|  |  |
| --- | --- |
| **Name of the student** | **NISHA MALLIKARJUN HAMPIHOLI** |
| **Roll No** | **130** |
| **USN** | **01FE22BEI036** |
| **Div** | **F** |

**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**.. In C programming, variables and data types are fundamental concepts but they serve different purposes.  1.Data type:  Data types in C programming specify the type of data that a variable can hold.   * + C provides several basic data types such as int, float, char, double, etc., each with its own range and format specifier.   + Data types determine the size and layout of the variable's memory.   + Examples:   + int age = 25; // integer data type   + float height = 5.8; // floating-point data type   + char grade = 'A'; // character data type   2.Variable:   * + A variable in C programming is a name given to a memory location, which can hold a value of a specific data type.   + It acts as a container for storing data during the execution of a program.   + Variables must be declared with a specific data type before they can be used.   + Examples:   + int age; // declaring an integer variable   + float height; // declaring a floating-point variable   + char grade; // declaring a character variable   Here's a comparison:   * Data Type: Specifies the type of data a variable can hold. * Variable: Represents a memory location used to store data of a specific type.   In essence, data types define the kind of data a variable can store, while variables provide a means to access and manipulate that data within a program.  Top of Form |
| **2**.What is the difference between a variable and a data type in C programming? Provide examples to illustrate. |
| **A**. In C programming, a data type is a classification that specifies which type of value a variable can hold. It defines the set of values that a variable can take and the operations that can be performed on those values. Data types in C are used to allocate memory space and interpret the stored value correctly.  Here are the different types of data types available in C:   1. Basic Data Types:    * int: Used to store integer values. It typically allocates 4 bytes of memory space. Example: int age = 25**;**    * float: Used to store floating-point (decimal) values. It typically allocates 4 bytes of memory space. Example: float height= 5.8**;**    * double: Used to store double-precision floating-point values. It typically allocates 8 bytes of memory space. Example: double weight = 68.5**;**    * char: Used to store single characters. It typically allocates 1 byte of memory space. Example: char grade = 'A'**;**    * \_Bool: Used to store Boolean values, which can be either true or false. It typically allocates 1 byte of memory space. Example: **\_**Bool isPassed= 1**;** (true) 2. Derived Data Types:    * Arrays: A collection of elements of the same data type. Arrays allow storing multiple values under a single variable name. Example: intnumbers[5] = {1, 2, 3, 4, 5};    * Pointers: Variables that store memory addresses of other variables. They are used for dynamic memory allocation and manipulation. Example: int \*ptr;    * Structures: Custom-defined data types that can hold multiple variables of different data types under a single name. Example:   Example**:**  struct Person  {  char name[20];  int age;  float height;  };  struct Person person1;   * + Unions: Similar to structures, but in a union, all members share the same memory location, and only one member can hold a value at a time. Example:   union Data  {  int i;  float f;  char c;  };  union Data data;   1. Enumeration Data Type:    * enum: A user-defined data type used to assign names to integral constants, making the code more readable and maintainable. Example:;    * These data types provide the building blocks for creating variables to store different kinds of data in C programming, allowing for flexibility and efficient memory usage in program development.   Top of Form |
| **3**.How are variables declared and initialized in C programming? Provide examples of variable declarations with different data types. |
| **A**. In C programming, variables are declared and initialized using specific syntax. Declaration involves specifying the data type and the name of the variable, while initialization assigns an initial value to the variable at the time of declaration.  Here's the general syntax for declaring and initializing variables in C: data\_type variable\_name = initial\_value;  Here are some examples of variable declarations with different data types:   1. Integer Variable Declaration and Initialization:   int age = 25;  In this example, an integer variable named **age** is declared and initialized with the value **25**.   1. Floating-Point Variable Declaration and Initialization:   float height = 5.8;  Here, a floating-point variable named **height** is declared and initialized with the value **5.8**.   1. Character Variable Declaration and Initialization:   char grade = 'A';  This declares a character variable named **grade** and initializes it with the character **'A'**.   1. Boolean Variable Declaration and Initialization:   \_Bool is Passed = 1;  In this case, a Boolean variable named **isPassed** is declared and initialized with the value **1**, which represents **true** in C.   1. Array Declaration and Initialization:   int numbers[5] = {1, 2, 3, 4, 5};  This declares an array of integers named **numbers** with a size of 5 elements and initializes it with the values **{1, 2, 3, 4, 5}**.   1. Pointer Variable Declaration:   int \*ptr;  This declares a pointer variable named **ptr** which can store the memory address of an integer variable.   1. Structure Variable Declaration and Initialization:   struct Person {  char name[20];  int age;  float height;  };  struct Person person1 = {"John", 30, 6.0}; Here, a structure type named **Person** is defined, and a structure variable **person1** of type **Person** is declared and initialized with values for its members.  These examples illustrate how variables are declared and initialized in C programming for various data types, including basic types, arrays, pointers, and structures.  Top of Form |
| **4**.Discuss the scope and lifetime of variables in C programming. What are global and local variables? |
| **A**.1) Scope determines where a variable can be accessed: global variables are accessible throughout the program, while local variables are limited to the block or function they're declared in.  2)Lifetime defines how long a variable exists: global variables have a static lifetime, persisting throughout the program, whereas local variables have an automatic lifetime, existing only within their enclosing block or function.  3)Global variables are declared outside any function and retain their values between function calls.  4)Local variables are declared within a block or function and do not retain their values between function calls.  Code:  #include <stdio.h>  // Global variable declaration  int globalVar = 10;  int main() {  // Printing global variable  printf("Global variable value: %d\n", globalVar);  return 0;  } |
| **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 is the process of converting a value from one data type to another. It's necessary when performing operations involving different data types or when assigning values to variables of different types. Type casting is performed by placing the desired data type in parentheses before the value or expression to be converted. It allows for compatibility between different data types in expressions or assignments. |
| **6**.Describe the purpose and usage of the ternary conditional operator (?:) in C programming. Provide an example demonstrating its usage?  **A**. The ternary conditional operator **?:** in C provides a compact way to write conditional expressions. It evaluates a condition and returns one of two values based on whether the condition is true or false. It's often used for simple conditional assignments.  Example: #include <stdio.h>  int main() {  int num = 10;  char\* result = (num % 2 == 0) ? "even" : "odd";  printf("Number %d is %s.\n", num, result);  return 0;  }  **7**.Discuss the bitwise operators available in C programming. Explain their usage with suitable examples.  **A**. Examples of bitwise operators along with a brief explanation:   1. **Bitwise AND (&):**   unsigned int a = 10; // 1010 in binary  unsigned int b = 6; // 0110 in binary  unsigned int result = a & b; // result is 0010 in binary (2 in decimal)= a & b; // result is 0010 in binary (2 in decimal)  Explanation: Bitwise AND sets a bit to 1 only if it is set in both operands.   1. **Bitwise OR (|):**   unsigned int a = 10; // 1010 in binary  unsigned int b = 6; // 0110 in binary  unsigned int result = a | b; // result is 1110 in binary (14 in decimal)= a | b; // result is 1110 in binary (14 in decimal)  Explanation: Bitwise OR sets a bit to 1 if it is set in either operand.   1. **Bitwise XOR (^)**:   unsigned int a = 10; // 1010 in binary  unsigned int b = 6; // 0110 in binary  unsigned int result = a ^ b; // result is 1100 in binary (12 in decimal)= a ^ b; // result is 1100 in binary (12 in decimal)  Explanation: Bitwise XOR sets a bit to 1 if it is set in one operand but not the other.   1. **Bitwise NOT (~):**   unsigned int a = 10; // 1010 in binary  unsigned int result = ~a; // result is 0101...101 in binary (depends on the size of int), which is -11 in decimal (depends on the size of int), which is -11 in decimal  Explanation: Bitwise NOT flips all bits in the operand.   1. **Left Shift (<<):**   unsigned int a = 5; // 0101 in binary  unsigned int result = a << 2; // result is 010100 in binary (20 in decimal)  Explanation: Left shift moves all bits to the left by a specified number of positions, appending zeros to the right.   1. **Right Shift (>>):**   unsigned int a = 20; // 10100 in binary  unsigned int result = a >> 2; // result is 0010 in binary (5 in decimal)is 0010 in binary (5 in decimal)  Explanation: Right shift moves all bits to the right by a specified number of positions, appending zeros to the left.  These operations are fundamental for manipulating individual bits within binary representations of numbers.  **8**.Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate?  **A**.Prefix Increment (++a):   * Example:   int a = 5;  int b = ++a;++a;   * Explanation: In this example, **a** is first incremented by 1, and then the updated value (6) is assigned to **b**. So, **a** becomes 6, and **b** also becomes 6.   Postfix Increment (a++):   * Example:   int a = 5;  int b = a++;++;   * Explanation: Here, the current value of **a** (5) is first assigned to **b**, and then **a** is incremented by 1. So, **a** becomes 6, and **b** becomes 5.   In short, with prefix increment, the value is first incremented, and then used. With postfix increment, the value is first used, and then incremented.  9) What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions?   * A.Logical AND (&&): Returns true if both operands are true. * Logical OR (||): Returns true if at least one operand is true.   Usage:   * **&&** is used when both conditions must be true. * **||** is used when at least one condition must be true.   **10)** Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation  A.Operator Precedence:   * Determines the order in which operators are evaluated. * Operators with higher precedence are evaluated first. * Example:   int result = 2 + 3 \* 4; // Multiplication (\*) has higher precedence than addition (+), so it's evaluated first  // Result: 14// Multiplication (\*) has higher precedence than addition (+), so it's evaluated first // Result: 14  Associativity:   * Defines the order of evaluation for operators of the same precedence. * Can be left-to-right or right-to-left. * Example:   int result2 = 10 / 2 \* 2; // Division and multiplication have the same precedence, evaluated left-to-right  // Result: 10and multiplication have the same precedence, evaluated left-to-right // Result: 10  **11)** Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement?  A.Switch Statement:   * Purpose: The switch statement is used for multi-way branching, allowing you to select among a set of alternatives based on the value of an expression. * Usage: You provide a value or expression to evaluate, and the switch statement executes code blocks associated with specific cases. * Example:   switch (expression) {  case value1:  // code block 1  break;  case value2:  // code block 2  break;  default:  // code block for default case  }   * Differences from if-else:   + switch is typically used when there are multiple conditions to evaluate against a single expression.   + if-else statements are more flexible and can handle a wide range of conditions and expressions.   **12**) Explain the concept of nested control structures in C programming. Provide an example demonstrating nested if-else statements?  A.Nested Control Structures in C:   * Purpose: Nested control structures allow for more complex decision-making and looping by placing one control structure within another. * Example - Nested if-else Statements:   #include <stdio.h>  int main() {  int x = 10;  int y = 5;  if (x > y) {  if (x % 2 == 0) {  printf("x is greater than y and even.\n");  } else {  printf("x is greater than y and odd.\n");  }  } else {  printf("y is greater than or equal to x.\n");  }  return 0;  }  Explanation: In this example, the outer if-else statement checks if **x** is greater than **y**. If true, it enters another if-else statement to check if **x** is even or odd. If false, it prints a different message.  13) Discuss the role of the break and continue statements in loop control in C programming. Provide examples to illustrate their usage?  A.break Statement:   * Usage: Terminates the loop it's contained in. * Example:   for (int i = 0; i < 10; i++) {  printf("%d\n", i);  if (i == 5) {  break; // Exit the loop when i equals 5  }  }5 } }   * Explanation: The loop terminates when **i** equals 5, and no further iterations occur.   continue Statement:   * Usage: Skips the remaining code in the current iteration and proceeds to the next iteration. * Example:   for (int i = 0; i < 10; i++) {  if (i % 2 == 0) {  continue; // Skip even numbers  }  printf("%d\n", i);  Explanation: Even numbers are skipped using the **continue** statement, and only odd numbers are printed  14) What are the advantages of using the for loop over the while loop in C programming? Provide examples comparing the two  A.Advantages of for loop over while loop**:**   1. Concise initialization, condition, and update:    * **for** loop allows you to specify all loop control operations in a single line.    * Example:   for (int i = 0; i < 5; i++) {  printf("%d\n", i);   1. Suited for iterating over a range:    * **for** loop is convenient for iterating over a known range of values.    * Example:   for (int i = 1; i <= 10; i++) {  printf("%d\n", i);}   1. Initialization and update within loop structure:    * Initialization and update operations are encapsulated within the loop structure.    * Helps in reducing errors related to missed updates.    * Example:   for (int i = 0; i < 5; ) {  printf("%d\n", i);  i++;  }}  In contrast, **while** loops are typically used when the number of iterations is not known beforehand or when the loop control logic is more complex. However, **for** loops are preferred when iterating over a range or when a simple counter is used for iteration.  **15)** Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements?  **A.**  Short-circuit evaluation in C means that the evaluation of a logical expression stops as soon as the final result can be determined based on the evaluation of a subset of the expression. This affects the evaluation of logical expressions in if statements by potentially skipping the evaluation of unnecessary parts of the expression, leading to improved efficiency and performance.  16) Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes?  A.Purpose and Structure of Function Prototype in C**:**   * Purpose: Declare a function before its implementation. * Structure: return\_type function\_name(parameter\_list); * Necessity:   1. Allows order independence of function calls.   2. Enables type checking and error detection at compile time.   3. Promotes code modularity and readability.   Top of Form  17) Explain the difference between call by value and call by reference in C programming. Provide examples to illustrate both concepts  A.Call by Value:   * Passes a copy of the parameter's value. * Changes made to the parameter inside the function do not affect the original value. * Example:   #include <stdio.h>  void increment(int x) {  x++; // Changes are local to the function  }  int main() {  int num = 5;  increment(num);  printf("Original value: %d\n", num); // Output: 5 (unchanged)  return 0;  }Call by Reference:   * Passes the address of the parameter. * Changes made to the parameter inside the function affect the original value. * Example:   #include <stdio.h>  void increment(int \*x) {  (\*x)++; // Changes are reflected in the original value  }  int main() {  int num = 5;  increment(&num);  printf("Modified value: %d\n", num); // Output: 6 (modified)  return 0;  }  18) Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works  A**. Recursion in C Programming:**   * Concept: A function calls itself to solve a problem by breaking it down into smaller instances. * Example: Calculating factorial of a number.   #include <stdio.h>  int factorial(int n) {  if (n == 0 || n == 1) {  return 1;  } else {  return n \* factorial(n - 1);  }  }  int main() {  int num = 5;  int result = factorial(num);  printf("Factorial of %d is %d\n", num, result);  return 0;   * }Explanation:   + The **factorial** function calculates the factorial of a number recursively.   + Base case: If **n** is 0 or 1, return 1.   + Recursive case: Call **factorial** function with **n - 1** and multiply the result by **n**.   + This process repeats until the base case is reached, and then the result is returned.   Recursion is a powerful technique for solving problems by breaking them down into smaller subproblems. It provides an elegant and concise way to express solutions to certain types of problems.  19) What is the significance of the return statement in C programming? How are values returned from functions  Significance of the return statement in C**:**   1. Termination of function execution: Used to exit a function and return control to the calling function. 2. Returning values from functions: Used to return a value from a function to its caller.   How values are returned from functions:   * The value following the **return** keyword is evaluated and returned to the caller. * Functions that have a return type other than **void** must use a **return** statement to return a value of the specified type.   **20)** Describe the role of function parameters and arguments in C programming. How are function arguments passed to parameters?  **A.** Role of Function Parameters and Arguments:   * Function Parameters: Variables declared in the function prototype and definition that receive values when the function is called. * Function Arguments: Values passed to the function when it is called, corresponding to the parameters.   Passing Function Arguments to Parameters:   * By Value: Copies of the arguments are passed to the parameters. Changes to parameters do not affect the original arguments. * By Reference: Addresses of the arguments are passed to the parameters. Changes to parameters directly affect the original arguments.   Short Explanation:   * Function Parameters: Variables in a function that receive values from the arguments when the function is called. * Function Arguments: Values passed to a function when calling it, corresponding to the parameters. * Passing: Arguments are passed to parameters either by value (copying the value) or by reference (passing the address).   **21**) Explain the concept of arrays in C programming. How are arrays declared and initialized?  **A.** Arrays in C are collections of elements of the same data type, stored in contiguous memory locations. They are declared by specifying the data type of elements and the number of elements. Arrays can be initialized either at the time of declaration or separately using assignment statements. Elements in an array are accessed using index notation.  **22)** Discuss the difference between a one-dimensional array and a multi-dimensional array in C programming. Provide examples of both   1. **A. One-dimensional Array**:    * Example: Storing a list of integers.    * Explanation: It's a linear collection of elements stored in contiguous memory locations. Accessing elements requires only a single index.   Example:  #include <stdio.h>  int main() {  int arr[5] = {1, 2, 3, 4, 5}; // One-dimensional array declaration and initialization    // Accessing and printing elements of the one-dimensional array  for (int i = 0; i < 5; i++) {  printf("%d ", arr[i]);  }    return 0;   1. }**Multi-dimensional Array**:    * Example: Storing a matrix of integers.    * Explanation: It's an array of arrays, where each element is itself an array. Accessing elements requires specifying multiple indices, one for each dimension.   Example:  #include <stdio.h>  int main() {  int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // Two-dimensional array declaration and initialization    // Accessing and printing elements of the two-dimensional array  for (int i = 0; i < 3; i++) {  for (int j = 0; j < 3; j++) {  printf("%d ", arr[i][j]);  }  printf("\n");  }    return 0;  }  **23)** Describe the process of accessing array elements in C programming. How are array indices used to access elements  **A.** In C programming, accessing array elements involves specifying an index within square brackets after the array name. Indices start from 0, and you can use either constants or variables as indices to access elements. Ensure that the index remains within the bounds of the array to avoid undefined behavior or segmentation faults.  **Top of Form**  **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.  The** null character (**'\0'**) in C strings marks the end of a string. It is used by C string functions to determine where the string ends. Strings in C are represented as arrays of characters terminated by the null character. This character has a value of zero and is automatically added to the end of string literals by the compiler. When processing strings in C, functions iterate through the characters until they encounter the null character to determine the end of the string.  **25)** Explain the concept of dynamic memory allocation for arrays in C programming. How are dynamic arrays allocated and deallocated?  **A.**  Dynamic memory allocation in C allows for the creation of arrays whose size is determined during runtime. This is done using functions like **malloc()**, **calloc()**, and **realloc()** to allocate memory and **free()** to deallocate it. These functions provide flexibility in managing memory, but it's important to handle allocation failures and free memory when it's no longer needed to avoid memory leaks.  **26)** Describe the purpose and usage of pointers in C programming. How are pointers declared and initialized?  **A**.  Pointers in C are variables that store memory addresses, allowing for indirect access to data. They are declared using the asterisk () symbol followed by the data type they point to. Pointers can be initialized with the address of a variable using the address-of operator (&). Dereferencing a pointer accesses the value stored at the memory address it points to, done using the dereference operator (). Pointers are commonly used for dynamic memory allocation, passing parameters by reference, and implementing complex data structures.  **27)** Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers.  **A.**  Pointer arithmetic in C allows you to perform arithmetic operations on pointers, such as addition and subtraction. When you add an integer value to a pointer, it increments the pointer by **n \* sizeof(type)** bytes, where **n** is the integer value, and **type** is the data type the pointer points to. Similarly, subtracting an integer value from a pointer decrements it by **n \* sizeof(type)** bytes.  Example of pointer addition:  int arr[] = {10, 20, 30, 40, 50};  int \*ptr = arr; // Pointer points to the first element of the array  ptr++; // Incrementing the pointer to point to the next element  Example of pointer subtraction:  int arr[] = {10, 20, 30, 40, 50};  int \*ptr = &arr[3]; // Pointer points to the fourth element of the array  ptr--; // Decrementing the pointer to point to the previous element  Pointer arithmetic is commonly used to navigate arrays and dynamically allocated memory, providing efficient access to elements without using array indices. However, caution should be exercised to ensure that pointers remain within the bounds of allocated memory to avoid undefined behavior.  **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.   1. A. **Pass by Value**:    * In pass by value, a copy of the argument's value is passed to the function.    * Modifications made to the parameter inside the function do not affect the original value outside the function.    * Example:   #include <stdio.h>  void increment(int x) {  x++; // Increment the copy of x  printf("Inside function: %d\n", x);  }  int main() {  int num = 5;  increment(num); // Pass by value  printf("Outside function: %d\n", num); // Original value remains unchanged  return 0;  } Output:  Inside function: 6  Outside function: 5  **Pass by Reference (using Pointers)**:   * + In pass by reference, the address (memory location) of the argument is passed to the function using pointers.   + Changes made to the parameter inside the function are reflected outside the function.   + Example:   #include <stdio.h>  void incrementByRef(int \*x) {  (\*x)++; // Increment the value at the address pointed by x  printf("Inside function: %d\n", \*x);  }  int main() {  int num = 5;  incrementByRef(&num); // Pass by reference using pointers  printf("Outside function: %d\n", num); // Original value is modified  return 0;  } Output:  Inside function: 6  Outside function: 6 6  In essence, pass by value creates a copy of the value, while pass by reference (using pointers) allows direct access to the original value. Use pass by value when you want to work with a copy of the value, and use pass by reference when you want to modify the original value efficiently.  **29).** Describe the concept of NULL pointers in C programming. How are NULL pointers used and checked for in programs? give shortly  **A.** In C programming, a NULL pointer is a pointer that does not point to any memory location. It's commonly used to represent an invalid or uninitialized pointer.   * **Concept**: NULL is a macro defined in several standard libraries, usually as **(void \*)0**. It's often used to initialize pointers when they are declared or to explicitly indicate that a pointer doesn't point to anything valid. * **Usage**: NULL pointers are checked in programs to prevent dereferencing invalid pointers, which can lead to undefined behavior or crashes. They're commonly used in conditions to ensure pointers are valid before accessing the memory they point to. * **Checking for NULL**: You can check if a pointer is NULL using an if statement or in conditions where pointer validity matters. For example:   int \*ptr = NULL;  // Check if ptr is NULL before dereferencing it  if (ptr != NULL) {  // Do something with ptr  } else {  // Handle NULL pointer case  } NULL pointers in C represent invalid or uninitialized pointers and are used to avoid accessing invalid memory locations. They're checked in programs to ensure pointer validity before dereferencing them, helping prevent runtime errors.  **30)**Explain the role of pointers in dynamic memory allocation in C programming. How are pointers used to allocate and deallocate memory dynamically? In C programming, pointers are crucial for dynamic memory allocation, enabling programs to request memory at runtime using functions like **malloc()** and **calloc()**. Pointers are used to access and manage dynamically allocated memory, ensuring efficient memory usage and preventing memory leaks. When memory is no longer needed, pointers are used to deallocate it using the **free()** function. Dynamic memory allocation with pointers is fundamental for implementing dynamic data structures like linked lists and trees.  **31)** Discuss the concept of strings in C programming. How are strings represented and manipulated in C?  The role of pointers in dynamic memory allocation in C programming is vital. Pointers allow programs to allocate and deallocate memory dynamically, providing flexibility in memory management. Here's a brief overview:   * **Allocation**: Pointers are used with functions like **malloc()**, **calloc()**, or **realloc()** to allocate memory on the heap at runtime. These functions return a pointer to the allocated memory block, which can be used to access and manipulate the allocated memory.   Example:  int \*ptr = (int \*)malloc(sizeof(int));   * **Deallocation**: Pointers are used to deallocate dynamically allocated memory using the **free()** function. This prevents memory leaks and ensures efficient memory usage by releasing memory when it's no longer needed.   Example:  free(ptr);   * **Dynamic Data Structures**: Pointers are essential for implementing dynamic data structures like linked lists, trees, and graphs. These structures require dynamic memory allocation and deallocation to manage their size and structure efficiently. * **Memory Management**: Pointers enable efficient memory management by allowing programs to allocate memory as needed during runtime. This flexibility helps optimize memory usage and prevents wastage of resources.   In summary, pointers play a crucial role in dynamic memory allocation in C programming, allowing programs to allocate and deallocate memory as needed, enabling efficient memory management and the implementation of dynamic data structures  **32).** Explain the difference between character arrays and string literals in C programming. Provide examples to illustrate both concepts.   * **Character Arrays**:   + Mutable sequences of characters stored in contiguous memory locations.   + Terminated by a null character **'\0'**.   + Can be modified directly.   + Provide flexibility for storing and manipulating strings. * **String Literals**:   + Constant sequences of characters enclosed in double quotation marks (**"**).   + Automatically null-terminated by the compiler.   + Immutable; attempts to modify result in undefined behavior.   + Primarily used for representing constant strings in code.   Understanding these differences is crucial for effective string handling in C programs, allowing developers to choose the appropriate approach based on their requirements.  **33)** Describe common string manipulation functions available in the C standard library. Provide examples of functions like strlen, strcpy, strcat, and strcmp?  Here's a description of common string manipulation functions available in the C standard library, along with examples for each:   1. **strlen()**:    * The **strlen()** function calculates the length of a string by counting the number of characters before the null terminator **'\0'**.   Example:  #include <stdio.h>  #include <string.h>  int main() {  char str[] = "Hello, world!";  int length = strlen(str);  printf("Length of string: %d\n", length); // Output: Length of string: 13  return 0;   1. }**strcpy()**:    * The **strcpy()** function copies a string from the source to the destination, including the null terminator. It overwrites the existing content of the destination string.   Example:  #include <stdio.h>  #include <string.h>  int main() {  char src[] = "Hello";  char dest[10];  strcpy(dest, src);  printf("Copied string: %s\n", dest); // Output: Copied string: Hello  return 0;   1. }**strcat()**:    * The **strcat()** function appends a copy of the source string to the end of the destination string, overwriting the null terminator of the destination and then adding a new null terminator at the end.   Example:  #include <stdio.h>  #include <string.h>  int main() {  char dest[20] = "Hello";  char src[] = ", world!";  strcat(dest, src);  printf("Concatenated string: %s\n", dest); // Output: Concatenated string: Hello, world!  return 0;   1. }**strcmp()**:    * The **strcmp()** function compares two strings lexicographically. It returns an integer less than, equal to, or greater than zero if the first string is found to be less than, equal to, or greater than the second string, respectively.   Example:  #include <stdio.h>  #include <string.h>  int main() {  char str1[] = "apple";  char str2[] = "banana";  int result = strcmp(str1, str2);  if (result < 0) {  printf("str1 is less than str2\n");  } else if (result > 0) {  printf("str1 is greater than str2\n");  } else {  printf("str1 is equal to str2\n");  }  return 0;  }These functions are just a subset of the string manipulation functions provided by the C standard library. Other commonly used functions include **strncmp()**, **strncpy()**, **strstr()**, **strchr()**, and more. They offer powerful tools for handling strings efficiently in C programming.  **34)** Discuss the concept of string tokenization in C programming. How are strings split into tokens using delimiter characters?  String tokenization in C programming involves breaking a string into smaller parts, called tokens, based on specified delimiter characters. This is commonly done using the **strtok()** function, which replaces delimiter characters with null characters to separate tokens. Subsequent calls to **strtok()** with a NULL pointer as the first argument continue tokenizing from the last position. Custom delimiters can be specified, and multiple delimiters can be used. It's important to note that **strtok()** modifies the original string and is not thread-safe, so **strtok\_r()** should be used in multithreaded programs. Overall, string tokenization is essential for parsing input, extracting information, and text processing in C programming.  **35**) Explain the importance of null-terminated strings in C programming. How does the null character ('\0') signify the end of a string?  Null-terminated strings are fundamental in C programming as they provide a convenient and efficient way to represent strings of characters. The null character **'\0'** serves as a sentinel value, marking the end of the string. Here's why null-terminated strings are important:   1. **End of String**: The null character **'\0'** is used to signify the end of a string. When a C-style string is stored in memory, each character is placed sequentially, and the string is terminated by a null character. This allows functions that operate on strings to determine the end of the string and prevents them from accessing memory beyond the end of the string. 2. **String Manipulation**: Most string manipulation functions in C, such as **strlen()**, **strcpy()**, **strcat()**, and **strcmp()**, rely on null-terminated strings. These functions iterate over the characters of a string until they encounter the null character, allowing them to perform their operations safely and efficiently. 3. **Memory Efficiency**: Null-terminated strings are memory-efficient because they store only the characters of the string and do not require additional metadata to store the length of the string. This simplicity makes null-terminated strings lightweight and suitable for memory-constrained environments. 4. **Backward Compatibility**: Null-terminated strings are part of the C programming language's heritage and are widely used in existing codebases and libraries. Understanding null-terminated strings is essential for working with legacy code and integrating with existing C libraries. 5. **Compatibility with Standard Library Functions**: The C standard library provides many functions for working with null-terminated strings. By adhering to the null-terminated string convention, C programs can take advantage of these functions to perform common string operations efficiently.   Null-terminated strings are essential in C programming for representing strings of characters. The null character **'\0'** serves as a sentinel value to mark the end of the string, allowing for safe and efficient string manipulation operations. Understanding null-terminated strings is crucial for effective C programming and interoperability with existing C codebases and libraries.  **36**) Describe the purpose and usage of structures in C programming. How are structures declared and accessed?  Structures in C programming allow grouping related data under a single name. They're declared using the **struct** keyword, followed by the structure's name and its members. Structures are used to define custom data types, group related data, and pass complex data to functions. Members of a structure are accessed using the dot **.** operator. Overall, structures provide a way to organize and manage complex data effectively in C programs.  **37)** Discuss the concept of structure members in C programming. How are individual members of a structure accessed and modified?  **A.**  In C programming, structure members are individual variables or data elements within a structure. They are accessed and modified using the dot **.** operator. This allows you to work with the different attributes of a structure separately.  **38**)Explain the difference between structures and unions in C programming. When would you choose one over the other?  **A.** In C programming, structures and unions are both used to group together multiple variables of different data types under a single name. However, they have distinct differences in their behavior and usage.  **Structures:**   * **Definition:** Structures allow you to define composite data types containing multiple members of possibly different data types. * **Memory Allocation:** Each member of a structure is allocated its own memory space. The total memory required for a structure is the sum of the memory required for each member. * **Usage:** Structures are typically used when you want to group related data that may have different data types. Each member retains its own separate memory space.   **Unions:**   * **Definition:** Unions also allow you to define composite data types containing multiple members, but all members share the same memory location. * **Memory Allocation:** Unlike structures, where each member has its own memory space, unions allocate memory such that all members share the same memory location. As a result, the size of a union is determined by the largest member. * **Usage:** Unions are often used when you want to store different types of data in the same memory location. They are useful for conserving memory when you only need one member of the union to be active at any given time.   **Choosing Between Structures and Unions:**   * Use structures when you need to group related data that may have different data types and you want each member to have its own memory space. * Use unions when you need to conserve memory and only one member of the union needs to be active at any given time.   **Example:**  // Example of a structure  struct Person {  char name[50];  int age;  float height;  };  // Example of a union  union Data {  int intValue;  float floatValue;  char stringValue[20];  }; structures and unions in C programming serve similar purposes of grouping multiple variables, but they differ in how they allocate memory for their members. Structures allocate separate memory for each member, while unions share the same memory location for all members. The choice between structures and unions depends on your specific requirements for data grouping and memory usage.  **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 another structure. This concept allows you to organize and group related data in a hierarchical manner. Just like regular structure members, nested structures can have their own members, including other nested structures.  **Defining Nested Structures:**  Nested structures are defined by declaring a structure within the definition of another structure. Here's the syntax:  struct OuterStructure {  // Outer structure members  int outerMember;    // Nested structure  struct InnerStructure {  // Inner structure members  int innerMember;  float innerFloat;  } inner;  };In this example, **InnerStructure** is a nested structure defined within **OuterStructure**.  **Accessing Nested Structure Members:**  You can access members of nested structures using the dot **.** operator in a hierarchical manner. Here's how you can access members of nested structures:  // Declare a variable of type OuterStructure  struct OuterStructure outerVar;  // Access outer structure member  outerVar.outerMember = 10;  // Access nested structure member  outerVar.inner.innerMember = 20;  outerVar.inner.innerFloat = 3.14;In this example, **inner** is a member of **OuterStructure**, which itself is a structure. To access members of the nested structure **InnerStructure**, you first access **inner** using the dot **.** operator, and then access its members in the same way.  **Example:**  #include <stdio.h>  // Define a nested structure  struct Address {  char city[50];  char state[50];  };  struct Person {  char name[50];  int age;  struct Address address;  };  int main() {  // Declare a variable of type Person  struct Person person;  // Assign values to members  strcpy(person.name, "John");  person.age = 30;  strcpy(person.address.city, "New York");  strcpy(person.address.state, "NY");  // Print the details  printf("Name: %s\n", person.name);  printf("Age: %d\n", person.age);  printf("Address: %s, %s\n", person.address.city, person.address.state);  return 0;  }In summary, nested structures in C programming allow you to organize related data in a hierarchical manner. They are defined by declaring a structure within the definition of another structure. Nested structure members can be accessed using the dot **.** operator in a hierarchical manner  **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 programming, **typedef** is used to create aliases for existing data types, including primitive types and user-defined structures and unions. It enhances code readability, simplifies maintenance, and facilitates the declaration of variables with complex data types.  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 involves working with files on the system, including opening, reading from, writing to, and closing files. The standard library **<stdio.h>** provides functions for performing file operations in C.  **Basic Concepts:**   1. **File Pointer:**    * A file pointer is a variable used to keep track of the current position within a file. It is represented by the **FILE** data type.    * File pointers are used to perform various operations on files, such as opening, reading, writing, and closing. 2. **File Modes:**    * When opening a file, you specify a mode that determines the type of operations that can be performed on the file.    * Common file modes include **"r"** for reading, **"w"** for writing (creating a new file or overwriting an existing file), and **"a"** for appending (writing to the end of an existing file).   **File Handling Functions:**   1. **Opening a File:**    * To open a file, you use the **fopen()** function, which takes two arguments: the file name (including the path) and the mode in which to open the file.    * If the file is opened successfully, **fopen()** returns a pointer to the **FILE** structure representing the file. If there is an error, it returns **NULL**. 2. **Reading from a File:**    * To read from a file, you use functions like **fscanf()** or **fgets()** to read formatted or string input, respectively, from the file.    * You can also use functions like **fread()** to read a specific number of bytes from the file. 3. **Writing to a File:**    * To write to a file, you use functions like **fprintf()** or **fputs()** to write formatted or string output, respectively, to the file.    * You can also use functions like **fwrite()** to write a specific number of bytes to the file. 4. **Closing a File:**    * After performing file operations, it's essential to close the file using the **fclose()** function.    * Closing a file ensures that any buffered data is written to the file and releases system resources associated with the file.   **Example:**  #include <stdio.h>  int main() {  FILE \*filePtr;  char text[100];  // Open file for writing  filePtr = fopen("example.txt", "w");    // Write to the file  fprintf(filePtr, "Hello, world!\n");  // Close the file  fclose(filePtr);  // Open file for reading  filePtr = fopen("example.txt", "r");    // Read from the file  fscanf(filePtr, "%[^\n]", text);  // Print the read text  printf("Data from file: %s\n", text);  // Close the file  fclose(filePtr);  return 0;  }In this example, we open a file named "example.txt" for writing, write "Hello, world!" to it, and then close the file. Later, we open the same file for reading, read the content using **fscanf()**, and print it. Finally, we close the file again.  In summary, file handling in C involves opening, reading from, writing to, and closing files using standard file handling functions provided by the **<stdio.h>** library. Understanding file handling is essential for working with files in C programs, such as reading input from files, writing output to files, and storing data persistently.  **42)**Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files?  A. File pointers in C programming serve as handles to manage file operations such as reading from, writing to, and navigating within files. They keep track of the current position within a file and facilitate various file operations. Here's a brief overview:   1. **Opening Files:** File pointers are used to open files using the **fopen()** function. This function returns a file pointer that points to the beginning of the file. 2. **Navigating within Files:** File pointers are moved within a file using functions like **fseek()** and **rewind()**. **fseek()** allows you to move the file pointer to a specified position within the file, while **rewind()** moves the file pointer to the beginning of the file. 3. **Reading from Files:** File pointers are used to read data from files using functions like **fread()**, **fscanf()**, and **fgets()**. These functions read data from the file at the current position indicated by the file pointer. 4. **Writing to Files:** File pointers are used to write data to files using functions like **fwrite()**, **fprintf()**, and **fputs()**. These functions write data to the file at the current position indicated by the file pointer. 5. **Closing Files:** After performing file operations, file pointers are used to close files using the **fclose()** function. This ensures that any buffered data is written to the file and releases system resources associated with the file.   In summary, file pointers play a crucial role in managing file operations in C programming by facilitating navigation and manipulation of files. They are used to open, navigate, re  **43)** Discuss the difference between text files and binary files in C programming. How are they opened and processed differently?  A. In C programming, text files and binary files are two primary types of files used for storing data, and they differ in their representation and how they are processed.  **Text Files:**   * **Representation:** Text files store data in a human-readable format, where each character is represented using a character encoding such as ASCII or UTF-8. Text files typically contain plain text, such as letters, numbers, and symbols, and are terminated by special characters like newline (**'\n'**) or carriage return (**'\r'**). * **Content:** Text files are primarily used for storing textual data, such as source code, configuration files, and documents. Each character in a text file has a one-to-one correspondence with its representation in memory. * **Opening and Processing:** Text files are opened and processed using standard input/output functions like **fopen()**, **fscanf()**, **fprintf()**, **fgets()**, and **fputs()**. These functions handle text data as strings and perform operations like reading, writing, and manipulating text.   **Binary Files:**   * **Representation:** Binary files store data in a binary format, where data is represented as sequences of bytes with no specific character encoding. Binary files can contain any type of data, including numeric data, images, audio, video, and structured data. * **Content:** Binary files can store any type of data, including non-textual data and complex data structures. They are used for storing data in its raw, machine-readable form, preserving the exact representation of the data in memory. * **Opening and Processing:** Binary files are opened and processed using standard file handling functions like **fopen()**, **fread()**, **fwrite()**, and **fclose()**. These functions operate on raw bytes of data and are not concerned with the textual representation of the data. Binary files require careful handling to ensure that data is read and written correctly, especially when dealing with data types of different sizes and endianness.   **Key Differences:**   1. **Representation:** Text files store data in a human-readable format using character encoding, while binary files store data in its raw, binary form. 2. **Content:** Text files are used for storing textual data, while binary files can store any type of data. 3. **Processing:** Text files are processed using text-oriented functions that handle strings, while binary files are processed using functions that operate on raw bytes of data.   In summary, text files and binary files serve different purposes and are processed differently in C programming. Text files are suitable for storing textual data and are processed using text-oriented functions, while binary files are used for storing any type of data and require handling raw bytes of data using binary-oriented functions.  **44**). Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc?  A. In C programming, file modes are used to specify the type of operations that can be performed on a file when it is opened. They determine whether the file will be opened for reading, writing, or appending, and whether the file must already exist or not. File modes are passed as the second argument to the **fopen()** function when opening a file.  Here are the most commonly used file modes in C programming:   1. **"r" (Read Mode):**    * Opens a file for reading.    * If the file does not exist, fopen() returns NULL.    * The file pointer is positioned at the beginning of the file.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "r");  2.**"w" (Write Mode):**   * + Opens a file for writing.   + If the file does not exist, it creates a new file. If the file exists, it truncates the file to zero length.   + The file pointer is positioned at the beginning of the file.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "w");  3.**"a" (Append Mode):**   * + Opens a file for appending.   + If the file does not exist, it creates a new file. If the file exists, it appends data to the end of the file without truncating it.   + The file pointer is positioned at the end of the file.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "a");  4.**"r+" (Read/Write Mode):**   * + Opens a file for both reading and writing.   + The file must exist; otherwise, fopen() returns NULL.   + The file pointer is positioned at the beginning of the file.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "r+");  **5."w+" (Read/Write Mode):**   * + Opens a file for both reading and writing.   + If the file does not exist, it creates a new file. If the file exists, it truncates the file to zero length.   + The file pointer is positioned at the beginning of the file.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "w+");  **6. "a+" (Append/Read Mode):**   * 1. Opens a file for reading and appending.   2. If the file does not exist, it creates a new file. If the file exists, it appends data to the end of the file without truncating it.   3. The file pointer is positioned at the end of the file.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "a+");  These file modes provide flexibility in how files are opened and processed in C programming, allowing for various read and write operations on files. It's essential to choose the appropriate file mode based on the intended file operations to ensure correct behavior.  **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 file operations in C programming involves detecting and responding to errors that may occur while performing file-related operations such as opening, reading from, writing to, or closing files. Here are some common error handling techniques used in file operations:   1. **Check Return Values:**    * Functions like **fopen()**, **fclose()**, **fread()**, **fwrite()**, etc., return special values (like **NULL** or **EOF**) to indicate errors.    * Always check the return value of these functions after calling them to ensure that the operation was successful.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "r");  if (filePtr == NULL) {  perror("Error opening file");  exit(EXIT\_FAILURE);  }\_FAILURE); }   1. **Use perror():**    * The **perror()** function can be used to print a descriptive error message to stderr, based on the value of the **errno** variable.    * It helps in identifying the cause of the error and provides useful information for debugging.   Example:  FILE \*filePtr;  filePtr = fopen("example.txt", "r");  if (filePtr == NULL) {  perror("Error opening file");  exit(EXIT\_FAILURE);   1. }**Handle File I/O Errors:**    * Use error handling mechanisms to deal with errors that occur during file input/output operations.    * For example, check for errors while reading from or writing to files using functions like **fread()**, **fwrite()**, **fscanf()**, **fprintf()**, etc. 2. **Cleanup Resources:**    * Make sure to close the file properly after handling any errors to release system resources associated with the file.    * Use **fclose()** to close the file and free up any system resources it may be using. 3. **Graceful Termination:**    * If an error occurs that cannot be recovered from, terminate the program gracefully to avoid leaving resources in an inconsistent state.    * Use functions like **exit()** or **return** with appropriate exit codes to indicate failure. 4. **Logging and Debugging:**    * Log error messages along with relevant information such as file names, line numbers, and error codes for debugging purposes.    * Use logging libraries or custom logging functions to maintain a record of errors encountered during file operations.   By employing these error handling techniques, you can ensure that your C programs handle file-related errors gracefully and provide meaningful feedback to users in case of failures. Proper error handling improves the robustness and reliability of file processing code. |

**Part- B**

|  |
| --- |
| 1. **Title of the program** |
| Code  1)**Hello world code**  #include<stdio.h>  #include<ctype.h>  int main()  {  char a[10];  gets(a);      puts(a);  }  **(2) : Calculate the factorial of a given number.**  #include<stdio.h>  main()  {  int number, factorial=1,i;  printf("Enter a number for factorial\n");  scanf("%d",& number);  for(i=1;i<=number;i++)  {  factorial=factorial\*i;  }  printf("%d!=%d\n", number, factorial);  }  **(3) : Determine whether a given number is prime**  #include<stdio.h>  main()  {  int num,i,count=0;  printf("Enter a number to check whether it is prime or not\n");  scanf("%d",&num);  for(i=1;i<=num;i++)  {  if(num%i==0)  {  count++;  }  }  if(count==2)  {  printf("The given number %d is Prime Number\n,num");  }  else  {  printf("The given number %d is not Prime Number\n");  for(i=1;i<=num;i++)  {  if(num%i==0)  {  printf("%d\n",i);  }  }  }  }  **(4) : Generate the Fibonacci series up to a certain limit**.  #include <stdio.h>  int main()  {  int prev2 = 0, prev1 = 1;  int newFibo;  printf("%d\n", prev2);  printf("%d\n", prev1);  for(int fibo = 0; fibo < 18; fibo++) {  newFibo = prev1 + prev2;  printf("%d\n", newFibo);  prev2 = prev1;  prev1 = newFibo;  }  return 0;  }  **(5): Calculate the sum of digits of a given number.**  #include<stdio.h>  main()  {  int dummy,n,sum=0,x;  printf("Enter a number\n");  scanf("%d",&n);  dummy=n;  while(n>0)  {  x=n%10;  sum=sum+x;  n=n/10;  }  **(6) : Reverse the digits of a given number**  #include<stdio.h>  main()  {  int dummy,n,rev=0,x;  printf("Enter a number\n");  scanf("%d",&n);  dummy=n;  while(n>0)  {  x=n%10;  rev=rev\*10+x;  n=n/10;  }  printf("The reverse of %d is %d\n",dummy,rev);  }  printf("The sum of all digit in %d is %d\n",dummy,sum);  **(7): Check if a given number or string is a palindrome**.  #include<stdio.h>  #include<string.h>  main()  {  int i,j=0;  char str1[100],str2[100]={0};  printf("Enter a string to know whether it is palindrome or not\n");  gets(str1);  for(i=strlen(str1)-1;i>=0;i--)  {  str2[j]=str1[i];  j++;  }  str2[j]='\0';  printf("The reverse of the given string is---> %s\n",str2);  if(strcmp(str1,str2)==0)  {  printf("The given string is Palindrome\n");  }  else  {  printf("The given string is Not Palindrome\n");  }  }  **8) Calculate the area of shapes like rectangle, triangle, and circle**  #include<stdio.h>  #include<math.h>  int main() {  int choice;  printf("Enter\n1 for Triangle\n2 for Square\n3 for Circle\n4 for Rectangle\n5 for Parallelogram\n");  scanf("%d", &choice);    switch(choice) {  case 1: {  int a, b, c;  float s, area;  printf("Enter sides of triangle\n");  scanf("%d %d %d", &a, &b, &c);  s = (float)(a + b + c) / 2;  area = sqrt(s \* (s - a) \* (s - b) \* (s - c));  printf("Area of Triangle with sides %d,%d,%d is %f\n", a, b, c, area);  break;  }  case 2: {  float side, area;  printf("Enter Side of Square\n");  scanf("%f", &side);  area = side \* side;  printf("Area of Square is %f\n", area);  break;  }  case 3: {  float radius, area;  printf("Enter Radius of Circle\n");  scanf("%f", &radius);  area = 3.14159 \* radius \* radius;  printf("Area of Circle with radius %f is %f\n", radius, area);  break;  }  case 4: {  float length, breadth, area;  printf("Enter Length and Breadth of Rectangle\n");  scanf("%f %f", &length, &breadth);  area = length \* breadth;  printf("Area of Rectangle is %f\n", area);  break;  }  case 5: {  float base, height, area;  printf("Enter base and height of Parallelogram\n");  scanf("%f %f", &base, &height);  area = base \* height;  printf("Area of Parallelogram is %f\n", area);  break;  }  default: {  printf("Invalid Choice\n");  break;  }  }  return 0;  }  **9)** **Implement a basic calculator with arithmetic operations.**  #include<stdio.h>  char ch;  int main()  {  int a,b,c;  while(1)  {  printf("enter case\n");  scanf("%c",&ch);  printf("enter two no's:\n");  scanf("%d",&a);  scanf("%d",&b);  switch(ch)  {  case '+':  c=a+b;  break;  case'-':  c=a-b;  break;  case'\*':  c=a\*b;  break;  case'/':  c=a/b;  break;  }  printf("%d\n",c);  }      return 0;  }  **10): Perform operations like finding the largest/smallest element, sum,and average of an array**.  #include <stdio.h>  #define MAX\_SIZE 100  int main() {  int arr[MAX\_SIZE];  int i, n;  int sum = 0;  float average;  int largest, smallest;  printf("Enter the number of elements in the array (max %d): ", MAX\_SIZE);  scanf("%d", &n);  printf("Enter %d elements:\n", n);  for (i = 0; i < n; i++) {  scanf("%d", &arr[i]);  sum += arr[i];  }  // Finding largest and smallest elements  largest = smallest = arr[0];  for (i = 1; i < n; i++) {  if (arr[i] > largest)  largest = arr[i];  if (arr[i] < smallest)  smallest = arr[i];  }  // Calculating average  average = (float)sum / n;  printf("Sum of elements: %d\n", sum);  printf("Average of elements: %.2f\n", average);  printf("Largest element: %d\n", largest);  printf("Smallest element: %d\n", smallest);  return 0;  }  **11) :Manipulate strings such as concatenation, copying, and comparison**  #include <stdio.h>  #define MAX\_SIZE 100  int linearSearch(int arr[], int size, int key) {  for (int i = 0; i < size; i++) {  if (arr[i] == key)  return i; // Return the index if the key is found  }  return -1; // Return -1 if the key is not found  }  int main() {  int arr[MAX\_SIZE];  int size, key;  printf("Enter the number of elements in the array (max %d): ", MAX\_SIZE);  scanf("%d", &size);  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Enter the element to search: ");  scanf("%d", &key);  int index = linearSearch(arr, size, key);  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;  }  **(12): Linear Search: Search for an element in an array using linear search.**  #include <stdio.h>  #define MAX\_SIZE 100  int linearSearch(int arr[], int size, int key) {  for (int i = 0; i < size; i++) {  if (arr[i] == key)  return i; // Return the index if the key is found  }  return -1; // Return -1 if the key is not found  }  int main() {  int arr[MAX\_SIZE];  int size, key;  printf("Enter the number of elements in the array (max %d): ", MAX\_SIZE);  scanf("%d", &size);  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Enter the element to search: ");  scanf("%d", &key);  int index = linearSearch(arr, size, key);  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;  }  **(13): Binary Search: Search for an element in a sorted array using binary search.**  #include <stdio.h>  #define MAX\_SIZE 100  void selectionSort(int arr[], int size) {  int i, j, minIndex, temp;  for (i = 0; i < size - 1; i++) {  minIndex = i;  for (j = i + 1; j < size; j++) {  if (arr[j] < arr[minIndex])  minIndex = j;  }  if (minIndex != i) {  // Swap arr[i] and arr[minIndex]  temp = arr[i];  arr[i] = arr[minIndex];  arr[minIndex] = temp;  }  }  }  int main() {  int arr[MAX\_SIZE];  int size;  printf("Enter the number of elements in the array (max %d): ", MAX\_SIZE);  scanf("%d", &size);  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  // Sort the array using selection sort  selectionSort(arr, size);  printf("Sorted array in ascending order:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  **(14): Selection Sort: Sort an array using the selection sort algorithm.**  #include <stdio.h>  #define MAX\_SIZE 100  void bubbleSort(int arr[], int size) {  int i, j, temp;  for (i = 0; i < size - 1; i++) {  for (j = 0; j < size - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  // Swap arr[j] and arr[j + 1]  temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  }  int main() {  int arr[MAX\_SIZE];  int size;  printf("Enter the number of elements in the array (max %d): ", MAX\_SIZE);  scanf("%d", &size);  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  // Sort the array using bubble sort  bubbleSort(arr, size);  printf("Sorted array in ascending order:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  **(15): Bubble Sort: Sort an array using the bubble sort algorithm.**  #include <stdio.h>  #define MAX\_SIZE 100  void insertionSort(int arr[], int size) {  int i, j, key;  for (i = 1; i < size; i++) {  key = arr[i];  j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key;  }  }  int main() {  int arr[MAX\_SIZE];  int size;  printf("Enter the number of elements in the array (max %d): ", MAX\_SIZE);  scanf("%d", &size);  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  // Sort the array using insertion sort  insertionSort(arr, size);  printf("Sorted array in ascending order:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  }  **16.: Sort an array using the insertion sort algorithm**  #include <stdio.h>  // Function to perform insertion sort  void insertionSort(int arr[], int n) {  int i, key, j;  for (i = 1; i < n; i++) {  key = arr[i];  j = i - 1;  /\* Move elements of arr[0..i-1], that are greater than key, to one position ahead  of their current position \*/  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key;  }  }  void printArray(int arr[], int n) {  int i;  for (i = 0; i < n; i++)  printf("%d ", arr[i]);  printf("\n");  }  int main() {  int arr[] = {12, 11, 13, 5, 6};  int n = sizeof(arr) / sizeof(arr[0]);  printf("Original array: \n");  printArray(arr, n);  insertionSort(arr, n);  printf("Sorted array: \n");  printArray(arr, n);  return 0;  }  **17)** **Perform matrix addition, subtraction, multiplication, and transpose.**  #include <stdio.h>  #define N 3 // Assuming matrices are of size 3x3  void addMatrix(int A[][N], int B[][N], int C[][N]) {  int i, j;  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  C[i][j] = A[i][j] + B[i][j];  }  }  }  void subtractMatrix(int A[][N], int B[][N], int C[][N]) {  int i, j;  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  C[i][j] = A[i][j] - B[i][j];  }  }  }  void multiplyMatrix(int A[][N], int B[][N], int C[][N]) {  int i, j, k;  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  C[i][j] = 0;  for (k = 0; k < N; k++) {  C[i][j] += A[i][k] \* B[k][j];  }  }  }  }  void transposeMatrix(int A[][N], int C[][N]) {  int i, j;  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  C[j][i] = A[i][j];  }  }  }  void printMatrix(int A[][N]) {  int i, j;  for (i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%d\t", A[i][j]);  }  printf("\n");  }  }  int main() {  int A[N][N] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  int B[N][N] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};  int C[N][N];  printf("Matrix A:\n");  printMatrix(A);  printf("\nMatrix B:\n");  printMatrix(B);  printf("\nAddition:\n");  addMatrix(A, B, C);  printMatrix(C);  printf("\nSubtraction:\n");  subtractMatrix(A, B, C);  printMatrix(C);  printf("\nMultiplication:\n");  multiplyMatrix(A, B, C);  printMatrix(C);  printf("\nTranspose of Matrix A:\n");  transposeMatrix(A, C);  printMatrix(C);  return 0;  } |
| Output  2)  3)  4)  5)  6)  7)  8)  9)  10)  11)    12)    13)    14)  15)  16)  17) |

**Part- C**

|  |
| --- |
| **Programs** |
| **1). Write a C program to find the minimum element in a queue.**  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the queue  // Structure to represent the queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  int size;  int minElement; // Variable to store the minimum element  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = -1;  queue->rear = -1;  queue->size = 0;  queue->minElement = INT\_MAX; // Initialize minElement with maximum possible integer value  }  // Function to check if the queue is empty  bool isEmpty(struct Queue \*queue) {  return (queue->size == 0);  }  // Function to check if the queue is full  bool isFull(struct Queue \*queue) {  return (queue->size == MAX\_SIZE);  }  // Function to enqueue an element into the queue  void enqueue(struct Queue \*queue, int value) {  if (isFull(queue)) {  printf("Queue overflow! Cannot enqueue more elements.\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  queue->size++;  // Update minElement if the enqueued value is smaller  if (value < queue->minElement) {  queue->minElement = value;  }  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedItem = queue->items[queue->front];  queue->front = (queue->front + 1) % MAX\_SIZE;  queue->size--;  // Recalculate minElement if the dequeued element was the minimum  if (dequeuedItem == queue->minElement) {  queue->minElement = INT\_MAX;  for (int i = queue->front; i <= queue->rear; i++) {  if (queue->items[i] < queue->minElement) {  queue->minElement = queue->items[i];  }  }  }  return dequeuedItem;  }  // Function to store the minimum element in the queue in a text file  void storeMinElementToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write the minimum element to the file  fprintf(file, "Minimum element in the queue: %d\n", queue->minElement);  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Enqueue some elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 5);  enqueue(&queue, 20);  enqueue(&queue, 3);  enqueue(&queue, 15);  // Dequeue some elements from the queue  dequeue(&queue);  dequeue(&queue);  // Store the minimum element in the queue in a text file  storeMinElementToFile(&queue, "min\_element\_queue.txt");  printf("Minimum element in the queue has been stored in 'min\_element\_queue.txt'.\n");  return 0;  }  Output: |
| **2).. Write a C program to implement a queue using an array. Programs should contain functions for inserting elements into the queue, displaying queue elements, and checking whether the queue is empty or not.**  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the queue  // Structure to represent the queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = -1;  queue->rear = -1;  }  // Function to check if the queue is empty  bool isEmpty(struct Queue \*queue) {  return (queue->front == -1);  }  // Function to check if the queue is full  bool isFull(struct Queue \*queue) {  return ((queue->rear + 1) % MAX\_SIZE == queue->front);  }  // Function to insert an element into the queue  void enqueue(struct Queue \*queue, int value) {  if (isFull(queue)) {  printf("Queue overflow! Cannot enqueue more elements.\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->items[queue->front];  if (queue->front == queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front = (queue->front + 1) % MAX\_SIZE;  }  return dequeuedValue;  }  // Function to display the queue elements  void displayQueue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty.\n");  return;  }  printf("Queue elements: ");  int i = queue->front;  while (i != queue->rear) {  printf("%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", queue->items[i]);  }  // Function to store queue elements in a text file  void storeQueueToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  if (isEmpty(queue)) {  fprintf(file, "Queue is empty.\n");  } else {  fprintf(file, "Queue elements:\n");  int i = queue->front;  while (i != queue->rear) {  fprintf(file, "%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(file, "%d\n", queue->items[i]);  }  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Insert elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  // Display queue elements  displayQueue(&queue);  // Check if the queue is empty  if (isEmpty(&queue)) {  printf("Queue is empty.\n");  } else {  printf("Queue is not empty.\n");  }  // Store queue elements in a text file  storeQueueToFile(&queue, "queue\_elements.txt");  printf("Queue elements have been stored in 'queue\_elements.txt'.\n");  return 0;  }  Output: |

**3)Write a program in C to create and display a Singly Linked List**

**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 end of the linked list

void insertNodeEnd(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 display the linked list

void displayList(struct Node\* head) {

printf("Linked List: ");

while (head != NULL) {

printf("%d -> ", head->data);

head = head->next;

}

printf("NULL\n");

}

// Function to write the linked list data into a text file

void writeToFile(struct Node\* head, const char\* filename) {

FILE\* fp = fopen(filename, "w");

if (fp == NULL) {

printf("Error opening file %s\n", filename);

exit(1);

}

while (head != NULL) {

fprintf(fp, "%d ", head->data);

head = head->next;

}

fclose(fp);

}

int main() {

struct Node\* head = NULL;

// Insert some data into the linked list

insertNodeEnd(&head, 10);

insertNodeEnd(&head, 20);

insertNodeEnd(&head, 30);

insertNodeEnd(&head, 40);

// Display the linked list

displayList(head);

// Write the linked list data into a text file

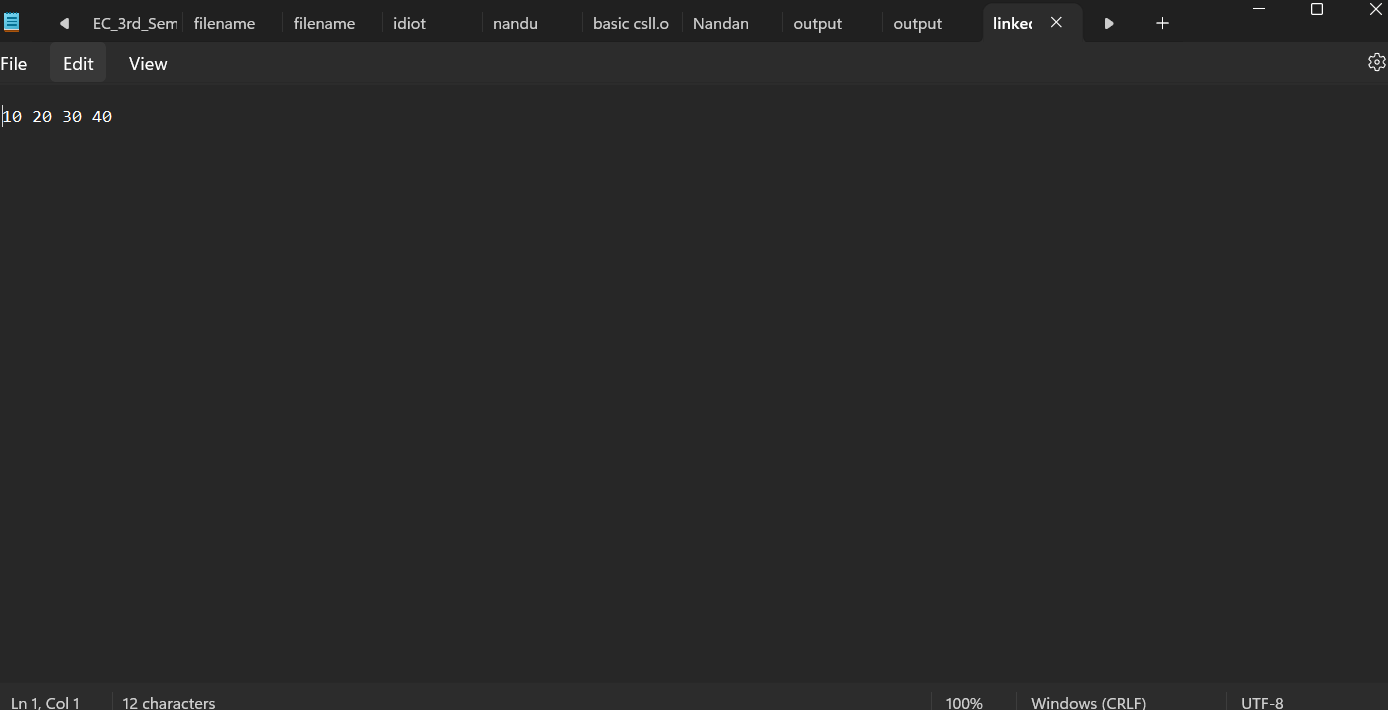
writeToFile(head, "linked\_list\_data.txt");

printf("Data written to file successfully.\n");

return 0;

}

Output:



|  |
| --- |
| **4). Write a program in C to create a singly linked list of n nodes and display it in reverse order.** |
| 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 end of the linked list  void insertNodeEnd(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 display the linked list in reverse order  void displayReverse(struct Node\* head) {  if (head == NULL)  return;  displayReverse(head->next);  printf("%d -> ", head->data);  }  // Function to write the linked list data into a text file  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file %s\n", filename);  exit(1);  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  int main() {  struct Node\* head = NULL;  int n, data;  printf("Enter the number of nodes: ");  scanf("%d", &n);  printf("Enter the data for each node:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &data);  insertNodeEnd(&head, data);  }  // Display the linked list in reverse order  printf("Linked List in reverse order: ");  displayReverse(head);  printf("NULL\n");  // Write the linked list data into a text file  writeToFile(head, "linked\_list\_data\_reverse.txt");  printf("Data written to file successfully.\n");  return 0;  } |
| Output: |
| **5). Write a program in C to create a singly linked list of n nodes and count the number of nodes**. |
| 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 end of the linked list  void insertNodeEnd(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 count the number of nodes in the linked list  int countNodes(struct Node\* head) {  int count = 0;  while (head != NULL) {  count++;  head = head->next;  }  return count;  }  // Function to write the linked list data and node count into a text file  void writeToFile(struct Node\* head, int nodeCount, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file %s\n", filename);  exit(1);  }  fprintf(fp, "Number of nodes: %d\n", nodeCount);  fprintf(fp, "Linked List Data:\n");  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  int main() {  struct Node\* head = NULL;  int n, data;  printf("Enter the number of nodes: ");  scanf("%d", &n);  printf("Enter the data for each node:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &data);  insertNodeEnd(&head, data);  }  // Count the number of nodes  int nodeCount = countNodes(head);  printf("Number of nodes: %d\n", nodeCount);  // Write the linked list data and node count into a text file  writeToFile(head, nodeCount, "linked\_list\_data\_count.txt");  printf("Data written to file successfully.\n");  return 0;  } |
| Output: |
|  |
| **6). Write a program in C to insert a new node at the beginning of a Singly Linked List.** |
| **Code:** |
| #include <stdio.h>  #include <stdlib.h>  struct node  {  int num;  struct node \*nextptr;  }\*stnode;  void createNodeList(int n);  void NodeInsertatBegin(int num);  void displayList();  int main()  {  int n,num;  printf("\n\n Linked List : Insert a new node at the beginning of a Singly Linked List:\n");  printf("------------------------------------------------------------------------------\n");  printf(" Input the number of nodes : ");  scanf("%d", &n);  createNodeList(n);  printf("\n Data entered in the list are : \n");  displayList();  printf("\n Input data to insert at the beginning of the list : ");  scanf("%d", &num);  NodeInsertatBegin(num);  printf("\n Data after inserted in the list are : \n");  displayList();  return 0;  }  void createNodeList(int n)  {  struct node \*fnNode, \*tmp;  int num, i;  stnode = (struct node \*)malloc(sizeof(struct node));  if(stnode == NULL)  {  printf(" Memory can not be allocated.");  }  else  {  printf(" Input data for node 1 : ");  scanf("%d", &num);  stnode-> num = num;  stnode-> nextptr = NULL;  tmp = stnode;  for(i=2; i<=n; i++)  {  fnNode = (struct node \*)malloc(sizeof(struct node));  if(fnNode == NULL)  {  printf(" Memory can not be allocated.");  break;  }  else  {  printf(" Input data for node %d : ", i);  scanf(" %d", &num);  fnNode->num = num;  fnNode->nextptr = NULL;  tmp->nextptr = fnNode;  tmp = tmp->nextptr;  }  }  }  }  void NodeInsertatBegin(int num)  {  struct node \*fnNode;  fnNode = (struct node\*)malloc(sizeof(struct node));  if(fnNode == NULL)  {  printf(" Memory can not be allocated.");  }  else  {  fnNode->num = num;  fnNode->nextptr = stnode;  stnode = fnNode;  }  }  void displayList()  {  struct node \*tmp;  if(stnode == NULL)  {  printf(" No data found in the list.");  }  else  {  tmp = stnode;  while(tmp != NULL)  {  printf(" Data = %d\n", tmp->num);  tmp = tmp->nextptr;  }  }  } |
| Output: |
|  |
| **7). Write a C program to merge two sorted singly linked lists into a single sorted linked list.** |
| 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 end of the linked list  void insertNodeEnd(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 merge two sorted linked lists  struct Node\* mergeSortedLists(struct Node\* list1, struct Node\* list2) {  struct Node\* mergedList = NULL;  // Base cases  if (list1 == NULL)  return list2;  else if (list2 == NULL)  return list1;  // Choose the smaller value and recur  if (list1->data <= list2->data) {  mergedList = list1;  mergedList->next = mergeSortedLists(list1->next, list2);  } else {  mergedList = list2;  mergedList->next = mergeSortedLists(list1, list2->next);  }  return mergedList;  }  // Function to display the linked list  void displayList(struct Node\* head) {  printf("Linked List: ");  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  // Function to write the linked list data into a text file  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file %s\n", filename);  exit(1);  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  int main() {  struct Node\* list1 = NULL;  struct Node\* list2 = NULL;  // Insert some data into the first sorted linked list  insertNodeEnd(&list1, 1);  insertNodeEnd(&list1, 3);  insertNodeEnd(&list1, 5);  insertNodeEnd(&list1, 7);  // Insert some data into the second sorted linked list  insertNodeEnd(&list2, 2);  insertNodeEnd(&list2, 4);  insertNodeEnd(&list2, 6);  insertNodeEnd(&list2, 8);  // Display the original lists  printf("Original Lists:\n");  printf("List 1: ");  displayList(list1);  printf("List 2: ");  displayList(list2);  // Merge the lists  struct Node\* mergedList = mergeSortedLists(list1, list2);  // Display the merged list  printf("\nMerged List:\n");  displayList(mergedList);  // Write the merged list data into a text file  writeToFile(mergedList, "merged\_sorted\_list.txt");  printf("Merged list data written to file successfully.\n");  return 0;  } |
| Output: |
|  |
| **8). Write a C program to check if a singly linked list is a palindrome or not** |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  // Define the structure for a node in the linked list  struct Node {  char data;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(char 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 insertNodeEnd(struct Node\*\* head, char 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 reverse a linked list  struct Node\* reverseList(struct Node\* head) {  struct Node \*prev = NULL, \*current = head, \*next = NULL;  while (current != NULL) {  next = current->next;  current->next = prev;  prev = current;  current = next;  }  return prev;  }  // Function to check if a linked list is a palindrome  bool isPalindrome(struct Node\* head) {  struct Node\* slow = head;  struct Node\* fast = head;  // Find the middle of the linked list  while (fast != NULL && fast->next != NULL) {  slow = slow->next;  fast = fast->next->next;  }  // Reverse the second half of the linked list  struct Node\* secondHalf = reverseList(slow);  // Compare the first half with the reversed second half  while (head != NULL && secondHalf != NULL) {  if (head->data != secondHalf->data)  return false;  head = head->next;  secondHalf = secondHalf->next;  }  return true;  }  // Function to display the linked list  void displayList(struct Node\* head) {  printf("Linked List: ");  while (head != NULL) {  printf("%c -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  // Function to write the result into a text file  void writeResultToFile(bool isPalin, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file %s\n", filename);  exit(1);  }  fprintf(fp, "Is Palindrome: %s", isPalin ? "Yes" : "No");  fclose(fp);  }  int main() {  struct Node\* head = NULL;  char data;  int n;  printf("Enter the number of characters: ");  scanf("%d", &n);  printf("Enter the characters for the linked list:\n");  for (int i = 0; i < n; i++) {  scanf(" %c", &data);  insertNodeEnd(&head, data);  }  // Display the linked list  displayList(head);  // Check if the linked list is a palindrome  bool isPalin = isPalindrome(head);  // Write the result into a text file  writeResultToFile(isPalin, "palindrome\_result.txt");  printf("Result written to file successfully.\n");  return 0;  } |
| **O**utput: |
| **9). Write a C program to delete alternate nodes of a singly linked list.** |
| #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 end of the linked list  void insertNodeEnd(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 delete alternate nodes of the linked list  void deleteAlternateNodes(struct Node\* head) {  struct Node\* temp = head;  while (temp != NULL && temp->next != NULL) {  struct Node\* nextNode = temp->next;  temp->next = nextNode->next;  free(nextNode);  temp = temp->next;  }  }  // Function to display the linked list  void displayList(struct Node\* head) {  printf("Linked List: ");  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  // Function to write the linked list data into a text file  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file %s\n", filename);  exit(1);  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  int main() {  struct Node\* head = NULL;  int data, n;  printf("Enter the number of nodes: ");  scanf("%d", &n);  printf("Enter the data for each node:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &data);  insertNodeEnd(&head, data);  }  // Display the original linked list  printf("Original Linked List:\n");  displayList(head);  // Delete alternate nodes  deleteAlternateNodes(head);  // Display the modified linked list  printf("\nLinked List after deleting alternate nodes:\n");  displayList(head);  // Write the modified linked list data into a text file  writeToFile(head, "modified\_linked\_list.txt");  printf("Modified linked list data written to file successfully.\n");  return 0;  } |
| Output: |
|  |
| **10). Write a program in C to insert a new node at the end of a Singly Linked List.** |
| #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 end of the linked list  void insertNodeEnd(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 display the linked list  void displayList(struct Node\* head) {  printf("Linked List: ");  while (head != NULL) {  printf("%d -> ", head->data);  head = head->next;  }  printf("NULL\n");  }  // Function to write the linked list data into a text file  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file %s\n", filename);  exit(1);  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  int main() {  struct Node\* head = NULL;  int data, n;  printf("Enter the number of nodes: ");  scanf("%d", &n);  printf("Enter the data for each node:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &data);  insertNodeEnd(&head, data);  }  // Display the linked list  displayList(head);  // Write the linked list data into a text file  writeToFile(head, "linked\_list\_data\_end.txt");  printf("Data written to file successfully.\n");  return 0;  } |
| Output: |
|  |
| **11).Write a program in C to search an element in a circular linked list.** |
| #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 end of the circular linked list  void insertNodeEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head; // Circular link  } else {  struct Node\* temp = \*head;  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode;  newNode->next = \*head; // Circular link  }  }  // Function to search for an element in the circular linked list  int searchElement(struct Node\* head, int key) {  struct Node\* current = head;  do {  if (current->data == key) {  return 1; // Element found  }  current = current->next;  } while (current != head);  return 0; // Element not found  }  // Function to display the circular linked list  void displayList(struct Node\* head) {  if (head == NULL) {  printf("Circular Linked List is empty.\n");  return;  }  struct Node\* current = head;  do {  printf("%d -> ", current->data);  current = current->next;  } while (current != head);  printf(" (head)\n");  }  // Function to write the circular linked list data into a text file  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file %s\n", filename);  exit(1);  }  struct Node\* current = head;  if (current == NULL) {  fprintf(fp, "Circular Linked List is empty.\n");  } else {  do {  fprintf(fp, "%d ", current->data);  current = current->next;  } while (current != head);  fprintf(fp, "(head)\n");  }  fclose(fp);  }  int main() {  struct Node\* head = NULL;  int n, data, key;  printf("Enter the number of nodes: ");  scanf("%d", &n);  printf("Enter the data for each node:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &data);  insertNodeEnd(&head, data);  }  // Display the circular linked list  printf("Circular Linked List:\n");  displayList(head);  // Prompt the user to enter the element to search  printf("Enter the element to search: ");  scanf("%d", &key);  // Search for the element in the circular linked list  if (searchElement(head, key)) {  printf("Element %d found in the circular linked list.\n", key);  } else {  printf("Element %d not found in the circular linked list.\n", key);  }  // Write the circular linked list data into a text file  writeToFile(head, "circular\_linked\_list.txt");  printf("Circular linked list data written to file successfully.\n");  return 0;  } |
| Output: |
|  |
| **12).Write a C program to sort a given linked list by bubble sort.** |
| #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 swap data of two nodes  void swap(int \*a, int \*b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  // Function to perform bubble sort on linked list  void bubbleSort(struct Node \*start) {  int swapped, i;  struct Node \*ptr1;  struct Node \*lptr = NULL;  // Checking for empty list  if (start == NULL)  return;  do {  swapped = 0;  ptr1 = start;  while (ptr1->next != lptr) {  if (ptr1->data > ptr1->next->data) {  swap(&(ptr1->data), &(ptr1->next->data));  swapped = 1;  }  ptr1 = ptr1->next;  }  lptr = ptr1;  } while (swapped);  }  // Function to insert a new node at the beginning of the linked list  void push(struct Node\*\* head\_ref, int new\_data) {  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));  new\_node->data = new\_data;  new\_node->next = (\*head\_ref);  (\*head\_ref) = 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");  }  // Function to store the sorted linked list in a text file  void storeListToFile(struct Node \*node, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  while (node != NULL) {  fprintf(file, "%d ", node->data);  node = node->next;  }  fclose(file);  }  int main() {  struct Node \*start = NULL;  int n, data;    printf("Enter the number of elements: ");  scanf("%d", &n);  printf("Enter the elements: ");  for (int i = 0; i < n; i++) {  scanf("%d", &data);  push(&start, data);  }  printf("Linked list before sorting: ");  printList(start);  bubbleSort(start);  printf("Linked list after sorting: ");  printList(start);  // Store the sorted linked list in a text file  storeListToFile(start, "sorted\_linked\_list.txt");  return 0;  }}   |  | | --- | | **13). Write a C program to implement a queue using an array. Programs should contain functions for inserting elements into the queue, displaying queue elements, and checking whether the queue is empty or not.**  **Code:**  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the queue  // Structure to represent the queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = -1;  queue->rear = -1;  }  // Function to check if the queue is empty  bool isEmpty(struct Queue \*queue) {  return (queue->front == -1);  }  // Function to check if the queue is full  bool isFull(struct Queue \*queue) {  return ((queue->rear + 1) % MAX\_SIZE == queue->front);  }  // Function to insert an element into the queue  void enqueue(struct Queue \*queue, int value) {  if (isFull(queue)) {  printf("Queue overflow! Cannot enqueue more elements.\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->items[queue->front];  if (queue->front == queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front = (queue->front + 1) % MAX\_SIZE;  }  return dequeuedValue;  }  // Function to display the queue elements  void displayQueue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty.\n");  return;  }  printf("Queue elements: ");  int i = queue->front;  while (i != queue->rear) {  printf("%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", queue->items[i]);  }  // Function to store queue elements in a text file  void storeQueueToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  if (isEmpty(queue)) {  fprintf(file, "Queue is empty.\n");  } else {  fprintf(file, "Queue elements:\n");  int i = queue->front;  while (i != queue->rear) {  fprintf(file, "%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(file, "%d\n", queue->items[i]);  }  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Insert elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  // Display queue elements  displayQueue(&queue);  // Check if the queue is empty  if (isEmpty(&queue)) {  printf("Queue is empty.\n");  } else {  printf("Queue is not empty.\n");  }  // Store queue elements in a text file  storeQueueToFile(&queue, "queue\_elements.txt");  printf("Queue elements have been stored in 'queue\_elements.txt'.\n");  return 0;  }  Output: | | **14)Write a C program to implement a queue using an array. Create a function that removes an element from the queue**  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the queue  // Structure to represent the queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = -1;  queue->rear = -1;  }  // Function to check if the queue is empty  bool isEmpty(struct Queue \*queue) {  return (queue->front == -1);  }  // Function to check if the queue is full  bool isFull(struct Queue \*queue) {  return ((queue->rear + 1) % MAX\_SIZE == queue->front);  }  // Function to insert an element into the queue  void enqueue(struct Queue \*queue, int value) {  if (isFull(queue)) {  printf("Queue overflow! Cannot enqueue more elements.\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  }  // Function to remove an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->items[queue->front];  if (queue->front == queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front = (queue->front + 1) % MAX\_SIZE;  }  return dequeuedValue;  }  // Function to display the queue elements  void displayQueue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty.\n");  return;  }  printf("Queue elements: ");  int i = queue->front;  while (i != queue->rear) {  printf("%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", queue->items[i]);  }  // Function to store queue elements in a text file  void storeQueueToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  if (isEmpty(queue)) {  fprintf(file, "Queue is empty.\n");  } else {  fprintf(file, "Queue elements:\n");  int i = queue->front;  while (i != queue->rear) {  fprintf(file, "%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(file, "%d\n", queue->items[i]);  }  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Insert elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  // Display queue elements  displayQueue(&queue);  // Remove an element from the queue  int removedElement = dequeue(&queue);  printf("Removed element from the queue: %d\n", removedElement);  // Display queue elements after removal  displayQueue(&queue);  // Store queue elements in a text file  storeQueueToFile(&queue, "queue\_elements.txt");  printf("Queue elements have been stored in 'queue\_elements.txt'.\n");  return 0;  }  Output: | | **15) Write a C program to implement a queue using a linked list. Programs should contain functions for inserting elements into the queue, displaying queue elements, and checking whether the queue is empty or not**  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  // Structure to represent a node in the queue  struct Node {  int data;  struct Node\* next;  };  // Structure to represent the queue  struct Queue {  struct Node\* front;  struct Node\* rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = NULL;  queue->rear = NULL;  }  // Function to check if the queue is empty  bool isEmpty(struct Queue \*queue) {  return (queue->front == NULL);  }  // Function to insert an element into the queue  void enqueue(struct Queue \*queue, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = value;  newNode->next = NULL;  if (isEmpty(queue)) {  queue->front = newNode;  } else {  queue->rear->next = newNode;  }  queue->rear = newNode;  }  // Function to remove an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->front->data;  struct Node\* temp = queue->front;  queue->front = queue->front->next;  free(temp);  if (queue->front == NULL) {  queue->rear = NULL;  }  return dequeuedValue;  }  // Function to display the queue elements  void displayQueue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty.\n");  return;  }  printf("Queue elements: ");  struct Node\* current = queue->front;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  // Function to store queue elements in a text file  void storeQueueToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  if (isEmpty(queue)) {  fprintf(file, "Queue is empty.\n");  } else {  fprintf(file, "Queue elements:\n");  struct Node\* current = queue->front;  while (current != NULL) {  fprintf(file, "%d ", current->data);  current = current->next;  }  fprintf(file, "\n");  }  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Insert elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  // Display queue elements  displayQueue(&queue);  // Remove an element from the queue  int removedElement = dequeue(&queue);  printf("Removed element from the queue: %d\n", removedElement);  // Display queue elements after removal  displayQueue(&queue);  // Store queue elements in a text file  storeQueueToFile(&queue, "queue\_elements.txt");  printf("Queue elements have been stored in 'queue\_elements.txt'.\n");  return 0;  }  Output: |  |  | | --- | | **16)  Write a C program to implement a queue using an array. Create a function that removes an element from the queue**.  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the queue  // Structure to represent the queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = -1;  queue->rear = -1;  }  // Function to check if the queue is empty  bool isEmpty(struct Queue \*queue) {  return (queue->front == -1);  }  // Function to check if the queue is full  bool isFull(struct Queue \*queue) {  return ((queue->rear + 1) % MAX\_SIZE == queue->front);  }  // Function to insert an element into the queue  void enqueue(struct Queue \*queue, int value) {  if (isFull(queue)) {  printf("Queue overflow! Cannot enqueue more elements.\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  }  // Function to remove an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->items[queue->front];  if (queue->front == queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front = (queue->front + 1) % MAX\_SIZE;  }  return dequeuedValue;  }  // Function to display the queue elements  void displayQueue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty.\n");  return;  }  printf("Queue elements: ");  int i = queue->front;  while (i != queue->rear) {  printf("%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", queue->items[i]);  }  // Function to store queue elements in a text file  void storeQueueToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  if (isEmpty(queue)) {  fprintf(file, "Queue is empty.\n");  } else {  fprintf(file, "Queue elements:\n");  int i = queue->front;  while (i != queue->rear) {  fprintf(file, "%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(file, "%d\n", queue->items[i]);  }  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Insert elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  // Display queue elements  displayQueue(&queue);  // Remove an element from the queue  int removedElement = dequeue(&queue);  printf("Removed element from the queue: %d\n", removedElement);  // Display queue elements after removal  displayQueue(&queue);  // Store queue elements in a text file  storeQueueToFile(&queue, "queue\_elements.txt");  printf("Queue elements have been stored in 'queue\_elements.txt'.\n");  return 0;  }  Output: | | **17). Write a C program to count the number of elements in a queue.**  Code:  #include <stdio.h>  #include <stdlib.h>  // Structure to represent a node in the queue  struct Node {  int data;  struct Node\* next;  };  // Structure to represent the queue  struct Queue {  struct Node\* front;  struct Node\* rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = NULL;  queue->rear = NULL;  }  // Function to check if the queue is empty  int isEmpty(struct Queue \*queue) {  return (queue->front == NULL);  }  // Function to enqueue an element into the queue  void enqueue(struct Queue \*queue, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = value;  newNode->next = NULL;  if (isEmpty(queue)) {  queue->front = newNode;  } else {  queue->rear->next = newNode;  }  queue->rear = newNode;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->front->data;  struct Node\* temp = queue->front;  queue->front = queue->front->next;  if (queue->front == NULL) {  queue->rear = NULL;  }  free(temp);  return dequeuedValue;  }  // Function to count the number of elements in the queue  int countElements(struct Queue \*queue) {  int count = 0;  struct Node\* current = queue->front;  while (current != NULL) {  count++;  current = current->next;  }  return count;  }  // Function to store the number of elements in a text file  void storeCountToFile(int count, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  fprintf(file, "Number of elements in the queue: %d\n", count);  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Enqueue some elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  enqueue(&queue, 50);  // Count the number of elements in the queue  int count = countElements(&queue);  // Store the count of elements in a text file  storeCountToFile(count, "count\_elements\_queue.txt");  printf("Number of elements in the queue has been stored in 'count\_elements\_queue.txt'.\n");  return 0;  }  Output: | | **18). Write a C program to reverse the elements of a queue.**  Code:  #include <stdio.h>  #include <stdlib.h>  // Structure to represent a node in the queue  struct Node {  int data;  struct Node\* next;  };  // Structure to represent the queue  struct Queue {  struct Node\* front;  struct Node\* rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = NULL;  queue->rear = NULL;  }  // Function to check if the queue is empty  int isEmpty(struct Queue \*queue) {  return (queue->front == NULL);  }  // Function to enqueue an element into the queue  void enqueue(struct Queue \*queue, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = value;  newNode->next = NULL;  if (isEmpty(queue)) {  queue->front = newNode;  } else {  queue->rear->next = newNode;  }  queue->rear = newNode;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->front->data;  struct Node\* temp = queue->front;  queue->front = queue->front->next;  if (queue->front == NULL) {  queue->rear = NULL;  }  free(temp);  return dequeuedValue;  }  // Function to reverse the elements of the queue  void reverseQueue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty. Nothing to reverse.\n");  return;  }  struct Node\* stack = NULL; // Stack to hold elements temporarily  // Push all elements from the queue onto the stack  while (!isEmpty(queue)) {  int value = dequeue(queue);  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->next = stack;  stack = newNode;  }  // Pop all elements from the stack and enqueue them back into the queue  while (stack != NULL) {  enqueue(queue, stack->data);  struct Node\* temp = stack;  stack = stack->next;  free(temp);  }  }  // Function to store the elements of the queue in a text file  void storeQueueToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write queue contents to the file  struct Node\* current = queue->front;  while (current != NULL) {  fprintf(file, "%d\n", current->data);  current = current->next;  }  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Enqueue some elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  enqueue(&queue, 50);  // Reverse the elements of the queue  reverseQueue(&queue);  // Store the reversed elements of the queue in a text file  storeQueueToFile(&queue, "reversed\_queue.txt");  printf("Reversed elements of the queue have been stored in 'reversed\_queue.txt'.\n");  return 0;  }  Output: | | **19). Write a C program to calculate the sum of the elements in a queue**  Code: #include <stdio.h>  #include <stdlib.h>  // Structure to represent a node in the queue  struct Node {  int data;  struct Node\* next;  };  // Structure to represent the queue  struct Queue {  struct Node\* front;  struct Node\* rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = NULL;  queue->rear = NULL;  }  // Function to check if the queue is empty  int isEmpty(struct Queue \*queue) {  return (queue->front == NULL);  }  // Function to enqueue an element into the queue  void enqueue(struct Queue \*queue, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = value;  newNode->next = NULL;  if (isEmpty(queue)) {  queue->front = newNode;  } else {  queue->rear->next = newNode;  }  queue->rear = newNode;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->front->data;  struct Node\* temp = queue->front;  queue->front = queue->front->next;  if (queue->front == NULL) {  queue->rear = NULL;  }  free(temp);  return dequeuedValue;  }  // Function to calculate the sum of the elements in the queue  int calculateSum(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty. Sum is 0.\n");  return 0;  }  int sum = 0;  struct Node\* current = queue->front;  while (current != NULL) {  sum += current->data;  current = current->next;  }  return sum;  }  // Function to store the sum of elements in a text file  void storeSumToFile(int sum, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  fprintf(file, "Sum of elements in the queue: %d\n", sum);  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Enqueue some elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  enqueue(&queue, 50);  // Calculate the sum of elements in the queue  int sum = calculateSum(&queue);  // Store the sum of elements in a text file  storeSumToFile(sum, "sum\_of\_elements\_queue.txt");  printf("Sum of elements in the queue has been stored in 'sum\_of\_elements\_queue.txt'.\n");  return 0;  }  Output: | | **20) Write a C program to compute the average value of the elements in a queue**  **Code:**  #include <stdio.h>  #include <stdlib.h>  // Structure to represent a node in the queue  struct Node {  int data;  struct Node\* next;  };  // Structure to represent the queue  struct Queue {  struct Node\* front;  struct Node\* rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = NULL;  queue->rear = NULL;  }  // Function to check if the queue is empty  int isEmpty(struct Queue \*queue) {  return (queue->front == NULL);  }  // Function to enqueue an element into the queue  void enqueue(struct Queue \*queue, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = value;  newNode->next = NULL;  if (isEmpty(queue)) {  queue->front = newNode;  } else {  queue->rear->next = newNode;  }  queue->rear = newNode;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->front->data;  struct Node\* temp = queue->front;  queue->front = queue->front->next;  if (queue->front == NULL) {  queue->rear = NULL;  }  free(temp);  return dequeuedValue;  }  // Function to compute the average value of elements in the queue  float computeAverage(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty. Average value cannot be computed.\n");  exit(EXIT\_FAILURE);  }  int sum = 0;  int count = 0;  struct Node\* current = queue->front;  while (current != NULL) {  sum += current->data;  count++;  current = current->next;  }  return (float)sum / count;  }  // Function to store the average value in a text file  void storeAverageToFile(float average, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  fprintf(file, "Average value of elements in the queue: %.2f\n", average);  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Enqueue some elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 20);  enqueue(&queue, 30);  enqueue(&queue, 40);  enqueue(&queue, 50);  // Compute the average value of elements in the queue  float average = computeAverage(&queue);  // Store the average value in a text file  storeAverageToFile(average, "average\_queue.txt");  printf("Average value of elements in the queue has been stored in 'average\_queue.txt'.\n");  return 0;  }  Output: | |  | | **21). Write a C program to find the maximum element in a queue.**  Code:  #include <stdio.h>  #include <stdlib.h>  // Structure to represent a node in the queue  struct Node {  int data;  struct Node\* next;  };  // Structure to represent the queue  struct Queue {  struct Node\* front;  struct Node\* rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = NULL;  queue->rear = NULL;  }  // Function to check if the queue is empty  int isEmpty(struct Queue \*queue) {  return (queue->front == NULL);  }  // Function to enqueue an element into the queue  void enqueue(struct Queue \*queue, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = value;  newNode->next = NULL;  if (isEmpty(queue)) {  queue->front = newNode;  } else {  queue->rear->next = newNode;  }  queue->rear = newNode;  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedValue = queue->front->data;  struct Node\* temp = queue->front;  queue->front = queue->front->next;  if (queue->front == NULL) {  queue->rear = NULL;  }  free(temp);  return dequeuedValue;  }  // Function to find the maximum element in the queue  int findMax(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue is empty. No maximum element found.\n");  exit(EXIT\_FAILURE);  }  int max = queue->front->data;  struct Node\* current = queue->front->next;  while (current != NULL) {  if (current->data > max) {  max = current->data;  }  current = current->next;  }  return max;  }  // Function to store the maximum element in a text file  void storeMaxToFile(int max, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  fprintf(file, "Maximum element in the queue: %d\n", max);  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Enqueue some elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 30);  enqueue(&queue, 50);  enqueue(&queue, 20);  enqueue(&queue, 40);  // Find the maximum element in the queue  int max = findMax(&queue);  // Store the maximum element in a text file  storeMaxToFile(max, "max\_element\_queue.txt");  printf("Maximum element in the queue has been stored in 'max\_element\_queue.txt'.\n");  return 0;  }  Output: | | **22) Write a C program to find the minimum element in a queue.**  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the queue  // Structure to represent the queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  int size;  int minElement; // Variable to store the minimum element  };  // Function to initialize the queue  void initializeQueue(struct Queue \*queue) {  queue->front = -1;  queue->rear = -1;  queue->size = 0;  queue->minElement = INT\_MAX; // Initialize minElement with maximum possible integer value  }  // Function to check if the queue is empty  bool isEmpty(struct Queue \*queue) {  return (queue->size == 0);  }  // Function to check if the queue is full  bool isFull(struct Queue \*queue) {  return (queue->size == MAX\_SIZE);  }  // Function to enqueue an element into the queue  void enqueue(struct Queue \*queue, int value) {  if (isFull(queue)) {  printf("Queue overflow! Cannot enqueue more elements.\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  queue->size++;  // Update minElement if the enqueued value is smaller  if (value < queue->minElement) {  queue->minElement = value;  }  }  // Function to dequeue an element from the queue  int dequeue(struct Queue \*queue) {  if (isEmpty(queue)) {  printf("Queue underflow! Cannot dequeue from an empty queue.\n");  exit(EXIT\_FAILURE);  }  int dequeuedItem = queue->items[queue->front];  queue->front = (queue->front + 1) % MAX\_SIZE;  queue->size--;  // Recalculate minElement if the dequeued element was the minimum  if (dequeuedItem == queue->minElement) {  queue->minElement = INT\_MAX;  for (int i = queue->front; i <= queue->rear; i++) {  if (queue->items[i] < queue->minElement) {  queue->minElement = queue->items[i];  }  }  }  return dequeuedItem;  }  // Function to store the minimum element in the queue in a text file  void storeMinElementToFile(struct Queue \*queue, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write the minimum element to the file  fprintf(file, "Minimum element in the queue: %d\n", queue->minElement);  fclose(file);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Enqueue some elements into the queue  enqueue(&queue, 10);  enqueue(&queue, 5);  enqueue(&queue, 20);  enqueue(&queue, 3);  enqueue(&queue, 15);  // Dequeue some elements from the queue  dequeue(&queue);  dequeue(&queue);  // Store the minimum element in the queue in a text file  storeMinElementToFile(&queue, "min\_element\_queue.txt");  printf("Minimum element in the queue has been stored in 'min\_element\_queue.txt'.\n");  return 0;  }  Output:     |  | | --- | | **23). Write a program in C to find the maximum value in a doubly linked list.** |  |  | | --- | | #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 swap data of two nodes  void swap(int \*a, int \*b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  // Function to perform bubble sort on linked list  void bubbleSort(struct Node \*start) {  int swapped, i;  struct Node \*ptr1;  struct Node \*lptr = NULL;  // Checking for empty list  if (start == NULL)  return;  do {  swapped = 0;  ptr1 = start;  while (ptr1->next != lptr) {  if (ptr1->data > ptr1->next->data) {  swap(&(ptr1->data), &(ptr1->next->data));  swapped = 1;  }  ptr1 = ptr1->next;  }  lptr = ptr1;  } while (swapped);  }  // Function to insert a new node at the beginning of the linked list  void push(struct Node\*\* head\_ref, int new\_data) {  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));  new\_node->data = new\_data;  new\_node->next = (\*head\_ref);  (\*head\_ref) = 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");  }  // Function to store the sorted linked list in a text file  void storeListToFile(struct Node \*node, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  while (node != NULL) {  fprintf(file, "%d ", node->data);  node = node->next;  }  fclose(file);  }  int main() {  struct Node \*start = NULL;  int n, data;    printf("Enter the number of elements: ");  scanf("%d", &n);  printf("Enter the elements: ");  for (int i = 0; i < n; i++) {  scanf("%d", &data);  push(&start, data);  }  printf("Linked list before sorting: ");  printList(start);  bubbleSort(start);  printf("Linked list after sorting: ");  printList(start);  // Store the sorted linked list in a text file  storeListToFile(start, "sorted\_linked\_list.txt");  return 0;  }  Output: | | **24).Write a C program to implement a stack using an array with push and pop operations.** | | #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100 // Maximum size of the stack  // Structure to represent the stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack \*stack) {  stack->top = -1;  }  // Function to check if the stack is full  int isFull(struct Stack \*stack) {  return (stack->top == MAX\_SIZE - 1);  }  // Function to check if the stack is empty  int isEmpty(struct Stack \*stack) {  return (stack->top == -1);  }  // Function to push an element onto the stack  void push(struct Stack \*stack, int value) {  if (isFull(stack)) {  printf("Stack overflow! Cannot push more elements.\n");  return;  }  stack->top++;  stack->items[stack->top] = value;  }  // Function to pop an element from the stack  int pop(struct Stack \*stack) {  if (isEmpty(stack)) {  printf("Stack underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return stack->items[stack->top--];  }  // Function to store the stack contents in a text file  void storeStackToFile(struct Stack \*stack, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write stack contents to the file  for (int i = stack->top; i >= 0; i--) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  int main() {  struct Stack stack;  initializeStack(&stack);  // Push some elements onto the stack  push(&stack, 10);  push(&stack, 20);  push(&stack, 30);  push(&stack, 40);  push(&stack, 50);  // Pop an element from the stack  printf("Popped element: %d\n", pop(&stack));  // Store the stack contents in a text file  storeStackToFile(&stack, "stack\_contents.txt");  return 0;  }  Output: | | **25) Write a C program to implement a stack using a singly linked list.** | | #include <stdio.h>  #include <stdlib.h>  // Structure to represent a node in the linked list  struct Node {  int data;  struct Node\* next;  };  // Structure to represent the stack  struct Stack {  struct Node\* top;  };  // Function to initialize the stack  void initializeStack(struct Stack \*stack) {  stack->top = NULL;  }  // Function to check if the stack is empty  int isEmpty(struct Stack \*stack) {  return (stack->top == NULL);  }  // Function to push an element onto the stack  void push(struct Stack \*stack, int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = value;  newNode->next = stack->top;  stack->top = newNode;  }  // Function to pop an element from the stack  int pop(struct Stack \*stack) {  if (isEmpty(stack)) {  printf("Stack underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  int poppedValue = stack->top->data;  struct Node\* temp = stack->top;  stack->top = stack->top->next;  free(temp);  return poppedValue;  }  // Function to store the stack contents in a text file  void storeStackToFile(struct Stack \*stack, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write stack contents to the file  struct Node\* current = stack->top;  while (current != NULL) {  fprintf(file, "%d\n", current->data);  current = current->next;  }  fclose(file);  }  int main() {  struct Stack stack;  initializeStack(&stack);  // Push some elements onto the stack  push(&stack, 10);  push(&stack, 20);  push(&stack, 30);  push(&stack, 40);  push(&stack, 50);  // Pop an element from the stack  printf("Popped element: %d\n", pop(&stack));  // Store the stack contents in a text file  storeStackToFile(&stack, "stack\_contents.txt");  return 0;  }  Output: | | **.26)  Write a C program that accepts a string and reverse it using a stack.** | | #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MAX\_SIZE 100 // Maximum size of the stack  // Structure to represent the stack  struct Stack {  char items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack \*stack) {  stack->top = -1;  }  // Function to check if the stack is empty  int isEmpty(struct Stack \*stack) {  return (stack->top == -1);  }  // Function to push a character onto the stack  void push(struct Stack \*stack, char value) {  if (stack->top == MAX\_SIZE - 1) {  printf("Stack overflow! Cannot push more elements.\n");  return;  }  stack->top++;  stack->items[stack->top] = value;  }  // Function to pop a character from the stack  char pop(struct Stack \*stack) {  if (isEmpty(stack)) {  printf("Stack underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return stack->items[stack->top--];  }  // Function to reverse a string using a stack  void reverseString(char \*str) {  struct Stack stack;  initializeStack(&stack);  // Push each character of the string onto the stack  for (int i = 0; str[i] != '\0'; i++) {  push(&stack, str[i]);  }  // Pop characters from the stack to reverse the string  int i = 0;  while (!isEmpty(&stack)) {  str[i++] = pop(&stack);  }  }  int main() {  char input[MAX\_SIZE];  printf("Enter a string: ");  fgets(input, MAX\_SIZE, stdin);  // Remove the newline character if present  if (input[strlen(input) - 1] == '\n') {  input[strlen(input) - 1] = '\0';  }  reverseString(input);  printf("Reversed string: %s\n", input);  // Store the reversed string in a text file  FILE \*file = fopen("reversed\_string.txt", "w");  if (file == NULL) {  printf("Error opening file.\n");  return 1;  }  fprintf(file, "%s", input);  fclose(file);  printf("Reversed string has been stored in 'reversed\_string.txt'.\n");  return 0;  }  Output: | | **27). Write a C program to implement two stacks in a single array and performs push and pop operations for both stacks.**  Code:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define STACK1\_START 0 // Starting index of stack 1  #define STACK2\_START (MAX\_SIZE / 2) // Starting index of stack 2  // Structure to represent the two stacks  struct TwoStacks {  int items[MAX\_SIZE];  int top1;  int top2;  };  // Function to initialize the two stacks  void initializeTwoStacks(struct TwoStacks \*twoStacks) {  twoStacks->top1 = STACK1\_START - 1;  twoStacks->top2 = STACK2\_START - 1;  }  // Function to check if stack 1 is full  int isFullStack1(struct TwoStacks \*twoStacks) {  return (twoStacks->top1 == STACK2\_START - 1);  }  // Function to check if stack 2 is full  int isFullStack2(struct TwoStacks \*twoStacks) {  return (twoStacks->top2 == MAX\_SIZE - 1);  }  // Function to check if stack 1 is empty  int isEmptyStack1(struct TwoStacks \*twoStacks) {  return (twoStacks->top1 == STACK1\_START - 1);  }  // Function to check if stack 2 is empty  int isEmptyStack2(struct TwoStacks \*twoStacks) {  return (twoStacks->top2 == STACK2\_START - 1);  }  // Function to push an element onto stack 1  void pushStack1(struct TwoStacks \*twoStacks, int value) {  if (isFullStack1(twoStacks)) {  printf("Stack 1 overflow! Cannot push more elements.\n");  return;  }  twoStacks->top1++;  twoStacks->items[twoStacks->top1] = value;  }  // Function to push an element onto stack 2  void pushStack2(struct TwoStacks \*twoStacks, int value) {  if (isFullStack2(twoStacks)) {  printf("Stack 2 overflow! Cannot push more elements.\n");  return;  }  twoStacks->top2++;  twoStacks->items[twoStacks->top2] = value;  }  // Function to pop an element from stack 1  int popStack1(struct TwoStacks \*twoStacks) {  if (isEmptyStack1(twoStacks)) {  printf("Stack 1 underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return twoStacks->items[twoStacks->top1--];  }  // Function to pop an element from stack 2  int popStack2(struct TwoStacks \*twoStacks) {  if (isEmptyStack2(twoStacks)) {  printf("Stack 2 underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return twoStacks->items[twoStacks->top2--];  }  // Function to store the information about both stacks in a text file  void storeTwoStacksToFile(struct TwoStacks \*twoStacks, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write stack 1 contents to the file  fprintf(file, "Stack 1:\n");  for (int i = STACK1\_START; i <= twoStacks->top1; i++) {  fprintf(file, "%d\n", twoStacks->items[i]);  }  // Write stack 2 contents to the file  fprintf(file, "\nStack 2:\n");  for (int i = STACK2\_START; i <= twoStacks->top2; i++) {  fprintf(file, "%d\n", twoStacks->items[i]);  }  fclose(file);  }  int main() {  struct TwoStacks twoStacks;  initializeTwoStacks(&twoStacks);  // Push some elements onto stack 1  pushStack1(&twoStacks, 10);  pushStack1(&twoStacks, 20);  pushStack1(&twoStacks, 30);  // Push some elements onto stack 2  pushStack2(&twoStacks, 40);  pushStack2(&twoStacks, 50);  pushStack2(&twoStacks, 60);  // Pop elements from stack 1  printf("Popped element from Stack 1: %d\n", popStack1(&twoStacks));  printf("Popped element from Stack 1: %d\n", popStack1(&twoStacks));  // Pop elements from stack 2  printf("Popped element from Stack 2: %d\n", popStack2(&twoStacks));  // Store the information about both stacks in a text file  storeTwoStacksToFile(&twoStacks, "two\_stacks\_contents.txt");  printf("Information about both stacks has been stored in 'two\_stacks\_contents.txt'.\n");  return 0;  }  Output: |  |  | | --- | | **28).  Write a C program to sort a given stack using another stack.**  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the stack  // Structure to represent the stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack \*stack) {  stack->top = -1;  }  // Function to check if the stack is empty  bool isEmpty(struct Stack \*stack) {  return (stack->top == -1);  }  // Function to check if the stack is full  bool isFull(struct Stack \*stack) {  return (stack->top == MAX\_SIZE - 1);  }  // Function to push an element onto the stack  void push(struct Stack \*stack, int value) {  if (isFull(stack)) {  printf("Stack overflow! Cannot push more elements.\n");  return;  }  stack->top++;  stack->items[stack->top] = value;  }  // Function to pop an element from the stack  int pop(struct Stack \*stack) {  if (isEmpty(stack)) {  printf("Stack underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return stack->items[stack->top--];  }  // Function to peek the top element of the stack  int peek(struct Stack \*stack) {  if (isEmpty(stack)) {  printf("Stack underflow! Cannot peek from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return stack->items[stack->top];  }  // Function to sort a stack using another stack  void sortStack(struct Stack \*inputStack) {  struct Stack tempStack;  initializeStack(&tempStack);  while (!isEmpty(inputStack)) {  int temp = pop(inputStack);  while (!isEmpty(&tempStack) && peek(&tempStack) > temp) {  push(inputStack, pop(&tempStack));  }  push(&tempStack, temp);  }  // Copy sorted elements from tempStack to inputStack  while (!isEmpty(&tempStack)) {  push(inputStack, pop(&tempStack));  }  }  // Function to store the stack contents in a text file  void storeStackToFile(struct Stack \*stack, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write stack contents to the file  for (int i = stack->top; i >= 0; i--) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  int main() {  struct Stack inputStack;  initializeStack(&inputStack);  // Push some elements onto the stack  push(&inputStack, 30);  push(&inputStack, -5);  push(&inputStack, 18);  push(&inputStack, 14);  push(&inputStack, -3);  // Sort the stack  sortStack(&inputStack);  // Display the sorted stack  printf("Sorted stack:\n");  while (!isEmpty(&inputStack)) {  printf("%d\n", pop(&inputStack));  }  // Store the sorted stack in a text file  storeStackToFile(&inputStack, "sorted\_stack.txt");  printf("Sorted stack has been stored in 'sorted\_stack.txt'.\n");  return 0;  }  Output: | | **29). Write a C program that checks whether a string of parentheses is balanced or not using stack.**  Code: #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <string.h>  #define MAX\_SIZE 100 // Maximum size of the stack  // Structure to represent the stack  struct Stack {  char items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack \*stack) {  stack->top = -1;  }  // Function to check if the stack is empty  bool isEmpty(struct Stack \*stack) {  return (stack->top == -1);  }  // Function to check if the stack is full  bool isFull(struct Stack \*stack) {  return (stack->top == MAX\_SIZE - 1);  }  // Function to push a character onto the stack  void push(struct Stack \*stack, char value) {  if (isFull(stack)) {  printf("Stack overflow! Cannot push more elements.\n");  return;  }  stack->top++;  stack->items[stack->top] = value;  }  // Function to pop a character from the stack  char pop(struct Stack \*stack) {  if (isEmpty(stack)) {  printf("Stack underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return stack->items[stack->top--];  }  // Function to check if a string of parentheses is balanced  bool isBalanced(char \*str) {  struct Stack stack;  initializeStack(&stack);  // Traverse the string  for (int i = 0; str[i] != '\0'; i++) {  if (str[i] == '(') {  push(&stack, '(');  } else if (str[i] == ')') {  // If ')' is encountered and the stack is empty, parentheses are not balanced  if (isEmpty(&stack)) {  return false;  }  // Pop '(' from the stack for every ')' encountered  pop(&stack);  }  }  // If the stack is empty after traversal, parentheses are balanced  return isEmpty(&stack);  }  // Function to store the result in a text file  void storeResultToFile(bool result, const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  if (result) {  fprintf(file, "The string of parentheses is balanced.");  } else {  fprintf(file, "The string of parentheses is not balanced.");  }  fclose(file);  }  int main() {  char input[MAX\_SIZE];  printf("Enter a string of parentheses: ");  fgets(input, MAX\_SIZE, stdin);  // Remove the newline character if present  if (input[strlen(input) - 1] == '\n') {  input[strlen(input) - 1] = '\0';  }  // Check if the string of parentheses is balanced  bool result = isBalanced(input);  // Display the result  if (result) {  printf("The string of parentheses is balanced.\n");  } else {  printf("The string of parentheses is not balanced.\n");  }  // Store the result in a text file  storeResultToFile(result, "parentheses\_balance\_result.txt");  printf("Result has been stored in 'parentheses\_balance\_result.txt'.\n");  return 0;  }  Output: | | **30). Write a C program to find the next greater element for each element in an array using a stack. Return -1 if there is no next-larger element.**  Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100 // Maximum size of the stack  // Structure to represent the stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack \*stack) {  stack->top = -1;  }  // Function to check if the stack is empty  bool isEmpty(struct Stack \*stack) {  return (stack->top == -1);  }  // Function to check if the stack is full  bool isFull(struct Stack \*stack) {  return (stack->top == MAX\_SIZE - 1);  }  // Function to push an element onto the stack  void push(struct Stack \*stack, int value) {  if (isFull(stack)) {  printf("Stack overflow! Cannot push more elements.\n");  return;  }  stack->top++;  stack->items[stack->top] = value;  }  // Function to pop an element from the stack  int pop(struct Stack \*stack) {  if (isEmpty(stack)) {  printf("Stack underflow! Cannot pop from an empty stack.\n");  exit(EXIT\_FAILURE);  }  return stack->items[stack->top--];  }  // Function to find the next greater element for each element in an array  void findNextGreaterElement(int arr[], int n, int nextGreater[]) {  struct Stack stack;  initializeStack(&stack);  // Traverse the array from right to left  for (int i = n - 1; i >= 0; i--) {  // Pop elements from the stack that are smaller than or equal to arr[i]  while (!isEmpty(&stack) && stack.items[stack.top] <= arr[i]) {  pop(&stack);  }  // If stack is empty, there is no next greater element  if (isEmpty(&stack)) {  nextGreater[i] = -1;  } else {  // The next greater element is at the top of the stack  nextGreater[i] = stack.items[stack.top];  }  // Push arr[i] onto the stack  push(&stack, arr[i]);  }  }  // Function to store the next greater elements in a text file  void storeNextGreaterToFile(int arr[], int n, int nextGreater[], const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file.\n");  return;  }  // Write array elements and their next greater elements to the file  for (int i = 0; i < n; i++) {  fprintf(file, "%d -> ", arr[i]);  if (nextGreater[i] != -1) {  fprintf(file, "%d\n", nextGreater[i]);  } else {  fprintf(file, "-1\n");  }  }  fclose(file);  }  int main() {  int arr[] = {4, 6, 3, 2, 8, 1, 7};  int n = sizeof(arr) / sizeof(arr[0]);  int nextGreater[n];  // Find the next greater element for each element in the array  findNextGreaterElement(arr, n, nextGreater);  // Store the next greater elements in a text file  storeNextGreaterToFile(arr, n, nextGreater, "next\_greater\_elements.txt");  printf("Next greater elements have been stored in 'next\_greater\_elements.txt'.\n");  return 0;  }  Output: | | |