# Experiment:- 1

**Aim** - To write a program that stores the first 10 natural numbers in an array and calculates their average.

## Objectives -

- To understand the concept of natural numbers.
- To learn how to store data in an array.
- To implement looping for inserting and displaying elements.
- To calculate the sum and average of numbers using an array.

## Algorithm -

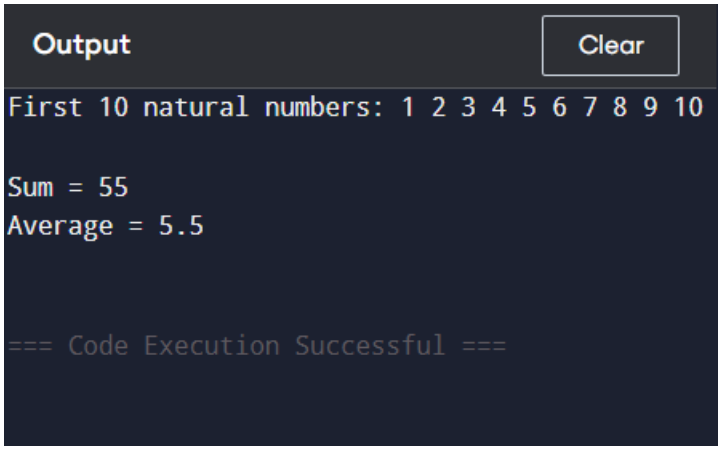
Step 1: Start
Step 2: Declare an array and variables
Step 3: Store the first 10 natural numbers in the array
Step 4: Calculate the sum of array elements
Step 5: Find the average (sum ÷ total numbers)
Step 6: Display the numbers, sum, and average
Step 7: Stop

## Program –

```
#include <iostream>
using namespace std;
int main() {
   int numbers[10];
   int sum = 0;
   float average;
   // Store first 10 natural numbers in array
   for (int i = 0; i < 10; i++) {
      numbers[i] = i + 1;
   }
   // Calculate sum
   for (int i = 0; i < 10; i++) {
      sum += numbers[i];
   }

   // Calculate average
```

```
    average = sum / 10.0;
    // Display numbers
    cout << "First 10 natural numbers: ";
    for (int i = 0; i < 10; i++) {
        cout << numbers[i] << " ";
    }
    cout << "\nSum = " << sum;
    cout << "\nAverage = " << average << endl;
    return 0;
}
```

**Output -**



# Experiment:- 2

**Aim -** Write a program to insert and delete an element in an array.

## Objectives –

- To understand how array elements are stored in contiguous memory.
- To learn how to perform insertion of an element at a specific position in an array.
- To learn how to perform deletion of an element from a specific position in an array.
- To practice shifting elements in an array during insertion and deletion.
- To implement basic array operations using C programming.

## Algorithm -

1. Input the number of elements n.
2. Read n elements into the array.
3. Input the position pos where the new element should be inserted.
4. Input the value val of the new element.
5. Shift elements of the array to the right.
    - For i = n down to pos
        - array[i] = array [i - 1]
6. Insert the new element:
    - array [pos - 1] = val
7. Increase the size of array:
    - n = n + 1
8. Display the updated array.
9. Stop

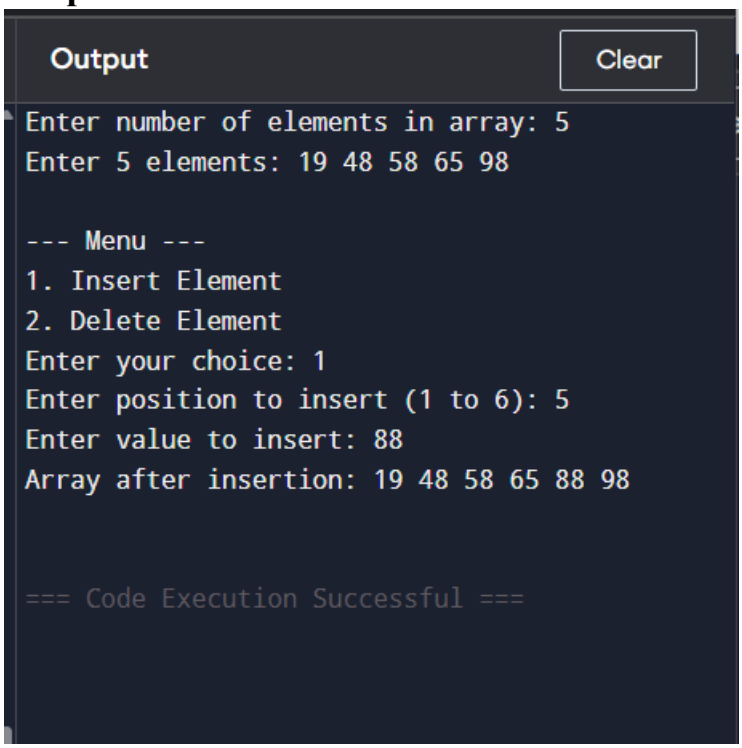
## Program –

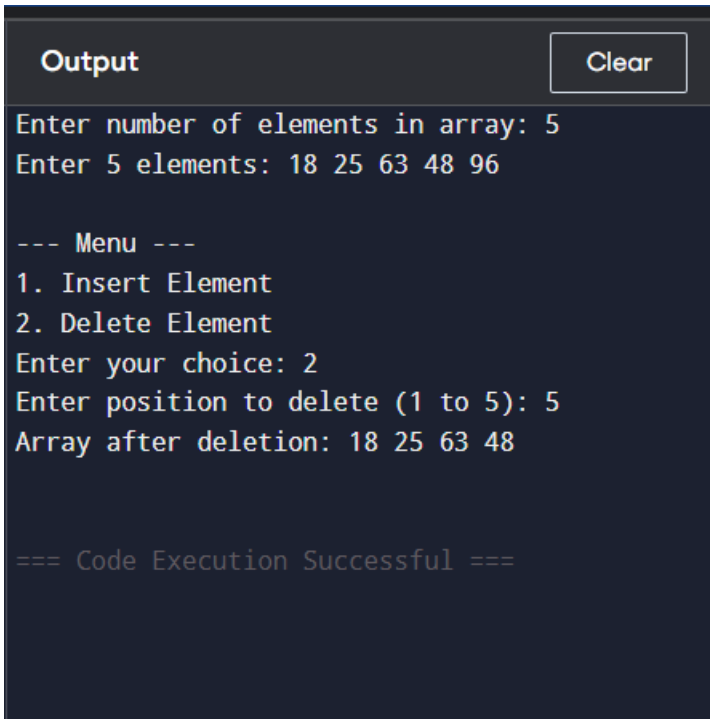
```
#include <iostream>
using namespace std;

int main() {
   int arr[100], n, pos, value, choice;

   cout << "Enter number of elements in array: ";
   cin >> n;
```

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

cout << "\n--- Menu ---";
cout << "\n1. Insert Element";
cout << "\n2. Delete Element";
cout << "\nEnter your choice: ";
cin >> choice;

switch (choice) {
   case 1:
      cout << "Enter position to insert (1 to " << n + 1 << "): ";
      cin >> pos;
      cout << "Enter value to insert: ";
      cin >> value;

      if (pos < 1 || pos > n + 1)
         cout << "Invalid position!";
      else {
         for (int i = n; i >= pos; i--)
            arr[i] = arr[i - 1];
         arr[pos - 1] = value;
         n++;

         cout << "Array after insertion: ";
         for (int i = 0; i < n; i++)
            cout << arr[i] << " ";
         cout << endl;
      }
      break;

   case 2:
      cout << "Enter position to delete (1 to " << n << "): ";
      cin >> pos;

      if (pos < 1 || pos > n)
         cout << "Invalid position!";
      else {
         for (int i = pos - 1; i < n - 1; i++)
            arr[i] = arr[i + 1];
         n--;
```

```
            cout << "Array after deletion: ";
            for (int i = 0; i < n; i++)
                cout << arr[i] << " ";
            cout << endl;
        }
        break;

    default:
        cout << "Invalid choice!";
  }

  return 0;
}
```

**Output :–**




Output
Clear
Enter number of elements in array: 5
Enter 5 elements: 18 25 63 48 96
--- Menu ---
1. Insert Element
2. Delete Element
Enter your choice: 2
Enter position to delete (1 to 5): 5
Array after deletion: 18 25 63 48
=== Code Execution Successful ===

# Experiment:- 3

**Aim -** To implement an algorithm for insertion and deletion operations in a Circular Queue using an array.

## Objective –

- To understand the concept and working of a Circular Queue data structure.
- To learn how to perform insertion (enqueue) and deletion (dequeue) operations in a circular queue.
- To implement a Circular Queue using arrays (static memory allocation).
- To understand and handle overflow and underflow conditions efficiently.
- To utilize the modulus operator (%) for achieving circular movement in the queue.

## Theory -

A Circular Queue is a type of queue in which the last position is connected back to the first position, forming a circle.
It helps in efficiently using memory by reusing empty spaces created by deleted elements.

## Key Points –

- It follows the FIFO (First In, First Out) principle.
- When the queue reaches the end of the array, it wraps around to the beginning using the modulus operator (%).
- Two pointers are used:
- front → indicates the position of the first element.
- rear → indicates the position of the last element.

## Conditions -

- Queue Empty: front == -1
- Queue Full: (rear + 1) % SIZE == front

## Algorithm -

q. Insertion (Enqueue)

1. Check if the queue is full.
    - If (rear + 1) % SIZE == front, then overflow.
2. If the queue is empty (front == -1), set front = 0 and rear = 0.
3. Else, set rear = (rear + 1) % SIZE.

4. Insert the new element at queue[rear].

2. Deletion (Dequeue)
   1. Check if the queue is empty.
      - If front == -1, then underflow.
   2. Retrieve the element from queue[front].
   3. If front == rear, set both front = -1 and rear = -1 (queue becomes empty).
   4. Else, set front = (front + 1) % SIZE.
3. Display
   1. If the queue is empty, print "Queue is empty".
   2. Else, print all elements from front to rear circularly using modulo.

## Program -

```
#include <iostream>

using namespace std;

#define SIZE 5

int queue[SIZE], front = -1, rear = -1;

bool isFull() {
   return (front == (rear + 1) % SIZE);
}

bool isEmpty() {
   return (front == -1);
}

void enqueue(int value) {
   if (isFull()) {
      cout << "Queue is full!\n";
      return;
   }
   if (front == -1) front = 0;
   rear = (rear + 1) % SIZE;
   queue[rear] = value;
   cout << value << " inserted.\n";
}
```

```
void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty!\n";
        return;
    }
    cout << queue[front] << " deleted.\n";
    if (front == rear)
        front = rear = -1;
    else
        front = (front + 1) % SIZE;
}
void display() {
    if (isEmpty()) {
        cout << "Queue is empty!\n";
        return;
    }
    cout << "Queue elements: ";
    int i = front;
    while (true) {
        cout << queue[i] << " ";
        if (i == rear) break;
        i = (i + 1) % SIZE;
    }
    cout << endl;
}
int main() {
    int choice, val;
    do {
```

```
    cout << "\n1.Insert  2.Delete  3.Display  4.Exit\nEnter choice: ";
    cin >> choice;
    switch (choice) {
        case 1: cout << "Enter value: "; cin >> val; enqueue(val); break;
        case 2: dequeue(); break;
        case 3: display(); break;
    }
  } while (choice != 4);
  return 0;
}
```

**Output –**

```
Output                                                    Clear

1.Insert  2.Delete  3.Display  4.Exit
Enter choice: 1
Enter value: 12
12 inserted.

1.Insert  2.Delete  3.Display  4.Exit
Enter choice: 1
Enter value: 45
45 inserted.

1.Insert  2.Delete  3.Display  4.Exit
Enter choice: 3
Queue elements: 12 45

1.Insert  2.Delete  3.Display  4.Exit
Enter choice:
```

# Experiment:- 4

**Aim** - write a program to implement stack operations PUSH, POP, and DISPLAY using static memory allocation in C++.

## Objectives -

- To understand the concept and working of the Stack data structure.
- To learn how to implement PUSH, POP, and DISPLAY operations using an array.
- To demonstrate static memory allocation for managing stack elements.
- To identify and handle stack overflow and underflow conditions.
- To develop problem-solving skills using LIFO (Last In, First Out) logic.

## Theory -

A Stack is a linear data structure that follows the LIFO (Last In, First Out) principle — the element inserted last is removed first.

## Basic Operations -

1. PUSH: To insert an element into the stack.
2. POP: To remove the top element from the stack.
3. DISPLAY: To show all elements present in the stack.

## Static Memory Allocation -

In this method, a fixed-size array is used to store the stack elements — its size is defined at compile time.

## Stack Conditions -

| Condition | Description |
|---|---|
| top == -1 | Stack is empty |
| top == SIZE - 1 | Stack is full |

## Algorithm -

1. PUSH Operation

    1. Check if top == SIZE - 1
       → Stack Overflow

2. Else, increment top by 1.
3. Insert the new element at stack[top].

2. POP Operation

1. Check if top == -1
   → Stack Underflow
2. Else, print stack[top] as deleted element.
3. Decrement top by 1.

3. DISPLAY Operation

1. If top == -1 → Stack is empty.
2. Else, print all elements from top to 0.

## Program -

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

#define SIZE 5 // Static memory size for stack

class Stack {
int stack [SIZE];
int top;

public:
// Constructor to initialize top
Stack () {
top = -1;
}

// Function to push an element
void push (int value) {
if (top == SIZE - 1) {
cout << "Stack Overflow! Cannot push " << value << endl;
} else {
top++;
stack[top] = value;
cout << value << " pushed into stack." << endl;
}
}// Function to pop an element
```

```
void pop () {
if (top == -1) {
cout << "Stack Underflow! No element to pop." << endl;
} else {
cout << stack[top] << " popped from stack." << endl;
top--;
}
}// Function to display stack elements
void display () {
if (top == -1) {
cout << "Stack is empty." << endl;
} else {
cout << "Stack elements are: ";
for (int i = top; i >= 0; i--) {
cout << stack[i] << " ";
}
cout << endl;
}
}
};// Main function
int main () {
Stack s;
int choice, value;
while (true) {
cout << "\n----- Stack Menu -----" << endl;
cout << "1. PUSH\n2. POP\n3. DISPLAY\n4. EXIT" << endl;
cout << "Enter your choice: ";
Cin >> choice;
switch (choice) {
case 1:
cout << "Enter value to push: ";
Cin >> value;
s.push(value);
break;
case 2:
s.pop();
break;
case 3:
s.display();
break;
case 4:
cout << "Exiting program..." << endl;
return 0;
```

```
default:
cout << "Invalid choice! Try again." << endl;
}
}
}
```

**Output: –**

```
Output

----- Stack Menu -----
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 45
45 pushed into stack.

----- Stack Menu -----
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 88
88 pushed into stack.

----- Stack Menu -----
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 3
Stack elements are: 88 45
```

# Experiment:- 5

**Aim -** To write a C++ program to implement PUSH, POP, and DISPLAY operations on a stack using dynamic memory allocation.

## Objectives -

- To understand the concept of stack as a LIFO (Last In, First Out) data structure.
- To learn how to implement stack operations using pointers and dynamic memory allocation.
- To perform PUSH, POP, and DISPLAY operations using linked list implementation.
- To manage memory efficiently at runtime using new and delete operators in C++.
- To identify and handle overflow and underflow conditions in dynamic stacks.

## Algorithm -

1. PUSH Operation
   1. Create a new node dynamically using new.
   2. Store the element (data) in the new node.
   3. Point the new node's next pointer to the current top node.
   4. Make the new node as the new top.

2. POP Operation
   1. Check if the stack is empty (top == NULL).
   2. If empty, print "Stack Underflow".
   3. Else, delete the top node and move the top pointer to the next node.

3. DISPLAY Operation
   1. Check if the stack is empty.
   2. If not, traverse from the top node and print all elements.

## C++ Program -

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

// Node structure for dynamic stack
struct Node {
```

```
int data;
Node* next;
};

class Stack {
Node* top; // pointer to the top of the stack

public:
Stack () {
top = NULL;
}

// PUSH operation
void push (int value) {
Node* new Node = new Node (); // dynamic allocation
new Node->data = value;
new Node->next = top;
top = new Node;
cout << value << " pushed into stack." << endl;
}

// POP operation
void pop () {
if (top == NULL) {
cout << "Stack Underflow! Cannot pop." << endl;
} else {
Node* temp = top;
cout << temp->data << " popped from stack." << endl;
top = top->next;
delete temp; // deallocate memory
}
}

// DISPLAY operation
void display () {
if (top == NULL) {
cout << "Stack is empty!" << endl;
} else {
Node* temp = top;
cout << "Stack elements are: ";
while (temp! = NULL) {
cout << temp->data << " ";
temp = temp->next;
```

```
}
cout << endl;
}
}
};

// Main function
int main () {
Stack s;
int choice, value;

do {
cout << "\n--- Stack Operations using Dynamic Memory ---" << endl;
cout << "1. PUSH\n2. POP\n3. DISPLAY\n4. EXIT" << endl;
cout << "Enter your choice: ";
Cin >> choice;

switch (choice) {
case 1:
cout << "Enter value to PUSH: ";
Cin >> value;
s. push(value);
break;
case 2:
soup ();
break;
case 3:
s. display ();
break;
case 4:
cout << "Exiting program..." << endl;
break;
default:
cout << "Invalid choice!" << endl;
}
} while (choice! = 4);

return 0;
}
```

**Output -**

Output Clear

```
--- Stack Operations using Dynamic Memory ---
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to PUSH: 3
3 pushed into stack.
```

# Experiment:- 6

**Aim -** To write a C++ program to implement Insertion (Enqueue), Deletion (Dequeue), and Display operations on a Linear Queue using arrays (static memory allocation).

## Objectives -

- To understand the concept of Queue as a FIFO (First In, First Out) data structure.
- To learn how to implement queue operations using arrays.
- To perform Insertion, Deletion, and Display operations with static memory allocation.
- To handle Overflow and Underflow conditions in queues.
- To improve problem-solving skills in implementing linear data structures.

## Algorithm -

1. Enqueue (Insertion)
    1. Check if the queue is full (rear == SIZE - 1).
    2. If full, print "Queue Overflow".
    3. Else, increment rear and insert the new element at queue[rear].

2. Dequeue (Deletion)
    1. Check if the queue is empty (front == -1 or front > rear).
    2. If empty, print "Queue Underflow".
    3. Else, delete the element at queue[front] and increment front.

3. Display
    1. Check if the queue is empty.
    2. If not, display all elements from front to rear.

## C++ Program -

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

#define SIZE 5   // maximum size of the queue

class Linear Queue {
int queue [SIZE];
```

```
int front, rear;

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

// Function to insert an element
void enqueue (int value) {
if (rear == SIZE - 1) {
cout << "Queue Overflow! Cannot insert " << value << endl;
return;
}

if (front == -1) {
front = 0; // first element insertion
}

rear++;
queue[rear] = value;
cout << value << " inserted into the queue." << endl;
}

// Function to delete an element
void dequeue () {
if (front == -1 || front > rear) {
cout << "Queue Underflow! Cannot delete." << endl;
return;
}

cout << queue[front] << " deleted from the queue." << endl;
front++;
}

// Function to display all elements
void display () {
if (front == -1 || front > rear) {
cout << "Queue is empty!" << endl;
return;
}
```

```
cout << "Queue elements are: ";
for (int i = front; i <= rear; i++) {
cout << queue[i] << " ";
}
cout << endl;
}
};

// Main function
int main () {
Linear Queue q;
int choice, value;

do {
cout << "\n--- Linear Queue Operations using Static Memory Allocation ---" << endl;
cout << "1. Enqueue (Insert)\n2. Dequeue (Delete)\n3. Display\n4. Exit" << endl;
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:
q. display ();
break;

case 4:
cout << "Exiting program..." << endl;
break;

default:
cout << "Invalid choice! Try again." << endl;
}
} while (choice! = 4);
```

```
return 0;
}
```

**Output -**

Output Clear

```
--- Linear Queue Operations using Static Memory Allocation ---
1. Enqueue (Insert)
2. Dequeue (Delete)
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 55
55 inserted into the queue.
```

# Experiment:-7

**Aim -** To write a C++ program to implement Insertion (Enqueue), Deletion (Dequeue), and Display operations on a Linear Queue using dynamic memory allocation (linked list).

## Objectives -

- To understand the concept of Queue as a FIFO (First In, First Out) data structure.
- To implement queue operations using dynamic memory allocation with linked lists.
- To perform Insertion, Deletion, and Display operations using pointers.
- To handle Overflow and Underflow conditions efficiently.
- To learn how to manage memory dynamically using the new and delete operators.

## Algorithm -

1. Enqueue (Insertion)
    1. Create a new node dynamically using new.
    2. Store the element (data) in the new node.
    3. If the queue is empty, set both front and rear to this node.
    4. Otherwise, link the new node at the end (rear->next = new Node) and update rear.

2. Dequeue (Deletion)
    1. Check if the queue is empty (front == NULL).
    2. If empty, print "Queue Underflow".
    3. Else, delete the node pointed by front, move front to the next node, and free memory using delete.

3. Display
    1. Check if the queue is empty.
    2. If not, traverse from front to rear and print all elements.

## Program -

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

// Node structure for dynamic queue
```

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

class Queue {
Node* front;
Node* rear;

public:
// Constructor
Queue () {
front = NULL;
rear = NULL;
}

// Enqueue operation (Insertion)
void enqueue (int value) {
Node* new Node = new Node (); // dynamic memory allocation
new Node->data = value;
new Node->next = NULL;

if (rear == NULL) {
// Queue is empty
front = rear = new Node;
} else {
rear->next = new Node;
rear = new Node;
}

cout << value << " inserted into the queue." << endl;
}

// Dequeue operation (Deletion)
void dequeue () {
if (front == NULL) {
cout << "Queue Underflow! Cannot delete." << endl;
return;
}

Node* temp = front;
cout << temp->data << " deleted from the queue." << endl;
front = front->next;
```

```
if (front == NULL) {
rear = NULL; // if queue becomes empty
}

delete temp; // free memory
}

// Display operation
void display () {
if (front == NULL) {
cout << "Queue is empty!" << endl;
return;
}

Node* temp = front;
cout << "Queue elements are: ";
while (temp! = NULL) {
cout << temp->data << " ";
temp = temp->next;
}
cout << endl;
}
};

// Main function
int main () {
Queue q;
int choice, value;

do {
cout << "\n--- Linear Queue Operations using Dynamic Memory Allocation ---" << endl;
cout << "1. Enqueue (Insert)\n2. Dequeue (Delete)\n3. Display\n4. Exit" << endl;
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:
q. display ();
break;
case 4:
cout << "Exiting program..." << endl;
break;
default:
cout << "Invalid choice! Try again." << endl;
}
} while (choice! = 4);

return 0;
}
```

**Output:-**

Output Clear

```
--- Linear Queue Operations using Dynamic Memory Allocation ---
1. Enqueue (Insert)
2. Dequeue (Delete)
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 54
54 inserted into the queue.
```

# Experiment:- 8

**Aim** - To write a C++ program to implement various operations on a Linear Linked List, such as insertion, deletion, display, and search, using dynamic memory allocation.

## Objectives -

- To understand the concept and structure of a Linked List.
- To learn how to implement a Linear Linked List using pointers and dynamic memory allocation.
- To perform the main operations: Insertion, Deletion, Display, and Search.
- To understand how nodes are dynamically created and deleted at runtime using new and delete.
- To improve programming and problem-solving skills in handling dynamic data structures.

## Algorithm -

1. Insertion

1. Create a new node dynamically using new.
2. If the list is empty, make the new node the head.
3. Otherwise, traverse to the end of the list and insert the new node there.

2. Deletion

1. Check if the list is empty.
2. If not, search for the node to be deleted.
3. Adjust pointers to skip that node and deallocate its memory using delete.

3. Display

1. Start from the head node.
2. Traverse through the list and print all node values until NULL is reached.

4. Search

1. Start from the head node.
2. Compare each node's data with the search key.
3. If found, print its position; else display "Element not found".

## Program -

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

// Node structure
struct Node {
int data;
```

```
Node* next;
};

class LinkedList {
Node* head;

public:
// Constructor
LinkedList () {
head = NULL;
}

// Function to insert a new node at the end
void insert (int value) {
Node* new Node = new Node (); // dynamic memory allocation
new Node->data = value;
new Node->next = NULL;

if (head == NULL) {
head = new Node;
cout << value << " inserted as the first node." << endl;
} else {
Node* temp = head;
while (temp->next! = NULL) {
temp = temp->next;
}
temp->next = new Node;
cout << value << " inserted at the end." << endl;
}
}

// Function to delete a node
void delete Node (int value) {
if (head == NULL) {
cout << "List is empty! Nothing to delete." << endl;
return;
}

Node* temp = head;
Node* Prev = NULL;

// If head node itself holds the key
if (temp! = NULL && temp->data == value) {
```

```
head = temp->next;
delete temp;
cout << value << " deleted from the list." << endl;
return;
}

// Search for the node to be deleted
while (temp! = NULL && temp->data! = value) {
Prev = temp;
temp = temp->next;
}

// If value was not found
if (temp == NULL) {
cout << "Element " << value << " not found!" << endl;
return;
}

// Unlink and delete the node
Prev->next = temp->next;
delete temp;
cout << value << " deleted from the list." << endl;
}

// Function to display all nodes
void display () {
if (head == NULL) {
cout << "List is empty!" << endl;
return;
}

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

// Function to search for an element
void search (int value) {
Node* temp = head;
```

```
int pos = 1;
while (temp! = NULL) {
if (temp->data == value) {
cout << value << " found at position " << pos << "." << endl;
return;
}
temp = temp->next;
pos++;
}
cout << value << " not found in the list." << endl;
}
};

// Main function
int main () {
LinkedList list;
int choice, value;

do {
cout << "\n--- Linear Linked List Operations ---" << endl;
cout << "1. Insert\n2. Delete\n3. Display\n4. Search\n5. Exit" << endl;
cout << "Enter your choice: ";
Cin >> choice;

switch (choice) {
case 1:
cout << "Enter value to insert: ";
Cin >> value;
list. Insert(value);
break;

case 2:
cout << "Enter value to delete: ";
Cin >> value;
list.deleteNode(value);
break;

case 3:
list.display();
break;

case 4:
cout << "Enter value to search: ";
```

```
Cin >> value;
list.search(value);
break;

case 5:
cout << "Exiting program..." << endl;
break;

default:
cout << "Invalid choice! Try again." << endl;
}

} while (choice != 5);

return 0;
}
```

**Output -**

Output Clear

```
--- Linear Linked List Operations ---
1. Insert
2. Delete
3. Display
4. Search
5. Exit
Enter your choice: 1
Enter value to insert: 5
5 inserted as the first node.
```

# Experiment:- 9

**Aim** - To write a C++ program to implement Insertion, Deletion, Display, and Search operations on a Circular Linked List using dynamic memory allocation.

## Objectives -

- To understand the concept and working of a Circular Linked List.
- To learn how to implement circular linked lists using pointers and dynamic memory allocation.
- To perform insertion, deletion, display, and search operations efficiently.
- To understand the circular nature where the last node points back to the first node.
- To practice memory management using new and delete in C++.

## Algorithm:

1. Insertion
    1. Create a new node using new.
    2. If the list is empty, point new Node->next to itself.
    3. If not, traverse to the last node, adjust pointers so that the new node becomes part of the circle.

2. Deletion
    1. If the list is empty, print “List is empty.”
    2. If there is only one node, delete it and set head = NULL.
    3. Otherwise, traverse to find the node, update pointers, and delete it.

3. Display
    1. Start from the head node.
    2. Traverse the list until you return to the head again.
    3. Print each node’s data.

4. Search
    1. Start from head and traverse circularly.
    2. Compare each node’s data with the search key.
    3. If found, print position; else, display “Not found”.

## Program:

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

// Node structure
```

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

class Circular LinkedList {
Node* head;

public:
// Constructor
Circular LinkedList () {
head = NULL;
}

// Function to insert a node at the end
void insert (int value) {
Node* new Node = new Node ();
new Node->data = value;
new Node->next = NULL;

if (head == NULL) {
head = new Node;
new Node->next = head;
cout << value << " inserted as the first node." << endl;
} else {
Node* temp = head;
while (temp->next! = head)
temp = temp->next;
temp->next = new Node;
new Node->next = head;
cout << value << " inserted into the list." << endl;
}
}

// Function to delete a node
void delete Node (int value) {
if (head == NULL) {
cout << "List is empty! Nothing to delete." << endl;
return;
}

Node* current = head;
Node* previous = NULL;
```

```
// Case 1: Only one node
if (head->data == value && head->next == head) {
delete head;
head = NULL;
cout << value << " deleted, list is now empty." << endl;
return;
}

// Case 2: Deleting head node with multiple elements
if (head->data == value) {
Node* temp = head;
while (temp->next! = head)
temp = temp->next;
temp->next = head->next;
Node* to Delete = head;
head = head->next;
delete to Delete;
cout << value << " deleted from the list." << endl;
return;
}

// Case 3: Deleting non-head node
do {
previous = current;
current = current->next;
} while (current! = head && current->data! = value);

if (current->data == value) {
previous->next = current->next;
delete current;
cout << value << " deleted from the list." << endl;
} else {
cout << "Element " << value << " not found in the list!" << endl;
}
}

// Function to display the circular linked list
void display () {
if (head == NULL) {
cout << "List is empty!" << endl;
return;
}
```

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

// Function to search for a value
void search (int value) {
if (head == NULL) {
cout << "List is empty!" << endl;
return;
}

Node* temp = head;
int pos = 1;
do {
if (temp->data == value) {
cout << value << " found at position " << pos << "." << endl;
return;
}
temp = temp->next;
pos++;
} while (temp! = head);

cout << value << " not found in the list." << endl;
}
};

// Main function
int main () {
Circular LinkedList call;
int choice, value;

do {
cout << "\n--- Circular Linked List Operations ---" << endl;
cout << "1. Insert\n2. Delete\n3. Display\n4. Search\n5. Exit" << endl;
cout << "Enter your choice: ";
Cin >> choice;
```

```
switch (choice) {
case 1:
cout << "Enter value to insert: ";
Cin >> value;
cloister(value);
break;

case 2:
cout << "Enter value to delete: ";
Cin >> value;
cll. delete Node(value);
break;

case 3:
cll.display();
break;

case 4:
cout << "Enter value to search: ";
Cin >> value;
cll.search(value);
break;

case 5:
cout << "Exiting program..." << endl;
break;

default:
cout << "Invalid choice! Try again." << endl;
}

} while (choice! = 5);

return 0;
}
```

**Outcome –**

**Output**

```
--- Circular Linked List Menu ---
1. Insert
2. Delete
3. Display
4. Search
5. Exit
Enter your choice:
1
Enter value to insert: 89
89 inserted successfully.

--- Circular Linked List Menu ---
1. Insert
2. Delete
3. Display
4. Search
5. Exit
Enter your choice: 1
Enter value to insert: 45
45 inserted successfully.
```

# Experiment:-10

**Aim:-** Implementation of Bubble Sort.

**Objective:-**

- To understand the concept of comparison-based sorting algorithms.
- To learn how Bubble Sort works by repeatedly swapping adjacent elements.
- To implement and analyze the time complexity of Bubble Sort.

## Theory:-

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until the list becomes sorted.
In each pass, the largest element "bubbles up" to its correct position at the end of the list. The algorithm requires multiple passes through the list until no swaps are needed.

## Algorithm:-

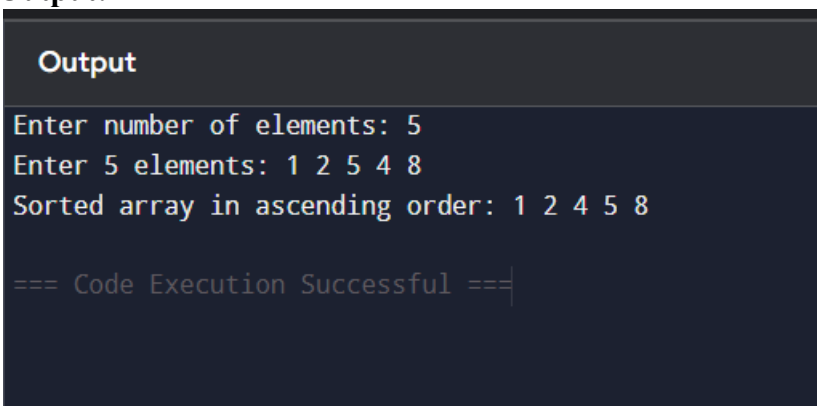
1. Start
2. Input the number of elements, n.
3. Input n elements into an array.
4. For i = 0 to n-1
    a. For j = 0 to n-i-2
        i. If arr[j] > arr[j+1], then swap(arr[j], arr[j+1])
5. Display the sorted array.
6. Stop

**Program:-**

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

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

   int arr[100];
   cout << "Enter " << n << " elements: ";
   for (int i = 0; i < n; i++) {
      cin >> arr[i];
   }
```

```
    // Bubble Sort
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    cout << "Sorted array in ascending order: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

**Output:-**




## Experiment:-11

**Aim:-**To write a program to implement Insertion Sort to sort a list of elements in ascending order.

**Objective:-**

- To understand the concept and working of the Insertion Sort algorithm.
- To implement Insertion Sort using C++.
- To analyze its time and space complexity.

**Theory:-** Insertion Sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is divided into a **sorted** and an **unsorted** part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

## Algorithm:-

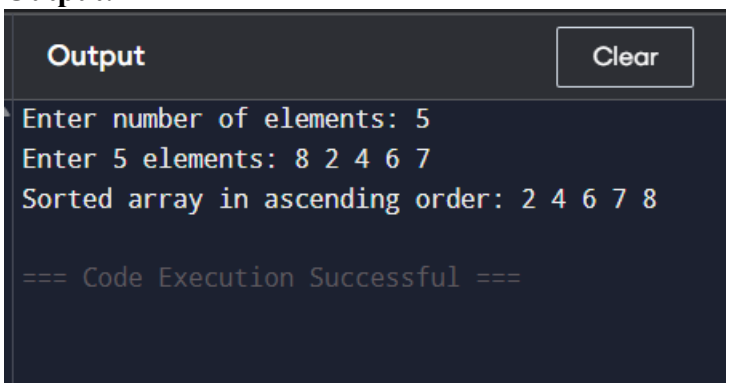
1. Start
2. Input the number of elements, n.
3. Input n elements into an array.
4. For i = 1 to n-1
    a. key = arr[i]
    b. j = i - 1
    c. While (j ≥ 0 and arr[j] > key)
        i. arr[j + 1] = arr[j]
        ii. j = j - 1
    d. arr[j + 1] = key
5. Display the sorted array.
6. Stop

**Program:-**

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

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

   int arr[100];
   cout << "Enter " << n << " elements: ";
   for (int i = 0; i < n; i++) {
      cin >> arr[i];
   }

   // Insertion Sort
   for (int i = 1; i < n; i++) {
      int key = arr[i];
      int j = i - 1;
```

```
    // Move elements greater than key to one position ahead
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
}

cout << "Sorted array in ascending order: ";
for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
}

return 0;
}
```

**Output:-**



**Experiment:-12**

**Aim:-** To write a program to implement Merge Sort to sort a list of elements in ascending order.

**Objective:-**

- To understand the concept of Divide and Conquer strategy in sorting.
- To implement Merge Sort algorithm using recursion.
- To analyze the time and space complexity of Merge Sort.

**Theory:-**

Merge Sort is a highly efficient sorting algorithm based on the divide and conquer technique. It divides the array into two halves, recursively sorts them, and then merges the sorted halves to produce the final sorted array.

Working Principle:

1. Divide the unsorted array into two halves.
2. Recursively sort both halves.
3. Merge the two sorted halves into a single sorted array.

**Algorithm:-**

1. Start
2. Divide the array into two halves.
3. Recursively sort the two halves.
4. Merge the two sorted halves into a single sorted array.
5. Continue merging until the entire array is sorted.
6. Display the sorted array.
7. Stop

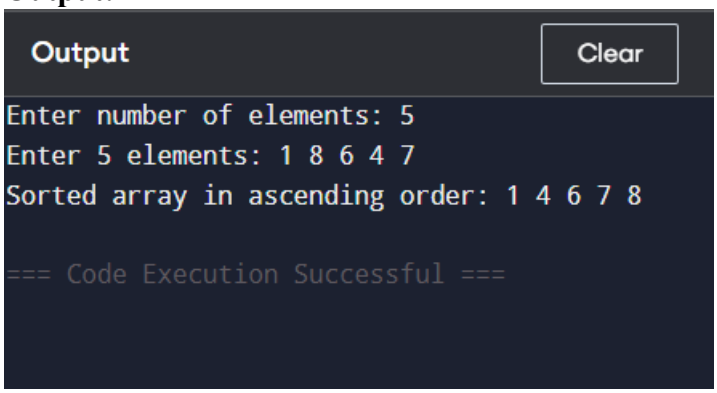
**Program:-**

```
#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;
   int n2 = right - mid;

   int L[100], R[100]; // 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];
```

```
    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++;
    }
}

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

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

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

```
    int arr[100];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array in ascending order: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

**Output:-**



**Experiment:-13**

**Aim:-** To write a program to implement Heap Sort in C++ to sort a list of elements in ascending order.

## Objective:-

- To understand the concept of heap data structure and its use in sorting.
- To implement Heap Sort using the max-heap approach.
- To analyze the time and space complexity of Heap Sort.

## Theory:-

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. A max heap is used to sort the array in ascending order — the largest element is repeatedly moved to the end of the array and then the heap is rebuilt for the remaining elements.

**Working Principle:**

1. Build a max heap from the input data.
2. The largest element (root of the heap) is placed at the end of the array.
3. Reduce the heap size and heapify the remaining elements.
4. Repeat the process until all elements are sorted.

**Algorithm:-**

1. Start
2. Build a max heap from the given array.
3. Swap the first (maximum) element with the last element of the heap.
4. Reduce the heap size by one and heapify the root.
5. Repeat step 3 and 4 until all elements are sorted.
6. Display the sorted array.
7. Stop

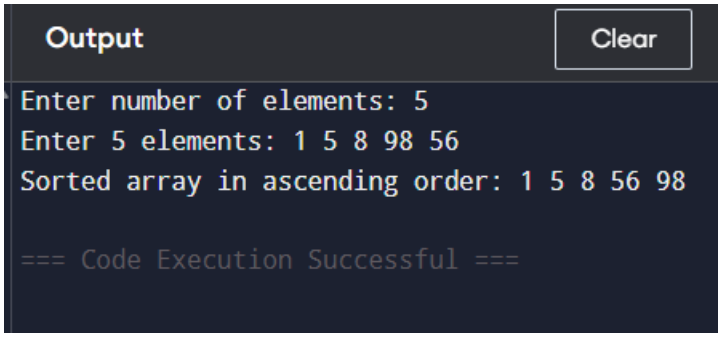
**Program:-**

```
#include <iostream>

using namespace std;

// Function to heapify a subtree rooted at index i

void heapify(int arr[], int n, int i) {

    int largest = i;       // Initialize largest as root
```

```
    int left = 2 * i + 1;  // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;
    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;
    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);}}
// Main function to perform Heap Sort
void heapSort(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // Call max heapify on the reduced heap
```

```
    heapify(arr, i, 0);
  }}
int main() {
  int n;
  cout << "Enter number of elements: ";
  cin >> n;
  int arr[100];
  cout << "Enter " << n << " elements: ";
  for (int i = 0; i < n; i++) {
    cin >> arr[i];
  }
  heapSort(arr, n);
  cout << "Sorted array in ascending order: ";
  for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }
  return 0;}
```

Output:-



**Experiment:-14**

**Aim:-** To write a program to implement Quick Sort in C++ to sort a list of elements in ascending order.

**Objective:-**

- To understand the concept of the Divide and Conquer strategy used in sorting.
- To implement Quick Sort algorithm using recursion.
- To analyze the time and space complexity of Quick Sort.

**Theory:-**

Quick Sort is one of the fastest sorting algorithms, based on the Divide and Conquer technique.It works by selecting a pivot element from the array and partitioning the remaining elements into two sub-arrays — one containing elements less than the pivot and the other containing elements greater than the pivot.
The same process is then recursively applied to the sub-arrays until the array is sorted.

**Working Principle:-**

1. Choose a pivot element (commonly the last element).
2. Partition the array so that elements smaller than the pivot come before it and greater ones come after.
3. Recursively apply the same logic to the left and right subarrays

**Algorithm:-**

1. Start
2. Choose a pivot element.
3. Rearrange the array so that elements less than pivot are on the left and greater on the right.
4. Recursively apply the same logic to left and right subarrays.
5. Combine the sorted subarrays.
6. Stop

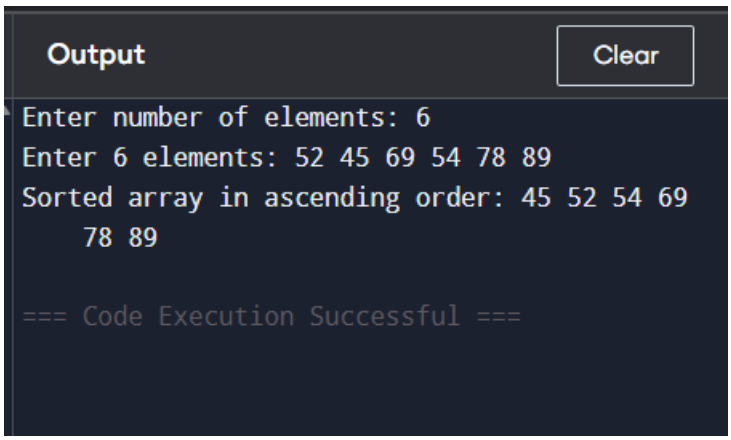
**Program:-** #include <iostream>

```
using namespace std;

// Function to swap two elements

void swapElements(int &a, int &b) {

   int temp = a;

   a = b;
```

```
    b = temp;
}
// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // choose the last element as pivot
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swapElements(arr[i], arr[j]);
        }
    }
    swapElements(arr[i + 1], arr[high]);
    return (i + 1);
}
// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);   // sort left part
        quickSort(arr, pi + 1, high);  // sort right part
    }
```

```
}
int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
   int arr[100];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    quickSort(arr, 0, n - 1);
    cout << "Sorted array in ascending order: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    } return 0;
}
```

**Output:-**



**Experiment:-15**

**Aim:-** To write a program construct a Binary Search Tree (BST) and perform deletion and inorder traversal operations on it.

## Objective:-

- To understand the concept and structure of a Binary Search Tree (BST).
- To perform insertion of nodes into a BST.
- To perform deletion of a node from the BST.
- To traverse the BST using Inorder Traversal

## Theory:-

A Binary Search Tree (BST) is a type of binary tree where each node has a key greater than all the keys in its left subtree and smaller than those in its right subtree.

**Properties of BST:**

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both left and right subtrees are also Binary Search Trees

## Algorithm:

**To Insert a Node:-**

1. If the tree is empty, create a new node as root.
2. If the key to insert is smaller than root, insert in left subtree.
3. If the key to insert is greater than root, insert in right subtree.
4. Repeat until correct position is found.

**To Delete a Node:-**

1. Find the node to delete.
2. If the node has no child → delete it.
3. If the node has one child → replace the node with its child.
4. If the node has two children → find its **inorder successor**, replace node's value with successor's value, and delete the successor.

**Inorder Traversal:-**

1. Traverse left subtree.
2. Visit root node.
3. Traverse right subtree.

**Program:-**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
};
// Function to create a new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = newNode->right = nullptr;
    return newNode;
}
// Function to insert a node in BST
Node* insert(Node* root, int value) {
    if (root == nullptr)
        return createNode(value);
    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
```

```
    return root;
}
// Function for inorder traversal
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}
// Function to find minimum node (used in deletion)
Node* findMin(Node* root) {
    while (root->left != nullptr)
        root = root->left;
    return root;
}
// Function to delete a node
Node* deleteNode(Node* root, int key) {
    if (root == nullptr)
        return root;
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
```

```
        root->right = deleteNode(root->right, key);
    else {
        // Node with one or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        // Node with two children
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
int main() {
    Node* root = nullptr;
    int n, val, del;
    cout << "Enter number of elements to insert in BST: ";
```

```
    cin >> n;
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> val;
        root = insert(root, val);
    }
    cout << "Inorder Traversal of BST: ";
    inorder(root);
    cout << "\nEnter element to delete: ";
    cin >> del;
    root = deleteNode(root, del);
    cout << "Inorder Traversal after deletion: ";
    inorder(root);
    return 0;
}
```

**Output:-**