D Y PATIL
INTERNATIONAL
UNIVERSITY
AKURDI PUNE

D Y PATIL INTERNATIONAL UNIVERSITY
SCHOOLOFCOMPUTERSCIENCEENGINEERINGANDAPPLICATIONS

Program Name: B. Tech (CSE)

Academic Year 2025-26

First Year B. Tech (CSE), Sem I

LAB MANUAL

Subject Code: 1004

Data Structures using C

Course Instructor : Dr. Amol Dhakne

Lab Instructor : Ms. Priyanka Sajnani

Submitted By: Balwant Mundhe

PRN: 202508022296

Department of Computer Science, Engineering and Applications
D. Y. Patil International University, Akurdi, Pune - 411035

D Y PATIL
INTERNATIONAL
UNIVERSITY
AKURDI PUNE

SCHOOL OF COMPUTER SCIENCE ENGINEERING AND APPLICATIONS

CERTIFICATE

This is to certify that Mr. Balwant Shankar Mundhe , of First Year B. Tech CSE student, Sem I , PRN No. 20250802296 , has successfully completed the lab manual for the course Data Structure Using C in partial fulfilment of the requirements for the degree of Bachelor of Technology in Computer Science Engineering for the Academic Year 2025-26 .

Ms. Priyanka Sajnani
Teaching Associate

Dr. Amol Dhakne
Course Instructor

INDEX

Sr. No.	Name of Experiment	Lab Conduction Date	Page No.	Teacher's Sign
1	a) Write a C program to demonstrate use and operations on 1D and 2D Arrays			
2	a) Write a C program to demonstrate working of linear search algorithm b) Write a C program to demonstrate working of binary search algorithm			
3	a) Write a C program to demonstrate working of Bubble sort algorithm b) Write a C program to demonstrate working of selection sort algorithm			
4	a) Write a C program to demonstrate working of Insertion sort algorithm b) Write a C program to demonstrate working of Merge sort algorithm			
5	a) Write a C program to demonstrate working of Quick sort algorithm b) Write a C program to demonstrate working of Radix sort algorithm			
6	a) Write a C program to demonstrate various operations on stack			
7	a) Write a C program to demonstrate various operations on queue b) Write a C program to demonstrate various operations on circular queue			
8	a) Write a C program to demonstrate various operations on Single linked list b) Write a C program to demonstrate various operations on double linked list c) Write a C program to demonstrate various operations on circular linked list			
9	a) Write a C program to demonstrate various operations on Tree data structure b) Write a C program to demonstrate tree traversal methods			
10	a) Write a C program to demonstrate various operations on graph data structure b) Write a C program to demonstrate DFS graph traversal c) Write a C program to demonstrate BFS graph traversal			

(Note: The dates will be announced a week before the practical, write it with pencil)

Data Structures Using C PRACTICAL: 1

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: A C program to demonstrate use and operations on 1D and 2D Arrays

Theory/Concept:

Arrays are fundamental data structures used to store collections of elements of the same data type in contiguous memory locations. They can be one-dimensional (1D) or multi-dimensional, such as two-dimensional (2D).

Algorithm/Pseudo code:

```
// 1D Array Operations
FUNCTION main()

  // Declare and Initialize a 1D Array
  DECLARE int oneDArray[5] = {10, 20, 30, 40, 50}

  // Accessing elements
  PRINT "Element at index 0 of 1D array: ", oneDArray[0]

  // Modifying elements
  oneDArray[2] = 35
  PRINT "Modified element at index 2 of 1D array: ", oneDArray[2]

  // Traversing and printing all elements
  PRINT "Elements of 1D array:"
  FOR i FROM 0 TO 4
    PRINT oneDArray[i]
  END FOR

  // Calculating sum of elements
  DECLARE int sum1D = 0
  FOR i FROM 0 TO 4
    sum1D = sum1D + oneDArray[i]
  END FOR
  PRINT "Sum of 1D array elements: ", sum1D

  // 2D Array Operations
```

```
  //Declare and Initialize a 2D Array (3 rows, 4 columns)
  DECLARE int twoDArray[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}

  //Accessing elements
  PRINT "Element at row 1, column 2 of 2D array: ", twoDArray[1][2]

  //Modifying elements twoDArray[0][3] = 100 PRINT "Modified element at
  row 0, column 3 of 2D array: ", twoDArray[0][3]


  //Traversing and printing all elements
  PRINT "Elements of 2D array:"
  FORrow FROM 0 TO 2
    FOR col FROM 0 TO 3
      PRINT twoDArray[row][col], " "
    END FOR
    PRINT NEWLINE // New line after each row
  ENDFOR

  //Calculating sum of all elements
  DECLARE int sum2D = 0
  FORrow FROM 0 TO 2
    FOR col FROM 0 TO 3
      sum2D = sum2D + twoDArray[row][col]
    END FOR
  ENDFOR
  PRINT "Sum of 2D array elements: ", sum2D

ENDFUNCTION
```

Program code:

```c
#include <stdio.h>

int main() {
    // --- 1D Array Operations ---
    printf("--- 1D Array Operations ---\n");

    // Declaration and Initialization of a 1D array
    int oneDArray[5] = {10, 20, 30, 40, 50};

    // Accessing and Printing elements of a 1D array
    printf("Elements of 1D Array: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", oneDArray[i]);
    }
    printf("\n");

    // Modifying an element in a 1D array
    oneDArray[2] = 35;
    printf("1D Array after modification: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", oneDArray[i]);
    }
    printf("\n");

    // Calculating the sum of elements in a 1D array
    int sum1D = 0;
    for (int i = 0; i < 5; i++) {
        sum1D += oneDArray[i];
    }
    printf("Sum of elements in 1D Array: %d\n\n", sum1D);

    // --- 2D Array Operations ---
    printf("--- 2D Array Operations ---\n");

    // Declaration and Initialization of a 2D array (matrix)
    int twoDArray[3][3] = {
        {1, 2, 3},
```

```c
        {4, 5, 6},
        {7, 8, 9}
    };

    // Accessing and Printing elements of a 2D array
    printf("Elements of 2D Array:\n");
    for (int i = 0; i < 3; i++) { // Iterate through rows
        for (int j = 0; j < 3; j++) { // Iterate through columns
            printf("%d ", twoDArray[i][j]);
        }
        printf("\n"); // New line after each row
    }

    // Modifying an element in a 2D array
    twoDArray[1][1] = 55;
    printf("2D Array after modification:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", twoDArray[i][j]);
        }
        printf("\n");
    }

    // Calculating the sum of all elements in a 2D array
    int sum2D = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            sum2D += twoDArray[i][j];
        }
    }
    printf("Sum of elements in 2D Array: %d\n", sum2D);

    return 0;
}
```

Output/Results:

```
--- 1D Array Operations ---
Elements of 1D Array: 10 20 30 40 50
1D Array after modification: 10 20 35 40 50
Sum of elements in 1D Array: 155

--- 2D Array Operations ---
Elements of 2D Array:
1 2 3
4 5 6
7 8 9
2D Array after modification:
1 2 3
4 55 6
7 8 9
Sum of elements in 2D Array: 95


=== Code Execution Successful ===
```

Relative Applications:

1D Arrays:

A 1D array is a linear collection of elements, where each element is identified by a single index.

- Real-life uses:
  - Shopping List: A list of items to buy, where each item is an element in the array.
  - Contact List: Storing phone numbers or names sequentially.
  - Leaderboard: Ranking players in a game based on their scores.
  - Playlist: Organizing songs in a specific order.

2D Arrays:

A 2D array, also known as a matrix, is a collection of elements arranged in rows and columns, where each element is identified by two indices (row and column).

- Real-life uses:

- Spreadsheet/Table: Representing data in rows and columns, like a financial spreadsheet or a product catalog.
- Image Processing: An image can be viewed as a 2D array of pixels, where each pixel has color information.
- Game Boards: Representing the layout of a chessboard or a tic-tac-toe board.
- Seating Arrangements: Mapping seats in a theater or airplane by row and seat number.

Conclusion: The choice between 1D and 2D arrays depends on the inherent structure of the data being stored. 1D arrays offer simplicity and efficiency for linear data, while 2D arrays provide a structured way to manage and operate on tabular or grid-based information. Understanding their respective uses and operations is crucial for efficient data management and problem-solving in programming.

Data Structures Using C
PRACTICAL: 2

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: A). C program to demonstrate working of linear search algorithm.
B).C program to demonstrate working of binary search algorithm.

Theory/Concept:

Linear Search

Linear search, also known as sequential search, operates by examining each element in a list one by one, in sequential order, until the target element is found or the end of the list is reached. It does not require the list to be sorted.

Binary Search

Binary search is an efficient search algorithm that works on sorted lists or arrays. It repeatedly divides the search interval in half. The algorithm compares the target value to the middle element of the array. If the target value is less than the middle element, it searches in the left half; if greater, it searches in the right half. This process continues until the target element is found or the interval becomes empty.

Algorithm/Pseudo code:

1. Linear Search Pseudocode:

```
FUNCTION LinearSearch(array, target_element):
  FOR index FROM 0 TO length(array) - 1:
    IF array[index] IS EQUAL TO target_element:
      RETURN index // Element found at this index
  RETURN -1 // Element not found
```

2. Binary Search Pseudocode :

```
FUNCTION BinarySearch(array, target_element):
  SET low = 0
  SET high = length(array) - 1

  WHILE low IS LESS THAN OR EQUAL TO high:
```

SET mid = low + (high - low) / 2 // Calculate the middle index

IF array[mid] IS EQUAL TO target_element:
RETURN mid // Element found at this index
ELSE IF array[mid] IS LESS THAN target_element:
SET low = mid + 1 // Search in the right half
ELSE:
SET high = mid - 1 // Search in the left half

RETURN -1 // Element not found

Program code:

For Linear search

```c
#include <stdio.h>

// Function to perform linear search
int linearSearch(int arr[], int size, int key) {
    // Iterate through each element of the array
    for (int i = 0; i < size; i++) {
        // If the current element matches the key, return its index
        if (arr[i] == key) {
            return i;
        }
    }
    // If the key is not found in the array, return -1
    return -1;
}

int main() {
    int arr[] = {10, 25, 35, 45, 50, 60}; // Example array
    int size = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array
    int key; // Element to search for

    printf("Enter the element to search: ");
    scanf("%d", &key); // Get the key from user input

    int result = linearSearch(arr, size, key); // Call the linear search function

    // Print the result based on the return value of linearSearch
    if (result != -1) {
        printf("Element %d found at index %d\n", key, result);
    } else {
        printf("Element %d not found in the array\n", key);
    }

    return 0;
}
```

For Binary search:

```c
#include <stdio.h>

// Function to perform binary search
int binarySearch(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;

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

        // Check if target is present at mid
        if (arr[mid] == target) {
            return mid; // Element found, return its index
        }
        // If target is greater, ignore left half
        else if (arr[mid] < target) {
            low = mid + 1;
        }
        // If target is smaller, ignore right half
        else {
            high = mid - 1;
        }
    }

    return -1; // Element not found, return -1
```

```c
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int size = sizeof(arr) / sizeof(arr[0]); // Calculate array size
    int target = 23; // Element to search for

    int result = binarySearch(arr, size, target);

    if (result == -1) {
        printf("Element %d is not present in the array.\n", target);
    } else {
        printf("Element %d is present at index %d.\n", target, result);
    }

    // Example of a target not found
    target = 100;
    result = binarySearch(arr, size, target);
    if (result == -1) {
        printf("Element %d is not present in the array.\n", target);
    } else {
        printf("Element %d is present at index %d.\n", target, result);
    }

    return 0;
}
```

Output/Results:

For Linear search:

```
Enter the element to search: 35
Element 35 found at index 2


=== Code Execution Successful ===
```

For Binary search:

```
Element 23 is present at index 5.
Element 100 is not present in the array.


=== Code Execution Successful ===
```

Relative Applications:

Linear Search (Sequential Search):

Unsorted or Small Datasets:
Used when data is not sorted or when the dataset is small enough that the overhead of sorting (for binary search) is not justified.
Example: Searching for a specific contact in a small, unsorted list on a basic phone, or finding a particular item in a small inventory list.

Binary Search:

Sorted and Large Datasets:
Ideal for efficiently searching through large datasets that are already sorted.

Example: Looking up a word in a dictionary (words are sorted alphabetically), finding a specific book in a library's catalog (if sorted by title or author), or searching for a value in a sorted database index.

Conclusion:

The project on linear and binary search algorithms demonstrates two fundamental approaches to data retrieval, each with distinct characteristics and optimal use cases. Linear search, while simple to implement and applicable to both sorted and unsorted data, exhibits a time complexity of O(n), making it less efficient for large datasets. In contrast, binary search, leveraging a divide-and-conquer strategy, offers significantly improved performance with a time complexity of O(log n) for large, sorted datasets.
The choice between these algorithms is contingent upon the specific requirements of the application, primarily the size and organization of the data. For smaller datasets or when data is unsorted, the simplicity and versatility of linear search prove advantageous. However, for large, sorted collections where fast lookups are paramount, binary search emerges as the superior and more efficient solution. Understanding these distinctions is crucial for selecting the appropriate search algorithm to optimize performance in real-world scenarios.

Data Structures Using C
PRACTICAL: 3

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: a) C program to demonstrate working of Bubble sort algorithm.
b) C program to demonstrate working of selection sort algorithm.

Theory/Concept:

Bubble Sort:

Bubble sort operates by repeatedly stepping through the list, comparing each pair of adjacent elements and swapping them if they are in the wrong order. The process is repeated until no swaps are needed, indicating that the list is sorted. Larger elements "bubble" to the end of the list with each pass.

Selection Sort:

Selection sort works by repeatedly finding the minimum element from the unsorted part of the list andputting it at the beginning of the sorted part.

Algorithm/Pseudo code:

BubbleSort Pseudocode:

```
FUNCTION BubbleSort(array)
  n=length of array
  FORiFROM 0 TO n-2
    FORjFROM 0 TO n-i-2
      IFarray[j] > array[j+1] THEN
        SWAP array[j] AND array[j+1]
      ENDIF
    ENDFOR
  ENDFOR
ENDFUNCTION
```

Selection Sort Pseudocode:

```
FUNCTIONSelectionSort(array)
  n = length of array
  FORiFROM0TOn-2
    minIndex = i
    FORjFROMi+1TO n-1
      IFarray[j]<array[minIndex] THEN
        minIndex = j
    END IF
    END FOR
    SWAParray[i]AND array[minIndex]
  END FOR
END FUNCTION
```

Program code:

For Bubble Sort:

```c
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Last i elements are already in place
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
```

```c

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array (Bubble Sort): ");
    printArray(arr, n);

    return 0;
}
```

For Selection Sort:

```c
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first element of the
            unsorted subarray
        swap(&arr[min_idx], &arr[i]);
```

```c
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array (Selection Sort): ");
    printArray(arr, n);

    return 0;
}
```

Output/Results:

For Bubble Sort:

```
Original array: 64 25 12 22 11
Sorted array (Bubble Sort): 11 12 22 25 64


=== Code Execution Successful ===
```

For Selection Sort:

```
Original array: 64 25 12 22 11
Sorted array (Selection Sort): 11 12 22 25 64


=== Code Execution Successful ===
```

Relative Applications:

Bubble sort real-life applications

Small databases:
Used to sort small, simple databases where performance is not a major concern.
Embedded systems:
Suitable for systems with limited resources where simplicity and predictable behavior are key.

Selection sort real-life applications

Small class lists: Sorting a small list of students by grades or names.
Organizing files: Sorting a small directory of files by creation date or size.

Conclusion:

Think of it like organizing a messy deck of cards:

Bubble Sort:

This is like repeatedly going through the cards, comparing each card to the one next to it. If they're in the wrong order (e.g., a 7 before a 5), you swap them. You keep doing this over and over until no more swaps are needed, meaning the cards are all in order. It's simple to understand but can be slow if you have a lot of cards, especially if they're very out of order.

Selection Sort:

This is like finding the smallest card in the entire messy pile and putting it at the beginning of your sorted pile. Then, you find the next smallest card from the remaining messy pile and put it next in your sorted pile. You repeat this until all cards are moved from the messy pile to the sorted pile. This method can be more efficient than Bubble Sort for many situations, as it generally involves fewer swaps, but it still has to look through a lot of cards to find the smallest one each time.

Data Structures Using C
PRACTICAL: 4

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: **A) A C program to demonstrate working of Insertion sort algorithm.**
B) A C program to demonstrate working of Merge sort algorithm.

Theory/Concept:

Insertion sort: It's like sorting a hand of playing cards. You keep a sorted portion on your left and take one card at a time from the unsorted pile on your right, inserting it into the correct place in your hand.

Merge sort: A "divide and conquer" algorithm that breaks a large problem into smaller, more manageable sub-problems, solves them, and then combines the solutions.

Algorithm/Pseudo code:

For Insertion sort:

```
FUNCTION InsertionSort(ARRAY arr)
FOR i FROM 1 TO LENGTH(arr) - 1
   key = arr[i]
   j = i - 1
   WHILE j >= 0 AND arr[j] > key
     arr[j + 1] = arr[j]
     j = j - 1
   END WHILE
   arr[j + 1] = key
  END FOR
END FUNCTION
```

For Merge sort:

```
FUNCTION MergeSort(ARRAY arr, INT left, INT right)
  IF left < right
    mid = (left + right) / 2
    MergeSort(arr, left, mid)
    MergeSort(arr, mid + 1, right)
    Merge(arr, left, mid, right)
  END IF
```

```
END FUNCTION

FUNCTION Merge(ARRAY arr, INT left, INT mid, INT right)
 // Create temporary arrays for the two halves
 n1 = mid - left + 1
 n2 = right - mid
 LEFT_ARRAY = new ARRAY of size n1
 RIGHT_ARRAY = new ARRAY of size n2

 // Copy data to temporary arrays
 FOR i FROM 0 TO n1 - 1
  LEFT_ARRAY[i] = arr[left + i]
 END FOR
 FOR j FROM 0 TO n2 - 1
  RIGHT_ARRAY[j] = arr[mid + 1 + j]
 END FOR

 // Merge the temporary arrays back into arr
 i = 0 // Initial index of first sub-array
 j = 0 // Initial index of second sub-array
 k = left // Initial index of merged sub-array

 WHILE i < n1 AND j < n2
  IF LEFT_ARRAY[i] <= RIGHT_ARRAY[j]
   arr[k] = LEFT_ARRAY[i]
   i = i + 1
  ELSE
   arr[k] = RIGHT_ARRAY[j]
   j = j + 1
  END IF
  k = k + 1
 END WHILE

 // Copy the remaining elements of LEFT_ARRAY, if any
 WHILE i < n1
  arr[k] = LEFT_ARRAY[i]
  i = i + 1
  k = k + 1
 END WHILE

 // Copy the remaining elements of RIGHT_ARRAY, if any
 WHILE j < n2
  arr[k] = RIGHT_ARRAY[j]
  j = j + 1
  k = k + 1
 END WHILE
END FUNCTION
```

Program code:

For Insertion sort:

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i]; // Element to be inserted
        j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key; // Insert the key at the correct position
    }
}

// Driver code to test insertion sort
int main() {
```

```c
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    insertionSort(arr, n);

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

    return 0;
}
```

For Merge Sort:

```c
#include <stdio.h>
#include <stdlib.h> // For malloc and free

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temporary arrays
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into arr[l..r]
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
```

```
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L); // Free dynamically allocated memory
    free(R);
}

// l is for left index and r is right index of the sub-array of arr to be sorted
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for large l and h
        int m = l + (r - l) / 2;

```

```
61          // Sort first and second halves
62          mergeSort(arr, l, m);
63          mergeSort(arr, m + 1, r);
64
65          merge(arr, l, m, r);
66      }
67  }
68
69  // Driver code to test merge sort
70  int main() {
71      int arr[] = {12, 11, 13, 5, 6, 7};
72      int n = sizeof(arr) / sizeof(arr[0]);
73
74      printf("Unsorted array: ");
75      for (int i = 0; i < n; i++) {
76          printf("%d ", arr[i]);
77      }
78      printf("\n");
79
80      mergeSort(arr, 0, n - 1);
81
82      printf("Sorted array (Merge Sort): ");
83      for (int i = 0; i < n; i++) {
84          printf("%d ", arr[i]);
85      }
86      printf("\n");
87
88      return 0;
89  }
```

Output/Results:

For Insertion Sort:

```
Unsorted array: 12 11 13 5 6
Sorted array (Insertion Sort): 5 6 11 12 13


=== Code Execution Successful ===
```

For Merge Sort:

```
Unsorted array: 12 11 13 5 6 7
Sorted array (Merge Sort): 5 6 7 11 12 13


=== Code Execution Successful ===
```

Relative Applications:

Insertion Sort Real-Time Example:

Sorting Playing Cards: When you arrange a hand of playing cards, you typically pick up one card at a time and insert it into its correct position within the already sorted cards in your hand. This process directly mirrors the Insertion Sort algorithm, where elements are taken one by one and inserted into their proper place in a growing sorted sub-array.

Merge Sort Real-Time Example:

Sorting Large Datasets in Databases: Imagine a large database system that needs to sort a massive amount of data, perhaps for indexing or generating reports. Merge Sort is often employed in such scenarios. The data might be too large to fit into memory, so it's divided into smaller, manageable chunks (sub-arrays). Each chunk is sorted independently, and then these sorted chunks are merged back together to form a fully sorted dataset. This divide-and-conquer approach is the core of Merge Sort.

Conclusion:

Insertion Sort is a simple, intuitive, and in-place sorting algorithm that builds a sorted array one element at a time. It excels with small datasets or nearly sorted arrays, achieving a best-case time complexity of O(n) in such scenarios. Its O(1) auxiliary space complexity is also advantageous in memory-constrained environments. However, for larger, unsorted datasets, its $O(n^2)$ worst and average-case time complexity makes it less efficient compared to more advanced algorithms.

Merge Sort, on the other hand, is a highly efficient, stable, and recursive algorithm based on the divide-and-conquer paradigm. It consistently delivers an O(n log n) time complexity across all cases (best, average, and worst), making it a preferred choice for large datasets where consistent performance is crucial. Its primary drawback is its O(n) auxiliary space complexity, as it requires extra space for temporary subarrays during the merging process.

In conclusion, the choice between Insertion Sort and Merge Sort depends on the specific requirements of the application. Insertion Sort is suitable for small or nearly sorted arrays due to its simplicity, in-place nature, and low overhead. Merge Sort is the superior choice for large datasets, offering guaranteed efficiency and stability at the cost of higher memory consumption. Understanding the strengths and weaknesses of each algorithm allows for informed decision-making in various sorting scenarios.

Data Structures Using C
PRACTICAL: 5

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: A) A C program to demonstrate working of Quick sort algorithm.
B) A C program to demonstrate working of Radix sort algorithm.

Theory/Concept:

Quick Sort: Quick Sort is a comparison-based sorting algorithm that employs a "divide and conquer" strategy.

Radix Sort: Radix Sort is a non-comparison-based sorting algorithm that sorts elements by processing individual digits (or characters) from least significant to most significant (LSD Radix Sort) or vice-versa (MSD Radix Sort).

Algorithm/Pseudocode:

For Quick sort :

```
FUNCTIONquickSort(array, low, high):
  IF low < high:
    pivotIndex=partition(array, low, high)
    quickSort(array,low,pivotIndex - 1)
    quickSort(array,pivotIndex + 1, high)

FUNCTIONpartition(array, low, high):
  pivot = array[high]
  i=low-1
  FORjFROMlowTOhigh - 1:
    IF array[j] <= pivot:
      i = i + 1
      SWAParray[i]ANDarray[j]
  SWAParray[i+1]ANDarray[high]
  RETURN i + 1
```

For Radix sort:

```
FUNCTION radixSort(array):
  maxVal = FIND_MAXIMUM_VALUE(array)
  exp = 1
  WHILE maxVal / exp > 0:
    countingSort(array, exp)
    exp = exp * 10

FUNCTION countingSort(array, exp):
  n= LENGTH(array)
  outputArray = NEW ARRAY OF SIZE n
  countArray = NEW ARRAY OF SIZE 10, INITIALIZED TO ZEROS

  FOR i FROM 0 TO n - 1:
    digit = (array[i] / exp) MOD 10
    countArray[digit] = countArray[digit] + 1

  FOR i FROM 1 TO 9:
    countArray[i] = countArray[i] + countArray[i - 1]

  FOR i FROM n - 1 DOWN TO 0:
    digit = (array[i] / exp) MOD 10
    outputArray[countArray[digit] - 1] = array[i]
    countArray[digit] = countArray[digit] - 1

  FOR i FROM 0 TO n - 1:
    array[i] = outputArray[i]
```

Program code:

For Quick Sort :

```c
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose the last element as pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot) {
            i++; // Increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Main Quick Sort function
```

```c
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index, arr[pi] is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array (Quick Sort): ");
    printArray(arr, n);
    return 0;
}
```

For Radix Sort:

```c
#include <stdio.h>

// Function to get the maximum value in the array
int getMax(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp) {
    int output[n]; // output array
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
```

```c
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
void radixSort(int arr[], int n) {
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    printArray(arr, n);
    radixSort(arr, n);
    printf("Sorted array (Radix Sort): ");
    printArray(arr, n);
    return 0;
}
```

Output/Results:

For Quick sort:

```
Original array: 10 7 8 9 1 5
Sorted array (Quick Sort): 1 5 7 8 9 10


=== Code Execution Successful ===
```

For Radix sort:

```
Original array: 170 45 75 90 802 24 2 66
Sorted array (Radix Sort): 2 24 45 66 75 90 170 802


=== Code Execution Successful ===
```

Relative Applications:

Quick Sort:

Imagine organizing a large physical document archive. You have many folders, each containing various papers, and you need to sort them alphabetically by folder name.
Pick a folder: You choose a folder (the "pivot") and decide where it belongs in the final sorted order.

Divide: You quickly separate the remaining folders into two piles: those that come before your chosen folder alphabetically and those that come after.

Conquer: You then repeat this process (recursively) with each of the two piles until all folders are in their correct alphabetical position.

Radix Sort:

Consider a large pile of physical punch cards, each containing a multi-digit number (like a five-digit ID). You need to sort these cards numerically.

Sort by last digit: You first sort all the cards based only on their last digit, placing them into separate bins for each digit (0-9).
Combine and sort by next digit: You then gather the cards from the bins in order (0 through 9) and repeat the sorting process, this time based on the second-to-last digit.
Repeat: You continue this process, sorting by each digit position from right to left (least significant to most significant), ensuring stability at each step.

Conclusion:

Quick Sort Conclusion: Quicksort is a highly efficient, comparison-based sorting algorithm that utilizes a divide-and-conquer strategy. Its average-case time complexity of O(n log n) makes it a popular choice for general-purpose sorting, especially for large datasets. While it offers in-place sorting and good average performance, its worst-case scenario can lead to $O(n^2)$ complexity, making careful pivot selection crucial.

Radix Sort Conclusion: Radix sort is a non-comparison-based sorting algorithm particularly effective for integers or fixed-length data. It sorts elements by processing individual digits or characters, often achieving a linear time complexity of O(nk) where 'k' is the number of digits/passes. Radix sort excels in specific scenarios where 'k' is small or the data type is well-suited for digit-by-digit processing, offering predictable and stable performance. However, it typically requires more space than in-place algorithms like Quicksort and is less versatile for varied data types.

Data Structures Using C
PRACTICAL: 6

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: A C program to demonstrate various operations on stack.

Theory/Concept:

The main operations on a stack are push (add an element to the top), pop (remove the top element), and peek or top (view the top element without removing it). Other common operations include isEmpty (check if the stack is empty) and isFull (check if the stack is full, if a size limit is imposed).

Algorithm/Pseudo code:

```
// Initialize Stack
PROCEDURE INITIALIZE_STACK(STACK, TOP, MAX_SIZE)
  SET TOP = -1 // Indicates an empty stack
END PROCEDURE

// Push Operation (Add an element to the top)
PROCEDURE PUSH(STACK, TOP, MAX_SIZE, ITEM)
  IF TOP = MAX_SIZE - 1 THEN
    DISPLAY "Stack Overflow: Cannot push element, stack is full."
  ELSE
    INCREMENT TOP
    SET STACK[TOP] = ITEM
  END IF
END PROCEDURE

// Pop Operation (Remove and return the top element)
FUNCTION POP(STACK, TOP) RETURNS ITEM
  IF TOP = -1 THEN
    DISPLAY "Stack Underflow: Cannot pop from an empty stack."
    RETURN NULL // Or handle error appropriately
  ELSE
    SET ITEM_TO_RETURN = STACK[TOP]
    DECREMENT TOP
    RETURN ITEM_TO_RETURN
```

```
   END IF
ENDFUNCTION

//Peek Operation (Return the top element without removing it)
FUNCTION PEEK(STACK, TOP) RETURNS ITEM
   IFTOP = -1 THEN
      DISPLAY "Stack is empty: No element to peek."
      RETURN NULL // Or handle error appropriately
   ELSE
      RETURN STACK[TOP]
   END IF
ENDFUNCTION

//IsEmpty Operation (Check if the stack is empty)
FUNCTION IS_EMPTY(TOP) RETURNS BOOLEAN
   IFTOP = -1 THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   END IF
ENDFUNCTION

//IsFull Operation (Check if the stack is full)
FUNCTION IS_FULL(TOP, MAX_SIZE) RETURNS BOOLEAN
   IFTOP = MAX_SIZE - 1 THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   END IF
ENDFUNCTION
```

Program code:

```c
#include <stdio.h>
#include <stdlib.h> // For exit()

#define MAX_SIZE 5 // Define the maximum size of the stack

int stack[MAX_SIZE]; // Array to store stack elements
int top = -1;        // Index of the top element, -1 indicates an empty stack

// Function to check if the stack is empty
int isEmpty() {
    return (top == -1);
}

// Function to check if the stack is full
int isFull() {
    return (top == MAX_SIZE - 1);
}

// Function to push an element onto the stack
void push(int value) {
    if (isFull()) {
        printf("Stack Overflow: Cannot push %d, stack is full.\n", value);
    } else {
        top++;
        stack[top] = value;
        printf("Pushed %d onto the stack.\n", value);
    }
}

// Function to pop an element from the stack
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow: Cannot pop, stack is empty.\n");
        return -1; // Indicate an error or empty stack
```

```c
    } else {
        int poppedValue = stack[top];
        top--;
        printf("Popped %d from the stack.\n", poppedValue);
        return poppedValue;
    }
}

// Function to peek at the top element of the stack without removing it
int peek() {
    if (isEmpty()) {
        printf("Stack is empty, no element to peek.\n");
        return -1; // Indicate an error or empty stack
    } else {
        return stack[top];
    }
}

// Function to display the elements of the stack
void display() {
    if (isEmpty()) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements (top to bottom):\n");
        for (int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}

int main() {
    int choice, value;

    while (1) {
```

```c
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                value = peek();
                if (value != -1) {
                    printf("Top element: %d\n", value);
                }
                break;
            case 4:
                display();
                break;
            case 5:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Output/Results:

```
Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 2
Pushed 2 onto the stack.
```

Relative Applications:

A common real-life example illustrating stack operations is a stack of plates in a cafeteria.

Push Operation:

When clean plates are brought out and added to the pile, this represents a push operation. Each new plate is placed on top of the previous one, increasing the size of the stack.

Pop Operation:

When a diner takes a plate, they always take the one from the very top of the pile. This action corresponds to a pop operation, where the last item added to the stack is the first one removed. The size of the stack decreases.

LIFO Principle:

This scenario clearly demonstrates the Last-In-First-Out (LIFO) principle, which is fundamental to stacks. The plate that was placed on the stack most recently is the first one to be taken off.

Conclusion:

Stack operations, fundamentally based on the Last-In, First-Out (LIFO) principle, provide a structured and efficient method for managing data in various computational scenarios. The core operations, primarily push (for insertion) and pop (for deletion), along with auxiliary functions like peek (to view the top element), isEmpty, and isFull, enable controlled manipulation of data.

Data Structures Using C
PRACTICAL: 7

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: A) A C program to demonstrate various operations on queue.
B) A C program to demonstrate various operations on circular queue.

Theory/Concept:

Operations on a Queue:

A queue is a linear data structure that follows the First-In, First-Out (FIFO) principle. This means the element added first is the first one to be removed.

Operations on a Circular Queue:

A circular queue is an enhanced version of a linear queue where the last element connects back to the first element, forming a circle. This allows for better memory utilization by reusing empty spaces created by dequeued elements.

Algorithm/Pseudo code:

For Queue:

```
// Initialize Queue
PROCEDURE InitializeQueue(queue, maxSize)
  SET queue.array TO new array of size maxSize
  SET queue.front TO -1
  SET queue.rear TO -1
  SET queue.size TO 0
END PROCEDURE

// Enqueue (Insert element)
PROCEDURE Enqueue(queue, item)
  IF queue.size == queue.maxSize THEN
    PRINT "Queue is full (Overflow)"
    RETURN
  END IF
  IF queue.front == -1 THEN // Queue is empty
```

```
    SETqueue.front TO 0
  END IF

  SETqueue.rear TO queue.rear + 1
  SETqueue.array[queue.rear] TO item
  SETqueue.size TO queue.size + 1
ENDPROCEDURE

//Dequeue (Remove element)
PROCEDURE Dequeue(queue)
  IFqueue.size == 0 THEN
    PRINT "Queue is empty (Underflow)"
    RETURN NULL
  END IF
  SETitem TO queue.array[queue.front]
  SETqueue.front TO queue.front + 1
  SETqueue.size TO queue.size - 1

  IFqueue.size == 0 THEN // Last element removed, reset queue
    SETqueue.front TO -1
    SETqueue.rear TO -1
  END IF
  RETURN item
ENDPROCEDURE

//Peek(Get front element without removing)
PROCEDURE Peek(queue)
  IFqueue.size == 0 THEN
    PRINT "Queue is empty"
    RETURN NULL
  END IF
  RETURN queue.array[queue.front]
ENDPROCEDURE
//IsEmpty
PROCEDURE IsEmpty(queue)
RETURN queue.size == 0

ENDPROCEDURE

// IsFull
PROCEDURE IsFull(queue)
  RETURN queue.size == queue.maxSize
ENDPROCEDURE
```

For Circular queue:

```
// Initialize Circular Queue
PROCEDURE InitializeCircularQueue(queue, maxSize)
  SET queue.array TO new array of size maxSize
  SET queue.front TO -1
  SET queue.rear TO -1
  SET queue.maxSize TO maxSize
END PROCEDURE

// Enqueue (Insert element)
PROCEDURE EnqueueCircular(queue, item)
  IF (queue.rear + 1) MOD queue.maxSize == queue.front THEN
    PRINT "Circular Queue is full (Overflow)"
    RETURN
  END IF

  IF queue.front == -1 THEN // Queue is empty
    SET queue.front TO 0
  END IF

  SET queue.rear TO (queue.rear + 1) MOD queue.maxSize
  SET queue.array[queue.rear] TO item
END PROCEDURE

// Dequeue (Remove element)
PROCEDURE DequeueCircular(queue)
  IF queue.front == -1 THEN
    PRINT "Circular Queue is empty (Underflow)"
    RETURN NULL
  END IF

  SET item TO queue.array[queue.front]

  IF queue.front == queue.rear THEN // Only one element in queue
    SET queue.front TO -1
    SET queue.rear TO -1
  ELSE
    SET queue.front TO (queue.front + 1) MOD queue.maxSize
  END IF
  RETURN item
END PROCEDURE

// Peek (Get front element without removing)
PROCEDURE PeekCircular(queue)
  IF queue.front == -1 THEN
    PRINT "Circular Queue is empty"
    RETURN NULL
```

```
    END IF
    RETURN queue.array[queue.front]
END PROCEDURE

//IsEmpty
PROCEDURE IsEmptyCircular(queue)
    RETURN queue.front == -1
END PROCEDURE

//IsFull
PROCEDURE IsFullCircular(queue)
    RETURN (queue.rear + 1) MOD queue.maxSize == queue.front
END PROCEDURE
```

Program code:

For Queue:

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

#define MAX_SIZE 5 // Define the maximum size of the queue

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

// Function to check if the queue is full
int isFull() {
    return rear == MAX_SIZE - 1;
}

// Function to check if the queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Function to enqueue (insert) an element
void enqueue(int data) {
    if (isFull()) {
        printf("Queue overflow\n");
        return;
    }
    if (front == -1) {
        front = 0;
    }
    rear++;
    queue[rear] = data;
    printf("Element %d inserted\n", data);
}

// Function to dequeue (remove) an element
int dequeue() {
    if (isEmpty()) {
        printf("Queue underflow\n");
        return -1; // Indicate an error
```

```c
    }
    int data = queue[front];
    front++;
    if (front > rear) { // Reset queue if it becomes empty
        front = -1;
        rear = -1;
    }
    return data;
}

// Function to display the queue elements
void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    printf("Dequeued element: %d\n", dequeue());
    display();
    enqueue(40);
    enqueue(50);
    enqueue(60); // This should show "Queue overflow"
    display();
    return 0;
}
```

For Circular Queue:

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

#define MAX_SIZE 5 // Define the maximum size of the circular queue

int c_queue[MAX_SIZE];
int c_front = -1, c_rear = -1;

// Function to check if the circular queue is full
int c_isFull() {
    return (c_rear + 1) % MAX_SIZE == c_front;
}

// Function to check if the circular queue is empty
int c_isEmpty() {
    return c_front == -1;
}

// Function to enqueue (insert) an element in circular queue
void c_enqueue(int data) {
    if (c_isFull()) {
        printf("Circular Queue overflow\n");
        return;
    }
    if (c_front == -1) {
        c_front = 0;
    }
    c_rear = (c_rear + 1) % MAX_SIZE;
    c_queue[c_rear] = data;
    printf("Element %d inserted into Circular Queue\n", data);
}

// Function to dequeue (remove) an element from circular queue
int c_dequeue() {
    if (c_isEmpty()) {
        printf("Circular Queue underflow\n");
        return -1; // Indicate an error
    }
```

```
    int data = c_queue[c_front];
    if (c_front == c_rear) { // Reset queue if it becomes empty
        c_front = -1;
        c_rear = -1;
    } else {
        c_front = (c_front + 1) % MAX_SIZE;
    }
    return data;
}

// Function to display the circular queue elements
void c_display() {
    if (c_isEmpty()) {
        printf("Circular Queue is empty\n");
        return;
    }
    printf("Circular Queue elements: ");
    int i = c_front;
    while (i != c_rear) {
        printf("%d ", c_queue[i]);
        i = (i + 1) % MAX_SIZE;
    }
    printf("%d\n", c_queue[c_rear]); // Print the last element
}

int main() {
    c_enqueue(10);
    c_enqueue(20);
    c_enqueue(30);
    c_display();
    printf("Dequeued element from Circular Queue: %d\n", c_dequeue());
    c_display();
    c_enqueue(40);
    c_enqueue(50);
    c_enqueue(60); // This should show "Circular Queue overflow"
    c_display();
    return 0;
}
```

Output/Results:

For Queue:

```
Element 10 inserted
Element 20 inserted
Element 30 inserted
Queue elements: 10 20 30
Dequeued element: 10
Queue elements: 20 30
Element 40 inserted
Element 50 inserted
Queue overflow
Queue elements: 20 30 40 50


=== Code Execution Successful ===
```

For Circular Queue:

```
Element 10 inserted into Circular Queue
Element 20 inserted into Circular Queue
Element 30 inserted into Circular Queue
Circular Queue elements: 10 20 30
Dequeued element from Circular Queue: 10
Circular Queue elements: 20 30
Element 40 inserted into Circular Queue
Element 50 inserted into Circular Queue
Element 60 inserted into Circular Queue
Circular Queue elements: 20 30 40 50 60


=== Code Execution Successful ===
```

Relative Applications: Queue: A real-life example of a queue is a line of customers waiting at a checkout counter in a supermarket. The first person in line is the first to be served, and new customers join the end of the line. This illustrates the "First-In, First-Out" (FIFO) principle of a queue. Circular Queue: A real-life example of a circular queue is a traffic light system. The sequence of red, yellow, and green lights cycles continuously. Once the green light finishes, it eventually cycles back to red, reusing the same sequence of states. This demonstrates how a circular queue reuses available space by wrapping around from the end to the beginning when the end is reached.

Conclusion: A standard queue is simple and follows the FIFO principle but is inefficient in space because it cannot reuse empty slots at the front. A circular queue is more memory-efficient because it connects the rear to the front, forming a circle to reuse empty space, making it ideal for applications needing continuous processing like buffering or round-robin scheduling.

Data Structures Using C PRACTICAL: 8

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: **A) A C program to demonstrate various operations on Single linked list.**
B) A C program to demonstrate various operations on double linked list.
C) A C program to demonstrate various operations on circular linked list.

Theory/Concept:

Single linked list
Concept: Each node contains data and a pointer to the next node in the sequence. The last node's next pointer is NULL.
Traversal: Can only move in one direction, from head to tail.
Operations: Supports traversal, insertion, and deletion, but modifying a node requires traversing from the head.

Double linked list

Concept: Eachnode has two pointers: one to the next node and one to the previous node.
Traversal: Canmove both forward and backward.
Operations: Insertion and deletion can be more efficient for certain operations (like deleting from the end) becauseyouhave access to the previous node.

Circular linked list

Concept:
The lastnode'spointer points back to the first node, creating a loop. It can be a singly or doubly circular list.
Traversal:
Can be traversedindefinitely by following the pointers, with no NULL terminator. You can start from any node andloopback to it.
Operations:
Operationslikeinsertion and deletion require special handling to maintain the circular structure, particularlybyupdating the last node's pointer.

Algorithm/Pseudo code:

Single linked list:

```
//Node structure for a Single Linked List
STRUCT Node:
    DATA value
    POINTER next // Points to the next node

//Basic operations
FUNCTION InsertAtBeginning(HEAD, newValue):
    newNode = CREATE Node
    newNode.value = newValue
    newNode.next = HEAD
    HEAD = newNode
    RETURN HEAD

FUNCTION InsertAtEnd(HEAD, newValue):
    newNode = CREATE Node
    newNode.value = newValue
    newNode.next = NULL
    IFHEAD IS NULL:
        HEAD = newNode
        RETURN HEAD
    currentNode = HEAD
    WHILE currentNode.next IS NOT NULL:
        currentNode = currentNode.next
    currentNode.next = newNode
    RETURN HEAD

FUNCTION DeleteNode(HEAD, valueToDelete):
    IFHEAD IS NULL:
        RETURN NULL
    IFHEAD.value == valueToDelete:
        HEAD = HEAD.next
        RETURN HEAD
    currentNode = HEAD
    WHILE currentNode.next IS NOT NULL AND currentNode.next.value IS NOT valueToDelete:
        currentNode = currentNode.next
    IFcurrentNode.next IS NOT NULL:
        currentNode.next = currentNode.next.next
    RETURN HEAD

FUNCTION TraverseList(HEAD):
    currentNode = HEAD
    WHILE currentNode IS NOT NULL:
        PRINT currentNode.value
        currentNode = currentNode.next
```

Double Linked list:

```
// Node structure for a Double Linked List
STRUCT Node:
    DATA value
    POINTER prev // Points to the previous node
    POINTER next // Points to the next node

// Basic operations
FUNCTION InsertAtBeginning(HEAD, newValue):
    newNode = CREATE Node
    newNode.value = newValue
    newNode.prev = NULL
    newNode.next = HEAD
    IF HEAD IS NOT NULL:
        HEAD.prev = newNode
    HEAD = newNode
    RETURN HEAD

FUNCTION InsertAtEnd(HEAD, newValue):
    newNode = CREATE Node
    newNode.value = newValue
    newNode.next = NULL
    IF HEAD IS NULL:
        newNode.prev = NULL
        HEAD = newNode
        RETURN HEAD
    currentNode = HEAD
    WHILE currentNode.next IS NOT NULL:
        currentNode = currentNode.next
    currentNode.next = newNode
    newNode.prev = currentNode
    RETURN HEAD

FUNCTION DeleteNode(HEAD, valueToDelete):
    IF HEAD IS NULL:
        RETURN NULL
    IF HEAD.value == valueToDelete:
        HEAD = HEAD.next
        IF HEAD IS NOT NULL:
            HEAD.prev = NULL
        RETURN HEAD
    currentNode = HEAD
    WHILE currentNode IS NOT NULL AND currentNode.value IS NOT valueToDelete:
        currentNode = currentNode.next
    IF currentNode IS NOT NULL:
        IF currentNode.prev IS NOT NULL:
            currentNode.prev.next = currentNode.next
        IF currentNode.next IS NOT NULL:
            currentNode.next.prev = currentNode.prev
```

```
    RETURN HEAD

FUNCTION TraverseListForward(HEAD):
    currentNode = HEAD
    WHILE currentNode IS NOT NULL:
        PRINT currentNode.value
        currentNode = currentNode.next

FUNCTION TraverseListBackward(TAIL): // Assuming TAIL pointer is maintained
    currentNode = TAIL
    WHILE currentNode IS NOT NULL:
        PRINT currentNode.value
        currentNode = currentNode.prev
```

Circular Linked list:

```
// Node structure for a Circular Linked List (Singly)
STRUCT Node:
    DATA value
    POINTER next // Points to the next node

// Basic operations
FUNCTION InsertAtBeginning(HEAD, newValue):
    newNode = CREATE Node
    newNode.value = newValue
    IF HEAD IS NULL:
        HEAD = newNode
        newNode.next = HEAD // Points to itself
    ELSE:
        currentNode = HEAD
        WHILE currentNode.next IS NOT HEAD:
            currentNode = currentNode.next
        newNode.next = HEAD
        currentNode.next = newNode
        HEAD = newNode
    RETURN HEAD

FUNCTION InsertAtEnd(HEAD, newValue):
    newNode = CREATE Node
    newNode.value = newValue
    IF HEAD IS NULL:
        HEAD = newNode
        newNode.next = HEAD // Points to itself
    ELSE:
        currentNode = HEAD
        WHILE currentNode.next IS NOT HEAD:
            currentNode = currentNode.next
        currentNode.next = newNode
```

```
    newNode.next = HEAD
  RETURN HEAD

FUNCTION DeleteNode(HEAD, valueToDelete):
  IFHEAD IS NULL:
    RETURN NULL
  IFHEAD.value == valueToDelete:
    IFHEAD.next == HEAD: // Only one node in the list
      HEAD = NULL
      RETURN NULL
    currentNode = HEAD
    WHILE currentNode.next IS NOT HEAD:
      currentNode = currentNode.next
    currentNode.next = HEAD.next
    HEAD = HEAD.next
    RETURN HEAD
  currentNode = HEAD
  WHILE currentNode.next IS NOT HEAD AND currentNode.next.value IS NOT valueToDelete:
    currentNode = currentNode.next
  IFcurrentNode.next IS NOT HEAD:
    currentNode.next = currentNode.next.next
  RETURN HEAD

FUNCTION TraverseList(HEAD):
  IFHEAD IS NULL:
    RETURN
  currentNode = HEAD
  REPEAT:
    PRINT currentNode.value
    currentNode = currentNode.next
  UNTIL currentNode IS HEAD
```

Program code:

Single linked list:

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

// Node structure for a singly linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
```

```
26      *head = newNode;
27  }
28
29  // Function to print the singly linked list
30  void printSinglyLinkedList(struct Node* head) {
31      struct Node* current = head;
32      while (current != NULL) {
33          printf("%d -> ", current->data);
34          current = current->next;
35      }
36      printf("NULL\n");
37  }
38
39  int main() {
40      struct Node* head = NULL;
41
42      insertAtBeginning(&head, 3);
43      insertAtBeginning(&head, 2);
44      insertAtBeginning(&head, 1);
45
46      printf("Singly Linked List: ");
47      printSinglyLinkedList(head);
48
49      return 0;
50  }
```

Double linked list:

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

// Node structure for a doubly linked list
struct DoublyNode {
    int data;
    struct DoublyNode* prev;
    struct DoublyNode* next;
};

// Function to create a new doubly node
struct DoublyNode* createDoublyNode(int data) {
    struct DoublyNode* newNode = (struct DoublyNode*)malloc(sizeof(struct
        DoublyNode));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the doubly linked list
void insertAtBeginningDoubly(struct DoublyNode** head, int data) {
    struct DoublyNode* newNode = createDoublyNode(data);
    newNode->next = *head;
```

```c
    if (*head != NULL) {
        (*head)->prev = newNode;
    }
    *head = newNode;
}

// Function to print the doubly linked list (forward)
void printDoublyLinkedListForward(struct DoublyNode* head) {
    struct DoublyNode* current = head;
    while (current != NULL) {
        printf("%d <-> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct DoublyNode* head = NULL;

    insertAtBeginningDoubly(&head, 30);
    insertAtBeginningDoubly(&head, 20);
    insertAtBeginningDoubly(&head, 10);

    printf("Doubly Linked List (Forward): ");
    printDoublyLinkedListForward(head);

    return 0;
}
```

Circular linked list:

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

// Node structure for a circular linked list
struct CircularNode {
    int data;
    struct CircularNode* next;
};

// Function to create a new circular node
struct CircularNode* createCircularNode(int data) {
    struct CircularNode* newNode = (struct CircularNode*)malloc(sizeof(struct Circular
        ));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL; // Will be set to point to itself or another node later
    return newNode;
}

// Function to insert a node at the end of a circular linked list
void insertAtEndCircular(struct CircularNode** head, int data) {
    struct CircularNode* newNode = createCircularNode(data);
    if (*head == NULL) {
        *head = newNode;
        newNode->next = *head; // Points to itself
    } else {
        struct CircularNode* current = *head;
        while (current->next != *head) {
```

```
            current = current->next;
        }
        current->next = newNode;
        newNode->next = *head;
    }
}

// Function to print the circular linked list
void printCircularLinkedList(struct CircularNode* head) {
    if (head == NULL) {
        printf("Circular Linked List is empty.\n");
        return;
    }
    struct CircularNode* current = head;
    do {
        printf("%d -> ", current->data);
        current = current->next;
    } while (current != head);
    printf("(Head)\n"); // Indicates the cycle
}

int main() {
    struct CircularNode* head = NULL;

    insertAtEndCircular(&head, 100);
    insertAtEndCircular(&head, 200);
    insertAtEndCircular(&head, 300);

    printf("Circular Linked List: ");
    printCircularLinkedList(head);

    return 0;
}
```

Output/Results:

Single linked list:

```
Singly Linked List: 1 -> 2 -> 3 -> NULL


=== Code Execution Successful ===
```

Double linked list:

```
Doubly Linked List (Forward): 10 <-> 20 <-> 30 <-> NULL


=== Code Execution Successful ===
```

Circular linked list:

```
Circular Linked List: 100 -> 200 -> 300 -> (Head)


=== Code Execution Successful ===
```

Relative Applications:

Singly Linked List:
Example: A simple playlist of songs in a music player where you can only advance to the next song. Each song "node" points to the subsequent song, but there's no direct way to go back to a previous song without restarting the list from the beginning.

Doubly Linked List:

Example: The undo/redo functionality in a text editor. Each change to the document is a "node," and you can easily move forward (redo) or backward (undo) through the history of changes because each node points to both the next and the previous change.

Circular Linked List:

Example: A multiplayer game where players take turns in a fixed order. Each player is a "node," and after the last player takes their turn, the turn cycles back to the first player, forming a continuous loop without a defined end.

Conclusion:

Singly, doubly, and circular linked lists are all linear data structures, but they differ in how nodes are connected. A singly linked list has one pointer to the next node, while a doubly linked list has both a next and a previous pointer, allowing backward traversal. A circular linked list (which can be singly or doubly) forms a loop by having the last node point back to the first node.

Data Structures Using C
PRACTICAL: 9

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: **A) A C program to demonstrate various operations on Tree data structure.**
B) A C program to demonstrate tree traversal methods.

Theory/Concept:

Operations on Tree Data Structures

Operations on tree data structures include fundamental actions to manage the tree's content:

- **Insertion:** Adding a new node to the tree while maintaining its structural properties (e.g., in a Binary Search Tree, maintaining sorted order).
- **Deletion:** Removing a node from the tree and reorganizing the remaining nodes to preserve the tree's integrity.
- **Searching:** Locating a specific node or value within the tree.
- **Updating:** Modifying the value of an existing node.

Tree Traversal Methods

Tree traversal refers to the process of visiting each node in a tree exactly once in a systematic order. The primary methods are:

1. Depth-First Traversal (DFT): Explores as far as possible along a branch before backtracking.

- **Pre-order Traversal:**

Visit the root, then the left subtree, then the right subtree (Root-Left-Right). Useful for creating a copy of the tree.

- **In-order Traversal:**

Visit the left subtree, then the root, then the right subtree (Left-Root-Right). Commonly used in Binary Search Trees to retrieve elements in sorted order.

- **Post-order Traversal:**

Visit the left subtree, then the right subtree, then the root (Left-Right-Root). Useful for deleting nodes or evaluating expressions.

Algorithm/Pseudo code:

Tree data structure:

```
CLASS Node
   PROPERTY data : ANY_TYPE
   PROPERTY left_child : Node // For binary trees, or an array/list of children for general trees
   PROPERTY right_child : Node // For binary trees only

   // Constructor
   FUNCTION Node(value)
      SET data = value
      SET left_child = NULL
      SET right_child = NULL
   END FUNCTION
END CLASS
```

Tree traversal methods:

1.Pre-order Traversal (Root, Left, Right)

```
FUNCTION preOrderTraversal(node)
   IF node IS NOT NULL THEN
      PRINT node.data // Visit the root
      preOrderTraversal(node.left_child) // Traverse left subtree
      preOrderTraversal(node.right_child) // Traverse right subtree
   END IF
END FUNCTION
```

2.In-order Traversal (Left, Root, Right)

```
FUNCTION inOrderTraversal(node)
   IF node IS NOT NULL THEN
      inOrderTraversal(node.left_child) // Traverse left subtree
      PRINT node.data // Visit the root
      inOrderTraversal(node.right_child) // Traverse right subtree
   END IF
END FUNCTION
```

3.Post-order Traversal (Left, Right, Root)

```
FUNCTION postOrderTraversal(node)
   IF node IS NOT NULL THEN
      postOrderTraversal(node.left_child) // Traverse left subtree
      postOrderTraversal(node.right_child) // Traverse right subtree
      PRINT node.data // Visit the root
   END IF
END FUNCTION
```

Program code:
For Tree Data Structure:

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

// Define the structure for a tree node
typedef struct Node {
    int data;
    struct Node *left;
    struct Node *right;
} Node;

// Function to create a new tree node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        perror("Failed to allocate memory for new node");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node into a binary search tree (without explicit traversal)
// This function implicitly navigates to the correct insertion point.
Node* insert(Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }

```

```c
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    // If data is equal, we typically do nothing for a simple BST
    return root;
}

// Main function to demonstrate tree creation and insertion
int main() {
    Node* root = NULL;

    // Insert nodes into the tree
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    printf("Tree constructed successfully without explicit traversal.\n");
    printf("Root node data: %d\n", root->data);
    printf("Left child of root: %d\n", root->left->data);
    printf("Right child of root: %d\n", root->right->data);

    // Free allocated memory (a proper free function would involve traversal)
    // For this example, we're omitting a full free function as it would require traversal.
    // In a real application, you would need to free all nodes to prevent memory leaks.

    return 0;
}
```

For Tree traversal method:

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

// Structure for a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Inorder Traversal (Left -> Root -> Right)
void inorderTraversal(struct Node* root) {
    if (root == NULL)
        return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}

// Preorder Traversal (Root -> Left -> Right)
void preorderTraversal(struct Node* root) {
    if (root == NULL)
        return;
    printf("%d ", root->data);
    preorderTraversal(root->left);
```

```c
    preorderTraversal(root->right);
}

// Postorder Traversal (Left -> Right -> Root)
void postorderTraversal(struct Node* root) {
    if (root == NULL)
        return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
}

int main() {
    // Creating a sample binary tree
    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: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");

    // Freeing allocated memory (important for preventing memory leaks)
    free(root->left->left);
    free(root->left->right);
    free(root->left);
    free(root->right);
    free(root);

    return 0;
}
```

Output/Results:

For Tree data structure:

```
Tree constructed successfully without explicit traversal.
Root node data: 50
Left child of root: 30
Right child of root: 70


=== Code Execution Successful ===
```

For Tree traversal methods:

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


=== Code Execution Successful ===
```

Relative Applications: Tree Data Structure Example (File System): The root of the tree is the main drive (e.g., `C:\` on
Windows or / on Unix-like systems). Directories (folders) are parent nodes, and the files and subdirectories within them are child nodes. This forms a hierarchical structure, where each file or directory has a unique path from the root.

Tree Traversal Example (Finding a File): When you search for a specific file on your computer, the operating system effectively performs a tree traversal.

Depth-First Traversal (Preorder): Imagine a search that systematically explores each directory and its contents fully before moving to the next sibling directory. This is similar to a preorder traversal, where you visit the current directory (node) and then recursively explore its subdirectories (children).

Conclusion:

Atree is a hierarchical, non-linear data structure that efficiently organizes and stores data with parent-child relationships, making it suitable for representing hierarchical information like file systems or organizational charts.

Tree traversal refers to the process of visiting each node in a tree exactly once in a systematic order. Unlike linear data structures, trees offer multiple traversal methods.

Data Structures Using C
PRACTICAL: 10

Name: BALWANT MUNDHE
PRN: 20250802296
Batch: B3

AIM: **A) A C program to demonstrate various operations on graph data structure.**
B) A C program to demonstrate DFS graph traversal.
C) A C program to demonstrate BFS graph traversal.

Theory/Concept:

Operations on Graph Data Structure

A graph data structure, composed of vertices (nodes) and edges, supports various operations:

- **Adding/Removing Vertices:** Inserting or deleting individual nodes from the graph.
- **Adding/Removing Edges:** Establishing or breaking connections between two existing vertices.

Depth-First Search (DFS):

- **Concept:** Explores as far as possible along each branch before backtracking. It goes "deep" into the graph.
- **Mechanism:** Starts at a chosen vertex, visits it, and then recursively visits an unvisited adjacent vertex. This continues until no unvisited adjacent vertices are found, at which point it backtracks to the previous vertex and explores other branches.
- **Implementation:** Typically uses a stack (explicitly or implicitly via recursion).

Breadth-First Search (BFS):

- **Concept:** Explores the graph layer by layer, visiting all neighbors at the current level before moving to the next level. It goes "wide" in the graph.
- **Mechanism:** Starts at a chosen vertex, visits it, then visits all its immediate neighbors, then all their unvisited neighbors, and so on.
- **Implementation:** Typically uses a queue.

Algorithm/Pseudo code:

For Graph data structure:

```
GRAPH:
  VERTICES:Asetofnodesinthegraph.
  EDGES:Asetofconnectionsbetween vertices.
      Represented as adjacency list:
       ADJACENCY_LIST:Adictionarywhere keys are vertices and values are lists of adjacent vertices.
      Or as adjacency matrix:
       ADJACENCY_MATRIX:A2Darray where MATRIX[i][j] is 1 if an edge exists between vertex i and
vertex j, 0 otherwise.
```

For DFS traversal:

```
FUNCTION DFS(graph, start_node):
  visited_nodes = SET()
  stack = STACK()

  PUSH start_node onto stack
  ADD start_node to visited_nodes

  WHILE stack IS NOT EMPTY:
    current_node = POP from stack

    FOREACHneighborOFcurrent_node IN graph.ADJACENCY_LIST:
      IFneighborISNOTINvisited_nodes:
        ADD neighbor to visited_nodes
        PUSH neighbor onto stack
```

For BFS traversal:

```
FUNCTION BFS(graph, start_node):
  visited_nodes = SET()
  queue = QUEUE()

  ENQUEUE start_node into queue
  ADD start_node to visited_nodes

  WHILE queue IS NOT EMPTY:
    current_node=DEQUEUEfromqueue

    FOREACHneighborOFcurrent_node IN graph.ADJACENCY_LIST:
      IFneighborISNOTINvisited_nodes:
        ADD neighbor to visited_nodes
        ENQUEUE neighbor into queue
```

Program code:

For Graph data structure:

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

// Structure to represent a graph
struct Graph {
    int numVertices; // Number of vertices in the graph
    int** adjMatrix; // Adjacency matrix to store connections
};

// Function to create a new graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    // Allocate memory for the adjacency matrix
    graph->adjMatrix = (int**) malloc(vertices * sizeof(int*));
    for (int i = 0; i < vertices; i++) {
        graph->adjMatrix[i] = (int*) malloc(vertices * sizeof(int));
    }

    // Initialize all entries in the adjacency matrix to 0 (no edges initially)
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            graph->adjMatrix[i][j] = 0;
        }
    }
    return graph;
}

// Function to add an edge between two vertices
void addEdge(struct Graph* graph, int src, int dest) {
    // For an undirected graph, add edges in both directions
    graph->adjMatrix[src][dest] = 1;
    graph->adjMatrix[dest][src] = 1;
}

// Function to print the adjacency matrix of the graph
```

```c
void printGraph(struct Graph* graph) {
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < graph->numVertices; i++) {
        for (int j = 0; j < graph->numVertices; j++) {
            printf("%d ", graph->adjMatrix[i][j]);
        }
        printf("\n");
    }
}

// Function to free the memory allocated for the graph
void freeGraph(struct Graph* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        free(graph->adjMatrix[i]);
    }
    free(graph->adjMatrix);
    free(graph);
}

int main() {
    int numVertices = 5;
    struct Graph* graph = createGraph(numVertices);

    // Add some edges
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    printGraph(graph);

    freeGraph(graph); // Free allocated memory
    return 0;
}
```

For DFS graph traversal:

```c
#include <stdio.h>
#include <stdbool.h> // For using boolean type

#define MAX_VERTICES 20

int graph[MAX_VERTICES][MAX_VERTICES];
bool visited[MAX_VERTICES];
int numVertices;

// Function to perform DFS traversal
void dfs(int vertex) {
    printf("%d ", vertex); // Print the current vertex
    visited[vertex] = true; // Mark the current vertex as visited

    // Iterate through all adjacent vertices
    for (int i = 0; i < numVertices; i++) {
        // If there's an edge and the adjacent vertex hasn't been visited
        if (graph[vertex][i] == 1 && !visited[i]) {
            dfs(i); // Recursively call DFS for the unvisited adjacent vertex
        }
    }
}

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

    printf("Enter the number of vertices (max %d): ", MAX_VERTICES);
    scanf("%d", &numVertices);

    // Initialize visited array and adjacency matrix
    for (i = 0; i < numVertices; i++) {
        visited[i] = false;
        for (j = 0; j < numVertices; j++) {
            graph[i][j] = 0;
        }
    }

    printf("Enter the adjacency matrix (0 or 1 for no/yes edge):\n");
    for (i = 0; i < numVertices; i++) {
        for (j = 0; j < numVertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter the starting vertex for DFS (0 to %d): ", numVertices - 1);
    scanf("%d", &startVertex);

    printf("DFS Traversal: ");
    dfs(startVertex);
    printf("\n");

    return 0;
}
```

For BFS graph traversal:

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

#define MAX_VERTICES 100

// Function to add an edge to the graph (for an undirected graph)
void addEdge(int graph[MAX_VERTICES][MAX_VERTICES], int start, int end) {
    graph[start][end] = 1;
    graph[end][start] = 1; // For undirected graph
}

// Function to perform BFS traversal
void BFS(int graph[MAX_VERTICES][MAX_VERTICES], int vertices, int startVertex) {
    int visited[MAX_VERTICES] = {0}; // Initialize all vertices as not visited
    int queue[MAX_VERTICES];
    int front = -1, rear = -1;

    // Mark the startVertex as visited and enqueue it
    visited[startVertex] = 1;
    queue[++rear] = startVertex;

    printf("BFS Traversal Order: ");

    while (front != rear) {
        int currentVertex = queue[++front];
        printf("%d ", currentVertex);

        // Explore all adjacent vertices of the currentVertex
        for (int i = 0; i < vertices; i++) {
            if (graph[currentVertex][i] == 1 && !visited[i]) {
```

```
                visited[i] = 1;
                queue[++rear] = i;
            }
        }
    }
    printf("\n");
}

int main() {
    int vertices, edges;

    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES] = {0}; // Initialize adjacency matrix with zeros

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

    if (edges < 0 || edges > vertices * (vertices - 1) / 2) {
        printf("Invalid number of edges. Exiting...\n");
        return 1;
    }

    printf("Input the edges (start_vertex end_vertex):\n");
    for (int i = 0; i < edges; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        if (u < 0 || u >= vertices || v < 0 || v >= vertices) {
            printf("Invalid vertex in edge. Skipping...\n");
            continue;
        }
        addEdge(graph, u, v);
    }

    int startNode;
    printf("Enter the starting node for BFS: ");
    scanf("%d", &startNode);

    if (startNode < 0 || startNode >= vertices) {
        printf("Invalid starting node. Exiting...\n");
        return 1;
    }

    BFS(graph, vertices, startNode);

    return 0;
}
```

Output/Results:

For Graph data structure:

```
Adjacency Matrix:
0 1 0 0 1
1 0 1 1 1
0 1 0 1 0
0 1 1 0 1
1 1 0 1 0


=== Code Execution Successful ===
```

For DFS graph traversal:

```
Enter the number of vertices (max 20): 17
Enter the adjacency matrix (0 or 1 for no/yes edge):
yes
Enter the starting vertex for DFS (0 to 16): DFS Traversal: 0


=== Code Execution Successful ===
```

For BFS graph traversal:

```
Input the number of vertices: 12
Input the number of edges: 2
Input the edges (start_vertex end_vertex):
0
0
12
3
Invalid vertex in edge. Skipping...
Enter the starting node for BFS: 2
BFS Traversal Order: 2


=== Code Execution Successful ===
```

Relative Applications:

Graph Data Structures:

Social Networks:

Representing users as nodes and connections as edges (e.g., Facebook friend networks, LinkedIn professional connections).

Navigation Systems:

Modeling road networks, public transport routes, or flight paths (e.g., Google Maps, GPS).

Depth-First Search (DFS): A graph traversal algorithm with various real-life applications.

Pathfinding and Maze Solving:

DFS can find a path between two points in a graph, making it suitable for solving mazes or finding routes in navigation systems.

Cycle Detection:

It can detect cycles in graphs, useful in network analysis to identify feedback loops or circular dependencies.

BFS is used for:

Social networks:

Can find all friends or connections within a certain distance (k-levels) of a person.

Network broadcasting:

Used to send a message to all nodes in a network by spreading it level by level.

Conclusion:

Graph data structures provide a powerful framework for modeling interconnected entities and their relationships, finding applications in diverse fields from social networks to routing algorithms. Depth-First Search (DFS) and Breadth-First Search (BFS) are fundamental algorithms for traversing these graphs, each offering distinct advantages based on the problem's requirements.

BFS systematically explores the graph level by level, utilizing a queue to process nodes in the order they are discovered. This approach guarantees finding the shortest path in unweighted graphs and is suitable for scenarios where the goal is likely to be close to the starting point or when exploring all nodes at a given "distance" from the start is important.

DFS, in contrast, delves as deeply as possible along each branch before backtracking, typically employing a stack (either explicitly or implicitly through recursion). This makes DFS well-suited for problems involving pathfinding, topological sorting, and situations where exploring the entire depth of a particular path is crucial, even if it doesn't guarantee the shortest path in all cases.