1) Implement Queue operations (Insert, delete, display front & rear values) - 04/10/23

**source code**

#include <stdio.h>

#define MAX\_SIZE 100

int queue[MAX\_SIZE];

int front = -1, rear = -1;

void enqueue(int value)

{

if (rear == MAX\_SIZE - 1)

{

printf("Queue is full. Cannot enqueue.\n");

return;

}

if (front == -1)

front = 0;

rear++;

queue[rear] = value;

}

void dequeue()

{

if (front == -1)

{

printf("Queue is empty. Cannot dequeue.\n");

return;

}

printf("Dequeued element: %d\n", queue[front]);

if (front == rear)

front = rear = -1;

else

front++;

}

void displayFrontRear()

{

if (front == -1)

{

printf("Queue is empty.\n");

return;

}

printf("Front element: %d\n", queue[front]);

printf("Rear element: %d\n", queue[rear]);

}

int main()

{

int choice, value;

do {

printf("\nQueue Operations:\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display Front & Rear\n");

printf("4. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(value);

break;

case 2:

dequeue();

break;

case 3:

displayFrontRear();

break;

case 4:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

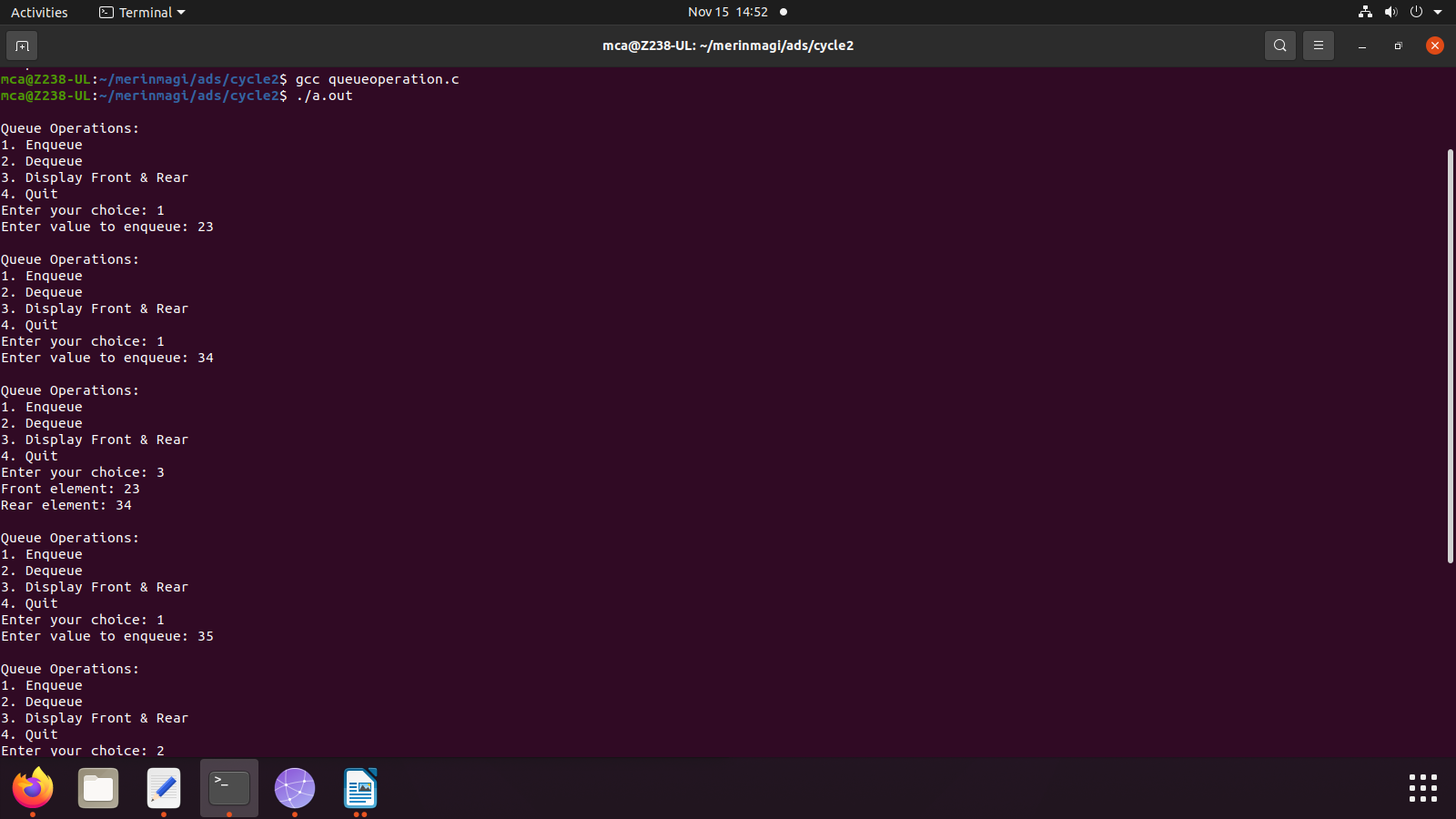
break;

}

} while (choice != 4);

return 0;

}



2) Implement Circular Queue operations (Insert, delete, display front & rear values)

**Source code**

#include <stdio.h>

#define MAX\_SIZE 100

int circularQueue[MAX\_SIZE];

int front = -1, rear = -1;

void enqueue(int value) {

if ((rear + 1) % MAX\_SIZE == front) {

printf("Queue is full. Cannot enqueue.\n");

return;

}

if (front == -1)

front = 0;

rear = (rear + 1) % MAX\_SIZE;

circularQueue[rear] = value;

}

void dequeue() {

if (front == -1) {

printf("Queue is empty. Cannot dequeue.\n");

return;

}

printf("Dequeued element: %d\n", circularQueue[front]);

if (front == rear)

front = rear = -1;

else

front = (front + 1) % MAX\_SIZE;

}

void displayFrontRear() {

if (front == -1) {

printf("Queue is empty.\n");

return;

}

printf("Front element: %d\n", circularQueue[front]);

printf("Rear element: %d\n", circularQueue[rear]);

}

int main() {

int choice, value;

do {

printf("\nCircular Queue Operations:\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display Front & Rear\n");

printf("4. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to enqueue: ");

scanf("%d", &value);

enqueue(value);

break;

case 2:

dequeue();

break;

case 3:

displayFrontRear();

break;

case 4:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

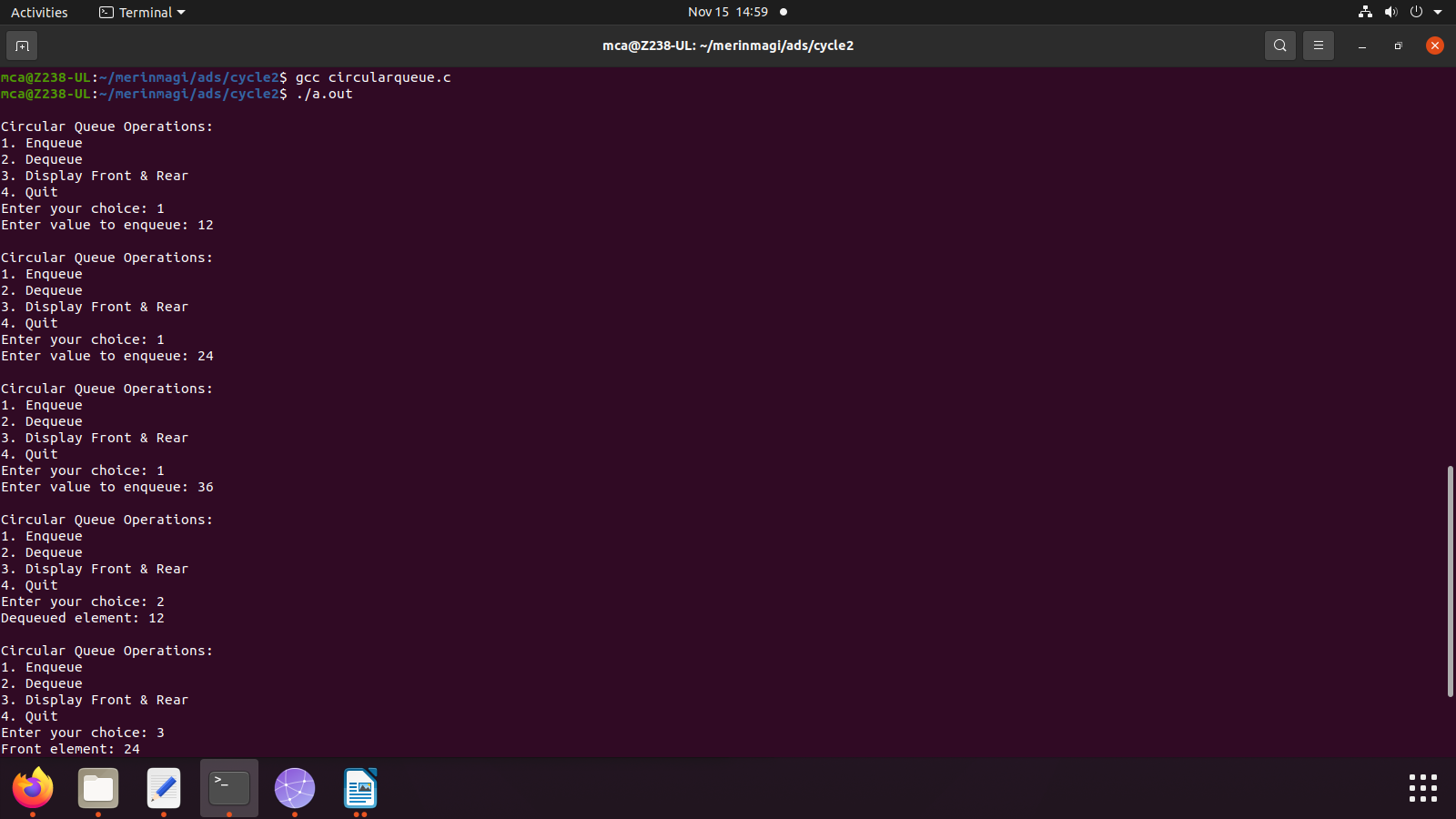
break;

}

} while (choice != 4);

return 0;

}



3) Implement singly linked list (Insert at the head, insert at tail, insert at a position, delete at the

head, delete at tail, delete form a position, search an element).

**Source code**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

void insertAtHead(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = head;

head = newNode;

}

void insertAtTail(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = NULL;

if (head == NULL) {

head = newNode;

return;

}

struct Node\* temp = head;

while (temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

void insertAtPosition(int value, int position) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if (position == 1) {

newNode->next = head;

head = newNode;

return;

}

struct Node\* temp = head;

for (int i = 1; i < position - 1 && temp != NULL; i++)

temp = temp->next;

if (temp == NULL) {

printf("Invalid position.\n");

return;

}

newNode->next = temp->next;

temp->next = newNode;

}

void deleteAtHead() {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

struct Node\* temp = head;

head = head->next;

free(temp);

}

void deleteAtTail() {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

if (head->next == NULL) {

free(head);

head = NULL;

return;

}

struct Node\* temp = head;

while (temp->next->next != NULL)

temp = temp->next;

free(temp->next);

temp->next = NULL;

}

void deleteAtPosition(int position) {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

if (position == 1) {

struct Node\* temp = head;

head = head->next;

free(temp);

return;

}

struct Node\* temp = head;

for (int i = 1; i < position - 1 && temp != NULL; i++)

temp = temp->next;

if (temp == NULL || temp->next == NULL) {

printf("Invalid position.\n");

return;

}

struct Node\* toDelete = temp->next;

temp->next = temp->next->next;

free(toDelete);

}

void search(int value) {

struct Node\* temp = head;

int position = 1;

while (temp != NULL) {

if (temp->data == value) {

printf("%d found at position %d.\n", value, position);

return;

}

temp = temp->next;

position++;

}

printf("%d not found in the list.\n", value);

}

void display() {

struct Node\* temp = head;

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

}

int main() {

int choice, value, position;

do {

printf("\nSingly Linked List Operations:\n");

printf("1. Insert at Head\n");

printf("2. Insert at Tail\n");

printf("3. Insert at Position\n");

printf("4. Delete at Head\n");

printf("5. Delete at Tail\n");

printf("6. Delete at Position\n");

printf("7. Search\n");

printf("8. Display\n");

printf("9. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

insertAtHead(value);

break;

case 2:

printf("Enter value to insert: ");

scanf("%d", &value);

insertAtTail(value);

break;

case 3:

printf("Enter value to insert: ");

scanf("%d", &value);

printf("Enter position: ");

scanf("%d", &position);

insertAtPosition(value, position);

break;

case 4:

deleteAtHead();

break;

case 5:

deleteAtTail();

break;

case 6:

printf("Enter position: ");

scanf("%d", &position);

deleteAtPosition(position);

break;

case 7:

printf("Enter value to search: ");

scanf("%d", &value);

search(value);

break;

case 8:

display();

break;

case 9:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

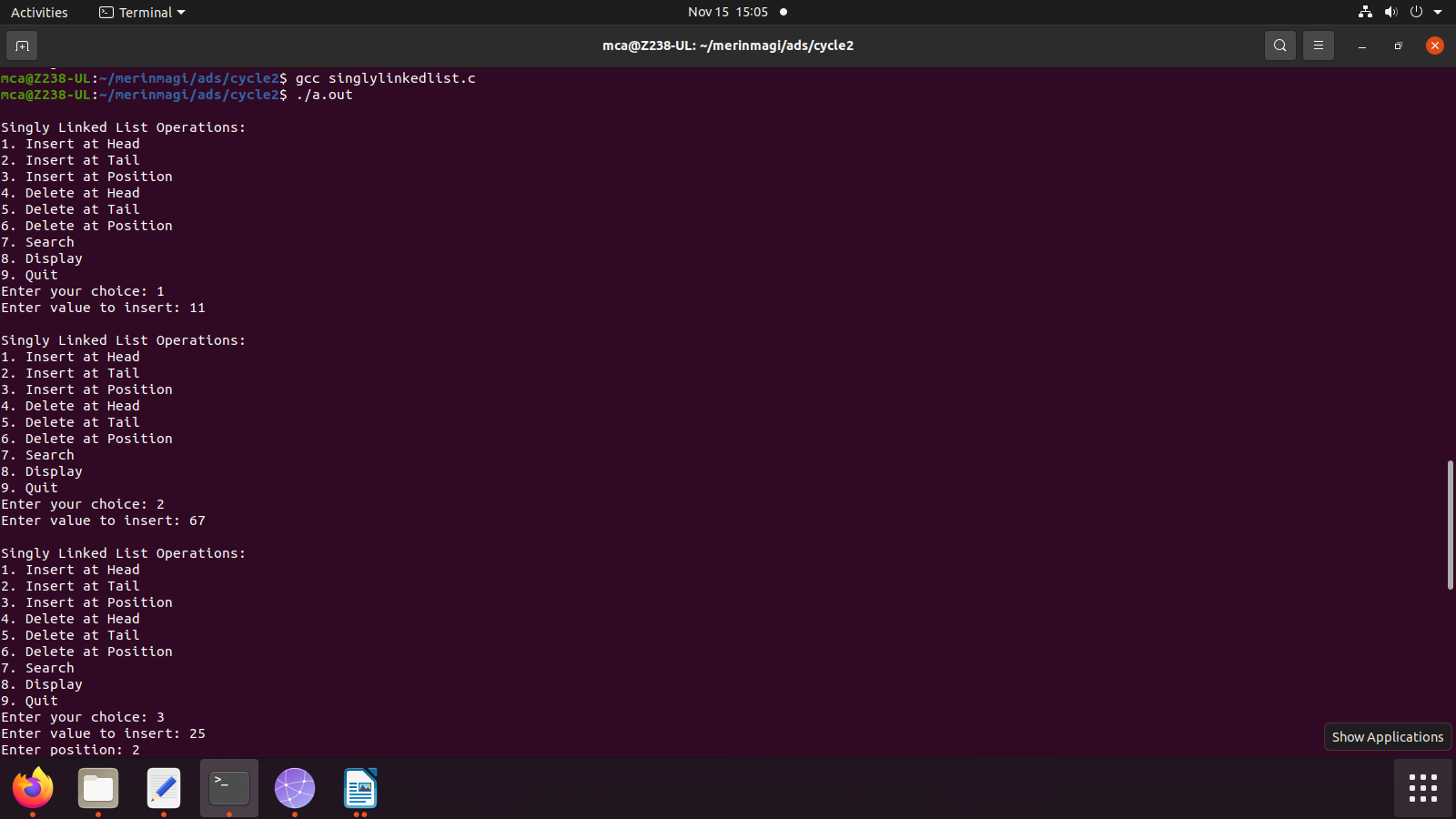
break;

}

} while (choice != 9);

return 0;

}



4) Implement doubly linked list (Insert at the head, insert at tail, insert at a position, delete at the

head, delete at tail, delete form a position, search an element).

**Source code**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

struct Node\* head = NULL;

void insertAtHead(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = head;

newNode->prev = NULL;

if (head != NULL)

head->prev = newNode;

head = newNode;

}

void insertAtTail(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = NULL;

if (head == NULL) {

newNode->prev = NULL;

head = newNode;

return;

}

struct Node\* temp = head;

while (temp->next != NULL)

temp = temp->next;

temp->next = newNode;

newNode->prev = temp;

}

void insertAtPosition(int value, int position) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if (position == 1) {

newNode->next = head;

newNode->prev = NULL;

if (head != NULL)

head->prev = newNode;

head = newNode;

return;

}

struct Node\* temp = head;

for (int i = 1; i < position - 1 && temp != NULL; i++)

temp = temp->next;

if (temp == NULL) {

printf("Invalid position.\n");

return;

}

newNode->next = temp->next;

newNode->prev = temp;

if (temp->next != NULL)

temp->next->prev = newNode;

temp->next = newNode;

}

void deleteAtHead() {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

struct Node\* temp = head;

head = head->next;

if (head != NULL)

head->prev = NULL;

free(temp);

}

void deleteAtTail() {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

if (head->next == NULL) {

free(head);

head = NULL;

return;

}

struct Node\* temp = head;

while (temp->next != NULL)

temp = temp->next;

temp->prev->next = NULL;

free(temp);

}

void deleteAtPosition(int position) {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

if (position == 1) {

struct Node\* temp = head;

head = head->next;

if (head != NULL)

head->prev = NULL;

free(temp);

return;

}

struct Node\* temp = head;

for (int i = 1; i < position - 1 && temp != NULL; i++)

temp = temp->next;

if (temp == NULL || temp->next == NULL) {

printf("Invalid position.\n");

return;

}

struct Node\* toDelete = temp->next;

temp->next = temp->next->next;

if (temp->next != NULL)

temp->next->prev = temp;

free(toDelete);

}

void search(int value) {

struct Node\* temp = head;

int position = 1;

while (temp != NULL) {

if (temp->data == value) {

printf("%d found at position %d.\n", value, position);

return;

}

temp = temp->next;

position++;

}

printf("%d not found in the list.\n", value);

}

void display() {

struct Node\* temp = head;

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

}

int main() {

int choice, value, position;

do {

printf("\nDoubly Linked List Operations:\n");

printf("1. Insert at Head\n");

printf("2. Insert at Tail\n");

printf("3. Insert at Position\n");

printf("4. Delete at Head\n");

printf("5. Delete at Tail\n");

printf("6. Delete at Position\n");

printf("7. Search\n");

printf("8. Display\n");

printf("9. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

insertAtHead(value);

break;

case 2:

printf("Enter value to insert: ");

scanf("%d", &value);

insertAtTail(value);

break;

case 3:

printf("Enter value to insert: ");

scanf("%d", &value);

printf("Enter position: ");

scanf("%d", &position);

insertAtPosition(value, position);

break;

case 4:

deleteAtHead();

break;

case 5:

deleteAtTail();

break;

case 6:

printf("Enter position: ");

scanf("%d", &position);

deleteAtPosition(position);

break;

case 7:

printf("Enter value to search: ");

scanf("%d", &value);

search(value);

break;

case 8:

display();

break;

case 9:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

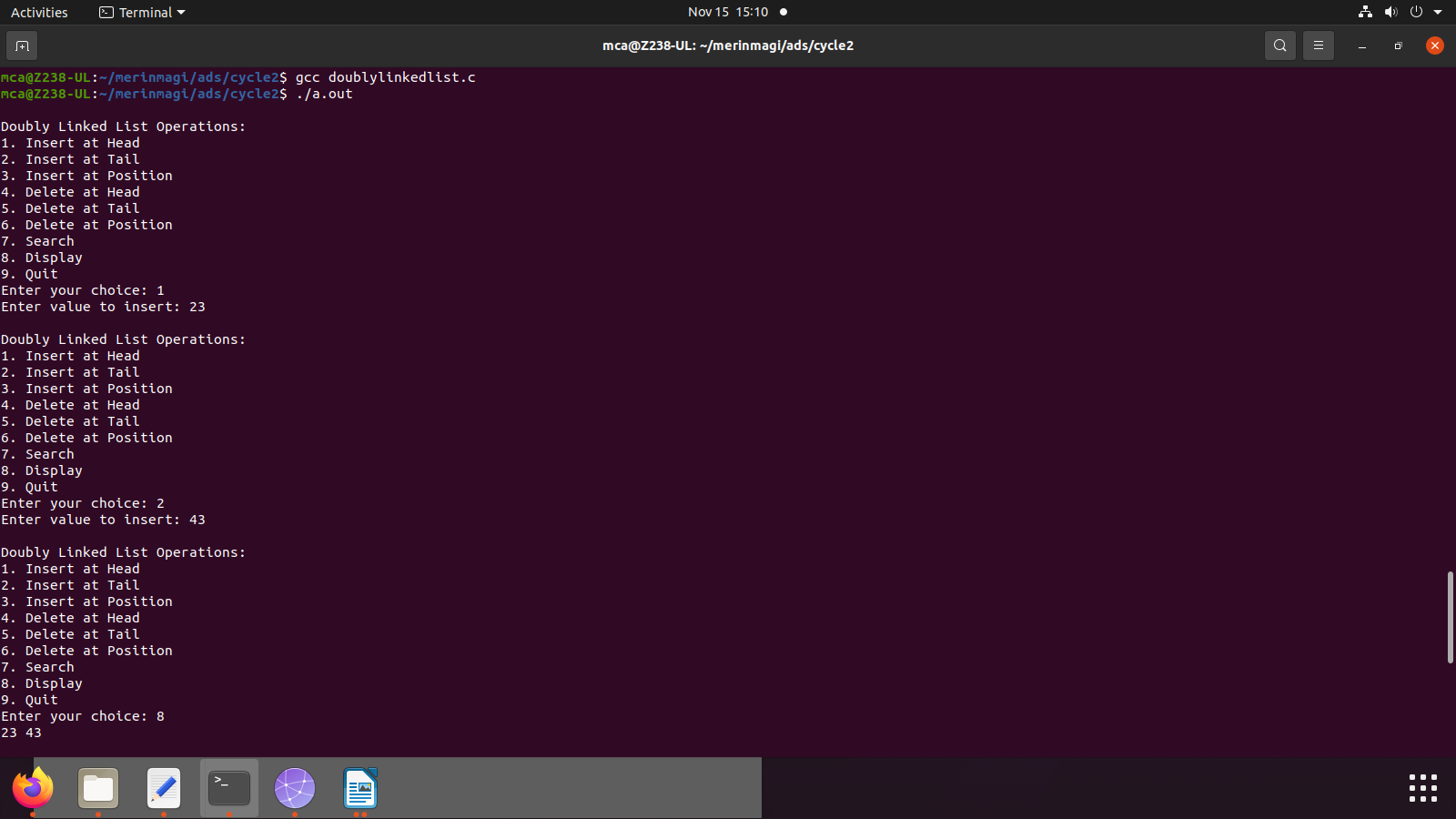
break;

}

} while (choice != 9);

return 0;

}



5) Implement circular linked list (Insert at the head, insert at tail, insert at a position, delete at the

head, delete at tail, delete form a position, search an element)

Source code

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

void insertAtHead(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if (head == NULL) {

newNode->next = newNode;

head = newNode;

return;

}

struct Node\* temp = head;

while (temp->next != head)

temp = temp->next;

temp->next = newNode;

newNode->next = head;

head = newNode;

}

void insertAtTail(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if (head == NULL) {

newNode->next = newNode;

head = newNode;

return;

}

struct Node\* temp = head;

while (temp->next != head)

temp = temp->next;

temp->next = newNode;

newNode->next = head;

}

void insertAtPosition(int value, int position) {

if (position == 1) {

insertAtHead(value);

return;

}

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

struct Node\* temp = head;

for (int i = 1; i < position - 1 && temp->next != head; i++)

temp = temp->next;

newNode->next = temp->next;

temp->next = newNode;

}

void deleteAtHead() {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

if (head->next == head) {

free(head);

head = NULL;

return;

}

struct Node\* temp = head;

while (temp->next != head)

temp = temp->next;

temp->next = head->next;

struct Node\* toDelete = head;

head = head->next;

free(toDelete);

}

void deleteAtTail() {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

if (head->next == head) {

free(head);

head = NULL;

return;

}

struct Node\* temp = head;

while (temp->next->next != head)

temp = temp->next;

struct Node\* toDelete = temp->next;

temp->next = head;

free(toDelete);

}

void deleteAtPosition(int position) {

if (head == NULL) {

printf("List is empty. Cannot delete.\n");

return;

}

if (position == 1) {

deleteAtHead();

return;

}

struct Node\* temp = head;

for (int i = 1; i < position - 1 && temp->next != head; i++)

temp = temp->next;

if (temp->next == head) {

printf("Invalid position.\n");

return;

}

struct Node\* toDelete = temp->next;

temp->next = temp->next->next;

free(toDelete);

}

void search(int value) {

struct Node\* temp = head;

int position = 1;

do {

if (temp->data == value) {

printf("%d found at position %d.\n", value, position);

return;

}

temp = temp->next;

position++;

} while (temp != head);

printf("%d not found in the list.\n", value);

}

void display() {

if (head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = head;

do {

printf("%d ", temp->data);

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

int choice, value, position;

do {

printf("\nCircular Linked List Operations:\n");

printf("1. Insert at Head\n");

printf("2. Insert at Tail\n");

printf("3. Insert at Position\n");

printf("4. Delete at Head\n");

printf("5. Delete at Tail\n");

printf("6. Delete at Position\n");

printf("7. Search\n");

printf("8. Display\n");

printf("9. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

insertAtHead(value);

break;

case 2:

printf("Enter value to insert: ");

scanf("%d", &value);

insertAtTail(value);

break;

case 3:

printf("Enter value to insert: ");

scanf("%d", &value);

printf("Enter position: ");

scanf("%d", &position);

insertAtPosition(value, position);

break;

case 4:

deleteAtHead();

break;

case 5:

deleteAtTail();

break;

case 6:

printf("Enter position: ");

scanf("%d", &position);

deleteAtPosition(position);

break;

case 7:

printf("Enter value to search: ");

scanf("%d", &value);

search(value);

break;

case 8:

display();

break;

case 9:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

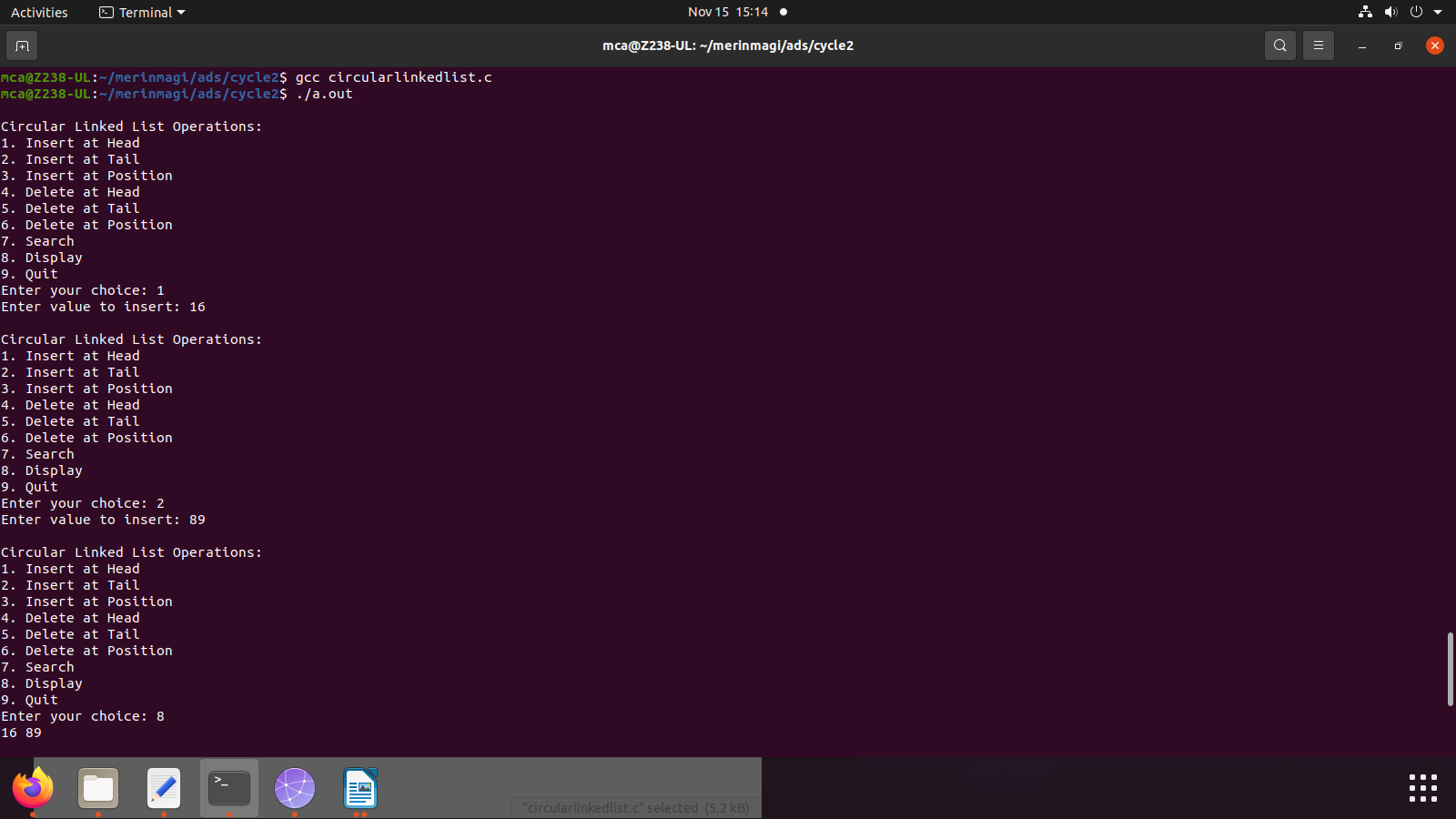
break;

}

} while (choice != 9);

return 0;

}



6) Implement binary search tree

source code

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Node {

int data;

int height;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->height = 1;

newNode->left = newNode->right = NULL;

return newNode;

}

//calculate height for rotation.

int calculateHeight(struct Node\* node) {

int leftHeight = (node->left) ? node->left->height : 0;

int rightHeight = (node->right) ? node->right->height : 0;

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

}

//update height as it rotates.

void updateHeight(struct Node\* node) {

node->height = calculateHeight(node);

}

//rotate node to right and update height

struct Node\* rightRotate(struct Node\* y) {

struct Node\* x = y->left;

struct Node\* T = x->right;

x->right = y;

y->left = T;

updateHeight(y);

updateHeight(x);

return x;

}

//rotate node to left and update height

struct Node\* leftRotate(struct Node\* x) {

struct Node\* y = x->right;

struct Node\* T = y->left;

y->left = x;

x->right = T;

updateHeight(x);

updateHeight(y);

return y;

}

//check for balance and update nodes if not-balanced

struct Node\* balanceNode(struct Node\* node) {

int balance = (node->left ? node->left->height : 0) - (node->right ? node->right->height : 0);

if (balance > 1) {

if (node->left->left) {

return rightRotate(node);

} else {

node->left = leftRotate(node->left);

return rightRotate(node);

}

}

if (balance < -1) {

if (node->right->right) {

return leftRotate(node);

} else {

node->right = rightRotate(node->right);

return leftRotate(node);

}

}

return node;

}

//adding new nodes to the tree and thus updating height and check balance

struct Node\* insert(struct Node\* root, int value) {

if (root == NULL) {

struct Node\* newNode = createNode(value);

return newNode;

}

if (value < root->data)

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

else if (value > root->data)

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

updateHeight(root);

return balanceNode(root);

}

//find the min (new node) for attaching after deleted a node

struct Node\* findMin(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

//delete node and update values accordingly ( reccursion )

struct Node\* deleteNode(struct Node\* root, int value) {

if (root == NULL)

return root;

if (value < root->data)

root->left = deleteNode(root->left, value);

else if (value > root->data)

root->right = deleteNode(root->right, value);

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 = findMin(root->right);

root->data = temp->data;

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

}

updateHeight(root);

return balanceNode(root);

}

//searching for a node ( based on value )

struct Node\* search(struct Node\* root, int value) {

if (root == NULL || root->data == value)

return root;

if (value < root->data)

return search(root->left, value);

return search(root->right, value);

}

//traverse through ( display ) all existing nodes ( use in-order as default )

void inOrderTraversal(struct Node\* root) {

if (root != NULL) {

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

int main() {

struct Node\* root = NULL;

int choice, value;

do {

printf("\nBalanced Binary Search Tree Operations:\n");

printf("1. Insert\n");

printf("2. Delete\n");

printf("3. Search\n");

printf("4. In-order Traversal\n");

printf("5. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

if (search(root, value) == NULL)

root = insert(root, value);

else

printf("%d is already in the tree.\n", value);

break;

case 2:

printf("Enter value to delete: ");

scanf("%d", &value);

if (search(root, value) != NULL)

root = deleteNode(root, value);

else

printf("%d not found in the tree.\n", value);

break;

case 3:

printf("Enter value to search: ");

scanf("%d", &value);

struct Node\* result = search(root, value);

if (result != NULL)

printf("%d found in the tree.\n", value);

else

printf("%d not found in the tree.\n", value);

break;

case 4:

printf("In-order Traversal: ");

inOrderTraversal(root);

printf("\n");

break;

case 5:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

break;

}

} while (choice != 5);

// Free memory (cleanup)

while (root != NULL) {

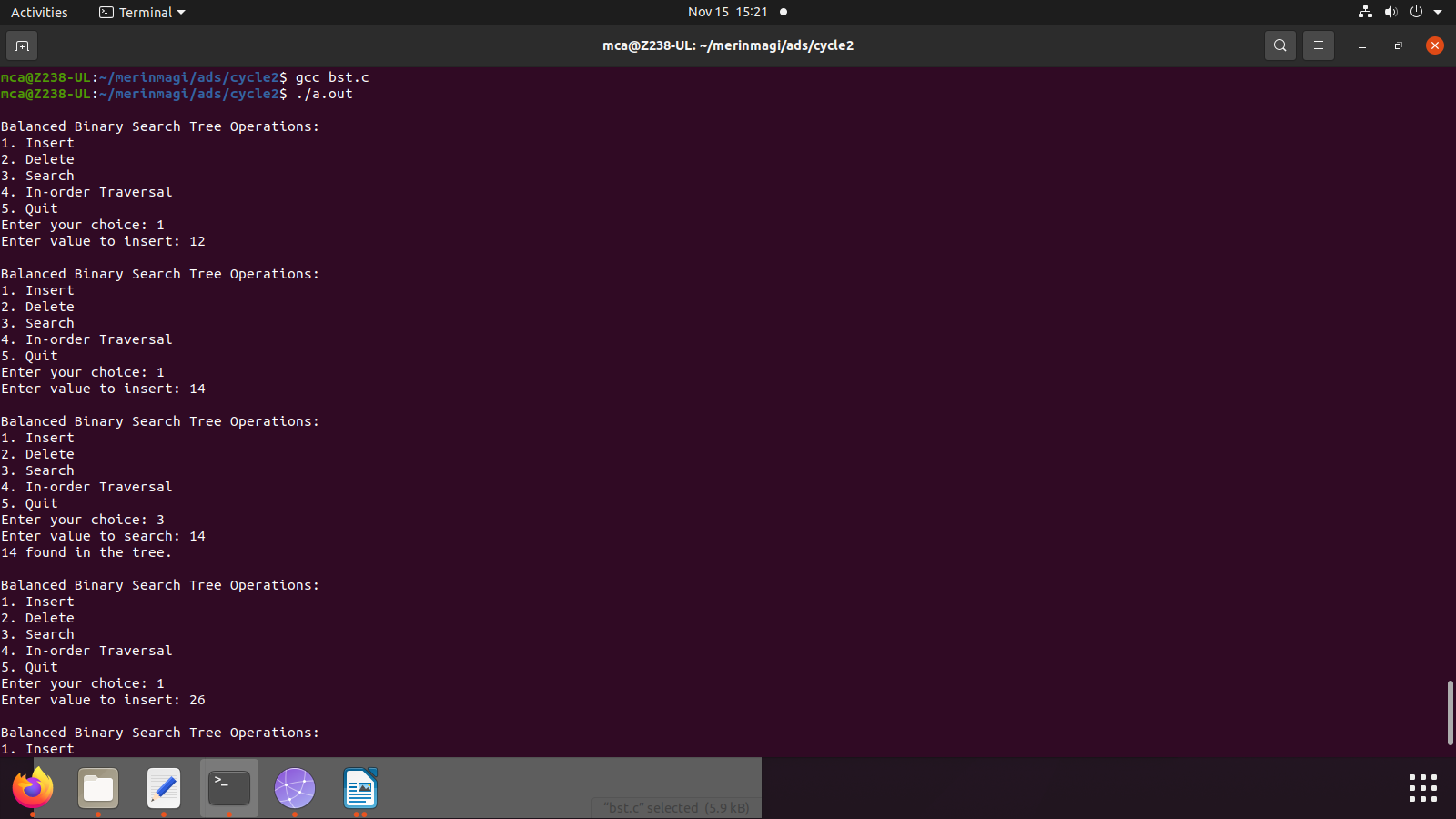
struct Node\* temp = root;

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

}

return 0;

}



7) Implement balanced-binary-search tree

source code

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Node {

int data;

int height;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->height = 1;

newNode->left = newNode->right = NULL;

return newNode;

}

//calculate height for rotation.

int calculateHeight(struct Node\* node) {

int leftHeight = (node->left) ? node->left->height : 0;

int rightHeight = (node->right) ? node->right->height : 0;

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

}

//update height as it rotates.

void updateHeight(struct Node\* node) {

node->height = calculateHeight(node);

}

//rotate node to right and update height

struct Node\* rightRotate(struct Node\* y) {

struct Node\* x = y->left;

struct Node\* T = x->right;

x->right = y;

y->left = T;

updateHeight(y);

updateHeight(x);

return x;

}

//rotate node to left and update height

struct Node\* leftRotate(struct Node\* x) {

struct Node\* y = x->right;

struct Node\* T = y->left;

y->left = x;

x->right = T;

updateHeight(x);

updateHeight(y);

return y;

}

//check for balance and update nodes if not-balanced

struct Node\* balanceNode(struct Node\* node) {

int balance = (node->left ? node->left->height : 0) - (node->right ? node->right->height : 0);

if (balance > 1) {

if (node->left->left) {

return rightRotate(node);

} else {

node->left = leftRotate(node->left);

return rightRotate(node);

}

}

if (balance < -1) {

if (node->right->right) {

return leftRotate(node);

} else {

node->right = rightRotate(node->right);

return leftRotate(node);

}

}

return node;

}

//adding new nodes to the tree and thus updating height and check balance

struct Node\* insert(struct Node\* root, int value) {

if (root == NULL) {

struct Node\* newNode = createNode(value);

return newNode;

}

if (value < root->data)

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

else if (value > root->data)

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

updateHeight(root);

return balanceNode(root);

}

//find the min (new node) for attaching after deleted a node

struct Node\* findMin(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

//delete node and update values accordingly ( reccursion )

struct Node\* deleteNode(struct Node\* root, int value) {

if (root == NULL)

return root;

if (value < root->data)

root->left = deleteNode(root->left, value);

else if (value > root->data)

root->right = deleteNode(root->right, value);

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 = findMin(root->right);

root->data = temp->data;

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

}

updateHeight(root);

return balanceNode(root);

}

//searching for a node ( based on value )

struct Node\* search(struct Node\* root, int value) {

if (root == NULL || root->data == value)

return root;

if (value < root->data)

return search(root->left, value);

return search(root->right, value);

}

//traverse through ( display ) all existing nodes ( use in-order as default )

void inOrderTraversal(struct Node\* root) {

if (root != NULL) {

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

int main() {

struct Node\* root = NULL;

int choice, value;

do {

printf("\nBalanced Binary Search Tree Operations:\n");

printf("1. Insert\n");

printf("2. Delete\n");

printf("3. Search\n");

printf("4. In-order Traversal\n");

printf("5. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

if (search(root, value) == NULL)

root = insert(root, value);

else

printf("%d is already in the tree.\n", value);

break;

case 2:

printf("Enter value to delete: ");

scanf("%d", &value);

if (search(root, value) != NULL)

root = deleteNode(root, value);

else

printf("%d not found in the tree.\n", value);

break;

case 3:

printf("Enter value to search: ");

scanf("%d", &value);

struct Node\* result = search(root, value);

if (result != NULL)

printf("%d found in the tree.\n", value);

else

printf("%d not found in the tree.\n", value);

break;

case 4:

printf("In-order Traversal: ");

inOrderTraversal(root);

printf("\n");

break;

case 5:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

break;

}

} while (choice != 5);

// Free memory (cleanup)

while (root != NULL) {

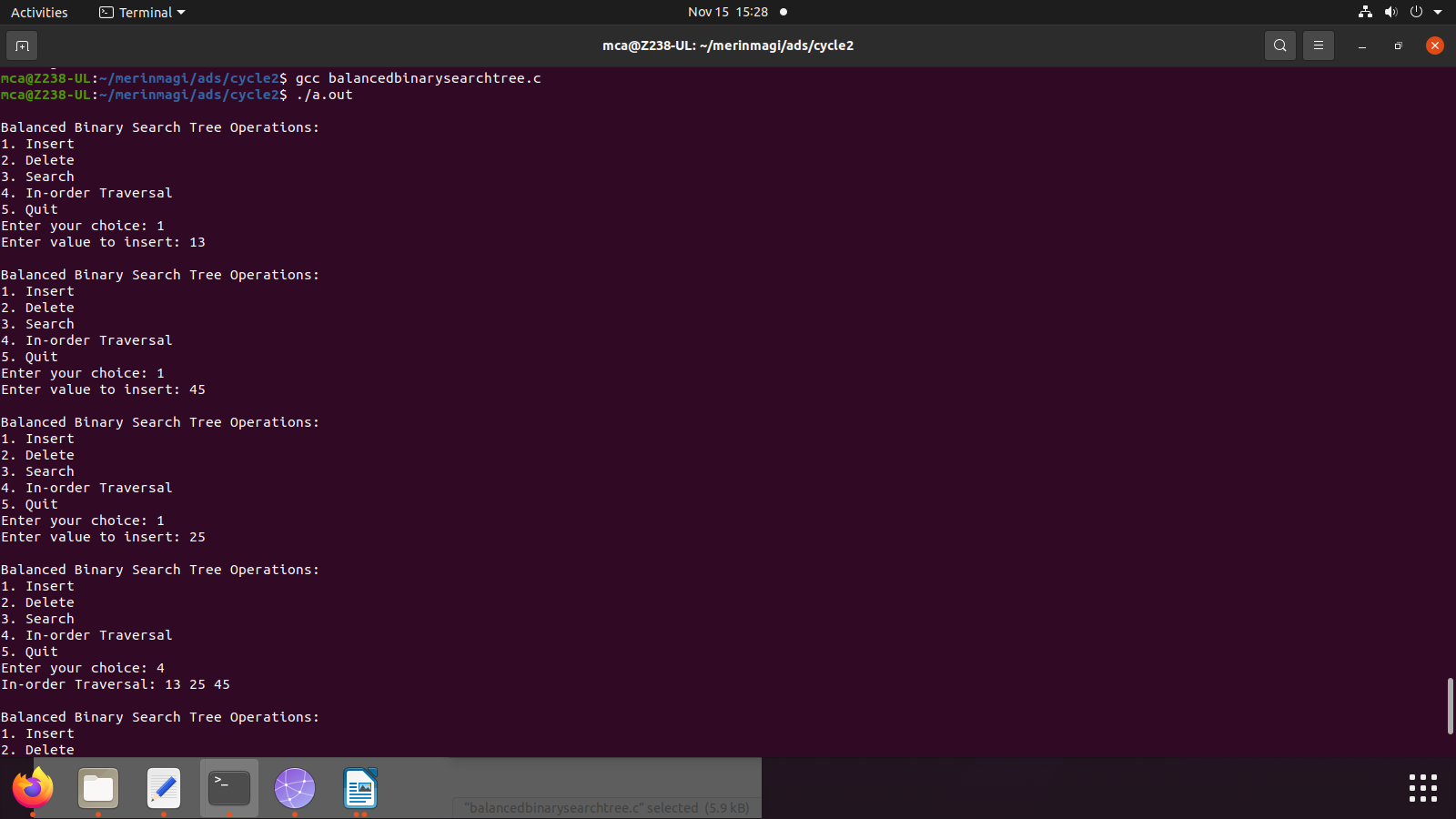
struct Node\* temp = root;

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

}

return 0;

}



8) Implement set operations (union, intersection, difference)

source code

#include <stdio.h>

#define MAX\_SIZE 100

// Function to read elements of a set from the user

int readSet(int set[]) {

int size, i;

printf("Enter the size of the set: ");

scanf("%d", &size);

printf("Enter elements of the set:\n");

for (i = 0; i < size; i++) {

scanf("%d", &set[i]);

}

return size;

}

// Function to display the elements of a set

void displaySet(int set[], int size) {

int i;

printf("Set: { ");

for (i = 0; i < size; i++) {

printf("%d ", set[i]);

}

printf("}\n");

}

// Function to perform set union

int setUnion(int set1[], int size1, int set2[], int size2, int result[]) {

int i, j, k = 0;

for (i = 0; i < size1; i++) {

result[k++] = set1[i];

}

for (i = 0; i < size2; i++) {

int found = 0;

for (j = 0; j < size1; j++) {

if (set2[i] == set1[j]) {

found = 1;

break;

}

}

if (!found) {

result[k++] = set2[i];

}

}

return k;

}

// Function to perform set intersection

int setIntersection(int set1[], int size1, int set2[], int size2, int result[]) {

int i, j, k = 0;

for (i = 0; i < size1; i++) {

for (j = 0; j < size2; j++) {

if (set1[i] == set2[j]) {

result[k++] = set1[i];

break;

}

}

}

return k;

}

// Function to perform set difference (set1 - set2)

int setDifference(int set1[], int size1, int set2[], int size2, int result[]) {

int i, j, k = 0;

for (i = 0; i < size1; i++) {

int found = 0;

for (j = 0; j < size2; j++) {

if (set1[i] == set2[j]) {

found = 1;

break;

}

}

if (!found) {

result[k++] = set1[i];

}

}

return k;

}

int main() {

int set1[MAX\_SIZE], set2[MAX\_SIZE], result[MAX\_SIZE];

int size1, size2, resultSize;

int choice;

size1 = readSet(set1);

size2 = readSet(set2);

do {

printf("\nSet Operations:\n");

printf("1. Union\n");

printf("2. Intersection\n");

printf("3. Difference (set1 - set2)\n");

printf("4. Display Sets\n");

printf("5. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

resultSize = setUnion(set1, size1, set2, size2, result);

displaySet(result, resultSize);

break;

case 2:

resultSize = setIntersection(set1, size1, set2, size2, result);

displaySet(result, resultSize);

break;

case 3:

resultSize = setDifference(set1, size1, set2, size2, result);

displaySet(result, resultSize);

break;

case 4:

printf("\nSets:\n");

printf("Set 1: ");

displaySet(set1, size1);

printf("Set 2: ");

displaySet(set2, size2);

break;

case 5:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

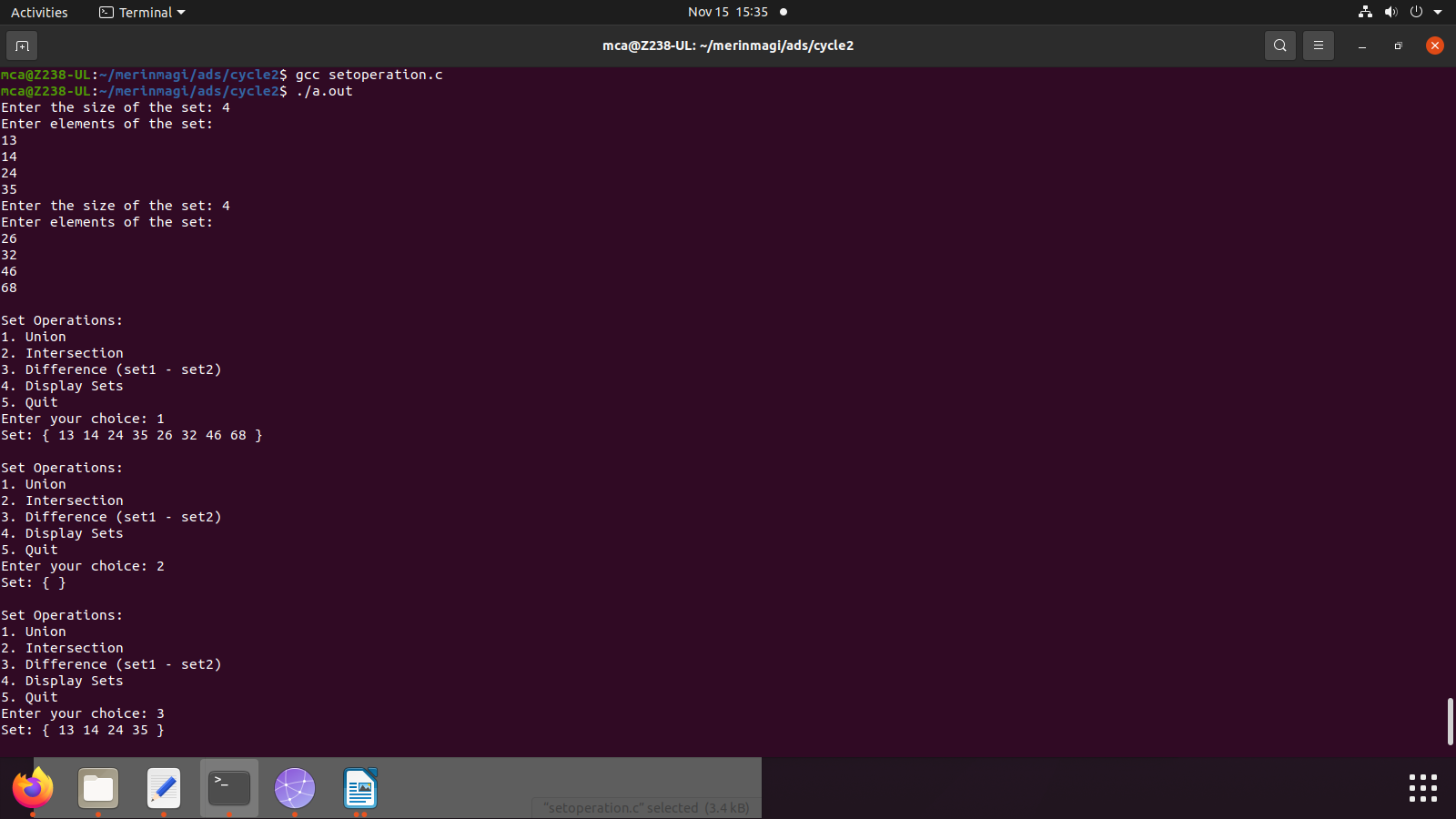
break;

}

} while (choice != 5);

return 0;

}



9) Implement disjoint set operations.

Source code

#include <stdio.h>

#define MAX 100

int parent[MAX], rank[MAX], n;

// Function to find the representative of the set containing element x

int find(int x) {

if (x != parent[x])

parent[x] = find(parent[x]); // Path Compression

return parent[x];

}

int main() {

printf("Enter the number of elements: ");

if (scanf("%d", &n) != 1 || n <= 0 || n > MAX) {

printf("Invalid input. Please enter a positive integer less than or equal to %d.\n", MAX);

return 1;

}

// Initialize sets

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

parent[i] = i;

rank[i] = 0;

}

int choice, x, y;

while (1) {

printf("\nOperations:\n1. Union\n2. Find\n3. Display Set Representatives\n4. Exit\nEnter your choice: ");

if (scanf("%d", &choice) != 1) {

printf("Invalid input. Please enter an integer.\n");

continue;

}

switch (choice) {

case 1:

// Union operation

printf("Enter elements to perform union: ");

if (scanf("%d %d", &x, &y) != 2 || x < 0 || x >= n || y < 0 || y >= n) {

printf("Invalid input. Please enter valid elements.\n");

} else {

int rootX = find(x);

int rootY = find(y);

if (rootX == rootY) {

printf("%d and %d are already in the same set.\n", x, y);

} else {

// Merge sets

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

printf("Union of %d and %d is performed.\n", x, y);

}

}

break;

case 2:

// Find operation

printf("Enter element to find its set: ");

if (scanf("%d", &x) != 1 || x < 0 || x >= n) {

printf("Invalid input. Please enter a valid element.\n");

} else {

printf("Set representative of %d is %d\n", x, find(x));

}

break;

case 3:

// Display set representatives

printf("Set Representatives:\n");

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

printf("Element %d belongs to set with representative %d\n", i, find(i));

}

break;

case 4:

return 0;

default:

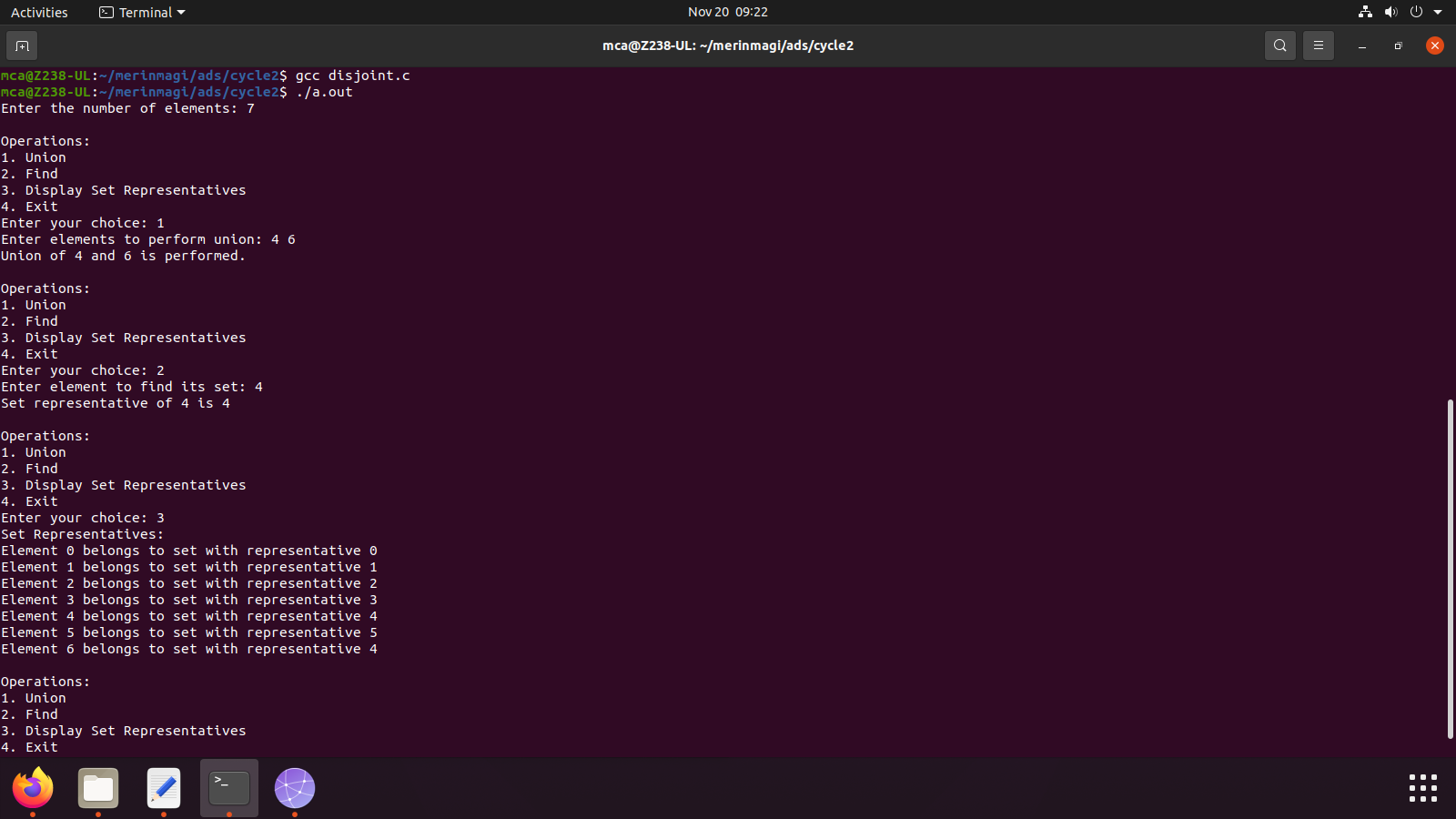
printf("Invalid choice. Please enter a valid option.\n");

break;

}

}

}



10) Implement tree traversal methods DFS ( In-order, Pre-Order, Post-Order ), and BFS

source code

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Binary tree node definition

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new binary tree node

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to perform in-order traversal (DFS)

void inOrderTraversal(struct TreeNode\* root) {

if (root == NULL) return;

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

// Function to perform pre-order traversal (DFS)

void preOrderTraversal(struct TreeNode\* root) {

if (root == NULL) return;

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

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

// Function to perform post-order traversal (DFS)

void postOrderTraversal(struct TreeNode\* root) {

if (root == NULL) return;

postOrderTraversal(root->left);

postOrderTraversal(root->right);

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

}

// Function to perform breadth-first traversal (BFS)

void breadthFirstTraversal(struct TreeNode\* root) {

if (root == NULL) return;

// Create a queue for BFS

struct TreeNode\* queue[100];

int front = 0, rear = 0;

queue[rear++] = root;

while (front < rear) {

struct TreeNode\* currentNode = queue[front++];

printf("%d ", currentNode->data);

if (currentNode->left != NULL) {

queue[rear++] = currentNode->left;

}

if (currentNode->right != NULL) {

queue[rear++] = currentNode->right;

}

}

}

int main() {

// Create a sample binary tree

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->left = createNode(6);

root->right->right = createNode(7);

int choice;

while (1) {

printf("\nMenu:\n");

printf("1. In-Order Traversal\n");

printf("2. Pre-Order Traversal\n");

printf("3. Post-Order Traversal\n");

printf("4. Breadth-First Traversal\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("In-Order Traversal: ");

inOrderTraversal(root);

break;

case 2:

printf("Pre-Order Traversal: ");

preOrderTraversal(root);

break;

case 3:

printf("Post-Order Traversal: ");

postOrderTraversal(root);

break;

case 4:

printf("Breadth-First Traversal: ");

breadthFirstTraversal(root);

break;

case 5:

exit(0);

default:

printf("Invalid choice. Please enter a valid option.\n");

}

}

return 0;

}

