CSE 331

Debugging

Kevin Zatloukal

Administrivia

- HW7 closely tied with Section 7
  - section defines functions (with and without) used in HW7
  - explains the basic idea of one of the ADTs
- Worth your time!
- HW7 has multiple concrete representations for an ADT
  - like last time, changing rep to speed up some operations

Recall: Mutable Queue ADT

- Mutable versions has mutators instead of producers

```
// Mutable array that only supports adding to the front
// and removing from the end.
interface MutableNumberQueue {

  // @returns obj
  elements(): number[];

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue(x: number): void;

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue(): number;
}
```

observer

mutator

mutator

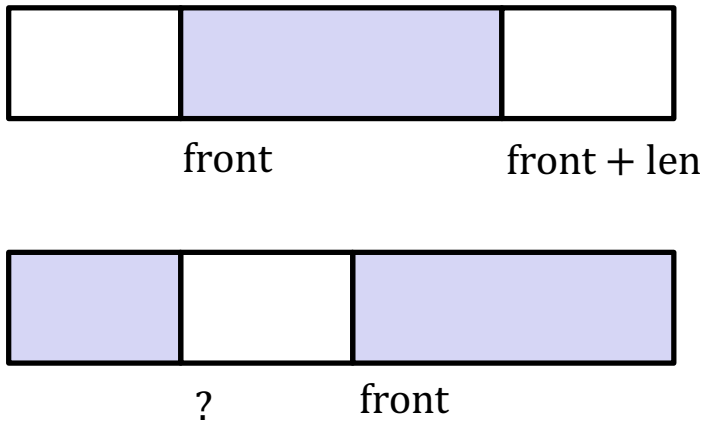
Recall: Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  readonly front: number[];
  readonly back: number[];

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue = (x: number): void => {
    this.front.push(x);
  };
```

- AF & RI explain the idea of the representation
- Use our reasoning tools to check correctness

Recall: Mutable Queue with One Array

- **Will record** front **index and** len **of queue**
- **The array looks one of these**



- **Abstraction function has two cases**
  - **multiple cases in every correctness argument**

    enqueue has three cases!

    dequeue does also (end of slides from last time)

Morals of the Course So far

- More mutation can give us better efficiency
- More mutation means more complex reasoning
  - more facts to keep track of
  - more ways to make mistakes
- Heap state is shared between methods (Level 3)
  - mutation in one method can break another method
  - error could occur long after, far away from the bug

Debugging

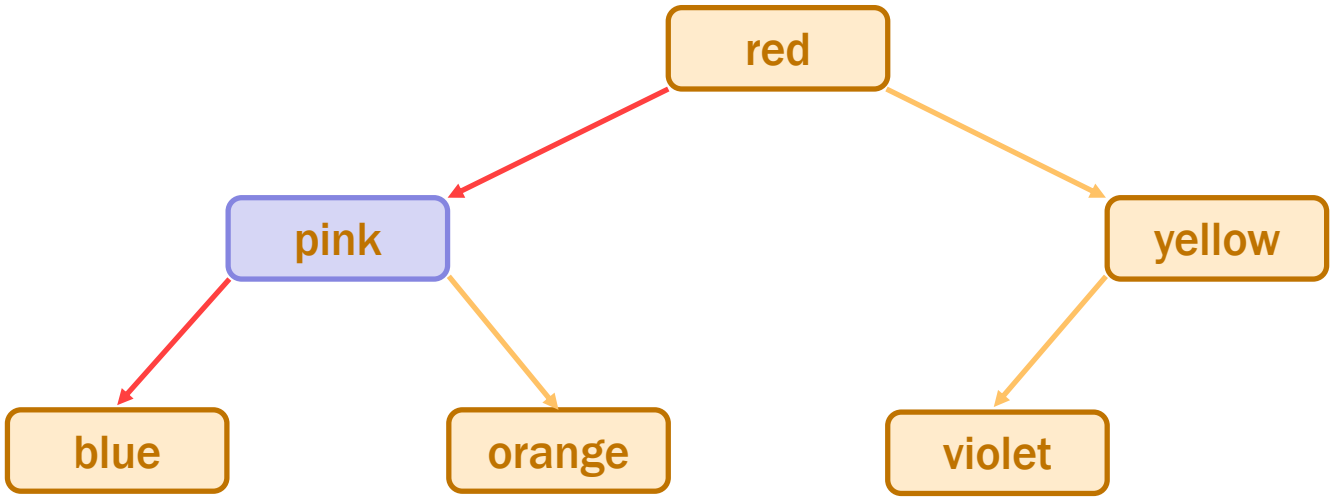
A Bug's Life

- **Defect** ("the bug"): mistake made by a human
- **Error**: computation performed incorrectly
- **Failure**: mistake visible to the user

Debugging is the search
from failure back to defect

Horrible Bug #1

- **Suppose we called `makeFavColor` on green record…**
  - **it is mutated into pink**
  - **now this happens when we look for <u>orange</u>:**



  - **it can no longer be found!**

    we violated the BST invariant

Horrible Bug #1

- **Failure** occurs somewhere after color is not found
  - user gets an error message saying no color called “orange”
- **Error** occurs when the mutation happens
  - in an invalid state, with BST invariant violated
- **Defect** is… somewhere else
  - `makeFavColor` does what it is supposed to
  - may not be in `ColorTree`

    bad design (rep exposure) to give away a reference to this node

    but could have a spec that says not to modify it
  - bug in call passing reference that could be modified

Horrible Bug #2

- Suppose we have the following class

```
class Subarray {

  // RI: 0 <= start < start+len <= vals.length
  // AF: obj = vals[start .. start+len-1]
  readonly vals: number[];
  readonly start: number;
  readonly len: number;

  get = (i: number): number => vals[start + i];

  …
```

- Lets us use a part of a subarray like an array
  - saves us from making a copy

Horrible Bug #2

- **Suppose someone mutates the array:**

```
vals.pop();  // shorten array by one, now [1, 2, 3]
```

- **RI of the class is violated if** $\text{start} = 2$ **and** $\text{len} = 2$

```
// RI: 0 <= start < start+len <= vals.length
readonly vals: number[];  // length is now 3
readonly start: number;   // start is still 2
readonly len: number;     // start+len is still 4
```

- **Call to `get` can return an unexpected value**

```
get = (i: number): number => vals[start + i];
```

i = 1

start + i = 3

vals[start + i] = undefined

Horrible Bug #2

- **Suppose someone mutates the array:**

```
vals.pop();  // shorten array by one, now [1, 2, 3]
```

- **Call to `get` can return undefined**

```
get = (i: number): number => vals[start + i];
```

- **Arithmetic can produce unexpected results**

```
const amount: number = S.get(1) + 10;
```

NaN

- **User sees an unexpected error result**

Account Balance (with $10 coupon): $NaN

Horrible Bug #2

- **Failure** occurs when the balance is shown
  - user sees a balance that is obviously wrong
- **Error** occurs when the mutation happens
  - in an invalid state, with RI violated
- **Defect** is… somewhere else
  - not necessarily in the code that calls `vals.pop()`
  - bug in call passing reference that could be modified
  - was not in Subarray

    bad design, allowing rep exposure

    safety requires making a copy (but then why make this class at all?)

Level 3 = Horrible Debugging

- Both examples mutated heap state
  - not an accident
- Mutation of heap state allows painful debugging
  - correctness depends on code far away from failure
    aliases can be passed all over the code base
  - correctness depends on code run much earlier
    rep invariants can become invalid *after* the constructor
    arbitrary amount of time between mutation and later read
- Level 0–2 code can be understood locally
  - debugging can still be bad as errors ripple through the code
    e.g., undefined becomes NaN

Tips to Avoid Debugging

1. Let the tools catch any bugs they can
   - never ignore type errors
   - never use unchecked type casts

larger program means
a larger search space

2. Get it right the first time
   - reason carefully through your code
   - test each function thoroughly

3. Make errors **immediately visible**
   - shorten the search by “failing fast”
   - program **defensively** by adding extra checks
   - include useful information when crashing, “fail friendly”

Defensive Programming Tip #1

- Write code to check preconditions
  - don't trust other programmers

```
// @requires 0 <= i < len(obj)
get = (i: number): number => {
  if (0 < i || this.len <= i)
    throw new Error(`bad index ${i} (${this.len})`);
  return this.vals[this.start + i];
};
```

- Fail immediately instead of returning undefined
  - fail fast and friendly

Defensive Programming Tip #1

- **Write code to check preconditions**
  - don't trust other programmers

```
// RI: 0 <= start < start+len <= vals.length
constructor(
    vals: number[], start: number, len: number) {
  if (start < 0) throw new Error(`bad start ${start}`);
  if (len < 0) throw new Error(`bad length ${len}`);
  if (vals.length < start + len)
    throw new Error(`${start+len} < ${vals.length}`);

  this.vals = vals;
  this.start = start;
  this.len = len;
};
```

Defensive Programming Tip #2

- Can also write code to check postconditions
  - don't trust yourself either!

- Useful to check loop invariants when debugging
  - see `CheckInv1` in HW6

    if it fails, then it helped you find an error
  - can comment this out later

- Most useful checks are to prevent worst debugging…

Defensive Programming Tip #3

- **Write code to check representation invariants**

```
// @requires 0 <= i < len(obj)
get = (i: number): number => {
  if (i < 0 || this.len <= i)
    throw new Error(`bad index ${i} (${this.len})`);
  this.checkRep();
  return vals[start + i];
};

checkRep = () => {
  if (this.vals.length < this.start + this.len)
    throw new Error(`RI violated: ${…}`);
};
```

Will catch RI violation in
first method call after mutation

don't need to check, e.g., 0 ≤ start
since start is never changed

Defensive Programming Tip #3

- **Write code to check representation invariants**
  - **can use `checkRep` to check your work also**
  - **example from `ArrayQueue`**

```
constructor() {
  ... set field values for an empty queue ...
  this.checkRep();  // make sure RI holds
};

dequeue = (): number => {
  this.checkRep();  // catch rep exposure early
  ... update field values ...
  this.checkRep();  // make sure RI holds again
  return x;
};
```

Debugging

- **Debugging is different from coding**
  - **only happens when states are not as expected**

    variable has an unexpected type

    state does not satisfy the expected assertions

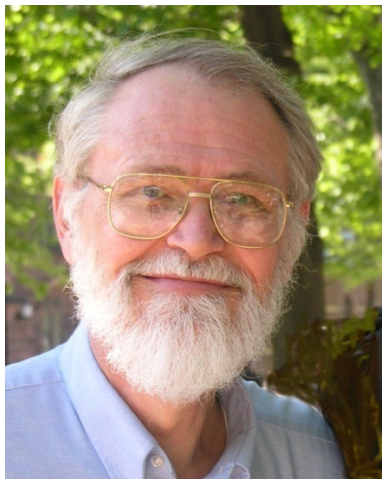
- **Arguably harder than coding:**

  "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

  - **write code as simply as possible**

    if not level 0, then level 1

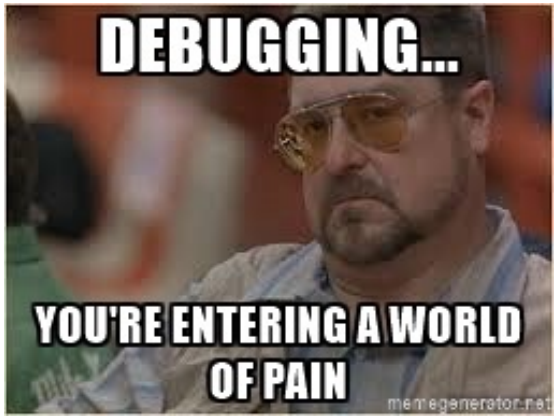
    if not level 1, then level 2



Brian Kernighan

Gets Even Worse

- Can have bugs that only happen *sometimes*
  - often called “heisenbugs”
  - must figure out exactly when it happens
    - only under high load
    - only when several servers are specific states
    - only on certain dates / phases of the moon



- Can have bugs that go away when you debug!
  - common with multi-threaded applications
  - common when writing compilers

- These are the **most likely** to cause debugging
  - most likely to sneak past tools, testing, and reasoning

Debugging Tip #1

- **Create a minimal example that demonstrates the bug**
  - **easier to look through everything in the debugger**

- **Shrink the input that fails:**

  Find “very happy” in “Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all.”

  Find “very happy” in “I am very very happy to see you all.”

  not the accent characters

  Find “very happy” in “very very happy”

  something to do with partial match

  Find “ab” in “aab”

How to Fix a Bug

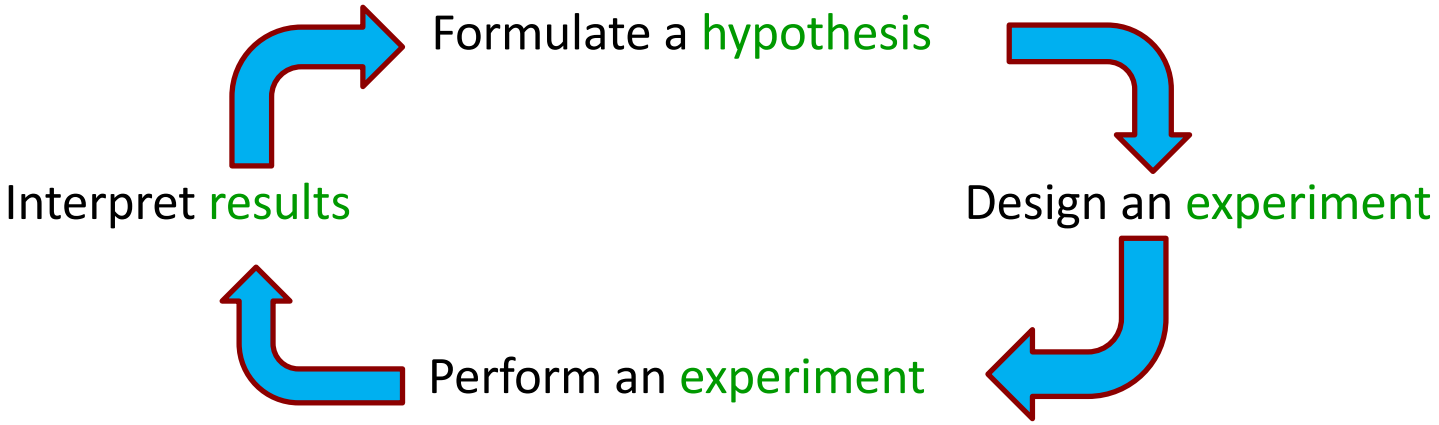
- Start with a test that **fails**
  - make sure you see it fail!
  - can mistakenly write a test that worked already
- Understand why it fails
  - understand where your reasoning was wrong
- Fix the bug
- Make sure the all the tests now pass
  - new test and all previous tests

Defensive Programming Tip #4

- If you spent 30+ min debugging, make it a **test case**
  - solid evidence that it's a tricky case
- Bugs that happen once often come back
  - code is changed in the future
  - good chance the same error will happen in the new version
- These are called "regression tests"
  - avoid the bug coming back ("regressing")

Debugging Tip #2

- After 20+ min debugging, be systematic
  - don't just try things you think might fix it
- Use the Scientific Method:



- Write down what you have tried
  - don't try the same thing again and again

Debugging Tip #3

- Use Binary Search to find the error

  RI holds when the object is created

  …

  RI is violated when user clicks “submit”

- Find an event that happens somewhere in the middle

  RI holds when the object is created

  …

  does it hold when the user clicks on the dropdown?

  …

  RI is violated when user clicks “submit”

  – save an alias to the object when created

Debugging Tip #4

- Check the easy stuff
  - make sure all the files are saved
  - restart the server
  - restart your computer
  - make sure someone didn't already fix it
- If it is one of these, you will not find it debugging
  - every minute you spend until you hit save / restart is wasted

Debugging Tip #5

- **Look for common silly mistakes**
  - **in Java, comparing strings with `==` instead of `equals`**
    thankfully, not a problem in TypeScript
  - **misspelling the name of a method you were implementing**
    in Java, implementing "`equal`" instead of "`equals`"
  - **passing arguments in the wrong order**

- **Easy for these to slip past reasoning**
  - **better chance of finding them with tools or testing**
    tools will miss wrong order if both arguments have the same type
  - **but some will slip through**

Debugging Tip #6

- Make sure it is a bug!
  - check the spec carefully
  - tricky specs can trick you

- These are the absolute worst
  - spend hours and then discover the code was right all along

Debugging Tip #7

- Try explaining the problem to someone / something
  - can even be a rubber duck

    Pragmatic Programmer calls this “rubber ducking”

- Talking through the problem often helps you spot it
  - this happens all the time

Debugging Tip #8

- Get some sleep!
  - the later it gets, the dumber I get
  - often don't realize it until 4–5am
- Common to wake up and instantly see the problem
- Important to start early!
  - can't do this the night it is due

Debugging Tip #9

- Get some help!
  - easy for bugs to hide in your blind spots
- After some number of hours, continuing is not helpful
  - need new ideas about where to look
- Important to start early!
  - no office hours late at night