**A logo of a globe with yellow rings around it

Description automatically generated**

**GROUP ASSIGNMENT**

**CT077-3-2-DSTR**

**DATA STRUCTUR**

**APD2F2309CS(CYB)**

**HANDOUT DATE:  7th week**

**HAND-IN DATE:  14th week**

**WEIGHTAGE: 60%**

|  |  |
| --- | --- |
| **NAME:** | **TP NUMBER:** |
| IBRAHEEM NOHAMMED IMADELDIN AWAD | TP070765 |
| ABDULRAHMAN GAMIL MOHAMMED AHMED | TP071012 |
| FOUAD MOHAMED FOUAD ISMAIL | TP069613 |
| SAIF WALEED ABDELRAHMAN SAYED ABDELSALAM | TP068702 |
| NOUR AHMED ABDELMONEIM ABDELAZIZ | TP068768 |

Table of Contents

[1.0 WORK BREAKDOWN STRUCTURE (WBS) 3](#_Toc168947829)

[2.0 System Flowchart 5](#_Toc168947830)

[3.0 SYSTEM SCREEN SHOTS 15](#_Toc168947831)

[4.0 Result, Summary and Reflection Section – Discussions 28](#_Toc168947832)

[Execution Times 28](#_Toc168947833)

[Observation and Thoughts During Development: 28](#_Toc168947834)

[Strengths: 30](#_Toc168947835)

[Weaknesses: 31](#_Toc168947836)

[Fixing Systemic Weaknesses and Future Work Approaches: 32](#_Toc168947837)

[Practical Experience and Feedback on Practical Implementation: 34](#_Toc168947838)

[New Solutions and Ideas to Address System Weaknesses and Inefficiencies 36](#_Toc168947839)

[Personal Thoughts on the Assignment 37](#_Toc168947840)

[References 38](#_Toc168947841)

LIST OF FIGURES

[Figure 1: System Flowchart 5](#_Toc168947809)

[Figure 2: Linked list Class (1) 15](#_Toc168947810)

[Figure 3: Linked List Class (2) 16](#_Toc168947811)

[Figure 4: Linked List Class (3) 16](#_Toc168947812)

[Figure 5: Linked List Class (4) 17](#_Toc168947813)

[Figure 6: Player Class (1) 18](#_Toc168947814)

[Figure 7: Player Class (2) 19](#_Toc168947815)

[Figure 8: Player Class (3) 20](#_Toc168947816)

[Figure 9: Game Class (1) 21](#_Toc168947817)

[Figure 10: Game Class (2) 22](#_Toc168947818)

[Figure 11: Game Class (3) 23](#_Toc168947819)

[Figure 12: Game Class (4) 23](#_Toc168947820)

[Figure 13: Binary Search Tree Class (1) 24](#_Toc168947821)

[Figure 14: Binary Search Tree Class (2) 24](#_Toc168947822)

[Figure 15: Binary Search Tree Class (3) 25](#_Toc168947823)

[Figure 16: Main Function (1) 25](#_Toc168947824)

[Figure 17: Main Function (2) 26](#_Toc168947825)

[Figure 18: Program Output (Switching Decks if Discarded is empty) 27](#_Toc168947826)

[Figure 19: Program Output (Leader For all students output) 27](#_Toc168947827)

[Figure 20: Program Output (Tree Leaderboard For First 30 Students) 27](#_Toc168947828)

# 1.0 WORK BREAKDOWN STRUCTURE (WBS)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ABDULRAHMAN GAMIL MOHAMMED AHMED | NOUR AHMED ABDELMONEIM ABDELAZIZ | SAIF WALEED ABDELRAHMAN SAYED ABDELSALAM | IBRAHEEM MOHAMMED IMADELDIN AWAD | FOUAD MOHAMED FOUAD ISMAIL | Total |
| Documentation | 20% | 20% | 20% | 20% | 20% | 100% |
| Game class | 20% | 20% | 20% | 20% | 20% | 100% |
| BinarySearchTree Class | 20% | 20% | 20% | 20% | 20% | 100% |
| LinkedList Class | 20% | 20% | 20% | 20% | 20% | 100% |
| Player Class | 20% | 20% | 20% | 20% | 20% | 100% |

# 2.0 System Flowchart

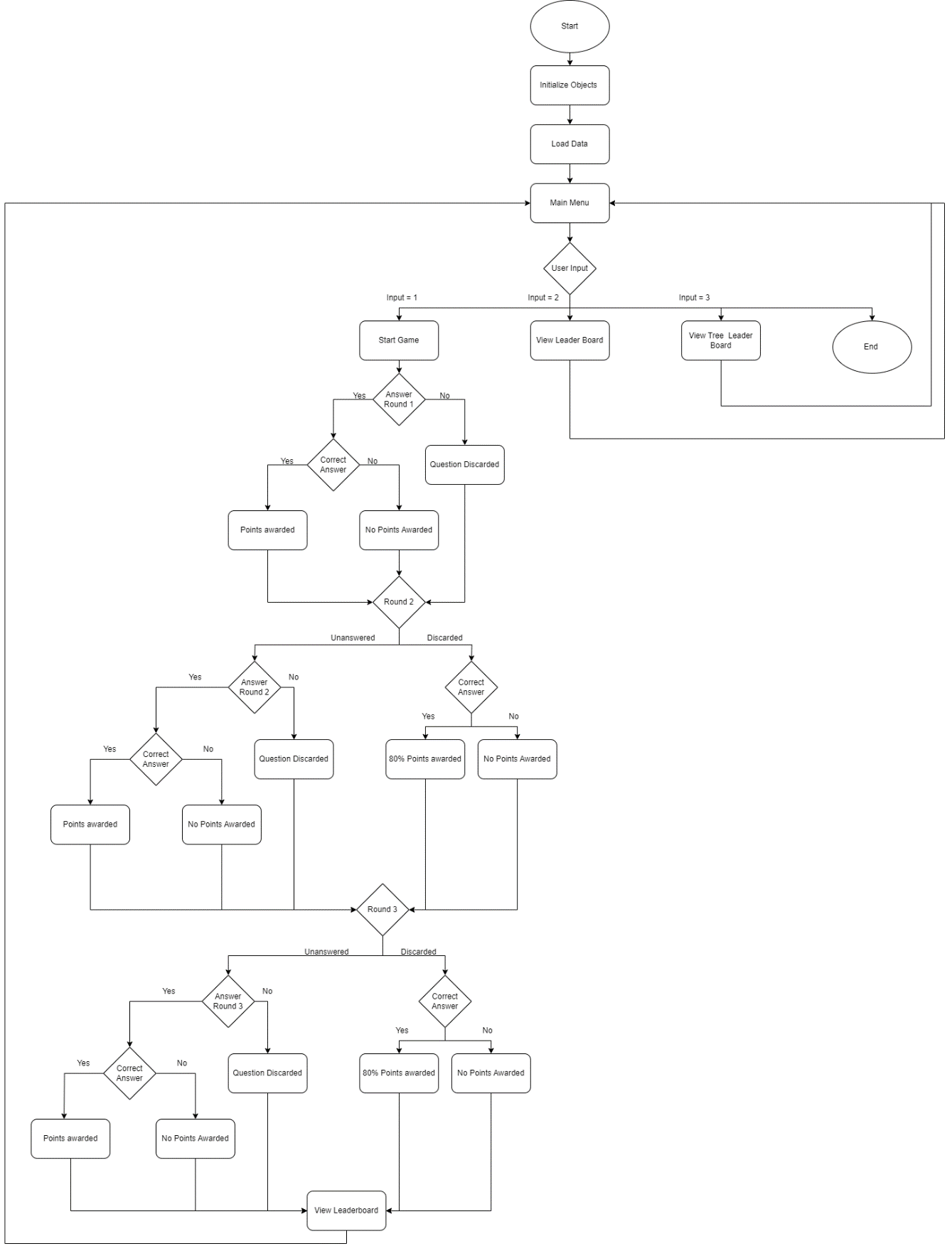


Figure 1: System Flowchart

**BinarySearchTree Class**

**TreeNode Class**

The TreeNode class represents a node in the binary search tree (BST). It contains:

* A pointer to a Player object (player).
* Two pointers to its left and right children (left and right), which are also TreeNode objects.

**Data Manipulation Functions**

**insert Function**

The insert function is used to add a new Player to the BST. It uses recursion to find the correct spot for the new node based on the Player's points:

* If the current node (node) is nullptr, a new TreeNode is created and returned.
* If the Player's points are greater than the current node's player points, the function recurses to the right child.
* Otherwise, it recurses to the left child.

This ensures that players with higher points are placed on the right side of the tree and players with lower points on the left, maintaining the BST property.

**collectNodes Function**

The collectNodes function is a helper for gathering all nodes in the tree into an array:

* It performs an in-order traversal (left, root, right) to collect the nodes in a sorted order based on their points.
* The nodes array collects the Player pointers, and index tracks the current position in the array.

**sortPlayers Function**

The sortPlayers function sorts the collected players array in descending order by points:

* A simple bubble sort is used to sort the array.
* This ensures the highest-scoring players are at the beginning of the array.

**Display Function**

The display function manually displays the players in a structured format. It:

* Collects the nodes using collectNodes.
* Sorts the players using sortPlayers.
* Displays the players in a hierarchical manner, reflecting their order in the tree.

**Design Choices**

**Binary Search Tree (BST):**

* + The Binary Search Tree is chosen for its efficient insertion and search operations. Since players are inserted based on their points, the Binary Search Tree helps maintain an ordered structure.

**In-order Traversal:**

* + This traversal ensures that nodes are collected in ascending order of points, which simplifies sorting and display operations.

**Bubble Sort:**

* + Bubble sort is used for its simplicity.

**Manual Display:**

* + The display function shows the hierarchy of players. While it effectively demonstrates the tree structure, it is not scalable for large datasets. For more players, dynamic formatting or a different display method would be needed.

**Game Class**

The Game class manages a quiz game, where players answer questions drawn from different decks (unanswered, answered, and discarded). It tracks the players' scores and their responses in each round.

The displayDetailedLeaderboard function displays the leaderboard showing the top 30 players with their scores and round-by-round performance.

**Data Structures and Manipulation**

**Player Class**

This class represents each player and manages their:

* Name
* Total points
* Round-by-round answers and questions

**LinkedList Class**

This class represents the decks of questions, managing:

* Unanswered questions
* Answered questions
* Discarded questions

Each list node presumably contains a Card object, representing a quiz question with its details.

**Game Class Functions**

**playGame Function**

The playGame function orchestrates the entire gameplay across multiple rounds:

**Rounds and Players:**

* + The game consists of multiple rounds, iterating through each player for each round.

**Drawing Questions:**

* + In the first round, questions are drawn from the unanswered deck.
  + In subsequent rounds, players choose to draw from either the unanswered or discarded deck.

**Answering Questions:**

* + Players can choose to answer or skip the question.
  + If they answer correctly, points are awarded, and the question is moved to the answered deck.
  + Incorrect answers and skipped questions move the card to the discarded deck.

**Deck Management:**

* The discardedDeck is shuffled before drawing a question.
* If the discardedDeck is empty, players are redirected to the unansweredDeck.

**Updating Player Records:**

* Player's answers and the corresponding questions are recorded for each round.

**Score Calculation:**

* Correct answers in the first round get full points, while in subsequent rounds, correct answers from the discardedDeck get 80% of the points.

**displayDetailedLeaderboard Function**

The displayDetailedLeaderboard function displays the leaderboard:

* Shows the top 30 players based on their scores.
* Displays the player's name, round-by-round performance, and overall score.
* Provides a detailed view of the questions answered in each round.

**Design Choices**

**LinkedList for Decks:**

* Using a linked list allows efficient insertions and deletions, which are common operations when moving cards between decks.

**Player Interaction:**

* Players interact with the game by choosing whether to answer or skip questions and by selecting the deck from which to draw questions in later rounds.

**Randomness:**

* The discarded deck is shuffled to introduce randomness and ensure fairness when drawing questions.

**Detailed Tracking:**

* Player responses and the questions they encounter are meticulously tracked to provide detailed feedback on the leaderboard.

**Linked List Class**

The LinkedList class manages a collection of Card objects, where each Card represents a quiz question, its answer, score, and whether it has been answered. This class provides essential functionalities such as adding (push), removing (pop), checking if empty (isEmpty), getting size (getSize), loading from file (loadQuestionsFromFile), shuffling (shuffleDeck), and displaying (display) the linked list.

**Card Struct**

* **string question:** The quiz question.
* **string answer**: The correct answer to the quiz question.
* **int score:** The score associated with the question.
* **bool answered:** Indicates whether the question has been answered.

**Node Class**

The Node class represents a node in the linked list, encapsulating:

* **Card data:** The card data.
* **Node\* next:** A pointer to the next node in the list.

**LinkedList Class**

The LinkedList class manages the list of Node objects and includes the following functionalities:

**Constructor and Destructor:**

* LinkedList(): Initializes an empty list.
* **LinkedList():** Properly deallocates memory when the list is destroyed.

**push():**

* Adds a new card to the end of the list. If the list is empty, both head and tail point to the new node. Otherwise, the new node is appended, and tail is updated.

**pop():**

* Removes and returns the card from the front of the list. If the list is empty, it throws an exception.

**isEmpty():**

* Checks if the list is empty.

**getSize():**

* Returns the current size of the list.

**loadQuestionsFromFile():**

* Loads questions from a specified file into the list. Each line in the file should be formatted as question,answer,score points.
* Parses each line and creates a Card object, which is then added to the list using push.
* Calls shuffleDeck to shuffle the list after loading the questions.

**shuffleDeck():**

* Shuffles the elements of the linked list using the Fisher-Yates shuffle algorithm.
* First, converts the linked list to an array, shuffles the array, and then converts it back to a linked list.

**display():**

* Prints the questions and scores of all cards in the list for debugging or visualization purposes.

**getQuestion():**

* Finds and returns the question associated with a given answer. Returns "N/A" if the answer is not found.

**Design Choices**

**Linked List for Decks:**

* Using a linked list allows efficient insertion and deletion operations, which are frequent in managing the decks of cards during the game.

**Card Shuffling:**

* The shuffle operation uses an array to facilitate random access and shuffling, which is then converted back to a linked list.

**File Loading:**

* The loadQuestionsFromFile function facilitates loading questions from an external file, making it easier to manage and update the question set.

**Error Handling:**

* Proper error handling is implemented in the pop and loadQuestionsFromFile functions to handle cases when the deck is empty.

**Player Class**

The Player class manages individual players in a game, including their details, scores, and round-specific answers and questions. The class also provides functionalities for loading player data from a file, sorting players based on points, and displaying the leaderboard.

**Member Variables**

* int id: The unique identifier for the player.
* string name: The name of the player.
* int points: The total points accumulated by the player.
* string roundAnswers[3]: Array to store the answers for each of the three rounds.
* string roundQuestions[3]: Array to store the questions for each of the three rounds.
* int roundScores[3]: Array to store the scores for each of the three rounds.

**Constructor**

* Player (int id, const string& name, int points): Initializes a player with the given id, name, and points, and sets default values for round-specific arrays.

**Member Functions**

* int getId() const: Returns the player's ID.
* const string& getName() const: Returns the player's name.
* int getPoints() const: Returns the player's total points.
* void addPoints(int points): Adds points to the player's total.
* void subtractPoints(int points): Subtracts points from the player's total.

**Round-Specific Methods**

* void addRoundAnswer(int round, const string& answer): Stores the answer for a specific round.
* string getRoundAnswer(int round) const: Retrieves the answer for a specific round.
* void addRoundQuestion(int round, const string& question): Stores the question for a specific round.
* string getRoundQuestion(int round) const: Retrieves the question for a specific round.
* void addRoundScore(int round, int score): Stores the score for a specific round.
* int getRoundScore(int round) const: Retrieves the score for a specific round.

**Static Functions**

* static void loadStudentsFromFile(const string& filename, Player\*\* players, int& numPlayers): Loads player data from a file. Each line in the file should be in the format name,score, where score is an integer. The function initializes players and updates numPlayers.
* static void bubbleSortPlayers(Player\*\* players, int count): Sorts the players in descending order based on their points using the bubble sort algorithm.
* static void displayLeaderboard(Player\*\* players, int count): Displays the top 100 players (or fewer if there are less than 100 players) in the leaderboard.

**Design Choices**

**Array for Rounds:**

* Using arrays for round-specific answers, questions, and scores ensures fixed-size storage and easy access based on round indices.

**Static Methods for File I/O and Sorting:**

* Static methods like loadStudentsFromFile and bubbleSortPlayers are used to manage player data and sorting, allowing these operations to be performed independently of any specific player instance.

**Detailed Leaderboard Display:**

* The displayLeaderboard function is designed to show up to 100 players, which helps in managing large datasets without overwhelming the user interface.

# 3.0 SYSTEM SCREEN SHOTS

A screen shot of a computer program

Description automatically generated

Figure 2: Linked list Class (1)

A screen shot of a computer program

Description automatically generated

Figure 3: Linked List Class (2)

A screenshot of a computer program

Description automatically generated

Figure 4: Linked List Class (3)

A screenshot of a computer program

Description automatically generated

Figure 5: Linked List Class (4)

A screen shot of a computer code

Description automatically generated

Figure 6: Player Class (1)

A screen shot of a computer program

Description automatically generated

Figure 7: Player Class (2)

A computer screen shot of a program code

Description automatically generated

Figure 8: Player Class (3)

A screen shot of a computer program

Description automatically generated

Figure 9: Game Class (1)

A screenshot of a computer program

Description automatically generated

Figure 10: Game Class (2)

A screen shot of a computer program

Description automatically generated

Figure 11: Game Class (3)

A screen shot of a computer

Description automatically generated

Figure 12: Game Class (4)

A computer screen shot of a program

Description automatically generated

Figure 13: Binary Search Tree Class (1)

A screen shot of a computer program

Description automatically generated

Figure 14: Binary Search Tree Class (2)

A computer screen with many lights

Description automatically generated with medium confidence

Figure 15: Binary Search Tree Class (3)

A computer screen shot of a program code

Description automatically generated

Figure 16: Main Function (1)

A screen shot of a computer program

Description automatically generated

Figure 17: Main Function (2)

A black screen with white text

Description automatically generated

Figure 18: Program Output (Switching Decks if Discarded is empty)

A screenshot of a computer program

Description automatically generated

Figure 19: Program Output (Leader For all students output)

A screenshot of a computer

Description automatically generated

Figure 20: Program Output (Tree Leaderboard For First 30 Students)

# 4.0 Result, Summary and Reflection Section – Discussions

**Execution Times**

Different time complexities and efficiencies are exhibited by the operations of some other classes. In LinkedList class, insertion, and deletion, both have O(1) time complexity, which means these operations, will take constant-time. Insertion- Changes the Next Pointer of the Last Node and Deletion- Changes the Head Pointer. Time Complexity for Loading questions from the file is O(n) since it must read every line and then insert that question into the linked list. In addition, thanks to creating an array, copying elements into it, shuffling it with a shuffling algorithm and copying them back, shuffle takes O(n).

Insertion in the BinarySearchTree class typically takes O(log n) when the tree is balanced but can degenerate to O(n) when unbalanced. Time Complexity for In Order Traversal to Display the Tree O(n) Bubble Sort has a worse case time complexity of O(n^2), O(n) to display the leaderboard and quick sort is O(n log n).

**Observation and Thoughts During Development:**

**1. Code Structure:** The code is divided into several files, each of which has a certain related group of classes or functions. This modular approach helps code readability, maintainability, and allows for the ease of collaboration amongst developers.

**2. Class Design:** Classes like `Game`, `LinkedList`, and `Player` are designed well to ensure that related data and behaviours belong together in an object-oriented fashion. This object-oriented design increases code reusability and scalability.

**3. Error Handling:** There is an implementation of error-handling mechanisms with the use of exceptions. It has improved code robustness, providing proper error messages for guidance to the users in case of unexpected situations.

**4. Input Validation:** The system in place to validate user inputs will ensure that the entries made are within certain expected boundaries. This will prevent runtime errors and grant the user a glitch-free experience.

**5. Algorithm Choices:** Though it is easy to implement some algorithms, such as Bubble Sort, these may not be efficient when the system's data is large. Alternative algorithms for sorting, as well as for other operation choices, could be evaluated.

**6. Memory Management:** The application shows good memory management and handling practices, including dynamic memory allocation and deallocation, thus avoiding memory leaks and ensuring no resource wastage.

**7. File Handling:** File reading and parsing are done rightly to make sure that data from external files is correctly read into a program, easing update support for game content.

**8. Randomisation:** Shuffling the deck of playing cards adds an element of uncertainty or unpredictability to the game due to randomisation. This leads to the increase of replay value and overall player engagement.

**9. Documentation:** The code doesn't seem to have extensive comments or any documentation regarding complex algorithms or design decisions. The clarification of obscured variables through comments may also be used for better code understanding.

**10. User Interface:** The program centres only on the backend logics of the game without implementing the graphical or interactive user interface. This can be done so that a wider democratic audience can be attracted by the game and have better user experience.

**Strengths:**

**1. Modularity:** Code is structured in separate classes and functions, improving ‎modularity and reusability of the code. It is easy to understand, maintain and extend.‎

**2. Object-Oriented Design:** Classes (`Game`, `LinkedList`, `Player`) are used that ‎encapsulate data and behaviour according to the object-oriented approach. This helps ‎in improving code organization and readability.‎

**3. Error Handling:** Exception-based error handling for increased strength against ‎unexpected situations arising during file I/O operations. Error messages are ‎informative and clear for easy diagnosis and resolution.

**4. Input Validation:** The system has mechanisms in place that check the user input ‎validity to make the program stable and correct. This gives a better user experience ‎since the user will get directions on the correct input.‎

**5. Memory Management:** Correct memory management has been followed, so no ‎memory leaks are possible, with the proper utilization of resources. Dynamic memory ‎allocation and deallocation are properly managed to avoid memory leaks and ensure ‎efficient usage of memory.‎

**6. File Handling:** Reads and parses data from external files in the right way to ensure ‎the right way to manage and update game content in files easily. This supports ‎flexibility and ease of maintenance.‎

**7. Randomization:** Good implementation of randomization added to shuffling the deck ‎of cards to add uncertainty to the gameplay. This enhances replay ability and player ‎engagement.‎

**Weaknesses:**

**1. Algorithm Efficiency:** Some algorithms, like bubble sort in leaderboard sorting, may not ‎be the best when dealing with large volumes of data. Better algorithms could be used to ‎enhance efficiency with larger datasets.‎

**2. Scalability:** The simplicity in data structures and algorithms used by the code may bring ‎scalability issues with an increase in the number of players or questions. Changes must be ‎made on the data structures and algorithms to optimize the scalability and thus handle ‎bigger datasets.‎

**3. Documentation:** The code lacks extensive comments or proper documentation ‎elucidating complex algorithms or design decisions. Extensive documentation would ‎probably help in understanding the codebase better and, therefore, in easier ‎maintenance by future developers.‎

**4. User Interface:** The code is entirely oriented towards the back end logic of the game ‎without any graphical or interactive user interface. Implementing a GUI can boost the ‎user interface and experience, making the game even more appealing to a larger number ‎of people.‎

**Fixing Systemic Weaknesses and Future Work Approaches:**

**1. Algorithm Efficiency:**

- Research and implement better sorting algorithms, such as quicksort or mergesort, for more leaderboard efficiency. This will greatly help in terms of performance, especially as the number of players becomes very large.

- See if the data structures used in the game, especially with large data sizes, could use more efficient data structures, like hash tables or balanced trees, for player data to minimize getting the player data or updating leaderboards.

**2. Scalability:**

- Refactor the data structures and algorithms to ensure the scaling of the system as data sizes become larger.

- Implement strategies for resizing data structures or dynamic memory allocation techniques that can scale to many players and questions without performance loss.

**3. Documentation:**

- Improve code documentation with more comments and explanatory notes on complicated algorithms, classes, and functions. This will really help new people understand your codebase and will make maintaining and adding new features easier.

- Create developer documentation or README documents explaining the project structure, project dependencies, and how to use the platform so that one can easily onboard new contributors.

**4. User Interface:**

- Design an intuitive graphical user interface, using a toolkit like Qt or wxWidgets, that includes several features and functionalities for an elastic and attractive user experience.

- Develop and design interactive elements that provide users with visual feedback, increasing their interest in and engagement with the game.

- Include accessibility features that allow the game to be available for all and sundry, particularly people with disabilities.

**5. Performance Profiling:**

Perform performance profiling and benchmarking to identify performance bottlenecks and areas for optimization.

- Use profiling tools to analyse CPU and memory usage, identify hotspots in the code, and prioritize optimization efforts effectively.

**6. Testing and Validation:**

- Implement comprehensive unit tests and integration tests to validate the correctness and robustness of the codebase.

- Perform stress testing with large datasets and edge cases to ensure the system behaves as expected under various scenarios.

- Solicit feedback from users or beta testers to identify usability issues and areas for improvement in the user experience.

**Practical Experience and Feedback on Practical Implementation:**

**1. Setup and Initial Configuration:**

- Experience: The initial setup was mostly about configuring the development environment—compilers and setting up libraries. The file structure and ensuring that all dependencies are inserted just right did indeed take some time.

- Feedback: Documentation on the setup process and its prerequisites can be made clearer. A setup script or a Docker container can help further in the configuration of the environment.

**2. Code Compilation and Debugging :**

- Experience: It was easy to compile the code, and finding syntax errors was easy. However, to debug logical errors, it required going through the code flow and data structures in detail.

- Feedback: Integrating extensive logging mechanisms would assist in fast issue tracking. Likewise, a detailed flowchart or diagram of the program logic could help in easy understanding and debugging.

**3. File Handling and Data Loading :**

- Experience: Working with the file to load the questions, parse the data, and similar functionalities has worked as intended. File-handling code has multiple formats of the same, ensuring the data is accurate.

- Feedback: Error handling for file operations could be improved for a more specific way of giving feedback. Sample data files can be included with some checks allowing the data to be consistent and correct.

**4. User Interaction and Input Validation:**

- Experience: There were user inputs to be handled and validations were placed to ensure they are within bounds. Interactive prompts helped guide users.

- Feedback: The enhancement of user prompts with a more verbose explanation and examples is a good way to improve user experience. A timeout mechanism for user inputs might also be useful when the game is played in a real-time mode.

**5. Algorithm Performance:**

- Experience: Sorting and shuffling algorithms were good for the size of the current data set; they slightly showed a dip in performance with bigger data sets.

- Feedback: Algorithms and data structures must be optimized to perform better with large datasets. Profiling the code would assist in finding the bottlenecks for selective optimization.

**6. Memory Management:**

- Experience: Dynamic memory allocation and deallocation were working well with no apparent memory leaks. Replacing raw pointers with smart pointers may be a good way of improving memory management.

- Feedback: Memory profiling tools that could keep track of memory usage and identify leaks at an early stage would be a good investment in the development process. Of paramount importance is to ensure that there is proper cleanup in all parts of the code because it can be troublesome to locate leaks in long-running applications.

**7. User Interface:**

- Experience: The user interface, though being text-based, was good enough to make the game functional. It was one-dimensional, taking user inputs from the console.

- Feedback: Developing a GUI using a framework like Qt or SDL will greatly improve the user experience integrated with many nice functionalities. Some of these features include graphical images, buttons, and real-time feedback.

**8. Testing and Validation:**

- Experience: Manual testing was performed to validate various scenarios and edge cases. The small suite of automated tests was mainly prepared for the core functionalities.

- Feedback: An automated suite of unit and integration tests can not only form a basis for ensuring the reliability of the code but also make regression testing possible. A testing framework like Google Test would standardize the testing setup.

**9. Collaboration and Version Control:**

- Experience: Version control through Git enabled efficient collaborative development and change tracking. Branching and merging strategies made it easier to work on different features and bug fixes.

- Feedback: A more structured workflow, providing for code reviews and continuous integration, would go a long way in developing the code quality and collaboration. Documenting the coding standards and best practices would ensure uniformity across the team.

**New Solutions and Ideas to Address System Weaknesses and Inefficiencies**

To overcome the above-mentioned weaknesses and inefficiencies, there are several fruitful solutions and ideas that can be implemented. It will greatly boost performance with larger datasets if the sorting algorithm can be improved from the bubble sort to an effective one such as quicksort or mergesort. Parallel sorting ideas can further optimize the process. Data structures such as hash tables could be implemented for efficient, rapid player lookup, and balanced trees for maintaining the order of scores. Some basic scalability and reliability can be achieved by shifting to a database management system such as SQLite or MySQL.

There should be good documentation to make it maintainable. Code comments and their explanations can be increased in depth, and interactive documentation can be generated with the help of Doxygen, enabling new developers to better understand the codebase. It is important to develop a GUI using frameworks such as Qt or SDL, providing a great user experience which is more intuitive and visually appealing. Implementing touch and gesture controls will make the application more accessible (jaycontest, 2024).

Performance profiling, which greatly identifies the bottlenecks and refines them, is quite needed here. Lazy loading of questions and players will reduce the initial load time and memory footprint. Enhanced error handling mechanisms will give details of the errors, incorporating several error types with file operations/user input, while a comprehensive error logging system will help in better debugging.

Automated testing facilities like Google Test and continuous integration pipelines can assure consistency and reliability.

**Personal Thoughts on the Assignment**

This has perhaps been one of the most enriching experiences for me. I have been allowed to explore much more about software development such as how algorithm optimization, data structure efficiency, and proper user interface design can enhance the outcome of a perfectly working software. Moreover, how theoretical knowledge, e.g. sorting algorithms, and data structure selection, comes on the same practical performance and scalability level of a system.

I also got to appreciate the importance of proper documentation and testing. Proper documentation helps current development with trouble-free maintenance and understanding, while at the same time, facilitating future developers to understand and maintain the system. Similarly, testing—both individually at the unit level and performance profiling—is necessary to make robust and reliable software.

One challenge was improving system performance while keeping the code readable and simple. Sometimes, this took quite a bit of refactoring and applying golden principles in coding. Further, perhaps the challenge presented by integrating a graphical user interface is how to make the interface both intuitive and accessible.

To sum up, this project has proved the iteration process of software: there is always something new to be added, either by optimizing the current code, integrating other technologies, or simply by refining the user interaction. It has been a great learning journey that will inform the projects I do in the future.

# References

jaycontest. (2024, May 24). *Doxygen: The Importance of Software Documentation*. Jaycon | Product Design, PCB & Injection Molding. https://www.jaycon.com/doxygen-the-importance-of-software-documentation/

‌