10.implement Binary search Tree and its operations (creation, insertion, deletion).

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

// Node structure definition
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Insert a node in the BST
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) {
        root = createNode(data);
        return root;
    }
```

```
    if (data < root->data)
        root->left = insertNode(root->left, data);
    else if (data > root->data)
        root->right = insertNode(root->right, data);

    return root;
}

// Find the minimum value node (used in deletion)
struct Node* findMin(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

// Delete a node in the BST
struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else {
        // Node with one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
```

```c
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: get the inorder successor (smallest in the right subtree)
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// In-order traversal (left, root, right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    root = insertNode(root, 50);
    insertNode(root, 30);
```

```c
insertNode(root, 20);
insertNode(root, 40);
insertNode(root, 70);
insertNode(root, 60);
insertNode(root, 80);

printf("Inorder traversal of the BST: ");
inorderTraversal(root);
printf("\n");

printf("\nDelete 20\n");
root = deleteNode(root, 20);
printf("Inorder traversal after deleting 20: ");
inorderTraversal(root);
printf("\n");

printf("\nDelete 30\n");
root = deleteNode(root, 30);
printf("Inorder traversal after deleting 30: ");
inorderTraversal(root);
printf("\n");

printf("\nDelete 50\n");
root = deleteNode(root, 50);
printf("Inorder traversal after deleting 50: ");
inorderTraversal(root);
printf("\n");

return 0;
```

```
}
```

11. Implement Traversals Preorder Inorder Postorder on BST.

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

// Node structure definition
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

// Insert a node in the BST
struct Node* insertNode(struct Node* root, int data) {
   if (root == NULL) {
      root = createNode(data);
      return root;
```

```
    }

    if (data < root->data)
        root->left = insertNode(root->left, data);
    else if (data > root->data)
        root->right = insertNode(root->right, data);

    return root;
}

// Inorder Traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Preorder Traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

// Postorder Traversal (Left, Right, Root)
```

```
void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    struct Node* root = NULL;
    root = insertNode(root, 50);
    insertNode(root, 30);
    insertNode(root, 20);
    insertNode(root, 40);
    insertNode(root, 70);
    insertNode(root, 60);
    insertNode(root, 80);

    printf("Inorder traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");
```

```
    return 0;

}
```

12. Implement Graphs and represent using adjaceny list and adjacency matrix and implement basic operations with traversals
(BFS and DFS).

```
#include <stdio.h>

#include <stdlib.h>


#define MAX 100


// Node structure for adjacency list

struct Node {

    int vertex;

    struct Node* next;

};


// Graph structure using adjacency list

struct GraphList {

    int numVertices;

    struct Node** adjLists;

    int* visited;

};


// Graph structure using adjacency matrix

struct GraphMatrix {

    int numVertices;

    int adjMatrix[MAX][MAX];
```

```
};

// Function to create a node for adjacency list
struct Node* createNode(int v) {
   struct Node* newNode = malloc(sizeof(struct Node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}

// Function to create a graph with n vertices (Adjacency List)
struct GraphList* createGraphList(int vertices) {
   struct GraphList* graph = malloc(sizeof(struct GraphList));
   graph->numVertices = vertices;
   graph->adjLists = malloc(vertices * sizeof(struct Node*));
   graph->visited = malloc(vertices * sizeof(int));

   for (int i = 0; i < vertices; i++) {
      graph->adjLists[i] = NULL;
      graph->visited[i] = 0;
   }
   return graph;
}

// Function to create a graph with n vertices (Adjacency Matrix)
struct GraphMatrix* createGraphMatrix(int vertices) {
   struct GraphMatrix* graph = malloc(sizeof(struct GraphMatrix));
   graph->numVertices = vertices;
```

```
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            graph->adjMatrix[i][j] = 0;
        }
    }
    return graph;
}

// Add edge (Adjacency List)
void addEdgeList(struct GraphList* graph, int src, int dest) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src (Undirected Graph)
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Add edge (Adjacency Matrix)
void addEdgeMatrix(struct GraphMatrix* graph, int src, int dest) {
    graph->adjMatrix[src][dest] = 1;
    graph->adjMatrix[dest][src] = 1; // For undirected graph
}

// Print the adjacency list representation of the graph
void printGraphList(struct GraphList* graph) {
```

```
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

// Print the adjacency matrix representation of the graph
void printGraphMatrix(struct GraphMatrix* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        for (int j = 0; j < graph->numVertices; j++) {
            printf("%d ", graph->adjMatrix[i][j]);
        }
        printf("\n");
    }
}

// BFS algorithm
void bfs(struct GraphList* graph, int startVertex) {
    int queue[MAX], front = -1, rear = -1;

    // Mark the starting vertex as visited and enqueue it
    graph->visited[startVertex] = 1;
    queue[++rear] = startVertex;
    front++;
```

```
    while (front <= rear) {
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);

        struct Node* temp = graph->adjLists[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                queue[++rear] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

// DFS algorithm
void dfs(struct GraphList* graph, int vertex) {
    struct Node* adjList = graph->adjLists[vertex];
    struct Node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("%d ", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->vertex;
```

```
        if (graph->visited[connectedVertex] == 0) {
            dfs(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

// Reset visited array
void resetVisited(struct GraphList* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        graph->visited[i] = 0;
    }
}

int main() {
    int vertices = 5;

    // Creating Graph using Adjacency List
    struct GraphList* graphList = createGraphList(vertices);
    addEdgeList(graphList, 0, 1);
    addEdgeList(graphList, 0, 4);
    addEdgeList(graphList, 1, 2);
    addEdgeList(graphList, 1, 3);
    addEdgeList(graphList, 1, 4);
    addEdgeList(graphList, 2, 3);
    addEdgeList(graphList, 3, 4);

    // Print adjacency list
```

```
    printf("Adjacency List Representation:\n");
    printGraphList(graphList);

    // Perform BFS and DFS
    printf("\nBFS Traversal starting from vertex 0:\n");
    bfs(graphList, 0);
    resetVisited(graphList);
    printf("\n\nDFS Traversal starting from vertex 0:\n");
    dfs(graphList, 0);

    // Creating Graph using Adjacency Matrix
    struct GraphMatrix* graphMatrix = createGraphMatrix(vertices);
    addEdgeMatrix(graphMatrix, 0, 1);
    addEdgeMatrix(graphMatrix, 0, 4);
    addEdgeMatrix(graphMatrix, 1, 2);
    addEdgeMatrix(graphMatrix, 1, 3);
    addEdgeMatrix(graphMatrix, 1, 4);
    addEdgeMatrix(graphMatrix, 2, 3);
    addEdgeMatrix(graphMatrix, 3, 4);

    // Print adjacency matrix
    printf("\n\nAdjacency Matrix Representation:\n");
    printGraphMatrix(graphMatrix);

    return 0;
}
```