**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**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.NO.** | **Date** | **Title** | **Teacher’s Sign** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

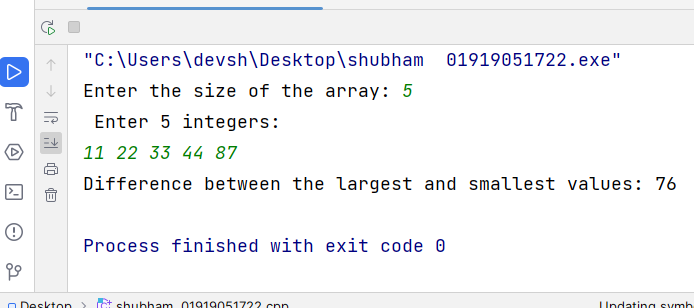
**Lab – 1**

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-



**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

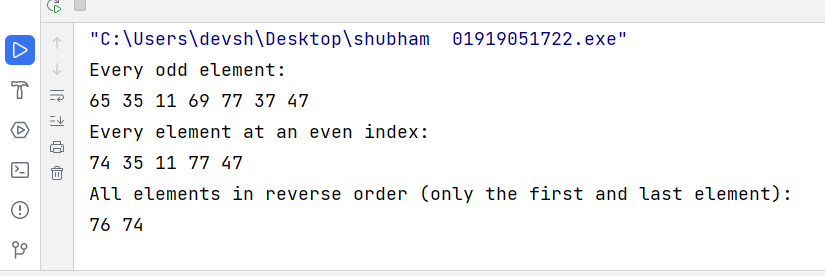
b. Every element at an even index

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

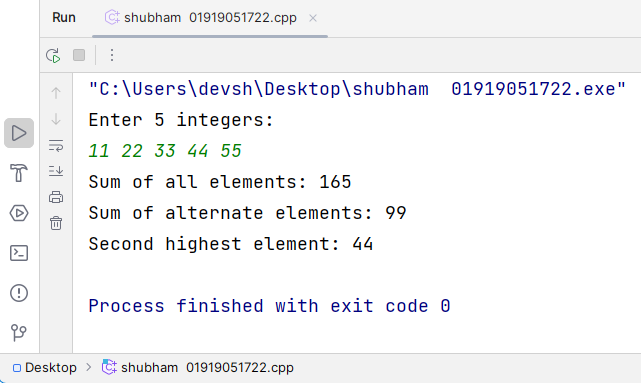
b. Sum of alternate elements in the array

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-



**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

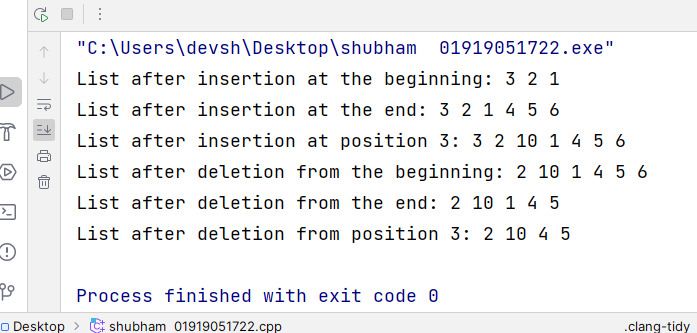
e. Deletion from the end

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

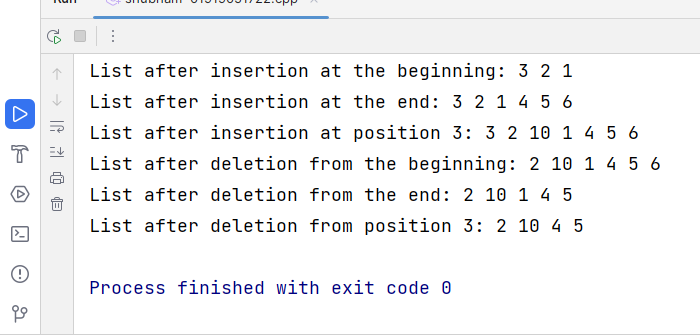
e. Deletion from the end

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

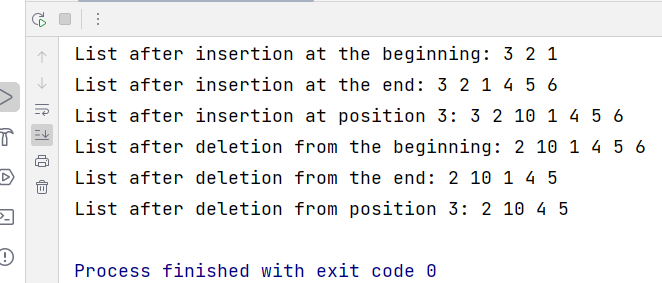
e. Deletion from the end

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;  
}

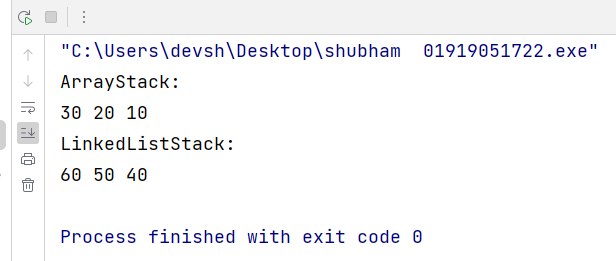
Output-



**Lab – 7**

Aim- Write a program to implement stack using arrays and linked lists.

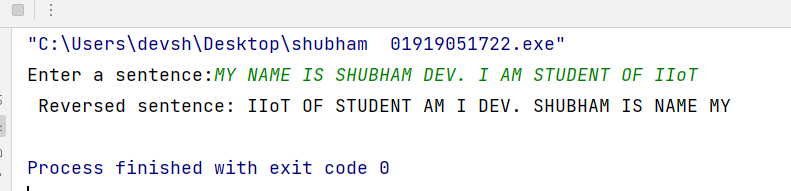
Code- #include <iostream>  
*// Node structure for linked list implementation*struct Node {  
 int data;  
 Node\* next;  
};  
*// Stack class using array*class ArrayStack {  
private:  
 static const int MAX\_SIZE = 100;  
 int arr[MAX\_SIZE];  
 int top;  
public:  
 ArrayStack() {  
 top = -1; *// Initialize top to -1 to indicate an empty stack* }  
 *// Function to push element onto the stack* void push(int val) {  
 if (top >= MAX\_SIZE - 1) {  
 std::cout << "Stack Overflow\n";  
 return;  
 }  
 arr[++top] = val;  
 }  
 *// Function to pop element from the stack* int pop() {  
 if (top < 0) {  
 std::cout << "Stack Underflow\n";  
 return -1; *// Returning -1 for underflow* }  
 return arr[top--];  
 }  
 *// Function to check if stack is empty* bool isEmpty() {  
 return top < 0;  
 }  
 *// Function to get the top element of the stack* int peek() {  
 if (top < 0) {  
 std::cout << "Stack is empty\n";  
 return -1; *// Returning -1 for empty stack* }  
 return arr[top];  
 }  
};  
*// Stack class using linked list*class LinkedListStack {  
private:  
 Node\* top;  
public:  
 LinkedListStack() {  
 top = nullptr; *// Initialize top pointer to nullptr to indicate an empty stack* }  
 *// Function to push element onto the stack* void push(int val) {  
 Node\* newNode = new Node;  
 newNode->data = val;  
 newNode->next = top;  
 top = newNode;  
 }  
 *// Function to pop element from the stack* int pop() {  
 if (top == nullptr) {  
 std::cout << "Stack Underflow\n";  
 return -1; *// Returning -1 for underflow* }  
 int val = top->data;  
 Node\* temp = top;  
 top = top->next;  
 delete temp;  
 return val;  
 }  
 *// Function to check if stack is empty* bool isEmpty() {  
 return top == nullptr;  
 }  
 *// Function to get the top element of the stack* int peek() {  
 if (top == nullptr) {  
 std::cout << "Stack is empty\n";  
 return -1; *// Returning -1 for empty stack* }  
 return top->data;  
 }  
};  
int main() {  
 *// ArrayStack implementation* ArrayStack arrStack;  
 arrStack.push(10);  
 arrStack.push(20);  
 arrStack.push(30);  
 std::cout << "ArrayStack:\n";  
 while (!arrStack.isEmpty()) {  
 std::cout << arrStack.pop() << " ";  
 }  
 std::cout << std::endl;  
 *// LinkedListStack implementation* LinkedListStack llStack;  
 llStack.push(40);  
 llStack.push(50);  
 llStack.push(60);  
 std::cout << "LinkedListStack:\n";  
 while (!llStack.isEmpty()) {  
 std::cout << llStack.pop() << " ";  
 }  
 std::cout << std::endl;  
 return 0;  
}

Output- 

**Lab – 8**

Aim- Write a program to reverse a sentence using stack.

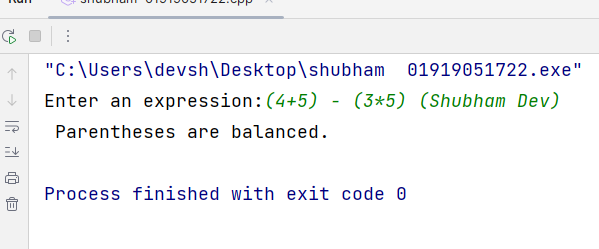
Code- #include <iostream>  
#include <stack>  
#include <string>  
  
using namespace std;  
  
*// Function to reverse a sentence using stack*string reverseSentence(string sentence) {  
 stack<string> wordsStack;  
 string word = "";  
  
 *// Iterate through each character in the sentence* for (char& c : sentence) {  
 *// If the character is not a space, append it to the current word* if (c != ' ') {  
 word += c;  
 }  
 *// If the character is a space and the current word is not empty, push it onto the stack* else if (!word.empty()) {  
 wordsStack.push(word);  
 word = ""; *// Reset the current word* }  
 }  
  
 *// Push the last word onto the stack (if not empty)* if (!word.empty()) {  
 wordsStack.push(word);  
 }  
  
 string reversedSentence = "";  
  
 *// Pop each word from the stack and append it to the reversed sentence* while (!wordsStack.empty()) {  
 reversedSentence += wordsStack.top() + " ";  
 wordsStack.pop();  
 }  
  
 *// Remove the trailing space* if (!reversedSentence.empty()) {  
 reversedSentence.pop\_back();  
 }  
  
 return reversedSentence;  
}  
  
int main() {  
 string sentence;  
  
 *// Input the sentence* cout << "Enter a sentence: ";  
 getline(cin, sentence);  
  
 *// Reverse the sentence* string reversedSentence = reverseSentence(sentence);  
  
 *// Output the reversed sentence* cout << "Reversed sentence: " << reversedSentence << endl;  
  
 return 0;  
}

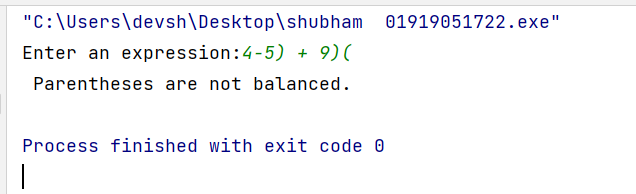
Output- 

**Lab – 9**

Aim- Write a program to check for balanced parentheses in a given expression.

Code- #include <iostream>  
#include <stack>  
#include <string>  
  
using namespace std;  
  
*// Function to check for balanced parentheses*bool areParenthesesBalanced(string expr) {  
 stack<char> s;  
  
 *// Iterate through each character in the expression* for (char& c : expr) {  
 *// If the character is an opening parenthesis, push it onto the stack* if (c == '(' || c == '[' || c == '{') {  
 s.push(c);  
 }  
 *// If the character is a closing parenthesis* else if (c == ')' || c == ']' || c == '}') {  
 *// If the stack is empty, there's no matching opening parenthesis, return false* if (s.empty()) {  
 return false;  
 }  
  
 *// Get the top element from the stack* char topChar = s.top();  
 s.pop();  
  
 *// Check if the current closing parenthesis matches the top element of the stack* if ((c == ')' && topChar != '(') || (c == ']' && topChar != '[') || (c == '}' && topChar != '{')) {  
 return false;  
 }  
 }  
 }  
  
 *// If the stack is empty after processing all characters, return true* return s.empty();  
}  
  
int main() {  
 string expr;  
  
 *// Input the expression* cout << "Enter an expression: ";  
 getline(cin, expr);  
  
 *// Check if parentheses are balanced* if (areParenthesesBalanced(expr)) {  
 cout << "Parentheses are balanced.\n";  
 } else {  
 cout << "Parentheses are not balanced.\n";  
 }  
  
 return 0;  
}

Output- 

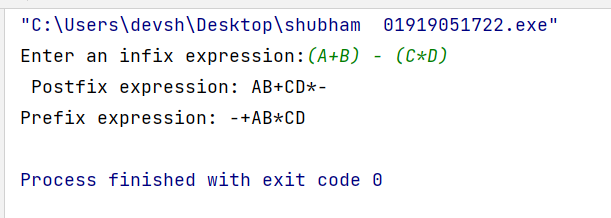


**Lab – 10**

Aim- Write a program to convert infix expressions to prefix and postfix expressions.

Code- #include <iostream>  
#include <stack>  
#include <string>  
#include <algorithm>  
using namespace std;  
*// Function to check if a character is an operator*bool isOperator(char c) {  
 return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');  
}  
*// Function to check if a character is an operand*bool isOperand(char c) {  
 return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');  
}  
*// Function to get the precedence of an operator*int getPrecedence(char op) {  
 if (op == '^')  
 return 3;  
 else if (op == '\*' || op == '/')  
 return 2;  
 else if (op == '+' || op == '-')  
 return 1;  
 else  
 return -1;  
}  
*// Function to convert infix expression to postfix expression*string infixToPostfix(string infix) {  
 stack<char> s;  
 string postfix = "";  
 *// Iterate through each character in the infix expression* for (char& c : infix) {  
 *// If character is an operand, add it to the postfix expression* if (isOperand(c)) {  
 postfix += c;}  
 *// If character is an opening parenthesis, push it onto the stack* else if (c == '(') {  
 s.push(c);}  
 *// If character is a closing parenthesis, pop and add operators from the stack to the postfix expression until an opening parenthesis is encountered* else if (c == ')') {  
 while (!s.empty() && s.top() != '(') {  
 postfix += s.top();  
 s.pop();  
 }  
 if (!s.empty() && s.top() == '(') {  
 s.pop();  
 }}  
 *// If character is an operator* else if (isOperator(c)) {  
 *// Pop and add operators from the stack to the postfix expression until an operator with lower precedence or an opening parenthesis is encountered* while (!s.empty() && getPrecedence(c) <= getPrecedence(s.top())) {  
 postfix += s.top();  
 s.pop();  
 }  
 s.push(c); *// Push the current operator onto the stack* }}  
  
 *// Pop and add remaining operators from the stack to the postfix expression* while (!s.empty()) {  
 postfix += s.top();  
 s.pop();  
 }  
 return postfix;  
}  
  
*// Function to convert infix expression to prefix expression*string infixToPrefix(string infix) {  
 reverse(infix.begin(), infix.end());  
 *// Replace opening parentheses with closing parentheses and vice versa* for (char& c : infix) {  
 if (c == '(')  
 c = ')';  
 else if (c == ')')  
 c = '(';  
 }  
 string postfix = infixToPostfix(infix);  
 reverse(postfix.begin(), postfix.end());  
 return postfix;  
}  
int main() {  
 string infixExpression;  
 *// Input the infix expression* cout << "Enter an infix expression: ";  
 getline(cin, infixExpression);  
 *// Convert infix expression to postfix* string postfixExpression = infixToPostfix(infixExpression);  
 cout << "Postfix expression: " << postfixExpression << endl;  
 *// Convert infix expression to prefix* string prefixExpression = infixToPrefix(infixExpression);  
 cout << "Prefix expression: " << prefixExpression << endl;  
 return 0;  
}

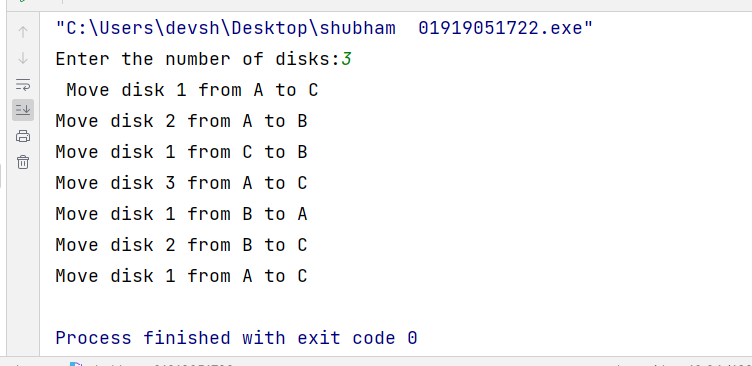
Output-



**Lab – 11**

Aim- Write a program to implement Tower of Hanoi using stacks.

Code- #include <iostream>  
#include <stack>  
  
using namespace std;  
  
*// Function to move a disk from one tower to another*void moveDisk(stack<int>& source, stack<int>& dest, char sourceName, char destName) {  
 int topDisk = source.top();  
 source.pop();  
 dest.push(topDisk);  
 cout << "Move disk " << topDisk << " from " << sourceName << " to " << destName << endl;  
}  
  
*// Function to perform Tower of Hanoi recursively*void towerOfHanoi(int numDisks, stack<int>& source, stack<int>& aux, stack<int>& dest, char sourceName, char auxName, char destName) {  
 if (numDisks == 1) {  
 moveDisk(source, dest, sourceName, destName);  
 } else {  
 towerOfHanoi(numDisks - 1, source, dest, aux, sourceName, destName, auxName);  
 moveDisk(source, dest, sourceName, destName);  
 towerOfHanoi(numDisks - 1, aux, source, dest, auxName, sourceName, destName);  
 }  
}  
  
int main() {  
 int numDisks;  
  
 *// Input the number of disks* cout << "Enter the number of disks: ";  
 cin >> numDisks;  
  
 *// Create stacks for each tower* stack<int> source, aux, dest;  
  
 *// Initialize the source tower with disks* for (int i = numDisks; i >= 1; --i) {  
 source.push(i);  
 }  
  
 *// Call the Tower of Hanoi function* towerOfHanoi(numDisks, source, aux, dest, 'A', 'B', 'C');  
  
 return 0;  
}

Output- 

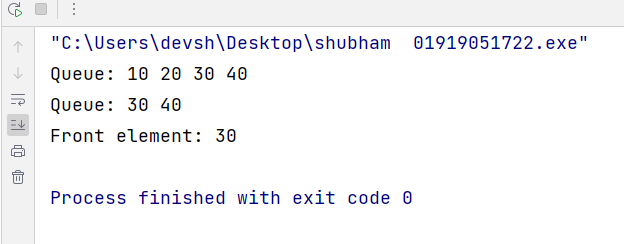
**Lab – 12**

Aim- Write a program to implement Linear Queue using

1. Array

Code-

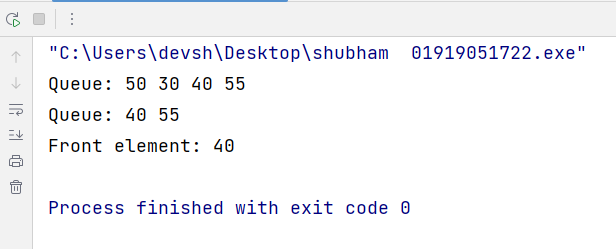
#include <iostream>  
  
using namespace std;  
  
#define **MAX\_SIZE** 100  
  
class ArrayQueue {  
private:  
 int queue[**MAX\_SIZE**];  
 int front, rear;  
  
public:  
 ArrayQueue() {  
 front = -1;  
 rear = -1;  
 }  
  
 *// Function to check if the queue is empty* bool isEmpty() {  
 return front == -1 && rear == -1;  
 }  
  
 *// Function to check if the queue is full* bool isFull() {  
 return rear == **MAX\_SIZE** - 1;  
 }  
  
 *// Function to add an element to the queue* void enqueue(int data) {  
 if (isFull()) {  
 cout << "Queue Overflow\n";  
 return;  
 }  
 if (isEmpty()) {  
 front = rear = 0;  
 } else {  
 rear++;  
 }  
 queue[rear] = data;  
 }  
  
 *// Function to remove an element from the queue* void dequeue() {  
 if (isEmpty()) {  
 cout << "Queue Underflow\n";  
 return;  
 }  
 if (front == rear) {  
 front = rear = -1;  
 } else {  
 front++;  
 }  
 }  
  
 *// Function to get the front element of the queue* int peek() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return -1;  
 }  
 return queue[front];  
 }  
  
 *// Function to display the queue* void display() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return;  
 }  
 cout << "Queue: ";  
 for (int i = front; i <= rear; i++) {  
 cout << queue[i] << " ";  
 }  
 cout << endl;  
 }  
};  
  
int main() {  
 ArrayQueue q;  
  
 q.enqueue(10);  
 q.enqueue(20);  
 q.enqueue(30);  
 q.enqueue(40);  
  
 q.display(); *// Output: Queue: 10 20 30 40* q.dequeue();  
 q.dequeue();  
 q.display(); *// Output: Queue: 30 40* cout << "Front element: " << q.peek() << endl; *// Output: Front element: 30* return 0;  
}

Output- 

1. Linked Lists.

Code- #include <iostream>  
using namespace std;  
*// Node structure for Linked List*struct Node {  
 int data;  
 Node\* next;  
};  
class LinkedListQueue {  
private:  
 Node\* front;  
 Node\* rear;  
public:  
 LinkedListQueue() {  
 front = nullptr;  
 rear = nullptr;  
 }  
 *// Function to check if the queue is empty* bool isEmpty() {  
 return front == nullptr && rear == nullptr;  
 }  
 *// Function to add an element to the queue* void enqueue(int data) {  
 Node\* newNode = new Node;  
 newNode->data = data;  
 newNode->next = nullptr;  
 if (isEmpty()) {  
 front = rear = newNode;  
 } else {  
 rear->next = newNode;  
 rear = newNode;  
 }  
 }  
 *// Function to remove an element from the queue* void dequeue() {  
 if (isEmpty()) {  
 cout << "Queue Underflow\n";  
 return;  
 }  
 if (front == rear) {  
 delete front;  
 front = rear = nullptr;  
 } else {  
 Node\* temp = front;  
 front = front->next;  
 delete temp;  
 }  
 }  
 *// Function to get the front element of the queue* int peek() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return -1;  
 }  
 return front->data;  
 }  
 *// Function to display the queue* void display() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return;  
 }  
 cout << "Queue: ";  
 Node\* current = front;  
 while (current != nullptr) {  
 cout << current->data << " ";  
 current = current->next;  
 }  
 cout << endl;  
 }  
};  
int main() {  
 LinkedListQueue q;  
  
 q.enqueue(50);  
 q.enqueue(30);  
 q.enqueue(40);  
 q.enqueue(55);  
  
 q.display(); *// Output: Queue: 50 30 40 55* q.dequeue();  
 q.dequeue();  
  
 q.display(); *// Output: Queue: 40 55* cout << "Front element: " << q.peek() << endl; *// Output: Front element: 40* return 0;  
}

Output-



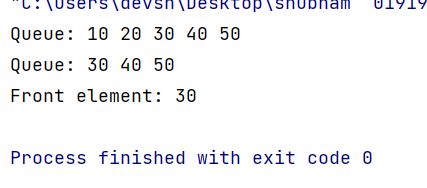
**Lab – 13**

Aim- Write a program to implement Circular Queue using

1. Array

Code- #include <iostream>  
  
using namespace std;  
  
#define **MAX\_SIZE** 5  
  
class CircularQueueArray {  
private:  
 int queue[**MAX\_SIZE**];  
 int front, rear;  
  
public:  
 CircularQueueArray() {  
 front = -1;  
 rear = -1;  
 }  
  
 *// Function to check if the queue is empty* bool isEmpty() {  
 return front == -1 && rear == -1;  
 }  
  
 *// Function to check if the queue is full* bool isFull() {  
 return (rear + 1) % **MAX\_SIZE** == front;  
 }  
  
 *// Function to add an element to the queue* void enqueue(int data) {  
 if (isFull()) {  
 cout << "Queue Overflow\n";  
 return;  
 }  
 if (isEmpty()) {  
 front = rear = 0;  
 } else {  
 rear = (rear + 1) % **MAX\_SIZE**;  
 }  
 queue[rear] = data;  
 }  
  
 *// Function to remove an element from the queue* void dequeue() {  
 if (isEmpty()) {  
 cout << "Queue Underflow\n";  
 return;  
 }  
 if (front == rear) {  
 front = rear = -1;  
 } else {  
 front = (front + 1) % **MAX\_SIZE**;  
 }  
 }  
  
 *// Function to get the front element of the queue* int peek() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return -1;  
 }  
 return queue[front];  
 }  
  
 *// Function to display the queue* void display() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return;  
 }  
 cout << "Queue: ";  
 int i = front;  
 do {  
 cout << queue[i] << " ";  
 i = (i + 1) % **MAX\_SIZE**;  
 } while (i != (rear + 1) % **MAX\_SIZE**);  
 cout << endl;  
 }  
};  
  
int main() {  
 CircularQueueArray cq;  
  
 cq.enqueue(10);  
 cq.enqueue(20);  
 cq.enqueue(30);  
 cq.enqueue(40);  
 cq.enqueue(50); *// Overflow* cq.display(); *// Output: Queue: 10 20 30 40* cq.dequeue();  
 cq.dequeue();  
  
 cq.display(); *// Output: Queue: 30 40* cout << "Front element: " << cq.peek() << endl; *// Output: Front element: 30* return 0;  
}

Output-

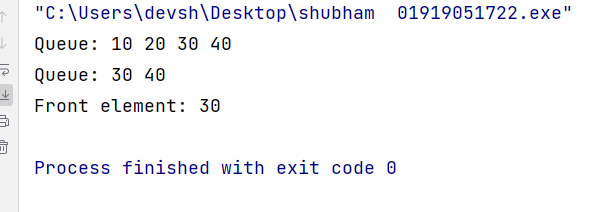


Aim- Implement Circular Queue using

1. Linked Lists.

Code- #include <iostream>  
  
using namespace std;  
  
*// Node structure for the Circular Queue Linked List*struct Node {  
 int data;  
 Node\* next;  
  
 Node(int val) {  
 data = val;  
 next = nullptr;  
 }  
};  
  
class CircularQueueLinkedList {  
private:  
 Node\* front;  
 Node\* rear;  
  
public:  
 CircularQueueLinkedList() {  
 front = nullptr;  
 rear = nullptr;  
 }  
  
 *// Function to check if the queue is empty* bool isEmpty() {  
 return front == nullptr;  
 }  
  
 *// Function to add an element to the queue* void enqueue(int data) {  
 Node\* newNode = new Node(data);  
 if (isEmpty()) {  
 front = rear = newNode;  
 rear->next = front; *// Make the new node point to itself in a circular manner* } else {  
 rear->next = newNode;  
 rear = newNode;  
 rear->next = front; *// Make rear point back to front in a circular manner* }  
 }  
  
 *// Function to remove an element from the queue* void dequeue() {  
 if (isEmpty()) {  
 cout << "Queue Underflow\n";  
 return;  
 }  
 if (front == rear) {  
 delete front;  
 front = rear = nullptr;  
 } else {  
 Node\* temp = front;  
 front = front->next;  
 rear->next = front; *// Update rear to point back to front after dequeuing* delete temp;  
 }  
 }  
  
 *// Function to get the front element of the queue* int peek() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return -1;  
 }  
 return front->data;  
 }  
  
 *// Function to display the queue* void display() {  
 if (isEmpty()) {  
 cout << "Queue is empty\n";  
 return;  
 }  
 cout << "Queue: ";  
 Node\* current = front;  
 do {  
 cout << current->data << " ";  
 current = current->next;  
 } while (current != front);  
 cout << endl;  
 }  
};  
  
int main() {  
 CircularQueueLinkedList cq;  
  
 cq.enqueue(10);  
 cq.enqueue(20);  
 cq.enqueue(30);  
 cq.enqueue(40);  
  
 cq.display(); *// Output: Queue: 10 20 30 40* cq.dequeue();  
 cq.dequeue();  
  
 cq.display(); *// Output: Queue: 30 40* cout << "Front element: " << cq.peek() << endl; *// Output: Front element: 30* return 0;  
}

Output-

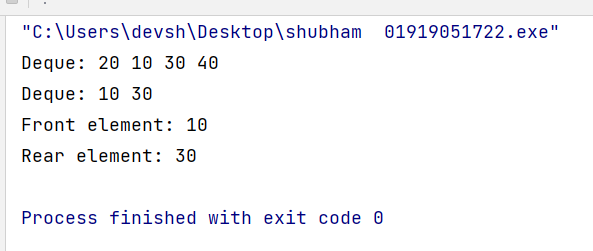


**Lab – 14**

Aim- Write a program to implement Doubly Ended Queue using

1. Array

Code- #include <iostream>  
  
using namespace std;  
  
#define **MAX\_SIZE** 100  
  
class ArrayDeque {  
private:  
 int deque[**MAX\_SIZE**];  
 int front, rear;  
  
public:  
 ArrayDeque() {  
 front = -1;  
 rear = -1;  
 }  
  
 *// Function to check if the deque is empty* bool isEmpty() {  
 return front == -1 && rear == -1;  
 }  
  
 *// Function to check if the deque is full* bool isFull() {  
 return (front == 0 && rear == **MAX\_SIZE** - 1) || (front == rear + 1);  
 }  
  
 *// Function to add an element at the front of the deque* void addFront(int data) {  
 if (isFull()) {  
 cout << "Deque Overflow\n";  
 return;  
 }  
 if (isEmpty()) {  
 front = rear = 0;  
 } else if (front == 0) {  
 front = **MAX\_SIZE** - 1;  
 } else {  
 front--;  
 }  
 deque[front] = data;  
 }  
  
 *// Function to add an element at the rear of the deque* void addRear(int data) {  
 if (isFull()) {  
 cout << "Deque Overflow\n";  
 return;  
 }  
 if (isEmpty()) {  
 front = rear = 0;  
 } else if (rear == **MAX\_SIZE** - 1) {  
 rear = 0;  
 } else {  
 rear++;  
 }  
 deque[rear] = data;  
 }  
  
 *// Function to remove an element from the front of the deque* void removeFront() {  
 if (isEmpty()) {  
 cout << "Deque Underflow\n";  
 return;  
 }  
 if (front == rear) {  
 front = rear = -1;  
 } else if (front == **MAX\_SIZE** - 1) {  
 front = 0;  
 } else {  
 front++;  
 }  
 }  
  
 *// Function to remove an element from the rear of the deque* void removeRear() {  
 if (isEmpty()) {  
 cout << "Deque Underflow\n";  
 return;  
 }  
 if (front == rear) {  
 front = rear = -1;  
 } else if (rear == 0) {  
 rear = **MAX\_SIZE** - 1;  
 } else {  
 rear--;  
 }  
 }  
  
 *// Function to get the front element of the deque* int peekFront() {  
 if (isEmpty()) {  
 cout << "Deque is empty\n";  
 return -1;  
 }  
 return deque[front];  
 }  
  
 *// Function to get the rear element of the deque* int peekRear() {  
 if (isEmpty()) {  
 cout << "Deque is empty\n";  
 return -1;  
 }  
 return deque[rear];  
 }  
  
 *// Function to display the deque* void display() {  
 if (isEmpty()) {  
 cout << "Deque is empty\n";  
 return;  
 }  
 cout << "Deque: ";  
 int i = front;  
 while (true) {  
 cout << deque[i] << " ";  
 if (i == rear) {  
 break;  
 }  
 i = (i + 1) % **MAX\_SIZE**;  
 }  
 cout << endl;  
 }  
};  
  
int main() {  
 ArrayDeque dq;  
  
 dq.addFront(10);  
 dq.addFront(20);  
 dq.addRear(30);  
 dq.addRear(40);  
  
 dq.display(); *// Output: Deque: 20 10 30 40* dq.removeFront();  
 dq.removeRear();  
  
 dq.display(); *// Output: Deque: 10 30* cout << "Front element: " << dq.peekFront() << endl; *// Output: Front element: 10* cout << "Rear element: " << dq.peekRear() << endl; *// Output: Rear element: 30* return 0;  
}

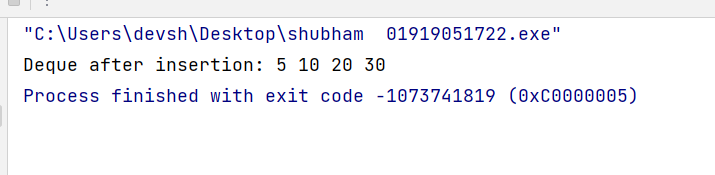
Output- 

Aim- Write a program to implement Doubly Ended Queue using

1. Linked Lists

Code- #include <iostream>  
  
using namespace std;  
  
*// Node structure for the doubly linked list*struct Node {  
 int data;  
 Node\* prev;  
 Node\* next;  
};  
  
*// Deque class*class Deque {  
private:  
 Node\* front;  
 Node\* rear;  
 int size;  
  
public:  
 Deque() {  
 front = nullptr;  
 rear = nullptr;  
 size = 0;  
 }  
  
 *// Check if the deque is empty* bool isEmpty() {  
 return size == 0;  
 }  
  
 *// Insert an element at the front of the deque* void pushFront(int data) {  
 Node\* newNode = new Node;  
 newNode->data = data;  
  
 if (isEmpty()) {  
 front = rear = newNode;  
 } else {  
 newNode->next = front;  
 front->prev = newNode;  
 front = newNode;  
 }  
 size++;  
 }  
  
 *// Insert an element at the back of the deque* void pushBack(int data) {  
 Node\* newNode = new Node;  
 newNode->data = data;  
  
 if (isEmpty()) {  
 front = rear = newNode;  
 } else {  
 rear->next = newNode;  
 newNode->prev = rear;  
 rear = newNode;  
 }  
 size++;  
 }  
  
 *// Remove an element from the front of the deque* int popFront() {  
 if (isEmpty()) {  
 cout << "Deque is empty!" << endl;  
 return -1;  
 }  
  
 Node\* temp = front;  
 int data = temp->data;  
  
 if (front == rear) {  
 front = rear = nullptr;  
 } else {  
 front = front->next;  
 front->prev = nullptr;  
 }  
  
 delete temp;  
 size--;  
 return data;  
 }  
  
 *// Remove an element from the back of the deque* int popBack() {  
 if (isEmpty()) {  
 cout << "Deque is empty!" << endl;  
 return -1;  
 }  
  
 Node\* temp = rear;  
 int data = temp->data;  
  
 if (front == rear) {  
 front = rear = nullptr;  
 } else {  
 rear = rear->prev;  
 rear->next = nullptr;  
 }  
  
 delete temp;  
 size--;  
 return data;  
 }  
  
 *// Get the front element of the deque* int getFront() {  
 if (isEmpty()) {  
 cout << "Deque is empty!" << endl;  
 return -1;  
 }  
 return front->data;  
 }  
  
 *// Get the back element of the deque* int getBack() {  
 if (isEmpty()) {  
 cout << "Deque is empty!" << endl;  
 return -1;  
 }  
 return rear->data;  
 }  
  
 *// Print the contents of the deque* void printDeque() {  
 if (isEmpty()) {  
 cout << "Deque is empty!" << endl;  
 return;  
 }  
  
 Node\* temp = front;  
 while (temp != nullptr) {  
 cout << temp->data << " ";  
 temp = temp->next;  
 }  
 cout << endl;  
 }  
};  
  
int main() {  
 Deque deque;  
  
 deque.pushFront(10);  
 deque.pushBack(20);  
 deque.pushFront(5);  
 deque.pushBack(30);  
  
 cout << "Deque after insertion: ";  
 deque.printDeque();  
  
 cout << "Front element: " << deque.getFront() << endl;  
 cout << "Back element: " << deque.getBack() << endl;  
  
 deque.popFront();  
 deque.popBack();  
  
 cout << "Deque after deletion: ";  
 deque.printDeque();  
  
 return 0;  
}

Output-

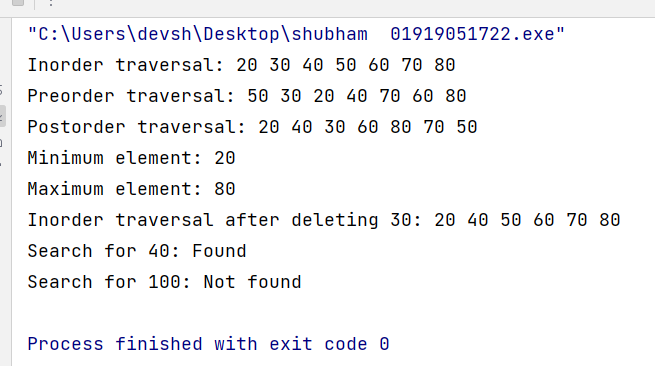


**Lab – 15**

Aim- Write a Program to implement Binary Search Tree operations.

Code- #include <iostream>  
  
using namespace std;  
  
*// Node structure for Binary Search Tree*struct TreeNode {  
 int data;  
 TreeNode\* left;  
 TreeNode\* right;  
  
 TreeNode(int val) {  
 data = val;  
 left = nullptr;  
 right = nullptr;  
 }  
};  
  
class BinarySearchTree {  
private:  
 TreeNode\* root;  
  
 *// Private helper functions for recursive implementation* TreeNode\* insertRecursive(TreeNode\* node, int data) {  
 if (node == nullptr) {  
 return new TreeNode(data);  
 }  
 if (data < node->data) {  
 node->left = insertRecursive(node->left, data);  
 } else if (data > node->data) {  
 node->right = insertRecursive(node->right, data);  
 }  
 return node;  
 }  
  
 TreeNode\* findMinNode(TreeNode\* node) {  
 while (node->left != nullptr) {  
 node = node->left;  
 }  
 return node;  
 }  
  
 TreeNode\* deleteNodeRecursive(TreeNode\* node, int data) {  
 if (node == nullptr) {  
 return nullptr;  
 }  
 if (data < node->data) {  
 node->left = deleteNodeRecursive(node->left, data);  
 } else if (data > node->data) {  
 node->right = deleteNodeRecursive(node->right, data);  
 } else {  
 *// Case 1: Node with no child or only one child* if (node->left == nullptr) {  
 TreeNode\* temp = node->right;  
 delete node;  
 return temp;  
 } else if (node->right == nullptr) {  
 TreeNode\* temp = node->left;  
 delete node;  
 return temp;  
 }  
 *// Case 2: Node with two children* TreeNode\* temp = findMinNode(node->right);  
 node->data = temp->data;  
 node->right = deleteNodeRecursive(node->right, temp->data);  
 }  
 return node;  
 }  
  
 bool searchRecursive(TreeNode\* node, int data) {  
 if (node == nullptr) {  
 return false;  
 }  
 if (data < node->data) {  
 return searchRecursive(node->left, data);  
 } else if (data > node->data) {  
 return searchRecursive(node->right, data);  
 } else {  
 return true;  
 }  
 }  
  
 void inorderRecursive(TreeNode\* node) {  
 if (node != nullptr) {  
 inorderRecursive(node->left);  
 cout << node->data << " ";  
 inorderRecursive(node->right);  
 }  
 }  
  
 void preorderRecursive(TreeNode\* node) {  
 if (node != nullptr) {  
 cout << node->data << " ";  
 preorderRecursive(node->left);  
 preorderRecursive(node->right);  
 }  
 }  
  
 void postorderRecursive(TreeNode\* node) {  
 if (node != nullptr) {  
 postorderRecursive(node->left);  
 postorderRecursive(node->right);  
 cout << node->data << " ";  
 }  
 }  
  
public:  
 BinarySearchTree() {  
 root = nullptr;  
 }  
  
 *// Function to insert a value into the BST* void insert(int data) {  
 root = insertRecursive(root, data);  
 }  
  
 *// Function to delete a value from the BST* void remove(int data) {  
 root = deleteNodeRecursive(root, data);  
 }  
  
 *// Function to search for a value in the BST* bool search(int data) {  
 return searchRecursive(root, data);  
 }  
  
 *// Function to perform inorder traversal of the BST* void inorderTraversal() {  
 inorderRecursive(root);  
 cout << endl;  
 }  
  
 *// Function to perform preorder traversal of the BST* void preorderTraversal() {  
 preorderRecursive(root);  
 cout << endl;  
 }  
  
 *// Function to perform postorder traversal of the BST* void postorderTraversal() {  
 postorderRecursive(root);  
 cout << endl;  
 }  
  
 *// Function to find the minimum value in the BST* int findMin() {  
 if (root == nullptr) {  
 cout << "Tree is empty\n";  
 return -1;  
 }  
 TreeNode\* minNode = findMinNode(root);  
 return minNode->data;  
 }  
  
 *// Function to find the maximum value in the BST* int findMax() {  
 if (root == nullptr) {  
 cout << "Tree is empty\n";  
 return -1;  
 }  
 TreeNode\* maxNode = root;  
 while (maxNode->right != nullptr) {  
 maxNode = maxNode->right;  
 }  
 return maxNode->data;  
 }  
};  
  
int main() {  
 BinarySearchTree bst;  
  
 bst.insert(50);  
 bst.insert(30);  
 bst.insert(70);  
 bst.insert(20);  
 bst.insert(40);  
 bst.insert(60);  
 bst.insert(80);  
  
 cout << "Inorder traversal: ";  
 bst.inorderTraversal(); *// Output: 20 30 40 50 60 70 80* cout << "Preorder traversal: ";  
 bst.preorderTraversal(); *// Output: 50 30 20 40 70 60 80* cout << "Postorder traversal: ";  
 bst.postorderTraversal(); *// Output: 20 40 30 60 80 70 50* cout << "Minimum element: " << bst.findMin() << endl; *// Output: Minimum element: 20* cout << "Maximum element: " << bst.findMax() << endl; *// Output: Maximum element: 80* bst.remove(30);  
 cout << "Inorder traversal after deleting 30: ";  
 bst.inorderTraversal(); *// Output: 20 40 50 60 70 80* cout << "Search for 40: " << (bst.search(40) ? "Found" : "Not found") << endl; *// Output: Search for 40: Found* cout << "Search for 100: " << (bst.search(100) ? "Found" : "Not found") << endl; *// Output: Search for 100: Not found* return 0;  
}

Output-



**Lab – 16**

Aim- Write a program to implement Bubble Sort, Selection Sort, Quick Sort, Merge Sort and

Insertion Sort algorithm.

Code- #include <iostream>  
#include <vector>  
  
using namespace std;  
  
*// Function to perform Bubble Sort*void bubbleSort(vector<int>& arr) {  
 int n = arr.size();  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = 0; j < n - i - 1; j++) {  
 if (arr[j] > arr[j + 1]) {  
 swap(arr[j], arr[j + 1]);  
 }  
 }  
 }  
}  
  
*// Function to perform Selection Sort*void selectionSort(vector<int>& arr) {  
 int n = arr.size();  
 for (int i = 0; i < n - 1; i++) {  
 int minIndex = i;  
 for (int j = i + 1; j < n; j++) {  
 if (arr[j] < arr[minIndex]) {  
 minIndex = j;  
 }  
 }  
 if (minIndex != i) {  
 swap(arr[i], arr[minIndex]);  
 }  
 }  
}  
  
*// Function to perform Quick Sort*void quickSort(vector<int>& arr, int low, int high) {  
 if (low < high) {  
 int pivot = arr[high];  
 int i = low - 1;  
 for (int j = low; j <= high - 1; j++) {  
 if (arr[j] < pivot) {  
 i++;  
 swap(arr[i], arr[j]);  
 }  
 }  
 swap(arr[i + 1], arr[high]);  
 int partitionIndex = i + 1;  
 quickSort(arr, low, partitionIndex - 1);  
 quickSort(arr, partitionIndex + 1, high);  
 }  
}  
  
*// Function to perform Merge Sort*void merge(vector<int>& arr, int left, int mid, int right) {  
 int n1 = mid - left + 1;  
 int n2 = right - mid;  
 vector<int> L(n1), R(n2);  
  
 for (int i = 0; i < n1; i++) {  
 L[i] = arr[left + i];  
 }  
 for (int j = 0; j < n2; j++) {  
 R[j] = arr[mid + 1 + j];  
 }  
  
 int i = 0, j = 0, k = left;  
 while (i < n1 && j < n2) {  
 if (L[i] <= R[j]) {  
 arr[k] = L[i];  
 i++;  
 } else {  
 arr[k] = R[j];  
 j++;  
 }  
 k++;  
 }  
  
 while (i < n1) {  
 arr[k] = L[i];  
 i++;  
 k++;  
 }  
 while (j < n2) {  
 arr[k] = R[j];  
 j++;  
 k++;  
 }  
}  
  
void mergeSort(vector<int>& arr, int left, int right) {  
 if (left < right) {  
 int mid = left + (right - left) / 2;  
 mergeSort(arr, left, mid);  
 mergeSort(arr, mid + 1, right);  
 merge(arr, left, mid, right);  
 }  
}  
  
*// Function to perform Insertion Sort*void insertionSort(vector<int>& arr) {  
 int n = arr.size();  
 for (int i = 1; i < n; i++) {  
 int key = arr[i];  
 int j = i - 1;  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 j--;  
 }  
 arr[j + 1] = key;  
 }  
}  
  
int main() {  
 vector<int> arr = **{**64, 25, 12, 22, 11**}**;  
 cout << "Original array: ";  
 for (int num : arr) {  
 cout << num << " ";  
}  
 cout << endl;  
 *// Bubble Sort* vector<int> arrBubble = arr;  
 bubbleSort(arrBubble);  
 cout << "Sorted array using Bubble Sort: ";  
 for (int num : arrBubble) {  
 cout << num << " ";}  
 cout << endl;  
 *// Selection Sort* vector<int> arrSelection = arr;  
 selectionSort(arrSelection);  
 cout << "Sorted array using Selection Sort: ";  
 for (int num : arrSelection) {  
 cout << num << " ";}  
 cout << endl;  
 *// Quick Sort* vector<int> arrQuick = arr;  
 quickSort(arrQuick, 0, arrQuick.size() - 1);  
 cout << "Sorted array using Quick Sort: ";  
 for (int num : arrQuick) {  
 cout << num << " ";}  
 cout << endl;  
 *// Merge Sort* vector<int> arrMerge = arr;  
 mergeSort(arrMerge, 0, arrMerge.size() - 1);  
 cout << "Sorted array using Merge Sort: ";  
 for (int num : arrMerge) {  
 cout << num << " ";}  
 cout << endl;  
 *// Insertion Sort* vector<int> arrInsertion = arr;  
 insertionSort(arrInsertion);  
 cout << "Sorted array using Insertion Sort: ";  
 for (int num : arrInsertion) {  
 cout << num << " ";}  
 cout << endl;  
 return 0;}

Output-

