**Work Documentation**

**Work Description**

This document explains in detail the thought process and problem-solving methods used in the file *"ergasia.cpp"*. The purpose of this project is to develop a program that implements singly linked lists within the *Data Structures* course.

**This file implements:**

* Loading participant data from a file.
* Sorting and managing participants based on various criteria such as ID, ranking, and timestamp.
* Performing operations such as deleting participants, finding a specific player by name, and printing participant details in various orders.

**Technologies and Knowledge Used:**

* The system is built using **C++** for object-oriented programming (OOP) and memory management.
* **File Handling:** The system uses file input and output to read participant data and store it.

**Required Functions**

1. Reading players from a file in the order they appear.
2. Reading players from a file sorted by ID.
3. Sorting the list based on player ranking.
4. Deleting a player by ID.
5. Deleting the entire player list.
6. Searching for a player by name.
7. Printing the player list in its current order.
8. Printing the player list based on registration timestamp.
9. Exiting the program.

**Detailed Explanation:**

(Here, the corresponding images from the original document are included for each function description.)

**Code Explanation**

**MAIN()**

The *main()* function is kept as minimal as possible, as the menu display and function implementations are handled by:

* **Menu()**
* **Actions(int answ)**

The *menu* function is called repeatedly, and if its return value is one of the valid actions (1-9), the *actions* function executes the corresponding operation. If an out-of-range number or an invalid input (e.g., non-numeric) is given, error handling is performed accordingly.

**Linked List Implementation**

Data storage is achieved using a singly linked list. This is implemented with two object classes:

* **participant**
* **participantList**

**participant:**

This class creates the objects that form the nodes of the list.

(Here, images illustrating the participant class structure are included.)

**participantList:**

This class creates the linked list itself and implements the required operations.

(Here, images illustrating the participantList class structure and its methods are included.)

**Function Explanations**

**participant Class**

* **parseData(string& str)**: Used in the class constructor. Takes a string as input and assigns values to the private members (fullname, ID, origin, ranking, timestamp).
* **Getter functions** (e.g., getName()): Provide access to private class members.
* **swapWith(participant\* other)**: Takes another node as a pointer and swaps data values using the built-in swap function of C++. This function does not modify pointers to change node positions in the list but instead swaps the values of private members.

(Here, images showing code snippets for these functions are included.)

**participantList Class**

* **length()**: Helper function that traverses the list and returns its size.
* **readfile(string filename)**: Implements the first function. If the list is empty and the file exists, it reads the file linearly using getline() and creates the list sequentially.
* **readfileOrdered(string filename)**: Implements the second function. Works similarly to readfile() but inserts nodes in sorted order based on ID.
* **sortlist()**: Implements the third function. It does not take parameters and sorts the elements in descending ranking order using *selection sort*. The maximum ranking node is found in successive iterations, and swapWith() is used to swap elements.
* **deletebyID(int id)**: Implements the fourth function. The ID is given as a parameter and is sequentially checked against the nodes' IDs. If the ID is not found, no node is deleted.
* **findPlayer(string name1, string lname1)**: Implements the sixth function. Takes a case-insensitive first and last name as parameters and returns the participant's details if found.
* **printAll()**: Implements the seventh function. Sequentially prints all participants.
* **printbyTimestamporder()**: Implements the eighth function. Uses a 100-position array (since there are at most 100 users), which is filled with timestamps and sorted using *bubble sort*. The list is then traversed to print all nodes in order of matching timestamps.