K.R. MANGALAM UNIVERSITY

THE COMPLETE WORLD OF EDUCATION

NAME : VANSH GOEL
COURSE : B.TECH CSE (FSD)
ROLL NO : 2401350002
CourseName : Data Structure
Faculty : Dr.Vandna Batra
LAB FILE

Q1.Given an array of integers, perform the following operations:

- Traversing
- Insertion
- Deletion

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

#define MAX 100

// Function to traverse the array
void traverse(int arr[], int n) {
    cout << "Array elements: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Function to insert element at a given position
int insertElement(int arr[], int n, int pos, int value) {
    if (n >= MAX) {
        cout << "Array is full! Cannot insert." << endl;
        return n;
    }

    // Shift elements to the right
    for (int i = n; i > pos; i--) {
        arr[i] = arr[i - 1];
    }

    arr[pos] = value;
    return n + 1;
}

// Function to delete element at a given position
int deleteElement(int arr[], int n, int pos) {
    if (pos < 0 || pos >= n) {
        cout << "Invalid position! Cannot delete." << endl;
        return n;
    }
```

```
Click to add a breakpoint
38      // Shift elements to the left
39      for (int i = pos; i < n - 1; i++) {
40          arr[i] = arr[i + 1];
41      }
42
43      return n - 1;
44  }
45
46  int main() {
47      int arr[MAX] = {10, 20, 30, 40, 50};
48      int n = 5;
49
50      cout << "Initial array:\n";
51      traverse(arr, n);
52
53      // Insertion
54      n = insertElement(arr, n, 2, 99); // Insert 99 at index 2
55      cout << "\nAfter insertion (99 at index 2):\n";
56      traverse(arr, n);
57
58      // Deletion
59      n = deleteElement(arr, n, 3); // Delete element at index 3
60      cout << "\nAfter deletion (element at index 3):\n";
61      traverse(arr, n);
62
63      return 0;
64  }
65
```

OUTPUT

```
Initial array:
Array elements: 10 20 30 40 50

After insertion (99 at index 2):
Array elements: 10 20 99 30 40 50

After deletion (element at index 3):
Array elements: 10 20 99 40 50
```

Q2. Write a program to implement a singly linked list with methods to insert an element at the head, insert an element at the tail, delete an element
by value, and traverse the list to print all elements.

CODE-

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

// Node structure
struct Node {
    int data;
    Node* next;

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

// Singly Linked List class
class LinkedList {
private:
    Node* head;

public:
    LinkedList() {
        head = nullptr;
    }

    // Insert at head
    void insertAtHead(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
        cout << value << " inserted at head.\n";
    }

    // Insert at tail
    void insertAtTail(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newNode;
        }

        cout << value << " inserted at tail.\n";
    }

    // Delete a node by value
    void deleteByValue(int value) {
        if (head == nullptr) {
            cout << "List is empty! Cannot delete.\n";
            return;
        }

        // If value is at head
        if (head->data == value) {
            Node* temp = head;
            head = head->next;
            delete temp;
            cout << value << " deleted from list.\n";
            return;
        }

        // Search for the value
        Node* curr = head;
        while (curr->next != nullptr && curr->next->data != value) {
            curr = curr->next;
        }
```

```
        if (curr->next == nullptr) {
            cout << "Value " << value << " not found in list.\n";
        } else {
            Node* temp = curr->next;
            curr->next = curr->next->next;
            delete temp;
            cout << value << " deleted from list.\n";
        }
    }

    // Traverse list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty.\n";
            return;
        }

        Node* temp = head;
        cout << "List elements: ";
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;

    list.insertAtHead(10);
    list.insertAtTail(20);
```

```
        Node* temp = head;
        cout << "List elements: ";
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;

    list.insertAtHead(10);
    list.insertAtTail(20);
    list.insertAtTail(30);
    list.insertAtHead(5);

    cout << endl;
    list.traverse();

    cout << endl;
    list.deleteByValue(20);
    list.traverse();

    return 0;
}
```

OUTPUT-

```
10 inserted at head.
20 inserted at tail.
30 inserted at tail.
5 inserted at head.

List elements: 5 10 20 30

20 deleted from list.
List elements: 5 10 30
```

Q3 Write a class to implement a circular linked list with methods to insert an element at the head, insert an element at the tail, delete an element
by value, and traverse the list to print all elements

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

// Node structure
struct Node {
    int data;
    Node* next;

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

// Circular Linked List class
class CircularLinkedList {
private:
    Node* head;

public:
    CircularLinkedList() {
        head = nullptr;
    }

    // Insert at head
    void insertAtHead(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            newNode->next = head;  // Points to itself
        } else {
            Node* temp = head;
            while (temp->next != head) {  // Reach the last node
                temp = temp->next;
            }
```

```cpp
            newNode->next = head;
            temp->next = newNode;
            head = newNode;
        }

        cout << value << " inserted at head.\n";
    }

    // Insert at tail
    void insertAtTail(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            newNode->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {  // Reach the last node
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }

        cout << value << " inserted at tail.\n";
    }

    // Delete by value
    void deleteByValue(int value) {
        if (head == nullptr) {
            cout << "List is empty! Cannot delete.\n";
            return;
        }
```

```
70
71          Node* curr = head;
72          Node* prev = nullptr;
73
74          // Case 1: Deleting head node
75          if (head->data == value) {
76              // If only one node exists
77              if (head->next == head) {
78                  delete head;
79                  head = nullptr;
80                  cout << value << " deleted from list.\n";
81                  return;
82              }
83
84              // Find last node to fix circular link
85              Node* temp = head;
86              while (temp->next != head) {
87                  temp = temp->next;
88              }
89
90              temp->next = head->next;  // Last node points to second node
91              Node* toDelete = head;
92              head = head->next;
93              delete toDelete;
94
95              cout << value << " deleted from list.\n";
96              return;
97          }
98
99          // Case 2: Deleting any other node
100         do {
101             prev = curr;
102             curr = curr->next;
103
103
104                 if (curr->data == value) {
105                     prev->next = curr->next;
106                     delete curr;
107                     cout << value << " deleted from list.\n";
108                     return;
109                 }
110
111             } while (curr != head);
112
113             cout << "Value " << value << " not found in list.\n";
114         }
115
116         // Traverse the list
117         void traverse() {
118             if (head == nullptr) {
119                 cout << "List is empty.\n";
120                 return;
121             }
122
123             Node* temp = head;
124             cout << "Circular List elements: ";
125
126             do {
127                 cout << temp->data << " ";
128                 temp = temp->next;
129             } while (temp != head);
130
131             cout << endl;
132         }
133     };
134
135     int main() {
```

```
136     CircularLinkedList clist;
137
138     clist.insertAtHead(10);
139     clist.insertAtTail(20);
140     clist.insertAtTail(30);
141     clist.insertAtHead(5);
142
143     cout << endl;
144     clist.traverse();
145
146     cout << endl;
147     clist.deleteByValue(20);
148     clist.traverse();
149
150     return 0;
151 }
152
```

OUTPUT-

```
10 inserted at head.
20 inserted at tail.
30 inserted at tail.
5 inserted at head.

Circular List elements: 5 10 20 30

20 deleted from list.
Circular List elements: 5 10 30
```

Q4. Implement a doubly linked list with methods to insert an element at the head, insert an element at the tail, delete an element by value, and
reverse the list. Ensure that all operations handle edge cases appropriately

```
        }

        cout << value << " inserted at head.\n";
    }

    // Insert at the tail
    void insertAtTail(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            cout << value << " inserted at tail.\n";
            return;
        }

        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }

        temp->next = newNode;
        newNode->prev = temp;

        cout << value << " inserted at tail.\n";
    }

    // Delete a node by its value
    void deleteByValue(int value) {
        if (head == nullptr) {
            cout << "List is empty! Cannot delete.\n";
            return;
        }

#include <iostream>
using namespace std;

// Node structure for doubly linked list
struct Node {
    int data;
    Node* next;
    Node* prev;

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

class DoublyLinkedList {
private:
    Node* head;

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

    // Insert at the head
    void insertAtHead(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
```

```
        Node* temp = head;

        // Case 1: Value at head
        if (temp->data == value) {
            if (temp->next == nullptr) {
                delete temp;
                head = nullptr;  // list becomes empty
            } else {
                head = temp->next;
                head->prev = nullptr;
                delete temp;
            }

            cout << value << " deleted from list.\n";
            return;
        }

        // Search for the value
        while (temp != nullptr && temp->data != value) {
            temp = temp->next;
        }

        if (temp == nullptr) {
            cout << "Value " << value << " not found.\n";
            return;
        }

        // Case 2: Node is in middle or tail
        if (temp->next != nullptr)
            temp->next->prev = temp->prev;

        if (temp->prev != nullptr)
            temp->prev->next = temp->next;
    delete temp;
    cout << value << " deleted from list.\n";
}

// Reverse the doubly linked list
void reverse() {
    if (head == nullptr || head->next == nullptr) {
        cout << "List reversed (no change needed).\n";
        return;
    }

    Node* curr = head;
    Node* temp = nullptr;

    // Swap next and prev for all nodes
    while (curr != nullptr) {
        temp = curr->prev;
        curr->prev = curr->next;
        curr->next = temp;

        curr = curr->prev;  // Move to next node (which is prev now)
    }

    // Reset head to the last node processed
    if (temp != nullptr) {
        head = temp->prev;
    }

    cout << "List reversed.\n";
}

// Traverse and print
void traverse() {
```

```
136         void traverse() {
137             if (head == nullptr) {
138                 cout << "List is empty.\n";
139                 return;
140             }
141 
142             Node* temp = head;
143             cout << "List elements: ";
144             while (temp != nullptr) {
145                 cout << temp->data << " ";
146                 temp = temp->next;
147             }
148             cout << endl;
149         }
150     };
151 
152     int main() {
153         DoublyLinkedList list;
154 
155         list.insertAtHead(10);
156         list.insertAtTail(20);
157         list.insertAtTail(30);
158         list.insertAtHead(5);
159 
160         cout << endl;
161         list.traverse();
162 
163         list.deleteByValue(20);
164         list.traverse();
165 
166         list.reverse();
167         list.traverse();
168 
169         return 0;
170     }
```

OUTPUT-

```
10 inserted at head.
20 inserted at tail.
30 inserted at tail.
5 inserted at head.

List elements: 5 10 20 30
20 deleted from list.
List elements: 5 10 30
List reversed.
List elements: 30 10 5
```

Q5. Implement a stack using arrays with methods for push, pop, and peek operations.

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

#define MAX 100   // Maximum size of stack

class Stack {
private:
    int arr[MAX];
    int top;

public:
    Stack() {
        top = -1;   // Stack initially empty
    }

    // Push operation
    void push(int value) {
        if (top == MAX - 1) {
            cout << "Stack Overflow! Cannot push " << value << endl;
            return;
        }
        arr[++top] = value;
        cout << value << " pushed onto stack.\n";
    }

    // Pop operation
    int pop() {
        if (top == -1) {
            cout << "Stack Underflow! Cannot pop.\n";
            return -1;  // Error value
        }
        cout << arr[top] << " popped from stack.\n";
        return arr[top--];
    }

    // Peek operation
    int peek() {
        if (top == -1) {
            cout << "Stack is empty. Nothing to peek.\n";
            return -1;
        }
        cout << "Top element: " << arr[top] << endl;
        return arr[top];
    }

    // Display full stack (optional)
    void display() {
        if (top == -1) {
            cout << "Stack is empty.\n";
            return;
        }
        cout << "Stack elements: ";
        for (int i = top; i >= 0; i--) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Stack st;

    st.push(10);
    st.push(20);
    st.push(30);

    st.peek();
    st.display();

    st.pop();
    st.pop();
```

```
67      st.peek();
68      st.display();
69
70      st.pop();
71      st.pop();
72
73      st.display();
74
75      return 0;
76  }
77
```

OUTPUT-

```
10 pushed onto stack.
20 pushed onto stack.
30 pushed onto stack.
Top element: 30
Stack elements: 30 20 10
30 popped from stack.
20 popped from stack.
Stack elements: 10
```

Q6. Write a function to convert an infix expression to a postfix expression using a stack. The function should handle parentheses and operator precedence correctly

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

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

// Associativity: ^ is right-associative
bool isRightAssociative(char op) {
    return op == '^';
}

// Check if character is an operator
bool isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}

// Convert infix to postfix
string infixToPostfix(string infix) {
    stack<char> st;
    string postfix = "";

    for (char ch : infix) {

        // 1. If operand, add to output
        if (isalnum(ch)) {
            postfix += ch;
        }
        // 2. If '(', push to stack
        else if (ch == '(') {
            st.push(ch);
        }
        // 3. If ')', pop until '('
        else if (ch == ')') {
            while (!st.empty() && st.top() != '(') {
                postfix += st.top();
                st.pop();
            }
            st.pop();  // remove '('
        }
        // 4. Operator encountered
        else if (isOperator(ch)) {
            while (!st.empty() && isOperator(st.top())) {
                if ((precedence(ch) < precedence(st.top())) ||
                    (precedence(ch) == precedence(st.top()) && !isRightAssociative(ch))) {
                    postfix += st.top();
                    st.pop();
                } else break;
            }
            st.push(ch);
        }
    }

    // Pop remaining operators
    while (!st.empty()) {
        postfix += st.top();
        st.pop();
    }

    return postfix;
}

int main() {
```

```
    string infix;
    cout << "Enter infix expression: ";
    cin >> infix;

    string postfix = infixToPostfix(infix);
    cout << "Postfix expression: " << postfix << endl;

    return 0;
}

```

OUTPUT-

```
Enter infix expression: A*(B+C)/D
Postfix expression: ABC+*D/
```

Q7 Create a linear queue using an array with methods for enqueue, dequeue, and checking if the queue is empty or full.

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

#define MAX 100   // Maximum queue size

class Queue {
private:
    int arr[MAX];
    int front, rear;

public:
    Queue() {
        front = -1;
        rear = -1;
    }

    // Check if queue is empty
    bool isEmpty() {
        return (front == -1);
    }

    // Check if queue is full
    bool isFull() {
        return (rear == MAX - 1);
    }

    // Enqueue operation
    void enqueue(int value) {
        if (isFull()) {
            cout << "Queue Overflow! Cannot enqueue " << value << endl;
            return;
        }

        if (isEmpty()) {
            front = 0;  // First element
        }

        arr[++rear] = value;
        cout << value << " enqueued.\n";
    }

    // Dequeue operation
    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow! Cannot dequeue.\n";
            return -1;
        }

        int removed = arr[front];

        if (front == rear) {
            // Queue becomes empty
            front = rear = -1;
        } else {
            front++;
        }

        cout << removed << " dequeued.\n";
        return removed;
    }

    // Display queue
    void display() {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return;
        }

        cout << "Queue elements: ";
        for (int i = front; i <= rear; i++) {
```

```
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Queue q;

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    q.display();

    q.dequeue();
    q.display();

    q.dequeue();
    q.dequeue();

    q.dequeue();  // Underflow example

    return 0;
}
```

OUTPUT-

```
10 enqueued.
20 enqueued.
30 enqueued.
Queue elements: 10 20 30
10 dequeued.
Queue elements: 20 30
20 dequeued.
30 dequeued.
Queue Underflow! Cannot dequeue.
```

Q8 Create a circular queue using array with methods for enqueue, dequeue, and checking if the queue is empty or full. Ensure that the circular
nature of the queue is maintained after each operation

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

#define MAX 5   // Maximum size of the circular queue

class CircularQueue {
private:
    int arr[MAX];
    int front, rear;

public:
    CircularQueue() {
        front = -1;
        rear = -1;
    }

    // Check if queue is full
    bool isFull() {
        return ((rear + 1) % MAX == front);
    }

    // Check if queue is empty
    bool isEmpty() {
        return (front == -1);
    }

    // Enqueue operation
    void enqueue(int value) {
        if (isFull()) {
            cout << "Queue Overflow! Cannot enqueue " << value << endl;
            return;
        }

        if (isEmpty()) {
            front = rear = 0;
        } else {
```

```cpp
            rear = (rear + 1) % MAX;
        }

        arr[rear] = value;
        cout << value << " enqueued.\n";
    }

    // Dequeue operation
    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow! Cannot dequeue.\n";
            return -1;
        }

        int removed = arr[front];

        if (front == rear) {
            // Queue becomes empty
            front = rear = -1;
        } else {
            front = (front + 1) % MAX;
        }

        cout << removed << " dequeued.\n";
        return removed;
    }

    // Display queue
    void display() {
        if (isEmpty()) {
            cout << "Queue is empty.\n";
            return;
        }
```

```
71              cout << "Queue elements: ";
72              int i = front;
73              while (true) {
74                  cout << arr[i] << " ";
75                  if (i == rear) break;
76                  i = (i + 1) % MAX;
77              }
78              cout << endl;
79          }
80      };
81
82      int main() {
83          CircularQueue q;
84
85          q.enqueue(10);
86          q.enqueue(20);
87          q.enqueue(30);
88          q.enqueue(40);
89
90          q.display();
91
92          q.dequeue();
93          q.dequeue();
94
95          q.display();
96
97          q.enqueue(50);
98          q.enqueue(60);  // Should wrap around
99
100         q.display();
101
102         return 0;
103
104
```

OUTPUT-

```
10 enqueued.
20 enqueued.
30 enqueued.
40 enqueued.
Queue elements: 10 20 30 40
10 dequeued.
20 dequeued.
Queue elements: 30 40
50 enqueued.
60 enqueued.
Queue elements: 30 40 50 60
```

Q9.Implement linear search

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

int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;   // return index where target is found
        }
    }
    return -1;  // not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);

    int target;
    cout << "Enter element to search: ";
    cin >> target;

    int index = linearSearch(arr, n, target);

    if (index != -1) {
        cout << "Element found at index " << index << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```

OUTPUT-

```
Enter element to search: 30
Element found at index 2
```

Q10 Implement binary search(iterative and recursive)

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

// Iterative Binary Search
int binarySearchIterative(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;   // Element found
        else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }

    return -1;  // Element not found
}

// Recursive Binary Search
int binarySearchRecursive(int arr[], int left, int right, int target) {
    if (left > right) return -1;  // Base case: not found

    int mid = left + (right - left) / 2;

    if (arr[mid] == target)
        return mid;
    else if (arr[mid] < target)
        return binarySearchRecursive(arr, mid + 1, right, target);
    else
        return binarySearchRecursive(arr, left, mid - 1, target);
}

```

```cpp
int main() {
    int arr[] = {5, 10, 15, 20, 25, 30};
    int n = sizeof(arr) / sizeof(arr[0]);

    int target;
    cout << "Enter element to search: ";
    cin >> target;

    // Iterative
    int indexIter = binarySearchIterative(arr, n, target);
    if (indexIter != -1)
        cout << "Element found at index " << indexIter << " (Iterative)" << endl;
    else
        cout << "Element not found (Iterative)" << endl;

    // Recursive
    int indexRec = binarySearchRecursive(arr, 0, n - 1, target);
    if (indexRec != -1)
        cout << "Element found at index " << indexRec << " (Recursive)" << endl;
    else
        cout << "Element not found (Recursive)" << endl;

    return 0;
}
```

OUTPUT-

```
Enter element to search: 25
Element found at index 4 (Iterative)
Element found at index 4 (Recursive)
```

Q11 Implement various sorting algorithms including Insertion sort, selection sort, bubble sort, and analyze their performance on different input sizes.

```cpp
#include <iostream>
#include <vector>
#include <ctime>    // For measuring time
using namespace std;

// Swap utility
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// 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;
    }
}

// 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;
        }
        swap(arr[i], arr[minIndex]);
    }
}

// Bubble Sort
void bubbleSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break; // Optimization: stop if already sorted
    }
}

// Print array
void printArray(const vector<int> &arr) {
    for (int x : arr) cout << x << " ";
    cout << endl;
}

// Generate random array
vector<int> generateRandomArray(int size, int maxValue=1000) {
    vector<int> arr(size);
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % maxValue;
    }
    return arr;
}

int main() {
```

```
    int size;
    cout << "Enter size of array: ";
    cin >> size;

    vector<int> arr = generateRandomArray(size);

    cout << "\nOriginal Array (first 20 elements only for large arrays): ";
    for (int i = 0; i < min(size, 20); i++) cout << arr[i] << " ";
    cout << endl;

    // Insertion Sort
    vector<int> arr1 = arr;
    clock_t start = clock();
    insertionSort(arr1);
    clock_t end = clock();
    cout << "Insertion Sort Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;

    // Selection Sort
    vector<int> arr2 = arr;
    start = clock();
    selectionSort(arr2);
    end = clock();
    cout << "Selection Sort Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;

    // Bubble Sort
    vector<int> arr3 = arr;
    start = clock();
    bubbleSort(arr3);
    end = clock();
    cout << "Bubble Sort Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;

    cout << "\nSorted Array (first 20 elements): ";
    for (int i = 0; i < min(size, 20); i++) cout << arr1[i] << " ";
    cout << endl;
```

OUTPUT-

```
Enter size of array: 10

Original Array (first 20 elements only for large arrays): 113 846 618 568 368 258 950
    869 567 717
Insertion Sort Time: 3e-06 seconds
Selection Sort Time: 2e-06 seconds
Bubble Sort Time: 2e-06 seconds

Sorted Array (first 20 elements): 113 258 368 567 568 618 717 846 869 950
```

Q12 Implement various sorting algorithms including Quick Sort, Merge Sort, Heap Sort, and analyze their performance on different input sizes Ensure the implementation handles edge cases such as duplicate values and nearly sorted
arrays.

```cpp
#include <iostream>
#include <vector>
#include <ctime>
#include <algorithm> // For std::is_sorted
using namespace std;

// -------------------- Quick Sort --------------------
int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[high]; // choose last element as pivot
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) { // <= handles duplicates
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// -------------------- 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 i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];
```

```cpp
    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(vector<int> &arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

// -------------------- Heap Sort --------------------
void heapify(vector<int> &arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

```cpp
void heapSort(vector<int> &arr) {
    int n = arr.size();
    // Build heap
    for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
    // Extract elements
    for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// -------------------- Utility Functions --------------------
vector<int> generateRandomArray(int size, int maxValue = 1000) {
    vector<int> arr(size);
    for (int i = 0; i < size; i++) arr[i] = rand() % maxValue;
    return arr;
}

void printArray(const vector<int> &arr, int limit = 20) {
    for (int i = 0; i < min((int)arr.size(), limit); i++) cout << arr[i] << " ";
    if (arr.size() > limit) cout << "...";
    cout << endl;
}

// -------------------- Main Function --------------------
int main() {
    srand(time(0));

    int size;
    cout << "Enter size of array: ";
    cin >> size;

    vector<int> originalArray = generateRandomArray(size);

    cout << "\nOriginal Array (first 20 elements): ";
    printArray(originalArray);

    // ---------------- Quick Sort ----------------
    vector<int> arrQS = originalArray;
    clock_t start = clock();
    quickSort(arrQS, 0, arrQS.size() - 1);
    clock_t end = clock();
    cout << "Quick Sort Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds\n";

    // ---------------- Merge Sort ----------------
    vector<int> arrMS = originalArray;
    start = clock();
    mergeSort(arrMS, 0, arrMS.size() - 1);
    end = clock();
    cout << "Merge Sort Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds\n";

    // ---------------- Heap Sort ----------------
    vector<int> arrHS = originalArray;
    start = clock();
    heapSort(arrHS);
    end = clock();
    cout << "Heap Sort Time: " << double(end - start) / CLOCKS_PER_SEC << " seconds\n";

    cout << "\nSorted Array (first 20 elements): ";
    printArray(arrMS);

    // Verify if sorted correctly
    if (is_sorted(arrMS.begin(), arrMS.end())) cout << "Array is sorted correctly.\n";
    else cout << "Sorting error!\n";

    return 0;
}
```

OUTPUT-

```
Enter size of array: 10

Original Array (first 20 elements): 799 542 919 480 547 760 372 147 7 857
Quick Sort Time: 3e-06 seconds
Merge Sort Time: 1.2e-05 seconds
Heap Sort Time: 3e-06 seconds

Sorted Array (first 20 elements): 7 147 372 480 542 547 760 799 857 919
Array is sorted correctly.
```

QUESTION 13 : Given preorder and inorder traversal of a tree, construct the binary tree.

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

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

    Node(int val) {
        data = val;
        left = right = nullptr;
    }
};

// Function to build tree recursively
Node* buildTreeHelper(int preorder[], int inorder[], int inStart, int inEnd,
                      int &preIndex, unordered_map<int, int> &inMap) {
    if (inStart > inEnd) return nullptr;

    int rootVal = preorder[preIndex++];
    Node* root = new Node(rootVal);

    // If no children
    if (inStart == inEnd) return root;

    int inIndex = inMap[rootVal];

    root->left = buildTreeHelper(preorder, inorder, inStart, inIndex - 1, preIndex, inMap);
    root->right = buildTreeHelper(preorder, inorder, inIndex + 1, inEnd, preIndex, inMap);

    return root;
}

// Main function to build tree
```

```cpp
Node* buildTree(int preorder[], int inorder[], int n) {
    unordered_map<int, int> inMap; // value -> index
    for (int i = 0; i < n; i++) {
        inMap[inorder[i]] = i;
    }

    int preIndex = 0;
    return buildTreeHelper(preorder, inorder, 0, n - 1, preIndex, inMap);
}

// Function to print inorder traversal
void printInorder(Node* root) {
    if (!root) return;
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

// Function to print preorder traversal
void printPreorder(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}

int main() {
    int preorder[] = {3, 9, 20, 15, 7};
    int inorder[] = {9, 3, 15, 20, 7};
    int n = sizeof(preorder) / sizeof(preorder[0]);

    Node* root = buildTree(preorder, inorder, n);

    cout << "Inorder of constructed tree: ";
    printInorder(root);
    cout << endl;
```

```
69
70      cout << "Inorder of constructed tree: ";
71      printInorder(root);
72      cout << endl;
73
74      cout << "Preorder of constructed tree: ";
75      printPreorder(root);
76      cout << endl;
77
78      return 0;
79  }
80
```

OUTPUT-

```
Inorder of constructed tree: 9 3 15 20 7
Preorder of constructed tree: 3 9 20 15 7
```

Q14 Perform the traversal of graph(DFS,BFS)

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

// Graph class using adjacency list
class Graph {
private:
    int V; // Number of vertices
    vector<vector<int>> adj;

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

    // Add edge (undirected)
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u); // Remove this line for directed graph
    }

    // DFS utility function
    void DFSUtil(int v, vector<bool> &visited) {
        visited[v] = true;
        cout << v << " ";

        for (int u : adj[v]) {
            if (!visited[u])
                DFSUtil(u, visited);
        }
    }

    // DFS traversal
    void DFS(int start) {
        vector<bool> visited(V, false);
        cout << "DFS Traversal starting from vertex " << start << ": ";
        DFSUtil(start, visited);
        cout << endl;
    }

    // BFS traversal
    void BFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;

        visited[start] = true;
        q.push(start);

        cout << "BFS Traversal starting from vertex " << start << ": ";

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

            for (int u : adj[v]) {
                if (!visited[u]) {
                    visited[u] = true;
                    q.push(u);
                }
            }
        }
        cout << endl;
    }
};

int main() {
    int V = 6;
```

```
70      int V = 6;
71      Graph g(V);
72
73      // Adding edges
74      g.addEdge(0, 1);
75      g.addEdge(0, 2);
76      g.addEdge(1, 3);
77      g.addEdge(1, 4);
78      g.addEdge(2, 4);
79      g.addEdge(3, 5);
80      g.addEdge(4, 5);
81
82      g.DFS(0);
83      g.BFS(0);
84
85      return 0;
86  }
87
```

OUTPUT-

```
DFS Traversal starting from vertex 0: 0 1 3 5 4 2
BFS Traversal starting from vertex 0: 0 1 2 3 4 5
```

Q15 Implement Prim's and Kruskal's algorithm to find the minimum spanning tree of a graph

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

// -------------------- Prim's Algorithm --------------------
void primMST(const vector<vector<int>> &graph, int V) {
    vector<int> key(V, INT_MAX);      // Minimum edge weight to include vertex
    vector<int> parent(V, -1);        // Store MST
    vector<bool> inMST(V, false);     // Whether vertex is included

    key[0] = 0;  // Start from vertex 0

    for (int count = 0; count < V - 1; count++) {
        // Find vertex with minimum key not in MST
        int u = -1;
        int minKey = INT_MAX;
        for (int v = 0; v < V; v++) {
            if (!inMST[v] && key[v] < minKey) {
                minKey = key[v];
                u = v;
            }
        }

        inMST[u] = true;

        // Update keys of adjacent vertices
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !inMST[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
                parent[v] = u;
            }
        }
    }

    // Print MST
    cout << "Prim's MST Edges:\n";
    int totalWeight = 0;
    for (int i = 1; i < V; i++) {
        cout << parent[i] << " - " << i << " weight: " << graph[i][parent[i]] << endl;
        totalWeight += graph[i][parent[i]];
    }
    cout << "Total weight of MST: " << totalWeight << endl;
}

// -------------------- Kruskal's Algorithm --------------------
struct Edge {
    int u, v, weight;
};

bool compareEdge(const Edge &a, const Edge &b) {
    return a.weight < b.weight;
}

// Disjoint Set Union (Union-Find)
struct DSU {
    vector<int> parent, rank;
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) parent[i] = i;
    }
    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }
    void unite(int x, int y) {
        int xr = find(x);
        int yr = find(y);
        if (xr == yr) return;
```

```
        if (rank[xr] < rank[yr]) parent[xr] = yr;
        else if (rank[xr] > rank[yr]) parent[yr] = xr;
        else {
            parent[yr] = xr;
            rank[xr]++;
        }
    }
};

void kruskalMST(int V, vector<Edge> &edges) {
    sort(edges.begin(), edges.end(), compareEdge);
    DSU dsu(V);

    cout << "Kruskal's MST Edges:\n";
    int totalWeight = 0;
    for (Edge e : edges) {
        if (dsu.find(e.u) != dsu.find(e.v)) {
            cout << e.u << " - " << e.v << " weight: " << e.weight << endl;
            totalWeight += e.weight;
            dsu.unite(e.u, e.v);
        }
    }
    cout << "Total weight of MST: " << totalWeight << endl;
}

// -------------------- Main Function --------------------
int main() {
    int V = 5;

    // Adjacency matrix for Prim's algorithm (0 = no edge)
    vector<vector<int>> graph = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
```

```
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    primMST(graph, V);

    // Edges list for Kruskal's algorithm
    vector<Edge> edges = {
        {0, 1, 2}, {0, 3, 6}, {1, 2, 3}, {1, 3, 8}, {1, 4, 5},
        {2, 4, 7}, {3, 4, 9}
    };

    kruskalMST(V, edges);

    return 0;
}
```

OUTPUT-

```
Prim's MST Edges:
0 - 1 weight: 2
1 - 2 weight: 3
0 - 3 weight: 6
1 - 4 weight: 5
Total weight of MST: 16
Kruskal's MST Edges:
0 - 1 weight: 2
1 - 2 weight: 3
1 - 4 weight: 5
0 - 3 weight: 6
Total weight of MST: 16
```

QUESTION 16 : Implement Dijkstra's algorithm to find the shortest path from a source vertex to all other vertices in a weighted graph. Use both adjacency matrix and adjacency list representations for the graph. Ensure the algorithm handles negative weights appropriately.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

// -------------------- Adjacency Matrix Representation --------------------
void dijkstraMatrix(const vector<vector<int>> &graph, int src) {
    int V = graph.size();
    vector<int> dist(V, INT_MAX);
    vector<bool> visited(V, false);

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        // Find the vertex with minimum distance not visited
        int u = -1, minDist = INT_MAX;
        for (int i = 0; i < V; i++) {
            if (!visited[i] && dist[i] < minDist) {
                minDist = dist[i];
                u = i;
            }
        }

        if (u == -1) break; // Remaining vertices not reachable
        visited[u] = true;

        // Update distances of adjacent vertices
        for (int v = 0; v < V; v++) {
            if (graph[u][v] != 0 && dist[u] != INT_MAX &&
                dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    cout << "\nDijkstra (Adjacency Matrix) from vertex " << src << ":\n";
    for (int i = 0; i < V; i++) {
        if (dist[i] == INT_MAX) cout << i << ": INF\n";
        else cout << i << ": " << dist[i] << "\n";
    }
}

// -------------------- Adjacency List Representation --------------------
void dijkstraList(int V, const vector<vector<pair<int,int>>> &adj, int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    // Min-heap: pair(distance, vertex)
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        int d = pq.top().first;
        pq.pop();

        if (d > dist[u]) continue;

        for (auto &edge : adj[u]) {
            int v = edge.first;
            int w = edge.second;

            if (w < 0) {
                cout << "Error: Negative edge detected! Dijkstra cannot handle negative weights.\n
                return;
            }

            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
```

```
            }
        }
    }

    cout << "\nDijkstra (Adjacency List) from vertex " << src << ":\n";
    for (int i = 0; i < V; i++) {
        if (dist[i] == INT_MAX) cout << i << ": INF\n";
        else cout << i << ": " << dist[i] << "\n";
    }
}

// ------------------- Main Function -------------------
int main() {
    int V = 5;
    int src = 0;

    // --------- Adjacency Matrix ---------
    vector<vector<int>> graphMatrix = {
        {0, 10, 0, 5, 0},
        {0, 0, 1, 2, 0},
        {0, 0, 0, 0, 4},
        {0, 3, 9, 0, 2},
        {7, 0, 6, 0, 0}
    };

    dijkstraMatrix(graphMatrix, src);

    // --------- Adjacency List ---------
    vector<vector<pair<int,int>>> graphList(V);
    graphList[0] = {{1,10},{3,5}};
    graphList[1] = {{2,1},{3,2}};
```

```
    graphList[1] = {{2,1},{3,2}};
    graphList[2] = {{4,4}};
    graphList[3] = {{1,3},{2,9},{4,2}};
    graphList[4] = {{0,7},{2,6}};

    dijkstraList(V, graphList, src);

    return 0;
}
```

OUTPUT-

```
Dijkstra (Adjacency Matrix) from vertex 0:
0: 0
1: 8
2: 9
3: 5
4: 7

Dijkstra (Adjacency List) from vertex 0:
0: 0
1: 8
2: 9
3: 5
4: 7
```