Data Structure Module 1

Data Structure Classification

Data may be organized in many different ways. The logical or mathematical model of a particular organization of data is called a data structure.

Data structures are generally classified into two main categories:

1. Primitive Data Structures: Primitive data structures are the fundamental data types supported by a programming language. These include basic data types such as integer, float, character, and boolean. They are also known as simple data types as they consist of indivisible values.
2. Non-Primitive Data Structures: Non-primitive data structures are created using primitive data structures. Examples include complex numbers, linked lists, stacks, trees, and graphs.

   Non-Primitive Data Structures are further classified into:

   A. Linear Data Structures: A data structure is linear if its elements form a sequence or a linear list. Linear data structures can be represented in memory in two ways:

   1. Using sequential memory locations (arrays)
   2. Using pointers or links (linked lists)

   Common examples of linear data structures are arrays, queues, stacks, and linked lists.

   B. Non-Linear Data Structures: A data structure is non-linear if the data is not arranged in a sequence. Insertion and deletion are not possible in a linear fashion. These structures represent hierarchical relationships between elements.

   Examples of non-linear data structures are trees and graphs.

Data Structure Operations

1. Traversing: Accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)
2. Searching: Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.
3. Inserting: Adding a new node/record to the structure.
4. Deleting: Removing a node/record from the structure.

The following two operations are used in special situations:

1. Sorting: Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as Aadhaar number or bank account number)
2. Merging: Combining the records in two different sorted files into a single sorted file.

Array

An Array is defined as an ordered set of similar data items stored in consecutive memory locations.

- All data items in an array are of the same type.
- Each data item can be accessed using the array name and a unique index value.
- An array is a set of pairs <index, value>, where each index has an associated value.
  <index, value> pairs could be:
  < 0 , 25 > list[0] = 25
  < 1 , 15 > list[1] = 15

Array Declaration in C:

A one-dimensional array in C is declared by adding brackets [] to the variable name.
Example: int list[5], plist[5];

- list[5] defines 5 integers, with indexes from 0 to 4 (list[0], list[1], ..., list[4]).
- plist[5] defines an array of 5 pointers to integers.

Array Implementation:

- When an array list[5] is declared, the compiler allocates 5 consecutive memory locations, each large enough to hold an integer.
- The address of the first element (list[0]) is called the Base Address.
- To compute the address of list[i], the compiler uses:
  address(list[i]) = Base Address + i * sizeof(int)

Difference between `int *list1;` and `int list2[5];` in C is:

`int *list1;`

- `list1` is a pointer variable that can store the address of an integer.
- It does not allocate any memory for storing an integer value.
- It just holds the address of an integer value stored somewhere else in memory.

`int list2[5];`

- `list2` is an array of 5 integers.
- It allocates contiguous memory locations to store 5 integer values.
- The name `list2` can be used to access the base address (address of `list2[0]`) of the array.
- `int *list1;` declares a pointer that can point to an integer value.
- `int list2[5];` declares an array that can store 5 integer values.
- The main difference is that `list1` holds the address of an integer, while `list2` is an array that has memory allocated to store 5 integer values directly.

Array as Parameters to Function

- In C, when an array is passed as a parameter to a function, the array name decays into a pointer to the first element of the array.
- Arrays are passed by reference (as pointers) to functions in C.
- The array size must be passed as a separate argument, or accessed as a global variable.
- Inside the function, the array is accessed using the pointer notation `arr[i]`.
- The function can modify the array elements, but it cannot resize or reallocate the array.

```c
#include <stdio.h>
#define MAX_SIZE 5

float calculateSum(float arr[], int size) {
    float sum = 0.0;
    for (int i = 0; i < size; i++) {
        sum += arr[i]; // Accessing array elements
    }
    return sum;
}

int main() {
    float numbers[MAX_SIZE] = {1.2, 3.4, 5.6, 7.8, 9.0};
    float result = calculateSum(numbers, MAX_SIZE); // Passing array and size
    printf("The sum of the array elements is: %.2f\n", result);
    return 0;
}
```

- In the `main` function, we declare an array `numbers` of size `MAX_SIZE` (which is defined as 5) and initialize it with some float values.
- Inside the `calculateSum` function:
  - The parameter `arr` is now a pointer to the first element of the `numbers` array passed from `main`.
  - The parameter `size` holds the value `MAX_SIZE` (5), which is the size of the array.
  - The function uses a `for` loop to iterate over the elements of the array, accessing them through the pointer `arr` and the array subscript notation `arr[i]`. Since `arr` holds the address of the first element, `arr[i]` accesses the subsequent elements in the array.

Structure

Initialization and Declaration

In C programming language, a structure is a way to group diverse data elements of different data types under a single name. It allows for the organization and storage of related data items as a single unit. The members of a structure can be of different data types such as integers, floats, characters, or even other structures.

struct structure_name
{
data_type member1;
data_type member2;
...
data_type memberN;
};
typedef struct struct_name Type_name;

- `struct` is the keyword used to define a structure.
- `structure_name` is the name given to the structure.
- `member1`, `member2`, ..., `memberN` are the names of the individual data elements or members within the structure.
- `data_type` specifies the data type of each member, such as `int`, `float`, `char`, etc.
- `typedef` is the keyword used to define a new type name or alias for the structure.
- `Type_name` is the user-defined data type name or alias for the structure.

Example:

```
struct Employee {
    char name[50];
    int age;
    float salary;
};
```

In the above example, a structure named `Employee` is defined with three members:

- `name`: a character array of size 50 to store the employee's name.
- `age`: an integer to store the employee's age.
- `salary`: a float to store the employee's salary.

```
struct Employee emp1, emp2;

strcpy(emp1.name, "Rahul Sharma");
emp1.age = 35;
emp1.salary = 75000.0;
```

A self-referential structure, also known as a recursive structure, is a data structure in which each instance of the structure contains a pointer or reference to another instance of the same type. In other words, the structure contains a member that refers to objects of the same structure type.

Structure and Union difference

Feature	Structures	Unions
Declaration	`struct` keyword followed by a structure tag	`union` keyword followed by a union tag
Memory Usage	Each member has its own memory space	Members share the same memory space
Size Calculation	Size is the sum of sizes of all members	Size is the size of the largest member
Access	Members accessed individually	Only one member can be accessed at a time
Usage	Used when multiple pieces of data are needed	Used when data needs to be overlapped or shared between members
Memory Overlap	Members do not overlap	Members may overlap if they're of different sizes

Write syantax and examples

Pointer

A pointer is a variable that stores the memory address of another variable. Pointers are used to access and manipulate data indirectly through memory addresses.

Declaration: `data_type *pointer_name;`

Example: `int i, *pi; // i is an integer variable, pi is a pointer to an integer`

Operators:

1. & (Address Operator): The unary operator & is used to obtain the memory address of a variable.
   `pi = &i; // pi stores the address of i`
2. - (Indirection/Dereference Operator): The *operator is used to access the value stored at the memory address pointed to by the pointer.*
   `pi = 10; // Assigns 10 to the integer variable i`

Null Pointer:

- A null pointer is a pointer that does not point to any valid memory location.
- In C, the value NULL (defined as 0) represents a null pointer.
- Dereferencing a null pointer can lead to a segmentation fault or undefined behavior.
  `int *ptr = NULL; // ptr is a null pointer`

Dangers of Pointers:

1. Accessing out-of-range memory: Attempting to access memory locations that are not allocated to the program or not within the valid range can lead to undefined behavior or program crashes.
   `int arr[5] = {1, 2, 3, 4, 5}; int *p = arr; printf("%d", *(p + 6)); // Accessing out-of-range memory`
2. Dereferencing a null pointer: Dereferencing a null pointer (attempting to access the value at address 0) can cause a segmentation fault or undefined behavior.
   `int *ptr = NULL; *ptr = 10; // Dereferencing a null pointer (Segmentation Fault)`
3. Type casting between pointer types: Improper type casting between pointer types can lead to unexpected results or memory corruption.
   `int *pi = malloc(sizeof(int)); float *pf = (float*)pi; // Type casting between pointer types`
4. Incorrect return type for functions: If a function's return type is omitted, it defaults to int, which can be interpreted as a pointer type, leading to unexpected behavior.

Pointers are dangerous

Pointer can be very dangerous if they are misused. The pointers are dangerous in the following situations:

1. Pointer can be dangerous when an attempt is made to access an area of memory that is either out of range of the program or that does not contain a pointer reference to a legitimate object.

```c
int main() {
    int *p;
    int pa = 10;
```

```
    p = &pa;
    printf("%d", *p); // Output = 10
    printf("%d", *(p+1)); // Accessing memory which is out of range
}
```

2. It is dangerous when a NULL pointer is de-referenced, because on some computers it may return 0 and permitting execution to continue, or it may return the result stored in location zero, so it may produce a serious error.
3. Pointer is dangerous when using explicit type casts in converting between pointer types.

```
int *pi = malloc(sizeof(int));
float *pf = (float*)pi; // Type casting between pointer types
```

4. In some systems, pointers have the same size as the `int` type. Since `int` is the default type specifier, some programmers omit the return type when defining a function. The return type defaults to `int`, which can later be interpreted as a pointer. This has proven to be a dangerous practice on some computers, and the programmer should define explicit types for functions.

Dynamic Allocation

Dynamic memory allocation refers to the process of manually managing memory allocation and deallocation during the runtime of a program using standard library functions. This allows for more flexibility and efficient memory usage compared to static memory allocation. Here's an explanation of each function used for dynamic memory allocation in C:

1. `malloc()` (Memory Allocation):
   - This function is used to allocate a block of contiguous memory of a specified size.
   - The syntax is: `ptr = (data_type *) malloc(size_in_bytes);`
   - `malloc()` returns a pointer to the first byte of the allocated memory block.
   - If the allocation is successful, the returned pointer can be used to access and manipulate the allocated memory.
   - If the allocation fails (e.g., due to insufficient memory), `malloc()` returns a null pointer.
   - Example: `int *ptr = (int *) malloc(10 * sizeof(int));` allocates memory for an array of 10 integers.
2. `calloc()` (Contiguous Allocation):
   - This function is used to allocate a block of contiguous memory for an array of elements and initializes all bytes to zero.
   - The syntax is: `ptr = (data_type *) calloc(num_elements, element_size);`
   - `calloc()` returns a pointer to the first element of the allocated memory block.
   - If the allocation is successful, the returned pointer can be used to access and manipulate the allocated memory.
   - If the allocation fails (e.g., due to insufficient memory), `calloc()` returns a null pointer.
   - Example: `float *ptr = (float *) calloc(25, sizeof(float));` allocates memory for an array of 25 floats, all initialized to zero.
3. `realloc()` (Re-allocation):
   - This function is used to resize (increase or decrease) the memory block previously allocated with `malloc()` or `calloc()`.
   - The syntax is: `ptr = realloc(ptr, new_size);`

- `realloc()` returns a pointer to the reallocated memory block, which may be different from the original pointer if the memory needs to be moved.
- If the reallocation is successful, the returned pointer can be used to access and manipulate the reallocated memory.
- If the reallocation fails (e.g., due to insufficient memory), `realloc()` returns a null pointer, and the original memory block remains unchanged.
- Example: `ptr = realloc(ptr, 20 * sizeof(int));` resizes the memory block pointed to by `ptr` to accommodate an array of 20 integers.

4. `free()` (Memory Deallocation):
   - This function is used to deallocate (release) the memory block previously allocated with `malloc()`, `calloc()`, or `realloc()`.
   - The syntax is: `free(ptr);`
   - After calling `free()`, the memory pointed to by `ptr` is released and can be reused by the system.
   - Attempting to access the memory after calling `free()` can lead to undefined behavior or a segmentation fault.
   - It is important to call `free()` on dynamically allocated memory when it is no longer needed to avoid memory leaks.
   - Example: `free(ptr);` releases the memory block pointed to by `ptr`.

Dynamic memory allocation is essential when the size of the required memory is not known at compile-time or when the memory requirements change during program execution. It provides flexibility and efficient memory usage but also introduces the risk of memory leaks and other memory-related errors if not used correctly.

String operations

int strlen(char* str)	length of the char array
char* strcat(char* dest, char* src)	concatenate dest and src strings return result in dest
char* strcat(char* dest, char* src, int n)	concatenate dest and n characters from src strings return result in dest
char* strcpy(char* dest, char* src)	copy src into dest ; return dest
char* strncpy(char* dest, char* src, int n)	copy n characters from src string into dest ; return dest
char* strcmp(char* str1, char* str2)	compares two strings . returns <0 if str1<str2, returns 0 if str1=str2, returns >0 if str1>str2
char* strcmpi(char* str1, char* str2) or char* stricmp(char* str1, char* str2)	compares two strings . returns <0 if str1<str2, returns 0 if str1=str2, returns >0 if str1>str2 **it ignores case.**
char* strcmp(char* str1, char* str2)	compare first n characters of two strings . returns <0 if str1<str2, returns 0 if str1=str2, returns >0 if str1>str2
char* strchr(char*s, int c)	returns pointer to the first occurrence of c in s; return NULL if not present.
char* strrchr(char*s, int c)	returns pointer to the last occurrence of c in s; return NULL if not present.
char* strstr(char* s, char* pat)	Return pointer to start of pat in s
char* strtok(char* s, char delimiters)	Return a token from s, token is surrounded by delimiters
char* strspn(char* s, char* spanset)	Scan s for characters in spanset , return length of span
char* strcspn(char* s, char* spanset)	Scan s for characters not in spanset , return length of span
strlwr()	converts from upper case to lower case
struper()	converts from upper case to lower case
strrev()	reverses a string
strset(char* s, char c)	It sets all characters in string to character c
strnset(char* s, char c)	It sets first n characters in string to character c

Polynomial

A polynomial is a mathematical expression consisting of variables (also known as indeterminates or unknowns) and coefficients, combined using addition, subtraction, multiplication, and non-negative integer exponents. It typically takes the form:

The degree of a polynomial is the highest power of the variable in the polynomial expression.

Sparse Matrix

A sparse matrix is a type of matrix that contains mostly zero elements. In contrast to dense matrices, which have a majority of non-zero elements, sparse matrices have a high proportion of zero entries relative to the total number of elements.

Data Structures - Module 1 - Programs

Array Operations

```c
#include <stdio.h>

// Function to delete an element from the array
void deleteElement(int arr[], int *size, int key) {
    int pos = search(arr, *size, key); // Search for the element
    if (pos != -1) {
        // Shift elements to the left to overwrite the deleted element
        for (int i = pos; i < *size - 1; i++) {
            arr[i] = arr[i + 1];
        }
        (*size)--; // Reduce the size of the array
        printf("Element %d deleted successfully.\n", key);
    } else {
        printf("Element %d not found in the array.\n", key);
    }
}

// Function to insert an element into the array
void insertElement(int arr[], int *size, int key, int index) {
    if (index >= 0 && index <= *size) {
        // Shift elements to the right to make space for the new element
        for (int i = *size; i > index; i--) {
            arr[i] = arr[i - 1];
        }
        arr[index] = key; // Insert the new element at the specified index
        (*size)++; // Increase the size of the array
        printf("Element %d inserted at index %d.\n", key, index);
    } else {
        printf("Invalid index for insertion.\n");
    }
}

// Function to search for an element in the array
int search(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Return the index of the element if found
        }
    }
    return -1; // Return -1 if the element is not found
}


// Function to update an element in the array
void updateElement(int arr[], int size, int key, int newVal) {
    int pos = search(arr, size, key); // Search for the element
```

```c
    if (pos != -1) {
        arr[pos] = newVal; // Update the element with the new value
        printf("Element %d updated to %d.\n", key, newVal);
    } else {
        printf("Element %d not found in the array.\n", key);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    printf("Array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main()
{
    int arr[100] = {1, 2, 3, 4, 5};
    int size = 5;

    printf("Original ");
    printArray(arr, size);

    int searchKey = 3;
    int indexToDelete = 2;
    int indexToInsert = 1;
    int newElement = 6;

    printf("Searching for %d: Found at index %d\n", searchKey, search(arr,
size, searchKey));
    deleteElement(arr, &size, 4);
    insertElement(arr, &size, newElement, indexToInsert);
    updateElement(arr, size, 2, 10);

    printf("Final ");
    printArray(arr, size);

    return 0;
}
```

Array Alogorithms for Searching and Sorting

Linear Search

```c
#include <stdio.h>

void LinearSearch(int a[], int n, int key) {
```

```
    int flag = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] == key) {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
        printf("Element found\n");
    else
        printf("Element not found\n");
}

int main() {
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 10;
    LinearSearch(arr, n, key);
    return 0;
}
```

Binary Search

```
#include <stdio.h>

void BinarySearch(int a[], int n, int key) {
    int low = 0, high = n - 1, mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (a[mid] == key) {
            printf("Element found\n");
            return;
        } else if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    printf("Element not found\n");
}

int main() {
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 10;
    BinarySearch(arr, n, key);
    return 0;
}
```

Bubble Sort

```c
#include <stdio.h>

int main() {
    int data[100], i, n, step, temp;
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        printf("%d. Enter element: ", i + 1);
        scanf("%d", &data[i]);
    }
    for (step = 0; step < n - 1; ++step) {
        for (i = 0; i < n - step - 1; ++i) {
            if (data[i] > data[i + 1]) {
                temp = data[i];
                data[i] = data[i + 1];
                data[i + 1] = temp;
            }
        }
    }
    printf("In ascending order: ");
    for (i = 0; i < n; ++i)
        printf("%d ", data[i]);
    return 0;
}
```

Selection Sort

```c
#include <stdio.h>

int main() {
    int array[100], n, i, j, position, swap;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers: ", n);
    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);
    for (i = 0; i < (n - 1); i++) {
        position = i;
        for (j = i + 1; j < n; j++) {
            if (array[position] > array[j])
                position = j;
        }
        if (position != i) {
            swap = array[i];
            array[i] = array[position];
            array[position] = swap;
        }
    }
    printf("Sorted list in ascending order:\n");
    for (i = 0; i < n; i++)
        printf("%d\n", array[i]);
```

```
    return 0;
}
```

Dynamic Allocated Array

```
#include <stdio.h>
#include <stdlib.h>

// Function to create and display a 1D array
void createAndDisplay1DArray(int n) {
    // Dynamically allocate memory for the 1D array
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }

    // Initialize the array elements
    printf("Enter %d elements for the 1D array:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Display the array elements
    printf("1D Array elements are:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(arr);
}

// Function to create and display a 2D array
void createAndDisplay2DArray(int rows, int cols) {
    // Dynamically allocate memory for the 2D array
    int **arr = (int **)malloc(rows * sizeof(int *));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }
    for (int i = 0; i < rows; i++) {
        arr[i] = (int *)malloc(cols * sizeof(int));
        if (arr[i] == NULL) {
            printf("Memory allocation failed!\n");
            return;
        }
    }
```

```
    // Initialize the array elements
    printf("Enter %d elements for the %dx%d 2D array:\n", rows * cols, rows,
cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &arr[i][j]);
        }
    }

    // Display the array elements
    printf("2D Array elements are:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    // Free the allocated memory
    for (int i = 0; i < rows; i++) {
        free(arr[i]);
    }
    free(arr);
}

int main() {
    int n, rows, cols;

    // Input size of 1D array
    printf("Enter the size of the 1D array: ");
    scanf("%d", &n);
    createAndDisplay1DArray(n);

    // Input size of 2D array
    printf("\nEnter the number of rows and columns for the 2D array: ");
    scanf("%d %d", &rows, &cols);
    createAndDisplay2DArray(rows, cols);

    return 0;
}
```

Structure

Employee details - Find employee with highest salary

```
#include <stdio.h>

#define MAX_EMPLOYEES 10
```

```c
// Structure to store employee details
struct Employee {
    char EName[50];
    int EmpID;
    struct {
        int date;
        int month;
        int year;
    } DOJ;
    struct {
        float Basic;
        float DA;
        float HRA;
    } Salary;
};

int main() {
    // Array to store details of 10 employees
    struct Employee employees[MAX_EMPLOYEES];

    // Read data for 10 employees
    printf("Enter details for 10 employees:\n");
    for (int i = 0; i < MAX_EMPLOYEES; i++) {
        printf("\nEmployee %d:\n", i + 1);
        printf("Name: ");
        scanf("%s", employees[i].EName);
        printf("Employee ID: ");
        scanf("%d", &employees[i].EmpID);
        printf("Date of Joining (DD MM YYYY): ");
        scanf("%d %d %d", &employees[i].DOJ.date, &employees[i].DOJ.month,
&employees[i].DOJ.year);
        printf("Basic Salary: ");
        scanf("%f", &employees[i].Salary.Basic);
        printf("Dearness Allowance (DA): ");
        scanf("%f", &employees[i].Salary.DA);
        printf("House Rent Allowance (HRA): ");
        scanf("%f", &employees[i].Salary.HRA);
    }

    // Display details of all employees
    printf("\nDetails of all employees:\n");
    for (int i = 0; i < MAX_EMPLOYEES; i++) {
        printf("\nEmployee %d:\n", i + 1);
        printf("Name: %s\n", employees[i].EName);
        printf("Employee ID: %d\n", employees[i].EmpID);
        printf("Date of Joining: %02d-%02d-%d\n", employees[i].DOJ.date,
employees[i].DOJ.month, employees[i].DOJ.year);
        printf("Basic Salary: %.2f\n", employees[i].Salary.Basic);
        printf("Dearness Allowance (DA): %.2f\n", employees[i].Salary.DA);
        printf("House Rent Allowance (HRA): %.2f\n", employees[i].Salary.HRA);
    }
```

```
    // Find the employee with the highest salary
    int highest_salary_index = 0;
    float highest_salary = employees[0].Salary.Basic + employees[0].Salary.DA +
employees[0].Salary.HRA;
    for (int i = 1; i < MAX_EMPLOYEES; i++) {
        float total_salary = employees[i].Salary.Basic + employees[i].Salary.DA +
employees[i].Salary.HRA;
        if (total_salary > highest_salary) {
            highest_salary = total_salary;
            highest_salary_index = i;
        }
    }

    // Display details of the employee with the highest salary
    printf("\nEmployee with the highest salary:\n");
    printf("Name: %s\n", employees[highest_salary_index].EName);
    printf("Employee ID: %d\n", employees[highest_salary_index].EmpID);
    printf("Total Salary: %.2f\n", highest_salary);

    return 0;
}
```

Student details - Nested Structure

```
#include <stdio.h>

// Structure definition for Date of Birth
struct Date {
    int day;
    int month;
    int year;
};

// Structure definition for Student
struct Student {
    char name[50];
    char USN[15];
    char gender;
    struct Date dob; // Date of Birth
    float marks_S1;
    float marks_S2;
    float marks_S3;
};

int main() {
    // Declare an array of 50 students
    struct Student students[50];

    // Initialize information for all 50 students using a loop
```

```
    for (int i = 0; i < 50; i++) {
        printf("Enter details for student %d:\n", i + 1);

        printf("Name: ");
        scanf("%s", students[i].name);

        printf("USN: ");
        scanf("%s", students[i].USN);

        printf("Gender: ");
        scanf(" %c", &students[i].gender);

        printf("Date of Birth (DD MM YYYY): ");
        scanf("%d %d %d", &students[i].dob.day, &students[i].dob.month,
&students[i].dob.year);

        printf("Marks in Subject 1: ");
        scanf("%f", &students[i].marks_S1);

        printf("Marks in Subject 2: ");
        scanf("%f", &students[i].marks_S2);

        printf("Marks in Subject 3: ");
        scanf("%f", &students[i].marks_S3);
    }

    // Display the information for all 50 students
    printf("\nStudent Information:\n");
    for (int i = 0; i < 50; i++) {
        printf("Student %d:\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("USN: %s\n", students[i].USN);
        printf("Gender: %c\n", students[i].gender);
        printf("Date of Birth: %d-%d-%d\n", students[i].dob.day,
students[i].dob.month, students[i].dob.year);
        printf("Marks in Subject 1: %.2f\n", students[i].marks_S1);
        printf("Marks in Subject 2: %.2f\n", students[i].marks_S2);
        printf("Marks in Subject 3: %.2f\n", students[i].marks_S3);
        printf("\n");
    }

    return 0;
}
```

Self referential Stuctue

```
#include <stdio.h>
#include <stdlib.h>
```

```c
// Define the structure for a linked list node
struct node {
    int info;
    struct node* link; // Self-referential pointer to the next node
};

int main() {
    // Create nodes for a linked list
    struct node* head = NULL;
    struct node* second = NULL;
    struct node* third = NULL;

    // Allocate memory for nodes
    head = (struct node*)malloc(sizeof(struct node));
    second = (struct node*)malloc(sizeof(struct node));
    third = (struct node*)malloc(sizeof(struct node));

    // Assign data and set link pointers
    head->info = 1;
    head->link = second;

    second->info = 2;
    second->link = third;

    third->info = 3;
    third->link = NULL; // Last node points to NULL indicating end of the list

    // Traversing the linked list and printing data
    struct node* current = head;
    printf("Linked list data: ");
    while (current != NULL) {
        printf("%d ", current->info);
        current = current->link;
    }

    // Free dynamically allocated memory
    free(head);
    free(second);
    free(third);

    return 0;
}
```

Pattern

KMP alogorithm

```c
#include <stdio.h>
#include <string.h>
```

```c
#define MAX_LENGTH 1000

// Function prototypes
void fail(char *pat);
int pmatch(char *string, char *pat);

// Failure function array
int failure[MAX_LENGTH];

int main() {
    char text[] = "abcxabcdabxabcdabcdabcy";
    char pattern[] = "abcdabcy";

    int index = pmatch(text, pattern);

    if (index != -1) {
        printf("Pattern found at index: %d\n", index);
    } else {
        printf("Pattern not found in the text.\n");
    }

    return 0;
}

// Knuth-Morris-Pratt algorithm
int pmatch(char *string, char *pat) {
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    fail(pat); // Preprocess failure function

    while (i < lens && j < lenp) {
        if (string[i] == pat[j]) {
            i++;
            j++;
        } else if (j == 0) {
            i++;
        } else {
            j = failure[j - 1] + 1;
        }
    }

    return (j == lenp) ? (i - lenp) : -1;
}

// Failure function calculation
void fail(char *pat) {
    int n = strlen(pat);
    failure[0] = -1;
    for (int j = 1; j < n; j++) {
        int i = failure[j - 1];
```

```
        while ((pat[j] != pat[i + 1]) && (i >= 0)) {
            i = failure[i];
        }
        if (pat[j] == pat[i + 1]) {
            failure[j] = i + 1;
        } else {
            failure[j] = -1;
        }
    }
}
```

Normal pattern matching

```
#include <stdio.h>
#include <string.h>

int nfind(char *string, char *pat);

int main() {
    char text[1000], pattern[1000];
    int result;

    printf("Enter the text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0';  // Remove trailing newline character

    printf("Enter the pattern to search: ");
    fgets(pattern, sizeof(pattern), stdin);
    pattern[strcspn(pattern, "\n")] = '\0';  // Remove trailing newline character

    result = nfind(text, pattern);

    if (result != -1) {
        printf("Pattern found at index %d\n", result);
    } else {
        printf("Pattern not found in the text.\n");
    }

    return 0;
}

int nfind(char *string, char *pat) {
    int i, j, start = 0;
    int lasts = strlen(string) - 1;
    int lastp = strlen(pat) - 1;
    int endmatch = lastp;

    for (i = 0; endmatch <= lasts; endmatch++, start++) {
        if (string[endmatch] == pat[lastp]) {
            for (j = 0, i = start; j < lastp && string[i] == pat[j]; i++, j++) {
                // Empty loop body
```

```
            }
            if (j == lastp) {
                return start;
            }
        }
    }
    return -1; // Pattern not found
}
```

String Operation

```c
#include <stdio.h>
#include <string.h>

int main() {
    // Initialize strings
    char str1[50] = "Hello, World!";
    char str2[50] = "World";
    char str3[50];
    char str4[50];

    // Reverse string
    strrev(str1);
    printf("Reversed string: %s\n", str1);

    // Copy string
    strcpy(str3, str2);
    printf("Copied string: %s\n", str3);

    // Concatenate strings
    strcat(str1, " ");
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);

    // Compare strings
    int result = strcmp(str1, str3);
    if (result < 0) {
        printf("str1 is less than str3\n");
    } else if (result > 0) {
        printf("str1 is greater than str3\n");
    } else {
        printf("str1 is equal to str3\n");
    }

    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Hello, World!";
    char substr[10];

    substr(substr, str, 0, 5); // Extract the first 5 characters from str
    printf("Substring: %s\n", substr); // Output: Substring: Hello

    substr(substr, str, 7, 5); // Extract the characters from the 7th position
with a length of 5
    printf("Substring: %s\n", substr); // Output: Substring: World

    return 0;
}
```

Polynomial

```c
void padd(int starta, int finisha, int startb, int finishb, int *startd, int
*finishd)
{

    float coefficient;
    *startd = avail;

    while (starta <= finisha && startb <= finishb) {
        switch (COMPARE(terms[starta].expon, terms[startb].expon)) {
            case -1:
                attach(terms[startb].coef, terms[startb].expon);
                startb++;
                break;
            case 0:
                coefficient = terms[starta].coef + terms[startb].coef;
                if (coefficient) {
                    attach(coefficient, terms[starta].expon);
                }
                starta++;
                startb++;
                break;
            case 1:
                attach(terms[starta].coef, terms[starta].expon);
                starta++;
                break;
        }
    }


    for (; starta <= finisha; starta++) {
```

```
        attach(terms[starta].coef, terms[starta].expon);
    }


    for (; startb <= finishb; startb++) {
        attach(terms[startb].coef, terms[startb].expon);
    }

    *finishd = avail - 1;
}

void attach(float coefficient, int exponent)
{
    if (avail >= MAX_TERMS) {
        fprintf(stderr, "Too many terms in the polynomial\n");
        exit(EXIT_FAILURE);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

Sparse Matrix

Transpose

```
#include <stdio.h>

#define MAX_TERMS 101
#define MAX_COL 10

typedef struct {
    int row;
    int col;
    int value;
} Term;

void transpose(Term a[], Term b[]);

int main() {
    int row, col, num, i, j;
    Term a[MAX_TERMS], b[MAX_TERMS];

    printf("Enter number of rows and columns of matrix: ");
    scanf("%d %d", &row, &col);

    printf("Enter number of non-zero elements: ");
    scanf("%d", &num);

    printf("Enter row, column, and value of non-zero elements:\n");
```

```
    for (i = 0; i < num; i++) {
        scanf("%d %d %d", &a[i].row, &a[i].col, &a[i].value);
    }

    transpose(a, b);

    printf("\nTranspose of the matrix:\n");
    printf("Row\tColumn\tValue\n");
    for (i = 0; i < num; i++) {
        printf("%d\t%d\t%d\n", b[i].row, b[i].col, b[i].value);
    }

    return 0;
}

void transpose(Term a[], Term b[]) {
    int i, j, num, col, row_terms[MAX_COL], start_pos[MAX_COL];
    num = a[0].value;
    col = a[0].col;

    b[0].row = col;
    b[0].col = a[0].row;
    b[0].value = num;

    if (num > 0) {
        for (i = 0; i < col; i++) {
            row_terms[i] = 0;
        }
        for (i = 1; i <= num; i++) {
            row_terms[a[i].col]++;
        }
        start_pos[0] = 1;
        for (i = 1; i < col; i++) {
            start_pos[i] = start_pos[i - 1] + row_terms[i - 1];
        }
        for (i = 1; i <= num; i++) {
            j = start_pos[a[i].col]++;
            b[j].row = a[i].col;
            b[j].col = a[i].row;
            b[j].value = a[i].value;
        }
    }
}
```

Multiplication

```
#include <stdio.h>

#define MAX_TERMS 101
#define MAX_COL 10
```

```c
typedef struct {
    int row;
    int col;
    int value;
} Term;

void multiply(Term a[], Term b[], Term c[]);

int main() {
    int row1, col1, num1, row2, col2, num2, i;
    Term a[MAX_TERMS], b[MAX_TERMS], c[MAX_TERMS];

    printf("Enter number of rows and columns of first matrix: ");
    scanf("%d %d", &row1, &col1);

    printf("Enter number of non-zero elements in first matrix: ");
    scanf("%d", &num1);

    printf("Enter row, column, and value of non-zero elements of first
matrix:\n");
    for (i = 0; i < num1; i++) {
        scanf("%d %d %d", &a[i].row, &a[i].col, &a[i].value);
    }

    printf("Enter number of rows and columns of second matrix: ");
    scanf("%d %d", &row2, &col2);

    printf("Enter number of non-zero elements in second matrix: ");
    scanf("%d", &num2);

    printf("Enter row, column, and value of non-zero elements of second
matrix:\n");
    for (i = 0; i < num2; i++) {
        scanf("%d %d %d", &b[i].row, &b[i].col, &b[i].value);
    }

    multiply(a, b, c);

    printf("\nResult of matrix multiplication:\n");
    printf("Row\tColumn\tValue\n");
    for (i = 0; i < c[0].value; i++) {
        printf("%d\t%d\t%d\n", c[i].row, c[i].col, c[i].value);
    }

    return 0;
}

void multiply(Term a[], Term b[], Term c[]) {
    int i, j, k, col, total_terms, current_row, current_value;

    if (a[0].col != b[0].row) {
        printf("Error: Number of columns in first matrix must be equal to number
```

```
of rows in second matrix\n");
        return;
    }

    total_terms = 0;
    col = b[0].col;
    for (i = 0; i < a[0].row; i++) {
        for (j = 0; j < col; j++) {
            current_value = 0;
            for (k = 0; k < a[0].col; k++) {
                current_value += (a[i * a[0].col + k + 1].value * b[j + k * col +
1].value);
            }
            if (current_value != 0) {
                total_terms++;
                c[total_terms].row = i;
                c[total_terms].col = j;
                c[total_terms].value = current_value;
            }
        }
    }

    c[0].row = a[0].row;
    c[0].col = b[0].col;
    c[0].value = total_terms;
}
```