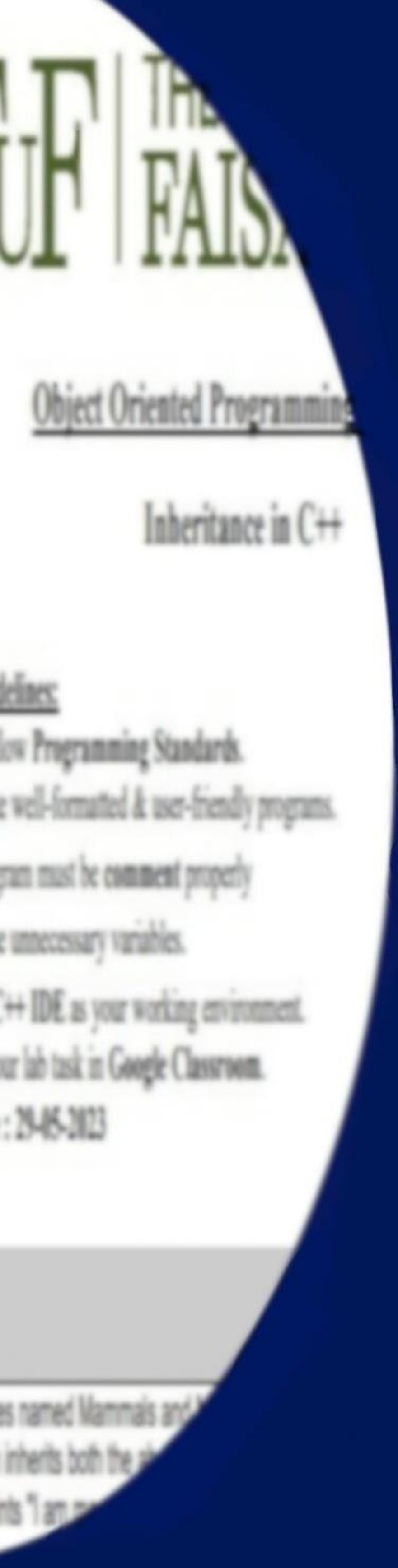
2023-2027

SUBMITTED TO

MISS IRSHA


UF | THE
FAIS

Object Oriented Programming

Inheritance in C++

delines:

low Programming Standards.

e well-formatted & user-friendly programs.

ram must be comment properly

e unnecessary variables.

C++ IDE as your working environment.

ur lab task in Google Classroom.

: 29-05-2023

es named Mammals and

inherits both the

nts "I am

LAB MANUAL

MISHAL NADEEM

2023-BS-AI-020

```
        show() {
      . "Base class" << endl
  
  
  erived : public Base {
  :
  void show() override {
      cout << "Derived class" << e
  }
;

int main() {
    Base *b;             // Base clas
    Derived d;           // Derived c
    b = &d;              // Base point
    b->show();           // Calls der
    return 0;
```

DATA STRUCTURES

INTRODUCTORY LAB

Data Structures

Data structures are ways of organizing, storing, and managing data so that it can be accessed and manipulated efficiently. They are fundamental to computer science and programming and are used to implement algorithms and handle data in various applications.

Linear Data Structures

1. **Array**
   - **Description**: A collection of elements, each identified by an index.
   - **Operations**: Access, search, insertion, deletion.
   - **Use Cases**: Storing data in contiguous memory for quick access.
2. **Linked List**
   - **Description**: A sequence of nodes where each node points to the next.
   - **Types**: Singly Linked List, Doubly Linked List, Circular Linked List.
   - **Use Cases**: Dynamic memory allocation, implementing stacks/queues.
3. **Stack**
   - **Description**: Follows Last In, First Out (LIFO) principle.
   - **Operations**: push (add), pop (remove), peek (view top).
   - **Use Cases**: Undo functionality, parsing expressions.
4. **Queue**
   - **Description**: Follows First In, First Out (FIFO) principle.
   - **Variants**: Circular Queue, Priority Queue, Deque.
   - **Use Cases**: Task scheduling, buffering.

Non-Linear Data Structures

1. **Tree**
   - **Description**: Hierarchical structure with a root and child nodes.
   - **Types**: Binary Tree, Binary Search Tree, AVL Tree, Heap, etc.

- **Use Cases**: Database indexing, file systems, AI decision trees.

2. **Graph**
   - **Description**: A set of nodes (vertices) connected by edges.
   - **Types**: Directed, Undirected, Weighted, Unweighted.
   - **Use Cases**: Networking, social media connections, route planning.

ARRAY

An **array** is a collection of elements, typically of the same data type, stored in contiguous memory locations. Arrays allow for fast and efficient access to their elements using an index.

Key Features of Arrays:

1. **Fixed Size**: The size of an array is determined at the time of its creation.
2. **Indexed Access**: Elements can be accessed directly using their index.
3. **Homogeneous Elements**: Arrays store elements of the same type.
4. **Efficient Memory Access**: As elements are stored in contiguous memory, accessing them is fast.

Types of Arrays:

1. **One-Dimensional Array**: A linear collection of elements. Example: [1, 2, 3, 4, 5]
2. **Multi-Dimensional Array**: Arrays containing arrays, such as 2D or 3D arrays. Example (2D Array):
3. **Dynamic Array**: A resizable array (e.g., Python's list or Java's ArrayList).

Array Operations:

1. **Access**: Retrieve an element using its index.
   - array[index] (e.g., arr[0] to access the first element).
2. **Insertion**: Add elements (in static arrays, this may require resizing).
3. **Deletion**: Remove an element; elements may need to shift.
4. **Traversal**: Loop through each element in the array.
5. **Search**: Find an element by its value.
6. **Sort**: Rearrange elements in a specific order.

Applications of Arrays:

1. **Data Storage**: Representing collections of data like scores, names, etc.
2. **Matrix Representation**: Storing data in rows and columns.
3. **Implementation of Other Data Structures**: Like stacks, queues, and heaps.
4. **Searching and Sorting**: Used in algorithms like binary search and quicksort.

LAB 1

Deletion at End

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

int main() {
    int arr[10] = {10, 20, 30, 40}; // Array with initial values
    int n = 4; // Number of elements

    // Deletion at end
    if (n > 0) {
        n--; // Just reduce the array size
    } else {
        cout << "Array is already empty!" << endl;
    }

    // Display the array
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

OUTPUT

```
10 20 30
```

Deletion at Mid

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

int main() {
    int arr[10] = {10, 20, 30, 40}; // Array with initial values
    int n = 4; // Number of elements
    int position = 1; // Index to delete from

    // Deletion at mid
    if (n > 0 && position >= 0 && position < n) {
        // Shift elements to the left
        for (int i = position; i < n - 1; i++) {
            arr[i] = arr[i + 1];
        }
        n--; // Decrement the array size
    } else {
        cout << "Invalid Position!" << endl;
    }

    // Display the array
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
   return 0;}
```

OUTPUT

```
10 30 40
```

Deletion at Start

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

int main() {
    int arr[10] = {10, 20, 30, 40}; // Array with initial values
    int n = 4; // Number of elements

    // Deletion at start
    if (n > 0) {
        // Shift elements to the left
        for (int i = 0; i < n - 1; i++) {
            arr[i] = arr[i + 1];
        }
        n--; // Decrement the array size
    } else {
        cout << "Array is already empty!" << endl;
    }

    // Display the array
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
   return 0;
}
```

OUTPUT

```
20 30 40
```

LAB 2

Index by value

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

int main() {
    int arr[] = {10, 20, 30, 20, 40, 20, 50}; // Array with multiple occurrences of value
    int n = sizeof(arr) / sizeof(arr[0]); // Size of the array
    int value = 20; // Value to find

    // Find indices where the value occurs
    cout << "Value " << value << " found at indices: ";
    for (int i = 0; i < n; i++) {
        if (arr[i] == value) {
            cout << i << " ";
        }  }
    cout << endl;

    return 0;}
```

OUTPUT

```
Value 20 found at indices: 1 3 5
```

Find value

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

int main() {
    int arr[10] = {10, 20, 30, 40, 50}; // Array with initial values
    int n = 5; // Number of elements
    int value = 30; // Value to find
    int index = -1; // Initialize index as -1 to indicate not found by default
    // Loop through the array to find the value
    for (int i = 0; i < n; i++) {
        if (arr[i] == value) {
            index = i; // Store the index where the value is found
            break; // Stop the loop once the value is found}   }
    // Output the result
    if (index != -1) {
        cout << "Value " << value << " found at index: " << index << endl;
    } else {
        cout << "Value " << value << " not found in the array." << endl;  }
    return 0;}
```

OUTPUT

```
Value 60 not found in the array.
```

Insertion at End

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

int main() {
    int arr[10] = {10, 20, 30, 40}; // Array with initial values
    int n = 4; // Number of elements
    int value = 50; // Value to be inserted at the end

    // Insertion at end
    arr[n] = value; // Insert the new value at the end
    n++; // Increment the array size
    // Display the array
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;}
```

OUTPUT

```
10 20 30 40 50
```

Insertion at Mid

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

int main() {
    int arr[10] = {10, 20, 30, 40}; // Array with initial values
    int n = 4; // Number of elements
    int value = 25; // Value to be inserted
    int position = 2; // Index where to insert
    // Insertion at mid
    if (position >= 0 && position <= n) {
        // Shift elements to the right
        for (int i = n; i > position; i--) {
            arr[i] = arr[i - 1];       }
        // Insert the new value at the position
        arr[position] = value;
        n++; // Increment the array size
    } else {
        cout << "Invalid Position!" << endl;   }
    // Display the array
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;}
```

OUTPUT

```
10 20 25 30 40
```

LAB 3

Insertion at start

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

int main() {
    int arr[10] = {10, 20, 30, 40}; // Array with initial values
    int n = 4; // Number of elements
    int value = 5; // Value to be inserted at the start

    // Insertion at start
    // Shift all elements one position to the right
    for (int i = n; i > 0; i--) {
        arr[i] = arr[i - 1]; //shift i-1 to i  }
    // Insert the new value at the start
    arr[0] = value;
    n++; // Increment the array size
    // Display the array
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
return 0;}
```

OUTPUT

```
5 10 20 30 40
```

Sum matices

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

const int ROWS = 2; // Number of rows
const int COLS = 3; // Number of columns

void addMatrices(int a[ROWS][COLS], int b[ROWS][COLS], int result[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }}}
void printMatrix(int matrix[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            cout << matrix[i][j] << " ";  }
        cout << endl;}}

int main() {
    int a[ROWS][COLS] = {{1, 2, 3}, {4, 5, 6}};
    int b[ROWS][COLS] = {{7, 8, 9}, {10, 11, 12}};
    int result[ROWS][COLS];
    addMatrices(a, b, result);
    cout << "Matrix A:\n";
    printMatrix(a);
    cout << "Matrix B:\n";
    printMatrix(b);
    cout << "Sum of Matrices:\n";
    printMatrix(result);
    return 0;}
```

OUTPUT

```
Matrix A:

1 2 3

4 5 6

Matrix B:

7 8 9

10 11 12

Sum of Matrices:

8 10 12

14 16 18
```

Value by index

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

int main() {
    int arr[10] = {10, 20, 30, 40, 50}; // Array with initial values
    int n = 5; // Number of elements
    int index; // Index to find the value
   cout << "Enter the index (0 to " << n - 1 << "): ";
    cin >> index;
    // Check if the index is valid
    if (index >= 0 && index < n) {
        // Access the value at the specified index
        cout << "Value at index " << index << " is: " << arr[index] << endl;
    } else {
        cout << "Invalid index!" << endl;
    }
    return 0;}
```

OUTPUT

```
Enter the index (0 to 4): 2

Value at index 2 is: 30
```

LINKED LIST

A **linked list** is a linear data structure where elements (called nodes) are connected by pointers. Unlike arrays, linked lists do not use contiguous memory locations. Each node in a linked list contains:

1. **Data**: The value or information the node holds.
2. **Pointer/Reference**: A link to the next node in the sequence.

Types of Linked Lists:

1. **Singly Linked List**:
   - Each node points to the next node in the sequence.
   - The last node's pointer is null (or None in Python).
2. **Doubly Linked List**:
   - Each node has two pointers: one to the next node and one to the previous node.
3. **Circular Linked List**:
   - The last node points back to the first node, forming a loop.

Key Operations:

1. Insertion:

- **At the beginning**: Add a new node and point it to the current head.
- **At the end**: Traverse to the last node and update its pointer to the new node.
- **At a specific position**: Traverse to the desired position and adjust pointers to insert the node.

2. Deletion:

- **From the beginning**: Update the head to the next node.

- **From the end**: Traverse to the second-last node and update its pointer to null.
- **From a specific position**: Adjust pointers to bypass the node being removed.

3. Traversal:

- Start from the head and follow the pointers to visit each node

LAB 4

Insertion and Deletion in Circular Linked List

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

class Node
{
public:
    int val;
    Node *next;

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

class CircularLinkedList
{
public:
    Node *head;
    CircularLinkedList()
    {
        head = nullptr;
    }
    // INSERTIONS----------------------------------
    void insertAtHead(int val)
    {
        Node *n = new Node(val);
        if (head == nullptr)
        {
            head = n;
            n->next = head; // same thing n->next=n;   //cll
            return;
        }
        Node *tail = head;
        while (tail->next != head)
        {
            tail = tail->next;
        }
        // tail is pointing to the last node
        tail->next = n;
        n->next = head;
        head = n;
    }
    void insertAtTail(int val)
    {
        Node *n = new Node(val);
        if (head == nullptr)
        {
            head = n;
            n->next = head; // same thing n->next=n;   //cll
            return;
        }
        Node *tail = head;
        while (tail->next != head)
        {
            tail = tail->next;
        }
        // tail is pointing to the last node
        tail->next = n;
        n->next = head;
        tail = n;
    }
    // INSERTION AT POSITION REMAINS SAME AS SINGLY LINKED LIST

    // DELETIONS-----------------------------------

    void deleteAtHead()
    {

        if (head == nullptr)
        {
            return;
        }
        Node *temp = head;
        Node *tail = head;
        while (tail->next != head)
        {
            tail = tail->next;
        }
        // tail is pointing to the last node
        head = head->next;
        tail->next = head;
        delete temp;
    }
    void deleteAtTail()
    {
        if (head == nullptr)
        {
            return;
        }

        Node *tail = head;
        while (tail->next->next != head)
        {
            tail = tail->next;
        }
        // tail is pointing to the second last node

        Node *temp = tail->next; // to be deleted
        tail->next = head;
        delete temp;
    }

    // DELETION AT POSITION REMAINS SAME AS SINGLY LINKED LIST

    void display()
    {
        Node *temp = head;
        do
        {
            cout << temp->val << "->";
            temp = temp->next;
        } while (temp != head);
        cout << endl;
    }
    void printCircular()
    {
        Node *temp = head;
        for (int i = 0; i < 15; i++)
        {
            cout << temp->val << "->";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main()
{
    CircularLinkedList cll;
    cll.insertAtHead(2);
    cll.display();
    cll.insertAtHead(1);
    cll.display();

    // cll.printCircular();
    // cll.display();

    cll.insertAtTail(7);
    cll.display();

    cll.deleteAtHead();
    cll.display();

    cll.deleteAtTail();
    cll.display();

    return 0;
}
```

OUTPUT

```
2->

1->2->

1->2->7->

2->7->

7->
```

Insertion and Deletion in Doubly Linked List

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

class Node
{
public:
    int val;
    Node *prev;
    Node *next;

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

class DoublyLinkedList
{
public:
    Node *head;
    Node *tail;
```

```
DoublyLinkedList()
{
    head = nullptr;
    tail = nullptr;
}

void display()
{
    Node *temp = head;
    while (temp != nullptr)
    {
        cout << temp->val << "<->";
        temp = temp->next;
    }
    cout << endl;
}

// INSERTIONS----------------------------------
void insertAtHead(int val)
{ // we use & beacause we are doing changes in the ll
    Node *n = new Node(val);
    if (head == nullptr)
    {
        head = n;
        tail = n;
        return;
    }
    n->next = head;
    head->prev = n;
    head = n;
    return;
}
void insertAtEnd(int val)
{ // we use & beacause we are doing changes in the ll
    Node *n = new Node(val);
    if (tail == nullptr)
    {
        head = n;
        tail = n;
        return;
    }
    tail->next = n;
    n->prev = tail;
    tail = n;
    return;
}
void insertAtPosition(int val, int k)
{
    // assuming k is less than or equal to length of doubly linked list
    Node *temp = head;
    int count = 1;
    while (count < (k - 1))
    {
        temp = temp->next;
        count++;
    }
    // temp will be pointing to the node at (k-1) position
    Node *n = new Node(val);
    n->next = temp->next;
    temp->next = n;

    n->prev = temp;
    n->next->prev = n;
    return;
}

// DELETIONS----------------------------------
void deleteAtHead()
{
    if (head == nullptr)
    {
        return;
    }
    Node *temp = head;
    head = head->next;
    if (head != nullptr)
    { // if dll has only one node
        tail = nullptr;
    }
    else
```

```
        {
            head->prev = nullptr;
        }
        delete (temp);
        return;
    }

    void deleteAtPosition(int k)
    {

        // assuming k is less than or equal to length of doubly linked list
        Node *temp = head;
        int counter = 1;
        while (counter < k)
        {
            temp = temp->next;
            counter++;
        }
        // temp will be pointing to the node at k position
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
        delete temp;
        return;
    }
};

int main()
{
    DoublyLinkedList dll;

    dll.insertAtHead(6);
    dll.insertAtHead(2);
    dll.insertAtHead(7);
    dll.display();

    dll.insertAtEnd(8);
    dll.display();

    dll.insertAtPosition(58, 3);
    dll.display();
    dll.deleteAtHead();
    dll.display();

    // dll.deleteAtEnd();
    // dll.display();

    dll.deleteAtPosition(3);
    dll.display();
    return 0;}
```

OUTPUT

```
7<->2<->6<->

7<->2<->6<->8<->

7<->2<->58<->6<->8<->

2<->58<->6<->8<->

2<->58<->6<->
```

LAB 5

Insertion and Deletion in Singly Linked List

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

class Node
{
public:
    int val;
    Node *next;

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

//INSERTIONS----------------------------------
void insertAtHead(Node *&head, int val)
{ // we use & beacause we are doing changes in the ll
    Node *n = new Node(val);
    n->next = head;
    head = n;
}
void insertAtTail(Node *&head, int val)
{
    Node *n = new Node(val);
    Node *temp = head;
    while (temp->next != nullptr)
    {
        temp = temp->next;
    }
    // temp has reached last node
    temp->next = n;
}
void insertAtPosition(Node *&head, int val, int pos)
{
    if (pos == 0)
    {
        insertAtHead(head, val);
        return;
    }
    Node *n = new Node(val);
    Node *temp = head;
    int curr_pos = 0;
    while (curr_pos != pos - 1)
    {
        temp = temp->next;
        curr_pos++;
    }
    // temp is pointing to node at pos-1
    n->next = temp->next;
    temp->next = n;
}

//UPDATION-----------------------------------
void updateAtPosition(Node *&head, int val, int k)
{
    Node *temp = head;
    int curr_pos = 0;
    while (curr_pos != k)
    {
        temp = temp->next;
        curr_pos++;
    }
    // temp is pointing to k node
    temp->val = val;
}

//DELETIONS-----------------------------------
void deleteAtHead(Node *&head)
{ // we use & beacause we are doing changes in the ll
    Node *temp = head;
    head = head->next;
    delete (temp);
}
```

```
void deleteAtTail(Node *&head)
{
    Node *second_last = head;
    while (second_last->next->next != nullptr)
    {
        second_last = second_last->next;
    }
    Node *temp = second_last->next; // node to be deleted
    second_last->next = nullptr;
    delete (temp);
}

void deleteAtPosition(Node *&head, int pos)
{
    if (pos == 0)
    {
        deleteAtHead(head);
        return;
    }

    int curr_pos = 0;
    Node *prev = head;
    while (curr_pos != pos - 1)
    {
        prev = prev->next;
        curr_pos++;
    }
    // prev is pointing to node at pos-1
    Node *temp = prev->next; // node to be deleted
    prev->next = prev->next->next;
    delete (temp);
}

void display(Node *head)
{
    Node *temp = head;
    while (temp != nullptr)
    {
        cout << temp->val << "->";
        temp = temp->next;
    }
    cout << "null" << endl;}
int main()
{
    Node *head = nullptr;
    insertAtHead(head, 2);
    display(head);
    insertAtHead(head, 1);
    display(head);

    insertAtTail(head, 7);
    display(head);

    insertAtPosition(head, 55, 1);
    display(head);

    updateAtPosition(head, 77, 2);
    display(head);

    deleteAtPosition(head, 2);
    display(head);
    return 0;}
```

OUTPUT

```
2->null

1->2->null

1->2->7->null

1->55->2->7->null

1->55->77->7->null

1->55->7->null
```

QUEUE

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. This means the first element added to the queue is the first one to be removed.

Characteristics of a Queue:

1. **Enqueue**: Adding an element to the end of the queue.
2. **Dequeue**: Removing an element from the front of the queue.
3. **Front**: The element at the front of the queue.
4. **Rear**: The element at the rear (end) of the queue.
5. **FIFO Principle**: The first element added is the first one removed.

Types of Queues:

1. **Simple Queue**:
   - Basic queue with FIFO behavior.
2. **Circular Queue**:
   - The last position connects back to the first position, forming a circle.
   - Avoids wasted space in a static queue implementation.
3. **Priority Queue**:
   - Elements are dequeued based on priority, not just their order.
4. **Double-Ended Queue (Deque)**:
   - Elements can be added or removed from both ends.

Common Operations:

1. **Enqueue**: Add an element to the rear of the queue.

2. **Dequeue**: Remove an element from the front of the queue.
3. **Peek/Front**: View the element at the front without removing it.
4. **IsEmpty**: Check if the queue is empty.
5. **IsFull**: Check if the queue is full (for fixed-size queues).

Lab 6

Circular Queue

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

// Queue variables
int sz = 4;
int f = 0;
int r = 0;
int* arr = new int[sz]; // Allocate memory for the queue

// Function to check if the queue is empty
bool isEmpty() {
    return (r == f);
}

// Function to check if the queue is full
bool isFull() {
    return ((r + 1) % sz == f);
}

// Function to add an element to the queue
void enqueue(int val) {
    if (isFull()) {
        cout << "This Queue is full" << endl;
    } else {
        r = (r + 1) % sz;
        arr[r] = val;
        cout << "Enqueued element: " << val << endl;
    }
}

// Function to remove and return an element from the queue
int dequeue() {
    int a = -1;
    if (isEmpty()) {
        cout << "This Queue is empty" << endl;
    } else {
        f = (f + 1) % sz;
        a = arr[f];
    }
    return a;
}

int main() {
    // Enqueue a few elements
    enqueue(12);
    enqueue(15);
    enqueue(1);
    cout << "Dequeuing element " << dequeue() << endl;
    cout << "Dequeuing element " << dequeue() << endl;
    cout << "Dequeuing element " << dequeue() << endl;
    enqueue(45);
    enqueue(45);
    enqueue(45);

    if (isEmpty()) {
        cout << "Queue is empty" << endl;
    }
    if (isFull()) {
        cout << "Queue is full" << endl;
    }

    delete[] arr; // Free allocated memory
    return 0;
}
```

OUTPUT

```
Enqueued element: 12

Enqueued element: 15

Enqueued element: 1

Dequeuing element 12
Dequeuing element 15

Dequeuing element 1

Enqueued element: 45

Enqueued element: 45

Enqueued element: 45

Queue is full
```

Dequeue

```
//FIFO is not followed
//Insertion/Deletion from both Front and rear
#include <iostream>
using namespace std;

// Deque variables
int sz = 4;
int f = -1;  // Front pointer
int r = -1;  // Rear pointer
int* arr = new int[sz];  // Allocate memory for the deque

// Function to check if the deque is empty
bool isEmpty() {
   return f == -1;  // Deque is empty if front is -1
}

// Function to check if the deque is full
bool isFull() {
   return (f == 0 && r == sz - 1) || (r == f - 1); // Deque is full if front and rear are at max
capacity
}

// Function to enqueue an element at the front of the deque
void enqueueFront(int val) {
   if (isFull()) {
        cout << "Deque is full. Cannot enqueue at front." << endl;
    } else {
        if (f == -1) {  // If the deque is empty
            f = r = 0;
        } else if (f == 0) {  // If the front is at the first position, wrap around
            f = sz - 1;
        } else {
            f--;  // Move front pointer to the left
        }
        arr[f] = val;
        cout << "Enqueued element " << val << " at front." << endl;
    }
}

// Function to enqueue an element at the rear of the deque
void enqueueRear(int val) {
    if (isFull()) {
        cout << "Deque is full. Cannot enqueue at rear." << endl;
    } else {
        if (f == -1) {  // If the deque is empty
            f = r = 0;
        } else if (r == sz - 1) {  // If rear is at the last position, wrap around
            r = 0;
        } else {
            r++;  // Move rear pointer to the right
        }
        arr[r] = val;
        cout << "Enqueued element " << val << " at rear." << endl;
    }
}

// Function to dequeue an element from the front of the deque
int dequeueFront() {
    int val = -1;
    if (isEmpty()) {
        cout << "Deque is empty. Cannot dequeue from front." << endl;
    } else {
        val = arr[f];
        if (f == r) {  // If there is only one element left
            f = r = -1;  // Reset the deque to empty state
        } else if (f == sz - 1) {  // Wrap around when front reaches the end
            f = 0;
        } else {
            f++;  // Move front pointer to the right
        }
        cout << "Dequeued element " << val << " from front." << endl;
    }
    return val;
}

// Function to dequeue an element from the rear of the deque
int dequeueRear() {
    int val = -1;
    if (isEmpty()) {
        cout << "Deque is empty. Cannot dequeue from rear." << endl;
    } else {
        val = arr[r];
        if (f == r) {  // If there is only one element left
            f = r = -1;  // Reset the deque to empty state
        } else if (r == 0) {  // Wrap around when rear reaches the start
            r = sz - 1;
        } else {
            r--;  // Move rear pointer to the left
        }
        cout << "Dequeued element " << val << " from rear." << endl;
    }
    return val;
}

// Function to print the deque elements
void printDeque() {
    if (isEmpty()) {
        cout << "Deque is empty." << endl;
        return;
    }
    cout << "Deque elements: ";
    int i = f;
    while (true) {
        cout << arr[i] << " ";
        if (i == r) break;
        i = (i + 1) % sz;
    }
    cout << endl;
}

int main() {
    // Enqueue elements at the rear
    enqueueRear(12);
    enqueueRear(15);
    enqueueRear(1);

    // Enqueue elements at the front
    enqueueFront(10);
    enqueueFront(5);

    // Print the deque
    printDeque();

    // Dequeue elements from the front
    dequeueFront();
    dequeueFront();

    // Print the deque after dequeuing from the front
    printDeque();

    // Dequeue elements from the rear
    dequeueRear();
    dequeueRear();

    // Print the deque after dequeuing from the rear
    printDeque();

    // Check if the deque is empty or full
    if (isEmpty()) {
        cout << "Deque is empty" << endl;
    }
    if (isFull()) {
        cout << "Deque is full" << endl;
    }
    // Free allocated memory
    delete[] arr;

    return 0;
}
```

OUTPUT

```
Enqueued element 12 at rear.

Enqueued element 15 at rear.

Enqueued element 1 at rear.

Enqueued element 10 at front.

Enqueued element 5 at front.

Deque elements: 5 10 12 15 1

Dequeued element 5 from front.

Dequeued element 10 from front.

Deque elements: 12 15 1

Dequeued element 1 from rear.

Dequeued element 15 from rear.

Deque elements: 12

Deque is empty
```

Priority Queue

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

// Priority Queue variables
int sz = 4;
int f = -1;  // Front pointer
int r = -1;  // Rear pointer
int* arr = new int[sz];  // Allocate memory for the queue

// Function to check if the queue is empty
bool isEmpty() {
    return f == -1 || f > r;
}

// Function to check if the queue is full
bool isFull() {
    return r == sz - 1;
}

// Function to enqueue an element into the priority queue
void enqueue(int val) {
    if (isFull()) {
        cout << "This Priority Queue is full" << endl;
    } else {
        if (f == -1) {
            f = r = 0;  // If the queue is empty, initialize front and rear
            arr[r] = val;
        } else {
            // Insert the element while maintaining the sorted order
            int i = r;
            while (i >= f && arr[i] > val) { // Move elements that are smaller than 'val'
                arr[i + 1] = arr[i];
                i--;
            }
            arr[i + 1] = val;  // Insert the new element at the correct position
            r++;  // Increment rear pointer
        }
        cout << "Enqueued element: " << val << endl;
    }
}

// Function to dequeue an element from the priority queue
int dequeue() {
    int val = -1;
    if (isEmpty()) {
        cout << "This Priority Queue is empty" << endl;
    } else {
        val = arr[f];  // Remove the highest priority element (front of the queue)
        f++;  // Increment front pointer
        if (f > r) {  // Reset the queue when empty
            f = r = -1;
        }
    }
    return val;
}

int main() {
    // Enqueue elements into the priority queue
    enqueue(12);
    enqueue(15);
    enqueue(1);

    cout << "Dequeuing element: " << dequeue() << endl;  // Dequeue highest priority
(15)
    cout << "Dequeuing element: " << dequeue() << endl;  // Dequeue next highest
priority (12)
    cout << "Dequeuing element: " << dequeue() << endl;  // Dequeue last element (1)

    enqueue(45);
    enqueue(30);
    enqueue(10);

    // Check if the queue is empty or full
    if (isEmpty()) {
        cout << "Priority Queue is empty" << endl;
    }
    if (isFull()) {
        cout << "Priority Queue is full" << endl;
    }

    // Dequeue elements to verify the priority order
    cout << "Dequeuing element: " << dequeue() << endl;  // Dequeue 45
    cout << "Dequeuing element: " << dequeue() << endl;  // Dequeue 30
    cout << "Dequeuing element: " << dequeue() << endl;  // Dequeue 10

    // Free allocated memory
    delete[] arr;
    return 0;
}
```

OUTPUT

```
Enqueued element: 12

Enqueued element: 15

Enqueued element: 1

Dequeuing element: 15

Dequeuing element: 12

Dequeuing element: 1

Enqueued element: 45

Enqueued element: 30

Enqueued element: 10

Priority Queue is full

Dequeuing element: 45

Dequeuing element: 30

Dequeuing element: 10
```

Lab 7

Queue

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

// Queue variables
int sz = 4;
int f = -1;
int r = -1;
int* qu = new int[sz]; // Allocate memory for the queue

// Function to check if the queue is empty
bool isEmpty() {
    return f == -1 || f > r;
}

// Function to check if the queue is full
bool isFull() {
    return r == sz - 1;
}

// Function to enqueue an element into the queue
void enqueue(int val) {
    if (isFull()) {
        cout << "This Queue is full" << endl;
    } else {
        if (f == -1) {
            f = 0; // Set front to 0 if it's the first element
        }
        r++;
        qu[r] = val;
        cout << "Enqueued element: " << val << endl;   }
}
// Function to dequeue an element from the queue
int dequeue() {
    int a = -1;
    if (isEmpty()) {
        cout << "This Queue is empty" << endl;
    } else {
        a = qu[f]; // Set 'a' to the front element in the queue
        f++;        // Increment the front pointer to remove the element
        if (f > r) { // Reset queue when empty
            f = r = -1;
        }
    }
    return a;
}

// Function to peek at the front element without removing it
int peek() {
    if (isEmpty()) {
        cout << "Queue is empty, nothing to peek." << endl;
        return -1; // Return -1 to indicate the queue is empty
    } else {
        return qu[f]; // Return the front element without dequeuing
    }
}

int main() {
    // Enqueue a few elements
    enqueue(12);
    enqueue(15);
    enqueue(1);

    // Peek at the front element
    cout << "Peek at front element: " << peek() << endl;

    // Dequeue a few elements
    cout << "Dequeuing element " << dequeue() << endl;
    cout << "Dequeuing element " << dequeue() << endl;

    // Peek again after dequeuing
    cout << "Peek at front element: " << peek() << endl;

    // Dequeue the last element
    cout << "Dequeuing element " << dequeue() << endl;
    // Check if the queue is empty or full
    if (isEmpty()) {
        cout << "Queue is empty" << endl;
    }
    if (isFull()) {
        cout << "Queue is full" << endl;
    }
    // Enqueue more elements
    enqueue(45);
    enqueue(50);
    // Check the front element after new enqueue
    cout << "Peek at front element after enqueue: " << peek() << endl;
    // Free allocated memory
    delete[] qu;
    return 0;
}
```

OUTPUT

```
Enqueued element: 12

Enqueued element: 15

Enqueued element: 1

Peek at front element: 12

Dequeuing element 12

Dequeuing element 15

Peek at front element: 1

Dequeuing element 1

Queue is empty

Queue is full

Enqueued element: 45

Enqueued element: 50

Peek at front element after enqueue: 1
```

Queue using Linked List

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

struct Node {
   int data;
   Node* link;
};

Node* f = nullptr; // Front of the queue
Node* r = nullptr; // Rear of the queue

// Function to traverse the linked list
void linkedListTraversal(Node* ptr) {
   cout << "Printing the elements of this linked list\n";
   while (ptr != nullptr) {
      cout << "Element: " << ptr->data << endl;
      ptr = ptr->link;}}
// Function to enqueue an element
void enqueue(int val) {
   Node* n = new Node; // Allocate memory for new node
   if (n == nullptr) {
      cout << "Queue is Full" << endl;
   } else {
      n->data = val;
      n->link = nullptr;
      if (f == nullptr) { // If the queue is empty
         f = r = n; // Both front and rear point to the new node
      } else {
         r->link = n; // Link the old rear to the new node
         r = n; // Update rear to point to the new node
   } }}
// Function to dequeue an element
int dequeue() {
   int val = -1;
   if (f == nullptr) { // If the queue is empty
      cout << "Queue is Empty" << endl;
   } else {
      Node* ptr = f; // Pointer to the front node
```

```
        f = f->link; // Move front to the link node
        val = ptr->data; // Get the value of the dequeued node
        delete ptr; // Free the memory of the dequeued node
    }
    return val;
}
int main() {
    linkedListTraversal(f); // Traverse the empty queue
    cout << "Dequeuing element " << dequeue() << endl; // Try to dequeue from empty
    // Enqueue some elements
    enqueue(34);
    enqueue(4);
    enqueue(7);
    enqueue(17);
    // Dequeue elements
    cout << "Dequeuing element " << dequeue() << endl;
    cout << "Dequeuing element " << dequeue() << endl;
    cout << "Dequeuing element " << dequeue() << endl;
    cout << "Dequeuing element " << dequeue() << endl;
    linkedListTraversal(f); // Traverse the queue after dequeuing
    return 0;}
```

OUTPUT

```
Printing the elements of this linked list

Queue is Empty

Dequeuing element -1

Enqueued element: 34

Enqueued element: 17

Dequeuing element 34

Dequeuing element 4

Printing the elements of this linked list
```

STACK

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means the last element added to the stack is the first one to be removed.

Characteristics of a Stack:

1. **Push**: Adding an element to the top of the stack.
2. **Pop**: Removing the top element from the stack.
3. **Peek/Top**: Viewing the top element without removing it.
4. **LIFO Principle**: The last element added is the first one removed.

Real-Life Analogy:

Think of a stack as a pile of plates:

- You add (push) plates on top of the stack.
- You remove (pop) plates from the top.

Common Operations:

1. **Push**: Insert an element at the top of the stack.
2. **Pop**: Remove and return the top element.
3. **Peek/Top**: Get the value of the top element without removing it.
4. **IsEmpty**: Check if the stack is empty.

Lab 8

Infix to Postfix

```
#include <iostream>
#include <cstring>

// Stack size and variables
int size;
int top;
char* st;

// Function to initialize the stack
void initializeStack(int stackSize) {
   size = stackSize;
   top = -1;
   st = new char[size];
}

// Function to delete the stack
void deleteStack() {
   delete[] st;
}

// Function to get the top element of the stack
char stackTop() {
   return st[top];
}

// Function to check if the stack is empty
bool isEmpty() {
   return top == -1;
}

// Function to check if the stack is full
bool isFull() {
   return top == size - 1;
}
```

```
// Function to push an element onto the stack
void push(char val) {
    if (isFull()) {
        std::cout << "Stack Overflow! Cannot push " << val << " to the stack" << std::endl;
    } else {
        top++;
        st[top] = val;
    }
}

// Function to pop an element from the stack
char pop() {
    if (isEmpty()) {
        std::cout << "Stack Underflow! Cannot pop from the stack" << std::endl;
        return -1; // Return -1 for error indication
    } else {
        char val = st[top];
        top--;
        return val;
    }
}

// Function to determine the precedence of operators
int precedence(char ch) {
    if (ch == '*' || ch == '/') {
        return 3;
    } else if (ch == '+' || ch == '-') {
        return 2;
    } else {
        return 0;
    }
}

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Function to convert infix expression to postfix
char* infixToPostfix(const char* infix) {
    int stackSize = strlen(infix);
    initializeStack(stackSize);

    char* postfix = new char[strlen(infix) + 1];
    int i = 0;  // Track infix traversal
    int j = 0;  // Track postfix addition

    while (infix[i] != '\0') {
        if (!isOperator(infix[i])) {
            postfix[j++] = infix[i]; // Add operand to postfix
            i++;
        } else {
            while (!isEmpty() && precedence(infix[i]) <= precedence(stackTop())) {
                postfix[j++] = pop(); // Pop from stack to postfix
            }
            push(infix[i]); // Push operator to stack
            i++;
      }    }
    while (!isEmpty()) {
        postfix[j++] = pop(); // Pop remaining elements
    }
    postfix[j] = '\0'; // Null-terminate the string

    deleteStack(); // Free memory allocated for stack stay
    return postfix;}

int main() {
    const char* infix = "x-y/z-k*d"; // Infix expression
    char* postfix = infixToPostfix(infix);
    std::cout << "Postfix is " << postfix << std::endl;

    delete[] postfix; // Free allocated memory for postfix
    return 0;
}
```

OUTPUT

```
Postfix is xyz/-kd*-
```

Multiple Parenthesis

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

int stackSize;
int stackTop;
char* stackArr;

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

bool isFull() {
   return stackTop == stackSize - 1;
}

void push(char val) {
   if (isFull()) {
      cout << "Stack Overflow! Cannot push " << val << " to the stack" << endl;
   } else {
      stackTop++;
      stackArr[stackTop] = val;}}
char pop() {
   if (isEmpty()) {
      cout << "Stack Underflow! Cannot pop from the stack" << endl;
      return -1;
   } else {
      char val = stackArr[stackTop];
      stackTop--;
      return val;
   }}
char stackTopElement() {
   return stackArr[stackTop];}
bool match(char a, char b) {
   return (a == '{' && b == '}') || (a == '(' && b == ')') || (a == '[' && b == ']');
}
bool parenthesisMatch(const char* exp) {
   stackSize = 100;
```

```
    stackTop = -1;
    stackArr = new char[stackSize];

    for (int i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(' || exp[i] == '{' || exp[i] == '[') {
            push(exp[i]);
        } else if (exp[i] == ')' || exp[i] == '}' || exp[i] == ']') {
            if (isEmpty()) {
                delete[] stackArr;
                return false;
            }
            char popped_ch = pop();
            if (!match(popped_ch, exp[i])) {
                delete[] stackArr;
                return false;
            }
        }
    }

    bool isBalanced = isEmpty();
    delete[] stackArr;
    return isBalanced;
}

int main() {
    const char* exp = "[4-6]((8){(9-8)})";

    if (parenthesisMatch(exp)) {
        cout << "The parenthesis is balanced" << endl;
    } else {
        cout << "The parenthesis is not balanced" << endl;
    }
    return 0;
}
```

OUTPUT

```
The parenthesis is balanced
```

Parenthesis Matching

//<cstring> library in C++ provides functions for manipulating C-style strings (null-terminated character stays) and dealing with memory management and string operations. It's an essential library when you're working with raw character arrays and need to perform common string operations like copying, comparing, or finding lengths.

//<cctype> library in C++ provides functions for character classification and manipulation. These functions allow you to test whether a character is of a certain type (e.g., digit, letter, whitespace) or to convert characters to different cases (uppercase, lowercase).

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

bool isEmpty(int top) {
   return top == -1;
}

bool isFull(int top, int size) {
   return top == size - 1;}
void push(char* st, int& top, int size, char val) {
   if (isFull(top, size)) {
      cout << "Stack Overflow! Cannot push " << val << " to the stack" << endl;
   } else {
      top++;
      st[top] = val;
   }
}

char pop(char* st, int& top) {
   if (isEmpty(top)) {
      cout << "Stack Underflow! Cannot pop from the stack" << endl;
      return -1;
   } else {
      char val = st[top];
      top--;
      return val;
   }
```

```
}

bool parenthesisMatch(const char* exp) {
    int size = 100;
    int top = -1;
    char* st = new char[size];

    for (int i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(') {
            push(st, top, size, '(');
        } else if (exp[i] == ')') {
            if (isEmpty(top)) {
                //this line deletes the dynamically allocated memory for the stack (st).
                //It also returns false, indicating that the expression is unbalanced (i.e., there
is a closing parenthesis without a corresponding opening parenthesis).
                delete[] st;
                return false;
            }
            pop(st, top);
        }}
    bool isMatched = isEmpty(top);
    delete[] st;
    return isMatched;
}
int main() {
    const char* exp = "((8)(*--$$9))";
    if (parenthesisMatch(exp)) {
        cout << "The parenthesis is matching" << endl;
    } else {
        cout << "The parenthesis is not matching" << endl;
    }
    return 0;
}
```

OUTPUT

```
The parenthesis is matching
```

Lab 9

Postfix Evaluation

```
#include <iostream>
#include <cstring>
#include <cctype>
#include <cmath>

// Stack size and variables
int size;
int top;
int* arr;

// Function to initialize the stack
void initializeStack(int stackSize) {
    size = stackSize;
    top = -1;
    arr = new int[size];
}

// Function to delete the stack
void deleteStack() {
    delete[] arr;
}

// Function to push an element onto the stack
void push(int val) {
    if (top == size - 1) {
        std::cout << "Stack Overflow! Cannot push " << val << " to the stack" << std::endl;
    } else {
        arr[++top] = val;
    }
}

// Function to pop an element from the stack
int pop() {
    if (top == -1) {
        std::cout << "Stack Underflow! Cannot pop from the stack" << std::endl;
        return -1;  // Return -1 for error indication
    } else {
        return arr[top--];
    }
}

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Function to perform arithmetic operation
int applyOperator(int left, int right, char op) {
    switch (op) {
        case '+': return left + right;
        case '-': return left - right;
        case '*': return left * right;
        case '/': return left / right;
        default: return 0;
    }
}

// Function to evaluate the postfix expression
int evaluatePostfix(const char* postfix) {
    int i = 0;
    while (postfix[i] != '\0') {
        // If the current character is a digit, push it onto the stack
        if (isdigit(postfix[i])) {
            push(postfix[i] - '0'); // Convert char to int and push
        }
        // If the current character is an operator
        else if (isOperator(postfix[i])) {
            int right = pop(); // Pop right operand
            int left = pop();  // Pop left operand

            int result = applyOperator(left, right, postfix[i]);
            push(result); // Push result back onto the stack
        }
        i++;
    }
    return pop(); // The result will be the only item left in the stack
}

int main() {
    const char* postfix = "23*54*+9-"; // Postfix expression (should evaluate to 17)

    int stackSize = strlen(postfix);  // The stack size can be the size of the postfix
expression
    initializeStack(stackSize);

    int result = evaluatePostfix(postfix);
    std::cout << "Result of postfix evaluation: " << result << std::endl;

    deleteStack(); // Free memory allocated for the stack

    return 0;
}
```

OUTPUT

```
Result of postfix evaluation: 17
```

Stack Operations

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

int sz = 50;
int top = -1;
int* st = new int[sz];

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

bool isFull() {
    return top == sz - 1;
}

void push(int val) {
    if (isFull()) {
        cout << "Stack Overflow! Cannot push " << val << " to the stack" << endl;
    } else {
        top++;
        st[top] = val;
    }
}

int pop() {
    if (isEmpty()) {
        cout << "Stack Underflow! Cannot pop from the stack" << endl;
        return -1;
    } else {
        int val = st[top];
        top--;
        return val;
    }
}

int peek(int i) {
    int stayInd = top - i + 1;
    if (stayInd < 0) {
        cout << "Not a valid position for the stack" << endl;
        return -1;
    } else {
        return st[stayInd];
    }
}

int stackTop() {
    return st[top];
}

int stackBottom() {
    return st[0];
}
```

```
int main() {
    cout << "Stack has been created successfully" << endl;

    cout << "Before pushing, Full: " << isFull() << endl;
    cout << "Before pushing, Empty: " << isEmpty() << endl;

    // Pushing values onto the stack
    push(1);
    push(23);
    push(99);
    push(75);
    push(3);
    push(64);
    push(57);
    push(46);
    push(89);
    push(6);
    push(5);
    push(75);

    // Printing values from the stack
    for (int j = 1; j <= top + 1; j++) {
        cout << "The value at position " << j << " is " << peek(j) << endl;
    }
    cout<<"size: "<<top;

    // Clean up
    delete[] st;

    return 0;
}
```

OUTPUT

```
Stack has been created successfully

Before pushing, Full: 0

Before pushing, Empty: 1
```

```
The value at position 1 is 1

The value at position 2 is 23

The value at position 3 is 99

The value at position 4 is 75

The value at position 5 is 3

The value at position 6 is 64

The value at position 7 is 57

The value at position 8 is 46

The value at position 9 is 89

The value at position 10 is 6

The value at position 11 is 5

The value at position 12 is 75

size: 12
```

Stack using Linked List

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

struct Node {
    int data;
    Node* link;
};

Node* top = nullptr;

void linkedListTraversal(Node* ptr) {
    while (ptr != nullptr) {
        cout << "Element: " << ptr->data << endl;
        ptr = ptr->link;
    }
}

bool isEmpty(Node* top) {
    return top == nullptr;
}

bool isFull(Node* top) {
    Node* p = new(nothrow) Node;
    if (p == nullptr) {
        return true;
    } else {
        delete p;
        return false;
    }
}

Node* push(Node* top, int x) {
    if (isFull(top)) {
        cout << "Stack Overflow" << endl;
    } else {
        Node* n = new Node;
        n->data = x;
        n->link = top;
        top = n;
    }
    return top;
}

int pop() {
    if (isEmpty(top)) {
        cout << "Stack Underflow" << endl;
        return -1;
    } else {
        Node* n = top;
        top = top->link;
        int x = n->data;
        delete n;
        return x;
    }}

// Peek method to view the top element without removing it
int peek() {
    if (isEmpty(top)) {
        cout << "Stack is empty" << endl;
        return -1; // or you can throw an exception here
    } else {
        return top->data; // Return the data of the top node
    }}
int main() {
    top = push(top, 78);
    top = push(top, 7);
    top = push(top, 8);

    // Peek the top element of the stack
    int peekElement = peek();
    if (peekElement != -1) {
        cout << "Top element is " << peekElement << endl;
    }
    // Pop the top element
    int element = pop();
    cout << "Popped element is " << element << endl;

    // Traverse the remaining stack
    linkedListTraversal(top);

    return 0;}
```

OUTPUT

```
Top element is 8

Popped element is 8

Element: 7

Element: 78
```

Tree

BINARY SEARCH TREES (BST)

DEFINITION

A Binary Search Tree is a hierarchical data structure in which each node has at most two children, referred to as the left and right child, and it satisfies the following **properties**:

- All nodes of the left subtree are lesser.
- All nodes of the right subtree are greater.
- Left and right subtrees are also BST.
- There are no duplicate nodes.
- In-order traversal of a BST gives an ascending sorted array.

APPLICATIONS OF BST

1. **Searching**: Efficient lookups in dynamic datasets.
2. **Sorting**: In-order traversal produces sorted order.
3. **Range Queries**: Quickly find all keys in a given range.
4. **Database Indexing**: For storing and retrieving records.

Lab 10

INSERTION AND DUPLICATE

```
#include <iostream>

using namespace std;

struct Node {

   int data;

   Node* left;

   Node* right;};

// Function to create a new node

Node* createNode(int data) {

   Node* n = new Node(); // Memory heap mein allocate kar rahe hain

   n->data = data;       // Data set kar rahe hain

   n->left = nullptr;    // Left child ko nullptr set kar rahe hain

   n->right = nullptr;   // Right child ko nullptr set kar rahe hain

   return n;    }        // New node return kar rahe hain

// In-order traversal

void inOrder(Node* root) {

   if (root != nullptr) {

      inOrder(root->left);           // Pehle left subtree traverse karen

      cout << root->data << " ";     // Phir current node ka data print karen

      inOrder(root->right);   }}     // Phir right subtree traverse karen

// Function to insert a new node in BST

void insert(Node* root, int key) {
    Node* prev = nullptr;             // Pehle ka pointer null set karen
    while (root != nullptr) {
        prev = root;                  // Current node ko prev set karen
        if (key == root->data) {      // Duplicate check karen
            cout << "Cannot insert " << key << ", already in BST" << endl;
            return;                   // Duplicate milne par function se return karen
        } else if (key < root->data) {
            root = root->left;        // Agar key chhoti hai, left subtree mein jaen
        } else {
            root = root->right;  }   }  // Agar key badi hai, right subtree mein jaen
    Node* newNode = createNode(key);  // New node create karen
    if (key < prev->data) {
        prev->left = newNode;         // Agar key chhoti hai, left child set karen
    } else {
        prev->right = newNode;  }}    // Agar key badi hai, right child set karen
int main() {
    // Root node banaye - Function ka istemal karke
    Node* p = createNode(40);
    Node* p1 = createNode(20);
    Node* p2 = createNode(60);
    Node* p3 = createNode(10);
    Node* p4 = createNode(30);
    Node* p5 = createNode(50);
    Node* p6 = createNode(70);
    // Root node ko left aur right children se link karein
```

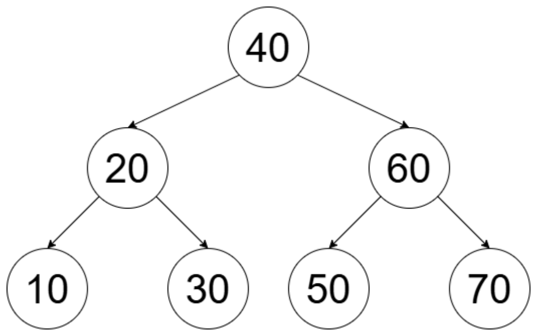
```
p->left = p1;
p->right = p2;
p1->left = p3;
p1->right = p4;
p2->left = p5;
p2->right = p6;

cout << "Before Insertion:" << endl;
inOrder(p);                 // Pehle in-order traversal karen
cout << "\n";
// Nayi node ko insert karen jiska value 16 hai
insert(p, 16);
cout << "After Insertion:" << endl;
inOrder(p);                 // Dobara in-order traversal karen
cout << "\n";
return 0;}
```

OUTPUT

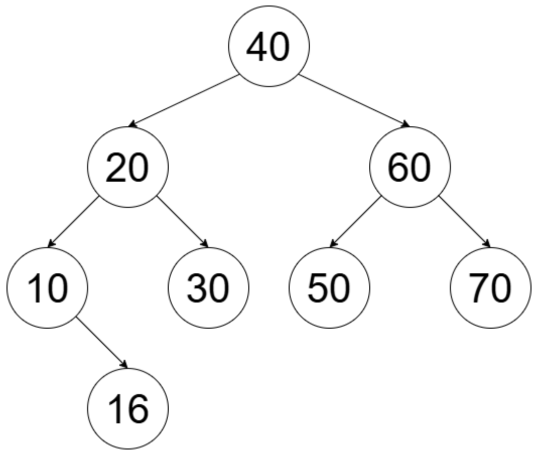
```
Before Insertion:
10 20 30 40 50 60 70
After Insertion:
10 16 20 30 40 50 60 70
```

VISUAL REPRESENTATION

BEFORE INSERTION



AFTER INSERTION



DELETION

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;};
Node* createNode(int data) {
    Node* n = new Node(); // Memory new keyword se allocate kar rahe hain
    n->data = data;       // Node ka data set kar rahe hain
    n->left = nullptr;    // Left child ko nullptr set karte hain
    n->right = nullptr;   // Right child ko nullptr set karte hain
    return n;  }          // Nayi node return karte hain
// Tree ki in-order traversal ka function
void inOrder(Node* root) {
    if (root != nullptr) {        // Agar tree empty nahi hai
        inOrder(root->left);      // Pehle left subtree traverse karo
        cout << root->data << " "; // Node ka data print karo
        inOrder(root->right);  }}   // Phir right subtree traverse karo
// In-order predecessor dhundhne ka function (left subtree ka sabse bara node)
Node* inOrderPredecessor(Node* root) {
    root = root->left;          // Left subtree par jao
    while (root->right != nullptr) { // Jab tak rightmost node na mil jaye
        root = root->right; }
    return root;   }            // Predecessor node return karo
// Tree se kisi node ko delete karne ka function
Node* deleteNode(Node* root, int value) {
    Node* iPre;                 // In-order predecessor ka pointer
    if (root == nullptr) {      // Agar tree empty hai
        return nullptr; }       // Return null
    // Leaf node ka case handle karo
    if (root->left == nullptr && root->right == nullptr) {
        delete root;            // Memory free karo
        return nullptr;}        // Return null
    if (value < root->data) {    // Agar value chhoti hai, left subtree mein jao
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) { // Agar value badi hai, right subtree mein jao
```

```
        root->right = deleteNode(root->right, value);}
    else {
        iPre = inOrderPredecessor(root); // In-order predecessor dhundo
        root->data = iPre->data;         // Predecessor ka data copy karo
        // Predecessor node ko recursively delete karo
        root->left = deleteNode(root->left, iPre->data);}
    return root;        }         // Updated root return karo
int main() {
    Node* p = createNode(12);
    Node* p1 = createNode(7);
    Node* p2 = createNode(15);
    Node* p3 = createNode(5);
    Node* p4 = createNode(9);
    Node* p5 = createNode(8);
    // Root node ko left aur right children ke saath link karo
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;
    p4->left = p5;
    // Delete karne se pehle in-order traversal print karo
    cout << "Before Deletion:" << endl;
    inOrder(p);
    cout << "\n";
    // Node jiska value 3 hai usko delete karo
```

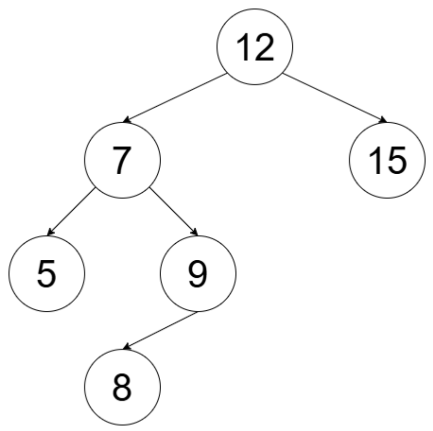
```
deleteNode(p, 3);

// Delete karne ke baad in-order traversal print karo

cout << "After Deletion:" << endl;

inOrder(p);

cout << "\n";

return 0;}
```

OUTPUT

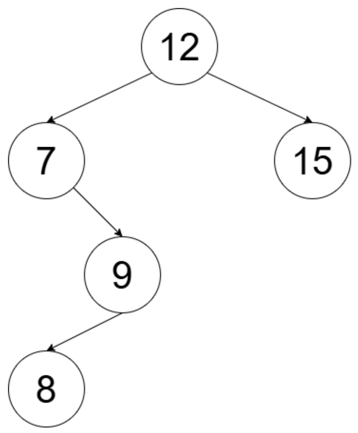
```
Before Deletion:

5 7 8 9 12 15

After Deletion:

7 8 9 12 15
```

VISUAL REPRESENTATION

BEFORE DELETION



AFTER DELETION



TRAVERSAL

```
#include <iostream>

using namespace std;

struct Node {

   int data;        // Node ka data store karne ke liye

   Node* left;       // Left child ka pointer

   Node* right;  };    // Right child ka pointer

// Nayi node banane ka function

Node* createNode(int data) {

   Node* n = new Node(); // Memory heap mein allocate karte hain

   n->data = data;       // Node ka data set karte hain

   n->left = nullptr;    // Left child ko nullptr set karte hain

   n->right = nullptr;   // Right child ko nullptr set karte hain

   return n;      }       // Nayi node return karte hain

// Pre-order traversal ka function

void preOrder(Node* root) {

   if (root != nullptr) {

      cout << root->data << " ";  // Root node ka data print karo

      preOrder(root->left);       // Left subtree ko traverse karo

      preOrder(root->right);   }}   // Right subtree ko traverse karo

// Post-order traversal ka function

void postOrder(Node* root) {

   if (root != nullptr) {

      postOrder(root->left);      // Left subtree ko traverse karo

      postOrder(root->right);     // Right subtree ko traverse karo
```

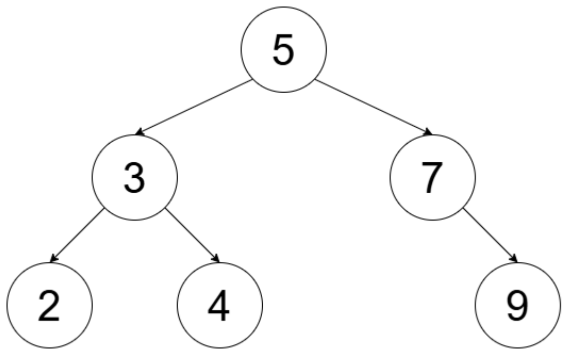
```
        cout << root->data << " "; }} // Root node ka data print karo
// In-order traversal ka function
void inOrder(Node* root) {
    if (root != nullptr) {
        inOrder(root->left);        // Left subtree ko traverse karo
        cout << root->data << " ";  // Root node ka data print karo
        inOrder(root->right);   }}    // Right subtree ko traverse karo
int main() {
    // Root aur uske children nodes banate hain
    Node* p = createNode(5);
    Node* p1 = createNode(3);
    Node* p2 = createNode(7);
    Node* p3 = createNode(2);
    Node* p4 = createNode(4);
    Node* p5 = createNode(9);
    // Tree ke nodes ko link karte hain
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;
    p2->right = p5;
    // In-order traversal ka output
    cout << "Inorder Traversal:" << endl;
    inOrder(p); // Root node se start karke tree ko traverse karo
    cout << "\n";
```

```
// Pre-order traversal ka output
cout << "Preorder Traversal:" << endl;
preOrder(p); // Root, Left, Right order mein traverse karo
cout << "\n";
// Post-order traversal ka output
cout << "Postorder Traversal:" << endl;
postOrder(p); // Left, Right, Root order mein traverse karo
cout << "\n";
// Memory cleanup
delete p4;
delete p3;
delete p2;
delete p1;
delete p;
return 0;} // Program successful execution
```

OUTPUT

```
Inorder Traversal:
2 3 4 5 7 9
Preorder Traversal:
5 3 2 4 7 9
Postorder Traversal:
2 4 3 9 7 5
```

VISUAL REPRESENTATION



SEARCH

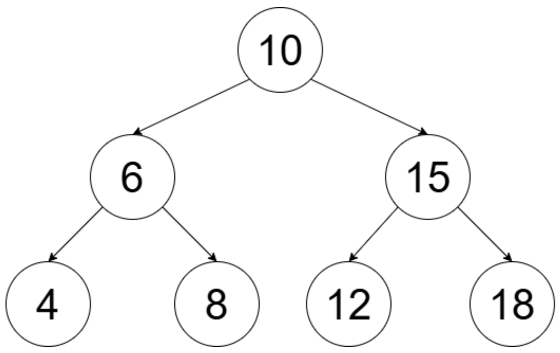
```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;};
// Nayi node banane ka function
Node* createNode(int data) {
    Node* n = new Node(); // Memory new keyword se allocate karte hain
    n->data = data;       // Node ka data set karte hain
    n->left = nullptr;    // Left child ko nullptr set karte hain
    n->right = nullptr;   // Right child ko nullptr set karte hain
    return n;   }         // Nayi node return karte hai
// BST mein key ko search karne ka function
Node* search(Node* root, int key) {
    if (root == nullptr) {  // Agar tree empty hai ya node nahi mili
        return nullptr; }    // Null return karo
    if (key == root->data) { // Agar key mil gayi
        return root;         // Node return karo
    } else if (key < root->data) { // Agar key chhoti hai, left subtree mein search
karo
        return search(root->left, key);
    } else {                 // Agar key badi hai, right subtree mein search karo
        return search(root->right, key);}}
int main() {
    // Root node banane ka kaam
    Node* p = createNode(10);
    Node* p1 = createNode(6);
    Node* p2 = createNode(15);
    Node* p3 = createNode(4);
    Node* p4 = createNode(8);
    Node* p5 = createNode(12);
    Node* p6 = createNode(18);
    // Root node ko left aur right children ke saath link karte hain
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;
    p2->left = p5;
    p2->right = p6;
```

```
// BST mein kisi value ko search karna
Node* n = search(p, 55); // Value 55 ko search karo
if (n != nullptr) {      // Agar node mil gayi
    cout << "Found: " << n->data << endl; // Node ka data print karo
} else {                 // Agar node nahi mili
    cout << "Element not found" << endl;} // Message display karo
// Memory cleanup ka kaam
delete p4;
delete p3;
delete p2;
delete p1;
delete p;
return 0;}
```

OUTPUT

```
Element not found
```

VISUAL REPRESENTATION


10
6
15
4
8
12
18