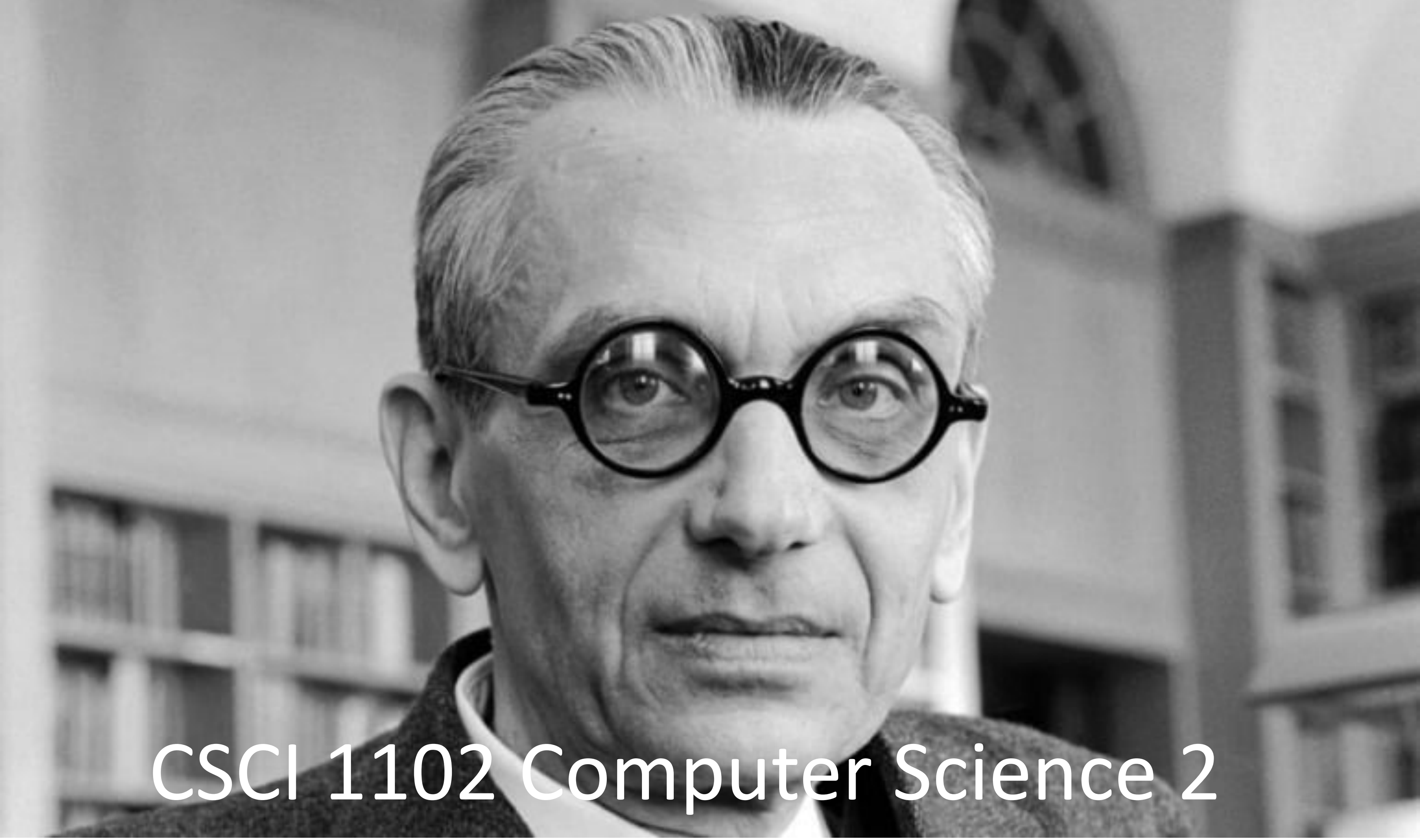
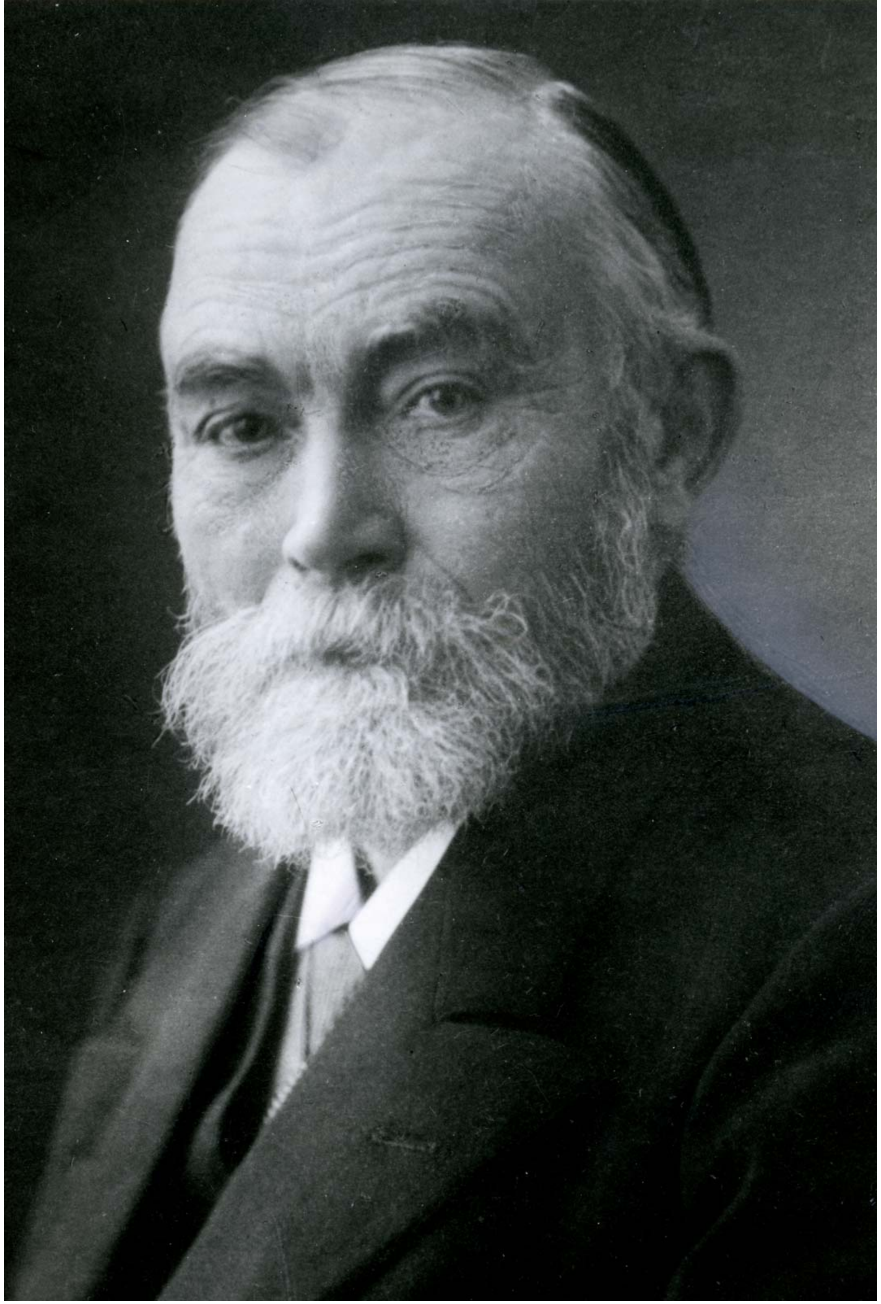
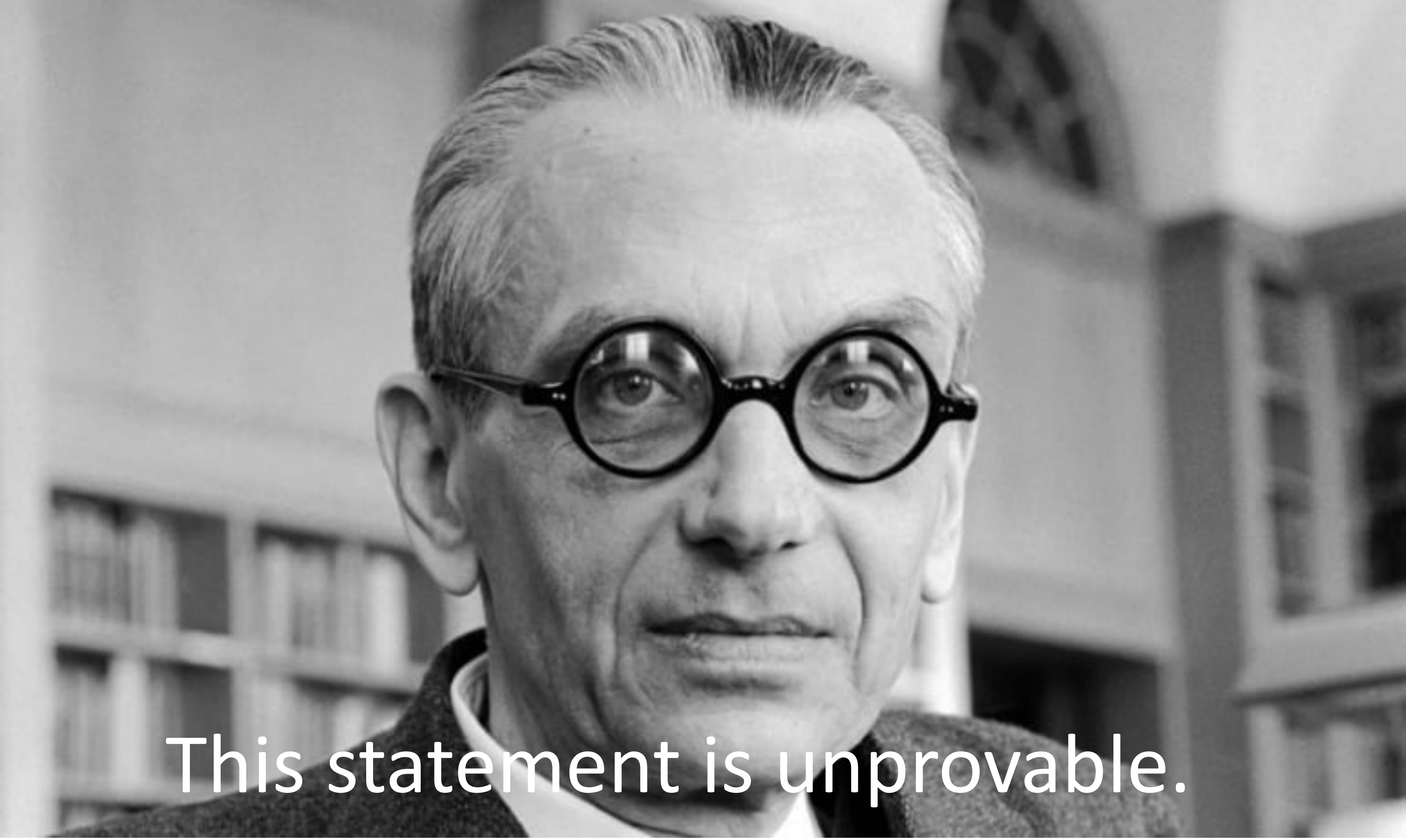


# CSCI 1102 Computer Science 2

## Meeting 12: Thursday 3/11/2021

## Recursion & Iteration; Mutability; Order





This statement is unprovable.

# Today

- Quiz Review
  - Iteration and Recursion
  - Mutation

- $2^N$ and $\log_2 N$

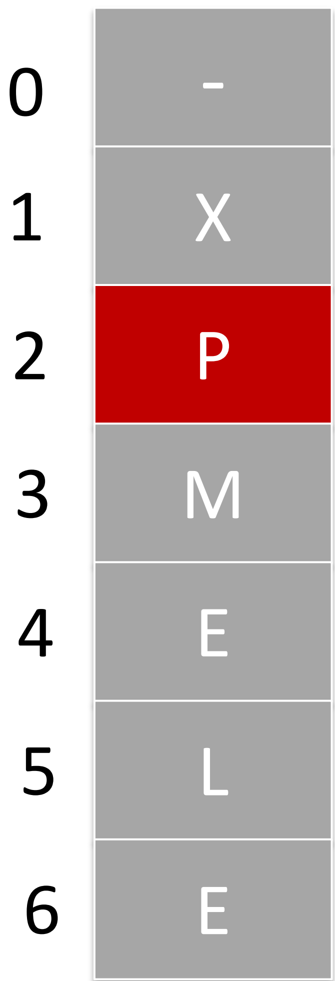
# PLs & their Native Ways

- Python & Java are *imperative* languages
  - imperative control forms `for`, `while` & mutable structures are natural;
  - recursive control & recursive structures are admissible but less natural in the case of Java and much less natural in the case of Python.

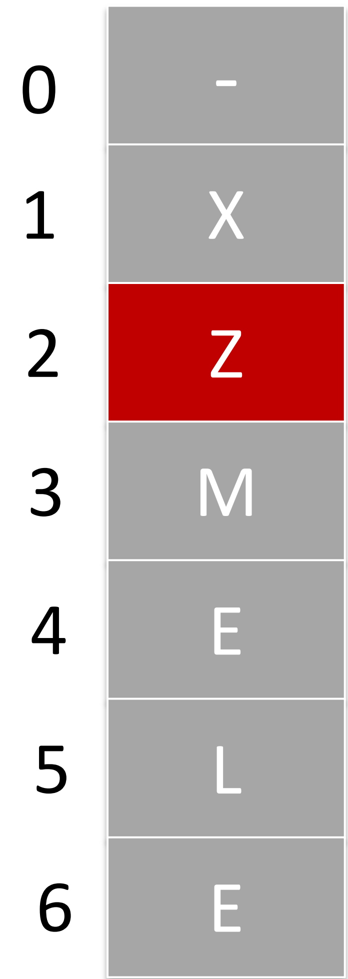
# PLs & their Native Ways

- Ocaml and related languages are *functional* (*value-oriented, expression-oriented*);
  - recursive control, pattern matching & immutable recursive structures are natural;
  - imperative forms & mutable structures are admissible but less natural.

# Mutability & Immutability

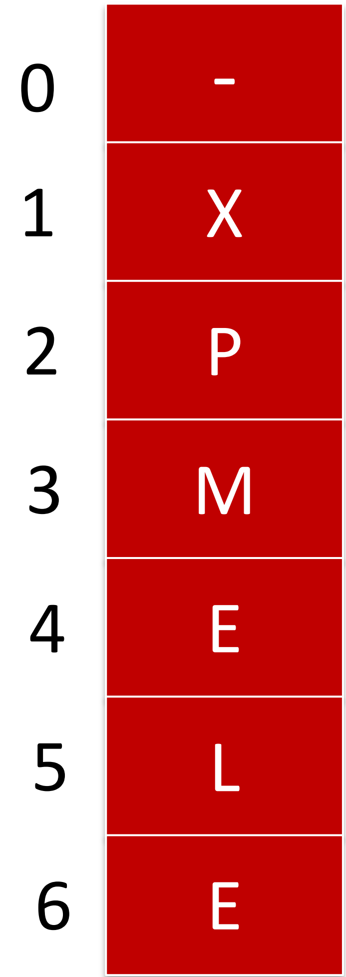



a[2] = Z;



# Generally Speaking …

If your concrete representation types involve block-storage/arrays, your first thought should be to consider imperative control forms `for` or `while`, probably together with mutation.



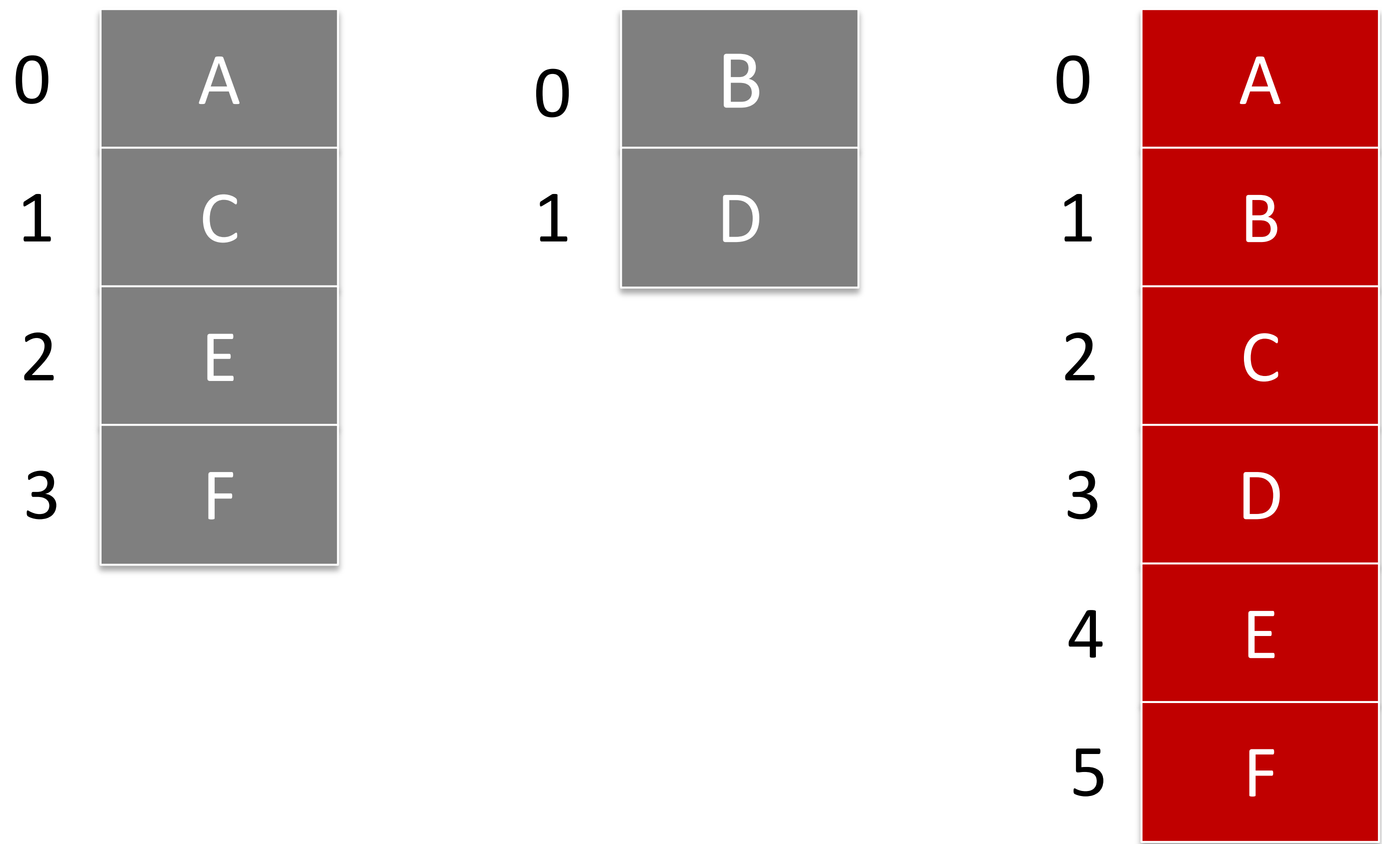
# Generally Speaking ...

If your concrete types are defined recursively, your first thought should be to use recursive control to process the recursively defined parts.

```
private class Node {
  T info;
  Node next;

  private Node(T info, Node next) {
    this.info = info;
    this.next = next;
  }
}
```

And you should carefully consider whether or not the data structure should be immutable.

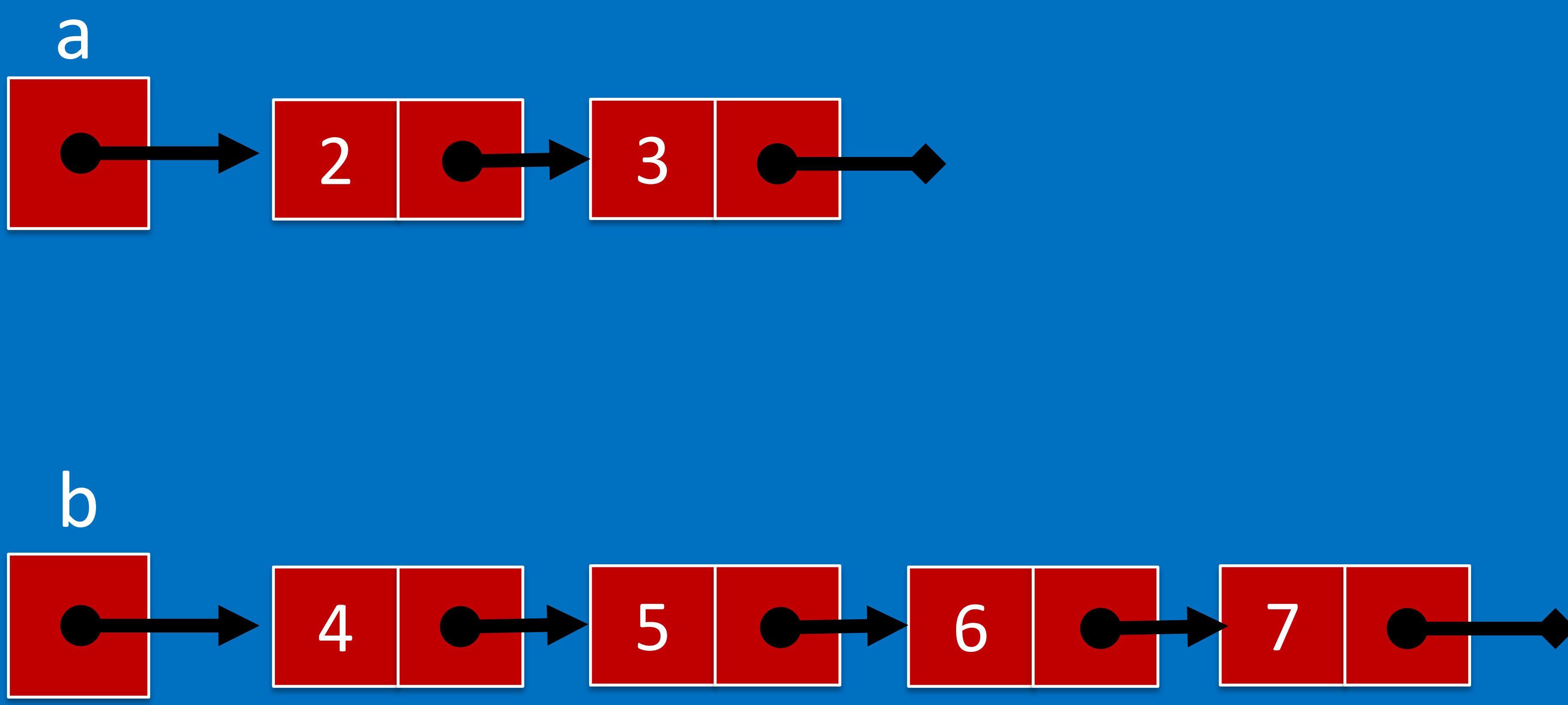
# Example: Merge 2 Ascending Arrays



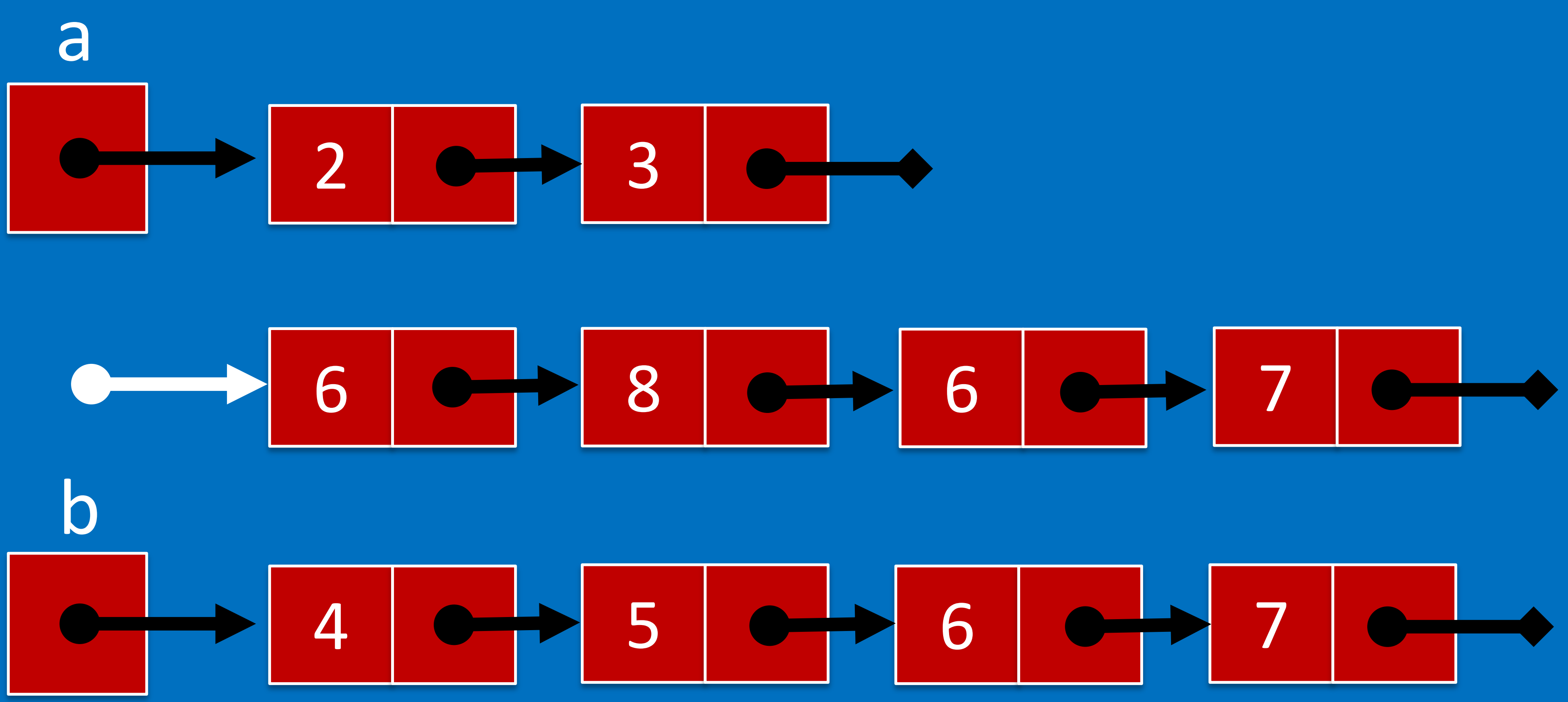
# Example: Merge 2 Ascending Arrays

```java
// 2.2B: merge two ascending sorted int arrays
public static int[] merge(int[] a, int[] b) {
    int[] c = new int[a.length + b.length];
    int i = 0, j = 0, k = 0;
    while (i < a.length && j < b.length)
        c[k++] = a[i] <= b[j] ? a[i++] : b[j++];
    while (i < a.length) c[k++] = a[i++];
    while (j < b.length) c[k++] = b[j++];
    return c;
}
```

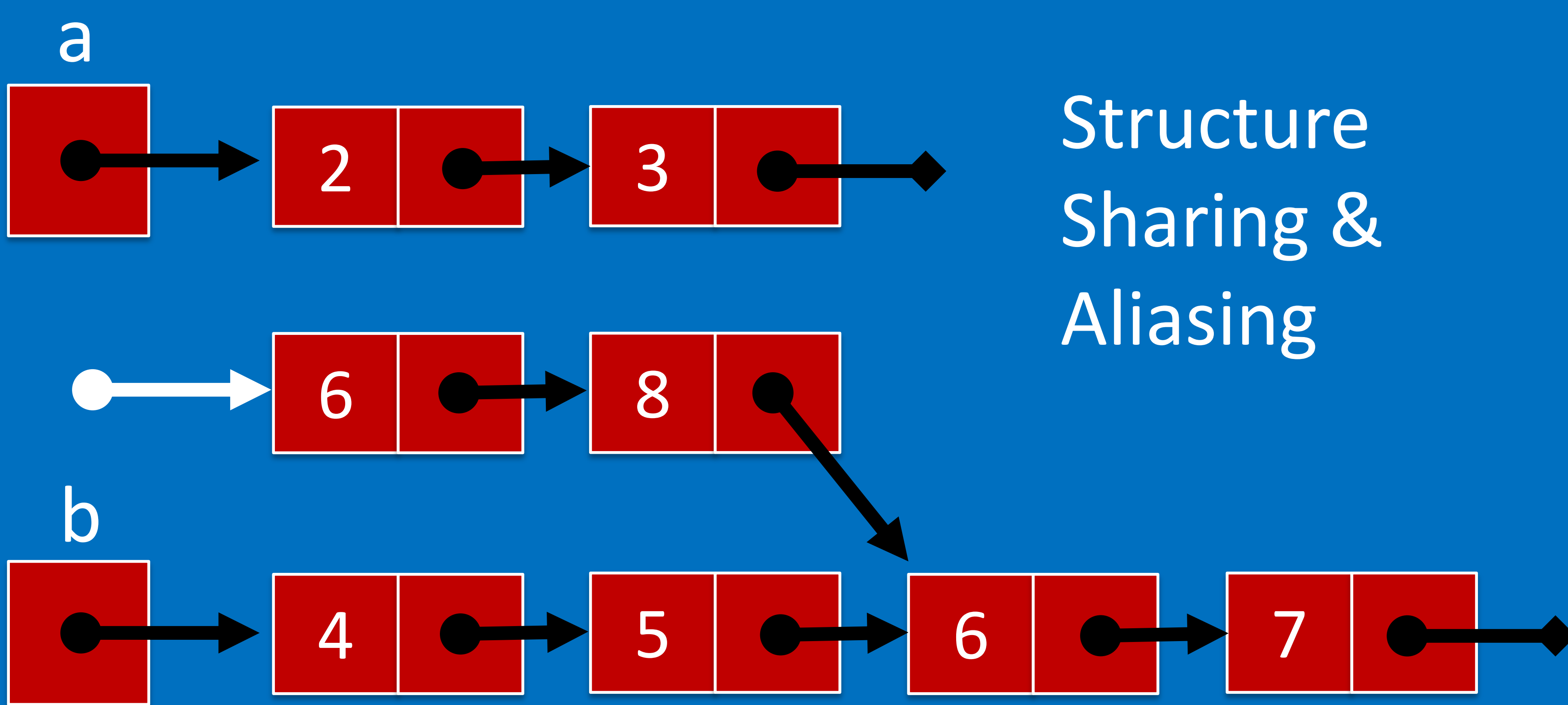
# Node add(Node a, Node b)



# Node add(Node a, Node b)



# Node add(Node a, Node b)




Structure Sharing & Aliasing

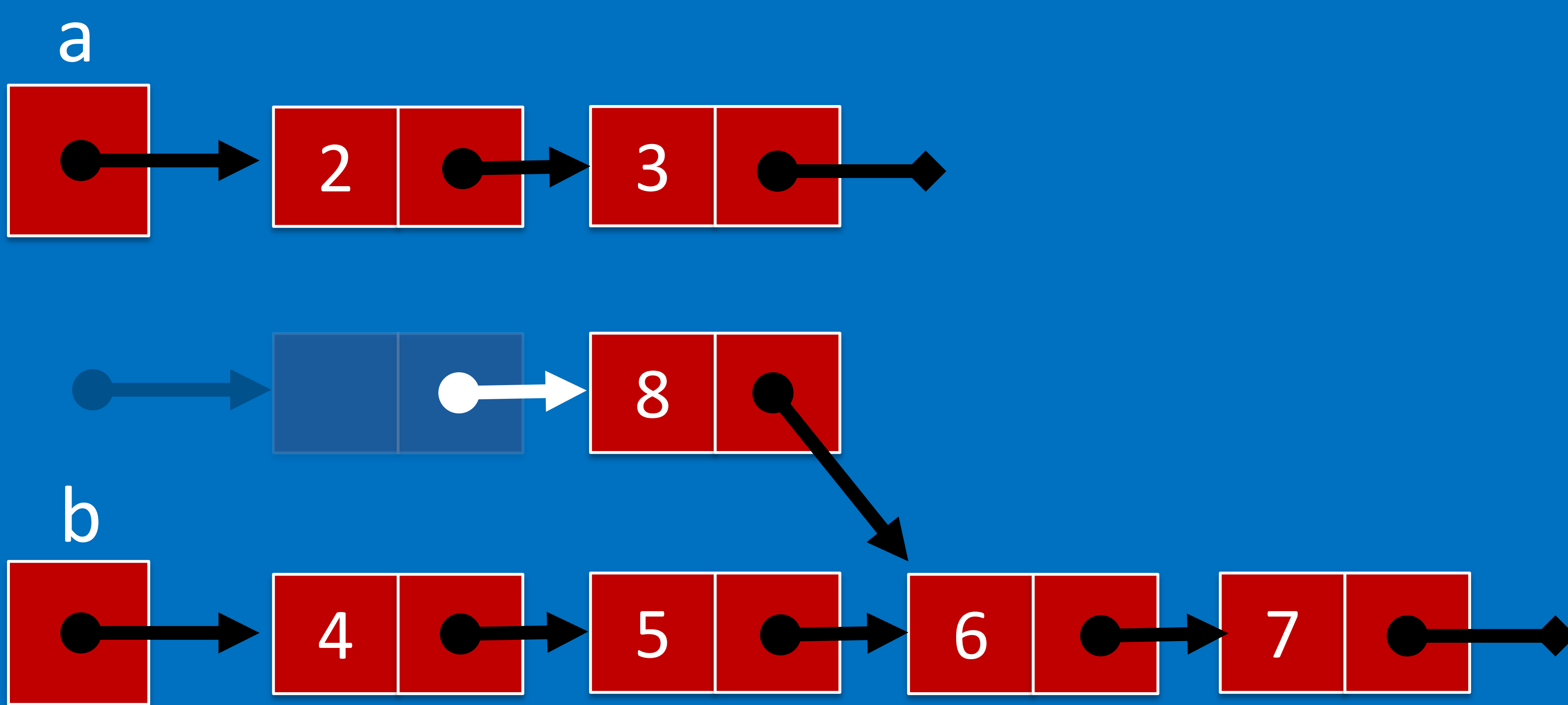
```java
private static Node add1(Node a, Node b) {
  if (a == null) return b;
  if (b == null) return a;
  Node
    result = new Node(0, null),
    current = result;
  while (a != null & b != null) {
    current.info = a.info + b.info;
    a = a.next;
    b = b.next;
    if (a != null && b != null) {
      current.next = new Node(0, null);
      current = current.next;
    }
  }
  current.next = (a == null) ? b : a;
  return result;
}
```

```java
private static class Node {
    int info;
    Node next;

    private Node() {}

    private Node(int info, Node next) {
        this.info = info;
        this.next = next;
    }
}
```

# How to Think about Recursion

- No sweat, if I handle the base case(s) correctly, a recursive call on a recursively defined field will always give me a complete, good-to-go result for the slightly smaller part;

# add(a.next, b.next)

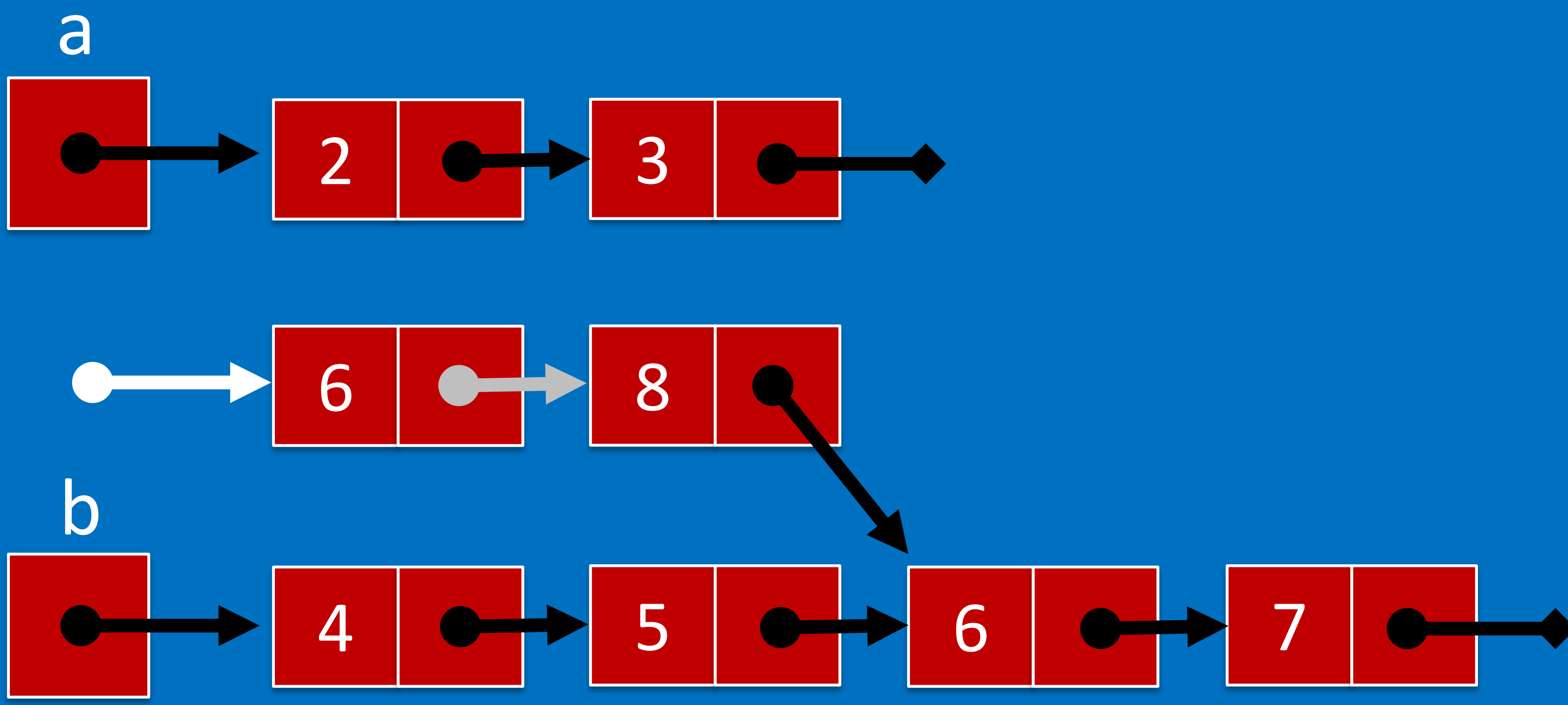


# How to Think about Recursion

- No sweat, if I handle the base case(s) correctly, a recursive call on a recursively defined field will always give me a complete, good-to-go result for the slightly smaller part;

- Now what additional work needs to be done to use this smaller result to get the full result?

new Node(a.info + b.info, add(a.next, b.next))



```java
private static Node add1(Node a, Node b) {
    if (a == null) return b;
    if (b == null) return a;
    Node
        result = new Node(0, null),
        current = result;
    while (a != null & b != null) {
        current.info = a.info + b.info;
        a = a.next;
        b = b.next;
        if (a != null && b != null) {
            current.next = new Node(0, null);
            current = current.next;
        }
    }
    current.next = (a == null) ? b : a;
    return result;
}
```

```java
private static Node add1r(Node a, Node b) {
  if (a == null && b == null) return null;
  if (a == null) return b;
  if (b == null) return a;
  return new Node(a.info + b.info, add1r(a.next, b.next));
}
```

# How to Think about Recursion

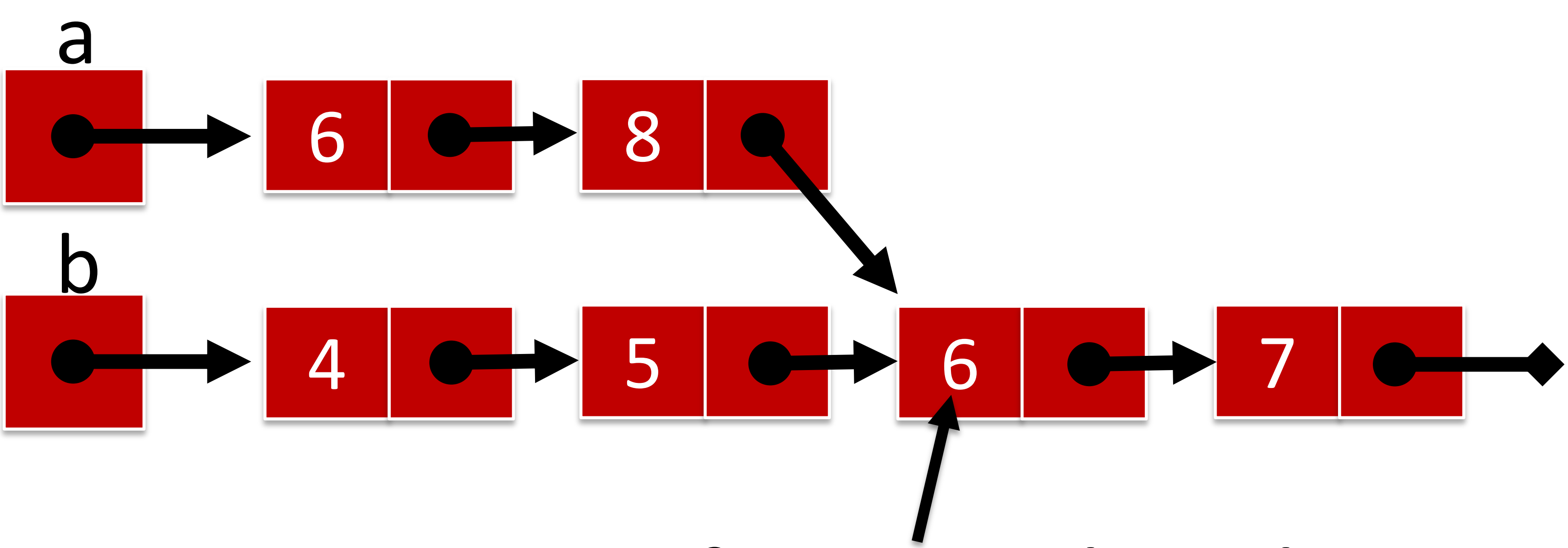
- In addition to being the natural way to process recursively defined structures, recursion is natural for other ordered types such as non-negative integers;

- There are many, many other cases where the algorithms are naturally recursive.

# Issues with Mutation

# Issues with Mutation

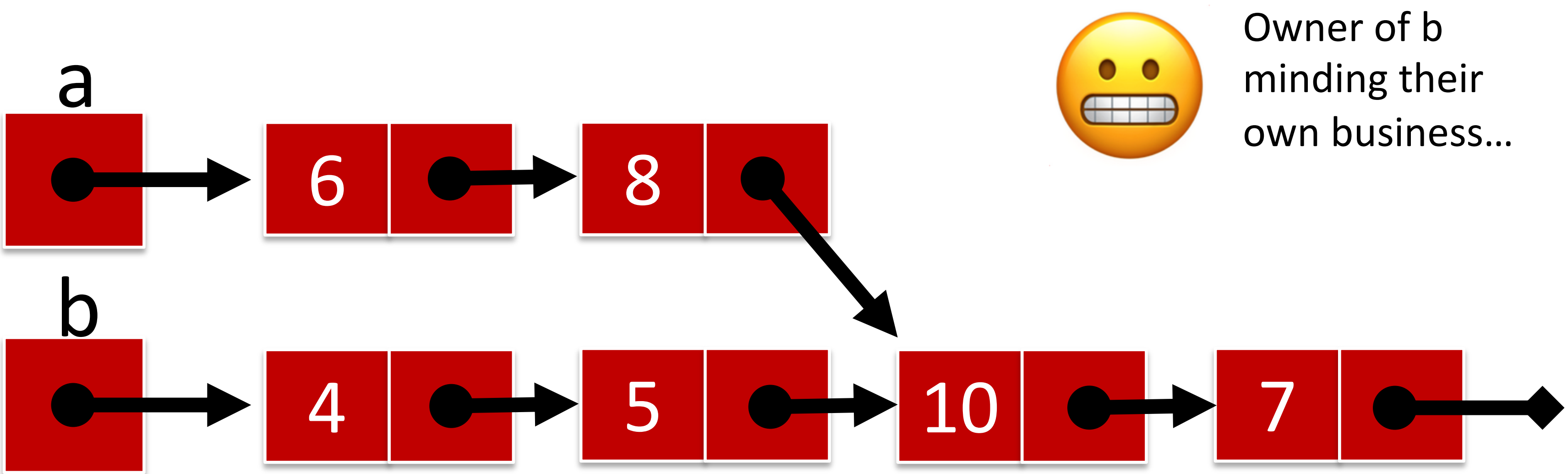
- Mutation complicates/disqualifies structure sharing




a.next.next.info = 10;  alters b

# Issues with Mutation

- Mutation complicates/disqualifies structure sharing




a.next.next.info = 10;  alters b

# Issues with Mutation

Mutation can *silently* corrupt order sensitive data structures.

```
class Point {
  private int x, y;

  // ...
  public void move(int dx, int dy) {
    this.x += dx;
    this.y += dy;
  }

  public int compareTo(Point other) {
    return this.x.compareTo(other.getX());
  }
}
```

# Issues with Mutation

```
void keysInOrderedDataStructuresMustBeImmutable() {
  Point p1 = new Point(2, 3);
  Point p2 = new Point(4, 5);

  PriorityQueue<Point> pq = new BinaryHeap<Point>();
  pq.enqueue(p1);
  pq.enqueue(p2);
  p1.move(500, 0);
}
```

# Issues with Mutation

- Mutation compromises *compositional reasoning* about code.

```
int z = f(g(x));
```

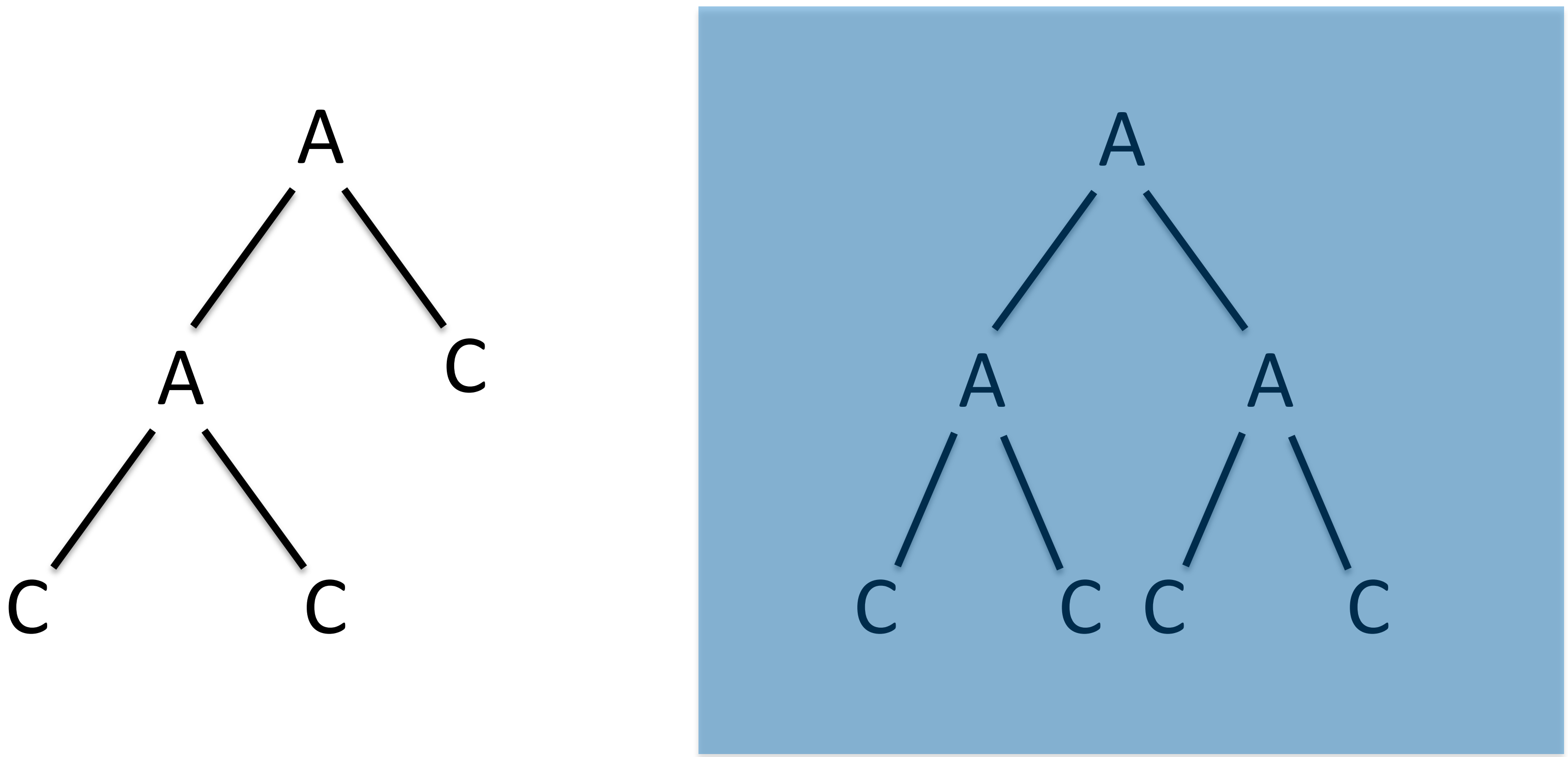
# Issues with Mutation

- Mutation greatly complicates multi-threaded code.

- Mutation generally superimposes a web of **dependencies** in code and the people and organizations involved with the code.

# Reasonable & Unreasonable Numbers

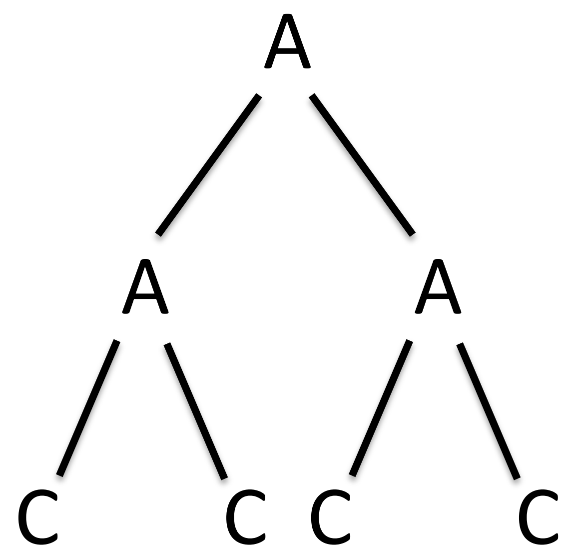
# *Perfect Binary Tree* – All depths are full



# Properties of Perfect Binary Trees

A perfect binary tree of height k has

- $2^k$ leaves
- $2^k - 1$ interior nodes
- $2^{k+1} - 1$ nodes



# Two Sides of the Coin

- A perfect binary tree of height k, has $2^k$ leaves. For not very large k, $2^k$ can be stupendously large.
- A perfect binary tree with N leaves has height $\log_2 N$. Even for stupendously large N, $\log_2 N$ is quite manageable.
- For not very large height, visiting every leaf is infeasible but traveling from the root to a given leaf is very fast.

# Example: the exponent side

Assuming your algorithm can process a leaf node in a very zippy 1 nanosecond ($10^{-9}$ = 1 billionth of a second). Then processing all of the leaves of a perfect binary tree of height

- 60 takes …
- 70 takes …

# Example: the exponent side

Assuming your algorithm can process a leaf node in a very zippy 1 nanosecond ($10^{-9}$ = 1 billionth of a second). Then processing all of the leaves of a perfect binary tree of height

- 60 takes > 32 years;
- 70 takes ...

# Example: the exponent side

Assuming your algorithm can process a leaf node in a very zippy 1 nanosecond ($10^{-9}$ = 1 billionth of a second). Then processing all of the leaves of a perfect binary tree of height

- 60 takes > 32 years;
- 70 takes > 32 thousand years.

# Example: the $\log_2$ side

- Stuck with ~$10^{21}$ (one *sextillion*) data values? Can they be organized as nodes in a perfect binary tree?

- If so, you can get to any one of them in ~70 steps.