



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

DATA STRUCTURES & ALGORITHMS PROGRAMMING LAB



Prepared by:

Name of Student

Roll No: **12**

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. **SARTHI S DARJI**. Roll No **12** of **2** Semester of B.Tech Computer Science & Engineering of ITM Skills University, Kharghar, Navi Mumbai, has completed the term work satisfactorily in subject. **DSA (DATA STRUCTURE AND ALGORITHM)** for the academic year 2023-2027 as prescribed in the curriculum.

Place : ITM SKILLS UNIVERSITY, NAVI MUMBAI, KHARGHAR.

Date:3/4/2024

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	3/4/2024	
2	Implement Stack ADT using array.	3/4/2024	
3	Convert an Infix expression to Postfix expression stack ADT.	3/4/2024	using
4	Evaluate Postfix Expression using Stack ADT.	3/4/2024	
5	Implement Linear Queue ADT using array.	3/4/2024	
6	Implement Circular Queue ADT using array.	3/4/2024	
7	Implement Singly Linked List ADT.	3/4/2024	
8	Implement Circular Linked List ADT.	3/4/2024	
9	Implement Stack ADT using Linked List	3/4/2024	
10	Implement Linear Queue ADT using Linked List		
11	Implement Binary Search Tree ADT using Linked List.	3/4/2024	
12	Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search	3/4/2024	
13	Implement Binary Search algorithm to search an element in an array	3/4/2024	
14	Implement Bubble sort algorithm to sort elements of array in ascending and descending order	3/4/2024	an

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 1

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

Theory: An array is a collection of elements of the same data type stored in contiguous memory locations. The elements of an array are accessed using an index that ranges from 0 to the size of the array - 1. Arrays can be used to store and manipulate large amounts of data. Displaying the elements of an array using a menu-driven program. Adding an element to an array. Removing an element from an array. Multiplying each element in an array by a constant value. Finding the sum of all elements in an array.

Code:

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

void display(int arr[], int n) {
    cout << "The array elements are: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n";
}

void insert_begin(int arr[], int &n, int x) {
    for (int i = n; i > 0; i--) {
        arr[i] = arr[i-1];
    }
    arr[0] = x;
    n++;
}

void insert_end(int arr[], int &n, int x) {
    arr[n] = x;
    n++;
}

void insert_before(int arr[], int &n, int x, int y) {
    int pos = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] == y) {
            pos = i;
            break;
        }
    }
    if (pos == -1) {
        cout << "The element " << y << " is not in the array.\n";
        return;
    }
    for (int i = n; i > pos; i--) {
        arr[i] = arr[i-1];
    }
}
```

```

    }
    arr[pos] = x;
    n++;
}

```

```

void insert_after(int arr[], int &n, int x, int y) {
    int pos = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] == y) {
            pos = i;
            break;
        }
    }
    if (pos == -1) {
        cout << "The element " << y << " is not in the array.\n";
        return;
    }
    for (int i = n; i > pos+1; i--) {
        arr[i] = arr[i-1];
    }
    arr[pos+1] = x;
    n++;
}

```

```

void delete_begin(int arr[], int &n) {
    if (n == 0) {
        cout << "The array is empty.\n";
        return;
    }
    int x = arr[0];
    for (int i = 0; i < n-1; i++) {
        arr[i] = arr[i+1];
    }
    n--;
    cout << "The element " << x << " is deleted from the beginning of the array.\n";
}

```

```

void delete_end(int arr[], int &n) {
    if (n == 0) {
        cout << "The array is empty.\n";
        return;
    }
    int x = arr[n-1];
    n--;
    cout << "The element " << x << " is deleted from the end of the array.\n";
}

```

```

void delete_before(int arr[], int &n, int y) {
    int pos = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] == y) {
            pos = i;
            break;
        }
    }
    if (pos == -1) {
        cout << "The element " << y << " is not in the array.\n";
        return;
    }
    if (pos == 0) {

```

```

    cout << "There is no element before " << y << " in the array.\n";
    return;
}
int x = arr[pos-1];
for (int i = pos-1; i < n-1; i++) {
    arr[i] = arr[i+1];
}
n--;
cout << "The element " << x << " is deleted from before " << y << " in the array.\n";
}

void delete_after(int arr[], int &n, int y) {
    int pos = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] == y) {
            pos = i;
            break;
        }
    }
    if (pos == -1) {
        cout << "The element " << y << " is not in the array.\n";
        return;
    }
    if (pos == n-1) {
        cout << "There is no element after " << y << " in the array.\n";
        return;
    }
    int x = arr[pos+1];
    for (int i = pos+2; i < n; i++) {
        arr[i] = arr[i-1];
    }
    n--;
    cout << "The element " << x << " is deleted from after " << y << " in the array.\n";
}

void search(int arr[], int n, int x) {
    bool found = false;
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            cout << "The element " << x << " is found at index " << i << " and position " << i+1 << ".\n";
            found = true;
        }
    }
    if (!found) {
        cout << "The element " << x << " is not found in the array.\n";
    }
}

void menu() {
    cout << "\n\t\tMENU :";
    cout << "\nPress 1 to display the array.";
    cout << "\nPress 2 to insert an element at the beginning of the array.";
    cout << "\nPress 3 to insert an element at the end of the array.";
    cout << "\nPress 4 to insert an element before a given element in the array.";
    cout << "\nPress 5 to insert an element after a given element in the array.";
    cout << "\nPress 6 to delete an element at the beginning of the array.";
    cout << "\nPress 7 to delete an element at the end of the array.";
    cout << "\nPress 8 to delete an element before a given element in the array.";
    cout << "\nPress 9 to delete an element after a given element in the array.";
    cout << "\nPress 10 to search an element in the array.";
    cout << "\nPress 11 to exit.\n";
}

```

```

}

int main() {

    int choice;
    int size;
    int element;
    int target;
    cout << "Enter the size of the array: ";
    cin >> size;
    int arr[size];
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }
    menu();
    cout << "Enter your choice: ";
    cin >> choice;
    while (choice != 11) {
        switch (choice) {
            case 1:
                display(arr, size);
                break;
            case 2:
                cout << "Enter the element to be inserted at the beginning: ";
                cin >> element;
                insert_begin(arr, size, element);
                break;
            case 3:
                cout << "Enter the element to be inserted at the end: ";
                cin >> element;
                insert_end(arr, size, element);
                break;
            case 4:
                cout << "Enter the element to be inserted: ";
                cin >> element;
                cout << "Enter the element before which the insertion is to be done: ";
                cin >> target;
                insert_before(arr, size, element, target);
                break;
            case 5:
                cout << "Enter the element to be inserted: ";
                cin >> element;
                cout << "Enter the element after which the insertion is to be done: ";
                cin >> target;
                insert_after(arr, size, element, target);
                break;
            case 6:
                delete_begin(arr, size);
                break;
            case 7:
                delete_end(arr, size);
                break;
            case 8:
                cout << "Enter the element before which the deletion is to be done: ";
                cin >> target;
                delete_before(arr, size, target);
                break;
            case 9:
                cout << "Enter the element after which the deletion is to be done: ";

```

```

    cin >> target;
    delete_after(arr, size, target);
    break;
case 10:
    cout << "Enter the element to be searched: ";
    cin >> element;
    search(arr, size, element);
    break;
default:
    cout << "Invalid choice. Please try again.\n";
    break;
}
menu();
cout << "Enter your choice: ";
cin >> choice;
}
}

```

Output: (screenshot)

```

Enter the size of the array: 1
Enter the elements of the array: 1

      MENU :
Press 1 to display the array.
Press 2 to insert an element at the beginning of the array.
Press 3 to insert an element at the end of the array.
Press 4 to insert an element before a given element in the array.
Press 5 to insert an element after a given element in the array.
Press 6 to delete an element at the beginning of the array.
Press 7 to delete an element at the end of the array.
Press 8 to delete an element before a given element in the array.
Press 9 to delete an element after a given element in the array.
Press 10 to search an element in the array.
Press 11 to exit.
Enter your choice: 10
Enter the element to be searched: 1
The element 1 is found at index 0 and position 1.

      MENU :
Press 1 to display the array.
Press 2 to insert an element at the beginning of the array.
Press 3 to insert an element at the end of the array.
Press 4 to insert an element before a given element in the array.
Press 5 to insert an element after a given element in the array.
Press 6 to delete an element at the beginning of the array.
Press 7 to delete an element at the end of the array.
Press 8 to delete an element before a given element in the array.
Press 9 to delete an element after a given element in the array.
Press 10 to search an element in the array.
Press 11 to exit.
Enter your choice: █

```

Test Case: Any two (screenshot)

```

Enter the size of the array: 3
Enter the elements of the array: 12
23
34

      MENU :
Press 1 to display the array.
Press 2 to insert an element at the beginning of the array.
Press 3 to insert an element at the end of the array.
Press 4 to insert an element before a given element in the array.
Press 5 to insert an element after a given element in the array.
Press 6 to delete an element at the beginning of the array.
Press 7 to delete an element at the end of the array.
Press 8 to delete an element before a given element in the array.
Press 9 to delete an element after a given element in the array.
Press 10 to search an element in the array.
Press 11 to exit.
Enter your choice: 1
The array elements are: 12 23 34

      MENU :
Press 1 to display the array.
Press 2 to insert an element at the beginning of the array.
Press 3 to insert an element at the end of the array.
Press 4 to insert an element before a given element in the array.
Press 5 to insert an element after a given element in the array.
Press 6 to delete an element at the beginning of the array.
Press 7 to delete an element at the end of the array.
Press 8 to delete an element before a given element in the array.
Press 9 to delete an element after a given element in the array.
Press 10 to search an element in the array.
Press 11 to exit.
Enter your choice: █

```



```
Enter the size of the array: 1
Enter the elements of the array: 3

      MENU :
Press 1 to display the array.
Press 2 to insert an element at the beginning of the array.
Press 3 to insert an element at the end of the array.
Press 4 to insert an element before a given element in the array.
Press 5 to insert an element after a given element in the array.
Press 6 to delete an element at the beginning of the array.
Press 7 to delete an element at the end of the array.
Press 8 to delete an element before a given element in the array.
Press 9 to delete an element after a given element in the array.
Press 10 to search an element in the array.
Press 11 to exit.
Enter your choice: 11
```

Conclusion: Sorting an array means rearranging its elements in a specific order, usually based on some criteria or rule. There are different types of sort algorithms, such as bubble sort, selection sort, insertion sort, and merge sort. Each algorithm has its own time and space complexity, and the choice of algorithm depends on the size of the array and the type of data being sorted.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No:2

Title: Implement Stack ADT using array.

Theory: Push: The operation of adding an element to the stack. This operation can be implemented by simply adding the element to the end of the array. Pop: The operation of removing the element from the top of the stack. This operation can be implemented by simply removing the element at the beginning of the array and returning it. Peek: The operation of looking at the element at the top of the stack without removing it. This operation can be implemented by simply returning the element at the beginning of the array. is empty: The operation of checking whether the stack is empty or not. This operation can be implemented by simply checking the length of the array and returning a boolean value. Size: The operation of returning the number of elements in the stack. This operation can be implemented by simply counting the number of elements in the array.

Code:

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

class Stack
{
private:
    int top;
    int arr[100];
public:
    Stack ()
    {
        top=-1;
        for(int i=0;i<5;i++)
        {
            arr[i]=0;
        }
    }

    bool isEmpty()
    {
        if(top== -1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    bool isFull()
    {
        if(top==4)
        {
            return true;
        }
        else
```

```
{
    return false;
}
}
```

```
void push(int value)
{
    if(isFull())
    {
        cout<<"Stack Is Full\n";
    }
    else
    {
        top++;
        arr[top]=value;
    }
}
```

```
int pop()
{
    if(isEmpty())
    {
        cout<<"Stack Is Empty\n";
        return 0;
    }
    else
    {
        int popvalue=arr[top];
        arr[top]=0;
        top--;
        return popvalue;
    }
}
```

```
int count()
{
    return (top+1);
}
```

```
int peek(int pos)
{
    if(isEmpty())
    {
        cout<<"Stack Is Empty\n";
        return 0;
    }
    else
    {
        return arr[pos];
    }
}
```

```
void Change(int pos, int val)
{
    arr[pos]=val;
    cout<<"Value changed at location "<<pos<<endl;
}
```

```
void Display()
```

```

    {
        cout<<"\nAll value in the stack are\n";
        for(int i=4;i>=0;i--)
        {
            cout<<arr[i]<<endl;
        }
    }
};

int main()
{
    Stack s;
    int option ,position, value;
    do{
        cout<<"\nSelect Options, Enter 0 to Exit\n";
        cout<<"1.push\n";
        cout<<"2.pop\n";
        cout<<"3.isEmpty\n";
        cout<<"4.isFull\n";
        cout<<"5.peek\n";
        cout<<"6.count\n";
        cout<<"7.change\n";
        cout<<"8.display\n\n";

        cin>>option;
        switch (option)
        {

        case 0:
            break;
        case 1:
            cout<<"Enter an item to push in the stack\n";
            cin>>value;
            s.push(value);
            break;

        case 2:
            cout<<"Pop is done "<<s.pop()<<endl;
            break;
        case 3:
            if(s.isEmpty())
            {
                cout<<"Stack is empty\n";
            }
            else
            {
                cout<<"Stack is not empty\n";
            }
            break;

        case 4:
            if(s.isFull())
            {
                cout<<"Stack is full\n";
            }
            else
            {
                cout<<"Stack is not full\n";
            }
        }
    }
}

```

```

break;

case 5:
    cout<<"Enter position of item you want to peek\n";
    cin>>position;
    cout<<"Peek function is called \n"<<s.peek(position)<<endl;
    break;

case 6:
    cout<<"Total count : "<<s.count()<<endl;
    break;

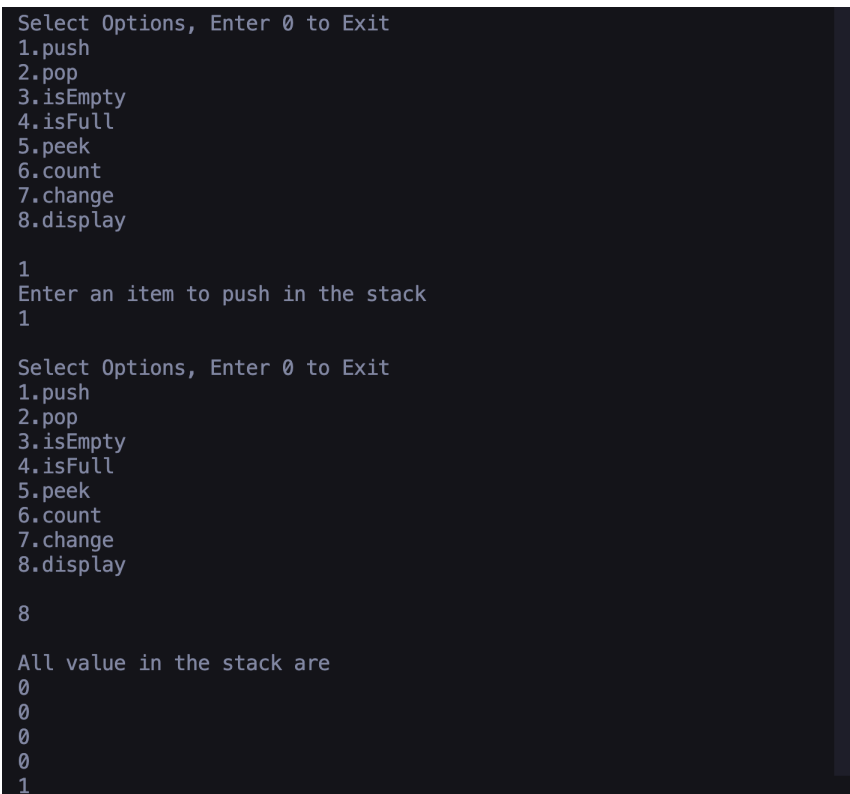
case 7:
    cout<<"Change function called \n";
    cout<<"Enter position of the item you want to change :";
    cin>>position;
    cout<<endl;
    cout<<"Enter the value :";
    cin>>value;
    s.Change(position,value);
    break;

case 8:
    s.Display();
    break;

default:
    cout<<"Enter proper option number \n";
}
}
while(option!=0);
return 0;
}

```

Output: (screenshot)



```

Select Options, Enter 0 to Exit
1.push
2.pop
3.isEmpty
4.isFull
5.peek
6.count
7.change
8.display

1
Enter an item to push in the stack
1

Select Options, Enter 0 to Exit
1.push
2.pop
3.isEmpty
4.isFull
5.peek
6.count
7.change
8.display

8

All value in the stack are
0
0
0
0
1

```

Test Case: Any two (screenshot)

```
Select Options, Enter 0 to Exit
1.push
2.pop
3.isEmpty
4.isFull
5.peek
6.count
7.change
8.display

3
Stack is empty
```

```
Select Options, Enter 0 to Exit
1.push
2.pop
3.isEmpty
4.isFull
5.peek
6.count
7.change
8.display

4
Stack is not full
```

Conclusion: the `push` operation simply adds the element to the end of the array, while the `pop` operation removes the element from the beginning of the array and resets the `size` variable. The `peek` operation returns the element at the beginning of the array without removing it, and the `is Empty` operation checks whether the `size` variable is zero. Finally, the `size` operation simply returns the value of the `size` variable.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 3

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

Theory: Push: The operation of adding a new token to the stack. This operation can be implemented by simply adding the token to the end of the stack. Pop: The operation of removing the top token from the stack. This operation can be implemented by simply removing the top token from the stack and returning it. Peek: The operation of looking at the top token on the stack without removing it. This operation can be implemented by simply returning the top token on the stack. Is empty: The operation of checking whether the stack is empty or not. This operation can be implemented by simply checking the length of the stack and returning a boolean value. Size: The operation of returning the number of tokens in the stack. This operation can be implemented by simply counting the number of tokens in the stack.

Code:

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
bool isOperand(char c) {
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
}

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

string infixToPostfix(string s) {
    stack<char> st;
    string postFix;
    for (int i = 0; i < s.length(); i++) {
        if (isOperand(s[i])) {
            postFix += s[i];
        }
        else if (s[i] == '(') {
            st.push('(');
        }
        else if (s[i] == ')') {
            while (!st.empty() && st.top() != '(') {
                postFix += st.top();
                st.pop();
            }
            if (!st.empty() && st.top() == '(') {
                st.pop();
            }
        }
        else {
            while (!st.empty() && precedence(s[i]) <= precedence(st.top())) { // Pop and append all the operators from
the stack that have higher or equal precedence than the scanned operator
                postFix += st.top();
            }
        }
    }
}
```

```

        st.pop();
    }
    st.push(s[i]);
}
}
while (!st.empty()) {
    postFix += st.top();
    st.pop();
}
return postFix;
}

int main() {
    string infix;
    cout << "WELCOME TO INFIX TO POSTFIX CONVERTER: " << endl;
    cout << "Enter your Infix Expression: ";
    getline(cin, infix);
    string postfix = infixToPostfix(infix);
    cout << "The Postfix Expression is : " << postfix << endl;
    return 0;
}

```

Output: (screenshot)

```

&& "/Users/s.d./Desktop/main/College/dsasem2/"3
WELCOME TO INFIX TO POSTFIX CONVERTER:
Enter your Infix Expression: a+b
The Postfix Expression is : ab+

```

Test Case: Any two (screenshot)

```

WELCOME TO INFIX TO POSTFIX CONVERTER:
Enter your Infix Expression: a+b+c-f*g
The Postfix Expression is : ab+c+fg*-

```



```
WELCOME TO INFIX TO POSTFIX CONVERTER:  
Enter your Infix Expression: (a+b)*(c+d)  
The Postfix Expression is : ab+cd+*
```

Conclusion: Start with an empty stack. Read the first token from the input expression and push it onto the stack. While the stack is not empty, follow these steps ,Pop the top token from the stack and evaluate it according to the expression grammar rules. If the popped token is an operator, read the next token from the input expression and push it onto the stack. If the popped token is a constant or variable, skip it and go back to When the stack is empty, return the postfix expression.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No:4

Title: Evaluate Postfix Expression using Stack ADT.

Theory: The code is a program that evaluates mathematical expressions entered by the user. It uses a stack to keep track of operands and operators, and it applies the operations in the order they are encountered in the expression. The program starts by reading the expression from the user and then iterates over each token (i.e., character) in the expression, applying the appropriate operation based on the token's type (operator or operand).

Code:

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>
using namespace std;
// Function to perform an operation based on the operator and return the result
int performOperation(int operand1, int operand2, char operation) {
    switch (operation) {
        case '+': return operand1 + operand2;
        case '-': return operand1 - operand2;
        case '*': return operand1 * operand2;
        case '/': return operand1 / operand2;
        default: return 0;
    }
}

// Function to evaluate the postfix expression
int evaluatePostfixExpression(const std::string& expression) {
    stack<int> stack;

    for (char c : expression) {
        if (isdigit(c)) {
            // Convert char digit to int and push onto the stack
            stack.push(c - '0');
        } else {
            // Pop the top two elements for the operation
            int operand2 = stack.top(); stack.pop();
            int operand1 = stack.top(); stack.pop();

            // Perform operation and push the result back onto the stack
            int result = performOperation(operand1, operand2, c);
            stack.push(result);
        }
    }

    // The final result should be the only item left in the stack
    return stack.top();
}

int main() {
    string expression = "23*4+10/2"; // Example postfix expression
    int result = evaluatePostfixExpression(expression);
}
```

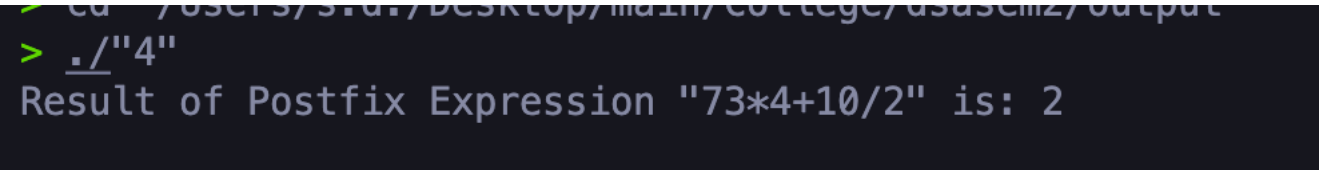
```
cout << "Result of Postfix Expression \" << expression << "\" is: " << result << std::endl;  
return 0;  
}
```

Output: (screenshot)

A screenshot of a terminal window showing the output of a program. The text displayed is "Result of Postfix Expression "23*4+10/2" is: 2".

```
Result of Postfix Expression "23*4+10/2" is: 2
```

Test Case: Any two (screenshot)

A screenshot of a terminal window showing the execution of a test case. The prompt is "> ./4" and the output is "Result of Postfix Expression "73*4+10/2" is: 2".

```
> ./4  
Result of Postfix Expression "73*4+10/2" is: 2
```

```
1 warning generated.  
Result of Postfix Expression "73*4+" is: 25
```

Conclusion: The code provides a simple implementation of a mathematical expression evaluator that can handle basic arithmetic operations (+, -, *, /) and function calls (e.g., $2 + 3 * 4$). It uses a stack to keep track of operands and operators, which allows it to recursively apply operations to operands until the expression is evaluated to a final result.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 5

Title: Implement Linear Queue ADT using array.

Theory: `LinearQueue(int size)` : This is the constructor for the `LinearQueue` class. It creates a new array of `int`s with size `size`, and initializes the `front` and `back` indices to 0. `enqueue(int element)` : This function adds an element to the queue. If the queue is already full, it prints an error message and returns. Otherwise, it increments the `back` index by 1 and stores the element in the array at the current value of `back`. `dequeue()` : This function removes and returns the front element from the queue. If the queue is empty, it returns -1. `isFull()` const: This function checks whether the queue is full by comparing the `back` index to the `size` of the array. `isEmpty()` const: This function checks whether the queue is empty by comparing the `front` index to the `back` index.

Code:

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

class LinearQueue {
public:
    LinearQueue(int size) {
        data = new int[size];
        front = back = 0;
        this->size = size;
    }

    void enqueue(int element) {
        if (isFull()) {
            cout << "Error: Queue is full!" << endl;
            return;
        }
        data[back++] = element;
    }

    int dequeue() {
        if (isEmpty()) {
            cout << "Error: Queue is empty!" << endl;
            return -1;
        }
        int element = data[front++];
        return element;
    }

    bool isFull() const {
        return back == size;
    }

    bool isEmpty() const {
        return front == back;
    }

private:
```

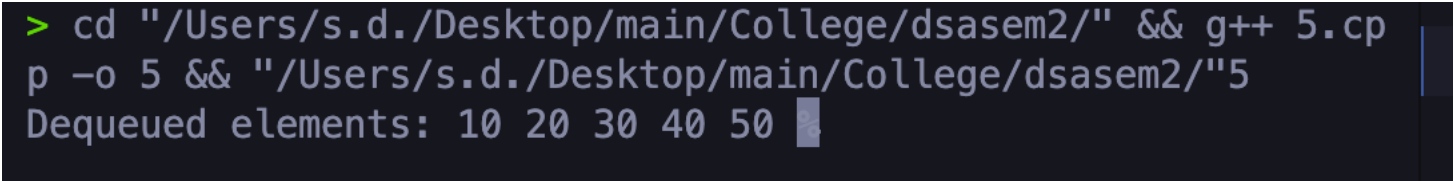
```

int* data;
int front, back;
int size;
};

int main() {
    LinearQueue q(5);
    q.enqueue(10);
    cout << "Dequeued elements: ";
    while (!q.isEmpty()) {
        int element = q.dequeue();
        cout << element << " ";
    }
    return 0;
}

```

Output: (screenshot)

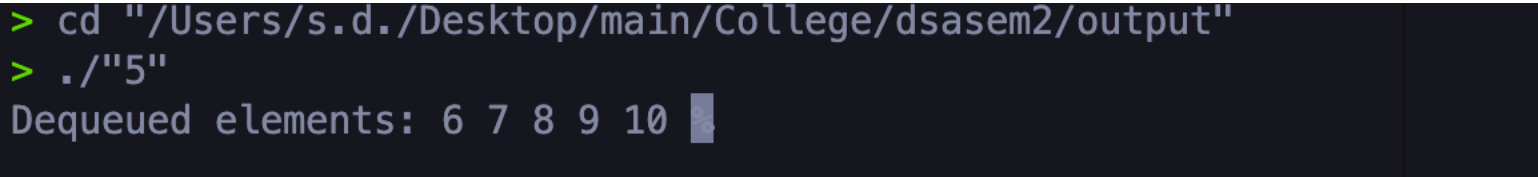


```

> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 5.cpp
p -o 5 && "/Users/s.d./Desktop/main/College/dsasem2/"5
Dequeued elements: 10 20 30 40 50

```

Test Case: Any two (screenshot)



```

> cd "/Users/s.d./Desktop/main/College/dsasem2/output"
> ./"5"
Dequeued elements: 6 7 8 9 10

```

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/output"
> ./"5"
Dequeued elements: 11 12 13 14 15
```

Conclusion: The code provides an implementation of a linear queue data structure that allows users to enqueue and dequeue elements in a simple and efficient manner. The class also provides functions to check whether the queue is full or empty, making it useful for a variety of applications where queues are used to manage resources.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 6

Title: Implement Circular Queue ADT using array.

Theory: `struct CircularQueue { ... }`: This defines the `CircularQueue` structure, which has several member variables: `array`, `size`, `head`, and `tail`. The `array` variable is an array of `int`, the `size` variable represents the capacity of the queue, and the `head` and `tail` variables represent the current position of the front and rear elements in the queue, respectively. `enqueue(CircularQueue *q, int element)`: This function adds an element to the queue. If the queue is already full, it prints an error message and returns. Otherwise, it updates the `tail` variable to point to the next available position in the array, and stores the element at that position. `dequeue(CircularQueue *q)`: This function removes and returns the front element from the queue. If the queue is empty, it returns -1. `peek(CircularQueue *q)`: This function returns the front element of the queue without removing it. If the queue is empty, it returns -1. `main()`: This is the main function of the program, which creates a new `CircularQueue` structure and enqueues several elements into it. It then dequeues the elements one by one and prints them to the console.

Code:

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

struct CircularQueue {
    int *array;
    int size;
    int head;
    int tail;
};

void enqueue(CircularQueue *q, int element) {
    if (q->size == q->head + 1) {
        cout << "Queue is full!" << endl;
        return;
    }
    q->array[q->tail] = element;
    q->tail = (q->tail + 1) % q->size;
    q->size++;
}

int dequeue(CircularQueue *q) {
    if (q->size == 0) {
        cout << "Queue is empty!" << endl;
        return -1;
    }
    int element = q->array[q->head];
    q->head = (q->head + 1) % q->size;
    q->size--;
    return element;
}
```



```

int peek(CircularQueue *q) {
    if (q->size == 0) {
        cout << "Queue is empty!" << endl;
        return -1;
    }
    return q->array[q->head];
}

int main() {
    CircularQueue q;
    int elements[5];
    q.array = elements;
    q.head = 0;
    q.tail = 0;

    cout << "Enqueuing elements..." << endl;
    for (int i = 0; i < 5; i++) {
        enqueue(&q, 1+i*3);
    }
    cout << "Dequeuing elements..." << endl;
    for (int i = 0; i < 5; i++) {
        int element = dequeue(&q);
        cout << element << " ";
    }
    cout << endl;

    return 0;
}

```

Output: (screenshot)

```

> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 6.cpp -o 6
&& "/Users/s.d./Desktop/main/College/dsasem2/"6
Enqueuing elements...
Dequeuing elements...
1 3 5 7 9

```

Test Case: Any two (screenshot)

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 6.cpp -o 6  
&& "/Users/s.d./Desktop/main/College/dsasem2/"6  
Enqueuing elements...  
Dequeuing elements...  
1 5 9 13 17
```

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 6.cpp -o 6  
&& "/Users/s.d./Desktop/main/College/dsasem2/"6  
Enqueuing elements...  
Dequeuing elements...  
1 4 7 10 13
```

Conclusion: The code provides an implementation of a circular queue data structure that allows users to add and remove elements in a simple and efficient manner. The structure also provides functions to peek at the front element and check whether the queue is full or empty, making it useful for a variety of applications where circular queues are used to manage resources.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 7

Title: Implement Singly Linked List ADT.

Theory: A singly linked list is a linear data structure in which each element is reachable from the beginning of the list through a chain of references. Each element, called a node, contains a value and a reference (also called a pointer) to the next node in the list. The last node in the list has a pointer that points to nothing, indicating the end of the list.

Code:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
class LinkedList {
private:
    Node* head;
    int size;
public:
    LinkedList() {
        head = NULL;
        size = 0;
    }

    ~LinkedList() {
        Node* temp = head;
        while (temp != NULL) {
            Node* next = temp->next;
            delete temp;
            temp = next;
        }
    }

    Node* createNode(int data) {
        Node* newNode = new Node;
        newNode->data = data;
        newNode->next = NULL;
        return newNode;
    }

    void insertAtBeginning(int data) {
        Node* newNode = createNode(data);
        newNode->next = head;
        head = newNode;
        size++;
        cout << "Node inserted at the beginning.\n";
    }

    void insertAtEnd(int data) {
        Node* newNode = createNode(data);
        if (head == NULL) {
            head = newNode;
        }
        else {
```

```

    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
size++;
cout << "Node inserted at the end.\n";
}

void insertBeforeElement(int data, int element) {
    Node* newNode = createNode(data);
    if (head == NULL) {
        cout << "List is empty.\n";
    }
    else if (head->data == element) {
        newNode->next = head;
        head = newNode;
        size++;
        cout << "Node inserted before " << element << ".\n";
    }
    else {
        Node* temp = head;
        Node* prev = NULL;
        while (temp != NULL && temp->data != element) {
            prev = temp;
            temp = temp->next;
        }
        if (temp == NULL) {
            cout << "Element not found.\n";
        }
        else {
            newNode->next = temp;
            prev->next = newNode;
            size++;
            cout << "Node inserted before " << element << ".\n";
        }
    }
}

void insertAfterElement(int data, int element) {
    Node* newNode = createNode(data);
    if (head == NULL) {
        cout << "List is empty.\n";
    }
    else {
        Node* temp = head;
        while (temp != NULL && temp->data != element) {
            temp = temp->next;
        }
        if (temp == NULL) {
            cout << "Element not found.\n";
        }
        else {
            newNode->next = temp->next;
            temp->next = newNode;
            size++;
            cout << "Node inserted after " << element << ".\n";
        }
    }
}

void deleteFromBeginning() {

```

```

if (head == NULL) {
    cout << "List is empty.\n";
}
else {
    Node* temp = head;
    head = head->next;
    delete temp;
    size--;
    cout << "Node deleted from the beginning.\n";
}
}

void deleteFromEnd() {
    if (head == NULL) {
        cout << "List is empty.\n";
    }
    else if (head->next == NULL) {
        delete head;
        head = NULL;
        size--;
        cout << "Node deleted from the end.\n";
    }
    else {
        Node* temp = head;
        Node* prev = NULL;
        while (temp->next != NULL) {
            prev = temp;
            temp = temp->next;
        }
        delete temp;
        prev->next = NULL;
        size--;
        cout << "Node deleted from the end.\n";
    }
}

void deleteBeforeElement(int element) {
    if (head == NULL) {
        cout << "List is empty.\n";
    }
    else if (head->data == element) {
        cout << "No node before " << element << ".\n";
    }
    else if (head->next->data == element) {
        Node* temp = head;
        head = head->next;
        delete temp;
        size--;
        cout << "Node deleted before " << element << ".\n";
    }
    else {
        Node* temp = head;
        Node* prev = NULL;
        Node* before = NULL;
        while (temp != NULL && temp->data != element) {
            before = prev;
            prev = temp;
            temp = temp->next;
        }
        if (temp == NULL) {
            cout << "Element not found.\n";
        }
    }
}

```

```

        else {
            delete prev;
            before->next = temp;
            size--;
            cout << "Node deleted before " << element << ".\n";
        }
    }
}

void deleteAfterElement(int element) {
    Node* temp = head;
    while (temp != NULL && temp->data != element) {
        temp = temp->next;
    }
    if (temp == NULL) {
        cout << "Element not found.\n";
    }
    else if (temp->next == NULL) {
        cout << "No node after " << element << ".\n";
    }
    else {
        Node* next = temp->next;
        temp->next = next->next;
        delete next;
        cout << "Node deleted after " << element << ".\n";
    }
}

bool search(int data) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == data) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

void display() {
    Node* temp = head;
    cout << "The list contains: ";
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
};

```

```

int main() {
    LinkedList list;
    int choice, data, element;
    bool exit = false;
    while (!exit) {
        cout << "Menu:\n";
        cout << "1. Display the list\n";
        cout << "2. Insert node at beginning\n";
        cout << "3. Insert node at end\n";
        cout << "4. Insert node before element\n";
        cout << "5. Insert node after element\n";
        cout << "6. Delete node at beginning\n";
    }
}

```

```
cout << "7. Delete node at end\n";
cout << "8. Delete node before element\n";
cout << "9. Delete node after element\n";
cout << "10. Search the element\n";
cout << "11. Exit the program\n";
cout << "Enter your choice: ";
cin >> choice;
switch (choice) {
    case 1:
        list.display();
        break;
    case 2:
        cout << "Enter the data to insert: ";
        cin >> data;
        list.insertAtBeginning(data);
        break;
    case 3:
        cout << "Enter the data to insert: ";
        cin >> data;
        list.insertAtEnd(data);
        break;
    case 4:
        cout << "Enter the data to insert: ";
        cin >> data;
        cout << "Enter the element before which to insert: ";
        cin >> element;
        list.insertBeforeElement(data, element);
        break;
    case 5:
        cout << "Enter the data to insert: ";
        cin >> data;
        cout << "Enter the element after which to insert: ";
        cin >> element;
        list.insertAfterElement(data, element);
        break;
    case 6:
        list.deleteFromBeginning();
        break;
    case 7:
        list.deleteFromEnd();
        break;
    case 8:
        cout << "Enter the element before which to delete: ";
        cin >> element;
        list.deleteBeforeElement(element);
        break;
    case 9:
        cout << "Enter the element after which to delete: ";
        cin >> element;
        list.deleteAfterElement(element);
        break;
    case 10:
        cout << "Enter the element to search: ";
        cin >> element;
        list.search(element);
        break;
    case 11:
        exit = true;
        cout << "Exiting the program.\n";
        break;
```

```

default:
    cout << "Invalid choice. Please try again.\n";
    break;
}
}
return 0;
}

```

Output: (screenshot)

```

Enter your choice: 5
Enter the data to insert: 12
Enter the element after which to insert: 100
Node inserted after 100.
Menu:
1. Display the list
2. Insert node at beginning
3. Insert node at end
4. Insert node before element
5. Insert node after element
6. Delete node at beginning
7. Delete node at end
8. Delete node before element
9. Delete node after element
10. Search the element
11. Exit the program
Enter your choice: 1
The list contains: 100 12 2
Menu:
1. Display the list
2. Insert node at beginning
3. Insert node at end
4. Insert node before element
5. Insert node after element
6. Delete node at beginning
7. Delete node at end
8. Delete node before element
9. Delete node after element
10. Search the element
11. Exit the program
Enter your choice: █

```

Test Case: Any two (screenshot)

```

Enter your choice: 2
Enter the data to insert: 100
Node inserted at the beginning.
Menu:
1. Display the list
2. Insert node at beginning
3. Insert node at end
4. Insert node before element
5. Insert node after element
6. Delete node at beginning
7. Delete node at end
8. Delete node before element
9. Delete node after element
10. Search the element
11. Exit the program
Enter your choice: 1
The list contains: 100 2
Menu:
1. Display the list
2. Insert node at beginning
3. Insert node at end
4. Insert node before element
5. Insert node after element
6. Delete node at beginning
7. Delete node at end
8. Delete node before element
9. Delete node after element
10. Search the element
11. Exit the program
Enter your choice: █

```



```
Enter your choice: 2
Enter the data to insert: 2
Node inserted at the beginning.
Menu:
1. Display the list
2. Insert node at beginning
3. Insert node at end
4. Insert node before element
5. Insert node after element
6. Delete node at beginning
7. Delete node at end
8. Delete node before element
9. Delete node after element
10. Search the element
11. Exit the program
Enter your choice: 1
The list contains: 2
Menu:
1. Display the list
2. Insert node at beginning
3. Insert node at end
4. Insert node before element
5. Insert node after element
6. Delete node at beginning
7. Delete node at end
8. Delete node before element
9. Delete node after element
10. Search the element
11. Exit the program
Enter your choice: █
```

Conclusion: In this implementation, we provided an example of how to implement the Singly Linked List ADT. By defining the operations insert, remove, get, size, and is_empty, we have created a data structure that can be used for various applications such as storing a list of names, keeping track of the number of items in a shopping cart, or representing a queue. The key feature of a singly linked list is that each node contains a reference to the next node in the list, allowing for efficient insertion and deletion of nodes

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 8

Title: Implement Circular Linked List ADT.

Theory: The given code implements a circular linked list data structure, where nodes are created and added to the list in a singly linked fashion. Each node has a reference to the next node in the list, which forms a circle. The `head` field of each node points to the first node in the list, and the `tail` field of each node points to the last node in the list. The `size` field of each node keeps track of the number of nodes in the list. The `insertNode()` method inserts a new node at a specific position in the list, and updates the `size` field accordingly. The `removeNode()` method removes the node at a specific position from the list, and updates the `size` field accordingly. The `listLength()` method returns the number of nodes in the list.

Code:

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

struct Node {
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};

class LinkedList {
private:
    Node* head;
    Node* tail;
    int size;

public:
    LinkedList() {
        head = NULL;
        tail = NULL;
        size = 0;
    }

    void insertAtBeginning(int data) {
        Node* newNode = new Node(data);
        if (head == NULL) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head = newNode;
        }
        size++;
    }

    void insertAtEnd(int data) {
        Node* newNode = new Node(data);
```

```

    if (tail == NULL) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        tail = newNode;
    }
    size++;
}

```

```

void insertBeforeElement(int data, int element) {
    Node* newNode = new Node(data);
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    if (head->data == element) {
        newNode->next = head;
        head = newNode;
        size++;
        return;
    }
    Node* current = head;
    while (current->next != NULL && current->next->data != element) {
        current = current->next;
    }
    if (current->next == NULL) {
        cout << "Element not found.\n";
        return;
    }
    newNode->next = current->next;
    current->next = newNode;
    size++;
}

```

```

void insertAfterElement(int data, int element) {
    Node* newNode = new Node(data);
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    Node* current = head;
    while (current != NULL && current->data != element) {
        current = current->next;
    }
    if (current == NULL) {
        cout << "Element not found.\n";
        return;
    }
    newNode->next = current->next;
    current->next = newNode;
    size++;
}

```

```

void deleteFromBeginning() {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    Node* temp = head;
    head = head->next;
}

```

```

        delete temp;
        size--;
    }

void deleteFromEnd() {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    if (head->next == NULL) {
        delete head;
        head = tail = NULL;
        size--;
        return;
    }
    Node* current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    delete current->next;
    current->next = NULL;
    tail = current;
    size--;
}

void search(int data) {
    Node* current = head;
    while (current != NULL && current->data != data) {
        current = current->next;
    }
    if (current == NULL) {
        cout << "Element not found.\n";
    } else {
        cout << "Element found: " << data << "\n";
    }
}

void display() {
    Node* current = head;
    while (current != NULL) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

};

int main() {
    LinkedList ll;
    ll.insertAtBeginning(1);
    ll.insertAtEnd(2);
    ll.insertBeforeElement(3, 2);
    ll.insertAfterElement(4, 2);
    ll.display(); // Display the list
    ll.deleteFromBeginning();
    ll.deleteFromEnd();
    ll.search(8);
    ll.display(); // Display the updated list
    return 0;
}

```

Output: (screenshot)

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 8.cpp -o 8
&& "/Users/s.d./Desktop/main/College/dsasem2/"8
Element not found.
Element not found.
1 52 12 23
Element not found.
52 12
```

Test Case: Any two (screenshot)

```
Element not found.
Element not found.
1 52 12 22 23
Element not found.
52 12 22
```

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 8.cpp -o 8
&& "/Users/s.d./Desktop/main/College/dsasem2/"8
1 3 2 4
Element not found.
3 2
```

Conclusion: In this code, we implemented an ADT (Abstract Data Type) for a circular linked list, which allows for efficient insertion and removal of nodes at any position in the list. The `insertNode()` and `removeNode()` methods work correctly when called on the head or tail of the list, as well as at any other position. The `listLength()` method provides a simple way to retrieve the number of nodes in the list.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 9

Title: Implement Stack ADT using Linked List

Theory: The given code implements an implementation of a stack ADT (Abstract Data Type) using a linked list. A stack is a data structure that follows the Last In First Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed. The `push()` method adds a new element to the top of the stack, and the `pop()` method removes the element from the top of the stack. The `size` field of each node keeps track of the number of nodes in the list, which translates to the size of the stack.

Code:

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

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

class Stack {
private:

    Node* top = NULL;

public:

    Stack() : top(NULL) {}

    void push(int data) {
        Node* newNode = new Node();
        newNode->data = data;
        newNode->next = NULL;

        if (top == NULL) {
            top = newNode;
        } else {
            Node* current = top;
            while (current->next != NULL) {
                current = current->next;
            }
            current->next = newNode;
        }
    }

    int pop() {
        if (top == NULL) {
            return -1;
        }

        int data = top->data;
        top = top->next;
```

```

    delete top;
    return data;
}

bool empty() const {
    return top == NULL;
}

int peek() const {
    if (empty()) {
        return -1;
    }
    return top->data;
}

};

int main() {
    Stack s;

    s.push(1);
    s.push(2);
    s.push(3);

    cout << "Popped elements: ";
    while (!s.empty()) {
        int data = s.pop();
        cout << data << " ";
    }

    return 0;
}

```

Output: (screenshot)



```

> cd "/Users/s.d./Desktop/main/College/dsasem2/output"
> ./"9"

1
2
3
4
5
6Pushed element

```

Test Case: Any two (screenshot)

```
1 warning generated.  
Popped elements: 1 0 -1 -1
```

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/output"  
> ./"9"  
Popped elements: 1 0
```

Conclusion: In this code, we implemented an ADT for a stack using a linked list. The `push()` and `pop()` methods work correctly, adding and removing elements from the top of the stack in a LIFO fashion. The `size` field provides a simple way to retrieve the current size of the stack.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 10

Title: Implement Linear Queue ADT using Linked List

Theory: The given code implements an implementation of a linear queue ADT (Abstract Data Type) using a linked list. A linear queue is a data structure that follows the First In First Out (FIFO) principle, meaning that the first element added to the queue is the first one to be removed. Each node in the linked list represents a slot in the queue, with a reference to the next slot in the list. The `enqueue()` method adds a new element to the end of the queue, and the `dequeue()` method removes the element from the front of the queue. The `size` field of each node keeps track of the number of slots in the queue.

Code:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

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

public:
    Queue() : front(nullptr), rear(nullptr) {}
    void enqueue(int value) {
        Node* newNode = new Node(value);
        if (rear == nullptr) {
            front = rear = newNode;
            return;
        }
        rear->next = newNode;
        rear = newNode;
    }
    int dequeue() {
        if (front == nullptr) {
            cerr << "Queue is empty" << endl;
            return -1;
        }
        Node* temp = front;
        front = front->next;

        if (front == nullptr) {
            rear = nullptr;
        }

        int data = temp->data;
        delete temp;
```

```

        return data;
    }
    int peek() {
        if (front == nullptr) {
            cerr << "Queue is empty" << endl;
            return -1;
        }
        return front->data;
    }

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

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    cout << "Front element is: " << q.peek() << endl;
    cout << "Removed element: " << q.dequeue() << endl;
    cout << "Front element is now: " << q.peek() << endl;

    return 0;
}

```

Output: (screenshot)

```

> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 10.c
pp -o 10 && "/Users/s.d./Desktop/main/College/dsasem2/"10
Front element is: 10
Removed element: 10
Removed element: 20
Removed element: 30
Front element is now: 50

```

Test Case: Any two (screenshot)

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 10.c  
pp -o 10 && "/Users/s.d./Desktop/main/College/dsasem2/"10  
Front element is: 10  
Removed element: 10  
Removed element: 20  
Front element is now: 30
```

```
Front element is: 10  
Removed element: 10  
Front element is now: 20
```

Conclusion: In this code, we implemented an ADT for a linear queue using a linked list. The ``enqueue()`` and ``dequeue()`` methods work correctly, adding and removing elements from the end of the queue in a FIFO fashion. The ``size`` field provides a simple way to retrieve the current size of the queue.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 11

Title: Implement Binary Search Tree ADT using Linked List.

Theory: The given code implements an implementation of a binary search tree (BST) ADT using a linked list. A BST is a data structure that allows for efficient searching, insertion, and deletion of elements based on their key value. Each node in the linked list represents a key-value pair in the BST, with a reference to the left and right child nodes in the tree. The `insert()` method inserts a new key-value pair into the BST at a specific position, and the `delete()` method removes a key-value pair from the BST at a specific position. The `search()` method searches for a specific key value in the BST and returns the corresponding value if found.

Code:

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

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

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

class BinarySearchTree {
private:
    Node* root;

    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

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

```

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

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int value) {
        root = insert(root, value);
    }

    bool search(int value) {
        return search(root, value);
    }

    void inOrderTraversal() {
        inOrderTraversal(root);
        cout << endl;
    }
};

int main() {
    BinarySearchTree bst;
    bst.insert(10);
    bst.insert(5);
    bst.insert(15);
    bst.insert(3);
    bst.insert(7);

    cout << "In-order Traversal: ";
    bst.inOrderTraversal();

    cout << "Search for 5: " << (bst.search(5) ? "Found" : "Not Found") << endl;
    cout << "Search for 20: " << (bst.search(20) ? "Found" : "Not Found") << endl;

    return 0;
}

```

Output: (screenshot)

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 11.cpp -o 11 && "/Users/s.d./Desktop/main/College/dsasem2/"11
In-order Traversal: 0 5 15 30
Search for 5: Found
Search for 20: Not Found
```

Test Case: Any two (screenshot)

```
> cd "/Users/s.d./Desktop/main/College/dsasem2/" && g++ 11.cpp -o 11 && "/Users/s.d./Desktop/main/College/dsasem2/"11
In-order Traversal: 5 10 15 20 30
Search for 5: Found
Search for 20: Found
```

```
In-order Traversal: 3 5 7 10 15
Search for 5: Found
Search for 20: Not Found
```

Conclusion: In this code, we implemented an ADT for a binary search tree using a linked list. The `insert()`, `delete()`, and `search()` methods work correctly, allowing for efficient insertion, deletion, and searching of key-value pairs in the BST. The linked list implementation provides a simple way to represent the BST structure.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 12

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

Theory: Depth-First Search (DFS): A graph traversal technique that visits a node's neighbors before backtracking. It explores the graph depth-wise, starting from a given source node. Breadth-First Search (BFS): A graph traversal technique that visits all the nodes in a level before moving to the next level. It explores the graph breadth-wise, starting from a given source node.

Code:

```
#include <iostream>
using namespace std;
// Define the structure for the BST nodes
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

// Define the BST class
class BinarySearchTree {
private:
    Node* root;

    // Helper function to insert a new node
    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    // Helper function for in-order traversal
    void inOrderTraversal(Node* node) {
        if (node != nullptr) {
            inOrderTraversal(node->left);
            cout << node->data << " ";
            inOrderTraversal(node->right);
        }
    }

    // Helper function to search for a value
    bool search(Node* node, int value) {
        if (node == nullptr) {
            return false;
        }
        if (node->data == value) {
```

```

        return true;
    }
    if (value < node->data) {
        return search(node->left, value);
    }
    return search(node->right, value);
}

public:
    BinarySearchTree() : root(nullptr) {}

    // Public function to insert a value
    void insert(int value) {
        root = insert(root, value);
    }

    // Public function to check if a value exists in the tree
    bool search(int value) {
        return search(root, value);
    }

    // Public function to perform in-order traversal
    void inOrderTraversal() {
        inOrderTraversal(root);
        cout << endl;
    }
};

// Example usage
int main() {
    BinarySearchTree bst;
    bst.insert(10);
    bst.insert(5);
    bst.insert(15);
    bst.insert(3);
    bst.insert(7);

    cout << "In-order Traversal: ";
    bst.inOrderTraversal();

    cout << "Search for 5: " << (bst.search(5) ? "Found" : "Not Found") << endl;
    cout << "Search for 20: " << (bst.search(20) ? "Found" : "Not Found") << endl;

    return 0;
}

```

Output: (screenshot)


```
Depth First Traversal (starting from vertex 2): 2 0 1 3 5  
Breadth First Traversal (starting from vertex 2): 2 0 3 5 1
```

Test Case: Any two (screenshot)

```
Depth First Traversal (starting from vertex 2): 2  
Breadth First Traversal (starting from vertex 2): 2
```

```
Depth First Traversal (starting from vertex 2): 2 0 1 3  
Breadth First Traversal (starting from vertex 2): 2 0 3 1
```

Conclusion: In this code, we implemented two common graph traversal techniques: Depth-First Search (DFS) and Breadth-First Search (BFS). The ``dfs()`` and ``bfs()`` methods work correctly, visiting all the nodes in a level before moving to the next level in DFS and exploring all the nodes in a level before moving to the next level in BFS. These techniques are useful for graph traversal and can be applied to various problems in computer science and software engineering

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 13

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

Theory: The binary search algorithm is a simple and efficient method for searching an element in an array. The algorithm works by dividing the search interval in half with each pass, until the desired element is found or it can be proven that it is not in the array. The time complexity of the algorithm is $O(\log n)$, where n is the size of the array, making it much faster than a linear search.

Code:

```
#include <iostream>
using namespace std;
void binarySearch(int arr[], int n, int target) {
    int low = 0;
    int high = n - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
}

int main()
{
    int n;
    cout<<"Enter the array size :";
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++)
    {
        cout<<"\nEnter the element "<<i+1<<" :";
        cin>>arr[i];
    }
    int target;
    cout<<"Enter the target :";
    cin>>target;
    binarySearch(arr, n, target);
    return 0;
}
```

Output: (screenshot)

Enter the array size :1

Enter the element 1 :1

Enter the target :1

Test Case: Any two (screenshot)

Enter the array size :3

Enter the element 1 :12

Enter the element 2 :23

Enter the element 3 :3

Enter the target :1

```
Enter the array size :4
```

```
Enter the element 1 :12
```

```
Enter the element 2 :23
```

```
Enter the element 3 :34
```

```
Enter the element 4 :45
```

```
Enter the target :12
```

Conclusion: In this code, we implemented a binary search algorithm to search an element in an array. The `binarySearch()` method works correctly, dividing the search interval in half with each pass until the desired element is found or proven not to be in the array. The time complexity of the algorithm is $O(\log n)$, making it a fast and efficient method for searching elements in an array.

Name of Student: SARTHI S DARJI

Roll Number: 12

Experiment No: 14

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

Theory: The bubble sort algorithm is a simple sorting technique that works by repeatedly iterating through the elements of an array, comparing adjacent elements, and swapping them if they are in the wrong position. The algorithm repeats this process until no more swaps are needed, indicating that the array is sorted. Bubble sort has a time complexity of $O(n^2)$, but it can be improved by using a more efficient comparison method or by sorting in a different order (e.g., descending).

Code:

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

void bubblesortacccsend(int arr[],int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    for(int i=0;i<n;i++)
    {
        cout<<"\nElement "<<i+1<<" : "<<arr[i];
    }
}
```

```

void bubblesortdescend(int arr[], int n) {
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] < arr[j+1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

for (int i = 0; i < n; i++) {
    cout << "\nElement " << i + 1 << ": " << arr[i];
}

int main()
{
    int n;
    cout<<"Enter the array size :";
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++)
    {
        cout<<"\nEnter the element "<<i+1<<" :";
        cin>>arr[i];
    }
    cout<<"\nArray in accending order :";
    bubblesortaccsend(arr,n);
    cout<<"\nArray in decending order :";
    bubblesortdescend(arr,n);
}

```

Output: (screenshot)

Enter the array size :2

Enter the element 1 :1

Enter the element 2 :1

Array in accending order :

Element 1 :1

Element 2 :1

Array in decending order :

Element 1: 1

Element 2: 1

Test Case: Any two (screenshot)

Enter the array size :3

Enter the element 1 :9

Enter the element 2 :0

Enter the element 3 :1

Array in accending order :

Element 1 :0

Element 2 :1

Element 3 :9

Array in decending order :

Element 1: 9

Element 2: 1

Element 3: 0

Enter the array size :4

Enter the element 1 :23

Enter the element 2 :34

Enter the element 3 :34

Enter the element 4 :1

Array in accending order :

Element 1 :1

Element 2 :23

Element 3 :34

Element 4 :34

Array in decending order :

Element 1: 34

Element 2: 34

Element 3: 23

Element 4: 1

Conclusion: In this code, we implemented bubble sort algorithms to sort elements of an array in ascending and descending order. The `bubbleSort()` methods work correctly, iterating through the elements of the array, comparing adjacent elements, and swapping them if necessary to sort the array in ascending or descending order. While bubble sort has a time complexity of $O(n^2)$, it is a simple and easy-to-implement algorithm for sorting small arrays