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

DATA STRUCTURES & ALGORITHMS PROGRAMMING LAB

Prepared by:

Name of Student : Sahil Sable

Roll No: 10

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. __________________Sahil Sable_________________ Roll No. _____10______ Semester __2___ of B.Tech Computer Science & Engineering, ITM Skills University, Kharghar, Navi Mumbai , has completed the term work satisfactorily in subject _____Data Structures and Algorithms________ for the academic year 2023 - 2027 as prescribed in the curriculum.

Place: Kharghar

Date: _05/04/2024_

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	05/04/24	
2	Implement Stack ADT using array.	05/04/24	
3	Convert an Infix expression to Postfix expression using stack ADT.	05/04/24	
4	Evaluate Postfix Expression using Stack ADT.	05/04/24	
5	Implement Linear Queue ADT using array.	05/04/24	
6	Implement Circular Queue ADT using array.	05/04/24	
7	Implement Singly Linked List ADT.	05/04/24	
8	Implement Circular Linked List ADT.	05/04/24	
9	Implement Stack ADT using Linked List	05/04/24	
10	Implement Linear Queue ADT using Linked List	05/04/24	
11	Implement Binary Search Tree ADT using Linked List.	05/04/24	
12	Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search	05/04/24	
13	Implement Binary Search algorithm to search an element in an array	05/04/24	
14	Implement Bubble sort algorithm to sort elements of an array in ascending and descending order	05/04/24	

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

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

Theory: Array is a collection of elements of similar data types and has a fixed size. We can access an element of the array through it's index. Indexing starts from 0 till n-1(where n=size of array). An element can be inserted in the array by shifting all the elements of the array to the right and making space for the element. Similarly, to delete an element, we need to shift all the elements from the right of the deleted element to the left side in order to overwrite the deleted element. In order to search for an element, we need to traverse through the array and print the appropriate message if the element is found or not.

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

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

const int MAX_SIZE = 100;

class array_operations {
private:
    int arr[MAX_SIZE];
    int size;

public:
    array_operations() {
        size = 0;
    }
    array_operations(int initial_array[], int initial_size) {
        if (initial_size <= MAX_SIZE) {
            for (int i = 0; i < initial_size; ++i) {
                arr[i] = initial_array[i];
            }
            size = initial_size;
        } else {
            cerr << "Initial array size exceeds maximum size." <<
endl;
        }
    }
    void display_array() {
        if (size == 0) {
            cout << "Array is empty." << endl;
            return;
```

```
        }
        cout << "Array elements: ";
        for (int i = 0; i < size; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    void insert_at_end(int element) {
        if (size < MAX_SIZE) {
            arr[size++] = element;
            cout << "Element inserted at the end successfully." <<
endl;
        } else {
            cout << "Array is full. Cannot insert element." <<
endl;
        }
    }
    void insert_at_beginning(int element) {
        if (size < MAX_SIZE) {
            for (int i = size; i > 0; i--) {
                arr[i] = arr[i - 1];
            }
            arr[0] = element;
            size++;
            cout << "Element inserted at the beginning
successfully." << endl;
        } else {
            cout << "Array is full. Cannot insert element." <<
endl;
        }
    }
    void insert_before_element(int element, int target) {
        int index = -1;
        for (int i = 0; i < size; i++) {
            if (arr[i] == target) {
                index = i;
                break;
            }
        }
        if (index != -1) {
            if (size < MAX_SIZE) {
                for (int i = size; i > index; i--) {
                    arr[i] = arr[i - 1];
                }
                arr[index] = element;
                size++;
                cout << "Element inserted before " << target << "
successfully." << endl;
            } else {
```

```
                cout << "Array is full. Cannot insert element." <<
endl;
            }
        } else {
            cout << "Element not found in the array." << endl;
        }
    }
    void insert_after_element(int element, int target) {
        int index = -1;
        for (int i = 0; i < size; i++) {
            if (arr[i] == target) {
                index = i;
                break;
            }
        }
        if (index != -1) {
            if (size < MAX_SIZE) {
                for (int i = size; i > index + 1; i--) {
                    arr[i] = arr[i - 1];
                }
                arr[index + 1] = element;
                size++;
                cout << "Element inserted after " << target << " 
successfully." << endl;
            } else {
                cout << "Array is full. Cannot insert element." <<
endl;
            }
        } else {
            cout << "Element not found in the array." << endl;
        }
    }
    void delete_at_end() {
        if (size > 0) {
            size--;
            cout << "Element deleted from the end successfully." <<
endl;
        } else {
            cout << "Array is empty. Cannot delete element." <<
endl;
        }
    }
    void delete_at_beginning() {
        if (size > 0) {
            for (int i = 0; i < size - 1; i++) {
                arr[i] = arr[i + 1];
            }
            size--;
```

```
            cout << "Element deleted from the beginning 
successfully." << endl;
        } else {
            cout << "Array is empty. Cannot delete element." <<
endl;
        }
    }
    void delete_before_element(int target) {
        int index = -1;
        for (int i = 0; i < size; i++) {
            if (arr[i] == target) {
                index = i;
                break;
            }
        }
        if (index != -1 && index > 0) {
            for (int i = index - 1; i < size - 1; i++) {
                arr[i] = arr[i + 1];
            }
            size--;
            cout << "Element deleted before " << target << " 
successfully." << endl;
        } else {
            cout << "Element not found in the array or no element 
before it." << endl;
        }
    }
    void delete_after_element(int target) {
        int index = -1;
        for (int i = 0; i < size; i++) {
            if (arr[i] == target) {
                index = i;
                break;
            }
        }
        if (index != -1 && index < size - 1) {
            for (int i = index + 1; i < size - 1; i++) {
                arr[i] = arr[i + 1];
            }
            size--;
            cout << "Element deleted after " << target << " 
successfully." << endl;
        } else {
            cout << "Element not found in the array or no element 
after it." << endl;
        }
    }
    void search_element(int element) {
        bool found = false;
```

```
        for (int i = 0; i < size; i++) {
            if (arr[i] == element) {
                found = true;
                break;
            }
        }
        if (found) {
            cout << "Element " << element << " found in the array."
<< endl;
        } else {
            cout << "Element " << element << " not found in the
array." << endl;
        }
    }
    void count_elements() {
        cout << "Number of elements in the array: " << size <<
endl;
    }
};

int main() {
    int initial_array[] = {1, 2, 3, 4, 5};
    int initial_size = sizeof(initial_array) /
sizeof(initial_array[0]);

    array_operations array(initial_array, initial_size);
    int choice, element, target;
    do {
        cout << "\n---Menu Driven Program---" << endl;
        cout << "1) Display Array" << endl;
        cout << "2) Insert at End" << endl;
        cout << "3) Insert at Beginning" << endl;
        cout << "4) Insert Before an Element" << endl;
        cout << "5) Insert After an Element" << endl;
        cout << "6) Delete at End" << endl;
        cout << "7) Delete at Beginning" << endl;
        cout << "8) Delete Before an Element" << endl;
        cout << "9) Delete After an Element" << endl;
        cout << "10) Search an Element" << endl;
        cout << "11) Count Number of Elements" << endl;
        cout << "What do you want to perform :- ";
        cin >> choice;

        switch (choice) {
            case 1:
                array.display_array();
                break;
            case 2:
                cout << "Enter element to insert at end: ";
```

```
                cin >> element;
                array.insert_at_end(element);
                array.display_array();
                break;
            case 3:
                cout << "Enter element to insert at beginning: ";
                cin >> element;
                array.insert_at_beginning(element);
                array.display_array();
                break;
            case 4:
                cout << "Enter element to insert: ";
                cin >> element;
                cout << "Enter element before which to insert: ";
                cin >> target;
                array.insert_before_element(element, target);
                array.display_array();
                break;
            case 5:
                cout << "Enter element to insert: ";
                cin >> element;
                cout << "Enter element after which to insert: ";
                cin >> target;
                array.insert_after_element(element, target);
                array.display_array();
                break;
            case 6:
                array.delete_at_end();
                array.display_array();
                break;
            case 7:
                array.delete_at_beginning();
                array.display_array();
                break;
            case 8:
                cout << "Enter element before which to delete: ";
                cin >> target;
                array.delete_before_element(target);
                array.display_array();
                break;
            case 9:
                cout << "Enter element after which to delete: ";
                cin >> target;
                array.delete_after_element(target);
                array.display_array();
                break;
            case 10:
                cout << "Enter element to search: ";
                cin >> element;
```

```
                array.search_element(element);
                array.display_array();
                break;
            case 11:
                array.count_elements();
                break;
            default:
                cout << "Invalid choice. Please try again." <<
endl;
        }
    } while (true);
    return 0;
}
```

Output: (screenshot)

```
---Menu Driven Program---
1) Display Array
2) Insert at End
3) Insert at Beginning
4) Insert Before an Element
5) Insert After an Element
6) Delete at End
7) Delete at Beginning
8) Delete Before an Element
9) Delete After an Element
10) Search an Element
11) Count Number of Elements
What do you want to perform :-
```

Test Case: Any two (screenshot)

```
---Menu Driven Program---
1) Display Array
2) Insert at End
3) Insert at Beginning
4) Insert Before an Element
5) Insert After an Element
6) Delete at End
7) Delete at Beginning
8) Delete Before an Element
9) Delete After an Element
10) Search an Element
11) Count Number of Elements
What do you want to perform :- 3
Enter element to insert at beginning: 10
Element inserted at the beginning successfully.
Array elements: 10 1 2 3 4 5

---Menu Driven Program---
1) Display Array
2) Insert at End
3) Insert at Beginning
4) Insert Before an Element
5) Insert After an Element
6) Delete at End
7) Delete at Beginning
8) Delete Before an Element
9) Delete After an Element
10) Search an Element
11) Count Number of Elements
What do you want to perform :-
```

Conclusion: Therefore, using switch cases, we can perform multiple operations like insertion, deletion, and searching for an element in an array through traversal using index.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Stack ADT using Array.

Theory: Array is a collection of elements of similar data types and has a fixed size. We can access an element of the array through it's index. Indexing starts from 0 till n-1(where n=size of array).

Stack is an Abstract Data Type which can be implemented using Linked List or Array. It consists of a variable named Top which points to the topmost element of the stack. Stack follows LIFO principle(Last In, First Out) which means that the element which is inserted last will be deleted first. There are three operations in Stack: Push- insertion from top, Pop- deletion from top, Peek- returning the topmost element from the stack.

Code:

```
// stack operations(array)
#include <iostream>
using namespace std;

int main()
{
    int top = -1, element, op,n;
    cout<<"Enter size of stack: ";
    cin>>n;
    int stack[n];
    while (true)
    {
        cout << "\nStack operation:
\n1.Push\n2.Pop\n3.Peek\n4.Exit\n";
        cin >> op;
        switch (op)
        {
```

```
        case 1:
            if (top == n - 1)
            {
                cout << "Stack is full. Cannot add more elements.
\n";
                break;
            }
            else
            {
                cout << "Enter element: ";
                cin >> element;
                top++;
                stack[top] = element;
                cout << "Element added in stack\n";
            }
            break;
        case 2:
            if (top == -1)
            {
                cout << "Stack is empty.\n";
                break;
            }
            else
            {
                cout << stack[top] << " is popped from stack\n";
                top--;
            }
            break;
        case 3:
            if (top == -1)
            {
                cout << "Stack is empty.\n";
                break;
            }
            else
            {
                cout << "Top element: " << stack[top] << "\n";
            }
            break;
        case 4:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Wrong choice\n";
            break;
        }
    }
    return 0;
}
```

Output: (screenshot)

```
Enter size of stack: 5

Stack operation:
1.Push
2.Pop
3.Peek
4.Exit
1
Enter element: 8
Element added in stack

Stack operation:
1.Push
2.Pop
3.Peek
4.Exit
2
8 is popped from stack

Stack operation:
1.Push
2.Pop
3.Peek
4.Exit
```

Test Case: Any two (screenshot) Above

Conclusion: Therefore, using switch cases, we can perform multiple operations like push, pop, and peek in a stack using array.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

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

Theory: Stack is an Abstract Data Type which can be implemented using Linked List or Array. It consists of a variable named Top which points to the topmost element of the stack. Stack follows LIFO principle(Last In, First Out) which means that the element which is inserted last will be deleted first. There are three operations in Stack: Push- insertion from top, Pop- deletion from top, Peek- returning the topmost element from the stack. Using stack, we can convert an infix expression to postfix expression by pushing the operators and brackets in the stack and the operands to the expression and popping the elements to the expression through operator precedence after encountering a closing bracket.

Code:

```
// conversion of infix to postfix expression using stack(array)
#include <iostream>
using namespace std;

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

int main()
{
    string exp, result = "";
    char stack[100];
    int top = -1;
    cout << "Enter infix expression: ";
    getline(cin, exp);
```

```
    int n = exp.length();
    char express[n + 2];
    express[0] = '(';
    for (int i = 0; i < n; i++)
    {
        express[i + 1] = exp[i];
    }
    express[n + 1] = ')';
    for (int i = 0; i < n + 2; i++)
    {
        if (express[i] == '(')
        {
            top++;
            stack[top] = express[i];
        }
        else if (express[i] == ')')
        {
            while (stack[top] != '(' && top > -1)
            {
                result += stack[top];
                top--;
            }
            top--;
        }
        else if ((express[i] >= 'a' && express[i] <= 'z') ||
(express[i] >= 'A' && express[i] <= 'Z') || (express[i] >= '0' &&
express[i] <= '9'))
        {
            result += express[i];
        }
        else
        {
            while (top > -1 && precedence(stack[top]) >=
precedence(express[i]))
            {
                result += stack[top];
                top--;
            }
            top++;
            stack[top] = express[i];
        }
    }
    while (top > -1)
    {
        result += stack[top];
        top--;
    }
    cout << "Postflix Result: " << result << endl;
    return 0;
```

```
}
```

Output: (screenshot)

Test Case: Any two (screenshot)

```
Enter infix expression: a+b-c*d/e
Postflix Result: ab+cd*e/-
```

Conclusion: Therefore, using stack ADT, we can convert infix expression to postfix expression by operations like Push and Pop.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Evaluate Postfix expression using Stack ADT.

Theory: Stack is an Abstract Data Type which can be implemented using Linked List or Array. It consists of a variable named Top which points to the topmost element of the stack. Stack follows LIFO principle(Last In, First Out) which means that the element which is inserted last will be deleted first. There are three operations in Stack: Push- insertion from top, Pop- deletion from top, Peek- returning the topmost element from the stack. Using stack, we can evaluate a postfix expression by pushing the operands in the stack and popping them and evaluating them when an operator is encountered and popping the result back in the stack and printing the topmost element after the whole expression is evaluated.

Code:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string expression;
    char stack[100];
    int stack1[100];
    int top = -1, a, b, result = 0;
    cout << "Enter postfix expression: ";
    getline(cin, expression);
    for (int i = 0; i < expression.length(); i++)
    {
        stack[i] = expression[i];
    }
    stack[expression.length()] = ')';
    int i = 0;
    while (stack[i] != ')')
    {
        if (stack[i] == '*' || stack[i] == '/' || stack[i] == '%'
|| stack[i] == '-' || stack[i] == '+')
        {
            a = stack1[top];
            top--;
            b = stack1[top];
            top--;
            if (stack[i] == '*')
            {
                result = b * a;
            }
            else if (stack[i] == '/')
            {
                if (a != 0)
                {
                    result = b / a;
                }
                else
                {
                    cout << "Error: Division by zero." << endl;
                    return 1;
                }
            }
            else if (stack[i] == '%')
            {
                result = b % a;
            }
```

```
            else if (stack[i] == '+')
            {
                result = b + a;
            }
            else
            {
                result = b - a;
            }
            top++;
            stack1[top] = result;
        }
        else
        {
            top++;
            stack1[top] = int(stack[i]) - 48;
        }
        i++;
    }
    cout << "Result: " << stack1[top] << endl;
    return 0;
}
```

Output: (screenshot)

```
Enter postfix expression: 53+4*
Result: 32
```

Test Case: Any two (screenshot)

Conclusion: Therefore, using stack ADT, we can evaluate a postfix expression by operations like Push and Pop.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Linear Queue ADT using array.

Theory: Array is a collection of elements of similar data types and has a fixed size. We can access an element of the array through it's index. Indexing starts from 0 till n-1(where n=size of array).

Queue is an Abstract Data Type which can be implemented using Linked List or Array. It consists of two variables named Front and Rear which point to the first and last elements of the stack, respectively. Queue follows FIFO principle(First In, First Out) which means that the element which is inserted first will be deleted first. There are three operations in Stack: Enqueue- insertion from rear, Dequeue- deletion from front, Peek- returning the frontmost element from the queue.

Code:

```
#include <iostream>
using namespace std;
int main()
{
    int front = -1, rear = -1, choice, element, n;
    cout << "Enter size of queue: ";
    cin >> n;
    int queue[n];
    while (true)
    {
        cout << "\nQueue Operation:
\n1.Enqueue\n2.Dequeue\n3.Peek\n4.Exit\n";
        cin >> choice;
        switch (choice)
```

```
        {
        case 1:
            if (rear == n - 1)
            {
                cout << "Queue is full. Cannot add more elements.
\n";
            }
            else
            {
                cout << "Enter element: ";
                cin >> element;
                if (front == -1 && rear == -1)
                {
                    front = 0, rear = 0;
                }
                else
                {
                    rear++;
                }
                queue[rear] = element;
                cout << "Element added successfully.\n";
            }
            break;
        case 2:
            if (front == -1 || front > rear)
            {
                cout << "Queue is empty. Cannot delete more 
elements.\n";
            }
            else
            {
                element = queue[front];
                front++;
                cout << "Element " << element << " removed 
successfully.\n";
            }
            break;
        case 3:
            if (front == -1)
            {
                cout << "Queue is empty.\n";
            }
            else
            {
                cout << "Front element: " << queue[front] << endl;
            }
            break;
        case 4:
            cout << "Exiting...\n";
```

```
            return 0;
        default:
            cout << "Wrong choice.\n";
            break;
        }
    }
    return 0;
}
```

Output: (screenshot)

```
Enter size of queue: 5

Queue Operation:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
1
Enter element: 0
Element added successfully.

Queue Operation:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
1
Enter element: 2
Element added successfully.

Queue Operation:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
3
Front element: 0
```

Test Case: Any two (screenshot)

Conclusion: Therefore, using array, we can implement a linear queue and perform operations like Enqueue, Dequeue and Peek.

Title: Implement Circular Queue ADT using array.

Theory: Array is a collection of elements of similar data types and has a fixed size. We can access an element of the array through it's index. Indexing starts from 0 till n-1(where n=size of array).

Queue is an Abstract Data Type which can be implemented using Linked List or Array. It consists of two variables named Front and Rear which point to the first and last elements of the stack, respectively. Queue follows FIFO principle(First In, First Out) which means that the element which is inserted first will be deleted first. There are three operations in Stack: Enqueue- insertion from rear, Dequeue- deletion from front, Peek- returning the frontmost element from the queue. As size of array is fixed, in order to overcome the challenges, we can move the rear pointer to the start of the array if rear=n-1 and front is not at first index, so we can continue to insert elements.

Code:

```
#include <iostream>
using namespace std;
int main()
{
    int element, front = -1, rear = -1, n, choice;
    cout << "Enter size of queue: ";
    cin >> n;
```

```
    int queue[n];
    while (true)
    {
        cout << "\nCircular queue operations:\n1.Enqueue\n2.Dequeue\n3.Peek\n4.Exit\n";
        cin >> choice;
        switch (choice)
        {
        case 1:
            if ((front == 0 && rear == n - 1) || rear == front - 1)
            {
                cout << "Queue is full. Cannot add more elements.\n";
            }
            else
            {
                if (rear == n - 1 && front != 0)
                {
                    rear = (rear + 1) % n;
                }
                else if (front == -1 && rear == -1)
                {
                    front++;
                    rear++;
                }
                else
                {
                    rear++;
                }
                cout << "Enter element: ";
                cin >> element;
                queue[rear] = element;
                cout << "Element added successfully.\n";
            }
            break;
        case 2:
            if (front == -1)
            {
                cout << "Queue is empty. Cannot delete elements.\n";
            }
            else
            {
                element = queue[front];
                if (front == rear)
                {
                    front = -1;
                    rear = -1;
                }
```

```cpp
                else if (front == n - 1)
                {
                    front = (front + 1) % n;
                }
                else
                {
                    front++;
                }
                cout << "Element " << element << " is popped from the queue.\n";
            }
            break;
        case 3:
            if (front == -1)
            {
                cout << "Queue is empty.\n";
            }
            else
            {
                cout << "Top element: " << queue[front] << endl;
            }
            break;
        case 4:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Wrong input\n";
            break;
        }
    }
    return 0;
}
```

Output: (screenshot)

```
Enter size of queue: 5

Circular queue operations:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
1
Enter element: 1
Element added successfully.

Circular queue operations:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
1
Enter element: 2
Element added successfully.

Circular queue operations:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
2
Element 1 is popped from the queue.

Circular queue operations:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
```

Test Case: Any two (screenshot)

Conclusion: Therefore, using array, we can implement a circular queue and perform operations like Enqueue, Dequeue and Peek without being constrained by the limitation of the fixed size of the array.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Singly Linked List ADT.

Theory: Linked List is a data type which consists of nodes which contain data and a next pointer which points to the next node in the list. It stores the address of the next node. There is a start pointer in stack memory which points to the first node in the heap memory. It utilises dynamic memory and allocates heap memory to the nodes in the list. The last node's next pointer has NULL value to indicate it's the last node in the list.

Code:

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

class Node
{
public:
    int data;
    Node *next;
    Node()
    {
        cout << "Enter data: ";
        cin >> data;
        next = NULL;
    }
};

Node *createList(int n) // function to create a linked list
{
    Node *start = NULL;
    Node *ptr = NULL;
    for (int i = 0; i < n; i++)
    {
        Node *new_node = new Node();
        if (start == NULL)
        {
            start = new_node;
            ptr = start;
        }
        else
        {
            ptr->next = new_node;
            ptr = new_node;
        }
    }
    return start;
}
```

```
void insertAtStart(Node *&a) // function to insert a node at the start of list
{
    Node *new_node = new Node();
    if (new_node == NULL)
    {
        cout << "Overflow";
        return;
    }
    else
    {
        new_node->next = a;
        a = new_node;
    }
}

void insertAtEnd(Node *&a) // function to insert a node at the end of list
{
    Node *new_node = new Node();
    Node *ptr = a;
    while (ptr->next != NULL)
    {
        ptr = ptr->next;
    }
    ptr->next = new_node;
    new_node->next = NULL;
}

void insertAfterElement(Node *&a) // function to insert a node after a particular element in the list
{
    int n;
    cout << "Enter element after which to add a node: ";
    cin >> n;
    Node *new_node = new Node();
    Node *ptr = a;
    Node *preptr = ptr;
    if (new_node == NULL)
    {
        cout << "Overflow" << endl;
        return;
    }
    else
    {
        while (preptr->data != n)
        {
            preptr = ptr;
            ptr = ptr->next;
```

```
        }
        if (ptr == NULL)
        {
            cout << "No element found" << endl;
        }
        else if (ptr == a)
        {
            new_node->next = ptr->next;
            a->next = new_node;
        }
        else
        {
            new_node->next = ptr;
            preptr->next = new_node;
        }
    }
}

void insertBeforeElement(Node *&a) // function to insert a node 
before a particular element in the list
{
    Node *ptr = a;
    Node *preptr = ptr;
    int b;
    cout << "Enter element to add a node before it: ";
    cin >> b;
    Node *new_node = new Node();
    if (new_node == NULL)
    {
        cout << "Overflow" << endl;
        return;
    }
    else
    {
        while (ptr->data != b)
        {
            preptr = ptr;
            ptr = ptr->next;
        }
        if (ptr == NULL)
        {
            cout << "No element found" << endl;
        }
        else if (ptr == a)
        {
            new_node->next = ptr;
            a = new_node;
        }
        else
```

```
            {
                preptr->next = new_node;
                new_node->next = ptr;
            }
        }
}

void deleteFirstNode(Node *&a) // function to delete the first node in the list
{
    Node *ptr = a;
    if (a == NULL)
    {
        cout << "Underflow" << endl;
        return;
    }
    else
    {
        a = ptr->next;
        delete ptr;
    }
}

void deleteLastNode(Node *&a) // function to delete the last node in the list
{
    Node *ptr = a;
    Node *preptr = ptr;
    if (a == NULL)
    {
        cout << "Underflow" << endl;
        return;
    }
    else
    {
        while (ptr->next != NULL)
        {
            preptr = ptr;
            ptr = ptr->next;
        }
        if (preptr == ptr) // if there is only one node, delete it and make the start pointer NULL
        {
            delete ptr;
            a = NULL;
        }
        else
        {
            preptr->next = NULL;
```

```
            delete ptr;
        }
    }
}

void deleteBeforeElement(Node *&a) // function to delete the node 
before a particular element in the list
{
    Node *ptr = a;
    Node *preptr = NULL;
    Node *temp = a;
    int b;
    cout << "Enter element to delete a node before it: ";
    cin >> b;
    if (a == NULL)
    {
        cout << "Underflow" << endl;
        return;
    }
    else
    {
        while (ptr->data != b)
        {
            temp = preptr;
            preptr = ptr;
            ptr = ptr->next;
            if (ptr == NULL)
            {
                cout << "Element not found" << endl;
                return;
            }
        }
        if (preptr == NULL) // if first node is selected, print 
error message
        {
            cout << "Element not found" << endl;
        }
        else
        {
            if (preptr == a)
            {
                a = ptr;
            }
            else
            {
                temp->next = ptr;
            }
            delete preptr;
        }
```

```
    }
}

void deleteAfterElement(Node *&a) // function to delete the node
after a particular element in the list
{
    Node *ptr = a;
    Node *preptr = a;
    Node *temp = NULL;
    int b;
    cout << "Enter element to delete node after: ";
    cin >> b;
    ptr = a;
    if (a == NULL)
    {
        cout << "Underflow" << endl;
        return;
    }
    else
    {
        while (ptr->data != b)
        {
            preptr = ptr;
            ptr = ptr->next;
            if (ptr == NULL)
            {
                cout << "Element not found" << endl;
                return;
            }
        }
        if (ptr == NULL)
        {
            cout << "Element not found" << endl;
        }
        else
        {
            if (ptr->next == NULL)
            {
                cout << "No element to delete" << endl;
            }
            else
            {
                preptr = ptr;
                temp = ptr->next;
                preptr->next = temp->next;
                delete temp;
            }
        }
    }
```

```
}

void searchElement(Node *a, int b) // function to search for a particular element in the list
{
    Node *ptr = a;
    Node *pos = NULL;
    while (ptr != NULL)
    {
        if (ptr->data == b)
        {
            pos = ptr;
            break;
        }
        else
        {
            ptr = ptr->next;
        }
    }
    if (pos == NULL)
    {
        cout << "Element not found" << endl;
    }
    else
    {
        cout << "Element " << pos->data << " found at " << pos << endl;
    }
}

void showList(Node *a) // function to display the list and print number of nodes in the list
{
    int count = 0;
    Node *ptr = a;
    while (ptr != NULL)
    {
        cout << ptr->data << " ";
        ptr = ptr->next;
        count++;
    }
    cout << endl
         << "Number of nodes: " << count << endl;
}

void deleteList(Node *&a) // function to delete the list
{
    Node *ptr = a;
    Node *temp = NULL;
```

```
    while (ptr != NULL)
    {
        temp = ptr;
        ptr = ptr->next;
        delete temp;
    }
    a = NULL;
}

int main()
{
    int n;
    cout << "Enter number of nodes: ";
    cin >> n;
    Node *start = createList(n);
    int choice;
    char ans = 'y';
    do
    {
        cout << "Enter your choice: \n1.Insert a node at beginning\n2.Insert a node at end\n3.Search the list for an element\n4.Insert a node after an element\n5.Insert a node before an element\n6.Delete first node\n7.Delete last node\n8.Delete a node after a particular element\n9.Delete a node before a particular element\n10.Show list\n11.Exit\n";
        cin >> choice;
        switch (choice)
        {
        case 1:
            insertAtStart(start);
            break;
        case 2:
            insertAtEnd(start);
            break;
        case 3:
            int element;
            cout << "Enter the element to search for: ";
            cin >> element;
            searchElement(start, element);
            break;
        case 4:
            insertAfterElement(start);
            break;
        case 5:
            insertBeforeElement(start);
            break;
        case 6:
            deleteFirstNode(start);
            break;
```

```cpp
        case 7:
            deleteLastNode(start);
            break;
        case 8:
            deleteAfterElement(start);
            break;
        case 9:
            deleteBeforeElement(start);
            break;
        case 10:
            showList(start);
            break;
        case 11:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Wrong choice" << endl;
        }
        cout << "Do you want to continue? (y/n): ";
        cin >> ans;
    } while (ans == 'y');
    deleteList(start);
    return 0;
}
```

Output: (screenshot)

```
Enter number of nodes: 5
Enter data: 1
Enter data: 3
Enter data: 5
Enter data: 7
Enter data: 9
Enter your choice:
1.Insert a node at beginning
2.Insert a node at end
3.Search the list for an element
4.Insert a node after an element
5.Insert a node before an element
6.Delete first node
7.Delete last node
8.Delete a node after a particular element
9.Delete a node before a particular element
10.Show list
11.Exit
```

Test Case: Any two (screenshot)

Conclusion: Therefore, we can implement a linked list by using class or structure and allocate heap memory for the node by using new operator or malloc function. We can deallocate memory for the node by using free function or delete operator.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Circular Linked List ADT.

Theory: Linked List is a data type which consists of nodes which contain data and a next pointer which points to the next node in the list. It stores the address of the next node. There is a start pointer in stack memory which points to the first node in the heap memory. It utilises dynamic memory and allocates heap memory to the nodes in the list. The last node's next pointer has the address of first node, hence it's called circular linked list.

Code:

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

class Node
{
public:
    int data;
    Node *next;
    Node()
    {
        cout << "Enter data: ";
        cin >> data;
        next = NULL;
    }
};
```

```
Node *createList(int n)
{
    Node *start = NULL;
    Node *ptr = start;
    for (int i = 0; i < n; i++)
    {
        Node *newNode = new Node();
        if (start == NULL)
        {
            start = newNode;
            ptr = newNode;
        }
        else
        {
            ptr->next = newNode;
            ptr = ptr->next;
        }
    }
    ptr->next = start;
    return start;
}

void searchElement(Node *&a)
{
    int element;
    cout << "Enter element to search: ";
    cin >> element;
    Node *ptr = a;
    Node *preptr = ptr;
    Node *temp = NULL;
    while (preptr->next != a)
    {
        if (ptr->data == element)
        {
            cout << "Element " << ptr->data << " found in node " <<
ptr << endl;
            temp = ptr;
            break;
        }
        preptr = ptr;
        ptr = ptr->next;
    }
    if (temp == NULL)
    {
        cout << "No element found" << endl;
    }
}
```

```cpp
void traverseList(Node *&a)
{
    int count = 0;
    Node *ptr = a;
    Node *preptr = ptr;
    cout << "Circular Linked List: " << endl;
    while (preptr->next != a)
    {
        cout << ptr->data << endl;
        preptr = ptr;
        ptr = ptr->next;
        count++;
    }
    cout << "Number of nodes: " << count << endl;
}

void insertAtBegin(Node *&a)
{
    Node *ptr = a;
    Node *newNode = new Node();
    if (newNode == NULL)
    {
        cout << "Overflow" << endl;
        return;
    }
    newNode->next = a;
    while (ptr->next != a)
    {
        ptr = ptr->next;
    }
    ptr->next = newNode;
    a = newNode;
}

void insertAtEnd(Node *&a)
{
    Node *ptr = a;
    Node *newNode = new Node();
    if (newNode == NULL)
    {
        cout << "Overflow" << endl;
        return;
    }
    newNode->next = a;
    while (ptr->next != a)
    {
        ptr = ptr->next;
    }
    ptr->next = newNode;
```

```
}

void insertBeforeElement(Node *&a)
{
    Node *ptr = a;
    Node *preptr = a;
    Node *newNode = new Node();
    if (newNode == NULL)
    {
        cout << "Overflow" << endl;
        return;
    }
    else
    {
        int element;
        cout << "Enter element to insert a node before it: ";
        cin >> element;
        if (ptr->data == element)
        {
            newNode->next = a;
            while (ptr->next != a)
            {
                ptr = ptr->next;
            }
            ptr->next = newNode;
            a = newNode;
            return;
        }
        else
        {
            do
            {
                if (ptr->data == element)
                {
                    preptr->next = newNode;
                    newNode->next = ptr;
                    return;
                }
                preptr = ptr;
                ptr = ptr->next;
            } while (ptr != a);
            if (ptr == a)
            {
                cout << "Element not found" << endl;
                return;
            }
        }
    }
}
```

```cpp
void insertAfterElement(Node *&a)
{
    Node *ptr = a;
    Node *preptr = ptr;
    Node *newNode = new Node();
    if (newNode == NULL)
    {
        cout << "Overflow" << endl;
        return;
    }
    else
    {
        int element;
        cout << "Enter element to insert a node after it: ";
        cin >> element;
        do
        {
            if (preptr->data == element)
            {
                if (preptr == a)
                {
                    newNode->next = a->next;
                    a->next = newNode;
                    return;
                }
                if (preptr->next == a)
                {
                    preptr->next = newNode;
                    newNode->next = a;
                    a = newNode;
                    return;
                }
                preptr->next = newNode;
                newNode->next = ptr;
                return;
            }
            preptr = ptr;
            ptr = ptr->next;
        } while (ptr != a);
        if (ptr == a)
        {
            cout << "Element not found" << endl;
            return;
        }
    }
}

void deleteAtBegin(Node *&a)
```

```
{
    Node *ptr = a;
    if (a == NULL)
    {
        cout << "Underflow" << endl;
        return;
    }
    while (ptr->next != a)
    {
        ptr = ptr->next;
    }
    Node *temp = a;
    ptr->next = temp->next;
    a = temp->next;
    delete temp;
}

void deleteAtEnd(Node *&a)
{
    Node *ptr = a;
    Node *preptr = ptr;
    if (a == NULL)
    {
        cout << "Underflow" << endl;
        return;
    }
    while (ptr->next != a)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = a;
    delete ptr;
}

void deleteBeforeElement(Node *&a)
{
    int element;
    cout << "Enter element to delete node before it: ";
    cin >> element;
    Node *ptr = a;
    Node *preptr = NULL;
    Node *temp = NULL;
    if (a == NULL)
    {
        cout << "Underflow" << endl;
        return;
    }
    else
```

```
    {
        if (element == a->data)
        {
            cout << "Cannot delete before first element" << endl;
            return;
        }
        else
        {
            do
            {
                if (ptr->data == element)
                {
                    if (temp == NULL)
                    {
                        ptr = a;
                        while (ptr->next != a)
                        {
                            ptr = ptr->next;
                        }
                        temp = a;
                        ptr->next = temp->next;
                        a = temp->next;
                        delete temp;
                        return;
                    }
                    temp->next = ptr;
                    delete preptr;
                }
                temp = preptr;
                preptr = ptr;
                ptr = ptr->next;
            } while (ptr != a);
            return;
            if (ptr == a)
            {
                cout << "Element not found" << endl;
                return;
            }
        }
    }
}

void deleteAfterElement(Node *&a)
{
    int element;
    cout << "Enter element to delete node after it: ";
    cin >> element;
    if (a == NULL)
    {
```

```
        cout << "Underflow" << endl;
        return;
    }
    Node *ptr = a;
    Node *preptr = NULL;
    do
    {
        if (ptr->data == element)
        {
            if (ptr->next == a)
            {
                Node *temp = ptr->next;
                ptr->next = temp->next;
                delete temp;
                a = ptr->next;
                return;
            }
            else
            {
                preptr = ptr;
                ptr = ptr->next;
                preptr->next = ptr->next;
                delete ptr;
                return;
            }
        }
        preptr = ptr;
        ptr = ptr->next;
    } while (ptr != a);
    return;
    cout << "Element not found" << endl;
}

void deleteList(Node *&a)
{
    Node *ptr = a;
    Node *preptr = ptr;
    while (ptr->next != a)
    {
        preptr = ptr;
        ptr = ptr->next;
        delete preptr;
    }
    a = NULL;
}

int main()
{
    int n;
```

```
    cout << "Enter number of nodes: ";
    cin >> n;
    Node *start = createList(n);
    int choice;
    char ans = 'y';
    do
    {
        cout << "Enter your choice: \n1.Insert a node at beginning\n2.Insert a node at end\n3.Search the list for an element\n4.Insert a node after an element\n5.Insert a node before an element\n6.Delete first node\n7.Delete last node\n8.Delete a node after a particular element\n9.Delete a node before a particular element\n10.Show list\n11.Exit\n";
        cin >> choice;
        switch (choice)
        {
        case 1:
            insertAtBegin(start);
            break;
        case 2:
            insertAtEnd(start);
            break;
        case 3:
            searchElement(start);
            break;
        case 4:
            insertAfterElement(start);
            break;
        case 5:
            insertBeforeElement(start);
            break;
        case 6:
            deleteAtBegin(start);
            break;
        case 7:
            deleteAtEnd(start);
            break;
        case 8:
            deleteAfterElement(start);
            break;
        case 9:
            deleteBeforeElement(start);
            break;
        case 10:
            traverseList(start);
            break;
        case 11:
            cout << "Exiting...\n";
            return 0;
```

```
        default:
            cout << "Wrong choice" << endl;
        }
        cout << "Do you want to continue? (y/n): ";
        cin >> ans;
    } while (ans == 'y');
    deleteList(start);
    return 0;
}
```

Output: (screenshot)

```
Enter number of nodes: 6
Enter data: 0
Enter data: 1
Enter data: 2
Enter data: 3
Enter data: 4
Enter data: 5
Enter your choice:
1.Insert a node at beginning
2.Insert a node at end
3.Search the list for an element
4.Insert a node after an element
5.Insert a node before an element
6.Delete first node
7.Delete last node
8.Delete a node after a particular element
9.Delete a node before a particular element
10.Show list
11.Exit
8
Enter element to delete node after it: 2
Do you want to continue? (y/n): y
Enter your choice:
1.Insert a node at beginning
2.Insert a node at end
3.Search the list for an element
4.Insert a node after an element
5.Insert a node before an element
6.Delete first node
7.Delete last node
8.Delete a node after a particular element
9.Delete a node before a particular element
10.Show list
11.Exit
10
Circular Linked List:
0
1
2
4
5
Number of nodes: 5
Do you want to continue? (y/n): n
```

Test Case: Any two (screenshot) Above

Conclusion: Therefore, we can implement a circular linked list by using class or structure and allocate heap memory for the node by using new operator or malloc function. We can deallocate memory for the node by using free function or delete operator.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Stack ADT using Linked List.

Theory: Stack is an Abstract Data Type which can be implemented using Linked List or Array. It consists of a variable named Top which points to the topmost element of the stack. Stack follows LIFO principle(Last In, First Out) which means that the element which is inserted last will be deleted first. There are three operations in Stack: Push- insertion from top, Pop- deletion from top, Peek- returning the topmost element from the stack. We can implement insertion at

beginning, deletion from beginning algorithms to implement Stack using Linked List.

Linked List is a data type which consists of nodes which contain data and a next pointer which points to the next node in the list. It stores the address of the next node. There is a start pointer in stack memory which points to the first node in the heap memory. It utilises dynamic memory and allocates heap memory to the nodes in the list. The last node's next pointer has the address of first node, hence it's called circular linked list.

Code:

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

class Node
{
public:
    int element;
    Node *next;
    Node()
    {
        cout << "Enter element: ";
        cin >> element;
        next = NULL;
    }
};

void pushList(Node *&a)
{
    Node *newnode = new Node();
    if (a == NULL)
    {
        a = newnode;
    }
    else
    {
        newnode->next = a;
        a = newnode;
    }
    cout << "Element pushed successfully\n";
}

void popList(Node *&a)
{
    Node *ptr = a;
    if (a == NULL)
```

```cpp
    {
        cout << "Stack is empty\n";
    }
    else
    {
        cout << "Element " << a->element << " popped 
successfully\n";
        a = a->next;
        delete ptr;
    }
}

void peekList(Node *&a)
{
    if (a == NULL)
    {
        cout << "Stack is empty\n";
    }
    else
    {
        cout << "Top element: " << a->element << endl;
    }
}

void deleteList(Node *&a)
{
    Node *ptr = a;
    Node *temp = ptr;
    if (a == NULL)
    {
        return;
    }
    else
    {
        while (ptr)
        {
            temp = ptr;
            ptr = ptr->next;
            delete temp;
        }
    }
    a = NULL;
}

void seeList(Node *&a)
{
    Node *ptr = a;
    if (a == NULL)
    {
```

```
            cout << "Empty stack\n";
        }
        else
        {
            while (ptr)
            {
                cout << ptr->element << endl;
                ptr = ptr->next;
            }
        }
}

int main()
{
    Node *top = NULL;
    int choice;
    while (true)
    {
        cout << "\nStack operation:
\n1.Push\n2.Pop\n3.Peek\n4.Exit\n";
        cin >> choice;
        switch (choice)
        {
        case 1:
            pushList(top);
            break;
        case 2:
            popList(top);
            break;
        case 3:
            peekList(top);
            break;
        case 4:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Wrong choice\n";
            break;
        }
    }
    deleteList(top);
    return 0;
}
```

Output: (screenshot)

```
Stack operation:
1.Push
2.Pop
3.Peek
4.Exit
1
Enter element: 7
Element pushed successfully

Stack operation:
1.Push
2.Pop
3.Peek
4.Exit
2
Element 7 popped successfully

Stack operation:
1.Push
2.Pop
3.Peek
4.Exit
```

Test Case: Any two (screenshot)

Conclusion: Therefore, we can implement Stack by linked list by using class or structure and allocate heap memory for the node by using new operator or malloc function. We can deallocate memory for the node by using free function or delete operator. We can implement push and pop operations through insertion at beginning and deletion from beginning algorithms.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Linear Queue ADT using Linked List.

Theory: Queue is an Abstract Data Type which can be implemented using Linked List or Array. It consists of two variables named Front and Rear which point to the first and last elements of the stack, respectively. Queue follows FIFO principle(First In, First Out) which means that the element which is inserted first will be deleted first. There are three operations in Stack: Enqueue- insertion from rear, Dequeue- deletion from front, Peek- returning the frontmost element from the queue. It can be implemented by insertion at end and deletion from beginning algorithms.

Linked List is a data type which consists of nodes which contain data and a next pointer which points to the next node in the list. It stores the address of the next node. There is a start pointer in stack memory which points to the first node in the heap memory. It utilises dynamic memory and allocates heap memory to the nodes in the list. The last node's next pointer has the address of first node, hence it's called circular linked list.

Code:

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

class Node
{
public:
    int data;
    Node *next;
    Node()
    {
        cout << "Enter data: ";
        cin >> data;
        next = NULL;
    }
};

void enqueue(Node *&start, Node *&end)
{
    Node *newnode = new Node();
    if (start == NULL)
    {
        start = newnode;
        end = newnode;
    }
```

```
    else
    {
        end->next = newnode;
        end = newnode;
    }
    cout << "Element added successfully.\n";
}

void dequeue(Node *&start, Node *&end)
{
    Node *ptr = NULL;
    if (start == NULL)
    {
        cout << "Queue is empty.\n";
        return;
    }
    else
    {
        ptr = start;
        start = start->next;
        cout << "Element " << ptr->data << " deleted successfully.
\n";
        delete ptr;
    }
}

void peek(Node *&start)
{
    if (start == NULL)
    {
        cout << "Queue is empty.\n";
        return;
    }
    else
    {
        cout << "Top element: " << start->data << endl;
    }
}

void deleteQueue(Node *&start)
{
    Node *ptr = start;
    Node *temp = NULL;
    if (start == NULL)
    {
        return;
    }
    else
    {
```

```
        while (ptr != NULL)
        {
            temp = ptr;
            ptr = ptr->next;
            delete temp;
        }
    }
    start = NULL;
}

void showQueue(Node *start)
{
    Node *ptr = start;
    while (ptr != NULL)
    {
        cout << ptr->data << endl;
        ptr = ptr->next;
    }
}

int main()
{
    Node *front = NULL;
    Node *rear = NULL;
    int choice;
    while (true)
    {
        cout << "\nQueue Operations:
\n1.Enqueue\n2.Dequeue\n3.Peek\n4.Exit\n";
        cin >> choice;
        switch (choice)
        {
        case 1:
            enqueue(front, rear);
            break;
        case 2:
            dequeue(front, rear);
            break;
        case 3:
            peek(front);
            break;
        case 4:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Wrong choice.\n";
            break;
        }
    }
```

```
    return 0;
}
```

Output: (screenshot)

```
Queue Operations:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
1
Enter data:
6
Element added successfully.

Queue Operations:
1.Enqueue
2.Dequeue
3.Peek
4.Exit
3
Top element: 6
```

Test Case: Any two (screenshot)

Conclusion: Therefore, we can implement Linear Queue by linked list by using class or structure and allocate heap memory for the node by using new operator or malloc function. We can deallocate memory for the node by using free function or delete operator. We can implement enqueue and dequeue operations through insertion at end and deletion from beginning algorithms.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Binary Search Tree ADT using Linked List.

Theory:

A binary tree is a non-linear data structure in which there is a root node and each parent node has 0,1 or 2 child nodes at most. In binary search tree, all the nodes having values less than that of the root node are present in the left subtree of the root node and all the nodes having values greater than or equal to that of the root node are present in the right subtree of the root node.

Code:

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

class Node
{
public:
    int data;
    Node *left;
    Node *right;
    Node()
    {
        cout << "Enter data: ";
        cin >> data;
        left = right = NULL;
    }
};

class BST
{
public:
    Node *root;
    BST()
    {
        root = NULL;
    }
```

```
    void insert(Node *node)
    {
        if (root == NULL)
        {
            root = node;
            return;
        }
        Node *temp = root;
        while (temp != NULL)
        {
            if (node->data < temp->data)
            {
                if (temp->left == NULL)
                {
                    temp->left = node;
                    return;
                }
                temp = temp->left;
            }
            else
            {
                if (temp->right == NULL)
                {
                    temp->right = node;
                    return;
                }
                temp = temp->right;
            }
        }
    }
    void inorder(Node *node)
    {
        if (node == NULL)
            return;
        inorder(node->left);
        cout << node->data << " ";
        inorder(node->right);
    }
    void preorder(Node *node)
    {
        if (node == NULL)
            return;
        cout << node->data << " ";
        preorder(node->left);
        preorder(node->right);
    }
    void postorder(Node *node)
    {
        if (node == NULL)
```

```
                return;
            postorder(node->left);
            postorder(node->right);
            cout << node->data << " ";
    }
};
```

```
int main()
{
    BST bst;
    int n;
    cout << "Enter number of nodes: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        Node *node = new Node();
        bst.insert(node);
    }
    cout << "Inorder: ";
    bst.inorder(bst.root);
    cout << endl;
    cout << "Preorder: ";
    bst.preorder(bst.root);
    cout << endl;
    cout << "Postorder: ";
    bst.postorder(bst.root);
    cout << endl;
    return 0;
}
```

Output: (screenshot)

```
Enter number of nodes: 5
Enter data: 7
Enter data: 34
Enter data: 5
Enter data: 2
Enter data: 9
Inorder: 2 5 7 9 34
Preorder: 7 5 2 34 9
Postorder: 2 5 9 34 7
```

Test Case: Any two (screenshot)

Conclusion: Therefore, we can implement Binary Search Tree ADT using Linked List.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search

Theory: A Graph is a non-linear data structure which can have parent-child as well as other complex relationships between the nodes. It is a set of edges and vertices, where vertices are the nodes, and the edges are the links connecting the nodes. We can implement a graph using adjacency matrix or adjacency list.

Code:

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

using namespace std;

const int MAXN = 100;

// Depth First Search (DFS)
void dfs(int graph[][MAXN], bool visited[], int n, int node)
{
    visited[node] = true;
    cout << node << " ";

    for (int i = 0; i < n; ++i)
    {
        if (graph[node][i] == 1 && !visited[i])
```

```
            {
                dfs(graph, visited, n, i);
            }
        }
}

// Breadth First Search (BFS)
void bfs(int graph[][MAXN], bool visited[], int n, int start)
{
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        cout << node << " ";

        for (int i = 0; i < n; ++i)
        {
            if (graph[node][i] == 1 && !visited[i])
            {
                q.push(i);
                visited[i] = true;
            }
        }
    }
}

int main()
{
    int n;
    cout << "Enter the number of vertices: ";
    cin >> n;

    int graph[MAXN][MAXN]; // Adjacency matrix
    cout << "Enter the adjacency matrix:" << endl;
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            cin >> graph[i][j];
        }
    }

    bool visited[MAXN] = {false}; // Visited array to keep track of
visited nodes
```

```
    cout << "Depth First Search (DFS): ";
    for (int i = 0; i < n; ++i)
    {
        if (!visited[i])
        {
            dfs(graph, visited, n, i);
        }
    }
    cout << endl;

    // Resetting visited array for BFS
    fill(visited, visited + n, false);

    cout << "Breadth First Search (BFS): ";
    for (int i = 0; i < n; ++i)
    {
        if (!visited[i])
        {
            bfs(graph, visited, n, i);
        }
    }
    cout << endl;

    return 0;
}
```

Output: (screenshot)

```
Enter the number of vertices: 3
Enter the adjacency matrix:
0 1 0
1 1 1
0 0 0
Depth First Search (DFS): 0 1 2
Breadth First Search (BFS): 0 1 2
```

Test Case: Any two (screenshot)

Conclusion: Therefore, we can implement Graph Traversal techniques by Depth First and Breadth First using adjacency matrix.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

Title: Implement Binary Search algorithm to search an element in the array.

Theory:

Binary Search is a searching algorithm which is used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Code:

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

int binarySearch(int arr[], int n, int a)
{
    int l = 0, r = n - 1;
    while (l <= r)
    {
        int m = l + (r - l) / 2;
```

```
        if (arr[m] == a)
        {
            return m;
        }
        if (arr[m] < a)
        {
            l = m + 1;
        }
        else
        {
            r = m - 1;
        }
    }
    return -1;
}
```

```
int main()
{
    int n, a;
    cout << "Enter size of array: ";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter " << i + 1 << " element: ";
        cin >> arr[i];
    }
    cout << "Enter element to search for: ";
    cin >> a;
    int b = binarySearch(arr, n, a);
    if (b == -1)
    {
        cout << "Element not found." << endl;
    }
    else
    {
        cout << "Element found at index " << b << endl;
    }
    return 0;
}
```

Output: (screenshot)

```
Enter size of array: 5
Enter 1 element: 1
Enter 2 element: 2
Enter 3 element: 5
Enter 4 element: 8
Enter 5 element: 9
Enter element to search for: 5
Element found at index 2
```

Test Case: Any two (screenshot)

Conclusion: Therefore, we can implement Binary Search algorithm in a sorted array to search the index location of an element present in the array in an efficient manner.

Name of Student:Sahil Sable

Roll Number: 150096723010

Experiment No: 1

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

Theory:

In Bubble Sort algorithm, we traverse from left and compare adjacent elements and the higher one is placed at right side. In this way, the largest element is moved to the rightmost end at first. This process is then continued to find the second largest and place it and so on until the data is sorted.

Code:

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

int main()
{
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter " << i + 1 << " element: ";
        cin >> arr[i];
    }
    cout << "Array: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }

    // ascending order
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    cout << "\nArray in ascending order: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }

    // descending order
    for (int i = 0; i < n - 1; i++)
```

```
    {
        for (int j = 0; j < n - 1 - i; j++)
        {
            if (arr[j] < arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    cout << "\nArray in descending order: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout<<endl;
    return 0;
}
```

Output: (screenshot)

```
Enter number of elements: 6
Enter 1 element: 9
Enter 2 element: 3
Enter 3 element: 4
Enter 4 element: 2
Enter 5 element: 10
Enter 6 element: 8
Array: 9 3 4 2 10 8
Array in ascending order: 2 3 4 8 9 10
Array in descending order: 10 9 8 4 3 2
```

Test Case: Any two (screenshot)

Conclusion: Therefore, we can implement Bubble Sort algorithm to sort the array in ascending or descending order by traversing through the array and comparing the elements to the adjacent elements.