**Data Structures And Algorithms Practical File**

**(IT-663)**

***Submitted in partial fulfillment of the requirements for the award of the degree***

***Of***

**Masters of Computer Application (SE)**

**Submitted to: Submitted By:**

**Dr. Reena Gupta Name : Thakur Rudraksha Singh**

**Associate Professor Enrollment No : 1616404524**

**Semester - I**

****

**University School of Information , Communication & Technology**

**Guru Gobind Singh Indraprastha University**

**Sector-16C , Dwarka New Delhi - 110078**

**(2024-2026)**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S No.** | **Practical** | **Page No.** |
| 1 | Implement operations (traverse, insert (beg, loc, end), delete (beg, loc, end), linear search, binary search using recursion, insertion sort) on an array. |  |
| 2 | Implement Sparse Array. |  |
| 3 | Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on single linked list and count the number of nodes & reverse the single linked list. |  |
| 4 | Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on circular single linked list. |  |
| 5 | Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on double linked list. |  |
| 6 | Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on circular double linked list. |  |
| 7 | Write a program to merge two sorted linked list and display the final sorted linked list. |  |
| 8 | Implement operations (push, pop) on a stack using arrays. Check the status of the stack whether there is underflow or overflow. |  |
| 9 | Implement the conversion of infix notation to postfix notation and evaluation of postfix notation using stacks. |  |
| 10 | Implement operations (enqueue, dequeue) on a queue using arrays. Check the status of the queue whether it is empty or full. |  |
| 11 | Implement circular queue using arrays and linked list. |  |
| 12 | Implement stacks and queues using linked list. |  |
| 13 | Implement operations on Binary Search Tree (Insertion, Deletion, Search, Traversals (using recursion)- Inorder, Preorder, Postorder). |  |
| 14 | Implement traversals on Binary Search Tree (using stacks) - Inorder, Preorder, Postorder. |  |
| 15 | Implement AVL tree. |  |
| 16 | Implement graph traversals (DFS & BFS). |  |
| 17 | Implement dijkstra’s shortest path greedy algorithm and Kruskal’s minimum spanning tree greedy algorithm. |  |
| 18 | Implement Longest Common Subsequence dynamic programming algorithm. |  |
| 19 | Perform various sorting techniques (heap, count, radix, bucket). |  |
| 20 | Perform divide and conquer sorting techniques (merge, quick). |  |

1. **Implement operations (traverse, insert (beg, loc, end), delete (beg, loc, end), linear search, binary search using recursion, insertion sort) on an array.**

#include <iostream>

using namespace std;

void traverse(int arr[], int n){

    for (int i = 0; i < n; i++)

        cout << arr[i] << " ";

    cout << endl;

}

void insertAtBeginning(int arr[], int &n, int value, int capacity){

    if (n >= capacity){

        cout << "Array is full." << endl;

        return;

    }

    for (int i = n; i > 0; i--)

        arr[i] = arr[i - 1];

    arr[0] = value;

    n++;

    cout << "Value " << value << " is inserted at the beginning of the array." << endl;

    cout << "Your updated array is: " << endl;

    traverse(arr, n);

}

void insertAtEnd(int arr[], int &n, int value, int capacity){

    if (n >= capacity){

        cout << "Array is full." << endl;

        return;

    }

    arr[n] = value;

    n++;

    cout << "Value " << value << " is inserted at the end of the array." << endl;

    cout << "Your updated array is: " << endl;

    traverse(arr, n);

}

void insertAtPosition(int arr[], int &n, int value, int position, int capacity){

    if (position < 0 || position > n){

        cout << "Invalid position." << endl;

        return;

    }

    if (n >= capacity){

        cout << "Array is full." << endl;

        return;

    }

    for (int i = n; i > position; i--)

        arr[i] = arr[i - 1];

    arr[position] = value;

    n++;

    cout << "Value " << value << " is inserted at position " << position << "." << endl;

    cout << "Your updated array is: " << endl;

    traverse(arr, n);

}

void deleteFromBeginning(int arr[], int &n){

    if (n <= 0){

        cout << "Array is empty." << endl;

        return;

    }

    for (int i = 0; i < n - 1; i++)

        arr[i] = arr[i + 1];

    n--;

    cout << "Element deleted from the beginning of the array." << endl;

    cout << "Your updated array is: " << endl;

    traverse(arr, n);

}

void deleteFromEnd(int arr[], int &n){

    if (n <= 0){

        cout << "Array is empty." << endl;

        return;

    }

    n--;

    cout << "Element deleted from the end of the array." << endl;

    cout << "Your updated array is: " << endl;

    traverse(arr, n);

}

void deleteFromPosition(int arr[], int &n, int position){

    if (position < 0 || position >= n){

        cout << "Invalid position." << endl;

        return;

    }

    for (int i = position; i < n - 1; i++)

        arr[i] = arr[i + 1];

    n--;

    cout << "Element deleted from position " << position << "." << endl;

    cout << "Your updated array is: " << endl;

    traverse(arr, n);

}

int linearSearch(int arr[], int n, int key){

    for (int i = 0; i < n; i++){

        if (arr[i] == key)

            return i;

    }

    return -1;

}

int binarySearch(int arr[], int left, int right, int key){

    if (right >= left){

        int mid = left + (right - left) / 2;

        if (arr[mid] == key)

            return mid;

        if (arr[mid] > key)

            return binarySearch(arr, left, mid - 1, key);

        return binarySearch(arr, mid + 1, right, key);

    }

    return -1;

}

void insertionSort(int arr[], int n){

    for (int i = 1; i < n; i++)   //j is initialized to i - 1, which points to the last element of the sorted portion.

    {

        int key = arr[i];

        int j = i - 1;

        while (j >= 0 && arr[j] > key){

            arr[j + 1] = arr[j];

            j--;

        }

        arr[j + 1] = key;

    }

}

int main(){

     int capacity = 100; // Set the maximum capacity of the array

    int arr[capacity];

    int n = 0; // Initialize the size of the array to 0

    cout << "Enter the number of elements in your array (max: " << capacity << "): ";

    int numElements;

    cin >> numElements;

    if (numElements > capacity) {

        cout << "Number of elements exceeds capacity. Setting size to " << capacity << "." << endl;

        cout << "as you have given larger than capacity we are taking number of elemts as 5 by default" << endl;

        numElements = 5;

    }

    cout << "Enter the elements of the array: ";

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

        cin >> arr[i];

    }

    n = numElements; // Update the size of the array

    bool continueProgram = true;

    while (continueProgram){

        cout << "\nMenu:\n";

        cout << "1. Traverse Array        ";

        cout << "2. Insert at Beginning     ";

        cout << "3. Insert at End\n";

        cout << "4. Insert at Position    ";

        cout << "5. Delete from Beginning   ";

        cout << "6. Delete from End\n";

        cout << "7. Delete from Position  ";

        cout << "8. Linear Search           ";

        cout << "9. Binary Search (Array should be sorted)\n";

        cout << "10. Insertion Sort\n";

        cout << "Enter your choice: ";

        int choice, value, position;

        cin >> choice;

        switch (choice){

        case 1:

            traverse(arr, n);

            break;

        case 2:

            cout << "Enter value to insert: ";

            cin >> value;

            insertAtBeginning(arr, n, value, capacity);

            break;

        case 3:

            cout << "Enter value to insert: ";

            cin >> value;

            insertAtEnd(arr, n, value, capacity);

            break;

        case 4:

            cout << "Enter value to insert : ";

            cin >> value;

            cout << "Enter position to insert (0-based index) from 0 to "<<n<<" : ";

            cin >> position;

            insertAtPosition(arr, n, value, position, capacity);

            break;

        case 5:

            deleteFromBeginning(arr, n);

            break;

        case 6:

            deleteFromEnd(arr, n);

            break;

        case 7:

            cout << "Enter position to delete (0-based index) 0 to "<<n-1<<": ";

            cin >> position;

            deleteFromPosition(arr, n, position);

            break;

        case 8:

            cout << "Enter value to search: ";

            cin >> value;

            position = linearSearch(arr, n, value);

            if (position != -1)

                cout << "Element found at position: " << position << endl;

            else

                cout << "Element not found." << endl;

            break;

        case 9:

            cout << "Enter value to search: ";

            cin >> value;

            position = binarySearch(arr, 0, n - 1, value);

            if (position != -1)

                cout << "Element found at position : " << position << endl;

            else

                cout << "Element not found." << endl;

            break;

        case 10:

            insertionSort(arr, n);

            cout << "Array sorted successfully." << endl;

            break;

        default:

            cout << "Invalid choice. Try again." << endl;

        }

        // Ask if the user wants to continue

        char cont;

        cout << "\nDo you want to continue? (y/n): ";

        cin >> cont;

        if (cont != 'y' && cont != 'Y'){

            continueProgram = false;

            cout << "Exiting program." << endl;

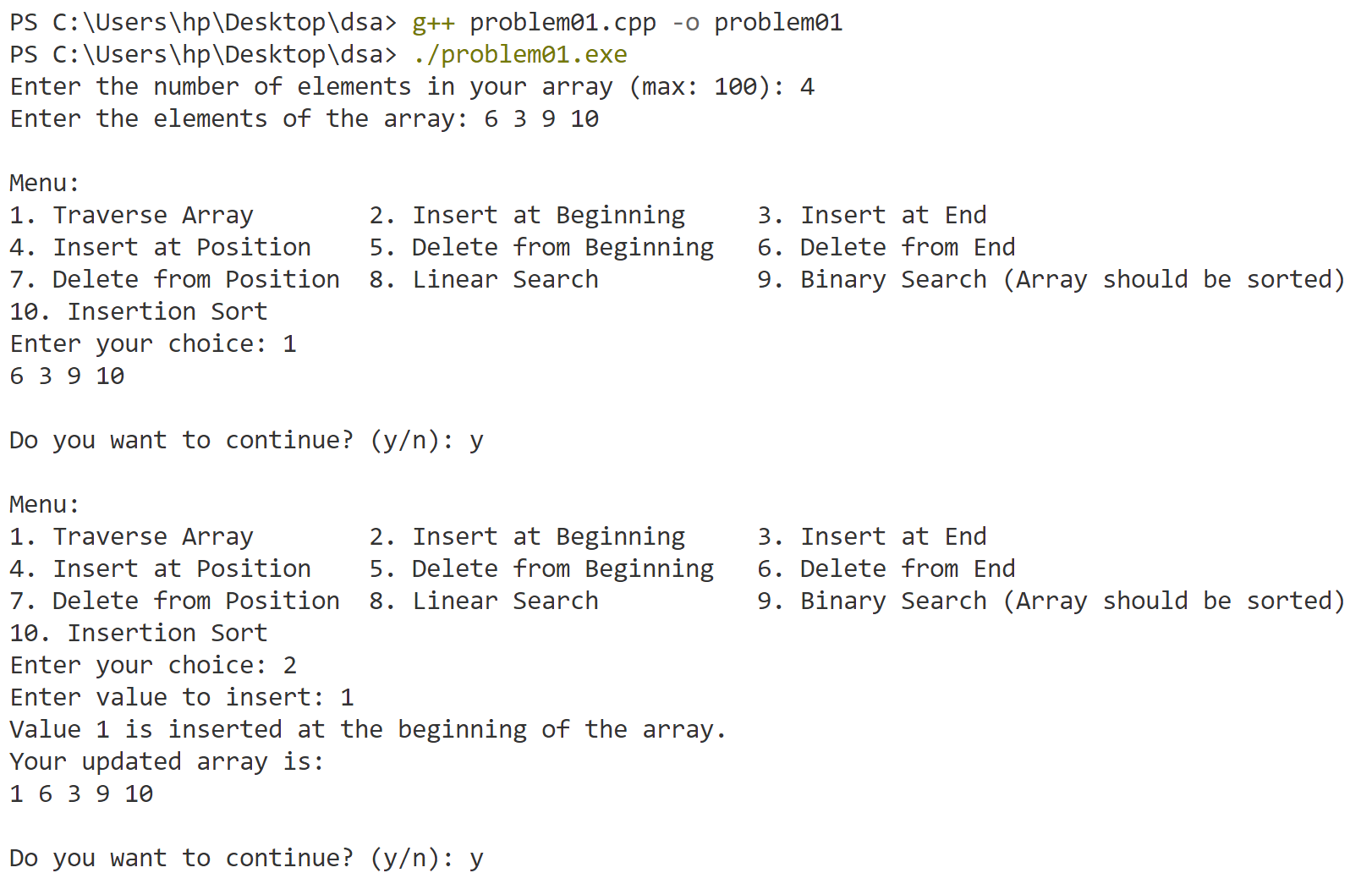
        }

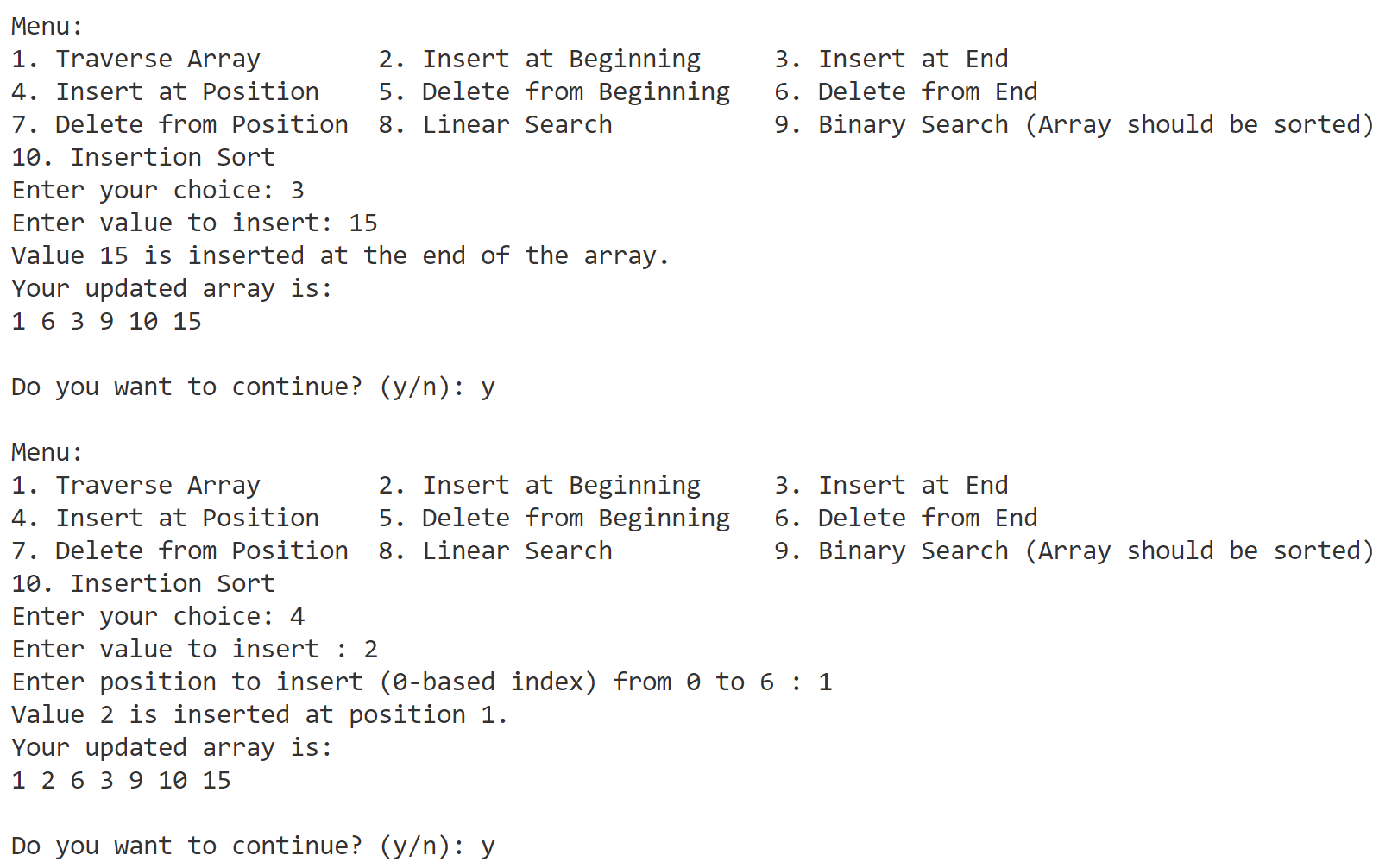
    }

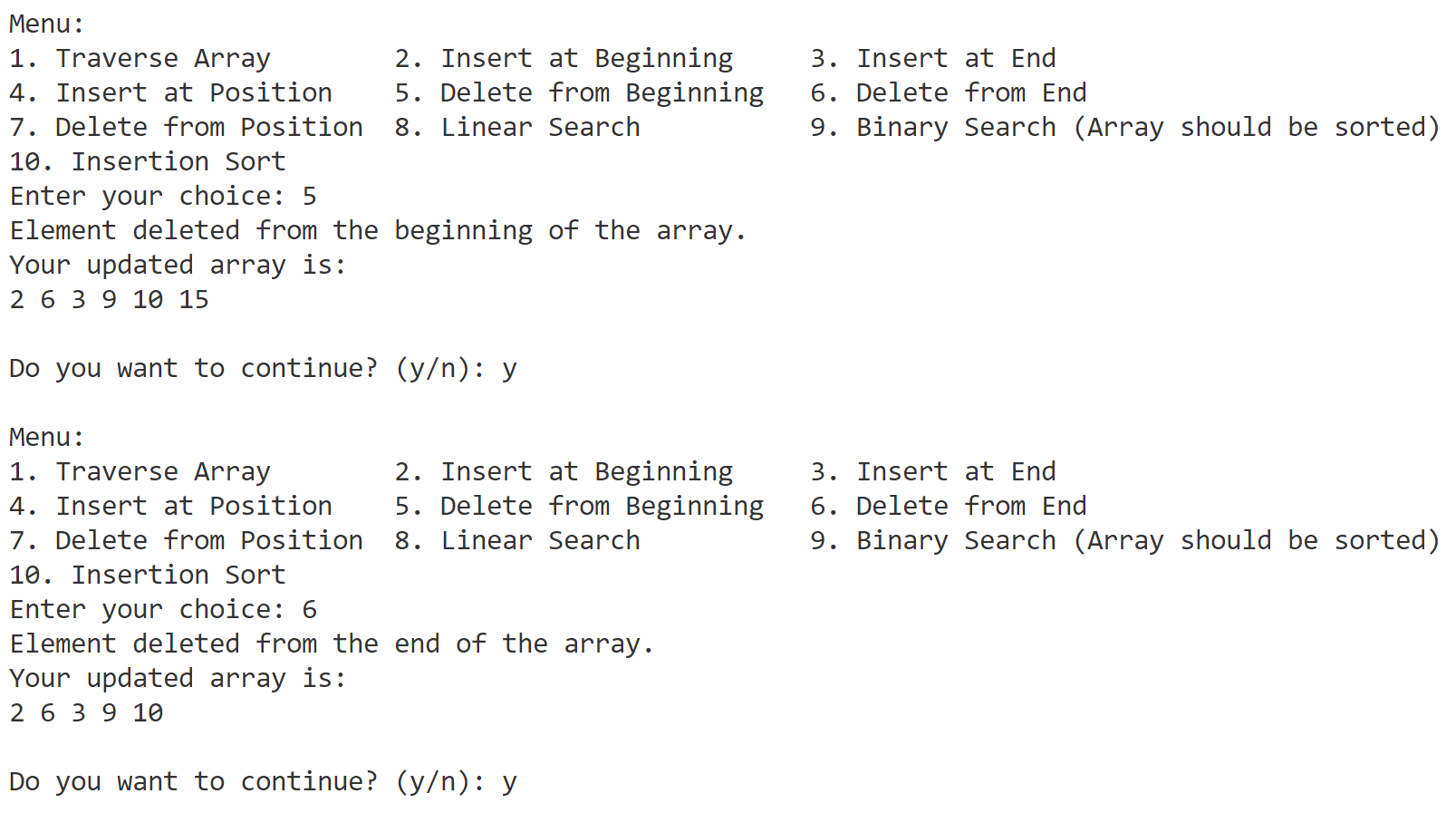
return 0;

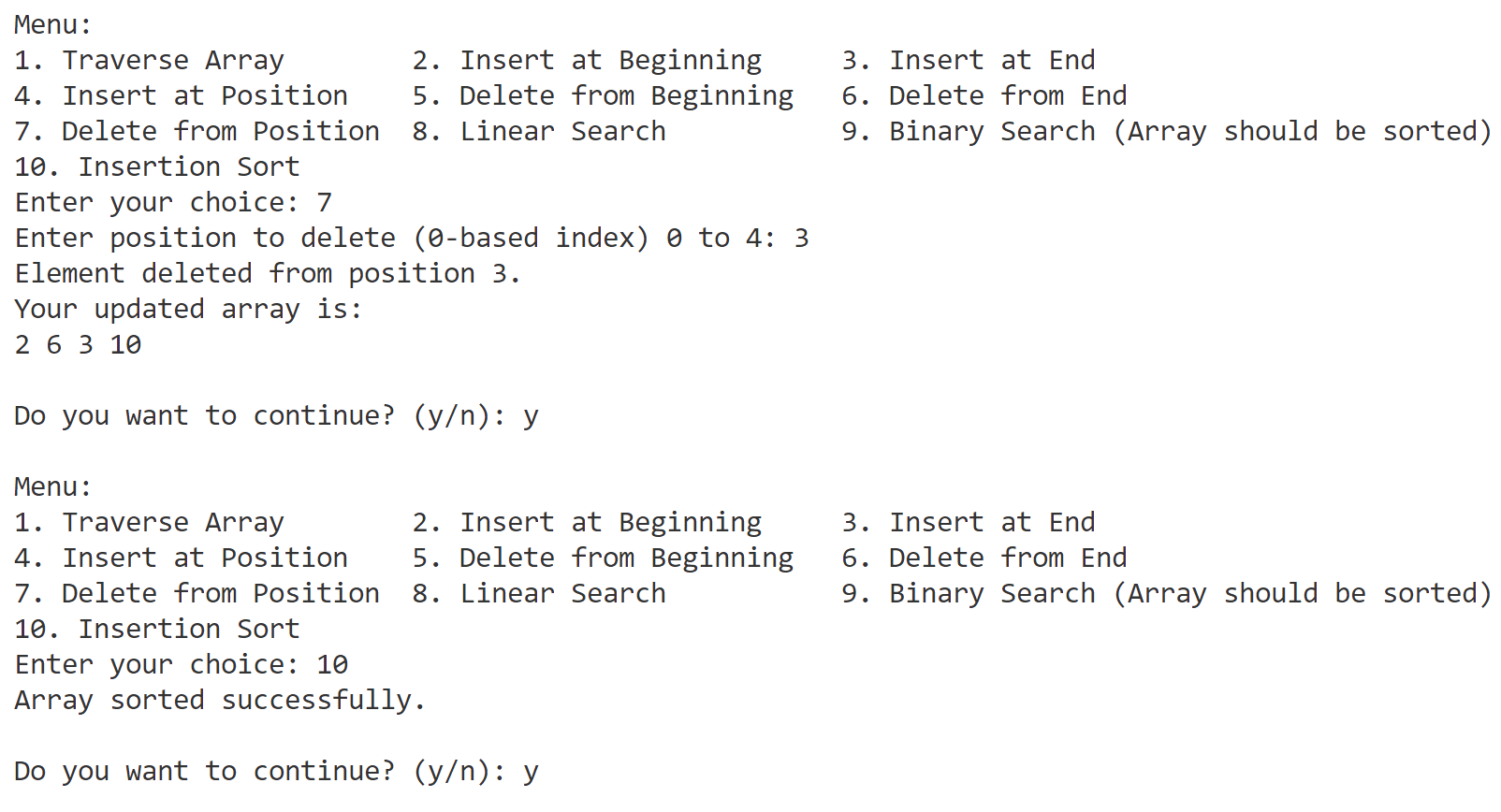
}

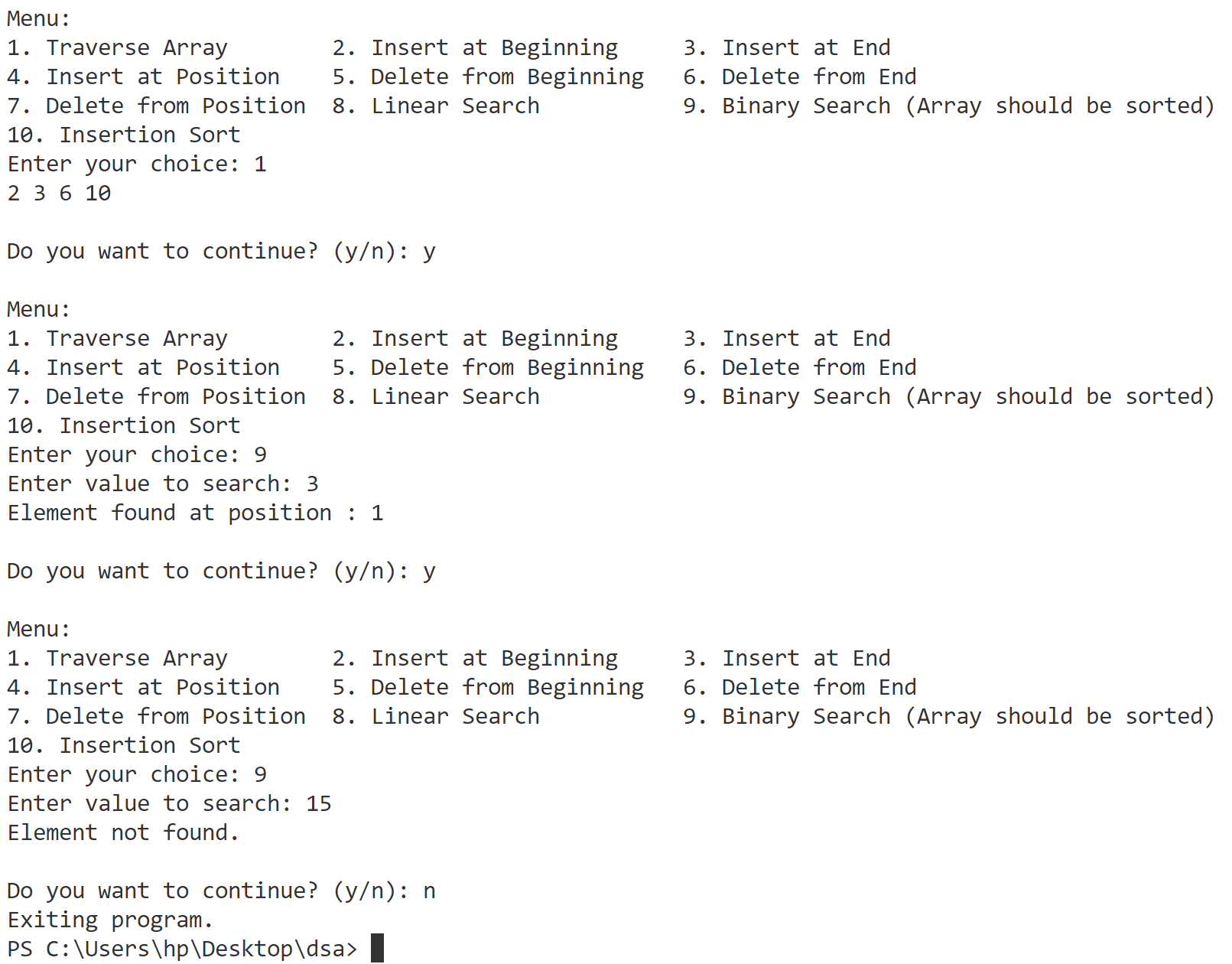
**Output**



****

****

****

****

1. **Implement Sparse Array.**

//sparse matrix

#include <bits/stdc++.h>

using namespace std;

int main() {

    ///////////////////////////Array////////////////////////////////////

    int rows, columns;

    cout << "Enter Rows(max:10) and Columns(max:10) :" << endl;

    cin >> rows >> columns;

    if (rows == 0 && columns == 0) {

        cout << "both rows and columns cannot be zero" << endl;

        return 0;

    } else if (rows > 10 || columns > 10) {

        cout << "rows and columns cannot be greater than 10" << endl;

        return 0;

    }

    int arr[rows][columns];

    cout << "Enter elements of matrix:" << endl;

    for (int i = 0; i < rows; i++) {

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

            cin >> arr[i][j];

        }

    }

    cout << "Input Matrix: " << endl;

    for (int i = 0; i < rows; i++) {

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

            cout << arr[i][j] << " ";

        }

        cout << endl;

    }

    ///////////////////////////Sparse Matrix////////////////////////////////////

    int cnt = 0;

    for (int i = 0; i < rows; i++) {

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

            if (arr[i][j] == 0)

                cnt++;

        }

    }

    // total elements in array

    int totalElements = rows \* columns;

    if (totalElements - cnt > (totalElements / 4)) {

        cout << "not sparse matrix" << endl;

        cout << "exiting program." << endl;

        return 0;

    }

    cout << "Sparse Matrix" << endl;

    int nonZeroElement = totalElements - cnt;

    int sparseMatrix[nonZeroElement + 1][3];

    int row = 0;

    sparseMatrix[row][0] = rows;

    sparseMatrix[row][1] = columns;

    sparseMatrix[row][2] = nonZeroElement;

    row++;

    for (int i = 0; i < rows; i++) {

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

            if (arr[i][j] != 0) {

                sparseMatrix[row][0] = i;

                sparseMatrix[row][1] = j;

                sparseMatrix[row][2] = arr[i][j];

                row++;

            }

        }

    }

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

        for (int j = 0; j < 3; j++) {

            cout << sparseMatrix[i][j] << " ";

        }

        cout << endl;

    }

    cout << "used zero based indexing" << endl;

    return 0;

}

**Output**

****

1. **Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on single linked list and count the number of nodes & reverse the single linked list.**

//Singly Linked List

#include <bits/stdc++.h>

using namespace std;

struct Node {

    int data;

    Node \* next;

};

Node \* head = nullptr;

void traverseList() {

    cout << "---------------------------------" << endl;

    if (head == nullptr) {

        cout << "The list is empty." << endl;

        return;

    }

    Node \* temp = head;

    cout << "The linked list is: ";

    while (temp != nullptr) {

        cout << temp -> data << " ";

        temp = temp -> next;

    }

    cout << endl;

    cout << "---------------------------------" << endl;

}

void insertAtBeginning() {

    int value;

    cout << "Enter value to enter : ";

    cin >> value;

    cout << endl;

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

    newNode -> data = value;

    newNode -> next = head;

    head = newNode;

    cout << "Updated LinkedList :" << endl;

    traverseList();

}

void insertAtEnd(int value) {

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

    newNode -> data = value;

    newNode -> next = nullptr;

    // list is empty

    if (head == nullptr) {

        head = newNode;

    } else {

        // Traverse to the end of the list

        Node \* temp = head;

        while (temp -> next != nullptr) {

            temp = temp -> next;

        }

        temp -> next = newNode;

    }

}

void insertAfterGivenNode() {

    int value, givenNode;

    cout << "Enter value to insert: ";

    cin >> value;

    cout << "Enter the value of the node after which to insert: ";

    cin >> givenNode;

    cout << endl;

    // Traverse the list to find the node with the value givenNode

    Node \* temp = head;

    while (temp != nullptr && temp -> data != givenNode) {

        temp = temp -> next;

    }

    if (temp != nullptr) {

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

        newNode -> data = value;

        newNode -> next = temp -> next;

        temp -> next = newNode;

        cout << "Node with value " << value << " has been inserted after node with value " << givenNode << "." << endl;

    } else {

        cout << "Node with value " << givenNode << " not found." << endl;

    }

    traverseList();

}

void deleteAtBeginning() {

    if (head == nullptr) {

        cout << "The list is empty, nothing to delete." << endl;

        return;

    }

    Node \* temp = head;

    head = head -> next;

    free(temp);

    traverseList();

}

void deleteAtEnd() {

    if (head == nullptr) {

        cout << "The list is empty, nothing to delete." << endl;

        return;

    }

    // Check if there is only one node

    if (head -> next == nullptr) {

        free(head);

        head = nullptr;

        cout << "The only node in the list has been deleted." << endl;

        traverseList();

        return;

    }

    Node \* temp = head;

    while (temp -> next -> next != nullptr) {

        temp = temp -> next;

    }

    free(temp -> next);

    temp -> next = nullptr;

    traverseList();

}

void deleteTheGivenNode() {

    int givenNode;

    cout << "Enter value of node to delete: ";

    cin >> givenNode;

    cout << endl;

    if (head == nullptr) {

        cout << "The list is empty, nothing to delete." << endl;

        return;

    }

    // node to be deleted is head

    if (head -> data == givenNode) {

        Node \* temp = head;

        head = head -> next;

        free(temp);

        cout << "Node with value " << givenNode << " has been deleted." << endl;

        traverseList();

        return;

    }

    // Traverse the list to find the node to delete

    Node \* current = head;

    Node \* previous = nullptr;

    while (current != nullptr && current -> data != givenNode) {

        previous = current;

        current = current -> next;

    }

    // If the node was not found

    if (current == nullptr) {

        cout << "Node with value " << givenNode << " not found." << endl;

        return;

    }

    // Node found,  deletion

    previous -> next = current -> next;

    free(current);

    cout << "Node with value " << givenNode << " has been deleted." << endl;

    traverseList();

}

void count() {

    Node \* temp = head;

    int count = 0;

    while (temp != nullptr) {

        count++;

        temp = temp -> next;

    }

    cout << "There are " << count << " nodes in the list." << endl;

}

void reverseSingleLL() {

    Node \* current = head;

    Node \* prev = nullptr;

    Node \* next = nullptr;

    while (current != nullptr) {

        next = current -> next;

        current -> next = prev;

        prev = current;

        current = next;

    }

    head = prev;

    traverseList();

}

int main() {

    bool Continue = true;

    int n;

    cout << "Enter the number of elements you want to add to the linked list: ";

    cin >> n;

    for (int i = 0; i < n; ++i) {

        int value;

        cout << "Enter data for node " << i + 1 << ": ";

        cin >> value;

        insertAtEnd(value);

    }

    traverseList();

    while (Continue) {

        char Char;

        cout << "Do you want to continue (y/n)" << endl;

        cin >> Char;

        if (Char != 'y' && Char != 'Y') {

            Continue = false;

            cout << "Exiting program." << endl;

            break;

        }

        cout << "\nMenu:\n";

        cout << "1. Traverse In LinkedList   ";

        cout << "2. Insert at Beginning                  ";

        cout << "3. Insert at End\n";

        cout << "4. Insert after given node  ";

        cout << "5. Delete from Beginning                ";

        cout << "6. Delete from End\n";

        cout << "7. Delete the given node   ";

        cout << "8. Count number of nodes in LinkedList  ";

        cout << "9. Reverse LinkedList\n";

        cout << "Enter your choice: ";

        int choice, value;

        cin >> choice;

        switch (choice) {

        case 1:

            traverseList();

            break;

        case 2:

            insertAtBeginning();

            break;

        case 3:

            int value;

            cout << "Enter value to enter : ";

            cin >> value;

            cout << endl;

            insertAtEnd(value);

            traverseList();

            break;

        case 4:

            insertAfterGivenNode();

            break;

        case 5:

            deleteAtBeginning();

            break;

        case 6:

            deleteAtEnd();

            break;

        case 7:

            deleteTheGivenNode();

            break;

        case 8:

            count();

            break;

        case 9:

            reverseSingleLL();

            break;

        default:

            break;

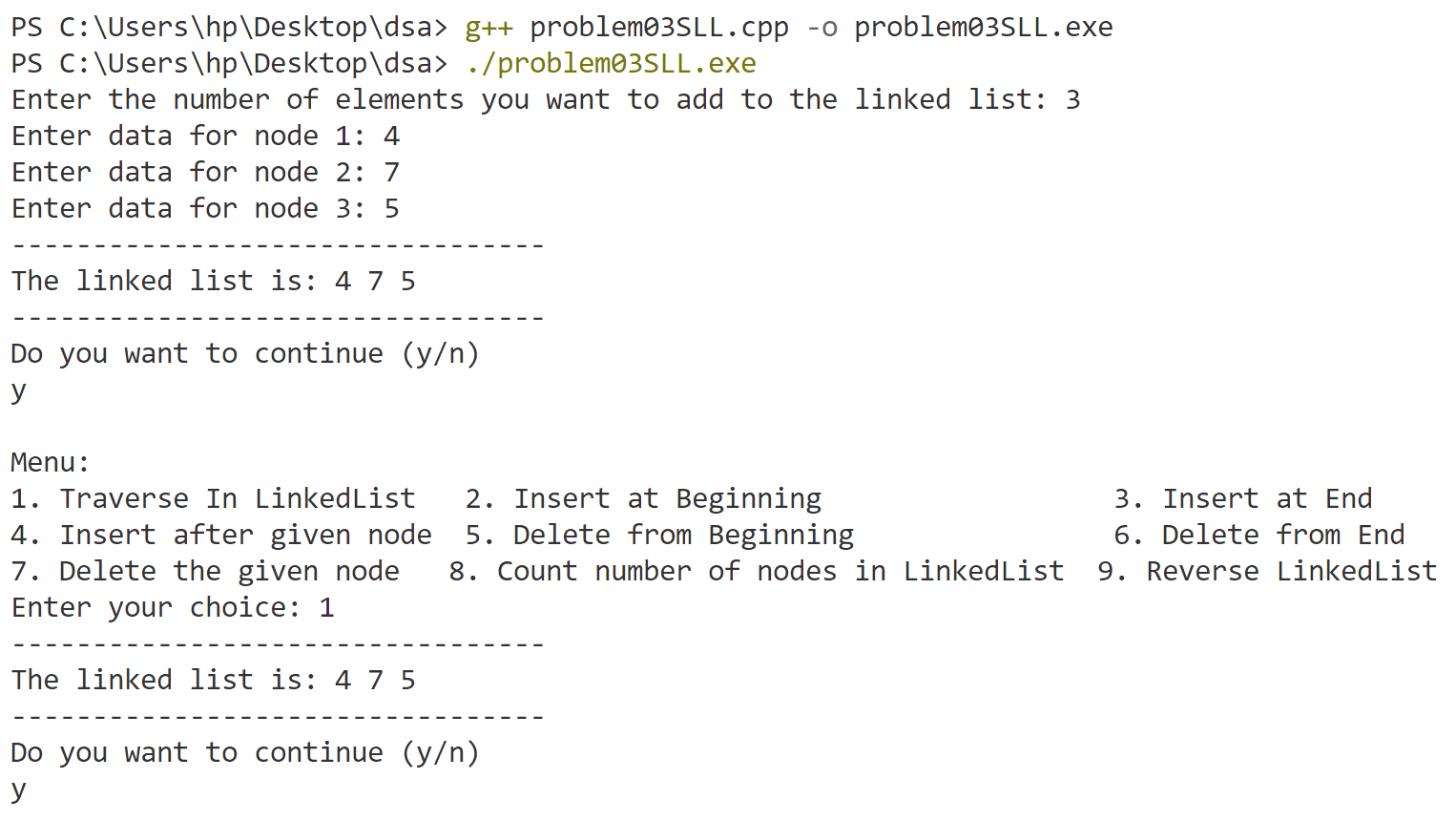
        }

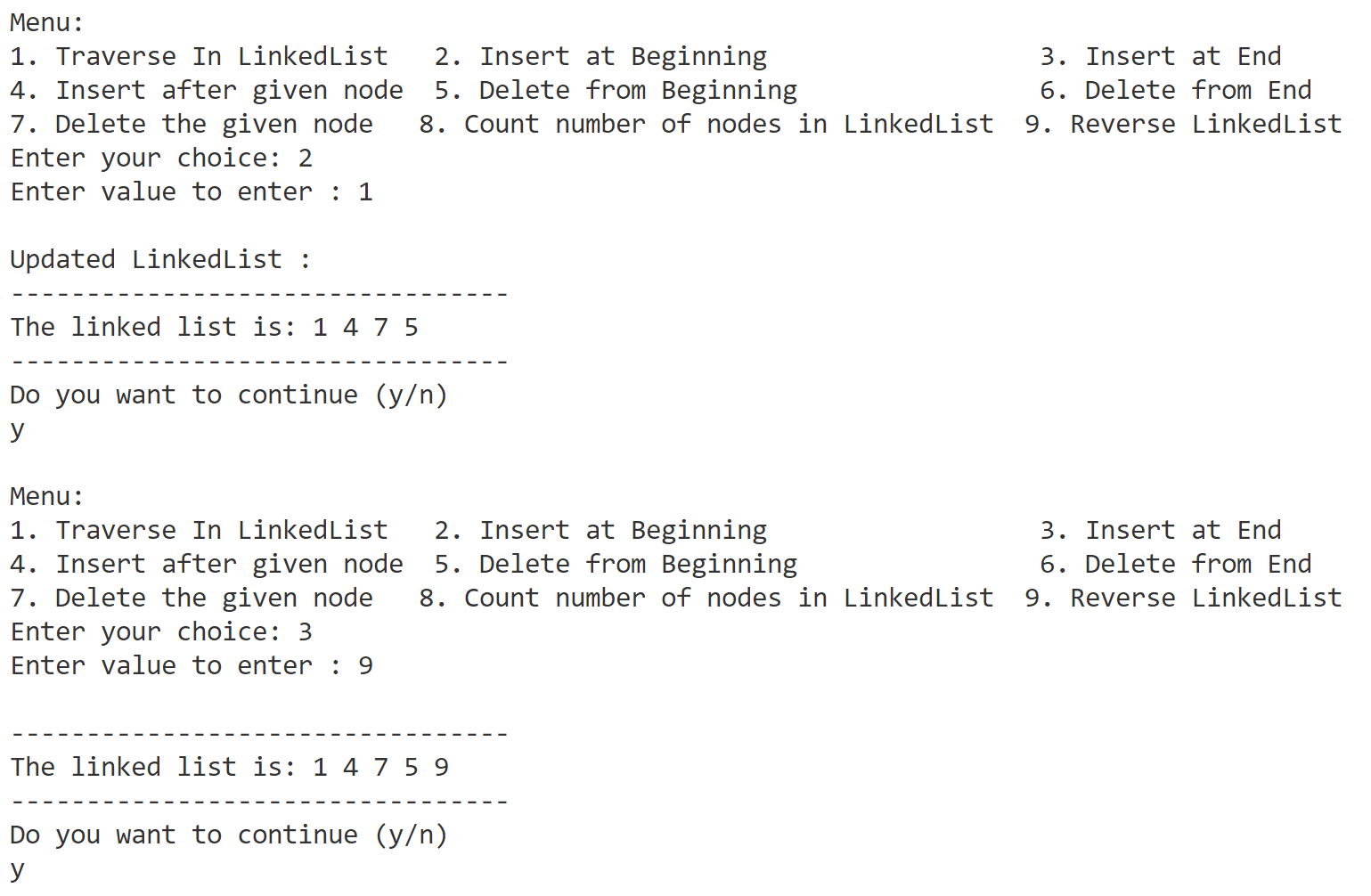
    }

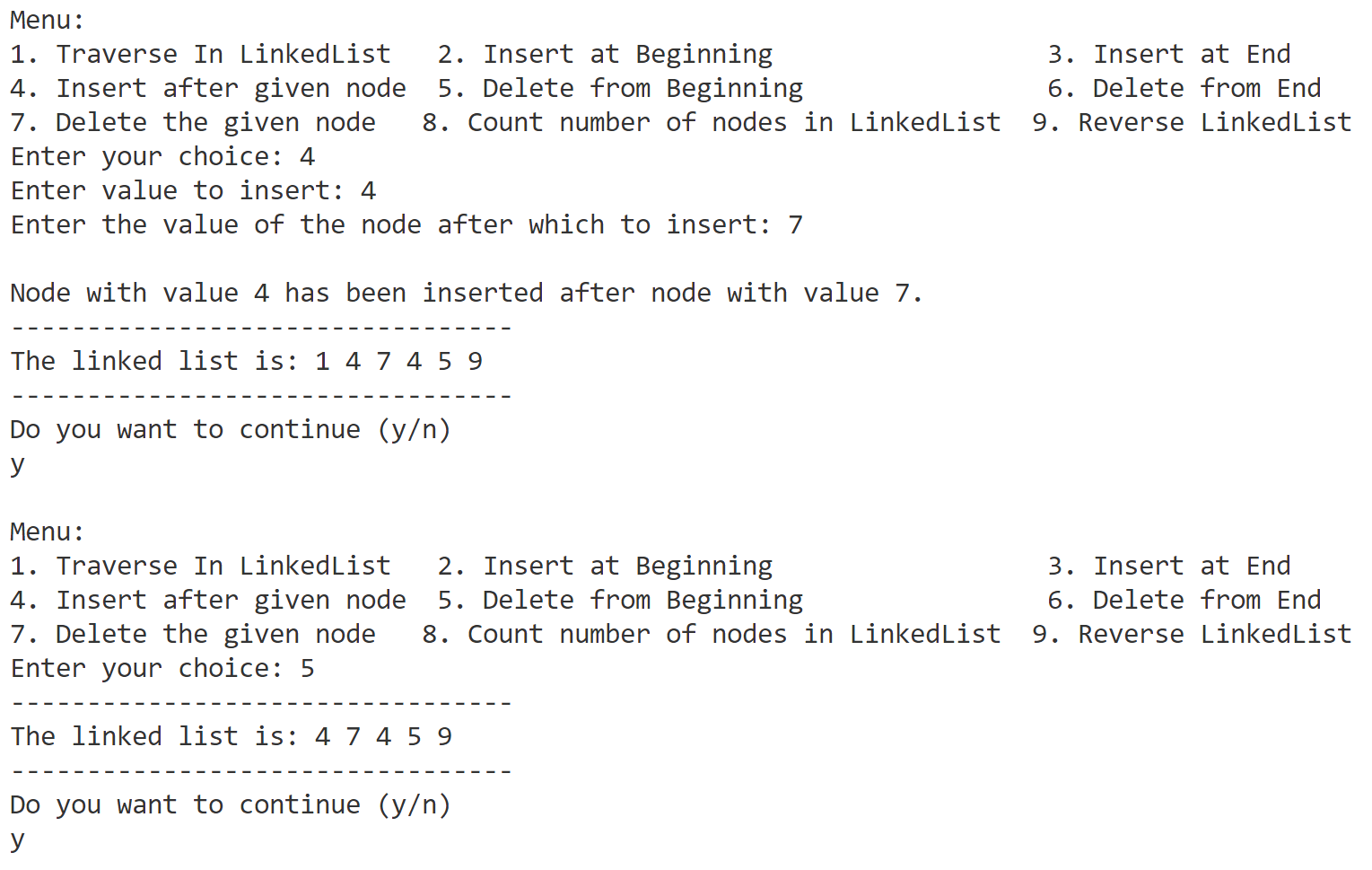
    return 0;

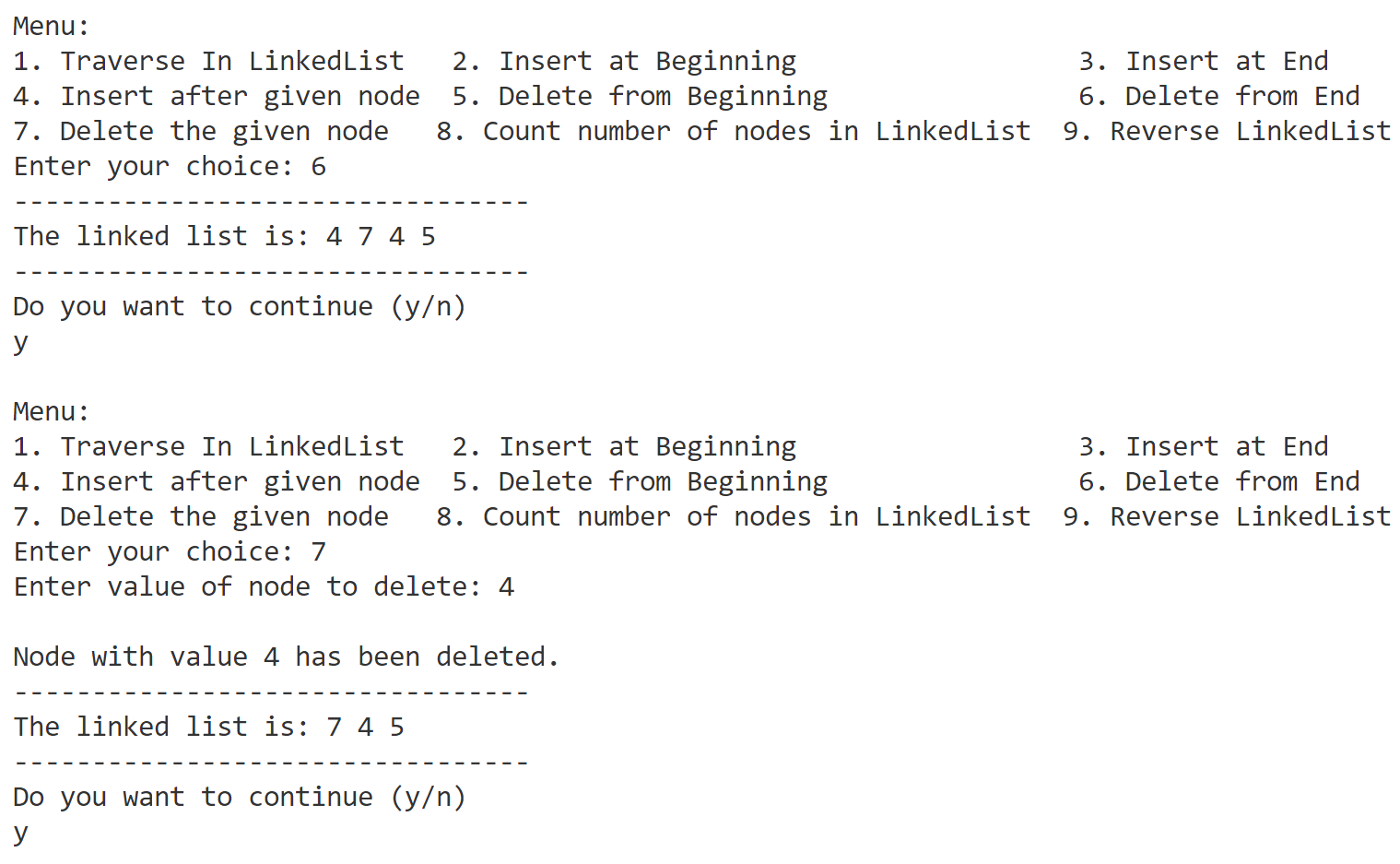
}

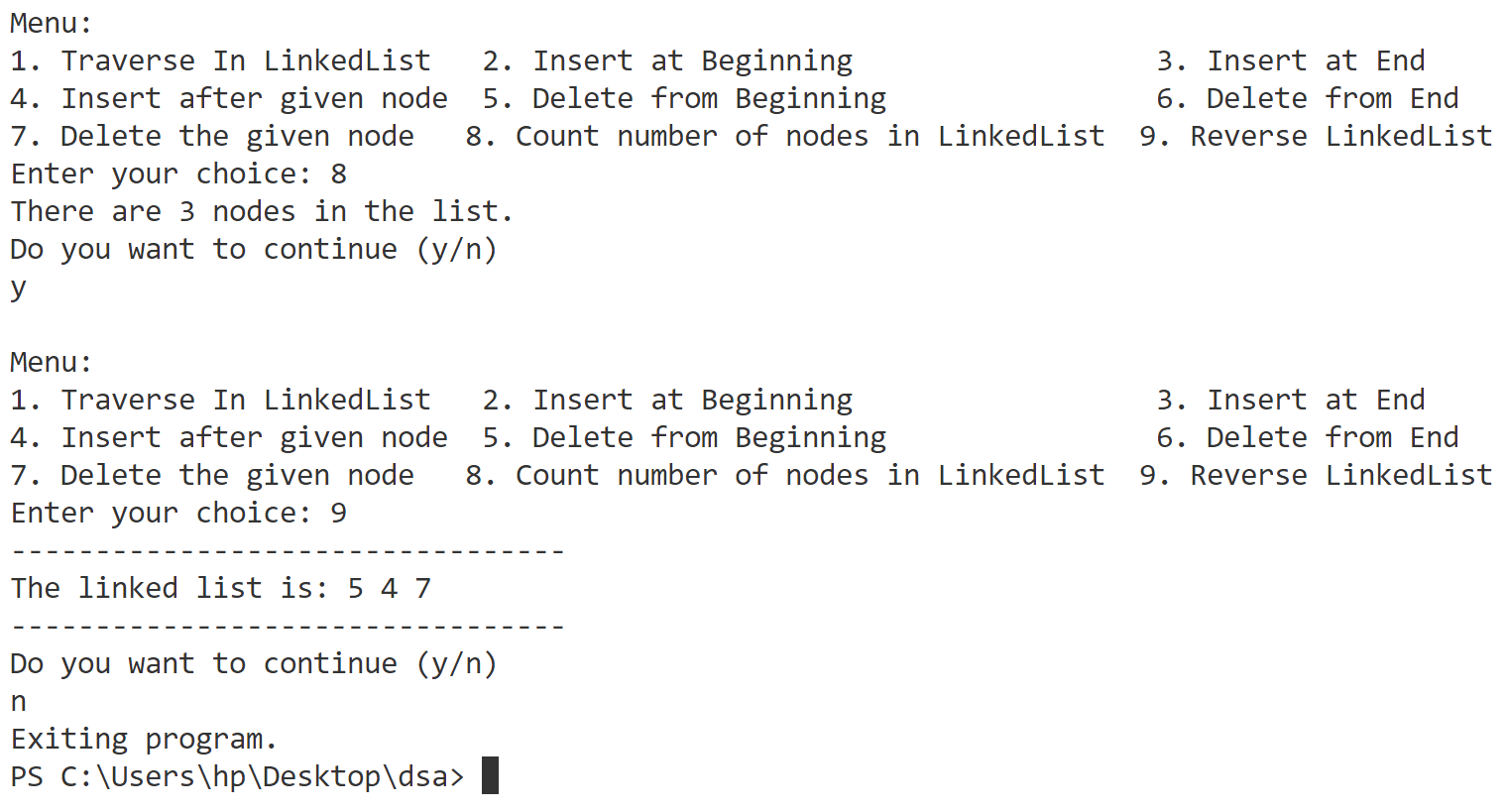
**Output**

****

****

****

****

****

1. **Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on circular single linked list.**

// circularSingleLinkedList

#include <bits/stdc++.h>

using namespace std;

struct Node {

    int data;

    Node \* next;

};

Node \* head = nullptr;

void traverseList() {

    cout << "---------------------------------" << endl;

    if (head == nullptr) {

        cout << "The list is empty." << endl;

        return;

    }

    Node \* temp = head;

    cout << "The circular singly linked list is: ";

    while (temp -> next != head) {

        cout << temp -> data << " ";

        temp = temp -> next;

    }

    cout << temp -> data << endl;

    cout << "---------------------------------" << endl;

}

void insertAtBeginning() {

    int value;

    cout << "Enter value to enter : ";

    cin >> value;

    cout << endl;

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

    newNode -> data = value;

    if (head == nullptr) {

        head = newNode;

        head -> next = head;

        cout << "Updated LinkedList :" << endl;

        traverseList();

        return;

    }

    newNode -> next = head;

    Node \* temp = head;

    while (temp -> next != head) {

        temp = temp -> next;

    }

    temp -> next = newNode;

    head = newNode;

    cout << "Updated LinkedList :" << endl;

    traverseList();

    return;

}

void insertAtEnd(int value) {

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

    newNode -> data = value;

    if (head == nullptr) {

        head = newNode;

        head -> next = head;

        return;

    }

    Node \* temp = head;

    while (temp -> next != head) {

        temp = temp -> next;

    }

    temp -> next = newNode;

    newNode -> next = head;

    return;

}

void insertAfterGivenNode() {

    int value, givenNode;

    cout << "Enter value to insert: ";

    cin >> value;

    cout << "Enter the value of the node after which to insert: ";

    cin >> givenNode;

    cout << endl;

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

    newNode -> data = value;

    if (head == nullptr) {

        newNode -> next = newNode;

        head = newNode;

        cout << "list is empty so cannot add after a node .so we are inserting at begining" << endl;

    } else {

        Node \* temp = head;

        while (true) {

            if (temp -> data == givenNode) {

                newNode -> next = temp -> next;

                temp -> next = newNode;

                break;

            }

            temp = temp -> next;

            if (temp == head) {

                cout << "cannot find the given node" << endl;

                break;

            }

        }

    }

    traverseList();

}

void deleteAtBeginning() {

    if (head == NULL) {

        cout << "List is empty. Nothing to delete.\n" <<

            endl;

        return;

    }

    if (head -> next == head) {

        free(head);

        head = NULL;

        cout << "Node deleted. List is now empty." << endl;

        return;

    }

    Node \* temp = head;

    Node \* last = head;

    while (last -> next != head) {

        last = last -> next;

    }

    last -> next = head -> next;

    head = head -> next;

    free(temp);

    traverseList();

}

void deleteAtEnd() {

    if (head == NULL) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    if (head -> next == head) {

        free(head);

        head = NULL;

        cout << "Node deleted. List is now empty." << endl;

        return;

    }

    Node \* sLast = head;

    while (sLast -> next -> next != head) {

        sLast = sLast -> next;

    }

    Node \* last = sLast -> next;

    sLast -> next = head;

    free(last);

    traverseList();

}

void deleteTheGivenNode() {

    int givenNode;

    cout << "give the value of node you want to delete" << endl;

    cin >> givenNode;

    if (head == NULL) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    if (head -> next == head) {

        if (head -> data == givenNode) {

            free(head);

            head = NULL;

            cout << "Element deleted. List is now empty." << endl;

        } else {

            cout << "Node with value " << givenNode << " not found." << endl;

        }

        traverseList();

        return;

    }

    Node \* temp = head;

    Node \* prev = nullptr;

    while (true) {

        if (temp -> data == givenNode) {

            if (prev == NULL) { // Deleting the head node

                Node \* last = head;

                while (last -> next != head) {

                    last = last -> next;

                }

                head = head -> next;

                last -> next = head;

            } else {

                prev -> next = temp -> next;

            }

            free(temp);

            cout << "Node deleted." << endl;

            traverseList();

            return;

        }

        // Move to the next node

        prev = temp;

        temp = temp -> next;

        // If we've looped back to the head, the node wasn't found

        if (temp == head) {

            cout << "Node with value " << givenNode << " not found." << endl;

            break;

        }

    }

}

int main() {

    bool Continue = true;

    int n;

    cout << "Enter the number of elements you want to add to the circular singly linked list: ";

    cin >> n;

    for (int i = 0; i < n; ++i) {

        int value;

        cout << "Enter data for node " << i + 1 << ": ";

        cin >> value;

        insertAtEnd(value);

    }

    traverseList();

    while (Continue) {

        char Char;

        cout << "Do you want to continue (y/n)" << endl;

        cin >> Char;

        if (Char != 'y' && Char != 'Y') {

            Continue = false;

            cout << "Exiting program." << endl;

            break;

        }

        cout << "\nMenu:\n";

        cout << "1. Traverse In LinkedList   ";

        cout << "2. Insert at Beginning    ";

        cout << "3. Insert at End\n";

        cout << "4. Insert after given node  ";

        cout << "5. Delete from Beginning  ";

        cout << "6. Delete from End\n";

        cout << "7. Delete the given node\n";

        cout << "Enter your choice: ";

        int choice, value;

        cin >> choice;

        switch (choice) {

        case 1:

            traverseList();

            break;

        case 2:

            insertAtBeginning();

            break;

        case 3:

            int value;

            cout << "Enter value to enter : ";

            cin >> value;

            cout << endl;

            insertAtEnd(value);

            traverseList();

            break;

        case 4:

            insertAfterGivenNode();

            break;

        case 5:

            deleteAtBeginning();

            break;

        case 6:

            deleteAtEnd();

            break;

        case 7:

            deleteTheGivenNode();

            break;

        default:

            break;

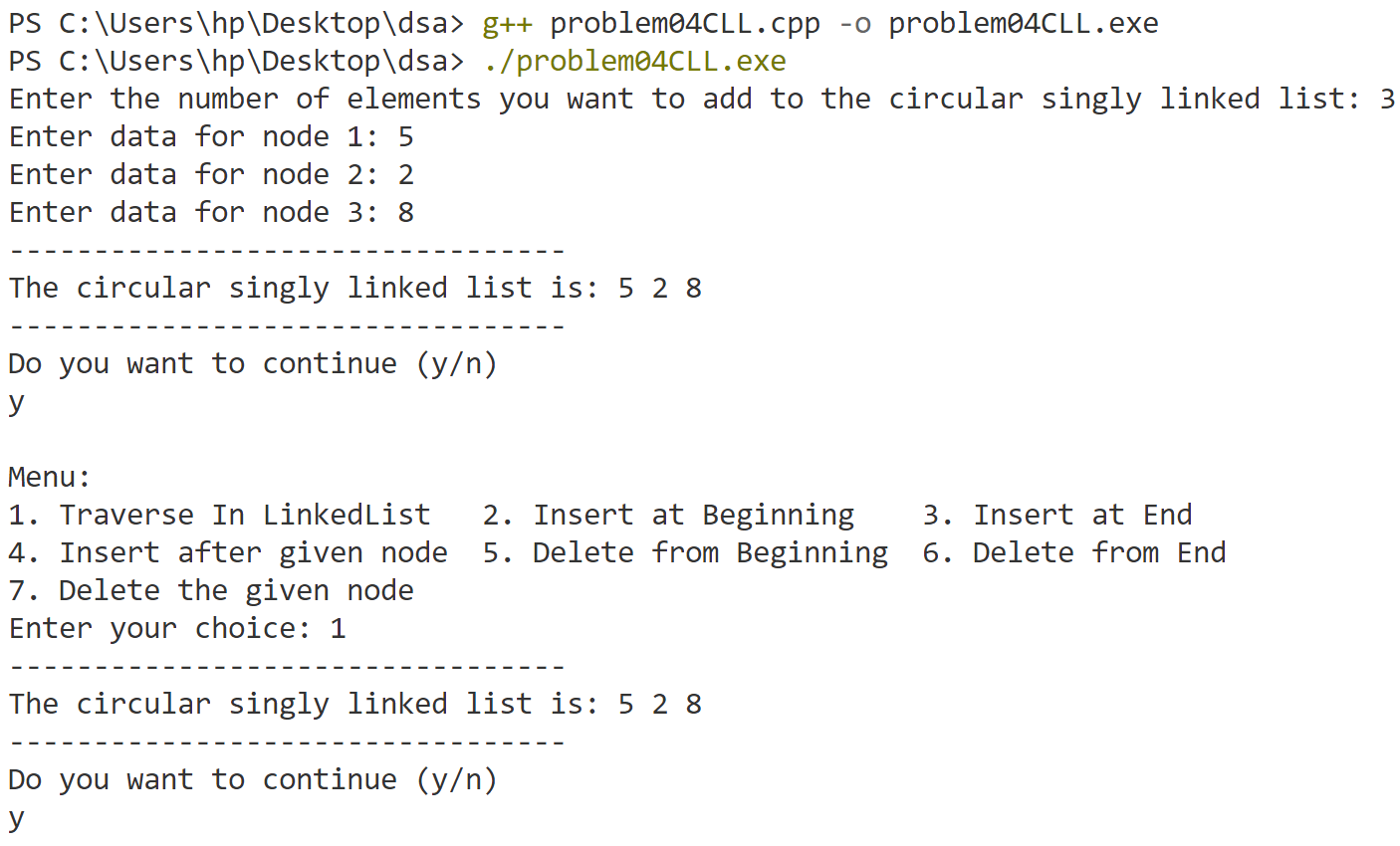
        }

    }

    return 0;

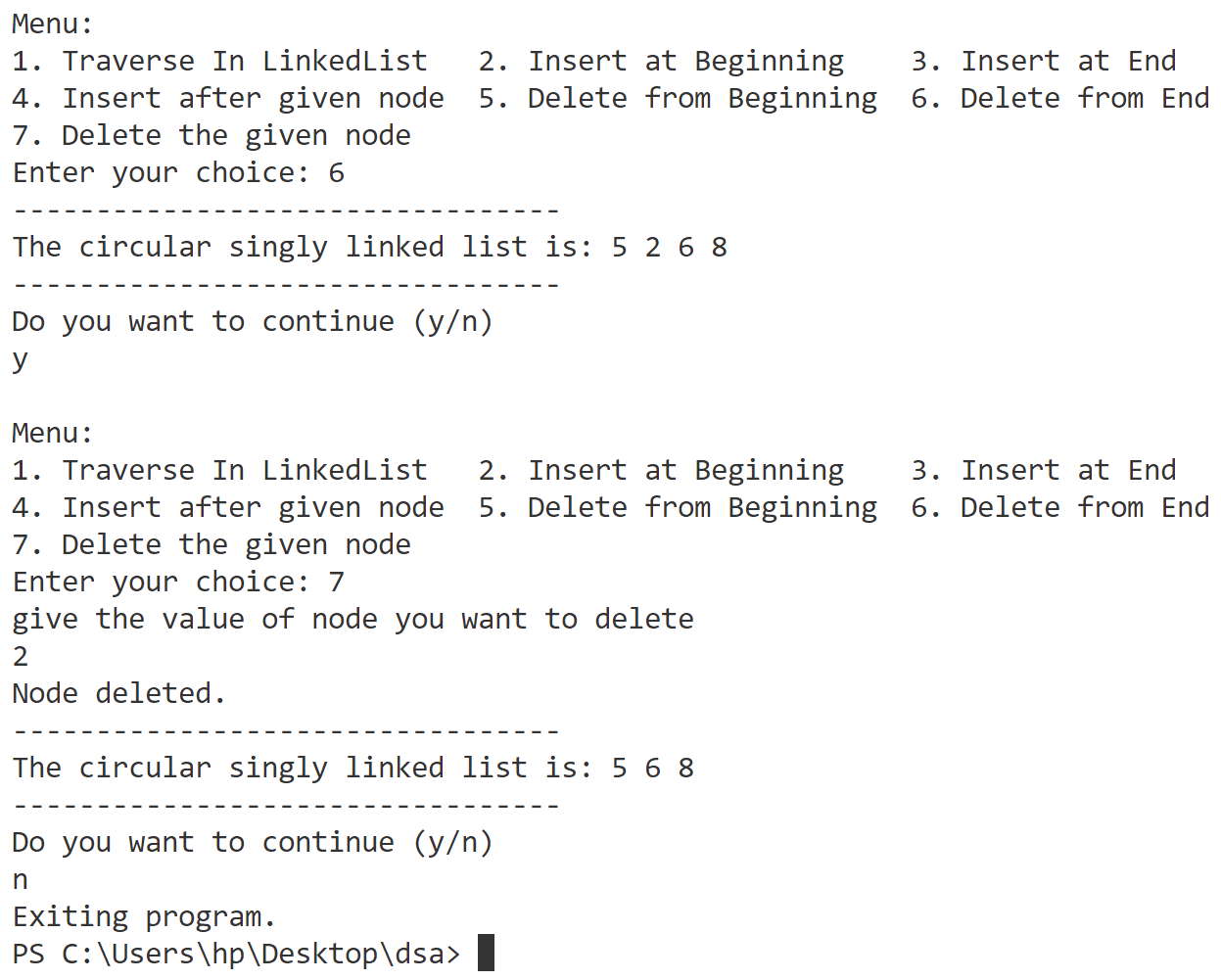
}

**Output**

****

****

****

****

1. **Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on double linked list.**

// doublyLinkedList

#include <bits/stdc++.h>

using namespace std;

struct Node {

    int data;

    Node \* next;

    Node \* back;

};

Node \* head = nullptr;

void traverseList() {

    cout << "---------------------------------" << endl;

    if (head == nullptr) {

        cout << "The list is empty." << endl;

        return;

    }

    Node \* temp = head;

    cout << "The doubly linked list is: ";

    while (temp -> next != nullptr) {

        cout << temp -> data << " ";

        temp = temp -> next;

    }

    cout << temp -> data << endl;

    cout << "---------------------------------" << endl;

}

void insertAtBeginning() {

    int value;

    cout << "Enter value to enter : ";

    cin >> value;

    cout << endl;

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

    newNode -> data = value;

    newNode -> back = nullptr;

    if (head == nullptr) {

        head = newNode;

        head -> next = nullptr;

    } else {

        newNode -> next = head;

        head -> back = newNode;

        head = newNode;

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

    return;

}

void insertAtEnd(int value) {

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

    newNode -> data = value;

    newNode -> next = nullptr;

    if (head == nullptr) {

        head = newNode;

        head -> back = nullptr;

        return;

    }

    Node \* temp = head;

    while (temp -> next != nullptr) {

        temp = temp -> next;

    }

    temp -> next = newNode;

    newNode -> back = temp;

    return;

}

void insertAfterGivenNode() {

    int value, givenNode;

    cout << "Enter value to insert: ";

    cin >> value;

    cout << "Enter the value of the node after which to insert: ";

    cin >> givenNode;

    cout << endl;

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

    newNode -> data = value;

    if (head == nullptr) {

        newNode -> next = nullptr;

        newNode -> back = nullptr;

        head = newNode;

        cout << "list is empty so cannot add after a node .so we are inserting at begining" << endl;

    } else {

        Node \* temp = head;

        bool found = false;

        while (temp != nullptr) {

            if (temp -> data == givenNode) {

                newNode -> next = temp -> next;

                newNode -> back = temp;

                if (temp -> next != nullptr) {

                    temp -> next -> back = newNode;

                }

                temp -> next = newNode;

                found = true;

                break;

            }

            temp = temp -> next;

        }

        if (!found) {

            cout << "Node with value " << givenNode << " not found." << endl;

            return;

        }

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

    return;

}

void deleteAtBeginning() {

    if (head == nullptr) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    if (head -> next == nullptr) {

        free(head);

        head = nullptr;

        cout << "Node deleted. List is now empty." << endl;

    } else {

        Node \* temp = head;

        head = head -> next;

        free(temp);

        head -> back = nullptr;

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

    return;

}

void deleteAtEnd() {

    if (head == nullptr) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    Node \* temp = head;

    while (temp -> next != nullptr) {

        temp = temp -> next;

    }

    if (temp == head) {

        free(temp);

        head = nullptr;

        cout << "Node deleted. List is now empty." << endl;

        return;

    }

    temp -> back -> next = nullptr;

    free(temp);

    cout << "Updated LinkedList :" << endl;

    traverseList();

}

void deleteTheGivenNode() {

    int givenNode;

    cout << "give the value of node you want to delete" << endl;

    cin >> givenNode;

    if (head == nullptr) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    Node \* temp = head;

    bool found = false;

    while (temp != nullptr) {

        if (temp -> data == givenNode) {

            if (temp == head) {

                head = temp -> next;

                if (head != nullptr) {

                    head -> back = nullptr;

                }

                free(temp);

                found = true;

                break;

            } //Node is in the middle or end

            else {

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

                if (temp -> next != nullptr) {

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

                }

                free(temp);

                found = true;

                break;

            }

        }

        temp = temp -> next;

    }

    if (!found) {

        cout << "Node with value " << givenNode << " not found." << endl;

        return;

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

}

int main() {

    bool Continue = true;

    int n;

    cout << "Enter the number of elements you want to add to the doubly linked list: ";

    cin >> n;

    for (int i = 0; i < n; ++i) {

        int value;

        cout << "Enter data for node " << i + 1 << ": ";

        cin >> value;

        insertAtEnd(value);

    }

    traverseList();

    while (Continue) {

        char Char;

        cout << "Do you want to continue (y/n)" << endl;

        cin >> Char;

        if (Char != 'y' && Char != 'Y') {

            Continue = false;

            cout << "Exiting program." << endl;

            break;

        }

        cout << "\nMenu:\n";

        cout << "1. Traverse In LinkedList   ";

        cout << "2. Insert at Beginning    ";

        cout << "3. Insert at End\n";

        cout << "4. Insert after given node  ";

        cout << "5. Delete from Beginning  ";

        cout << "6. Delete from End\n";

        cout << "7. Delete the given node\n";

        cout << "Enter your choice: ";

        int choice, value;

        cin >> choice;

        switch (choice) {

        case 1:

            traverseList();

            break;

        case 2:

            insertAtBeginning();

            break;

        case 3:

            int value;

            cout << "Enter value to enter : ";

            cin >> value;

            cout << endl;

            insertAtEnd(value);

            traverseList();

            break;

        case 4:

            insertAfterGivenNode();

            break;

        case 5:

            deleteAtBeginning();

            break;

        case 6:

            deleteAtEnd();

            break;

        case 7:

            deleteTheGivenNode();

            break;

        default:

            break;

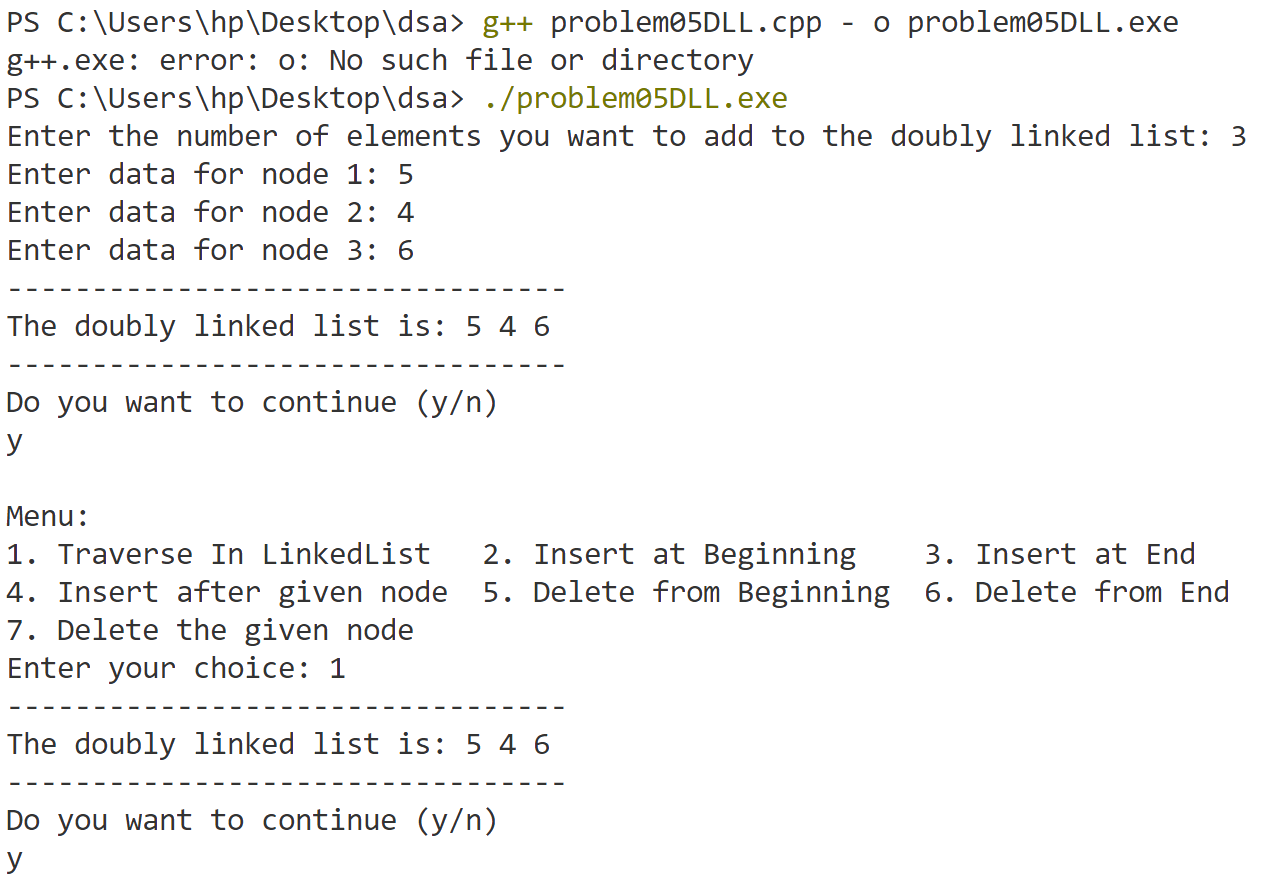
        }

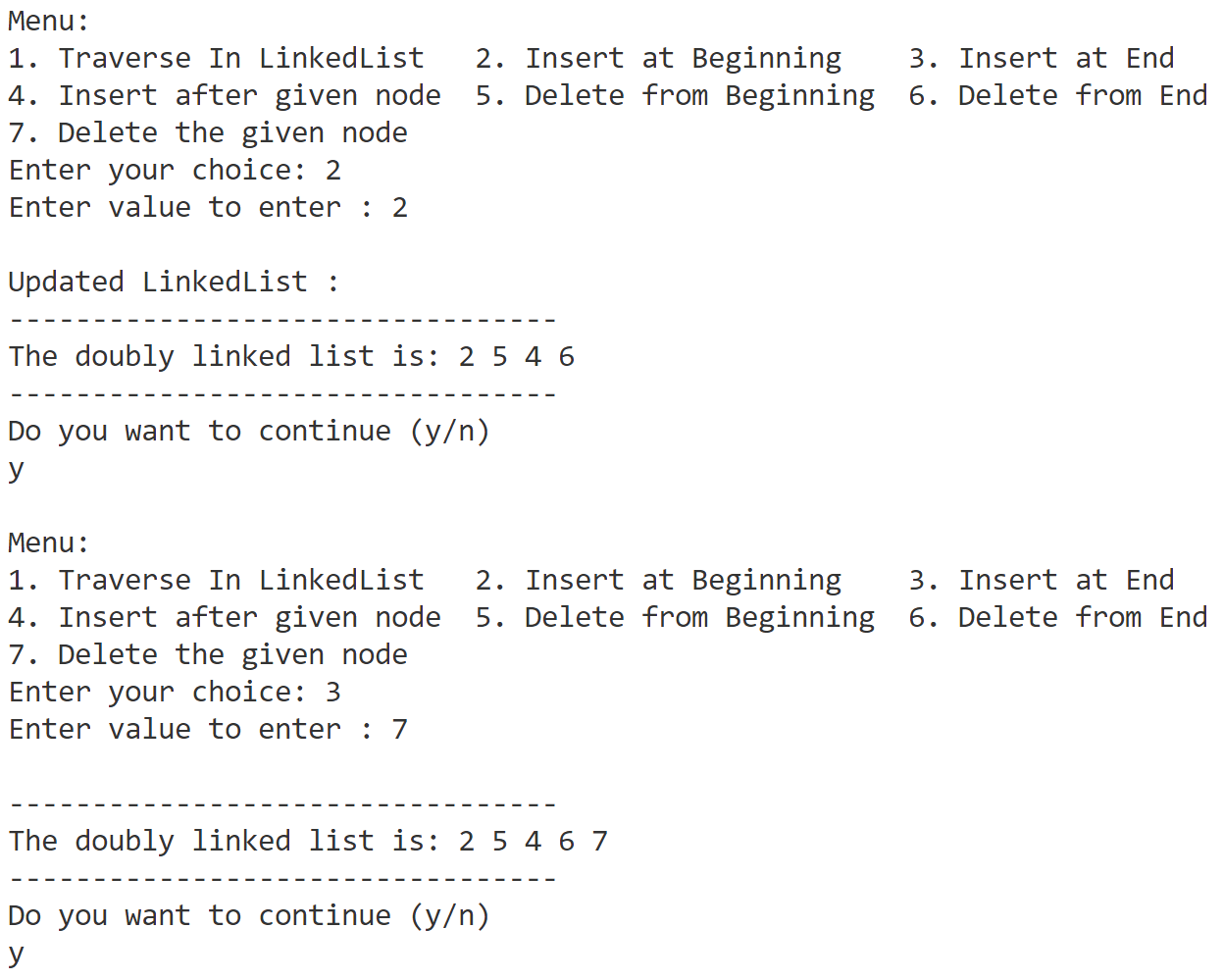
    }

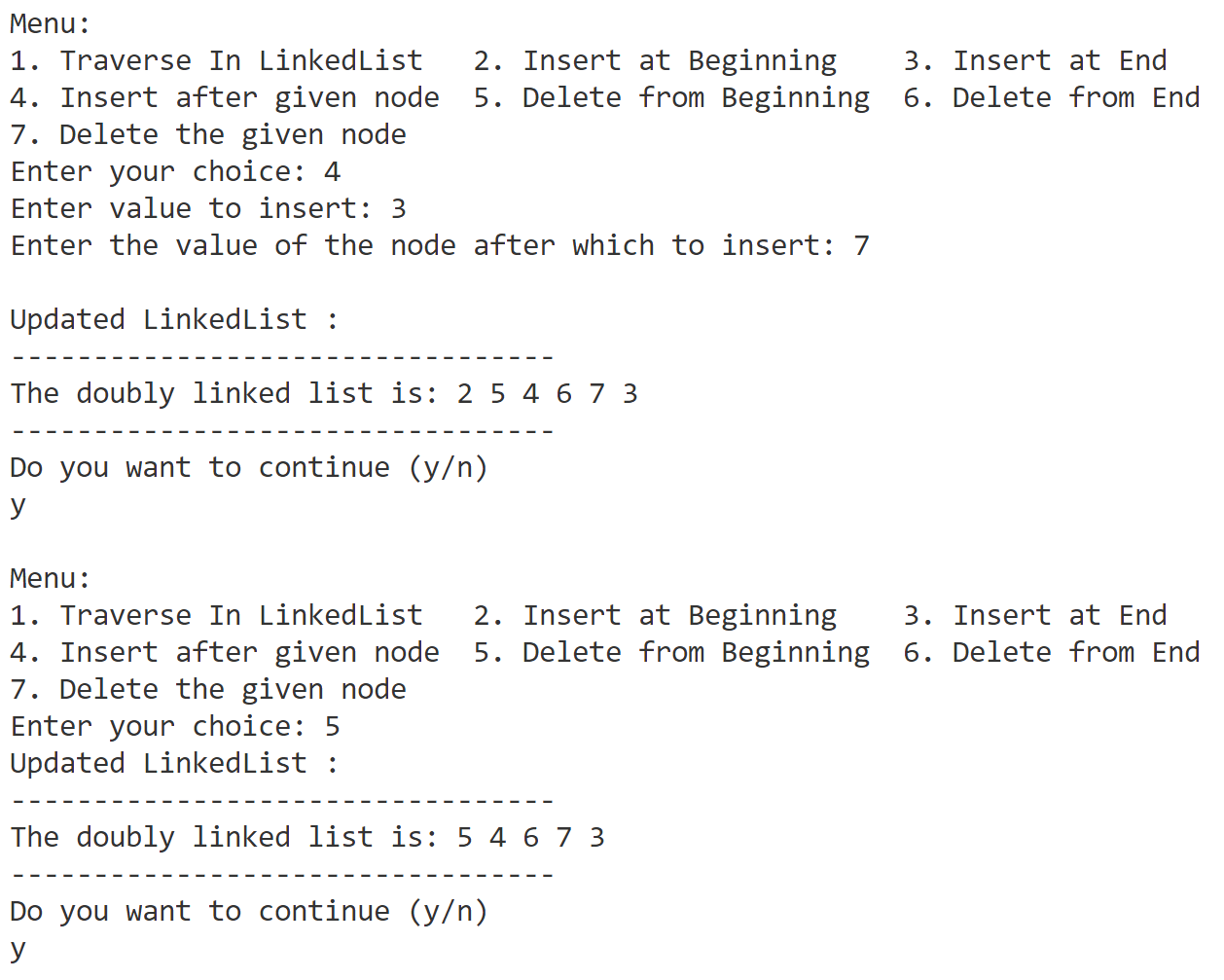
    return 0;

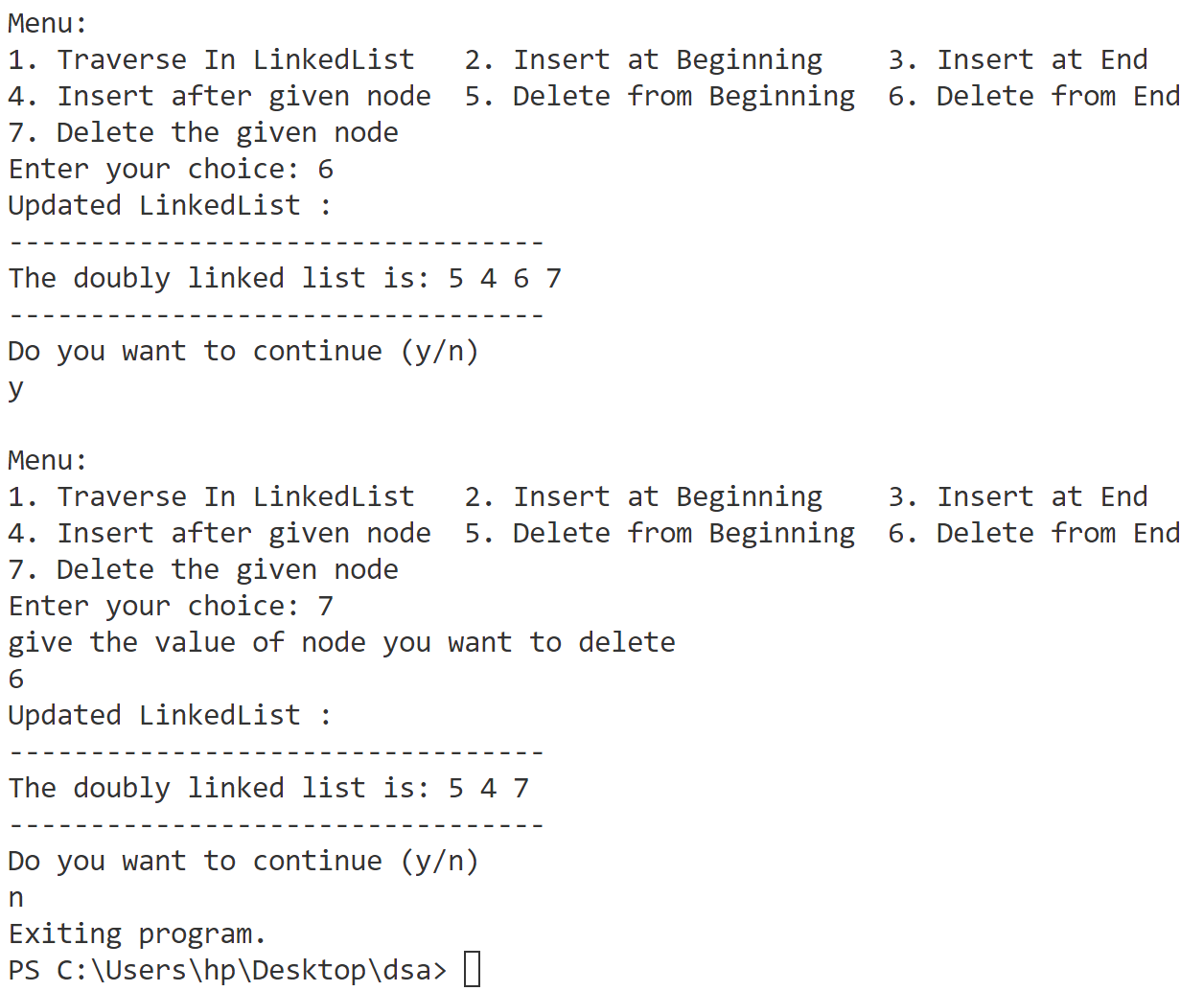
}

**Output**

****

****

****

****

1. **Implement insertion (at the beginning, at specified location, at the end) and deletion (at the beginning, at specified location, at the end) on circular double linked list.**

// circulardoublyLinkedList

#include <bits/stdc++.h>

using namespace std;

// structor of 1d linkedList

struct Node {

    int data;

    Node \* next;

    Node \* back;

};

Node \* head = nullptr;

void traverseList() {

    cout << "---------------------------------" << endl;

    if (head == nullptr) {

        cout << "The list is empty." << endl;

        return;

    }

    Node \* temp = head;

    cout << "The doubly linked list is: ";

    while (true) {

        cout << temp -> data << " ";

        temp = temp -> next;

        if (temp == head) {

            break;

        }

    }

    cout << endl;

    cout << "---------------------------------" << endl;

}

void insertAtBeginning() {

    int value;

    cout << "Enter value to enter : ";

    cin >> value;

    cout << endl;

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

    newNode -> data = value;

    if (head == nullptr) {

        newNode -> next = newNode;

        newNode -> back = newNode;

        head = newNode;

    } else {

        newNode -> next = head;

        newNode -> back = head -> back;

        head -> back -> next = newNode;

        head -> back = newNode;

        head = newNode;

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

    return;

}

void insertAtEnd(int value) {

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

    newNode -> data = value;

    if (head == nullptr) {

        newNode -> back = newNode;

        newNode -> next = newNode;

        head = newNode;

    } else {

        Node \* last = head -> back;

        last -> next = newNode;

        newNode -> back = last;

        newNode -> next = head;

        head -> back = newNode;

    }

    return;

}

void insertAfterGivenNode() {

    int value, givenNode;

    cout << "Enter value to insert: ";

    cin >> value;

    cout << "Enter the value of the node after which to insert: ";

    cin >> givenNode;

    cout << endl;

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

    newNode -> data = value;

    if (head == nullptr) {

        newNode -> next = newNode;

        newNode -> back = newNode;

        head = newNode;

        cout << "list is empty so cannot add after a node .so we are inserting at begining" << endl;

    } else {

        Node \* temp = head;

        bool found = false;

        while (true) {

            if (temp -> data == givenNode) {

                newNode -> next = temp -> next;

                newNode -> back = temp;

                temp -> next -> back = newNode;

                temp -> next = newNode;

                found = true;

                break;

            }

            temp = temp -> next;

            if (temp == head) {

                break;

            }

        }

        if (!found) {

            cout << "Node with value " << givenNode << " not found." << endl;

            return;

        }

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

    return;

}

void deleteAtBeginning() {

    if (head == nullptr) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    if (head -> next == head) {

        free(head);

        head = nullptr;

        cout << "Node deleted. List is now empty." << endl;

        return;

    } else {

        Node \* temp = head;

        head = head -> next;

        head -> back = temp -> back;

        temp -> back -> next = head;

        free(temp);

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

    return;

}

void deleteAtEnd() {

    if (head == nullptr) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    if (head -> next == head) {

        free(head);

        head = nullptr;

        cout << "Node deleted. List is now empty." << endl;

        return;

    } else {

        Node \* temp = head -> back;

        head -> back -> back -> next = head;

        head -> back = head -> back -> back;

        free(temp);

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

}

void deleteTheGivenNode() {

    int givenNode;

    cout << "give the value of node you want to delete" << endl;

    cin >> givenNode;

    if (head == nullptr) {

        cout << "List is empty. Nothing to delete." << endl;

        return;

    }

    Node \* temp = head;

    bool found = false;

    while (true) {

        if (temp -> data == givenNode) {

            if (temp == head) {

                found = true;

                deleteAtBeginning();

                return;

            } else {

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

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

                free(temp);

                found = true;

                break;

            }

        }

        temp = temp -> next;

        if (temp == head) {

            break;

        }

    }

    if (!found) {

        cout << "Node with value " << givenNode << " not found." << endl;

        return;

    }

    cout << "Updated LinkedList :" << endl;

    traverseList();

}

int main() {

    bool Continue = true;

    int n;

    cout << "Enter the number of elements you want to add to the circular doubly linked list: ";

    cin >> n;

    for (int i = 0; i < n; ++i) {

        int value;

        cout << "Enter data for node " << i + 1 << ": ";

        cin >> value;

        insertAtEnd(value);

    }

    traverseList();

    while (Continue) {

        char Char;

        cout << "Do you want to continue (y/n)" << endl;

        cin >> Char;

        if (Char != 'y' && Char != 'Y') {

            Continue = false;

            cout << "Exiting program." << endl;

            break;

        }

        cout << "\nMenu:\n";

        cout << "1. Traverse In LinkedList   ";

        cout << "2. Insert at Beginning    ";

        cout << "3. Insert at End\n";

        cout << "4. Insert after given node  ";

        cout << "5. Delete from Beginning  ";

        cout << "6. Delete from End\n";

        cout << "7. Delete the given node\n";

        cout << "Enter your choice: ";

        int choice, value;

        cin >> choice;

        switch (choice) {

        case 1:

            traverseList();

            break;

        case 2:

            insertAtBeginning();

            break;

        case 3:

            int value;

            cout << "Enter value to enter : ";

            cin >> value;

            cout << endl;

            insertAtEnd(value);

            traverseList();

            break;

        case 4:

            insertAfterGivenNode();

            break;

        case 5:

            deleteAtBeginning();

            break;

        case 6:

            deleteAtEnd();

            break;

        case 7:

            deleteTheGivenNode();

            break;

        default:

            break;

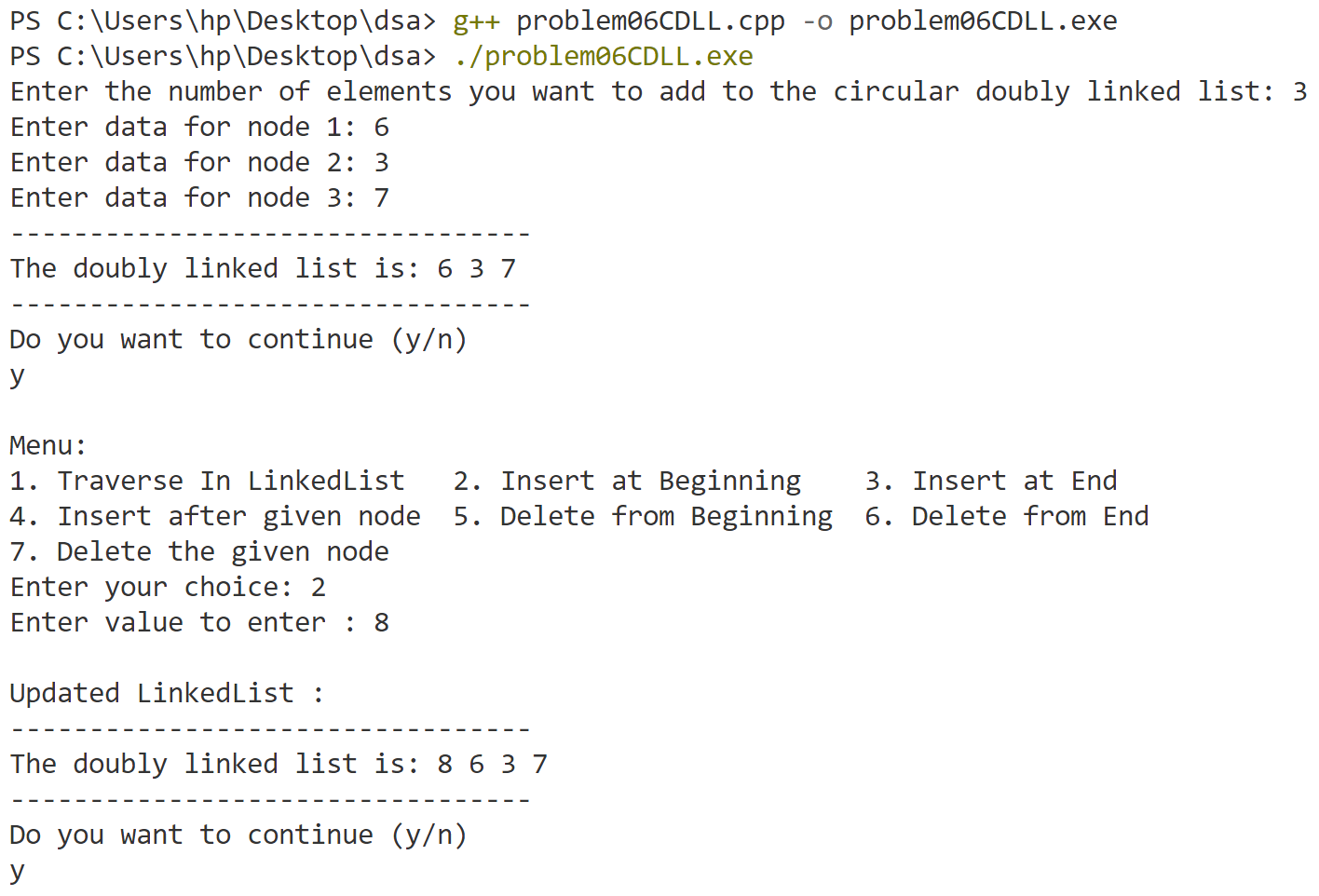
        }

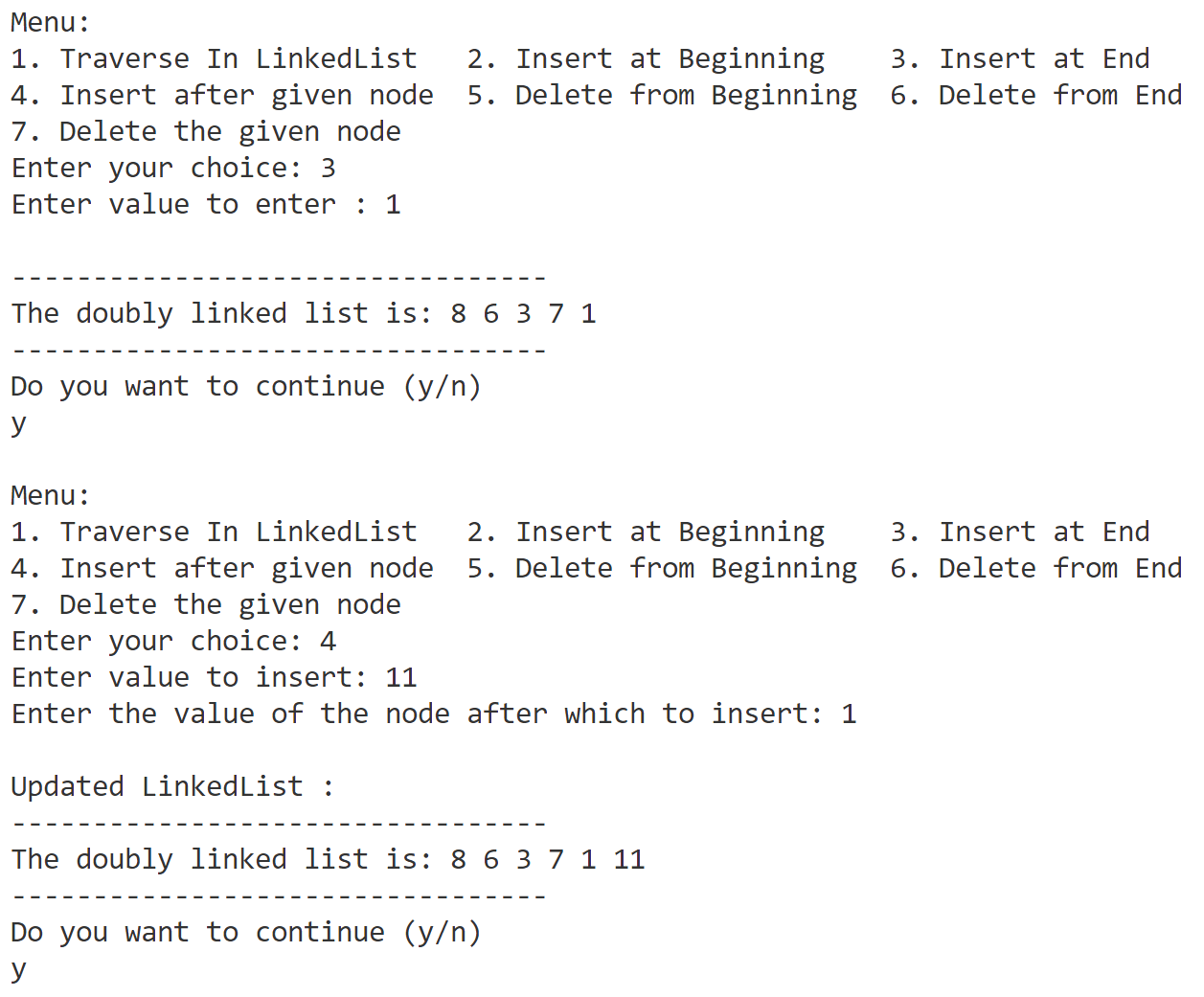
    }

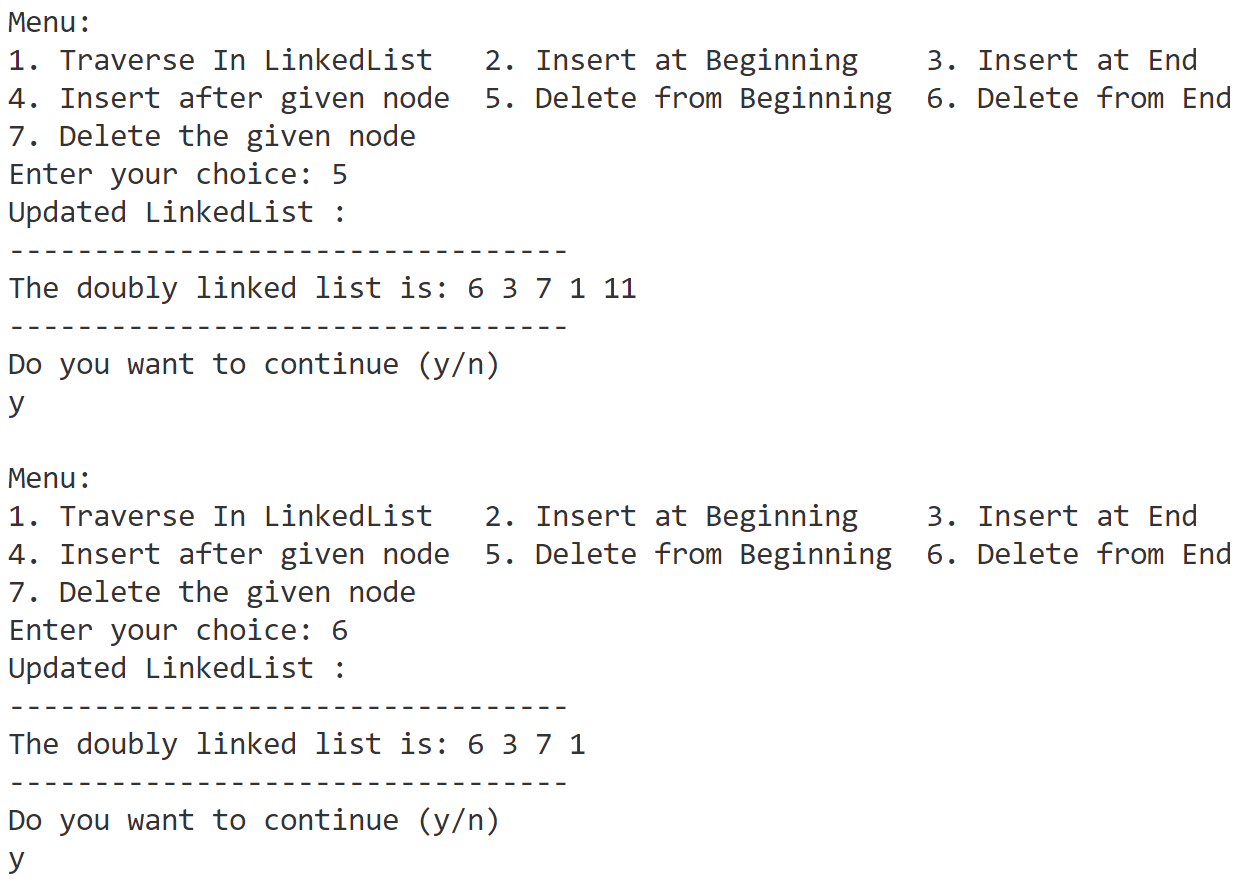
    return 0;

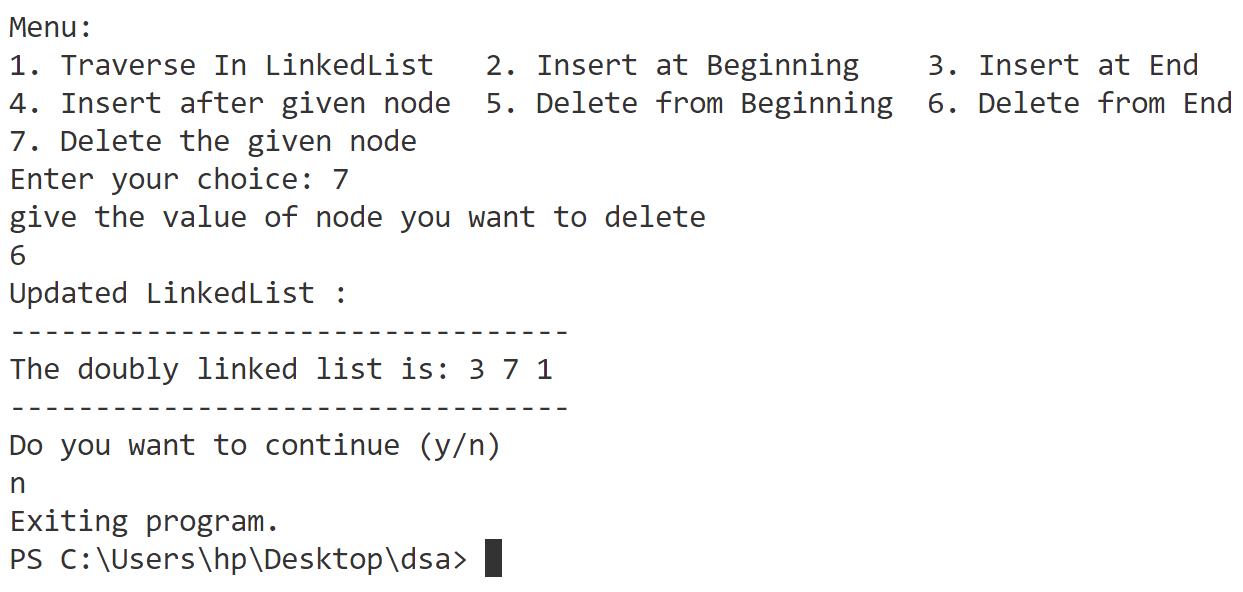
}

**Output**

****

****

****

****

1. **Write a program to merge two sorted linked list and display the final sorted linked list.**

#include <iostream>

#include <vector>

using namespace std;

struct Node {

    int data;

    Node \* next;

};

void insertAtEnd(int value, Node \* & head) {

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

    newNode -> data = value;

    newNode -> next = nullptr;

    // list is empty

    if (head == nullptr) {

        head = newNode;

    } else {

        // Traverse to the end of the list

        Node \* temp = head;

        while (temp -> next != nullptr) {

            temp = temp -> next;

        }

        temp -> next = newNode;

    }

}

Node \* MergeLinkedList(Node \* head1, Node \* head2) {

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

    Node \* temp = newNode;

    while (head1 != nullptr && head2 != nullptr) {

        if (head1 -> data < head2 -> data) {

            temp -> next = head1;

            head1 = head1 -> next;

        } else if (head1 -> data > head2 -> data) {

            temp -> next = head2;

            head2 = head2 -> next;

        } else {

            temp -> next = head1;

            head1 = head1 -> next;

            temp = temp -> next;

            temp -> next = head2;

            head2 = head2 -> next;

        }

        temp = temp -> next;

    }

    // the remaining nodes

    if (head1 != nullptr)

        temp -> next = head1;

    if (head2 != nullptr)

        temp -> next = head2;

    return newNode -> next; // Return merged list

}

void Display(Node \* head) {

    cout << "----------------------" << endl;

    if (head == nullptr) {

        cout << "List is empty!" << endl;

        return;

    }

    while (head != nullptr) {

        cout << head -> data << " ";

        head = head -> next;

    }

    cout << endl;

    cout << "----------------------" << endl;

}

int main() {

    Node \* head1 = nullptr;

    Node \* head2 = nullptr;

    int n;

    cout << "Enter the number of elements you want to add to the linked list: ";

    cin >> n;

    for (int i = 0; i < n; ++i) {

        int value;

        cout << "Enter data for First LL node " << i + 1 << ": ";

        cin >> value;

        insertAtEnd(value, head1);

    }

    cout << "LL1:" << endl;

    Display(head1);

    int m;

    cout << "Enter the number of elements you want to add to the linked list: ";

    cin >> m;

    for (int i = 0; i < m; ++i) {

        int value;

        cout << "Enter data for second LL node " << i + 1 << " : ";

        cin >> value;

        insertAtEnd(value, head2);

    }

    cout << "LL2:" << endl;

    Display(head2);

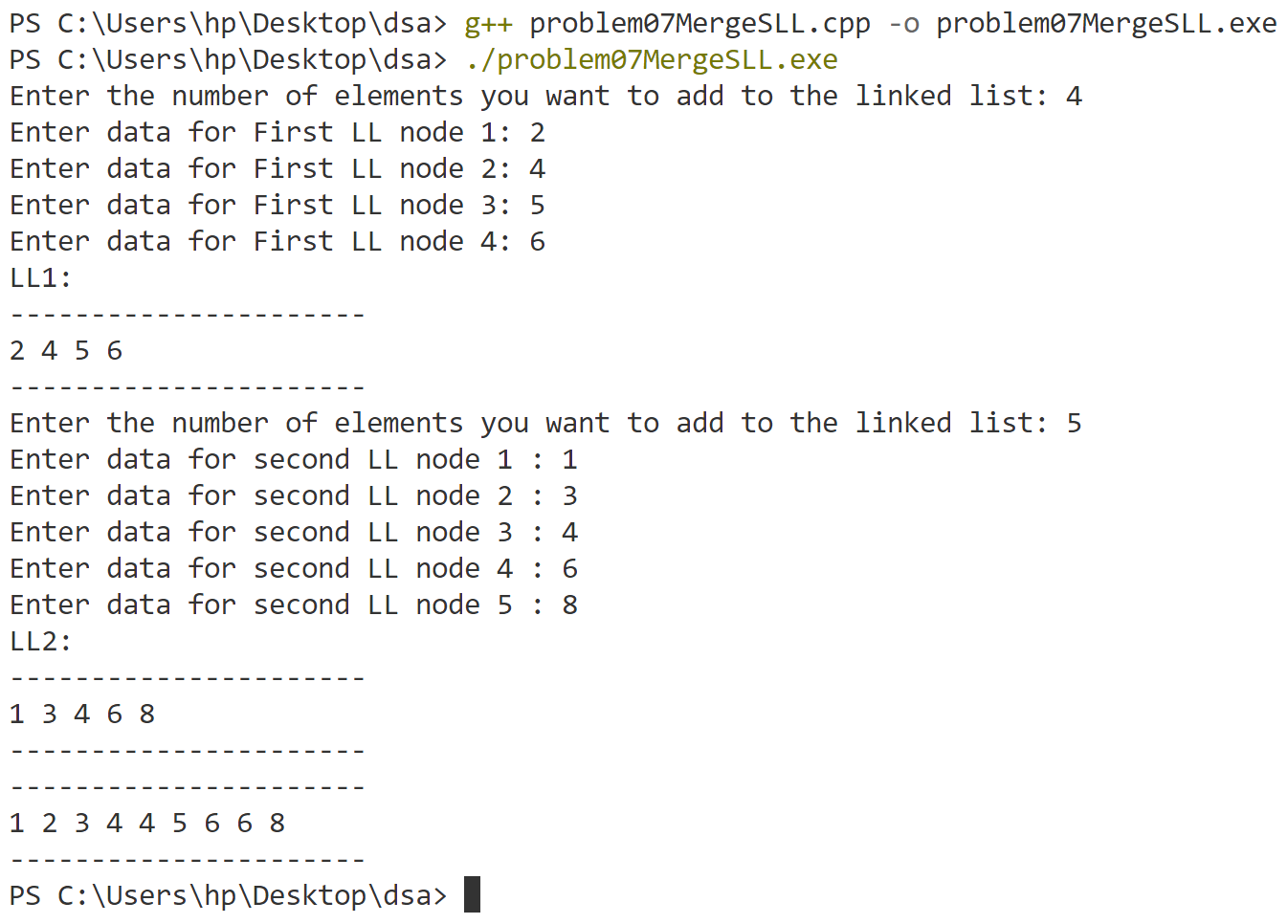
    Node \* head = MergeLinkedList(head1, head2);

    Display(head);

    return 0;

}

**Output**



1. **Implement operations (push, pop) on a stack using arrays. Check the status of the stack whether there is underflow or overflow.**

#include <iostream>

using namespace std;

int stack[100];

int n = 0, top = -1;

void createStack() {

    cout << "Enter the size of the stack (max 100): ";

    cin >> n;

    if (n > 100 || n <= 0) {

        cout << "Invalid size. Setting stack size to 100." << endl;

        n = 100;

    }

    top = -1;

}

void push(int val) {

    if (top >= n - 1) {

        cout << "Stack Overflow" << endl;

    } else {

        top++;

        stack[top] = val;

        cout << val << " pushed to stack." << endl;

    }

}

void pop() {

    if (top <= -1) {

        cout << "Stack Underflow" << endl;

    } else {

        cout << "The popped element is " << stack[top] << endl;

        top--;

    }

}

int peek() {

    if (top <= -1) {

        cout << "Stack is empty." << endl;

        return -1; // Indicate empty stack

    } else {

        return stack[top];

    }

}

bool isEmpty() {

    return (top == -1);

}

bool isFull() {

    return (top >= n - 1);

}

int size() {

    return top + 1; // Size is top index + 1

}

int main() {

    int ch, val;

    cout << "Creating Stack :" << endl;

    createStack();

    cout << "STACK CREATED" << endl;

    cout << endl;

    bool conti = true;

    while (conti) {

        cout << "=========================================================" << endl;

        cout << "(1) Push in stack             ";

        cout << "(2) Pop from stack             ";

        cout << "(3) Peek at top element" << endl;

        cout << "(4) Check if stack is empty   ";

        cout << "(5) Check if stack is full     ";

        cout << "(6) Get stack size       " << endl;

        cout << "Enter choice: ";

        cin >> ch;

        switch (ch) {

        case 1: {

            cout << "Enter value to be pushed [0-9]:";

            cin >> val;

            push(val);

            break;

        }

        case 2: {

            pop();

            break;

        }

        case 3: {

            int topElement = peek();

            if (topElement != -1) {

                cout << "Top element is: " << topElement << endl;

            }

            break;

        }

        case 4: {

            cout << (isEmpty() ? "Stack is empty." : "Stack is not empty.") << endl;

            break;

        }

        case 5: {

            cout << (isFull() ? "Stack is full." : "Stack is not full.") << endl;

            break;

        }

        case 6: {

            cout << "Stack size: " << size() << endl;

            break;

        }

        default: {

            cout << "Invalid Choice" << endl;

        }

        }

        cout << endl;

        char choice;

        cout << "Do you want to continue (y/n)?" << endl;

        cin >> choice;

        if (choice == 'y' || choice == 'Y') {

            conti = true;

        } else {

            conti = false;

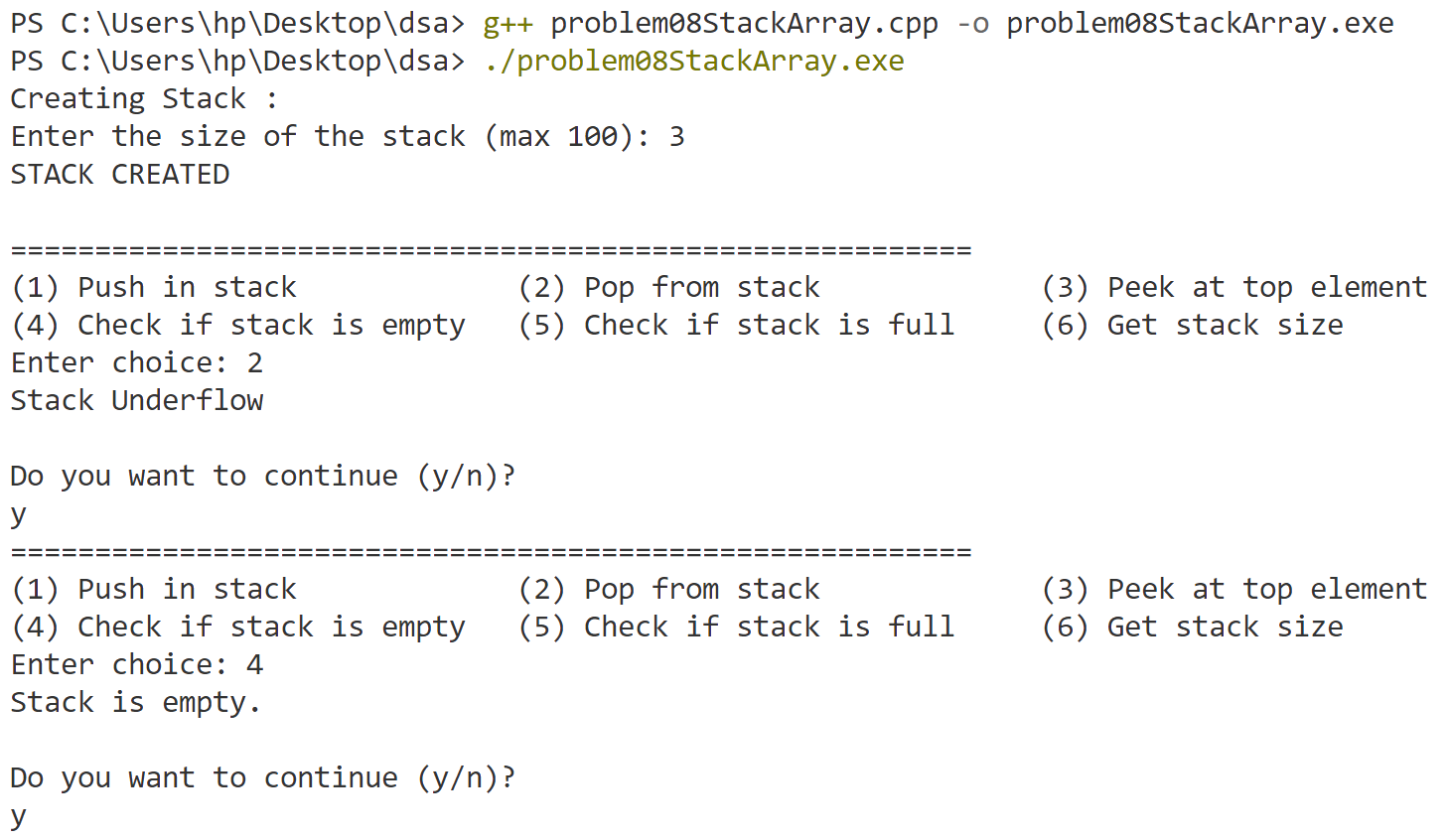
        }

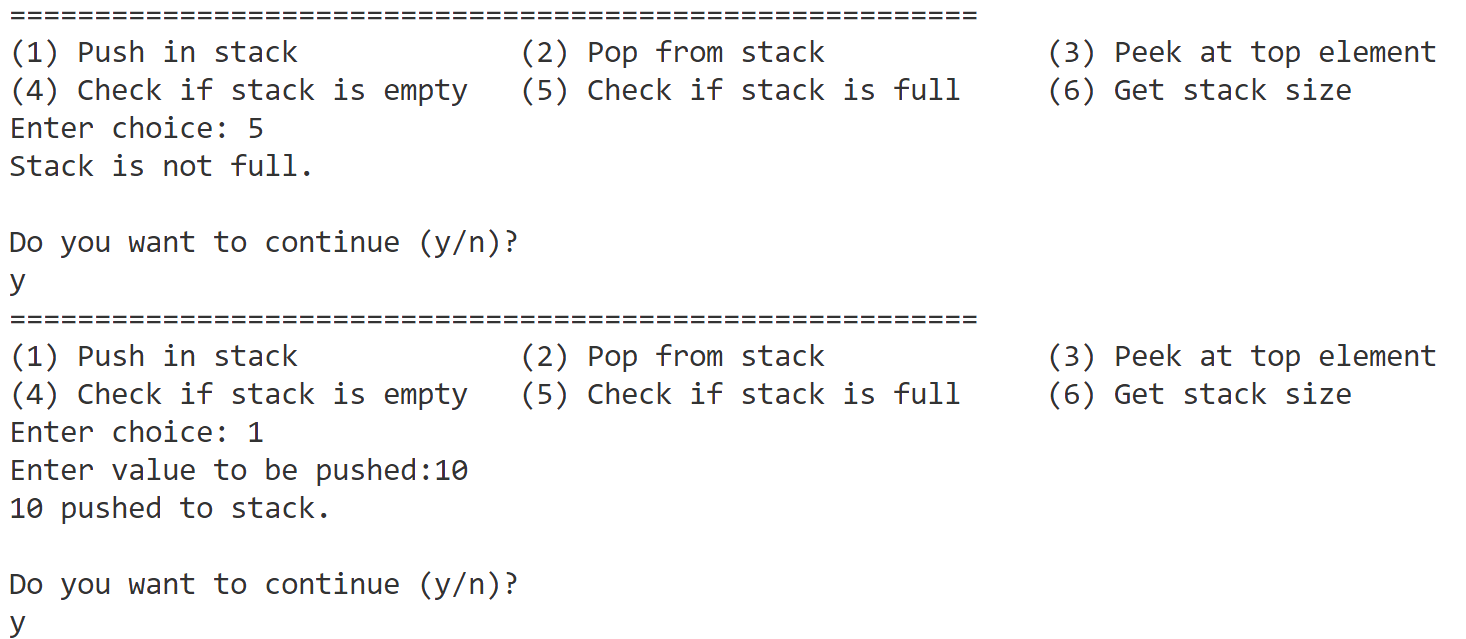
    }

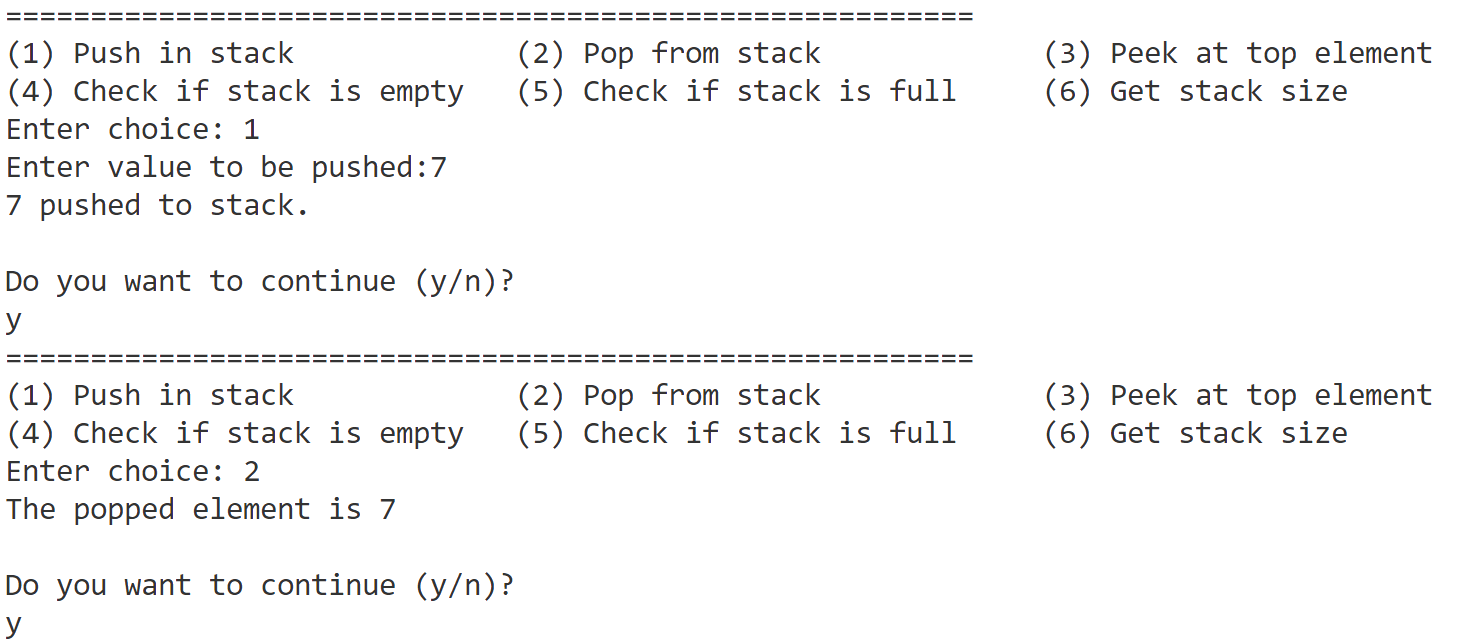
    return 0;

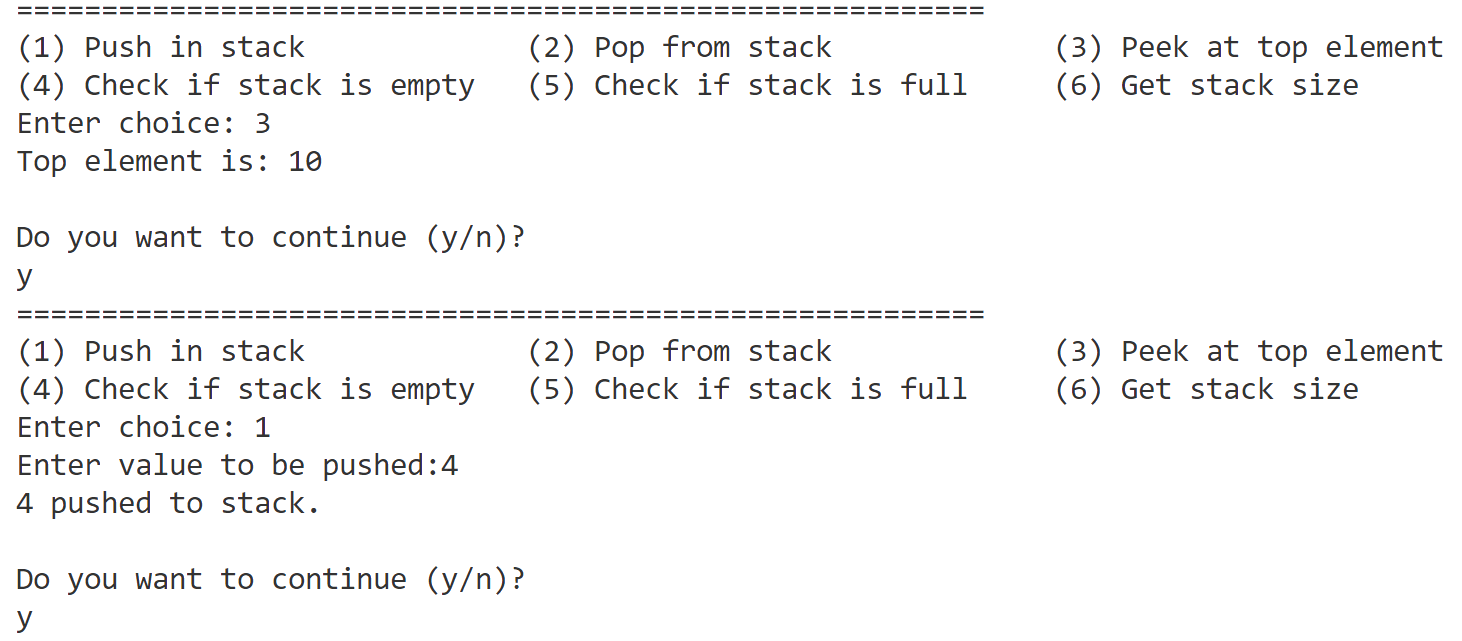
}

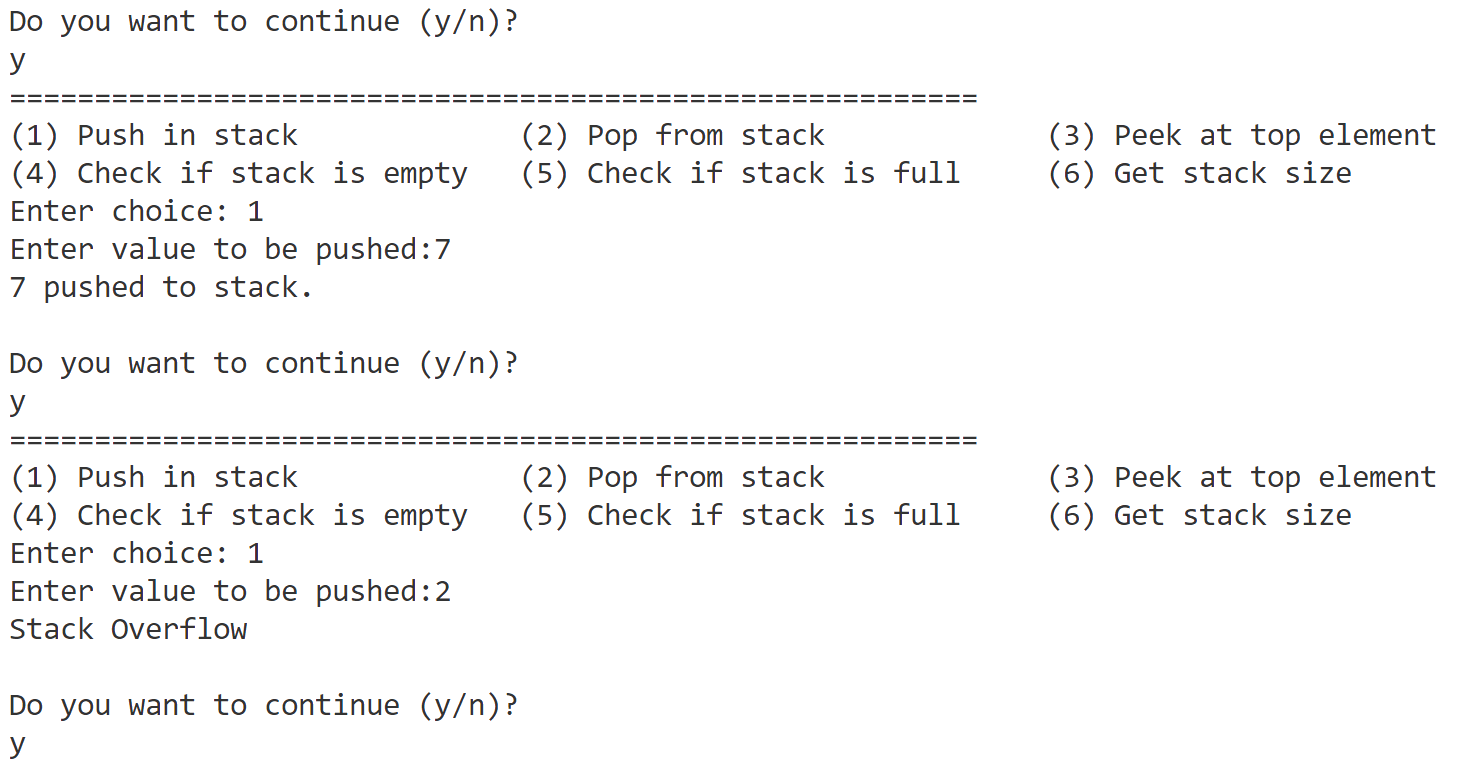
**Output**

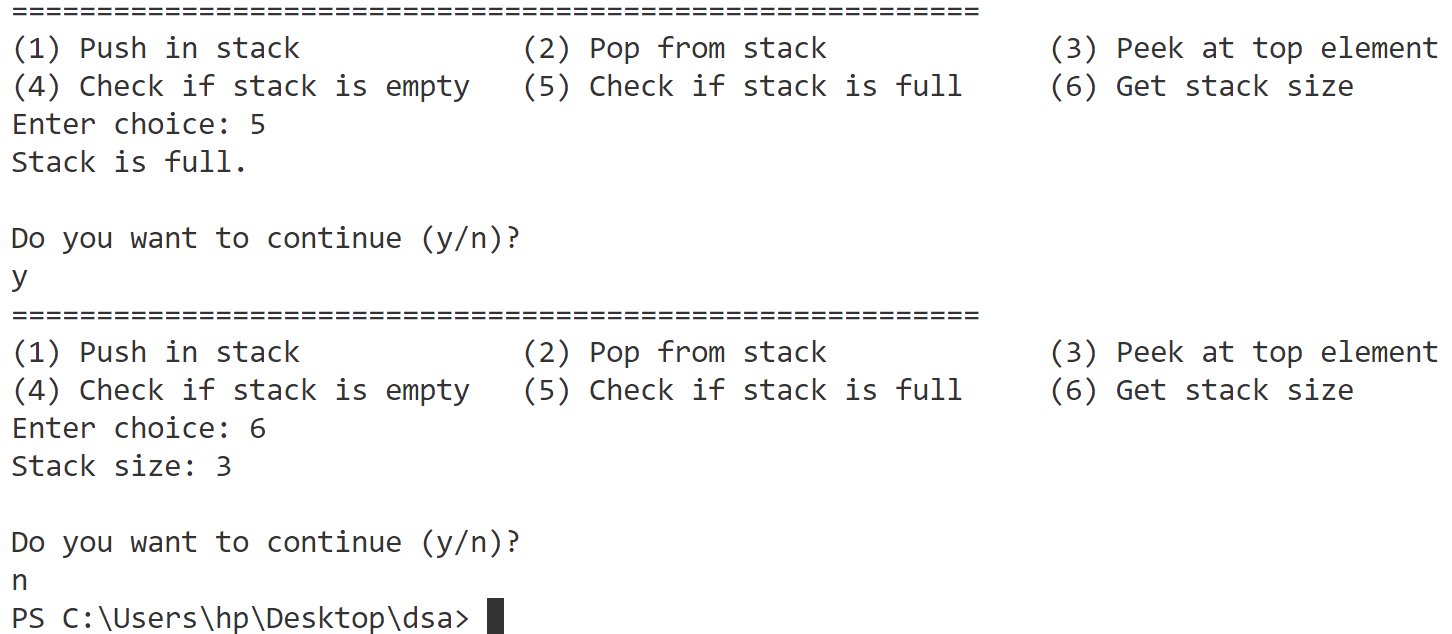












1. **Implement the conversion of infix notation to postfix notation and evaluation of postfix notation using stacks.**

#include <bits/stdc++.h>

using namespace std;

const int MAX\_SIZE = 100;

char Stack[MAX\_SIZE];

int top = -1;

void push(char val) {

    if (top >= MAX\_SIZE - 1) {

        cout << "Stack overflow\n";

    } else {

        top++;

        Stack[top] = val;

    }

}

void pop() {

    if (top <= -1) {

        cout << "Stack underflow\n";

    } else {

        top--;

    }

}

char peek() {

    if (top <= -1) {

        return '\0';

    } else {

        return Stack[top];

    }

}

int prec(char c) {

    if (c == '^')

        return 3;

    else if (c == '/' || c == '\*')

        return 2;

    else if (c == '+' || c == '-')

        return 1;

    else

        return -1;

}

bool isValidInfix(string s) {

    int cnt = 0;

    for (int i = 0; i < s.length(); i++) {

        if (!((s[i] >= 'A' && s[i] <= 'Z') || (s[i] >= 'a' && s[i] <= 'z') ||

                (s[i] >= '0' && s[i] <= '9') || s[i] == '+' || s[i] == '-' ||

                s[i] == '\*' || s[i] == '/' || s[i] == '^' || s[i] == '(' || s[i] == ')')) {

            cout << "Invalid operator or operand: " << s[i] << endl;

            return false;

        } else if (s[i] == '(') {

            cnt++;

        } else if (s[i] == ')') {

            cnt--;

            if (cnt < 0) {

                cout << "Mismatched parentheses" << endl;

                return false;

            }

        }

    }

    if (cnt != 0) {

        cout << "Mismatched parentheses" << endl;

        return false;

    }

    return true;

}

string infixToPostfix(string s) {

    string result;

    for (int i = 0; i < s.length(); i++) {

        char c = s[i];

        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')) {

            result += c;

        } else if (c == '(') {

            push(c);

        } else if (c == ')') {

            while (peek() != '\0' && peek() != '(') {

                result += peek();

                pop();

            }

            pop(); // Remove '(' from the stack

        } else {

            while (top >= 0 && prec(c) <= prec(peek())) {

                result += peek();

                pop();

            }

            push(c);

        }

    }

    while (top >= 0) {

        result += peek();

        pop();

    }

    return result;

}

int evaluatePostfix(const string postfix) {

    for (char c: postfix) {

        if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {

            cout << "cannot evaluate postfix having characters" << endl;

            return INT\_MIN;

        }

        if (c == ' ')

            continue;

        if (c >= '0' && c <= '9') {

            push(c - '0'); // Convert char to int and push

        } else {

            int operand2 = peek();

            pop();

            int operand1 = peek();

            pop();

            int result;

            switch (c) {

            case '+':

                result = operand1 + operand2;

                break;

            case '-':

                result = operand1 - operand2;

                break;

            case '\*':

                result = operand1 \* operand2;

                break;

            case '/':

                result = operand1 / operand2;

                break;

            case '^':

                result = pow(operand1, operand2);

                break;

            }

            push(result); // Push the result back onto the evaluation stack

        }

    }

    return peek(); // The final result is the top of the evaluation stack

}

int main() {

    string expression;

    cout << "Enter infix expression: ";

    cin >> expression;

    if (!isValidInfix(expression)) {

        return 0;

    }

    string postfixResult = infixToPostfix(expression);

    cout << "Postfix expression: " << postfixResult << endl;

    char evaluate;

    cout << "Want to evaluate the postfix expression? (y/n): ";

    cin >> evaluate;

    if (evaluate == 'y' || evaluate == 'Y') {

        top = -1;

        int result = evaluatePostfix(postfixResult);

        if (result != INT\_MIN) {

            cout << "Evaluation result: " << result << endl;

        }

    }

    return 0;

}

**Output**



1. **Implement operations (enqueue, dequeue) on a queue using arrays. Check the status of the queue whether it is empty or full.**

#include <iostream>

using namespace std;

int queue[100];

int maxSize, front, rear;

void createQueue() {

    cout << "Enter the size of the Queue (max 100): ";

    cin >> maxSize;

    if (maxSize > 100 || maxSize <= 0) {

        cout << "Invalid size. Setting queue size to 100." << endl;

        maxSize = 100;

    }

    front = rear = -1;

}

void Enqueue(int val) {

    if (rear >= maxSize - 1) // Check for queue overflow

    {

        cout << "Queue Overflow" << endl;

    } else {

        if (front == -1) // First element being added

        {

            front = 0;

        }

        rear++;

        queue[rear] = val;

        cout << val << " added to queue." << endl;

    }

}

void Dequeue() {

    if (front == -1) {

        cout << "Queue Underflow" << endl;

    } else {

        cout << "Dequeued element is " << queue[front] << endl;

        front++;

        // Reset queue if it becomes empty

        if (front > rear) {

            front = rear = -1;

        }

    }

}

int getFront() {

    if (front == -1) {

        cout << "Queue is empty." << endl;

        return -1;

    } else {

        return queue[front];

    }

}

bool isEmpty() {

    return (front == -1); // Queue is empty if front is -1

}

bool isFull() {

    return (rear >= maxSize - 1); // Queue is full

}

int CurrentSize() {

    if (front == -1) return 0; // If queue is empty, size is 0

    return rear - front + 1; // Calculate current size

}

int main() {

    int ch, val;

    cout << "Creating Queue :" << endl;

    createQueue();

    cout << "QUEUE CREATED" << endl;

    cout << endl;

    bool conti = true;

    while (conti) {

        cout << "========================================================" << endl;

        cout << "(1) Enqueue in Queue             ";

        cout << "(2) Dequeue from Queue           ";

        cout << "(3) Peek at front element        " << endl;

        cout << "(4) Check if Queue is empty      ";

        cout << "(5) Check if Queue is full       ";

        cout << "(6) Get queue size               " << endl;

        cout << "Enter choice: ";

        cin >> ch;

        switch (ch) {

        case 1:

            cout << "Enter value to be enqueued: ";

            cin >> val;

            Enqueue(val);

            break;

        case 2:

            Dequeue();

            break;

        case 3:

            cout << "Front element is: " << getFront() << endl;

            break;

        case 4:

            cout << (isEmpty() ? "Queue is empty." : "Queue is not empty.") << endl;

            break;

        case 5:

            cout << (isFull() ? "Queue is full." : "Queue is not full.") << endl;

            break;

        case 6:

            cout << "Queue size: " << CurrentSize() << endl;

            break;

        default:

            cout << "Invalid Choice" << endl;

        }

        cout << endl;

        char choice;

        cout << "Do you want to continue (y/n)? ";

        cin >> choice;

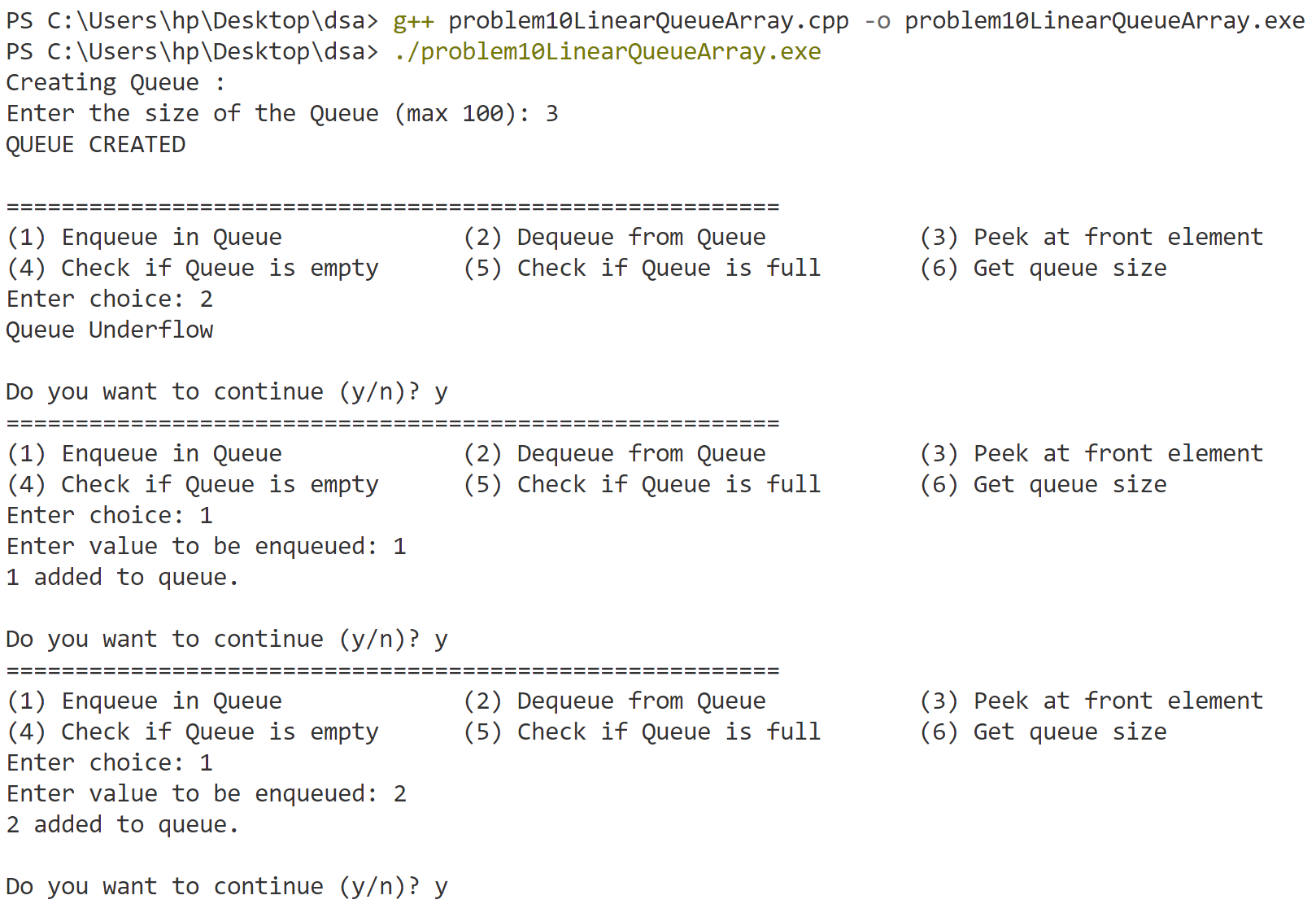
        conti = (choice == 'y' || choice == 'Y');

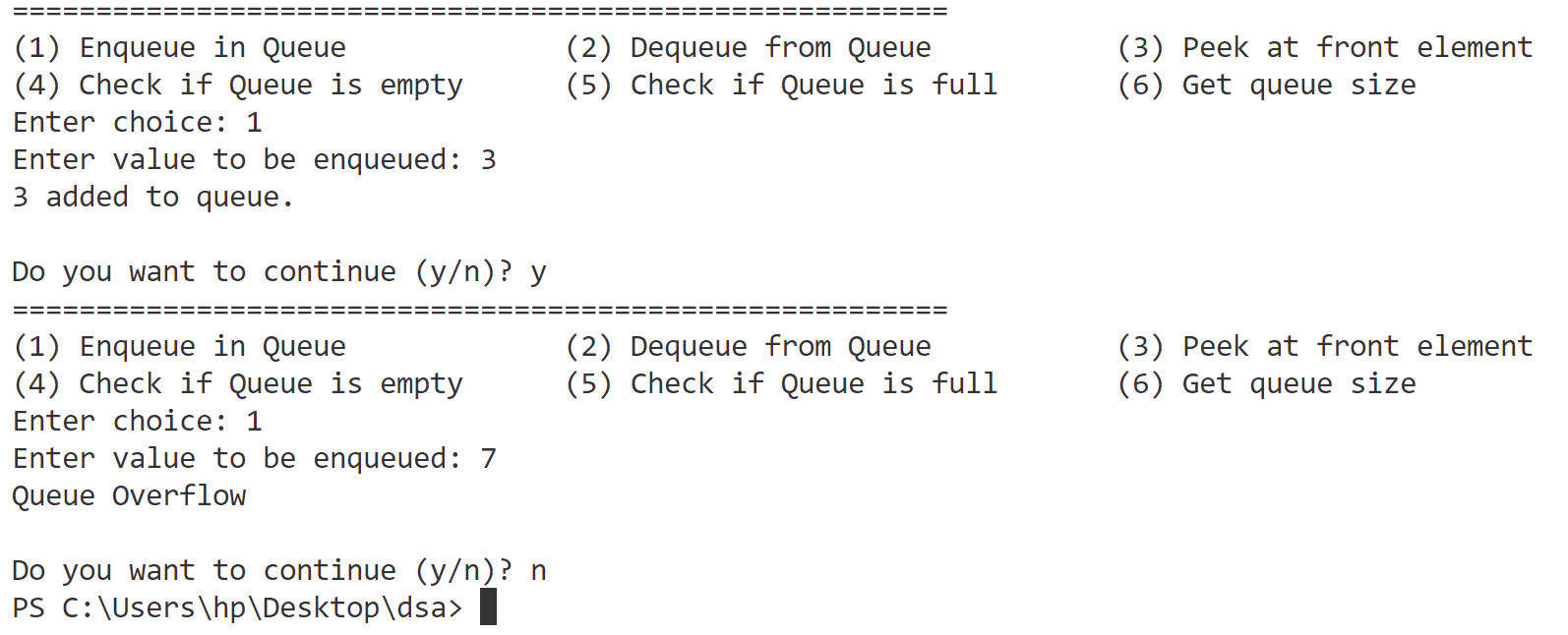
    }

    return 0;

}

**Output**





1. **Implement circular queue using arrays and linked list.**

#include <iostream>

using namespace std;

struct Node {

    int data;

    Node \* next;

};

Node \* front = nullptr;

Node \* rear = nullptr;

Node \* createNode(int val) {

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

    newNode -> data = val;

    newNode -> next = nullptr;

    return newNode;

}

void EnqueueLL(int val) {

    Node \* newNode = createNode(val);

    if (front == nullptr) {

        front = rear = newNode;

        rear -> next = front;

    } else {

        rear -> next = newNode;

        rear = newNode;

        rear -> next = front;

    }

    cout << val << " added to queue." << endl;

}

void DequeueLL() {

    if (front == nullptr) {

        cout << "Queue Underflow" << endl;

        return;

    }

    Node \* temp = front;

    cout << "Dequeued element is " << temp -> data << endl;

    if (front == rear) {

        front = rear = nullptr;

    } else {

        front = front -> next;

        rear -> next = front;

    }

    free(temp);

}

int getFrontLL() {

    if (front == nullptr) {

        cout << "Queue is empty." << endl;

        return -1;

    }

    return front -> data;

}

bool isEmptyLL() {

    return (front == nullptr);

}

int CurrentSizeLL() {

    if (isEmptyLL())

        return 0;

    int size = 0;

    Node \* temp = front;

    do {

        size++;

        temp = temp -> next;

    } while (temp != front);

    return size;

}

int queue[100];

int maxSize, frontA, rearA;

void createQueue() {

    cout << "Enter the size of the Queue (max 100): ";

    cin >> maxSize;

    if (maxSize > 100 || maxSize <= 0) {

        cout << "Invalid size. Setting queue size to 100." << endl;

        maxSize = 100;

    }

    frontA = rearA = -1;

}

void Enqueue(int val) {

    if ((rearA + 1) % maxSize == frontA) {

        cout << "Queue Overflow" << endl;

    } else {

        if (frontA == -1) { // If the queue is empty

            frontA = 0;

        }

        rearA = (rearA + 1) % maxSize;

        queue[rearA] = val;

        cout << val << " added to queue." << endl;

    }

}

void Dequeue() {

    if (frontA == -1) {

        cout << "Queue Underflow" << endl;

    } else {

        cout << "Dequeued element is " << queue[frontA] << endl;

        if (frontA == rearA) { // If the queue becomes empty

            frontA = rearA = -1;

        } else {

            frontA = (frontA + 1) % maxSize;

        }

    }

}

int getFront() {

    if (frontA == -1) {

        cout << "Queue is empty." << endl;

        return -1;

    } else {

        return queue[frontA];

    }

}

bool isEmpty() {

    return (frontA == -1);

}

bool isFull() {

    return ((rearA + 1) % maxSize == frontA);

}

int CurrentSize() {

    if (isEmpty())

        return 0;

    return (rearA - frontA + maxSize) % maxSize + 1;

}

int main() {

    int k;

    cout << "How to make queue enter 1 for LL and 2 for Array" << endl;

    cin >> k;

    if (k == 1) {

        int ch, val;

        bool conti = true;

        while (conti) {

            cout << "=======================================================" << endl;

            cout << "(1) Enqueue in Queue             ";

            cout << "(2) Dequeue from Queue           ";

            cout << "(3) Peek at front element        " << endl;

            cout << "(4) Check if Queue is empty      ";

            cout << "(5) Get queue size               " << endl;

            cout << "Enter choice: ";

            cin >> ch;

            switch (ch) {

            case 1:

                cout << "Enter value to be enqueued: ";

                cin >> val;

                EnqueueLL(val);

                break;

            case 2:

                DequeueLL();

                break;

            case 3:

                cout << "Front element is: " << getFrontLL() << endl;

                break;

            case 4:

                cout << (isEmptyLL() ? "Queue is empty." : "Queue is not empty.") << endl;

                break;

            case 5:

                cout << "Queue size: " << CurrentSizeLL() << endl;

                break;

            default:

                cout << "Invalid Choice" << endl;

            }

            cout << endl;

            char choice;

            cout << "Do you want to continue (y/n)? ";

            cin >> choice;

            conti = (choice == 'y' || choice == 'Y');

        }

    } else if (k == 2) {

        int ch, val;

        cout << "Creating Queue :" << endl;

        createQueue();

        cout << "QUEUE CREATED" << endl;

        cout << endl;

        bool conti = true;

        while (conti) {

            cout << "============================================" << endl;

            cout << "(1) Enqueue in Queue             ";

            cout << "(2) Dequeue from Queue           ";

            cout << "(3) Peek at front element        " << endl;

            cout << "(4) Check if Queue is empty      ";

            cout << "(5) Check if Queue is full       ";

            cout << "(6) Get queue size               " << endl;

            cout << "Enter choice: ";

            cin >> ch;

            switch (ch) {

            case 1:

                cout << "Enter value to be enqueued: ";

                cin >> val;

                Enqueue(val);

                break;

            case 2:

                Dequeue();

                break;

            case 3:

                cout << "Front element is: " << getFront() << endl;

                break;

            case 4:

                cout << (isEmpty() ? "Queue is empty." : "Queue is not empty.") << endl;

                break;

            case 5:

                cout << (isFull() ? "Queue is full." : "Queue is not full.") << endl;

                break;

            case 6:

                cout << "Queue size: " << CurrentSize() << endl;

                break;

            default:

                cout << "Invalid Choice" << endl;

            }

            cout << endl;

            char choice;

            cout << "Do you want to continue (y/n)? ";

            cin >> choice;

            conti = (choice == 'y' || choice == 'Y');

        }

    } else {

        cout << "invalid exiting program" << endl;

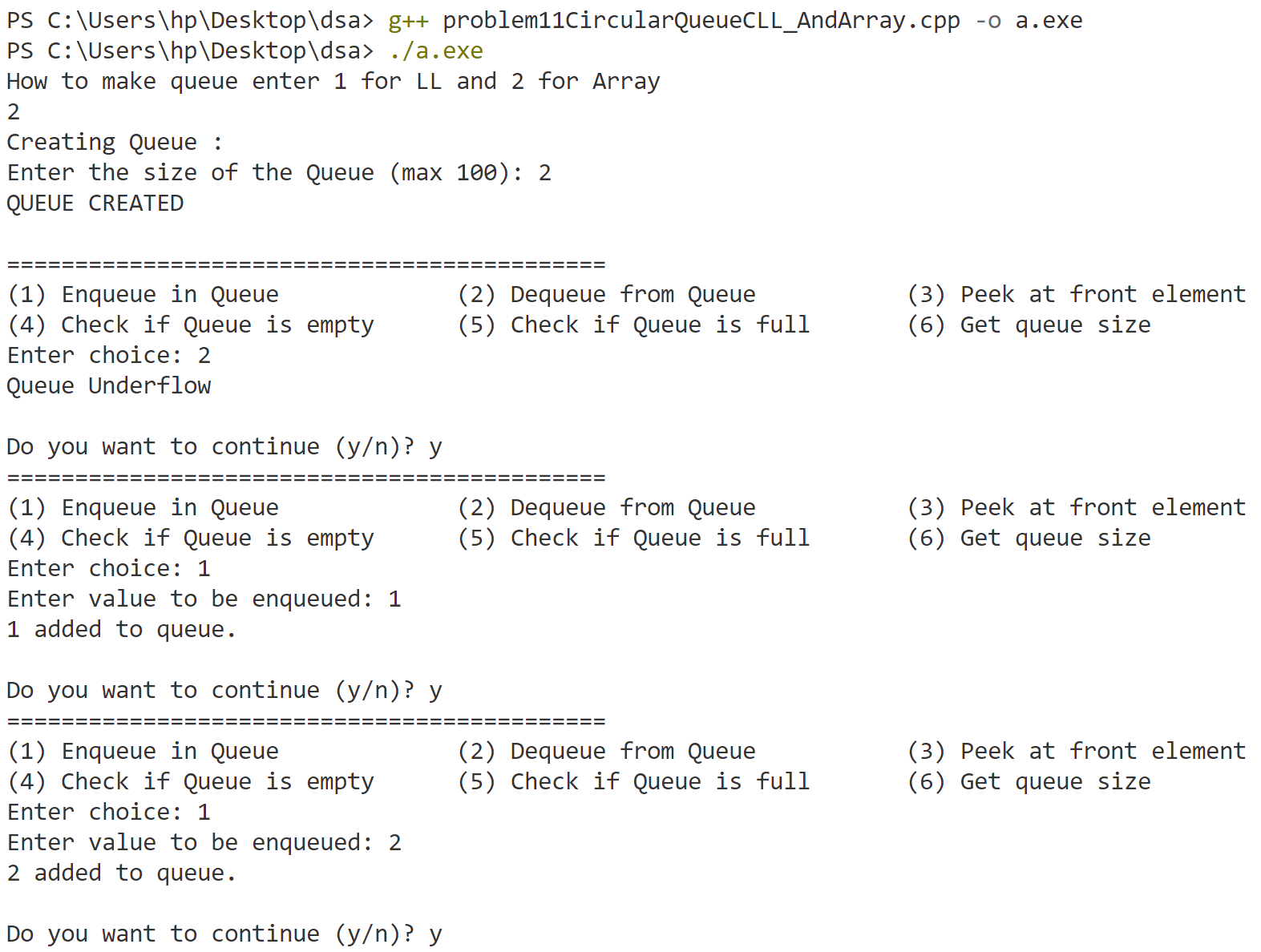
    }

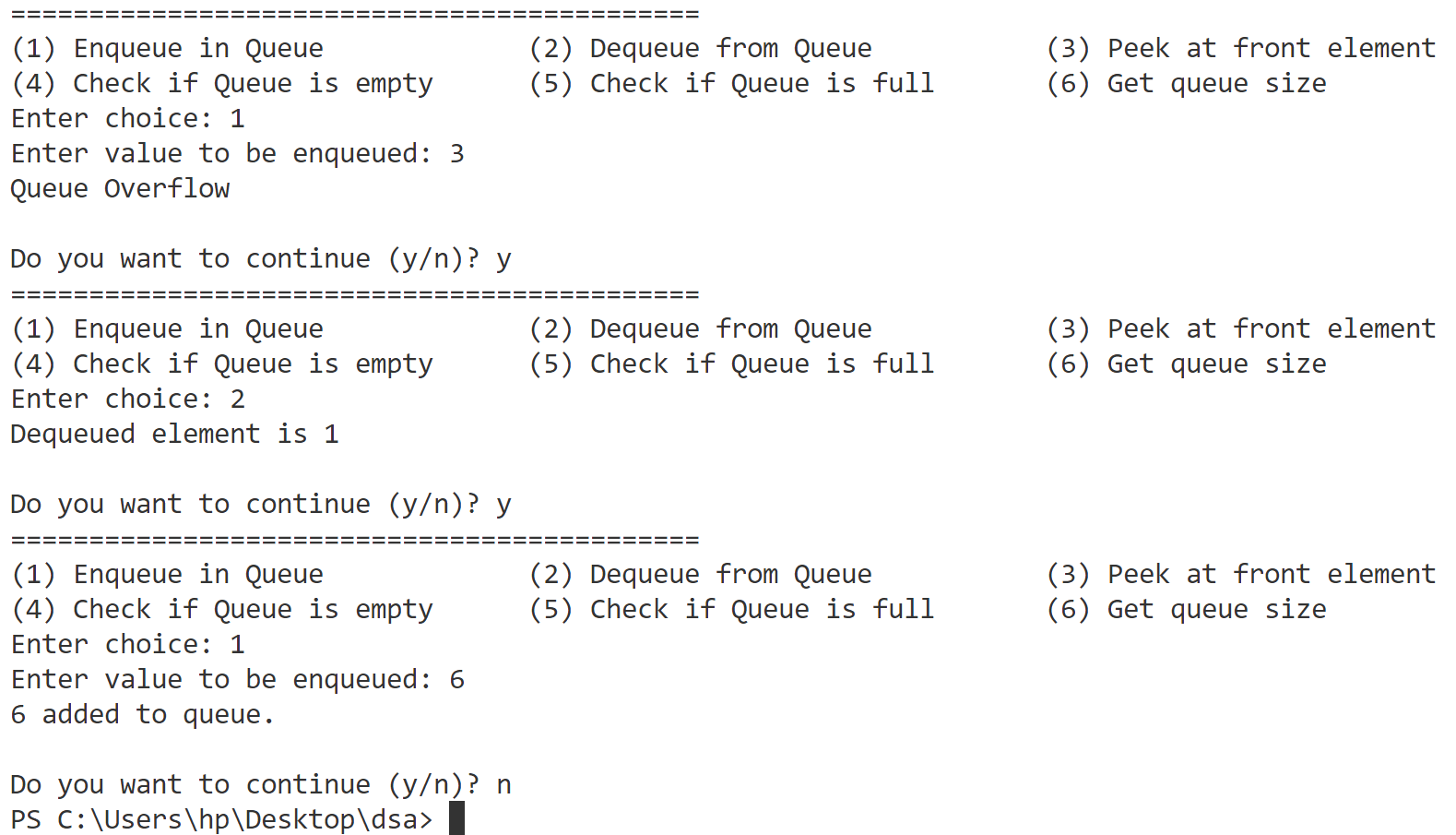
    return 0;

}

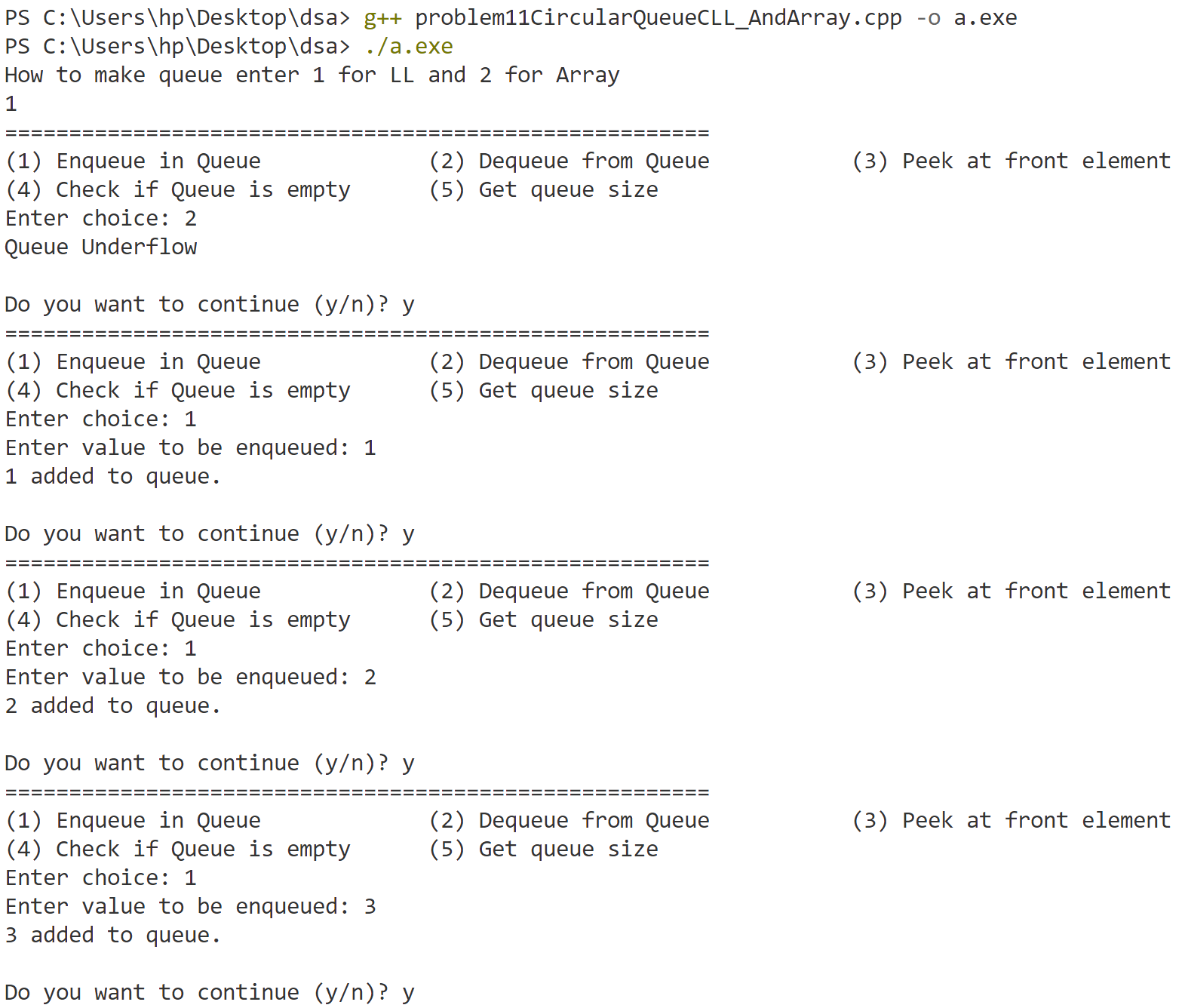
**Output**

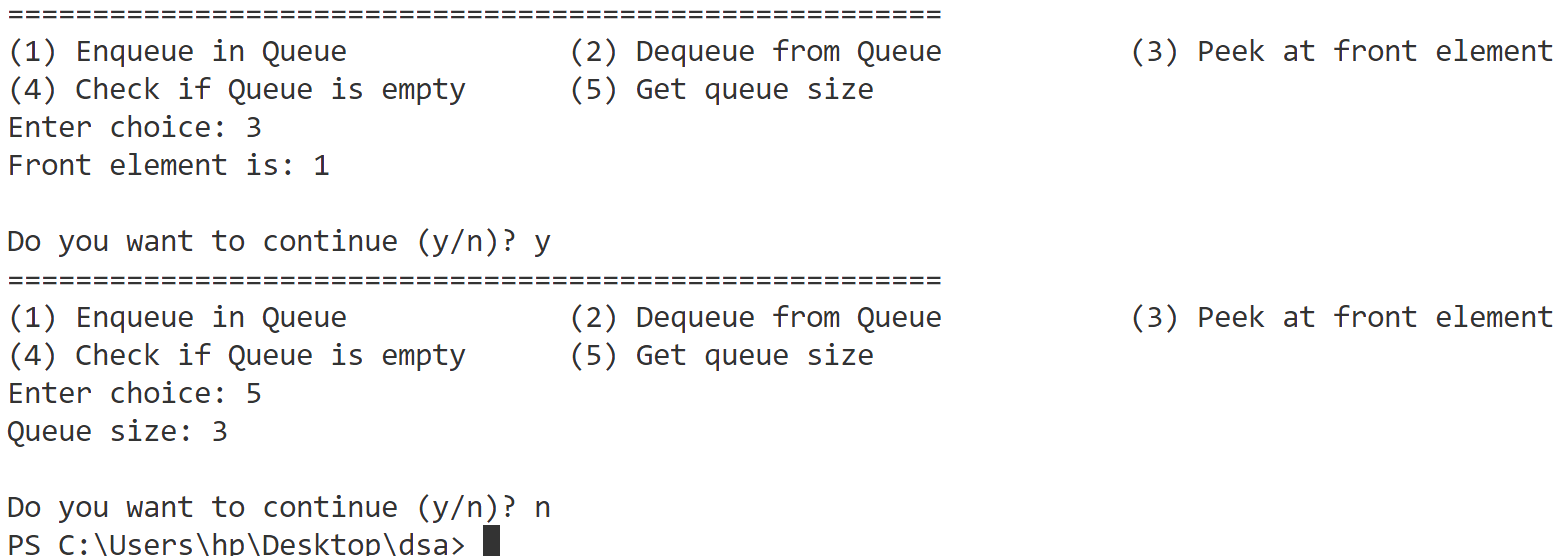
**Using Array**





**Using Linked list**





1. **Implement stacks and queues using linked list.**

#include <bits/stdc++.h>

using namespace std;

struct Node {

    int data;

    Node \* next;

};

Node \* head = nullptr;

int getSize() {

    int count = 0;

    Node \* temp = head;

    while (temp != nullptr) {

        count++;

        temp = temp -> next;

    }

    return count;

}

void push(int val) {

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

    newNode -> data = val;

    newNode -> next = head;

    head = newNode;

    cout << val << " pushed to stack." << endl;

}

void pop() {

    if (head == nullptr) {

        cout << "Stack Underflow" << endl;

        return;

    }

    cout << "The popped element is " << head -> data << endl;

    Node \* temp = head;

    head = head -> next;

    free(temp);

}

int peek() {

    if (head == nullptr) {

        cout << "Stack is empty." << endl;

        return -1;

    }

    return head -> data;

}

bool isEmpty() {

    return head == nullptr;

}

Node \* front = nullptr;

Node \* rear = nullptr;

Node \* createNode(int val) {

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

    newNode -> data = val;

    newNode -> next = nullptr;

    return newNode;

}

void Enqueue(int val) {

    Node \* newNode = createNode(val);

    if (rear == nullptr) { // Queue is empty

        front = rear = newNode;

    } else {

        rear -> next = newNode;

        rear = newNode;

    }

    cout << val << " added to queue." << endl;

}

void Dequeue() {

    if (front == nullptr) {

        cout << "Queue Underflow" << endl;

        return;

    }

    Node \* temp = front;

    cout << "Dequeued element is " << temp -> data << endl;

    front = front -> next;

    if (front == nullptr) { // If queue is now empty

        rear = nullptr;

    }

    free(temp); // Free the dequeued node

}

int getFront() {

    if (front == nullptr) {

        cout << "Queue is empty." << endl;

        return -1;

    }

    return front -> data;

}

bool isEmptyQ() {

    return (front == nullptr);

}

int CurrentSize() {

    int size = 0;

    Node \* temp = front;

    while (temp) {

        size++;

        temp = temp -> next;

    }

    return size;

}

int main() {

    int k;

    cout << "How to make queue enter " << endl << "     1 for Stack using LL " << endl << "     2 for queue using LL" << endl;

    cin >> k;

    if (k == 1) {

        int ch, val;

        cout << "STACK CREATED" << endl;

        cout << endl;

        bool conti = true;

        while (conti) {

            cout << "========================================================" << endl;

            cout << "(1) Push in stack             ";

            cout << "(2) Pop from stack             ";

            cout << "(3) Peek at top element" << endl;

            cout << "(4) Check if stack is empty   ";

            cout << "(5) Get stack size" << endl;

            cout << "Enter choice: ";

            cin >> ch;

            switch (ch) {

            case 1:

                cout << "Enter value to be pushed: ";

                cin >> val;

                push(val);

                break;

            case 2:

                pop();

                break;

            case 3: {

                int topElement = peek();

                if (topElement != -1) {

                    cout << "Top element is: " << topElement << endl;

                }

                break;

            }

            case 4:

                cout << (isEmpty() ? "Stack is empty." : "Stack is not empty.") << endl;

                break;

            case 5:

                cout << "Stack size: " << getSize() << endl;

                break;

            default:

                cout << "Invalid Choice" << endl;

                break;

            }

            cout << endl;

            char choice;

            cout << "Do you want to continue (y/n)? ";

            cin >> choice;

            conti = (choice == 'y' || choice == 'Y');

        }

    } else if (k == 2) {

        int ch, val;

        bool conti = true;

        while (conti) {

            cout << "=======================================================" << endl;

            cout << "(1) Enqueue in Queue             ";

            cout << "(2) Dequeue from Queue           ";

            cout << "(3) Peek at front element        " << endl;

            cout << "(4) Check if Queue is empty      ";

            cout << "(5) Get queue size               " << endl;

            cout << "Enter choice: ";

            cin >> ch;

            switch (ch) {

            case 1:

                cout << "Enter value to be enqueued: ";

                cin >> val;

                Enqueue(val);

                break;

            case 2:

                Dequeue();

                break;

            case 3:

                cout << "Front element is: " << getFront() << endl;

                break;

            case 4:

                cout << (isEmptyQ() ? "Queue is empty." : "Queue is not empty.") << endl;

                break;

            case 5:

                cout << "Queue size: " << CurrentSize() << endl;

                break;

            default:

                cout << "Invalid Choice" << endl;

            }

            cout << endl;

            char choice;

            cout << "Do you want to continue (y/n)? ";

            cin >> choice;

            conti = (choice == 'y' || choice == 'Y');

        }

    } else {

        cout << "invalid exiting program" << endl;

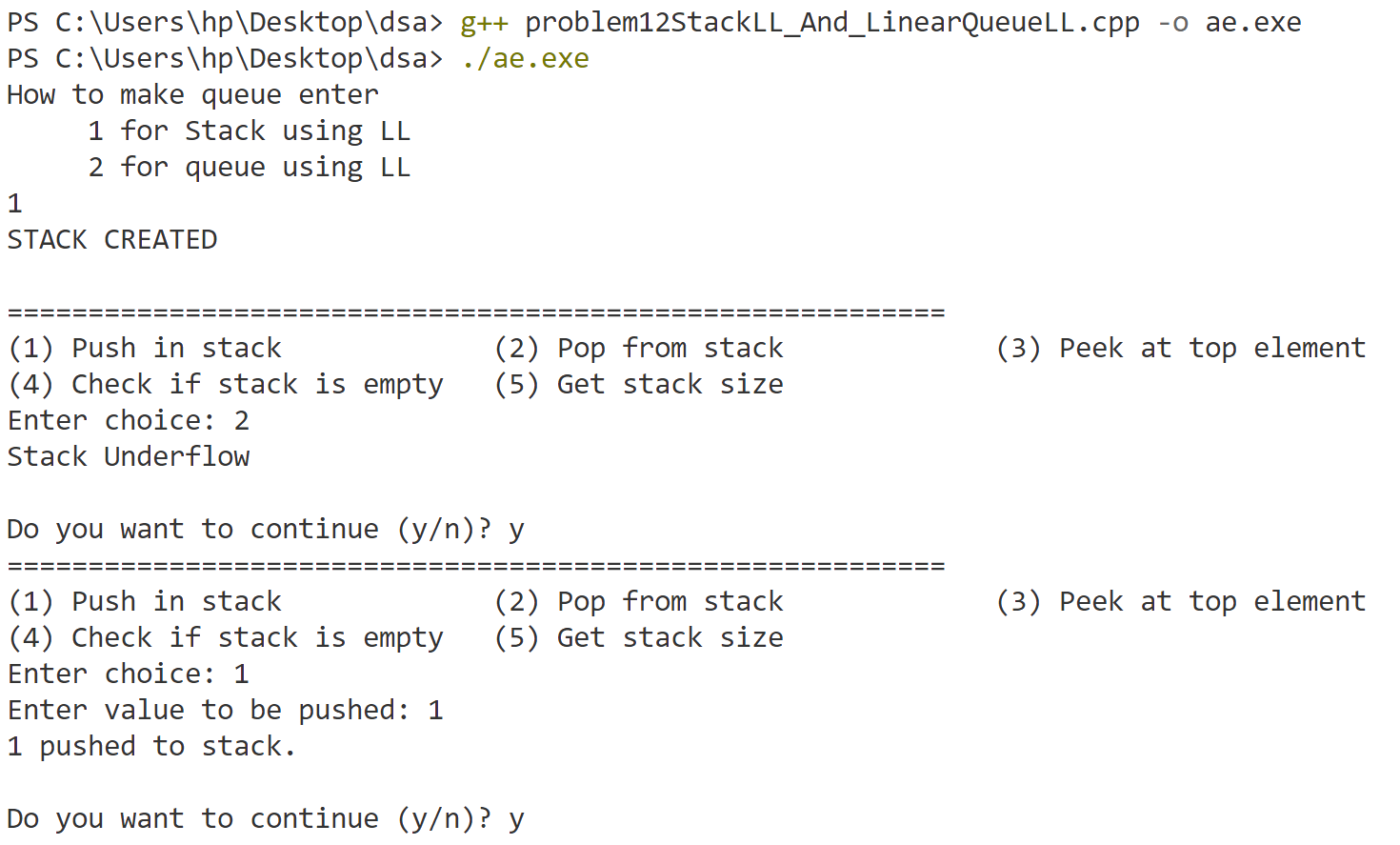
    }

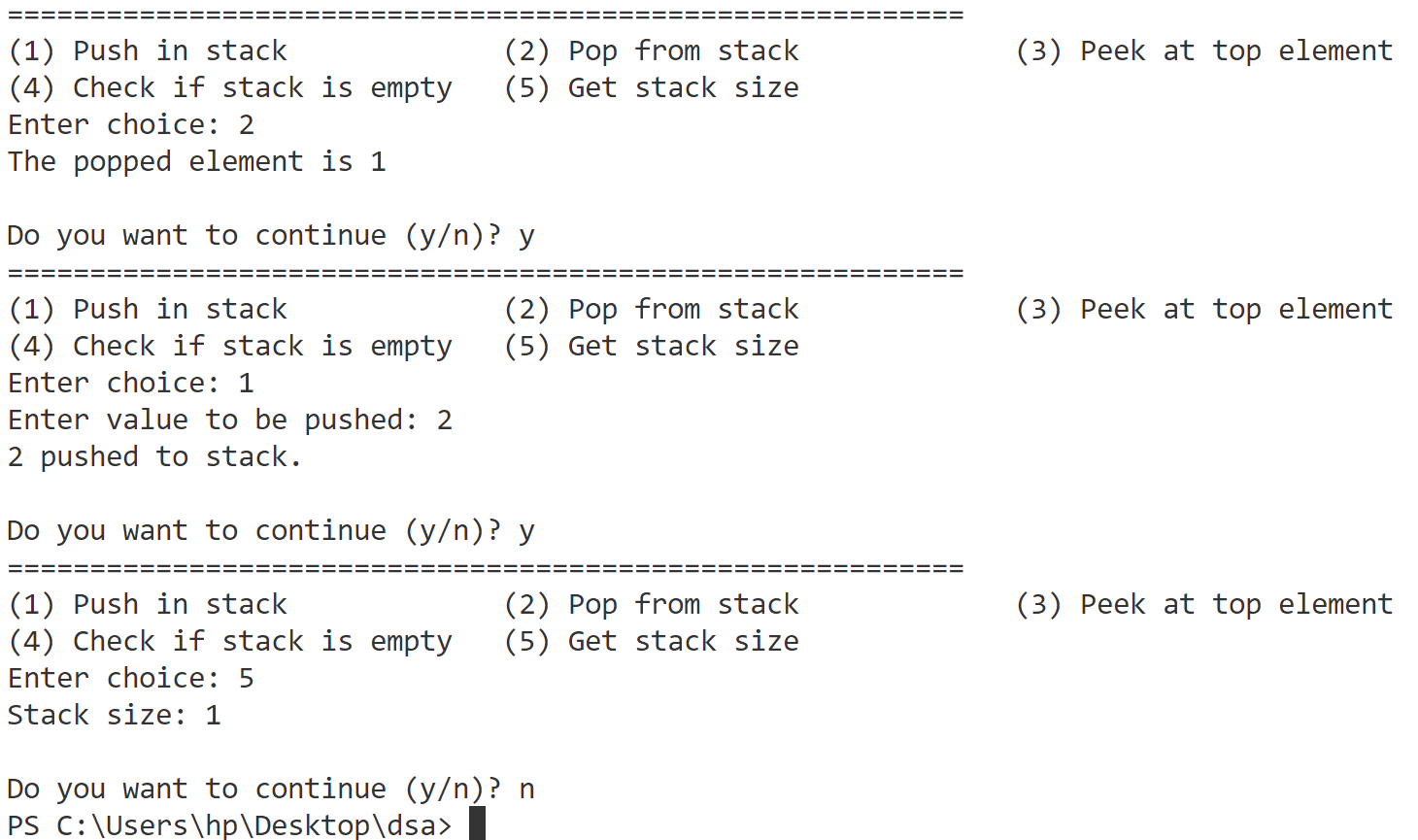
    return 0;

}

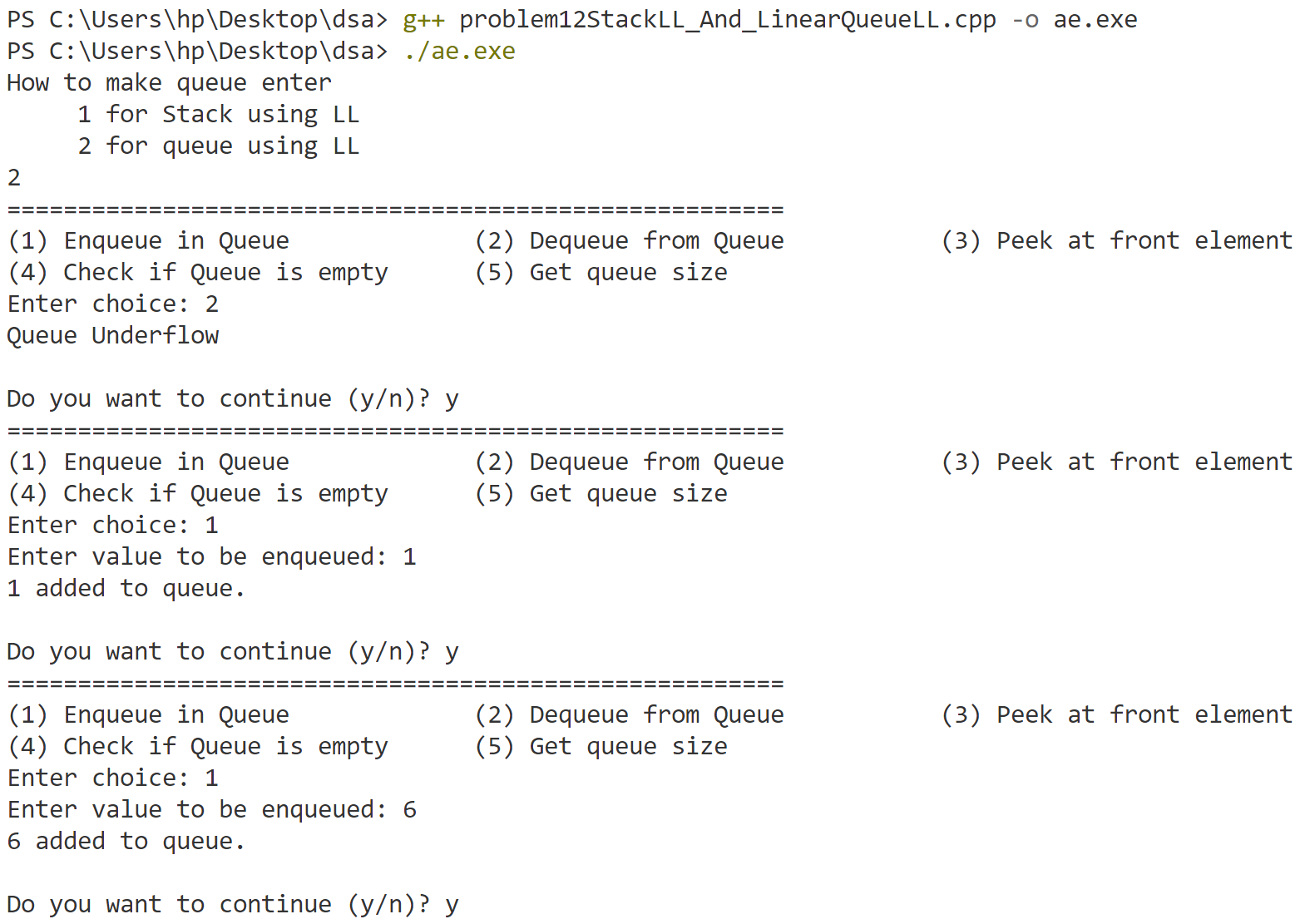
**Output**

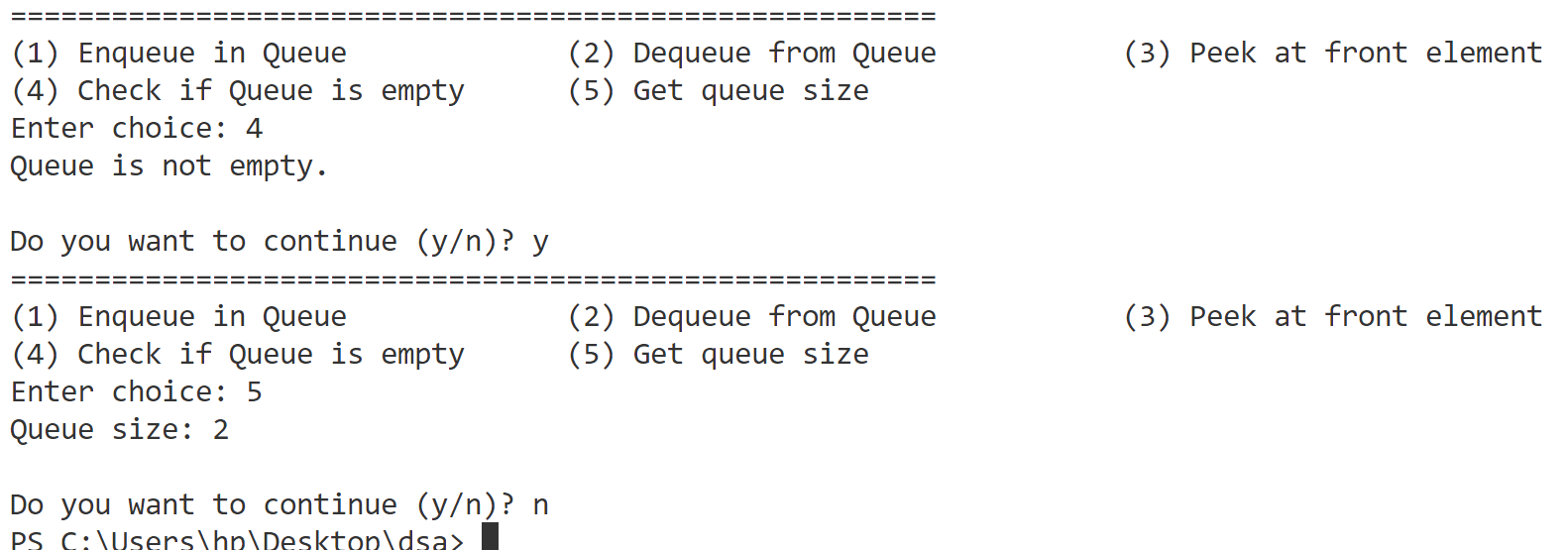
**Stack**





**Queue**



****

1. **Implement operations on Binary Search Tree (Insertion, Deletion, Search, Traversals (using recursion)- Inorder, Preorder, Postorder).**

#include <iostream>

using namespace std;

struct Node {

    int data;

    Node\* left;

    Node\* right;

};

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

    if (root == nullptr) {

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

        newNode->data = data;

        newNode->left = newNode->right = nullptr;

        return newNode;

    }

    if (data < root->data) {

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

    } else if (data > root->data) {

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

    } else {

        cout << "Duplicate value " << data << " not inserted." << endl;

        return root;

    }

    return root;

}

// Function to find the minimum value node in a right subtree

Node\* minValueNode(Node\* node) {

    Node\* current = node;

    while (current && current->left != nullptr) {

        current = current->left;

    }

    return current;

}

// Function to delete a node from the BST

Node\* deleteNode(Node\* root, int data) {

    if (root == nullptr) return root;

    if (data < root->data) {

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

    } else if (data > root->data) {

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

    } else {

        //0 child

        if (root->left == nullptr && root->right == nullptr) {

            free(root);

            return nullptr;

        }

        //1 child

        if (root->left == nullptr && root->right != nullptr) {//right child

            Node\* temp = root->right;

            free(root);

            return temp;

        }

        //1 child

        if (root->right == nullptr && root->left != nullptr) {//left child

            Node\* temp = root->left;

            free(root);

            return temp;

        }

        //2 child

        if (root->left != nullptr && root->right != nullptr)

        {

            Node\* temp = minValueNode(root->right);//from right subtree

            root->data = temp->data;

            root->right = deleteNode(root->right, temp->data);//delete the min value from subtree

            return root;

        }

    }

}

Node\* search(Node\* root, int data) {

    if (root == nullptr || root->data == data) {

        return root;

    }

    if (data < root->data) {

        return search(root->left, data);

    } else {

        return search(root->right, data);

    }

}

void inorder(Node\* root) {

    if (root != nullptr) {

        inorder(root->left);

        cout << root->data << " ";

        inorder(root->right);

    }

}

// Preorder traversal (Root, Left, Right)

void preorder(Node\* root) {

    if (root != nullptr) {

        cout << root->data << " ";

        preorder(root->left);

        preorder(root->right);

    }

}

// Postorder traversal (Left, Right, Root)

void postorder(Node\* root) {

    if (root != nullptr) {

        postorder(root->left);

        postorder(root->right);

        cout << root->data << " ";

    }

}

int main() {

    Node\* root = nullptr;

    int choice, value, data;

    bool Continue = true;

    cout << "Enter elements with space to insert in tree and enter -1 to stop: ";

    cin >> data;

    while (data != -1) {

        root = insert(root, data);

        cin >> data;

    }

    while (Continue) {

        cout << "\nMenu:\n";

        cout << "1. Insert";

        cout << "              2. Delete";

        cout << "              3. Search\n";

        cout << "4. Inorder Traversal";

        cout << "   5. Preorder Traversal";

        cout << "  6. Postorder Traversal\n";

        cout << "7. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                cout << "Enter value to insert: ";

                cin >> value;

                root = insert(root, value);

                break;

            case 2:

                cout << "Enter value to delete: ";

                cin >> value;

                root = deleteNode(root, value);

                break;

            case 3:

                cout << "Enter value to search: ";

                cin >> value;

                if (search(root, value) != nullptr) {

                    cout << "Value " << value << " found in the tree." << endl;

                } else {

                    cout << "Value " << value << " not found in the tree." << endl;

                }

                break;

            case 4:

                if (root == nullptr) {

                    cout << "Tree is empty." << endl;

                    break;

                }

                cout << "Inorder Traversal: ";

                inorder(root);

                cout << endl;

                break;

            case 5:

                if (root == nullptr) {

                    cout << "Tree is empty." << endl;

                    break;

                }

                cout << "Preorder Traversal: ";

                preorder(root);

                cout << endl;

                break;

            case 6:

                if (root == nullptr) {

                    cout << "Tree is empty." << endl;

                    break;

                }

                cout << "Postorder Traversal: ";

                postorder(root);

                cout << endl;

                break;

            case 7:

                Continue = false;

                break;

            default:

                cout << "Invalid choice. Please try again." << endl;

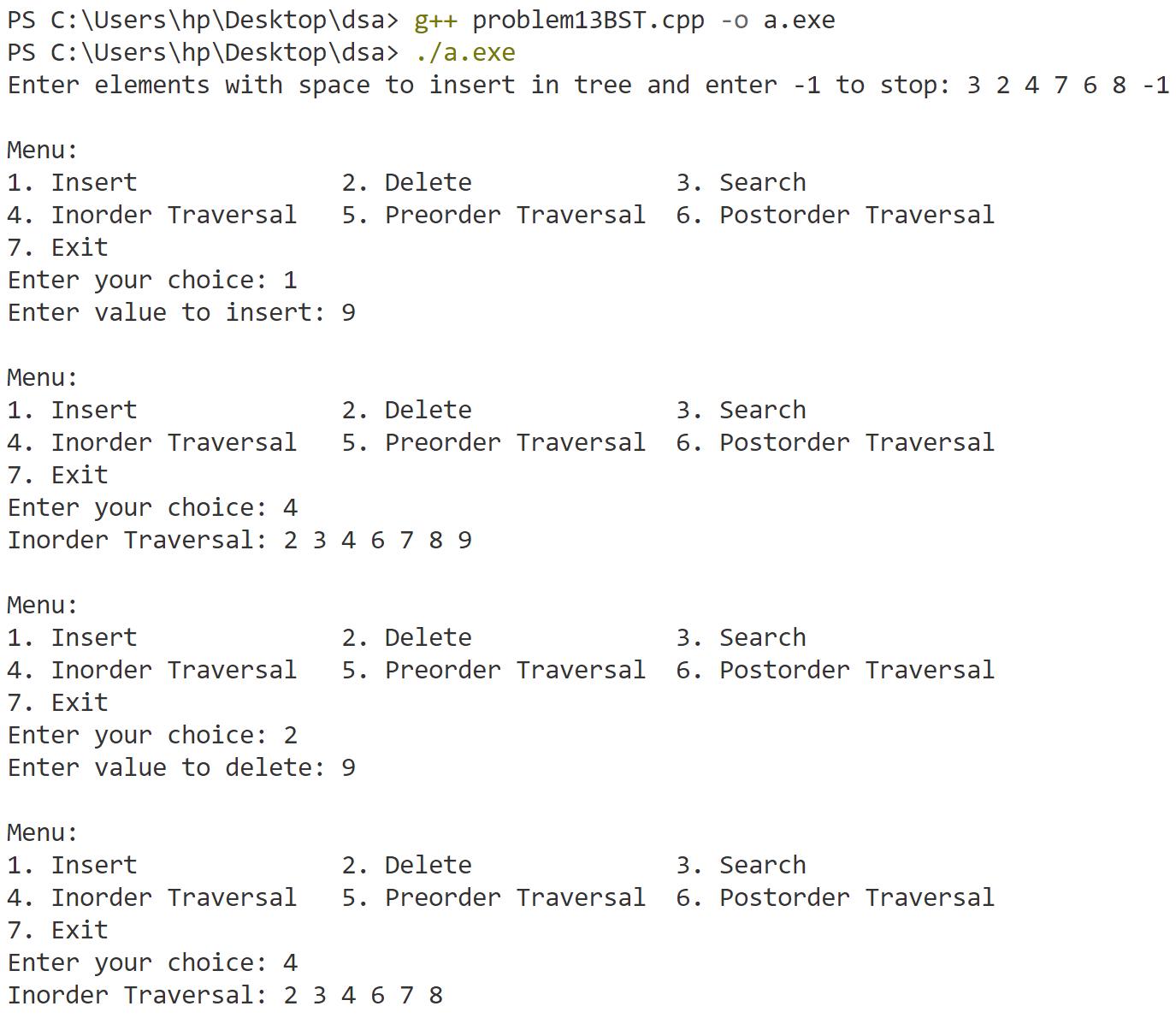
        }

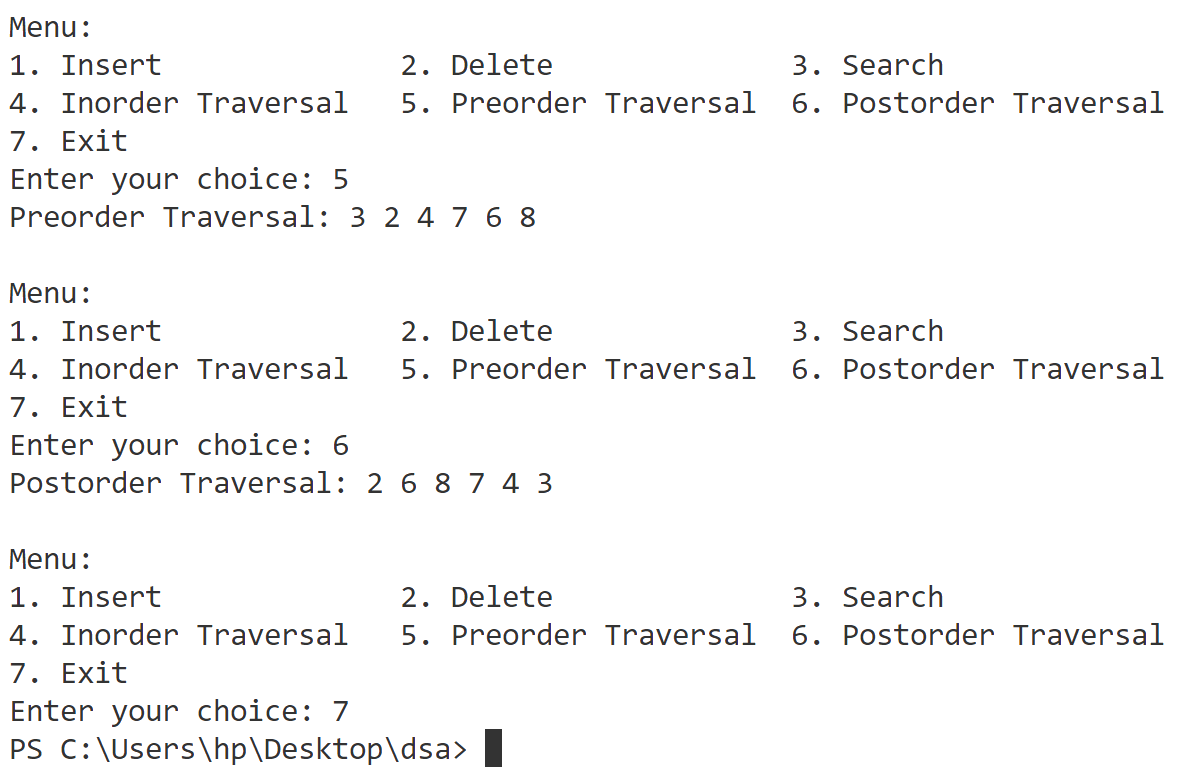
    }

    return 0;

}

**Output**





1. **Implement traversals on Binary Search Tree (using stacks) - Inorder, Preorder, Postorder.**

#include <iostream>

#include <cstdlib> // for malloc and free

using namespace std;

struct Node {

    int data;

    Node\* left;

    Node\* right;

};

// Top -> [SecondNode] -> [FirstNode] -> nullptr

struct StackNode {

    Node\* treeNode;

    StackNode\* next;

};

bool isEmpty(StackNode\* top) {

    return top == nullptr;

}

void push(StackNode\*& top, Node\* node) {

    StackNode\* newStackNode = (StackNode\*)malloc(sizeof(StackNode));

    newStackNode->treeNode = node;

    newStackNode->next = top;

    top = newStackNode;

}

void pop(StackNode\*& top) {

    if (isEmpty(top)) {

        cout << "Stack underflow\n";

        return;

    }

    StackNode\* temp = top;

    top = top->next;

    free(temp);

}

Node\* top(StackNode\* top) {

    if (isEmpty(top)) {

        return nullptr;

    }

    return top->treeNode;

}

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

    if (root == nullptr) {

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

        newNode->data = data;

        newNode->left = newNode->right = nullptr;

        return newNode;

    }

    if (data < root->data) {

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

    } else if (data > root->data) {

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

    } else {

        cout << "Duplicate value " << data << " not inserted." << endl;

        return root;

    }

    return root;

}

// Inorder Traversal using Stack (Left, Root, Right)

void inorder(Node\* root) {

    if (root == nullptr) return;

    StackNode\* stack = nullptr;

    Node\* current = root;

    while (true)

    {

        if (current != nullptr)

        {

            push(stack, current);

            current = current->left;

        }else{

            if (isEmpty(stack)) break; // no more node to travel

            current = top(stack);

            pop(stack);

            cout << current->data << " ";

            current = current->right;

        }

    }

}

// Preorder Traversal using Stack (Root, Left, Right)

void preorder(Node\* root) {

    if (root == nullptr) return;

    StackNode\* stack = nullptr;

    push(stack, root);

    while (!isEmpty(stack)) {

        Node\* current = top(stack);

        pop(stack);

        cout << current->data << " ";  // Print value

        // Push right child first so that left child is processed first

        if (current->right != nullptr) {

            push(stack, current->right);

        }

        if (current->left != nullptr) {

            push(stack, current->left);

        }

    }

}

// Postorder Traversal using two Custom Stacks (Left, Right, Root)

void postorder(Node\* root) {

    if (root == nullptr) return;

    StackNode\* s1 = nullptr;

    StackNode\* s2 = nullptr;

    push(s1, root);

    while (!isEmpty(s1)) {

        Node\* current = top(s1);

        pop(s1);

        push(s2, current);//put from first stack to second and put child in 1 stack

        if (current->left != nullptr) {

            push(s1, current->left);

        }

        if (current->right != nullptr) {

            push(s1, current->right);

        }

    }

    // Print all elements from the second stack

    while (!isEmpty(s2)) {

        cout << top(s2)->data << " ";

        pop(s2);

    }

}

int main() {

    Node\* root = nullptr;

    int choice, value, data;

    cout << "Enter elements with space to insert in tree and enter -1 to stop: ";

    cin >> data;

    while (data != -1) {

        root = insert(root, data);

        cin >> data;

    }

    while (true) {

        cout << "\nMenu:\n";

        cout << "1. Inorder Traversal";

        cout << "   2. Preorder Traversal";

        cout << "   3. Postorder Traversal\n";

        cout << "4. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                cout << "Inorder Traversal: ";

                inorder(root);

                cout << endl;

                break;

            case 2:

                cout << "Preorder Traversal: ";

                preorder(root);

                cout << endl;

                break;

            case 3:

                cout << "Postorder Traversal: ";

                postorder(root);

                cout << endl;

                break;

            case 4:

                return 0;

            default:

                cout << "Invalid choice. Please try again." << endl;

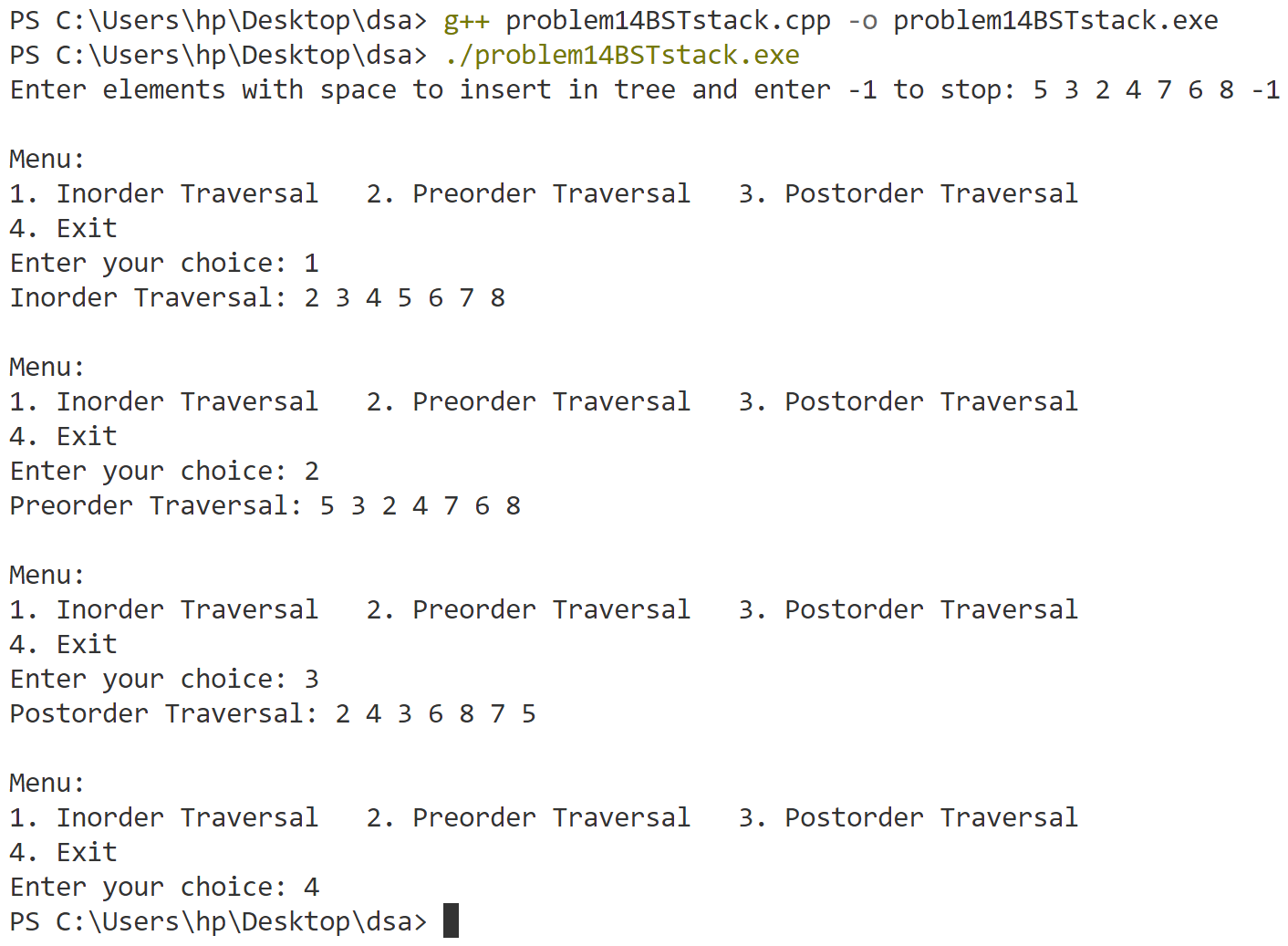
        }

    }

    return 0;

}

**Output**



1. **Implement AVL tree.**

#include <iostream>

#include <cstdlib>

using namespace std;

struct Node {

    int key;

    Node \* left;

    Node \* right;

    int height;

};

int height(Node \* node) {

    return (node == nullptr) ? 0 : node -> height;

}

int max(int a, int b) {

    return (a > b) ? a : b;

}

Node \* createNode(int key) {

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

    node -> key = key;

    node -> left = nullptr;

    node -> right = nullptr;

    node -> height = 1;

    return node;

}

// Right rotate a subtree rooted with y

Node \* rightRotate(Node \* y) {

    Node \* x = y -> left;

    Node \* T2 = x -> right;

    // Perform rotation

    x -> right = y;

    y -> left = T2;

    // Update heights

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

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

    // Return new root

    return x;

}

// Left rotate a subtree rooted with x

Node \* leftRotate(Node \* x) {

    Node \* y = x -> right;

    Node \* T2 = y -> left;

    // Perform rotation

    y -> left = x;

    x -> right = T2;

    // Update heights

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

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

    // Return new root

    return y;

}

int getBalance(Node \* node) {

    if (node == nullptr)

        return 0;

    return height(node -> left) - height(node -> right);

}

Node \* insert(Node \* node, int key) {

    // Perform the normal BST insertion

    if (node == nullptr)

        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 // Duplicate keys are not allowed in the AVL tree

        return node;

    // Update the height of ancestor node and insertion

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

    // Get the balance factor to check whether this node became unbalanced

    int balance = getBalance(node);

    // If the node becomes unbalanced, there are 4 cases

    // Left Left Case

    if (balance > 1 && key < node -> left -> key)

        return rightRotate(node);

    // Right Right Case

    if (balance < -1 && key > node -> right -> key)

        return leftRotate(node);

    // Left Right Case

    if (balance > 1 && key > node -> left -> key) {

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

        return rightRotate(node);

    }

    // Right Left Case

    if (balance < -1 && key < node -> right -> key) {

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

        return leftRotate(node);

    }

    return node;

}

// Function to delete a node from the AVL tree

Node \* deleteNode(Node \* root, int key) {

    // Perform standard BST delete

    if (!root)

        return NULL;

    if (key < root -> key) {

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

    } else if (key > root -> key) {

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

    } else {

        // 0 child leaf node

        if (!root -> left && !root -> right) {

            free(root);

            return NULL;

        } else if (root -> left && !root -> right) // left child

        {

            Node \* temp = root -> left;

            free(root);

            return temp;

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

            Node \* temp = root -> right;

            free(root);

            return temp;

        } else { // both child exist

            Node \* curr = root -> right;

            while (curr -> left) {

                curr = curr -> left;

            }

            root -> key = curr -> key;

            root -> right = deleteNode(root -> right, curr -> key);

        }

    }

    // If the tree had only one node then return

    if (root == nullptr)

        return root;

    // Update height of the current node

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

    // Get the balance factor of this node to check whether this node became unbalanced

    int balance = getBalance(root);

    if (balance > 1) {

        // Left Left Case

        if (getBalance(root -> left) >= 0) {

            return rightRotate(root);

        } else // Left Right Case

        {

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

            return rightRotate(root);

        }

    } else if (balance < -1) {

        // Right Right Case

        if (getBalance(root -> right) <= 0) {

            return leftRotate(root);

        } else // Right Left Case

        {

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

            return leftRotate(root);

        }

    }

    return root;

}

void inorder(Node \* root) {

    if (root != nullptr) {

        inorder(root -> left);

        cout << root -> key << " ";

        inorder(root -> right);

    }

}

void preOrder(Node \* root) {

    if (root != nullptr) {

        cout << root -> key << " ";

        preOrder(root -> left);

        preOrder(root -> right);

    }

}

void postOrder(Node \* root) {

    if (root != nullptr) {

        postOrder(root -> left);

        postOrder(root -> right);

        cout << root -> key << " ";

    }

}

Node \* search(Node \* root, int data) {

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

        return root;

    }

    if (data < root -> key) {

        return search(root -> left, data);

    } else {

        return search(root -> right, data);

    }

}

int main() {

    Node \* root = nullptr;

    int choice, value, data;

    cout << "Enter elements to insert in the tree (enter -1 to stop): ";

    cin >> data;

    while (data != -1) {

        root = insert(root, data);

        cin >> data;

    }

    while (true) {

        cout << "\nMenu:\n";

        cout << "1. Inorder Traversal\n";

        cout << "2. Preorder Traversal\n";

        cout << "3. Postorder Traversal\n";

        cout << "4. Insert a Node\n";

        cout << "5. Delete a Node\n";

        cout << "6. Search" << endl;

        cout << "7. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

        case 1:

            cout << "Inorder Traversal: ";

            inorder(root);

            cout << endl;

            break;

        case 2:

            cout << "Preorder Traversal: ";

            preOrder(root);

            cout << endl;

            break;

        case 3:

            cout << "Postorder Traversal: ";

            postOrder(root);

            cout << endl;

            break;

        case 4:

            cout << "Enter value to insert: ";

            cin >> value;

            root = insert(root, value);

            break;

        case 5:

            cout << "Enter value to delete: ";

            cin >> value;

            root = deleteNode(root, value);

            break;

        case 6:

            cout << "Enter value to search: ";

            cin >> value;

            if (search(root, value) == nullptr) {

                cout << "Value not found." << endl;

            } else {

                cout << "Value found." << endl;

            }

            break;

        case 7:

            return 0;

        default:

            cout << "Invalid choice. Please try again." << endl;

        }

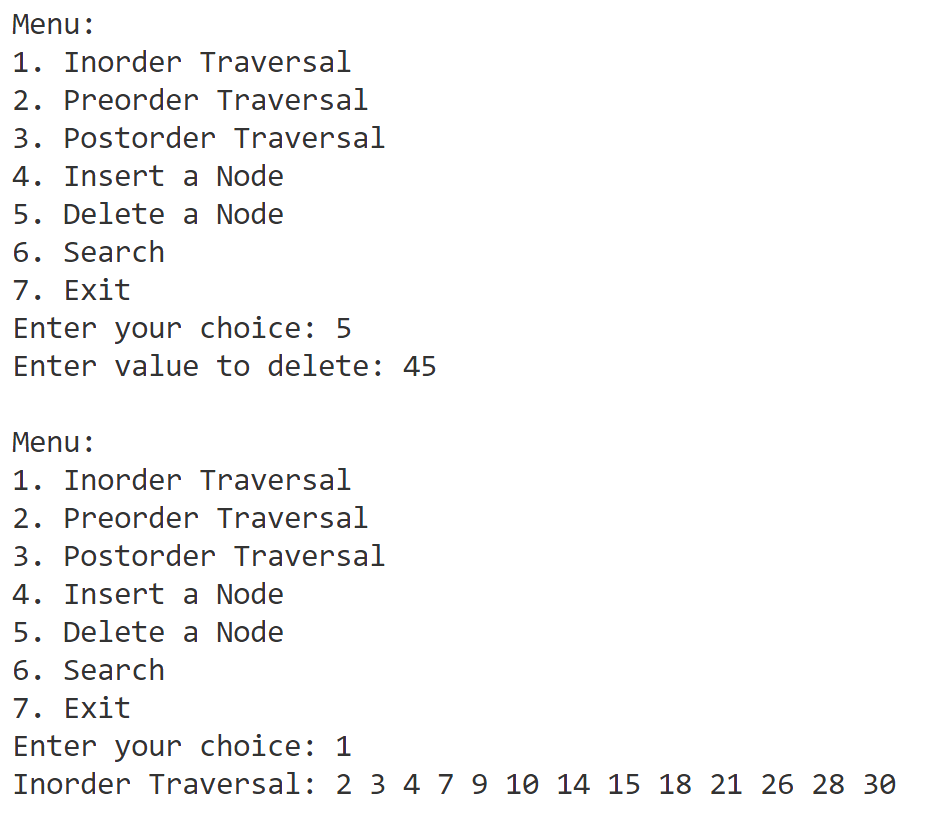
    }

    return 0;

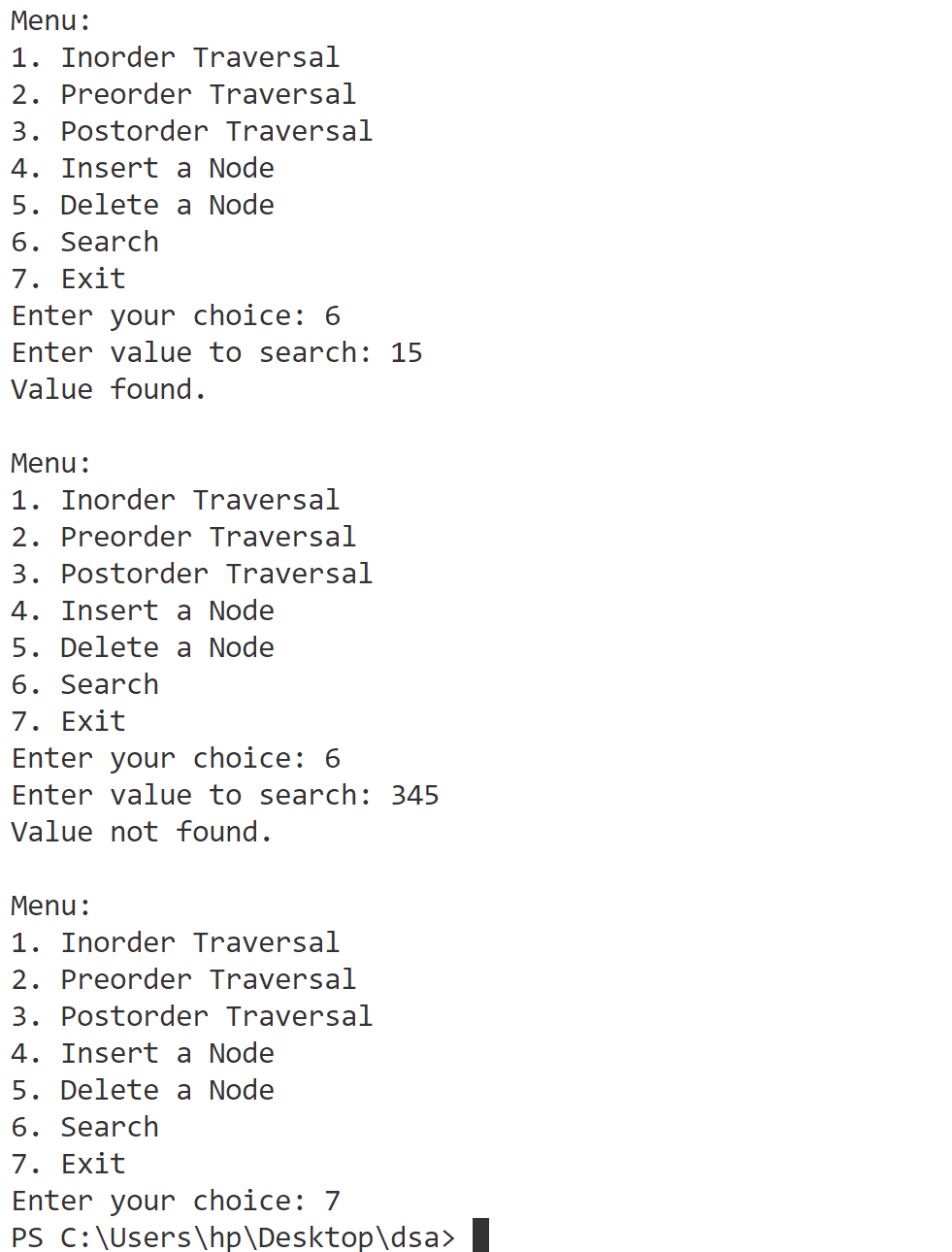
}

**Output**



****

****

****

1. **Implement graph traversals (DFS & BFS).**

#include <iostream>

#include <cstdlib>

using namespace std;

// Global variables for Queue

int\* queue;

int front = -1, rear = -1, queue\_capacity;

// Global visited array and adjacency matrix

int\* visited;

int\*\* adjMatrix;

int n;  // Number of vertices

void enqueue(int value) {

    if (rear >= queue\_capacity - 1) {

        cout << "Queue Overflow!" << endl;

        return;

    }

    if (front == -1) front = 0;

    queue[++rear] = value;

}

int dequeue() {

    if (front == -1 || front > rear) {

        cout << "Queue Underflow!" << endl;

        return -1;

    }

    int dequeuedValue = queue[front++];

    if (front > rear) {

        front = rear = -1;

    }

    return dequeuedValue;

}

bool isQueueEmpty() {

    return (front == -1 || front > rear);

}

bool isQueueFull() {

    return (rear == queue\_capacity - 1);

}

void Dfs(int v) {

    cout << v << " ";

    visited[v] = 1;  // Mark node as visited (1)

    for (int i = 0; i < n; i++) {  // Explore all neighbors

        if (adjMatrix[v][i] == 1 && visited[i] == 0) {  // If not visited

            Dfs(i);

        }

    }

}

void bfs(int v) {

    enqueue(v);  // Enqueue the starting vertex

    visited[v] = 1;

    cout << v << " ";

    while (!isQueueEmpty()) {

        int node = dequeue();  // Dequeue a vertex from the queue

        for (int i = 0; i < n; i++) {

            if (adjMatrix[node][i] == 1 && visited[i] == 0) {  // If not visited

                visited[i] = 1;  // Mark node as visited

                cout << i << " ";  // Print the visited vertex

                enqueue(i);

            }

        }

    }

}

int main() {

    int m, choice, source;

    cout << "Enter the number of vertices: ";

    cin >> n;

    // Dynamically allocate memory for the adjacency matrix

    adjMatrix = (int\*\*)malloc(n \* sizeof(int\*));

    for (int i = 0; i < n; i++) {

        adjMatrix[i] = (int\*)malloc(n \* sizeof(int));

    }

    // Initialize the adjacency matrix to 0 (no edges)

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            adjMatrix[i][j] = 0;

        }

    }

    cout << "Enter the number of edges: ";

    cin >> m;

    cout << "Enter the edges (u v) where u and v are vertices connected by an edge:" << endl;

    for (int i = 0; i < m; i++) {

        int u, v;

        cin >> u >> v;

        adjMatrix[u][v] = 1;

        adjMatrix[v][u] = 1; // For undirected graph

    }

    visited = (int\*)malloc(n \* sizeof(int));

    for (int i = 0; i < n; i++) {

        visited[i] = 0;  // Mark all vertices as unvisited initially

    }

    queue\_capacity = n;

    queue = (int\*)malloc(queue\_capacity \* sizeof(int));

    cout << "\nChoose traversal method:" << endl;

    cout << "1. DFS\n2. BFS" << endl;

    cin >> choice;

    cout << "Enter the source node for traversal: ";

    cin >> source;

    switch (choice) {

        case 1:

            cout << "\nDFS Traversal starting from node " << source << ": ";

            Dfs(source);

            break;

        case 2:

            cout << "\nBFS Traversal starting from node " << source << ": ";

            bfs(source);

            break;

        default:

            cout << "Invalid choice!" << endl;

    }

    for (int i = 0; i < n; i++) {

        free(adjMatrix[i]);

    }

    free(adjMatrix);

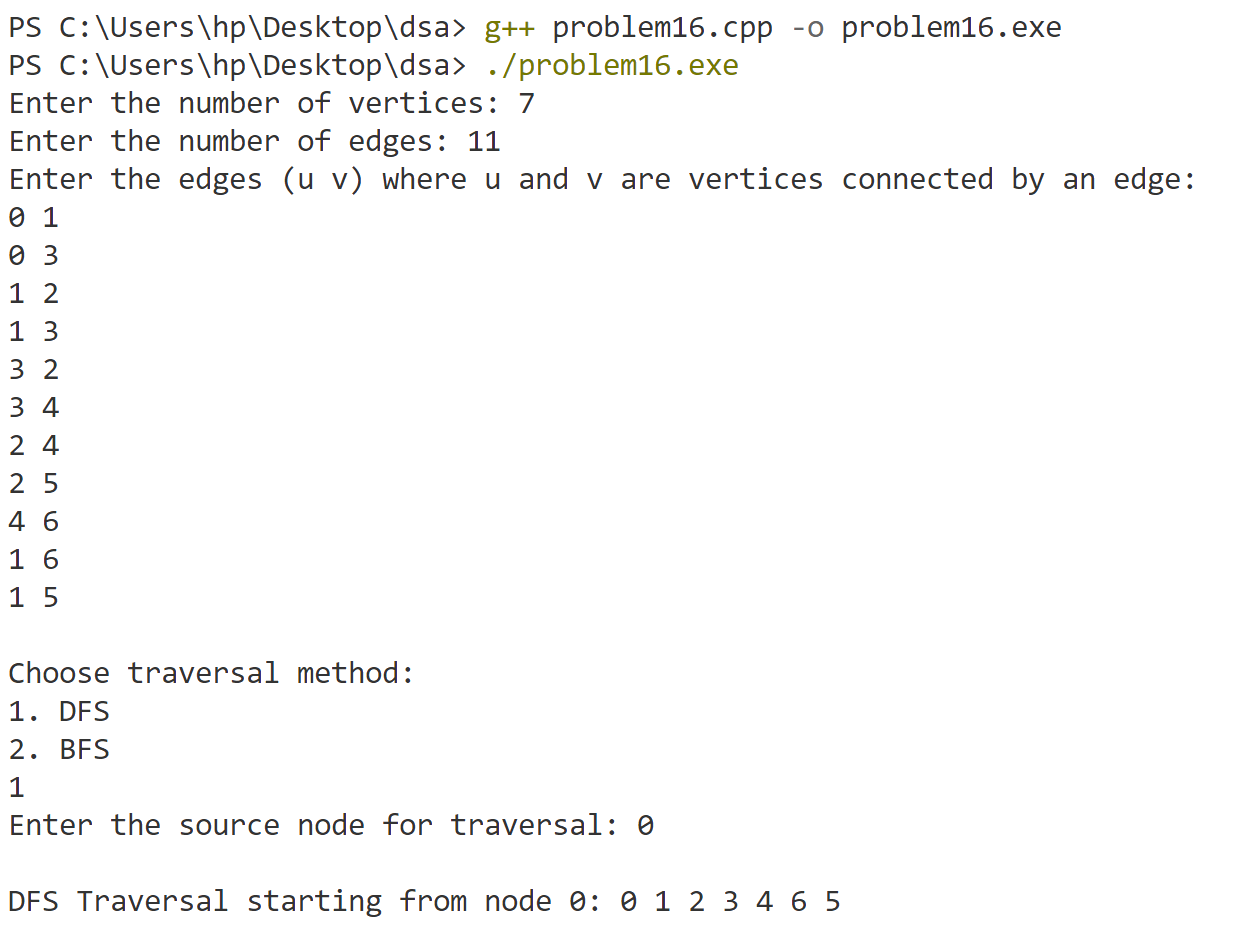
    free(visited);

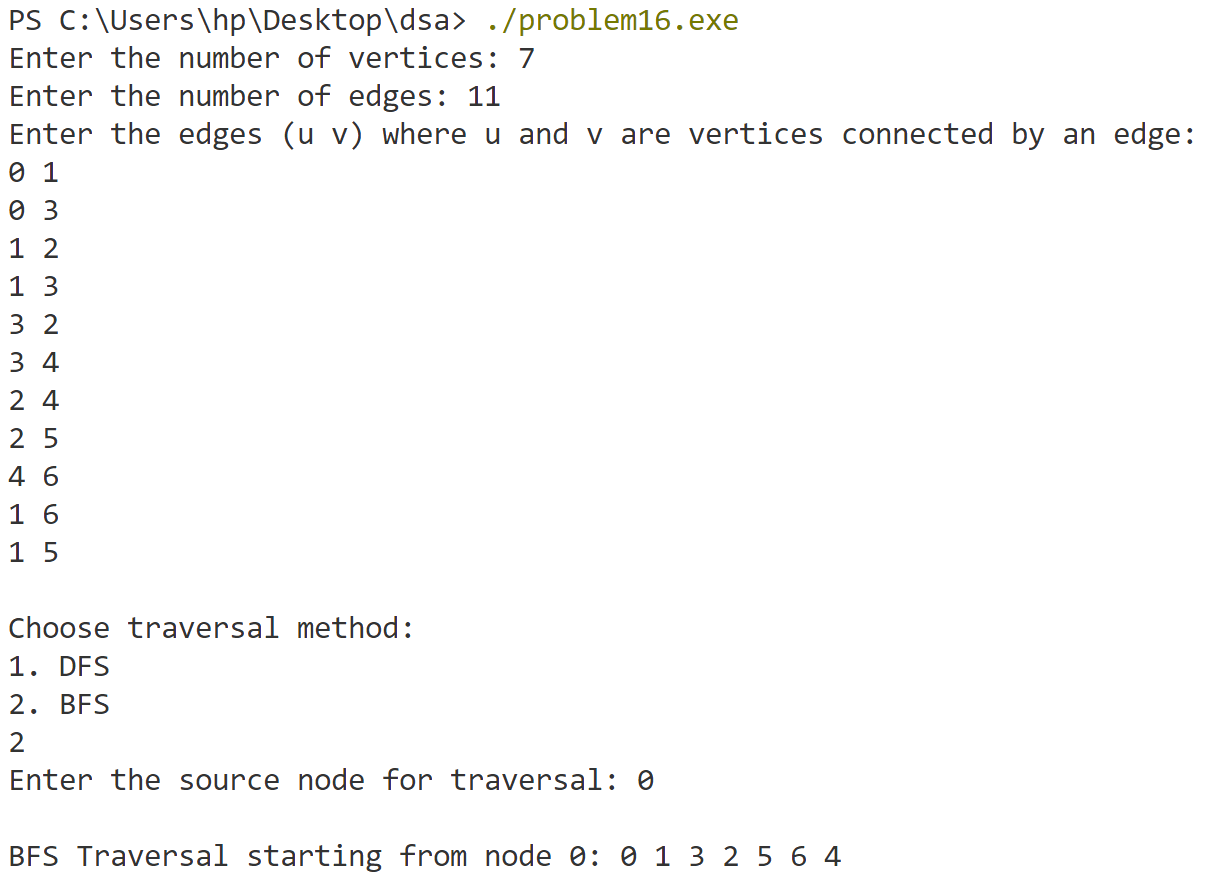
    free(queue);

    return 0;

}

**Output**

****

****

1. **Implement dijkstra’s shortest path greedy algorithm and Kruskal’s minimum spanning tree greedy algorithm.**

**Output**

1. **Implement Longest Common Subsequence dynamic programming algorithm.**

**Output**

1. **Perform various sorting techniques (heap, count, radix, bucket).**

**Output**

1. **Perform divide and conquer sorting techniques (merge, quick).**

**Output**