Q1

a. Define Data Structures. Explain with neat block diagrams different types of data structures with examples. What are the primitive operations that can be performed?

Answer:
Data structures are ways of organizing and storing data so that they can be accessed and modified efficiently. They define the layout of data in memory and the operations that can be performed on them.

Types of Data Structures:

1. **Primitive Data Structures:**
    - **Examples:** int, float, char, double.
    - **Operations:** assignment, addition, subtraction, etc.
2. **Non-Primitive Data Structures:**
    - **Linear Data Structures:**
        - **Arrays:** Contiguous memory allocation, easy to index.
        - **Linked Lists:** Non-contiguous memory, dynamic size.
        - **Stacks:** LIFO (Last In First Out).
        - **Queues:** FIFO (First In First Out).
    - **Non-Linear Data Structures:**
        - **Trees:** Hierarchical structure, consists of nodes.
        - **Graphs:** Consists of vertices and edges, can be directed or undirected.

Primitive Operations:

- Insertion, deletion, traversal, searching, updating.

b. Differentiate between stack and queue structures giving examples of both.

Answer:

- **Stack:**
  - **Definition:** A linear data structure following LIFO (Last In First Out).
  - **Operations:** push (insert), pop (delete), peek (retrieve top element).
  - **Example:** Stack of books, browser history.
- **Queue:**
  - **Definition:** A linear data structure following FIFO (First In First Out).
  - **Operations:** enqueue (insert), dequeue (delete), front (retrieve first element).
  - **Example:** Line of people at a ticket counter, printer queue.

c. What do you mean by pattern matching? Outline the Naive, Rabin-Karp, and Knuth-Morris-Pratt algorithms.

Answer:
Pattern matching refers to the process of checking if a specific sequence of characters (pattern) exists within a larger sequence (text).

- **Naive Algorithm:**
  - **Approach:** Check each position in the text to see if the pattern matches.
  - **Complexity:** O((n-m+1)m), where n is the length of the text, and m is the length of the pattern.
- **Rabin-Karp Algorithm:**
  - **Approach:** Use hash values to find any substrings of the text that match the hash value of the pattern.
  - **Complexity:** O(nm) in the worst case, O(n+m) on average.
- **Knuth-Morris-Pratt (KMP) Algorithm:**
  - **Approach:** Use preprocessing to create a partial match table (prefix function) to skip unnecessary comparisons.
  - **Complexity:** O(n + m).

Module 2

Q2

a. Define stack. Explain the implementation of push(), pop() and display() functions by using arrays for empty and full conditions.

Answer:
A stack is a linear data structure that follows the LIFO (Last In First Out) principle.

- **push() Function:**

```
void push(int stack[], int *top, int value) { if (*top == MAX - 1) { printf("Stack Overflow\n"); } else { stack[++(*top)] = value; } }
```

- **pop() Function:**

```
int pop(int stack[], int *top) { if (*top == -1) { printf("Stack Underflow\n"); return -1; } else { return stack[(*top)--]; } }
```

- **display() Function:**

```
void display(int stack[], int top) { if (top == -1) { printf("Stack is empty\n"); } else { for (int i = 0; i <= top; i++) { printf("%d
", stack[i]); } printf("\n"); } }
```

b. Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression: 6 3 * 2 1 + /

Answer:
Algorithm:

1. Initialize an empty stack.
2. Scan the postfix expression from left to right.
3. For each token:
    - If the token is an operand, push it onto the stack.
    - If the token is an operator, pop the top two elements from the stack, apply the operator, and push the result back onto the stack.
4. The final result will be the only element remaining in the stack.

Example:
Postfix Expression: 6 3 * 2 1 + /

Steps:

1. Push 6: [6]
2. Push 3: [6, 3]
3. Multiply: 6 * 3 = 18, push 18: [18]
4. Push 2: [18, 2]
5. Push 1: [18, 2, 1]
6. Add: 2 + 1 = 3, push 3: [18, 3]
7. Divide: 18 / 3 = 6, push 6: [6]
8. Result: 6

c. Write a prefix form for the following using stack:

Given infix: A + ((B * C) / D) - E + (F * (G + H / I))

Answer:

1. A + ((B * C) / D) - E + (F * (G + H / I))
2. Postfix: A B C * D / + E - F G H I / + * +

Prefix: + - + A / * B C D E * F + G / H I

Module 3

Q3

a. What are the disadvantages of an ordinary queue? Explain the implementation of circular queues.

Answer:
Disadvantages of Ordinary Queue:

- Inefficiency: After multiple enqueue and dequeue operations, front and rear pointers may drift towards the end of the array, wasting space.
- Overflow: An empty space in the front cannot be reused unless the queue is reset.

Circular Queue Implementation:

1. Use a circular array.
2. Increment rear and front pointers modulo the size of the array.
3. Enqueue operation: Add element to the position pointed by rear, then update rear = (rear + 1) % size.
4. Dequeue operation: Remove element from the position pointed by front, then update front = (front + 1) % size.

b. Write a C function to reverse a string using a stack.

Answer:

```
#include <stdio.h> #include <string.h> #define MAX 100 void push(char stack[], int *top, char value) { if (*top == MAX - 1) { printf("Stack Overflow\n"); } else { stack[++(*top)] = value; } } char pop(char stack[], int *top) { if (*top == -1) { printf("Stack Underflow\n"); return '\0'; } else { return stack[(*top)--]; } } void reverseString(char str[]) { int n = strlen(str); char stack[MAX]; int top = -1; for (int i = 0; i < n; i++) { push(stack, &top, str[i]); } for (int i = 0; i < n; i++) { str[i] = pop(stack, &top); } } int main() { char str[] = "Hello, World!"; reverseString(str); printf("Reversed String: %s\n", str); return 0; }
```

c. Define queue. Explain how to represent queues using dynamic arrays.

Answer:
Queue: A queue is a linear data structure that follows the FIFO (First In First Out) principle.

Dynamic Array Implementation:

1. Initialize an array with a dynamic size.
2. Use front and rear pointers to keep track of the first and last elements.
3. When the array is full, create a new array with double the size, copy the elements, and update the front and rear pointers.
4. Enqueue operation: Add element to the position pointed by rear, then update rear.
5. Dequeue operation: Remove element from the position pointed by front, then update front.

Q4

a. What is a linked list? Explain the different types of linked lists with neat diagrams.

Answer:
Linked List: A linked list is a linear data structure where elements (nodes) are linked using pointers. Each node contains data and a reference (pointer) to the next node in the sequence.

Types of Linked Lists:

1. **Singly Linked List:**
   - **Structure:** Each node points to the next node.
   - **Diagram:**

```
[data|next] -> [data|next] -> [data|next] -> NULL
```

2. **Doubly Linked List:**
   - **Structure:** Each node points to both the next and previous nodes.
   - **Diagram:**

```
NULL <- [prev|data|next] <-> [prev|data|next] <-> [prev|data|next] -> NULL
```

3. **Circular Linked List:**
   - **Structure:** The last node points back to the first node.
   - **Diagram:**

```
[data|next] -> [data|next] -> [data|next] --+ ^ | |-----------------------------------+
```

4. **Circular Doubly Linked List:**
   - **Structure:** Each node points to both the next and previous nodes, and the last node points back to the first node.
   - **Diagram:**

```
NULL <- [prev|data|next] <-> [prev|data|next] <-> [prev|data|next] -> NULL ^ | |-----------------------------------------------
-----------------------+
```

b. Give the structure declaration of a singly linked list (SLL). Write a function to:

- **(i) Insert an element at the end of SLL.**
- **(ii) Delete a node at the beginning of SLL.**

Answer:

```
```

```
#include <stdio.h> #include <stdlib.h> struct Node { int data; struct Node* next; }; // Insert at the end void insertEnd(struct Node** head, int data) { struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); struct Node* last = *head; newNode->data = data; newNode->next = NULL; if (*head == NULL) { *head = newNode; return; } while (last->next != NULL) { last = last->next; } last->next = newNode; } // Delete from the beginning void deleteBeginning(struct Node** head) { if (*head == NULL) { printf("List is empty\n"); return; } struct Node* temp = *head; *head = (*head)->next; free(temp); }
```

c. Write a C function to add two polynomials. Show the linked list representation of the two polynomials:

Given:

$p(x) = 3x^5 + 2x^3 + x + 1$

$q(x) = 5x^5 + 4x^2 + 2x + 6$

Answer:

```
#include <stdio.h> #include <stdlib.h> struct PolyNode { int coeff; int power; struct PolyNode* next; }; struct PolyNode* createNode(int coeff, int power) { struct PolyNode* newNode = (struct PolyNode*)malloc(sizeof(struct PolyNode)); newNode->coeff = coeff; newNode->power = power; newNode->next = NULL; return newNode; } void appendNode(struct PolyNode** head, int coeff, int power) { struct PolyNode* newNode = createNode(coeff, power); struct PolyNode* last = *head; if (*head == NULL) { *head = newNode; return; } while (last->next != NULL) { last = last->next; } last->next = newNode; } struct PolyNode* addPolynomials(struct PolyNode* poly1, struct PolyNode* poly2) { struct PolyNode* result = NULL; while (poly1 != NULL && poly2 != NULL) { if (poly1->power > poly2->power) { appendNode(&result, poly1->coeff, poly1->power); poly1 = poly1->next; } else if (poly1->power < poly2->power) { appendNode(&result, poly2->coeff, poly2->power); poly2 = poly2->next; } else { appendNode(&result, poly1->coeff + poly2->coeff, poly1->power); poly1 = poly1->next; poly2 = poly2->next; } } while (poly1 != NULL) { appendNode(&result, poly1->coeff, poly1->power); poly1 = poly1->next; } while (poly2 != NULL) { appendNode(&result, poly2->coeff, poly2->power); poly2 = poly2->next; } return result; } void printPolynomial(struct PolyNode* poly) { while (poly != NULL) { printf("%dx^%d", poly->coeff, poly->power); if (poly->next != NULL) printf(" + "); poly = poly->next; } printf("\n"); } int main() { struct PolyNode* poly1 = NULL; struct PolyNode* poly2 = NULL; // Polynomial p(x) = 3x^5 + 2x^3 + x + 1 appendNode(&poly1, 3, 5); appendNode(&poly1, 2, 3); appendNode(&poly1, 1, 1); appendNode(&poly1, 1, 0); // Polynomial q(x) = 5x^5 + 4x^2 + 2x + 6 appendNode(&poly2, 5, 5); appendNode(&poly2, 4, 2); appendNode(&poly2, 2, 1); appendNode(&poly2, 6, 0); struct PolyNode* result = addPolynomials(poly1, poly2); printf("Polynomial p(x): "); printPolynomial(poly1); printf("Polynomial q(x): "); printPolynomial(poly2); printf("Sum: "); printPolynomial(result); return 0; }
```

Module 4

Q5

a. Write a C function for the following operations on a doubly linked list (DLL):

- (i) Addition of a node.
- (ii) Concatenation of two DLLs.

Answer:

```
#include <stdio.h> #include <stdlib.h> struct Node { int data; struct Node* prev; struct Node* next; }; // Add node at the end void addNode(struct Node** head, int data) { struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); struct Node* last = *head; newNode->data = data; newNode->next = NULL; if (*head == NULL) { newNode->prev = NULL; *head = newNode; return; } while (last->next != NULL) { last = last->next; } last->next = newNode; newNode->prev = last; } // Concatenate two DLLs void concatenate(struct Node** head1, struct Node* head2) { if (*head1 == NULL) { *head1 = head2; return; } struct Node* last = *head1; while (last->next != NULL) { last = last->next; } last->next = head2; if (head2 != NULL) { head2->prev = last; } } void printList(struct Node* node) { while (node != NULL) { printf("%d ", node->data); node = node->next; } printf("\n"); } int main() { struct Node* dll1 = NULL; struct Node* dll2 = NULL; // DLL 1 addNode(&dll1, 1); addNode(&dll1, 2); addNode(&dll1, 3); // DLL 2 addNode(&dll2, 4); addNode(&dll2, 5); addNode(&dll2, 6); printf("DLL 1: "); printList(dll1); printf("DLL 2: "); printList(dll2); concatenate(&dll1, dll2); printf("Concatenated DLL: "); printList(dll1); return 0; }
```

b. Write C functions for the following operations on circular linked list:

- (i) Inserting at the front of the list.
- (ii) Deleting the last node of a circular list.

Answer:

```
```

```c
#include <stdio.h> #include <stdlib.h> struct Node { int data; struct Node* next; }; // Insert at the front void insertFront(struct Node** head, int data) { struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); struct Node* last = *head; newNode->data = data; newNode->next = *head; if (*head != NULL) { while (last->next != *head) { last = last->next; } last->next = newNode; } else { newNode->next = newNode; } *head = newNode; } // Delete the last node void deleteLast(struct Node** head) { if (*head == NULL) { printf("List is empty\n"); return; } struct Node* temp = *head; struct Node* prev = NULL; if (temp->next == *head) { free(temp); *head = NULL; return; } while (temp->next != *head) { prev = temp; temp = temp->next; } prev->next = *head; free(temp); } void printList(struct Node* head) { struct Node* temp = head; if (head != NULL) { do { printf("%d ", temp->data); temp = temp->next; } while (temp != head); } printf("\n"); } int main() { struct Node* cll = NULL; // Insert at the front insertFront(&cll, 1); insertFront(&cll, 2); insertFront(&cll, 3); printf("Circular Linked List: "); printList(cll); // Delete the last node deleteLast(&cll); printf("After deleting last node: "); printList(cll); return 0; }
```

Module 5

Q6

a. Define AVL tree. Write a C function to perform single rotations in AVL trees.

Answer:
AVL Tree: An AVL tree is a self-balancing binary search tree where the difference in heights of left and right subtrees of any node is at most one.

Single Rotations:

1. **Left Rotation (LL Rotation):**

```c
struct Node* leftRotate(struct Node* x) { struct Node* y = x->right; struct Node* T2 = y->left; // Perform rotation y->left = x; x->right = T2; // Update heights x->height = max(height(x->left), height(x->right)) + 1; y->height = max(height(y->left), height(y->right)) + 1; // Return new root return y; }
```

2. **Right Rotation (RR Rotation):**

```
struct Node* rightRotate(struct Node* y) { struct Node* x = y->left; struct Node* T2 = x->right; // Perform rotation x->right = y; y->left = T2; // Update heights y->height = max(height(y->left), height(y->right)) + 1; x->height = max(height(x->left), height(x->right)) + 1; // Return new root return x; }
```

b. What are threaded binary trees? Explain with an example.

Answer:
Threaded Binary Trees: In a threaded binary tree, null pointers are replaced with pointers to the in-order predecessor or successor to facilitate in-order traversal without using a stack or recursion.

Example:
Consider the following binary tree:

```
10 / \ 5 20 / \ 3 30
```

Threaded Binary Tree Representation:

```
10 / \ 5 20 / \ 3 30
```

In the above example, the left child of 5 and the right child of 20 are threaded to point to their respective in-order predecessors and successors.

c. Write a C function to perform insertion in a binary search tree (BST).

Answer:

```
#include <stdio.h> #include <stdlib.h> struct Node { int data; struct Node* left; struct Node* right; }; struct Node* createNode(int data) { struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); newNode->data = data; newNode->left = newNode->right = NULL; return newNode; } struct Node* insert(struct Node* root, int data) { if (root == NULL) { return createNode(data); } if (data < root->data) { root->left = insert(root->left, data); } else if (data > root->data) { root->right = insert(root->right, data); } return root; } void inorder(struct Node* root) { if (root != NULL) { inorder(root->left); printf("%d ", root->data); inorder(root->right); } } int main() { struct Node* root = NULL; root = insert(root, 50); insert(root, 30); insert(root, 20); insert(root, 40); insert(root, 70); insert(root, 60); insert(root, 80); printf("Inorder traversal: "); inorder(root); return 0; }
```

Module 6

Q7

a. Define graph. Explain different ways to represent graphs.

Answer:
Graph: A graph is a collection of vertices (nodes) and edges (connections between nodes).

Representation of Graphs:

1. **Adjacency Matrix:**
   - **Description:** A 2D array where `matrix[i][j]` is 1 if there is an edge from vertex i to vertex j, otherwise 0.
   - **Example:**

```
0 1 0 1 0 1 0 1 0
```

2. **Adjacency List:**
   - **Description:** An array of linked lists. Each index represents a vertex, and the linked list at that index represents all adjacent vertices.

- Example:

```
0 -> 1 1 -> 0 -> 2 2 -> 1
```

3. **Incidence Matrix:**

- **Description:** A 2D array where rows represent vertices and columns represent edges. **`matrix[i][j]`** is 1 if vertex i is incident to edge j, otherwise 0.
- Example:

```
1 0 1 1 1 0 0 1 1
```

b. Write a C function to perform DFS traversal on a graph.

Answer:

```
#include <stdio.h> #include <stdlib.h> #define MAX 100 int adj[MAX][MAX]; int visited[MAX]; int n; // Number of vertices void DFS(int v) { visited[v] = 1; printf("%d ", v); for (int i = 0; i < n; i++) { if (adj[v][i] == 1 && !visited[i]) { DFS(i); } } } int main() { printf("Enter the number of vertices: "); scanf("%d", &n); printf("Enter the adjacency matrix:\n"); for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { scanf("%d", &adj[i][j]); } } for (int i = 0; i < n; i++) { visited[i] = 0; } printf("DFS Traversal starting from vertex 0:\n"); DFS(0); return 0; }
```

c. Write a C function to perform BFS traversal on a graph.

Answer:

```
```

```c
#include <stdio.h> #include <stdlib.h> #define MAX 100 int adj[MAX][MAX]; int visited[MAX]; int n; // Number of vertices void BFS(int start) { int queue[MAX], front = 0, rear = -1; visited[start] = 1; queue[++rear] = start; while (front <= rear) { int v = queue[front++]; printf("%d ", v); for (int i = 0; i < n; i++) { if (adj[v][i] == 1 && !visited[i]) { queue[++rear] = i; visited[i] = 1; } } } } int main() { printf("Enter the number of vertices: "); scanf("%d", &n); printf("Enter the adjacency matrix:\n"); for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { scanf("%d", &adj[i][j]); } } for (int i = 0; i < n; i++) { visited[i] = 0; } printf("BFS Traversal starting from vertex 0:\n"); BFS(0); return 0; }
```

Module 7

Q8

a. Define sorting. Explain quicksort algorithm with an example.

Answer:
Sorting: Sorting is the process of arranging elements in a specific order (ascending or descending).

Quicksort Algorithm:

1. Choose a pivot element.
2. Partition the array into two sub-arrays such that elements less than the pivot are on the left and elements greater than the pivot are on the right.
3. Recursively apply the same steps to the left and right sub-arrays.

Example:
Array: [10, 80, 30, 90, 40, 50, 70]

Steps:

1. Choose pivot: 50
2. Partition: [10, 30, 40, 50, 90, 80, 70]
3. Recursively sort left and right sub-arrays.

b. Write a C function to perform merge sort.

Answer:

```
#include <stdio.h> #include <stdlib.h> void merge(int arr[], int l, int m, int r) { int n1 = m - l + 1; int n2 = r - m; int L[n1], R[n2]; for (int i = 0; i < n1; i++) { L[i] = arr[l + i]; } for (int j = 0; j < n2; j++) { R[j] = arr[m + 1 + j]; } int i = 0, j = 0, k = l; while (i < n1 && j < n2) { if (L[i] <= R[j]) { arr[k++] = L[i++]; } else { arr[k++] = R[j++]; } } while (i < n1) { arr[k++] = L[i++]; } while (j < n2) { arr[k++] = R[j++]; } } void mergeSort(int arr[], int l, int r) { if (l < r) { int m = l + (r - l) / 2; mergeSort(arr, l, m); mergeSort(arr, m + 1, r); merge(arr, l, m, r); } } void printArray(int arr[], int size) { for (int i = 0; i < size; i++) { printf("%d ", arr[i]); } printf("\n"); } int main() { int arr[] = {12, 11, 13, 5, 6, 7}; int arr_size = sizeof(arr) / sizeof(arr[0]); printf("Given array is \n"); printArray(arr, arr_size); mergeSort(arr, 0, arr_size - 1); printf("Sorted array is \n"); printArray(arr, arr_size); return 0; }
```

c. What are the applications of sorting?

Answer:

1. **Searching:** Sorted data allows efficient searching algorithms like binary search.
2. **Data Organization:** Sorting helps in organizing data for better readability and management.
3. **Algorithms:** Many algorithms like median finding and convex hull rely on sorting.
4. **Database Management:** Sorting is crucial for operations like indexing and query optimization.
5. **Data Analysis:** Sorting is used in data analysis for finding statistical values and trends.