1. Implementing Linked Lists:

i. Implement singly linked lists with methods for insertion, deletion, and traversal

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

class SinglyLinkedList {
   struct Node {
      int data;
      Node* next;
      Node(int val) : data(val), next(nullptr) {}
   };
   Node* head;

public:
   SinglyLinkedList() : head(nullptr) {}

   void insert(int val) {
      Node* newNode = new Node(val);
      if (!head) {
         head = newNode;
      } else {
         Node* temp = head;
         while (temp->next) temp = temp->next;
         temp->next = newNode;
      }
   }

   void deleteNode(int val) {
      if (!head) return;
      if (head->data == val) {
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }
        Node* temp = head;
        while (temp->next && temp->next->data != val) temp = temp->next;
        if (temp->next) {
            Node* toDelete = temp->next;
            temp->next = toDelete->next;
            delete toDelete;
        }
    }

    void traverse() {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    SinglyLinkedList sll;
    sll.insert(1);
    sll.insert(2);
    sll.deleteNode(1);
    sll.traverse(); // Expected Output: 2
    return 0;
```

```
}
```

Output:

```
2
```

ii. Implement doubly linked lists with methods for insertion, deletion, and traversal.

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

class DoublyLinkedList {
    struct Node {
        int data;
        Node* next;
        Node* prev;
        Node(int val) : data(val), next(nullptr), prev(nullptr) {}
    };
    Node* head;

public:
    DoublyLinkedList() : head(nullptr) {}

    void insert(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next) temp = temp->next;
            temp->next = newNode;
            newNode->prev = temp;
        }
    }

    void deleteNode(int val) {
        if (!head) return;
        if (head->data == val) {
            Node* temp = head;
            head = head->next;
            if (head) head->prev = nullptr;
            delete temp;
            return;
        }
        Node* temp = head;
        while (temp && temp->data != val) temp = temp->next;
        if (temp) {
            temp->prev->next = temp->next;
            if (temp->next) temp->next->prev = temp->prev;
            delete temp;
        }
    }

    void traverse() {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

```
int main() {
    DoublyLinkedList dll;
    dll.insert(1);
    dll.insert(2);
    dll.deleteNode(2);
    dll.traverse(); // Expected Output: 1
    return 0;
}
```

Output:

```
1
```

iii. Implement circular linked lists with methods for insertion, deletion, and traversal

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

class CircularLinkedList {
    struct Node {
        int data;
        Node* next;
        Node(int val) : data(val), next(nullptr) {}
    };
    Node* head;

public:
    CircularLinkedList() : head(nullptr) {}

    void insert(int val) {
        Node* newNode = new Node(val);
        if (!head) {
```

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

    void traverse() {
        if (!head) return;
        Node* temp = head;
        do {
            cout << temp->data << "->";
            temp = temp->next;
        } while (temp != head);
        cout << temp->data << endl; // Prints the first node's data again to show the loop
    }
};

int main() {
    CircularLinkedList cll;
    cll.insert(1);
    cll.insert(2);
    cll.traverse(); // Expected Output: 1->2->1
    return 0;
}
```

Output:

```
1->2->1
```

2. Applications:

i. Postfix Calculator: Implement a stack-based solution for evaluating postfix expressions

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

class PostfixCalculator {
public:
    int evaluate(const string& expr) {
        stack<int> s;
        for (char ch : expr) {
            if (isdigit(ch)) {
                s.push(ch - '0'); // Convert char to integer
            } else {
                int b = s.top(); s.pop();
                int a = s.top(); s.pop();
                switch (ch) {
                    case '+': s.push(a + b); break;
                    case '-': s.push(a - b); break;
                    case '*': s.push(a * b); break;
                    case '/': s.push(a / b); break;
                }
            }
        }
        return s.top();
    }
};
```

```
int main() {
   PostfixCalculator calculator;
   string expression = "512+4*+3-";
   cout << "Postfix Expression Result: " << calculator.evaluate(expression) << endl;
   return 0;
}
```

Output:

```
Postfix Expression Result: 14
```

ii.   Queue-Based System Simulation: Simulate a queue-based ticketing system.

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

class TicketQueue {
   queue<string> tickets;

public:
   void enqueue(const string& ticket) {
      tickets.push(ticket);
   }

   string dequeue() {
      if (tickets.empty()) {
         return "No tickets to process!";
      }
      string ticket = tickets.front();
      tickets.pop();
```

```
        return ticket;
    }
};

int main() {
    TicketQueue tq;

    tq.enqueue("ticket1");
    tq.enqueue("ticket2");

    cout << "Processed Ticket: " << tq.dequeue() << endl; // Output: ticket1
    return 0;
}
```

Output:

```
Processed Ticket: ticket1
```

iii. Priority Queue Using Heaps: Implement a priority queue using a heap data structure.

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

class PriorityQueue {
    priority_queue<int, vector<int>, greater<int>> pq;

public:
    void insert(int val) {
        pq.push(val);
    }
    int remove() {
        if (pq.empty()) {
            return -1; // Indicate empty queue
        }
        int top = pq.top();
        pq.pop();
        return top;
    }
};

int main() {
    PriorityQueue pq;

    pq.insert(3);
    pq.insert(1);
    pq.insert(2);

    cout << "Removed Element: " << pq.remove() << endl; // Output: 1
    return 0;
}
```

Output:

```
Removed Element: 1
```