COE 3SM4 Lab 2 – Max Heap of Card Hands [110 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 HandsMaxHeap.txt and ModelCode_CardGame.txt to *Avenue Dropbox***

Lab Deadline

The deadline for Lab 2 is Saturday, February 7, 2026, at 11:59 pm.

This lab was designed by Prof. Scott Chen.

Deliverables

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

- HandsMaxHeap.txt and ModelCode_CardGame.txt (i.e. change .java extension to .txt)
- You further will demonstrate your Model Code and the correct aggressive AI behaviour.

Premise

Lab 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 2 focuses on the implementation of Max Heap of card hands using an array-based binary max heap, and the integration of Max Heap with the provided model code to realize the required *Aggressive AI* behaviours.

The scope of this lab is the following:

- Implementation of the basic Max Heap of card hands using the provided Card and Hands classes.
- Integration of the completed and tested Max Heap into the model code to deliver the aggressive AI behaviour.
- Passing all the tests in the Autograder.
- Completion of the lab demo of your integrated model code.

Note that the TA will run additional tests with your code and will double check the code to make sure that it satisfies all the requirements.

Coding Objectives

Use the provided test bench **in the same class**, complete all the implementation of HandsMaxHeap.java.

The Max Heap 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.

You further have to implement the public static method heapSort() to sort an array of card hands in **Descending Order**.

Review Lecture Notes on the Heap implementation using Arrays.

Model Code Integration Objectives

Once you have fully implemented HandsMaxHeap.java (and passed all the test cases), you need to complete the model code integration with the following specifications:

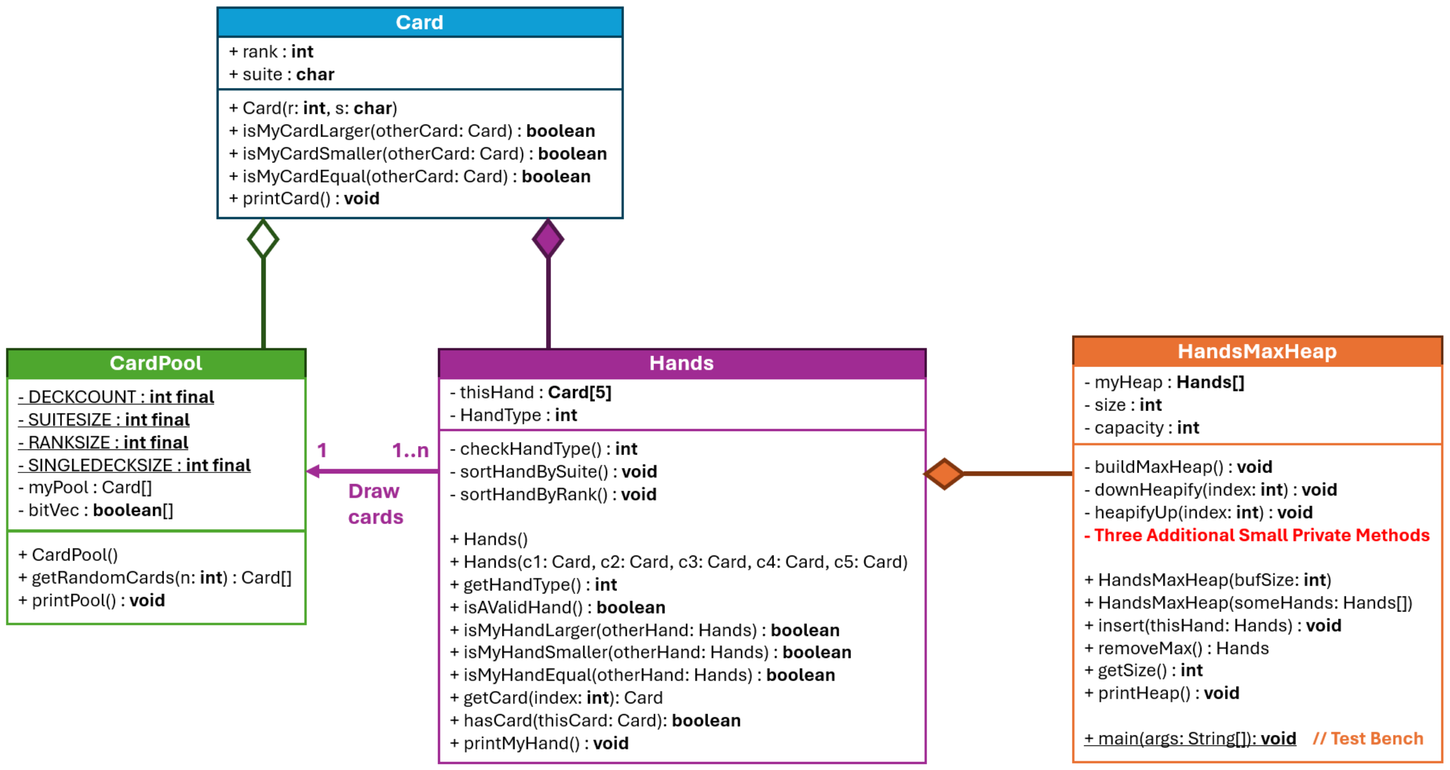
- Upon launching the model code, the program draws 25 cards from the card pool and sorts them in rank + suite order in the initialization stage. Next, it generates all the possible 5-card valid hands and stores them into the Max Heap of Hands.
- Once into the program loop, the program should always play the highest possible 5-card hand from the 25 pocket cards, then remove the 5 played cards from the pocket and update the Max Heap by removing all the hands in the Max Heap that contain any of the 5 consumed cards.
- The program must continue playing until all cards in the pocket are used. If the pocket is not empty, but it is impossible to construct any valid hand, the program should pick any 5 cards from the pocket as a "Pass Move".
- The program must display all hands played by the AI in chronological order to show its “aggressiveness” for logic validation.
- The program stops when all cards have been used.

Marking Scheme

- **[Total 65 Marks] Autograding Section – Max Heap Code Implementation and Behaviour**
  - **[2 Marks]** Implement private method computing parent index – 1 mark
    Passing testParent() test case – 1 mark
  - **[2 Marks]** Implement private method computing left child index – 1 mark
    Passing testLeftChild() test case – 1 mark
  - **[2 Marks]** Implement private method computing right child index – 1 mark
    Passing testRightChild() test case – 1 mark
  - **[4 Marks]** Implement HandsMaxHeap() constructor1 – 2 marks
    Passing testHandsMaxHeap1() test case – 1 mark
    Passing testHandsMaxHeap2() test case – 1 mark
  - **[16 Marks]** Implement heapifyUp() to support insert() – 4 marks
    Implement insert() – 2 marks
    Passing testInsert1() – 2 marks
    Passing testInsert2() – 2 marks
    Designing and Passing CustomTestInsert1() – 3 marks
    Designing and Passing CustomTestInsert2() – 3 marks
  - **[17 Marks]** Implement downHeapify() to support remove() – 4 marks
    Implement removeMax() – 3 marks
    Passing testRemoveMax1() – 2 marks
    Passing testRemoveMax 2() – 2 marks
    Designing and Passing CustomTestRemoveMax1() – 3 marks
    Designing and Passing CustomTestRemoveMax2() – 3 marks
  - **[3 Marks]** Implement getSize()– 1 mark
    Passing testGetSize1() test case – 1 mark
    Passing testGetSize2() test case – 1 mark
  - **[8 Marks]** Implement buildMaxHeap() – 4 marks
    Passing 2 test cases run by the TA - 4 marks
  - **[9 Marks]** Implement heapSort() - 5 marks
    Passing testHeapSort1() test case – 2 mark
    Passing testHeapSort2() test case – 2 mark
  - **[2 Marks]** Instructive comments in the code
  - **Note 1:** 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.
  - **Note 2:** Deductions will be applied if the sorting in HeapSort is not IN PLACE.
  -
- **[Total 45 Marks] Lab Demo Section – Aggressive AI Logic Implementation using Hands Max Heap**
  - **[13 Marks]** Demonstration of current generateHands() behaviour in the Model Code – 10 marks
    Demonstration of printHeap() feature in HandsMaxHeap class – 3 marks
  - **[10 Marks]** Demonstration of Aggressive Card Selection in the Model Code with printed Hand
  - **[10 Marks]** Demonstration of Cards Removal after used – 4 marks
    Demonstration of Updated Max Heap after card removal – 6 mark
  - **[2 Marks]** Program ends with exactly 5 hands (25-card pocket, 5-card hand per play)
  - **[10 Marks]** **Discussion:** running time and memory required by your implementation; how does your solution compare with other possibilities in terms of resource utilization? Could you improve the efficiency if you were given more time to work on this lab? How?

System Overview

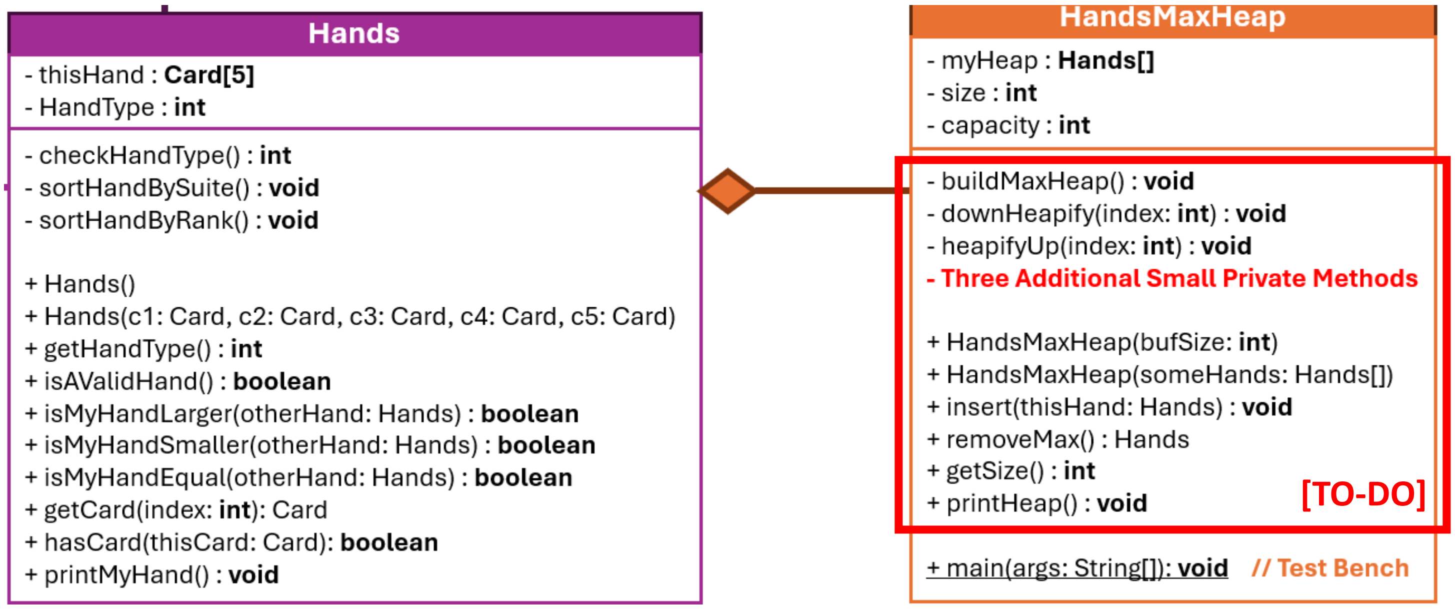
You are assigned with the task to implement the *Aggressive AI* logic, where the game logic should go for the Strongest Hand for every 5-card move. Being well-versed in data structures, you immediately see the high degree of resemblance between this AI logic requirement and the priority queue, which is typically constructed using a Max Heap. As a result, after reviewing the new features added to the Card and the Hand classes, you propose the following UML diagram for the AI-side of the game.


Card
+ rank : int
+ suite : char
+ Card(r: int, s: char)
+ isMyCardLarger(otherCard: Card) : boolean
+ isMyCardSmaller(otherCard: Card) : boolean
+ isMyCardEqual(otherCard: Card) : boolean
+ printCard() : void
CardPool
- DECKCOUNT : int final
- SUITESIZE : int final
- RANKSIZE : int final
- SINGLEDECKSIZE : int final
- myPool : Card[]
- bitVec : boolean[]
+ CardPool()
+ getRandomCards(n: int) : Card[]
+ printPool() : void
1
1..n
Draw cards
Hands
- thisHand : Card[5]
- HandType : int
- checkHandType() : int
- sortHandBySuite() : void
- sortHandByRank() : void
+ Hands()
+ Hands(c1: Card, c2: Card, c3: Card, c4: Card, c5: Card)
+ getHandType() : int
+ isAValidHand() : boolean
+ isMyHandLarger(otherHand: Hands) : boolean
+ isMyHandSmaller(otherHand: Hands) : boolean
+ isMyHandEqual(otherHand: Hands) : boolean
+ getCard(index: int): Card
+ hasCard(thisCard: Card): boolean
+ printMyHand() : void
HandsMaxHeap
- myHeap : Hands[]
- size : int
- capacity : int
- buildMaxHeap() : void
- downHeapify(index: int) : void
- heapifyUp(index: int) : void
- Three Additional Small Private Methods
+ HandsMaxHeap(bufSize: int)
+ HandsMaxHeap(someHands: Hands[])
+ insert(thisHand: Hands) : void
+ removeMax() : Hands
+ getSize() : int
+ printHeap() : void
+ main(args: String[]): void
// Test Bench

This design approach ensures that the Max Heap implementation has design dependencies on only the Hands class, so to minimize the module coupling and simplify the development tasks. Your game plan is to first implement the basic Max Heap of Hands and then test it using the predesigned test bench, then complete the game integration to achieve the Aggressive AI design requirements.

Max Heap of Hands – Design Specifications

This section provides the design specifications of each member of the HandsMaxHeap class, as well as the useful methods from the Hands class.


Hands
- thisHand : Card[5]
- HandType : int
- checkHandType() : int
- sortHandBySuite() : void
- sortHandByRank() : void
+ Hands()
+ Hands(c1: Card, c2: Card, c3: Card, c4: Card, c5: Card)
+ getHandType() : int
+ isAValidHand() : boolean
+ isMyHandLarger(otherHand: Hands) : boolean
+ isMyHandSmaller(otherHand: Hands) : boolean
+ isMyHandEqual(otherHand: Hands) : boolean
+ getCard(index: int): Card
+ hasCard(thisCard: Card): boolean
+ printMyHand() : void
HandsMaxHeap
- myHeap : Hands[]
- size : int
- capacity : int
- buildMaxHeap() : void
- downHeapify(index: int) : void
- heapifyUp(index: int) : void
- Three Additional Small Private Methods
+ HandsMaxHeap(bufSize: int)
+ HandsMaxHeap(someHands: Hands[])
+ insert(thisHand: Hands) : void
+ removeMax() : Hands
+ getSize() : int
+ printHeap() : void
[TO-DO]
+ main(args: String[]): void
// Test Bench

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

I. HandsMaxHeap Members

- **private** Hands**[]** myHeap
  - An array of Hands where the heap items are to be stored
  - Design Philosophy – "HandsMaxHeap is the Composition of Hands".

- **private int** size
  - The number of valid items (i.e., Hands) in the Max Heap.

- **private int** capacity
  - The largest number of items (i.e., Hands) the heap can store

- **public** HandsMaxHeap(Hands[] someHands)
  - Constructor, constructs a heap out of the array someHands
  - the first element in the array is treated as a dummy, the remaining elements are organized as a heap using the private method bulidMaxHeap, which you have to implement

- [TO-DO] **private void** buildMaxHeap()
  - When this method is invoked by the constructor, the array myHeap is not organized as a heap yet;
  - the method organizes the array as a heap (disregarding the element at index 0) using the O(n)-time algorithm
  - **This method is not autograded separtely, but it is implicitly tested via the tests for heapSort**

- [TO-DO] HandsMaxHeap(**int** bufSize)
  - Constructor, instantiates myHeap with the capacity specified by bufSize.

    - Pay attention to the initial value of size.
    - Pay attention to the initial contents of the first element of the myHeap array.

- [TO-DO] **private void** downHeapify(**int** index)
    - Percolates down the Hand at index, until it fits.

- [TO-DO] **private void** heapifyUp(**int** index)
    - Percolates up the Hand at index, until it fits.

- [TO-DO] Three additional private methods to help with the Heap operations (if you find it necessary)
    - A private method that **calculates the <u>parent</u> index from the given index**
    - A private method that **calculates the <u>left child</u> index from the given index**
    - A private method that **calculates the <u>right child</u> index from the given index**

- [TO-DO] **public void** insert(Hands thisHand)
    - Inserts thisHand into the Max Heap
    - Invokes heapifyUp()

- [TO-DO] **public** Hands removeMax()
    - Removes the largest hand from the Max Heap and returns it
    - Invokes downHeapify()

- [TO-DO] **public int** getSize()
    - Returns the size of the Max Heap

- [TO-DO] **public void** printHeap()
    - **This method is not autograded, but you will need it to demonstrate it in the in-person demo as part of the game logic.**
    - Iterates through the Max Heap in Level-Order and prints the Hand at every visited node using the existing printHand() method from the Hand class.

- [TO-DO] **public static void** heapSort(**Hands[]** myHands, **int** size)
    - Implements the heapsort algorithm to sort IN PLACE the array myHands in Descending Order.

II. 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 levels
    - **<u>IMPORTANT: Custom 5-Card Hands Comparison Rule (Specific to Our Lab)</u>**
        - **Royal Flush > Straight Flush > Four-of-a-Kind > Full House > Straight**
        - **Special Rules for Comparison of hands with the same type**

    - **Royal Flush** and **Straight Flush**: the hand with the larger leading card is larger
      NOTE: card1 is larger than card2 if card1 has larger rank or they have the same rank and card1 has larger suit (Club < Diamond < Heart < Spade)
    - **Straight:** the hand with the larger leading card is larger; if the leading cards are equal in both rank and suit the second leading cards are compared and so on.
    - **Four-of-a-Kind**: the hand with the quadruple of larger rank is larger; if the quadruples are identical, the hand with the largest singlet is larger
    - **Full House**: the hand with the largest rank in the triplet is larger; if the triplets have the same rank the triplet with the highest suit card is considered larger. Only if the triplets are completely identical in both the rank and suits, the pairs are compared in a similar manner.
    - **Ex.** Hand 1: {3H, 3S, 3C, 6H, 6D}   Hand 2: {3H, 3D, 3C, 8H, 8S}
      Hand 1 is considered larger than Hand 2 because the highest suit card in the triplet is Spade of 3 in Hand 1, whereas that of Hand 2 is Heart of 3. Given Spade > Heart, Hand 1 is deemed larger despite its pair being lower in rank than Hand 2.
    - **Note that some of these special rules are not commonly used in other card games.**

- **public void** printMyHand()
  - Prints 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()
  - Checks if the 5 cards in this Hand make it a valid 5-card hand.

Integration Plan

Once the HandsMaxHeap class is implemented and thoroughly tested (passing the autograder locally and on Git), here are the additional steps you need to do to demonstrate your Aggressive AI features in the main game, which should recreate or enhance beyond the sample executable demonstrated in the briefing video.

I. Find all possible valid hands using the incoming pocket cards

First step of integration is to make sure that we can generate all possible valid hands from the pocket cards, and populate them into the max heap. This guarantees that the removeMax operation always removes the strongest hand. Thus, the AI can simply use removeMax to stay aggressive.

- **public static void** generateHands(Card[] thisPocket)
    - Find all possible valid hands in thisPocket card array and store them into the Max Heap
    - If the pocketSize is less than or equal to 5, no valid hand can be generated (the heap will be empty).

II. Develop the Aggressive AI card player prototype

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

- A pool of 52 cards is created first in myCardPool.
- 25 cards are randomly drawn from myCardPool into the pocket card deck myCards. They are sorted in ascending order and then printed.
- generateHands() is invoked to generate the valid hands from the initial pocket of 25 cards. Next printHeap() is invoked to print the contents of the initial heap.
- A program loop that will continue until the pocket size is less than 5 cards, which indicates that there are no more 5-card hands available.

You are required to implement the Aggressive AI logic from line 64 onwards. Comments are provided in the skeleton code to help you complete the implementation. Here are the details steps:

- **Step 1 – Make the Most Aggressive Move**
    - Check if the Max Heap contains any valid hand.
        - If yes, remove the largest hand from the Max Heap the most aggressive move for this round.
        - Otherwise, the AI is out of valid hands. Just pick any 5 cards from the pocket cards as an "Invalid Passing Hand".
    - **You must print this chosen hand to validate your Aggressive AI logic.**
- **Step 2 – Remove the Used Cards and Update the Max Heap**
    - Remove all the cards used in the move from the pocket cards and update the Max Heap.
    - Print the remaining cards and the contents of the heap
    - Note that the heap can only be manipulated using the public methods specified in the class HandMaxHeap. You are not allowed to add any other public methods.
    - **You will be asked in the lab demo to discuss the running time and the memory needs of your implementation. Is your choice the most efficient one time-wise, memory-wise? Could you make it more efficient if you had more time to work on the assignment?**