1. Write a C program to create a binary tree using recursive function and display that level wise. Write a C program to identify the height of a binary tree. 5. Write a C program to identify degree of a given node. 6. Write a C program to count number of leaf node present in a binary tree. 7. Write a C program to count number of internal node present in a binary tree. 10. Write a C program to count number of siblings present in a binary tree. menu driven user input

```
#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 = NULL;

   newNode->right = NULL;

   return newNode;

}


struct Node* insert(struct Node* root, int data) {

   if (root == NULL) {

      root = createNode(data);

   } else {

      if (data <= root->data) {

         root->left = insert(root->left, data);

      } else {

         root->right = insert(root->right, data);

      }
```

```
    }
    return root;
}

void printLevelOrder(struct Node* root) {
    if (root == NULL) return;

    // Create an empty queue for level order traversal
    struct Node** queue = (struct Node**)malloc(100 * sizeof(struct Node*));
    int front = 0, rear = 0;
    queue[rear++] = root;

    while (front < rear) {
        struct Node* current = queue[front++];
        printf("%d ", current->data);

        if (current->left != NULL) {
            queue[rear++] = current->left;
        }
        if (current->right != NULL) {
            queue[rear++] = current->right;
        }
    }

    free(queue);
}

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
int findHeight(struct Node* root) {
   if (root == NULL) return -1;

   return max(findHeight(root->left), findHeight(root->right)) + 1;
}

int findDegree(struct Node* root, int data) {
   if (root == NULL) return -1;

   if (root->data == data) {
      if (root->left != NULL && root->right != NULL) {
         return 2;
      } else if (root->left != NULL || root->right != NULL) {
         return 1;
      } else {
         return 0;
      }
   }

   int leftDegree = findDegree(root->left, data);
   int rightDegree = findDegree(root->right, data);

   if (leftDegree != -1) return leftDegree;
   if (rightDegree != -1) return rightDegree;

   return -1;
}

int countLeafNodes(struct Node* root) {
   if (root == NULL) return 0;
```

```
    if (root->left == NULL && root->right == NULL) return 1;

    return countLeafNodes(root->left) + countLeafNodes(root->right);
}

int countInternalNodes(struct Node* root) {
    if (root == NULL || (root->left == NULL && root->right == NULL)) return 0;

    return 1 + countInternalNodes(root->left) + countInternalNodes(root->right);
}

int countSiblings(struct Node* root, int data) {
    if (root == NULL || (root->left == NULL && root->right == NULL)) return 0;

    if ((root->left != NULL && root->left->data == data) || (root->right != NULL && root->right->data ==
data)) {
        return 1;
    }

    return countSiblings(root->left, data) + countSiblings(root->right, data);
}

int main() {
    struct Node* root = NULL;
    int choice, data, nodeData;

    do {
        printf("\n\n1. Insert\n");
        printf("2. Display Level-wise\n");
        printf("3. Height of the Binary Tree\n");
        printf("4. Degree of a Given Node\n");
```

```
printf("5. Count Leaf Nodes\n");
printf("6. Count Internal Nodes\n");
printf("7. Count Siblings\n");
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch(choice) {
   case 1:
      printf("Enter data to insert: ");
      scanf("%d", &data);
      root = insert(root, data);
      break;
   case 2:
      printf("Level-wise display: ");
      printLevelOrder(root);
      break;
   case 3:
      printf("Height of the binary tree: %d", findHeight(root));
      break;
   case 4:
      printf("Enter node data to find its degree: ");
      scanf("%d", &nodeData);
      printf("Degree of %d: %d", nodeData, findDegree(root, nodeData));
      break;
   case 5:
      printf("Number of leaf nodes: %d", countLeafNodes(root));
      break;
   case 6:
      printf("Number of internal nodes: %d", countInternalNodes(root));
      break;
```

```
        case 7:
          printf("Enter node data to count its siblings: ");
          scanf("%d", &nodeData);
          printf("Number of siblings of %d: %d", nodeData, countSiblings(root, nodeData));
          break;
        case 8:
          printf("Exiting program...");
          break;
        default:
          printf("Invalid choice! Please enter a valid option.");
      }
    } while(choice != 8);

    return 0;
}
```

2. Write a C program to create a binary tree using non-recursive function and display that level wise. 3. Write a C program to create a binary tree using array only and display the tree level wise.

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

#define MAX_SIZE 100

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 = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
void insert(struct Node* root, int data) {
   struct Node* newNode = createNode(data);
   struct Node* current = root;
   struct Node* parent = NULL;

   while (current != NULL) {
      parent = current;
      if (data <= current->data) {
         current = current->left;
      } else {
         current = current->right;
      }
   }

   if (data <= parent->data) {
      parent->left = newNode;
   } else {
      parent->right = newNode;
   }
}

void printLevelOrder(struct Node* root) {
   if (root == NULL) return;

   struct Node* queue[MAX_SIZE];
   int front = 0, rear = 0;
   queue[rear++] = root;

   while (front < rear) {
      struct Node* current = queue[front++];
      printf("%d ", current->data);

      if (current->left != NULL) {
         queue[rear++] = current->left;
      }
      if (current->right != NULL) {
         queue[rear++] = current->right;
      }
   }
}

int main() {
   struct Node* root = createNode(10);
   insert(root, 5);
   insert(root, 15);
   insert(root, 3);
   insert(root, 7);
```

```
    insert(root, 12);
    insert(root, 20);

    printf("Level-wise display: ");
    printLevelOrder(root);

    return 0;
}
```

2. Create a binary tree using array only and display it level-wise:

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

#define MAX_SIZE 100

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 = NULL;
    newNode->right = NULL;
    return newNode;
}

struct Node* createBinaryTreeFromArray(int arr[], int size, int index) {
    struct Node* root = NULL;

    if (index < size) {
        root = createNode(arr[index]);
        root->left = createBinaryTreeFromArray(arr, size, 2 * index + 1);
        root->right = createBinaryTreeFromArray(arr, size, 2 * index + 2);
    }

    return root;
}

void printLevelOrder(struct Node* root) {
    if (root == NULL) return;

    struct Node* queue[MAX_SIZE];
    int front = 0, rear = 0;
```

```
    queue[rear++] = root;

    while (front < rear) {
        struct Node* current = queue[front++];
        printf("%d ", current->data);

        if (current->left != NULL) {
            queue[rear++] = current->left;
        }
        if (current->right != NULL) {
            queue[rear++] = current->right;
        }
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int size = sizeof(arr) / sizeof(arr[0]);

    struct Node* root = createBinaryTreeFromArray(arr, size, 0);

    printf("Level-wise display: ");
    printLevelOrder(root);

    return 0;
}
```

8. Write a C program to count number of node present in a given binary tree using linked list. 9. Write a C program to count number of node present in a given binary tree using array.

```
#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 = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
int countNodes(struct Node* root) {
  if (root == NULL) return 0;

  return 1 + countNodes(root->left) + countNodes(root->right);
}

int main() {
  struct Node* root = createNode(10);
  root->left = createNode(5);
  root->right = createNode(15);
  root->left->left = createNode(3);
  root->left->right = createNode(7);
  root->right->left = createNode(12);
  root->right->right = createNode(20);

  printf("Number of nodes in the binary tree: %d\n", countNodes(root));

  return 0;
}
```

2. Counting the number of nodes in a binary tree using an array:

```
#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 = NULL;
  newNode->right = NULL;
  return newNode;
}

struct Node* createBinaryTreeFromArray(int arr[], int size, int index) {
  struct Node* root = NULL;

  if (index < size) {
    root = createNode(arr[index]);
    root->left = createBinaryTreeFromArray(arr, size, 2 * index + 1);
    root->right = createBinaryTreeFromArray(arr, size, 2 * index + 2);
  }
```

```
    return root;
}

int countNodesArray(struct Node* root) {
    if (root == NULL) return 0;

    int count = 0;
    struct Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;

    while (front < rear) {
        struct Node* current = queue[front++];
        count++;

        if (current->left != NULL) {
            queue[rear++] = current->left;
        }
        if (current->right != NULL) {
            queue[rear++] = current->right;
        }
    }

    return count;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int size = sizeof(arr) / sizeof(arr[0]);

    struct Node* root = createBinaryTreeFromArray(arr, size, 0);

    printf("Number of nodes in the binary tree (using array): %d\n", countNodesArray(root));

    return 0;
}
```

ASSIGNMNET 3 Write a C program to create a binary search tree using recursive function and display that. Write a C program to insert (by using a function) a specific element into an existing binary search tree and then display that. 4. Write a C program to search an element in a BST and show the result. Write a C program to display a BST using In-order, Pre-order, Post-order. 8. Write a C program to Count the number of nodes present in an existing BST and display the highest element present in the BST. make it menu driven user input

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

```c
#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 = NULL;
   newNode->right = NULL;
   return newNode;
}

struct Node* insert(struct Node* root, int data) {
   if (root == NULL) {
      root = createNode(data);
   } else {
      if (data <= root->data) {
         root->left = insert(root->left, data);
      } else {
         root->right = insert(root->right, data);
      }
   }
   return root;
}

struct Node* search(struct Node* root, int key) {
   if (root == NULL || root->data == key) {
      return root;
   }

   if (key < root->data) {
      return search(root->left, key);
   }

   return search(root->right, key);
}

void inorder(struct Node* root) {
   if (root == NULL) return;

   inorder(root->left);
   printf("%d ", root->data);
   inorder(root->right);
}
```

```c
void preorder(struct Node* root) {
    if (root == NULL) return;

    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

void postorder(struct Node* root) {
    if (root == NULL) return;

    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}

int countNodes(struct Node* root) {
    if (root == NULL) return 0;

    return 1 + countNodes(root->left) + countNodes(root->right);
}

int findMax(struct Node* root) {
    if (root == NULL) {
        printf("The tree is empty.\n");
        return -1;
    }

    while (root->right != NULL) {
        root = root->right;
    }

    return root->data;
}

int main() {
    struct Node* root = NULL;
    int choice, data, key;

    do {
        printf("\n\n1. Insert element into BST\n");
        printf("2. Display BST (In-order)\n");
        printf("3. Display BST (Pre-order)\n");
        printf("4. Display BST (Post-order)\n");
        printf("5. Search element in BST\n");
        printf("6. Count nodes in BST\n");
        printf("7. Find highest element in BST\n");
```

```
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
            case 2:
                printf("BST (In-order): ");
                inorder(root);
                break;
            case 3:
                printf("BST (Pre-order): ");
                preorder(root);
                break;
            case 4:
                printf("BST (Post-order): ");
                postorder(root);
                break;
            case 5:
                printf("Enter element to search: ");
                scanf("%d", &key);
                if (search(root, key) != NULL) {
                    printf("%d found in BST.\n", key);
                } else {
                    printf("%d not found in BST.\n", key);
                }
                break;
            case 6:
                printf("Number of nodes in BST: %d\n", countNodes(root));
                break;
            case 7:
                printf("Highest element in BST: %d\n", findMax(root));
                break;
            case 8:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while(choice != 8);

    return 0;
}
```

Write a C program to create a binary search tree using non-recursive function and display that. Write a C program to take user name as input and display the sorted sequence of characters using BST. Write a C program to sort a given set of integers using BST.

```
#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 = NULL;
   newNode->right = NULL;
   return newNode;
}

void insert(struct Node** root, int data) {
   struct Node* newNode = createNode(data);

   if (*root == NULL) {
      *root = newNode;
      return;
   }

   struct Node* current = *root;
   struct Node* parent = NULL;

   while (1) {
      parent = current;

      if (data <= current->data) {
         current = current->left;
         if (current == NULL) {
            parent->left = newNode;
            return;
         }
      } else {
         current = current->right;
         if (current == NULL) {
            parent->right = newNode;
```

```
            return;
        }
    }
  }
}

void inorder(struct Node* root) {
  if (root == NULL) return;

  inorder(root->left);
  printf("%d ", root->data);
  inorder(root->right);
}

int main() {
  struct Node* root = NULL;
  int arr[] = {50, 30, 70, 20, 40, 60, 80};
  int size = sizeof(arr) / sizeof(arr[0]);

  for (int i = 0; i < size; i++) {
    insert(&root, arr[i]);
  }

  printf("Binary Search Tree (In-order): ");
  inorder(root);

  return 0;
}
```

2. **Take user name as input and display the sorted sequence of characters using BST:**

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

struct Node {
  char data;
  struct Node* left;
  struct Node* right;
};

struct Node* createNode(char data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->left = NULL;
  newNode->right = NULL;
```

```
    return newNode;
}

void insert(struct Node** root, char data) {
    struct Node* newNode = createNode(data);

    if (*root == NULL) {
        *root = newNode;
        return;
    }

    struct Node* current = *root;
    struct Node* parent = NULL;

    while (1) {
        parent = current;

        if (data <= current->data) {
            current = current->left;
            if (current == NULL) {
                parent->left = newNode;
                return;
            }
        } else {
            current = current->right;
            if (current == NULL) {
                parent->right = newNode;
                return;
            }
        }
    }
}

void inorder(struct Node* root) {
    if (root == NULL) return;

    inorder(root->left);
    printf("%c ", root->data);
    inorder(root->right);
}

int main() {
    char name[100];
    printf("Enter your name: ");
    scanf("%s", name);

    struct Node* root = NULL;
    int len = strlen(name);
```

```
    for (int i = 0; i < len; i++) {
        insert(&root, name[i]);
    }

    printf("Sorted sequence of characters in your name: ");
    inorder(root);

    return 0;
}
```

3. **Sort a given set of integers using BST:**

```
#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 = NULL;
    newNode->right = NULL;
    return newNode;
}

void insert(struct Node** root, int data) {
    struct Node* newNode = createNode(data);

    if (*root == NULL) {
        *root = newNode;
        return;
    }

    struct Node* current = *root;
    struct Node* parent = NULL;

    while (1) {
        parent = current;

        if (data <= current->data) {
            current = current->left;
            if (current == NULL) {
```

```
            parent->left = newNode;
            return;
          }
        } else {
          current = current->right;
          if (current == NULL) {
            parent->right = newNode;
            return;
          }
        }
    }
}

void inorder(struct Node* root) {
    if (root == NULL) return;

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

int main() {
    int arr[] = {50, 30, 70, 20, 40, 60, 80};
    int size = sizeof(arr) / sizeof(arr[0]);
    struct Node* root = NULL;

    for (int i = 0; i < size; i++) {
        insert(&root, arr[i]);
    }

    printf("Sorted sequence of integers: ");
    inorder(root);

    return 0;
}
```

ASSIGNMENT4 1. Write a C program to search an element recursively in a binary search tree. 2. Write a C program to delete a node having no child from a binary search tree. 3. Write a C program to delete a node having one child from a binary search tree. 4. Write a C program to delete a node having two children from a binary search tree. 5. Write a C program to delete a node from a binary search tree. user input

1. **Search an element recursively in a binary search tree:**

```
#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 = NULL;
    newNode->right = NULL;
    return newNode;
}

struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key) {
        return root;
    }

    if (key < root->data) {
        return search(root->left, key);
    }

    return search(root->right, key);
}

int main() {
    struct Node* root = createNode(50);
    root->left = createNode(30);
    root->right = createNode(70);
    root->left->left = createNode(20);
    root->left->right = createNode(40);
    root->right->left = createNode(60);
    root->right->right = createNode(80);

    int key;
    printf("Enter element to search: ");
    scanf("%d", &key);

    if (search(root, key) != NULL) {
        printf("%d found in BST.\n", key);
    } else {
        printf("%d not found in BST.\n", key);
    }

    return 0;
```

```
}
```

2. **Delete a node having no child from a binary search tree:**

```
#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 = NULL;
   newNode->right = NULL;
   return newNode;
}

struct Node* insert(struct Node* root, int data) {
   if (root == NULL) {
      root = createNode(data);
   } else {
      if (data <= root->data) {
         root->left = insert(root->left, data);
      } else {
         root->right = insert(root->right, data);
      }
   }
   return root;
}

struct Node* deleteNode(struct Node* root, int key) {
   if (root == NULL) return root;

   if (key < root->data) {
      root->left = deleteNode(root->left, key);
   } else if (key > root->data) {
      root->right = deleteNode(root->right, key);
   } else {
      if (root->left == NULL) {
         struct Node* temp = root->right;
         free(root);
         return temp;
      } else if (root->right == NULL) {
```

```
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        struct Node* temp = root->right;
        while (temp->left != NULL) {
            temp = temp->left;
        }

        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}

void inorder(struct Node* root) {
    if (root == NULL) return;

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

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

    printf("Binary Search Tree (In-order) before deletion: ");
    inorder(root);
    printf("\n");

    int key;
    printf("Enter element to delete: ");
    scanf("%d", &key);

    root = deleteNode(root, key);

    printf("Binary Search Tree (In-order) after deletion: ");
    inorder(root);
    printf("\n");
```

```
    return 0;
}
```

3. **Delete a node having one child from a binary search tree:**

```
#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 = NULL;
    newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        root = createNode(data);
    } else {
        if (data <= root->data) {
            root->left = insert(root->left, data);
        } else {
            root->right = insert(root->right, data);
        }
    }
    return root;
}

struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) return root;

    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } else {
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
```

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

        struct Node* temp = root->right;
        while (temp->left != NULL) {
            temp = temp->left;
        }

        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}

void inorder(struct Node* root) {
    if (root == NULL) return;

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

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

    printf("Binary Search Tree (In-order) before deletion: ");
    inorder(root);
    printf("\n");

    int key;
    printf("Enter element to delete: ");
    scanf("%d", &key);

    root = deleteNode(root, key);

    printf("Binary Search Tree (In-order) after deletion: ");
```

```c
    inorder(root);
    printf("\n");

    return 0;
}
```

1. Write a C program to store the Graph using Adjacency Matrix & display that. 3. Write a C program to count number of vertices and edges present in a graph. 4. Write a C program to detect a cycle in a graph. 5. Write a C program to identify number of odd degree vertices and number of even degree vertices in a graph. 6. Write a C program to check whether a given graph is complete or not. menu driven user input

```c
#include <stdio.h>

#include <stdlib.h>


#define MAX_VERTICES 10


int adjMatrix[MAX_VERTICES][MAX_VERTICES];

int numVertices = 0;


void initializeGraph() {

  numVertices = 0;

  for (int i = 0; i < MAX_VERTICES; i++) {

    for (int j = 0; j < MAX_VERTICES; j++) {

      adjMatrix[i][j] = 0;

    }

  }

}


void addEdge(int src, int dest) {

  if (src >= 0 && src < MAX_VERTICES && dest >= 0 && dest < MAX_VERTICES) {

    adjMatrix[src][dest] = 1;

    adjMatrix[dest][src] = 1; // Assuming an undirected graph

    if (src > numVertices) numVertices = src;
```

```
        if (dest > numVertices) numVertices = dest;
    } else {
        printf("Invalid edge!\n");
    }
}

void displayGraph() {
    printf("Adjacency Matrix Representation of the Graph:\n");
    printf(" ");
    for (int i = 0; i <= numVertices; i++) {
        printf(" %d", i);
    }
    printf("\n");
    for (int i = 0; i <= numVertices; i++) {
        printf("%d ", i);
        for (int j = 0; j <= numVertices; j++) {
            printf("%2d", adjMatrix[i][j]);
        }
        printf("\n");
    }
}

int countVertices() {
    return numVertices + 1;
}

int countEdges() {
    int count = 0;
    for (int i = 0; i <= numVertices; i++) {
        for (int j = 0; j <= numVertices; j++) {
            if (adjMatrix[i][j] == 1) {
```

```
            count++;
        }
      }
    }
    return count / 2; // Since the graph is undirected, divide by 2
}

int hasCycleUtil(int v, int visited[], int parent) {
    visited[v] = 1;
    for (int i = 0; i <= numVertices; i++) {
        if (adjMatrix[v][i]) {
            if (!visited[i]) {
                if (hasCycleUtil(i, visited, v)) {
                    return 1;
                }
            } else if (i != parent) {
                return 1;
            }
        }
    }
    return 0;
}

int hasCycle() {
    int visited[MAX_VERTICES] = {0};
    for (int i = 0; i <= numVertices; i++) {
        if (!visited[i]) {
            if (hasCycleUtil(i, visited, -1)) {
                return 1;
            }
        }
```

```
    }
    return 0;
}

int countOddDegreeVertices() {
    int degree[MAX_VERTICES] = {0};
    for (int i = 0; i <= numVertices; i++) {
        for (int j = 0; j <= numVertices; j++) {
            if (adjMatrix[i][j]) {
                degree[i]++;
            }
        }
    }
    int oddCount = 0;
    for (int i = 0; i <= numVertices; i++) {
        if (degree[i] % 2 != 0) {
            oddCount++;
        }
    }
    return oddCount;
}

int countEvenDegreeVertices() {
    int degree[MAX_VERTICES] = {0};
    for (int i = 0; i <= numVertices; i++) {
        for (int j = 0; j <= numVertices; j++) {
            if (adjMatrix[i][j]) {
                degree[i]++;
            }
        }
    }
```

```
    int evenCount = 0;
    for (int i = 0; i <= numVertices; i++) {
        if (degree[i] % 2 == 0) {
            evenCount++;
        }
    }
    return evenCount;
}

int isCompleteGraph() {
    for (int i = 0; i <= numVertices; i++) {
        for (int j = 0; j <= numVertices; j++) {
            if (i != j && !adjMatrix[i][j]) {
                return 0;
            }
        }
    }
    return 1;
}

int main() {
    int choice, src, dest;
    initializeGraph();

    do {
        printf("\n1. Add an edge to the graph\n");
        printf("2. Display the graph\n");
        printf("3. Count number of vertices\n");
        printf("4. Count number of edges\n");
        printf("5. Detect cycle in the graph\n");
        printf("6. Identify number of odd degree vertices\n");
```

```c
printf("7. Identify number of even degree vertices\n");
printf("8. Check whether the graph is complete\n");
printf("9. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch(choice) {
  case 1:
    printf("Enter source and destination vertices of the edge: ");
    scanf("%d %d", &src, &dest);
    addEdge(src, dest);
    break;
  case 2:
    displayGraph();
    break;
  case 3:
    printf("Number of vertices in the graph: %d\n", countVertices());
    break;
  case 4:
    printf("Number of edges in the graph: %d\n", countEdges());
    break;
  case 5:
    if (hasCycle()) {
      printf("Graph has a cycle.\n");
    } else {
      printf("Graph does not have a cycle.\n");
    }
    break;
  case 6:
    printf("Number of odd degree vertices in the graph: %d\n", countOddDegreeVertices());
    break;
```

```
        case 7:
          printf("Number of even degree vertices in the graph: %d\n", countEvenDegreeVertices());
          break;
        case 8:
          if (isCompleteGraph()) {
            printf("Graph is complete.\n");
          } else {
            printf("Graph is not complete.\n");
          }
          break;
        case 9:
          printf("Exiting program...\n");
          break;
        default:
          printf("Invalid choice! Please enter a valid option.\n");
      }
  } while(choice != 9);

  return 0;
}
```

2. Write a C program to store the f Graph using Adjacency List & display that. user input

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

// Structure to represent a node in the adjacency list
struct Node {
  int vertex;
  struct Node* next;
};
```

```
// Structure to represent the adjacency list
struct Graph {
    int numVertices;
    struct Node** adjList;
};

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

// Function to create a graph with 'V' vertices
struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = V;

    // Create an array of adjacency lists. Size of the array will be V
    graph->adjList = (struct Node**)malloc(V * sizeof(struct Node*));

    // Initialize each adjacency list as empty by making each head NULL
    for (int i = 0; i < V; i++) {
        graph->adjList[i] = NULL;
    }

    return graph;
}
```

```
// Function to add an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest) {
   // Add an edge from src to dest
   struct Node* newNode = createNode(dest);
   newNode->next = graph->adjList[src];
   graph->adjList[src] = newNode;

   // Since the graph is undirected, add an edge from dest to src as well
   newNode = createNode(src);
   newNode->next = graph->adjList[dest];
   graph->adjList[dest] = newNode;
}

// Function to print the adjacency list representation of the graph
void printGraph(struct Graph* graph) {
   for (int v = 0; v < graph->numVertices; v++) {
      struct Node* temp = graph->adjList[v];
      printf("\nAdjacency list of vertex %d\n head ", v);
      while (temp) {
         printf("-> %d", temp->vertex);
         temp = temp->next;
      }
      printf("\n");
   }
}

int main() {
   int numVertices, numEdges, src, dest;

   printf("Enter the number of vertices in the graph: ");
   scanf("%d", &numVertices);
```

```
    struct Graph* graph = createGraph(numVertices);

    printf("Enter the number of edges in the graph: ");
    scanf("%d", &numEdges);

    for (int i = 0; i < numEdges; i++) {
        printf("Enter edge %d (source destination): ", i + 1);
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }

    printf("\nAdjacency List Representation of the Graph:\n");
    printGraph(graph);

    return 0;
}
```

Write a C program to traverse the following graph using Depth First Search (DFS) algorithm.
Write a C program to traverse the following graph using Breadth First Search (BFS) algorithm.
menu driven user input simple easy code

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

#define MAX_VERTICES 10

struct Node {
    int vertex;
    struct Node* next;
```

```
};

struct Graph {
   int numVertices;
   struct Node** adjList;
   int* visited;
};

struct Node* createNode(int v) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}

struct Graph* createGraph(int vertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->numVertices = vertices;

   graph->adjList = (struct Node**)malloc(vertices * sizeof(struct Node*));
   graph->visited = (int*)malloc(vertices * sizeof(int));

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

   return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
```

```
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

void dfs(struct Graph* graph, int vertex) {
    struct Node* adjList = graph->adjList[vertex];
    struct Node* temp = adjList;

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

    while (temp != NULL) {
        int connectedVertex = temp->vertex;
        if (graph->visited[connectedVertex] == 0) {
            dfs(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

void bfs(struct Graph* graph, int startVertex) {
    struct Node* queue[MAX_VERTICES];
    int front = 0, rear = 0;
    queue[rear++] = createNode(startVertex);
    graph->visited[startVertex] = 1;
```

```
    while (front < rear) {
        struct Node* currentNode = queue[front++];
        printf("Visited %d\n", currentNode->vertex);

        struct Node* temp = graph->adjList[currentNode->vertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (graph->visited[adjVertex] == 0) {
                queue[rear++] = createNode(adjVertex);
                graph->visited[adjVertex] = 1;
            }
            temp = temp->next;
        }
    }
}

int main() {
    int choice, vertices, edges, src, dest;

    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &vertices);
    struct Graph* graph = createGraph(vertices);

    printf("Enter the number of edges in the graph: ");
    scanf("%d", &edges);

    for (int i = 0; i < edges; i++) {
        printf("Enter edge %d (source destination): ", i + 1);
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }
```

```c
do {
    printf("\nGraph Traversal Menu\n");
    printf("1. Depth First Search (DFS)\n");
    printf("2. Breadth First Search (BFS)\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            for (int i = 0; i < vertices; i++) {
                if (graph->visited[i] == 0) {
                    dfs(graph, i);
                }
            }
            break;
        case 2:
            for (int i = 0; i < vertices; i++) {
                if (graph->visited[i] == 0) {
                    bfs(graph, i);
                }
            }
            break;
        case 3:
            printf("Exiting program...\n");
            break;
        default:
            printf("Invalid choice! Please enter a valid option.\n");
    }
} while (choice != 3);
```

```
    return 0;
}
```