CIS 120 Final Exam 2 May 2018

SOLUTIONS

1. OCaml: Binary Search Trees (15 points)

Recall the type of generic binary trees in OCaml. The `range` function below operates over these trees. (Note that we have deliberately omitted the type annotations on this function.)

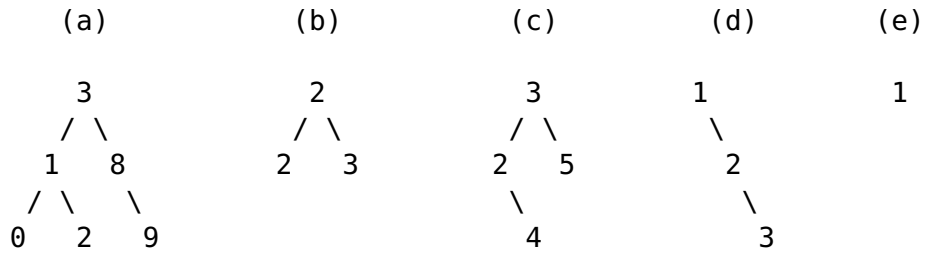
```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

let rec range t low hi =
  begin match t with
  | Empty -> []
  | Node(lt,x,rt) -> if low <= x && x <= hi
    then (range lt low hi) @ [x] @ (range rt low hi)
    else if x < low
     then range rt low hi
     else range lt low hi
  end
```

a. (2 points) Write the type of the `range` function, as you might see it in a `.mli` file or signature. Your type should be generic and allow this function to work with trees containing any type of data.

`val range : ____ 'a tree -> 'a -> 'a -> 'a list` _____

1 point for correct but non generic type.

b. (5 points) Circle all trees below that **satisfy** the binary search tree invariant.



Valid trees are (a), (d) and (e). (b) has two copies of 2, and (c) has 4 to the left of 3.

c. (6 points) What is the result of the `range` function on the trees shown above with the given arguments?

i. `range a 4 5 =` _____`[]`_____

ii. `range b 2 3 =` _____`[2;2;3]`_____
no partial credit for `[2;3]`

iii. `range c 4 5 =` _____`[5]`_____

d. (2 points) Does the `range` function above take advantage of the binary search tree invariant?

☒ yes ☐ no

2. OCaml: Higher-order Functions (10 points)

Recall the definition of the generic `fold` function for lists.

```
let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list) : 'b =
  begin match l with
  | [] -> base
  | hd :: tl -> combine hd (fold combine base tl)
  end
```

Fill in each blank below with one of the following function names so that the definition matches the function description. Each definition is implemented using `fold`, and optionally, some of the other functions. Not all function names will be used.

- `identity` returns the same list as the input
- `append` takes two lists and returns a new list containing all of the elements of the first list followed by all of the elements of the second list
- `elem` determines whether a particular element appears in the list
- `sum` returns the sum of all elements in the list
- `product` returns the product of all elements in the list
- `find` takes a predicate and a list and returns the first element of the list that satisfies the function, or `None` if there is no such element
- `length` returns the length of the input list
- `reverse` returns the list with the elements in reverse order from the input
- `maximum` returns the largest value in the list
- `all` returns whether every value in the list satisfies a given predicate

a. *length*

```
let ____________________ l = fold (fun _ x -> x + 1) 0 l
```

b. *product*

```
let ____________________ l = fold (fun x y -> x * y) 1 l
```

c. *identity*

```
let ____________________ l = fold (fun x y -> x::y) [] l
```

d. *elem*

```
let ____________________ e l = fold (fun x y -> (x = e) || y) false l
```

e. *append*

```
let ____________________ l1 l2 = fold (fun x y -> x :: y) l2 l1
```

3. OCaml: Queue programming (13 points)

Consider the following OCaml function that works with mutable queues. For reference, the definition of mutable queues that we discussed in class appears in Appendix **A**.

```
let rec func (q: 'a queue) (x : 'a) (y : 'a) : unit =
  let rec loop (qno : 'a qnode option) : unit =
    begin match qno with
      | None -> ()
      | Some qn ->
       if qn.v = x then
         let old_next = qn.next in
         (qn.next <- Some {v = y; next = qn.next};
         loop old_next)
       else
         loop qn.next
end
in loop q.head
```

Your job is to figure out what this function does with various imputs. For example, here is one test case that demonstrates the behavior of `func`.

```
;; run_test "func1" (fun () ->
    let q1 = from_list [1;2;3] in
    func q1 5 4;
    to_list q1 = [1;2;3] )
```

Fill in the blank so that the remaining test cases also pass.

a. (3 points)

```
;; run_test "func2" (fun () ->
    let q2 = from_list [1;2;3] in
    func q2 2 4;

    to_list q2 = [1;2;4;3])}
```

b. (3 points)

```
;; run_test "func3" (fun () ->
    let q3 = from_list [1;2;3] in
    func q3 3 4;

    to_list q3 = [1;2;3;4])
```

c. (3 points)

```
;; run_test "func4" (fun () ->
    let q4 = from_list [] in
    func q4 5 4;

    to_list q4 = [])
```

d. (4 points) Put a check next to any queue that is **invalid** at the end of its test. Alternatively, if this function preserves the queue invariant, check *all queues are valid*.

☐ `q1` ☐ `q2` ☒ `q3` ☐ `q4` ☐ *all queues are valid*

The function is "insert after", where func q x y inserts the value y after each x that appears in the queue. It is buggy and does not update the tail reference when inserting at the end of the queue.

4. Java semantics (18 points)

This problem uses the definitions for the classes `IntPair` and `Exam` shown in Appendix **B**.

Consider the following snippet of code that occurs elsewhere in the program (i.e. not as part of any method of `IntPair` or `Exam`).

```
IntPair p1 = ... // unspecified
Exam exam = new Exam();
System.out.println(exam.m(p1));
```

In this problem, you will be given a definition of the method `m` and will need to indicate the output of this code for *any* definition of `p1`. Because we don't know `p1`, there may be several answers.

For example, if the definition of method `m` in class `Exam` were

```
public boolean m(IntPair p1) {
  return (this.p2.equals(p1));
}
```

then you should answer

☒ **true** ☒ **false** ☐ `NullPointerException` ☐ `StackOverflowError`

i.e. you should check both **true** (this is printed when `p1` is **new** `IntPair(1,1)`) and **false** (this is printed when `p1` is, for example, **new** `IntPair(2,2)`). However, you should leave `NullPointerException` and `StackOverflowError` unchecked because there is no definition of `p1` that could cause this definition of `m` to trigger the exception or loop forever.

a. (3 points)

```
public boolean m(IntPair p1) {
  return this.p2 == new IntPair(1,1);
}
```

☐ **true** ☒ **false** ☐ `NullPointerException` ☐ `StackOverflowError`

The == is reference equality, so even though the new IntPair object has the same values as `p2`*, it is not stored at the same location on the heap.*

b. (3 points)

```
public boolean m(IntPair p1) {
  return p1 == this.p2;
}
```

☐ **true** ☒ **false** ☐ `NullPointerException` ☐ `StackOverflowError`

This example also uses reference equality, but with an externally supplied reference. However, because **this**`.p2` *is encapsulated by the* `Exam` *class, references to this object cannot escape, and so therefore cannot be passed in as* `p1` *by external code.*

c. (3 points)

```
public boolean m(IntPair p1) {
  return (this == null);
}
```

☐ **true** ☒ **false** ☐ `NullPointerException` ☐ `StackOverflowError` *In any method call, the* **this** *reference refers to the object that was used to invoke the method. It can never be* `null`.

d. (3 points)

```
public boolean m(IntPair p1) {
  this.p2 = p1;
  return this.p2.equals(p1);
}
```

☒ **true** ☐ **false** ☒ NullPointerException ☐ StackOverflowError

If the supplied IntPair *is not* null*, it is structurally equal to itself. If* null*, will raise an exception with the* .equals *call.*

e. (3 points) (See the definition of method k in Appendix **B**.)

```
public boolean m(IntPair p1) {
 try {
   this.p2 = p1;
   return k();
 } catch (NullPointerException e) {
   return false;
 }
}
```

☒ **true** ☒ **false** ☐ NullPointerException ☐ StackOverflowError

The argument p1 *may or may not be structurally equal to* **new** IntPair(0,0)*. However, if it is* null*, the exception will be caught by the* **try**/catch*.*

f. (3 points) Now suppose that the definition of method m in class Exam is

```
public boolean m(IntPair p1) {
  return k();
}
```

and that we add a new class Exam2

```
public class Exam2 extends Exam {
 @Override
 public boolean k() {
   return m(new IntPair(1,1));
 }
 public Exam2() { super(); }
}
```

and modify the code above to create an instance of Exam2 instead of Exam.

```
IntPair p1 = ... // unspecified
Exam exam = new Exam2();
System.out.println(exam.m(p1));
```

What could be printed by this code?

☐ **true** ☐ **false** ☐ NullPointerException ☒ StackOverflowError

The Exam2 *class overrides* k*, so the call to* m *will call* k*, which will call* m*, etc.*

5. Java Array programming

The following question asks you to implement methods that work with 2-dimensional character arrays, such as `input` shown below.

```
char[][] input = { {'a','b'}, {'c'}, {'d','e','f','g'} };
```

a. (10 points) Fill in the blanks to complete the implementation of the `maxLength` method.

This function should determine the *maximum length* of the inner character arrays. For example, the method invocation `maxLength(input)` should return `4`, as that is the length of the array `{'d','e','f','g'}`.

If the length of the `input` array is `0`, then `maxLength` should return `0`.

The function may throw a `NullPointerException` if `input` or any of the inner arrays are **`null`**.

```
private static int maxLength(char[][] input) {

  int max = _____________0_____________________;


  for (int i = 0; i < ____input.length_________________; i++ ) {


   if ( _____input[i].length > max ____________________________) {


      max = ___input[i].length ___________________;

   }

  }

  return ___max_____________________;

}
```

Each blank is worth two points. Can use $\geq$ instead of $>$ for the condition in the third blank. Getting a blank "mostly right" is one point – i.e. using the wrong variable name. No deduction for `input[i].length()` *in the third or fourth blank.*

b. (10 points) Now complete the implementation of the static method `leftPad`. This function should *modify* the inner arrays of its input so that they all have the same length by adding leading spaces. For example, the call `leftPad(input)` should modify `input` so that it is equivalent to the 2-d array, `modInput` below. Note that each of the inner arrays now has length 4.

```
char[][] modInput = { {' ', ' ', 'a', 'b'},    // two space chars added
                      {' ', ' ', ' ', 'c'},    // three space chars added
                      {'d', 'e', 'f', 'g'} };  // same chars as original
```

As before, if `input` or any of the inner arrays are **`null`**, this method may throw a `NullPointerException`.

```
public static void leftPad(char[][] input) {

    int max = maxLength(input);


    for (int i = 0; i < __________input.length___________ ; i++) {

      char[] old = input[i];


      input[i] = _____new char[max]_____________________ ;


      for (int j = 0; j < _______max___________ ; j++) {


        if (j < _____max - old.length______ ) {

          input[i][j] = ' ';

        } else {

          input[i][j] = __old[j - max + old[i].length]___;

        }

      }

    }
}
```

Two points per blank.

The next two problems concern an abstract type of *invertible maps* implemented in Java.

Before continuing, read the description of invertible maps and the `InvertibleMap` interface in Appendix **C**, and the partial implementation of the `InvertibleTreeMap` class in Appendix **D**. For reference, documentation for the `Map` interface, from the Java Collections library, appears in Appendix **E**.

6. Java Type System (10 points)

Select *all* possible static types for each of the following Java variable definitions. Due to subtyping, there may be more than one type. Check *ill-typed* if the expression does not type check. These definitions refer to the variables `invmap` and `set`, shown below.

```
InvertibleMap<String,Set<String>> invmap = new InvertibleTreeMap<String,Set<String>>();

Set<String> set = new TreeSet<String>();
```

2 points each. 1 point partial credit. Because `InvertibleMap` *extends* `Map`*, methods from both interfaces are available. Furthermore if the expression is well-typed,* `Object` *is always a valid answer.*

a. _____ `a = invmap.put("CIS 120",set);`

☐ `String` ☒ `Set<String>` ☐ `TreeSet<String>`
☒ `Object` ☐ `Map<String,Set<String>>` ☐ *ill-typed*

b. _____ `b = invmap.getKey(set);`

☒ `String` ☐ `Set<String>` ☐ `TreeSet<String>`
☒ `Object` ☐ `Map<String,Set<String>>` ☐ *ill-typed*

c. _____ `c = invmap.getKey(null);`

☒ `String` ☐ `Set<String>` ☐ `TreeSet<String>`
☒ `Object` ☐ `Map<String,Set<String>>` ☐ *ill-typed*

d. _____ `e = invmap.invert();`

☐ `Map<String,Set<String>>` ☐ `InvertibleMap<String,Set<String>>`
☒ `Map<Set<String>,String>` ☒ `InvertibleMap<Set<String>,String>`
☐ `String` ☐ *ill-typed*

e. _____ `f = invmap.values();`

☐ `Set<String>` ☐ `Set<Set<String>>` ☐ `TreeSet<String>`
☐ `Collection<String>` ☒ `Collection<Set<String>>` ☐ *ill-typed*

7. Java implementation

Now implement the `InvertibleMap` interface by completing the `InvertibleTreeMap` class. Part of this implementation is shown in Appendix **D**. The key idea of this implementation is to use two separate `TreeMaps`, a `forward` map storing the mapping from keys to values and a `reverse` map storing the mapping from values to keys. That way, both directions of the map can be accessed efficiently.

This implementation does not allow users to store **`null`** keys or values.

This implementation should maintain the following invariant:

> *For any* `key` *and any* `value` *that are not* **`null`**, *we have* `value.equals(forward.get(key))` *if and only if* `key.equals(reverse.get(value))`.

a. (5 points) Select the **correct** and **most efficient** implementation of the `containsValue` method.

```
/** Returns true if this a key is mapped to the specified non−null value.
    More formally, this method returns true if and only if this map contains
    exactly one mapping to a value v such that value.equals(v).

    @throws NullPointerException if the specified value is null
*/
@Override
public boolean containsValue(Object value) {
   ... // select from below
}
```

Check only one implementation.

☐ `return forward.containsValue(value);`
Correct, but inefficient.

☒ `return reverse.containsKey(value);`
Only efficient answer.

☐ `return forward.containsValue(value) && reverse.containsKey(value);`
Correct, but inefficient. Doesn't take advantage of invariant.

☐
```
for (V v : forward.values()) {
  if (value.equals(v)) {
     return true;
  }
}
return false;
```
Correct, but inefficient.

☐
```
for (V v : reverse.keySet()) {
  if (value.equals(v)) {
     return true;
  }
}
return false;
```
Correct, but inefficient.

b. (5 points) Select the **correct** and **most efficient** implementation of the `invert` method.

```
/** Gets a version of this map where the keys and values are reversed.
    The returned InvertibleMap should share the same underlying data as this data structure.
    (In other words, modifications to the returned invertible map should also affect this map.)
 */
@Override
public InvertibleMap<V, K> invert() {
    // select from below
}
```

Check only one implementation.

☐
```
TreeMap<K,V> temp = this.forward;
this.forward = this.reverse;
this.reverse = temp;
return this;
```
2 points. This code does not type check. `forward` *and* `reverse` *have different types, but it captures the idea of shar*

☒
```
return new InvertibleTreeMap<V,K>(this.reverse, this.forward);
```
5 points.

☐
```
InvertibleTreeMap<V,K> inverted = new InvertibleTreeMap<V,K>();
for (K key : forward.keySet() ) {
  inverted.put (key, forward.get(key));
}
return inverted;
```
0 points. Doesn't type check.

☐
```
InvertibleTreeMap<V,K> inverted = new InvertibleTreeMap<V,K>();
for (K key : forward.keySet() ) {
  inverted.put (forward.get(key),key);
}
return inverted;
```
2 points. This code makes a copy of the data structures and does not share state.

☐
```
TreeMap<V,K> newForward = new TreeMap<V,K>();
TreeMap<K,V> newReverse = new TreeMap<K,V>();
return new InvertibleTreeMap<V,K>(newForward, newReverse);
```
0 points. Returned map is empty.

c. (12 points) Implement the `put` operation.

Be sure to read the comment before the method carefully. Also make sure that your implementation preserves the invariant of this data structure described on page 10.

```
/* Associate the given value with the given key.
 * Any prior associations with either the key or the value
 * are removed from the map.
 *
 * @returns any previous value associated with that key
 * @throws IllegalArgumentException if either key or
 *   value is null. */
@Override
public V put(K key, V value) {
  if (key == null || value == null) {
    throw new IllegalArgumentException();
  }
  V v1 = forward.put(key, value);
  if (v1 != null) {
    reverse.remove(v1);
  }
  K k1 = reverse.put(value, key);
  if (k1 != null) {
    forward.remove(k1);
  }
  return v1;
}
```

- *2 points, put key and value into forward map*
- *2 points, put key and value into reverse map*
- *2 points, remove prior value from reverse map*
- *2 points, remove prior key from forward map*
- *2 points, test inputs for null*
- *2 points, correct return value*

8. Java: Swing and Inner Classes (12 points)

This question refers to the Java code for a variant of the Swing `LightBulb` program we saw in class. You can find the code in Appendix **F**.

a. (3 points) Which line of code defines an anonymous inner class? (Choose one.)

☐ line 2 ☐ line 7 ☐ line 21 ☒ line 32 ☐ line 45

b. (3 points) Suppose we add the following code at line 39.

```
JButton button2 = new JButton("On/Off2");
panel.add(button2);
```

How would this change affect the program behavior? (Choose one.)

☐ The program would fail to compile (there is a type error).

☐ The program would compile, but fail with an exception when run.

☒ The program would compile, and when run would have two buttons. The new button does nothing when clicked.

☐ The program would compile, and when run would have two buttons. The new button flips the bulb state of the *same* lightbulb as the original button.

☐ The program would compile, and when run would have two buttons. The new button flips the bulb state of a *new* lightbulb, which is not displayed.

c. (3 points) Note that none of the `OnOff` code directly invokes the `paintComponent` method of the `LightBulb` class. Which of the following explanations best describes why that is unnecessary? (Choose one.)

☐ The `Lightbulb` `paintComponent` method is called as a static proxy for a `JComponent` reference delegate object.

☒ The `Lightbulb` `paintComponent` method is called via dynamic dispatch from somewhere inside the Swing library, where the `bulb` object is treated as a `JComponent`.

☐ The `Lightbulb` `paintComponent` method is invoked by using `instanceOf` and a type cast operation from somewhere inside the Swing library.

☐ The `@Override` directive (on line 6) causes the `Lightbulb` class to overwrite the `JComponent`'s class table entry for `paintComponent` with its own code.

d. (3 points) Now consider the object created by the **`new`** keyword on line 32.

True ☒ False ☐ The *static* type of the argument to `button.addActionListener` is `ActionListener`.

True ☐ False ☒ The *dynamic* class of this object is `ActionListener`.

True ☒ False ☐ This object must store a reference for `bulb` on the heap.

(Use this page for scratch work, but no answers here.)

Appendix

Do not write answers in this portion of the exam. (But feel free to use it as scratch paper.)

Do not open until the exam begins.

A OCaml queues

```
type 'a qnode = { v : 'a; mutable next : 'a qnode option; }

type 'a queue = {
  mutable head : 'a qnode option;
  mutable tail : 'a qnode option;
}

let create () : 'a queue =
  { head = None; tail = None }

let is_empty (q:'a queue) : bool =
  q.head = None

let enq (x:'a) (q:'a queue) : unit =
  let newnode_opt = Some { v = x; next = None} in
  begin match q.tail with
    | None -> q.head <- newnode_opt;
      q.tail <- newnode_opt
    | Some qn2 ->
      qn2.next <- newnode_opt;
      q.tail <- newnode_opt
  end

let deq (q:'a queue) : 'a =
  begin match q.head with
    | None -> failwith "error: empty queue"
    | Some qn ->
       q.head <- qn.next;
       (if qn.next = None then q.tail <- None);
       qn.v
  end

let to_list (q : 'a queue) : 'a list =
  let rec loop (qn : 'a qnode option) (acc : 'a list) : 'a list =
    begin match qn with
      | None -> List.rev acc
      | Some qn1 -> loop qn1.next (qn1.v :: acc)
    end in
  loop q.head []

let from_list (xs : 'a list) =
   let q = create () in
   List.iter (fun x -> enq x q) xs;
   q
```

B Appendix: Java Semantics

```
class IntPair {
  // private data
  private int i1;
  private int i2;

  // constructor
  public IntPair (int i1, int i2) { this.i1 = i1; this.i2 = i2; }

  // This code is correct and overrides the equality method for the IntPair class,
  // implementing structural equality as discussed in class. Note that even
  // though the fields i1 and i2 are private, Java allows access in methods of the same class.
  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    IntPair other = (IntPair) obj;
    if (i1 != other.i1) {
      return false;
    }
    if (i2 != other.i2) {
      return false;
    }
    return true;
  }

}

public class Exam {
  // private data
  private IntPair p2;

  // constructor
  public Exam() { this.p2 = new IntPair(1,1); }


  public boolean k() {
    return this.p2.equals(new IntPair(0,0));
  }

  public boolean m(IntPair p1) {
    // see options in problem ...
  }

}
```

C `InvertibleMap` interface

An *invertible map* is a variant of a finite map that allows bidirectional lookup between keys and values. Not only can you ask for the value associated with a particular key, but you can also efficiently ask for the key associated with a particular value.

Invertible maps have a one-to-one relationship between keys and values. For any value, there is exactly one key that maps to that value. This value can be accessed with the `getKey` method. Furthermore, each direction of the invertible map can be used as a standard map, via the `invert` method.

```
public interface InvertibleMap<K,V> extends Map<K,V> {

  /** Access the key associated with a particular value. */
  public K getKey(Object value);

  /** Gets a version of this map where the keys and values are reversed.
      The returned InvertibleMap should be a view of this data structure and share the same underlying data.
      (In other words, modifications to the returned invertible map should also affect this map.)
   */
  public InvertibleMap<V,K> invert();

}
```

D `InvertibleTreeMap` implementation (excerpt)

```
public class InvertibleTreeMap<K,V> implements InvertibleMap<K,V> {

  // invariant: forward and reverse are not null
  // invariant: if v = forward.get(k) then k = reverse.get(v)
  private final TreeMap<K,V> forward;
  private final TreeMap<V,K> reverse;

  /** Constructor, creates an empty map */
  public InvertibleTreeMap() {
    this.forward = new TreeMap<K,V>();
    this.reverse = new TreeMap<V,K>();
  }

  /** internal constructor: f and r must not be null */
  private InvertibleTreeMap(TreeMap<K,V>f , TreeMap<V,K> r) {
    this.forward = f;
    this.reverse = r;
  }

  /** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
      @throws NullPointerException if key is null */
  @Override
  public V get(Object key) {
    return forward.get(key);
  }

  ...
}
```

E Excerpt from the Java Collections Framework (Map and TreeMap)

```
interface Map<K,V> {

  public 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. (There can be at most one such mapping.)
  // @throws NullPointerException if this map does not support null keys

  public V put(K key, V value)
  // Associates the specified value with the specified key in this map
  // Returns the value to which this map previously associated the
  // key, or null if the map contained no mapping for the key.
  // @throws NullPointerException if this map does not support null keys

  public V remove(Object key)
  // Removes the mapping for a key from this map if it is present.
  // Returns the value to which this map previously associated the
  // key, or null if the map contained no mapping for the key.
  // @throws NullPointerException if this map does not support null keys

  public boolean containsKey(Object key)
  // Returns true if this map contains a mapping for the specified key.
  // @throws NullPointerException if this map does not support null keys

  public boolean containsValue(Object value)
  // Returns true if this map maps one or more keys to the specified value.
  // @throws NullPointerException if this map does not support null values

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

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

  public Set<K> keySet()
  // Returns a Set view of the keys contained in this map. The set is backed
  // by the map, so changes to the map are reflected in the set, and
  // vice-versa.

  public Collection<V> values()
  // Returns a Collection view of the values contained in this map. The
  // collection is backed by the map, so changes to the map are reflected in
  // the collection, and vice-versa.
}


class TreeMap<K,V> implements Map<K,V> {

  public TreeMap()
  // Constructs a new, empty tree map, using the natural ordering of its keys.
  // This implementation does not support null keys.

  // ... methods specified by interface
}
```

F Java GUI Lightbulb Program

This code is for use with problem 8. Appendix **G** describes relevant parts of the Swing libraries.

```
1  // imports omitted to save space (this code compiles)
2  class LightBulb extends JComponent {
3        private boolean isOn = false;
4        public void flip() { isOn = !isOn; }
5
6        @Override
7        public void paintComponent(Graphics gc) {
8              if (isOn) {
9                    gc.setColor(Color.YELLOW);
10             } else {
11                   gc.setColor(Color.BLACK);
12             }
13             gc.fillRect(0, 0, 100, 100);
14       }
15
16       @Override
17       public Dimension getPreferredSize() { return new Dimension(100,100); }
18 }
19
20 class OnOff implements Runnable {
21   public void run() {
22      JFrame frame = new JFrame("On/Off Switch");
23      JPanel panel = new JPanel();
24      frame.add(panel);
25
26      JButton button = new JButton("On/Off");
27      panel.add(button);
28
29      final LightBulb bulb = new LightBulb();
30      panel.add(bulb);
31
32      button.addActionListener(new ActionListener() {
33             @Override
34             public void actionPerformed(ActionEvent e) {
35                   bulb.flip();
36                   bulb.repaint();
37             }
38      });
39
40      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
41      frame.pack();
42      frame.setVisible(true);
43   }
44   public static void main(String[] args) {
45      SwingUtilities.invokeLater(new OnOff());
46   }
47 }
```

G Relevant Excerpt from the Swing Library

```
public class Component {
   public void repaint()
}
public class Container extends Component {
  public Component add(Component comp)
   // Appends the specified component to the end of this container.
}
public class JFrame extends Component {
 public void pack()
  // Causes this Window to be sized to fit the preferred size and layouts of its subcomponents.

}

public class JComponent extends Container {
  protected void paintComponent(Graphics g)
}
public class JPanel extends JComponent {
  JPanel()
   // Creates a new JPanel with a double buffer and a flow layout.
}

class JPanel extends JComponent {
  public JPanel()
  // Constructor: Creates a new JPanel with a double buffer and a flow layout.

  public void add(JComponent c)
  // Appends the specified component to the end of this container.

  public void setLayout(LayoutManager mgr)
  // Sets the layout manager for this container.
}

class JButton extends JComponent {

   public JButton(String text)
   // Constructor: Creates a button with text.

   public void addActionListener(ActionListener l)
   // Adds an ActionListener to the button.
}

interface ActionListener {
   public void actionPerformed(ActionEvent e)
   // Invoked when an action occurs
}
```