1.Write a Python program to manage the borrowing records of books in a library.

Implement the following functionalities:

• Compute the average number of books borrowed by all library members.

• Find the book with the highest and lowest number of borrowings in the library.

• Count the number of members who have not borrowed any books

(denoted by a borrow count of 0).

• Display the most frequently borrowed

```
# Library borrowing record program
# Book name : number of times borrowed
books = {"Book A": 10, "Book B": 5, "Book C": 0, "Book D": 15,"Book E": 7 }

# Number of books borrowed by each member
m = [3, 0, 5, 2, 0, 4]

# 1 Average books borrowed
avg = sum(m) / len(m)
print("Average borrowed:", avg)

# 2 Book with max and min borrowings
max_b = max(books, key=books.get)
min_b = min(books, key=books.get)

print("Most borrowed:", max_b, "(", books[max_b], ")")
print("Least borrowed:", min_b, "(", books[min_b], ")")

# 3 Count members with 0 borrowings
zero_m = m.count(0)
print("Members with 0 borrowings:", zero_m)

# 4 Most frequently borrowed book
print("Most frequent book:", max_b)
```

2.In an e-commerce system, customer account IDs are stored in a list, and you are tasked With writing a program that implements the following:

Linear Search: Check if a particular customer account ID exists in the list.

```
#include <iostream>
```

```
using namespace std;
int main() {
    int n, id, found = 0;
    cout << "Enter number of customer accounts: ";
    cin >> n;

    int acc[n];
    // Input account IDs
    cout << "Enter " << n << " account IDs:\n";
    for (int i = 0; i < n; i++) {
        cin >> acc[i];
    }
    // Input account ID to search
    cout << "Enter account ID to search: ";
    cin >> id;
    // Linear search
    for (int i = 0; i < n; i++) {
        if (acc[i] == id) {
            found = 1;
            cout << "Account ID " << id << " found at position " << i + 1 << endl;
            break;
        }
    }
    // If not found
    if (!found) {
        cout << "Account ID " << id << " not found." << endl;
    }
    return 0;
}
```

3. In an e-commerce system, customer account IDs are stored in a list, and you are tasked with writing a program that implements the following:

Binary Search: Implement Binary search to find if a customer account ID exists, improving the search efficiency over the basic linear

```
#include <iostream>
```

```
using namespace std;
int main() {
    int n, id;
    // Get number of customer accounts
    cout << "Enter number of customer accounts: ";
    cin >> n;
    int acc[n];
    // Input sorted account IDs (Binary Search requires sorted data)
    cout << "Enter " << n << " account IDs in sorted order:\n";
    for (int i = 0; i < n; i++) {
        cin >> acc[i];
    }
    // Input the account ID to search
    cout << "Enter account ID to search: ";
    cin >> id;
    // Binary Search
    int low = 0, high = n - 1, mid;
    bool found = false;

    while (low <= high) {
        mid = (low + high) / 2;

        if (acc[mid] == id) {
            cout << "Account ID " << id << " found at position " << mid + 1 << endl;
            found = true;
            break;
        }
        else if (acc[mid] < id) {
            low = mid + 1; // Search right half
        }
        else {
            high = mid - 1; // Search left half
        }
    }
    if (!found) {
```

```
        cout << "Account ID " << id << " not found." << endl;
    }
    return 0;
}
```

4.In a company, employee salaries are stored in a list as floating-point numbers. Write a Python program that sorts the employee salaries in ascending order using the following algorithms:

Selection Sort: Sort the salaries using the selection sort algorithm.

After sorting the salaries, the program should display top five highest salaries in the company.

```
# Program to sort employee salaries using Selection Sort
# and display top 5 highest salaries
# Sample list of employee salaries
salaries = [45000.50, 32000.75, 55000.00, 28000.25, 60000.10, 75000.30, 49000.40, 38000.90]

# Selection Sort algorithm (ascending order)
n = len(salaries)
for i in range(n):
    min_index = i
    for j in range(i + 1, n):
        if salaries[j] < salaries[min_index]:
            min_index = j
    # Swap
    salaries[i], salaries[min_index] = salaries[min_index], salaries[i]

# Display sorted salaries
print("Salaries in ascending order:")
print(salaries)

# Display top 5 highest salaries
print("\nTop 5 highest salaries:")
# Slicing last 5 elements and reversing for descending order
for s in salaries[-5:][::-1]:
    print(s)
```

5.In a company, employee salaries are stored in a list as floating-point numbers. Write a Python program that sorts the employee salaries in ascending order using the following two algorithms:

Bubble Sort: Sort the salaries using the bubble sort algorithm.

After sorting the salaries, the program should display top five highest salaries in the company

```
# Program to sort employee salaries using Bubble Sort
# and display the top 5 highest salaries
# Sample list of employee salaries
salaries = [45000.50, 32000.75, 55000.00, 28000.25, 60000.10, 75000.30, 49000.40, 38000.90]


# Bubble Sort algorithm (ascending order)
n = len(salaries)
for i in range(n - 1):
    for j in range(n - i - 1):
        if salaries[j] > salaries[j + 1]:
            # Swap if current salary is greater than next salary
            salaries[j], salaries[j + 1] = salaries[j + 1], salaries[j]


# Display sorted salaries
print("Salaries in ascending order:")
print(salaries)
# Display top 5 highest salaries
print("\nTop 5 highest salaries:")
for s in salaries[-5:][::-1]:
    print(s)
```

6. Implementing a real-time undo/redo system for a text editing application using a Stack data structure. The system should support the following operations:

• Make a Change: A new change to the document is made.

• Undo Action: Revert the most recent change and store it for potential redo.

• Redo Action: Reapply the most recently undone action.

• Display Document State: Show the current state of the document after undoing or redoing an action.

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
int main() {
    queue<string> q;   // queue for events
    int ch;
```

```
string e;
while (true) {
    cout << "\n1. Add Event\n2. Process Event\n3. Show Events\n4. Cancel Event\n5. Exit\n";
    cout << "Enter choice: ";
    cin >> ch;
    cin.ignore();

    if (ch == 1) {
        cout << "Enter event: ";
        getline(cin, e);
        q.push(e);
        cout << "Event added.\n";
    }
    else if (ch == 2) {
        if (!q.empty()) {
            cout << "Processed: " << q.front() << endl;
            q.pop();
        } else {
            cout << "No events.\n";
        }
    }
    else if (ch == 3) {
        if (q.empty()) cout << "No pending events.\n";
        else {
            cout << "Pending:\n";
            queue<string> temp = q;
            while (!temp.empty()) {
                cout << "- " << temp.front() << endl;
                temp.pop();
            }
        }
    }
    else if (ch == 4) {
        if (q.empty()) cout << "No events to cancel.\n";
        else {
```

```
            cout << "Enter event to cancel: ";
            getline(cin, e);
            queue<string> temp;
            bool found = false;

            while (!q.empty()) {
               if (q.front() == e) {
                  cout << "Canceled: " << e << endl;
                  found = true;
               } else temp.push(q.front());
               q.pop();
            }
            q = temp;
            if (!found) cout << "Event not found.\n";
         }
      }
      else if (ch == 5) {
         cout << "Goodbye!\n";
         break;
      }
      else {
         cout << "Invalid choice!\n";
      }
   }
   return 0;
}
```

7. Implement a real-time event processing system using a Queue data structure. The System should support the following features:

• Add an Event: When a new event occurs, it should be added to the event queue.

• Process the Next Event: The system should process and remove the event that has been in the queue the longest.

• Display Pending Events: Show all the events currently waiting to be processed.

• Cancel an Event: An event can be canceled if it has not been processed.

```
#include <iostream>
#include <queue>
```

```
using namespace std;

int main() {
    queue<string> q;   // queue for events
    int ch;
    string e;

    while (true) {
        cout << "\n1. Add Event\n2. Process Event\n3. Show Events\n4. Cancel Event\n5. Exit\n";
        cout << "Enter choice: ";
        cin >> ch;

        if (ch == 1) {
            cout << "Enter event name (one word): ";
            cin >> e;
            q.push(e);
            cout << "Event added.\n";
        }
        else if (ch == 2) {
            if (!q.empty()) {
                cout << "Processing: " << q.front() << endl;
                q.pop();
            } else {
                cout << "No events to process.\n";
            }
        }
        else if (ch == 3) {
            if (q.empty()) {
                cout << "No pending events.\n";
            } else {
                cout << "Pending events:\n";
                queue<string> temp = q;
                while (!temp.empty()) {
                    cout << temp.front() << endl;
                    temp.pop();
```

```
            }
        }
    }
    else if (ch == 4) {
        if (q.empty()) {
            cout << "No events to cancel.\n";
        } else {
            cout << "Enter event to cancel: ";
            cin >> e;
            queue<string> temp;
            bool found = false;

            while (!q.empty()) {
                if (q.front() == e) {
                    cout << "Canceled: " << e << endl;
                    found = true;
                } else {
                    temp.push(q.front());
                }
                q.pop();
            }
            q = temp;
            if (!found) cout << "Event not found.\n";
        }
    }
    else if (ch == 5) {
        cout << "Goodbye!\n";
        break;
    }
    else {
        cout << "Invalid choice!\n";
    }
  }
  return 0;
}
```

8. Implement a real-time event processing system using a Circular Queue data structure using array. The System should support the following features:

• Add an Event: When a new event occurs, it should be added to the event queue.

• Process the Next Event: The system should process and remove the event

that has been in the queue the longest.

• Display Pending Events: Show all the events currently waiting to be processed.

• Cancel an Event: An event can be canceled if it has not been processed.

```
#include <iostream>
using namespace std
#define SIZE 5
int main() {
   int q[SIZE];       // queue array
   int front = 0, rear = -1, count = 0;
   int ch, e;

   while (1) {
      cout << "\n1. Add Event\n2. Process Event\n3. Show Events\n4. Exit\n";
      cout << "Enter your choice: ";
      cin >> ch;

      // Add event
      if (ch == 1) {
         if (count == SIZE)
            cout << "Queue full!\n";
         else {
            cout << "Enter event ID: ";
            cin >> e;
            rear = (rear + 1) % SIZE;
            q[rear] = e;
            count++;
            cout << "Event added.\n";
         }
      }
      // Process next event
      else if (ch == 2) {
         if (count == 0)
```

```
            cout << "No events to process.\n";
        else {
            cout << "Processing event " << q[front] << endl;
            front = (front + 1) % SIZE;
            count--;
        }
    }
    // Show all pending events
    else if (ch == 3) {
        if (count == 0)
            cout << "No pending events.\n";
        else {
            cout << "Pending events: ";
            int i = front;
            for (int c = 0; c < count; c++) {
                cout << q[i] << " ";
                i = (i + 1) % SIZE;
            }
            cout << endl;
        }
    }
    // Exit
    else if (ch == 4) {
        cout << "Goodbye!\n";
        break;
    }
    else
        cout << "Invalid choice!\n";
  }
  return 0;
}
```

9. Create a Student Record Management System using linked list

• Use a singly/doubly linked list to store student data (Roll No, Name, Marks).

• Perform operations: Add, Display, Delete, Update.

• Display records in ascending/descending order based on marks or roll number.

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

struct Student {
    int roll;
    char name[20];
    float marks;
    Student* next;
};
Student* head = NULL;
void add() {
    Student* s = new Student;
    cout << "Enter Roll No: ";
    cin >> s->roll;
    cout << "Enter Name: ";
    cin >> s->name;
    cout << "Enter Marks: ";
    cin >> s->marks;

    s->next = head;
    head = s;

    cout << "Student Added!\n";
}
void display() {
    if (head == NULL) {
        cout << "No records!\n";
        return;
    }
    Student* t = head;
    cout << "\n--- Student Records ---\n";
    while (t != NULL) {
        cout << "Roll No: " << t->roll << endl;
        cout << "Name: " << t->name << endl;
        cout << "Marks: " << t->marks << endl << endl;
```

```
        t = t->next;
    }
}
int main() {
    int ch;
    while (1) {
        cout << "\n1. Add Student\n2. Display Students\n3. Exit\n";
        cout << "Enter your choice: ";
        cin >> ch;

        if (ch == 1)
            add();
        else if (ch == 2)
            display();
        else if (ch == 3)
            break;
        else
            cout << "Invalid choice!\n";
    }
    return 0;
}
```

10. Create a Student Record Management System using linked list

• Use a singly/doubly linked list to store student data (Roll No, Name, Marks).

• Perform operations: Add, Display, Search, and Sort.

• Display records in ascending/descending order based on marks or roll number.

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

struct Student {
    int roll;
    char name[20];
    float marks;
    Student* next;
};
```

```
Student* head = NULL;

void add() {
    Student* s = new Student;
    cout << "Roll No: "; cin >> s->roll;
    cout << "Name: "; cin >> s->name;
    cout << "Marks: "; cin >> s->marks;
    s->next = head;
    head = s;
}

void display() {
    Student* t = head;
    while (t) {
        cout << t->roll << " " << t->name << " " << t->marks << endl;
        t = t->next;
    }
}

void search() {
    int r; cout << "Enter roll: "; cin >> r;
    Student* t = head;
    while (t) {
        if (t->roll == r) {
            cout << "Found: " << t->name << " " << t->marks << endl;
            return;
        }
        t = t->next;
    }
    cout << "Not found!\n";
}
void sortList() {
    for (Student* i = head; i && i->next; i = i->next)
        for (Student* j = i->next; j; j = j->next)
```

```
            if (i->roll > j->roll)
                swap(i->roll, j->roll), swap(i->marks, j->marks), swap(i->name, j->name);
    cout << "Sorted by roll number!\n";
}
int main() {
    int ch;
    do {
        cout << "\n1.Add 2.Display 3.Search 4.Sort 5.Exit: ";
        cin >> ch;
        if (ch == 1) add();
        else if (ch == 2) display();
        else if (ch == 3) search();
        else if (ch == 4) sortList();
    } while (ch != 5);
}
```

11. Implement a hash table of size 10 and use the division method as a hash function. In case of a collision, use chaining. Implement the following operations:

• Insert(key): Insert key-value pairs into the hash table.

• Search(key): Search for the value associated with a given key.

• Display(): Displays the hash table.

```
#include <iostream>
using namespace std;
struct Node {
    int key;
    Node* next;
};

Node* table[10] = {NULL};

int hashFunc(int key) {
    return key % 10;   // Division method
}
void insert(int key) {
    int index = hashFunc(key);
    Node* n = new Node;
    n->key = key;
```

```
    n->next = NULL;

    if (table[index] == NULL)
        table[index] = n;
    else {
        Node* t = table[index];
        while (t->next != NULL)
            t = t->next;
        t->next = n;
    }
    cout << "Inserted!\n";
}

void search(int key) {
    int index = hashFunc(key);
    Node* t = table[index];
    while (t != NULL) {
        if (t->key == key) {
            cout << "Key found!\n";
            return;
        }
        t = t->next;
    }
    cout << "Key not found!\n";
}

void display() {
    for (int i = 0; i < 10; i++) {
        cout << i << ": ";
        Node* t = table[i];
        while (t != NULL) {
            cout << t->key << " -> ";
            t = t->next;
        }
        cout << "NULL\n";
```

```
    }
}
int main() {
    int ch, key;
    do {
        cout << "\n1.Insert 2.Search 3.Display 4.Exit: ";
        cin >> ch;

        if (ch == 1) {
            cout << "Enter key: ";
            cin >> key;
            insert(key);
        }
        else if (ch == 2) {
            cout << "Enter key: ";
            cin >> key;
            search(key);
        }
        else if (ch == 3)
            display();
    } while (ch != 4);
    return 0;
}
```

12. Implement a hash table of size 10 and use the division method as a hash function. In case of a collision, use chaining. Implement the following operations:

• Insert(key): Insert key-value pairs into the hash table.

• Delete(key): Delete a key-value pair from the hash table.

• Display(): Displays the hash table.

```
#include <iostream>
using namespace std;
struct Node { int key; Node* next; };
Node* table[10] = {NULL};
int hashFunc(int k) { return k % 10; }
void insert(int k) {
    int i = hashFunc(k);
    Node* n = new Node{k, NULL};
```

```
    if (!table[i]) table[i] = n;
    else {
        Node* t = table[i];
        while (t->next) t = t->next;
        t->next = n;
    }
    cout << "Inserted!\n";
}
void del(int k) {
    int i = hashFunc(k);
    Node* t = table[i], *p = NULL;
    while (t) {
        if (t->key == k) {
            if (!p) table[i] = t->next;
            else p->next = t->next;
            delete t;
            cout << "Deleted!\n";
            return;
        }
        p = t; t = t->next;
    }
    cout << "Not found!\n";
}

void display() {
    for (int i = 0; i < 10; i++) {
        cout << i << ": ";
        for (Node* t = table[i]; t; t = t->next)
            cout << t->key << " -> ";
        cout << "NULL\n";
    }
}
int main() {
    int ch, k;
    do {
```

```
        cout << "\n1.Insert 2.Delete 3.Display 4.Exit: ";
        cin >> ch;
        if (ch == 1) { cout << "Enter key: "; cin >> k; insert(k); }
        else if (ch == 2) { cout << "Enter key: "; cin >> k; del(k); }
        else if (ch == 3) display();
    } while (ch != 4);
}
```

13. & 14. Design and implement a hash table of fixed size. Use the division method for the hash function and resolve collisions using linear probing. Allow the user to perform the following operations:

- **Insert a key**
- **Search for a key**
- **Display the table**
- **Delete a key**

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

const int SIZE = 10;
int hashTable[SIZE];
bool isOccupied[SIZE] = {false};

int hashFunc(int key) {
    return key % SIZE;
}
void insertKey(int key) {
    int index = hashFunc(key);
    for (int i = 0; i < SIZE; i++) {
        int pos = (index + i) % SIZE;
        if (!isOccupied[pos]) {
            hashTable[pos] = key;
            isOccupied[pos] = true;
            cout << "Inserted!\n";
            return;
        }
    }
    cout << "Table full!\n";
}
```

```
void searchKey(int key) {
    int index = hashFunc(key);
    for (int i = 0; i < SIZE; i++) {
        int pos = (index + i) % SIZE;
        if (isOccupied[pos] && hashTable[pos] == key) {
            cout << "Key found at index " << pos << endl;
            return;
        }
    }
    cout << "Key not found!\n";
}

void deleteKey(int key) {
    int index = hashFunc(key);
    for (int i = 0; i < SIZE; i++) {
        int pos = (index + i) % SIZE;
        if (isOccupied[pos] && hashTable[pos] == key) {
            isOccupied[pos] = false;
            cout << "Key deleted!\n";
            return;
        }
    }
    cout << "Key not found!\n";
}

void display() {
    cout << "\nHash Table:\n";
    for (int i = 0; i < SIZE; i++) {
        if (isOccupied[i])
            cout << i << ": " << hashTable[i] << endl;
        else
            cout << i << ": Empty\n";
    }
}
```

```
int main() {
   int ch, key;
   do {
      cout << "\n1.Insert 2.Search 3.Delete 4.Display 5.Exit: ";
      cin >> ch;
      if (ch == 1) { cout << "Enter key: "; cin >> key; insertKey(key); }
      else if (ch == 2) { cout << "Enter key: "; cin >> key; searchKey(key); }
      else if (ch == 3) { cout << "Enter key: "; cin >> key; deleteKey(key); }
      else if (ch == 4) display();
   } while (ch != 5);
}
```

15. A pizza shop receives multiple orders from several locations. Assume that one pizza boy is tasked with delivering pizzas in nearby locations, which is represented using a graph.The time required to reach from one location to another represents node connections. Solve the problem of delivering a pizza to all customers in the minimum time. Use appropriate data structures.

```
#include <iostream>
using namespace std;
int main() {
   int n = 5; // number of locations
   int graph[5][5] = {
      {0, 10, 0, 30, 100},
      {10, 0, 50, 0, 0},
      {0, 50, 0, 20, 10},
      {30, 0, 20, 0, 60},
      {100, 0, 10, 60, 0}
   };

   int dist[5], visited[5];
   int start = 0; // starting location

   for (int i = 0; i < n; i++) {
      dist[i] = 9999;  // large number
      visited[i] = 0;
   }
   dist[start] = 0;
```

```
    for (int i = 0; i < n - 1; i++) {
        int min = 9999, u = -1;
        for (int j = 0; j < n; j++) {
            if (!visited[j] && dist[j] < min) {
                min = dist[j];
                u = j;
            }
        }
        visited[u] = 1;

        for (int v = 0; v < n; v++) {
            if (graph[u][v] && !visited[v] && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
        }
    }
    cout << "Minimum time from location " << start << ":\n";
    for (int i = 0; i < n; i++)
        cout << "To location " << i << " = " << dist[i] << endl;
    return 0;
}
```

16. Construct an expression tree from the given prefix expression, e.g., +--a*bc/def, traverse it using post-order traversal (non-recursive), and then delete the entire tree.

```
#include <iostream>
#include <stack>
using namespace std;
struct Node {
    char data;
    Node *left, *right;
};
Node* newNode(char c) {
    Node* n = new Node;
    n->data = c;
    n->left = n->right = NULL;
    return n;
}
```

```
// Build tree from prefix expression
Node* buildTree(string pre) {
    stack<Node*> s;
    for (int i = pre.size() - 1; i >= 0; i--) {
        char c = pre[i];
        Node* n = newNode(c);
        if (isalpha(c))
            s.push(n);
        else {
            n->left = s.top(); s.pop();
            n->right = s.top(); s.pop();
            s.push(n);
        }
    }
    return s.top();
}

// Non-recursive postorder
void postOrder(Node* root) {
    stack<Node*> s1, s2;
    s1.push(root);
    while (!s1.empty()) {
        Node* temp = s1.top(); s1.pop();
        s2.push(temp);
        if (temp->left) s1.push(temp->left);
        if (temp->right) s1.push(temp->right);
    }
    cout << "Postorder: ";
    while (!s2.empty()) {
        cout << s2.top()->data;
        s2.pop();
    }
}
// Delete tree
void deleteTree(Node* root) {
```

```
    if (root) {
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
}
int main() {
    string pre = "+--a*bc/def";
    Node* root = buildTree(pre);
    postOrder(root);
    deleteTree(root);
    cout << "\nTree deleted.";
}
```

17.In a computer system, multiple functions call one another during program execution. To manage these nested function calls efficiently, a Stack data structure is used to store the state of each function call. When a function is called, its details are pushed onto the stack, and when the function returns, its details are popped. Implement this mechanism using a Linked List, which allows dynamic allocation of memory and avoids stack overflow issues seen in static arrays. Demonstrate operations such as Push, Pop, and Display, simulating how function call management is performed in real-time systems like compilers or interpreters.

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

struct Node {
    string name;
    Node* next;
};
Node* top = NULL;

void push(string n) {
    Node* t = new Node;
    t->name = n;
    t->next = top;
    top = t;
    cout << "Call: " << n << endl;
}
```

```
void pop() {
    if (!top) cout << "No function to return!\n";
    else {
        cout << "Return: " << top->name << endl;
        Node* temp = top;
        top = top->next;
        delete temp;
    }
}
void show() {
    cout << "Stack:\n";
    for (Node* t = top; t; t = t->next)
        cout << t->name << endl;
}
int main() {
    push("main()");
    push("A()");
    push("B()");
    show();
    pop();
    show();
    pop();
    pop();
    pop();
}
```

18. Consider a real-time scenario where customers arrive at a ticket booking counter one after another. The customers must be served in the same order they arrive — the First In First Out (FIFO) manner. Design a Queue using a Linked List to handle this system dynamically. Each node in the linked list represents a customer waiting in line. Implement operations such as Enqueue (customer joins the queue), Dequeue (customer is served and leaves), and Display (show waiting customers). This practical simulates real-time service queue management systems used in banks, hospitals, or ticketing applications.

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

struct Node {
    string name;
    Node* next;
```

```
};
Node *front = NULL, *rear = NULL;

void enqueue(string n) {
    Node* t = new Node;
    t->name = n;
    t->next = NULL;
    if (rear == NULL)
        front = rear = t;
    else {
        rear->next = t;
        rear = t;
    }
    cout << n << " joined the queue.\n";
}

void dequeue() {
    if (front == NULL)
        cout << "No customers to serve.\n";
    else {
        cout << front->name << " is served and leaves.\n";
        Node* temp = front;
        front = front->next;
        if (front == NULL) rear = NULL;
        delete temp;
    }
}

void display() {
    if (front == NULL)
        cout << "No waiting customers.\n";
    else {
        cout << "Waiting customers:\n";
        for (Node* t = front; t; t = t->next)
            cout << t->name << endl;
```

```
    }
}

int main() {
    enqueue("Customer1");
    enqueue("Customer2");
    enqueue("Customer3");
    display();
    dequeue();
    display();
    dequeue();
    dequeue();
    dequeue();
}
```

19. There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight takes to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.

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

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

    int graph[10][10];
    cout << "Enter adjacency matrix (0 = no flight):\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> graph[i][j];

    // Check connectivity using simple visit logic
    int visited[10] = {0};
    visited[0] = 1; // start from city 0
    for (int k = 0; k < n; k++)
```

```
    for (int i = 0; i < n; i++)
        if (visited[i])
            for (int j = 0; j < n; j++)
                if (graph[i][j] && !visited[j])
                    visited[j] = 1;

  int connected = 1;
  for (int i = 0; i < n; i++)
    if (!visited[i]) connected = 0;

  cout << "\nAdjacency Matrix:\n";
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << graph[i][j] << " ";
    cout << endl;
  }

  if (connected)
    cout << "\nGraph is CONNECTED (all cities can reach each other)\n";
  else
    cout << "\nGraph is NOT CONNECTED (some cities are isolated)\n";

  cout << "\nJustification:\n";
  cout << "- Adjacency matrix is easy to use and understand.\n";
  cout << "- It quickly shows if a flight exists between two cities.\n";
  cout << "- Suitable for small or medium networks.\n";

  return 0;
}
```

20. Implement a real-time event processing system using a Circular Queue data structure using linked list. The System should support the following features:

• Add an Event: When a new event occurs, it should be added to the event queue.• Process the Next Event: The system should process and remove the event that has been in the queue the longest.

• Display Pending Events: Show all the events currently waiting to be processed.

• Cancel an Event: An event can be canceled if it has not been processed.

```
#include <iostream>
using namespace std;
struct Node {
    string event;
    Node* next;
};
Node *front = NULL, *rear = NULL;

void add(string e) {
    Node* n = new Node{e, NULL};
    if (!front) front = rear = n, n->next = front;
    else rear->next = n, rear = n, rear->next = front;
    cout << e << " added\n";
}

void process() {
    if (!front) return void(cout << "No events\n");
    cout << "Processing: " << front->event << endl;
    if (front == rear) front = rear = NULL;
    else rear->next = front->next, delete front, front = rear->next;
}

void show() {
    if (!front) return void(cout << "No events\n");
    Node* t = front;
    do { cout << t->event << " "; t = t->next; } while (t != front);
    cout << endl;
}

int main() {
    add("E1"); add("E2"); add("E3");
    show();
    process(); show();
    process(); process(); show();
}
```