CS 202: Data Structures

Programming Assignment 3

Project Evolve: The Analytics Engine

Lead TAs: Shahzaib Ali (GOAT), Abdullah Mushtaq, Umer Irfan, Sania Hassan

Deadline: Tuesday, 25th November

Contents

I Part 1: The Core Data Structures **5**

Part 1: The Core Data Structures **5**

1 The `MinHeap` Class **5**

The MinHeap Class **5**

1.1 Constructor and Destructor . . . 5

Constructor and Destructor . . . 5

1.2 Copy Operations . . . 6

Copy Operations . . . 6

1.3 Index Helper Functions . . . 6

Index Helper Functions . . . 6

1.4 Core Heap Operations . . . 6

Core Heap Operations . . . 6

1.5 Key Update Operations . . . 7

Key Update Operations . . . 7

1.6 Heap Maintenance . . . 7

Heap Maintenance . . . 7

2 The `SocialMediaSystem` Class **7**

The SocialMediaSystem Class **7**

2.1 Core Functionality . . . 8

Core Functionality . . . 8

3 The `HashTable` Class **9**

The HashTable Class **9**

3.1 Constructor and Destructor . . . 9

Constructor and Destructor . . . 9

3.2 Core Operations . . . 9

Core Operations . . . 9

3.3 Collision Handling: Linear Probing . . . 10

Collision Handling: Linear Probing . . . 10

3.4 Collision Handling: Quadratic Probing . . . 10

Collision Handling: Quadratic Probing . . . 10

3.5 Collision Handling: Separate Chaining . . . 10

Collision Handling: Separate Chaining . . . 10

3.6 Table Maintenance . . . 10

Table Maintenance . . . 10

4 Marks Distribution (Part 1) **11**

Marks Distribution (Part 1) **11**

II Part 2: The Analytics Engine **12**

Part 2: The Analytics Engine 12

5 Introduction & Architectural Evolution 12

Introduction & Architectural Evolution 12

6 Algorithm Preliminaries 12

Algorithm Preliminaries 12
6.1 Dijkstra's Algorithm . . . 12
Dijkstra's Algorithm . . . 12
6.2 Kruskal's Algorithm & DSU . . . 13
Kruskal's Algorithm DSU . . . 13
6.3 PageRank Algorithm . . . 13
PageRank Algorithm . . . 13
6.4 Kahn's Algorithm (Topological Sort) . . . 13
Kahn's Algorithm (Topological Sort) . . . 13

7 The `SocialGraph` Class 14

The SocialGraph Class 14
7.1 Core Graph Management . . . 14
Core Graph Management . . . 14
7.2 Core Algorithm: Pathfinding . . . 15
Core Algorithm: Pathfinding . . . 15
7.3 Advanced Analysis: Echo Chamber Detection . . . 15
Advanced Analysis: Echo Chamber Detection . . . 15
7.4 Advanced Analysis: Influence Ranking . . . 15
Advanced Analysis: Influence Ranking . . . 15
7.5 Advanced Analysis: Community Detection . . . 16
Advanced Analysis: Community Detection . . . 16

8 The `GeographicNetwork` Class 17

The GeographicNetwork Class 17
8.1 Core Graph Management . . . 17
Core Graph Management . . . 17
8.2 Core Algorithm: Lowest Latency Routing . . . 18
Core Algorithm: Lowest Latency Routing . . . 18
8.3 Core Algorithm: Network Backbone Design . . . 18
Core Algorithm: Network Backbone Design . . . 18
8.4 Advanced Analysis: Critical Node Identification . . . 18
Advanced Analysis: Critical Node Identification . . . 18
8.5 Advanced Analysis: Minimum Effort Path . . . 19
Advanced Analysis: Minimum Effort Path . . . 19
8.6 Advanced Analysis: Finding the Optimal City . . . 19
Advanced Analysis: Finding the Optimal City . . . 19

9 The `InteractionGraph` Class 20

The InteractionGraph Class 20
9.1 Core Graph Management . . . 20
Core Graph Management . . . 20

9.2 Advanced Analysis: Finding Digital Twins . . . 21
Advanced Analysis: Finding Digital Twins . . . 21
9.3 Advanced Analysis: The Content Recommender . . . 21
Advanced Analysis: The Content Recommender . . . 21
9.4 Advanced Analysis: Identifying Viral Content . . . 21
Advanced Analysis: Identifying Viral Content . . . 21
9.5 Advanced Analysis: Dependency Scheduling . . . 22
Advanced Analysis: Dependency Scheduling . . . 22

10 Marks Distribution (Part 2: The Analytics Engine) 23

Marks Distribution (Part 2: The Analytics Engine) 23

11 Testing Your Implementation 24

Testing Your Implementation 24
11.1 Running the Complete Test Suite . . . 24
11.2 Running Part-Specific Suites . . . 24
11.3 Running Individual Component Tests . . . 24
11.4 Cleaning Up . . . 24

12 Submission Instructions 25

Submission Instructions 25
12.1 File Naming and Format . . . 25
12.2 Directory Structure . . . 25

Part I

Part 1: The Core Data Structures

1 The `MinHeap` Class

*This section focuses on implementing and applying the **MinHeap** data structure to manage a social media feed based on post engagement.*

MinHeap
// Constructor & Destructor
+ MinHeap(int cap)
+ ~MinHeap()
// Copy Operations
+ MinHeap(const MinHeap& other)
+ operator=(const MinHeap& other): MinHeap&
// Index Helpers
+ parent(int i) const: int
+ left(int i) const: int
+ right(int i) const: int
// Core Heap Operations
+ insertKey(int k)
+ extractMin(): int
+ getMin() const: int
// Key Update Operations
+ decreaseKey(int i, int new_val)
+ increaseKey(int i, int new_val)
+ deleteKey(int i)
// Heap Maintenance
+ minHeapify(int i)
+ size() const: int
+ elementAt(int i) const: int

A MinHeap maintains a hierarchical ordering where the smallest value (minimum key) always sits at the root. This ensures fast access to the lowest-priority element — in our social media example, that could mean the least-liked post, the oldest story, or the least urgent report.

In a real-world analogy, think of a social media app like Instagram or X. Every post has a "priority" based on engagement. Using a heap, we can efficiently manage posts as their popularity changes — adding new ones, updating like counts, and clearing out those with low engagement.

The `MinHeap` class stores integer values representing post priorities (e.g., like counts). It supports the following functionalities:

1.1 Constructor and Destructor

```
1 MinHeap(int cap);
```

- **Task:** Initializes a new heap with a given maximum capacity.

```
1 ~MinHeap();
```

- **Task:** Frees all dynamically allocated memory used by the heap.

1.2 Copy Operations

```
MinHeap(const MinHeap& other);
```

- **Task:** Creates a new heap as a deep copy of another heap.

```
MinHeap& operator=(const MinHeap& other);
```

- **Task:** Assigns one heap to another by copying all its internal data.

1.3 Index Helper Functions

These utility functions define relationships between nodes in the heap's underlying array:

```
int parent(int i);
```

- **Task:** Returns the parent index of node `i`. Formula: $\text{parent}(i) = \lfloor \frac{i-1}{2} \rfloor$

```
int left(int i);
```

- **Task:** Returns the left child index of node `i`. Formula: $\text{left}(i) = 2i + 1$

```
int right(int i);
```

- **Task:** Returns the right child index of node `i`. Formula: $\text{right}(i) = 2i + 2$

1.4 Core Heap Operations

```
void insertKey(int k);
```

- **Task:** Adds a new post (represented by its like count) into the feed and restores the heap order by moving it upward if necessary.

```
int extractMin();
```

- **Task:** Removes and returns the least-liked post (the root element). The last element is moved to the top, and the heap is rebalanced using `minHeapify()`.

```
int getMin() const;
```

- **Task:** Returns the current minimum like count without removing it — useful for checking which post is performing worst.

1.5 Key Update Operations

```
void decreaseKey(int i, int new_val);
```

- **Task:** Reduces the like count at index `i` to a smaller value, simulating a drop in engagement. The heap property is restored by moving the element upward.

```
void increaseKey(int i, int new_val);
```

- **Task:** Increases the like count at index `i`, simulating a surge in engagement. The node may move downward to maintain heap order.

```
void deleteKey(int i);
```

- **Task:** Deletes a post from the heap by setting its value to negative infinity, moving it to the root, and then calling `extractMin()`.

1.6 Heap Maintenance

```
void minHeapify(int i);
```

- **Task:** Ensures that the subtree rooted at index `i` satisfies the min-heap property — if any child is smaller than the parent, the elements are swapped recursively.

```
int size() const;
```

- **Task:** Returns the total number of posts currently stored in the heap.

```
int elementAt(int i) const;
```

- **Task:** Returns the value (like count) stored at index `i`. Used internally to traverse and inspect the heap.

2 The `SocialMediaSystem` Class

The `SocialMediaSystem` class applies the heap to simulate a simplified social media environment. Each post's like count is stored in a heap, which automatically prioritizes posts based on engagement. The root of the heap always represents the least-engaged post.

SocialMediaSystem
// Core Functionality
+ findPostIndex(int value): int
+ buildUndiscoveredFeed(std::vector¡int¿ likes)
+ updatePostLikes(int oldLikes, int newLikes)
+ clearLowEngagementPosts(int threshold)
+ getTopNPosts(int N): std::vector¡int¿
+ decreasePostLikes(int currentLikes, int newLikes)

2.1 Core Functionality

```
int findPostIndex(int value);
```

- **Task:** Searches through the heap to find the index of a specific post (by like count).

```
void buildUndiscoveredFeed(vector<int> likes);
```

- **Task:** Builds an initial feed by inserting multiple posts (each represented by a like count) into the heap.

```
void updatePostLikes(int oldLikes, int newLikes);
```

- **Task:** Simulates a post gaining popularity. Locates the post, increases its like count using `increaseKey()`, and rebalances the heap.

```
void clearLowEngagementPosts(int threshold);
```

- **Task:** Removes all posts with likes below a certain threshold by repeatedly extracting the minimum element.

```
vector<int> getTopNPosts(int N);
```

- **Task:** Returns the top `N` posts by like count without changing the original heap. This is achieved by copying the heap and extracting the smallest elements from the copy.

```
void decreasePostLikes(int currentLikes, int newLikes);
```

- **Task:** Simulates a post losing popularity by lowering its like count via `decreaseKey()`.

3 The `HashTable` Class

*This section focuses on building an efficient storage system using a **hash table**.*

HashTable¡T¿
// Constructor & Destructor
+ HashTable(...)
+ H̃ashTable()
// Core Operations
+ insert(int key, T value)
+ search(int key): T
+ remove(int key)
// Collision Handlers
+ insertLinearProbing(...)
+ searchLinearProbing(...)
+ removeLinearProbing(...)
+ insertQuadraticProbing(...)
+ searchQuadraticProbing(...)
+ removeQuadraticProbing(...)
+ insertSeparateChaining(...)
+ searchSeparateChaining(...)
+ removeSeparateChaining(...)
// Table Maintenance
+ resizeAndRehash()
+ calculateLoadFactor()
+ displayProbingTable()

You are provided with a class template `HashTable<T>`. Each entry will store a `key` (integer ID) and a `value` (any data type). Think of each user as having an ID, and each ID needs to be linked to some stored profile. When millions of users exist, we can't just keep searching one by one — we need a mapping structure. That's what your hash table will do. You must complete the following functions.

3.1 Constructor and Destructor

- **Task:** Initialize the table with a fixed starting size. Initialize the probing table to some size dynamically and update the table size accordingly. The destructor should clean up all dynamic memory.

3.2 Core Operations

```
1 void insert(int key, T value);
```

- **Task:** Adds a new key–value pair to the table using hashing and collision handling.

```
1 T search(int key);
```

- **Task:** Looks up a key and returns the stored value. Internally, this performs a hashing operation and probes the table as needed.

```
1 void remove(int key);
```

- **Task:** Deletes a key–value pair from the table while maintaining probe integrity.

3.3 Collision Handling: Linear Probing

```
void insertLinearProbing(int key, T value);
void searchLinearProbing(int key);
void removeLinearProbing(int key);
```

- **Task:** Handle collisions by checking the next slot sequentially.

3.4 Collision Handling: Quadratic Probing

```
void insertQuadraticProbing(int key, T value);
void searchQuadraticProbing(int key);
void removeQuadraticProbing(int key);
```

- **Task:** Handle collisions by skipping in quadratic intervals to reduce clustering.
- **Algorithm Notes:** Quadratic probing resolves collisions by checking slots at increasing square intervals. After a collision, it probes $(h_0 + i^2)$ mod table_size, helping spread entries out and reduce clustering.

3.5 Collision Handling: Separate Chaining

```
void insertSeparateChaining(int key, T value);
void searchSeparateChaining(int key);
void removeSeparateChaining(int key);
```

- **Task:** Store colliding elements in linked chains (vectors) under the same index.
- **Algorithm Notes:** Separate chaining stores multiple key–value pairs at the same index using a small list or vector. If two keys hash to the same slot, they're simply linked in that chain — like grouping users under the same hash bucket.

3.6 Table Maintenance

```
void resizeAndRehash();
```

- **Task:** Expands the table and re-inserts all data when the load factor exceeds the threshold.

```
void calculateLoadFactor();
```

- **Task:** Updates the ratio of used slots to total size and triggers rehashing when needed.

```
void displayProbingTable();
```

- **Task:** Displays all stored key–value pairs for debugging or visualization.

4 Marks Distribution (Part 1)

Component	Task	Marks
Heaps (60)		**60**
	`MinHeap` Implementation (Core Ops, Updates)	30
	`SocialMediaSystem` Application (Feed Logic)	30
Hash Table (40)		**40**
	`HashTable` Implementation (Probing, Chaining, Rehash)	40
Total (Part 1)		**100**

Part II

Part 2: The Analytics Engine

5 Introduction & Architectural Evolution

Welcome back, developers. In Programming Assignment 1, you successfully built the foundational backend for a social media platform. While functional for storing data, the simple `FollowList` is highly inefficient for deep, network-wide analysis.

In Programming Assignment 3, you will upgrade the system from a simple database to a powerful analytics engine by replacing the decentralized `FollowList` with three new, centralized graph data structures. You will be provided with the corrected, functional base code from PA1. Your task is to build these graph components, which together form a new suite of analytical capabilities.

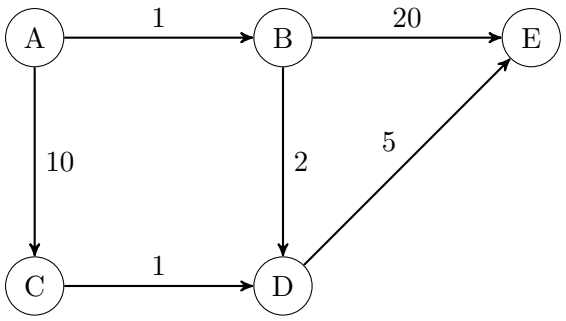
6 Algorithm Preliminaries

This section provides a high-level overview and a dry run of the major algorithms you will be implementing. You should refer back to this section when you begin implementing the corresponding functions.

6.1 Dijkstra's Algorithm

Description: Dijkstra's algorithm finds the shortest path from a single source node to all other nodes in a weighted graph where all edge weights are non-negative. It works by maintaining a set of visited nodes and a map of distances from the source. It greedily selects the unvisited node with the smallest known distance and explores its neighbors, updating their distances if a shorter path is found. A priority queue is used to make this selection efficient.

Dry Run: Consider the graph below. We want to find the shortest path from A to E.



Step	Visit	Distances {A:0, B:∞, C:∞, D:∞, E:∞}	Priority Queue
1	A	Update B:1, C:10	{ (1,B), (10,C) }
2	B	Update D: (1+2)=3, E:(1+20)=21	{ (3,D), (10,C), (21,E) }
3	D	Update E: min(21, 3+5)=8	{ (8,E), (10,C) }
4	E	**Goal reached!**	{ (10,C) }

The path is reconstructed backwards from E using parent pointers: E ← D ← B ← A. The shortest path is A → B → D → E with a total cost of 8.

6.2 Kruskal's Algorithm & DSU

Description: Kruskal's algorithm finds a Minimum Spanning Tree (MST) for a weighted, undirected graph. An MST is a subset of the edges that connects all vertices together with the minimum possible total edge weight, without forming any cycles. The algorithm works by sorting all edges by weight and greedily adding the next cheapest edge, as long as it does not form a cycle. A Disjoint Set Union (DSU) data structure is used to efficiently detect if adding an edge would connect two already-connected components (i.e., form a cycle).

6.3 PageRank Algorithm

Description: PageRank calculates a "centrality" or "influence" score for each node in a directed graph. The core idea is that a node is more influential if it is linked to by other influential nodes. It's an iterative algorithm where each node's rank is repeatedly updated based on the ranks of the nodes that point to it.

Formula: The rank of a user U, with damping factor d and total users N, is given by: $PR(U) = \frac{1-d}{N} + d \times \sum_{V \in \text{Followers}(U)} \frac{PR(V)}{\text{Out-Degree}(V)}$

6.4 Kahn's Algorithm (Topological Sort)

Description: Topological Sort produces a linear ordering of nodes in a Directed Acyclic Graph (DAG) such that for every directed edge from node A to node B, node A comes before node B in the ordering. Kahn's algorithm works by first finding all nodes with an "in-degree" of 0. It adds these to a queue. Then, it repeatedly dequeues a node, adds it to the sorted list, and "removes" its outgoing edges by decrementing the in-degree of its neighbors. Any neighbor whose in-degree becomes 0 is then added to the queue.

7 The `SocialGraph` Class

*Your first task is to model the "who follows whom" network. This graph is **directed** and **unweighted**. Your implementation must use an Adjacency List.*

SocialGraph
- adjList: `std::unordered_map<int, std::vector<int>>`
// Core Graph Management + addVertex(int node) + removeVertex(int node) + addEdge(int from, int to, int weight = 1) + removeEdge(int from, int to) + hasEdge(int from, int to) const + getAdjacent(int from) const *// Core Algorithm* + findShortestPath(int start, int end) const: `std::optional<...>` *// Advanced Analysis* + findEchoChambers() const: `std::vector<std::pair<int, int>>` + calculatePageRank(...): `std::unordered_map<...>` + findCommunities() const: `std::vector<std::vector<int>>`

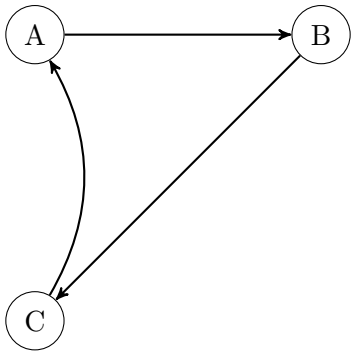


Figure 1: An instance of a SocialGraph showing A follows B, B follows C, and C follows A.

7.1 Core Graph Management

These are the foundational methods for building and modifying your graph.

```
1 void addVertex(int node);
```

- **Task:** Adds a new, isolated user (vertex) to the graph.

```
1 void removeVertex(int node);
```

- **Task:** Removes a user (vertex) and all associated edges (both incoming and outgoing) from the graph.

```
1 void addEdge(int from, int to, int weight = 1);
```

- **Task:** Creates a **directed "follow"** edge from user `from` to user `to`. This is a one-way operation. The `weight` parameter is ignored for this unweighted graph.

```
void removeEdge(int from, int to);
```

- **Task:** Removes the specific directed "follow" edge from `from` to `to`.

```
bool hasEdge(int from, int to) const;
```

- **Task:** Checks if a directed "follow" edge exists from `from` to `to`.

```
std::vector<int> getAdjacent(int from) const;
```

- **Task:** Returns a list of all user IDs that `from` directly follows.

7.2 Core Algorithm: Pathfinding

The marketing team needs a tool to find the shortest chain of follows that connects any two users.

```
std::optional<std::vector<int>> findShortestPath(int start, int end) const;
```

- **Task:** Calculates the shortest path between two users in terms of number of "hops" (i.e., the minimum number of follows). Since this is an unweighted graph, a breadth-first traversal is a suitable approach.
- **Returns:** A vector of user IDs for the path, or `std::nullopt` if no path exists.

7.3 Advanced Analysis: Echo Chamber Detection

This function finds mutual "follow-back" relationships.

```
std::vector<std::pair<int, int>> findEchoChambers() const;
```

- **Task:** Detects all pairs of users (A, B) where A follows B **AND** B follows A.
- **Returns:** A vector of pairs, where each pair represents a mutual follow relationship.

7.4 Advanced Analysis: Influence Ranking

This function will calculate a sophisticated influence score for every user using the iterative **PageRank algorithm**.

```
std::unordered_map<int, double> calculatePageRank(double damping = 0.85, int iterations = 10) const;
```

- **Task:** To identify true "influencers" where a follow from a major celebrity counts more than a new user, based on the PageRank algorithm.
- **Algorithm Pointers:**
  - Initialize the rank of all users to `1.0 / num_users`.
  - For each iteration, calculate a new rank for every user based on the sum of contributions from their followers (users who link to them).

- The contribution from a follower is their own `rank / out-degree`. Apply the full PageRank formula shown in the Preliminaries.

- **Returns:** A map from `userID` to their final PageRank score.

7.5 Advanced Analysis: Community Detection

This function must identify "communities"—groups of users who are all highly interconnected—by finding the **Strongly Connected Components (SCCs)**.

```
std::vector<std::vector<int>> findCommunities() const;
```

- **Task:** To find cohesive sub-networks where every user can reach every other user in that same group through some chain of follows.

- **Algorithm Pointers:** This is a challenging algorithm that requires multiple DFS passes. One common approach involves:

  - **Pass 1:** Perform a first DFS pass on the original graph to build a stack of nodes in order of finishing times.

  - **Pass 2:** Compute the "transposed" graph (reverse all edges).

  - **Pass 3:** Perform a second DFS pass on the transposed graph, visiting nodes in the order from the stack, to identify the SCCs.

- **Returns:** A vector of vectors, where each inner vector is a community of userIDs.

8 The `GeographicNetwork` Class

*Your next task is to model the physical server network. This graph is **undirected** and **weighted**. When adding an edge, you must add it in **both directions**.*

GeographicNetwork
- adjList: `std::unordered_map<int, std::vector<Edge>>`
// Core Graph Management + addVertex(int node) + removeVertex(int node) + addEdge(int from, int to, int weight) + removeEdge(int from, int to) + hasEdge(int from, int to) const + getAdjacent(int from) const *// Core Algorithms* + findShortestPath(int start, int end) const: `std::optional<...>` + calculateMinimumSpanningTree() const: `std::vector<MstEdge>` *// Advanced Analysis* + findCriticalNodes() const: `std::vector<int>` + findPathWithMinEffort(int start, int end) const: `std::optional<...>` + findBestCity(int distanceThreshold) const: int

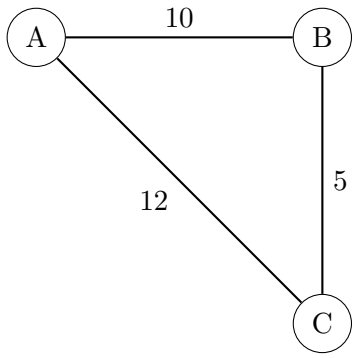


Figure 2: An instance of a GeographicNetwork showing weighted, undirected connections.

8.1 Core Graph Management

These methods manage the physical structure of your weighted, undirected network.

```
1 void addVertex(int node);
```

- **Task:** Adds a new, isolated city (vertex) to the network.

```
1 void removeVertex(int node);
```

- **Task:** Removes a city (vertex) and all physical connections (edges) to and from it.

```
1 void addEdge(int from, int to, int weight);
```

- **Task:** Creates a **bi-directional (undirected)** connection between `from` and `to` with a latency (cost) of `weight`. This is a two-way operation.

```
void removeEdge(int from, int to);
```

- **Task:** Removes the bi-directional connection between `from` and `to`. This is also a two-way operation.

```
bool hasEdge(int from, int to) const;
```

- **Task:** Checks if a direct connection exists between `from` and `to`.

```
std::vector<int> getAdjacent(int from) const;
```

- **Task:** Returns a list of all city IDs directly connected to `from`.

8.2 Core Algorithm: Lowest Latency Routing

Your function must find the absolute fastest route for a message by calculating the path with the minimum possible **sum of latencies (weights)**.

```
std::optional<std::vector<int>> findShortestPath(int start, int end) const;
```

- **Task:** Find the path with the minimum total cost.
- **Algorithm Pointers:** This function must find the shortest path in a *weighted* graph. A standard approach involves iteratively exploring nodes, always picking the unvisited node with the smallest known distance from the start, and updating its neighbors' distances. A priority queue is essential for doing this efficiently.
- **Returns:** A vector of node IDs for the path, or `std::nullopt` if no path exists.

8.3 Core Algorithm: Network Backbone Design

This function must find the optimal network layout by producing a **Minimum Spanning Tree (MST)**.

```
std::vector<MstEdge> calculateMinimumSpanningTree() const;
```

- **Task:** Design the cheapest possible network backbone that connects all locations.
- **Algorithm Pointers:** A common approach to find an MST is to process all edges in increasing order of weight. You add an edge to your tree as long as it doesn't form a cycle. You will need an efficient way to check if two nodes are already in the same component (i.e., to detect cycles).
- **Returns:** A vector of `MstEdge` structs representing the edges in the final MST.

8.4 Advanced Analysis: Critical Node Identification

```
std::vector<int> findCriticalNodes() const;
```

- **Task:** To identify all "critical cities" that represent single points of failure. A critical node (or articulation point) is one whose removal increases the number of connected components in the graph.
- **Algorithm Pointers:** This requires a single DFS-based traversal. During the traversal, you must track the "discovery time" and a "low-link" value for each node to determine if any of its subtrees are "cut off" from the rest of the graph if this node is removed.
- **Returns:** A vector of node IDs that are critical points of failure.

8.5 Advanced Analysis: Minimum Effort Path

```
std::optional<std::vector<int>> findPathWithMinEffort(int start, int end) const;
```

- **Task:** Find a path that minimizes the "bottleneck latency". Unlike the standard shortest path problem which minimizes the *sum* of latencies, this function seeks to minimize the *single worst-case* latency on any one leg of the journey.
- **Algorithm Pointers:** This can be solved with a modified shortest path algorithm. The "cost" or "distance" to a node should not be the *sum* of the path, but rather the *maximum edge weight* seen so far on the path to that node. A priority queue can be used to explore paths efficiently based on this "effort" cost.
- **Returns:** An std::optional containing the path with the minimum effort, or `std::nullopt` if no path exists.

8.6 Advanced Analysis: Finding the Optimal City

```
int findBestCity(int distanceThreshold) const;
```

- **Task:** Finds the city with the smallest number of neighbors reachable within a given `distanceThreshold`.
- **Algorithm Pointers:** This requires knowing the shortest path between *all pairs* of nodes. You'll need an algorithm that can compute this all-pairs shortest path matrix. Once you have the matrix, iterate through each city to count its reachable neighbors.
- **Returns:** The ID of the optimal city. Returns -1 if the graph is empty.

9 The `InteractionGraph` Class

*This part focuses on modeling user engagement with content. This is a **bipartite graph** with two types of nodes (Users and Posts). Edges are **directed** (from User to Post) and **weighted**.*

InteractionGraph
- userToPostEdges: `std::unordered_map<int, std::vector<Interaction>>`
- postToUserEdges: `std::unordered_map<int, std::vector<Interaction>>`
// Core Graph Management
+ addVertex(int id, NodeType type)
+ removeVertex(int id, NodeType type)
+ addInteraction(int userID, int postID, int weight)
// Advanced Analysis
+ findSimilarUsers(int userID, int topN) const: `std::vector<...>`
+ recommendPosts(int userID, int topN) const: `std::vector<int>`
+ calculateTrendScores(...): `std::unordered_map<...>`
+ getProcessingOrder() const: `std::optional<...>`

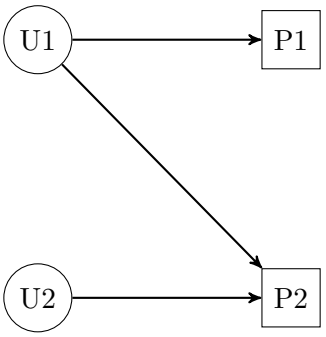


Figure 3: An instance of an InteractionGraph. Users U1 and U2 interact with Posts P1 and P2.

9.1 Core Graph Management

This graph is bipartite, so its management functions are unique. It maintains two separate adjacency lists.

```
1 void addVertex(int id, NodeType type);
```

- **Task:** Adds a new vertex of a specific `type` (`USER` or `POST`) to the graph.
- **Hint:** This should add an entry to the appropriate map (`userToPostEdges` or `postToUserEdges`).

```
1 void removeVertex(int id, NodeType type);
```

- **Task:** Removes a vertex (and all its interactions) from the graph.
- **Hint:** This is the most complex management function. If you remove a `USER`, you must also iterate through the `postToUserEdges` map to remove them from every post's interaction list. If you remove a `POST`, you must do the reverse.

```
1 void addInteraction(int userID, int postID, int weight);
```

- **Task:** Creates a weighted, directed edge from a `userID` to a `postID`.
- **Hint:** This must update *both* adjacency lists to keep them in sync. Add the post to the user's list, and add the user to the post's list.

9.2 Advanced Analysis: Finding Digital Twins

```
std::vector<std::pair<int, double>> findSimilarUsers(int userID, int topN) const;
```

- **Task:** To quantify how "similar" two users' tastes are. In this context, a user's 'taste' is defined purely by the **set of posts** they have interacted with.
- **Algorithm Pointers:**
  - For the target user and every other user, build a set of post IDs they have interacted with.
  - For each pair, calculate the **Jaccard Similarity**: size of the *intersection* of their sets divided by the size of the *union* of their sets.
- **Returns:** A vector of pairs {`similarUserID, similarityScore`}, sorted by score.

9.3 Advanced Analysis: The Content Recommender

```
std::vector<int> recommendPosts(int userID, int topN) const;
```

- **Task:** Suggest new posts to a user based on the activity of their "digital twins".
- **Algorithm Pointers:** Implement **collaborative filtering**.
  - First, call your `findSimilarUsers` function to get a small group of users with similar tastes. For testing purposes, you **must use 5** for this number.
  - Aggregate all the posts this group has interacted with.
  - Filter out any posts the original user has already seen.
  - Rank the remaining posts by a "recommendation score", where a post gets more points if it was liked by a more similar user.
- **Returns:** A vector of the `topN` recommended post IDs.

9.4 Advanced Analysis: Identifying Viral Content

```
std::unordered_map<int, double> calculateTrendScores(const std::unordered_map<int, double>& pageRanks) const;
```

- **Task:** To calculate a "trend score" for each post by weighting interactions by the influence of the user who performed them.
- **Algorithm Pointers:** This is a synthesis task.
  - Iterate through every post in your `postToUserEdges` map.
  - For each post, iterate through all the users who have interacted with it.
  - Get the user's influence score from the input `pageRanks` map (which is the **output from your `SocialGraph`'s `calculatePageRank` function**).

- The post's score is the sum of `(user_pageRank * interaction_weight)` for all its interactions.

- **Returns:** A map of {`postID -> trendScore`}.

9.5 Advanced Analysis: Dependency Scheduling

```
std::optional<std::vector<int>> getProcessingOrder() const;
```

- **Task:** Model a dependency system where posts rely on user interactions and find a valid processing order (Users first, then Posts).

- **Algorithm Pointers:** Perform a **topological sort**. A common method involves using a queue and tracking the "in-degree" (number of incoming edges) for each node.
  - In this bipartite graph, all `USER` nodes will have an in-degree of 0. All `POST` nodes will have an in-degree ¿ 0.
  - Initialize your queue with all the `USER` nodes and proceed with the sort.

- **Returns:** A vector of node IDs (both users and posts) in a valid topological order, or `std::nullopt` if a cycle is detected.

10 Marks Distribution (Part 2: The Analytics Engine)

Part	Component / Task	Marks
2.1: SocialGraph (30)		**30**
	Core Management & `findShortestPath`	10
	Influence Analysis (`calculatePageRank` & `findEchoChambers`)	10
	`findCommunities` (SCCs)	10
2.2: GeographicNetwork (30)		**30**
	Core Management & `findShortestPath`	10
	`calculateMinimumSpanningTree` (Kruskal's)	10
	Advanced Analysis (`findCriticalNodes`, `findPathWithMinEffort`, `findBestCity`)	10
2.3: InteractionGraph (40)		**40**
	Core Management (Bipartite Graph)	5
	Similarity & Recommendations (`findSimilarUsers` & `recommendPosts`)	20
	Trend Analysis & Scheduling (`calculateTrendScores` & `getProcessingOrder`)	15
Total		**100**

How Your Score is Calculated

When you run a test suite (e.g., via the `make` command), you receive a **Raw Score** based on the points for each passed test. This raw score is then converted to the final weighted marks for that component.

Formula: `Your Marks = (Your Raw Score / Max Raw Score) * Final Weight`

Example:

- You run the `SocialGraph` test suite and earn a Raw Score of 90 / 100.
- The `SocialGraph` component has a Final Weight of 30 Marks.
- Your final marks for this part would be: `(90 / 100) * 30 = 27 Marks`.

The `make test` command runs all suites, and the final summary table performs this calculation for you, showing you the grand total out of 100.

11 Testing Your Implementation

A robust testing system has been provided to help you validate your implementation. You can run tests for the entire project, for specific parts, or even for individual components. All commands should be run from the root directory of your project (`CS202_PA3_SOLUTION`). All code must be run on WSL or UNIX systems the test suite wont run on windows

11.1 Running the Complete Test Suite

To compile and run all tests for both Part 1 and Part 2 and see a final grand total score, use the main test command.

- `make test`

11.2 Running Part-Specific Suites

If you want to focus on one part of the assignment and see a sub-total for just that part, use the following commands.

- `make part1`: Runs all tests for the Heaps and Hash Tables and shows a total score for Part 1.
- `make part2`: Runs all tests for the Graph components and shows a total score for Part 2.

11.3 Running Individual Component Tests

These commands are useful for debugging a specific data structure. They will compile and run only the tests for that single component.

- `make test-heap`: Runs only the MinHeap tests.
- `make test-hash`: Runs only the HashTable tests.
- `make test-social-media`: Runs only the SocialMediaSystem tests.
- `make test-social-graph`: Runs only the SocialGraph tests.
- `make test-geographic-network`: Runs only the GeographicNetwork tests.
- `make test-interaction-graph`: Runs only the InteractionGraph tests.

11.4 Cleaning Up

To remove all compiled files and the `bin` directory, use the clean command. This is useful for ensuring a fresh build.

- `make clean`

12 Submission Instructions

Please follow these instructions carefully to ensure your submission is graded correctly.

12.1 File Naming and Format

Your final submission must be a single compressed ZIP file. The file must be named according to your roll number.

- **Format:** `<roll_number>_PA3.zip`
- **Example:** `25100123_PA3.zip`

12.2 Directory Structure

The contents of your ZIP file must match the following structure exactly. Only the 'Part1' and 'Part2' folders should be at the root of the zip. Inside each, you should only include your 'headers' and 'src' directories.

Do NOT include your `Makefile`, any test files, runner files, compiled executables, or the `bin` directory.

```
<roll_number>_PA3.zip
|
|-- Part1/
|   |-- headers/
|   |   |-- HashTables.h
|   |   |-- Heap.h
|   |   `-- SocialMediaSystem.h
|   |
|   `-- src/
|       |-- HashTables.cpp
|       |-- Heap.cpp
|       `-- SocialMediaSystem.cpp
|
`-- Part2/
    |-- include/
    |   |-- geographic_network.h
    |   |-- graph.h
    |   |-- interaction_graph.h
    |   |-- linked_list.h
    |   |-- post.h
    |   |-- post_list.h
    |   |-- post_pool.h
    |   |-- social_graph.h
    |   `-- user.h
    |
    `-- src/
        |-- geographic_network.cpp
        |-- interaction_graph.cpp
        |-- post_list.cpp
        |-- post_pool.cpp
        |-- social_graph.cpp
        `-- user.cpp
```

Your code is more than just an implementation. It is an implementation (hehe)
Good luck