**Phase 1: Identification of the problem**

The game will be set in a castle, where the player will be set captive on top of the highest tower of said castle, the objective of the game is for the player to escape by descending to the very bottom of the tower, where the player will find each floor locked and must find all keys to open the exit door, while they try to do this, an enemy will follow the player, a game over will be reached were the enemy to touch the player. The game will be divided in sections, each section will contain a different number of rooms, and the final room will be a race against the enemy where at the end the player would move on to the next section, up until the final section, where a mage would be controlling different enemies, the player would then have to collect three sets of keys, each keyset would open a lock, when all three are opened, the ending would be reached.

Each screen will be divided into tiles, each tile would be represented as a vertex in the logic behind, where they could contain a key, an enemy, or the player, each movement would be represented as the player object moving to the pertinent adjacent vertex, while also moving the graphic representation one tile to the indicated direction, where collision will be dictated by the existence of a connection in the given direction.

The enemies will path-find to the position of the player by finding the shortest path towards the player, in later levels, there may be hazards on the floor that affect the movement speed of anything that moves through it, the enemies must avoid them whenever possible. Apart from the conditions mentioned before, the enemy must also become the stop condition may it ever touch the player, touching defined as being in the same tile at the same time. The movement speed of the enemies is to be determined by both the room and the section they are in, the speed gets faster the further along the player is in the game, except for the last rooms in each section, where the speed of the enemy increases constantly.

* **Requirements analysis:**
  + **R1: User Interface:** As this is a game, there must be a way for the user to interact with the digital world they are presented with, therefore there must be a representation of the level, with differing graphics for obstacles depending on their behavior, there must also be a leaderboard with every record achieved by the player.
  + **R2: Database:** The database must hold a record of every single attempt the user has done, it must hold time, section and floor reached; the record must be in order of section reached, meaning a higher section would be higher, then the number of the last floor reached, and lastly the time they spent in game.
  + **R3: Playfield:** The playfield must allow both player and enemy to move freely in four directions. The playfield must be divided in tiles, each tile would be connected to others around it, if there is no connection to a certain direction the entity must not be able to move. The connections can vary, as some may be slower than other, this must be considered for the movement.
  + **R4: Player interaction:** There must be a way for the user to interact with the game presented to them, therefore there must be a representation of the position of the player in the map, this player object must react to the movement inputs of the user (arrow keys or wasd), this object must also be capable of “picking up” a “key” object when it collides with it (is in the same tile), this object must interact with the rest of the program to “open” a gate and proceed to the next level, this action would happen as enough key objects are collected.
  + **R5: Enemy Interactions:** In the playfield there must be an enemy whose function is to chase the player. The enemy must find the shortest path between it and the player and follow it, this path is to be updated each time the player moves, it must also take into account the obstacles that may be in the way of it and the player. When the enemy reaches the player (when they both are in the same tile), a game over state must be reached where the time, section and floor must be recoded and saved into the database. As the player advances through the floors and sections, the speed of the enemy must increase.
  + **R6: Endgame:** There must be a final floor where the player must avoid several enemies at a time, at the end, a small animation must be played showing the player that they reached the end and encourage them to go for a lower time.

**Phase 2: Research the necessary information**

**BFS:** The breadth

**BST:**

**AVL:** This algorithm is named after its creators, Adelson-Velskii y Landis developed a tree algorithm that uses the height and balance of every father node and reassigns its nodes to make the tree the most balanced it could be.

**GSON:** Gson is a library that can be used to convert objects into JSON format and vice versa. This tool can be useful to save and load the data of the scoreboard.

**Phase 3: Search of creative solutions:**

To solve this needs we require an algorithm able to accomplish the requirements described recently, this algorithm also must be in Java, including data structures as TAD’S that allow the correct operation of the game.

The list of the attributes that go along with the problematic of the functioning of the game go with those of the Player such as: name, score.

**Interaction solutions:**

**Alternative 1:**

The game has a graphic interface

**Alternative 2:**

The game has a console interface

**Database solutions:**

**Alternative 1:**

This alternative must include a database in where all the players and their scores will be registered, in a format of csv.

**Alternative 2:**

This alternative must include a database in where all the players and their scores will be registered, in a Json database.

**Alternative 3:**

This alternative must include a database in where all the players and their scores will be registered, in a txt database.

**Alternative 4:**

This alternative must include a database in where all the players and their scores will be registered, in a SQL database.

**Register Player solutions:**

**Alternative 1:**

Register on an array

**Alternative 2:**

Register on a BST

**Alternative 3:**

Register on a queue

**Alternative 4:**

Register on an arraylist

**Alternative 5:**

Register on a linked list

**Get the max score of the players solutions:**

**Alternative 1:**

Maxheap

**Alternative 2:**

Inorder

**Phase 4: Transition of the ideas to preliminary designs**

**These ideas** **don’t meet the requirements**

**Interaction solutions:**

**Alternative 2:**

The Console interface compared to the graphic interface is the worst alternative because it will be impossible to move smoothly, also the graphics will be ASCII characters. So, it won’t be the most optimal alternative.

**Database solutions:**

**Alternative 1:**

The .csv alternative is not viable because its format is based on data separated by commas, so the organization and search with this alternative won’t be the most optimal.

**Alternative 4:**

.SQL is an alternative that can accomplish the requirement of store scores, but this is not the most optimal alternative to search and order said scores.

**Register Player solutions:**

**Alternative 1:**

Register on an array, we discard this idea because the design of the game is that the registration process can be automatable, and it’s durable, and with the array we have limited positions, and it depends on how many players have registered for us to continue increasing the size of the array.

**Alternative 3:**

Register on a queue, for the game this option may be suboptimal.

**Alternative 5:**

Register on a linked list, this option it´s too slow to compile added to the fact that with the weight of the entire algorithm the program will be slower.

**Get the max score of the players solutions:**

**Alternative 1:**

Using Maxheap won’t be the most optimal alternative because it will bring the greatest score, this is alright, but we need to bring the top 10 scores. So, we must make a loop to accomplish the top 10 requirement part.

**These ideas meet the requirement**

**Interaction solutions:**

**Alternative 1:**

The graphic interface is the most optimal alternative because we can set parameters to all the game functionalities, also we can

**Database solutions:**

**Alternative 2:**

This alternative is viable because this method of storing data can be useful to represent objects as a string, this string will be stored in a file, Then, we can load said file and it will automatically generate the objects. This can be done with the gson library.

**Alternative 3:**

This alternative is viable because we can use the same method of the alternative 2, but in this case, we are using a .txt file instead of a .json.

**Register player solutions:**

**Alternative 2:**

Register on a BST, this alternative is suitable because we have several player nodes with the values: name and score, and what is going to do the BST is organize the tree with the scores, with that we fulfil the requirement 2, in which we need an organization of scores to save the players in the database.

**Alternative 4:**

Register on an ArrayList, this alternative is also viable because storages the node data of the players, although it doesn’t generate many pluses to another requirement.

**Get the max score of the players solutions:**

**Alternative 2:**

Since we are considering BST, the inorder method will be perfect to get the top 10 scores.

**Phase 5:**

**Database solution:** For a makeshift database, it was decided to evaluate two attributes: Ease of implementation (5 simplest, 1 hardest) and versatility (5 most versatile, 1 least versatile) The results from the evaluation are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ease of implementation | Versatility | Total |
| .txt | 4 | 4 | 8 |
| .json | 3 | 4 | 7 |

As shown above, both solutions are remarkably similar, but, as the .txt implementation is easier to implement, it is the chosen implementation.

**Register player solution:** For the register player method in which the database is to be loaded as the program is running, it was decided to evaluate two attributes: sorting efficiency (5 most efficient, 1 least efficient) and ease of implementation (5 easiest, 1 hardest) The results are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sorting efficiency | Ease of implementation | Total |
| BST | 4 | 5 | 9 |
| ArrayList | 2 | 4 | 6 |

As shown above, the BST has the best sorting efficiency, and this is very important to get the top 10 of the scores, so it will be the chosen implementation.