How randomized progression is built in a complete LevelMap

The randomization is done solely by one class, the **LevelRandomizer**, with **ProgressTree** data structure as helper to build and store Progress Information. **ProgressTree** class is destroyed at the end of the progress, since it only helps setting the parameters for the **PointOfInterests** in the LevelMap. Once those parameters are set, the progression randomization is accomplished.

There is no need tu further store the ProgressTree object which generated the progression, thus it is deleted.

This mechanism also removes deadlock situations, in which a room contains the key **keyA** which unlocks **doorA**. But **doorA** is locked and it needs **keyB**, but **keyB** is locked behind **doorB**, which is unlocked by **keyA.**

The only other thing to know is that PointOfInterests in the same room share the same **roomID**.

LevelMap class:

LevelMap
+ lockeds: HashMap<Location, PointOfInterest>
+ unlocks: HashMap<Location, PointOfInterest>
+ others: HashMap<Location, PointOfInterest>
+ rooms: HashMap<Location, Room>

Class responsible of storing and handling Level data. The matrix represent the collisions (walls, forniture and other solid inert objects). The PointOfInterest are all the objects that the robber can interact with. Those include Doors, Keys, Alarms, Safes, SecurityGuards, and so on). Moreover, to make room chaining possible, each LevelMap can have PointOfInterest that are Rooms: during LevelBuilding the game will generate other rooms (LevelMaps) where those points are. For example corridors are LevelMaps which only contain PlayerEntryPoint and Rooms as PointOfInterests.

PointOfInterest class

PointOfInterest
+ pointType: enumPointType
+ size: (int, int)
+ requiredKeyID: int
+ roomID: int

Class that represents an interactive object on the map and defines its behaviour. They can be anything. LevelBuilder computes their data to set progression (which key opens which door?), puzzles (which minigame will deactivate an alarm?) and challenges/map obstacles (which zoned-alarm will be put in this part of the room?).

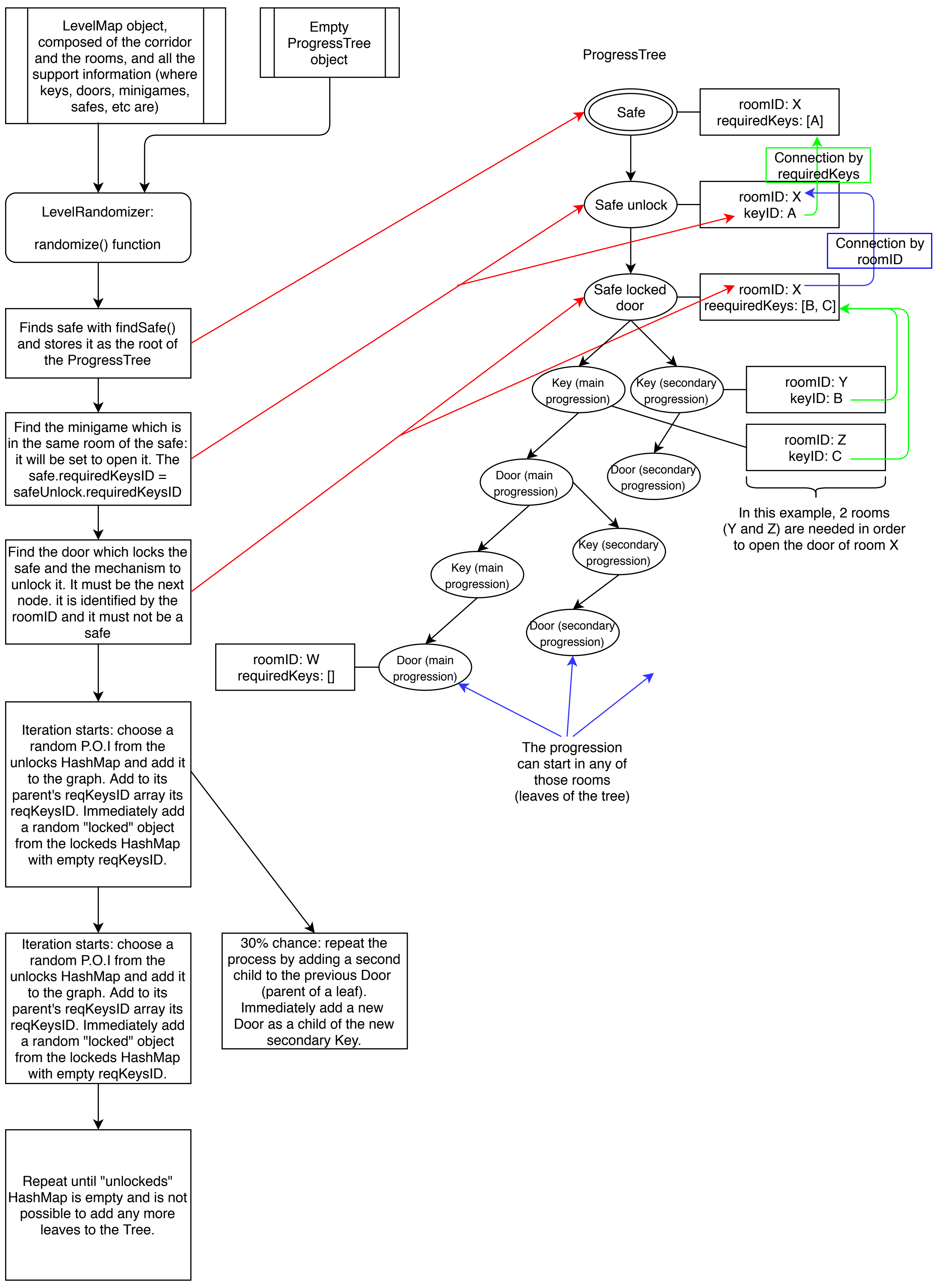
NOTE: roomIDs of the P.O.I. will be already set in the LevelBuilder class, which act *before* this class will be run. This is an assumption that LevelRandomizer class needs and is satisfied by running LevelBuilder class first on a LevelMap. Refer to document "Level Building Process" for more details.

ProgressTree class

ProgressTree<T>
+ getRoot(): Node<T>
+ insert(T, Node<T>): boolean
+ contains(T, Node<T>): boolean
+ getNode(T key, Node<T>): Node<T>

Generic class. Very similar to a generic BinaryTree. One difference is that on insert (T key, Node<T> parent), if the parent's left child is null, then it will set a node Node with the key as the left child of Node parent, otherwise right child. Right node cannot be non-null if left child is null. This is an assumption widely used in the LevelRandomized class and is a property of this data structure.

Phase 1: Progress Tree Building



LevelMap object, composed of the corridor and the rooms, and all the support information (where keys, doors, minigames, safes, etc are)

Empty ProgressTree object

LevelRandomizer:

randomize() function

Finds safe with findSafe() and stores it as the root of the ProgressTree

Find the minigame which is in the same room of the safe: it will be set to open it. The safe.requiredKeysID = safeUnlock.requiredKeysID

Find the door which locks the safe and the mechanism to unlock it. It must be the next node. it is identified by the roomID and it must not be a safe

Iteration starts: choose a random P.O.I from the unlocks HashMap and add it to the graph. Add to its parent's reqKeysID array its reqKeysID. Immediately add a random "locked" object from the lockeds HashMap with empty reqKeysID.

30% chance: repeat the process by adding a second child to the previous Door (parent of a leaf). Immediately add a new Door as a child of the new secondary Key.

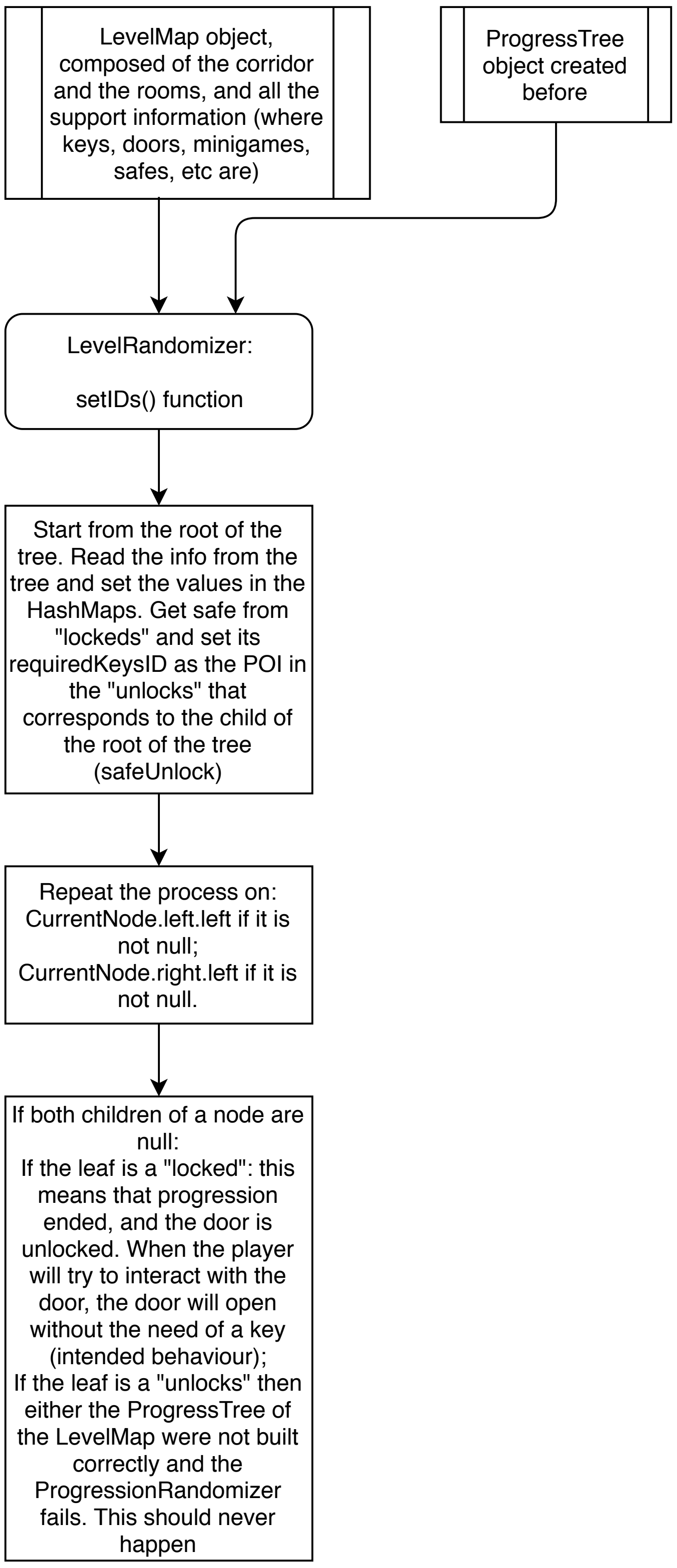
Iteration starts: choose a random P.O.I from the unlocks HashMap and add it to the graph. Add to its parent's reqKeysID array its reqKeysID. Immediately add a random "locked" object from the lockeds HashMap with empty reqKeysID.

Repeat until "unlockeds" HashMap is empty and is not possible to add any more leaves to the Tree.

In this example, 2 rooms (Y and Z) are needed in order to open the door of room X

The progression can start in any of those rooms (leaves of the tree)

Phase 2: PointOfInterest data setting



LevelMap object, composed of the corridor and the rooms, and all the support information (where keys, doors, minigames, safes, etc are)

ProgressTree object created before

LevelRandomizer:

setIDs() function

Start from the root of the tree. Read the info from the tree and set the values in the HashMaps. Get safe from "lockeds" and set its requiredKeysID as the POI in the "unlocks" that corresponds to the child of the root of the tree (safeUnlock)

Repeat the process on:
CurrentNode.left.left if it is not null;
CurrentNode.right.left if it is not null.

If both children of a node are null:
If the leaf is a "locked": this means that progression ended, and the door is unlocked. When the player will try to interact with the door, the door will open without the need of a key (intended behaviour);
If the leaf is a "unlocks" then either the ProgressTree of the LevelMap were not built correctly and the ProgressionRandomizer fails. This should never happen