CSC207.01 2014S, Class 20: Reasoning About Loops with Loop Invariants

Overview

- Preliminaries.
  - Admin.
  - Upcoming work.
  - HW5.
- Topics.
  - Writing correct iterative algorithms.
  - The state of a program.
  - Loop invariants.
  - Loop termination.
  - An exercise: Binary search.
- Lab.

Preliminaries

Admin

- Exam 1 due *now*!
- Yesterday's review session may or may not have some interesting notes.
- Yet another reminder to think about internships.
- Extra credit:
  - CS Table today: Skip Lists
  - Town hall, Tuesday, noon or 7 pm
  - CS Extras next week: The new CS Curriculum
  - More?

Upcoming Work

- Reading for Monday: Anonymous Inner Classes.
  - Yes, it's ready.
  - Yes, I'm likely to update it anyway.
  - No, it won't matter whether you read it before or after the update.
- No writeup for today.
- Homework 5.

Homework 5

Choose your own partners. You may also work alone. You will be tested on many of these topics, so "divide and conquer" is not a good approach.

Part one: Solve some recurrence relations.

Part two: Implement Dutch National Flag, using invariants.

- Details on what the problem is today.
- Assumption:

```
public interface Classifier<T>
{
  /**
   * Classify val into one of three categories, which we call
   * "red", "white", and "blue" for convenience.  If val is red,
   * returns a negative number.  If val is white, returns zero.
   * If val is blue, returns a positive number.
   */
  public int classify(T val);
} // interface Classifier
```

or

```
public interface StringClassifier
{
  /**
   * Classify val into one of three categories, which we call
   * "red", "white", and "blue" for convenience.  If val is red,
   * returns a negative number.  If val is white, returns zero.
   * If val is blue, returns a positive number.
   */
  public int classify(String val);
} // interface StringClassifier
```

or

```
public interface StringClassifier
  implements Classifier<String>
{
} // interface StringClassifier
```

Part three: Implement Skip lists of strings, using invariants.

- Implementation details in the CS table article.
- Interface

```
public interface StringSet
{
  /**
   * Determine if the set contains a particular string.
```

```
   */
  public boolean contains(String str);


  /**
   * Add an element to the set.
   *
   * @post contains(str)
   */
  public void add(String str);


  /**
   * Remove an element from the set.
   *
   * @post !contains(str)
   */
  public void remove(String str);
} // interface StringSet
```

Part four: Implement iterative logn exponentiation, using invariants

Writing correct iterative algorithms

- Testing is one approach to correctness. But it's retrospective.
- We'd also like tools that help us think about design of algorithms.

The state of a program

- In replacing elements in MutableStrings, we probably thought about
  - An array of characters
  - The position of the first matched pattern
  - The relative sizes of the pattern and replacement
  - The expected result
  - Where we look next
- We think better about state if we write it down and reason about it.

Loop invariants

- Loops are chunks of code that do something again and again and again with a hope of achieving some goal.
- We want to think carefully about what one pass through the loop does.
- Loop invariants: Something you can assume that the invariant holds at the beginning of the loop.
  - Also something you can be sure of at the end of the loop
- This may not sound like progress, but we'll work with it in a way that assures progress.
- Goal: The loop invariant is something that is useful.
  - You know the loop invariant holds when the loop finishes.
  - You know the loop has finished.

- Hopefully, this combination of facts implies that we've achieved our goal.

Writing Loop Invariants

- Write or draw a picture that indicates what you expect about the state of the system.
- Figure out how to make it true before the loop starts.
- Figure out what else you need to know at the end.

Loop termination

- Some metric of size (work remaining)
- Show that size decreases at every repetition.

An exercise: Binary search

```
/**
 * Find the index of val in values.
 */
public static int binarySearch(int[] values, int val)
{
  int lb = 0;
  int ub = values.length;

  // Invariant A: for all i, 0 <= i < lb, values[i] < val
  // Invariant B: for all i, ub <= i < length, values[i] > val
  while ((ub - lb) > 0)
    {
      int mid = average(lb, ub);
      if (values[mid] < val)
        lb = mid+1;
      else if (values[mid] > val)
        ub = mid;
      else
        return mid;
    } // while
  // At this point
  // Invariant A: for all i, 0 <= i < lb, values[i] < val
  // Invariant B: for all i, ub <= i < length, values[i] > val
  // lb >= ub
  // The element is not there
  throw new NotFound();

} // binarySearch(int[], int)
```

Lab

Copyright (c) 2013-14 Samuel A. Rebelsky.

This work is licensed under a Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.