

**National University of Computer and Emerging**

**Sciences**

**Chiniot-Faisalabad Campus BS (Artificial Intelligence)**

|  |  |
| --- | --- |
| **Name** | **M.Abdul Hanan** |
| **Reg.NO.** | **22F-3104** |
| **Section** | **BS(Ai)-3A1** |
| **Course** | **Data Structures** |
| **Department** | **CS Department** |
| **Assignment** | **#2** |

**Task no 1:**

#include <iostream>

#include <string>

using namespace std;

struct Node {

char Data;

Node\* next;

Node(char Data) : Data(Data), next(NULL) {}

};

class Stack {

private:

public:

Node\* top;

Stack() : top(NULL) {}

// Push a character onto the stack

void push(char Data) {

Node\* newNode = new Node(Data);

newNode->next = top;

top = newNode;

display(Data);

}

// Pop and return a character from the stack

char pop() {

if (isEmpty()) {

cout << "\nStack is Empty" << endl;

return '\0';

}

char Data = top->Data;

Node\* temp = top;

top = top->next;

delete temp;

display();

return Data;

}

// Check if the stack is empty

bool isEmpty() {

return top == NULL;

}

// Display the contents of the stack, with an optional operation character

void display(char operation = '\0') {

cout << "\nStack: ";

Node\* current = top;

while (current != NULL) {

cout << current->Data << " ";

current = current->next;

}

if (operation != '\0') {

cout << operation;

}

cout << endl;

}

// Check if a character is an operator

bool isOperator(char Opera) {

return (Opera == '+' || Opera == '-' || Opera == '\*' || Opera == '/' || Opera == '%');

}

// Get the precedence of an operator

int getPrecedence(char Opera) {

if (Opera == '+' || Opera == '-') {

return 1;

}

else if (Opera == '\*' || Opera == '/' || Opera == '%') {

return 2;

}

else {

return 0;

}

}

// Convert an infix expression to postfix notation

string infixToPostfix(string infix) {

string result;

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

char ch = infix[i];

if (ch >= '0' && ch <= '9') {

result += ch;

}

else if (ch == '(') {

push(ch);

}

else if (ch == ')') {

while (!isEmpty() && top->Data != '(') {

result += pop();

}

pop(); // Pop the '('

}

else if (isOperator(ch)) {

while (!isEmpty() && getPrecedence(top->Data) >= getPrecedence(ch) && top->Data != '(') {

result += pop();

}

push(ch);

}

}

while (!isEmpty()) {

if (top->Data == '(') {

pop(); // Pop the remaining '('

}

else {

result += pop();

}

}

return result;

}

};

int main() {

Stack s1;

string infixExp;

cout << "Enter infix: ";

cin >> infixExp;

cout << "Infix: " << infixExp;

string postfixExp = s1.infixToPostfix(infixExp);

// Reverse the infix expression

reverse(infixExp.begin(), infixExp.end());

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

if (infixExp[i] == ')') {

infixExp[i] = '(';

}

else {

infixExp[i] == ')';

}

}

// Convert the reversed infix expression to prefix notation

string prefixExp = s1.infixToPostfix(infixExp);

// Reverse the prefix expression back

reverse(prefixExp.begin(), prefixExp.end());

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

if (prefixExp[i] == ')') {

prefixExp[i] = '(';

}

else {

prefixExp[i] == ')';

}

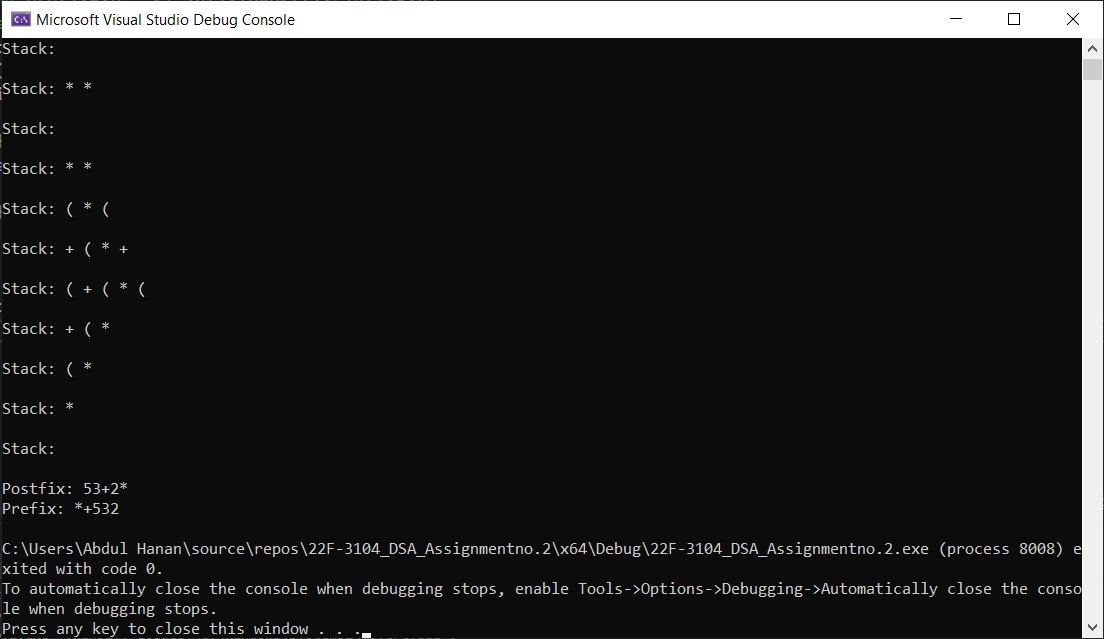
}

cout <<endl<< "Postfix: " << postfixExp << endl;

cout << "Prefix: " << prefixExp << endl;

return 0;

}

**Screen Shot:** ****

**Task no 2:**

#include<iostream>

using namespace std;

class Node

{

public:

string data;

Node\* next;

Node(string value)

{

data = value;

next = NULL;

}

};

class Stack

{

private:

Node\* Top;

Node\* Redo\_Top;

public:

Stack()

{

Top = NULL;

Redo\_Top = NULL;

}

// Method to check if the stack is empty

bool isEmpty()

{

return Top == NULL;

}

// Method to add a text to the stack

void push(string data)

{

Node\* newNode = new Node(data);

newNode->next = Top;

Top = newNode;

}

// Method to undo the last text

void undo()

{

if (isEmpty())

{

cout << "Stack is empty! Underflow" << endl;

return;

}

Node\* temp = Top;

Top = Top->next;

temp->next = Redo\_Top;

Redo\_Top = temp;

}

// Method to redo a text by moving it back to the main stack

void Redo()

{

if (Redo\_Top == NULL)

{

cout << "Nothing to Redo :" << endl;

return;

}

Node\* redoNode = Redo\_Top;

Redo\_Top = Redo\_Top->next;

redoNode->next = Top;

Top = redoNode;

}

// Method to display the stack text data

void display()

{

if (isEmpty())

{

cout << "Stack is Empty" << endl;

}

else

{

Node\* temp = Top;

while (temp != nullptr)

{

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

}

// Destructor to free dynamically allocated memory and clear the stack

~Stack()

{

while (!isEmpty())

{

undo();

}

}

};

int main()

{

Stack s;

string choice, data;

while (true)

{

cout << "1. Add Text" << endl;

cout << "2. Undo Text" << endl;

cout << "3. Redo Text" << endl;

cout << "4. Display Text" << endl;

cout << "5. Exit" << endl;

cout << "Enter your choice : ";

cin >> choice;

if (choice == "1")

{

cout << "Enter Command : ";

cin >> data;

s.push(data);

}

else if (choice == "2")

{

s.undo();

}

else if (choice == "3")

{

s.Redo();

}

else if (choice == "4")

{

s.display();

}

else if (choice == "5")

{

cout << "Goodbye!" << endl;

break;

}

else

{

cout << "Invalid choice, please try again" << endl;

}

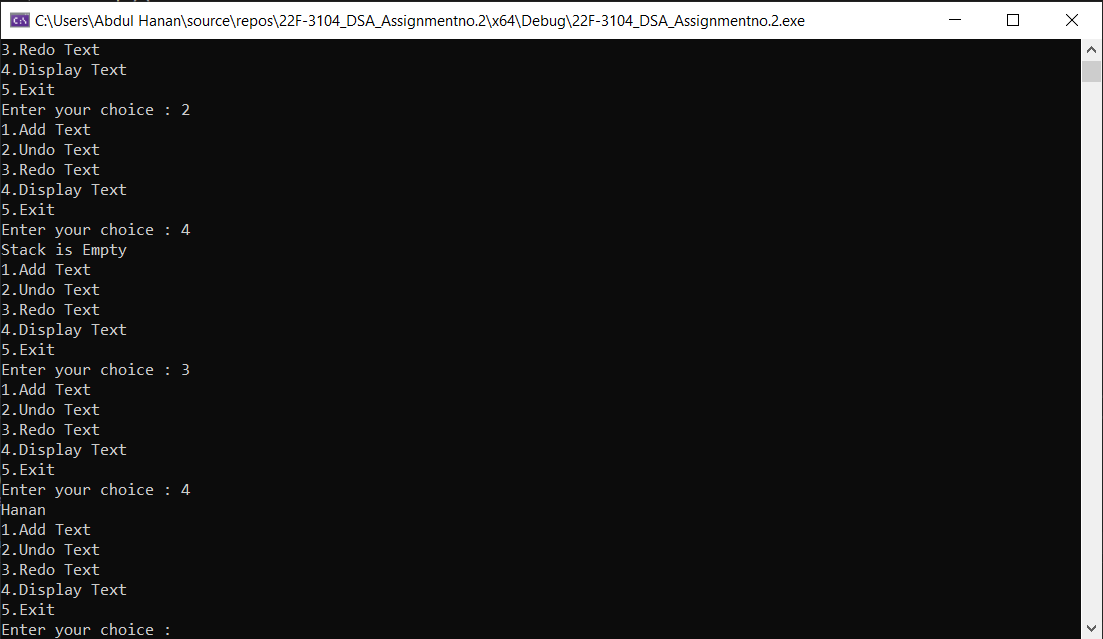
}

system("pause>0");

return 0;

}

**Screen Shot:**

****

**Task no 3 :**

#include <iostream>

using namespace std;

class Stack {

public:

int\* arr;

int top;

int capacity;

Stack(int size) {

capacity = size;

arr = new int[capacity];

top = -1;

}

bool isEmpty() {

return top == -1; // The stack is empty if the top index is -1

}

bool isFull() {

return top == capacity - 1; // The stack is full if the top index equals capacity - 1

}

void push(int value) {

if (isFull()) {

cout << "Stack is full" << endl;

return;

}

arr[++top] = value; // Increment the top index and add the value to the stack

}

int pop() {

if (isEmpty()) {

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

return -1;

}

return arr[top--]; // Return and decrement the top index to remove the top element

}

int peek() {

if (isEmpty()) {

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

return -1;

}

return arr[top]; // Return the top element

}

~Stack() {

delete[] arr; // Deallocate the memory

}

};

class QueueUsingStacks {

private:

Stack stack1; // First stack for enqueue operation

Stack stack2; // Second stack for dequeue operation

public:

QueueUsingStacks(int size) : stack1(size), stack2(size) {} // Constructor for the queue

bool isEmpty() {

return stack1.isEmpty() && stack2.isEmpty(); // The queue is empty if both stacks are empty

}

bool isFull() {

return stack1.isFull() || stack2.isFull(); // The queue is full if either stack is full

}

void enqueue(int value) {

if (isFull()) {

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

return;

}

// transfer elements from stack1 to stack2 to maintain the order for enqueue

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

stack1.push(value);

// Transfer elements back to stack1 to keep the correct order for dequeue

while (!stack2.isEmpty()) {

stack1.push(stack2.pop());

}

}

int dequeue() {

if (isEmpty()) {

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

return -1;

}

return stack1.pop(); // Dequeue by popping the top element from stack1

}

void display() {

if (isEmpty()) {

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

}

else {

cout << "Queue Contents: ";

for (int i = 0; i <= stack1.top; i++) {

cout << stack1.arr[i] << " ";

}

cout << endl;

}

}

};

int main() {

int size;

cout << "Enter size of the Queue: ";

cin >> size;

QueueUsingStacks queue(size);

int choice, value;

while (true) {

cout << "Queue Menu:" << endl;

cout << "1. Enqueue" << endl;

cout << "2. Dequeue" << endl;

cout << "3. Display Queue" << endl;

cout << "4. Check if Empty" << endl;

cout << "5. Check if Full" << endl;

cout << "6. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter a value to enqueue: ";

cin >> value;

queue.enqueue(value);

break;

case 2:

value = queue.dequeue();

if (value != -1) {

cout << "Dequeued value: " << value << endl;

}

break;

case 3:

queue.display();

break;

case 4:

if (queue.isEmpty()) {

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

}

else {

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

}

break;

case 5:

if (queue.isFull()) {

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

}

else {

cout << "Queue is not full." << endl;

}

break;

case 6:

cout << "Exit" << endl;

return 0;

default:

cout << "Invalid choice" << endl;

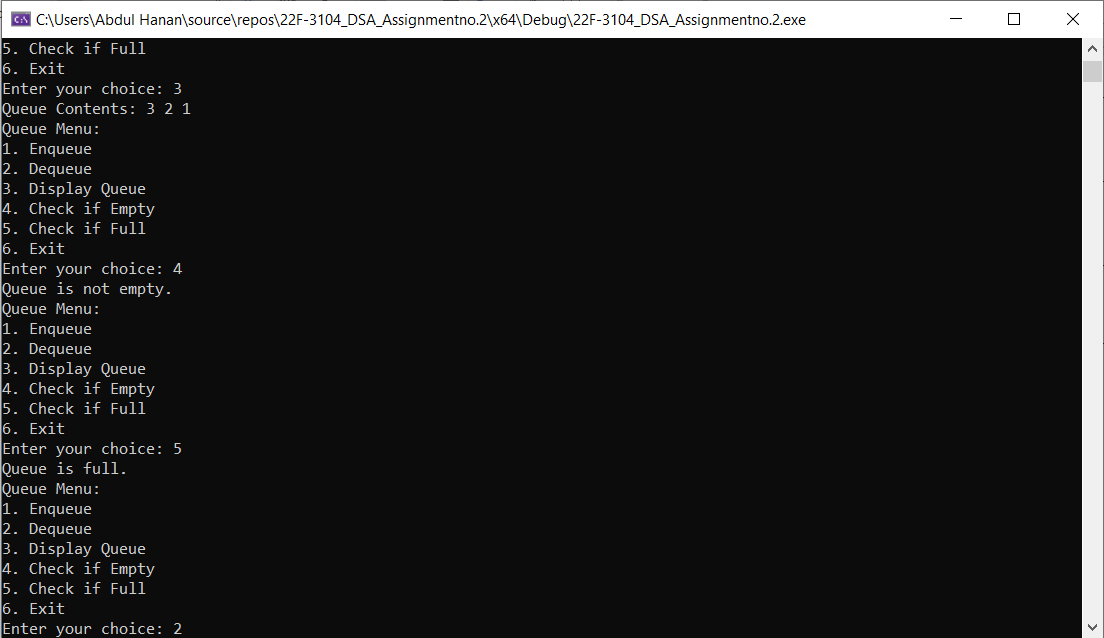
}

}

return 0;

}

**Screen Shot:**

****

**Task no 4:**

#include <iostream>

#include <Windows.h>

using namespace std;

template<typename T>

class Deque {

private:

T\* dequeArr;

int maxSize;

int rear, front;

public:

Deque(int size);

// Method to check if the deque is empty

bool isEmpty();

// Method to check if the deque is full

bool isFull();

// Method to insert an element at the front of the deque

void insertFront(T value);

// Method to insert an element at the back of the deque

void insertBack(T value);

// Method to remove an element from the front of the deque

T removeFront();

// Method to remove an element from the back of the deque

T removeBack();

// Method to display the contents of the deque

void display();

// free dynamically allocated memory.

~Deque();

};

template<typename T>

Deque<T>::Deque(int size) {

maxSize = size;

dequeArr = new T[maxSize];

front = rear = -1;

}

// Method to check if the deque is empty

template<typename T>

bool Deque<T>::isEmpty() {

if (front == -1) {

cout << "Status: Queue is Empty." << endl;

return true;

}

cout << "Status: Queue is not Empty." << endl;

return false;

}

// Method to check if the deque is full

template<typename T>

bool Deque<T>::isFull() {

if ((front == 0 && rear == maxSize - 1) || front == rear + 1) {

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

return true;

}

cout << "Queue is not Full." << endl;

return false;

}

// Method to insert an element at the front of the deque

template<typename T>

void Deque<T>::insertFront(T value) {

if (isFull()) {

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

return;

}

if (front == -1) {

front = 0;

rear = 0;

}

else if (front == 0) {

front = maxSize - 1;

}

else {

front = front - 1;

}

dequeArr[front] = value;

cout << value << " has been enqueued in from the front." << endl;

}

// Method to insert an element at the back of the deque

template<typename T>

void Deque<T>::insertBack(T value) {

if (isFull()) {

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

return;

}

if (front == -1) {

front = 0;

rear = 0;

}

else if (rear == maxSize - 1) {

rear = 0;

}

else {

rear = rear + 1;

}

dequeArr[rear] = value;

cout << value << " has been enqueued in from the back." << endl;

}

// Method to remove an element from the front of the deque

template<typename T>

T Deque<T>::removeFront() {

if (isEmpty()) {

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

return T();

}

T value = dequeArr[front];

if (front == rear) {

front = -1;

rear = -1;

}

else if (front == maxSize - 1) {

front = 0;

}

else {

front = front + 1;

}

cout << value << " has been dequeued from the front." << endl;

return value;

}

// Method to remove an element from the back of the deque.

template<typename T>

T Deque<T>::removeBack() {

if (isEmpty()) {

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

return T();

}

T value = dequeArr[rear];

if (front == rear) {

front = -1;

rear = -1;

}

else if (rear == 0) {

rear = maxSize - 1;

}

else {

rear = rear - 1;

}

cout << value << " has been dequeued from the back." << endl;

return value;

}

// Method to display the contents of the deque

template<typename T>

void Deque<T>::display() {

cout << "\nYour queue: ";

if (front == -1) {

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

return;

}

int i = front;

do {

cout << dequeArr[i] << " ";

i = (i + 1) % maxSize;

} while (i != (rear + 1) % maxSize);

cout << endl;

}

// Destructor to free dynamically allocated memory

template<typename T>

Deque<T>::~Deque() {

delete[] dequeArr;

}

int main() {

int size, choice;

int value;

cout << "Enter the size of the queue: ";

cin >> size;

Deque<int> obj(size);

while (true) {

cout << "1. Insert element from front\n";

cout << "2. Insert element from back\n";

cout << "3. Remove element from front\n";

cout << "4. Remove element from back\n";

cout << "5. Display\n";

cout << "6. Check if queue is empty\n";

cout << "7. Check if queue is full\n";

cout << "0. Exit\n";

cout << "Enter the choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the data to enqueue: ";

cin >> value;

obj.insertFront(value);

break;

case 2:

cout << "Enter the data to enqueue: ";

cin >> value;

obj.insertBack(value);

break;

case 3:

obj.removeFront();

break;

case 4:

obj.removeBack();

break;

case 5:

obj.display();

break;

case 6:

obj.isEmpty();

break;

case 7:

obj.isFull();

break;

case 0:

return 0;

default:

cout << "Invalid choice" << endl;

}

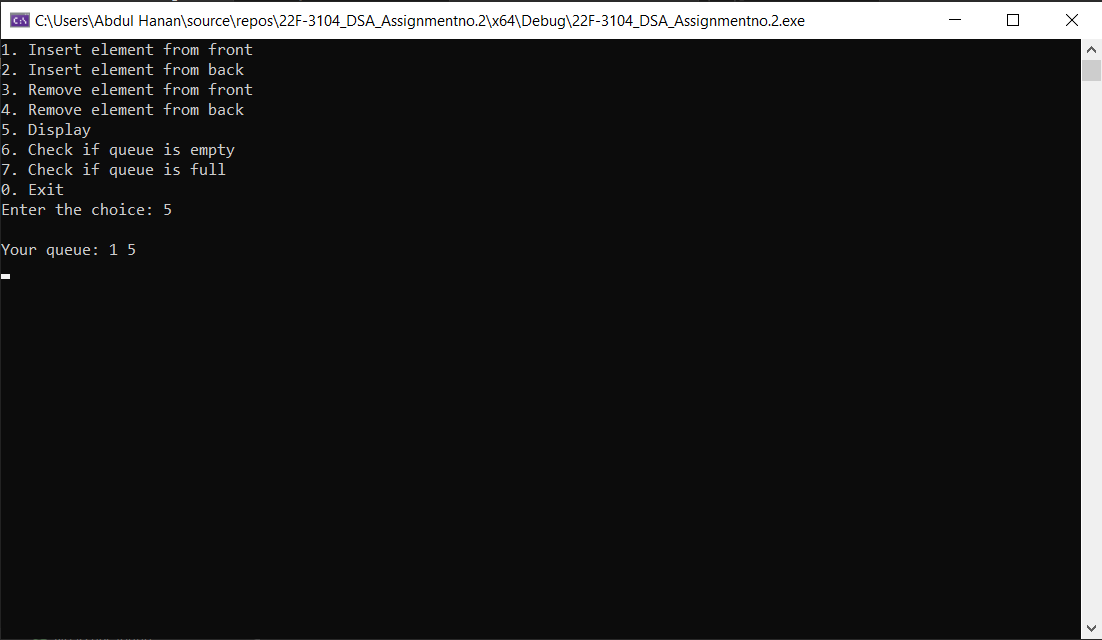
Sleep(2000);

system("cls");

}

}

**Screen Shot:**

****