|  |  |
| --- | --- |
| **Name of the student** | **ANANYA BADAGI** |
| **Roll No** | **247** |
| **USN** | **01FE22BEC077** |
| **Div** | **B** |

**Part- A**

|  |
| --- |
| **Variables and Data Types** |
| 1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate. |
| Ans:  Data Type:  A data type in C programming specifies the type of data that a variable can hold.  It defines the size and format of the data that can be stored in memory.  C provides several built-in data types such as int, float, char, etc.  Data types specify how the data should be interpreted and how it should behave during operations.  Examples:  int: Represents integer numbers.  float: Represents floating-point numbers.  char: Represents single characters.  int age = 25;  float height = 5.8;  char grade = 'A';  Variable:  A variable in C programming is a named memory location used to store data.  Variables must be declared with a specific data type before they can be used.  They hold values that can be changed during program execution.  Variables are used to manipulate and process data in a program.  Examples:  age: Stores an integer value representing a person's age.  height: Stores a floating-point value representing a person's height.  grade: Stores a character representing a student's grade.  int age;  float height;  char grade;  age = 25;  height = 5.8;  grade = 'A'; |
| 2. Explain the concept of data types in C programming. Discuss the different types of data types available in C ? |
| In C programming, data types are used to define the type of data that a variable can hold. They determine the size and format of the data stored in memory, as well as the operations that can be performed on that data. Data types help ensure that operations are performed correctly and efficiently.  Basic Data Types:  int: Represents integer numbers. Typically, it occupies 4 bytes of memory on most systems.  float: Represents single-precision floating-point numbers. It typically occupies 4 bytes of memory.  double: Represents double-precision floating-point numbers. It typically occupies 8 bytes of memory.  char: Represents a single character. It typically occupies 1 byte of memory.  void: Represents the absence of type. It is commonly used as a return type for functions that do not return a value or as a pointer type.  Derived Data Types:  Array: Represents a collection of elements of the same data type. Arrays have a fixed size and are accessed using an index.  Pointer: Represents a memory address. Pointers can point to variables, arrays, or functions.  Structure: Represents a collection of variables of different data types grouped together under a single name.  Union: Similar to a structure, but all members share the same memory location. Only one member of a union can be accessed at a time.  Enumeration: Represents a set of named integer constants. Enumerations provide a way to define symbolic names for integral values.  User-Defined Data Types:  C also allows programmers to define their own data types using typedef keyword. This feature allows for the creation of custom data types with meaningful names, which can improve code readability and maintainability. |
|  |
|  |

|  |
| --- |
| 1. How are variables declared and initialized in C programming? Provide examples of variable declarations with different data types. 2. In C programming, variables are declared and optionally initialized using the following syntax:   data\_type variable\_name; // Declaration  data\_type variable\_name = initial\_value; // Declaration with initialization  Example:  #include <stdio.h>  int main() {  Declaration of variables with different data types  int integerVariable; Declaration of an integer variable  float floatVariable; Declaration of a float variable  char characterVariable; Declaration of a character variable  double doubleVariable; Declaration of a double variable  Initialization of variables  integerVariable = 42; Initialization of integerVariable with value 42  floatVariable = 3.14; Initialization of floatVariable with value 3.14  characterVariable = 'A'; Initialization of characterVariable with character 'A'  doubleVariable = 123.456; Initialization of doubleVariable with value 123.456  Printing values of variables  printf("Integer Variable: %d\n", integerVariable);  printf("Float Variable: %f\n", floatVariable);  printf("Character Variable: %c\n", characterVariable);  printf("Double Variable: %lf\n", doubleVariable);  return 0;  } |
| 1. Discuss the scope and lifetime of variables in C programming. What are global and local variables? 2. In C programming, the scope and lifetime of variables are fundamental concepts that determine where variables can be accessed within a program and how long they persist in memory. Additionally, global and local variables are two types of variables with distinct scopes and lifetimes.  | **Aspect** | **Global Variables** | **Local Variables** | | --- | --- | --- | | Declaration | Declared outside of any function or block. | Declared within a function or block. | | Scope | Accessible from any part of the program. | Accessible only within the block or function where declared. | | Access | Can be accessed and modified by any function. | Accessible only within the block or function of declaration. | | Memory Allocation | Allocated when the program starts and deallocated when the program terminates. | Allocated when the block or function is entered and deallocated when exited. | | Lifetime | Remains in memory throughout the entire execution of the program. | Limited to the execution of the block or function in which they are declared. | | Visibility | Visible to all functions in the program. | Visible only within the function or block of declaration. | | Initialization | Automatically initialized to zero if not explicitly initialized. | Not automatically initialized; must be explicitly initialized before use. | |
| 1. Explain the concept of type casting in C programming. When is type casting necessary, and how is it performed? 2. Type casting in C programming refers to the process of converting a value from one data type to another. This is necessary when you want to perform operations that involve different data types, such as assigning a value of one type to a variable of another type, or performing arithmetic operations between values of different types.   Type casting is necessary in the following situations:  --->Implicit Type Conversion:When you assign a value of one data type to a variable of another data type, C automatically converts the value to the type of the variable. This is known as implicit type conversion.However, in some cases, you may need to explicitly specify the type conversion to ensure that the conversion is done as desired.  --->Arithmetic Operations:When performing arithmetic operations involving operands of different data types, C may promote the operands to a common type before performing the operation.Explicit type casting can be used to force the operands to a specific type to control the result of the operation.  EXAMPLE :  float result;  int num1 = 10;  int num2 = 3;  result = (float) num1 / num2; // Explicit type casting of num1 to float |
| Operators: |
| 1. Describe the purpose and usage of the ternary conditional operator (?:) in C programming. Provide an example demonstrating its usage. 2. The ternary conditional operator ?: in C programming is a shorthand way of writing an if-else statement. It allows you to evaluate a condition and return one of two expressions based on the result of the condition. It has the following syntax:   condition ? expression1 : expression2  -->If the condition evaluates to true (non-zero), the value of expression1 is returned.  -->If the condition evaluates to false (zero), the value of expression2 is returned.  #include <stdio.h>  int main() {  int num = 10;  char\* result;  Using ternary conditional operator to assign result  result = (num % 2 == 0) ? "Even" : "Odd";  Printing result  printf("The number is %s.\n", result);  return 0;  } |
| 1. Discuss the bitwise operators available in C programming. Explain their usage with suitable examples. 2. In C programming, bitwise operators are used to perform operations at the bit level. These operators manipulate individual bits of operands, which are typically integers. Bitwise operators are particularly useful in low-level programming, such as device drivers, embedded systems, and cryptography. There are six bitwise operators in C:   BITWISE (AND):Performs a bitwise AND operation between the corresponding bits of two operands.If both bits are 1, the result is 1. Otherwise, the result is 0.  EXAMPLE :  unsigned int a = 5; // 0101 in binary  unsigned int b = 3; // 0011 in binary  unsigned int result = a & b; // Result: 0001 (1 in decimal)  Bitwise OR (|):Performs a bitwise OR operation between the corresponding bits of two operands.If either bit is 1, the result is 1. Otherwise, the result is 0.  EXAMPLE:  unsigned int a = 5; // 0101 in binary  unsigned int b = 3; // 0011 in binary  unsigned int result = a | b; // Result: 0111 (7 in decimal)  Bitwise XOR (^):Performs a bitwise exclusive OR operation between the corresponding bits of two operands.If the bits are different, the result is 1. If the bits are the same, the result is 0.  EXAMPLE:  unsigned int a = 5; // 0101 in binary  unsigned int b = 3; // 0011 in binary  unsigned int result = a ^ b; // Result: 0110 (6 in decimal)  Bitwise NOT (~):Performs a bitwise NOT operation, which inverts each bit of the operand  EXAMPLE:  unsigned int a = 5; // 0101 in binary  unsigned int result = ~a; // Result: 1010 (10 in decimal)  Left Shift (<<):Shifts the bits of the left operand to the left by a specified number of positions.  EXAMPLE:  unsigned int a = 5; // 0000 0101 in binary  unsigned int result = a << 2; // Result: 0001 0100 (20 in decimal)  Right Shift (>>):Shifts the bits of the left operand to the right by a specified number of positions.  EXAMPLES:  unsigned int a = 20; // 0001 0100 in binary  unsigned int result = a >> 2; // Result: 0000 0101 (5 in decimal) |
| 1. Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate. 2. In C programming, the increment operator ++ can be used in two different forms: postfix and prefix. The main difference between them is  | **Aspect** | **Postfix Increment (variable++)** | **Prefix Increment (++variable)** | | --- | --- | --- | | Order of Evaluation | Evaluates the variable first and then increments it. | Increments the variable first and then evaluates it. | | Value Returned | Returns the original value of the variable, then increments it. | Returns the incremented value of the variable. | | Example | **c int result = num++;** | **c int result = ++num;** | | Variable's Value After | Variable is incremented after the expression is evaluated. | Variable is incremented before the expression is evaluated. |   Postfix Increment Operation:  int num = 5;  int result = num++; // Postfix increment  printf("Result: %d\n", result); // Output: Result: 5  printf("num: %d\n", num); // Output: num: 6  Prefix Increment :  int num = 5;  int result = ++num; // Prefix increment  printf("Result: %d\n", result); // Output: Result: 6  printf("num: %d\n", num); // Output: num: 6 |
| 1. What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions? 2. In C programming, the logical AND (&&) and logical OR (||) operators are used to perform logical operations on boolean expressions. These operators are commonly used in conditional statements and expressions to make decisions based on multiple conditions   Logical AND (&&): The logical AND operator (&&) returns true if both of its operands are true, and false otherwise. It evaluates the second operand only if the first operand evaluates to true. If the first operand is false, the second operand is not evaluated because the overall result will always be false.  EXAMPLE:  if (x > 0 && y < 10) {  // Executes if both x is greater than 0 AND y is less than 10  }  Logical OR (||): The logical OR operator (||) returns true if at least one of its operands is true, and false otherwise. It evaluates the second operand only if the first operand evaluates to false. If the first operand is true, the second operand is not evaluated because the overall result will always be true  EXAMPLE:  if (x == 0 || y == 0) {  // Executes if either x is equal to 0 OR y is equal to 0  } |
| 1. Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation 2. Operator precedence and associativity are two important concepts in C programming that determine the order in which operators are evaluated in expressions:   Operator Precedence: Operator precedence defines the priority of operators in an expression. It specifies which operators are evaluated first and which are evaluated later.  Operators with higher precedence are evaluated before operators with lower precedence.  For example, in the expression a + b \* c, the multiplication operator (\*) has higher precedence than the addition operator (+), so b \* c is evaluated first, followed by the addition of a. Parentheses can be used to override the default precedence and force certain operations to be evaluated first.  EXAMPLE: int result = 10 + 5 \* 2; // result is 20, multiplication is evaluated first  Operator Associativity: Operator associativity defines the direction in which operators of the same precedence are grouped and evaluated. It determines whether operators are evaluated from left to right (left associativity) or from right to left (right associativity).  For example, in the expression a - b - c, the subtraction operator (-) has left associativity, so a - b is evaluated first, followed by subtracting c from the result.  Some operators, such as assignment (=) and unary operators, have right associativity.  EXAMPLE: int result = 10 - 5 - 2; // result is 3, subtraction is evaluated left-to-right |
| Control Structures: |
| 1. Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement? 2. The switch statement in C programming provides a convenient way to execute different blocks of code based on the value of a variable or an expression. It is commonly used when you have a single expression that can take on multiple values, and you want to perform different actions based on each possible value. The switch statement enhances code readability and maintainability, especially when dealing with multiple conditional branches.   Purpose and Usage of the switch Statement:  Purpose: The switch statement is used to select one of many code blocks to be executed based on the value of a variable or an expression.  Usage: The general syntax of the switch statement is as follows:  switch (expression) {  case constant1:  // code block 1  break;  case constant2:  // code block 2  break;  // more cases as needed  default:  // default code block  }   | **Aspect** | **Switch Statement** | **If-Else Statement** | | --- | --- | --- | | Usage | Used to select one of many code blocks based on the value of a variable or an expression. | Used to execute code blocks based on conditions. | | Syntax | **c switch (expression) { case constant1: // code block 1 break; case constant2: // code block 2 break; // more cases as needed default: // default code block }** | **c if (condition1) { // code block 1 } else if (condition2) { // code block 2 } // more conditions as needed else { // default code block }** | | Conditions | Conditions are specified as constant values or expressions to be compared with the switch expression. | Conditions can be arbitrary boolean expressions. | | Fall-through Behavior | Control falls through to subsequent cases unless a break statement is encountered. | Control does not automatically fall through to subsequent conditions. | | Equality Comparison | Performs equality comparison between the expression and case constants. | Allows for more complex conditions using relational and logical operators. | | Constant Cases | Cases must be constant expressions. | Conditions can be arbitrary boolean expressions. | | Execution | More efficient when multiple conditions are based on the value of a single expression. | More suitable for complex conditions and non-constant expressions. | |
| 1. Explain the concept of nested control structures in C programming. Provide an example demonstrating nested if-else statements. 2. In C programming, nested control structures refer to the practice of using one control structure (such as an if-else statement, a switch statement, a loop, etc.) within another control structure. This means that control structures can be nested within each other to create more complex decision-making processes or repetitive tasks.   Purpose and Usage of Nested Control Structures:  Purpose: Nested control structures allow for the creation of more complex logic by combining multiple decision-making or iterative processes.  Usage: Nested control structures are used when the decision-making or repetitive task requires multiple levels of conditions or iterations  EXAMPLE:  #include <stdio.h>  int main() {  int num = 25;  if (num > 0) {  printf("Number is positive.\n");  if (num % 2 == 0) {  printf("Number is even.\n");  } else {  printf("Number is odd.\n");  }  } else if (num < 0) {  printf("Number is negative.\n");  } else {  printf("Number is zero.\n");  }  return 0;  }  Nested control structures allow for the creation of more intricate decision-making processes or iterative tasks, making programs more flexible and capable of handling various scenarios. However, it's important to ensure clarity and readability when using nested control structures to avoid confusion and facilitate maintenance. |
| 1. Discuss the role of the break and continue statements in loop control in C programming. Provide examples to illustrate their usage. 2. In C programming, the break and continue statements are used for loop control, allowing you to alter the flow of execution within loops. They provide mechanisms to skip iterations, prematurely exit loops, or jump to the next iteration based on certain conditions.   1. break Statement:  The break statement is used to terminate the execution of a loop prematurely.  When encountered within a loop, it immediately exits the loop, regardless of the loop condition.  It is commonly used to exit a loop early when a specific condition is met.  EXAMPLE:  #include <stdio.h>  int main() {  int i;  for (i = 1; i <= 10; i++) {  if (i == 5) {  break; // Exit the loop when i is equal to 5  }  printf("%d ", i);  }  return 0;  }  continue Statement:  The continue statement is used to skip the remaining code within a loop for the current iteration.  When encountered, it causes the loop to immediately jump to the next iteration, skipping any subsequent code within the loop's body.  It is commonly used to skip certain iterations based on specific conditions without exiting the loop entirely.  EXAMPLE:  #include <stdio.h>  int main() {  int i;  for (i = 1; i <= 10; i++) {  if (i % 2 == 0) {  continue; // Skip even numbers  }  printf("%d ", i);  }  return 0;  }  Role of break and continue Statements:  Control Flow Alteration: Both break and continue statements provide mechanisms to alter the flow of control within loops.  Loop Termination: break is used to prematurely terminate the loop based on a condition.  Iteration Skipping: continue is used to skip the remaining code within the loop for the current iteration and move to the next iteration.  Flexibility: These statements enhance the flexibility and control over loop execution, allowing for the implementation of more complex algorithms and conditions. |
| 1. Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements? 2. In C programming, both the for loop and the while loop are used for iteration, but they have different syntax and usage patterns. While both loops can achieve similar results, each has its advantages in specific situations.   Advantages of the for Loop over the while Loop:  Compact Syntax: The for loop has a more compact syntax, making it suitable for iterating over a range of values with a clear initialization, condition, and update expression, all in one line.  Initialization and Update Expression: The for loop allows for the initialization and update of loop variables within the loop header, which can improve code readability by keeping loop-related operations together.  Convenience for Count-Controlled Iteration: The for loop is especially convenient for count-controlled iteration, where the number of iterations is predetermined or based on a counter.  Example Comparing for and while Loops:  Using a for Loop:  #include <stdio.h>  int main() {  // Example: Print numbers from 1 to 5 using a for loop  printf("Using a for loop:\n");  for (int i = 1; i <= 5; i++) {  printf("%d ", i);  }  printf("\n");  return 0;  }  Using a while Loop:  #include <stdio.h>  int main() {  // Example: Print numbers from 1 to 5 using a while loop  printf("Using a while loop:\n");  int i = 1;  while (i <= 5) {  printf("%d ", i);  i++;  }  printf("\n");  return 0;  } |
| 1. Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements? 2. Short-circuit evaluation is a concept in C programming (and many other programming languages) where the evaluation of a logical expression stops as soon as the final result can be determined. In other words, the evaluation "short-circuits" once the outcome is known, without evaluating the rest of the expression unnecessarily.   How Short-Circuit Evaluation Works:  Logical AND (&&): In a logical AND expression (A && B), if A evaluates to false, there's no need to evaluate B because the overall result will always be false regardless of the value of B. Therefore, if A is false, the evaluation stops, and B is not evaluated. This is short-circuiting.  Logical OR (||): In a logical OR expression (A || B), if A evaluates to true, there's no need to evaluate B because the overall result will always be true regardless of the value of B. Therefore, if A is true, the evaluation stops, and B is not evaluated. This is short-circuiting  Impact on if Statements:  Short-circuit evaluation has a significant impact on the evaluation of logical expressions in if statements, especially when expressions involve multiple conditions connected by logical AND (&&) or logical OR (||) operators.  Example with Logical AND (&&):  int a = 5;  int b = 0;  if (a > 0 && b > 0) {  // Code here will not be executed because b > 0 will not be evaluated  }  Example with Logical OR (||):  int a = 5;  int b = 0;  if (a > 0 || b > 0) {  // Code here will be executed because a > 0 is true, and b > 0 will not be evaluated  }  Benefits of Short-Circuit Evaluation:  Efficiency: Short-circuit evaluation can improve performance by avoiding unnecessary evaluations, especially in complex logical expressions or when evaluating expressions involving expensive function calls.  Safety: Short-circuit evaluation can prevent errors or exceptions that may occur due to evaluating expressions with undefined behavior or invalid operands. |
| Functions: |
| 1. Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes? 2. In C programming, a function prototype serves as a declaration of a function before its actual implementation. It provides the compiler with information about the function's name, return type, and parameters, allowing the compiler to perform type checking and validation when the function is called. The purpose and structure of a function prototype are as follows:   Purpose of Function Prototypes:  Type Checking: Function prototypes enable the compiler to perform type checking to ensure that the function is used correctly, with the correct number and types of arguments, and that the return value is handled appropriately.  Forward Declaration: Function prototypes allow functions to be declared before they are defined or implemented in the code, enabling modular programming and separating interface from implementation.  Documentation: Function prototypes serve as documentation for other programmers, providing information about the function's name, return type, and parameters.  Structure of a Function Prototype:  Return Type: Specifies the data type of the value returned by the function.  Function Name: Specifies the name of the function.  Parameter List: Specifies the data types and names of the parameters (if any) passed to the function. If the function takes no parameters, the parameter list is left empty.  Semicolon: Ends the function prototype declaration with a semicolon.  // Function prototype declaration  return\_type function\_name(parameter1\_type parameter1, parameter2\_type parameter2, ...);  Why is it Necessary to Declare Function Prototypes?  Enable Early Detection of Errors: Function prototypes enable the compiler to perform early detection of errors, such as mismatched data types or incorrect number of arguments, during compilation rather than at runtime.  Modular Programming: Function prototypes allow for modular programming by separating the interface (prototype) from the implementation (definition), enabling code to be organized into separate files or modules.  Avoid Implicit Declarations: Without function prototypes, the compiler may implicitly declare functions based on their usage in the code, leading to potential errors and inconsistencies. Function prototypes provide explicit declarations, ensuring consistency and predictability in function usage. |
| 1. Explain the difference between call by value and call by reference in C programming. Provide examples to illustrate both concepts   A)In C programming, functions can be called using two different methods: call by value and call by reference. These methods determine how arguments are passed to functions and how changes made to those arguments within the function affect the original variables passed as arguments.  Call by Value:  In call by value, a copy of the actual parameter's value is passed to the function. The function works with this copy, and any changes made to the copy do not affect the original variable in the calling function.  Example of Call by Value:  #include <stdio.h>  // Function to increment a value passed by value  void incrementByValue(int x) {  x++;  printf("Inside function: x = %d\n", x);  }  int main() {  int num = 5;  printf("Before function call: num = %d\n", num);  incrementByValue(num); // Call by value  printf("After function call: num = %d\n", num);  return 0;  }  Call by Reference:  In call by reference, the address (reference) of the actual parameter is passed to the function. This allows the function to directly access and modify the original variable using its address.  Example of Call by Reference  #include <stdio.h>  // Function to increment a value passed by reference  void incrementByReference(int \*ptr) {  (\*ptr)++;  printf("Inside function: \*ptr = %d\n", \*ptr);  }  int main() {  int num = 5;  printf("Before function call: num = %d\n", num);  incrementByReference(&num); // Call by reference  printf("After function call: num = %d\n", num);  return 0;  } |
| 1. Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works 2. Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. It's a fundamental concept in computer science and is widely used to solve problems that can be broken down into smaller, similar subproblems. In C programming, recursion provides an elegant and concise solution to such problems.   How Recursion Works:  Base Case: Every recursive function must have one or more base cases, which are conditions under which the function stops calling itself and returns a result without further recursion. Base cases prevent infinite recursion.  Recursive Case: The function continues to call itself with modified arguments until it reaches the base case(s). Each recursive call typically reduces the problem into a smaller, simpler subproblem.  Termination: Once the base case is reached, the recursion stops, and the function returns a result. The results of all recursive calls are combined to solve the original problem. |
| 1. What is the significance of the return statement in C programming? How are values returned from functions? 2. In C programming, the return statement serves the crucial purpose of returning a value from a function to the caller. It signifies the end of the function's execution and provides a mechanism for passing back a result or data to the part of the program that called the function. The significance of the return statement lies in its ability to:   Provide Results: Functions often perform computations or operations and need to return the result to the caller for further processing or usage.  Terminate Function Execution: The return statement terminates the execution of the function, allowing control to pass back to the calling part of the program.  Pass Control Back: It transfers control back to the calling function or program, allowing the program to continue execution after the function call.  How Values are Returned from Functions:  Values are returned from functions using the return statement followed by the value to be returned. The syntax of the return statement is as follows:  #include <stdio.h>  // Function to calculate the sum of two integers  int sum(int a, int b) {  return a + b; // Return the sum of a and b  }  int main() {  int result;  result = sum(5, 3); // Call the sum function and store the result  printf("The sum is: %d\n", result); // Print the result  return 0;  } |
| 1. Describe the role of function parameters and arguments in C programming. How are function arguments passed to parameters?   A)In C programming, function parameters and arguments play vital roles in defining and using functions. They enable the passing of data between different parts of a program, allowing functions to perform operations on input values and produce output results. Here's an overview of their roles and how function arguments are passed to parameters:  Function Parameters:  Definition: Function parameters are variables declared within the parentheses of a function declaration or definition. They represent the input values that the function expects to receive when it is called.  Purpose: Parameters define the data that a function requires to perform its task. They act as placeholders for values that will be passed to the function when it is invoked.  Syntax: Parameters are declared within the parentheses following the function name. Each parameter includes its data type and, optionally, a name.  Example:  void calculateSum(int num1, int num2) {  // Function body  }  Function Arguments:  Definition: Function arguments are the actual values passed to a function when it is called. They provide the data that the function will operate on.  Purpose: Arguments supply the necessary input values to the function, enabling it to perform its task with specific data.  Syntax: Arguments are the values passed within the parentheses when calling a function. They must match the data types and order of the corresponding parameters in the function declaration or definition.  int a = 10, b = 20;  calculateSum(a, b); // Function call with arguments  Passing Arguments to Parameters:  In C programming, function arguments are passed to parameters using one of the following methods:  Pass by Value:  In pass by value, a copy of the argument's value is passed to the function parameter.  The function operates on this copy, and any changes made to the parameter do not affect the original argument.  It's the default method of passing arguments to parameters in C  void calculateSum(int x, int y) {  // x and y are copies of the arguments passed to the function  }  Pass by Reference (Using Pointers):  In pass by reference, the address (reference) of the argument is passed to the function parameter using pointers.  The function can then directly access and modify the original argument using its address.  This method allows functions to modify the original data passed as arguments.  Example:  void calculateSum(int \*ptr1, int \*ptr2) {  // \*ptr1 and \*ptr2 are references to the original values passed to the function  } |

|  |
| --- |
| **Arrays** |
| **1.** Explain the concept of arrays in C programming. How are arrays declared and initialized?  A. Arrays allow you to store multiple values of the same data type under a single variable name, making it easier to manage and access related data.  To declare an array in C, you specify the data type of the elements in the array, followed by the array name and the size of the array in square brackets []. The syntax for declaring an array is as follows:  datatype arrayName[arraySize];  Here, datatype represents the type of elements that will be stored in the array (e.g., int, float, char), arrayName is the name of the array, and arraySize specifies the number of elements the array can hold. For example:  int numbers[5]; // Declares an integer array named numbers with 5 elements  Arrays can be initialized at the time of declaration using an initializer list, which contains the initial values enclosed in curly braces {}. The number of elements in the initializer list must match the size of the array declared. Here's the syntax for initializing an array:  datatype arrayName[arraySize] = {value1, value2, ..., valueN}; |
| **2.** Discuss the difference between a one-dimensional array and a multi dimensional array in C programming. Provide examples of both.  A. One-dimensional array: A one-dimensional array is a collection of elements arranged in a linear sequence. It requires only one index to access individual elements. One-dimensional arrays are often used to represent lists or sequences of data.  Example of a one-dimensional array in C:  int numbers[5]; // Declaration of a one-dimensional array named 'numbers' with 5 elements  // Initialization of the array  numbers[0] = 10;  numbers[1] = 20;  numbers[2] = 30;  numbers[3] = 40;  numbers[4] = 50;  Multi-dimensional array: A multi-dimensional array is an array with more than one dimension. It requires multiple indices to access individual elements. Multi-dimensional arrays are often used to represent tables, matrices, or higher-dimensional data structures.  Example of a two-dimensional array in C:  int matrix[3][3]; // Declaration of a two-dimensional array named 'matrix' with 3 rows and 3 columns  // Initialization of the array  matrix[0][0] = 1;  matrix[0][1] = 2;  matrix[0][2] = 3;  matrix[1][0] = 4;  matrix[1][1] = 5;  matrix[1][2] = 6;  matrix[2][0] = 7;  matrix[2][1] = 8;  matrix[2][2] = 9; |
| **3.** Describe the process of accessing array elements in C programming. How are array indices used to access elements?  A. Accessing array elements in C programming involves using array indices to specify the position of the desired element within the array. Array indices are integer values that represent the position of an element relative to the beginning of the array. Here's how the process works:  Array Declaration: First, you declare an array of a specific data type and size. For example:  int numbers[5]; // Declares an integer array named 'numbers' with 5 elements  Array Initialization (Optional): You may initialize the array with values at the time of declaration or later in the program. For example:  int numbers[5] = {10, 20, 30, 40, 50}; // Initializes the array with values  Accessing Array Elements: To access an element of the array, you use square brackets [] along with the index of the element you want to access. Array indices in C are zero-based, meaning the first element of the array has an index of 0, the second element has an index of 1, and so on.  For example, to access the third element of the array 'numbers', you would use:  int thirdElement = numbers[2];  In this example, numbers[2] refers to the third element of the array because array indices start from 0. Similarly, numbers[0] refers to the first element, numbers[1] refers to the second element, and so forth.  Using Variables as Indices: You can also use variables as indices to access array elements. For example:  int index = 3;  int element = numbers[index];  In this case, the value of the variable index determines which element of the array 'numbers' will be accessed.  Bounds Checking: It's important to ensure that the index used to access an array element is within the bounds of the array. Accessing elements outside the bounds of the array can lead to undefined behavior or segmentation faults.  Iterating Through Arrays: Arrays are often accessed in loops for operations such as initialization, searching, or manipulation. For example:  for (int i = 0; i < 5; i++) {  printf("%d ", numbers[i]); // Prints each element of the 'numbers' array  }  This loop iterates through the array 'numbers' and prints each element sequentially. |
| **4.** What is the significance of the null character ('\0') in C strings? How is it used to determine the end of a string?  A. strings are represented as arrays of characters terminated by a special character called the null character ('\0'). The null character has a numeric value of zero and is used to denote the end of a string. It serves as a sentinel value indicating where the string ends.  The significance of the null character lies in its role as a string terminator. When processing strings in C, functions that operate on strings, such as strlen() (string length), strcpy() (string copy), strcmp() (string comparison), and others, rely on the presence of the null character to determine the end of the string.  For example, when you declare and initialize a string in C, the compiler automatically appends a null character ('\0') at the end of the string literal to mark its termination. Here's an example:  char myString[] = "Hello"; // Compiler automatically adds '\0' at the end  In this case, the string "Hello" is actually stored as 'H', 'e', 'l', 'l', 'o', '\0' in memory. The presence of '\0' after the last character ('o') indicates the end of the string.  Functions that process strings typically iterate through each character of the string until they encounter the null character. When the null character is encountered, it signifies the end of the string, and the processing stops.  Here's an example of how the null character is used to determine the length of a string using the strlen() function:  #include <stdio.h>  #include <string.h>  int main() {  char myString[] = "Hello";  int length = strlen(myString); // strlen() returns the length of the string excluding '\0'  printf("Length of the string: %d\n", length); // Output: Length of the string: 5      return 0;  } |
| **5.** Explain the concept of dynamic memory allocation for arrays in C programming. How are dynamic arrays allocated and deallocated?  A. Allocation of Dynamic Arrays:  malloc(): This function allocates a specified number of bytes of memory and returns a pointer to the allocated memory. You need to specify the number of bytes to allocate. Here's the syntax:  type ptr = (type) malloc(numBytes);  For example, to allocate memory for an array of integers containing 5 elements:  int arr = (int) malloc(5 \* sizeof(int));  calloc(): This function allocates memory for an array of elements, initialized to zero. It requires two arguments: the number of elements to allocate and the size of each element. Here's the syntax:  type ptr = (type) calloc(numElements, sizeof(type));  For example, to allocate memory for an array of 5 integers initialized to zero:  int arr = (int) calloc(5, sizeof(int));  realloc(): This function reallocates memory for an existing dynamic array, allowing you to resize it. It requires two arguments: a pointer to the existing memory block and the new size in bytes. Here's the syntax:  ptr = (type\*) realloc(ptr, newSize);  For example, to resize the previously allocated array arr to hold 10 integers:  arr = (int\*) realloc(arr, 10 \* sizeof(int));  Deallocation of Dynamic Arrays:  free(): After you've finished using dynamically allocated memory, it's essential to deallocate it to prevent memory leaks. The free() function is used to release dynamically allocated memory. It takes a pointer to the memory block as an argument. Here's the syntax:  free(ptr);  For example, to deallocate the dynamically allocated array arr:  free(arr);  It's important to note that dynamically allocated memory persists until explicitly deallocated using free(). Failure to deallocate dynamically allocated memory can lead to memory leaks, where memory is no longer accessible to the program but remains allocated, consuming system resources. Additionally, it's crucial to check if memory allocation operations (malloc(), calloc(), realloc()) are successful by verifying if the returned pointer is not NULL. |

|  |
| --- |
| **Pointers** |
| 1.Describe the purpose and usage of pointers in C programming. How are pointers declared and initialized?  -Pass Arguments by Reference  -Accessing Array Elements  -Return Multiple Values from Function  -Dynamic Memory Allocation  -Implementing Data Structures  -In System-Level Programming where memory addresses are useful.  In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the ( \* ) dereference operator before its name.  Example  int \*ptr;  Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the ( & ) addressof operator to get the memory address of a variable and then store it in the pointer variable.  Example  int var = 10;  int \* ptr;  ptr = &var; |
| 2. Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers.  Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The pointer variables store the memory address of another variable. It doesn’t store any value.  The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:  Increment/Decrement of a Pointer  Addition of integer to a pointer  Subtraction of integer to a pointer  Subtracting two pointers of the same type  Comparison of pointers  Addition:  #include <stdio.h>  int main()  {  int N = 4;  int \*ptr1, \*ptr2;  ptr1 = &N;  ptr2 = &N;  printf("Pointer ptr2 before Addition: ");  printf("%p \n", ptr2);  ptr2 = ptr2 + 3;  printf("Pointer ptr2 after Addition: ");  printf("%p \n", ptr2);  return 0;  }  Sub:  #include <stdio.h>  int main()  {  int N = 4;  int \*ptr1, \*ptr2;  ptr1 = &N;  ptr2 = &N;  printf("Pointer ptr2 before Subtraction: ");  printf("%p \n", ptr2);  ptr2 = ptr2 - 3;  printf("Pointer ptr2 after Subtraction: ");  printf("%p \n", ptr2);  return 0;  } |
| 3.Discuss the difference between pass by value and pass by reference in function arguments using pointers in C programming. Provide examples to illustrate both approaches.  A. a) Effect on Original Variable: Pass by value does not affect the original variable, while pass by reference using pointers allows modifications to the original variable.  b) Memory Overhead: Pass by value involves copying data, which can lead to memory overhead for large data types. Pass by reference using pointers avoids this overhead by directly accessing the memory.  c) Pointer Syntax: Pass by reference requires the use of pointers and the address-of operator (&) to pass the memory address of the variable.  Pass by value:  #include <stdio.h>  void increment(int num) {  num++; // Increment num  printf("Inside function: %d\n", num);  }  int main() {  int x = 5;  increment(x); // Pass x by value  printf("Outside function: %d\n", x);      return 0;  }  Pass by reference  #include <stdio.h>  void increment\_by\_reference(int \*num) {  (\*num)++; // Increment value at the memory address stored in num  printf("Inside function: %d\n", \*num);  }  int main() {  int x = 5;  increment\_by\_reference(&x); // Pass address of x  printf("Outside function: %d\n", x);  return 0;  } |
| **4.** Describe the concept of NULL pointers in C programming. How are NULL pointers used and checked for in programs?  A. Concept of NULL Pointers:  No Memory Address: A NULL pointer does not point to any valid memory location.  Initialization: Pointers are often initialized to NULL when they are declared but not yet assigned a valid memory address.  Usage of NULL Pointers:  Initialization: Initialize pointers to NULL to avoid accidental dereferencing of uninitialized pointers.  Error Handling: Use NULL to indicate failure in memory allocation functions like malloc, calloc, and realloc.  Checking for NULL Pointers:  Equality Check: To check if a pointer is NULL, compare it with the literal NULL or use the macro NULL defined in <stddef.h>. |
| **5.** Explain the role of pointers in dynamic memory allocation in C programming. How are pointers used to allocate and deallocate memory dynamically?  A. Role of Pointers in Dynamic Memory Allocation:  Allocation: Pointers are used to allocate memory dynamically using functions like malloc, calloc, and realloc. These functions return a pointer to the allocated memory block.  Deallocation: Pointers are also used to deallocate dynamically allocated memory using the free function. This releases the memory back to the system for reuse.  Steps for Dynamic Memory Allocation:  Allocation:  Use malloc to allocate memory for a specified number of bytes.  This function returns a void pointer (void \*), which can be cast to the appropriate data type.  c  Copy code  int \*ptr = (int \*)malloc(10 \* sizeof(int));  Deallocation:  Use free to release the memory allocated dynamically.  After calling free, the pointer should no longer be used to access the memory.  c  Copy code  free(ptr);  ptr = NULL; // Optional but a good practice to avoid dangling pointers  Example:  c  Copy code  #include <stdio.h>  #include <stdlib.h>  int main() {  int \*ptr;  int n = 5;  // Allocate memory for an array of n integers  ptr = (int \*)malloc(n \* sizeof(int));  if (ptr == NULL) {  printf("Memory allocation failed\n");  return 1;  }  // Initialize array elements  for (int i = 0; i < n; i++) {  ptr[i] = i + 1;  }  // Print array elements  printf("Array elements: ");  for (int i = 0; i < n; i++) {  printf("%d ", ptr[i]);  }  printf("\n");  // Deallocate memory  free(ptr);  ptr = NULL;      return 0;  } |
|  |
|  |

|  |
| --- |
| **Strings**  **1.** Discuss the concept of strings in C programming. How are strings represented and manipulated in C?  A. strings are represented as arrays of characters terminated by a null character ('\0'), also called the null terminator. This null character marks the end of the string.  Representation of Strings:  Strings in C are represented as arrays of characters. The last element of the array is always the null character ('\0'), indicating the end of the string. Here's an example of a string declaration:  char str[6] = "Hello";  String Manipulation:  C provides several standard library functions for string manipulation, declared in the <string.h> header. Some commonly used functions include:  strcpy and strncpy: Copy strings.  strcpy(destination, source);  strncpy(destination, source, num);  strcmp and strncmp: Compare strings.  strcpy and strncpy: Compare strings.  int result = strcmp(str1, str2);  int result = strncmp(str1, str2, num);  strcat and strncat: Concatenate strings.  strcpy and strncpy: Concatenate strings.  strcat(destination, source);  strncat(destination, source, num); |
| **2.** Explain the difference between character arrays and string literals in C programming. Provide examples to illustrate both concepts.  A. Character Arrays:  A character array is a sequence of characters stored in contiguous memory locations, typically declared as an array of characters. It allows for manipulation of individual characters and modification of the string content.  Example:  #include <stdio.h>  int main() {  char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};  printf("Character Array: %s\n", str); // Output: Hello      return 0;  }  String Literals:  A string literal is a sequence of characters enclosed in double quotes ("). It represents a constant string and is stored in read-only memory. String literals are automatically null-terminated by the compiler.  Example:  #include <stdio.h>  int main() {  char \*str = "Hello";  printf("String Literal: %s\n", str); // Output: Hello      return 0;  } |
| **3.**  A. 1. strlen:  Description: Calculates the length of a string.  Syntax: size\_t strlen(const char \*str);  Example:  #include <stdio.h>  #include <string.h>  int main() {  char str[] = "Hello";  size\_t length = strlen(str);  printf("Length of string: %zu\n", length); // Output: 5      return 0;  }  2. strcpy:  Description: Copies a string.  Syntax: char \*strcpy(char \*destination, const char \*source);  Example:  #include <stdio.h>  #include <string.h>  int main() {  char source[] = "Hello";  char destination[10];  strcpy(destination, source);  printf("Copied string: %s\n", destination); // Output: Hello  return 0;  }  3. strcat:  Description: Concatenates two strings.  Syntax: char \*strcat(char \*destination, const char \*source);  Example:  #include <stdio.h>  #include <string.h>  int main() {  char str1[10] = "Hello";  char str2[] = " World";  strcat(str1, str2);  printf("Concatenated string: %s\n", str1); // Output: Hello World  return 0;  }  4. strcmp:  Description: Compares two strings.  Syntax: int strcmp(const char \*str1, const char \*str2);  Example:  #include <stdio.h>  #include <string.h>  int main() {  char str1[] = "Hello";  char str2[] = "World";  int result = strcmp(str1, str2);  if (result == 0) {  printf("Strings are equal\n");  } else if (result < 0) {  printf("str1 is less than str2\n");  } else {  printf("str1 is greater than str2\n");  }  return 0;  }  5. strncpy:  Description: Copies a specified number of characters from one string to another.  Syntax: char \*strncpy(char \*destination, const char \*source, size\_t num);  Example:  #include <stdio.h>  #include <string.h>  int main() {  char source[] = "Hello";  char destination[10];  strncpy(destination, source, 3); // Copy only first 3 characters  destination[3] = '\0'; // Null terminate the destination string  printf("Copied string: %s\n", destination); // Output: Hel      return 0;  } |
| **4.** Discuss the concept of string tokenization in C programming. How are strings split into tokens using delimiter characters?  A. String tokenization in C programming involves breaking a string into smaller parts or tokens based on specified delimiter characters. This process is useful for parsing input strings or extracting meaningful information from text data. The C standard library provides the strtok function for tokenizing strings.  Concept of String Tokenization:  Delimiter: A delimiter is a character used to separate tokens within a string. Common delimiter characters include spaces, commas, semicolons, etc.  Token: A token is a substring of the original string that is separated by delimiter characters.  strtok Function:  Description: strtok function breaks a string into tokens based on specified delimiter characters.  Syntax: char \*strtok(char \*str, const char \*delimiters);  Parameters:  str: The string to tokenize.  delimiters: A string containing delimiter characters. |
| **5.** Explain the importance of null-terminated strings in C programming. How does the null character ('\0') signify the end of a string?  A. The importance of null-terminated strings in C programming lies in their simplicity and efficiency for representing and manipulating strings. The null character ('\0') serves as a sentinel value that signifies the end of a string. Here's why null-terminated strings are significant:  \*Simple Representation: Null-terminated strings are represented as arrays of characters where the end of the string is marked by the null character ('\0'). This representation is straightforward and easy to understand.  \*Efficient String Manipulation: Null-terminated strings allow for efficient string manipulation using pointer arithmetic and standard library functions. Functions like strlen, strcpy, strcmp, etc., rely on the null character to determine the length and boundaries of strings.  \*Compatible with Standard Library: The C standard library provides numerous functions for working with null-terminated strings, making them a standard and widely accepted convention in C programming.  \*Memory Efficient: Null-terminated strings require only one extra byte for the null character, making them memory-efficient compared to other string representations that require storing the length separately.  \*Compiler Support: C compilers and standard library functions are designed to work with null-terminated strings, ensuring compatibility and portability across different platforms and environments. |

|  |
| --- |
| **Structures and Unions** |
| **1.** Describe the purpose and usage of structures in C programming. How are structures declared and accessed?  Purpose and Usage of Structures:  \*Organizing Data: Structures allow you to organize related data elements into a single unit, making it easier to manage and manipulate data.  \*Creating Custom Data Types: Structures enable you to define custom data types that represent real-world entities with multiple properties.  \*Passing Complex Data to Functions: Structures are often used to pass complex data to functions by encapsulating related variables into a single parameter.  \*Memory Allocation: Structures facilitate memory allocation by providing a contiguous block of memory for all its members.  Declaration:  Structs are declared using the struct keyword followed by the name of the structure and a list of member variables enclosed in curly braces {}.  struct Student {  int id;  char name[50];     float gpa;  };  Accessing Members:  Individual members of a structure can be accessed using the dot (.) operator.  printf("ID: %d\n", s1.id);  printf("Name: %s\n", s1.name);  printf("GPA: %.2f\n", s1.gpa); |
| **2.** Discuss the concept of structure members in C programming. How are individual members of a structure accessed and modified?  **A.**Concept of Structure Members:  Data Type: Each member of a structure can have a different data type, including basic data types (int, float, char, etc.), arrays, pointers, or even other structures.  Name: Structure members are identified by their unique names, which are specified when the structure is defined.  Memory Layout: Structure members are stored in contiguous memory locations, and their order in memory is the same as their declaration order within the structure.  Accessing and Modifying Structure Members:  Accessing Members:  Individual members of a structure are accessed using the dot (.) operator followed by the member name.  struct Point {  int x;  int y;  };  struct Point p1 = {3, 5};  printf("X coordinate: %d\n", p1.x);  printf("Y coordinate: %d\n", p1.y);  Modifying Members:  Members of a structure can be modified by assigning new values using the dot (.) operator.  p1.x = 10;  p1.y = 20; |
| **3.** Explain the difference between structures and unions in C programming. When would you choose one over the other?  **A.**  Structures:  Memory Allocation:  Each member of a structure is allocated its own memory space.  The total memory allocated for a structure is the sum of the memory allocated for each member.  Usage:  Structures are used when you need to store and manage multiple related data elements simultaneously.  Each member of a structure holds different types of data, and all members are accessible simultaneously.  Unions:  Memory Allocation:  Unions allocate memory that is large enough to hold the largest member.  Only one member of a union can be active (i.e., contain a valid value) at any given time.  Usage:  Unions are used when you need to store different types of data in the same memory location.  Unions are useful for conserving memory when you only need one of several possible data types at any given time.  Choosing Between Structures and Unions:  Choose structures when you need to store and access multiple related data elements simultaneously.  Choose unions when you need to conserve memory and only one type of data is needed at any given time. |
| **4.**  A.In C programming, nested structures refer to structures that are defined within other structures. This allows for hierarchical organization of data, where a structure member itself is another structure. Nested structures are useful for representing complex data relationships and hierarchies. Here's how nested structures are defined and accessed  Syntax:  Nested structures are defined by placing one structure definition inside another structure definition.  struct OuterStruct {  int outerMember;  struct InnerStruct {  int innerMember;      } inner;  };  Dot Operator:  Members of nested structures are accessed using the dot (.) operator multiple times, first to access the outer structure member containing the inner structure, and then to access the inner structure member.  struct OuterStruct outer;  outer.inner.innerMember = 10; |
| **5.** Discuss the concept of typedef in C programming. How is typedef used to define custom data types, including structures and unions?  A. The syntax for typedef is as follows:  typedef existing\_data\_type new\_data\_type;  Purpose of typedef:  Creating Custom Data Types:  typedef is used to create custom data types with descriptive names, making the code more self-explanatory and easier to understand.  Abstraction and Encapsulation:  It abstracts the underlying data type, allowing changes to be made easily without affecting the rest of the code.  Improved Readability:  It enhances code readability by providing meaningful names for data types, especially when dealing with complex data structures.  Usage with Structures:  typedef is commonly used with structures to define custom data types representing complex data structures.  typedef struct {  int day;  int month;  int year;  } Date;  Usage with Unions:  Similarly, typedef can be used with unions to define custom data types representing variant data types.  typedef union {  int intValue;  float floatValue;  } Number; |

|  |
| --- |
| **FILE HANDLING:** |

|  |
| --- |
| 1.Explain the concept of file handling in C programming. How are files opened, read from, and written to using standard file handling functions?  Ans:  File handling in C programming allows you to work with files stored on the system's disk. This includes tasks such as opening files, reading data from them, writing data to them, and closing them when you're done. The <stdio.h> header file provides functions and macros for performing file handling operations in C.  Opening Files: Before you can read from or write to a file, you need to open it using the fopen() function. This function takes two parameters: the name of the file and the mode in which you want to open the file (e.g., read mode, write mode, append mode, etc.). The fopen() function returns a pointer to a FILE object, which you can then use in subsequent file operations.  Reading from Files: Once a file is opened for reading, you can use functions like fscanf() or fgets() to read data from the file into variables or arrays in your program.  Writing to Files: Similarly, when a file is opened for writing, you can use functions like fprintf() or fputs() to write data from your program into the file.  Closing Files: After you have finished working with a file, it's important to close it using the fclose() function. This ensures that any buffered data is written to the file and that system resources associated with the file are released  2. Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files?  Ans:  In C programming, file pointers play a crucial role in managing files. A file pointer is essentially a special type of pointer that points to the current position within a file. It's used to navigate through the file, read data from it, and write data to it. Here's how file pointers are typically used to navigate and manipulate files:  Positioning within the File: File pointers keep track of the current position within the file. When you open a file, the file pointer initially points to the beginning of the file. As you read or write data, the file pointer moves along with the operations. You can also explicitly move the file pointer to a specific position within the file using functions like fseek().    Here, fp is a pointer to a FILE object, offset specifies the number of bytes to move relative to origin, and origin specifies the reference point for the movement (SEEK\_SET for the beginning of the file, SEEK\_CUR for the current position, and SEEK\_END for the end of the file).  Reading Data: File pointers are used to read data from files using functions like fread(), fscanf(), or fgets(). These functions automatically move the file pointer as they read data, ensuring that subsequent reads start from the correct position.  Writing Data: Similarly, when writing data to a file using functions like fwrite(), fprintf(), or fputs(), the file pointer is automatically updated to reflect the new position in the file after the write operation is complete.  Closing Files: Finally, when you're done working with a file, you should close it using fclose(). Closing a file not only flushes any buffered data to disk but also releases system resources associated with the file, including the file pointer.  3. Discuss the difference between text files and binary files in C programming. How are they opened and processed differently?  Ans:  Text Files:  Text files store data in a human-readable format, typically consisting of characters encoded using ASCII or Unicode standards.  Data in text files is stored as a sequence of lines, with each line terminated by a newline character (\n).  Text files are commonly used for storing textual data such as configuration files, source code, plain text documents, etc.  When working with text files, C programming provides functions like fscanf(), fprintf(), fgets(), and fputs() to read from and write to text files.  Text files are opened in text mode ("r", "w", "a", etc.) using the fopen() function.  Binary Files:  Binary files store data in a format that is not human-readable and may contain any sequence of bytes, including data structures, images, executables, etc.  Unlike text files, binary files do not have a specific format for representing end-of-line characters or text encoding.  Binary files preserve the exact byte representation of the data, making them suitable for storing non-textual data or complex data structures.  When working with binary files, C programming provides functions like fread() and fwrite() to read from and write to binary files.  Binary files are opened in binary mode ("rb", "wb", "ab", etc.) using the fopen() function.  Differences and Considerations:  Data Representation: Text files store data as human-readable characters, while binary files store data in its raw byte representation.  End-of-Line Handling: Text files typically use newline characters to indicate the end of a line, while binary files do not have any specific end-of-line representation.  Processing Overhead: Reading and writing text files may involve additional processing overhead, such as character encoding/decoding and newline handling, compared to binary files.  Portability: Text files are more portable across different systems and platforms due to their reliance on standard character encodings. Binary files may require special handling for portability.  Size Consideration: Binary files may be more space-efficient for certain types of data since they do not require additional characters for encoding or newline characters.  4. Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc.  Ans:  "r" (Read Mode):  Opens the file for reading.  The file must exist; otherwise, the fopen() function will return NULL.  The file pointer is positioned at the beginning of the file.  Existing data in the file can be read, but writing operations are not permitted.  FILE \*fp;  fp = fopen("example.txt", "r");  "w" (Write Mode):  Opens the file for writing.  If the file already exists, its contents are truncated (i.e., deleted).  If the file does not exist, a new file is created.  The file pointer is positioned at the beginning of the file.  FILE \*fp;  fp = fopen("example.txt", "w");  "a" (Append Mode):  Opens the file for appending data to the end of the file.  If the file does not exist, it is created.  The file pointer is positioned at the end of the file, allowing data to be written without overwriting existing content.  FILE \*fp;  fp = fopen("example.txt", "a");  "r+" (Read/Write Mode):  Opens the file for both reading and writing.  The file must exist.  The file pointer is positioned at the beginning of the file.  FILE \*fp;  fp = fopen("example.txt", "r+");  "w+" (Read/Write Mode):  Opens the file for both reading and writing.  If the file exists, its contents are truncated.  If the file does not exist, a new file is created.  The file pointer is positioned at the beginning of the file.  FILE \*fp;  fp = fopen("example.txt", "w+");  "a+" (Read/Append Mode):  Opens the file for both reading and appending.  If the file does not exist, it is created.  The file pointer is positioned at the end of the file, allowing data to be appended without overwriting existing content.  FILE \*fp;  fp = fopen("example.txt", "a+");  5. Describe error handling techniques in file operations in C programming. How are errors detected and handled when working with files?  Ans:  Error handling is crucial when working with files in C programming to ensure that file operations are performed correctly and gracefully handle any unexpected situations. Here are some common error handling techniques used in file operations:  Check Return Values: Most file handling functions in C return a value that indicates whether the operation was successful or not. Always check the return value of file handling functions to detect errors. For example, when opening a file with fopen(), check if the returned FILE pointer is NULL to detect any errors in opening the file.  Set errno: In addition to returning NULL or an error code, file handling functions often set the global variable errno to indicate the specific type of error that occurred. You can use errno in conjunction with functions like perror() to print descriptive error messages.  Close Files Properly: Always remember to close files properly after you're done working with them. Closing files using fclose() ensures that any buffered data is written to the file and that system resources associated with the file are released.  Handle Errors Gracefully: When an error occurs during file operations, handle it gracefully. This may involve logging the error, informing the user, or taking corrective actions if possible.  Use ferror(): The ferror() function can be used to check if an error occurred during a file operation. It returns a non-zero value if an error occurred on the specified file stream. |

**Part- B**

|  |
| --- |
| 1. **Hello World** |
| Code  #include <stdio.h>  int main()  {  printf("Hello World");  return 0;  } |
|  |

2. Factorial: Calculate the factorial of a given number.

#include <stdio.h>

int factorial(unsigned int n)

{

int result = 1, i;

for (i = 2; i <= n; i++) {

result \*= i;

}

return result;

}

int main()

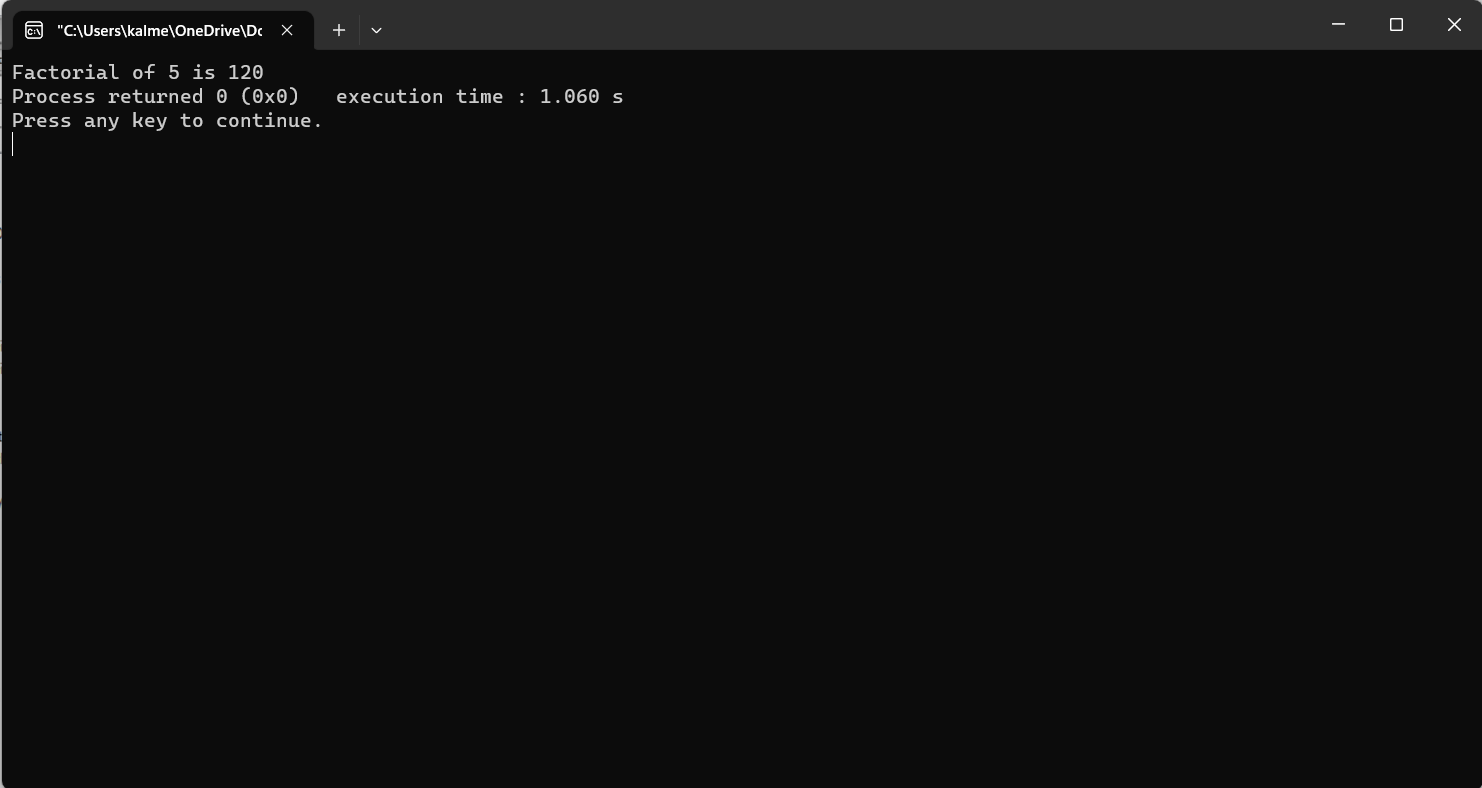
{

int num = 5;

printf("Factorial of %d is %d", num, factorial(num));

return 0;

}



3. Prime Numbers: Determine whether a given number is prime.

#include <stdio.h>

void checkPrime(int N)

{

int flag = 1;

for (int i = 2; i <= N / 2; i++) {

if (N % i == 0) {

flag = 0;

break;

}

}

if (flag) {

printf("The number %d is a Prime Number\n", N);

}

else {

printf("The number %d is not a Prime Number\n", N);

}

return;

}

int main()

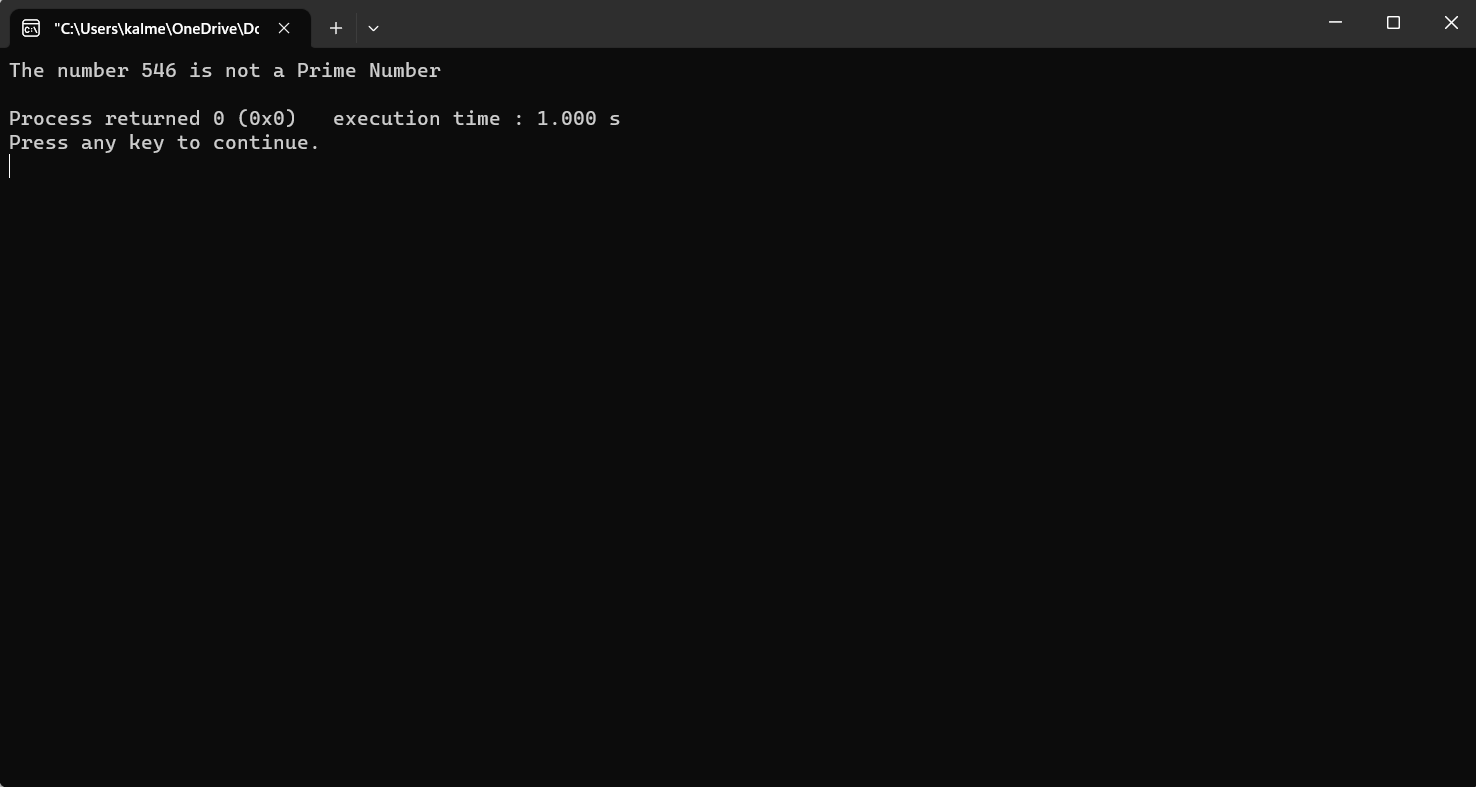
{

int N = 546;

checkPrime(N);

return 0;

}



4. Fibonacci Series: Generate the Fibonacci series up to a certain limit.

#include <stdio.h>

int main() {

int i, n;

int t1 = 0, t2 = 1;

int nextTerm = t1 + t2;

printf("Enter the number of terms: ");

scanf("%d", &n);

printf("Fibonacci Series: %d, %d, ", t1, t2);

for (i = 3; i <= n; ++i) {

printf("%d, ", nextTerm);

t1 = t2;

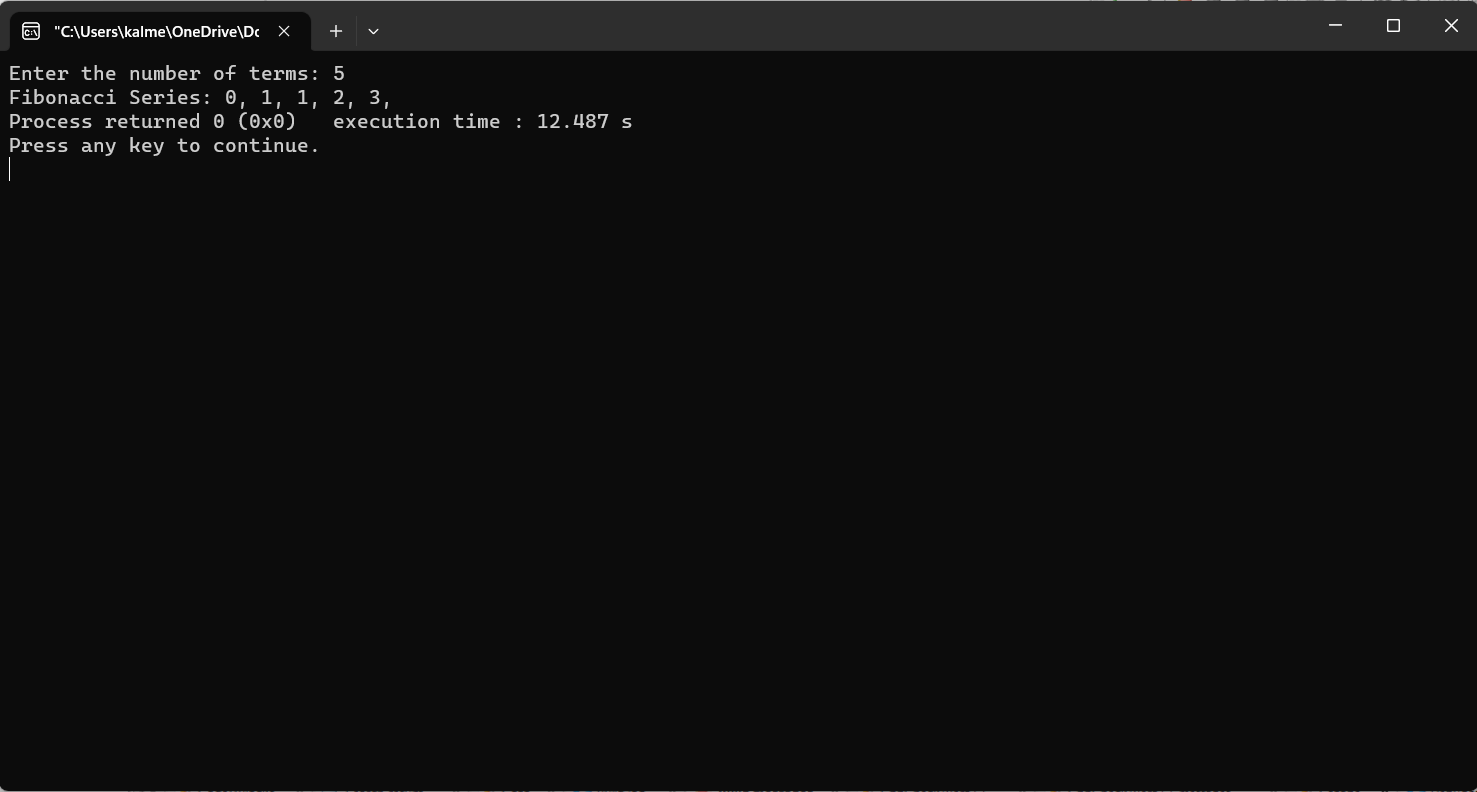
t2 = nextTerm;

nextTerm = t1 + t2;

}

return 0;

}



5. Sum of Digits: Calculate the sum of digits of a given number.

#include <stdio.h>

int main()

{

int num, sum=0;

printf("Enter any number to find sum of its digit: ");

scanf("%d", &num);

while(num!=0)

{

sum += num % 10;

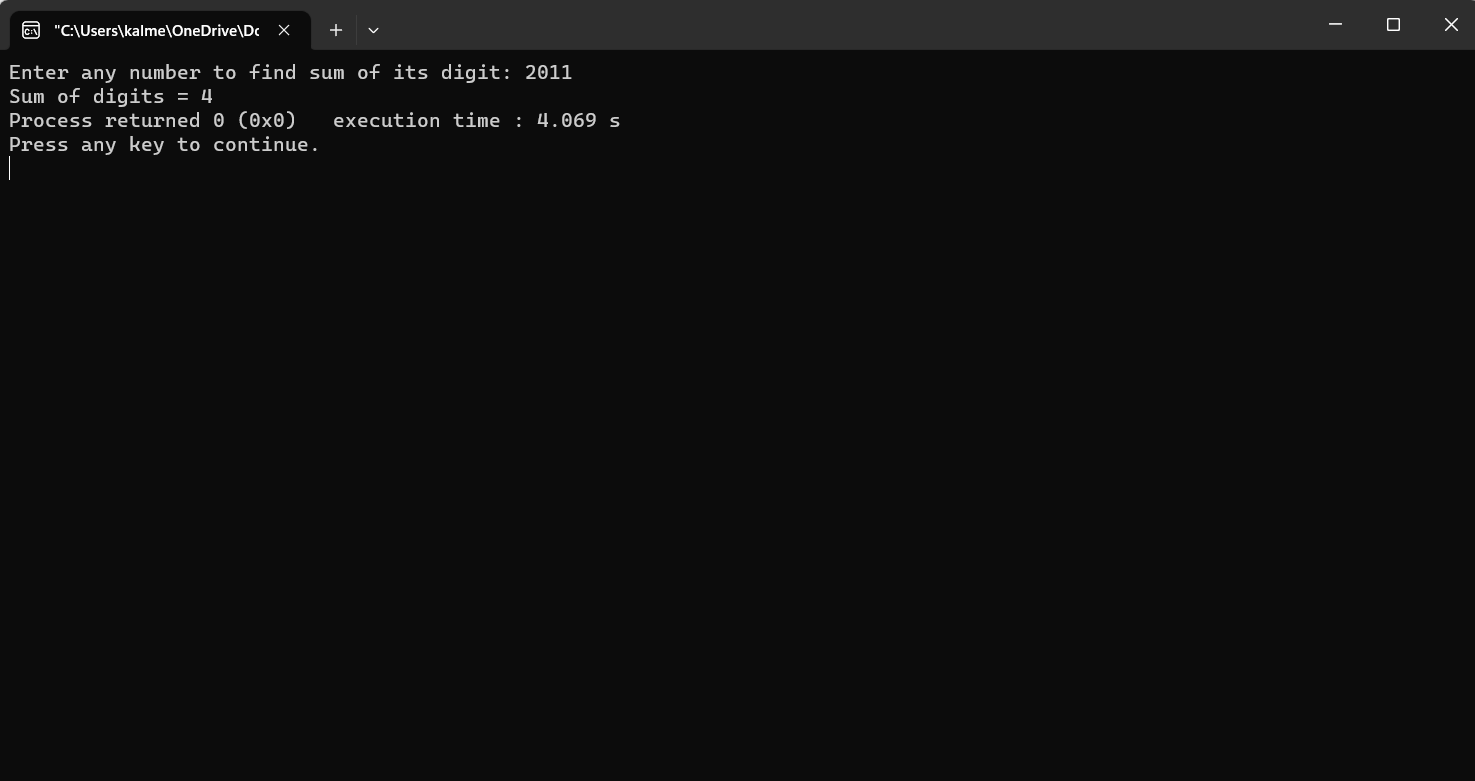
num = num / 10;

}

printf("Sum of digits = %d", sum);

return 0;

}



6. Reverse a Number: Reverse the digits of a given number.

#include <stdio.h>

int reverseDigits(int num)

{

int rev\_num = 0;

while (num > 0) {

rev\_num = rev\_num \* 10 + num % 10;

num = num / 10;

}

return rev\_num;

}

int main()

{

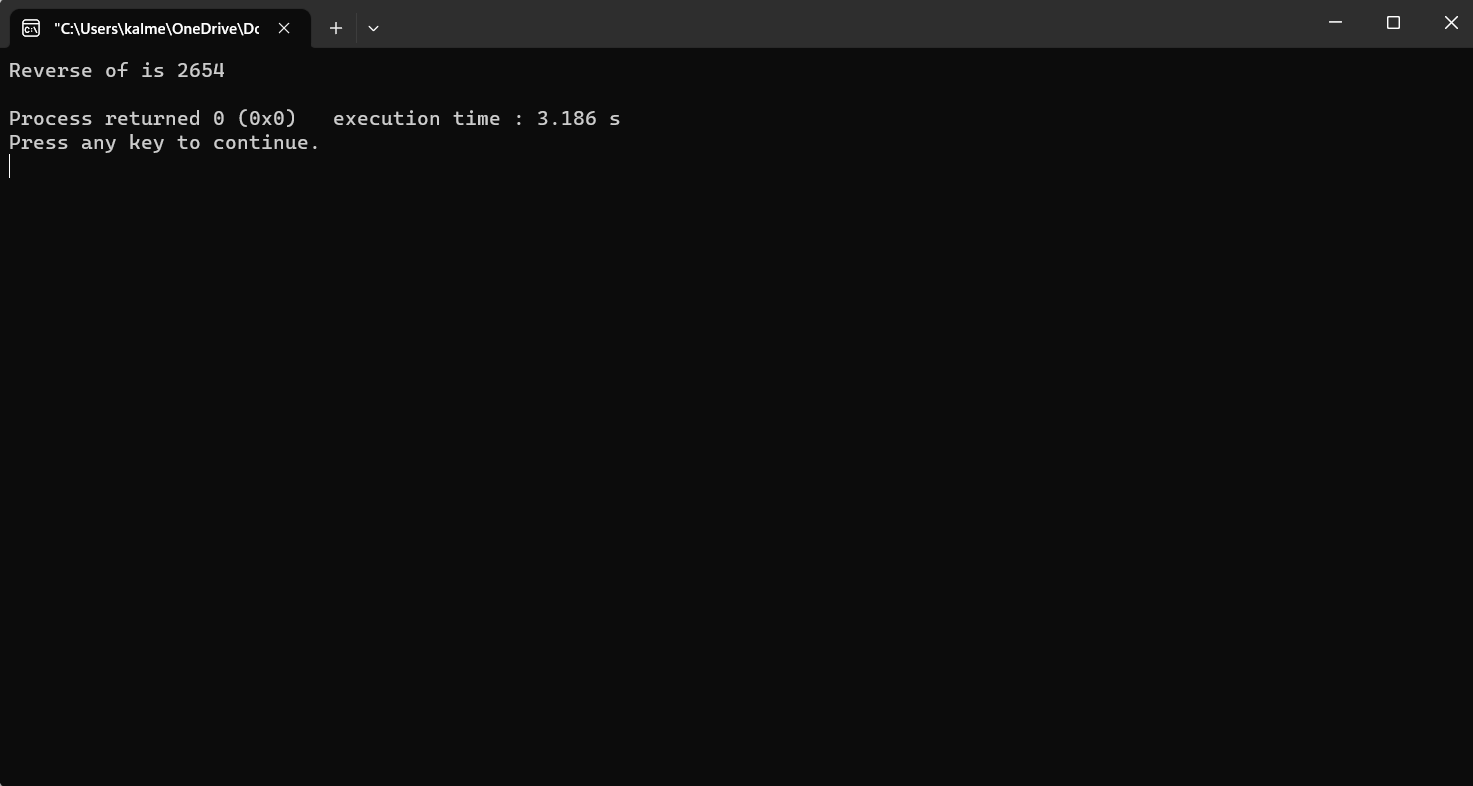
int num = 4562;

printf("Reverse of is %d", reverseDigits(num));

getchar();

return 0;

}



7. Palindrome Check: Check if a given number or string is a palindrome.

#include <stdio.h>

#include <string.h>

int main() {

char string1[20];

int i, length;

int flag = 0;

printf("Enter a string: ");

scanf("%s", string1);

length = strlen(string1);

for (i = 0; i < length / 2; i++) {

if (string1[i] != string1[length - i - 1]) {

flag = 1;

break;

}

}

if (flag) {

printf("%s is not a palindrome\n", string1);

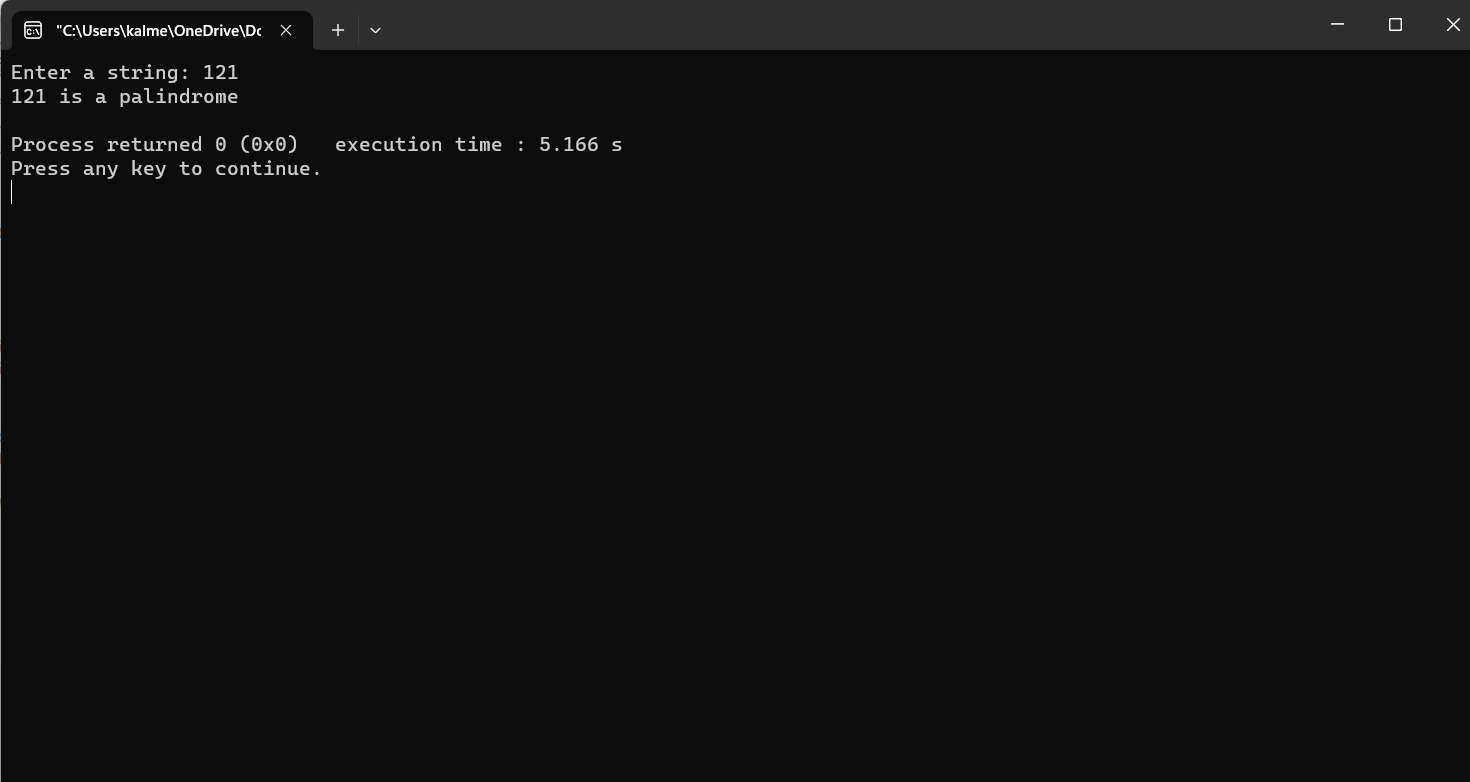
} else {

printf("%s is a palindrome\n", string1);

}

return 0;

}



8. Area of Shapes: Calculate the area of shapes like rectangle, triangle, and circle.

#include <stdio.h>

void main() {

int fig\_code;

float side, base, length, breadth, height, area, radius;

printf("-------------------------\n");

printf(" 1 --> Circle\n");

printf(" 2 --> Rectangle\n");

printf(" 3 --> Triangle\n");

printf(" 4 --> Square\n");

printf("-------------------------\n");

printf("Enter the Figure code\n");

scanf("%d", & fig\_code);

switch (fig\_code) {

case 1:

printf("Enter the radius\n");

scanf("%f", & radius);

area = 3.142 \* radius \* radius;

printf("Area of a circle=%f\n", area);

break;

case 2:

printf("Enter the breadth and length\n");

scanf("%f %f", & breadth, & length);

area = breadth \* length;

printf("Area of a Rectangle=%f\n", area);

break;

case 3:

printf("Enter the base and height\n");

scanf("%f %f", & base, & height);

area = 0.5 \* base \* height;

printf("Area of a Triangle=%f\n", area);

break;

case 4:

printf("Enter the side\n");

scanf("%f", & side);

area = side \* side;

printf("Area of a Square=%f\n", area);

break;

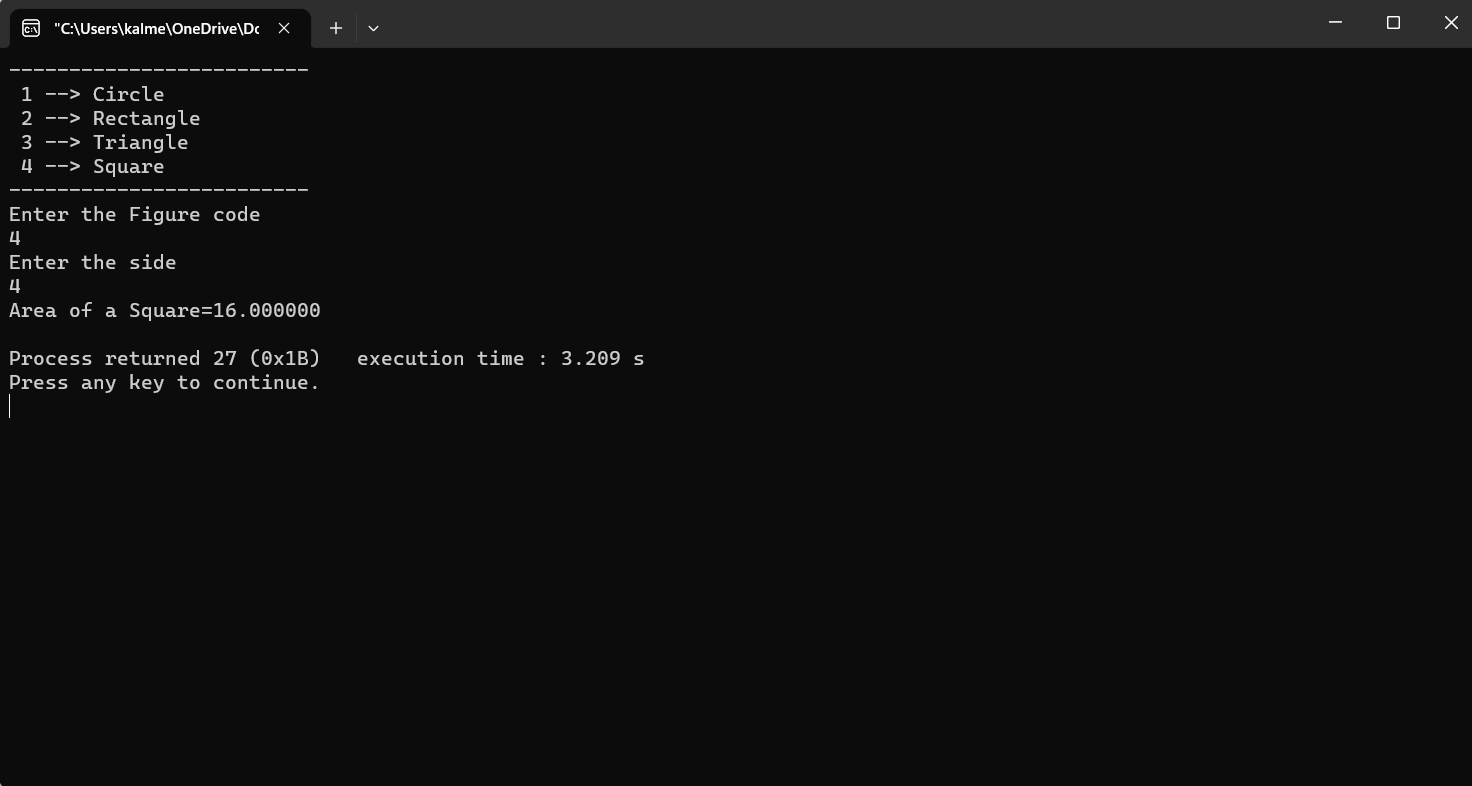
default:

printf("Error in figure code\n");

break;

}

}



9. Simple Calculator: Implement a basic calculator with arithmetic operations.

#include <stdio.h>

#include <stdlib.h>

int main()

{

char ch;

double a, b;

while (1) {

printf("Enter an operator (+, -, \*, /), "

"if want to exit press x: ");

scanf(" %c", &ch);

if (ch == 'x')

exit(0);

printf("Enter two first and second operand: ");

scanf("%lf %lf", &a, &b);

switch (ch) {

case '+':

printf("%.1lf + %.1lf = %.1lf\n", a, b, a + b);

break;

case '-':

printf("%.1lf - %.1lf = %.1lf\n", a, b, a - b);

break;

case '\*':

printf("%.1lf \* %.1lf = %.1lf\n", a, b, a \* b);

break;

case '/':

printf("%.1lf / %.1lf = %.1lf\n", a, b, a / b);

break;

default:

printf(

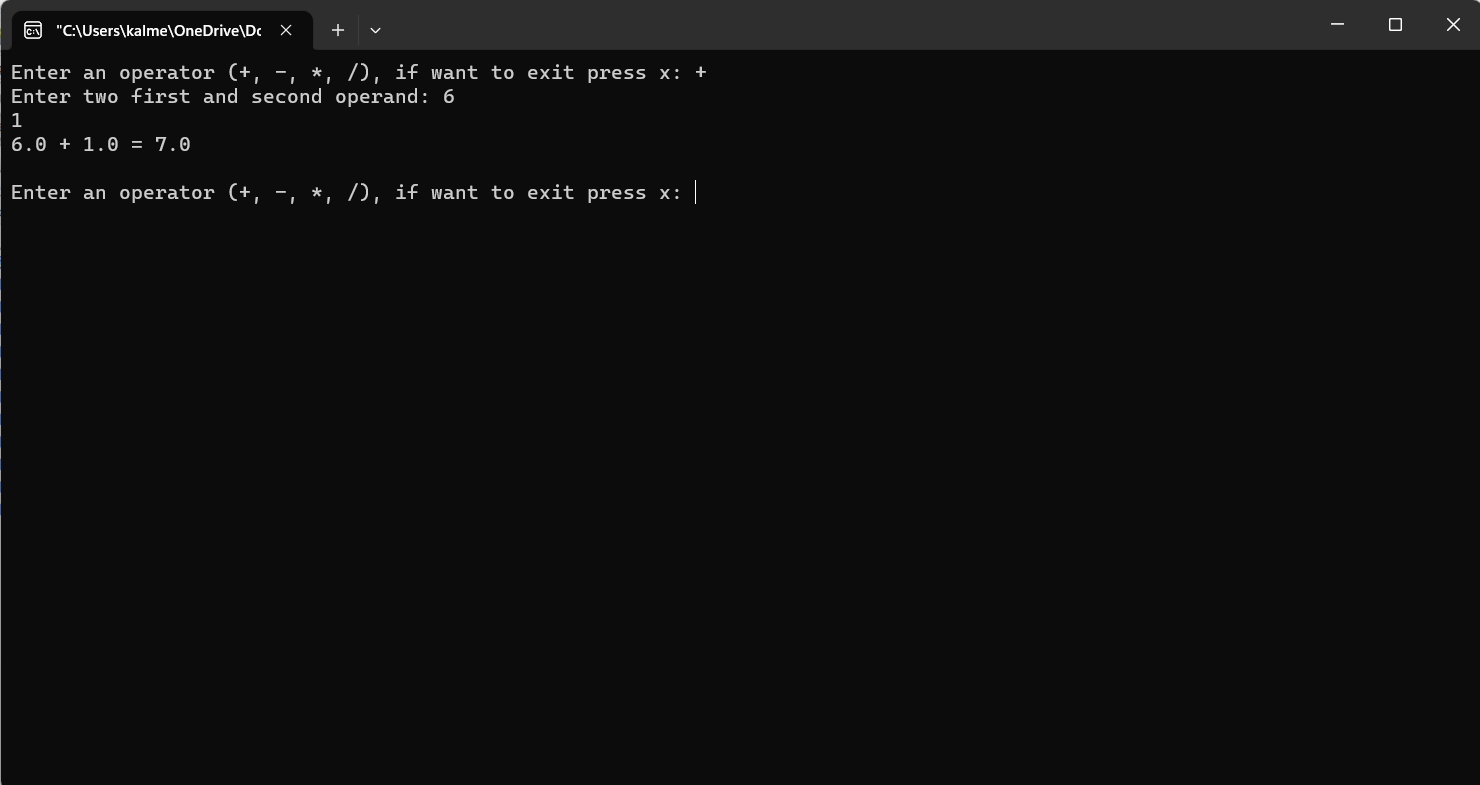
"Error! please write a valid operator\n");

}

printf("\n");

}

}



10. Array Operations: Perform operations like finding the largest/smallest element, sum, and average of an array.

#include <stdio.h>

int min(int a,int b)

{

int min = a;

if(min > b)

min = b;

return min;

}

int max(int a,int b)

{

int max = a;

if(max < b)

max = b;

return max;

}

int getMin(int arr[], int n)

{

int res = arr[0];

for (int i = 1; i < n; i++)

res = min(res, arr[i]);

return res;

}

int getMax(int arr[], int n)

{

int res = arr[0];

for (int i = 1; i < n; i++)

res = max(res, arr[i]);

return res;

}

// Function to get Sum

int findSum(int arr[], int n)

{

int min = getMin(arr, n);

int max = getMax(arr, n);

return min + max;

}

// Function to get product

int findProduct(int arr[], int n)

{

int min = getMin(arr, n);

int max = getMax(arr, n);

return min \* max;

}

// Driver Code

int main()

{

int arr[] = { 12, 1234, 45, 67, 1 };

int n = sizeof(arr) / sizeof(arr[0]);

// Sum of min and max element

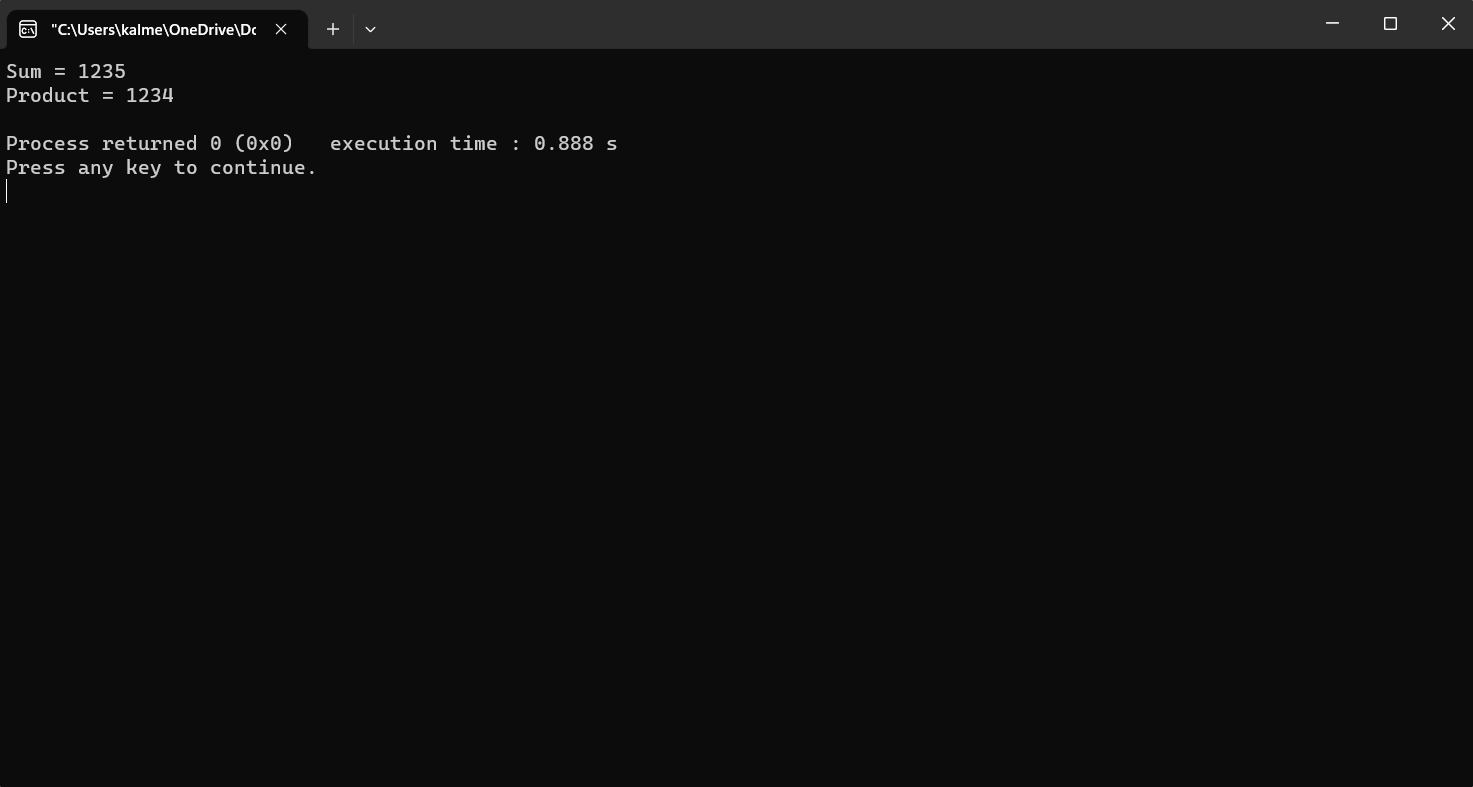
printf("Sum = %d\n",findSum(arr, n));

// Product of min and max element

printf("Product = %d\n",findProduct(arr, n));

return 0;

}



11. String Operations: Manipulate strings such as concatenation, copying, and comparison.

A. Concatenation

#include <stdio.h>

int main()

{

char str1[100] = "Hello", str2[100] = "World";

char str3[100];

int i = 0, j = 0;

printf("\nFirst string: %s", str1);

printf("\nSecond string: %s", str2);

while (str1[i] != '\0') {

str3[j] = str1[i];

i++;

j++;

}

i = 0;

while (str2[i] != '\0') {

str3[j] = str2[i];

i++;

j++;

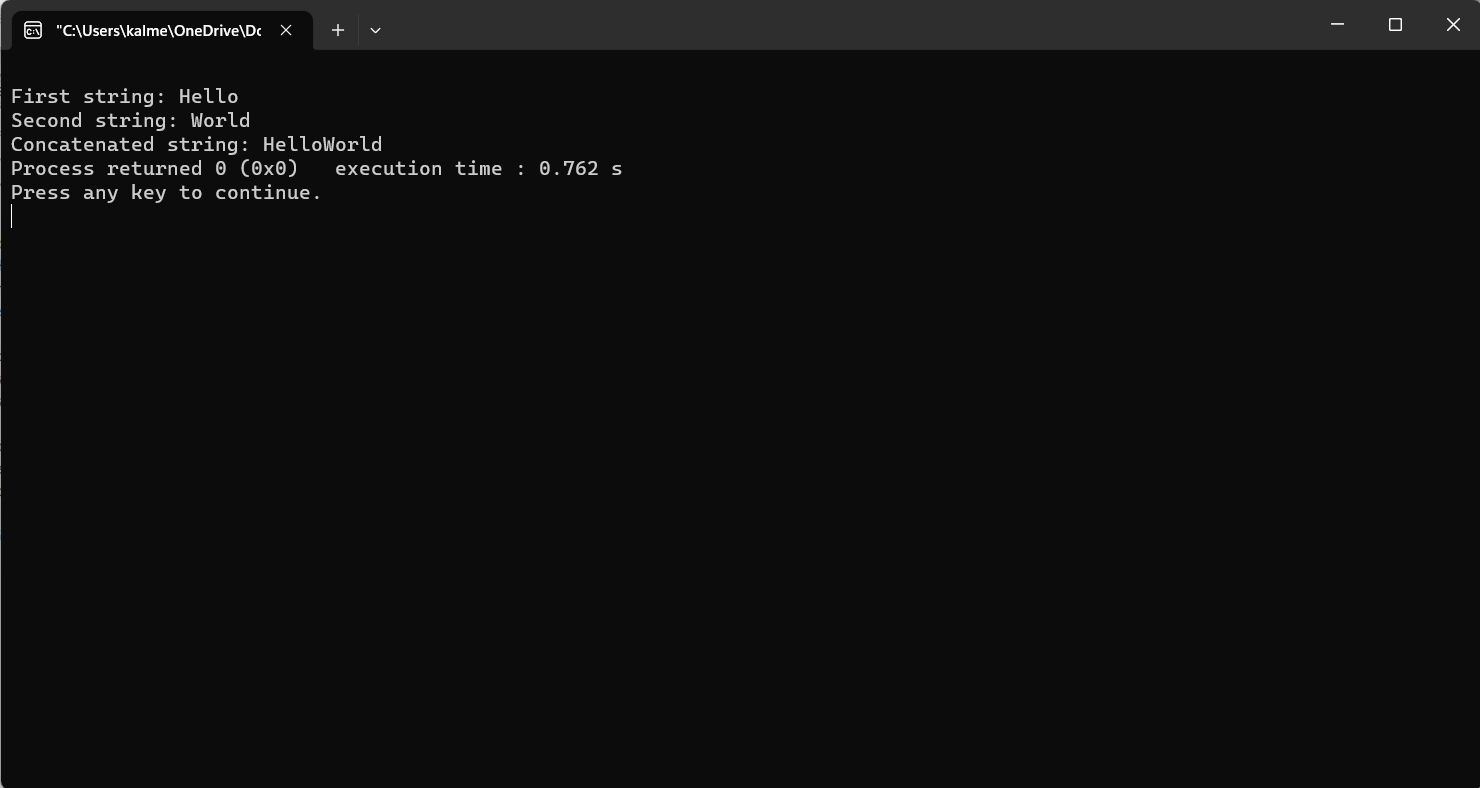
}

str3[j] = '\0';

printf("\nConcatenated string: %s", str3);

return 0;

}



B.Copying

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* copyString(char s[])

{

char\* s2;

s2 = (char\*)malloc(20);

strcpy(s2, s);

return (char\*)s2;

}

int main()

{

char s1[20] = "HelloWorld";

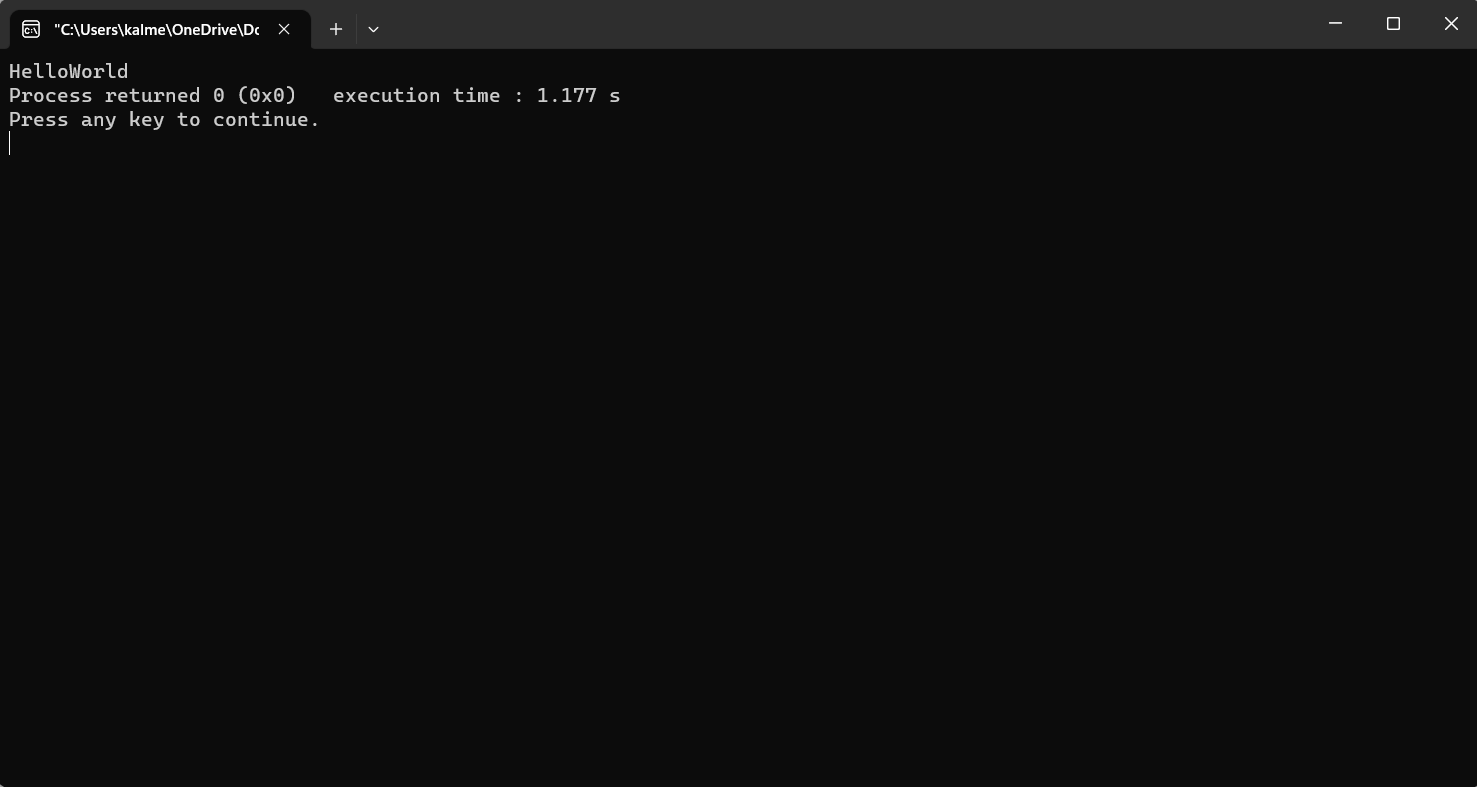
char\* s2;

s2 = copyString(s1);

printf("%s", s2);

return 0;

}



C. Comaprison

#include <stdio.h>

#include <string.h>

int main()

{

char\* first\_str = "Hello";

char\* second\_str = "World";

printf("First String: %s\n", first\_str);

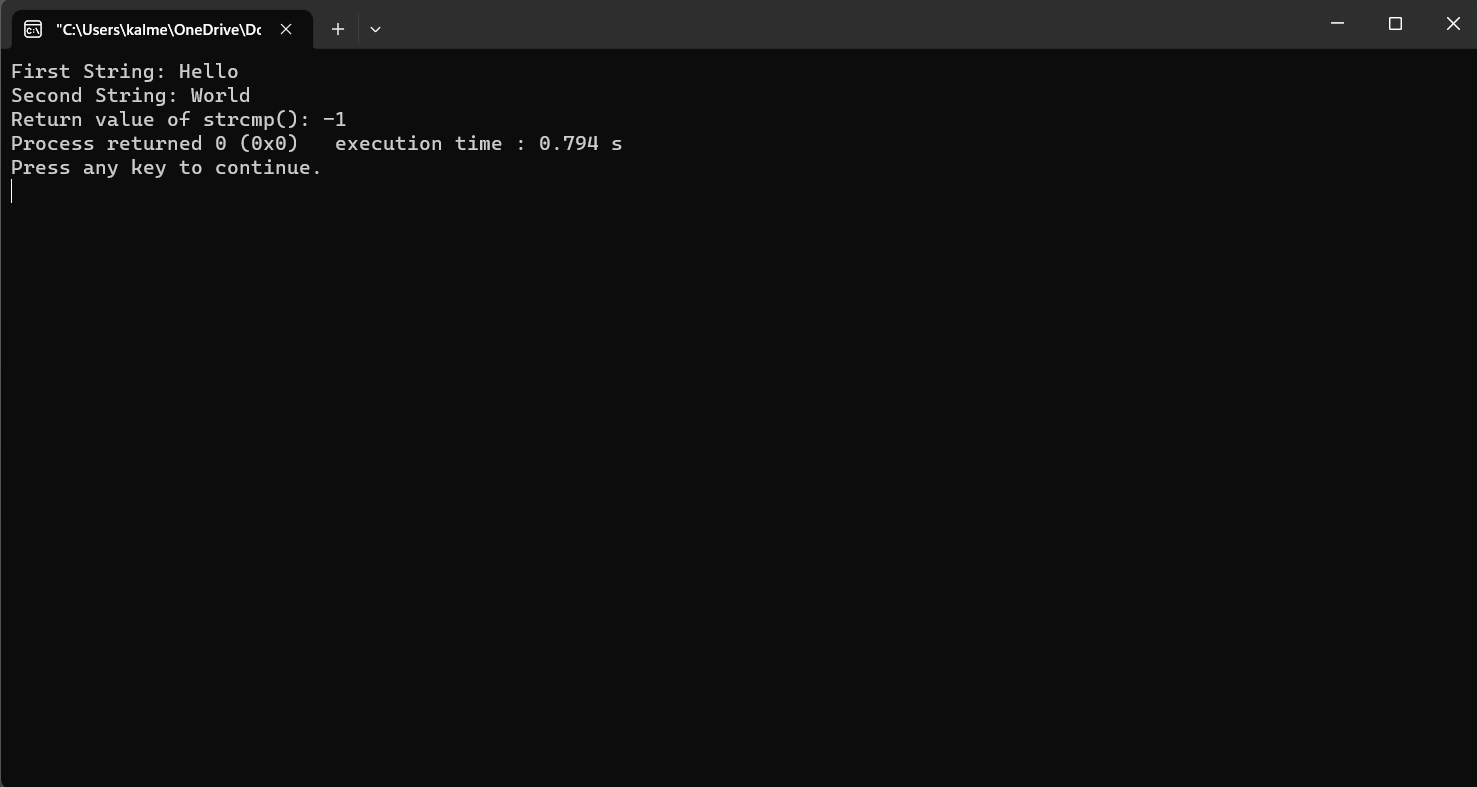
printf("Second String: %s\n", second\_str);

printf("Return value of strcmp(): %d",

strcmp(first\_str, second\_str));

return 0;

}



12. Linear Search: Search for an element in an array using linear search.

#include <stdio.h>

int linearSearch(int\* arr, int size, int key)

{

for (int i = 0; i < size; i++) {

if (arr[i] == key) {

return i;

}

}

return -1;

}

int main()

{

int arr[10] = { 3, 4, 1, 7, 5, 8, 11, 42, 3, 13 };

int size = sizeof(arr) / sizeof(arr[0]);

int key = 4;

int index = linearSearch(arr, size, key);

if (index == -1) {

printf("The element is not present in the arr.");

}

else {

printf("The element is present at arr[%d].", index);

}

return 0;

}



13. Binary Search: Search for an element in a sorted array using binary search.

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {

// Repeat until the pointers low and high meet each other

while (low <= high) {

int mid = low + (high - low) / 2;

if (array[mid] == x)

return mid;

if (array[mid] < x)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main(void) {

int array[] = {3, 4, 5, 6, 7, 8, 9};

int n = sizeof(array) / sizeof(array[0]);

int x = 4;

int result = binarySearch(array, x, 0, n - 1);

if (result == -1)

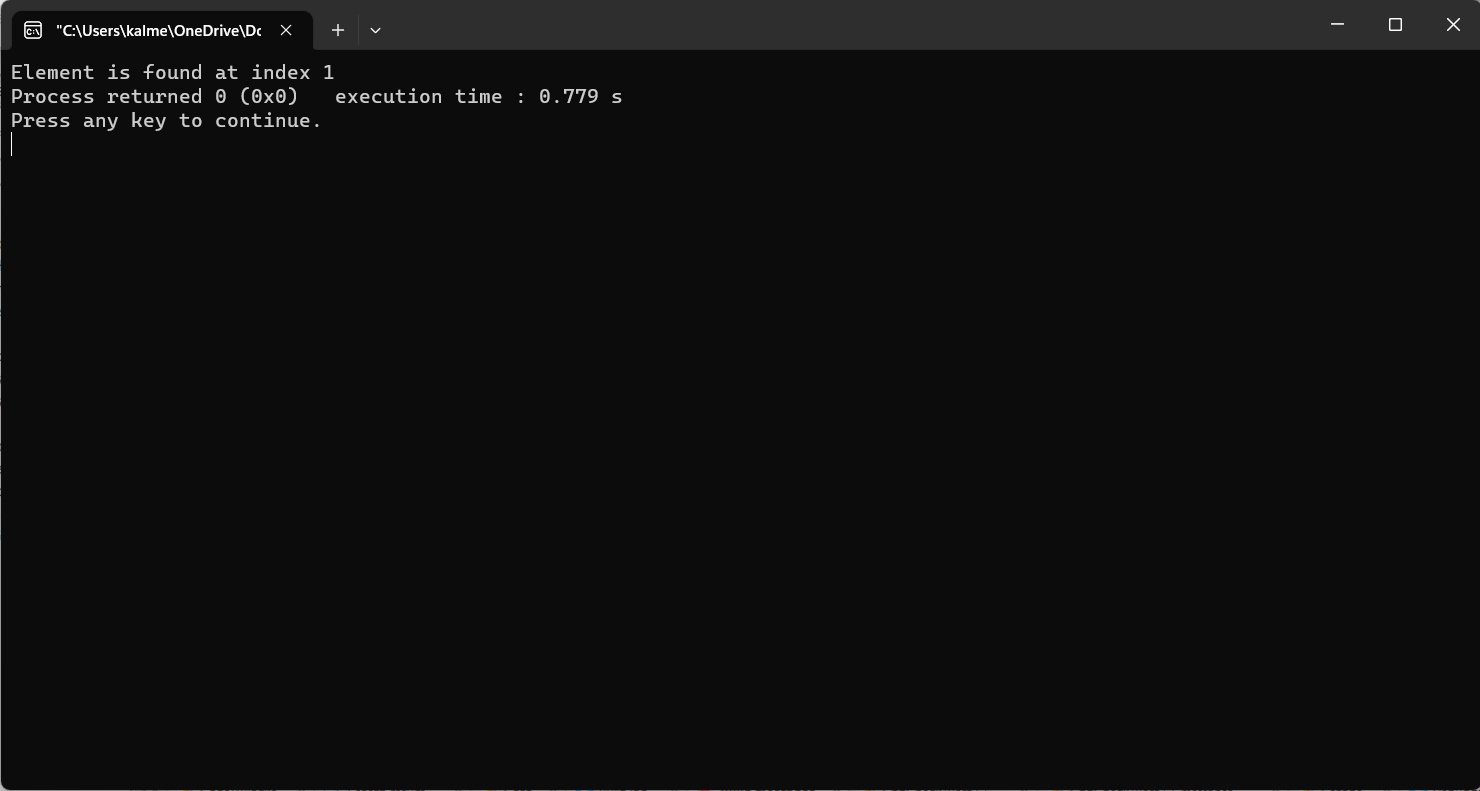
printf("Not found");

else

printf("Element is found at index %d", result);

return 0;

}



14. Selection Sort: Sort an array using the selection sort algorithm.

#include<stdio.h>

int main(){

int a[10],i;

int j,temp,num;

printf("Enter the number to give\n");

scanf("%d",&num);

for(i=0; i<num; i++){

printf("a[%d]=\t",i);

scanf("%d",&a[i]);

}

for(i=0; i<num-1; i++){

for(j=i+1;j<num; j++){

if(a[i]>a[j]){

temp=a[i];

a[i]=a[j];

a[j]=temp;

}

}

}

printf("Selection Sort in C\n");

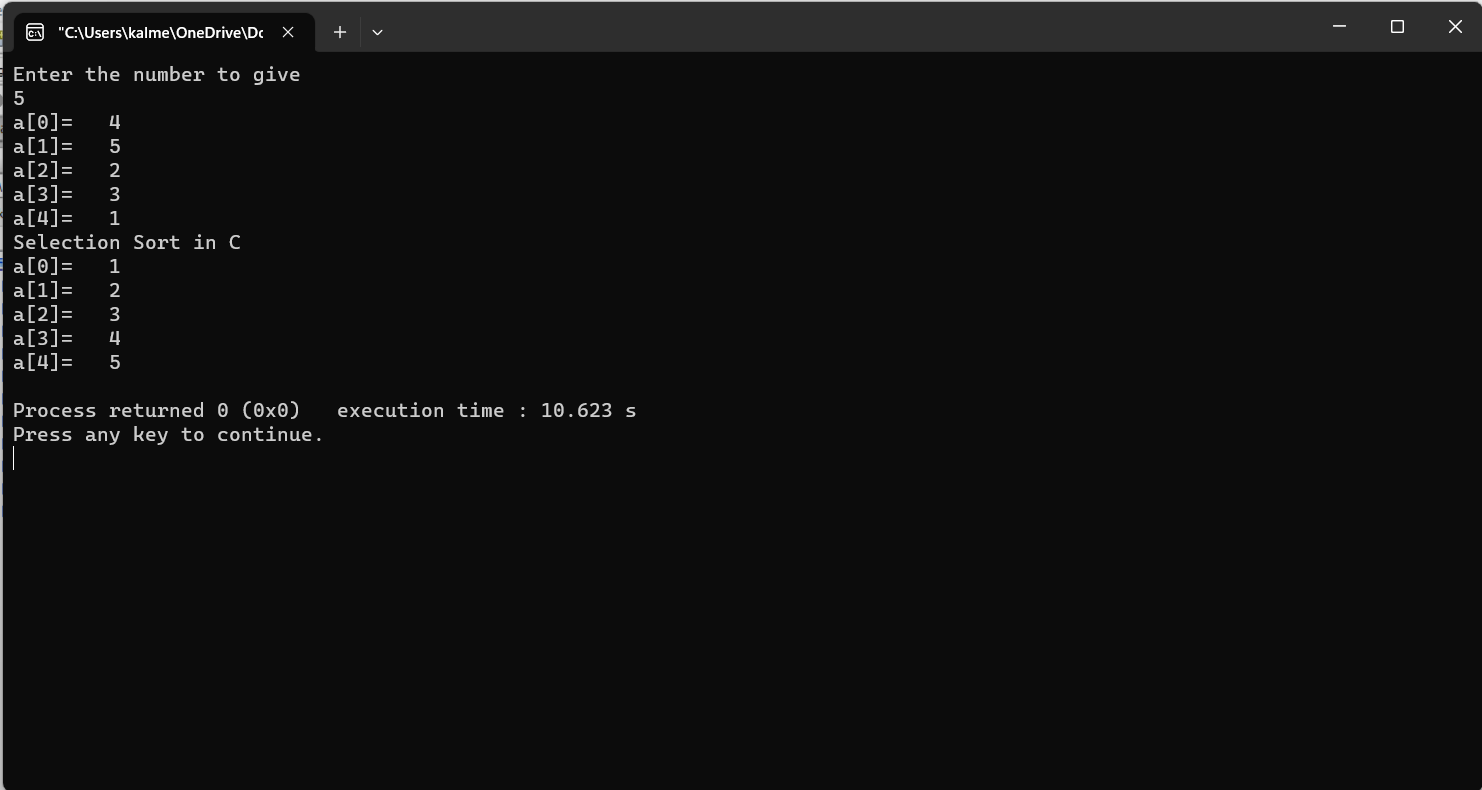
for(i=0; i<num; i++){

printf("a[%d]=\t%d\n",i,a[i]);

}

return 0;

}



15. Bubble Sort: Sort an array using the bubble sort algorithm.

#include <stdio.h>

void bubbleSort(int array[], int size) {

for (int step = 0; step < size - 1; ++step) {

for (int i = 0; i < size - step - 1; ++i) {

if (array[i] > array[i + 1]) {

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

}

}

}

}

void printArray(int array[], int size) {

for (int i = 0; i < size; ++i) {

printf("%d ", array[i]);

}

printf("\n");

}

int main() {

int data[] = {-2, 45, 0, 11, -9};

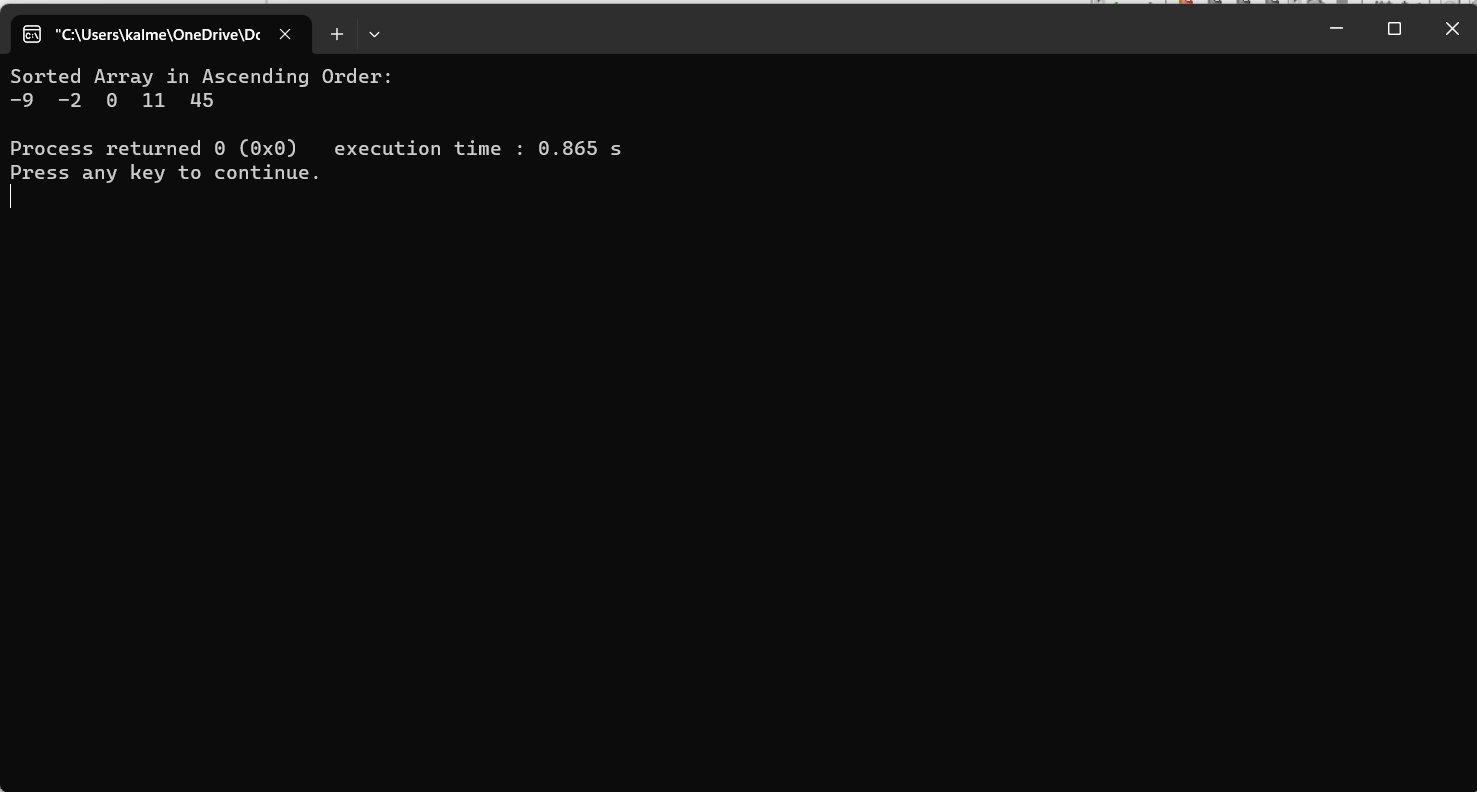
int size = sizeof(data) / sizeof(data[0]);

bubbleSort(data, size);

printf("Sorted Array in Ascending Order:\n");

printArray(data, size);

}



16. Insertion Sort: Sort an array using the insertion sort algorithm.

#include <math.h>

#include <stdio.h>

void insertionSort(int arr[], int n)

{

int i, key, j;

for (i = 1; i < n; 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;

}

}

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};

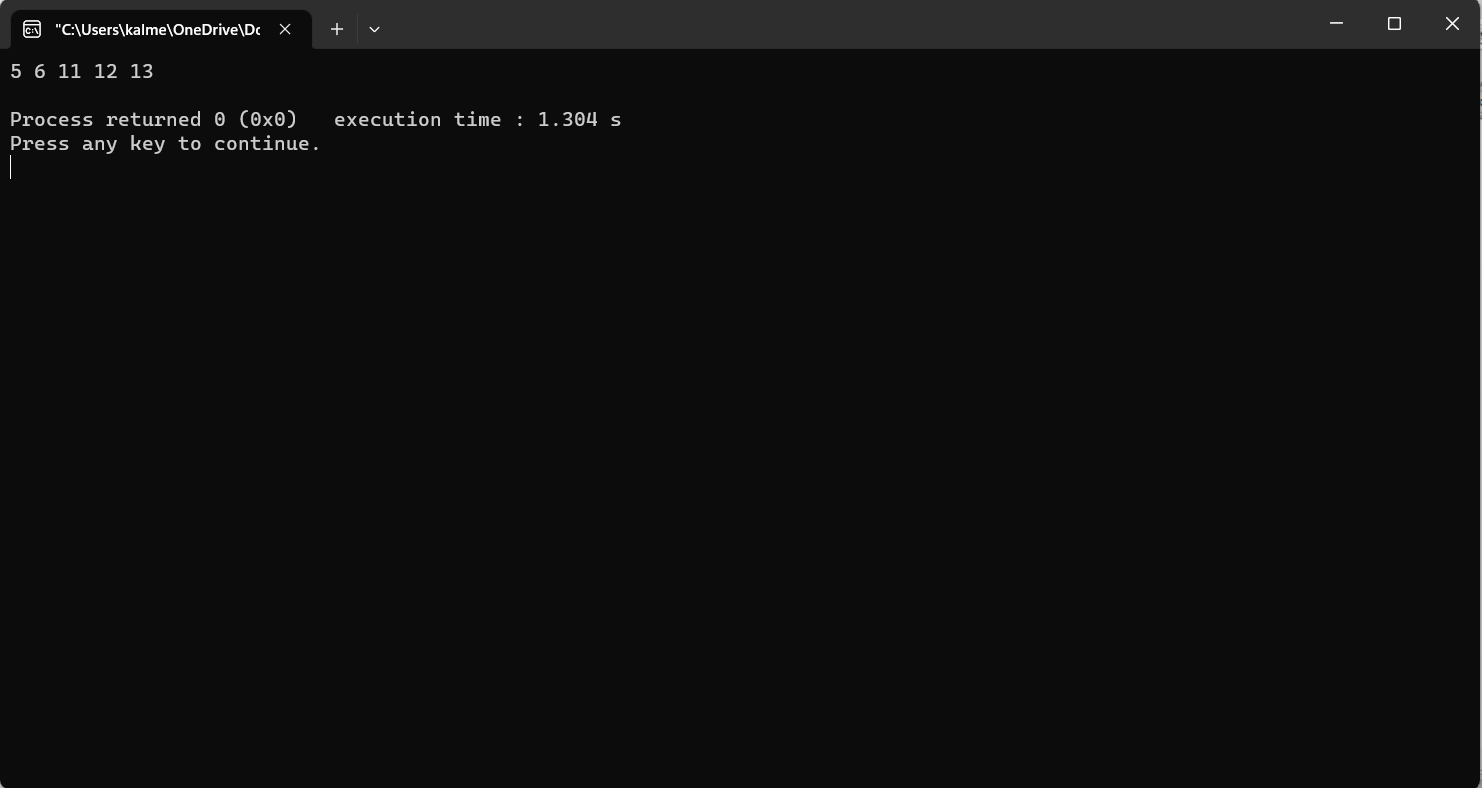
int n = sizeof(arr) / sizeof(arr[0]);

insertionSort(arr, n);

printArray(arr, n);

return 0;

}



17. Matrix Operations: Perform matrix addition, subtraction, multiplication, and transpose.

#include<stdio.h>

#include<stdlib.h>

void add(int m[3][3], int n[3][3], int sum[3][3])

{

for(int i=0;i<3;i++)

for(int j=0;j<3;j++)

sum[i][j] = m[i][j] + n[i][j];

}

void subtract(int m[3][3], int n[3][3], int result[3][3])

{

for(int i=0;i<3;i++)

for(int j=0;j<3;j++)

result[i][j] = m[i][j] - n[i][j];

}

void multiply(int m[3][3], int n[3][3], int result[3][3])

{

for(int i=0; i < 3; i++)

{

for(int j=0; j < 3; j++)

{

result[i][j] = 0;

for (int k = 0; k < 3; k++)

result[i][j] += m[i][k] \* n[k][j];

}

}

}

void transpose(int matrix[3][3], int trans[3][3])

{

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

trans[i][j] = matrix[j][i];

}

void display(int matrix[3][3])

{

for(int i=0; i<3; i++)

{

for(int j=0; j<3; j++)

printf("%d\t",matrix[i][j]);

printf("\n");

}

}

int main()

{

int a[][3] = { {5,6,7}, {8,9,10}, {3,1,2} };

int b[][3] = { {1,2,3}, {4,5,6}, {7,8,9} };

int c[3][3];

printf("First Matrix:\n");

display(a);

printf("Second Matrix:\n");

display(b);

int choice;

do

{

printf("\nChoose the matrix operation,\n");

printf("----------------------------\n");

printf("1. Addition\n");

printf("2. Subtraction\n");

printf("3. Multiplication\n");

printf("4. Transpose\n");

printf("5. Exit\n");

printf("----------------------------\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

add(a, b, c);

printf("Sum of matrix: \n");

display(c);

break;

case 2:

subtract(a, b, c);

printf("Subtraction of matrix: \n");

display(c);

break;

case 3:

multiply(a, b, c);

printf("Multiplication of matrix: \n");

display(c);

break;

case 4:

printf("Transpose of the first matrix: \n");

transpose(a, c);

display(c);

printf("Transpose of the second matrix: \n");

transpose(b, c);

display(c);

break;

case 5:

printf("Thank You.\n");

exit(0);

default:

printf("Invalid input.\n");

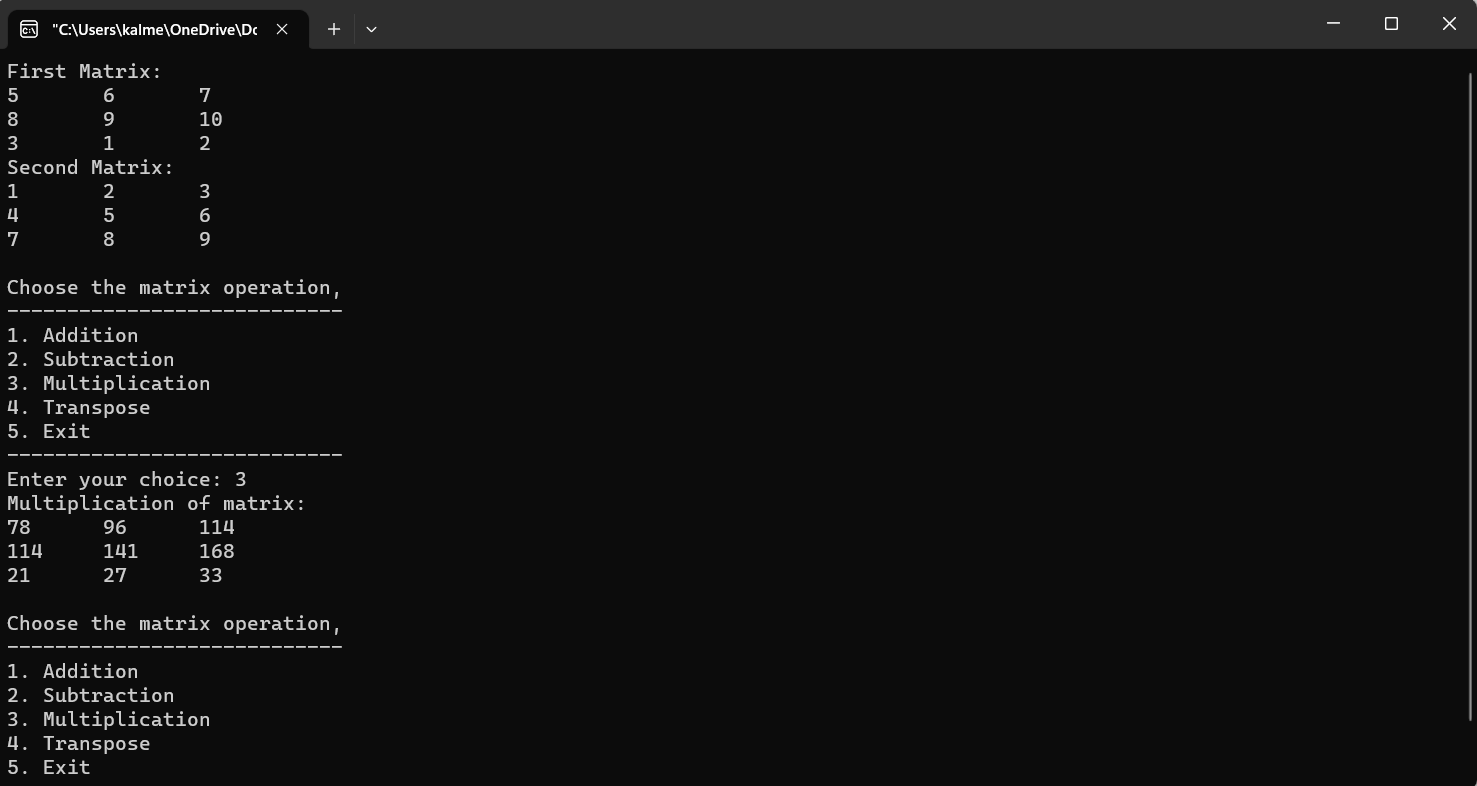
printf("Please enter the correct input.\n");

}

}while(1);

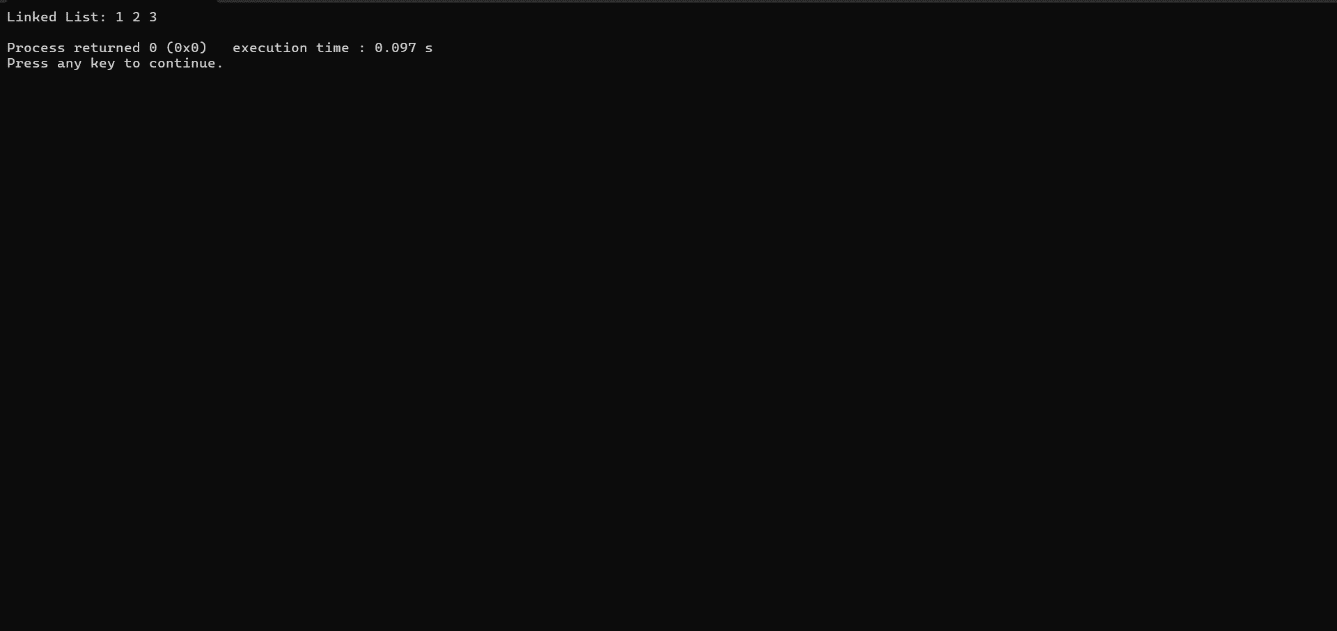
return 0;

}



**Part- C**

|  |
| --- |
| **Linkedlists** |
| 1. **creating and travsering of linked**  CODE:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void printLinkedList(struct Node\* head) {  struct Node\* temp = head;  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;  struct Node\* second = NULL;  struct Node\* third = NULL;  head = (struct Node\*)malloc(sizeof(struct Node));  second = (struct Node\*)malloc(sizeof(struct Node));  third = (struct Node\*)malloc(sizeof(struct Node));  head->data = 1;  head->next = second;  second->data = 2;  second->next = third;  third->data = 3;  third->next = NULL;  printf("Linked List: ");  printLinkedList(head);  free(head);  free(second);  free(third);  return 0;  } |

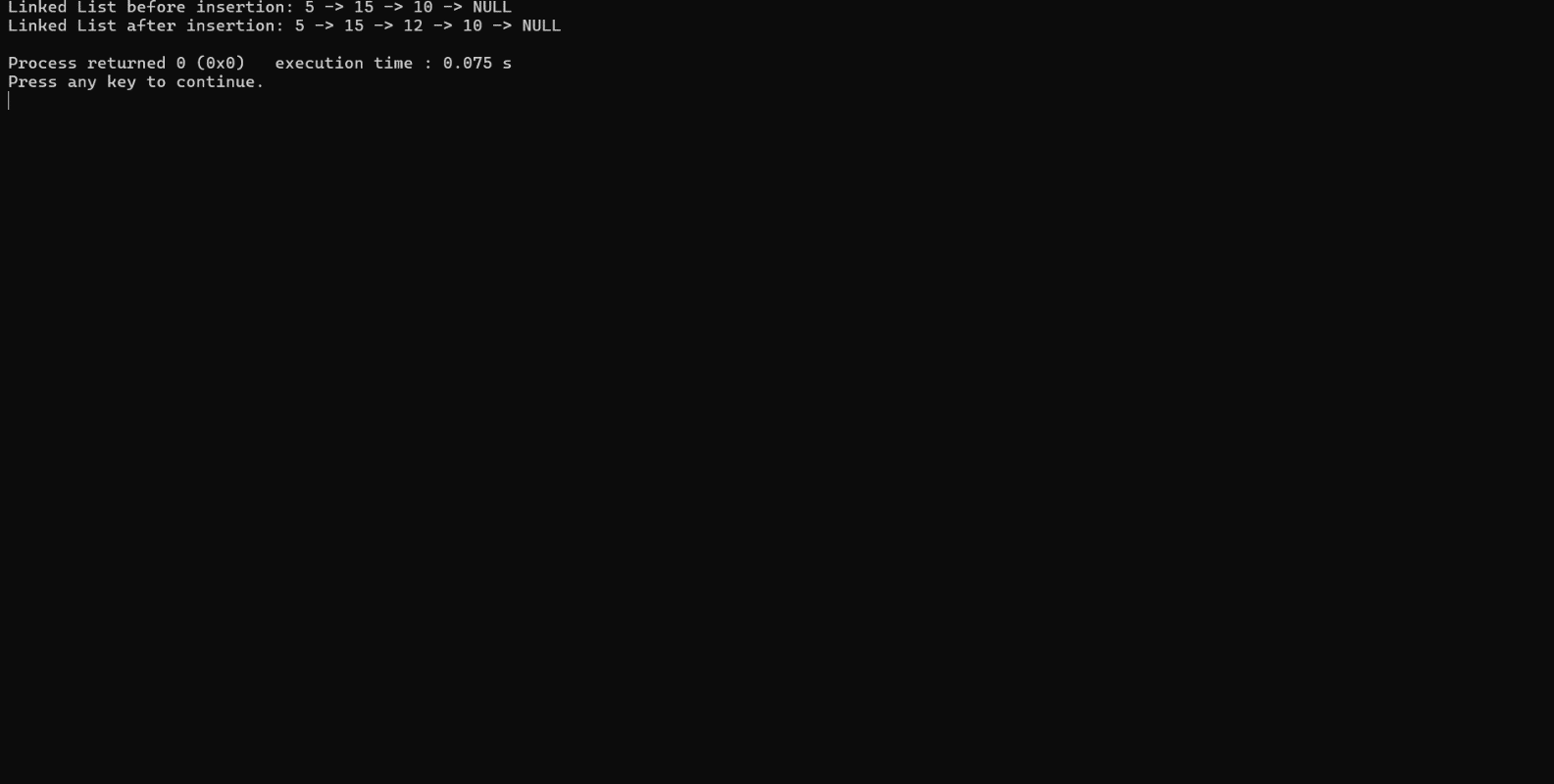


|  |
| --- |
| 2.INSERT AT BEGINNING  Code:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));  new\_node->data = new\_data;  new\_node->next = \*head\_ref;  \*head\_ref = new\_node;  }  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  insertAtBeginning(&head, 5);  insertAtBeginning(&head, 10);  insertAtBeginning(&head, 15);  printf("Linked List: ");  printList(head);  return 0;  }  OUTPUT: |

|  |
| --- |
| 3.INSERTING AT END  CODE:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtBeginning(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;  }  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  insertAtBeginning(&head, 5);  insertAtBeginning(&head, 10);  insertAtBeginning(&head, 15);  printf("Linked List: ");  printList(head);  return 0;  }  OUTPUT: |

|  |
| --- |
| 3.INSERT AT ANY POISTIOIN:  CODE:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtPosition(struct Node\*\* head\_ref, int new\_data, int position) {  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));  new\_node->data = new\_data;  if (position == 0) {  new\_node->next = \*head\_ref;  \*head\_ref = new\_node;  return;  }  struct Node\* current = \*head\_ref;  for (int i = 0; i < position - 1 && current != NULL; i++) {  current = current->next;  }  if (current == NULL) {  printf("Invalid position!\n");  return;  }  new\_node->next = current->next;  current->next = new\_node;  }  void printList(struct Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  insertAtPosition(&head, 5, 0); // Insert at position 0 (beginning)  insertAtPosition(&head, 10, 1); // Insert at position 1  insertAtPosition(&head, 15, 1); // Insert at position 1  printf("Linked List before insertion: ");  printList(head);  insertAtPosition(&head, 12, 2); // Insert at position 2  printf("Linked List after insertion: ");  printList(head);  return 0;  } |

OUTPUT:



4.DELETE FROM BEGINNING:

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtBeginning(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;

}

void deleteFromBeginning(struct Node\*\* head\_ref) {

if (\*head\_ref == NULL)

return;

struct Node\* temp = \*head\_ref;

\*head\_ref = (\*head\_ref)->next;

free(temp);

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 5);

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 15);

printf("Linked List before deletion: ");

printList(head);

deleteFromBeginning(&head);

printf("Linked List after deletion: ");

printList(head);

return 0;

}



5.DELETE FROM END:

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtEnd(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 = NULL;

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

struct Node\* last = \*head\_ref;

while (last->next != NULL)

last = last->next;

last->next = new\_node;

}

void deleteFromEnd(struct Node\*\* head\_ref) {

// If the linked list is empty, do nothing

if (\*head\_ref == NULL)

return;

if ((\*head\_ref)->next == NULL) {

free(\*head\_ref);

\*head\_ref = NULL;

return;

}

struct Node\* second\_last = \*head\_ref;

while (second\_last->next->next != NULL)

second\_last = second\_last->next;

free(second\_last->next);

second\_last->next = NULL;

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtEnd(&head, 5);

insertAtEnd(&head, 10);

insertAtEnd(&head, 15);

printf("Linked List before deletion: ");

printList(head);

deleteFromEnd(&head);

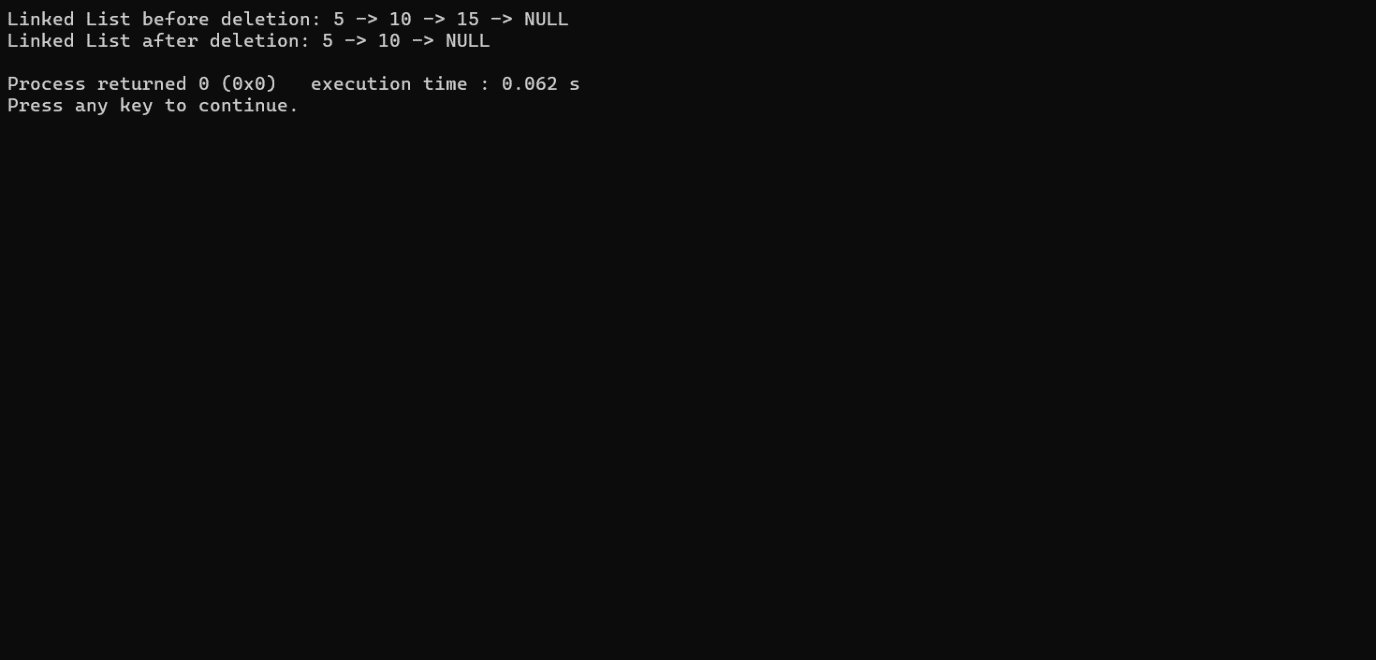
printf("Linked List after deletion: ");

printList(head);

return 0;

}

OUTPUT:



6.DELETE AT POSTION:

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtBeginning(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;

}

void deleteAtPosition(struct Node\*\* head\_ref, int position) {

if (\*head\_ref == NULL)

return;

struct Node\* temp = \*head\_ref;

if (position == 0) {

\*head\_ref = temp->next;

free(temp);

return;

}

for (int i = 0; temp != NULL && i < position - 1; i++)

temp = temp->next;

if (temp == NULL || temp->next == NULL) {

printf("Invalid position!\n");

return;

}

struct Node\* next\_node = temp->next->next;

free(temp->next);

temp->next = next\_node;

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 5);

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 15);

printf("Linked List before deletion: ");

printList(head);

deleteAtPosition(&head, 1);

printf("Linked List after deletion: ");

printList(head);

return 0;

}

OUTPUT:

7:REVERSE LINKED LIST

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtBeginning(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;

}

void reverseLinkedList(struct Node\*\* head\_ref) {

struct Node\* prev = NULL;

struct Node\* current = \*head\_ref;

struct Node\* next = NULL;

while (current != NULL) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

\*head\_ref = prev;

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 5);

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 15);

printf("Original Linked List: ");

printList(head);

reverseLinkedList(&head);

printf("Reversed Linked List: ");

printList(head);

return 0;

}

OUTPUT:



8 MERGING OF TWO LINKED LISTS

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtEnd(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 = NULL;

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

struct Node\* last = \*head\_ref;

while (last->next != NULL)

last = last->next;

last->next = new\_node;

}

struct Node\* mergeSortedLists(struct Node\* list1, struct Node\* list2) {

if (list1 == NULL)

return list2;

if (list2 == NULL)

return list1;

struct Node\* result = NULL;

if (list1->data <= list2->data) {

result = list1;

result->next = mergeSortedLists(list1->next, list2);

} else {

result = list2;

result->next = mergeSortedLists(list1, list2->next);

}

return result;

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* list1 = NULL;

struct Node\* list2 = NULL;

insertAtEnd(&list1, 1);

insertAtEnd(&list1, 3);

insertAtEnd(&list1, 5);

insertAtEnd(&list2, 2);

insertAtEnd(&list2, 4);

insertAtEnd(&list2, 6);

printf("List 1: ");

printList(list1);

printf("List 2: ");

printList(list2);

struct Node\* mergedList = mergeSortedLists(list1, list2);

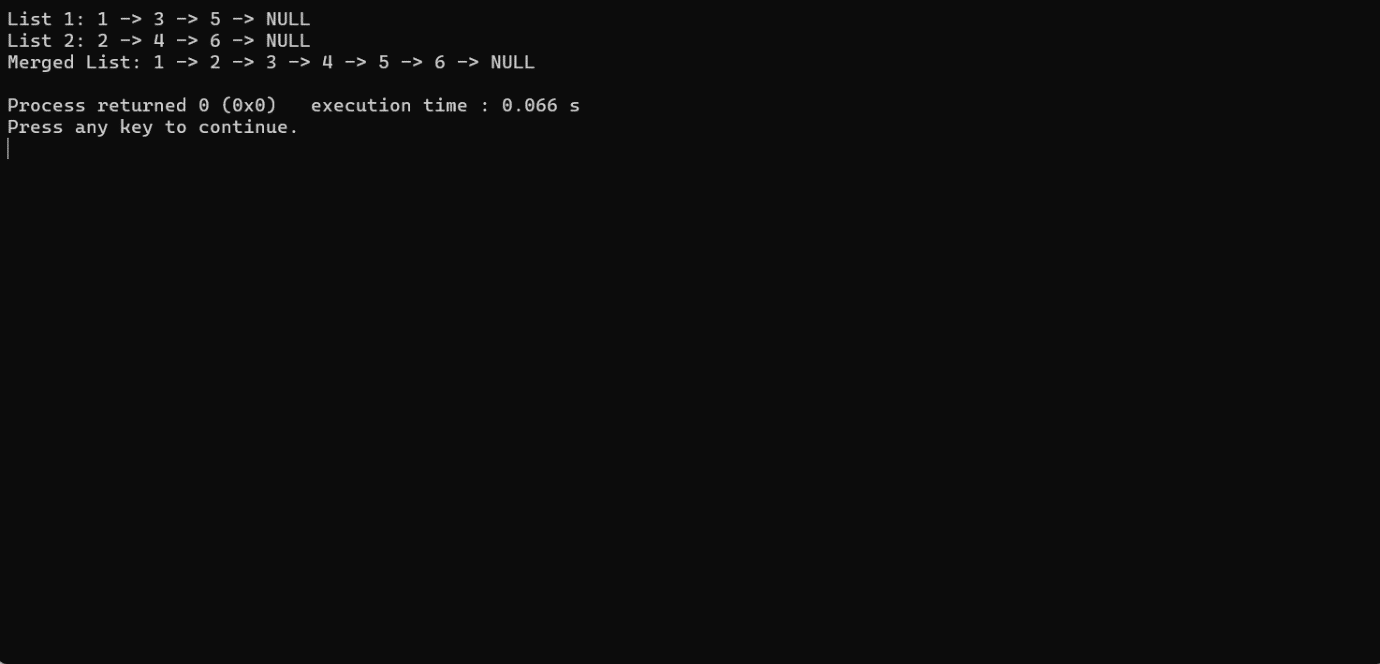
printf("Merged List: ");

printList(mergedList);

return 0;

}

OUTPUT:



8:INSERT AT BEGINNING DOUBLY LINKED LIST:

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {

// Create a new node

struct Node\* new\_node = createNode(new\_data);

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

(\*head\_ref)->prev = new\_node;

new\_node->next = \*head\_ref;

\*head\_ref = new\_node;

}

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d <-> ", node->data);

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 5);

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 15);

printf("Doubly Linked List after insertion at beginning: ");

printList(head);

return 0;

}

OUTPUT:



9:INSERT AT THE END DOUBLY LINKED LIST

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = createNode(new\_data);

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

struct Node\* last = \*head\_ref;

while (last->next != NULL)

last = last->next;

last->next = new\_node;

new\_node->prev = last;

}

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d <-> ", node->data);

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

// Insert some nodes at the end

insertAtEnd(&head, 5);

insertAtEnd(&head, 10);

insertAtEnd(&head, 15);

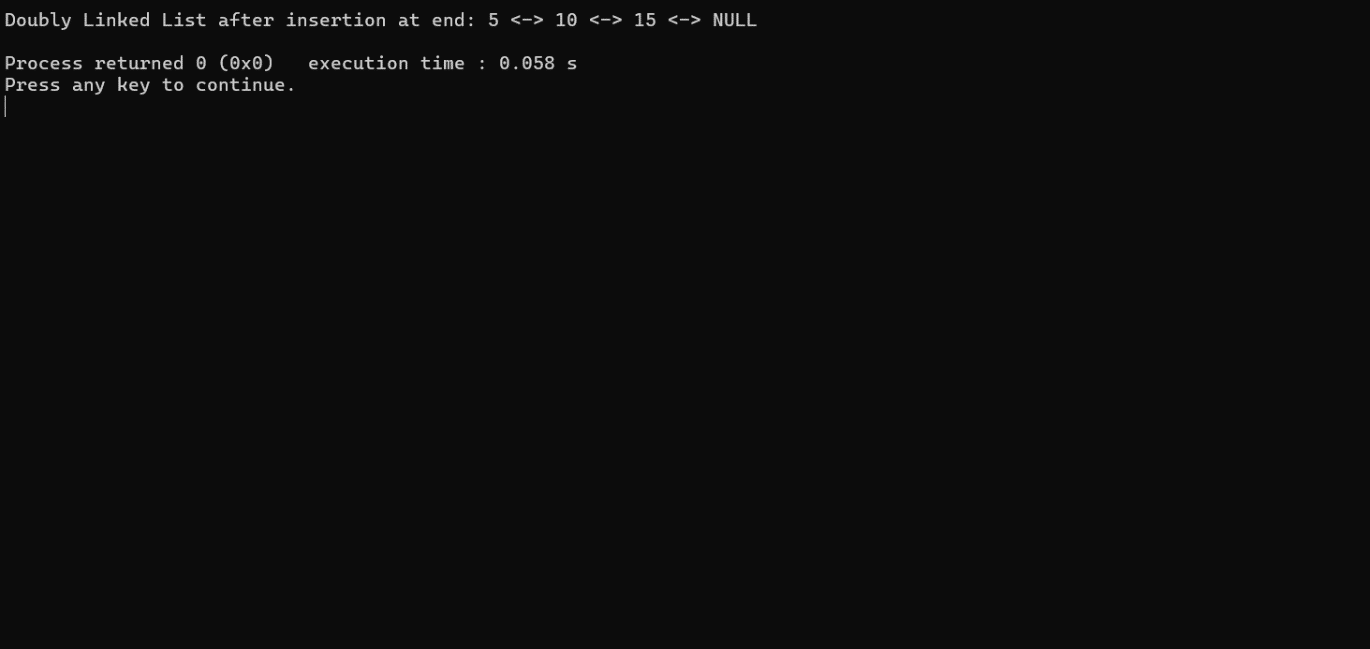
printf("Doubly Linked List after insertion at end: ");

printList(head);

return 0;

}

OUTPUT:



10.INSERT AT ANY POSITION:

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

void insertAtPosition(struct Node\*\* head\_ref, int new\_data, int position) {

struct Node\* new\_node = createNode(new\_data);

if (\*head\_ref == NULL || position <= 0) {

new\_node->next = \*head\_ref;

if (\*head\_ref != NULL)

(\*head\_ref)->prev = new\_node;

\*head\_ref = new\_node;

return;

}

struct Node\* current = \*head\_ref;

for (int i = 0; current != NULL && i < position - 1; i++) {

current = current->next;

}

if (current == NULL) {

while ((\*head\_ref)->next != NULL)

(\*head\_ref) = (\*head\_ref)->next;

(\*head\_ref)->next = new\_node;

new\_node->prev = \*head\_ref;

return;

}

new\_node->next = current->next;

new\_node->prev = current;

if (current->next != NULL)

current->next->prev = new\_node;

current->next = new\_node;

}

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d <-> ", node->data);

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtPosition(&head, 5, 0); // Insert at position 0 (beginning)

insertAtPosition(&head, 10, 1); // Insert at position 1

insertAtPosition(&head, 15, 1); // Insert at position 1

insertAtPosition(&head, 20, 3); // Insert at position 3

insertAtPosition(&head, 25, 5); // Insert at position 5 (end)

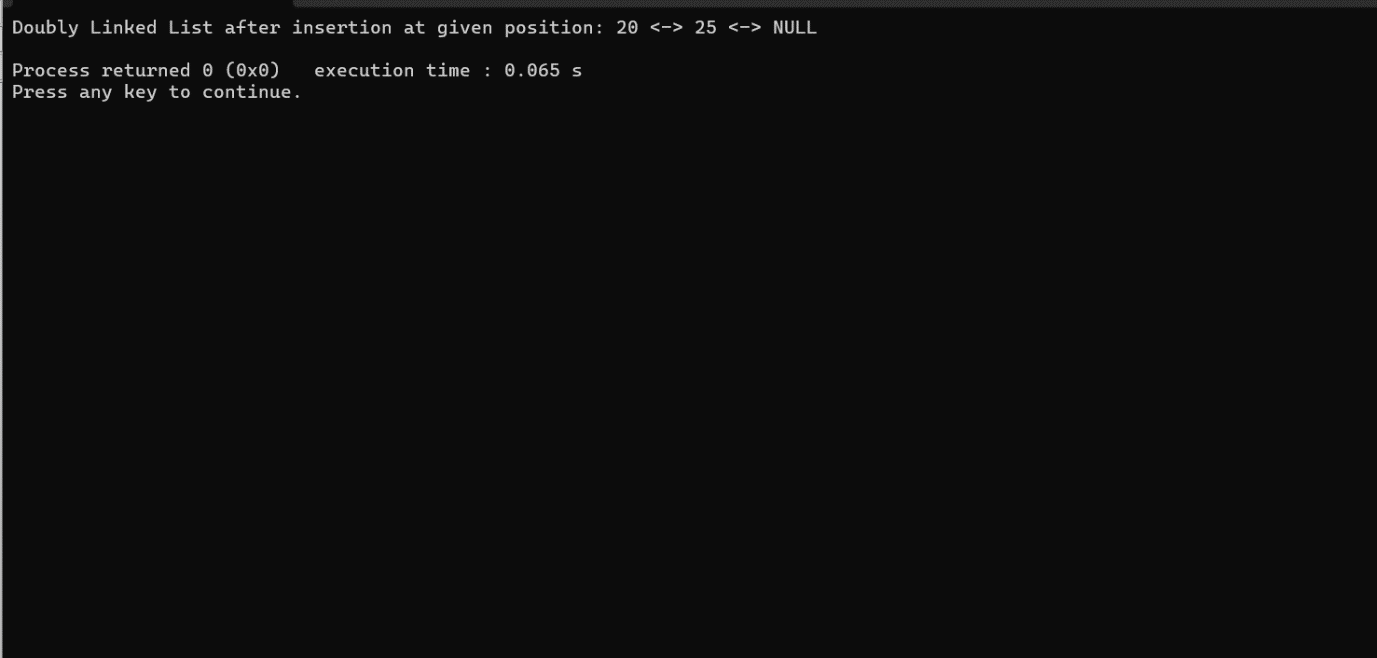
printf("Doubly Linked List after insertion at given position: ");

printList(head);

return 0;

}

OUTPUT:



11.DELETE FROM BEGINNING

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = createNode(new\_data);

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

struct Node\* last = \*head\_ref;

while (last->next != NULL)

last = last->next;

last->next = new\_node;

new\_node->prev = last;

}

void deleteFromBeginning(struct Node\*\* head\_ref) {

if (\*head\_ref == NULL)

return;

struct Node\* temp = \*head\_ref;

\*head\_ref = (\*head\_ref)->next;

if (\*head\_ref != NULL)

(\*head\_ref)->prev = NULL;

free(temp);

}

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d <-> ", node->data);

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtEnd(&head, 5);

insertAtEnd(&head, 10);

insertAtEnd(&head, 15);

printf("Doubly Linked List before deletion from beginning: ");

printList(head);

deleteFromBeginning(&head);

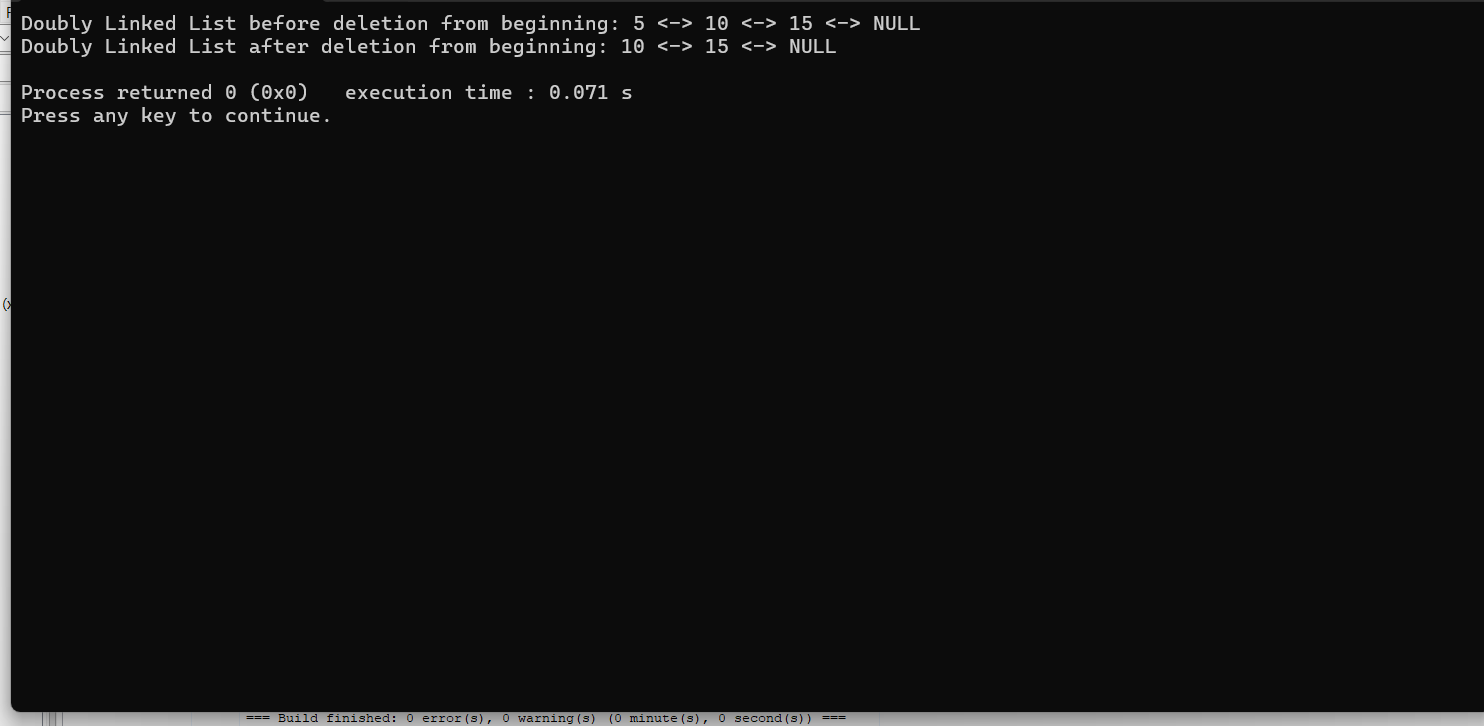
printf("Doubly Linked List after deletion from beginning: ");

printList(head);

return 0;

}

OUTPUT:



12 .DELETE AT POSTION DOUBLY LINKED LIST

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtBeginning(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;

}

void deleteAtPosition(struct Node\*\* head\_ref, int position) {

if (\*head\_ref == NULL)

return;

struct Node\* temp = \*head\_ref;

if (position == 0) {

\*head\_ref = temp->next;

free(temp);

return;

}

for (int i = 0; temp != NULL && i < position - 1; i++)

temp = temp->next;

if (temp == NULL || temp->next == NULL) {

printf("Invalid position!\n");

return;

}

struct Node\* next\_node = temp->next->next;

free(temp->next);

temp->next = next\_node;

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 5);

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 15);

printf("Linked List before deletion: ");

printList(head);

deleteAtPosition(&head, 1);

printf("Linked List after deletion: ");

printList(head);

return 0;

}

OUTPUT:



13.INSERT AT BEGINNING CIRCULAR LINKED LIST

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void insertAtBeginning(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

struct Node\* last = \*head;

while (last->next != \*head)

last = last->next;

newNode->next = \*head;

last->next = newNode;

\*head = newNode;

}

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 4);

insertAtBeginning(&head, 3);

insertAtBeginning(&head, 2);

insertAtBeginning(&head, 1);

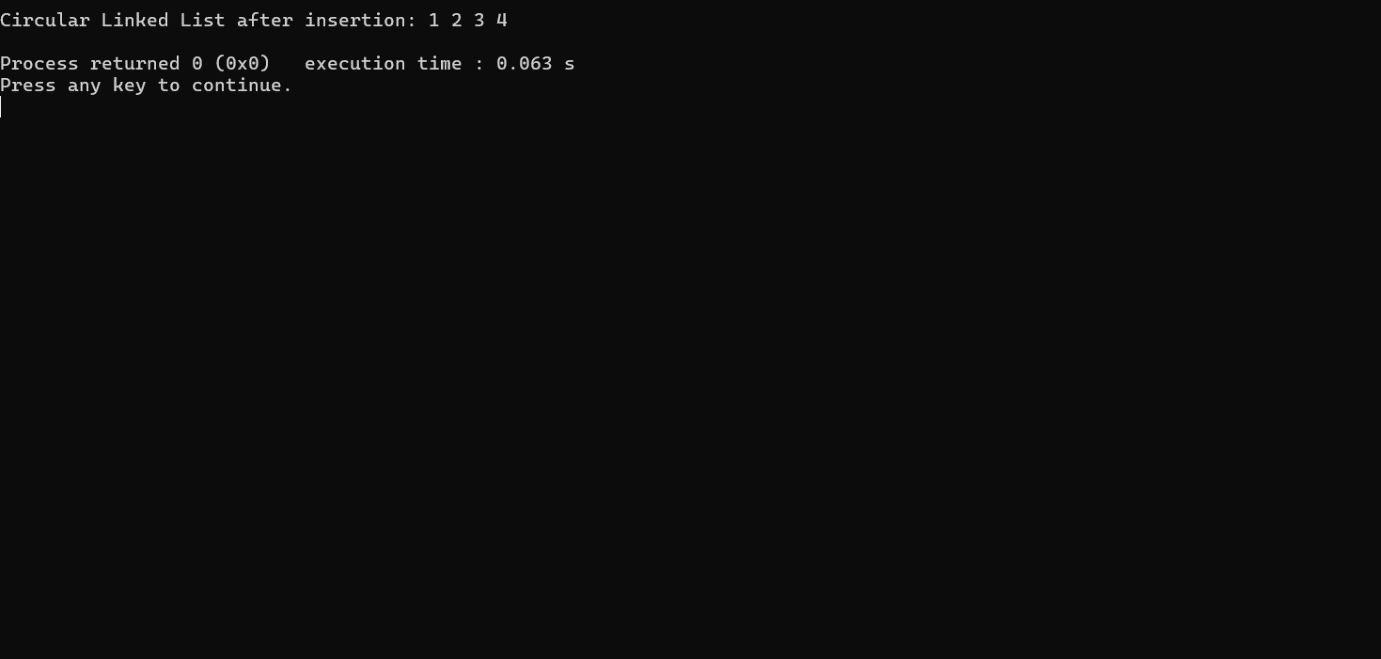
printf("Circular Linked List after insertion: ");

printList(head);

return 0;

}

OUTPUT:



14. INSERT AT END CIRCULAR LINKED LIST

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void insertAtEnd(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

struct Node\* last = \*head;

while (last->next != \*head)

last = last->next;

last->next = newNode;

newNode->next = \*head;

}

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = NULL;

insertAtEnd(&head, 1);

insertAtEnd(&head, 2);

insertAtEnd(&head, 3);

insertAtEnd(&head, 4);

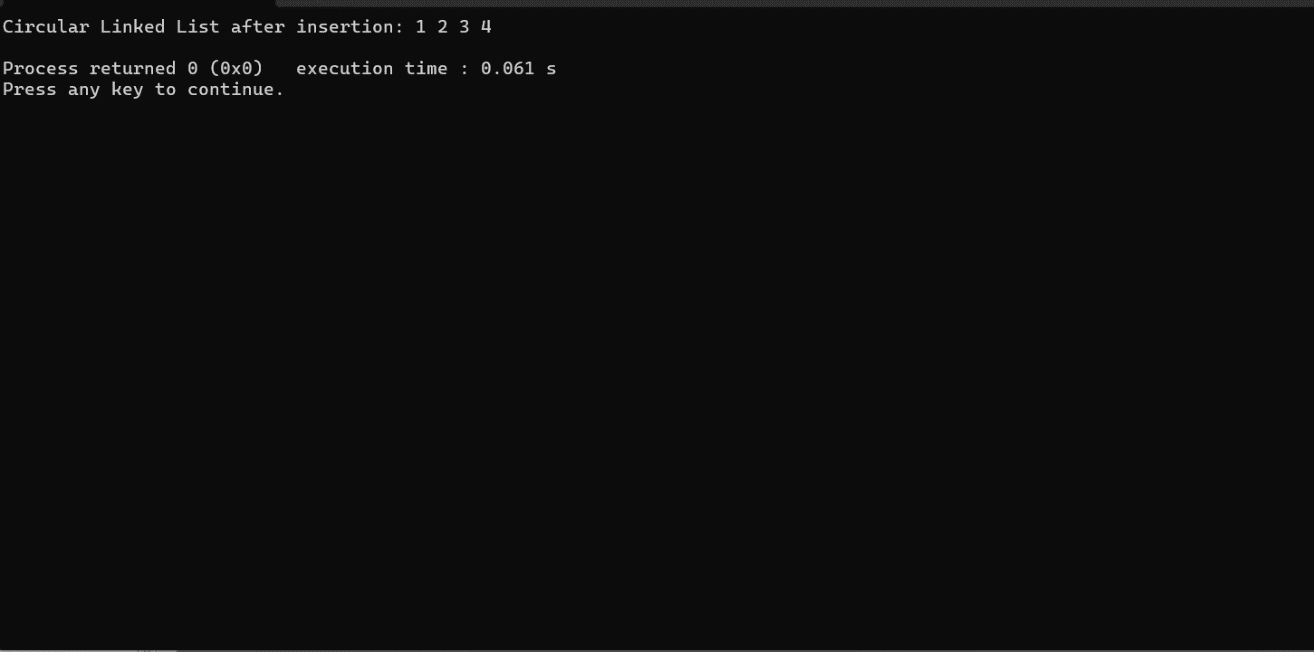
printf("Circular Linked List after insertion: ");

printList(head);

return 0;

}

OUTPUT:



15. INSERT AT GIVEN CIRCULAR LINKED LIST

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void insertAtPosition(struct Node\*\* head, int data, int position) {

if (position < 0) {

printf("Invalid position!\n");

return;

}

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

if (position == 0) {

newNode->next = newNode;

\*head = newNode;

} else {

printf("List is empty, cannot insert at position %d.\n", position);

}

} else {

struct Node\* temp = \*head;

for (int i = 0; i < position - 1; i++) {

temp = temp->next;

if (temp == \*head) {

printf("Invalid position!\n");

return;

}

}

newNode->next = temp->next;

temp->next = newNode;

if (position == 0) {

\*head = newNode;

}

}

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = NULL;

insertAtPosition(&head, 1, 0); // Insert 1 at position 0

insertAtPosition(&head, 3, 1); // Insert 3 at position 1

insertAtPosition(&head, 2, 1); // Insert 2 at position 1

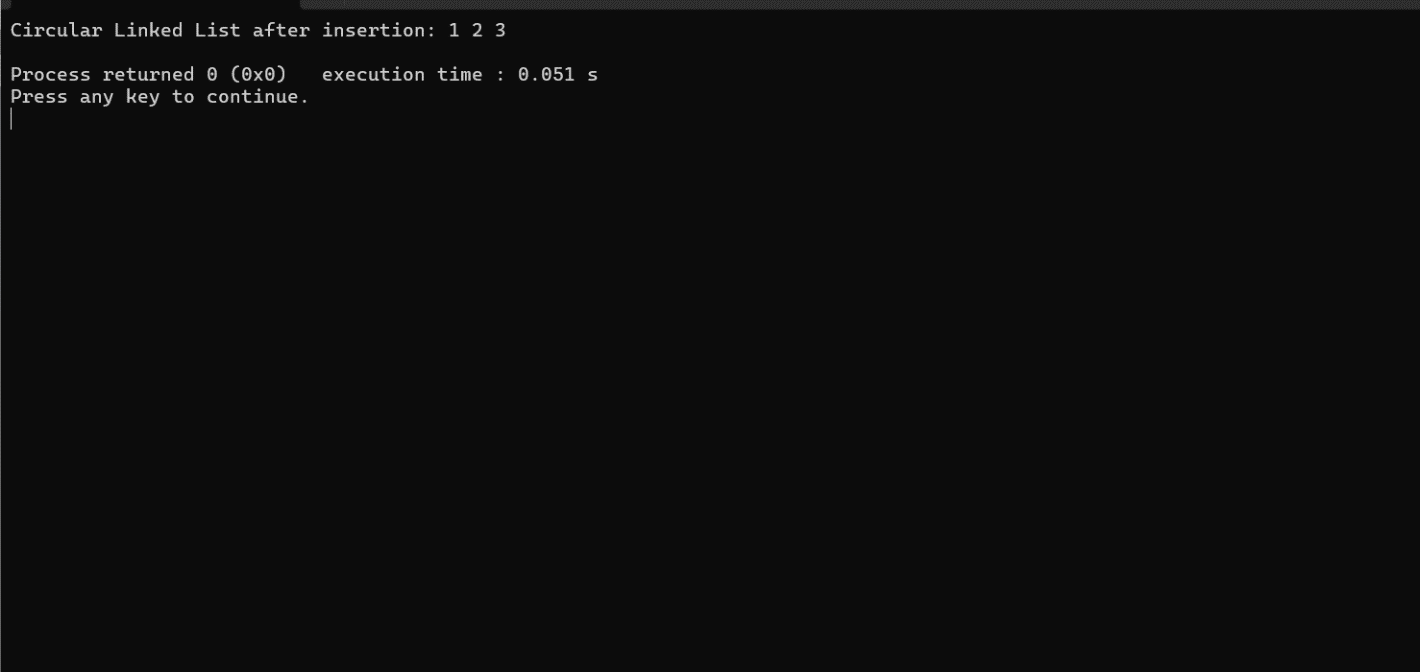
printf("Circular Linked List after insertion: ");

printList(head);

return 0;

}

OUTPUT:



16. INSERT BEFORE GIVEN NODE CIRCULAR LINKED LIST

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void insertAfterNode(struct Node\* prevNode, int data) {

if (prevNode == NULL) {

printf("Previous node cannot be NULL!\n");

return;

}

struct Node\* newNode = createNode(data);

newNode->next = prevNode->next;

prevNode->next = newNode;

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = head;

printf("Circular Linked List before insertion: ");

printList(head);

insertAfterNode(head->next, 4);

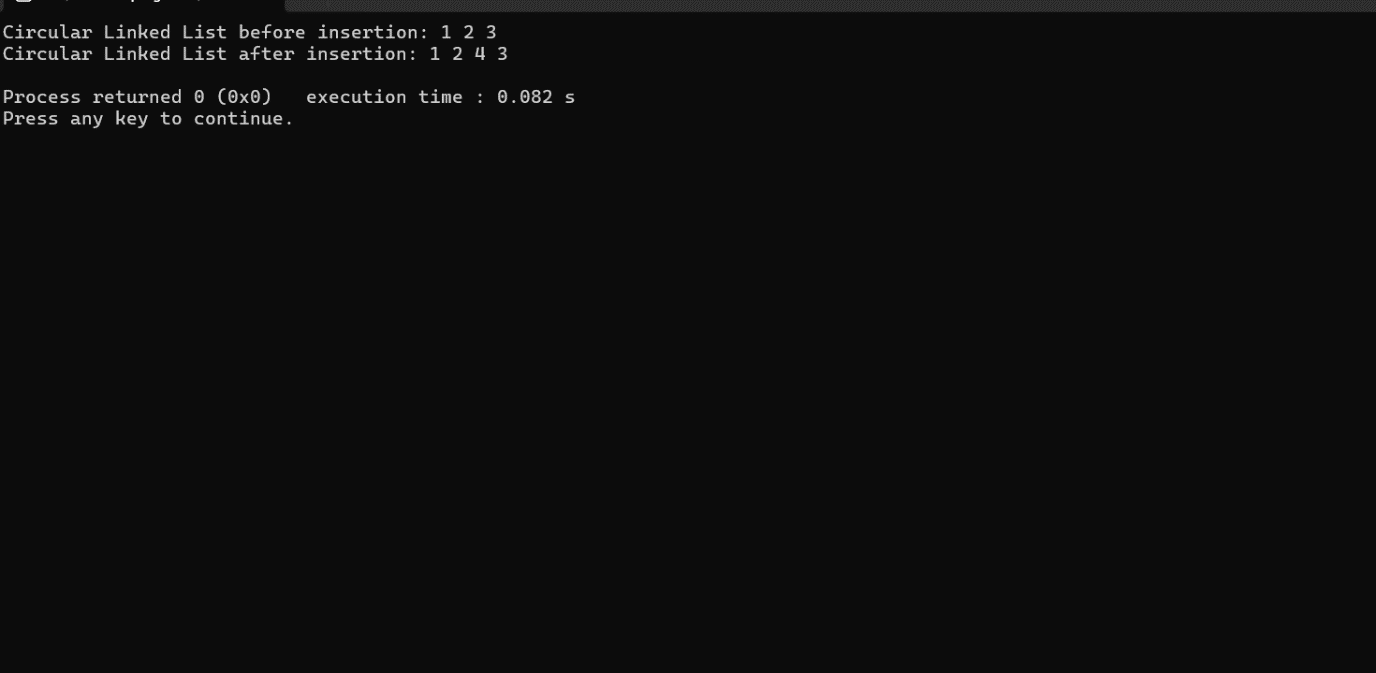
printf("Circular Linked List after insertion: ");

printList(head);

return 0;

}

OUTPUT:



17. DELETE FIRST NODE USING CIRCULAR

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void deleteFirstNode(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty, nothing to delete!\n");

return;

}

struct Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

struct Node\* toDelete = \*head;

if ((\*head)->next == \*head) { // Only one node in the list

\*head = NULL;

} else {

temp->next = (\*head)->next;

\*head = (\*head)->next;

}

free(toDelete);

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = head; // Making the list circular

printf("Circular Linked List before deletion: ");

printList(head);

deleteFirstNode(&head);

printf("Circular Linked List after deletion: ");

printList(head);

return 0;

}

OUTPUT:



18.DELETE LAST NODE USING CIRCULAR

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void deleteLastNode(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty, nothing to delete!\n");

return;

}

struct Node\* temp = \*head;

struct Node\* prev = NULL;

while (temp->next != \*head) {

prev = temp;

temp = temp->next;

}

if (prev == NULL) { // Only one node in the list

free(temp);

\*head = NULL;

} else {

prev->next = (\*head)->next;

free(temp);

}

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = head; // Making the list circular

printf("Circular Linked List before deletion: ");

printList(head);

deleteLastNode(&head);

printf("Circular Linked List after deletion: ");

printList(head);

return 0;

}

OUTPUT:



18.DELETE A NODE AT GIVEN POSTION

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void deleteNodeAtPosition(struct Node\*\* head, int position) {

if (\*head == NULL) {

printf("List is empty, nothing to delete!\n");

return;

}

struct Node\* current = \*head;

struct Node\* prev = NULL;

// Move to the node to be deleted

for (int i = 0; i < position; i++) {

prev = current;

current = current->next;

if (current == \*head) {

printf("Invalid position!\n");

return;

}

}

// If the node to be deleted is the head node

if (current == \*head) {

struct Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

if (\*head == (\*head)->next) {

\*head = NULL;

} else {

temp->next = (\*head)->next;

\*head = temp->next;

}

} else {

prev->next = current->next;

}

free(current);

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = head; // Making the list circular

printf("Circular Linked List before deletion: ");

printList(head);

deleteNodeAtPosition(&head, 1); // Delete node at position 1

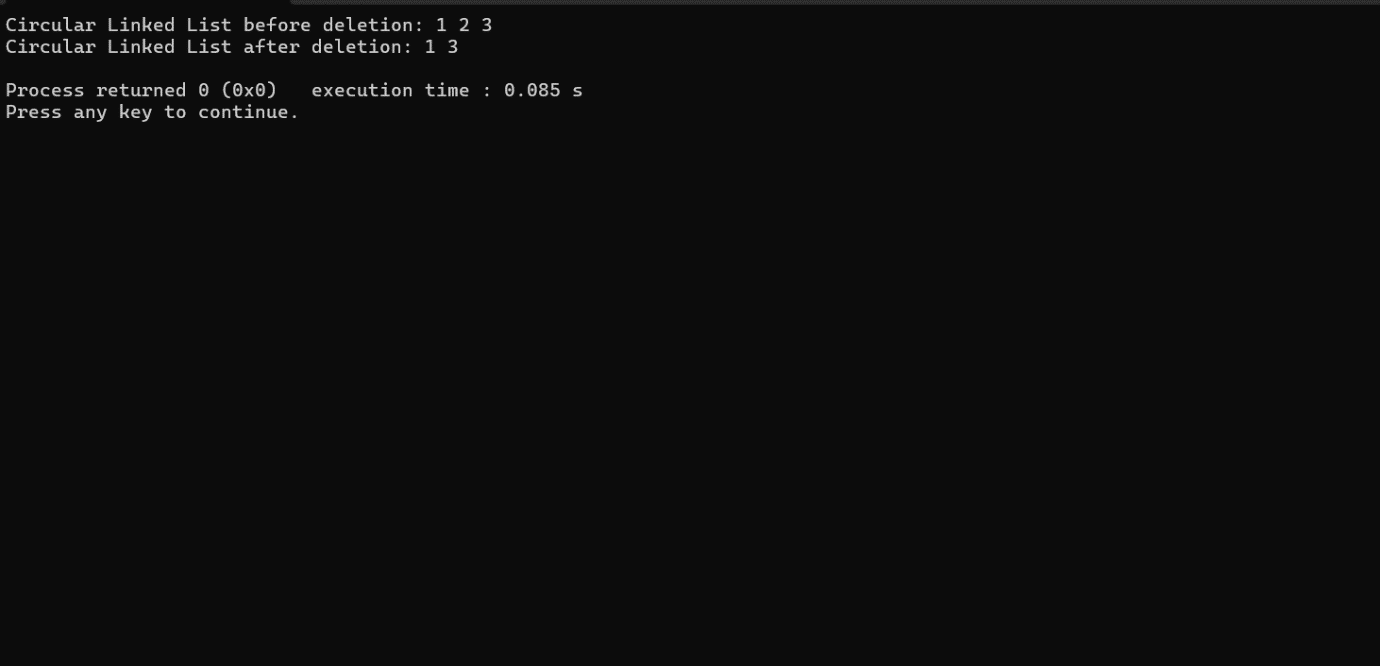
printf("Circular Linked List after deletion: ");

printList(head);

return 0;

}

OUTPUT:



19:DELETE THE NODE OF GIVEN VALUE CIRCULAR

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void deleteNodeWithValue(struct Node\*\* head, int value) {

if (\*head == NULL) {

printf("List is empty, nothing to delete!\n");

return;

}

struct Node\* current = \*head;

struct Node\* prev = NULL;

// Traverse the list to find the node with the given value

while (current->data != value && current->next != \*head) {

prev = current;

current = current->next;

}

// If the node with the given value is found

if (current->data == value) {

// If the node to be deleted is the head node

if (current == \*head) {

struct Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

if (\*head == (\*head)->next) {

\*head = NULL;

} else {

temp->next = (\*head)->next;

\*head = temp->next;

}

} else {

prev->next = current->next;

}

free(current);

} else {

printf("Node with value %d not found in the list!\n", value);

}

}

void printList(struct Node\* head) {

if (head == NULL)

return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = head; // Making the list circular

printf("Circular Linked List before deletion: ");

printList(head);

deleteNodeWithValue(&head, 2); // Delete node with value 2

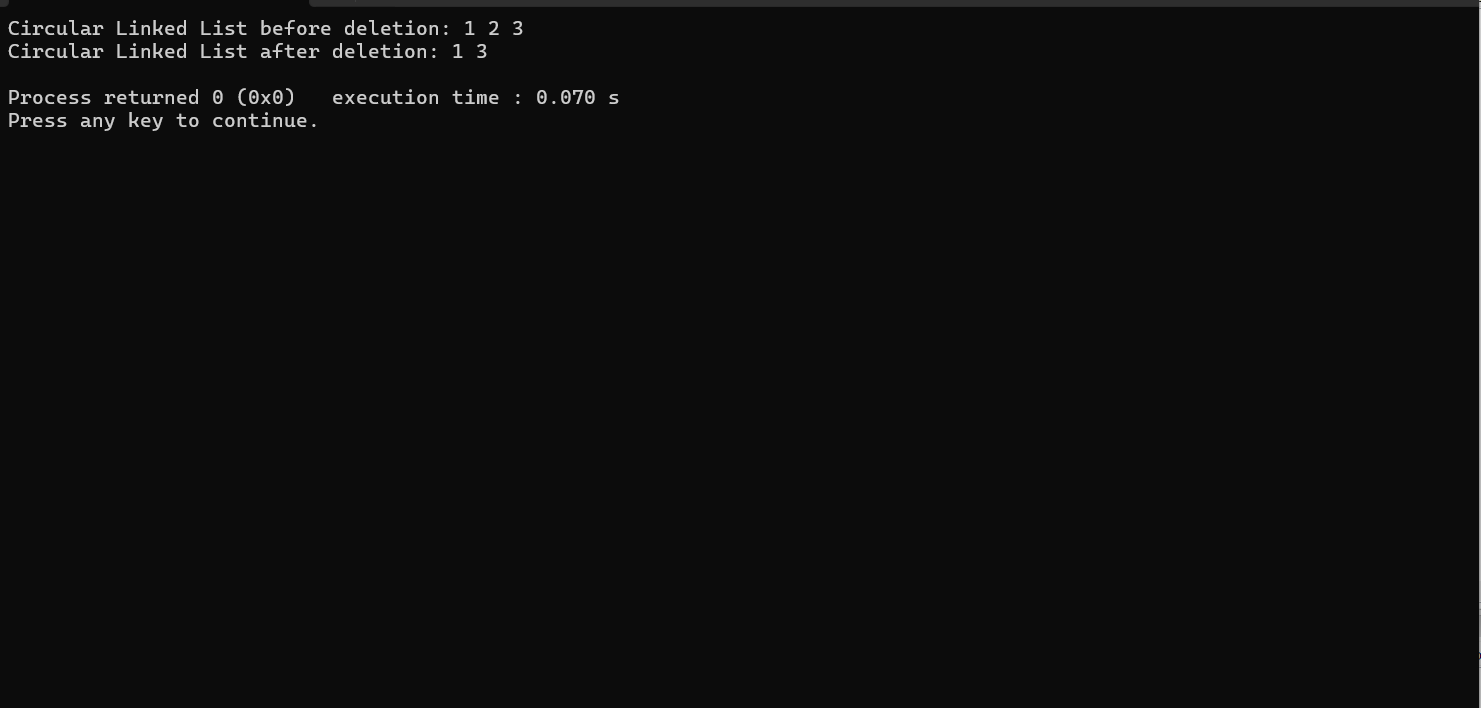
printf("Circular Linked List after deletion: ");

printList(head);

return 0;

}

OUTPUT:



20.DELETE ENTIRE LIST CIRCULAR

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void deleteList(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty, nothing to delete!\n");

return;

}

struct Node\* current = \*head;

struct Node\* temp;

while (current->next != \*head) {

temp = current;

current = current->next;

free(temp);

}

free(current);

\*head = NULL;

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = head;

printf("Circular Linked List before deletion: ");

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

deleteList(&head);

printf("Circular Linked List after deletion: ");

if (head == NULL) {

printf("List is empty.\n");

} else {

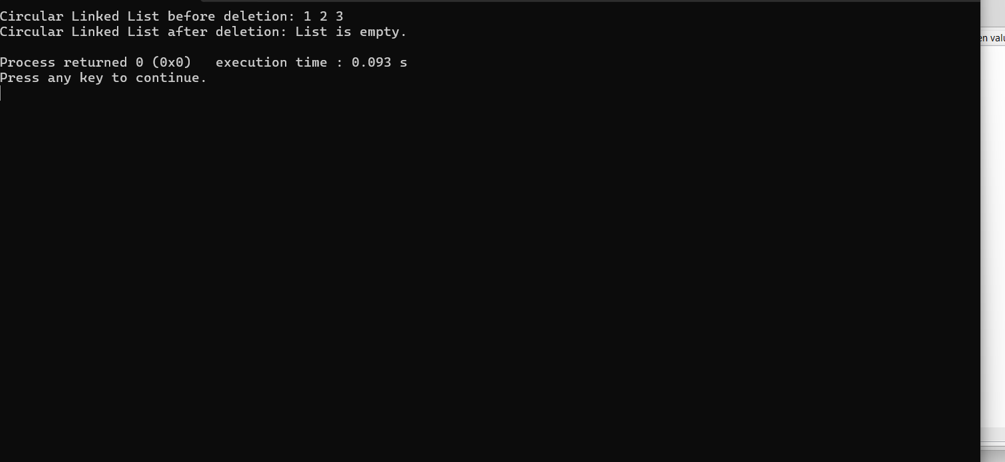
printf("List still exists.\n");

}

return 0;

}

OUTPUT:



21. Traverse and print all elements of the list CIRCULAR

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

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;

}

void printList(struct Node\* head) {

if (head == NULL) {

printf("List is empty!\n");

return;

}

struct Node\* current = head;

do {

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

current = current->next;

} while (current != head);

printf("\n");

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = head; // Making the list circular

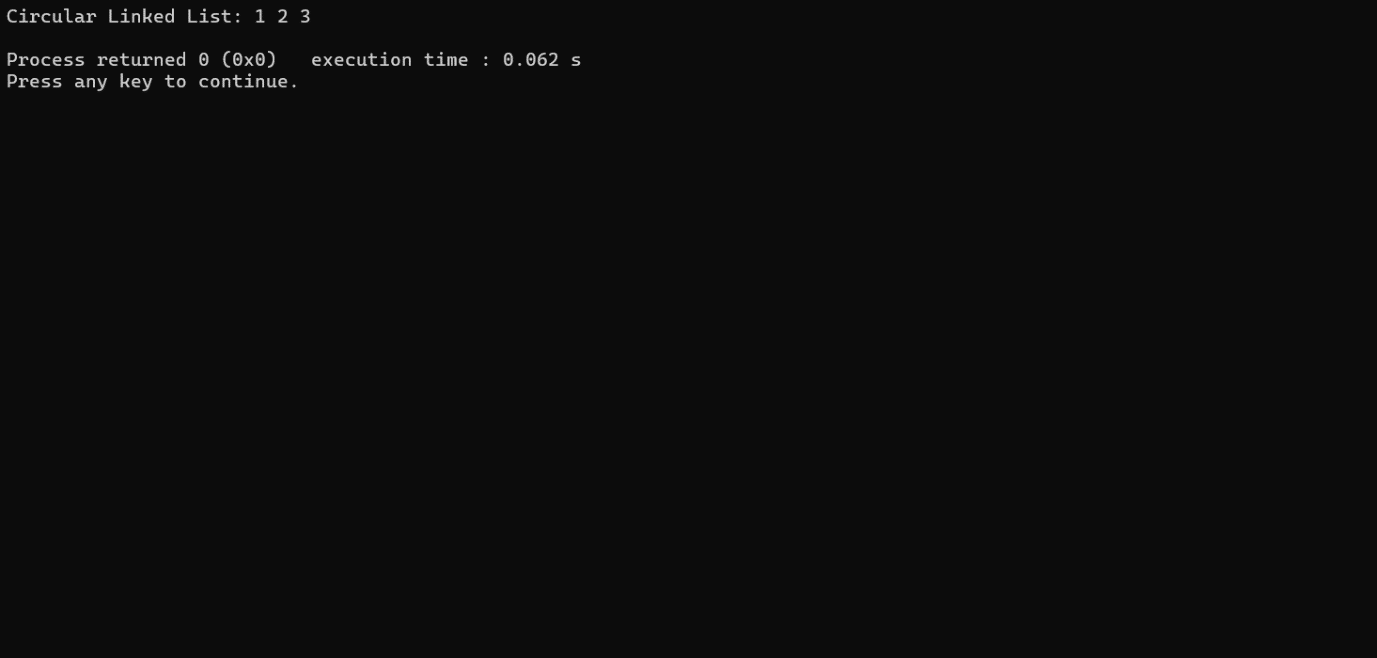
printf("Circular Linked List: ");

printList(head);

return 0;

}

OUTPUT:



STACKS

22.PUSH

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Stack {

int items[MAX\_SIZE];

int top;

};

void initializeStack(struct Stack \*s) {

s->top = -1;

}

int isEmpty(struct Stack \*s) {

return s->top == -1;

}

int isFull(struct Stack \*s) {

return s->top == MAX\_SIZE - 1;

}

void push(struct Stack \*s, int value) {

if (isFull(s)) {

printf("Stack overflow!\n");

exit(1);

}

s->items[++s->top] = value;

}

void printStack(struct Stack \*s) {

if (isEmpty(s)) {

printf("Stack is empty!\n");

return;

}

printf("Stack: ");

for (int i = 0; i <= s->top; i++) {

printf("%d ", s->items[i]);

}

printf("\n");

}

int main() {

struct Stack s;

initializeStack(&s);

push(&s, 1);

push(&s, 2);

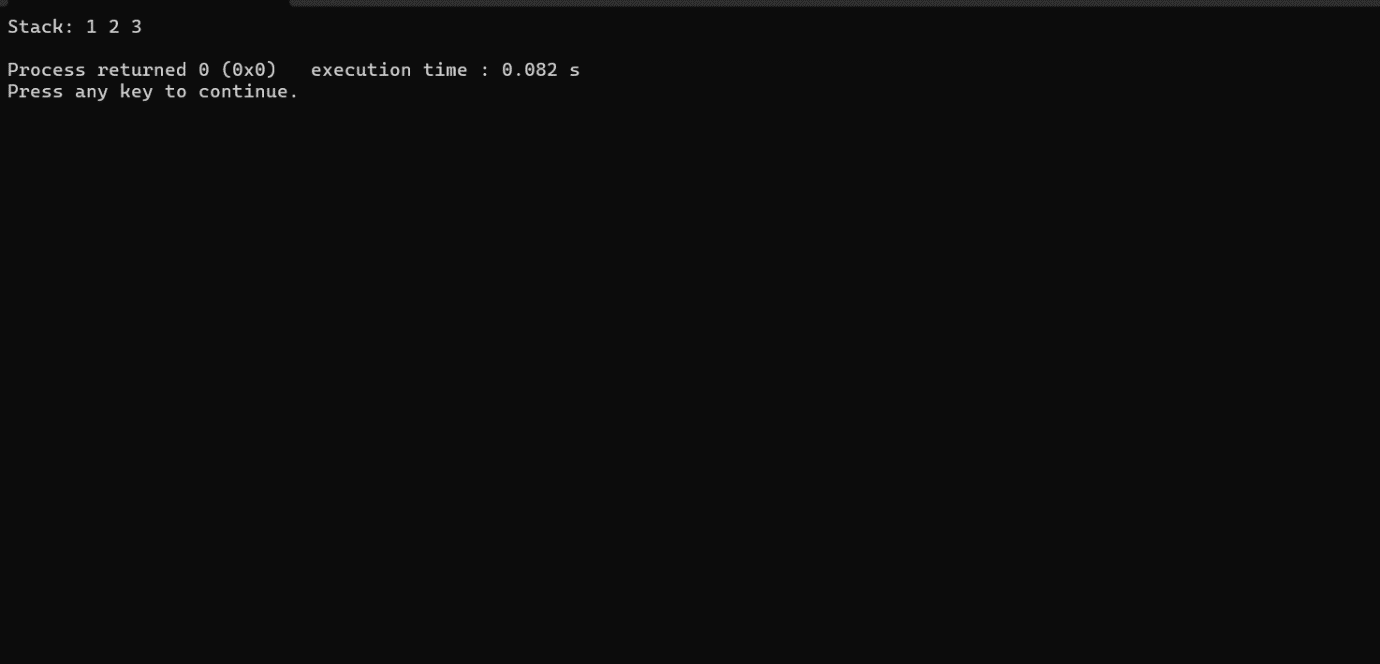
push(&s, 3);

printStack(&s);

return 0;

}

OUTPUT:



23.POP

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Stack {

int items[MAX\_SIZE];

int top;

};

void initializeStack(struct Stack \*s) {

s->top = -1;

}

int isEmpty(struct Stack \*s) {

return s->top == -1;

}

int isFull(struct Stack \*s) {

return s->top == MAX\_SIZE - 1;

}

void push(struct Stack \*s, int value) {

if (isFull(s)) {

printf("Stack overflow!\n");

exit(1);

}

s->items[++s->top] = value;

}

int pop(struct Stack \*s) {

if (isEmpty(s)) {

printf("Stack underflow!\n");

exit(1);

}

return s->items[s->top--];

}

void printStack(struct Stack \*s) {

if (isEmpty(s)) {

printf("Stack is empty!\n");

return;

}

printf("Stack: ");

for (int i = 0; i <= s->top; i++) {

printf("%d ", s->items[i]);

}

printf("\n");

}

int main() {

struct Stack s;

initializeStack(&s);

push(&s, 1);

push(&s, 2);

push(&s, 3);

printf("Before popping:\n");

printStack(&s);

int popped = pop(&s);

printf("\nPopped element: %d\n", popped);

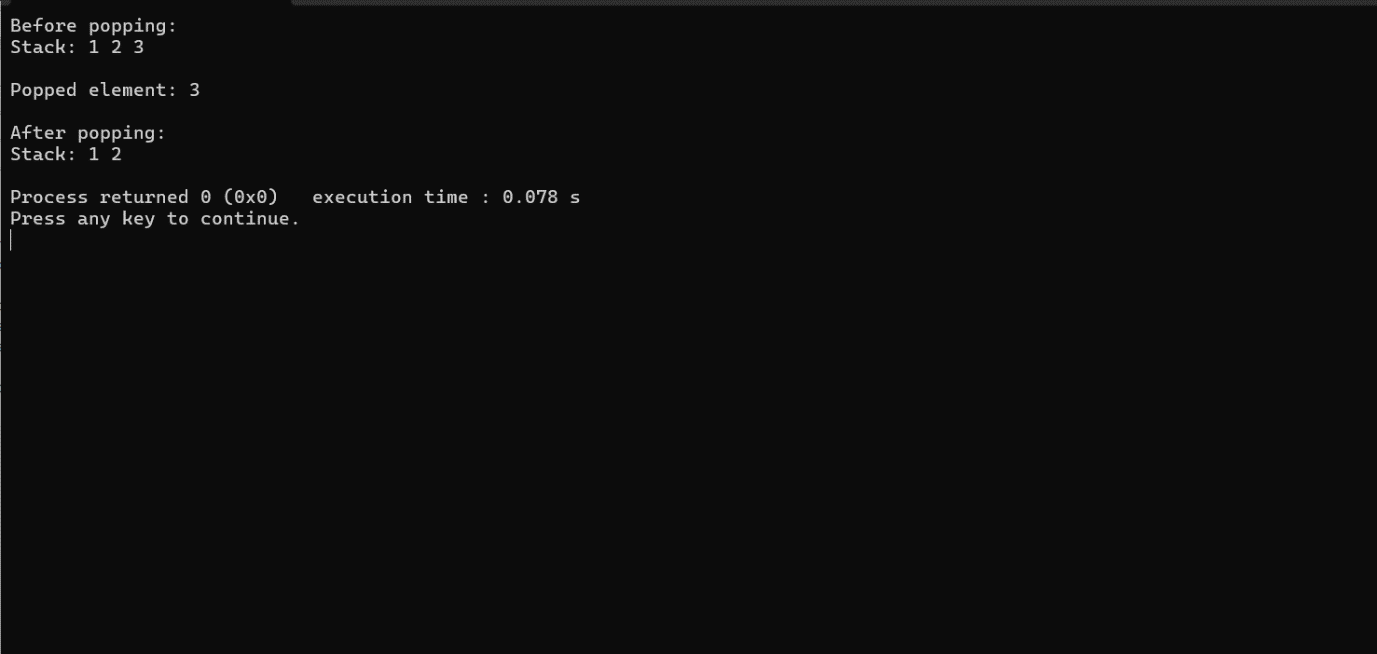
printf("\nAfter popping:\n");

printStack(&s);

return 0;

}

OUTPUT:



23.PEEK

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Stack {

int items[MAX\_SIZE];

int top;

};

void initializeStack(struct Stack \*s) {

s->top = -1;

}

int isEmpty(struct Stack \*s) {

return s->top == -1;

}

int isFull(struct Stack \*s) {

return s->top == MAX\_SIZE - 1;

}

void push(struct Stack \*s, int value) {

if (isFull(s)) {

printf("Stack overflow!\n");

exit(1);

}

s->items[++s->top] = value;

}

int pop(struct Stack \*s) {

if (isEmpty(s)) {

printf("Stack underflow!\n");

exit(1);

}

return s->items[s->top--];

}

int peek(struct Stack \*s) {

if (isEmpty(s)) {

printf("Stack is empty!\n");

exit(1);

}

return s->items[s->top];

}

void printStack(struct Stack \*s) {

if (isEmpty(s)) {

printf("Stack is empty!\n");

return;

}

printf("Stack: ");

for (int i = 0; i <= s->top; i++) {

printf("%d ", s->items[i]);

}

printf("\n");

}

int main() {

struct Stack s;

initializeStack(&s);

push(&s, 1);

push(&s, 2);

push(&s, 3);

printf("Before popping:\n");

printStack(&s);

int popped = pop(&s);

printf("\nPopped element: %d\n", popped);

printf("\nAfter popping:\n");

printStack(&s);

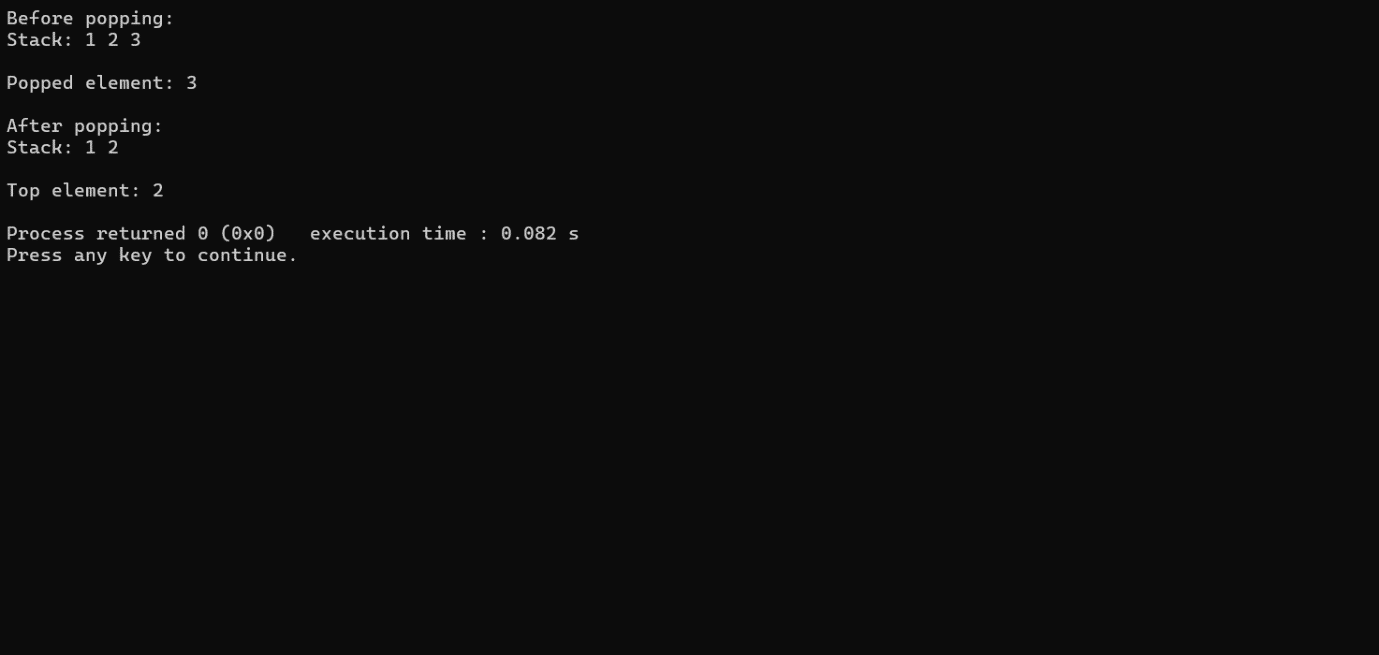
int topElement = peek(&s);

printf("\nTop element: %d\n", topElement);

return 0;

}

OUTOUT:



QUEUES:

24.ENQUEUE

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Queue {

int items[MAX\_SIZE];

int front, rear;

};

void initializeQueue(struct Queue \*q) {

q->front = -1;

q->rear = -1;

}

int isEmpty(struct Queue \*q) {

return q->front == -1;

}

int isFull(struct Queue \*q) {

return (q->rear == MAX\_SIZE - 1 && q->front == 0) || (q->rear == q->front - 1);

}

void enqueue(struct Queue \*q, int value) {

if (isFull(q)) {

printf("Queue overflow!\n");

exit(1);

}

if (q->rear == -1) {

q->front = q->rear = 0;

} else {

q->rear = (q->rear + 1) % MAX\_SIZE;

}

q->items[q->rear] = value;

}

void printQueue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty!\n");

return;

}

printf("Queue: ");

int i = q->front;

do {

printf("%d ", q->items[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (q->rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

struct Queue q;

initializeQueue(&q);

enqueue(&q, 1);

enqueue(&q, 2);

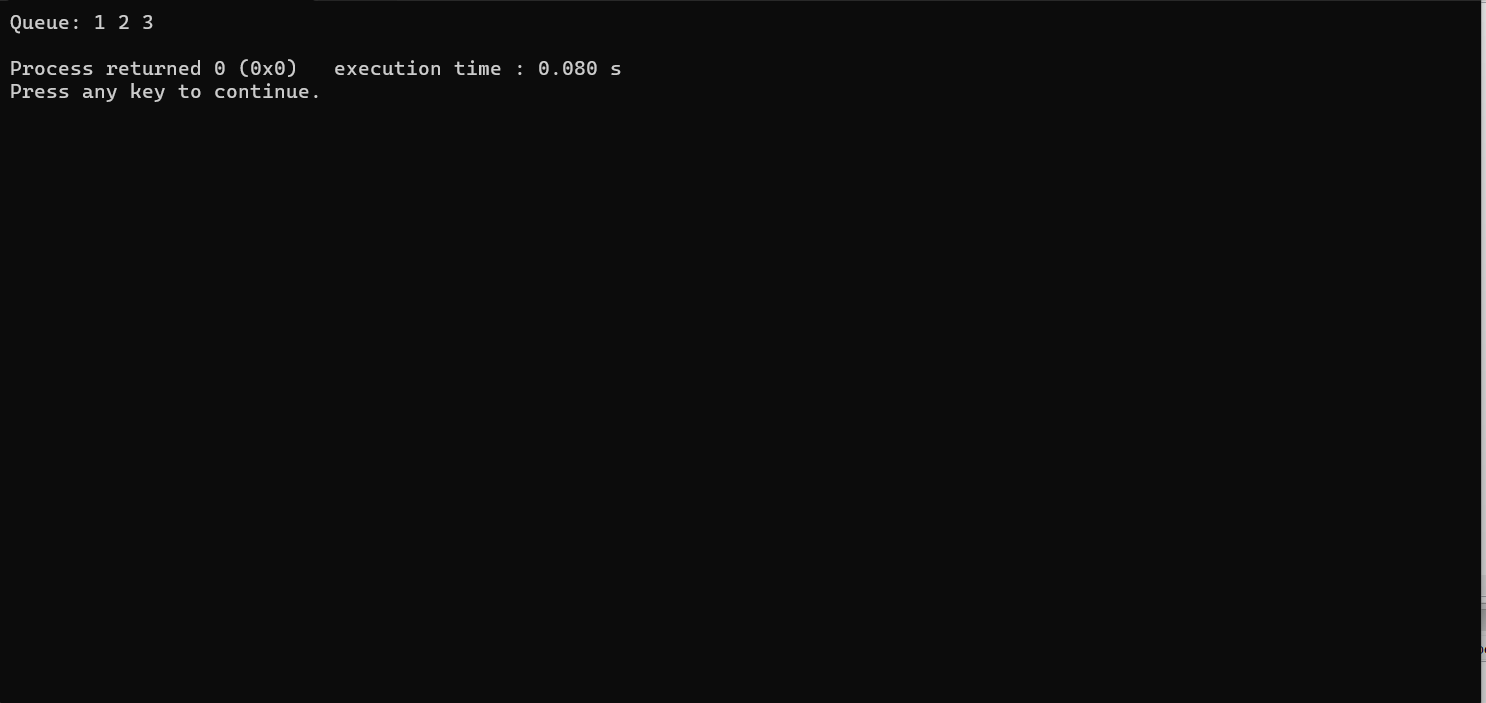
enqueue(&q, 3);

printQueue(&q);

return 0;

}

OUTPUT:



25.DEQEUE

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Queue {

int items[MAX\_SIZE];

int front, rear;

};

void initializeQueue(struct Queue \*q) {

q->front = -1;

q->rear = -1;

}

int isEmpty(struct Queue \*q) {

return q->front == -1;

}

int isFull(struct Queue \*q) {

return (q->rear == MAX\_SIZE - 1 && q->front == 0) || (q->rear == q->front - 1);

}

void enqueue(struct Queue \*q, int value) {

if (isFull(q)) {

printf("Queue overflow!\n");

exit(1);

}

if (q->rear == -1) {

q->front = q->rear = 0;

} else {

q->rear = (q->rear + 1) % MAX\_SIZE;

}

q->items[q->rear] = value;

}

int dequeue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue underflow!\n");

exit(1);

}

int dequeuedItem = q->items[q->front];

if (q->front == q->rear) {

q->front = q->rear = -1;

} else {

q->front = (q->front + 1) % MAX\_SIZE;

}

return dequeuedItem;

}

void printQueue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty!\n");

return;

}

printf("Queue: ");

int i = q->front;

do {

printf("%d ", q->items[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (q->rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

struct Queue q;

initializeQueue(&q);

enqueue(&q, 1);

enqueue(&q, 2);

enqueue(&q, 3);

printf("Before dequeue:\n");

printQueue(&q);

int dequeuedItem = dequeue(&q);

printf("\nDequeued element: %d\n", dequeuedItem);

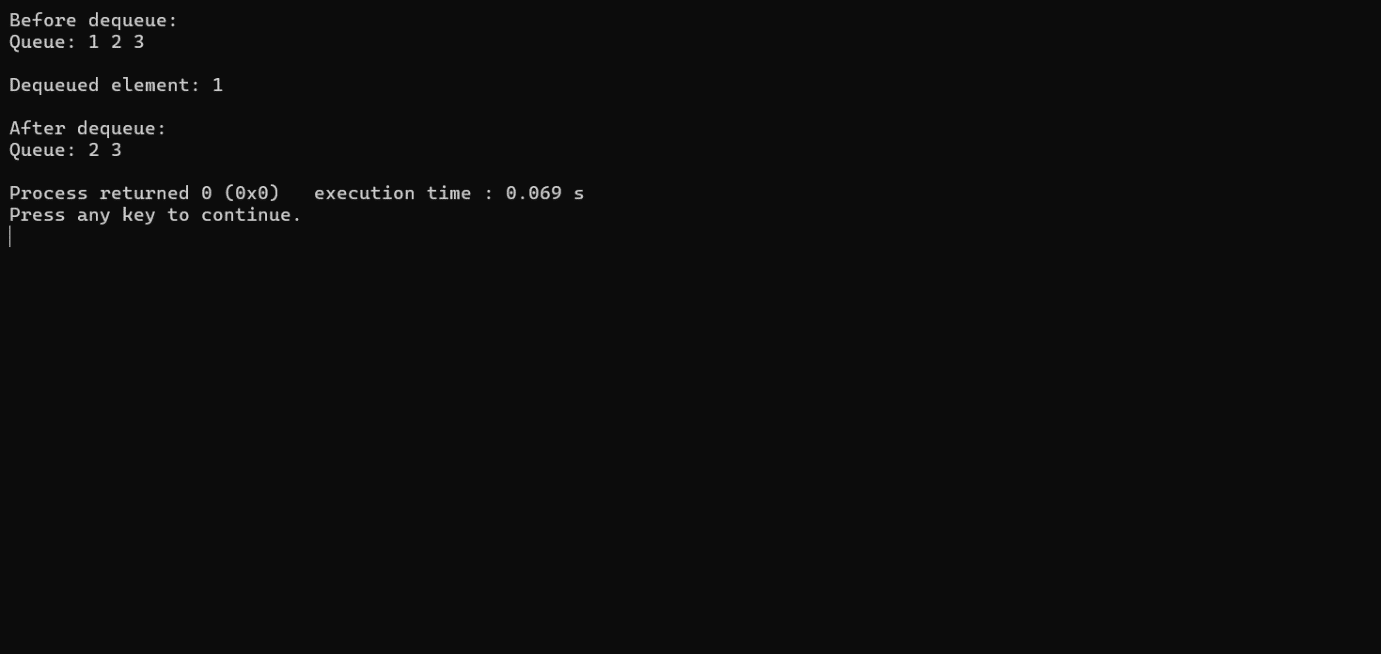
printf("\nAfter dequeue:\n");

printQueue(&q);

return 0;

}

OUTPUT:



26.

Front: Return the element at the front of the queue without removing it

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Queue {

int items[MAX\_SIZE];

int front, rear;

};

void initializeQueue(struct Queue \*q) {

q->front = -1;

q->rear = -1;

}

int isEmpty(struct Queue \*q) {

return q->front == -1;

}

int isFull(struct Queue \*q) {

return (q->rear == MAX\_SIZE - 1 && q->front == 0) || (q->rear == q->front - 1);

}

void enqueue(struct Queue \*q, int value) {

if (isFull(q)) {

printf("Queue overflow!\n");

exit(1);

}

if (q->rear == -1) {

q->front = q->rear = 0;

} else {

q->rear = (q->rear + 1) % MAX\_SIZE;

}

q->items[q->rear] = value;

}

int dequeue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue underflow!\n");

exit(1);

}

int dequeuedItem = q->items[q->front];

if (q->front == q->rear) {

q->front = q->rear = -1;

} else {

q->front = (q->front + 1) % MAX\_SIZE;

}

return dequeuedItem;

}

int front(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty!\n");

exit(1);

}

return q->items[q->front];

}

void printQueue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty!\n");

return;

}

printf("Queue: ");

int i = q->front;

do {

printf("%d ", q->items[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (q->rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

struct Queue q;

initializeQueue(&q);

enqueue(&q, 1);

enqueue(&q, 2);

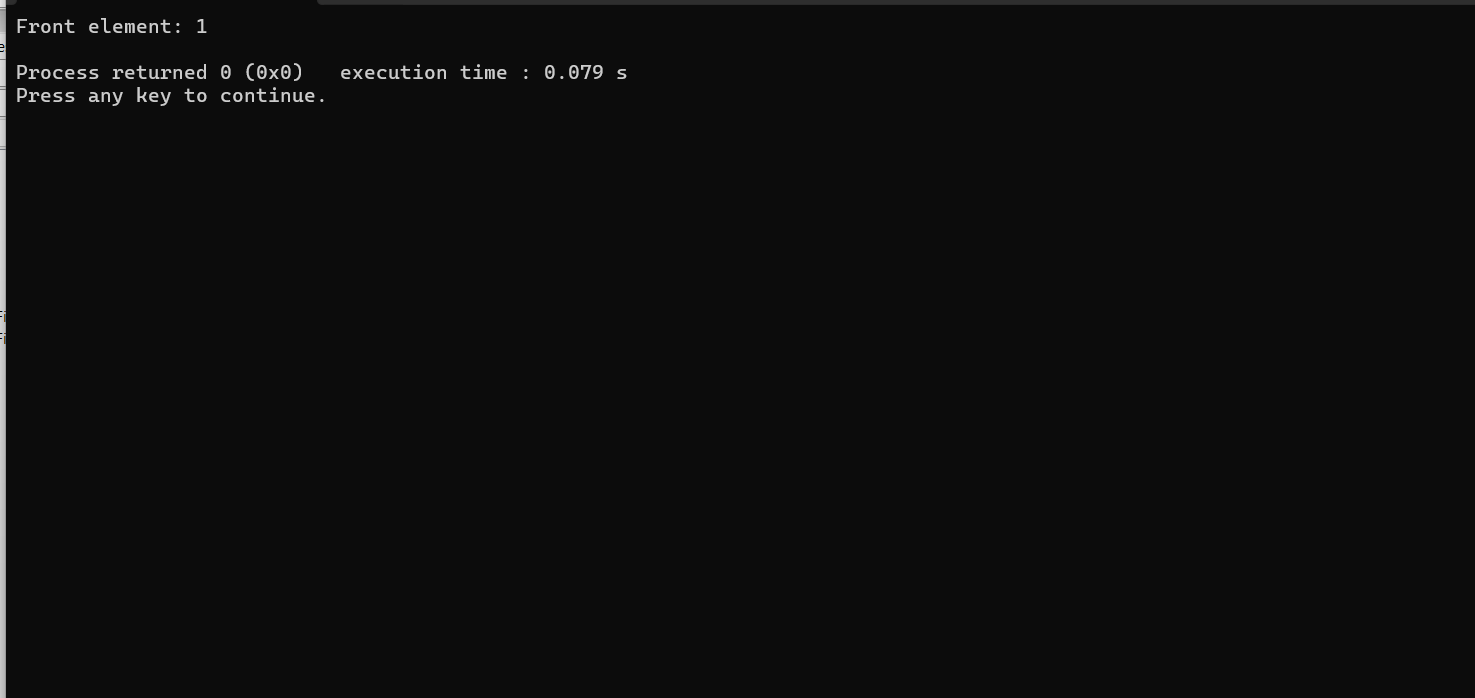
enqueue(&q, 3);

printf("Front element: %d\n", front(&q));

return 0;

}

OUTPUT:



27.CIRCULAR QUEUE:

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 5 // Maximum size of the circular queue

struct CircularQueue {

int items[MAX\_SIZE];

int front, rear;

};

void initializeQueue(struct CircularQueue \*cq) {

cq->front = -1;

cq->rear = -1;

}

int isEmpty(struct CircularQueue \*cq) {

return cq->front == -1;

}

int isFull(struct CircularQueue \*cq) {

return (cq->rear + 1) % MAX\_SIZE == cq->front;

}

void enqueue(struct CircularQueue \*cq, int value) {

if (isFull(cq)) {

printf("Queue overflow!\n");

return;

}

if (isEmpty(cq))

cq->front = 0;

cq->rear = (cq->rear + 1) % MAX\_SIZE;

cq->items[cq->rear] = value;

printf("Enqueued: %d\n", value);

}

int dequeue(struct CircularQueue \*cq) {

if (isEmpty(cq)) {

printf("Queue underflow!\n");

exit(1);

}

int dequeuedItem = cq->items[cq->front];

if (cq->front == cq->rear)

cq->front = cq->rear = -1;

else

cq->front = (cq->front + 1) % MAX\_SIZE;

printf("Dequeued: %d\n", dequeuedItem);

return dequeuedItem;

}

void printQueue(struct CircularQueue \*cq) {

if (isEmpty(cq)) {

printf("Queue is empty!\n");

return;

}

printf("Queue: ");

int i = cq->front;

do {

printf("%d ", cq->items[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (cq->rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

struct CircularQueue cq;

initializeQueue(&cq);

enqueue(&cq, 1);

enqueue(&cq, 2);

enqueue(&cq, 3);

printQueue(&cq);

dequeue(&cq);

printQueue(&cq);

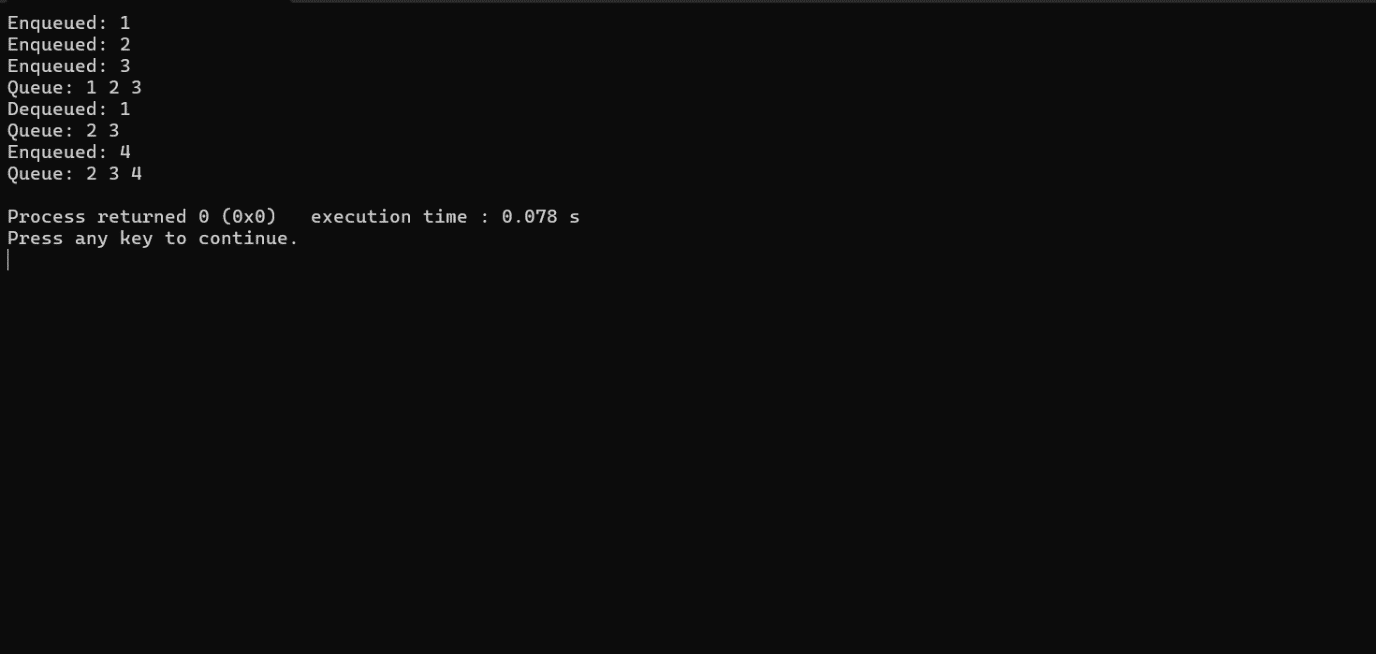
enqueue(&cq, 4);

printQueue(&cq);

return 0;

}

OUTPUT:



28. ADDING AN ELEMENT TO THE FRONT OF A DOUBLE-ENDED QUEUE

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Deque {

int items[MAX\_SIZE];

int front, rear;

};

void initializeDeque(struct Deque \*dq) {

dq->front = -1;

dq->rear = -1;

}

int isEmpty(struct Deque \*dq) {

return dq->front == -1;

}

int isFull(struct Deque \*dq) {

return (dq->front == 0 && dq->rear == MAX\_SIZE - 1) || (dq->rear == (dq->front - 1) % (MAX\_SIZE - 1));

}

void addToFront(struct Deque \*dq, int value) {

if (isFull(dq)) {

printf("Deque overflow!\n");

exit(1);

}

if (dq->front == -1) {

dq->front = dq->rear = 0;

} else if (dq->front == 0) {

dq->front = MAX\_SIZE - 1;

} else {

dq->front = dq->front - 1;

}

dq->items[dq->front] = value;

printf("Added %d to the front of the deque.\n", value);

}

void printDeque(struct Deque \*dq) {

if (isEmpty(dq)) {

printf("Deque is empty!\n");

return;

}

printf("Deque: ");

int i = dq->front;

do {

printf("%d ", dq->items[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (dq->rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

struct Deque dq;

initializeDeque(&dq);

addToFront(&dq, 1);

addToFront(&dq, 2);

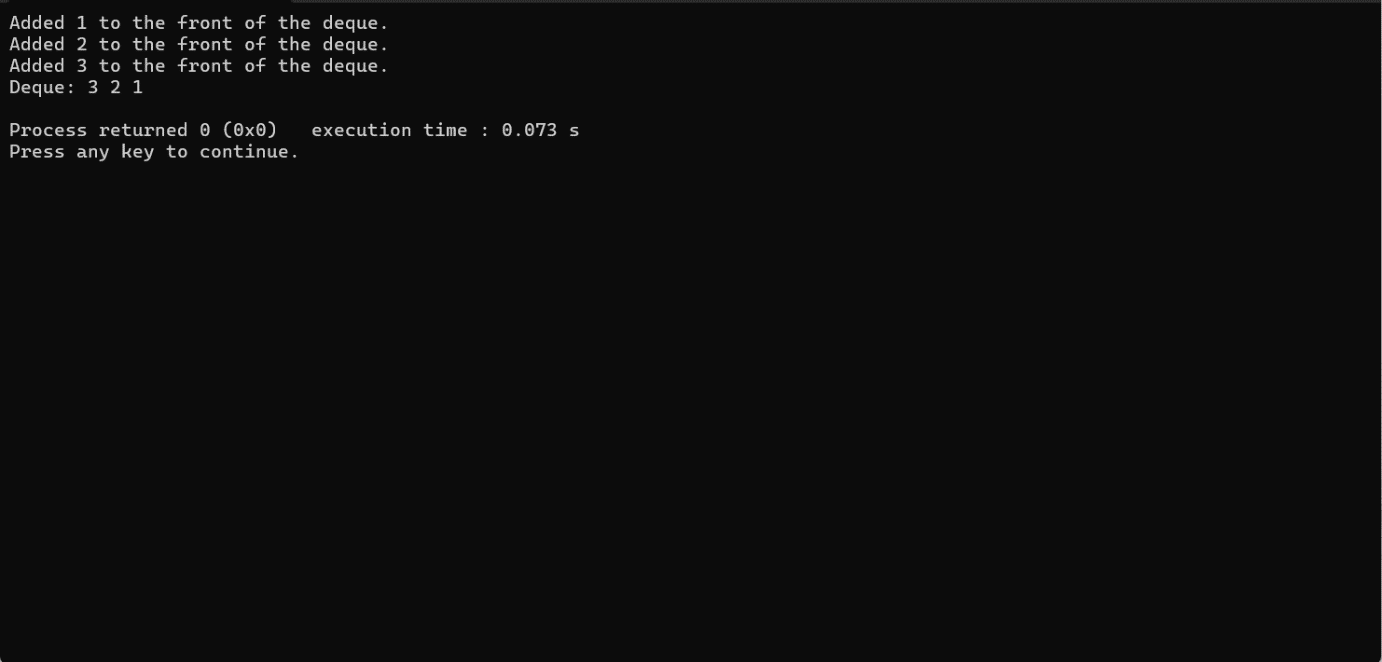
addToFront(&dq, 3);

printDeque(&dq);

return 0;

}

OUTOUT:



29.REMOVE AN ELEMENT FROM FRONT DOUBLE ENDED QUEUE

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Deque {

int items[MAX\_SIZE];

int front, rear;

};

void initializeDeque(struct Deque \*dq) {

dq->front = -1;

dq->rear = -1;

}

int isEmpty(struct Deque \*dq) {

return dq->front == -1;

}

int isFull(struct Deque \*dq) {

return (dq->front == 0 && dq->rear == MAX\_SIZE - 1) || (dq->rear == (dq->front - 1) % (MAX\_SIZE - 1));

}

int removeFromFront(struct Deque \*dq) {

if (isEmpty(dq)) {

printf("Deque underflow!\n");

exit(1);

}

int removedItem = dq->items[dq->front];

if (dq->front == dq->rear) {

dq->front = dq->rear = -1;

} else {

dq->front = (dq->front + 1) % MAX\_SIZE;

}

printf("Removed %d from the front of the deque.\n", removedItem);

return removedItem;

}

void printDeque(struct Deque \*dq) {

if (isEmpty(dq)) {

printf("Deque is empty!\n");

return;

}

printf("Deque: ");

int i = dq->front;

do {

printf("%d ", dq->items[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (dq->rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

struct Deque dq;

initializeDeque(&dq);

for (int i = 1; i <= 5; i++) {

dq.items[i - 1] = i;

dq.rear++;

if (dq.front == -1) {

dq.front = 0;

}

}

printDeque(&dq);

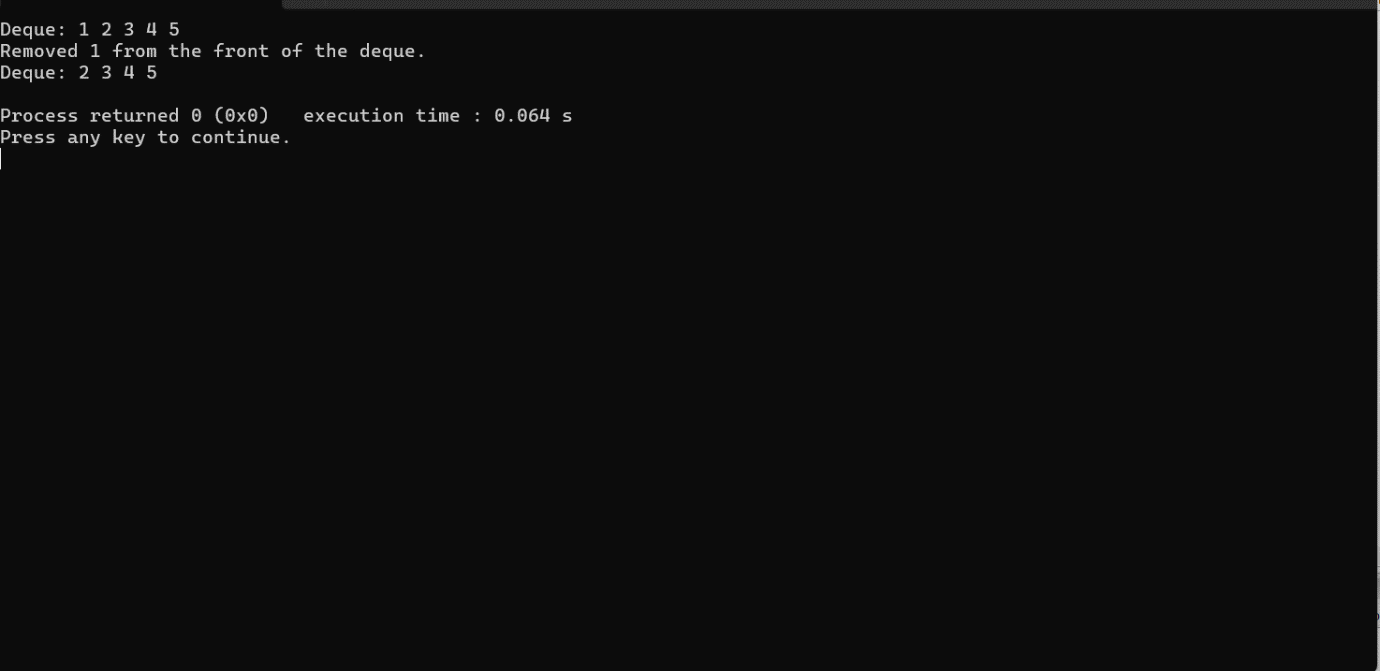
removeFromFront(&dq);

printDeque(&dq);

return 0;

}

OUTPUT:



30.INSERTION OF AN ELEMENT WITH A PRIORITY INTO A PRIORITY QUEUE

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct QueueElement {

int data;

int priority;

};

struct PriorityQueue {

struct QueueElement items[MAX\_SIZE];

int rear;

};

void initializePriorityQueue(struct PriorityQueue \*pq) {

pq->rear = -1;

}

int isEmpty(struct PriorityQueue \*pq) {

return pq->rear == -1;

}

int isFull(struct PriorityQueue \*pq) {

return pq->rear == MAX\_SIZE - 1;

}

void insert(struct PriorityQueue \*pq, int data, int priority) {

if (isFull(pq)) {

printf("Priority queue overflow!\n");

exit(1);

}

int i;

for (i = pq->rear; i >= 0; i--) {

if (priority > pq->items[i].priority) {

pq->items[i + 1] = pq->items[i];

} else {

break;

}

}

pq->items[i + 1].data = data;

pq->items[i + 1].priority = priority;

pq->rear++;

}

void printPriorityQueue(struct PriorityQueue \*pq) {

if (isEmpty(pq)) {

printf("Priority queue is empty!\n");

return;

}

printf("Priority queue: ");

for (int i = 0; i <= pq->rear; i++) {

printf("(%d, %d) ", pq->items[i].data, pq->items[i].priority);

}

printf("\n");

}

int main() {

struct PriorityQueue pq;

initializePriorityQueue(&pq);

insert(&pq, 5, 2);

insert(&pq, 10, 1);

insert(&pq, 3, 3);

printPriorityQueue(&pq);

return 0;

}

OUTPUT:

