**Question 1**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*"Write a program to   (a.) create an array of integers and initialize it at compile-time  (b.) create another array of floating values and initialize it at run-time  (c.) display the elements of both the arrays with proper headings"\*/ #include <stdio.h>  int main() {  // (a) Create and initialize an array of integers at compile-time  int intArray[] = {10, 20, 30, 40, 50};   int intArraySize = sizeof(intArray) / sizeof(intArray[0]);   // (b) Create another array of floating values and initialize it at run-time  float floatArray[5]; // Array for 5 floating-point numbers  int floatArraySize = sizeof(floatArray) / sizeof(floatArray[0]);   printf("\nEnter %d floating-point values:\n", floatArraySize);  for (int i = 0; i < floatArraySize; i++) {  scanf("%f", &floatArray[i]);  }   // (c) Display both arrays with proper headings  printf("\nInteger Array (Compile-time initialized):\n");  for (int i = 0; i < intArraySize; i++) {  printf("%d ", intArray[i]);  }  printf("\n");   printf("\nFloating-point Array (Run-time initialized):\n");  for (int i = 0; i < floatArraySize; i++) {  printf("%.2f ", floatArray[i]);  }  printf("\n");   return 0; } |

**Question 2**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*"Write a Program to implement Linear Search for   (a.) First occurence of search item  (b.) All occurences of search item"\*/ #include <stdio.h> #include <stdlib.h> // Required for malloc and free  void searchFirstOccurrence(int arr[], int size, int key) {  for (int i = 0; i < size; i++) {  if (arr[i] == key) {  printf("First occurrence of %d found at index %d\n", key, i);  return;  }  }  printf("Element %d not found in the array.\n", key); }  void searchAllOccurrences(int arr[], int size, int key) {  int found = 0;  printf("All occurrences of %d found at indices: ", key);  for (int i = 0; i < size; i++) {  if (arr[i] == key) {  printf("%d ", i);  found = 1;  }  }  if (!found) {  printf("Element %d not found in the array.", key);  }  printf("\n"); }  int main() {  int size, key;   // Input array size  printf("\nEnter the size of the array: ");  scanf("%d", &size);   // Dynamic memory allocation  int \*arr = (int \*)malloc(size \* sizeof(int));  if (arr == NULL) {  printf("Memory allocation failed!\n");  return 1;  }   // Input array elements  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }   // Input key to search  printf("Enter the element to search: ");  scanf("%d", &key);   // (a) Find first occurrence  searchFirstOccurrence(arr, size, key);   // (b) Find all occurrences  searchAllOccurrences(arr, size, key);   // Free allocated memory  free(arr);   return 0; } |

**Question 3**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*"Write a program to   (a.) create an array of integers and initialize it  (b.) Find minimum and maximum elements in the array  (c.) Find sum and average of array elements"\*/ #include <stdio.h> #include <stdlib.h> // For malloc() and free()  // Function to find the minimum element in the array int findMin(int arr[]) {  int size = 0;  while (arr[size] != '\0') size++; // Calculate size dynamically   int min = arr[0]; // Initialize min with the first element  for (int i = 1; i < size; i++) {  if (arr[i] < min) {  min = arr[i];  }  }  return min; }  // Function to find the maximum element in the array int findMax(int arr[]) {  int size = 0;  while (arr[size] != '\0') size++; // Calculate size dynamically   int max = arr[0]; // Initialize max with the first element  for (int i = 1; i < size; i++) {  if (arr[i] > max) {  max = arr[i];  }  }  return max; }  // Function to calculate the sum of array elements int calculateSum(int arr[]) {  int size = 0, sum = 0;  while (arr[size] != '\0') { // Calculate size dynamically  sum += arr[size];  size++;  }  return sum; }  // Function to calculate the average by calling calculateSum float calculateAverage(int arr[]) {  int size = 0;  while (arr[size] != '\0') size++; // Calculate size dynamically   int sum = calculateSum(arr); // Call sum function  return (float)sum / size; }  int main() {  int size;   // (a) Input array size  printf("\nEnter the size of the array: ");  scanf("%d", &size);   // Dynamic memory allocation  int \*arr = (int \*)malloc((size + 1) \* sizeof(int)); // +1 to add NULL termination  if (arr == NULL) {  printf("Memory allocation failed!\n");  return 1;  }   // Input array elements  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  arr[size] = '\0'; // Null termination for size calculation   // (b) Find minimum and maximum elements  int min = findMin(arr);  int max = findMax(arr);   // (c) Calculate sum and average  int sum = calculateSum(arr);  float average = calculateAverage(arr);   // Display results  printf("\nMinimum element: %d\n", min);  printf("Maximum element: %d\n", max);  printf("Sum of elements: %d\n", sum);  printf("Average of elements: %.2f\n", average);   // Free allocated memory  free(arr);   return 0; } |

**Question 4**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Write a program to Merge unsorted arrays #include <stdio.h> #include <stdlib.h> // For malloc() and free()  // Function to merge two arrays void mergeArrays(int \*arr1, int size1, int \*arr2, int size2, int \*mergedArr) {  int i, j;    // Copy elements of arr1 to mergedArr  for (i = 0; i < size1; i++) {  mergedArr[i] = arr1[i];  }   // Copy elements of arr2 to mergedArr  for (j = 0; j < size2; j++) {  mergedArr[i + j] = arr2[j];  } }  int main() {  int size1, size2;   // Input size of first array  printf("Enter size of first array: ");  scanf("%d", &size1);   // Dynamic memory allocation for first array  int \*arr1 = (int \*)malloc(size1 \* sizeof(int));  if (arr1 == NULL) {  printf("Memory allocation failed!\n");  return 1;  }   // Input elements of first array  printf("Enter %d elements of first array:\n", size1);  for (int i = 0; i < size1; i++) {  scanf("%d", &arr1[i]);  }   // Input size of second array  printf("Enter size of second array: ");  scanf("%d", &size2);   // Dynamic memory allocation for second array  int \*arr2 = (int \*)malloc(size2 \* sizeof(int));  if (arr2 == NULL) {  printf("Memory allocation failed!\n");  free(arr1); // Free previously allocated memory before exiting  return 1;  }   // Input elements of second array  printf("Enter %d elements of second array:\n", size2);  for (int i = 0; i < size2; i++) {  scanf("%d", &arr2[i]);  }   // Allocate memory for merged array  int \*mergedArr = (int \*)malloc((size1 + size2) \* sizeof(int));  if (mergedArr == NULL) {  printf("Memory allocation failed!\n");  free(arr1);  free(arr2);  return 1;  }   // Merge the two arrays  mergeArrays(arr1, size1, arr2, size2, mergedArr);   // Display merged array  printf("\nMerged Array:\n");  for (int i = 0; i < size1 + size2; i++) {  printf("%d ", mergedArr[i]);  }  printf("\n");   // Free allocated memory  free(arr1);  free(arr2);  free(mergedArr);   return 0; } |

**Question 5**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Write a program to Marge sorted arrays #include <stdio.h> #include <stdlib.h> // For malloc() and free()  // Function to merge two sorted arrays into a single sorted array void mergeSortedArrays(int \*arr1, int size1, int \*arr2, int size2, int \*mergedArr) {  int i = 0, j = 0, k = 0;   // Merge elements in sorted order  while (i < size1 && j < size2) {  if (arr1[i] <= arr2[j]) {  mergedArr[k++] = arr1[i++];  } else {  mergedArr[k++] = arr2[j++];  }  }   // Copy remaining elements of arr1 (if any)  while (i < size1) {  mergedArr[k++] = arr1[i++];  }   // Copy remaining elements of arr2 (if any)  while (j < size2) {  mergedArr[k++] = arr2[j++];  } }  int main() {  int size1, size2;   // Input size of first array  printf("Enter size of first sorted array: ");  scanf("%d", &size1);   // Dynamic memory allocation for first array  int \*arr1 = (int \*)malloc(size1 \* sizeof(int));  if (arr1 == NULL) {  printf("Memory allocation failed!\n");  return 1;  }   // Input elements of first sorted array  printf("Enter %d elements (in sorted order) for first array:\n", size1);  for (int i = 0; i < size1; i++) {  scanf("%d", &arr1[i]);  }   // Input size of second array  printf("Enter size of second sorted array: ");  scanf("%d", &size2);   // Dynamic memory allocation for second array  int \*arr2 = (int \*)malloc(size2 \* sizeof(int));  if (arr2 == NULL) {  printf("Memory allocation failed!\n");  free(arr1);  return 1;  }   // Input elements of second sorted array  printf("Enter %d elements (in sorted order) for second array:\n", size2);  for (int i = 0; i < size2; i++) {  scanf("%d", &arr2[i]);  }   // Allocate memory for merged array  int \*mergedArr = (int \*)malloc((size1 + size2) \* sizeof(int));  if (mergedArr == NULL) {  printf("Memory allocation failed!\n");  free(arr1);  free(arr2);  return 1;  }   // Merge the two sorted arrays  mergeSortedArrays(arr1, size1, arr2, size2, mergedArr);   // Display merged sorted array  printf("\nMerged Sorted Array:\n");  for (int i = 0; i < size1 + size2; i++) {  printf("%d ", mergedArr[i]);  }  printf("\n");   // Free allocated memory  free(arr1);  free(arr2);  free(mergedArr);   return 0; } |

**Question 6**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Write a program to insert a number at a given location in an array. #include <stdio.h>  #define MAX\_SIZE 100 // Define a maximum size for safety  int arrLen;  void arrPrint(int arr[]) {  printf("\n");  for (int i = 0; i < arrLen; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  void arrInsert(int arr[], int num, int pos) {  if (pos < 1 || pos > arrLen + 1) {  printf("Invalid position!\n");  return;  }   if (arrLen >= MAX\_SIZE) {  printf("Array is full! Cannot insert.\n");  return;  }   for (int i = arrLen; i >= pos; i--) {  arr[i] = arr[i - 1];  }  arr[pos - 1] = num;  arrLen++;   printf("Updated Array: ");  arrPrint(arr); }  int main() {  int arr[MAX\_SIZE], num, pos;   printf("Enter Number of Elements in Array: ");  scanf("%d", &arrLen);   for (int i = 0; i < arrLen; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &arr[i]);  }   printf("Enter element to insert: ");  scanf("%d", &num);  printf("Enter position: ");  scanf("%d", &pos);   arrInsert(arr, num, pos);    return 0; } |

**Question 7**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Write a program to delete a number from a given location in an array. #include <stdio.h>  #define MAX\_SIZE 100 // Define a maximum size for safety  int arrLen;  void arrPrint(int arr[]) {  printf("\n");  for (int i = 0; i < arrLen; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  void arrDelete(int arr[], int pos) {  if (pos < 1 || pos > arrLen) {  printf("Invalid position!\n");  return;  }   for (int i = pos - 1; i < arrLen - 1; i++) {  arr[i] = arr[i + 1];  }  arrLen--;   printf("Updated Array: ");  arrPrint(arr); }  int main() {  int arr[MAX\_SIZE], pos;   printf("Enter Number of Elements in Array: ");  scanf("%d", &arrLen);   for (int i = 0; i < arrLen; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &arr[i]);  }   printf("Enter position to delete: ");  scanf("%d", &pos);   arrDelete(arr, pos);   return 0; } |

**Question 8**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Write a program to search a number in an array and delete it, if found. #include <stdio.h>  #define MAX\_SIZE 100 // Define a maximum size for safety  int arrLen;  void arrPrint(int arr[]) {  printf("\n");  for (int i = 0; i < arrLen; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  void arrDelete(int arr[], int pos) {  for (int i = pos - 1; i < arrLen - 1; i++) {  arr[i] = arr[i + 1];  }  arrLen--; }  void arrSearchDel(int arr[], int num) {  int pos = -1;  for (int i = 0; i < arrLen; i++) {  if (arr[i] == num) {  pos = i + 1;  break;  }  }   if (pos == -1) {  printf("Element not found!\n");  return;  }   arrDelete(arr, pos);  printf("Updated Array: ");  arrPrint(arr); }  int main() {  int arr[MAX\_SIZE], num;   printf("Enter Number of Elements in Array: ");  scanf("%d", &arrLen);   for (int i = 0; i < arrLen; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &arr[i]);  }   printf("Enter element to search and delete: ");  scanf("%d", &num);   arrSearchDel(arr, num);   return 0; } |

**Question 9**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Write a Program to implement Binary Search #include <stdio.h>  #define MAX\_SIZE 100 // Define a maximum size for safety  int arrLen; void arrPrint(int arr[]) {  printf("\n");  for (int i = 0; i < arrLen; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  void arrSort(int arr[]) {  for (int i = 0; i < arrLen - 1; i++) {  for (int j = i + 1; j < arrLen; j++) {  if (arr[i] > arr[j]) {  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  }  printf("Sorted Array: ");  arrPrint(arr); }  void arrBiSearch(int arr[], int num) {  int low = 0, high = arrLen - 1, mid;   while (low <= high) {  mid = (low + high) / 2;   if (arr[mid] == num) {  printf("Element found at Position: %d\n", mid + 1);  return;  } else if (arr[mid] < num) {  low = mid + 1;  } else {  high = mid - 1;  }  }  printf("Element not found!\n"); }  int main() {  int arr[MAX\_SIZE], num;   printf("Enter Number of Elements in Array: ");  scanf("%d", &arrLen);   for (int i = 0; i < arrLen; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &arr[i]);  }   arrSort(arr);   printf("Enter element to search: ");  scanf("%d", &num);   arrBiSearch(arr, num);   return 0; } |

**Question 10**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Write a Program to perform elimination of duplicate elements from an existing list of elements. #include <stdio.h>  #define MAX\_SIZE 100 // Define a maximum size for safety  int arrLen;  void arrPrint(int arr[]) {  printf("\n");  for (int i = 0; i < arrLen; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  void arrDelete(int arr[], int pos) {  for (int i = pos - 1; i < arrLen - 1; i++) {  arr[i] = arr[i + 1];  }  arrLen--; }  void arrElmDup(int arr[]) {  for (int i = 0; i < arrLen; i++) {  for (int j = i + 1; j < arrLen; ) {  if (arr[i] == arr[j]) {  arrDelete(arr, j + 1);  } else {  j++; // Only increment when no deletion occurs  }  }  }   printf("Updated Array: ");  arrPrint(arr); }  int main() {  int arr[MAX\_SIZE];   printf("Enter Number of Elements in Array: ");  scanf("%d", &arrLen);   for (int i = 0; i < arrLen; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &arr[i]);  }   arrElmDup(arr);   return 0; } |

**Question 11**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // C Program for Matrix Operations Using Switch-Case #include <stdio.h>  #define SIZE 3 // Define matrix size  void inputMatrix(int matrix[SIZE][SIZE], char name) {  printf("Enter elements of matrix %c (%dx%d):\n", name, SIZE, SIZE);  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  printf("%c[%d][%d]: ", name, i, j);  scanf("%d", &matrix[i][j]);  }  } }  void printMatrix(int matrix[SIZE][SIZE]) {  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  printf("%d\t", matrix[i][j]);  }  printf("\n");  } }  void addMatrices(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]) {  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  result[i][j] = A[i][j] + B[i][j];  }  } }  void subtractMatrices(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]) {  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  result[i][j] = A[i][j] - B[i][j];  }  } }  void multiplyMatrices(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]) {  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  result[i][j] = 0;  for (int k = 0; k < SIZE; k++) {  result[i][j] += A[i][k] \* B[k][j];  }  }  } }  void transposeMatrix(int A[SIZE][SIZE], int result[SIZE][SIZE]) {  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  result[j][i] = A[i][j];  }  } }  int main() {  int A[SIZE][SIZE], B[SIZE][SIZE], result[SIZE][SIZE];  int choice;   // Input matrices  inputMatrix(A, 'A');  inputMatrix(B, 'B');   // Menu  printf("\nChoose operation:\n");  printf("1. Addition\n2. Subtraction\n3. Multiplication\n4. Transpose (of A)\n");  printf("Enter your choice: ");  scanf("%d", &choice);   switch (choice) {  case 1:  addMatrices(A, B, result);  printf("\nResultant Matrix after Addition:\n");  printMatrix(result);  break;  case 2:  subtractMatrices(A, B, result);  printf("\nResultant Matrix after Subtraction:\n");  printMatrix(result);  break;  case 3:  multiplyMatrices(A, B, result);  printf("\nResultant Matrix after Multiplication:\n");  printMatrix(result);  break;  case 4:  transposeMatrix(A, result);  printf("\nTranspose of Matrix A:\n");  printMatrix(result);  break;  default:  printf("\nInvalid choice!\n");  }   return 0; } |

**Question 12**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Read and Display Upper, Lower and Tri-diagonal Triangle. #include <stdio.h> #include <stdlib.h> // Required for malloc()  // Function to input a dynamically allocated matrix void inputMatrix(int SIZE, int \*\*matrix) {  printf("Enter elements of the matrix (%dx%d):\n", SIZE, SIZE);  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  printf("matrix[%d][%d]: ", i, j);  scanf("%d", &matrix[i][j]);  }  } }  // Function to print a dynamically allocated matrix void printMatrix(int SIZE, int \*\*matrix) {  printf("\nMatrix is:\n");  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  printf("%4d ", matrix[i][j]);  }  printf("\n");  } }  // Function to print lower triangle matrix void lowerTri(int SIZE, int \*\*matrix){  printf("\nLower Triangular Matrix:\n");  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  if (i>=j){  printf("%4d ", matrix[i][j]);  }  }  printf("\n");  } }  // Function to print upper triangle matrix void upperTri(int SIZE, int \*\*matrix){  printf("\nUpper Triangular Matrix:\n");  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  if (i<=j){  printf("%4d ", matrix[i][j]);  }  else{  printf(" ");  }  }  printf("\n");  } }  // Function to print tri diagonal matrix void triDiagonal(int SIZE, int \*\*matrix){  printf("\nTri Diagonal Matrix:\n");  for (int i = 0; i < SIZE; i++) {  for (int j = 0; j < SIZE; j++) {  if (i==j || i-1 == j || i+1 == j){  printf("%4d ", matrix[i][j]);  }  else{  printf(" ");  }  }  printf("\n");  } }   // Main Function int main() {  int SIZE;    printf("Enter Dimension of Square Matrix: ");  scanf("%d", &SIZE);   // Dynamically allocate 2D array  int \*\*matrix = (int \*\*)malloc(SIZE \* sizeof(int \*));  for (int i = 0; i < SIZE; i++) {  matrix[i] = (int \*)malloc(SIZE \* sizeof(int));  }   inputMatrix(SIZE, matrix);  printMatrix(SIZE, matrix);  lowerTri(SIZE, matrix);  upperTri(SIZE, matrix);  triDiagonal(SIZE, matrix);   // Free allocated memory  for (int i = 0; i < SIZE; i++) {  free(matrix[i]);  }  free(matrix);   return 0; } |

**Question 13**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // Implementing Sparse Matrix using 3-tuple Representation #include <stdio.h> #include <stdlib.h>  // // Global Variables // int \*\*sparse; // int \*\*matrix;  // Function to input a dynamically allocated matrix void inputMatrix(int rowSIZE, int colSIZE, int \*\*matrix) {  printf("Enter elements of the matrix (%dx%d):\n", rowSIZE, colSIZE);  for(int i = 0; i < rowSIZE; i++) {  for(int j = 0; j < colSIZE; j++) {  printf("matrix[%d][%d]: ", i, j);  scanf("%d", &matrix[i][j]);  }  } }  // Function to print a dynamically allocated matrix void printMatrix(int rowSIZE, int colSIZE, int \*\*matrix) {  printf("Matrix is:\n");  for(int i = 0; i < rowSIZE; i++) {  for(int j = 0; j < colSIZE; j++) {  printf("%4d ", matrix[i][j]);  }  printf("\n");  } }  // Function to create a Sparse Matrix void toSparseMatrix(int rowSIZE, int colSIZE, int \*\*matrix){  int nonZeros = 0;   for(int i=0; i<rowSIZE; i++){  for(int j=0; j<colSIZE; j++){  if(matrix[i][j] != 0){  nonZeros++;  }  }  }   // Allocating Memory for Sparse Matrix  int \*\*sparse = (int \*\*)malloc((nonZeros+1) \* sizeof(int \*));  for(int i=0; i<nonZeros+1; i++){  sparse[i] = (int \*)malloc(3 \* sizeof(int));  }   // Storing dimentions and no. of nonZeros in First row  sparse[0][0] = rowSIZE;  sparse[0][1] = colSIZE;  sparse[0][2] = nonZeros;   // Storing Values in Sparse Matrix  int k=1; // k is index 1 of sparse array as 0th index contains data of og matrix  for(int i=0; i<rowSIZE; i++){  for(int j=0; j<colSIZE; j++){  if(matrix[i][j] != 0){  sparse[k][0] = i;  sparse[k][1] = j;  sparse[k][2] = matrix[i][j];  k++;  }  }  }  printf("Sparsed ");  printMatrix(nonZeros+1, 3, sparse);    }  // Coverting sparse to normal matrix void toNormalMatrix(int rowCount, int \*\*sparse){  int rowSIZE = sparse[0][0];  int colSIZE = sparse[0][1];   // Dynamically allocate 2D array  int \*\*matrix = (int \*\*)malloc(rowSIZE \* sizeof(int \*));  for(int i = 0; i < rowSIZE; i++) {  matrix[i] = (int \*)malloc(colSIZE \* sizeof(int));  }   // Initialising Matix to 0  for(int i=0; i<rowSIZE; i++){  for(int j=0; j<colSIZE; j++){  matrix[i][j] = 0;  }  }   // Fetching and Placing non-zero values  for(int i=1; i<rowCount+1; i++){  matrix[sparse[i][0]][sparse[i][1]] = sparse[i][2];  }   printMatrix(rowSIZE, colSIZE, matrix);     } // Main Driver Function int main() {  int rowSIZE, colSIZE;  int choice;  printf("\nNormal to Sparse Matrix (1)\n");  printf("Sparse to Normal Matrix (2)\n");  printf("Enter Choice: ");  scanf("%d", &choice);   if(choice == 1){  printf("Enter Number of Rows: ");  scanf("%d", &rowSIZE);   printf("Enter Number of Columns: ");  scanf("%d", &colSIZE);   // Dynamically allocate 2D array  int \*\*matrix = (int \*\*)malloc(rowSIZE \* sizeof(int \*));  for(int i = 0; i < rowSIZE; i++) {  matrix[i] = (int \*)malloc(colSIZE \* sizeof(int));  }   inputMatrix(rowSIZE, colSIZE, matrix);  printMatrix(rowSIZE, colSIZE, matrix);  toSparseMatrix(rowSIZE, colSIZE, matrix);   // Free allocated memory  for(int i = 0; i < rowSIZE; i++) {  free(matrix[i]);  }  free(matrix);  }  else if(choice == 2){  int nonZero;  printf("\nEnter Number of non-zero values: ");  scanf("%d", &nonZero);   // Dynamically allocate 2D array  int \*\*sparse = (int \*\*)malloc((nonZero+1) \* sizeof(int \*));  for(int i = 0; i < nonZero+1; i++) {  sparse[i] = (int \*)malloc(3 \* sizeof(int));  }   inputMatrix(nonZero+1, 3, sparse);  toNormalMatrix(nonZero, sparse);   for(int i = 0; i < nonZero+1; i++) {  free(sparse[i]);  }  free(sparse);   }  else{  printf("\nINVALID INPUT\n");  }     return 0; } |

**Question 14**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| #include <stdio.h> #include <stdlib.h> // For malloc and free  void selectionSort(int arr[], int n) {  int i, j, minIndex, temp;    for (i = 0; i < n - 1; i++) {  minIndex = i;    // Find the minimum element in the unsorted part of the array  for (j = i + 1; j < n; j++) {  if (arr[j] < arr[minIndex]) {  minIndex = j;  }  }    // Swap the found minimum element with the element at i  if (minIndex != i) {  temp = arr[i];  arr[i] = arr[minIndex];  arr[minIndex] = temp;  }  } }  void printArray(int arr[], int n) {  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  int main() {  int n;    // Prompt the user to enter the size of the array  printf("Enter the number of elements: ");  scanf("%d", &n);    // Dynamically allocate memory for the array based on the user input  int \*arr = (int \*)malloc(n \* sizeof(int)); // Using malloc for dynamic memory allocation    if (arr == NULL) { // Check if memory allocation was successful  printf("Memory allocation failed!\n");  return 1;  }    // Prompt the user to input the elements of the array  printf("Enter %d elements: ", n);  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }    printf("Original Array: ");  printArray(arr, n);    selectionSort(arr, n);    printf("Sorted Array: ");  printArray(arr, n);    // Free the dynamically allocated memory  free(arr);    return 0; } |

**Question 15**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to implement Insertion Sort #include <stdio.h> #include <stdlib.h>  void insertionSort(int arr[], int n){  int i, key, j;  for (i = 1; i < n; i++) {  key = arr[i];  j = i - 1;   // Move elements that are greater than key to one position ahead  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key; // Insert key at the correct position  } }  void printArray(int arr[], int n) {  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  int main() {  int n;    // Prompt the user to enter the size of the array  printf("Enter the number of elements: ");  scanf("%d", &n);    // Dynamically allocate memory for the array based on the user input  int \*arr = (int \*)malloc(n \* sizeof(int)); // Using malloc for dynamic memory allocation    if (arr == NULL) { // Check if memory allocation was successful  printf("Memory allocation failed!\n");  return 1;  }    // Prompt the user to input the elements of the array  printf("Enter %d elements: ", n);  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }    printf("Original Array: ");  printArray(arr, n);    insertionSort(arr, n);   printf("Sorted Array: ");  printArray(arr, n);    // Free the dynamically allocated memory  free(arr);    return 0; } |

**Question 16**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to implement Bubble Sort using malloc #include <stdio.h> #include <stdlib.h>  // Function to perform Bubble Sort void bubbleSort(int arr[], int n) {  int i, j, temp;  for (i = 0; i < n - 1; i++) {  for (j = 0; j < n - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  // Swap  temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  } }  // Function to print array void printArray(int arr[], int n) {  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  int main() {  int n;   // Input size of array  printf("Enter number of elements: ");  scanf("%d", &n);   // Dynamically allocate memory  int \*arr = (int \*)malloc(n \* sizeof(int));   if (arr == NULL) {  printf("Memory allocation failed!\n");  return 1;  }   // Input array elements  printf("Enter %d elements: ", n);  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }   printf("Original Array: ");  printArray(arr, n);   // Bubble Sort  bubbleSort(arr, n);   printf("Sorted Array: ");  printArray(arr, n);   // Free memory  free(arr);   return 0; } |

**Question 17**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| #include <stdio.h> #include <stdlib.h>  // Function to merge two halves void merge(int arr[], int left, int mid, int right) {  int n1 = mid - left + 1; // Size of the left subarray  int n2 = right - mid; // Size of the right subarray   // Create temporary arrays for left and right subarrays  int \*L = (int \*)malloc(n1 \* sizeof(int));  int \*R = (int \*)malloc(n2 \* sizeof(int));   // Check if memory allocation was successful  if (L == NULL || R == NULL) {  printf("Memory allocation failed!\n");  exit(1);  }   // Copy data to temporary arrays L[] and R[]  for (int i = 0; i < n1; i++) {  L[i] = arr[left + i];  }  for (int j = 0; j < n2; j++) {  R[j] = arr[mid + 1 + j];  }   int i = 0, j = 0, k = left;    // Merge the temp arrays back into the original array  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  arr[k] = L[i];  i++;  } else {  arr[k] = R[j];  j++;  }  k++;  }   // Copy remaining elements of L[] (if any)  while (i < n1) {  arr[k] = L[i];  i++;  k++;  }   // Copy remaining elements of R[] (if any)  while (j < n2) {  arr[k] = R[j];  j++;  k++;  }   // Free the dynamically allocated memory for temporary arrays  free(L);  free(R); }  // Function to implement merge sort void mergeSort(int arr[], int left, int right) {  if (left < right) {  int mid = left + (right - left) / 2;   // Recursively sort the two halves  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);   // Merge the sorted halves  merge(arr, left, mid, right);  } }  // Function to print the array void printArray(int arr[], int n) {  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n"); }  int main() {  int n;   // Prompt the user to enter the size of the array  printf("Enter the number of elements: ");  scanf("%d", &n);   // Dynamically allocate memory for the array  int \*arr = (int \*)malloc(n \* sizeof(int));   if (arr == NULL) { // Check if memory allocation was successful  printf("Memory allocation failed!\n");  return 1;  }   // Prompt the user to input the elements of the array  printf("Enter %d elements: ", n);  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }   printf("Original Array: ");  printArray(arr, n);   // Call mergeSort function to sort the array  mergeSort(arr, 0, n - 1);   printf("Sorted Array: ");  printArray(arr, n);   // Free the dynamically allocated memory  free(arr);   return 0; } |

**Question 18**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*WAP to implement Singly Linked List that stores data as integer and perform following operations: Traverse the list to display each element Search for a specific element in the list\*/  #include <stdio.h> #include <stdlib.h>  //Structure for a node in the singly linked list struct Node {  int data; //Stores integer data  struct Node\* next; //Pointer to the next node };  //Function to insert a node at the end of the list void insertNode(struct Node\*\* head, int value) {  //Create a new node  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = NULL;   //If the list is empty, make the new node the head  if (\*head == NULL) {  \*head = newNode;  return;  }   //Traverse to the last node  struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }   //Insert the new node at the end  temp->next = newNode; }  //Function to traverse and display all elements in the list void traverseList(struct Node\* head) {  struct Node\* temp = head;   //If list is empty  if (temp == NULL) {  printf("List is empty!\n");  return;  }   printf("Linked List Elements: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n"); }  //Function to search for an element in the list void searchElement(struct Node\* head, int key) {  struct Node\* temp = head;  int position = 1;   while (temp != NULL) {  if (temp->data == key) {  printf("Element %d found at position %d.\n", key, position);  return;  }  temp = temp->next;  position++;  }   printf("Element %d not found in the list.\n", key); }  int main() {  struct Node\* head = NULL;  int n, value, searchKey;   //Prompt user for number of elements  printf("Enter the number of elements to insert: ");  scanf("%d", &n);   //Insert elements into the linked list  printf("Enter %d elements: \n", n);  for (int i = 0; i < n; i++) {  printf("Enter Element %d: ", i+1);  scanf("%d", &value);  insertNode(&head, value);  }   //Display the linked list  traverseList(head);   //Search for the element in the linked list  printf("Enter the element to search: ");  scanf("%d", &searchKey);  searchElement(head, searchKey);   return 0; } |

**Question 19**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*WAP to implement Singly Linked List that stores data as integer and perform following operations: Count the number of nodes in the list Find minimum and maximum value in the linked list.\*/  #include <stdio.h> #include <stdlib.h>  //Structure fora node in the singly linked list struct Node {  int data; //Stores integer data  struct Node\* next; //Pointer to the next node };  //Function to insert a node at the end of the list void insertNode(struct Node\*\* head, int value){  //Create a new node  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = NULL;   //ifthe list is empty, make the new node the head  if(\*head == NULL){  \*head = newNode;  return;  }   //Traverse to the last node  struct Node\* temp = \*head;  while(temp->next != NULL){  temp = temp->next;  }   //Insert the new node at the end  temp->next = newNode; }  //Function to count elements in list void countListElm(struct Node\* head){  struct Node\* temp = head;  int count = 0;  if(temp == NULL){  printf("List is empty!\n");  return;  }  while(temp != NULL){  count++;  temp = temp->next;  }  printf("Total Elements = %d\n", count); }  //Function to Find Max and Min number in List void maxMinList(struct Node\* head) {  if (head == NULL) { // Check if the list is empty  printf("Empty List\n");  return;  }   struct Node\* temp = head;  int max = temp->data;  int min = temp->data;   while (temp != NULL) {  if (temp->data > max) {  max = temp->data;  }  if (temp->data < min) {  min = temp->data;  }  temp = temp->next; // Move to the next node  }   printf("Maximum Element in List is %d\n", max);  printf("Minimum Element in List is %d\n", min); }  int main(){  struct Node\* head = NULL;  int n, value, searchKey;   //Prompt user for number of elements  printf("Enter the number of elements to insert: ");  scanf("%d", &n);   //Insert elements into the linked list  printf("Enter %d elements: \n", n);  for(int i = 0; i < n; i++){  printf("Enter Element %d: ", i+1);  scanf("%d", &value);  insertNode(&head, value);  }   //Display the linked list  countListElm(head);   //Search for the Maximum and Mimimum element in the linked list  maxMinList(head);   return 0; } |

**Question 20**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| #include <stdio.h> #include <stdlib.h>  // Structure for a node in the singly linked list struct Node {  int data;  struct Node\* next; };  // Function to display all elements in the list void printList(struct Node\* head) {  struct Node\* temp = head;   if (temp == NULL) {  printf("List is empty!\n\n");  return;  }   printf("Linked List Elements: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n\n"); }  // Function to insert a node at the end of the list void insertNode(struct Node\*\* head, int value){  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = NULL;   if (\*head == NULL){  \*head = newNode;  return;  }   struct Node\* temp = \*head;  while (temp->next != NULL){  temp = temp->next;  }   temp->next = newNode; }  // Insert at the beginning void insertBeg(struct Node\*\* head, int val){  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = val;  newNode->next = \*head;  \*head = newNode; }  // Insert at the end (with head pointer check) void insertEnd(struct Node\*\* head, int val){  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = val;  newNode->next = NULL;   if (\*head == NULL) {  \*head = newNode;  return;  }   struct Node\* temp = \*head;  while (temp->next != NULL){  temp = temp->next;  }   temp->next = newNode; }  // Insert after a given position (1-based) void insertAfter(struct Node\* head, int val, int pos){  struct Node\* temp = head;  int count = 1;   while (count < pos && temp != NULL){  temp = temp->next;  count++;  }   if (temp == NULL){  printf("Invalid position to insert after.\n\n");  return;  }   struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = val;  newNode->next = temp->next;  temp->next = newNode; }  // Insert before a given position (1-based) void insertBefore(struct Node\*\* head, int val, int pos) {  if (pos <= 1 || \*head == NULL) {  insertBeg(head, val);  return;  }   struct Node\* temp = \*head;  int count = 1;   while (count < pos - 1 && temp->next != NULL){  temp = temp->next;  count++;  }   if (temp->next == NULL){  printf("Invalid position to insert before.\n\n");  return;  }   struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = val;  newNode->next = temp->next;  temp->next = newNode; }  int main(){  struct Node\* head = NULL;  int n, value;   printf("Enter the number of elements to insert: ");  scanf("%d", &n);   printf("Enter %d elements:\n", n);  for(int i = 0; i < n; i++){  printf("Enter Element %d: ", i + 1);  scanf("%d", &value);  insertNode(&head, value);  }   printList(head);   // Insert at beginning  int newVal, pos;  printf("Enter Element to Insert at Beginning: ");  scanf("%d", &newVal);  insertBeg(&head, newVal);  printList(head);   // Insert at end  printf("Enter Element to Insert at End: ");  scanf("%d", &newVal);  insertEnd(&head, newVal);  printList(head);   // Insert after a given position  printf("Enter Element to Insert: ");  scanf("%d", &newVal);  printf("Enter Position After which to Insert (1-based): ");  scanf("%d", &pos);  insertAfter(head, newVal, pos);  printList(head);   // Insert before a given position  printf("Enter Element to Insert: ");  scanf("%d", &newVal);  printf("Enter Position Before which to Insert (1-based): ");  scanf("%d", &pos);  insertBefore(&head, newVal, pos);  printList(head);   return 0; } |

**Question 21**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*WAP to implement two Singly Linked List that stores data as integer and perform following operation: Merge the two lists to create a new sorted list\*/  #include <stdio.h> #include <stdlib.h>  // Structure for a node in the singly linked list struct Node {  int data; // Stores integer data  struct Node\* next; // Pointer to the next node };  // Function to insert a node at the end of the list void insertNode(struct Node\*\* head, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = NULL;   if (\*head == NULL) {  \*head = newNode;  return;  }   struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }   temp->next = newNode; }  // Function to display all elements in the list void printList(struct Node\* head) {  if (head == NULL) {  printf("List is empty!\n");  return;  }   struct Node\* temp = head;  printf("Linked List Elements: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n"); }  // Function to merge two sorted linked lists into a new sorted list struct Node\* mergeSortedLists(struct Node\* list1, struct Node\* list2) {  // Create a dummy node to simplify merging  struct Node dummy;  struct Node\* tail = &dummy;  dummy.next = NULL;   while (list1 != NULL && list2 != NULL) {  if (list1->data <= list2->data) {  tail->next = list1;  list1 = list1->next;  } else {  tail->next = list2;  list2 = list2->next;  }  tail = tail->next;  }   // Attach remaining elements  if (list1 != NULL) {  tail->next = list1;  } else {  tail->next = list2;  }   return dummy.next; }  // Main function int main() {  struct Node\* head1 = NULL;  struct Node\* head2 = NULL;  struct Node\* mergedHead = NULL;  int n, value;   // Creating first linked list  printf("Enter the number of elements for first sorted list: ");  scanf("%d", &n);  printf("Enter %d elements in sorted order:\n", n);  for (int i = 0; i < n; i++) {  scanf("%d", &value);  insertNode(&head1, value);  }   // Creating second linked list  printf("\nEnter the number of elements for second sorted list: ");  scanf("%d", &n);  printf("Enter %d elements in sorted order:\n", n);  for (int i = 0; i < n; i++) {  scanf("%d", &value);  insertNode(&head2, value);  }   // Merging the two sorted lists  mergedHead = mergeSortedLists(head1, head2);   // Display the merged sorted list  printf("\nMerged Sorted List:\n");  printList(mergedHead);   return 0; } |

**Question 22**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to implement a SIngly Linked List that sotres data as intee and perform following operations: // Create a new ist that is reverse of the first linked list  #include <stdio.h> #include <stdlib.h>  struct Node{  int data;  struct Node \*next; };  //Function to insert a node at the end of the list void insertNode(struct Node\*\* head, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = NULL;   //If the list is empty, make the new node the head  if (\*head == NULL) {  \*head = newNode;  return;  }   struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }   temp->next = newNode; }  // Function to display all elements in the list void printList(struct Node\* head) {  if (head == NULL) {  printf("List is empty!\n");  return;  }   struct Node\* temp = head;  printf("Linked List Elements: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n"); }  // Funtion to insert a node in the beginning void insertBeg(struct Node\*\* head, int val){  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = val;  newNode->next = \*head;  \*head = newNode; }  void reverseLL(struct Node \*start) {  struct Node \*ptr = start;  struct Node \*revLL = NULL;    int i = 0;  int value;  while(ptr != NULL){  value = ptr->data;  insertBeg(&revLL, value);  ptr = ptr->next;  }   printf("\nReversed List is: \n");  printList(revLL);  }  int main() {  struct Node\* head = NULL;  int n, value;   printf("Enter the number of elements to insert: ");  scanf("%d", &n);   printf("Enter %d elements: \n", n);  for (int i = 0; i < n; i++) {  printf("Enter Element %d: ", i+1);  scanf("%d", &value);  insertNode(&head, value);  }   reverseLL(head);  return 0; } |

**Question 23**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| #include <stdio.h> #include <stdlib.h>  struct Node {  int data;  struct Node\* next; };  // Insert node at the end void insertNode(struct Node\*\* head, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = NULL;   if (\*head == NULL) {  \*head = newNode;  return;  }   struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }   temp->next = newNode; }  // Display the list void printList(struct Node\* head) {  if (head == NULL) {  printf("List is empty!\n\n");  return;  }   printf("Linked List Elements: ");  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n\n"); }  // Delete first node struct Node\* deleteFirst(struct Node\* head) {  if (head == NULL) {  printf("List is already empty.\n");  return NULL;  }   struct Node\* temp = head;  head = head->next;  free(temp);   return head; }  // Delete last node struct Node\* deleteLast(struct Node\* head) {  if (head == NULL) {  printf("List is already empty.\n");  return NULL;  }   if (head->next == NULL) {  free(head);  return NULL;  }   struct Node\* temp = head;  while (temp->next->next != NULL) {  temp = temp->next;  }   free(temp->next);  temp->next = NULL;   return head; }  // Delete node after a given value struct Node\* deleteAfter(struct Node\* head, int val) {  struct Node\* temp = head;   while (temp != NULL && temp->data != val) {  temp = temp->next;  }   if (temp == NULL || temp->next == NULL) {  printf("No node found after value %d.\n", val);  return head;  }   struct Node\* toDelete = temp->next;  temp->next = toDelete->next;  free(toDelete);   return head; }  // Delete a node with specific value (search and delete) struct Node\* deleteValue(struct Node\* head, int val) {  if (head == NULL) {  printf("List is empty.\n");  return NULL;  }   // If value is in the head node  if (head->data == val) {  struct Node\* temp = head;  head = head->next;  free(temp);  return head;  }   struct Node\* temp = head;  while (temp->next != NULL && temp->next->data != val) {  temp = temp->next;  }   if (temp->next == NULL) {  printf("Element %d not found in the list.\n", val);  return head;  }   struct Node\* toDelete = temp->next;  temp->next = toDelete->next;  free(toDelete);   return head; }  // Main function int main() {  struct Node\* start = NULL;  int n, value;   printf("Enter the number of elements to insert: ");  scanf("%d", &n);   printf("Enter %d elements:\n", n);  for (int i = 0; i < n; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &value);  insertNode(&start, value);  }   printList(start);   // Delete First Node  printf("Deleting First Node...\n");  start = deleteFirst(start);  printList(start);   // Delete Last Node  printf("Deleting Last Node...\n");  start = deleteLast(start);  printList(start);   // Delete After Value  printf("Enter Value whose next node is to be deleted: ");  scanf("%d", &value);  start = deleteAfter(start, value);  printList(start);   // Search and Delete a Value  printf("Enter the Element to Delete (Search and Delete): ");  scanf("%d", &value);  start = deleteValue(start, value);  printList(start);   return 0; } |

**Question 24**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| #include <stdio.h> #include <stdlib.h>  struct Node {  struct Node \*prev;  int data;  struct Node \*next; };  // Function to insert a node at the end of the list struct Node\* insertNode(struct Node\* head, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->prev = NULL;  newNode->data = value;  newNode->next = NULL;   // If the list is empty, make the new node the head  if (head == NULL) {  return newNode; // Return the new head  }   struct Node\* temp = head;  while (temp->next != NULL) {  temp = temp->next;  }   temp->next = newNode;  newNode->prev = temp;   return head; // Return the updated head }  // Function to display all elements in the list void printList(struct Node\* head) {  if (head == NULL) {  printf("List is empty!\n");  return;  }   struct Node\* temp = head;  printf("Linked List Elements: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n"); }  // Function to search for a specific element in the list void searchList(struct Node\* head, 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 %d not found in the list\n", key); }  int main() {  struct Node\* start = NULL;  int n, value, key;   printf("Enter the number of elements to insert: ");  scanf("%d", &n);   printf("Enter %d elements: \n", n);  for (int i = 0; i < n; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &value);  start = insertNode(start, value); // Fix: Update start with the new head  }   printf("\nThe List is: \n");  printList(start);   // Search for an element  printf("\nEnter the element to search for: ");  scanf("%d", &key);  searchList(start, key);   return 0; // Fix: Standard return statement for main() } |

**Question 25**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*WAP to implement Header Linked List with Operations:   Insertion (Start Between End)  Deletion (Start Between End)  Traverse\*/  #include <stdio.h> #include <stdlib.h>  // Node structure struct Node {  int data;  struct Node\* next; };  // Header Linked List structure struct HeaderList {  int count;  struct Node\* head; };  // Function to create a header linked list struct HeaderList\* createHeaderList() {  struct HeaderList\* hlist = (struct HeaderList\*)malloc(sizeof(struct HeaderList));  hlist->count = 0;  hlist->head = NULL;  return hlist; }  // Function to insert a node at the start void insertBeg(struct HeaderList\* hlist, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = hlist->head;  hlist->head = newNode;  hlist->count++; }  // Function to insert a node at the end void insertEnd(struct HeaderList\* hlist, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = NULL;   if (hlist->head == NULL) {  hlist->head = newNode;  } else {  struct Node\* temp = hlist->head;  while (temp->next != NULL) {  temp = temp->next;  }  temp->next = newNode;  }  hlist->count++; }  // Function to insert a node at a given position void insertPos(struct HeaderList\* hlist, int value, int position) {  if (position < 1 || position > hlist->count + 1) {  printf("Invalid position!\n");  return;  }   if (position == 1) {  insertBeg(hlist, value);  return;  }   struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;   struct Node\* temp = hlist->head;  for (int i = 1; i < position - 1; i++) {  temp = temp->next;  }   newNode->next = temp->next;  temp->next = newNode;  hlist->count++; }  // Function to delete a node from the start void deleteBeg(struct HeaderList\* hlist) {  if (hlist->head == NULL) {  printf("List is empty!\n");  return;  }   struct Node\* temp = hlist->head;  hlist->head = hlist->head->next;  free(temp);  hlist->count--; }  // Function to delete a node from the end void deleteEnd(struct HeaderList\* hlist) {  if (hlist->head == NULL) {  printf("List is empty!\n");  return;  }   struct Node\* temp = hlist->head;  struct Node\* prev = NULL;   if (temp->next == NULL) {  hlist->head = NULL;  } else {  while (temp->next != NULL) {  prev = temp;  temp = temp->next;  }  prev->next = NULL;  }   free(temp);  hlist->count--; }  // Function to delete a node from a given position void deletePos(struct HeaderList\* hlist, int position) {  if (position < 1 || position > hlist->count) {  printf("Invalid position!\n");  return;  }   if (position == 1) {  deleteBeg(hlist);  return;  }   struct Node\* temp = hlist->head;  struct Node\* prev = NULL;   for (int i = 1; i < position; i++) {  prev = temp;  temp = temp->next;  }   prev->next = temp->next;  free(temp);  hlist->count--; }  // Function to traverse the linked list void traverseList(struct HeaderList\* hlist) {  if (hlist->head == NULL) {  printf("List is empty!\n");  return;  }   struct Node\* temp = hlist->head;  printf("Linked List: ");  while (temp != NULL) {  printf("%d -> ", temp->data);  temp = temp->next;  }  printf("NULL\nTotal Nodes: %d\n", hlist->count); }  // Main function int main() {  struct HeaderList\* hlist = createHeaderList();  int n, value, choice, pos;   printf("Enter Number of Elements: ");  scanf("%d", &n);   for (int i = 0; i < n; i++) {  printf("Enter Element %d: ", i+1);  scanf("%d", &value);  insertEnd(hlist, value);  }   traverseList(hlist);   while (1) {  printf("\nOperations:\n");  printf("1. Insert at beginning\n2. Insert at end\n3. Insert at position\n");  printf("4. Delete from beginning\n5. Delete from end\n6. Delete from position\n");  printf("7. Traverse list\n8. Exit\n");  printf("Enter your choice: ");  scanf("%d", &choice);   switch (choice) {  case 1:  printf("Enter value to insert at beginning: ");  scanf("%d", &value);  insertBeg(hlist, value);  traverseList(hlist);  break;   case 2:  printf("Enter value to insert at end: ");  scanf("%d", &value);  insertEnd(hlist, value);  traverseList(hlist);  break;   case 3:  printf("Enter value and position to insert: ");  scanf("%d %d", &value, &pos);  insertPos(hlist, value, pos);  traverseList(hlist);  break;   case 4:  deleteBeg(hlist);  traverseList(hlist);  break;   case 5:  deleteEnd(hlist);  traverseList(hlist);  break;   case 6:  printf("Enter position to delete: ");  scanf("%d", &pos);  deletePos(hlist, pos);  traverseList(hlist);  break;   case 7:  traverseList(hlist);  break;   case 8:  printf("Exiting...\n");  return 0;   default:  printf("Invalid choice! Please try again.\n");  }  }   return 0; } |

**Question 26**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*WAP to implement Circular Linked List with Operations:   Insertion (Start Between End)  Deletion (Start Between End)  Traverse  Linear Search\*/   #include <stdio.h>  #include <stdlib.h>    // Node structure  struct Node {  int data;  struct Node\* next;  };    // Circular Linked List structure  struct CircularList {  struct Node\* head;  };    // Function to create an empty circular linked list  struct CircularList\* createCircularList() {  struct CircularList\* clist = (struct CircularList\*)malloc(sizeof(struct CircularList));  clist->head = NULL;  return clist;  }    // Function to insert a node at the beginning  void insertBeg(struct CircularList\* clist, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;    if (clist->head == NULL) {   newNode->next = newNode;  clist->head = newNode;  } else {  struct Node\* temp = clist->head;  while (temp->next != clist->head) {  temp = temp->next;  }  newNode->next = clist->head;  temp->next = newNode;  clist->head = newNode;  }  }    // Function to insert a node at the end  void insertEnd(struct CircularList\* clist, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;    if (clist->head == NULL) {  newNode->next = newNode;  clist->head = newNode;  } else {  struct Node\* temp = clist->head;  while (temp->next != clist->head) {  temp = temp->next;  }  temp->next = newNode;  newNode->next = clist->head;  }  }    // Function to insert a node at a given position  void insertPos(struct CircularList\* clist, int value, int position) {  if (position < 1) {  printf("Invalid position!\n");  return;  }    if (position == 1) {  insertBeg(clist, value);  return;  }    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;    struct Node\* temp = clist->head;  for (int i = 1; i < position - 1; i++) {  if (temp->next == clist->head) {  printf("Position out of range!\n");  free(newNode);  return;  }  temp = temp->next;  }    newNode->next = temp->next;  temp->next = newNode;  }    // Function to delete a node from the beginning  void deleteBeg(struct CircularList\* clist) {  if (clist->head == NULL) {  printf("List is empty!\n");  return;  }    struct Node\* temp = clist->head;  struct Node\* last = clist->head;    if (clist->head->next == clist->head) {   clist->head = NULL;  } else {  while (last->next != clist->head) {  last = last->next;  }  clist->head = clist->head->next;  last->next = clist->head;  }  free(temp);  }    // Function to delete a node from the end  void deleteEnd(struct CircularList\* clist) {  if (clist->head == NULL) {  printf("List is empty!\n");  return;  }    struct Node\* temp = clist->head;  struct Node\* prev = NULL;    if (clist->head->next == clist->head) {   clist->head = NULL;  } else {  while (temp->next != clist->head) {  prev = temp;  temp = temp->next;  }  prev->next = clist->head;  }  free(temp);  }    // Function to delete a node from a given position  void deletePos(struct CircularList\* clist, int position) {  if (clist->head == NULL || position < 1) {  printf("List is empty or invalid position!\n");  return;  }    struct Node\* temp = clist->head;  struct Node\* prev = NULL;    if (position == 1) {  deleteBeg(clist);  return;  }    for (int i = 1; i < position; i++) {  prev = temp;  temp = temp->next;  if (temp == clist->head) {  printf("Position out of range!\n");  return;  }  }    prev->next = temp->next;  free(temp);  }    // Function to traverse and display the circular linked list  void traverseList(struct CircularList\* clist) {  if (clist->head == NULL) {  printf("List is empty!\n");  return;  }    struct Node\* temp = clist->head;  printf("Circular Linked List: ");  do {  printf("%d -> ", temp->data);  temp = temp->next;  } while (temp != clist->head);  printf("(Back to Start)\n");  }    // Function to search for an element in the list  void searchElement(struct CircularList\* clist, int key) {  if (clist->head == NULL) {  printf("List is empty!\n");  return;  }    struct Node\* temp = clist->head;  int pos = 1;  do {  if (temp->data == key) {  printf("Element %d found at position %d\n", key, pos);  return;  }  temp = temp->next;  pos++;  } while (temp != clist->head);    printf("Element %d not found in the list.\n", key);  }    // Main function  int main() {  struct CircularList\* clist = createCircularList();  int n, value, choice, pos;    printf("Enter the number of elements: ");  scanf("%d", &n);    for (int i = 0; i < n; i++) {  printf("Enter Element %d: ", i+1);  scanf("%d", &value);  insertEnd(clist, value);  }    traverseList(clist);    while (1) {  printf("\nOperations:\n");  printf("1. Insert at beginning\n2. Insert at end\n3. Insert at position\n");  printf("4. Delete from beginning\n5. Delete from end\n6. Delete from position\n");  printf("7. Search element\n8. Traverse list\n9. Exit\n");  printf("Enter your choice: ");  scanf("%d", &choice);    switch (choice) {  case 1:  printf("Enter value to insert at beginning: ");  scanf("%d", &value);  insertBeg(clist, value);  traverseList(clist);  break;    case 2:  printf("Enter value to insert at end: ");  scanf("%d", &value);  insertEnd(clist, value);  traverseList(clist);  break;    case 3:  printf("Enter value and position to insert: ");  scanf("%d %d", &value, &pos);  insertPos(clist, value, pos);  traverseList(clist);  break;    case 4:  deleteBeg(clist);  traverseList(clist);  break;    case 5:  deleteEnd(clist);  traverseList(clist);  break;    case 6:  printf("Enter position to delete: ");  scanf("%d", &pos);  deletePos(clist, pos);  traverseList(clist);  break;    case 7:  printf("Enter value to search: ");  scanf("%d", &value);  searchElement(clist, value);  break;    case 8:  traverseList(clist);  break;    case 9:  printf("Exiting...\n");  return 0;    default:  printf("Invalid choice! Please try again.\n");  }  }    return 0;  } |

**Question 27**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to perform Polynomial Addition using Linked List  #include <stdio.h> #include <stdlib.h>  // Structure for a polynomial term struct Node {  int coeff; // Coefficient  int exp; // Exponent  struct Node\* next; };  // Function to create a new term struct Node\* createNode(int coeff, int exp) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->coeff = coeff;  newNode->exp = exp;  newNode->next = NULL;  return newNode; }  // Function to insert a term in sorted order (descending exponent) void insertTerm(struct Node\*\* poly, int coeff, int exp) {  struct Node\* newNode = createNode(coeff, exp);  if (\*poly == NULL || (\*poly)->exp < exp) {  newNode->next = \*poly;  \*poly = newNode;  } else {  struct Node\* temp = \*poly;  while (temp->next != NULL && temp->next->exp > exp) {  temp = temp->next;  }  newNode->next = temp->next;  temp->next = newNode;  } }  // Function to display a polynomial void displayPolynomial(struct Node\* poly) {  if (poly == NULL) {  printf("0\n");  return;  }    while (poly != NULL) {  printf("%dx^%d", poly->coeff, poly->exp);  if (poly->next != NULL)  printf(" + ");  poly = poly->next;  }  printf("\n"); }  // Function to add two polynomials struct Node\* addPolynomials(struct Node\* poly1, struct Node\* poly2) {  struct Node\* sumPoly = NULL;    while (poly1 != NULL || poly2 != NULL) {  int coeff, exp;   if (poly1 != NULL && (poly2 == NULL || poly1->exp > poly2->exp)) {  coeff = poly1->coeff;  exp = poly1->exp;  poly1 = poly1->next;  } else if (poly2 != NULL && (poly1 == NULL || poly2->exp > poly1->exp)) {  coeff = poly2->coeff;  exp = poly2->exp;  poly2 = poly2->next;  } else { // If exponents are the same, add coefficients  coeff = poly1->coeff + poly2->coeff;  exp = poly1->exp;  poly1 = poly1->next;  poly2 = poly2->next;  }   // Insert only if the coefficient is non-zero  if (coeff != 0) {  insertTerm(&sumPoly, coeff, exp);  }  }   return sumPoly; }  // Main function int main() {  struct Node \*poly1 = NULL, \*poly2 = NULL, \*sumPoly = NULL;  int n1, n2, coeff, exp;   // Input for first polynomial  printf("Enter the number of terms in the first polynomial: ");  scanf("%d", &n1);  printf("Enter terms (coefficient exponent):\n");  for (int i = 0; i < n1; i++) {  scanf("%d %d", &coeff, &exp);  insertTerm(&poly1, coeff, exp);  }   // Input for second polynomial  printf("Enter the number of terms in the second polynomial: ");  scanf("%d", &n2);  printf("Enter terms (coefficient exponent):\n");  for (int i = 0; i < n2; i++) {  scanf("%d %d", &coeff, &exp);  insertTerm(&poly2, coeff, exp);  }   // Display the polynomials  printf("\nFirst Polynomial: ");  displayPolynomial(poly1);   printf("Second Polynomial: ");  displayPolynomial(poly2);   // Add polynomials  sumPoly = addPolynomials(poly1, poly2);   // Display result  printf("Sum of Polynomials: ");  displayPolynomial(sumPoly);   return 0; } |

**Question 28**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to perform pop, push, peek, operations on a stack using array. #include <stdio.h> #include <stdlib.h>  int top = -1;  int push(int stk[], int val, int maxSize) {  if (top == maxSize - 1) {  printf("Stack Overflow! Cannot push %d\n", val);  return 0;  }  stk[++top] = val;  return 1; }  int pop(int stk[]) {  if (top == -1) {  printf("Stack Underflow! No elements to pop\n");  return -1;  }  return stk[top--]; }  int peek(int stk[]) {  if (top == -1) {  printf("Stack is empty!\n");  return -1;  }  printf("Top element: %d\n", stk[top]);  return stk[top]; }  int main() {  int len, i, elm, choice;    printf("Stack Creation and Operations \n");  printf("Enter number of elements in Stack: ");  scanf("%d", &len);   int \*stk = (int\*)malloc(len \* sizeof(int));   if (stk == NULL) {  printf("Memory allocation failed\n");  return 1;  }   for (i = 0; i < len; i++) {  printf("Enter Element %d: ", i + 1);  scanf("%d", &elm);  push(stk, elm, len);  }   do {  printf("\nChoose operation:\n");  printf("1. Push\n2. Pop\n3. Peek\n4. Exit\n");  printf("Enter choice: ");  scanf("%d", &choice);   switch (choice) {  case 1:  printf("Enter value to push: ");  scanf("%d", &elm);  push(stk, elm, len);  break;   case 2:  elm = pop(stk);  if (elm != -1)  printf("Popped element: %d\n", elm);  break;   case 3:  peek(stk);  break;   case 4:  printf("Exiting...\n");  break;   default:  printf("Invalid choice! Try again.\n");  }  } while (choice != 4);   free(stk);  return 0; } |

**Question 29**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| //(Optional) to convert a Infix Expression into a Postfix Expression using Stack #include <stdio.h> #include <ctype.h> #include <string.h>  #define MAX 100  char stack[MAX]; int top = -1;  void push(char c) {  if (top == MAX - 1) {  printf("Stack Overflow\n");  return;  }  stack[++top] = c; }  char pop() {  if (top == -1) {  return '\0';  }  return stack[top--]; }  char peek() {  if (top == -1) {  return '\0';  }  return stack[top]; }  int precedence(char c) {  if (c == '^')  return 3;  if (c == '\*' || c == '/')  return 2;  if (c == '+' || c == '-')  return 1;  return 0; }  void infixToPostfix(char\* infix, char\* postfix) {  int i, j = 0;  for (i = 0; infix[i] != '\0'; i++) {  if (isalnum(infix[i])) {   postfix[j++] = infix[i];   }   else if (infix[i] == '(') {  push(infix[i]);  }   else if (infix[i] == ')') {  while (top != -1 && peek() != '(') {  postfix[j++] = pop();  }  pop();   }   else {   while (top != -1 && precedence(peek()) >= precedence(infix[i])) {  postfix[j++] = pop();  }  push(infix[i]);  }  }    while (top != -1) {  postfix[j++] = pop();  }    postfix[j] = '\0'; }  int main() {  char infix[MAX], postfix[MAX];   printf("Enter Infix Expression: ");  scanf("%s", infix);   infixToPostfix(infix, postfix);   printf("Postfix Expression: %s\n", postfix);   return 0; } |

**Question 30**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| //(Optional) to convert a Infix Expression into a Prefix Expression using Stack#include <stdio.h> #include <stdio.h> #include <ctype.h> #include <string.h>  #define MAX 100  char stack[MAX]; int top = -1;  void push(char c) {  if (top == MAX - 1) return;  stack[++top] = c; }  char pop() {  if (top == -1) return '\0';  return stack[top--]; }  char peek() {  if (top == -1) return '\0';  return stack[top]; }  int precedence(char c) {  if (c == '^') return 3;  if (c == '\*' || c == '/') return 2;  if (c == '+' || c == '-') return 1;  return 0; }  void reverseInfix(char\* str) {  int len = strlen(str);  int i, j;  char temp;   for (i = 0, j = len - 1; i < j; i++, j--) {  temp = str[i];  str[i] = str[j];  str[j] = temp;  }   for (i = 0; i < len; i++) {  if (str[i] == '(') str[i] = ')';  else if (str[i] == ')') str[i] = '(';  } }  void infixToPostfix(char\* infix, char\* postfix) {  int i, j = 0;  top = -1;    for (i = 0; infix[i] != '\0'; i++) {  if (isalnum(infix[i])) {  postfix[j++] = infix[i];  } else if (infix[i] == '(') {  push(infix[i]);  } else if (infix[i] == ')') {  while (top != -1 && peek() != '(') postfix[j++] = pop();  pop();  } else {  while (top != -1 && precedence(peek()) > precedence(infix[i])) {  postfix[j++] = pop();  }  push(infix[i]);  }  }    while (top != -1) {  postfix[j++] = pop();  }    postfix[j] = '\0'; }  int main() {  char infix[MAX], prefix[MAX];   printf("Enter Infix Expression: ");  scanf("%s", infix);   reverseInfix(infix);  infixToPostfix(infix, prefix);  reverseInfix(prefix);   printf("Prefix Expression: %s\n", prefix);   return 0; } |

**Question 31**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to Evaluate Postfix Expression #include <stdio.h> #include <ctype.h> #include <string.h> #include <math.h> #include <stdlib.h>  #define MAX 100  int stack[MAX]; int top = -1;  void push(int val) {  if (top == MAX - 1) {  printf("Stack Overflow\n");  return;  }  stack[++top] = val; }  int pop() {  if (top == -1) {  printf("Stack Underflow\n");  exit(1);  }  return stack[top--]; }  int calVal(int a, int b, char op) {  switch (op) {  case '+': return (a + b);  case '-': return (a - b);  case '\*': return (a \* b);  case '/': return (a / b);  case '^': return (pow(a, b));  default:  printf("Invalid Operator Encountered!\n");  exit(1);  } }  int evalPostfix(char\* postfix) {  int i;  for (i = 0; postfix[i] != '\0'; i++) {  if (isdigit(postfix[i])) {   push(postfix[i] - '0');   } else {   int B = pop();  int A = pop();  int val = calVal(A, B, postfix[i]);  push(val);  }  }  return pop();  }  int main() {  char postfix[MAX], eval[MAX];   printf("Enter Postfix Expression: ");  scanf("%s", postfix);    int result = evalPostfix(postfix);   printf("Evaluated Expression: %d\n", result);   return 0; } |

**Question 32**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to Reverse a String using Stack  #include <stdio.h> #include <string.h>  #define MAX 100  char stack[MAX]; int top = -1;  void push(char c) {  if (top == MAX - 1) {  return;  }  stack[++top] = c; }  char pop() {  if (top == -1) {  return '\0';  }  return stack[top--]; }  void reverseString(char str[]) {  int len = strlen(str);    for (int i = 0; i < len; i++) {  push(str[i]);  }   for (int i = 0; i < len; i++) {  str[i] = pop();  } }  int main() {  char str[MAX];  printf("Enter String: ");  scanf("%s", str);   reverseString(str);   printf("Reversed String: %s\n", str);   return 0; } |

**Question 33**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to perform different operations with Queue such as Insert, Delete, Display of elements using dynamically allocated array (Linear Queue)  #include <stdio.h> #include <stdlib.h>  int front = -1, rear = -1;  // Enqueue Operation int enqueue(int\* queue, int val, int size) {  if (rear == size - 1) {  printf("Queue Overflow!\n");  return 0;  } else {  if (front == -1)  front = 0;  rear++;  queue[rear] = val;  printf("Element Inserted Successfully!\n");  return 1;  } }  // Dequeue Operation int dequeue(int\* queue) {  if (front == -1 || front > rear) {  printf("Queue Underflow!\n");  return -1;  } else {  int deleted = queue[front];  front++;  if (front > rear) {  // Reset queue if it's now empty  front = -1;  rear = -1;  }  return deleted;  } }  // Display Queue Elements void displayQ(int\* queue) {  if (front == -1 || front > rear) {  printf("Queue is empty.\n");  } else {  printf("Queue elements: ");  for (int i = front; i <= rear; i++) {  printf("%d ", queue[i]);  }  printf("\n");  } }  int main() {  int size;  printf("Enter Size of Queue: ");  scanf("%d", &size);   int\* queue = (int\*)malloc(size \* sizeof(int));  if (!queue) {  printf("Memory allocation failed!\n");  return 1;  }   int choice;  while (1) {  printf("\n1) Enqueue Element");  printf("\n2) Dequeue Element");  printf("\n3) Display Elements");  printf("\n0) Exit");  printf("\n\nEnter Choice: ");  scanf("%d", &choice);   switch (choice) {  case 1: {  int val;  printf("Enter Value: ");  scanf("%d", &val);  enqueue(queue, val, size);  break;  }   case 2: {  int delVal = dequeue(queue);  if (delVal != -1)  printf("Deleted Element: %d\n", delVal);  break;  }   case 3:  displayQ(queue);  break;   case 0:  printf("Exiting Program.\n");  free(queue);  exit(0);   default:  printf("Invalid Choice. Please try again.\n");  }  }   return 0; } |

**Question 34**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to perform different queue operations such as insert, delete, display of elements using circular queue #include<stdio.h> #include <stdlib.h>  int front = -1, rear = -1;  int enqueue(int\* queue, int val, int size){  if((rear+1) % size == front){  printf("Quere Overflow!");  return 0;  }  else{  if(front == -1)  front = 0;  rear = (rear + 1) % size;  queue[rear] = val;  }  return 1; }  int dequeue(int\* queue, int size){  if(front == -1){  printf("Queue Underflow!");  return -1;  }  else{  int val = queue[front];  if(front == rear){  front = -1;  rear = -1;  }  else{  front = (front + 1) % size;  }  printf("Element Inserted Successfully!");  return val;  } }  void displayQ(int\* queue, int size) {  if(front == -1) {  printf("Queue is empty.\n");  }   else{  printf("Queue elements: ");  int i = front;  while(1) {  printf("%d ", queue[i]);  if(i == rear)  break;  i = (i + 1) % size;  }  printf("\n");  } }  int main(){  int size;  printf("Enter Size of Queue: ");  scanf("%d", &size);   int\* queue = (int\*)malloc(size \* sizeof(int));  if (!queue) {  printf("Memory allocation failed!\n");  return 1;  }    int choice;  while(1){  printf("\n1) Enqueue Element: ");  printf("\n2) Dequeue Element: ");  printf("\n3) Display Elements: ");  printf("\n0) Exit: ");  printf("\n\n Enter Choice: ");  scanf("%d", &choice);   switch(choice){  case 1: {  int val;  printf("Enter Value: ");  scanf("%d", &val);  enqueue(queue, val, size);  break;  }    case 2: {  int delVal = dequeue(queue, size);  if(delVal != -1)  printf("Deleted Element: %d", delVal);  break;  }    case 3:  displayQ(queue, size);  break;    case 0:  printf("Exiting Program.");  exit(0);  }  } } |

**Question 35**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to perform stack operations (push, pop, peek, display) using a dynamic stack (linked list) #include <stdio.h> #include <stdlib.h>  typedef struct ListNode {  int data;  struct ListNode\* next; } NODE;  NODE\* createNode(int val) {  NODE\* newNode = (NODE\*)malloc(sizeof(NODE));  newNode->data = val;  newNode->next = NULL;  return newNode; }  void push(NODE\*\* head, int val) {  NODE\* newNode = createNode(val);  newNode->next = \*head;  \*head = newNode;  printf("Element %d Pushed Successfully!\n", val); }  int pop(NODE\*\* head) {  if (\*head == NULL) {  printf("Stack Underflow!\n");  return -1;  }   NODE\* temp = \*head;  int val = temp->data;  \*head = (\*head)->next;   free(temp);   printf("Element %d Popped Successfully!\n", val);  return val; }  int peek(NODE\* head) {  if (head == NULL) {  printf("Stack is Empty!\n");  return -1;  }   return head->data; }  void display(NODE\* head) {  if (head == NULL) {  printf("Stack is Empty!\n");  return;  }   NODE\* temp = head;  printf("Stack Elements: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n"); }  int main() {  NODE\* head = NULL;  int choice, val;   while (1) {  printf("\n1) Push Element\n");  printf("2) Pop Element\n");  printf("3) Peek Element\n");  printf("4) Display Stack\n");  printf("0) Exit\n");  printf("\nEnter your choice: ");  scanf("%d", &choice);   switch (choice) {  case 1:  printf("Enter Value to Push: ");  scanf("%d", &val);  push(&head, val);  break;   case 2:  val = pop(&head);  if (val != -1) {  printf("Popped Element: %d\n", val);  }  break;   case 3:  val = peek(head);  if (val != -1) {  printf("Top Element: %d\n", val);  }  break;   case 4:  display(head);  break;   case 0:  printf("Exiting Program.\n");  exit(0);   default:  printf("Invalid Choice! Please try again.\n");  }  }   return 0; } |

**Question 36**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to perform stack operations (enqueue, dequeue, display) using a dynamic queue (linked list) #include <stdio.h> #include <stdlib.h>  typedef struct ListNode {  int data;  struct ListNode\* next; } NODE;  // Function to create a new node NODE\* createNode(int val) {  NODE\* newNode = (NODE\*)malloc(sizeof(NODE));  newNode->data = val;  newNode->next = NULL;  return newNode; }  // Enqueue operation int enqueue(NODE\*\* front, NODE\*\* rear, int val) {  NODE\* newNode = createNode(val);   if (\*rear == NULL) {  \*front = newNode;  \*rear = newNode;  } else {  (\*rear)->next = newNode;  \*rear = newNode;  }   printf("Element %d inserted successfully!\n", val);  return 1; }  // Dequeue operation int dequeue(NODE\*\* front, NODE\*\* rear) {  if (\*front == NULL) {  printf("Queue Underflow!\n");  return -1;  }   int val = (\*front)->data;  NODE\* temp = \*front;  \*front = (\*front)->next;   if (\*front == NULL)  \*rear = NULL;   free(temp);  return val; }  // Display queue void display(NODE\* head) {  if (head == NULL) {  printf("Queue is Empty!\n");  return;  }   NODE\* temp = head;  printf("Queue Elements: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n"); }  // Main menu-driven system int main() {  NODE\* front = NULL;  NODE\* rear = NULL;  int choice, val;   while (1) {  printf("\n====== Queue Operations Menu ======\n");  printf("1) Enqueue Element\n");  printf("2) Dequeue Element\n");  printf("3) Display Queue\n");  printf("0) Exit\n");  printf("Enter your choice: ");  scanf("%d", &choice);   switch (choice) {  case 1:  printf("Enter value to insert: ");  scanf("%d", &val);  enqueue(&front, &rear, val);  break;   case 2:  val = dequeue(&front, &rear);  if (val != -1)  printf("Deleted Element: %d\n", val);  break;   case 3:  display(front);  break;   case 0:  printf("Exiting program. Goodbye!\n");  exit(0);   default:  printf("Invalid choice. Please try again.\n");  }  }   return 0; } |

**Question 37**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to implement Priority Queue with 3 priority values as -- 1: Low, 2: Medium, 3: High #include <stdio.h> #include <stdlib.h>  #define MAX 100  typedef struct {  int data;  int priority; // 1: Low, 2: Medium, 3: High } Element;  Element queue[MAX]; int size = 0;  void enqueue(int data, int priority) {  if (size == MAX) {  printf("Queue Overflow!\n");  return;  }   queue[size].data = data;  queue[size].priority = priority;  size++;   printf("Element %d with priority %d inserted successfully!\n", data, priority); }  void dequeue() {  if (size == 0) {  printf("Queue Underflow!\n");  return;  }   int highest = -1;  int index = -1;  for (int i = 0; i < size; i++) {  if (queue[i].priority > highest) {  highest = queue[i].priority;  index = i;  }  }   printf("Dequeued Element: %d (Priority: %d)\n", queue[index].data, queue[index].priority);  for (int i = index; i < size - 1; i++) {  queue[i] = queue[i + 1];  }  size--; }  void display() {  if (size == 0) {  printf("Queue is empty.\n");  return;  }   printf("Queue Elements [Data (Priority)]:\n");  for (int i = 0; i < size; i++) {  printf("%d (%d) ", queue[i].data, queue[i].priority);  }  printf("\n"); }  int main() {  int choice, data, priority;   while (1) {  printf("\n----- Priority Queue Menu -----\n");  printf("1) Enqueue Element\n");  printf("2) Dequeue Element\n");  printf("3) Display Queue\n");  printf("0) Exit\n");  printf("Enter choice: ");  scanf("%d", &choice);   switch (choice) {  case 1:  printf("Enter value: ");  scanf("%d", &data);  printf("Enter priority (1: Low, 2: Medium, 3: High): ");  scanf("%d", &priority);  enqueue(data, priority);  break;   case 2:  dequeue();  break;   case 3:  display();  break;   case 0:  printf("Exiting program.\n");  exit(0);   default:  printf("Invalid choice. Try again.\n");  }  }   return 0; } |

**Question 38**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to find factorial of a number using recursion #include<stdio.h>  int factorial(int num){  if(num == 0 || num == 1){  return 1;  }  return num \* factorial(num - 1); }  int main(){  int num;  printf("Enter Number: ");  scanf("%d", &num);   int fac = factorial(num);  printf("Factorial of %d is %d", num, fac);   return 0; } |

**Question 39**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to find Fibonacci Series of n terms; #include<stdio.h>  void fibonacci(int a, int b, int n){  if(n == 0){  return;  }  printf("%d ", a);  fibonacci(b, a+b, n-1); }  int main(){  int num;  printf("Enter Number of Elements: ");  scanf("%d", &num);   printf("Fibonacci Series:\n");  fibonacci(0, 1, num);   return 0; } |

**Question 40**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to calculate GCD of two numbers using recrusion #include<stdio.h>  int gcd(int a, int b) {  if (b == 0)  return a;  return gcd(b, a % b); }  int main(){  int a, b;  printf("Enter First Number: ");  scanf("%d", &a);  printf("Enter Second Number: ");  scanf("%d", &b);   int result = gcd(a,b);  printf("Greatest Common Divisor of %d and %d is %d.", a, b, result);   return 0; } |

**Question 41**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| // WAP to create a Binary tree and traverse the tree in Inorder, Preorder and Postorder manner using recursive functions #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 inorderTraversal(struct Node\* root) {  if (root == NULL)  return;  inorderTraversal(root->left);  printf("%d ", root->data);  inorderTraversal(root->right); }  void preorderTraversal(struct Node\* root) {  if (root == NULL)  return;  printf("%d ", root->data);  preorderTraversal(root->left);  preorderTraversal(root->right); }  void postorderTraversal(struct Node\* root) {  if (root == NULL)  return;  postorderTraversal(root->left);  postorderTraversal(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: ");  inorderTraversal(root);  printf("\n");   printf("Preorder Traversal: ");  preorderTraversal(root);  printf("\n");   printf("Postorder Traversal: ");  postorderTraversal(root);  printf("\n");   return 0; } |

**Question 42**

|  |
| --- |
| **Source Code** |

|  |
| --- |
| /\*WAP to implement following recursive operations on a Binary Search Tree (BST)  a. Find an element b. Insert an element c. Delete an element  d. Count the number of nodes e. Find maximum element  f. Find minimum element  g. Find height of the tree\*/  #include<stdio.h> #include<stdlib.h>  struct Node {  int data;  struct Node\* left;  struct Node\* right; };  // Function to create a new node struct Node\* createNode(int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->left = newNode->right = NULL;  return newNode; }  // Insert an element into BST 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; }  // Find an element in BST struct Node\* find(struct Node\* root, int value) {  if (root == NULL || root->data == value)  return root;  if (value < root->data)  return find(root->left, value);  else  return find(root->right, value); }  // Find minimum value node struct Node\* findMin(struct Node\* root) {  if (root == NULL || root->left == NULL)  return root;  return findMin(root->left); }  // Find maximum value node struct Node\* findMax(struct Node\* root) {  if (root == NULL || root->right == NULL)  return root;  return findMax(root->right); }  // Delete an element from BST struct Node\* deleteNode(struct Node\* root, int value) {  if (root == NULL)  return root;    if (value < root->data)  root->left = deleteNode(root->left, value);  else if (value > root->data)  root->right = deleteNode(root->right, value);  else {  // Node with one child or no child  if (root->left == NULL) {  struct Node\* temp = root->right;  free(root);  return temp;  }  else if (root->right == NULL) {  struct Node\* temp = root->left;  free(root);  return temp;  }  // Node with two children  struct Node\* temp = findMin(root->right);  root->data = temp->data;  root->right = deleteNode(root->right, temp->data);  }  return root; }  // Count number of nodes int countNodes(struct Node\* root) {  if (root == NULL)  return 0;  return 1 + countNodes(root->left) + countNodes(root->right); }  // Find height of BST int height(struct Node\* root) {  if (root == NULL)  return -1; // height of empty tree is -1  int leftHeight = height(root->left);  int rightHeight = height(root->right);  return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1; }  // Inorder Traversal (to view the tree) void inorderTraversal(struct Node\* root) {  if (root == NULL)  return;  inorderTraversal(root->left);  printf("%d ", root->data);  inorderTraversal(root->right); }  int main(){  struct Node\* root = NULL;  int choice, value;  struct Node\* temp;   while (1) {  printf("\n--- Binary Search Tree Menu ---\n");  printf("1. Insert an element\n");  printf("2. Find an element\n");  printf("3. Delete an element\n");  printf("4. Count number of nodes\n");  printf("5. Find maximum element\n");  printf("6. Find minimum element\n");  printf("7. Find height of the tree\n");  printf("8. Display Inorder Traversal\n");  printf("9. Exit\n");  printf("Enter your choice: ");  scanf("%d", &choice);   switch(choice) {  case 1:  printf("Enter value to insert: ");  scanf("%d", &value);  root = insert(root, value);  break;  case 2:  printf("Enter value to find: ");  scanf("%d", &value);  temp = find(root, value);  if (temp != NULL)  printf("Element %d found in the BST.\n", value);  else  printf("Element %d not found in the BST.\n", value);  break;  case 3:  printf("Enter value to delete: ");  scanf("%d", &value);  root = deleteNode(root, value);  printf("Element %d deleted (if it existed).\n", value);  break;  case 4:  printf("Total number of nodes: %d\n", countNodes(root));  break;  case 5:  temp = findMax(root);  if (temp != NULL)  printf("Maximum element: %d\n", temp->data);  else  printf("Tree is empty.\n");  break;  case 6:  temp = findMin(root);  if (temp != NULL)  printf("Minimum element: %d\n", temp->data);  else  printf("Tree is empty.\n");  break;  case 7:  printf("Height of the tree: %d\n", height(root));  break;  case 8:  printf("Inorder Traversal: ");  inorderTraversal(root);  printf("\n");  break;  case 9:  printf("Exiting program.\n");  return 0;  default:  printf("Invalid choice! Please try again.\n");  }  } } |