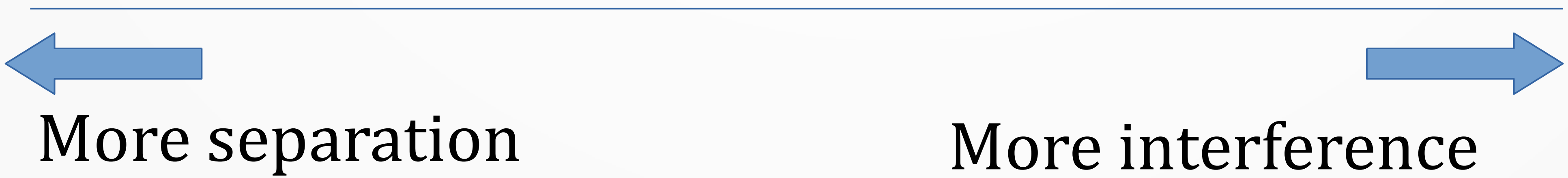
# Logic for Verification 2

Nisansala Yatapanage
ANU Logic Summer School

# Concurrency: Separation vs. Interference

- Understanding which logics are best for which problems is essential.
- Concurrency can be viewed as a spectrum with full separation on one end and full interference on the other.



# Abstraction and Refinement

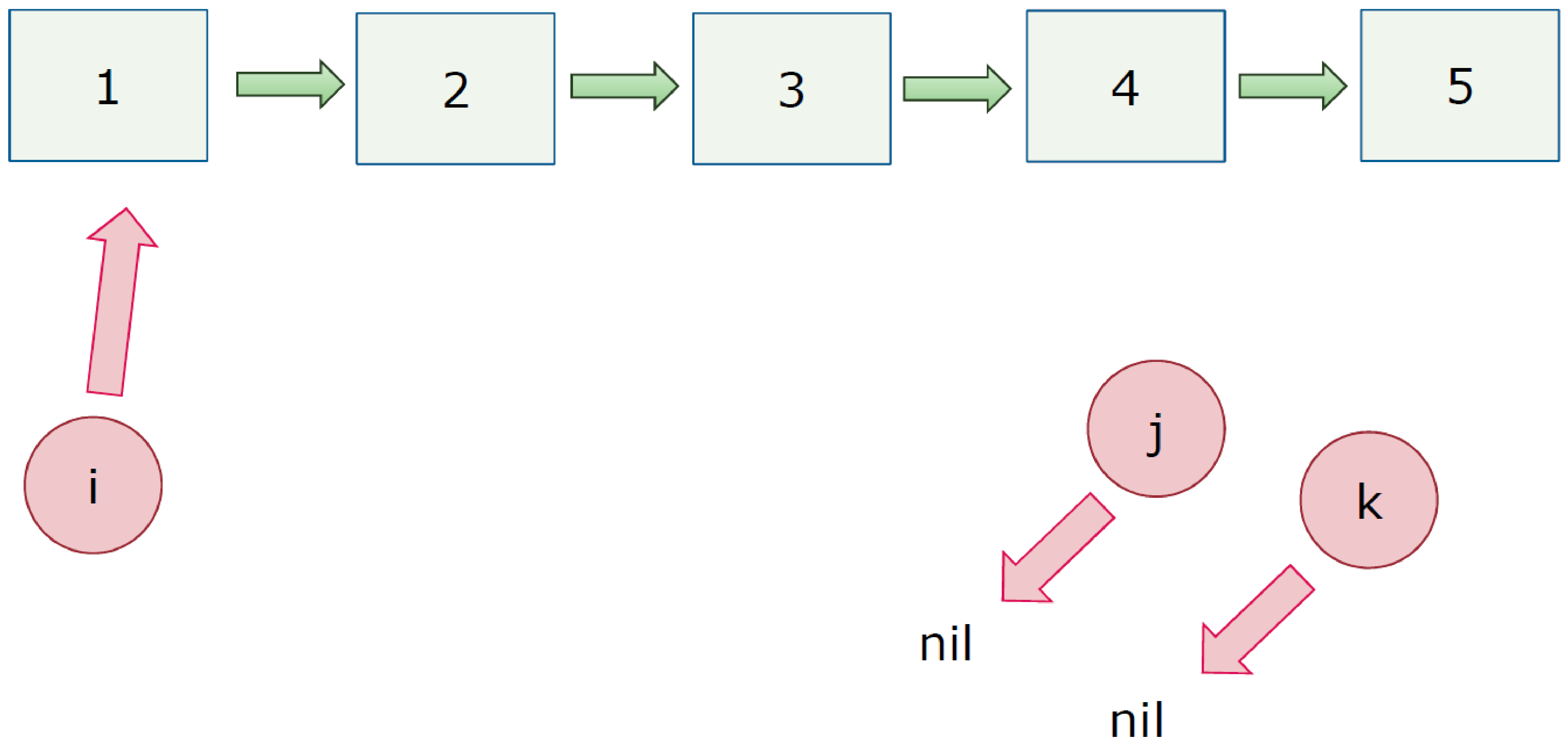
- We can work at the level of program code, but that's not the best way to understand a program.
- Look at the next slide's program. How easy is it to understand compared to the diagram?

# Separation – Reynold's list reversal

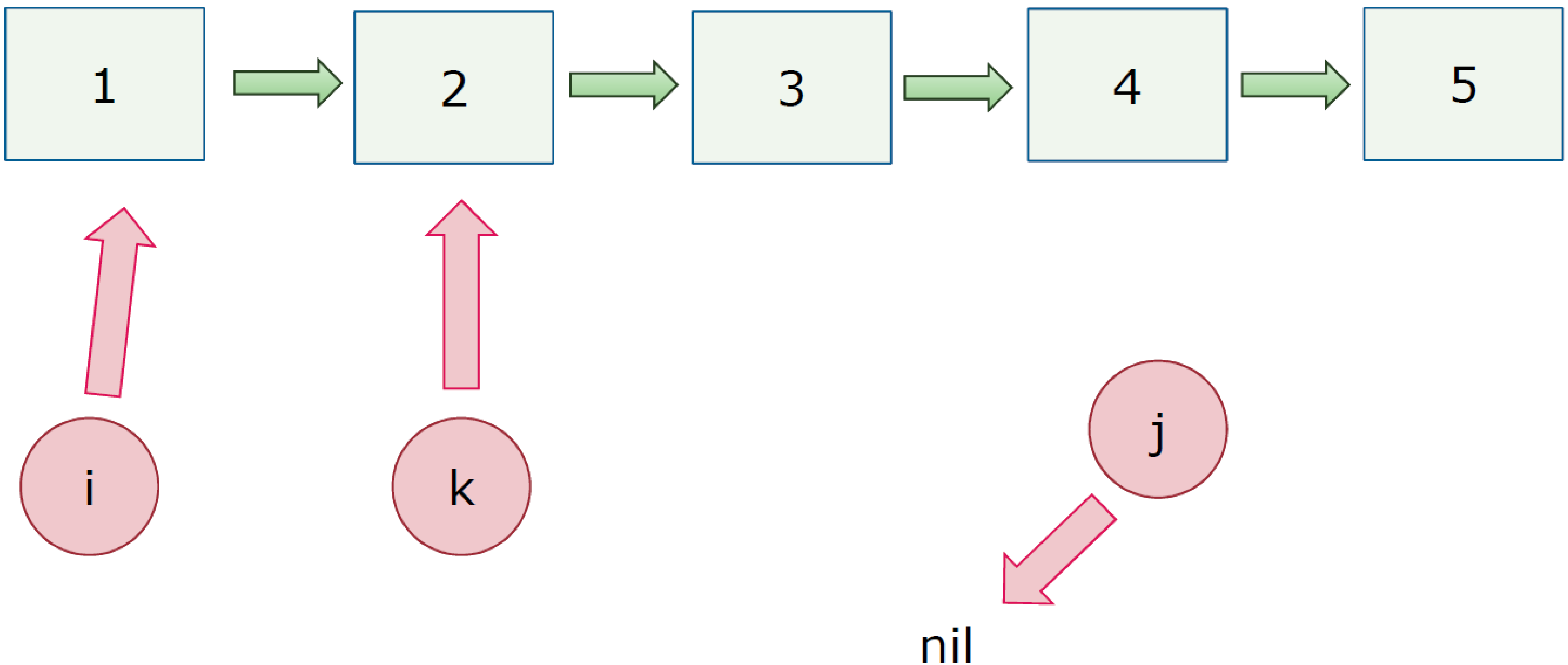
```
j = null;
while (i != null) {
  k = *(i+1);
  *(i+1) = j;
  j = i;
  i = k;
}
```

$$post\text{-}REVERSE_0((s, r), (s', r')) \triangleq r' = rev(s)$$

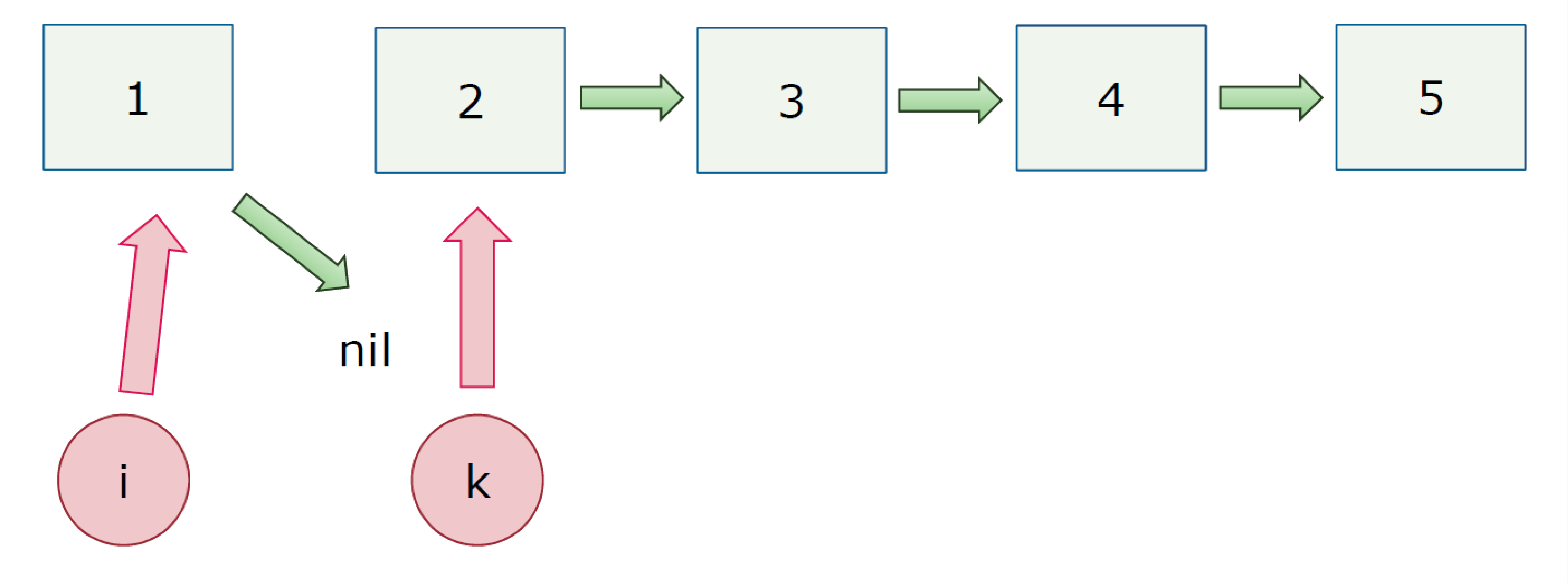
# Separation – Reynold's list reversal



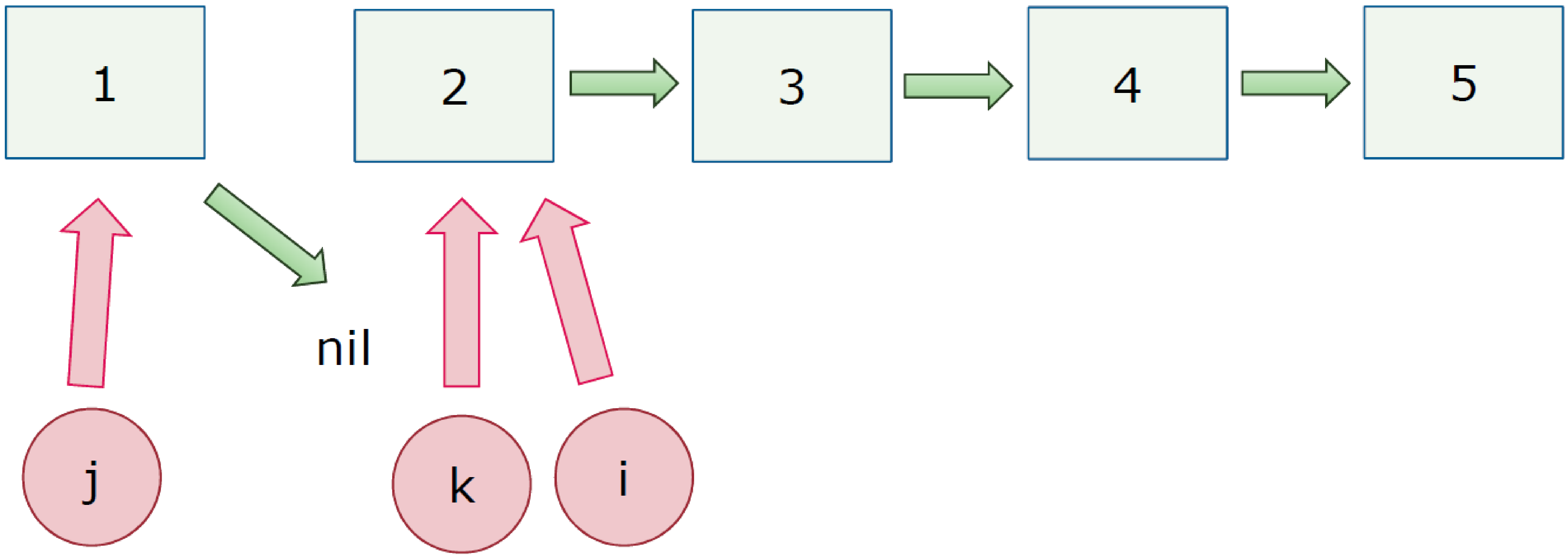
# Separation - Reynold's list reversal



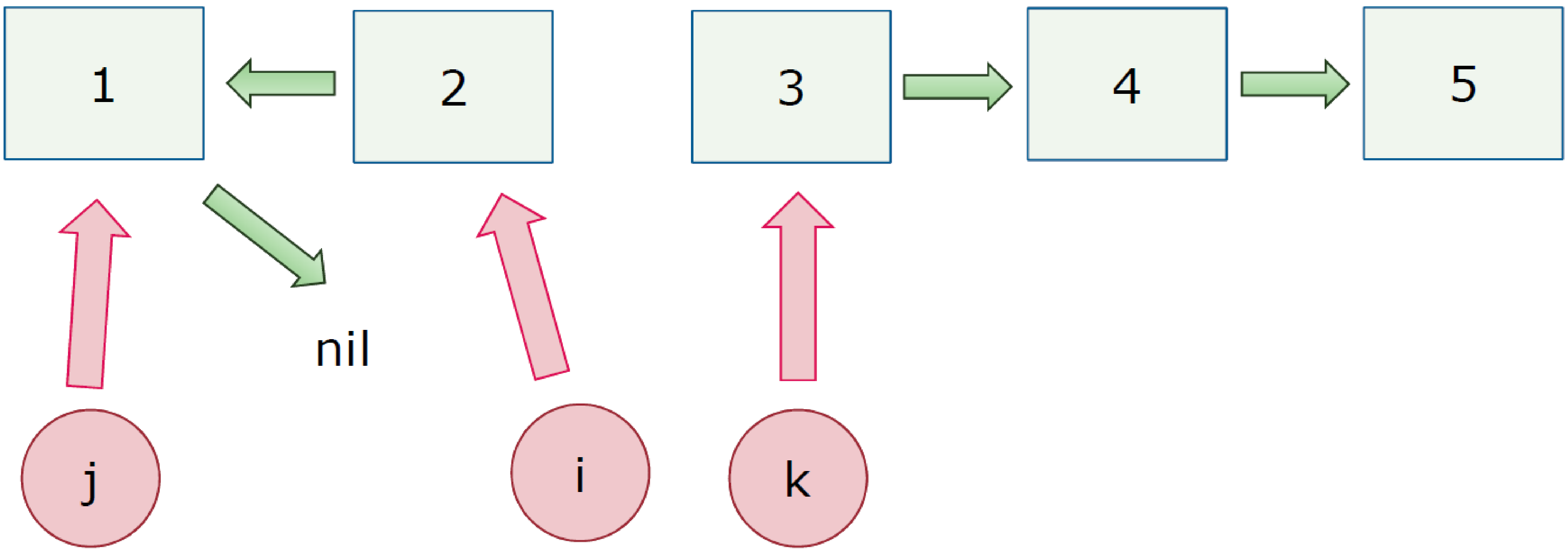
# Separation - Reynold's list reversal



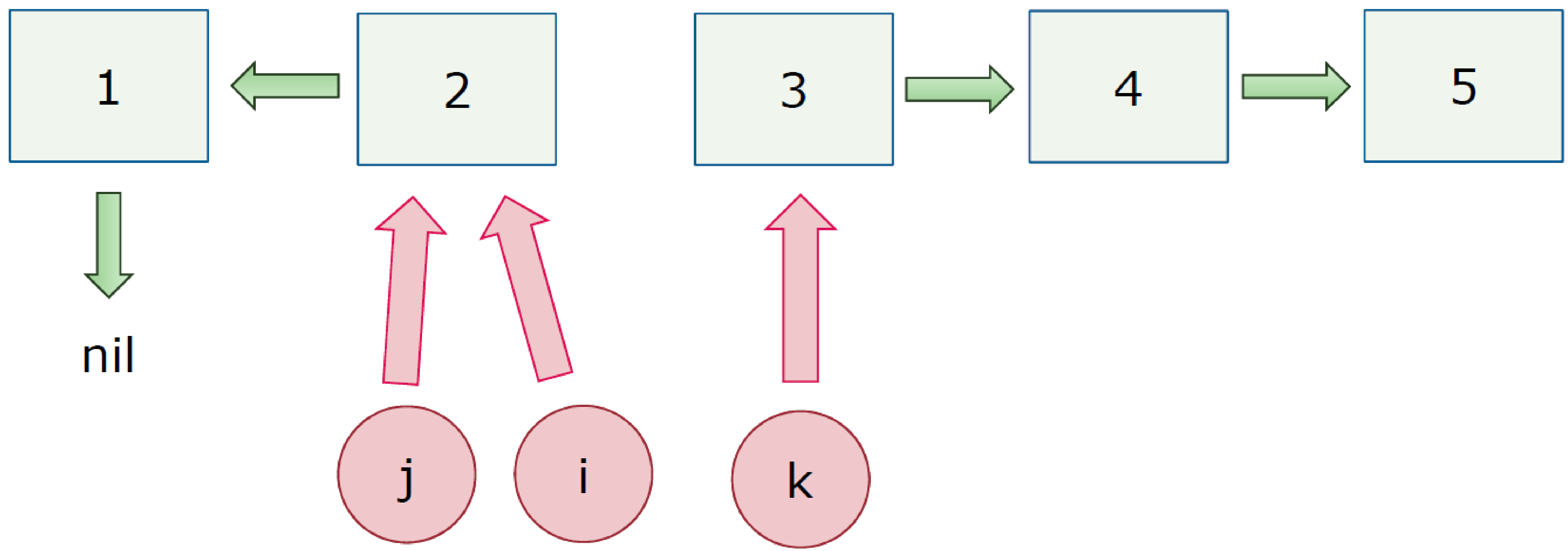
# Separation - Reynold's list reversal



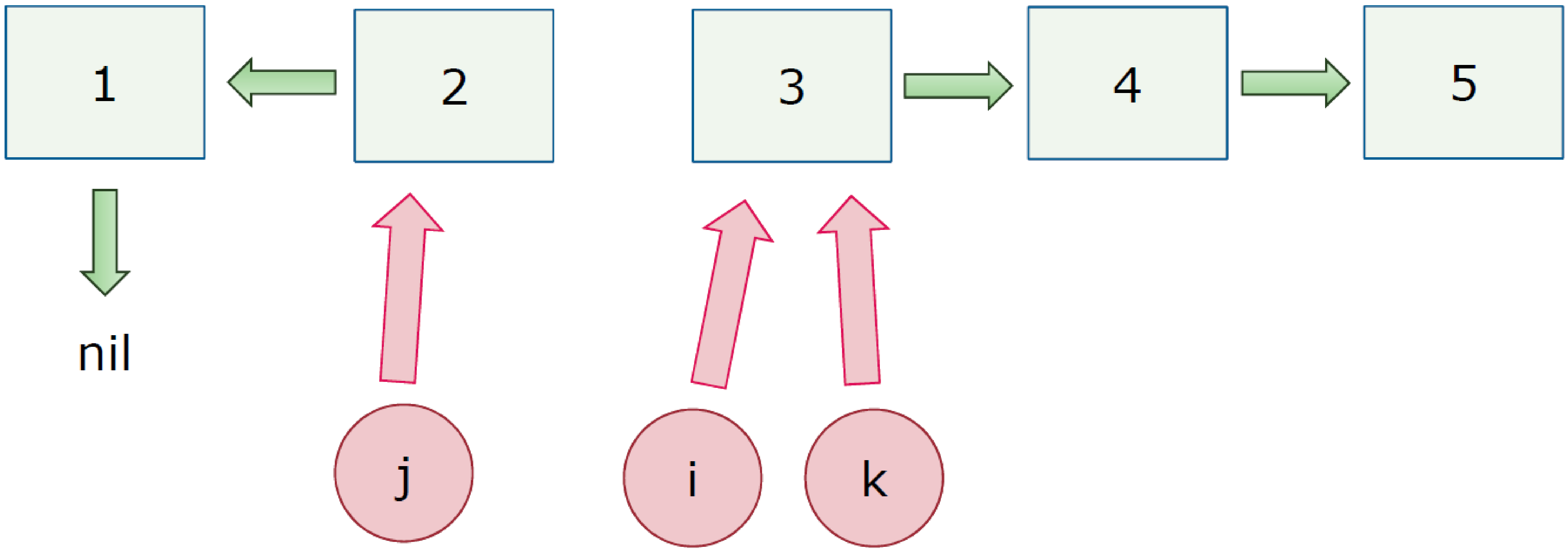
# Separation - Reynold's list reversal



# Separation - Reynold's list reversal



# Separation - Reynold’s list reversal



# Abstraction and Refinement

- Abstraction makes the core concepts clearer.
- Several concrete specifications can be refinements of the same abstract specification.
- For example, a stack could be implemented in many ways, but must maintain the same abstract behaviour:

  push a value

  pop an item from the stack

# Abstract level

$$
\begin{array}{l}
r \leftarrow [\,];\\
\textbf{while}\ \ s \neq [\,]\ \ \textbf{do}\\
\quad STEP_0\\
\textbf{end}\ \ \textbf{while}
\end{array}
$$

$$pre\text{-}STEP_0((r, s)) \triangleq s \neq [\,]$$

$$post\text{-}STEP_0((r, s), (r', s')) \triangleq r' = [\mathbf{hd}\ s] \curvearrowright r \wedge s' = \mathbf{tl}\ s$$

- On this level, *r* and *s* are assumed to be separate.

# More concrete level

- Srep (a sub-heap)

$$\Sigma_1 = (Srep \times Srep)$$

**where**

$$inv\text{-}\Sigma_1((sr, rr)) \quad \triangleq \quad sep(sr, rr)$$

$$sep : Srep \times Srep \to \mathbb{B}$$

$$sep(sr, rr) \quad \triangleq \quad \mathbf{dom}\ sr \cap \mathbf{dom}\ rr = \{\,\}$$

# Relating concrete and abstract levels

$$retr_0 : \Sigma_1 \to \Sigma_0$$

$$retr_0((sr, rr)) \quad \triangleq \quad (gather(sr), gather(rr))$$

$$retr_1 : \Sigma_2 \to \Sigma_1$$

$$retr_1((hp, i, j)) \quad \triangleq \quad (trace(hp, i) \lhd hp, trace(hp, j) \lhd hp)$$

$$\forall (sr, rr) \in \Sigma_1 \cdot \exists (hp, i, j) \in \Sigma_2 \cdot retr_1((hp, i, j)) = (sr, rr)$$

A minimal heap is constructed that contains the pointers in *sr* and *rr*, which are disjoint.

# Separation - mergesort

$$mergesort : Val^* \to Val^*$$

$$\begin{array}{l} mergesort(s) \quad \triangleq \\ \qquad \textbf{if len}\ s \leq 1 \\ \qquad \textbf{then}\ \ s \\ \qquad \textbf{else let}\ \ s1, s2\ \textbf{be st}\ \ s1 \curvearrowright s2 = s \wedge s1 \neq [\,] \wedge s2 \neq [\,]\ \ \textbf{in} \\ \qquad\qquad merge(mergesort(s1), mergesort(s2)) \end{array}$$

# Separation - mergesort

- Again, this was modelled using three levels:
- Abstract level
- Level with Sreps
- Level with the heap and pointers.

# Separation - mergesort

- Since mergesort is concurrent, we need to have rely and guar conditions to specify the interference.

- Only this process changes the sequence starting with *p*.

$$rely\text{-}MSORT_2 \colon p' = p \wedge trace(hp, p) \lhd hp' = trace(hp, p) \lhd hp$$
$$guar\text{-}MSORT_2 \colon trace(hp, p) \ntriangleleft hp' = trace(hp, p) \ntriangleleft hp$$

# Concurrency: Separation vs. Interference

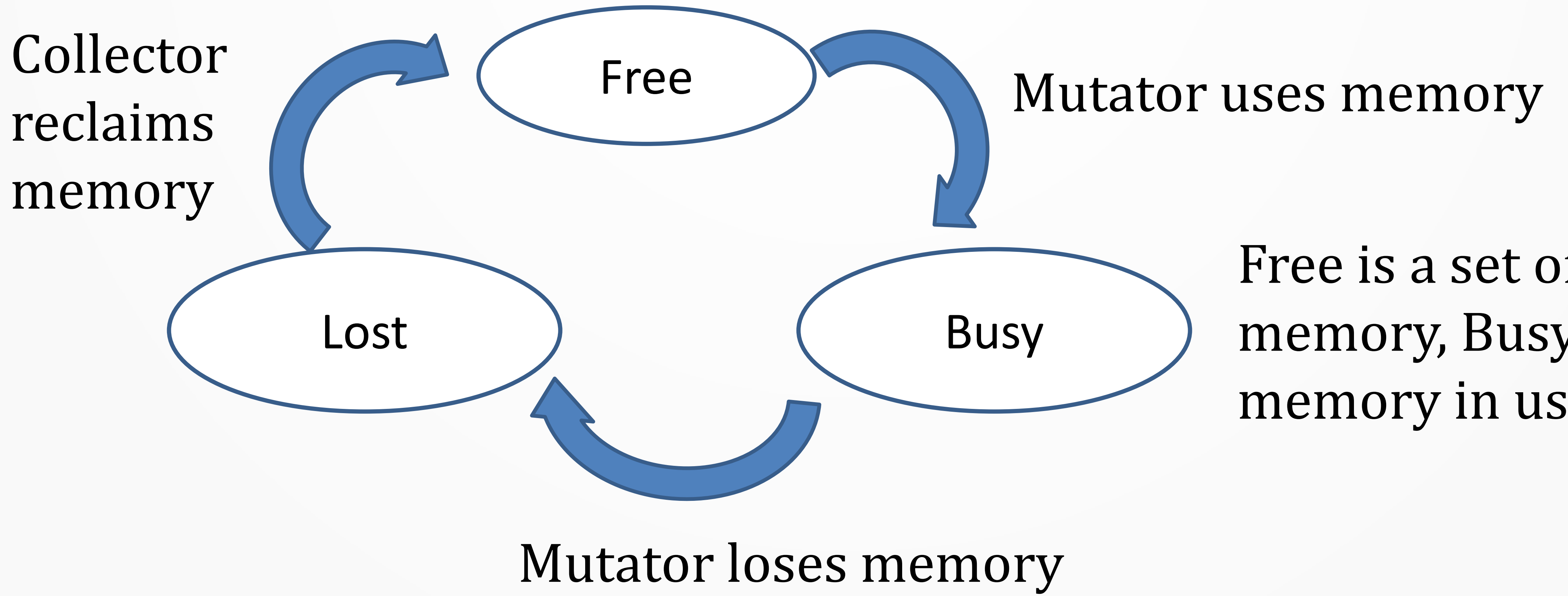
- The interference and separation carried through from the abstract to the concrete levels.
- It can help to start with a sequential version and then introduce the concurrency.
- Understanding the core issues are essential for understanding the different problems, the different approaches and which work well with each.

# Concurrency: Separation vs. Interference

- Viewing separation in terms of abstraction – helps to understand it.
- Looking at problems at the boundaries reveals the core issues:
  - Non-blocking algorithms that lie on the border of what **rely/guarantee** can handle.
  - Ben-Ari's garbage collection algorithm – revealed that standard rely guarantee cannot be applied without some additions.

# Concurrency: Separation vs. Interference

Looking at problems at the boundaries reveals the core issues.




Free is a set of free memory, Busy is memory in use.

# Abstract Specification of the Collector

$Collector$

**ext wr** $free$
**rd** $busy$

**pre true**

**rely** $free' \subseteq free \land (busy' - busy) \subseteq free$

**guar** $free \subseteq free'$

**post** $(Addr - busy) \subseteq \bigcup \overleftarrow{free}$

*rely-Collector* ensures that any addresses the Mutator adds to **busy** are taken from **free**.

*guar-Collector* ensures that **free** addresses will be preserved, but more can be added to the set.

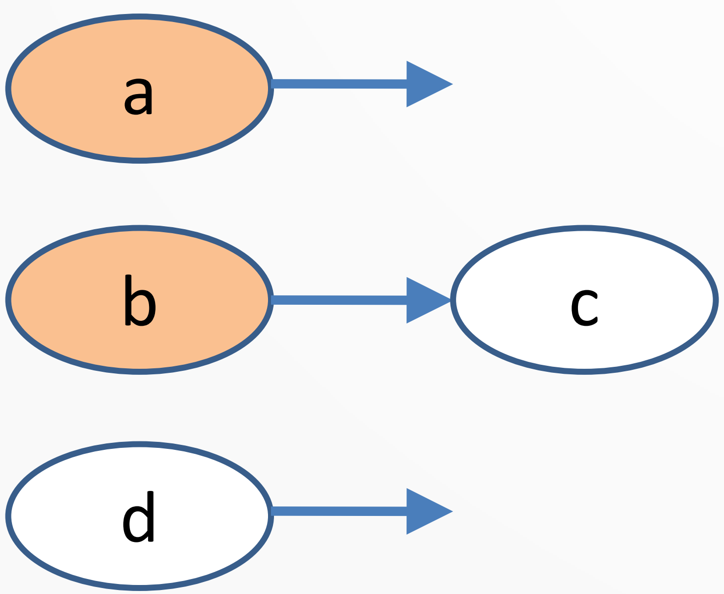
If it was sequential, *post-Collector* could be written as: free' = Addr – busy

However, remember the Mutator could take things out of free.

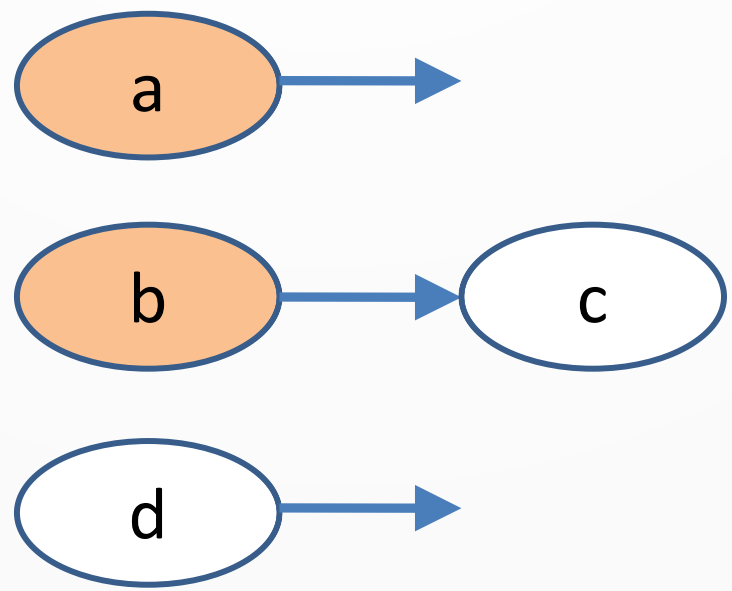
# Concurrency: Separation vs. Interference

The Collector is in the Marking phase. Meanwhile, the Mutator changes some links…

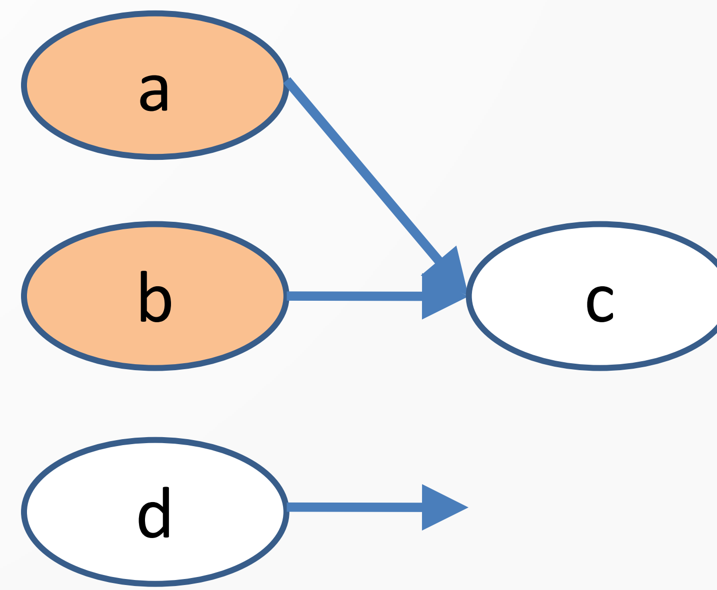
Original state:




Collector marks a's children.

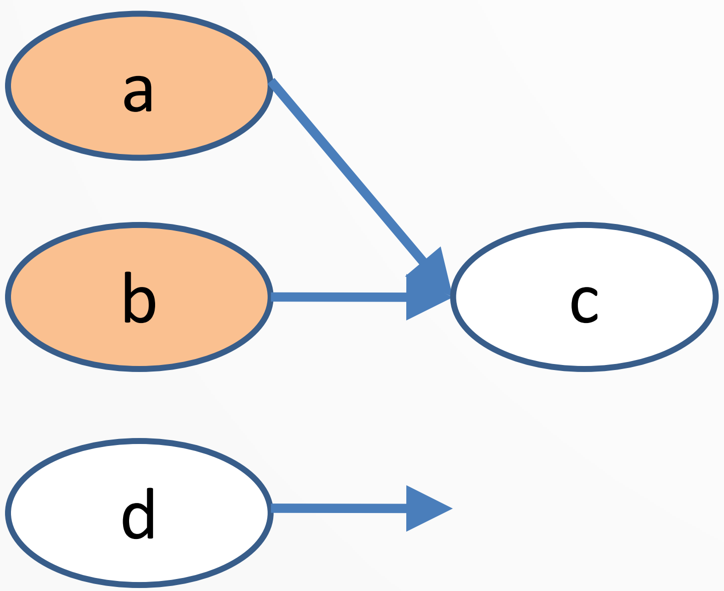



Mutator changes link.

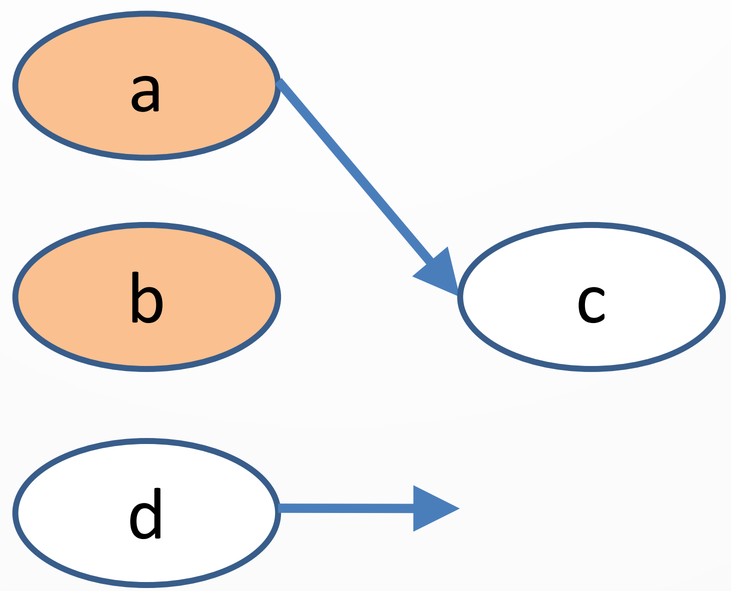


# Concurrency: Separation vs. Interference

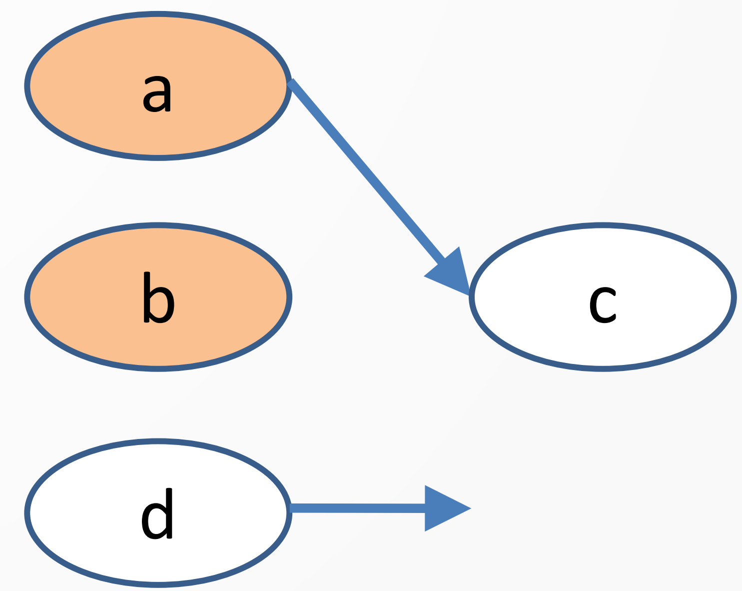
Mutator changes link.




Mutator removes link.

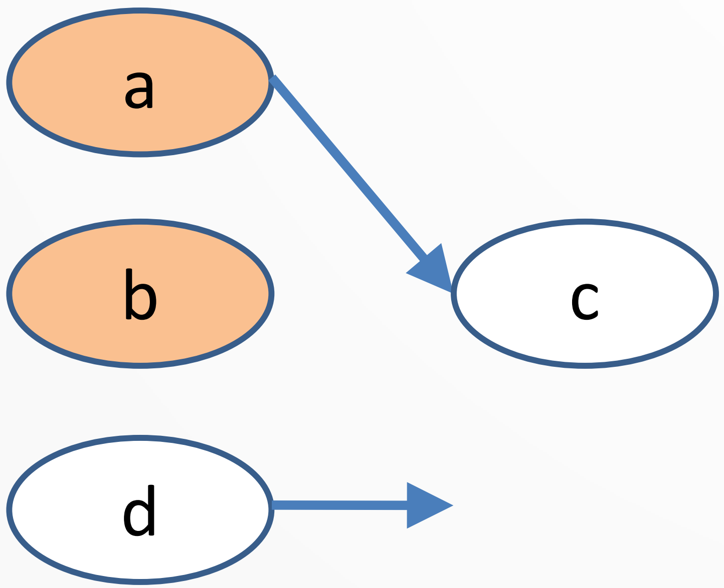



Collector marks b's children.

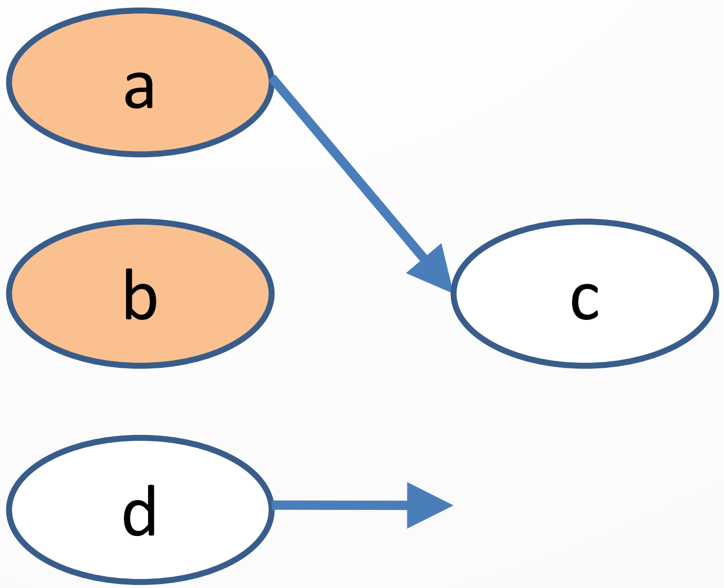


# Concurrency: Separation vs. Interference

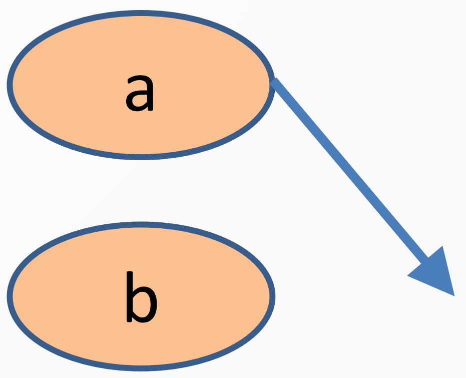
Collector marks b's children.




Collector finishes and goes into Sweep phase.



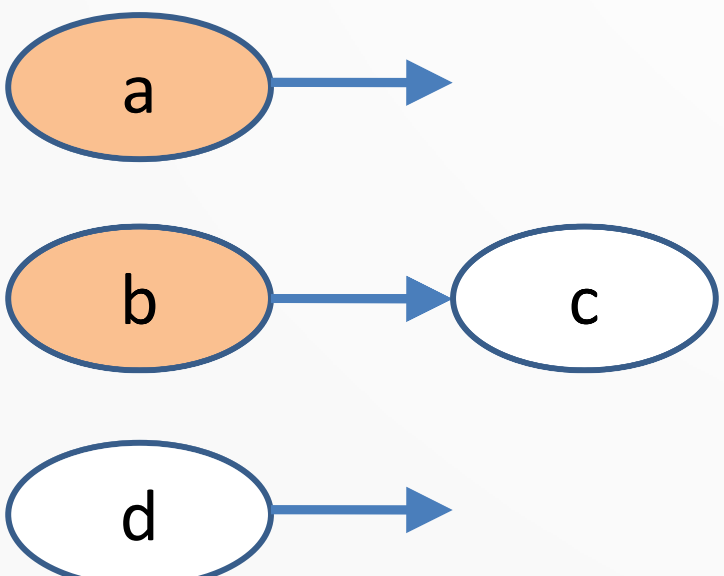

Node c is deleted!



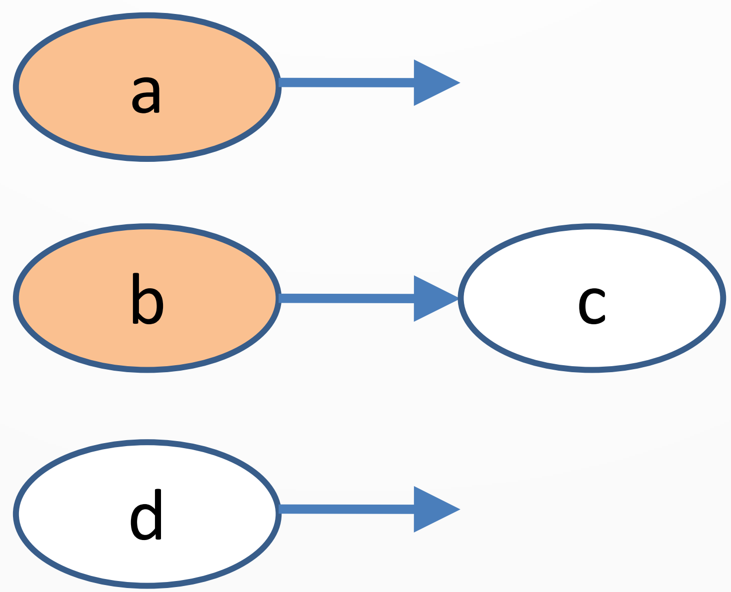
# Concurrency: Separation vs. Interference

It can't happen if the Mutator marks in between changes:

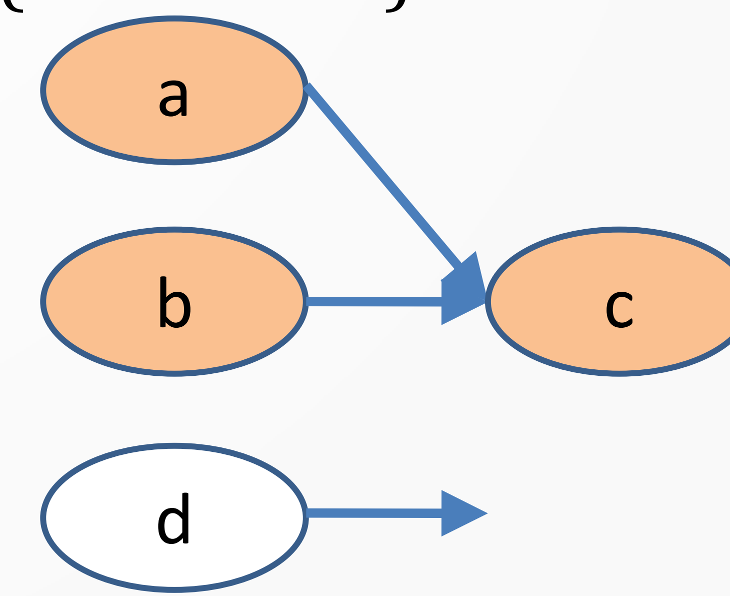
Original state:




Collector marks a's children.

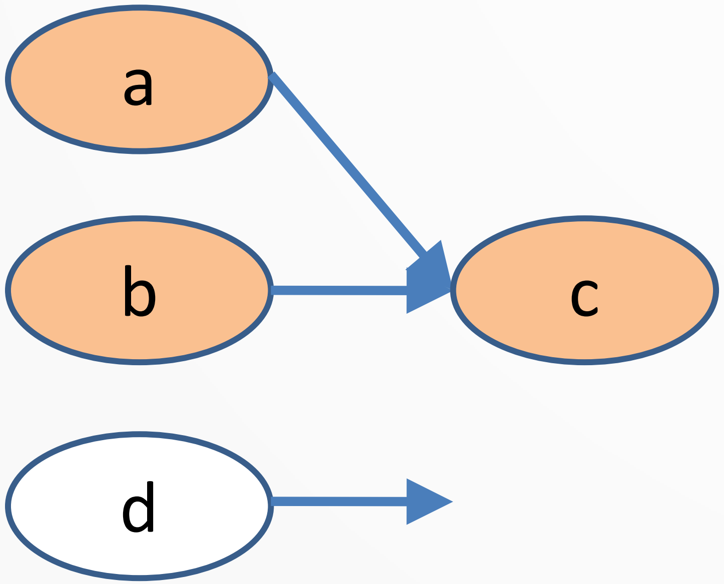



Mutator changes link (and marks).

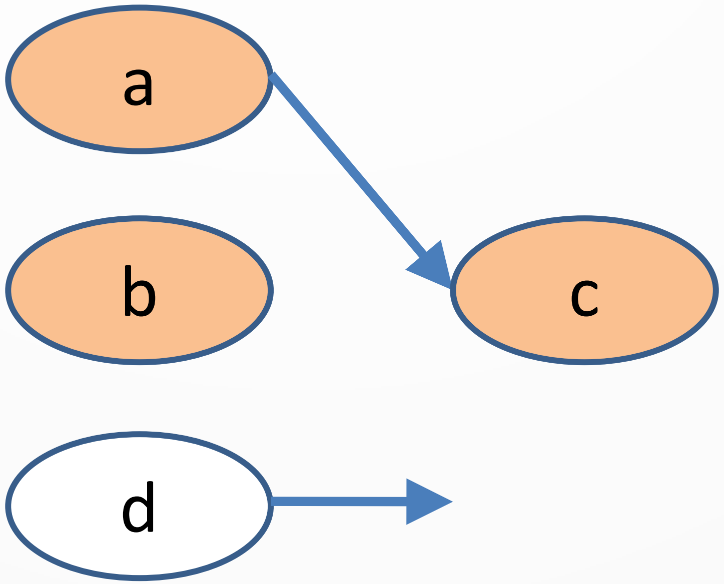


# Concurrency: Separation vs. Interference

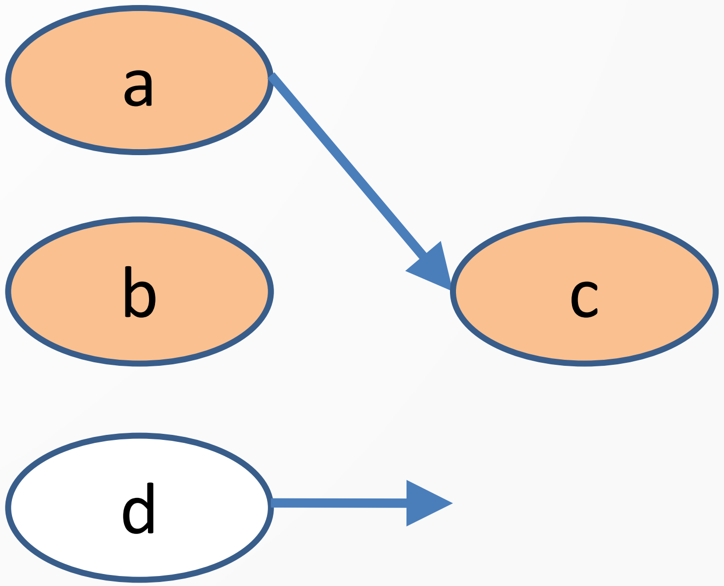
Mutator changes link (and marks).




Mutator removes link.

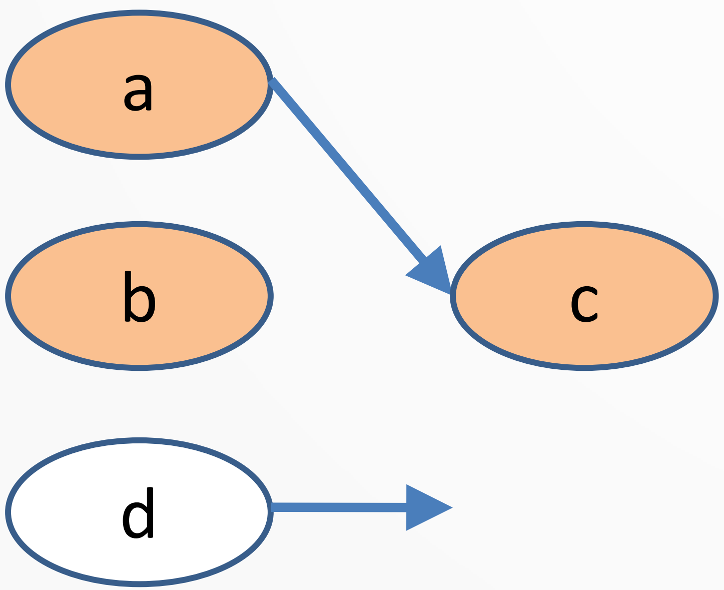



Collector marks b's children.

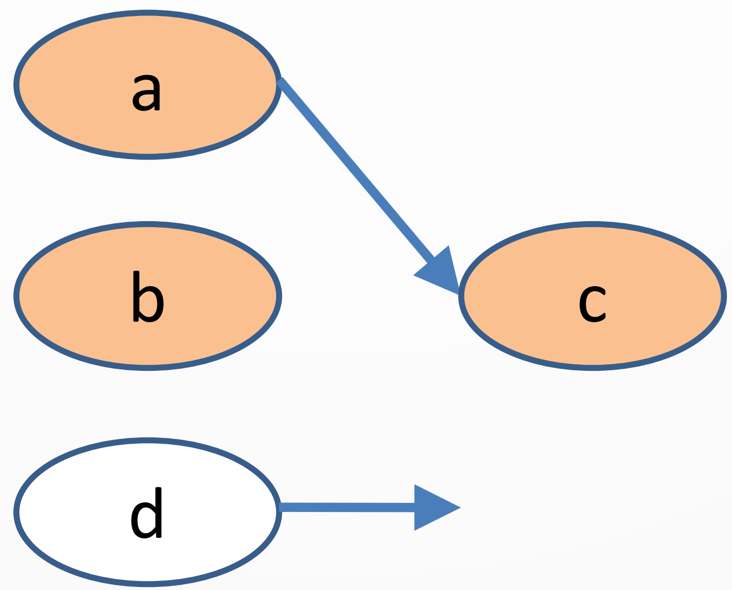


# Concurrency: Separation vs. Interference

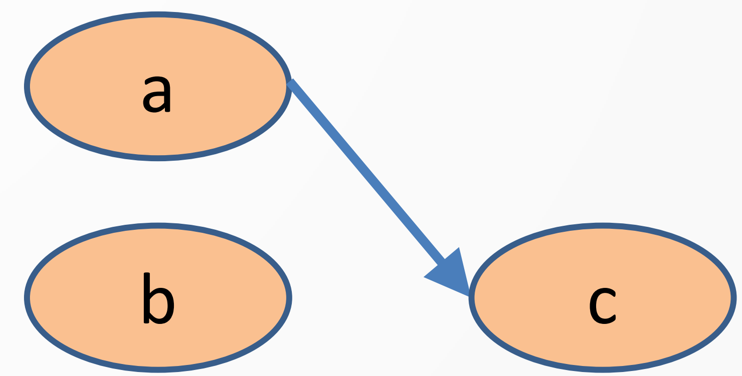
Collector marks b's children.




Collector finishes but the marked nodes has increased so does another run.




Sweep phase: only d is removed.



# Concurrent Collector

$$
\begin{array}{l}
Collector_a \\
\textbf{ext wr } free, marked \\
\quad\quad \textbf{rd } roots, hp \\
\textbf{pre true} \\
\textbf{rely } free' \subseteq free \wedge \\
\quad\quad (reach(roots, hp') - reach(roots, hp)) \subseteq free \wedge \\
\quad\quad marked \subset marked' \wedge \\
\quad\quad \forall (a, i) \in \textbf{dom } hp \cdot \\
\quad\quad\quad\quad hp'(a, i) \neq hp(a, i) \wedge hp'(a, i) \in Addr \;\; \Rightarrow \;\; hp'(a, i) \in marked' \\
\textbf{guar } free \subseteq free' \\
\textbf{post } (Addr - reach(roots, hp)) \subseteq \bigcup \overleftarrow{free}
\end{array}
$$

# Shared Ghost Variables

Introduce a shared ghost variable *tbm*.

The Mutator sets *tbm* atomically when it changes a link.

$$< \textbf{if } hp(a,i) \neq b \textbf{ then } hp(a,i), tbm := b, \{b\} \textbf{ fi} >;$$
$$< marked, tbm := marked \cup b, \{\} >$$

Rely-collector can now use *tbm*.

$$(\forall (a,i) \in \textbf{dom}\, hp \cdot$$
$$hp'(a,i) \neq hp(a,i) \wedge hp'(a,i) \in Addr \Rightarrow$$
$$hp'(a,i) \in marked' \vee tbm' = \{hp'(a,i)\}) \wedge$$
$$(tbm \neq \{\} \wedge tbm' \neq tbm \Rightarrow tbm \subseteq marked' \wedge tbm' = \{\})$$

# Concurrency: Separation vs. Interference

- Result: Standard rely/guarantee conditions are not enough.
- Proposed solutions all break compositionality.
- This study makes it clearer what the core issues are.

  \- allows us to identify the features that make a program suitable for a particular logic.
- Similarly to the separation as abstraction work, this showed how interference carried through the refinement

  \- *possible values* was needed even at abstract levels.

# Concurrent Garbage Collector

Key observations:

The interference carried through all the way from the abstract model – shows that it is an inherent property of the problem. – Note that possible values were needed even on the abstract version.

Abstraction helps to reason about the core issues without worrying about program level details.

Compositionality cannot be fully maintained in the presence of strong inteference.

# Concurrency: Links between techniques

- Non-blocking algorithms such as the Treiber stack, Herlihy-Wing Queue

  - these also lie on the border of rely/guarantee.

  - Note that Simpson’s 4-slot algorithm can be verified.

- Investigating the common properties and how they can be verified using rely guarantee is important.

- There appears to be a link between rely/guarantee and linearisability.