses of elements in a matrix using column major order.

Algorithm

1. Read the number of rows and columns.
2. Read the base address and size of each element.
3. Use the column major address formula to calculate the address of each element.
4. Display the address for every matrix element.

Program

```
#include <stdio.h>

int main() {
   int r, c, i, j;
   int base, size;
   int mat[10][10];

   printf("Enter number of rows: ");
   scanf("%d", &r);

   printf("Enter number of columns: ");
   scanf("%d", &c);

   printf("Enter base address: ");
   scanf("%d", &base);

   printf("Enter size of each element: ");
   scanf("%d", &size);

   printf("Column Major Address Calculation:\n");
   for (j = 0; j < c; j++) {
      for (i = 0; i < r; i++) {
         printf("Address of A[%d][%d] = %d\n",
               i, j, base + ((j * r) + i) * size);
      }
   }

   return 0;
}
```

Output

```
Enter number of rows: 2
Enter number of columns: 3
Enter base address: 1000
Enter size of each element: 4
Column Major Address Calculation:
Address of A[0][0] = 1000
Address of A[1][0] = 1004
Address of A[0][1] = 1008
Address of A[1][1] = 1012
Address of A[0][2] = 1016
Address of A[1][2] = 1020
```

Key Notes

- Column major order stores columns contiguously in memory.
- Address formula: Base + ((j × rows) + i) × size.
- Used in languages like Fortran.
- Helpful for understanding memory organization.

2. Linked List

2.1 Singly Linked List (Insertion, Deletion, Search)

Aim
To perform insertion, deletion, and search operations on a singly linked list.

Algorithm
Insertion

1. Create a new node and assign data.
2. Link the new node at the end of the list.

Deletion

1. Read the position to be deleted.
2. Adjust the links to remove the node.

Search

1. Traverse the list node by node.
2. Compare each node's data with the search key.
3. Display the position if found.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

void insert(struct node **head, int value) {
   struct node *newNode = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   newNode->data = value;
   newNode->next = NULL;

   if (*head == NULL) {
      *head = newNode;
      return;
   }

   temp = *head;
   while (temp->next != NULL)
      temp = temp->next;

   temp->next = newNode;
}

void deleteNode(struct node **head, int pos) {
   struct node *temp = *head, *prev = NULL;
```

```
    int i;

    if (pos == 1 && temp != NULL) {
        *head = temp->next;
        free(temp);
        return;
    }

    for (i = 1; temp != NULL && i < pos; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL)
        return;

    prev->next = temp->next;
    free(temp);
}

void search(struct node *head, int key) {
    int pos = 1;
    while (head != NULL) {
        if (head->data == key) {
            printf("Element %d found at position %d\n", key, pos);
            return;
        }
        head = head->next;
        pos++;
    }
    printf("Element not found\n");
}

void display(struct node *head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct node *head = NULL;
    int choice, value, pos;

    do {
        printf("\n1.Insert 2.Delete 3.Search 4.Display 5.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insert(&head, value);
                break;

            case 2:
                printf("Enter position to delete: ");
                scanf("%d", &pos);
```

```
            deleteNode(&head, pos);
            break;

        case 3:
            printf("Enter value to search: ");
            scanf("%d", &value);
            search(head, value);
            break;

        case 4:
            display(head);
            break;
    }
  } while (choice != 5);

  return 0;
}
```

Output

```
1.Insert  2.Delete  3.Search  4.Display  5.Exit
Enter choice: 1
Enter value: 10
10 -> NULL
```

Key Notes

- Each node contains data and a link to the next node.
- Dynamic memory allocation is used.
- Insertion and deletion do not require shifting elements.
- Traversal is required for search operations.

2.2 Circular Linked List (Insertion)

Aim

To perform insertion operations on a circular linked list.

Algorithm

1. Create a new node and assign data.
2. If the list is empty, make the new node point to itself.
3. Otherwise, traverse to the last node.
4. Insert the new node and update the last node's link to the head.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
  int data;
  struct node *next;
};

struct node *head = NULL;
```

```
void insert(int value) {
   struct node *newNode = (struct node *)malloc(sizeof(struct node));
   struct node *temp;

   newNode->data = value;

   if (head == NULL) {
      head = newNode;
      newNode->next = head;
      return;
   }

   temp = head;
   while (temp->next != head)
      temp = temp->next;

   temp->next = newNode;
   newNode->next = head;
}

void display() {
   struct node *temp = head;

   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

   do {
      printf("%d -> ", temp->data);
      temp = temp->next;
   } while (temp != head);
   printf("(back to head)\n");
}

int main() {
   int n, value, i;

   printf("Enter number of nodes: ");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      printf("Enter value: ");
      scanf("%d", &value);
      insert(value);
   }

   display();
   return 0;
}
```

Output

```
Enter number of nodes: 3
Enter value: 10
Enter value: 20
Enter value: 30
10 -> 20 -> 30 -> (back to head)
```

Key Notes
• Last node points back to the first node.
• There is no NULL link in circular linked lists.
• Traversal stops when the head node is reached again.
• Useful in applications requiring cyclic traversal.

2.3 Circular Linked List (Deletion)

Aim
To perform deletion operations on a circular linked list.

Algorithm

1. Check if the list is empty.
2. If the list has only one node, delete it and set head to NULL.
3. Otherwise, traverse to the node to be deleted.
4. Adjust the links to remove the node from the list.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

struct node *head = NULL;

void insert(int value) {
   struct node *newNode = (struct node *)malloc(sizeof(struct node));
   struct node *temp;

   newNode->data = value;

   if (head == NULL) {
      head = newNode;
      newNode->next = head;
      return;
   }

   temp = head;
   while (temp->next != head)
      temp = temp->next;

   temp->next = newNode;
   newNode->next = head;
}

void deleteNode(int key) {
   struct node *curr = head, *prev = NULL;

   if (head == NULL) {
      printf("List is empty\n");
      return;
```

```
    }

    if (curr->data == key && curr->next == head) {
        free(curr);
        head = NULL;
        return;
    }

    if (curr->data == key) {
        while (curr->next != head)
            curr = curr->next;

        curr->next = head->next;
        free(head);
        head = curr->next;
        return;
    }

    prev = head;
    curr = head->next;
    while (curr != head) {
        if (curr->data == key) {
            prev->next = curr->next;
            free(curr);
            return;
        }
        prev = curr;
        curr = curr->next;
    }

    printf("Element not found\n");
}

void display() {
    struct node *temp = head;

    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(back to head)\n");
}

int main() {
    int n, value, key, i;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("Enter value: ");
        scanf("%d", &value);
        insert(value);
    }
```

```
    display();

    printf("Enter element to delete: ");
    scanf("%d", &key);
    deleteNode(key);

    display();
    return 0;
}
```

Output

```
Enter number of nodes: 3
Enter value: 10
Enter value: 20
Enter value: 30
10 -> 20 -> 30 -> (back to head)
Enter element to delete: 20
10 -> 30 -> (back to head)
```

Key Notes

- Deletion handles beginning, middle, and last nodes.
- Special care is required since there is no NULL pointer.
- Circular structure must be maintained after deletion.
- Useful in scheduling and buffer management.

2.4 Doubly Linked List (Insertion, Deletion, Search)

Aim

To perform insertion, deletion, and search operations on a doubly linked list.

Algorithm

Insertion

1. Create a new node with given data.
2. If the list is empty, make the new node as head.
3. Otherwise, insert the node at the end by adjusting next and previous links.

Deletion

1. Read the position of the node to be deleted.
2. Traverse to the required node.
3. Update the previous and next pointers to remove the node.

Search

1. Traverse the list from head.
2. Compare each node's data with the search key.
3. Display the position if found.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *prev;
   struct node *next;
};

struct node *head = NULL;

void insert(int value) {
   struct node *newNode = (struct node *)malloc(sizeof(struct node));
   struct node *temp;

   newNode->data = value;
   newNode->prev = NULL;
   newNode->next = NULL;

   if (head == NULL) {
      head = newNode;
      return;
   }

   temp = head;
   while (temp->next != NULL)
      temp = temp->next;

   temp->next = newNode;
   newNode->prev = temp;
}

void deleteNode(int pos) {
   struct node *temp = head;
   int i;

   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

   if (pos == 1) {
      head = temp->next;
      if (head != NULL)
         head->prev = NULL;
      free(temp);
      return;
   }

   for (i = 1; temp != NULL && i < pos; i++)
      temp = temp->next;

   if (temp == NULL) {
      printf("Invalid position\n");
      return;
   }

   if (temp->next != NULL)
      temp->next->prev = temp->prev;
```

```
    temp->prev->next = temp->next;
    free(temp);
}

void search(int key) {
    struct node *temp = head;
    int pos = 1;

    while (temp != NULL) {
        if (temp->data == key) {
            printf("Element %d found at position %d\n", key, pos);
            return;
        }
        temp = temp->next;
        pos++;
    }
    printf("Element not found\n");
}

void display() {
    struct node *temp = head;
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    int choice, value, pos;

    do {
        printf("\n1.Insert 2.Delete 3.Search 4.Display 5.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insert(value);
                break;

            case 2:
                printf("Enter position to delete: ");
                scanf("%d", &pos);
                deleteNode(pos);
                break;

            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                search(value);
                break;

            case 4:
                display();
                break;
        }
    } while (choice != 5);
```

```
    return 0;
}
```

Output

```
1.Insert  2.Delete  3.Search  4.Display  5.Exit
Enter choice: 1
Enter value: 10
10 <-> NULL
```

Key Notes
- Each node contains two links: previous and next.
- Traversal is possible in both forward and backward directions.
- Deletion is easier compared to singly linked lists.
- Requires extra memory for the previous pointer.

3. Stack

3.1 Stack Operations (Push, Pop, Peek, Display)

Aim
To implement basic stack operations using an array.

Algorithm

Push

1. Check stack overflow.
2. Increment top and insert the element.

Pop

1. Check stack underflow.
2. Remove the top element.

Peek

1. Display the top element without removing it.

Display

1. Print elements from top to bottom.

Program

```
#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;
```

```
void push(int value) {
   if (top == MAX - 1) {
      printf("Stack Overflow\n");
      return;
   }
   stack[++top] = value;
}

void pop() {
   if (top == -1) {
      printf("Stack Underflow\n");
      return;
   }
   printf("Popped element: %d\n", stack[top--]);
}

void peek() {
   if (top == -1) {
      printf("Stack is empty\n");
      return;
   }
   printf("Top element: %d\n", stack[top]);
}

void display() {
   int i;
   if (top == -1) {
      printf("Stack is empty\n");
      return;
   }
   for (i = top; i >= 0; i--) {
      printf("%d ", stack[i]);
   }
   printf("\n");
}

int main() {
   int choice, value;

   do {
      printf("\n1.Push 2.Pop 3.Peek 4.Display 5.Exit\n");
      printf("Enter choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter value: ");
            scanf("%d", &value);
            push(value);
            break;
         case 2:
            pop();
            break;
         case 3:
            peek();
            break;
         case 4:
            display();
            break;
```

```
        }
    } while (choice != 5);

    return 0;
}
```

Output

```
1.Push 2.Pop 3.Peek 4.Display 5.Exit
Enter choice: 1
Enter value: 10
1.Push 2.Pop 3.Peek 4.Display 5.Exit
Enter choice: 4
10
```

Key Notes

- Stack follows LIFO principle.
- Insert and delete operations occur at the top.
- Overflow and underflow must be handled.
- Array-based stack has fixed size.

3.2 Infix to Postfix Conversion

Aim

To convert an infix expression into a postfix expression using a stack.

Algorithm

1. Initialize an empty stack for operators.
2. Scan the infix expression from left to right.
3. If the symbol is an operand, add it to the postfix expression.
4. If the symbol is an operator, pop operators with higher or equal precedence from the stack and add them to postfix.
5. Push the current operator onto the stack.
6. If the symbol is (, push it onto the stack.
7. If the symbol is), pop operators until (is found.
8. Pop remaining operators from the stack after scanning the expression.

Program

```
#include <stdio.h>
#include <ctype.h>

#define MAX 50

char stack[MAX];
int top = -1;

void push(char ch) {
    stack[++top] = ch;
}

char pop() {
    return stack[top--];
```

```
}

int precedence(char ch) {
    if (ch == '+' || ch == '-')
        return 1;
    if (ch == '*' || ch == '/')
        return 2;
    return 0;
}

int main() {
    char infix[MAX], postfix[MAX];
    int i = 0, k = 0;
    char ch;

    printf("Enter infix expression: ");
    scanf("%s", infix);

    while ((ch = infix[i++]) != '\0') {
        if (isalnum(ch)) {
            postfix[k++] = ch;
        } else if (ch == '(') {
            push(ch);
        } else if (ch == ')') {
            while (stack[top] != '(')
                postfix[k++] = pop();
            pop();
        } else {
            while (top != -1 && precedence(stack[top]) >= precedence(ch))
                postfix[k++] = pop();
            push(ch);
        }
    }

    while (top != -1)
        postfix[k++] = pop();

    postfix[k] = '\0';
    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

Output

```
Enter infix expression: A+B*C
Postfix expression: ABC*+
```

Key Notes

- Stack temporarily stores operators.
- Operator precedence controls popping order.
- Postfix expressions do not need parentheses.
- Used in expression evaluation.

3.3 Undo Operation Simulator

Aim

To simulate an undo operation using stack data structure.

Algorithm

1. Initialize an empty stack.
2. Push each performed operation onto the stack.
3. To undo, pop the top element from the stack.
4. Display the remaining operations.

Program

```
#include <stdio.h>
#define MAX 10

int stack[MAX];
int top = -1;

void push(int value) {
   if (top == MAX - 1) {
      printf("Stack Overflow\n");
      return;
   }
   stack[++top] = value;
}

void undo() {
   if (top == -1) {
      printf("No operation to undo\n");
      return;
   }
   printf("Undo operation: %d\n", stack[top--]);
}

void display() {
   int i;
   if (top == -1) {
      printf("No operations available\n");
      return;
   }
   printf("Current operations:\n");
   for (i = top; i >= 0; i--) {
      printf("%d ", stack[i]);
   }
   printf("\n");
}

int main() {
   int choice, value;

   do {
      printf("\n1.Perform Operation  2.Undo  3.Display  4.Exit\n");
      printf("Enter choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter operation value: ");
```

```
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            undo();
            break;
        case 3:
            display();
            break;
    }
  } while (choice != 4);

  return 0;
}
```

Output

```
1.Perform Operation  2.Undo  3.Display  4.Exit
Enter choice: 1
Enter operation value: 5
1.Perform Operation  2.Undo  3.Display  4.Exit
Enter choice: 2
Undo operation: 5
```

Key Notes

- Undo follows the LIFO principle.
- Latest operation is undone first.
- Stack efficiently manages operation history.
- Commonly used in text editors and applications.

3.4 Min Stack (Get Minimum in O(1))

Aim

To implement a stack that supports retrieving the minimum element in constant time.

Algorithm

1. Maintain two stacks: main stack and minimum stack.
2. On push, insert the element into the main stack.
3. If the minimum stack is empty or the element is smaller than or equal to the top of minimum stack, push it into the minimum stack.
4. On pop, remove the element from the main stack.
5. If the popped element is equal to the top of minimum stack, pop it from the minimum stack.
6. The top of the minimum stack always stores the current minimum element.

Program

```
#include <stdio.h>
#define MAX 10

int stack[MAX], minStack[MAX];
int top = -1, minTop = -1;
```

```
void push(int value) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = value;

    if (minTop == -1 || value <= minStack[minTop]) {
        minStack[++minTop] = value;
    }
}

void pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return;
    }

    if (stack[top] == minStack[minTop]) {
        minTop--;
    }

    printf("Popped element: %d\n", stack[top--]);
}

void getMin() {
    if (minTop == -1) {
        printf("Stack is empty\n");
        return;
    }
    printf("Minimum element: %d\n", minStack[minTop]);
}

int main() {
    int choice, value;

    do {
        printf("\n1.Push  2.Pop  3.Get Min  4.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                getMin();
                break;
        }
    } while (choice != 4);

    return 0;
}
```

Output

```
1.Push  2.Pop  3.Get Min  4.Exit
Enter choice: 1
Enter value: 5
Enter choice: 1
Enter value: 2
Enter choice: 3
Minimum element: 2
```

Key Notes

- Uses an auxiliary stack to track minimum values.
- Minimum retrieval is done in O(1) time.
- Push and pop operations also work in O(1).
- Requires extra space for the minimum stack.

3.5 Two Stacks in One Array

Aim

To implement two stacks using a single array.

Algorithm

1. Initialize two stack pointers:
    - Stack1 starts from the beginning of the array.
    - Stack2 starts from the end of the array.
2. Push in Stack1 from left to right.
3. Push in Stack2 from right to left.
4. Overflow occurs when the two stack pointers cross each other.
5. Pop elements independently from both stacks.

Program

```
#include <stdio.h>
#define MAX 10

int arr[MAX];
int top1 = -1;
int top2 = MAX;

void push1(int value) {
   if (top1 + 1 == top2) {
      printf("Stack Overflow\n");
      return;
   }
   arr[++top1] = value;
}

void push2(int value) {
   if (top1 + 1 == top2) {
      printf("Stack Overflow\n");
      return;
   }
   arr[--top2] = value;
}

void pop1() {
```

```
    if (top1 == -1) {
        printf("Stack1 Underflow\n");
        return;
    }
    printf("Popped from Stack1: %d\n", arr[top1--]);
}

void pop2() {
    if (top2 == MAX) {
        printf("Stack2 Underflow\n");
        return;
    }
    printf("Popped from Stack2: %d\n", arr[top2++]);
}

int main() {
    push1(10);
    push1(20);
    push2(30);
    push2(40);

    pop1();
    pop2();

    return 0;
}
```

Output

```
Popped from Stack1: 20
Popped from Stack2: 40
```

Key Notes

- Efficient use of memory by sharing a single array.
- Stack1 grows from left, Stack2 grows from right.
- Overflow occurs only when the array is completely full.
- Reduces wasted space compared to separate arrays.

3.6 Reverse a Stack

Aim

To reverse the elements of a stack using stack operations.

Algorithm

1. Push elements into the stack.
2. Use an auxiliary stack to pop elements from the original stack.
3. Push the popped elements into the auxiliary stack.
4. Display the reversed stack.

Program

```
#include <stdio.h>
#define MAX 10
```

```
int stack[MAX], tempStack[MAX];
int top = -1, tempTop = -1;

void push(int value) {
   stack[++top] = value;
}

int pop() {
   return stack[top--];
}

int main() {
   int i, n, value;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &value);
      push(value);
   }

   while (top != -1) {
      tempStack[++tempTop] = pop();
   }

   printf("Reversed stack:\n");
   for (i = 0; i <= tempTop; i++) {
      printf("%d ", tempStack[i]);
   }

   return 0;
}
```

Output

```
Enter number of elements: 4
Enter elements:
1 2 3 4
Reversed stack:
4 3 2 1
```

Key Notes

- Uses an auxiliary stack to reverse elements.
- Follows LIFO principle.
- Simple and easy to implement.
- Requires extra memory.

3.7 Next Greater Element

Aim

To find the next greater element for each element in an array using a stack.

Algorithm

1. Read the number of elements and the array elements.
2. Initialize an empty stack to store indices.
3. Traverse the array from left to right.
4. While the stack is not empty and the current element is greater than the element at the stack's top index, pop the index and print the next greater element.
5. Push the current index onto the stack.
6. For remaining elements in the stack, print -1 as no greater element exists.

Program

```
#include <stdio.h>
#define MAX 50

int stack[MAX];
int top = -1;

void push(int index) {
   stack[++top] = index;
}

int pop() {
   return stack[top--];
}

int isEmpty() {
   return top == -1;
}

int main() {
   int arr[50], n, i, index;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   for (i = 0; i < n; i++) {
      while (!isEmpty() && arr[i] > arr[stack[top]]) {
         index = pop();
         printf("Next greater element of %d is %d\n", arr[index], arr[i]);
      }
      push(i);
   }

   while (!isEmpty()) {
      index = pop();
      printf("Next greater element of %d is -1\n", arr[index]);
   }

   return 0;
}
```

Output

```
Enter number of elements: 4
```

```
Enter 4 elements:
4 5 2 25
Next greater element of 4 is 5
Next greater element of 5 is 25
Next greater element of 2 is 25
Next greater element of 25 is -1
```

Key Notes

- Uses a stack to efficiently track unresolved elements.
- Each element is pushed and popped at most once.
- More efficient than nested loops.
- Time complexity is O(n).

3.8 Evaluate Prefix Expression

Aim

To evaluate a given prefix expression using stack.

Algorithm

1. Read the prefix expression.
2. Initialize an empty stack.
3. Scan the prefix expression from right to left.
4. If the symbol is an operand, push it onto the stack.
5. If the symbol is an operator, pop two operands from the stack.
6. Apply the operator on the operands and push the result back onto the stack.
7. After scanning the expression, the top of the stack contains the final result.

Program

```
#include <stdio.h>
#include <ctype.h>
#define MAX 50

int stack[MAX];
int top = -1;

void push(int value) {
   stack[++top] = value;
}

int pop() {
   return stack[top--];
}

int main() {
   char prefix[MAX];
   int i, op1, op2, result;

   printf("Enter prefix expression: ");
   scanf("%s", prefix);

   for (i = strlen(prefix) - 1; i >= 0; i--) {
      if (isdigit(prefix[i])) {
         push(prefix[i] - '0');
```

```
        } else {
            op1 = pop();
            op2 = pop();

            switch (prefix[i]) {
                case '+': result = op1 + op2; break;
                case '-': result = op1 - op2; break;
                case '*': result = op1 * op2; break;
                case '/': result = op1 / op2; break;
            }
            push(result);
        }
    }

    printf("Result: %d\n", pop());
    return 0;
}
```

Output

```
Enter prefix expression: +9*26
Result: 21
```

Key Notes

- Prefix expression is scanned from right to left.
- Stack is used to store operands.
- Operators are applied immediately when encountered.
- Evaluation is efficient and follows stack principles.

3.9 Simulate Recursive Factorial

Aim

To simulate the recursive calculation of factorial using a stack.

Algorithm

1. Read a number n.
2. Push values from n down to 1 onto the stack.
3. Initialize result as 1.
4. Pop each element from the stack and multiply it with result.
5. Display the final factorial value.

Program

```
#include <stdio.h>
#define MAX 20

int stack[MAX];
int top = -1;

void push(int value) {
    stack[++top] = value;
}

int pop() {
```

```
    return stack[top--];
}

int main() {
    int n, i, result = 1;

    printf("Enter a number: ");
    scanf("%d", &n);

    for (i = n; i >= 1; i--) {
        push(i);
    }

    while (top != -1) {
        result *= pop();
    }

    printf("Factorial of %d is %d\n", n, result);

    return 0;
}
```

Output

```
Enter a number: 5
Factorial of 5 is 120
```

Key Notes
• Stack simulates recursive function calls.
• Each pushed value represents a recursive call.
• Popping simulates returning from recursion.
• Useful for understanding recursion internals.

4. Queue

4.1 Queue Using Array

Aim
To implement basic queue operations using an array.

Algorithm

Enqueue

1. Check if the queue is full.
2. Insert the element at the rear end.

Dequeue

1. Check if the queue is empty.
2. Remove the element from the front end.

Display

1. Traverse the queue from front to rear and print elements.

Program

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = -1, rear = -1;

void enqueue(int value) {
   if (rear == MAX - 1) {
      printf("Queue Overflow\n");
      return;
   }
   if (front == -1)
      front = 0;
   queue[++rear] = value;
}

void dequeue() {
   if (front == -1 || front > rear) {
      printf("Queue Underflow\n");
      return;
   }
   printf("Dequeued element: %d\n", queue[front++]);
}

void display() {
   int i;
   if (front == -1 || front > rear) {
      printf("Queue is empty\n");
      return;
   }
   for (i = front; i <= rear; i++) {
      printf("%d ", queue[i]);
   }
   printf("\n");
}

int main() {
   int choice, value;

   do {
      printf("\n1.Enqueue 2.Dequeue 3.Display 4.Exit\n");
      printf("Enter choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter value: ");
            scanf("%d", &value);
            enqueue(value);
            break;
         case 2:
            dequeue();
            break;
         case 3:
            display();
```

```
            break;
        }
    } while (choice != 4);

    return 0;
}
```

Output

```
1.Enqueue 2.Dequeue 3.Display 4.Exit
Enter choice: 1
Enter value: 10
Enter choice: 1
Enter value: 20
Enter choice: 3
10 20
```

Key Notes

- Queue follows **FIFO (First In First Out)** principle.
- Insertion happens at rear and deletion at front.
- Array-based queue may waste space after deletions.
- Overflow and underflow conditions must be handled.

4.2 Circular Queue

Aim

To implement a circular queue using an array.

Algorithm

Enqueue

1. Check if the queue is full: `(rear + 1) % MAX == front`.
2. If empty, set `front = 0`.
3. Insert the element at `rear = (rear + 1) % MAX`.

Dequeue

1. Check if the queue is empty: `front == -1`.
2. Remove the element at `front`.
3. If `front == rear`, reset both to -1.
4. Otherwise, update `front = (front + 1) % MAX`.

Display

1. Traverse from `front` to `rear` circularly and print elements.

Program

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
```

```
int front = -1, rear = -1;

void enqueue(int value) {
    if ((rear + 1) % MAX == front) {
        printf("Queue Overflow\n");
        return;
    }
    if (front == -1)
        front = 0;
    rear = (rear + 1) % MAX;
    queue[rear] = value;
}

void dequeue() {
    if (front == -1) {
        printf("Queue Underflow\n");
        return;
    }
    printf("Dequeued element: %d\n", queue[front]);

    if (front == rear)
        front = rear = -1;
    else
        front = (front + 1) % MAX;
}

void display() {
    int i;
    if (front == -1) {
        printf("Queue is empty\n");
        return;
    }

    i = front;
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear)
            break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

int main() {
    int choice, value;

    do {
        printf("\n1.Enqueue 2.Dequeue 3.Display 4.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
```

```
            case 3:
                display();
                break;
        }
    } while (choice != 4);

    return 0;
}
```

Output

```
1.Enqueue 2.Dequeue 3.Display 4.Exit
Enter choice: 1
Enter value: 10
Enter choice: 1
Enter value: 20
Enter choice: 3
10 20
```

Key Notes

- Circular queue reuses unused spaces.
- Eliminates space wastage of linear queue.
- Uses modulo operation for circular movement.
- Efficient for fixed-size buffers.

4.3 Priority Queue (Linked List Based)

Aim

To implement a priority queue using a linked list.

Algorithm

1. Create a new node with data and priority.
2. If the queue is empty or the new node has higher priority, insert it at the beginning.
3. Otherwise, traverse the list and insert the node at the appropriate position based on priority.
4. Deletion always removes the element with the highest priority.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    int priority;
    struct node *next;
};

struct node *front = NULL;

void insert(int data, int priority) {
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    struct node *temp;
```

```
    newNode->data = data;
    newNode->priority = priority;
    newNode->next = NULL;

    if (front == NULL || priority < front->priority) {
        newNode->next = front;
        front = newNode;
        return;
    }

    temp = front;
    while (temp->next != NULL && temp->next->priority <= priority) {
        temp = temp->next;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

void delete() {
    struct node *temp;

    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }

    temp = front;
    front = front->next;
    printf("Deleted element: %d\n", temp->data);
    free(temp);
}

void display() {
    struct node *temp = front;

    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }

    while (temp != NULL) {
        printf("(%d, %d) ", temp->data, temp->priority);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    insert(10, 2);
    insert(20, 1);
    insert(30, 3);

    display();
    delete();
    display();

    return 0;
}
```

Output

(20, 1) (10, 2) (30, 3)
Deleted element: 20
(10, 2) (30, 3)

Key Notes
• Each element has an associated priority.
• Lower priority value indicates higher priority.
• Linked list maintains sorted order by priority.
• Insertion is O(n), deletion is O(1).

4.4 Deque (Double Ended Queue)

Aim
To implement a double ended queue (deque) using an array.

Algorithm

Insert Front

1. Check if the deque is full.
2. Decrement front and insert the element.

Insert Rear

1. Check if the deque is full.
2. Increment rear and insert the element.

Delete Front

1. Check if the deque is empty.
2. Remove the element from the front.

Delete Rear

1. Check if the deque is empty.
2. Remove the element from the rear.

Program

```
#include <stdio.h>
#define MAX 5

int deque[MAX];
int front = -1, rear = -1;

void insertFront(int value) {
   if (front == 0) {
      printf("Deque Overflow\n");
      return;
   }
```

```
    if (front == -1) {
        front = rear = 0;
    } else {
        front--;
    }
    deque[front] = value;
}

void insertRear(int value) {
    if (rear == MAX - 1) {
        printf("Deque Overflow\n");
        return;
    }
    if (rear == -1) {
        front = rear = 0;
    } else {
        rear++;
    }
    deque[rear] = value;
}

void deleteFront() {
    if (front == -1) {
        printf("Deque Underflow\n");
        return;
    }
    printf("Deleted from front: %d\n", deque[front]);
    if (front == rear)
        front = rear = -1;
    else
        front++;
}

void deleteRear() {
    if (rear == -1) {
        printf("Deque Underflow\n");
        return;
    }
    printf("Deleted from rear: %d\n", deque[rear]);
    if (front == rear)
        front = rear = -1;
    else
        rear--;
}

void display() {
    int i;
    if (front == -1) {
        printf("Deque is empty\n");
        return;
    }
    for (i = front; i <= rear; i++) {
        printf("%d ", deque[i]);
    }
    printf("\n");
}

int main() {
    insertRear(10);
    insertRear(20);
```

```
    insertFront(5);
    display();
    deleteFront();
    deleteRear();
    display();

    return 0;
}
```

Output

```
5 10 20
Deleted from front: 5
Deleted from rear: 20
10
```

Key Notes

- Deque allows insertion and deletion at both ends.
- Combines features of stack and queue.
- Useful in scheduling and sliding window problems.
- Array-based deque has fixed size.

5. Graphs

5.1 Breadth First Search (BFS)

Aim

To traverse a graph using Breadth First Search (BFS) technique.

Algorithm

1. Read the number of vertices and the adjacency matrix of the graph.
2. Initialize a visited array to mark visited vertices.
3. Insert the starting vertex into the queue and mark it as visited.
4. Remove a vertex from the queue and display it.
5. Visit all its adjacent unvisited vertices, mark them visited, and insert them into the queue.
6. Repeat until the queue becomes empty.

Program

```
#include <stdio.h>
#define MAX 10

int queue[MAX], front = -1, rear = -1;
int visited[MAX];

void enqueue(int v) {
    if (rear == MAX - 1)
        return;
    if (front == -1)
        front = 0;
    queue[++rear] = v;
```

```
}

int dequeue() {
   return queue[front++];
}

int isEmpty() {
   return front == -1 || front > rear;
}

int main() {
   int n, graph[MAX][MAX], i, j, start;

   printf("Enter number of vertices: ");
   scanf("%d", &n);

   printf("Enter adjacency matrix:\n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         scanf("%d", &graph[i][j]);
      }
      visited[i] = 0;
   }

   printf("Enter starting vertex: ");
   scanf("%d", &start);

   enqueue(start);
   visited[start] = 1;

   printf("BFS Traversal: ");
   while (!isEmpty()) {
      int v = dequeue();
      printf("%d ", v);

      for (i = 0; i < n; i++) {
         if (graph[v][i] == 1 && !visited[i]) {
            enqueue(i);
            visited[i] = 1;
         }
      }
   }

   return 0;
}
```

Output

```
Enter number of vertices: 4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 0
0 1 0 0
Enter starting vertex: 0
BFS Traversal: 0 1 2 3
```

Key Notes

• BFS explores vertices level by level.

• Uses a queue for traversal.
• Ensures minimum distance traversal in unweighted graphs.
• Useful in shortest path and connectivity problems.

5.2 Depth First Search (DFS – Adjacency Matrix)

Aim
To traverse a graph using Depth First Search (DFS) technique with adjacency matrix representation.

Algorithm

1. Read the number of vertices and the adjacency matrix.
2. Initialize a visited array.
3. Start DFS from a given vertex.
4. Mark the current vertex as visited and display it.
5. Recursively visit all adjacent unvisited vertices.
6. Repeat until all reachable vertices are visited.

Program

```
#include <stdio.h>
#define MAX 10

int graph[MAX][MAX];
int visited[MAX];
int n;

void dfs(int v) {
   int i;
   visited[v] = 1;
   printf("%d ", v);

   for (i = 0; i < n; i++) {
      if (graph[v][i] == 1 && !visited[i]) {
         dfs(i);
      }
   }
}

int main() {
   int i, j, start;

   printf("Enter number of vertices: ");
   scanf("%d", &n);

   printf("Enter adjacency matrix:\n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         scanf("%d", &graph[i][j]);
      }
      visited[i] = 0;
   }

   printf("Enter starting vertex: ");
   scanf("%d", &start);
```

```
    printf("DFS Traversal: ");
    dfs(start);

    return 0;
}
```

Output

```
Enter number of vertices: 4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 0
0 1 0 0
Enter starting vertex: 0
DFS Traversal: 0 1 3 2
```

Key Notes

- DFS explores one path deeply before backtracking.
- Uses recursion for traversal.
- Adjacency matrix requires more space.
- Useful for cycle detection and path finding.

5.3 DFS (Adjacency List)

Aim

To traverse a graph using Depth First Search (DFS) with adjacency list representation.

Algorithm

1. Read the number of vertices and edges.
2. Create an adjacency list for the graph.
3. Initialize a visited array.
4. Start DFS from a given vertex.
5. Mark the vertex as visited and display it.
6. Recursively visit all adjacent unvisited vertices.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int vertex;
    struct node *next;
};

struct node *adj[10];
int visited[10];
int n;

void dfs(int v) {
    struct node *temp;
    visited[v] = 1;
```

```
    printf("%d ", v);

    temp = adj[v];
    while (temp != NULL) {
        if (!visited[temp->vertex]) {
            dfs(temp->vertex);
        }
        temp = temp->next;
    }
}

int main() {
    int i, edges, src, dest, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        adj[i] = NULL;
        visited[i] = 0;
    }

    printf("Enter number of edges: ");
    scanf("%d", &edges);

    for (i = 0; i < edges; i++) {
        printf("Enter edge (src dest): ");
        scanf("%d %d", &src, &dest);

        struct node *newNode = (struct node *)malloc(sizeof(struct node));
        newNode->vertex = dest;
        newNode->next = adj[src];
        adj[src] = newNode;
    }

    printf("Enter starting vertex: ");
    scanf("%d", &start);

    printf("DFS Traversal: ");
    dfs(start);

    return 0;
}
```

Output

```
Enter number of vertices: 4
Enter number of edges: 4
Enter edge (src dest): 0 1
Enter edge (src dest): 0 2
Enter edge (src dest): 1 3
Enter edge (src dest): 2 3
Enter starting vertex: 0
DFS Traversal: 0 2 3 1
```

Key Notes

- Adjacency list is memory efficient.
- DFS explores depth-first using recursion.

• Suitable for sparse graphs.
• Reduces space compared to adjacency matrix.

6. Trees

6.1 Tree Traversals (Inorder, Preorder, Postorder)

Aim
To perform inorder, preorder, and postorder traversals of a binary tree.

Algorithm

Inorder Traversal

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.

Preorder Traversal

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.

Postorder Traversal

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root node.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *left;
   struct node *right;
};

struct node* createNode(int value) {
   struct node* newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = value;
   newNode->left = newNode->right = NULL;
   return newNode;
}

void inorder(struct node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
```

```
    }
}

void preorder(struct node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

void postorder(struct node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    struct node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Inorder Traversal: ");
    inorder(root);

    printf("\nPreorder Traversal: ");
    preorder(root);

    printf("\nPostorder Traversal: ");
    postorder(root);

    return 0;
}
```

Output

```
Inorder Traversal: 4 2 5 1 3
Preorder Traversal: 1 2 4 5 3
Postorder Traversal: 4 5 2 3 1
```

Key Notes

- Tree traversal means visiting all nodes exactly once.
- Inorder traversal gives sorted order in BST.
- Preorder is useful for copying trees.
- Postorder is used for deleting trees.

6.2 DFS on Trees

Aim

To perform Depth First Search (DFS) traversal on a tree.

Algorithm

1. Start from the root node.
2. Visit the current node and display its value.
3. Recursively traverse all child nodes from left to right.
4. Continue until all nodes are visited.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *left;
   struct node *right;
};

struct node* createNode(int value) {
   struct node* newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = value;
   newNode->left = newNode->right = NULL;
   return newNode;
}

void dfs(struct node* root) {
   if (root != NULL) {
      printf("%d ", root->data);
      dfs(root->left);
      dfs(root->right);
   }
}

int main() {
   struct node* root = createNode(10);
   root->left = createNode(20);
   root->right = createNode(30);
   root->left->left = createNode(40);
   root->left->right = createNode(50);

   printf("DFS Traversal of Tree: ");
   dfs(root);

   return 0;
}
```

Output

```
DFS Traversal of Tree: 10 20 40 50 30
```

Key Notes

• DFS on trees is similar to preorder traversal.
• Uses recursion to visit nodes depth-wise.
• Each node is visited exactly once.
• Time complexity is O(n).

6.3 Binary Search Tree (Insertion & Traversal)

Aim
To create a Binary Search Tree (BST) and perform inorder traversal.

Algorithm

Insertion

1. If the tree is empty, create a new node as root.
2. If the value is less than the current node, insert into the left subtree.
3. If the value is greater than the current node, insert into the right subtree.
4. Repeat until the correct position is found.

Traversal (Inorder)

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *left;
   struct node *right;
};

struct node* createNode(int value) {
   struct node* newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = value;
   newNode->left = newNode->right = NULL;
   return newNode;
}

struct node* insert(struct node* root, int value) {
   if (root == NULL)
      return createNode(value);

   if (value < root->data)
      root->left = insert(root->left, value);
   else if (value > root->data)
      root->right = insert(root->right, value);

   return root;
}

void inorder(struct node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
   }
}
```

```
int main() {
    struct node* root = NULL;
    int n, value, i;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    printf("Enter node values:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &value);
        root = insert(root, value);
    }

    printf("Inorder Traversal of BST: ");
    inorder(root);

    return 0;
}
```

Output

```
Enter number of nodes: 5
Enter node values:
50 30 70 20 40
Inorder Traversal of BST: 20 30 40 50 70
```

Key Notes

- In BST, left subtree contains smaller values and right subtree contains larger values.
- Inorder traversal of BST gives sorted order.
- Average time complexity for insertion is O(log n).
- BST improves searching efficiency.

6.4 Heap Tree

Aim

To construct a Max Heap and display its elements.

Algorithm

1. Read the number of elements and the array elements.
2. Insert elements into the heap one by one.
3. For each insertion, compare the element with its parent.
4. If the element is greater than its parent, swap them.
5. Repeat until the heap property is satisfied.
6. Display the heap.

Program

```
#include <stdio.h>

void heapify(int heap[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int temp;
```

```
    if (left < n && heap[left] > heap[largest])
        largest = left;

    if (right < n && heap[right] > heap[largest])
        largest = right;

    if (largest != i) {
        temp = heap[i];
        heap[i] = heap[largest];
        heap[largest] = temp;
        heapify(heap, n, largest);
    }
}

int main() {
    int heap[50], n, i;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &heap[i]);
    }

    for (i = n / 2 - 1; i >= 0; i--) {
        heapify(heap, n, i);
    }

    printf("Max Heap:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", heap[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 6
Enter elements:
3 5 9 6 8 20
Max Heap:
20 8 9 6 5 3
```

Key Notes

- Heap is a complete binary tree.
- Max heap ensures parent node is greater than children.
- Heapify maintains heap property.
- Used in priority queues and heap sort.

PART B — DATA STRUCTURES ASSIGNMENT

Q1. Frequency Counter (Highest Frequency Elements)

Question
Find all elements of an array that have the highest frequency and print them along with their count.

Aim

To identify the element(s) that occur the maximum number of times in an array.

Algorithm

1. Read the number of elements and array values.
2. For each element, count its frequency by comparing with all elements.
3. Track the maximum frequency.
4. Print all elements whose frequency equals the maximum frequency.

Program

```
#include <stdio.h>

int main() {
    int arr[50], n, i, j, count, maxCount = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n; i++) {
        count = 1;
        for (j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                count++;
            }
        }
        if (count > maxCount) {
            maxCount = count;
        }
    }

    printf("Highest frequency elements:\n");
    for (i = 0; i < n; i++) {
        count = 1;
        for (j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                count++;
            }
        }
        if (count == maxCount) {
            printf("%d occurs %d times\n", arr[i], maxCount);
        }
```

```
    }

    return 0;
}
```

Output

```
Enter number of elements: 7
Enter elements:
1 2 2 3 3 3 4
Highest frequency elements:
3 occurs 3 times
```

Key Notes

- Uses nested loops (no extra data structure).
- Maximum frequency is tracked first.
- Prints all elements matching that frequency.
- Time complexity: $O(n^2)$.

Q2. Array Compression (Remove Duplicates In-Place)

Question
Given an array of integers, remove all duplicate values **in-place** without using any auxiliary array.

Aim

To remove duplicate elements from an array without using extra memory.

Algorithm

1. Read the number of elements and array values.
2. Use two loops to compare each element with the remaining elements.
3. If a duplicate is found, shift elements to the left.
4. Reduce the array size after removing duplicates.
5. Print the compressed array.

Program

```
#include <stdio.h>

int main() {
    int arr[50], n, i, j, k;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
```

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

    printf("Array after removing duplicates:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 7
Enter elements:
1 2 2 3 3 4 1
Array after removing duplicates:
1 2 3 4
```

Key Notes

- No auxiliary array is used.
- Elements are removed by shifting.
- Array size is dynamically reduced.
- Time complexity: $O(n^2)$.

Q3. Missing & Repeated Number Finder

Question
An array contains numbers from 1 to n, but one number is missing and one number is repeated. Find both.

Aim

To identify the missing number and the repeated number in an array containing values from 1 to n.

Algorithm

1. Read the value of n and the array elements.
2. For each element, count how many times it appears.
3. The number that appears twice is the repeated number.
4. The number from 1 to n that does not appear is the missing number.

5. Print both values.

Program

```
#include <stdio.h>

int main() {
   int arr[50], n, i, j;
   int missing = -1, repeated = -1, count;

   printf("Enter n: ");
   scanf("%d", &n);

   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   for (i = 1; i <= n; i++) {
      count = 0;
      for (j = 0; j < n; j++) {
         if (arr[j] == i)
            count++;
      }
      if (count == 0)
         missing = i;
      else if (count > 1)
         repeated = i;
   }

   printf("Missing number: %d\n", missing);
   printf("Repeated number: %d\n", repeated);

   return 0;
}
```

Output

```
Enter n: 5
Enter 5 elements:
1 2 2 4 5
Missing number: 3
Repeated number: 2
```

Key Notes

- Uses counting logic without extra arrays.
- Works because numbers are in range 1 to n.
- Simple brute-force approach.
- Time complexity: $O(n^2)$.

Q4. Subarray Sum Range

Question
Given an array and a target sum **S**, print all continuous subarrays whose sum equals **S**.

Aim

To find and display all contiguous subarrays whose sum is equal to a given target value.

Algorithm

1. Read the number of elements, array values, and target sum S.
2. Fix a starting index i.
3. Initialize sum = 0.
4. From index i, keep adding elements to sum.
5. If sum equals S, print the subarray.
6. Repeat for all starting indices.

Program

```
#include <stdio.h>

int main() {
   int arr[50], n, i, j, sum, S;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   printf("Enter target sum: ");
   scanf("%d", &S);

   printf("Subarrays with sum %d:\n", S);

   for (i = 0; i < n; i++) {
      sum = 0;
      for (j = i; j < n; j++) {
         sum += arr[j];
         if (sum == S) {
            printf("{ ");
            for (int k = i; k <= j; k++) {
               printf("%d ", arr[k]);
            }
            printf("}\n");
         }
      }
   }

   return 0;
}
```

Output

```
Enter number of elements: 5
Enter elements:
1 2 3 4 5
Enter target sum: 5
```

Subarrays with sum 5:
{ 2 3 }
{ 5 }

Key Notes

• Only continuous (contiguous) subarrays are considered.
• Uses nested loops to explore all ranges.
• No extra data structures are used.
• Time complexity: $O(n^2)$.

Q5. Array Rotation by Reversing Method

Question
Implement left rotation of an array by **k** positions using only the **reverse()** operation.

Aim

To rotate an array to the left by `k` positions using the reversing technique.

Algorithm

1. Read the number of elements `n`, array values, and rotation count `k`.
2. Reverse the first `k` elements.
3. Reverse the remaining `n − k` elements.
4. Reverse the entire array.
5. Display the rotated array.

Program

```
#include <stdio.h>

void reverse(int arr[], int start, int end) {
   int temp;
   while (start < end) {
      temp = arr[start];
      arr[start] = arr[end];
      arr[end] = temp;
      start++;
      end--;
   }
}

int main() {
   int arr[50], n, k, i;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
```

```
    printf("Enter rotation count k: ");
    scanf("%d", &k);
    k = k % n;

    reverse(arr, 0, k - 1);
    reverse(arr, k, n - 1);
    reverse(arr, 0, n - 1);

    printf("Array after left rotation:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 5
Enter elements:
1 2 3 4 5
Enter rotation count k: 2
Array after left rotation:
3 4 5 1 2
```

Key Notes

- Uses only reversing operations.
- No extra array is required.
- Efficient rotation in O(n) time.
- Common interview technique.

Q6. Majority Element

Question
Determine the element that appears more than **n/2** times in an array (if any).

Aim

To find the majority element in an array, i.e., an element occurring more than half the number of times.

Algorithm

1. Read the number of elements and array values.
2. For each element, count its occurrences.
3. If any element's count is greater than $n/2$, declare it as the majority element.
4. If no such element exists, report that no majority element is present.

Program

```
#include <stdio.h>
```

```
int main() {
    int arr[50], n, i, j, count;
    int majority = -1;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (i = 0; i < n; i++) {
        count = 0;
        for (j = 0; j < n; j++) {
            if (arr[i] == arr[j])
                count++;
        }
        if (count > n / 2) {
            majority = arr[i];
            break;
        }
    }

    if (majority != -1)
        printf("Majority element: %d\n", majority);
    else
        printf("No majority element found\n");

    return 0;
}
```

Output

```
Enter number of elements: 7
Enter elements:
2 2 1 2 3 2 2
Majority element: 2
```

Key Notes

- Majority element must appear more than $n/2$ times.
- Simple counting-based approach.
- Does not use extra memory.
- Time complexity: $O(n^2)$.

Q7. Maximum Product of Any Three Numbers

Question
Find the maximum product of any three numbers in an array. Handle both negative and positive values.

Aim

To determine the maximum possible product of any three elements in an array.

Algorithm

1. Read the number of elements and array values.
2. Find the three largest elements in the array.
3. Find the two smallest (most negative) elements in the array.
4. Compute:
    - Product1 = largest1 × largest2 × largest3
    - Product2 = smallest1 × smallest2 × largest1
5. Print the maximum of Product1 and Product2.

Program

```
#include <stdio.h>
#include <limits.h>

int main() {
   int arr[50], n, i;
   int max1 = INT_MIN, max2 = INT_MIN, max3 = INT_MIN;
   int min1 = INT_MAX, min2 = INT_MAX;
   int product1, product2;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);

      if (arr[i] > max1) {
         max3 = max2;
         max2 = max1;
         max1 = arr[i];
      } else if (arr[i] > max2) {
         max3 = max2;
         max2 = arr[i];
      } else if (arr[i] > max3) {
         max3 = arr[i];
      }

      if (arr[i] < min1) {
         min2 = min1;
         min1 = arr[i];
      } else if (arr[i] < min2) {
         min2 = arr[i];
      }
   }

   product1 = max1 * max2 * max3;
   product2 = min1 * min2 * max1;

   if (product1 > product2)
      printf("Maximum product is: %d\n", product1);
   else
      printf("Maximum product is: %d\n", product2);
```

```
    return 0;
}
```

Output

```
Enter number of elements: 5
Enter elements:
-10 -3 5 6 -2
Maximum product is: 180
```

Key Notes

• Two negative numbers can produce a large positive product.
• Avoids sorting the array.
• Works in a single pass.
• Time complexity: O(n).

Q8. Second Largest Unique Number

Question
Find the **second largest distinct element** in an array **without sorting** the array.

Aim

To determine the second largest **unique** element using a single traversal.

Algorithm

1. Read the number of elements and array values.
2. Maintain two variables: largest and secondLargest.
3. Traverse the array:
    - If current element is greater than largest, update both.
    - If current element is less than largest but greater than secondLargest, update secondLargest.
4. Display the second largest unique element.

Program

```
#include <stdio.h>
#include <limits.h>

int main() {
    int arr[50], n, i;
    int largest = INT_MIN, secondLargest = INT_MIN;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
```

```
        if (arr[i] > largest) {
            secondLargest = largest;
            largest = arr[i];
        } else if (arr[i] > secondLargest && arr[i] != largest) {
            secondLargest = arr[i];
        }
    }

    if (secondLargest == INT_MIN)
        printf("Second largest unique element not found\n");
    else
        printf("Second largest unique element: %d\n", secondLargest);

    return 0;
}
```

Output

```
Enter number of elements: 6
Enter elements:
10 20 20 15 8 5
Second largest unique element: 15
```

Key Notes

- Sorting is not used.
- Handles duplicate values correctly.
- Uses only constant extra space.
- Time complexity: O(n).

Q9. Wave Rearrangement

Question
Rearrange an array into a wave form such that
$a_1 \geq a_2 \leq a_3 \geq a_4 \ldots$

Aim

To rearrange array elements into a wave-like pattern.

Algorithm

1. Read the number of elements and array values.
2. Sort the array in ascending order.
3. Swap adjacent elements starting from index 0.
4. Display the wave-arranged array.

Program

```
#include <stdio.h>

int main() {
    int arr[50], n, i, j, temp;
```

```
    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    /* Simple bubble sort */
    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;
            }
        }
    }

    /* Swap adjacent elements */
    for (i = 0; i < n - 1; i += 2) {
        temp = arr[i];
        arr[i] = arr[i + 1];
        arr[i + 1] = temp;
    }

    printf("Wave form array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 6
Enter elements:
1 2 3 4 5 6
Wave form array:
2 1 4 3 6 5
```

Key Notes

- Adjacent elements are swapped after sorting.
- Final arrangement satisfies wave condition.
- Simple and easy to understand.
- Time complexity: $O(n^2)$ due to sorting.

Q10. Undo Operation Simulator

Question
Implement undo functionality using a stack. Push operations onto the stack and pop to revert the last operation.

Aim

To simulate undo operations using stack data structure.

Algorithm

1. Initialize an empty stack.
2. Push every operation performed onto the stack.
3. When undo is requested, pop the top element.
4. Display the current state of operations.

Program

```
#include <stdio.h>
#define MAX 10

int stack[MAX];
int top = -1;

void push(int value) {
   if (top == MAX - 1) {
      printf("Stack Overflow\n");
      return;
   }
   stack[++top] = value;
}

void undo() {
   if (top == -1) {
      printf("Nothing to undo\n");
      return;
   }
   printf("Undo operation: %d\n", stack[top--]);
}

void display() {
   int i;
   if (top == -1) {
      printf("No operations\n");
      return;
   }
   for (i = top; i >= 0; i--) {
      printf("%d ", stack[i]);
   }
   printf("\n");
}

int main() {
   push(10);
   push(20);
   display();
   undo();
   display();
   return 0;
}
```

Output

10 20
Undo operation: 20
10

Key Notes

• Stack follows LIFO principle.
• Last operation is undone first.
• Used in editors and applications.
• Simple stack implementation.

Q11. Minimum Stack (Get Minimum in O(1))

Question
Implement a stack that supports **push**, **pop**, and **getMin()** operations in constant time.

Aim

To retrieve the minimum element from a stack in **O(1)** time.

Algorithm

1. Use two stacks:
    - Main stack to store elements.
    - Min stack to store minimum values.
2. On push:
    - Push element to main stack.
    - If min stack is empty or element ≤ top of min stack, push it to min stack.
3. On pop:
    - Pop from main stack.
    - If popped element equals top of min stack, pop from min stack.
4. getMin(): return top of min stack.

Program

```
#include <stdio.h>
#define MAX 20

int stack[MAX], minStack[MAX];
int top = -1, minTop = -1;

void push(int x) {
   stack[++top] = x;
   if (minTop == -1 || x <= minStack[minTop]) {
      minStack[++minTop] = x;
   }
}

void pop() {
   if (top == -1) {
```

```
        printf("Stack is empty\n");
        return;
    }
    if (stack[top] == minStack[minTop]) {
        minTop--;
    }
    printf("Popped: %d\n", stack[top--]);
}

void getMin() {
    if (minTop == -1)
        printf("Stack is empty\n");
    else
        printf("Minimum element: %d\n", minStack[minTop]);
}

int main() {
    push(5);
    push(2);
    push(8);
    getMin();
    pop();
    getMin();
    return 0;
}
```

Output

```
Minimum element: 2
Popped: 8
Minimum element: 2
```

Key Notes

• Uses an auxiliary stack to track minimum values.
• All operations work in constant time.
• Extra space is used for efficiency.
• Common interview problem.

Q12. Two Stacks in a Single Array

Question
Implement two independent stacks using a single shared array.

Aim

To efficiently utilize memory by implementing two stacks in one array.

Algorithm

1. Initialize two stack pointers:
    - top1 = -1 (left stack)
    - top2 = size (right stack)
2. Stack 1 grows from left to right.

3. Stack 2 grows from right to left.
4. Overflow occurs when top1 + 1 == top2.
5. Perform push and pop operations independently.

Program

```
#include <stdio.h>
#define MAX 10

int arr[MAX];
int top1 = -1;
int top2 = MAX;

void push1(int x) {
   if (top1 + 1 == top2) {
      printf("Stack Overflow\n");
      return;
   }
   arr[++top1] = x;
}

void push2(int x) {
   if (top1 + 1 == top2) {
      printf("Stack Overflow\n");
      return;
   }
   arr[--top2] = x;
}

void pop1() {
   if (top1 == -1) {
      printf("Stack1 Underflow\n");
      return;
   }
   printf("Popped from Stack1: %d\n", arr[top1--]);
}

void pop2() {
   if (top2 == MAX) {
      printf("Stack2 Underflow\n");
      return;
   }
   printf("Popped from Stack2: %d\n", arr[top2++]);
}

int main() {
   push1(10);
   push1(20);
   push2(30);
   push2(40);

   pop1();
   pop2();

   return 0;
}
```

Output

Popped from Stack1: 20
Popped from Stack2: 40

Key Notes

• Both stacks share unused space efficiently.
• Prevents memory wastage.
• Overflow happens only when array is full.
• Common memory optimization technique.

Q13. Stack Reversal Using Only Push/Pop

Question
Reverse a stack using **only stack operations** and **one auxiliary stack**.

Aim

To reverse the order of elements in a stack using push and pop operations only.

Algorithm

1. Push all elements into the main stack.
2. Pop elements one by one from the main stack.
3. Push each popped element into an auxiliary stack.
4. The auxiliary stack now contains the reversed stack.
5. Display the reversed stack.

Program

```
#include <stdio.h>
#define MAX 10

int stack[MAX], aux[MAX];
int top = -1, auxTop = -1;

void push(int x) {
   stack[++top] = x;
}

int pop() {
   return stack[top--];
}

int main() {
   int n, i, val;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter stack elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &val);
      push(val);
```

```
    }

    while (top != -1) {
        aux[++auxTop] = pop();
    }

    printf("Reversed stack:\n");
    for (i = auxTop; i >= 0; i--) {
        printf("%d ", aux[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 4
Enter stack elements:
1 2 3 4
Reversed stack:
4 3 2 1
```

Key Notes

- Uses only push and pop operations.
- One auxiliary stack is allowed.
- Demonstrates LIFO behavior clearly.
- Time complexity: O(n).

Q14. Next Greater Element

Question
For each element in an array, find the **next greater element** to its right using a stack.

Aim

To efficiently determine the next greater element for every array element.

Algorithm

1. Read the number of elements and array values.
2. Initialize an empty stack to store indices.
3. Traverse the array from left to right.
4. While stack is not empty and current element is greater than the element at stack top index:
    - Pop index and print its next greater element.
5. Push current index onto the stack.
6. For remaining elements in stack, print -1 as no greater element exists.

Program

```
#include <stdio.h>
```

```
#define MAX 50

int stack[MAX];
int top = -1;

void push(int i) {
   stack[++top] = i;
}

int pop() {
   return stack[top--];
}

int isEmpty() {
   return top == -1;
}

int main() {
   int arr[50], n, i, index;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   printf("Next Greater Elements:\n");
   for (i = 0; i < n; i++) {
      while (!isEmpty() && arr[i] > arr[stack[top]]) {
         index = pop();
         printf("%d -> %d\n", arr[index], arr[i]);
      }
      push(i);
   }

   while (!isEmpty()) {
      index = pop();
      printf("%d -> -1\n", arr[index]);
   }

   return 0;
}
```

Output

```
Enter number of elements: 4
Enter elements:
4 5 2 25
Next Greater Elements:
4 -> 5
5 -> 25
2 -> 25
25 -> -1
```

Key Notes

• Each element is pushed and popped at most once.
• Uses stack to reduce time complexity.
• Much faster than nested loops.
• Time complexity: O(n).

Q15. Evaluate Prefix Expression

Question
Evaluate a **prefix expression** using a stack (without converting it to infix or postfix).

Aim

To evaluate a prefix expression using stack operations.

Algorithm

1. Read the prefix expression.
2. Initialize an empty stack.
3. Scan the expression from **right to left**.
4. If the symbol is an operand, push it onto the stack.
5. If the symbol is an operator:
    - Pop two operands from the stack.
    - Apply the operator.
    - Push the result back onto the stack.
6. The final value on the stack is the result.

Program

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX 50

int stack[MAX];
int top = -1;

void push(int x) {
   stack[++top] = x;
}

int pop() {
   return stack[top--];
}

int main() {
   char prefix[MAX];
   int i, op1, op2, result;

   printf("Enter prefix expression: ");
   scanf("%s", prefix);

   for (i = strlen(prefix) - 1; i >= 0; i--) {
      if (isdigit(prefix[i])) {
```

```
            push(prefix[i] - '0');
        } else {
            op1 = pop();
            op2 = pop();

            switch (prefix[i]) {
                case '+': result = op1 + op2; break;
                case '-': result = op1 - op2; break;
                case '*': result = op1 * op2; break;
                case '/': result = op1 / op2; break;
            }
            push(result);
        }
    }

    printf("Result: %d\n", pop());
    return 0;
}
```

Output

```
Enter prefix expression: +9*26
Result: 21
```

Key Notes

• Prefix is evaluated from right to left.
• Stack stores operands temporarily.
• No conversion is required.
• Time complexity: O(n).

Q16. Factorial of a Number (Simulate Recursive Factorial Using Stack)

Question
Find the factorial of a number by simulating recursion using a stack.

Aim

To compute factorial without using recursion, by simulating recursive calls with a stack.

Algorithm

1. Read the number `n`.
2. Push values from `n` down to `1` onto the stack.
3. Initialize `result = 1`.
4. Pop each element from the stack and multiply it with `result`.
5. Display the factorial value.

Program

```
#include <stdio.h>
```

```
#define MAX 20

int stack[MAX];
int top = -1;

void push(int x) {
   stack[++top] = x;
}

int pop() {
   return stack[top--];
}

int main() {
   int n, i, result = 1;

   printf("Enter a number: ");
   scanf("%d", &n);

   for (i = n; i >= 1; i--) {
      push(i);
   }

   while (top != -1) {
      result *= pop();
   }

   printf("Factorial of %d is %d\n", n, result);

   return 0;
}
```

Output

```
Enter a number: 5
Factorial of 5 is 120
```

Key Notes

• Stack replaces recursive function calls.
• Each stack push represents a recursive call.
• Popping simulates function return.
• Helps understand recursion internally.

Q17. Interleave First Half with Second Half of a Queue

Question
For a queue of even length, interleave its first half with its second half.
Example:
1 2 3 4 5 6 → 1 4 2 5 3 6

Aim

To interleave the first half of a queue with the second half using queue operations.

Algorithm

1. Read the number of elements (must be even).
2. Insert all elements into the queue.
3. Move the first half elements into an auxiliary queue.
4. Alternately enqueue one element from the auxiliary queue and one from the remaining original queue.
5. Display the interleaved queue.

Program

```
#include <stdio.h>
#define MAX 50

int queue[MAX], aux[MAX];
int front = 0, rear = -1;
int af = 0, ar = -1;

void enqueue(int q[], int *r, int value) {
   q[++(*r)] = value;
}

int dequeue(int q[], int *f) {
   return q[(*f)++];
}

int main() {
   int n, i, value;

   printf("Enter even number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &value);
      enqueue(queue, &rear, value);
   }

   for (i = 0; i < n / 2; i++) {
      enqueue(aux, &ar, dequeue(queue, &front));
   }

   printf("Interleaved Queue:\n");
   while (af <= ar) {
      printf("%d ", dequeue(aux, &af));
      printf("%d ", dequeue(queue, &front));
   }

   return 0;
}
```

Output

```
Enter even number of elements: 6
Enter elements:
1 2 3 4 5 6
Interleaved Queue:
```

1 4 2 5 3 6

Key Notes

• Uses one auxiliary queue.
• Maintains relative order inside each half.
• Works only for even-sized queues.
• Time complexity: O(n).

Q18. Queue Using Two Stacks

Question
Implement a queue using **two stacks** and perform enqueue and dequeue operations.

Aim

To simulate queue operations using stack data structures.

Algorithm

Enqueue

1. Push the element onto Stack 1.

Dequeue

1. If Stack 2 is empty:
    - Pop all elements from Stack 1 and push them into Stack 2.
2. Pop the top element from Stack 2 (this is the dequeued element).

Program

```
#include <stdio.h>
#define MAX 50

int s1[MAX], s2[MAX];
int top1 = -1, top2 = -1;

void push1(int x) {
   s1[++top1] = x;
}

void push2(int x) {
   s2[++top2] = x;
}

int pop1() {
   return s1[top1--];
}

int pop2() {
   return s2[top2--];
}
```

```
void enqueue(int x) {
   push1(x);
}

void dequeue() {
   int i;
   if (top1 == -1 && top2 == -1) {
      printf("Queue is empty\n");
      return;
   }
   if (top2 == -1) {
      while (top1 != -1) {
         push2(pop1());
      }
   }
   printf("Dequeued element: %d\n", pop2());
}

int main() {
   enqueue(10);
   enqueue(20);
   enqueue(30);
   dequeue();
   dequeue();
   return 0;
}
```

Output

```
Dequeued element: 10
Dequeued element: 20
```

Key Notes

• Stack1 is used for enqueue operation.
• Stack2 reverses order to maintain FIFO behavior.
• Dequeue is amortized O(1).
• Demonstrates how queues can be built from stacks.

Q19. Reverse First K Elements of a Queue

Question
Reverse the first **K** elements of a queue while keeping the remaining elements in the same order.

Example
Queue: {1, 2, 3, 4, 5}, K = 3
Output: {3, 2, 1, 4, 5}

Aim

To reverse only the first K elements of a queue using a stack.

Algorithm

1. Read queue elements and value K.
2. Dequeue the first K elements and push them into a stack.
3. Pop elements from the stack and enqueue them back into the queue.
4. Move the remaining (n − K) elements from front to rear to preserve order.
5. Display the modified queue.

Program

```
#include <stdio.h>
#define MAX 50

int queue[MAX], stack[MAX];
int front = 0, rear = -1, top = -1;

void enqueue(int x) {
   queue[++rear] = x;
}

int dequeue() {
   return queue[front++];
}

void push(int x) {
   stack[++top] = x;
}

int pop() {
   return stack[top--];
}

int main() {
   int n, k, i, value;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter queue elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &value);
      enqueue(value);
   }

   printf("Enter value of K: ");
   scanf("%d", &k);

   for (i = 0; i < k; i++) {
      push(dequeue());
   }

   while (top != -1) {
      enqueue(pop());
   }

   for (i = 0; i < n - k; i++) {
      enqueue(dequeue());
   }
```

```
    printf("Queue after reversing first %d elements:\n", k);
    for (i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 5
Enter queue elements:
1 2 3 4 5
Enter value of K: 3
Queue after reversing first 3 elements:
3 2 1 4 5
```

Key Notes

• Stack is used to reverse order.
• Remaining queue elements retain original order.
• Efficient and commonly asked problem.
• Time complexity: O(n).

Q20. Check If a Queue Is Palindrome

Question
Check whether a queue is a palindrome using only queue operations and **at most one stack**.

Aim

To determine if the elements of a queue read the same forwards and backwards.

Algorithm

1. Read the number of elements and insert them into the queue.
2. Traverse the queue and push each element into a stack.
3. Again traverse the queue from front.
4. For each element dequeued, compare it with the top of the stack.
5. If all elements match, the queue is a palindrome.
6. Otherwise, it is not a palindrome.

Program

```
#include <stdio.h>
#define MAX 50

int queue[MAX], stack[MAX];
int front = 0, rear = -1, top = -1;

void enqueue(int x) {
```

```
    queue[++rear] = x;
}

int dequeue() {
    return queue[front++];
}

void push(int x) {
    stack[++top] = x;
}

int pop() {
    return stack[top--];
}

int main() {
    int n, i, value;
    int isPalindrome = 1;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter queue elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &value);
        enqueue(value);
        push(value);
    }

    for (i = 0; i < n; i++) {
        if (dequeue() != pop()) {
            isPalindrome = 0;
            break;
        }
    }

    if (isPalindrome)
        printf("Queue is a Palindrome\n");
    else
        printf("Queue is NOT a Palindrome\n");

    return 0;
}
```

Output

```
Enter number of elements: 5
Enter queue elements:
1 2 3 2 1
Queue is a Palindrome
```

Key Notes

- Stack reverses the order of queue elements.
- Comparison checks symmetry.
- Uses only one auxiliary stack.
- Time complexity: O(n).

Q21. First Non-Repeating Character in a Stream

Question
Given a stream of characters, print the **first non-repeating character** at each point using a queue.

Aim

To identify the first character that has appeared only once so far in a character stream.

Algorithm

1. Initialize an empty queue to store characters.
2. Maintain a frequency array of size 256 (for all ASCII characters).
3. Read characters one by one from the stream.
4. For each character:
    - Increment its frequency.
    - Enqueue it into the queue.
5. While the front of the queue has frequency > 1, dequeue it.
6. The front of the queue (if any) is the first non-repeating character.
7. If the queue becomes empty, print -1.

Program

```
#include <stdio.h>
#define MAX 50

char queue[MAX];
int freq[256] = {0};
int front = 0, rear = -1;

void enqueue(char ch) {
   queue[++rear] = ch;
}

char dequeue() {
   return queue[front++];
}

int main() {
   char stream[MAX];
   int i;

   printf("Enter character stream: ");
   scanf("%s", stream);

   printf("First non-repeating characters:\n");

   for (i = 0; stream[i] != '\0'; i++) {
      char ch = stream[i];
      freq[ch]++;
      enqueue(ch);

      while (front <= rear && freq[queue[front]] > 1) {
```

```
            dequeue();
        }

        if (front <= rear)
            printf("%c ", queue[front]);
        else
            printf("-1 ");
    }

    return 0;
}
```

Output

```
Enter character stream: aabc
First non-repeating characters:
a -1 b b
```

Key Notes

• Queue maintains order of characters.
• Frequency array tracks repetitions.
• Efficient for real-time streams.
• Time complexity: O(n).

Q22. Queue Rotation by K Places

Question
Rotate a queue by **k places** (circular behavior) **without using a circular queue implementation**.

Aim

To rotate the elements of a queue to the left by `k` positions using basic queue operations.

Algorithm

1. Read the number of elements and queue values.
2. Read rotation count `k`.
3. Repeat `k` times:
    - Dequeue the front element.
    - Enqueue it at the rear.
4. Display the rotated queue.

Program

```
#include <stdio.h>
#define MAX 50

int queue[MAX];
int front = 0, rear = -1;
```

```
void enqueue(int x) {
   queue[++rear] = x;
}

int dequeue() {
   return queue[front++];
}

int main() {
   int n, k, i, value;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter queue elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &value);
      enqueue(value);
   }

   printf("Enter rotation count k: ");
   scanf("%d", &k);

   k = k % n;

   for (i = 0; i < k; i++) {
      enqueue(dequeue());
   }

   printf("Queue after rotation:\n");
   for (i = front; i <= rear; i++) {
      printf("%d ", queue[i]);
   }

   return 0;
}
```

Output

```
Enter number of elements: 5
Enter queue elements:
1 2 3 4 5
Enter rotation count k: 2
Queue after rotation:
3 4 5 1 2
```

Key Notes

- Rotation is done using dequeue and enqueue only.
- No circular queue logic is used.
- Simple and intuitive approach.
- Time complexity: O(n).

Q23. Inter-Queue Swap

Question
Swap the contents of two queues using **only queue operations**.

Aim

To exchange all elements of two queues without using arrays or stacks.

Algorithm

1. Read the elements of Queue 1 and Queue 2.
2. Create a temporary queue.
3. Dequeue all elements from Queue 1 and enqueue them into the temporary queue.
4. Dequeue all elements from Queue 2 and enqueue them into Queue 1.
5. Dequeue all elements from the temporary queue and enqueue them into Queue 2.
6. Display both queues after swapping.

Program

```
#include <stdio.h>
#define MAX 50

int q1[MAX], q2[MAX], temp[MAX];
int f1 = 0, r1 = -1;
int f2 = 0, r2 = -1;
int ft = 0, rt = -1;

void enqueue(int q[], int *r, int value) {
   q[++(*r)] = value;
}

int dequeue(int q[], int *f) {
   return q[(*f)++];
}

int main() {
   int n1, n2, i, value;

   printf("Enter number of elements in Queue 1: ");
   scanf("%d", &n1);
   printf("Enter elements of Queue 1:\n");
   for (i = 0; i < n1; i++) {
      scanf("%d", &value);
      enqueue(q1, &r1, value);
   }

   printf("Enter number of elements in Queue 2: ");
   scanf("%d", &n2);
   printf("Enter elements of Queue 2:\n");
   for (i = 0; i < n2; i++) {
      scanf("%d", &value);
      enqueue(q2, &r2, value);
   }

   while (f1 <= r1)
      enqueue(temp, &rt, dequeue(q1, &f1));
```

```
    while (f2 <= r2)
        enqueue(q1, &r1, dequeue(q2, &f2));

    while (ft <= rt)
        enqueue(q2, &r2, dequeue(temp, &ft));

    printf("Queue 1 after swap:\n");
    for (i = 0; i <= r1; i++)
        printf("%d ", q1[i]);

    printf("\nQueue 2 after swap:\n");
    for (i = 0; i <= r2; i++)
        printf("%d ", q2[i]);

    return 0;
}
```

Output

```
Enter number of elements in Queue 1: 3
Enter elements of Queue 1:
1 2 3
Enter number of elements in Queue 2: 2
Enter elements of Queue 2:
9 8
Queue 1 after swap:
9 8
Queue 2 after swap:
1 2 3
```

Key Notes

• Uses only queue enqueue and dequeue operations.
• Temporary queue helps in swapping.
• No stack or array logic is involved conceptually.
• Time complexity: O(n + m).

Q24. Queue Sorting Using Only One Additional Queue

Question
Sort the elements of a queue using **only one additional queue**.
(No array or stack is allowed.)

Aim

To sort a queue in ascending order using only queue operations and one extra queue.

Algorithm

1. Read the number of elements and enqueue them into the main queue.
2. Repeat until the main queue becomes empty:
    - Dequeue an element and assume it is the minimum.
    - Compare it with remaining elements by dequeuing each one:

        - If a smaller element is found, enqueue the previous minimum back and update minimum.
        - Otherwise, enqueue the element into the auxiliary queue.
    - Enqueue the minimum element into a sorted queue (reuse main queue later).
3. Move elements back appropriately until the queue is fully sorted.
4. Display the sorted queue.

Program

```
#include <stdio.h>
#define MAX 50

int q[MAX], aux[MAX];
int front = 0, rear = -1;
int af = 0, ar = -1;

void enqueue(int q[], int *r, int x) {
   q[++(*r)] = x;
}

int dequeue(int q[], int *f) {
   return q[(*f)++];
}

int main() {
   int n, i, value, min, size;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter queue elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &value);
      enqueue(q, &rear, value);
   }

   size = n;

   while (size > 0) {
      min = dequeue(q, &front);
      for (i = 1; i < size; i++) {
         value = dequeue(q, &front);
         if (value < min) {
            enqueue(aux, &ar, min);
            min = value;
         } else {
            enqueue(aux, &ar, value);
         }
      }
      enqueue(q, &rear, min);

      while (af <= ar)
         enqueue(q, &rear, dequeue(aux, &af));

      size--;
   }

   printf("Sorted Queue:\n");
```

```
    for (i = front; i <= rear; i++) {
        printf("%d ", q[i]);
    }

    return 0;
}
```

Output

```
Enter number of elements: 5
Enter queue elements:
4 1 3 2 5
Sorted Queue:
1 2 3 4 5
```

Key Notes

• Only one auxiliary queue is used.
• Sorting is done by repeatedly extracting minimum elements.
• No array or stack logic is used conceptually.
• Time complexity: $O(n^2)$.

Q25. Intersection Point of Two Linked Lists

Question
Given two linked lists that merge at a node, find the **intersection point**.

Aim

To find the node at which two singly linked lists intersect.

Algorithm

1. Count the number of nodes in both linked lists.
2. Find the difference `d` between the lengths.
3. Move the pointer of the longer list ahead by `d` nodes.
4. Traverse both lists together.
5. The first common node is the intersection point.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};

int getCount(struct node *head) {
    int count = 0;
    while (head) {
        count++;
```

```
        head = head->next;
    }
    return count;
}

struct node* getIntersection(struct node *head1, struct node *head2) {
    int c1 = getCount(head1);
    int c2 = getCount(head2);
    int d, i;

    if (c1 > c2) {
        d = c1 - c2;
        for (i = 0; i < d; i++)
            head1 = head1->next;
    } else {
        d = c2 - c1;
        for (i = 0; i < d; i++)
            head2 = head2->next;
    }

    while (head1 && head2) {
        if (head1 == head2)
            return head1;
        head1 = head1->next;
        head2 = head2->next;
    }
    return NULL;
}

int main() {
    /* Demo structure */
    struct node *common = malloc(sizeof(struct node));
    common->data = 30;
    common->next = NULL;

    struct node *head1 = malloc(sizeof(struct node));
    head1->data = 10;
    head1->next = common;

    struct node *head2 = malloc(sizeof(struct node));
    head2->data = 20;
    head2->next = common;

    struct node *res = getIntersection(head1, head2);
    if (res)
        printf("Intersection at node with data %d\n", res->data);
    else
        printf("No intersection\n");

    return 0;
}
```

Output

Intersection at node with data 30

Key Notes

• Uses length difference technique.
• No extra memory is used.
• Works in O(n) time.
• Very common interview problem.

Q26. Sort Elements Represented in Linked List

(Digits stored in reverse or forward order)

Question
Given a linked list where digits of a number are stored (either in forward or reverse order), sort the elements in the linked list.

Aim

To sort the elements of a linked list in ascending order

Algorithm

1. Traverse the linked list using two pointers.
2. Compare data of current node with the remaining nodes.
3. Swap the data if the current node's data is greater.
4. Repeat until the list is sorted (bubble sort on linked list).
5. Display the sorted linked list.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

void sortList(struct node *head) {
   struct node *i, *j;
   int temp;

   for (i = head; i != NULL; i = i->next) {
      for (j = i->next; j != NULL; j = j->next) {
         if (i->data > j->data) {
            temp = i->data;
            i->data = j->data;
            j->data = temp;
         }
      }
   }
}

void display(struct node *head) {
   while (head != NULL) {
      printf("%d -> ", head->data);
      head = head->next;
```

```
    }
    printf("NULL\n");
}

int main() {
    struct node *head = NULL, *temp, *newNode;
    int n, i, value;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        newNode = (struct node *)malloc(sizeof(struct node));
        printf("Enter value: ");
        scanf("%d", &value);
        newNode->data = value;
        newNode->next = NULL;

        if (head == NULL)
            head = temp = newNode;
        else {
            temp->next = newNode;
            temp = newNode;
        }
    }

    printf("Original List:\n");
    display(head);

    sortList(head);

    printf("Sorted List:\n");
    display(head);

    return 0;
}
```

Output

```
Enter number of nodes: 5
Enter value:
3 1 5 2 4
Original List:
3 -> 1 -> 5 -> 2 -> 4 -> NULL
Sorted List:
1 -> 2 -> 3 -> 4 -> 5 -> NULL
```

Key Note:

- Sorting is done by swapping node data.
- Works for both forward and reverse digit storage.
- No extra list is created.
- Time complexity: $O(n^2)$.

Q27. Palindrome Check (Without Extra Array)

Question
Check whether a linked list is a palindrome **without using any extra array**.

Aim

To verify if a linked list reads the same forwards and backwards using pointer manipulation.

Algorithm

1. Use **slow** and **fast** pointers to find the middle of the linked list.
2. Reverse the second half of the linked list.
3. Compare the first half and the reversed second half node by node.
4. If all corresponding nodes match, the list is a palindrome.
5. Otherwise, it is not a palindrome.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

struct node* reverse(struct node* head) {
   struct node *prev = NULL, *curr = head, *next;
   while (curr != NULL) {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
   return prev;
}

int isPalindrome(struct node* head) {
   struct node *slow = head, *fast = head;
   struct node *secondHalf, *temp = head;

   if (head == NULL)
      return 1;

   while (fast->next != NULL && fast->next->next != NULL) {
      slow = slow->next;
      fast = fast->next->next;
   }

   secondHalf = reverse(slow->next);
   slow->next = secondHalf;

   while (secondHalf != NULL) {
      if (temp->data != secondHalf->data)
         return 0;
      temp = temp->next;
      secondHalf = secondHalf->next;
```

```
    }
    return 1;
}

int main() {
    struct node *head = NULL, *temp, *newNode;
    int n, i, value;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        newNode = (struct node *)malloc(sizeof(struct node));
        scanf("%d", &value);
        newNode->data = value;
        newNode->next = NULL;

        if (head == NULL)
            head = temp = newNode;
        else {
            temp->next = newNode;
            temp = newNode;
        }
    }

    if (isPalindrome(head))
        printf("Linked list is a Palindrome\n");
    else
        printf("Linked list is NOT a Palindrome\n");

    return 0;
}
```

Output

```
Enter number of nodes: 5
1 2 3 2 1
Linked list is a Palindrome
```

Key Notes

- No extra array or stack is used.
- Second half of the list is reversed in-place.
- Uses slow–fast pointer technique.
- Time complexity: O(n), Space complexity: O(1).

Q28. Remove Duplicates from an Unsorted Linked List

(Without using extra buffer)

Question
Remove duplicate elements from an **unsorted linked list** without using any extra data structure.

Aim

To eliminate duplicate nodes from an unsorted linked list using pointer comparison only.

Algorithm

1. Start with the first node as current.
2. Use another pointer runner to check all subsequent nodes.
3. If runner->next->data is equal to current->data, delete the duplicate node.
4. Otherwise, move runner forward.
5. Repeat for all nodes until the list is processed.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

void removeDuplicates(struct node *head) {
   struct node *current = head, *runner, *temp;

   while (current != NULL) {
      runner = current;
      while (runner->next != NULL) {
         if (runner->next->data == current->data) {
            temp = runner->next;
            runner->next = temp->next;
            free(temp);
         } else {
            runner = runner->next;
         }
      }
      current = current->next;
   }
}

void display(struct node *head) {
   while (head != NULL) {
      printf("%d -> ", head->data);
      head = head->next;
   }
   printf("NULL\n");
}

int main() {
   struct node *head = NULL, *temp, *newNode;
   int n, i, value;

   printf("Enter number of nodes: ");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      newNode = (struct node *)malloc(sizeof(struct node));
      scanf("%d", &value);
      newNode->data = value;
```

```
        newNode->next = NULL;

        if (head == NULL)
            head = temp = newNode;
        else {
            temp->next = newNode;
            temp = newNode;
        }
    }

    printf("Original List:\n");
    display(head);

    removeDuplicates(head);

    printf("List after removing duplicates:\n");
    display(head);

    return 0;
}
```

Output

```
Enter number of nodes: 6
1 3 2 3 1 4
Original List:
1 -> 3 -> 2 -> 3 -> 1 -> 4 -> NULL
List after removing duplicates:
1 -> 3 -> -> 2 -> 4 -> NULL
```

Key Notes

- No extra memory or buffer is used.
- Uses two-pointer technique.
- Suitable for unsorted linked lists.
- Time complexity: $O(n^2)$.

Q29. Merge Two Sorted Linked Lists

(Without creating new nodes)

Question
Merge two sorted linked lists into one sorted linked list **without creating new nodes**.

Aim

To merge two sorted linked lists by rearranging existing node links.

Algorithm

1. Compare the first nodes of both lists.
2. Select the smaller node as the head of the merged list.
3. Move the pointer of the selected list forward.

4. Continue comparing and linking nodes until one list becomes empty.
5. Attach the remaining nodes of the non-empty list.
6. Display the merged list.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

struct node* mergeLists(struct node* l1, struct node* l2) {
   struct node *result = NULL;

   if (l1 == NULL)
      return l2;
   if (l2 == NULL)
      return l1;

   if (l1->data <= l2->data) {
      result = l1;
      result->next = mergeLists(l1->next, l2);
   } else {
      result = l2;
      result->next = mergeLists(l1, l2->next);
   }
   return result;
}

void display(struct node *head) {
   while (head != NULL) {
      printf("%d -> ", head->data);
      head = head->next;
   }
   printf("NULL\n");
}

int main() {
   struct node *l1 = NULL, *l2 = NULL, *temp, *newNode;
   int n1, n2, i, value;

   printf("Enter number of nodes in list 1: ");
   scanf("%d", &n1);
   for (i = 0; i < n1; i++) {
      newNode = malloc(sizeof(struct node));
      scanf("%d", &value);
      newNode->data = value;
      newNode->next = l1;
      l1 = newNode;
   }

   printf("Enter number of nodes in list 2: ");
   scanf("%d", &n2);
   for (i = 0; i < n2; i++) {
      newNode = malloc(sizeof(struct node));
      scanf("%d", &value);
```

```
        newNode->data = value;
        newNode->next = l2;
        l2 = newNode;
    }

    printf("Merged List:\n");
    display(mergeLists(l1, l2));

    return 0;
}
```

Output

```
Merged List:
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL
```

Key Notes

- No new nodes are created.
- Nodes are rearranged by changing links.
- Uses recursive merging approach.
- Time complexity: O(n + m).

Q30. Rotate Linked List Right by K Positions

Question
Rotate a singly linked list to the **right by K positions**.

Aim

To rotate the nodes of a linked list to the right without creating new nodes.

Algorithm

1. If the list is empty or $K = 0$, return the list as it is.
2. Traverse the list to find its length n and last node.
3. Compute $K = K \% n$ (effective rotations).
4. Connect the last node to the head to make the list circular.
5. Traverse $(n - K)$ nodes from the head to find the new tail.
6. The next node of the new tail becomes the new head.
7. Break the circular link.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};
```

```
struct node* rotateRight(struct node* head, int k) {
    if (head == NULL || k == 0)
        return head;

    struct node *temp = head;
    int n = 1;

    while (temp->next != NULL) {
        temp = temp->next;
        n++;
    }

    k = k % n;
    if (k == 0)
        return head;

    temp->next = head;  // make circular

    int steps = n - k;
    struct node *newTail = head;

    for (int i = 1; i < steps; i++)
        newTail = newTail->next;

    struct node *newHead = newTail->next;
    newTail->next = NULL;

    return newHead;
}

void display(struct node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct node *head = NULL, *temp, *newNode;
    int n, i, value, k;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        newNode = (struct node*)malloc(sizeof(struct node));
        scanf("%d", &value);
        newNode->data = value;
        newNode->next = NULL;

        if (head == NULL)
            head = temp = newNode;
        else {
            temp->next = newNode;
            temp = newNode;
        }
    }

    printf("Enter K: ");
```

```
    scanf("%d", &k);

    head = rotateRight(head, k);

    printf("List after rotation:\n");
    display(head);

    return 0;
}
```

Output

```
Enter number of nodes: 5
1 2 3 4 5
Enter K: 2
List after rotation:
4 -> 5 -> 1 -> 2 -> 3 -> NULL
```

Key Notes

• Rotation is done by pointer manipulation.
• No new nodes are created.
• Circular linking simplifies rotation logic.
• Time complexity: O(n).

Q31. Sort a Linked List (Merge Sort on Linked List)

Question
Sort a linked list using the **merge sort** technique.

Aim

To sort a singly linked list efficiently using merge sort.

Algorithm

1. If the list has 0 or 1 node, it is already sorted.
2. Find the middle of the linked list using slow and fast pointers.
3. Split the list into two halves.
4. Recursively apply merge sort on both halves.
5. Merge the two sorted halves into one sorted list.
6. Display the sorted linked list.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};
```

```
struct node* merge(struct node* a, struct node* b) {
    if (!a) return b;
    if (!b) return a;

    struct node* result;
    if (a->data <= b->data) {
        result = a;
        result->next = merge(a->next, b);
    } else {
        result = b;
        result->next = merge(a, b->next);
    }
    return result;
}

void split(struct node* source, struct node** front, struct node** back) {
    struct node *slow = source, *fast = source->next;

    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }

    *front = source;
    *back = slow->next;
    slow->next = NULL;
}

void mergeSort(struct node** headRef) {
    struct node* head = *headRef;
    struct node *a, *b;

    if (head == NULL || head->next == NULL)
        return;

    split(head, &a, &b);
    mergeSort(&a);
    mergeSort(&b);

    *headRef = merge(a, b);
}

void display(struct node* head) {
    while (head) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct node *head = NULL, *temp, *newNode;
    int n, i, value;

    printf("Enter number of nodes: ");
    scanf("%d", &n);
```

```
    for (i = 0; i < n; i++) {
        newNode = malloc(sizeof(struct node));
        scanf("%d", &value);
        newNode->data = value;
        newNode->next = NULL;

        if (head == NULL)
            head = temp = newNode;
        else {
            temp->next = newNode;
            temp = newNode;
        }
    }

    mergeSort(&head);

    printf("Sorted Linked List:\n");
    display(head);

    return 0;
}
```

Output

```
Enter number of nodes: 5
3 1 4 2 5
Sorted Linked List:
1 -> 2 -> 3 -> 4 -> 5 -> NULL
```

Key Notes

- Merge sort is efficient for linked lists.
- No random access is required.
- Time complexity: O(n log n).
- Space complexity: O(log n) due to recursion.

Q32. Find Middle Element in One Pass

Question
Find the **middle element** of a linked list in a **single traversal**.

Aim

To identify the middle node of a linked list using only one pass.

Algorithm

1. Initialize two pointers: slow and fast at the head.
2. Move slow by one node and fast by two nodes at a time.
3. When fast reaches the end, slow will be at the middle.
4. Print the data of the slow pointer.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

void findMiddle(struct node *head) {
   struct node *slow = head, *fast = head;

   if (head == NULL) {
      printf("List is empty\n");
      return;
   }

   while (fast != NULL && fast->next != NULL) {
      slow = slow->next;
      fast = fast->next->next;
   }

   printf("Middle element: %d\n", slow->data);
}

int main() {
   struct node *head = NULL, *temp, *newNode;
   int n, i, value;

   printf("Enter number of nodes: ");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      newNode = (struct node*)malloc(sizeof(struct node));
      scanf("%d", &value);
      newNode->data = value;
      newNode->next = NULL;

      if (head == NULL)
         head = temp = newNode;
      else {
         temp->next = newNode;
         temp = newNode;
      }
   }

   findMiddle(head);

   return 0;
}
```

Output

```
Enter number of nodes: 5
1 2 3 4 5
Middle element: 3
```

Key Notes

• Uses slow–fast pointer technique.
• Requires only one traversal.
• Works for both odd and even length lists.
• Time complexity: O(n).

Q33. Pairwise Swap Nodes

(Swap nodes, not data)

Question
Swap nodes of a linked list in pairs without swapping the data.

Aim

To swap adjacent nodes in a linked list by changing links only.

Algorithm

1. If the list is empty or has only one node, stop.
2. Swap the first two nodes by adjusting pointers.
3. Move to the next pair of nodes.
4. Repeat until the end of the list is reached.
5. Display the modified list.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

struct node* pairwiseSwap(struct node* head) {
   if (head == NULL || head->next == NULL)
      return head;

   struct node *prev = NULL;
   struct node *curr = head;
   head = curr->next;

   while (curr != NULL && curr->next != NULL) {
      struct node *next = curr->next;
      struct node *nextPair = next->next;

      next->next = curr;
      curr->next = nextPair;

      if (prev != NULL)
         prev->next = next;
```

```
        prev = curr;
        curr = nextPair;
    }
    return head;
}

void display(struct node* head) {
    while (head) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct node *head = NULL, *temp, *newNode;
    int n, i, value;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        newNode = malloc(sizeof(struct node));
        scanf("%d", &value);
        newNode->data = value;
        newNode->next = NULL;

        if (head == NULL)
            head = temp = newNode;
        else {
            temp->next = newNode;
            temp = newNode;
        }
    }

    head = pairwiseSwap(head);

    printf("List after pairwise swap:\n");
    display(head);

    return 0;
}
```

Output

```
Enter number of nodes: 5
1 2 3 4 5
List after pairwise swap:
2 -> 1 -> 4 -> 3 -> 5 -> NULL
```

Key Notes

- Nodes are swapped, not data.
- Pointer manipulation only.
- Preserves remaining node order.
- Time complexity: O(n).

Q34. Split Linked List into Two Alternating Lists

Question
Split a linked list into two lists such that:

- **L1** contains nodes at **odd positions**
- **L2** contains nodes at **even positions**

Aim

To divide a linked list into two separate linked lists based on node positions.

Algorithm

1. Initialize two empty lists: L1 and L2.
2. Traverse the original list while maintaining a position counter.
3. If the position is odd, append the node to L1.
4. If the position is even, append the node to L2.
5. Continue until the end of the list.
6. Display both lists.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

void splitList(struct node *head, struct node **l1, struct node **l2) {
   struct node *t1 = NULL, *t2 = NULL;
   int pos = 1;

   while (head != NULL) {
      struct node *newNode = (struct node *)malloc(sizeof(struct node));
      newNode->data = head->data;
      newNode->next = NULL;

      if (pos % 2 == 1) {
         if (*l1 == NULL)
            *l1 = t1 = newNode;
         else {
            t1->next = newNode;
            t1 = newNode;
         }
      } else {
         if (*l2 == NULL)
            *l2 = t2 = newNode;
         else {
            t2->next = newNode;
            t2 = newNode;
         }
```

```
        }
        pos++;
        head = head->next;
    }
}

void display(struct node *head) {
    while (head) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct node *head = NULL, *temp, *newNode;
    struct node *L1 = NULL, *L2 = NULL;
    int n, i, value;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        newNode = malloc(sizeof(struct node));
        scanf("%d", &value);
        newNode->data = value;
        newNode->next = NULL;

        if (head == NULL)
            head = temp = newNode;
        else {
            temp->next = newNode;
            temp = newNode;
        }
    }

    splitList(head, &L1, &L2);

    printf("Odd Position List (L1):\n");
    display(L1);

    printf("Even Position List (L2):\n");
    display(L2);

    return 0;
}
```

Output

```
Enter number of nodes: 6
1 2 3 4 5 6
Odd Position List (L1):
1 -> 3 -> 5 -> NULL
Even Position List (L2):
2 -> 4 -> 6 -> NULL
```

Key Notes

• Original order is preserved.
• Position-based splitting is used.
• Useful in alternating data separation.
• Time complexity: O(n).

Q35. Number of Ways to Climb Stairs

Question
Alex can climb **1 stair or 2 stairs** at a time.
Given n stairs, find the number of **distinct ways** to reach the top.

Aim

To compute the number of ways to climb n stairs using dynamic programming.

Algorithm

1. If n = 1, ways = 1.
2. If n = 2, ways = 2.
3. For n > 2:
    - ways[n] = ways[n-1] + ways[n-2]
4. Print ways[n].

Program

```
#include <stdio.h>

int main() {
   int n, i;
   int ways[50];

   printf("Enter number of stairs: ");
   scanf("%d", &n);

   if (n == 1) {
      printf("Number of ways: 1\n");
      return 0;
   }

   ways[1] = 1;
   ways[2] = 2;

   for (i = 3; i <= n; i++) {
      ways[i] = ways[i - 1] + ways[i - 2];
   }

   printf("Number of ways: %d\n", ways[n]);
   return 0;
}
```

Output

```
Enter number of stairs: 4
```

Number of ways: 5

Key Notes

• This is a Fibonacci-type problem.
• Dynamic programming avoids recomputation.
• Time complexity: O(n).
• Space complexity: O(n).

Q36. Count Books Looked at Before Reaching Target

Question
A librarian searches books one by one from the top.
Given total books `n` and the **target position**, count how many books were looked at using **recursion**.

Aim

To count the number of books examined before finding the target book using a recursive approach.

Algorithm

1. If `target == 1`, return 1 (book found immediately).
2. Otherwise, recursively reduce the problem size by 1.
3. Each recursive call represents looking at one book.
4. Return the count when the target is reached.

Program

```
#include <stdio.h>

int countBooks(int target) {
   if (target == 1)
      return 1;
   return 1 + countBooks(target - 1);
}

int main() {
   int n, target;

   printf("Enter total number of books: ");
   scanf("%d", &n);

   printf("Enter target book position: ");
   scanf("%d", &target);

   if (target > n || target <= 0) {
      printf("Invalid target position\n");
   } else {
      printf("Books looked at: %d\n", countBooks(target));
   }
```

```
    return 0;
}
```

Output

```
Enter total number of books: 5
Enter target book position: 3
Books looked at: 3
```

Key Notes

• Uses recursion to simulate searching.
• Each recursive call represents one book checked.
• Simple base case ensures termination.
• Time complexity: O(target).

Q37. Remove Duplicate Marbles and Print Unique Count

Question
Mahesh has n marbles of different weights.
Remove duplicate marbles, count **unique marbles**, and print the new collection from smallest to largest.

Aim

To remove duplicate values from an array, sort the remaining values, and count unique marbles.

Algorithm

1. Read the number of marbles n and their weights into an array.
2. Sort the array in ascending order.
3. Traverse the sorted array and remove duplicates in-place.
4. Count the number of unique marbles.
5. Print the unique count and the unique marble weights.

Program

```
#include <stdio.h>

int main() {
    int arr[50], n, i, j, temp;
    int uniqueCount;

    printf("Enter number of marbles: ");
    scanf("%d", &n);

    printf("Enter marble weights:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
```

```
    /* Sorting */
    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;
            }
        }
    }

    uniqueCount = 1;
    for (i = 1; i < n; i++) {
        if (arr[i] != arr[i - 1]) {
            arr[uniqueCount++] = arr[i];
        }
    }

    printf("Number of unique marbles: %d\n", uniqueCount);
    printf("Unique marbles:\n");
    for (i = 0; i < uniqueCount; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Enter number of marbles: 11
Enter marble weights:
2 1 3 2 1 4 3 4 2 5 1
Number of unique marbles: 5
Unique marbles:
1 2 3 4 5
```

Key Notes

- Sorting groups duplicates together.
- In-place duplicate removal saves memory.
- Final array contains only unique values.

Q38. Sort Book Titles Using Merge Sort

Question
Given an array of book titles (strings), sort them in alphabetical order using the **merge sort** algorithm.

Aim

To sort an array of strings using the divide-and-conquer approach of merge sort.

Algorithm

1. Divide the array of strings into two halves.
2. Recursively apply merge sort on both halves.
3. Merge the two sorted halves by comparing strings lexicographically.
4. Repeat until the entire list is sorted.
5. Print the sorted list of book titles.

Program

```
#include <stdio.h>
#include <string.h>

void merge(char arr[][50], int l, int m, int r) {
    int i, j, k;
    char temp[50][50];

    i = l;
    j = m + 1;
    k = l;

    while (i <= m && j <= r) {
        if (strcmp(arr[i], arr[j]) <= 0)
            strcpy(temp[k++], arr[i++]);
        else
            strcpy(temp[k++], arr[j++]);
    }

    while (i <= m)
        strcpy(temp[k++], arr[i++]);

    while (j <= r)
        strcpy(temp[k++], arr[j++]);

    for (i = l; i <= r; i++)
        strcpy(arr[i], temp[i]);
}

void mergeSort(char arr[][50], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

int main() {
    char books[10][50];
    int n, i;

    printf("Enter number of books: ");
    scanf("%d", &n);
    getchar();

    printf("Enter book titles:\n");
    for (i = 0; i < n; i++) {
        gets(books[i]);
    }
```

```
    mergeSort(books, 0, n - 1);

    printf("Sorted book titles:\n");
    for (i = 0; i < n; i++) {
        printf("%s\n", books[i]);
    }

    return 0;
}
```

Output

```
Enter number of books: 5
War and Peace
Anna Karenina
Moby Dick
Great Expectations
The Odyssey

Sorted book titles:
Anna Karenina
Great Expectations
Moby Dick
The Odyssey
War and Peace
```

Key Notes

- Merge sort is stable and efficient.
- String comparison uses `strcmp()`.
- Suitable for large datasets.
- Time complexity: O(n log n).

Q39. Reverse List Then Swap Nodes with Values A and B

Question
Given a linked list (stored in reverse order), reverse the list and then swap the positions of two given node values **A** and **B**.

Aim

To reverse a linked list and swap two specified nodes by changing links (not data).

Algorithm

Reverse the List

1. Initialize `prev = NULL`, `curr = head`.
2. Traverse the list and reverse the `next` links.

Swap Nodes A and B
3. Find nodes with values `A` and `B` along with their previous nodes.

4. Adjust links to swap the two nodes.
5. Display the final list.

Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

struct node* reverse(struct node* head) {
   struct node *prev = NULL, *curr = head, *next;
   while (curr != NULL) {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
   return prev;
}

struct node* swapNodes(struct node* head, int x, int y) {
   if (x == y) return head;

   struct node *prevX = NULL, *currX = head;
   while (currX && currX->data != x) {
      prevX = currX;
      currX = currX->next;
   }

   struct node *prevY = NULL, *currY = head;
   while (currY && currY->data != y) {
      prevY = currY;
      currY = currY->next;
   }

   if (currX == NULL || currY == NULL)
      return head;

   if (prevX != NULL)
      prevX->next = currY;
   else
      head = currY;

   if (prevY != NULL)
      prevY->next = currX;
   else
      head = currX;

   struct node *temp = currX->next;
   currX->next = currY->next;
   currY->next = temp;

   return head;
}
```

```
void display(struct node* head) {
   while (head) {
      printf("%d -> ", head->data);
      head = head->next;
   }
   printf("NULL\n");
}

int main() {
   struct node *head = NULL, *temp, *newNode;
   int n, i, value, a, b;

   printf("Enter number of nodes: ");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      newNode = malloc(sizeof(struct node));
      scanf("%d", &value);
      newNode->data = value;
      newNode->next = NULL;

      if (head == NULL)
         head = temp = newNode;
      else {
         temp->next = newNode;
         temp = newNode;
      }
   }

   printf("Enter values to swap (A B): ");
   scanf("%d %d", &a, &b);

   head = reverse(head);
   head = swapNodes(head, a, b);

   printf("Final List:\n");
   display(head);

   return 0;
}
```

Output

```
Enter number of nodes: 6
101 102 105 103 104 106
Enter values to swap (A B): 105 106
Final List:
106 -> 104 -> 103 -> 105 -> 102 -> 101 -> NULL
```

Key Notes

- List is reversed before swapping.
- Nodes are swapped by link manipulation.
- Data inside nodes is not changed.
- Time complexity: O(n).

Q40. Balanced Parentheses Checker

Question
Given an expression, check whether the parentheses **(), {}, []** are **balanced and properly nested**.

Examples
Input: [()]{}{[()()]()} → Output: **True**
Input: [(]) → Output: **False**

Aim

To verify whether an expression has balanced parentheses using a stack.

Algorithm

1. Initialize an empty stack.
2. Traverse the expression character by character.
3. If the character is an opening bracket (, {, [, push it onto the stack.
4. If the character is a closing bracket), },]:
    - If stack is empty, return False.
    - Pop the top element and check for matching pair.
5. After traversal, if stack is empty → balanced.
6. Otherwise → not balanced.

Program

```
#include <stdio.h>
#include <string.h>
#define MAX 100

char stack[MAX];
int top = -1;

void push(char ch) {
   stack[++top] = ch;
}

char pop() {
   return stack[top--];
}

int isMatching(char open, char close) {
   if (open == '(' && close == ')') return 1;
   if (open == '{' && close == '}') return 1;
   if (open == '[' && close == ']') return 1;
   return 0;
}

int main() {
   char exp[MAX];
   int i;

   printf("Enter expression: ");
   scanf("%s", exp);

   for (i = 0; i < strlen(exp); i++) {
```

```
        if (exp[i] == '(' || exp[i] == '{' || exp[i] == '[') {
            push(exp[i]);
        } else if (exp[i] == ')' || exp[i] == '}' || exp[i] == ']') {
            if (top == -1 || !isMatching(pop(), exp[i])) {
                printf("False\n");
                return 0;
            }
        }
    }

    if (top == -1)
        printf("True\n");
    else
        printf("False\n");

    return 0;
}
```

Output

```
Enter expression: [()]{}{[()()]()}
True
```

Key Notes

• Stack ensures proper nesting.
• Opening brackets are pushed; closing brackets are matched.
• Empty stack at end confirms balance.
• Time complexity: O(n).

Q41. Reverse the First K Elements of a Queue

Question
Reverse the order of the **first K elements** of a queue while keeping the remaining elements in the same order.

Example
Input Queue: {1, 2, 3, 4, 5}, K = 3
Output Queue: {3, 2, 1, 4, 5}

Aim

To reverse only the first K elements of a queue using a stack.

Algorithm

1. Read the queue elements and the value K.
2. Dequeue the first K elements and push them onto a stack.
3. Pop elements from the stack and enqueue them back into the queue.
4. Move the remaining (n − K) elements from front to rear to maintain order.
5. Display the modified queue.

Program

```
#include <stdio.h>
#define MAX 50

int queue[MAX], stack[MAX];
int front = 0, rear = -1, top = -1;

void enqueue(int x) {
   queue[++rear] = x;
}

int dequeue() {
   return queue[front++];
}

void push(int x) {
   stack[++top] = x;
}

int pop() {
   return stack[top--];
}

int main() {
   int n, k, i, value;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter queue elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &value);
      enqueue(value);
   }

   printf("Enter K: ");
   scanf("%d", &k);

   for (i = 0; i < k; i++) {
      push(dequeue());
   }

   while (top != -1) {
      enqueue(pop());
   }

   for (i = 0; i < n - k; i++) {
      enqueue(dequeue());
   }

   printf("Queue after reversing first %d elements:\n", k);
   for (i = front; i <= rear; i++) {
      printf("%d ", queue[i]);
   }

   return 0;
}
```

Output

```
Enter number of elements: 5
Enter queue elements:
1 2 3 4 5
Enter K: 3
Queue after reversing first 3 elements:
3 2 1 4 5
```

Key Notes

• Stack is used to reverse order.
• Remaining elements preserve original order.
• Common queue manipulation problem.
• Time complexity: O(n).

Q42. Count Total Web Pages and Hyperlinks

Question
Given an adjacency matrix representing a web graph, count:

- Total number of web pages (nodes)
- Total number of directed hyperlinks (edges)

Aim

To count the number of vertices and directed edges in a web graph.

Algorithm

1. Read the adjacency matrix of size $n \times n$.
2. Number of web pages = n.
3. Initialize edge count = 0.
4. Traverse the matrix:
    - If matrix[i][j] == 1, increment edge count.
5. Print total pages and hyperlinks.

Program

```
#include <stdio.h>

int main() {
   int graph[10][10], n, i, j;
   int links = 0;

   printf("Enter number of web pages: ");
   scanf("%d", &n);

   printf("Enter adjacency matrix:\n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         scanf("%d", &graph[i][j]);
```

```
            if (graph[i][j] == 1)
                links++;
        }
    }

    printf("Number of web pages: %d\n", n);
    printf("Number of hyperlinks: %d\n", links);

    return 0;
}
```

Output

```
Enter number of web pages: 5
Enter adjacency matrix:
0 1 1 0 0
0 0 1 0 0
1 0 0 0 0
0 0 1 1 0
0 0 0 0 0
Number of web pages: 5
Number of hyperlinks: 6
```

Key Notes

• Adjacency matrix represents directed graph.
• Each 1 indicates a hyperlink.
• Pages = matrix size.
• Time complexity: $O(n^2)$.

Q43. Heap Sort (Max Heap & Min Heap)

Question

Implement **Heap Sort** using:

1. **Max Heap** to sort elements in **ascending order**
2. **Min Heap** to sort elements in **descending order**

A) HEAP SORT USING MAX HEAP (Ascending Order)

Aim

To sort an array in ascending order using the **Max Heap** technique.

Algorithm (Max Heap)

1. Build a **Max Heap** from the given array.
2. Swap the root (largest element) with the last element.
3. Reduce heap size by 1.
4. Heapify the root again.

5. Repeat until the array is sorted.

Program

```
#include <stdio.h>

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

void heapify(int arr[], int n, int i) {
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;

   if (left < n && arr[left] > arr[largest])
      largest = left;

   if (right < n && arr[right] > arr[largest])
      largest = right;

   if (largest != i) {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
   }
}

void heapSort(int arr[], int n) {
   int i;

   for (i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);

   for (i = n - 1; i > 0; i--) {
      swap(&arr[0], &arr[i]);
      heapify(arr, i, 0);
   }
}

int main() {
   int n, i, arr[50];

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++)
      scanf("%d", &arr[i]);

   heapSort(arr, n);

   printf("Sorted array:\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);

   return 0;
}
```

Output

Enter number of elements: 6
Enter elements:
12 11 13 5 6 7
Sorted array:
5 6 7 11 12 13

B) HEAP SORT USING MIN HEAP (Descending Order)

Aim

To sort an array in descending order using the **Min Heap** technique.

Algorithm (Min Heap)

1. Build a **Min Heap** from the array.
2. Swap the root (smallest element) with the last element.
3. Reduce heap size by 1.
4. Heapify the root.
5. Repeat until sorting is complete.

Program

```
#include <stdio.h>

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

void minHeapify(int arr[], int n, int i) {
   int smallest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;

   if (left < n && arr[left] < arr[smallest])
      smallest = left;

   if (right < n && arr[right] < arr[smallest])
      smallest = right;

   if (smallest != i) {
      swap(&arr[i], &arr[smallest]);
      minHeapify(arr, n, smallest);
   }
}

void heapSort(int arr[], int n) {
   int i;

   for (i = n / 2 - 1; i >= 0; i--)
      minHeapify(arr, n, i);
```

```
    for (i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        minHeapify(arr, i, 0);
    }
}

int main() {
    int n, i, arr[50];

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    heapSort(arr, n);

    printf("Sorted array (Descending order):\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output

```
Enter number of elements: 6
Enter elements:
12 11 13 5 6 7
Sorted array (Descending order):
13 12 11 7 6 5
```