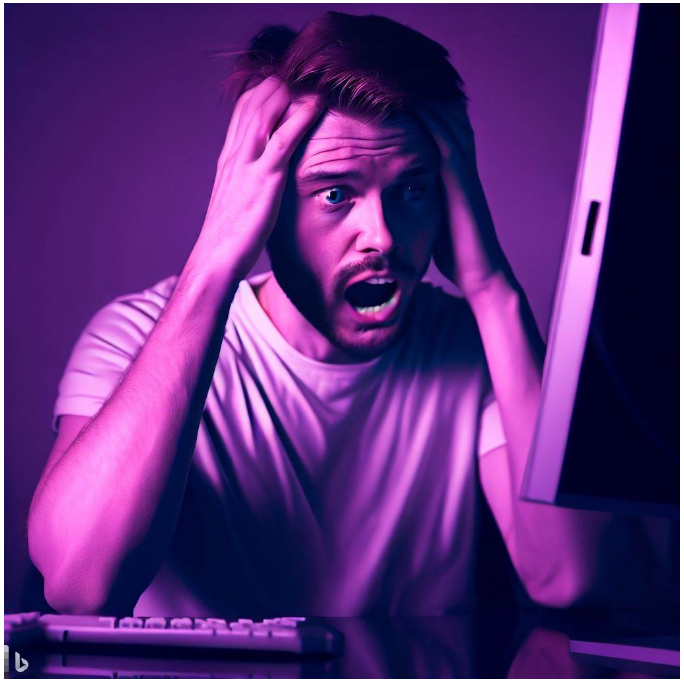


CSE 331

Aliasing

Kevin Zatloukal

HW9 Reminders

- HW9 released last night
  - another debugging assignment
  - make sure you **understand** all the pieces
- HW9 is individual (not group) work
  - will compare solutions for similarity
  - only person you can copy from is **me** (e.g., Auctions)
- Tests your knowledge of lecture content
  - not knowledge of libraries
  - **linter** to updated to further exclude non-331 code

Revisiting HW5

- In HW5, color information in a `ColorInfo` record
  - we used a triple, but a record also works

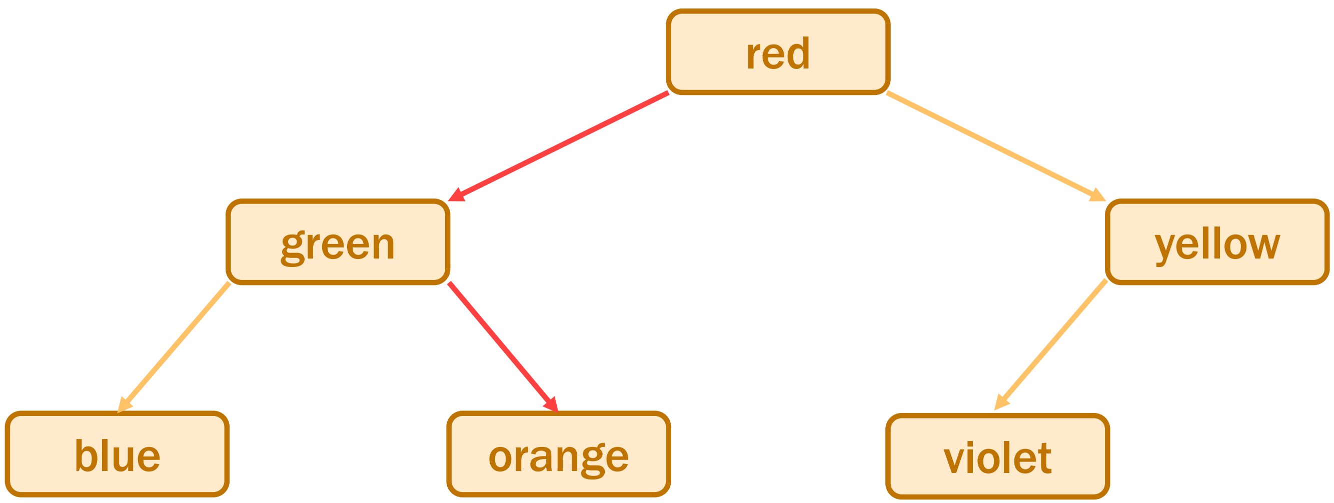
```
type ColorInfo = {
    name: string, cssColor: string, dark: boolean};
```

- Could also write functions that mutate them:

```
const makeFavColor = (c: ColorInfo): ColorInfo => {
  c.name = "pink";
  c.cssColor = "#FFC0CB";
  c.dark = false;
  return c;
};
```

Revisiting HW5

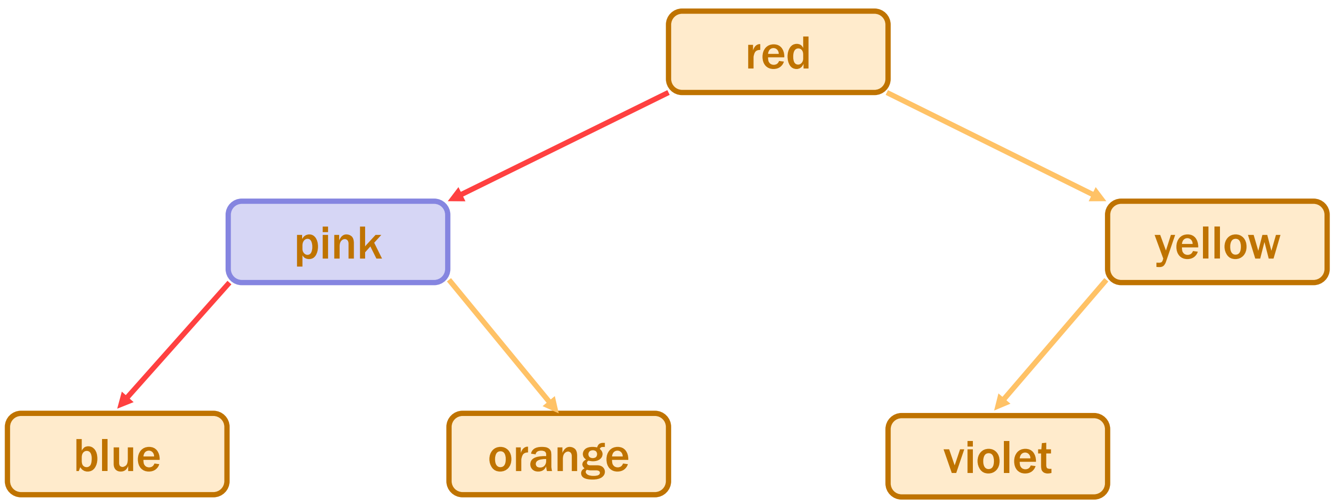
- In HW5, we had a BST of `ColorInfo` records
  - faster way to look up color information
  - e.g., find **orange** like this



- Suppose we called `makeFavColor` on the green record...

Revisiting HW5

- **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

Revisiting HW5

- In HW5, color information in a `ColorInfo` record
  - we used a triple, but a record also works

```
type ColorInfo = {
    name: string, cssColor: string, dark: boolean};
```

- Could also write functions that mutate them:

```
const makeFavColor = (c: ColorInfo): ColorInfo => {
  c.name = "pink";
  c.cssColor = "#FFC0CB";
  c.dark = false;
  return c;
};
```

Scary Bugs

- <u>Do not</u> fear crashes
  - those are easy to spot and fix

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

- <u>Do</u> 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

    anyone who mutates a `ColorInfo` could have caused it
  - could take *weeks* to track it down

Recall: Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	straight from spec	heuristics	type checking	code reviews
1	no mutation	“	libraries	calculation induction
2	local variable mutation	“	“	Floyd logic
3	array / object mutation	“	“	rep invariants

Level 3: Mutable Heap State

- "With great power, comes great responsibility"
- With arrays:
  - gain the ability to easily access any element
  - must keep track of information about the whole array
- Additional references to the same object are "aliases"
- With 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

Easy Ways to Stay Safe

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

2. a) Do not hand out aliases
   - never give anyone else an alias
   - 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

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

2. a) Do not hand out aliases

only one reference to an object
(no aliases)

   - never give anyone else an alias
   - create the state in your constructor and don't share it

   b) Make a copy of anything you want to keep
   - you have the only reference to the newly created copy
   - does not matter if the caller later mutates the original

An Advanced (Two-Stage) Approach

- Mutable object has only one reference (owner)
  - one reference that is allowed to use & mutate it
- Must track ownership of each mutable object
  - can be passed in a function call
  - passed permanently or just "borrowed"
    - borrowing returns ownership back when the call ends
  - Rust programming language has built-in support for this
    - type system ensures that there is only one owner
- Object can be "frozen", making it immutable
  - no longer necessary to track ownership

Mutable ADTs

ADTs

- Main place we have heap state is in an ADT
- Previously:
  - state was immutable
  - set in the constructor and then never changed

    only need to confirm RI holds at the end of the constructor

    if RI holds there, then it holds forever
- Now:
  - allow state to be changed by methods

ADTs

- Main place we have heap state is in an ADT
- New Power:
  - allow state to be changed by methods
- New Responsibilities:
  - more complex specifications

    add `@effects` and `@modifies`
  - must check the RI holds after any method that mutates

    often a good idea to write code to check this at runtime
  - must avoid aliasing of anything mutable

    we call this “representation exposure”

Recall: List ADT with a Fast `getLast`

```
// Represents an (immutable) list of numbers.
interface FastList {

  // @returns cons(x, obj)
  cons: (x: number) => FastList;

  // @returns last(obj)
  getLast: () => number|undefined;

  // @returns obj
  toList: () => List<number>;
};

const makeFastList = (): FastList => {
  return new FastListImpl(nil);
};
```

producer method

Mutable List ADT with a Fast `getLast`

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons: (x: number) => void;

  …
```

mutator method

- Method `cons` changes the list, putting `x` in front
  - now returns `void`
  - mutation explained in `@modifies` and `@effects`

    abstract state is the old abstract state with x put in front

Mutable List ADT with a Fast `getLast`

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons: (x: number) => void;

  …
```

mutator method

- Method `cons` changes the list, putting `x` in front
  - now a mutable data type

    clients need to worry about aliasing
  - don't make a tree of these!

    some languages (e.g., Python) don't allow this

Recall: One Concrete Rep for `FastList`

```
class FastListImpl implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  readonly last: number | undefined;
  readonly list: List<number>;

  constructor(list: List<number>) {
    this.list = list;
    this.last = last(this.list);
  }
```

- We can use the same rep for a mutable version

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    this.list = cons(x, this.list);
  };
```

- **Let's check correctness…**

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    this.list = cons(x, this.list);
    {{ this.list = cons(x, this.list_0) }}
    {{ Post: obj = cons(x, obj_0) }}
  };
```

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    this.list = cons(x, this.list);
```

{{ $\text{this.list} = \text{cons}(x, \text{this.list}_0)$ }}

{{ **Post**: $\text{obj} = \text{cons}(x, \text{obj}_0)$ }}

```
  };
```

What is missing?

Also, need the RI to hold!

obj	$= \text{this.list}$	by AF
	$= \text{cons}(x, \text{this.list}_0)$	since $\text{this.list} = \text{cons}(x, \text{this.list}_0)$
	$= \text{cons}(x, \text{obj}_0)$	by AF

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    this.list = cons(x, this.list);
    {{ this.list = cons(x, this.list_0) }}
    {{ Post: obj = cons(x, obj_0) and
             this.last = last(this.list) }}
  };
```

Also, need the RI to hold!

Does it? No!

- Postcondition is `@returns`, `@effects`, and <u>RI</u>

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = cons(x, this.list_0) and this.last = last(this.list) }}
    {{ Post: obj = cons(x, obj_0) and this.last = last(this.list) }}
  };
```

Rep Invariant now holds

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    this.last = last(this.list);
```

{{ **this.last = last(this.list)** }}

```
    this.list = cons(x, this.list);
```

{{ this.list = cons(x, this.list$_0$) and **this.last = last(this.list$_0$)** }}

{{ **Post**: obj = cons(x, obj$_0$) and this.last = last(this.list) }}

```
  };
```

Rep Invariant would not hold if we switched the order

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    this.list = cons(x, this.list);
    this.last = last(this.list);
```

{{ this.list = cons(x, $this.list_0$) and this.last = last(this.list) }}

{{ **Post**: obj = cons(x, obj_0) and this.last = last(this.list) }}

```
};
```

This version is obviously correct, but O(n).

Can we do it faster?

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: number): void => {
    if (this.list === nil)
      this.last = x;
    this.list = cons(x, this.list);
    {{ ____________________________________ }}
    {{ Post: obj = cons(x, obj₀) and this.last = last(this.list) }}
  };
```

O(1) version, but more complex reasoning (two branches)

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {

  cons = (x: number): void => {
    if (this.list === nil)
      this.last = x;
    this.list = cons(x, this.list);
    {{ this.list = cons(x, this.list₀) and this.list₀ = nil and this.last = x }}
    {{ Post: obj = cons(x, obj₀) and this.last = last(this.list) }}
  };
```

Case "then":

last(this.list)	$=$ last(cons(x, this.list$_0$))	**since** this.list $=$ cons(x, ...)
	$=$ last(cons(x, nil))	**since** this.list$_0$ $=$ nil
	$=$ x	**def of** last
	$=$ this.last	**since** x $=$ this.last

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {

  cons = (x: number): void => {
    if (this.list === nil)
      this.last = x;
    this.list = cons(x, this.list);
    {{ this.list = cons(x, this.list_0) and this.list_0 ≠ nil and this.last = this.last_0 }}
    {{ Post: obj = cons(x, obj_0) and this.last = last(this.list) }}
  };
```

Case "else":

last(this.list)	$=$ last(cons(x, this.list$_0$))	**since** this.list $=$ cons(x, ...)
	$=$ last(this.list$_0$)	**since** this.list$_0 \neq$ nil
	$=$ this.last$_0$	**by RI**
	$=$ this.last	**since** this.last $=$ this.last$_0$

Moral of the Story for Level 3

- More mutation gave us better efficiency
  - saved memory
  - immutable version could be just as fast (level 1)

- More mutation means more complex reasoning
  - more facts to keep track of
  - more ways to make mistakes
  - more work to make sure we did it right

Recall: Immutable Queue ADT

- A queue is a list that can *only* be changed two ways:
  - add elements to the front
  - remove elements from the back

```
// List that only supports adding to the front and
// removing from the end
interface NumberQueue {

  // @returns len(obj)
  size: () => number;

  // @returns cons(x, obj)
  enqueue: (x: number) => NumberQueue;

  // @requires len(obj) > 0
  // @returns (x, Q) with obj = concat(Q, cons(x, nil))
  dequeue: ()=> [number, NumberQueue];
}
```

observer

producer

producer

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: Implementing a Queue with Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = concat(this.front, rev(this.back))
  // RI: if this.back = nil, then this.front = nil
  readonly front: List;
  readonly back: List;

  // makes obj = concat(front, rev(back))
  constructor(front: List, back: List) {
    …
  }
```

- Queue was in two parts, front and back
  - back stored in reverse order
  - full list was concat(this.front, rev(this.back))

Implementing Mutable Queue with Two Arrays

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

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

  // makes obj = vals
  constructor(vals: number[]) {
    this.front = [];
    this.back = vals;
  }
```

We should check this...

Implementing Mutable Queue with Two Arrays

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

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

  // makes obj = vals
  constructor(vals: number[]) {
    this.front = [];
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Implementing Mutable Queue with Two Arrays

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

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

  // makes obj = vals
  constructor(vals: number[]) {
    this.front = [];
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Is this really correct?

No way to say!

obj	$= \text{rev}(\text{this.front}) + \text{this.back}$	by AF
	$= \text{rev}([]) + \text{this.back}$	since this.front = []
	$= [] + \text{this.back}$	def of rev
	$= \text{this.back} = \text{vals}$	since this.back = vals

Implementing Mutable Queue with Two Arrays

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

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

  // makes obj = vals
  constructor(vals: number[]) {
    this.front = [];
    this.back = vals.slice(0, vals.length);
  }
```

- Must make a copy of the array!
  - then, we have the only reference to it (no aliases)

Implementing Mutable Queue with Two Arrays

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

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

  // @returns obj
  elements = (): number[] => {
    let revFront: number[] =
      this.front.slice(0, this.front.length);
    revFront.reverse();
    return revFront.concat(this.back);
  };
```

This is O(n)...

We can optimize it if front = [].

$$\mathrm{rev}([\,]) \mathbin{+\!\!+} \mathrm{this.back} = [\,] \mathbin{+\!\!+} \mathrm{this.back} = \mathrm{this.back}$$

Implementing Mutable Queue with Two Arrays

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

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

  // @returns obj
  elements = (): number[] => {
    if (this.front.length === 0) {
      return this.back;   // O(1) when this.front = []
    } else {
      let revFront: number[] =
        this.front.slice(0, this.front.length);
      revFront.reverse();
      return revFront.concat(this.back);
    }
  };
```

Is this correct?

No way to say!

Implementing Mutable Queue with Two Arrays

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

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

  // @returns obj
  elements = (): number[] => {
    let revFront: number[] = this.front.slice(0);
    revFront.reverse();
    return revFront.concat(this.back);
  };
```

- Cannot return an alias to `this.back`
  - must make a copy in all cases

Avoiding Representation Exposure

- Prevent aliasing of mutable state
  - otherwise, code outside your class can break it

- Options for avoiding representation exposure:

  1. Use immutable types

     lists are immutable, so you can freely accept and return them

  2. Copy In, Copy Out

     store copies of mutable values passed to you

     return copies of not aliases to mutable state

     don’t take their word that they haven’t kept an alias

- Professionals are untrusting about aliases