INSTITUTE OF TECHNOLOGY AND MANAGEMENT
SKILLS UNIVERSITY,
KHARGHAR, NAVI MUMBAI

DATA STRUCTURES & ALGORITHMS
PROGRAMMING LAB

Prepared by:

Name of Student: Gaurang Jadhav

Roll No: (150096723009)

Batch: 2023-27

Dept. of CSE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INSTITUTE OF TECHNOLOGY AND MANAGEMENT
SKILLS UNIVERSITY,
KHARGHAR, NAVI MUMBAI

CERTIFICATE

This is to certify that Mr. / Ms. Gaurang Jadhav Roll No. 150096723009 Semester Two of B.Tech Computer Science & Engineering, ITM Skills University, Kharghar, Navi Mumbai , has completed the term work satisfactorily in subject Data Structures & Algorithms for the academic year 20 24 - 20 25 as prescribed in the curriculum.

Place: Mumbai

Date: 08-04-2024

Subject I/C **HOD**

Exp. No	List of Experiment	Date of Submission	Sign
1	Implement Array and write a menu driven program to perform all the operation on array elements	08-04-2024	
2	Implement Stack ADT using array.	08-04-2024	
3	Convert an Infix expression to Postfix expression using stack ADT.	08-04-2024	
4	Evaluate Postfix Expression using Stack ADT.	08-04-2024	
5	Implement Linear Queue ADT using array.	08-04-2024	
6	Implement Circular Queue ADT using array.	08-04-2024	
7	Implement Singly Linked List ADT.	08-04-2024	
8	Implement Circular Linked List ADT.	08-04-2024	
9	Implement Stack ADT using Linked List	08-04-2024	
10	Implement Linear Queue ADT using Linked List	08-04-2024	
11	Implement Binary Search Tree ADT using Linked List.	08-04-2024	
12	Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search	08-04-2024	
13	Implement Binary Search algorithm to search an element in an array	08-04-2024	
14	Implement Bubble sort algorithm to sort elements of an array in ascending and descending order	08-04-2024	

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 01

Title:

1. Implement Array and write a menu driven program to perform all the operations on array element

Theory:

This code defines a class Array to handle various operations on an array such as initialisation, traversal, insertion, deletion, searching, sorting, and reversal. The main function prompts the user to input the capacity of the array and its elements, then provides a menu-driven interface to perform different operations.

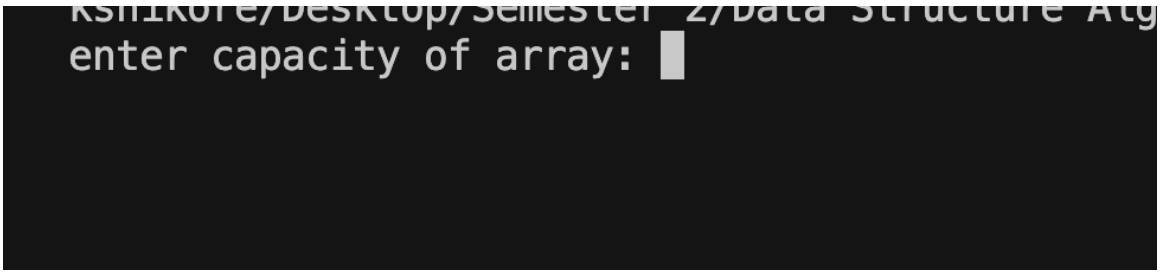
Code:

```
#include <iostream>
using namespace std;
class Array {
private:
    int capacity, size;
    int *arr;
public:
    void init(int cap) {
        capacity = cap;
        arr = new int[capacity];
        size = 0;
    }
    void traverse()  {
        cout << "Array elements: ";
        for (int i = 0; i < size; ++i)  {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    void insertBeginning(int element)  {
        if (size < capacity) {
            for (int i = size; i > 0; --i)  {
                arr[i] = arr[i - 1];
            }
            arr[0] = element;
            size++;
            cout << "element inserted at beginning!" << endl;
        }  else  {
            cout << "array is full" << endl;
        }
    }
    void insertEnd(int element)  {
        if (size < capacity) {
            arr[size++] = element;
            cout << "element inserted at end!" << endl;
        }  else {
            cout << "array is full" << endl;
        }
    }
    void insertIndex(int element, int index)  {
        if (index >= 0 && index <= size && size < capacity)  {
            for (int i = size; i > index; --i)  {
```

```
                arr[i] = arr[i - 1];
            }
            arr[index] = element;
            size++;
            cout << "element inserted at index " << index << endl;
        }  else  {
            cout << "invalid index or array is full" << endl;
        }
    }
    void deleteElement(int index)  {
        if (index >= 0 && index < size) {
            for (int i = index; i < size - 1; ++i) {
                arr[i] = arr[i + 1];
            }
            size--;
            cout << "element deleted!" << endl;
        }  else  {
            cout << "invalid index" << endl;
        }
    }
    int search(int element)  {
        for (int i = 0; i < size; ++i) {
            if (arr[i] == element) {
                return i;
            }
        }
        return -1;
    }
    void sort()  {
        for (int i = 0; i < size - 1; ++i)  {
            for (int j = 0; j < size - i - 1; ++j)  {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
        cout << "array sorted!" << endl;
    }
    void reverse() {
        for (int i = 0; i < size / 2; ++i)  {
            int temp = arr[i];
            arr[i] = arr[size - i - 1];
            arr[size - i - 1] = temp;
        }
        cout << "array reversed!" << endl;
    }
};
int main() {
    Array arr;
    int capacity;
    cout << "enter capacity of array: ";
    cin >> capacity;
    arr.init(capacity);
    cout << "enter elements for array: " << endl;
    for (int i = 0; i < capacity; ++i)  {
        int element;
        cout << "element " << i + 1 << ": ";
        cin >> element;
        arr.insertEnd(element);
    }
```

```
    int choice, element, index;
    do {
        cout << "1. traverse\n";
        cout << "2. insert at beginning\n";
        cout << "3. insert at end\n";
        cout << "4. insert at any index\n";
        cout << "5. delete\n";
        cout << "6. search\n";
        cout << "7. sort\n";
        cout << "8. reverse\n";
        cout << "9. exit\n";
        cout << "enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                arr.traverse();
                break;
            case 2:
                cout << "enter element to insert at beginning: ";
                cin >> element;
                arr.insertBeginning(element);
                break;
            case 3:
                cout << "enter element to insert at end: ";
                cin >> element;
                arr.insertEnd(element);
                break;
            case 4:
                cout << "enter element to insert: ";
                cin >> element;
                cout << "enter index no. to insert at: ";
                cin >> index;
                arr.insertIndex(element, index);
                break;
            case 5:
                cout << "enter index of element to delete: ";
                cin >> index;
                arr.deleteElement(index);
                break;
            case 6:
                cout << "enter element to search: ";
                cin >> element;
                index = arr.search(element);
                if (index != -1) {
                    cout << "element found at index " << index << endl;
                }  else {
                    cout << "element not found" << endl;
                }
                break;
            case 7:
                arr.sort();
                break;
            case 8:
                arr.reverse();
                break;
            case 9:
                cout << "exiting!\n";
                break;
            default:
                cout << "invalid choice" << endl;
        }
    } while (choice != 9);
    return 0   }
```

Output: (screenshot)



Test Case: Any two (screenshot)

```
enter capacity of array: 4
enter elements for array:
element 1: 1
element inserted at end!
element 2: 2
element inserted at end!
element 3: 3
element inserted at end!
element 4: 4
element inserted at end!
1. traverse
2. insert at beginning
3. insert at end
4. insert at any index
5. delete
6. search
7. sort
8. reverse
9. exit
enter your choice: 5
enter index of element to delete: 2
element deleted!
```

```
enter capacity of array: 4
enter elements for array:
element 1: 13
element inserted at end!
element 2: 9
element inserted at end!
element 3: 21
element inserted at end!
element 4: 54
element inserted at end!
1. traverse
2. insert at beginning
3. insert at end
4. insert at any index
5. delete
6. search
7. sort
8. reverse
9. exit
enter your choice: 6
enter element to search: 9
element found at index 1
```

Conclusion:

The code provides a comprehensive implementation for array manipulation, offering functionalities like insertion, deletion, searching, sorting, and reversal. It offers user-friendly interaction through a menu-driven interface, making it easy to use for array operations.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 02

Title:

2. Implement Stack ADT using array in CPP.

Theory:

This code implements a stack using a class Stack with functionalities like initialization, checking if the stack is empty or full, pushing elements onto the stack, popping elements from the stack, and peeking at the top element. The main function provides a menu-driven interface for users to interact with the stack.

Code:

```
#include <iostream>
using namespace std;
class Stack {
private:
    int capacity;
    int *arr;
    int top;
public:
    void init(int cap) {
        capacity = cap;
        arr = new int[capacity];
        top = -1;
    }
    bool isEmpty() {
        return top == -1;
    }
    bool isFull() {
        return top == capacity - 1;
    }
    void push(int element) {
        if (!isFull()) {
            arr[++top] = element;
            cout << element << " pushed in stack." << endl;
        } else {
            cout << "stack overflow" << endl;
        }
    }
    int pop() {
        if (!isEmpty()) {
            int element = arr[top--];
            cout << element << " popped from stack" << endl;
            return element;
        } else {
            cout << "stack underflow" << endl;
            return -1;
        }
    }
    int peek() {
```

```
        if (!isEmpty()) {
            return arr[top];
        } else {
            cout << "stack is empty" << endl;
            return -1;
        }
    }
};
int main() {
    Stack stack;
    int capacity;
    cout << "enter capacity of stack: ";
    cin >> capacity;
    stack.init(capacity);
    int num_elements;
    cout << "how many elements do you want to push in stack? ";
    cin >> num_elements;
    int element;
    for (int i = 0; i < num_elements; ++i) {
        cout << "enter element " << i + 1 << ": ";
        cin >> element;
        stack.push(element);
    }
    int choice;
    do {
        cout << "\n1. push\n";
        cout << "2. pop\n";
        cout << "3. peek\n";
        cout << "4. exit\n";
        cout << "enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "enter element to push in stack: ";
                cin >> element;
                stack.push(element);
                break;
            case 2:
                stack.pop();
                break;
            case 3:
                element = stack.peek();
                if (element != -1) {
                    cout << "top element of stack: " << element << endl;
                }
                break;
            case 4:
                cout << "exited!\n";
                break;
            default:
                cout << "invalid choice" << endl;
        }
    }
    while (choice != 4);

    return 0;
}
```

Output: (screenshot)

```
ck_array
enter capacity of stack:
```

Test Case: Any two (screenshot)

```
enter capacity of stack: 5
how many elements do you want to push in stack? 5
enter element 1: 32
32 pushed in stack.
enter element 2: 45
45 pushed in stack.
enter element 3: 12
12 pushed in stack.
enter element 4: 87
87 pushed in stack.
enter element 5: 6
6 pushed in stack.

1. push
2. pop
3. peek
4. exit
enter your choice: 2
6 popped from stack
```

```
enter capacity of stack: 5
how many elements do you want to push in stack? 4
enter element 1: 67
67 pushed in stack.
enter element 2: 19
19 pushed in stack.
enter element 3: 53
53 pushed in stack.
enter element 4: 41
41 pushed in stack.

1. push
2. pop
3. peek
4. exit
enter your choice: 1
enter element to push in stack: 32
32 pushed in stack.
```

Conclusion:

The code offers a basic implementation of a stack data structure with essential operations like push, pop, and peek. It provides a user-friendly menu interface for interacting with the stack, allowing users to push elements onto the stack, pop elements from it, and view the top element.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 03

Title:

3. Convert an Infix expression to Postfix expression using stack ADT.

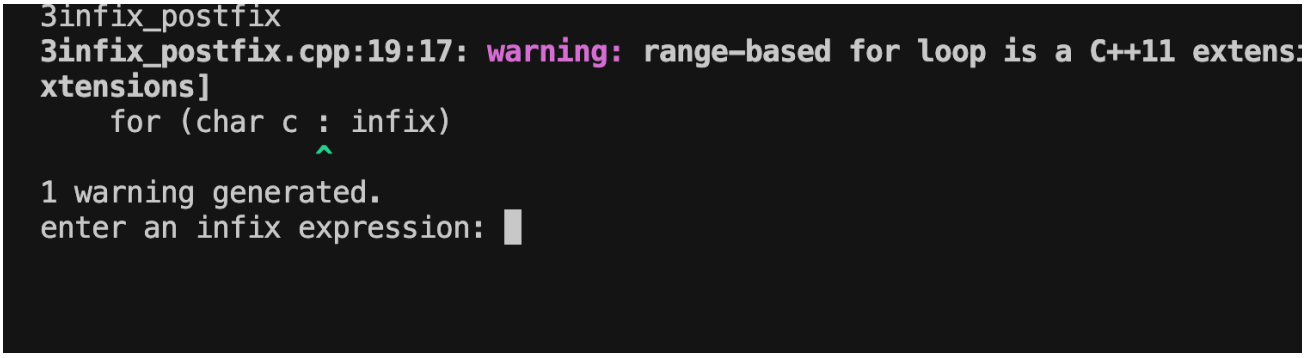
Theory:

This code converts an infix expression to a postfix expression using a stack. It reads an infix expression from the user, iterates through each character, and based on the precedence of operators and parentheses, constructs the corresponding postfix expression.

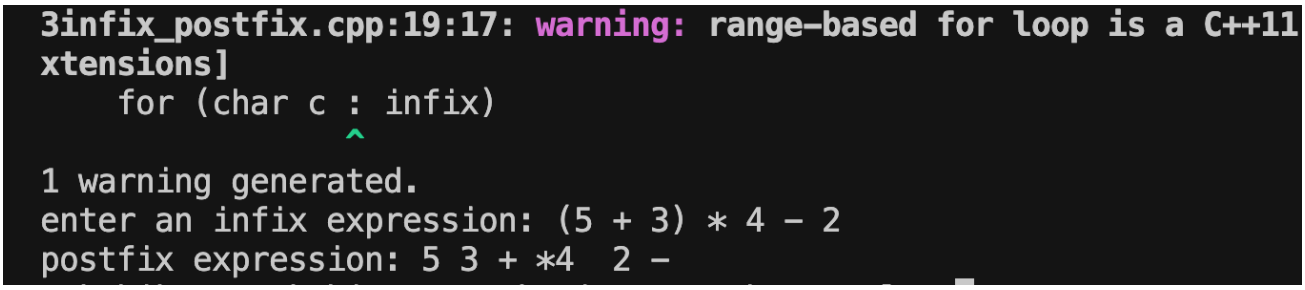
Code:

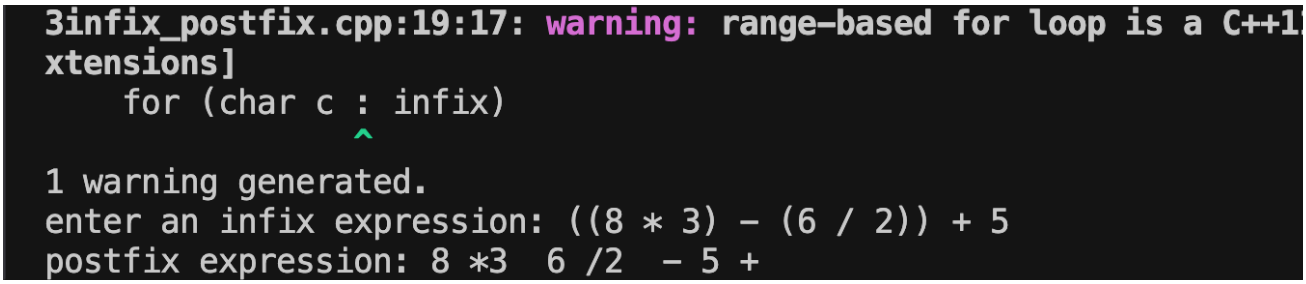
```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<char> s;
    string infix, postfix;
    cout << "enter an infix expression: ";
    getline(cin, infix);
    for (char c : infix) {
        if (isalnum(c)) {
            postfix += c;
        } else if (c == '(') {
            s.push(c);
        } else if (c == ')') {
            while (s.top() != '(') {
                postfix += s.top(); s.pop();
            }
            s.pop();
        } else {
            while (!s.empty() && s.top() != '(' && ((c == '+' || c
== '-') ? 1 : 2) <= ((s.top() == '+' || s.top() == '-') ? 1 : 2)) {
                postfix += s.top(); s.pop();
            }
            s.push(c);
        }
    }
    while (!s.empty()) {
        postfix += s.top(); s.pop();
    }
    cout << "postfix expression: " << postfix << endl;
    return 0;
}
```

Output: (screenshot)



Test Case: Any two (screenshot)





Conclusion:

The code efficiently converts infix expressions to postfix expressions using a stack-based approach, handling operands, operators, and parentheses while maintaining operator precedence. It provides a straightforward implementation for converting expressions, useful in various parsing and evaluation algorithms.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 04

Title:

4. Evaluate Postfix Expression using Stack ADT.

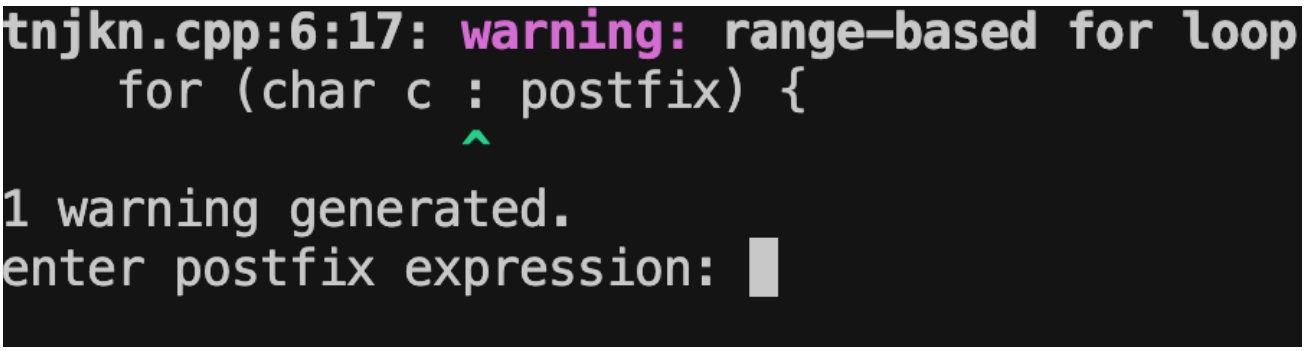
Theory:

This code evaluates a postfix expression by iterating through each character of the expression and using a stack to perform the necessary arithmetic operations. It pushes operands onto the stack and when encountering an operator, it pops the required number of operands from the stack, performs the operation, and pushes the result back onto the stack.

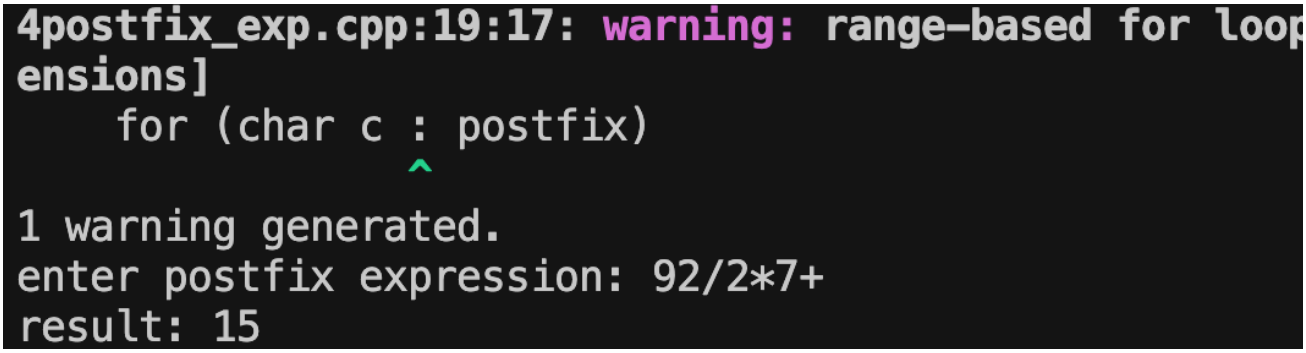
Code:

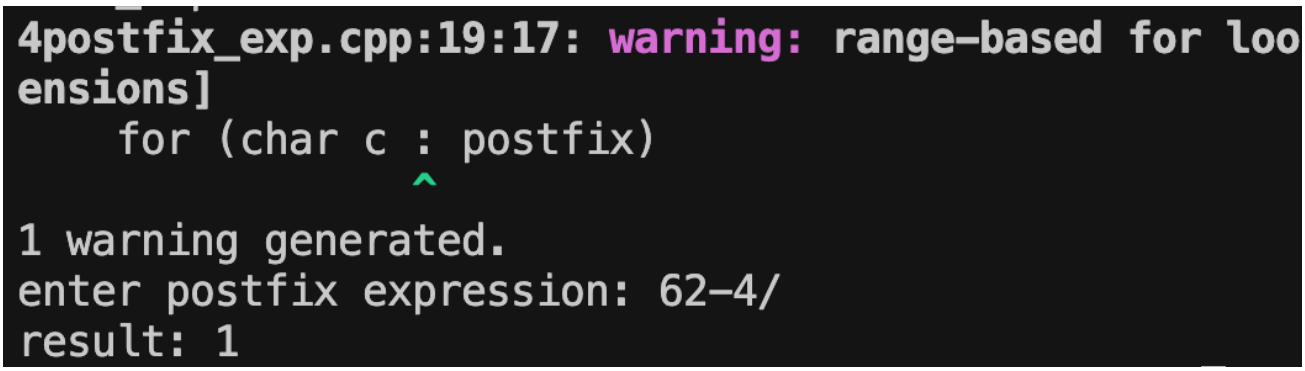
```
#include <iostream>
#include <stack>
using namespace std;
int evaluate(const string& postfix) {
    stack<int> s;
    for (char c : postfix) {
        if (isdigit(c)) {
            s.push(c - '0');
        } else {
            int operand2 = s.top(); s.pop();
            int operand1 = s.top(); s.pop();
            switch(c) {
                case '+': s.push(operand1 + operand2);
                break;
                case '-': s.push(operand1 - operand2);
                break;
                case '*': s.push(operand1 * operand2);
                break;
                case '/': s.push(operand1 / operand2);
                break;
            }
        }
    }
    return s.top();
}
int main() {
    string postfixexp;
    cout << "enter postfix expression: ";
    getline(cin, postfixexp);
    cout << "result: " << evaluate(postfixexp) << endl;
    return 0;
}
```

Output: (screenshot)



Test Case: Any two (screenshot)





Conclusion:

The code efficiently evaluates postfix expressions using a stack-based approach, handling arithmetic operations such as addition, subtraction, multiplication, and division. It provides a straightforward implementation for expression evaluation, useful in various mathematical and computing applications.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 05

Title:

5. Implement Linear Queue ADT using array.

Theory:

This code implements a queue data structure using an array. It provides functionalities to enqueue elements into the queue, dequeue elements from the queue, and display the elements currently in the queue. The main function offers a menu-driven interface for users to interact with the queue.

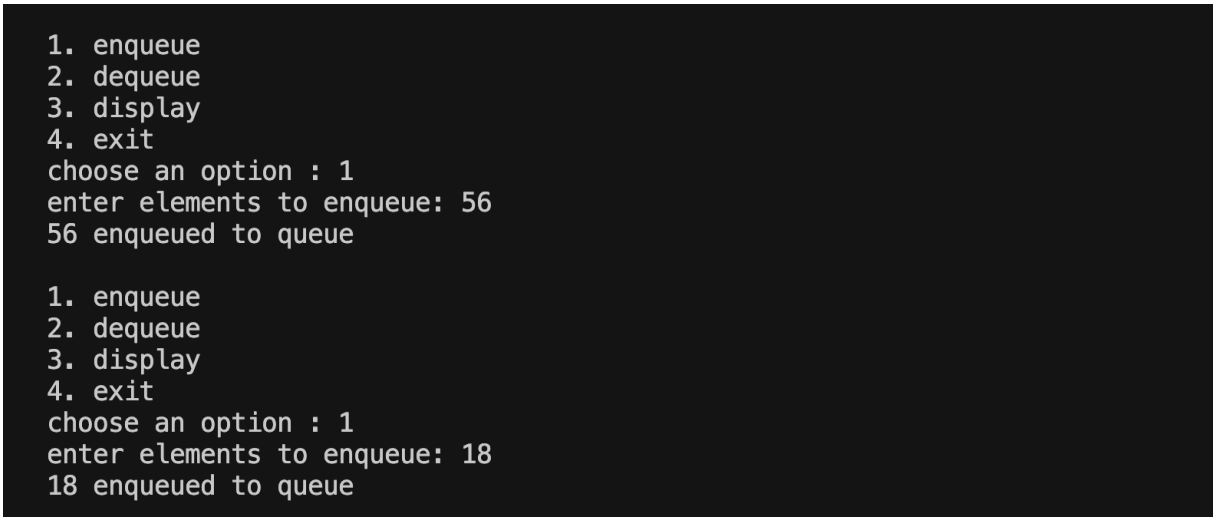
Code:

```
#include <iostream>
using namespace std;
const int MAX_SIZE = 100;
class Queue {
private:
    int front, rear;
    int arr[MAX_SIZE];
public:
    Queue() {
        front = -1;
        rear = -1;
    }
    bool isEmpty() {
        return (front == -1 && rear == -1);
    }
    bool isFull() {
        return (rear == MAX_SIZE - 1);
    }
    void enqueue(int data) {
        if (isFull()) {
            cout << "queue is full" << endl;
            return;
        } else if (isEmpty()) {
            front = rear = 0;
        } else {
            rear++;
        }
        arr[rear] = data;
        cout << data << " enqueued to queue" << endl;
    }
    void dequeue() {
        if (isEmpty()) {
            cout << "queue is empty" << endl;
            return;
        } else if (front == rear) {
```

```
            cout << arr[front] << " dequeued from queue" << endl;
            front = rear = -1;
        } else {
            cout << arr[front] << " dequeued from queue." << endl;
            front++;
        }
    }
    void display() {
        if (isEmpty()) {
            cout << "queue is empty" << endl;
            return;
        }
        cout << "elements in the queue: ";
        for (int i = front; i <= rear; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};
int main() {
    Queue q;
    int choice, data;
    do {
        cout << "\n1. enqueue\n";
        cout << "2. dequeue\n";
        cout << "3. display\n";
        cout << "4. exit\n";
        cout << "choose an option : ";
        cin >> choice;
        switch(choice) {
            case 1:
                cout << "enter elements to enqueue: ";
                cin >> data;
                q.enqueue(data);
                break;
            case 2:
                q.dequeue();
                break;
            case 3:
                q.display();
                break;
            case 4:
                cout << "exited!" << endl;
                break;
            default:
                cout << "invalid choice." << endl;
        }
    }
     while (choice != 4);
    return 0;
}
```

Output: (screenshot)

Test Case: Any two (screenshot)



Conclusion:

The code offers a basic implementation of a queue using an array, providing essential operations such as enqueue, dequeue, and display. It offers a user-friendly menu interface for interacting with the queue, allowing users to enqueue elements, dequeue elements, and view the elements currently in the queue.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 06

Title:

6. Implement Circular Queue ADT using array.

Theory:

This code implements a queue data structure using an array with circular buffering to optimize space usage. It provides functionalities to enqueue elements into the queue, dequeue elements from the queue, and display the elements currently in the queue. The circular buffering technique ensures efficient utilization of the array.

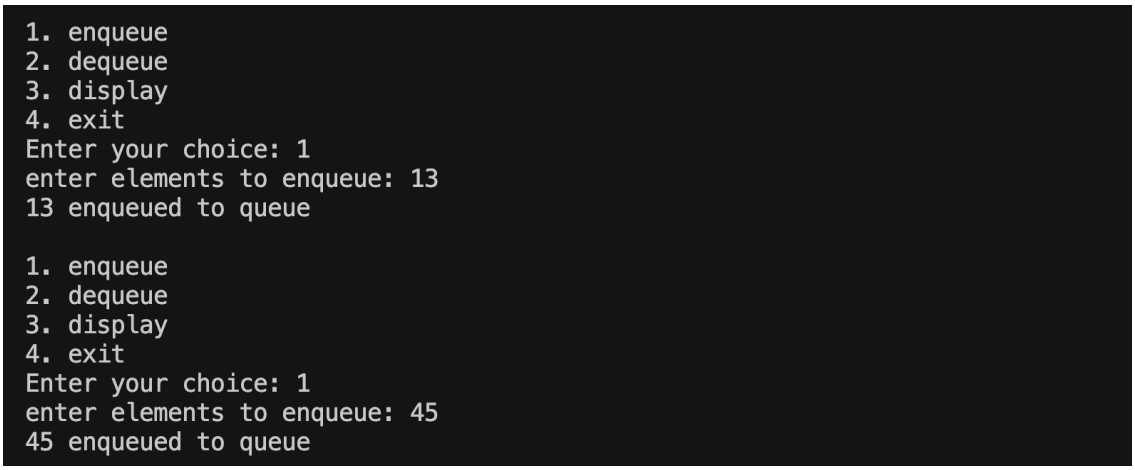
Code:

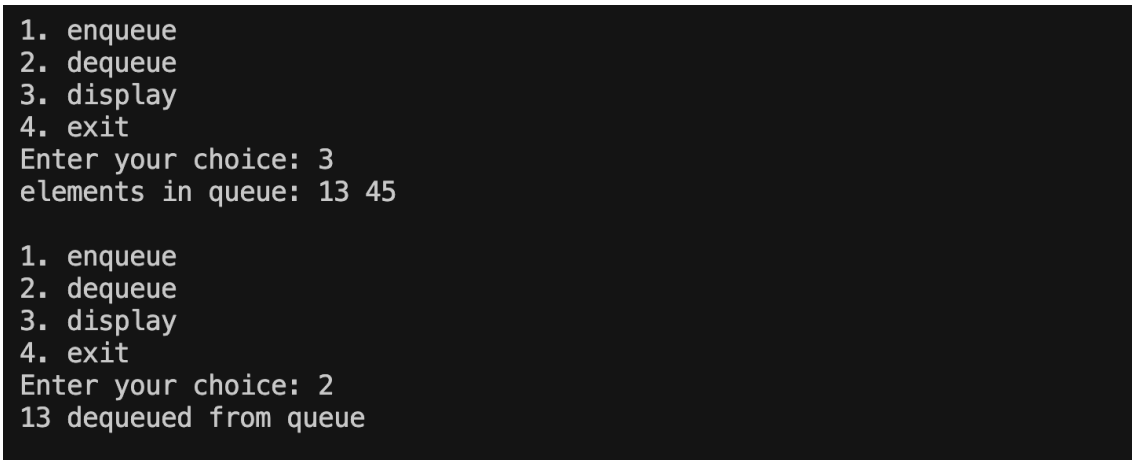
```
#include <iostream>
using namespace std;
const int MAX_SIZE = 100;
class Queue {
private:
    int front, rear;
    int arr[MAX_SIZE];
public:
    Queue() {
        front = rear = -1;
    }
    bool isEmpty() {
        return (front == -1);
    }
    bool isFull() {
        return ((front == 0 && rear == MAX_SIZE - 1) || (front == rear + 1));
    }
    void enqueue(int data) {
        if (isFull()) {
            cout << "queue is full" << endl;
            return;
        } else if (isEmpty()) {
            front = rear = 0;
        } else if (rear == MAX_SIZE - 1) {
            rear = 0;
        } else {
            rear++;
        }
        arr[rear] = data;
        cout << data << " enqueued to queue" << endl;
    }
    void dequeue() {
        if (isEmpty()) {
            cout << "queue is empty" << endl;
            return;
        } else if (front == rear) {
            cout << arr[front] << " dequeued from queue" << endl;
            front = rear = -1;
        } else if (front == MAX_SIZE - 1) {
            cout << arr[front] << " dequeued from queue" << endl;
```

```
            front = 0;
        } else {
            cout << arr[front] << " dequeued from queue" << endl;
            front++;
        }
    }
    void display() {
        if (isEmpty()) {
            cout << "queue is empty" << endl;
            return;
        }
        cout << "elements in queue: ";
        if (rear >= front) {
            for (int i = front; i <= rear; i++) {
                cout << arr[i] << " ";
            }
        } else {
            for (int i = front; i < MAX_SIZE; i++) {
                cout << arr[i] << " ";
            }
            for (int i = 0; i <= rear; i++) {
                cout << arr[i] << " ";
            }
        }
        cout << endl;
    }
};
int main() {
    Queue q;
    int choice, data;
    do {
        cout << "\n1. enqueue\n";
        cout << "2. dequeue\n";
        cout << "3. display\n";
        cout << "4. exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice) {
            case 1:
                cout << "enter elements to enqueue: ";
                cin >> data;
                q.enqueue(data);
                break;
            case 2:
                q.dequeue();
                break;
            case 3:
                q.display();
                break;
            case 4:
                cout << "exited!" << endl;
                break;
            default:
                cout << "invalid choice" << endl;
        }
    } while (choice != 4);
    return 0;
}
```

Output: (screenshot)

Test Case: Any two (screenshot)





Conclusion:

The code offers an optimized implementation of a queue using an array with circular buffering, providing essential operations such as enqueue, dequeue, and display. It offers a user-friendly menu interface for interacting with the queue, allowing users to enqueue elements, dequeue elements, and view the elements currently in the queue.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 07

Title:

7. Implement Singly Linked List ADT.

Theory:

This code implements a singly linked list data structure with functionalities to append elements to the list, display the elements in the list, and clear the list. It utilizes a Node class to represent individual elements and a singlylist class to manage the list operations. The main function provides a menu-driven interface for users to interact with the list.

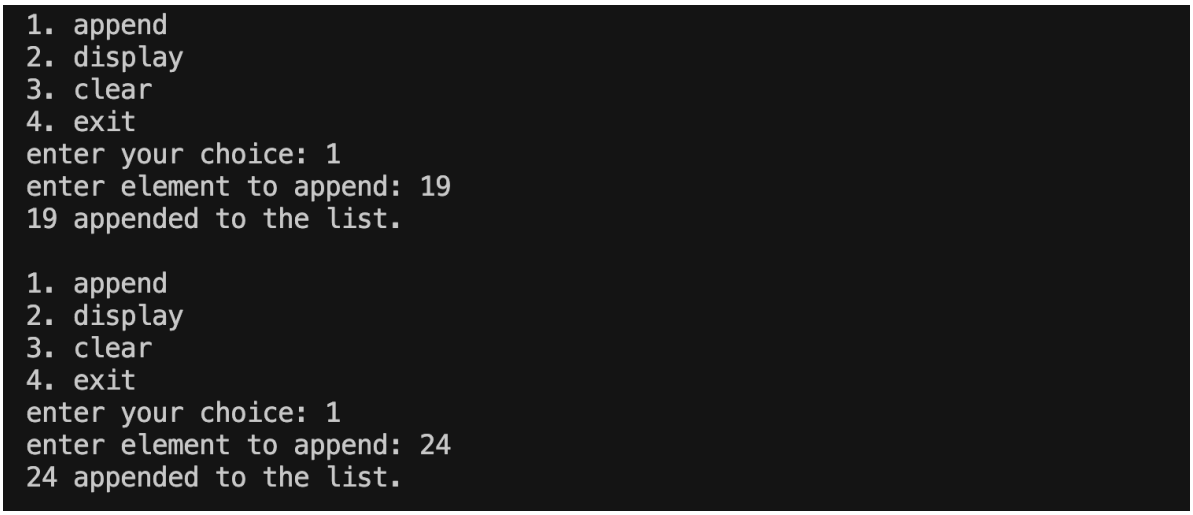
Code:

```
#include <iostream>
using namespace std;
class Node {
public:
    int data;
    Node* next;
    Node(int value) {
        data = value;
        next = nullptr;
    }
};
class singlylist {
private:
    Node* head;
    Node* tail;
public:
    singlylist() {
        head = nullptr;
        tail = nullptr;
    }
    void append(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
        cout << value << " appended to the list." << endl;
    }
    void display() {
        if (head == nullptr) {
            cout << "list is empty" << endl;
            return;
```

```
        }
        cout << "elements in list: ";
        Node* current = head;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
    void clear() {
        while (head != nullptr) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
        tail = nullptr;
        cout << "list cleared." << endl;
    }
};
int main() {
    singlylist list;
    int choice, data;
    do {
        cout << "\n1. append\n";
        cout << "2. display\n";
        cout << "3. clear\n";
        cout << "4. exit\n";
        cout << "enter your choice: ";
        cin >> choice;
        switch(choice) {
            case 1:
                cout << "enter element to append: ";
                cin >> data;
                list.append(data);
                break;
            case 2:
                list.display();
                break;
            case 3:
                list.clear();
                break;
            case 4:
                cout << "exited!" << endl;
                break;
            default:
                cout << "invalid choice" << endl;
        }
    } while (choice != 4);
    return 0;
}
```

Output: (screenshot)

Test Case: Any two (screenshot)



Conclusion:

The code offers a basic implementation of a singly linked list, allowing users to append elements to the list, display the elements currently in the list, and clear the list. It provides a user-friendly menu interface for interacting with the list, making it easy to perform operations on the list.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 08

Title:

8. Implement Circular Linked List ADT.

Theory:

This code implements a circular singly linked list data structure with functionalities to append elements to the list, display the elements in the list, and clear the list. It utilizes a Node class to represent individual elements and a circularlist class to manage the list operations. The main function provides a menu-driven interface for users to interact with the list.

Code:

```
#include <iostream>
using namespace std;
class Node {
public:
    int data;
    Node* next;
    Node(int value) {
        data = value;
        next = nullptr;
    }
};
class circularlist {
private:
    Node* head;
public:
    circularlist() {
        head = nullptr;
    }
    void append(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            head->next = head;
        } else {
            Node* current = head;
            while (current->next != head) {
                current = current->next;
            }
            current->next = newNode;
            newNode->next = head;
        }
        cout << value << " appended to the list." << endl;
    }
    void display() {
        if (head == nullptr) {
            cout << "list is empty." << endl;
            return;
```

```
        }
        cout << "elements in list: ";
        Node* current = head;
        do {
            cout << current->data << " ";
            current = current->next;
        }
        while (current != head);
        cout << endl;
    }
    void clear() {
        if (head == nullptr) {
            cout << "list is empty." << endl;
            return;
        }
        Node* current = head;
        while (current->next != head) {
            Node* temp = current;
            current = current->next;
            delete temp;
        }
        delete current;
        head = nullptr;
        cout << "list cleared." << endl;
    }
};
int main() {
    circularlist list;
    int choice, data;
    do {
        cout << "\n1. append\n";
        cout << "2. display\n";
        cout << "3. clear\n";
        cout << "4. exit\n";
        cout << "enter your choice: ";
        cin >> choice;
        switch(choice) {
            case 1:
                cout << "enter element to append: ";
                cin >> data;
                list.append(data);
                break;
            case 2:
                list.display();
                break;
            case 3:
                list.clear();
                break;
            case 4:
                cout << "exited!" << endl;
                break;
            default:
                cout << "invalid choice" << endl;
        }
    } while (choice != 4);
    return 0;
}
```

Output: (screenshot)

Test Case: Any two (screenshot)

```
1. append
2. display
3. clear
4. exit
enter your choice: 1
enter element to append: 25
25 appended to the list.

1. append
2. display
3. clear
4. exit
enter your choice: 1
enter element to append: 16
16 appended to the list.
```

```
1. append
2. display
3. clear
4. exit
enter your choice: 2
elements in list: 25 16

1. append
2. display
3. clear
4. exit
enter your choice: 3
list cleared.
```

Conclusion:

The code offers an implementation of a circular singly linked list, allowing users to append elements to the list, display the elements currently in the list, and clear the list. It utilizes circular linking to ensure that the last node points back to the head, creating a circular structure. The provided menu interface makes it easy for users to perform operations on the list.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 09

Title:

9. Implement Stack ADT using Linked List

Theory:

This code implements a stack data structure using a singly linked list. It provides functionalities to push elements onto the stack, pop elements from the stack, display the elements in the stack, and check if the stack is empty. The main function offers a menu-driven interface for users to interact with the stack.

Code:

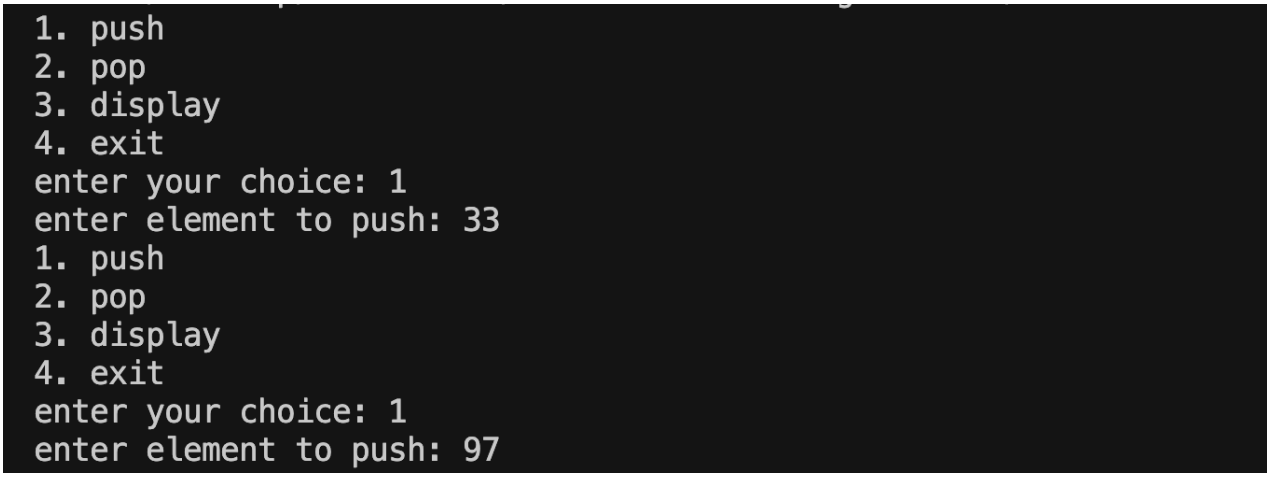
```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
class Stack {
private:
    Node* top;
public:
    Stack() {
        top = nullptr;
    }
    void push(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = top;
        top = newNode;
    }
    int pop() {
        if (isEmpty()) {
            cout << "stack is empty\n";
            return -1;
        }
        int poppedvalue = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return poppedvalue;
    }
    bool isEmpty() {
        return top == nullptr;
    }
```

```
    void display() {
        if (isEmpty()) {
            cout << "stack is empty\n";
            return;
        }
        cout << "stack: ";
        Node* current = top;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};
int main() {
    Stack stack;
    char choice;
    int value;
    do {
        cout << "1. push\n";
        cout << "2. pop\n";
        cout << "3. display\n";
        cout << "4. exit\n";
        cout << "enter your choice: ";
        cin >> choice;

        switch(choice) {
            case '1':
                cout << "enter element to push: ";
                cin >> value;
                stack.push(value);
                break;
            case '2':
                if (!stack.isEmpty())
                    cout << "popped element: " << stack.pop() <<
endl;
                break;
            case '3':
                stack.display();
                break;
            case '4':
                cout << "exited!\n";
                break;
            default:
                cout << "invalid choice\n";
        }
    } while(choice != '4');
    return 0;
}
```

Output: (screenshot)

Test Case: Any two (screenshot)



Conclusion:

The code offers a flexible implementation of a stack using a singly linked list, providing essential operations such as push, pop, and display. It utilizes dynamic memory allocation to manage nodes, allowing for efficient memory usage. The provided menu interface makes it easy for users to perform operations on the stack.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 10

Title:

10. Implement Linear Queue ADT using Linked List

Theory:

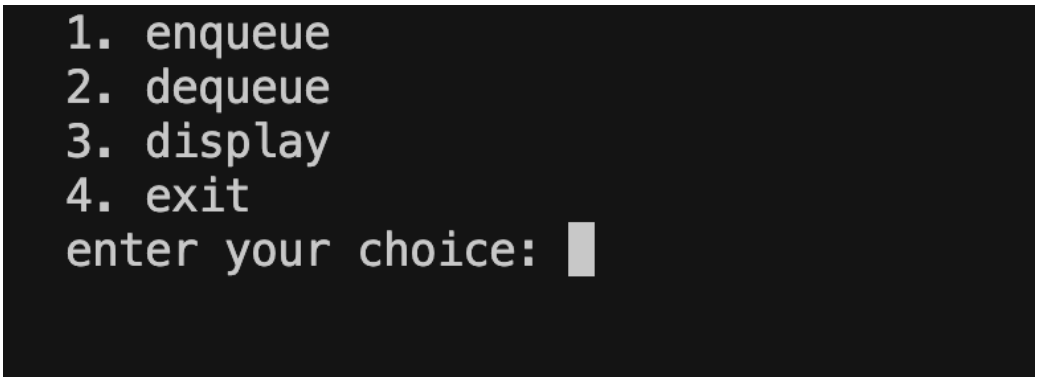
This code implements a queue data structure using a singly linked list. It provides functionalities to enqueue elements into the queue, dequeue elements from the queue, display the elements in the queue, and check if the queue is empty. The main function offers a menu-driven interface for users to interact with the queue.

Code:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
class Queue {
private:
    Node* front;
    Node* rear;
public:
    Queue() {
        front = nullptr;
        rear = nullptr;
    }
    void enqueue(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = nullptr;
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
    }
    int dequeue() {
        if (isEmpty()) {
            cout << "queue is empty\n";
            return -1;
        }
        int dequeuedvalue = front->data;
        Node* temp = front;
        front = front->next;
        if (front == nullptr) {
            rear = nullptr;
        }
        delete temp;
```

```
        return dequeuedvalue;
    }
    bool isEmpty() {
        return front == nullptr;
    }
    void display() {
        if (isEmpty()) {
            cout << "queue is empty\n";
            return;
        }
        cout << "queue: ";
        Node* current = front;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};
int main() {
    Queue queue;
    char choice;
    int value;
    do {
        cout << "1. enqueue\n";
        cout << "2. dequeue\n";
        cout << "3. display\n";
        cout << "4. exit\n";
        cout << "enter your choice: ";
        cin >> choice;
        switch(choice) {
            case '1':
                cout << "enter element to enqueue: ";
                cin >> value;
                queue.enqueue(value);
                break;
            case '2':
                if (!queue.isEmpty())
                    cout << "dequeued value: " << queue.dequeue() << endl;
                break;
            case '3':
                queue.display();
                break;
            case '4':
                cout << "exited!\n";
                break;
            default:
                cout << "invalid choice\n";
        }
    }
    while(choice != '4');
    return 0;
}
```

Output: (screenshot)



Test Case: Any two (screenshot)

```
1. enqueue
2. dequeue
3. display
4. exit
enter your choice: 1
enter element to enqueue: 39
1. enqueue
2. dequeue
3. display
4. exit
enter your choice: 1
enter element to enqueue: 60
1. enqueue
2. dequeue
3. display
4. exit
enter your choice: 1
enter element to enqueue: 74
1. enqueue
2. dequeue
3. display
4. exit
enter your choice: 3
queue: 39 60 74
1. enqueue
2. dequeue
3. display
4. exit
enter your choice: 2
dequeued value: 39
```

Conclusion:

The code offers a flexible implementation of a queue using a singly linked list, providing essential operations such as enqueue, dequeue, and display. It utilizes dynamic memory allocation to manage nodes, allowing for efficient memory usage. The provided menu interface makes it easy for users to perform operations on the queue.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 11

Title:

11. Implement Binary Search Tree ADT using Linked List.

Theory:

This code implements a binary search tree (BST) data structure with functionalities to insert elements into the tree, search for elements in the tree, and display the elements of the tree in sorted order. It utilizes a recursive approach for insertion and traversal. The main function offers a menu-driven interface for users to interact with the BST.

Code:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
};
class bst {
private:
    Node* root;
    Node* createnode(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->left = nullptr;
        newNode->right = nullptr;
        return newNode;
    }
    Node* insert(Node* root, int value) {
        if (root == nullptr) {
            return createnode(value);
        }
        if (value < root->data) {
            root->left = insert(root->left, value);
        } else if (value > root->data) {
            root->right = insert(root->right, value);
        }
        return root;
    }
    void traversal(Node* root) {
        if (root != nullptr) {
            traversal(root->left);
            cout << root->data << " ";
            traversal(root->right);
        }
    }
public:
    bst() {
        root = nullptr;
    }
    void insert(int value) {
```

```
        root = insert(root, value);
    }
    bool search(int value) {
        Node* current = root;
        while (current != nullptr) {
            if (value == current->data) {
                return true;
            } else if (value < current->data) {
                current = current->left;
            } else {
                current = current->right;
            }
        }
        return false;
    }
    void display() {
        if (root == nullptr) {
            cout << "binary search tree is empty\n";
        } else {
            cout << "binary search tree: ";
            traversal(root);
            cout << endl;
        }
    }
};
int main() {
    bst bst;
    char choice;
    int value;
    do {
        cout << "1. insert\n";
        cout << "2. search\n";
        cout << "3. display\n";
        cout << "4. exit\n";
        cout << "enter your choice: ";
        cin >> choice;
        switch(choice) {
            case '1':
                cout << "enter element to insert: ";
                cin >> value;
                bst.insert(value);
                break;
            case '2':
                cout << "enter element to search: ";
                cin >> value;
                if (bst.search(value)) {
                    cout << "found in binary search tree.\n";
                } else {
                    cout << "not found in binary search tree.\n";
                }
                break;
            case '3':
                bst.display();
                break;
            case '4':
                cout << "exited!\n";
                break;
            default:
                cout << "invalid choice\n";
        }
    } while(choice != '4');
    return 0;
}
```

Output: (screenshot)

Test Case: Any two (screenshot)

```
1. insert
2. search
3. display
4. exit
enter your choice: 1
enter element to insert: 13
1. insert
2. search
3. display
4. exit
enter your choice: 1
enter element to insert: 53
1. insert
2. search
3. display
4. exit
enter your choice: 1
enter element to insert: 22
1. insert
2. search
3. display
4. exit
enter your choice: 2
enter element to search: 53
found in binary search tree.
1. insert
2. search
3. display
4. exit
enter your choice: 3
binary search tree: 13 22 53
```

Conclusion:

The code provides a flexible implementation of a binary search tree, allowing users to insert elements, search for elements, and display the elements in sorted order. It utilizes a recursive approach for insertion and traversal, ensuring efficient operations on the tree. The provided menu interface makes it easy for users to perform operations on the binary search tree.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 12(a)

Title:

12. Implement Graph Traversal techniques: a) Depth First Search

Theory:

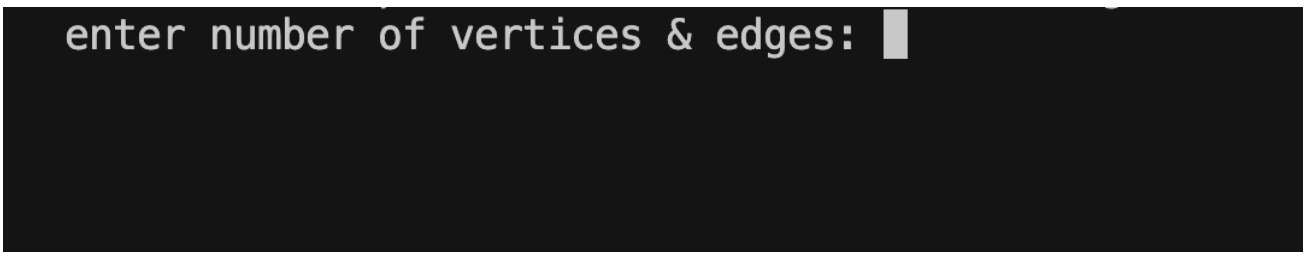
This code implements depth-first search (DFS) traversal on an undirected graph represented using an adjacency matrix. It constructs the graph by adding edges between vertices, then performs DFS traversal starting from a specified vertex. DFS explores as far as possible along each branch before backtracking.

Code:

```
#include <iostream>
#include <stack>
using namespace std;
const int MAX_VERTICES = 100;
class Graph {
    int V;
    int** adj;
public:
    Graph(int V) {
        this->V = V;
        adj = new int*[V];
        for (int i = 0; i < V; ++i) {
            adj[i] = new int[V];
            for (int j = 0; j < V; ++j) {
                adj[i][j] = 0;
            }
        }
    }
    void addEdge(int v, int w) {
        adj[v][w] = 1;
        adj[w][v] = 1;
    }
    void DFS(int v) {
        bool* visited = new bool[V];
        for (int i = 0; i < V; ++i)
            visited[i] = false;
        stack<int> stack;
        visited[v] = true;
        stack.push(v);
        while (!stack.empty()) {
```

```
            v = stack.top();
            stack.pop();
            cout << v << " ";
            for (int i = 0; i < V; ++i) {
                if (adj[v][i] && !visited[i]) {
                    visited[i] = true;
                    stack.push(i);
                }
            }
        }
        delete[] visited;
    }
};
int main() {
    int V, E;
    cout << "enter number of vertices & edges: ";
    cin >> V >> E;
    Graph g(V);
    cout << "enter edges:" << endl;
    for (int i = 0; i < E; ++i) {
        int v, w;
        cin >> v >> w;
        g.addEdge(v, w);
    }
    int start_vertex;
    cout << "enter starting vertex: ";
    cin >> start_vertex;
    cout << "Depth First Traversal: ";
    g.DFS(start_vertex);
    return 0;
}
```

Output: (screenshot)



Test Case: Any two (screenshot)

```
enter number of vertices & edges: 5 5
enter edges:
0 1
0 2
1 3
2 4
3 4
enter starting vertex: 0
Depth First Traversal: 0 2 4 3 1 %
```

```
enter number of vertices & edges: 4 3
enter edges:
0 1
0 2
1 3
enter starting vertex: 1
Depth First Traversal: 1 3 0 2 %
```

Conclusion:

The code efficiently performs DFS traversal on an undirected graph using an adjacency matrix representation. It constructs the graph by adding edges between vertices and then executes DFS from a specified starting vertex. The DFS algorithm explores each vertex and its adjacent vertices in depth-first manner, ensuring that all reachable vertices are visited. Overall, the code provides a clear and effective implementation of DFS traversal on an undirected graph.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 12(b)

Title:

12. Implement Graph Traversal techniques: b) Breadth First Search

Theory:

This code implements breadth-first search (BFS) traversal on an undirected graph using an adjacency matrix, starting from a specified vertex. It marks visited vertices and explores adjacent vertices iteratively, maintaining a queue for traversal.

Code:

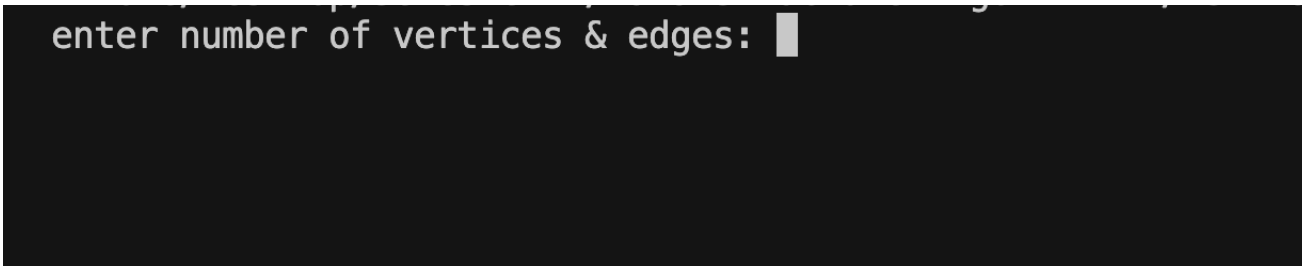
```
#include <iostream>
#include <queue>

using namespace std;
const int MAX_VERTICES = 100;
class Graph
 {
    int V;
    int** adj;

public:
    Graph(int V) {
        this->V = V;
        adj = new int*[V];
        for (int i = 0; i < V; ++i) {
            adj[i] = new int[V];
            for (int j = 0; j < V; ++j) {
                adj[i][j] = 0;
            }
        }
    }
    void addEdge(int v, int w) {
        adj[v][w] = 1;
        adj[w][v] = 1;
    }
    void BFS(int start) {
        bool* visited = new bool[V];
        for (int i = 0; i < V; ++i)
            visited[i] = false;
        queue<int> q;
        visited[start] = true;
        q.push(start);
        while (!q.empty()) {
```

```
            int v = q.front();
            q.pop();
            cout << v << " ";
             for (int i = 0; i < V; ++i) {
                if (adj[v][i] && !visited[i]) {
                    visited[i] = true;
                    q.push(i);
                }
            }
        }
        delete[] visited;
    }
};
int main()
 {
    int V, E;
    cout << "enter number of vertices & edges: ";
    cin >> V >> E;
    Graph g(V);
    cout << "enter edges: " << endl;
    for (int i = 0; i < E; ++i) {
        int v, w;
        cin >> v >> w;
        g.addEdge(v, w);
    }
    int start_vertex;
    cout << "enter starting vertex: ";
    cin >> start_vertex;
    cout << "Breadth First Traversal: ";
    g.BFS(start_vertex);
    return 0;
}
```

Output: (screenshot)



Test Case: Any two (screenshot)

```
enter number of vertices & edges: 3 2
enter edges:
0 1
1 2
enter starting vertex: 0
Breadth First Traversal: 0 1 2 %
```

```
enter number of vertices & edges: 4 3
enter edges:
0 1
1 2
1 3
enter starting vertex: 3
Breadth First Traversal: 3 1 0 2 %
```

```
enter number of vertices & edges: 5 4
enter edges:
0 1
1 2
2 3
3 4
enter starting vertex: 0
Breadth First Traversal: 0 1 2 3 4 %
```

Conclusion:

In conclusion, the code efficiently performs BFS traversal on an undirected graph represented using an adjacency matrix, demonstrating a fundamental graph traversal algorithm for exploring connected components in a graph.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 13

Title:

13. Implement Binary Search algorithm to search an element in an array

Theory:

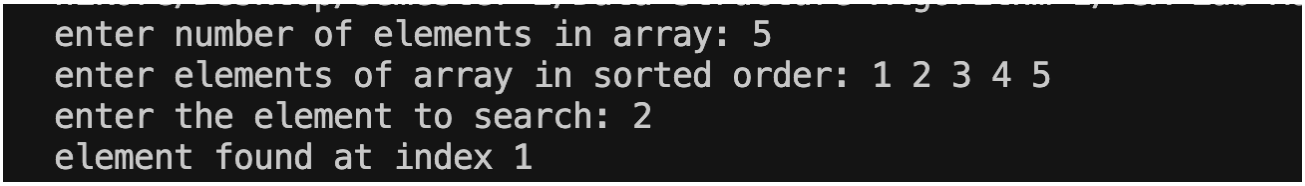
This code implements the binary search algorithm, which efficiently searches for a target element in a sorted array. It divides the array into halves repeatedly until the target element is found or the search interval becomes empty. By comparing the target with the middle element of the array, it determines whether to continue the search in the left or right subarray, effectively reducing the search space with each iteration.

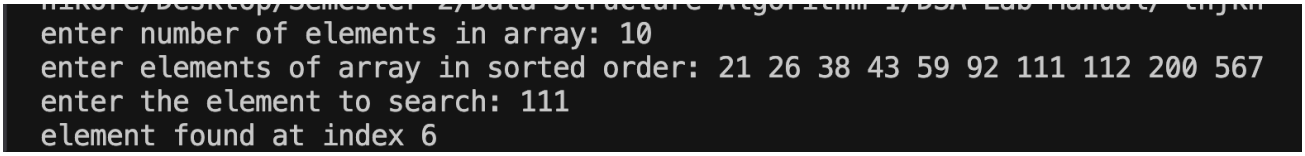
Code:

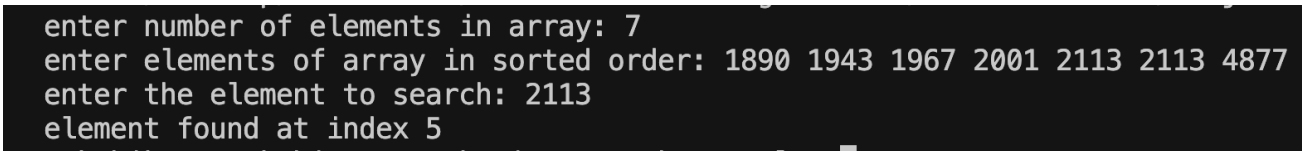
```
#include <iostream>
using namespace std;
int bst(int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)
            return mid;
        if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
int main() {
    int n;
    cout << "enter number of elements in array: ";
    cin >> n;
    int arr[n];
    cout << "enter elements of array in sorted order: ";
    for (int i = 0; i < n; ++i)
        cin >> arr[i];
    int target;
    cout << "enter the element to search: ";
    cin >> target;
    int index = bst(arr, 0, n - 1, target);
    if (index != -1)
        cout << "element found at index " << index << endl;
    else
        cout << "element not found" << endl;
    return 0;
}
```

Output: (screenshot)

Test Case: Any two (screenshot)







Conclusion:

The binary search algorithm showcased in the code offers a highly efficient approach to find a target element in a sorted array. With a time complexity of O(log n), where n is the number of elements in the array, it provides a significant improvement over linear search, particularly for large datasets.

Name of Student: Gaurang Jadhav

Roll Number: 150096723009

Experiment No: 14

Title:

14. Implement Bubble sort algorithm to sort elements of an array in ascending and descending order.

Theory:

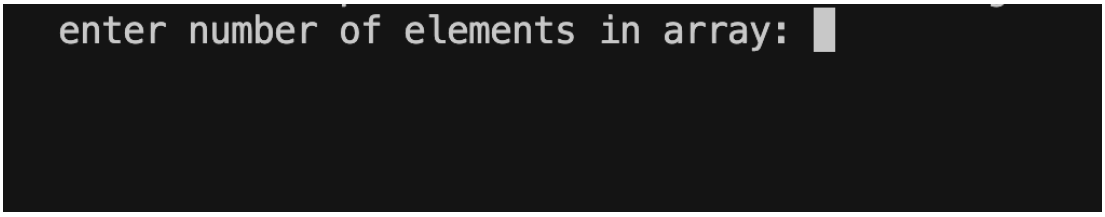
This code demonstrates two sorting functions: ascendingsort and descendingsort. Both functions implement the bubble sort algorithm to sort an array of integers in ascending and descending order, respectively. Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the array is sorted.

Code:

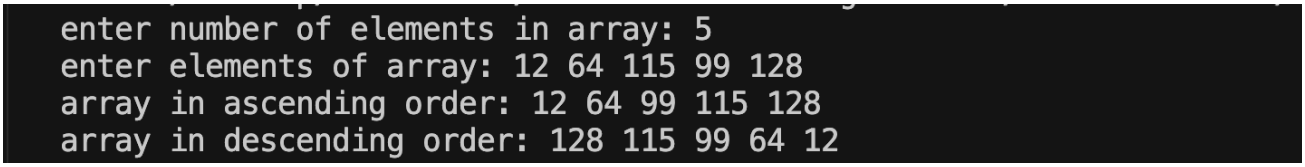
```
#include <iostream>
using namespace std;
void ascendingsort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
void descendingsort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] < arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
int main() {
    int n;
    cout << "enter number of elements in array: ";
    cin >> n;
    int arr[n];
    cout << "enter elements of array: ";
    for (int i = 0; i < n; ++i)
        cin >> arr[i];
```

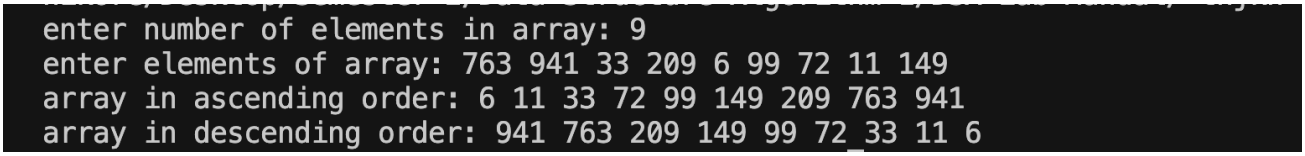
```
    ascendingsort(arr, n);
    cout << "array in ascending order: ";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
    descendingsort(arr, n);
    cout << "array in descending order: ";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output: (screenshot)



Test Case: Any two (screenshot)





Conclusion:

The bubble sort algorithm presented in the code provides a simple approach to sorting elements in an array. However, it has a time complexity of O(n^2), making it inefficient for large datasets compared to more efficient sorting algorithms like merge sort or quicksort.