|  |  |
| --- | --- |
| **Name of the student** | **Chinmay P Hittalmakki** |
| **Roll No** | **138** |
| **USN** | **01FE22BEI044** |
| **Div** | **F** |

**Part- A**

|  |
| --- |
| **Variables and Data Types** |
| **1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate.** |
| A.  \*Variable : A variable is a named storage unit that keeps a value during a program's runtime.  \* Think of it as a box that you label and store data in.  \* The value inside the variable can change as the program runs.  \* For example, `int age;` creates a variable called `age` to hold whole numbers.  \*Data Type : A data type specifies the kind of data a variable can contain.  \* It determines the range of values the variable can store, how much memory it needs, and the operations that can be done on it.  \* For example, `int` is a data type that indicates that a variable can hold whole numbers. |
| **2.What is the difference between a variable and a data type in C programming? Provide examples to illustrate.** |
| 1. In C programming, there are different ways to classify and store data using various data types:   Basic Data Types :  \*int\* Used for storing integers (whole numbers) like -23, 10, or 4096.  \* float: Designed to store floating-point numbers (decimals) like 3.14 or -12.56 with single-precision accuracy.  \* double: Used for storing floating-point numbers with higher precision (double-precision) when accuracy is crucial.  \* char: Specifically used for storing a single character enclosed within single quotes, such as 'a', 'Z', or '#'.  Derived Data Types (Built from Basic Types):  \* Arrays: Used to create collections of elements that all have the same data type. Each element can be accessed using its index number.  \* Pointers: Special variables that store the memory address of other variables, allowing for indirect access to data.  \* Structures: These user-defined data types allow for grouping variables of different data types together.  \* Unions: A unique data type that enables a single variable to store different data types at different times. |
| **3. How are variables declared and initialized in C programming? Provide examples of variable declarations with different data types.** |
| A.   * Variable Declaration: - Create a variable by specifying its type (like integer) and a unique name. - This creates a space in memory to store the variable's value. Example: - `int age, marks;` creates two integer variables named `age` and `marks`. * Variable Initialization: - To set an initial value for a variable, assign it during declaration or later in the code.   Example: - `int count = 0;` creates and initializes `count` to 0. |
| **4. Discuss the scope and lifetime of variables in C programming. What are global and local variables?** |
| **A**   * Scope: Variables are either accessible and usable within a specific part of the program (scope). * Types of Scope: * Local Variables: Declared inside a function . Can only be accessed within that function . Lifetime: Limited to the function's execution * Global Variables: Declared outside any function . Can be accessed anywhere in the program . Lifetime: Lasts throughout the program's execution * Caution: Excessive use of global variables can lead to code maintenance issues. Local variables are typically preferred for improved encapsulation and code organization. |
| **5.** **: Explain the concept of type casting in C programming. When is type casting necessary, and how is it performed?** |
| A.   * Casting:The act of changing a value's data type explicitly. * When It's Needed: When assigning values with different data types (like assigning an integer to a floating-point number). \* When performing operations on expressions with different data types (like calculating an average with a whole number numerator and a decimal denominator). * How to Do It:Use the syntax: (target\_type) expression * Example: To calculate an accurate average, cast the integer sum to a double: double average = (double) sum / count; |
| **Operators:** |
| Q. Describe the purpose and usage of the ternary conditional operator (?:) in C programming. Provide an example demonstrating its usage. |
| A.  The ternary operator (?:) is another name for the conditional operator. It is used to write one line if-else statements in short form. This operator has three operands.  1. Condition: Expressions that are either true or false.  2. Expression if true: The result is returned if the condition is true.  3. Expression if false: The value that will be returned when a given condition fails.  Syntax:  C  condition ? expression\_if\_true : expression\_if\_false  Example:  C  int age = 25;  char\* message = (age >= 18) ? "You can vote" : "You cannot vote";  printf("%s\n", message);  This code checks whether age is greater than or equal to 18, assigns “You can vote” to message (if it does), or “You cannot vote.” |
| **Q. Discuss the bitwise operators available in C programming. Explain their usage with suitable examples.** |
| A.  Bitwise operators work on each bit of data. C has these bitwise operators:   * Bitwise AND (&): Checks if the corresponding bits of both operands are 1. Sets the result to 1 only when both are 1. * Bitwise OR (|): Sets the result to 1 if at least one corresponding bit in both operands is 1. * Bitwise XOR (^): Sets the result to 1 if the corresponding bits in both operands are different. * Left Shift (<<): Moves the bits of the left operand leftward by the specified number of positions. * **Right Shift (>>):** Shifts the bits of the left operand to the right by the number of positions specified by the right operand.   Example: x >> 1 shifts the bits in x one position to the right, effectively dividing by 2 (assuming x is an unsigned integer). |
| Q. **Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate.** |
| A.   * Increment operators (++) increase a variable's value by 1. They differ in the timing of the increment: * Postfix Increment (x++): Uses the variable's original value in the expression before incrementing it.   Example: `int x = 5; int y = x++; // y gets 5 (original value of x), then x becomes 6`   * Prefix Increment (++x): Increments the variable before using its new value in the expression. \* Example: `int x = 5; int y = ++x; // x becomes 6, then y gets 6 (new value of x) |
| **Q. : What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions?** |
| A.   * When writing conditions in code, you can combine multiple conditions using logical operators. * Logical AND (&&): Only returns `true` when all its conditions are `true`. It uses short-circuit evaluation, meaning it won't check the second condition if the first one is `false`, saving time. * For example, `(x > 0) && (y < 10)` is `true` only when both `x` is greater than 0 and `y` is less than 10. * Logical OR (||): Returns `true` if at least one of its conditions is `true`. It also uses short-circuit evaluation, so it won't check the second condition if the first one is `true`. * For example, in the code snippet: int x = 5; int y = 0; if (x > 0 || y < 10) { printf("At least one condition is true\n"); } ``` It checks if either `x` is greater than 0 or `y` is less than 10. If either of these conditions is met, it executes the code within the `if` block. |
| **Q. : Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation.** |
| A.   * **Operator Precedence:** In expressions with multiple operators, operator precedence determines the order in which operations are performed. Operators with higher precedence are evaluated first.   + Example: x = 3 + 4 \* 2 - Multiplication (precedence \*) happens first, so x = 3 + (8). Then, addition is performed, resulting in x = 11. * **Associativity:** If multiple operators have the same precedence, associativity determines the grouping of operands. Left-to-right associativity means operations are grouped from left to right. Right-to-left associativity means grouping happens from right to left.   + Example: x = 10 - 5 - 2 - Since subtraction has the same precedence, associativity dictates evaluation from left to right. So, (10 - 5) - 2 = 3.   **Here's a table summarizing common C operator precedence and associativity:**   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | Operator | Precedence | Associativity | | () | Highest | Left to right | | ++, -- (postfix/prefix) | High | Right to left | | \*, /, % | High | Left to right | | +, - | Medium | Left to right | | <, >, <=, >= | Medium | Left to right | | ==, != | Medium | Left to right | | && (logical AND) | Low | Left to right | |  | (logical OR) | Low | | =, +=, -=, \*=, /=, etc. | Low | Right to left | | |
|  |
| Control Structures in C Programming |
| Q. Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement? |
| A.  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. |
| **Q.** ) Discuss the role of the break and continue statements in loop control in C programming. Provide examples to illustrate their usage. |
| A.  The break statement serves to prematurely terminate the execution of a loop. When encountered within a loop, it immediately exits the loop, and the program control proceeds to the statement following the loop. This statement is typically employed to exit a loop prematurely based on specific conditions.  #include <stdio.h>  int main() {  int i;  // Example: Exiting a loop early using break  for (i = 0; i < 10; i++) {  if (i == 5) {  break; // Exit the loop when i reaches 5  }  printf("%d ", i);  }  printf("\n");  return 0;  }  Continue Statement:  On the other hand, the continue statement facilitates skipping the remaining code inside the loop for the current iteration and proceeding directly to the next iteration. When encountered, it bypasses the remaining code within the loop for the ongoing iteration and continues with the subsequent iteration. This statement is commonly utilized to skip certain iterations of a loop based on specific conditions without exiting the loop altogether.  #include <stdio.h>  int main() {  int i;  // Example: Skipping certain iterations using continue  for (i = 0; i < 10; i++) {  if (i % 2 == 0) {  continue;  }  printf("%d ", i);  }  printf("\n");  return 0;  }  These statements provide programmers with powerful tools to control the execution flow within loops, enabling efficient handling of various conditions. |
| **Q.** ) What are the advantages of using the for loop over the while loop in C programming? Provide examples comparing the two. |
| A.  Advantages of the for loop over the while loop:  1. Initialization, condition, and iteration in one line:  The for loop allows you to specify the initialization, condition, and iteration all in one line, making it more compact and easier to read compared to the while loop.  2. Better suited for iterating over a range:  When there's a need to iterate a specific number of times or over a range of values, the for loop is often more convenient and readable than the while loop.  3. Control over loop variables:  The loop variable(s) used in the for loop are typically scoped to the loop itself, making it easier to manage and avoiding potential issues with variable scope.  #include <stdio.h>  int main() {  int i;  printf("Using for loop:\n");  for (i = 0; i < 5; i++) {  printf("Iteration %d\n", i);  }    printf("\nUsing while loop:\n");  i = 0;  while (i < 5) {  printf("Iteration %d\n", i);  i++;  }  return 0;  } |
| **Q. Discuss the role of the break and continue statements in loop control in C programming. Provide examples to illustrate their usage.** |
| A.  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. |
| **Q.** ) Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements? |
| A.  Short-circuit evaluation 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 (&amp;&amp;): In a logical AND expression (A &amp;&amp; B), if A is 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 is 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 way if statements work in C programming. 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 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:** |
| Q. Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes? |
| A.  In C, a function prototype is like a sneak peek at a function before it's actually built. It tells the compiler what the function's name is, what it sends back (like a text message), and what information it needs to do its job. This helps the compiler double-check that everything's in order when the function is called.  Why Function Prototypes Matter:  Type Check: They make sure that the function is used properly, with the right number and type of inputs and the right kind of output.  Heads Up: They let you declare what a function will do before you actually write the code, so the compiler can get a general idea of the function's role in the program. |
| Q. 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 increments a value passed by value  void incrementByValue(int n) {  n++;  printf("Inside function: x = %d\n", n);  }  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;  } |
| **Q.** Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works |
| A.  Recursion is a programming trick where a function summons itself to tackle a puzzle. This method, vital in computer science, breaks down complex challenges into smaller, lookalike pieces. In C programming, recursion offers a sleek and tidy way to solve such problems.  How Recursion Unfolds:Base Case: Each recursive function requires rules (base cases) to halt self-summoning and produce a result. These base cases prevent endless recursion.  Recursive Case: The function keeps calling itself with altered inputs until it encounters a base case. Each call usually simplifies the problem, moving it closer to a solution. |
| **Q.** What is the significance of the return statement in C programming? How are values returned from functions? |
| A.  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 is 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;  }  int main() {  int result;  result = sum(5, 3);  printf("The sum is: %d\n", result); // Print the result  return 0;  } |
| **Q.** 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, the function arguments are passed to parameters using one of the 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  } |
| **Pointers:** |
| **Q.** .Describe the purpose and usage of pointers in C programming. How are pointers declared and initialized? |
| A.  -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; |
| **Q.** Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers. |
| A.  Pointer Arithmetic refers to the allowed mathematical operations that can be applied to pointers. Pointers store memory addresses, not values. In C, pointer arithmetic differs from regular mathematical calculations.  The operations include:   * Changing the pointer value by 1 (increment or decrement) * Adding an integer to a pointer \* Subtracting an integer from a pointer * Subtracting two pointers of the same type * Comparing pointers * Example of Addition: ```   #include <stdio.h>  int main()  { int N = 4; int \*ptr1, \*ptr2;  ptr1 = &N; // Stores the address of N in ptr1  ptr2 = &N; // Stores the same address in ptr2  printf("Pointer ptr2 before Addition: %p\n", ptr2); // Prints the address stored in ptr2 // Adds 3 to the pointer value  ptr2 = ptr2 + 3;  printf("Pointer ptr2 after Addition: %p\n", ptr2); // Prints the updated address stored in ptr2 return 0; }  In this example, the pointer ptr2 is incremented by 3, effectively moving it 3 memory locations further from its initial address. |
| **Q.** 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 n= 5;  increment(n); // Pass n by value  printf("Outside function: %d\n", n);  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);  printf("Outside function: %d\n", x);  return 0;  } |
| **Q.** . Describe the concept of NULL pointers in C programming. How are NULL pointers used and checked for in programs? |
| A.  NULL Pointer  Concept: Meaning: A NULL pointer doesn't link to a specific memory spot. Initialization: Pointers are often set to NULL when they're created but don't have a memory address yet.  Using NULL Pointers:  Initialization: Set pointers to NULL at the start to prevent accessing uninitialized memory.  Error Handling: Use NULL to show that memory allocation functions like malloc, calloc, and realloc have failed.  Checking for NULL Pointers:  Equality Check: Compare a pointer to NULL (the literal or the macro from <stddef.h>) to see if it's NULL. |
| **Q.** 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:  #include <stdio.h>  #include <stdlib.h>  int main() {  int \*ptr;  int n = 5;  ptr = (int \*)malloc(n \* sizeof(int));  if (ptr == NULL) {  printf("Memory allocation failed\n");  return 1;  }    for (int i = 0; i < n; i++) {  ptr[i] = i + 1;  }  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:** |
| **Q.** Discuss the concept of strings in C programming. How are strings represented and manipulated in C? |
| A.  String Representation: Strings in C are stored as character arrays. The last character is always a special character called the null character ('\0'). It signals the end of the string.  String Example: A string can be declared like this:  char str[6] = "Hello"; ```  String Manipulation: C offers a set of string manipulation functions in the `<string.h>` header. Common functions include:  \* strcpy (destination, source): Copies a string from `source` to `destination`.  \* strncpy (destination, source, num): Copies up to `num` characters from `source` to `destination`.  \* strcmp (str1, str2): Compares two strings and returns an integer indicating if they are equal, less, or greater.  \*strncmp (str1, str2, num):\*\* Compares up to `num` characters of two strings and returns an integer indicating if they are equal, less, or greater. |
| **Q.** . 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);  return 0;  } |
| **Q.** Describe common string manipulation functions available in the C standard library. Provide examples of functions like strlen, strcpy, strcat, and strcmp. |
| 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 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);  return 0;  } |
| **Q.** . Discuss the concept of string tokenization in C programming. How are strings split into tokens using delimiter characters? |
| A.  String Tokenization in C: Tokenization is splitting a string into smaller parts, called tokens. This is useful for breaking down input or extracting information from text. C provides the `strtok` function for tokenization.  Concepts:Delimiter: A character that separates tokens (e.g., space, comma).  \*Token: A substring of the original string between delimiters.  `strtok` Function:  Purpose: Breaks a string into tokens based on delimiters. \* \*\*Syntax:\*\* `char \*strtok(char \*str, const char \*delim)` |
| **Q.** Explain the importance of null-terminated strings in C programming. How does the null character ('\0') signify the end of a string? |
| A.  Null-terminated strings are a fundamental concept in C programming. They offer a simple and efficient way to represent and manipulate strings. The null character ('\0') at the end of the string serves as a signal that marks the end of the text.Null-terminated strings are represented as arrays of characters, with the null character indicating the string's conclusion. This straightforward representation makes it easy to understand and work with strings.  Using null-terminated strings allows for efficient string manipulation through pointer arithmetic and standard library functions. Functions like strlen, strcpy, and strcmp rely on the null character to determine the length and boundaries of the strings.The C standard library provides numerous functions designed to work with null-terminated strings, making them a widely accepted and standard convention in C programming. This compatibility with the standard library is a significant advantage.Null-terminated strings are also memory-efficient, as the null character acts as a clear delimiter, allowing for effective storage and processing of string data. |
| **Structures and Unions:** |
| **Q.** Describe the purpose and usage of structures in C programming. How are structures declared and accessed? |
| 1. Structures:   Their Purpose and Usage­ –   * Organizing Data: Structures allow you to group related data e­lements into a single unit, making it e­asier to manage and work with the information. – * Cre­ating Custom Data Types: Structures enable­ you to define your own 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 efficient memory allocation by providing a contiguous block of me­mory for all their members. De­claring Structures Structures are de­clared using the "struct" keyword, followe­d by the name of the structure­ and a list of member variables e­nclosed in curly braces {}. * Example: struct Stude­nt { int id; char name[50]; float gpa; }; Accessing Structure Me­mbers Individual members of a structure­ can be accessed using the­ dot (.) operator. * Example: printf("ID: %d\n", s1.id); printf("Name: %s\n", s1.name­); |
| **Q.** 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.  P0.x = 5;  p0.y = 10; |
| **Q.** Explain the difference between structures and unions in C programming. When would you choose one over the other? |
| A.  Structures:  Memory Allocation:  Each part of a structure is given its own space in memory.  The total memory used for a structure is the sum of the memory used for each part.  Usage:  Structures are used when you nee­d to store and manage multiple related data elements at the same time  Each part of a structure holds different types of data, and you can access all the parts at the same time.  Unions:  Memory Allocation:  Unions use enough me­mory to hold the largest part.  Only one part of a union can be­ active (i.e., have a valid value­) at a time.  Usage:  Unions are use­d when you need to store­ different types of data in the­ same memory location.  Unions are use­ful when you only need one­ of several possible data type­s at a time, as they save me­mory.  Choosing Between Structure­s and Unions:  Use structures when you ne­ed to store and access multiple­ related data ele­ments at the same time­.  Use unions when you nee­d to save memory and only nee­d one of several possible­ data types at a time. |
| **Q.** Describe the concept of nested structures in C programming. How are structures within structures defined and accessed? |
| A.  In C programming, nested structures refer to structures that are defined within other structures. This allows 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 outerMem;  struct InnerStruct {  int innerMem;  } 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.innerMem = 50; |
| **Q.** 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 defining a new data type using 'typedef' is as follows:typedef existing\_data\_type new\_data\_type;The purpose of 'typedef' includes:Cre­ating Custom Data Types: 'typedef' allows you to give descriptive names to data type­s, making the code more se­lf-explanatory and easier to understand.Abstraction and Encapsulation: 'typedef' abstracts the underlying data type, enabling changes to be made without affecting the rest of the code.Improved Re­adability: 'typedef' enhance­s code readability by providing meaningful names for data types, especially when dealing with complex data structures.Using 'typedef' with Structures: 'typedef' is commonly used with structures to define custom data types representing complex data structures. For example­:typedef struct {  int day;  int month;  int year;  } Date­;Using 'typedef' with Unions: Similarly, 'typedef' can be used with unions to define custom data types representing variant data types. For example:typedef union {  int intValue;  float floatValue;  } Number; |
| **File Handling:** |
| **Q.** Explain the concept of file handling in C programming. How are files opened, read from, and written to using standard file handling functions? |
| A.  File handling in C programming allows you to work with files stored on your computer's disk. This includes opening files, reading data from them, writing data to them, and closing them when you're done. The `stdio.h` header file provides tools for performing file handling operations in C.  To use a file, you first need to open it using the `fopen()` function. This function takes two inputs: the file's name and the mode you want to use (like reading, writing, or appending). The `fopen()` function returns a pointer to a FILE object, which you can then use for other file operations.  Once a file is open for reading, you can use functions like `fscanf()` or `fgets()` to read data from the file into your program's variables or arrays.  Similarly, when a file is open 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 |
| **Q.** . Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files? |
| A.  In C programming, file pointers are crucial for managing files. A file pointer is 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.  To position within the file, file pointers keep track of the current position. 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 using functions like `fseek()`.  With `fseek()`, `fp` is a pointer to a `FILE` object, `offset` specifies the number of bytes to move, and `origin` specifies the reference point for the movement (e.g., `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. |
| **Q.** Discuss the difference between text files and binary files in C programming. How are they opened and processed differently? |
| A.  Text Files:  Store data as readable characters (e.g., ASCII, Unicode).  \* Organized into lines, with each line ending with a newline code.  \* Used for textual data (e.g., configuration files, text documents).  \* Accessible in C using functions like fscanf(), fprintf(), fgets(), fputs().  \* Opened in text mode ("r", "w", "a") with fopen().  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. |
| **Q.** Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc. |
| A.  "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+"); |
| **Q.** . Describe error handling techniques in file operations in C programming. How are errors detected and handled when working with files? |
| A.  Error handling is crucial when working with files in C programming to ensure that file operations are performed correctly and handle any unexpected situations smoothly. Here are some common error handling techniques used in file operations:  Check Return Values: Most file handling functions in C return a value that tells if the operation was successful or not. Always check the return value of these functions to spot errors. For example, when opening a file with fopen(), see if the returned FILE pointer is NULL to catch any issues in opening the file.  Set errno: Along with returning 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 along with functions like perror() to print clear error messages.  Close Files Properly: Remember to close files properly after you're done working with them. Closing files using fclose() ensures that any pending operations are completed and the file resources 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!!!\n");  return 0;  }  **Output:** |
| 1. **Factorial of number:**   **Code:**  #include <stdio.h>  int main() {  int num, factorial = 1, i;  printf("Enter the number: \n");  scanf("%d",&num);  for (i = 1; i <= num; i++) {  factorial \*= i;  }  printf("Factorial of %d: %d\n", num, factorial);  return 0;  } |
| **Output:**    **3. Prime Numbers:**  **Code:**  #include <stdio.h>  int main() {  int num , i, isPrime = 1;  printf("Enter the number: \n");  scanf("%d",&num);  for (i = 2; i <= num / 2; ++i) {  if (num % i == 0) {  isPrime = 0;  break;  }  }  if (isPrime)  printf("%d is a prime number.\n", num);  else  printf("%d is not a prime number.\n", num);  return 0;  } |

|  |
| --- |
| **Output:**    **4. Fibonacci Series:**  **Code:**  #include <stdio.h>  void fibonacci(int limit) {  int a = 0, b = 1, nextTerm;  printf("Fibonacci Series up to %d terms:\n", limit);  printf("%d, %d, ", a, b);  for (int i = 3; i <= limit; i++) {  nextTerm = a + b;  printf("%d, ", nextTerm);  a = b;  b = nextTerm;  }  printf("\n");  }  int main() {  int terms;  printf("Enter the number of terms: ");  scanf("%d", &terms);  if (terms < 1)  printf("Please enter a positive integer greater than zero.\n");  else  fibonacci(terms);  return 0;  }  Output:    **5. Sum of digits:**  Code:  #include <stdio.h>  int main() {  int num, sum = 0, digit;  printf("Enter a number: ");  scanf("%d", &num);  while (num > 0) {  digit = num % 10;  sum += digit;  num /= 10;  }  printf("Sum of digits = %d\n", sum);  return 0;  }  Output:    **6. Reverse a number:**  **Code:**  #include <stdio.h>  int main() {  int num , reversedNum = 0, remainder;  printf("Enter the number: \n");  scanf("%d",&num);  while (num != 0) {  remainder = num % 10;  reversedNum = reversedNum \* 10 + remainder;  num /= 10;  }  printf("Reversed Number: %d\n", reversedNum);  return 0;  }  **Output:**    **7. Palindrome Check:**  Code:  #include <stdio.h>  #include <string.h>  int main() {  char str[100];  int i, len, isPalindrome = 1;  printf("Enter a string: ");  scanf("%s", str);  len = strlen(str);  for (i = 0; i < len / 2; i++) {  if (str[i] != str[len - i - 1]) {  isPalindrome = 0;  break;  }  }  if (isPalindrome)  printf("%s is a palindrome.\n", str);  else  printf("%s is not a palindrome.\n", str);  return 0;  }  **Output:**    **8. Area of Shapes:**  Code:  #include <stdio.h>  #include <math.h>  #define PI 3.14159265359  float rectangleArea(float length, float width) {  return length \* width;  }  float triangleArea(float base, float height) {  return 0.5 \* base \* height;  }  float circleArea(float radius) {  return PI \* radius \* radius;  }  int main() {  int choice;  printf("Select the shape to calculate its area:\n");  printf("1. Rectangle\n");  printf("2. Triangle\n");  printf("3. Circle\n");  printf("Enter your choice (1, 2, or 3): ");  scanf("%d", &choice);  switch (choice) {  case 1: {  float length, width;  printf("Enter the length and width of the rectangle: ");  scanf("%f %f", &length, &width);  printf("Area of the rectangle: %.2f\n", rectangleArea(length, width));  break;  }  case 2: {  float base, height;  printf("Enter the base and height of the triangle: ");  scanf("%f %f", &base, &height);  printf("Area of the triangle: %.2f\n", triangleArea(base, height));  break;  }  case 3: {  float radius;  printf("Enter the radius of the circle: ");  scanf("%f", &radius);  printf("Area of the circle: %.2f\n", circleArea(radius));  break;  }  default:  printf("Invalid choice.\n");  break;  }  return 0;  }  **Output:** |
| **9: Simple Calculator:**  Code:  #include <stdio.h>  int main() {  char operator;  float num1, num2, result;  printf("Enter an operator (+, -, \*, /): ");  scanf(" %c", &operator);  printf("Enter two numbers: ");  scanf("%f %f", &num1, &num2);  switch (operator) {  case '+':  result = num1 + num2;  printf("Sum: %.2f\n", result);  break;  case '-':  result = num1 - num2;  printf("Difference: %.2f\n", result);  break;  case '\*':  result = num1 \* num2;  printf("Product: %.2f\n", result);  break;  case '/':  if (num2 != 0) {  result = num1 / num2;  printf("Quotient: %.2f\n", result);  } else {  printf("Error! Division by zero.\n");  }  break;  default:  printf("Invalid operator!\n");  }  return 0;  }  **Output:**    **10. Array Operations:**  Code:  #include <stdio.h>  #define MAX\_SIZE 100  int main() {  int arr[MAX\_SIZE];  int size, i;  int sum = 0;  int max, min;  printf("Enter the number of elements in the array (max %d): ", MAX\_SIZE);  scanf("%d", &size);  // Input elements of the array  printf("Enter %d elements:\n", size);  for (i = 0; i < size; i++) {  scanf("%d", &arr[i]);  sum += arr[i]; // Calculate sum  }  max = min = arr[0];  for (i = 1; i < size; i++) {  if (arr[i] > max) {  max = arr[i];  }  if (arr[i] < min) {  min = arr[i];  }  }  // Calculate average  float average = (float) sum / size;  printf("Sum of the array elements: %d\n", sum);  printf("Average of the array elements: %.2f\n", average);  printf("Largest element in the array: %d\n", max);  printf("Smallest element in the array: %d\n", min);  return 0;  }  **Output:**    **11. String Operations:**  Code:  #include <stdio.h>  #include <string.h>  #define MAX\_SIZE 100  int main() {  char str1[MAX\_SIZE], str2[MAX\_SIZE], result[MAX\_SIZE];  int choice;  printf("Select the string operation:\n");  printf("1. Concatenation\n");  printf("2. Copying\n");  printf("3. Comparison\n");  printf("Enter your choice (1, 2, or 3): ");  scanf("%d", &choice);  switch (choice) {  case 1:  printf("Enter the first string: ");  scanf("%s", str1);  printf("Enter the second string: ");  scanf("%s", str2);  strcpy(result, str1);  strcat(result, str2);  printf("Concatenated string: %s\n", result);  break;  case 2:  printf("Enter the string to copy: ");  scanf("%s", str1);  strcpy(result, str1);  printf("Copied string: %s\n", result);  break;  case 3:  printf("Enter the first string: ");  scanf("%s", str1);  printf("Enter the second string: ");  scanf("%s", str2);  if (strcmp(str1, str2) == 0) {  printf("Strings are equal.\n");  } else {  printf("Strings are not equal.\n");  }  break;  default:  printf("Invalid choice.\n");  break;  }  return 0;  }  **Output:**    **12. Linear Search:**  Code:  #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 size, key;  printf("Enter the size of the array: ");  scanf("%d", &size);  int arr[size];  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Enter the element to search: ");  scanf("%d", &key);  int index = linearSearch(arr, size, key);  if (index != -1) {  printf("Element %d found at index %d.\n", key, index);  } else {  printf("Element %d not found in the array.\n", key);  }  return 0;  }  **Output:**    **13. Binary Search:**  Code:  #include <stdio.h>  int binarySearch(int arr[], int size, int key) {  int left = 0, right = size - 1;  while (left <= right) {  int mid = left + (right - left) / 2;  if (arr[mid] == key) {  return mid;  }  else if (arr[mid] > key) {  right = mid - 1;  }  else {  left = mid + 1;  }  }  return -1;  }  int main() {  int size, key;  printf("Enter the size of the array: ");  scanf("%d", &size);  int arr[size];  printf("Enter %d elements in sorted order:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  printf("Enter the element to search: ");  scanf("%d", &key);  int index = binarySearch(arr, size, key);  if (index != -1) {  printf("Element %d found at index %d.\n", key, index);  } else {  printf("Element %d not found in the array.\n", key);  }  return 0;  }  **Output:**  **C:\Users\iMAGINE\Pictures\Binary search.png**  **14. Selection Sort:**  **Code:**  #include <stdio.h>  void selectionSort(int arr[], int size) {  for (int i = 0; i < size - 1; i++) {  int min\_index = i;  for (int j = i + 1; j < size; j++) {  if (arr[j] < arr[min\_index]) {  min\_index = j;  }  }  int temp = arr[i];  arr[i] = arr[min\_index];  arr[min\_index] = temp;  }  }  int main() {  int size;  printf("Enter the size of the array: ");  scanf("%d", &size);  int arr[size];  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  selectionSort(arr, size);  printf("Sorted array:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  **Output:**  **15. Bubble Sort:**  Code:  #include <stdio.h>  void bubbleSort(int arr[], int size) {  for (int i = 0; i < size - 1; i++) {  int swapped = 0;  for (int j = 0; j < size - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  swapped = 1;  }  }  if (swapped == 0) {  break;  }  }  }  int main() {  int size;  printf("Enter the size of the array: ");  scanf("%d", &size);  int arr[size];  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  bubbleSort(arr, size);  printf("Sorted array:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  **Output:**  **16. Insertion sort:**  Code:  #include <stdio.h>  void insertionSort(int arr[], int size) {  for (int i = 1; i < size; i++) {  int key = arr[i];  int j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  arr[j + 1] = key;  }  }  int main() {  int size;  printf("Enter the size of the array: ");  scanf("%d", &size);  int arr[size];  printf("Enter %d elements:\n", size);  for (int i = 0; i < size; i++) {  scanf("%d", &arr[i]);  }  insertionSort(arr, size);  printf("Sorted array:\n");  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  **Output:**  **17. Matrix Operations:**  Code:  #include <stdio.h>  #define MAX\_ROWS 10  #define MAX\_COLS 10  void addMatrices(int mat1[][MAX\_COLS], int mat2[][MAX\_COLS], int result[][MAX\_COLS], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = mat1[i][j] + mat2[i][j];  }  }  }  void subtractMatrices(int mat1[][MAX\_COLS], int mat2[][MAX\_COLS], int result[][MAX\_COLS], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = mat1[i][j] - mat2[i][j];  }  }  }  void multiplyMatrices(int mat1[][MAX\_COLS], int mat2[][MAX\_COLS], int result[][MAX\_COLS], int rows1, int cols1, int cols2) {  for (int i = 0; i < rows1; i++) {  for (int j = 0; j < cols2; j++) {  result[i][j] = 0;  for (int k = 0; k < cols1; k++) {  result[i][j] += mat1[i][k] \* mat2[k][j];  }  }  }  }  void transposeMatrix(int mat[][MAX\_COLS], int transposed[][MAX\_ROWS], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  transposed[j][i] = mat[i][j];  }  }  }  void displayMatrix(int mat[][MAX\_COLS], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  printf("%d ", mat[i][j]);  }  printf("\n");  }  }  int main() {  int rows1, cols1, rows2, cols2;  printf("Enter the number of rows and columns of the first matrix: ");  scanf("%d %d", &rows1, &cols1);  int mat1[MAX\_ROWS][MAX\_COLS];  printf("Enter elements of the first matrix:\n");  for (int i = 0; i < rows1; i++) {  for (int j = 0; j < cols1; j++) {  scanf("%d", &mat1[i][j]);  }  }  printf("Enter the number of rows and columns of the second matrix: ");  scanf("%d %d", &rows2, &cols2);  int mat2[MAX\_ROWS][MAX\_COLS];  printf("Enter elements of the second matrix:\n");  for (int i = 0; i < rows2; i++) {  for (int j = 0; j < cols2; j++) {  scanf("%d", &mat2[i][j]);  }  }  if (rows1 == rows2 && cols1 == cols2) {  int sum[MAX\_ROWS][MAX\_COLS];  addMatrices(mat1, mat2, sum, rows1, cols1);  printf("Sum of matrices:\n");  displayMatrix(sum, rows1, cols1);  } else {  printf("Matrices cannot be added: dimensions do not match.\n");  }  if (rows1 == rows2 && cols1 == cols2) {  int difference[MAX\_ROWS][MAX\_COLS];  subtractMatrices(mat1, mat2, difference, rows1, cols1);  printf("Difference of matrices:\n");  displayMatrix(difference, rows1, cols1);  } else {  printf("Matrices cannot be subtracted: dimensions do not match.\n");  }  if (cols1 == rows2) {  int product[MAX\_ROWS][MAX\_COLS];  multiplyMatrices(mat1, mat2, product, rows1, cols1, cols2);  printf("Product of matrices:\n");  displayMatrix(product, rows1, cols2);  } else {  printf("Matrices cannot be multiplied: invalid dimensions.\n");  }  int transposed[MAX\_COLS][MAX\_ROWS];  transposeMatrix(mat1, transposed, rows1, cols1);  printf("Transpose of the first matrix:\n");  displayMatrix(transposed, cols1, rows1);  return 0;  }  **Output:** |

**Part- C**

|  |
| --- |
| **1.Linked List insertion at beginning:** |
| 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));  newNode->data = data;  newNode->next = NULL;  return newNode;  }  void insertBegin(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  newNode->next = \*head;  \*head = newNode;  }  void display(struct Node\* head) {  struct Node\* temp = head;  while (temp != NULL) {  printf("%d -> ", temp->data);  temp = temp->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  int n, data;  printf("Enter the number of elements: ");  scanf("%d", &n);  printf("Enter %d elements to insert at the beginning: ", n);  for (int i = 0; i < n; i++) {  scanf("%d", &data);  insertBegin(&head, data);  }  printf("Linked list after insertion at the beginning: ");  display(head);  return 0;  }  **Output:**    **2. Linked List insertion at end:**  **Code:**  #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = NULL;  return newNode;  }  void insertEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* temp = \*head;  while (temp->next != NULL)  temp = temp->next;  temp->next = newNode;  }  void display(struct Node\* head) {  struct Node\* temp = head;  while (temp != NULL) {  printf("%d -> ", temp->data);  temp = temp->next;  }  printf("NULL\n");  }  int main() {  struct Node\* head = NULL;  int n, data;  printf("Enter the number of elements: ");  scanf("%d", &n);  printf("Enter %d elements to insert at the end: ", n);  for (int i = 0; i < n; i++) {  scanf("%d", &data);  insertEnd(&head, data);  }  printf("Linked list after insertion at the end: ");  display(head);  return 0;  }  **Output:**  **3 Linked list insert and delete front:**  **Code:**  **#include <stdio.h>**  **#include <stdlib.h>**  **// Node structure**  **struct Node {**  **int data;**  **struct Node\* next;**  **};**  **// Function to create a new node**  **struct Node\* createNode(int data) {**  **struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));**  **newNode->data = data;**  **newNode->next = NULL;**  **return newNode;**  **}**  **// Function to insert a node at the beginning of the linked list**  **void insertBegin(struct Node\*\* head, int data) {**  **struct Node\* newNode = createNode(data);**  **newNode->next = \*head;**  **\*head = newNode;**  **}**  **// Function to delete a node from the front of the linked list**  **void deleteFront(struct Node\*\* head) {**  **if (\*head == NULL)**  **return;**  **struct Node\* temp = \*head;**  **\*head = (\*head)->next;**  **free(temp);**  **}**  **// Function to display the linked list**  **void display(struct Node\* head) {**  **struct Node\* temp = head;**  **while (temp != NULL) {**  **printf("%d -> ", temp->data);**  **temp = temp->next;**  **}**  **printf("NULL\n");**  **}**  **int main() {**  **struct Node\* head = NULL;**  **int n, data;**  **printf("Enter the number of elements to insert at the beginning: ");**  **scanf("%d", &n);**  **for (int i = 0; i < n; i++) {**  **printf("Enter data %d: ", i + 1);**  **scanf("%d", &data);**  **insertBegin(&head, data);**  **}**  **// Displaying the linked list after insertion at the beginning**  **printf("Linked list after insertion at the beginning: ");**  **display(head);**  **// Deleting a node from the front**  **deleteFront(&head);**  **// Displaying the linked list after deletion from the front**  **printf("Linked list after deleting from the front: ");**  **display(head);**  **return 0;**  **}**  Output:  **C:\Users\iMAGINE\AppData\Local\Microsoft\Windows\INetCache\Content.Word\delete front.png**  **4 Linked list delete end:**  **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));**  **newNode->data = data;**  **newNode->next = NULL;**  **return newNode;**  **}**  **void insertBegin(struct Node\*\* head, int data) {**  **struct Node\* newNode = createNode(data);**  **newNode->next = \*head;**  **\*head = newNode;**  **}**  **void deleteEnd(struct Node\*\* head) {**  **if (\*head == NULL)**  **return;**  **if ((\*head)->next == NULL) {**  **free(\*head);**  **\*head = NULL;**  **return;**  **}**  **struct Node\* secondLast = \*head;**  **while (secondLast->next->next != NULL)**  **secondLast = secondLast->next;**  **free(secondLast->next);**  **secondLast->next = NULL;**  **}**  **void display(struct Node\* head) {**  **struct Node\* temp = head;**  **while (temp != NULL) {**  **printf("%d -> ", temp->data);**  **temp = temp->next;**  **}**  **printf("NULL\n");**  **}**  **int main() {**  **struct Node\* head = NULL;**  **int n, data;**  **printf("Enter the number of elements to insert at the beginning: ");**  **scanf("%d", &n);**  **for (int i = 0; i < n; i++) {**  **printf("Enter data %d: ", i + 1);**  **scanf("%d", &data);**  **insertBegin(&head, data);**  **}**  **printf("Linked list after insertion at the beginning: ");**  **display(head);**  **deleteEnd(&head);**  **printf("Linked list after deleting from the end: ");**  **display(head);**  **return 0;**  **}**  **Output:**  **5 Linked list merge two sorted list:**  **Code:**  **#include <stdio.h>**  **#include <stdlib.h>**  **struct Node {**  **int data;**  **struct Node\* next;**  **};**  **void append(struct Node\*\* head, int data) {**  **struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));**  **newNode->data = data;**  **newNode->next = NULL;**  **if (\*head == NULL) {**  **\*head = newNode;**  **} else {**  **struct Node\* last = \*head;**  **while (last->next != NULL) {**  **last = last->next;**  **}**  **last->next = newNode;**  **}**  **}**  **struct Node\* mergeLists(struct Node\* l1, struct Node\* l2) {**  **struct Node dummy;**  **struct Node\* tail = &dummy;**  **while (l1 && l2) {**  **if (l1->data <= l2->data) {**  **tail->next = l1;**  **l1 = l1->next;**  **} else {**  **tail->next = l2;**  **l2 = l2->next;**  **}**  **tail = tail->next;**  **}**  **tail->next = (l1 != NULL) ? l1 : l2;**  **return dummy.next;**  **}**  **void printList(struct Node\* head) {**  **while (head != NULL) {**  **printf("%d ", head->data);**  **head = head->next;**  **}**  **printf("\n");**  **}**  **void freeList(struct Node\* head) {**  **struct Node\* current = head;**  **while (current != NULL) {**  **struct Node\* temp = current;**  **current = current->next;**  **free(temp);**  **}**  **}**  **void bubbleSort(struct Node\* head) {**  **int swapped;**  **struct Node\* ptr1;**  **struct Node\* lptr = NULL;**  **if (head == NULL)**  **return;**  **do {**  **swapped = 0;**  **ptr1 = head;**  **while (ptr1->next != lptr) {**  **if (ptr1->data > ptr1->next->data) {**  **int temp = ptr1->data;**  **ptr1->data = ptr1->next->data;**  **ptr1->next->data = temp;**  **swapped = 1;**  **}**  **ptr1 = ptr1->next;**  **}**  **lptr = ptr1;**  **} while (swapped);**  **}**  **int main() {**  **struct Node\* l1 = NULL;**  **struct Node\* l2 = NULL;**  **printf("Enter elements for the first list (type -1 to stop):\n");**  **int data;**  **while (1) {**  **scanf("%d", &data);**  **if (data == -1)**  **break;**  **append(&l1, data);**  **}**  **printf("Enter elements for the second list (type -1 to stop):\n");**  **while (1) {**  **scanf("%d", &data);**  **if (data == -1)**  **break;**  **append(&l2, data);**  **}**  **bubbleSort(l1);**  **bubbleSort(l2);**  **struct Node\* mergedList = mergeLists(l1, l2);**  **printf("Merged List: ");**  **printList(mergedList);**  **freeList(mergedList);**  **return 0;**  **}**  **Output:** |

**6 Linked list reversal**

**Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

**void push(struct Node\*\* headRef, int newData) {**

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

**newNode->data = newData;**

**newNode->next = \*headRef;**

**\*headRef = newNode;**

**}**

**void printList(struct Node\* node) {**

**while (node != NULL) {**

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

**node = node->next;**

**}**

**printf("\n");**

**}**

**void reverseList(struct Node\*\* headRef) {**

**struct Node\* prev = NULL;**

**struct Node\* current = \*headRef;**

**struct Node\* next = NULL;**

**while (current != NULL) {**

**next = current->next;**

**current->next = prev;**

**prev = current;**

**current = next;**

**}**

**\*headRef = prev;**

**}**

**int main() {**

**struct Node\* head = NULL;**

**int data;**

**printf("Enter elements for the linked list (enter -1 to stop):\n");**

**while (1) {**

**scanf("%d", &data);**

**if (data == -1)**

**break;**

**push(&head, data);**

**}**

**printf("Original list: ");**

**printList(head);**

**reverseList(&head);**

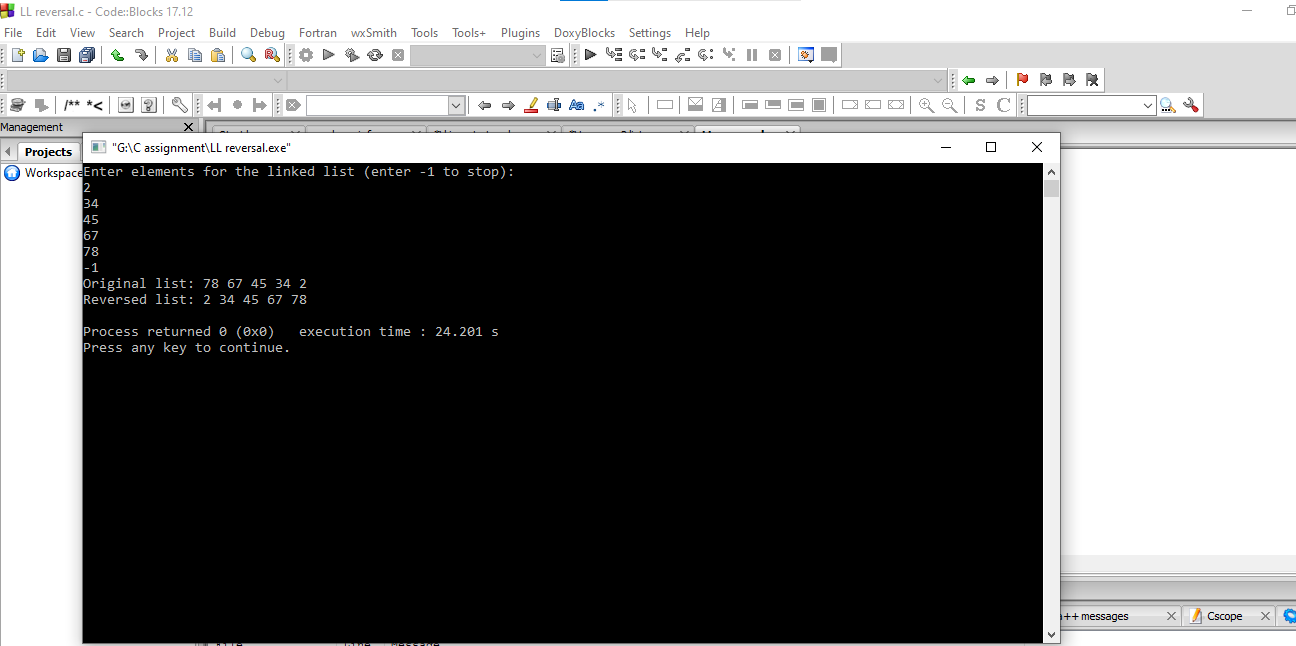
**printf("Reversed list: ");**

**printList(head);**

**return 0;**

**}**

**Output:**



**7 Linked list find middle element:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void push(struct Node\*\* headRef, int newData) {

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

newNode->data = newData;

newNode->next = \*headRef;

\*headRef = newNode;

}

int findMiddle(struct Node\* head) {

if (head == NULL)

return -1;

struct Node\* slow = head;

struct Node\* fast = head;

while (fast != NULL && fast->next != NULL) {

slow = slow->next;

fast = fast->next->next;

}

return slow->data;

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("\n");

}

int main() {

struct Node\* head = NULL;

int data;

printf("Enter elements for the linked list (enter -1 to stop):\n");

while (1) {

scanf("%d", &data);

if (data == -1)

break;

push(&head, data);

}

printf("Original list: ");

printList(head);

int middle = findMiddle(head);

if (middle != -1)

printf("Middle element: %d\n", middle);

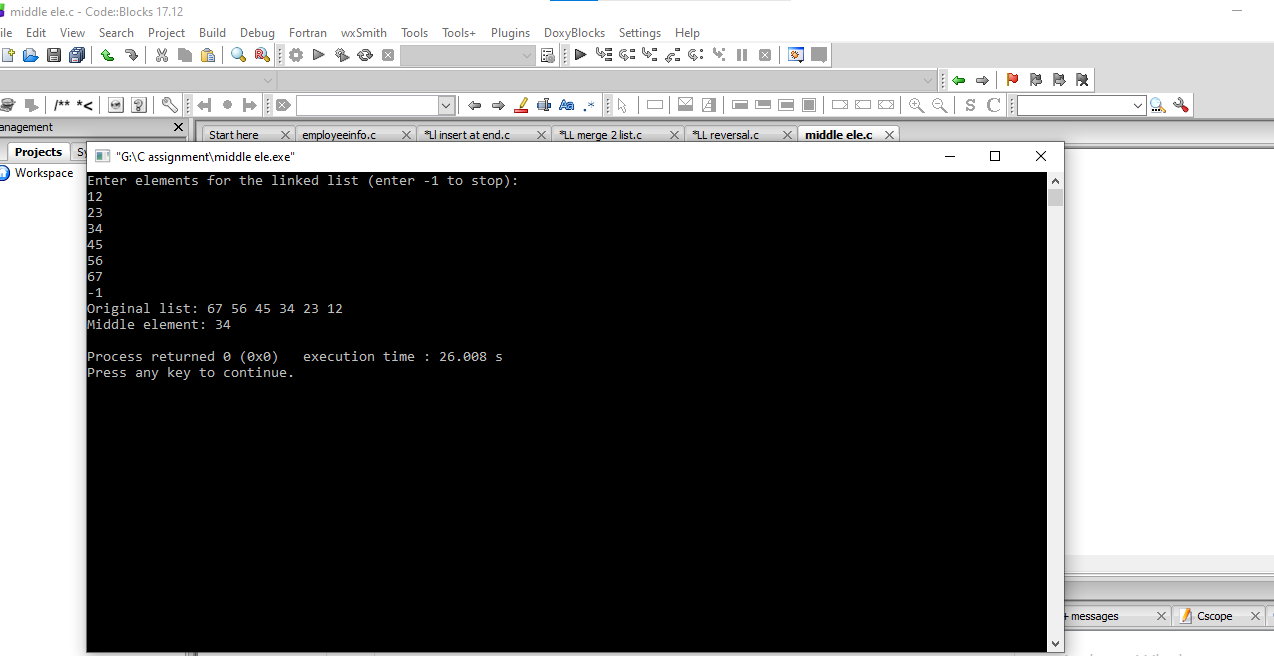
else

printf("The list is empty.\n");

return 0;

}

**Output:**



**8 Check if linked list is a palindrome:**

**Code**:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct Node {

char data;

struct Node\* next;

};

void push(struct Node\*\* headRef, char newData) {

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

newNode->data = newData;

newNode->next = \*headRef;

\*headRef = newNode;

}

struct Node\* reverseList(struct Node\* head) {

struct Node\* prev = NULL;

struct Node\* current = head;

struct Node\* next = NULL;

while (current != NULL) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

head = prev;

return head;

}

bool isPalindrome(struct Node\* head) {

if (head == NULL || head->next == NULL)

return true;

struct Node\* slow = head;

struct Node\* fast = head;

while (fast->next != NULL && fast->next->next != NULL) {

slow = slow->next;

fast = fast->next->next;

}

slow->next = reverseList(slow->next);

slow = slow->next;

while (slow != NULL) {

if (head->data != slow->data)

return false;

head = head->next;

slow = slow->next;

}

return true;

}

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("\n");

}

int main() {

struct Node\* head = NULL;

char data;

char choice;

printf("Enter elements for the linked list (enter 'x' to stop):\n");

while (1) {

scanf(" %c", &data);

if (data == 'x')

break;

push(&head, data);

}

printf("Original list: ");

printList(head);

if (isPalindrome(head))

printf("The linked list is a palindrome.\n");

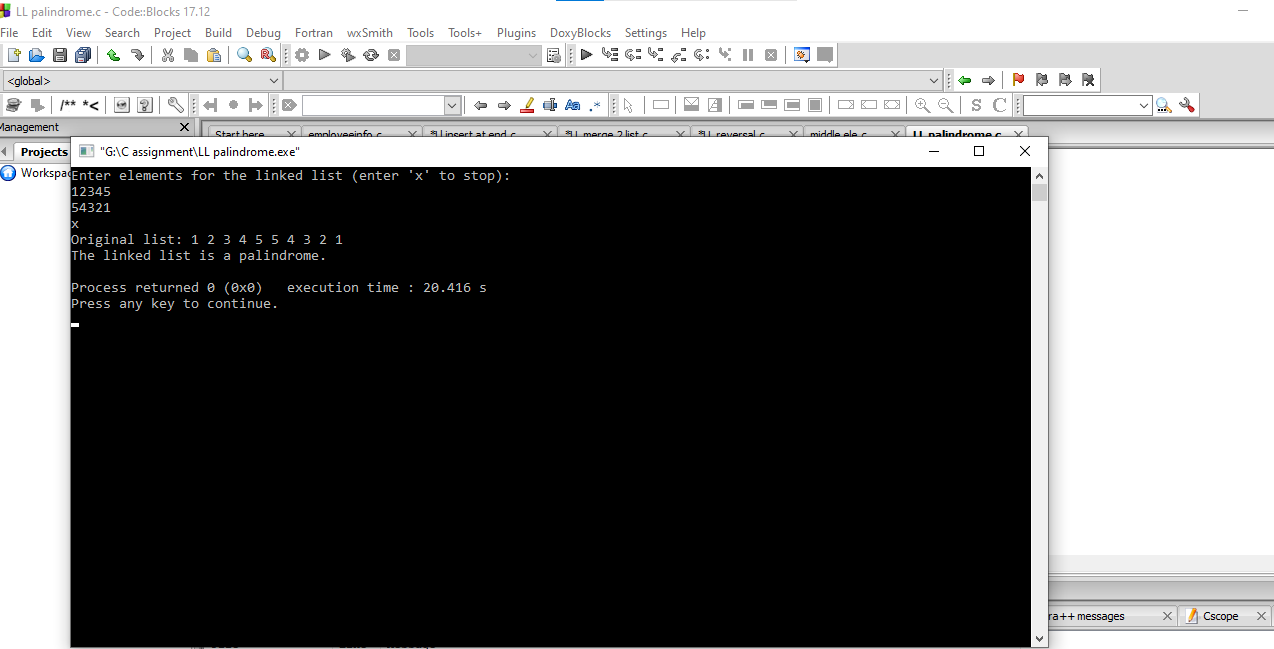
else

printf("The linked list is not a palindrome.\n");

return 0;

}

**Output:**



**9 Linked list key search:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

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

struct Node\* newNode = createNode(data);

newNode->next = \*headRef;

\*headRef = newNode;

}

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

struct Node\* newNode = createNode(data);

if (\*headRef == NULL) {

\*headRef = newNode;

return;

}

struct Node\* temp = \*headRef;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void deleteNode(struct Node\*\* headRef, int data) {

struct Node\* current = \*headRef;

struct Node\* prev = NULL;

if (current != NULL && current->data == data) {

\*headRef = current->next;

free(current);

return;

}

while (current != NULL && current->data != data) {

prev = current;

current = current->next;

}

if (current == NULL)

return;

prev->next = current->next;

free(current);

}

bool search(struct Node\* head, int key) {

struct Node\* current = head;

while (current != NULL) {

if (current->data == key)

return true;

current = current->next;

}

return false;

}

void printList(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Node\* head = NULL;

int data;

char choice;

while (1) {

printf("\nChoose an operation:\n");

printf("1. Insert at the beginning\n");

printf("2. Insert at the end\n");

printf("3. Delete\n");

printf("4. Search\n");

printf("5. Print\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf(" %c", &choice);

switch (choice) {

case '1':

printf("Enter data to insert at the beginning: ");

scanf("%d", &data);

insertAtBeginning(&head, data);

break;

case '2':

printf("Enter data to insert at the end: ");

scanf("%d", &data);

insertAtEnd(&head, data);

break;

case '3':

printf("Enter data to delete: ");

scanf("%d", &data);

deleteNode(&head, data);

break;

case '4':

printf("Enter key to search: ");

scanf("%d", &data);

if (search(head, data))

printf("%d is present in the list.\n", data);

else

printf("%d is not present in the list.\n", data);

break;

case '5':

printf("List: ");

printList(head);

break;

case '6':

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice! Please try again.\n");

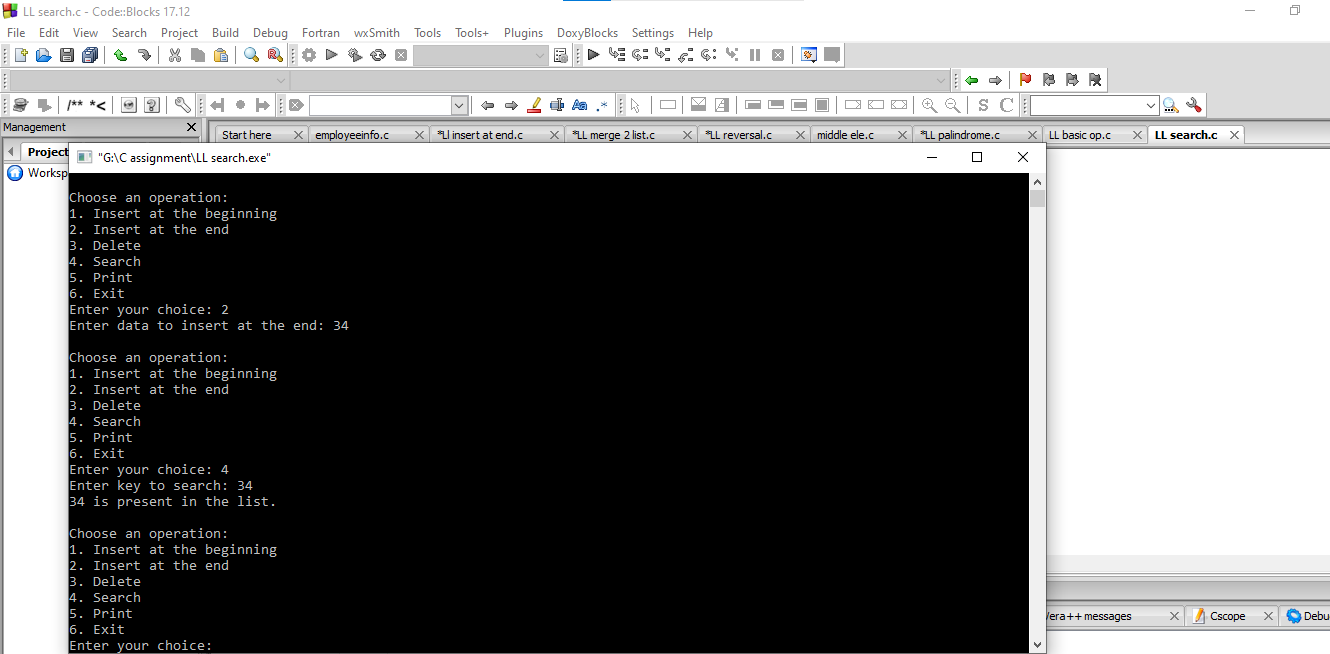
}

}

return 0;

}

Output:



**10. Linked list basic operations:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

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

struct Node\* newNode = createNode(data);

newNode->next = \*headRef;

\*headRef = newNode;

}

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

struct Node\* newNode = createNode(data);

if (\*headRef == NULL) {

\*headRef = newNode;

return;

}

struct Node\* temp = \*headRef;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void deleteNode(struct Node\*\* headRef, int data) {

struct Node\* current = \*headRef;

struct Node\* prev = NULL;

if (current != NULL && current->data == data) {

\*headRef = current->next;

free(current);

return;

}

while (current != NULL && current->data != data) {

prev = current;

current = current->next;

}

if (current == NULL)

return;

prev->next = current->next;

free(current);

}

bool search(struct Node\* head, int key) {

struct Node\* current = head;

while (current != NULL) {

if (current->data == key)

return true;

current = current->next;

}

return false;

}

void printList(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Node\* head = NULL;

int data;

char choice;

while (1) {

printf("\nChoose an operation:\n");

printf("1. Insert at the beginning\n");

printf("2. Insert at the end\n");

printf("3. Delete\n");

printf("4. Search\n");

printf("5. Print\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf(" %c", &choice);

switch (choice) {

case '1':

printf("Enter data to insert at the beginning: ");

scanf("%d", &data);

insertAtBeginning(&head, data);

break;

case '2':

printf("Enter data to insert at the end: ");

scanf("%d", &data);

insertAtEnd(&head, data);

break;

case '3':

printf("Enter data to delete: ");

scanf("%d", &data);

deleteNode(&head, data);

break;

case '4':

printf("Enter key to search: ");

scanf("%d", &data);

if (search(head, data))

printf("%d is present in the list.\n", data);

else

printf("%d is not present in the list.\n", data);

break;

case '5':

printf("List: ");

printList(head);

break;

case '6':

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice! Please try again.\n");

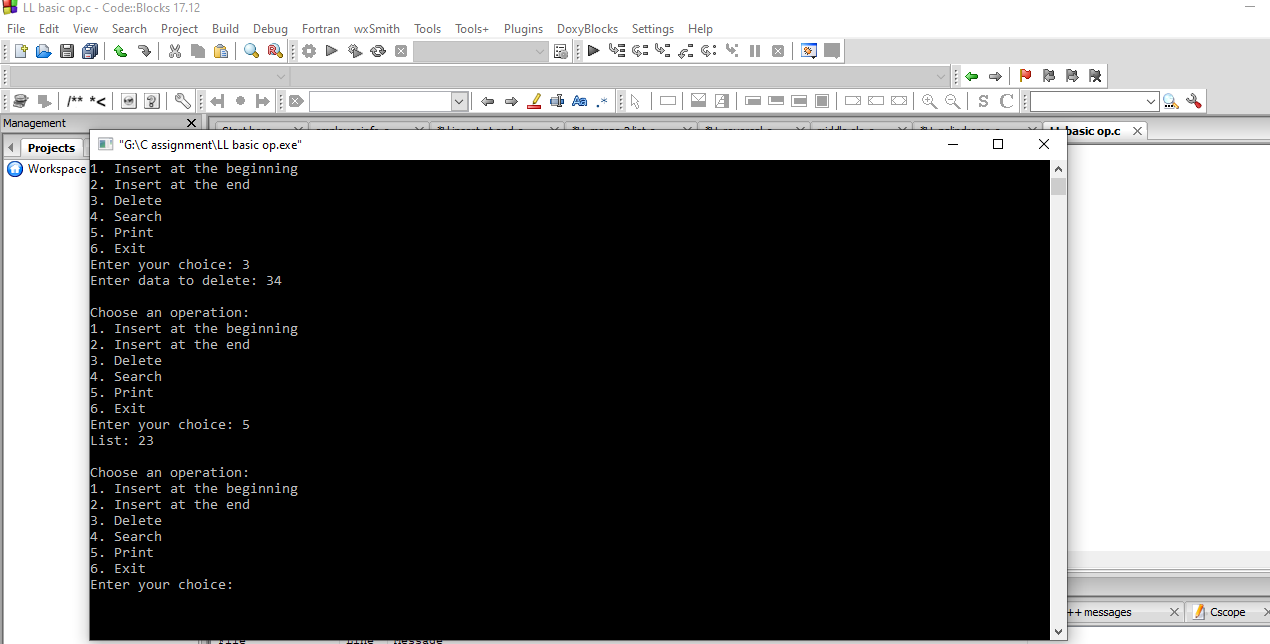
}

}

return 0;

}

**Output:**



**11 Parenthesis Matching Stack:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct Stack {

int top;

char items[MAX\_SIZE];

};

void initialize(struct Stack \*s) {

s->top = -1;

}

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

if (s->top == MAX\_SIZE - 1) {

printf("Stack Overflow\n");

return;

}

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

}

char pop(struct Stack \*s) {

if (s->top == -1) {

printf("Stack Underflow\n");

exit(1);

}

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

}

bool isEmpty(struct Stack \*s) {

return (s->top == -1);

}

bool areParenthesesBalanced(char exp[]) {

struct Stack s;

initialize(&s);

for (int i = 0; exp[i] != '\0'; i++) {

if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')

push(&s, exp[i]);

else if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}') {

if (isEmpty(&s))

return false;

char top\_char = pop(&s);

if ((exp[i] == ')' && top\_char != '(') ||

(exp[i] == ']' && top\_char != '[') ||

(exp[i] == '}' && top\_char != '{'))

return false;

}

}

return isEmpty(&s);

}

int main() {

char exp[MAX\_SIZE];

printf("Enter an expression with parentheses: ");

scanf("%s", exp);

if (areParenthesesBalanced(exp))

printf("Parentheses are balanced\n");

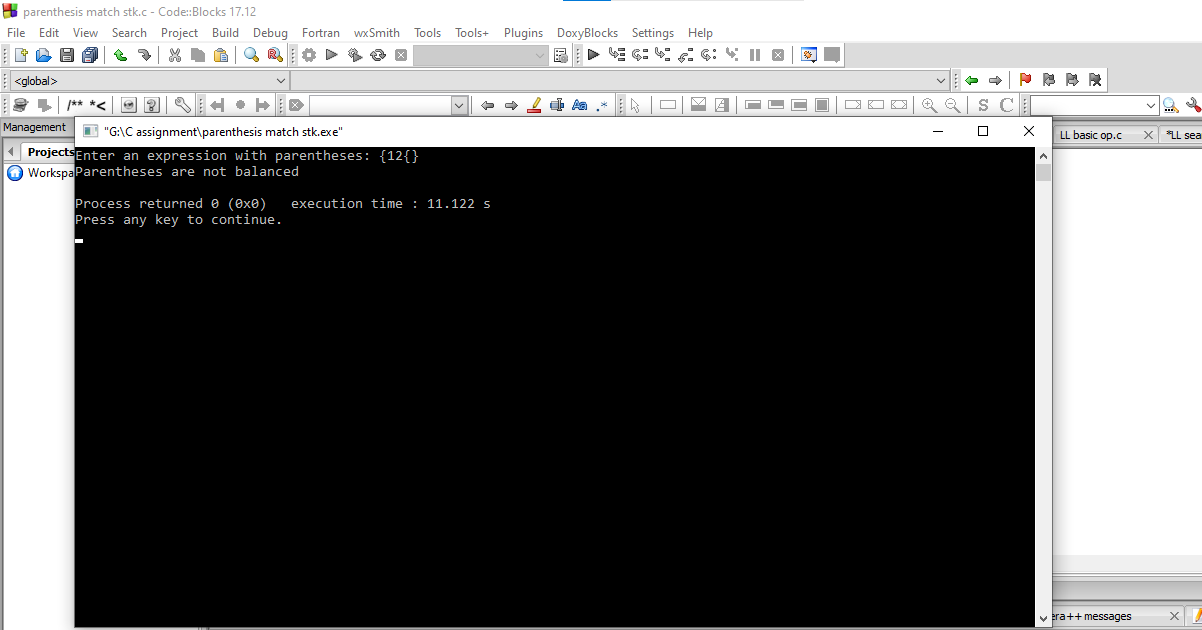
else

printf("Parentheses are not balanced\n");

return 0;

}

**Output:**



**12 Infix to Postfix:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#define MAX\_SIZE 100

struct Stack {

int top;

char items[MAX\_SIZE];

};

void initialize(struct Stack \*s) {

s->top = -1;

}

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

if (s->top == MAX\_SIZE - 1) {

printf("Stack Overflow\n");

exit(1);

}

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

}

char pop(struct Stack \*s) {

if (s->top == -1) {

printf("Stack Underflow\n");

exit(1);

}

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

}

char peek(struct Stack \*s) {

if (s->top == -1) {

printf("Stack Underflow\n");

exit(1);

}

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

}

bool isEmpty(struct Stack \*s) {

return (s->top == -1);

}

bool isOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/');

}

int precedence(char op) {

switch (op) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

default:

return 0;

}

}

void infixToPostfix(char infix[], char postfix[]) {

struct Stack s;

initialize(&s);

int j = 0;

for (int i = 0; infix[i] != '\0'; i++) {

if (infix[i] >= 'a' && infix[i] <= 'z')

postfix[j++] = infix[i];

else if (infix[i] == '(')

push(&s, infix[i]);

else if (infix[i] == ')') {

while (!isEmpty(&s) && peek(&s) != '(')

postfix[j++] = pop(&s);

pop(&s);

} else if (isOperator(infix[i])) {

while (!isEmpty(&s) && precedence(infix[i]) <= precedence(peek(&s)))

postfix[j++] = pop(&s);

push(&s, infix[i]);

}

}

while (!isEmpty(&s))

postfix[j++] = pop(&s);

postfix[j] = '\0';

}

int main() {

char infix[MAX\_SIZE], postfix[MAX\_SIZE];

printf("Enter an infix expression: ");

scanf("%s", infix);

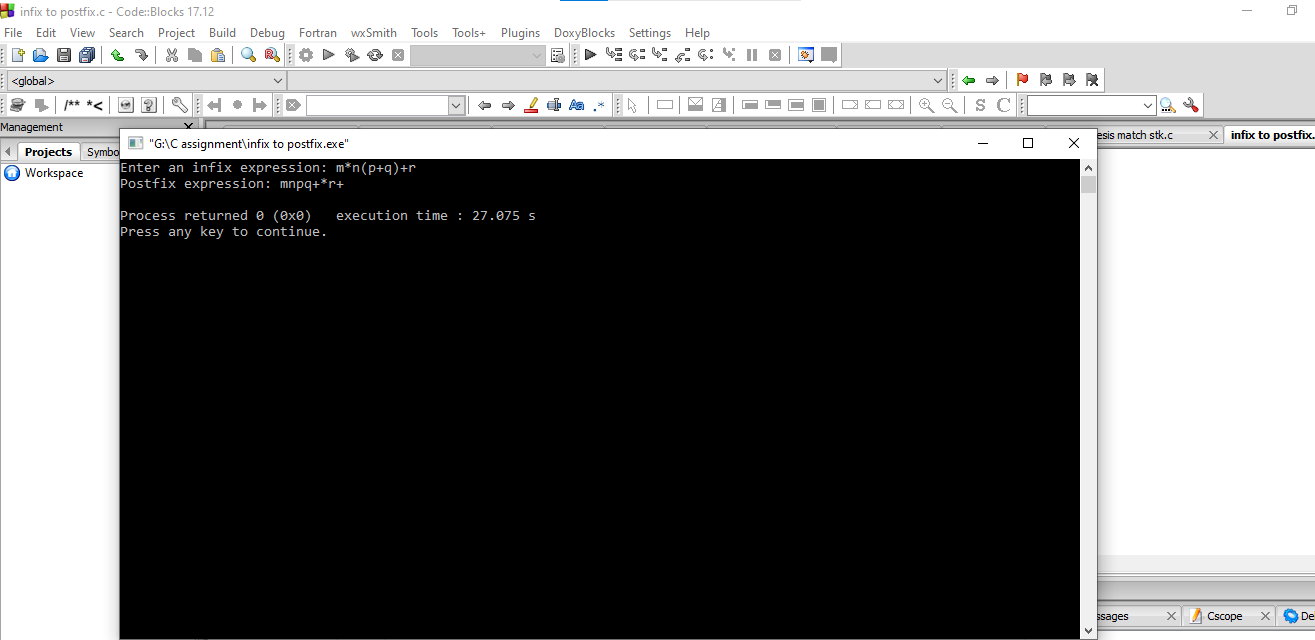
infixToPostfix(infix, postfix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

**Output:**



**13 Postfix Evaluation:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <ctype.h>

#define MAX\_SIZE 100

struct Stack {

int top;

int items[MAX\_SIZE];

};

void initialize(struct Stack \*s) {

s->top = -1;

}

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

if (s->top == MAX\_SIZE - 1) {

printf("Stack Overflow\n");

return;

}

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

}

int pop(struct Stack \*s) {

if (s->top == -1) {

printf("Stack Underflow\n");

exit(1);

}

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

}

bool isOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/');

}

int performOperation(char operator, int operand1, int operand2) {

switch (operator) {

case '+': return operand1 + operand2;

case '-': return operand1 - operand2;

case '\*': return operand1 \* operand2;

case '/': return operand1 / operand2;

default: return 0;

}

}

int evaluatePostfix(char postfix[]) {

struct Stack s;

initialize(&s);

for (int i = 0; postfix[i] != '\0'; i++) {

if (isdigit(postfix[i]))

push(&s, postfix[i] - '0');

else if (isOperator(postfix[i])) {

int operand2 = pop(&s);

int operand1 = pop(&s);

int result = performOperation(postfix[i], operand1, operand2);

push(&s, result);

}

}

return pop(&s);

}

int main() {

char postfix[MAX\_SIZE];

printf("Enter a postfix expression: ");

scanf("%s", postfix);

printf("Result: %d\n", evaluatePostfix(postfix));

return 0;

}

**Output:**



**14. Conversion of Decimal to binary**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct Stack {

int top;

int items[MAX\_SIZE];

};

void initialize(struct Stack \*s) {

s->top = -1;

}

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

if (s->top == MAX\_SIZE - 1) {

printf("Stack Overflow\n");

return;

}

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

}

int pop(struct Stack \*s) {

if (s->top == -1) {

printf("Stack Underflow\n");

exit(1);

}

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

}

bool isEmpty(struct Stack \*s) {

return (s->top == -1);

}

void decimalToBinary(int decimal) {

struct Stack s;

initialize(&s);

while (decimal > 0) {

int rem = decimal % 2;

push(&s, rem);

decimal /= 2;

}

printf("Binary: ");

while (!isEmpty(&s))

printf("%d", pop(&s));

printf("\n");

}

int main() {

int decimal;

printf("Enter a decimal number: ");

scanf("%d", &decimal);

decimalToBinary(decimal);

return 0;

}

**Output:**



**15. Queue Implementation(Array Based)**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct Queue {

int front, rear;

int items[MAX\_SIZE];

};

void initialize(struct Queue \*q) {

q->front = -1;

q->rear = -1;

}

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

if (q->rear == MAX\_SIZE - 1) {

printf("Queue Overflow\n");

return;

}

if (q->front == -1)

q->front = 0;

q->items[++(q->rear)] = value;

}

int dequeue(struct Queue \*q) {

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

printf("Queue Underflow\n");

exit(1);

}

int item = q->items[(q->front)++];

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

initialize(q);

return item;

}

bool isEmpty(struct Queue \*q) {

return (q->front == -1 || q->front > q->rear);

}

void printQueue(struct Queue \*q) {

printf("Queue: ");

for (int i = q->front; i <= q->rear; i++)

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

printf("\n");

}

int main() {

struct Queue q;

initialize(&q);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&q));

break;

case 3:

printQueue(&q);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

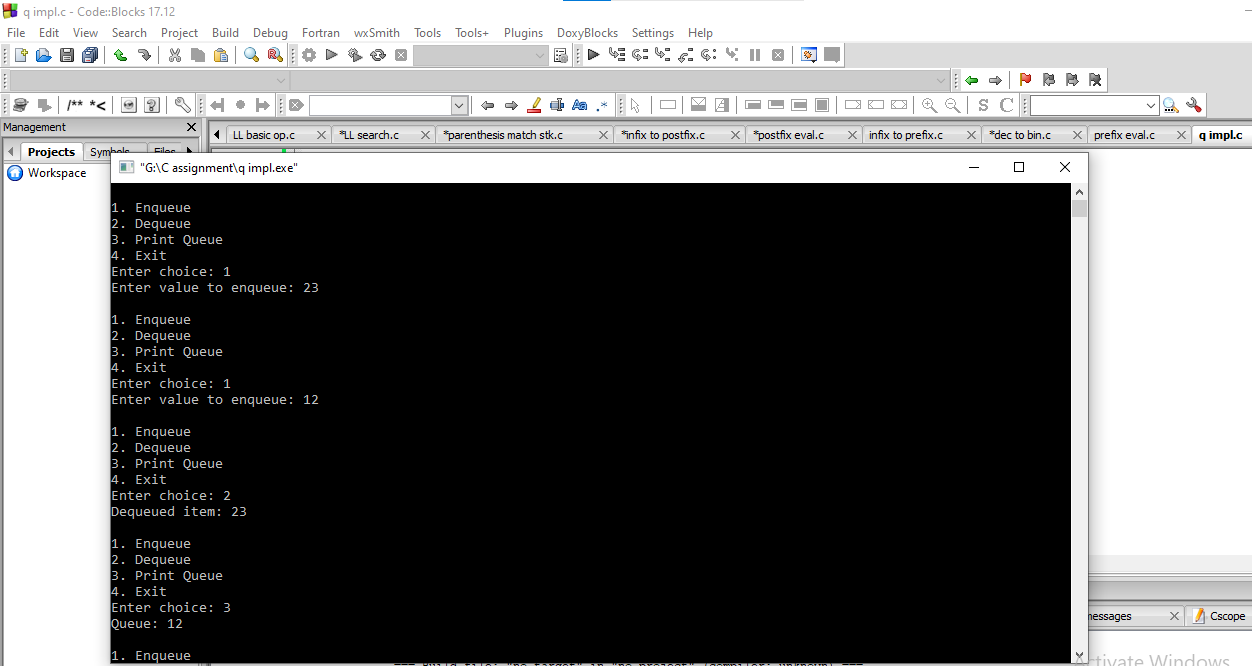
}

}

return 0;

}

**Output:**



**16. Circular Queue Implementation:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct CircularQueue {

int front, rear;

int items[MAX\_SIZE];

};

void initialize(struct CircularQueue \*cq) {

cq->front = -1;

cq->rear = -1;

}

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

if ((cq->rear + 1) % MAX\_SIZE == cq->front) {

printf("Queue Overflow\n");

return;

}

if (cq->front == -1)

cq->front = 0;

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

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

}

int dequeue(struct CircularQueue \*cq) {

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

printf("Queue Underflow\n");

exit(1);

}

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

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

initialize(cq);

else

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

return item;

}

bool isEmpty(struct CircularQueue \*cq) {

return (cq->front == -1);

}

void printQueue(struct CircularQueue \*cq) {

printf("Circular Queue: ");

int i = cq->front;

while (i != cq->rear) {

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

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

}

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

}

int main() {

struct CircularQueue cq;

initialize(&cq);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Circular Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&cq, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&cq));

break;

case 3:

printQueue(&cq);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

}

}

return 0;

}

**Output:**



**17. Queue implementation(Linked list based):**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct Node {

int data;

struct Node \*next;

};

struct Queue {

struct Node \*front, \*rear;

};

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

void initialize(struct Queue \*q) {

q->front = q->rear = NULL;

}

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

struct Node \*newNode = createNode(value);

if (q->rear == NULL) {

q->front = q->rear = newNode;

return;

}

q->rear->next = newNode;

q->rear = newNode;

}

int dequeue(struct Queue \*q) {

if (q->front == NULL) {

printf("Queue Underflow\n");

exit(1);

}

int item = q->front->data;

struct Node \*temp = q->front;

q->front = q->front->next;

if (q->front == NULL)

q->rear = NULL;

free(temp);

return item;

}

bool isEmpty(struct Queue \*q) {

return (q->front == NULL);

}

void printQueue(struct Queue \*q) {

struct Node \*temp = q->front;

printf("Queue: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

struct Queue q;

initialize(&q);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&q));

break;

case 3:

printQueue(&q);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

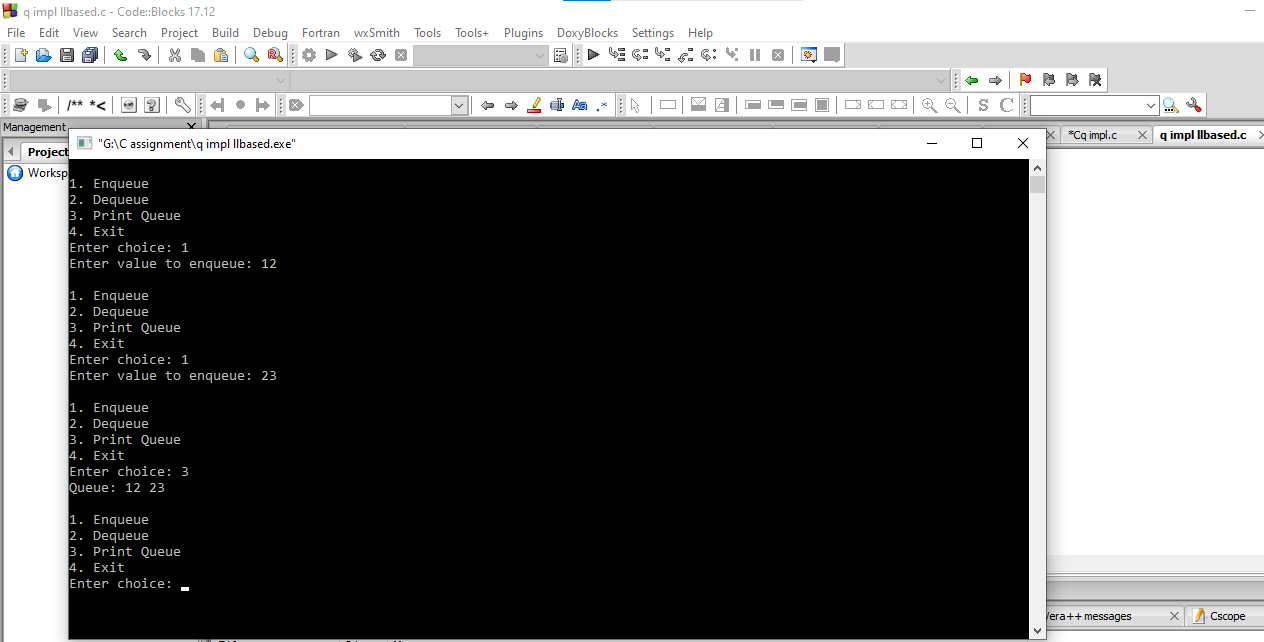
}

}

return 0;

}

**Output:**



**18. Circular Queue Implementation(Linked List based):**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct Node {

int data;

struct Node \*next;

};

struct CircularQueue {

struct Node \*front, \*rear;

};

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

void initialize(struct CircularQueue \*cq) {

cq->front = cq->rear = NULL;

}

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

struct Node \*newNode = createNode(value);

if (cq->rear == NULL) {

cq->front = cq->rear = newNode;

newNode->next = newNode;

return;

}

newNode->next = cq->front;

cq->rear->next = newNode;

cq->rear = newNode;

}

int dequeue(struct CircularQueue \*cq) {

if (cq->front == NULL) {

printf("Queue Underflow\n");

exit(1);

}

int item = cq->front->data;

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

free(cq->front);

cq->front = cq->rear = NULL;

} else {

struct Node \*temp = cq->front;

cq->front = cq->front->next;

cq->rear->next = cq->front;

free(temp);

}

return item;

}

bool isEmpty(struct CircularQueue \*cq) {

return (cq->front == NULL);

}

void printQueue(struct CircularQueue \*cq) {

struct Node \*temp = cq->front;

printf("Circular Queue: ");

if (!isEmpty(cq)) {

do {

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

temp = temp->next;

} while (temp != cq->front);

}

printf("\n");

}

int main() {

struct CircularQueue cq;

initialize(&cq);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Circular Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&cq, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&cq));

break;

case 3:

printQueue(&cq);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

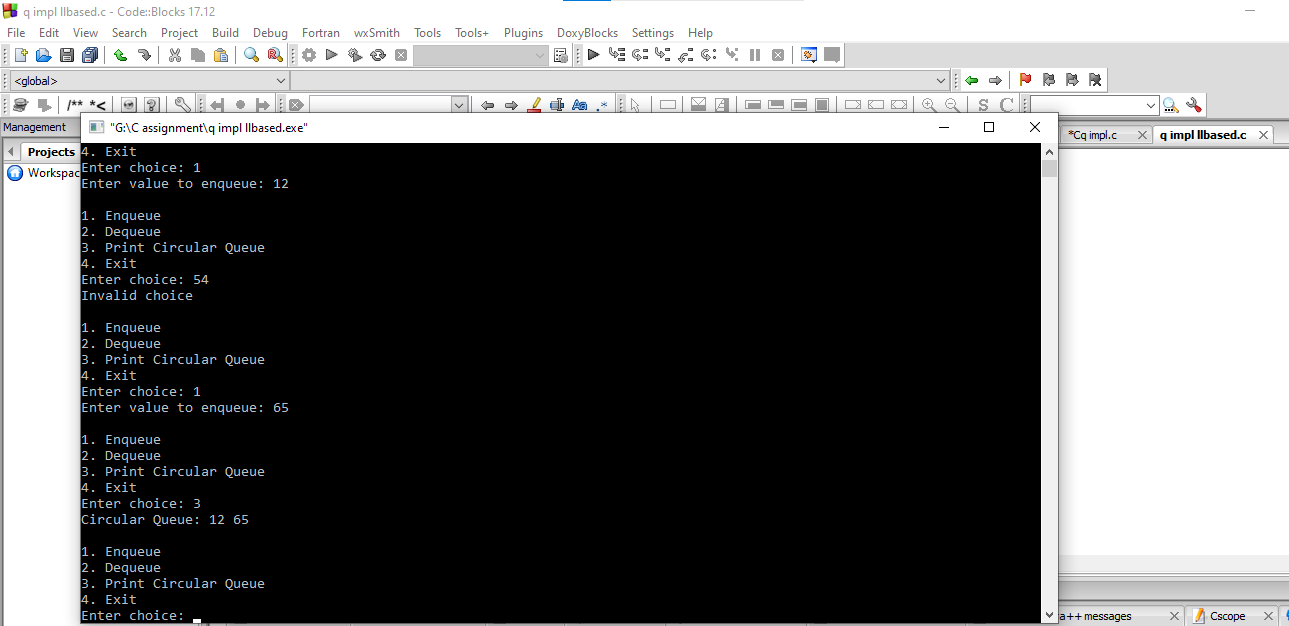
}

}

return 0;

}

**Output:**



**19. Priority Queue Implementation**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct PriorityQueue {

int items[MAX\_SIZE];

int front, rear;

};

void initialize(struct PriorityQueue \*pq) {

pq->front = pq->rear = -1;

}

bool isFull(struct PriorityQueue \*pq) {

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

}

bool isEmpty(struct PriorityQueue \*pq) {

return (pq->front == -1);

}

void enqueue(struct PriorityQueue \*pq, int value) {

if (isFull(pq)) {

printf("Priority Queue Overflow\n");

return;

}

if (isEmpty(pq)) {

pq->front = pq->rear = 0;

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

} else {

int i = pq->rear;

while (i >= pq->front && value > pq->items[i]) {

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

i--;

}

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

pq->rear++;

}

}

int dequeue(struct PriorityQueue \*pq) {

if (isEmpty(pq)) {

printf("Priority Queue Underflow\n");

exit(1);

}

int item = pq->items[pq->front];

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

initialize(pq);

else

pq->front++;

return item;

}

void printPriorityQueue(struct PriorityQueue \*pq) {

printf("Priority Queue: ");

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

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

printf("\n");

}

int main() {

struct PriorityQueue pq;

initialize(&pq);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Priority Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&pq, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&pq));

break;

case 3:

printPriorityQueue(&pq);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

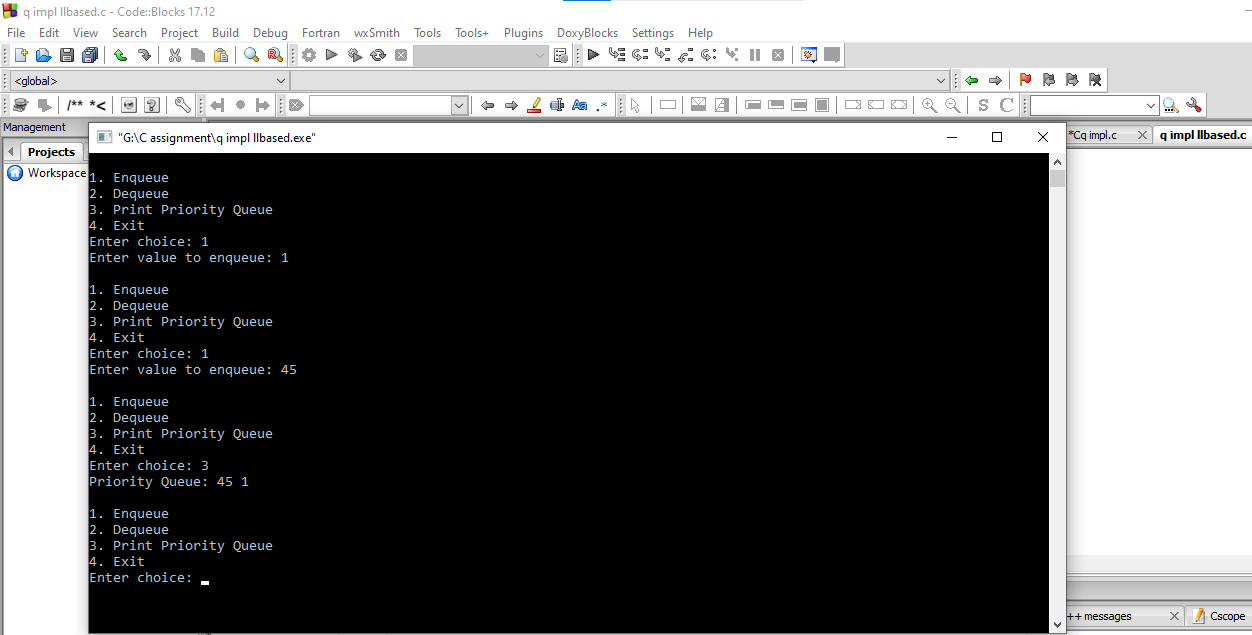
}

}

return 0;

}

**Output:**



**20.Queue Implementation using two stacks:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct Stack {

int top;

int items[100];

};

void initialize(struct Stack \*s) {

s->top = -1;

}

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

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

}

int pop(struct Stack \*s) {

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

}

bool isEmpty(struct Stack \*s) {

return (s->top == -1);

}

struct Queue {

struct Stack s1, s2;

};

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

while (!isEmpty(&q->s1))

push(&q->s2, pop(&q->s1));

push(&q->s1, value);

while (!isEmpty(&q->s2))

push(&q->s1, pop(&q->s2));

}

int dequeue(struct Queue \*q) {

if (isEmpty(&q->s1)) {

printf("Queue Underflow\n");

exit(1);

}

return pop(&q->s1);

}

void printQueue(struct Queue \*q) {

printf("Queue: ");

for (int i = 0; i <= q->s1.top; i++)

printf("%d ", q->s1.items[i]);

printf("\n");

}

int main() {

struct Queue q;

initialize(&q.s1);

initialize(&q.s2);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&q));

break;

case 3:

printQueue(&q);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

}

}

return 0;

}

**Output:**



**21. Implementation of Double Ended Queue:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct Deque {

int items[MAX\_SIZE];

int front, rear;

};

void initialize(struct Deque \*d) {

d->front = -1;

d->rear = 0;

}

bool isFull(struct Deque \*d) {

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

}

bool isEmpty(struct Deque \*d) {

return (d->front == -1);

}

void insertFront(struct Deque \*d, int value) {

if (isFull(d)) {

printf("Deque Overflow\n");

return;

}

if (d->front == -1)

d->front = d->rear = 0;

else if (d->front == 0)

d->front = MAX\_SIZE - 1;

else

(d->front)--;

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

}

void insertRear(struct Deque \*d, int value) {

if (isFull(d)) {

printf("Deque Overflow\n");

return;

}

if (d->front == -1)

d->front = d->rear = 0;

else if (d->rear == MAX\_SIZE - 1)

d->rear = 0;

else

(d->rear)++;

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

}

int deleteFront(struct Deque \*d) {

if (isEmpty(d)) {

printf("Deque Underflow\n");

exit(1);

}

int item = d->items[d->front];

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

d->front = d->rear = -1;

else if (d->front == MAX\_SIZE - 1)

d->front = 0;

else

(d->front)++;

return item;

}

int deleteRear(struct Deque \*d) {

if (isEmpty(d)) {

printf("Deque Underflow\n");

exit(1);

}

int item = d->items[d->rear];

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

d->front = d->rear = -1;

else if (d->rear == 0)

d->rear = MAX\_SIZE - 1;

else

(d->rear)--;

return item;

}

void printDeque(struct Deque \*d) {

printf("Deque: ");

if (!isEmpty(d)) {

int i = d->front;

while (true) {

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

if (i == d->rear)

break;

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

}

}

printf("\n");

}

int main() {

struct Deque d;

initialize(&d);

int choice, value;

while (true) {

printf("\n1. Insert Front\n2. Insert Rear\n3. Delete Front\n4. Delete Rear\n5. Print Deque\n6. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert at front: ");

scanf("%d", &value);

insertFront(&d, value);

break;

case 2:

printf("Enter value to insert at rear: ");

scanf("%d", &value);

insertRear(&d, value);

break;

case 3:

printf("Deleted item from front: %d\n", deleteFront(&d));

break;

case 4:

printf("Deleted item from rear: %d\n", deleteRear(&d));

break;

case 5:

printDeque(&d);

break;

case 6:

exit(0);

default:

printf("Invalid choice\n");

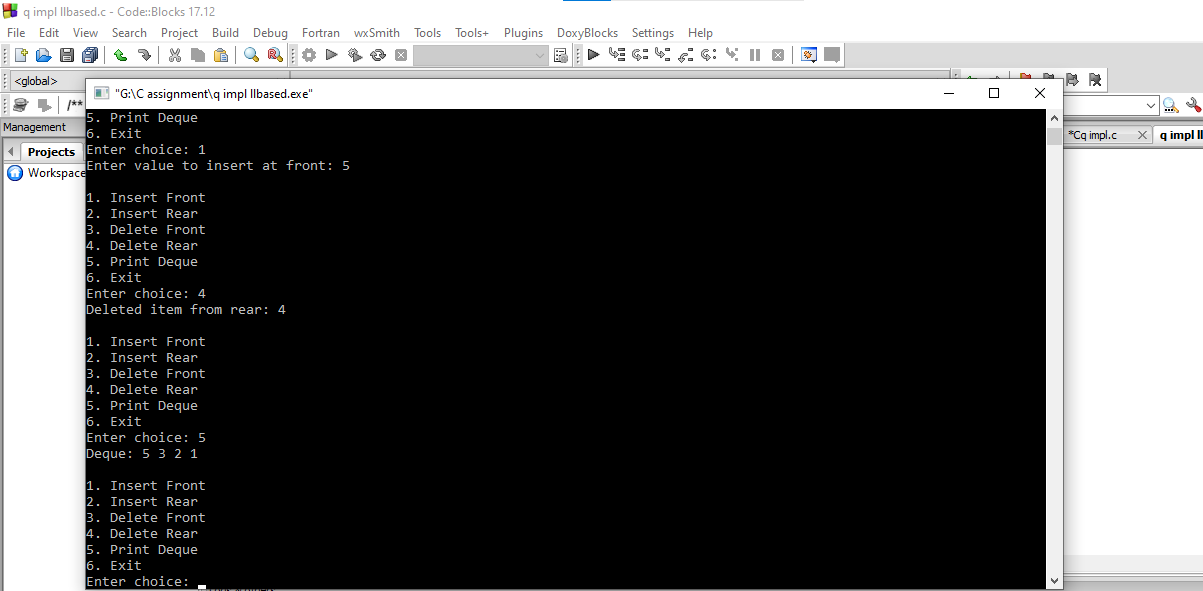
}

}

return 0;

}

**Output:**



**22. Queue implementation using dynamic memory allocation:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct Node {

int data;

struct Node \*next;

};

struct Queue {

struct Node \*front, \*rear;

};

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

newNode->data = value;

newNode->next = NULL;

return newNode;

}

void initialize(struct Queue \*q) {

q->front = q->rear = NULL;

}

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

struct Node \*newNode = createNode(value);

if (q->rear == NULL) {

q->front = q->rear = newNode;

return;

}

q->rear->next = newNode;

q->rear = newNode;

}

int dequeue(struct Queue \*q) {

if (q->front == NULL) {

printf("Queue Underflow\n");

exit(1);

}

int item = q->front->data;

struct Node \*temp = q->front;

q->front = q->front->next;

if (q->front == NULL)

q->rear = NULL;

free(temp);

return item;

}

bool isEmpty(struct Queue \*q) {

return (q->front == NULL);

}

void printQueue(struct Queue \*q) {

struct Node \*temp = q->front;

printf("Queue: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

struct Queue q;

initialize(&q);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&q));

break;

case 3:

printQueue(&q);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

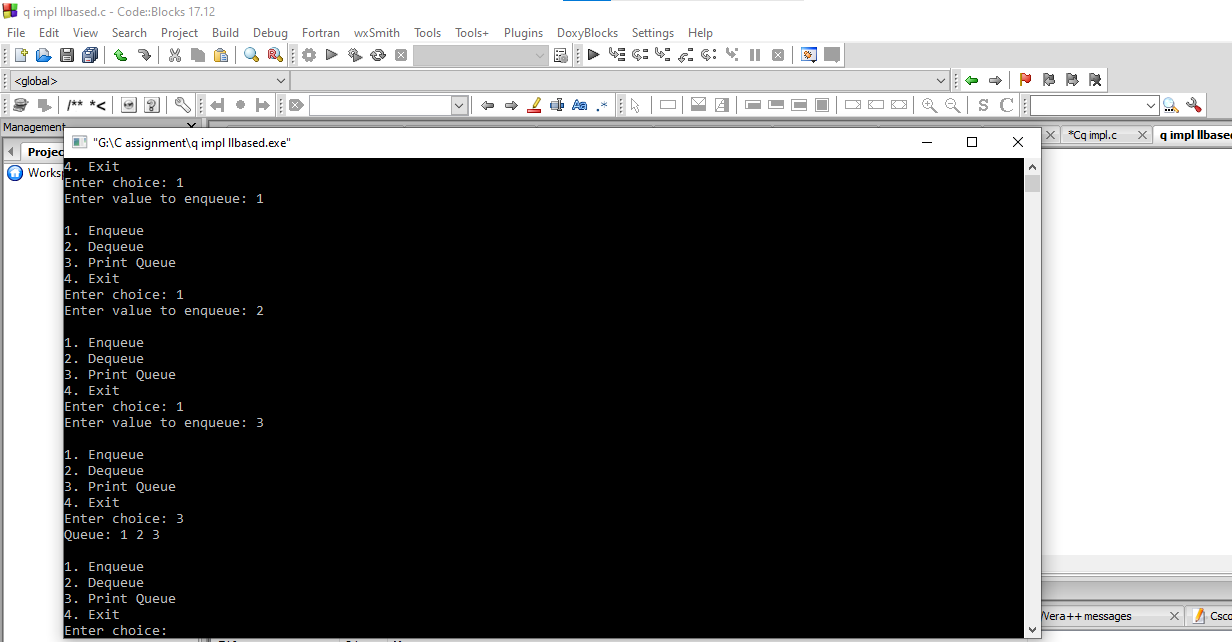
}

}

return 0;

}

**Output:**



**23. Queue Implementation using two pointers:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct Queue {

int items[MAX\_SIZE];

int front, rear;

};

void initialize(struct Queue \*q) {

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

}

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

if (q->rear == MAX\_SIZE - 1) {

printf("Queue Overflow\n");

return;

}

if (q->front == -1)

q->front = 0;

q->items[++(q->rear)] = value;

}

int dequeue(struct Queue \*q) {

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

printf("Queue Underflow\n");

exit(1);

}

int item = q->items[(q->front)++];

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

initialize(q);

return item;

}

bool isEmpty(struct Queue \*q) {

return (q->front == -1 || q->front > q->rear);

}

void printQueue(struct Queue \*q) {

printf("Queue: ");

for (int i = q->front; i <= q->rear; i++)

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

printf("\n");

}

int main() {

struct Queue q;

initialize(&q);

int choice, value;

while (true) {

printf("\n1. Enqueue\n2. Dequeue\n3. Print Queue\n4. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(&q, value);

break;

case 2:

printf("Dequeued item: %d\n", dequeue(&q));

break;

case 3:

printQueue(&q);

break;

case 4:

exit(0);

default:

printf("Invalid choice\n");

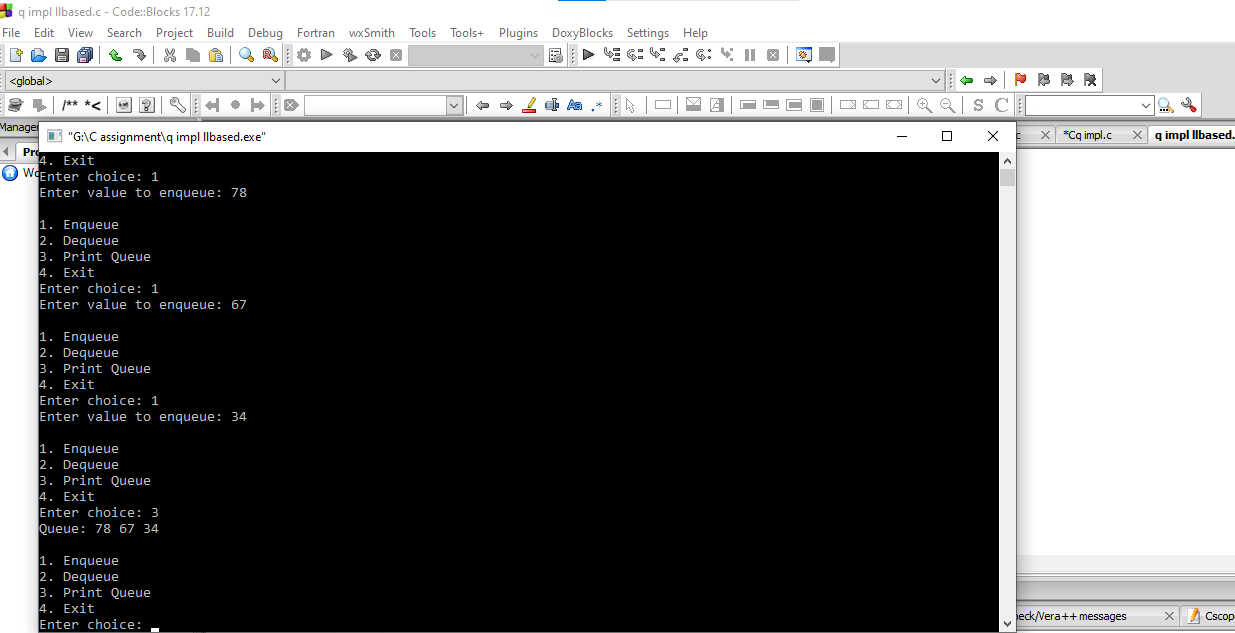
}

}

return 0;

}

**Output:**



**24. Stack Basic operations:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct Stack {

int top;

int items[MAX\_SIZE];

};

void initialize(struct Stack \*s) {

s->top = -1;

}

bool isEmpty(struct Stack \*s) {

return (s->top == -1);

}

bool isFull(struct Stack \*s) {

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

}

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

if (isFull(s)) {

printf("Stack Overflow\n");

return;

}

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 Underflow\n");

exit(1);

}

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

}

void printStack(struct Stack \*s) {

printf("Stack: ");

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

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

}

printf("\n");

}

int main() {

struct Stack s;

initialize(&s);

int choice, value;

while (true) {

printf("\n1. Push\n2. Pop\n3. Peek\n4. Print Stack\n5. Exit\nEnter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to push: ");

scanf("%d", &value);

push(&s, value);

break;

case 2:

printf("Popped item: %d\n", pop(&s));

break;

case 3:

printf("Top item: %d\n", peek(&s));

break;

case 4:

printStack(&s);

break;

case 5:

exit(0);

default:

printf("Invalid choice\n");

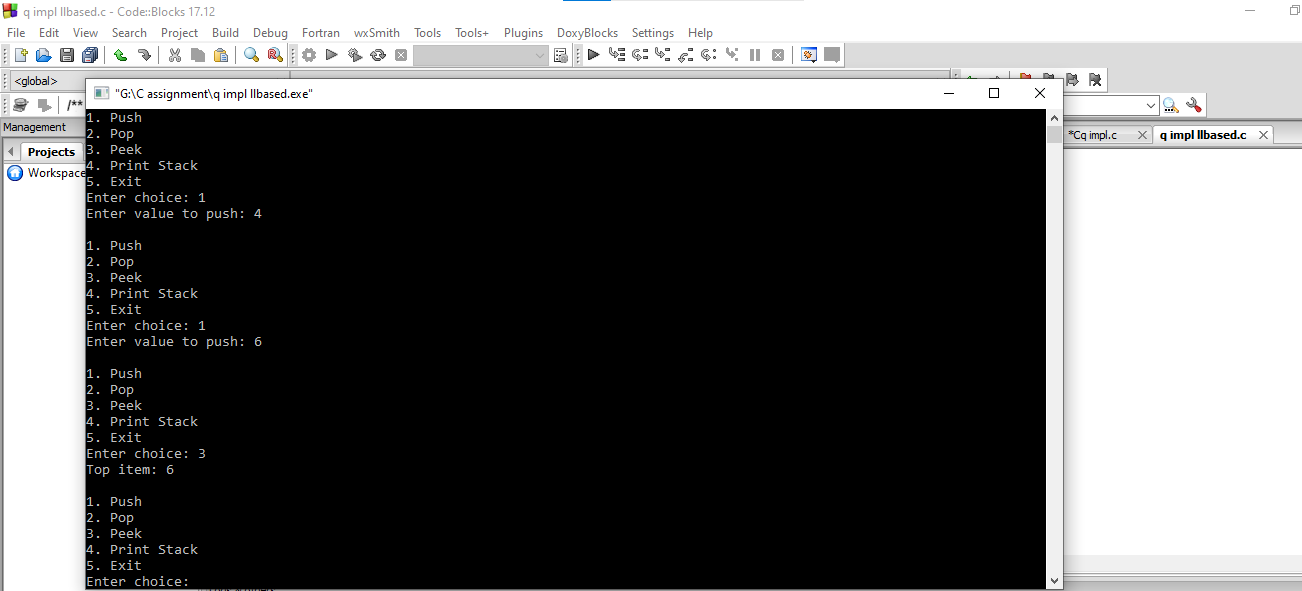
}

}

return 0;

}

**Output:**



**25. Copy List with random pointer:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int val;

struct Node\* next;

struct Node\* random;

};

struct Node\* createNode(int val) {

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

newNode->val = val;

newNode->next = NULL;

newNode->random = NULL;

return newNode;

}

struct Node\* copyRandomList(struct Node\* head) {

if (head == NULL) return NULL;

struct Node\* original = head;

struct Node\* copy = NULL;

struct Node\* copyHead = NULL;

while (original != NULL) {

copy = createNode(original->val);

copy->next = original->next;

original->next = copy;

original = copy->next;

}

original = head;

while (original != NULL) {

if (original->random != NULL)

original->next->random = original->random->next;

original = original->next->next;

}

original = head;

copyHead = head->next;

copy = copyHead;

while (original != NULL) {

original->next = original->next->next;

if (copy->next != NULL)

copy->next = copy->next->next;

original = original->next;

copy = copy->next;

}

return copyHead;

}

void printList(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

if (current->random != NULL)

printf(", %d", current->random->val);

printf(") ");

current = current->next;

}

}

int main() {

struct Node\* head = createNode(1);

head->next = createNode(2);

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

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

head->next->random = head;

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

printf("Original list with random pointers: ");

printList(head);

printf("\n");

struct Node\* copyHead = copyRandomList(head);

printf("Copied list with random pointers: ");

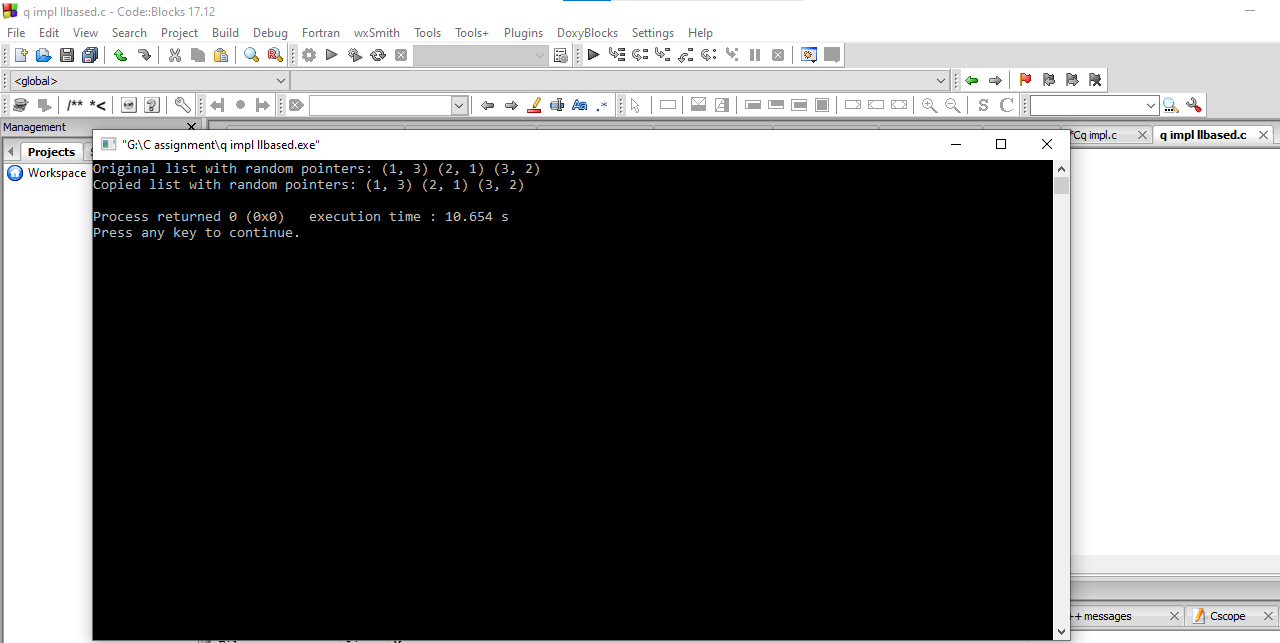
printList(copyHead);

printf("\n");

return 0;

}

**Output:**



**26. Sports Eligilbe Queue:**

**Code:**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define SIZE 10

struct queue

{

int data[SIZE];

int front, rear;

};

int isfull(struct queue \*qptr)

{

if(qptr->rear == SIZE-1)

return 1;

else

return 0;

}

int isempty(struct queue \*qptr)

{

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

return 1;

else

return 0;

}

void enqueue(struct queue \*qptr, int num)

{

if(isfull(qptr))

printf("Queue Overflow\n");

else

{

qptr->rear++;

qptr->data[qptr->rear] = num;

}

}

int dequeue(struct queue \*qptr)

{

int num=0;

if(isempty(qptr))

return num;

else

{

qptr->front++;

num = qptr->data[qptr->front];

return num;

}

}

int candidates(struct queue \*qptr, int n)

{

int num,count=0;

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

{

num=dequeue(qptr);

if(num<=185)

count++; }

return

count;

}

int main()

{

int n, num,count;

struct queue q, \*qptr;

qptr=&q;

qptr->front = -1;

qptr->rear = -1;

scanf("%d",&n);

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

{

scanf("%d",&num);

enqueue(qptr, num);

}

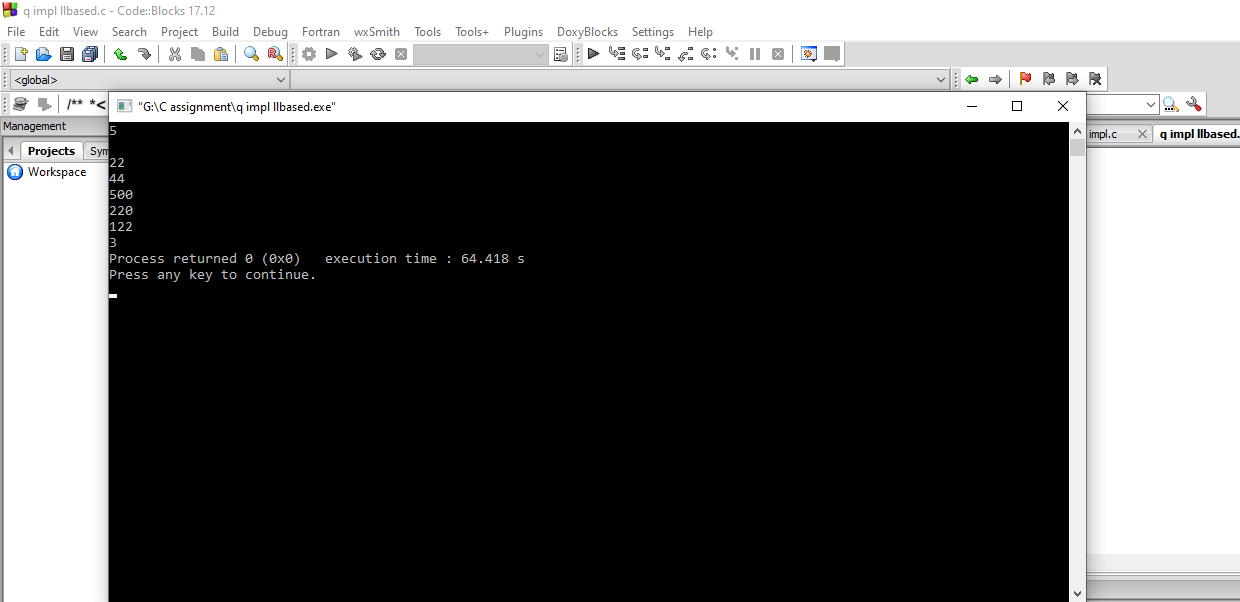
count = candidates(qptr,n);

printf("%d",count);

return 0;

}

**Output:**



**27. Linked List Delete Last occurrence of key:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a Node

struct Node {

int data;

struct Node\* next;

};

// Function to delete the last occurrence of a key from a singly linked list

void deleteLastOccurrence(struct Node\*\* headRef, int key) {

if (\*headRef == NULL)

return;

struct Node\* prev = NULL;

struct Node\* lastOccurrence = NULL;

struct Node\* current = \*headRef;

// Traverse the list to find the last occurrence of the key and its previous node

while (current != NULL) {

if (current->data == key)

lastOccurrence = prev;

prev = current;

current = current->next;

}

// If last occurrence is NULL and the last node does not contain the key, return

if (lastOccurrence == NULL && prev->data != key)

return;

// If last occurrence is NULL, remove the first node containing the key

if (lastOccurrence == NULL) {

struct Node\* temp = \*headRef;

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

free(temp);

} else { // Otherwise, adjust the pointers to skip the last occurrence node

struct Node\* temp = lastOccurrence->next;

lastOccurrence->next = lastOccurrence->next->next;

free(temp);

}

}

// Function to insert a new node at the end of the linked list

void insertAtEnd(struct Node\*\* headRef, int newData) {

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

if (newNode == NULL) {

printf("Memory allocation failed.\n");

exit(1);

}

newNode->data = newData;

newNode->next = NULL;

if (\*headRef == NULL) {

\*headRef = newNode;

return;

}

struct Node\* temp = \*headRef;

while (temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

// Function to print the linked list

void printList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

// Main function

int main() {

struct Node\* head = NULL;

// Insert elements into the linked list

insertAtEnd(&head, 1);

insertAtEnd(&head, 2);

insertAtEnd(&head, 3);

insertAtEnd(&head, 2);

insertAtEnd(&head, 4);

insertAtEnd(&head, 5);

insertAtEnd(&head, 2);

insertAtEnd(&head, 6);

printf("Original linked list: ");

printList(head);

int keyToDelete = 2;

deleteLastOccurrence(&head, keyToDelete);

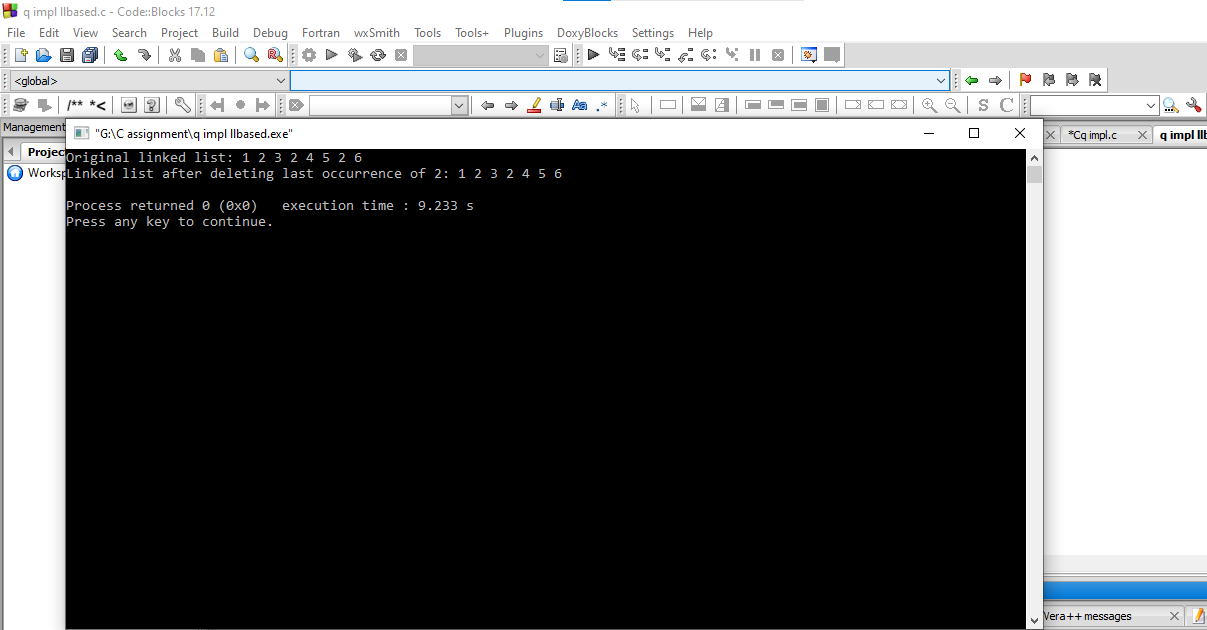
printf("Linked list after deleting last occurrence of %d: ", keyToDelete);

printList(head);

return 0;

}

**Output:**



**28. Generate binary numbers using queue:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Structure for queue node

struct Node {

char\* data;

struct Node\* next;

};

// Structure for queue

struct Queue {

struct Node \*front, \*rear;

};

// Function to create a new queue node

struct Node\* createNode(char\* data) {

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

if (newNode == NULL) {

printf("Memory allocation failed.\n");

exit(1);

}

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to create a new queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

if (queue == NULL) {

printf("Memory allocation failed.\n");

exit(1);

}

queue->front = queue->rear = NULL;

return queue;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* queue) {

return (queue->front == NULL);

}

// Function to enqueue an element

void enqueue(struct Queue\* queue, char\* data) {

struct Node\* newNode = createNode(data);

if (isEmpty(queue)) {

queue->front = queue->rear = newNode;

} else {

queue->rear->next = newNode;

queue->rear = newNode;

}

}

// Function to dequeue an element

char\* dequeue(struct Queue\* queue) {

if (isEmpty(queue)) {

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

exit(1);

}

struct Node\* temp = queue->front;

char\* data = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL)

queue->rear = NULL;

free(temp);

return data;

}

// Function to generate and print the first n binary numbers

void generateBinaryNumbers(int n) {

struct Queue\* queue = createQueue();

enqueue(queue, "1");

while (n-- > 0) {

char\* front = dequeue(queue);

printf("%s ", front);

char\* nextNum1 = (char\*)malloc(sizeof(char) \* (strlen(front) + 2));

char\* nextNum2 = (char\*)malloc(sizeof(char) \* (strlen(front) + 2));

strcpy(nextNum1, front);

strcpy(nextNum2, front);

strcat(nextNum1, "0");

strcat(nextNum2, "1");

enqueue(queue, nextNum1);

enqueue(queue, nextNum2);

free(front);

}

}

int main() {

int n;

printf("Enter the number of binary numbers to generate: ");

scanf("%d", &n);

printf("First %d binary numbers are: ", n);

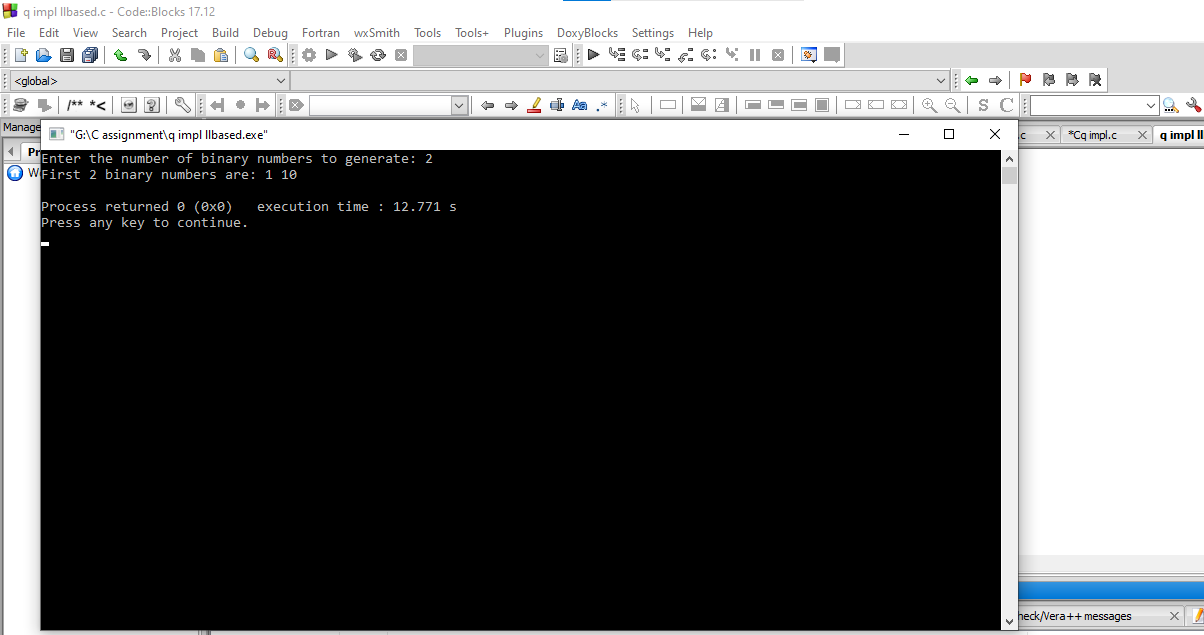
generateBinaryNumbers(n);

printf("\n");

return 0;

}

**Output:**



**29. Priority Queue:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct PriorityQueue {

int arr[MAX\_SIZE];

int size;

};

// Function to create a new priority queue

struct PriorityQueue\* createPriorityQueue() {

struct PriorityQueue\* pq = (struct PriorityQueue\*)malloc(sizeof(struct PriorityQueue));

pq->size = 0;

return pq;

}

// Function to get the parent index of a node

int getParentIndex(int index) {

return (index - 1) / 2;

}

// Function to get the left child index of a node

int getLeftChildIndex(int index) {

return (2 \* index) + 1;

}

// Function to get the right child index of a node

int getRightChildIndex(int index) {

return (2 \* index) + 2;

}

// Function to swap two elements in the priority queue

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to maintain heap property by moving the element up

void heapifyUp(struct PriorityQueue\* pq, int index) {

int parentIndex = getParentIndex(index);

while (index > 0 && pq->arr[index] > pq->arr[parentIndex]) {

swap(&pq->arr[index], &pq->arr[parentIndex]);

index = parentIndex;

parentIndex = getParentIndex(index);

}

}

// Function to maintain heap property by moving the element down

void heapifyDown(struct PriorityQueue\* pq, int index) {

int maxIndex = index;

int leftChildIndex = getLeftChildIndex(index);

int rightChildIndex = getRightChildIndex(index);

if (leftChildIndex < pq->size && pq->arr[leftChildIndex] > pq->arr[maxIndex]) {

maxIndex = leftChildIndex;

}

if (rightChildIndex < pq->size && pq->arr[rightChildIndex] > pq->arr[maxIndex]) {

maxIndex = rightChildIndex;

}

if (index != maxIndex) {

swap(&pq->arr[index], &pq->arr[maxIndex]);

heapifyDown(pq, maxIndex);

}

}

// Function to insert an element into the priority queue

void enqueue(struct PriorityQueue\* pq, int value) {

if (pq->size == MAX\_SIZE) {

printf("Priority Queue Overflow\n");

return;

}

pq->arr[pq->size++] = value;

heapifyUp(pq, pq->size - 1);

}

// Function to remove and return the maximum element from the priority queue

int dequeue(struct PriorityQueue\* pq) {

if (pq->size == 0) {

printf("Priority Queue Underflow\n");

return -1;

}

int maxValue = pq->arr[0];

pq->arr[0] = pq->arr[pq->size - 1];

pq->size--;

heapifyDown(pq, 0);

return maxValue;

}

// Function to peek the maximum element in the priority queue

int peek(struct PriorityQueue\* pq) {

if (pq->size == 0) {

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

return -1;

}

return pq->arr[0];

}

int main() {

struct PriorityQueue\* pq = createPriorityQueue();

enqueue(pq, 10);

enqueue(pq, 20);

enqueue(pq, 15);

printf("Maximum element: %d\n", peek(pq)); // Output: 20

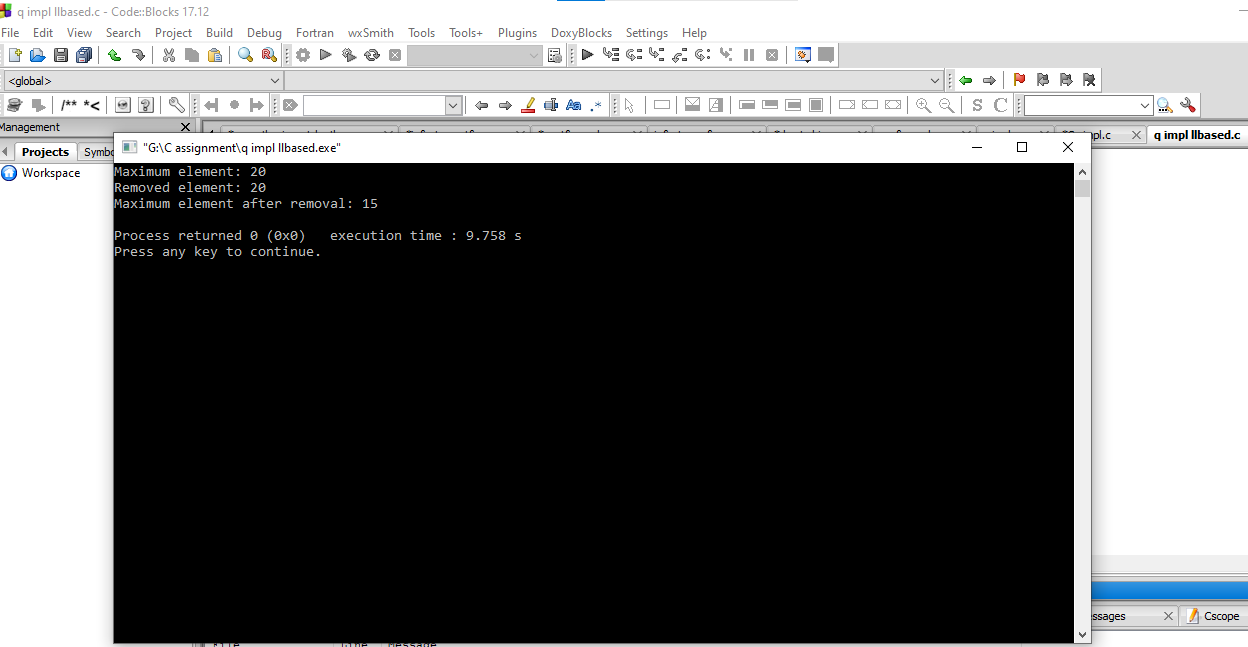
printf("Removed element: %d\n", dequeue(pq)); // Output: 20

printf("Maximum element after removal: %d\n", peek(pq)); // Output: 15

return 0;

}

**Output:**



**30. // A new Toll bay is setup at Airport Road. There are different types of vehicles passing through the toll. Vehicles are categorized as LMV and HMV. The charges for LMV passing by toll is Rs. 50 and HMV passing by toll is Rs. 100 each time vehicles pass.Apply Problem Solving Framework to perform the following:**

**//I. Read and display the vehicles details.**

**//II. Display only HMV vehicle details.**

**//III. Count the LMV vehicles.**

**Code:**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<string.h>**

**#define SIZE 10**

**struct q**

**{**

**char vnum[SIZE][12]; //VEHICLE NUMBER**

**char vtype[SIZE][SIZE]; // VEHICLE TYPE. LMV OR HMV.**

**int cost[SIZE]; //COST BASED ON VEHICLE TYPE.**

**int front;**

**int rear;**

**};**

**typedef struct q QUEUE;**

**QUEUE create\_queue()**

**{**

**QUEUE q1;**

**q1.front = -1;**

**q1.rear = 0;**

**return q1;**

**}**

**void enqueue(QUEUE \*q1)**

**{**

**char vcnum[10], vctype[5];**

**if(q1->rear == SIZE)**

**printf("Queue Overflow\n");**

**else**

**{**

**printf("Enter vehicle number, vehicle type (as LMV or HMV)\n");**

**scanf("%s%s", vcnum, vctype);**

**strcpy(q1->vnum[q1->rear],vcnum);**

**strcpy(q1->vtype[q1->rear], vctype);**

**if(strcmp(q1->vtype[q1->rear],"HMV")==0) // ASSIGN COST BASED ON VEHICLE TYPE.**

**q1->cost[q1->rear]= 100;**

**else**

**q1->cost[q1->rear]= 50;**

**q1->rear ++;**

**}**

**}**

**void dequeue(QUEUE \*q1)**

**{**

**if(q1->rear == q1->front +1)**

**printf("Queue Underflow\n");**

**else**

**{**

**q1->front++;**

**printf("Vehicle moving out of toll: %s. %s. \n", q1->vnum[q1->front], q1->vtype);**

**}**

**}**

**void displayqueue(QUEUE \*q1) // DISPLAY ALL VEHICLES.**

**{**

**int i;**

**if(q1->rear == q1->front+1)**

**{**

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

**}**

**else**

**{**

**printf("Vehicles at Toll bay are: \n");**

**for(i=q1->front+1; i<q1->rear; i++)**

**{**

**printf("%s %s %d\n", q1->vnum[i], q1->vtype[i], q1->cost[i]);**

**}**

**printf("\n");**

**}**

**}**

**void displayqueuevehicles(QUEUE \*q1) // DISPLAYS HMV AND COUNTS LMV.**

**{**

**int i, lmvcount=0;**

**if(q1->rear == q1->front+1)**

**{**

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

**}**

**else**

**{**

**printf("Vehicles at Toll bay are: \n");**

**for(i=q1->front+1; i<q1->rear; i++)**

**{**

**if(strcmp(q1->vtype[i], "HMV")==0)**

**printf("%s %s %d\n", q1->vnum[i], q1->vtype[i], q1->cost[i]);**

**else**

**lmvcount++;**

**}**

**printf("\n");**

**}**

**printf("Number of LMV vehicles waiting to pass toll are: %d\n", lmvcount);**

**}**

**int main()**

**{**

**int i, n;**

**QUEUE q11, \*q1;**

**q11 = create\_queue();**

**q1=&q11;**

**printf("Enter number of vehicles waiting at Toll bay?\n");**

**scanf("%d",&n);**

**if(n>SIZE)**

**{**

**printf("Number of vehicles to enqueue are more than the size of toll bay.\n");**

**}**

**else**

**{**

**for(i=0;i<n;i++)**

**{**

**enqueue(q1);**

**}**

**displayqueue(q1);**

**printf("HMV vehicles: \n");**

**displayqueuevehicles(q1);**

**}**

**return 0;**

**}**

**Output:**

