|  |  |
| --- | --- |
| **Name of the student** | **Amit Kumar** |
| **Roll No** | **220** |
| **USN** | **01FE22BEC045** |
| **Div** | **B** |

**Part- A**

|  |
| --- |
| **Variables and Data Types** |
| 1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate. |
| A. Variable: A variable is the title of a reserved region allocated in memory. In other words, it may be referred to as the name of a memory location.  Role: It stores information and can hold different values during program execution.  Example: int age = 25; Here, age is a variable of type integer that stores the value 25.  Data Type: Data types specify the different sizes and values that can be stored in the variable. Role: It restricts the kind of data that can be stored in a variable.  Example: char grade = 'A'; In this case, char is the data type specifying that the variable grade can only store a single character. |
| 2. Explain the concept of data types in C programming. Discuss the different types of data types available in C. |
| A. Data types are fundamental building blocks in C programming. They define the type of data a variable can hold, which determines:  Size: The amount of memory allocated to store the variable's value.  Range: The set of valid values the variable can hold (e.g., integers within a specific range, characters).  Operations: The allowed operations that can be performed on the data (e.g., addition for integers, concatenation for strings).  Different Types of Data Types in C:  Primitive Data Types:  Integer (int): Stores whole numbers without decimal points.  Character (char): Represents ASCII characters within single quotes.  Floating-point (float): Holds real number values with decimal points.  Double (double): Used for larger numeric values than float.  Void: Represents an empty type with no value.  Derived Data Types:  Union, Structure, Array: Data types derived from fundamental data types.  Pointers: Variables that store memory addresses.  Function Types: Represent the type of functions.  Data Type Modifiers:  Signed, Unsigned: Modify the storage of data types to allow for positive, negative, or only positive values.  Short, Long: Alter the size or length of data types.  User-Defined Data Types:  Defined by the programmer to create custom data structures. |
| 3.How are variables declared and initialized in C programming? Provide examples of variable declarations with different data types |
| A.  int Integer int age = 30;  float Single-precision floating-point number float pi = 3.14159;  double Double-precision floating-point number double distance = 100.5;  char Character (single ASCII code) char initial = 'B';  bool Boolean (true or false) bool isStudent = true;  array Collection of elements of the same type int numbers[5] = {1, 2, 3, 4, 5}; |
| 4. Discuss the scope and lifetime of variables in C programming. What are global and local variables? |
| A. Scope refers to the visibility of a variable within the program.  Lifetime refers to the duration for which a variable exists in memory during program execution.  Global variables have a global scope and exist throughout the program's execution.  Local variables have a limited scope and exist only within the block in which they are declared. |
| 5. Explain the concept of type casting in C programming. When is type casting necessary, and how is it performed? |
| A. Type casting in C programming refers to the process of converting a value from one data type to another. It's necessary when operations involve different data types or when assigning a value of one type to a variable of another type. Type casting is performed by placing the desired data type in parentheses before the variable or expression to be converted. For example, `(float)x` would cast the integer variable `x` to a floating-point value. |
| 6.Describe the purpose and usage of the ternary conditional operator (?:) in C programming. Provide an example demonstrating its usage |
| A. Purpose: A function prototype in C serves to declare the function before its actual implementation. It includes the function's name, return type, and parameter types. This informs the compiler about the function's signature for type checking during compilation.  Structure: return\_type function\_name(parameter\_type1, parameter\_type2, ...);  Necessity: It's necessary to declare function prototypes to catch errors in function calls, such as passing wrong arguments. It allows defining functions in any order within or across source files. |
| 7.Discuss the bitwise operators available in C programming. Explain their usage with suitable examples. |
| A. Call by Value: A copy of the parameter's value is passed to the function. Modifications inside the function don't affect the actual parameters.  Example:  void changeValue(int x) {  x = 10;  }  Call by Reference: The memory address of the parameter is passed. Changes inside the function affect the actual parameters.  Example:  void changeValue(int \*x) {  \*x = 10;  } |
| 8. Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate |
| A. Concept: Recursion is a technique where a function calls itself. It solves problems by breaking them into smaller, similar subproblems.  Example:  int factorial(int n) {  if (n == 0 || n == 1)  return 1;  else  return n \* factorial(n - 1);  }  Explanation: The factorial function calls itself with a smaller argument until reaching a base case. Each call contributes to the final result. |
| 9. What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions? |
| A. It's a conditional operator that evaluates an expression and returns one value if the expression is true and another value if the expression is false.  Syntax: condition ? value\_if\_true : value\_if\_false  Example:  int x = 10;  int result = (x > 5) ? 1 : 0; |
| 10. Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation. |
| A. Bitwise operators perform operations on individual bits of integer operands.  Examples include AND (&), OR (|), XOR (^), left shift (<<), right shift (>>), etc.  Example:  int a = 5; // 101 in binary  int b = 3; // 011 in binary  int result = a & b; // Bitwise AND: 001 |
| 11. Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement? |
| A. Postfix increment (x++) increments the value of x after it's been used in the expression.  Prefix increment (++x) increments the value of x before it's used in the expression.  Example:  int x = 5;  int y = x++; // y will be 5, x will be 6  int z = ++x; // z will be 7, x will be 7 |
| 12. Explain the concept of nested control structures in C programming. Provide an example demonstrating nested if-else statements. |
| A. A:-The logical AND (&&) and logical OR (||) operators in C programming are crucial for constructing conditional expressions, allowing control of program flow based on multiple conditions.  && returns true only if both operands are true; otherwise, it returns false.  || returns true if at least one operand is true; otherwise, it returns false.  They are used in conditional statements like if, while, and for, specifying conditions that must be met or defining alternatives. Example:  if (x > 0 && y < 10) {  // Code block executes if both x > 0 and y < 10  }  if (grade == 'A' || grade == 'B') {  // Code block executes if grade is 'A' or 'B'  } |
| 13. Discuss the role of the break and continue statements in loop control in C programming. Provide examples to illustrate their usage |
| A.Operator precedence and associativity dictate the order of evaluation in expressions.  Precedence determines which operators are evaluated first.  Associativity defines the order of evaluation for operators of the same precedence.  Example:  int result1 = a + b \* c; // Precedence: multiplication (\*) is evaluated first  int result2 = a = b = c; // Associativity: right-to-left for assignment (=)  In result1, multiplication is evaluated before addition due to precedence. In result2, the assignment operator is evaluated right-to-left due to associativity. |
| 14. What are the advantages of using the for loop over the while loop in C programming? Provide examples comparing the two. |
| A.The switch statement allows a program to evaluate an expression and execute different blocks of code based on its value.  It differs from the if-else statement by providing a more concise way to handle multiple conditions.  Example:  int choice;  switch (choice) {  case 1:  // Code block for choice 1  break;  case 2:  // Code block for choice 2  break;  default:  // Default code block  break;  } |
| 15. Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements? |
| A:-Nested control structures are control structures placed within other control structures.  Example:  if (condition1) {  if (condition2) {  // Code block  } else {  // Code block  }  } else {  // Code block  } |
| 16. Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes? |
| A:-The break statement is used to terminate the loop or switch statement and transfer control to the statement immediately following it.  The continue statement is used to skip the remaining code in the loop and move to the next iteration.  Example:for (int i = 0; i < 10; i++) {  if (i == 5) {  continue; // Skip iteration when i is 5  }  if (i == 8) {  break; // Exit the loop when i is 8  }  // Code block  } |
| 17. Explain the difference between call by value and call by reference in C programming. Provide examples to illustrate both concepts. |
| A:-The for loop is typically used when the number of iterations is known before the loop starts.  It provides a compact way to write loops by combining initialization, condition, and increment/decrement in a single line.  Example:for (int i = 0; i < 5; i++) {  // Code block  } |
| 18. Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works. |
| A.Short-circuit evaluation is a mechanism where the evaluation of logical expressions stops as soon as the result is determined.  In an if statement with logical AND (&&) or logical OR (||) operators, if the left operand determines the result, the right operand is not evaluated.  Example:  if (x > 0 && y > 0) {  // Code block  }  In the above example, if x is not greater than 0, y > 0 won't be evaluated because the result of the expression is already determined to be false. |
| 19. What is the significance of the return statement in C programming? How are values returned from functions? |
| A:-The return statement in C programming allows functions to provide a result back to the calling code. Values are returned using the return keyword, matching the function's declared return type. It exits the function and passes control back to the caller along with the returned value.  int add(int a, int b) {  return a + b; // Returns the sum of a and b  }  int result = add(3, 5); // Calls the add function and assigns its return value (8) to result |
| 20. Describe the role of function parameters and arguments in C programming. How are function arguments passed to parameters? |
| A:-Function parameters in C define the types and order of data expected by a function, while arguments are the actual values passed to the function. Arguments are passed to parameters using "pass by value," where the value of each argument is copied into the corresponding parameter.  #include <stdio.h>  // Function prototype with parameters  void greet(char name[]) {  printf("Hello, %s!\n", name);  }  int main() {  char myName[] = "John";  // Function call with arguments  greet(myName); // "myName" is passed as an argument to the "name" parameter  return 0;  } |
| 21. Explain the concept of arrays in C programming. How are arrays declared and initialized? |
| A:-Arrays in C are a collection of elements of the same data type stored in contiguous memory locations. They provide a way to store multiple values of the same type under a single variable name.  Declaration and Initialization:  Declaration: type array\_name[size];  Initialization:  Static Initialization: type array\_name[size] = {value1, value2, ...};  Dynamic Initialization: type array\_name[size]; array\_name[index] = value; |
| 22.Discuss the difference between a one-dimensional array and a multi-dimensional array in C programming. Provide examples of both. |
| A:-Arrays:  One-dimensional Array: It's a linear collection of elements stored in a single row or column.  int arr1D[5] = {1, 2, 3, 4, 5};  Multi-dimensional Array: It's an array of arrays, organized in rows and columns.  int arr2D[2][3] = {{1, 2, 3}, {4, 5, 6}}; |
| 23. Describe the process of accessing array elements in C programming. How are array indices used to access elements? |
| A:-Process: Array elements are accessed using their indices.  Syntax: array\_name[index];  Indices: Start from 0 for the first element and go up to size - 1. |
| 24. What is the significance of the null character ('\0') in C strings? How is it used to determine the end of a string? |
| A:-Purpose: The null character ('\0') marks the end of a string in C. It indicates where the string's contents end in memory.  Usage: Functions that work with strings in C, like printf and strlen, use the null character to determine the string's length and where it ends. |
| 25. Explain the concept of dynamic memory allocation for arrays in C programming. How are dynamic arrays allocated and deallocated? |
| A:-Concept: Dynamic memory allocation allows the creation of arrays whose size is determined during runtime.  Functions: In C, malloc(), calloc(), and realloc() are used to allocate dynamic memory for arrays.  Allocation: ptr = (castType\*) malloc(size \* sizeof(type));  Deallocation: free(ptr);  Example:  int \*arr;  int size = 5;  arr = (int\*)malloc(size \* sizeof(int));  Note: Dynamic memory must be deallocated using free() to avoid memory leaks. |
| 26. Describe the purpose and usage of pointers in C programming. How are pointers declared and initialized? |
| A:-Purpose: Pointers are variables that store memory addresses. They are used to store addresses of variables or arrays, enabling dynamic memory allocation and efficient manipulation of data structures.  Usage: Pointers are extensively used in tasks like dynamic memory allocation, passing parameters by reference, and building complex data structures like linked lists, trees, etc.  Declaration and Initialization:  Declaration: type \*ptr\_name;  Initialization: type \*ptr\_name = &variable; |
| 27. Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers. |
| A:-Concept: Pointer arithmetic involves adding or subtracting an integer from a pointer, which moves it to a different memory location.  Examples:  int arr[5] = {1, 2, 3, 4, 5};  int \*ptr = arr;  ptr++; // Moves pointer to the next element in the array  ptr--; // Moves pointer back to the previous element  EX;  int arr[5] = {10, 20, 30, 40, 50};  int \*ptr = arr; // Points to the first element of the array  ptr = ptr + 2; // Moves pointer to the third element  printf("Value at new pointer position: %d\n", \*ptr); // Output: Value at new pointer position: 30 |
| 28. Discuss the difference between pass by value and pass by reference in function arguments using pointers in C programming. Provide examples to illustrate both approaches. |
| A:-Pass by Value:  - Function receives a copy of the argument's value.  - Modifications inside the function do not affect the original value.  - Example:  void incrementByValue(int x) {  x++;  }  int main() {  int num = 5;  incrementByValue(num);  // num remains 5  }  Pass by Reference using Pointers:  - Function receives the memory address of the argument.  - Modifications inside the function directly affect the original value.  - Example:  void incrementByReference(int \*x) {  (\*x)++;  }  int main() {  int num = 5;  incrementByReference(&num);  // num becomes 6  }  In pass by reference, we use pointers to pass the address of the variable, allowing direct modification of its value within the function. |
| 29. Describe the concept of NULL pointers in C programming. How are NULL pointers used and checked for in programs? |
| A:In C programming, a NULL pointer is a pointer that does not point to any memory location. It is represented by the value 0 or by the macro NULL.  Usage of NULL Pointers  1. Initialization: NULL pointers are commonly used to initialize pointers before they are assigned a valid memory address.  2. ndication of Absence: They are used to indicate that a pointer does not currently point to a valid object or memory location.  Checking for NULL Pointers:  1. Equality Check: To check if a pointer is NULL, you simply compare it with NULL or the integer 0.  if (ptr == NULL) {  // Pointer is NULL  }  2. Dereferencing Safeguard: Before dereferencing a pointer, it's a good practice to check if it's NULL to avoid accessing invalid memory.  if (ptr != NULL) {  // Pointer is not NULL, safe to dereference  printf("Value at pointer: %d\n", \*ptr);  }  - NULL pointers are checked for equality with NULL or the integer 0 to determine if they are valid or not before dereferencing them. |
| 30. Explain the role of pointers in dynamic memory allocation in C programming. How are pointers used to allocate and deallocate memory dynamically? |
| A:-Pointers in Dynamic Memory Allocation in C:  In Cprogramming, pointers are vital for dynamic memory allocation, where memory is allocated and deallocated during program execution. Here's a concise overview:  1. Allocation of Memory:  - Pointers are used to request memory dynamically using functions like `malloc()`, `calloc()`, or `realloc()`.  int \*ptr;  ptr= (int \*)malloc(10 \* sizeof(int)); // Allocating memory for an array of 10 integer  2. Deallocation of Memory:  - Pointers are used to release dynamically allocated memory to avoid memory leaks.  free(ptr); // Deallocating the memory pointed to by ptR  Usage Example:  #include <stdio.h>  #include <stdlib.h>  int main() {  int \*ptr;  int size = 5;  // Dynamically allocate memory for an array of integers  ptr = (int \*)malloc(size \* sizeof(int));  if (ptr == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  // Use the dynamically allocated memory  for (int i = 0; i < size; i++) {  ptr[i] = i + 1;  }  // Print the values stored in the dynamically allocated memory  for (int i = 0; i < size; i++) {  printf("%d ", ptr[i]);  }  printf("\n");  // Free the dynamically allocated memory  free(ptr);  return 0;  }  In summary, pointers enable dynamic memory allocation in C, allowing programs to manage memory efficiently by requesting and releasing memory as needed during runtime. |
| 31. Discuss the concept of strings in C programming. How are strings represented and manipulated in C? |
| A:-In C programming, a NULL pointer is a pointer that doesn't point to any memory location. It's commonly used to signify that a pointer doesn't currently reference any valid object.  - Initialization: NULL pointers are often used to initialize pointers before assigning them valid memory addresses.  - Sentinel Values: They're used as sentinel values to indicate the end of data structures like linked lists or arrays.  - Error Handling: NULL pointers are checked to prevent dereferencing errors, avoiding crashes or undefined behavior.  - Checking for NULL Pointers:  - Prior to dereferencing a pointer, it's checked against NULL to ensure it's pointing to valid memory.  - Commonly done using an `if` statement or implicitly through operations that depend on pointer validity.  Example:  int \*ptr = NULL; // Initialization  if (ptr == NULL) {  // Handle the case where ptr is NULL  } else {  // Proceed with using ptr safely  }  By using NULL pointers and checking for them, C programs can handle memory safely and avoid unexpected crashes. |
| 32. Explain the difference between character arrays and string literals in C programming. Provide examples to illustrate both concepts. |
| A:-Certainly!  Character Arrays:  - Character arrays in C are sequences of characters stored in contiguous memory locations.  - Each element of the array holds a single character.  - They are mutable, meaning their contents can be modified after initialization.  - They must be terminated with a null character (`'\0'`) to indicate the end of the string.  - Example:  char tr1[10] = {'H', 'e', 'l', 'l', 'o', '\0'};  char str2[10];  strcpy(str2, "World"); // Using a string literal to initialize the array  String Literals:  - String literals are sequences of characters enclosed in double quotes.  - They are stored as null-terminated character arrays by the compiler.  - They are immutable, meaning their contents cannot be modified after compilation.  - They are automatically null-terminated by the compiler.  - Example:  char \*str3 = "Hello"; // Pointer to a string literal  char str4[] = "World"; // Character array initialized with a string literal  Key Differences:  - Mutability: Character arrays are mutable, allowing modifications to their contents. String literals are immutable and cannot be modified.  - Termination: Character arrays require explicit termination with a null character (`'\0'`). String literals are automatically null-terminated by the compiler.  - Usage: Character arrays are often used when mutable strings are required, while string literals are used for constant strings that do not need to be modified.  Understanding these differences is essential for efficient string handling and memory management in C programming. |
| 33. Describe common string manipulation functions available in the C standard library. Provide examples of functions like strlen, strcpy, strcat, and strcmp. |
| A:-Sure, here's a more concise overview:  1. strlen():  - Description: Calculates the length of a string (excluding the null terminator).  - Prototype: `size\_t strlen(const char \*str);`  - Example:  char str[] = "Hello";  size\_t length = strlen(str); // length = 5  2. strcpy():  - Description: Copies the contents of one string to another.  - Prototype: `char \*strcpy(char \*dest, const char \*src);`  - Example:  char src[] = "Hello";  char dest[10];  strcpy(dest, src); // dest = "Hello"  3. strcat();  - Description: Appends the content of the source string to the destination string.  - Prototype: `char \*strcat(char \*dest, const char \*src);`  - Example  char dest[20] = "Hello";  char src[] = " World";  strcat(dest, src); // dest = "Hello World"  4. strcmp():  - Description: Compares two strings lexicographically.  - Prototype: `int strcmp(const char \*str1, const char \*str2);`  - Example:  char str1[] = "Hello";  char str2[] = "Hello";  int result = strcmp(str1, str2); // result = 0 (equal)  These functions are commonly used for basic string operations in C programming, offering efficient ways to handle and manipulate strings. |
| 34. Discuss the concept of string tokenization in C programming. How are strings split into tokens using delimiter characters? |
| A:-String tokenization is the process of breaking a string into smaller pieces, known as tokens, based on specified delimiter characters. This concept is often used in C programming to parse strings and extract relevant information. Here's a more detailed explanation  1.Using `strtok()` FunctioN:  - C provides the `strtok()` function in the standard library for string tokenization.  - It takes two arguments: the string to be tokenized and a string containing delimiter characters.  - The function returns a pointer to the next token found in the string, or `NULL` if no more tokens are found.  2. Steps for String Tokenization:  - Call `strtok()` with the input string and delimiter string.  - The first call to `strtok()` returns a pointer to the first token found in the input string.  - Subsequent calls to `strtok()` with `NULL` as the first argument will continue tokenizing the string until no more tokens are found.  - Each call to `strtok()` modifies the input string by replacing delimiter characters with null characters (`'\0'`) to separate tokens.  3. Example:  #include <stdio.h>  #include <string.h>  int main() {  char str[] = "apple,banana,grape";  char \*token = strtok(str, ",");  while (token != NULL) {  printf("%s\n", token);  token = strtok(NULL, ",");  }  return 0;  }    Output:    apple  banana  grape  String tokenization is a fundamental technique for processing textual data, such as parsing command-line arguments, reading input from files, or extracting fields from structured data formats like CSV (comma-separated values). |
| 35. Explain the importance of null-terminated strings in C programming. How does the null character ('\0') signify the end of a string? |
| A:-Null-terminated strings are crucial in C programming because they serve as a fundamental way to represent and manipulate strings. The null character (`'\0'`) marks the end of a string, allowing C functions to determine the length of a string and perform string operations safely. Here's a brief overview of their importance:  1. String Length: Null-terminated strings enable functions like `strlen()` to determine the length of a string by iterating through its characters until the null character is encountered.  2. String Manipulation: Functions like `strcpy()`, `strcat()`, and `strcmp()` rely on the null character to perform string operations effectively and safely.  3. Memory Management: Null-terminated strings facilitate efficient memory management in C programs, allowing for the use of statically allocated character arrays or dynamically allocated memory for strings  4. Interoperability: Many C library functions and system calls expect null-terminated strings as input parameters, making them essential for interfacing with existing libraries and systems.  5. Compatibility: Null-terminated strings are widely used in C libraries, APIs, and protocols, ensuring compatibility and interoperability across different C programs.  Overall, null-terminated strings provide a simple and effective way to represent and manipulate textual data in C programming, making them a fundamental aspect of the language. |
| 36. Describe the purpose and usage of structures in C programming. How are structures declared and accessed? |
| A:-Purpose and Usage of Structures in C Programming:  - Purpose: Structures allow you to group different data types under a single name. They are used to create complex data types that can hold multiple pieces of related information.  - Usage: Structures are commonly used to represent real-world entities or complex data types such as employees, students, records, etc. They provide a way to organize related data elements into a single unit.  Declaration and Access of Structures:  - Declaration: Structures are declared using the `struct` keyword followed by the structure tag and a list of member variables inside curly braces.  struct Person {  char name[50];  int age;  float height;  };  - Access: You can access structure members using the dot (`.`) operator.  struct Person person1;  strcpy(person1.name, "John");  person1.age = 30;  person1.height = 5.9;  Brief Explanation:  - \*\*Purpose\*\*: Structures help organize related data items into a single unit for easier management and manipulation.  - \*\*Usage\*\*: They are extensively used in applications where complex data structures are needed, such as in databases, file systems, and graphical user interfaces.  - \*\*Declaration\*\*: Structures are declared using the `struct` keyword followed by a name and a list of member variables inside curly braces.  - \*\*Access\*\*: Structure members are accessed using the dot (`.`) operator followed by the member name.  In essence, structures provide a way to group related data elements together under a single name, making it easier to manage and manipulate complex data in C programming. |
| 37. Discuss the concept of structure members in C programming. How are individual members of a structure accessed and modified? |
| A:-Purpose and Usage of Structures in C Programming:  - Purpose. Structures allow you to group different data types under a single name. They are used to create complex data types that can hold multiple pieces of related information.  - Usage: Structures are commonly used to represent real-world entities or complex data types such as employees, students, records, etc. They provide a way to organize related data elements into a single unit.  Declaration and Access of Structures:  - Declaration: Structures are declared using the `struct` keyword followed by the structure tag and a list of member variables inside curly braces.  struct Person {  char name[50];  int age;  float height;  };  - Access: You can access structure members using the dot (`.`) operator.  struct Person person1;  strcpy(person1.name, "John");  person1.age = 30;  person1.height = 5.9;  Brief Explanation:  - Purpose: Structures help organize related data items into a single unit for easier management and manipulation.  - Usage: They are extensively used in applications where complex data structures are needed, such as in databases, file systems, and graphical user interfaces.  - Declaration: Structures are declared using the `struct` keyword followed by a name and a list of member variables inside curly braces.  - Access: Structure members are accessed using the dot (`.`) operator followed by the member name. |
| 38. Explain the difference between structures and unions in C programming. When would you choose one over the other? |
| A:-Structures vs Unions in C:  Structures:  - You use structures to group different kinds of data under one name.  - Each part of the structure has its own space in memory.  - You can access each part of the structure independently.  Unions:  - Unions also group different data types together.  - However, all parts of a union share the same memory space.  - Only one part of the union can be used at a time.When to Use Each:  - Structures are good when you need to store various types of data and access them separately.  - Unionsare useful when you want to save memory and only use one type of data at a time.  In simpler terms, structures let you keep different kinds of data organized, while unions let you save space by sharing memory between different types of data. Choose based on what you need for your program. |
| 39. Describe the concept of nested structures in C programming. How are structures within structures defined and accessed? |
| A:-In C programming, nested structures refer to structures that are defined within other structures. This allows you to create complex data structures where one structure contains another as a member.  Definition of Nested Structures:  Nested structures are defined similarly to regular structures, but the member of a structure can itself be another structure. Here's how you define a nested structure:  struct Address {  char street[50];  char city[50];  char state[20];  int zip;  };  struct Person {  char name[50];  int age;  struct Address address; // Nested structure  };  In the above example, the Person structure contains a member named address, which is of type Address structure.  Accessing Nested Structure Members:  You can access members of nested structures using the dot (.) operator. Here's how you access members of a nested structure:  EX:  struct Person person1;  strcpy(person1.name, "John");  person1.age = 30;  strcpy(person1.address.street, "123 Main St");  strcpy(person1.address.city, "Anytown");  strcpy(person1.address.state, "CA");  person1.address.zip = 12345;  In this example, person1.address.street, person1.address.city, person1.address.state, and person1.address.zip are accessed using dot notation just like any other structure member.  Benefits of Nested Structures:  Organized Data: Nested structures help in organizing related data in a hierarchical manner.  Encapsulation: They encapsulate related data together, making the code easier to understand and maintain.  Code Reusability: Nested structures facilitate code reusability as you can define complex data structures using smaller, reusable structures. |
| 40. Discuss the concept of typedef in C programming. How is typedef used to define custom data types, including structures and unions? |
| A:-  In C, `typedef` is used to create custom names for existing data types, including structures and unions. It enhances code readability and simplifies complex declarations.  1. Creating Custom Data Types:  - Syntax: `typedef existing\_type new\_type;`  - Example:  typedef int INT;  typedef struct {  int day;  int month;  int year;  } Date;  2. Simplifying Declarations:  - Example;  typedef int (\*CompareFunction)(int, int);  3. Improving Readability:  - Example:  Date currentDate;  4. Structures and Unions:  - Example:  typedef struct {  char name[50];  int age;  } Person;    typedef union {  int num;  float value;  } Number;  . |
| 41. Explain the concept of file handling in C programming. How are files opened, read from, and written to using standard file handling functions? |
| A:-File Handling in C Programming:  When programming in C, you often need to work with files to read data from them or write data to them. This process is known as file handling. Here's how it works in simple terms:  1. File Pointer:  - To work with a file in C, you use something called a file pointer. Think of it as a special marker that points to where you are in the file.  2. File Modes:  - When you open a file, you can specify how you want to interact with it. For example, you can open it for reading, writing, or both.  3. Opening a File:  - To start working with a file, you need to open it. You do this using the `fopen()` function, which takes the file name and mode as arguments.  FILE \*filePointer;  filePointer = fopen("example.txt", "r")  4. Reading from a File:  - Once the file is open, you can read data from it using functions like `fscanf()` or `fgets()`.  char buffer[100];  fscanf(filePointer, "%s", buffer); // Read a string from the file  5. Writing to a File:  - You can also write data to a file using functions like `fprintf()` or `fputs()`.  fprintf(filePointer, "This is a line written to the file\n");  6. Closing a File:  - After you're done working with a file, it's important to close it using the `fclose()` function to free up system resources.  fclose(filePointer);  Example:  Suppose you have a file named "example.txt" containing some text. Here's how you would read from it using C:  #include <stdio.h>  int main() {  FILE \*filePointer;  char buffer[100];  filePointer = fopen("example.txt", "r");  if (filePointer == NULL) {  printf("File opening failed.");  return 1;  }  fscanf(filePointer, "%s", buffer);  printf("Data read from file: %s\n", buffer);  fclose(filePointer);  return 0;  }  In this example, the program opens "example.txt" for reading, reads a string from it, and then prints that string. Finally, the file is closed. |
| 42. Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files? |
| A:-Role of File Pointers in C Programming:  In C, file pointers are like little guides that help us work with files. Here's how they work:  1. Opening Files  - When we open a file, we get a file pointer that helps us keep track of where we are in the file.  FILE \*filePointer;  filePointer = fopen("example.txt", "r");  2. Navigating within Files:  - We can move around the file using functions like `fseek()` or `rewind()`.  fseek(filePointer, 10, SEEK\_SET); // Move to 10th byte from the beginning  rewind(filePointer); // Move to the beginning  3. Reading and Writing Data:  - File pointers help us read data from files with functions like `fgets()` and write data to files with functions like `fprintf()`.  char buffer[100];  fgets(buffer, 100, filePointer); // Read a line from the file  fprintf(filePointer, "This is a line written to the file\n"); // Write a line  4.Closing Files:  - Finally, we need to close the file using `fclose()` when we're done with it.  fclose(filePointer);  In Short:  - File pointers guide us through files, helping us read, write, and move around efficiently.  - They're essential for managing files in C programs. |
| 43. Discuss the difference between text files and binary files in C programming. How are they opened and processed differently? |
| A:-Difference Between Text Files and Binary Files in C Programming:  Text Files:  - Human-Readable: Text files store data in a way that humans can easily understand, using character encoding like ASCII or UTF-8.  - Line-by-Line: They're processed line by line and opened in text mode (`"r"`, `"w"`, `"a"`, etc.).  - Examples: Documents, configuration files, source code.  Binary Files:  - Raw Data: Binary files store data as raw bytes, not meant for human consumption.  - Byte-by-Byte; They're processed byte by byte and opened in binary mode (`"rb"`, `"wb"`, `"ab"`, etc.).  - Examples: Complex data structures, like arrays or structures.  Opening and Processing:  - Text files are opened and processed differently from binary files due to their distinct formats and processing methods.  - Text files use functions like `fgets()` and `fprintf()` for processing, while binary files use `fread()` and `fwrite()`.  In short, text files are for human-readable data, while binary files are for raw data storage. They're opened and processed differently in C programs. |
| 44. Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc. |
| A:-Purpose of File Modes in C Programming:  File modes in C programming specify how a file should be opened and what operations are allowed on it. They control whether the file is opened for reading, writing, or appending. Here's a simple explanation with examples:  1. "r" (Read Mode):  - Opens the file for reading only.  FILE \*file = fopen("example.txt", "r");  2. "w" (Write Mode):  - Opens the file for writing only. If the file exists, it's truncated; if not, a new file is created.  FILE \*file = fopen("example.txt", "w");  3. "a" (Append Mode):  - Opens the file for writing only. Data is appended to the end of the file. If the file doesn't exist, a new file is created.  FILE \*file = fopen("example.txt", "a");  4. "r+" (Read/Write Mode):  - Opens the file for both reading and writing. The file must exist.  FILE \*file = fopen("example.txt", "r+");  5. "w+" (Read/Write Mode):  - Opens the file for both reading and writing. If the file exists, it's truncated; if not, a new file is created.  FILE \*file = fopen("example.txt", "w+");  6. "a+" (Read/Append Mode):  - Opens the file for both reading and appending. Data is appended to the end of the file. If the file doesn't exist, a new file is created.  FILE \*file = fopen("example.txt", "a+");  These modes provide flexibility in how files are opened and manipulated, allowing for various reading, writing, and appending operations. |
| 45. Describe error handling techniques in file operations in C programming. How are errors detected and handled when working with files? |
| A. Error Handling in C File Operations:  When working with files in C, it's crucial to handle errors properly. Here's how you can do it:  1. Check Return Values:  - Always check if file operations like `fopen()` succeed by examining their return values.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  // Handle error  }  2. Use perror() for Message  - If an error occurs, use `perror()` to print a helpful error message.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  3. Handle Specific Errors:  - Depending on the error, you may want to provide specific messages or actions.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  if (errno == ENOENT) {  printf("File does not exist\n");  } else {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  }  4. Cleanup Resources:  - Make sure to close the file and release resources if an error occurs.  c  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  // File processing code...  fclose(file);  5. Provide Clear Messages:  - Give meaningful error messages to users or log detailed information for debugging.  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file: %s\n", strerror(errno));  exit(EXIT\_FAILURE);  } |

**Part- B**

|  |
| --- |
| **1. Hello World** |
| Code:  #include <stdio.h>  int main()  {  printf("Hello World");  return 0;  } |
| Output |
| **2. Factorial** |
| Code:  #include <stdio.h>  int main()  {  int n, fac = 1;  printf("Enter the number: ");  scanf("%d", &n);  for (int i = 1; i <= n; i++)  {  fac = fac \* i;  }  printf("The Factorial is : %d", fac);  return 0;  } |
| Output |
| **3. Prime Numbers**  Code:  #include <stdio.h>  int main()  {  int n, i;  printf("Enter the value of n: ");  scanf("%d", &n);  int a = 0; // Flag concept  for (i = 2; i < n; i++)  {  if (n % i == 0)  {  a = 1;  break;  }  }  if (n == 1)  {  printf("It is Neither prime nor compositive");  }  else if (a == 0)  {  printf("It is PRIME number.");  }  else  {  printf("It is Composite number.");  }  return 0;  } |
| Output |
| **4. Fibonacci Series**  Code:  #include <stdio.h>  int main()  {  int n, fib3 = 1, fib1 = 1, fib2 = 1;  printf("Enter the number: ");  scanf("%d", &n);  printf("The Fibonacci is : ");  printf("%d %d ", fib1, fib2);  for (int i = 3; i <= n; i++)  {  fib3 = fib1 + fib2;  fib1 = fib2;  fib2 = fib3;  printf("%d ", fib3);  }  return 0;  } |
| Output |
| **5. Sum of Digits**  Code:  #include <stdio.h>  int main()  {  int n, ld, sum = 0;  printf("Enter the number: ");  scanf("%d", &n);  int count = 0;  while (n != 0)  {  ld = n % 10;  sum = sum + ld;  n = n / 10;  count++;  }  printf("The number of digits are: %d\n", count);  printf("The sum of digits are: %d", sum);  return 0;  } |
| Output |
| **6. Reverse a Number**  Code:  #include <stdio.h>  int main()  {  int n, ld = 0, rev = 0;  printf("Enter the number: ");  scanf("%d", &n);  int count = 0;  while (n != 0)  {  ld = n % 10;  rev = rev \* 10;  rev = rev + ld;  n = n / 10;  count++;  }  // rev/10; -->replace rev and rev  printf("The number of digits are: %d\n", count);  printf("The Reverse of digits are: %d", rev);  return 0;  } |
| Output |
| **7. Palindrome Check**  Code:  #include <stdio.h>  int main()  {  int n;  int arr[n];  printf("Enter the value for array you want: ");  scanf("%d", &n);  for (int i = 0; i < n; i++)  {  scanf("%d", &arr[i]);  }  int flag = 0;  for (int i = 0; i < n / 2; i++)  {  for (int j = n - 1; j > i; j--)  {  if (arr[i] == arr[j])  flag = 1;  }  }  if (flag == 0)  printf("It is NOT Palindrone.");  else  printf("It is Palindrone.");  return 0;  } |
| Output |
| **8. Area of Shapes**  Code:  #include <stdio.h>  #include <math.h>  #define PI 3.14159  // Function to calculate the area of a rectangle  float rectangleArea(float length, float width)  {  return length \* width;  }  // Function to calculate the area of a triangle  float triangleArea(float base, float height)  {  return 0.5 \* base \* height;  }  // Function to calculate the area of a circle  float circleArea(float radius)  {  return PI \* radius \* radius;  }  int main()  {  float length, width, base, height, radius;  // Input for rectangle  printf("Enter length and width of the rectangle: ");  scanf("%f %f", &length, &width);  printf("Area of rectangle: %.2f\n", rectangleArea(length, width));  // Input for triangle  printf("Enter base and height of the triangle: ");  scanf("%f %f", &base, &height);  printf("Area of triangle: %.2f\n", triangleArea(base, height));  // Input for circle  printf("Enter radius of the circle: ");  scanf("%f", &radius);  printf("Area of circle: %.2f\n", circleArea(radius));  return 0;  } |
| Output |
| **9. Simple Calculator**  Code:  #include <stdio.h>  int main()  {  int num1, num2;  char op;  printf("Enter the 2 numbers and Operator(+,-,\*,/): ");  scanf("%d %c%d", &num1, &op, &num2);  switch (op)  {  case '+':  int add = num1 + num2;  printf("The Addition of %d and %d is: %d ", num1, num2, add);  break;  case '-':  int sub = num1 - num2;  printf("The Subtraction of %d and %d is: %d ", num1, num2, sub);  break;  case '\*':  int mul = num1 \* num2;  printf("The Multiplication of %d and %d is: %d ", num1, num2, mul);  break;  case '/':  int div = num1 / num2;  if (num2 != 0)  printf("The Division of %d and %d is: %d ", num1, num2, div);  else  printf("Division by Zero error");  break;  default:  printf("Invalid Choice");  break;  }  return 0;  } |
| Output |
| **10. Array Operations**  Code:  #include <stdio.h>  // Function to find the largest element in an array  int findLargest(int arr[], int size)  {  int max = arr[0];  for (int i = 1; i < size; i++)  {  if (arr[i] > max)  {  max = arr[i];  }  }  return max;  }  // Function to find the smallest element in an array  int findSmallest(int arr[], int size)  {  int min = arr[0];  for (int i = 1; i < size; i++)  {  if (arr[i] < min)  {  min = arr[i];  }  }  return min;  }  // Function to find the sum of elements in an array  int findSum(int arr[], int size)  {  int sum = 0;  for (int i = 0; i < size; i++)  {  sum += arr[i];  }  return sum;  }  // Function to find the average of elements in an array  float findAverage(int arr[], int size)  {  int sum = findSum(arr, size);  return (float)sum / size;  }  int main()  {  int arr[] = {10, 20, 30, 40, 50};  int size = sizeof(arr) / sizeof(arr[0]);  // Find the largest element  int largest = findLargest(arr, size);  printf("Largest element in array: %d\n", largest);  // Find the smallest element  int smallest = findSmallest(arr, size);  printf("Smallest element in array: %d\n", smallest);  // Find the sum of elements  int sum = findSum(arr, size);  printf("Sum of elements in array: %d\n", sum);  // Find the average of elements  float average = findAverage(arr, size);  printf("Average of elements in array: %.2f\n", average);  return 0;  } |
| Output |
| 11.**String Operations**: Manipulate strings such as concatenation, copying, and comparison.  Code:  #include <stdio.h>  #include <string.h>  // Function to concatenate two strings  void concatenateStrings(char result[], const char str1[], const char str2[])  {  strcpy(result, str1); // Copy the first string into the result  strcat(result, str2); // Concatenate the second string onto the result  }  // Function to copy one string to another  void copyString(char dest[], const char source[])  {  strcpy(dest, source); // Copy source string into destination  }  // Function to compare two strings  int compareStrings(const char str1[], const char str2[])  {  return strcmp(str1, str2); // Return 0 if equal, positive if str1 > str2, negative if str1 < str2  }  int main()  {  char str1[50] = "Hello";  char str2[50] = " World!";  char result[100];  // Concatenate strings  concatenateStrings(result, str1, str2);  printf("Concatenated string: %s\n", result);  // Copy string  char copiedStr[50];  copyString(copiedStr, str1);  printf("Copied string: %s\n", copiedStr);  // Compare strings  int comparison = compareStrings(str1, str2);  if (comparison == 0)  {  printf("Strings are equal.\n");  }  else if (comparison > 0)  {  printf("First string is greater than second string.\n");  }  else  {  printf("First string is less than second string.\n");  }  return 0;  } |
| Output |
| **12. Linear Search:** Search for an element in an array using linear search  Code:  #include <stdio.h>  // Function to perform linear search  int linearSearch(int arr[], int size, int key)  {  for (int i = 0; i < size; i++)  {  if (arr[i] == key)  {  return i; // Return the index where the key is found  }  }  return -1; // Return -1 if key is not found  }  int main()  {  int arr[] = {10, 20, 30, 40, 50};  int size = sizeof(arr) / sizeof(arr[0]);  int key = 30; // Element to search for  // Perform linear search  int index = linearSearch(arr, size, key);  // Check if the element was found  if (index != -1)  {  printf("Element %d found at index %d\n", key, index);  }  else  {  printf("Element %d not found in the array\n", key);  }  return 0;  } |
| Output |
| **13. Binary Search:** Search for an element in a sorted array using binary search.  Code:  #include <stdio.h>  // Function to perform binary search  int binarySearch(int arr[], int size, int key)  {  int left = 0;  int right = size - 1;  while (left <= right)  {  int mid = left + (right - left) / 2;  // Check if key is present at mid  if (arr[mid] == key)  {  return mid;  }  // If key is greater, ignore left half  if (arr[mid] < key)  {  left = mid + 1;  }  // If key is smaller, ignore right half  else  {  right = mid - 1;  }  }  // If element is not present in array  return -1;  }  int main()  {  int arr[] = {10, 20, 30, 40, 50};  int size = sizeof(arr) / sizeof(arr[0]);  int key = 30; // Element to search for  // Perform binary search  int index = binarySearch(arr, size, key);  // Check if the element was found  if (index != -1)  {  printf("Element %d found at index %d\n", key, index);  }  else  {  printf("Element %d not found in the array\n", key);  }  return 0;  } |
| Output |
| **14. Selection Sort:** Sort an array using the selection sort algorithm.  Code:  #include <stdio.h>  #include <limits.h>  int main()  {  int arr[8] = {8, 7, 6, 5, 4, 3, 2, 1};  int n = 8;  printf("Before Sorting\n");  for (int i = 0; i < n; i++)  {  printf("%d", arr[i]);  }  // Selection Sort  for (int i = 0; i < n - 1; i++)  {  int min = INT\_MAX;  int minIndex = -1;  for (int j = i; j <= n - 1; j++)  {  if (min > arr[j])  {  min = arr[j];  minIndex = j;  }  }  // Swap 1st element and min  // i and minIndex  int temp = arr[i];  arr[i] = arr[minIndex];  arr[minIndex] = temp;  }  printf("\n\nAfter Sorting\n");  for (int i = 0; i < n; i++)  {  printf("%d", arr[i]);  }  return 0;  } |
| Output |
| **15. Bubble Sort:** Sort an array using the bubble sort algorithm.  Code:  #include <stdio.h>  #include <stdbool.h>  int main()  {  int arr[8] = {8, 7, 6, 5, 4, 3, 2, 1};  int n = 8;  for (int i = 0; i < n; i++)  {  printf("%d", arr[i]);  }  // Bubble Sort  for (int i = 0; i < n - 1; i++) // For no. of passes  {  bool flag = true; // Array is not sorted  for (int j = 0; j < n - 1 - i; j++) // For Swapping  {  if (arr[j] > arr[j + 1])  {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  flag = false;  }  }  if (flag == true)  break;  }  printf("\n");  for (int i = 0; i < n; i++)  {  printf("%d", arr[i]);  }  return 0;  } |
| Output |
| **16. Insertion Sort:** Sort an array using the insertion sort algorithm  Code:  #include <stdio.h>  #include <stdbool.h>  int main()  {  int arr[8] = {8, 7, 6, 5, 4, 3, 2, 1};  int n = 8;  for (int i = 0; i < n; i++)  {  printf("%d", arr[i]);  }  // Insertion Sort  for (int i = 1; i <= n - 1; i++)  {  int j = i;  while (arr[j] < arr[j - 1] && j > 0)  {  int temp = arr[j];  arr[j] = arr[j - 1];  arr[j - 1] = temp;  j--;  }  }  printf("\n");  for (int i = 0; i < n; i++)  {  printf("%d", arr[i]);  }  return 0;  } |
| Output |
| **17. Matrix Operations:** Perform matrix addition, subtraction, multiplication, and transpose.  Code:  #include <stdio.h>  // Function to add two matrices  void addMatrix(int mat1[][3], int mat2[][3], int result[][3], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = mat1[i][j] + mat2[i][j];  }  }  }  // Function to subtract two matrices  void subtractMatrix(int mat1[][3], int mat2[][3], int result[][3], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = mat1[i][j] - mat2[i][j];  }  }  }  // Function to multiply two matrices  void multiplyMatrix(int mat1[][3], int mat2[][3], int result[][3], int rows1, int cols1, int cols2) {  for (int i = 0; i < rows1; i++) {  for (int j = 0; j < cols2; j++) {  result[i][j] = 0;  for (int k = 0; k < cols1; k++) {  result[i][j] += mat1[i][k] \* mat2[k][j];  }  }  }  }  // Function to transpose a matrix  void transposeMatrix(int mat[][3], int result[][3], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[j][i] = mat[i][j];  }  }  }  // Function to display a matrix  void displayMatrix(int mat[][3], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  printf("%d ", mat[i][j]);  }  printf("\n");  }  }  int main() {  int mat1[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  int mat2[3][3] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};  int result[3][3];  printf("Matrix 1:\n");  displayMatrix(mat1, 3, 3);  printf("\nMatrix 2:\n");  displayMatrix(mat2, 3, 3);  // Matrix addition  printf("\nMatrix Addition:\n");  addMatrix(mat1, mat2, result, 3, 3);  displayMatrix(result, 3, 3);  // Matrix subtraction  printf("\nMatrix Subtraction:\n");  subtractMatrix(mat1, mat2, result, 3, 3);  displayMatrix(result, 3, 3);  // Matrix multiplication  printf("\nMatrix Multiplication:\n");  multiplyMatrix(mat1, mat2, result, 3, 3, 3);  displayMatrix(result, 3, 3);  // Transpose of Matrix 1  printf("\nTranspose of Matrix 1:\n");  transposeMatrix(mat1, result, 3, 3);  displayMatrix(result, 3, 3);  return 0;  } |
| Output: |

**Part- C**

|  |
| --- |
| **1. SLL CREATE A NODE** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node  struct Node {  int data;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  // Allocate memory for new node  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed!\n");  exit(1);  }    // Assign data to the new node  newNode->data = data;  newNode->next = NULL;    return newNode;  }  int main() {  int data;  printf("Enter data for the node: ");  scanf("%d", &data);    // Create a new node  struct Node\* newNode = createNode(data);    // Display the data stored in the new node  printf("New node created with data: %d\n", newNode->data);    // Free memory allocated for the node  free(newNode);    return 0;  } |
| Output |

|  |
| --- |
| **2.SLL INSERT FRONT** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node  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 front of the list  void insertFront(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  } else {  newNode->next = \*head;  \*head = newNode;  }  }  // Function to print the linked list  void printList(struct Node\* head) {  struct Node\* temp = head;  while (temp != NULL) {  printf("%d -> ", temp->data);  temp = temp->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  // Insert some nodes at the front  insertFront(&head, 5);  insertFront(&head, 10);  insertFront(&head, 15);  // Print the linked list  printf("Linked List: ");  printList(head);  return 0;  } |
| Output |
| **3.SLL INSERT END** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define the structure of a node  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 end of the linked list  void insertEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }  temp->next = newNode;  }  // Function to print the linked list  void printList(struct Node\* head) {  printf("Linked List: ");  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;  // Inserting nodes at the end  insertEnd(&head, 1);  insertEnd(&head, 2);  insertEnd(&head, 3);  // Printing the linked list  printList(head);  return 0;  } |
| Output |
| **4. SLL DELETE FRONT** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define the structure for a node in the 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 front of the linked list  void insertFront(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  newNode->next = \*head;  \*head = newNode;  }  // Function to delete the front node of the linked list  void deleteFront(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty, cannot delete\n");  return;  }  struct Node\* temp = \*head;  \*head = (\*head)->next;  free(temp);  }  // Function to print the linked list  void printList(struct Node\* head) {  printf("List: ");  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  }  // Main function to demonstrate the linked list operations  int main() {  struct Node\* head = NULL;  // Inserting elements at the front of the linked list  insertFront(&head, 3);  insertFront(&head, 7);  insertFront(&head, 9);  insertFront(&head, 5);  // Printing the original list  printf("Original ");  printList(head);  // Deleting the front node  deleteFront(&head);  // Printing the list after deletion  printf("After deleting front node ");  printList(head);  return 0;  } |
| Output |
| **5.SLL DELETE END** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node  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 delete the last node from the linked list  void deleteEnd(struct Node \*\*head) {  if (\*head == NULL) {  printf("List is empty. Nothing to delete.\n");  return;  }  if ((\*head)->next == NULL) {  // If there's only one node in the list  free(\*head);  \*head = NULL;  return;  }  struct Node \*temp = \*head;  struct Node \*prev = NULL;  while (temp->next != NULL) {  prev = temp;  temp = temp->next;  }  prev->next = NULL;  free(temp);  }  // Function to print the linked list  void printList(struct Node \*head) {  struct Node \*temp = head;  while (temp != NULL) {  printf("%d -> ", temp->data);  temp = temp->next;  }  printf("NULL\n");  }  int main() {  struct Node \*head = NULL;    // Creating nodes and adding them to the list  head = createNode(1);  head->next = createNode(2);  head->next->next = createNode(3);  printf("Original list: ");  printList(head);  // Deleting the last node  deleteEnd(&head);  printf("List after deleting last node: ");  printList(head);  return 0;  } |
| Output |
| **6.SLL INSERT POSITION** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define the structure of a node  struct Node {  int data;  struct Node\* next;  };  // Function to insert a new node at the beginning of the list  void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {  // Allocate memory for new node  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));  // Set data for the new node  new\_node->data = new\_data;  // Set the next of the new node to the current head  new\_node->next = \*head\_ref;  // Move the head to point to the new node  \*head\_ref = new\_node;  }  // Function to insert a new node after a given position  void insertAfterPosition(struct Node\* prev\_node, int new\_data) {  if (prev\_node == NULL) {  printf("Previous node cannot be NULL");  return;  }  // Allocate memory for new node  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));  // Set data for the new node  new\_node->data = new\_data;  // Set the next of the new node to the next of the previous node  new\_node->next = prev\_node->next;  // Set the next of the previous node to the new node  prev\_node->next = new\_node;  }  // Function to print the linked list  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d ", node->data);  node = node->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;  // Insert some nodes at the beginning  insertAtBeginning(&head, 3);  insertAtBeginning(&head, 2);  insertAtBeginning(&head, 1);  printf("Initial list: ");  printList(head);  // Insert a node after the second position (indexing from 0)  insertAfterPosition(head->next, 4);  printf("List after insertion: ");  printList(head);  return 0;  } |
| Output |
| **7.SLL DELETE POSITION** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define the structure of a node  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));  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at the beginning of the linked list  void insertAtBeginning(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  newNode->next = \*head;  \*head = newNode;  }  // Function to delete a node at a specified position  void deleteAtPosition(struct Node\*\* head, int position) {  if (\*head == NULL)  return;  struct Node\* temp = \*head;  // If head needs to be removed  if (position == 0) {  \*head = temp->next;  free(temp);  return;  }  // Find the previous node of the node to be deleted  for (int i = 0; temp != NULL && i < position - 1; i++)  temp = temp->next;  // If position is more than the number of nodes  if (temp == NULL || temp->next == NULL)  return;  // Node temp->next is the node to be deleted  struct Node\* nextNode = temp->next->next;  free(temp->next);  temp->next = nextNode;  }  // Function to print the linked list  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d ", node->data);  node = node->next;  }  printf("\n");  }  // Main function  int main() {  struct Node\* head = NULL;  // Insert some elements into the linked list  insertAtBeginning(&head, 5);  insertAtBeginning(&head, 4);  insertAtBeginning(&head, 3);  insertAtBeginning(&head, 2);  insertAtBeginning(&head, 1);  printf("Linked list before deletion: ");  printList(head);  // Delete node at position 2 (index starts from 0)  int positionToDelete = 2;  deleteAtPosition(&head, positionToDelete);  printf("Linked list after deletion at position %d: ", positionToDelete);  printList(head);  return 0;  } |
| Output |
| **8.SLL DELETE AFTER** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define the structure of a node  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));  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to insert a new node after a given node  void insertAfter(struct Node\* prevNode, int data) {  if (prevNode == NULL) {  printf("The given previous node cannot be NULL\n");  return;  }  struct Node\* newNode = createNode(data);  newNode->next = prevNode->next;  prevNode->next = newNode;  }  // Function to delete the node after a given node  void deleteAfter(struct Node\* prevNode) {  if (prevNode == NULL || prevNode->next == NULL) {  printf("There is no node to delete after the given node\n");  return;  }  struct Node\* temp = prevNode->next;  prevNode->next = temp->next;  free(temp);  }  // Function to print the linked list  void printList(struct Node\* head) {  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  }  int main() {  // Create the head node  struct Node\* head = createNode(1);  // Insert nodes  insertAfter(head, 2);  insertAfter(head->next, 3);  insertAfter(head->next->next, 4);  printf("Original List: ");  printList(head);  // Delete node after the second node  deleteAfter(head);    printf("List after deleting node after the first node: ");  printList(head);  return 0;  } |
| Output |
| **9.SLL DELETE BEFORE** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node in the 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));  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to delete a node before a given value  void deleteNodeBefore(struct Node\*\* head, int value) {  // Check if the list is empty or has only one node  if (\*head == NULL || (\*head)->next == NULL) {  printf("List is empty or contains only one node. No node to delete before.\n");  return;  }  // Find the node before the node with the given value  struct Node\* temp = \*head;  struct Node\* prev = NULL;  while (temp->next != NULL && temp->next->data != value) {  prev = temp;  temp = temp->next;  }  // If the value is not found  if (temp->next == NULL) {  printf("Value not found in the list. Cannot delete node before.\n");  return;  }  // If the value is found and it's the first node  if (prev == NULL) {  struct Node\* toDelete = \*head;  \*head = (\*head)->next;  free(toDelete);  printf("Node before %d deleted successfully.\n", value);  return;  }  // If the value is found and it's not the first node  struct Node\* toDelete = prev->next;  prev->next = toDelete->next;  free(toDelete);  printf("Node before %d deleted successfully.\n", value);  }  // Function to print the linked list  void printList(struct Node\* head) {  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  // Main function  int main() {  // Initialize an empty linked list  struct Node\* head = NULL;  // Create some nodes and link them together  head = createNode(1);  head->next = createNode(2);  head->next->next = createNode(3);  head->next->next->next = createNode(4);  // Print the initial list  printf("Initial list: ");  printList(head);  // Delete a node before a given value  deleteNodeBefore(&head, 3);  // Print the updated list  printf("Updated list: ");  printList(head);  // Free memory  while (head != NULL) {  struct Node\* temp = head;  head = head->next;  free(temp);  }  return 0;  } |
| Output |
| **10.SLL INSERT AFTER** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node  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 after a specific node  void insertAfter(struct Node\* prevNode, int data) {  if (prevNode == NULL) {  printf("Previous node cannot be NULL.\n");  return;  }  struct Node\* newNode = createNode(data);  newNode->next = prevNode->next;  prevNode->next = newNode;  }  // Function to print the linked list  void printList(struct Node\* head) {  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  // Main function  int main() {  // Create the head node  struct Node\* head = createNode(1);  // Insert nodes after the head  insertAfter(head, 2);  insertAfter(head->next, 3);  insertAfter(head->next->next, 4);  // Print the linked list  printf("Linked List: ");  printList(head);  return 0;  } |
| Output |
| **11.SLL INSERT BEFORE** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Structure of a node  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 before a specified node  void insertBefore(struct Node\*\* headRef, struct Node\* nextNode, int newData) {  struct Node\* newNode = createNode(newData);  if (\*headRef == NULL) {  printf("List is empty! Inserting as the first node.\n");  \*headRef = newNode;  } else {  struct Node\* current = \*headRef;  while (current->next != nextNode) {  current = current->next;  if (current == NULL) {  printf("Specified node not found!\n");  return;  }  }  newNode->next = current->next;  current->next = newNode;  }  }  // Function to print the linked list  void printList(struct Node\* head) {  struct Node\* current = head;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;  // Creating the initial list: 1 -> 2 -> 3 -> NULL  head = createNode(1);  head->next = createNode(2);  head->next->next = createNode(3);  printf("Initial list: ");  printList(head);  // Inserting 4 before the node with data 3  insertBefore(&head, head->next->next, 4);  printf("List after insertion: ");  printList(head);  // Freeing allocated memory  struct Node\* temp;  while (head != NULL) {  temp = head;  head = head->next;  free(temp);  }  return 0;  } |
| Output |
| **12. SLL DISPLAY NODE** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define structure for a node  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 display all nodes in the linked list  void displayList(struct Node\* head) {  struct Node\* current = head;  if (current == NULL) {  printf("List is empty.\n");  return;  }  printf("Nodes in the list: ");  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  int main() {  // Creating nodes  struct Node\* head = createNode(1);  struct Node\* second = createNode(2);  struct Node\* third = createNode(3);  // Linking nodes  head->next = second;  second->next = third;  // Displaying nodes  displayList(head);  // Freeing memory  free(head);  free(second);  free(third);  return 0;  } |
| Output |
| **13.SLL TRANSVERSING** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node in the 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 traverse and print the linked list  void traverseLinkedList(struct Node\* head) {  printf("Linked List: ");  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  int main() {  // Creating a sample linked list: 1 -> 2 -> 3 -> 4 -> NULL  struct Node\* head = createNode(1);  head->next = createNode(2);  head->next->next = createNode(3);  head->next->next->next = createNode(4);  // Traversing and printing the linked list  traverseLinkedList(head);  // Freeing allocated memory  while (head != NULL) {  struct Node\* temp = head;  head = head->next;  free(temp);  }  return 0;  } |
| Output |
| **14.SLL SORTING IN ASCENDING AND DESCENDING** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node  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 new node at the end of the linked list  void insertNode(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }  temp->next = newNode;  }  // Function to print the linked list  void printList(struct Node\* head) {  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  // Function to swap the data of two nodes  void swapData(struct Node\* a, struct Node\* b) {  int temp = a->data;  a->data = b->data;  b->data = temp;  }  // Function to sort the linked list in ascending order  void sortAscending(struct Node\* head) {  struct Node\* i, \*j;  for (i = head; i != NULL; i = i->next) {  for (j = i->next; j != NULL; j = j->next) {  if (i->data > j->data) {  swapData(i, j);  }  }  }  }  // Function to sort the linked list in descending order  void sortDescending(struct Node\* head) {  struct Node\* i, \*j;  for (i = head; i != NULL; i = i->next) {  for (j = i->next; j != NULL; j = j->next) {  if (i->data < j->data) {  swapData(i, j);  }  }  }  }  int main() {  struct Node\* head = NULL;  // Inserting nodes into the linked list  insertNode(&head, 5);  insertNode(&head, 3);  insertNode(&head, 8);  insertNode(&head, 1);  insertNode(&head, 6);  printf("Original Linked List: ");  printList(head);  // Sorting the linked list in ascending order  sortAscending(head);  printf("Sorted Linked List (Ascending): ");  printList(head);  // Sorting the linked list in descending order  sortDescending(head);  printf("Sorted Linked List (Descending): ");  printList(head);  return 0;  } |
| Output |
| **15. SLL MERGING NODES** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define the structure for a node  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 merge two linked lists  struct Node\* mergeLists(struct Node\* list1, struct Node\* list2) {  struct Node\* mergedList = NULL;  // Base cases  if (list1 == NULL)  return list2;  else if (list2 == NULL)  return list1;  // Compare the data of the two nodes and merge accordingly  if (list1->data <= list2->data) {  mergedList = list1;  mergedList->next = mergeLists(list1->next, list2);  } else {  mergedList = list2;  mergedList->next = mergeLists(list1, list2->next);  }  return mergedList;  }  // Function to print the linked list  void printList(struct Node\* head) {  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  }  int main() {  // Creating the first linked list: 1 -> 3 -> 5  struct Node\* list1 = createNode(1);  list1->next = createNode(3);  list1->next->next = createNode(5);  // Creating the second linked list: 2 -> 4 -> 6  struct Node\* list2 = createNode(2);  list2->next = createNode(4);  list2->next->next = createNode(6);  printf("List 1: ");  printList(list1);  printf("List 2: ");  printList(list2);  // Merging the two lists  struct Node\* mergedList = mergeLists(list1, list2);  printf("Merged List: ");  printList(mergedList);  // Free memory  while (mergedList != NULL) {  struct Node\* temp = mergedList;  mergedList = mergedList->next;  free(temp);  }  return 0;  } |
| Output |
| **16.DLL INSERT FRONT** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define structure for a node  struct Node {  int data;  struct Node\* next;  struct Node\* prev;  };  // 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;  newNode->prev = NULL;  return newNode;  }  // Function to insert a node at the front of the doubly linked list  void insertFront(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  } else {  newNode->next = \*head;  (\*head)->prev = newNode;  \*head = newNode;  }  }  // Function to print the doubly linked list  void printList(struct Node\* head) {  struct Node\* current = head;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  // Main function  int main() {  struct Node\* head = NULL;  // Inserting nodes at the front  insertFront(&head, 5);  insertFront(&head, 10);  insertFront(&head, 15);  // Printing the doubly linked list  printf("Doubly Linked List: ");  printList(head);  return 0;  } |
| Output |
| **17.DLL INSERT END** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node in the doubly linked list  struct Node {  int data;  struct Node\* prev;  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->prev = NULL;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at the end of the doubly linked list  void insertEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }  temp->next = newNode;  newNode->prev = temp;  }  // Function to display the doubly linked list  void display(struct Node\* head) {  struct Node\* temp = head;  printf("Doubly linked list: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;    // Inserting nodes at the end of the list  insertEnd(&head, 1);  insertEnd(&head, 2);  insertEnd(&head, 3);    // Display the doubly linked list  display(head);    return 0;  } |
| Output |
| **18. DLL DELETE FRONT** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Structure for a doubly linked list node  struct Node {  int data;  struct Node\* prev;  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->prev = NULL;  newNode->next = NULL;  return newNode;  }  // Function to delete the front node of the doubly linked list  void deleteFront(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty\n");  return;  }    struct Node\* temp = \*head;  \*head = (\*head)->next;  if (\*head != NULL)  (\*head)->prev = NULL;  free(temp);  }  // Function to print the doubly linked list  void printList(struct Node\* head) {  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  int main() {  // Creating a doubly linked list  struct Node\* head = createNode(1);  head->next = createNode(2);  head->next->prev = head;  head->next->next = createNode(3);  head->next->next->prev = head->next;  printf("Original list: ");  printList(head);  // Deleting the front node  deleteFront(&head);  printf("List after deleting front node: ");  printList(head);  return 0;  } |
| Output |
| **19.DLL DELETE END** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Structure of a node in doubly linked list  struct Node {  int data;  struct Node\* prev;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->prev = NULL;  newNode->next = NULL;  return newNode;  }  // Function to delete a node from the end  void deleteEnd(struct Node\*\* head) {  if (\*head == NULL)  return;  if ((\*head)->next == NULL) {  free(\*head);  \*head = NULL;  return;  }  struct Node\* secondLast = \*head;  while (secondLast->next->next != NULL) {  secondLast = secondLast->next;  }  free(secondLast->next);  secondLast->next = NULL;  }  // Function to print the doubly linked list  void printList(struct Node\* head) {  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;  // Inserting elements into the doubly linked list  head = createNode(1);  head->next = createNode(2);  head->next->prev = head;  head->next->next = createNode(3);  head->next->next->prev = head->next;  head->next->next->next = createNode(4);  head->next->next->next->prev = head->next->next;  printf("Doubly Linked List before deletion from end: ");  printList(head);  // Deleting node from the end  deleteEnd(&head);  printf("Doubly Linked List after deletion from end: ");  printList(head);  return 0;  } |
| Output |
| **20.CIRCULAR SLL(NODE,IF,IE,DF,DE)** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Node structure  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 node at the front of the list  void insertFront(struct Node \*\*head, int data) {  struct Node \*newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head;  } else {  newNode->next = (\*head)->next;  (\*head)->next = newNode;  int temp = newNode->data;  newNode->data = (\*head)->data;  (\*head)->data = temp;  }  }  // Function to insert node at the end of the list  void insertEnd(struct Node \*\*head, int data) {  struct Node \*newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head;  } else {  newNode->next = (\*head)->next;  (\*head)->next = newNode;  int temp = newNode->data;  newNode->data = (\*head)->data;  (\*head)->data = temp;  \*head = newNode;  }  }  // Function to delete node from the front of the list  void deleteFront(struct Node \*\*head) {  if (\*head == NULL) {  printf("List is empty. Deletion not possible.\n");  } else if ((\*head)->next == \*head) {  free(\*head);  \*head = NULL;  } else {  struct Node \*temp = (\*head)->next;  (\*head)->data = temp->data;  (\*head)->next = temp->next;  free(temp);  }  }  // Function to delete node from the end of the list  void deleteEnd(struct Node \*\*head) {  if (\*head == NULL) {  printf("List is empty. Deletion not possible.\n");  } else if ((\*head)->next == \*head) {  free(\*head);  \*head = NULL;  } else {  struct Node \*current = (\*head)->next;  struct Node \*prev = \*head;  while (current->next != \*head) {  prev = current;  current = current->next;  }  prev->next = current->next;  free(current);  \*head = prev;  }  }  // Function to display the list  void display(struct Node \*head) {  struct Node \*temp = head;  if (head == NULL) {  printf("List is empty.\n");  return;  }  do {  printf("%d ", temp->data);  temp = temp->next;  } while (temp != head);  printf("\n");  }  // Main function  int main() {  struct Node \*head = NULL;  // Inserting elements at the front  insertFront(&head, 3);  insertFront(&head, 2);  insertFront(&head, 1);  printf("List after inserting elements at the front: ");  display(head);  // Deleting element from the front  deleteFront(&head);  printf("List after deleting an element from the front: ");  display(head);  // Inserting elements at the end  insertEnd(&head, 4);  insertEnd(&head, 5);  printf("List after inserting elements at the end: ");  display(head);  // Deleting element from the end  deleteEnd(&head);  printf("List after deleting an element from the end: ");  display(head);  return 0;  } |
| Output |
| **21.CIRCULAR DLL(NODE,IF,IE,DF,DE)** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Structure of a node  struct Node {  int data;  struct Node \*prev;  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->prev = NULL;  newNode->next = NULL;  return newNode;  }  // Function to insert a node at the front of the list  void insertFront(struct Node\*\* head\_ref, int data) {  struct Node\* newNode = createNode(data);  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  newNode->next = newNode;  newNode->prev = newNode;  } else {  newNode->next = \*head\_ref;  newNode->prev = (\*head\_ref)->prev;  (\*head\_ref)->prev->next = newNode;  (\*head\_ref)->prev = newNode;  \*head\_ref = newNode;  }  }  // Function to insert a node at the end of the list  void insertEnd(struct Node\*\* head\_ref, int data) {  if (\*head\_ref == NULL) {  insertFront(head\_ref, data);  } else {  struct Node\* newNode = createNode(data);  newNode->next = (\*head\_ref);  newNode->prev = (\*head\_ref)->prev;  (\*head\_ref)->prev->next = newNode;  (\*head\_ref)->prev = newNode;  }  }  // Function to delete a node from the front of the list  void deleteFront(struct Node\*\* head\_ref) {  if (\*head\_ref == NULL) {  printf("List is empty, deletion not possible.\n");  } else {  struct Node\* temp = \*head\_ref;  if ((\*head\_ref)->next == (\*head\_ref)) {  \*head\_ref = NULL;  } else {  (\*head\_ref)->prev->next = (\*head\_ref)->next;  (\*head\_ref)->next->prev = (\*head\_ref)->prev;  \*head\_ref = (\*head\_ref)->next;  }  free(temp);  }  }  // Function to delete a node from the end of the list  void deleteEnd(struct Node\*\* head\_ref) {  if (\*head\_ref == NULL) {  printf("List is empty, deletion not possible.\n");  } else {  struct Node\* temp = (\*head\_ref)->prev;  if ((\*head\_ref)->next == (\*head\_ref)) {  \*head\_ref = NULL;  } else {  (\*head\_ref)->prev = temp->prev;  temp->prev->next = \*head\_ref;  }  free(temp);  }  }  // Function to display the circular doubly linked list  void display(struct Node\* head) {  if (head == NULL) {  printf("List is empty.\n");  return;  }  struct Node\* current = head;  do {  printf("%d ", current->data);  current = current->next;  } while (current != head);  printf("\n");  }  int main() {  struct Node\* head = NULL;  insertEnd(&head, 10);  insertEnd(&head, 20);  insertEnd(&head, 30);  insertFront(&head, 5);  printf("Circular Doubly Linked List: ");  display(head);  deleteFront(&head);  deleteEnd(&head);  printf("After deletion from front and end: ");  display(head);  return 0;  } |
| Output |
| **22.**STACK TO PUSH,POP,ISFULL,ISEMPTY,PEEK FUNCTIONS |
| Code:  #include <stdio.h>  #include <stdbool.h>  #define MAX\_SIZE 100  // Define the stack structure  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Initialize stack  void initialize(struct Stack \*s) {  s->top = -1;  }  // Check if the stack is full  bool isFull(struct Stack \*s) {  return (s->top == MAX\_SIZE - 1);  }  // Check if the stack is empty  bool isEmpty(struct Stack \*s) {  return (s->top == -1);  }  // Push an element onto the stack  void push(struct Stack \*s, int value) {  if (isFull(s)) {  printf("Stack Overflow\n");  return;  }  s->items[++s->top] = value;  printf("%d pushed to stack\n", value);  }  // Pop an element from the stack  int pop(struct Stack \*s) {  if (isEmpty(s)) {  printf("Stack Underflow\n");  return -1;  }  return s->items[s->top--];  }  // Peek the top element of the stack without removing it  int peek(struct Stack \*s) {  if (isEmpty(s)) {  printf("Stack is empty\n");  return -1;  }  return s->items[s->top];  }  int main() {  struct Stack stack;  initialize(&stack);  // Example usage  push(&stack, 10);  push(&stack, 20);  push(&stack, 30);  printf("Top element is %d\n", peek(&stack));  printf("%d popped from stack\n", pop(&stack));  printf("%d popped from stack\n", pop(&stack));  printf("%d popped from stack\n", pop(&stack));  printf("Is stack empty? %s\n", isEmpty(&stack) ? "Yes" : "No");  return 0;  } |
| Output |
| **23.INFIX TO POSTFIX(STACK)** |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAX\_SIZE 100  typedef struct {  char items[MAX\_SIZE];  int top;  } Stack;  void initialize(Stack \*s) {  s->top = -1;  }  int isFull(Stack \*s) {  return s->top == MAX\_SIZE - 1;  }  int isEmpty(Stack \*s) {  return s->top == -1;  }  void push(Stack \*s, char c) {  if (!isFull(s)) {  s->top++;  s->items[s->top] = c;  } else {  printf("Stack Overflow\n");  exit(EXIT\_FAILURE);  }  }  char pop(Stack \*s) {  if (!isEmpty(s)) {  char popped = s->items[s->top];  s->top--;  return popped;  } else {  printf("Stack Underflow\n");  exit(EXIT\_FAILURE);  }  }  int precedence(char c) {  if (c == '+' || c == '-')  return 1;  else if (c == '\*' || c == '/')  return 2;  else  return 0;  }  void infixToPostfix(char \*infix, char \*postfix) {  Stack s;  initialize(&s);  int i = 0, j = 0;  while (infix[i] != '\0') {  if (isdigit(infix[i]) || isalpha(infix[i])) {  postfix[j] = infix[i];  j++;  } else if (infix[i] == '(') {  push(&s, '(');  } else if (infix[i] == ')') {  while (!isEmpty(&s) && s.items[s.top] != '(') {  postfix[j] = pop(&s);  j++;  }  if (!isEmpty(&s) && s.items[s.top] == '(')  pop(&s);  else {  printf("Invalid expression\n");  exit(EXIT\_FAILURE);  }  } else {  while (!isEmpty(&s) && precedence(infix[i]) <= precedence(s.items[s.top])) {  postfix[j] = pop(&s);  j++;  }  push(&s, infix[i]);  }  i++;  }  while (!isEmpty(&s)) {  postfix[j] = pop(&s);  j++;  }  postfix[j] = '\0';  }  int main() {  char infix[MAX\_SIZE];  char postfix[MAX\_SIZE];  printf("Enter infix expression: ");  fgets(infix, MAX\_SIZE, stdin);  // Removing newline character from fgets input  infix[strcspn(infix, "\n")] = '\0';  infixToPostfix(infix, postfix);  printf("Postfix expression: %s\n", postfix);  return 0;  } |
| Output |
| **24.POSTFIX EVALUATION(STACK)** |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  #define MAX\_SIZE 100  // Structure to represent stack  struct Stack {  int top;  int data[MAX\_SIZE];  };  // Function to create a new stack  struct Stack\* createStack() {  struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));  stack->top = -1;  return stack;  }  // Function to check if stack is empty  int isEmpty(struct Stack\* stack) {  return (stack->top == -1);  }  // Function to push element onto stack  void push(struct Stack\* stack, int value) {  if (stack->top == MAX\_SIZE - 1) {  printf("Stack Overflow\n");  return;  }  stack->data[++stack->top] = value;  }  // Function to pop element from stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow\n");  return -1;  }  return stack->data[stack->top--];  }  // Function to evaluate postfix expression  int evaluatePostfix(char\* exp) {  struct Stack\* stack = createStack();  int i, operand1, operand2, result;  for (i = 0; exp[i] != '\0'; i++) {  if (isdigit(exp[i])) {  push(stack, exp[i] - '0');  } else {  operand2 = pop(stack);  operand1 = pop(stack);  switch (exp[i]) {  case '+':  result = operand1 + operand2;  break;  case '-':  result = operand1 - operand2;  break;  case '\*':  result = operand1 \* operand2;  break;  case '/':  result = operand1 / operand2;  break;  default:  printf("Invalid operator\n");  return -1;  }  push(stack, result);  }  }  return pop(stack);  }  int main() {  char exp[MAX\_SIZE];  printf("Enter a postfix expression: ");  fgets(exp, MAX\_SIZE, stdin);  int result = evaluatePostfix(exp);  printf("Result: %d\n", result);  return 0;  } |
| Output |
| **25.BALANCED PARANTHESIS(STACK)** |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100  // Define a structure for stack  struct Stack {  int top;  char items[MAX\_SIZE];  };  // Function to initialize the stack  void initialize(struct Stack \*s) {  s->top = -1;  }  // Function to push an element onto the stack  void push(struct Stack \*s, char c) {  if (s->top == MAX\_SIZE - 1) {  printf("Stack Overflow\n");  exit(EXIT\_FAILURE);  } else {  s->top++;  s->items[s->top] = c;  }  }  // Function to pop an element from the stack  char pop(struct Stack \*s) {  if (s->top == -1) {  printf("Stack Underflow\n");  exit(EXIT\_FAILURE);  } else {  return s->items[s->top--];  }  }  // Function to check if a character is an opening parenthesis  bool isOpening(char c) {  return (c == '(' || c == '[' || c == '{');  }  // Function to check if a character is a closing parenthesis  bool isClosing(char c) {  return (c == ')' || c == ']' || c == '}');  }  // Function to match opening and closing parentheses  bool isMatchingPair(char opening, char closing) {  if (opening == '(' && closing == ')')  return true;  else if (opening == '[' && closing == ']')  return true;  else if (opening == '{' && closing == '}')  return true;  else  return false;  }  // Function to check if the parentheses in a string are balanced  bool areParenthesesBalanced(char exp[]) {  struct Stack s;  initialize(&s);  for (int i = 0; exp[i] != '\0'; i++) {  if (isOpening(exp[i])) {  push(&s, exp[i]);  } else if (isClosing(exp[i])) {  if (s.top == -1 || !isMatchingPair(pop(&s), exp[i])) {  return false;  }  }  }  return (s.top == -1); // If stack is empty, parentheses are balanced  }  // Main function  int main() {  char exp[MAX\_SIZE];  printf("Enter expression: ");  fgets(exp, sizeof(exp), stdin);  if (areParenthesesBalanced(exp)) {  printf("Parentheses are balanced\n");  } else {  printf("Parentheses are not balanced\n");  }  return 0;  } |
| Output |
| **26.STACK USING LINKED LIST** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a stack node  struct Node {  int data;  struct Node\* next;  };  // Function to create a new node  struct Node\* newNode(int data) {  struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));  if (!node) {  printf("Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  node->data = data;  node->next = NULL;  return node;  }  // Function to push a new element onto the stack  void push(struct Node\*\* top, int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (!newNode) {  printf("Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  newNode->data = data;  newNode->next = \*top;  \*top = newNode;  printf("%d pushed to stack\n", data);  }  // Function to pop an element from the stack  int pop(struct Node\*\* top) {  if (\*top == NULL) {  printf("Stack is empty\n");  exit(EXIT\_FAILURE);  }  struct Node\* temp = \*top;  int popped = temp->data;  \*top = temp->next;  free(temp);  return popped;  }  // Function to check if the stack is empty  int isEmpty(struct Node\* top) {  return (top == NULL);  }  // Function to print the stack  void printStack(struct Node\* top) {  struct Node\* current = top;  printf("Stack: ");  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  int main() {  struct Node\* top = NULL;  // Push some elements onto the stack  push(&top, 10);  push(&top, 20);  push(&top, 30);  // Print the stack  printStack(top);  // Pop an element from the stack  printf("Popped element: %d\n", pop(&top));  // Print the stack after popping  printStack(top);  return 0;  } |
| Output |
| **27.LINEAR QUEUE** |
| Code:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 10  // Structure to represent a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  };  // Function to create a new queue  struct Queue\* createQueue() {  struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));  queue->front = -1;  queue->rear = -1;  return queue;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return queue->rear == MAX\_SIZE - 1;  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full, cannot enqueue.\n");  } else {  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear++;  queue->items[queue->rear] = value;  printf("Enqueued: %d\n", value);  }  }  // Function to remove an element from the front of the queue  int dequeue(struct Queue\* queue) {  int item;  if (isEmpty(queue)) {  printf("Queue is empty, cannot dequeue.\n");  return -1;  } else {  item = queue->items[queue->front];  if (queue->front >= queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front++;  }  printf("Dequeued: %d\n", item);  return item;  }  }  // Function to display the elements of the queue  void display(struct Queue\* queue) {  if (isEmpty(queue)) {  printf("Queue is empty.\n");  } else {  printf("Queue elements: ");  for (int i = queue->front; i <= queue->rear; i++) {  printf("%d ", queue->items[i]);  }  printf("\n");  }  }  int main() {  struct Queue\* queue = createQueue();  enqueue(queue, 10);  enqueue(queue, 20);  enqueue(queue, 30);  display(queue);  dequeue(queue);  display(queue);  return 0;  } |
| Output |
| **28.QUEUE USING LINKED LIST** |
| Code:  #include <stdio.h>  #include <stdlib.h>  // Define a structure for a node in the linked list  struct Node {  int data;  struct Node\* next;  };  // Define a structure for the queue  struct Queue {  struct Node \*front, \*rear;  };  // Function to create a new node  struct Node\* newNode(int data) {  struct Node\* temp = (struct Node\*)malloc(sizeof(struct Node));  temp->data = data;  temp->next = NULL;  return temp;  }  // Function to create a new queue  struct Queue\* createQueue() {  struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));  queue->front = queue->rear = NULL;  return queue;  }  // Function to enqueue an element into the queue  void enqueue(struct Queue\* queue, int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = NULL;  if (queue->rear == NULL) {  queue->front = queue->rear = newNode;  return;  }  queue->rear->next = newNode;  queue->rear = newNode;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue\* queue) {  if (queue->front == NULL) {  printf("Queue is empty\n");  return -1;  }  struct Node\* temp = queue->front;  int data = temp->data;  queue->front = queue->front->next;  if (queue->front == NULL)  queue->rear = NULL;  free(temp);  return data;  }  // Function to display the elements of the queue  void display(struct Queue\* queue) {  if (queue->front == NULL) {  printf("Queue is empty\n");  return;  }  struct Node\* temp = queue->front;  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n");  }  int main() {  struct Queue\* queue = createQueue();  // Enqueue some elements  enqueue(queue, 10);  enqueue(queue, 20);  enqueue(queue, 30);  enqueue(queue, 40);  enqueue(queue, 50);  // Display the queue  printf("Queue: ");  display(queue);  // Dequeue some elements  printf("Dequeued element: %d\n", dequeue(queue));  printf("Dequeued element: %d\n", dequeue(queue));  // Display the queue after dequeue  printf("Queue after dequeue: ");  display(queue);  return 0;  } |
| Output |
| **29.CIRCULAR QUEUE** |
| Code:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 5  struct Queue {  int items[MAX\_SIZE];  int front, rear;  };  // Function to create a circular queue  struct Queue\* createQueue() {  struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));  q->front = -1;  q->rear = -1;  return q;  }  // Function to check if the queue is full  int isFull(struct Queue\* q) {  if ((q->front == 0 && q->rear == MAX\_SIZE - 1) ||  (q->rear == (q->front - 1) % (MAX\_SIZE - 1))) {  return 1;  }  return 0;  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* q) {  if (q->front == -1) {  return 1;  }  return 0;  }  // Function to add an element to the queue  void enqueue(struct Queue\* q, int value) {  if (isFull(q)) {  printf("Queue is full\n");  return;  }  if (q->front == -1) {  q->front = 0;  }  q->rear = (q->rear + 1) % MAX\_SIZE;  q->items[q->rear] = value;  printf("%d enqueued to queue\n", value);  }  // Function to remove an element from the queue  int dequeue(struct Queue\* q) {  int item;  if (isEmpty(q)) {  printf("Queue is empty\n");  return -1;  }  item = q->items[q->front];  if (q->front == q->rear) {  q->front = -1;  q->rear = -1;  } else {  q->front = (q->front + 1) % MAX\_SIZE;  }  return item;  }  // Function to display the queue  void display(struct Queue\* q) {  int i;  if (isEmpty(q)) {  printf("Queue is empty\n");  return;  }  printf("Front -> ");  for (i = q->front; i != q->rear; i = (i + 1) % MAX\_SIZE) {  printf("%d -> ", q->items[i]);  }  printf("%d -> Rear\n", q->items[i]);  }  int main() {  struct Queue\* q = createQueue();  enqueue(q, 1);  enqueue(q, 2);  enqueue(q, 3);  enqueue(q, 4);  display(q);  printf("Dequeued: %d\n", dequeue(q));  printf("Dequeued: %d\n", dequeue(q));  display(q);  enqueue(q, 5);  enqueue(q, 6);  display(q);  return 0;  } |
| Output |
| **30.DOUBLE ENDED QUEUE** |
| Code:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 10  // Structure to represent a deque  typedef struct {  int data[MAX\_SIZE];  int front, rear;  } Deque;  // Function to initialize a deque  void initDeque(Deque \*deque) {  deque->front = -1;  deque->rear = -1;  }  // Function to check if deque is empty  int isEmpty(Deque \*deque) {  return (deque->front == -1 && deque->rear == -1);  }  // Function to check if deque is full  int isFull(Deque \*deque) {  return ((deque->rear + 1) % MAX\_SIZE == deque->front);  }  // Function to add an element to the front of the deque  void addToFront(Deque \*deque, int element) {  if (isFull(deque)) {  printf("Deque is full. Cannot add more elements.\n");  return;  } else if (isEmpty(deque)) {  deque->front = deque->rear = 0;  } else {  deque->front = (deque->front - 1 + MAX\_SIZE) % MAX\_SIZE;  }  deque->data[deque->front] = element;  }  // Function to add an element to the rear of the deque  void addToRear(Deque \*deque, int element) {  if (isFull(deque)) {  printf("Deque is full. Cannot add more elements.\n");  return;  } else if (isEmpty(deque)) {  deque->front = deque->rear = 0;  } else {  deque->rear = (deque->rear + 1) % MAX\_SIZE;  }  deque->data[deque->rear] = element;  }  // Function to remove an element from the front of the deque  int removeFromFront(Deque \*deque) {  int removedElement;  if (isEmpty(deque)) {  printf("Deque is empty. Cannot remove element.\n");  return -1;  } else if (deque->front == deque->rear) {  removedElement = deque->data[deque->front];  deque->front = deque->rear = -1;  } else {  removedElement = deque->data[deque->front];  deque->front = (deque->front + 1) % MAX\_SIZE;  }  return removedElement;  }  // Function to remove an element from the rear of the deque  int removeFromRear(Deque \*deque) {  int removedElement;  if (isEmpty(deque)) {  printf("Deque is empty. Cannot remove element.\n");  return -1;  } else if (deque->front == deque->rear) {  removedElement = deque->data[deque->rear];  deque->front = deque->rear = -1;  } else {  removedElement = deque->data[deque->rear];  deque->rear = (deque->rear - 1 + MAX\_SIZE) % MAX\_SIZE;  }  return removedElement;  }  // Function to display the elements of the deque  void displayDeque(Deque \*deque) {  if (isEmpty(deque)) {  printf("Deque is empty.\n");  return;  }  printf("Deque elements: ");  int i = deque->front;  while (i != deque->rear) {  printf("%d ", deque->data[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", deque->data[i]);  }  int main() {  Deque myDeque;  initDeque(&myDeque);  addToFront(&myDeque, 5);  addToRear(&myDeque, 10);  addToFront(&myDeque, 3);  displayDeque(&myDeque);  removeFromFront(&myDeque);  removeFromRear(&myDeque);  displayDeque(&myDeque);  return 0;  } |
| Output |