|  |  |
| --- | --- |
| **Name of the student** | **ANANYA HAMPIMATH** |
| **Roll No** | **141** |
| **USN** | **01FE22BEI047** |
| **Div** | **F** |

PART A

**1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate.**

ANS: A variable is a named storage location in the computer's memory where data can be stored and manipulated during program execution. Each variable has a specific data type, which determines the kind of data that can be stored in it and the operations that can be performed on that data.

A data type, on the other hand, is a classification that specifies which type of value a variable can hold. It defines the operations that can be done on the data, the meaning of the data, and the way values of that type can be stored.

Ex: int age;

age=25;

float height;

Char grade;

**age**, **height**, and **grade** are variables

**Int**, **float**, and **char** are data types specifying the kind of data that the variables can hold.Top of Form

**2. Explain the concept of data types in C programming. Discuss the different types of data types available in C.**

ANS: Data types are a fundamental concept that define the type of data that can be stored in a variable, the operations that can be performed on the data, and the amount of memory required to store the data. C provides several built-in data types, each designed to handle different kinds of values and to suit various programming needs.

**int**: Represents integer values.

**float**: Represents floating-point values.

**double**: Represents double-precision floating-point values.

**char**: Represents individual characters.

Arrays: Collections of elements of the same data type.

Pointers: Variables that store memory addresses of other variables.

Structures: User-defined data types that group together variables of different data types under one name.

**3.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 by specifying the data type followed by the variable name. Optionally, variables can be initialized at the time of declaration by assigning an initial value. Here's how variables are declared and initialized:

Declaration without initialization:

Int age;

Float height;

Char grade;

Declaration with initialization:

Int age=25;

Float pi=3.14;

Char letter=’A’;

Variables **age**, **height**, and **grade** are declared without initialization.

Variables **score**, **pi**, and **letter** are declared with initialization, where they are assigned initial values of 25 3.14, and 'A' respectively.

**4. 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 they can be accessed within a program. Scope refers to the region of code where a variable is visible and can be accessed. In C, variables can have either global or local scope.

Global variables are declared outside of any function, typically at the beginning of a program, and can be accessed from any part of the program. They exist throughout the entire execution of the program and retain their value until the program terminates. Global variables can be accessed by any function in the program.

Local variables are declared within a function or block and are only accessible within that function or block. They are created when the function is called and destroyed when the function exits. Local variables cannot be accessed from outside the function in which they are declared. Lifetime refers to the duration for which a variable exists in memory during program execution

**5. 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. Type casting is necessary when there is a need to convert data from one type to another, typically to ensure compatibility in operations or assignments. It's performed by specifying the desired data type in parentheses before the value or variable to be converted. Type casting helps in ensuring proper behavior in operations involving variables of different data types.

**6. 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 concise way to express conditional statements in C programming. It allows you to evaluate a condition and choose between two possible expressions based on the result of the condition.

It's essentially a shorthand for an if-else statement. The ternary operator is often used in situations where a simple decision needs to be made based on a condition, and using an if-else statement would be overly verbose.

Ex: #include <stdio.h>

int main() {

int num = 10;

printf("%s\n", (num % 2 == 0) ? "Even" : "Odd");

return 0;

}

if **num** is even, the ternary operator evaluates to **"Even"**, and if **num** is odd, it evaluates to **"Odd"**. The chosen string is then printed using **printf**.

**7. Discuss the bitwise operators available in C programming. Explain their usage with suitable examples.**

ANS: Bitwise operators are used to perform operations at the bit level. They allow manipulation of individual bits within integer values. The bitwise operators available in C are:

1. **Bitwise AND (&)**: Performs a bitwise AND operation on corresponding bits of two operands.
2. **Bitwise OR (|)**: Performs a bitwise OR operation on corresponding bits of two operands.
3. **Bitwise XOR (^)**: Performs a bitwise XOR (exclusive OR) operation on corresponding bits of two operands.
4. **Bitwise NOT (~)**: Flips all the bits of the operand, turning 0s into 1s and 1s into 0s.
5. **Left Shift (<<)**: Shifts the bits of the left operand to the left by a specified number of positions.
6. **Right Shift (>>)**: Shifts the bits of the left operand to the right by a specified number of positions.

These operators are handy for tasks such as setting or clearing specific bits, implementing algorithms for data compression, or optimizing certain operations for performance. For instance, bitwise AND (**&**) can be used to mask specific bits, while bitwise XOR (**^**) can toggle bits.

**8. Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate.**

ANS:Both postfix and prefix increment operators (**++**) are used to increment the value of a variable by 1. However, they have a key difference in how they behave:

**Prefix Increment (++variable)**: In prefix increment, the value of the variable is first incremented, and then the updated value is used in the expression where the operator appears.

**Postfix Increment (variable++)**: In postfix increment, the current value of the variable is used in the expression where the operator appears, and then the value of the variable is incremented.

#include <stdio.h>

int main() {

int num = 5;

// Prefix increment

printf("Prefix Increment: %d\n", ++num); // Output: 6 (num is incremented first, then used)

printf("Value after Prefix Increment: %d\n", num); // Output: 6

num = 5; // Reset num

// Postfix increment

printf("Postfix Increment: %d\n", num++); // Output: 5 (num is used first, then incremented)

printf("Value after Postfix Increment: %d\n", num); // Output: 6

return 0;

}

**9. 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 combine multiple conditions in conditional expressions.

**Logical AND (&&)**: Returns true if both operands are true.

**Logical OR (||)**: Returns true if at least one operand is true.

They are significant for constructing complex conditional statements and controlling program flow based on multiple conditions.

**10.Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation.**

ANS: **Operator Precedence**: It defines the priority of operators in an expression. Operators with higher precedence are evaluated before operators with lower precedence. For example, in the expression **2 + 3 \* 4**, multiplication (**\***) has higher precedence than addition (**+**), so **3 \* 4** is evaluated first, resulting in **12**, and then added to **2** to give **14**.

**Associativity**: It specifies the order in which operators of the same precedence level are evaluated. It can be either left-to-right or right-to-left. For example, the addition (**+**) operator is left-associative, so in the expression **2 + 3 + 4**, the leftmost **+** is evaluated first, resulting in **5**, and then added to **4** to give **9**.

**11. Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement?**

ANS: In C programming, the switch statement is used to select one of many blocks of code to execute based on the value of an expression. It provides a more concise and efficient way to handle multiple conditions compared to using multiple if-else statements. The switch statement evaluates the expression once and compares it against multiple constant values (cases), executing the code block associated with the first matching case. In contrast, the if-else statement evaluates each condition sequentially until a true condition is found, executing the corresponding code block and skipping the remaining conditions. Therefore, switch is preferred when there are multiple cases to be evaluated against a single expression, while if-else is suitable for handling multiple independent conditions.

**12. Explain the concept of nested control structures in C programming. Provide an example demonstrating nested if-else statements.**

ANS: Nested control structures in C programming refer to the situation where one control structure is placed inside another control structure. This nesting allows for more complex decision-making and looping within a program.

EXAMPLE:

#include <stdio.h>

int main() {

int num = 15;

if (num > 0) {

if (num % 2 == 0) {

printf("The number is positive and even.\n");

} else {

printf("The number is positive and odd.\n");

}

} else {

printf("The number is not positive.\n");

}

return 0;

}

**13. Discuss the role of the break and continue statements in loop control in C programming. Provide examples to illustrate their usage.**

ANS: The **break** statement is used to immediately terminate the execution of a loop (for, while, or do-while) and transfer control to the statement following the loop.

EX: #include <stdio.h>

int main() {

for (int i = 1; i <= 10; i++) {

if (i == 5) {

break; // Exit the loop when i becomes 5

}

printf("%d ", i);

}

printf("\n");

return 0;

}

The **continue** statement is used to skip the remaining code within a loop for the current iteration and proceed to the next iteration of the loop. It is typically used to avoid executing certain statements for specific conditions.

Ex: for (int i = 1; i <= 10; i++) {

if (i % 2 == 0) {

continue;

}

printf("%d ", i);

}

**14. What are the advantages of using the for loop over the while loop in C programming? Provide examples comparing the two.**

ANS: The for loop and while loop are both used for repetitive execution of code. The choice between them depends on the specific requirements of the program. However, the for loop often offers advantages over the while loop in certain situations:

**Initialization, condition, and increment in one line**: The for loop allows the initialization, condition check, and increment/decrement to be written in a single line, making the loop more concise and easier to understand.

// Using for loop

for (int i = 0; i < 5; i++) {

printf("%d ", i);

}

// Using while loop

int i = 0;

while (i < 5) {

printf("%d ", i);

i++;

}

**15. Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements?**

ANS: short-circuit evaluation is a mechanism where logical expressions are evaluated from left to right, and if the outcome can be determined by evaluating only a part of the expression, the remaining part is not evaluated. This behavior is exhibited by the logical AND (**&&**) and logical OR (**||**) operators. In if statements, short-circuit evaluation allows the program to optimize by avoiding unnecessary evaluations of expressions when the result can be determined based on the first part of the expression, improving efficiency.

**Top of Form**

**16. Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes?**

ANS: A function prototype declares the function's name, return type, and parameters before the function's actual implementation. It serves as a forward declaration, informing the compiler about the existence of the function, its return type, and the types of its parameters. Function prototypes are necessary because they allow functions to be called before their actual definition in the code, enabling modular programming and facilitating proper function usage throughout the program. Additionally, function prototypes help catch errors related to function calls with incorrect arguments or return types during compilation.

**17. Explain the difference between call by value and call by reference in C programming. Provide examples to illustrate both concepts.**

ANS: **Call by Value**: Copies the value of the actual parameter into the function parameter. Changes to the parameter inside the function do not affect the original value.

Ex: void changeValue(int num) {

num = num + 10;

}

**Call by Reference**: Passes the memory address of the actual parameter to the function parameter, allowing changes inside the function to directly affect the original value.

Ex: void changeValue(int \*num) {

\*num = \*num + 10;

}

18. Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works.

ANS: Recursion in C programming is a technique where a function calls itself directly or indirectly to solve a problem. This allows for elegant solutions to problems that can be broken down into smaller, similar sub-problems.

Ex: #include <stdio.h>

int factorial(int n) {

if (n == 0 || n == 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

int main() {

int num = 5;

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

return 0;

}

**19. What is the significance of the return statement in C programming? How are values returned from functions?**

ANS: In C programming, the **return** statement is used to exit a function and optionally return a value to the caller. It allows functions to provide a result or output that can be used by other parts of the program.

Values are returned from functions by using the **return** statement followed by the value to be returned.

For example:

int add(int a, int b) {

return a + b;

}

**20. Describe the role of function parameters and arguments in C programming. How are function arguments passed to parameters?**

ANS: Function parameters serve as placeholders within the function definition to receive values passed to the function during its invocation. Function arguments are the actual values passed to the function when it is called, which are then assigned to the function parameters.

Function arguments are passed to parameters in C programming by value by default. This means that the value of each argument is copied into the corresponding parameter within the function. Any modifications made to the parameters inside the function do not affect the original arguments outside the function.

**21. Explain the concept of arrays in C programming. How are arrays declared and initialized?**

ANS:An array is a collection of elements of the same data type, stored in contiguous memory locations. Arrays allow for the storage and manipulation of multiple values under a single name.

Arrays are declared and initialized in C by specifying the data type of the elements and the size of the array.EX:

int numbers[5]; // Declaration of an array named 'numbers' capable of holding 5 integers

int values[3] = {10, 20, 30}; // Declaration and initialization of an array named 'values' with 3 integers

**22. Discuss the difference between a one-dimensional array and a multidimensional array in C programming. Provide examples of both.**

ANS: **One-dimensional array**: A one-dimensional array is a linear collection of elements of the same data type, arranged sequentially in memory.

Ex; int arr[5]; // One-dimensional array of 5 integers

**Multidimensional array**: A multidimensional array is an array of arrays, where each element is itself an array. It creates a tabular or grid-like structure of elements.

Ex: int matrix[3][3]; // Multidimensional array representing a 3x3 matrix

**23. Describe the process of accessing array elements in C programming. How are array indices used to access elements**

ANS: In C programming, array elements are accessed using array indices. Each element in the array is associated with an index, which represents its position within the array.

To access an array element, specify the name of the array followed by the index of the element enclosed in square brackets.

int arr[5]; // Declaration of an array named 'arr' with 5 elements

arr[0] = 10; // Accessing and assigning value 10 to the first element of the array

**24. What is the significance of the null character ('\0') in C strings? How is it used to determine the end of a string?**

ANS: In C programming, the null character (**'\0'**) is used to denote the end of a string. It is a special character with ASCII value 0, and it is automatically appended to the end of every string literal in C.

The null character serves as a sentinel value that marks the end of the string when working with character arrays. Functions that operate on strings, such as **printf()**, **strlen()**, and **strcpy()**, use the null character to determine the length of the string and to know when the string ends.

**25. Explain the concept of dynamic memory allocation for arrays in C programming. How are dynamic arrays allocated and deallocated**

ANS: In C programming, dynamic memory allocation allows you to allocate memory for arrays at runtime, rather than at compile time. This is useful when you need to work with arrays whose size is determined during program execution.

Dynamic arrays are allocated using functions like **malloc()** or **calloc()**, which allocate memory from the heap. Memory allocated dynamically must be manually deallocated using the **free()** function to avoid memory leaks.

Example of dynamic memory allocation for an array:

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr;

int size = 5;

// Allocate memory for an array of 'size' integers

arr = (int \*)malloc(size \* sizeof(int));

// Use the dynamically allocated array

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

arr[i] = i;

}

// Deallocate memory

free(arr);

return 0;

}

**26. Describe the purpose and usage of pointers in C programming. How are pointers declared and initialized**

ANS: pointers are variables that store memory addresses. They are used to manipulate data indirectly by referencing the memory locations where the data is stored. Pointers are widely used for dynamic memory allocation, passing parameters to functions by reference, and implementing data structures like linked lists and trees.

Pointers are declared by specifying the data type they point to, followed by an asterisk (**\***). They are initialized by assigning the address of a variable using the address-of operator (**&**) or by assigning the result of memory allocation functions.

EX: int \*ptr; // Declaration of an integer pointer named 'ptr'

int x = 10;

ptr = &x; // Initialization of 'ptr' with the address of 'x'

**27. Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers**

ANS: In C programming, pointer arithmetic involves performing arithmetic operations on pointers, such as addition and subtraction. These operations manipulate the memory addresses stored in pointers based on the size of the data type they point to.

Ex: #include <stdio.h>

int main() {

int arr[] = {10, 20, 30, 40, 50};

int \*ptr = arr; // Pointer to the first element of the array

// Pointer addition

ptr = ptr + 2; // Move pointer two elements forward

printf("Value at ptr after addition: %d\n", \*ptr); // Output: 30

// Pointer subtraction

ptr = ptr - 1; // Move pointer one element backward

printf("Value at ptr after subtraction: %d\n", \*ptr); // Output: 20

return 0;

}

**28. Discuss the difference between pass by value and pass by reference in function arguments using pointers in C programming. Provide examples to illustrate both approaches.**

ANS: In pass by value, a copy of the argument's value is passed to the function parameter. Changes made to the parameter inside the function do not affect the original argument.

Ex; #include <stdio.h>

void increment(int num) {

num++;

}

int main() {

int x = 5;

increment(x);

printf("Value of x after increment: %d\n", x); // Output: 5 (unchanged)

return 0;

}

In pass by reference, the memory address of the argument is passed to the function parameter using pointers. Changes made to the parameter inside the function directly affect the original argument.

Ex: #include <stdio.h>

void increment(int \*num) {

(\*num)++;

}

int main() {

int x = 5;

increment(&x);

printf("Value of x after increment: %d\n", x); // Output: 6 (changed)

return 0;

}

**29. Describe the concept of NULL pointers in C programming. How are NULL pointers used and checked for in programs**

ANS: a NULL pointer is a pointer that does not point to any memory address. It is often used to indicate that a pointer is not currently pointing to a valid object or memory location.

NULL pointers are commonly used to initialize pointers before assigning them valid addresses or to check if a pointer has been properly initialized.

Example of using and checking NULL pointers:

#include <stdio.h>

int main() {

int \*ptr = NULL; // Initializing pointer to NULL

if (ptr == NULL) {

printf("Pointer is NULL\n");

} else {

printf("Pointer is not NULL\n");

}

return 0;

}

**30. Explain the role of pointers in dynamic memory allocation in C programming. How are pointers used to allocate and deallocate memory dynamically?**

ANS: Pointers play a crucial role in dynamic memory allocation. Dynamic memory allocation allows programs to allocate memory at runtime, as opposed to compile time. Pointers are used to manage dynamically allocated memory by storing the address of the allocated memory block.

To allocate memory dynamically, the **malloc()** or **calloc()** function is used, which returns a pointer to the allocated memory block. The pointer is then used to access and manipulate the dynamically allocated memory.

**31. Discuss the concept of strings in C programming. How are strings represented and manipulated in C?**

ANS: n C programming, strings are represented as arrays of characters, terminated by a null character (**'\0'**). Strings are manipulated using various library functions and standard syntax for array operations.

Example of string representation and manipulation:

#include <stdio.h>

#include <string.h>

int main() {

char str1[] = "Hello";

char str2[10];

// Copying string

strcpy(str2, str1);

// Concatenating strings

strcat(str2, " World");

// Printing strings

printf("str1: %s\n", str1);

printf("str2: %s\n", str2);

return 0;

}

In this example, **str1** and **str2** are string arrays. Strings can be copied using **strcpy()**, concatenated using **strcat()**, and manipulated using various other string manipulation functions provided by the C standard library.

**32. Explain the difference between character arrays and string literals in C programming. Provide examples to illustrate both concepts**

ANS: **Character Arrays**: Character arrays are sequences of characters stored in contiguous memory locations. They can be modified as they are mutable.

char array[] = "Hello"; // Character array declaration

**String Literals**: String literals are sequences of characters enclosed in double quotes. They are stored in read-only memory and cannot be modified directly.

char \*str = "Hello"; // String literal declaration

**33. Describe common string manipulation functions available in the C standard library. Provide examples of functions like strlen, strcpy, strcat, and strcmp.**

ANS: **strlen()**: This function is used to calculate the length of a string.

#include <stdio.h>

#include <string.h>

int main() {

char str[] = "Hello";

int length = strlen(str);

printf("Length of the string: %d\n", length);

return 0;

}

**strcpy()**: This function is used to copy one string to another.

#include <stdio.h>

#include <string.h>

int main() {

char source[] = "Hello";

char destination[10];

strcpy(destination, source);

printf("Copied string: %s\n", destination);

return 0;

}

**strcat()**: This function is used to concatenate (append) one string to another.

#include <stdio.h>

#include <string.h>

int main() {

char str1[20] = "Hello";

char str2[] = " World";

strcat(str1, str2);

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

return 0;

}

**strcmp()**: This function is used to compare two strings.

#include <stdio.h>

#include <string.h>

int main() {

char str1[] = "hello";

char str2[] = "world";

int result = strcmp(str1, str2);

if (result == 0) {

printf("Strings are equal\n");

} else {

printf("Strings are not equal\n");

}

return 0;

}

**34. Discuss the concept of string tokenization in C programming. How are strings split into tokens using delimiter characters?**

ANS:string tokenization is the process of splitting a string into smaller parts called tokens based on specified delimiter characters. The **strtok()** function from the C standard library is commonly used for this purpose

Here's how string tokenization works in a very short explanation:

1.First, include the **string.h** header file to use the **strtok()** function.

2.Use the **strtok()** function to tokenize the string. It takes two arguments: the string to be tokenized and a string containing delimiter characters.

3.Each time **strtok()** is called with the string and delimiter, it returns the next token from the string until no more tokens are left, or NULL is encountered.

**35. Explain the importance of null-terminated strings in C programming. How does the null character ('\0') signify the end of a string?**

ANS: In C programming, null-terminated strings are strings that end with a null character (**'\0'**). The null character is used to signify the end of the string.

The null-terminated string is a crucial concept in C because many standard library functions that operate on strings rely on the null character to determine the length of the string and to know when the string ends. Without the null character, these functions would not know where the string ends, leading to undefined behavior and potential memory access errors.

In short, the null character (**'\0'**) serves as a sentinel value that marks the end of the string, allowing C programs to accurately manipulate and process strings using standard library functions.

**36. Describe the purpose and usage of structures in C programming. How are structures declared and accessed?**

ANS: In C programming, structures (structs) are used to group different data types together under a single name for easier and organized management of data. They allow the representation of a "record" by combining different data types into one unit.

To declare a structure, use the **struct** keyword followed by a structure name and a block of members (variables) inside braces.

struct Person {

char name[50];

int age;

float salary;

};

Structures in C are foundational for organizing related data, making it easier to manage complex data models in programming projects.

**37. Discuss the concept of structure members in C programming. How are individual members of a structure accessed and modified?**

ANS: In C programming, structure members are accessed and modified using the dot (.) operator for structure variables and the arrow (->) operator for pointers to structures. The dot operator is used directly with the structure variable to access or modify its members. For example, **myStruct.member = value;** assigns a value to the member of a structure variable **myStruct**. When dealing with pointers to structures, the arrow operator is used instead to access or modify members. For example, **myStructPtr->member = value;** achieves the same but with a pointer **myStructPtr** pointing to the structure. These operators provide a convenient way to work with grouped data in structures, allowing individual members to be easily manipulated.

**38. Explain the difference between structures and unions in C programming. When would you choose one over the other?**

ANS: In C programming, structures (**struct**) and unions (**union**) both group different types of data under one name, but they store data differently. Structures allocate separate memory for each member, allowing simultaneous storage of multiple members. Unions, on the other hand, share the same memory space among all members, storing only one member at a time.

Choose a structure when you need to use all the members at once since each member has its own memory space. Opt for a union when you need a way to store one of several types of data in the same memory location at different times, thus saving space, as all members share the same memory address.

**39. Describe the concept of nested structures in C programming. How are structures within structures defined and accessed?**

ANS: In C programming, nested structures involve defining a structure within another structure. This allows for creating complex data models that better represent real-world data.

To define a nested structure, simply declare a structure type within another structure definition. Accessing members of a nested structure involves using the dot (.) operator successively to drill down through the layers.

struct Date {

int day;

int month;

int year;

};

struct Person {

char name[50];

struct Date birthday;

};

// Define and access

struct Person person;

person.birthday.day = 1;

person.birthday.month = 1;

person.birthday.year = 1990;

**40. Discuss the concept of typedef in C programming. How is typedef used to define custom data types, including structures and unions.**

ANS: In C programming, **typedef** is a keyword used to create aliases for existing data types, making them easier to understand and use, especially for complex data structures and unions. By using **typedef**, you can define custom names for data types, structures, and unions, enhancing code readability and maintainability.

typedef struct {

int x;

int y;

} Point;

typedef union {

int id;

char \*name;

} Identifier;

**41.Explain the concept of file handling in C programming. How are files opened, read from, and written to using standard file handling functions?**

ANS: File handling allows us to create, read, write, and manipulate files within our programs. It’s essential for tasks like storing data, reading configuration files, and managing persistent information. The key concepts and standard functions related to file handling:

File Operations:

Creating a New File: To create a new file, we use the fopen() function with attributes like "w" (write mode) or "w+" (read-write mode). For example:

o FILE\* file = fopen("my\_file.txt", "w");

o Opening an Existing File: To open an existing file, we use fopen() with attributes like "r" (read mode) or "r+" (read-write mode). For example:

o FILE\* file = fopen("existing\_file.txt", "r");

o Reading from a File:

 We can use fscanf() to read formatted data or fgets() to read lines of text from the file.

 Example using fscanf():

 int num;

 fscanf(file, "%d", &num);

o Writing to a File:

 We use fprintf() to write formatted data or fputs() to write strings to the file.

 Example using fprintf():

 fprintf(file, "Hello, world!\n");

**42. Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files?**

ANS: File Pointers:

 A file pointer is a crucial concept when working with files in C.

 It acts as a cursor that keeps track of the current position within a file.

 File pointers allow us to read, write, and seek to specific locations within the file.

o Navigating with File Pointers:

 When we open a file, a file pointer is automatically created and positioned at the beginning of the file.

 As we read or write data, the file pointer moves forward.

o Common File Pointer Operations:

 Reading Data:

 Use fscanf() or fgets() to read data from the file.

 The file pointer advances after each read operation.

 Writing Data:

 Use fprintf() or fputs() to write data to the file.

 The file pointer moves forward for writing.

**43. Discuss the difference between text files and binary files in C programming. How are they opened and processed differently?**

Text Files:

o Content:

 Text files store data in the form of alphabets, digits, and special symbols.

 They represent textual information and are human-readable.

 Examples include files with extensions like .txt, .c, etc.

o Storage Format:

 Text files store data by encoding characters using their ASCII values.

 Each character corresponds to a specific byte.

o Handling Newlines:

 Newline characters ('\n') are converted into a carriage return-linefeed combination when written to the disk.

 When read from a text file, this combination is converted back to a newline.

o Error Detection:

 Errors in textual files are often detectable and can be corrected.

Binary Files:

o Content:

 Binary files contain a sequence or collection of bytes.

 They are not human-readable.

 Examples include files with extensions like .exe, .mp3, etc.

o Storage Format:

 Binary files represent custom data structures.

 They do not follow a strict character-to-byte mapping.

o Handling Newlines:

 In binary files, no newline conversions occur.

 Data is stored exactly as provided.

o Error Detection:

 Errors in binary files can corrupt the file and are not easily detectable.

**44. Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc.**

File Modes: File modes determine how a file is opened and accessed. They specify whether we want to read, write, or append data to a file. Each mode serves a specific purpose.

o "r" (Read Mode):

 Opens an existing file for reading.

 If the file doesn’t exist, it returns NULL.

 Example:

 FILE\* file = fopen("my\_file.txt", "r");

o "w" (Write Mode):

 Creates a new file for writing.

 If the file already exists, it truncates its content.

 Example:

 FILE\* file = fopen("new\_file.txt", "w");

o "a" (Append Mode):

 Opens an existing file for appending data.

 If the file doesn’t exist, it creates a new one.

 Example:

FILE\* file = fopen("log.txt", "a");

**45. Describe error handling techniques in file operations in C programming. How are errors detected and handled when working with files?**

• If the file doesn’t exist or there’s an issue opening it, the pointer will be NULL.

• Check if the pointer is NULL to detect errors.

• When reading from a file, use feof() to check if you’ve reached the end of the file.

• When an error occurs, handle it gracefully.

• Close the file using fclose() to release resource.

• Use ferror() to check if an error occurred during read/write operations.

**Part-B- Programs to Ensure Learnings from Fundamentals of C Programming**

**1. Hello World: Print "Hello, World!" to the console.**

#include <stdio.h>

int main()

{ char str[100];

FILE\*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

// printf() displays the string inside quotation

while(!feof(input))

{

fscanf(input,"%s",str);

printf("%s ",str);

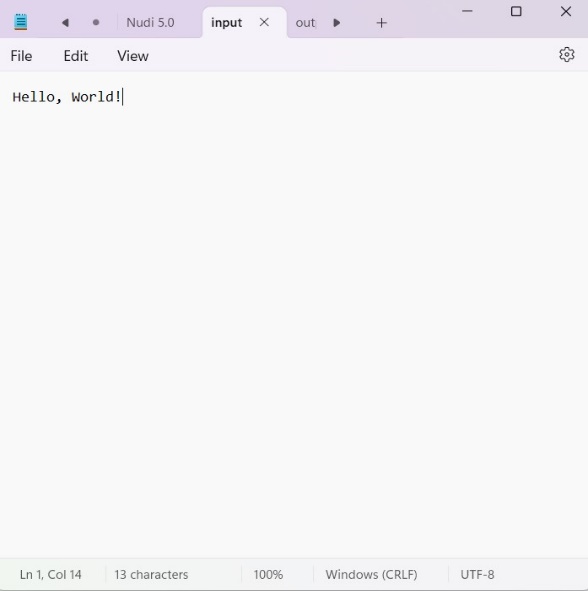
fprintf(output,"%s ",str);

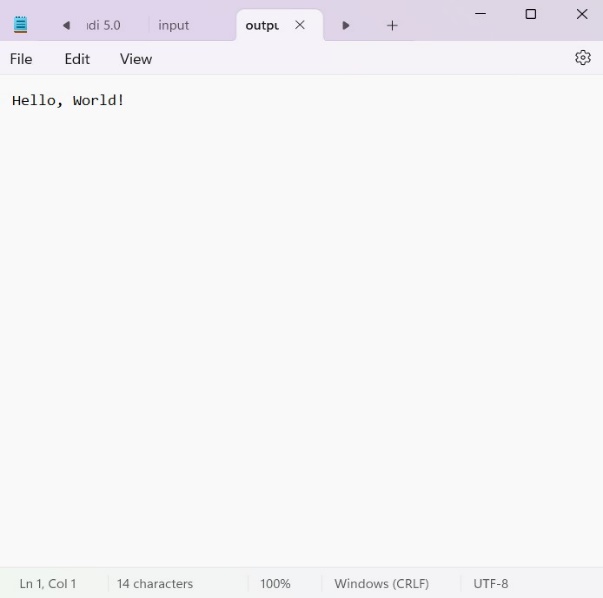
}

fclose(input);

fclose(output);

return 0;

}



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

#include <stdio.h>

int main()

{ int i=0,n,f=1;

FILE\*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

// printf() displays the string inside quotation

while(!feof(input))

{

fscanf(input,"%d",&n);

printf("%d ",n);

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

{

f=f\*i;

}

}

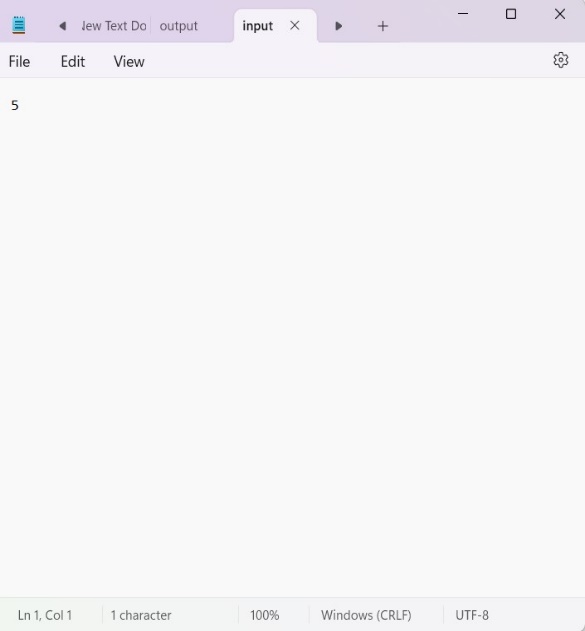
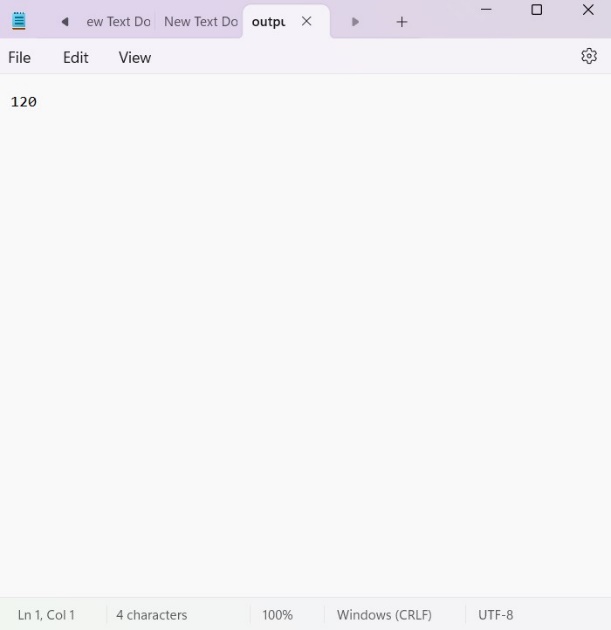
fprintf(output,"%d ",f);

fclose(input);

fclose(output);

return 0;

}

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

#include <stdio.h>

int main()

{

int n,i,count=0;

FILE \*input,\*output;

printf("Enter a positive integer: ");‍

scanf("%d",&n);

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

{

if (n%i==0)

{

count++;

}

}

if (count==2)

{

printf("%d is a prime number.", n);

}

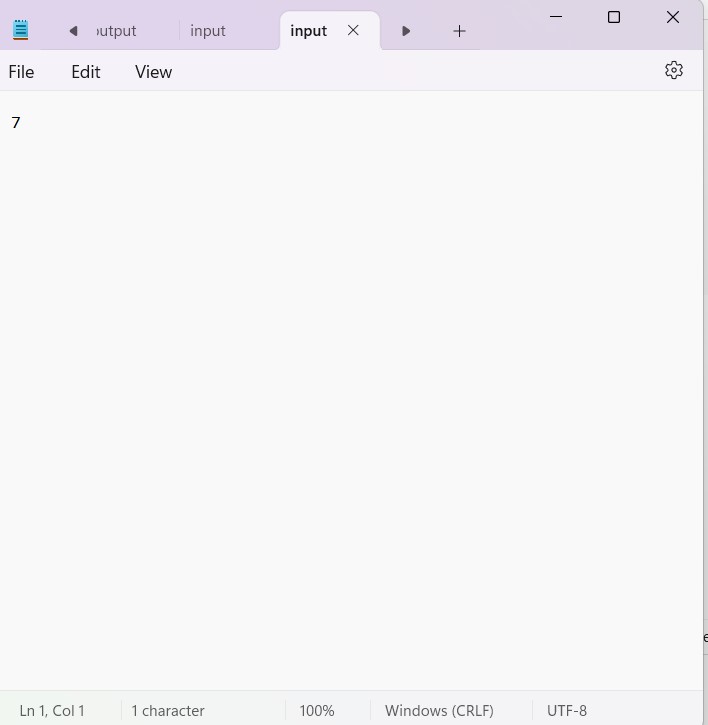
else

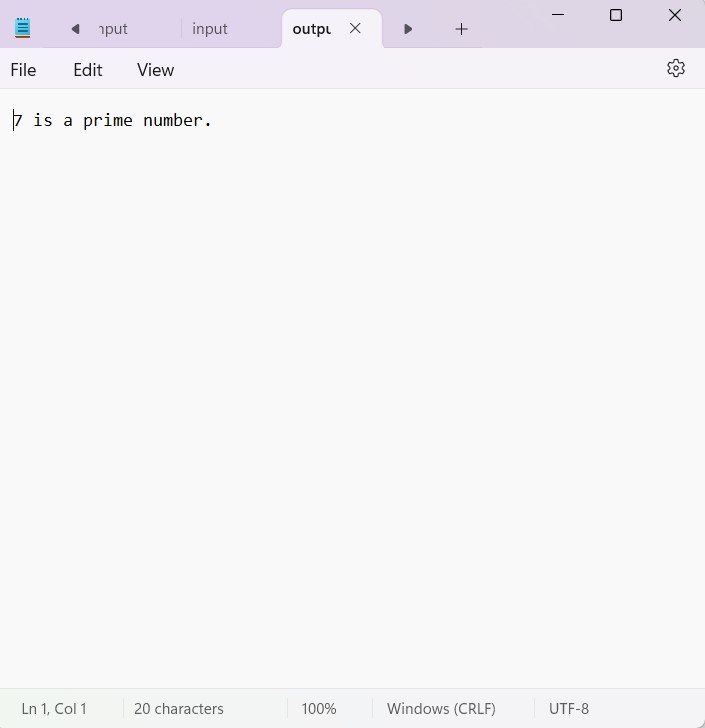
{

printf("%d is not a prime number.", n);

}

return 0;}





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

#include <stdio.h>

int main()

{ //Fibonacci series: 0 1 1 2 3 5 8 .... n

int n,a=0,b=1,c,i;

FILE\*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

while(!feof(input))

{

fscanf(input,"%d",&n);

printf("Enter the number of terms:%d ",n);

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

{

fprintf(output,"%d ",a);

c=a+b;

a=b;

b=c;

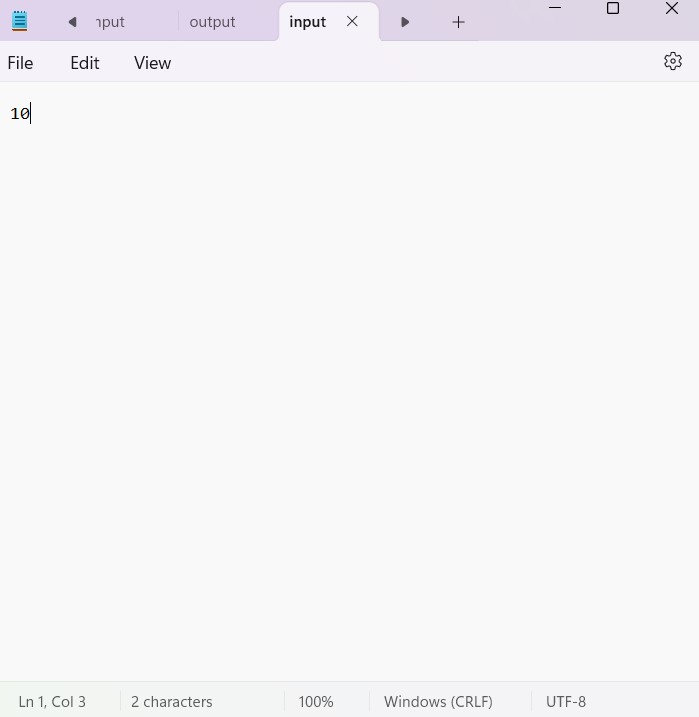
}}

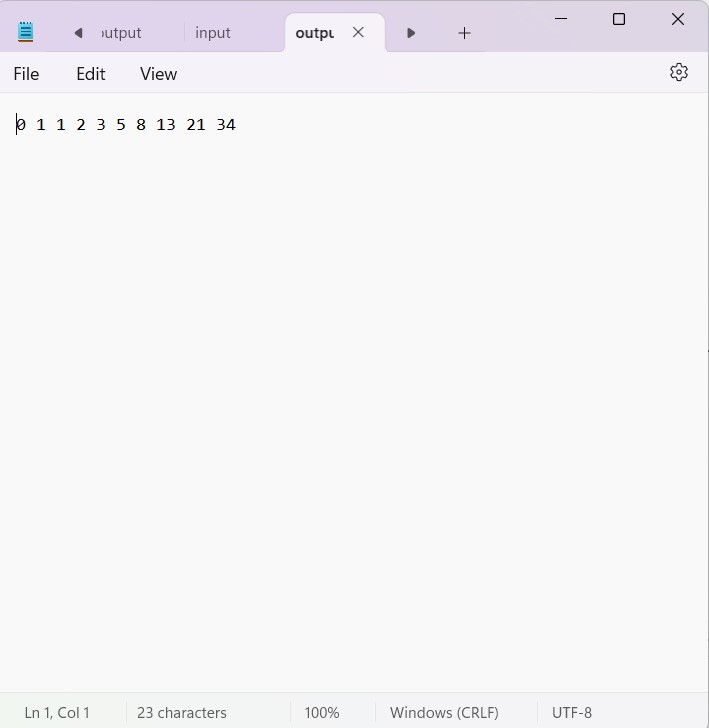
fclose(output);

fclose(input);

return 0;

}





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

#include<stdio.h>

int main()

{

//n=123=3+2+1=6

int i,n,r,sum=0,num;

FILE\*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

while(!feof(input))

{

fscanf(input,"%d",&n);

printf("Enter the number:%d",n);

num=n;

while(n>0)

{

r=n%10;

sum=sum+r;

n=n/10;

}}

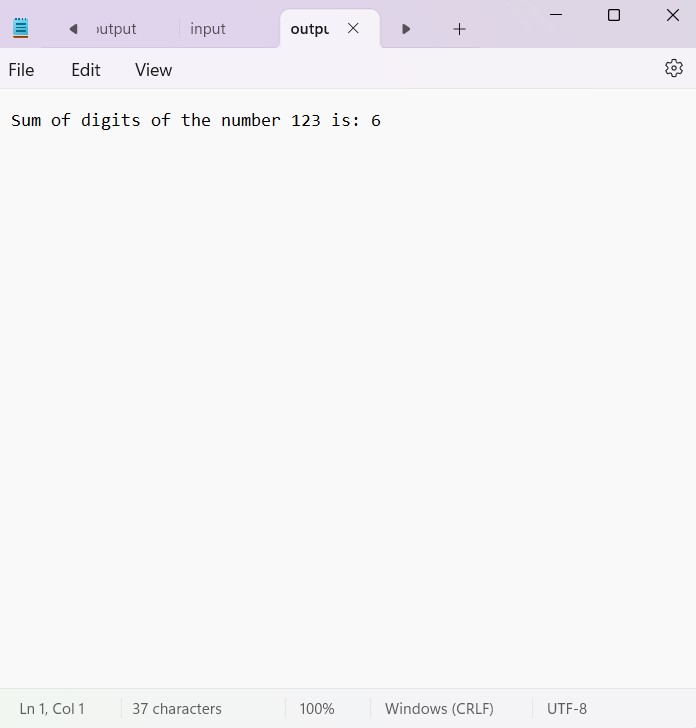
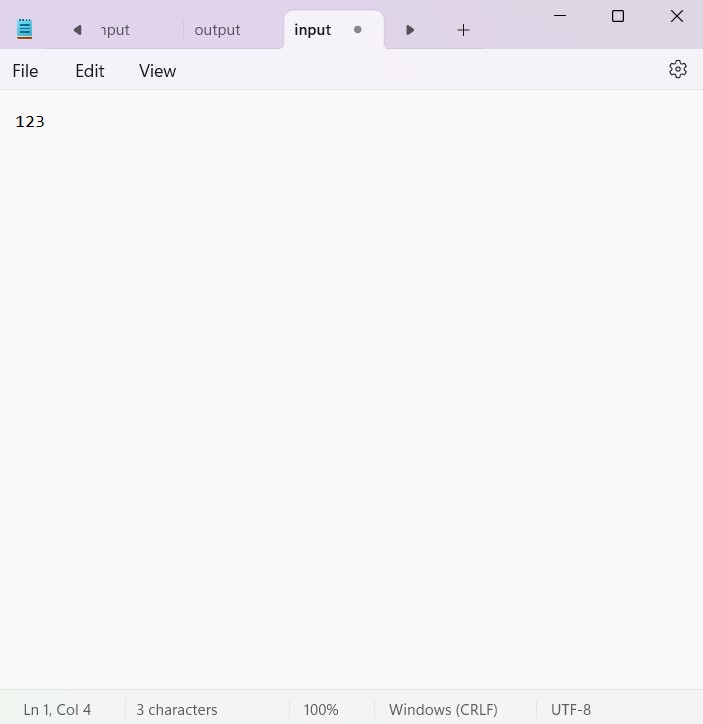
fprintf(output,"Sum of digits of the number %d is: %d",num,sum);

fclose(input);

fclose(output);

return 0;

}



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

#include<stdio.h>

int main()

{

//n=123=321

int n,i,r;

FILE \*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

while(!feof(input))

{

fscanf(input,"%d",&n);

printf("Enter the number:%d ",n);

while(n>0)

{

r=n%10;

fprintf(output,"%d",r);

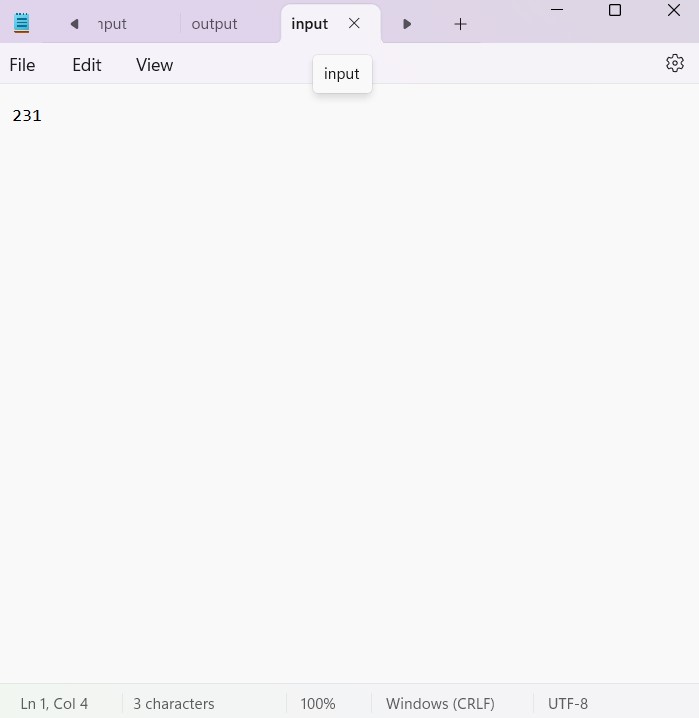
n=n/10;

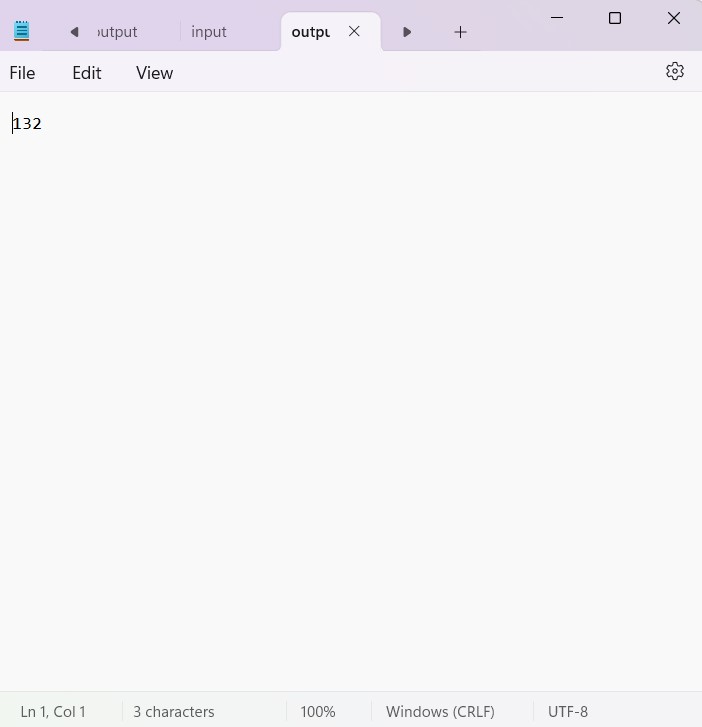
}}

fclose(input);

fclose(output);

return 0;}





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

#include<stdio.h>

int main()

{

//n=121=121

int n,num,s=0,r;

FILE \*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

while(!feof(input))

{

fscanf(input,"%d",&n);

printf("Enter the number:%d",n);

num=n;

while(n>0)

{

r=n%10;

s=r+(s\*10);

n=n/10;

}}

if(num==s)

{

fprintf(output,"%d is a palindrome.",num);

}

else

{

fprintf(output,"%d is not a palindrome.",num);

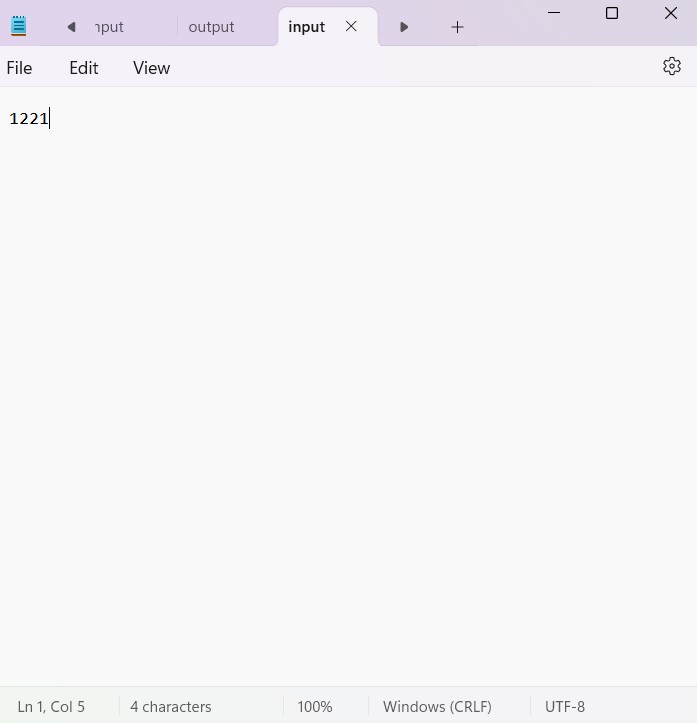
}

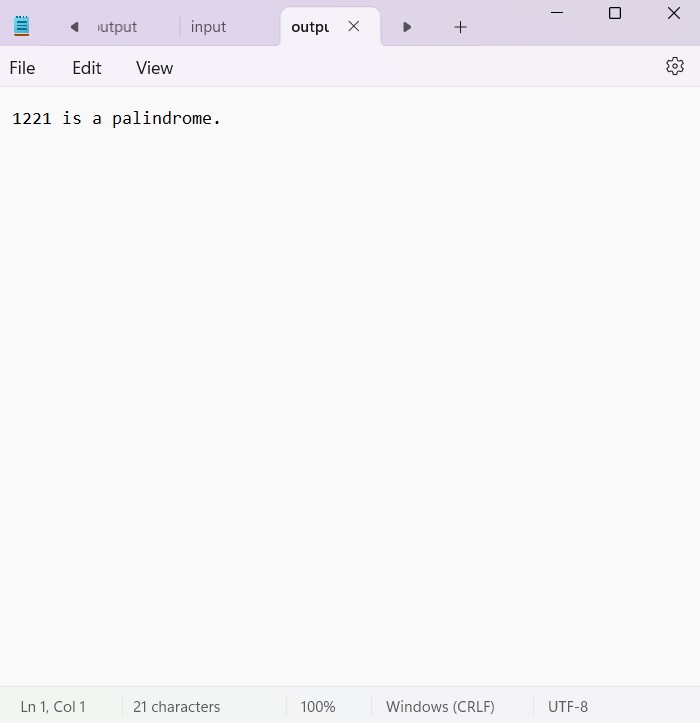
fclose(input);

fclose(output);

return 0;

}





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

**i) Area of circle:**

#include <stdio.h>

#include<conio.h>

int main() {

float pie = 3.14,area;

FILE \*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

int radius;

while(!feof(input))

{

fscanf(input,"%d",&radius);

printf("Enter The Radius of Cicle:%d",radius);

area = (float)(pie\* radius \* radius);

}

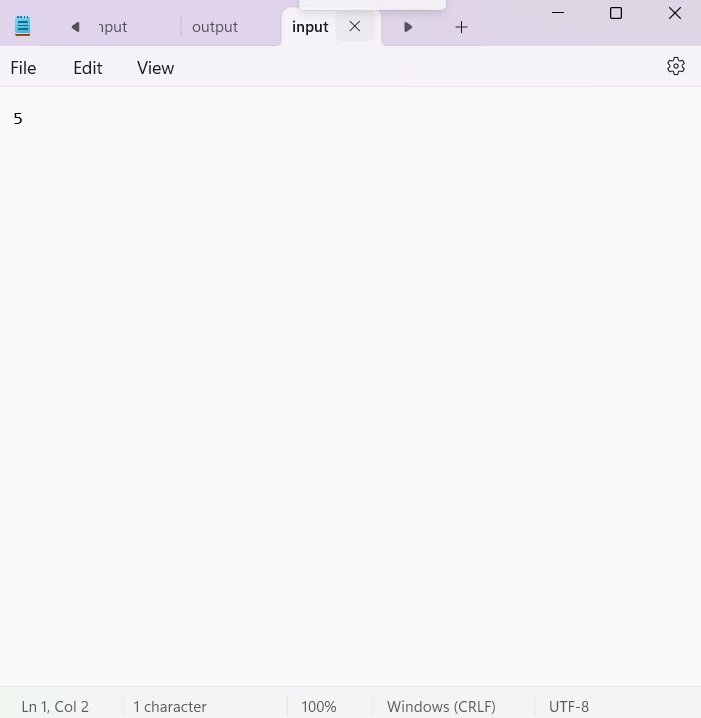
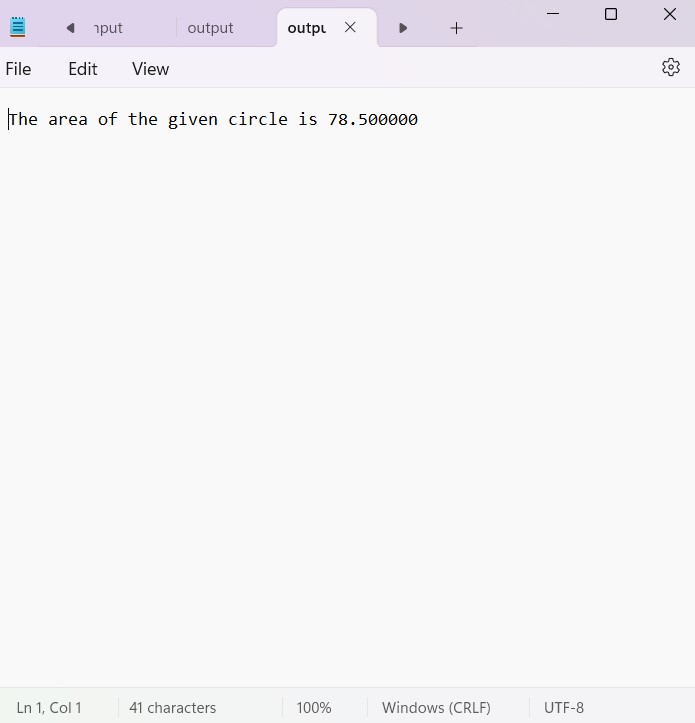
fprintf(output,"The area of the given circle is %f", area);

fclose(input);

fclose(output);

return 0;

}

**ii) Area of rectangle:**#include <stdio.h>

int main()

{

float length, width, area;

FILE \*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

while(!feof(input))

{

fscanf(input,"%f", &length);

printf("Enter the length of the rectangle:%.1f\n",length);

fscanf(input,"%f", &width);

printf("Enter the width of the rectangle:%.1f",width);

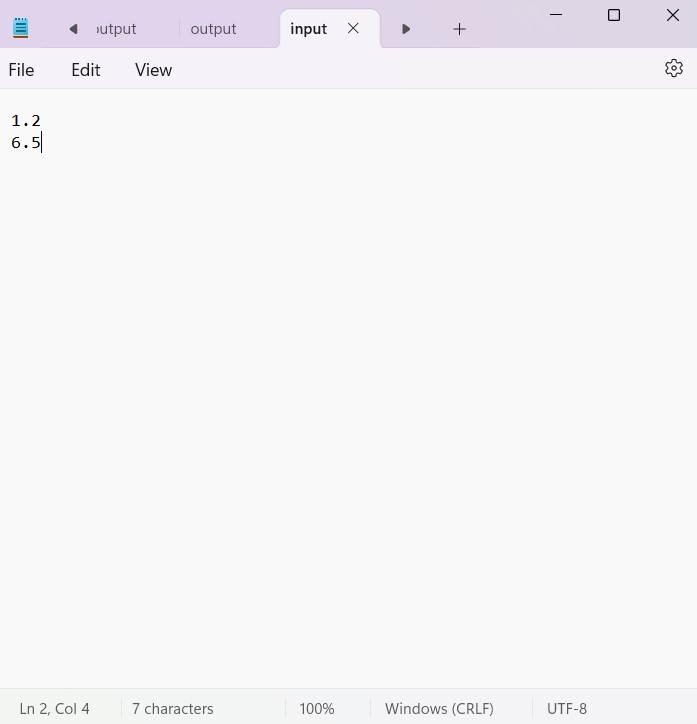
area = length \* width;

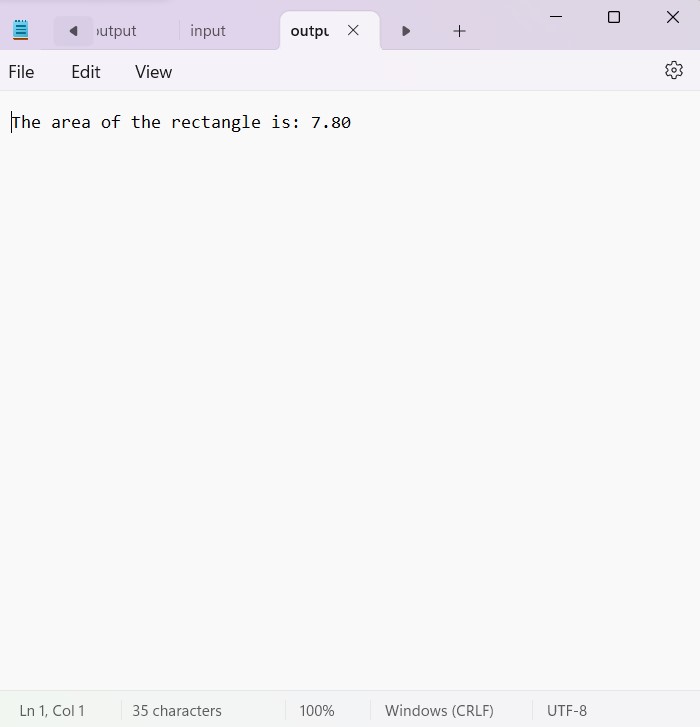
}

fprintf(output,"The area of the rectangle is: %.2f\n", area);

fclose(input);

fclose(output); return 0;}





**iii)Area of triangle:**

#include <stdio.h>

int main()

{

float base, height, area;

FILE \*input,\*output;

input=fopen("input.txt","r");

output=fopen("output.txt","w");

while(!feof(input))

{

fscanf(input,"%f",&base);

printf("Enter the base of the triangle:%.2f\n",base);

fscanf(input,"%f", &height);

printf("Enter the height of the triangle:%.2f\n",height);

area = 0.5 \* base \* height;

}

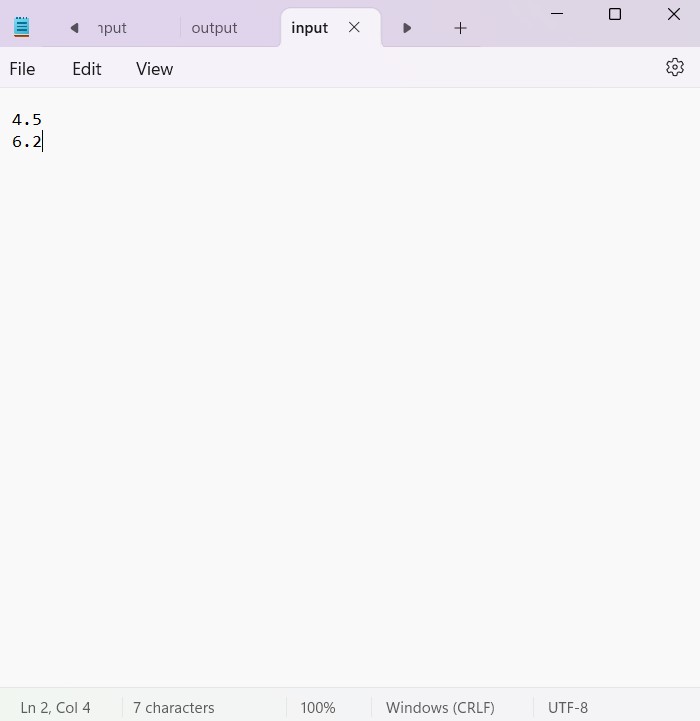
fprintf(output,"The area of the triangle is: %.2f\n", area);

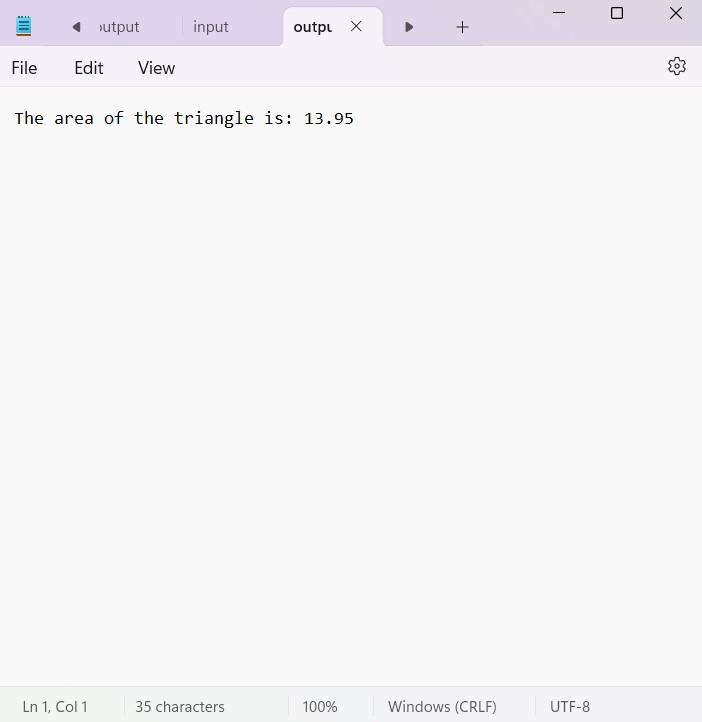
fclose(input);

fclose(output);

return 0;

}





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

#include<stdio.h>

#include<stdlib.h>

int main()

{

FILE \*fp,\*fp2;

fp=fopen("input.txt","r");

fp2=fopen("output.txt","w");

if (fp==NULL||fp2==NULL)

{

printf("not found\n");

}

int sw;

fscanf(fp,"%d",&sw);

printf("%d\n",sw);

int a,b,r;

fscanf(fp,"%d %d",&a,&b);

printf("%d %d\n",a,b);

switch(sw)

{

case 1:r=a+b;

fprintf(fp2,"%d\n",r);

break;

case 2:r=a-b;

fprintf(fp2,"%d\n",r);

break;

case 3:r=a\*b;

fprintf(fp2,"%d\n",r);

break;

case 4:r=a/b;

fprintf(fp2,"%d\n",r);

break;

case 5:r=a%b;

fprintf(fp2,"%d\n",r);

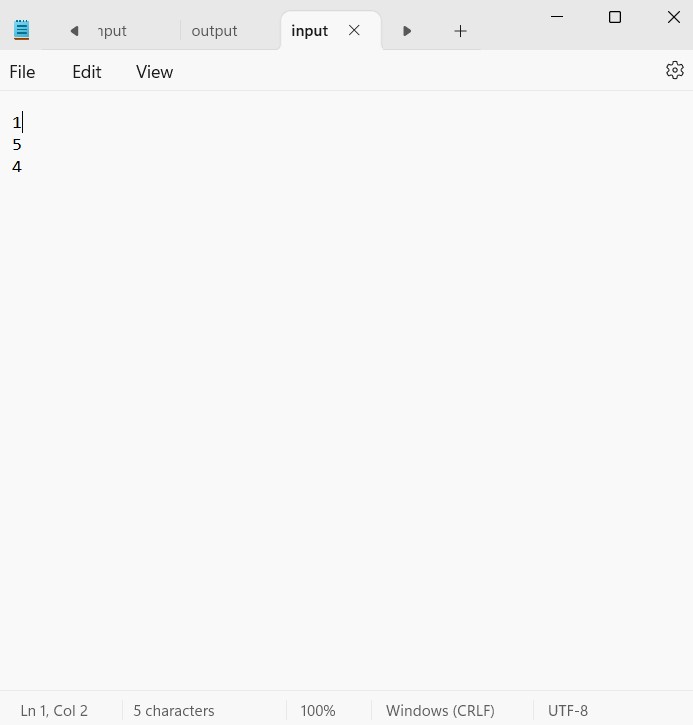
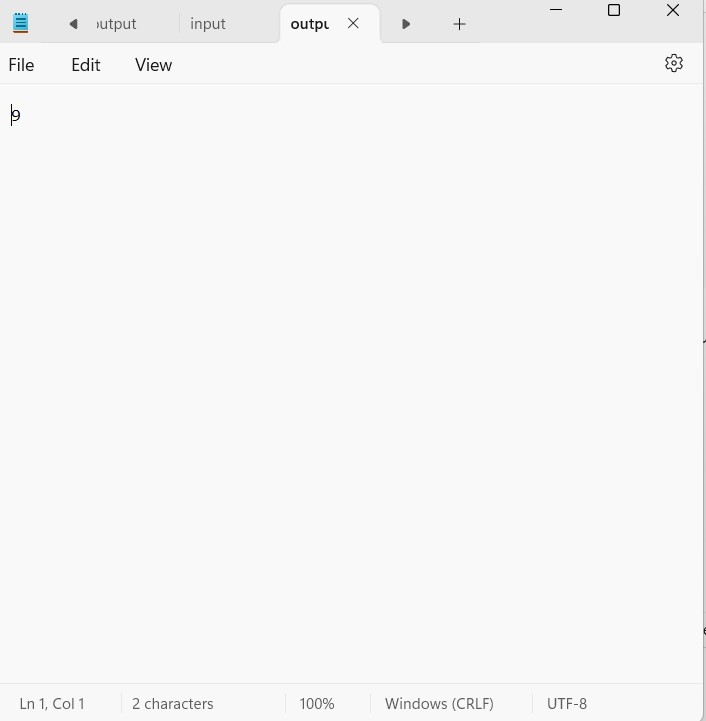
break;

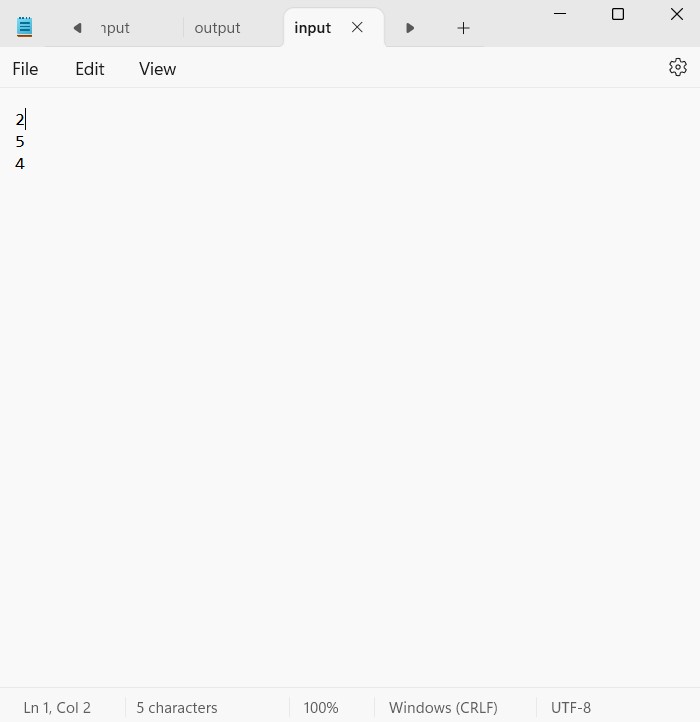
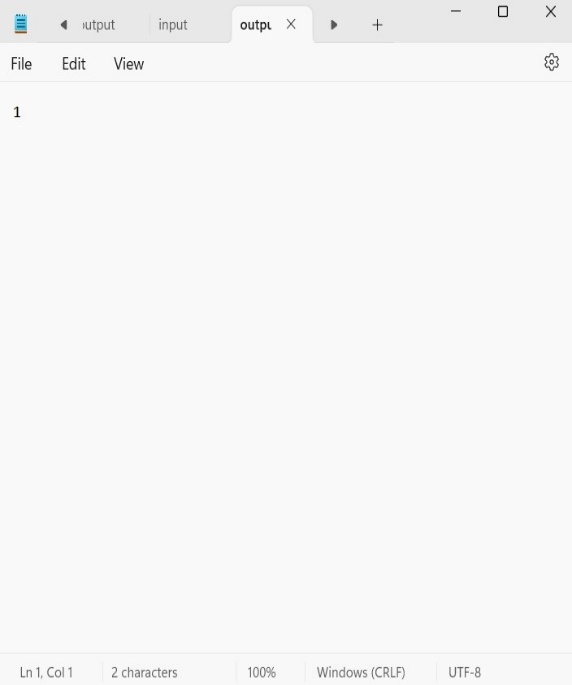
}

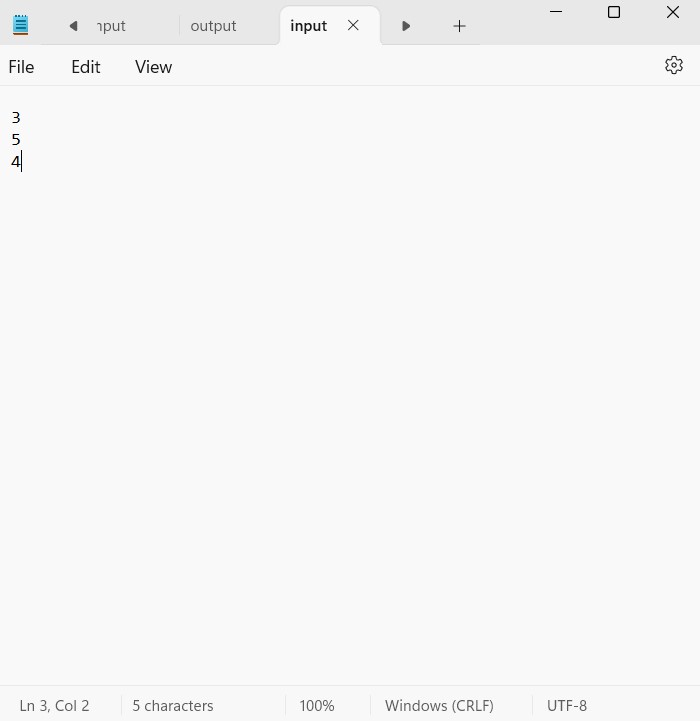
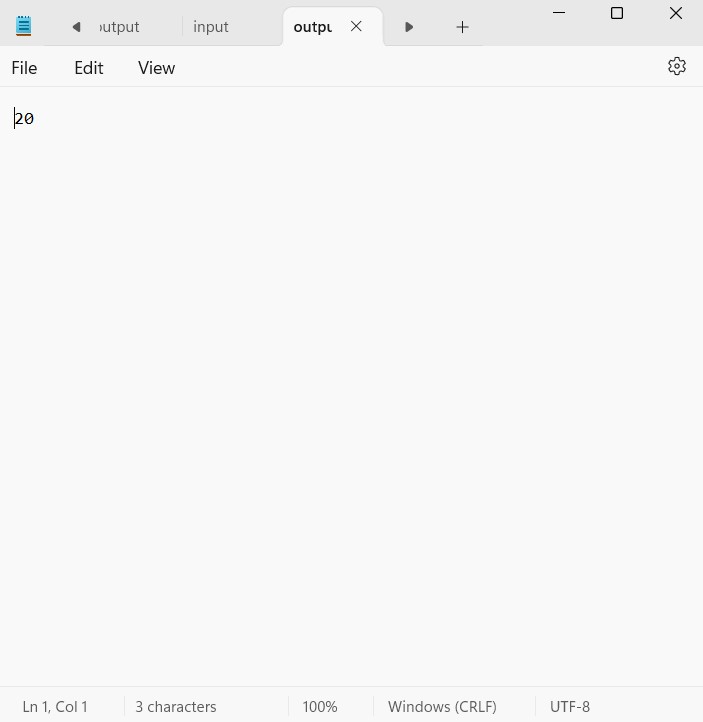
fclose(fp);

fclose(fp2);

}

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

#include <stdio.h>

int main() {

int a[100];

FILE \*fp, \*fp2;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

if (fp == NULL || fp2 == NULL) {

printf("File not found\n");

return 1;

}

int n, sw;

int s = 0;

int min, max;

fscanf(fp, "%d", &n);

printf("%d\n", n);

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

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

fprintf(fp2, "%d\t", a[i]);

}

fscanf(fp, "%d", &sw);

switch (sw) {

case 1:

max = a[0];

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

if (a[i] > max) {

max = a[i];

}

}

fprintf(fp2, "\nMax: %d\n", max);

break;

case 2:

min = a[0];

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

if (a[i] < min) {

min = a[i];

}

}

fprintf(fp2, "\nMin: %d\n", min);

break;

case 3:

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

s += a[i];

}

fprintf(fp2, "\nSum: %d\n", s);

double avg = (double)s / n;

fprintf(fp2, "Average: %lf\n", avg);

break;

default:

printf("Invalid switch case\n");

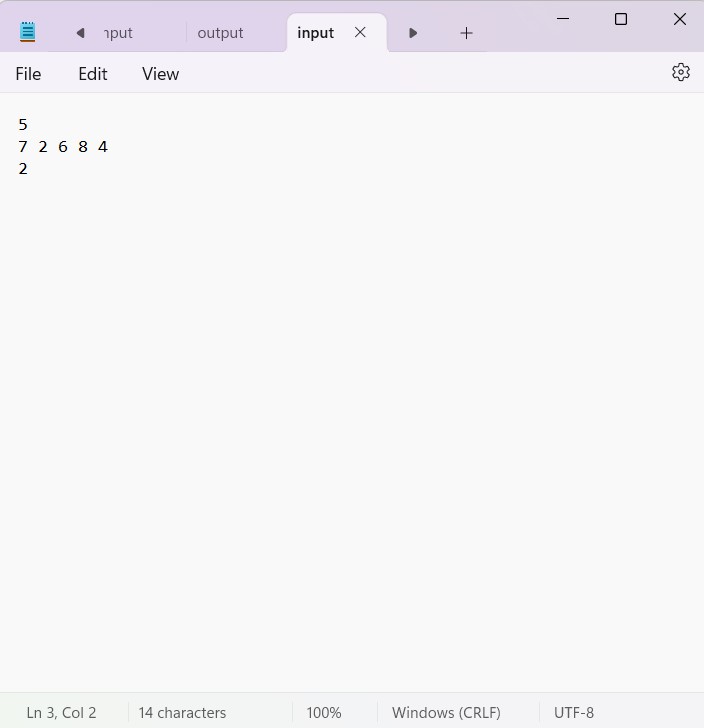
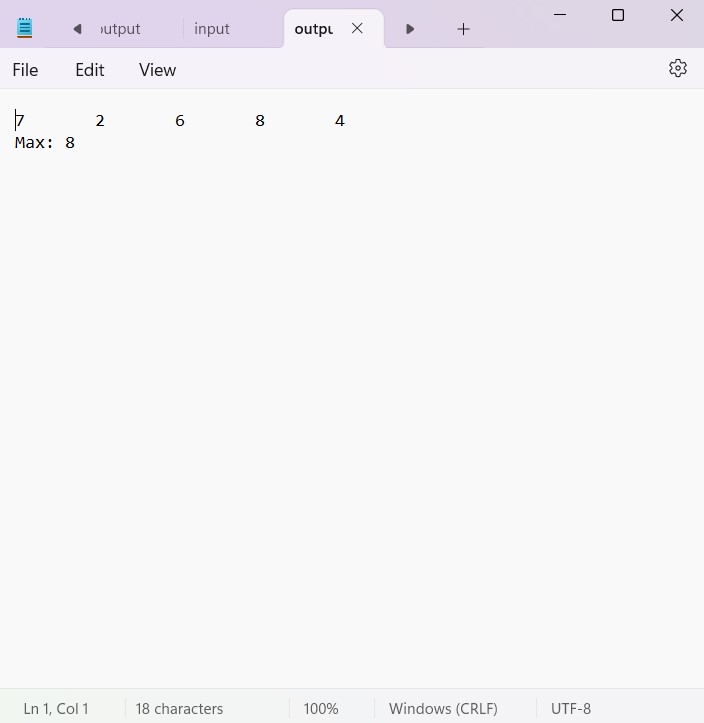
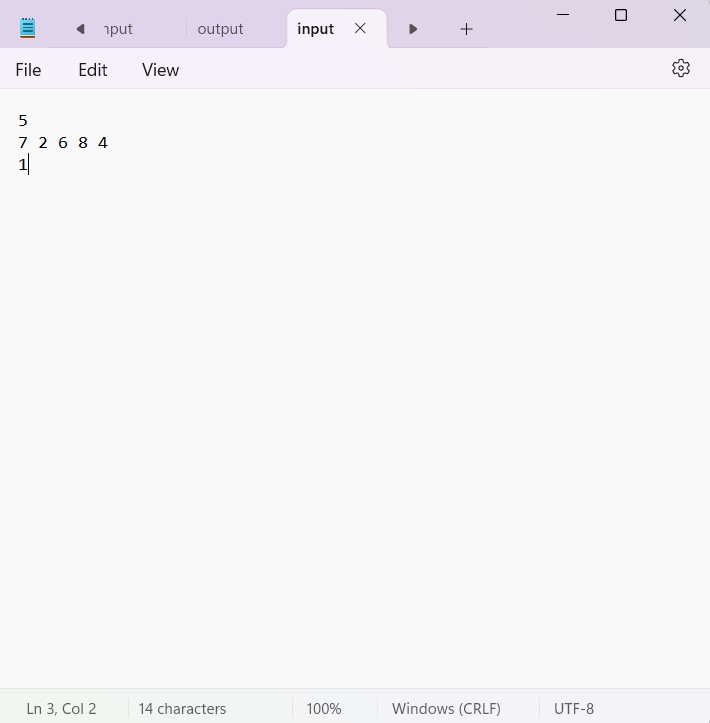
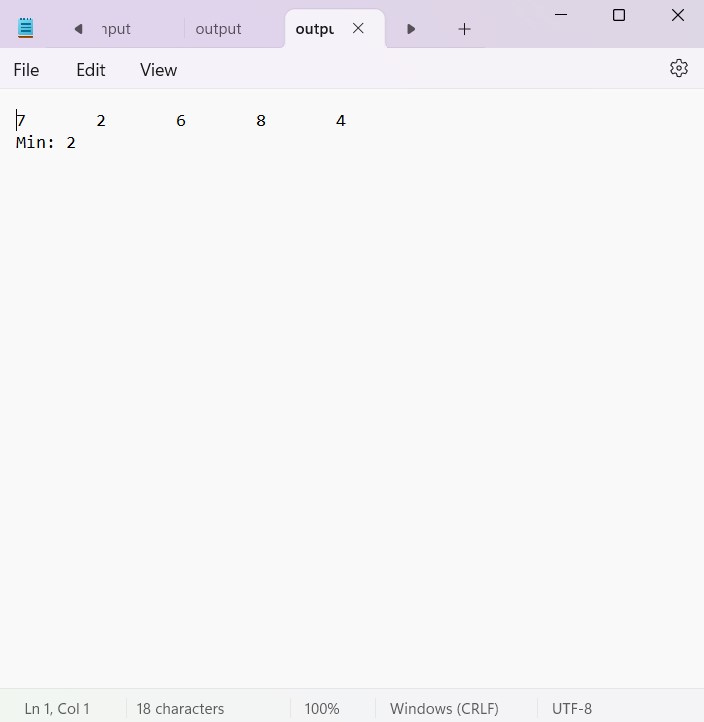
}

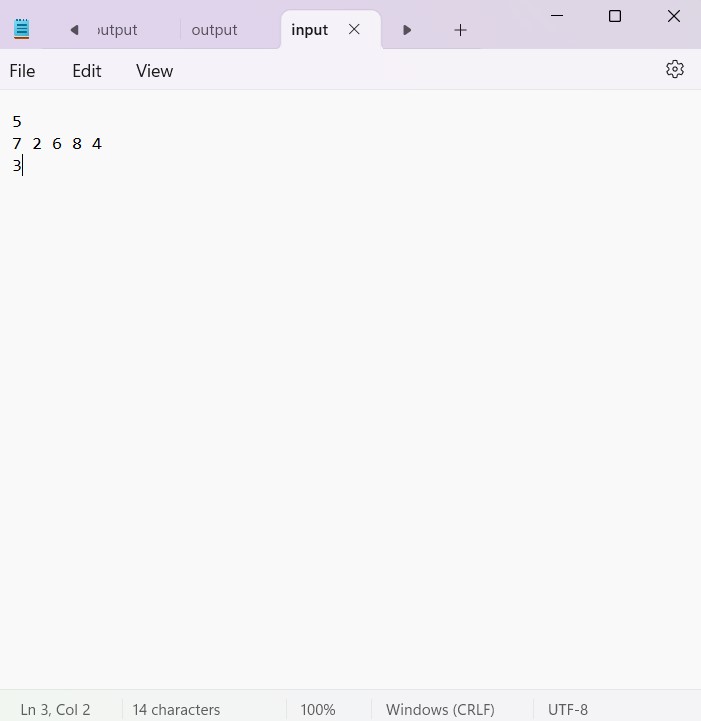
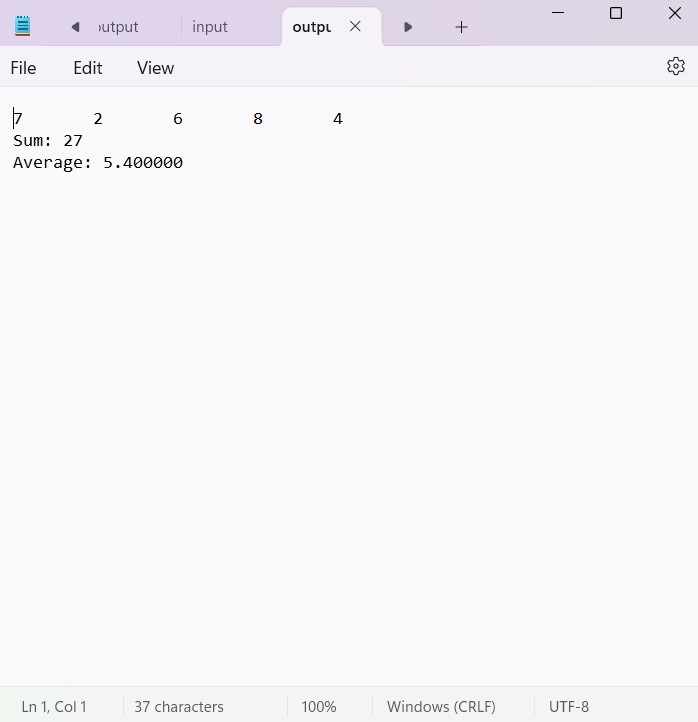
fclose(fp);

fclose(fp2);

return 0;

}

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

#include <stdio.h>

#include <string.h>

int main() {

char s[100];

char k[100];

FILE \*fp, \*fp2;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

if (fp == NULL || fp2 == NULL) {

printf("File not found\n");

return 1;

}

int sw;

fscanf(fp, "%d", &sw);

switch (sw) {

case 1:

fscanf(fp, "%s", s);

fscanf(fp, "%s", k);

strcat(s, k);

fprintf(fp2, "%s", s);

break;

case 2:

fscanf(fp, "%s", s);

fscanf(fp, "%s", k);

fprintf(fp2, "b/f Copying string: %s\n", s);

strcpy(s, k);

fprintf(fp2, "Copied string: %s", s);

break;

case 3:

fscanf(fp, "%s", s);

fscanf(fp, "%s", k);

int res = strcmp(s, k);

if (res == 0)

fprintf(fp2, "Strings are equal\n");

else

fprintf(fp2, "Strings are not equal\n");

break;

default:

printf("Invalid option\n");

break;

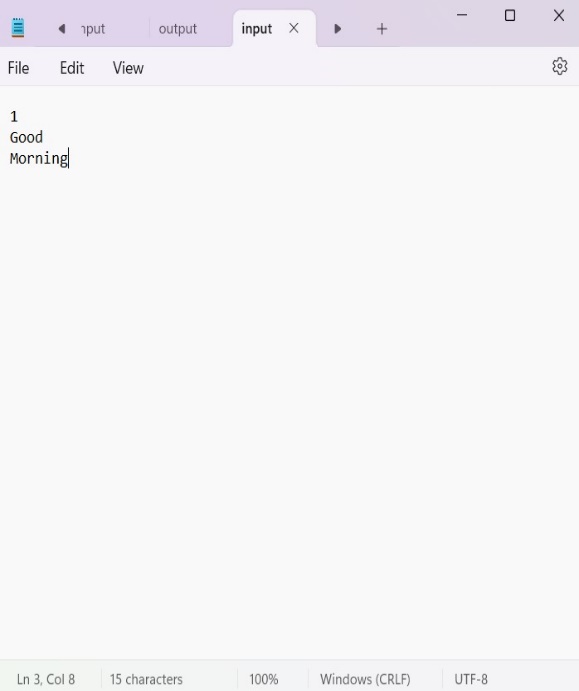
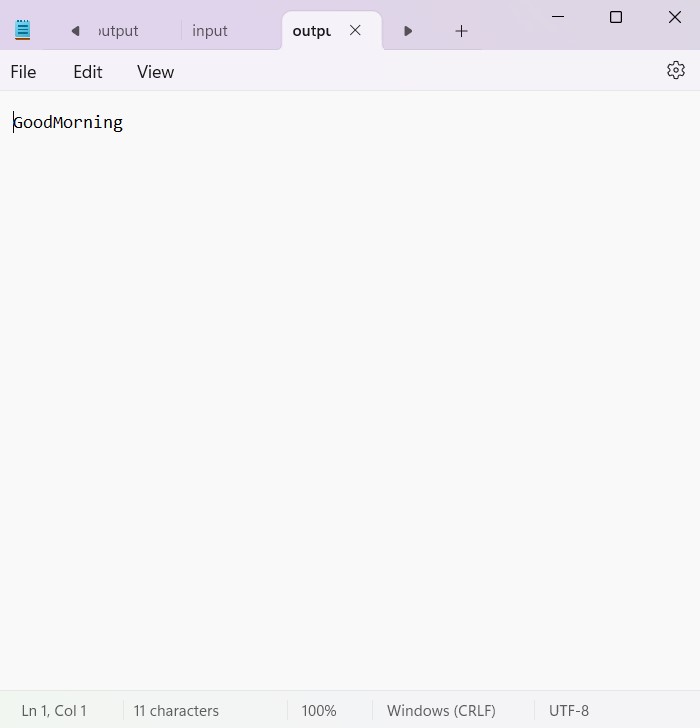
}

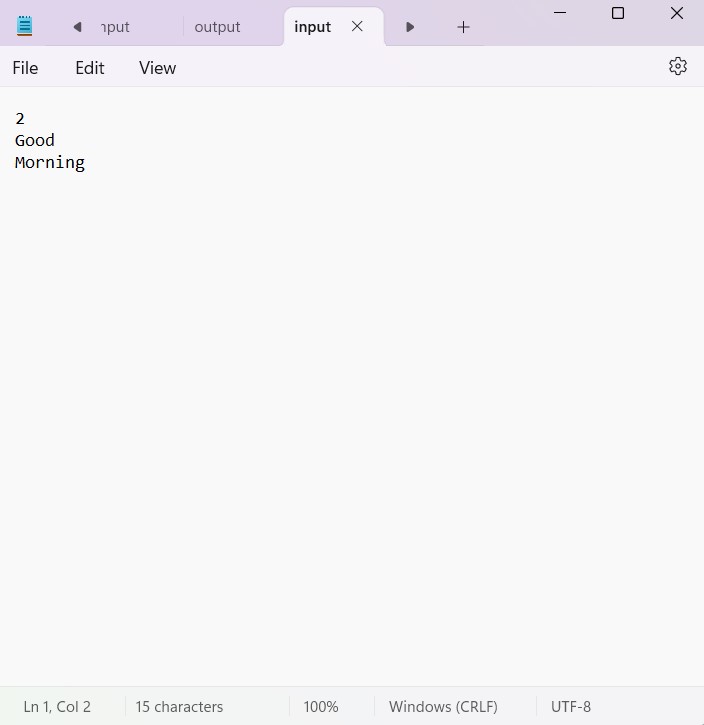
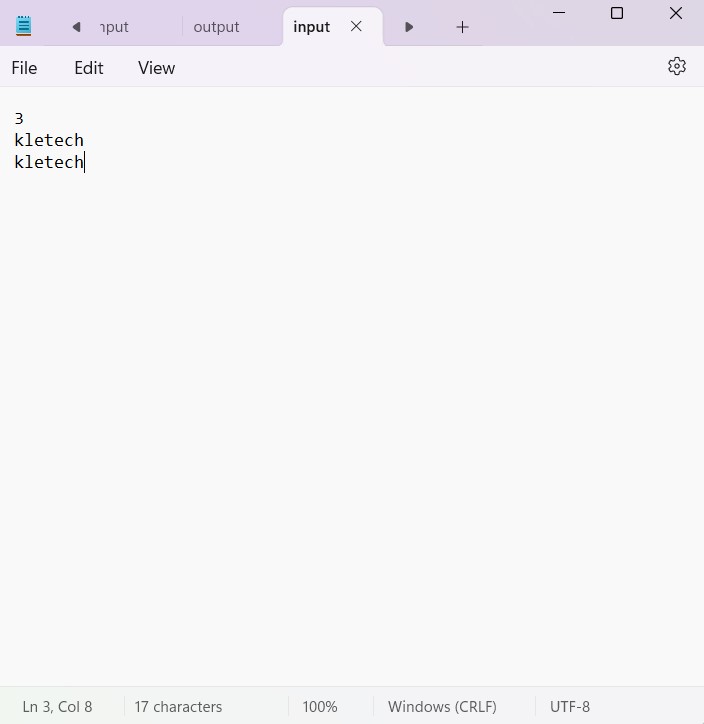
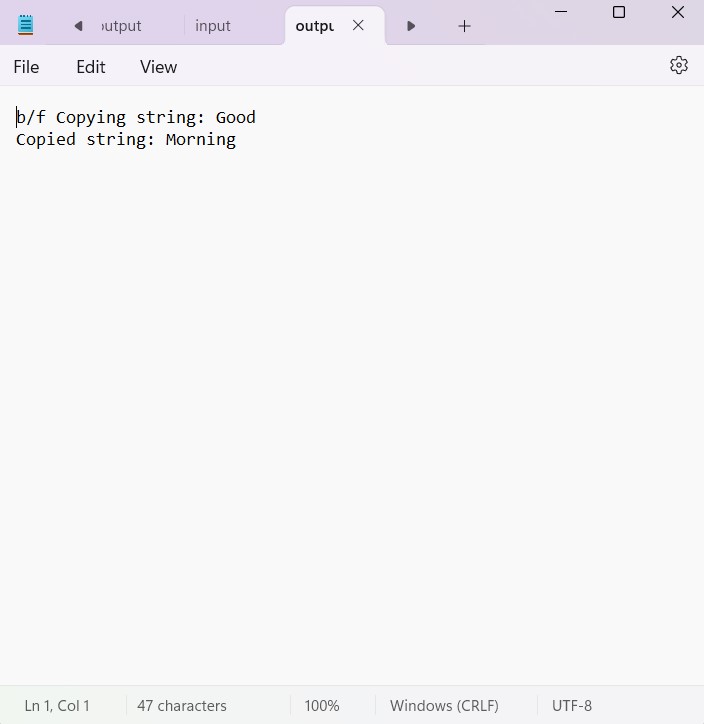
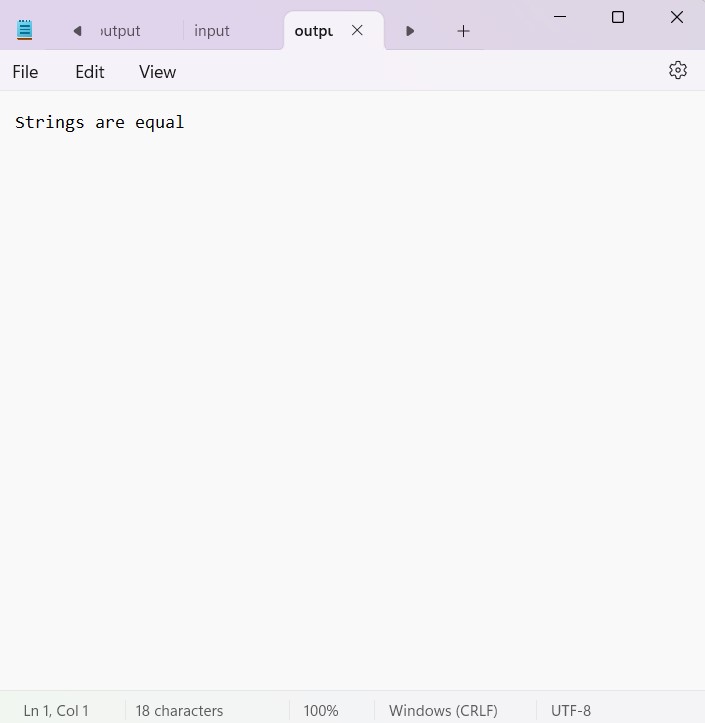
fclose(fp);

fclose(fp2);

return 0;

}

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

#include <stdio.h>

#include <stdio.h>

#include <string.h>

int main() {

FILE \*fp, \*fp2;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

int n, x, c = 0;

int a[100];

if (fp == NULL || fp2 == NULL) {

printf("File not found\n");

return 1;

}

fscanf(fp, "%d", &x);

fscanf(fp, "%d", &n);

printf("%d\n", n);

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

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

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

}

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

if (a[i] == x) {

c++;

fprintf(fp2, "found at position = %d\n", i + 1);

}

}

if (c == 0) {

fprintf(fp2, "not found\n");

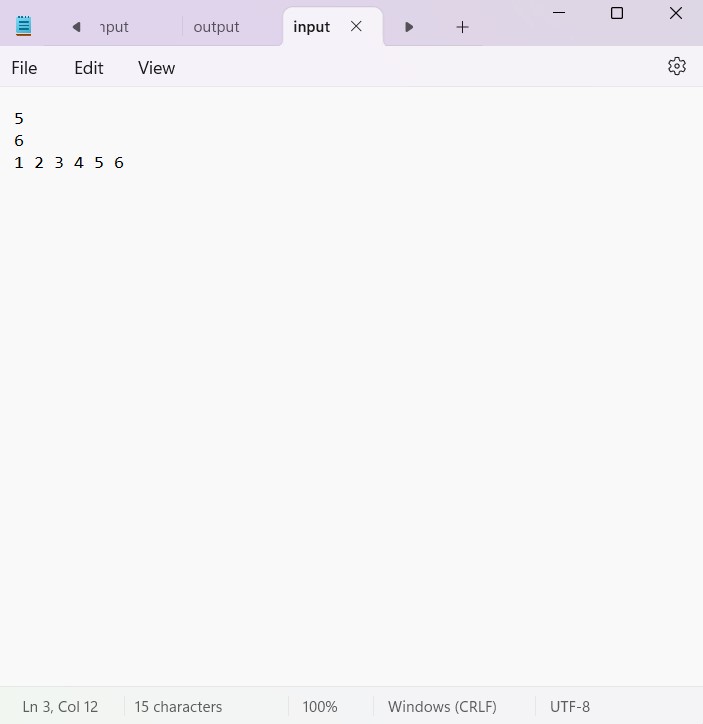
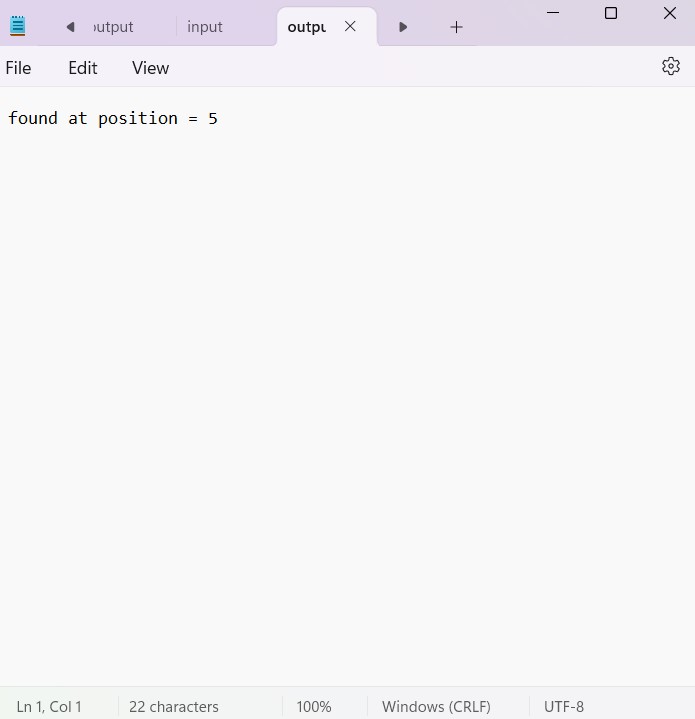
}

fclose(fp);

fclose(fp2);

return 0;

}

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

#include <stdio.h>

int main() {

FILE \*fp, \*fp2;

int flag = 0;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

int n;

fscanf(fp, "%d", &n);

printf("%d ",n);

int a[5];

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

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

}

int key;

fscanf(fp,"%d",&key);

printf("%d ",key);

int left = 0;

int right = n - 1;

int result = -1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (a[mid] == key) {

result = mid;

break;

}

if (a[mid] < key)

left = mid + 1;

else

right = mid - 1;

}

if (result != -1)

fprintf(fp2, "Element %d found at index %d\n", key, result);

else

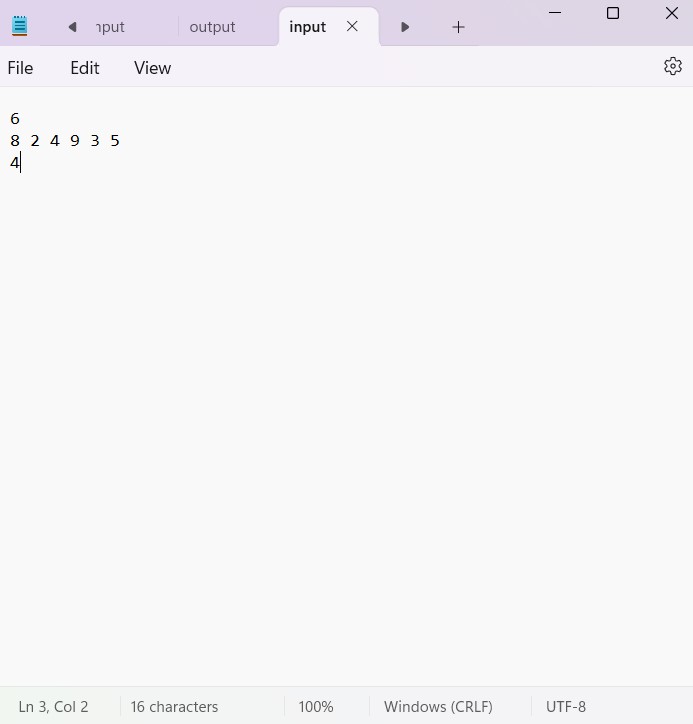
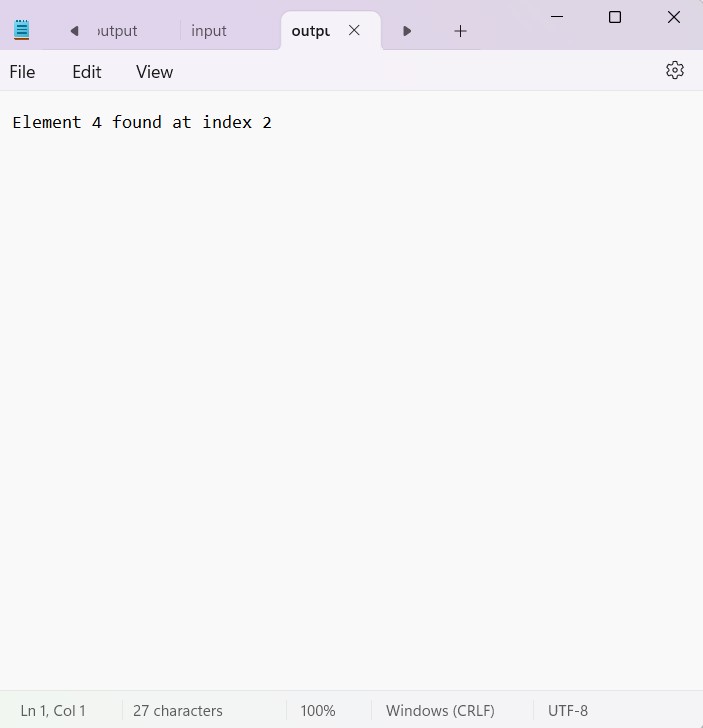
fprintf(fp2, "Element %d not found in the array\n", key);

fclose(fp);

fclose(fp2);

return 0;

}

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

#include <stdio.h>

void selectionSort(int arr[], int n) {

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

int min\_idx = i;

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

if (arr[j] < arr[min\_idx])

min\_idx = j;

}

int temp = arr[i];

arr[i] = arr[min\_idx];

arr[min\_idx] = temp;

}

}

int main() {

FILE \*fp, \*fp2;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

if (fp == NULL || fp2 == NULL) {

printf("File not found\n");

return 1;

} int n;

fscanf(fp, "%d", &n);

int arr[n];

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

fscanf(fp, "%d", &arr[i]);

}

selectionSort(arr, n);

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

fprintf(fp2, "%d ", arr[i]);

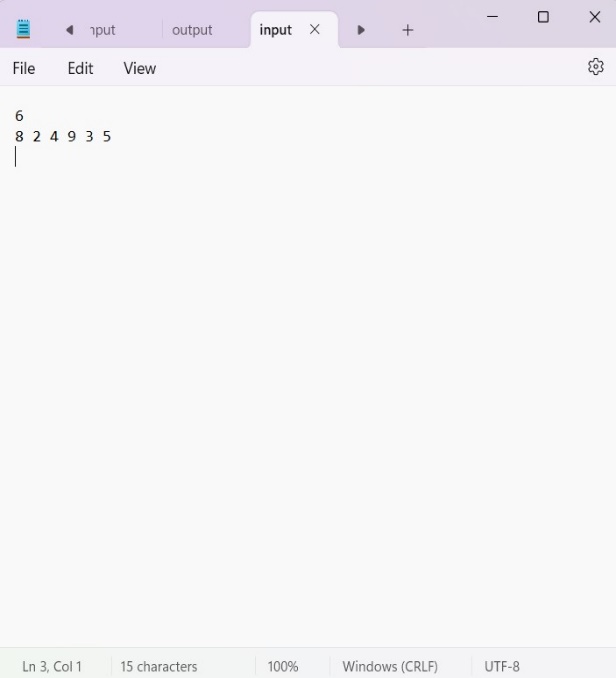
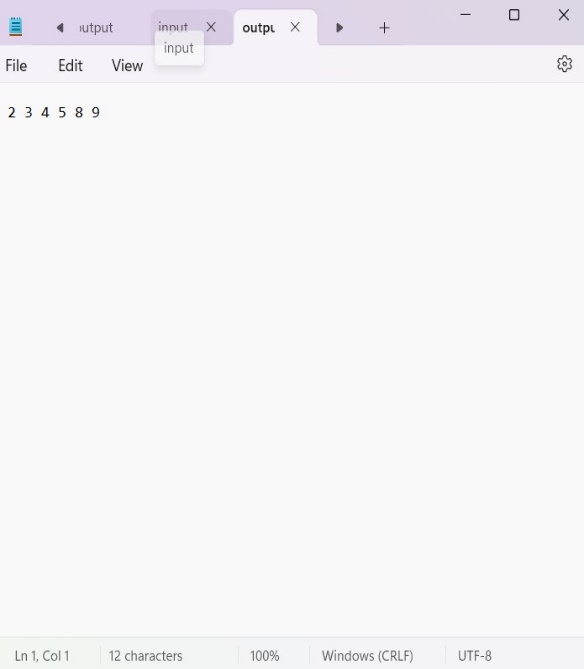
}

fclose(fp);

fclose(fp2);

return 0;

}

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

#include <stdio.h>

void bubbleSort(int arr[], int n) {

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

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

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

FILE \*fp, \*fp2;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

if (fp == NULL || fp2 == NULL) {

printf("File not found\n");

return 1;

}

int n;

fscanf(fp, "%d", &n);

int arr[n];

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

fscanf(fp, "%d", &arr[i]);

}

bubbleSort(arr, n);

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

fprintf(fp2, "%d ", arr[i]);

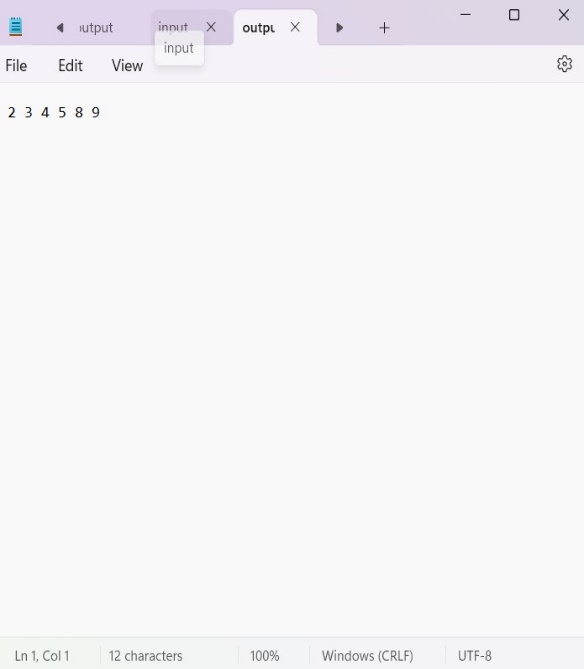
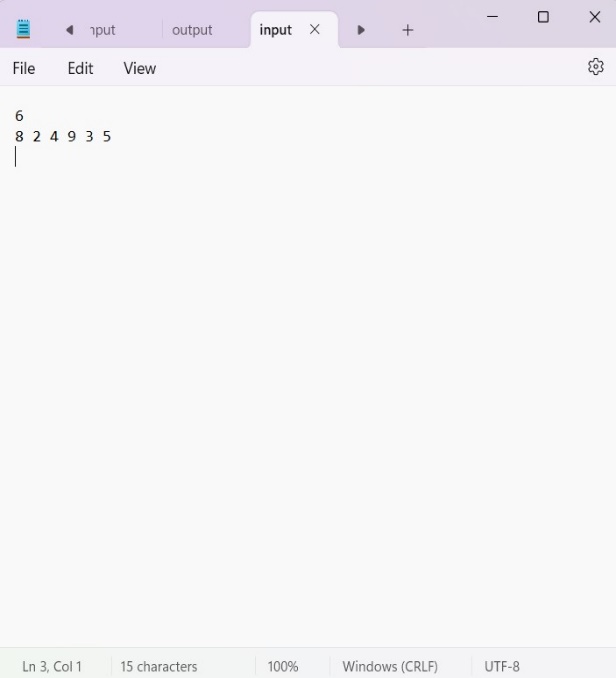
}

fclose(fp);

fclose(fp2);

return 0;

}



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

#include <stdio.h>

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

int main() {

FILE \*fp, \*fp2;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

if (fp == NULL || fp2 == NULL) {

printf("File not found\n");

return 1;

}

int n;

fscanf(fp, "%d", &n);

int arr[n];

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

fscanf(fp, "%d", &arr[i]);

}

insertionSort(arr, n);

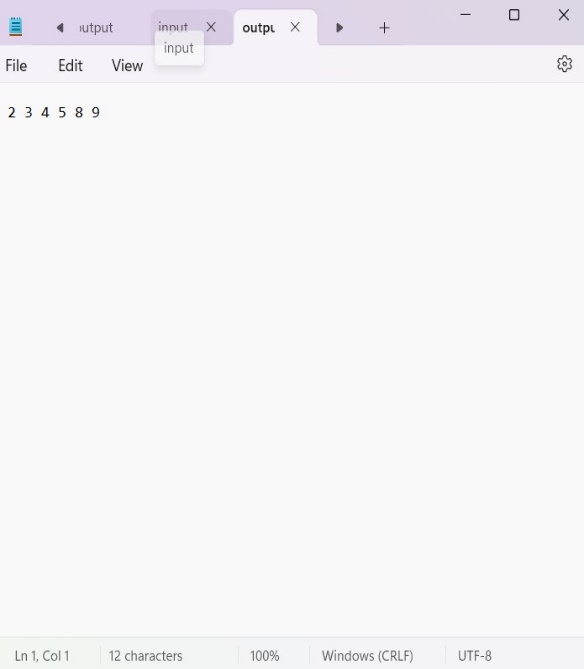
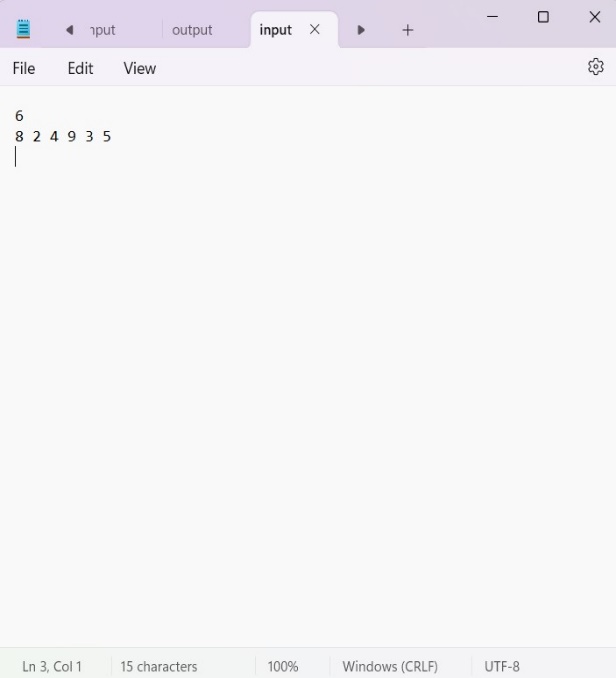
for (int i = 0; i < n; i++) {

fprintf(fp2, "%d ", arr[i]); }

fclose(fp);

fclose(fp2);

return 0;}



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

#include <stdio.h>

void readM(FILE \*fp, int m[][100], int r, int c) {

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

for (int j = 0; j < c; j++) {

fscanf(fp, "%d", &m[i][j]);

}

}

}

void print(FILE \*fp, int m[][100], int r, int c) {

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

for (int j = 0; j < c; j++) {

fprintf(fp, "%d ", m[i][j]);

}

fprintf(fp, "\n");

}

}

void addM(int m1[][100], int m2[][100], int res[][100], int r, int c) {

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

for (int j = 0; j < c; j++) {

res[i][j] = m1[i][j] + m2[i][j];

}

}

}

void subM(int m1[][100], int m2[][100], int res[][100], int r, int c) {

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

for (int j = 0; j < c; j++) {

res[i][j] = m1[i][j] - m2[i][j];

}

}

}

void mulM(int mat1[][100], int mat2[][100], int res[][100], int r1, int c1, int c2) {

for (int i = 0; i < r1; i++) {

for (int j = 0; j < c2; j++) {

res[i][j] = 0;

for (int k = 0; k < c1; k++) {

res[i][j] += mat1[i][k] \* mat2[k][j];

}

}

}

}

void transpose(int m[][100], int tr[][100], int r, int c) {

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

for (int j = 0; j < r; j++) {

tr[i][j] = m[j][i];

}

}

}

int main() {

FILE \*fp, \*fp2;

fp = fopen("input.txt", "r");

fp2 = fopen("output.txt", "w");

if (fp == NULL || fp2 == NULL) {

printf("File not found\n");

return 1;

}

int r1, c1, r2, c2;

fscanf(fp, "%d %d", &r1, &c1);

fscanf(fp, "%d %d", &r2, &c2);

if (r1 != r2 || c1 != c2) {

printf("Matrix operations require matrices of the same dimensions.\n");

fclose(fp);

fclose(fp2);

return 1;

}

int m1[100][100], m2[100][100], res[100][100], tr[100][100];

readM(fp, m1, r1, c1);

readM(fp, m2, r2, c2);

addM(m1, m2, res, r1, c1);

fprintf(fp2, "Matrix Addition Result:\n");

print(fp2, res, r1, c1);

subM(m1, m2, res, r1, c1);

fprintf(fp2, "\nMatrix Subtraction Result:\n");

print(fp2, res, r1, c1);

mulM(m1, m2, res, r1, c1, c2);

fprintf(fp2, "\nMatrix Multiplication Result:\n");

print(fp2, res, r1, c2);

transpose(m1, tr, r1, c1);

fprintf(fp2, "\nTranspose of Matrix 1:\n");

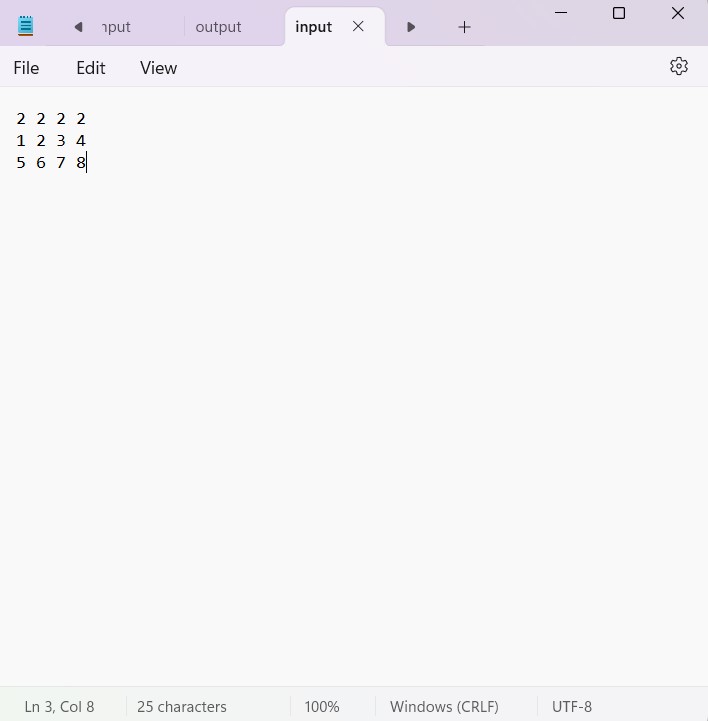
print(fp2, tr, c1, r1);

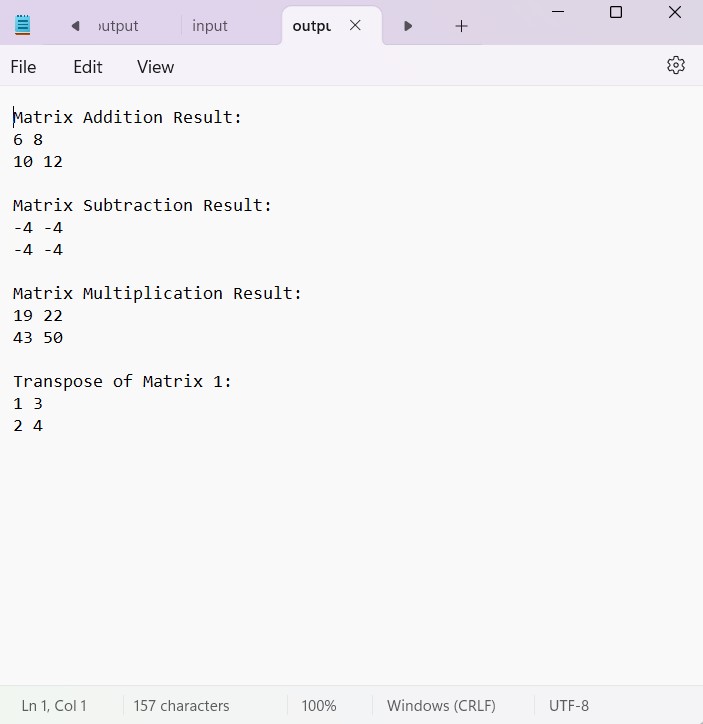
fclose(fp);

fclose(fp2);

return 0;

}





**Part- C**

|  |
| --- |
| **1.using sll insert at end using random function and store it in a file** |
| Code:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtEnd(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* last = \*head\_ref;  newNode->data = newData;  newNode->next = NULL;  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  void displayList(struct Node\* node) {  while (node != NULL) {  printf("%d\n", node->data);  node = node->next;  }  }  void writeToFile(struct Node\* head) {  FILE\* fp = fopen("linked\_list.txt", "w");  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  void readFromFile(struct Node\*\* head\_ref) {  FILE\* fp = fopen("linked\_list.txt", "r");  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(head\_ref, data);  }  fclose(fp);  }  int main() {  struct Node\* head = NULL;  int i;  for (i = 0; i < 5; i++) {  insertAtEnd(&head, rand() % 100); // Insert random numbers  }  printf("Initial linked list:\n ");  displayList(head);  writeToFile(head);  printf("Data written to file.\n");  printf("Reading from file...\n");  struct Node\* new\_head = NULL;  readFromFile(&new\_head);  printf("Linked list after reading from file:\n ");  displayList(new\_head);  return 0;  } |
| Output: |
|  |
| **2.using sll insert at end using random function and insert elements in the front and store it in a file** |
| Code: #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtEnd(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* last = \*head\_ref;  newNode->data = newData;  newNode->next = NULL;  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  void insertAtFront(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = newData;  newNode->next = \*head\_ref;  \*head\_ref = newNode;  }  void displayList(struct Node\* node) {  while (node != NULL) {  printf("%d ", node->data);  node = node->next;  }  printf("\n");  }  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file for writing.\n");  return;  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  void readFromFile(struct Node\*\* head\_ref, const char\* filename) {  FILE\* fp = fopen(filename, "r");  if (fp == NULL) {  printf("Error opening file for reading.\n");  return;  }  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(head\_ref, data);  }  fclose(fp);  }  int main() {  struct Node\* head = NULL;  const char\* filename = "linked\_list.txt";  int i;  // Insert elements at the end and store in a file  for ( i = 0; i < 5; i++) {  insertAtEnd(&head, rand() % 100); // Insert random numbers  }  printf("Initial linked list:\n");  displayList(head);  writeToFile(head, filename);  printf("Data written to file '%s'.\n", filename);  // Read from file, insert elements at the front, and store in another file  struct Node\* new\_head = NULL;  const char\* new\_filename = "updated\_list.txt";  readFromFile(&new\_head, filename);  printf("Linked list after reading from file:\n");  displayList(new\_head);  // Insert two elements at the front generated randomly  for ( i = 0; i <2; i++) {  insertAtFront(&new\_head, rand() % 100);  }  printf("Linked list after inserting elements at the front:\n");  displayList(new\_head);  // Write the updated list to another file  writeToFile(new\_head, new\_filename);  printf("Updated data written to file '%s'.\n", new\_filename);  return 0;  } |
|  |
|  |
|  |
| **3.using sll insert at end using random function and delete elements in the front and store it in a file** |
| Code:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtEnd(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* last = \*head\_ref;  newNode->data = newData;  newNode->next = NULL;  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  void insertAtFront(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = newData;  newNode->next = \*head\_ref;  \*head\_ref = newNode;  }  void displayList(struct Node\* node) {  while (node != NULL) {  printf("%d ", node->data);  node = node->next;  }  printf("\n");  }  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file for writing.\n");  return;  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  void readFromFile(struct Node\*\* head\_ref, const char\* filename) {  FILE\* fp = fopen(filename, "r");  if (fp == NULL) {  printf("Error opening file for reading.\n");  return;  }  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(head\_ref, data);  }  fclose(fp);  }  void deleteFront(struct Node\*\* head\_ref) {  if (\*head\_ref == NULL) {  printf("List is empty. Cannot delete.\n");  return;  }  struct Node\* temp = \*head\_ref;  \*head\_ref = (\*head\_ref)->next;  free(temp);  }  int main() {  struct Node\* head = NULL;  const char\* filename = "linked\_list.txt";  int i;  // Insert elements at the end and store in a file  for ( i = 0; i < 5; i++) {  insertAtEnd(&head, rand() % 100); // Insert random numbers  }  printf("Initial linked list:\n");  displayList(head);  writeToFile(head, filename);  printf("Data written to file '%s'.\n", filename);  // Read from file, insert elements at the front, and store in another file  struct Node\* new\_head = NULL;  const char\* new\_filename = "updated\_list.txt";  readFromFile(&new\_head, filename);  printf("Linked list after reading from file:\n");  displayList(new\_head);  // Delete first two elements from the front  deleteFront(&new\_head);  deleteFront(&new\_head);  printf("Linked list after deleting first two elements from the front:\n");  displayList(new\_head);  // Write the updated list to another file  writeToFile(new\_head, new\_filename);  printf("Updated data written to file '%s'.\n", new\_filename);  return 0;  } |
|  |
|  |
|  |
| **4.using sll insert at end using random function and delete elements in the front and store it in a file** |
| #include <stdio.h>  #include <stdlib.h>  // Define the structure for the linked list node  struct Node {  int data;  struct Node\* next;  };  // Function to insert a new node with data at the end of the linked list  void insertAtEnd(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* last = \*head\_ref;  newNode->data = newData;  newNode->next = NULL;  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  // Function to insert a new node with data at the front of the linked list  void insertAtFront(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = newData;  newNode->next = \*head\_ref;  \*head\_ref = newNode;  }  // Function to display the contents of the linked list  void displayList(struct Node\* node) {  while (node != NULL) {  printf("%d ", node->data);  node = node->next;  }  printf("\n");  }  // Function to write the contents of the linked list to a file  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file for writing.\n");  return;  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  // Function to read the contents of the linked list from a file  void readFromFile(struct Node\*\* head\_ref, const char\* filename) {  FILE\* fp = fopen(filename, "r");  if (fp == NULL) {  printf("Error opening file for reading.\n");  return;  }  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(head\_ref, data);  }  fclose(fp);  }  // Function to insert a new node with data at a specified position in the linked list  void insertAtPosition(struct Node\*\* head\_ref, int newData, int position) {  int i;  if (position <= 0) {  printf("Invalid position.\n");  return;  }  if (position == 1) {  insertAtFront(head\_ref, newData);  return;  }  struct Node\* current = \*head\_ref;  for ( i = 1; i < position - 1 && current != NULL; i++) {  current = current->next;  }  if (current == NULL) {  printf("Position exceeds the length of the list.\n");  return;  }  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = newData;  newNode->next = current->next;  current->next = newNode;  }  int main() {  struct Node\* head = NULL;  const char\* filename = "linked\_list.txt";  int position,i;  // Insert elements at the end and store in a file  for ( i = 0; i < 5; i++) {  insertAtEnd(&head, rand() % 100); // Insert random numbers  }  printf("Initial linked list:\n");  displayList(head);  writeToFile(head, filename);  printf("Data written to file '%s'.\n", filename);  // Read from file, insert elements at the specified position, and store in another file  struct Node\* new\_head = NULL;  const char\* new\_filename = "updated\_list.txt";  readFromFile(&new\_head, filename);  printf("Linked list after reading from file:\n");  displayList(new\_head);  printf("Enter the position to insert elements at the front: ");  scanf("%d", &position);  insertAtPosition(&new\_head, rand() % 100, position);  printf("Linked list after inserting elements at position %d:\n", position);  displayList(new\_head);  // Write the updated list to another file  writeToFile(new\_head, new\_filename);  printf("Updated data written to file '%s'.\n", new\_filename);  return 0;  } |
|  |
|  |
|  |
| **5.using cll insert at end using random function and store it in a file** |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtEnd(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* last = \*head\_ref;  newNode->data = newData;  newNode->next = newNode; // Circular singly linked list  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  while (last->next != \*head\_ref) {  last = last->next;  }  last->next = newNode;  newNode->next = \*head\_ref;  }  void displayList(struct Node\* head) {  struct Node\* temp = head;  if (head != NULL) {  do {  printf("%d\n", temp->data);  temp = temp->next;  } while (temp != head);  }  }  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  struct Node\* temp = head;  if (head != NULL) {  do {  fprintf(fp, "%d ", temp->data);  temp = temp->next;  } while (temp != head);  }  fclose(fp);  }  void readFromFile(struct Node\*\* head\_ref, const char\* filename) {  FILE\* fp = fopen(filename, "r");  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(head\_ref, data);  }  fclose(fp);  }  int main() {  int i;  struct Node\* head = NULL;  // Seed the random number generator  srand(time(NULL));  // Generate and insert random numbers  for ( i = 0; i < 5; ++i) {  insertAtEnd(&head, rand() % 100);  }  printf("Initial circular linked list:\n");  displayList(head);  writeToFile(head, "circular\_linked\_list.txt");  printf("Data written to file.\n");  struct Node\* new\_head = NULL;  readFromFile(&new\_head, "circular\_linked\_list.txt");  printf("Linked list after reading from file:\n");  displayList(new\_head);  writeToFile(new\_head, "updated\_circular\_linked\_list.txt");  printf("Updated list with two elements inserted at front written to file.\n");  return 0;  } |
|  |
|  |
|  |
| **6.using cll insert at end using random function and search a given element** |
| #include <stdlib.h>  #include <time.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtEnd(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* last = \*head\_ref;  newNode->data = newData;  newNode->next = newNode; // Circular singly linked list  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  while (last->next != \*head\_ref) {  last = last->next;  }  last->next = newNode;  newNode->next = \*head\_ref;  }  void displayList(struct Node\* head) {  struct Node\* temp = head;  if (head != NULL) {  do {  printf("%d\n", temp->data);  temp = temp->next;  } while (temp != head);  }  }  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  struct Node\* temp = head;  if (head != NULL) {  do {  fprintf(fp, "%d ", temp->data);  temp = temp->next;  } while (temp != head);  }  fclose(fp);  }  void readFromFile(struct Node\*\* head\_ref, const char\* filename) {  FILE\* fp = fopen(filename, "r");  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(head\_ref, data);  }  fclose(fp);  }  struct Node\* search(struct Node\* head, int key) {  struct Node\* temp = head;  if (head == NULL)  return NULL;  do {  if (temp->data == key)  return temp;  temp = temp->next;  } while (temp != head);  return NULL;  }  int main() {  int i;  struct Node\* head = NULL;  // Seed the random number generator  srand(time(NULL));  // Generate random numbers and insert into the list  for ( i = 0; i < 5; ++i) {  int newData = rand() % 100; // Generate random number between 0 and 99  insertAtEnd(&head, newData);  }  printf("Initial circular linked list:\n");  displayList(head);  writeToFile(head, "circular\_linked\_list.txt");  printf("Data written to file.\n");  struct Node\* new\_head = NULL;  readFromFile(&new\_head, "circular\_linked\_list.txt");  printf("Linked list after reading from file:\n");  displayList(new\_head);  // Ask the user for the element to search  int searchKey;  printf("Enter the element to search: ");  scanf("%d", &searchKey);  // Search for the element in the list  struct Node\* foundNode = search(new\_head, searchKey);  if (foundNode != NULL)  printf("Element %d found in the list.\n", searchKey);  else  printf("Element %d not found in the list.\n", searchKey);  FILE\* searchResultFile = fopen("search\_result.txt", "w");  if (foundNode != NULL)  fprintf(searchResultFile, "Element %d found in the list.\n", searchKey);  else  fprintf(searchResultFile, "Element %d not found in the list.\n", searchKey);  fclose(searchResultFile);  return 0;  } |
|  |
| 1. Find the nth node in a linked list |
| #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  void insertAtEnd(struct Node\*\* head\_ref, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed\n");  return;  }  newNode->data = newData;  newNode->next = NULL;  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  struct Node\* last = \*head\_ref;  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  int getNth(struct Node\* head, int index) {  struct Node\* current = head;  int count = 0;  while (current != NULL) {  if (count == index) {  return current->data;  }  count++;  current = current->next;  }  printf("Index out of bounds.\n");  return -1;  }  void writeToFile(struct Node\* head, int index) {  FILE\* fp = fopen("linked\_list.txt", "w");  if (fp == NULL) {  printf("Failed to open file\n");  return;  }  int count = 0;  while (head != NULL) {  if (count == index) {  fprintf(fp, "%d ", head->data);  fclose(fp);  return;  }  count++;  head = head->next;  }  printf("Index out of bounds.\n");  }  void readFromFile(struct Node\*\* head\_ref) {  FILE\* fp = fopen("linked\_list.txt", "r");  if (fp == NULL) {  printf("Failed to open file\n");  return;  }  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(head\_ref, data);  }  fclose(fp);  }  void displayList(struct Node\* head) {  while (head != NULL) {  printf("%d ", head->data);  head = head->next;  }  printf("\n");  }  int main() {  struct Node\* head = NULL;  int i;  for (i = 0; i < 5; i++) {  insertAtEnd(&head, rand() % 100); // Insert random numbers  }  printf("Linked list: ");  displayList(head);  int index = 2; // Get the element at index 2  printf("Element at index %d is: %d\n", index, getNth(head, index));  writeToFile(head, index);  printf("Element at index %d written to file.\n", index);  printf("Reading from file...\n");  struct Node\* new\_head = NULL;  readFromFile(&new\_head);  printf("Linked list after reading from file: ");  displayList(new\_head);  return 0;  } |
|  |
|  |
| 8.reverse a list using doubly linked list |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  // Structure for a node in doubly linked list  struct Node {  int data;  struct Node\* prev;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed\n");  exit(1);  }  newNode->data = data;  newNode->prev = NULL;  newNode->next = NULL;  return newNode;  }  // Function to insert a new node at the end of the list  void insertAtEnd(struct Node\*\* head\_ref, int data) {  struct Node\* newNode = createNode(data);  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  return;  }  struct Node\* current = \*head\_ref;  while (current->next != NULL) {  current = current->next;  }  current->next = newNode;  newNode->prev = current;  }  // Function to display the doubly linked list  void displayList(struct Node\* head) {  while (head != NULL) {  printf("%d \n", head->data);  head = head->next;  }  }  // Function to write the doubly linked list to a file  void writeToFile(struct Node\* head, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file for writing\n");  return;  }  while (head != NULL) {  fprintf(fp, "%d ", head->data);  head = head->next;  }  fclose(fp);  }  // Function to read the doubly linked list from a file  struct Node\* readFromFile(const char\* filename) {  struct Node\* head = NULL;  FILE\* fp = fopen(filename, "r");  if (fp == NULL) {  printf("Error opening file for reading\n");  return NULL;  }  int data;  while (fscanf(fp, "%d", &data) != EOF) {  insertAtEnd(&head, data);  }  fclose(fp);  return head;  }  // Function to reverse the doubly linked list  void reverseList(struct Node\*\* head\_ref) {  struct Node\* current = \*head\_ref;  struct Node\* temp = NULL;  while (current != NULL) {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  if (temp != NULL) {  \*head\_ref = temp->prev;  }  }  int main() {  struct Node\* head = NULL;  int i;  // Seed random number generator  srand(time(NULL));  // Generate random numbers and insert into doubly linked list  for ( i = 0; i < 5; i++) {  insertAtEnd(&head, rand() % 100);  }  // Display the doubly linked list  printf("Doubly linked list:\n");  displayList(head);  // Write the doubly linked list to a file  writeToFile(head, "linked\_list.txt");  printf("Doubly linked list written to file.\n");  // Reverse the doubly linked list  reverseList(&head);  printf("Reversed doubly linked list:\n");  displayList(head);  // Write the reversed doubly linked list to another file  writeToFile(head, "updated\_list.txt");  printf("Reversed doubly linked list written to file.\n");  // Free memory allocated for current list  struct Node\* temp = head;  while (head != NULL) {  temp = head;  head = head->next;  free(temp);  }  // Read the doubly linked list from the file  head = readFromFile("reversed\_doubly\_linked\_list.txt");  // Display the doubly linked list read from file  printf("Doubly linked list read from file:\n");  displayList(head);  // Free memory allocated for the list read from file  while (head != NULL) {  temp = head;  head = head->next;  free(temp);  }  return 0;  } |
|  |
|  |
|  |
| 1. Find the middle node in circular doubly linked list |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  // Structure for a node in circular doubly linked list  struct Node {  int data;  struct Node\* prev;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed\n");  exit(1);  }  newNode->data = data;  newNode->prev = NULL;  newNode->next = NULL;  return newNode;  }  // Function to insert a new node at the end of the list  void insertAtEnd(struct Node\*\* head\_ref, int data) {  struct Node\* newNode = createNode(data);  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  (\*head\_ref)->next = newNode;  (\*head\_ref)->prev = newNode;  return;  }  struct Node\* last = (\*head\_ref)->prev;  last->next = newNode;  newNode->prev = last;  newNode->next = \*head\_ref;  (\*head\_ref)->prev = newNode;  }  // Function to display the circular doubly linked list  void displayList(struct Node\* head) {  if (head == NULL) {  printf("Empty List\n");  return;  }  struct Node\* current = head;  do {  printf("%d\n ", current->data);  current = current->next;  } while (current != head);  }  // Function to generate a circular doubly linked list with random data  struct Node\* generateRandomList(int size) {  int i;  struct Node\* head = NULL;  srand(time(NULL)); // Seed the random number generator  for ( i = 0; i < size; i++) {  int randomData = rand() % 100; // Generate random data (0 to 99)  insertAtEnd(&head, randomData);  }  return head;  }  // Function to find the middle node of the circular doubly linked list  struct Node\* findMiddle(struct Node\* head) {  if (head == NULL) {  printf("Empty List\n");  return NULL;  }  struct Node\* slow = head;  struct Node\* fast = head;  while (fast->next != head && fast->next->next != head) {  slow = slow->next;  fast = fast->next->next;  }  return slow;  }  // Function to store the middle node's data in a file  void storeMiddleToFile(struct Node\* middleNode, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file\n");  exit(1);  }  fprintf(fp, "%d", middleNode->data);  fclose(fp);  }  int main() {  // Generate a circular doubly linked list with random data  struct Node\* head = generateRandomList(5); // Generate 5 random elements  // Display the circular doubly linked list  printf("Circular Doubly linked list with random data:\n");  displayList(head);  // Find the middle node of the list  struct Node\* middleNode = findMiddle(head);  if (middleNode != NULL) {  printf("Middle node has data: %d\n", middleNode->data);  // Store the middle node's data in a file  storeMiddleToFile(middleNode, "updated\_list.txt");  printf("Middle node's data stored in updated\_list.txt\n");  }  return 0;  } |
|  |
|  |
|  |
| 10.find the largest number in the list |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  // Structure for a node in the circular doubly linked list  struct Node {  int data;  struct Node\* prev;  struct Node\* next;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed\n");  exit(1);  }  newNode->data = data;  newNode->prev = NULL;  newNode->next = NULL;  return newNode;  }  // Function to insert a new node at the end of the list  void insertAtEnd(struct Node\*\* head\_ref, int data) {  struct Node\* newNode = createNode(data);  if (\*head\_ref == NULL) {  \*head\_ref = newNode;  (\*head\_ref)->next = newNode;  (\*head\_ref)->prev = newNode;  } else {  struct Node\* last = (\*head\_ref)->prev;  last->next = newNode;  newNode->prev = last;  newNode->next = \*head\_ref;  (\*head\_ref)->prev = newNode;  }  }  // Function to display the circular doubly linked list  void displayList(struct Node\* head) {  if (head == NULL) {  printf("List is empty\n");  return;  }  struct Node\* current = head;  do {  printf("%d ", current->data);  current = current->next;  } while (current != head);  printf("\n");  }  // Function to find the largest number in the circular doubly linked list  int findLargest(struct Node\* head) {  if (head == NULL)  return -1; // Assuming -1 represents an error or invalid value  int max = head->data;  struct Node\* current = head->next;  while (current != head) {  if (current->data > max) {  max = current->data;  }  current = current->next;  }  return max;  }  // Function to write the largest number in the list to a file  void writeLargestToFile(int largest, const char\* filename) {  FILE\* fp = fopen(filename, "w");  if (fp == NULL) {  printf("Error opening file\n");  exit(1);  }  fprintf(fp, "%d\n", largest); // Write the largest number followed by a newline  fclose(fp);  }  // Function to generate a random number between min and max  int getRandom(int min, int max) {  return rand() % (max - min + 1) + min;  }  int main() {  int i;  struct Node\* head = NULL;  // Seed the random number generator  srand(time(NULL));  // Generate a circular doubly linked list with random numbers  for ( i = 0; i < 5; i++) {  int randomNum = getRandom(1, 100); // Generate random number between 1 and 100  insertAtEnd(&head, randomNum);  }  // Display the circular doubly linked list  printf("Circular Doubly Linked List: ");  displayList(head);  // Find the largest number in the list  int largest = findLargest(head);  if (largest != -1) {  printf("Largest number in the list: %d\n", largest);  // Write the largest number to a file  writeLargestToFile(largest, "largest\_number.txt");  printf("Largest number written to file\n");  } else {  printf("List is empty\n");  }  return 0;  } |
|  |
|  |
| 11.To push the elements into stack |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to generate a random number between min and max  int getRandom(int min, int max) {  return rand() % (max - min + 1) + min;  }  // Function to display the elements in the stack  void display(struct Stack\* stack) {  int i;  if (isEmpty(stack)) {  printf("Stack is empty\n");  } else {  printf("Stack elements: ");  for ( i = 0; i <= stack->top; i++) {  printf("%d ", stack->items[i]);  }  printf("\n");  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack) {  int i;  FILE\* file = fopen(FILENAME, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 5; i++) {  int randomNum = getRandom(1, 100); // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Save stack elements to file  saveToFile(&stack);  // Display the stack  display(&stack);  return 0;  } |
|  |
|  |
| 12.store the alternate stack elements in a file |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define ALTERNATE\_FILENAME "alternate\_elements.txt" // Name of the file to store alternate elements  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to pop alternate elements from the stack and store them in a file  void popAndStoreAlternateToFile(struct Stack\* stack, const char\* filename) {  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  // Flag to toggle between popping and skipping elements  int popFlag = 1;  while (!isEmpty(stack)) {  if (popFlag) {  int element = pop(stack); // Pop an element from the stack  fprintf(file, "%d\n", element); // Write the popped element to the file  } else {  pop(stack); // Skip the element  }  popFlag = !popFlag; // Toggle the flag  }  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store alternate elements in a separate file  popAndStoreAlternateToFile(&stack, ALTERNATE\_FILENAME);  printf("Alternate elements popped from the stack and stored in the file: %s\n", ALTERNATE\_FILENAME);  return 0;  } |
|  |
|  |
| 13.file the maximum element in the stack |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define MAX\_ELEMENT\_FILE "alternate\_elements.txt" // Name of the file to store the maximum element  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to find the maximum element in the stack  int findMaxElement(struct Stack\* stack) {  int i;  int maxElement = stack->items[0]; // Assume first element as maximum  for ( i = 1; i <= stack->top; i++) {  if (stack->items[i] > maxElement) {  maxElement = stack->items[i]; // Update maxElement if current element is greater  }  }  return maxElement;  }  // Function to store the maximum element in a file  void storeMaxElementToFile(struct Stack\* stack, const char\* filename) {  int maxElement = findMaxElement(stack); // Find the maximum element  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  fprintf(file, "%d\n", maxElement); // Write the maximum element to the file  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Store maximum element in a separate file  storeMaxElementToFile(&stack, MAX\_ELEMENT\_FILE);  printf("Maximum element in the stack stored in the file: %s\n", MAX\_ELEMENT\_FILE);  return 0;  } |
|  |
|  |
|  |
| 14.find the minimum element in the stack |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define MIN\_ELEMENT\_FILE "alternate\_elements.txt" // Name of the file to store the minimum element  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to find the minimum element in the stack  int findMinElement(struct Stack\* stack) {  int i;  int minElement = stack->items[0]; // Assume first element as minimum  for ( i = 1; i <= stack->top; i++) {  if (stack->items[i] < minElement) {  minElement = stack->items[i]; // Update minElement if current element is smaller  }  }  printf("the minimum element is %d\n",minElement);  return minElement;  }  // Function to store the minimum element in a file  void storeMinElementToFile(struct Stack\* stack, const char\* filename) {  int minElement = findMinElement(stack); // Find the minimum element  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  fprintf(file, "%d\n", minElement); // Write the minimum element to the file  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Store minimum element in a separate file  storeMinElementToFile(&stack, MIN\_ELEMENT\_FILE);  printf("Minimum element in the stack stored in the file: %s\n", MIN\_ELEMENT\_FILE);  return 0;  } |
|  |
|  |
|  |
| 15.reverse a stack |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define REVERSED\_FILE "reversed\_stack.txt" // Name of the file to store the reversed stack elements  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to reverse the elements of the stack  void reverseStack(struct Stack\* stack) {  struct Stack tempStack;  initializeStack(&tempStack);  // Pop all elements from the original stack and push them onto the temporary stack  while (!isEmpty(stack)) {  push(&tempStack, pop(stack));  }  // Pop all elements from the temporary stack and push them back onto the original stack  while (!isEmpty(&tempStack)) {  push(stack, pop(&tempStack));  }  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Reverse the elements of the stack  reverseStack(&stack);  printf("Stack elements reversed\n");  // Store the reversed stack elements in a file  saveToFile(&stack, REVERSED\_FILE);  printf("Reversed stack elements stored in the file: %s\n", REVERSED\_FILE);  return 0;  } |
|  |
|  |
| 16.pop even elements from the stack |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define EVEN\_ELEMENTS\_FILE "alternate\_elements.txt" // Name of the file to store even elements  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to store even elements of the stack in a file  void popEvenElementsToFile(struct Stack\* stack, const char\* filename) {  FILE\* file = fopen(filename, "w");  printf("even elements are\n");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  while (!isEmpty(stack)) {  int element = pop(stack);  if (element % 2 == 0) {  fprintf(file, "%d\n", element);  }  printf("%d\n",element);  }  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Pop even elements from the stack and store them in a file  popEvenElementsToFile(&stack, EVEN\_ELEMENTS\_FILE);  printf("Even elements from the stack stored in the file: %s\n", EVEN\_ELEMENTS\_FILE);  return 0;  } |
|  |
|  |
|  |
| 17.find odd element in the stack |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define ODD\_ELEMENTS\_FILE "alternate\_elements.txt" // Name of the file to store odd elements  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to store odd elements of the stack in a file  void popOddElementsToFile(struct Stack\* stack, const char\* filename) {  FILE\* file = fopen(filename, "w");  printf("the odd elements are\n");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  while (!isEmpty(stack)) {  int element = pop(stack);  if (element % 2 != 0) {  fprintf(file, "%d\n", element);  printf("%d\n",element);  }  }  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Pop odd elements from the stack and store them in a file  popOddElementsToFile(&stack, ODD\_ELEMENTS\_FILE);  printf("Odd elements from the stack stored in the file: %s\n", ODD\_ELEMENTS\_FILE);  return 0;  } |
|  |
|  |
|  |
|  |
| 18. find the prime elements in the stack |
|  |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <math.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define PRIME\_ELEMENTS\_FILE "alternate\_elements.txt" // Name of the file to store prime elements  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to check if a number is prime  int isPrime(int num) {  int i;  if (num <= 1) {  return 0;  }  for ( i = 2; i <= sqrt(num); i++) {  if (num % i == 0) {  return 0;  }  }  return 1;  }  // Function to store prime elements of the stack in a file  void popPrimeElementsToFile(struct Stack\* stack, const char\* filename) {  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  printf("the prime elements in the stack are\n");  while (!isEmpty(stack)) {  int element = pop(stack);  if (isPrime(element)) {  fprintf(file, "%d\n", element);  printf("%d\n",element);  }  }  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Pop prime elements from the stack and store them in a file  popPrimeElementsToFile(&stack, PRIME\_ELEMENTS\_FILE);  printf("Prime elements from the stack stored in the file: %s\n", PRIME\_ELEMENTS\_FILE);  return 0;  } |
|  |
|  |
|  |
| 19.search an element I the file |
| #includeude <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <math.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define search\_element\_FILE "alternate\_elements.txt" // Name of the file to store prime elements  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to pop an element from the stack  int pop(struct Stack\* stack) {  if (isEmpty(stack)) {  printf("Stack Underflow: Cannot pop element, stack is empty\n");  return -1; // Return -1 indicating failure  } else {  int popped = stack->items[stack->top]; // Get the top element  stack->top--; // Decrement top  return popped; // Return the popped element  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to search for an element in the stack  int searchElement(struct Stack\* stack, int key) {  int i;  for ( i = stack->top; i >= 0; i--) {  if (stack->items[i] == key) {  return i; // Return index of the element if found  }  }  return -1; // Return -1 if element not found  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Search for an element in the stack  int key = 42; // Element to search for  int index = searchElement(&stack, key);  if (index != -1) {  printf("Element %d found at index %d in the stack\n", key, index);  } else {  printf("Element %d not found in the stack\n", key);  }  return 0;  } |
|  |
|  |
| 20.find the element with the staring digit 1 and store the index |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100 // Maximum size of the stack  #define FILENAME "stack\_elements.txt" // Name of the file to store stack elements  #define FIRST\_DIGIT\_ONE\_FILE "alternate\_elements.txt" // Name of the file to store elements with first digit 1  // Structure to represent a stack  struct Stack {  int items[MAX\_SIZE];  int top;  };  // Function to initialize the stack  void initializeStack(struct Stack\* stack) {  stack->top = -1; // Initialize top to -1 to indicate an empty stack  }  // Function to check if the stack is empty  int isEmpty(struct Stack\* stack) {  return stack->top == -1; // Returns 1 if the stack is empty, otherwise 0  }  // Function to check if the stack is full  int isFull(struct Stack\* stack) {  return stack->top == MAX\_SIZE - 1; // Returns 1 if the stack is full, otherwise 0  }  // Function to push an element onto the stack  void push(struct Stack\* stack, int value) {  if (isFull(stack)) {  printf("Stack Overflow: Cannot push element, stack is full\n");  } else {  stack->top++; // Increment top  stack->items[stack->top] = value; // Store the value at the top of the stack  printf("%d pushed to stack\n", value);  }  }  // Function to save stack elements to a file  void saveToFile(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = 0; i <= stack->top; i++) {  fprintf(file, "%d\n", stack->items[i]);  }  fclose(file);  }  // Function to check if the first digit of a number is 1  int isFirstDigitOne(int num) {  int firstDigit;  while (num >= 10) {  num /= 10;  }  firstDigit = num;  return firstDigit == 1;  }  // Function to search for elements where the first digit is 1 in the stack and store their indices and values in a file  void searchAndStoreFirstDigitOne(struct Stack\* stack, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  printf("the elements with digit 1 are at the following position\n");  for ( i = 0; i <= stack->top; i++) {  if (isFirstDigitOne(stack->items[i])) {  fprintf(file, "Index: %d, Value: %d\n", i, stack->items[i]);  printf("%d\n",i+1);  }  }  fclose(file);  }  int main() {  int i;  struct Stack stack;  initializeStack(&stack);  // Seed the random number generator  srand(time(NULL));  // Generate a stack with random elements  for ( i = 0; i < 10; i++) { // Generating more elements for demonstration purposes  int randomNum = rand() % 100 + 1; // Generate random number between 1 and 100  push(&stack, randomNum);  }  // Store stack elements in a file  saveToFile(&stack, FILENAME);  printf("Stack elements stored in the file: %s\n", FILENAME);  // Search for elements where the first digit is 1 and store their indices and values in a file  searchAndStoreFirstDigitOne(&stack, FIRST\_DIGIT\_ONE\_FILE);  printf("Elements where the first digit is 1 stored in the file: %s\n", FIRST\_DIGIT\_ONE\_FILE);  return 0;  } |
|  |
|  |
|  |
| 21. generate queues and store it in a file |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100  #define FILENAME "queue\_elements.txt"  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  }  // Function to display the elements of the queue  void display(struct Queue\* queue) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return;  }  int i = queue->front;  printf("Queue elements: ");  do {  printf("%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  } while (i != (queue->rear + 1) % MAX\_SIZE);  printf("\n");  }  // Function to write queue elements to a file  void writeToFile(struct Queue\* queue, const char\* filename) {  int i;  FILE\* file = fopen(filename, "w");  if (file == NULL) {  printf("Error opening file for writing\n");  return;  }  for ( i = queue->front; i != (queue->rear + 1) % MAX\_SIZE; i = (i + 1) % MAX\_SIZE) {  fprintf(file, "%d\n", queue->items[i]);  }  fclose(file);  }  // Function to generate random elements and enqueue them into the queue  void generateAndEnqueueRandomElements(struct Queue\* queue, int count) {  int i;  srand(time(NULL)); // Seed the random number generator  for ( i = 0; i < count; i++) {  int randomNumber = rand() % 100; // Generate a random number between 0 and 99  enqueue(queue, randomNumber);  }  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Generate 10 random elements and enqueue them into the queue  generateAndEnqueueRandomElements(&queue, 10);  // Write queue elements to a file  writeToFile(&queue, FILENAME);  printf("Queue elements stored in the file: %s\n", FILENAME);  // Display elements  printf("Initial queue:\n");  display(&queue);  return 0;  } |
|  |
|  |
| 21.reverse a queue |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  }  // Function to reverse an array  void reverseArray(int arr[], int size) {  int left = 0;  int right = size - 1;  while (left < right) {  // Swap elements at left and right indices  int temp = arr[left];  arr[left] = arr[right];  arr[right] = temp;  // Move to the next pair of elements  left++;  right--;  }  }  // Function to reverse the elements of a queue  void reverseQueue(struct Queue\* queue) {  int size = (queue->rear - queue->front + MAX\_SIZE + 1) % MAX\_SIZE;  int arr[MAX\_SIZE];  int i = 0, j = queue->front;  while (i < size) {  arr[i++] = queue->items[j];  j = (j + 1) % MAX\_SIZE;  }  reverseArray(arr, size);  i = 0, j = queue->front;  while (i < size) {  queue->items[j] = arr[i++];  j = (j + 1) % MAX\_SIZE;  }  }  // Function to display the elements of the queue  void display(struct Queue\* queue) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return;  }  printf("Queue elements: ");  int i = queue->front;  while (i != queue->rear) {  printf("%d ", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", queue->items[queue->rear]);  }  int main() {  struct Queue queue;  initializeQueue(&queue);  // Seed the random number generator  srand(time(NULL));  // Enqueue random elements into the queue and store them in a file  const char\* inputFilename = "queue\_elements.txt";  FILE\* inputFile = fopen(inputFilename, "w");  if (inputFile == NULL) {  printf("Error opening file for writing\n");  return 1;  }  int i;  for ( i = 0; i < 10; i++) {  int randomNumber = rand() % 100;  fprintf(inputFile, "%d ", randomNumber);  enqueue(&queue, randomNumber);  }  fclose(inputFile);  printf("Queue elements stored in the file: %s\n", inputFilename);  // Display the original queue  printf("Original queue:\n");  display(&queue);  // Store the reversed queue in another file  reverseQueue(&queue);  const char\* outputFilename = "output.txt";  FILE\* outputFile = fopen(outputFilename, "w");  if (outputFile == NULL) {  printf("Error opening file for writing\n");  return 1;  }  i = queue.front;  while (i != queue.rear) {  fprintf(outputFile, "%d ", queue.items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(outputFile, "%d", queue.items[queue.rear]);  fclose(outputFile);  printf("Reversed queue elements stored in the file: %s\n", outputFilename);  return 0;  } |
|  |
|  |
|  |
| 22. find the odd number sin the stack |
| Code:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  FILE\* file; // File to store the queue elements  FILE\* oddFile; // File to store odd elements  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->file = fopen("queue\_elements.txt", "w"); // Open file for writing  if (queue->file == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->oddFile = fopen("odd\_elements.txt", "w"); // Open file for writing odd elements  if (queue->oddFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Print the value to the console  printf("queue: %d\n", value);  // Write the value to the file  fprintf(queue->file, "%d\n", value);  // Check if the value is odd and write to odd elements file  if (value % 2 != 0) {  fprintf(queue->oddFile, "%d\n", value);  }  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  }  // Close the files after enqueueing  fclose(queue.file);  fclose(queue.oddFile);  printf("Queue elements stored in file: queue\_elements.txt\n");  printf("Odd elements stored in file: odd\_elements.txt\n");  return 0;  } |
|  |
|  |
|  |
| 23.find even number in the queue |
| Code:  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  FILE\* file; // File to store the queue elements  FILE\* evenFile; // File to store even elements  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->file = fopen("queue\_elements.txt", "w"); // Open file for writing  if (queue->file == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->evenFile = fopen("even\_elements.txt", "w"); // Open file for writing even elements  if (queue->evenFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Print the value to the console if even  if (value % 2 == 0) {  printf("Enqueued even number: %d\n", value);  // Write the even value to the file  fprintf(queue->evenFile, "%d\n", value);  }  // Write all values to the file  fprintf(queue->file, "%d\n", value);  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  }  // Close the files after enqueueing  fclose(queue.file);  fclose(queue.evenFile);  printf("Queue elements stored in file: queue\_elements.txt\n");  printf("Even elements stored in file: even\_elements.txt\n");  return 0;  } |
|  |
|  |
|  |
| 24.find the prime number in the queue |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  FILE\* file; // File to store the queue elements  FILE\* primeFile; // File to store prime elements  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->file = fopen("queue\_elements.txt", "w"); // Open file for writing  if (queue->file == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->primeFile = fopen("prime\_elements.txt", "w"); // Open file for writing prime elements  if (queue->primeFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Print the value to the console if prime  if (isPrime(value)) {  printf("Enqueued prime number: %d\n", value);  // Write the prime value to the file  fprintf(queue->primeFile, "%d\n", value);  }  // Write all values to the file  fprintf(queue->file, "%d\n", value);  }  // Function to check if a number is prime  isPrime(int num) {  int i;  if (num <= 1) {  return false;  }  for ( i = 2; i \* i <= num; i++) {  if (num % i == 0) {  return false;  }  }  return true;  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  }  // Close the files after enqueueing  fclose(queue.file);  fclose(queue.primeFile);  printf("Queue elements stored in file: queue\_elements.txt\n");  printf("Prime elements stored in file: prime\_elements.txt\n");  return 0;  } |
|  |
|  |
|  |
| 25.search an element in a queue |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  FILE\* file; // File to store the queue elements  FILE\* evenFile; // File to store even elements  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->file = fopen("queue\_elements.txt", "w"); // Open file for writing  if (queue->file == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->evenFile = fopen("even\_elements.txt", "w"); // Open file for writing even elements  if (queue->evenFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Write all values to the file  fprintf(queue->file, "%d\n", value);  }  // Function to search for an element in the queue  int searchElement(struct Queue\* queue, int element) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return 0;  }  int i = queue->front;  while (i != queue->rear) {  if (queue->items[i] == element) {  printf("Element %d found at index %d\n", element, i);  return 1;  }  i = (i + 1) % MAX\_SIZE;  }  if (queue->items[queue->rear] == element) {  printf("Element %d found at index %d\n", element, queue->rear);  return 1;  }  printf("Element %d not found in the queue\n", element);  return 0;  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  printf("%d\n",randomNum);  }  // Search for an element in the queue  int searchValue = 50; // Element to search  searchElement(&queue, searchValue);  // Close the files after enqueueing  fclose(queue.file);  fclose(queue.evenFile);  printf("Queue elements stored in file: queue\_elements.txt\n");  printf("Even elements stored in file: even\_elements.txt\n");  return 0;  } |
|  |
|  |
| 26.apply dequeue operation in queues |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  FILE\* file; // File to store the queue elements  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->file = fopen("queue\_elements.txt", "w"); // Open file for writing  if (queue->file == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Write all values to the file  fprintf(queue->file, "%d\n", value);  }  // Function to remove an element from the front of the queue  int dequeue(struct Queue\* queue) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return -1;  }  int frontItem = queue->items[queue->front];  if (queue->front == queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front = (queue->front + 1) % MAX\_SIZE;  }  return frontItem;  }  // Function to delete a specific element from the queue  void delete(struct Queue\* queue, int value) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return;  }  int found = 0;  int i = queue->front;  while (i != queue->rear) {  if (queue->items[i] == value) {  found = 1;  break;  }  i = (i + 1) % MAX\_SIZE;  }  if (queue->items[queue->rear] == value) {  found = 1;  }  if (found) {  printf("Element %d deleted from the queue\n", value);  // Shift elements to remove the found element  while (i != queue->rear) {  queue->items[i] = queue->items[(i + 1) % MAX\_SIZE];  i = (i + 1) % MAX\_SIZE;  }  queue->rear = (queue->rear - 1 + MAX\_SIZE) % MAX\_SIZE; // Update rear  } else {  printf("Element %d not found in the queue\n", value);  }  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  }  // Dequeue elements from the queue  printf("Dequeued elements: ");  while (!isEmpty(&queue)) {  int dequeuedItem = dequeue(&queue);  printf("%d ", dequeuedItem);  }  printf("\n");  // Close the file after enqueueing  fclose(queue.file);  printf("Queue elements stored in file: queue\_elements.txt\n");  // Delete an element from the queue  delete(&queue, 50);  return 0;  } |
|  |
|  |
| 27.delete alternate member of the queue |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  FILE\* originalFile; // File to store the original queue elements  FILE\* finalFile; // File to store the final queue elements  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->originalFile = fopen("original\_queue\_elements.txt", "w"); // Open file for writing original elements  if (queue->originalFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->finalFile = fopen("final\_queue\_elements.txt", "w"); // Open file for writing final elements  if (queue->finalFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Write the value to the original queue file  fprintf(queue->originalFile, "%d\n", value);  }  // Function to remove an element from the front of the queue  int dequeue(struct Queue\* queue) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return -1;  }  int frontItem = queue->items[queue->front];  if (queue->front == queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front = (queue->front + 1) % MAX\_SIZE;  }  return frontItem;  }  // Function to delete alternate elements from the queue  void deleteAlternate(struct Queue\* queue) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return;  }  int count = 1; // Start count from 1 to delete alternate elements  int i = queue->front;  while (i != queue->rear) {  if (count % 2 == 0) {  // Skip deletion for alternate elements  printf("Element %d skipped\n", queue->items[i]);  } else {  // Delete current element  printf("Element %d deleted\n", queue->items[i]);  queue->front = (queue->front + 1) % MAX\_SIZE;  }  count++;  i = (i + 1) % MAX\_SIZE;  }  // Delete last element if queue size is odd  if ((count % 2) == 0) {  printf("Element %d deleted\n", queue->items[i]);  queue->front = (queue->front + 1) % MAX\_SIZE;  }  queue->rear = (queue->rear - count + MAX\_SIZE) % MAX\_SIZE; // Update rear  }  // Function to store the final queue elements in a file  void storeFinalQueueToFile(struct Queue\* queue) {  int i = queue->front;  while (i != queue->rear) {  fprintf(queue->finalFile, "%d\n", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(queue->finalFile, "%d\n", queue->items[queue->rear]);  fclose(queue->finalFile);  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  }  // Display original queue  printf("Original queue:\n");  i = queue.front;  while (i != queue.rear) {  printf("%d ", queue.items[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", queue.items[queue.rear]);  // Store the original queue elements in a file  storeFinalQueueToFile(&queue);  printf("Original queue elements stored in file: original\_queue\_elements.txt\n");  // Delete alternate elements from the queue  deleteAlternate(&queue);  // Display modified queue  printf("Modified queue:\n");  i = queue.front;  while (i != queue.rear) {  printf("%d ", queue.items[i]);  i = (i + 1) % MAX\_SIZE;  }  printf("%d\n", queue.items[queue.rear]);  // Store the final queue elements in a file  storeFinalQueueToFile(&queue);  printf("Final queue elements stored in file: final\_queue\_elements.txt\n");  // Close the file after enqueueing  fclose(queue.originalFile);  return 0;  } |
|  |
|  |
|  |
| 28.find the max number in the queue |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  int max; // Maximum term in the queue  FILE\* originalFile; // File to store the original queue elements  FILE\* finalFile; // File to store the final queue elements  FILE\* maxFile; // File to store the maximum term  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->max = -1;  queue->originalFile = fopen("original\_queue\_elements.txt", "w"); // Open file for writing original elements  if (queue->originalFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->finalFile = fopen("final\_queue\_elements.txt", "w"); // Open file for writing final elements  if (queue->finalFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->maxFile = fopen("max\_queue\_element.txt", "w"); // Open file for writing maximum element  if (queue->maxFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Write the value to the original queue file  fprintf(queue->originalFile, "%d\n", value);  // Update maximum term if needed  if (value > queue->max) {  queue->max = value;  }  }  // Function to store the final queue elements in a file  void storeFinalQueueToFile(struct Queue\* queue) {  int i = queue->front;  while (i != queue->rear) {  fprintf(queue->finalFile, "%d\n", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(queue->finalFile, "%d\n", queue->items[queue->rear]);  fclose(queue->finalFile);  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  printf("%d\n",randomNum);  }  // Print the maximum term  printf("Maximum term in the queue: %d\n", queue.max);  // Store the original queue elements in a file  storeFinalQueueToFile(&queue);  printf("Original queue elements stored in file: original\_queue\_elements.txt\n");  // Store the maximum term in a file  fprintf(queue.maxFile, "%d\n", queue.max);  fclose(queue.maxFile);  printf("Maximum term stored in file: max\_queue\_element.txt\n");  return 0;  } |
|  |
|  |
|  |
| 29.find the maximum element in the queue |
|  |
|  |
|  |
| 30.to sort the queue in assending order |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <limits.h> // Include limits.h for INT\_MAX  #include <stdbool.h> // Include stdbool.h for boolean data type  #include <string.h> // Include string.h for strcpy function  #define MAX\_SIZE 100  // Structure for a queue  struct Queue {  int items[MAX\_SIZE];  int front;  int rear;  int min; // Minimum term in the queue  FILE\* originalFile; // File to store the original queue elements  FILE\* finalFile; // File to store the final queue elements  FILE\* minFile; // File to store the minimum term  };  // Function to initialize the queue  void initializeQueue(struct Queue\* queue) {  queue->front = -1;  queue->rear = -1;  queue->min = INT\_MAX; // Initialize min to maximum possible integer value  queue->originalFile = fopen("original\_queue\_elements.txt", "w"); // Open file for writing original elements  if (queue->originalFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->finalFile = fopen("final\_queue\_elements.txt", "w"); // Open file for writing final elements  if (queue->finalFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  queue->minFile = fopen("min\_queue\_element.txt", "w"); // Open file for writing minimum element  if (queue->minFile == NULL) {  printf("Error opening file for writing\n");  exit(1);  }  }  // Function to check if the queue is empty  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  // Function to check if the queue is full  int isFull(struct Queue\* queue) {  return (queue->rear + 1) % MAX\_SIZE == queue->front;  }  // Function to add an element to the rear of the queue  void enqueue(struct Queue\* queue, int value) {  if (isFull(queue)) {  printf("Queue is full\n");  return;  }  if (isEmpty(queue)) {  queue->front = 0;  }  queue->rear = (queue->rear + 1) % MAX\_SIZE;  queue->items[queue->rear] = value;  // Write the value to the original queue file  fprintf(queue->originalFile, "%d\n", value);  // Update minimum term if needed  if (value < queue->min) {  queue->min = value;  }  }  // Function to dequeue an element from the front of the queue  int dequeue(struct Queue\* queue) {  if (isEmpty(queue)) {  printf("Queue is empty\n");  return -1;  }  int frontItem = queue->items[queue->front];  if (queue->front == queue->rear) {  queue->front = -1;  queue->rear = -1;  } else {  queue->front = (queue->front + 1) % MAX\_SIZE;  }  return frontItem;  }  // Function to sort an array in ascending order  void sortArray(int arr[], int size) {  int i,j;  for ( i = 0; i < size - 1; i++) {  for ( j = i + 1; j < size; j++) {  if (arr[i] > arr[j]) {  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  }  }  // Function to sort the queue in ascending order  void sortQueue(struct Queue\* queue) {  int i;  // Dequeue all elements from the queue and store them in an array  int temp[MAX\_SIZE];  int count = 0;  while (!isEmpty(queue)) {  temp[count++] = dequeue(queue);  }  // Sort the array  sortArray(temp, count);  // Enqueue the sorted elements back into the queue  for ( i = 0; i < count; i++) {  enqueue(queue, temp[i]);  }  }  // Function to store the final queue elements in a file  void storeFinalQueueToFile(struct Queue\* queue) {  int i = queue->front;  while (i != queue->rear) {  fprintf(queue->finalFile, "%d\n", queue->items[i]);  i = (i + 1) % MAX\_SIZE;  }  fprintf(queue->finalFile, "%d\n", queue->items[queue->rear]);  fclose(queue->finalFile);  }  int main() {  int i;  struct Queue queue;  initializeQueue(&queue);  // Generate random elements and enqueue them into the queue  srand(time(NULL));  for ( i = 0; i < 10; i++) { // Generate 10 random elements for demonstration  int randomNum = rand() % 100; // Generate random number between 0 and 99  enqueue(&queue, randomNum);  }  // Print the original queue  printf("Original queue:\n");  for ( i = queue.front; i != queue.rear; i = (i + 1) % MAX\_SIZE) {  printf("%d ", queue.items[i]);  }  printf("%d\n", queue.items[queue.rear]);  // Sort the queue in ascending order  sortQueue(&queue);  // Print the sorted queue  printf("Sorted queue in ascending order:\n");  for ( i = queue.front; i != queue.rear; i = (i + 1) % MAX\_SIZE) {  printf("%d ", queue.items[i]);  }  printf("%d\n", queue.items[queue.rear]);  // Store the original and final queue elements in files  storeFinalQueueToFile(&queue);  printf("Original queue elements stored in file: original\_queue\_elements.txt\n");  printf("Final queue elements stored in file: final\_queue\_elements.txt\n");  return 0;  } |
|  |
|  |