Final Exam
CS 1102 Computer Science 2

Fall 2018

Tuesday December 18, 2018
Instructor Muller

KEY

Before reading further, please arrange to have an empty seat on either side of you. Now that you are seated, please write your name **on the back** of this exam.

This is a closed-notes and closed-book exam. Computers, calculators, and books are prohibited.

- Partial credit will be given so be sure to show your work.
- Feel free to write helper functions if you need them.
- **Please write neatly.**

Part	Points	Out Of
1		8
2		8
3		8
4		8
5		8
Total		40

Part 1: (8 Points) Arrays

1. (4 Points Total) Write a function `int lastOdd(int[] a)` that returns the last odd integer in `a`. If there are no odd integers in `a`, `lastOdd` should return `0`.

   Answer:

```
int lastOdd(int[] a) {
  for (int i = a.length - 1; i >= 0; i--)
    if (a[i] % 2 == 1) return a[i];
  return 0;
}
```

2. (4 Points) Let `a` and `b` be sorted arrays of integers with M and N elements (resp). Write a function

```
void commons(int[] a, int[] b);
```

   such that a call `commons(a, b)` prints all of the integers in common **and which works in fewer than $\mathcal{O}(M \cdot N)$ steps**. For example, if `a` and `b` were defined as in

```
int[] a = {2, 3, 4, 6, 8, 12},
      b = {3, 4, 10, 12, 20};
```

   `common` would print `3`, `4` and `12`. (One extra point for solving it in fewer than $\mathcal{O}(M \cdot \log_2 N)$ steps.)

   Answer:

```
void commons(int a[], int[] b) {
  for (int i = 0; i < a.length; i++)
    if (binarySearch(a[i], b, 0, b.length - 1))
      System.out.format("a and b both have %d%n", a[i]);
}
boolean binarySearch(int key, int[] b, int lo, int hi) {
  if (lo > hi) return false;
  int mid = (lo + hi) / 2;
  if (key == b[mid]) return true;
  if (key < b[mid])
    return binarySearch(key, b, lo, mid - 1);
  else
    return binarySearch(key, b, mid + 1, hi);
}
```

Part 2: (8 Points) Lists

The problems in this section relate to generic singly-linked lists of the form

```
class List<T> {
  T info;
  List<T> next;
  List(T info, List<T> next) { this.info = info; this.next = next; }
}
```

1. (4 Points) Let `xs` and `ys` be lists of equal length. The *Hamming Distance* between `xs` and `ys` is the number of positions in which they differ (i.e., they are not `equals`). Write a function

   ```
   int hammingDistance(List<T> xs, List<T> ys)
   ```

   which returns the Hamming Distance between `xs` and `ys`.

   Answer:

   ```
   int hammingDistance(List<T> xs, List<T> ys) {
     if (xs == null)
       return 0;
     else
       return hammingDistance(xs.next, ys.next) + (xs.info.equals(ys.info) ? 0 : 1);
   }

   int hammingDistance(List<T> xs, List<T> ys) {
     int answer = 0;
     while (xs != null) {
       if (!xs.info.equals(ys.info)) answer++;
       xs = xs.next;
       ys = ys.next;
     }
     return answer;
   }
   ```

2. (4 Points) It's common to append one list to another. Write either a *mutable* or an *immutable* version of `List<T> append(List<T> ys, List<T> zs);`. See the attachment for an illustration.

```
List xs, ys = new List(1, new List(2, null)),
         zs = new List(3, new List(4, new List(5, null)));

              +---+                         +---+
           ys | o |                      zs | o |
              +-|-+                         +-|-+
                v                             v
              +---+---+   +---+---+       +---+---+   +---+---+   +---+---+
              | 1 | o-+-->| 2 | o-+-+     | 3 | o-+-->| 4 | o-+-->| 5 | o-+-+
              +---+---+   +---+---+ =     +---+---+   +---+---+   +---+---+ =
```

Please **write only one solution** and specify explicitly which version you are writing.

Answer:

```
mutable:
List<T> append(List<T> xs, List<T> ys) {
  if (xs == null)
    return ys;
  List<T> p = xs;
  while(p.next != null) p = p.next;
  p.next = ys;
  return xs;
}

immutable:
List<T> append(List<T> xs, List<T> ys) {
  if (xs == null)
    return ys;
  else
    return new List<T>(xs.info, append(xs.next, ys));
}
```

Part 3: (8 Points) Binary Trees

The problems in this section relate to generic binary trees of the form

```
class Tree<T> {
  T key;
  Tree<T> left, right;
  Tree(T key, Tree<T> left, Tree<T> right) {
    this.key = key;
    this.left = left;
    this.right = right;
  }
}
```

A binary tree is either empty (in Java, this is normally represented by `null`) or it is a `Tree` with a `key` field and recursive `left` and `right` fields.

1. (4 Points) Write an efficient function `boolean duplicateKeys(Tree<T> t)` which returns `true` if `t` contains duplicate keys. Otherwise `duplicateKeys` should return `false`.

   Answer:

   ```
   boolean containsDuplicates(Tree<T> t) {
     return cdHelp(t, new HashSet<T>());
   }

   boolean cdHelp(Tree<T> t, Set<T> set) {
     if (t == null) return false;
     if (set.contains(t.key)) return true;
     set.add(t.key);
     return cdHelp(t.left, set) || cdHelp(t.right, set);
   }
   ```

2. (4 Points) Write a function `Queue<T> levelOrder(Tree<T> t)` that returns a queue containing a level-order traversal of `t`. For example, let `t3` be defined as in

```
Tree<String> t1 = new Tree("A", new Tree("B", null, null),              G
                                new Tree("C", null, null)),           /   \
             t2 = new Tree("D", new Tree("E", null, null),           A     D
                                new Tree("F", null, null)),         / \   / \
             t3 = new Tree("G", t1, t2);                           B   C E   F
```

then a call `levelOrder(t3)` would return the queue

```
front ->  "G", "A", "D", "B", "C", "E", "F"  <- back
```

Feel free to use the built-in `ArrayQueue` implementation of the `Queue` type.

Answer:

```
Queue<T> levelOrder(Tree<T> t) {
  Queue<T> answer = new ArrayQueue<T>();
  Queue<Tree<T>> walker = new ArrayQueue<Tree<T>>();
  walker.add(t);
  while(!walker.isEmpty()) {
    Tree<T> s = walker.poll();
    if (s != null) {
      answer.add(s.key);
      walker.add(s.left);
      walker.add(s.right);
    }
  }
  return answer;
}
```

Part 4: (8 Points) Abstract Data Types

1. (4 Points) Java's `java.util.Map<K, V>` interface specifies a generic map ADT. The library class `java.util.HashMap<K, V>` provides an efficient and widely used implementation. Write a function

   ```
   Map<Character, Integer> frequency(String s)
   ```

   that returns a *frequency table* of the characters occurring in string `s`. For example, the function call `frequency("ALABAMA")` would return the map

   ```
   {'A' = 4, 'L' = 1, 'B' = 1, 'M' = 1}
   ```

   Feel free to use the `String` function `char charAt(int i)` to access the individual characters in the input string. (E.g., `"BC".charAt(0)` would result in `'B'`.) The relevant portion of the `Map` API is on the attached sheet.

   Answer:

   ```
   Map<Character, Integer> frequency(String s) {
     Map<Character, Integer> map = new HashMap<Character, Integer>();
     for (int i = 0; i < s.length(); i++) {
       char c = s.charAt(i);
       map.put(c, map.containsKey(c) ? map.get(c) + 1 : 1);
       }
     return map;
   }
   ```

2. (4 Points) Consider the humble `Stack<T>` ADT.

```
public interface Stack<T> {
  void push(T item);
  T pop();
  int size();
  boolean isEmpty();
}
```

Sequential representations usually resize when the stack gets full. Develop a non-resizing sequential implementation, `StackC<T>`, in which a `push` onto a full stack causes items to just fall off the bottom. For example,

```
Stack<Integer> s = new StackC<Integer>(3);                // capacity is 3
s.push(1);    s.push(2);    s.push(3);    s.push(4);

    1             2             3             4
                  1             2             3
                                1             2
```

As suggested in the example, your implementation should have a constructor that accepts a capacity parameter.

Answer:

```
class StackC<T> implements Stack<T> {
  private T a[];
  private int N, top, capacity;

  StackC(int capacity) {
    this.capacity = capacity;
    a = (T[]) new Object[capacity];
    N = 0;
    top = 0;
  }

  public T pop() {
    if (isEmpty()) throw new NoSuchElementException();
    top = top - 1;
    if (top < 0) top = a.length - 1;
    N = Math.max(0, N - 1);
    return a[top];
  }

  public void push(T item) {
    a[top++] = item;
    if (top == a.length) top = 0;
    N = Math.min(this.capacity, N + 1);
  }

  public int size() { return this.N; }
  public boolean isEmpty() { return this.N == 0; }
}
```

Answer for problem 4.2.

Part 5: (8 Points) Storage Diagrams

1. (2 Points) Using the class `List<T>` from above, consider the following code and show the state of the stack and heap just before (1) is executed. Execution is initiated with the call `f(3)`.

```
List<Integer> f(int i) {
  List<Integer> a = null;
  while(i > 0) {
    a = new List<Integer>(i, a);
    i = i - 1;
  }
  return a;                                          (1)
}

           stack                                              heap
```

Answer:

```
  +---+
  | f |
  +---+----+
  |  i : 0 |                                     +---+---+   +---+---+   +---+---+
  |  a : o-+----------------------------------> | 1 | o-+-->| 2 | o-+-->| 3 | o-+-+
  +--------+                                     +---+---+   +---+---+   +---+---+ =
```

2. (3 Points) Show the stack and heap just before (1) is executed. The code is initiated with the call `isEven(3)`.

```
boolean isEven(int n) {
  if (n == 0)
    return true;
  else
    return isOdd(n - 1);
}
boolean isOdd(int n) {
  if (n == 0)
    return false;                      (1)
  else
    return isEven(n - 1);
}
```

```
           stack                                         heap
```

Answer:

```
+------+
|isOdd |
+------+--+
|  n : 0  |
+---------+

+------+
|isEven|
+------+--+
|  n : 1  |
+---------+

+------+
|isOdd |
+------+--+
|  n : 2  |
+---------+

+------+
|isEven|
+------+--+
|  n : 3  |
+---------+
```

3. (3 Points) Using the class `Tree<T>` from above, consider the following code and show the state of the stack and heap after (1) has executed but before (2) has executed. If stack frames are popped, you don't need to show them, but show everything that has been allocated in the heap. The code is initiated with the call `f(3, 4)`.

```
int g(Tree<Integer> a, int i) {
  Tree<Integer> b = new Tree(i, null, null),
  Tree<Integer> c = new Tree(i, a, b);
  return 5;
}
int f(int i, int j) {
  Tree<Integer> a = new Tree(i, null, null);
  int k = g(a, i + j);                              (1)
  return k;                                         (2)
}
```

```
           stack                                         heap
```

Answer:

```
                                          +---------+
                                          | key : 7 |
                                          +---------+     +---------+
                                          | o  | o--+---->| key : 7 |
                                          +-|--+----+     +---------+
                                            |             | o  | o  |
+---+                                       |             +-|--+-|--+
| f |                                       |               =    =
+---+----+                                  |
|  i : 3 |                                  v
|  j : 4 |                                +---------+
|  a : o-+------------------------------> | key : 3 |
|  k : 5 |                                +---------+
+--------+                                | o  | o  |
                                          +-|--+-|--+
                                            =    =
```

Attachment 1: Portions of the generic Map and Queue ADTs

Map

```
boolean containsKey(Object key) -- Returns true if this map contains
                                   a mapping for the specified key.

V get(Object key) -- Returns the value to which the specified key is
                     mapped, or null if this map contains no mapping
                     for the key.

boolean isEmpty() -- Returns true if this map contains no key-value
                     mappings.

V put(K key, V value) -- Associates the specified value with the
                         specified key in this map.  If the map
                         previously contained a mapping for the key,
                         the old value is replaced by the specified value.

int size() -- Returns the number of key-value mappings in this map.
```

Queue

```
boolean add(E e) -- Inserts the specified element into this queue if it
                    is possible to do so immediately without violating
                    capacity restrictions, returning true upon success
                    and throwing an IllegalStateException if no space
                    is currently available.

E element()      -- Retrieves, but does not remove, the head of this queue.

E peek()         -- Retrieves, but does not remove, the head of this queue,
                    or returns null if this queue is empty.

E poll()         -- Retrieves and removes the head of this queue, or returns
                    null if this queue is empty.

E remove()       --  Retrieves and removes the head of this queue.

boolean contains(Object o) -- Returns true if the queue contains the
                              specified element.
```

Attachment 2: Appending Two Lists

```
List xs, ys = new List(1, new List(2, null)),
         zs = new List(3, new List(4, new List(5, null)));

               +---+                       +---+
            ys | o |                    zs | o |
               +-|-+                       +-|-+
                 v                           v
               +---+---+   +---+---+       +---+---+   +---+---+   +---+---+
               | 1 | o-+-->| 2 | o-+-+     | 3 | o-+-->| 4 | o-+-->| 5 | o-+-+
               +---+---+   +---+---+ =     +---+---+   +---+---+   +---+---+ =


-------------------------------------------------------------------------------

xs = append(ys, zs); (Mutable Version)

               +---+                       +---+
            ys | o |                    zs | o |
               +-|-+                       +-|-+
                 v                           v
    +---+      +---+---+   +---+---+       +---+---+   +---+---+   +---+---+
 xs | o-+----> | 1 | o-+-->| 2 | o-+-----> | 3 | o-+-->| 4 | o-+-->| 5 | o-+-+
    +---+      +---+---+   +---+---+       +---+---+   +---+---+   +---+---+ =


-------------------------------------------------------------------------------

xs = append(ys, zs); (Immutable Version)

               +---+                       +---+
            ys | o |                    zs | o |
               +-|-+                       +-|-+
                 v                           v
               +---+---+   +---+---+       +---+---+   +---+---+   +---+---+
               | 1 | o-+-->| 2 | o-+-+     | 3 | o-+-->| 4 | o-+-->| 5 | o-+-+
               +---+---+   +---+---+ =     +---+---+   +---+---+   +---+---+ =
                                             ^
    +---+      +---+---+   +---+---+         |
 xs | o-+----> | 1 | o-+-->| 2 | o-+---------+
    +---+      +---+---+   +---+---+
```

Scrap