UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #1, Spring 2019

This test has 9 questions across 10 pages worth a total of 320 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement out below in the blank provided and sign. You may do this before the exam begins.**

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

Signature: _____

#	Points	#	Points
0	1	6	55
1	24	7	28
2	24	8	55
3	0	9	58
4	35		
5	40		
		TOTAL	320

Name: _____

SID: _____

GitHub Account # : sp19-s_____

Person to Left's # : sp19-s_____

Person to Right's #: sp19-s_____

Exam Room: _____

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.'**
- ○ 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. sp19-s185) in the corner of every page. Enjoy your free point ☺.

1. Static Swap.

a) **(12 points)** Consider the class shown below. On each line with a swap method, **fill in the boxes for every variable that is changed** by that **swap method call**. If a swap method causes no change, fill in "none" for that line. **No syntax errors or runtime errors occur.**

```
public class StaticSwap {
  public static int staticX = 5;
  public static int staticY = 10;
  public static void swap(int x, int y) {
    int temp = x; x = y; y = temp;
  }
  public static void staticSwap(int x, int y) {
    int temp = staticX; staticX = staticY; staticY = temp;
  }
  public static void main(String[] args) {
    int a = 5;
    int b = 10;
    swap(a, b);                    ☐ a ☐ b ☐ staticX ☐ staticY ☐ none

    swap(staticX, staticY);        ☐ a ☐ b ☐ staticX ☐ staticY ☐ none

    staticSwap(a, b);              ☐ a ☐ b ☐ staticX ☐ staticY ☐ none

    staticSwap(staticX,staticY);   ☐ a ☐ b ☐ staticX ☐ staticY ☐ none
  }
}
```

b) **(12 points)** Now imagine that every variable was of type **`String`** instead of type **`int`**. **No syntax errors or runtime errors occur.** Assume `swap` and `staticSwap` have been redefined to take **`String`**s.

```
  public static String staticX = "goose";
  public static String staticY = "hare";
  public static void main(String[] args) {
    String a = "moose";
    String b = "bear";
    swap(a, b);                    ☐ a ☐ b ☐ staticX ☐ staticY ☐ none

    swap(staticX, staticY);        ☐ a ☐ b ☐ staticX ☐ staticY ☐ none

    staticSwap(a, b);              ☐ a ☐ b ☐ staticX ☐ staticY ☐ none

    staticSwap(staticX,staticY);   ☐ a ☐ b ☐ staticX ☐ staticY ☐ none
```

2. JUnit (24 points). The Object class in Java defines the equals method as shown below.

```
public boolean equals(Object obj) { return (this == obj); }
```

The `org.junit.Assert.assertEquals` method looks like:

```
public static void assertEquals(Object expected, Object actual) {
    if (!expected.equals(actual)) { recordFailure(expected, actual); }
}
```

Suppose we define a class called `Dog` as follows:

```
1: public class Dog {
2:    private int size; private List<String> favoriteFoods;
3:    public Dog(int s, List<String> f) { size = s; favoriteFoods = f; }
4:    public boolean equals(Dog o) {
5:        if (this.size != o.size) {
6:            return false;
7:        }
8:        if (this.favoriteFoods != o.favoriteFoods) {
9:            return false;
10:        }
11:        return true;
12:    }
13:    public String toString() { ... };
14: }
```

Suppose we write a test as follows:

```
@Test
public void testBananaTofu() {
   Dog D1 = new Dog(5, List.of("banana", "tofu")); // List.of(...) creates a
   Dog D2 = new Dog(5, List.of("banana", "tofu")); // java.util.List<String>
   assertEquals(D1, D2);
}
```

Due to at least one error in the Dog class, this test fails with: "`expected:Dog<5, [banana, tofu]> but was:Dog<5, [banana, tofu]>`". Explain what you'd need to change so that the Dog class is correct and also passes `testBananaTofu`. **Refer to line numbers where possible. You may not need all lines.** Your changes should be written in English, but can include code.

Change 1: _____
Change 2: _____
Change 3: _____
Change 4: _____
Change 5: _____ _____

3. PNH (0 points) What famous work begins with this famous line? “In those days, in those far remote days, in those nights, in those faraway nights, in those years, in those far remote years, at that time the wise one who knew how to speak in elaborate words lived in the land.”

4. a) **More JUnit (10 points).** Suppose we add a method `equalLists` to our **`AList`** class with the signature below. This method returns `true` if the given **`List61B`** has all the same items as the current **`AList` in the same order.** Your midterm 1 reference sheet might be useful for this problem.

```
public boolean equalLists(List61B<T> otherList)
```

Write a JUnit test that verifies that this method correctly returns true when called with an **`SLList`** as an argument. Your lists should be of length 3. Assume all JUnit classes needed have been imported.

```
    public void testAListEqualToSLListOFLength3() {
        SLList<Integer> sll = new SLList<>();
        _____________________________________
        _____________________________________
        _____________________________________

        AList<Integer> all = new AList<>();
        _____________________________________
        _____________________________________
        _____________________________________


        _____________________________________
    }
```

b) **(25 points)** Write the method `equalLists`. **It should work for all possible inputs**, not just your JUnit test above. **Your method must be non-destructive.** You may not need all lines.

```
public class AList<T> { ...
   public boolean equalLists(List61B<T> otherList) {
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
        _____________________________________
   }
```

5. **SLList (40 points).** Suppose we have an **`SLList`** as defined in lecture, with a single sentinel node at the front. **See your midterm 1 reference sheet for the names of the fields.**

Fill in the recursive `insert` method below. **You may not need all lines. Do not write more than one statement on each line.** You may not use any `for` or `while` loops!

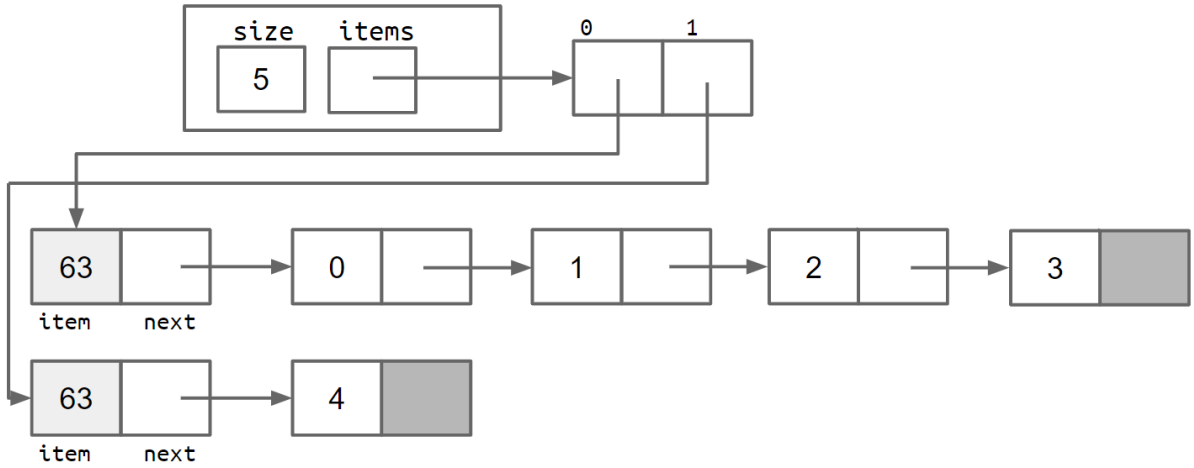
Assume this `insert` method is part of the `SLList` class.

```
/** Inserts item into given index. For example, if the list is currently
  * [0, 10, 20, 30], and we call insert(25, 3), the list will become
  * [0, 10, 20, 25, 30]. If we then call insert(-5, 0), the list will become
  * [-5, 0, 10, 20, 25, 30]. */
    public void insert(T item, int index) {
        if (index < 0 || index > size()) {
           throw new IllegalArgumentException();
        }
        ____________________________________;
        ____________________________________;
    }

    private void insert(T item, int index, _____________) {
        if (_________________) {
            ____________________________________;
            ____________________________________;
        } else {
            ____________________________________;
            ____________________________________;
        }
    }
```

6. XList. Let's define a new type of list known as an **`IntXList`** that only has an `addLast` operation. An **`IntXList`** is a hybrid of the `SLList` and `AList` ideas. In an `XList`, the data is stored as an array of **`IntSLLists`**, where each **`IntSLList`** must have size <= 4. When `addLast` is called, we always use the first list with available space. **`IntXLists`** store integers, i.e. are not generic.

For example, suppose we have created an **`IntXList`** and then call `addLast(0)`, `addLast(1)`, `addLast(2)`, `addLast(3)`, and `addLast(4)`. This would result in the box-and-pointer diagram below:



a) **(20 points)** Fill in the `addLast` method for the **`XList`** class so that it is correct and has reasonable performance. Assume that the resize method correctly resizes the **`IntSLList`** array. If you're stuck, consider doing part b first.

```
public class IntXList {
    private IntSLList[] items;
    private int size;
    public IntXList() {
        items = new IntSLList[1];
        items[0] = new IntSLList();
    }                                       // after calling resize,
    private void resize(int numSLLists) { // IntSLList[] items will be of
        …                                   // length numSLLists
    }
    public void addLast(int x) {
        if (_____________________________________) {
            resize(_____________________);
        }
        items[___________________]._____________________
        size += 1;
    }
}
```

b) **(35 points)** Fill in the `resize` method in the **`IntXList`** class. You may not need both loops or all lines.

```
    private void resize(int numSLLists) {
        IntSLList[] temp = ____________________;
        for (_______________;_______________;_______________) {
            ____________________________________;
        }
        for (_______________;_______________;_______________) {
            ____________________________________;
        }
        ____________________________________________;
        ____________________________________________;
    }
```

7) **Yeah One of These Problems (28 points).** Suppose we define the following two classes:

```
public class Deity {
    public void smite(Object o) { System.out.println("DO"); }
    public void smite(Deity o)  { System.out.println("DD"); }
    public void smite(Titan o)  { System.out.println("DT"); }
}
public class Titan extends Deity {
    public void smite(Object o) { System.out.println("TO"); }
    public void smite(Deity o)  { System.out.println("TD"); }
    public void smite(Titan o)  { System.out.println("TT"); }
}
```

For each of the lines below, fill in the bubble for the string that is printed. Or if the line has a compile or runtime error, fill in the "runtime error" (RE) or "compile error" (CE) bubble instead.

```
Titan T = new Titan();

Deity D = new Deity();
Deity Colette = new Titan();
```

`T.smite(Colette);`	○DO	○DD	○DT	○TO	○TD	○TT	○RE	○CE
`T.smite((Deity) Colette);`	○DO	○DD	○DT	○TO	○TD	○TT	○RE	○CE
`T.smite(D);`	○DO	○DD	○DT	○TO	○TD	○TT	○RE	○CE
`T.smite((Object) D);`	○DO	○DD	○DT	○TO	○TD	○TT	○RE	○CE
`((Deity) T).smite(D);`	○DO	○DD	○DT	○TO	○TD	○TT	○RE	○CE
`T.smite((Titan) D);`	○DO	○DD	○DT	○TO	○TD	○TT	○RE	○CE
`((Object) T).smite(D);`	○DO	○DD	○DT	○TO	○TD	○TT	○RE	○CE

8. **MetaComparison (55 points).** Given an **`IntList`** x, an **`IntList`** y, and a **`Comparator`**<**`Integer`**> C, the **`IntListMetaComparator`** performs a comparison between x and y.

Specifically, the **`IntListMetaComparator`** performs a pairwise comparison of all the items in x and y. If the lists are of different lengths, the extra items in the longer list are ignored. Let α be the number of items in x that are less than the corresponding item in y according to C. Let β be the number of items in x that are greater than the corresponding item in y according to C. If $\alpha < \beta$, then x is considered less than y. If $\alpha = \beta$, then x is considered equal to y. If $\alpha > \beta$, then x is considered greater than y. For example:

```
Comparator<Integer> c = new FiveCountComparator();//compares # of fives
IntList x = [ 55, 70,  90, 115, 5];               //e.g. 515 has 2 fives
IntList y = [150, 35, 215,  25];
IntListMetaComparator ilmc = new IntListMetaComparator(c);
ilmc.compare(x, y); // returns negative number
```

For the example above, according to the **`FiveCountComparator`** 55 > 150, 70 < 35, 90 < 215, 115 = 25. We have that $\alpha = 2$ and $\beta = 1$, and thus `ilmc.compare` will return a negative number.

```
public class IntListMetaComparator implements Comparator<IntList> {
    ____________________________________

    public IntListMetaComparator(Comparator<Integer> givenC) {
        ____________________________________
    }

    /* Returns negative number if more items in x are less,
       Returns positive number if more items in x are greater.
       If one list is longer than the other, extra items are ignored.
     */
    public int compare(IntList x, IntList y) {
        if ((__________________) || (____________________)) {
            ____________________________________
        }

        ____________________________________________
        if (_____________________) {
            return ____________________________;
        } else if (_____________________) {
            return ____________________________________;
        } else {
            return ____________________________________;
        }
} // Your reference sheet has a definition for IntList
```

9. Cool Iterators.

a) **(16 points)** Suppose we have an **`Iterator`** as defined below.

```
public class CoolIterator implements Iterator<Integer> {
    IntList L;

    public CoolIterator(IntList input) {
        L = input;
    }

    public boolean hasNext() {
        return L != null;
    }

    private IntList getNext(int x, IntList p) {
        if (p == null) { return null; }
        if (x == 0) { return p; }
        return getNext(x - 1, p.rest);
    }

    public Integer next() {
        int first = L.first;
        L = getNext(L.first, L);
        return first;
    }

    public static void main(String[] args) {
        IntList L = IntList.of(new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9});
        CoolIterator ci = new CoolIterator(L);
        while (ci.hasNext()) {
            System.out.print(ci.next());
        }
    }
}
```

What will be the output of the main method above? _____

b) **(42 points)** Suppose we want to build a **`BookendIterator`** class that iterates over only the first and last values provided by another **`Iterator`**. For example, if we run the code below, the code should print "cats" then "space".

```
List<String> L2 = List.of("cats", "live", "in", "space");
Iterator<String> it2 = L2.iterator();
Iterator<String> bit2 = new BookendIterator<>(it2);
while (bit2.hasNext()) {
    System.out.println(bit2.next());
}
```

Fill in the code for **`BookendIterator`** below. **You may assume that the input Iterator has at least two values** (i.e. it's OK if your code crashes or behaves strangely for an iterator with < 2 values)!

Partial credit will be especially hard to earn for this problem. **To receive 10% credit and skip this problem, fill in this box and leave the code below blank:** ☐

```
public class BookendIterator<T> implements Iterator<T> {
    ____________________________________________
    ____________________________________________
    ____________________________________________
    public BookendIterator(Iterator<T> input) {
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
    }
    public boolean hasNext() {
        ____________________________________________
        ____________________________________________
        ____________________________________________
    }
    public T next() {
        if (!hasNext()) { throw new NoSuchElementException(); }
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
    }
}
```