Lab 3 – Red-Black Tree of Card Hands [100 Marks]

A Partially Autograded Evaluation with Project Integration

IMPORTANT: **Your submission must compile without syntactic errors to receive grades. Non-compilable solutions will not be graded.**

IMPORTANT: **You must upload your HandsRBT.txt and ModelCode_CardGame.txt to *Avenue Dropbox* before the deadline to receive grades.**

Lab Deadline

The deadline for Lab 3 is Saturday, February 28, 2025, at 11:59 pm.

This lab was designed mostly by Prof. Scott Chen.

Deliverables

You must upload on or before the posted deadline the following items to the Avenue dropbox:

- HandsRBT.txt and ModelCode_CardGame.txt (i.e. change .java extension to .txt)
- You further will demonstrate your Model Code to the TA.

Premise

Labs 2 and 3 of the Winter 2026 COMPENG 3SM4 are the two software development iterations of the underlying logic of a simplified 5-card poker card game. The completed work in lab 3 will then be sent to the *imaginary* front-end design team to add fancy GUIs and art drawings for production. Your job as an intermediate developer is to collaborate with other developers, who have been working on other modules of the game, implement the two critical data structures to support the game logic, and assist in integrating the data structures into the main game logic.

Lab 2 – Max Heap of Card Hands

> This lab focuses on completing the logic implementation of the "Aggressive AI", which will pick always the most aggressive 5-card hand to compete against the player. Knowing the specification, you determined that Max Heap is one of the most suitable data structures, and will deploy the Max Heap class using the existing Cards and Hands classes designed by your senior colleagues.

Lab 3 – Red Black Tree of Card Hands

> This lab focuses on completing the implementation of Red-Black Tree of Card Hands to help the player identify whether the chosen 5-card hand is a valid hand for competition. At the first glance, RBT is an overkill of this application purpose, because the Hands class already has a method to determine valid hands. However, the game UI design team wants to be able to display minimally 10 different recommended hands in the finalized version of the game, thus requiring you to lay the groundwork. As a result, you now have to implement the RBT of Card Hands to support the UI design team features.

The end result of the two labs will be a command-line testable game logic prototype that enables more feature-rich game designs for other collaborating teams.

Objective Overview

Lab 3 focuses on the implementation of Red-Black Tree of card hands, and the integration of the Red-Black Tree module and the Max Heap module from Lab 2 to complete the 5-card game with *Aggressive AI* behaviour. Note that the provided Red-Black Tree of card hands has some of the features already implemented for you. You only need to complete the missing implementations indicated in the lab manual.

The scope of this lab is the following:

- Implementation of the **rotation methods** and the **insertion method** of the Red-Black Tree of card hands using the provided Cards and Hands classes.
- Implementation of the specialized method in the Red-Black Tree: **Remove all hands containing a certain Card**.
- Integration of the completed and tested Red-Black Tree and the Max Heap from Lab 2 into the model code to deliver the final 5-card competition game with aggressive AI logic.
- Passing all the test cases in the autograder.
- Passing the additional tests designed by the TA.
- Completion of the lab demo of your integrated model code.

Coding Objectives

Use the provided test bench **in the same class**, complete all the missing implementations of HandsRBT.java.

The Red-Black Tree of Card Hands uses the level of hands as the sorting key, and the Hands class contains the required level-comparison methods to support the sorting.

Model Code Integration Objectives

Once you have fully implemented HandsRBT.java (i.e. passing all the test cases), you need to complete the model code integration with the following specifications:

- Upon launching the model code, the program draws 25 cards for AI, and another 25 cards for Player from the card pool and sorts them in rank + suite order in the initialization stage.
- AI immediately generates all the possible 5-card hands and inserts them into the Max Heap of Hands using the same logic from Lab 2.
- Player generates all the possible 5-card hands and inserts them into the Red-Black Tree of Hands.
- The program loop runs for 5 rounds, and AI and Player each form their own 5-card hand move per round for competition. In each round:
    - AI always goes for the most aggressive hand, as in Lab 2.
    - Player allows user to choose a 5-card hand from the pocket card. However, the user must go for a valid 5-card hand unless running out of possible choices (Red-Black Tree empty).
    - The program determines the winner of the round by comparing the user-chosen hand against the AI hand, and increments the score of the respective party.
- The program displays the winner of the game at the end of the program.
- The program must display clearly the pocket cards and every individual 5-card hand move of both AI and Player in every round.

Marking Scheme

- **[Total 60 Marks] Autograding Section – RBT Code Implementation and Behaviour**
  - **[5 Marks]** Implement the private method for Left Rotation – 3 marks
    Passing testRotateLeft() test case – 2 marks
  - **[5 Marks]** Implement the private method for Right Rotation – 3 marks
    Passing testRotateRight() test case – 2 marks
  - **[3 Marks]** Implement the public method insert(), not including fixRedViolation
  - **[2 Marks]** Passing testInsertCase1NullRoot() test case – 1 mark
    Passing testInsertCase1BlackParent() test case – 1 mark
  - **[2 Marks]** Passing testInsertCase2A() test case – 1 mark
    Passing testInsertCase2B() test case – 1 mark
  - **[6 Marks]** Implement the Case Red Uncle (i.e., Sibling of the Parent is RED) in the private method fixRedViolation() – 3 Marks
    Passing testInsertCase3() – 3 marks
  - **[10 Marks]** Implement the Case Black Uncle, Inner Grandchild in the private method fixRedViolation() – 4 marks
    Passing testInsertCase4L() – 2 marks
    Passing testInsertCase4R() – 2 marks
    Designing and passing testCustomInsertCase4 () – 2 marks
  - **[10 Marks]** Implement the Case Black Uncle, Outergrandchild in the private method fixRedViolation() – 4 marks
    Passing testInsertCase5L() – 2 marks
    Passing testInsertCase5R() – 2 marks
    Designing and passing testCustomInsertCase5 () – 2 marks
  - **[10 Marks]** Implement the private method deleteHandsWithCard() – 6 marks
    Passing testDeleteHandsWithCard() – 2 marks
    Designing and passing testCustomDeleteHandsWithCard () – 2 marks
    [7 Marks] Passing the additional test cases designed by TA

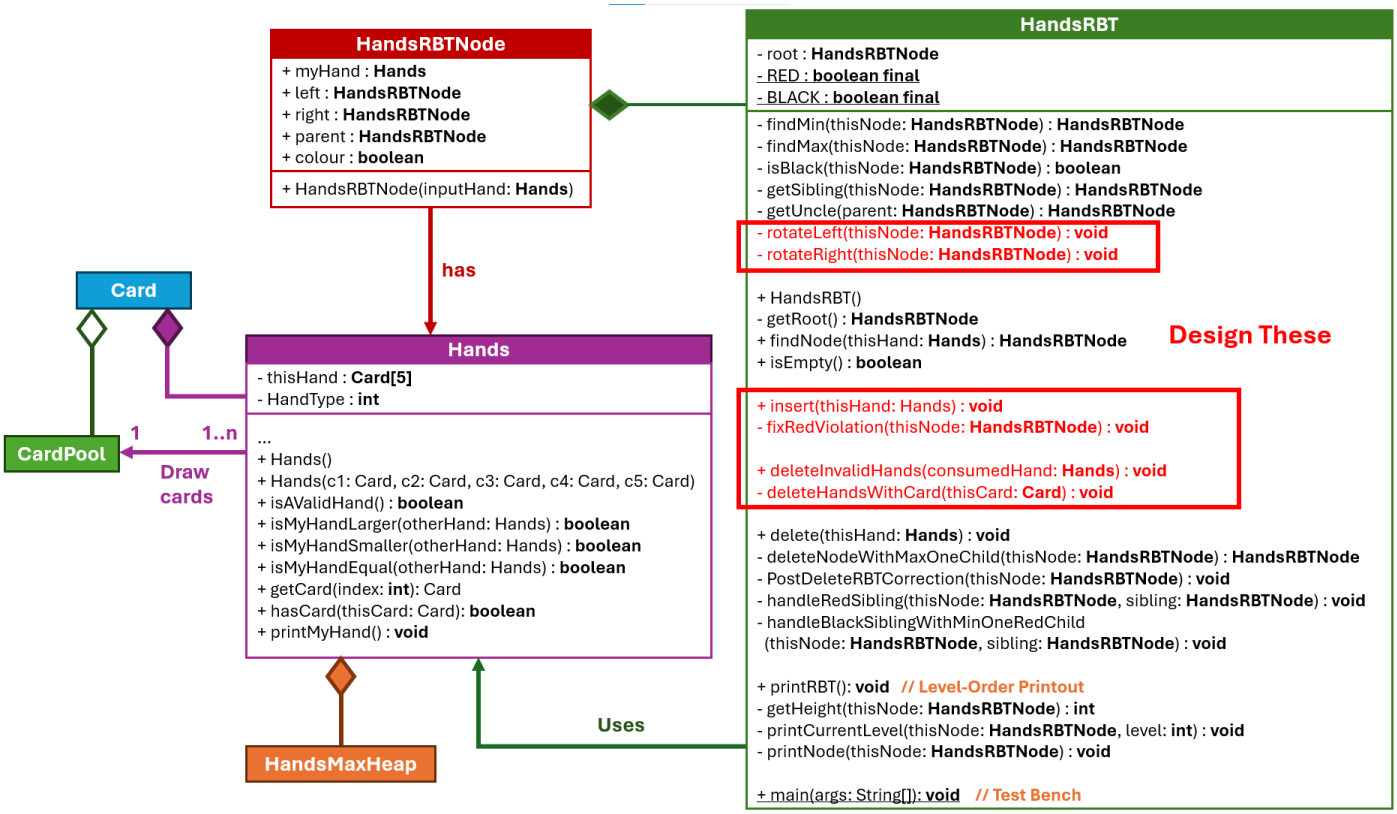
  - **Note:** If test case failed, the test case marks are deducted, and only partial marks will be awarded to the implementation marks based on the severity of the logic error.

- **[Total 40 Marks] Lab Demo Section – Demonstration of Model Code_CardGame**
  - **[10 Marks]** Demonstration of the updated generateHands() series of methods – 4 marks
  - **[8 Marks]** Demonstration of AI and Player each drawing and sorting 25 cards from the card pool with printed message – 4 marks
    Demonstration of AI and Player sorting their valid hands into the respective data structures (Max Heap or BST for AI, Red-Black Tree for Player) – 4 marks
  - **[4 Marks]** Demonstration of AI exhibiting the identical Aggressive AI logic from Lab 2
  - **[10 Marks]** Demonstration of the use of Red-Black Tree methods to determine the validity of the Player move – 3 marks
    Demonstration of removal of all Hands nodes from RBT after consuming the cards for the selected hand – 7 marks
  - **[3 Marks]** Demonstration of determining the winner/loser for each round.
  - **[5 Marks]** Program ends with exactly 5 hands (25-card pocket, 5-card hand per play)
    Program prints the end result of the game, with winner / loser messages.

System Overview

During Lab 2, the other team has been assigned to help you develop the Red-Black Tree implementation for the player-side of the game logic. However, as we have learned in class, RBT is not a straightforward data structure, particularly when the large number of algorithmic variations for the colour correction is concerned. The assisting team has worked relentlessly to complete half of the RBT implementation, so that when you have the bandwidth to work on the RBT class, you are left to complete the remaining three features: 1) Rotation Algorithms, 2) Insertion Algorithms with Colour Correction Logics, and 3) Method to Remove All Nodes Containing Hands with a Given Card. With these features implemented, you are ready to implement the game model and hopefully hand it off to the front-end team for productization after some tuning.

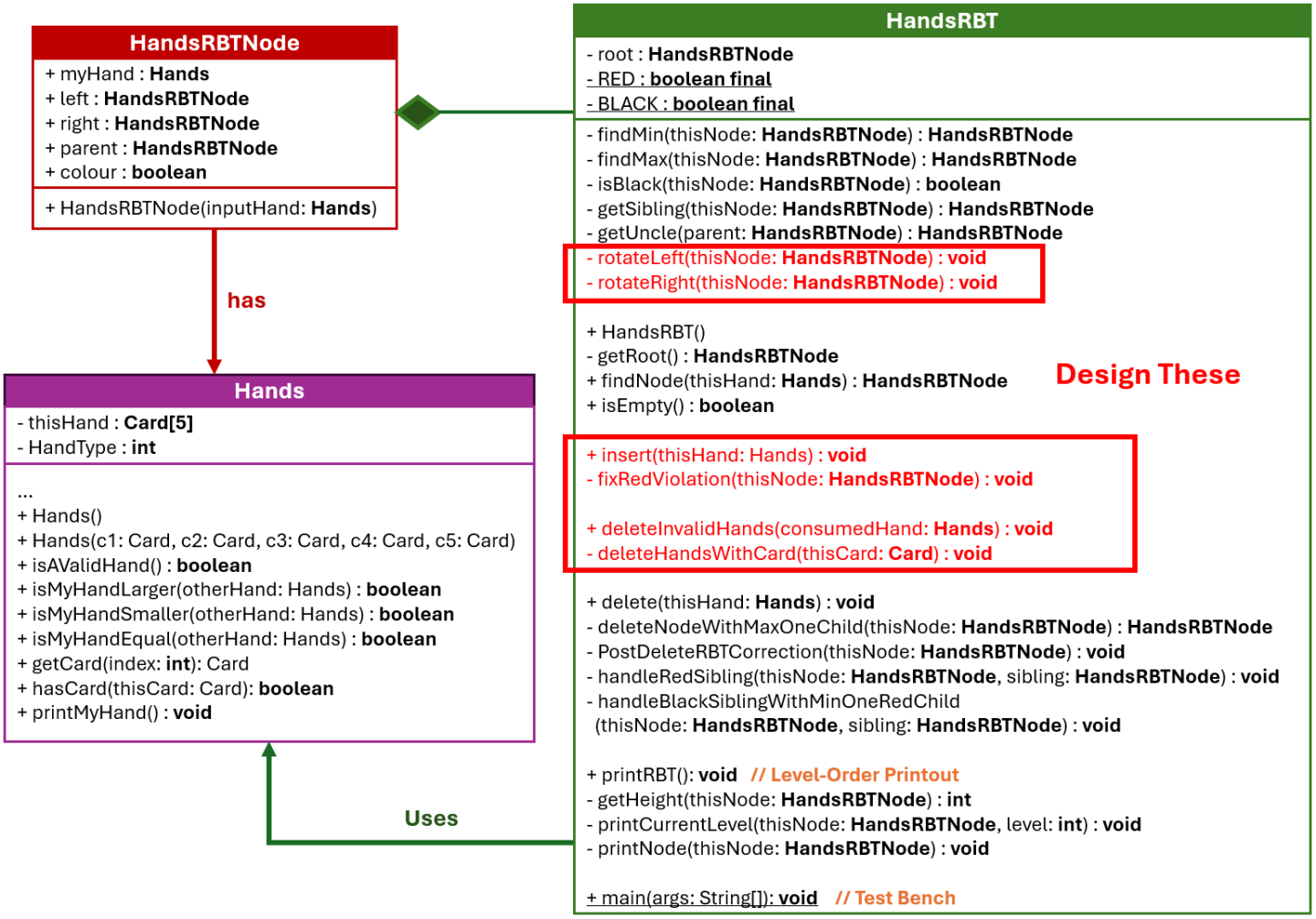
Below is the updated UML Diagram (with HandsMaxHeap simplified to focus on the design of Red-Black Tree of Card Hands.) The features requiring your input are highlighted in red.



This design approach is very similar to that of HandsMaxHeap, and is intended to limit the design dependencies of HandsRBT to only the Hands class, thus simplifying the development tasks. Your game plan is to first implement the remaining features of Red-Black Tree of Hands using the predesigned test bench, then complete the final game integration.

Red-Black Tree of Hands – Design Specifications

This section provides the design specifications of the missing members of the HandsRBT class, as well as the useful methods from the HandsRBT class and Hands class.



All methods that require your implementation will be tagged **[TO-DO]** in the skeleton code.

I. HandsRBT Members

- **private HandsRBTNode** root
  - The root of the Red-Black binary Tree.

- **private static final boolean** RED / **private static final boolean** BLACK
  - RED = false, BLACK = true
  - For colouring the nodes

- **private HandsRBTNode** findMin(**HandsRBTNode** thisNode)
- **private HandsRBTNode** findMax(**HandsRBTNode** thisNode)
  - Returning the Node with the smallest / largest hand from the RBT.

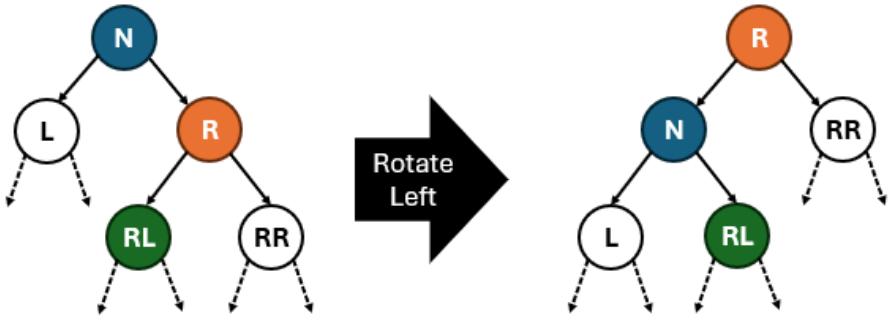
- **private boolean** isBlack(**HandsRBTNode** thisNode)

- Return TRUE if BLACK, and FALSE otherwise

- **private HandsRBTNode** getSibling(**HandsRBTNode** thisNode)

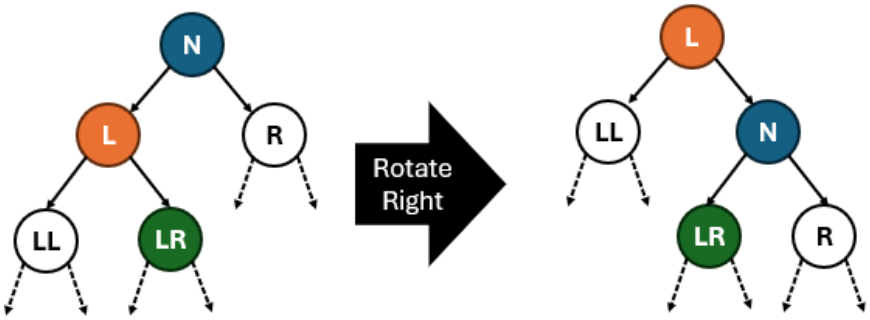
- **private HandsRBTNode** getUncle(**HandsRBTNode** parent)
  - Returning the reference of the **uncle node of the current node**, in other words, the **sibling node of the parent. Recall that the uncle could be a null pointer.**

- **[TO-DO] private void** rotateLeft(**HandsRBTNode** thisNode)
  - Complete the leftward rotate around thisNode (node **N** in the figure), in other words, the rotation between **N** and **R. You may assume that R is not null.**



  - Refer to lecture notes for details.

- **[TO-DO] private void** rotateRight(**HandsRBTNode** thisNode)
  - Complete the rightward rotate around thisNode (node **N** in the figure), in other words, the rotation between **N** and **L. You may assume that L is not null.**



  - Refer to lecture notes for details.

- **public** HandsRBT()
  - Constructor. Set root to null.

- **private HandsRBTNode** getRoot()
  - Return the reference to the root node of the RBT.

- **public Boolean** isEmpty()
  - Return true if RBT is empty

- **public HandsRBTNode** findNode(**Hands** thisHand)
  - Return the reference to the node containing the Hand matching thisHand; return null if thisHand is not in the RBT

- **[TO-DO] public void** insert(**Hands** thisHand)

- Insert a node containing thisHand into the RBT. If thisHand is already in the RBT, do nothing. Else, if the RBT is empty, insert the new node as the root and colour it BLACK. Else, insert the new node using the typical binary search tree algorithm and colour it RED. If a RED violation occurs, call the private method fixRedViolation() method to fix it.
- The implementation of the insertion in a binary search tree is provided in the HandsBST class. You may copy it and make the appropriate changes (the types of some variables need to be changed)

- **[TO-DO]** **private void** fixRedViolation(**HandsRBTNode** thisNode)
  - This method is used when thisNode is RED and its parent is RED
  - It corrects the red violation (after the method ends execution, the RBT properties are restored)
  - The method may be recursively invoked to correct the entire path from thisNode up to the root. Alternatively, you may correct the entire path non-recursively. It is your choice.

- **[TO-DO]** **public void** deleteInvalidHands(**Hands** consumedHand)
  - Remove all the Hands from RBT that contain any one of the 5 cards in the consumedHand. For this, invoke the private method deleteHandsWithCard() for all 5 cards in the consumedHand.

- **[TO-DO]** **private void** deleteHandsWithCard(**Card** thisCard)
  - Traverse the entire RBT, and register all the nodes with hands containing thisCard in a List.
  - Then, iterate through the List and delete every single registered node from RBT using the existing delete() method.
  - For the List, you may use the built-in generic java class ArrayList<E> in package java.util or any other built-in class that implements a list (e.g., LinkedList<E>, Vector<E>). See the Java API specification.

- **public void** delete(**Hands** thisHand)
  - Delete the node containing thisHand from the RBT.
  - Then, invoke PostDeleteRBTCorrection() method to balance the RBT.
- **private HandsRBTNode** deleteNodeWithMaxOneChild(**HandsRBTNode** thisNode)
  - A helper method handling the special deletion case.
- **private void** handleRedSibling(**HandsRBTNode** thisNode, **HandsRBTNode** sibling)
  - A helper method handling the red sibling special case during colour correction.
- **private void** handleBlackSiblingWithMinOneRedChild(**HandsRBTNode** thisNode, **HandsRBTNode** sibling)
  - A helper method handling the black sibling special case (with at least one red child) during colour correction.

- **public void** printRBT()
  - Print the RBT in level-order, so to display the RBT in tree-like printout.
- **private int** getHeight(**HandsRBTNode** thisNode)
  - A helper method getting the height of the RBT for level-order traversal.
- **private void** printCurrentLevel(**HandsRBTNode** thisNode, **int** level)
  - A helper method printing the nodes at the given level.
- **private void** printNode(**HandsRBTNode** thisNode)
  - A helper method printing the Hands in thisNode.

You may implement additional private methods as needed.

II. HandsRBTNode Members

The node class of the Red-Black Tree. Similar to the node class for a Binary Search Tree.

- **public Hands** myHand
    - The hand stored in this node.
- **public HandsRBTNode** left
- **public HandsRBTNode** right
- **public HandsRBTNode** parent
- **public boolean** colour
    - RED = false, BLACK = true.

- **public** HandsRBTNode(**Hands** inputHand)
    - Create the node with the inputHand as the value for myHand.

III. Private NilNode Class extends from HandsRBTNode

This is a null-node placeholder class with BLACK colour. Only used in certain post-deletion correction cases. We do not need to care about this class for our tasks.

IV. Useful Methods from Hand Class

While you may find other methods in Hands class also applicable to your design, these are the methods that will certainly be helpful. Use them wherever you see fit.

- Hands()
    - Default constructor of Hands class instantiates an INVALID hand.
    - Can be used to fill in dummy nodes. (think about when!!)

- **public boolean** isMyHandLarger(Hands otherHand)
- **public boolean** isMyHandSmaller(Hands otherHand)
- **public boolean** isMyHandEqual(Hands otherHand)
    - Comparison of Hands level
    - Use them to support all the Red-Black Tree data organization operations

- **public void** printMyHand()
    - Print all five cards in the Hand, sorted, and with Hand Type
    - Royal Flush (RF), Straight Flush (SF), Four-of-a-Kind (FK), Full House (FH), Straight (S), and Invalid Hand (INV)

- **public boolean** isAValidHand()
    - Check if the 5 cards in this Hand makes it a valid 5-card hand.

Integration Plan

Once the HandsRBT class is implemented and thoroughly tested (passing the autograder), here are the additional steps you need to do to implement the game loop logic.

I. Sort Pocket Cards and Generate All Valid Hands for both AI and Player

First, populate the implementation for generateHandsIntoHeap() and generateHandsIntoRBT() using the existing algorithms of generateHands() from Lab 2. Heap version is for the AI logic, and Tree version is for the Player logic using RBT. Minor logic changes may be required.

If you have not completed Lab 2, you need to complete the permutation logic to generate all valid hands using the pocket cards, and insert them into the relevant data structure. If you are behind in Lab 2, you can use the provided Binary Search Tree (BST) of Hands as a replacement to score the demo marks. However, you still need to complete the RBT implementation to score the full marks for this lab.

II. Develop the Turn-Based Game Logic

The main method in the model code already contains some boilerplate code:

- A pool of 52 cards is created first in myCardPool.
- Variables, arrays, and other data structures related to AI and Player logics. The skeleton code provides the basic BST of hands for students running behind in lab progress. If you have successfully completed Lab 2, you should replace the provided HandsBST with your HandsMaxHeap to improve performance.
- A method enabling users to type the serial number to choose 5 cards to form a hand.
    - **public static Hands** getUserHand(**Card[]** myCards)
        - myCards are the reference to the array of pocket cards.
        - Call this method whenever the program needs it
    - **Before calling this method, you should print all Player's pocket cards with serial number**.
- An Input Scanner object to enable synchronous keyboard input.

You are required to implement the main game logic between lines 22 and 84. ***Very detailed comments*** are provided in the skeleton code to help you complete the implementation. You are recommended to upgrade your main game logic from Lab 2, but it is not a must-have. Here are the same details steps:

- **General Game Rules and Mechanisms**
    - Each game consists of 5 rounds. AI and Player get 25 cards each at the beginning of the game, and both parties make a 5-card hand move in each round to compete.
    - In each round, the following actions take place:
        - AI pocket cards are printed for player analysis.
        - Player pocket cards are printed with serial number.
        - Player makes a 5-card hand choice, and only gets to proceed if the hand choice is valid.
        - AI then makes the most aggressive move.
        - The game compares AI's hand vs. Player's hand, and determines the winner.
        - The score of the winning party gets incremented by 1 at the end of the round.
        - AI and Player are not allowed to make an invalid 5-card hand move unless they are completely out of valid hands.
    - At the end of the game, the winner and the winning score are reported.

- **Step 1 – Initialization**
    - Given the CardPool instance, get 25 cards (defined by POCKETSIZE) for both AI and Player.
    - Sort their cards using sortCards().

- Instantiate a HandsRBT for Player, and invoke generateHandsIntoRBT() to populate the Player RBT with all possible hands from the pocket cards.
- Instantiate a HandsBST for AI, and invoke generateHandsIntoBST() to populate the AI BST with all possible hands from the pocket cards. Recall that the AI plays the strongest hand all the time. Use the getMaxHand() or removeMaxHand()from the HandsBST class to obtain this hand.
- If you have successfully completed Lab 2, you can replace the provided HandsBST with your own HandsMaxHeap to improve the program performance.

- **Step 2 – Deploy Game Loop Logic**
  - Given POCKETSIZE = 25, and a 5-card hand being consumed at each round from each party, the program loop should repeat 5 times. You can optionally parameterize the iteration count for scalability.

  - 2-1 Print both AI and Player pocket cards for Strategy Analysis
    - When printing the Player pocket cards, you MUST print with serial number to allow player to choose cards to form hands. The serial number is the index of the card in the current pocket, indexing starting with 1.
    - Also check if RBT is empty. If yes, notify the player that no more valid moves are available.

  - 2-2 Use the provided getUserHand() method to allow player to pick the 5-card hand from the pocket card.
    - After the hand is chosen, the program must check if the hand is valid by checking if it is in the HandsRBT. If the hand is not in the HandsRBT and the HandsRBT is not empty, notify the Player that there are still valid 5-card hands to make a move, and the Player cannot form an invalid hand to as a “pass” move.

  - 2-3 Save the chosen player hand as playerHand, and update both the pocket cards and the RBT.
    - With the consumed Hand, remember to delete all the hands that are no longer valid from the RBT using the deleteInvalidHands() method.
    - Remove the consumed 5 cards from the pocket card, and reduce the pocket size by 5.

  - 2-4 Construct the Aggressive AI Logic from Lab 2
    - If you have completed Lab 2, you should use HandsMaxHeap. Otherwise, you can use the provided HandsBST (and apply the learned knowledge from COE2SI3)
    - For every 5-card move made, remove the consumed 5 cards from AI pocket cards, then regenerate the MaxHeap/Hands BST (inefficient) or remove invalid hands from the MaxHeap/HandsBST (efficient, optional)
    - Save the chosen move as the aiHand.
    - Remember, once out of valid hands, AI should pick any 5 cards in the pocket to form a “pass” move.

  - 2-5 Determine the Win/Lose result for the round, and increment score for the winning party.
    - Print both playerHand and aiHand for visual confirmation
    - Compare hands, and increment the score for the respective winning party.
    - In an unlikely Draw (no winner) situation, no score increment will take place.

- Step 3 – Report the Game Results