**UC Berkeley – Computer Science**
CS61B: Data Structures

Midterm #1, Fall 2022

Write the statement *"I have neither given nor received any assistance in the taking of this exam."* below.

I have neither given nor received any assistance in the taking of exam.

Signature: _____

| # | Points | # | Points |
|---|---|---|---|
| 0 | 1 | 6 | 420 |
| 1 | 359 | | |
| 2 | 300 | | |
| 3 | 300 | | |
| 4 | 300 | | |
| 5 | 720 | | |
| | | **TOTAL** | 2400 |

Name: _____Jonathan_____

SID:

_____933067241_____

GitHub Account # : fa22-s_____

Person to Left's # : fa22-s_____

Person to Right's #: fa22-s_____

Exam Room: _____Soda 275_____

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- Not all information provided in a problem may be useful, and **you may not need all lines.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs in the exam, we'll announce a fix. **Unless we specifically give you the option, the correct answer is not 'does not compile.'**
- **Do not use ternary operators or lambda functions.**
- ○ indicates that only one circle should be filled in.
- ☐ indicates that more than one box may be filled in.
- For answers which involve filling in a ○ or ☐, please fill in the shape completely.

**0. So it begins (1 point).** Write your name and ID on the front page. Write the exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your GitHub account # (e.g. fa22-s185) in the corner of every page. If you are taking the exam remotely, make sure that you are screen sharing, are recording your workspace, and your mic is unmuted.

**1. Residents & Friends (359 points).**

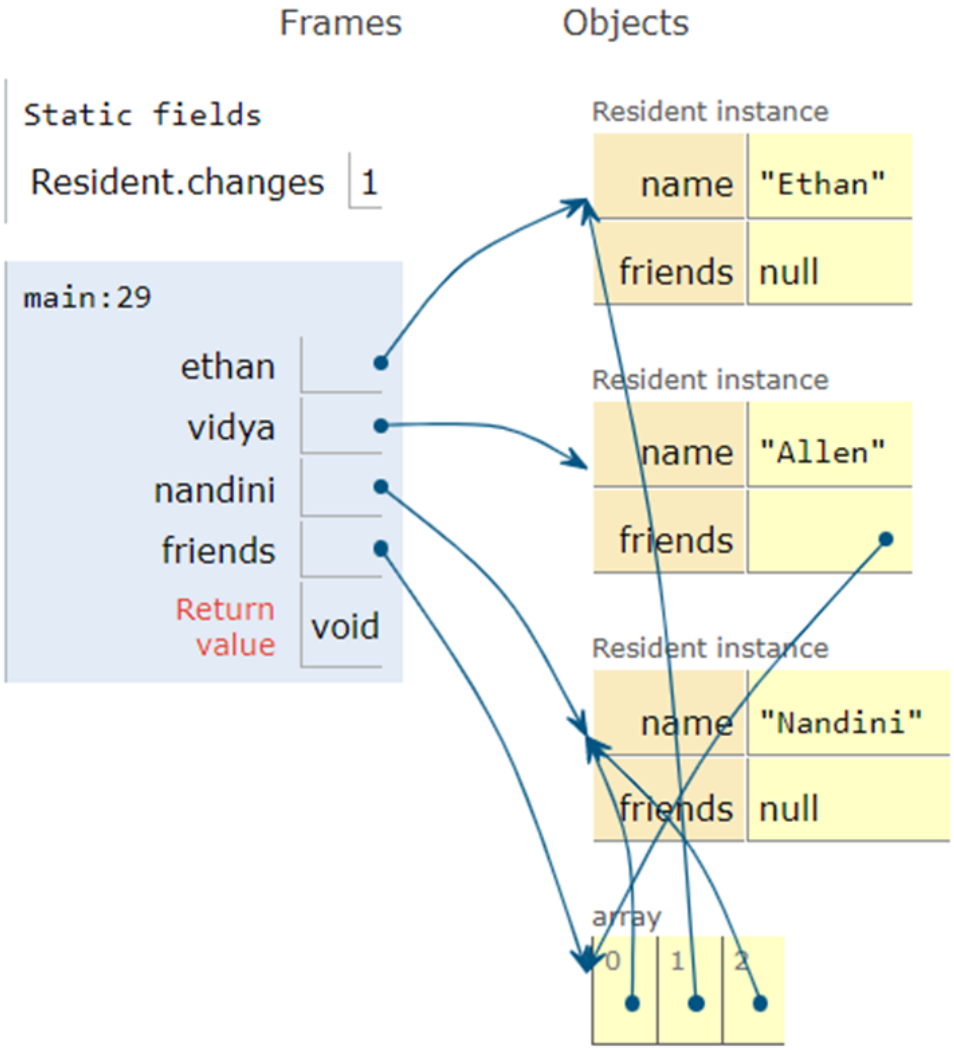
a) **(300 points)** Draw the box and pointer diagram (on the next page) resulting from executing the main method. **If a line would error**, write an "X" in the box next to the line and continue executing the program as if that line of code was never run.

```
public class Resident {
   public String name;
   private Resident[] friends;
   public static int changes = 0;

   public Resident(String n) {
       this.name = n;
       this.friends = null;
   }
   public Resident(String n, Resident[] friends) {
       this.name = n;
       this.friends = friends;
   }

   public void takeFriendsAndRenameOne(Resident a, String s) {
      this.friends = a.friends;
      this.friends[0].name = s;
      Resident.changes = Resident.changes + 1;
   }
   public static void main(String[] args) {
   [ ] Resident claire;
   [ ] Resident ethan = new Resident("Ethan");
   [ ] Resident vidya = new Resident("Vidya");
   [ ] Resident nandini = new Resident("Nandini");
   [ ] Resident[] friends = {vidya, ethan, nandini};
   [ ] vidya.takeFriendsAndRenameOne(new Resident("Jedi", friends), "Allen");
   [X] Resident.takeFriendsAndRenameOne(nandini, "Kyle");
   [ ] vidya.friends[0] = nandini;
   }
} // Reminder: If a line would error, follow the directions above.
```

`Box-and-pointer diagram`
Note: You may write Strings directly inside of the box instead of as a pointer to a `String` object. **Cross out all objects provided that are garbage collected or not used**.




b) **(59 points).** Say that instead of running the main method in the `Resident.java` class, we decide to run it in a new file called `CS61B.java`. Which line of code that ran successfully previously will now error, if any?

- ◯ `Resident claire;`
- ◯ `Resident[] friends = new Resident[]{vidya, ethan, nandini};`
- ◯ `vidya.takeFriendsAndRenameOne(new Resident("Jedi", friends), "Allen");`
- 🔴 `vidya.friends[0] = nandini;`
- ◯ `None of the above`

**2. IntList Of.**

**Background:** Java provides a useful shorthand called a "vararg" for when you want to pass an arbitrary number of values to a function without actually creating an array. For example, the function below sums all of the numbers provided to it. The input `items` is called a "vararg".

```
public static int sumMidFa22(int... items) {
    int sum = 0;
    for (int i = 0; i < items.length; i += 1) {
        sum += items[i];
    }
    return sum;
}
```

A function with a vararg input can be invoked with separate arguments, or can be invoked with an array of values. The example below shows the two different ways of calling `sumMidFa22`:

```
sumMidFa22(1, 2, 3, 4);  // sums four provided arguments, returns 10

int[] valuesToSum = new int[]{8, 10, 12};
sumMidFa22(valuesToSum); // sums contents of array, returns 30
```

**a. (300 Points).** Fill in the code below to write a method called `of` which returns an **`IntList`** of the given values, e.g. **`IntList`**`.of(1, 2, 3, 4)` would return an **`IntList`** with the values 1, then 2, then 3, then finally 4. See the reference sheet for a definition of the **`IntList`** class.

```
public static IntList of(int... x) {

    IntList L = null;

    for (int i = x.length - 1; i >= 0; i--) {

        L = new IntList(x[i], L);
    }

    return L;
}
```

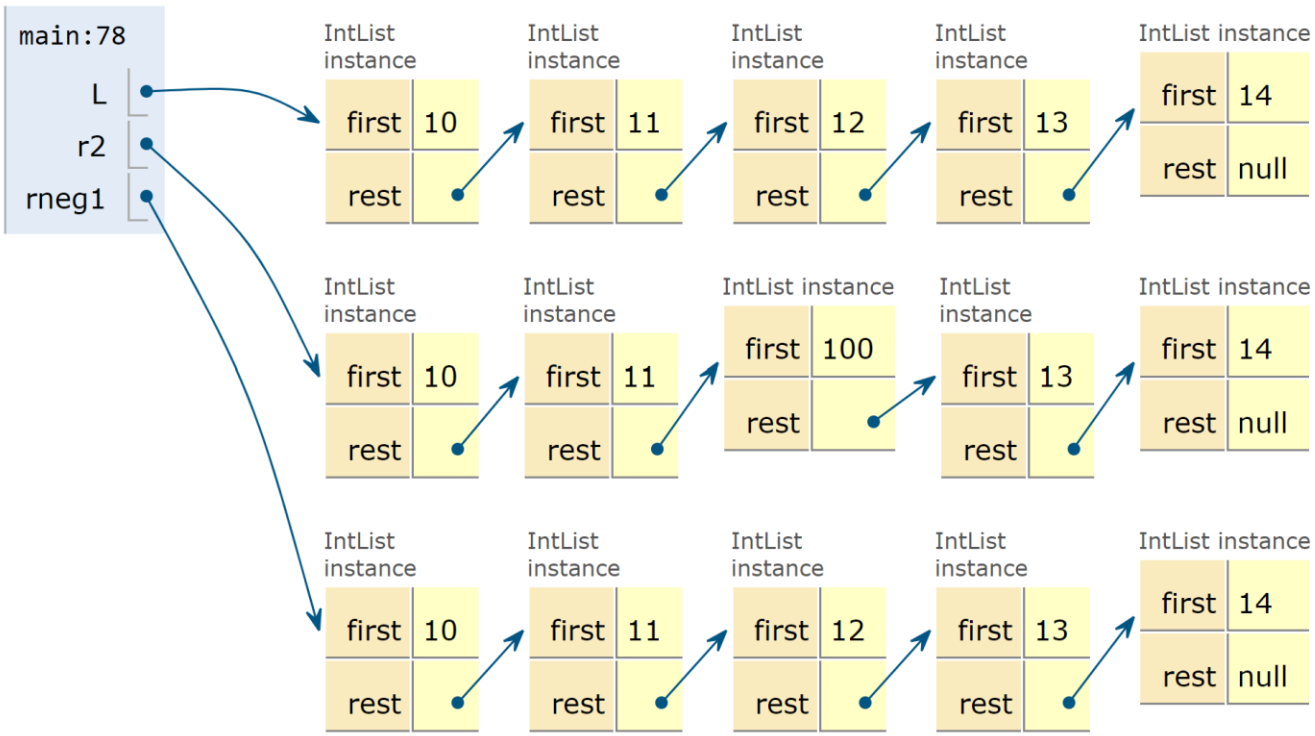
**b. (0 points).** What is the common term for modern pop music created in Kazakhstan?

**3. IntList replaceValue (300 Points).**

Fill in the `IntList` method below which replaces the value in the given position with x. For example, if our **`IntList`** is 10, 11, 12, 13, 14, then `replaceValue(L, 100, 2)` would return an IntList 10, 11, 100, 13, 14. **The method should not modify the `IntList` that is passed in, and none of the nodes in the original `IntList` should be referenced by the new list.** If `pos` is an invalid index, e.g. `pos` is greater than 4 or `pos` is less than 0 for the list containing 10, 11, 12, 13, 14, then your code should return a copy of the list with no changes made. **<u>See the reference sheet for a definition of the IntList class.</u>**

For example, the code below should result in the box and pointer diagram shown in the Visualizer:

```
IntList L     = IntList.of(10, 11, 12, 13, 14);
IntList r2    = IntList.replaceValue(L, 100, 2);
IntList rneg1 = IntList.replaceValue(L, 100, -1);
```




```
/** Returns copy of L where L[pos] = x. If pos invalid, returns copy of L.*/
    public static IntList replaceValue(IntList L, int x, int pos) {

        if (L == null) {
            return null;
        }
        if (pos == 0) {
            return new IntList(x, replaceValue(L.rest, x, pos - 1))
        }

        return new IntList(L.first, replaceValue(L.rest, x, pos - 1));
    }
```

**4. JUnit (300 points).** Fill in the JUnit tests below which provide evidence that `replaceValue` from problem 3 works correctly. Assume that `assertEquals` and `assertNotEquals` accurately compare the contents of the parameters when two IntLists are passed in. You may assume the `of` method from part 2 is implemented and works correctly. See the reference sheet for a list of JUnit methods.

Your two tests should show evidence of the following three properties of `replaceValue`:

1. **Test One**: The input **`IntList`** is not modified by `replaceValue`.
2. **Test One:** For a valid `pos` input, the desired value is changed in the returned list, and no other values are changed.
3. **Test Two:** The list returned does not contain references to nodes in the original list, i.e. all nodes are new.

**You do not have to use all provided lines.**

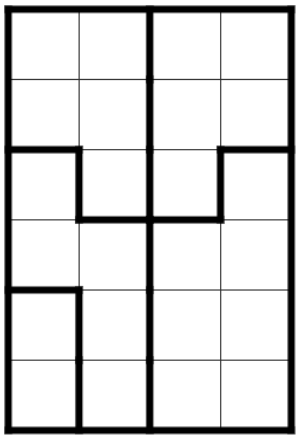
```
/** Calls replaceValue for a valid position and verifies that the original
list is not modified, and also verifies that the only changed value in the
copy of the list is the value in the desired position. */
@Test
public void testOne() {
   IntList L = IntList.of(1, 2);
   IntList w = IntList.replaceValue(L, 100, 1);
   assertEquals(IntList.of(1, 2), L);
   assertEquals(IntList.of(1, 100), w);
}
/** Calls replaceValue for an invalid position and verifies that the
returned list has no references to nodes in the original list. */
@Test
public void testTwo() {
   IntList L = IntList.of(1, 2);
   IntList W = IntList.replaceValue(L, 100, -1);
   assertTrue(L != W); //Necessary for full credit
   assertTrue(L.rest != W.rest); //Necessary for full credit
   assertTrue(L != W.rest);
   assertTrue(L.rest != W);
}
```

**You may complete this problem even if you did not correctly answer problem 3.**

**5. Norinori.**

"Norinori" is a genre of logic puzzles played on an NxM grid, divided into R regions. The goal is to place R dominoes (2 horizontally adjacent or 2 vertically adjacent shaded cells) on the grid such that no dominos are touching except at the corners, and that every region contains exactly two shaded cells, which are not necessarily part of the same domino.

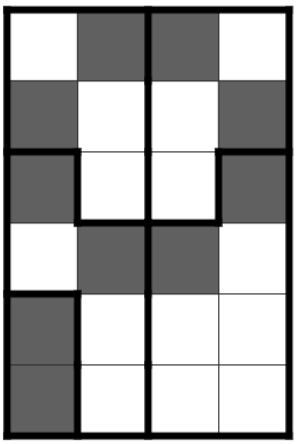
We represent ***the regions of a grid*** as a ***2D integer array***. Each location on the grid is labeled with a unique number from 0 to N-1, and the value of the array at arr[row][col] indicates the area that the grid cell at coordinates (row, col) is in. For example, this 6x4 grid may be represented by the 6x4 array. The top row [0, 0, 1, 1] tells us the two top left cells are in area 0, and the two top right cells are in area 1.



```
[
    [0, 0, 1, 1], // cells in areas 0,0,1,1
    [0, 0, 1, 1], // cells in areas 0,0,1,1
    [2, 0, 1, 3], // cells in areas 2,0,1,3
    [2, 2, 3, 3], // cells in areas 2,2,3,3
    [4, 2, 3, 3], // cells in areas 4,2,3,3
    [4, 2, 3, 3], // cells in areas 4,2,3,3
]
```

In the above diagram, (0, 0) would represent the top left corner of the grid, and (5, 3) the bottom right.

We represent ***prospective solutions to Norinori puzzles*** with a ***2D boolean array***. Let `true` indicate that a cell is shaded, and `false` indicate that a cell is unshaded. A (the) valid solution to the above puzzle is represented as follows:



```
[
    [false, true,  true,  false],
    [true,  false, false, true],
    [true,  false, false, true],
    [false, true,  true,  false],
    [true,  false, false, false],
    [true,  false, false, false],
]
```

Notice how the shaded cells all come in groups of 2, or dominoes. Notice that each region contains exactly two shaded squares. Notice that no 2 dominos are touching, except at their corners, i.e. any "orthogonally adjacent" shaded areas are part of the same domino. Your task in this problem will be to fill out the following 2 methods to fulfill their comments, ultimately building a method that checks whether a given norinori solution is valid.

a. **(270 points)** Fill in the code below.

```
/** Return the number of shaded cells orthogonally adjacent (up, left, down,
right) to cell[I][J]. You may assume I, J are valid inputs for the given
grid, e.g. for the example above, you may assume I is always between 0 and
5, and J is always between 0 and 3. You may assume that SHADED is
rectangular. Examples for the grid above:
   countAdjacentShaded(SHADED, 0, 0) = 2 // right and down are shaded
   countAdjacentShaded(SHADED, 3, 0) = 3 // up, right, and down are shaded
   countAdjacentShaded(SHADED, 3, 1) = 1 // right is shaded (self doesn't
 */                                                                ^^ count)

int countAdjacentShaded(boolean[][] shaded, int i, int j) {
      int result = 0;
      if (i != 0 && shaded[i - 1][j]) {
            result++;
      }
      if (j != 0 && shaded[i][j - 1]) {
            result++;
      }
      if (i != shaded.length - 1 && shaded[i + 1][j]) {
            result++;
      }
      if (j != shaded[0].length - 1 && shaded[i][j + 1]) {
            result++
      }
      return result;
}
```

---

61B STUDENT RELAXATION AND TRIVIA ZONE. What's something cool that happened in the summer? Feel free to write text or draw a picture.

b. **(450 points)** Fill in the code below.

```
/* Return true if the provided 2D array SHADED is a valid solution to the
norinori puzzle defined by the 2D array AREAS, in which there are NUMAREAS
distinct regions. You may assume that AREAS and SHADED are rectangular, and
the same dimensions. */
boolean validNorinoriSolution(int[][] areas, boolean[][] shaded, int numAreas) {
  int[] areaCounts = new int[numAreas];
  for (int i = 0; i < areas.length; i++) {
    for (int j = 0; j < areas[i].length; j++) {
      if (shaded[i][j]) {
        if (countAdjacentShaded(shaded, i, j) != 1) {
          return false;
        }
        areaCounts[areas[i][j]] += 1;
      }
    }
  }
  for (int c : areaCounts) {
    if (c != 2) {
      return false;
    }
  }
  return true;
}
```

Hint: If and only if:

- All shaded cells come in groups of two (i.e. all shaded cells belong to a 2 tile domino).
- No dominoes touch except their corner.

THEN:

- Every shaded cell on the grid has exactly one shaded neighbor to its up, down, left, or right, i.e. the one shaded neighbor is the other cell in its domino.

**6. Ooh baby a triple!! (420 Points).**
A "triply" linked list is a doubly linked list with an additional pointer field in each node:

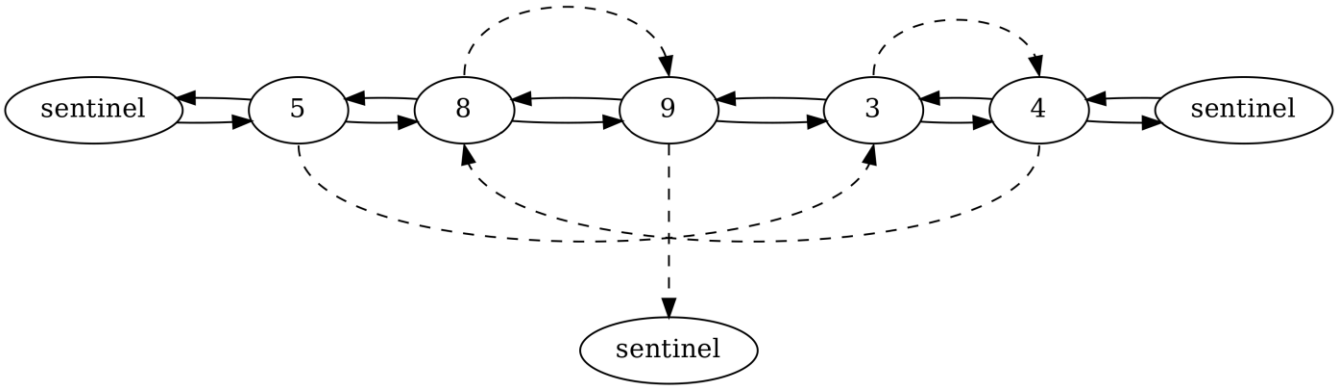
```
private static class TripleNode {
      private int value;
      private TripleNode left;
      private TripleNode right;
      private TripleNode next;  // dashed line in diagram below
      ...
}

public class TLList {
      private TripleNode sentinel; // there is only one sentinel node
      public TLList() {
          …
      }
}
```

For example, consider the Triply Linked List below. Some statements about this TLL:

- The sentinel's right is 5. 5's right is 8. … 4's right is the sentinel.
- The sentinel's left is 4. 4's left is 3 … 5's left is the sentinel.
- 5's next is 3. 3's next is 4. … 9's next is the sentinel.
- The sentinel's next is null.

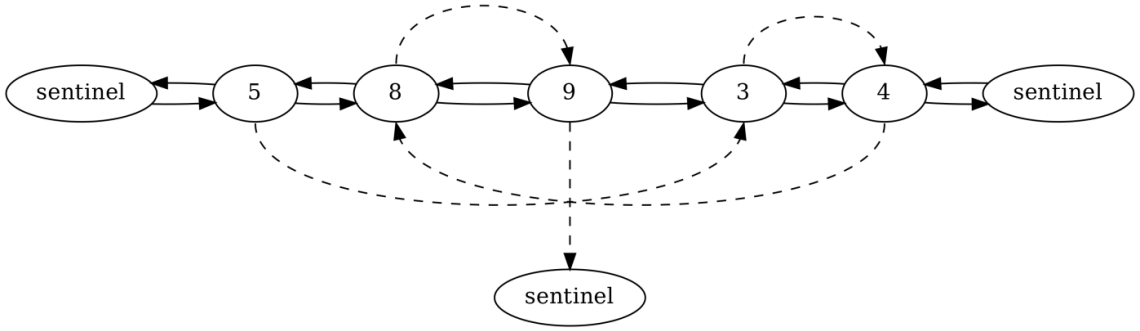
Note: There is only one sentinel node, but three copies are shown in the diagram[1] below to reduce visual clutter. If we showed the sentinel as one node, there would be arrows all over the place.



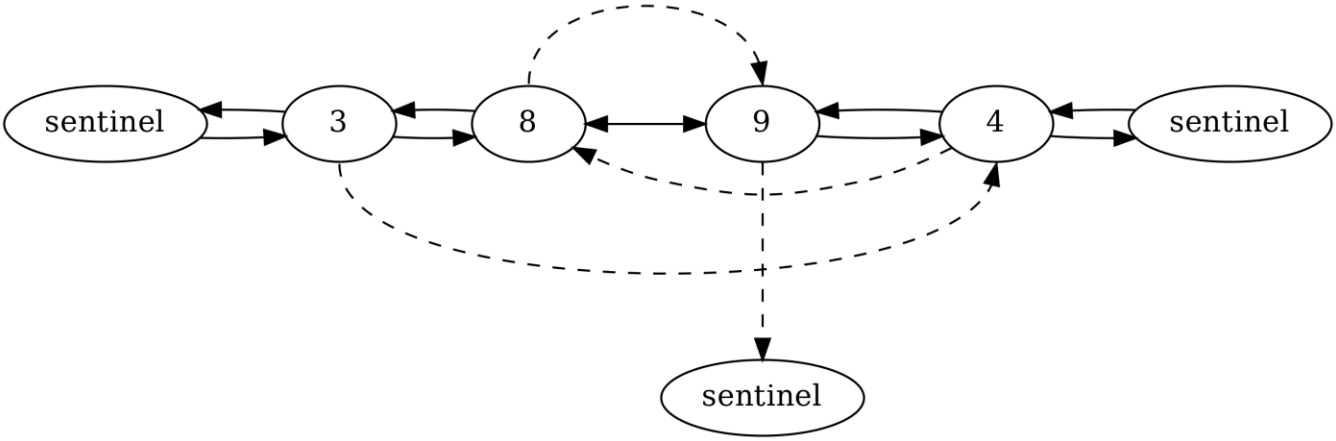

[1] In case you're curious, I used a tool called graphviz to automatically generate this diagram. It's useful! Though it can be a bit finicky. For example, on the next page, if you look closely, you'll see 8 and 9 connected with a single bidirectional arrow instead of two separate arrows. I really tried but couldn't get it to stop doing that! The intention in the diagram on the next page is that 8's right is 9, and 9's left is 8.

Let "alpha" be the node to the right of the sentinel, e.g. 5 in the diagram below. The `promoteNewAlphaDestroyOldAlpha` method removes the old alpha from the list. It also promotes a new alpha node by moving it to the leftmost position (i.e. to the right of the sentinel). The new alpha is the "next" item after the old alpha. No "next" pointers are changed by this operation.

For example if we start from the diagram on the previous page:




After the call to `promoteNewAlphaDestroyOldAlpha`, we have:




```
/**
* Remove the leftmost node and replace it with its "next" node. */
Hint: You may add more dot notation to each blank (e.g. for line 6, the left hand
side could read "newAlpha.left.next.right.next = …")
1: public void promoteNewAlphaDestroyOldAlpha() {
2:         TripleNode oldAlpha = sentinel.right;
3:         TripleNode newAlpha = oldAlpha.next;
4:
5:         newAlpha.left.right             = newAlpha.right;
6:         newAlpha.right.left = newAlpha.left;
7:
8:         newAlpha.right = oldAlpha.right;
9:         oldAlpha.right.left = newAlpha;
10:
11:        newAlpha.left = sentinel;
12:        sentinel.right = newAlpha;
13: }
```

**IntList**: **Reference Sheet**

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }
    /** Returns an IntList with the given numbers.*/
    public static IntList of(int... input) { ... }
}
```

JUnit methods:

```
assertEquals(Object x, Object y)
assertEquals(int x, int y)
assertEquals(double x, double y)
assertTrue(boolean b)
assertFalse(boolean b)
assertNotNull(Object x)
assertArrayEquals(Object[] x, Object[] y)
assertArrayEquals(int[] x, int[] y)
assertArrayEquals(double[] x, double[] y)
```