CIS 1200 Final Exam May 9, 2023

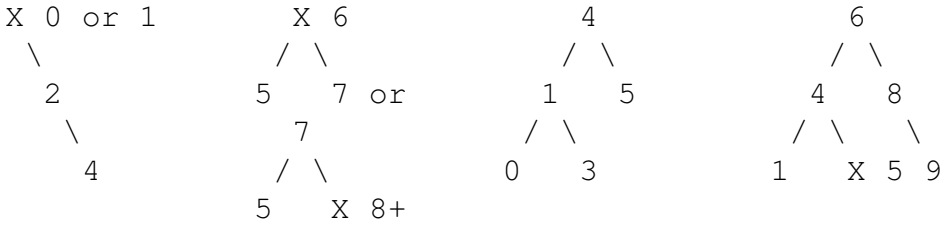
Steve Zdancewic instructor

SOLUTIONS

OCaml Concepts (27 points total)

Appendix A contains the definitions of several OCaml types and functions that should be familiar from class: `'a tree`, `'a ref`, `transform`, and `fold`.

Binary Search Trees (8 points) For each tree below, check the box indicating whether it satisfies the *binary search tree* invariants. **NOTE:** If it does not, cross out and replace *one* node value such that the resulting **`int`** `tree` *does* satisfy the invariants (the result does not need to have the same nodes).



☐ is a BST ☒ is not BST	☐ is a BST ☒ is not BST	☒ is a BST ☐ is not BST	☐ is a BST ☒ is not BST

Higher-order Functions (8 points)

Consider the following four functions defined using `transform` or `fold`.

```
let hof1 = transform (fun x -> insert x 120)
let hof2 = fold (fun x acc -> insert acc x) Empty
let hof3 = fold (fun x acc -> (insert x 120) :: acc) []
let hof4 = fold (fun x acc -> insert x 120) Empty
```

Match each of the functions to one of the following English descriptions of its behavior. (choose one option for each function; a choice may be used by more than one function)

(A) Returns a list of trees obtained by adding the value `120` to every tree in a given list.

(B) Returns either `Empty` or the tree obtained by adding `120` to the first tree in a given list.

(C) Returns a (binary search) tree containing the elements of a provided list.

(D) Returns a list of trees obtained by adding the value `120` to `Empty` for every tree in a given list.

- `hof1` is described by: ☒ A ☐ B ☐ C ☐ D
- `hof2` is described by: ☐ A ☐ B ☒ C ☐ D
- `hof3` is described by: ☒ A ☐ B ☐ C ☐ D
- `hof4` is described by: ☐ A ☒ B ☐ C ☐ D

Aliasing and mutable state Recall that OCaml supports mutable state via type declarations such as for the `'a ref` type shown below (and in Appendix A). Consider the following well-typed program:

```
type 'a ref = { mutable contents : 'a }

let x = { contents = { contents = "1200" } }
let y = { contents = "PENN" }
;; x.contents <- y
let z = y
;; y.contents <- "RULES"
let w = y.contents

;; print_endline ("x.contents.contents = " ^ x.contents.contents)
;; print_endline ("y.contents = " ^ y.contents)
;; print_endline ("z.contents = " ^ z.contents)
;; print_endline ("w = " ^ w)
```

(a) (4 points) What is the type of each variable in the above code? (choose one each)

• `x :`	☐ `string`	☐ `string ref`	☐ `unit ref`	☒ `(string ref) ref`
• `y :`	☐ `string`	☒ `string ref`	☐ `unit ref`	☐ `(string ref) ref`
• `z :`	☐ `string`	☒ `string ref`	☐ `unit ref`	☐ `(string ref) ref`
• `w :`	☒ `string`	☐ `string ref`	☐ `unit ref`	☐ `(string ref) ref`

(b) (4 points) Fill in the blanks below to show what gets printed when this program is run. Fill each slot with one of the strings `1200`, `PENN`, or `RULES`

```
x.contents.contents = RULES

y.contents = RULES

z.contents = RULES

w = RULES
```

(c) (3 points) After running the program above, which values will *alias* `y`? (mark all that apply)

☐ `x`	☒ `z`
☒ `x.contents`	☐ `z.contents`
☐ `x.contents.contents`	☐ `w`

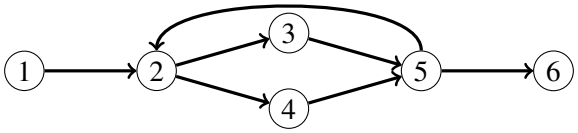
Java Programming

The next several questions refer to the Java code found in Appendix C. Some of the questions test your understanding of Java concepts; other questions will ask you to follow our design process to implement (parts of) a collection datatype for *graphs*. You may find the (excerpt of) the Java Documentation found in Appendix B to be useful.

Step 1: Understand the problem (4 points total)

Besides the *set* and *finite map* collections that we have studied in class, another frequently used collection type is the *graph*. A *graph* contains a set of *nodes* along with a collection of (directed) *edges*, each of which connects a *source* node to a *target* node. We consider graphs where there is at most one edge between any pair of nodes.

We draw nodes as labeled circles (n) and edges as arrows pointing from the source to the target (s)⟶(t). For example, the diagram below depicts a graph whose nodes are the integers 1 through 6, and whose edges are indicated by the seven arrows.



As with other collections, there are many operations that we might want our graphs to support. In addition to methods to add nodes and edges to the graph, we focus on just two other operations. First, determining the *neighbors* of a node, which is just the set of nodes reachable via an edge. For instance, in the graph above, the neighbors of (2) are (3) and (4), but node (6) has no neighbors. Second, we can determine whether there is a *path* from some node (s) to a node (t) that follows the edges in the graph. For instance, in the example graph above, there are paths (1)⟶(2)⟶(4)⟶(5) and (5)⟶(2)⟶(3) but there is no path from node (3) to node (1). Note that there is always a (trivial) path from any node to itself and that *a path might visit the same node multiple times*.

(a) (2 points) In the example graph above, which nodes are *neighbors* of node (5)? (mark all that apply)

☐ (1) ☒ (2) ☐ (3) ☐ (4) ☐ (5) ☒ (6)

(b) (2 points) How many paths are there from node (4) to node (3)? (choose one)

☐ 0

☐ 1

☐ 2

☒ there are infinitely many paths due to the cycle in the graph

Step 2: Design the interface (14 points total)

Just as with Java's `Set<E>` and `Map<K,V>` interfaces, which are generic in the data they store, we will make the interface `Graph<Node>` polymorphic in the type `Node`. The `Graph<Node>` interface supports `add` and `contains` operations with the same specification of those methods in `Set`; it also supports three new graph-specific methods: `addEdge`, `neighbors`, and `hasPath`. Code defining the interface is shown in Appendix C.1.

Java concepts: (12 points) With respect to `Graph<Node>`, indicate whether the following statements are true or false; if false give a brief justification.

1. According to the method signature on line 25, `addEdge` might throw an `IOException`.

   ☐ True
   ☒ False because: *Ans:* `IOException` must be declared using **`throws`**

2. According to the method signature on line 25, `addEdge` might throw a `NullPointerException`

   ☒ True
   ☐ False because: *Ans:*

3. The following snippet of code will compile successfully (but may produce warnings):

```
Graph<Integer> g = null;
g.contains("CIS1200");
```

   ☒ True
   ☐ False because: *Ans: Note that the contains method takes an argument of type* `Object`.

4. The following snippet of code will compile successfully (but may produce warnings):

```
Graph<Integer> g = new Graph<>();
g.contains(3);
```

   ☐ True
   ☒ False because: *Ans:* You can't create an instance of an interface.

Design question: (2 points) Suppose that, rather than `neighbors`, the `Graph<Node>` interface provides a method **`boolean`** `hasEdge(Node src, Node tgt)`, which returns **`true`** if there is an edge from `src` to `tgt` (and throws `NoSuchElementException` if `src` is not a node in the graph).

1. Consider implementing `hasEdge` using `neighbors`: (choose one)

   ☐ It is not possible to implement `hasEdge` using `neighbors`.

   ☐ `g.contains(src)&& g.neighbors(tgt)` implements `g.hasEdge(src,tgt)`

   ☒ `g.neighbors(src).contains(tgt)` implements `g.hasEdge(src,tgt)`

   ☐ `g.neighbors(tgt).contains(src)` implements `g.hasEdge(src,tgt)`

Step 3: Write test code for `Graph` (12 points total) Even before we have code that implements the `Graph<Node>` interface, we can write test cases that check our examples and properties. These test cases can use *subtype polymorphism* to implement tests that work for any implementation: we will assume that each such generic test is provided a instance of a graph object freshly created by **`new`**. For instance, in **Step 4** (later in the exam) you will work with `TreeGraph`. We will assume that a test for `TreeGraph` is called via `test(new TreeGraph())`.

(a) (4 points) Consider the following test case:

```
1   private void testNoSuchSource(Graph<Integer> g) {
2       assertFalse(g.contains(0));
3       assertThrows(NoSuchElementException.class, () -> g.neighbors(0));
4   }
```

Which of the following statements are true? (mark all that apply)

☒ This test case will succeed only if `g.contains(0)` returns **`false`**.

☐ The syntax `NoSuchElementException.class` (line 3) creates an instance of an *anonymous inner class*.

☒ The syntax `() -> g.neighbors(0)` (line 3) creates an instance of an *anonymous inner class*.

☐ The following test case is equivalent to the one above.

```
private void testNoSuchSourceAlternate(Graph<Integer> g) {
    assertFalse(g.contains(0));
    try {
        g.neighbors(0);
    } catch (NoSuchElementException e) { fail(); }
}
```

(b) (2 points) Complete the following test case so that all assertions succeed. It is based on the example graph pictured earlier:

```
private void testNeighbors(Graph<Integer> g) {
    g.addEdge(1,2);  g.addEdge(2,3);  g.addEdge(2,4);  g.addEdge(3,5);
    g.addEdge(4,5);  g.addEdge(5,6);  g.addEdge(5,2);

    TreeSet<Integer> expected1 = new TreeSet<>();

    expected1.add(___2_____);
    assertEquals(expected1, g.neighbors(1));

    TreeSet<Integer> expected2 = new TreeSet<>();
    expected2.add(3);
    expected2.add(____4_______);
    assertEquals(expected2, g.neighbors(___2_______));

    TreeSet<Integer> expected3 = new TreeSet<>();

    assertEquals(expected3, g.neighbors(____6______));
}
```

(c) (6 points) Briefly explain (in English and/or pseudocode) how to write a test case that can fail if the `neighbors` method does not properly encapsulate some state associated with the graph implementation.

> *Answer:* To test proper encapsulation, we must (1) obtain the set of nodes returned by a call to `neighbors`, (2) modify that set, and (3) check whether the modification affects the results of another operation on the set. One code example (of many possible ones) is:

```
private void testEncapsulation(Graph<Integer> g) {
    g.addEdge(1,2);
    Set<Integer> nbrs = g.neighbors(1);
    assertFalse(nbrs.contains(3));
    nbrs.add(3);
    Set<Integer> nbrs2 = g.neighbors(1);
    assertFalse(nbrs2.contains(3));
}
```

Step 4: Implement it As with the Java Collections library, we might have different implementations of the `Graph` interface that use different internal representations. We will follow the design there by using an **`abstract class`** to implement the `hasPath` algorithm once so that different representations of the graph abstract type can share that code.

Java concepts (16 points total) These questions are about the implementation of the `AbstractGraph` code in Appendix C.2. (Note: you should be able to answer these questions *without* understanding the details of the `hasPath` search algorithm; they are just about Java concepts.)

1. (4 points) Which of the following are *supertypes* of `AbstractGraph<Integer>`? (mark all that apply)

   ☒ `AbstractGraph<Integer>` ☐ `AbstractGraph<Object>`
   ☐ `AbstractSet<Integer>` ☐ `Set<Node>`
   ☐ `AbstractGraph<Node>` ☒ `Object`

2. (4 points) Suppose a well-typed program declares a variable `Graph<Integer> g = (* omitted *);`. Which of the following statements are true? (mark all that apply)

   ☒ The *static type* of `g` is `Graph<Integer>`.

   ☐ It is possible for the *dynamic class* associated with `g` to be `AbstractGraph<Integer>`

   ☒ It is possible for the *dynamic class* associated with `g` to be a *subtype* of `AbstractGraph<Integer>`

   ☒ In code after this declaration it is possible for the expression `g.equals(g)` to throw an exception.

3. (4 points) Note that there is a comment in the documentation for `AbstractGraph` indicating that the implementation assumes that `Node` implements the `Comparable<Node>` interface. Which of the following best explains why? (choose one)

   ☐ The implementation of **`this`**`.contains`, as used on line 11, requires `compareTo` in its implementation.

   ☐ The method `toSearch.removeFirst()`, as used on line 24, needs `compareTo` to find the smallest node to remove from the list.

   ☒ The implementation of `hasPath` uses a `TreeSet<Node>` to store the `alreadyVisited` nodes, as seen on lines 16, 25, and 30, and `TreeSet` requires its elements to support `compareTo`.

   ☐ The method `current.equals(tgt)`, as used on line 27, requires `compareTo` in its implementation.

4. (4 points) Which of the following best describes what would happen if we removed the use of **`this`** from the conditional guard on line 11, i.e., so it reads:

   ```
   if (!contains(src)|| !contains(tgt)){
   ```

   (choose one)

   ☐ The `hasPath` method would *always* throw a `NoSuchElementException`.

   ☐ The `hasPath` method would *never* throw a `NoSuchElementException`.

   ☐ The `hasPath` method would always go into an infinite loop.

   ☒ The behavior of the `hasPath` method would be unaffected.

Design question Now we will implement *two* different instances of the `Graph<Integer>` interface. Both inherit from `AbstractGraph<Integer>` and provide the missing `Graph<Integer>` methods. (Note that, for these types, the nodes are specialized to be `Integer` objects.)

(no more questions on this page)

Implementation 1: `ArrayGraph` (14 points total) The first implementation represents the nodes and edges of the graph using arrays of **`boolean`**s (sometimes called the "adjacency matrix"). Appendix C.3 contains code for this class.

The following are the *representation invariants* embodied by this code.

INV1 `nodes[n] == true` if and only if `n` is a node in the graph

INV2 `edges[src][tgt] == true` if and only if `src` and `tgt` are nodes in the graph and there is an edge from `src` to `tgt`

(a) (2 points) Which of the following best explains why the constructor (line 10) *establishes* these invariants? (choose one)

- ☐ The graph starts out with `maxNodes` nodes, and the default initializer for **`boolean`** is **`true`**.
- ☐ The graph starts out with `maxNodes` nodes, and the default initializer for **`boolean`** is **`false`**.
- ☐ The graph starts out empty, and the default initializer for **`boolean`** is **`true`**.
- ☒ The graph starts out empty, and the default initializer for **`boolean`** is **`false`**.

(b) (4 points) The `add` method is supposed to add a new node to the graph.

- Which line of code *relies on* INV1? (choose one):

  ☐ line 22 ☒ line 23 ☐ line 24 ☐ line 25

- Which line of code *establishes* INV1? (choose one):

  ☐ line 22 ☐ line 23 ☒ line 24 ☐ line 25

(c) (2 points) Which invariant would break if we delete line 37 from `addEdge`? (choose one)

☐ INV1 breaks ☒ INV2 breaks ☐ neither breaks ☐ both break

(d) (6 points) Complete the implementation of `neighbors`. (The code is not very long.)

```
@Override
public Set<Integer> neighbors(Integer src) {
    if (!this.contains(src)) { throw new NoSuchElementException(); }

    Set<Integer> nbrs = new TreeSet<>();
    for(int tgt=0; tgt < nodes.length; tgt++) {
        if (edges[src][tgt]) nbrs.add(tgt);
    }
    return nbrs;
}
```

Implementation 2: `TreeGraph` (17 points) The second implementation represents the nodes and edges of the graph using the Java `TreeMap` and/or `TreeSet` collection(s). From the options below, choose appropriate *representation type(s)* or mark "not needed" if you don't need that field. Then write down the invariant and complete the missing parts of the `TreeGraph` following that plan.

```
class TreeGraph extends AbstractGraph<Integer> {

    private final Map<Integer,Set<Integer>> edges;

    /* INVARIANT:
     * s is a node exactly when:        edges.containsKey(s)
     *
     * s -> t is an edge exactly when:  edges.get(s).contains(t)
     */

    public TreeGraph() { this.edges = new TreeMap<>(); }

    @Override
    public boolean add(Integer node) {
        if (!this.contains(node)) {
            edges.put(node, new TreeSet<>());
            return true;
        }
        return false;
    }
    @Override
    public boolean contains(Object o) {
        return edges.containsKey(o);
    }
    @Override
    public void addEdge(Integer src, Integer tgt) {
        this.add(src);
        this.add(tgt);
        Set<Integer> neighbors = edges.get(src);
        neighbors.add(tgt);
    }
    @Override
    public Set<Integer> neighbors(Integer src) {
        if (!this.contains(src)) { throw new NoSuchElementException(); }
        Set<Integer> nbrs = new TreeSet<>();
        for(Integer tgt : edges.get(src)) {
            nbrs.add(tgt);
        }
        return nbrs;
    }
}
```

Using a `Graph` (16 points total) Appendix C.4 contains a program that reads the edges of a graph from a file `example.txt` and then prints out, for each node `src` in the graph, a list of nodes that can be reached via a path from `src`. The file format is simple: nodes are numbers and each line of the file is of the form `s -> t`, representing an edge from `s` to `t` in the graph. Below you can see the sample `example.txt` associated with the example graph, along with the output printed to the console:

```
example.txt:              console output:
1 -> 2                    1 ==> 1 2 3 4 5 6
2 -> 3                    2 ==> 2 3 4 5 6
2 -> 4                    3 ==> 2 3 4 5 6
3 -> 5                    4 ==> 2 3 4 5 6
4 -> 5                    5 ==> 2 3 4 5 6
5 -> 6                    6 ==> 6
5 -> 2
```

Java Concepts (a) (2 points) Suppose that we change line 25 of `GraphApp` to instead use the **`this`** keyword:

```
Graph<Integer> g = this.readGraph(new FileReader(filename));
```

What would be the result? (choose one)

- ☐ The program would compile successfully and its behavior would be unchanged.
- ☐ The program would compile successfully but it would throw an exception when run.
- ☒ The program would not compile because `main` is declared as **`static`**.
- ☐ The program would not compile because `readGraph` is declared as **`static`**.

Java Concepts (b) (2 points) Recall that you used the `BufferedReader` as part of your `TwitterBot` homework. Which of the following are true properties of the `BufferedReader` class. (Note: we have intentionally *not* provided JavaDocs for `BufferedReader`.) (mark all that apply)

- ☒ `BufferedReader` is more efficient than just using `FileReader` to read individual characters from the input.
- ☐ The `BufferedReader` methods do not throw `IOException`s.
- ☒ The `BufferedReader` `readLine` method provides the ability to read a whole line of input at once as a `String`.
- ☒ A `BufferedReader` constructor can accept any `Reader` as an input.

Design Question (a) (5 points) This program works, but could be made both simpler and more general. Suggest a change you would make to the `Graph<Node>` interface and briefly describe (in English and/or pseudocode) how it would let your simplify the code on lines 26–36 of `GraphApp`. **Note:** Your change should be suitable for inclusion in a Java Collections library class, i.e., it should not simply contain the code of from `GraphApp` and it should not do I/O.

- Modify the `Graph<Node>` interface to add a method `Set<Node> getNodes()` and: rewrite the lines 26–36 as:

```
for(Integer src : g.getNodes()) {
  System.out.print(src + " ==>");
  for(Integer tgt : g.getNodes()) {
    if (g.hasPath(src, tgt)) System.out.print(" " + tgt);
  }
  System.out.println();
}
```

- Modify the `Graph<Node>` interface to `extend Iterable<Node>` and rewrite the lines 26–36 as:

```
for(Integer src : g) {
  System.out.print(src + " ==>");
  for(Integer tgt : g) {
    if (g.hasPath(src, tgt)) System.out.print(" " + tgt);
  }
  System.out.println();
}
```

Design Question (b) (3 points) As it is currently written, `readGraph` *declares* the `IOException` and `main` uses **`try`**/**`catch`** to handle it, which is a bit awkward. Which of the following changes would lead to a cleaner design with the same functionality? (choose one)

- ☐ Remove the **`try`**/**`catch`** code on lines 24/37–41 and add **`throws`** `IOException` on line 21.
- ☒ Move the **`try`**/**`catch`** code from lines 24/37–41 to surround lines 9-16, modify `readGraph` to take `String filename`, and call the `FileReader` constructor on line 9.
- ☐ Change `readGraph` to take a `BufferedReader b`, delete the **`throws`** `IOException` from line 7, delete line 9, and call the `BufferedReader` constructor on line 25.

Design question (c) (4 points) Our implementations of `Graph<Integer>` have not said anything about the `equals` method. Would *overriding* `equals` be justified in this case? Briefly explain why or why not.

Implementations of `Graph<Integer>` ☒ should ☐ should not override `equals`, because:

Answer: Two graphs should be considered structurally equivalent if they contain the same set of nodes and those nodes are connected by in the same way by the edges in each graph. Java's `Set` type uses a similar kind of structural comparison, and this would be consistent.

Scratch Space

*Use this page for work that you do not want us to grade. If you run out of space elsewhere in the exam and you **do** want to put something here that we should grade, make sure to put a clear note in the normal answer space for the problem in question.*

CIS1200 Final 2023 Spring
Appendices

A OCaml Code

Binary Trees

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

(* Inserts n into the binary search tree t *)
let rec insert (t: 'a tree) (n: 'a) : 'a tree =
  begin match t with
  | Empty -> Node(Empty, n, Empty)
  | Node(lt, x, rt) ->
     if x = n then t
     else if n < x then Node(insert lt n, x, rt)
     else Node(lt, x, insert rt n)
  end
```

Reference Types

```
type 'a ref = { mutable contents : 'a }
```

Recall that the OCaml syntax `r.contents <- e` mutates the `contents` field of `r` to be `e`.

Higher-order Functions: Transform and Fold

```
let rec transform (f: 'a -> 'b) (xs: 'a list): 'b list =
  begin match xs with
  | [] -> []
  | h::tl -> f h :: transform f tl
  end

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

B JavaDocs

class Integer **implements** Comparable<Integer>

The Integer class wraps a value of the primitive type **int** in an object. An object of type Integer contains a single field whose type is **int**. The Java compiler will transparently insert calls intValue and valueOf to convert between **int** literals and Integer objects as needed.

---

int intValue()

Returns the value of this Integer as an **int**.

---

static Integer parseInt(String s)

- **Returns:** the integer value represented by the argument s in decimal.
- **Throws:** NumberFormatException - if the string does not contain a parsable integer.

---

static Integer valueOf(**int** i)

Returns the Integer object corresponding to **int** i.

---

int compareTo(Integer anotherInteger)

Compares two integers numerically. x.compareTo(y) returns x - y.

interface `Iterator<E>`

---

boolean `hasNext()`

Returns **true** if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)

- **Returns:** **true** if the iteration has more elements

`E next()`

Returns the next element in the iteration.

- **Returns:** the next element in the iteration
- **Throws:** `NoSuchElementException` - if the iteration has no more elements

interface `Set<E>` **extends** `Collection<E>, Iterable<E>`
Type Parameters:

- `E` - the type of elements in this set

---

boolean `add(E e)`

Adds the specified element to this set if it is not already present (optional operation).

- **Returns:** **`true`** if this set did not already contain the specified element
- **Throws:**

  `ClassCastException` - if the class of the specified element prevents it from being added to this set

  `IllegalArgumentException` - if some property of the specified element prevents it from being added to this set

---

boolean `contains(Object o)`

- **Returns:** **`true`** if this set contains the specified element

---

boolean `isEmpty()`

- **Returns:** **`true`** if this set contains no elements.

`Iterator<E> iterator()`

- **Returns:** an iterator over the elements in this set

interface `Map<K,V>`
Type Parameters:

- `K` - the type of keys maintained by this map
- `V` - the type of mapped values

---

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.

`V put(K key, V value)`

Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for the key, the old value is replaced by the specified value. (A map `m` is said to contain a mapping for a key `k` if and only if `m.containsKey(k)` would return **`true`**.)

- **Parameters:**

  `key` - key with which the specified value is to be associated

  `value` - value to be associated with the specified key

- **Returns:** the previous value associated with key, or **`null`** if there was no mapping for `key`. (A **`null`** return can also indicate that the map previously associated **`null`** with `key`, if the implementation supports **`null`** values.)

`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. The set supports element removal, which removes the corresponding mapping from the map, via the `Set.remove` operation. It does not support the `Set.add` operation.

C Java Code

C.1 `Graph`

```
1
2  /**
3   * A Graph contains a set of nodes along with additional information about
4   * edges between them.
5   * @param <Node> - the type of nodes in the graph
6   */
7  interface Graph<Node> {
8
9      /* Follows the same specification as Set<Node> */
10     boolean add(Node n);
11
12     /* Follows the same specification as Set<Node> */
13     boolean contains(Object o);
14
15     /**
16      * Adds an edge to the graph from node src to node tgt. The nodes
17      * src and tgt are also added to the graph.
18      * @param src - the source node of the edge
19      * @param tgt - the target node of the edge
20      * @throws ClassCastException - if the class of the src or tgt
21      *   prevents it from being added to this set of nodes
22      * @throws IllegalArgumentException if either src or tgt is not a valid
23      *   node for the graph
24      */
25     void addEdge(Node src, Node tgt);
26
27     /**
28      * Returns the set of target nodes reachable by following edges
29      * with the source src. These are the immediate neighbors of src.
30      * @param src
31      * @return - set of nodes reachable by following edges with source src
32      * @throws - NoSuchElementException if node src is not in the graph
33      */
34     Set<Node> neighbors(Node src);
35
36     /**
37      * Determines whether there is a path from src to tgt following
38      * edges of the graph.
39      * @param src - start node of the path
40      * @param tgt - end node of the path
41      * @return - true if there is a sequence of edges connecting src to tgt
42      * @throws NoSuchElementException if either src or tgt is not a node of
43      *   the graph
44      */
45     boolean hasPath(Node src, Node tgt);
46 }
```

C.2 AbstractGraph

```
1
2  /**
3   *  Provides the hasPath operation implementation for Graph<Node>
4   *
5   *  Assumes: Node implements the Comparable<Node> interface.
6   *  */
7  abstract class AbstractGraph<Node> implements Graph<Node> {
8
9      public boolean hasPath(Node src, Node tgt) {
10         /* Check that the src and tgt are in the graph. */
11         if (!this.contains(src) || !this.contains(tgt)) {
12             throw new NoSuchElementException();
13         }
14
15         /* the set of nodes already visited, used to prevent infinite loops */
16         Set<Node> alreadyVisited = new TreeSet<>();
17
18         /* the list of nodes still to visit */
19         LinkedList<Node> toSearch = new LinkedList<>();
20         toSearch.add(src);
21
22         /* perform breadth-first search through the graph */
23         while(!toSearch.isEmpty()) {
24             Node current = toSearch.removeFirst();
25             alreadyVisited.add(current);
26
27             if (current.equals(tgt)) return true;
28
29             for(Node n : this.neighbors(current)) {
30                 if (!alreadyVisited.contains(n)) {
31                     toSearch.addLast(n);
32                 }
33             }
34         }
35         return false;
36     }
37 }
```

C.3 ArrayGraph

```
1  class ArrayGraph extends AbstractGraph<Integer> {
2
3      private final boolean[] nodes;
4      private final boolean[][] edges;
5
6      /**
7       * Constructs a graph capable of storing at most maxNodes nodes
8       * @param maxNodes - the maximum number of nodes that might be in the graph
9       */
10     public ArrayGraph(int maxNodes) {
11         nodes = new boolean[maxNodes];
12         edges = new boolean[maxNodes][maxNodes];
13     }
14
15     /* Helper method to determine validity of array indices. */
16     private boolean valid(Integer n) {
17         return (0 <= n && n < nodes.length);
18     }
19
20     @Override
21     public boolean add(Integer n) {
22         if (!valid(n)) { throw new IllegalArgumentException(); }
23         if (nodes[n]) { return false; }
24         nodes[n] = true;
25         return true;
26     }
27
28     @Override
29     public boolean contains(Object o) {
30         if (o == null || Integer.class != o.getClass()) return false;
31         Integer n = (Integer) o;
32         return valid(n) && nodes[n];
33     }
34
35     @Override
36     public void addEdge(Integer src, Integer tgt) {
37         add(src);
38         add(tgt);
39         edges[src][tgt] = true;
40     }
41
42     @Override
43     public Set<Integer> neighbors(Integer src) {
44         if (!this.contains(src)) { throw new NoSuchElementException(); }
45
46         Set<Integer> nbrs = new TreeSet<>();
43         /* TODO */
44     }
45 }
```

C.4 GraphApp

```
1
2  /**
3   * Reads a graph from a file and prints out path information
4   */
5  public class GraphApp {
6
7      public static Graph<Integer> readGraph(Reader r) throws IOException {
8          Graph<Integer> g = new TreeGraph();
9          BufferedReader b = new BufferedReader(r);
10
11         for(String s = b.readLine(); s != null; s = b.readLine()) {
12             String[] edge = s.split(" -> ");
13             Integer src = Integer.parseInt(edge[0]);
14             Integer tgt = Integer.parseInt(edge[1]);
15             g.addEdge(src, tgt);
16         }
17
18         return g;
19     }
20
21     public static void main(String[] args) {
22         String filename = "files/example.txt";
23
24         try {
25             Graph<Integer> g = readGraph(new FileReader(filename));
26             for(int src = 0; src < 10; src++) {
27                 if (g.contains(src)) {
28                     System.out.print(src + " ==>");
29                     for (int tgt = 0; tgt < 10; tgt++) {
30                         if (g.contains(tgt)) {
31                             if (g.hasPath(src, tgt)) System.out.print(" " + tgt);
32                         }
33                     }
34                     System.out.println();
35                 }
36             }
37         } catch (FileNotFoundException e) {
38             System.out.println("File " + filename + " not found.");
39         } catch (IOException e) {
40             System.out.println("IO Error");
41         }
42     }
43 }
```