# CSE 331
# Software Design & Implementation

Spring 2025

Section 10 – Final Review

# Administrivia

- Final
  - **Tuesday, 6/10, Kane 130 from 12:30 - 2:20**
  - **Please arrive a couple minutes early**
  - **Bring your id**
  - **No notecards, all needed definitions will be included**

- Final review session
  - **5:00-6:30pm, Monday 6/9**
  - **TA Breakout Floors 3, 4, and 5 and room 403 in Allen**
  - Bring questions related to practice exams or general concepts
  - More details coming in Ed announcement

# Administrivia

HW 9

- Due 11pm Friday, 6/6 (but your final is on Tuesday so finish early and study if possible!)
  - Saturday 11pm if using late day
  - Make sure to run the linter on your code!

  - ( Tiny tip for testing shortest path method: make both people meet at the same endpoint (same building) so you can know the exact lat/long :) )
  - ( Other tiny tip –for the final really. Testing requires coverage of all branches, but it’s okay if coverage for a branch is achieved on an iteration *after* the first iteration ).

# Course Evals!!

- Please fill them out!
- We appreciate the feedback
  - We will actually read them, so any suggestions will be considered!
  - Everyone should have received an email with the links

# Final Focus Topics From Lecture

- Proving code correctness
- Implementing TS functions according to a spec (small)
- Writing tests for code (using testing heuristics)
- Broader conceptual questions on all course topics (incl. debugging, client--server programming, OOP, etc)

# Longer List of General Final Topics

- Reasoning about Recursion
- Reasoning about Loops and Tail Recursion
- Writing Methods
- Testing
- Writing the code of a for loop, given the loop idea and invariant.
- Writing or proving correct the methods of classes that implement mutable or immutable ADTs
- Small questions on any other topics (all content is fair game)
- Proof by Calculation
- Structural Induction

← these two are **very** important

- One practice finals and one practice midterm are on the course website under Syllabus>>Exam Mechanics (2nd practice final coming soon!)

# ADT

- **`MutableIntCursor` ADT** represents a list of integers with the ability to insert new characters at the “cursor index” within the list.
  - cursor index can be moved forward or backward

- **`LineCountingCursor`** implements `MutableIntCursor` by:
  - using the abstract state (an index and a list of values) as its concrete state
  - + records the number of newline characters (so class can easily, quickly determine the number of lines in the text)

- **Reminder**: familiar functions on last page of WS!

# ADT Comprehension Cursor

Let’s take a second to understand the ADT…

Imagine we have a LineCountingCursor, ourLCC, which is (1, [3, 3, 1]).

Where is the cursor in [3, 3, 1]?

# ADT Comprehension Cursor

Let’s take a second to understand the ADT…

Imagine we have a LineCountingCursor, ourLCC, which is (1, [3, 3, 1]).

Where is the cursor in [3, 3, 1]?

# ADT Comprehension Insert Method

What would happen if we called ourLCC.insert(0)?

(1, [3, 3, 1]) (2, [3, 0, 3, 1])
[3, 3, 1] → [3, 0, 3, 1]

Looking at the effects tag and our AF, since we know obj0 = (1, [3, 3, 1])
Then obj = (1+1, concat([3], 0::[3, 1])
→ obj = (2, [3, 0, 3, 1])

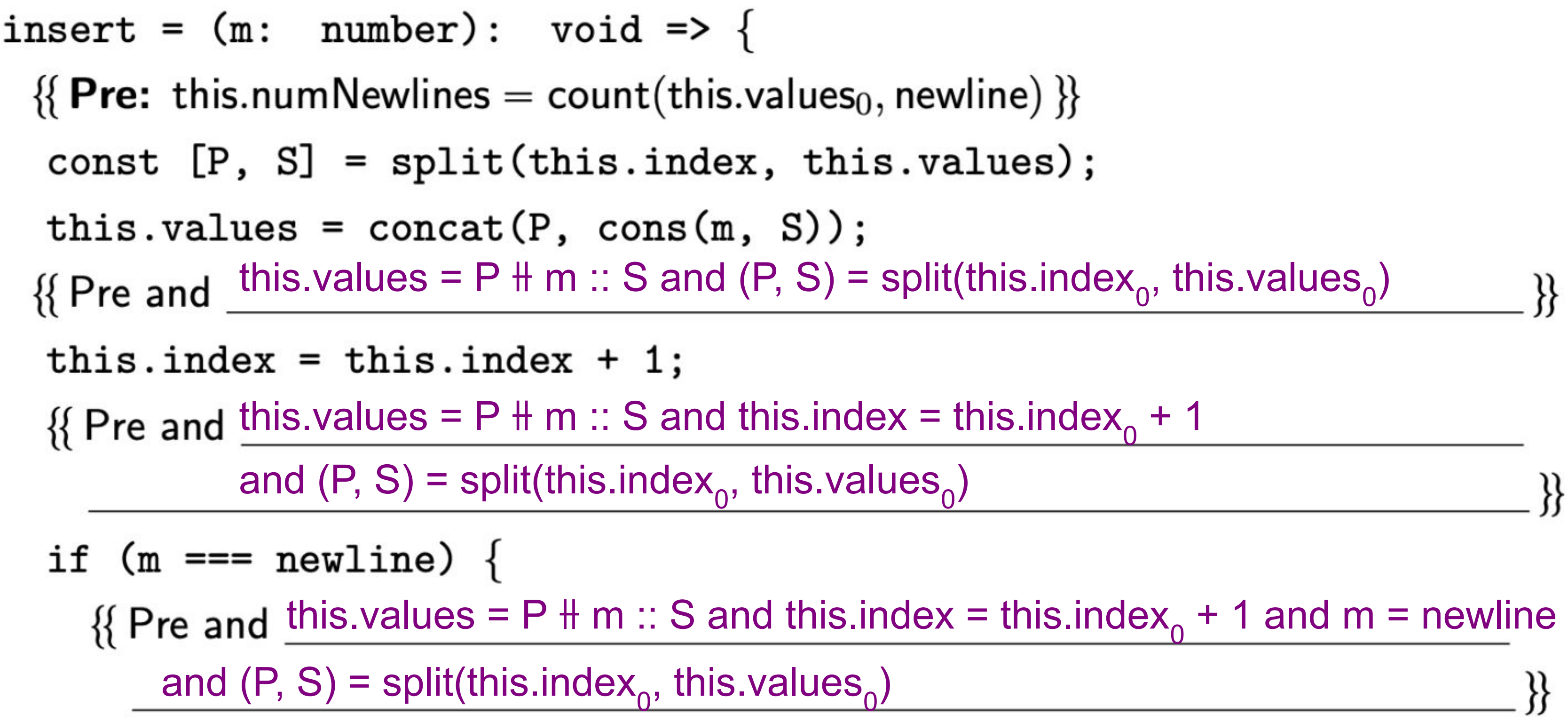
Our RI still holds because 0 <= 1 <= 3 → 0 <= 2 <= 4

# Problem 1a

Look at the code in the worksheet which claims to implement `insert` in `LineCountingCursor`. Use **forward reasoning** to fill in the blank assertions above, which go into the “then”

branch of the if statement.

# Problem 1a



```
insert = (m:  number):  void => {
```

{{ **Pre:** this.numNewlines = count($\text{this.values}_0$, newline) }}

```
  const [P, S] = split(this.index, this.values);
  this.values = concat(P, cons(m, S));
```

{{ Pre and this.values = P ⧺ m :: S and (P, S) = split($\text{this.index}_0$, $\text{this.values}_0$) }}

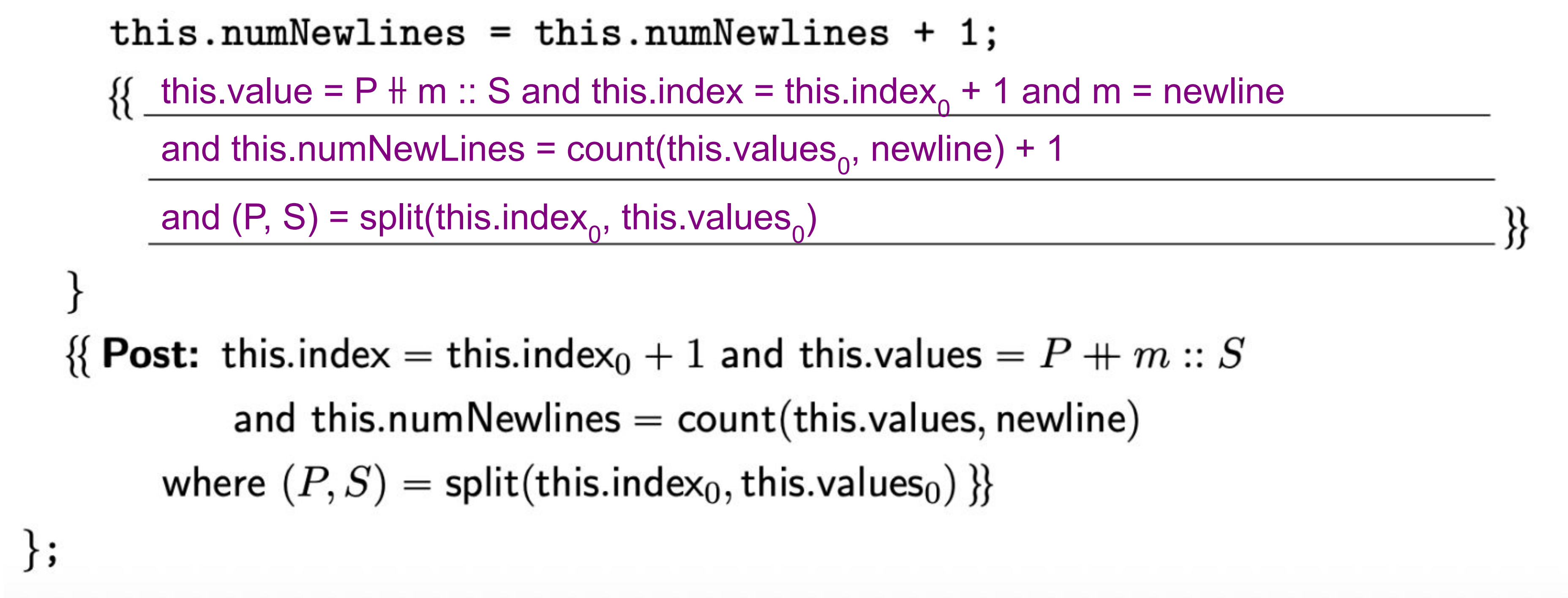
```
  this.index = this.index + 1;
```

{{ Pre and this.values = P ⧺ m :: S and this.index = $\text{this.index}_0$ + 1
and (P, S) = split($\text{this.index}_0$, $\text{this.values}_0$) }}

```
  if (m === newline) {
```

{{ Pre and this.values = P ⧺ m :: S and this.index = $\text{this.index}_0$ + 1 and m = newline
and (P, S) = split($\text{this.index}_0$, $\text{this.values}_0$) }}

# Problem 1a



```
this.numNewlines = this.numNewlines + 1;
```

{{ this.value = P ⧺ m :: S and this.index = this.index$_0$ + 1 and m = newline

and this.numNewLines = count(this.values$_0$, newline) + 1

and (P, S) = split(this.index$_0$, this.values$_0$) }}

}

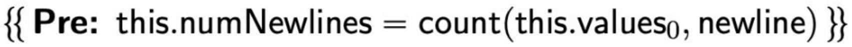
{{ **Post:** this.index $=$ this.index$_0 + 1$ and this.values $= P \mathbin{+\!\!+} m :: S$

and this.numNewlines $=$ count(this.values, newline)

where $(P, S) =$ split(this.index$_0$, this.values$_0$) }}

};

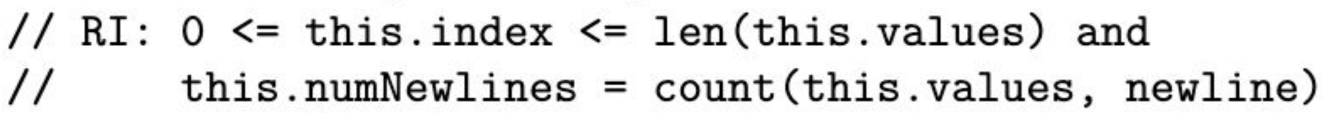
# Problem 1b



{{ **Pre:** this.numNewlines = count(this.values$_0$, newline) }}

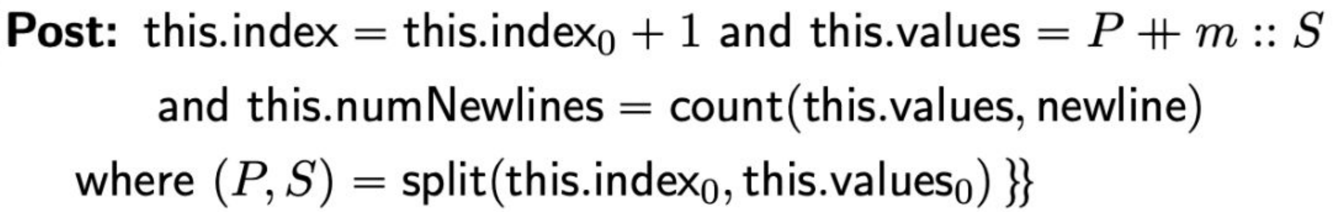
Explain, in English, why the facts listed in **Pre** will be true when the function is called:

- The fact from the representation invariant (RI), which we can assume to be true at the start of each method (before any fields are mutated)



```
// RI: 0 <= this.index <= len(this.values) and
//      this.numNewlines = count(this.values, newline)
```

# Problem 1c



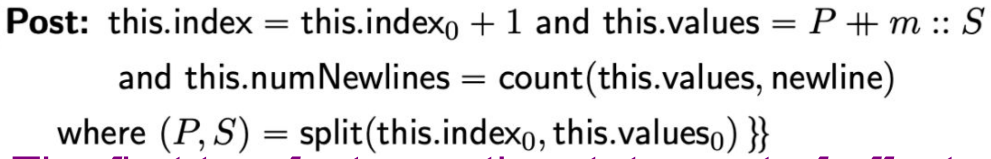
**Post:** $\text{this.index} = \text{this.index}_0 + 1$ and $\text{this.values} = P \mathbin{+\!\!+} m :: S$

and $\text{this.numNewlines} = \text{count}(\text{this.values}, \text{newline})$

where $(P, S) = \text{split}(\text{this.index}_0, \text{this.values}_0)$ }}

Explain, in English, why the facts listed in **Post** need to be true when the function completes in order for insert to be complete:

# Problem 1c

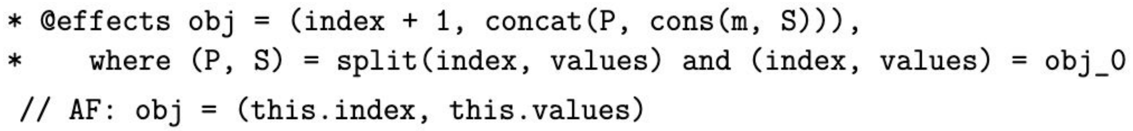


**Post:** $\text{this.index} = \text{this.index}_0 + 1$ and $\text{this.values} = P \mathbin{+\!\!+} m :: S$

and $\text{this.numNewlines} = \text{count}(\text{this.values}, \text{newline})$

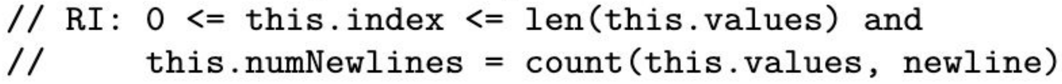
where $(P, S) = \text{split}(\text{this.index}_0, \text{this.values}_0)$ }}

- The first two facts are the statement of effects clause of the spec after we apply the abstraction function:
  - "index" part of abstract state is stored in `this.index` field
  - "values" part of abstract state is stored in `this.values` field.



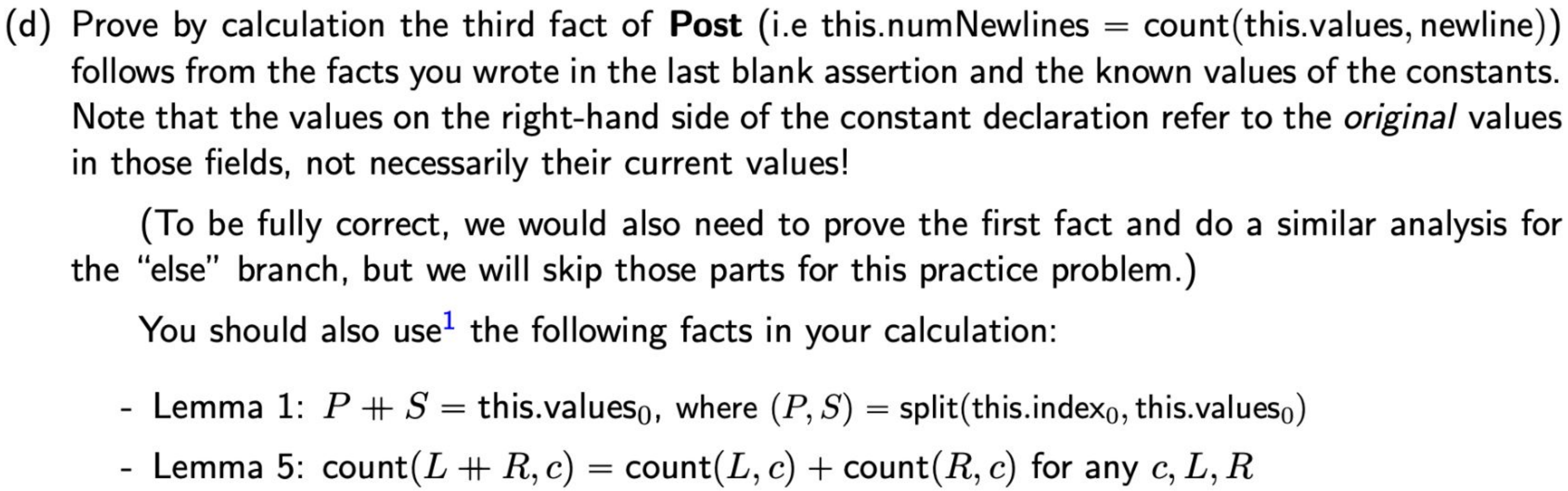
```
* @effects obj = (index + 1, concat(P, cons(m, S))),
*    where (P, S) = split(index, values) and (index, values) = obj_0
 // AF: obj = (this.index, this.values)
```

- The last fact is required by the representation invariant, which must be checked at the end of any mutator method.



```
// RI: 0 <= this.index <= len(this.values) and
//       this.numNewlines = count(this.values, newline)
```

# Problem 1d



(d) Prove by calculation the third fact of **Post** (i.e this.numNewlines = count(this.values, newline)) follows from the facts you wrote in the last blank assertion and the known values of the constants. Note that the values on the right-hand side of the constant declaration refer to the *original* values in those fields, not necessarily their current values!

(To be fully correct, we would also need to prove the first fact and do a similar analysis for the "else" branch, but we will skip those parts for this practice problem.)

You should also use[1] the following facts in your calculation:

- Lemma 1: $P \mathbin{+\!\!+} S = \text{this.values}_0$, where $(P, S) = \text{split}(\text{this.index}_0, \text{this.values}_0)$
- Lemma 5: $\text{count}(L \mathbin{+\!\!+} R, c) = \text{count}(L, c) + \text{count}(R, c)$ for any $c, L, R$

# Problem 1d

We can prove this fact as follows:

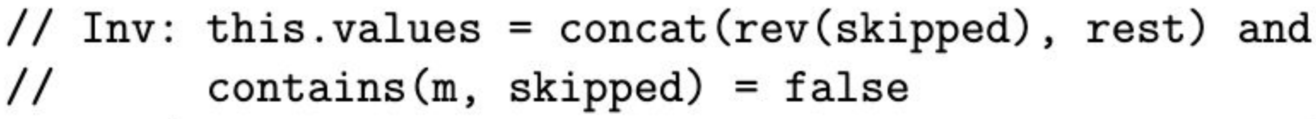
count(this.values, newline)

| | |
|---|---|
| = count(P ⧺ m::S, newline) | since this.values = . . . |
| = count(P, newline) + count(m::S, newline) | by Lemma 5 |
| = count(P, newline) + count(S, newline) + 1 | def of count |
| = count((P ⧺ S), newline) + 1 | by Lemma 5 |
| = count($\text{this.values}_0$, newline) + 1 | by Lemma 1 |
| = this.numNewlines | since this.numNewlines = |

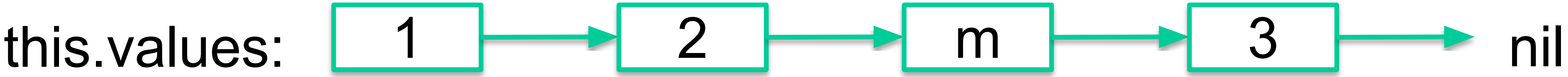
# Problem 2

- Fill in the missing parts of the method so it is correct with the *given invariant*
- **Loop idea:**
  - skip past elements in `this.values` until we reach one that equals the given number or we hit the end
- **Invariant:**
  - `this.values` is split up between skipped and rest, with skipped being the front part in reverse order
  - no element of skipped is equal to the number m
- Do not write any other loops or call any other methods. The only list functions that should be needed are `cons` and `len`

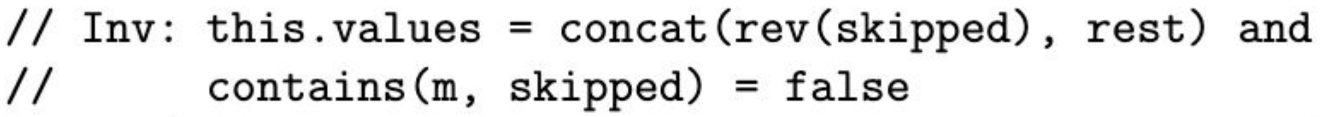
# Problem 2



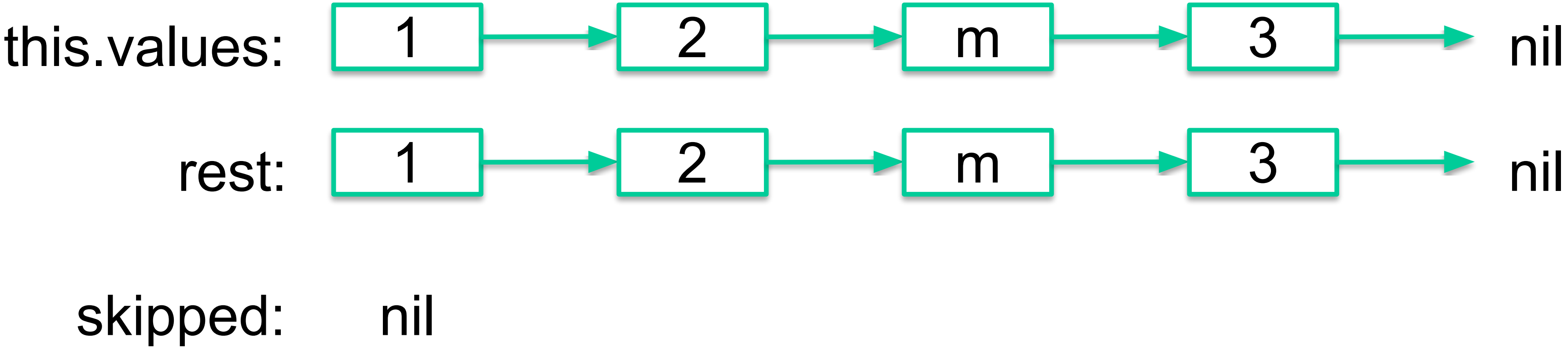
```
// Inv: this.values = concat(rev(skipped), rest) and
//        contains(m, skipped) = false
```



# Problem 2

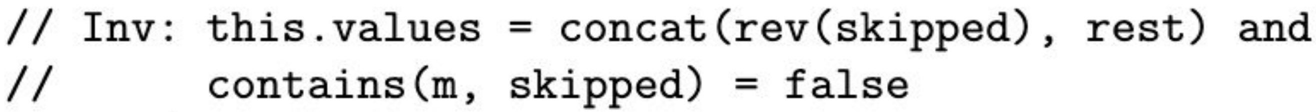


```
// Inv: this.values = concat(rev(skipped), rest) and
//      contains(m, skipped) = false
```

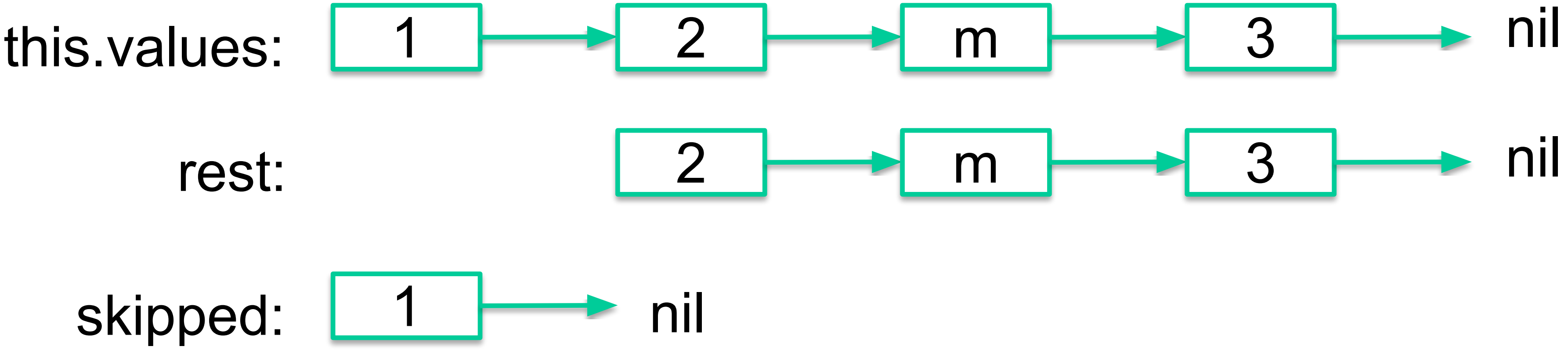


Easiest way to satisfy the invariant

# Problem 2



```
// Inv: this.values = concat(rev(skipped), rest) and
//         contains(m, skipped) = false
```



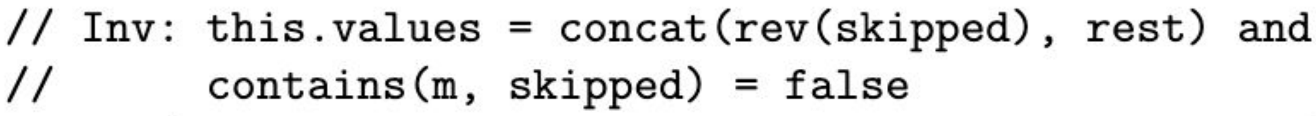
this.values: 1 → 2 → m → 3 → nil

rest: 2 → m → 3 → nil

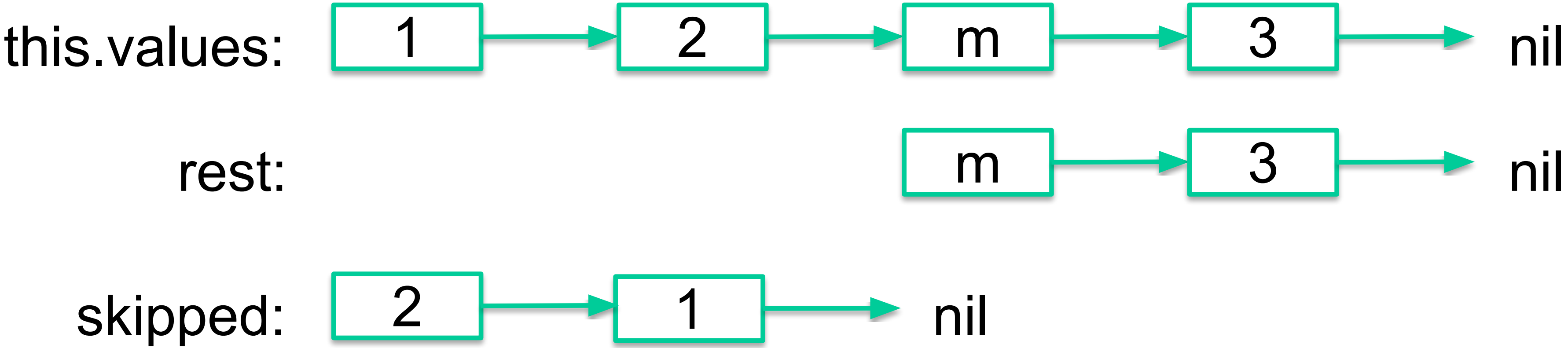
skipped: 1 → nil

While rest.hd != m (need to check rest != nil first),
remove and append rest.hd to skipped
(cons adds to front which reverses the list which matches the invariant)

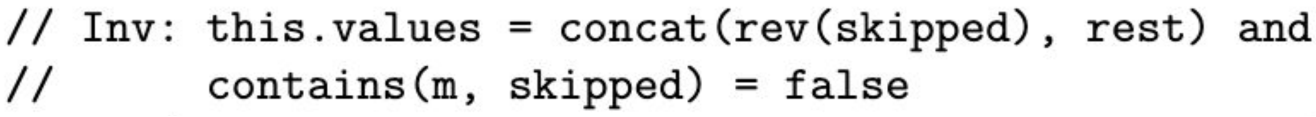
# Problem 2



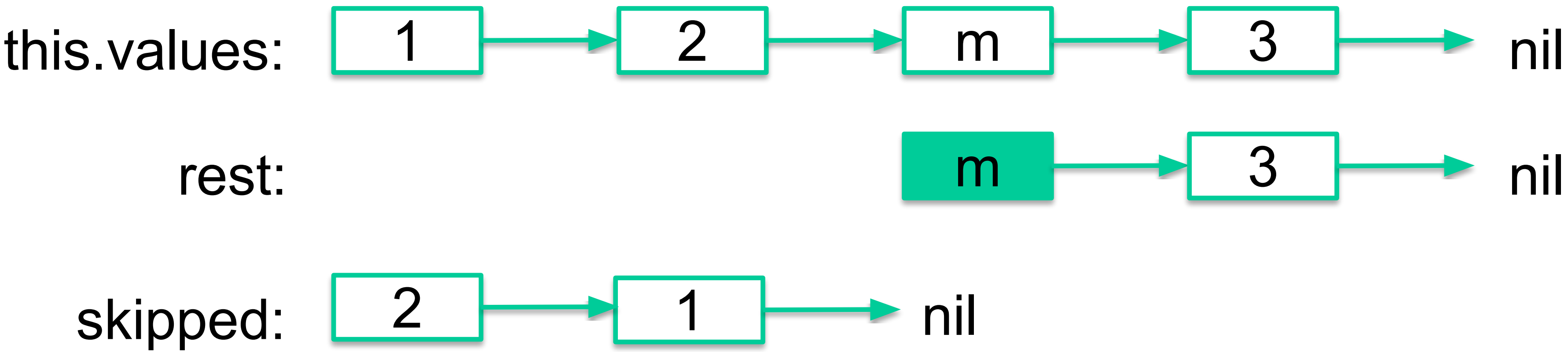
```
// Inv: this.values = concat(rev(skipped), rest) and
//      contains(m, skipped) = false
```



# Problem 2



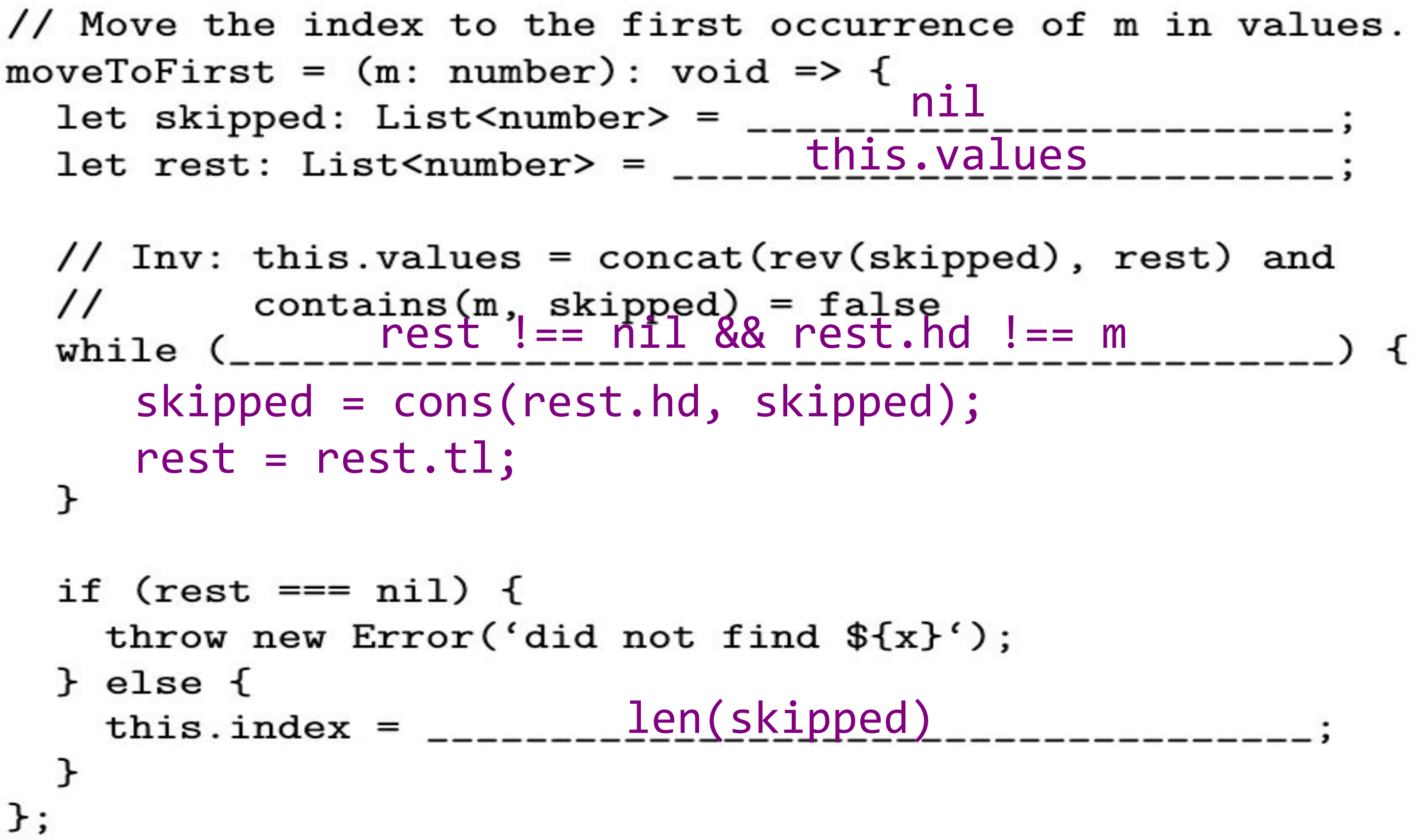
```
// Inv: this.values = concat(rev(skipped), rest) and
//       contains(m, skipped) = false
```



When we exit the loop

- If rest = nil then we didn’t find m
- Otherwise, Index of m is the length of the skipped list

# Problem 2



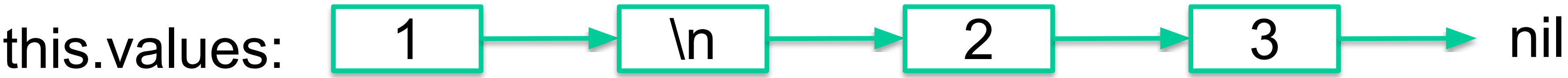
```
// Move the index to the first occurrence of m in values.
moveToFirst = (m: number): void => {
  let skipped: List<number> = _______nil_______________________;
  let rest: List<number> = _____this.values__________________;

  // Inv: this.values = concat(rev(skipped), rest) and
  //        contains(m, skipped) = false
  while (______rest !== nil && rest.hd !== m_____________) {
     skipped = cons(rest.hd, skipped);
     rest = rest.tl;
  }

  if (rest === nil) {
    throw new Error(‘did not find ${x}‘);
  } else {
    this.index = ________len(skipped)__________________;
  }
};
```

# Problem 3

- Fill **`removeNextLine`** so it removes all the text on the next line: text between the *first* and *second* newline characters *after* the cursor index
    - remove second newline, but leave cursor index in place
    - If there are no newlines after cursor, then do nothing
    - If there is only one newline after cursor, remove all text after it

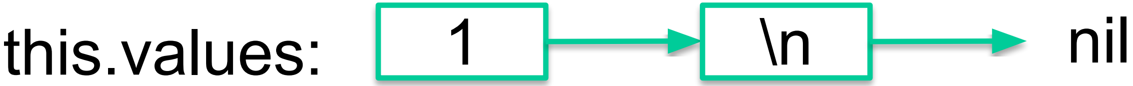
# removeNextLine Example

Imagine we have the list:



this.values: 1 → \n → 2 → 3 → nil

Assume our cursor is at index = 0, what would this.values be after the call LCC.removeNextLine()?



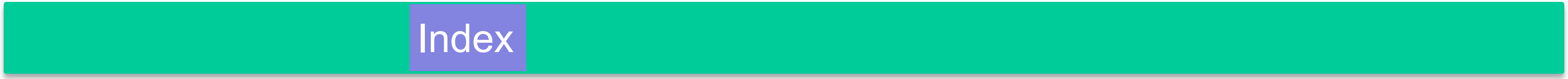
this.values: 1 → \n → nil

# Problem 3 Visualization

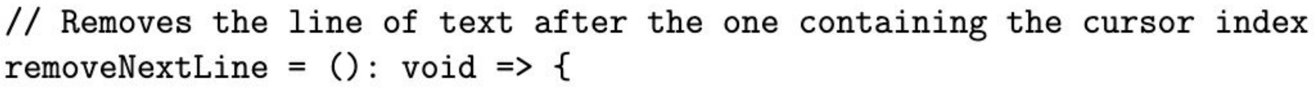
Take a moment to draw out the values list and what it will look like when it is split at the cursor.

# Problem 3

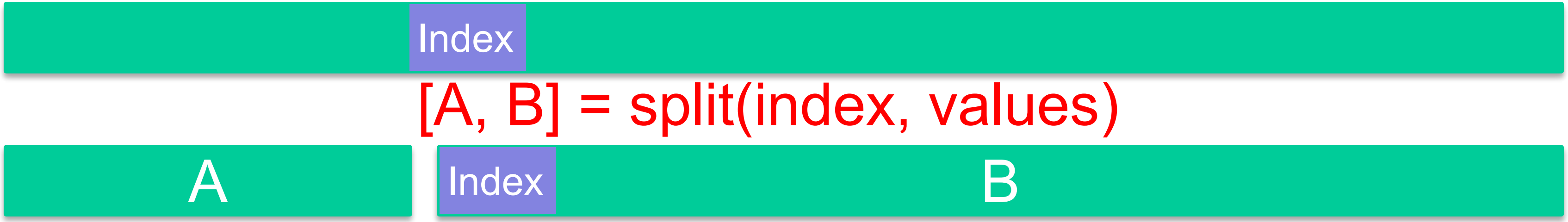
```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
```



# Problem 3



```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
```



[A, B] = split(index, values)

# Problem 3 Cases

Now that we see how our values list looks after we split it.

How many different cases do we have?

2 Cases (each time we split):

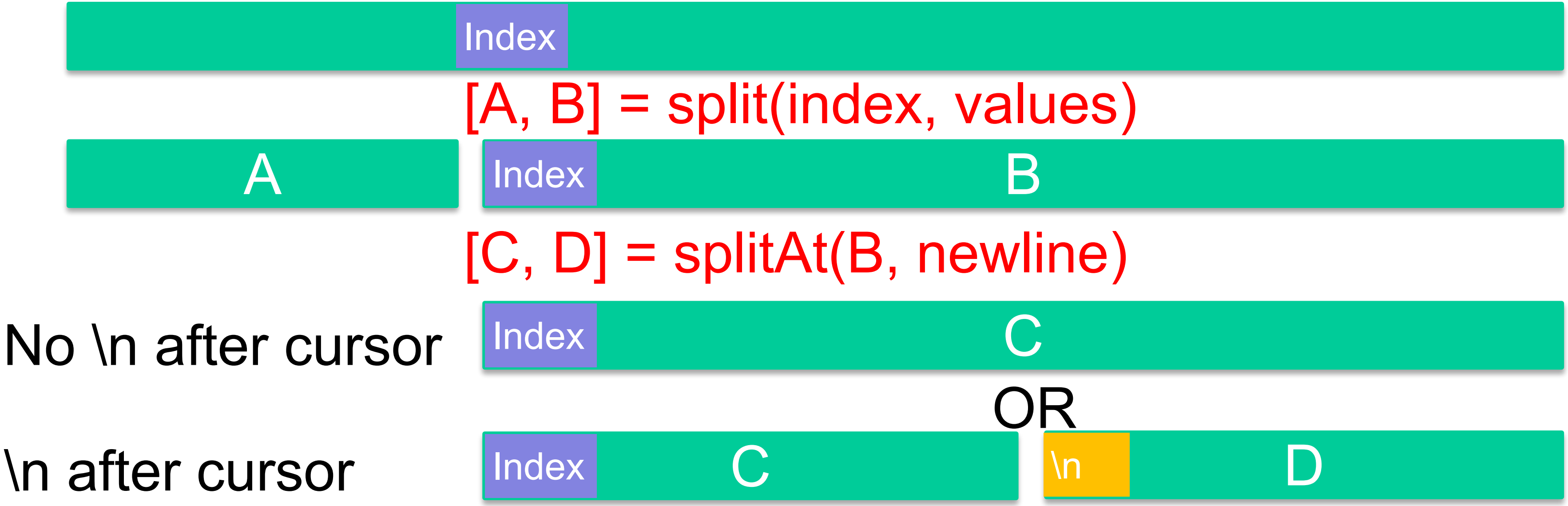
No ‘\n’ after the cursor

>= 1 ‘\n’ after the cursor

Now let’s draw out what we would do in each case…

# Problem 3

```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
```



[A, B] = split(index, values)

[C, D] = splitAt(B, newline)

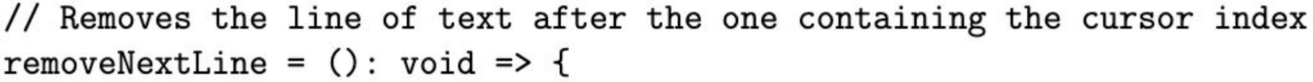
No \n after cursor

OR

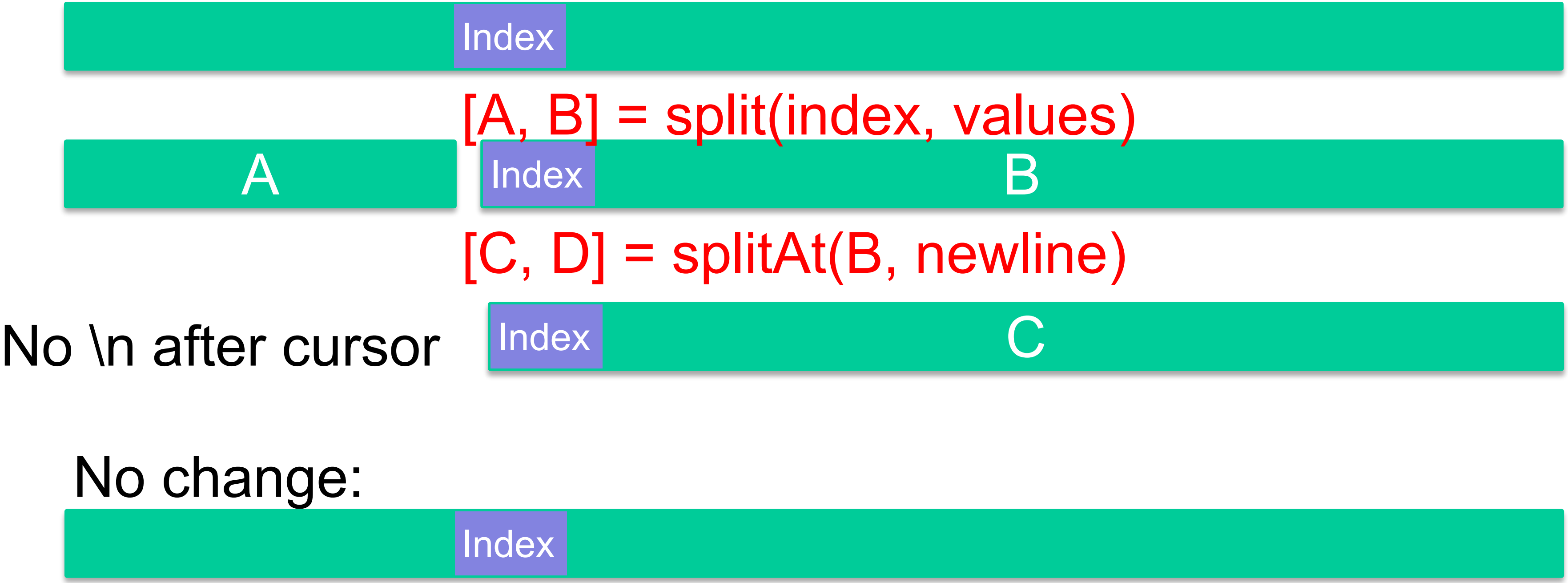
\n after cursor

hi

# Problem 3



```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
```



[A, B] = split(index, values)

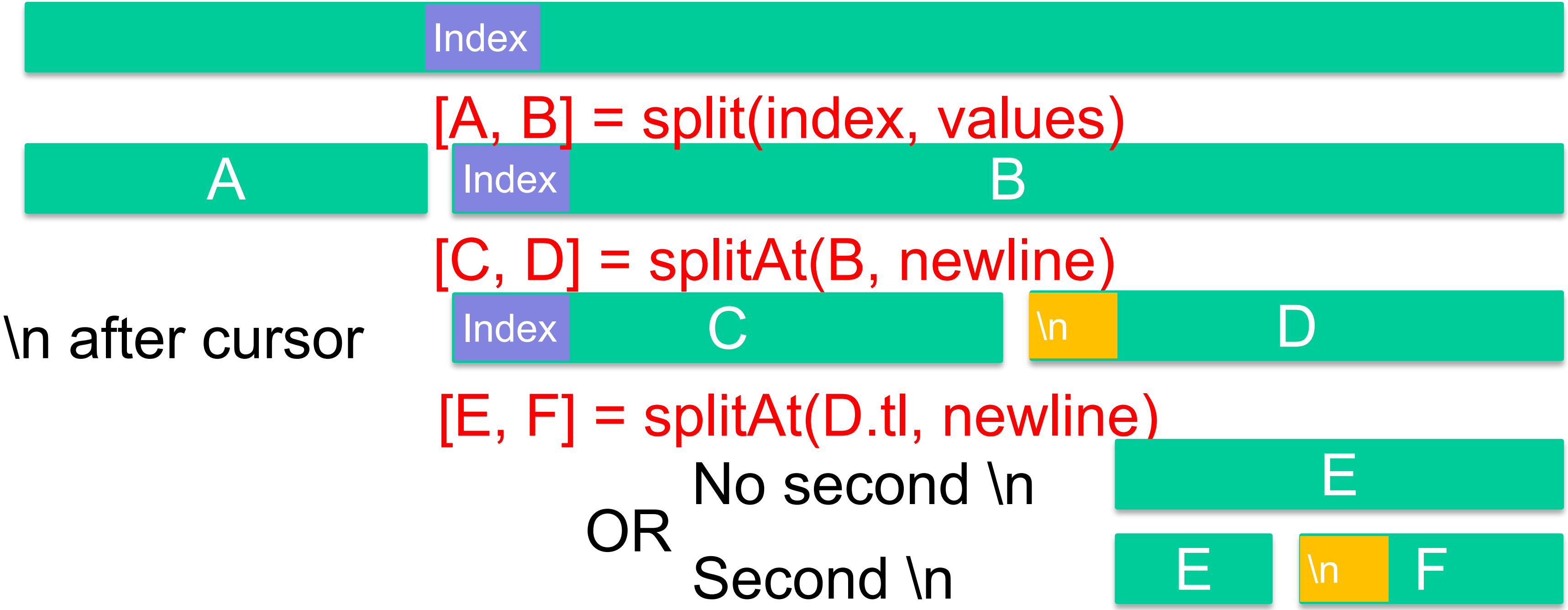
[C, D] = splitAt(B, newline)

No \n after cursor

No change:

# Problem 3

```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
```



[A, B] = split(index, values)

[C, D] = splitAt(B, newline)

\n after cursor

[E, F] = splitAt(D.tl, newline)

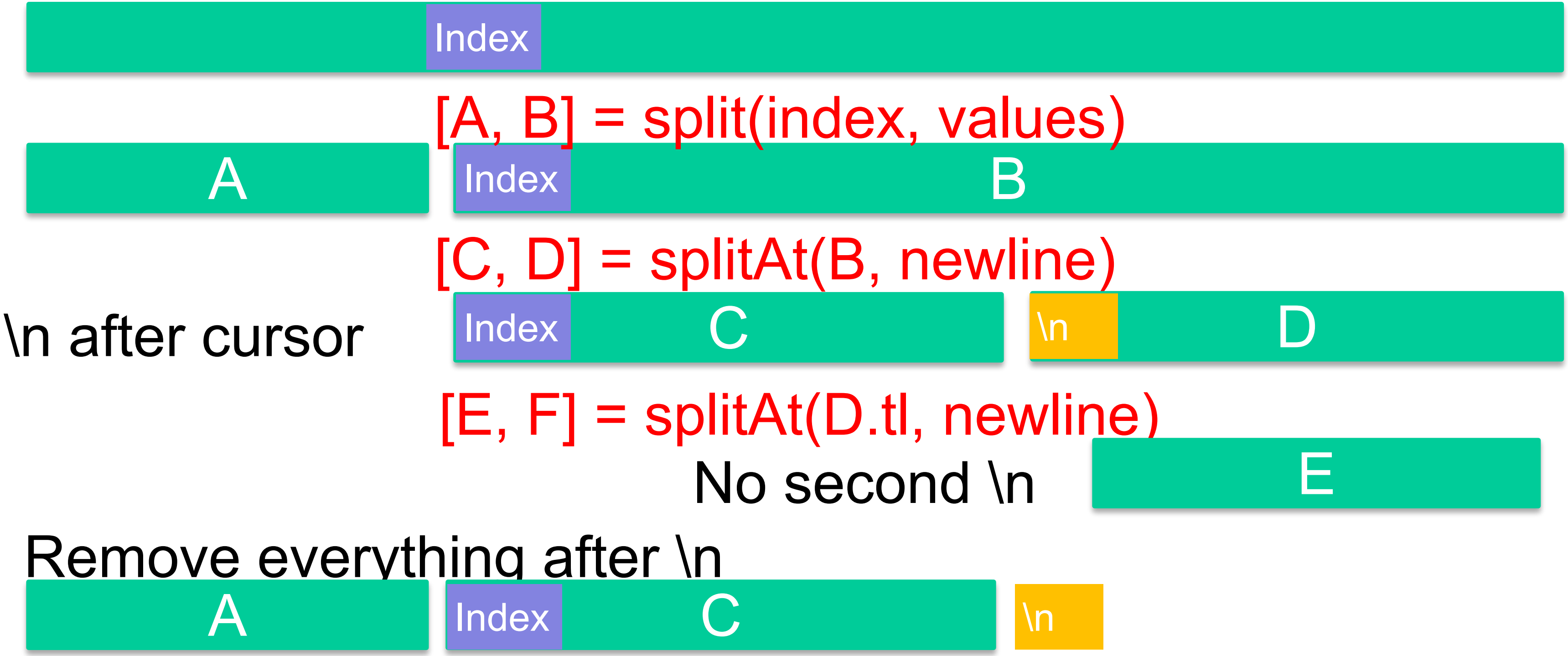
No second \n

OR

Second \n

# Problem 3

```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
```



[A, B] = split(index, values)

[C, D] = splitAt(B, newline)

\n after cursor

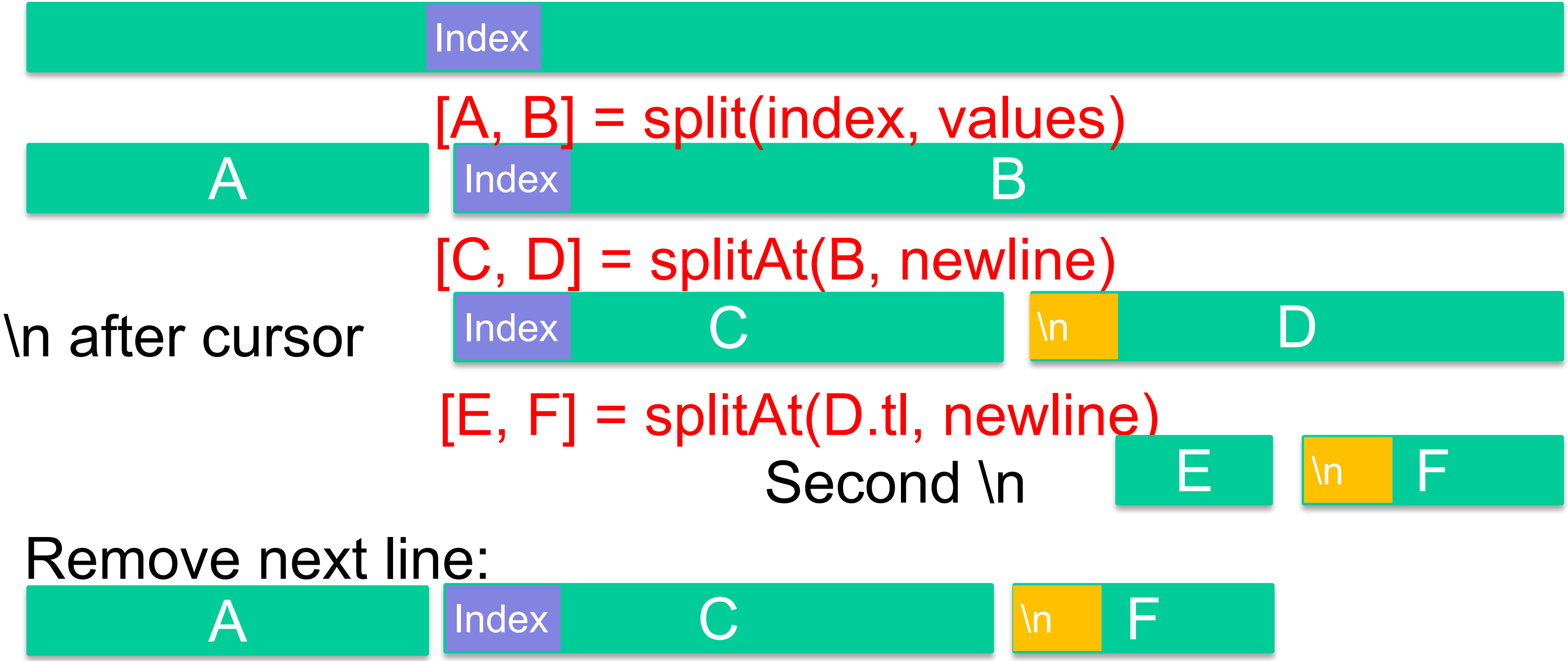
[E, F] = splitAt(D.tl, newline)

No second \n

Remove everything after \n

# Problem 3

```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
```



[A, B] = split(index, values)

[C, D] = splitAt(B, newline)

\n after cursor

[E, F] = splitAt(D.tl, newline)

Second \n

Remove next line:

# removeNextLine Hints

- removeNextLine is method of LineCountingCursor, so you can access `this.index` and `this.values`

- You can use any Familiar List Functions from final page and assume they've been translated to TS

- Hint: split-at function on the last page may be useful, assume the TS translation of it is called `splitAt`

# Problem 3

```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {
  const [A, B] = split(this.index, this.values);
  const [C, D] = splitAt(B, newline);
  if (D !== nil) {
    // after the newline
    const [E, F] = splitAt(D.tl, newline);
    if (F === nil) {
      this.values = concat(A, concat(C, cons(newline, nil)));
    } else {
      // drop one newline
      this.values = concat(A, concat(C, F));
      this.numNewLines = this.numNewlines - 1;
    }
  }
};
```

# You got this!

Puppy Dubs for good luck

https://tinyurl.com/331sp25secBD10