# CS 2223 D21 Term. Homework 2

## Homework Instructions

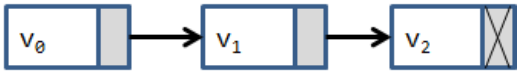
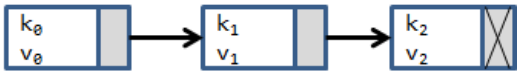
- This homework is to be completed individually. If you have any questions as to what constitutes improper behavior, review the examples as I have posted online http://web.cs.wpi.edu/~heineman/html/teaching_/cs2223/d21/#policies.
- Due Date for this assignment is **April 15th**. Homeworks received after 10AM receive a 25% late penalty. Homeworks received after 6PM will receive zero credit.
- Submit your assignments electronically using the canvas site for CS2223. Login to canvas.wpi.edu and locate HW2. You must submit a single ZIP file that contains all of your code as well as the written answers to the assignment.
- All of your Java classes must be defined in a package USERID where USERID is your CCC user id.

## Homework Context

This homework is concerned with how memory is allocated to store information. In Java (and most programming languages) you have two possible choices:

- Contiguous memory that stores an array of values
- Fragmented nodes with pointers to other nodes

But there are some interesting structural variations to understand, as seen below

| | |
|---|---|
| Fragmented | *Linked List of values*   *Linked List of (k,v) pairs*   |

## Getting Started

Copy the files from **`algs.hw2.submission`** into your **`USERID.hw2`** package.

40 pts **Q1.0** Linked Lists
10 pts **Q1.1** In() shuffles to return (1 bonus pt.)
10 pts **Q1.2** Out() shuffles to return (1 bonus pt.)
8 pts **Q1.3** In() shuffles to reverse (2 bonus pt.)
12 pts **Q2** Explorations using queues
20 pts **Q3** Mathematical Analysis (1 bonus pt.)
1 pt **BONUS** Continued Fractions

## Q1.0 Linked Lists (40 points)

The central structure for this homework assignment is a linked list of nodes representing a deck of playing cards. A standard deck of playing cards contains 52 cards of four *suits* (Clubs, Diamonds, Hearts, and Spades). For each suit there are thirteen cards, each with its own *rank* in order of Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King. Each card is represented by a number and an abbreviated suit, thus 8H is the eight of hearts, QD is the Queen of Diamonds, AS is the Ace of Spades, and KC is the King of Clubs.

When you open a factory-sealed deck of playing cards, the cards are in their proper sorted order, which is in order of suit (Clubs → Diamonds → Hearts → Spades) and then in order of rank from Ace to King.

AC → 2C → … QC → KC → AD → 2D → … QD → KD → AH → 2H → … → KH → AS → 2S → … → KS.

I have provided the following **`algs.hw2.Deck`** class that you will **extend** to implement your own Deck implementation. Details can be found in the code repository.

```
package algs.hw2;
public abstract class Deck {
  protected Node first; // first Node in the deck (i.e., the TOP CARD)
  protected Node last;  // last Node in the deck (i.e., the BOTTOM CARD)

  public abstract Card peekTop();               // O(1) performance [2 pts]
  public abstract Card peekBottom();            // O(1) performance [2 pts]
  public abstract Deck copy();                  // O(N) performance [5 pts]
  public abstract int size();                   // O(1) performance [2 pts]
  public abstract boolean match(Card c, int n); // O(N) performance [2 pts]

  public abstract boolean isInOrder();          // O(N) performance [2 pts]
  public abstract boolean isInReverseOrder();   // O(N) performance [2 pts]

  public abstract void out();                   // O(N) performance [8 pts]
  public abstract void in();                    // O(N) performance [8 pts]
  public abstract String representation();      // O(N) performance [2 pts]

  protected abstract Node cutInHalf();          // O(N) performance [5 pts]
}
```

In your own **`MyDeck`** class, which extends this class, you must define the constructor **`MyDeck(int max_rank)`** which creates a deck containing **`4*max_rank`** cards in factory order. Invoking **`MyDeck(3)`** creates a deck of twelve cards in the following order (AC is first and 3S is last):

AC → 2C → 3C → AD → 2D → 3D→ AH → 2H → 3H→ AS → 2S → 3S

The primary responsibility of your **`MyDeck`** implementation is to maintain the ordering of the cards while the deck is being manipulated. The **`in()`** and **`out()`** shuffle techniques split the deck into a Left and Right and perfectly shuffles the cards together to make a new deck. You are to do this entirely using linked lists.

## Out() performs a perfect Faro out shuffle

In an **`out()`** shuffle, the topmost and bottommost cards remain unchanged. Starting from a deck of N=8 cards in its factory ordering:

AC → 2C → AD → 2D → AH → 2H → AS → 2S

The deck is split into two subdecks of N/2 cards:

Left: AC → 2C → AD → 2D
Right: AH → 2H → AS → 2S

The deck is recombined by selecting cards from Left and Right, in alternating turns:

```
* LEFT:    AC    2C    AD    2D
* RIGHT:    | AH  | 2H  | AS  | 2S
*           |  |  |  |  |  |  |  |
*           v  v  v  v  v  v  v  v
*
* RESULT:  AC AH 2C 2H AD AS 2D 2S
```

The result is AC → AH → 2C → 2H → AD → AS → 2D → 2S

## In() performs a perfect Faro in shuffle

In an **`in()`** shuffle, the topmost and bottommost cards are changed. Starting from a deck of N=8 cards in its factory ordering:

AC → 2C → AD → 2D → AH → 2H → AS → 2S

The deck is split into two subdecks of N/2 cards:

Left: AC → 2C → AD → 2D
Right: AH → 2H → AS → 2S

The deck is recombined by selecting cards from Left and Right, in alternating turns:

```
* LEFT:       AC    2C    AD    2D
* RIGHT:   AH  | 2H  | AS  | 2S  |
*           |  |  |  |  |  |  |  |
*           v  v  v  v  v  v  v  v
*
* IN:      AH AC 2H 2C AS AD 2S 2D
```

The result is AH → AC → 2H → 2C → AS → AD → 2S → 2D

Once you can rearrange the contents of a deck simply by calling **`in()`** and **`out()`** to shuffle it repeatedly, it becomes natural to consider how the deck is ordered. The **`isInOrder()`** function returns **`True`** if the deck is in its original, factory-sealed ordering, while the **`isInReversedOrder()`** function returns **`True`** if the deck is the exact reverse of the factory-sealed ordering.

For a deck of eight cards (whose **max_rank** is 2), the reverse order would be:

2S → AS → 2H → AH → 2D → AD → 2C → AC

You will find it useful and necessary to implement the **representation()** method that produces a space-separated string representing the order of cards in the deck. For the above reversed factory order, the representation would be:

```
"2S 2H 2D 2C AS AH AD AC"
```

The last method to implement is **match(Card c, int n)** that determines whether the $n^{th}$ card in the deck (from the top) matches **c**.

> You should consider starting your development with a deck of 8 or 12 cards (with **max_rank** of 2 or 3) and then – once that is working – scale up to a full deck of 13 cards. This will make it easier to debug your code.

## Q1.1. How many in() shuffles to return deck to original position [10 points]

If you construct a deck using different **max_rank** (that is, other than 13 for a standard deck of playing cards) you will find that using a different number of **in()** shuffles will return it to its original configuration.

For example, given a standard deck of playing cards in some ordering, after 52 **in()** shuffles, the deck is restored back to that ordering, but with a **max_rank** of 5, it takes 6 **in()** shuffles to restore the ordering.

You are to write a program that computes the number of orderings needed to return a deck with any given **max_rank** ≥ 1 to its original configuration.

Your program should compute up to **max_rank**=20 (that is, a deck containing 80 cards) and print the full table like shown here.

| max_rank | #in() |
|---|---|
| 1 | 4 |
| 2 | 6 |
| 3 | 12 |
| 4 | 8 |
| **5** | **6** |
| 6 | 20 |
| 7 | 28 |
| 8 | 10 |
| 9 | 36 |
| 10 | 20 |
| 11 | 12 |
| 12 | 21 |
| **13** | **52** |

**Q1.1.1 Bonus Question (1 pt.)**

Can you come up with a formula that predicts the maximum # of **in()** shuffles for any **max_rank**, to return to the original configuration?

## Q1.2. How many out() shuffles to return deck to original position [10 points]

If you construct a deck using different **max_rank** (that is, other than 13 for a standard deck of playing cards) you will find that you can repeatedly shuffle the deck using just **out()** shuffles and it will eventually return to its original configuration.

For example, given a standard deck of playing cards in some ordering, after 8 **out()** shuffles, the deck is restored back to that ordering, but with a **max_rank** of 5, it takes 18 **out()** shuffles to restore the ordering.

Your program should compute up to **max_rank**=20 (that is, a deck containing 80 cards) and print the full table like you see here.

| max_rank | #out() |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 10 |
| 4 | 4 |
| **5** | **18** |
| 6 | 11 |
| 7 | 18 |
| 8 | 5 |
| 9 | 12 |
| 10 | 12 |
| 11 | 14 |
| 12 | 23 |
| **13** | **8** |

**Bonus Question 1.2.1 (1 pt.)**
Can you state the condition for which *k* **out()** shuffles returns a deck of **4*max_rank** cards to its original configuration?

## Q1.3. How many in() shuffles to reverse the state of a deck? [8 points]

For a standard deck of 13 cards, how many **in()** shuffles does it take to convert a deck into the reversed order? That is, starting from a standard 52-card deck in factory-sealed ordering, how many **in()** shuffles are needed to reverse the cards so the ordering is:

KS → … → 2S → AS → KH → … 2H → AD → KD → … 2D → AD → KC → … → AC

Write a program that computes and prints the number of **in()** shuffles to reverse the state of a regular 52-card deck of cards.

**Bonus Question 1.3.1 (1 pt.)**
What is the smallest **max_rank** for which no amount of **in()** shuffles produce the reversed ordering?

**Bonus Question 1.3.2 (1 pt.)**
Can you find a pattern for the values of **max_rank** for which no amount of **in()** shuffles reverses the deck? Note: I have not yet been able to see the pattern…

## Q2. Explorations Using Queues and Linked Lists [12 pts.]

Functional recursion uses a call stack to remember progress as it recursively solves smaller and smaller instances of the same problem, as you have seen with **Merge Sort**.

For this question, you will use a **Queue** to conduct a (potentially) infinite search which cannot be accomplished through recursion, mainly because there is no Base Case to stop a potentially infinite computation.

You will explore every possible arrangement of **`in()`** and **`out()`** shuffles to determine the smallest number of combined **`in()`** and **`out()`** shuffles from a factory-sealed deck that places a given card on the top of the deck. Just think about how a magician could use this to amaze an audience!

For example, to move the Four of Diamonds (4D) to the top of a factory-sealed deck of cards, simply do three **out()** shuffles, followed by one more **in**() shuffle, and the top card is a Four of Diamonds!

```
Deck d = new MyDeck(13);
d.out();
d.out();
d.out();
d.in();
System.out.println("is true:" + d.match(new Card("4D"), 1));
```

The Deck API has a **`match()`** method so you can confirm your results as shown above.

I provide a helper **`State`** class to record a **`Deck`** and a **`String`** representing the shuffle sequence to achieve that deck, using **`"I"`** and **`"O"`** characters to represent an **`in()`** and an **`out()`** shuffle. For example, the above deck (which placed a 4D on the top) would have a shuffle string of **`"OOOI"`**.

Copy the **`algs.hw2.submission.Q2`** Java class into your **`USERID.hw2`** package and modify it to use the following exploration algorithm which uses a **`Queue`** to maintain its exploration status, and a **`SequentialSearchST`** object, **`recorded`**, to keep track of the shuffles needed to place a given card on the top of the **`Deck`**, from the initial factory-sealed order.

```
AC
2C      IOIOI
3C      OIIIII
4C      OIOI
5C      IIIII
6C      IOIOII
7C      IOI
8C      OIIIOI
9C      OOOOI
10C     IIII
…
```

Start with a **`Queue`** that contains a single **`State`** class for the initial, factory-sealed **`Deck`** and an empty shuffle string **`""`**. The **while** loop will continue until the size of **`recorded`** is the size of a deck; it stops once it has found a sequence of shuffles for each card in the deck.

The key exploration step is to dequeue a past state, **`s`**, from the queue **and then enqueue two additional states** to reflect the fact that from state, **`s`**, you can generate two additional states by (a) performing an **`in()`** shuffle; and separately (b) performing an **`out()`** shuffle. The **`copy()`** method in your **`Deck`** class will prove to be essential here. For each state, **`s`**, you record the new shuffle sequence by appending **`"I"`** or **`"O"`** to the existing shuffle sequence, **`s.shuffle`**. Output a table for all 52 cards in order that looks like the above, but extends to show all 52 cards and their shuffle sequence. Use the **`algs.hw2.AllCards`** helper class to iterate over all cards in a regular deck.

## Q3. Mathematical Analysis [20 pts.]

This question is a more complicated version of what you will see on the midterm exam. You can find this code in the **`algs.hw2.submission.Q3`** class. Copy this class into **`USERID.hw2.Q3`** and modify it based on the requirements below.

Given the following `proc` function, let S(N) be the number of times `power(base, exp)` is invoked when calling `proc(a, 0, n-1)` on an array, **`a`**, of length **`n`** containing integer values from **`0`** to **`n-1`**.

```
static long power(int base, int exp) {
  return (long) Math.pow(base, exp);
}

public static long proc(int[] a, int lo, int hi) {
  if (lo == hi) {
    return power(a[lo], 2) + power(a[hi],2);
  }

  int m = (lo + hi) / 2;
  long total = proc(a, lo, m);
  while (hi > lo) {
    m = (lo + hi) / 2;
    total += power(a[m], 2);
    hi = m;
  }

  return total;
}
```

For this assignment, develop the recurrence relationship for S(N) and compute its closed-form formula. Then modify the `Q3` class to output an updated table that shows the computed counts and the result of your model.

**Question 3.1 (8 pts.)**

Identify the Base Case for S() and the Recursive Case for S(n). Refer back to lecture for the format of this question.

**Question 3.2 (12 pts.)**

Derive an exact solution to the recurrence for S(N) when N is a power of 2. Be sure to show your work.

**Bonus Question 3.3 (1pt.)**

Can you derive a formula that predicts the `Value` printed for `proc(a, 0, a.length-1)` when a contains the integers from 0 to n-1 and N is a power of 2.

## BONUS-1 Working with Continued Fractions (1 pts)

In mathematics, a continued fraction is an expression obtained through an iterative process of representing a number as the sum of its integer part and the reciprocal of another number, then writing this other number as the sum of its integer part and another reciprocal, and so on

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

The above continued fraction is represented as [$a_0$; $a_1$, $a_2$, $a_3$, …, $a_n$].

Here is how you find the continued fraction for a positive number **x**. Let $a_0$ be `floor(x)` or the largest integer that does not exceed **x**. Now set $b_0$ to the fractional leftover, or **x** – $a_0$.

For example, to find a continuous fraction for π, set $a_0$ = `floor(3.1415…)` = 3 and $b_0 \approx$ `0.14159`

Note that if you set $x_1 = \frac{1}{b_0}$ you have $x_1 \approx 7.062513$:

$$x_0 = a_0 + b_0$$
$$= a_0 + \frac{1}{x_1}$$

Truncate this last term, $x_1$, and our first approximation to π is $3 + \frac{1}{7}$ which you might be familiar with from grade school as $\frac{22}{7}$ = 3.1428… which is π accurate to two decimals. Define this expansion as [3; 7].

Let's conduct this expansion one more step, starting with $x_1$ and repeating the process. Let $a_1$ be `floor(`$x_1$`)` = 7 and $b_1 \approx$ `0.062513`. Note that if you set $x_2 = \frac{1}{b_1}$ you have $x_2 \approx 15.99659$:

$$x_1 = a_1 + b_1$$
$$= a_1 + \frac{1}{x_2}$$

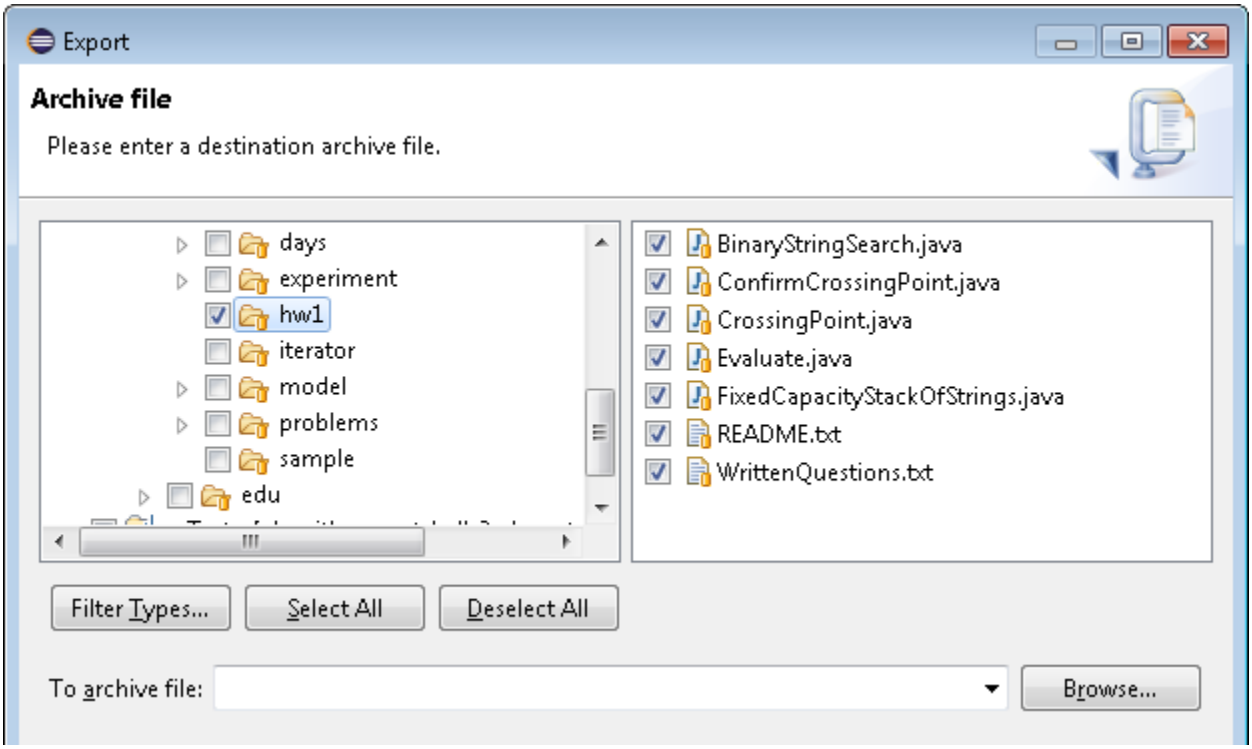
Truncate this last term, $x_2$, and our second approximation to π is $3 + \frac{1}{7+\frac{1}{15}}$ which evaluates to $3 + \frac{1}{\frac{106}{15}} = 3 + \frac{15}{106} = \frac{333}{106}$ = 3.141509… which is π accurate to four decimal places. Define this expansion as [3; 7, 15].

This process can be repeated for as long as the precision allows, or the final remainder is 0 and the continued fraction is finite. Using double precision, you should be able to compute that `[3;7,15,1,292]` is the continued fraction for π. Your implementation must use the provided Node class to construct linked lists.

## Submission Details

Each student is to submit a single ZIP file that will contain the implementations. In addition, there is a file "WrittenQuestions.txt" in which you are to complete the short answer problems on the homework.

The best way to prepare your ZIP file is to export your entire **`USERID.hw2`** package to a ZIP file using Eclipse. Select your package and then choose menu item "**Export…**" which will bring up the Export wizard. Expand the **General** folder and select **Archive File** then click **Next**.




You will see something like the above. Make sure that the entire "hw2" package is selected and all of the files within it will also be selected. Then click on **Browse…** to place the exported file on disk and call it USERID-HW2.zip or something like that. Then you will submit this single zip file in canvas.wpi.edu as your homework2 submission.

## Addendum

If you discover anything materially wrong with these questions, be sure to contact the professor or TA/SAs posting to the discussion forum for HW2 on Discord;

When I make changes to the questions, I enter my changes in red colored text as shown here.

## Change Log

1. Fixed mistaken description in Q1.2
2. Grrrr. My in() and out() tables were with the wrong question. Updated.
3. At top of page 4, I had the example of a reversed deck wrong. UPDATED