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

DATA STRUCTURES & ALGORITHMS PROGRAMMING LAB

Prepared by:

Name of Student : Prem Thatikonda

Roll No: (06)

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. ____ Roll No. _____________ Semester _____ of B.Tech Computer Science & Engineering, ITM Skills University, Kharghar, Navi Mumbai , has completed the term work satisfactorily in subject ____________________________________ for the academic year 20___ - 20___ as prescribed in the curriculum.

Place: _____________

Date: ______________

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	27/03/24	
2	Implement Stack ADT using an array.	27/03/24	
3	Convert an Infix expression to Postfix expression using stack ADT.	27/03/24	
4	Evaluate Postfix Expression using Stack ADT.	27/03/24	
5	Implement Linear Queue ADT using an array.	27/03/24	
6	Implement Circular Queue ADT using an array.	06/04/24	
7	Implement Singly Linked List ADT.	01/04/24	
8	Implement Circular Linked List ADT.	01/04/24	
9	Implement Stack ADT using Linked List	06/06/24	
10	Implement Linear Queue ADT using Linked List	27/03/24	
11	Implement Binary Search Tree ADT using Linked List.	06/04/24	
12	Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search	06/04/24	
13	Implement Binary Search algorithm to search an element in an array	01/04/24	
14	Implement Bubble sort algorithm to sort elements of an array in ascending and descending order	01/04/24	

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 1

Title: Implement Array and write a menu driven program to perform all the operation on array elements

Theory:

This C++ code implements various operations on a dynamic array, such as insertion and deletion at different positions. It uses functions like insertAtEnd, insertAtBeginning, insertBeforeElement, insertAfterElement, insertAtIndex, deleteElement, deleteElementAtIndex, deleteLastElement, and deleteFirstElement to perform these operations based on user input. The program presents a menu to the user, allowing them to choose the operation they want to perform.

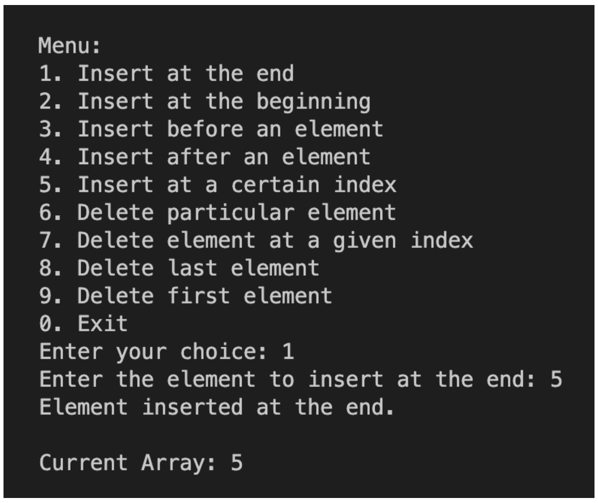
Code:

```
#include <iostream>
using namespace std;

const int MAX_SIZE = 100;  // Adjust the size as needed

void displayArray(int arr[], int size) {
   for (int i = 0; i < size; ++i) {
       cout << arr[i] << " ";
   }
   cout << endl;
}

void insertAtEnd(int arr[], int& size, int element) {
   if (size < MAX_SIZE) {
       arr[size++] = element;
       cout << "Element inserted at the end." << endl;
```

```cpp
    } else {
        cout << "Array is full. Cannot insert." << endl;
    }
}

void insertAtBeginning(int arr[], int& size, int element) {
    if (size < MAX_SIZE) {
        // Shift elements to make space for the new element
        for (int i = size; i > 0; --i) {
            arr[i] = arr[i - 1];
        }
        arr[0] = element;
        size++;
        cout << "Element inserted at the beginning." << endl;
    } else {
        cout << "Array is full. Cannot insert." << endl;
    }
}

void insertBeforeElement(int arr[], int& size, int element, int target) {
    int index = 0;
    while (index < size && arr[index] != target) {
        index++;
    }

    if (index < size) {
        // Shift elements to make space for the new element
        for (int i = size; i > index; --i) {
            arr[i] = arr[i - 1];
        }
        arr[index] = element;
        size++;
        cout << "Element inserted before the specified element." << endl;
    } else {
        cout << "Element not found. Cannot insert before." << endl;
    }
}

void insertAfterElement(int arr[], int& size, int element, int target) {
    int index = 0;
    while (index < size && arr[index] != target) {
        index++;
    }

    if (index < size - 1) {
        // Shift elements to make space for the new element
        for (int i = size; i > index + 1; --i) {
            arr[i] = arr[i - 1];
        }
        arr[index + 1] = element;
        size++;
        cout << "Element inserted after the specified element." << endl;
    } else if (index == size - 1) {
        // If the target is the last element, insert after
        arr[size++] = element;
        cout << "Element inserted after the specified element." << endl;
    } else {
        cout << "Element not found. Cannot insert after." << endl;
    }
}

void insertAtIndex(int arr[], int& size, int element, int index) {
    if (size < MAX_SIZE && index >= 0 && index <= size) {
        // Shift elements to make space for the new element
        for (int i = size; i > index; --i) {
            arr[i] = arr[i - 1];
        }
        arr[index] = element;
        size++;
        cout << "Element inserted at index " << index << "." << endl;
    } else if (size >= MAX_SIZE) {
        cout << "Array is full. Cannot insert." << endl;
    } else {
        cout << "Invalid index. Cannot insert." << endl;
    }
}

void deleteElement(int arr[], int& size, int target) {
    int index = 0;
    while (index < size && arr[index] != target) {
        index++;
    }
    if (index < size) {
        // Shift elements to remove the specified element
        for (int i = index; i < size - 1; ++i) {
            arr[i] = arr[i + 1];
        }
        size--;
        cout << "Element deleted." << endl;
    } else {
        cout << "Element not found. Cannot delete." << endl;
    }
}

void deleteElementAtIndex(int arr[], int& size, int index) {
    if (index >= 0 && index < size) {
        // Shift elements to remove the element at the specified index
        for (int i = index; i < size - 1; ++i) {
            arr[i] = arr[i + 1];
        }
        size--;
        cout << "Element at index " << index << " deleted." << endl;
    } else {
        cout << "Invalid index. Cannot delete." << endl;
    }
}

void deleteLastElement(int arr[], int& size) {
    if (size > 0) {
        size--;
        cout << "Last element deleted." << endl;
    } else {
        cout << "Array is empty. Cannot delete the last element." << endl;
    }
}

void deleteFirstElement(int arr[], int& size) {
    if (size > 0) {
        // Shift elements to remove the first element
        for (int i = 0; i < size - 1; ++i) {
            arr[i] = arr[i + 1];
        }
        size--;
        cout << "First element deleted." << endl;
    } else {
        cout << "Array is empty. Cannot delete the first element." << endl;
    }
}

int main() {
    int arr[MAX_SIZE];
    int size = 0;
    int choice, element, index;

    do {
        cout << "\nMenu:\n";
        cout << "1. Insert at the end\n";
        cout << "2. Insert at the beginning\n";
        cout << "3. Insert before an element\n";
        cout << "4. Insert after an element\n";
        cout << "5. Insert at a certain index\n";
        cout << "6. Delete particular element\n";
        cout << "7. Delete element at a given index\n";
        cout << "8. Delete last element\n";
        cout << "9. Delete first element\n";
        cout << "0. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter the element to insert at the end: ";
                cin >> element;
                insertAtEnd(arr, size, element);
                break;
            case 2:
                cout << "Enter the element to insert at the beginning: ";
                cin >> element;
                insertAtBeginning(arr, size, element);
                break;
            case 3:
                cout << "Enter the element to insert: ";
                cin >> element;
                cout << "Enter the element before which to insert: ";
                cin >> index;
                insertBeforeElement(arr, size, element, index);
                break;
            case 4:
                cout << "Enter the element to insert: ";
                cin >> element;
                cout << "Enter the element after which to insert: ";
                cin >> index;
                insertAfterElement(arr, size, element, index);
                break;
            case 5:
                cout << "Enter the element to insert: ";
                cin >> element;
                cout << "Enter the index at which to insert: ";
                cin >> index;
                insertAtIndex(arr, size, element, index);
                break;
            case 6:
                cout << "Enter the element to delete: ";
                cin >> element;
                deleteElement(arr, size, element);
                break;
            case 7:
                cout << "Enter the index to delete element at: ";
                cin >> index;
                deleteElementAtIndex(arr, size, index);
                break;
            case 8:
                deleteLastElement(arr, size);
                break;
            case 9:
                deleteFirstElement(arr, size);
                break;
            case 0:
                cout << "Exiting the program. Bye!\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
```

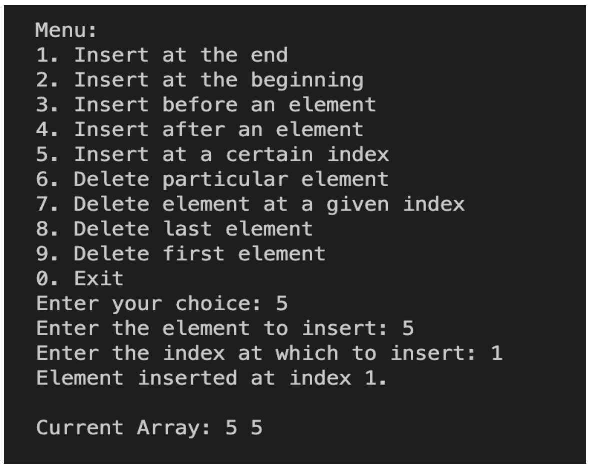
```
        cout << "\nCurrent Array: ";
        displayArray(arr, size);

    } while (choice != 0);

    return 0;
}
```

Output:(screenshot)

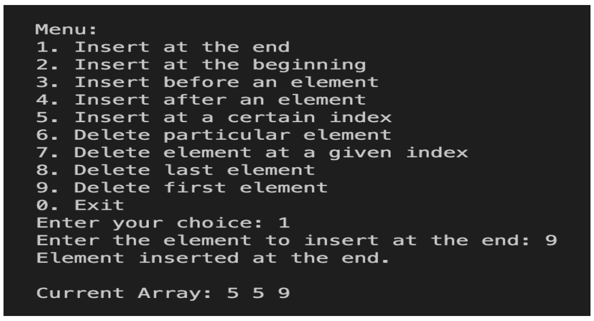


```
Menu:
1. Insert at the end
2. Insert at the beginning
3. Insert before an element
4. Insert after an element
5. Insert at a certain index
6. Delete particular element
7. Delete element at a given index
8. Delete last element
9. Delete first element
0. Exit
Enter your choice: 1
Enter the element to insert at the end: 5
Element inserted at the end.

Current Array: 5
```

Test Case: Any two (screenshot)



```
Menu:
1. Insert at the end
2. Insert at the beginning
3. Insert before an element
4. Insert after an element
5. Insert at a certain index
6. Delete particular element
7. Delete element at a given index
8. Delete last element
9. Delete first element
0. Exit
Enter your choice: 5
Enter the element to insert: 5
Enter the index at which to insert: 1
Element inserted at index 1.

Current Array: 5 5
```



```
Menu:
1. Insert at the end
2. Insert at the beginning
3. Insert before an element
4. Insert after an element
5. Insert at a certain index
6. Delete particular element
7. Delete element at a given index
8. Delete last element
9. Delete first element
0. Exit
Enter your choice: 1
Enter the element to insert at the end: 9
Element inserted at the end.

Current Array: 5 5 9
```

Conclusion:

Hence using switch-case statements in the main function, and functions for every array operation, I have made a menu-driven program for all array operations.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 2

Title: Implement Stack ADT using array.

Theory:

Implementing a Stack ADT using an array involves:

- Using a fixed-size array and a top index.
- Operations: push (add), pop (remove), peek (view top).

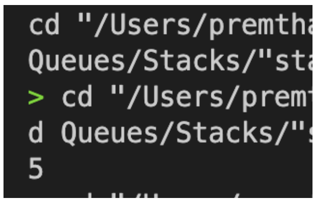
Code:

```cpp
#include <iostream>
using namespace std;

#define n 10

class Stack{
   int top;
   int * arr;

   public:
   Stack(){
       arr = new int[n];
       top = -1;
   }

   void push(int val){
       if (top == n-1) {
           cout << "Stack is full\n";
           return;
       }

       top++;
       arr[top] = val;
   }

   void pop(){
       if(top == -1){
           cout << "Stack is empty\n";
           return;
       }

       top--;
   }

   int topElement(){
       if(top == -1){
           cout << "No element in stack\n";
           return -1;
       }

       return arr[top];
```

```
    }

    bool isEmpty(){
        return top == -1;
    }
};

int main(){
    Stack s;

    s.pop();
    cout << s.topElement()<<endl;
    cout << s.isEmpty() << endl;

    return 0;
}
```

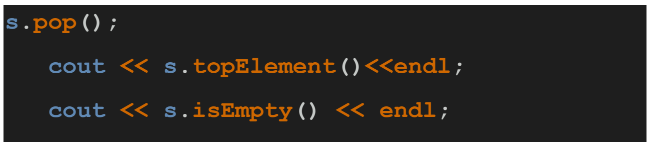
Output: (screenshot)

```
s.push(7);
s.push(5);
cout << s.topElement()<<endl;
```

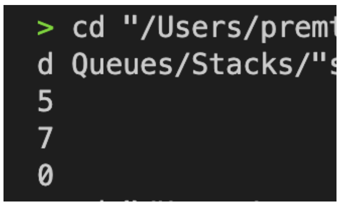


```
cd "/Users/premtha
Queues/Stacks/"sta
> cd "/Users/prem
d Queues/Stacks/"
5
```

Test Case: screenshot)



```
s.pop();
   cout << s.topElement()<<endl;
   cout << s.isEmpty() << endl;
```



```
> cd "/Users/prem
d Queues/Stacks/"
5
7
0
```

Conclusion:

Hence using an array of fixed size and creating functions to do the various operations on a stack ADT, the program has been implemented.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 3

Title: Convert an Infix expression to Postfix expression using stack ADT.

Theory:

Converting an infix expression to a postfix expression using a stack ADT involves:

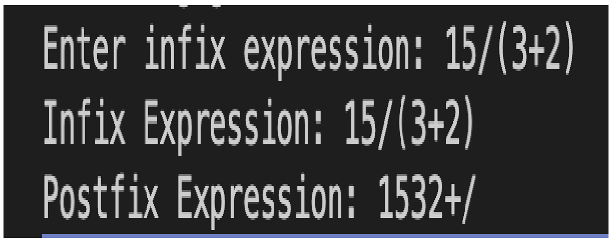
- Iterating through each character in the infix expression.
- Using a stack to store operators temporarily.
- Following operator precedence rules to determine the order of operations.
- Outputting operands immediately and operators after processing.
- Resulting postfix expression has operators placed after their operands.

Code:

```
#include <iostream>
#include <stack>
#include <cctype>

using namespace std;

int precedence(char op) {
   if (op == '+' || op == '-')
       return 1;
   if (op == '*' || op == '/')
       return 2;
   if(op == '^')
       return 3;
   return 0;
}

string infixToPostfix(string infix) {
   stack<char> operatorStack;
   string postfix;

   for (char c : infix) {
       if (isalnum(c)) {
```

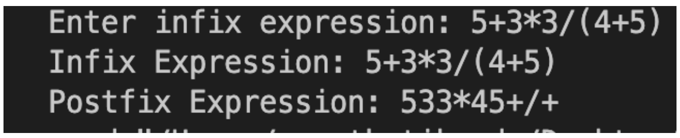
```cpp
            postfix += c;
        } else if (c == '(') {
            operatorStack.push(c);
        } else if (c == ')') {
            while (!operatorStack.empty() && operatorStack.top() != '(') {
                postfix += operatorStack.top();
                operatorStack.pop();
            }
            operatorStack.pop(); // Pop '('
        } else {
            while (!operatorStack.empty() && precedence(operatorStack.top()) >= precedence(c)) {
                postfix += operatorStack.top();
                operatorStack.pop();
            }
            operatorStack.push(c);
        }
    }

    while (!operatorStack.empty()) {
        postfix += operatorStack.top();
        operatorStack.pop();
    }

    return postfix;
}

int main() {
    string infixExpression;
    cout << "Enter infix expression: ";
    cin >> infixExpression;

    string postfixExpression = infixToPostfix(infixExpression);

    cout << "Infix Expression: " << infixExpression << endl;
    cout << "Postfix Expression: " << postfixExpression << endl;

    return 0;}
```

Output: (screenshot)

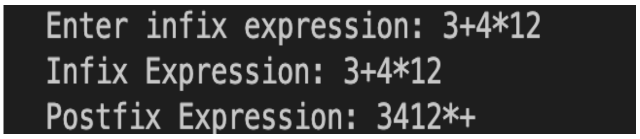


```
Enter infix expression: 15/(3+2)
Infix Expression: 15/(3+2)
Postfix Expression: 1532+/
```

Test Case: Any two (screenshot)



```
Enter infix expression: 5+3*3/(4+5)
Infix Expression: 5+3*3/(4+5)
Postfix Expression: 533*45+/+
```



```
Enter infix expression: 3+4*12
Infix Expression: 3+4*12
Postfix Expression: 3412*+
```

Conclusion:

Hence by simply using a stack structure and traversing through the infix expression, we can convert it to its postfix expression format.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 4

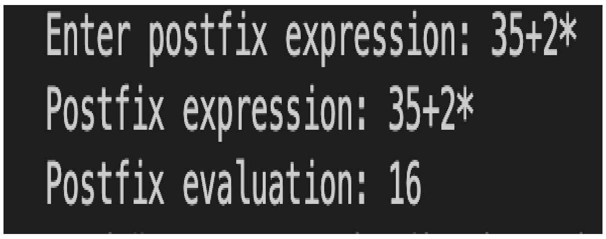
Title: Evaluate Postfix Expression using Stack ADT.

Theory: Traverse through the entire infix expression, and use precedence to figure out the order of calculation of the values. Output the final calculated value at the end.

Code:

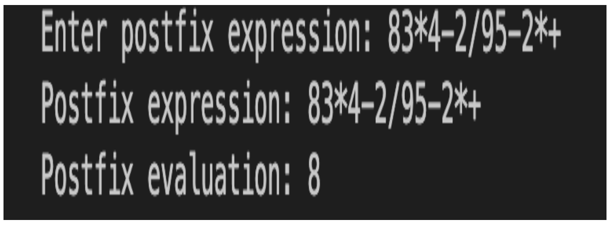
```
#include <iostream>
#include <stack>
#include <cmath>
using namespace std;

int postfixEvaluation(string s){
   stack <int> st;
   int a,b;
   for(char c: s){
       if(c <= '9' && c >= '0'){
           st.push((int)(c - '0'));
       }
       else{
           a = st.top();
           st.pop();
           b = st.top();
           st.pop();

           switch(c){
               case '+': st.push(a+b);
                         break;
               case '-': st.push(b-a);
                         break;
               case '*': st.push(a*b);
                         break;
               case '/': st.push(a/b);
                         break;
               case '^': st.push(pow(a,b));
                         break;
           }
       }
   }

   return st.top();
}

int main(){
   string postfixExpression;
   cout << "Enter postfix expression: ";
   cin >> postfixExpression;

   cout << "Postfix expression: " << postfixExpression << endl;
   cout << "Postfix evaluation: " << postfixEvaluation(postfixExpression) << endl;

   return 0;
}
```

Output: (screenshot)

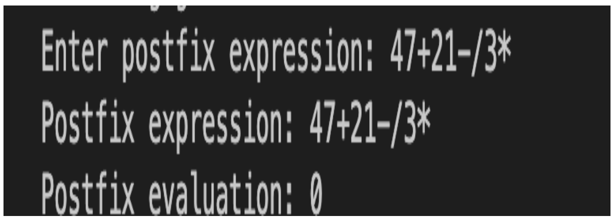


```
Enter postfix expression: 35+2*
Postfix expression: 35+2*
Postfix evaluation: 16
```

Test Case: Any two (screenshot)



```
Enter postfix expression: 83*4-2/95-2*+
Postfix expression: 83*4-2/95-2*+
Postfix evaluation: 8
```



```
Enter postfix expression: 47+21-/3*
Postfix expression: 47+21-/3*
Postfix evaluation: 0
```

Conclusion:

Hence, simply iterating through the postfix expression, we can keep adding 2 elements onto the stack, perform the corresponding operation, then push it back onto the stack.
At the end of the iteration, the element on the top of the stack is the final result.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 5

Title: Implement Linear Queue ADT using array.

Theory:

- Use a fixed-size array to store queue elements.
- Maintain front and rear pointers to track the queue's head and tail.

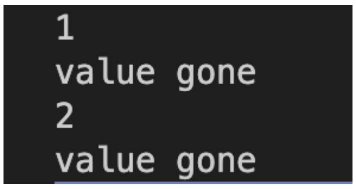
- Implement operations like enqueue (add), dequeue (remove), isEmpty, and isFull.
- Enqueue adds elements at the rear, dequeue removes from the front.

Code:

```
#include <iostream>
using namespace std;

#define n 5

class Queue{
   int *arr;
   int front, rear;

   public:
       Queue(){
           arr = new int[n];
           front = -1;
           rear = -1;
       }

       void enqueue(int x){
           if(rear == n-1){
               cout << "Overflow\n";
               return;
           }

           if(front == -1){
               front++;
           }
           rear++;
           arr[rear] = x;
       }

       void dequeue(){
           if(front > rear || front == -1){
               cout << "No elements\n";
               return;
           }
           cout << "value gone" << endl;
```

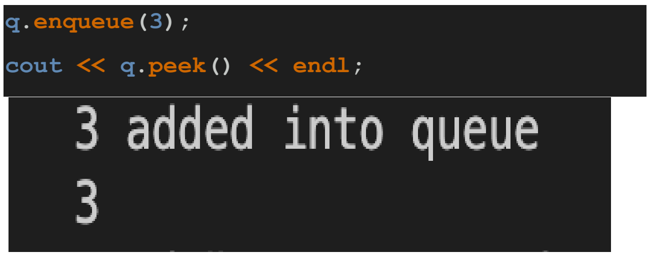
```
            front++;
        }

        int peek(){
            if(front > rear || front == -1){
                cout << "No elements\n";
                return -1;
            }
            return arr[front];
        }

        bool empty(){
            if(front == -1 || front > rear){
                return true;
            }
            return false;
        }
};

int main(){
    Queue q;
    q.enqueue(1);
    q.enqueue(2);
    cout << q.peek() << endl;
    q.dequeue();
    cout << q.peek() << endl;
    q.dequeue();


    return 0;
}
```

Output: (screenshot)



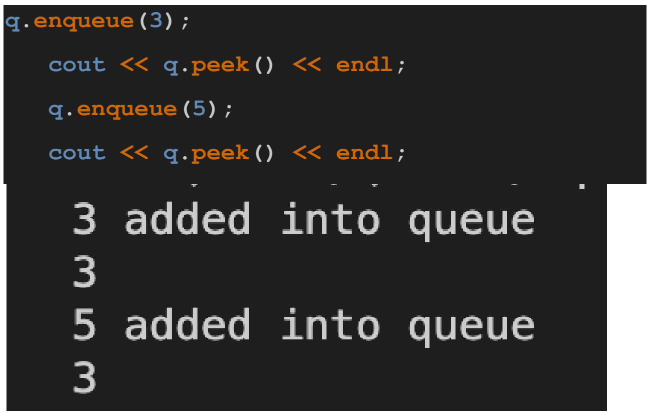
```
1
value gone
2
value gone
```

Test Case: Any two (screenshot)



```
q.enqueue(3);
cout << q.peek() << endl;
```

```
3 added into queue
3
```



```
q.enqueue(3);
    cout << q.peek() << endl;
    q.enqueue(5);
    cout << q.peek() << endl;
```

```
3 added into queue
3
5 added into queue
3
```

Conclusion:

Hence using an array of fixed size and by accessing its index values, we have implemented a linear queue with its operations like enqueue, dequeue, peek and isEmpty.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No. : 6

Title: Implementation of circular queue using an array

Theory:

- **Array: Stores the queue elements. Fixed size is defined beforehand.**
- **Front and Rear pointers:**

  front points to the first element in the queue.

  rear points to the last element (or the position to insert the next element).
- **Circular behavior: When rear reaches the end of the array (index size-1), it wraps around to the beginning (index 0) using the modulo operator (%). This simulates the circular structure.**

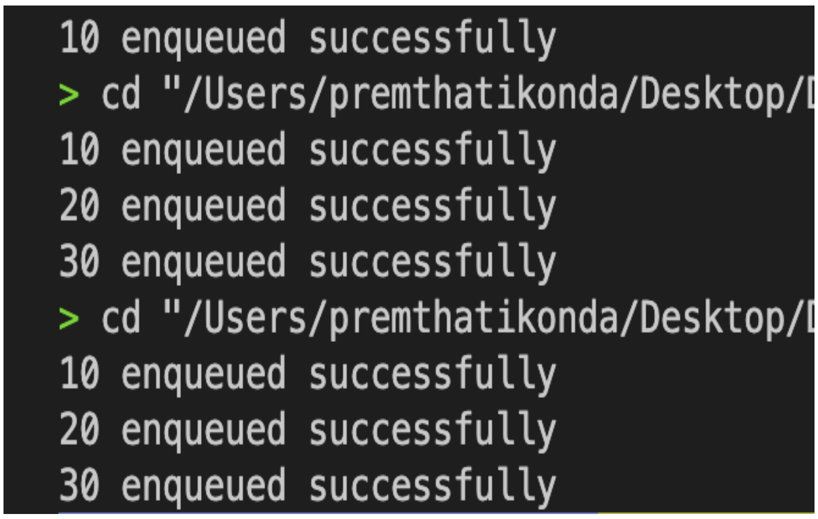
Code:

```
#include <iostream>
using namespace std;

class CircularQueue {
   private:
       int size;
       int* queue; // Array to store queue elements
       int front;
       int rear;

   public:
   CircularQueue(int size) {
       this->size = size;
       queue = new int[size];  // Allocate memory for the queue array
       front = rear = -1;
```

```cpp
}

~CircularQueue() {
    delete[] queue;  // Deallocate memory when the object goes out of scope
}

bool isEmpty() {
    return front == -1;
}

bool isFull() {
    return (rear + 1) % size == front;
}

void enqueue(int data) {
    if (isFull()) {
    cout << "Queue Overflow\n";
    return;
    }
    if (isEmpty()) {
    front = 0;
    }
    rear = (rear + 1) % size;
    queue[rear] = data;
    cout << data << " enqueued successfully\n";
}

int dequeue() {
    if (isEmpty()) {
    cout << "Queue Underflow\n";
    return -1;
    }
    int data = queue[front];
    if (front == rear) {
    front = rear = -1;
    } else {
    front = (front + 1) % size;
    }
    return data;
}
```

```
};

int main() {
 CircularQueue q(5);
 q.enqueue(10);
 q.enqueue(20);
 q.enqueue(30);
 cout << q.dequeue() << " dequeued\n";
 cout << q.dequeue() << " dequeued\n";
 return 0;
}
```

Output and test cases:



```
10 enqueued successfully
> cd "/Users/premthatikonda/Desktop/I
10 enqueued successfully
20 enqueued successfully
30 enqueued successfully
> cd "/Users/premthatikonda/Desktop/I
10 enqueued successfully
20 enqueued successfully
30 enqueued successfully
```

Conclusion:

Hence by using an array of fixed size and 2 pointers pointing at the locations of the queue elements, we can implement a circular queue using an array.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 7

Title: Implementation of singular linked list ADT.

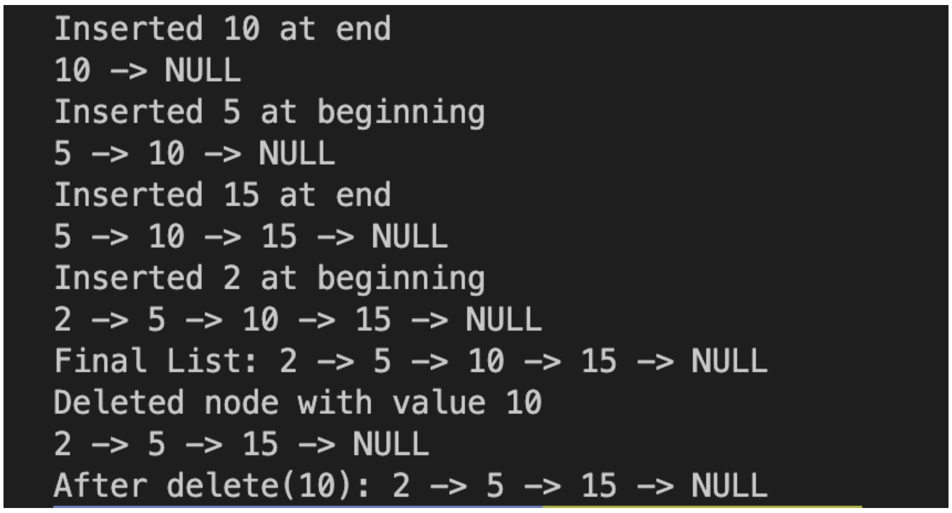
Theory: Create a ‘Node’ class which initializes the node’s value and the next pointer of the node.

Code:

```cpp
#include <iostream>
using namespace std;

// Node structure for the linked list
struct Node {
 int data;
 Node* next;
};

// SinglyLinkedList class
class SinglyLinkedList {
public:
 // Constructor
 SinglyLinkedList() {
   head = nullptr;
 }


 // Function to insert a node at the beginning of the list
 void insertAtBeginning(int data) {
   Node* newNode = new Node;
   newNode->data = data;
   newNode->next = head;
   head = newNode;
   printChange("Inserted " + to_string(data) + " at beginning");
 }
```

```cpp
// Function to insert a node at the end of the list
void insertAtEnd(int data) {
  Node* newNode = new Node;
  newNode->data = data;
  newNode->next = nullptr;

  if (head == nullptr) {
    head = newNode;
    printChange("Inserted " + to_string(data) + " at end");
    return;
  }

  Node* current = head;
  while (current->next != nullptr) {
    current = current->next;
  }
  current->next = newNode;
  printChange("Inserted " + to_string(data) + " at end");
}

// Function to delete a node with a specific value
void deleteNode(int value) {
  if (head == nullptr) {
    return;
  }

  Node* current = head;
  Node* previous = nullptr;

  while (current != nullptr && current->data != value) {
    previous = current;
    current = current->next;
  }

  if (current == nullptr) {
    // Value not found
    return;
  }



  if (previous == nullptr) {
     // Delete head node
     head = current->next;
   } else {
     previous->next = current->next;
   }

   delete current;
   printChange("Deleted node with value " + to_string(value));
 }

 // Function to print the contents of the list
 void printList() {
   Node* current = head;
   while (current != nullptr) {
     cout << current->data << " -> ";
     current = current->next;
   }
   cout << "NULL" << endl;
 }

 // Function to check if the list is empty
 bool isEmpty() {
   return head == nullptr;
 }

private:
 Node* head;  // Head pointer of the linked list

 // Function to clear the list (optional)
 void clear() {
   while (head != nullptr) {
     Node* temp = head;
     head = head->next;
     delete temp;
   }
 }

 // Helper function to print change message
 void printChange(const string& message) {
   cout << message << endl;
   printList();
 }
};

int main() {
 SinglyLinkedList list;

 list.insertAtEnd(10);
 list.insertAtBeginning(5);
 list.insertAtEnd(15);
 list.insertAtBeginning(2);

 cout << "Final List: ";
 list.printList();

 list.deleteNode(10);

 cout << "After delete(10): ";
 list.printList();

 return 0;
}
```

Output: (screenshot)



```
Inserted 10 at end
10 -> NULL
Inserted 5 at beginning
5 -> 10 -> NULL
Inserted 15 at end
5 -> 10 -> 15 -> NULL
Inserted 2 at beginning
2 -> 5 -> 10 -> 15 -> NULL
Final List: 2 -> 5 -> 10 -> 15 -> NULL
Deleted node with value 10
2 -> 5 -> 15 -> NULL
After delete(10): 2 -> 5 -> 15 -> NULL
```

Conclusion:

Hence, a menu-driven program using different functions for different operations on linked lists has been made. All the functions take in the value to add or delete as the parameter and are present inside the main class.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 8

Title: Implementation of circular linked list ADT.

Theory: Create a 'Node' class which initializes the node's value and the next pointer of the node. Make functions to create nodes, insert and delete nodes at the beginning and end of the linked list.

Code:

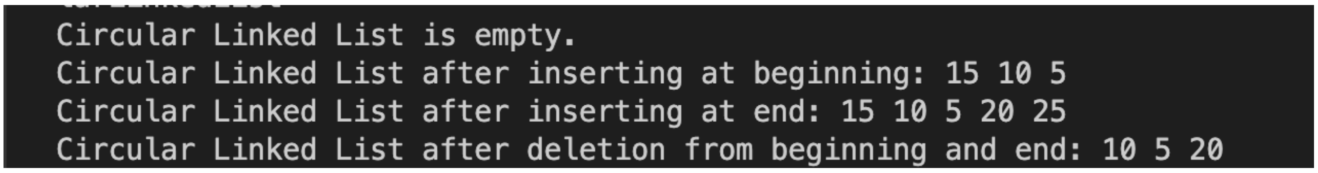
```
#include <iostream>
using namespace std;

class Node {
public:
   int data;
   Node* next;

   Node(int value) {
       data = value;
       next = nullptr;
   }
```

```
};



class CircularLinkedList {
private:
   Node* head;
public:
   CircularLinkedList() {
       head = nullptr;
   }

   void insertAtBeginning(int value) {
       Node* newNode = new Node(value);
       if (head == nullptr) {
           head = newNode;
           head->next = head;
       }
       else{
           Node* last = head;
           while (last->next != head) {
               last = last->next;
           }
           newNode->next = head;
           last->next = newNode;
           head = newNode;
       }
   }

   void insertAtEnd(int value) {
       Node* newNode = new Node(value);
       if (head == nullptr) {
           head = newNode;
           head->next = head;
       }
       else{
           Node* last = head;
           while (last->next != head) {
               last = last->next;
           }
           last->next = newNode;
           newNode->next = head;
```

```
        }
    }

    void deleteFromBeginning() {
        if (head == nullptr) {
            cout << "Circular Linked List is empty. Deletion is not possible." << endl;
            return;
        }
        Node* temp = head;
        if (head->next == head) {
            delete head;
            head = nullptr;
        } else {
            Node* last = head;
            while (last->next != head) {
                last = last->next;
            }
            head = head->next;
            last->next = head;
            delete temp;
        }
    }

    void deleteFromEnd() {
        if (head == nullptr) {
            cout << "Circular Linked List is empty. Deletion is not possible." << endl;
            return;
        }
        Node* temp = head;
        if (head->next == head) {
            delete head;
            head = nullptr;
        } else {
            Node* last = head;
            while (last->next->next != head) {
                last = last->next;
            }
            Node* toDelete = last->next;
            last->next = head;
            delete toDelete;
        }
```

```cpp
    }

    void display() {
        if (head == nullptr) {
            cout << "Circular Linked List is empty." << endl;
            return;
        }
        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        cout << endl;
    }
};

int main() {
    CircularLinkedList cll;
    cll.display();

    // Inserting elements into the circular linked list
    cll.insertAtBeginning(5);
    cll.insertAtBeginning(10);
    cll.insertAtBeginning(15);

    // Displaying elements of the circular linked list
    cout << "Circular Linked List after inserting at beginning: ";
    cll.display();

    // Inserting elements at the end
    cll.insertAtEnd(20);
    cll.insertAtEnd(25);

    // Displaying elements of the circular linked list
    cout << "Circular Linked List after inserting at end: ";
    cll.display();

    // Deleting elements from the beginning and end
    cll.deleteFromBeginning();
    cll.deleteFromEnd();
    // Displaying elements of the circular linked list
    cout << "Circular Linked List after deletion from beginning and end: ";
    cll.display();

    return 0;
}
```

Output and test case combined:



```
Circular Linked List is empty.
Circular Linked List after inserting at beginning: 15 10 5
Circular Linked List after inserting at end: 15 10 5 20 25
Circular Linked List after deletion from beginning and end: 10 5 20
```

Conclusion:

Hence using a class for initialisation of nodes, and functions to create and delete nodes in the beginning and end, a circular linked list has been implemented.

Name of student: Prem Thatikonda

Roll No. : 06

Experiment no. : 09

Title: Implementation of stack ADT using a linked list.

Theory:

- **Linked List: The stack is built on a singly linked list. Each node holds data and a pointer to the next node.**
- **Top Pointer: A pointer within the stack class that always points to the topmost element (head) of the linked list.**
- **Push:**

  Create a new node for the element to be pushed.

  Link the new node to the current top element.

  Update the top pointer to point to the new node (becomes the new top).
- **Pop:**

  Check if empty (top pointer is null).

  If not empty, store the data from the top element.

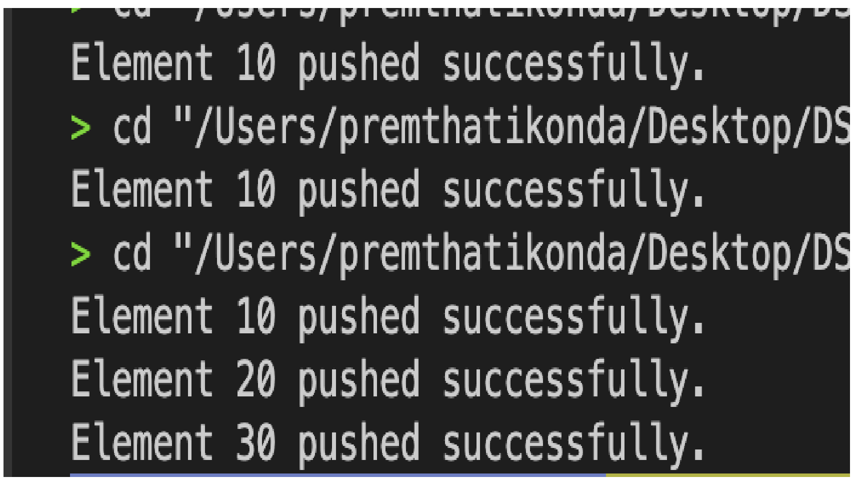
  Update the top pointer to point to the next element.

  Deallocate memory of the popped node.

Code:

```
#include <iostream>
using namespace std;

struct Node {
 int data;
 Node* next;
};

class Stack {
   private:
       Node* top;

   public:
   Stack() {
       top = nullptr;
   }

   void push(int data) {
       Node* newNode = new Node;
       newNode->data = data;
       newNode->next = top;
       top = newNode;
       cout << "Element " << data << " pushed successfully." << endl;
   }

   int pop() {
       if (isEmpty()) {
       cout << "Stack Underflow\n";
       return -1;
       }
       Node* temp = top;
       int data = temp->data;
       top = top->next;
       delete temp;
       return data;
   }

   bool isEmpty() {
       return top == nullptr;
   }
};

int main() {
 Stack s;
 s.push(10);
 s.push(20);
 s.push(30);
 cout << s.pop() << " popped\n";
 cout << s.pop() << " popped\n";
 return 0;
}
```

Output and test cases:



```
Element 10 pushed successfully.
> cd "/Users/premthatikonda/Desktop/DS
Element 10 pushed successfully.
> cd "/Users/premthatikonda/Desktop/DS
Element 10 pushed successfully.
Element 20 pushed successfully.
Element 30 pushed successfully.
```

Conclusion: Hence using a linked list data structure and a top pointer, we can implement a stack using that linked list.

Name of student: Prem Thatikonda

Roll no. : 06

Experiment No: 10

Title: Implementation of linear queue using a linked list.

Theory:

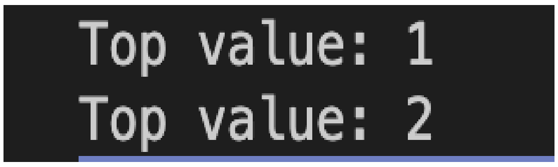
A linear queue containing 2 pointers, front and rear, is implemented using a linked list. Insertions happen at the rear end, and deletions happen from the front end.

Code:

```
#include <iostream>
using namespace std;

class Node{
   public:
       int data;
       Node * next;

       Node(int val){
           data = val;
           next = NULL;
       }
};

class Queue{
   public:
       Node * front;
       Node * rear;

       Queue(){
           front = NULL;
           rear = NULL;
       }

       void enqueue(int x){
           Node * n = new Node(x);

           if(front == NULL){
               rear = n;
```

```
                front = n;
                return;
            }

            rear->next = n;
            rear = n;
            // 2->3->4
        }

        void dequeue(){
            if(front == NULL){
                cout << "Underflow\n";
                return;
            }

            Node * temp = front;
            front = front->next;
            delete temp;
        }

        int peek(){
            if(front == NULL) {
                cout << "Empty queue" << endl;
                return -1;
            }

            return front->data;
        }

        bool empty(){
            if(front == NULL){
                return true;
            }
            return false;
        }
};


int main(){
    Queue q;
    q.enqueue(1);
    q.enqueue(2);
    cout << "Top value: "<<q.peek() << endl;
    q.dequeue();
    cout << "Top value: "<<q.peek() << endl;
    q.dequeue();

    return 0;
}
```

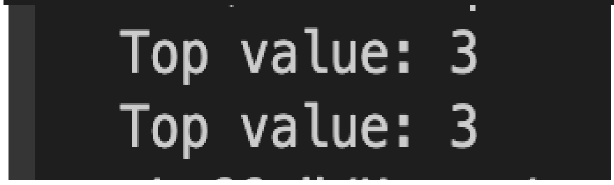
Output: (screenshot)



```
Top value: 1
Top value: 2
```

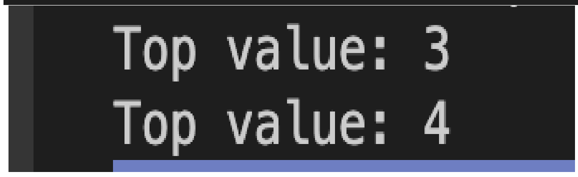
Test Case: Any two (screenshot)

```
     q.enqueue(3);
    cout << "Top value: "<<q.peek() << endl;
    q.enqueue(4);
    cout << "Top value: "<<q.peek() << endl;
    q.dequeue();
```



```
Top value: 3
Top value: 3
```

```
q.enqueue(3);
    cout << "Top value: "<<q.peek() << endl;
    q.dequeue();
    q.enqueue(4);
        cout  <<  "Top  value:  "<<q.peek()  <<
endl;
```



```
Top value: 3
Top value: 4
```

Conclusion:

Hence, using a linked list, we can implement a linear queue and do its operations like enqueue, dequeue, peek and isEmpty via member functions of the Queue class.

Each node must be created using the Node class and the new operator.

Name of Student: Prem Thatikonda
Roll Number:06
Experiment No:11

Title: Implementation of a binary search tree ADT using linked list.

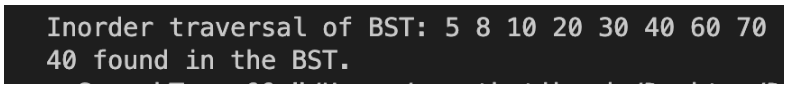
Theory: A node struct is made containing the data values ‘data’, ‘left’ and ‘right’, basically representing each node of the tree. There’s a function to insert nodes at the right place, search for a node, and also print the tree’s inorder traversal.

Code:

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node* left;
   Node* right;
};

Node* createNode(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->left = newNode->right = nullptr;
   return newNode;
}

Node* insert(Node* root, int value) {
   if (root == nullptr) {
```

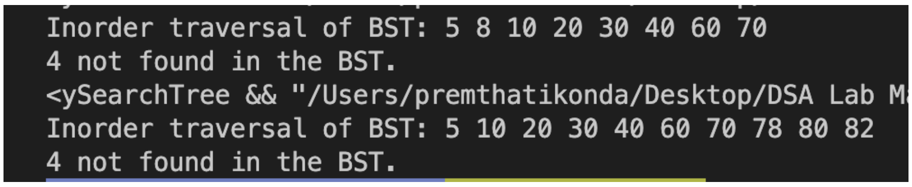
```cpp
        return createNode(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}

bool search(Node* root, int value) {
    if (root == nullptr) {
        return false;
    }
    if (root->data == value) {
        return true;
    } else if (value < root->data) {
        return search(root->left, value);
    } else {
        return search(root->right, value);
    }
}

void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}

int main() {
    Node* root = nullptr;
    root = insert(root, 20);
    insert(root, 30);
    insert(root, 10);
    insert(root, 40);
    insert(root, 5);
    insert(root, 60);
    insert(root, 70);
    insert(root, 8);
    cout << "Inorder traversal of BST: ";
    inorderTraversal(root);
    cout << endl;

    int searchValue = 40;
    if (search(root, searchValue)) {
        cout << searchValue << " found in the BST." << endl;
    } else {
        cout << searchValue << " not found in the BST." << endl;
    }

    return 0;
}
```

Output:



```
Inorder traversal of BST: 5 8 10 20 30 40 60 70
40 found in the BST.
```

Test Cases:



```
Inorder traversal of BST: 5 8 10 20 30 40 60 70
4 not found in the BST.
<ySearchTree && "/Users/premthatikonda/Desktop/DSA Lab M
Inorder traversal of BST: 5 10 20 30 40 60 70 78 80 82
4 not found in the BST.
```

Conclusion:

Hence using a struct to initialize the nodes and a function to create, insert and search for the nodes, the binary search tree has been implemented.

Name of student: Prem Thatikonda
Roll No. : 06
Experiment No. : 12

Title : Implementation of DFS and BFS graph traversal.

Theory:

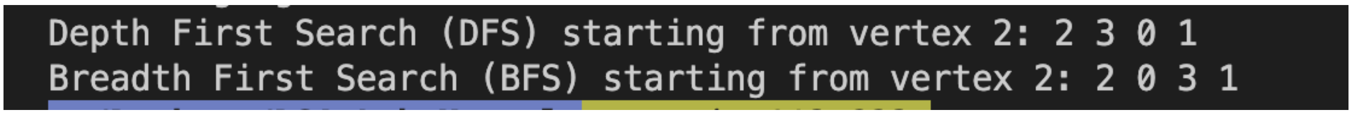
- **Create an adjacency list depicting the edges between nodes.**
- **Create a stack and queue as helper data structures for the DFS and BFS traversals respectively.**
- **Travel through the graph based off the boolean value present in a different 'visited' array.**

Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>

using namespace std;

class Graph {
private:
   int V; // Number of vertices
   vector<vector<int> > adj; // Adjacency list representation of graph

public:
   Graph(int vertices) : V(vertices) {
       adj.resize(V);
   }

   // Function to add an edge to the graph
   void addEdge(int v, int w) {
       adj[v].push_back(w); // Add w to v's list
   }

   // Depth First Search traversal starting from a given vertex
   void DFS(int start) {
       // Mark all the vertices as not visited
```

```
        vector<bool> visited(V, false);

        // Create a stack for DFS
        stack<int> stack;

        // Push the current source node
        stack.push(start);

        while (!stack.empty()) {
            // Pop a vertex from stack and print it
            int current = stack.top();
            stack.pop();

            // Stack may contain same vertex twice. So, we need to print the popped item only
if it is not visited.
            if (!visited[current]) {
                cout << current << " ";
                visited[current] = true; // Mark the current node as visited

                // Get all adjacent vertices of the popped vertex and push the adjacent
vertices to the stack if not already visited
                for (auto it = adj[current].begin(); it != adj[current].end(); ++it) {
                    if (!visited[*it]) {
                        stack.push(*it);
                    }
                }
            }
        }
    }

    // Breadth First Search traversal starting from a given vertex
    void BFS(int start) {
        // Mark all the vertices as not visited
        vector<bool> visited(V, false);

        // Create a queue for BFS
        queue<int> queue;

        // Mark the current node as visited and enqueue it
        visited[start] = true;
        queue.push(start);
        while (!queue.empty()) {
            // Dequeue a vertex from queue and print it
            int current = queue.front();
            cout << current << " ";
            queue.pop();

            // Get all adjacent vertices of the dequeued vertex current. If an adjacent vertex
has not been visited, then mark it visited and enqueue it
            for (auto it = adj[current].begin(); it != adj[current].end(); ++it) {
                if (!visited[*it]) {
                    visited[*it] = true;
                    queue.push(*it);
                }
            }
        }
    }
};

int main() {
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Depth First Search (DFS) starting from vertex 2: ";
    g.DFS(2);
    cout << endl;

    cout << "Breadth First Search (BFS) starting from vertex 2: ";
    g.BFS(2);
    cout << endl;

    return 0;
}
```

Output:



```
Depth First Search (DFS) starting from vertex 2: 2 3 0 1
Breadth First Search (BFS) starting from vertex 2: 2 0 3 1
```

Conclusion:

Hence using stack for DFS, and a queue for BFS as helper data structures, we have implemented the 2 graph traversal techniques.

Name of Student: Prem Thatikonda
Roll Number: 06
Experiment No:13

Title: Implementation of binary search algorithm to search for an element in an array.

Theory:

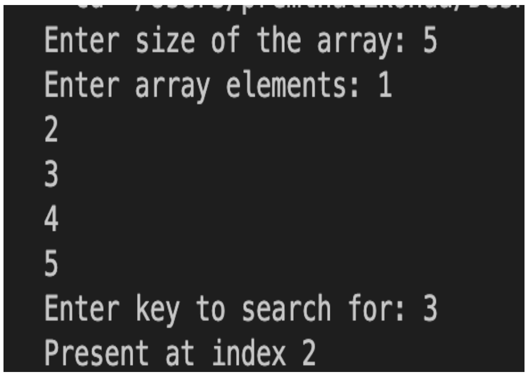
- Start with a sorted array.
- Compare the target with the middle element.
- If found, return the index at which the element is present.
- If less, search left; if greater, search right.
- Repeat until found or interval is empty.

Code:

```
#include <iostream>
using namespace std;

int binary(int arr[], int n, int key){
   int s = 0;
   int e = n - 1;
   int mid;

   while(s <= e){
```

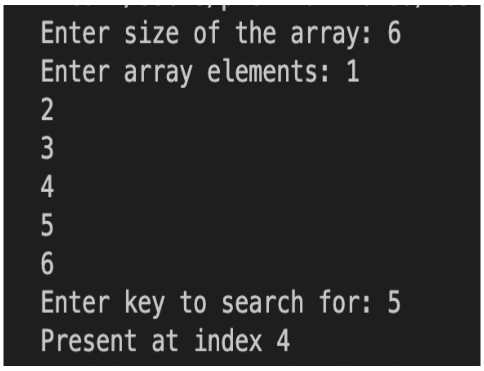
```cpp
        mid = (s+e) / 2;
        if(arr[mid] == key){
            return mid;
        }
        else if(arr[mid] > key){
            e = mid - 1;
        }
        else if(key > arr[mid]){
            s = mid + 1;
        }
    }
    return -1;
}

int main() {
    int size;
    cout << "Enter size of the array: ";
    cin >> size;

    int array[size];
    cout << "Enter array elements: ";
    for(int i = 0; i < size; i++){
        cin >> array[i];
    }

    int key;
    cout << "Enter key to search for: ";
    cin >> key;

    int index = binary(array,size,key);
    if(index != -1){
        cout << "Present at index " << index << endl;
    }
    else{
        cout << "Not present in array." << endl;
    }



    return 0;
}
```

Output: (screenshot)

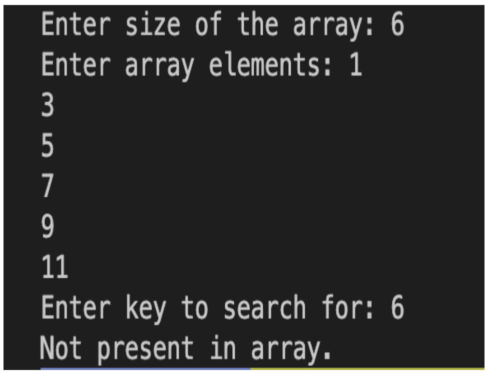


```
Enter size of the array: 5
Enter array elements: 1
2
3
4
5
Enter key to search for: 3
Present at index 2
```

Test Case: Any two (screenshot)



```
Enter size of the array: 6
Enter array elements: 1
2
3
4
5
6
Enter key to search for: 5
Present at index 4
```



```
Enter size of the array: 6
Enter array elements: 1
3
5
7
9
11
Enter key to search for: 6
Not present in array.
```

Conclusion:

The above code searches for the key using the binary search function and outputs whether the key is present in the array and its index if found. Overall, the code efficiently demonstrates the binary search algorithm for finding elements in a sorted array.

Name of Student: Prem Thatikonda

Roll Number: 06

Experiment No: 14

Title:

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

Theory:

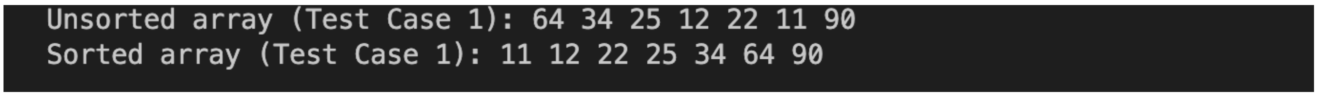
- Start from the beginning of the array.
- Compare adjacent elements; if out of order, swap.
- Repeat until no more swaps are needed, indicating the array is sorted.

- For descending order, reverse the comparison operator.

Code:

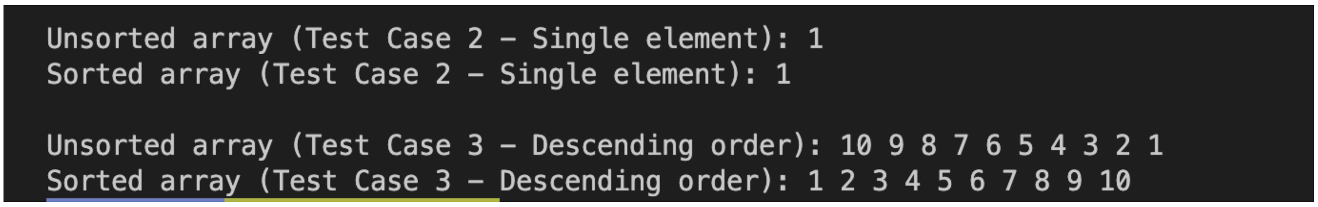
```
#include <iostream>

using namespace std;

// Function to swap two elements
void swap(int *a, int *b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}

// Function to implement bubble sort
void bubbleSort(int arr[], int n) {
 // Flag to track if any swaps occurred
 bool swapped;

 for (int i = 0; i < n - 1; i++) {
   swapped = false;
   for (int j = 0; j < n - i - 1; j++) {
     if (arr[j] > arr[j + 1]) {
       swap(&arr[j], &arr[j + 1]);
       swapped = true;
     }
   }

   // If we haven't needed any swaps in this pass, the array is already sorted
   if (!swapped) {
     break;
   }
 }
}

// Function to print the array
void printArray(int arr[], int n) {
 for (int i = 0; i < n; ++i) {
   cout << arr[i] << " ";
 }
 cout << "\n";
}

int main() {
 int arr1[] = {64, 34, 25, 12, 22, 11, 90};
 int arr2[] = {1};
 int arr3[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

 int n1 = sizeof(arr1) / sizeof(arr1[0]);
 int n2 = sizeof(arr2) / sizeof(arr2[0]);
 int n3 = sizeof(arr3) / sizeof(arr3[0]);

 cout << "Unsorted array (Test Case 1): ";
 printArray(arr1, n1);

 bubbleSort(arr1, n1);

 cout << "Sorted array (Test Case 1): ";
 printArray(arr1, n1);

 cout << "\nUnsorted array (Test Case 2 - Single element): ";
 printArray(arr2, n2);

 bubbleSort(arr2, n2);

 cout << "Sorted array (Test Case 2 - Single element): ";
 printArray(arr2, n2);

 cout << "\nUnsorted array (Test Case 3 - Descending order): ";
 printArray(arr3, n3);

 bubbleSort(arr3, n3);

 cout << "Sorted array (Test Case 3 - Descending order): ";
 printArray(arr3, n3);

 return 0;
}
```

Output: (screenshot)



```
Unsorted array (Test Case 1): 64 34 25 12 22 11 90
Sorted array (Test Case 1): 11 12 22 25 34 64 90
```

Test Case: Any two (screenshot)



```
Unsorted array (Test Case 2 - Single element): 1
Sorted array (Test Case 2 - Single element): 1

Unsorted array (Test Case 3 - Descending order): 10 9 8 7 6 5 4 3 2 1
Sorted array (Test Case 3 - Descending order): 1 2 3 4 5 6 7 8 9 10
```

Conclusion:

Hence by performing a repetitive comparing and swapping, the unsorted arrays have been sorted into a sorted manner by using bubble sort algorithm.