Mitchell Van Braeckel, 1002297, mvanbrae@uoguelph.ca

CIS*4150: Software Testing and Reliability
A1 - Unit Tests and Control Flow Graphs

Part 1: Questions from the Book

Question 1.7
The junior test designers of Chipmunk Computers are annoyed by the procedures for storing test cases together with scaffolding, test results, and related documentation. They blame the extra effort needed to produce and store such data for delays in test design and execution. They argue for reducing the data to store to the minimum required for re-executing test cases, eliminating details of test documentation, and limiting test results to the information needed for generating oracles. What argument(s) might the quality manager use to convince the junior test designers of the usefulness of storing all this information?

The test documentation details are important to have as artifacts that explain the context surrounding tests. For instance, the details of test documentation can give us information about what is being tested, why it is being tested, how it is being tested, and why the tests were implemented in a particular way. This information can significantly help new members and existing members that are unfamiliar with this area. Furthermore, this will help the team decide what other artifacts (ie. related code, tests, scaffolding, infrastructure, etc.) need to change in the future when code or requirements/specifications change.

Limiting test results to only the information needed for generating oracles can have consequences of lowering the overall quality and coverage of the test suites. For example, some cases may be overlooked if they aren't part of generating the article. In addition, this can affect bug identification and fixing such bugs because it's more difficult for a developer to determine exactly where the problem exists when there is a lacking quality and coverage of test cases. This human labour for investigation and fix is more expensive than automatically running more tests and generating more data.

Reducing the data to store only the minimum required for re-executing test cases can have similar negative consequences where we may inadvertently be damaging the quality and coverage of the data set, in turn making it less accurately represent the real data that we aim to test. Additional data can also be helpful when trying to determine the cause of errors.

Question 2.1
The Chipmunk marketing division is worried about the start-up time of the new version of the RodentOS operating system (an (imaginary) operating system of Chipmunk). The marketing division representative suggests a software requirement stating that the start-up time shall not be annoying to users.

Explain why this simple requirement is not verifiable and try to reformulate the requirement to make it verifiable.

This requirement is not verifiable because it does not explicitly define its goal since “annoying to users” is vague and subjective. Therefore, this can lead to a misunderstanding due to the ambiguity and disagreement between individuals when determining the threshold of being “annoying to users.”

To reformulate the requirement and make it verifiable, we should change the language and goal to be something quantifiable. For instance, we can instead suggest a requirement that the average start-up time must be less than X amount of time. The amount of time can be discussed and agreed upon by both developers and the marketing team this way and there’s also a clear objective given by the requirement. This requirement is specific can also be tested and even automated in a way that it can be tracked and expected. For instance, later if a new OS update has a side-effect of increasing the start up time, it would be caught by such tests.

Question 2.3

A calendar program should provide timely reminders; for example, it should remind the user of an upcoming event early enough for the user to take action, but not too early. Unfortunately, “early enough” and “too early” are qualities that can only be validated with actual users. How might you derive verifiable dependability properties from the timeliness requirement?

Since “early enough” and “too early” are vague and subjective to each individual user’s own idea of those definitions, we should aim to quantify these two statements such that they can be used in the requirements of the timely reminders. One of the ways a team may go about doing this is conducting research with users via surveys or focus groups in order to get feedback from their users about their idea of “early enough” and “too early.” The questions asked could simply be prompting them for how early the ideal calendar would send a timely reminder for an event as well as what they think the minimum and maximum reminder times are. Using the collected data from the research, the team can analyze the results and use it to quantify our terms “early enough” and “too early” in the requirements, thus making these two terms verifiable dependability properties within the timeliness requirement.

Question 2.5

When updating a program (e.g., for removing a fault, changing or adding a functionality), programmers may introduce new faults or expose previously hidden faults. To be sure that the updated version maintains the functionality provided by the previous version, it is common practice to re-execute the test cases designed for the former versions of the program. Re-Executing test cases designed for previous versions is called regression testing.

When testing large complex programs, the number of regression test cases may be large. If updated software must be expedited (e.g., to repair a security vulnerability before it is exploited), test designers may need to select a subset of regression test cases to be re-executed.

Subsets of test cases can be selected according to any of several different criteria. An interesting property of some regression test selection criteria is that they do not to exclude any test case that could possibly reveal a fault.

How would you classify such a property according to the sidebar of page 21?

On the sidebar of page 21, it explains three more notes on terminology about the terms safe, sound, and complete. The property of some regression test selection criteria to not exclude any test case that could possibly reveal a fault is putting emphasis on minimizing the number of bugs and errors in the program or software. In other words, the goal of this property is minimizing optimistic inaccuracy because the objective is to minimize the number of faults missed by tests. Therefore, this is a conservative/safe analysis.

If we look at the terms of sound and complete that are also mentioned on page 21, we can see that the selection criteria is not sound and that this is an optimistic analysis. For instance, if we missed a possible fault when picking a subset of regression tests that pass, then soundness is satisfied, but the criteria is violated. Furthermore, the program evaluated would return true if the criteria is satisfied when tests that would reveal a fault are excluded.

Question 3.1

Indicate which principles guided the following choices:

1. *Use an externally readable format also for internal files, when possible.*
2. *Collect and analyze data about faults revealed and removed from the code.*
3. *Separate test and debugging activities; that is, separate the design and execution of test cases to reveal failures (test) from the localization and removal of the corresponding faults (debugging).*
4. *Distinguish test case design from execution.*
5. *Produce complete fault reports.*
6. *Use information from test case design to improve requirements and design specifications.*
7. *Provide interfaces for fully inspecting the internal state of a class.*

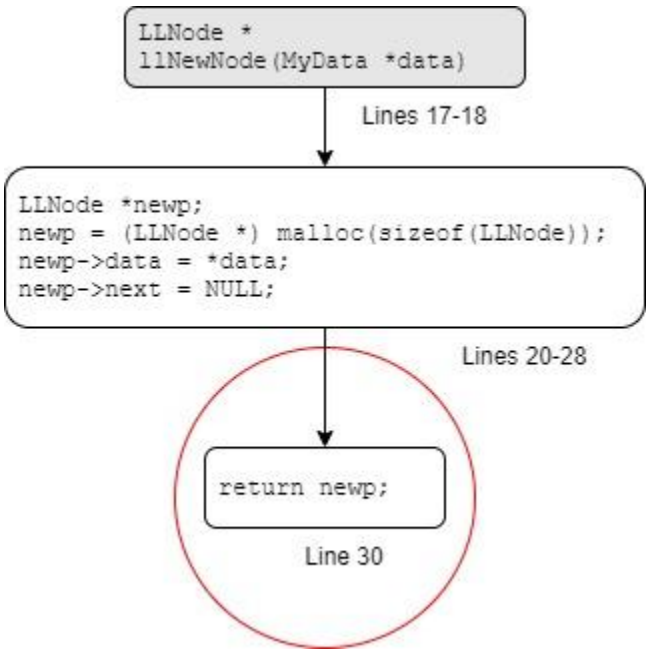
Matching principles to the above choices:

1. Visibility
2. Feedback
3. Partition
4. Redundancy
5. Visibility
6. Feedback
7. Visibility

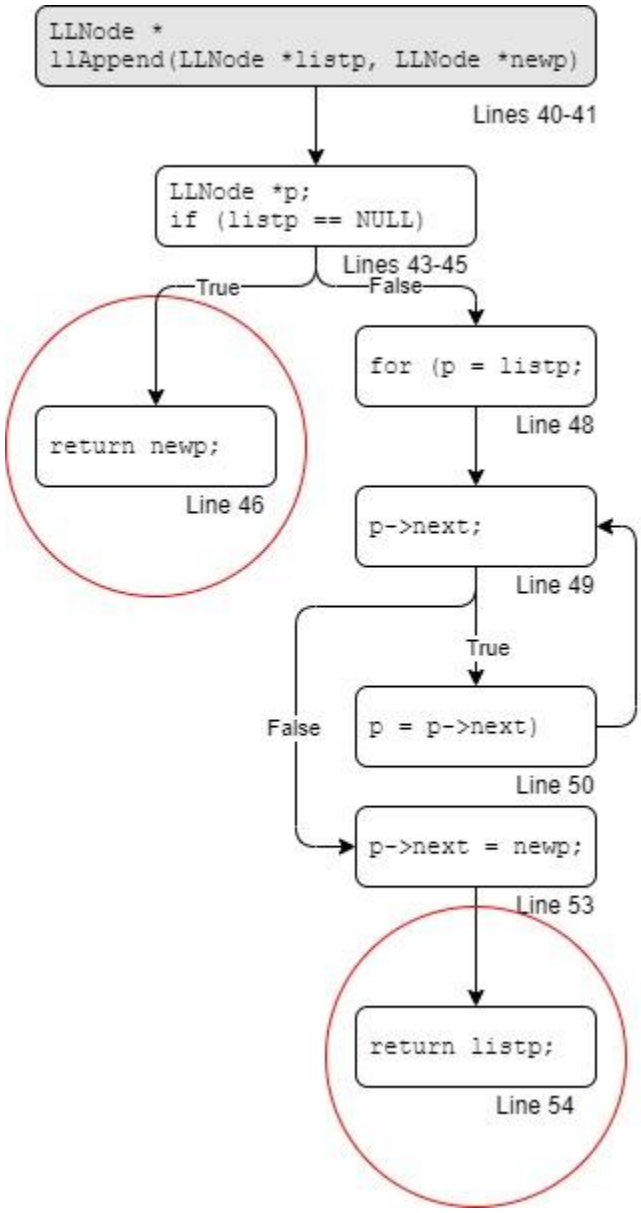
Part 2: Control Flow Graph Problems

With regards to the instruction “circle any nodes that represent a point where control flow leaves the graph,” I interpreted this as “leaving the graph using a ‘return’ or exit.”

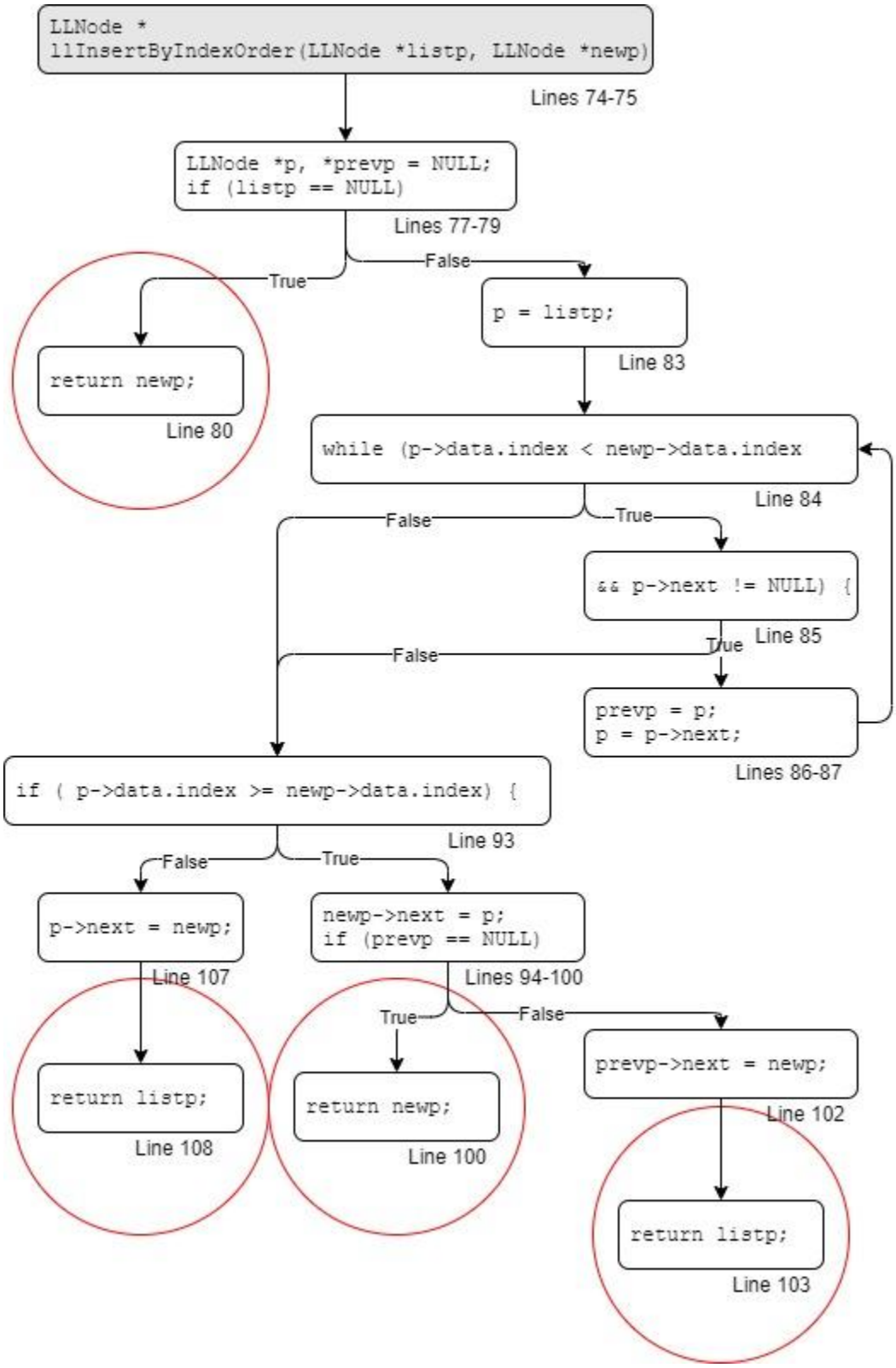
Graph 1: `llNewNode`



Graph 2: `llAppend`



Graph 3: `llInsertByIndexOrder`



Part 3: Programming Task

Python Implementation

Please see `A1-GPA-Calculator/python-GPA-calculator/runtests.py` for the Python tests created. These can be run on the machine `linux.socs.uoguelph.ca` via `python3 runtests.py` when in the `python-GPA-calculator` subdirectory.

Please see `A1-GPA-Calculator/python-GPA-calculator/README.md` for the list of all Python test cases created and whether each test case passes or fails.

Java Implementation

Please see `A1-GPA-Calculator/java-GPA-calculator/TestConversions.java` for the Java fixture and tests created. These can be run on the machine `linux.socs.uoguelph.ca` when in the `java-GPA-calculator` subdirectory by first building via `buildJavaTests.sh` and then running via `runJavaTests.sh`.

Please see `A1-GPA-Calculator/java-GPA-calculator/README.md` for the list of all Java tests created and whether each test case passes or fails.