**BAHRIA UNIVERSITY ISLAMABAD**

**Department of Computer Science**



**Data Structures and Algorithms**

**Assignment No.2**

**Name:** Saad Ahmad

**Section:** BS-CS-3A

**Date:** 26/11/2023

**Question 1: Queues**

a. Define a queue and explain its basic operations.

b. Implement a simple queue in Python. Include functions for enqueue and dequeue operations.

a) A queue is a data structure that follows the First In, First Out (FIFO) principle, which means that the first element added to the queue will be the first one to be removed. It can be visualized as a line of elements, where elements are added at one end (rear or tail) and removed from the other end (front or head). The basic operations of a queue include:

1. **Enqueue (Insertion):**

Add an element to the rear/tail of the queue.

1. **Dequeue (Deletion):**

Remove the element from the front/head of the queue.

1. **Front:**

Retrieve the front/head element of the queue without removing it.

1. **Empty:**

Check whether the queue is empty.

1. **Size:**

Get the number of elements present in the queue.

b) Code:

#include <iostream>

using namespace std;

// Constant to define the maximum size of the queue.

const int MAXSIZE = 5;

// Class representing a linear queue.

class LinearQueue {

int front; // Index of the front element.

int back; // Index of the back element.

int arr[MAXSIZE]; // Array to store queue elements.

public:

// Constructor to initialize the queue.

LinearQueue() {

front = -1;

back = -1;

}

// Function to enqueue an element into the queue.

void Enqueue(int temp) {

// Check if the queue is full.

if (Full()) {

cout << "Queue is full" << endl;

}

else {

back++;

arr[back] = temp;

}

// If the queue was empty, update the front index.

if (front == -1) {

front++;

}

}

// Function to dequeue an element from the queue.

void Dequeue() {

// Check if the queue is empty.

if (Empty()) {

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

}

else {

front++;

}

// If front surpasses back, reset indices to indicate an empty queue.

if (front > back) {

front = -1;

back = -1;

}

}

// Function to check if the queue is empty.

bool Empty() {

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

}

// Function to check if the queue is full.

bool Full() {

return (back == MAXSIZE - 1);

}

// Function to get the front element of the queue.

int getFront() {

return arr[front];

}

};

// Main function to test the LinearQueue class.

int main() {

// Create an instance of the LinearQueue class.

LinearQueue q1;

// Check if the queue is initially empty.

if (q1.Empty()) {

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

}

else {

cout << "Queue is not empty" << endl;

}

// Enqueue elements into the queue.

q1.Enqueue(5);

q1.Enqueue(4);

q1.Enqueue(3);

q1.Enqueue(2);

q1.Enqueue(1);

// Attempt to enqueue an element into a full queue.

q1.Enqueue(0);

// Display the front element of the queue.

cout << q1.getFront() << endl;

// Dequeue elements from the queue.

q1.Dequeue();

cout << q1.getFront() << endl;

q1.Dequeue();

cout << q1.getFront() << endl;

q1.Dequeue();

cout << q1.getFront() << endl;

q1.Dequeue();

cout << q1.getFront() << endl;

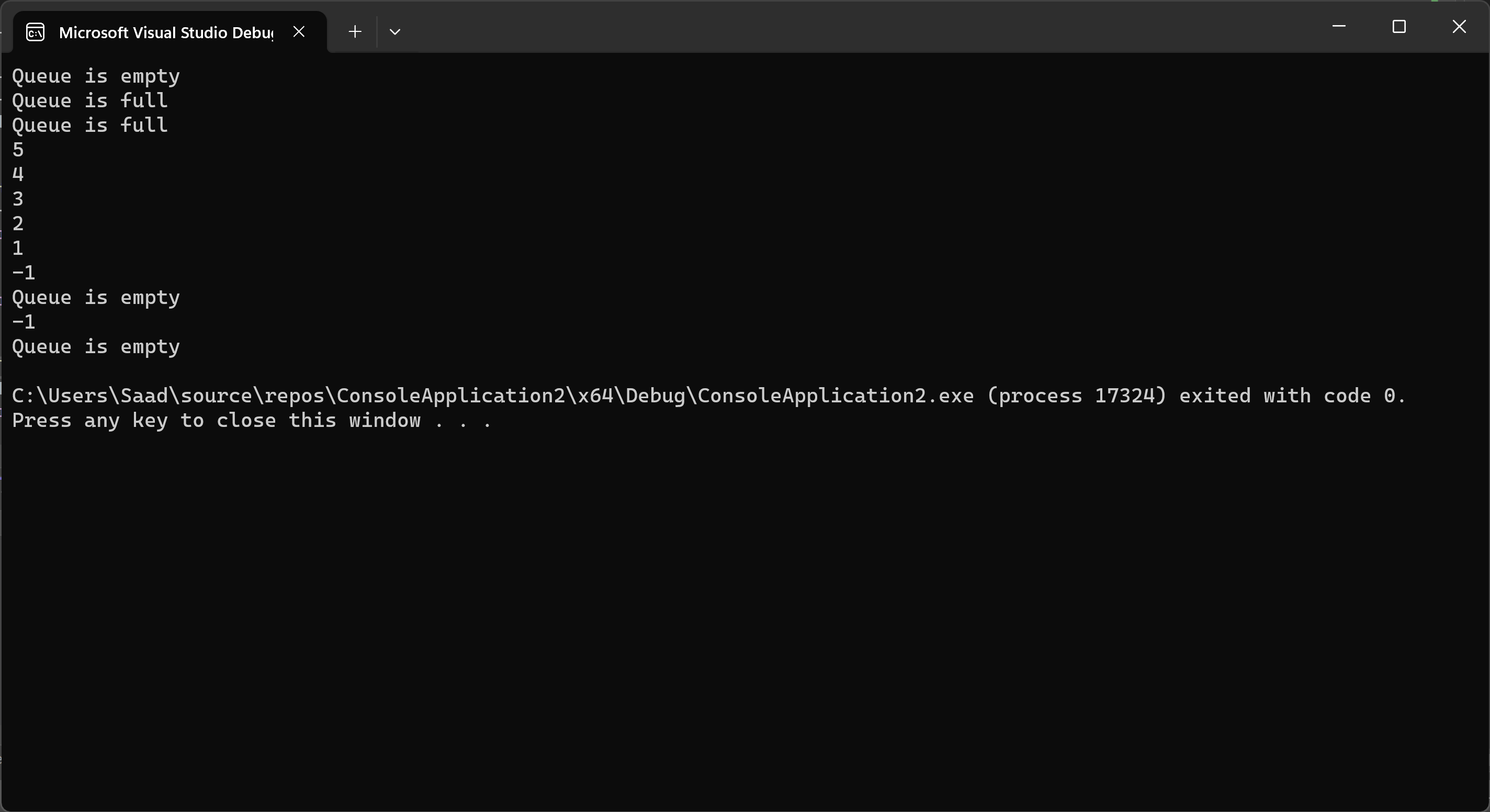
// Attempt to dequeue an element from an empty queue.

q1.Dequeue();

return 0;

}

Output:



**Question 2: Stacks**

a. Define a stack and explain its basic operations.

b. Write a Python function to check if a given string of parentheses is balanced using a stack.

a) A stack is a data structure that follows the Last In, First Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed. It can be visualized as a collection of elements with two main operations:

1. **Push (Insertion):**
   * Add an element to the top of the stack.
2. **Pop (Deletion):**
   * Remove the element from the top of the stack.
3. **Top (Peek):**
   * Retrieve the element at the top of the stack without removing it.
4. **Empty:**
   * Check whether the stack is empty.
5. **Size:**
   * Get the number of elements present in the stack.

b) Code:

void isBalanced() {

// Initialize a stack to store opening parentheses, braces, and brackets.

stack<char> s;

// Input the expression to be checked for balanced parentheses.

string input;

cin >> input;

// Iterate through each character in the input string.

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

// Check if the current character is an opening parenthesis, brace, or bracket.

if (input[i] == '(' || input[i] == '{' || input[i] == '[') {

s.push(input[i]);

}

// Check if the current character is a closing parenthesis, brace, or bracket

// and matches the corresponding opening character at the top of the stack.

else if ((input[i] == ')' && s.top() == '(') || (input[i] == '}' && s.top() == '{') || (input[i] == ']' && s.top() == '[')) {

// Pop the matching opening character from the stack.

s.pop();

}

// If the current character is none of the above, the expression is invalid.

else {

cout << "Invalid Expression!" << endl;

return;

}

}

// Check if the stack is empty, indicating that all opening characters had matching closings.

if (s.empty()) {

cout << "Expression is balanced" << endl;

}

// If the stack is not empty, there are unmatched opening characters, making the expression unbalanced.

else {

cout << "Expression is not balanced " << endl;

}

}

**Question 3: Single Linked List**

a. Describe the structure of a singly linked list.

b. Write a Python class to represent a singly linked list. Include methods to add a node at the end and display the list.

a) A singly linked list is a data structure composed of nodes where each node contains two parts: data and a reference (or link) to the next node in the sequence. The basic structure of a singly linked list includes:

1. **Node:**
   * Each node in the linked list contains two fields:
     + **Data:** This is the actual information or payload that the node holds.
     + **Next (Link or Pointer):** This field contains a reference to the next node in the sequence. It is a pointer or reference that points to the memory address of the next node.
2. **Head:**
   * The head is a reference or pointer to the first node in the linked list. It serves as the starting point for traversing the list.

b) Code:

#include <iostream>

using namespace std;

// Node structure for a singly linked list

struct Node {

int data; // Data of the node

Node\* next; // Pointer to the next node

};

// Class for a singly linked list

class List {

private:

Node\* head; // Pointer to the head of the list

public:

// Constructor: Initialize an empty list

List() {

head = NULL;

}

// Destructor: Clean up allocated memory when the list is destroyed

~List() {

delete head;

}

// Check if the list is empty

bool IsListEmpty() {

if (head == NULL) {

return true;

}

else {

return false;

}

}

// Insert a new node with a given value at the end of the list

void insertAtEnd(int value) {

// Create a new node

Node\* temp = new Node();

temp->data = value;

temp->next = NULL;

// If the list is empty, set the new node as the head

if (IsListEmpty()) {

head = temp;

}

else {

// Traverse to the end of the list and add the new node

Node\* check = head;

while (check->next != NULL) {

check = check->next;

}

check->next = temp;

}

}

// Traverse and print the elements of the list

void traverse() {

Node\* temp = head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

};

// Main function

int main() {

// Create a singly linked list object

List l1;

// Insert elements at the end of the list

l1.insertAtEnd(5);

l1.insertAtEnd(15);

l1.insertAtEnd(25);

l1.insertAtEnd(35);

l1.insertAtEnd(45);

l1.insertAtEnd(55);

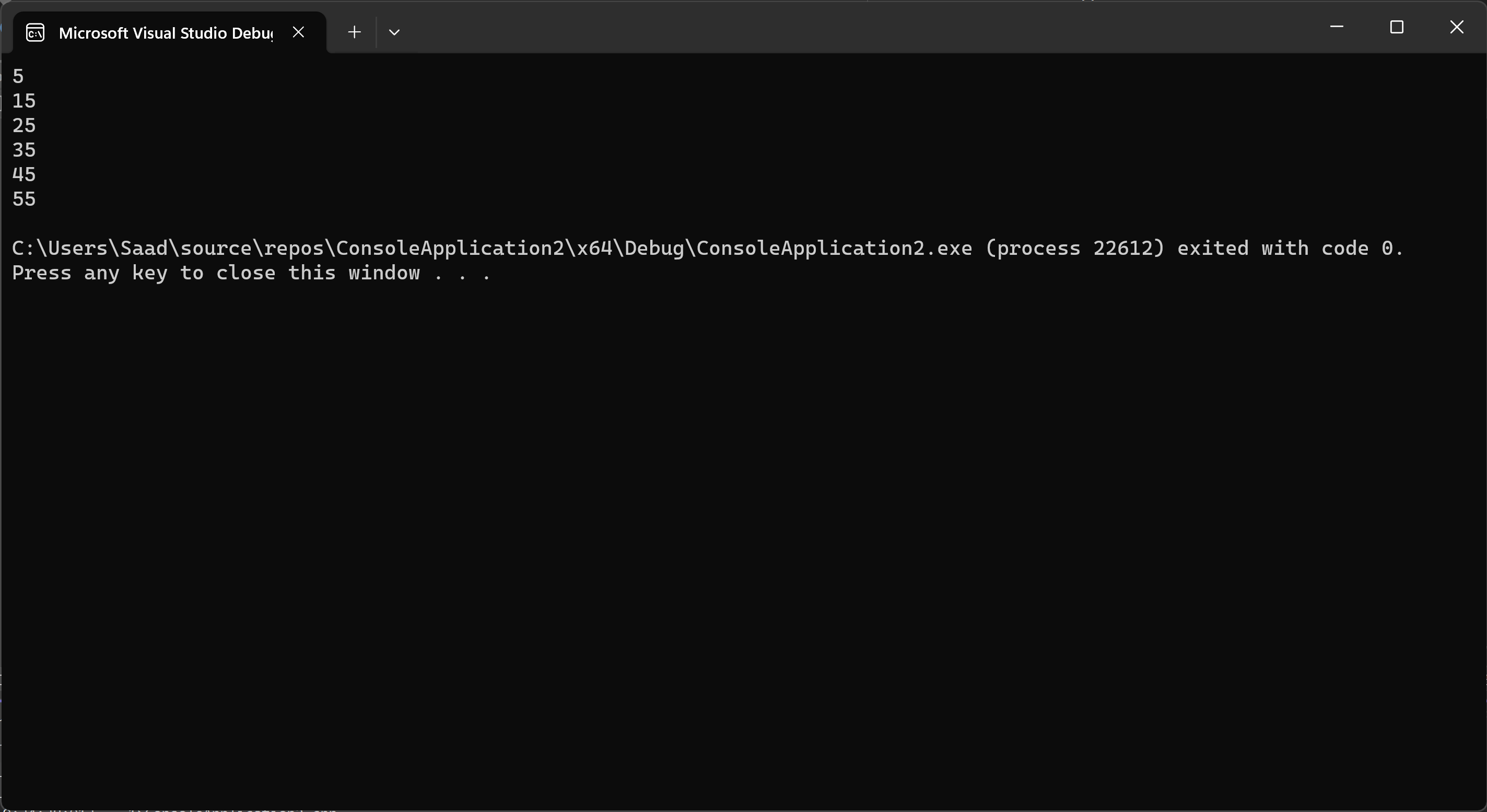
// Traverse and print the elements of the list

l1.traverse();

return 0;

}

Output:



**Question 4: Double Linked List**

a. Compare a double linked list with a singly linked list. Highlight the advantages of a double linked list.

b. Implement a Python class for a doubly linked list. Include methods to add a node at the beginning and display the list in reverse.

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Singly Linked List** | **Doubly Linked List** |
| **Structure** | Each node points to the next. | Each node points to the next and the previous. |
| **Traversal** | Forward traversal only. | Both forward and backward traversal are possible. |
| **Memory Usage** | Requires less memory per node (only next pointer). | Requires more memory per node (next and previous pointers). |
| **Insertion/Deletion (Beginning)** | Efficient. | Efficient. |
| **Insertion/Deletion (End/Middle)** | Less efficient. Requires traversing to find the previous node. | Efficient. No need to traverse to find the previous node. |
| **Reverse Iteration** | Not efficient. | Efficient. |
| **Flexibility** | Less flexible due to unidirectional pointers. | More flexible due to bidirectional pointers. |
| **Applications** | Commonly used in scenarios where backward traversal is not required, and memory efficiency is crucial. | Used when bidirectional traversal, efficient insertion/deletion, or reverse iteration is required. |

**Advantages of a Doubly Linked List:**

1. **Bidirectional Traversal:**
   * Doubly linked lists allow for efficient traversal in both forward and backward directions. This is advantageous in scenarios where backward traversal is a common operation.
2. **Ease of Deletion/Insertion:**
   * Deletion and insertion operations are more straightforward and efficient, especially when it comes to operations in the middle or end of the list. There's no need to traverse the list to find the previous node.
3. **Flexibility:**
   * Offers greater flexibility in certain operations due to bidirectional pointers.
4. **Reverse Iteration:**
   * Useful in scenarios where reverse iteration is required, such as in certain algorithms or applications.

#include <iostream>

using namespace std;

// Node structure for the doubly linked list

struct Node {

int value; // Data of the node

Node\* next; // Pointer to the next node in the list

Node\* prev; // Pointer to the previous node in the list

};

// Doubly Linked List class

class DLL {

Node\* head; // Pointer to the head of the list

public:

// Constructor to initialize an empty list

DLL() {

head = NULL;

}

// Function to insert a new element at the beginning of the list

void insert(int val) {

// Create a new node

Node\* temp = new Node();

temp->value = val;

// If the list is empty, set the new node as the head

if (head == NULL) {

temp->next = NULL;

temp->prev = NULL;

head = temp;

}

else {

// If the list is not empty, adjust pointers to insert at the beginning

head->prev = temp;

temp->next = head;

temp->prev = NULL;

head = temp;

}

}

// Function to display the elements of the list in reverse order

void display() {

Node\* temp = head;

// Move to the last node in the list

while (temp->next != NULL) {

temp = temp->next;

}

// Traverse the list in reverse and print the elements

while (temp != NULL) {

cout << temp->value << endl;

temp = temp->prev;

}

}

};

// Main function

int main() {

// Create a Doubly Linked List object

DLL D1;

// Insert elements into the list

D1.insert(15);

D1.insert(25);

D1.insert(35);

D1.insert(45);

D1.insert(55);

// Display the elements of the list in reverse order

D1.display();

return 0;

}

Output:

