Project 2: `donjon`

Summer of Programming 2021
Department of Computer Science
University of Copenhagen

August 12, 2021

1 Introduction

When storing and loading data, it is important to consider how to format said data, such that good efficiency is achieved.

For this project, you will be making a program for storing and loading the layout of simple *dungeons* (levels) in a *dungeon crawler* game.

2 Dungeon layout

All dungeons are composed of rooms connected by corridors. Each room has 0 to 2 *descending* corridors, which lead deeper into the dungeon, and exactly one *ascending* corridor, which leads further back to the entrance. There is always exactly one entrance room, which does not have an ascending corridor, but may only have descending corridors. A dungeon can thus be modelled as a *binary tree*[1], as seen in figure 1.

3 `donjon` specification

At its most basic, `donjon` should have two functionalities:

[1] It's not exactly a binary tree, since in the case of a room with single descending corridor, we do not distinguish said corridor as being either left or right descending.

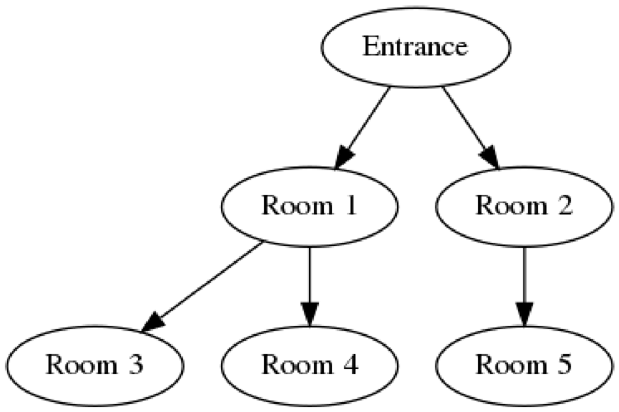

Figure 1: Example layout of a dungeon.

3.1 Generating dungeons

The user should be able to generate random dungeons from the command line by running `donjon` with the `-g` option:

```
./donjon -g <n> <output>
```

where `<n>` is the lowest level of the dungeon - that is, the maximum number of corridors that can be descended from the entrance in a row before reaching a dead end - and `<output>` is the name of the output file, containing the dungeon description formatted in a manner of your choice.

A lowest level of 0 or less should result in an error.

3.2 Displaying dungeons

The user should be able to print generated dungeons by running `donjon` with the `-p` option:

```
./donjon -p <input>
```

where `<input>` is the name of the input file containing a dungeon description (intended to be generated by the `-g` option).

Running `donjon` with this option should result in a printing of some textual representation of the dungeon layout. For the dungeon in figure 1, the program might print:

```
The entrance leads to room 1 and room 2.
Room 1 leads to room 3 and room 4.
Room 2 leads to room 5.
Room 3 is a dead end.
Room 4 is a dead end.
Room 5 is a dead end.
```

A nonexistent input file, or an input file with an invalid dungeon, should result in an error.

4 Hints

- Before you start implementing either of the two functionalities, consider how to format your dungeon file. A possible formatting might represent the dungeon in figure 1 in the following manner:

  ```
  0 1 2
  1 3 4
  2 5
  3
  4
  5
  ```

  Note that we are not concerned with saving any data about the rooms other than their relative position, so the room numbers you use may be different, but the dungeon structure should be the same regardless.

  The format you choose should be *general* (should be able to represent any arbitrary legal dungeon), but you may design it to be general only *to some extent*. For example, your format may only be able to represent dungeons with 256 rooms or less.

- It may also help to work out an abstraction for representing a dungeon in your program. In C#, this could be a Dungeon class which might look like:

```
public class Dungeon {
  public Room entrance;

  public Dungeon () {}
}

public class Room {
  public Room ascend;
  public Room left_descend;
  public Room right_descend;

  public Room () {}
}
```

  (good adherence to SOLID principles notwithstanding)

5 Challenge

If you are feeling particularly creative, try to design a representation that is general for any dungeon of at least level 5, that uses as few bytes as possible.