**Source Code:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

typedef struct Node Node;

Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

Node\* insertNode(Node\* head, int value) {

Node\* newNode = createNode(value);

if (head == NULL)

return newNode;

if (value < head->data) {

newNode->next = head;

return newNode;

}

Node\* current = head;

while (current->next != NULL && current->next->data < value)

current = current->next;

newNode->next = current->next;

current->next = newNode;

return head;

}

Node\* mergeLists(Node\* list1, Node\* list2) {

if (list1 == NULL)

return list2;

if (list2 == NULL)

return list1;

Node\* mergedList = NULL;

if (list1->data < list2->data) {

mergedList = list1;

mergedList->next = mergeLists(list1->next, list2);

} else {

mergedList = list2;

mergedList->next = mergeLists(list1, list2->next);

}

return mergedList;

}

void displayList(Node\* head) {

if (head == NULL) {

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

return;

}

Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("NULL\n");

}

void freeList(Node\* head) {

Node\* current = head;

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

Node\* list1 = NULL;

Node\* list2 = NULL;

Node\* mergedList = NULL;

int choice, value;

do {

printf("1. Insert into List 1\n");

printf("2. Insert into List 2\n");

printf("3. Merge Lists\n");

printf("4. Display Merged List\n");

printf("5. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to insert into List 1: ");

scanf("%d", &value);

list1 = insertNode(list1, value);

break;

case 2:

printf("Enter the value to insert into List 2: ");

scanf("%d", &value);

list2 = insertNode(list2, value);

break;

case 3:

mergedList = mergeLists(list1, list2);

printf("Lists merged successfully.\n");

break;

case 4:

printf("Merged List: ");

displayList(mergedList);

break;

case 5:

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

break;

default:

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

break;

}

printf("\n");

} while (choice != 5);

freeList(list1);

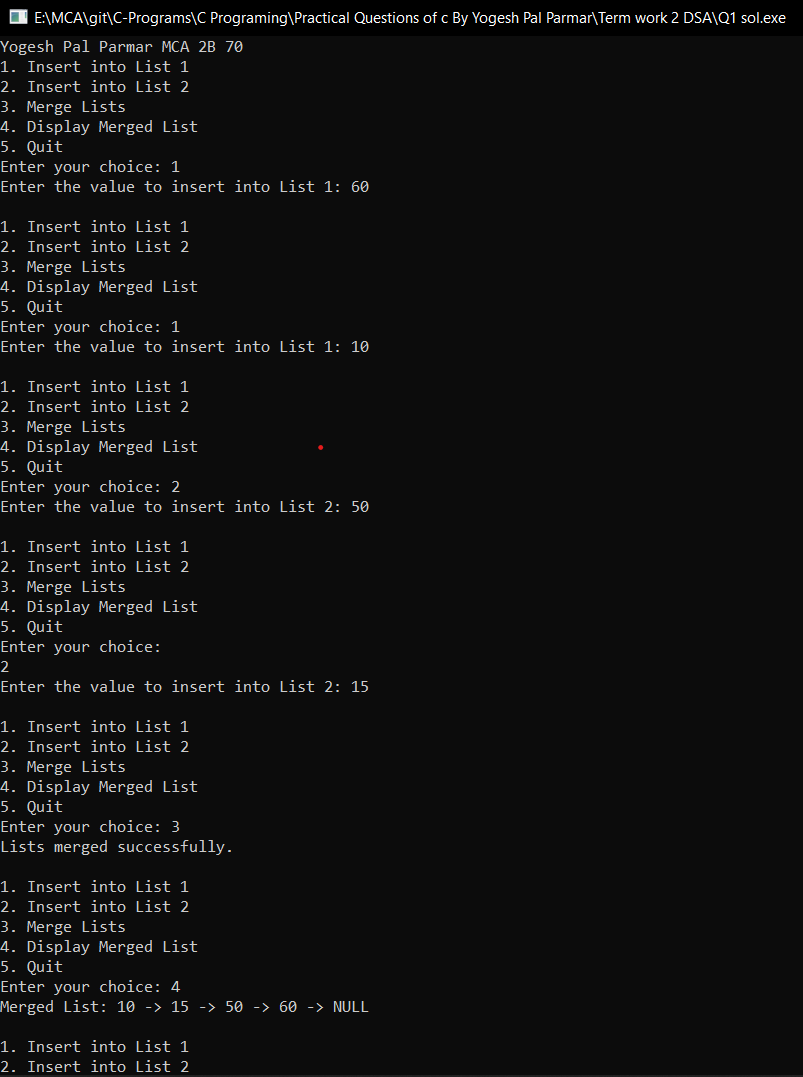
freeList(list2);

freeList(mergedList);

return 0;

}

**Output:**



**Source Code:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

typedef struct Node Node;

Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

Node\* insertNode(Node\* head, int value) {

Node\* newNode = createNode(value);

if (head == NULL)

return newNode;

Node\* current = head;

while (current->next != NULL)

current = current->next;

current->next = newNode;

return head;

}

void insertNth(Node\* head, int value, int position) {

Node\* newNode = createNode(value);

if (head == NULL) {

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

return;

}

Node\* current = head;

int count = 1;

while (current != NULL && count < position) {

current = current->next;

count++;

}

if (current == NULL) {

printf("Invalid position.\n");

return;

}

newNode->next = current->next;

current->next = newNode;

printf("Node inserted successfully.\n");

}

void displayList(Node\* head) {

if (head == NULL) {

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

return;

}

Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("NULL\n");

}

void freeList(Node\* head) {

Node\* current = head;

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

Node\* head = NULL;

int choice, value, position;

do {

printf("1. Insert into List\n");

printf("2. Insert Node After Nth Position\n");

printf("3. Display List\n");

printf("4. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to insert into the list: ");

scanf("%d", &value);

head = insertNode(head, value);

break;

case 2:

printf("Enter the value to insert: ");

scanf("%d", &value);

printf("Enter the position after which to insert: ");

scanf("%d", &position);

insertNth(head, value, position);

break;

case 3:

printf("Linked List: ");

displayList(head);

break;

case 4:

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

break;

default:

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

break;

}

printf("\n");

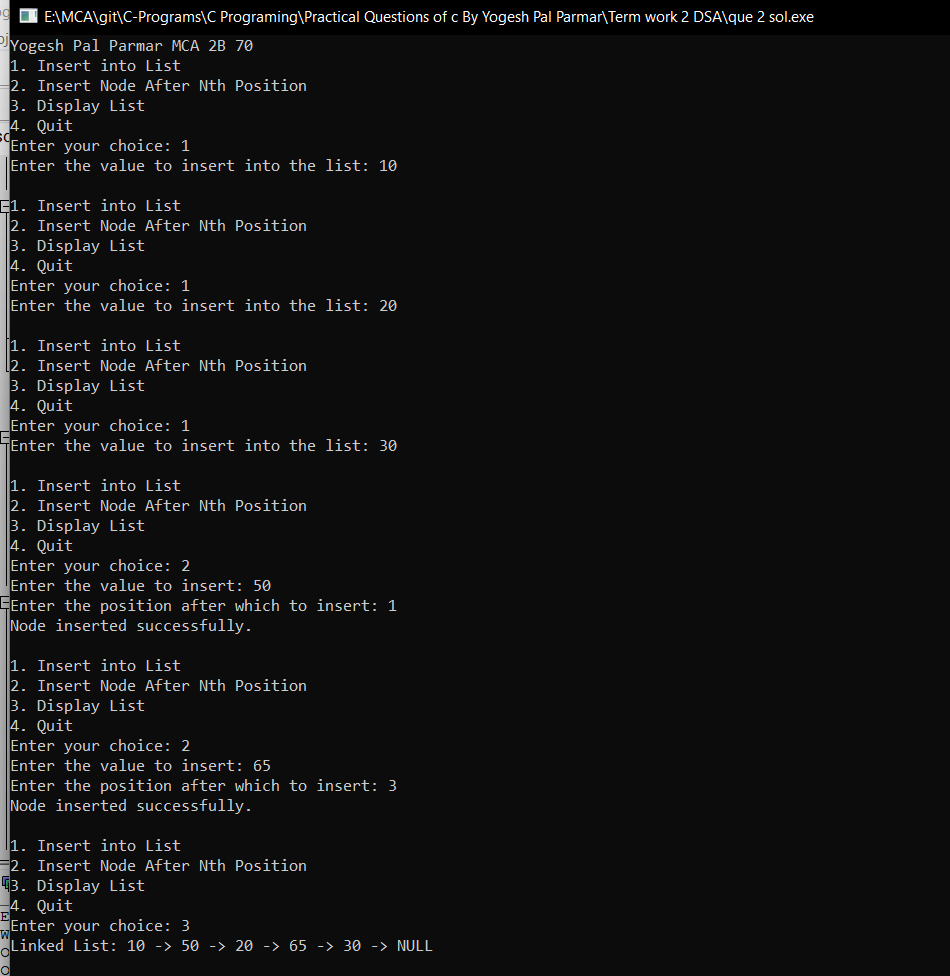
} while (choice != 4);

freeList(head);

return 0;

}

**Output:**



**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Node structure

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to insert a node at the end of the linked list

void insertNode(struct Node\*\* head, int data) {

if (\*head == NULL) {

\*head = createNode(data);

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = createNode(data);

}

// Function to remove duplicate nodes from an unsorted linked list

void removeDuplicates(struct Node\*\* head) {

if (\*head == NULL || (\*head)->next == NULL) {

return;

}

// Hash table to keep track of seen elements

int hash[1000] = {0};

struct Node\* current = \*head;

struct Node\* prev = NULL;

while (current != NULL) {

// Check if current data is already seen

if (hash[current->data] == 1) {

prev->next = current->next;

free(current);

current = prev->next;

} else {

hash[current->data] = 1;

prev = current;

current = current->next;

}

}

}

// Function to print the linked list

void printList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70");

struct Node\* head = NULL;

int choice, data;

while (1) {

printf("1. Insert a node\n");

printf("2. Remove duplicate nodes\n");

printf("3. Print the list\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the data: ");

scanf("%d", &data);

insertNode(&head, data);

break;

case 2:

removeDuplicates(&head);

printf("Duplicate nodes removed.\n");

break;

case 3:

printf("Linked List: ");

printList(head);

break;

case 4:

printf("Exiting program.\n");

exit(0);

default:

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

}

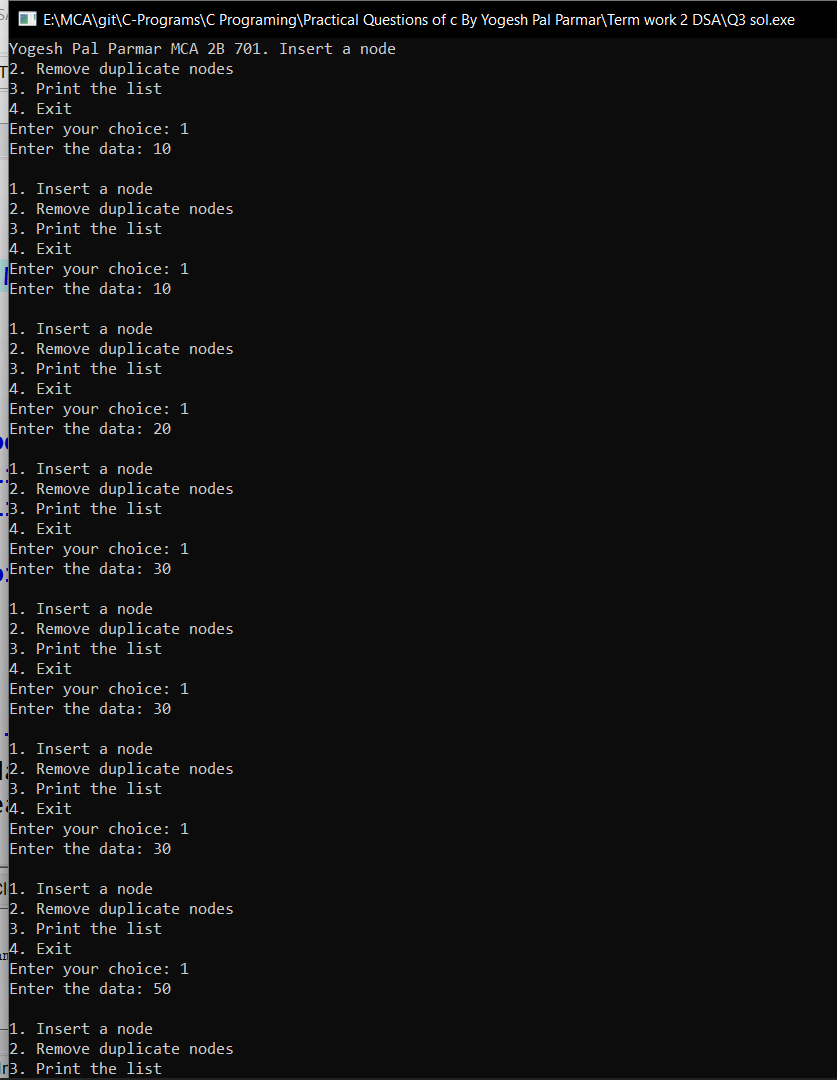
printf("\n");

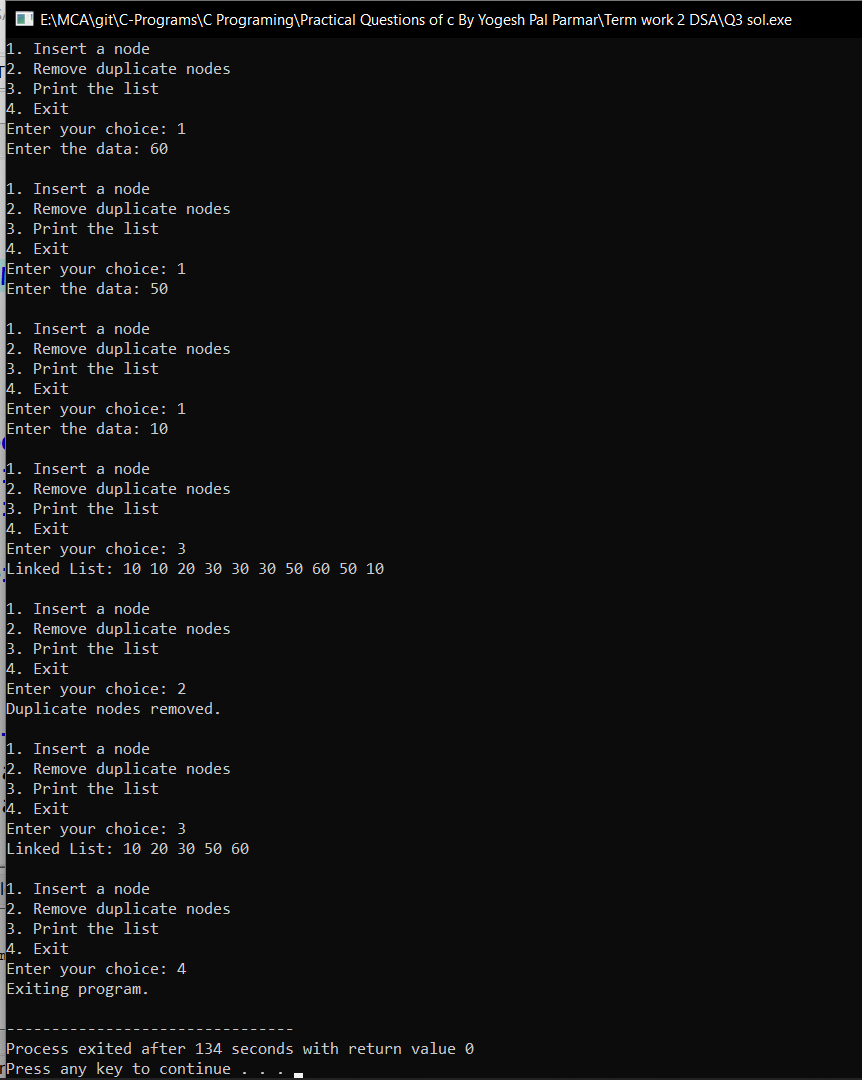
}

return 0;

}

**Output:**





**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Structure for a node

struct Node {

int data;

struct Node\* next;

};

// Function to insert a node at the beginning of a linked list

void insert(struct Node\*\* head, int data) {

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

newNode->data = data;

newNode->next = \*head;

\*head = newNode;

}

// Function to split the linked list into two lists

void split(struct Node\* P, struct Node\*\* Q, struct Node\*\* R) {

struct Node\* currentP = P;

struct Node\* currentQ = NULL;

struct Node\* currentR = NULL;

int position = 1;

while (currentP != NULL) {

if (position % 2 == 1) {

// Insert the node into Q (odd positions)

if (\*Q == NULL) {

\*Q = currentP;

currentQ = \*Q;

} else {

currentQ->next = currentP;

currentQ = currentQ->next;

}

} else {

// Insert the node into R (even positions)

if (\*R == NULL) {

\*R = currentP;

currentR = \*R;

} else {

currentR->next = currentP;

currentR = currentR->next;

}

}

position++;

currentP = currentP->next;

}

// Set the next pointers of the last nodes in Q and R as NULL

if (currentQ != NULL)

currentQ->next = NULL;

if (currentR != NULL)

currentR->next = NULL;

}

// Function to print a linked list

void printList(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

// Function to free the memory occupied by the linked list

void freeList(struct Node\* head) {

struct Node\* current = head;

struct Node\* next;

while (current != NULL) {

next = current->next;

free(current);

current = next;

}

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

struct Node\* P = NULL; // Original linked list

struct Node\* Q = NULL; // Linked list containing odd positions

struct Node\* R = NULL; // Linked list containing even positions

int choice, data;

while (1) {

printf("\nMenu:\n");

printf("1. Insert element into P\n");

printf("2. Split P into Q and R\n");

printf("3. Print Q and R\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element to insert: ");

scanf("%d", &data);

insert(&P, data);

break;

case 2:

split(P, &Q, &R);

printf("P has been split into Q and R.\n");

break;

case 3:

printf("Linked list Q: ");

printList(Q);

printf("Linked list R: ");

printList(R);

break;

case 4:

// Free the memory occupied by the linked lists

freeList(Q);

freeList(R);

exit(0);

default:

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

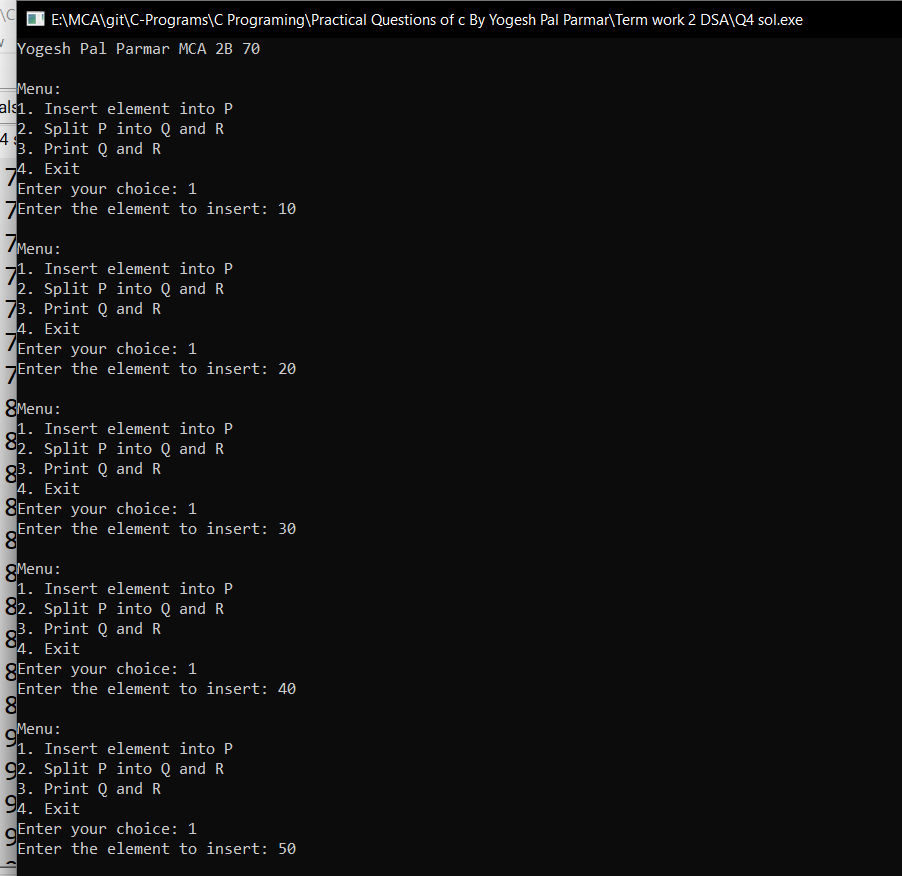
}

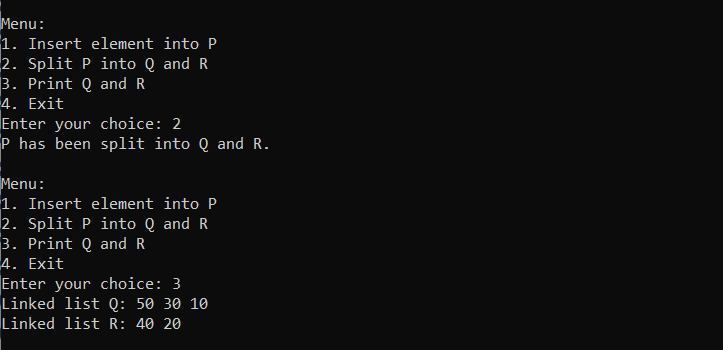
}

return 0;

}

**Output:**





**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Structure for 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\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a node into the binary search tree

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

if (root == NULL) {

return createNode(data);

} else if (data < root->data) {

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

} else if (data > root->data) {

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

}

return root;

}

// Function to search for a key in the binary search tree

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

if (root == NULL || root->data == key) {

return root;

} else if (key < root->data) {

return search(root->left, key);

} else {

return search(root->right, key);

}

}

// 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 {

// Node found, perform deletion

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\* minValue = minValueNode(root->right);

root->data = minValue->data;

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

}

return root;

}

// Function to count the total number of leaf nodes in the binary search tree

int countLeafNodes(struct Node\* root) {

if (root == NULL) {

return 0;

} else if (root->left == NULL && root->right == NULL) {

return 1;

} else {

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

}

}

// Function to count the nodes having both children in the binary search tree

int countNodesWithBothChildren(struct Node\* root) {

if (root == NULL || (root->left == NULL && root->right == NULL)) {

return 0;

} else if (root->left != NULL && root->right != NULL) {

return 1 + countNodesWithBothChildren(root->left) + countNodesWithBothChildren(root->right);

} else if (root->left != NULL) {

return countNodesWithBothChildren(root->left);

} else {

return countNodesWithBothChildren(root->right);

}

}

// Function to count the total number of nodes on the right and left sides of the root node

void countNodesOnSides(struct Node\* root, int key, int\* leftCount, int\* rightCount) {

if (root == NULL) {

return;

} else if (root->data == key) {

if (root->left != NULL) {

\*leftCount = countLeafNodes(root->left);

}

if (root->right != NULL) {

\*rightCount = countLeafNodes(root->right);

}

return;

} else if (key < root->data) {

countNodesOnSides(root->left, key, leftCount, rightCount);

} else {

countNodesOnSides(root->right, key, leftCount, rightCount);

}

}

// Function to free the memory occupied by the binary search tree

void freeTree(struct Node\* root) {

if (root == NULL) {

return;

}

freeTree(root->left);

freeTree(root->right);

free(root);

}

// Function to display the menu options

void displayMenu() {

printf("\nMenu:\n");

printf("1. Insert a node\n");

printf("2. Search for a key\n");

printf("3. Delete a node\n");

printf("4. Count total number of leaf nodes\n");

printf("5. Count nodes having both children\n");

printf("6. Count total number of nodes on left and right sides of the root\n");

printf("7. Exit\n");

printf("Enter your choice: ");

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70");

struct Node\* root = NULL;

int choice, key, leftCount, rightCount;

while (1) {

displayMenu();

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element to insert: ");

scanf("%d", &key);

root = insert(root, key);

printf("Node inserted successfully.\n");

break;

case 2:

printf("Enter the key to search: ");

scanf("%d", &key);

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

if (result != NULL) {

printf("Key found in the binary search tree.\n");

} else {

printf("Key not found in the binary search tree.\n");

}

break;

case 3:

printf("Enter the key to delete: ");

scanf("%d", &key);

root = deleteNode(root, key);

printf("Node deleted successfully.\n");

break;

case 4:

printf("Total number of leaf nodes: %d\n", countLeafNodes(root));

break;

case 5:

printf("Total number of nodes having both children: %d\n", countNodesWithBothChildren(root));

break;

case 6:

leftCount = 0;

rightCount = 0;

countNodesOnSides(root, root->data, &leftCount, &rightCount);

printf("Total number of nodes on the left side of the root: %d\n", leftCount);

printf("Total number of nodes on the right side of the root: %d\n", rightCount);

break;

case 7:

freeTree(root);

exit(0);

default:

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

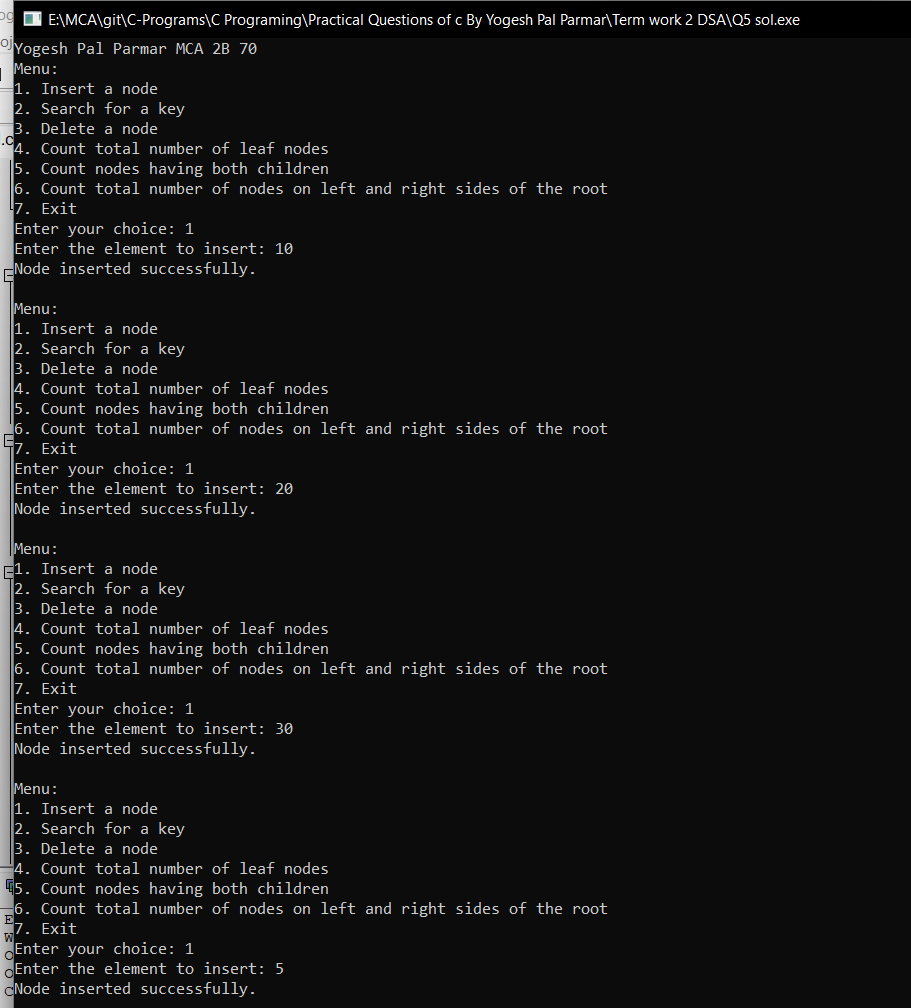
}

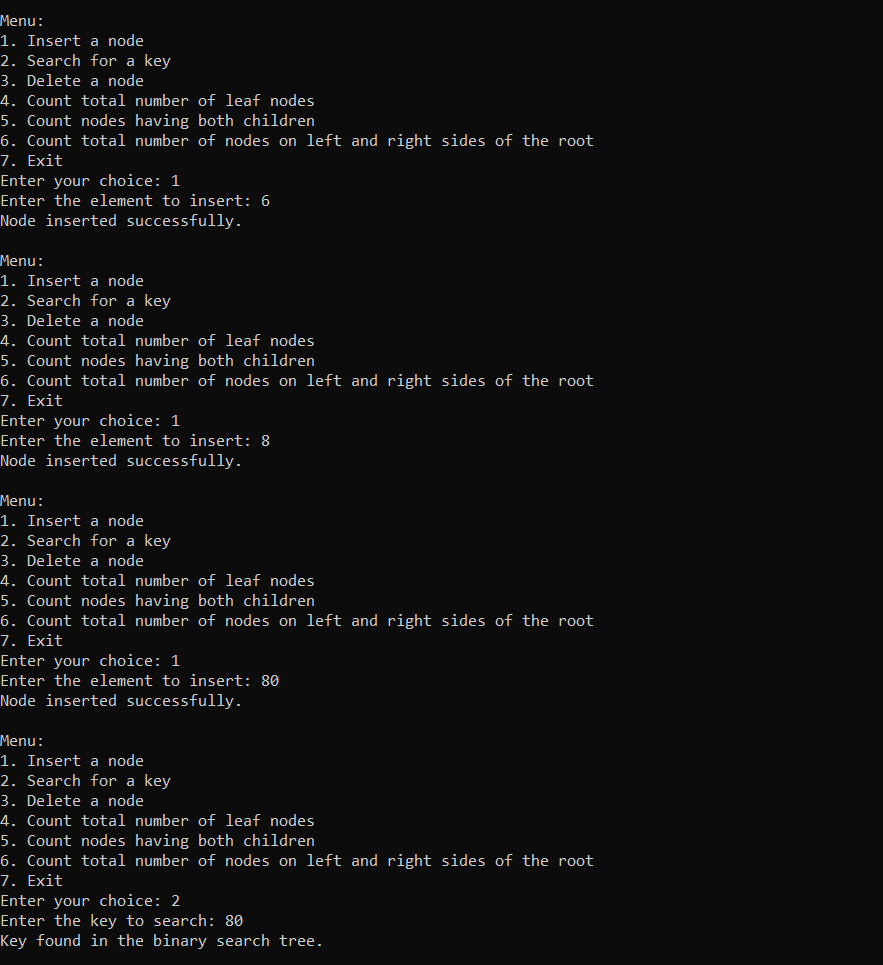
}

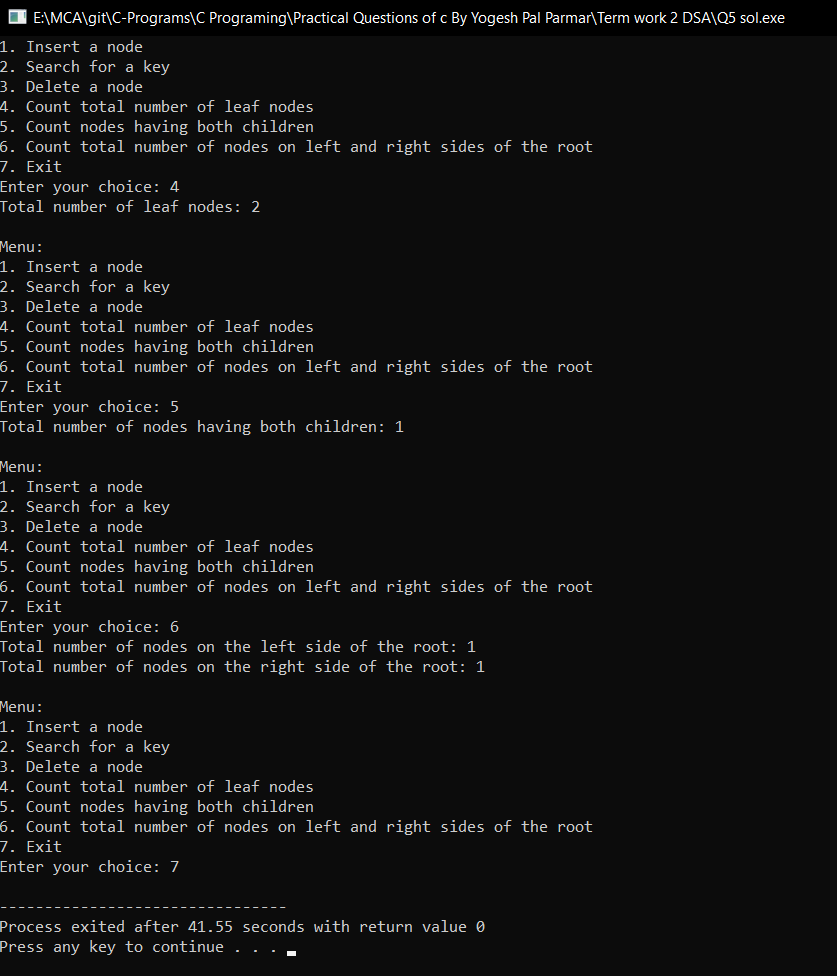
return 0;

}

**Output:**







**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// Structure for a term in the polynomial

struct Term {

int coefficient;

int exponent;

struct Term\* next;

};

// Function to create a new term with the given coefficient and exponent

struct Term\* createTerm(int coefficient, int exponent) {

struct Term\* newTerm = (struct Term\*)malloc(sizeof(struct Term));

newTerm->coefficient = coefficient;

newTerm->exponent = exponent;

newTerm->next = NULL;

return newTerm;

}

// Function to insert a term into the polynomial

void insertTerm(struct Term\*\* poly, int coefficient, int exponent) {

struct Term\* newTerm = createTerm(coefficient, exponent);

if (\*poly == NULL) {

\*poly = newTerm;

} else {

struct Term\* current = \*poly;

while (current->next != NULL) {

current = current->next;

}

current->next = newTerm;

}

}

// Function to add two polynomials and return the result

struct Term\* addPolynomials(struct Term\* poly1, struct Term\* poly2) {

struct Term\* result = NULL;

while (poly1 != NULL && poly2 != NULL) {

if (poly1->exponent > poly2->exponent) {

insertTerm(&result, poly1->coefficient, poly1->exponent);

poly1 = poly1->next;

} else if (poly1->exponent < poly2->exponent) {

insertTerm(&result, poly2->coefficient, poly2->exponent);

poly2 = poly2->next;

} else {

int sum = poly1->coefficient + poly2->coefficient;

if (sum != 0) {

insertTerm(&result, sum, poly1->exponent);

}

poly1 = poly1->next;

poly2 = poly2->next;

}

}

while (poly1 != NULL) {

insertTerm(&result, poly1->coefficient, poly1->exponent);

poly1 = poly1->next;

}

while (poly2 != NULL) {

insertTerm(&result, poly2->coefficient, poly2->exponent);

poly2 = poly2->next;

}

return result;

}

// Function to display the polynomial

void displayPolynomial(struct Term\* poly) {

if (poly == NULL) {

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

} else {

struct Term\* current = poly;

while (current != NULL) {

printf("(%dx^%d) ", current->coefficient, current->exponent);

if (current->next != NULL) {

printf("+ ");

}

current = current->next;

}

printf("\n");

}

}

// Function to free the memory occupied by the polynomial

void freePolynomial(struct Term\* poly) {

struct Term\* current = poly;

struct Term\* next;

while (current != NULL) {

next = current->next;

free(current);

current = next;

}

}

// Function to display the menu options

void displayMenu() {

printf("\nMenu:\n");

printf("1. Add a term to the first polynomial\n");

printf("2. Add a term to the second polynomial\n");

printf("3. Add the polynomials and display the result\n");

printf("4. Exit\n");

printf("Enter your choice: ");

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

struct Term\* poly1 = NULL;

struct Term\* poly2 = NULL;

struct Term\* result = NULL;

int choice, coefficient, exponent;

while (1) {

displayMenu();

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the coefficient: ");

scanf("%d", &coefficient);

printf("Enter the exponent: ");

scanf("%d", &exponent);

insertTerm(&poly1, coefficient, exponent);

printf("Term added to the first polynomial.\n");

break;

case 2:

printf("Enter the coefficient: ");

scanf("%d", &coefficient);

printf("Enter the exponent: ");

scanf("%d", &exponent);

insertTerm(&poly2, coefficient, exponent);

printf("Term added to the second polynomial.\n");

break;

case 3:

result = addPolynomials(poly1, poly2);

printf("Resultant polynomial: ");

displayPolynomial(result);

freePolynomial(result);

break;

case 4:

freePolynomial(poly1);

freePolynomial(poly2);

exit(0);

default:

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

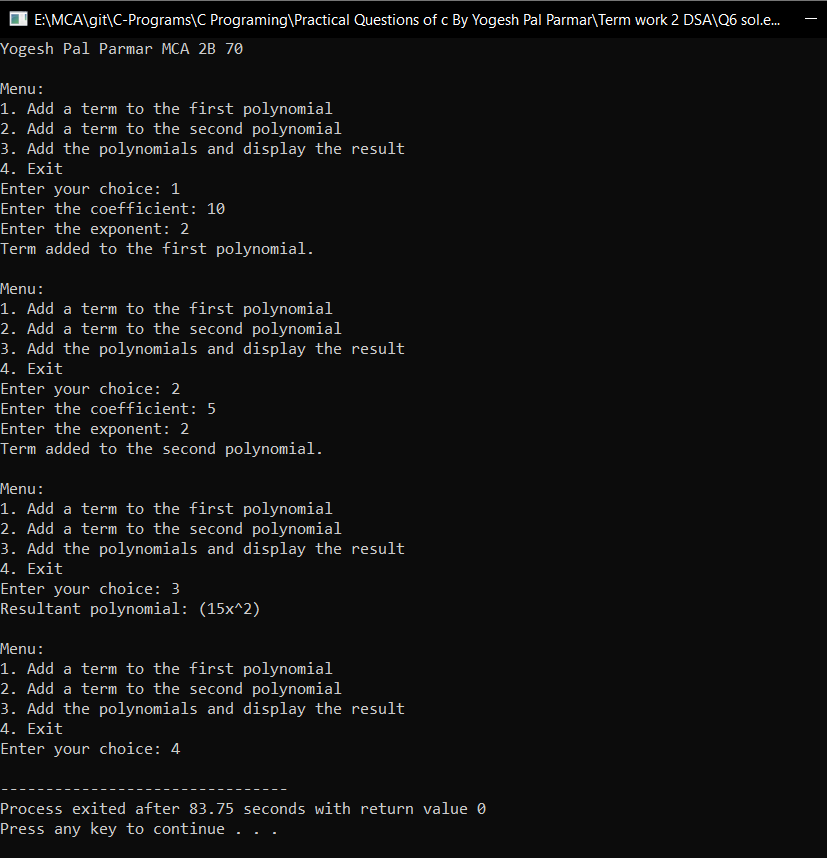
}

}

return 0;

}

**Output:**



**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to swap two characters

void swap(char\* a, char\* b) {

char temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to partition the array and return the pivot index

int partition(char arr[], int low, int high) {

char pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

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

return i + 1;

}

// Function to perform Quick sort on the array

void quickSort(char arr[], int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

// Function to display the array of characters

void displayArray(char arr[], int size) {

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

printf("%c ", arr[i]);

}

printf("\n");

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

char arr[100];

int size = 0;

int choice;

while (1) {

printf("\nMenu:\n");

printf("1. Enter a sequence of characters\n");

printf("2. Sort the sequence using Quick sort\n");

printf("3. Print the sorted array\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the sequence of characters: ");

scanf("%s", arr);

size = strlen(arr);

printf("Sequence of characters entered: %s\n", arr);

break;

case 2:

if (size > 0) {

quickSort(arr, 0, size - 1);

printf("Sequence of characters sorted using Quick sort.\n");

} else {

printf("Sequence of characters is empty. Please enter a sequence.\n");

}

break;

case 3:

if (size > 0) {

printf("Sorted array: ");

displayArray(arr, size);

} else {

printf("Sequence of characters is empty. Please enter a sequence.\n");

}

break;

case 4:

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

exit(0);

default:

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

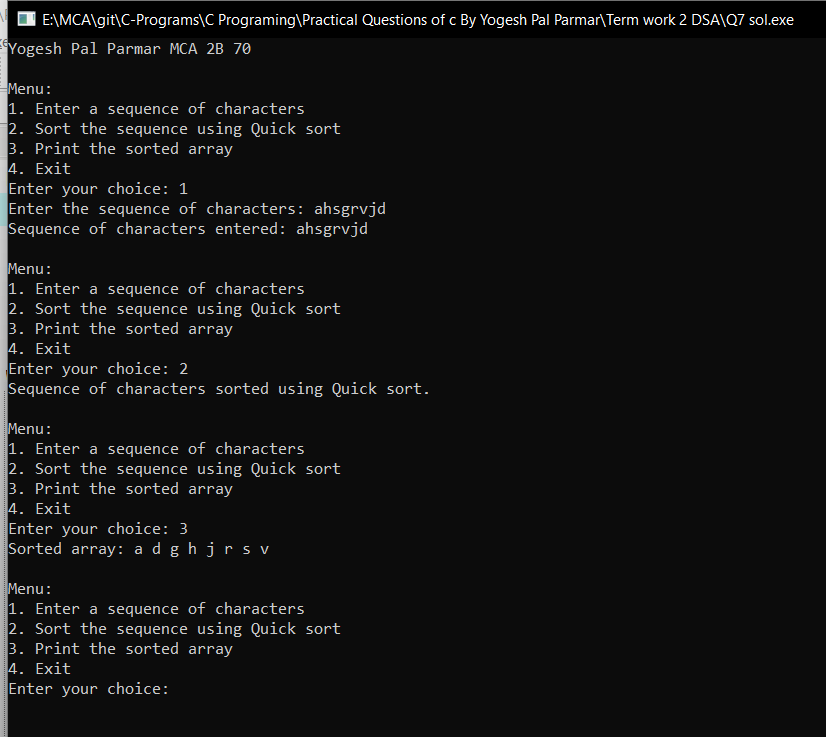
}

}

return 0;

}

**Output:**



**Source Code:**

#include <stdio.h>

#include <stdlib.h>

struct Process {

int processID;

int executionTime;

struct Process\* next;

};

// Function to create a new process

struct Process\* createProcess(int processID, int executionTime) {

struct Process\* newProcess = (struct Process\*)malloc(sizeof(struct Process));

newProcess->processID = processID;

newProcess->executionTime = executionTime;

newProcess->next = NULL;

return newProcess;

}

// Function to insert a process at the end of the circular linked list

void insertProcess(struct Process\*\* head, struct Process\*\* tail, int processID, int executionTime) {

struct Process\* newProcess = createProcess(processID, executionTime);

if (\*head == NULL) {

\*head = newProcess;

\*tail = newProcess;

(\*tail)->next = \*head;

} else {

(\*tail)->next = newProcess;

\*tail = newProcess;

(\*tail)->next = \*head;

}

}

// Function to allocate time slots to processes and print the completion time for each process

void allocateTimeSlots(struct Process\* head, int timeSlot) {

if (head == NULL) {

printf("No processes available.\n");

return;

}

struct Process\* current = head;

int totalTime = 0;

printf("Allocation of time slots:\n");

do {

if (current->executionTime <= timeSlot) {

printf("Process %d completed in %d ms.\n", current->processID, current->executionTime);

totalTime += current->executionTime;

struct Process\* temp = current;

current = current->next;

free(temp);

} else {

printf("Process %d completed in %d ms.\n", current->processID, timeSlot);

current->executionTime -= timeSlot;

totalTime += timeSlot;

current = current->next;

}

} while (current != head);

printf("\nTotal execution time: %d ms\n", totalTime);

}

// Function to display the menu options

void displayMenu() {

printf("\nMenu:\n");

printf("1. Add a process\n");

printf("2. Allocate time slots and print completion time\n");

printf("3. Exit\n");

printf("Enter your choice: ");

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

struct Process\* head = NULL;

struct Process\* tail = NULL;

int choice, processID, executionTime;

int timeSlot = 10;

while (1) {

displayMenu();

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the process ID: ");

scanf("%d", &processID);

printf("Enter the execution time: ");

scanf("%d", &executionTime);

insertProcess(&head, &tail, processID, executionTime);

printf("Process added.\n");

break;

case 2:

allocateTimeSlots(head, timeSlot);

head = NULL;

tail = NULL;

break;

case 3:

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

exit(0);

default:

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

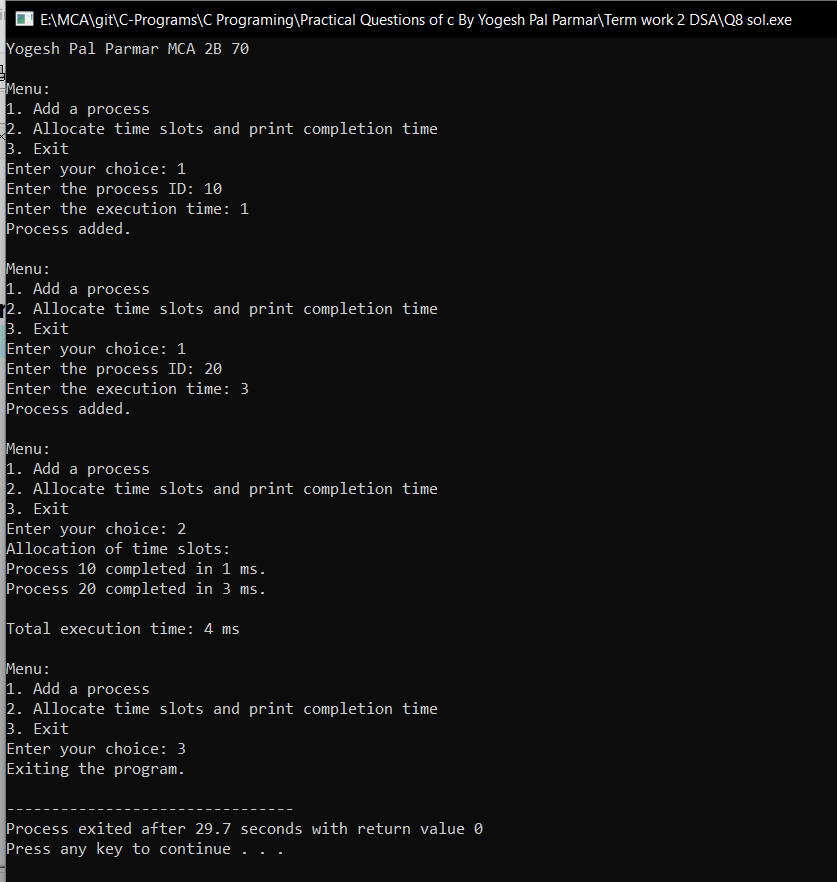
}

}

return 0;

}

**Output:**



**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 10

struct Vertex {

int vertexID;

int weight;

};

struct Graph {

int numVertices;

struct Vertex\* vertices[MAX\_VERTICES];

int adjacencyMatrix[MAX\_VERTICES][MAX\_VERTICES];

};

// Function to create a new vertex

struct Vertex\* createVertex(int vertexID, int weight) {

struct Vertex\* newVertex = (struct Vertex\*)malloc(sizeof(struct Vertex));

newVertex->vertexID = vertexID;

newVertex->weight = weight;

return newVertex;

}

// Function to create a new graph

struct Graph\* createGraph(int numVertices) {

struct Graph\* newGraph = (struct Graph\*)malloc(sizeof(struct Graph));

newGraph->numVertices = numVertices;

// Initialize the adjacency matrix with 0 weight for all edges

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

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

newGraph->adjacencyMatrix[i][j] = 0;

}

}

// Initialize the vertices array with NULL

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

newGraph->vertices[i] = NULL;

}

return newGraph;

}

// Function to add a vertex to the graph

void addVertex(struct Graph\* graph, int vertexID, int weight) {

if (vertexID >= graph->numVertices) {

printf("Invalid vertex ID. Vertex ID should be less than the number of vertices.\n");

return;

}

struct Vertex\* newVertex = createVertex(vertexID, weight);

graph->vertices[vertexID] = newVertex;

}

// Function to add an edge to the graph

void addEdge(struct Graph\* graph, int startVertexID, int endVertexID, int weight) {

if (startVertexID >= graph->numVertices || endVertexID >= graph->numVertices) {

printf("Invalid vertex IDs. Vertex IDs should be less than the number of vertices.\n");

return;

}

graph->adjacencyMatrix[startVertexID][endVertexID] = weight;

graph->adjacencyMatrix[endVertexID][startVertexID] = weight;

}

// Function to display the graph details

void displayGraph(struct Graph\* graph) {

printf("Graph Details:\n");

printf("Number of Vertices: %d\n", graph->numVertices);

printf("Adjacency Matrix:\n");

for (int i = 0; i < graph->numVertices; i++) {

for (int j = 0; j < graph->numVertices; j++) {

printf("%d ", graph->adjacencyMatrix[i][j]);

}

printf("\n");

}

printf("Vertex Details:\n");

for (int i = 0; i < graph->numVertices; i++) {

if (graph->vertices[i] != NULL) {

printf("Vertex ID: %d, Weight: %d\n", graph->vertices[i]->vertexID, graph->vertices[i]->weight);

}

}

}

// Function to display the menu options

void displayMenu() {

printf("\nMenu:\n");

printf("1. Add a vertex\n");

printf("2. Add an edge\n");

printf("3. Display graph details\n");

printf("4. Exit\n");

printf("Enter your choice: ");

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

struct Graph\* graph = NULL;

int numVertices, choice, startVertex, endVertex, weight;

while (1) {

displayMenu();

scanf("%d", &choice);

switch (choice) {

case 1:

if (graph != NULL) {

printf("Graph already created. Please choose another option.\n");

} else {

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

scanf("%d", &numVertices);

graph = createGraph(numVertices);

printf("Graph created.\n");

}

break;

case 2:

if (graph == NULL) {

printf("Graph not created. Please create the graph first.\n");

} else {

printf("Enter the start vertex ID: ");

scanf("%d", &startVertex);

printf("Enter the end vertex ID: ");

scanf("%d", &endVertex);

printf("Enter the weight: ");

scanf("%d", &weight);

addEdge(graph, startVertex, endVertex, weight);

printf("Edge added.\n");

}

break;

case 3:

if (graph == NULL) {

printf("Graph not created. Please create the graph first.\n");

} else {

displayGraph(graph);

}

break;

case 4:

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

exit(0);

default:

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

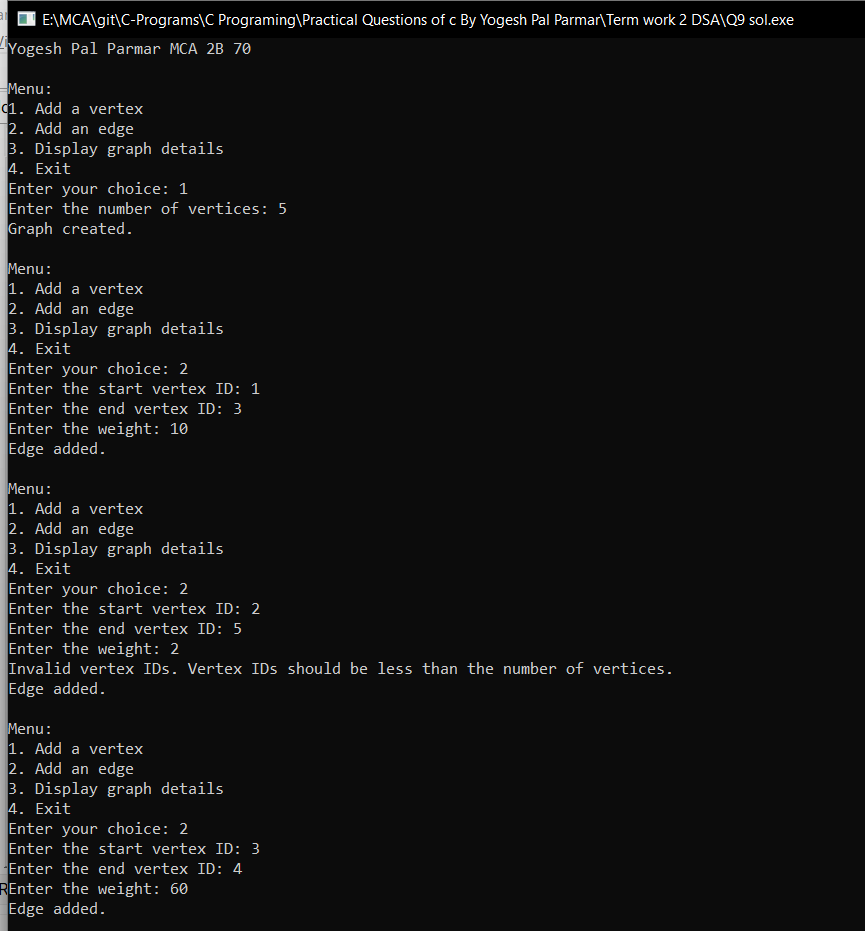
}

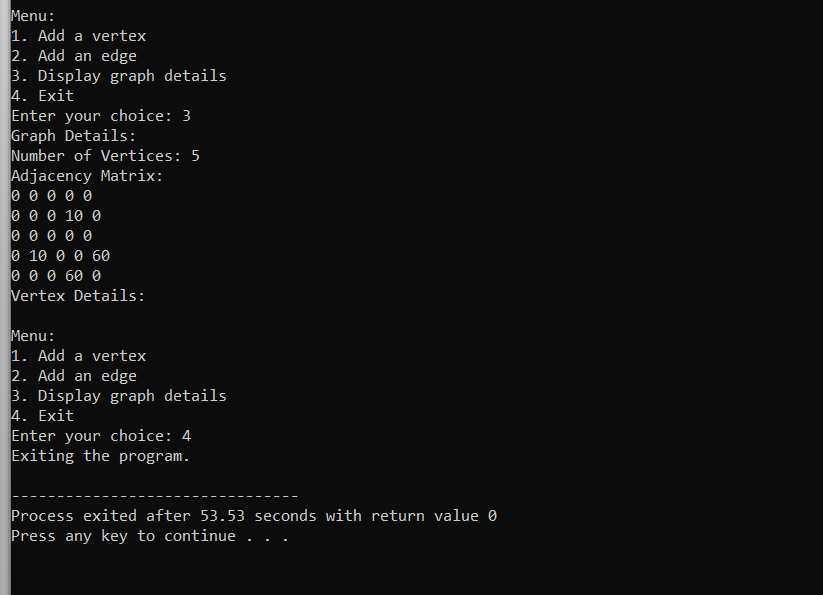
}

return 0;

}

**Output:**





**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 10

struct Queue {

int front, rear;

int items[MAX\_VERTICES];

};

struct Graph {

int numVertices;

int adjacencyMatrix[MAX\_VERTICES][MAX\_VERTICES];

};

// Function to create a new queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = -1;

queue->rear = -1;

return queue;

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* queue) {

return queue->front == -1;

}

// Function to enqueue an item into the queue

void enqueue(struct Queue\* queue, int item) {

if (queue->rear == MAX\_VERTICES - 1) {

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

return;

}

if (queue->front == -1)

queue->front = 0;

queue->rear++;

queue->items[queue->rear] = item;

}

// Function to dequeue an item from the queue

int dequeue(struct Queue\* queue) {

if (isEmpty(queue)) {

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

return -1;

}

int item = queue->items[queue->front];

if (queue->front == queue->rear)

queue->front = queue->rear = -1;

else

queue->front++;

return item;

}

// Function to create a new graph

struct Graph\* createGraph(int numVertices) {

struct Graph\* newGraph = (struct Graph\*)malloc(sizeof(struct Graph));

newGraph->numVertices = numVertices;

// Initialize the adjacency matrix with 0 for all edges

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

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

newGraph->adjacencyMatrix[i][j] = 0;

}

}

return newGraph;

}

// Function to add an edge to the graph

void addEdge(struct Graph\* graph, int startVertex, int endVertex) {

if (startVertex >= graph->numVertices || endVertex >= graph->numVertices) {

printf("Invalid vertex IDs. Vertex IDs should be less than the number of vertices.\n");

return;

}

graph->adjacencyMatrix[startVertex][endVertex] = 1;

graph->adjacencyMatrix[endVertex][startVertex] = 1;

}

// Function to perform Breadth-First Search (BFS) on the graph

void BFS(struct Graph\* graph, int startVertex) {

int visited[MAX\_VERTICES] = {0};

struct Queue\* queue = createQueue();

visited[startVertex] = 1;

enqueue(queue, startVertex);

printf("BFS Traversal: ");

while (!isEmpty(queue)) {

int currentVertex = dequeue(queue);

printf("%d ", currentVertex);

for (int i = 0; i < graph->numVertices; i++) {

if (graph->adjacencyMatrix[currentVertex][i] == 1 && visited[i] == 0) {

visited[i] = 1;

enqueue(queue, i);

}

}

}

printf("\n");

}

// Function to display the menu options

void displayMenu() {

printf("\nMenu:\n");

printf("1. Add an edge\n");

printf("2. Perform BFS\n");

printf("3. Exit\n");

printf("Enter your choice: ");

}

int main() {

printf("Yogesh Pal Parmar MCA 2B 70\n");

struct Graph\* graph = NULL;

int numVertices, choice, startVertex, endVertex;

while (1) {

displayMenu();

scanf("%d", &choice);

switch (choice) {

case 1:

if (graph == NULL) {

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

scanf("%d", &numVertices);

graph = createGraph(numVertices);

}

printf("Enter the start vertex ID: ");

scanf("%d", &startVertex);

printf("Enter the end vertex ID: ");

scanf("%d", &endVertex);

addEdge(graph, startVertex, endVertex);

printf("Edge added.\n");

break;

case 2:

if (graph == NULL) {

printf("Graph not created. Please create the graph first.\n");

} else {

printf("Enter the starting vertex for BFS: ");

scanf("%d", &startVertex);

BFS(graph, startVertex);

}

break;

case 3:

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

exit(0);

default:

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

}

}

return 0;

}

**Output:**

