Blocky

Topics: Spatial Data Structures/QuadTree, Recursive data structures, Tree Search, Algorithm analysis, Testing

Due Date: March 15 @ 11:59pm

Learning goals

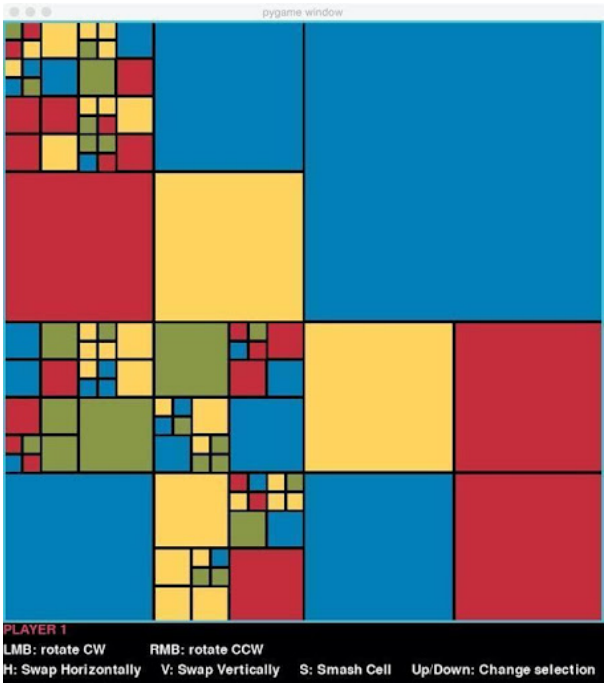
By the end of this assignment, you should be able to:

- Model hierarchical and spatial data using trees
- Implement recursive operations on trees (both non-mutating and mutating)
- Convert a tree into a flat, two-dimensional structure
- Explain and perform runtime analysis of the code you wrote

Starter Code&Design Analysis Template

Background & Introduction: The Blocky Game

Blocky is a game with simple moves on a simple structure, but like a Rubik's Cube, it is quite challenging to play. The game is played on a randomly-generated game board made of squares of different colors, such as this:



The goal of the game is to put the most possible units of a given color c on the outer perimeter of the board. The player's score is the total number of unit cells of color c that are on the perimeter. There is a premium on corner cells: they count twice towards the score. After each move, the player sees their score, determined by how well they have achieved their goal. The game continues for a certain number of turns or until the user runs out of moves (in this assignment, we will allow an unlimited number of moves).

Now let's look in more detail at the rules of the game and the different ways it can be configured for play. There are four key game parts to understand: the board, block levels, moves, and scoring.

The Blocky Board

We call the game board a 'block'. It is best defined recursively. A **block** is either:

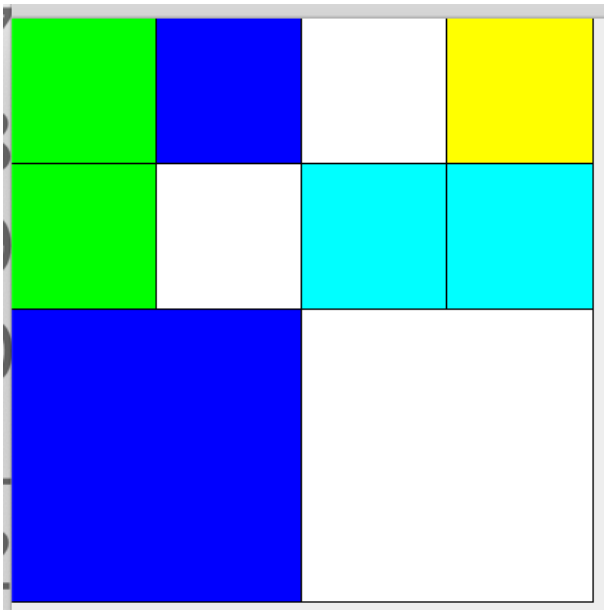
- A square of one color, or
- A square that is subdivided into 4 equal-sized blocks.

The largest block of all, containing the whole structure, is called the **top-level block**. We say that the top-level block is at **level 0**. If the top-level block is subdivided, we say that its four sub-blocks are at level 1. More generally, if a block at level k is subdivided, its four sub-blocks are at level k+1.

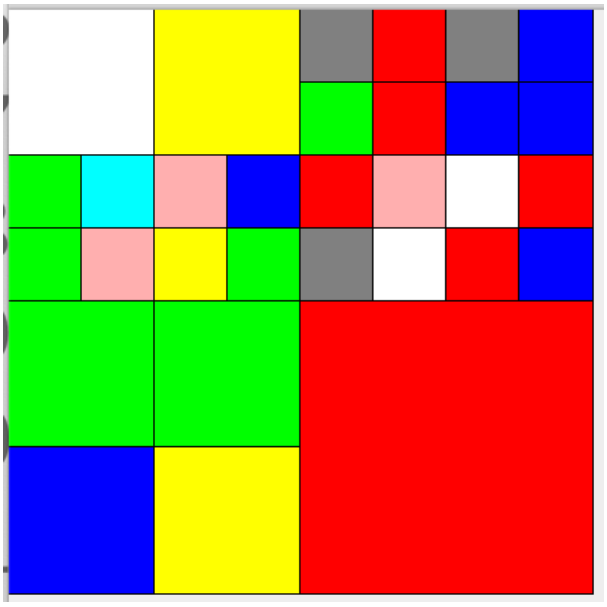
A Blocky board has a **maximum allowed depth**, which is the number of levels down it can go. A board with maximum allowed depth 0 would not be fun to play on – it couldn't be subdivided

beyond the top level, meaning that it would be of one solid color.

This board was generated with a maximum depth of 2:



This board was generated with a maximum depth of 3:



As you can see, the deeper the board the more blocks you might have.

For scoring, the units of measure are squares the size of the blocks at the maximum allowed depth. We will call these blocks **unit cells**.

> **TESTING TIP**
> For simplicity, we recommend limiting the maximum depth to 3 or 4. It's possible to allow a further maximum depth but testing may become more difficult.

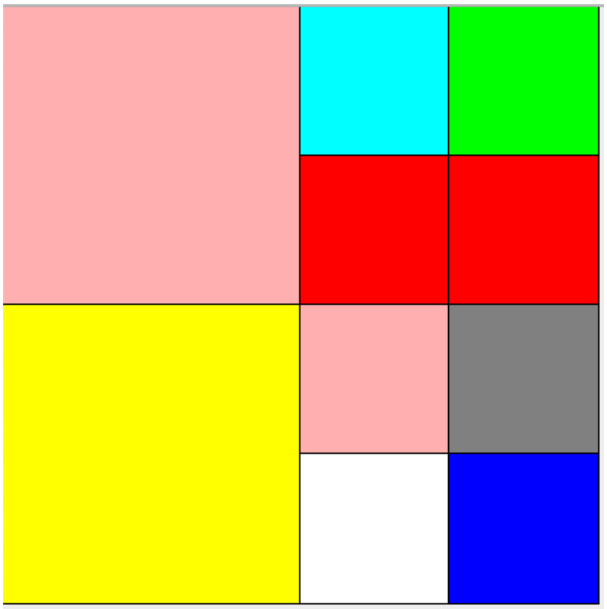
Moves

These are the moves that are allowed on a Blocky board:

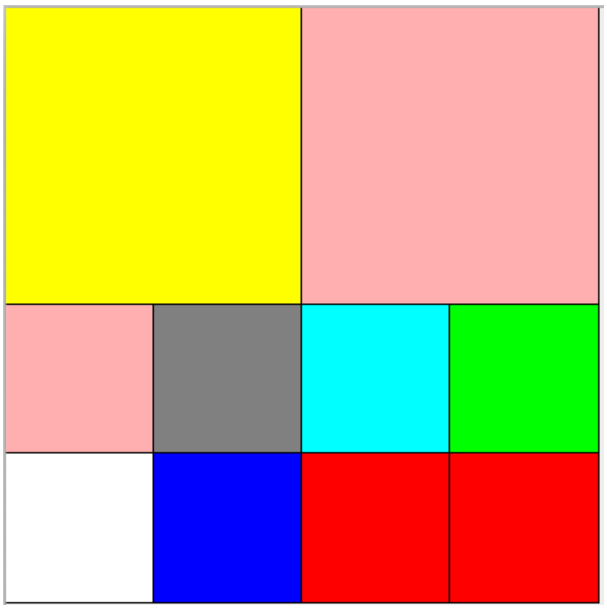
1. **Rotate** the selected block (90 degrees) clockwise. Implemented in `Block.java`
2. **Swap** two blocks (and their sub-blocks if any). Implemented in `Game.java`
3. **Smash** the selected block, giving it four new, randomly generated sub-blocks. Smashing the top-level block is not allowed – that would be creating a whole new game. And smashing a unit cell is also not allowed since it's already at the maximum allowed depth. Only leaf nodes can be smashed. Implemented in `Block.java`

Choosing Blocks & Levels

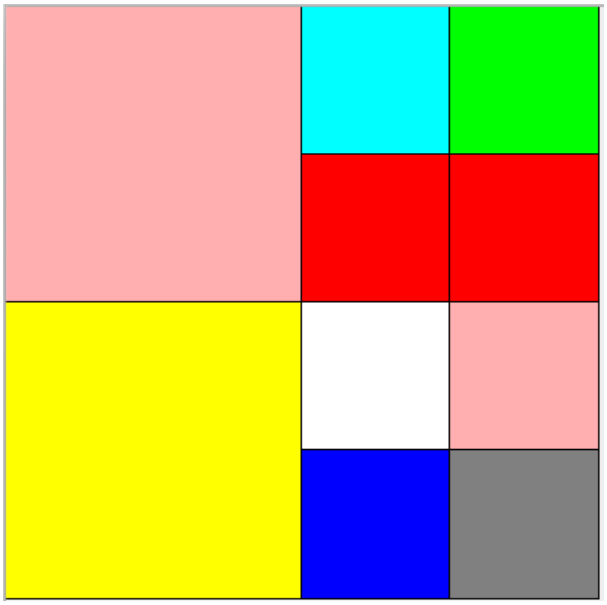
What makes moves interesting is that they can be applied to **any block at any level**. For example, if the user selects the entire top-level block for this board:



and chooses to **rotate** it, the resulting board is this:



But if instead, on the original board, they rotated the block with id 3 which is one level down from the top-level block in the bottom right-hand corner (block id labeling will be explained more in detail below), the resulting board is this:



Of course, there are many other blocks within the board at various levels that the player could have chosen.

Goals & Scoring

At the beginning of the game, the player is assigned a **target color** for the perimeter goal.

The player must aim to put the most possible blocks of a given target color c on the outer perimeter of the board. The player's score is the total number of cells of color c that are on the

perimeter. There is a premium on corner cells: they count twice towards the score.

Assignment Steps & Starter Code

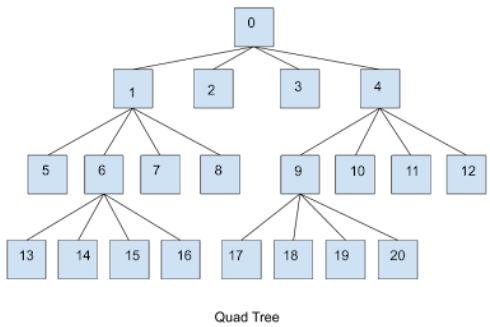
Please download the starter code files. Do not forget to test your code as you implement your solution. For an example of how we construct specific QuadTrees to test, take a look at the `Test.java` class.

Step 1: Understand the Block data structure and the Game class

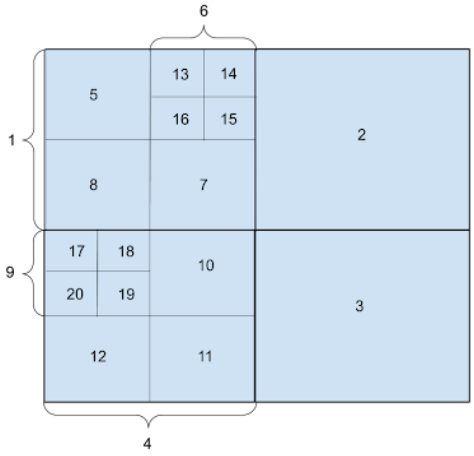
We will use a tree (QuadTree) to represent the nested structure of a block. Our trees will have some very strong restrictions on their structure and contents, however. For example, a node cannot have 3 children. This is because a block is either solid-colored or subdivided; if it is solid-colored, it is represented by a node with no children, and if it is subdivided, it is subdivided into exactly four subblocks.

How are the blocks numbered?

The blocks are numbered using a breadth-first traversal. To learn more about breadth-first searching, review the chapter from the book here, including the pseudocode. The image below shows the mapping of quadtree nodes to blocks ids. Note that only smashed blocks have their children numbered. The maximum depth of this blocky game is 3:



Quad Tree

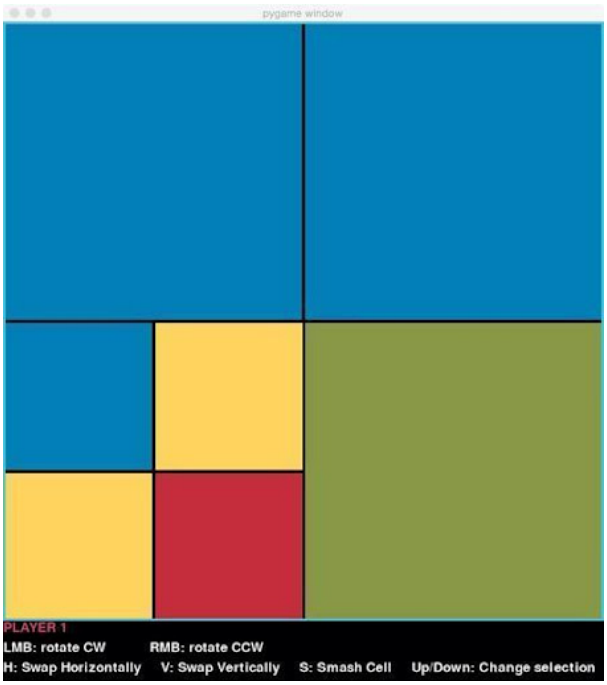


Open the `IBlock` interface and the implementing `Block` class. Read through the documentation carefully. Do not change any attribute names.

Open the `IGame` interface and the implementing `Game` class. Read through the class documentation carefully. The Game class represents an instance of the blocky game. It creates and maintains the Quadtree. The Game class performs the swap operation and computes the score of the player.

1. Open `Block.java` and implement the constructor and the attributes' getters and setters.
2. Manually draw (construct) the Block data structure corresponding to the game board below, assuming the maximum depth was 2 (and notice that it was indeed reached). **In this assignment, we will assume that the top-level block's top-left point is at (0,0) and its bottom-right point is at (8, 8).**

Use the `TestBlocky` class to check your code. To display the Block data structure, we add it to the GameFrame instance (using the addQuad method) and call `display()`.



Step 2: Initialize the game

With a good understanding of the data structure, you are ready to implement the Game and the Block classes.

1. Open `Block.java` and implement the `smash()` method. Verify that smash randomly assign colors to subblocks.
2. Now that we have the smash method ready, we can generate random boards. This is what function `randomInit` is for.

3 Create a `Game.java` class that implements `IGame.java`. Implement the constructor and `randomInit`. The method is outside the Block class because it doesn't need to refer to a specific Block.

> `randomInit` **IMPLEMENTATION STRATEGY**
> If a Block is not yet at its maximum depth, it can be subdivided; this function must decide whether or not to do so. To decide:
>
> - Use the function Math.random to generate a random number to randomly select a Block in the tree.
> - Subdivide if the Block at the random index is not already at maxDepth.
> - If a Block is not going to be subdivided (smashed), use a random integer to pick a color for it from the list of colors in `IBlocks.COLORS`. Notice that the randomly generated Block may not reach its maximum allowed depth. It all depends on what random numbers are generated.

Check your work: Use `TestBlocky.java` to confirm that your `smash()` and `randomInit()` methods work correctly.

Step 3: Complete the Block class

Implement the rest of the methods in the Block class **Check your work:** Thoroughly test your Block class and use `TestBlocky.java` to verify that your implementation is correct. Name your test class `BlockTest.java`

Step 4: Complete Game class

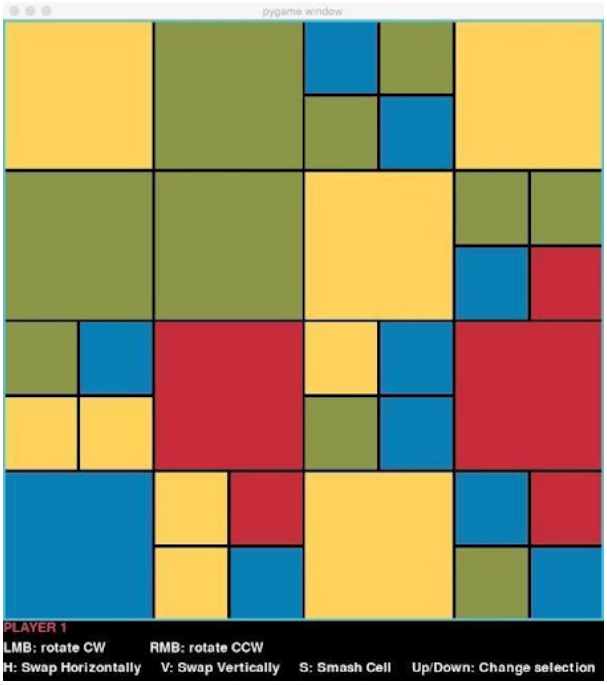
Now we have enough pieces to assemble a rudimentary game! Implement all the methods in `Game.java` except for `flatten()` and `perimeterScore`. **Check your work:** Thoroughly test your Game class and use `TestBlocky.java` to verify that your implementation is correct. Name your test class `GameTest.java` At the end of Task 4, you should have a functioning game without the scores.

Step 5: Implement scoring for perimeter goal

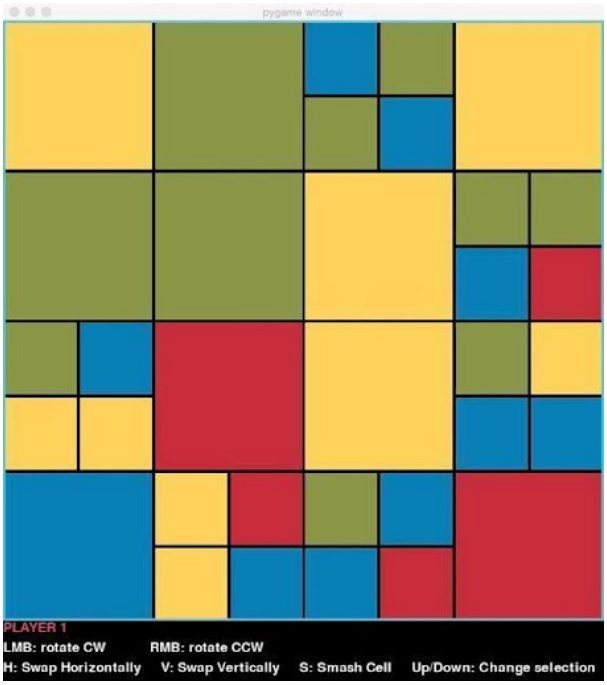
Now let's get scoring working. The unit we use when scoring against a goal is a unit cell. The size of a unit cell depends on the maximum depth in the Block. For example, with a maximum depth of

4, we might get this board:

If you count down through the levels, you'll see that the smallest blocks are at level 4. Those blocks are unit cells. It would be possible to generate that same board even if the maximum depth was 5. In that case, the unit cells would be one size smaller, even though no Block has been divided to that level. Notice that the perimeter may include unit cells of the target color as well as larger blocks of that color. For a larger block, only the unit-cell-sized portions on the perimeter count. For example, suppose maximum depth was 3, the target color was red, and the board was in this state:



Only the red blocks on the edge would contribute, and the score would be 4: one for each of the two unit cells on the right edge, and two for the unit cells inside the larger red block that are actually on the edge. (Notice that the larger red block isn't divided into four unit cells, but we still score as if it were.) Remember that corner cells count twice towards the score. So, if the player rotated the lower right block to put the big red block on the corner:



the score would rise to 6.

Now that we understand these details of scoring for a perimeter goal, we can implement it.

> `perimeterScore` **IMPLEMENTATION STRATEGY**
> It is **very difficult** to compute a score for a perimeter goal through the tree structure. (Think about that!) The goal is much more easily assessed by walking through a two-dimensional representation of the game board. So, your next task is to provide that possibility: In the Game class, define the method `flatten`.
>
> Now that your board can be represented differently, now implement the `perimeterScore` method in class Game to truly calculate the score. Begin by flattening the board to make your job easier!

Check your work: Now when you play the game, you should see the score changing. Check to confirm that it is changing correctly.

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to style and documentation warnings raised by the IDE. Fix them!

- Read through and polish your internal comments.
- Remove any code you added just for debugging, such as print statements.
- Remove the word “TODO” wherever you have completed the task.
- Take pride in your gorgeous code!

Grading

Your grade will consist of points you earn for submitting the homework on time and for the final product you submit.

The assignment is worth 205 points.

Description	Points
Autograder	130
Algorithm Analysis Document	5
Documentation/Style	15
Testing (correctness & code coverage)	50
Readme	5

This assignment was adapted by Eric Fouh. It was originally developed by Diane Horton and David Liu. It has been most recently updated by Harry Smith & Daniel Paik in 2023.
