Practical File

Data Structures

Code- ARI254

2023-24

Submitted by:
Name-> Shubham Dev
Branch-> IIOT-B1
Enrolment no-> 01919051722

Submitted To:
Dr. Abhishek Singh
Associate Professor
USAR, GGSIPU

University School of Automation and Robotics
East Campus, GGSIP University
Surajmal Vihar, New Delhi - 110092

Name :Shubham Dev Enrolment No : 01919051722 Batch : IIOT-B1

INDEX

S.NO.	Date	Title	Teacher's Sign

Name :Shubham Dev Enrolment No : 01919051722 Batch : IIOT-B1

Lab - 1

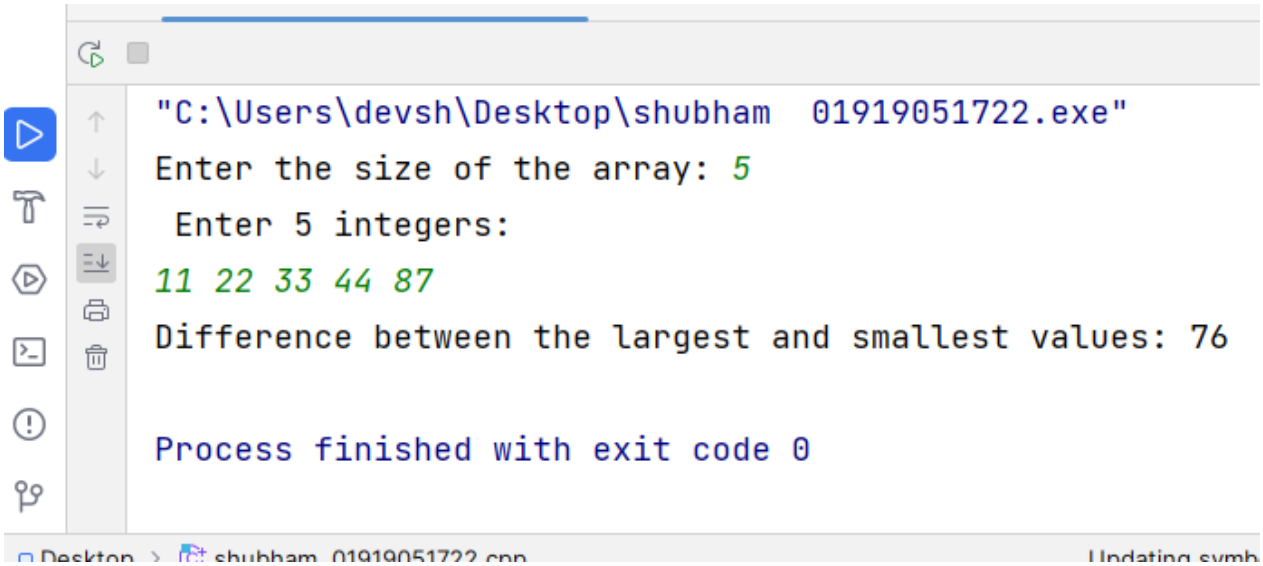
Aim- Create an array of integers with size n. Return the difference between the largest and the smallest value inside that array.

Code-

```
#include <iostream>
// Function to calculate the difference between the
largest and smallest values in an array
int differenceBetweenLargestAndSmallest(int arr[], int
n)
{
    // Initialize variables to store the largest and
smallest values
    int largest = arr[0];
    int smallest = arr[0];
    // Iterate through the array to find the largest and
smallest values
    for (int i = 1; i < n; ++i)
    {
        if (arr[i] > largest)
        {largest = arr[i];}
        if (arr[i] < smallest)
        {smallest = arr[i];}
    }
    // Calculate the difference between the largest and
smallest values
    int difference = largest - smallest;
    // Return the difference
    return difference;
}
int main()
{
    // Input the size of the array
    int n;
    std::cout << "Enter the size of the array: ";
    std::cin >> n;
    // Create an array of integers with size n
    int arr[n];
```

```
    // Input the elements of the array
    std::cout << "Enter " << n << " integers:" <<
std::endl;
    for (int i = 0; i < n; ++i)
    {
        std::cin >> arr[i];
    }
    // Call the function to calculate the difference
between the largest and smallest values
    int difference =
differenceBetweenLargestAndSmallest(arr, n);
    // Output the difference
    std::cout << "Difference between the largest and
smallest values: " << difference << std::endl;
    return 0;
}
```

Output-



Name :Shubham Dev Enrolment No : 01919051722 Batch : IIOT-B1

Lab – 2

Aim– Write a program that initializes an array with ten random integers and then prints four lines of output, containing:

a. Every odd element

b. Every element at an even index

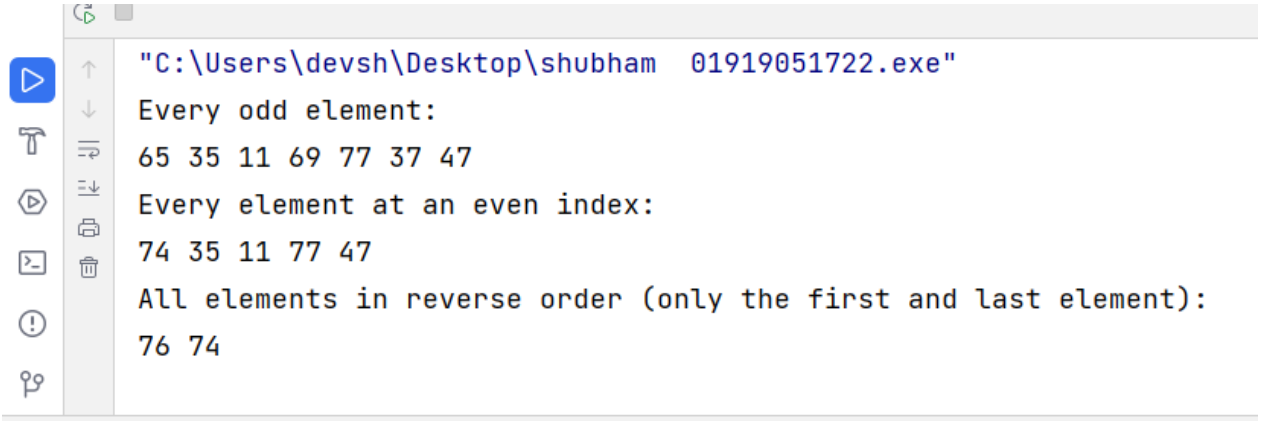
c. All elements in reverse order Only the first and last element

Code–

```
#include <iostream>
#include <cstdlib>
#include <ctime>
// Function to generate a random integer between min and
max (inclusive)
int generateRandomInt(int min, int max) {
    return min + rand() % (max - min + 1);
}
int main() {
    // Seed the random number generator
    srand(time(nullptr));
    // Initialize an array with ten random integers
    int arr[10];
    for (int i = 0; i < 10; ++i) {
        arr[i] = generateRandomInt(1, 100); // Adjust
the range as needed
    }
    // Print every odd element
    std::cout << "Every odd element:" << std::endl;
    for (int i = 0; i < 10; ++i) {
        if (arr[i] % 2 != 0) {
            std::cout << arr[i] << " ";
        }
    }
    std::cout << std::endl;
```

```
    // Print every element at an even index
    std::cout << "Every element at an even index:" <<
std::endl;
    for (int i = 0; i < 10; i += 2) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
    // Print all elements in reverse order, only the
first and last element
    std::cout << "All elements in reverse order (only
the first and last element):" << std::endl;
    std::cout << arr[9] << " " << arr[0] << std::endl;
    return 0;
}
```

Output-



Lab – 3

Aim– Write a program to read numbers in an integer array of size 5 and display the following:

a. Sum of all the elements

b. Sum of alternate elements in the array

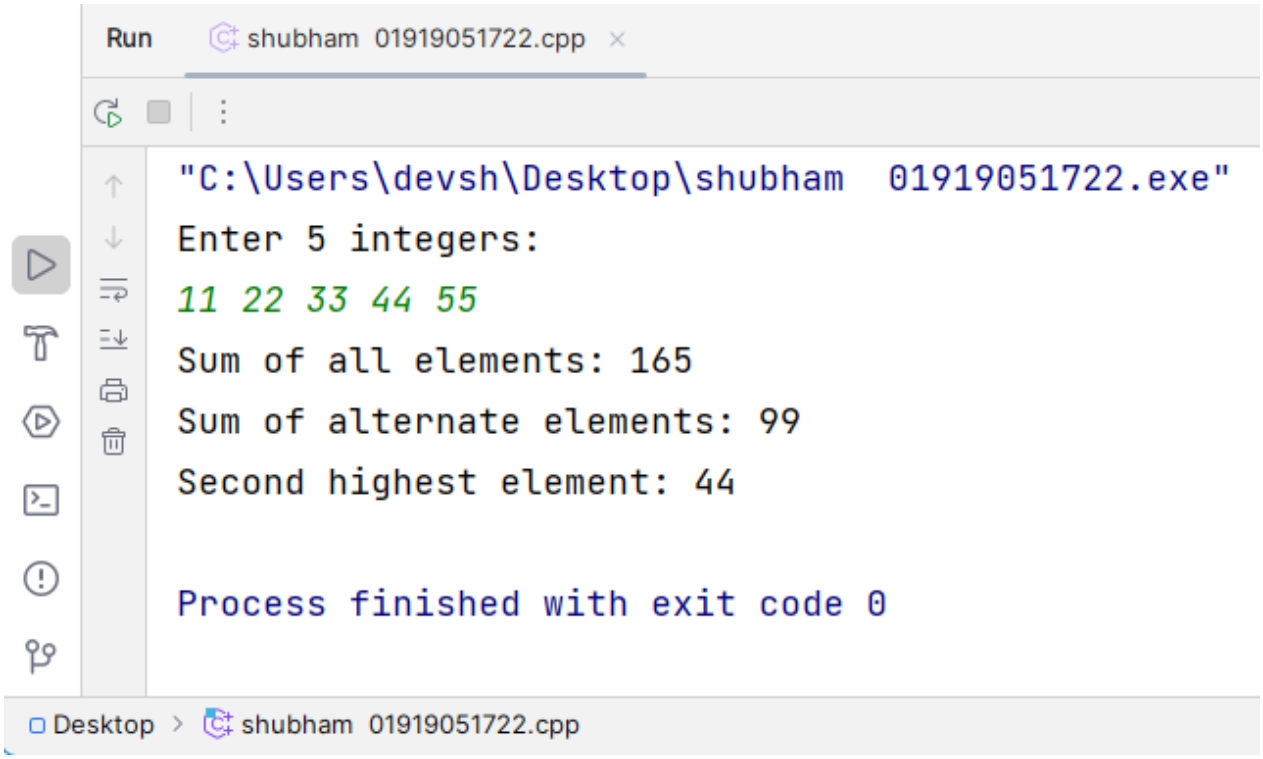
c. Second highest element in the array

Code–

```
#include <iostream>
#include <climits>
int main() {
    const int SIZE = 5;
    int arr[SIZE];
    // Input numbers into the array
    std::cout << "Enter " << SIZE << " integers:" <<
std::endl;
    for (int i = 0; i < SIZE; ++i) {
        std::cin >> arr[i];
    }
    // Calculate sum of all elements
    int sumAll = 0;
    for (int i = 0; i < SIZE; ++i) {
        sumAll += arr[i];
    }
    // Calculate sum of alternate elements
    int sumAlternate = 0;
    for (int i = 0; i < SIZE; i += 2) {
        sumAlternate += arr[i];
    }
    // Find second highest element
    int max = INT_MIN;
    int secondMax = INT_MIN;
    for (int i = 0; i < SIZE; ++i) {
        if (arr[i] > max) {
            secondMax = max;
            max = arr[i];
        } else if (arr[i] > secondMax && arr[i] != max) {
            secondMax = arr[i];
        }
```

```
    }
    // Display results
    std::cout << "Sum of all elements: " << sumAll <<
std::endl;
    std::cout << "Sum of alternate elements: " <<
sumAlternate << std::endl;
    std::cout << "Second highest element: " << secondMax
<< std::endl;
    return 0;
}
```

Output-



Name :Shubham Dev Enrolment No : 01919051722 Batch : IIOT-B1

Lab – 4

Aim–

Write a program to create a singly linked list of n nodes and perform:

a. Insertion at the beginning

b. Insertion at the end

c. Insertion at the specific location

d. Deletion from the beginning

e. Deletion from the end

f. Deletion from the specific location

Code–

```
#include <iostream>
// Node structure
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};
// Linked list class
class LinkedList {
private:
    Node* head;
public:
    // Constructor
    LinkedList() : head(nullptr) {}
    // Function to insert at the beginning
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        newNode->next = head;
        head = newNode;
    }
    // Function to insert at the end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
```

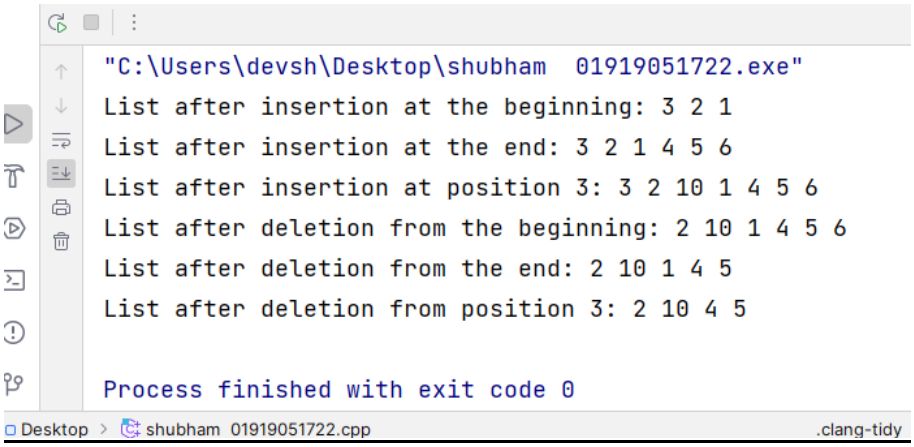
```
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    // Function to insert at a specific location
    void insertAtLocation(int val, int pos) {
        if (pos <= 0) {
            std::cout << "Invalid position." << std::endl;
            return;
        }
        if (pos == 1) {
            insertAtBeginning(val);
            return;
        }
        Node* newNode = new Node(val);
        Node* temp = head;
        for (int i = 1; i < pos - 1 && temp; ++i) {
            temp = temp->next;
        }
        if (!temp) {
            std::cout << "Position out of range." <<
std::endl;
            return;
        }
        newNode->next = temp->next;
        temp->next = newNode;
    }
    // Function to delete from the beginning
    void deleteFromBeginning() {
        if (!head) {
            std::cout << "List is empty." << std::endl;
            return;
        }
        Node* temp = head;
        head = head->next;
        delete temp;
    }
    // Function to delete from the end
    void deleteFromEnd() {
        if (!head) {
            std::cout << "List is empty." << std::endl;
```

```
            return;
        }
        if (!head->next) {
            delete head;
            head = nullptr;
            return;
        }
        Node* temp = head;
        while (temp->next->next) {
            temp = temp->next;
        }
        delete temp->next;
        temp->next = nullptr;
    }
    // Function to delete from a specific location
    void deleteFromLocation(int pos) {
        if (pos <= 0 || !head) {
            std::cout << "List is empty or invalid
position." << std::endl;
            return;
        }
        if (pos == 1) {
            deleteFromBeginning();
            return;
        }
        Node* temp = head;
        for (int i = 1; i < pos - 1 && temp; ++i) {
            temp = temp->next;
        }
        if (!temp || !temp->next) {
            std::cout << "Position out of range." <<
std::endl;
            return;
        }
        Node* toDelete = temp->next;
        temp->next = temp->next->next;
        delete toDelete;
    }
    // Function to display the linked list
    void display() {
        Node* temp = head;
        while (temp) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << std::endl;
    }
```

```
};
int main() {
    LinkedList list;
    // Insertion at the beginning
    list.insertAtBeginning(1);
    list.insertAtBeginning(2);
    list.insertAtBeginning(3);
    std::cout << "List after insertion at the beginning: ";
    list.display();
    // Insertion at the end
    list.insertAtEnd(4);
    list.insertAtEnd(5);
    list.insertAtEnd(6);
    std::cout << "List after insertion at the end: ";
    list.display();

    // Insertion at a specific location
    list.insertAtLocation(10, 3);
    std::cout << "List after insertion at position 3: ";
    list.display();
    // Deletion from the beginning
    list.deleteFromBeginning();
    std::cout << "List after deletion from the beginning: ";
    list.display();
    // Deletion from the end
    list.deleteFromEnd();
    std::cout << "List after deletion from the end: ";
    list.display();
    // Deletion from a specific location
    list.deleteFromLocation(3);
    std::cout << "List after deletion from position 3: ";
    list.display();
    return 0;
}
```

Output-



Lab – 5

Aim– Write a program to create a doubly linked list of n nodes and perform:

a. Insertion at the beginning

b. Insertion at the end

c. Insertion at the specific location

d. Deletion from the beginning

e. Deletion from the end

f. Deletion from the specific location

Code–

```
#include <iostream>
// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;
    Node(int val) : data(val), prev(nullptr),
next(nullptr) {}
};
// Doubly linked list class
class DoublyLinkedList {
private:
    Node* head;
    Node* tail;
public:
    // Constructor
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}
    // Function to insert at the beginning
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = tail = newNode;
        } else {
```

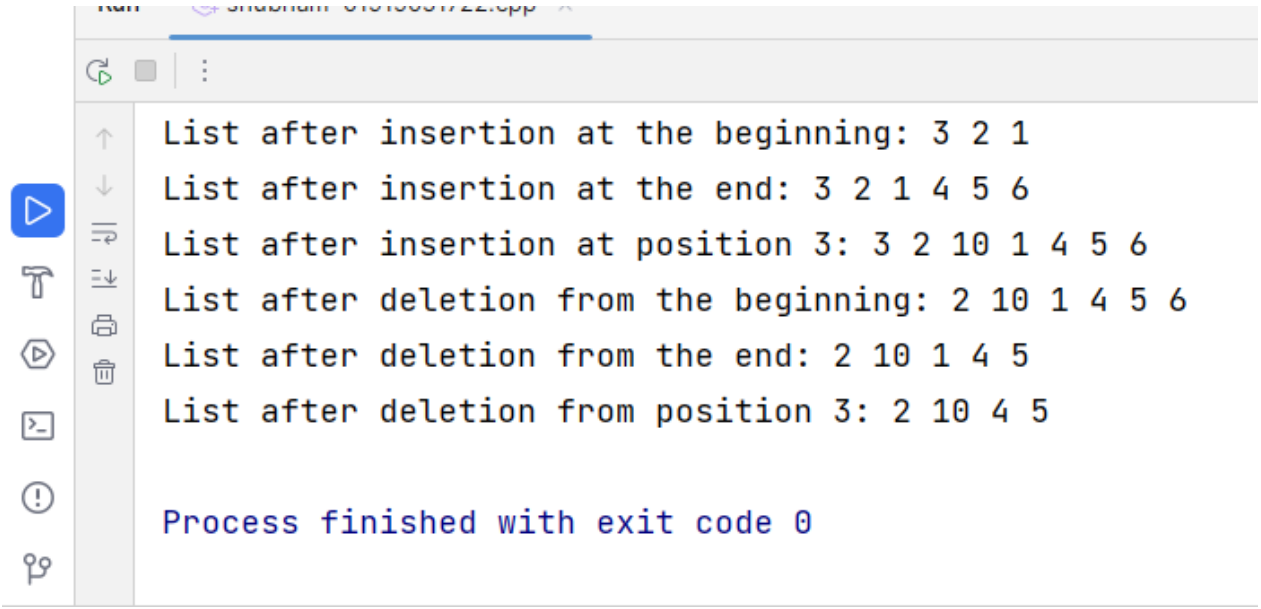
```
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }
    // Function to insert at the end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!tail) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
    // Function to insert at a specific location
    void insertAtLocation(int val, int pos) {
        if (pos <= 0) {
            std::cout << "Invalid position." <<
std::endl;
            return;
        }
        if (pos == 1) {
            insertAtBeginning(val);
            return;
        }
        Node* newNode = new Node(val);
        Node* temp = head;
        for (int i = 1; i < pos - 1 && temp; ++i) {
            temp = temp->next;
        }
        if (!temp) {
            std::cout << "Position out of range." <<
std::endl;
            return;
        }
        newNode->next = temp->next;
        newNode->prev = temp;
        if (temp->next) {
            temp->next->prev = newNode;
        }
        temp->next = newNode;
        if (!newNode->next) {
```

```
            tail = newNode;
        }
    }
    // Function to delete from the beginning
    void deleteFromBeginning() {
        if (!head) {
            std::cout << "List is empty." << std::endl;
            return;
        }
        Node* temp = head;
        head = head->next;
        if (head) {
            head->prev = nullptr;
        } else {
            tail = nullptr;
        }
        delete temp;
    }
    // Function to delete from the end
    void deleteFromEnd() {
        if (!tail) {
            std::cout << "List is empty." << std::endl;
            return;
        }
        Node* temp = tail;
        tail = tail->prev;
        if (tail) {
            tail->next = nullptr;
        } else {
            head = nullptr;
        }
        delete temp;
    }
    // Function to delete from a specific location
    void deleteFromLocation(int pos) {
        if (pos <= 0 || !head) {
            std::cout << "List is empty or invalid
position." << std::endl;
            return;
        }
        if (pos == 1) {
            deleteFromBeginning();
            return;
        }
```

```
        Node* temp = head;
        for (int i = 1; i < pos - 1 && temp; ++i) {
            temp = temp->next;
        }
        if (!temp || !temp->next) {
            std::cout << "Position out of range." <<
std::endl;
            return;
        }
        Node* toDelete = temp->next;
        temp->next = temp->next->next;
        if (temp->next) {
            temp->next->prev = temp;
        } else {
            tail = temp;
        }
        delete toDelete;
    }
    // Function to display the linked list
    void display() {
        Node* temp = head;
        while (temp) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << std::endl;
    }
};
int main() {
    DoublyLinkedList list;
    // Insertion at the beginning
    list.insertAtBeginning(1);
    list.insertAtBeginning(2);
    list.insertAtBeginning(3);
    std::cout << "List after insertion at the beginning:
";
    list.display();
    // Insertion at the end
    list.insertAtEnd(4);
    list.insertAtEnd(5);
    list.insertAtEnd(6);
    std::cout << "List after insertion at the end: ";
    list.display();
    // Insertion at a specific location
```

```
    list.insertAtLocation(10, 3);
    std::cout << "List after insertion at position 3: ";
    list.display();
    // Deletion from the beginning
    list.deleteFromBeginning();
    std::cout << "List after deletion from the
beginning: ";
    list.display();
    // Deletion from the end
    list.deleteFromEnd();
    std::cout << "List after deletion from the end: ";
    list.display();
    // Deletion from a specific location
    list.deleteFromLocation(3);
    std::cout << "List after deletion from position 3:
";
    list.display();
    return 0;
}
```

Output-



Lab – 6

Aim– Write a program to create a circular linked list of n nodes and perform:

a. Insertion at the beginning

b. Insertion at the end

c. Insertion at the specific location

d. Deletion from the beginning

e. Deletion from the end

f. Deletion from the specific location

Code–

```
#include <iostream>
// Node structure
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};
// Circular linked list class
class CircularLinkedList {
private:
    Node* head;
public:
    // Constructor
    CircularLinkedList() : head(nullptr) {}
    // Function to insert at the beginning
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            newNode->next = newNode; // Circular link to
itself
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != head) {
```

```
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
        head = newNode;
    }
}
// Function to insert at the end
void insertAtEnd(int val) {
    Node* newNode = new Node(val);
    if (!head) {
        newNode->next = newNode; // Circular link to
itself
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
    }
}
// Function to insert at a specific location
void insertAtLocation(int val, int pos) {
    if (pos <= 0) {
        std::cout << "Invalid position." << std::endl;
        return;
    }
    if (pos == 1) {
        insertAtBeginning(val);
        return;
    }
    Node* newNode = new Node(val);
    Node* temp = head;
    for (int i = 1; i < pos - 1 && temp && temp->next
!= head; ++i) {
        temp = temp->next;
    }
    if (!temp || temp->next == head) {
        std::cout << "Position out of range." <<
std::endl;
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
```

```
    }
    // Function to delete from the beginning
    void deleteFromBeginning() {
        if (!head) {
            std::cout << "List is empty." << std::endl;
            return;
        }
        Node* temp = head;
        if (head->next == head) { // Only one node
            delete head;
            head = nullptr;
        } else {
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = head->next;
            delete head;
            head = temp->next;
        }
    }
    // Function to delete from the end
    void deleteFromEnd() {
        if (!head) {
            std::cout << "List is empty." << std::endl;
            return;
        }
        if (head->next == head) { // Only one node
            delete head;
            head = nullptr;
            return;
        }
        Node* prev = nullptr;
        Node* temp = head;
        while (temp->next != head) {
            prev = temp;
            temp = temp->next;
        }
        prev->next = head; // Make the previous node point
to head
        delete temp;
    }
    // Function to delete from a specific location
    void deleteFromLocation(int pos) {
        if (pos <= 0 || !head) {
            std::cout << "List is empty or invalid
position." << std::endl;
```

```
            return;
        }
        if (pos == 1) {
            deleteFromBeginning();
            return;
        }
        Node* temp = head;
        Node* prev = nullptr;
        for (int i = 1; i < pos && temp && temp->next !=
head; ++i) {
            prev = temp;
            temp = temp->next;
        }
        if (!temp || temp->next == head) {
            std::cout << "Position out of range." <<
std::endl;
            return;
        }
        prev->next = temp->next;
        delete temp;
    }
    // Function to display the circular linked list
    void display() {
        if (!head) {
            std::cout << "List is empty." << std::endl;
            return;
        }
        Node* temp = head;
        do {
            std::cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        std::cout << std::endl;
    }
};
int main() {
    CircularLinkedList list;
    // Insertion at the beginning
    list.insertAtBeginning(1);
    list.insertAtBeginning(2);
    list.insertAtBeginning(3);
    std::cout << "List after insertion at the beginning:
";
    list.display();

    // Insertion at the end
```

```
    list.insertAtEnd(4);
    list.insertAtEnd(5);
    list.insertAtEnd(6);
    std::cout << "List after insertion at the end: ";
    list.display();

    // Insertion at a specific location
    list.insertAtLocation(10, 3);
    std::cout << "List after insertion at position 3: ";
    list.display();
    // Deletion from the beginning
    list.deleteFromBeginning();
    std::cout << "List after deletion from the beginning:
";
    list.display();
    // Deletion from the end
    list.deleteFromEnd();
    std::cout << "List after deletion from the end: ";
    list.display();
    // Deletion from a specific location
    list.deleteFromLocation(3);
    std::cout << "List after deletion from position 3: ";
    list.display();
    return 0;
}
```

Output-

```
List after insertion at the beginning: 3 2 1
List after insertion at the end: 3 2 1 4 5 6
List after insertion at position 3: 3 2 10 1 4 5 6
List after deletion from the beginning: 2 10 1 4 5 6
List after deletion from the end: 2 10 1 4 5
List after deletion from position 3: 2 10 4 5

Process finished with exit code 0
```