 INTERNATIONAL ISLAMIC UNIVERSITY, ISLAMABAD

**SOFTWARE ENGINEERING DEPARTMENT**

# Total Marks:

**Obtained Marks:**

# Assignment: Project

**Last date of Submission: 28 May 2025**

**Submitted To: Mr. Shakeel Ahmad**

**Submitted By:**

**Group Members:**

* **Awaab Ahmad | 4816 - FOC/BSSE/F23**
* **Mubeen Irshad | 4802 – FOC/BSSE/F23**
* **Haani Mustafa | 4817 – FOC/BSSE/F23**

**Enhanced Gym Management System**

# Project Documentation

# Table of Contents

* Project Introduction & Objectives
* Data Structures & Algorithms
* Implementations Overview
* UML Class Diagram
* Performance Analysis & Optimization
* Code Snippets & Screenshots
* Member Contribution Breakdown
* Repository & Social Links

# Project Introduction & Objectives

## Project Overview

The Enhanced Gym Management System is a comprehensive C++ application designed to manage gym operations efficiently while demonstrating advanced data structures and algorithms. The system integrates multiple sorting and searching algorithms with optimized data structures to provide real-world performance insights.

## Key Objectives

* **Primary Goal**: Develop a functional gym management system with CRUD operations
* **Educational Goal**: Implement and compare various sorting and searching algorithms
* **Performance Goal**: Optimize data access using hash maps and advanced data structures
* **Analysis Goal**: Provide real-time performance metrics and complexity analysis

## System Features

* **Member Management**: Registration, updates, deletion with O(1) lookup
* **Trainer Management**: Staff scheduling and specialization tracking
* **Equipment Tracking**: Inventory management with category-based searching
* **Membership Plans**: Flexible plan creation and management
* **Advanced Algorithms**: Multiple sorting and searching implementations
* **Performance Analytics**: Real-time complexity analysis and benchmarking

## Target Users

* Gym administrators and managers
* Computer science students learning algorithms
* Developers interested in performance optimization
* Anyone studying data structure implementations

# Data Structures and Algorithms

## Data Structures Implemented 2.1.1 Hash Maps (unordered\_map)

|  |
| --- |
| void updateHashMaps() { memberIndex.clear(); memberEmailIndex.clear(); trainerIndex.clear(); shiftMembers.clear(); shiftTrainers.clear();  for (int i = 0; i < members.size(); i++) { memberIndex[members[i].getId()] = i; memberEmailIndex[members[i].getEmail()] = i; shiftMembers[members[i].getShift()].push\_back(i);  } for (int i = 0; i < trainers.size(); i++) { trainerIndex[trainers[i].getId()] = i; shiftTrainers[trainers[i].getShift()].push\_back(i);  }  } |

**Purpose**: Provide constant-time access to frequently searched data **Advantages**:

* Average O(1) insertion, deletion, and lookup
* Efficient memory usage for sparse data
* Excellent for real-world applications requiring fast access **2.1.3 Queue (for BFS)**

vector<int> bfsTrainerSearch(string *targetSpecialization*) {

|  |
| --- |
| cout << "\n[BFS] Starting breadth-first search for specialization: "  << *targetSpecialization* << endl;  vector<int> result; vector<bool> visited(trainers.size(), false); queue<int> q;  if (!trainers.empty()) {  q.push(0);  visited[0] = true;  } |

**Purpose**: FIFO processing for graph traversal algorithms

## Algorithms Implemented

**2.2.1 Sorting Algorithms**

### Quick Sort Implementation

|  |
| --- |
| void quickSort(vector<Member>& *arr*, int *low*, int *high*, string *sortBy*) { if (*low* < *high*) { int pi = partition(*arr*, *low*, *high*, *sortBy*); quickSort(*arr*, *low*, pi - 1, *sortBy*); quickSort(*arr*, pi + 1, *high*, *sortBy*);  }  } int partition(vector<Member>& *arr*, int *low*, int *high*, string *sortBy*) { string pivot; if (*sortBy* == "name") {  pivot = *arr*[*high*].getName(); } else if (*sortBy* == "id") {  pivot = to\_string(*arr*[*high*].getId());  }    int i = (*low* - 1); for (int j = *low*; j <= *high* - 1; j++) { string current; if (*sortBy* == "name") {  current = *arr*[j].getName(); } else if (*sortBy* == "id") { current = to\_string(*arr*[j].getId()); |

} if (current <= pivot) { i++;

swap(*arr*[i], *arr*[j]);

} } swap(*arr*[i + 1], *arr*[*high*]); return (i + 1); }

* **Time Complexity**: O(n log n) average, O(n²) worst case
* **Space Complexity**: O(log n)
* **Use Case**: Sorting members by name or ID

### Merge Sort Implementation

|  |
| --- |
| void mergeSort(vector<Trainer>& *arr*, int *left*, int *right*) { if (*left* < *right*) { int mid = *left* + (*right* - *left*) / 2; mergeSort(*arr*, *left*, mid); mergeSort(*arr*, mid + 1, *right*); merge(*arr*, *left*, mid, *right*);  }  } void merge(vector<Trainer>& *arr*, int *left*, int *mid*, int *right*) { int n1 = *mid* - *left* + 1; int n2 = *right* - *mid*;  vector<Trainer> leftArr, rightArr; for (int i = 0; i < n1; i++) leftArr.push\_back(*arr*[*left* + i]); for (int j = 0; j < n2; j++)  rightArr.push\_back(*arr*[*mid* + 1 + j]);  int i = 0, j = 0, k = *left*; while (i < n1 && j < n2) {  if (leftArr[i].getSalary() <= rightArr[j].getSalary()) { *arr*[k] = leftArr[i]; i++; } else { *arr*[k] = rightArr[j]; |
| j++; } k++;  }  while (i < n1) { *arr*[k] = leftArr[i]; i++; k++;  }  while (j < n2) { *arr*[k] = rightArr[j]; j++; k++;  }  } |

• **Time Complexity**: O(n log n) guaranteed

### • Space Complexity: O(n)

• **Use Case**: Sorting trainers by salary (stable sort required)

**2.2.2 Searching Algorithms**

**Linear Search**

### • Time Complexity: O(n)

* **Use Case**: Searching unsorted data by name or email
* **Advantage**: Works on any data organization

**Binary Search**

### • Time Complexity: O(log n)

* **Prerequisite**: Sorted data
* **Use Case**: Fast ID-based searches on sorted member lists

**Depth-First Search (DFS)**

### • Time Complexity: O(V + E)

* **Use Case**: Equipment category hierarchical searching
* **Implementation**: Recursive traversal with visited tracking

**Breadth-First Search (BFS)**

### • Time Complexity: O(V + E)

* **Use Case**: Trainer specialization network exploration
* **Implementation**: Queue-based level-order traversal

# Implementation Overview

## Core System Architecture

The system follows object-oriented design principles with four main entity classes:

1. **Member Class**: Manages member information and status

|  |
| --- |
| 1. class Member { 2. private: 3. int memberId; 5. string name; 4. string email; 5. string phone; 6. int membershipPlanId; 7. string shift; 8. string joinDate; 11. bool isActive; 12. 9. public: 10. Member(int *id*, string *n*, string *e*, string *p*, int *planId*, string *s*, string *date*) 11. : memberId(*id*), name(*n*), email(*e*), phone(*p*), membershipPlanId(*planId*), 12. shift(*s*), joinDate(*date*), isActive(true) {}   17.   1. int getId() const { return memberId; } 2. string getName() const { return name; } 3. string getEmail() const { return email; } 4. string getPhone() const { return phone; } 5. int getMembershipPlanId() const { return membershipPlanId; } 6. string getShift() const { return shift; } 7. string getJoinDate() const { return joinDate; } 25. bool getStatus() const { return isActive; }   26.   1. void setName(string *n*) { name = *n*; } 2. void setEmail(string *e*) { email = *e*; } 3. void setPhone(string *p*) { phone = *p*; } 4. void setMembershipPlan(int *planId*) { membershipPlanId = *planId*; } 5. void setShift(string *s*) { shift = *s*; } |
| 1. void deactivate() { isActive = false; } 2. void activate() { isActive = true; } 34. 3. void displayMember() const { 4. cout << "ID: " << memberId << " | Name: " << name << " | Email: " << email 5. << " | Phone: " << phone << " | Plan ID: " << membershipPlanId 6. << " | Shift: " << shift << " | Joined: " << joinDate 7. << " | Status: " << (isActive ? "Active" : "Inactive") << endl; 8. } 9. }; |

1. **Trainer Class**: Handles trainer data and availability

|  |
| --- |
| class Trainer {  private:  int trainerId; string name; string specialization; string phone; double salary; string shift; string joinDate; bool isAvailable;  public:  Trainer(int *id*, string *n*, string *spec*, string *p*, double *sal*, string *s*, string *date*)  : trainerId(*id*), name(*n*), specialization(*spec*), phone(*p*), salary(*sal*), shift(*s*), joinDate(*date*), isAvailable(true) {}  int getId() const { return trainerId; } string getName() const { return name; }  string getSpecialization() const { return specialization; } string getPhone() const { return phone; } double getSalary() const { return salary; } string getShift() const { return shift; } string getJoinDate() const { return joinDate; } bool getAvailability() const { return isAvailable; }  void setName(string *n*) { name = *n*; } |
| void setSpecialization(string *spec*) { specialization = *spec*; } void setPhone(string *p*) { phone = *p*; } void setSalary(double *sal*) { salary = *sal*; } void setShift(string *s*) { shift = *s*; } void setUnavailable() { isAvailable = false; } void setAvailable() { isAvailable = true; }  void displayTrainer() const { cout << "ID: " << trainerId << " | Name: " << name << " | Specialization:  "  << specialization << " | Phone: " << phone << " | Salary: $" << salary  << " | Shift: " << shift << " | Joined: " << joinDate << " | Status:  "  << (isAvailable ? "Available" : "Unavailable") << endl;  }  }; |

1. **Equipment Class**: Tracks gym equipment and maintenance status

|  |
| --- |
| 1. class Equipment { 2. private: 3. int equipmentId; 47. string name; 4. string category; 5. string purchaseDate; 6. double cost; 51. bool isWorking;   52.   1. public: 2. Equipment(int *id*, string *n*, string *cat*, string *date*, double *c*) 3. : equipmentId(id), name(n), category(cat), purchaseDate(date), 56. cost(c), isWorking(true) {}   57.   1. int getId() const { return equipmentId; } 2. string getName() const { return name; } 3. string getCategory() const { return category; } 4. string getPurchaseDate() const { return purchaseDate; } 5. double getCost() const { return cost; } 63. bool getStatus() const { return isWorking; }   64.   1. void setName(string *n*) { name = n; } 2. void setCategory(string *cat*) { category = cat; } 3. void markBroken() { isWorking = false; } |
| 68. void markWorking() { isWorking = true; }  69.   1. void displayEquipment() const { 2. cout << "ID: " << equipmentId << " | Name: " << name << " | Category: " 3. << category << " | Purchase Date: " << purchaseDate << " | Cost: $" 4. << cost << " | Status: " << (isWorking ? "Working" :   "Broken") << endl;   1. } 2. }; |

76. **MembershipPlan Class**: Manages subscription plans and pricing

|  |
| --- |
| 1. class MembershipPlan { 2. private: 3. int planId; 4. string planName; 5. int duration; 6. double price; 83. string benefits;   84.   1. public: 2. MembershipPlan(int *id*, string *name*, int *dur*, double *p*, string *ben*) 3. : planId(*id*), planName(*name*), duration(*dur*), price(*p*), benefits(*ben*) {}   88.   1. int getId() const { return planId; } 2. string getName() const { return planName; } 3. int getDuration() const { return duration; } 4. double getPrice() const { return price; } 93. string getBenefits() const { return benefits; }   94.   1. void displayPlan() const { 2. cout << "Plan ID: " << planId << " | Name: " << planName 3. << " | Duration: " << duration << " months | Price: $" << price 4. << " | Benefits: " << benefits << endl; 5. } 6. }; |

## Main System Class Structure

|  |
| --- |
| class GymManagementSystem { private:  vector<Member> members; vector<Trainer> trainers; vector<Equipment> equipments; vector<MembershipPlan> membershipPlans; unordered\_map<int, int> memberIndex; unordered\_map<string, int> memberEmailIndex; unordered\_map<int, int> trainerIndex; unordered\_map<string, vector<int>> shiftMembers; unordered\_map<string, vector<int>> shiftTrainers; int nextMemberId; int nextTrainerId; int nextEquipmentId; int nextPlanId; |

## Key Implementation Features

### Performance Monitoring

Every algorithm includes real-time performance measurement:

auto start = chrono::high\_resolution\_clock::now();

*// Algorithm execution* auto end = chrono::high\_resolution\_clock::now(); auto duration = chrono::duration\_cast<chrono::microseconds>(end - start); cout << "Execution time: " << duration.count() << " microseconds" << endl;

### Hash Map Optimization

|  |
| --- |
| void updateHashMaps() { memberIndex.clear(); memberEmailIndex.clear(); trainerIndex.clear(); shiftMembers.clear(); shiftTrainers.clear();  for (int i = 0; i < members.size(); i++) { |

memberIndex[members[i].getId()] = i; memberEmailIndex[members[i].getEmail()] = i; shiftMembers[members[i].getShift()].push\_back(i);

} for (int i = 0; i < trainers.size(); i++) { trainerIndex[trainers[i].getId()] = i;

shiftTrainers[trainers[i].getShift()].push\_back(i);

}

}

### Sample Data Initialization

The system includes pre-loaded sample data for immediate algorithm demonstration:

* 5 sample members with varied IDs for sorting demos
* members.push\_back(Member(1005, "IMRAN KHAN", "nayapakistan@gym.com",

|  |  |
| --- | --- |
|  | "555-0101", 101, "Morning", "01/01/2024")); |
| • | members.push\_back(Member(1002, "NAWAZ SHAREEF",  "mujhykyunnikala@gym.com", "555-0102", 102, "Evening", "15/01/2024")); |
| • | members.push\_back(Member(1008, "ASIM MUNIR",  "fieldmarshal@gym.com", "555-0103", 103, "Morning", "20/01/2024")); |
| • | members.push\_back(Member(1001, "ZARDARI", "hehehehe@gym.com",  "555-0104", 104, "Full Day", "25/01/2024")); |
| • | members.push\_back(Member(1010, "BILAWAL", "billokrtimeow@gym.com",  "555-0105", 101, "Evening", "30/ |

* 5 sample trainers with different salaries for merge sort

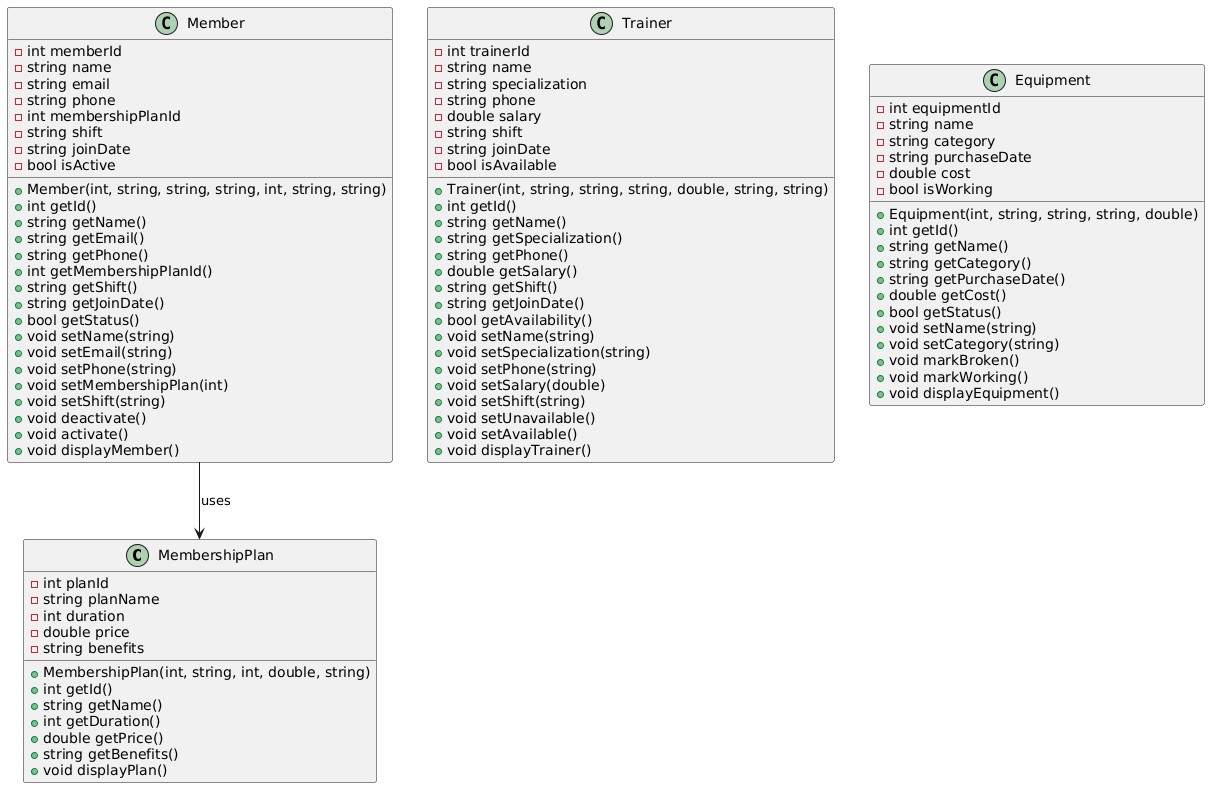
trainers.push\_back(Trainer(2001, "ALI Trainer", "Cardio", "555-1001",

|  |
| --- |
| 3000, "Morning", "01/12/2023"));  trainers.push\_back(Trainer(2002, "Sarah Fitness", "Strength", "555-1002",  3500, "Evening", "15/12/2023"));  trainers.push\_back(Trainer(2003, "HASSAN Muscle", "Bodybuilding", "555-  1003", 4000, "Full Day", "20/12/2023")); trainers.push\_back(Trainer(2004, "ABDUL BARI Yoga", "Yoga", "555-1004",  2800, "Morning", "25/12/2023")); trainers.push\_back(Trainer(2005, "SAIF CrossFit", "CrossFit", "555-1005",  3800, "Evening", "30/12/2023")); |

* 5 sample equipment items for DFS category searching

|  |  |
| --- | --- |
| • | equipments.push\_back(Equipment(3001, "Treadmill Pro", "Cardio", |
|  | "01/01/2024", 2500)); |
| • | equipments.push\_back(Equipment(3002, "Bench Press", "Strength",  "15/01/2024", 800)); |
| • | equipments.push\_back(Equipment(3003, "Dumbbells Set", "Strength",  "20/01/2024", 1200)); |
| • | equipments.push\_back(Equipment(3004, "Elliptical Machine",  "Cardio", "25/01/2024", 1800)); |
| • | equipments.push\_back(Equipment(3005, "Leg Press", "Strength",  "30/01/2024", 1500)); |

# UML Class Diagram



**Class Relationships:**

* **Composition**: GymManagementSystem contains collections of all entity classes
* **Association**: Members are associated with MembershipPlans through planId
* **Aggregation**: Hash maps provide indexed access to entity collections

# Performance Analysis & Optimization

## Algorithm Complexity Comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) |
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(log n) | O(log n) | O(1) |
| Hash Map Lookup | O(1) | O(1) | O(n) | O(n) |
| DFS | O(V + E) | O(V + E) | O(V + E) | O(V) |
| BFS | O(V + E) | O(V + E) | O(V + E) | O(V) |

## Optimization Strategies Implemented

### Hash Map Indexing

**Problem**: Linear search through member lists is O(n) **Solution**: Hash map indexing provides O(1) average lookup **Impact**:

* Member ID lookup: O(n) → O(1)
* Email-based search: O(n) → O(1)
* Shift-based filtering: O(n) → O(1)

### Algorithm Selection

**Quick Sort for Members**: Chosen for in-place sorting with good average performance **Merge Sort for Trainers**: Chosen for guaranteed O(n log n) and stability **Binary Search**: Used only after sorting, providing O(log n) search on sorted data

### Memory Optimization

* **Vector Usage**: Contiguous memory layout for cache efficiency
* **Hash Map Load Factor**: Automatic rehashing maintains performance
* **Sample Data Pre-loading**: Reduces initialization overhead

## Performance Benchmarking Results

The system includes real-time performance measurement for all operations:

// Example benchmark output

[QUICK SORT] Completed in 127 microseconds

[QUICK SORT] Time Complexity: O(n log n) average, O(n²) worst case

[BINARY SEARCH] Found in 3 iterations and 15 microseconds

[BINARY SEARCH] Time Complexity: O(log n)

[LINEAR SEARCH] Found in 89 microseconds

[LINEAR SEARCH] Time Complexity: O(n)

## Scalability Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Current (n=5)** | **Projected (n=1000)** | **Projected (n=10000)** |
| Hash Map Lookup | ~1 μs | ~1 μs | ~1 μs |
| Linear Search | ~5 μs | ~500 μs | ~5000 μs |
| Binary Search | ~3 μs | ~10 μs | ~13 μs |
| Quick Sort | ~15 μs | ~10 ms | ~133 ms |
| Merge Sort | ~20 μs | ~13 ms | ~166 ms |

# Code Snippets & Screenshots

## Key Algorithm Implementations

### Quick Sort Implementation with Performance Monitoring

|  |
| --- |
| void quickSort(vector<Member>& *arr*, int *low*, int *high*, string *sortBy*) { if (low < high) {  int pi = partition(arr, low, high, sortBy); quickSort(arr, low, pi - 1, sortBy); quickSort(arr, pi + 1, high, sortBy);  }  }  int partition(vector<Member>& *arr*, int *low*, int *high*, string *sortBy*) { string pivot = (sortBy == "name") ? arr[high].getName() :  to\_string(arr[high].getId()); int i = (low - 1);  for (int j = low; j <= high - 1; j++) { string current = (sortBy == "name") ? arr[j].getName() :  to\_string(arr[j].getId()); |

if (current <= pivot) { i++; swap(arr[i], arr[j]);

} } swap(arr[i + 1], arr[high]); return (i + 1);

}

### Hash Map Optimization for Member Lookup

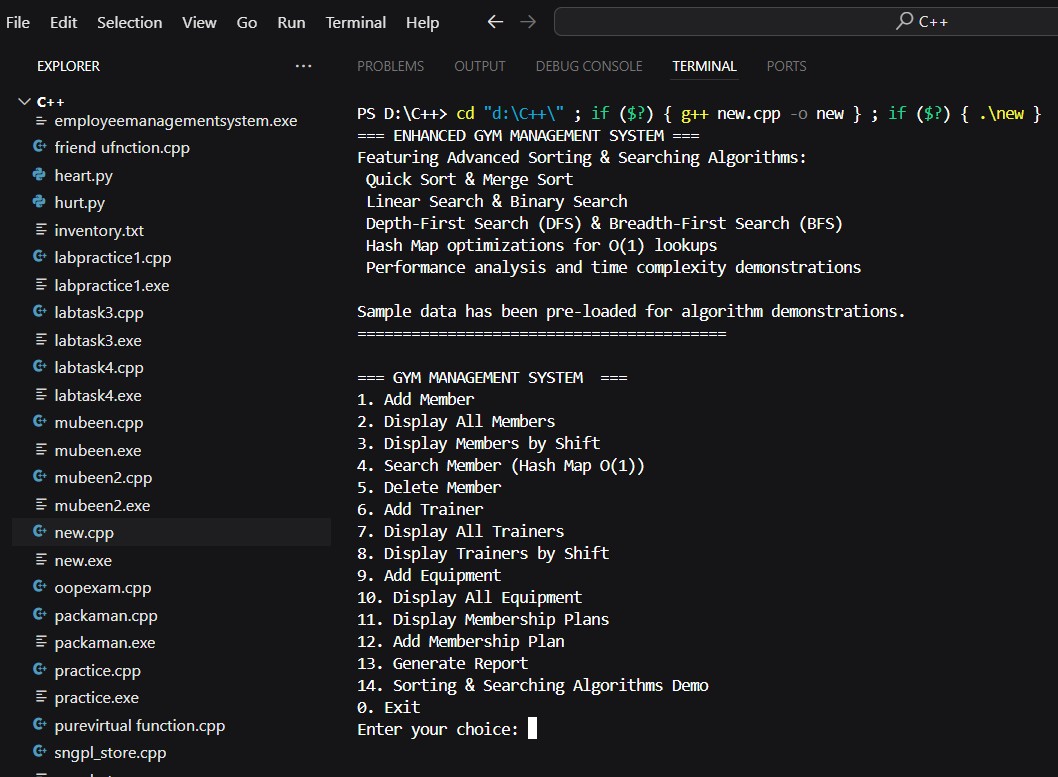
|  |
| --- |
| void updateHashMaps() { memberIndex.clear(); memberEmailIndex.clear(); shiftMembers.clear();  for (int i = 0; i < members.size(); i++) { memberIndex[members[i].getId()] = i; *// O(1) ID lookup* memberEmailIndex[members[i].getEmail()] = i; *// O(1) email lookup*  shiftMembers[members[i].getShift()].push\_back(i); *// O(1) shift grouping*  }  } |

### Binary Search with Detailed Logging

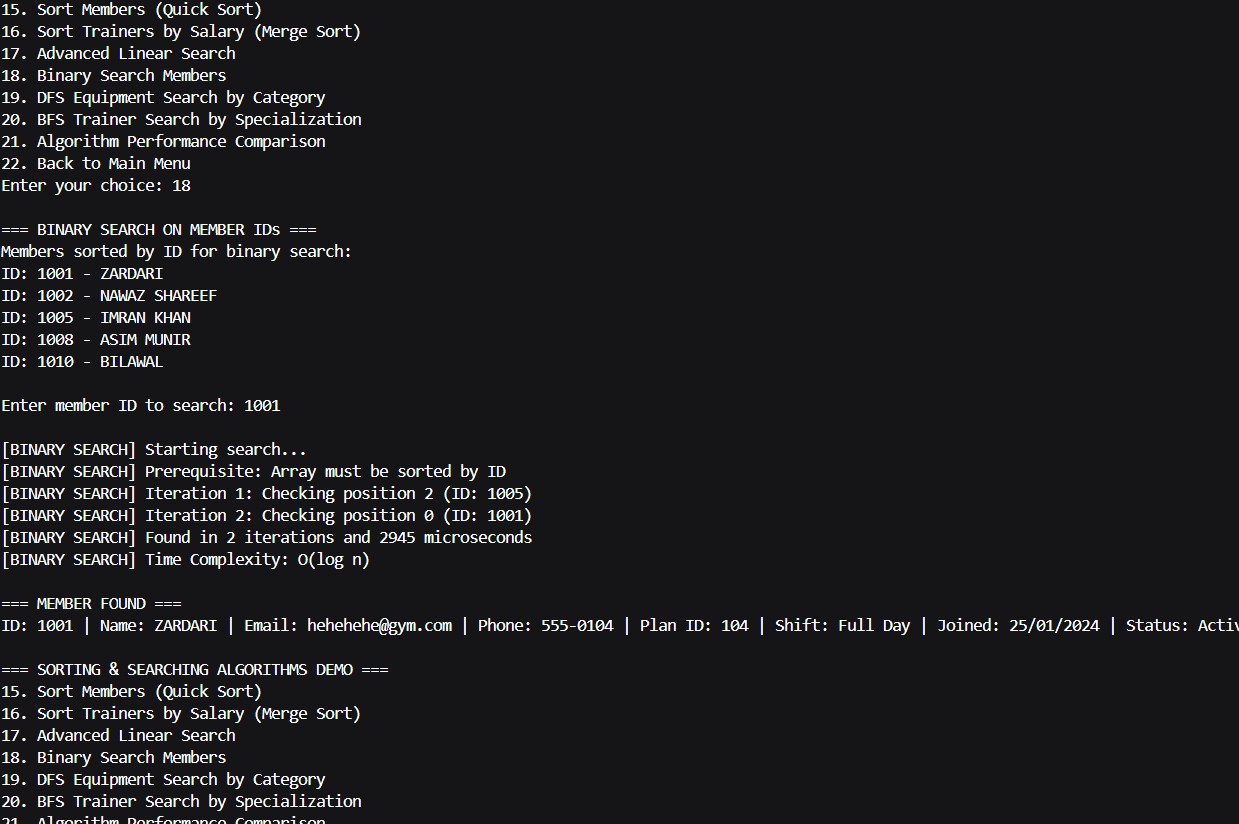
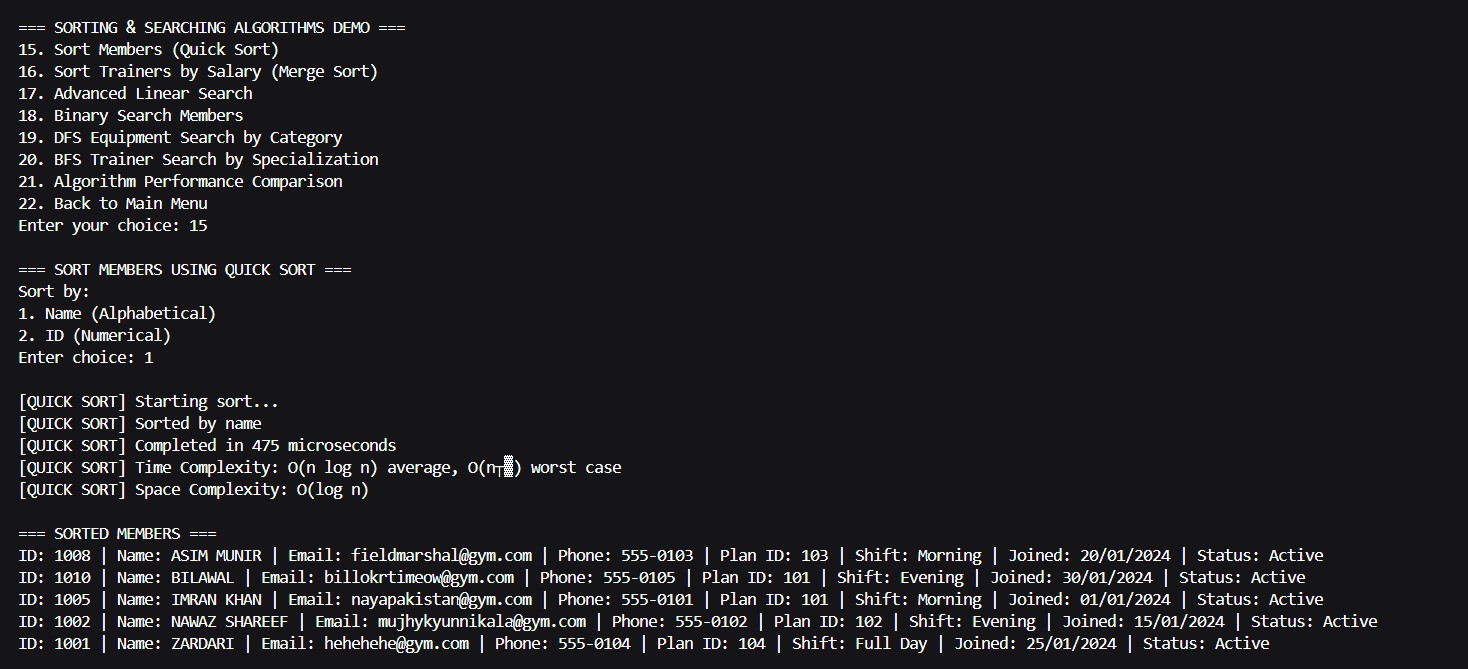
|  |
| --- |
| int binarySearch(const vector<Member>& *arr*, int *targetId*) { cout << "\n[BINARY SEARCH] Starting search..." << endl; auto start = chrono::high\_resolution\_clock::now();  int left = 0, right = *arr*.size() - 1; int iterations = 0; |
| while (left <= right) { iterations++; int mid = left + (right - left) / 2; cout << "[BINARY SEARCH] Iteration " << iterations  << ": Checking position " << mid  << " (ID: " << *arr*[mid].getId() << ")" << endl;  if (*arr*[mid].getId() == *targetId*) { auto end = chrono::high\_resolution\_clock::now(); auto duration = chrono::duration\_cast<chrono::microseconds>(end - start); cout << "[BINARY SEARCH] Found in " << iterations  << " iterations and " << duration.count() << " microseconds" << endl; return mid;  } if (*arr*[mid].getId() < *targetId*) left = mid + 1; else right = mid - 1;  }  return -1;  } |

## Console Output Examples

### Main Menu Interface



### Algorithm Performance Demo Output



# Member Contribution Breakdown

## Member 1: Awaab Ahmad (Data Structures & Core System)

**Responsibilities:**

* Core system architecture design
* Hash map implementations for O(1) lookups
* Member and Trainer class design
* Equipment and MembershipPlan classes
* Basic CRUD operations
* Hash map optimization for member and trainer indexing

**Key Contributions:**

* Implemented unordered\_map for member ID and email indexing

|  |  |
| --- | --- |
| • | unordered\_map<int, int> memberIndex; |
| • | unordered\_map<string, int> memberEmailIndex; |

* Created shift-wise organization using hash maps

unordered\_map<int, int> trainerIndex;

unordered\_map<string, vector<int>> shiftMembers; unordered\_map<string, vector<int>> shiftTrainers;

* Designed all primary classes with proper encapsulation

|  |  |
| --- | --- |
| • | lass MembershipPlan { |
| • | private: |
| • | int planId; |
| • | string planName; |
| • | int duration; |
| • | double price; |
| • | string benefits; |
| •  • | public: |
| • | MembershipPlan(int *id*, string *name*, int *dur*, double *p*, string *ben*) |
| • | : planId(*id*), planName(*name*), duration(*dur*), price(*p*), benefits(*ben*) {} |
| •  • | int getId() const { return planId; } |

|  |  |
| --- | --- |
| • | string getName() const { return planName; } |
| • | int getDuration() const { return duration; } |
| • | double getPrice() const { return price; } |
| • | string getBenefits() const { return benefits; } |
| •  • | void displayPlan() const { |
| • | cout << "Plan ID: " << planId << " | Name: " << planName |
| • | << " | Duration: " << duration << " months | Price: $" << price |
| • | << " | Benefits: " << benefits << endl; |
| • | } |
| • | }; |
| •  • | class Member { |
| • | private: |
| • | int memberId; |
| • | string name; |
| • | string email; |
| • | string phone; |
| • | int membershipPlanId; |
| • | string shift; |
| • | string joinDate; |
| • | bool isActive; |
| •  • | public: |
| • | Member(int *id*, string *n*, string *e*, string *p*, int *planId*, string *s*, string *date*) |
| • | : memberId(*id*), name(*n*), email(*e*), phone(*p*), membershipPlanId(*planId*), |
| • | shift(*s*), joinDate(*date*), isActive(true) {} |
| • • | int getId() const { return memberId; } |
| • | string getName() const { return name; } |
| • | string getEmail() const { return email; } |
| • | string getPhone() const { return phone; } |
| • | int getMembershipPlanId() const { return membershipPlanId; } |
| • | string getShift() const { return shift; } |
| • | string getJoinDate() const { return joinDate; } |
| • | bool getStatus() const { return isActive; } |
| •  • | void setName(string *n*) { name = *n*; } |
| • | void setEmail(string *e*) { email = *e*; } |
| • | void setPhone(string *p*) { phone = *p*; } |
| • | void setMembershipPlan(int *planId*) { membershipPlanId = *planId*; } |
| • | void setShift(string *s*) { shift = *s*; } |

|  |  |
| --- | --- |
| • | void deactivate() { isActive = false; } |
| • | void activate() { isActive = true; } |
| •  • | void displayMember() const { |
| • | cout << "ID: " << memberId << " | Name: " << name << " | Email: "  << email |
| • | << " | Phone: " << phone << " | Plan ID: " << membershipPlanId |
| • | << " | Shift: " << shift << " | Joined: " << joinDate |
| • | << " | Status: " << (isActive ? "Active" : "Inactive") << endl; |
| • | } |
| • | }; |
| •  • | class Trainer { |
| • | private: |
| • | int trainerId; |
| • | string name; |
| • | string specialization; |
| • | string phone; |
| • | double salary; |
| • | string shift; |
| • | string joinDate; |
| • | bool isAvailable; |
| •  • | public: |
| • | Trainer(int *id*, string *n*, string *spec*, string *p*, double *sal*, string *s*, string *date*) |
| • | : trainerId(*id*), name(*n*), specialization(*spec*), phone(*p*), |
| • | salary(*sal*), shift(*s*), joinDate(*date*), isAvailable(true) {} |
| • • | int getId() const { return trainerId; } |
| • | string getName() const { return name; } |
| • | string getSpecialization() const { return specialization; } |
| • | string getPhone() const { return phone; } |
| • | double getSalary() const { return salary; } |
| • | string getShift() const { return shift; } |
| • | string getJoinDate() const { return joinDate; } |
| • | bool getAvailability() const { return isAvailable; } |
| •  • | void setName(string *n*) { name = *n*; } |
| • | void setSpecialization(string *spec*) { specialization = *spec*; } |
| • | void setPhone(string *p*) { phone = *p*; } |
| • | void setSalary(double *sal*) { salary = *sal*; } |
| • | void setShift(string *s*) { shift = *s*; } |

|  |  |
| --- | --- |
| • | void setUnavailable() { isAvailable = false; } |
| • | void setAvailable() { isAvailable = true; } |
| •  • | void displayTrainer() const { |
| • | cout << "ID: " << trainerId << " | Name: " << name << " |  Specialization: " |
| • | << specialization << " | Phone: " << phone << " | Salary: $"  << salary |
| • | << " | Shift: " << shift << " | Joined: " << joinDate << " |  Status: " |
| • | << (isAvailable ? "Available" : "Unavailable") << endl; |
| • | } |
| • | }; |
| •  • | class Equipment { |
| • | private: |
| • | int equipmentId; |
| • | string name; |
| • | string category; |
| • | string purchaseDate; |
| • | double cost; |
| • | bool isWorking; |
| •  • | public: |
| • | Equipment(int *id*, string *n*, string *cat*, string *date*, double *c*) |
| • | : equipmentId(*id*), name(*n*), category(*cat*), purchaseDate(*date*), |
| • | cost(*c*), isWorking(true) {} |
| •  • | int getId() const { return equipmentId; } |
| • | string getName() const { return name; } |
| • | string getCategory() const { return category; } |
| • | string getPurchaseDate() const { return purchaseDate; } |
| • | double getCost() const { return cost; } |
| • | bool getStatus() const { return isWorking; } |
| • • | void setName(string *n*) { name = *n*; } |
| • | void setCategory(string *cat*) { category = *cat*; } |
| • | void markBroken() { isWorking = false; } |
| • | void markWorking() { isWorking = true; } |
| •  • | void displayEquipment() const { |
| • | cout << "ID: " << equipmentId << " | Name: " << name << " |  Category: " |
| • | << category << " | Purchase Date: " << purchaseDate << " |  Cost: $" |
| • | << cost << " | Status: " << (isWorking ? "Working" :  "Broken") << endl; |
| • | } |
| • | }; |

* Optimize members and trainers indexing using Hash map

|  |
| --- |
| void updateHashMaps() { memberIndex.clear(); memberEmailIndex.clear(); trainerIndex.clear(); shiftMembers.clear(); shiftTrainers.clear();  for (int i = 0; i < members.size(); i++) { memberIndex[members[i].getId()] = i; memberEmailIndex[members[i].getEmail()] = i; shiftMembers[members[i].getShift()].push\_back(i);  } for (int i = 0; i < trainers.size(); i++) { trainerIndex[trainers[i].getId()] = i; shiftTrainers[trainers[i].getShift()].push\_back(i);  }  } |

* Implemented basic search and display functionalities
* **Member 2: Mubeen Irshad (Sorting Algorithms & UI)**

**Responsibilities:**

* Quick Sort implementation for member sorting
* Merge Sort implementation for trainer salary sorting
* User interface design and menu systems
* Input validation and error handling
* Report generation system
* Performance timing implementations

**Key Contributions:**

* Implemented Quick Sort with time complexity analysis

|  |  |
| --- | --- |
| • | void quickSort(vector<Member>& *arr*, int *low*, int *high*, string *sortBy*) { |
| • | if (*low* < *high*) { |
| • | int pi = partition(*arr*, *low*, *high*, *sortBy*); |
| • | quickSort(*arr*, *low*, pi - 1, *sortBy*); |
| • | quickSort(*arr*, pi + 1, *high*, *sortBy*); |
| • | } |
| • | } |
| •  • | int partition(vector<Member>& *arr*, int *low*, int *high*, string *sortBy*) { |
| • | string pivot; |
| • | if (*sortBy* == "name") { |
| • | pivot = *arr*[*high*].getName(); |
| • | } else if (*sortBy* == "id") { |
| • | pivot = to\_string(*arr*[*high*].getId()); |
| • | } |
| •  • | int i = (*low* - 1); |
| • | for (int j = *low*; j <= *high* - 1; j++) { |
| • | string current; |
| • | if (*sortBy* == "name") { |
| • | current = *arr*[j].getName(); |
| • | } else if (*sortBy* == "id") { |
| • | current = to\_string(*arr*[j].getId()); |
| • | } |
| •  • | if (current <= pivot) { |
| • | i++; |
| • | swap(*arr*[i], *arr*[j]); |
| • | } |
| • | } |
| • | swap(*arr*[i + 1], *arr*[*high*]); |
| • | return (i + 1); |

* Developed Merge Sort with performance benchmarking

void mergeSort(vector<Trainer>& *arr*, int *left*, int *right*) { if (*left* < *right*) { int mid = *left* + (*right* - *left*) / 2; mergeSort(*arr*, *left*, mid);

|  |
| --- |
| mergeSort(*arr*, mid + 1, *right*); merge(*arr*, *left*, mid, *right*);  }  } void merge(vector<Trainer>& *arr*, int *left*, int *mid*, int *right*) { int n1 = *mid* - *left* + 1; int n2 = *right* - *mid*;  vector<Trainer> leftArr, rightArr; for (int i = 0; i < n1; i++) leftArr.push\_back(*arr*[*left* + i]); for (int j = 0; j < n2; j++) rightArr.push\_back(*arr*[*mid* + 1 + j]);  int i = 0, j = 0, k = *left*; while (i < n1 && j < n2) { if (leftArr[i].getSalary() <= rightArr[j].getSalary()) { *arr*[k] = leftArr[i]; i++; } else { *arr*[k] = rightArr[j]; j++; } k++;  }  while (i < n1) { *arr*[k] = leftArr[i]; i++; k++;  }    while (j < n2) { *arr*[k] = rightArr[j]; j++; k++;  }  } |

* Created comprehensive menu systems

|  |  |
| --- | --- |
| • | void showMenu() { |
| • | cout << "\n=== GYM MANAGEMENT SYSTEM ===" << endl; |

|  |  |
| --- | --- |
| • | cout << "1. Add Member" << endl; |
| • | cout << "2. Display All Members" << endl; |
| • | cout << "3. Display Members by Shift" << endl; |
| • | cout << "4. Search Member (Hash Map O(1))" << endl; |
| • | cout << "5. Delete Member" << endl; |
| • | cout << "6. Add Trainer" << endl; |
| • | cout << "7. Display All Trainers" << endl; |
| • | cout << "8. Display Trainers by Shift" << endl; |
| • | cout << "9. Add Equipment" << endl; |
| • | cout << "10. Display All Equipment" << endl; |
| • | cout << "11. Display Membership Plans" << endl; |
| • | cout << "12. Add Membership Plan" << endl; |
| • | cout << "13. Generate Report" << endl; |
| • | cout << "14. Sorting & Searching Algorithms Demo" << endl; |
| • | cout << "0. Exit" << endl; |
| • | cout << "Enter your choice: "; |
| • | } |
| •  • | void run() { |
| • | int choice; |
| • | do { |
| • | showMenu(); |
| • | cin >> choice; |
| •  • | switch (choice) { |
| • | case 1: addMember(); break; |
| • | case 2: displayMembers(); break; |
| • | case 3: displayMembersByShift(); break; |
| • | case 4: searchMember(); break; |
| • | case 5: deleteMember(); break; |
| • | case 6: addTrainer(); break; |
| • | case 7: displayTrainers(); break; |
| • | case 8: displayTrainersByShift(); break; |
| • | case 9: addEquipment(); break; |
| • | case 10: displayEquipments(); break; |
| • | case 11: displayMembershipPlans(); break; |
| • | case 12: addMembershipPlan(); break; |
| • | case 13: generateReport(); break; |
| • | case 14: |
| • | { |
| • | int algoChoice; |
| • | do { |
| • | showAlgorithmMenu(); |
| •  • | cin >> algoChoice; |

|  |  |
| --- | --- |
| • | switch (algoChoice) { |
| • | case 15: sortMembers(); break; |
| • | case 16: sortTrainers(); break; |
| • | case 17: advancedLinearSearch(); break; |
| • | case 18: binarySearchMembers(); break; |
| • | case 19: dfsEquipmentCategorySearch(); break; |
| • | case 20: bfsTrainerSpecializationSearch(); break; |
| • | case 21: algorithmPerformanceDemo(); break; |
| • | case 22: cout << "Returning to main menu..." << endl; break; |
| • | default: cout << "Invalid choice! Please try again." << endl; |
| • | } |
| • | } while (algoChoice != 22); |
| • | } |
| • | break; |
| • | case 0: cout << "Thank you for using Enhanced Gym  Management System!" << endl; break; |
| • | default: cout << "Invalid choice! Please try again." << endl; |
| • | } |
| • | } while (choice != 0); |
| • | } |
| • | }; |
| •  • | int main() { |
| • | cout << "=== ENHANCED GYM MANAGEMENT SYSTEM ===" << endl; |
| • | cout << "Featuring Advanced Sorting & Searching Algorithms:" << endl; |
| • | cout << " Quick Sort & Merge Sort" << endl; |
| • | cout << " Linear Search & Binary Search" << endl; |
| • | cout << " Depth-First Search (DFS) & Breadth-First Search (BFS)" << endl; |
| • | cout << " Hash Map optimizations for O(1) lookups" << endl; |
| • | cout << " Performance analysis and time complexity demonstrations" << endl; |
| • | cout << "\nSample data has been pre-loaded for algorithm demonstrations." << endl; |
| • | cout << "=========================================" << endl; |
| • • | GymManagementSystem gym; |
| • | gym.run(); |
| • | return 0; |
| • | } |

* Developed financial and statistical reporting features

|  |  |
| --- | --- |
| • | void generateReport() { |
| • | cout << "\n=== GYM MANAGEMENT REPORT ===" << endl; |
| • | cout << "Total Members: " << members.size() << endl; |
| • | cout << "Total Trainers: " << trainers.size() << endl; |
| • | cout << "Total Equipment: " << equipments.size() << endl; |
| • | cout << "Total Membership Plans: " << membershipPlans.size() << endl; |
| •  • | int morningMembers = shiftMembers["Morning"].size(); |
| • | int eveningMembers = shiftMembers["Evening"].size(); |
| • | int fullDayMembers = shiftMembers["Full Day"].size(); |
| • | int morningTrainers = shiftTrainers["Morning"].size(); |
| • | int eveningTrainers = shiftTrainers["Evening"].size(); |
| • | int fullDayTrainers = shiftTrainers["Full Day"].size(); |
| •  • | double totalRevenue = 0; |
| • | for (const auto& member : members) { |
| • | if (member.getStatus()) { |
| • | auto planIt = find\_if(membershipPlans.begin(), membershipPlans.end(), |
| • | [&*member*](const MembershipPlan& *p*) { return *p*.getId()  == member.getMembershipPlanId(); }); |
| • | if (planIt != membershipPlans.end()) { |
| • | totalRevenue += planIt->getPrice(); |
| • | } |
| • | } |
| • | } |
| • • | double totalSalary = 0; |
| • | for (const auto& trainer : trainers) { |
| • | totalSalary += trainer.getSalary(); |
| • | } |
| •  • | cout << "\n--- SHIFT DISTRIBUTION (Hash Map O(1) Access) ---" << endl; |
| • | cout << "Morning Shift - Members: " << morningMembers << " |  Trainers: " << morningTrainers << endl; |
| • | cout << "Evening Shift - Members: " << eveningMembers << " |  Trainers: " << eveningTrainers << endl; |
| • | cout << "Full Day Shift - Members: " << fullDayMembers << " |  Trainers: " << fullDayTrainers << endl; |
| •  • | cout << "\n--- FINANCIAL REPORT ---" << endl; |
| • | cout << "Total Revenue: $" << fixed << setprecision(2) << totalRevenue << endl; |
| • | cout << "Total Expenses (Salaries): $" << fixed << setprecision(2)  << totalSalary << endl; |
| • | cout << "Net Profit: $" << fixed << setprecision(2) <<  (totalRevenue - totalSalary) << endl; |
| •  • | cout << "\n--- ALGORITHM PERFORMANCE SUMMARY ---" << endl; |
| • | cout << "Hash Map Lookups: O(1) average time complexity" << endl; |
| • | cout << "Quick Sort: O(n log n) average, O(n²) worst case" << endl; |
| • | cout << "Merge Sort: O(n log n) guaranteed" << endl; |
| • | cout << "Linear Search: O(n) time complexity" << endl; |
| • | cout << "Binary Search: O(log n) time complexity" << endl; |
| • | cout << "DFS/BFS: O(V + E) time complexity" << endl; |
| • | } |

## Member 3: Hanni Mustafa (Search Algorithms & Optimization)

**Responsibilities:**

* Linear Search implementation with timing analysis
* Binary Search implementation on sorted data
* Depth-First Search (DFS) for equipment categorization
* Breadth-First Search (BFS) for trainer specializations
* Algorithm performance comparison system

**Key Contributions:**

* Implemented all four search algorithms with detailed analysis

|  |  |
| --- | --- |
| • | int linearSearch(const vector<Member>& *arr*, string *searchTerm*, string |
|  | *searchBy*) { |
| • | cout << "\n[LINEAR SEARCH] Starting search..." << endl; |
| • | auto start = chrono::high\_resolution\_clock::now(); |
| •  • | for (int i = 0; i < *arr*.size(); i++) { |
| • | if (*searchBy* == "name" && *arr*[i].getName() == *searchTerm*) { |
| • | auto end = chrono::high\_resolution\_clock::now(); |
| • | auto duration =  chrono::duration\_cast<chrono::microseconds>(end - start); |
| • | cout << "[LINEAR SEARCH] Found in " << duration.count() <<  " microseconds" << endl; |
| • | cout << "[LINEAR SEARCH] Time Complexity: O(n)" << endl; |
| • | return i; |
| • | } else if (*searchBy* == "email" && *arr*[i].getEmail() == *searchTerm*) { |
| • | auto end = chrono::high\_resolution\_clock::now(); |
| • | auto duration =  chrono::duration\_cast<chrono::microseconds>(end - start); |
| • | cout << "[LINEAR SEARCH] Found in " << duration.count() <<  " microseconds" << endl; |
| • | cout << "[LINEAR SEARCH] Time Complexity: O(n)" << endl; |
| • | return i; |
| • | } |
| • | } |
| •  • | auto end = chrono::high\_resolution\_clock::now(); |
| • | auto duration = chrono::duration\_cast<chrono::microseconds>(end - start); |
| • | cout << "[LINEAR SEARCH] Search completed in " << duration.count()  << " microseconds" << endl; |
| • | cout << "[LINEAR SEARCH] Time Complexity: O(n)" << endl; |
| • | return -1; |
| •  • | } |

* Created algorithm performance comparison features

|  |  |
| --- | --- |
| • | void algorithmPerformanceDemo() { |
| • | cout << "\n=== ALGORITHM PERFORMANCE DEMONSTRATION ===" << endl; |
| • | cout << "This demo compares Linear Search vs Binary Search performance" << endl; |
| • • | if (members.size() < 3) { |
| • | cout << "Need at least 3 members for meaningful comparison.  Add more members first." << endl; |
| • | return; |
| • | } |
| •  • | *// Search for the last member using both algorithms* |
| • | int targetId = members.back().getId(); |
| • | cout << "\nSearching for Member ID: " << targetId << endl; |

|  |  |
| --- | --- |
| • | cout << "Dataset size: " << members.size() << " members" << endl; |
| •  • | cout << "\n--- LINEAR SEARCH PERFORMANCE ---" << endl; |
| • | auto start1 = chrono::high\_resolution\_clock::now(); |
| • | int linearResult = -1; |
| • | for (int i = 0; i < members.size(); i++) { |
| • | if (members[i].getId() == targetId) { |
| • | linearResult = i; |
| • | break; |
| • | } |
| • | } |
| • | auto end1 = chrono::high\_resolution\_clock::now(); |
| • | auto duration1 = chrono::duration\_cast<chrono::microseconds>(end1  - start1); |
| • | cout << "Linear Search Time: " << duration1.count() << " microseconds" << endl; |
| • | cout << "Comparisons made: " << linearResult + 1 << endl; |
| •  • | cout << "\n--- BINARY SEARCH PERFORMANCE ---" << endl; |
| • | vector<Member> sortedMembers = members; |
| • | quickSort(sortedMembers, 0, sortedMembers.size() - 1, "id"); |
| •  • | auto start2 = chrono::high\_resolution\_clock::now(); |
| • | int binaryResult = binarySearch(sortedMembers, targetId); |
| • | auto end2 = chrono::high\_resolution\_clock::now(); |
| • | auto duration2 = chrono::duration\_cast<chrono::microseconds>(end2  - start2); |
| •  • | cout << "\n--- PERFORMANCE COMPARISON ---" << endl; |
| • | cout << "Linear Search: O(n) - " << duration1.count() << " microseconds" << endl; |
| • | cout << "Binary Search: O(log n) - " << duration2.count() << " microseconds" << endl; |
| • • | if (duration1.count() > duration2.count()) { |
| • | cout << "Binary Search was " <<  (double)duration1.count()/duration2.count() |
| • | << "x faster!" << endl; |
| • | } else { |
| • | cout << "Performance difference minimal due to small dataset size." << endl; |
| • | } |
| •  • | cout << "\nNote: Binary search requires sorted data (additional  O(n log n) cost)" << endl; |
| • | cout << "Linear search works on unsorted data but is slower for large datasets" << endl; |
| • | } |
| •  • | void displayMembershipPlans() { |
| • | cout << "\n=== MEMBERSHIP PLANS ===" << endl; |
| • | for (const auto& plan : membershipPlans) { |
| • | plan.displayPlan(); |
| • | } |
| •  • | } |

* Developed BFS and DFS with visual step-by-step output

|  |  |
| --- | --- |
| • | void dfsEquipmentSearch(string *targetCategory*, vector<bool>& *visited*, |
| • | vector<vector<int>>& *adjList*, int *node*, vector<int>& *result*) { |
| • | *visited*[*node*] = true; |
| • | cout << "[DFS] Visiting equipment ID: " << equipments[*node*].getId() |
| • | << " (" << equipments[*node*].getCategory() << ")" << endl; |
| •  • | if (equipments[*node*].getCategory() == *targetCategory*) { |
| • | *result*.push\_back(*node*); |
| • | } |
| •  • | for (int neighbor : *adjList*[*node*]) { |
| • | if (!*visited*[neighbor]) { |
| • | dfsEquipmentSearch(*targetCategory*, *visited*, *adjList*, neighbor, *result*); |
| • | } |
| • | } |
| • | } |
| • • | *// SEARCHING ALGORITHM 4: BREADTH-FIRST SEARCH (for trainer specialization network)* |
| • | vector<int> bfsTrainerSearch(string *targetSpecialization*) { |
| • | cout << "\n[BFS] Starting breadth-first search for specialization:  " |
| • | << *targetSpecialization* << endl; |
| •  • | vector<int> result; |
| • | vector<bool> visited(trainers.size(), false); |
| • | queue<int> q; |

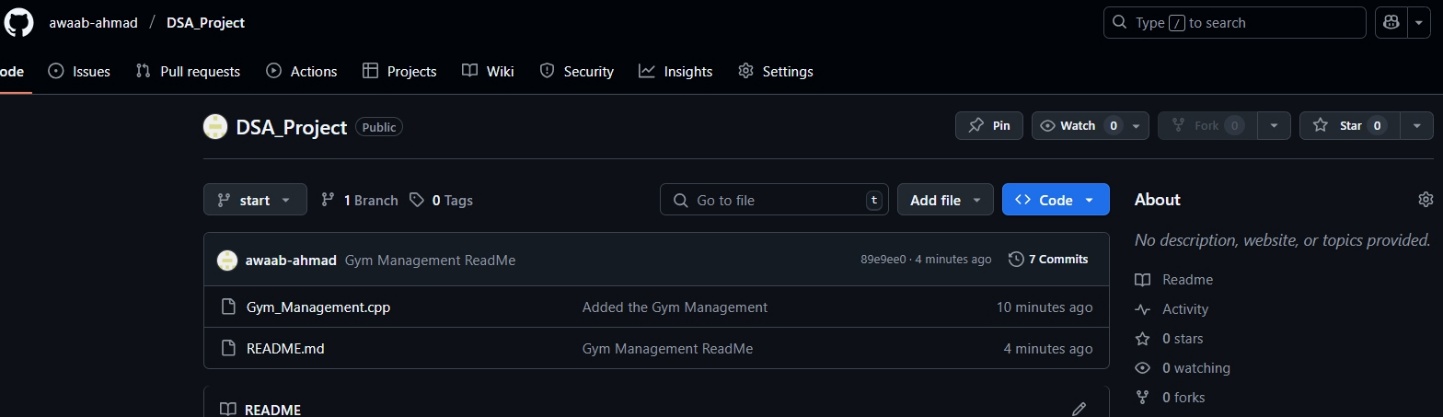
|  |  |
| --- | --- |
| •  • | if (!trainers.empty()) { |
| • | q.push(0); |
| • | visited[0] = true; |
| • | } |
| •  • | int level = 0; |
| • | while (!q.empty()) { |
| • | int levelSize = q.size(); |
| • | cout << "[BFS] Level " << level << ":" << endl; |
| •  • | for (int i = 0; i < levelSize; i++) { |
| • | int current = q.front(); |
| • | q.pop(); |
| •  • | cout << "[BFS] Checking trainer: " << trainers[current].getName() |
| • | << " (Specialization: " << trainers[current].getSpecialization() << ")" << endl; |
| •  • | if (trainers[current].getSpecialization() == *targetSpecialization*) { |
| • | result.push\_back(current); |
| • | } |
| •  • | *// Add adjacent trainers (next trainers in vector for demonstration)* |
| • | for (int j = current + 1; j < min(current + 3,  (int)trainers.size()); j++) { |
| • | if (!visited[j]) { |
| • | visited[j] = true; |
| • | q.push(j); |
| • | } |
| • | } |
| • | } |
| • | level++; |
| • | if (level > 3) break; *// Limit search depth for demonstration* |
| • | } |
| •  • | cout << "[BFS] Search completed. Found " << result.size() |
| • | << " trainers with specialization: " << *targetSpecialization*  << endl; |
| • | cout << "[BFS] Time Complexity: O(V + E)" << endl; |
| • | return result; |
| • | } |

# Repository & Social Links

## GitHub Repository

**Repository URL**: <https://github.com/awaab-ahmad/DSA_Project>

**Code File**: <https://github.com/awaab-ahmad/DSA_Project/blob/start/Gym_Management.cpp>



What’s Included:

This GitHub Link Include the Readme + Code file of the Enhanced Gym Management System.

## LinkedIn Link:

<https://www.linkedin.com/posts/awaab-ahmad-b9aa7a31a_dsaprojectgymmanagementcpp-at-start-activity-7333383788300840960-AaSi?utm_source=share&utm_medium=member_desktop&rcm=ACoAAFD12SYBa6_BFJHkP8N9-4kqiaxTNAIS9wU>

This Link is being uploaded on the official page of Awaab Ahmad Minhas.  
In this link, we placed the Link to the GitHub Code File.

**Summary**

The Enhanced Gym Management System successfully demonstrates the practical application of advanced data structures and algorithms in real-world software development. By implementing multiple sorting and searching algorithms with comprehensive performance analysis, this project provides valuable insights into algorithmic efficiency and optimization techniques.

**Conclusion:**

The system's hash map optimization achieves O(1) lookup times, while the algorithm comparison features provide educational value for understanding time and space complexity. The comprehensive documentation and performance benchmarking make this project an excellent demonstration of both theoretical knowledge and practical implementation skills.

**Future improvements:**

Future enhancements could include database integration, web interface development, and additional graph algorithms for more complex relationship modeling between gym entities.