|  |  |
| --- | --- |
| **Name of the student** | **SAYED SAMIULLA KHADRI** |
| **Roll No** | **355** |
| **USN** | **01FE22BEC052** |
| **Div** | **C** |

**Part- A**

|  |
| --- |
| **Variables and Data Types** |
| 1.What is the difference between a variable and a data type in C programming? Provide examples to illustrate. |
| A variable is an identifier which is used to store data. Each variable must be declared with a specific data type before it can be used.  Examples:  Age : Variable of type int to store a person's age.  price: Variable of type float to store the price of an item.  grade: Variable of type char to store a student's grade.  Example for usage  int age=25;  float price =10.5;  char grade=A;  A data type in C programming specifies the type of data that a variable can hold.  Example  integers, floating-point numbers, characters  int: Used to store integer values.  float: Used to store floating-point numbers.  char: Used to store single characters.  Example for usage  int age=25;  float price =10.5;  char grade=A; |
| 2. Explain the concept of data types in C programming. Discuss the different types  of data types available in C. |
| A data type is a classification that specifies the type of data that a variable can hold. Data types determine the size, format, and range of values that can be stored in variables. Using appropriate data types is essential for efficient memory usage and accurate data representation.  Basic data types:   * Integer Types (int, short, long)**:** Used to store whole numbers. The size and range of these types may vary depending on the platform. * Floating-point Types (float, double, long double) Used to store real numbers (numbers with fractional parts). They vary in precision and range. * Character Type (char): Used to store single characters. It can also be used to represent small integer values.   Derived Data Types: These data types are derived from basic data types. They include:   * Array: A collection of elements of the same data type arranged in contiguous memory locations. * Pointer: A variable that stores the memory address of another variable. Pointers allow dynamic memory allocation and manipulation. |
| 3 .How are variables declared and initialized in C programming? Provide  examples of variable declarations with different data types. |
| General syntax to declare a variable  <datatype><variable name>  General syntax to initialize a variable  <variable name>=<data>  Example   1. int a;   a=50;   1. float b;   b=10.56;   1. char ch;   ch=A; |
| 4.Discuss the scope and lifetime of variables in C programming. What are global  and local variables? |
| The scope of a variable is the area in which a variable is accessible. You cannot use a variable beyond its scope. In C programming, the scope of variables is local and global. A lifetime of a variable is the working time of a variable; till its lifetime a variable holds a memory place.  The local and global are two scopes for C variables. The local scope is limited to the code or function in which the variable is declared.  Global scope is the entire program. Global variables can be used anywhere throughout the program. |
| 5.Explain the concept of type casting in C programming. When is type casting  necessary, and how is it performed? |
| Type Casting is basically a process in C in which we change a variable belonging to one data type to another one. In type casting, the compiler automatically changes one data type to another one depending on what we want the program to do  Type casting is mainly used when both the data types in the equation aren’t compatible at all with each other. On the other hand, it is a must that both data types be compatible with each other in the case of type conversion.  #include <stdio.h>  main() {  int total = 17, values = 5;  double average;  average = (double) total / values;  printf(“The average of all the values available with us is : %f\n”, average );  } |
| • **Control Structures:** |
| 6.Describe the purpose and usage of the switch statement in C programming. How  does it differ from the if-else statement? |
| The switch statement in C programming is used to loop through a range of values and execute the same code block for each iteration, minimizing the need for repetitive code structures.  The main difference between the two is that an if/else statement evaluates a condition and executes a block of code if the condition is true, while a switch statement evaluates an expression and executes the code associated with the matching case statement. |
| 7.Explain the concept of nested control structures in C programming. Provide an  example demonstrating nested if-else statements. |
| A nested structure in C is a structure that contains one or more members that are themselves structures. Nested structures are useful for organizing complex data and can help improve a program's readability and maintainability.  Example 1: grading system  #include <stdio.h>  int main () {  int score = 88;  *// Check if score is greater than 90*  if (score > 90) {  printf ("Grade: A\n");  } else {  *// Score is 90 or below, check if score is greater than 80*  if (score > 80) {  *// Score is between 81 and 90, check if score is above 85*  if (score > 85) {  printf ("Grade: A-\n");  } else {  Printf ("Grade: B+\n");  }  } else {  // Score is 80 or below, check if score is above 70  if (score > 70) {  printf ("Grade: C\n");  } else {  Printf ("Grade: D\n");  }  }  }  return 0;  } |
| 8.Discuss the role of the break and continue statements in loop control in C  programming. Provide examples to illustrate their usage. |
| The break statement is used within a loop or switch statement to terminate the loop or exit the switch. It is typically used within conditional statements. The continue statement is used within a loop to skip the rest of the code in the current iteration and begin the next iteration.  Example for break statement  #include <stdio.h>  int main () {  int i;  for (i=0; i<20; i++) {  printf ("%d ",i);  if(i==8) //If this condition is true, break is executed  break;}  printf ("\n Exited the loop after i=%d", i);  return 0;  }  Output:  0 1 2 3 4 5 6 7 8 Exited the loop after i=8  Example for continue statement  #include <stdio.h>  int main () {  int i=1;  for (i=1; i<=10; i++) {  if(i==8) {  continue;}  printf ("%d ", i);  }  return 0;  }  OUTPUT:  1 2 3 4 5 6 7 8 9 10 |
| 9.What are the advantages of using the for loop over the while loop in C  programming? Provide examples comparing the two. |
| A while loop will always evaluate the condition first.  A do/while loop will always execute the code in the do{} block first and then evaluate the condition.  A for loop allows you to initiate a counter variable, a check condition, and a way to increment your counter all in one line.  Example for for loop  for (int x = 0; x < 100; x++) {  //executed until x >= 100  }  Example for while loop  int count = 0;  while (count < 100) {  //do stuff  count++;  }  In this case, there's just more stuff to keep up with and the count++; could get lost in the logic. This could end up being troublesome depending on where count gets incremented, and whether or not it should get incremented before or after the loop's logic. With a for loop, your counter variable is always incremented before the next iteration of the loop, which adds some uniformity to your code. |
| 10.Explain the concept of short-circuit evaluation in C programming. How does it  affect the evaluation of logical expressions in if statements? |
| [Short-circuiting](https://www.geeksforgeeks.org/short-circuiting-in-c-and-linux/) is a programming concept in which the [compiler](https://www.geeksforgeeks.org/introduction-of-compiler-design/) skips the execution or evaluation of some sub-expressions in a logical expression. The compiler stops evaluating the further sub-expressions as soon as the value of the expression is determined.  If the condition is met and the rest of the conditions won't affect the already evaluated result, the expression will short-circuit and return that result.  if (a == b || c == d || e == f) {      // do\_ something  }  In the above expression, If the expression a==b is true, then c==d and  are not evaluated because the expression’s result has already been determined. Similarly, if the logical AND operator(&&) is used instead of logical OR(||)and the expression a==b is false, the compiler will skip evaluating other sub-expressions. |
|  |
| **Operators:** |
| 11.Describe the purpose and usage of the ternary conditional operator (?:) in C  programming. Provide an example demonstrating its usage. |
| We use the ternary operator in C to run one code when the condition is true and another code when the condition is false. For example,  If(age>=18)  {  Printf (“you can vote”);  }  Else  Printf (“cannot vote”);  Here, when the age is greater than or equal to 18,can vote is printed. Otherwise, cannot vote is printed. |
| 12.Discuss the bitwise operators available in C programming. Explain their usage  with suitable examples. |
| Bitwise operators in C programing are tools for performing bit-level operations on integer data. These bit operators, including AND (&), OR (|), XOR (^), and NOT (~), allow programmers to manipulate individual bits within binary sequences/ representations. Bitwise operators are commonly used in low-level programming, embedded systems  Example for AND operation  #include <stdio.h>  int main () {  int x = 5;  int y = 3;  //Using bitwise AND operator on x and y  int result = x & y;  printf ("a & b = %d\n", result);  return 0;  }  The variable x is initialized with the decimal value 5, which is 101 in binary.  And y is initialized with the decimal value 3, which is 011 in binary.  When the bitwise AND operation is performed, comparing the binary representations of x and y bit by bit, the outcome is stored in the result variable, which is 1 (binary 001).  Example for OR operation  #include <stdio.h>  int main () {  int x = 5;  int y = 3;  int z = x | y;  printf ("a | b = %d\n", z);  return 0;  }  We initialize x with the decimal value 5, which is 101 in binary, and y with the decimal value 3, which is 011 in binary.  Then, we use the bitwise OR operation to perform a bitwise OR operation between x and y.  The result of the bitwise OR operation is stored in the integer variable z. In this case, the result is 7, which is 111 in binary.  Example for XOR operation  #include <stdio.h>  int main () {  int x = 5;  int y = 3;  int z = x ^ y;  printf ("a ^ y = %d\n", z);  return 0;  }  The result of the bitwise XOR operation is stored in the integer variable z. In this case, the result is 6, which is 110 in binary.  Example for compliment operation  #include <stdio.h>  int main () {  int p = 5;  int q = ~p;  printf ("~a = %d\n", q);  return 0;  }  The variable p is initialized with the value of 5.  In initializing variable q, we use the bitwise complement operationand assign q the value of p's complement. This operation inverts all the bits in p.  The result of the bitwise complement operation is stored in the integer variable q. In this case, the result is -6. |
| 13.Explain the difference between the postfix and prefix increment operators (++)  in C programming. Provide examples to illustrate |
| 'A++' is the postfix increment operator, which uses the current value of 'A' in the expression and then increments it by one. On the other hand, '++A' is the prefix increment operator, which increments 'A' first and then uses the updated value in the expression.  #include  int main () {  int num = 5;    printf ("Original value: %dn", num); // Output: Original value: 5    num++; // Postfix increment    printf ("After postfix increment: %dn", num); // Output: After postfix increment: 6    ++num; // Prefix increment    Printf ("After prefix increment: %dn", num); // Output: After prefix increment: 7    return 0;  } |
| 14.What is the significance of the logical AND (&&) and logical OR (||) operators  in C programming? How are they used in conditional expressions? |
| The logical AND (&&) operator checks whether all operands are true – the result is true only when all operands are true. Something to note here is that, when the first operand is false, the second operand is not evaluated.  Example for AND operation  #include <stdio.h>  int main(void) {  int a = 20;  int b = 30;  if (a > 10 && b > 10)  printf ("Both numbers are greater than 10\n");  }  // Both numbers are greater than 10  In the example above, the output is Both numbers are greater than 10 because the condition a > 10 && b > 10 is satisfied.  Both a > 10 and b > 10 are true, so the result is true.  If either a or b did not satisfy the condition, then there would be no output in the console since I have not specified an else condition.  Example for OR operation  #include <stdio.h>  int main(void) {  int a = 20;  int b = 5;  if (a > 10 || b > 10)  printf ("At least one of the numbers is greater than 10");  }  In the example above, the output is At least one of the numbers is greater than 10 because the condition a > 10 || b > 10 is satisfied – at least one of the operands is true.  The first condition, a > 10, is true, so the result is true.  If both a and b were false, there would be no output. |
| 15.Discuss the concept of operator precedence and associativity in C programming.  Provide examples to demonstrate how they affect expression evaluation. |
| Operator precedence helps us determine which of the operators in an expression must be evaluated first in case the expression consists of more than a single operator.  Example, 50 – 2 \* 15 is going to yield 20. It is because it gets evaluated as 50 – (2 \* 15), and not as (50 – 2) \* 15.  Operators with the same precedence are assessed from left to right according to left-to-right associativity. For instance, the subtraction operators have left-to-right associativity in the statement x = a - b - c. The expression is therefore calculated as (a - b) - (c). |
| **Functions:** |
| 16.Describe the purpose and structure of a function prototype in C programming.  Why is it necessary to declare function prototypes? |
| In C programming, a function prototype is used to declare the signature of a function, which includes its name, return type, and parameters.  Function prototypes are important because they inform the compiler of the function's interface before it is called, allowing for proper type checking and error handling. |
| 17.Explain the difference between call by value and call by reference in C  programming. Provide examples to illustrate both concepts. |
| Call by Value passes a copy of the variable . Call by Reference passes the variable itself . In Call by Value, the actual arguments and passed parameters of a function refer to different memory locations. In Call by Reference, the actual arguments and passed parameters of a function refer to the same memory location.  Example of Call by Value  #include <stdio.h>    // Function Prototype  void swap x (int x, int y);    // Main function  int main ()  {      int a = 10, b = 20;        // Pass by Values      Swap x (a, b); // Actual Parameters        Printf("In the Caller:\na = %d b = %d\n", a, b);        return 0;  }    // Swap functions that swaps  // two values  void swap x (int x, int y) // Formal Parameters  {      int t;        t = x;      x = y;      y = t;        printf("Inside Function:\nx = %d y = %d\n", x, y);  }  Inside Function:  x = 20 y = 10  In the Caller:  a = 10 b = 20  **Call by Reference**  #include <stdio.h>    // Function Prototype  void swapx(int\*, int\*);    // Main function  int main()  {      int a = 10, b = 20;        // Pass reference      Swap x (&a, &b); // Actual Parameters        Printf ("Inside the Caller:\na = %d b = %d\n", a, b);        return 0;  }    // Function to swap two variables  // by references  void swap x (int\* x, int\* y) // Formal Parameters  {      int t;        t = \*x;      \*x = \*y;      \*y = t;        Printf ("Inside the Function:\nx = %d y = %d\n", \*x, \*y);  }  Inside the Function:  x = 20 y = 10  Inside the Caller:  a = 20 b = 10 |
| 18.Discuss the concept of recursion in C programming. Provide an example of a  recursive function and explain how it works. |
| The recursion process in C refers to the process in which the program repeats a certain section of code in a similar way. Thus, in the programming languages, when the program allows the user to call any function inside the very same function, it is referred to as a recursive call in that function. #include <stdio.h>  unsigned long long int factorial (unsigned int x) {  if(x <= 1) {  return 1;  }  return x \* factorial (x – 1);  }  int main () {  int x = 12;  printf (“The factorial of the number %d is equal to %d\n”, x, factorial(x));  return 0;  }  The compilation and execution of the code mentioned above will ultimately produce the result given below:  The factorial of the number 12 is equal to 479001600 |
| 19.What is the significance of the return statement in C programming? How are  values returned from functions? |
| **C return statement** ends the execution of a function and returns the control to the function from where it was **called**. The return statement may or may not return a value depending upon the return type of the function. For example, int returns an integer value, void returns nothing, etc.  A function defined with a return type must include an expression containing the value to be returned |
| 20.Describe the role of function parameters and arguments in C programming.  How are function arguments passed to parameters? |
| A parameter is a special kind of variable used in a function to refer to one of the pieces of data provided as input to the function.  arguments are copied to the program stack at run time, where they are read by the function. |
| **Arrays:** |
| 21.Explain the concept of arrays in C programming. How are arrays declared and  initialized? |
| An array is defined as the collection of similar type of data items stored at contiguous memory locations.  int[] myArray = new int[3]; |
| 22.Discuss the difference between a one-dimensional array and a multi  dimensional array in C programming. Provide examples of both. |
| A one-dimensional array stores a single list of various elements having a similar data type. A two-dimensional array stores an array of various arrays, or a list of various lists, or an array of various one-dimensional arrays  int arr[5]; //an array with one row and five columns will be created. {a , b , c , d , e}  int arr[2][5]; //an array with two rows and five columns will be created. a b c d e f g h i j |
| 23.Describe the process of accessing array elements in C programming. How are  array indices used to access elements? |
| Elements of an array are accessed by specifying the index ( offset ) of the desired element within square [ ] brackets after the array name.  Array indexing is the same as accessing an array element. You can access an array element by referring to its index number. The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc. |
| 24What is the significance of the null character ('\0') in C strings? How is it used  to determine the end of a string? |
| The null character plays an indispensable role in string management. Without it, functions like strlen (), strcpy (), and many others would have no means of determining where a string ends. It acts as a sentinel value, signaling to various string-handling functions that the end of the string has been reached. |
| 25Explain the concept of dynamic memory allocation for arrays in C  programming. How are dynamic arrays allocated and deallocated? |
| Memory for dynamic arrays is allocated using pointers and memory allocation functions, such as malloc or calloc.  The **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size.  ptr = (cast-type\*) malloc(byte-size) **For Example:**  **ptr *=* int*\** malloc*100 \** sizeof(int*);***  **“**calloc**”** or **“**contiguousallocation**”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:  ptr = (cast-type\*)calloc(n, element-size); here, n is the no. of elements and element-size is the size of each element.  **For Example:**  ***ptr = (float\*) calloc(25, sizeof(float));*** |
| **Pointers:** |
| 26.Describe the purpose and usage of pointers in C programming. How are pointers  declared and initialized? |
| Pointer is a variable used to store the address of another variable or a memory location. It always stores a integer since it shares an address. Pointer is declared using special character '\*' along with datatype pointer points and name of the pointer as an identifier |
| 27.Explain the concept of pointer arithmetic in C programming. Provide examples  to illustrate addition and subtraction operations on pointers. |
| Pointer arithmetic is the process of manipulating pointer values by using arithmetic operators. C programming supports four basic arithmetic operations on pointers: addition, subtraction, increment, and decrement. In the example, we have an array of integers and a pointer pointing to the first element of the array.  **For Example:** Consider the same example as above where the **ptr**is an **integer pointer**that stores **1000**as an address. If we subtract integer 5 from it using the expression, **ptr = ptr – 5,** then, the final address stored in the ptr will be **ptr = 1000 – sizeof(int) \* 5 = 980.**  **For-Example:** Consider the same example as above where the **ptr**is an **integer pointer**that stores **1000**as an address. If we add integer 5 to it using the expression, **ptr = ptr + 5,** then, the final address stored in the ptr will be **ptr = 1000 + sizeof(int) \* 5 = 1020.** |
| 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. |
| The difference between pass-by-reference and pass-by-pointer is that pointers can be NULL or reassigned whereas references cannot. Use pass-by-pointer if NULL is a valid parameter value or if you want to reassign the pointer. Otherwise, use constant or non-constant references to pass arguments. |
| 29.Describe the concept of NULL pointers in C programming. How are NULL  pointers used and checked for in programs? |
| NULL Pointers are used to denote the absence of a valid reference when a function does not need to pass or return a valid memory address.  **Uses of NULL Pointer in C**   1. To initialize a pointer variable when that pointer variable hasn't been assigned any valid memory address yet. 2. To check for a null pointer before accessing any pointer variable. ... 3. To pass a null pointer to a function argument when we don't want to pass any valid memory address. |
| 30.Explain the role of pointers in dynamic memory allocation in C programming.  How are pointers used to allocate and deallocate memory dynamically? |
| A pointer value is the address of a variable or a chunk of dynamically-allocated memory.  In C, dynamic memory is allocated from the heap using some standard library functions. The two key dynamic memory functions are malloc() and free(). The malloc() function takes a single parameter, which is the size of the requested memory area in bytes. It returns a pointer to the allocated memory. |
| **Strings:** |
| 31.Discuss the concept of strings in C programming. How are strings represented  and manipulated in C? |
| A string in C is merely an array of characters. The length of a string is determined by a terminating null character: '\0' . So, a string with the contents, say, "abc" has four characters: 'a' , 'b' , 'c' , and the terminating null ( '\0' ) character. The terminating null character has the value zero.  Some of the common string functions in C includes strlen(), strcpy(), strcat(), and strcmp(). These functions are used to manipulate strings, including finding their length, copying them, concatenating them, and comparing them.  A string is represented by a variable of type char \*, which points to the zeroth character of the string |
| 32.Explain the difference between character arrays and string literals in C  programming. Provide examples to illustrate both concepts |
| String refers to a sequence of characters represented as a single data type. Character Array is a sequential collection of data type char. **Example** String my\_string = "JANE" ;  char [] ch = my\_string.toCharArray(); **Example** Char [] my\_char = {'J','A','N','E'};  String my\_str = new String(my\_char); |
| 33.Describe common string manipulation functions available in the C standard  library. Provide examples of functions like strlen, strcpy, strcat, and strcmp |
| * strcat - concatenate two strings * strchr - string scanning operation * strcmp - compare two strings * strcpy - copy a string * strlen - get string length * strncat - concatenate one string with part of another * strncmp - compare parts of two strings * strncpy - copy part of a string * strrchr - string scanning operation |
| 34.Discuss the concept of string tokenization in C programming. How are strings  split into tokens using delimiter characters? |
| In C, tokenization is the process of breaking the string into smaller parts using delimiters (characters treated as separators) like space, commas, a specific character, or even a string.   use the strtok() function to split a string (and specify the delimiter to use). |
| Explain the importance of null-terminated strings in C programming. How does  the null character ('\0') signify the end of a string? |
| The null character plays an indispensable role in string management. Without it, functions like strlen() , strcpy() , and many others would have no means of determining where a string ends. It acts as a sentinel value, signaling to various string-handling functions that the end of the string has been reached.  The Null Character in C is used to denote the end of a C string, indicating that there are no more characters in the sequence after it. |
| **Structures and Unions:** |
| Describe the purpose and usage of structures in C programming. How are  structures declared and accessed? |
| We can access structure members using structure variables. The use of structure in C is to store data of different data types. For example: If I want to store employees' details of a company then I need to store their name, age, department, salary, etc which will include string, integer, etc data types.  The structure is declared using the “struct” keyword. Structure allocates contiguous memory to all its member variables. |
| Discuss the concept of structure members in C programming. How are  individual members of a structure accessed and modified? |
| Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the struct   use a dot operator “.” (dot), then you have to mention the member element name. Modify the Data in a Structure by Passing It to a Function which accepts a structure as an input argument and returns a structure after modifying data in it |
| Explain the difference between structures and unions in C programming. When  would you choose one over the other? |
| In the case of a Structure, there is a specific memory location for every input data member. Thus, it can store multiple values of the various members. In the case of a Union, there is an allocation of only one shared memory for all the input data members. Thus, it stores one value at a time for all of its members. |
| Describe the concept of nested structures in C programming. How are structures  within structures defined and accessed? |
| A nested structure in C is a structure that contains one or more members that are themselves structures. Nested structures are useful for organizing complex data and can help improve a program's readability and maintainability.   Outer and inner structure variables are declared as normal variables and the data members of the outer structure are accessed using a single dot(.) and the data members of the inner structure are accessed using the two dots |
| Discuss the concept of typedef in C programming. How is typedef used to define  custom data types, including structures and unions? |
| The C typedef keyword is used to redefine the name of already existing data types. When names of datatypes become difficult to use in programs, typedef is used with user-defined datatypes, which behave similarly to defining an alias for commands. |
| **File Handling:** |
| Explain the concept of file handling in C programming. How are files opened,  read from, and written to using standard file handling functions? |
| 1. fopen() - create a new file or open a existing file. 2. fclose() - close a file. 3. getc() - reads a character from a file. 4. putc() - writes a character to a file. 5. fscanf() - reads a set of data from a file. 6. fprintf() - writes a set of data to a file. |
| Describe the role of file pointers in C programming. How are file pointers used  to navigate and manipulate files? |
| A file pointer in C is a data type that is used to point to a file. It is a structure that holds information such as the name of the file, its location, and the mode in which the file is accessed. A file pointer is used to read from and write to files, as well as to control its position inside the file.  A file pointer stores the current position of a read or write within a file. All operations within the file are made with reference to the pointer. |
| Discuss the difference between text files and binary files in C programming.  How are they opened and processed differently? |
| Text files are organized around lines, each of which ends with a newline character ('\n'). The source code files are themselves text files. A binary file is the one in which data is stored in the file in the same way as it is stored in the main memory for processing.  A text file consists of human readable characters, which can be opened by any text editor. On the other hand, binary files are made up of non-human readable characters and symbols, which require specific programs to access its contents. |
| Explain the purpose of file modes in C programming. Provide examples of  different file modes like "r", "w", "a", etc. |
| control whether a file is overwritten, created, opened, or some combination thereof. fptr = fopen("example.txt", "r"); *// Open the file in read mode*  filePtr = fopen("example.txt", "w"); |
| Describe error handling techniques in file operations in C programming. How  are errors detected and handled when working with files? |
| 1. perror() perror() function stands for print error and when called by the user, it displays a message describing about the most recent error that occured in the code. ... 2. strerror() strerror() function is contained in string. ... 3. ferror() ... 4. feof() ... 5. clearerr() ... 6. Exit Status. ... 7. Division by zero.   The system can detect errors when a file is opened, when a program device is acquired or released, during I/O operations to a file, and when the file is closed. |
|  |
|  |
|  |
|  |
|  |

**Part- B**

|  |
| --- |
| 1. **Hello World:** Print "Hello, World!" to the console. |
| Code  #include<stdio.h>  main()  {  printf("Hello,World!");  } |
| Output: |
|  |
| 2. **Factorial:** Calculate the factorial of a given number. |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define NUMBERS\_TO\_GENERATE 1  #define FILENAME "random\_numbers.txt"  #define FACTORIAL\_FILENAME "factorials.txt"  void generateRandomNumbers(const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  int num = rand() % 10 + 1; // Generate random number between 1 and 10  fprintf(file, "%d\n", num);  printf("%d\n",num);  }  fclose(file);  }  int factorial(int n) {  if (n == 0 || n == 1)  return 1;  else  return n \* factorial(n - 1);  }  void generateFactorials(const char \*srcFilename, const char \*destFilename) {  FILE \*srcFile = fopen(srcFilename, "r");  FILE \*destFile = fopen(destFilename, "w");  if (srcFile == NULL || destFile == NULL) {  perror("Error opening files");  exit(EXIT\_FAILURE);  }  int num;  while (fscanf(srcFile, "%d", &num) != EOF) {  int fact = factorial(num);  fprintf(destFile, "%d\n", fact);  printf("factorials are %d\n",fact);  }  fclose(srcFile);  fclose(destFile);  }  int main() {  generateRandomNumbers(FILENAME);  generateFactorials(FILENAME, FACTORIAL\_FILENAME);  printf("Factorials generated and stored successfully.\n");  return 0;  } |
| Output: |
|  |
|  |
|  |
|  |
| **Prime Numbers:** Determine whether a given number is prime |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define NUMBERS\_TO\_GENERATE 2 // You can adjust the number of random numbers generated  #define FILENAME "random\_numbers.txt"  #define PRIME\_FILENAME "prime\_numbers.txt"  void generateRandomNumbers(const char \*filename, int lower, int upper) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int num,i;  for (i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  num = (rand() % (upper - lower + 1)) + lower; // Generate random number within the range  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  int isPrime(int n) {  int i;  if (n < 2)  return 0;  for ( i = 2; i \* i <= n; i++) {  if (n % i == 0)  return 0;  }  return 1;  }  void findPrimes(const char \*srcFilename, const char \*destFilename) {  FILE \*srcFile = fopen(srcFilename, "r");  FILE \*destFile = fopen(destFilename, "w");  if (srcFile == NULL || destFile == NULL) {  perror("Error opening files");  exit(EXIT\_FAILURE);  }  int num;  while (fscanf(srcFile, "%d", &num) != EOF) {  if (isPrime(num))  fprintf(destFile, "%d\n", num);  }  fclose(srcFile);  fclose(destFile);  }  int main() {  int lower = 1; // Lower limit of the range  int upper = 100; // Upper limit of the range  generateRandomNumbers(FILENAME, lower, upper);  findPrimes(FILENAME, PRIME\_FILENAME);  printf("Prime numbers found and stored successfully.\n");  return 0;  } |
|  |
|  |
|  |
|  |
| **Fibonacci Series:** Generate the Fibonacci series up to a certain limit |
| Code:  #include <stdio.h>  #include <stdlib.h>  #define NUMBERS\_TO\_GENERATE 10 // You can adjust the number of Fibonacci numbers generated  #define FILENAME "fibonacci\_numbers.txt"  void generateFibonacciNumbers(const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  int i, first = 0, second = 1, next;  for (i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  if (i <= 1)  next = i;  else {  next = first + second;  first = second;  second = next;  }  fprintf(file, "%d\n", next);  printf("%d\n", next);  }  fclose(file);  }  int main() {  generateFibonacciNumbers(FILENAME);  printf("Fibonacci numbers generated and stored successfully.\n");  return 0;  } |
| Output: |
|  |
| **Sum of Digits:** Calculate the sum of digits of a given number. |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define NUMBERS\_TO\_GENERATE 1 // You can adjust the number of random numbers generated  #define FILENAME "random\_numbers.txt"  #define DIGIT\_SUM\_FILENAME "digit\_sum.txt"  void generateRandomNumbers(const char \*filename, int lower, int upper) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for (i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  int num = (rand() % (50 + 1)); // Generate random number within the range  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  int sumOfDigits(int number) {  int sum = 0;  while (number != 0) {  sum += number % 10; // Add the last digit to the sum  number /= 10; // Remove the last digit from the number  }  return sum;  }  void calculateAndSaveDigitSum(const char \*srcFilename, const char \*destFilename) {  FILE \*srcFile = fopen(srcFilename, "r");  FILE \*destFile = fopen(destFilename, "w");  if (srcFile == NULL || destFile == NULL) {  perror("Error opening files");  exit(EXIT\_FAILURE);  }  int num;  while (fscanf(srcFile, "%d", &num) != EOF) {  int sum = sumOfDigits(num);  fprintf(destFile, "%d\n", sum);  printf("Sum of digits for %d: %d\n", num, sum);  }  fclose(srcFile);  fclose(destFile);  }  int main() {  int lower = 1; // Lower limit of the range  int upper = 100; // Upper limit of the range  generateRandomNumbers(FILENAME, lower, upper);  calculateAndSaveDigitSum(FILENAME, DIGIT\_SUM\_FILENAME);  printf("Sum of digits calculated and stored successfully.\n");  return 0;  } |
|  |
|  |
|  |
| **Reverse a Number:** Reverse the digits of a given number |
| Code:  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define NUMBERS\_TO\_GENERATE 3 // You can adjust the number of random numbers generated  #define FILENAME "random\_numbers.txt"  #define REVERSED\_FILENAME "reversed\_numbers.txt"  void generateRandomNumbers(const char \*filename, int lower, int upper) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for (i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  int num = (rand() % (1000+ 1)); // Generate random number within the range  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  int reverseNumber(int number) {  int reversed = 0;  while (number != 0) {  int digit = number % 10; // Extract the last digit  reversed = reversed \* 10 + digit; // Append the extracted digit to the reversed number  number /= 10; // Remove the last digit  }  return reversed;  }  void reverseAndSaveNumbers(const char \*srcFilename, const char \*destFilename) {  FILE \*srcFile = fopen(srcFilename, "r");  FILE \*destFile = fopen(destFilename, "w");  if (srcFile == NULL || destFile == NULL) {  perror("Error opening files");  exit(EXIT\_FAILURE);  }  int num;  while (fscanf(srcFile, "%d", &num) != EOF) {  int reversed = reverseNumber(num);  fprintf(destFile, "%d\n", reversed);  printf("Reversed digits of %d: %d\n", num, reversed);  }  fclose(srcFile);  fclose(destFile);  }  int main() {  int lower = 1; // Lower limit of the range  int upper = 100; // Upper limit of the range  generateRandomNumbers(FILENAME, lower, upper);  reverseAndSaveNumbers(FILENAME, REVERSED\_FILENAME);  printf("Numbers with reversed digits calculated and stored successfully.\n");  return 0;  }  Output: |
|  |
|  |
|  |
| **Palindrome Check:** Check if a given number or string is a palindrome |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAX\_LENGTH 100  #define FILENAME "random\_string.txt"  void generateRandomString(const char \*filename, int length) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for (i = 0; i < length; i++) {  char randomChar = 'A' + rand() % 26; // Generate a random uppercase letter  fprintf(file, "%c", randomChar);  printf("%c", randomChar);  }  fclose(file);  }  int isPalindrome(const char \*str) {  int length = strlen(str);  int i, j;  for (i = 0, j = length - 1; i < j; i++, j--) {  if (tolower(str[i]) != tolower(str[j])) { // Convert characters to lowercase for case-insensitive comparison  return 0;  }  }  return 1;  }  int main() {  int length = 10; // Length of the generated string  generateRandomString(FILENAME, length);  char string[MAX\_LENGTH];  FILE \*file = fopen(FILENAME, "r");  if (file == NULL) {  perror("Error opening file for reading");  exit(EXIT\_FAILURE);  }  fgets(string, sizeof(string), file); // Read the generated string from the file  fclose(file);  printf("\nGenerated string: %s\n", string);  if (isPalindrome(string)) {  printf("The generated string is a palindrome.\n");  } else {  printf("The generated string is not a palindrome.\n");  }  return 0;  } |
| Output: |
|  |
| **Palindrome Check:** Check if a given number or string is a palindrome. |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define NUMBERS\_TO\_GENERATE 10 // You can adjust the number of random numbers generated  #define FILENAME "random\_numbers.txt"  #define PALINDROME\_FILENAME "palindrome\_numbers.txt"  void generateRandomNumbers(const char \*filename, int lower, int upper) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  int num = (rand() % (upper - lower + 1)) + lower; // Generate random number within the range  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  int isPalindrome(int number) {  int original = number;  int reverse = 0;  while (number > 0) {  reverse = reverse \* 10 + number % 10; // Build the reverse number digit by digit  number /= 10;  }  return original == reverse;  }  void storePalindromes(const char \*srcFilename, const char \*destFilename) {  FILE \*srcFile = fopen(srcFilename, "r");  FILE \*destFile = fopen(destFilename, "w");  if (srcFile == NULL || destFile == NULL) {  perror("Error opening files");  exit(EXIT\_FAILURE);  }  int num;  printf("palindrome numbers are\n");  while (fscanf(srcFile, "%d", &num) != EOF) {  if (isPalindrome(num)) {  fprintf(destFile, "%d\n", num);// Write palindrome number to the destination file  printf("%d\n",num);  }  }  fclose(srcFile);  fclose(destFile);  }  int main() {  int lower = 1; // Lower limit of the range  int upper = 100; // Upper limit of the range  generateRandomNumbers(FILENAME, lower, upper);  storePalindromes(FILENAME, PALINDROME\_FILENAME);  printf("Palindrome numbers stored successfully in %s.\n", PALINDROME\_FILENAME);  return 0;  } |
|  |
|  |
|  |
|  |
| **Area of Shapes:** Calculate the area of shapes like rectangle, triangle, and circle |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #define FILENAME "shapes\_areas.txt"  // Function to calculate the area of a rectangle  float rectangleArea(float length, float width) {  return length \* width;  }  // Function to calculate the area of a triangle  float triangleArea(float base, float height) {  return 0.5 \* base \* height;  }  // Function to calculate the area of a circle  float circleArea(float radius) {  return M\_PI \* radius \* radius;  }  // Function to store the calculated areas in a file  void storeAreas(const char \*filename, float rectangleArea, float triangleArea, float circleArea) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  fprintf(file, "Area of Rectangle: %.2f\n", rectangleArea);  fprintf(file, "Area of Triangle: %.2f\n", triangleArea);  fprintf(file, "Area of Circle: %.2f\n", circleArea);  fclose(file);  }  int main() {  // Define dimensions for shapes  float rectangleLength = 5.0;  float rectangleWidth = 3.0;  float triangleBase = 4.0;  float triangleHeight = 6.0;  float circleRadius = 2.5;  // Calculate areas  float areaRectangle = rectangleArea(rectangleLength, rectangleWidth);  float areaTriangle = triangleArea(triangleBase, triangleHeight);  float areaCircle = circleArea(circleRadius);  // Store areas in a file  storeAreas(FILENAME, areaRectangle, areaTriangle, areaCircle);  printf("Areas calculated and stored successfully in %s.\n", FILENAME);  return 0;  } |
|  |
|  |
| **Simple Calculator:** Implement a basic calculator with arithmetic operations |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define NUMBERS\_TO\_GENERATE 2 // You can adjust the number of random numbers generated  #define INPUT\_FILENAME "random\_numbers.txt"  #define OUTPUT\_FILENAME "calculation\_result.txt"  void generateRandomNumbers(const char \*filename, int lower, int upper) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  int num = (rand() % (upper - lower + 1)) + lower; // Generate random number within the range  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  double performCalculation(FILE \*file, char operator) {  double result;  int num;  fscanf(file, "%d", &num); // Read the first number  result = num; // Initialize the result with the first number  while (fscanf(file, "%d", &num) != EOF) {  switch(operator) {  case '+':  result += num; // Perform addition  break;  case '-':  result -= num; // Perform subtraction  break;  case '\*':  result \*= num; // Perform multiplication  break;  case '/':  if (num == 0) {  printf("Error: Division by zero\n");  exit(EXIT\_FAILURE);  }  result /= num; // Perform division  break;  default:  printf("Error: Invalid operator\n");  exit(EXIT\_FAILURE);  }  }  return result;  }  void storeResult(const char \*filename, double result) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  fprintf(file, "Result of calculation: %.2f\n", result);  fclose(file);  }  int main() {  int lower = 1; // Lower limit of the range  int upper = 100; // Upper limit of the range  // Generate random numbers and store them in a file  generateRandomNumbers(INPUT\_FILENAME, lower, upper);  // Open the file containing generated numbers for calculation  FILE \*inputFile = fopen(INPUT\_FILENAME, "r");  if (inputFile == NULL) {  perror("Error opening input file");  return EXIT\_FAILURE;  }  char operator;  printf("Enter operator (+, -, \*, /): ");  scanf(" %c", &operator);  // Perform calculation on the generated numbers  double result = performCalculation(inputFile, operator);  // Close the input file  fclose(inputFile);  // Store the result of the calculation in another file  storeResult(OUTPUT\_FILENAME, result);  printf("Calculation result stored successfully in %s.\n", OUTPUT\_FILENAME);  return 0;  } |
|  |
|  |
|  |
| **Array Operations:** Perform operations like finding the largest/smallest element, sum,  and average of an array. |
| #include <time.h>  #define NUMBERS\_TO\_GENERATE 10 // You can adjust the number of random numbers generated  #define INPUT\_FILENAME "random\_numbers.txt"  #define OUTPUT\_FILENAME "calculation\_result.txt"  void generateRandomNumbers(const char \*filename, int lower, int upper) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < NUMBERS\_TO\_GENERATE; i++) {  int num = (rand() % (100 + 1)); // Generate random number within the range  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  int findLargestElement(FILE \*file) {  int largest;  fscanf(file, "%d", &largest); // Read the first number  int num;  while (fscanf(file, "%d", &num) != EOF) {  if (num > largest)  largest = num;  }  return largest;  }  int findSmallestElement(FILE \*file) {  int smallest;  fscanf(file, "%d", &smallest); // Read the first number  int num;  while (fscanf(file, "%d", &num) != EOF) {  if (num < smallest)  smallest = num;  }  return smallest;  }  int calculateSum(FILE \*file) {  int sum = 0;  int num;  while (fscanf(file, "%d", &num) != EOF) {  sum += num;  }  return sum;  }  double calculateAverage(FILE \*file) {  int count = 0;  int sum = 0;  int num;  while (fscanf(file, "%d", &num) != EOF) {  sum += num;  count++;  }  if (count == 0) {  printf("Error: Division by zero\n");  exit(EXIT\_FAILURE);  }  return (double)sum / count;  }  void storeResult(const char \*filename, int result) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  fprintf(file, "Result: %d\n", result);  fclose(file);  }  int main() {  int lower = 1; // Lower limit of the range  int upper = 100; // Upper limit of the range  // Generate random numbers and store them in a file  generateRandomNumbers(INPUT\_FILENAME, lower, upper);  // Open the file containing generated numbers for calculation  FILE \*inputFile = fopen(INPUT\_FILENAME, "r");  if (inputFile == NULL) {  perror("Error opening input file");  return EXIT\_FAILURE;  }  // Find largest element  int largest = findLargestElement(inputFile);  printf("Largest element: %d\n", largest);  // Reset file pointer to the beginning  fseek(inputFile, 0, SEEK\_SET);  // Find smallest element  int smallest = findSmallestElement(inputFile);  printf("Smallest element: %d\n", smallest);  // Reset file pointer to the beginning  fseek(inputFile, 0, SEEK\_SET);  // Calculate sum  int sum = calculateSum(inputFile);  printf("Sum: %d\n", sum);  // Reset file pointer to the beginning  fseek(inputFile, 0, SEEK\_SET);  // Calculate average  double average = calculateAverage(inputFile);  printf("Average: %.2f\n", average);  // Close the input file  fclose(inputFile);  // Store the result of the calculation in another file  storeResult(OUTPUT\_FILENAME, sum);  printf("Calculation result stored successfully in %s.\n", OUTPUT\_FILENAME);  return 0;  } |
|  |
|  |
| **String Operations:** Manipulate strings such as concatenation, copying, and  comparison. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <time.h>  #define NUMBERS\_TO\_GENERATE 2// You can adjust the number of random strings generated  #define INPUT\_FILENAME "random\_strings.txt"  #define OUTPUT\_FILENAME "calculation\_result.txt"  #define MAX\_STRING\_LENGTH 10  void generateRandomStrings(const char \*filename, int numStrings) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  char alphabet[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";  int alphabetLength = strlen(alphabet);  srand(time(NULL));  int i,j;  for ( i = 0; i < numStrings; i++) {  char randomString[MAX\_STRING\_LENGTH];  int length = rand() % MAX\_STRING\_LENGTH + 1;  for (j = 0; j < length; j++) {  randomString[j] = alphabet[rand() % alphabetLength];  }  randomString[length] = '\0'; // Null-terminate the string  fprintf(file, "%s\n", randomString);  printf("%s\n", randomString);  }  fclose(file);  }  char \*concatenateStrings(const char \*str1, const char \*str2) {  int len1 = strlen(str1);  int len2 = strlen(str2);  char \*result = malloc(len1 + len2 + 1); // +1 for null terminator  if (result == NULL) {  perror("Error allocating memory for concatenation");  exit(EXIT\_FAILURE);  }  strcpy(result, str1);  strcat(result, str2);  return result;  }  void copyString(char \*dest, const char \*src) {  strcpy(dest, src);  }  int compareStrings(const char \*str1, const char \*str2) {  return strcmp(str1, str2);  }  void storeResult(const char \*filename, const char \*result) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  fprintf(file, "Result: %s\n", result);  fclose(file);  }  int main() {  int numStrings = NUMBERS\_TO\_GENERATE; // Adjust the number of random strings generated  // Generate random strings and store them in a file  generateRandomStrings(INPUT\_FILENAME, numStrings);  // Open the file containing generated strings for manipulation  FILE \*inputFile = fopen(INPUT\_FILENAME, "r");  if (inputFile == NULL) {  perror("Error opening input file");  return EXIT\_FAILURE;  }  char str1[MAX\_STRING\_LENGTH], str2[MAX\_STRING\_LENGTH];  fscanf(inputFile, "%s", str1);  fscanf(inputFile, "%s", str2);  // Concatenate strings  char \*concatenated = concatenateStrings(str1, str2);  printf("Concatenated string: %s\n", concatenated);  // Copy strings  char copied[MAX\_STRING\_LENGTH];  copyString(copied, str1);  printf("Copied string: %s\n", copied);  // Compare strings  int comparisonResult = compareStrings(str1, str2);  printf("Comparison result: %d\n", comparisonResult);  // Close the input file  fclose(inputFile);  // Store the result of the concatenation in another file  storeResult(OUTPUT\_FILENAME, concatenated);  printf("Concatenation result stored successfully in %s.\n", OUTPUT\_FILENAME);  // Free allocated memory  free(concatenated);  return 0;  } |
|  |
|  |
| **Binary Search:** Search for an element in a sorted array using binary search. |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define ARRAY\_SIZE 10  #define INPUT\_FILENAME "random\_numbers.txt"  #define OUTPUT\_FILENAME "search\_result.txt"  void generateRandomNumbers(const char \*filename, int size) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < size; i++) {  int num = rand() % 100; // Generate random number between 0 and 99  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  int compareIntegers(const void \*a, const void \*b) {  return (\*(int \*)a - \*(int \*)b);  }  int binarySearch(int arr[], int size, int key) {  int low = 0;  int high = size - 1;  while (low <= high) {  int mid = low + (high - low) / 2;  if (arr[mid] == key) {  return mid; // Element found, return index  } else if (arr[mid] < key) {  low = mid + 1; // Search the right half  } else {  high = mid - 1; // Search the left half  }  }  return -1; // Element not found  }  void storeResult(const char \*filename, int result) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  fprintf(file, "Result: %d\n", result);  fclose(file);  }  int main() {  int size = ARRAY\_SIZE;  // Generate random numbers and store them in a file  generateRandomNumbers(INPUT\_FILENAME, size);  // Open the file containing generated numbers for manipulation  FILE \*inputFile = fopen(INPUT\_FILENAME, "r");  if (inputFile == NULL) {  perror("Error opening input file");  return EXIT\_FAILURE;  }  // Read the numbers into an array  int numbers[ARRAY\_SIZE],i;  for ( i = 0; i < ARRAY\_SIZE; i++) {  fscanf(inputFile, "%d", &numbers[i]);  }  // Close the input file  fclose(inputFile);  // Sort the array of numbers  qsort(numbers, ARRAY\_SIZE, sizeof(int), compareIntegers);  // Perform binary search  int key = 42; // Element to search for  int index = binarySearch(numbers, ARRAY\_SIZE, key);  printf("Element %d found at index: %d\n", key, index);  // Store the result of the search in another file  storeResult(OUTPUT\_FILENAME, index);  printf("Search result stored successfully in %s.\n", OUTPUT\_FILENAME);  return 0;  } |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| 14. **Selection Sort:** Sort an array using the selection sort algorithm. |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define ARRAY\_SIZE 10  #define INPUT\_FILENAME "random\_numbers.txt"  #define OUTPUT\_FILENAME "sorted\_numbers.txt"  void generateRandomNumbers(const char \*filename, int size) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < size; i++) {  int num = rand() % 100; // Generate random number between 0 and 99  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  void selectionSort(int arr[], int size) {  int i,j;  for ( i = 0; i < size - 1; i++) {  int minIndex = i;  for ( j = i + 1; j < size; j++) {  if (arr[j] < arr[minIndex]) {  minIndex = j;  }  }  if (minIndex != i) {  // Swap arr[i] and arr[minIndex]  int temp = arr[i];  arr[i] = arr[minIndex];  arr[minIndex] = temp;  }  }  }  void storeSortedNumbers(const char \*filename, int arr[], int size) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  int i;  for ( i = 0; i < size; i++) {  fprintf(file, "%d\n", arr[i]);  }  fclose(file);  }  int main() {  int size = ARRAY\_SIZE;  // Generate random numbers and store them in a file  generateRandomNumbers(INPUT\_FILENAME, size);  // Open the file containing generated numbers for manipulation  FILE \*inputFile = fopen(INPUT\_FILENAME, "r");  if (inputFile == NULL) {  perror("Error opening input file");  return EXIT\_FAILURE;  }  // Read the numbers into an array  int numbers[ARRAY\_SIZE];  int i;  for ( i = 0; i < ARRAY\_SIZE; i++) {  fscanf(inputFile, "%d", &numbers[i]);  }  // Close the input file  fclose(inputFile);  // Sort the array using selection sort  selectionSort(numbers, ARRAY\_SIZE);  // Store the sorted numbers in another file  storeSortedNumbers(OUTPUT\_FILENAME, numbers, ARRAY\_SIZE);  printf("Numbers sorted and stored successfully in %s.\n", OUTPUT\_FILENAME);  return 0;  } |
|  |
|  |
|  |
| 15. **Bubble Sort:** Sort an array using the bubble sort algorithm. |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define ARRAY\_SIZE 10  #define INPUT\_FILENAME "random\_numbers.txt"  #define OUTPUT\_FILENAME "sorted\_numbers.txt"  void generateRandomNumbers(const char \*filename, int size) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < size; i++) {  int num = rand() % 100; // Generate random number between 0 and 99  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  void bubbleSort(int arr[], int size) {  int i,j;  for ( i = 0; i < size - 1; i++) {  for ( j = 0; j < size - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  // Swap arr[j] and arr[j+1]  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  }  void storeSortedNumbers(const char \*filename, int arr[], int size) {  int i;  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  for ( i = 0; i < size; i++) {  fprintf(file, "%d\n", arr[i]);  }  fclose(file);  }  int main() {  int size = ARRAY\_SIZE;  // Generate random numbers and store them in a file  generateRandomNumbers(INPUT\_FILENAME, size);  // Open the file containing generated numbers for manipulation  FILE \*inputFile = fopen(INPUT\_FILENAME, "r");  if (inputFile == NULL) {  perror("Error opening input file");  return EXIT\_FAILURE;  }  // Read the numbers into an array  int numbers[ARRAY\_SIZE],i;  for ( i = 0; i < ARRAY\_SIZE; i++) {  fscanf(inputFile, "%d", &numbers[i]);  }  // Close the input file  fclose(inputFile);  // Sort the array using bubble sort  bubbleSort(numbers, ARRAY\_SIZE);  // Store the sorted numbers in another file  storeSortedNumbers(OUTPUT\_FILENAME, numbers, ARRAY\_SIZE);  printf("Numbers sorted and stored successfully in %s.\n", OUTPUT\_FILENAME);  return 0;  } |
|  |
|  |
|  |
| 16. **Insertion Sort:** Sort an array using the insertion sort algorithm. |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define ARRAY\_SIZE 10  #define INPUT\_FILENAME "random\_numbers.txt"  #define OUTPUT\_FILENAME "sorted\_numbers.txt"  void generateRandomNumbers(const char \*filename, int size) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i;  for ( i = 0; i < size; i++) {  int num = rand() % 100; // Generate random number between 0 and 99  fprintf(file, "%d\n", num);  printf("%d\n", num);  }  fclose(file);  }  void insertionSort(int arr[], int size) {  int i;  for ( i = 1; i < size; i++) {  int key = arr[i];  int j = i - 1;  // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  arr[j + 1] = key;  }  }  void storeSortedNumbers(const char \*filename, int arr[], int size) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  int i;  for ( i = 0; i < size; i++) {  fprintf(file, "%d\n", arr[i]);  }  fclose(file);  }  int main() {  int size = ARRAY\_SIZE;  // Generate random numbers and store them in a file  generateRandomNumbers(INPUT\_FILENAME, size);  // Open the file containing generated numbers for manipulation  FILE \*inputFile = fopen(INPUT\_FILENAME, "r");  if (inputFile == NULL) {  perror("Error opening input file");  return EXIT\_FAILURE;  }  // Read the numbers into an array  int numbers[ARRAY\_SIZE],i;  for ( i = 0; i < ARRAY\_SIZE; i++) {  fscanf(inputFile, "%d", &numbers[i]);  }  // Close the input file  fclose(inputFile);  // Sort the array using insertion sort  insertionSort(numbers, ARRAY\_SIZE);  // Store the sorted numbers in another file  storeSortedNumbers(OUTPUT\_FILENAME, numbers, ARRAY\_SIZE);  printf("Numbers sorted and stored successfully in %s.\n", OUTPUT\_FILENAME);  return 0;  } |
|  |
|  |
|  |
|  |
| **Matrix Operations:** Perform matrix addition, subtraction, multiplication, and  transpose. |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define ROWS 3  #define COLS 3  void generateRandomMatrix(const char \*filename, int rows, int cols) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  perror("Error opening file for writing");  exit(EXIT\_FAILURE);  }  srand(time(NULL));  int i,j;  for (i = 0; i < rows; i++) {  for ( j = 0; j < cols; j++) {  int num = rand() % 100; // Generate random number between 0 and 99  fprintf(file, "%d ", num);  printf("%d ", num);  }  fprintf(file, "\n");  printf("\n");  }  fclose(file);  }  void readMatrix(const char \*filename, int matrix[ROWS][COLS], int rows, int cols) {  int i,j;  FILE \*file = fopen(filename, "r");  if (file == NULL) {  perror("Error opening file for reading");  exit(EXIT\_FAILURE);  }  for ( i = 0; i < rows; i++) {  for ( j = 0; j < cols; j++) {  if (fscanf(file, "%d", &matrix[i][j]) != 1) {  fprintf(stderr, "Error reading from file\n");  exit(EXIT\_FAILURE);  }  }  }  fclose(file);  }  void printMatrix(int matrix[ROWS][COLS], int rows, int cols) {  int i,j;  printf("Matrix:\n");  for ( i = 0; i < rows; i++) {  for ( j = 0; j < cols; j++) {  printf("%d ", matrix[i][j]);  }  printf("\n");  }  }  void addMatrices(int A[ROWS][COLS], int B[ROWS][COLS], int result[ROWS][COLS]) {  int i,j;  for ( i = 0; i < ROWS; i++) {  for ( j = 0; j < COLS; j++) {  result[i][j] = A[i][j] + B[i][j];  }  }  }  void subtractMatrices(int A[ROWS][COLS], int B[ROWS][COLS], int result[ROWS][COLS]) {  int i,j;  for ( i = 0; i < ROWS; i++) {  for ( j = 0; j < COLS; j++) {  result[i][j] = A[i][j] - B[i][j];  }  }  }  void multiplyMatrices(int A[ROWS][COLS], int B[ROWS][COLS], int result[ROWS][COLS]) {  int i,j,k;  for ( i = 0; i < ROWS; i++) {  for ( j = 0; j < COLS; j++) {  result[i][j] = 0;  for ( k = 0; k < COLS; k++) {  result[i][j] += A[i][k] \* B[k][j];  }  }  }  }  void transposeMatrix(int matrix[ROWS][COLS], int result[ROWS][COLS]) {  int i,j;  for ( i = 0; i < ROWS; i++) {  for ( j = 0; j < COLS; j++) {  result[j][i] = matrix[i][j];  }  }  }  int main() {  int matrixA[ROWS][COLS];  int matrixB[ROWS][COLS];  int result[ROWS][COLS];  // Generate random matrices and store them in files  generateRandomMatrix("matrixA.txt", ROWS, COLS);  generateRandomMatrix("matrixB.txt", ROWS, COLS);  // Read matrices from files  readMatrix("matrixA.txt", matrixA, ROWS, COLS);  readMatrix("matrixB.txt", matrixB, ROWS, COLS);  // Add matrices  addMatrices(matrixA, matrixB, result);  printf("Addition:\n");  printMatrix(result, ROWS, COLS);  // Subtract matrices  subtractMatrices(matrixA, matrixB, result);  printf("\nSubtraction:\n");  printMatrix(result, ROWS, COLS);  // Multiply matrices  multiplyMatrices(matrixA, matrixB, result);  printf("\nMultiplication:\n");  printMatrix(result, ROWS, COLS);  // Transpose matrix A  transposeMatrix(matrixA, result);  printf("\nTranspose of matrix A:\n");  printMatrix(result, ROWS, COLS);  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;  } |
|  |
|  |

. . . .. . . . . . . . . . ..

; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;