CSE 331
Software Design & Implementation

Autumn 2021

Section 5 – HW5, Rep Invariants, Equals + Hashcode

Administrivia

- HW5 Part 1 due tonight (at 11PM)!
  - **hw5-part1-final** tag
  - Do not include any ADT implementation in this commit/tag

- HW5 part 2 (ADT implementation) due next Thursday.
  - Reminder (1): *No generics for now!*
  - Reminder (2): Be sure to add/commit/push new files in git
  - Reminder (3): Remember to commit and push your code often, even if your assignment isn’t finished yet!

Agenda

- HW5
- Rep Invariant and AF Practice
- Managing an expensive **`checkRep`**
- **`equals`** and **`hashCode`**
- Brief mid-point summary/review

Refresher: Format of script tests

Each script test expressed as text-based script ***foo*.test**

- One command per line, of the form: **Command** *arg*$_1$ *arg*$_2$ …
- Script's output compared against ***foo*.expected**
- Precise details specified in the homework
- Match format ***exactly***, including whitespace and output order!

Command (in *foo*.test)	Output (in *foo*.expected)
CreateGraph *name*	**created graph** *name*
AddNode *graph label*	**added node** *label* **to** *graph*
AddEdge *graph parent child label*	**added edge** *label* **from** *parent* **to** *child* **in** *graph*
ListNodes *graph*	*graph* **contains:** *label*$_{node}$ …
ListChildren *graph parent*	**the children of** *parent* **in** *graph* **are:** *child* **(***label*$_{edge}$**)** …
# *This is comment text …*	**#** *This is comment text …*

Refresher: `example.test`

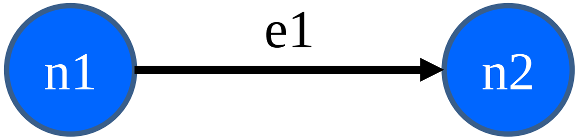
```
# Create a graph
CreateGraph graph1

# Add a pair of nodes
AddNode graph1 n1
AddNode graph1 n2

# Add an edge
AddEdge graph1 n1 n2 e1

# Print all nodes in the graph
ListNodes graph1

# Print all child nodes of n1 with outgoing edge
ListChildren graph1 n1
```



Refresher: **example.expected**

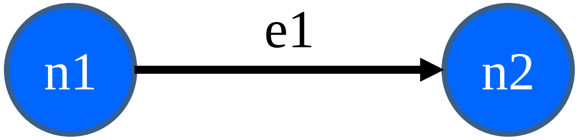
```
# Create a graph
created graph graph1

# Add a pair of nodes
added node n1 to graph1
added node n2 to graph1

# Add an edge
added edge e1 from n1 to n2 in graph1

# Print all nodes in the graph
graph1 contains: n1 n2

# Print all child nodes of n1 with outgoing edge
the children of n1 in graph1 are: n2(e1)
```



Graph Test Driver

- **`GraphTestDriver`** calls a method to “do” each verb
  - **`CreateGraph`**, **`AddNode`**, **`AddEdge`** …
  - One method stub per script command for you to fill with calls to your graph code
- Note: Completed test driver should sort lists before printing for ListNodes and ListChildren
  - Just to ensure predictable, deterministic output
  - Your graph implementation itself should not worry about sorting

Graph Test Driver Output

- The Graph Test Driver is a client of our graph…
  - …but not the only client.
  - Your graph should not be designed to be exclusively used for the test driver.

- ListChildren in the test driver should print out: "**`the children of`** *parent* **`in`** *graph* **`are:`** *child* **`(`**$label_{edge}$**`)`** …"

- This does not mean that you should have a method on your graph called ListChildren that returns this String
  - Because that would make it very hard for other clients to use that don't want this exact format

Sorting with the driver

- **Use the test driver appropriately!**
  - From last slide: “Completed test driver should sort lists before printing.”

- Script test output for hw5 needs to be sorted so we can mechanically check it.

- This means sorted output for tests does ***NOT*** mean sorted internal storage in graph.
  - If sorting behavior is needed, Graph ADT clients (including the test driver) can sort those labels.

In other words…

The Graph ADT in general should ***NOT*** assume that node or edge labels are sorted.

Script Tests vs. Junit Tests

- If you’re able to test a case with script tests, use script tests:
  - i.e. any input/output covered by the script test commands
  - These are Graph agnostic (if you wanted to overhaul your Graph class, you would only need to change your test driver)

- Otherwise, use Junit tests:
  - i.e. bad input, additional methods, …
  - If you want to overhaul the graph class, you would need to change all of the tests

Rep Invariants and AFs

- Let’s do the worksheet!

- In pairs/groups

Expensive **`checkRep`**s

- A complicated rep. invariant can be expensive to check
  - Especially iterating over internal collection(s)
  - For example, examining every edge in a graph

- A slow **`checkRep`** could cause our grading scripts to time-out
  - Can be really useful during testing/deugging, but
  - Need to disable the really slow checks before submitting

- We have a tension between two goals:
  - Thorough, possibly slow checking for development
  - Essential, necessarily fast checking for production/grading

- What to do?

Use a debug flag to tune `checkRep`

- Repeatedly (un)commenting sections of code is a poor solution

- Instead, use a class-level constant as a toggle
  - Ex.: `private static final boolean DEBUG = …;`
    - `false` for only the fast, essential checks
    - `true` for all the slow, thorough checks
  - Real-world code often has several such “debug levels”

```
private void checkRep() {
    assert fast_checks();
    if (DEBUG)
        assert slow_checks();
}
```

The `equals` method (review)

- Specification mandates several properties:
  - *Reflexive*: **`x.equals(x)`** is **`true`**
  - *Symmetric*: **`x.equals(y)`** ⇔ **`y.equals(x)`**
  - *Transitive*: **`x.equals(y)`** ∧ **`y.equals(z)`** ⇒ **`x.equals(z)`**
  - *Consistent*: **`x.equals(y)`** shouldn't change, unless perhaps **`x`** or **`y`** did
  - *Null uniqueness*: **`x.equals(null)`** is **`false`**

- Several notions of equality:
  - *Referential*: literally the same object in memory
  - *Behavioral*: no sequence of operations could tell apart
  - *Observational*: no sequence of <u>observer</u> operations could tell apart

The **`hashCode`** method (review)

- Specification mandates several properties:
  - *Self-consistent*: **`x.hashCode()`** shouldn't change, unless **`x`** did
  - *Equality-consistent*: **`x.equals(y)`** ⇒ **`x.hashCode() == y.hashCode()`**

- Equal objects *must* have the same hash code.
  - Implementations of **`equals`** and **`hashCode`** work together for this
  - If you override **`equals`**, you ***must*** override **`hashCode`** as well

Overriding **`equals`** and **`hashCode`**

- A subclass method overrides a superclass method, when…
  - They have the exact same name
  - They have the exact same argument types

- An overriding method should satisfy the overridden method’s spec.

- Always use **`@override`** tag when overriding **`equals`** and **`hashCode`** (or any other overridden method)

- Note: Method overloading is not the same as overriding
  - Same name but distinguished by different argument types

- Keep these details in mind if you override **`equals`** and **`hashCode`**.

Your turn!

Spend a few minutes on the worksheet problems, then we’ll go over answers.

Topics covered so far

- **Reasoning about code:**
  Hoare logic, forward/backward reasoning, loop invariants, …
- **Specification:**
  JavaDoc, stronger *v*. weaker, satisfaction, substitutability, …
- **Data abstraction:**
  ADT spec./impl., abstraction functions, rep. invariants, …
  - Including **`checkRep`** as covered in lecture/section
- **Testing:**
  unit *v*. system, black-box *v*. clear-box, spec. *v*. impl., …
- **Modularity:**
  (de)composition, cohesion, coupling, open-closed principle, …
- **Object identity:**
  equivalence relation, **`equals`**, **`hashCode`**, …