|  |  |
| --- | --- |
| **Name:** | **TRISHA RAO** |
| **Roll No:** | **372** |
| **USN:** | **01fe21BEC024** |
| **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. |
| 1. A variable is an identifier or a item which stores the data or a value,   A data type is a type of data that a variable can hold such as integer, float etc |
| 2. Explain the concept of data types in C programming. Discuss the different types of data types available in C. |
| A data types are used to define the type of data that a variable can hold.  Each data type specifies the size and format of the data that can be stored in memory |
| int: Used to store integer values. Typically takes 4 bytes of memory on most systems |
| float: Used to store floating-point numbers (decimal numbers). Takes 4 bytes of memory. |
| char: used to store single characters ,takes 1 b it of memory |
| double: Used to store double-precision floating-point numbers. Takes 8 bytes of memory. |
|  |
| 3.How are variables declared and initialized in C programming? Provide examples of variable declarations with different data types. |
| - Variables are declared and initialized in C programming using a specific syntax |
| EX- Variables are declared and initialized in C programming using a specific syntax |
| int age; // Declares an integer variable named age  float price; // Declares a floating-point variable named price  char grade; // Declares a character variable named grad |
|  |
| 4. Discuss the scope and lifetime of variables in C programming. What are global and local variables? |
| A- Scope**:** Refers to the program area where a variable is accessible. It's like a zone where the variable's name is recognized.  Lifetime**:** Determines how long the memory allocated for the variable remains valid during program execution. |
|  |
| Explain the concept of type casting in C programming. When is type casting necessary, and how is it performed? |
| A-Type casting in C lets you explicitly convert a variable from one data type to another. Imagine it as forcing a value into a different sized box. |
| Casting is done using the cast operator (data\_type) before the variable you want to convert. For example, (float)age converts age (int) to a float. Be cautious though, as casting can lead to data loss if the target type can't hold the original value's range. |
|  |
|  |
| Operators: |
| 1. Describe the purpose and usage of the ternary conditional operator (?:) in C programming. Provide an example demonstrating its usage |
| A-It is a concise way to write conditional expressions in C programming. It allows you to make decisions and choose between two values based on a condition |
|  |
| 2. Discuss the bitwise operators available in C programming. Explain their usage with suitable examples. |
| A-bitwise operators are used to perform operations on individual bits of integer operands  Bitwise OR (|): Sets a bit to 1 if at least one corresponding bit in either operand is 1. x | y: 1010 | 0110 = 1110  Bitwise AND (&): Sets a bit to 1 only if the corresponding bits in both operands are 1. x & y: 1010 & 0110 = 0010  Bitwise NOT (~):The bitwise NOT operator (~) performs a bitwise negation operation on each bit of its operand. It flips all bits (0s become 1s and 1s become 0s).  Left shift (<<): Shifts bits of the first operand to the left by the number of bits specified by the second operand. x << 2: 1010 << 2 = 101000  Right shift (>>): Shifts bits of the first operand to the right by the number of bits specified by the second operand. x >> 1: 1010 >> 1 = 0101 |
| 3. Explain the difference between the postfix and prefix increment operators (++) in C programming. Provide examples to illustrate. |
| * A- Prefix **(++)**: Increments the variable first, then uses the new value.   + Example: int x = 5; ++x; // x becomes 6 (increment happens before assignment) * Postfix **(++):** Uses the variable's current value first, then increments it.   + Example: int x = 5; int y = x++; // y becomes 5 (original value), then x becomes 6 (assignment uses original value, then increment happens) |
|  |
| 4. What is the significance of the logical AND (&&) and logical OR (||) operators in C programming? How are they used in conditional expressions? |
| * A- **Logical OR (||):** Represents "at least one condition must be true". The expression returns true if **any** operand evaluates to true. Evaluation stops as soon as a true operand is encountered (short-circuit evaluation). * **Logical AND (&&):** Represents "both conditions must be true". The expression returns true only if **all** operands evaluate to true. If the first operand is false, the second operand isn't even evaluated (short-circuit evaluation). |
| 5. Discuss the concept of operator precedence and associativity in C programming. Provide examples to demonstrate how they affect expression evaluation. |
| A-Operator precedence refers to the priority of operators in an expression. Operators with higher precedence are evaluated before operators with lower precedence.  Associativity refers to the order in which operators of the same precedence are evaluated. Operators can be left-associative or right-associative:  Left-associative: Operators are evaluated from left to right.  Right-associative: Operators are evaluated from right to left. |
| **Control Structures**: |
| 1.Describe the purpose and usage of the switch statement in C programming. How does it differ from the if-else statement? |
| A-It is used to make decisions based on the value of an expression. It provides a more structured and efficient way to handle multiple possible cases compared to using multiple if-else statements.  The switch statement is used when the value of an expression needs to be compared against multiple constant values.  The if-else statement is more versatile and can handle complex conditions involving relational operators, logical operators, and compound condition |
| 2. Explain the concept of nested control structures in C programming. Provide an example demonstrating nested if-else statements |
| A-Nested control structures in C allow you to create complex decision-making logic by embedding one control structure within another. It's like creating layers of conditions.  Example (Nested if-else):  int age = 20;  char initial = 'A';  if (age >= 18) {  printf(" adult.\n");  if (initial == 'A' || initial == 'E' || initial == 'I' || initial == 'O' || initial == 'U') {  printf(" vowel.\n");  } else {  printf(" consonant.\n");  }  } else {  printf("You are not an adult.\n");  } |
|  |
| 3. Discuss the role of the break and continue statements in loop control in C programming. Provide examples to illustrate their usage.  **A**.Purpose of break: Terminates the loop entirely, exiting the loop's body as soon as it's encountered.  }  Purpose of continue**:** Skips the current iteration of the loop and jumps to the beginning of the next iteration. The remaining code within the current iteration is not executed. |
|  |
| 4. **4.** What are the advantages of using the for loop over the while loop in C programming? Provide examples comparing the two.  **A**. Advantages  Clarity and Readability:   * for loops explicitly combine initialization, condition check, and increment/decrement in a single line, making the loop's purpose clearer at a glance.   Conciseness:   * This combined structure often leads to more concise code compared to while loops, especially when all three components (initialization, condition, increment) are } |
|  |
| 5. **5.** Explain the concept of short-circuit evaluation in C programming. How does it affect the evaluation of logical expressions in if statements?  **A.** Short-circuit evaluation is an optimization technique used in C programming for logical operators (&& - AND, || - OR). It ensures expressions are evaluated only as far as necessary to determine the final outcome.  Benefits:  1.Improves efficiency by avoiding unnecessary calculations, especially when dealing with functions that might have side effects.  2.Can enhance code readability by allowing you to write conditions where the outcome of the second operand depends on the first. |
|  |
| **Functions:** |
| 1. Describe the purpose and structure of a function prototype in C programming. Why is it necessary to declare function prototypes?  A-A function prototype acts like a blueprint or an announcement for a function. It provides essential information about the function to the compiler **before** the actual function definition is encountered.   * Code Organization: Prototypes improve code readability and maintainability by documenting function details in a separate header file. This allows you to use functions across multiple source files without worrying about the definition order. * Type Checking: The compiler uses the prototype to ensure the function call matches the declared parameters and return type. This helps catch errors like passing incorrect data types or missing arguments during compilation. |
|  |
| 2. Explain the difference between call by value and call by reference in C programming. Provide examples to illustrate both concepts. |
| Call by Value:   * A copy of the actual argument's value is passed to the function. * Changes made to the parameter inside the function do not affect the original variable in the calling code.   call by Reference:   * The address (memory location) of the actual argument is passed to the function. * Modifications made to the parameter through the address directly affect the original variable. |
|  |
| 3. **3**.Discuss the concept of recursion in C programming. Provide an example of a recursive function and explain how it works. |
| Recursion in C programming involves a function calling itself. It's a technique for solving problems by breaking them down into smaller, self-similar subproblems.  Example: Factorial Function  int factorial(int n) {  if (n == 0) {  return 1;  } else {  return n \* factorial(n - 1);  }  } |
|  |
| 5, Describe the role of function parameters and arguments in C programming. How are function arguments passed to parameters |
| functions operate on data but often need input from the calling code. This is where function parameters and arguments come into play:  Arguments**:** These are the actual values passed to the function when it's called. .They are listed within parentheses after the function name in the call statement.  Parameters: These are variables declared within the function definition. They act like placeholders that receive the values passed during the function call. |
| **Arrays:** |
| 1. **1.** Explain the concept of arrays in C programming. How are arrays declared and initialized? |
| **1.** Explain the concept of arrays in C programming. How are arrays declared and initialized?  rrays in C act like collections of elements of the same data type. Imagine them as a fixed-size box where each slot stores a value. You declare them by specifying the data type, array name, and size enclosed in square brackets []. Initialization (assigning values) can be done during declaration or later. |
|  |
| 2. Discuss the difference between a one-dimensional array and a multidimensional array in C programming. Provide examples of both |

|  |
| --- |
| One-dimensional arrays hold a single list of elements.  Example: int numbers[5] = {1, 2, 3, 4, 5};  Multi-dimensional arrays (like 2D arrays) represent a grid-like structure with rows and columns.  Example: int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}}; |
|  |
| 3. Describe the process of accessing array elements in C programming. How are array indices used to access elements? |
| To access elements in a C array, you use the array name followed by the element's index enclosed in square brackets []. The index starts from 0, so the first element has index 0, the second has index 1, and so on. |
|  |
| 4. What 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 (\0) acts as a special sentinel marking the string's end. It's a non-printable character with an ASCII value of 0. |
|  |
| 5. Explain the concept of dynamic memory allocation for arrays in C programming. How are dynamic arrays allocated and deallocated |
| Dynamic memory allocation offers flexibility to create arrays with sizes determined at runtime. Here's a basic overview  1.Functions like malloc (allocate memory) and calloc (allocate and initialize) are used to request memory blocks of a desired size (in bytes) based on the number of elements needed.  2.A pointer variable is used to store the memory address returned by malloc or calloc. This pointer acts like a reference to the dynamically allocated array.  3.After use, the allocated memory must be explicitly released using free to prevent memory leaks. This frees the memory block back to the system. |
| **Pointers:** |
| **1**.Describe the purpose and usage of pointers in C programming. How are pointers declared and initialized |
| Pointers are variables that store memory addresses. They act like signposts pointing to locations in memory where other data resides.  Pointers are declared by specifying an asterisk (\*) before the data type they point to, followed by the pointer variable name. Initialization can involve assigning the address of a variable (using the & operator) or NULL |
|  |
| **2.** Explain the concept of pointer arithmetic in C programming. Provide examples to illustrate addition and subtraction operations on pointers |
| Addition  int arr[5] = {1, 2, 3, 4, 5};  int \*ptr = arr; // ptr points to the first element (arr[0])  ptr++; // Moves ptr one element forward (points to arr[1])  printf("Value at ptr: %d\n", \*ptr); // Output: 2 (value at arr[1])  Here, ptr++ increments the pointer by the size of an integer (usually 4 bytes), effectively moving it to the next element in the array.  Subtraction  int numbers[4] = {10, 20, 30, 40};  int \*p1 = &numbers[2]; // p1 points to numbers[2]  int \*p2 = numbers; // p2 points to numbers[0]  int difference = p1 - p2; // Calculates the number of elements between p1 and p2  printf("Difference in elements: %d\n", difference); // Output: 2 (p1 points to the 3rd element, 2 elements ahead of p2) |
| **3.** Discuss the difference between pass by value and pass by reference in function arguments using pointers in C programming. Provide examples to illustrate both approaches |
| C primarily uses pass by value for function arguments. This means a copy of the argument's value is passed to the function. Changes made inside the function only affect the copy, not the original variable. For example, a function trying to increment an integer passed by value would only modify the copy.Pass by reference, achieved using pointers, allows functions to modify the original variable. |
|  |
| 4. Describe the concept of NULL pointers in C programming. How are NULL pointers used and checked for in programs? |
| A null pointer in C is a special pointer variable that doesn't hold a valid memory address. It essentially points to "nothing." We use the NULL macro to assign a null pointer.Null pointers are used to indicate uninitialized pointers or the end of a linked list. |
|  |
| **5.** Explain the role of pointers in dynamic memory allocation in C programming. How are pointers used to allocate and deallocate memory dynamically? |
| 1. In dynamic memory allocation, pointers act as handles. Functions like malloc and calloc return pointers to the allocated memory block. These pointers allow us to access and manipulate the dynamically allocated data. We use free with the pointer to release the memory when it's no longer needed, preventing memory leaks   **STRINGS:** |
| **1.** Discuss the concept of strings in C programming. How are strings represented and manipulated in C? |
| strings are simply character arrays terminated by a null character (\0).  String manipulation functions like strcpy, strcat, and strlen work with pointers to manage these character arrays. |
| **2.** Describe common string manipulation functions available in the C standard library. Provide examples of functions like strlen, strcpy, strcat, and strcmp |
| **A.** C provides a set of powerful tools in the <string.h> header for manipulating strings. These functions make working with text data more efficient. The strlen function calculates the length of a string, excluding the null terminator that marks the string's end. You can use strcpy to copy one string to another, but be cautious! strcpy doesn't check if the destination string has enough space, which can lead to program crashes. For safer copying, consider strncpy where you specify the maximum number of characters to copy. Similar caution applies to strcat which appends strings but can also cause buffer overflows. Safer alternatives include strncat with a limit on characters appended or pre-allocating enough space in the destination string. Finally, the strcmp function helps compare strings, returning 0 if they are identical. |
| **4.**Discuss the concept of string tokenization in C programming. How are strings split into tokens using delimiter characters? |
| **A**. String tokenization in C involves splitting a string into smaller substrings (tokens) based on separator characters ( delimiters). The strtok function (from <string.h>) is commonly used for this purpose. It modifies the original string, keeping track of its position with subsequent calls. |
|  |
| **5.** Explain the importance of null-terminated strings in C programming. How does the null character signify the end of a string? |
| Null-terminated strings are fundamental in C for managing character arrays as strings. The null character (\0), with a value of zero, acts as a silent sentinel at the end of the string |
|  |
| • Structures and Unions: |
| 1. Describe the purpose and usage of structures in C programming. How are structures declared and accessed? |
| A.Structures group variables of potentially different data types under a single user-defined type. This allows us to represent real-world entities with various attributes. |
|  |
| 2. Discuss the concept of structure members in C programming. How are individual members of a structure accessed and modified? |
| Structure members are individual variables of various data types bundled within a structure. They represent specific attributes of the entity the structure defines. We access and modify them using the dot (.) operator after the structure variable name followed by the member's name |
| 3.Explain the difference between structures and unions in C programming. When would you choose one over the other? |
| Structures group different data types together under one name, allowing access to each member individually. Unions, on the other hand, share the same memory space for all members, allowing only one member to be active at a time. |
|  |
| **4.** Describe the concept of nested structures in C programming. How are structures within structures defined and accessed? |
| **A**. Nested structures allow you to create complex data hierarchies in C. One structure can contain another structure as a member, enabling you to model entities with composed attributes. You define nested structures by placing a structure declaration within another structure's member definition. To access members of a nested structure, you use the dot (.) operator chained with both the outer and inner structure member names. |
|  |
| **5**. Discuss the concept of typedef in C programming. How is typedef used to define custom data types, including structures and unions? |
| Typedef is a keyword used to create aliases or alternative names for existing data types, including primitive data types, structures, unions, and pointers. It allows user to define custom data types with more descriptive names, improving code readability and maintainability. |
|  |
| **File handling:** |
| **1.** Explain the concept of file handling in C programming. How are files opened, read from, and written to using standard file handling functions? |
| File handling in C programming refers to the process of working with files on a computer's file system. It involves operations such as opening files, reading data from files, writing data to files, closing files, and performing other file-related tasks. fopen to open a file in a specific mode (read, write, append). Then, fread reads data from the file into a buffer, and fwrite writes data from a buffer to the file. Finally, fclose closes the file, releasing resources. |
| **2**. Describe the role of file pointers in C programming. How are file pointers used to navigate and manipulate files? |
| file pointers (**FILE \***) play a crucial role in file handling operations. A file pointer is a special type of pointer that is used to navigate and manipulate files. |
|  |
| **3**. Discuss the difference between text files and binary files in C programming. How are they opened and processed differently? |
| . In C, text files store human-readable characters using ASCII encoding. Binary files store raw data (numbers, images, etc.) as sequences of bytes.(1).**Text Files:** Opened in text mode ("r", "w", "a") with functions like fgets (read) and fputs (write) that handle newline characters appropriately.(**2).Binary Files:** Opened in binary mode ("rb", "wb", "ab") using fread and fwrite to read/write raw byte sequences without newline interpretation. |
|  |
| **4**. Explain the purpose of file modes in C programming. Provide examples of different file modes like "r", "w", "a", etc. |
| In C programming, file modes determine how files are opened and accessed. They dictate whether a file is opened for reading ("r"), writing ("w"), or appending ("a"). Additional modes include "r+" for reading and writing, "w+" for reading and writing, and "a+" for reading and appending |
|  |
| **5.** Describe error handling techniques in file operations in C programming. How are errors detected and handled when working with files? |
| Error handling in file operations in C programming is essential to detect and handle potential issues that may occur during file handling operations. Errors can occur due to various reasons such as file not found, insufficient permissions, disk full, and so on. Proper error handling techniques help ensure that your program can gracefully handle such situations and provide appropriate feedback to the user. |

**Part- B**

|  |
| --- |
| 1. **Hello world** |
| Code  #include<stdio.h>  main()  {  printf("hello world");  } |
| Output |

|  |
| --- |
| Factorial: Calculate the factorial of a given number. |
| Code  #include <stdio.h>  int main(){  int i, f = 1, num;  printf("Input the number : ");  scanf("%d", &num  for(i = 1; i <= num; i++)  f = f \* i; .  printf("The Factorial of %d is: %d\n", num, f);  return 0;  } |
| Output    3.Prime number  #include <stdio.h>  void checkPrime(int N)  {  int flag = 1;  for (int i = 2; i <= N/2; i++) {  if (N % i == 0) {  flag = 0;  break;  }  }  if (flag) {  printf("The number %d is a Prime Number\n", N);  }  else {  printf("The number %d is not a Prime Number\n", N);  }  return;  }  int main()  {  int N = 546;  checkPrime(N);  return 0; |
| **4.Fibonacci Series**  #include<stdio.h>  void main()  {  int i,j,n,k,l;  printf("ENTER THE LENGHT OF THE SERIES:");  scanf("%d",&n);  if(n==2)  {  printf("\n FIBONACCI SERIES:0,0");  exit(0);  }  printf("\n FIBONACCI SERIES:0,1,");  i=0;  j=1;  for(k=0;k<n-2;k++)  {  l=i+j;  printf("%d",l);  if(k!=n-3)  printf(",");  i=j;  j=l;  }  } |
|  |
| **5.Sum of digits**  #include<stdio.h>  void main()  {  int i,j=0,k,n;  printf("ENTER THE SIZE OF SERIES :");  scanf("%d",&n);  printf("\nSERIES:");  for(i=1;i<=n;i++)  {  k=i\*i;  j=j+k;  printf("%d + ",k);  }  printf("\nSUM OF SERIES= %d",j);  } |
|  |
| 6**6. Reverse a Number**  **Code**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int main()  {  int num, reversed = 0, digit;  FILE \*fp;  srand(time(NULL));  printf("Enter a number (or 0 to use a random number): ");  scanf("%d", &num);  if (num == 0) {  num = rand() % 100000 + 1;  printf("Using random number: %d\n", num);  }  fp = fopen("reversed\_number.txt", "w");  if (fp == NULL) {  perror("Error opening file");  return 1;  }  while (num != 0) {  digit = num % 10;  reversed = reversed \* 10 + digit;  num /= 10;  }  fprintf(fp, "%d\n", reversed);  fclose(fp);  printf("Reversed number written to reversed\_number.txt\n");  } |
|  |
| 7.PALINDROME  #include <stdio.h>  int main() {  int n, reversed = 0, remainder, original;  printf("Enter an integer: ");  scanf("%d", &n);  original = n;    while (n != 0) {  remainder = n % 10;  reversed = reversed \* 10 + remainder;  n /= 10;  }  if (original == reversed)  printf("%d is a palindrome.", original);  else  printf("%d is not a palindrome.", original);  return 0;  } |
|  |
| **8. Area of Shapes**  #include <stdio.h>  int main()  {  int l = 10, b = 10;  printf("Area of rectangle is : %d", l \* b);  printf("\nPerimeter of rectangle is : %d", 2 \* (l + b));  return 0;  } |
|  |
| **9. . Simple Calculator:**    **Code**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int main()  {  FILE \*outputFile;  char operations[] = {'+', '-', '\*', '/'};  int num1, num2, result;  char operation;  srand(time(NULL));  outputFile = fopen("calculator\_output.txt", "w");  if (outputFile == NULL) {  perror("Error opening output file");  exit(EXIT\_FAILURE);  }  num1 = rand() % 100;  num2 = rand() % 100;  operation = operations[rand() % 4];  switch (operation) {  case '+':  result = num1 + num2;  break;  case '-':  result = num1 - num2;  break;  case '\*':  result = num1 \* num2;  break;  case '/':  if (num2 == 0) {  fprintf(outputFile, "Error: Division by zero\n");  fclose(outputFile);  exit(EXIT\_FAILURE);  }  result = num1 / num2;  break;  }  fprintf(outputFile, "Number 1: %d\n", num1);  fprintf(outputFile, "Number 2: %d\n", num2);  fprintf(outputFile, "Operation: %c\n", operation);  fprintf(outputFile, "Result: %d\n", result);  fclose(outputFile);  printf("Calculation complete. Check calculator\_output.txt for results.\n");  } |
|  |
| 10. **10. Array Operations**  **Code**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define ARRAY\_SIZE 10  void generateRandomNumbers(int arr[], int size) {  for (int i = 0; i < size; i++)  {  arr[i] = rand() % 100;  }  }  int findLargest(int arr[], int size) {  int largest = arr[0];  for (int i = 1; i < size; i++) {  if (arr[i] > largest) {  largest = arr[i];  }  }  return largest;  }  int findSmallest(int arr[], int size) {  int smallest = arr[0];  for (int i = 1; i < size; i++) {  if (arr[i] < smallest) {  smallest = arr[i];  }  }  return smallest;  }  int findSum(int arr[], int size) {  int sum = 0;  for (int i = 0; i < size; i++) {  sum += arr[i];  }  return sum;  }  double findAverage(int sum, int size) {  return (double)sum / size;  }  int main() {  int numbers[ARRAY\_SIZE];  int largest, smallest, sum;  double average;  FILE \*outputFile;  srand(time(NULL));  generateRandomNumbers(numbers, ARRAY\_SIZE);  largest = findLargest(numbers, ARRAY\_SIZE);  smallest = findSmallest(numbers, ARRAY\_SIZE);  sum = findSum(numbers, ARRAY\_SIZE);  average = findAverage(sum, ARRAY\_SIZE);  outputFile = fopen("array\_stats.txt", "w");  if (outputFile == NULL) {  perror("Error opening output file");  exit(EXIT\_FAILURE);  }  fprintf(outputFile, "Largest: %d\n", largest);  fprintf(outputFile, "Smallest: %d\n", smallest);  fprintf(outputFile, "Sum: %d\n", sum);  fprintf(outputFile, "Average: %.2f\n", average);  fclose(outputFile);  printf("Array statistics calculated. Check array\_stats.txt for results.\n");  } |
|  |
|  |
| **11. String Operations**  #include <stdio.h>  #include <string.h>  void concatenateStrings(char \*dest, const char \*src) {  strcat(dest, src);  }  void copyString(char \*dest, const char \*src) {  strcpy(dest, src);  }  int compareStrings(const char \*str1, const char \*str2) {  return strcmp(str1, str2);  }  int main() {  char str1[100] = "Hello, ";  char str2[] = "world!";  char str3[100];  int result;  concatenateStrings(str1, str2);  printf("Concatenated string: %s\n", str1);    copyString(str3, str1);  printf("Copied string: %s\n", str3);  result = compareStrings(str1, str3);  if (result == 0) {  printf("Strings are equal.\n");  } else {  printf("Strings are not equal.\n");  }  return 0;  } |
|  |
| **12.Linear search**  **Code**  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <time.h>  #define ARRAY\_SIZE 10  int linearSearch(int arr[], int size, int key)  {  for (int i = 0; i < size; i++) {  if (arr[i] == key) {  return i;  }  return -1;  }  int main() {  FILE \*outputFile;  int arr[ARRAY\_SIZE];  int searchKey, result;  srand(time(NULL));  for (int i = 0; i < ARRAY\_SIZE; i++) {  arr[i] = rand() % 100;  }  outputFile = fopen("array\_elements.txt", "w");  if (outputFile == NULL) {  perror("Error opening output file");  exit(EXIT\_FAILURE);  }  fprintf(outputFile, "Array elements:\n");  for (int i = 0; i < ARRAY\_SIZE; i++) {  fprintf(outputFile, "%d ", arr[i]);  }  fprintf(outputFile, "\n");  fclose(outputFile);    printf("Enter the element to search for: ");  scanf("%d", &searchKey);  result = linearSearch(arr, ARRAY\_SIZE, searchKey);  if (result != -1) {  printf("Element %d found at index %d.\n", searchKey, result);  } else {  printf("Element %d not found in the array.\n", searchKey);  }  } |
| **13.Binary search**  code  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <time.h>  #define ARRAY\_SIZE 10  int cmpfunc(const void \*x const void \*y)  {  return (\*(int \*)x- \*(int \*)y);  }  int binarySearch(int arr[], int size, int tar) {  int left = 0, right = size - 1;  while (left <= right) {  int mid = left + (right - left) / 2;  if (arr[mid] == target) {  return mid;  }  if (arr[mid] < target) {  left = mid + 1;  } else {  right = mid - 1;  }  }  return -1;  }  int main() {  int arr[ARRAY\_SIZE];  int target, index;  FILE \*outputFile;  srand(time(NULL));  for (int i = 0; i < ARRAY\_SIZE; i++) {  arr[i] = rand() % 100;  }  qsort(arr, ARRAY\_SIZE, sizeof(int), cmpfunc);    outputFile = fopen("binary\_search\_results.txt", "w");  if (outputFile == NULL) {  perror("Error opening output file");  exit(EXIT\_FAILURE);  }  target = rand() % 100;  printf("Searching for target: %d\n", target);  index = binarySearch(arr, ARRAY\_SIZE, target);    if (index != -1) {  fprintf(outputFile, "Target %d found at index %d in the sorted array.\n", target, index);  } else {  fprintf(outputFile, "Target %d not found in the sorted array.\n", target);  }    fclose(outputFile);  printf("Binary search complete. Check binary\_search\_results.txt for results.\n");  } |
| **14. Selection Sort**  #include <stdio.h>  **int** main()  {  **int** a[100], n, i, j, position, swap;  printf("Enter number of elementsn");  scanf("%d", &n);  printf("Enter %d Numbersn", n);  **for** (i = 0; i < n; i++)  scanf("%d", &a[i]);  **for**(i = 0; i < n - 1; i++)  {  position=i;  **for**(j = i + 1; j < n; j++)  {  **if**(a[position] > a[j])  position=j;  }  **if**(position != i)  {  swap=a[i];  a[i]=a[position];  a[position=swap;  }  }  printf("Sorted Array:n");  **for**(i = 0; i < n; i++)  printf("%dn", a[i]);  **return** 0;  } |
|  |
| **16.Insertion sort**  Code  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void insertionSort(int arr[], int n) {  int i, key, j;  for (i = 1; i < n; i++) {  key = arr[i];  j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key;  }  }  int main() {  FILE \*outputFile;  int arr[10];  srand(time(NULL));  for (int i = 0; i < 10; i++) {  arr[i] = rand() % 100;  }  insertionSort(arr, 10);  outputFile = fopen("sorted\_array.txt", "w");  if (outputFile == NULL) {  perror("Error opening output file");  exit(EXIT\_FAILURE);  }  fprintf(outputFile, "Sorted Array:\n");  for (int i = 0; i < 10; i++) {  fprintf(outputFile, "%d ", arr[i]);  }  fclose(outputFile);  printf("Sorting complete. Check sorted\_array.txt for results.\n");  } |
|  |
| **17. Matrix Operations**  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  void readMatrixFromFile(const char \*filename, int matrix[MAX\_SIZE][MAX\_SIZE], int \*rows, int \*cols) {  FILE \*file = fopen(filename, "r");  if (file == NULL) {  printf("Error opening file %s.\n", filename);  exit(1);  }  fscanf(file, "%d %d", rows, cols);  for (int i = 0; i < \*rows; i++) {  for (int j = 0; j < \*cols; j++) {  fscanf(file, "%d", &matrix[i][j]);  }  }  fclose(file);  }  void printMatrix(int matrix[MAX\_SIZE][MAX\_SIZE], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  printf("%d ", matrix[i][j]);  }  printf("\n");  }  }  void addMatrices(int matrix1[MAX\_SIZE][MAX\_SIZE], int matrix2[MAX\_SIZE][MAX\_SIZE], int result[MAX\_SIZE][MAX\_SIZE], int rows, int cols) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = matrix1[i][j] + matrix2[i][j];  }  }  }  void multiplyMatrices(int matrix1[MAX\_SIZE][MAX\_SIZE], int matrix2[MAX\_SIZE][MAX\_SIZE], int result[MAX\_SIZE][MAX\_SIZE], int rows1, int cols1, int cols2) {  for (int i = 0; i < rows1; i++) {  for (int j = 0; j < cols2; j++) {  result[i][j] = 0;  for (int k = 0; k < cols1; k++) {  result[i][j] += matrix1[i][k] \* matrix2[k][j];  }  }  }  }  int main() {  int matrix1[MAX\_SIZE][MAX\_SIZE], matrix2[MAX\_SIZE][MAX\_SIZE], result[MAX\_SIZE][MAX\_SIZE];  int rows1, cols1, rows2, cols2;  readMatrixFromFile("matrix1.txt", matrix1, &rows1, &cols1);  readMatrixFromFile("matrix2.txt", matrix2, &rows2, &cols2);  if (rows1 != rows2 || cols1 != cols2) {  printf("Matrices are not of the same dimensions, addition and multiplication are not possible.\n");  return 1;  }  printf("Matrix 1:\n");  printMatrix(matrix1, rows1, cols1);  printf("\n");  printf("Matrix 2:\n");  printMatrix(matrix2, rows2, cols2);  printf("\n");  printf("Matrix Addition:\n");  addMatrices(matrix1, matrix2, result, rows1, cols1);  printMatrix(result, rows1, cols1);  printf("\n");  if (cols1 == rows2) {  printf("Matrix Multiplication:\n");  multiplyMatrices(matrix1, matrix2, result, rows1, cols1, cols2);  printMatrix(result, rows1, cols2);  } else {  printf("Matrix multiplication is not possible due to mismatched dimensions.\n");  }  return 0;  } |
| PART C |
| Basic operations |
| Code  #include <stdio.h>  #include <stdlib.h>  struct Node  {  int data;  struct Node\* next;  };  void insertAtBeginning(struct Node\*\* head, int newData);  void insertAtEnd(struct Node\*\* head, int newData);  void deleteNode(struct Node\*\* head, int key);  void printList(struct Node\* head);  int main() {  struct Node\* head = NULL;  insertAtBeginning(&head, 3);  insertAtBeginning(&head, 7);  insertAtBeginning(&head, 9);  printf("Linked list after inserting at the beginning: ");  printList(head);  printf("\n");  insertAtEnd(&head, 5);  printf("Linked list after inserting at the end: ");  printList(head);  printf("\n");  deleteNode(&head, 7);  printf("Linked list after deleting node with value 7: ");  printList(head);  printf("\n");  return 0;  }  void insertAtBeginning(struct Node\*\* head, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = newData;  newNode->next = \*head;  \*head = newNode;  }  void insertAtEnd(struct Node\*\* head, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* last = \*head;  newNode->data = newData;  newNode->next = NULL;  if (\*head == NULL) {  \*head = newNode;  return;  }  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  void deleteNode(struct Node\*\* head, int key) {  struct Node\* temp = \*head;  struct Node\* prev = NULL;  if (temp != NULL && temp->data == key) {  \*head = temp->next;  free(temp);  return;  }  while (temp != NULL && temp->data != key) {  prev = temp;  temp = temp->next;  }  if (temp == NULL) {  return;  }  prev->next = temp->next;  free(temp);  }  void printList(struct Node\* head) {  struct Node\* temp = head;  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  }    2)STUDENT DETAILS  #include <stdio.h>  #include <stdlib.h>  struct Node  {  int rollno;  char name[25];  struct Node \*next;  };  struct Node \*getNode()  {  return malloc(sizeof(struct Node));  }  void freeNode(struct Node \*p)  {  free(p);  }  void readInfo(struct Node \*p)  {  printf("\nEnter student details roll and name: \n");  scanf("%d%s",&p->rollno,p->name);  }  void displayInfo(struct Node \*p)  {  printf("\nRollno : %d Name : %s",p->rollno,p->name);  }  struct Node \*insertFront(struct Node \*first)  {  struct Node \*nw;  nw = getNode();  readInfo(nw);  nw->next = first;  return nw;  };  struct Node \*insertRear(struct Node \*first)  {  struct Node \*nw,\*last;  nw = getNode();  readInfo(nw);  nw->next = NULL;  if(!first)  {  return nw;  }  else  {  for(last=first;last->next != NULL; last=last->next);  last->next = nw;  }  return first;  };  void displayList(struct Node \*cur)  {  printf("\n--------------------LIST-------------------------------------");  for(;cur != NULL;cur=cur->next)  {  displayInfo(cur);  }  }  int main()  {  int ch;  struct Node \*first = NULL;  do{  printf("\n0->Exit 1->Insert Front 2->Insert Rear 3->Display list");  scanf("%d",&ch);  switch(ch)  {  case 0: break;  case 1: first = insertFront(first);  break;  case 2: first = insertRear(first);  break;  case 3: displayList(first);  break;  }  }while(ch!=0);  return 0;  } |
| Single circular list |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  typedef struct student {  char name[20];  int r\_num;  struct student \*next;  } NODE;  NODE \*init(NODE \*head) {  head = NULL;  return head;  }  NODE \*create\_node(void) {  NODE \*temp;  temp = (NODE  \*)malloc(sizeof(NODE));  if (temp == NULL) {  printf("Insufficient  memory\n");  return 0;  }  temp->next = NULL;  return temp;  }  NODE \*read\_details(NODE \*t) {  printf("Enter the name and  registration number\n");  scanf("%s%d", t->name, &t-  >r\_num);  return t;  }  NODE \*insert\_front(NODE \*head,  NODE \*t) {  if (head == NULL) {  head = t;  t->next = t; // Point to itself to  form a circular loop  } else {  t->next = head->next;  head->next = t;  }  return head;  }  NODE \*insert\_end(NODE \*head,  NODE \*t) {  if (head == NULL) {  head = t;  t->next = t; // Point to itself to  form a circular loop  } else {  t->next = head->next;  head->next = t;  head = t; // Move the head to  the newly inserted node to  maintain circularity  }  return head;  }  int count(NODE \*head) {  int cnt = 0;  if (head == NULL)  return cnt;  else {  NODE \*cur = head->next;  cnt = 1;  while (cur != head) {  cnt++;  cur = cur->next;  }  }  return cnt;  }  int search(NODE \*head, char  name[20]) {  if (head == NULL)  return 0;  NODE \*cur = head->next;  while (cur != head) {  if (strcmp(cur->name, name)  == 0) {  return 1;  }  cur = cur->next;  }  return 0;  }  void display(NODE \*head) {  if (head == NULL) {  printf("Empty list\n");  } else {  NODE \*cur = head->next;  printf("\tSTUDENT\_NAME\t  REGISTRATION\_NUMBER\n");  while (cur != head) {  printf("\t%s\t\t%d\n", cur-  >name, cur->r\_num);  cur = cur->next;  }  }  }  NODE  \*insert\_specific\_name(NODE  \*head, NODE \*t, char name[10]) {  if (head == NULL) {  printf("Empty list\n");  } else if (head->next == head) {  if (strcmp(head->name,  name) == 0) {  t->next = head->next;  head->next = t;  } else {  printf("Name not  found\n");  }  } else {  NODE \*cur = head->next;  int f = 0;  while (cur != head) {  if (strcmp(cur->name,  name) == 0) {  t->next = cur->next;  cur->next = t;  f = 1;  break;  }  cur = cur->next;  }  if (f == 0) {  printf("Not found\n");  }  }  return head;  }  NODE \*delete\_front(NODE \*head)  {  if (head == NULL) {  printf("Empty List\n");  } else if (head->next == head) {  free(head);  head = NULL;  } else {  NODE \*cur = head->next;  head->next = cur->next;  free(cur);  }  return head;  }  NODE \*delete\_end(NODE \*head) {  if (head == NULL) {  printf("Empty list\n");  } else if (head->next == head) {  free(head);  head = NULL;  } else {  NODE \*cur = head->next;  NODE \*prev = NULL;  while (cur->next != head) {  prev = cur;  cur = cur->next;  }  prev->next = head;  free(cur);  }  return head;  }  int main() {  int ch, cnt;  NODE \*head, \*t;  head = init(head);  while (1) {  printf("\n\n 1. Insert front\n  2. Insert end\n 3. Count\n 4.  Search\n 5. Display\n 6. Insert  specific\n 7. Delete front\n 8.  Delete end\n 9. Exit\n");  printf("\n\nEnter the  choice\n");  scanf("%d", &ch);  switch (ch) {  case 1:  t = create\_node();  t = read\_details(t);  head = insert\_front(head,  t);  break;  case 2:  t = create\_node();  t = read\_details(t);  head = insert\_end(head, t);  break;  case 3:  cnt = count(head);  printf("\n Total number of  counts = %d\n", cnt);  break;  case 4:  if (head == NULL)  printf("Empty list\n");  else {  int Search;  char name[20];  printf("Enter the name to  be searched\n");  scanf("%s", name);  Search = search(head,  name);  if (Search == 1)  printf("Search is  successful\n");  else  printf("Search is  unsuccessful\n");  }  break;  case 5:  display(head);  break;  case 6:  if (head == NULL)  printf("Empty list\n");  else {  char name[10];  printf("Enter the name of  the student:\n");  scanf("%s", name);  t = create\_node();  t = read\_details(t);  head =  insert\_specific\_name(head, t,  name);  }  break;  case 7:  head = delete\_front(head);  break;  case 8:  head = delete\_end(head);  break;  case 9:  exit(0);  }  }  return 0;  } |
| 4.employe details |
| #include <stdio.h>  #include <stdlib.h>  struct Node  {  int empid;  char name[25];  struct Node \*next;  };  struct Node \*getNode()  {  return malloc(sizeof(struct Node));  }  void freeNode(struct Node \*p)  {  free(p);  }  void readInfo(struct Node \*p)  {  printf("\nEnter employee's details employee ID and Name: \n");  scanf("%d%s",&p->empid,p->name);  }  void displayInfo(struct Node \*p)  {  printf("\nEmploye ID : %d Name : %s",p->empid,p->name);  }  struct Node \*insertFront(struct Node \*first)  {  struct Node \*nw;  nw = getNode();  readInfo(nw);  nw->next = first;  return nw;  };  struct Node \*insertRear(struct Node \*first)  {  struct Node \*nw,\*last;  nw = getNode();  readInfo(nw);  nw->next = NULL;  if(!first)  {  return nw;  }  else  {  for(last=first;last->next != NULL; last=last->next);  last->next = nw;  }  return first;  };  void displayList(struct Node \*cur)  {  printf("\n--------------------LIST-------------------------------------");  for(;cur != NULL;cur=cur->next)  {  displayInfo(cur);  }  printf("\n-------------------------------------------------------------");  }  struct Node \*deleteFront(struct Node \*first)  {  struct Node \*tmp;  if(first==NULL)  printf("EMPTY");  else{  tmp=first;  printf("Deleted Node:\n");  displayInfo(first);  first=first->next;  freeNode(tmp);  }  return(first);  };  int main()  {  int ch;  struct Node \*first = NULL;  do{  printf("\n0->Exit 1->Insert Front 2->Insert Rear 3->Delete front 4->displayList");  scanf("%d",&ch);  switch(ch)  {  case 0:break;  case 1:first = insertFront(first);  break;  case 2:first = insertRear(first);  break;  case 3:first=deleteFront(first);  break;  case 4:displayList(first);  break;  }  }while(ch!=0);  return 0;  } |
|  |
| **5.Reversing the link list**  Code  #include <stdio.h>  #include <stdlib.h>  struct Node  {  int data;  struct Node\* next;  };  struct Node\* createNode(int data);  void insertAtEnd(struct Node\*\* head, int newData);  void printList(struct Node\* head);  struct Node\* reverseList(struct Node\* head);  int main() {  struct Node\* head = NULL;  insertAtEnd(&head, 1);  insertAtEnd(&head, 2);  insertAtEnd(&head, 3);  insertAtEnd(&head, 4);  insertAtEnd(&head, 5);  printf("Original linked list: ");  printList(head);  printf("\n");  head = reverseList(head);  printf("Reversed linked list: ");  printList(head);  printf("\n");  return 0;  }  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed!\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  void insertAtEnd(struct Node\*\* head, int newData) {  struct Node\* newNode = createNode(newData);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* last = \*head;  while (last->next != NULL) {  last = last->next;  }  last->next = newNode;  }  void printList(struct Node\* head) {  struct Node\* temp = head;  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  }  struct Node\* reverseList(struct Node\* head) {  struct Node\* prev = NULL;  struct Node\* current = head;  struct Node\* next = NULL;  while (current != NULL) {  next = current->next;  current->next = prev;  prev = current;  current = next;  }  return prev;  } |
|  |
| 6)Bank transction  #include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  struct Node  {  int transID,amt,d,m,y,h,min,s;  char name[30],user[30],benID[30];  struct Node \*next;  };  struct Node \*getNode()  {  return malloc(sizeof(struct Node));  }  void freeNode(struct Node \*p)  {  free(p);  }  void readInfo(struct Node \*p)  {  printf("\nTrans id,Name,Amount,UserID,BenID : \n");  scanf("%d%s%d%s%s",&p->transID,p->name,&p->amt,p->user,p->benID);  }  void displayInfo(struct Node \*p)  {  printf("\n%d %d %s %s %s ",p->transID,p->amt,p->name,p->user,p->benID);  }  struct Node \*insertRear(struct Node \*first)  {  struct Node \*nw,\*last;  nw = getNode();  readInfo(nw);  nw->next = NULL;  if(!first)  {  return nw;  }  else  {  for(last=first;last->next != NULL; last=last->next);  last->next = nw;  }  return first;  };  void displayList(struct Node \*cur)  {  for(;cur != NULL;cur=cur->next)  {  displayInfo(cur);  }  }  struct Node \*Search(struct Node \*first, int id)  {  struct Node \*prev,\*cur,\*last;  for(prev=NULL,cur=first;cur!=NULL && cur->transID!=id;prev=cur,cur=cur->next);  if(cur==NULL)  {  printf("No transaction with transaction id %d",id);  }  else  {  displayInfo(cur);  }  };  int main() {  int id,n,i;  struct Node \*first=NULL;  printf("Enter number of customers:\n");  scanf("%d",&n);  for(i=0;i<n;i++)  {  first=insertRear(first);  }  displayList(first);  scanf("%d",&id);  Search(first,id);  return 0;  } |
| 7.Player  #include<stdio.h>  #include<stdlib.h>  #include<string.h>  #include<ctype.h>  #include<math.h>  typedef struct player  {  char nm[20];  int points;  struct player \*next;  }NODE;  NODE \*create\_node()  {  NODE \*t=NULL;  t=(NODE\*)malloc(sizeof(NODE));  if(t==NULL)  {  printf("insufficient memory");  return 0;  }  else  {  t->next=t;  return t;  }  }  void read(NODE \*nw)  {  printf("enter the player details name and points\n");  scanf("%s%d",nw->nm,&nw->points);  return;  }  void display(NODE \*nw)  {  printf("player name|player points\n");  printf("%s|%d\n",nw->nm,nw->points);  return;  }  void display\_list(NODE \*head)  {  NODE \*cur=head;  if(head==NULL)  {  printf("list empty\n");  return;  }  else  {  printf("player name|player points\n");  do  {  printf("%s|%d\n",cur->nm,cur->points);  cur=cur->next;  }while(cur!=head);  }  return;  }  NODE \*insert\_rear(NODE \*head,NODE \*nw)  {  NODE \*cur=head;  if(head==NULL)  return nw;  while(cur->next!=head)  cur=cur->next;  cur->next=nw;  nw->next=head;  return head;  }  void second\_highest(NODE \*head)  {  NODE \*cur=head->next;  NODE \*max=head;  do  {  if(cur->points>max->points)  max=cur;  cur=cur->next;  }while(cur!=head);  NODE \*second=head;  cur=head->next;  do  {  if(cur==max)  cur=cur->next;  else if(second==max)  second =second->next;  else  {  if(cur->points>second->points)  second=cur;  cur=cur->next;  }  }while(cur!=head);  printf("player with second highest points:\n");  display(second);  return ;  }  int main()  {  NODE \*head=NULL;  NODE \*nw;  int ch,K,N;  char name[20];  do  {  printf("enter 1->insert player\n 2->second highest player\n 3->display players\n 7->exit\n");  scanf("%d",&ch);  switch(ch)  {  case 1: nw=create\_node();  read(nw);  head=insert\_rear(head,nw);  break;  case 2: second\_highest(head);  break;  case 3: display\_list(head);  break;  case 7:break;  }  }while(ch!=7);  } |
|  |
| 8.struct stack  #include<stdio.h>  #include<stdlib.h>  #define MAX 5  struct stack  {  int data[MAX];  int top;  };  typedef struct stack stack;  int full(stack s)  {  if(s.top==MAX-1)  return(1);  else  return(0);  }  int empty(stack s)  {  if(s.top==-1)  return(1);  else  return(0);  }  void PUSH(stack \*ps,int num)  {  if(full(\*ps))  {  printf("\nStack Overflow!!");  return;  }  ps->top++;  ps->data[ps->top]=num;  return;  }  int POP(stck \*ps)  {  int temp;  if(empty(\*ps))  {  printf("\nStack Underflow");  return;  }  temp=ps->data[ps->top];  ps->top--;  return(temp);  }  void DISPLAY(stack s)  {  int i;  if(empty(s))  {  printf("\nStack is empty");  return;  }  for(i=0; i<=s.top; i++)  {  printf("%5d",s.data[i]);  }  }  int main()  {  stack s;  int ch,done=1,num,x;  s.top=-1;//Initially stack top  points to -1  while(done)  {  printf("\n1:PUSH  \n2:POP \n3:DISPLAY \nEXIT");  printf("\nEnter any  option[1-4]");  scanf("%d",&ch);  switch(ch)  {  case 1: printf("\nEnter  the element to be pushed");  scanf("%d",&num)  PUSH(&s,num);  break;  case 2:x=POP(&s);  if(x!=-  1) {  printf("\n%d is  poped",x); }  break;  case 3: DISPLAY(s);  break;  case 4:  default :done=0;  }  }  return 0; |
|  |
| 9.stack  #include <stdio.h>  #include <stdlib.h>  #define STACKSIZE 5  typedef struct  {  char phoneNumber[15];  char message[256];  }Message;  typedef struct  {  Message s[STACKSIZE];  int top;  }Stack;  void readInfo(Message \*p)  {  printf("\nEnter phone number : ");  fflush(stdin);  gets(p->phoneNumber);  printf("\nEnter message : ");  gets(p->message);  }  void displayInfo(Message data)  {  printf("\nPhone Number %s Message : %s",data.phoneNumber,data.message);  }  int full(Stack stk)  {  if(stk.top == STACKSIZE-1)  return 1;  else  return 0;  }  int empty(Stack stk)  {  if(stk.top == -1)  return 1;  else  return 0;  }  void push(Stack \*p,Message data)  {  p->top=p->top + 1;  p->s[p->top] = data;  }  Message pop(Stack \*p)  {  Message data = p->s[p->top];  p->top = p->top - 1;  return data;  }  void displayStack(Stack stack)  {  int i;  printf("\nStack elements : ");  for(i=stack.top;i>=0;i--)  displayInfo(stack.s[i]);  }  Message stackTop(Stack stack)  {  return stack.s[stack.top];  }  int main()  {  Stack stack;  int ch;  Message data;  stack.top = -1;  do  {  printf("\n0->Exit 1->Push 2->Pop 3->Stack top 4->Display : ");  scanf("%d",&ch);  switch(ch)  {  case 0: break;  case 1: if(full(stack))  printf("\nStack overflow");  else  {  readInfo(&data);  push(&stack,data);  }  break;  case 2: if(empty(stack))  printf("\nStack underflow");  else  {  data = pop(&stack);  printf("\nPopped message : ");  displayInfo(data);  }  break;  case 3: if(empty(stack))  printf("\nStack empty");  else  {  data = stackTop(stack);  printf("\nStack top element : ");  displayInfo(data);  }  break;  case 4: if(empty(stack))  printf("\nStack empty");  else  {  displayStack(stack);  }  break;  }  }while(ch!=0);  return 0;  }  \_\_ |
|  |
| 10.INFIX TO POSTFIX  #include <stdio.h>  #include <ctype.h>  struct stack  {  char s[100];  int top;  };  void push(struct stack \*ps, char data)  {  (ps->top)++;  ps->s[ps->top] = data;  }  char pop(struct stack \*ps)  {  char data;  data = ps->s[ps->top];  (ps->top)--;  return data;  }  char stackTop(struct stack stk)  {  return stk.s[stk.top];  }  int empty(struct stack stk)  {  return (stk.top == -1);  }  int isoperand(char ch)  {  return (isalpha(ch) || isdigit(ch));  }  int AssocRL(char op)  {  if (op == '^')  {  return 1;  }  else  {  return 0;  }  }  int AssocLR(char op)  {  switch (op)  {  case '+':  case '-':  case '/':  case '\*':  return 1;  default:  return 0;  }  }  int preced(char op)  {  if (op == '+' || op == '-')  {  return 1;  }  else if (op == '\*' || op == '/')  {  return 2;  }  else  {  return 3;  }  }  void convert(char inf[], char postf[])  {  int i, p;  char inp;  struct stack stk;  stk.top = -1;  for (i = p = 0; (inp = inf[i]) != '\0'; i++)  {  if (isoperand(inp))  {  postf[p++] = inp;  }  else if (inp == ')')  {  while (stackTop(stk) != '(')  {  postf[p++] = pop(&stk);  }  pop(&stk);  }  else  {  do  {  if (empty(stk) || inp == '(' || stackTop(stk) == '(')  break;  else if (AssocRL(inp) && preced(inp) >= preced(stackTop(stk)))  break;  else if (AssocLR(inp) && preced(inp) > preced(stackTop(stk)))  break;  else  postf[p++] = pop(&stk);  } while (1);  push(&stk, inp);  }  }  while (!empty(stk))  {  postf[p++] = pop(&stk);  }  postf[p] = '\0';  }  int main(void)  {  char infix[100], postfix[100];  fgets(infix, sizeof(infix), stdin);  convert(infix, postfix);  printf("%s\n", postfix);  return 0;  } |
| 12.PLAYER DETAILS |
| #include<stdio.h>  #include<stdlib.h>  #include<string.h>  struct Node  {  char player[30];  int number;  struct Node \*previous,\*next;  };  struct Node \*getNode()  {  return malloc(sizeof(struct Node));  }  void readInfo(struct Node \*p)  {  printf("\nEnter PLAYER DETAILS, Name and Jersey number:\n");  scanf("%s%d",p->player,&p->number);  }  void displayInfo(struct Node \*p)  {  printf("\nPLAYER NAME: %s JERSEY NO: %d", p->player, p->number);  }  void freeNode(struct Node \*p)  {  free(p);  }  struct Node \*insertFront(struct Node \*first)  {  struct Node \*nw;  nw= getNode();  readInfo(nw);  nw->previous = NULL;  nw->next = first;  if (first != NULL)  {  first->previous = nw;  }  first = nw;  return first;  }  void DisplayList(struct Node \*first)  {  struct Node \*current;  current=first;  printf("--------------------------------");  for(current=first;current!=NULL;current=current->next)  {  displayInfo(current);  }  printf("\n------------------------------");  }  void insertAfter(struct Node \*first, int playerNumber)  {  struct Node \*cur = first;  if (cur == NULL) {  printf("\nList is empty.\n");  return;  }  while (cur != NULL && cur->number != playerNumber) {  cur = cur->next;  }  if (cur == NULL) {  printf("\nNode not found.\n");  return;  }  struct Node \*newNode = getNode();  readInfo(newNode);  newNode->previous = cur;  newNode->next = cur->next;  if (cur->next != NULL) {  cur->next->previous = newNode;  }  cur->next = newNode;  printf("\nNode inserted after the specified player number.\n");  }  int main()  {  struct Node \*first = NULL;  int choice,num;  char sname[20];  while (1) {  printf("\n1->InsertFront 2->InsertAfter 3->DISPLAY");  scanf("%d", &choice);  switch (choice) {  case 1:  first = insertFront(first);  break;  case 2:printf("\nEnter the player number after which you want to insert: ");  int playerNumber;  scanf("%d", &playerNumber);  insertAfter(first, playerNumber);  break;  case 3: printf("\nPlayer Information:\n");  DisplayList(first);  break;  case 10: exit(0);    }  }  return 0;  } |
| 13.LINEAR QUEUE |
| #include <stdio.h>  #include<stdlib.h>  #define SIZE 5  struct QueueRecord  {  int front;  int rear;  int array[SIZE];  };  typedef struct QueueRecord  Queue;  int IsFULL(Queue);  int IsEmpty(Queue);  void Enqueue(Queue \*,int);  int Dequeue(Queue \*);  void display(Queue );  Queue CreateQueue();  Queue CreateQueue()  {  Queue Q;  Q.rear = - 1;  Q.front = -1;  return Q;  }  int IsFULL(Queue Q)  {  if(Q.rear == SIZE - 1)  return 1;  else  return 0;  }  int IsEmpty(Queue Q)  {  if (Q.rear == -1 || Q.front >=  Q.rear)  return 1;  else  return 0;  }  void Enqueue(Queue \*Q,int  element)  {  if (IsFULL(\*Q))  {  printf("Queue is full\n");  }  else  {  Q->rear = Q->rear + 1;  Q->array[Q->rear] = element;  }  }  int Dequeue(Queue \*Q)  {  if (IsEmpty(\*Q))  {  printf("Queue is Empty \n");  return -1;  }  else  {  Q->front++;  return Q->array[Q->front];  }  }  void display(Queue Q)  {  int i;  if (IsEmpty(Q))  printf("Queue is empty \n");  else  {  printf("Queue is : \n");  for (i = Q.front+1; i <= Q.rear;  i++)  printf("%d ", Q.array[i]);  printf("\n");  }  }  int main()  {  int i,ch,n,item,x;  Queue Q;  Q=CreateQueue();  while(1)  {  printf("1-Insert\n2-Remove  Element\n3-Display\n4-Exit\n");  printf("Enter Choice\n");  scanf("%d",&ch);  switch(ch)  {  case 1: printf("Enter the  element to be inserted\n");  scanf("%d",&item);  Enqueue(&Q,item);  break;  case 2:x=Dequeue(&Q);  if(x!=-1)  {  printf("Removed  Element is = %d\n",x);  }  break;  case 3: display(Q);  break;  case 4: exit(0);  break;  }  }  return 0;  } |
|  |
| 14.CIRCULAR QUEUE |
| include <stdio.h>  #include<stdlib.h>  #define SIZE 5  struct QueueRecord  {  int front;  int rear;  int array[SIZE];  };  typedef struct QueueRecord  Queue;  int IsFULL(Queue);  int IsEmpty(Queue);  void Enqueue(Queue \*,int);  int Dequeue(Queue \*);  void display(Queue );  Queue CreateQueue();  Queue CreateQueue()  {  Queue Q;  Q.rear = - 1;  Q.front = -1;  return Q;  }  int IsFULL(Queue Q)  {  if(((Q.rear + 1)%SIZE==Q.front) )  return 1;  else  return 0;  }  int IsEmpty(Queue Q)  {  if(Q.rear == -1)  return 1;  else  return 0;  }  void Enqueue(Queue \*Q,int  element)  {  if(IsFULL(\*Q))  printf("\n Queue is full!! \n");  else  {  if(Q->front == -1)  Q->front = 0;  Q->rear = (Q->rear + 1) %  SIZE;  Q->array[Q->rear] = element;  printf("\n Inserted -> %d",  element);  }  }  int Dequeue(Queue \*Q)  {  int element;  if(IsEmpty(\*Q)) {  printf("\n Queue is empty !!  \n");  return(-1);  } else {  element = Q->array[Q->front];  if (Q->front ==Q-> rear)  {  Q->front = -1;  Q->rear = -1;  } else {  Q->front = (Q->front + 1) %  SIZE;  }  printf("\n Deleted element ->  %d \n", element);  return(element);  }  }  void display(Queue Q)  {  int i;  if(IsEmpty(Q))  printf(" \n Empty Queue\n");  else  {  //printf("\n Front -> %d  ",Q.front);  printf("\n Items -> ");  for( i = Q.front; i!=Q.rear;  i=(i+1)%SIZE) {  printf("%d ",Q.array[i]);  }  printf("%d ",Q.array[i]);  //printf("\n Rear -> %d  \n",Q.rear);  }  }  int main()  {  int i,ch,n,item,x;  Queue Q;  Q=CreateQueue();  while(1)  {  printf("\n1-Insert\n2-Remove  Element\n3-Display\n4-Exit\n");  printf("Enter Choice\n");  scanf("%d",&ch);  switch(ch)  {  case 1: printf("Enter the  element to be inserted\n");  scanf("%d",&item);  Enqueue(&Q,item);  break;  case 2:x=Dequeue(&Q);  if(x!=-1)  {  printf("Removed  Element is = %d\n",x);  }  break;  case 3: display(Q);  break;  case 4: exit(0);  break;  }  }  return 0;  } |
| 15.palindrome |
| #include <stdio.h>  #include <stdlib.h>  struct node  {  int num;  struct node \*next;  };  int create(struct node \*\*);  int palin\_check (struct node \*, int);  void release(struct node \*\*);  int main()  {  struct node \*p = NULL;  int result, count;  printf("Enter data into the list\n");  count = create(&p);  result = palin\_check(p, count);  if (result == 1)  {  printf("The linked list is a palindrome.\n");  }  else  {  printf("The linked list is not a palindrome.\n");  }  release (&p);  return 0;  }  int palin\_check (struct node \*p, int count)  {  int i = 0, j;  struct node \*front, \*rear;  while (i != count / 2)  {  front = rear = p;  for (j = 0; j < i; j++)  {  front = front->next;  }  for (j = 0; j < count - (i + 1); j++)  {  rear = rear->next;  }  if (front->num != rear->num)  {  return 0;  }  else  {  i++;  }  }  return 1;  }  int create (struct node \*\*head)  {  int c, ch, count = 0;  struct node \*temp;  do  {  printf("Enter number: ");  scanf("%d", &c);  count++;  temp = (struct node \*)malloc(sizeof(struct node));  temp->num = c;  temp->next = \*head;  \*head = temp;  printf("Do you wish to continue [1/0]: ");  scanf("%d", &ch);  }while (ch != 0);  printf("\n");  return count;  }  void release (struct node \*\*head)  {  struct node \*temp = \*head;  while ((\*head) != NULL)  {  (\*head) = (\*head)->next;  free(temp);  temp = \*head;  }  } |
|  |
| 16.queues basic |
| #include<stdio.h>  # define SIZE 5  typedef struct  {  int s[SIZE];  int front,rear;  }QUEUE;  void enqueue(QUEUE \*q,int data)  {  q->s[++q->rear]=data;  }  int dequeue(QUEUE \*q)  {  return(q->s[q->front++]);  }  int empty(QUEUE q)  {  return(q.front>q.rear);  }  int full(QUEUE q)  {  return(q.rear==SIZE-1);  }  void display(QUEUE q)  {  if(empty(q))  {  printf("queue empty\n");  return;  }  int i=q.front;  while(i<=q.rear)  {  printf("%d\n",q.s[i]);  i++;  }  }  int main()  {  QUEUE q;  q.front=0;  q.rear=-1;  int ch,data;  do  {  printf("enter 1]enqueue\n 2]dequeue\n 3]display\n");  scanf("%d",&ch);  switch(ch)  {  case 1: if(full(q))  {  printf("queue overflow\n");  break;  }  printf("enter data to enqueue\n");  scanf("%d",&data);  enqueue(&q,data);  break;  case 2: if(empty(q))  {  printf("queue underflow\n");  break;  }  data=dequeue(&q);  printf("dequeued element:%d\n",data);  break;  case 3:display(q);  }  }while(ch!=4);  } |
| 18.LOTTERY QUEUES |
| #include<stdio.h>  #define SIZE 10  typedef struct {  int num;  char nm[20];  } data;  typedef struct {  data s[SIZE];  int front, rear;  } QUEUE;  data read() {  data dt;  printf("Enter the name of the customer and the coupon number: ");  scanf("%s %d", dt.nm, &dt.num);  return dt;  }  void display(data dt) {  printf("Customer Name: %s\nCoupon Number: %d\n\n", dt.nm, dt.num);  }  void enqueue(QUEUE \*q, data dt) {  q->s[++q->rear] = dt;  }  data dequeue(QUEUE \*q) {  return q->s[q->front++];  }  int empty(QUEUE q) {  return (q.front > q.rear);  }  int full(QUEUE q) {  return (q.rear == SIZE - 1);  }  void displayAllCustomers(QUEUE q, int N) {  printf("Details of Customers:\n");  for (int i = 0; i < N; i++) {  display(q.s[i]);  }  }  void displayCouponWinners(QUEUE q, int N, int divisor) {  printf("Coupon Winners (Divisible by %d):\n", divisor);  int totalWinners = 0;  for (int i = 0; i < N; i++) {  if (q.s[i].num % divisor == 0) {  display(q.s[i]);  totalWinners++;  }  }  printf("Total Coupon Winners (Divisible by %d): %d\n", divisor, totalWinners);  }  int main() {  QUEUE q;  q.front = 0;  q.rear = -1;  int N;  printf("Enter the number of customers: ");  scanf("%d", &N);  for (int i = 0; i < N; i++) {  data dt = read();  enqueue(&q, dt);  }  displayAllCustomers(q, N);  int divisor = 7;  displayCouponWinners(q, N, divisor);  printf("Total Coupons Issued: %d\n", N);  return 0;  } |
|  |
| 19.POSTFIX TO INFIX |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAX\_SIZE 100  struct Stack {  int top;  char items[MAX\_SIZE];  };  void push(struct Stack \*stack, char item) {  if (stack->top == MAX\_SIZE - 1) {  printf("Stack Overflow\n");  exit(1);  }  stack->items[++stack->top] = item;  }  char pop(struct Stack \*stack) {  if (stack->top == -1) {  printf("Stack Underflow\n");  exit(1);  }  return stack->items[stack->top--];  }  int isOperator(char c) {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  void postfixToInfix(char postfix[], char infix[]) {  struct Stack stack;  stack.top = -1;  int length = strlen(postfix);  for (int i = 0; i < length; i++) {  char c = postfix[i];  if (isalnum(c)) {  push(&stack, c);  } else if (isOperator(c)) {  char operand2 = pop(&stack);  char operand1 = pop(&stack);  sprintf(infix + strlen(infix), "(%c%c%c)", operand1, c, operand2);  push(&stack, infix[strlen(infix) - 1]);  }  }  }  int main() {  char postfix[MAX\_SIZE];  printf("Enter postfix expression: ");  scanf("%s", postfix);  char infix[MAX\_SIZE];  postfixToInfix(postfix, infix);  printf("Infix expression: %s\n", infix);  return 0;  } |
|  |
| 20.BROWSER HISTORY |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_PAGES 100  struct BrowserHistory {  char pages[MAX\_PAGES][100];  int top;  };  struct BrowserHistory\* createBrowserHistory() {  struct BrowserHistory\* history = (struct BrowserHistory\*)malloc(sizeof(struct BrowserHistory));  history->top = -1;  return history;  }  void pushPage(struct BrowserHistory\* history, char\* page) {  if (history->top == MAX\_PAGES - 1) {  printf("History is full. Cannot push page.\n");  return;  }  history->top++;  strcpy(history->pages[history->top], page);  }  char\* getCurrentPage(struct BrowserHistory\* history) {  if (history->top == -1) {  return "No page currently open.";  }  return history->pages[history->top];  }  bool canGoBack(struct BrowserHistory\* history) {  return history->top > 0;  }  bool canGoForward(struct BrowserHistory\* history) {  return history->top < MAX\_PAGES - 1 && history->top > -1;  }  char\* goBack(struct BrowserHistory\* history) {  if (!canGoBack(history)) {  return "Cannot go back. No previous pages.";  }  history->top--;  return history->pages[history->top];  }  char\* goForward(struct BrowserHistory\* history) {  if (!canGoForward(history)) {  return "Cannot go forward. No next pages.";  }  history->top++;  return history->pages[history->top];  }  int main() {  struct BrowserHistory\* history = createBrowserHistory();  pushPage(history, "Google");  pushPage(history, "GitHub");  pushPage(history, "Stack Overflow");  printf("Current Page: %s\n", getCurrentPage(history));  printf("Going back...\n");  printf("Previous Page: %s\n", goBack(history));  printf("Going forward...\n");  printf("Next Page: %s\n", goForward(history));  free(history);  return 0;  } |
|  |
| * 22 **Traversal of Circular Linked List**   Code  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  struct Node\* createNode(int data);  void insertAtEnd(struct Node\*\* head, int newData);  void traverseCircularLinkedList(struct Node\* head);  int main() {  struct Node\* head = NULL;  insertAtEnd(&head, 3);  insertAtEnd(&head, 7);  insertAtEnd(&head, 9);  printf("Circular linked list: ");  traverseCircularLinkedList(head);  return 0;  }  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed!\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  void insertAtEnd(struct Node\*\* head, int newData) {  struct Node\* newNode = createNode(newData);  if (\*head == NULL) {  \*head = newNode;  newNode->next = newNode;  } else {  struct Node\* temp = \*head;  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode;  newNode->next = \*head;  }  void traverseCircularLinkedList(struct Node\* head) {  if (head == NULL) {  printf("Circular linked list is empty\n");  return;  }  struct Node\* temp = head;  do {  printf("%d ", temp->data);  temp = temp->next;  } while (temp != head);  } |
| 23.HOTEL |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_PLATES 10  struct Stack {  int top;  int plates[MAX\_PLATES];  };  struct Stack\* createStack() {  struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));  stack->top = -1;  return stack;  }  void push(struct Stack\* stack, int plateNumber) {  if (stack->top == MAX\_PLATES - 1) {  printf("Stack Overflow: Cannot add more plates.\n");  return;  }  stack->plates[++stack->top] = plateNumber;  printf("Plate %d added.\n", plateNumber);  }  int pop(struct Stack\* stack) {  if (stack->top == -1) {  printf("Stack Underflow: No plates to remove.\n");  return -1;  }  int plateNumber = stack->plates[stack->top--];  printf("Plate %d removed.\n", plateNumber);  return plateNumber;  }  void displayStack(struct Stack\* stack) {  if (stack->top == -1) {  printf("Stack is empty.\n");  return;  }  printf("Current stack of plates:\n");  for (int i = stack->top; i >= 0; i--) {  printf("| %d |\n", stack->plates[i]);  }  }  int main() {  struct Stack\* plateStack = createStack();  push(plateStack, 1);  push(plateStack, 2);  push(plateStack, 3);  displayStack(plateStack);  pop(plateStack);  displayStack(plateStack);  pop(plateStack);  pop(plateStack);  pop(plateStack);  displayStack(plateStack);  free(plateStack);  return 0;  } |
| 24. SPLITTING OF 2 QUEUES |
| #include<stdio.h>  #define SIZE 20  typedef struct {  int s[SIZE];  int front, rear;  } QUEUE;  void enqueue(QUEUE \*q, int data) {  q->s[++q->rear] = data;  }  int dequeue(QUEUE \*q) {  return (q->s[q->front++]);  }  int empty(QUEUE q) {  return (q.front > q.rear);  }  int full(QUEUE q) {  return (q.rear == SIZE - 1);  }  void display(QUEUE q) {  if (empty(q)) {  printf("Queue empty\n");  return;  }  int i = q.front;  while (i <= q.rear) {  printf("%d\n", q.s[i]);  i++;  }  }  void splitQueue(QUEUE \*original, QUEUE \*q1, QUEUE \*q2, int size1, int size2) {  int i;  for (i = 0; i < size1; i++) {  if (!empty(\*original)) {  enqueue(q1, dequeue(original));  }  }  for (i = 0; i < size2; i++) {  if (!empty(\*original)) {  enqueue(q2, dequeue(original));  }  }  }  void displaySplitterQueues(QUEUE q1, QUEUE q2) {  printf("Splitter Queue 1:\n");  display(q1);  printf("Splitter Queue 2:\n");  display(q2);  }  int main() {  QUEUE q, q1, q2;  q.front = 0;  q.rear = -1;  q1.front = 0;  q1.rear = -1;  q2.front = 0;  q2.rear = -1;  int ch, data, size1, size2;  do {  printf("Enter 1] Enqueue\n 2] Dequeue\n 3] Display\n 4] Split Queue\n 5] Exit\n");  scanf("%d", &ch);  switch (ch) {  case 1:  if (full(q)) {  printf("Queue overflow\n");  break;  }  printf("Enter data to enqueue: ");  scanf("%d", &data);  enqueue(&q, data);  break;  case 2:  if (empty(q)) {  printf("Queue underflow\n");  break;  }  data = dequeue(&q);  printf("Dequeued element: %d\n", data);  break;  case 3:  printf("Queue elements:\n");  display(q);  break;  case 4:  printf("Enter size for Queue 1: ");  scanf("%d", &size1);  printf("Enter size for Queue 2: ");  scanf("%d", &size2);  splitQueue(&q, &q1, &q2, size1, size2);  printf("Queues split!\n");  displaySplitterQueues(q1, q2);  break;  }  } while (ch != 5);  return 0;  } |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |