CSE 331
Software Design & Implementation

Autumn 2025

Section 6 – ADT Correctness

Administrivia

- HW6 released tonight - **Due @ 6pm Friday**
- **No section on Nov 11th!** - Veterans Day
  - Section video will be posted for HW7 Prep

Review - Defensive Programming

Idea: Try to catch bugs early so that they do not propagate

- reduces debugging

Tools:

- Type checking
- Reasoning (i.e., floyd logic)
- Testing

Examples:

- Add checks for invalid inputs
- Check RI holds at the start and the end of mutators

Review - ADT Correctness

- To prove a method of a *mutable* ADT correct, we need to:
  - show the spec's postcondition holds
  - show the RI holds before and after each mutator
- In other words, these make up the postcondition to prove

- For *immutable* ADTs
  - only need to prove RI is established by the constructor

    because fields don't change, we know RI will remain true

- AF doesn't need to be proven (it always holds when RI holds)

    it is just a given relationship between fields and obj to use in other proofs

    think of it like a fact :)

Review - Mutable ADT Correctness

Correctness checks required for mutable ADTS:

Constructors:

- creates concrete state satisfying the RI
- creates abstract state satisfying the AF
- ensure that there are no aliases

Observers:

- check that the return value is the one required by the spec

Mutators

- abstract state produced is the one required by the spec
- check the RI still holds at end and beginning of the method
- ensure that there are no aliases

Aliases

(a) Class that maintains an array in a specific order

```
public class MyClass {
    // RI: vals is sorted
    private List<String> vals;
    …
    public List<String> values() {
        ~~return this.vals;~~          // unsafe!
        return new ArrayList<>(vals); // make a copy
    }
}
…
```

- Do not hand out access to your own array

Aliases

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

```
public class MyClass {
    // RI: vals is sorted
    private List<String> vals;
    …
    // @requires A is sorted
    public MyClass(List<String> A) {
        ~~this.vals = A;~~          // unsafe!
        this.vals = new ArrayList<>(A); // make a copy
}
…
```

- **Do not make your own fields be something someone else has access to.**

Aliases

- Objects in “Heap State” means that its still being used after the call stack finishes.

- Extra references to these objects are called “aliases”

- When having aliases to mutable heap state:
    - We can gain efficiency in some cases.
    - We must keep track of all aliases that can mutate that state.

- For 331, mutable aliasing across files is a BUG!
    - Allows other portions of your code to break you code
        - **Representation Exposure**
    - “Copy in, copy out” to avoid aliases

```
/**
 * A **mutable** list of integers that can only be modified by adding
 * and removing elements at the front or emptying the entire list.
 */
public interface IntStack {
    /**
     * Returns the length of the list.
     * @returns len(obj)
     */
    int length();

    /**
     * Adds the given number to the front of the list.
     * @param n The number to add to the list.
     * @modifies obj
     * @effects obj = n :: obj_0
     */
    void push(int n);

    /**
     * Removes and returns the first element in the list.
     * @requires len(obj) >= 0
     * @modifies obj
     * @effects obj_0 = n :: obj
     * @returns n
     */
    int pop();

    /**
     * Removes all elements in the list.
     * @modifies obj
     * @effects obj = nil
     */
    void clear();
}
```

$$\textbf{type } \mathsf{PList} := \mathsf{pnil} \mid \mathsf{pcons}((\mathbb{Z}, \mathbb{N}), \mathsf{PList})$$

We will use the shorthand "::", when applied to a PList, to refer to a pcons operation. For example, the expression "$(1, 2)$ :: $(3, 4)$ :: pnil" is shorthand for pcons($(1, 2)$, pcons($(3, 4)$, pnil)). Hopefully, this will not cause confusion with cons on List.

With that definition in hand, our concrete representation will store the list 1 :: 1 :: 1 :: 2 :: 3 :: 3 :: nil, which contains runs of 1s and 3s, as the shorter list $(1, 3)$ :: $(2, 1)$ :: $(3, 2)$ :: pnil. The following class uses this concrete representation:

```
public class CompressedIntStack implements IntStack {
    // AF: obj = expand(this.pairs)
    private PairList pairs;
```

This uses the `PairList` class, defined below, to store a PList, and the function expand : (PList) → List, which is defined as follows:

$$\begin{aligned}
\mathsf{expand}(\mathsf{pnil}) &:= \mathsf{nil} \\
\mathsf{expand}((n, 0) :: L) &:= \mathsf{expand}(L) \\
\mathsf{expand}((n, c+1) :: L) &:= n :: \mathsf{expand}((n, c) :: L)
\end{aligned}$$

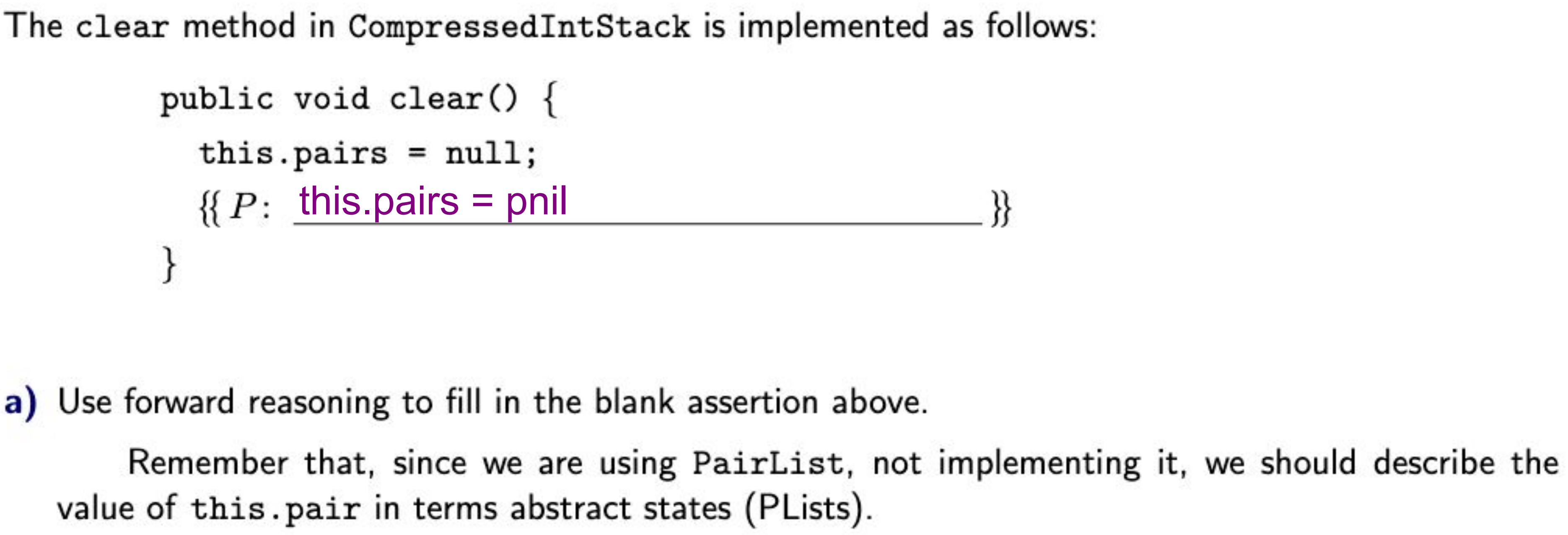
The abstraction function of `CompressedIntStack` says that the abstract state is the list that you would get by expanding the PList stored in the field `pairs`.

Finally, the `PairList`, which stores a PList directly as a linked list, is defined as follows:

```
/** Represents a list of pairs. */
private static class PairList {
    public final int value;
    public final int count;
    public final PairList next;

    public PairList(int value, int count, PairList next) {
        this.value = value;
        this.count = count;
        this.next = next;
    }
}
```

Task 1 – Hold My Clear



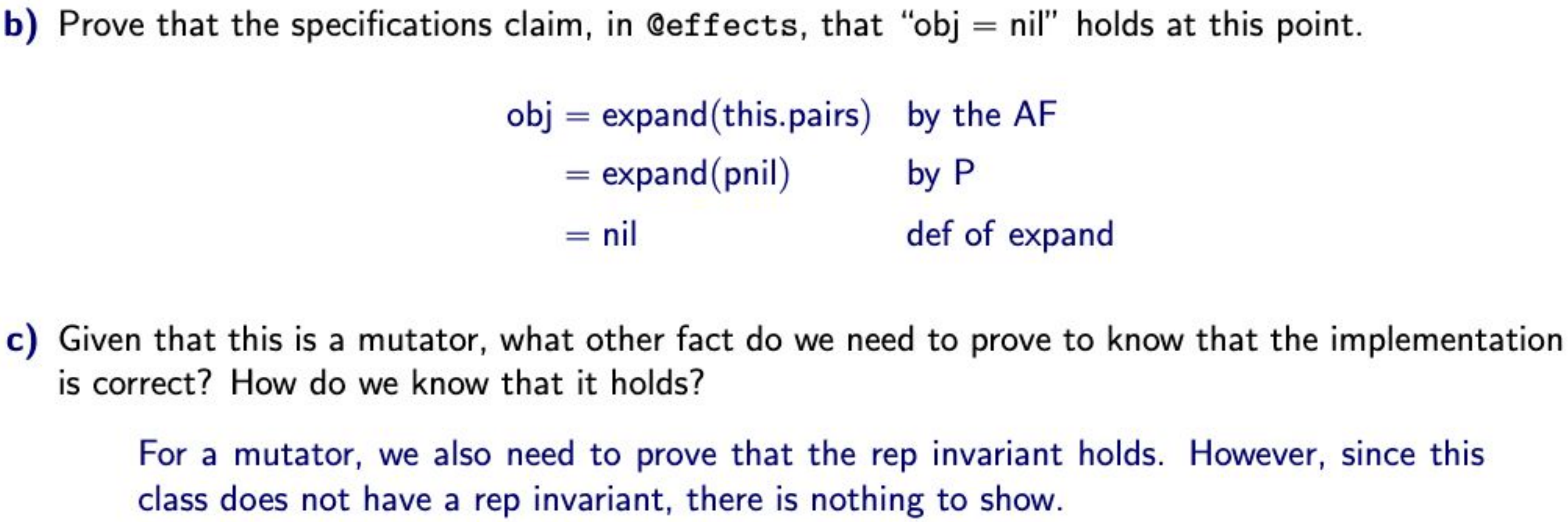
The clear method in CompressedIntStack is implemented as follows:

```
public void clear() {
  this.pairs = null;
  {{ P: this.pairs = pnil }}
}
```

a) Use forward reasoning to fill in the blank assertion above.

Remember that, since we are using PairList, not implementing it, we should describe the value of this.pair in terms abstract states (PLists).

Task 1 – Hold My Clear



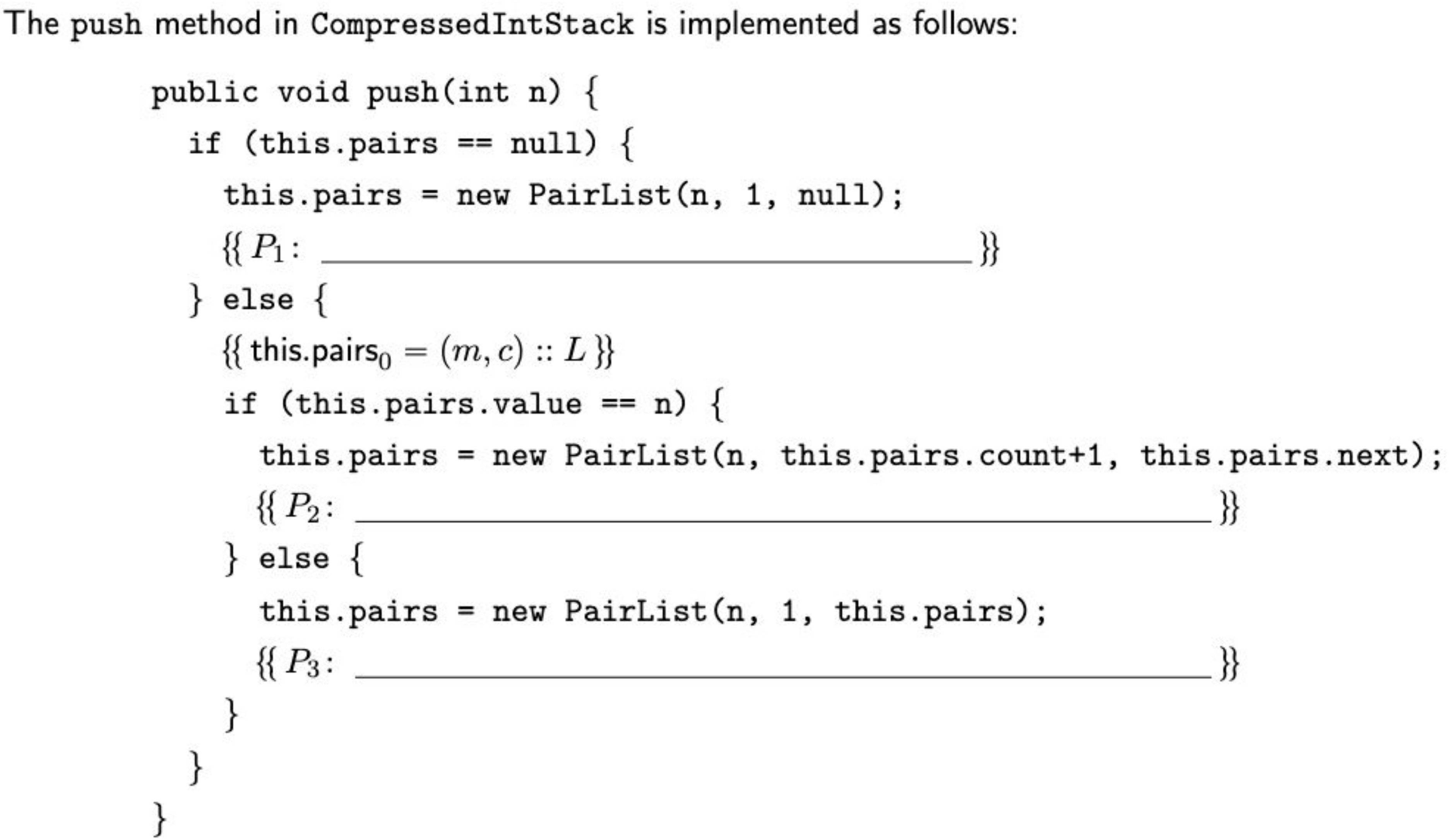
b) Prove that the specifications claim, in `@effects`, that "obj = nil" holds at this point.

$$\begin{aligned} \text{obj} &= \text{expand}(\text{this.pairs}) && \text{by the AF} \\ &= \text{expand}(\text{pnil}) && \text{by P} \\ &= \text{nil} && \text{def of expand} \end{aligned}$$

c) Given that this is a mutator, what other fact do we need to prove to know that the implementation is correct? How do we know that it holds?

For a mutator, we also need to prove that the rep invariant holds. However, since this class does not have a rep invariant, there is nothing to show.

Task 2 – Loops-a-Daisy



The push method in CompressedIntStack is implemented as follows:

```
public void push(int n) {
  if (this.pairs == null) {
    this.pairs = new PairList(n, 1, null);
    {{ $P_1$: ________________________________ }}
  } else {
    {{ $\text{this.pairs}_0 = (m, c) :: L$ }}
    if (this.pairs.value == n) {
      this.pairs = new PairList(n, this.pairs.count+1, this.pairs.next);
      {{ $P_2$: ____________________________________________ }}
    } else {
      this.pairs = new PairList(n, 1, this.pairs);
      {{ $P_3$: ____________________________________________ }}
    }
  }
}
```

Task 2 – Loops-a-Daisy

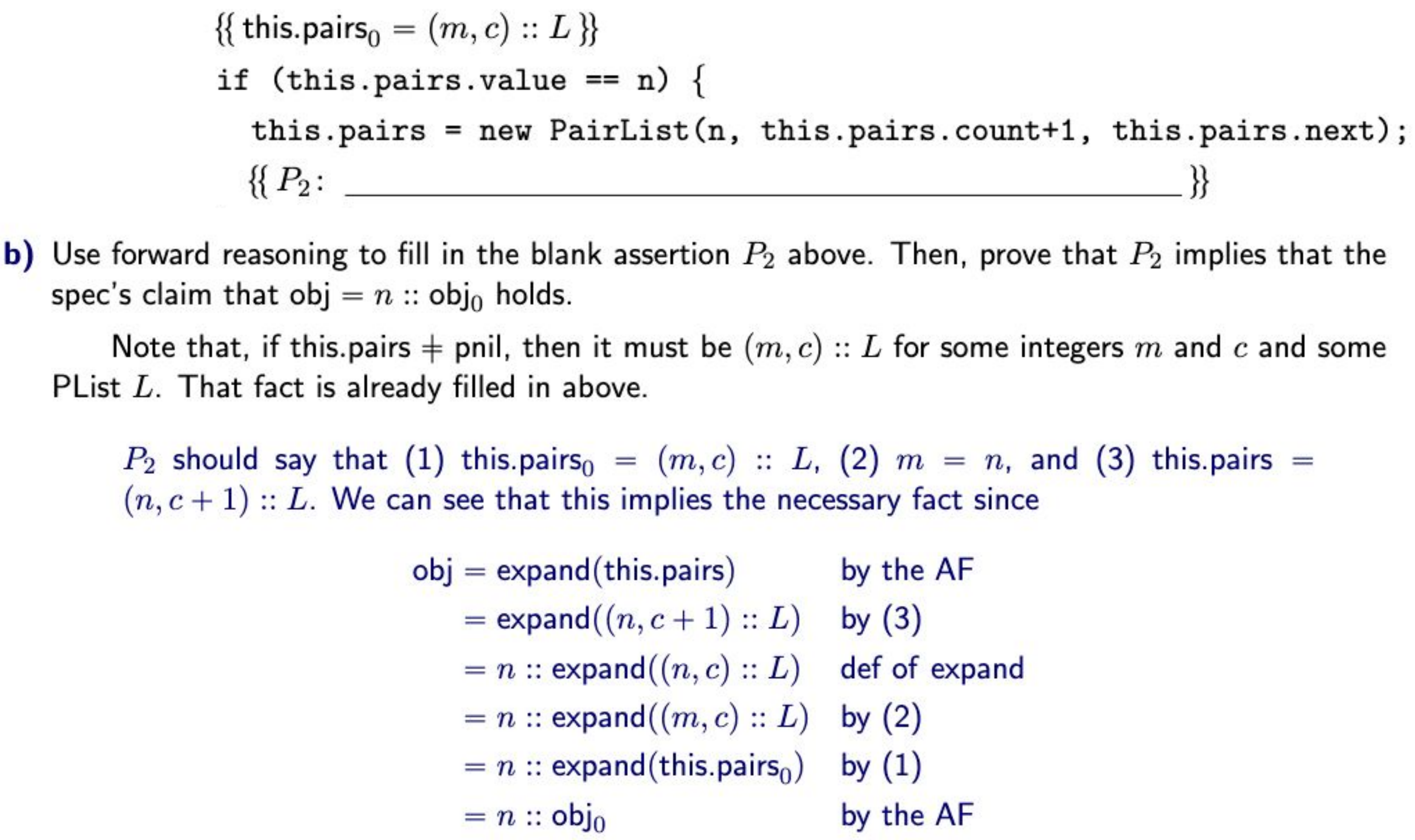
```
public void push(int n) {
  if (this.pairs == null) {
    this.pairs = new PairList(n, 1, null);
    {{ P1: ________________________________ }}
```

a) Use forward reasoning to fill in the blank assertion P_1 above. Then, prove that P_1 implies that the spec's claim that obj $= n ::$ obj$_0$ holds.

P_1 should say (1) this.pairs$_0$ = pnil and (2) this.pairs $= (n, 1) ::$ pnil. We can see that this implies the necessary fact since

$$
\begin{aligned}
\text{obj} &= \text{expand}(\text{this.pairs}) && \text{by the AF} \\
&= \text{expand}((n,1) :: \text{pnil}) && \text{by (2)} \\
&= \text{expand}((n,1) :: \text{this.pairs}_0) && \text{by (1)} \\
&= n :: \text{expand}((n,0) :: \text{this.pairs}_0) && \text{def of expand} \\
&= n :: \text{expand}(\text{this.pairs}_0) && \text{def of expand} \\
&= n :: \text{obj}_0 && \text{by the AF}
\end{aligned}
$$

Task 2 – Loops-a-Daisy



$\{\{\ \text{this.pairs}_0 = (m, c) :: L\ \}\}$

```
if (this.pairs.value == n) {
  this.pairs = new PairList(n, this.pairs.count+1, this.pairs.next);
```

$\{\{\ P_2 : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\ \}\}$

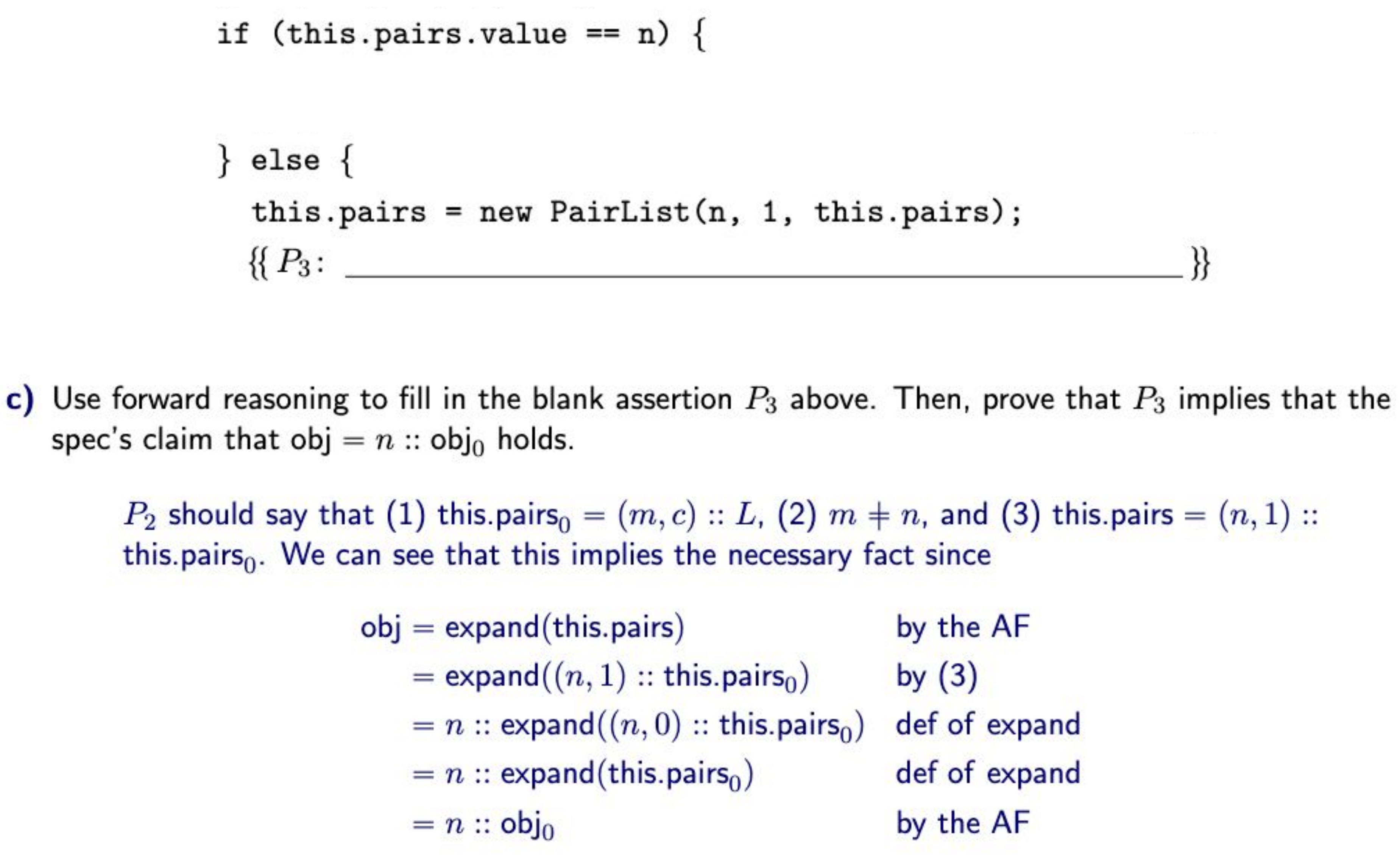
b) Use forward reasoning to fill in the blank assertion P_2 above. Then, prove that P_2 implies that the spec's claim that obj $= n ::$ obj$_0$ holds.

Note that, if this.pairs $\neq$ pnil, then it must be $(m, c) :: L$ for some integers m and c and some PList L. That fact is already filled in above.

P_2 should say that (1) this.pairs$_0$ $= (m, c) :: L$, (2) $m = n$, and (3) this.pairs $= (n, c+1) :: L$. We can see that this implies the necessary fact since

$$
\begin{aligned}
\text{obj} &= \text{expand(this.pairs)} && \text{by the AF} \\
&= \text{expand}((n, c+1) :: L) && \text{by (3)} \\
&= n :: \text{expand}((n, c) :: L) && \text{def of expand} \\
&= n :: \text{expand}((m, c) :: L) && \text{by (2)} \\
&= n :: \text{expand}(\text{this.pairs}_0) && \text{by (1)} \\
&= n :: \text{obj}_0 && \text{by the AF}
\end{aligned}
$$

Task 2 – Loops-a-Daisy



```
if (this.pairs.value == n) {


} else {
  this.pairs = new PairList(n, 1, this.pairs);
  {{ P3: ________________________________________ }}
```

c) Use forward reasoning to fill in the blank assertion P_3 above. Then, prove that P_3 implies that the spec's claim that obj $= n ::$ obj$_0$ holds.

P_2 should say that (1) this.pairs$_0 = (m, c) :: L$, (2) $m \neq n$, and (3) this.pairs $= (n, 1) ::$ this.pairs$_0$. We can see that this implies the necessary fact since

$$\begin{aligned}
\text{obj} &= \text{expand}(\text{this.pairs}) && \text{by the AF} \\
&= \text{expand}((n,1) :: \text{this.pairs}_0) && \text{by (3)} \\
&= n :: \text{expand}((n,0) :: \text{this.pairs}_0) && \text{def of expand} \\
&= n :: \text{expand}(\text{this.pairs}_0) && \text{def of expand} \\
&= n :: \text{obj}_0 && \text{by the AF}
\end{aligned}$$