LAB ASSIGNMENT

TREES

1.Write a Program to create a Binary Tree and perform following non recursive operations on it. a. Preorder Traversal b. Postorder Traversal c. Count total no. of nodes d. Display height of a 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;
}

void insert(struct node **root, int data) {
    if (*root == NULL) {
        *root = createNode(data);
        return;
    }
    struct node *temp = *root;
    while (temp != NULL) {
        if (data < temp->data) {
            if (temp->left == NULL) {
                temp->left = createNode(data);
                return;
            }
            temp = temp->left;
        } else if (data > temp->data) {
            if (temp->right == NULL) {
                temp->right = createNode(data);
                return;
            }
            temp = temp->right;
        } else {
            return;
```

```
        }
    }
}

void preorder(struct node *root) {
    if (root == NULL) {
        return;
    }
    struct node *stack[100];
    int top = -1;
    stack[++top] = root;
    while (top >= 0) {
        struct node *temp = stack[top--];
        printf("%d ", temp->data);
        if (temp->right != NULL) {
            stack[++top] = temp->right;
        }
        if (temp->left != NULL) {
            stack[++top] = temp->left;
        }
    }
}

void postorder(struct node *root) {
    if (root == NULL) {
        return;
    }
    struct node *stack1[100], *stack2[100];
    int top1 = -1, top2 = -1;
    stack1[++top1] = root;
    while (top1 >= 0) {
        struct node *temp = stack1[top1--];
        stack2[++top2] = temp;
        if (temp->left != NULL) {
            stack1[++top1] = temp->left;
        }
        if (temp->right != NULL) {
            stack1[++top1] = temp->right;
        }
    }
    while (top2 >= 0) {
        struct node *temp = stack2[top2--];
        printf("%d ", temp->data);
    }
}

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

```
    }
    int count = 1;
    struct node *stack[100];
    int top = -1;
    stack[++top] = root;
    while (top >= 0) {
        struct node *temp = stack[top--];
        if (temp->right != NULL) {
            stack[++top] = temp->right;
            count++;
        }
        if (temp->left != NULL) {
            stack[++top] = temp->left;
            count++;
        }
    }
    return count;
}

int height(struct node *root) {
    if (root == NULL) {
        return -1;
    }
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}

int main() {
    struct node *root

root = NULL;
insert(&root, 10);
insert(&root, 5);
insert(&root, 15);
insert(&root, 3);
insert(&root, 7);
insert(&root, 13);
insert(&root, 18);
printf("Preorder traversal: ");
preorder(root);
printf("\n");
printf("Postorder traversal: ");
postorder(root);
printf("\n");
printf("Total number of nodes: %d\n", countNodes(root));
printf("Height of the tree: %d\n", height(root));
return 0;
}
```

2. Write a Program to create a Binary Tree and perform following nonrecursive operations on it. a. Levelwise display b. Mirror image c. Display height of a tree.

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

struct node {
   int data;
   struct node *left;
   struct node *right;
};

struct node *createNode(int value) {
   struct node *newNode = malloc(sizeof(struct node));
   newNode->data = value;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

struct node *insertLeft(struct node *root, int value) {
   root->left = createNode(value);
   return root->left;
}

struct node *insertRight(struct node *root, int value) {
   root->right = createNode(value);
   return root->right;
}

void levelwiseDisplay(struct node *root) {
   if(root == NULL)
      return;

   struct node *queue[100];
   int front = -1, rear = -1;
   queue[++rear] = root;

   while(front != rear) {
      struct node *temp = queue[++front];
      printf("%d ", temp->data);

      if(temp->left != NULL)
         queue[++rear] = temp->left;

      if(temp->right != NULL)
         queue[++rear] = temp->right;
```

```
    }
}

void mirrorImage(struct node *root) {
    if(root == NULL)
        return;

    mirrorImage(root->left);
    mirrorImage(root->right);

    struct node *temp = root->left;
    root->left = root->right;
    root->right = temp;
}

int getHeight(struct node *root) {
    if(root == NULL)
        return 0;

    int leftHeight = getHeight(root->left);
    int rightHeight = getHeight(root->right);

    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}

int main() {
    struct node *root = createNode(1);
    insertLeft(root, 2);
    insertRight(root, 3);
    insertLeft(root->left, 4);
    insertRight(root->left, 5);

    printf("Levelwise Display: ");
    levelwiseDisplay(root);
    printf("\n");

    mirrorImage(root);

    printf("Levelwise Display after Mirror Image: ");
    levelwiseDisplay(root);
    printf("\n");

    printf("Height of the Tree: %d\n", getHeight(root));

    return 0;
}
```

3. Write a program to illustrate operations on a BST holding numeric keys.

The menu must include: • Insert • Delete • Find • Show

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

struct node {
    int key;
    struct node *left, *right;
};

struct node *newNode(int item) {
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

void inorderTraversal(struct node *root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->key);
        inorderTraversal(root->right);
    }
}

struct node* insert(struct node* node, int key) {
    if (node == NULL) return newNode(key);

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    return node;
}

struct node * minValueNode(struct node* node) {
    struct node* current = node;
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

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

```
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        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 = minValueNode(root->right);

        root->key = temp->key;

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

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

    if (root->key < key)
       return search(root->right, key);

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

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

    while(1) {
        printf("\n1. Insert\n2. Delete\n3. Find\n4. Show\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter key to insert: ");
```

```
            scanf("%d", &key);
            root = insert(root, key);
            printf("%d inserted successfully.\n", key);
            break;

        case 2:
            printf("Enter key to delete: ");
            scanf("%d", &key);
            root = deleteNode(root, key);
            printf("%d deleted successfully.\n", key);
            break;

        case 3:
            printf("Enter key to find: ");
            scanf("%d", &key);
            if(search(root, key) != NULL)
                printf("%d found in the tree.\n", key);
            else
                printf("%d not found in the tree.\n", key);
            break;

        case 4:
            printf("Tree contents: ");
            inorderTraversal(root);
            printf("\n");
            break;

        case 5:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
            break;
      }
   }

   return 0;
}
```

4. Write a program to illustrate operations on a BST holding numeric keys. The menu must include: • Insert • Mirror Image • Find • Post order (nonrecursive)

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

typedef struct node {
   int key;
```

```
    struct node* left;
    struct node* right;
} node;

node* create_node(int key) {
    node* new_node = (node*) malloc(sizeof(node));
    new_node->key = key;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

node* insert(node* root, int key) {
    if (root == NULL) {
        return create_node(key);
    }
    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }
    return root;
}

node* mirror(node* root) {
    if (root == NULL) {
        return NULL;
    }
    node* new_root = create_node(root->key);
    new_root->left = mirror(root->right);
    new_root->right = mirror(root->left);
    return new_root;
}

node* find(node* root, int key) {
    while (root != NULL && root->key != key) {
        if (key < root->key) {
            root = root->left;
        } else {
            root = root->right;
        }
    }
    return root;
}

void post_order(node* root) {
    if (root == NULL) {
        return;
    }
```

```
    node* stack[100];
    int top = -1;
    node* current = root;
    while (1) {
        while (current != NULL) {
            if (current->right != NULL) {
                stack[++top] = current->right;
            }
            stack[++top] = current;
            current = current->left;
        }
        if (top == -1) {
            break;
        }
        current = stack[top--];
        if (current->right != NULL && top != -1 && current->right == stack[top]) {
            stack[top--] = current;
            current = current->right;
        } else {
            printf("%d ", current->key);
            current = NULL;
        }
    }
}

int main() {
    node* root = NULL;
    int choice, key;
    while (1) {
        printf("\nMenu:\n1. Insert\n2. Mirror Image\n3. Find\n4. Post order\n5. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;
            case 2:
                root = mirror(root);
                printf("Mirror image created.\n");
                break;
            case 3:
                printf("Enter key to find: ");
                scanf("%d", &key);
                if (find(root, key) != NULL) {
                    printf("Key found.\n");
                } else {
                    printf("Key not found.\n");
                }
```

```
            break;
        case 4:
            printf("Post order traversal: ");
            post_order(root);
            printf("\n");
            break;
        case 5:
            exit(0);
        default:
            printf("Invalid choice.\n");
      }
   }
   return 0;
}
```

5. Write a Program to create a Binary Tree and perform following Nonrecursive operations on it. a. Inorder Traversal b. Preorder Traversal c. Display Number of Leaf Nodes d. Mirror Image

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

typedef struct node {
   int data;
   struct node* left;
   struct node* right;
} node;

node* create_node(int data) {
   node* new_node = (node*) malloc(sizeof(node));
   new_node->data = data;
   new_node->left = NULL;
   new_node->right = NULL;
   return new_node;
}

node* insert(node* root, int data) {
   if (root == NULL) {
      return create_node(data);
   }
   if (data < root->data) {
      root->left = insert(root->left, data);
   } else if (data > root->data) {
      root->right = insert(root->right, data);
   }
```

```
    return root;
}

void inorder(node* root) {
    node* stack[100];
    int top = -1;
    node* current = root;
    while (1) {
        while (current != NULL) {
            stack[++top] = current;
            current = current->left;
        }
        if (top == -1) {
            break;
        }
        current = stack[top--];
        printf("%d ", current->data);
        current = current->right;
    }
}

void preorder(node* root) {
    if (root == NULL) {
        return;
    }
    node* stack[100];
    int top = -1;
    stack[++top] = root;
    while (top != -1) {
        node* current = stack[top--];
        printf("%d ", current->data);
        if (current->right != NULL) {
            stack[++top] = current->right;
        }
        if (current->left != NULL) {
            stack[++top] = current->left;
        }
    }
}

int count_leaves(node* root) {
    if (root == NULL) {
        return 0;
    }
```

```
    node* stack[100];
    int top = -1;
    stack[++top] = root;
    int count = 0;
    while (top != -1) {
        node* current = stack[top--];
        if (current->right == NULL && current->left == NULL) {
            count++;
        }
        if (current->right != NULL) {
            stack[++top] = current->right;
        }
        if (current->left != NULL) {
            stack[++top] = current->left;
        }
    }
    return count;
}

node* mirror(node* root) {
    if (root == NULL) {
        return NULL;
    }
    node* new_root = create_node(root->data);
    new_root->left = mirror(root->right);
    new_root->right = mirror(root->left);
    return new_root;
}

int main() {
    node* root = NULL;
    int choice, data;
    while (1) {
        printf("\nMenu:\n1. Insert\n2. Inorder traversal\n3. Preorder traversal\n4. Count leaves\n5.
Mirror image\n6. Exit\nEnter 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("Inorder traversal: ");
```

```
            inorder(root);
            printf("\n");
            break;
        case 3:
            printf("Preorder traversal: ");
            preorder(root);
            printf("\n”);
break;
    case 4:
        printf("Number of leaf nodes: %d\n", count_leaves(root));
        break;
    case 5:
        printf("Mirror image:\n");
        node* mirror_root = mirror(root);
        printf("Inorder traversal: ");
        inorder(mirror_root);
        printf("\n");
        printf("Preorder traversal: ");
        preorder(mirror_root);
        printf("\n");
        break;
    case 6:
        exit(0);
    default:
        printf("Invalid choice\n");
  }
}
return 0;
}
```

6. Write a Program to create a Binary Tree and perform following Nonrecursive operations on it. a. Inorder Traversal b. Preorder Traversal c. Display Height of a tree d. Find Maximum

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

typedef struct node {
  int data;
  struct node* left;
  struct node* right;
} node;

node* create_node(int data) {
```

```
    node* new_node = (node*) malloc(sizeof(node));
    new_node->data = data;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

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

void inorder(node* root) {
    node* stack[100];
    int top = -1;
    node* current = root;
    while (1) {
        while (current != NULL) {
            stack[++top] = current;
            current = current->left;
        }
        if (top == -1) {
            break;
        }
        current = stack[top--];
        printf("%d ", current->data);
        current = current->right;
    }
}

void preorder(node* root) {
    if (root == NULL) {
        return;
    }
    node* stack[100];
    int top = -1;
    stack[++top] = root;
```

```
    while (top != -1) {
        node* current = stack[top--];
        printf("%d ", current->data);
        if (current->right != NULL) {
            stack[++top] = current->right;
        }
        if (current->left != NULL) {
            stack[++top] = current->left;
        }
    }
}

int height(node* root) {
    if (root == NULL) {
        return 0;
    }
    node* stack[100];
    int top = -1;
    stack[++top] = root;
    int max_height = 0;
    int current_height = 0;
    node* last_popped = NULL;
    while (top != -1) {
        node* current = stack[top];
        if (last_popped == NULL || last_popped->left == current || last_popped->right == current) {
            current_height++;
            if (current->left != NULL) {
                stack[++top] = current->left;
            } else if (current->right != NULL) {
                stack[++top] = current->right;
            }
        } else if (current->left == last_popped) {
            if (current->right != NULL) {
                stack[++top] = current->right;
            }
        } else {
            current_height--;
        }
        last_popped = current;
        if (current_height > max_height) {
            max_height = current_height;
        }
        top--;
    }
```

```
    return max_height;
}

int find_max(node* root) {
    node* current = root;
    while (current->right != NULL) {
        current = current->right;
    }
    return current->data;
}

int main() {
    node* root = NULL;
    int choice, data;
    while (1) {
        printf("\nMenu:\n1. Insert\n2. Inorder traversal\n3. Preorder traversal\n4. Display height\n5.
Find maximum\n6. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {

        case 1:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            root = insert(root, data);
            printf("Data inserted successfully!\n");
            break;
        case 2:
            printf("Inorder traversal: ");
            inorder(root);
            printf("\n");
            break;
        case 3:
            printf("Preorder traversal: ");
            preorder(root);
            printf("\n");
            break;
        case 4:
            printf("Height of the tree: %d\n", height(root));
            break;
        case 5:
            printf("Maximum element in the tree: %d\n", find_max(root));
            break;
        case 6:
            exit(0);
```

```
        default:
            printf("Invalid choice\n");
    }
}
return 0;
}
```

7. You have to maintain information for a shop owner. For each of the products sold in his/hers shop the following information is kept: a unique code, a name, a price, amount in stock, date received, expiration date. For keeping track of its stock, the clerk would use a computer program based on a search tree data structure. Write a program to help this person, by implementing the following operations: • Insert an item with all its associated data. • Find an item by its code, and support updating of the item found. • List valid items in lexicographic order of their names.

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

// Define the structure for each product
typedef struct {
    char code[10];
    char name[50];
    float price;
    int stock;
    char date_received[11];
    char expiration_date[11];
} Product;

// Define the structure for the search tree node
typedef struct Node {
    Product product;
    struct Node* left;
    struct Node* right;
} Node;

// Function to create a new node with a given product
Node* createNode(Product product) {
    Node* node = (Node*) malloc(sizeof(Node));
    node->product = product;
    node->left = NULL;
    node->right = NULL;
```

```
    return node;
}

// Function to insert a product into the search tree
Node* insert(Node* root, Product product) {
    if (root == NULL) {
        return createNode(product);
    }
    if (strcmp(product.code, root->product.code) < 0) {
        root->left = insert(root->left, product);
    } else {
        root->right = insert(root->right, product);
    }
    return root;
}

// Function to search for a product by its code
Node* search(Node* root, char code[]) {
    if (root == NULL || strcmp(code, root->product.code) == 0) {
        return root;
    }
    if (strcmp(code, root->product.code) < 0) {
        return search(root->left, code);
    }
    return search(root->right, code);
}

// Function to update the details of a product
void updateProduct(Node* node) {
    printf("Enter the new name: ");
    fgets(node->product.name, 50, stdin);
    printf("Enter the new price: ");
    scanf("%f", &node->product.price);
    printf("Enter the new stock: ");
    scanf("%d", &node->product.stock);
    printf("Enter the new date received: ");
    fgets(node->product.date_received, 11, stdin);
    printf("Enter the new expiration date: ");
    fgets(node->product.expiration_date, 11, stdin);
}

// Function to list all valid products in lexicographic order of their names
void listProducts(Node* root) {
    if (root != NULL) {
```

```
        listProducts(root->left);
        if (strcmp(root->product.expiration_date, "N/A") != 0 &&
strcmp(root->product.expiration_date, "Expired") != 0) {
            printf("%s, %s, %.2f, %d, %s, %s\n", root->product.code, root->product.name,
root->product.price, root->product.stock, root->product.date_received,
root->product.expiration_date);
        }
        listProducts(root->right);
    }
}

int main() {
    Node* root = NULL;
    int choice;
    char code[10];

    do {
        printf("\n1. Insert a product");
        printf("\n2. Find and update a product");
        printf("\n3. List valid products in lexicographic order");
        printf("\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        getchar(); // to consume the newline character

        switch (choice) {
            case 1: {
                Product product;
                printf("\nEnter the product code: ");
                fgets(product.code, 10, stdin);
                printf("Enter the product name: ");
                fgets(product.name, 50, stdin);
                printf("Enter the product price : ");
scanf("%f", &product.price);
printf("Enter the product stock: ");
scanf("%d", &product.stock);
getchar(); // to consume the newline character
printf("Enter the date received (dd/mm/yyyy): ");
fgets(product.date_received, 11, stdin);
printf("Enter the expiration date (dd/mm/yyyy or N/A): ");
fgets(product.expiration_date, 11, stdin);
root = insert(root, product);
printf("\nProduct inserted successfully!\n");
break;
```

```
}
case 2: {
printf("\nEnter the product code to search: ");
fgets(code, 10, stdin);
Node* node = search(root, code);
if (node == NULL) {
printf("Product not found!\n");
} else {
updateProduct(node);
printf("Product details updated successfully!\n");
}
break;
}
case 3: {
printf("\nList of valid products in lexicographic order:\n");
listProducts(root);
break;
}
case 4: {
printf("\nExiting program...\n");
break;
}
default: {
printf("\nInvalid choice, please try again!\n");
}
}
} while (choice != 4);
return 0;}
```

8. You have to maintain information for a shop owner. For each of the products sold in his/hers shop the following information is kept: a unique code, a name, a price, amount in stock, date received, expiration date. For keeping track of its stock, the clerk would use a computer program based on a search tree data structure. Write a program to help this person, by implementing the following operations: • Insert an item with all its associated data. • Find an item by its code, and support updating of the item found. • List valid items in lexicographic order of their names.

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

```
#include <string.h>

// Define a struct for a product
struct Product {
    int code;
    char name[50];
    float price;
    int stock;
    char date_received[11];
    char expiration_date[11];
    struct Product *left;
    struct Product *right;
};

// Define a function to create a new product
struct Product* create_product(int code, char name[], float price, int stock, char date_received[],
char expiration_date[]) {
    struct Product* new_product = (struct Product*) malloc(sizeof(struct Product));
    new_product->code = code;
    strcpy(new_product->name, name);
    new_product->price = price;
    new_product->stock = stock;
    strcpy(new_product->date_received, date_received);
    strcpy(new_product->expiration_date, expiration_date);
    new_product->left = NULL;
    new_product->right = NULL;
    return new_product;
}

// Define a function to insert a product into the binary search tree
struct Product* insert_product(struct Product* root, struct Product* new_product) {
    // If the root is null, set the new product as the root
    if (root == NULL) {
        return new_product;
    }

    // Otherwise, insert the product into the appropriate position in the tree
    if (new_product->code < root->code) {
        root->left = insert_product(root->left, new_product);
    } else if (new_product->code > root->code) {
        root->right = insert_product(root->right, new_product);
    }

    return root;
```

```
}

// Define a function to find a product by its code and update it
void update_product(struct Product* root, int code) {
    // Traverse the tree to find the product with the given code
    struct Product* current = root;
    while (current != NULL) {
        if (code == current->code) {
            // Update the product
            printf("Enter updated information for product with code %d:\n", code);
            printf("Name: ");
            scanf("%s", current->name);
            printf("Price: ");
            scanf("%f", &current->price);
            printf("Stock: ");
            scanf("%d", &current->stock);
            printf("Date received (YYYY-MM-DD): ");
            scanf("%s", current->date_received);
            printf("Expiration date (YYYY-MM-DD): ");
            scanf("%s", current->expiration_date);
            printf("Product updated successfully.\n");
            return;
        } else if (code < current->code) {
            current = current->left;
        } else {
            current = current->right;
        }
    }
    printf("Product with code %d not found.\n", code);
}

// Define a function to list all valid products in lexicographic order of their names
void list_products(struct Product* root) {
    if (root != NULL) {
        list_products(root->left);
        if (strcmp(root->expiration_date, "N/A") != 0 && strcmp(root->expiration_date,
"0000-00-00") != 0 && strcmp(root->expiration_date, "") != 0 && strcmp(root->date_received, "")
!= 0) {
            printf("Code: %d\nName: %s\nPrice: %.2f\nStock: %d\nDate received: %s\nExpiration
date: %s\n\n", root->code, root->name, root->price, root->stock, root->date_received,
root->expiration_date);
        }
        List_products (struct Product* root->right);
    }
```

```
}

int main() {
    // Declare variables for the product data
    int code, stock;
    char name[50], date_received[11], expiration_date[11];
    float price;

    // Declare the root of the binary search tree
    struct Product* root = NULL;

    // Display the menu of operations
    int choice;
    do {
        printf("\nShop Inventory Management\n");
        printf("1. Insert a new product\n");
        printf("2. Update an existing product\n");
        printf("3. List all valid products in lexicographic order of their names\n");
        printf("4. Exit\n");
        printf("Enter your choice (1-4): ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Prompt the user for the product data
                printf("Enter product code: ");
                scanf("%d", &code);
                printf("Enter product name: ");
                scanf("%s", name);
                printf("Enter product price: ");
                scanf("%f", &price);
                printf("Enter product stock: ");
                scanf("%d", &stock);
                printf("Enter date received (YYYY-MM-DD): ");
                scanf("%s", date_received);
                printf("Enter expiration date (YYYY-MM-DD or N/A): ");
                scanf("%s", expiration_date);
                // Create a new product with the given data
                struct Product* new_product = create_product(code, name, price, stock,
date_received, expiration_date);
                // Insert the new product into the binary search tree
                root = insert_product(root, new_product);
                printf("Product added successfully.\n");
                break;
```

```
            case 2:
                // Prompt the user for the product code to update
                printf("Enter product code to update: ");
                scanf("%d", &code);
                // Update the product with the given code
                update_product(root, code);
                break;

            case 3:
                // List all valid products in lexicographic order of their names
                printf("Valid Products:\n\n");
                list_products(root);
                break;

            case 4:
                // Exit the program
                printf("Exiting program...\n");
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

9. Write a Program to create a Binary Search Tree and perform following nonrecursive operations on it. a. Preorder Traversal b. Inorder Traversal c. Display Number of Leaf Nodes d. Mirror Image

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

// Define a struct to represent a node in the binary search tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node with the given data
struct Node* create_node(int data) {
```

```
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

// Function to insert a node into the binary search tree
struct Node* insert_node(struct Node* root, int data) {
    // If the tree is empty, set the new node as the root
    if (root == NULL) {
        return create_node(data);
    }
    // Otherwise, compare the data of the nodes
    if (data < root->data) {
        // If the new node data is less than the root node data,
        // insert the new node into the left subtree
        struct Node* left_child = insert_node(root->left, data);
        root->left = left_child;
    } else if (data > root->data) {
        // If the new node data is greater than the root node data,
        // insert the new node into the right subtree
        struct Node* right_child = insert_node(root->right, data);
        root->right = right_child;
    }
    return root;
}

// Function to perform preorder traversal of the binary search tree nonrecursively
void preorder_traversal(struct Node* root) {
    // Create an empty stack to store the nodes
    struct Node* stack[100];
    int top = -1;
    // Push the root node onto the stack
    stack[++top] = root;
    // Repeat until the stack is empty
    while (top >= 0) {
        // Pop the top node from the stack and visit it
        struct Node* node = stack[top--];
        printf("%d ", node->data);
        // Push the right child onto the stack (if it exists)
        if (node->right != NULL) {
            stack[++top] = node->right;
        }
```

```
        // Push the left child onto the stack (if it exists)
        if (node->left != NULL) {
            stack[++top] = node->left;
        }
    }
    printf("\n");
}

// Function to perform inorder traversal of the binary search tree nonrecursively
void inorder_traversal(struct Node* root) {
    // Create an empty stack to store the nodes
    struct Node* stack[100];
    int top = -1;
    // Set the current node to the root node
    struct Node* current_node = root;
    // Repeat until the current node and the stack are both empty
    while (current_node != NULL || top >= 0) {
        // Push all left child nodes onto the stack
        while (current_node != NULL) {
            stack[++top] = current_node;
            current_node = current_node->left;
        }
        // Pop the top node from the stack and visit it
        current_node = stack[top--];
        printf("%d ", current_node->data);
        // Set the current node to the right child (if it exists)
        current_node = current_node->right;
    }
    printf("\n");
}

// Function to display the number of leaf nodes in the binary search tree
int count_leaf_nodes(struct Node* root) {
    // If the tree is empty, return 0
    if (root == NULL) {
        return 0;
    }
    // If the node is a leaf node, return 1
    if (root->left == NULL && root->right == NULL) {
        return 1;
    }
    // Otherwise, recursively count the leaf nodes in the left and right subtrees
    return count_leaf_nodes(root->left) + count_leaf_nodes(root->right);
}
```

```
// Function to create the mirror image of the binary search tree
void mirror_image(struct Node* root) {
    // If the tree is empty, return
    if (root == NULL) {
        return;
    }
    // Swap the left and right child nodes
    struct Node* temp = root->left;
    root->left = root->right;
    root->right = temp;
    // Recursively create the mirror image of the left and right subtrees
    mirror_image(root->left);
    mirror_image(root->right);
}

// Function to display the binary search tree
void display_tree(struct Node* root) {
    if (root == NULL) {
        return;
    }
    printf("%d ", root->data);
    display_tree(root->left);
    display_tree(root->right);
}

// Main function to test the binary search tree program
int main() {
    // Create the binary search tree
    struct Node* root = NULL;
    root = insert_node(root, 50);
    insert_node(root, 30);
    insert_node(root, 20);
    insert_node(root, 40);
    insert_node(root, 70);
    insert_node(root, 60);
    insert_node(root, 80);
    // Print the binary search tree using preorder traversal
    printf("Preorder Traversal: ");
    preorder_traversal(root);
    // Print the binary search tree using inorder traversal
    printf("Inorder Traversal: ");
    inorder_traversal(root);
    // Display the number of leaf nodes in the binary search tree
```

```
    printf("Number of Leaf Nodes: %d\n", count_leaf_nodes(root));
    // Create the mirror image of the binary search tree
    mirror_image(root);
    // Print the binary search tree using preorder traversal
    printf("Mirror Image Preorder Traversal: ");
    preorder_traversal(root);
    // Display the number of leaf nodes in the binary search tree
    printf("Number of Leaf Nodes: %d\n", count_leaf_nodes(root));
    return 0;
}
```

10. Write a Program to create a Binary Search Tree and perform following nonrecursive operations on it. a. Preorder Traversal b. Postorder Traversal c. Display total Number of Nodes d. Display Leaf nodes.

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

// Define a node structure for the binary search tree
struct node {
    int data;
    struct node *left;
    struct node *right;
};

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

// Function to insert a node into the binary search tree
struct node* insert(struct node* node, int data) {
    // If the tree is empty, return a new node
    if (node == NULL) {
        return(newNode(data));
    }
    // Otherwise, recur down the tree
    if (data < node->data) {
        node->left = insert(node->left, data);
    } else if (data > node->data) {
```

```
        node->right = insert(node->right, data);
    }
    // Return the (unchanged) node pointer
    return(node);
}

// Function to perform a preorder traversal of the binary search tree (nonrecursive)
void preorder(struct node* root) {
    if (root == NULL) {
        return;
    }
    struct node* stack[100];
    int top = -1;
    stack[++top] = root;
    while (top >= 0) {
        struct node* temp = stack[top--];
        printf("%d ", temp->data);
        if (temp->right != NULL) {
            stack[++top] = temp->right;
        }
        if (temp->left != NULL) {
            stack[++top] = temp->left;
        }
    }
}

// Function to perform a postorder traversal of the binary search tree (nonrecursive)
void postorder(struct node* root) {
    if (root == NULL) {
        return;
    }
    struct node* stack1[100];
    struct node* stack2[100];
    int top1 = -1;
    int top2 = -1;
    stack1[++top1] = root;
    while (top1 >= 0) {
        struct node* temp = stack1[top1--];
        stack2[++top2] = temp;
        if (temp->left != NULL) {
            stack1[++top1] = temp->left;
        }
        if (temp->right != NULL) {
            stack1[++top1] = temp->right;
```

```
        }
    }
    while (top2 >= 0) {
        struct node* temp = stack2[top2--];
        printf("%d ", temp->data);
    }
}

// Function to count the total number of nodes in the binary search tree
int countNodes(struct node* root) {
    if (root == NULL) {
        return 0;
    }
    int count = 1;
    struct node* stack[100];
    int top = -1;
    stack[++top] = root;
    while (top >= 0) {
        struct node* temp = stack[top--];
        if (temp->right != NULL) {
            stack[++top] = temp->right;
            count++;
        }
        if (temp->left != NULL) {
            stack[++top] = temp->left;
            count++;
        }
    }
    return count;
}

// Function to display the leaf nodes of the binary search tree
void displayLeaves(struct node* root) {
    if (root == NULL) {
        return;
    }
    struct node* stack[100];
int top = -1;
stack[++top] = root;
while (top >= 0) {
    struct node* temp = stack[top--];
    if (temp->right != NULL) {
        stack[++top] = temp->right;
    }
```

```
    if (temp->left != NULL) {
        stack[++top] = temp->left;
    }
    if (temp->left == NULL && temp->right == NULL) {
        printf("%d ", temp->data);
    }
}
```

```
}

// Main function to test the binary search tree and its operations
int main() {
struct node* root = NULL;
root = insert(root, 50);
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);
printf("Preorder traversal: ");
preorder(root);
printf("\n");
printf("Postorder traversal: ");
postorder(root);
printf("\n");
printf("Total number of nodes: %d\n", countNodes(root));
printf("Leaf nodes: ");
displayLeaves(root);
printf("\n");
return 0;

}
```

11. Write a Program to create a Binary Search Tree and perform deletion of

a node from it. Also display the tree in nonrecursive postorder way.

```
#include <stdio.h>

#include <stdlib.h>
```

```
// Definition of a node in the binary search tree
struct node {
    int data;
    struct node* left;
    struct node* right;
};

// Function to create a new node in the binary search tree
struct node* newNode(int data) {
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to insert a new node into the binary search tree
struct node* insert(struct node* node, int data) {
    if (node == NULL) {
        return newNode(data);
    } else if (data < node->data) {
        node->left = insert(node->left, data);
    } else {
```

```
        node->right = insert(node->right, data);
    }
    return node;
}

// Function to find the minimum value node in the binary search tree
struct node* minValueNode(struct node* node) {
    struct node* current = node;
    while (current->left != NULL) {
        current = current->left;
    }
    return current;
}

// Function to delete a node from the binary search tree
struct node* deleteNode(struct node* root, int key) {
    if (root == NULL) {
        return root;
    } else 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 = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Function to display the binary search tree in nonrecursive postorder way
void postorder(struct node* root) {
    if (root == NULL) {
        return;
    }
    struct node* stack1[100];
    struct node* stack2[100];
```

```
    int top1 = -1;
    int top2 = -1;
    stack1[++top1] = root;
    while (top1 >= 0) {
        struct node* temp = stack1[top1--];
        stack2[++top2] = temp;
        if (temp->left != NULL) {
            stack1[++top1] = temp->left;
        }
        if (temp->right != NULL) {
            stack1[++top1] = temp->right;
        }
    }
    while (top2 >= 0) {
        printf("%d ", stack2[top2--]->data);
    }
}

// Main function to test the binary search tree and its operations
int main() {
    struct node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
```

```
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    printf("Original binary search tree: ");
    postorder(root);
    printf("\n");
    root = deleteNode(root, 20);
printf("Binary search tree after deletion of node with value 20: ");
postorder(root);
printf("\n");
return 0;
}
```

12. Write a Program to create a Binary Search Tree and display it levelwise. Also perform deletion of a node from it.

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

// Definition of a node in the binary search tree
struct node {
    int data;
```

```
    struct node* left;
    struct node* right;
};

// Function to create a new node in the binary search tree
struct node* newNode(int data) {
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to insert a new node into the binary search tree
struct node* insert(struct node* node, int data) {
    if (node == NULL) {
        return newNode(data);
    } else if (data < node->data) {
        node->left = insert(node->left, data);
    } else {
        node->right = insert(node->right, data);
    }
    return node;
```

```
}

// Function to display the binary search tree levelwise
void levelOrder(struct node* root) {
    if (root == NULL) {
        return;
    }
    struct node* queue[100];
    int front = 0;
    int rear = 0;
    queue[rear++] = root;
    while (front < rear) {
        struct node* temp = queue[front++];
        printf("%d ", temp->data);
        if (temp->left != NULL) {
            queue[rear++] = temp->left;
        }
        if (temp->right != NULL) {
            queue[rear++] = temp->right;
        }
    }
}
```

```
// Function to find the minimum value node in the binary search tree
struct node* minValueNode(struct node* node) {
    struct node* current = node;
    while (current->left != NULL) {
        current = current->left;
    }
    return current;
}

// Function to delete a node from the binary search tree
struct node* deleteNode(struct node* root, int key) {
    if (root == NULL) {
        return root;
    } else 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 = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Main function to test the binary search tree and its operations
int main() {
    struct node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    printf("Binary search tree levelwise: ");
    levelOrder(root);
```

```
    printf("\n");
    root = deleteNode(root, 20);
    printf("Binary search tree levelwise after deletion of node with value 20: ");
    levelOrder(root);
    printf("\n");
    return 0;
}
```

13. Write a Program to create a Binary Search Tree and display its mirror image with and without disturbing the original tree. Also display height of a tree using nonrecursion.

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

// Definition of a node in the binary search tree
struct node {
    int data;
    struct node* left;
    struct node* right;
};

// Function to create a new node in the binary search tree
struct node* newNode(int data) {
```

```
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to insert a new node into the binary search tree
struct node* insert(struct node* node, int data) {
    if (node == NULL) {
        return newNode(data);
    } else if (data < node->data) {
        node->left = insert(node->left, data);
    } else {
        node->right = insert(node->right, data);
    }
    return node;
}

// Function to display the binary search tree in inorder way
void inorder(struct node* node) {
    if (node == NULL) {
        return;
```

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

// Function to display the mirror image of the binary search tree without disturbing the original tree
struct node* mirror(struct node* node) {
    if (node == NULL) {
        return node;
    }
    struct node* left = mirror(node->left);
    struct node* right = mirror(node->right);
    node->left = right;
    node->right = left;
    return node;
}

// Function to display the binary search tree levelwise
void levelOrder(struct node* root) {
    if (root == NULL) {
        return;
    }
```

```
    struct node* queue[100];
    int front = 0;
    int rear = 0;
    queue[rear++] = root;
    while (front < rear) {
        struct node* temp = queue[front++];
        printf("%d ", temp->data);
        if (temp->left != NULL) {
            queue[rear++] = temp->left;
        }
        if (temp->right != NULL) {
            queue[rear++] = temp->right;
        }
    }
}

// Function to find the height of the binary search tree using nonrecursion
int height(struct node* root) {
    if (root == NULL) {
        return 0;
    }
    struct node* queue[100];
    int front = 0;
```

```
    int rear = 0;
    int level = 0;
    queue[rear++] = root;
    while (front < rear) {
        int size = rear - front;
        while (size > 0) {
            struct node* temp = queue[front++];
            if (temp->left != NULL) {
                queue[rear++] = temp->left;
            }
            if (temp->right != NULL) {
                queue[rear++] = temp->right;
            }
            size--;
        }
        level++;
    }
    return level;
}

// Main function to test the binary search tree and its operations
int main() {
    struct node* root = NULL;
```

```
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    printf("Original binary search tree inorder: ");
    inorder(root);
    printf("\n");
    printf("Levelwise display of binary search tree: ");
    levelOrder(root);
    printf("\n");
    printf("Height of binary search tree: %d\n", height(root));
    printf("Mirror image of binary search tree without disturbing original tree: \n");
    mirror(root);
    printf("Binary search tree inorder after mirror operation: ");
    inorder(root);
    printf("\n");
    return 0;
}
```

14. You have to maintain information for a shop owner. For each of the

products sold in his/hers shop the following information is kept: a

unique code, a name, a price, amount in stock, date received, expiration date. For keeping track of its stock, the clerk would use a computer program based on a search tree data structure. Write a program to help this person, by implementing the following operations: • Insert an item with all its associated data. • List expired items in Prefix order of their names. • List all items. • Delete an item given by its code. • Delete all expired items.

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

struct item {
    int code;
    char name[50];
    float price;
    int stock;
    char date_received[11];
    char expiration_date[11];
    struct item *left;
    struct item *right;
};
```

```
struct item *root = NULL;



// Helper function to create a new item node

struct item *create_item(int code, char name[], float price, int stock, char date_received[], char
expiration_date[]) {

    struct item *new_item = (struct item *) malloc(sizeof(struct item));

    new_item->code = code;

    strcpy(new_item->name, name);

    new_item->price = price;

    new_item->stock = stock;

    strcpy(new_item->date_received, date_received);

    strcpy(new_item->expiration_date, expiration_date);

    new_item->left = NULL;

    new_item->right = NULL;

    return new_item;

}



// Function to insert an item into the binary search tree

void insert_item(int code, char name[], float price, int stock, char date_received[], char expiration_date[]) {

    if (root == NULL) {

        root = create_item(code, name, price, stock, date_received, expiration_date);

        return;

    }
```

```
    struct item *curr = root;
    while (1) {
        if (code < curr->code) {
            if (curr->left == NULL) {
                curr->left = create_item(code, name, price, stock, date_received, expiration_date);
                return;
            }
            curr = curr->left;
        } else if (code > curr->code) {
            if (curr->right == NULL) {
                curr->right = create_item(code, name, price, stock, date_received, expiration_date);
                return;
            }
            curr = curr->right;
        } else {
            printf("Item with code %d already exists.\n", code);
            return;
        }
    }
}

// Function to list expired items in prefix order of their names
```

```
void list_expired_items(struct item *curr) {
    if (curr == NULL) {
        return;
    }

    list_expired_items(curr->left);
    if (strcmp(curr->expiration_date, "00/00/0000") != 0) {
        printf("%s\n", curr->name);
    }
    list_expired_items(curr->right);
}

// Function to list all items in the binary search tree
void list_all_items(struct item *curr) {
    if (curr == NULL) {
        return;
    }

    list_all_items(curr->left);
    printf("%d %s %.2f %d %s %s\n", curr->code, curr->name, curr->price, curr->stock, curr->date_received,
curr->expiration_date);
    list_all_items(curr->right);
}
```

```
// Helper function to find the item node with the given code
struct item *find_item(int code) {
    struct item *curr = root;
    while (curr != NULL) {
        if (code < curr->code) {
            curr = curr->left;
        } else if (code > curr->code) {
            curr = curr->right;
        } else {
            return curr;
        }
    }
    return NULL;
}

// Function to delete an item with the given code from the binary search tree
void delete_item(int code) {
    struct item *curr = root;
    struct item *parent = NULL;

    while (curr != NULL && curr->code != code) {
        parent = curr;
```

```
    if (code < curr->code) {
        curr = curr->left;
    } else {
        curr = curr->right;
    }
}

if (curr == NULL) {
    printf("Item with code %d does not exist.\n", code);
    return;
}

// Case 1: The item to be deleted has no children
if (curr->left == NULL && curr->right == NULL) {
    if (curr == root) {
        root = NULL;
    } else if (curr == parent->left) {
        parent->left = NULL;
    } else {
        parent->right = NULL;
    }
    free(curr);
}
```

```
// Case 2: The item to be deleted has one child
else if (curr->left == NULL || curr->right == NULL) {
    struct item *child = curr->left == NULL ? curr->right : curr->left;
    if (curr == root) {
        root = child;
    } else if (curr == parent->left) {
        parent->left = child;
    } else {
        parent->right = child;
    }
    free(curr);
}

// Case 3: The item to be deleted has two children
else {
    struct item *successor = curr->right;
    while (successor->left != NULL) {
        successor = successor->left;
    }
    int successor_code = successor->code;
    char successor_name[50];
    float successor_price = successor->price;
```

```
    int successor_stock = successor->stock;
    char successor_date_received[11];
    char successor_expiration_date[11];
    strcpy(successor_name, successor->name);
    strcpy(successor_date_received, successor->date_received);
    strcpy(successor_expiration_date, successor->expiration_date);
    delete_item(successor->code);
    curr->code = successor_code;
    strcpy(curr->name, successor_name);
    curr->price = successor_price;
    curr->stock = successor_stock;
    strcpy(curr->date_received, successor_date_received);
    strcpy(curr->expiration_date, successor_expiration_date);
}
}
// Function to delete all expired items from the binary search tree
void delete_expired_items(struct item *curr, struct item *parent) {
if (curr == NULL) {
return;
}
delete_expired_items(curr->left, curr);
```

```
if (strcmp(curr->expiration_date, "00/00/0000") != 0) {
    if (curr == root) {
        root = curr->right != NULL ? curr->right : curr->left;
    } else if (curr == parent->left) {
        parent->left = curr->right != NULL ? curr->right : curr->left;
    } else {
        parent->right = curr->right != NULL ? curr->right : curr->left;
    }
    free(curr);
}

delete_expired_items(parent->right, parent);

}
int main() {
// Example usage
insert_item(1, "Apple", 1.50, 10, "05/01/2023", "05/07/2023");
insert_item(2, "Banana", 0.99, 20, "05/01/2023", "05/08/2023");
insert_item(3, "Orange", 0.75, 15, "05/02/2023", "05/10/2023");
insert_item(4, "Grapes", 2.99, 5, "05/03/2023", "05/06/2023");
// List all items
printf("All items:\n");
```

```
list_items(root);

// List expired items in Prefix order of their names
printf("Expired items:\n");
list_expired_items(root);

// Delete item with code 2
delete_item(2);

// Delete all expired items
delete_expired_items(root, NULL);

// List all items after deletions
printf("All items after deletions:\n");
list_items(root);

return 0;
}
```

15. Write a program, using trees, to assign the roll nos. to the students of your class as per their previous years result. i.e topper will be roll no. 1.

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

```
// define a structure for a student node in the binary search tree
struct node {
    int roll_number;
    float previous_year_marks;
    struct node *left_child;
    struct node *right_child;
};

// function to create a new node for a student
struct node *create_node(int roll_number, float previous_year_marks) {
    struct node *new_node = (struct node *) malloc(sizeof(struct node));
    new_node->roll_number = roll_number;
    new_node->previous_year_marks = previous_year_marks;
    new_node->left_child = NULL;
    new_node->right_child = NULL;
    return new_node;
}

// function to insert a new student node into the binary search tree
struct node *insert_node(struct node *root, int roll_number, float previous_year_marks) {
    if (root == NULL) {
        return create_node(roll_number, previous_year_marks);
    } else if (previous_year_marks > root->previous_year_marks) {
```

```
        root->right_child = insert_node(root->right_child, roll_number, previous_year_marks);
    } else {
        root->left_child = insert_node(root->left_child, roll_number, previous_year_marks);
    }
    return root;
}

// function to assign roll numbers to students in the binary search tree
void assign_roll_numbers(struct node *root, int *roll_number_ptr) {
    if (root == NULL) {
        return;
    }
    assign_roll_numbers(root->left_child, roll_number_ptr);
    root->roll_number = (*roll_number_ptr)++;
    assign_roll_numbers(root->right_child, roll_number_ptr);
}

int main() {
    struct node *root = NULL;
    int num_students;
    int roll_number = 1;

    // get the number of students from the user
```

```
printf("Enter the number of students: ");
scanf("%d", &num_students);

// get the previous year's marks for each student and insert them into the binary search tree
for (int i = 0; i < num_students; i++) {
    float marks;
    printf("Enter the previous year's marks for student %d: ", i + 1);
    scanf("%f", &marks);
    root = insert_node(root, 0, marks);
}

// assign roll numbers to the students in the binary search tree
assign_roll_numbers(root, &roll_number);

// print the roll numbers for each student in the binary search tree
printf("Roll Numbers:\n");
for (int i = 0; i < num_students; i++) {
    struct node *current_node = root;
    float marks;
    printf("Student %d: ", i + 1);
    scanf("%f", &marks);
    while (current_node->previous_year_marks != marks) {
        if (marks > current_node->previous_year_marks) {
```

```
            current_node = current_node->right_child;
        } else {
            current_node = current_node->left_child;
        }
    }
    printf("%d\n", current_node->roll_number);
  }

  return 0;
}
```

16. Write a program to efficiently search a particular employee record by using Tree data structure. Also sort the data on empid in ascending order.

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

// Define the structure of an employee record
typedef struct EmployeeRecord {
  int id;
  char name[50];
  struct EmployeeRecord* left;
```

```
    struct EmployeeRecord* right;
} EmployeeRecord;

// Function to create a new employee record node
EmployeeRecord* createNode(int id, char* name) {
    EmployeeRecord* node = (EmployeeRecord*) malloc(sizeof(EmployeeRecord));
    node->id = id;
    strcpy(node->name, name);
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to insert a new employee record node into the binary search tree
EmployeeRecord* insertNode(EmployeeRecord* root, int id, char* name) {
    if (root == NULL) {
        return createNode(id, name);
    }
    if (id < root->id) {
        root->left = insertNode(root->left, id, name);
    } else if (id > root->id) {
        root->right = insertNode(root->right, id, name);
    }
```

```
    return root;
}

// Function to search for an employee record by ID
EmployeeRecord* searchNode(EmployeeRecord* root, int id) {
    if (root == NULL || root->id == id) {
        return root;
    }
    if (id < root->id) {
        return searchNode(root->left, id);
    }
    return searchNode(root->right, id);
}

// Function to print the binary search tree in ascending order of ID
void printTree(EmployeeRecord* root) {
    if (root == NULL) {
        return;
    }
    printTree(root->left);
    printf("Employee ID: %d, Name: %s\n", root->id, root->name);
    printTree(root->right);
}
```

```
// Main function to test the binary search tree implementation
int main() {
    EmployeeRecord* root = NULL;

    // Insert some sample employee records
    root = insertNode(root, 102, "John");
    root = insertNode(root, 205, "Jane");
    root = insertNode(root, 311, "Mary");
    root = insertNode(root, 407, "Bob");
    root = insertNode(root, 512, "Alice");

    // Search for an employee record by ID
    int searchId = 311;
    EmployeeRecord* searchResult = searchNode(root, searchId);
    if (searchResult == NULL) {
        printf("Employee record with ID %d not found\n", searchId);
    } else {
        printf("Employee record with ID %d found, Name: %s\n", searchId, searchResult->name);
    }

    // Print the binary search tree in ascending order of ID
    printf("Sorted employee records by ID:\n");
```

```
    printTree(root);



    return 0;
}
```

Advance Trees

1. **Write a Program to create Inorder Threaded Binary Tree and Traverse itin Inorder and Preorder way.**

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

struct Node {
    int data;
    struct Node *left, *right;
    int isThreaded;
};

// Function to create a new node with the given data
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
```

```
    node->left = NULL;
    node->right = NULL;
    node->isThreaded = 0;
    return node;
}

// Function to perform Inorder traversal of the Threaded Binary Tree
void inOrder(struct Node* root) {
    struct Node* curr = root;

    while (curr != NULL) {
        // If left child is not NULL, traverse the left subtree
        if (curr->left != NULL)
            curr = curr->left;

        // Print the current node
        printf("%d ", curr->data);

        // If the node is threaded, move to the inorder successor
        if (curr->isThreaded)
            curr = curr->right;
```

```
        else {
            // Otherwise, move to the right child
            curr = curr->right;

            // While the left child is not NULL, traverse the left subtree
            while (curr != NULL && curr->left != NULL)
                curr = curr->left;
        }
    }
}

// Function to perform Preorder traversal of the Threaded Binary Tree
void preOrder(struct Node* root) {
    struct Node* curr = root;

    while (curr != NULL) {
        // Print the current node
        printf("%d ", curr->data);

        // If left child is not NULL, traverse the left subtree
        if (curr->left != NULL)
```

```
            curr = curr->left;

        // If the node is threaded, move to the inorder successor
        else if (curr->isThreaded)
            curr = curr->right;
        else {
            // Otherwise, move to the right child
            curr = curr->right;

            // While the left child is not NULL, traverse the left subtree
            while (curr != NULL && curr->left != NULL)
                curr = curr->left;
        }
    }
}

// Function to create an Inorder Threaded Binary Tree
void createInorderThreadedBinaryTree(struct Node* root) {
    if (root == NULL)
        return;
```

```
// Initialize previous node as NULL
struct Node* prev = NULL;

// Call the function recursively for the left subtree
if (root->left != NULL)
    createInorderThreadedBinaryTree(root->left);

// If the left child is NULL, set the left child as the previous node
if (root->left == NULL)
    root->left = prev;

// If the previous node is not NULL and its right child is NULL,
// set the right child as the current node and mark it as threaded
if (prev != NULL && prev->right == NULL) {
    prev->right = root;
    prev->isThreaded = 1;
}

// Set the previous node as the current node
prev = root;
```

```
    // Call the function recursively for the right subtree
    if (root->right != NULL)
        createInorderThreadedBinaryTree(root->right);
}

int main() {
    // Create the Binary Tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    // Create the Inorder Threaded Binary Tree
    createInorderThreadedBinaryTree(root);

    // Traverse the Inorder Threaded Binary Tree in Inorder way
    printf("Inorder Traversal:\n");
    inOrder(root);
```

```
    printf("\n");

    // Traverse the Inorder Threaded Binary Tree in Preorder way
    printf("Preorder Traversal:\n");
    preOrder(root);
    printf("\n");

    return 0;
}
```

2.Write a Program to create Inorder Threaded Binary Tree and Traverse it in Inorder and Postorder way.

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

struct Node {
    int data;
    struct Node *left, *right;
    int is_threaded; // flag to indicate if right pointer is a thread or a child
};
```

```
struct Node* createNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    node->is_threaded = 0;
    return node;
}

void createThreadedBST(struct Node* root, struct Node** prev) {
    if (root == NULL) return;
    createThreadedBST(root->left, prev);
    if (*prev != NULL && (*prev)->right == NULL) {
        (*prev)->right = root;
        (*prev)->is_threaded = 1;
    }
    *prev = root;
    createThreadedBST(root->right, prev);
}

struct Node* leftmost(struct Node* root) {
    while (root != NULL && root->left != NULL)
```

```
        root = root->left;
    return root;
}

void inorderTraversal(struct Node* root) {
    struct Node* current = leftmost(root);
    while (current != NULL) {
        printf("%d ", current->data);
        if (current->is_threaded)
            current = current->right;
        else
            current = leftmost(current->right);
    }
}

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

```
int main() {
    struct Node* root = createNode(6);
    root->left = createNode(3);
    root->left->left = createNode(1);
    root->left->right = createNode(4);
    root->right = createNode(8);
    root->right->left = createNode(7);
    root->right->right = createNode(10);

    struct Node* prev = NULL;
    createThreadedBST(root, &prev);

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

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

```
    return 0;
}
```

3.Write a Program to implement AVL tree and perform different rotations on it.

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;
};

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

int getheight(struct Node* node) {
```

```
    if (node == NULL)

        return 0;

    return node->height;

}



struct Node* newNode(int data) {

    struct Node* node = (struct Node*)malloc(sizeof(struct Node));

    node->data = data;

    node->left = NULL;

    node->right = NULL;

    node->height = 1;

    return (node);

}



struct Node* rightRotate(struct Node* y) {

    struct Node* x = y->left;

    struct Node* T2 = x->right;

    x->right = y;

    y->left = T2;

    y->height = max(getheight(y->left), getheight(y->right)) + 1;

    x->height = max(getheight(x->left), getheight(x->right)) + 1;
```

```
    return x;
}

struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(getheight(x->left), getheight(x->right)) + 1;
    y->height = max(getheight(y->left), getheight(y->right)) + 1;
    return y;
}

int getBalance(struct Node* node) {
    if (node == NULL)
        return 0;
    return getheight(node->left) - getheight(node->right);
}

struct Node* insert(struct Node* node, int data) {
    if (node == NULL)
```

```
        return (newNode(data));
    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;
    node->height = 1 + max(getheight(node->left), getheight(node->right));
    int balance = getBalance(node);
    //left-left
    if (balance > 1 && data < node->left->data)
        return rightRotate(node);
        //right-right
    if (balance < -1 && data > node->right->data)
        return leftRotate(node);


    //left-right
    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```

```
    //right -left
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

void printLevel(struct Node* root, int level) {
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1) {
        printLevel(root->left, level - 1);
        printLevel(root->right, level - 1);
    }
}

void levelOrder(struct Node* root) {
    int h = getheight(root);
```

```
    int i;
    for (i = 1; i <= h; i++) {
        printLevel(root, i);
        printf("\n");
    }
}

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

void postOrder(struct Node *root)
{
    if(root != NULL)
    {
```

```
        postOrder(root->left);

        postOrder(root->right);

        printf("%d ", root->key);

    }

}



void inOrder(struct Node *root)

{

    if(root != NULL)

    {

        inOrder(root->left);

        printf("%d ", root->key);

        inOrder(root->right);

    }

}




int main() {

    struct Node* root = NULL;

    root = insert(root, 10);

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

```
    root = insert(root, 30);

    root = insert(root, 40);

    root = insert(root, 50);

    root = insert(root, 60);




    printf("Avl Tree Pre-order Traversal: \n");

    preorder(root);



    printf("\n\nLevel wise dsiplay \n");

    levelOrder(root);



    return 0;

}
```

****

#include <stdio.h>

#include <stdlib.h>

```
struct Node {
    int key;
    int height;
    struct Node* left;
    struct Node* right;
};

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

int height(struct Node* node) {
    if (node == NULL) return 0;
    return node->height;
}

int getBalance(struct Node* node) {
    if (node == NULL) return 0;
    return height(node->left) - height(node->right);
}
```

```
struct Node* createNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->height = 1;
    node->left = NULL;
    node->right = NULL;
    return node;
}

struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* t2 = x->right;
    x->right = y;
    y->left = t2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
```

```
    struct Node* t2 = y->left;
    y->left = x;
    x->right = t2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}

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

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
```

```
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void printTree(struct Node* root, int space) {
    if (root == NULL) return;
    space += 10;
    printTree(root->right, space);
    printf("\n");
```

```
    for (int i = 10; i < space; i++)
        printf(" ");
    printf("%d\n", root->key);
    printTree(root->left, space);
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("AVL tree:\n");
    printTree(root, 0);

    return 0;}
```

4.Write a Program to create Inorder Threaded Binary Tree and Traverse it in Postorder and Preorder way.

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

// Definition of threaded binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int isThreaded;
};

// Helper function to create a new node
struct Node* createNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->isThreaded = 0;
    return node;
}
```

```
// Function to create inorder threaded binary tree
struct Node* createInorderThreadedTree(int inorder[], int n) {
    if (n == 0) {
        return NULL;
    }

    // Create root node
    struct Node* root = createNode(inorder[0]);

    // Initialize previous node and set its right pointer to root
    struct Node* prev = NULL;
    if (n > 1) {
        prev = createNode(inorder[1]);
        prev->isThreaded = 1;
        prev->right = root;
    }

    // Traverse the inorder array and create threaded binary tree
    int i = 2;
    struct Node* curr = root;
    while (i < n) {
```

```
    if (inorder[i] <= curr->data) {
        // Insert node in left subtree
        struct Node* node = createNode(inorder[i]);
        curr->left = node;
        node->isThreaded = 1;
        node->right = curr;
        curr = node;
        i++;
    } else {
        // Insert node in right subtree
        if (curr->isThreaded) {
            // Update prev's right pointer
            prev->right = createNode(inorder[i]);
            prev->right->isThreaded = 1;
            prev->right->right = curr->right;
            curr->right = prev->right;
        } else {
            curr->right = createNode(inorder[i]);
        }
        prev = curr;
        curr = curr->right;
```

```
            i++;
        }
    }
    return root;
}

// Function to traverse inorder threaded binary tree in preorder way
void preorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    printf("%d ", root->data);
    if (!root->isThreaded) {
        preorderTraversal(root->left);
    }
    preorderTraversal(root->right);
}

// Function to traverse inorder threaded binary tree in postorder way
void postorderTraversal(struct Node* root) {
```

```
    if (root == NULL) {
        return;
    }

    if (!root->isThreaded) {
        postorderTraversal(root->left);
    }
    postorderTraversal(root->right);
    printf("%d ", root->data);
}

// Driver code
int main() {
    int inorder[] = {4, 2, 5, 1, 6, 3, 7};
    int n = sizeof(inorder) / sizeof(inorder[0]);

    struct Node* root = createInorderThreadedTree(inorder, n);

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

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

    return 0;
}
```

GRAPHS

1. Write a Program to accept a graph from user and represent it with Adjacency Matrix and perform BFS and DFS traversals on it.

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

#define MAX_VERTICES 100

int adj_matrix[MAX_VERTICES][MAX_VERTICES];
int visited[MAX_VERTICES];

void addEdge(int u, int v) {
   adj_matrix[u][v] = 1;
   adj_matrix[v][u] = 1;
}

void bfs(int s, int n) {
   int queue[MAX_VERTICES];
   int front = 0, rear = 0;
   int u, v;
   printf("BFS Traversal: ");
   visited[s] = 1;
   queue[rear++] = s;
   while(front != rear) {
      u = queue[front++];
      printf("%d ", u);
      for(v = 0; v < n; v++) {
         if(adj_matrix[u][v] == 1 && visited[v] == 0) {
            visited[v] = 1;
            queue[rear++] = v;
         }
```

```
        }
    }
    printf("\n");
}

void dfs(int u, int n) {
    int v;
    printf("%d ", u);
    visited[u] = 1;
    for(v = 0; v < n; v++) {
        if(adj_matrix[u][v] == 1 && visited[v] == 0) {
            dfs(v, n);
        }
    }
}

int main() {
    int n, e, i, u, v;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    printf("Enter the number of edges: ");
    scanf("%d", &e);
    for(i = 0; i < e; i++) {
        printf("Enter edge %d: ", i+1);
        scanf("%d %d", &u, &v);
        addEdge(u, v);
    }
    printf("\nAdjacency Matrix:\n");
    for(i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            printf("%d ", adj_matrix[i][j]);
        }
        printf("\n");
    }
    for(i = 0; i < n; i++) {
        visited[i] = 0;
    }
    printf("\nEnter the starting vertex for BFS: ");
    scanf("%d", &u);
    bfs(u, n);
    for(i = 0; i < n; i++) {
        visited[i] = 0;
    }
    printf("\nEnter the starting vertex for DFS: ");
```

```
    scanf("%d", &u);
    printf("DFS Traversal: ");
    dfs(u, n);
    printf("\n");
    return 0;
}
```

2.Write a Program to implement Prim's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency List to represent a Graph.

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

#define INF 9999999 // Define infinity as a very large number

// Create a structure for a node
struct node {
    int vertex;
    int weight;
    struct node* next;
};

// Create a structure for an adjacency list
struct adj_list {
    struct node* head;
};

// Create a structure for a graph
struct graph {
    int vertices;
    struct adj_list* array;
};

// Create a new node
struct node* new_node(int dest, int weight) {
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->vertex = dest;
    new_node->weight = weight;
    new_node->next = NULL;
    return new_node;
}

// Create a new graph with V vertices
struct graph* create_graph(int vertices) {
    struct graph* graph = (struct graph*) malloc(sizeof(struct graph));
```

```
    graph->vertices = vertices;
    graph->array = (struct adj_list*) malloc(vertices * sizeof(struct adj_list));
    for (int i = 0; i < vertices; ++i) {
        graph->array[i].head = NULL;
    }
    return graph;
}

// Add an edge to the graph
void add_edge(struct graph* graph, int src, int dest, int weight) {
    struct node* new_node = new_node(dest, weight);
    new_node->next = graph->array[src].head;
    graph->array[src].head = new_node;
    new_node = new_node(src, weight);
    new_node->next = graph->array[dest].head;
    graph->array[dest].head = new_node;
}

// Find the vertex with minimum key value
int min_key(int key[], int mst_set[], int vertices) {
    int min = INF, min_index;
    for (int v = 0; v < vertices; ++v) {
        if (mst_set[v] == 0 && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

// Print the minimum spanning tree
void print_mst(int parent[], int graph[][100], int vertices) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; ++i) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}

// Implement Prim's algorithm
void prim_mst(struct graph* graph) {
    int vertices = graph->vertices;
    int key[vertices], parent[vertices], mst_set[vertices];
    for (int i = 0; i < vertices; ++i) {
        key[i] = INF;
        mst_set[i] = 0;
    }
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < vertices - 1; ++count) {
```

```
        int u = min_key(key, mst_set, vertices);
        mst_set[u] = 1;
        struct node* ptr = graph->array[u].head;
        while (ptr != NULL) {
            int v = ptr->vertex;
            int weight = ptr->weight;
            if (mst_set[v] == 0 && weight < key[v]) {
                parent[v] = u;
                key[v] = weight;
            }
            ptr = ptr->next;
        }
    }
    print_mst(parent, graph, vertices

// function to implement Prim's algorithm
void primMST(int graph[V][V]) {
    int parent[V]; // to store the parent of each vertex in MST
    int key[V]; // to store the key value of each vertex
    bool mstSet[V]; // to keep track of vertices already included in MST

    // initialize key values and mstSet
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    // set the key value of the first vertex as 0 and parent as -1
    key[0] = 0;
    parent[0] = -1;

    // iterate over all vertices to construct MST
    for (int count = 0; count < V-1; count++) {
        // find the vertex with minimum key value from the set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // add the selected vertex to MST
        mstSet[u] = true;

        // update the key values and parent of the adjacent vertices of the selected vertex
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // print the MST
```

```
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// driver function
int main() {
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

    primMST(graph);

    return 0;
}
```

3.Write a Program to implement Kruskal's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency List to represent a Graph.

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

#define MAX_EDGES 1000
#define MAX_VERTICES 100

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge edges[MAX_EDGES];
};

struct Subset {
    int parent;
    int rank;
};

struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    return graph;
}
```

```
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int compare(const void* a, const void* b) {
    struct Edge* edgeA = (struct Edge*) a;
    struct Edge* edgeB = (struct Edge*) b;
    return edgeA->weight - edgeB->weight;
}

void kruskalMST(struct Graph* graph) {
    struct Subset* subsets = (struct Subset*) malloc(graph->V * sizeof(struct Subset));
    for (int i = 0; i < graph->V; i++) {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compare);

    struct Edge result[graph->V];
    int e = 0;
    int i = 0;
    while (e < graph->V - 1 && i < graph->E) {
        struct Edge next_edge = graph->edges[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
```

```
    }

    printf("Minimum Spanning Tree:\n");
    for (int i = 0; i < e; i++) {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
    }
    return;
}

int main() {
    int V, E;
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &V);
    printf("Enter the number of edges in the graph: ");
    scanf("%d", &E);

    struct Graph* graph = createGraph(V, E);

    printf("Enter the edges in the format [source] [destination] [weight]:\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &graph->edges[i].src, &graph->edges[i].dest, &graph->edges[i].weight);
    }

    kruskalMST(graph);

    return 0;
}
```

4.Write a Program to implement Dijkstra's algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency List to represent a graph.

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

#define MAX_NODES 1000

// node structure for adjacency list
struct AdjListNode {
    int dest;
    int weight;
    struct AdjListNode* next;
};

// graph structure
struct Graph {
    struct AdjListNode* array[MAX_NODES];
```

```
    int num_nodes;
};

// create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight) {
    struct AdjListNode* newNode = (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// create a new graph with n nodes
struct Graph* createGraph(int n) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->num_nodes = n;
    int i;
    for (i = 0; i < n; i++)
        graph->array[i] = NULL;
    return graph;
}

// add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest, int weight) {
    // add edge from src to dest
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src];
    graph->array[src] = newNode;

    // add edge from dest to src
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest];
    graph->array[dest] = newNode;
}

// find the node with the minimum distance
int minDistance(int dist[], int visited[], int n) {
    int minDist = INT_MAX, minIndex = -1;
    int i;
    for (i = 0; i < n; i++) {
        if (!visited[i] && dist[i] <= minDist) {
            minDist = dist[i];
            minIndex = i;
        }
    }
    return minIndex;
}

// print the shortest distance from src to dest
void printShortestPath(int dist[], int n, int src, int dest) {
    printf("Shortest distance from node %d to node %d is %d\n", src, dest, dist[dest]);
```

```
    // print the path from src to dest
    printf("Path: ");
    int i;
    int path[MAX_NODES], count = 0;
    path[count++] = dest;
    int curr = dest;
    while (curr != src) {
        for (i = 0; i < n; i++) {
            struct AdjListNode* node = graph->array[curr];
            while (node != NULL) {
                if (dist[i] + node->weight == dist[curr]) {
                    path[count++] = i;
                    curr = i;
                    break;
                }
                node = node->next;
            }
        }
    }
    for (i = count - 1; i >= 0; i--)
        printf("%d ", path[i]);
    printf("\n");
}

// calculate the shortest path using Dijkstra's algorithm
void dijkstra(struct Graph* graph, int src, int dest) {
    int dist[MAX_NODES], visited[MAX_NODES];
    int i;
    for (i = 0; i < graph->num_nodes; i++) {
        dist[i] = INT_MAX;
        visited[i] = 0;
    }
    dist[src] = 0;
    for (i = 0); i < graph->num_nodes - 1; i++) {
    int u = minDistance(dist, visited, graph->num_nodes);
    visited[u] = 1;
    struct AdjListNode* node = graph->array[u];
    while (node != NULL) {
        int v = node->dest;
        if (!visited[v] && dist[u] != INT_MAX && dist[u] + node->weight < dist[v]) {
            dist[v] = dist[u] + node->weight;
        }
        node = node->next;
    }
}

printShortestPath(dist, graph->num_nodes, src, dest);


}
```

```
// main function to test the implementation
int main() {
int n, e;
printf("Enter the number of nodes in the graph: ");
scanf("%d", &n);
struct Graph* graph = createGraph(n);
printf("Enter the number of edges in the graph: ");
scanf("%d", &e);
printf("Enter the edges and their weights: \n");
int i;
for (i = 0; i < e; i++) {
int src, dest, weight;
scanf("%d %d %d", &src, &dest, &weight);
addEdge(graph, src, dest, weight);
}
int src, dest;
printf("Enter the source node: ");
scanf("%d", &src);
printf("Enter the destination node: ");
scanf("%d", &dest);
dijkstra(graph, src, dest);
return 0;

}
```

5.Write a Program to accept a graph from user and represent it with

Adjacency List and perform BFS and DFS traversals on it.

```
#include<stdio.h>

#include<stdlib.h>



#define MAX_NODES 100



struct node {

   int vertex;

   struct node* next;
```

```
};

struct node* create_node(int v);
void add_edge(struct node* adjacency_list[], int src, int dest);
void print_graph(struct node* adjacency_list[], int n);
void bfs(struct node* adjacency_list[], int start_node, int n);
void dfs(struct node* adjacency_list[], int start_node, int visited[]);

int main() {
    int n, i, j, src, dest, start_node;
    struct node* adjacency_list[MAX_NODES] = { NULL };

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

    printf("Enter the edges as (source, destination) pairs:\n");
    while (1) {
        printf("Source: ");
        scanf("%d", &src);
        printf("Destination: ");
        scanf("%d", &dest);

        if (src == -1 && dest == -1) {
            break;
```

```
    }

    if (src < 0 || src >= n || dest < 0 || dest >= n) {
        printf("Invalid edge! Try again.\n");
        continue;
    }

    add_edge(adjacency_list, src, dest);
}

printf("Enter the starting node: ");
scanf("%d", &start_node);

printf("Adjacency List:\n");
print_graph(adjacency_list, n);

printf("BFS Traversal:\n");
bfs(adjacency_list, start_node, n);

printf("DFS Traversal:\n");
int visited[MAX_NODES] = { 0 };
dfs(adjacency_list, start_node, visited);

return 0;
```

```
}

struct node* create_node(int v) {
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
    new_node->vertex = v;
    new_node->next = NULL;
    return new_node;
}

void add_edge(struct node* adjacency_list[], int src, int dest) {
    struct node* new_node = create_node(dest);
    new_node->next = adjacency_list[src];
    adjacency_list[src] = new_node;
}

void print_graph(struct node* adjacency_list[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        struct node* current_node = adjacency_list[i];
        printf("%d: ", i);
        while (current_node != NULL) {
            printf("%d ", current_node->vertex);
            current_node = current_node->next;
        }
```

```
        printf("\n");
    }
}

void bfs(struct node* adjacency_list[], int start_node, int n) {
    int visited[MAX_NODES] = { 0 };
    int queue[MAX_NODES], front = 0, rear = 0;
    visited[start_node] = 1;
    queue[rear++] = start_node;

    while (front != rear) {
        int current_node = queue[front++];
        printf("%d ", current_node);

        struct node* temp = adjacency_list[current_node];
        while (temp != NULL) {
            int adj_node = temp->vertex;
            if (visited[adj_node] == 0) {
                visited[adj_node] = 1;
                queue[rear++] = adj_node;
            }
            temp = temp->next;
        }
    }
```

```
    printf("\n");
}

void dfs(struct node* adjacency_list[], int start_node,

void dfs(struct node* adjacency_list[], int start_node, int visited[]) {
    visited[start_node] = 1;
    printf("%d ", start_node);

    struct node* temp = adjacency_list[start_node];
    while (temp != NULL) {
        int adj_node = temp->vertex;
        if (visited[adj_node] == 0) {
            dfs(adjacency_list, adj_node, visited);
        }
        temp = temp->next;
    }
}
```

6.Write a Program to implement Kruskal's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency Matrix to represent a graph.

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

```
#define MAX 1000

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge edge[MAX];
};

struct Subset {
    int parent, rank;
};

struct Graph createGraph(int V, int E) {
    struct Graph graph;
    graph.V = V;
    graph.E = E;
    return graph;
}

int find(struct Subset subsets[], int i) {
```

```
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int compare(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}
```

```
void KruskalMST(struct Graph graph) {
  int V = graph.V;
  struct Edge result[V];
  int e = 0;
  int i = 0;
  qsort(graph.edge, graph.E, sizeof(graph.edge[0]), compare);
  struct Subset subsets[V];
  for (int v = 0; v < V; v++) {
    subsets[v].parent = v;
    subsets[v].rank = 0;
  }
  while (e < V - 1 && i < graph.E) {
    struct Edge next_edge = graph.edge[i++];
    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);
    if (x != y) {
      result[e++] = next_edge;
      Union(subsets, x, y);
    }
  }
  printf("Edges in minimum spanning tree:\n");
  int minimumCost = 0;
  for (i = 0; i < e; ++i) {
    printf("%d - %d : %d\n", result[i].src, result[i].dest, result[i].weight);
```

```
        minimumCost += result[i].weight;
    }
    printf("Minimum Cost = %d\n", minimumCost);
}

int main() {
    int V, E;
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);
    struct Graph graph = createGraph(V, E);
    printf("Enter edges and their weights:\n");
    for (int i = 0; i < E; ++i) {
        scanf("%d %d %d", &graph.edge[i].src, &graph.edge[i].dest, &graph.edge[i].weight);
    }
    KruskalMST(graph);
    return 0;
}
```

7.Write a Program to implement Dijkstra's algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency Matrix to represent a graph.

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

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

#define MAX 1000

int minDistance(int dist[], int visited[], int V) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (visited[v] == 0 && dist[v] <= min)
         min = dist[v], min_index = v;
   return min_index;
}

void printSolution(int dist[], int V) {
   printf("Vertex \t Distance from Source\n");
   for (int i = 0; i < V; i++)
      printf("%d \t %d\n", i, dist[i]);
}

void dijkstra(int graph[MAX][MAX], int src, int dest, int V) {
   int dist[V];
   int visited[V];
   for (int i = 0; i < V; i++) {
      dist[i] = INT_MAX;
```

```
    visited[i] = 0;
  }
  dist[src] = 0;
  for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, visited, V);
    visited[u] = 1;
    for (int v = 0; v < V; v++)
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
  }
  printf("Shortest distance between node %d and node %d: %d\n", src, dest, dist[dest]);
}

int main() {
  int V, E;
  printf("Enter number of vertices: ");
  scanf("%d", &V);
  printf("Enter number of edges: ");
  scanf("%d", &E);
  int graph[MAX][MAX];
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      graph[i][j] = 0;
  printf("Enter edges and their weights:\n");
```

```
    for (int i = 0; i < E; ++i) {
        int src, dest, weight;
        scanf("%d %d %d", &src, &dest, &weight);
        graph[src][dest] = weight;
    }
    int src, dest;
    printf("Enter source node: ");
    scanf("%d", &src);
    printf("Enter destination node: ");
    scanf("%d", &dest);
    dijkstra(graph, src, dest, V);
    return 0;
}
```

8. Write a Program to implement Prim's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency List to represent a graph.

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

#define MAX 1000

struct AdjListNode {
```

```
    int dest;
    int weight;
    struct AdjListNode* next;
};

struct AdjList {
    struct AdjListNode* head;
};

struct Graph {
    int V;
    struct AdjList* array;
};

struct AdjListNode* newAdjListNode(int dest, int weight) {
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
```

```
    graph->V = V;
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest, int weight) {
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

int minKey(int key[], int visited[], int V) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (visited[v] == 0 && key[v] < min)
            min = key[v], min_index = v;
    return min_index;
}
```

```
void printMST(int parent[], int graph[MAX][MAX], int V) {
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}


void primMST(struct Graph* graph) {
   int parent[MAX];
   int key[MAX];
   int visited[MAX];
   for (int i = 0; i < graph->V; i++) {
      key[i] = INT_MAX;
      visited[i] = 0;
   }
   key[0] = 0;
   parent[0] = -1;
   for (int count = 0; count < graph->V - 1; count++) {
      int u = minKey(key, visited, graph->V);
      visited[u] = 1;
      struct AdjListNode* pCrawl = graph->array[u].head;
      while (pCrawl != NULL) {
         int v = pCrawl->dest;
         int weight = pCrawl->weight;
         if (visited[v] == 0 && weight < key[v]) {
```

```
            parent[v] = u;
            key[v] = weight;
          }
          pCrawl = pCrawl->next;
        }
    }
    printMST(parent, graph, graph->V);
}

int main() {
    int V, E;
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    struct Graph* graph = createGraph(V);
    printf("Enter number of edges: ");
    scanf("%d", &E);
    printf("Enter edges and their weights:\n");
    for (int i = 0; i < E; ++i) {
        int src, dest, weight;
    scanf("%d %d %d", &src, &dest, &weight);
    addEdge(graph, src, dest, weight);
}
primMST(graph);
return 0;
```

```
}
```

9. Write a Program to implement Kruskal's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency List to represent a Graph.

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

#define MAX 1000

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
```

```
    graph->E = E;
    graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

int find(int parent[], int i) {
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

void Union(int parent[], int x, int y) {
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

int myComp(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

void kruskalMST(struct Graph* graph) {
```

```
    int V = graph->V;
    struct Edge result[V];
    int e = 0;
    int i = 0;
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);
    int parent[V];
    memset(parent, -1, sizeof(parent));
    while (e < V - 1 && i < graph->E) {
        struct Edge next_edge = graph->edge[i++];
        int x = find(parent, next_edge.src);
        int y = find(parent, next_edge.dest);
        if (x != y) {
            result[e++] = next_edge;
            Union(parent, x, y);
        }
    }
    printf("Edge \tWeight\n");
    for (i = 0; i < e; ++i)
        printf("%d - %d \t%d \n", result[i].src, result[i].dest, result[i].weight);
}

int main() {
    int V, E;
    printf("Enter number of vertices: ");
```

```
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);
    struct Graph* graph = createGraph(V, E);
    printf("Enter edges and their weights:\n");
    for (int i = 0; i < E; ++i) {
        int src, dest, weight;
        scanf("%d %d %d", &src, &dest, &weight);
        graph->edge[i].src = src;
        graph->edge[i].dest = dest;
        graph->edge[i].weight = weight;
    }
    kruskalMST(graph);
    return 0;
}
```

10. Write a Program to implement Dijkstra's algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency List to represent a graph.

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

#define MAX 1000
```

```
struct AdjListNode {
    int dest;
    int weight;
    struct AdjListNode* next;
};

struct AdjList {
    struct AdjListNode* head;
};

struct Graph {
    int V;
    struct AdjList* array;
};

struct MinHeapNode {
    int v;
    int dist;
};

struct MinHeap {
    int size;
    int capacity;
```

```
    int* pos;
    struct MinHeapNode** array;
};

struct AdjListNode* newAdjListNode(int dest, int weight) {
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest, int weight) {
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
```

```
    graph->array[src].head = newNode;
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

struct MinHeapNode* newMinHeapNode(int v, int dist) {
    struct MinHeapNode* minHeapNode = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

struct MinHeap* createMinHeap(int capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->pos = (int*)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
```

```
    *a = *b;
    *b = t;
}

void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;
    if (left < minHeap->size && minHeap->array[left]->dist < minHeap->array[smallest]->dist)
        smallest = left;
    if (right < minHeap->size && minHeap->array[right]->dist < minHeap->array[smallest]->dist)
        smallest = right;
    if (smallest != idx) {
        struct MinHeapNode* smallestNode = minHeap->array[smallest];
        struct MinHeapNode* idxNode = minHeap->array[idx];
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
}
}
int isEmpty(struct MinHeap* minHeap) {
return minHeap->size == 0;
```

```
}
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
if (isEmpty(minHeap))
return NULL;
struct MinHeapNode* root = minHeap->array[0];
struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
minHeap->array[0] = lastNode;
minHeap->pos[root->v] = minHeap->size - 1;
minHeap->pos[lastNode->v] = 0;
--minHeap->size;
minHeapify(minHeap, 0);
return root;
}
void decreaseKey(struct MinHeap* minHeap, int v, int dist) {
int i = minHeap->pos[v];
minHeap->array[i]->dist = dist;
while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist) {
minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);
i = (i - 1) / 2;
}
}
int isInMinHeap(struct MinHeap* minHeap, int v) {
```

```
if (minHeap->pos[v] < minHeap->size)

return 1;

return 0;

}

void printPath(int parent[], int j) {

if (parent[j] == -1)

return;

printPath(parent, parent[j]);

printf("%d ", j);

}

void printSolution(int dist[], int n, int parent[], int src, int dest) {

printf("Shortest path from %d to %d is: ", src, dest);

printPath(parent, dest);

printf("%d ", src);

printf("\nShortest distance is: %d\n", dist[dest]);

}

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

int V = graph->V;

int dist[V];

int parent[V];

struct MinHeap* minHeap = createMinHeap(V);

for (int v = 0; v < V; ++v) {

dist[v] = INT_MAX;

parent[v] = -1;
```

```
minHeap->array[v] = newMinHeapNode(v, dist[v]);
minHeap->pos[v] = v;
}
minHeap->array[src] = newMinHeapNode(src, dist[src]);
minHeap->pos[src] = src;
dist[src] = 0;
decreaseKey(minHeap, src, dist[src]);
minHeap->size = V;
while (!isEmpty(minHeap)) {
struct MinHeapNode* minHeapNode = extractMin(minHeap);
int u = minHeapNode->v;
struct AdjListNode* adjListNode = graph->array[u].head;
while (adjListNode != NULL) {
int v = adjListNode->dest;
if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX && adjListNode->weight + dist[u] < dist[v]) {
dist[v] = dist[u] + adjListNode->weight;
parent[v] = u;
decreaseKey(minHeap, v, dist[v]);
}
```

11.Write a Program to implement Prim's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency Matrix to represent a graph.

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

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

#define V 5 // Maximum number of vertices in the graph

int findMinKey(int key[], int mstSet[]) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}

void primMST(int graph[V][V]) {
```

```
int parent[V];

int key[V];

int mstSet[V];



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

   key[i] = INT_MAX;

   mstSet[i] = 0;

}



key[0] = 0;

parent[0] = -1;



for (int i = 0; i < V - 1; i++) {

   int u = findMinKey(key, mstSet);

   mstSet[u] = 1;



   for (int v = 0; v < V; v++) {

      if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

         parent[v] = u;

         key[v] = graph[u][v];

      }

   }

}
```

```
    printMST(parent, graph);
}



int main() {
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};
    primMST(graph);
    return 0;
}
```

SEARCHING SORTING

WAP to implement Bubble sort and Quick Sort on 1D array of Student structure (contains student_name, student_roll_no, total_marks), with key as student_roll_no. And count the number of swap performed.

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

struct Student {
    char student_name[50];
    int student_roll_no;
    int total_marks;
};

// Function to swap two Student structures
void swap(struct Student *a, struct Student *b) {
    struct Student temp = *a;
    *a = *b;
    *b = temp;
}

// Bubble Sort implementation
int bubbleSort(struct Student arr[], int n) {
    int swapCount = 0;
    int i, j;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j].student_roll_no > arr[j+1].student_roll_no) {
                swap(&arr[j], &arr[j+1]);
                swapCount++;
            }
        }
    }
    return swapCount;
```

```
}

// Quick Sort implementation
int partition(struct Student arr[], int low, int high, int *swapCount) {
    int pivot = arr[high].student_roll_no;
    int i = low - 1;
    int j;
    for (j = low; j <= high-1; j++) {
        if (arr[j].student_roll_no < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
            (*swapCount)++;
        }
    }
    swap(&arr[i+1], &arr[high]);
    (*swapCount)++;
    return (i + 1);
}

void quickSort(struct Student arr[], int low, int high, int *swapCount) {
    if (low < high) {
        int p = partition(arr, low, high, swapCount);
        quickSort(arr, low, p-1, swapCount);
        quickSort(arr, p+1, high, swapCount);
    }
}

// Function to print the Student array
void printArray(struct Student arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("Student Name: %s, Roll No.: %d, Total Marks: %d\n",
arr[i].student_name, arr[i].student_roll_no, arr[i].total_marks);
    }
}
```

```
// Driver function
int main() {
    struct Student arr[] = {
        {"John", 3, 85},
        {"Mary", 2, 90},
        {"Peter", 4, 80},
        {"Alice", 1, 95}
    };
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original Array:\n");
    printArray(arr, n);

    // Bubble Sort
    int bubbleSwapCount = bubbleSort(arr, n);
    printf("\nAfter Bubble Sort:\n");
    printArray(arr, n);
    printf("Number of swaps performed: %d\n", bubbleSwapCount);

    // Resetting the array to original order
    swap(&arr[0], &arr[3]);
    swap(&arr[1], &arr[2]);

    // Quick Sort
    int quickSwapCount = 0;
    quickSort(arr, 0, n-1, &quickSwapCount);
    printf("\nAfter Quick Sort:\n");
    printArray(arr, n);
    printf("Number of swaps performed: %d\n", quickSwapCount);

    return 0;
}
```

WAP to implement Insertion sort and Merge Sort on 1D array of Student structure (contains student_name, student_roll_no, total_marks), with key as student_roll_no. And count the number of swap performed.

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

struct Student {
    char student_name[50];
    int student_roll_no;
    int total_marks;
};

// Function to swap two Student structures
void swap(struct Student *a, struct Student *b) {
    struct Student temp = *a;
    *a = *b;
    *b = temp;
}

// Insertion Sort implementation
int insertionSort(struct Student arr[], int n) {
    int swapCount = 0;
    int i, j;
    struct Student key;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j].student_roll_no > key.student_roll_no) {
            arr[j+1] = arr[j];
            j--;
            swapCount++;
        }

        arr[j+1] = key;
```

```
    }
    return swapCount;
}

// Merge Sort implementation
void merge(struct Student arr[], int l, int m, int r, int *swapCount) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    struct Student L[n1], R[n2];

    for (i = 0; i < n1; i++) {
        L[i] = arr[l+i];
    }

    for (j = 0; j < n2; j++) {
        R[j] = arr[m+1+j];
    }

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (L[i].student_roll_no <= R[j].student_roll_no) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
            (*swapCount)++;
        }
        k++;
    }
```

```
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(struct Student arr[], int l, int r, int *swapCount) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m, swapCount);
        mergeSort(arr, m+1, r, swapCount);
        merge(arr, l, m, r, swapCount);
    }
}

// Function to print the Student array
void printArray(struct Student arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("Student Name: %s, Roll No.: %d, Total Marks: %d\n",
arr[i].student_name, arr[i].student_roll_no, arr[i].total_marks);
    }
}

// Driver function
int main() {
    struct Student arr[] = {
```

```
        {"John", 3, 85},
        {"Mary", 2, 90},
        {"Peter", 4, 80},
        {"Alice", 1, 95}
    };
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original Array:\n");
    printArray(arr, n);

    // Insertion Sort
    int insertionSwapCount = insertionSort(arr, n);
    printf("\nAfter Insertion Sort:\n");
    printArray(arr)
```

WAP to implement Selection sort and Bucket Sort on 1D array of Employee structure (contains employee_name, emp_no, emp_salary), with key as emp_no. And count the number of swap performed

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

struct Employee {
    char employee_name[50];
    int emp_no;
    int emp_salary;
};

// Function to swap two Employee structures
void swap(struct Employee *a, struct Employee *b) {
    struct Employee temp = *a;
    *a = *b;
    *b = temp;
```

```
}

// Selection Sort implementation
int selectionSort(struct Employee arr[], int n) {
    int swapCount = 0;
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j].emp_no < arr[min_idx].emp_no) {
                min_idx = j;
            }
        }
        if (min_idx != i) {
            swap(&arr[i], &arr[min_idx]);
            swapCount++;
        }
    }
    return swapCount;
}

// Bucket Sort implementation
void bucketSort(struct Employee arr[], int n, int maxEmpNo, int
*swapCount) {
    struct Employee *buckets[maxEmpNo+1];
    int i, j;
    for (i = 0; i <= maxEmpNo; i++) {
        buckets[i] = NULL;
    }

    for (i = 0; i < n; i++) {
        int bucketIndex = arr[i].emp_no;
        if (buckets[bucketIndex] == NULL) {
            buckets[bucketIndex] = (struct Employee*)malloc(sizeof(struct
Employee));
```

```
            *buckets[bucketIndex] = arr[i];
        } else {
            struct Employee *bucketPtr = buckets[bucketIndex];
            while (bucketPtr->emp_no != bucketIndex && bucketPtr->emp_no
!= -1) {
                bucketPtr++;
            }
            *bucketPtr = arr[i];
        }
    }

    int index = 0;
    for (i = 0; i <= maxEmpNo; i++) {
        if (buckets[i] != NULL) {
            int bucketSize = sizeof(buckets[i])/sizeof(struct Employee);
            for (j = 0; j < bucketSize; j++) {
                if (buckets[i][j].emp_no != -1) {
                    arr[index++] = buckets[i][j];
                }
            }
            free(buckets[i]);
        }
    }
}

// Function to print the Employee array
void printArray(struct Employee arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("Employee Name: %s, Emp No.: %d, Emp Salary: %d\n",
arr[i].employee_name, arr[i].emp_no, arr[i].emp_salary);
    }
}

// Driver function
```

```
int main() {
    struct Employee arr[] = {
        {"John", 3, 85000},
        {"Mary", 2, 90000},
        {"Peter", 4, 80000},
        {"Alice", 1, 95000}
    };
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original Array:\n");
    printArray(arr, n);

    // Selection Sort
    int selectionSwapCount = selectionSort(arr, n);
    printf("\nAfter Selection Sort:\n");
    printArray(arr, n);

    // Bucket Sort
    int maxEmpNo = arr[n-1].emp_no;
    bucketSort(arr, n, maxEmpNo, &selection
```

WAP to implement Shell sort and Heap Sort on 1D array of Employee structure (contains employee_name, emp_no, emp_salary), with key as emp_no. And count the number of swap performed

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

struct Employee {
    char employee_name[50];
    int emp_no;
    int emp_salary;
```

```
};

// Function to swap two Employee structures
void swap(struct Employee *a, struct Employee *b) {
    struct Employee temp = *a;
    *a = *b;
    *b = temp;
}

// Shell Sort implementation
int shellSort(struct Employee arr[], int n) {
    int swapCount = 0;
    int gap, i, j;
    struct Employee temp;
    for (gap = n/2; gap > 0; gap /= 2) {
        for (i = gap; i < n; i += 1) {
            temp = arr[i];
            for (j = i; j >= gap && arr[j-gap].emp_no > temp.emp_no; j -= gap) {
                arr[j] = arr[j-gap];
                swapCount++;
            }
            arr[j] = temp;
        }
    }
    return swapCount;
}

// Heap Sort implementation
void heapify(struct Employee arr[], int n, int i, int *swapCount) {
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;
    if (l < n && arr[l].emp_no > arr[largest].emp_no) {
        largest = l;
    }
```

```
    if (r < n && arr[r].emp_no > arr[largest].emp_no) {
        largest = r;
    }
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        (*swapCount)++;
        heapify(arr, n, largest, swapCount);
    }
}

void heapSort(struct Employee arr[], int n, int *swapCount) {
    int i;
    for (i = n/2-1; i >= 0; i--) {
        heapify(arr, n, i, swapCount);
    }
    for (i = n-1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        (*swapCount)++;
        heapify(arr, i, 0, swapCount);
    }
}

// Function to print the Employee array
void printArray(struct Employee arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("Employee Name: %s, Emp No.: %d, Emp Salary: %d\n",
arr[i].employee_name, arr[i].emp_no, arr[i].emp_salary);
    }
}

// Driver function
int main() {
    struct Employee arr[] = {
        {"John", 3, 85000},
```

```
        {"Mary", 2, 90000},
        {"Peter", 4, 80000},
        {"Alice", 1, 95000}
    };
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original Array:\n");
    printArray(arr, n);

    // Shell Sort
    int shellSwapCount = shellSort(arr, n);
    printf("\nAfter Shell Sort:\n");
    printArray(arr, n);

    // Heap Sort
    int heapSwapCount = 0;
    heapSort(arr, n, &heapSwapCount);
    printf("\nAfter Heap Sort:\n");
    printArray(arr, n);

    printf("\nNumber of swaps performed in Shell Sort: %d\n",
shellSwapCount);
    printf("Number of swaps performed);
```

WAP to implement Insertion sort and Quick Sort on 1D array of Student structure (contains student_name, student_roll_no, total_marks), with key as student_roll_no. And count the number of swap performed.

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

struct Student {
    char student_name[50];
    int student_roll_no;
    int total_marks;
```

```
};

// Function to swap two Student structures
void swap(struct Student *a, struct Student *b) {
    struct Student temp = *a;
    *a = *b;
    *b = temp;
}

// Insertion Sort implementation
int insertionSort(struct Student arr[], int n) {
    int swapCount = 0;
    int i, j;
    struct Student temp;
    for (i = 1; i < n; i++) {
        temp = arr[i];
        for (j = i-1; j >= 0 && arr[j].student_roll_no > temp.student_roll_no; j--) {
            arr[j+1] = arr[j];
            swapCount++;
        }
        arr[j+1] = temp;
    }
    return swapCount;
}

// Quick Sort implementation
int partition(struct Student arr[], int low, int high, int *swapCount) {
    struct Student pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j].student_roll_no < pivot.student_roll_no) {
            i++;
            swap(&arr[i], &arr[j]);
            (*swapCount)++;
        }
```

```
    }
    swap(&arr[i+1], &arr[high]);
    (*swapCount)++;
    return i+1;
}

void quickSort(struct Student arr[], int low, int high, int *swapCount) {
    if (low < high) {
        int pi = partition(arr, low, high, swapCount);
        quickSort(arr, low, pi-1, swapCount);
        quickSort(arr, pi+1, high, swapCount);
    }
}

// Function to print the Student array
void printArray(struct Student arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("Student Name: %s, Roll No.: %d, Total Marks: %d\n",
arr[i].student_name, arr[i].student_roll_no, arr[i].total_marks);
    }
}

// Driver function
int main() {
    struct Student arr[] = {
        {"John", 3, 85},
        {"Mary", 2, 90},
        {"Peter", 4, 80},
        {"Alice", 1, 95}
    };
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Original Array:\n");
    printArray(arr, n);
```

```
    // Insertion Sort
    int insertionSwapCount = insertionSort(arr, n);
    printf("\nAfter Insertion Sort:\n");
    printArray(arr, n);

    // Quick Sort
    int quickSwapCount = 0;
    quickSort(arr, 0, n-1, &quickSwapCount);
    printf("\nAfter Quick Sort:\n");
    printArray(arr, n);

    printf("\nNumber of swaps performed in Insertion Sort: %d\n",
insertionSwapCount);
    printf("Number of swaps performed in Quick Sort: %d\n",
quickSwapCount);

    return 0;
}
```

WAP to implement Selection sort and Merge Sort on 1D array of Student structure (contains student_name, student_roll_no, total_marks), with key as student_roll_no. And count the number of swap performed.

```
#include <iostream>
#include <string>
using namespace std;

struct Student {
    string student_name;
    int student_roll_no;
    int total_marks;
};

// Selection Sort
```

```
int selectionSort(Student arr[], int n) {
    int swap_count = 0;
    for (int i = 0; i < n-1; i++) {
        int min_idx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j].student_roll_no < arr[min_idx].student_roll_no) {
                min_idx = j;
            }
        }
        if (min_idx != i) {
            swap(arr[i], arr[min_idx]);
            swap_count++;
        }
    }
    return swap_count;
}

// Merge Sort
void merge(Student arr[], int left, int mid, int right, int& swap_count) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    Student L[n1], R[n2];

    // Copy data to temporary arrays
    for (i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    // Merge the temporary arrays back into arr[l..r]
```

```
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (L[i].student_roll_no <= R[j].student_roll_no) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
            swap_count += n1 - i;
        }
        k++;
    }

    // Copy the remaining elements of L[]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[]
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(Student arr[], int left, int right, int& swap_count) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
```

```
        mergeSort(arr, left, mid, swap_count);
        mergeSort(arr, mid + 1, right, swap_count);

        // Merge the sorted halves
        merge(arr, left, mid, right, swap_count);
    }
}

int main() {
    Student arr[] = {
        {"Alice", 3, 90},
        {"Bob", 1, 85},
        {"Charlie", 4, 80},
        {"David", 2, 95},
        {"Emily", 5, 75}
    };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Selection Sort
    cout << "Selection Sort: " << endl;
    int selection_swap_count = selectionSort(arr, n);
    for (int i = 0; i < n; i++) {
        cout << "Name: " << arr[i].student_name << ", Roll No: " <<
arr[i].student_roll_no << ", Total Marks: " << arr[i].total_marks << endl;
    }
    cout <<
```

Sorting & Searching WAP to implement Bubble sort and Heap Sort on 1D array of Employee structure (contains employee_name, emp_no, emp_salary), with key as emp_no. And count the number of swap performed

orting & Searching WAP to implement Bubble sort and Heap Sort on 1D array of Employee structure (contains employee_name, emp_no, emp_salary), with key as emp_no. And count the number of swap performed

```
#include <iostream>
#include <string>
using namespace std;

struct Employee {
    string employee_name;
    int emp_no;
    int emp_salary;
};

// Bubble Sort
int bubbleSort(Employee arr[], int n) {
    int swap_count = 0;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j].emp_no > arr[j+1].emp_no) {
                swap(arr[j], arr[j+1]);
                swap_count++;
            }
        }
    }
    return swap_count;
}

// Heap Sort
void heapify(Employee arr[], int n, int i, int& swap_count) {
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    // If left child is larger than root
    if (l < n && arr[l].emp_no > arr[largest].emp_no) {
        largest = l;
    }
```

```
    // If right child is larger than largest so far
    if (r < n && arr[r].emp_no > arr[largest].emp_no) {
        largest = r;
    }

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        swap_count++;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest, swap_count);
    }
}

void heapSort(Employee arr[], int n, int& swap_count) {
    // Build heap (rearrange array)
    for (int i = n/2 - 1; i >= 0; i--) {
        heapify(arr, n, i, swap_count);
    }

    // One by one extract an element from heap
    for (int i = n-1; i >= 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);
        swap_count++;

        // call max heapify on the reduced heap
        heapify(arr, i, 0, swap_count);
    }
}

int main() {
    Employee arr[] = {
```

```
        {"Alice", 3, 50000},
        {"Bob", 1, 75000},
        {"Charlie", 4, 40000},
        {"David", 2, 90000},
        {"Emily", 5, 60000}
    };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Bubble Sort
    cout << "Bubble Sort: " << endl;
    int bubble_swap_count = bubbleSort(arr, n);
    for (int i = 0; i < n; i++) {
        cout << "Name: " << arr[i].employee_name << ", Emp No: " <<
arr[i].emp_no << ", Emp Salary: " << arr[i].emp_salary << endl;
    }
    cout << "Number of Swaps: " << bubble_swap_count << endl;

    // Heap Sort
    cout << "Heap Sort: " << endl;
    int heap_swap_count = 0;
    heapSort(arr, n, heap_swap_count);
    for (int i = 0; i < n; i++) {
        cout << "Name: " << arr[i].employee_name << ", Emp No: " <<
arr[i].emp_no << ", Emp Salary: " << arr[i].emp_salary << endl;
```

WAP to implement Bucket Sort and Quick sort on 1D array of Faculty structure (contains faculty_name, faculty_ID, subject_codes, class_names), with key as faculty_ID. And count the number of swap performed

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

```
#define MAX_ID 1000000 // Maximum value of faculty_ID

// Structure to represent a faculty member
struct Faculty {
    char faculty_name[50];
    int faculty_ID;
    char subject_codes[50];
    char class_names[50];
};

// Function to swap two Faculty objects
void swap(struct Faculty* a, struct Faculty* b, int* count) {
    struct Faculty temp = *a;
    *a = *b;
    *b = temp;
    (*count)++;
}

// Function to implement Bucket Sort
void bucketSort(struct Faculty arr[], int n, int* count) {
    // Create buckets
    struct Faculty buckets[MAX_ID];
    memset(buckets, 0, sizeof(buckets));

    // Place each Faculty object in the corresponding bucket
    for (int i = 0; i < n; i++) {
        int bucket_index = arr[i].faculty_ID - 1;
        buckets[bucket_index] = arr[i];
    }

    // Copy the sorted Faculty objects back to the original array
    int index = 0;
    for (int i = 0; i < MAX_ID; i++) {
        if (buckets[i].faculty_ID != 0) {
```

```
            arr[index++] = buckets[i];
        }
    }
}

// Function to implement Quick Sort
void quickSort(struct Faculty arr[], int low, int high, int* count) {
    if (low < high) {
        int pivot = arr[high].faculty_ID;
        int i = low - 1;

        for (int j = low; j <= high - 1; j++) {
            if (arr[j].faculty_ID < pivot) {
                i++;
                swap(&arr[i], &arr[j], count);
            }
        }

        swap(&arr[i+1], &arr[high], count);

        int partition = i + 1;

        quickSort(arr, low, partition - 1, count);
        quickSort(arr, partition + 1, high, count);
    }
}

int main() {
    int n;
    printf("Enter the number of faculty members: ");
    scanf("%d", &n);

    // Create an array of Faculty objects
    struct Faculty arr[n];
```

```
    // Read the input Faculty objects
    for (int i = 0; i < n; i++) {
        printf("Enter details of faculty member %d:\n", i+1);
        printf("Name: ");
        scanf("%s", arr[i].faculty_name);
        printf("ID: ");
        scanf("%d", &arr[i].faculty_ID);
        printf("Subject codes: ");
        scanf("%s", arr[i].subject_codes);
        printf("Class names: ");
        scanf("%s", arr[i].class_names);
    }

    // Implement Bucket Sort and count the number of swaps
    int bucket_sort_count = 0;
    bucketSort(arr, n, &bucket_sort_count);

    // Implement Quick Sort and count the number of swaps
    int quick_sort_count = 0;
    quickSort(arr, 0, n-1, &quick_sort_count);

    // Print the sorted array
    printf("\nBucket Sort:\n");
    for (int i = 0; i < n; i++) {
        printf("%s %d %s %s\n", arr[i].faculty
```

WAP to implement Merge Sort and Heap Sort on 1D array of Faculty structure (contains faculty_name, faculty_ID, subject_codes, class_names), with key as faculty_ID. And count the number of swap performed

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

```
// Faculty structure
struct Faculty {
    char faculty_name[50];
    int faculty_ID;
    char subject_codes[20];
    char class_names[20];
};

// Merge function for Merge Sort
void merge(struct Faculty arr[], int l, int m, int r, int *swap_count) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temp arrays
    struct Faculty L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2) {
        // Compare and merge
        if (L[i].faculty_ID <= R[j].faculty_ID) {
            arr[k] = L[i];
            i++;
        }
        else {
```

```
            arr[k] = R[j];
            j++;
        }
        *swap_count += 1; // Increment swap count
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Merge Sort function
void merge_sort(struct Faculty arr[], int l, int r, int *swap_count) {
    if (l < r) {
        int m = l + (r - l) / 2;

        // Sort first and second halves
        merge_sort(arr, l, m, swap_count);
        merge_sort(arr, m + 1, r, swap_count);

        // Merge the sorted halves
        merge(arr, l, m, r, swap_count);
    }
}
```

```
// Heapify function for Heap Sort
void heapify(struct Faculty arr[], int n, int i, int *swap_count) {
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // Left child index
    int r = 2 * i + 2; // Right child index

    // If left child is larger than root
    if (l < n && arr[l].faculty_ID > arr[largest].faculty_ID)
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r].faculty_ID > arr[largest].faculty_ID)
        largest = r;

    // If largest is not root
    if (largest != i) {
        // Swap arr[i] with arr[largest]
        struct Faculty temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        *swap_count += 1; // Increment swap count

        // Recursively heap
```

You have a fleet of N cars waiting for repair, with the estimated repair times rk for the car Ci , 1 ? k ? N. What is the best repairschedule (order of repairs) to minimize the total lost time for being outofservice. How much computation is needed to find the lost servicetimes all schedules?

To minimize the total lost time for being out-of-service, we can use the Shortest Job First (SJF) scheduling algorithm. This algorithm schedules the jobs in ascending order of their estimated repair times.

To find the lost service times for all schedules, we need to calculate the waiting time for each car. The waiting time for a car is the time from when it arrives until it starts being repaired. The total lost service time for all schedules is the sum of the waiting times for all cars.

To calculate the waiting time for a car, we need to know the repair times of all the cars that arrived before it. So, we need to consider all possible permutations of the N cars to find the best repair schedule that minimizes the total lost time for being out-of-service.

The total number of permutations of N cars is N! (N factorial), which is a very large number. Therefore, finding the best repair schedule for a large fleet of cars can be computationally very expensive.

One way to reduce the computational cost is to use a heuristic approach. For example, we can use the nearest neighbor algorithm, which starts with a random car and selects the car with the shortest estimated repair time that has not been scheduled yet. This algorithm may not always give the optimal solution, but it can significantly reduce the computational cost while still producing reasonably good results.
The total lost service time for all schedules is the sum of the waiting times for all cars.

To calculate the waiting time for a car, we need to know the repair times of all the cars that arrived before it. So, we need to consider all possible permutations of the N cars to find the best repair schedule that minimizes the total lost time for being out-of-service.

The total number of permutations of N cars is N! (N factorial), which is a very large number. Therefore, finding the best repair schedule for a large fleet of cars can be computationally very expensive.
One way to reduce the computational cost is to use a heuristic approach. For example, we can use the nearest neighbor algorithm, which starts with a random car and selects the car with the shortest estimated repair time

that has not been scheduled yet. This algorithm may not always give the optimal solution, but it can significantly reduce the computational cost while still producing reasonably good results.

Write a program to arrange the data of the faculties recruited in the institute. There are three categories of faculties in the every deartment namely professor, Associate professor, and assistant professor. Recruitment is done as mentioned below. 1. Every professor has two associate professors. 2. Every Associate has two assistant professors. Data is given randomly. Suggest suitable sorting method and implement.

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

struct faculty {
    char name[20];
    char category[20];
    int recruitment_order;
};

void merge(struct faculty arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    struct faculty L[n1], R[n2];

    for (i = 0; i < n1; i++) {
        strcpy(L[i].name, arr[left + i].name);
        strcpy(L[i].category, arr[left + i].category);
        L[i].recruitment_order = arr[left + i].recruitment_order;
    }
```

```
    for (j = 0; j < n2; j++) {
        strcpy(R[j].name, arr[mid + 1 + j].name);
        strcpy(R[j].category, arr[mid + 1 + j].category);
        R[j].recruitment_order = arr[mid + 1 + j].recruitment_order;
    }

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (strcmp(L[i].category, R[j].category) < 0 ||
            (strcmp(L[i].category, R[j].category) == 0 && L[i].recruitment_order <
R[j].recruitment_order)) {
            strcpy(arr[k].name, L[i].name);
            strcpy(arr[k].category, L[i].category);
            arr[k].recruitment_order = L[i].recruitment_order;
            i++;
        }
        else {
            strcpy(arr[k].name, R[j].name);
            strcpy(arr[k].category, R[j].category);
            arr[k].recruitment_order = R[j].recruitment_order;
            j++;
        }
        k++;
    }

    while (i < n1) {
        strcpy(arr[k].name, L[i].name);
        strcpy(arr[k].category, L[i].category);
        arr[k].recruitment_order = L[i].recruitment_order;
        i++;
        k++;
    }
```

```
    while (j < n2) {
        strcpy(arr[k].name, R[j].name);
        strcpy(arr[k].category, R[j].category);
        arr[k].recruitment_order = R[j].recruitment_order;
        j++;
        k++;
    }
}

void mergeSort(struct faculty arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

int main() {
    struct faculty faculties[] = {
        {"John Doe", "associate professor", 3},
        {"Jane Smith", "assistant professor", 2},
        {"Mike Johnson", "professor", 1},
        {"Lisa Brown", "assistant professor", 5},
```

Assume that an array A with n elements was sorted in an ascending order, but two of its elements swapped their positions by a mistake while maintaining the array. Write a code to identify the swapped pair of elements and their positions in the asymptotically best possible time. [Assume that all given elements are distinct integers.]

```
#include<stdio.h>

void findSwappedPair(int arr[], int n) {
    int i, j;
    int first = -1, second = -1;

    for (i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            first = i;
            break;
        }
    }

    if (first == -1) {
        printf("The array is already sorted.\n");
        return;
    }

    for (j = first + 1; j < n - 1; j++) {
        if (arr[j] > arr[j + 1]) {
            second = j;
            break;
        }
    }

    if (second == -1) {
        second = n - 1;
    }

    printf("The swapped pair is (%d, %d) at positions (%d, %d).\n", arr[first],
arr[second], first, second);
}

int main() {
    int arr[] = {1, 3, 5, 4, 2, 6};
```

```
    int n = sizeof(arr) / sizeof(arr[0]);

    findSwappedPair(arr, n);

    return 0;
}
```

Using Quick sort, assign the roll nos. to the students of your class as per Searching their previous years result. i.e. topper will be roll no. 1

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

typedef struct student {
    char name[50];
    int roll;
    float previous_year_result;
} Student;

void swap(Student* a, Student* b) {
    Student temp = *a;
    *a = *b;
    *b = temp;
}

int partition(Student arr[], int low, int high) {
    float pivot = arr[high].previous_year_result;
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j].previous_year_result >= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
```

```
    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);
}

void quicksort(Student arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

int main() {
    int n;

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

    Student* students = (Student*) malloc(n * sizeof(Student));

    printf("Enter the students' names and previous year's results:\n");

    for (int i = 0; i < n; i++) {
        printf("Enter the name of student %d: ", i + 1);
        scanf("%s", students[i].name);

        printf("Enter the previous year's result of student %d: ", i + 1);
        scanf("%f", &students[i].previous_year_result);

        students[i].roll = i + 1;
    }
```

```
    quicksort(students, 0, n - 1);

    printf("The assigned roll numbers are:\n");

    for (int i = 0; i < n; i++) {
        printf("%d. %s (previous year's result: %.2f)\n", i + 1, students[i].name,
students[i].previous_year_result);

        if (students[i].previous_year_result ==
students[0].previous_year_result) {
            printf("The topper is %s with roll number 1.\n", students[i].name);
        }

        students[i].roll = i + 1;
    }

    free(students);

    return 0;
}

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

typedef struct employee {
    char name[50];
    float height;
    float weight;
} Employee;

void merge(Employee arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
```

```
Employee L[n1], R[n2];

for (i = 0; i < n1; i++)
    L[i] = arr[left + i];
for (j = 0; j < n2; j++)
    R[j] = arr[mid + 1 + j];

i = 0;
j = 0;
k = left;

while (i < n1 && j < n2) {
    if ((L[i].height + L[i].weight) / 2 >= (R[j].height + R[j].weight) / 2) {
        arr[k] = L[i];
        i++;
    }
    else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
```

```
}

void mergesort(Employee arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergesort(arr, left, mid);
        mergesort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

int main() {
    int n;

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

    Employee* employees = (Employee*) malloc(n * sizeof(Employee));

    printf("Enter the employees' names, heights, and weights:\n");

    for (int i = 0; i < n; i++) {
        printf("Enter the name of employee %d: ", i + 1);
        scanf("%s", employees[i].name);

        printf("Enter the height of employee %d (in meters): ", i + 1);
        scanf("%f", &employees[i].height);

        printf("Enter the weight of employee %d (in kilograms): ", i + 1);
        scanf("%f", &employees[i].weight);
    }

    mergesort(employees, 0, n - 1);
```

```
    printf("The arranged list of employees based on the average of their
height and weight is:\n");

    for (int i = 0; i < n; i++) {
        printf("%d. %s (average of height and weight: %.2f)\n", i + 1,
employees[i].name, (employees[i].height + employees[i].weight) / 2);
    }

    free(employees);

    return 0;
}
```

Given a set of points Pi , 1 ? i ? N (? 2) on the x axis, find Pi and P j such that |Pi ? P j | is minimum. e.g. P1 | P2 | P3 | P4 | P5 | P6 {P4 , P6} is the closest pair

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

typedef struct {
    double x;
    double y;
} Point;

int cmp_x(const void* a, const void* b) {
    Point* p1 = (Point*) a;
    Point* p2 = (Point*) b;
    if (p1->x < p2->x) return -1;
    if (p1->x > p2->x) return 1;
    return 0;
}
```

```
int cmp_y(const void* a, const void* b) {
    Point* p1 = (Point*) a;
    Point* p2 = (Point*) b;
    if (p1->y < p2->y) return -1;
    if (p1->y > p2->y) return 1;
    return 0;
}

double dist(Point p1, Point p2) {
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y;
    return sqrt(dx*dx + dy*dy);
}

double brute_force(Point* points, int n) {
    double min_dist = INFINITY;
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            double d = dist(points[i], points[j]);
            if (d < min_dist) {
                min_dist = d;
            }
        }
    }
    return min_dist;
}

double closest_pair(Point* points, Point* temp, int left, int right) {
    if (right - left <= 3) {
        return brute_force(points + left, right - left);
    }
    int mid = (left + right) / 2;
    double d1 = closest_pair(points, temp, left, mid);
    double d2 = closest_pair(points, temp, mid, right);
```

```
    double d = fmin(d1, d2);
    int k = 0;
    for (int i = left; i < right; i++) {
        if (fabs(points[i].x - points[mid].x) < d) {
            temp[k++] = points[i];
        }
    }
    qsort(temp, k, sizeof(Point), cmp_y);
    for (int i = 0; i < k-1; i++) {
        for (int j = i+1; j < k && temp[j].y - temp[i].y < d; j++) {
            double dij = dist(temp[i], temp[j]);
            if (dij < d) {
                d = dij;
            }
        }
    }
    return d;
}

int main() {
    int n;
    printf("Enter the number of points: ");
    scanf("%d", &n);
    Point* points = malloc(n * sizeof(Point));
    Point* temp = malloc(n * sizeof(Point));
    for (int i = 0; i < n; i++) {
        printf("Enter x and y coordinates of point %d: ", i+1);
        scanf("%lf %lf", &points[i].x, &points[i].y);
    }
    qsort(points, n, sizeof(Point), cmp_x);
    double d = closest_pair
```

WAP to search a particular students record in 'n' number of students pool by using Binary search with and without recursive function. Assume suitable data for student record.

```
#include <stdio.h>

// Define a struct for a student record
struct student {
    int roll_no;
    char name[50];
    float marks;
};

// Binary search function without recursion
int binary_search(struct student students[], int n, int roll_no) {
    int low = 0;
    int high = n - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (students[mid].roll_no == roll_no) {
            return mid;
        } else if (students[mid].roll_no < roll_no) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1;
}

// Binary search function with recursion
```

```
int binary_search_recursive(struct student students[], int low, int high, int
roll_no) {
    if (low > high) {
        return -1;
    }

    int mid = (low + high) / 2;

    if (students[mid].roll_no == roll_no) {
        return mid;
    } else if (students[mid].roll_no < roll_no) {
        return binary_search_recursive(students, mid + 1, high, roll_no);
    } else {
        return binary_search_recursive(students, low, mid - 1, roll_no);
    }
}

int main() {
    int n, roll_no, index;
    printf("Enter the number of students: ");
    scanf("%d", &n);

    // Declare an array of student records
    struct student students[n];

    // Populate the array with student records
    for (int i = 0; i < n; i++) {
        printf("Enter the details for student %d:\n", i + 1);
        printf("Roll no: ");
        scanf("%d", &students[i].roll_no);
        printf("Name: ");
        scanf("%s", students[i].name);
        printf("Marks: ");
        scanf("%f", &students[i].marks);
    }
```

```
    // Prompt the user to enter the roll no to search for
    printf("Enter the roll no to search for: ");
    scanf("%d", &roll_no);

    // Perform binary search using the non-recursive function
    index = binary_search(students, n, roll_no);
    if (index != -1) {
        printf("Student found:\n");
        printf("Roll no: %d\n", students[index].roll_no);
        printf("Name: %s\n", students[index].name);
        printf("Marks: %.2f\n", students[index].marks);
    } else {
        printf("Student not found.\n");
    }

    // Perform binary search using the recursive function
    index = binary_search_recursive(students, 0, n - 1, roll_no);
    if (index != -1) {
        printf("Student found:\n");
        printf("Roll no: %d\n", students[index].roll_no);
        printf("Name: %s\n", students[index].name);
        printf("Marks: %.2f\n", students[index].marks);
    } else {
        printf("Student not found.\n");
    }

    return 0;
}
```

WAP to perform addition o f two polynomials using singly linked list

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

```c
#include <stdlib.h>
struct Node {
   int coef;
   int exp;
   struct Node* next;
};
typedef struct Node Node;
void insert(Node** poly, int coef, int exp) {
   Node* temp = (Node*) malloc(sizeof(Node));
   temp->coef = coef;
   temp->exp = exp;
   temp->next = NULL;

   if (*poly == NULL) {
       *poly = temp;
       return;
   }

   Node* current = *poly;

   while (current->next != NULL) {
       current = current->next;
   }

   current->next = temp;
}
void print(Node* poly) {
   if (poly == NULL) {
       printf("0\n");
       return;
   }

   Node* current = poly;

   while (current != NULL) {
       printf("%dx^%d", current->coef, current->exp);
       if (current->next != NULL) {
           printf(" + ");
       }
       current = current->next;
   }

   printf("\n");
}
Node* add(Node* poly1, Node* poly2) {
   Node* result = NULL;
```

```
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exp == poly2->exp) {
            insert(&result, poly1->coef + poly2->coef, poly1->exp);
            poly1 = poly1->next;
            poly2 = poly2->next;
        } else if (poly1->exp > poly2->exp) {
            insert(&result, poly1->coef, poly1->exp);
            poly1 = poly1->next;
        } else {
            insert(&result, poly2->coef, poly2->exp);
            poly2 = poly2->next;
        }
    }

    while (poly1 != NULL) {
        insert(&result, poly1->coef, poly1->exp);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        insert(&result, poly2->coef, poly2->exp);
        poly2 = poly2->next;
    }

    return result;
}
int main() {
    Node* poly1 = NULL;
    insert(&poly1, 5, 4);
    insert(&poly1, 3, 2);
    insert(&poly1, 1, 0);
    Node* poly2 = NULL;
    insert(&poly2, 4, 4);
    insert(&poly2, 2, 2);
    insert(&poly2, 1, 1);
    printf("First polynomial: ");
    print(poly1);
    printf("Second polynomial: ");
    print(poly2);
    Node* result = add(poly1, poly2);
    printf("Result: ");
    print(result);
    return 0;
}
```

WAP to perform Multiplication o f two polynomials using singly linked i

```
#include<stdi
o.h>
                #include<stdlib.h>


                struct node

                {

                int coefficient,exponent;

                struct node *next;

                };


                struct node *hPtr1,*hPtr2,*hPtr3;


                struct node *buildNode(int coefficient, int exponent)

                {

                struct node *ptr=(struct node *)malloc(sizeof(struct node));

                ptr->coefficient=coefficient;

                ptr->exponent=exponent;

                ptr->next=NULL;

                return ptr;

                }


                void polynomial_insert(struct node ** myNode,int coefficient,int
                exponent)
                {

                struct node *lPtr,*pPtr,*qPtr=*myNode;

                lPtr=buildNode(coefficient,exponent);

                if (*myNode==NULL || (*myNode)->exponent<exponent)
```

```
{
 *myNode=lPtr;
 (*myNode)->next=qPtr;
 return;
}
while(qPtr)
{
 pPtr=qPtr;
 qPtr=qPtr->next;
 if(!qPtr)
 {
  pPtr->next=lPtr;
  break;
 }
 else if((exponent<pPtr->exponent) && (exponent>qPtr->exponent))
 {
  lPtr->next = qPtr;
  pPtr->next = lPtr;
  break;
 }
}
return;
}


void polynomial_add(struct node **n1,int coefficient,int exponent)
{
struct node *x=NULL,*temp=*n1;
if (*n1==NULL || (*n1)->exponent<exponent)
{
 *n1=x=buildNode(coefficient,exponent);
 (*n1)->next=temp;
```

```
}
else
{
 while(temp)
 {
  if(temp->exponent==exponent)
  {
   temp->coefficient=temp->coefficient+coefficient;
   return;
  }
  if(temp->exponent>exponent && (!temp->next ||
temp->next->exponent<exponent))
  {
   x=buildNode(coefficient,exponent);
   x->next=temp->next;
   temp->next=x;
   return;
  }
  temp=temp->next;
 }
 x->next=NULL;
 temp->next=x;
}
}

void polynomial_multiply(struct node **n1,struct node *n2,struct
node *n3)
{
struct node * temp;
int coefficient, exponent;
temp = n3;
if(!n2 && !n3)
{
```

```
 return;
}
if(!n2)
{
 *n1 = n3;
}
else if(!n3)
{
 *n1 = n2;
}
else
{
 while(n2)
 {
  while(n3)
  {
   coefficient = n2->coefficient * n3->coefficient;
   exponent = n2->exponent + n3->exponent;
   n3 = n3->next;
   polynomial_add(n1, coefficient, exponent);
  }
  n3 = temp;
  n2 = n2->next;
 }
}
return;
}


struct node *polynomial_deleteList(struct node *ptr)
{
struct node *temp;
```

```
while(ptr)
{
 temp=ptr->next;
 free(ptr);
 ptr = temp;
}
return NULL;
}

void polynomial_view(struct node *ptr)
{
int i = 0;
int flag=0;
while (ptr)
{
 if(ptr->exponent!=0 || ptr->exponent!= 1)
 {
  if(ptr->coefficient>0 && flag==0)
  {
   printf("%dx^%d", ptr->coefficient,ptr->exponent);
   flag++;
  }
  else if(ptr->coefficient>0 && flag==1)
  {
   printf("+%dx^%d", ptr->coefficient,ptr->exponent);
  }
  else if(ptr->coefficient<0)
  {
   printf("%dx^%d",ptr->coefficient,ptr->exponent);
  }
 }
 else if(ptr->exponent==0)
```

```
{
 if(ptr->coefficient>0 && flag==0 )
 {
  printf("%d",ptr->coefficient);
    flag++;
 }
 else if(ptr->coefficient>0 && flag==1)
 {
  printf("+%d", ptr->coefficient);
 }
 else if(ptr->coefficient < 0)
 {
  printf("%d", ptr->coefficient);
 }
}
else if(ptr->exponent==1)
{
 if(ptr->coefficient>0 && flag==0)
 {
  printf("%dx",ptr->coefficient);
  flag++;
 }
 else if(ptr->coefficient > 0 && flag==1)
 {
  printf("+%dx", ptr->coefficient);
 }
 else if(ptr->coefficient < 0)
 {
  printf("%dx", ptr->coefficient);
 }
}
ptr=ptr->next;
```

```
 i++;
}
printf("\n");
return;
}

int main(int argc,char *argv[])
{
int coefficient,exponent,i,n;
int count;
printf("Multiplication of Two Polynomials\n");
printf("Enter the number of coefficients in the multiplicand:");
scanf("%d",&count);
for(i=0;i<count;i++)
{
 printf("Enter the coefficient part:");
 scanf("%d", &coefficient);
 printf("Enter the exponent part:");
 scanf("%d",&exponent);
 polynomial_insert(&hPtr1, coefficient, exponent);
}
printf("Enter the number of coefficients in the multiplier:");
scanf("%d",&count);
for(i=0;i<count;i++)
{
 printf("Enter the coefficient part:");
 scanf("%d", &coefficient);
 printf("Enter the exponent part:");
 scanf("%d",&exponent);
 polynomial_insert(&hPtr2, coefficient, exponent);
}
printf("Polynomial Expression 1: ");
```

```
polynomial_view(hPtr1);

printf("Polynomial Expression 2: ");

polynomial_view(hPtr2);

polynomial_multiply(&hPtr3, hPtr1, hPtr2);

printf("Output:\n");

polynomial_view(hPtr3);

hPtr1 = polynomial_deleteList(hPtr1);

hPtr2 = polynomial_deleteList(hPtr2);

hPtr3 = polynomial_deleteList(hPtr3);

return 0;

}
```

WAP to st
ore at most 10 digit integer in a Doubly linked list and perform arithmetic operations on it.

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

// Define the structure of the node in the doubly linked list
typedef struct node {
    int data;
    struct node* next;
    struct node* prev;
} Node;

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

```
// Function to insert a new node at the beginning of the doubly linked list
void insert(Node** headRef, int data) {
    Node* newNode = createNode(data);
    if (*headRef == NULL) {
        *headRef = newNode;
    } else {
        newNode->next = *headRef;
        (*headRef)->prev = newNode;
        *headRef = newNode;
    }
}

// Function to delete a node from the doubly linked list
void delete(Node** headRef, Node* nodeToDelete) {
    if (*headRef == NULL || nodeToDelete == NULL) {
        return;
    }
    if (*headRef == nodeToDelete) {
        *headRef = nodeToDelete->next;
    }
    if (nodeToDelete->next != NULL) {
        nodeToDelete->next->prev = nodeToDelete->prev;
    }
    if (nodeToDelete->prev != NULL) {
        nodeToDelete->prev->next = nodeToDelete->next;
    }
    free(nodeToDelete);
}

// Function to delete the entire doubly linked list
void deleteList(Node** headRef) {
    Node* current = *headRef;
    while (current != NULL) {
        Node* next = current->next;
```

```
        free(current);
        current = next;
    }
    *headRef = NULL;
}

// Function to display the contents of the doubly linked list
void display(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d", current->data);
        current = current->next;
    }
    printf("\n");
}

// Function to add two numbers stored in the doubly linked list
Node* add(Node* num1, Node* num2) {
    Node* result = NULL;
    Node* current1 = num1;
    Node* current2 = num2;
    int carry = 0;
    while (current1 != NULL || current2 != NULL) {
        int sum = carry;
        if (current1 != NULL) {
            sum += current1->data;
            current1 = current1->next;
        }
        if (current2 != NULL) {
            sum += current2->data;
            current2 = current2->next;
        }
        carry = sum / 10;
        sum = sum % 10;
        insert(&result, sum);
```

```
    }
    if (carry > 0) {
        insert(&result, carry);
    }
    return result;
}

// Function to subtract two numbers stored in the doubly linked list
Node* subtract(Node* num1, Node* num2) {
    Node* result = NULL;
    Node* current1 = num1;
    Node* current2 = num2;
    int borrow = 0;
    while (current1 != NULL || current2 != NULL) {
        int diff = borrow;
        if (current1 != NULL) {
            diff += current1->data;
            current1 = current1->next;
        }
        if (current2 != NULL) {
            diff
```

WAP to convert a given Infix expression into its equivalent Postfix expression and evaluate it using stack.

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

#define MAX_SIZE 100

// Stack data structure
typedef struct {
```

```
    int top;
    int data[MAX_SIZE];
} Stack;

// Function to initialize stack
void initialize(Stack *s) {
    s->top = -1;
}

// Function to check if stack is empty
int isEmpty(Stack *s) {
    return (s->top == -1);
}

// Function to check if stack is full
int isFull(Stack *s) {
    return (s->top == MAX_SIZE - 1);
}

// Function to push an element onto the stack
void push(Stack *s, int element) {
    if (isFull(s)) {
        printf("Error: Stack is full\n");
        exit(0);
    }
    s->data[++(s->top)] = element;
}

// Function to pop an element from the stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Error: Stack is empty\n");
        exit(0);
    }
    return s->data[(s->top)--];
```

```
}

// Function to get the top element of the stack
int top(Stack *s) {
    if (isEmpty(s)) {
        printf("Error: Stack is empty\n");
        exit(0);
    }
    return s->data[s->top];
}

// Function to check if a character is an operator
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Function to get the precedence of an operator
int precedence(char c) {
    if (c == '+' || c == '-') {
        return 1;
    } else if (c == '*' || c == '/') {
        return 2;
    } else {
        return 0;
    }
}

// Function to convert an infix expression to a postfix expression
void infixToPostfix(char *infix, char *postfix) {
    Stack s;
    initialize(&s);
    int i, j;
    char c;
    for (i = 0, j = 0; infix[i] != '\0'; i++) {
        c = infix[i];
```

```
        if (isdigit(c)) {
            postfix[j++] = c;
        } else if (isOperator(c)) {
            while (!isEmpty(&s) && precedence(top(&s)) >= precedence(c)) {
                postfix[j++] = pop(&s);
            }
            push(&s, c);
        } else if (c == '(') {
            push(&s, c);
        } else if (c == ')') {
            while (top(&s) != '(') {
                postfix[j++] = pop(&s);
            }
            pop(&s);
        }
    }
    while (!isEmpty(&s)) {
        postfix[j++] = pop(&s);
    }
    postfix[j] = '\0';
}

// Function to evaluate a postfix expression
int evaluatePostfix(char *postfix) {
    Stack s;
    initialize(&s);
    int i, x, y, result;
    char c;
    for (i = 0; postfix[i] != '\0'; i++) {
        c = postfix[i];
        if (isdigit(c)) {
            push(&s, c - '0');
        } else if (isOperator(c)) {
            y = pop(&s);
            x = pop(&s);
```

```
switch (c) {
    case '+':
        result = x + y;
        break;
```