|  |  |
| --- | --- |
| **Name of the student** | **PRUTHVIRAJ** |
| **Roll No** | **347** |
| **USN** | **01FE22BEC145** |
| **Div** | **‘C’** |

**Part- A**

|  |
| --- |
| **Variables and Data Types** |
| 1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate. |
| A In C programming, variables and data types are fundamental concepts, but they serve different purposes.   1. Data Types:   Data types in C specify how the data stored in memory should be interpreted. They define the set of values that a variable can hold and the operations that can be performed on those values.  C provides several basic data types such as int, float, char, double, etc. Each data type has a specific size and range of values.  Examples:  int: Used to store integer values.  float: Used to store single-precision floating-point numbers.  char: Used to store single characters.  double: Used to store double-precision floating-point numbers.  int age = 25;  float temperature = 98.6;  char grade = 'A';  double pi = 3.14159;   1. Variables:   Variables are named memory locations used to store data during program execution. They provide a way to refer to data by a symbolic name instead of using explicit memory addresses.  Variables must be declared with a specific data type before they can be used. This tells the compiler how much memory to allocate for the variable and how to interpret the data stored in it.  Examples:  int score; // Declaration of an integer variable named "score"  float height; // Declaration of a float variable named "height"  char initial; // Declaration of a character variable named "initial"  double distance; // Declaration of a double variable named "distance" |
| 2.What is the difference between a variable and a data type in C programming? Provide examples to illustrate. |
| Ans:  In C programming, a data type is an essential concept that defines the type of data that a variable can hold, as well as the operations that can be performed on that data. Data types in C specify the size and format of the stored data, enabling the compiler to allocate memory and interpret the data correctly during program execution. Here's a discussion of the different types of data types available in C:  1. \*\*Basic Data Types\*\*:  - \*\*int\*\*: Used to store integer values. It typically allocates 4 bytes of memory on most systems.  - \*\*char\*\*: Used to store single characters. It typically allocates 1 byte of memory.  - \*\*float\*\*: Used to store single-precision floating-point numbers. It typically allocates 4 bytes of memory.  - \*\*double\*\*: Used to store double-precision floating-point numbers. It typically allocates 8 bytes of memory.  2. \*\*Derived Data Types\*\*:  - \*\*Array\*\*: A collection of elements of the same data type arranged in contiguous memory locations.  - \*\*Pointer\*\*: A variable that stores the memory address of another variable. Pointers allow dynamic memory allocation and manipulation of memory addresses.  - \*\*Structure\*\*: A composite data type that groups together variables of different data types under one name. Structures enable the creation of complex data structures.  - \*\*Union\*\*: Similar to structures, but all members share the same memory location. Unions are used to save memory when only one member is accessed at a time.  - \*\*Enumeration\*\*: A user-defined data type used to assign names to integral constants, making the code more readable and maintainable.  3. \*\*Modifiers\*\*:  - \*\*signed\*\*: Indicates that a variable can store both positive and negative values. It's the default for int and char types.  - \*\*unsigned\*\*: Indicates that a variable can store only non-negative values, effectively doubling the range of positive values. It's commonly used with int and char types.  - \*\*short\*\*: Specifies a smaller range of values for an integer type. It typically allocates 2 bytes of memory.  - \*\*long\*\*: Specifies a larger range of values for an integer type. It typically allocates 4 bytes of memory.  - \*\*long long\*\*: Specifies an even larger range of values for an integer type. It typically allocates 8 bytes of memory.  4. \*\*Void Data Type\*\*:  - \*\*void\*\*: Represents the absence of a type. It's commonly used in function declarations to indicate that the function does not return a value or to specify generic pointers.  These data types provide flexibility and efficiency in storing and manipulating data within C programs, allowing developers to create a wide range of applications from simple to complex. |
| 1. How are variables declared and initialized in C programming? Provide examples of variable declarations with different data types. |
| **Ans:**  In C programming, variables are declared and initialized using specific syntax. Variable declaration involves specifying the data type of the variable, followed by the variable name. Initialization, which is optional, involves assigning an initial value to the variable at the time of declaration.  Here's the general syntax for declaring and initializing variables in C:  ```c  data\_type variable\_name = initial\_value;  ```  And here are examples of variable declarations with different data types:  1. \*\*Integer Variable Declaration and Initialization\*\*:  int age = 25;    2. \*\*Floating-point Variable Declaration and Initialization\*\*:    float temperature = 98.6;  3. \*\*Character Variable Declaration and Initialization\*\*:    char grade = 'A';    4. \*\*Double Variable Declaration and Initialization\*\*:  double pi = 3.14159;  5. \*\*Unsigned Integer Variable Declaration and Initialization\*\*:  unsigned int count = 100;  6. \*\*Short Integer Variable Declaration and Initialization\*\*:  short int distance = 500;    7. \*\*Long Integer Variable Declaration and Initialization\*\*:  long int population = 1000000;  8. \*\*Long Long Integer Variable Declaration and Initialization\*\*:  long long int big\_number = 1234567890123456;    9. \*\*Array Declaration and Initialization\*\*:  int numbers[5] = {1, 2, 3, 4, 5};    10. \*\*Pointer Declaration\*\*:  int \*ptr = NULL; // Initializing pointer to NULL    11. \*\*Structure Declaration and Initialization\*\*:    struct Point {  int x;  int y;  };    struct Point p1 = {10, 20};    These examples demonstrate how variables can be declared and optionally initialized with different data types in C programming. It's important to note that variables must be declared before they can be used in C programs. Additionally, the initial value assigned during initialization must be compatible with the declared data type of the variable. |
| 1. Discuss the scope and lifetime of variables in C programming. What are global and local variables? |
| **Ans:**  In C programming, the scope and lifetime of variables determine where and for how long a variable remains accessible and usable within a program.  1. \*\*Scope of Variables\*\*:  - \*\*Scope\*\* refers to the region of the program where a variable is visible and accessible.  - There are mainly two types of scope in C:  - \*\*Global Scope\*\*: Variables declared outside of any function, typically at the top of the file, have global scope. Global variables can be accessed from any part of the program, including all functions.  - \*\*Local Scope\*\*: Variables declared within a function have local scope. They are accessible only within the block of code in which they are declared, such as within the function itself.  2. \*\*Lifetime of Variables\*\*:  - \*\*Lifetime\*\* refers to the duration for which a variable exists in memory during program execution.  - Like scope, there are two primary types of variable lifetimes:  - \*\*Static Lifetime\*\*: Variables with static storage duration exist throughout the entire execution of the program. This includes both global variables and variables declared with the `static` keyword inside functions. They retain their values between function calls.  - \*\*Automatic Lifetime\*\*: Variables with automatic storage duration exist only within the block in which they are declared. They are created when the block is entered and destroyed when the block is exited. This includes local variables declared without the `static` keyword inside functions.  - Additionally, memory for dynamic variables, such as those allocated using `malloc()` or `calloc()`, is allocated and deallocated explicitly by the programmer, typically with `free()`. The lifetime of dynamic variables depends on when they are allocated and deallocated.  3. \*\*Global and Local Variables\*\*:  - \*\*Global Variables\*\*:  - Declared outside of any function.  - Have global scope and static lifetime.  - Can be accessed and modified by any part of the program.  - Should be used judiciously to avoid unintended side effects and namespace pollution.  - \*\*Local Variables\*\*:  - Declared within a function or block.  - Have local scope and automatic lifetime.  - Accessible only within the function or block in which they are declared.  - Can shadow variables with the same name in outer scopes.  Here's an example illustrating global and local variables:  #include <stdio.h>  int globalVar = 10; // Global variable  void func() {  int localVar = 20; // Local variable    printf("Global variable: %d\n", globalVar);  printf("Local variable: %d\n", localVar);  }  int main() {  printf("Global variable: %d\n", globalVar);  // printf("Local variable: %d\n", localVar); // Error: localVar is not accessible here  func(); // Call function  return 0;  }  In this example, `globalVar` is accessible from both `main()` and `func()`, while `localVar` is accessible only within the `func()` function. |
| 1. Explain the concept of type casting in C programming. When is type casting necessary, and how is it performed? |
| **Ans:**  In C programming, type casting is the process of converting a value from one data type to another. This is often necessary when you want to perform operations that involve operands of different data types or when assigning a value of one data type to a variable of a different data type.  Type casting is necessary in the following situations:  1. \*\*Mixed Data Type Operations\*\*:  - When performing arithmetic or logical operations involving operands of different data types, C requires that the operands have the same data type. In such cases, you may need to cast one or more operands to ensure they have compatible types.  2. \*\*Assigning Values\*\*:  - When assigning a value of one data type to a variable of another data type, type casting may be necessary to avoid loss of data or precision.  3. \*\*Function Arguments\*\*:  - When calling a function with arguments of different data types than those expected by the function, type casting may be needed to match the function's parameter types.  Type casting in C can be performed using the following syntax:  (type) expression  where `(type)` specifies the desired data type to which the expression should be cast.  Here's an example illustrating type casting:  #include <stdio.h>  int main() {  int num1 = 10;  double num2 = 3.5;  double result;  // Performing mixed data type operation without type casting  result = num1 + num2; // Implicit conversion, num1 is promoted to double  printf("Result without casting: %lf\n", result);  // Performing mixed data type operation with type casting  result = num1 + (double)num2; // Type casting num2 to double  printf("Result with casting: %lf\n", result);  return 0;  }  In this example, `num1` is an integer, and `num2` is a double. When performing the addition operation without type casting, `num1` is implicitly converted to a double before the addition is performed. However, if you want to explicitly cast `num2` to an integer, you can use `(double)` to perform the conversion explicitly.  Type casting should be used judiciously and with caution to ensure that the resulting data type is appropriate for the intended use and to avoid potential loss of data or precision. |
| 1. Describe the purpose and usage of the ternary conditional operator (?:) in C programming. Provide an example demonstrating its usage. |
| **Ans:**  The ternary conditional operator `(?:)` is a compact way to write conditional expressions in C programming. It allows you to make decisions and choose between two values based on a condition. The ternary conditional operator has the following syntax:  condition ? expression1 : expression2  Here's how it works:  - The condition is evaluated first.  - If the condition is true, `expression1` is evaluated and becomes the result of the entire expression.  - If the condition is false, `expression2` is evaluated and becomes the result of the entire expression.  The ternary conditional operator is particularly useful when you want to assign a value to a variable based on a condition in a single line of code, without the need for an `if` statement.  Here's an example demonstrating the usage of the ternary conditional operator:  #include <stdio.h>  int main() {  int x = 10;  int y = 20;  int max;  // Using ternary conditional operator to find the maximum value  max = (x > y) ? x : y;  printf("The maximum value is: %d\n", max);  return 0;  }  In this example, we want to find the maximum value between `x` and `y`. We use the ternary conditional operator `(x > y) ? x : y` to evaluate the condition `x > y`. If this condition is true, the value of `x` is assigned to `max`; otherwise, the value of `y` is assigned. Finally, we print the maximum value.  The ternary conditional operator allows for concise and readable code, especially when the conditional expression is simple and the resulting operation is straightforward. However, it's essential to use it judiciously to maintain code clarity and avoid overly complex expressions. |
| 1. Discuss the bitwise operators available in C programming. Explain their usage with suitable examples. |
| **Ans:**  In C programming, bitwise operators are used to perform manipulation and operations at the bit level of variables. These operators are particularly useful in scenarios where you need to manipulate individual bits within integer types. There are six bitwise operators available in C:  1. \*\*Bitwise AND (`&`)\*\*:  - The bitwise AND operator performs a bitwise AND operation on each pair of corresponding bits of two operands.  - The result is 1 only if both bits are 1; otherwise, the result is 0.  - Example:    int a = 5; // 00000101 in binary  int b = 3; // 00000011 in binary  int result = a & b; // Result is 00000001 in binary, which is 1 in decimal  2. \*\*Bitwise OR (`|`)\*\*:  - The bitwise OR operator performs a bitwise OR operation on each pair of corresponding bits of two operands.  - The result is 1 if at least one of the bits is 1; otherwise, the result is 0.  - Example:  int a = 5; // 00000101 in binary  int b = 3; // 00000011 in binary  int result = a | b; // Result is 00000111 in binary, which is 7 in decimal  3. \*\*Bitwise XOR (`^`)\*\*:  - The bitwise XOR (exclusive OR) operator performs a bitwise XOR operation on each pair of corresponding bits of two operands.  - The result is 1 if the bits are different; otherwise, the result is 0.  - Example:    int a = 5; // 00000101 in binary  int b = 3; // 00000011 in binary  int result = a ^ b; // Result is 00000110 in binary, which is 6 in decimal    4. \*\*Bitwise NOT (`~`)\*\*:  - The bitwise NOT (complement) operator flips each bit of the operand.  - It reverses the bits; 1 becomes 0, and 0 becomes 1.  - Example:  int a = 5; // 00000101 in binary  int result = ~a; // Result is 11111010 in binary, which is -6 in decimal    5. \*\*Left Shift (`<<`)\*\*:  - The left shift operator shifts all bits in a number to the left by a specified number of positions.  - The vacated bits are filled with zeros.  - Example:  int a = 5; // 00000101 in binary  int result = a << 2; // Result is 00010100 in binary, which is 20 in decimal  6. \*\*Right Shift (`>>`)\*\*:  - The right shift operator shifts all bits in a number to the right by a specified number of positions.  - For unsigned types, the vacated bits are filled with zeros. For signed types, the vacated bits are filled with the sign bit (0 for positive numbers, 1 for negative numbers).  - Example:    int a = 20; // 00010100 in binary  int result = a >> 2; // Result is 00000101 in binary, which is 5 in decimal  Bitwise operators are commonly used in low-level programming, such as device driver development, cryptography, and optimization of algorithms involving bit-level manipulation. Understanding and using bitwise operators effectively can lead to more efficient and concise code in such scenarios. |
| 1. Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate |
| **Ans:**  In C programming, the increment operator `++` is used to increase the value of a variable by 1. There are two forms of the increment operator: postfix (`variable++`) and prefix (`++variable`). The difference between them lies in the order of operations and the value they return.  1. \*\*Postfix Increment Operator (`variable++`)\*\*:  - The postfix increment operator first returns the current value of the variable and then increments it.  - The value returned by the expression is the original value of the variable before the increment.  - Example:    int num = 5;  int result = num++; // Returns 5, then increments num to 6  printf("Result: %d\n", result); // Output: Result: 5  printf("Updated num: %d\n", num); // Output: Updated num: 6  ```  2. \*\*Prefix Increment Operator (`++variable`)\*\*:  - The prefix increment operator first increments the value of the variable and then returns the updated value.  - The value returned by the expression is the incremented value of the variable.  - Example:    int num = 5;  int result = ++num; // Increments num to 6, then returns 6  printf("Result: %d\n", result); // Output: Result: 6  printf("Updated num: %d\n", num); // Output: Updated num: 6  In both examples, `num` is initially 5. With the postfix increment operator `num++`, the original value of `num` (5) is returned, and then `num` is incremented to 6. Conversely, with the prefix increment operator `++num`, `num` is first incremented to 6, and then the updated value (6) is returned.  It's important to note that the behavior of postfix and prefix increment operators can lead to different results, especially when they are used as part of larger expressions or in loop constructs. Understanding the distinction between them is crucial for writing correct and predictable code |
| 1. What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions? |
| Ans:  In C programming, the logical AND (`&&`) and logical OR (`||`) operators are used to perform logical operations on boolean values or expressions. These operators are primarily used in conditional expressions to control the flow of program execution based on certain conditions. Here's the significance of each operator:  1. \*\*Logical AND (`&&`)\*\*:  - The logical AND operator (`&&`) returns true (1) if both operands evaluate to true; otherwise, it returns false (0).  - If the first operand of `&&` evaluates to false, the second operand is not evaluated because the entire expression will always be false regardless of the value of the second operand. This is known as short-circuit evaluation.  - Example:  ```c  int num = 10;  if (num > 0 && num < 20) {  printf("Number is between 0 and 20.\n");  }  ```  2. \*\*Logical OR (`||`)\*\*:  - The logical OR operator (`||`) returns true (1) if at least one of the operands evaluates to true; otherwise, it returns false (0).  - If the first operand of `||` evaluates to true, the second operand is not evaluated because the entire expression will always be true regardless of the value of the second operand. This is also known as short-circuit evaluation.  - Example:  ```c  int num = 10;  if (num < 0 || num > 20) {  printf("Number is outside the range of 0 to 20.\n");  }  ```  In both examples, `&&` and `||` are used in conditional expressions to evaluate whether certain conditions are met. If the condition evaluates to true, the corresponding block of code is executed; otherwise, it is skipped.  These logical operators are essential for controlling the flow of program execution, making decisions based on multiple conditions, and writing more complex conditional expressions. Understanding their behavior, including short-circuit evaluation, is crucial for writing efficient and correct C programs. |
| 1. Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation. |
| **Ans:**  In C programming, operator precedence and associativity are essential concepts that dictate the order in which operators are evaluated in expressions. Understanding these concepts is crucial for writing correct and predictable code.  1. \*\*Operator Precedence\*\*:  - Operator precedence determines the order in which different types of operators are evaluated within an expression.  - Operators with higher precedence are evaluated before operators with lower precedence.  - For example, multiplication (`\*`) has a higher precedence than addition (`+`), so multiplication operations are performed before addition operations.  - Parentheses can be used to explicitly specify the order of evaluation, overriding the default precedence rules.  - Here's a brief overview of some common operators and their precedence levels (from highest to lowest precedence):  1. Parentheses `( )`  2. Unary operators (e.g., `+`, `-`, `!`)  3. Multiplication, division, and remainder (`\*`, `/`, `%`)  4. Addition and subtraction (`+`, `-`)  5. Relational operators (e.g., `<`, `>`, `<=`, `>=`)  6. Equality operators (`==`, `!=`)  7. Logical AND (`&&`)  8. Logical OR (`||`)  9. Assignment operators (`=`, `+=`, `-=`, etc.)  2. \*\*Associativity\*\*:  - Associativity determines the order in which operators with the same precedence are evaluated.  - Operators can be either left-associative or right-associative.  - Left-associative operators are evaluated from left to right.  - Right-associative operators are evaluated from right to left.  - For example, the addition operator (`+`) is left-associative, so `a + b + c` is evaluated as `(a + b) + c`.  - Assignment operators (`=`, `+=`, `-=`) are right-associative, so `a = b = c` is evaluated as `a = (b = c)`.  Here are examples demonstrating how operator precedence and associativity affect expression evaluation:  ```c  #include <stdio.h> |
| int main() {  int result = 2 + 3 \* 4; // Multiplication has higher precedence than addition  printf("Result: %d\n", result); // Output: Result: 14  int result2 = (2 + 3) \* 4; // Parentheses override precedence  printf("Result2: %d\n", result2); // Output: Result2: 20  int result3 = 10 / 2 \* 3; // Left-associative: 10 / 2 first, then \* 3  printf("Result3: %d\n", result3); // Output: Result3: 15  int result4 = 10 - 5 + 2; // Left-associative: 10 - 5 first, then + 2  printf("Result4: %d\n", result4); // Output: Result4: 7  int a = 10;  int b = 5;  int c = 2;  int result5 = a = b = c; // Right-associative: b = c first, then a = (b = c)  printf("Result5: %d\n", result5); // Output: Result5: 2, and a, b, and c will all be 2  return 0;  }  ```  In each example, the output demonstrates how the expression is evaluated based on operator precedence and associativity. Understanding these concepts is crucial for writing expressions that produce the desired results in C programming.   1. Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement? |
| **Ans:**  The switch statement in C programming provides a way to execute different blocks of code based on the value of an expression. It is primarily used to simplify code that involves multiple nested if-else statements, especially when there are many possible values to check against. The switch statement can improve readability and maintainability in such scenarios.  Here's the basic syntax of the switch statement in C:  ```c  switch (expression) {  case constant1:  // Code block executed if expression equals constant1  break;  case constant2:  // Code block executed if expression equals constant2  break;  // More case statements for other constants...  default:  // Code block executed if expression doesn't match any case  break;  }  ```  Here's an explanation of how the switch statement works:  - The expression inside the switch statement is evaluated.  - The value of the expression is compared against each case constant.  - If the value matches a case constant, the corresponding code block is executed.  - The `break` statement is used to exit the switch statement after executing the code block associated with the matched case. This prevents fall-through to subsequent cases.  - If none of the case constants match the value of the expression, the code block associated with the `default` case (if present) is executed.  Here's an example demonstrating the usage of the switch statement:  ```c  #include <stdio.h>  int main() {  int choice;  printf("Enter a number between 1 and 3: ");  scanf("%d", &choice);  switch (choice) {  case 1:  printf("You chose option 1.\n");  break;  case 2:  printf("You chose option 2.\n");  break;  case 3:  printf("You chose option 3.\n");  break;  default:  printf("Invalid choice.\n");  break;  }  return 0;  }  ```  Now, let's discuss the differences between the switch statement and the if-else statement:  1. \*\*Usage\*\*:  - The switch statement is typically used when you have a single expression with multiple possible values to check against.  - The if-else statement is used for more general conditional branching, including cases where the condition involves complex boolean expressions.  2. \*\*Syntax\*\*:  - The switch statement provides a more concise syntax when checking multiple values against a single expression, especially when using many constant values.  - The if-else statement is more flexible and can handle a wider range of conditional expressions, including relational and logical operators.  3. \*\*Fall-through\*\*:  - In a switch statement, if a case block does not contain a `break` statement, control falls through to the next case. This can lead to unexpected behavior and is typically avoided.  - In an if-else statement, each condition is evaluated independently, and there's no concept of fall-through.  Overall, the choice between the switch statement and the if-else statement depends on the specific requirements of the code and the complexity of the conditions being evaluated. |

**Part- B**

|  |
| --- |
| 1. **Title of the program: Hello world** |
| Code  #include<stdio.h>  main()  {  printf("Hello world");  } |
| Output   |  | | --- | | **2.Title of the program: Factorial** |   Code  #include <stdio.h>  int main() {  int n, i;  unsigned long long fact = 1;  printf("Enter an integer: ");  scanf("%d", &n);  // shows error if the user enters a negative integer  if (n < 0)  printf("Error! Factorial of a negative number doesn't exist.");  else {  for (i = 1; i <= n; ++i) {  fact \*= i;  }  printf("Factorial of %d = %llu", n, fact);  }  return 0;  }  C:\Users\Asus\Pictures\Screenshots\Screenshot 2024-03-18 233358.pngOUTPUT     |  | | --- | | **3.Title of Program: Prime number** |   Code  #include <stdio.h>  int main() {  int n, i, flag = 0;  printf("Enter a positive integer: ");  scanf("%d", &n);  // 0 and 1 are not prime numbers  // change flag to 1 for non-prime number  if (n == 0 || n == 1)  flag = 1;  for (i = 2; i <= n / 2; ++i) {  // if n is divisible by i, then n is not prime  // change flag to 1 for non-prime number  if (n % i == 0) {  flag = 1;  break;  }  }  // flag is 0 for prime numbers  if (flag == 0)  printf("%d is a prime number.", n);  else  printf("%d is not a prime number.", n);  return 0;  }  C:\Users\Asus\Pictures\Screenshots\Screenshot 2024-03-18 234915.pngOUTPUT   |  | | --- | | **4.Title of program ;Fibonacci series** |   Code  #include<stdio.h>  void fib (int n);  main()  {  int n;  printf("Enter a number\n");  scanf("%d",&n);  fib(n);  }  void fib(int n)  {  int i;  int t1=0;  int t2=1;  int t3;  for(i=0;i<=n;i++)  {  t3=t1+t2;  printf("%d\n",t3);  t1=t2;  t2=t3;  }  }  OUTPUT   |  | | --- | | **5.Title of program ; Sum of digits** |   Code    #include<stdio.h>  int main()  {  int n,sum=0,m;  printf("Enter a number:");  scanf("%d",&n);  while(n>0)  {  m=n%10;  sum=sum+m;  n=n/10;  }  printf("Sum is=%d",sum);  return 0;  }  OUTPUT    **6.Title of program: Reverse a number**  Code  #include <stdio.h>  int main() {  int n, reverse = 0, remainder;  printf("Enter an integer: ");  scanf("%d", &n);  while (n != 0) {  remainder = n % 10;  reverse = reverse \* 10 + remainder;  n /= 10;  }  printf("Reversed number = %d", reverse);  return 0;  }  OUTPUT  **7.Title of program: Palindrom check**  code  #include<stdio.h>  int rev(int n);  main()  {  int n;  int reverse;  printf("Enter a number\n");  scanf("%d",&n);  reverse=rev(n);  if (reverse==n)  {  printf("palindrom");  }  else  {  printf("not a palindrom");  }  }  int rev(int n)  {  int ld;  int reverse=0;  while(n!=0)  {  ld=n%10;  reverse=reverse\*10+ld;  n=n/10;  }  return reverse;  }  OUTPUT  **8.Title of program: Area of triangle**  Code  #include <stdio.h>  int main() {  float base, height, area;  printf("Enter the base of the triangle: ");  scanf("%f", &base);  printf("Enter the height of the triangle: ");  scanf("%f", &height);  area = (base \* height) / 2;  printf("The area of the triangle is: %.2f\n", area);  return 0;  }  OUTPUT  **9.Title of program: Simple calculator**  code  #include <stdio.h>  int main() {  char op;  double first, second;  printf("Enter an operator (+, -, \*, /): ");  scanf("%c", &op);  printf("Enter two operands: ");  scanf("%lf %lf", &first, &second);  switch (op) {  case '+':  printf("%.1lf + %.1lf = %.1lf", first, second, first + second);  break;  case '-':  printf("%.1lf - %.1lf = %.1lf", first, second, first - second);  break;  case '\*':  printf("%.1lf \* %.1lf = %.1lf", first, second, first \* second);  break;  case '/':  printf("%.1lf / %.1lf = %.1lf", first, second, first / second);  break;  // operator doesn't match any case constant  default:  printf("Error! operator is not correct");  }  return 0;  }  OUTPUT  **10.Title of program: Array of operation**  code  #include<stdio.h>  void read(int a[],int n);  void maximum(int a[],int n);  void minimum(int a[],int n);  main()  {  int n;  int a[100];  printf("Enter the size of an arrays");  scanf("%d",&n);  printf("Enter the array elements\n");  read(a,n);  maximum(a,n);  minimum(a,n);  }  void read(int a[100],int n)  {  int i;  for(i=0;i<n;i++)  {  scanf("%d",&a[i]);  }  }  void maximum(int a[100],int n)  {  int i;  int max=a[0];  for(i=1;i<n;i++)  {  if(a[i]>max)  {  max=a[i];  }  }  printf("Maximum=%d\n",max);  }  void minimum(int a[100],int n)  {  int i;  int min=a[0];  for(i=1;i<n;i++)  {  if(a[i]<min)  {  min=a[i];  }  }  printf("Minimum=%d\n",min);  }  OUTPUT    **11. Title of program; string operation**  code  #include <stdio.h>  #include <string.h>  int main() {  char str1[20] = "hello";  char str2[20] = "world";  int result = strcmp(str1, str2);  if (result == 0) {  printf("The strings are equal\n");  }  else if (result > 0) {  printf("%s is greater than %s\n", str1, str2);  }  else {  printf("%s is less than %s\n", str1, str2);  }  return 0;  }  OUTPUT    **12.Title of program ; Linear search**  code  #include <stdio.h>  void main()  {  int num;  int i, key, element\_found = 0;  printf("Enter number of elements you would like to take as input: ");  scanf("%d", &num);  int arr[num];  printf("\nEnter all the elements of your choice:");  for (i = 0; i < num; i++)  {  scanf("%d", &arr[i]);  }  printf("\nEnter the key element that you would like to be searched: ");  scanf("%d", &key);  /\* Linear search starts \*/  for (i = 0; i < num ; i++)  {  if (key == arr[i] )  {  element\_found = 1;  break;  }  }  if (element\_found == 1)  printf("we got the element at index %d",i+1);  else  printf("we haven’t got element at any index in the array\n");  }  OUTPUT  **13. Title of program; Binary search**  code  #include <stdio.h>  int main()  {  int c, first, last, middle, n, search, array[100];  printf("Enter number of elements\n");  scanf("%d", &n);  printf("Enter %d integers\n", n);  for (c = 0; c < n; c++)  scanf("%d", &array[c]);  printf("Enter value to find\n");  scanf("%d", &search);  first = 0;  last = n - 1;  middle = (first+last)/2;  while (first <= last) {  if (array[middle] < search)  first = middle + 1;  else if (array[middle] == search) {  printf("%d found at location %d.\n", search, middle+1);  break;  }  else  last = middle - 1;  middle = (first + last)/2;  }  if (first > last)  printf("Not found! %d isn't present in the list.\n", search);  return 0;  }  OUTPUT  **14. Title of program; Selection sort**  code  // C program for implementation of selection sort  #include <stdio.h>  void swap(int\* xp, int\* yp)  {  int temp = \*xp;  \*xp = \*yp;  \*yp = temp;  }  void selectionSort(int arr[], int n)  {  int i, j, min\_idx;  // One by one move boundary of unsorted subarray  for (i = 0; i < n - 1; i++) {  // Find the minimum element in unsorted array  min\_idx = i;  for (j = i + 1; j < n; j++)  if (arr[j] < arr[min\_idx])  min\_idx = j;  // Swap the found minimum element with the first  // element  swap(&arr[min\_idx], &arr[i]);  }  }  /\* Function to print an array \*/  void printArray(int arr[], int size)  {  int i;  for (i = 0; i < size; i++)  printf("%d ", arr[i]);  printf("\n");  }  // Driver program to test above functions  int main()  {  int arr[] = { 64, 25, 12, 22, 11 };  int n = sizeof(arr) / sizeof(arr[0]);  selectionSort(arr, n);  printf("Sorted array: \n");  printArray(arr, n);  return 0;  }  OUTPUT  **15. Title of program; Bubble sort**  code  #include<stdio.h>  void read(int a[],int n);  void display(int a[],int n);  void sort(int a[], int n);  main()  {  int a[10];  int n,k;  printf("Enter the number of elements in arrays\n");  scanf("%d%d",&n,&k);  read(a,n);  printf("Enterd array is\n");  display(a,n);  sort(a,n);  }  void read(int a[10],int n)  {  int i;  for(i=0;i<n;i++)  {  scanf("%d",&a[i]);  }  }  void display(int a[10],int n)  {  int i;  for(i=0;i<n;i++)  {  printf("%d\t",a[i]);  }  }  void sort(int a[10],int n)  {  int i;  int j;  int temp;  int k;  for(i=0;i<n;i++)  {  for(j=i+1;j<n;j++)  {  if(a[i]>a[j])  {  temp=a[i];  a[i]=a[j];  a[j]=temp;  }  } printf("\n");  }  printf("The asecending order of array=\n");  {  for(i=0;i<n;i++)  {  printf("%d\t",a[i]);  }  }  }  OUTPUT    **16. Title of program; insertion sort**  Code  #include <stdio.h>  void insertionSort(int arr[], int n) {  int i, key, j;  for (i = 1; i < n; i++) {  key = arr[i];  j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key;  }  }  int main() {  int arr[] = { 12, 11, 13, 5, 6 };  int n = sizeof(arr) / sizeof(arr[0]);  int i;  insertionSort(arr, n);  for ( i = 0; i < n; i++)  printf("%d ", arr[i]);  printf("\n");  return 0;  }  OUTPUT  **17. Title of program: matrix opreation**  code  #include <stdio.h>  #define N 3 // Define matrix size, adjust as needed  // Function prototypes  void printMatrix(int matrix[N][N]);  void addMatrices(int A[N][N], int B[N][N], int result[N][N]);  void subtractMatrices(int A[N][N], int B[N][N], int result[N][N]);  void multiplyMatrices(int A[N][N], int B[N][N], int result[N][N]);  void transposeMatrix(int A[N][N], int result[N][N]);  int main() {  int A[N][N] = {{1, 2, 3},  {4, 5, 6},  {7, 8, 9}};  int B[N][N] = {{9, 8, 7},  {6, 5, 4},  {3, 2, 1}};  int result[N][N]; // To store the result  int i,j;  // Matrix Addition  addMatrices(A, B, result);  printf("Addition of A and B:\n");  printMatrix(result);  // Matrix Subtraction  subtractMatrices(A, B, result);  printf("\nSubtraction of A and B:\n");  printMatrix(result);  // Matrix Multiplication  multiplyMatrices(A, B, result);  printf("\nMultiplication of A and B:\n");  printMatrix(result);  // Matrix Transpose  transposeMatrix(A, result);  printf("\nTranspose of A:\n");  printMatrix(result);  return 0;  }  // Function to print a matrix  void printMatrix(int matrix[N][N]) {  for ( i = 0; i < N; i++) {  for (j = 0; j < N; j++) {  printf("%d ", matrix[i][j]);  }  printf("\n");  }  }  // Function to add two matrices  void addMatrices(int A[N][N], int B[N][N], int result[N][N]) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  result[i][j] = A[i][j] + B[i][j];  }  }  }  // Function to subtract two matrices  void subtractMatrices(int A[N][N], int B[N][N], int result[N][N]) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  result[i][j] = A[i][j] - B[i][j];  }  }  }  // Function to multiply two matrices  void multiplyMatrices(int A[N][N], int B[N][N], int result[N][N]) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  result[i][j] = 0;  for (int k = 0; k < N; k++) {  result[i][j] += A[i][k] \* B[k][j];  }  }  }  }  // Function to transpose a matrix  void transposeMatrix(int A[N][N], int result[N][N]) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  result[j][i] = A[i][j];  }  }  }  OUTPUT |

**Part- C**

|  |
| --- |
| **18.Title of the program: LINKED LIST OPREATION** |
| 1. 1)Amazon Logistics (AMZL) delivers packages and smileys to Amazon customers every day. On the occasion of the Dussehra   and Diwali festivals, the amazon has launched Amazon Great Indian Festival.  To cope with the large number of orders, delivery associate prepares the list of orders based on location.  Every time he finds a new packet he adds details at the end of the list. A delivery associate will deliver packets to  specified address. Everyday, delivery associate starts distribution of the orders at 10.00 am in the Lingarajnagar colony  (in the order, houses are connected) and returns to the stock office at 5.00 pm.  Before submitting the summary to the manager he will calculate the amount collected from delivered packets. End of the  day he removes the packet where the price is less than 500\*/  #include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct node  {  char name[20],month[20];  int hno,date,year;  float amt;  struct node \*link;  };  struct node\* create()  {  struct node \*newnode=NULL;  newnode=(struct node\*)malloc(sizeof(struct node));  if(newnode==NULL)  printf("Memory not allocated");  else  {  scanf("%s%d%d%s%d%f",newnode->name,&newnode->hno,&newnode->date,newnode->month,&newnode->year,&newnode->amt);  newnode->link=newnode;  }  return newnode;  }  struct node\* insertendcsll(struct node \*head)  {  struct node \*cur=head;  struct node \*newnode=create();  if(head==NULL)  head=newnode;  else  {  while(cur->link!=head)  cur=cur->link;  cur->link=newnode;  newnode->link=head;  }  return head;  }  void displaycsll(struct node \*head)  {  struct node \*cur=head;  if(head==NULL)  printf("Empty.");  else  {  do  {  printf("%s %d %d %s %d %0.2f\n",cur->name,cur->hno,cur->date,cur->month,cur->year,cur->amt);  cur=cur->link;  }  while(cur!=head);  }  }  float totalamt(struct node \*head)  {  struct node \*cur=head;  float amount=0;  if(head==NULL)  printf("Empty.");  else  {  do  {  amount=amount+cur->amt;  cur=cur->link;  }  while(cur!=head);  }  return amount;  }  struct node\* deleteorder(struct node \*head)  {  struct node \*cur=head,\*temp=head,\*prev=head;  int s=0;  if(head==NULL)  return NULL;  else if(head->link==head&&head->amt<500)  { s=1;  printf("%s %d %d %s %d %0.2f\n",head->name,head->hno,head->date,head->month,head->year,head->amt);  free(head);  head=NULL;  }  else if(head->amt<500)  { s=1;  printf("%s %d %d %s %d %0.2f\n",head->name,head->hno,head->date,head->month,head->year,head->amt);  while(cur->link!=head)  cur=cur->link;  cur->link=head->link;  head=head->link;  free(temp);  }  else  {  cur=head;  do  {  if(cur->amt<500)  {  s=1;  printf("%s %d %d %s %d %0.2f\n",cur->name,cur->hno,cur->date,cur->month,cur->year,cur->amt);  break;  }  prev=cur;  cur=cur->link;  }  while(cur!=head);  if(s==0)  return head;  else  {  prev->link=cur->link;  free(cur);  }  }  return head;  }  int main() {  struct node \*head=NULL;  float amount;  int n,i;  scanf("%d",&n);  if(n>0)  {  for(i=0;i<n;i++)  head=insertendcsll(head);  amount=totalamt(head);  printf("Total amount: Rs.%0.2f\n",amount);  printf("\nDeleted order:\n");  head=deleteorder(head);  printf("\nFinal orders list:\n");  displaycsll(head);  }  else  printf("Invalid input");  return 0;  }  OUTPUT   1. **Insertion at the beginning in singly linked list**   Code  #include<stdio.h>  #include<stdlib.h>  void beginsert(int);  struct node  {  int data;  struct node \*next;  };  struct node \*head;  void main ()  {  int choice,item;  do  {  printf("\nEnter the item which you want to insert?\n");  scanf("%d",&item);  beginsert(item);  printf("\nPress 0 to insert more ?\n");  scanf("%d",&choice);  }while(choice == 0);  }  void beginsert(int item)  {  struct node \*ptr = (struct node \*)malloc(sizeof(struct node \*));  if(ptr == NULL)  {  printf("\nOVERFLOW\n");  }  else  {  ptr->data = item;  ptr->next = head;  head = ptr;  printf("\nNode inserted\n");  }  }   1. **Insertion a node at the end of linked list**   Code  #include <stdio.h>  #include <stdlib.h>  // Define the structure for a node  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function to create a new node with given data  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node)); // Allocate memory for the new node  if (newNode == NULL) {  printf("Memory allocation failed\n");  exit(1); // Exit if memory allocation fails  }  newNode->data = data; // Set the data part  newNode->next = NULL; // Next is NULL for the new last node  return newNode;  }  // Function to insert a node at the end of the list  void insertEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data); // Create a new node with the given data  if (\*head == NULL) {  // If the list is empty, make the new node as head  \*head = newNode;  } else {  // Traverse to the last node  Node\* last = \*head;  while (last->next != NULL) {  last = last->next;  }  // Insert the new node at the end  last->next = newNode;  }  }  // Function to print the list  void printList(Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  // Main function  int main() {  Node\* head = NULL; // Start with the empty list  // Inserting nodes at the end of the list  insertEnd(&head, 1);  insertEnd(&head, 2);  insertEnd(&head, 3);  insertEnd(&head, 4);  printf("Linked list: ");  printList(head);  // Remember to free the allocated memory  Node\* current = head;  Node\* next;  while (current != NULL) {  next = current->next;  free(current);  current = next;  }  return 0;  }  OUTPUT   1. **Delete a node from beginning in singly linked list**   Code  #include <stdio.h>  #include <stdlib.h>  // Define the structure for a node  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function to create a new node with given data  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node)); // Allocate memory for the new node  if (!newNode) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE); // Exit if memory allocation fails  }  newNode->data = data; // Set the data  newNode->next = NULL; // The next pointer of a new node is always NULL  return newNode;  }  // Function to insert a node at the end of the list  void insertEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data); // Create a new node  if (\*head == NULL) {  // If the list is empty, the new node is the head  \*head = newNode;  } else {  // Traverse to the last node  Node\* last = \*head;  while (last->next != NULL) {  last = last->next;  }  // Insert the new node at the end  last->next = newNode;  }  }  // Function to delete the first node  void deleteBeginning(Node\*\* head) {  if (\*head == NULL) {  printf("The list is already empty.\n");  return;  }  Node\* temp = \*head; // Temporarily store the head to free it later  \*head = (\*head)->next; // Change head to the second node  free(temp); // Free the old head  }  // Function to print the list  void printList(Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  // Main function  int main() {  Node\* head = NULL; // Start with an empty list  // Inserting nodes at the end of the list for demonstration  insertEnd(&head, 10);  insertEnd(&head, 20);  insertEnd(&head, 30);  printf("Initial list: ");  printList(head);  // Deleting a node from the beginning  deleteBeginning(&head);  printf("List after deleting the first node: ");  printList(head);  // Free remaining nodes to avoid memory leaks  while (head != NULL) {  deleteBeginning(&head);  }  return 0;  }  OUTPU   1. **Delete a node from the end in singly linked list using c programming**   **Code**  #include <stdio.h>  #include <stdlib.h>  // Define the structure for a node  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function to create a new node with given data  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node)); // Allocate memory for the new node  if (!newNode) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE); // Exit if memory allocation fails  }  newNode->data = data; // Set the data  newNode->next = NULL; // The next pointer of a new node is always NULL  return newNode;  }  // Function to insert a node at the end of the list  void insertEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data); // Create a new node  if (\*head == NULL) {  // If the list is empty, the new node is the head  \*head = newNode;  } else {  // Traverse to the last node  Node\* last = \*head;  while (last->next != NULL) {  last = last->next;  }  // Insert the new node at the end  last->next = newNode;  }  }  // Function to delete the last node of the list  void deleteEnd(Node\*\* head) {  if (\*head == NULL) {  printf("The list is already empty.\n");  return;  }  // If the list only has one node  if ((\*head)->next == NULL) {  free(\*head); // Free the node  \*head = NULL; // Update the head to NULL  return;  }  // Traverse to the second-to-last node  Node\* temp = \*head;  while (temp->next->next != NULL) {  temp = temp->next;  }  // Free the last node and update the second-to-last node's next to NULL  free(temp->next);  temp->next = NULL;  }  // Function to print the list  void printList(Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  // Main function  int main() {  Node\* head = NULL; // Start with an empty list  // Inserting nodes at the end of the list for demonstration  insertEnd(&head, 10);  insertEnd(&head, 20);  insertEnd(&head, 30);  printf("Initial list: ");  printList(head);  // Deleting the last node  deleteEnd(&head);  printf("List after deleting the last node: ");  printList(head);  // Deleting remaining nodes to avoid memory leaks  while (head != NULL) {  deleteEnd(&head);  }  return 0;  }  OUTPUT   1. **Add a node to specific positon in singly linked list using c programming**   Code  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtPosition(Node\*\* head, int data, int position);  void displayList(Node\* head);  void freeList(Node\*\* head);  int main() {  Node\* head = NULL; // Initially, the list is empty.  int i;  // Example usage  insertAtPosition(&head, 10, 1); // Insert 10 at position 1.  insertAtPosition(&head, 20, 2); // Insert 20 at position 2.  insertAtPosition(&head, 5, 1); // Insert 5 at position 1.  insertAtPosition(&head, 15, 3); // Insert 15 at position 3.  printf("Linked list: ");  displayList(head);  // Free the allocated memory  freeList(&head);  return 0;  }  // Create a new Node  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(EXIT\_FAILURE);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Insert a node at a specific position in the list  void insertAtPosition(Node\*\* head, int data, int position) { int i;  Node\* newNode = createNode(data);  if (position == 1) {  newNode->next = \*head;  \*head = newNode;  } else {  Node\* temp = \*head;  for ( i = 1; temp != NULL && i < position - 1; i++) {  temp = temp->next;  }  if (temp == NULL) {  printf("The previous node is null.\n");  } else {  newNode->next = temp->next;  temp->next = newNode;  }  }  }  // Display the linked list  void displayList(Node\* head) {  Node\* temp = head;  while (temp != NULL) {  printf("%d -> ", temp->data);  temp = temp->next;  }  printf("NULL\n");  }  // Free the allocated memory for the list  void freeList(Node\*\* head) {  Node\* temp;  while (\*head != NULL) {  temp = \*head;  \*head = (\*head)->next;  free(temp);  }  }  OUTPUT   1. **Delete a node from specified location in singly linked list c programming**   Code    #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtEnd(Node\*\* head, int data); // Helper function to populate the list  void deleteAtPosition(Node\*\* head, int position);  void displayList(Node\* node);  void freeList(Node\*\* head);  int main() {  Node\* head = NULL;  // Populating the list  insertAtEnd(&head, 10);  insertAtEnd(&head, 20);  insertAtEnd(&head, 30);  insertAtEnd(&head, 40);  insertAtEnd(&head, 50);  printf("Initial List: ");  displayList(head);  // Delete a node at a specific position  deleteAtPosition(&head, 3); // Deletes the node at position 3  printf("List after deleting node at position 3: ");  displayList(head);  // Free the allocated memory  freeList(&head);  return 0;  }  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL) {  printf("Memory allocation error\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  Node\* last = \*head;  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  void deleteAtPosition(Node\*\* head, int position) { int i;  if (\*head == NULL) {  printf("List is empty.\n");  return;  }  Node\* temp = \*head;  if (position == 1) {  \*head = temp->next; // Change head  free(temp); // free old head  return;  }  // Find previous node of the node to be deleted  for ( i = 1; temp != NULL && i < position - 1; i++) {  temp = temp->next;  }  if (temp == NULL || temp->next == NULL) {  printf("Position does not exist.\n");  return;  }  // Node temp->next is the node to be deleted  // Store pointer to the next of node to be deleted  Node\* next = temp->next->next;  free(temp->next); // Free memory  temp->next = next; // Unlink the deleted node from list  }  void displayList(Node\* node) {  while (node != NULL) {  printf("%d -> ", node->data);  node = node->next;  }  printf("NULL\n");  }  void freeList(Node\*\* head) {  Node\* current = \*head;  Node\* next;  while (current != NULL) {  next = current->next;  free(current);  current = next;  }  \*head = NULL;  }  OUTPUT   1. **Delete a node from end in Doubly linked list using c programming**   Code  #include <stdio.h>  #include <stdlib.h>  // Define the structure for a node in a doubly linked list  typedef struct Node {  int data;  struct Node\* next;  struct Node\* prev;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtEnd(Node\*\* head, int data);  void displayList(Node\* head);  int main() {  Node\* head = NULL; // Initially, the list is empty  // Inserting nodes at the end of the list  insertAtEnd(&head, 10);  insertAtEnd(&head, 20);  insertAtEnd(&head, 30);  insertAtEnd(&head, 40);  // Displaying the list  printf("Doubly Linked List: ");  displayList(head);  // Freeing allocated memory is left as an exercise  // Remember to iterate through the list and free each node  return 0;  }  // Function to create a new node with given data  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(1); // Exit program if memory allocation fails  }  newNode->data = data;  newNode->next = NULL; // Next pointer of a new node is always NULL  newNode->prev = NULL; // Prev pointer is also NULL initially  return newNode;  }  // Function to insert a node at the end of the doubly linked list  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  // If the list is empty, the new node is the head  \*head = newNode;  return;  }  Node\* last = \*head;  // Traverse to the last node of the list  while (last->next != NULL) {  last = last->next;  }  // Update pointers to insert the new node at the end  last->next = newNode;  newNode->prev = last;  }  // Function to display the list from the beginning  void displayList(Node\* node) {  while (node != NULL) {  printf("%d <-> ", node->data);  node = node->next;  }  printf("NULL\n");  }  OUTPUT   1. **Add a node at the beginning in doubly linked list**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  struct Node\* prev;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtBeginning(Node\*\* head, int data);  void displayList(Node\* head);  int main() {  Node\* head = NULL; // Initially, the list is empty.  // Inserting nodes at the beginning of the list  insertAtBeginning(&head, 30);  insertAtBeginning(&head, 20);  insertAtBeginning(&head, 10);  // Displaying the list  printf("Doubly Linked List: ");  displayList(head);  // Freeing allocated memory is left as an exercise.  // Remember to iterate through the list and free each node.  return 0;  }  // Function to create a new node with given data  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(1); // Exit program if memory allocation fails  }  newNode->data = data;  newNode->next = NULL;  newNode->prev = NULL;  return newNode;  }  // Function to insert a node at the beginning of the doubly linked list  void insertAtBeginning(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  // If the list is empty, make the new node the head  \*head = newNode;  } else {  // Adjust pointers to insert the new node at the beginning  newNode->next = \*head;  (\*head)->prev = newNode;  \*head = newNode;  }  }  // Function to display the list from the beginning  void displayList(Node\* node) {  while (node != NULL) {  printf("%d <-> ", node->data);  node = node->next;  }  printf("NULL\n");  }  OUTPUT   1. **Insert a node at End of doubly linked list using c programming**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  struct Node\* prev;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtEnd(Node\*\* head, int data);  void displayList(Node\* head);  int main() {  Node\* head = NULL; // Initially, the list is empty  // Inserting nodes at the end of the list  insertAtEnd(&head, 10);  insertAtEnd(&head, 20);  insertAtEnd(&head, 30);  insertAtEnd(&head, 40);  // Displaying the list  printf("Doubly Linked List: ");  displayList(head);  // Note: Freeing allocated memory is important to avoid memory leaks.  // This step is left as an exercise for the reader.  return 0;  }  // Function to create a new node with given data  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(1); // Exit program if memory allocation fails  }  newNode->data = data;  newNode->next = NULL;  newNode->prev = NULL;  return newNode;  }  // Function to insert a node at the end of the doubly linked list  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  // If the list is empty, make the new node the head  \*head = newNode;  return;  }  Node\* last = \*head;  // Traverse to the last node of the list  while (last->next != NULL) {  last = last->next;  }  // Adjust pointers to insert the new node at the end  last->next = newNode;  newNode->prev = last;  }  // Function to display the list from the beginning  void displayList(Node\* node) {  while (node != NULL) {  printf("%d <-> ", node->data);  node = node->next;  }  printf("NULL\n");  }  OUTPUT   1. **Delete a node from beginning of doubly linked list using c programming**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  struct Node\* prev;  } Node;  // Function prototypes  void insertAtEnd(Node\*\* head, int data);  void deleteFromBeginning(Node\*\* head);  void displayList(Node\* head);  Node\* createNode(int data);  int main() {  Node\* head = NULL; // Initially, the list is empty  // Inserting nodes at the end for demonstration  insertAtEnd(&head, 10);  insertAtEnd(&head, 20);  insertAtEnd(&head, 30);  insertAtEnd(&head, 40);  printf("Initial Doubly Linked List: ");  displayList(head);  // Deleting a node from the beginning  deleteFromBeginning(&head);  printf("After Deleting from Beginning: ");  displayList(head);  // Note: The program does not include code for freeing memory,  // which is necessary to avoid memory leaks in a real application.  return 0;  }  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Failed to allocate memory.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  newNode->prev = NULL;  return newNode;  }  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  Node\* temp = \*head;  while (temp->next != NULL) {  temp = temp->next;  }  temp->next = newNode;  newNode->prev = temp;  }  void deleteFromBeginning(Node\*\* head) {  if (\*head == NULL) {  printf("The list is already empty.\n");  return;  }  Node\* temp = \*head;  if (temp->next == NULL) {  // Only one node in the list  \*head = NULL;  } else {  \*head = temp->next;  (\*head)->prev = NULL;  }  free(temp);  }  void displayList(Node\* node) {  while (node != NULL) {  printf("%d <-> ", node->data);  node = node->next;  }  printf("NULL\n");  }  OUTPUT   1. **Add a node at specified location in doubly linked list using c programming**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  struct Node\* prev;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtPosition(Node\*\* head, int data, int position);  void displayList(Node\* head);  int main() {  Node\* head = NULL; // Initially, the list is empty.  // Example usage  insertAtPosition(&head, 10, 1); // Insert 10 at position 1  insertAtPosition(&head, 20, 2); // Insert 20 at position 2  insertAtPosition(&head, 15, 2); // Insert 15 at position 2  insertAtPosition(&head, 5, 1); // Insert 5 at position 1  insertAtPosition(&head, 25, 5); // Insert 25 at position 5 (end)  // Note: Positions are 1-based  // Displaying the list  printf("Doubly Linked List: ");  displayList(head);  // Freeing allocated memory is left as an exercise.  // Iterate through the list and free each node.  return 0;  }  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  newNode->prev = NULL;  return newNode;  }  void insertAtPosition(Node\*\* head, int data, int position) { int i;  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode; // List is empty, make newNode the head  } else if (position == 1) {  // Insert at beginning  newNode->next = \*head;  (\*head)->prev = newNode;  \*head = newNode;  } else {  Node\* temp = \*head;  for ( i = 1; i < position - 1 && temp->next != NULL; i++) {  temp = temp->next;  }  // Insert at the end  if (temp->next == NULL) {  temp->next = newNode;  newNode->prev = temp;  } else {  // Insert in the middle  newNode->next = temp->next;  newNode->prev = temp;  temp->next->prev = newNode;  temp->next = newNode;  }  }  }  void displayList(Node\* node) {  while (node != NULL) {  printf("%d <-> ", node->data);  node = node->next;  }  printf("NULL\n");  }  OUTPUT   1. **Delete a node from specified location in doubly linked list using c**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  struct Node\* prev;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtEnd(Node\*\* head, int data);  void deleteAtPosition(Node\*\* head, int position);  void displayList(Node\* node);  void freeList(Node\*\* head);  int main() {  Node\* head = NULL;  // Inserting nodes at the end for demonstration  insertAtEnd(&head, 10);  insertAtEnd(&head, 20);  insertAtEnd(&head, 30);  insertAtEnd(&head, 40);  insertAtEnd(&head, 50);  printf("Initial Doubly Linked List: ");  displayList(head);  // Deleting a node at a specified position  int position = 3; // Example position  printf("Deleting node at position %d...\n", position);  deleteAtPosition(&head, position);  printf("Doubly Linked List after Deletion: ");  displayList(head);  // Freeing allocated memory  freeList(&head);  return 0;  }  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  newNode->prev = NULL;  return newNode;  }  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  } else {  Node\* last = \*head;  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  newNode->prev = last;  }  }  void deleteAtPosition(Node\*\* head, int position) {  if (\*head == NULL) {  return; // List is empty  }  Node\* temp = \*head;  if (position == 1) {  // Delete the first node  \*head = temp->next;  if (\*head != NULL) {  (\*head)->prev = NULL;  }  free(temp);  return;  }  for (int i = 1; temp != NULL && i < position; i++) {  temp = temp->next;  }  if (temp == NULL) {  return; // Position is beyond the length of the list  }  if (temp->next != NULL) {  temp->next->prev = temp->prev;  }  if (temp->prev != NULL) {  temp->prev->next = temp->next;  }  free(temp);  }  void displayList(Node\* node) {  while (node != NULL) {  printf("%d <-> ", node->data);  node = node->next;  }  printf("NULL\n");  }  void freeList(Node\*\* head) {  Node\* temp;  while (\*head != NULL) {  temp = \*head;  \*head = (\*head)->next;  free(temp);  }  }  OUTPUT     1. **Add a node at specified location in singly circular linked list using c programming**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtPosition(Node\*\* head, int data, int position);  void displayList(Node\* head);  int main() {  Node\* head = NULL; // Initially, the list is empty.  // Inserting nodes for demonstration. The list will end up with 10 -> 20 -> 30 -> 10  insertAtPosition(&head, 10, 1); // Insert 10 at position 1  insertAtPosition(&head, 20, 2); // Insert 20 at position 2  insertAtPosition(&head, 30, 3); // Insert 30 at position 3  // Inserting a node at a specific position  insertAtPosition(&head, 15, 2); // Insert 15 at position 2  // Displaying the list  printf("Singly Circular Linked List: ");  displayList(head);  // Freeing allocated memory is left as an exercise.  // Note: Be careful with freeing nodes in a circular linked list to avoid infinite loops.  return 0;  }  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  void insertAtPosition(Node\*\* head, int data, int position) { int i;  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = newNode; // Points to itself, making it circular  } else if (position == 1) {  // Insert at beginning  Node\* temp = \*head;  while (temp->next != \*head) { // Find the last node  temp = temp->next;  }  newNode->next = \*head;  \*head = newNode;  temp->next = \*head; // Update last node to point to new head  } else {  Node\* temp = \*head;  for ( i = 1; i < position - 1 && temp->next != \*head; i++) {  temp = temp->next;  }  newNode->next = temp->next;  temp->next = newNode;  }  }  void displayList(Node\* head) {  if (head == NULL) {  return;  }  Node\* temp = head;  do {  printf("%d -> ", temp->data);  temp = temp->next;  } while (temp != head);  printf("(%d as head)\n", head->data); // Indicate the start of the list  }  OUTPUT   1. **Delete a node from specified location in singly circular linked list**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function prototypes  Node\* createNode(int data);  void insertAtEnd(Node\*\* head, int data); // Helper function to populate the list  void deleteAtPosition(Node\*\* head, int position);  void displayList(Node\* head);  int main() {  Node\* head = NULL; // Initially, the list is empty.  // Inserting nodes for demonstration  insertAtEnd(&head, 10);  insertAtEnd(&head, 20);  insertAtEnd(&head, 30);  insertAtEnd(&head, 40);  printf("Initial Singly Circular Linked List: ");  displayList(head);  // Deleting a node at a specified position  int position = 3; // Example: delete the node at position 3  deleteAtPosition(&head, position);  printf("Singly Circular Linked List after Deletion: ");  displayList(head);  // Freeing allocated memory is left as an exercise.  // Be careful to avoid infinite loops when freeing a circular linked list.  return 0;  }  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head; // Make it circular  } else {  Node\* temp = \*head;  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode;  newNode->next = \*head; // Connect the new node back to the head  }  }  void deleteAtPosition(Node\*\* head, int position) {  if (\*head == NULL) {  printf("List is empty.\n");  return;  }  Node \*temp = \*head, \*prev = NULL;  if (position == 1) {  // Deleting the head node  while (temp->next != \*head) { // Find the last node  temp = temp->next;  }  // If there's only one node in the list  if (temp == \*head) {  free(temp);  \*head = NULL;  } else {  temp->next = (\*head)->next;  free(\*head);  \*head = temp->next;  }  } else {  // Find the node at the specified position and its previous node  int i;  for (i = 1; temp->next != \*head && i < position; i++) {  prev = temp;  temp = temp->next;  }  if (i != position) {  printf("Position exceeds list size.\n");  return;  }  prev->next = temp->next;  free(temp);  }  }  void displayList(Node\* head) {  if (head == NULL) {  printf("List is empty.\n");  return;  }  Node\* temp = head;  do {  printf("%d -> ", temp->data);  temp = temp->next;  } while (temp != head);  printf("(%d as head)\n", head->data); // To indicate the start of the list  }  OUTPUT   1. **Inserting a node at the beginning of the singly circular linked list**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function to create a new node  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation error!\n");  exit(-1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to add a node at the beginning of the list  void insertAtBeginning(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head; // Points to itself, making the list circular  } else {  Node\* temp = \*head;  // Traverse till the last node  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode; // Point the last node to the new node  newNode->next = \*head; // Point the new node to the old head  \*head = newNode; // Update the head to point to the new node  }  }  // Function to print the contents of the circular linked list  void printList(Node\* head) {  if (head == NULL) {  printf("The list is empty.\n");  return;  }  Node\* temp = head;  do {  printf("%d -> ", temp->data);  temp = temp->next;  } while (temp != head);  printf("(head node data: %d)\n", head->data);  }  int main() {  Node\* head = NULL; // Start with an empty list  // Adding nodes at the beginning  insertAtBeginning(&head, 10); // List: 10  insertAtBeginning(&head, 20); // List: 20 -> 10  insertAtBeginning(&head, 30); // List: 30 -> 20 -> 10  printf("Singly Circular Linked List: ");  printList(head);  // Freeing the allocated memory is left as an exercise to the reader  // Be careful to avoid memory leaks  return 0;  }  OUTPUT’     1. **Delete a node from the End of singly circular linked list using c program**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function to create a new node  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation error!\n");  exit(-1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to add a node at the end of the list - helps in demonstrating deletion  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head; // Points to itself, making the list circular  } else {  Node\* temp = \*head;  // Traverse till the last node  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode; // Point the last node to the new node  newNode->next = \*head; // Point the new node to the head  }  }  // Function to delete the node at the beginning of the list  void deleteAtBeginning(Node\*\* head) {  if (\*head == NULL) {  printf("The list is empty. Nothing to delete.\n");  return;  }  Node\* temp = \*head;  if (temp->next == \*head) { // Only one node in the list  free(temp);  \*head = NULL;  } else {  Node\* last = \*head;  // Find the last node  while (last->next != \*head) {  last = last->next;  }  \*head = (\*head)->next; // Move head pointer to the next node  last->next = \*head; // Update the last node's next pointer to the new head  free(temp); // Free the old head  }  }  // Function to print the contents of the circular linked list  void printList(Node\* head) {  if (head == NULL) {  printf("The list is empty.\n");  return;  }  Node\* temp = head;  do {  printf("%d -> ", temp->data);  temp = temp->next;  } while (temp != head);  printf("(head node data: %d)\n", head->data);  }  int main() {  Node\* head = NULL; // Start with an empty list  // Adding nodes to demonstrate deletion  insertAtEnd(&head, 10); // List: 10  insertAtEnd(&head, 20); // List: 10 -> 20  insertAtEnd(&head, 30); // List: 10 -> 20 -> 30  printf("Singly Circular Linked List before deletion: ");  printList(head);  // Deleting the node at the beginning  deleteAtBeginning(&head);  printf("Singly Circular Linked List after deletion: ");  printList(head);  // Note: Freeing the remaining allocated memory is left as an exercise to the reader  // Be careful to avoid memory leaks  return 0;  }  OUTPUT   1. **Insert a node at End of singly circular linked list using c program**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function to create a new node  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to add a node at the end of the list  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head; // Point to itself, making the list circular  } else {  Node\* temp = \*head;  // Find the last node  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode; // Add the new node at the end  newNode->next = \*head; // Point the new node to the head, maintaining circularity  }  }  // Function to print the list  void printList(Node\* head) {  if (head == NULL) {  printf("The list is empty.\n");  return;  }  Node\* temp = head;  do {  printf("%d -> ", temp->data);  temp = temp->next;  } while (temp != head); // Since it's circular, it will end where it started  printf("(head node data: %d)\n", head->data); // Show the circular nature by ending with the head node's data  }  int main() {  Node\* head = NULL; // Start with an empty list  // Adding nodes at the end  insertAtEnd(&head, 10); // List becomes: 10  insertAtEnd(&head, 20); // List becomes: 10 -> 20  insertAtEnd(&head, 30); // List becomes: 10 -> 20 -> 30  // Print the circular linked list  printf("Circular Linked List: ");  printList(head);  // Note: Remember to free the allocated memory to avoid memory leaks. This example doesn't include memory deallocation for simplicity.  return 0;  }  OUTPUT     1. **Delete a node from the end of singly circular linked list using c program**   **Code**  #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int data;  struct Node\* next;  } Node;  // Function to create a new node  Node\* createNode(int data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (!newNode) {  printf("Memory allocation failed.\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to add a node at the end of the list  void insertAtEnd(Node\*\* head, int data) {  Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head;  } else {  Node\* temp = \*head;  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode;  newNode->next = \*head;  }  }  // Function to delete the last node of the list  void deleteAtEnd(Node\*\* head) {  if (\*head == NULL) {  printf("List is empty.\n");  return;  }  Node\* temp = \*head;  Node\* prev = NULL;  // If the list contains only one node  if (temp->next == \*head) {  free(temp);  \*head = NULL;  return;  }  // Traverse the list to find the second-last node  while (temp->next != \*head) {  prev = temp;  temp = temp->next;  }  prev->next = temp->next; // Or simply prev->next = \*head;  free(temp);  }  // Function to print the list  void printList(Node\* head) {  if (head == NULL) {  printf("The list is empty.\n");  return;  }  Node\* temp = head;  do {  printf("%d -> ", temp->data);  temp = temp->next;  } while (temp != head);  printf("(back to head)\n");  }  int main() {  Node\* head = NULL;  // Adding nodes at the end  insertAtEnd(&head, 10);  insertAtEnd(&head, 20);  insertAtEnd(&head, 30);  printf("Initial List: ");  printList(head);  // Deleting the last node  deleteAtEnd(&head);  printf("After Deleting the Last Node: ");  printList(head);  // Free the remaining elements in the list to avoid memory leaks  return 0;  }  OUTPUT   |  | | --- | | **19, STACK DATA STRUCUTER IMPLEMENTION** |  1. **Basic operation of stack push and poping the element using c programming**   Code  #include <stdio.h>  #include <stdlib.h>  #define MAX 10 // Maximum size of the Stack  typedef struct Stack {  int arr[MAX]; // Array to store stack elements  int top; // Top represents the index of the topmost element  } Stack;  // Function to initialize the stack  void initStack(Stack\* s) {  s->top = -1; // -1 indicates that the stack is empty  }  // Function to check if the stack is full  int isFull(Stack\* s) {  return s->top == MAX - 1;  }  // Function to check if the stack is empty  int isEmpty(Stack\* s) {  return s->top == -1;  }  // Function to add an element to the stack  void push(Stack\* s, int data) {  if (isFull(s)) {  printf("Stack is full. Cannot push %d.\n", data);  return;  }  s->arr[++s->top] = data; // Increment top and add data  printf("%d pushed to stack.\n", data);  }  // Function to remove and return the top element of the stack  int pop(Stack\* s) {  if (isEmpty(s)) {  printf("Stack is empty. Cannot pop.\n");  return -1; // Return -1 or appropriate error code  }  return s->arr[s->top--]; // Return top element and decrement top  }  // Function to return the top element of the stack without removing it  int peek(Stack\* s) {  if (isEmpty(s)) {  printf("Stack is empty. Cannot peek.\n");  return -1; // Return -1 or appropriate error code  }  return s->arr[s->top];  }  // Main function  int main() {  Stack s;  initStack(&s);  push(&s, 10);  push(&s, 20);  push(&s, 30);  printf("Top element is %d\n", peek(&s));  printf("Popped %d from stack\n", pop(&s));  printf("Now, the top element is %d\n", peek(&s));  // Checking if stack is empty  if (isEmpty(&s)) {  printf("Stack is empty\n");  } else {  printf("Stack is not empty\n");  }  return 0;  }  OUTPUT     1. **Write a modular C program to check an infix expression has balanced brackets. Balanced brackets represents that an infix expression should have an appropriate opening and closing bracket.**   An open '(' bracket should have an appropriate close ')' in its place.  Similarly, an open '{' bracket should have an appropriate close '}' in its place.  Also an open '[' bracket should have an appropriate close ']' in its place.\*/  #include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #define SIZE 50  struct stack  {  char ex[SIZE];  int top;  };  void push(struct stack \*sptr,char expr)  {  if(sptr->top==SIZE-1)  printf("Stack overflow.\n");  else  {  sptr->top++;  sptr->ex[sptr->top]=expr;  }  }  char pop(struct stack \*sptr)  {  char ch;  if(sptr->top==-1)  return -1;  else  {  ch=sptr->ex[sptr->top];  sptr->top--;  return ch;  }  }  int brackets(struct stack \*sptr,char expr[50])  {  int i=0;  char ch;  while(expr[i]!='\0')  {  if(expr[i]=='('||expr[i]=='{'||expr[i]=='[')  {  push(sptr,expr[i]);  }  else if(expr[i]==')')  {  if(sptr->top==-1)  return 0;  else  {  ch=pop(sptr);  if(ch!='(')  return 0;  }  }  else if(expr[i]=='}')  {  if(sptr->top==-1)  return 0;  else  {  ch=pop(sptr);  if(ch!='{')  return 0;  }  }  else if(expr[i]==']')  {  if(sptr->top==-1)  return 0;  else  {  ch=pop(sptr);  if(ch!='[')  return 0;  }  }  i++;  }  if(sptr->top==-1)  return 1;  else  return 0;  }  int main() {  struct stack s,\*sptr;  sptr=&s;  sptr->top=-1;  char e[SIZE];  scanf("%s",e);  int v=brackets(sptr,e);  if(v==1)  printf("Valid expression.");  else  printf("Invalid expression.");  return 0;  }  OUTPUT   1. **Write a modular c program to reverse the given string using stack data structurs**   Code    #include <stdio.h>  #include <string.h>  #define max 100  int top,stack[max];  void push(char x){  // Push(Inserting Element in stack) operation  if(top == max-1){  printf("stack overflow");  } else {  stack[++top]=x;  }  }  void pop(){  // Pop (Removing element from stack)  printf("%c",stack[top--]);  }  main()  { printf("Given string is :sri lanka\n");  char str[]="sri lanka";  int len = strlen(str);  int i;  for(i=0;i<len;i++)  push(str[i]);  for(i=0;i<len;i++)  pop();  }  OUTPUT   1. Write a c program to swap the elements using stack data structer   Code  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  // Global stack arrays and top variable  int stack[MAX\_SIZE]; // Original stack  int sorted\_Stack[MAX\_SIZE]; // Stack used for sorting  int top = -1; // Top of the original stack  // Function to push an element onto the original stack  void push(int data) {  if (top == MAX\_SIZE - 1) {  printf("Overflow stack!\n");  return;  }  top++;  stack[top] = data;  }  // Function to pop an element from the original stack  int pop() {  if (top == -1) {  printf("Empty Stack!\n");  return -1;  }  int data = stack[top];  top--;  return data;  }  // Function to sort the original stack  void sort\_stack() {  int temp;  int sortedTop = -1; // Top of the stack used for sorting  // Sorting logic  while (top != -1) {  temp = pop();  while (sortedTop != -1 && sorted\_Stack[sortedTop] < temp) {  push(sorted\_Stack[sortedTop]);  sortedTop--;  }  sortedTop++;  sorted\_Stack[sortedTop] = temp;  }  // Pushing sorted elements back to the original stack  while (sortedTop != -1) {  push(sorted\_Stack[sortedTop]);  sortedTop--;  }  }  // Main function  int main() { int i;  // Push elements onto the original stack  push(1);  push(5);  push(5);  push(2);  push(3);  push(8);  // Print original stack elements  printf("Original stack: ");  for ( i = 0; i <= top; i++) {  printf("%d ", stack[i]);  }  printf("\n");  // Sort the original stack  sort\_stack();  // Print sorted stack elements  printf("Sorted stack: ");  for ( i = 0; i <= top; i++) {  printf("%d ", stack[i]);  }  printf("\n");  return 0;  }  OUTPUT   1. **Write a C program to find the next greater element for each element in an array using a stack. Return -1 if there is no next-larger element.**   **Code**  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  // Global stack and top variables  int stack[MAX\_SIZE];  int top = -1;  // Function to push an element onto the stack  void push(int data) {  if (top == MAX\_SIZE - 1) {  printf("Overflow stack!\n");  return;  }  top++;  stack[top] = data;  }  // Function to pop an element from the stack  int pop() {  if (top == -1) {  printf("Empty stack!\n");  return -1;  }  int data = stack[top];  top--;  return data;  }  // Function to print the next greater elements  void print\_next\_greater\_element(int arr[], int n) {  int i, next, element;  // Push the first element onto the stack  push(arr[0]);  // Iterate through the array  for (i = 1; i < n; i++) {  next = arr[i];  if (top != -1) {  element = pop();  // Find the next greater element  while (element < next) {  printf("%d --> %d\n", element, next);  if (top == -1) {  break;  }  element = pop();  }  if (element > next) {  push(element);  }  }  // Push the current element onto the stack  push(next);  }  // Remaining elements in the stack have no greater element  while (top != -1) {  element = pop();  next = -1;  printf("%d --> %d\n", element, next);  }  }  int main() {  int n=6 ;  int i=0 ;  int arr[6] = {1, 2, 3, 4, 5, 6};  printf("Elements in the array are: ");  for (i = 0; i < 6; i++) {  printf("%d ", arr[i]);  }  printf("\nThe next larger elements are: \n");  print\_next\_greater\_element(arr, n);  // More arrays to test  // ...  return 0;  }  OUTPUT   1. **Write a C program to find the minimum element in a stack.**   **CODE**  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define MAX\_SIZE 100  // Arrays to maintain the main stack and the stack for tracking minimum elements  int mainStack[MAX\_SIZE];  int minStack[MAX\_SIZE];  int top = -1; // Top index of the main stack  int min\_Top = -1; // Top index of the minimum stack  // Function to push an element onto the main stack  void push(int element) {  if (top >= MAX\_SIZE - 1) {  printf("Stack is full\n");  return;  }  // Push the element onto the main stack  top++;  mainStack[top] = element;  // If the minStack is empty or the element is less than or equal to the top element in minStack,  // push the element onto the minStack to track minimum elements  if (min\_Top == -1 || element <= minStack[min\_Top]) {  min\_Top++;  minStack[min\_Top] = element;  }  }  // Function to pop an element from the main stack  int pop() {  if (top < 0) {  printf("Stack is empty\n");  return INT\_MIN;  }  // Pop the top element from the main stack  int element = mainStack[top];  top--;  // If the popped element is the top element in minStack, also pop it from minStack  if (element == minStack[min\_Top]) {  min\_Top--;  }  return element;  }  // Function to get the minimum element from the main stack  int getMin() {  if (min\_Top < 0) {  printf("Stack is empty\n");  return INT\_MIN;  }  return minStack[min\_Top];  }  // Function to print the elements of the stack  void printstack(int \*stack) { int i;  printf("Current stack elements:\n");  for ( i = 0; i <= top; i++) {  printf("%d ", stack[i]);  }  }  int main() {  // Example usage of the stack functions  push(9);  push(2);  push(4);  push(2);  push(4);  printstack(mainStack);  printf("\nMinimum element: %d\n", getMin());  pop();  pop();  printf("\nAfter removing two elements:\n");  printstack(mainStack);  printf("\nMinimum element: %d\n", getMin());  push(1);  printf("\nAfter adding one element:\n");  printstack(mainStack);  printf("\nMinimum element: %d\n", getMin());  return 0;  }  OUTPUT     |  | | --- | | **QUEUE DATA STRUCTUER IMPLEMENTION** |  1. **Write a C program to count the number of elements in a queue.**   **Code**  #include <stdio.h>  #define MAX\_SIZE 100  int queue[MAX\_SIZE];  int front = -1; // Initialize front of the queue  int back = -1; // Initialize back of the queue  // Function to insert an element into the queue  void enqueue(int item) {  if (back == MAX\_SIZE - 1) { // Check if the queue is full  printf("Error: Queue is full\n");  return;  }  if (front == -1) {  front = 0; //  }  back++; // Increment the rear pointer  queue[back] = item; // Insert the item into the queue  }  // Function to display the elements in the queue  void display() { int i;  if (front == -1 || front > back) { ;//  printf("Queue is empty\n"); // Print message if the queue is empty  return;  }  printf("Queue elements are: "); /  for ( i = front; i <= back; i++) {  printf("%d ", queue[i]); // Print each element  }  printf("\n");  }  void dequeue() {  if (front == -1 || front > back) {  printf("Error: Queue is empty\n");  return;  }  front++;  }  // Function to check whether the queue is empty  int is\_empty() {  if (front == -1 || front > back) { // Check if the queue is empty  return 1; // Return 1 if the queue is empty  }  return 0; // Otherwise, return 0  }  // Function to count the number of elements in the queue  int count() { int i;  int count = 0;  if (front != -1 && back != -1) {  for (i = front; i <= back; i++) {  count++;  }  }  return count;  }  int main() {  printf("Initialize a queue!"); // Print message to indicate initializing a queue  // Insert some elements into the queue.  printf("\nCheck the queue is empty or not? %s\n", is\_empty() ? "Yes" : "No"); //  printf("Number of elements in queue: %d\n", count());  printf("\nInsert some elements into the queue:\n");  enqueue(1); //  enqueue(2); //  enqueue(3); //  display(); //  printf("Number of elements in queue: %d\n", count());  printf("\nDelete two elements from the said queue:\n");  dequeue();  dequeue();  display(); // Display the updated elements of the queue  printf("Number of elements in queue: %d\n", count());  printf("\nInsert another element into the queue:\n");  enqueue(4);  display();  printf("Number of elements in the queue: %d\n", count());  return 0; // Return from the main function  }  OUTPUT   1. **Write a C program to calculate the sum of the elements in a queue**   **Code**  #include <stdio.h>  #define MAX\_SIZE 100  // Define a structure for the queue  typedef struct {  int items[MAX\_SIZE];  int front; // Index of the front element  int rear; // Index of the rear element  } Queue;  // Function to create an empty queue  Queue\* createQueue() {  Queue\* q = (Queue\*)malloc(sizeof(Queue)); // Allocate memory for the queue  q->front = -1; // Initialize front to -1  q->rear = -1; // Initialize rear to -1  return q; // Return the created queue  }  // Function to check if the queue is empty  int isEmpty(Queue\* q) {  return q->rear == -1; // Return 1 if rear is -1, indicating an empty queue  }  // Function to check if the queue is full  int isFull(Queue\* q) {  return q->rear == MAX\_SIZE - 1; // Return 1 if rear is at the maximum size, indicating a full queue  }  // Function to add an element to the queue  void enqueue(Queue\* q, int value) {  if (isFull(q)) { // Check if the queue is full  printf("Queue is full\n"); // Print error message if the queue is full  } else {  if (q->front == -1) { // Check if front is -1 (indicating an empty queue)  q->front = 0; // Set front to 0  }  q->rear++; // Increment rear  q->items[q->rear] = value; // Add the element to the rear of the queue  }  }  // Function to remove an element from the queue  int dequeue(Queue\* q) {  int item;  if (isEmpty(q)) { // Check if the queue is empty  printf("Queue is empty\n"); // Print error message if the queue is empty  item = -1; // Return -1 as an indication of an empty queue  } else {  item = q->items[q->front]; // Get the element at the front of the queue  q->front++; // Increment front  if (q->front > q->rear) { // Check if front has crossed rear (indicating an empty queue)  q->front = q->rear = -1; // Reset front and rear to -1  }  }  return item; // Return the removed element  }  // Function to reverse the elements in a queue  void reverse\_queue(Queue\* q) {  int temp[MAX\_SIZE]; // Temporary array to store elements  int top = -1; // Initialize top of the temporary array  // Dequeue elements from the queue and store them in the temporary array  while (!isEmpty(q)) {  temp[++top] = dequeue(q);  }  // Enqueue elements from the temporary array back to the queue, effectively reversing the order  while (top != -1) {  enqueue(q, temp[top--]);  }  }  // Function to display the elements in the queue  void display(Queue\* q) { int i;  if (isEmpty(q)) { // Check if the queue is empty  printf("Queue is empty\n"); // Print message if the queue is empty  } else {  // Loop through the elements of the queue and print each element  for ( i = q->front; i <= q->rear; i++) {  printf("%d ", q->items[i]); // Print the element at index i  }  printf("\n"); // Print a newline after displaying all elements  }  }  // Main function to test the program  int main() {  Queue\* q = createQueue(); // Create an empty queue  // Add elements to the queue  enqueue(q, 1);  enqueue(q, 2);  enqueue(q, 3);  enqueue(q, 4);  enqueue(q, 5);  // Display the elements in the queue  printf("Queue elements are:\n");  display(q);  // Reverse the elements in the queue  printf("Reverse Queue, elements are:\n");  reverse\_queue(q);  // Display the elements in the reversed queue  display(q);  // Add two elements to the queue  enqueue(q, 100);  enqueue(q, 200);  printf("Add two elements to the said queue:\n");  printf("Queue elements are:\n");  display(q);  // Reverse the elements in the queue  printf("Reverse Queue, elements are:\n");  reverse\_queue(q);  display(q);  return 0; // Return from the main function  }  OUTPUT   1. **Write a C program to sort the elements of a queue in ascending order**   **Code**  #include <stdio.h>  #define MAX\_SIZE 100 /  int queue[MAX\_SIZE];  int front = -1, back = -1;  // Function to insert an element into the queue  void enqueue(int item) {  if (back == MAX\_SIZE - 1) { // Check if the queue is full  printf("Error: Queue is full\n");  return;  }  if (front == -1) { // Check if the queue is empty  front = 0; // If empty, set front to 0  }  back++; //  queue[back] = item  }  int dequeue() {  if (front == -1 || front > back) {  printf("Error: Queue is empty\n");  return -1; // Return -1 to indicate an empty queue  }  int item = queue[front];  front++;  return item;  }  // Function to display the elements in the queue  void display() { int i;  if (front == -1) {  printf("Error: Queue is empty\n"); /  return;  }  for ( i = front; i <= back; i++) {  printf("%d ", queue[i]); //  }  printf("\n"); /  }  // Function to sort the queue in ascending order  void sort\_queue\_asc() {  int i, j, temp;  int n = back - front + 1; // Calculate the number of elements in the queue  for (i = 0; i < n - 1; i++) {  for (j = i + 1; j < n; j++) {  if (queue[i] > queue[j]) {  temp = queue[i];  queue[i] = queue[j];  queue[j] = temp;  }  }  }  }  int main() {  // Insert elements into the queue  printf("Input some elements into the queue:");  enqueue(4);  enqueue(2);  enqueue(7);  enqueue(5);  enqueue(1);    printf("\nElements of the queue:\n");  display();  printf("\nSort the said queue:");  sort\_queue\_asc();  printf("\nElements of the sorted queue in ascending order:\n");  display();  printf("\nInput two more elements into the queue:");  enqueue(-1);  enqueue(3);  printf("\nElements of the queue:\n");  display();  printf("\nSort the said queue:");  sort\_queue\_asc();  printf("\nElements of the sorted queue in ascending order:\n");  display();  return 0;  }  OUTPUT |
| 1. Write a code to implement circular queue using c programm   Code  #include <stdio.h>  #include <stdlib.h>  #define SIZE 5  // Define a structure for the circular queue  typedef struct {  int items[SIZE];  int front, rear;  } CircularQueue;  // Function to initialize the queue  void initializeQueue(CircularQueue \*q) {  q->front = -1;  q->rear = -1;  }  // Check if the queue is full  int isFull(CircularQueue \*q) {  if ((q->front == q->rear + 1) || (q->front == 0 && q->rear == SIZE - 1)) return 1;  return 0;  }  // Check if the queue is empty  int isEmpty(CircularQueue \*q) {  if (q->front == -1) return 1;  return 0;  }  // Adding an element to the queue  void enqueue(CircularQueue \*q, int element) {  if (isFull(q))  printf("\nQueue is full!\n");  else {  if (q->front == -1) q->front = 0;  q->rear = (q->rear + 1) % SIZE;  q->items[q->rear] = element;  printf("\nInserted -> %d\n", element);  }  }  // Removing an element from the queue  int dequeue(CircularQueue \*q) {  int element;  if (isEmpty(q)) {  printf("\nQueue is empty!\n");  return (-1);  } else {  element = q->items[q->front];  if (q->front == q->rear) {  // Queue has only one element, so we reset the queue after deleting it.  q->front = -1;  q->rear = -1;  } else {  q->front = (q->front + 1) % SIZE;  }  printf("\nDeleted element -> %d\n", element);  return (element);  }  }  // Function to display the queue  void display(CircularQueue \*q) {  int i;  if (isEmpty(q))  printf(" \nEmpty Queue\n");  else {  printf("\nFront -> %d ", q->front);  printf("\nItems -> ");  for (i = q->front; i != q->rear; i = (i + 1) % SIZE) {  printf("%d ", q->items[i]);  }  printf("%d ", q->items[i]);  printf("\nRear -> %d \n", q->rear);  }  }  int main() {  CircularQueue q;  // Initialize the queue  initializeQueue(&q);  // Demonstrating enqueue operations  enqueue(&q, 1);  enqueue(&q, 2);  enqueue(&q, 3);  enqueue(&q, 4);  enqueue(&q, 5);  // Queue is full at this point  enqueue(&q, 6);  display(&q);  int elem = dequeue(&q); // Remove an element  if (elem != -1)  printf("\nDequeued value: %d\n", elem);  display(&q);  enqueue(&q, 7);  display(&q);  return 0;  }  OUTPUT     1. Implement priority queue using c programming   Code  // Priority Queue implementation in C  #include <stdio.h>  int size = 0;  void swap(int \*a, int \*b) {  int temp = \*b;  \*b = \*a;  \*a = temp;  }  // Function to heapify the tree  void heapify(int array[], int size, int i) {  if (size == 1) {  printf("Single element in the heap");  } else {  // Find the largest among root, left child and right child  int largest = i;  int l = 2 \* i + 1;  int r = 2 \* i + 2;  if (l < size && array[l] > array[largest])  largest = l;  if (r < size && array[r] > array[largest])  largest = r;  // Swap and continue heapifying if root is not largest  if (largest != i) {  swap(&array[i], &array[largest]);  heapify(array, size, largest);  }  }  }  // Function to insert an element into the tree  void insert(int array[], int newNum) { int i;  if (size == 0) {  array[0] = newNum;  size += 1;  } else {  array[size] = newNum;  size += 1;  for ( i = size / 2 - 1; i >= 0; i--) {  heapify(array, size, i);  }  }  }  // Function to delete an element from the tree  void deleteRoot(int array[], int num) {  int i;  for (i = 0; i < size; i++) {  if (num == array[i])  break;  }  swap(&array[i], &array[size - 1]);  size -= 1;  for ( i = size / 2 - 1; i >= 0; i--) {  heapify(array, size, i);  }  }  // Print the array  void printArray(int array[], int size) { int i;  for (i = 0; i < size; ++i)  printf("%d ", array[i]);  printf("\n");  }  // Driver code  int main() {  int array[10];  insert(array, 3);  insert(array, 4);  insert(array, 9);  insert(array, 5);  insert(array, 2);  printf("Max-Heap array: ");  printArray(array, size);  deleteRoot(array, 4);  printf("After deleting an element: ");  printArray(array, size);  }  OUTPUT |