Homework # 5
due Monday, October 8, 10:00 PM

In this assignment we implement “endogenous” “doubly-linked” lists. See the “Linked List Variations” handout to see what this means. You will be implementing an abstract data type for lists of card objects used in simulated card games. You will also complete a simple strategy for a variation of the child’s game of “War.”

1 Concerning the ADT Card

Each Card instance represents a different physical card. Each card has a suit and rank that cannot change and previous, next and group pointers for hooking a card into group. The class defines getters for each field, but no setters: the first two fields are final, and the last three are used by the Card.Group ADT (see below). The Card ADT is already implemented for you, but you should read the code to understand the implementation.

2 Concerning the enumerated types Suit and Rank

The Card class includes two enumerated types: Suit and Rank. An enumerated type is one in which the values are “enumerated” in advance, listed in the definition. In our case, we have the following definition of Suit:

    public enum Suit { CLUB, DIAMOND, HEART, SPADE };

This definition says there are exactly four values of the type Suit and they are (respectively) CLUB, DIAMOND, HEART and SPADE. Values of enumerated types have a number of predefined methods:

    compareTo(E) Compare this value to another value of the same enumerated type.
    ordinal() Return the index of this value (0-based).
    toString() Return the name of this value.

The enumerated type itself has a number of (static) methods:

    values() Return an array of all values of this type.
    valueOf(String) Return the value that has the given name (or throw an exception).

For this Homework, you will be given code that uses enumerated types and their methods, but you will not need to write any such code.

2.1 Sorting Cards

What order should cards be sorted in? It depends on the game. In some games, aces are high (as in bridge), and in others (rummy), they are low. In Sheepshead (a German card game popular in Milwaukee) the 10 ranks higher than the king, and all queens and jacks (regardless of suit) are ranked above the ace of trumps. So it doesn’t make sense to have a single way to compare cards.

The Java library defines a generic interface Comparator to represent a way of comparing two things. The interface is as follows:

    public interface Comparator<T> {
      /**
       * Compares its two arguments for order. Returns a negative integer,
       * zero, or a positive integer as the first argument is less than, equal
       * to, or greater than the second.
       */
    }

    public interface Card { /* Card methods */ }
3 Concerning the ADT Card.Group

This ADT models a pile of cards, or a hand. Unusually, this ADT is implemented by a class (named “Group”) nested in the Card class. Each group keeps track of the first and last card in the group and requires that the cards in the group are linked together with the previous and next pointers. If the group is empty, both the first and last pointer are null. The first card’s (if any) previous pointer will always be null as will the last card’s (if any) next pointer. All the cards in the group should have the group pointer set to this group. When a card is not in a group, all of its three pointers should be null.

This data structure is highly redundant, and thus it has a non-trivial invariant. The cards must be linked up without any cycles in which each card points to the next (if any) and previous (if any) cards in the group. Each card in the group has its group field set appropriately. Finally, the size of the group must equal the number of cards. It is part of this assignment to implement the invariant.

The Group class provides methods to add and remove cards from a group. It also has a method to sort the cards using the “insertion sort” algorithm.

The textbook includes an extensive discussion of insertion sort on arrays (pp. 622ff, or 606ff, 3rd ed.). Please read this to understand the algorithm. In particular, note that insertion sort has excellent performance on already sorted lists. In our case we will be using insertion sort on a doubly-linked list, which means that insertion is done by adding and removing elements in a linked list (avoiding the need to shift elements over).

4 The Game “War”

We define a fast-moving variant of the child’s game of “War.” Each player is dealt 10 cards. Each round, each puts forward a card. The higher card wins. If there is a tie (called a war), each plays a second card, and the winner collects all cards. If that is a tie also, the war continues.

The default implementation simply plays the next card in its hand each time. You should define a strategy that overrides the play method to select instead the highest card remaining in the hand if there is a war (do not sort!). The PlayWar.main method creates an instance of your subclass SmartWarPlayer which should be defined in its own file. It will play against the default player. See if you can tell whether it achieves any better result.

5 Your task

Your task is first to implement the following methods of the Cards.Group nested class:
wellFormed() The private method to evaluate the invariant. Hint: first and last are similar to head and tail; you need to check that prev and next pointers are consistent; you need to check each card’s group pointer; and the size field must be consistent with the structure.

isEmpty() Return true if the group is empty.

count() Return the number of cards in the group.

add(Card) Add a new card to the end of the group.

draw() Remove the first card from the group.

remove(Card) Remove a card from the group.

sort(Comparator<Card>) Sort the group using the given comparator.

Except for the invariant checker and sort, all of these methods should have efficient, constant-time \(O(1)\) (no loops!) implementations. The sort method must implement insertion sort efficiently; in particular, it should operate in linear \(O(n)\) time if the list is already sorted or almost sorted. Moreover, in such a case, when the list is already sorted, it should not change any pointers in the cards, even temporarily.

Your second task is to define a class SmartWarPlayer that tries to use the strategy given above to win “War” often. It should extend PlayWar and override play but no other method.

6 Files

The homework5.git repo includes the following:

src/TestCard.java A test driver to test the Card and Card.group ADTs.

src/TestInvariant.java A test suite for the invariant.

src/TestEfficiency.java A test suite for efficiency tests. If it takes more than a few seconds, you are implementing things wrong.

src/edu/uwm/cs351/Card.java The complete ADT and incomplete Card.group ADT.

src/edu/uwm/cs351/CardUtil.java Some example comparators for cards, including one for Sheepshead.

src/edu/uwm/cs351/PlayWar.java A program to play a simple card game.

You need to add the following file:

src/edu/uwm/cs351/SmartWarPlayer.java The strategic “War” player. Make sure you add this file to the repo!