This page last changed on Jul 30, 2007 by dswood.

Graph - Can Reach

Session Objectives

By the end of this session students should ...

- Understand recursion and be able to implement it
- Know the 3 parts of recursion
    - original question
    - recursive question
    - termination condition

Session Overview

Activity
• Notes on the Graph example • ◦ Airline booking   ◦ Flavors of recursion   ◦ General strategy   ◦ Conclusion • Lab1: Can Reach • Recursion Discussion • Lab2: Finish Can Reach

Session Notes

Notes on the Graph example

Airline booking

To make this slightly more realistic and less textbook, the graph example can be presented as the logic portion of an airline ticket booking system, e.g orbitz, travelocity. Each of the nodes can be named for a city starting with the appropriate letter. As an example:

- Atlanta
- Bloomington
- Charleston
- Denver
- Edmonton
- Fairfax
- Georgetown
- Halifax

Later, when cost is introduced, each of the flights can have an associated integer dollar cost instead of weight, and as long as the rouge from C to E through D costs less than either of the direct flights from C to E, the problem will not change. In addition, this allows a concrete explanation for there being two ways to get from C to E directly - one is a first class flight, the other is coach. An example of the weights

- h->b 87
- b->a 179
- a->f 72
- b->c 402
- c->d 126
- d->e 79
- c->e 749
- c->e 320
- e->b 312

Flavors of recursion

The calculations of recursion can be done two ways. First, "on the way down" with a collecting parameter passed (by reference if that matters in your language) and added to for whatever that means for the particular algorithm. Second, "on the way up" where the answer at each level or recursion is returned from the recursive method and added to. In pseudocode these look like

```
Error formatting macro: code: Traceback (innermost last): File "<string>", line 1, in ? ImportError: no module named pygments

recursiveFunction(collector) {
if (baseCase())

Unknown macro: { collector.add(something) }
else
Unknown macro: { collector.add(calculation()); recursiveFunction(collector) }
}
Error formatting macro: code: Traceback (innermost last): File "<string>", line 1, in ? ImportError: no module named pygments

int recursiveFunction() {
if (baseCase())

Unknown macro: { return something }
else
Unknown macro: { return recursiveFunction() + calculation(); }
}
```

Participants, especially those unfamiliar with recursion, will tend to favor the first example, but the second can often be cleaner, with one less parameter to pass around, and a few less lines of code.

General strategy

The graph problem gets more complex as more wrinkles are added. After the first few days, participants may often be reluctant to have any duplicated code at any time, instead trying to edit algorithms in place, or rewrite them from scratch. While this works, it is harder to keep working code and make small changes. Emphasis should be placed on the **removal** of duplcation, instead of the non-existence of it. When new questions are asked of the graph, code should just be copied with the new code modified to answer the old question. When both the new and the old code are working, the next step is to work on removal of duplication. Pointing to the hierarchy of simple design, the first test is "does it work?" and later is "is there no duplication?".

Conclusion

The Graph example is often a slog. Recursion is difficult to master and is also a bit textbook in nature, so participants may wonder why they've gone through the effort.

It may be helpful to emphasize that the point of the exercise is not about recursion or graph theory, but about what good encapsulation, along with the focus on TDD can bring you in terms of design. Ask the participants what they would have started out with? In contrast to a graph object, which would probably end up doing everything, by manipulation strings or arrays, this object never materializes in the course of the exercise, and instead we have two objects in a collaboration relationship that end up being much smaller and simpler.

Lab1: Can Reach

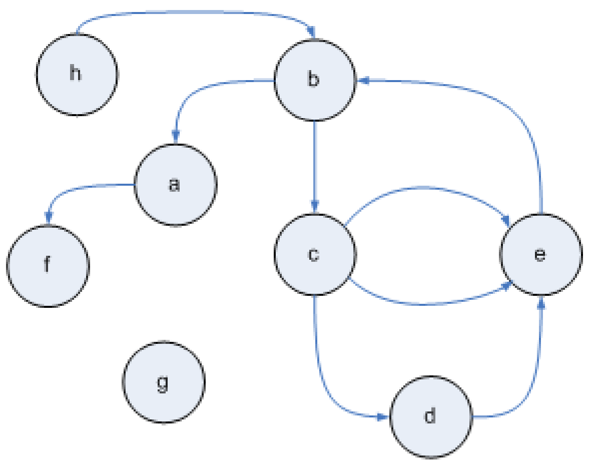
Show the students the graph below. Tell them their first challenge will be to implement canReach() for a graph. Have a 5 minute class design discussion.

Have the students implement.

Things to look for

- Node's job: understands its neighbors
- NodeTest's job: ensures correctness of Node
- First test should be testNodeReachesSelf()
- Iterative solution instead of recursive
- The 'neighbors' field in Node cannot be a Set because there are 2 links from 'c' to 'e'
- In canReach() don't need to check if other.equals(neighbor) because that check will be done in the recursive call.

It is likely the students will start spinning their wheels on this problem. At that point stop and start the recursion discussion.



Recursion Discussion

In this graph we see a lot churn. Churning often leads to *recursion*.

If recursion was not covered on its own earlier, it may be necessary to discuss it here. Ask the students to define recursion. *Defining a function or method in terms of itself.*

Word Reversal Exercise
Ask 4 students to the front of the room. Write letters on notecards, give them to the students and line the students up to spell 'WORD'. Ask 'W' to *give me the list reversed*. 'W' will probably do all the work and tell each letter where to move. Once they are done, point out that 'W' did all the work. 'W' is acting as a God object. Tell students to try again, but this time letters can only talk to their neighbors. Look for

each letter to ask its neighbor to *give me yourself reversed*.

Draw out that the result is built up after they hit the end point 'D' and work their way back. The result is not built up on the way down, it's built up on the way back.

Ask students what the original request was that we made of the word. It was to reverse itself. This is the *original question*. Write on board.

Ask students when they stopped recursing down the word? When they got to the last letter 'D' and there were no further letters. That is a *termination condition*. Add to board leaving space for *recursive question*. All recursion has at least one termination condition.

Elict termination conditions for graph problem:

- am I at the destination?
- do I have any more neighbors to ask?
- have I been seen before in this search?

Lab2: Finish Can Reach

Have the students complete canReach(). Help them as needed.

There is a loop b->c->e->b that will cause a stack overflow. To solve this the students will need to create a set of seen nodes. Introduce the *recursive question* here. It is very common to see this scenario - a public method that is called once to answer the *original question* and then a private method that is called many times and answers the *recursive question*. Ask the students what the recursive question is: can you reach this destination given you have already seen these nodes.

Write *recursive question* between *original question* and *termination condition* on board.