CSE 331
Software Design & Implementation

Autumn 2024

Section 10 – Final Review

Administrivia

- Final
  - **Tuesday, 12/10, BAG 131 and 154 from 12:30 - 2:20**
  - **Please arrive a couple minutes early**
  - **No notecards, all needed definitions will be included**
  - Assigned rooms based on Last name, details incoming (watch Ed!)

- Final review session
  - **4pm, Monday 12/09**
  - **TA Breakout Floors in Allen (all of them)**
  - Bring questions related to practice exams or general concepts
  - More details coming in Ed announcement

Administrivia

HW 9

- Due 11pm Saturday, 12/7 (but your final is on Tuesday so finish early and study if possible!)
    - 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 :))

Course Evals!!

- Please fill them out!

- We appreciate the feedback
  - We will actually read them, so any suggestions will be considered!

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 ADTs
- Small questions on any other topics (all content is fair game)

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!

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: $\text{this.numNewlines}_0 = \text{count}(\text{this.values}_0, \text{newline})$ }}
  const [P, S] = split(this.index, this.values);
  this.values = concat(P, cons(m, S));
 {{ Pre and ________________________________________________ }}
  this.index = this.index + 1;
  {{ Pre and ________________________________________________
     ________________________________________________________ }}
  if (m === newline) {
    {{ Pre and _____________________________________________
       _____________________________________________________ }}
```

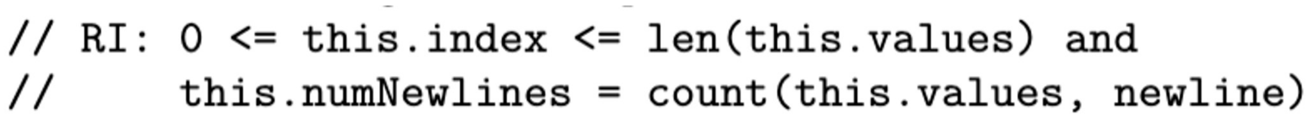
Problem 1a

```
    this.numNewlines = this.numNewlines + 1;
    {{ Pre and ________________________________________________

      ________________________________________________________

      ________________________________________________________ }}
  }
  {{ Post: this.index = $\text{this.index}_0 + 1$ and this.values = concat($P$, cons($m$, $S$))
             and this.numNewlines = count(this.values, newline)
         where ($P$, $S$) = split($\text{this.index}_0$, $\text{this.values}_0$) }}
};
```

Problem 1b

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

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



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

Problem 1c

$$
\begin{aligned}
\{\{\ \textbf{Post:}\ & \mathsf{this.index} = \mathsf{this.index}_0 + 1 \ \mathsf{and}\ \mathsf{this.values} = \mathsf{concat}(P, \mathsf{cons}(m, S)) \\
& \mathsf{and}\ \mathsf{this.numNewlines} = \mathsf{count}(\mathsf{this.values}, \mathsf{newline}) \\
& \mathsf{where}\ (P, S) = \mathsf{split}(\mathsf{this.index}_0, \mathsf{this.values}_0)\ \}\}
\end{aligned}
$$

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:** this.index $=$ this.index$_0 + 1$ and this.values $=$ concat$(P,$ cons$(m, S))$
and this.numNewlines $=$ count(this.values, newline)
where $(P, S) =$ split(this.index$_0$, this.values$_0$) $\}\}$

```
* @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)
```

```
// 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** 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:

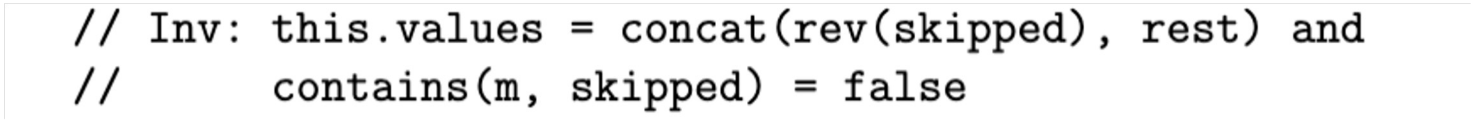
- Lemma 1: $\text{concat}(P, S) = \text{this.values}_0$, where $(P, S) = \text{split}(\text{this.index}_0, \text{this.values}_0)$
- Lemma 5: $\text{count}(\text{concat}(L, R), c) = \text{count}(L, c) + \text{count}(R, c)$ for any c, L, R

Problem 1d

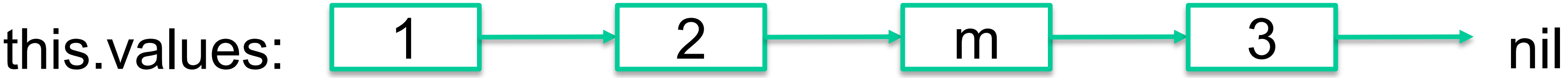
Problem 2

- 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
```



this.values:

1

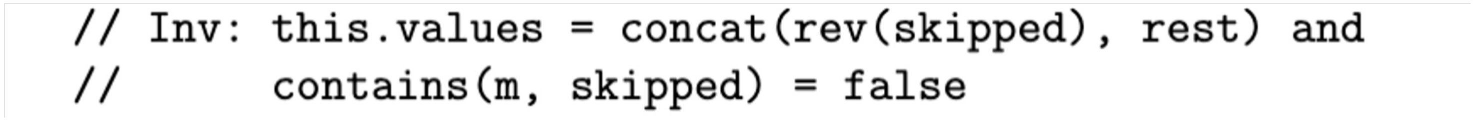
2

m

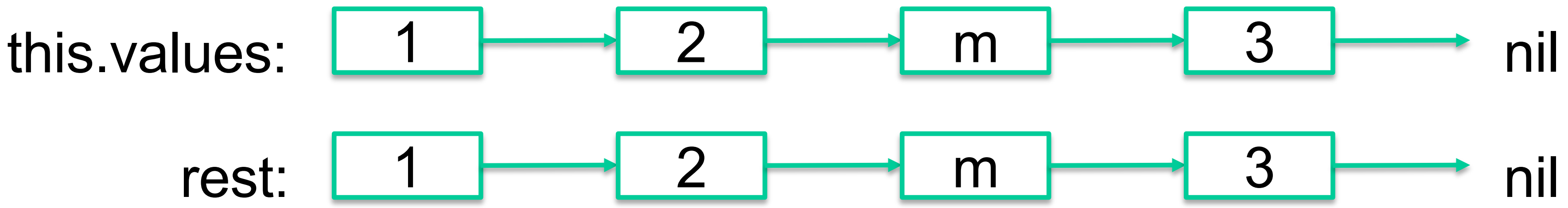
3

nil

Problem 2



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



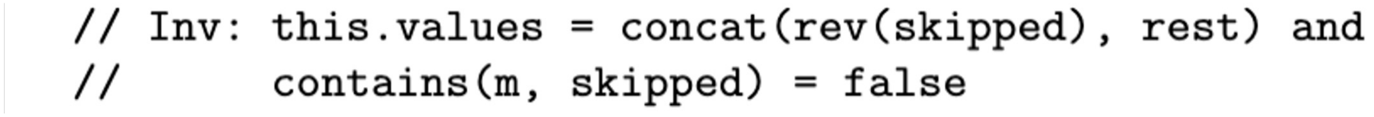
this.values: 1 → 2 → m → 3 → nil

rest: 1 → 2 → m → 3 → nil

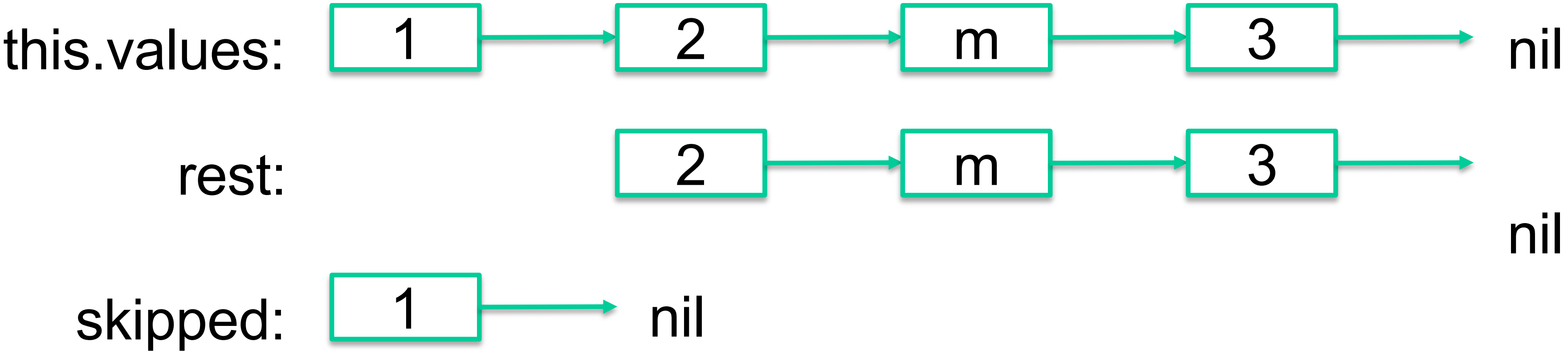
skipped: nil

Easiest way to satisfy the invariant

Problem 2



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



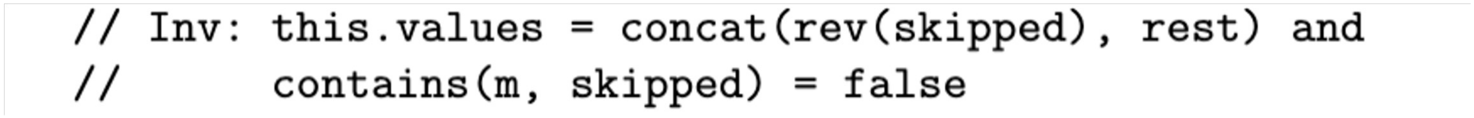
this.values: 1 → 2 → m → 3 → nil

rest: 2 → m → 3 → nil

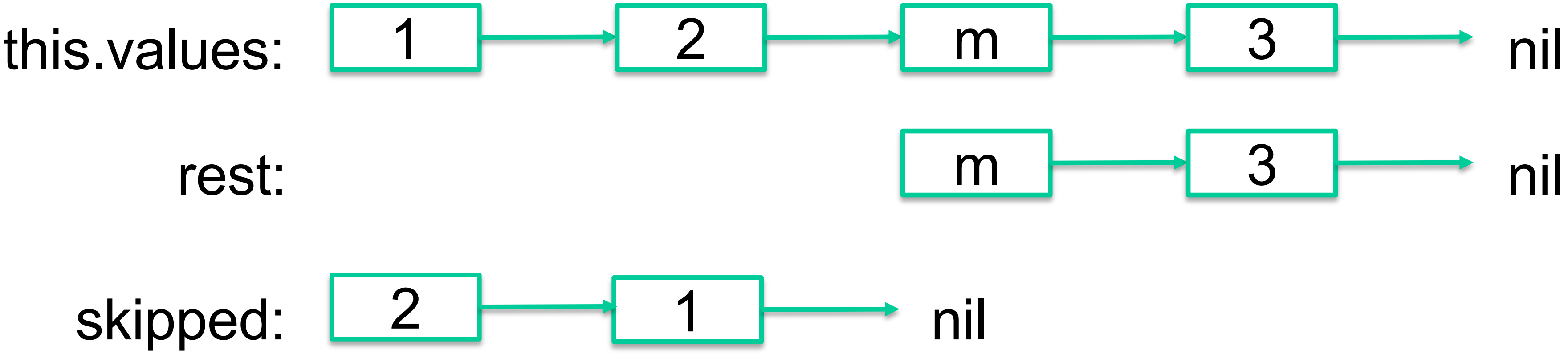
skipped: 1 → nil

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)

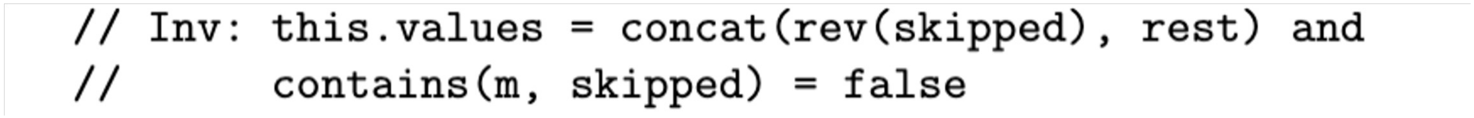
Problem 2



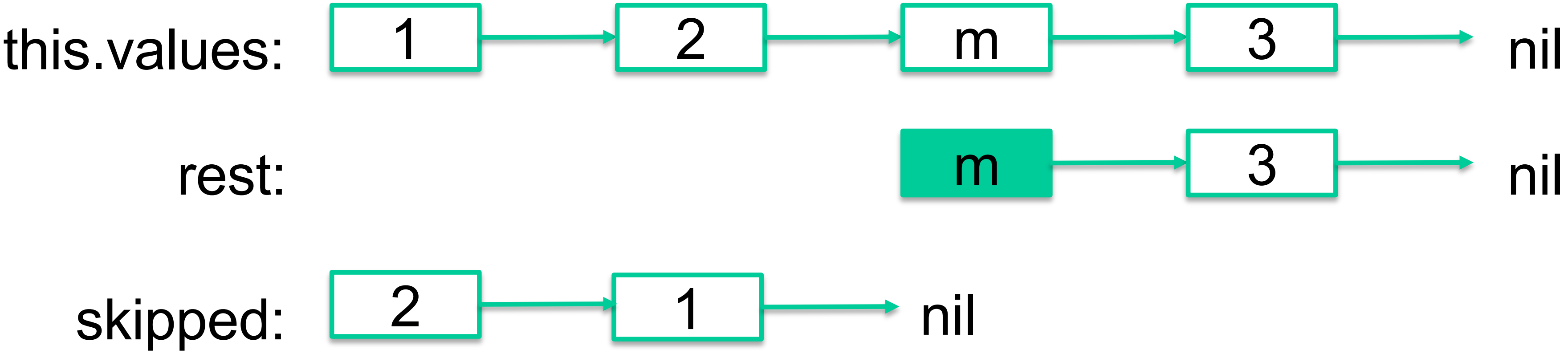
```
// Inv: this.values = concat(rev(skipped), rest) and
//        contains(m, skipped) = false
```



Problem 2



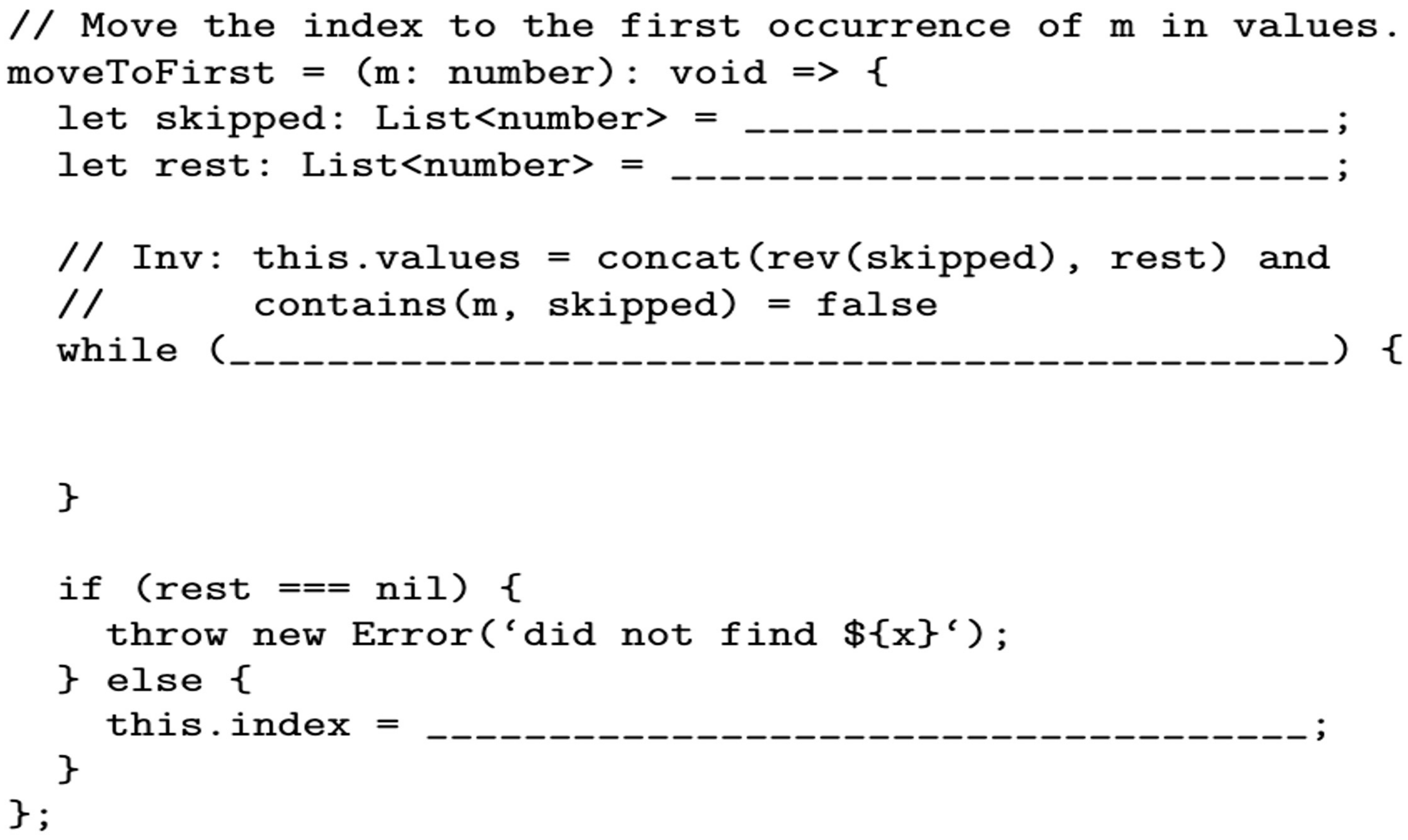
```
// Inv: this.values = concat(rev(skipped), rest) and
//      contains(m, skipped) = false
```



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



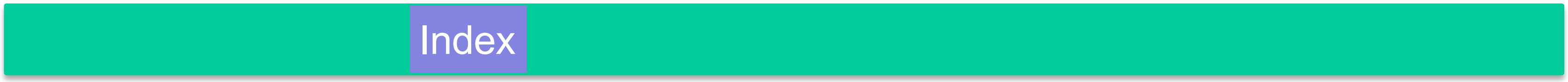
```
// Move the index to the first occurrence of m in values.
moveToFirst = (m: number): void => {
  let skipped: List<number> = ________________________;
  let rest: List<number> = ___________________________;

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


  }

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

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
- method of LineCountingCursor, so you can access `this.index` and `this.values`
- Can use any Familiar List Functions from final page and assume they've been translated to TS
- Hint: split-at function from HW5 may be useful, assume the TS translation of it is called `splitAt`

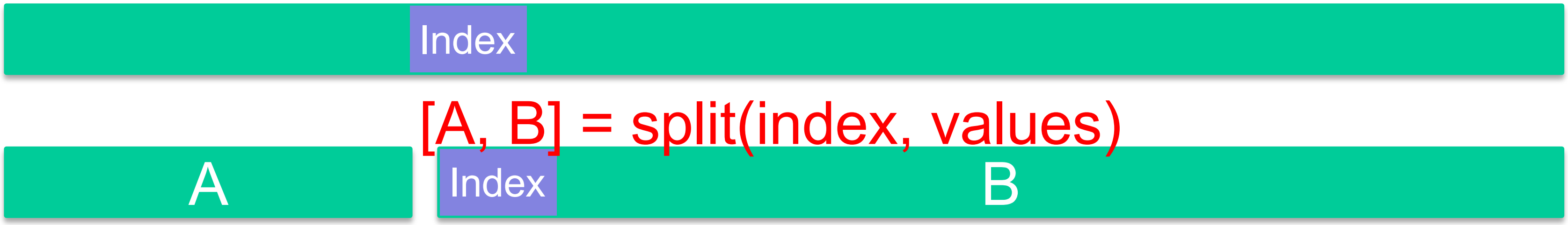
Problem 3

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



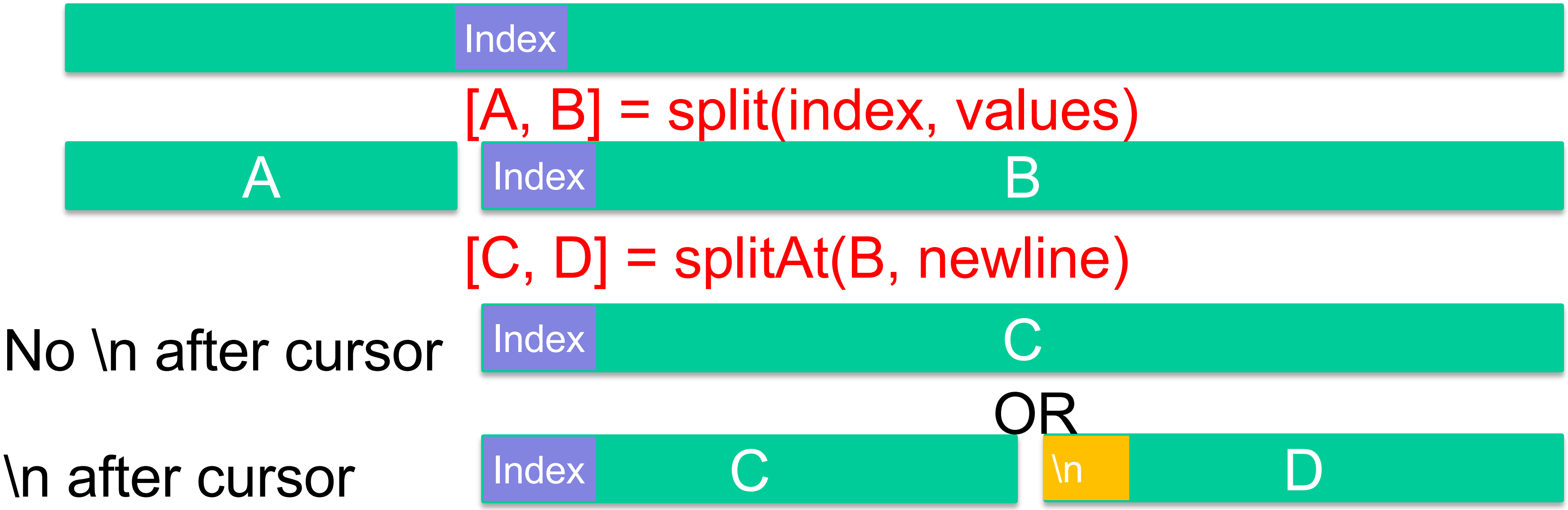
Problem 3

```
// 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 => {
```



Index

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

A

Index B

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

No \n after cursor

Index C

OR

\n after cursor

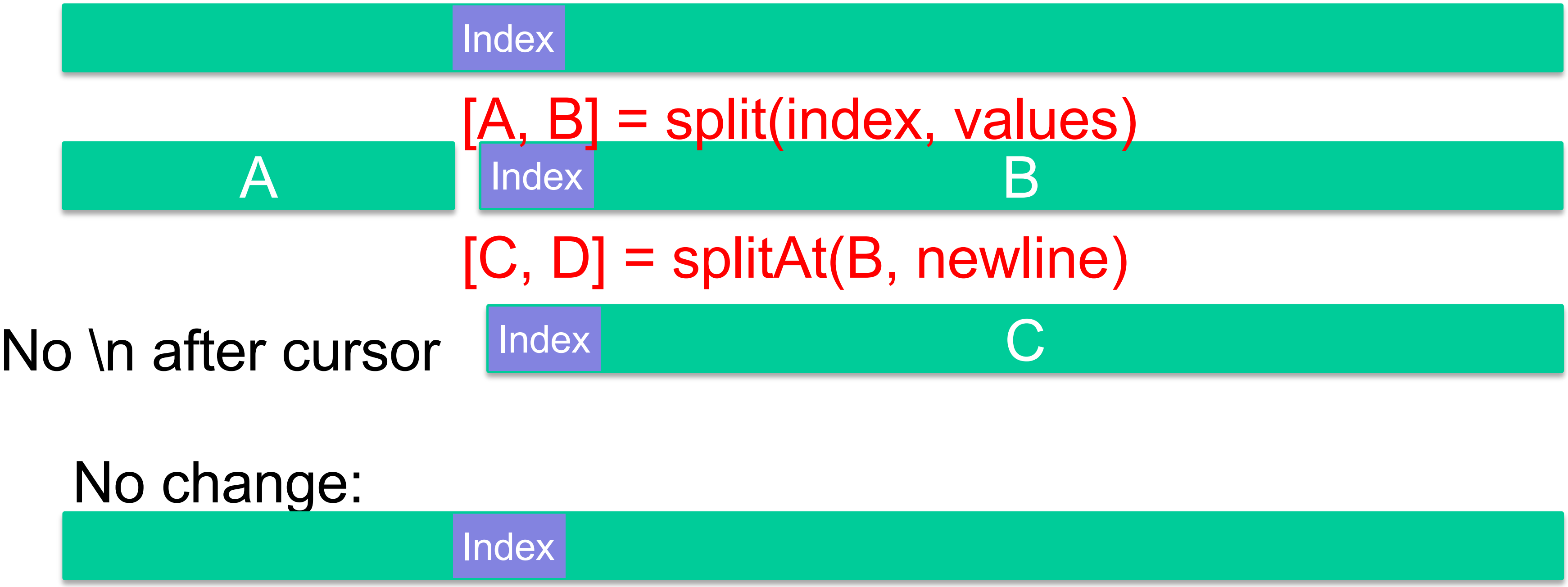
Index C

\n D

hi

Problem 3

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



Index

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

A

Index

B

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

No \n after cursor

Index

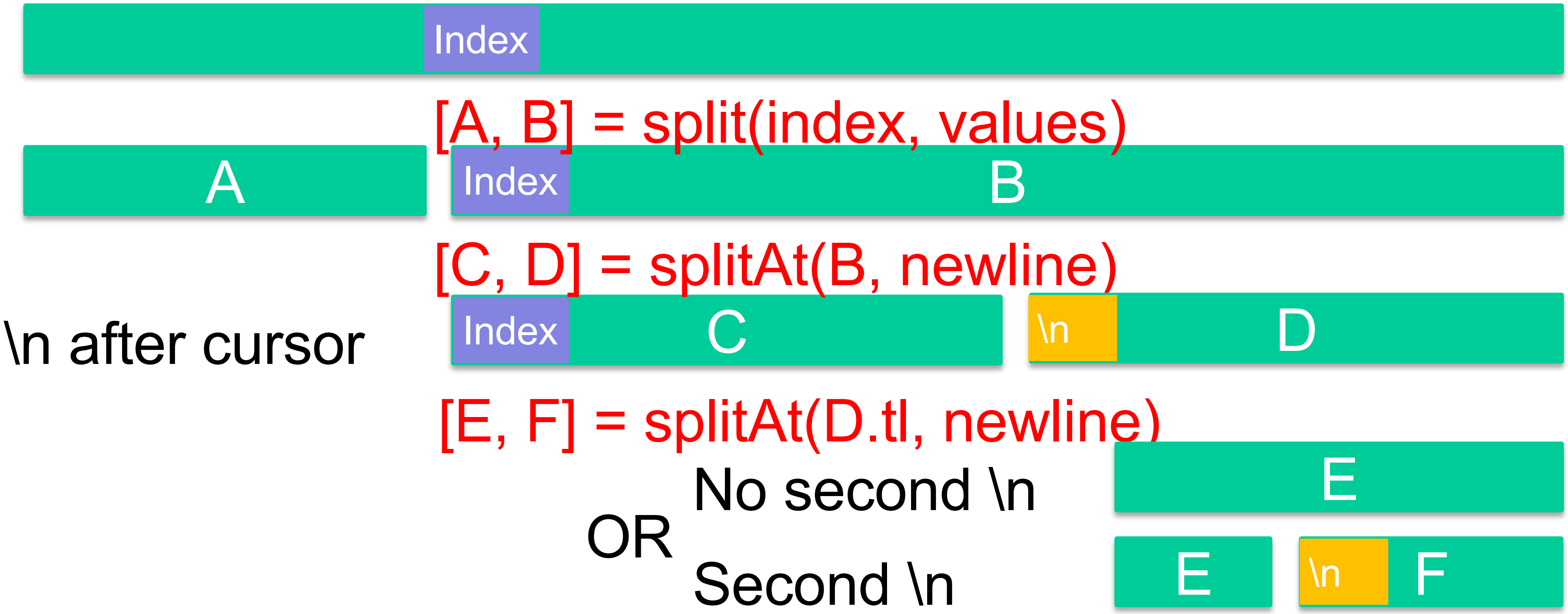
C

No change:

Index

Problem 3

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



Index

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

A

Index B

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

\n after cursor

Index C

\n D

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

No second \n

E

OR

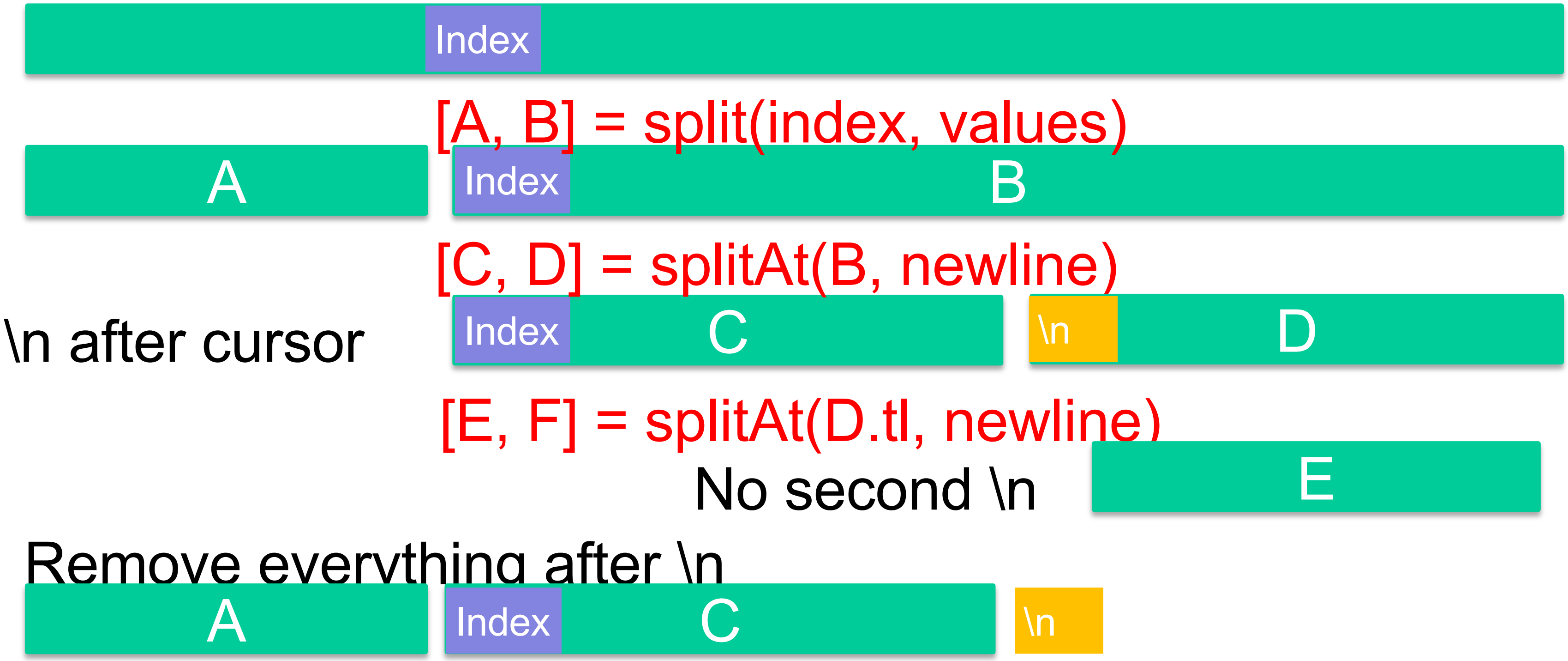
Second \n

E

\n F

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

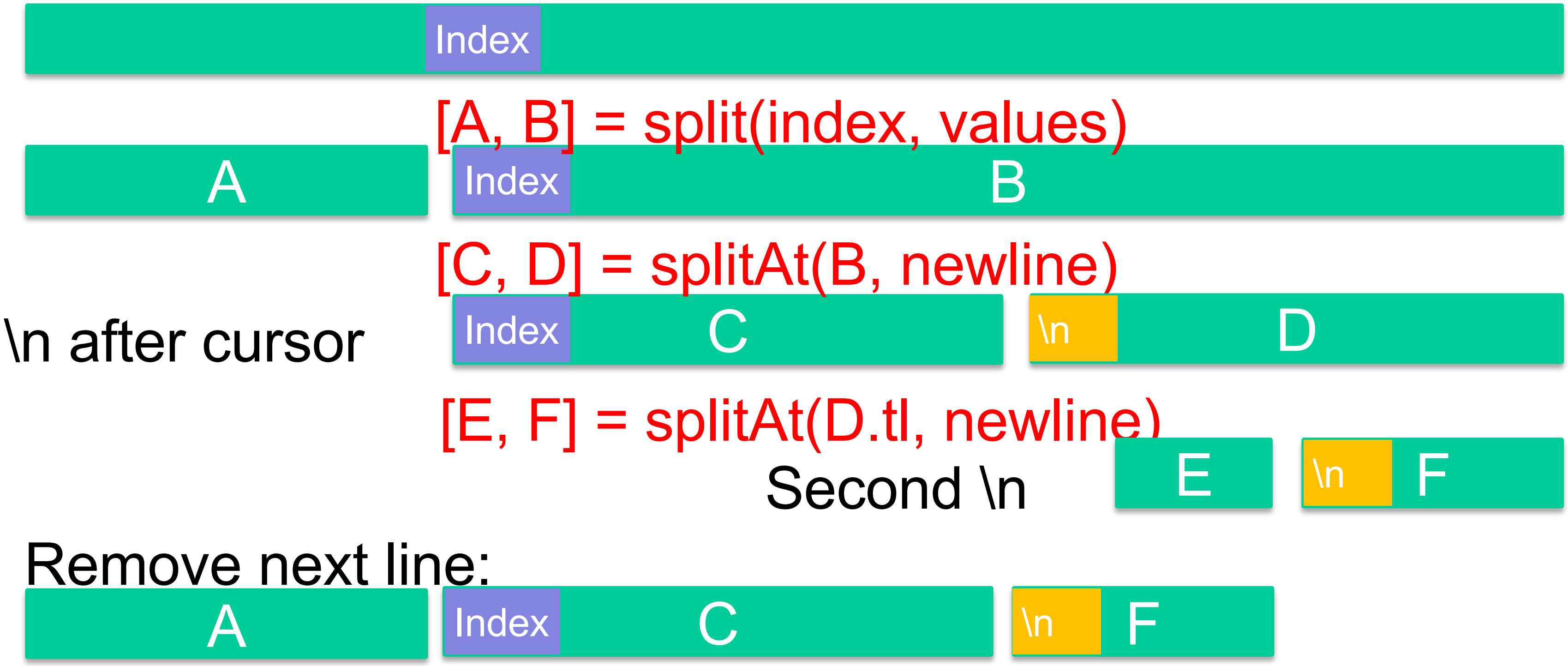
[E, F] = splitAt(D.tl, newline)

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:

Problem 3

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















};
```

You got this!

Puppy Dubs for good luck