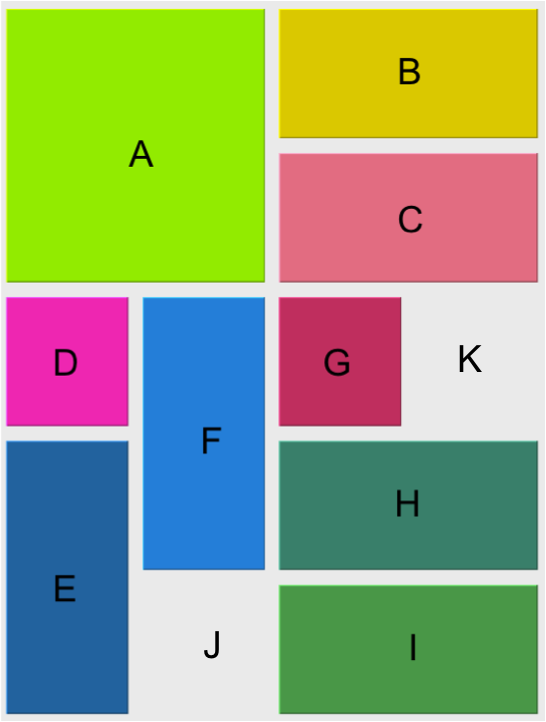
***Division of labor***

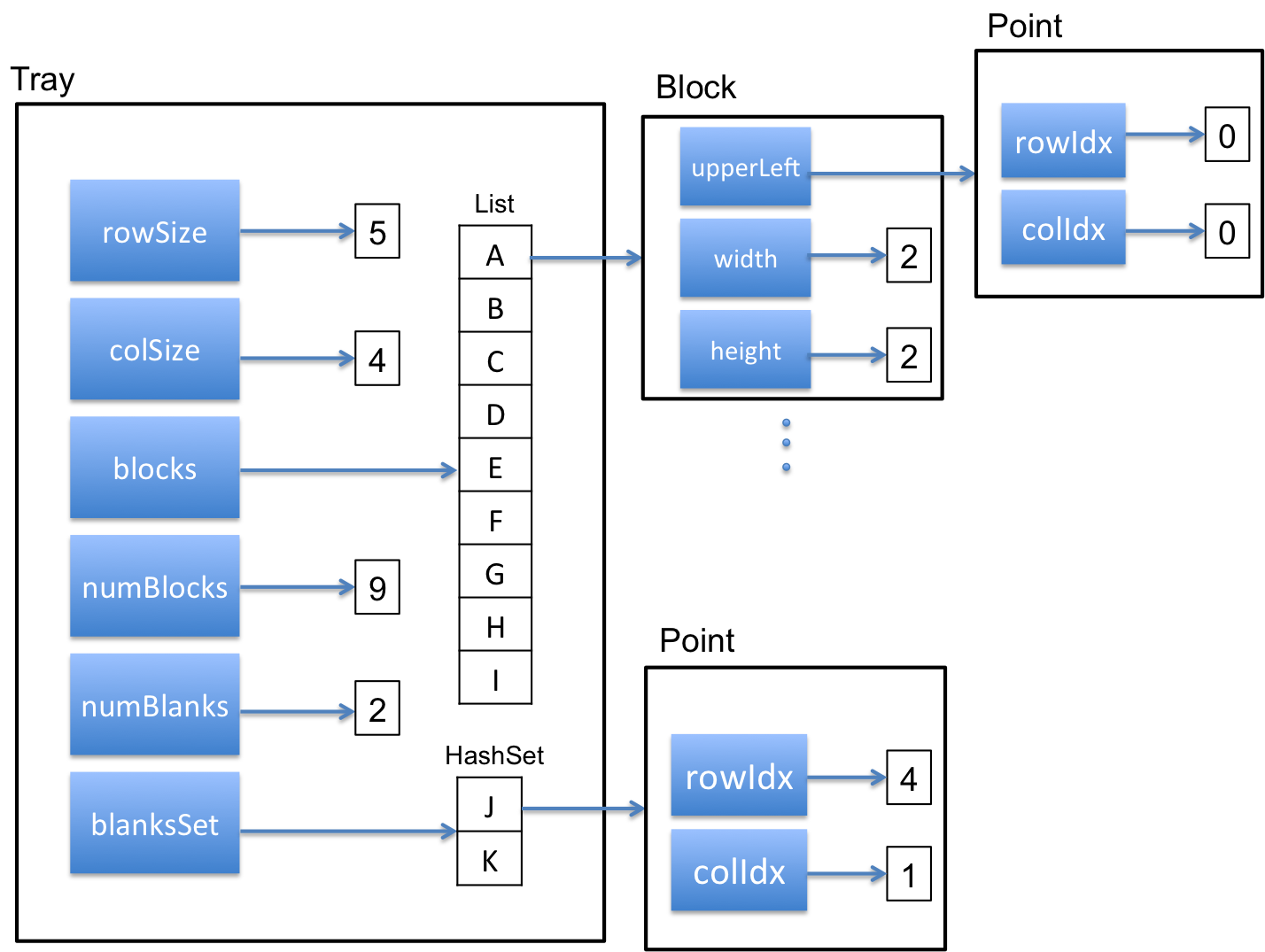
We began the project by assigning tasks during each group meeting. A recorder would document the constant updates during that meeting. A pair would address the programming sections, while the last member would consider tests for that day’s developments. These tasks would occasionally rotate. This was done to avoid Project 2 mishaps, such as lack of constant documentation and test-driven development. As the project progressed, each member found their area of interest and focused their attentions on optimizing it.

The Block class and Point class were a collaborative effort between all members. The JUnit tests written for the Block, Point, and Tray class were also collaborative efforts by the group. The Direction class, Direction, QueueFringe, SolveFringe, SolveFringeElement, StackFringe, Tray class, and Solver class were extensive works of YJ with input from other members.

***Design***

Below is a diagram showing the correspondence between an abstract tray and our tray implementation. Further descriptions can be found from the attached Javadocs of all the project classes, which provide information on all classes, methods, and variables.



******

***Debugging***

Our debugging output facility and how to enable it is described in length in the attached Javadocs. The isOK method was used to test that the following invariants of the Tray class were satisfied:

* The dimension of every Block must fit in the dimension of this Tray.
* The sum of the areas of every Block must not be bigger than this Tray’s area.
* This Tray must have a dimension bigger than or equal to 1X1.
* The number of blocks must not be bigger than this Tray's area.
* The number of blanks must not be bigger than this Tray's area.
* All four endpoints of every Block must be inside this Tray.
* No Block collides with the other Blocks in this Tray.
* numBlocks must equal blocksList.size().
* numBlanks must equal blanksSet.size().
* The sum of the areas of every Block, combined with numBlanks, must be equal to this Tray's area.

An IllegalStateException is thrown when any of the invariants has been violated. The isOk method was useful in revealing bugs in the program. They were quickly and handled as encountered. This allowed us to have greater confidence that all the were being maintained.

***Evaluating tradeoffs***

Throughout the Javadocs, we used Visual VM Memory Analyzer to evaluate results of our design choice (please refer to attached Javadocs for reference).

A Tray implicitly makes use of two distinct data structures. The first data structure is an ArrayList of Blocks. We chose to use ArrayList because our puzzle solving process "looks up" Blocks a lot, when it moves them around. So we needed an O(1) process for the following task: given a Block, find the Block in a Tray as quickly as possible. ArrayList was the way to go.

The second data structure is a HashSet of the blank Points within a Tray. Such a structure is necessary when our search process performs blank-wise search (refer to Solver.java for information on blank-wise search). An important aspect about this collection of blank Points is that there will be a lot of searches, additions and deletions performed on this collection. When a Block gets moved, some of the old blank spots will now be occupied, and there will be as many new blank spots. Hence, we needed a structure that can search, add and delete in O(1) time. HashSet was the way to go. ArrayList does not work in this case; although search can be performed in O(1), additions and deletions are O(N).

**Disclaimers**

Our project proved to be generally successful, but with more time, we could have implemented some improvements. One would be to change the way we iterate over blocks. Currently, we use indices to iterate, but using an iterator format would have been more efficient. We could have also come up with a better heuristic function.

***Program development***

Our initial approach to this project was to construct three classes: a Point class, Board class, and a Tray class. We begin by defining a block as an object with a Point for the upper-left position and a width and height, calculated using its lower-right position. A Tray is defined as an ArrayList of blocks. With this configuration, our assumption was that moving blocks would not be very difficult. We thought to use a HashSet in order to detect configurations that have previously been seen. When we recursively search for a conclusion, we would have to memoize. The entire tray as a whole will be hashed.

Our next step was to meet with Professor Clancy and receive feedback on this approach. Professor Clancy clarified that this program would not be implemented explicitly using a tree. Instead, there is an implicit tree resulting from considering different move options. The professor also stressed writing the program in such a way so that you can “plug” and “unplug” different algorithms and data structures. Other results from the meeting was that using Javadoc was encouraged and could be included as part of the final write-up. We also decided on an alternate means of sharing files amongst members through GitHub.

Beginning the project by creating these fundamental classes allowed us to visualize a better approach to how Solver would be implemented. Once our Point, Block, and Tray structures were written, we had solid structures that could be manipulated.

The testing for each of these three classes soon followed. Our goal was to practice blackbox testing and write tests using only the Javadocs within the class as reference. This allowed the tester to better consider boundary cases.

Solver was the next class written. The first problem tackled was reading the initial configuration and representing this as a Tray object. The final configuration was represented as a list of desired blocks. These would then be passed into the solve method which would make use of the Move class to find a solution. Solver was tested using the given puzzles. After significant progress was made, the group tested Solver on lab computers to ensure that all easy, medium, and hard puzzles resulted in accurate output from Solver.

Later additions include the following classes: Direction, QueueFringe, SolveFringe, SolveFringeElement and StackFringe. These were done after the initial implementation of Solver to address speed and efficiency.