bst array implementation

```
#include <stdio.h>
#define TREE_SIZE 10  // Maximum size of the binary search tree

// Structure to represent a node in the binary search tree
typedef struct TreeNode {
    int value;    // Data value of the node
    int isValid;  // Flag to indicate if the node is valid (occupied)
} TreeNode;

// Function to insert a value into the binary search tree
void insert(TreeNode *binaryTree, int value, int index) {
    // Check if the current position in the tree is vacant and within bounds
    if (binaryTree[index].isValid == 0 && index < TREE_SIZE) {
        binaryTree[index].value = value;  // Assign the value to the node
        binaryTree[index].isValid = 1;   // Mark the node as valid
    }
    // Recur to the left subtree if the value is less than or equal to the
current node's value
    else if (value <= binaryTree[index].value && index < TREE_SIZE) {
        index = 2 * index + 1;  // Calculate the left child index
        insert(binaryTree, value, index);
    }
    // Recur to the right subtree if the value is greater than the current
node's value
    else if (value > binaryTree[index].value && index < TREE_SIZE) {
        index = 2 * index + 2;  // Calculate the right child index
        insert(binaryTree, value, index);
    } else {
        printf("Insertion not possible\n");  // Handle insertion failures
    }
}

// Function for in-order traversal of the binary search tree
void inorderTraversal(TreeNode *binaryTree, int index) {
    if (binaryTree[index].isValid && index < TREE_SIZE) {
        inorderTraversal(binaryTree, 2 * index + 1);  // Traverse left
subtree
        printf("%d ", binaryTree[index].value);      // Print current node
value
        inorderTraversal(binaryTree, 2 * index + 2);  // Traverse right
subtree
    }
}

// Function for post-order traversal of the binary search tree
void postorderTraversal(TreeNode *binaryTree, int index) {
    if (binaryTree[index].isValid && index < TREE_SIZE) {
        postorderTraversal(binaryTree, 2 * index + 1);  // Traverse left
subtree
        postorderTraversal(binaryTree, 2 * index + 2);  // Traverse right
subtree
        printf("%d ", binaryTree[index].value);        // Print current node
value
    }
}

// Function to print the binary search tree as an array representation
void printTree(TreeNode *binaryTree) {
    int i;
    printf("Index:\t");
    for (i = 0; i < TREE_SIZE; i++) {
        printf("%d\t", i);  // Print the array indices
    }
    printf("\nValue:\t");
    for (i = 0; i < TREE_SIZE; i++) {
        if (binaryTree[i].isValid) {
            printf("%d\t", binaryTree[i].value);  // Print node values
        } else {
            printf("-\t");  // Print dash for vacant nodes
        }
    }
    printf("\nValid:\t");
    for (i = 0; i < TREE_SIZE; i++) {
        printf("%d\t", binaryTree[i].isValid);  // Print validity flags
    }
    printf("\n");
```

```
}

// Main function
int main() {
    int inputValue, continueFlag = 0;
    TreeNode binaryTree[TREE_SIZE] = {};  // Initialize the tree with all
flags set to 0

    do {
        printf("Enter a value to insert: ");
        scanf("%d", &inputValue);
        insert(binaryTree, inputValue, 0);  // Insert the value into the
tree
        printf("Press 1 to continue inserting, or 0 to stop: ");
        scanf("%d", &continueFlag);
    } while (continueFlag);

    printf("\nIn-Order Traversal: ");
    inorderTraversal(binaryTree, 0);
    printf("\n");

    printf("\nPost-Order Traversal: ");
    postorderTraversal(binaryTree, 0);
    printf("\n");

    printf("\nBinary Search Tree Array Representation:\n");
    printTree(binaryTree);

    return 0;
}
```

binary search tree implementation

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

// Structure representing a single node in the Binary Search Tree (BST)
struct TreeNode {
    int data;                   // Value stored in the node
    struct TreeNode *left;      // Pointer to the left child
    struct TreeNode *right;     // Pointer to the right child
};
typedef struct TreeNode TreeNode;

// Structure representing the Binary Tree
struct BinaryTree {
    TreeNode *root;             // Pointer to the root node of the tree
};
typedef struct BinaryTree BinaryTree;

// Function prototypes
void initializeTree(BinaryTree*);
void insertNode(BinaryTree*, int);
TreeNode* recursiveInsertNode(TreeNode*, int);
void recursiveInsert(BinaryTree*, int);

void inorderTraversal(BinaryTree*);
void preorderTraversal(BinaryTree*);
void postorderTraversal(BinaryTree*);
void inorder(TreeNode*);
void preorder(TreeNode*);
void postorder(TreeNode*);

int countNodes(BinaryTree*);
int count(TreeNode*);
int countLeafNodes(BinaryTree*);
int countLeaf(TreeNode*);
int calculateHeightTree(BinaryTree*);
int calculateHeight(TreeNode*);

int iterativeSearch(BinaryTree*, int);
int recursiveSearchTree(BinaryTree*, int);
int recursiveSearch(TreeNode*, int);

void deleteNode(BinaryTree*, int);
int findMinimum(BinaryTree*);
int findMaximum(BinaryTree*);

// Main function for menu-driven implementation
```

```
int main() {
    BinaryTree tree;
    int choice, element, result, minValue, maxValue;

    initializeTree(&tree);  // Initialize the tree

    while (1) {
        printf("\n1. Insert Element");
        printf("\n2. Preorder Traversal");
        printf("\n3. Postorder Traversal");
        printf("\n4. Inorder Traversal");
        printf("\n5. Count Nodes");
        printf("\n6. Count Leaf Nodes");
        printf("\n7. Calculate Height");
        printf("\n8. Delete Node");
        printf("\n9. Insert Recursively");
        printf("\n10. Search Iteratively");
        printf("\n11. Search Recursively");
        printf("\n12. Find Smallest Element");
        printf("\n13. Find Largest Element");
        printf("\n14. Exit");

        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to insert: ");
                scanf("%d", &element);
                insertNode(&tree, element);
                break;

            case 2:
                preorderTraversal(&tree);
                break;

            case 3:
                postorderTraversal(&tree);
                break;
            case 4:
                inorderTraversal(&tree);
                break;

            case 5:
                result = countNodes(&tree);
                printf("\nNumber of nodes in the tree: %d", result);
                break;

            case 6:
                result = countLeafNodes(&tree);
                printf("\nNumber of leaf nodes in the tree: %d", result);
                break;

            case 7:
                result = calculateHeightTree(&tree);
                printf("\nHeight of the tree: %d", result);
                break;

            case 8:
                printf("Enter the element to delete: ");
                scanf("%d", &element);
                deleteNode(&tree, element);
                break;

            case 9:
                printf("Enter the element to insert recursively: ");
                scanf("%d", &element);
                recursiveInsert(&tree, element);
                break;

            case 10:
                printf("Enter the element to search: ");
                scanf("%d", &element);
                result = iterativeSearch(&tree, element);
                printf(result ? "Element found.\n" : "Element not
found.\n");
                break;

            case 11:
                printf("Enter the element to search recursively: ");
                scanf("%d", &element);
                result = recursiveSearchTree(&tree, element);
                printf(result ? "Element found.\n" : "Element not
found.\n");
                break;

            case 12:
                minValue = findMinimum(&tree);
                printf("\nSmallest element in the tree: %d", minValue);
                break;

            case 13:
                maxValue = findMaximum(&tree);
                printf("\nLargest element in the tree: %d", maxValue);
                break;

            case 14:
                exit(0);

            default:
                printf("\nInvalid choice. Please try again.");
        }
    }
    return 0;
}

// Initialize the binary tree
void initializeTree(BinaryTree *tree) {
    tree->root = NULL;
}

// Function to insert a node iteratively
void insertNode(BinaryTree *tree, int value) {
    TreeNode *current = tree->root, *parent = NULL, *newNode;

    // Create a new node
    newNode = (TreeNode*)malloc(sizeof(TreeNode));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
```

```c
    // If the tree is empty, set the root to the new node
    if (current == NULL) {
        tree->root = newNode;
        return;
    }

    // Find the appropriate position to insert the new node
    while (current != NULL) {
        parent = current;
        if (value < current->data)
            current = current->left;
        else
            current = current->right;
    }

    // Insert the node as a child of the parent
    if (value < parent->data)
        parent->left = newNode;
    else
        parent->right = newNode;
}

// Recursive insertion helper function
TreeNode* recursiveInsertNode(TreeNode *node, int value) {
    if (node == NULL) {
        TreeNode *newNode = (TreeNode*)malloc(sizeof(TreeNode));
        newNode->data = value;
        newNode->left = newNode->right = NULL;
        return newNode;
    }
    if (value < node->data)
        node->left = recursiveInsertNode(node->left, value);
    else
        node->right = recursiveInsertNode(node->right, value);
    return node;
}

// Recursive insertion
void recursiveInsert(BinaryTree *tree, int value) {
    tree->root = recursiveInsertNode(tree->root, value);
}

// Inorder traversal of the tree
void inorderTraversal(BinaryTree *tree) {
    printf("Inorder Traversal: ");
    inorder(tree->root);
    printf("\n");
}

void inorder(TreeNode *node) {
    if (node != NULL) {
        inorder(node->left);
        printf("%d ", node->data);
        inorder(node->right);
    }
}

// Preorder traversal of the tree
void preorderTraversal(BinaryTree *tree) {
    printf("Preorder Traversal: ");
    preorder(tree->root);
    printf("\n");
}

void preorder(TreeNode *node) {
    if (node != NULL) {
        printf("%d ", node->data);
        preorder(node->left);
        preorder(node->right);
    }
}

// Postorder traversal of the tree
void postorderTraversal(BinaryTree *tree) {
    printf("Postorder Traversal: ");
    postorder(tree->root);
    printf("\n");
}
```

```
void postorder(TreeNode *node) {
    if (node != NULL) {
        postorder(node->left);
        postorder(node->right);
        printf("%d ", node->data);
    }
}

// Function to count all nodes in the tree
int countNodes(BinaryTree *tree) {
    return count(tree->root);
}

int count(TreeNode *node) {
    if (node == NULL)
        return 0;
    return 1 + count(node->left) + count(node->right);
}

// Function to count leaf nodes in the tree
int countLeafNodes(BinaryTree *tree) {
    return countLeaf(tree->root);
}

int countLeaf(TreeNode *node) {
    if (node == NULL)
        return 0;
    if (node->left == NULL && node->right == NULL)
        return 1;
    return countLeaf(node->left) + countLeaf(node->right);
}

// Function to calculate the height of the tree
int calculateHeightTree(BinaryTree *tree) {
    return calculateHeight(tree->root);
}

int calculateHeight(TreeNode *node) {
    if (node == NULL)
        return -1; // Height of an empty tree is -1
    int leftHeight = calculateHeight(node->left);
    int rightHeight = calculateHeight(node->right);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}

// Iterative search in the tree
int iterativeSearch(BinaryTree *tree, int value) {
    TreeNode *current = tree->root;

    while (current != NULL) {
        if (value == current->data)
            return 1; // Element found
        else if (value < current->data)
            current = current->left;
        else
            current = current->right;
    }
    return 0; // Element not found
}

// Recursive search helper function
int recursiveSearch(TreeNode *node, int value) {
    if (node == NULL)
        return 0;
    if (value == node->data)
        return 1;
    if (value < node->data)
        return recursiveSearch(node->left, value);
    return recursiveSearch(node->right, value);
}

// Recursive search
int recursiveSearchTree(BinaryTree *tree, int value) {
    return recursiveSearch(tree->root, value);
}

// Function to delete a node from the tree
void deleteNode(BinaryTree *tree, int value) {
    TreeNode *parent = NULL, *current = tree->root, *successor,
*successorParent;
```

```c
    // Find the node to delete and its parent
    while (current != NULL && current->data != value) {
        parent = current;
        if (value < current->data)
            current = current->left;
        else
            current = current->right;
    }

    // If node not found, return
    if (current == NULL) {
        printf("Element not found in the tree.\n");
        return;
    }

    // Case 1: Node to be deleted has no children (leaf node)
    if (current->left == NULL && current->right == NULL) {
        if (current != tree->root) {
            if (parent->left == current)
                parent->left = NULL;
            else
                parent->right = NULL;
        } else {
            tree->root = NULL;
        }
        free(current);
    }

    // Case 2: Node to be deleted has two children
    else if (current->left != NULL && current->right != NULL) {
        successorParent = current;
        successor = current->right;

        // Find the in-order successor (leftmost child of the right subtree)
        while (successor->left != NULL) {
            successorParent = successor;
            successor = successor->left;
        }
        // Copy the successor's value to the current node
        current->data = successor->data;

        // Delete the successor
        if (successorParent->left == successor)
            successorParent->left = successor->right;
        else
            successorParent->right = successor->right;

        free(successor);
    }

    // Case 3: Node to be deleted has only one child
    else {
        TreeNode *child = (current->left != NULL) ? current->left : current-
>right;

        if (current != tree->root) {
            if (parent->left == current)
                parent->left = child;
            else
                parent->right = child;
        } else {
            tree->root = child;
        }
        free(current);
    }
}

// Find the smallest element in the tree
int findMinimum(BinaryTree *tree) {
    if (tree->root == NULL) {
        printf("Tree is empty.\n");
        return -1;
    }

    TreeNode *current = tree->root;
    while (current->left != NULL)
        current = current->left;
    return current->data;
```

```
}

// Find the largest element in the tree
int findMaximum(BinaryTree *tree) {
    if (tree->root == NULL) {
        printf("Tree is empty.\n");
        return -1;
    }

    TreeNode *current = tree->root;
    while (current->right != NULL)
        current = current->right;
    return current->data;
}
```

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

struct tnode {
  int data;
  struct tnode *left;
  struct tnode *right;
};
typedef struct tnode tnode_t;

struct tree {
  tnode_t *root;
};
typedef struct tree tree_t;

void init(tree_t *);
void inorder(tnode_t *);
void preorder(tnode_t *);
void postorder(tnode_t *);
void insert(tree_t *, int);
int count(tnode_t *);
int leafcount(tnode_t *);
int height(tnode_t *);
tnode_t *rinsert(tnode_t *, int);
```

```
void tdelete_t(tree_t *, int);
int search(tree_t *, int);
int rsearch(tnode_t *, int);
int minimum(tree_t *);
int maximum(tree_t *);

void main() {
  tree_t t;
  int ch, num, k, n;
  init(&t);

  while (1) {
    printf("\n1.Insert");
    printf("\n2.Preorder");
    printf("\n3.Postorder");
    printf("\n4.Inorder");
    printf("\n5.No. of nodes");
    printf("\n6.No. of Leaf nodes");
    printf("\n7.Height of a tree");
    printf("\n8.delete a node");
    printf("\n9..recursively insert..");
    printf("\n10..search..");
    printf("\n11..search using recursion..");
    printf("\n13.find smallest");
    printf("\n14.find largest");
    printf("\n15..exit");
    scanf("%d", &ch);

    switch (ch) {
      case 1:
        printf("Enter the element");
        scanf("%d", &num);
        insert(&t, num);
        break;
      case 2:
        preorder(t.root);
        break;
      case 3:
        postorder(t.root);
        break;
      case 4:
        inorder(t.root);
        break;
      case 5:
        k = count(t.root);
        printf("\nthe number of nodes=%d", k);
        break;
      case 6:
        k = leafcount(t.root);
        printf("\nthe number of nodes=%d", k);
        break;
      case 7:
        k = height(t.root);
        printf("\nthe height of tree=%d", k);
        break;
      case 8:
        printf("Enter the node to be deleted\n");
        scanf("%d", &num);
        tdelete_t(&t, num);
        break;
      case 9:
        printf("Enter the element");
        scanf("%d", &num);
        t.root = rinsert(t.root, num);
        break;
      case 10:
        printf("Enter the element");
        scanf("%d", &num);
        if (search(&t, num))
          printf("Key found\n");
        else
          printf("Not found..\n");
        break;
      case 11:
        printf("Enter the element");
        scanf("%d", &num);
        if (rsearch(t.root, num))
          printf("Key found\n");
        else
          printf("Not found..\n");
        break;
      case 13:
        n = minimum(&t);
        printf("\nThe smallest element = %d\n", n);
        break;
      case 14:
        n = maximum(&t);
        printf("\nthe largest element = %d\n", n);
        break;
      case 15:
        exit(0);
    }
  }
}

void init(tree_t *t) {
  t->root = NULL;
}

void insert(tree_t *t, int x) {
  tnode_t *curr, *prev, *newnode;
  curr = t->root;
  prev = NULL;

  newnode = (tnode_t *)malloc(sizeof(tnode_t));
  newnode->data = x;
  newnode->left = newnode->right = NULL;

  if (curr == NULL) {
    t->root = newnode;
  } else {
    while (curr != NULL) {
      prev = curr;
      if (x > curr->data)
        curr = curr->right;
      else
        curr = curr->left;
    }
    if (x > prev->data)
      prev->right = newnode;
    else
      prev->left = newnode;
  }
}

void preorder(tnode_t *root) {
  if (root != NULL) {
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
  }
}

void inorder(tnode_t *root) {
  if (root != NULL) {
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
  }
}

void postorder(tnode_t *root) {
  if (root != NULL) {
    postorder(root->left);
    postorder(root->right);
    printf("%d  ", root->data);
  }
}

int minimum(tree_t *t) {
  tnode_t *root = t->root;
  if (root == NULL) {
    printf("Empty tree");
    return -1;
  } else {
    while (root->left != NULL)
      root = root->left;
    return root->data;
  }
}
```

```
int maximum(tree_t *t) {
  tnode_t *root = t->root;
  if (root == NULL) {
    printf("Empty tree");
    return -1;
  } else {
    while (root->right != NULL)
      root = root->right;
    return root->data;
  }
}

int height(tnode_t *root) {
  if (root == NULL)
    return -1;
  int l = height(root->left);
  int r = height(root->right);
  return (l > r ? l : r) + 1;
}

int leafcount(tnode_t *root) {
  if (root == NULL)
    return 0;
  if (root->left == NULL && root->right == NULL)
    return 1;
  return leafcount(root->left) + leafcount(root->right);
}

int count(tnode_t *root) {
  if (root == NULL)
    return 0;
  return count(root->left) + count(root->right) + 1;
}

tnode_t *rinsert(tnode_t *root, int key) {
  if (root == NULL) {
    tnode_t *newnode = (tnode_t *)malloc(sizeof(tnode_t));
    newnode->data = key;
    newnode->left = newnode->right = NULL;
    return newnode;
  }
  if (key < root->data)
    root->left = rinsert(root->left, key);
  else
    root->right = rinsert(root->right, key);
  return root;
}

int search(tree_t *t, int key) {
  tnode_t *curr = t->root;
  while (curr != NULL) {
    if (key == curr->data)
      return 1;
    if (key < curr->data)
      curr = curr->left;
    else
      curr = curr->right;
  }
  return 0;
}

int rsearch(tnode_t *root, int key) {
  if (root == NULL)
    return 0;
  if (key == root->data)
    return 1;
  if (key < root->data)
    return rsearch(root->left, key);
  return rsearch(root->right, key);
}

void tdelete_t(tree_t *t, int key) {
  tnode_t *curr = t->root, *prev = NULL, *q, *temp;

  while (curr != NULL && curr->data != key) {
    prev = curr;
    if (key < curr->data)
      curr = curr->left;
    else
```

```
      curr = curr->right;
    }

    if (curr == NULL) {
      printf("Key not found");
      return;
    }

    if (curr->left == NULL || curr->right == NULL) {
      q = curr->left ? curr->left : curr->right;
      if (prev == NULL)
        t->root = q;
      else if (curr == prev->left)
        prev->left = q;
      else
        prev->right = q;
      free(curr);
    } else {
      prev = NULL;
      temp = curr->right;
      while (temp->left != NULL) {
        prev = temp;
        temp = temp->left;
      }
      curr->data = temp->data;
      if (prev != NULL)
        prev->left = temp->right;
      else
        curr->right = temp->right;
      free(temp);
    }
  }
```

height of a binary tree (array)

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

// Function to calculate the height of a binary tree stored in an array
int calculateHeight(int arr[], int size) {
    if (size == 0)
        return 0; // An empty tree has height 0

    int height = 0;

    // Calculate the height of the tree
    while (pow(2, height) - 1 < size) {
        height++;
    }

    return height;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50, 60}; // Example tree
    int size = sizeof(arr) / sizeof(arr[0]);

    int height = calculateHeight(arr, size);
    printf("Height of the binary tree: %d\n", height);

    return 0;
}
```

Sum of Elements in a Binary Tree (Dynamic Allocation)

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

// Define a node in the binary tree
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};

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

// Recursive function to calculate the sum of elements in the binary tree
int calculateSum(struct TreeNode* root) {
    if (root == NULL)
        return 0;

    // Sum of current node and its subtrees
    return root->data + calculateSum(root->left) + calculateSum(root-
>right);
}

int main() {
    // Create a sample binary tree
    struct TreeNode* root = createNode(10);
    root->left = createNode(20);
    root->right = createNode(30);
    root->left->left = createNode(40);
    root->left->right = createNode(50);
    root->right->left = createNode(60);

    int sum = calculateSum(root);
    printf("Sum of all elements in the binary tree: %d\n", sum);

    return 0;
}
```

iterative traversal

```
#include<stdio.h>
#include<stdlib.h> // Added for malloc

#define MAX 100 // Increased stack size
```

```c
typedef struct node {
    int data;
    struct node *left;
    struct node *right;
} NODE;

typedef struct stack {
    NODE *stack[MAX];
    int top;
} STACK;

void push(NODE *t, STACK *s) {
    if (s->top == MAX - 1) {
        printf("Stack Overflow\n");
    } else {
        s->stack[++(s->top)] = t;
    }
}

NODE* pop(STACK *s) {
    if (s->top == -1) {
        printf("Stack Underflow\n");
        return NULL;
    } else {
        return s->stack[(s->top)--];
    }
}

NODE* createNode(int data) {
    NODE *nn = (NODE*)malloc(sizeof(NODE));
    nn->data = data;
    nn->left = NULL;
    nn->right = NULL;
    return nn;
}

int isEmpty(NODE *t) {
    return (t == NULL);
}
```

```
NODE* insertNode(NODE *root, int data) {
    if (isEmpty(root)) {
        root = createNode(data);
    } else if (data <= root->data) {
        root->left = insertNode(root->left, data);
    } else {
        root->right = insertNode(root->right, data);
    }
    return root;
}

void inorder(NODE *t) {
    if (isEmpty(t)) {
        printf("Tree is empty\n");
        return;
    }

    STACK *s = (STACK*)malloc(sizeof(STACK));
    s->top = -1;
    NODE *temp = NULL;

    do {
        while (t != NULL) {
            push(t, s);
            t = t->left;
        }

        temp = pop(s);
        if (temp != NULL) {
            printf("%d ", temp->data);
            t = temp->right;
        }
    } while (t != NULL || s->top != -1);

    free(s); // Free the allocated stack memory
}

void preorder(NODE *t) {
    if (isEmpty(t)) {
        printf("Tree is empty\n");
        return;
    }

    STACK *s = (STACK*)malloc(sizeof(STACK));
    s->top = -1;
    push(t, s);

    while (s->top != -1) {
        NODE *temp = pop(s);
        printf("%d ", temp->data);

        if (temp->right != NULL) {
            push(temp->right, s);
        }
        if (temp->left != NULL) {
            push(temp->left, s);
        }
    }

    free(s); // Free the allocated stack memory
}

void postorder(NODE *t) {
    if (isEmpty(t)) {
        printf("Tree is empty\n");
        return;
    }

    STACK *s1 = (STACK*)malloc(sizeof(STACK));
    STACK *s2 = (STACK*)malloc(sizeof(STACK));
    s1->top = -1;
    s2->top = -1;
    NODE *temp = NULL;

    push(t, s1);
    while (s1->top != -1) {
        temp = pop(s1);
        push(temp, s2);
        if (temp->left != NULL) {
            push(temp->left, s1);
        }
        if (temp->right != NULL) {
            push(temp->right, s1);
        }
    }

    while (s2->top != -1) {
        temp = pop(s2);
        printf("%d ", temp->data);
    }

    free(s1); // Free the allocated stack memory
    free(s2);
}

int main() {
    int n, d ,i;
    NODE *root = NULL;

    printf("\nEnter the number of nodes: ");
    scanf("%d", &n);

    printf("\nEnter the data: ");
    for ( i = 0; i < n; i++) {
        scanf("%d", &d);
        root = insertNode(root, d);
    }

    printf("\nPreorder traversal: ");
    preorder(root);
    printf("\nInorder traversal: ");
    inorder(root);
    printf("\nPostorder traversal: ");
    postorder(root);

    return 0;
}
```

```
iterativePostorder(root)
s1 = emptyStack ; s2 = emptyStack ; push(s1, root)
while(!isEmpty(s1)) {
        current = pop(s1)
        push(s2,current)
        if(current->left !=NULL)
                push(s1, current->left)
        if(current->right !=NULL)
                push(s1, current->right)
}
while(!isEmpty(s2)) {   //Print all the elements of stack2
        current = pop(s2)
        print current->info
}
```

Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
        while(current != null)
        {       /* Travel down left branches as far as possible
                saving pointers to nodes passed in the stack*/
                push(s, current)
                current = current->left
        } //At this point, the left subtree is empty
        poppedNode = pop(s)
        print poppedNode ->info        //visit the node
        current = poppedNode ->right //traverse right subtree
} while(!isEmpty(s) or current != null)
```

```
iterativePreorder(root)
current=root
if (current == null)
   return
s = emptyStack
push(s, current)
while(!isEmpty(s)) {
          current = pop(s)
          print current->info
          //right child is pushed first so that left is processed first
          if(current->right !=NULL)
                    push(s, current->right)
          if(current->left !=NULL)
                    push(s, current->left)
}
```

expression trees

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

#define STACK_MAX 10  // Maximum stack size for storing expression tree nodes

// Define structure for a node in the expression tree
typedef struct treeNode {
    int nodeType;  // Indicates whether the node is an operand (1) or an operator (2)
    union {  // Union to store either an operand or an operator
        char operator;   // Stores the operator (+, -, *, /)
        float operand;   // Stores the operand as a floating-point number
    } data;
    struct treeNode *leftChild;   // Pointer to the left child node
    struct treeNode *rightChild;  // Pointer to the right child node
} ExpressionNode;

static int stackTop = -1;  // Stack pointer initialized to -1, indicating an
empty stack

// Function to push a node onto the stack
// Parameters:
// - node: pointer to the node to be pushed onto the stack
// - stack: array of ExpressionNode pointers used as the stack
void push(ExpressionNode* node, ExpressionNode* stack[]) {
    if (stackTop < STACK_MAX - 1) {  // Ensure we don't exceed the stack
capacity
        stack[++stackTop] = node;  // Increment stack pointer and add node
to stack
    } else {
        printf("Stack is full\n");  // Print error if stack is full
    }
}

// Function to pop a node from the stack
// Parameters:
// - stack: array of ExpressionNode pointers used as the stack
// Returns: pointer to the popped node, or NULL if stack is empty
ExpressionNode* pop(ExpressionNode* stack[]) {
    if (stackTop == -1) {  // Check if stack is empty
        printf("Stack is empty\n");  // Print error if stack is empty
        return NULL;  // Return NULL if stack is empty
    }
    return stack[stackTop--];  // Return the top node and decrement the
stack pointer
}

// Function to construct an expression tree from a postfix expression
// Parameters:
// - expression: character array containing the postfix expression
// Returns: pointer to the root of the constructed expression tree, or NULL
if invalid
ExpressionNode* constructTree(char expression[]) {
    ExpressionNode *node, *stack[STACK_MAX];  // Array of pointers for the
```

```
stack
    int index = 0;  // Index to traverse the expression

    while (expression[index]) {  // Traverse until end of the expression
        if (isdigit(expression[index])) {  // Check if the character is a
digit (operand)
            node = (ExpressionNode*)malloc(sizeof(ExpressionNode));  //
Allocate memory for new node
            node->nodeType = 1;  // Set nodeType to 1 for operand
            node->data.operand = expression[index] - '0';  // Convert
character to float and store as operand
            node->leftChild = NULL;  // Operand node has no children
            node->rightChild = NULL;
            push(node, stack);  // Push the operand node onto the stack
        } else {  // Operator case
            node = (ExpressionNode*)malloc(sizeof(ExpressionNode));  //
Allocate memory for new node
            node->nodeType = 2;  // Set nodeType to 2 for operator
            node->data.operator = expression[index];  // Store operator in
data union
            // Pop two nodes for right and left children
            node->rightChild = pop(stack);
            node->leftChild = pop(stack);
            if (node->rightChild == NULL || node->leftChild == NULL) {  //
Check for invalid expression
                printf("Invalid expression\n");  // Print error if
expression is invalid
                free(node);  // Free allocated memory
                return NULL;  // Return NULL for invalid expression
            }
            push(node, stack);  // Push the operator node onto the stack
        }
        index++;  // Move to the next character in expression
    }
    return pop(stack);  // Final pop to get the root of the constructed
expression tree
}

// Function to perform in-order traversal and print the expression tree
// Parameters:
```

```
// - tree: pointer to the root of the expression tree
void inOrderTraversal(ExpressionNode *tree) {
    if (tree != NULL) {  // Check if tree node is not null
        inOrderTraversal(tree->leftChild);  // Traverse left child first
        if (tree->nodeType == 1)  // If node is an operand
            printf("%g ", tree->data.operand);  // Print operand value
        else  // If node is an operator
            printf("%c ", tree->data.operator);  // Print operator symbol
        inOrderTraversal(tree->rightChild);  // Traverse right child
    }
}

// Function to calculate the result of an operation
// Parameters:
// - leftOperand: left operand of the operation
// - operator: operator to be applied (+, -, *, /)
// - rightOperand: right operand of the operation
// Returns: the result of applying the operator to left and right operands
float calculate(float leftOperand, char operator, float rightOperand) {
    float result;
    switch (operator) {
        case '+':
            result = leftOperand + rightOperand;
            break;
        case '-':
            result = leftOperand - rightOperand;
            break;
        case '*':
            result = leftOperand * rightOperand;
            break;
        case '/':
            if (rightOperand != 0)  // Check for division by zero
                result = leftOperand / rightOperand;
            else {
                printf("Error: Division by zero\n");  // Print error for
division by zero
                exit(1);  // Exit program if division by zero
            }
            break;
        default:
            printf("Error: Invalid operator\n");  // Print error for invalid
operator
            exit(1);  // Exit program if invalid operator
    }
    return result;  // Return the calculated result
}

// Function to evaluate the expression tree recursively
// Parameters:
// - tree: pointer to the root of the expression tree
// Returns: the evaluated result of the expression
float evaluateExpressionTree(ExpressionNode *tree) {
    if (tree->nodeType == 1)  // If node is an operand
        return tree->data.operand;  // Return operand value
    float leftValue = evaluateExpressionTree(tree->leftChild);  // Evaluate
left subtree
    float rightValue = evaluateExpressionTree(tree->rightChild);  //
Evaluate right subtree
    return calculate(leftValue, tree->data.operator, rightValue);  // Apply
operator on left and right values
}

// Main function to demonstrate expression tree construction and evaluation
int main() {
    ExpressionNode *expressionTree;
    char postfixExpression[STACK_MAX] = "567*+8-";  // Postfix expression to
construct tree

    // Construct the expression tree from the postfix expression
    expressionTree = constructTree(postfixExpression);
    if (expressionTree == NULL) {  // Check if tree construction failed
        printf("Failed to construct expression tree\n");
        return 1;  // Exit program with error code
    }

    // Perform in-order traversal and print the expression tree
    printf("In-order traversal: ");
    inOrderTraversal(expressionTree);
    printf("\n");
    // Evaluate the expression tree and print the result
    printf("Evaluation result: %g\n", 
evaluateExpressionTree(expressionTree));
    return 0;  // Exit program
}
```

threaded binary tree

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

// Define the structure of a node in a threaded binary tree
typedef struct node {
    int data; // Holds the value of the node
    int hasLeftThread; // 1 if there's no left child (threaded), 0 if there 
is a left child
    int hasRightThread; // 1 if there's no right child (threaded), 0 if 
there is a right child
    struct node *leftChild; // Points to the left child or the inorder 
predecessor (if threaded)
    struct node *rightChild; // Points to the right child or the inorder 
successor (if threaded)
} NODE;

// Function to create a new node in the tree
NODE* createNode(int value) {
    // Allocate memory for a new node and set its data
    NODE *newNode = (NODE*)malloc(sizeof(NODE));
    newNode->data = value;

    // Set initial conditions for threads
    newNode->hasLeftThread = 1; // Initially, we assume no left child 
(threaded)
    newNode->hasRightThread = 1; // Initially, we assume no right child 
(threaded)
    newNode->leftChild = NULL; // No left child initially
    newNode->rightChild = NULL; // No right child initially

    return newNode;
```

```
}

// Function to insert a node in the threaded binary tree
NODE* insertNode(NODE *root, int value) {
    NODE *current = root; // Start from the root of the tree
    NODE *parent = NULL; // Track the parent of the current node
    NODE *newNode = createNode(value); // Create the new node with the given
value

    // If the tree is empty, the new node becomes the root
    if (current == NULL) {
        return newNode;
    }

    // Traverse the tree to find where the new node should be inserted
    while (current != NULL) {
        parent = current; // Keep track of the parent node
        if (value < current->data) { // Go to the left subtree if the value
is smaller
            if (current->hasLeftThread == 0) // Move left if there's a left
child
                current = current->leftChild;
            else // If no left child, stop here (this is the insertion
point)
                break;
        } else if (value > current->data) { // Go to the right subtree if
the value is greater
            if (current->hasRightThread == 0) // Move right if there's a
right child
                current = current->rightChild;
            else // If no right child, stop here (this is the insertion
point)
                break;
        } else {
            printf("\nDuplicate Key"); // Duplicate keys are not allowed
            return root;
        }
    }

    // Insert the new node as a left or right child of the parent
    if (value < parent->data) { // Insert as the left child
        newNode->leftChild = parent->leftChild; // Set thread to parent’s
current left child
        newNode->rightChild = parent; // Set the thread to parent as its
inorder successor
        parent->leftChild = newNode; // Update parent's left child to new
node
        parent->hasLeftThread = 0; // Update parent to indicate it has a
left child now
    } else { // Insert as the right child
        newNode->leftChild = parent; // Set the thread to parent as its
inorder predecessor
        newNode->rightChild = parent->rightChild; // Set thread to parent's
current right child
        parent->rightChild = newNode; // Update parent's right child to new
node
        parent->hasRightThread = 0; // Update parent to indicate it has a
right child now
    }
    return root;
}

// Function to find the inorder successor of a given node in a threaded
binary tree
NODE* inorderSuccessor(NODE *node) {
    // If there is no right child, return the right thread (inorder
successor)
    if (node->hasRightThread == 1)
        return node->rightChild;

    // If there is a right child, find the leftmost node in the right
subtree
    node = node->rightChild;
    while (node->hasLeftThread == 0)
        node = node->leftChild;
    return node;
}

// Function to perform an inorder traversal of the threaded binary tree
void inorderTraversal(NODE *root) {
    NODE *current = root;

    // Start at the leftmost node (smallest value) to begin inorder
traversal
    if (root != NULL) {
        while (current->hasLeftThread == 0)
            current = current->leftChild;

        // Traverse using the threads to visit each node in order
        while (current != NULL) {
            printf("%d ", current->data);
            current = inorderSuccessor(current); // Move to the inorder
successor
        }
    }
}

// Main function to interact with the user and perform operations on the
tree
int main() {
    NODE *threadedBST = NULL; // Initialize the root of the tree to NULL
    int value, choice; // Variables to hold the user's input value and
choice

    do {
        printf("\nEnter the data: ");
        scanf("%d", &value); // Read the integer value to be inserted
        threadedBST = insertNode(threadedBST, value); // Insert the value
into the tree

        printf("\nInorder Traversal: ");
        inorderTraversal(threadedBST); // Print the inorder traversal of the
tree

        printf("\nPress 1 to Continue, 0 to Quit: ");
        scanf("%d", &choice); // Ask the user if they want to continue or
quit
    } while (choice); // Repeat while the user enters 1

    printf("\nFinal Inorder Traversal: ");
    inorderTraversal(threadedBST); // Final traversal after all insertions
are done

    return 0;
}
```

heap

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

#define SIZE 20 // Define the maximum size of the heap

// Define a structure for the heap
// The heap will store elements in an array, and we track the size of the
heap.
typedef struct heap {
    int heap[SIZE];  // Array to store heap elements. Index 0 is unused
(sentinel value).
    int heapSize;    // Current number of elements in the heap (not counting
index 0).
} Heap;

// Function to initialize the heap
// Initializes an empty heap, with size 0, and a sentinel value at index 0.
Heap* initHeap(Heap *h) {
    h->heapSize = 0;          // Initialize the heap size to 0 (no elements
in the heap)
    h->heap[0] = INT_MAX;     // Sentinel value at index 0 (not used for
actual heap data)
    return h;                 // Return the initialized heap structure
}

// Function to swap two integers, used to maintain heap properties
// Swaps the values of the two integers passed as arguments.
void swap(int *a, int *b) {
    int temp = *a;  // Store the value of *a in a temporary variable
    *a = *b;        // Set *a to the value of *b
    *b = temp;      // Set *b to the value stored in temp
}

// Function to insert a new value into the heap using the Top-Down approach
// Adds an element at the end and then "bubbles it up" to maintain the max-
heap property.
Heap* insertTopDown(Heap *h, int data) {
    int childIndex, parentIndex;

    // Insert the new data at the end of the heap and increment heap size
    h->heap[++(h->heapSize)] = data;

    // Start with the last element added, and move upwards to restore max-
heap property
    childIndex = h->heapSize;      // The last added element is at the
current heapSize
    parentIndex = childIndex / 2;  // Parent index is half of the current
child index

    // "Bubble up" to maintain the max-heap property
    // As long as the parent is smaller than the child, swap them
    while (h->heap[parentIndex] < h->heap[childIndex]) {
        swap(&h->heap[parentIndex], &h->heap[childIndex]); // Swap the
parent and child
        childIndex = parentIndex;   // Move up to the parent
        parentIndex = childIndex / 2;  // Update parent index
    }
    return h;  // Return the heap after insertion
}

// Function to construct the heap using the Bottom-Up approach
// Starts from the last non-leaf node and "bubbles down" to maintain the
max-heap property.
void insertBottomUp(Heap *h) {
    int parentIndex, childIndex;

    // Start from the last non-leaf node (heapSize/2) and work upwards
    for (int i = h->heapSize / 2; i >= 1; i--) {
        parentIndex = i;             // Set the current node as the parent
        childIndex = 2 * parentIndex; // Left child index (right child is
childIndex + 1)

        // Move down the tree to restore the max-heap property
        while (childIndex <= h->heapSize) {
            // If the right child exists and is larger than the left, choose
the right child
            if (childIndex + 1 <= h->heapSize && h->heap[childIndex + 1] >
h->heap[childIndex]) {
                childIndex++;  // Set child index to the right child if it's
larger
            }

            // If the parent is smaller than the larger child, swap them
            if (h->heap[parentIndex] < h->heap[childIndex]) {
                swap(&h->heap[parentIndex], &h->heap[childIndex]);
            } else {
                break;  // Stop if the parent is larger than both children
            }

            // Update parent and child indices to continue down the tree
            parentIndex = childIndex;
            childIndex = 2 * parentIndex;
        }
    }
}

// Function to display the contents of the heap in level order
// Prints the heap array, which represents the heap in level order.
void displayHeap(Heap *h) {
    if (h != NULL) {  // Check if the heap is initialized
        for (int i = 1; i <= h->heapSize; i++) {  // Iterate through the
heap array
            printf("%d ", h->heap[i]);  // Print each element
        }
    }
}

// Function to delete the root element (highest priority element) from the
heap
// Replaces the root with the last element, then "bubbles down" to restore
```

```
heap property.
int deleteRoot(Heap *h) {
    int rootData, parentIndex, childIndex;

    rootData = h->heap[1];  // Store root element to return later (max
element)
    h->heap[1] = h->heap[h->heapSize--];  // Move the last element to the
root and decrease heap size

    parentIndex = 1;        // Start at the root
    childIndex = 2 * parentIndex;  // Left child of the root

    // Bubble down the root element to restore the heap property
    while (childIndex <= h->heapSize) {
        // If the right child exists and is larger than the left, choose the
right child
        if (childIndex + 1 <= h->heapSize && h->heap[childIndex + 1] > h-
>heap[childIndex]) {
            childIndex++;  // Choose the right child if it's larger
        }

        // If the parent is smaller than the selected child, swap them
        if (h->heap[parentIndex] < h->heap[childIndex]) {
            swap(&h->heap[parentIndex], &h->heap[childIndex]);
        } else {
            break;  // Stop if the max-heap property is restored
        }

        // Update parent and child indices to continue down the tree
        parentIndex = childIndex;
        childIndex = 2 * parentIndex;
    }
    return rootData;  // Return the original root element (max element)
}

// Main function to demonstrate heap operations
void main() {
    Heap *topDownHeap, *bottomUpHeap;
    int numNodes, data;
    // Allocate memory for the top-down heap and initialize it
    topDownHeap = (Heap*)malloc(sizeof(Heap));
    topDownHeap = initHeap(topDownHeap);

    // Allocate memory for the bottom-up heap and initialize it
    bottomUpHeap = (Heap*)malloc(sizeof(Heap));
    bottomUpHeap = initHeap(bottomUpHeap);

    // Input: ask user for the number of elements to insert
    printf("Enter the number of nodes: ");
    scanf("%d", &numNodes);

    // Insert each element into both heaps
    for (int i = 1; i <= numNodes; i++) {
        printf("Enter data: ");
        scanf("%d", &data);

        // Insert data into the top-down heap using the Top-Down approach
        topDownHeap = insertTopDown(topDownHeap, data);

        // Insert data into the bottom-up heap without heapifying (just
inserting in level order)
        bottomUpHeap->heap[++(bottomUpHeap->heapSize)] = data;
    }

    // Display the heap created by Top-Down insertion
    printf("\nTop-Down Heap Construction:\n");
    displayHeap(topDownHeap);

    // Now, heapify the bottom-up heap to make it a valid max-heap
    insertBottomUp(bottomUpHeap);
    printf("\nBottom-Up Heap Construction:\n");
    displayHeap(bottomUpHeap);

    // Demonstrate priority queue behavior by repeatedly removing the max
element
    printf("\n\n***Priority Queue (Removing max elements in order)***\n");
    numNodes = bottomUpHeap->heapSize; // Store the number of elements in
the heap
    for (int i = 1; i <= numNodes; i++) {
        printf("%d ", deleteRoot(bottomUpHeap));  // Delete and print each
max element
    }
}
```

heapify function

```
void heapify(Heap *h, int i) {
    int largest = i;
    int left = 2 * i;
    int right = 2 * i + 1;

    // If left child exists and is greater than the current largest, update
largest
    if (left <= h->heapSize && h->heap[left] > h->heap[largest]) {
        largest = left;
    }

    // If right child exists and is greater than the current largest, update
largest
    if (right <= h->heapSize && h->heap[right] > h->heap[largest]) {
        largest = right;
    }

    // If largest is not the current node, swap and recursively heapify the
affected subtree
    if (largest != i) {
        swap(&h->heap[i], &h->heap[largest]);
        heapify(h, largest);  // Recursively heapify the affected subtree
    }
}
```

graph adjaceny matrix

using linked list

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

#define MAX 10  // Define a constant for the maximum number of vertices

// Define a structure 'node' to represent each node in the adjacency list
typedef struct node {
    int vertexNumber;        // Stores the vertex number (for each node in
the adjacency list)
    struct node *nextVertex; // Pointer to the next node in the linked list
(next adjacent vertex)
} NODE;

// Function to create the adjacency list
void createAdjacencyList(NODE *adjList) {
    int totalVertices, fromVertex, toVertex;     // Variables to hold the
number of vertices and the from/to nodes for edges
    NODE *traverseNode, *newNode;        // traverseNode: pointer to
traverse the linked list, newNode: new node to be added

    // Prompt the user to input the number of nodes (vertices)
    printf("\nEnter the number of vertices: ");
    scanf("%d", &totalVertices);

    // Store the number of nodes in the first element of the adjacency list
array
    // This will be our "header" node that stores information about the
graph (total nodes)
    adjList[0].vertexNumber = totalVertices;  // adjList[0] stores the
number of vertices in the graph
    adjList[0].nextVertex = NULL;             // Initialize the 'next'
pointer of the first node to NULL

    // Initialize the adjacency list for each vertex (starting from vertex
1)
    // For each vertex, create an entry with its own data and set the next
pointer to NULL
    for (int vertex = 1; vertex <= adjList[0].vertexNumber; vertex++) {
        adjList[vertex].vertexNumber = vertex;  // The vertexNumber field of
the node stores the vertex number (1 to totalVertices)
```

```
        adjList[vertex].nextVertex = NULL;      // Initially, there are no
edges, so the 'nextVertex' pointer is NULL
    }

    // Start taking input of edges (pairs of vertices)
    // We will loop indefinitely to allow the user to input multiple edges
    while (1) {
        // Prompt the user to enter the 'from' and 'to' vertices for an edge
        printf("\nEnter the 'from' and 'to' vertices (enter out of range
numbers to stop): ");
        scanf("%d %d", &fromVertex, &toVertex);

        // Check if the input vertices are within the valid range
        // Ensure that both 'fromVertex' and 'toVertex' vertices are valid
(1 <= fromVertex, toVertex <= totalVertices)
        if (fromVertex > 0 && fromVertex <= totalVertices && toVertex > 0 &&
toVertex <= totalVertices) {
            // If the vertices are valid, we will add an edge between them

            // Find the node corresponding to the 'fromVertex' in the
adjacency list
            traverseNode = &adjList[fromVertex];  // 'traverseNode' points
to the node representing the 'fromVertex'

            // Traverse the adjacency list for the 'fromVertex' to find the
last node
            while (traverseNode->nextVertex != NULL) {  // Move
'traverseNode' to the last node in the list for this vertex
                traverseNode = traverseNode->nextVertex;  // Move to the
next node
            }

            // Create a new node to represent the 'toVertex' and add it to
the adjacency list
            newNode = (NODE*)malloc(sizeof(NODE)); // Dynamically allocate
memory for the new node

            newNode->vertexNumber = toVertex;  // Set the 'vertexNumber'
field of the new node to the 'toVertex' number
            newNode->nextVertex = NULL;        // The 'nextVertex' pointer
of the new node is NULL because it will be the last node

            // Link the new node to the adjacency list of the 'fromVertex'
            traverseNode->nextVertex = newNode;  // Add the new node to the
end of the linked list of the 'fromVertex'
        } else {
            // If the input vertices are out of range (not valid), break the
loop
            break; // Exit the edge input loop when invalid input is
detected
        }
    }
}

// Function to display the adjacency list
void displayAdjacencyList(NODE *adjList) {
    NODE *traverseNode;  // 'traverseNode' is a pointer used to traverse the
linked list for each vertex

    // Iterate through each vertex and print its adjacency list
    for (int vertex = 1; vertex <= adjList[0].vertexNumber; vertex++) {  //
Loop from vertex 1 to the total number of vertices (adjList[0].vertexNumber)
        printf("\n%d->", adjList[vertex].vertexNumber);  // Print the
current vertex number (adjList[vertex].vertexNumber)

        // Start from the first node in the adjacency list of the current
vertex 'vertex'
        traverseNode = &adjList[vertex];  // 'traverseNode' points to the
node representing the current vertex

        // Traverse the adjacency list for this vertex and print each
adjacent vertex
        while (traverseNode->nextVertex != NULL) {  // Traverse the list
until we reach the last node
            traverseNode = traverseNode->nextVertex;  // Move to the next
node in the list
            printf("%d->", traverseNode->vertexNumber);  // Print the
'toVertex' of the current edge
        }
        // After printing all adjacent vertices, print NULL to indicate the
end of the list for this vertex
        printf("NULL");
    }
}

// Main function - the entry point of the program
int main() {
    NODE adjList[MAX];  // Declare an array of 'NODE' structures to hold the
adjacency list

    // Create the adjacency list by taking input from the user
    createAdjacencyList(adjList);

    // Display the adjacency list (the representation of the graph)
    displayAdjacencyList(adjList);

    return 0;  // Return 0 to indicate successful execution of the program
}
```

using matrix

```
#include<stdio.h>
#define MAX 10  // Define a constant for the maximum number of vertices

// Define a structure to represent the graph
typedef struct graph {
    int totalVertices;           // Total number of vertices in the graph
    int adjacencyMatrix[MAX][MAX]; // 2D array to store the adjacency matrix
} GRAPH;

// Function to create the adjacency matrix for the graph
void createAdjacencyMatrix(GRAPH *graph) {
    int fromVertex, toVertex; // Variables to hold the starting and ending
vertices for edges

    // Prompt the user to input the total number of vertices
    printf("\nEnter the number of vertices: ");
    scanf("%d", &graph->totalVertices);
    // Initialize the adjacency matrix
    // Row 0 and column 0 will be used to label the vertices for display
purposes
    for (int row = 1; row <= graph->totalVertices; row++) {
        graph->adjacencyMatrix[0][row] = row; // Label columns with vertex
numbers
        graph->adjacencyMatrix[row][0] = row; // Label rows with vertex
numbers

        for (int col = 1; col <= graph->totalVertices; col++) {
            graph->adjacencyMatrix[row][col] = 0; // Initialize all values
to 0 (no edges)
        }
    }

    // Input edges and update the adjacency matrix
    while (1) {
        // Prompt the user to input an edge between two vertices
        printf("\nEnter the 'from' and 'to' vertices (enter out of range
values to stop): ");
        scanf("%d %d", &fromVertex, &toVertex);

        // Validate the input vertices
        if (fromVertex > 0 && fromVertex <= graph->totalVertices &&
            toVertex > 0 && toVertex <= graph->totalVertices) {
            // Add an edge by setting the corresponding matrix entry to 1
            graph->adjacencyMatrix[fromVertex][toVertex] = 1;

            // Uncomment the next line for an undirected graph
            // graph->adjacencyMatrix[toVertex][fromVertex] = 1;
        } else {
            // Break the loop if the input vertices are out of range
            break;
        }
    }
}

// Function to display the adjacency matrix
void displayAdjacencyMatrix(GRAPH graph) {
    // Iterate through the rows and columns of the matrix for display
    for (int row = 0; row <= graph.totalVertices; row++) {
        printf("\n"); // Move to the next row for display

        for (int col = 0; col <= graph.totalVertices; col++) {
            // Print labels for the first row and column
            if (row == 0 && col == 0) {
                printf("   "); // Empty space for the top-left corner
            } else {
                // Print the matrix value or labels
                printf("%d ", graph.adjacencyMatrix[row][col]);
            }
        }
    }
}

int main() {
    GRAPH graph; // Declare a variable to represent the graph

    // Create the adjacency matrix for the graph
    createAdjacencyMatrix(&graph);

    // Display the adjacency matrix
    displayAdjacencyMatrix(graph);

    return 0; // Return 0 to indicate successful execution of the program
}
```

graph

bfs connected

```
#include <stdio.h>
#define MAX 10

// Function prototypes
void createGraph(int adjacencyMatrix[MAX][MAX], int vertices);
int connectivity(int adjacencyMatrix[MAX][MAX], int vertices);
void bfs(int adjacencyMatrix[MAX][MAX], int vertices, int visited[MAX], int
source);

int main() {
    int adjacencyMatrix[MAX][MAX] = {0}; // Initialize adjacency matrix with
0
    int vertices;

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

    printf("Enter the adjacency information:\n");
    createGraph(adjacencyMatrix, vertices);

    // Check if the graph is connected
    if (connectivity(adjacencyMatrix, vertices))
        printf("The undirected graph is connected.\n");
    else
        printf("The undirected graph is disconnected.\n");

    return 0;
}

// Function to create the graph using adjacency matrix
void createGraph(int adjacencyMatrix[MAX][MAX], int vertices) {
    int src, dest;

    printf("Enter the source and destination vertices of the edges (enter -1
-1 to stop):\n");

    while (1) {
        scanf("%d%d", &src, &dest);
        if (src == -1 && dest == -1) // Stop when user enters -1 -1
            break;

        // Add edge to the adjacency matrix (for undirected graph)
        adjacencyMatrix[src][dest] = 1;
        adjacencyMatrix[dest][src] = 1;
    }
}
```

```c
// BFS traversal of the graph from the source vertex
void bfs(int adjacencyMatrix[MAX][MAX], int vertices, int visited[MAX], int
source) {
    int queue[MAX];  // Queue for BFS
    int front = 0, rear = -1; // Initialize queue pointers

    // Enqueue the source vertex and mark it as visited
    queue[++rear] = source;
    visited[source] = 1;

    while (front <= rear) {
        int currentVertex = queue[front++]; // Dequeue a vertex

        // Explore all adjacent vertices of the current vertex
        for (int i = 0; i < vertices; i++) {
            if (adjacencyMatrix[currentVertex][i] == 1 && visited[i] == 0) {
                queue[++rear] = i; // Enqueue the adjacent vertex
                visited[i] = 1;    // Mark the adjacent vertex as visited
            }
        }
    }
}

// Function to check if the graph is connected
int connectivity(int adjacencyMatrix[MAX][MAX], int vertices) {
    int visited[MAX] = {0}; // Array to track visited vertices

    // Perform BFS starting from vertex 0
    bfs(adjacencyMatrix, vertices, visited, 0);

    // Check if all vertices are visited
    for (int i = 0; i < vertices; i++) {
        if (visited[i] == 0) {
            return 0; // Graph is disconnected
        }
    }
    return 1; // Graph is connected
}
```

dfs connected

```
#include <stdio.h>
#define MAX 10

// Function prototypes
void createGraph(int adjacencyMatrix[MAX][MAX], int vertices);
void dfs(int adjacencyMatrix[MAX][MAX], int vertices, int visited[MAX], int
source);
int connectivity(int adjacencyMatrix[MAX][MAX], int vertices);

int main() {
    int adjacencyMatrix[MAX][MAX] = {0}; // Initialize adjacency matrix with
0
    int vertices;

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

    printf("Enter the adjacency information:\n");
    createGraph(adjacencyMatrix, vertices);

    // Check if the graph is connected
    if (connectivity(adjacencyMatrix, vertices))
        printf("The undirected graph is connected.\n");
    else
        printf("The undirected graph is disconnected.\n");

    return 0;
}

// Function to create a graph using an adjacency matrix
void createGraph(int adjacencyMatrix[MAX][MAX], int vertices) {
    int src, dest;

    printf("Enter the source and destination vertices of the edges (enter -1
-1 to stop):\n");

    while (1) {
        scanf("%d%d", &src, &dest);
        if (src == -1 && dest == -1) // Stop when user enters -1 -1
            break;

        // Add edge to the adjacency matrix (for undirected graph)
        adjacencyMatrix[src][dest] = 1;
        adjacencyMatrix[dest][src] = 1;
    }
}

// DFS traversal of the graph from the source vertex
void dfs(int adjacencyMatrix[MAX][MAX], int vertices, int visited[MAX], int
source) {
    visited[source] = 1; // Mark the source vertex as visited

    // Explore all adjacent vertices
    for (int i = 0; i < vertices; i++) {
        if (adjacencyMatrix[source][i] == 1 && visited[i] == 0) {
            dfs(adjacencyMatrix, vertices, visited, i);
        }
    }
}

// Function to check if the graph is connected
int connectivity(int adjacencyMatrix[MAX][MAX], int vertices) {
    int visited[MAX] = {0}; // Array to track visited vertices

    // Perform DFS starting from vertex 0
    dfs(adjacencyMatrix, vertices, visited, 0);

    // Check if all vertices are visited
    for (int i = 0; i < vertices; i++) {
        if (visited[i] == 0) {
            return 0; // Graph is disconnected
        }
    }
    return 1; // Graph is connected
}
```

bfs strong weak connected

```
#include <stdio.h>
#define MAX 10 // Maximum number of vertices allowed in the graph

// Function prototypes
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices);
int isStronglyConnected(int adjacencyMatrix[MAX][MAX], int numVertices);
int isWeaklyConnected(int undirectedMatrix[MAX][MAX], int numVertices);
void convertToUndirectedGraph(int directedMatrix[MAX][MAX], int
undirectedMatrix[MAX][MAX], int numVertices);
void bfsTraversal(int graphMatrix[MAX][MAX], int numVertices, int
visitedVertices[MAX], int startVertex);

int main() {
    int directedGraph[MAX][MAX] = {0};  // Directed graph adjacency matrix
    int undirectedGraph[MAX][MAX] = {0}; // Undirected graph adjacency
matrix
    int numVertices; // Number of vertices in the graph

    // Input the number of vertices
    printf("Enter the number of vertices:\n");
    scanf("%d", &numVertices);

    // Input adjacency information to create the directed graph
    printf("Enter the adjacency information:\n");
    createGraph(directedGraph, numVertices);

    // Check if the graph is strongly connected
    if (isStronglyConnected(directedGraph, numVertices)) {
        printf("The digraph is strongly connected\n");
    } else {
        // Convert the directed graph to an undirected graph
        convertToUndirectedGraph(directedGraph, undirectedGraph,
numVertices);

        // Check if the graph is weakly connected
        if (isWeaklyConnected(undirectedGraph, numVertices)) {
            printf("The digraph is weakly connected\n");
        } else {
            // If neither strongly nor weakly connected, it's disconnected
            printf("The digraph is disconnected\n");
        }
    }
}

// Function to create a directed graph based on user input
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices) {
    printf("Enter the source and destination vertex of the edges (e.g., 0 
1):\n");
    printf("Enter -1 -1 to stop entering edges\n");
    int sourceVertex, destVertex;
    while (1) {
        scanf("%d%d", &sourceVertex, &destVertex);
        if (sourceVertex == -1 && destVertex == -1) // Stop condition
            break;
        adjacencyMatrix[sourceVertex][destVertex] = 1; // Mark the edge in 
the adjacency matrix
    }
}

// Function to perform Breadth-First Search (BFS) on the graph
void bfsTraversal(int graphMatrix[MAX][MAX], int numVertices, int 
visitedVertices[MAX], int startVertex) {
    int queue[MAX];  // Queue to process BFS
    int front = 0, rear = -1; // Initialize the queue pointers

    queue[++rear] = startVertex; // Enqueue the start vertex
    visitedVertices[startVertex] = 1; // Mark the start vertex as visited

    while (front <= rear) { // While the queue is not empty
        int currentVertex = queue[front++]; // Dequeue the front element

        // Traverse all adjacent vertices
        for (int adjVertex = 0; adjVertex < numVertices; adjVertex++) {
            // If there is an edge and the vertex is not visited
            if (graphMatrix[currentVertex][adjVertex] == 1 && 
visitedVertices[adjVertex] == 0) {
                queue[++rear] = adjVertex; // Enqueue the adjacent vertex
                visitedVertices[adjVertex] = 1; // Mark it as visited
            }
        }
    }
}

// Function to check if the directed graph is strongly connected
int isStronglyConnected(int adjacencyMatrix[MAX][MAX], int numVertices) {
    int visitedVertices[MAX]; // Array to track visited vertices

    for (int startVertex = 0; startVertex < numVertices; startVertex++) {
        // Reset visited array for each vertex
        for (int i = 0; i < numVertices; i++) {
            visitedVertices[i] = 0;
        }

        // Perform BFS from the current vertex
        bfsTraversal(adjacencyMatrix, numVertices, visitedVertices,
startVertex);

        // Check if all vertices are visited
        for (int i = 0; i < numVertices; i++) {
            if (visitedVertices[i] == 0) {
                return 0; // Not strongly connected if any vertex is not
visited
            }
        }
    }
    return 1; // All vertices are reachable from every other vertex
}

// Function to convert a directed graph to an undirected graph
void convertToUndirectedGraph(int directedMatrix[MAX][MAX], int
undirectedMatrix[MAX][MAX], int numVertices) {
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            if (directedMatrix[i][j] == 1) { // If there is a directed edge
                undirectedMatrix[i][j] = 1; // Add the edge (i -> j)
                undirectedMatrix[j][i] = 1; // Add the reverse edge (j -> i)
            }
        }
    }
}
```

```
// Function to check if an undirected graph is connected
int isWeaklyConnected(int graphMatrix[MAX][MAX], int numVertices) {
    int visitedVertices[MAX] = {0}; // Array to track visited vertices

    // Perform BFS from the first vertex
    bfsTraversal(graphMatrix, numVertices, visitedVertices, 0);

    // Check if all vertices are visited
    for (int i = 0; i < numVertices; i++) {
        if (visitedVertices[i] == 0) {
            return 0; // Not connected if any vertex is not visited
        }
    }
    return 1; // All vertices are reachable in the undirected graph
}
```

bfs check path

```
#include <stdio.h>
#define MAX 10  // Maximum number of vertices in the graph

// Function prototypes
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices);
int bfsCheckPath(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int sourceVertex, int destinationVertex, int parent[MAX]);
void printPath(int parent[MAX], int sourceVertex, int destinationVertex);

int main() {
    int adjacencyMatrix[MAX][MAX] = {0}; // Adjacency matrix to represent
the graph
    int numVertices; // Number of vertices in the graph
    int visited[MAX] = {0}; // Array to track visited vertices during BFS
    int sourceVertex, destinationVertex; // Source and destination vertices
for the path check
    int parent[MAX]; // Array to store the parent of each vertex for path
reconstruction

    // Input the number of vertices
```

```c
    printf("Enter the number of vertices\n");
    scanf("%d", &numVertices);

    // Create the graph by filling the adjacency matrix
    printf("Enter the adjacency information\n");
    createGraph(adjacencyMatrix, numVertices);

    // Input the source and destination vertices
    printf("Enter the source & destination vertex\n");
    scanf("%d%d", &sourceVertex, &destinationVertex);

    // Perform BFS to check if a path exists from source to destination
    if (bfsCheckPath(adjacencyMatrix, numVertices, visited, sourceVertex,
destinationVertex, parent)) {
        printf("There exists a path from %d to %d\n", sourceVertex,
destinationVertex);
        printf("The path is: ");
        printPath(parent, sourceVertex, destinationVertex); // Print the
reconstructed path
        printf("\n");
    } else {
        printf("There exists no path from %d to %d\n", sourceVertex,
destinationVertex);
    }

    return 0;
}

// Function to create the graph by inputting edges into the adjacency matrix
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices) {
    printf("Enter the src & dest vertex of the edge\n");
    printf("Enter -1 -1 to stop\n");
    int sourceVertex, destinationVertex;

    // Input edges until the user enters -1 -1
    while (1) {
        scanf("%d%d", &sourceVertex, &destinationVertex);
        if (sourceVertex == -1 && destinationVertex == -1) {
            break; // Stop input when -1 -1 is entered
        }
        adjacencyMatrix[sourceVertex][destinationVertex] = 1; // Add edge
source -> destination
        adjacencyMatrix[destinationVertex][sourceVertex] = 1; // Add edge
destination -> source (undirected graph)
    }
}

// Function to perform BFS and check if a path exists from source to
destination
int bfsCheckPath(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int sourceVertex, int destinationVertex, int parent[MAX]) {
    int queue[MAX]; // Queue to facilitate BFS
    int front = 0, rear = -1; // Initialize queue front and rear pointers

    queue[++rear] = sourceVertex; // Enqueue the source vertex
    visited[sourceVertex] = 1; // Mark the source vertex as visited
    parent[sourceVertex] = -1; // Source vertex has no parent

    int currentVertex;
    while (front <= rear) { // Continue BFS until the queue is empty
        currentVertex = queue[front++]; // Dequeue a vertex from the queue

        // Iterate over all vertices to check for adjacency
        for (int adjacentVertex = 0; adjacentVertex < numVertices;
adjacentVertex++) {
            // If there is an edge and the adjacent vertex is not yet
visited
            if (adjacencyMatrix[currentVertex][adjacentVertex] == 1 &&
visited[adjacentVertex] == 0) {
                parent[adjacentVertex] = currentVertex; // Set parent of the
adjacent vertex
                if (adjacentVertex == destinationVertex) { // If destination
is found
                    return 1; // Path exists
                }
                queue[++rear] = adjacentVertex; // Enqueue the adjacent
vertex
                visited[adjacentVertex] = 1; // Mark the adjacent vertex as
visited
            }
        }
    }
    return 0; // No path exists from source to destination
}

// Function to reconstruct and print the path from source to destination
void printPath(int parent[MAX], int sourceVertex, int destinationVertex) {
    if (destinationVertex == -1) {
        return; // Base case: reached the source vertex
    }

    int path[MAX]; // Array to store the path
    int pathLength = 0; // Variable to track the path length

    // Backtrack from destination to source using the parent array
    while (destinationVertex != -1) {
        path[pathLength++] = destinationVertex; // Add the current vertex to
the path
        destinationVertex = parent[destinationVertex]; // Move to the parent
of the current vertex
    }

    // Print the path in the correct order (reverse of the backtracking
order)
    for (int i = pathLength - 1; i >= 0; i--) {
        printf("%d ", path[i]);
    }
}
```

dfs path check

```
#include <stdio.h>
#define MAX 10  // Maximum number of vertices in the graph

// Function prototypes
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices);
int dfsCheckPath(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int sourceVertex, int destinationVertex, int parent[MAX]);
void printPath(int parent[MAX], int sourceVertex, int destinationVertex);
```

```c
int main() {
    int adjacencyMatrix[MAX][MAX] = {0}; // Adjacency matrix to represent
the graph
    int numVertices; // Number of vertices in the graph
    int visited[MAX] = {0}; // Array to track visited vertices during DFS
    int sourceVertex, destinationVertex; // Source and destination vertices
for the path check
    int parent[MAX]; // Array to store the parent of each vertex for path
reconstruction

    // Input the number of vertices
    printf("Enter the number of vertices\n");
    scanf("%d", &numVertices);

    // Create the graph by filling the adjacency matrix
    printf("Enter the adjacency information\n");
    createGraph(adjacencyMatrix, numVertices);

    // Input the source and destination vertices
    printf("Enter the source & destination vertex\n");
    scanf("%d%d", &sourceVertex, &destinationVertex);

    // Perform DFS to check if a path exists from source to destination
    if (dfsCheckPath(adjacencyMatrix, numVertices, visited, sourceVertex,
destinationVertex, parent)) {
        printf("There exists a path from %d to %d\n", sourceVertex,
destinationVertex);
        printf("The path is: ");
        printPath(parent, sourceVertex, destinationVertex); // Print the
reconstructed path
        printf("\n");
    } else {
        printf("There exists no path from %d to %d\n", sourceVertex,
destinationVertex);
    }

    return 0;
}
```

```
// Function to create the graph by inputting edges into the adjacency matrix
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices) {
    printf("Enter the source and destination vertex of the edge\n");
    printf("Enter -1 -1 to stop\n");
    int sourceVertex, destinationVertex;

    // Input edges until the user enters -1 -1
    while (1) {
        scanf("%d%d", &sourceVertex, &destinationVertex);
        if (sourceVertex == -1 && destinationVertex == -1) {
            break; // Stop input when -1 -1 is entered
        }
        adjacencyMatrix[sourceVertex][destinationVertex] = 1; // Add edge
source -> destination
    }
}

// Recursive DFS function to check if a path exists
int dfsCheckPath(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int sourceVertex, int destinationVertex, int parent[MAX]) {
    visited[sourceVertex] = 1; // Mark the current vertex as visited

    // Base case: if source matches destination
    if (sourceVertex == destinationVertex) {
        return 1;
    }

    // Explore all vertices adjacent to the current vertex
    for (int adjacentVertex = 0; adjacentVertex < numVertices;
adjacentVertex++) {
        // Check if there is an edge and the vertex is not yet visited
        if (adjacencyMatrix[sourceVertex][adjacentVertex] == 1 &&
!visited[adjacentVertex]) {
            parent[adjacentVertex] = sourceVertex; // Record the parent of
the adjacent vertex
            // Recursive call to check for a path from the adjacent vertex
            if (dfsCheckPath(adjacencyMatrix, numVertices, visited,
adjacentVertex, destinationVertex, parent)) {
                return 1;
            }
        }
    }

    return 0; // No path exists
}

// Function to reconstruct and print the path from source to destination
void printPath(int parent[MAX], int sourceVertex, int destinationVertex) {
    if (destinationVertex == -1) {
        return; // Base case: reached the source vertex
    }

    int path[MAX]; // Array to store the path
    int pathLength = 0; // Variable to track the path length

    // Backtrack from destination to source using the parent array
    while (destinationVertex != -1) {
        path[pathLength++] = destinationVertex; // Add the current vertex to
the path
        destinationVertex = parent[destinationVertex]; // Move to the parent
of the current vertex
    }

    // Print the path in the correct order (reverse of the backtracking
order)
    for (int i = pathLength - 1; i >= 0; i--) {
        printf("%d ", path[i]);
    }
}
```

connection

```
#include <stdio.h>
#define MAX 10 // Maximum number of vertices in the graph

// Function prototypes
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices);
int isConnected(int adjacencyMatrix[MAX][MAX], int numVertices);
int hasCycle(int adjacencyMatrix[MAX][MAX], int numVertices);
int bfsCycleDetection(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int startVertex, int parent[MAX]);
void bfsTraversal(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int startVertex);

int main() {
    int adjacencyMatrix[MAX][MAX] = {0}; // Adjacency matrix to represent
the graph
    int numVertices; // Number of vertices in the graph

    // Input the number of vertices
    printf("Enter the number of vertices\n");
    scanf("%d", &numVertices);

    // Input the adjacency matrix by defining edges
    printf("Enter the adjacency information\n");
    createGraph(adjacencyMatrix, numVertices);

    // Check graph connectivity
    if (isConnected(adjacencyMatrix, numVertices)) {
        printf("The undirected graph is connected\n");
    } else {
        printf("The undirected graph is disconnected\n");
    }

    // Check if the graph contains a cycle
    if (hasCycle(adjacencyMatrix, numVertices)) {
        printf("The graph contains a cycle\n");
    } else {
        printf("The graph does not contain a cycle\n");
    }

    return 0;
}

// Function to create the adjacency matrix by adding edges
void createGraph(int adjacencyMatrix[MAX][MAX], int numVertices) {
    printf("Enter the source and destination vertex of the edge\n");
    printf("Enter -1 -1 to stop\n");
    int sourceVertex, destinationVertex;
    while (1) {
        scanf("%d%d", &sourceVertex, &destinationVertex);
        if (sourceVertex == -1 && destinationVertex == -1) {
            break; // Exit when -1 -1 is entered
        }
        adjacencyMatrix[sourceVertex][destinationVertex] = 1; // Add edge
source -> destination
        adjacencyMatrix[destinationVertex][sourceVertex] = 1; // Add edge
destination -> source (undirected)
    }
}

// Function to detect a cycle in the graph using BFS
int bfsCycleDetection(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int startVertex, int parent[MAX]) {
    int queue[MAX]; // Queue for BFS
    int front = 0, rear = -1; // Queue front and rear pointers

    queue[++rear] = startVertex; // Enqueue the starting vertex
    visited[startVertex] = 1; // Mark the starting vertex as visited
    parent[startVertex] = -1; // Set the parent of the starting vertex to -1

    while (front <= rear) { // While the queue is not empty
        int currentVertex = queue[front++]; // Dequeue a vertex

        // Check all adjacent vertices of the current vertex
        for (int adjacentVertex = 0; adjacentVertex < numVertices;
adjacentVertex++) {
            if (adjacencyMatrix[currentVertex][adjacentVertex] == 1) { // If
there is an edge
                if (!visited[adjacentVertex]) { // If the adjacent vertex is
not visited
                    queue[++rear] = adjacentVertex; // Enqueue the adjacent
vertex
                    visited[adjacentVertex] = 1; // Mark it as visited
                    parent[adjacentVertex] = currentVertex; // Set its
parent
                } else if (parent[currentVertex] != adjacentVertex) { //
Cycle detected
```

```
                    return 1;
                }
            }
        }
    }
    return 0; // No cycle found
}

// BFS function to check connectivity of the graph
void bfsTraversal(int adjacencyMatrix[MAX][MAX], int numVertices, int
visited[MAX], int startVertex) {
    int queue[MAX]; // Queue for BFS
    int front = 0, rear = -1; // Queue front and rear pointers

    queue[++rear] = startVertex; // Enqueue the starting vertex
    visited[startVertex] = 1; // Mark the starting vertex as visited

    while (front <= rear) { // While the queue is not empty
        int currentVertex = queue[front++]; // Dequeue a vertex

        // Check all adjacent vertices of the current vertex
        for (int adjacentVertex = 0; adjacentVertex < numVertices;
adjacentVertex++) {
            if (adjacencyMatrix[currentVertex][adjacentVertex] == 1 &&
!visited[adjacentVertex]) { // If there's an edge and not visited
                queue[++rear] = adjacentVertex; // Enqueue the adjacent
vertex
                visited[adjacentVertex] = 1; // Mark it as visited
            }
        }
    }
}

// Function to check if the graph is connected
int isConnected(int adjacencyMatrix[MAX][MAX], int numVertices) {
    int visited[MAX] = {0}; // Array to track visited vertices
    bfsTraversal(adjacencyMatrix, numVertices, visited, 0); // Perform BFS
from vertex 0

    // Check if all vertices are visited
    for (int i = 0; i < numVertices; i++) {
        if (!visited[i]) {
            return 0; // Graph is disconnected
        }
    }
    return 1; // Graph is connected
}

// Function to check if the graph contains a cycle
int hasCycle(int adjacencyMatrix[MAX][MAX], int numVertices) {
    int visited[MAX] = {0}; // Array to track visited vertices
    int parent[MAX] = {-1}; // Array to track parent of each vertex

    // Perform BFS for each connected component of the graph
    for (int i = 0; i < numVertices; i++) {
        if (!visited[i]) { // If the vertex is not visited
            if (bfsCycleDetection(adjacencyMatrix, numVertices, visited, i,
parent)) {
                return 1; // Cycle found
            }
        }
    }
    return 0; // No cycle found
}
```