SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

FACULTY OF SCIENCE AND HUMANITIES

DEPARTMENT OF COMPUTER APPLICATIONS

PRACTICAL RECORD NOTE

STUDENT NAME :

REGISTER NUMBER :

CLASS : **MSc ADS** **SEC : B**

YEAR & SEMESTER : **I YEAR & I SEM**

SUBJECT CODE : **PAD25C03J**

SUBJECT TITLE : **DATA STRUCTURES AND ALGORITHMS**

OCTOBER - 2025

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
FACULTY OF SCIENCE AND HUMANITIES
DEPARTMENT OF COMPUTER APPLICATIONS
SRM Nagar, Kattankulathur – 603 203

CERTIFICATE

Certified to be the bonafied record of practical work done by

____

*Register No.*__________________________ *of* ***I MSc ADS*** *Degree course for* ***DATA STRUCTURES AND ALGORITHMS (PAD25C03J)*** *in the Computer lab in SRM Institute of Science and Technology during the academic year 2025-2026.*

Staff In-charge ***Head of the Department***

Submitted for Semester Practical Examination held on ______________________

Internal Examiner 1 ***Internal Examiner2***

INDEX

Ex No 1

Date: 14-07-2025

Reading and Printing N Integer Elements of an Array

Aim:

To write a C++ program to read and print an array of N integer elements

Algorithm:

1. Start the program.
2. Declare an integer variable n and an integer array a[20].
3. Input the number of elements n.
4. Repeat steps 5–6 for each i from 0 to n - 1:
   Prompt the user to enter element at position i + 1.
   Store the value in a[i].
5. Display the message “The array elements are:”
6. Repeat for each i from 0 to n - 1:
   Print the element number (i + 1) and its value a[i].
7. Stop the program.

Program:

```
#include <iostream>
using namespace std;
int main()
{
int n,a[20],i;
    cout << "Enter the number of elements: ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
    cout << "Element " << i + 1 << ": ";
    cin >> a[i];
    }
    // Display the array
    cout << "The array elements are:\n";
    for (int i = 0; i < n; i++)
    {
        cout << "Element " << i + 1 << ": " << a[i] << endl;
    }
            return 0;
}
```

Output:

Enter the number of elements: 8
Element 1: 12
Element 2: 45
Element 3: 38
Element 4: 90
Element 5: 29
Element 6: 46
Element 7: 81
Element 8: 63
The array elements are:
Element 1: 12
Element 2: 45
Element 3: 38
Element 4: 90
Element 5: 29
Element 6: 46
Element 7: 81
Element 8: 63

Result:

The above program was successfully compiled and executed.

Ex No 2

Date: 15-07-2025

Array Operations

Aim:

To write a C++ program to implement the following operations

- Input and display array elements
- Insert element at a specific position
- Delete element from a specific position
- Search for an element

Algorithm:

1. Start the program.
2. Define a class `Array` with:
   An integer array `a` and integer variable `size`.
   Constructor to initialize `size` to 0.
   Member functions: `input()`, `display()`, `insert()`, `remove()`, and `search()`.
3. Input the number of elements from the user.
4. Accept array elements from the user using the `input()` function.
5. Display the menu:
   1. Display
   2. Insert
   3. Delete
   4. Search
   5. Exit
6. Repeat the following steps until the user selects Exit:

   Read the user’s choice.
   If choice is 1: Call `display()` to show array elements.
   If choice is 2: Prompt for position and value, then call `insert(pos, value)` to add the value at the given position.
   If choice is 3: Prompt for position, then call `remove(pos)` to delete element from this position.
   If choice is 4: Prompt for value to search, then call `search(value)` to find the element.
   If choice is 5: Exit the menu loop.
   If invalid choice: Display "Invalid choice".

7. End the program.

Program:

```
#include <iostream>
using namespace std;
#define MAX 100  // Maximum size of array
class Array
{
   int a[20];
   int size;
public:
   Array()
           {
      size = 0;
           }
   void input(int n)
            {
      size = n;
      cout << "Enter " << size << " elements:\n";
      for (int i = 0; i < size; i++)
              cin >> a[i];
      }
   void display()
            {
      cout << "Array elements: ";
      for (int i = 0; i < size; i++)
         cout << a[i] << " ";
      cout << endl;
   }
   void insert(int pos, int element)
            {
       for (int i = size; i > pos; i--)
         a[i] = a[i - 1];
      a[pos] = element;
      size++;
      cout << "Element inserted.\n";
   }
   void remove(int pos)
            {
      for (int i = pos; i < size - 1; i++)
         a[i] = a[i + 1];
      size--;
      cout << "Element deleted.\n";
     }
   void search(int element)
           {
      for (int i = 0; i < size; i++)
              {
         if (a[i] == element)
            {
            cout << "Element found at index " << i << endl;
```

```
            return;
          }
        }
        cout << "Element not found.\n";
    }
};
int main()
 {
    Array a;
    int n, choice, pos, val;
    cout << "Enter number of elements: ";
    cin >> n;
    a.input(n);

    do
            {
        cout << "\nMenu:\n";
        cout << "1. Display\n2. Insert\n3. Delete\n4. Search\n5. Exit\n";
        cout << "Enter choice: ";
        cin >> choice;
        switch (choice)
             {
        case 1:
            a.display();
            break;
        case 2:
            cout << "Enter position and value to insert: ";
            cin >> pos >> val;
            a.insert(pos, val);
            break;
        case 3:
            cout << "Enter position to delete: ";
            cin >> pos;
            a.remove(pos);
            break;
        case 4:
            cout << "Enter value to search: ";
            cin >> val;
            a.search(val);
            break;
        case 5:
            cout << "Exiting...\n";
            break;
              default:
            cout << "Invalid choice.\n";
        }
    } while (choice != 5);

    return 0;
}
```

Output:

Enter number of elements: 3
Enter 3 elements:
12
8
17
Menu:
1. Display
2. Insert
3. Delete
4. Search
5. Exit
Enter choice: 1
Array elements: 12 8 17

Menu:
1. Display
2. Insert
3. Delete
4. Search
5. Exit
Enter choice: 2
Enter position and value to insert: 3
13
Element inserted.

Menu:
1. Display
2. Insert
3. Delete
4. Search
5. Exit
Enter choice: 3
Enter position to delete: 2
Element deleted.

Menu:
1. Display
2. Insert
3. Delete
4. Search
5. Exit
Enter choice: 1
Array elements: 12 8 13

Menu:
1. Display
2. Insert
3. Delete
4. Search
5. Exit
Enter choice: 4
Enter value to search: 13
Element found at index 2

Menu:
1. Display
2. Insert
3. Delete
4. Search
5. Exit
Enter choice: 5
Exiting...

Result:

The array implementation was successfully executed and verified.

Ex No 3

Date: 21-07-2025

Insert an Element into a Linked List

Aim:

To write a C++ program to implement insertion operations in a singly linked list such as inserting at the beginning, at the end, and at a specific position.

Algorithm:

Step 1: Start the program.
Step 2: Define a structure `Node` containing:
`data` → to store integer value
`link` → pointer to the next node
Step 3: Define a function `createNode(value)` to:
Dynamically allocate memory for a new node
Assign the given value to `data`
Initialize `link` as `NULL`
Return the new node
Step 4: Define a function `insertAtBeginning(head, value)` to:
Create a new node using `createNode(value)`
Set the new node's `link` to point to the current `head`
Update `head` to point to the new node
Step 5: Define a function `insertAtEnd(head, value)` to:
Create a new node using `createNode(value)`
If `head` is `NULL`, make the new node as `head`
Otherwise, traverse to the last node and link the new node at the end
Step 6: Define a function `insertAtPosition(head, value, position)` to:
If `position` is 1, call `insertAtBeginning()`
Else, traverse to the node just before the given position
Insert the new node by adjusting links properly
If position is invalid, display an error message
Step 7: Define a function `displayList(head)` to:
Traverse from `head` and print each node's data until `NULL` is reached
Step 8: In the `main()` function:

1. Initialize `head = NULL`
2. Display the menu:
   - Insert at Beginning
   - Insert at End
   - Insert at Position
   - Display List
   - Exit
3. Perform the chosen operation using appropriate function calls
4. Repeat until user chooses to exit

Step 9: Stop the program.

Program:

```
#include <iostream>
using namespace std;
// Node structure
struct Node {
    int data;
    Node* link;
};

// Function to create a new node
Node* createNode(int value) {
    Node* newNode = new Node();  // dynamically create node
    newNode->data = value;
    newNode->link = nullptr;
    return newNode;
}

// Function to insert at the beginning
void insertAtBeginning(Node*& head, int value) {
    Node* newNode = createNode(value);
    newNode->link = head;
    head = newNode;
}

// Function to insert at the end
void insertAtEnd(Node*& head, int value) {
    Node* newNode = createNode(value);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->link != nullptr) {
            temp = temp->link;
        }
        temp->link = newNode;
    }
}

// Function to insert at a specific position (1-based index)
void insertAtPosition(Node*& head, int value, int position) {
    if (position <= 0) {
        cout << "Invalid position.\n";
        return;
    }
    if (position == 1) {
        insertAtBeginning(head, value);
        return;
    }
```

```
    Node* newNode = createNode(value);
    Node* temp = head;

    for (int i = 1; temp != nullptr && i < position - 1; i++) {
        temp = temp->link;
    }
    if (temp == nullptr) {
        cout << "Position out of range.\n";
        return;
    }
    newNode->link = temp->link;
    temp->link = newNode;
}

// Function to display the list
void displayList(Node* head) {
    cout << "Linked List: ";
    while (head != nullptr) {
        cout << head->data << " -> ";
        head = head->link;
    }
    cout << "NULL\n";
}

// Main function
int main() {
    Node* head = nullptr;
    int choice, value, position;

    do {
        cout << "\n1. Insert at Beginning\n2. Insert at End\n3. Insert at Position\n4. Display
List\n5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter value to insert at beginning: ";
                cin >> value;
                insertAtBeginning(head, value);
                break;
            case 2:
                cout << "Enter value to insert at end: ";
                cin >> value;
                insertAtEnd(head, value);
                break;
            case 3:
                cout << "Enter value to insert: ";
                cin >> value;
                cout << "Enter position: ";
```

```
            cin >> position;
            insertAtPosition(head, value, position);
            break;
        case 4:
            displayList(head);
            break;
        case 5:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice.\n";
        }
    } while (choice != 5);

    return 0;
}
```

Output:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 1
Enter value to insert at beginning: 16

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 2
Enter value to insert at end: 32

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 3
Enter value to insert: 17
Enter position: 2

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 4
Linked List: 16 -> 17 -> 32 -> NULL

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Display List
5. Exit
Enter your choice: 5
Exiting...

Result:

Thus, the C++ program to implement insertion operations in a singly linked list namely insertion at the beginning, at the end, and at a specific position was successfully executed and the linked list elements were displayed correctly.

Ex.No 4

Date: 22-07-2025

Delete an Element from Singly Linked List

Aim:

To write a C++ program to delete an element from the Linked List

- Delete the first node (head)
- Delete the last node
- Delete a node with a specific value

Algorithm:

Step 1: Start the program.

Step 2: Define a structure `Node` containing:

- `data` → integer to store the value
- `next` → pointer to the next node

Step 3: Define a function `createNode(data)` to:

- Dynamically allocate memory for a new node
- Assign the data value and set `next` as `NULL`
- Return the new node

Step 4: Define `insertAtEnd(head, data)` to:

- Create a new node
- If the list is empty, set `head = newNode`
- Otherwise, traverse to the end and link the new node

Step 5: Define `deleteAtBeginning(head)` to:

- If the list is empty, display "List is empty"
- Else, move `head` to the next node and delete the first node

Step 6: Define `deleteAtEnd(head)` to:

- If the list is empty, display "List is empty"
- If only one node, delete it and set `head = NULL`
- Otherwise, traverse to the second last node, delete the last node, and set its `next` to `NULL`

Step 7: Define `deleteByValue(head, value)` to:

- If the list is empty, display "List is empty"
- If the first node matches the value, call `deleteAtBeginning()`
- Otherwise, traverse until the node before the one to delete is found
- Adjust the link and delete the matched node
- If value not found, display appropriate message

Step 8: Define `displayList(head)` to:

- Traverse from `head` and print each node's data until `NULL`

Step 9: In `main()`:

1. Initialize `head = NULL`
2. Insert a few nodes using `insertAtEnd()`
3. Display the list
4. Perform deletions (beginning, end, specific value)
5. Display the list after each operation

Step 10: Stop the program.

Program:

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

// Node structure for singly linked list
struct Node {
   int data;
   Node* next;
};
// Function to create a new node
Node* createNode(int data) {
   Node* newNode = new Node();
   newNode->data = data;
   newNode->next = nullptr;
   return newNode;
}
// Function to insert a node at the end
void insertAtEnd(Node*& head, int data) {
   Node* newNode = createNode(data);
   if (head == nullptr) {
      head = newNode;
      return;
   }
   Node* temp = head;
   while (temp->next != nullptr)
      temp = temp->next;
   temp->next = newNode;
}

// Function to delete the first node
void deleteAtBeginning(Node*& head) {
   if (head == nullptr) {
      cout << "List is empty.\n";
      return;
   }
   Node* temp = head;
   head = head->next;
   delete temp;
}
// Function to delete the last node
void deleteAtEnd(Node*& head) {
   if (head == nullptr) {
      cout << "List is empty.\n";
      return;
   }
   if (head->next == nullptr) {
      delete head;
      head = nullptr;
      return;
```

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

// Function to delete a node by value
void deleteByValue(Node*& head, int value) {
    if (head == nullptr) {
        cout << "List is empty.\n";
        return;
    }
    if (head->data == value) {
        deleteAtBeginning(head);
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr && temp->next->data != value)
        temp = temp->next;

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

    Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;
    delete nodeToDelete;
}

// Function to display the linked list
void displayList(Node* head) {
    cout << "Linked List: ";
    while (head != nullptr) {
        cout << head->data << " -> ";
        head = head->next;
    }
    cout << "NULL\n";
}

int main() {
    Node* head = nullptr;

    // Inserting elements
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
```

```
    insertAtEnd(head, 40);

    cout << "Original List:\n";
    displayList(head);

    deleteAtBeginning(head);
    cout << "After deleting from beginning:\n";
    displayList(head);

    deleteAtEnd(head);
    cout << "After deleting from end:\n";
    displayList(head);

    deleteByValue(head, 20);
    cout << "After deleting value 20:\n";
    displayList(head);

    return 0;
}
```

Output:

Original List:

Linked List: 10 -> 20 -> 30 -> 40 -> NULL
After deleting from beginning:
Linked List: 20 -> 30 -> 40 -> NULL
After deleting from end:
Linked List: 20 -> 30 -> NULL
After deleting value 20:
Linked List: 30 -> NULL

Result:

The above program was executed successfully and the desired output is obtained.

Ex No 5

Date: 28-07-2025

Inserting Element in a Doubly Linked List

Aim:

To write a C++ program to insert a new element into a doubly linked list

- At the beginning
- At the end
- After a given node (by key)

Algorithm:

Step 1: Start
Step 2: Create a linked list with nodes containing data and link fields.
Step 3: Insert elements at the end of the list.
Step 4: To delete a node at the beginning:

- Move the head pointer to the next node.
- Delete the first node.

Step 5: To delete a node at the end:

- Traverse the list to find the second last node.
- Set its link to NULL and delete the last node.

Step 6: To delete a node by value:

- Traverse the list to find the node with the given value.
- Adjust the links to skip that node and delete it.

Step 7: Display the final linked list after each deletion.
Step 8: Stop

Program:

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

// Node structure for Doubly Linked List
struct Node {
    int data;
    Node* prev;
    Node* next;
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->prev = newNode->next = nullptr;
    return newNode;
}

// Function to insert at the beginning
```

```
void insertAtBeginning(Node*& head, int data) {
    Node* newNode = createNode(data);
    if (head != nullptr) {
        newNode->next = head;
        head->prev = newNode;
    }
    head = newNode;
}

// Function to insert at the end
void insertAtEnd(Node*& head, int data) {
    Node* newNode = createNode(data);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr)
        temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

// Function to insert after a given node
void insertAfter(Node* head, int key, int data) {
    Node* temp = head;
    while (temp != nullptr && temp->data != key)
        temp = temp->next;

    if (temp == nullptr) {
        cout << "Node with data " << key << " not found.\n";
        return;
    }

    Node* newNode = createNode(data);
    newNode->next = temp->next;
    newNode->prev = temp;

    if (temp->next != nullptr)
        temp->next->prev = newNode;
    temp->next = newNode;
}

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    cout << "Doubly Linked List: ";
    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->next;
```

```
    }
    cout << "NULL\n";
}

int main() {
    Node* head = nullptr;

    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
    cout << "After inserting at end:\n";
    displayList(head);
    insertAtBeginning(head, 5);
    cout << "After inserting at beginning:\n";
    displayList(head);
    insertAfter(head, 20, 25);
    cout << "After inserting 25 after 20:\n";
    displayList(head);
    return 0;
}
```

Output:

After inserting at end:
Doubly Linked List: 10 <-> 20 <-> 30 <-> NULL
After inserting at beginning:
Doubly Linked List: 5 <-> 10 <-> 20 <-> 30 <-> NULL
After inserting 25 after 20:
Doubly Linked List: 5 <-> 10 <-> 20 <-> 25 <-> 30 <-> NULL

Result:

The above program was executed successfully and the desired output is obtained.

Ex No 6

Date: 29-07-2025

Operations of Stack Using Array

Aim:

To implement various deletion operations in a singly linked list such as deleting a node from the beginning, end, and by a specific value using C++.

Algorithm:

Step 1: Start
Step 2: Initialize an array `stack` and a variable `top = -1`.
Step 3: To push a value:
- If `top` is less than `MAX - 1`, increment `top` and store the value in `stack[top]`.
- Else, display "Stack Overflow".

Step 4: To pop a value:
- If `top` is greater than or equal to `0`, display `stack[top]` and decrement `top`.
- Else, display "Stack Underflow".

Step 5: To display the stack:
- If `top >= 0`, print all elements from `stack[top]` to `stack[0]`.
- Else, display "Stack is empty".

Step 6: Repeat steps 3–5 until the user chooses to exit.
Step 7: Stop

Program:

```
#include <iostream>
using namespace std;
#define MAX 100
int stack[MAX];
int top = -1;
// Push operation
void push(int value) {
    if (top >= MAX - 1) {
        cout << "Stack Overflow! Cannot push " << value << "\n";
        return;
    }
    stack[++top] = value;
    cout << value << " pushed to stack.\n";
}
// Pop operation
void pop() {
    if (top < 0) {
        cout << "Stack Underflow! Nothing to pop.\n";
        return;
    }
    cout << stack[top--] << " popped from stack.\n";
}
```

```
// Display operation
void display() {
    if (top < 0) {
        cout << "Stack is empty.\n";
        return;
    }
    cout << "Stack elements (top to bottom): ";
    for (int i = top; i >= 0; i--)
        cout << stack[i] << " ";
    cout << "\n";
}

int main() {
    int choice, value;

    do {
        cout << "\n--- Stack Menu ---\n";
        cout << "1. Push\n2. Pop\n3. Display\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter value to push: ";
            cin >> value;
            push(value);
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice!\n";
        }
    } while (choice != 4);

    return 0;
}
```

Output:

```
--- Stack Menu ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 20
20 pushed to stack.

--- Stack Menu ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 26
26 pushed to stack.

--- Stack Menu ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements (top to bottom): 26 20

--- Stack Menu ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
26 popped from stack.

--- Stack Menu ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting...
```

Result:

The above program was executed successfully and produce the expected result.

Ex No 7

Date: 4-08-2025

Operations of Queue Using Array

Aim:
To write and execute a C++ program to perform basic operations on a queue (Enqueue, Dequeue, and Display) using an array implementation.

Algorithm:

Step 1: Start
Step 2: Initialize `queue` array, `front = -1`, `rear = -1`.
Step 3: To enqueue an element:
- If the queue is not full, increment `rear` and insert the element.
- If the queue is empty (`front == -1`), set `front = 0`.

Step 4: To dequeue an element:
- If the queue is not empty, delete the element at `front` and increment `front`.

Step 5: To display the queue:
- If the queue is not empty, print all elements from `front` to `rear`.

Step 6: Repeat steps 3–5 until the user chooses to exit.
Step 7: Stop

Program:

```
#include <iostream>
using namespace std;
#define SIZE 5   // Maximum size of the queue
int queue[SIZE];
int front = -1;
int rear = -1;
// Function to check if the queue is full
bool isFull() {
    return (rear == SIZE - 1);
}
// Function to check if the queue is empty
bool isEmpty() {
    return (front == -1 || front > rear);
}
// Function to insert an element into the queue
void enqueue(int element) {
    if (isFull()) {
        cout << "Queue is full. Cannot insert " << element << endl;
    } else {
        if (front == -1) front = 0;  // first element
        rear++;
        queue[rear] = element;
        cout << element << " inserted into the queue." << endl;
    }
}
// Function to delete an element from the queue
```

```
void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty. Cannot delete element." << endl;
    } else {
        cout << queue[front] << " deleted from the queue." << endl;
        front++;
    }
}
// Function to display the queue
void display() {
    if (isEmpty()) {
        cout << "Queue is empty." << endl;
    } else {
        cout << "Queue elements are: ";
        for (int i = front; i <= rear; i++) {
            cout << queue[i] << " ";
        }
        cout << endl;
    }
}
// Main function
int main() {
    int choice, value;
    cout << "Queue Implementation using Array (without class)\n";
    cout << "------------------------------------------------\n";
    while (true) {
        cout << "\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter value to insert: ";
                cin >> value;
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                cout << "Exiting program.\n";
                return 0;
            default:
                cout << "Invalid choice. Try again.\n";
        }
    }
    return 0;
}
```

Output:

```
Queue Implementation using Array (without class)
----------------------------------------------

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 8
8 inserted into the queue.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 22
22 inserted into the queue.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements are: 8 22

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
8 deleted from the queue.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
Exiting program.
```

Result:

The above program was executed successfully and produce the expected output.

Ex No 8

Date: 5-08-2025

Evaluation of Postfix Expression

Aim:

To implement a C++ program to evaluate a postfix expression using a stack**.**

Algorithm:

Step 1: Start
Step 2: Read the postfix expression as a string.
Step 3: Initialize an empty stack.
Step 4: For each character in the expression:

- If it is a digit, convert it to integer and push onto the stack.
- If it is an operator (+, -, *, /):
  Pop two elements from the stack.
  Apply the operator to these two elements.
  Push the result back onto the stack.

Step 5: Repeat Step 4 for all characters in the expression.
Step 6: The final result will be the top element of the stack.
Step 7: Display the result.
Step 8: Stop

Program:

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>  // for isdigit()
using namespace std;

// Function to evaluate postfix expression
int evaluatePostfix(string expression)
{
    stack<int> s;
    for (char c : expression)
     {
        // If the character is a digit, push it to the stack
        if (isdigit(c)) {
            s.push(c - '0');  // Convert char to int
        }
        // If the character is an operator, pop two elements and apply the operator
        else {
            int val2 = s.top(); s.pop();
            int val1 = s.top(); s.pop();

            switch (c) {
                case '+': s.push(val1 + val2); break;
```

```
                case '-': s.push(val1 - val2); break;
                case '*': s.push(val1 * val2); break;
                case '/': s.push(val1 / val2); break;
            }
        }
    }

    // The result will be on the top of the stack
    return s.top();
}

int main() {
    string expression;
    cout << "Enter a postfix expression (e.g., 231*+9-): ";
    cin >> expression;

    int result = evaluatePostfix(expression);
    cout << "Result of the postfix expression: " << result << endl;

    return 0;
}
```

Output:

Enter a postfix expression (e.g., 231*+9-): 231*+9-
Result of the postfix expression: -4

Result:

The program successfully evaluates a postfix expression using a stack.

Ex No 9

Date: 12-08-2025

Conversion of Infix to Postfix Expression

Aim:

To write and execute a C++ program to convert a given infix expression into its equivalent postfix expression using a stack data structure.

Algorithm:

Step 1: Start
Step 2: Read the infix expression as a string.
Step 3: Initialize an empty stack for operators and an empty string for postfix expression.
Step 4: Scan each character of the infix expression:

- If it is an operand (letter or digit), append it to the postfix string.
- If it is `'('`, push it onto the stack.
- If it is `')'`, pop from the stack and append to postfix until `'('` is found, then remove `'('`.
- If it is an operator (`+`, `-`, `*`, `/`, `^`):
  While the stack is not empty and the precedence of the operator at the top of the stack is greater than or equal to the current operator, pop from the stack and append to postfix.
  Push the current operator onto the stack.

Step 5: After scanning the infix expression, pop all remaining operators from the stack and append them to postfix.
Step 6: Display the postfix expression.
Step 7: Stop

Program:

```
#include <iostream>
#include <stack>
#include <cctype>  // for isalpha() and isdigit()
using namespace std;
// Function to return precedence of operators
int precedence(char op) {
    if (op == '^')
        return 3;
    else if (op == '*' || op == '/')
        return 2;
    else if (op == '+' || op == '-')
        return 1;
    else
        return 0;
}
// Function to check if character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}
```

```
// Function to convert Infix to Postfix
string infixToPostfix(string infix) {
    stack<char> s;
    string postfix = "";

    for (int i = 0; i < infix.length(); i++) {
        char ch = infix[i];

        // If character is an operand, add to output
        if (isalnum(ch)) {
            postfix += ch;
        }
        // If character is '(', push to stack
        else if (ch == '(') {
            s.push(ch);
        }
        // If character is ')', pop until '('
        else if (ch == ')') {
            while (!s.empty() && s.top() != '(') {
                postfix += s.top();
                s.pop();
            }
            s.pop();  // remove '('
        }
        // If character is an operator
        else if (isOperator(ch)) {
            while (!s.empty() && precedence(s.top()) >= precedence(ch)) {
                postfix += s.top();
                s.pop();
            }
            s.push(ch);
        }
    }

    // Pop remaining operators from the stack
    while (!s.empty()) {
        postfix += s.top();
        s.pop();
    }
    return postfix;
}
// Main function
    int main() {
    string infix;
    cout << "Enter an infix expression: ";
    cin >> infix;
    string postfix = infixToPostfix(infix);
    cout << "Postfix Expression: " << postfix << endl;
    return 0;
}
```

Output:

Enter an infix expression: a+b
Postfix Expression: ab+

Enter an infix expression: a*b+c
Postfix Expression: ab*c+

Result:

The program successfully converts an infix expression to a postfix expression using a stack.

Ex No 10

Date: 20-08-2025

Bubble Sort

Aim :

To write a C++ program that accepts n elements from the user, stores them in an array, and sorts the array in ascending order using the Bubble Sort algorithm**.**

Algorithm:

Step 1: Start

Step 2: Read the number of elements `n` and the array elements.

Step 3: Repeat the following for `i = 0` to `n-2`:

- For `j = 0` to `n-i-2`:

  If `a[j] > a[j+1]`, swap `a[j]` and `a[j+1]`.

Step 4: After all passes, the array will be sorted in ascending order**.**

Step 5: Display the sorted array.

Step 6: Stop

Program:

```
#include <iostream>
using namespace std;
void bubbleSort(int a[], int n)
 {
   for (int i = 0; i < n - 1; i++)
       for (int j = 0; j < n - i - 1; j++)
           if (a[j] > a[j + 1])
           {
               int temp = a[j];
               a[j] = a[j + 1];
               a[j + 1] = temp;
           }
       }
  int main()
   {
   int a[20], n;
   cout << "Enter the value of n: ";
   cin >> n;
   cout << "Enter " << n << " values one by one: ";
   for (int i = 0; i < n; i++)
         cin >> a[i];
       cout << "Original array: ";
   for (int i = 0; i < n; i++)
         cout << a[i] << " ";
       cout << endl;
   bubbleSort(a, n);
   cout << "Sorted array: ";
   for (int i = 0; i < n; i++)
         cout << a[i] << " ";
       cout << endl;
   return 0;
}
```

Output:

Enter the value of n: 6
Enter 6 values one by one: 78
34
90
39
100
389
Original array: 78 34 90 39 100 389
Sorted array: 34 39 78 90 100 389

Result:
The program successfully sorts the given array using Bubble Sort**.**

Ex No 11

Date: 26-08-2025

Merge Sort Implementation

Aim :

To implement the Merge Sort algorithm in C++ to sort a list of elements in ascending order using the divide and conquer technique.

Algorithm:

Step 1: Start

Step 2: Read the number of elements `n` and the array elements.

Step 3: If the array has more than one element:

- Find the middle index `mid`.
- Recursively divide the array into two halves: left (`left` to `mid`) and right (`mid+1` to `right`).

Step 4: Merge the two sorted halves:

- Compare elements from both halves and copy the smaller element to the original array.
- Copy any remaining elements from both halves.

Step 5: Repeat Step 3 and Step 4 until the entire array is sorted.

Step 6: Display the sorted array.

Step 7: Stop

Program:

```
#include <iostream>
using namespace std;
// Function to merge two halves
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;   // size of left subarray
    int n2 = right - mid;      // size of right subarray

    // Create temporary arrays

    int L[n1], R[n2];
    // Copy data to temporary arrays
    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];

    // Merge the temporary arrays back into arr[left...right]

    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++;
    }

    // Copy remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Recursive Merge Sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        // Recursively divide the array
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to display the array
void display(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Main function
int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[n];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++)
```

```
        cin >> arr[i];

    cout << "\nUnsorted array: ";
    display(arr, n);

    mergeSort(arr, 0, n - 1);

    cout << "\nSorted array (Merge Sort): ";
    display(arr, n);

    return 0;
}
```

Output:

Enter number of elements: 6
Enter 6 elements: 38 27 43 3 9 82

Unsorted array: 38 27 43 3 9 82
Sorted array (Merge Sort): 3 9 27 38 43 82

Result:

The program successfully sorts the given array using Merge Sort**.**

Ex No 12

Date: 2-09-2025

Linear Search

Aim :

To write a C++ program that accepts n elements from the user, stores them in an array, and searches for a given element using the Linear Search technique**.**

Algorithm:

Step 1: Start

Step 2: Read the number of elements `n` and the array elements `a[0...n-1]`.

Step 3: Read the key element to be searched.

Step 4: For each element `a[i]` in the array (from `i = 0` to `n-1`):

- If `a[i] == key`, return the index `i` (element found).

Step 5: If the element is not found after checking all elements, return `-1`.

Step 6: Display the result:

- If the returned value is `-1`, print "Element not found".
- Otherwise, print the position and index of the element.

Step 7: Stop

Program:

```
#include <iostream>
using namespace std;
int linearSearch(int a[], int n, int key)
 {
    for (int i = 1; i < =n; i++)
    {
        if (a[i] == key)
            return i;
    }
    return -1;
}
int main()
{
    int a[20], n, key;
    cout << "Enter the number of elements: ";
    cin >> n;
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++)
        cin >> a[i];
    cout << "Enter the element to search: ";
    cin >> key;
    int result = linearSearch(a, n, key);
    if (result == -1)
        cout << "Element not found in the array." << endl;
    else
        cout << "Element found at position " << result + 1 << " (index " << result << ")." <<
endl;
    return 0;
}
```

Output:

Enter the number of elements: 6
Enter 6 elements: 45
12
78
37
91
23
Enter the element to search: 23
Element found at position 6 (index 5).

Result:

The program successfully searches for an element in the array using Linear Search**.**

Ex No 13

Date: 3-09-2025

Binary Search

Aim :

To write a C++ program that accepts n sorted elements from the user, stores them in an array, and searches for a given element using the Binary Search technique**.**

Algorithm:

Step 1: Start
Step 2: Read the number of elements `n` and the array elements (must be in sorted order**).**
Step 3: Read the element `key` to be searched.
Step 4: Initialize `low = 0` and `high = n - 1`.
Step 5: Repeat while `low <= high`:

- Calculate `mid = (low + high) / 2`.
- If `a[mid] == key`, the element is found → return `mid`.
- Else if `a[mid] < key`, search in the right half → set `low = mid + 1`.
- Else search in the left half → set `high = mid - 1`.

Step 6: If `low > high`, the element is not found → return `-1`.
Step 7: Display the result:

- If returned value is `-1`, print "Element not found".
- Otherwise, print the position and index of the element.

Step 8: Stop

Program:

```
#include <iostream>
using namespace std;
int binarySearch(int a[], int n, int key)
 {
    int low = 0, high = n - 1;
    while (low <= high)
     {
        int mid = (low + high) / 2;
        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;    }
    return -1; // not found
}

int main()
{
    int a[20], n, key;
    cout << "Enter the number of elements: ";
    cin >> n;
```

```
    cout << "Enter " << n << " elements in sorted order: ";
    for (int i = 0; i < n; i++)
        cin >> a[i];
    cout << "Enter the element to search: ";
    cin >> key;
    int result = binarySearch(a, n, key);
    if (result == -1)
        cout << "Element not found in the array." << endl;
    else
        cout << "Element found at position " << result + 1 << " (index " << result << ")." <<
endl;
    return 0;
}
```

Output:

Enter the number of elements: 4
Enter 4 elements in sorted order: 8
14
21
33
Enter the element to search: 21
Element found at position 3 (index 2).

Result:

The program successfully searches for an element in a sorted array using Binary Search**.**

Ex No 14

Date: 11-09-2025

Binary Tree Traversal

Aim:

To implement Binary Tree Traversal (Inorder, Preorder, and Postorder) in C++ using recursion.

Algorithm:

Step 1: Start
Step 2: Create a structure `Node` with fields:
- `data` → stores the value of the node
- `left` → pointer to the left child
- `right` → pointer to the right child

Step 3: Create the binary tree by linking nodes appropriately.
Step 4: Inorder Traversal (Left → Root → Right):
- If the current node is not `nullptr`:
    - Traverse the left subtree recursively.
    - Print the current node's data.
    - Traverse the right subtree recursively.

Step 5: Preorder Traversal (Root → Left → Right):
- If the current node is not `nullptr`:
    - Print the current node's data.
    - Traverse the left subtree recursively.
    - Traverse the right subtree recursively.

Step 6: Postorder Traversal (Left → Right → Root):
- If the current node is not `nullptr`:
    - Traverse the left subtree recursively.
    - Traverse the right subtree recursively.
    - Print the current node's data.

Step 7: Call the traversal functions on the root node to display the nodes in different orders.
Step 8: Stop

Program:

```
#include <iostream>
using namespace std;
// Structure for a node in the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) {
        data = value;
```

```
        left = right = nullptr;
    }
};

// Function for Inorder Traversal (Left → Root → Right)
void inorder(Node* root) {
    if (root == nullptr)
        return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

// Function for Preorder Traversal (Root → Left → Right)
void preorder(Node* root) {
    if (root == nullptr)
        return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

// Function for Postorder Traversal (Left → Right → Root)
void postorder(Node* root) {
    if (root == nullptr)
        return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

int main() {
    // Creating a sample binary tree
    //        1
    //       / \
    //      2   3
    //     / \
    //    4   5

    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);

    cout << "Inorder Traversal: ";
    inorder(root);

    cout << "\nPreorder Traversal: ";
    preorder(root);
```

```
    cout << "\nPostorder Traversal: ";
    postorder(root);

    return 0;
}
```

Output:

Inorder Traversal: 4 2 5 1 3
Preorder Traversal: 1 2 4 5 3
Postorder Traversal: 4 5 2 3 1

Result:

The program successfully performs Inorder, Preorder, and Postorder traversals on a binary tree.

Ex No 15

Date: 18-09-2025

Depth First Search using Recursion

Aim:

To implement the Depth First Search (DFS) algorithm in C++ to traverse all the vertices of a graph using recursion.

Algorithm:

Step 1: Start
Step 2: Read the number of vertices and edges.
Step 3: Create an adjacency list to represent the graph.

- For each edge `(u, v)`, add `v` to the adjacency list of `u` and `u` to the adjacency list of `v` (for undirected graph).

Step 4: Initialize a visited array of size `vertices + 1` with all values as `false`.
Step 5: Perform DFS starting from a given node (e.g., vertex `1`):

- Mark the current node as visited.
- Print the current node.
- For each neighbor of the current node:
    - If the neighbor is not visited, recursively call DFS on that neighbor.

Step 6: Repeat Step 5 until all reachable vertices are visited.
Step 7: Stop

Program:

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

// Function to perform DFS traversal
void DFS(int node, vector<int> adj[], vector<bool> &visited) {
    // Mark the current node as visited
    visited[node] = true;
    cout << node << " ";  // Print the visited node

    // Recur for all adjacent vertices
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            DFS(neighbor, adj, visited);
        }
    }
}

int main() {
    int vertices, edges;
    cout << "Enter number of vertices: ";
```

```
    cin >> vertices;
    cout << "Enter number of edges: ";
    cin >> edges;

    vector<int> adj[vertices + 1];  // Adjacency list

    cout << "Enter edges (u v):" << endl;
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // For undirected graph
    }

    vector<bool> visited(vertices + 1, false);
    cout << "\nDFS Traversal starting from vertex 1: ";
    DFS(1, adj, visited);

    return 0;
}
```

Output:

Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
1 2
1 3
2 4
3 5

DFS Traversal starting from vertex 1: 1 2 4 3 5

Result:
The program successfully performs DFS traversal of the graph.

Ex No 16

Date: 25-09-2025

Breadth First Search Traversal

Aim:

To implement Breadth First Search (BFS) algorithm in C++ to traverse all the vertices of a graph using a queue**.**

Algorithm:

Step 1: Start
Step 2: Read the number of vertices and edges**.**
Step 3: Create an adjacency list to represent the graph.

- For each edge `(u, v)`, add `v` to the list of `u` and `u` to the list of `v` (for undirected graph).

Step 4: Read the starting vertex for BFS.
Step 5: Initialize a visited array of size `vertices + 1` with all values `false`.
Step 6: Initialize a queue and:

- Mark the starting vertex as visited.
- Enqueue the starting vertex.

Step 7: While the queue is not empty:

- Dequeue a vertex `node` and print it.
- For each adjacent vertex of `node`:
    - If it is not visited, mark it as visited and enqueue it.

Step 8: Repeat Step 7 until all reachable vertices are visited.
Step 9: Stop

Program:

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

// Function to perform BFS traversal
void BFS(int start, vector<int> adj[], int vertices)
{
    vector<bool> visited(vertices + 1, false); // To keep track of visited nodes
    queue<int> q;                              // Queue for BFS

    visited[start] = true;
    q.push(start);
    cout << "BFS Traversal: ";
    while (!q.empty())
            {
        int node = q.front();
        q.pop();
        cout << node << " ";
```

```
        // Visit all adjacent unvisited vertices
        for (int neighbor : adj[node])
              {
            if (!visited[neighbor])
                          {
                visited[neighbor] = true;
                q.push(neighbor);
               }
        }
    }
}

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

    vector<int> adj[vertices + 1]; // Adjacency list

    cout << "Enter edges (u v):" << endl;
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // For undirected graph
    }
    int start;
    cout << "Enter starting vertex: ";
    cin >> start;
    BFS(start, adj, vertices);
    return 0;
}
```

Output:

Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
1 2
1 3
2 4
3 5
Enter starting vertex: 1
BFS Traversal: 1 2 3 4 5

Result:
The program successfully performs BFS traversal on the given graph.