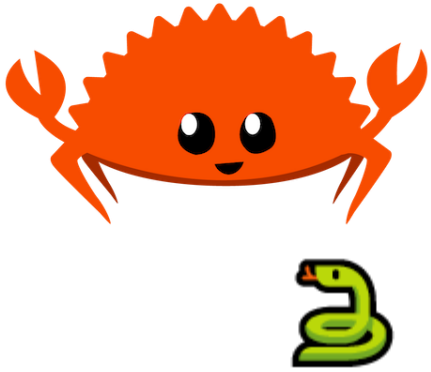
CSE 331 Summer 2025

Mutation

Jaela Field



Administrivia

- **HW3 released last night**
  - **Example responses (based on Th section) will be posted on Ed later today**

    Gradescope will be updated with links
  - **Last question asks for feedback!**

    Feel free to mention HW2 notes there also

Mutation

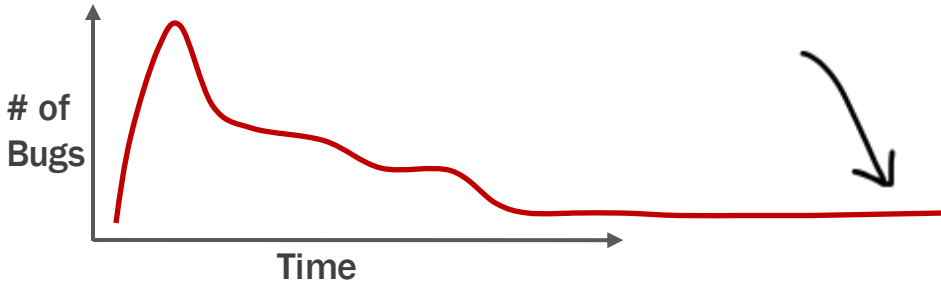
HW2 – mutation?

- In HW2, we asked you about "mutation bugs"
  - code mutated something that it wasn't supposed to

    i.e. didn't "own" the variable, directly reassigning instead of using proper functions
- Historically,
  - students report ~10% of bugs are mutation related
  - such bugs took significantly longer to debug!
  - the bugs that students have but *don't* find on this assignment are generally mutation related

HW2 – mutation?

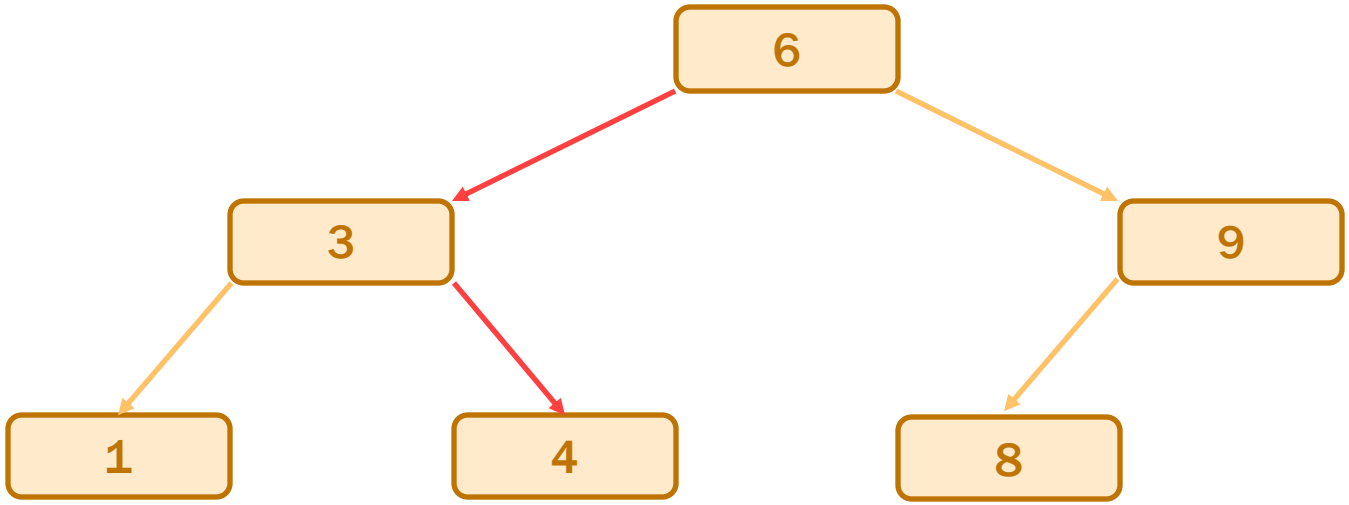
- not-as-common as type related errors, but **much nastier** to debug



- our goal: help you build ability to recognize indicators of potential problems  without running code (or seeing all of it)

Recall: Binary Search Trees

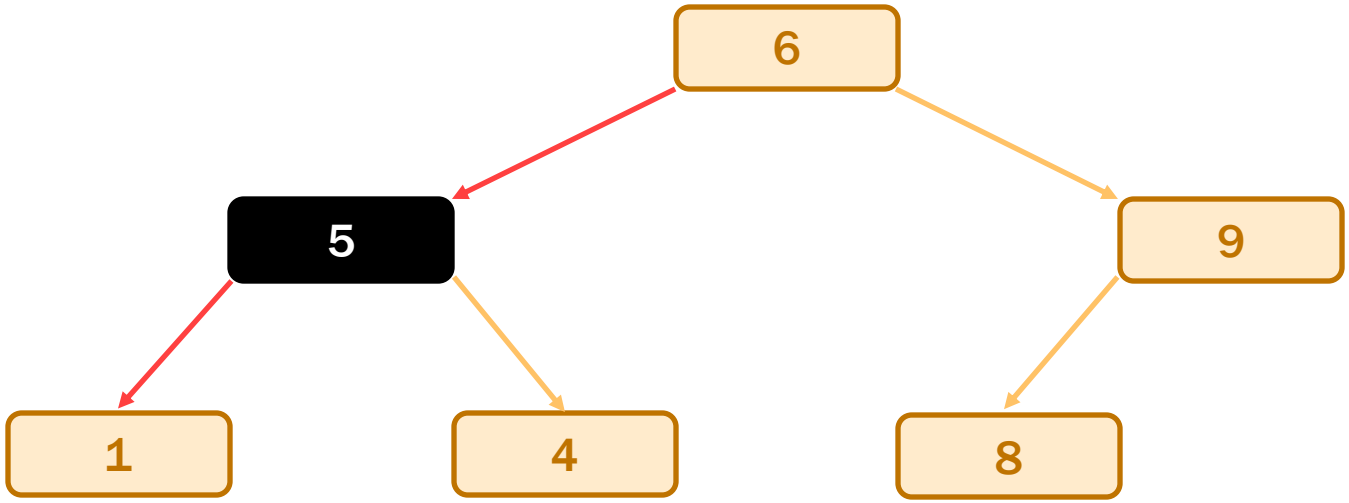
- Consider the following tree
  - searching for "4" proceeds as follows:



- Suppose someone changed "3" into "5"...

Binary Search Trees & Mutation

- **Suppose someone changed "3" into "5"...**
  - **now this happens when we search for "4":**



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

    Doesn't crash. It's just not found.
  - **Problem doesn't occur on the line with the change**

HW2 Debugging via User Report

- **User reports the following bug:**

  ”Uh, sometimes, I can't click on one of the markers.

  Usually, it it works fine. But occasionally, you can't click on it.”

- **How do you debug this?**
  - **Reproducing it is challenging enough!**

    key reason why event-driven debugging is harder
  - **No error message, or exception to go off of**

    No line number to start with
  - **have to learn how** `App.tsx` **works, then how** `marker_tree.ts` **works**

Scary Bugs

- Do not fear crashes
  - often no debugging at all

    get a stack trace that tells you exactly where it went wrong

- Do fear unexpected mutation
  - failure will give you no clue what went wrong

    will take a long time to realize the BST invariant was violated by mutation
  - bug could be almost anywhere in the code
  - could take *weeks* to track it down

Think Pair Share: M-you-tation

```
const todos: Array<TodoItem> = /* ... */;
const incompleteTodos: Array<TodoItem> =
                findAllIncompleteTodos(todos);

incompleteTodos.shuffle();
// suspicious mystery FUNCTION HERE :))

console.log(`Why not try: ${incompleteTodos[0]}`);
```

Consider these functions – which could break this feature? How?

1. mystery(todos)
2. mystery(incompleteTodos)
3. mystery(todos[0])
4. mystery(incompleteTodos[0])

sli.do #cse331

Aliasing

Heap State

- "Heap state" = still used after the call stack finishes
  - after current function and those calling it all return
  - state could be arrays or records
- Extra references to the objects are called "aliases"
- No different from before when *immutable*
  - we don't care who reads the data
- Vastly more complex when **mutable**…
  - common with event-driven applications
  - creates the potential for failures far from bugs

Coupling

- High-quality code needs to be "modular"
  - split into pieces that can be understood *individually*
- When not possible, pieces are "coupled"
  - must understand both parts to understand each one
- Mutable heap state creates coupling
  - all pieces must know who else has aliases
  - all pieces must know who is allowed to mutate
- Coupling creates potential for **painful** debugging
  - bugs in one piece can cause failures in another

Mutable Heap State

- "With great power, comes great responsibility"
  - from Uncle Ben (1972, 2002-*)

- With aliases to mutable heap state:
  - gain efficiency in some cases
  - must keep track of every alias that could mutate that state

    any alias, anywhere in the *entire* program could cause a bug

"Programmers overestimate the importance of **efficiency** and underestimate the difficulty of **correctness**."

— Class slogan #2

Easy Ways to Stay Safe

1. Do not mutate heap state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases…
   - create the state in your constructor and **don't share** it

```
class MyClass {
  vals: Array<string>;

  constructor() {
    this.vals = new Array(0);  // only alias
  }
  …
```

Easy Ways to Stay Safe: Copy-on-Write

2. Do not allow aliases

(a) do not hand out aliases yourself

- return copies instead

```
class MyClass {
  // RI: vals is sorted
  vals: Array<string>;

  …

  values: (): Array<string> => {
    return this.vals;              // unsafe!
    return this.vals.slice(0);     // make a copy
  };

  …
```

Easy Ways to Stay Safe: Copy-on-Read

2. Do not allow aliases

(b) make a copy of anything you want to keep

- does not matter if the caller mutates the original

```
class MyClass {
  // RI: vals is sorted
  vals: Array<string>;

  …

  // @requires A is sorted
  constructor(A: Array<string>) {
    this.vals = A;                // unsafe!
    this.vals = A.slice(0);       // make a copy
  };

  …
```

Staying Safe in 331

1. Do not use mutable state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases to *mutable* state
   a) do not hand out aliases yourself
   b) make a copy of anything you want to keep

ensures only one reference to the object (no aliases)

- For 331, mutable aliasing across files is a bug!
  - gives other parts the ability to break your code
  - we will stick to these simple strategies for avoiding it

An Advanced (Two-Stage) Approach

- **Mutable object has only one reference (owner)**
  - **one reference that is allowed to use & mutate it**
- **Object is eventually "frozen", making it immutable**
  - **no longer necessary to track ownership**
- **Example: Java's `StringBuilder` vs `String`**
  - `StringBuilder` **is mutable (be careful!)**
  - `StringBuilder.toString` **returns the value as a** `String`
  - `String` **is immutable**

Rules of Thumb: Mutation XOR Aliasing

Client Side

1. Data is small
   - anything on screen is O(1)
2. Aliasing is common
   - UI design forces modules
   - data is widely shared

Rule: avoid mutation
- create new values instead
- performance will be fine
- (local-only mutation can be OK)

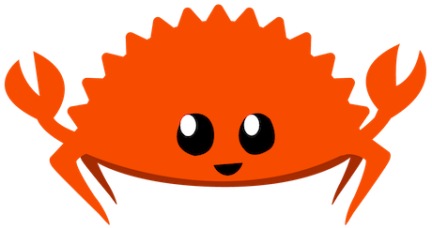
Server Side

1. Data is large
   - efficiency matters
2. Aliasing is avoidable
   - you decide on modules
   - data is not widely shared

Rule: avoid aliases
- do not allow aliases to your data
- hand out copies not aliases
- (good enough for us in 331)

Language Features & Aliasing

- Most recent languages have some answer to this…
- Java chose to make `String` immutable
  - most keys in maps are strings
  - hugely controversial at the time, but great decision
- Python chose to only allow immutable keys in maps
  - only numbers, strings, and tuples allowed
  - surprisingly, not that inconvenient



- Rust has built-in support for “mutation XOR aliasing”
  - ownership of value can be “borrowed” and returned
  - type system ensures there is only one usable alias

Readonly in TypeScript (1/2)

- TypeScript can ensure values aren't modified
  - extremely useful!
  - but, only a compile-time check (not a runtime guarantee)

- Readonly tuples:

```
type IntPair = readonly [bigint, bigint];
```

- Readonly fields of records:

```
type IntPoint = {readonly x: bigint,
                 readonly y: bigint};
```

Readonly in TypeScript (2/2)

- **Readonly fields of records:**

```
type IntPoint = {readonly x: bigint,
                 readonly y: bigint};
```

- **Readonly records:**

```
type IntPoint = Readonly<{x: bigint, y: bigint}>;
```

  - `this.props` **is** `Readonly<MyPropsType>`

- **More readonly...**

```
ReadonlyArray<bigint>
ReadonlyMap<string, bigint>
ReadonlySet<string>
```

comfy-tslint

comfy-tslint

- we've written a TS linter for this class that enforces some of our conventions, e.g.
  - requiring type annotations for functions
  - disallowing the `any` type
  - naming & structure conventions for React methods
- available...
  - as a VSCode extension
  - as an npm module (run with `npm run lint`)
- please:
  - See the comfy-tslint resource for enforced rules
  - take a careful look at the HW3 spec + autograder