All About Trees

Quote of the week: “When you get into an argument, ask yourself if you want to be happy or if you want to be right, because there are hills you can die on that just aren’t worth fighting for.”

Project 1 grades…

- Hopefully out at the end of the week
- Also, we caught a couple of people cheating. Please confess before the 7th week for a lighter punishment

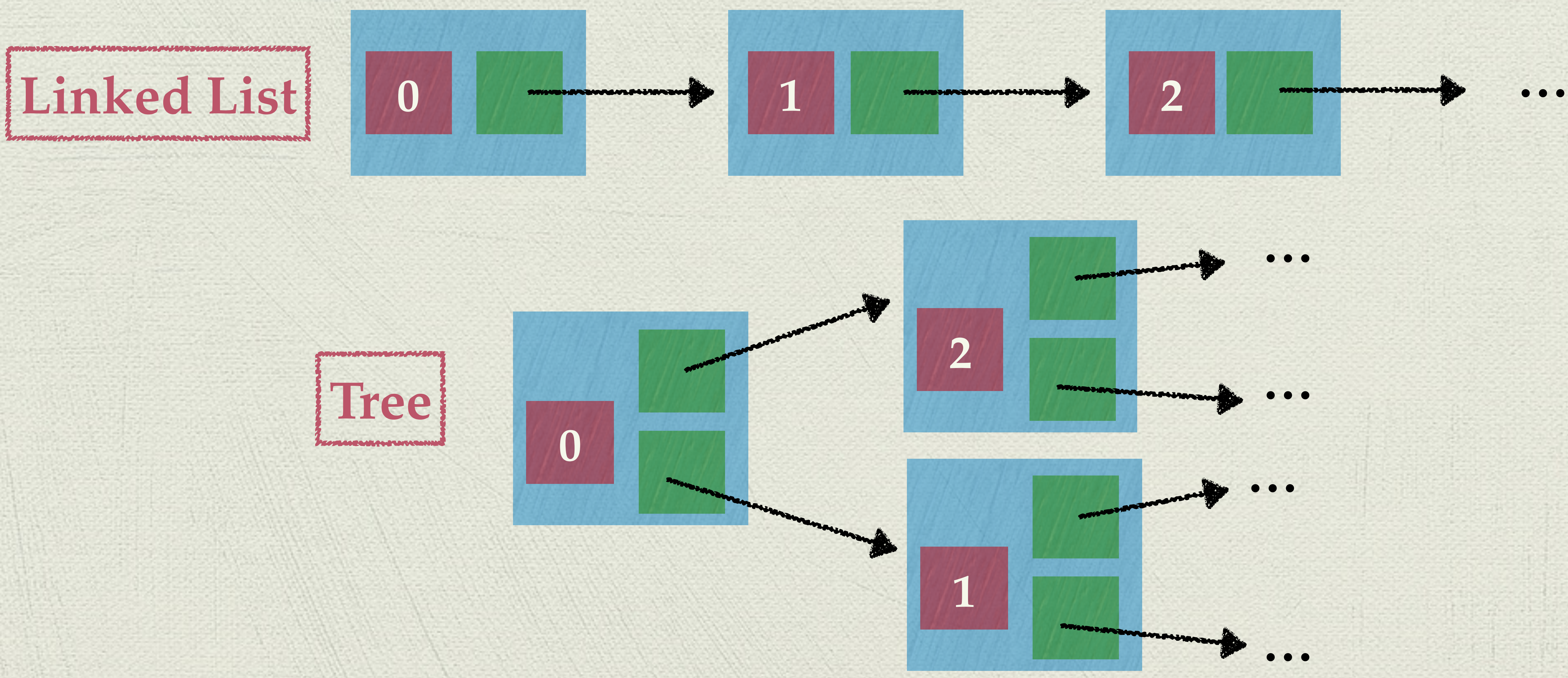
Midterm 2

- Is coming up in 1.5 weeks
- It's harder than midterm 1

So what's a tree?

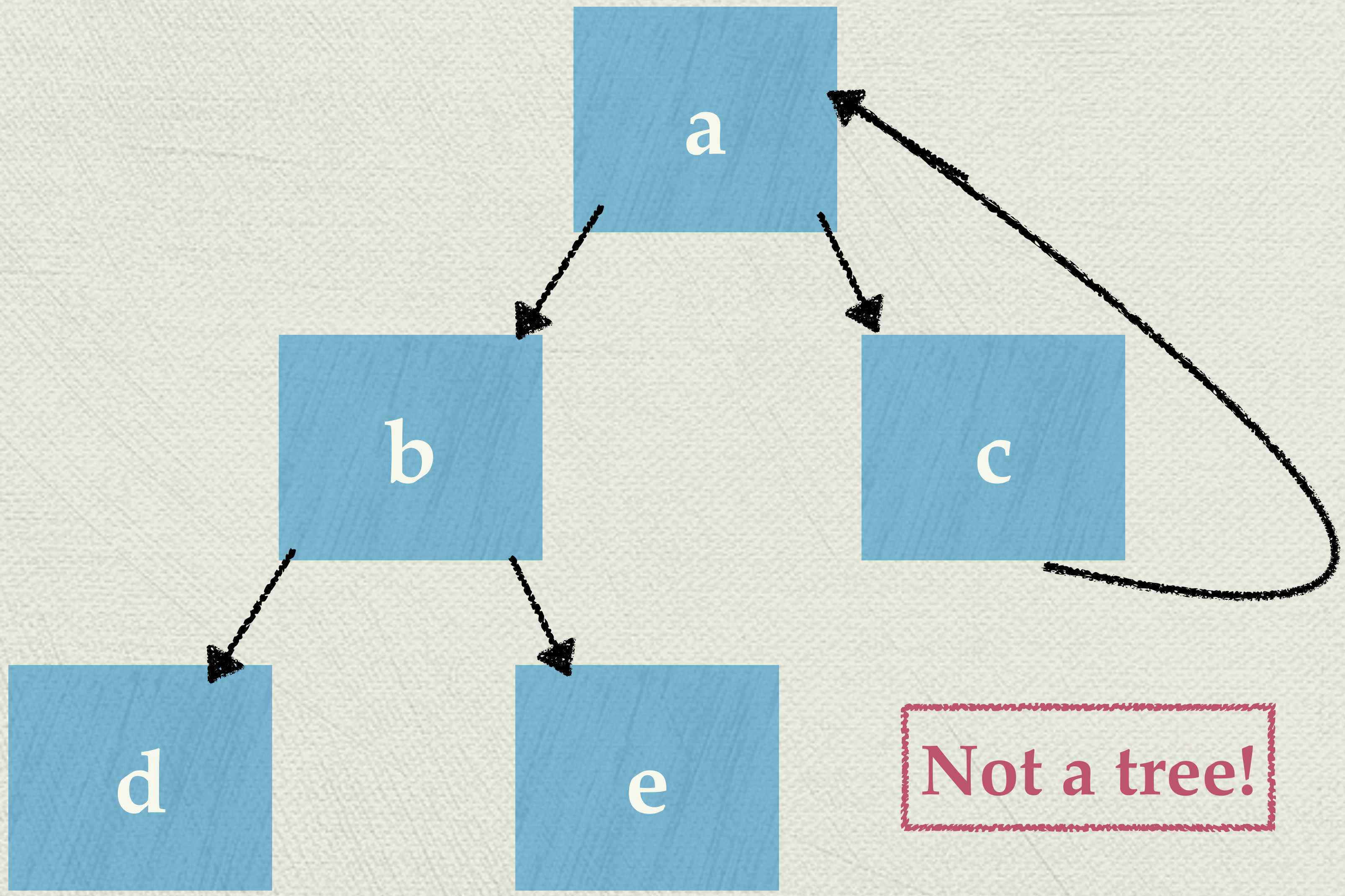
So what's a tree?

- Kinda like a linked list, except each node can have *multiple* nexts



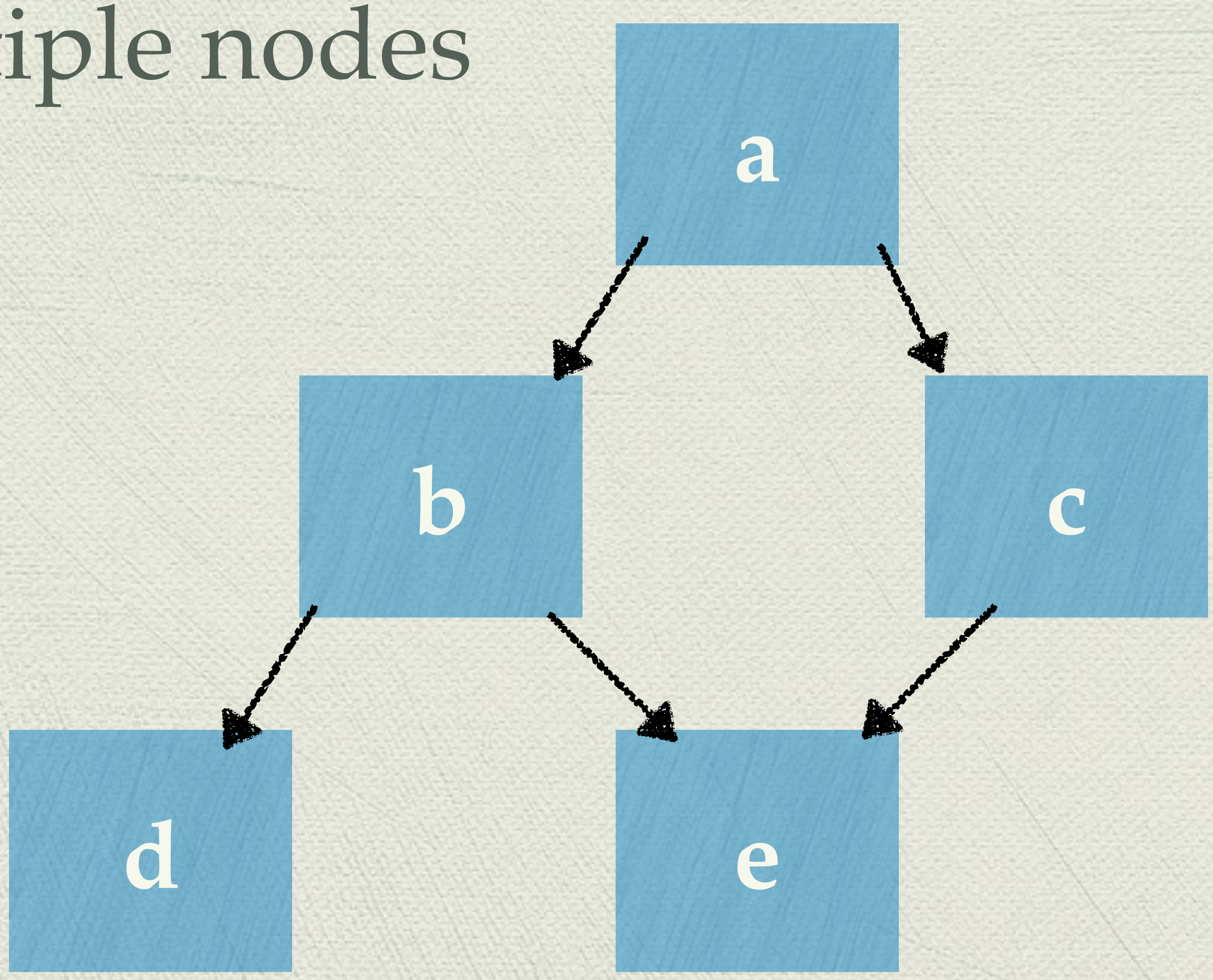
So what *is* a tree?

- **Special rule**: edges can't point back up the tree



So what's a tree?

- **Special rule**: nodes can't be descended from multiple nodes



Also not a tree!

So I noticed Java doesn't have a `Tree` class

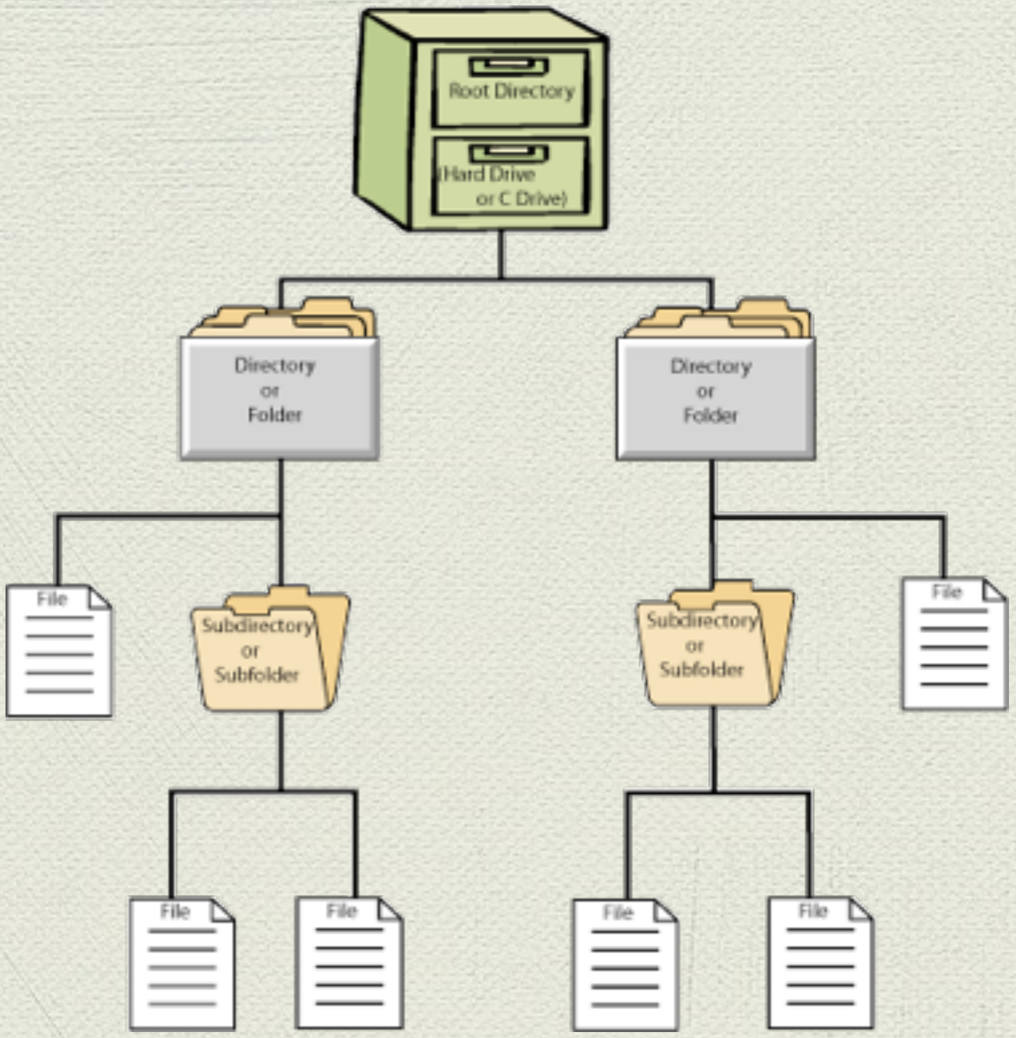
- Good observation!
- That's because we don't usually think of a tree as a container for data
- Instead, we use the metaphor that the data itself is implicitly organized as a tree

Tree examples

- In lab, you worked with an **amoeba family**
- Notice that if an `AmoebaFamily` contains an `Amoeba` object, and each `Amoeba` object contains references to its kids, then the data is *implicitly* organized like a tree
- We did *not* build a `Tree<Amoeba>`

Tree examples

- A file system, where every folder contains references to folders and files inside it, is implicitly a tree

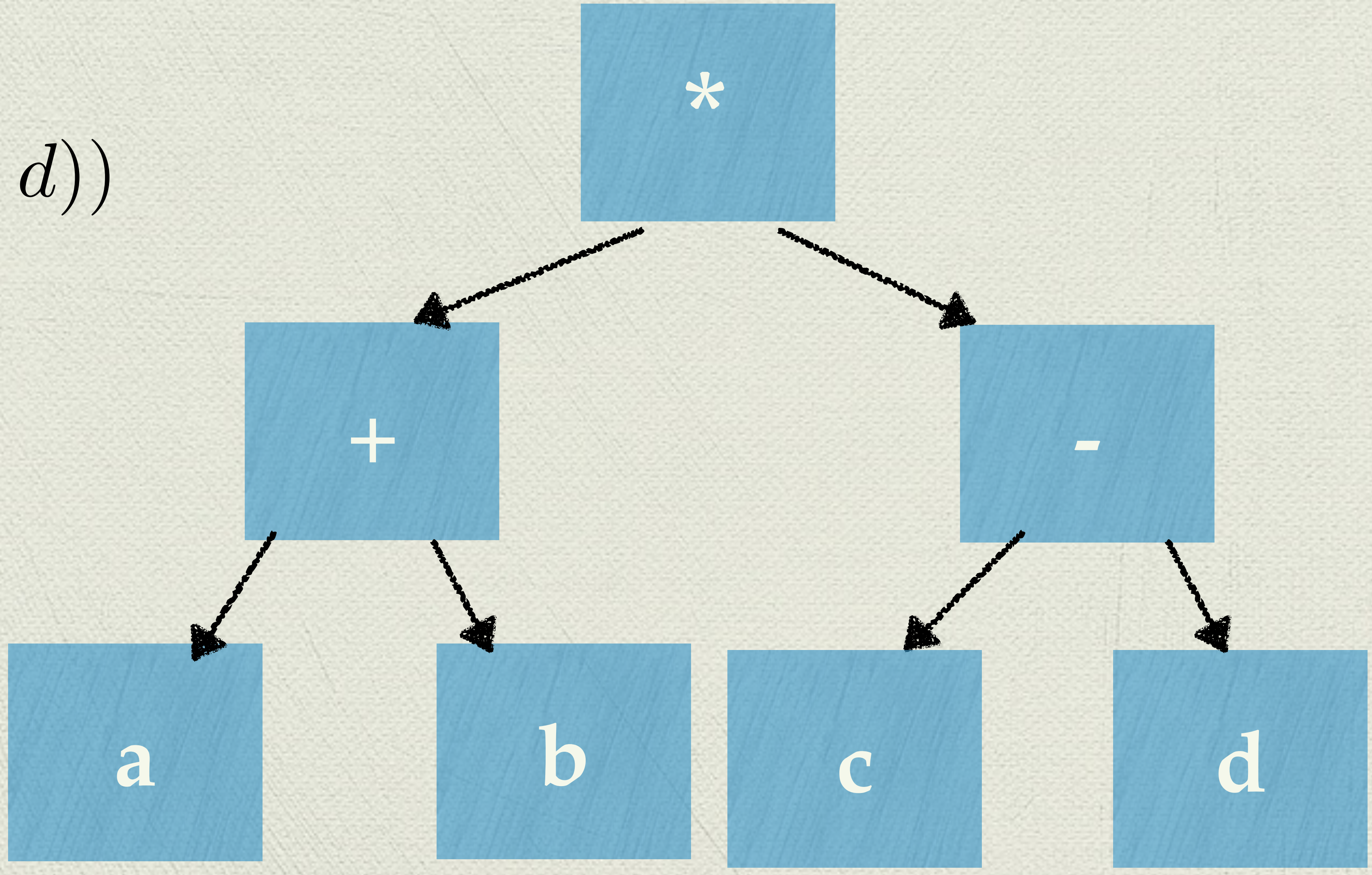


https://www.cs.colostate.edu/~cs155/Fall15/Lecture1

Tree examples

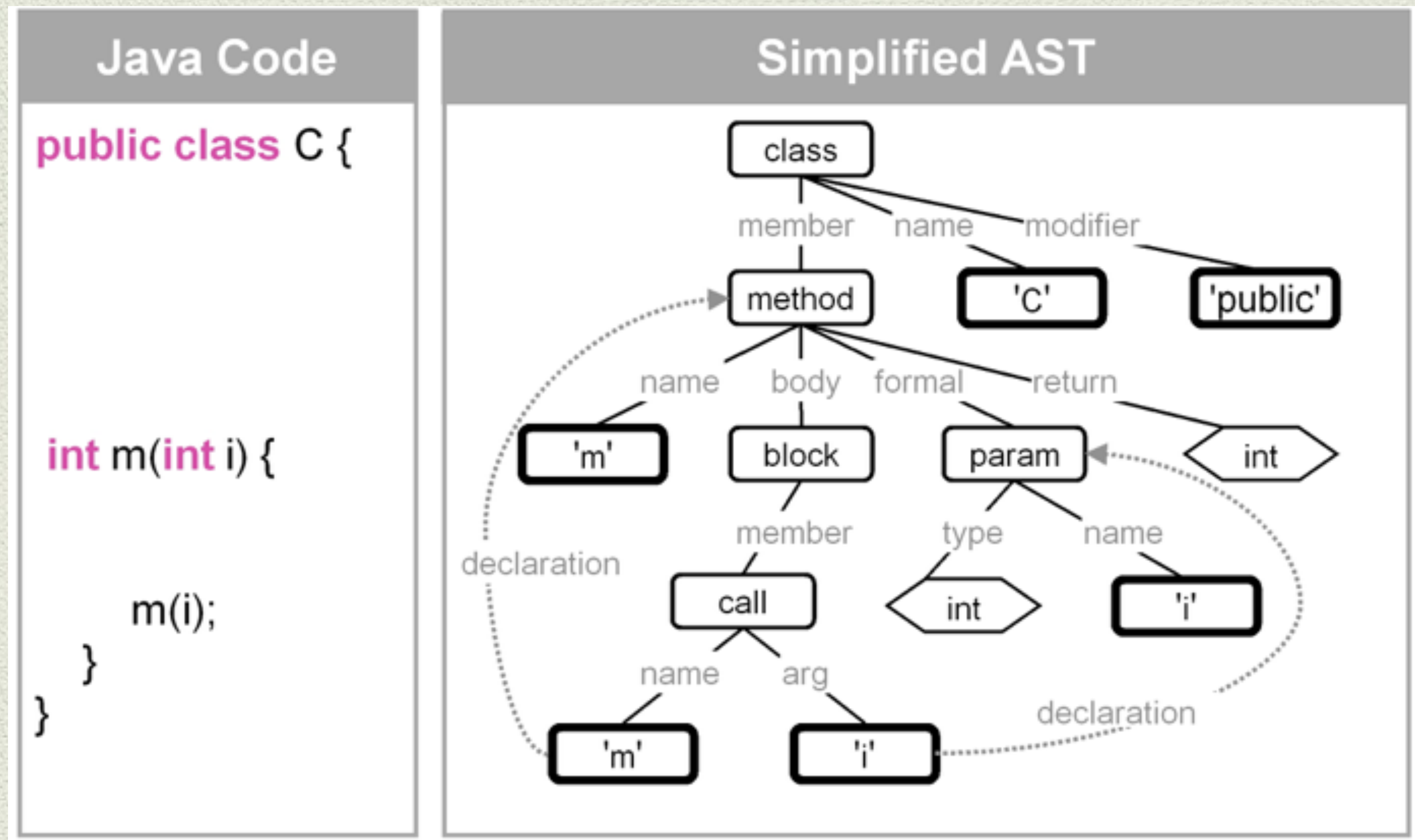
- A mathematical expression is implicitly a tree

$$((a + b) * (c - d))$$



Tree examples

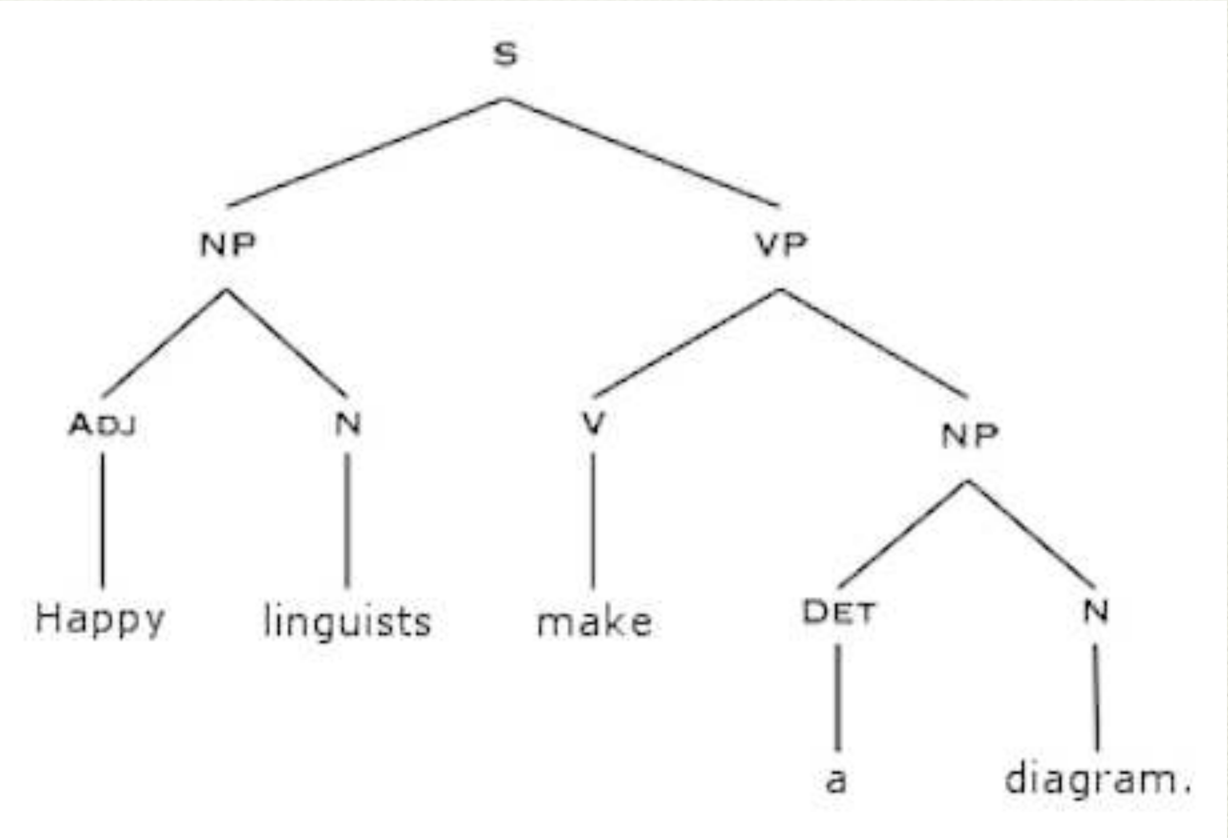
- Java code is a tree!
- Eclipse has library functions that can help you traverse it…



Credit: http://blog.brunobonacci.com/

Tree examples

- NLP researchers hope that human language can be modeled by a tree…

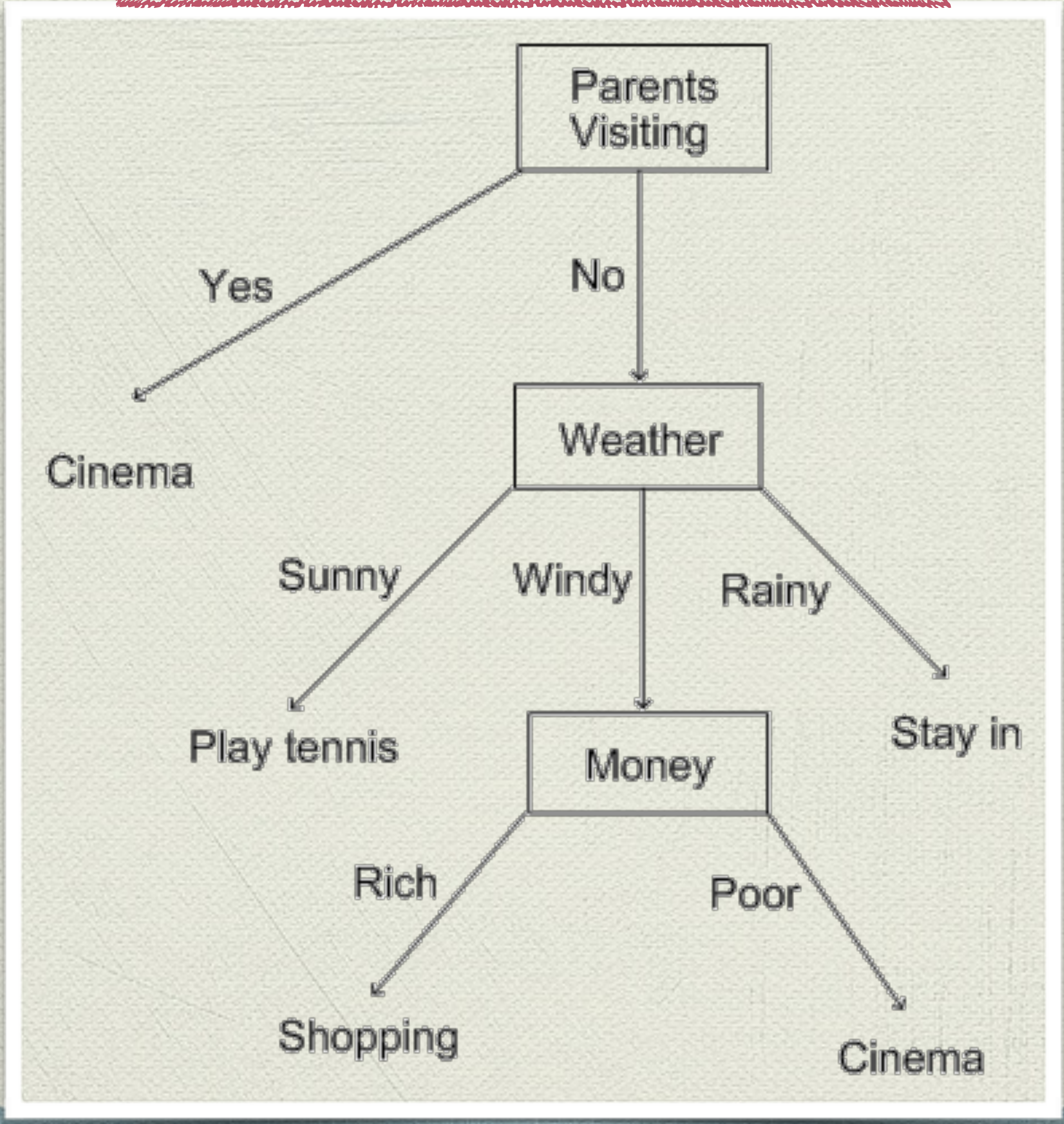


Credit: http://cdn.ymaservices.com/editorial_service/media/images/000/068/213/compressed/mGsNb.jpg?1415467331

Tree examples

- Decision-making processes are implicitly trees

What should I do today?



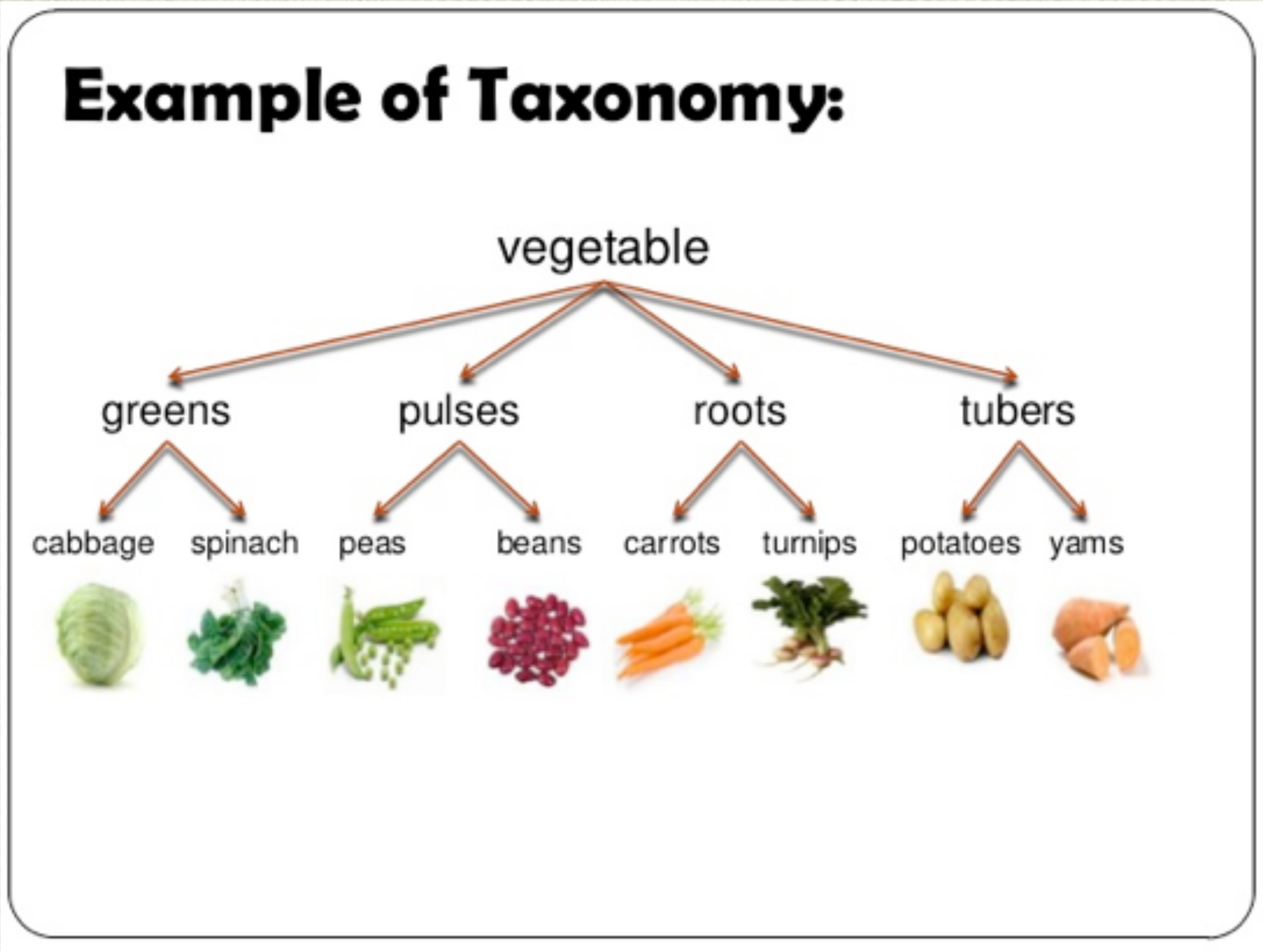
Credit: http://study.com/cimages/multimages/16/decision_tree.gif

Tree examples

- The sequence of possible moves you could make when playing checkers is implicitly a tree
- (We’ll see this in project 3)

Tree examples

- Categorization/typing systems are trees



Credit: http://www.slideshare.net/Andriyanieka12/13-semantics-synonym-antonym-homonym-hyponym-polyseme-idioms-18509523

Trees *can* be useful as containers for data…

- …but only in the service of another ADT.
- For example, we'll see how we can implement the **Map/Set** ADT using a tree (instead of hashing)
- We'll also implement the **Priority Queue** ADT using a tree (next lecture)

Representations

Tree representations

- Node-based
- Array-based (?!)

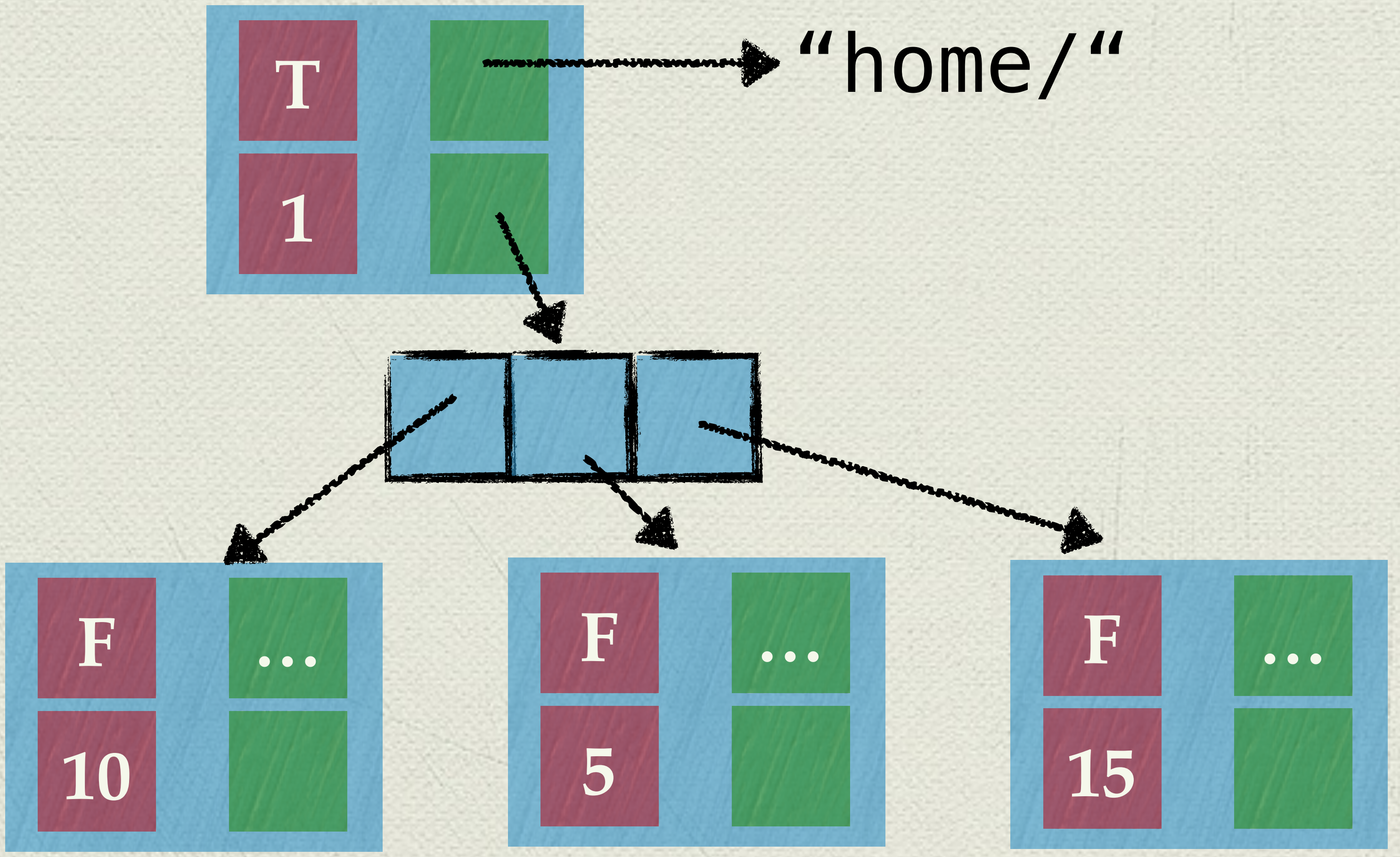
Nodes that can variable children

```
public class File {
  public String myName;
  public int mySize;
  public boolean isFolder;
  public File[] myContainedFiles;
}
```

Could have different number of contained files…

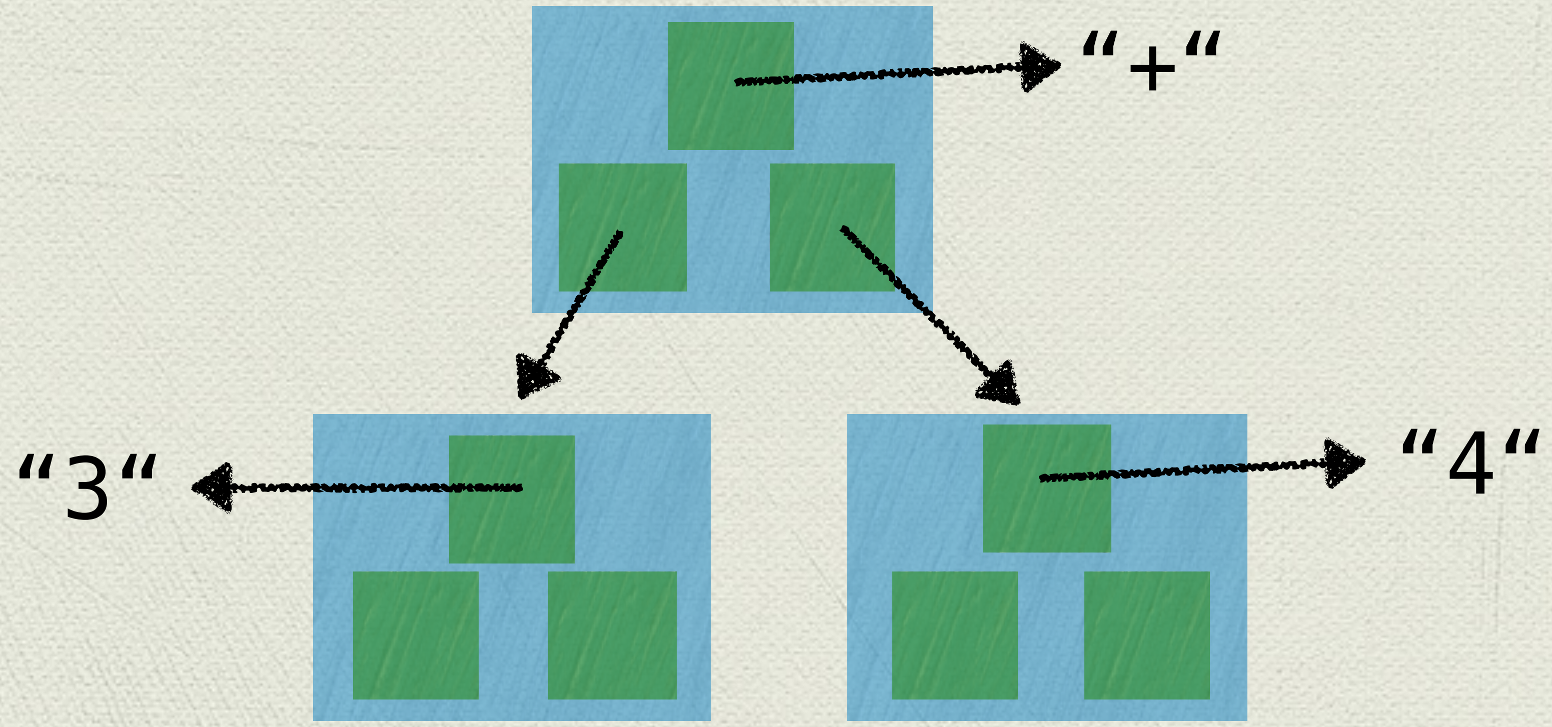
Nodes that can variable children

```
public class File {
  public String myName;
  public int mySize;
  public boolean isFolder;
  public File[] myContainedFiles;
}
```



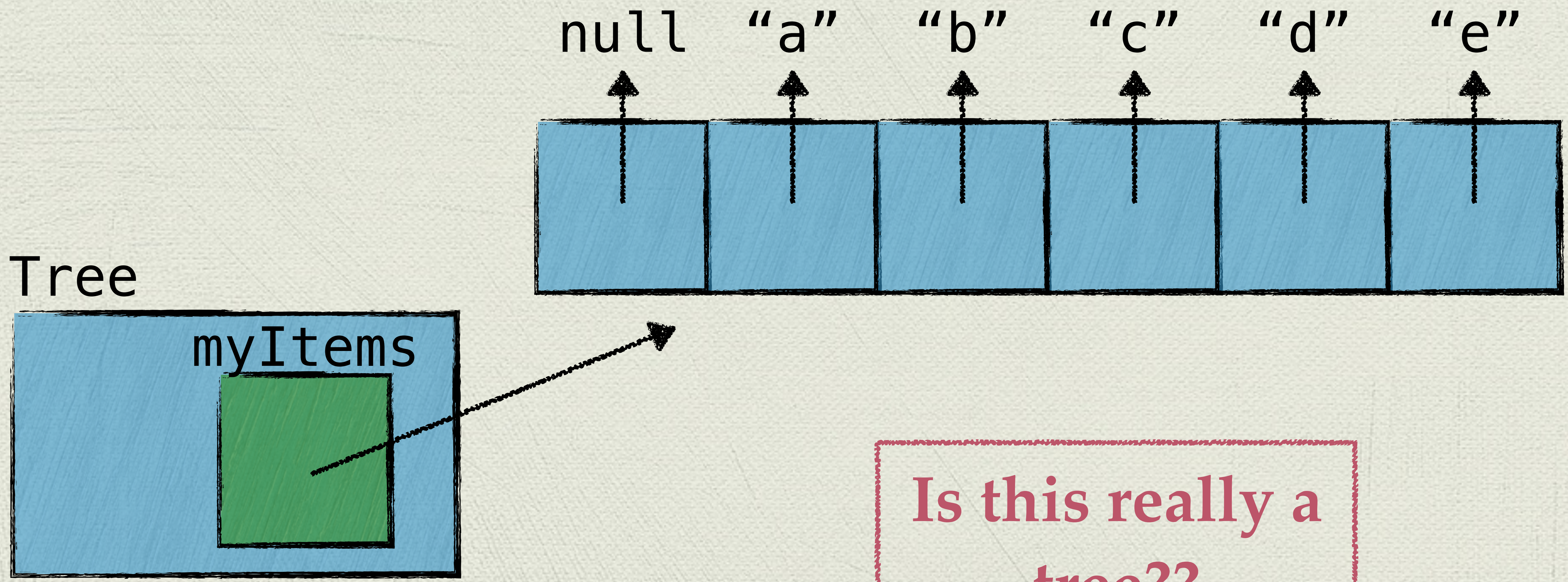
Nodes that always have two or fewer children

```
public class Expression {
    private String myItem;
    private Expression myLeftOperand;
    private Expression myRightOperand;
  }
}
```



A tree with an array?!

```
public class Tree {
  private Object[] myItems;
}
```

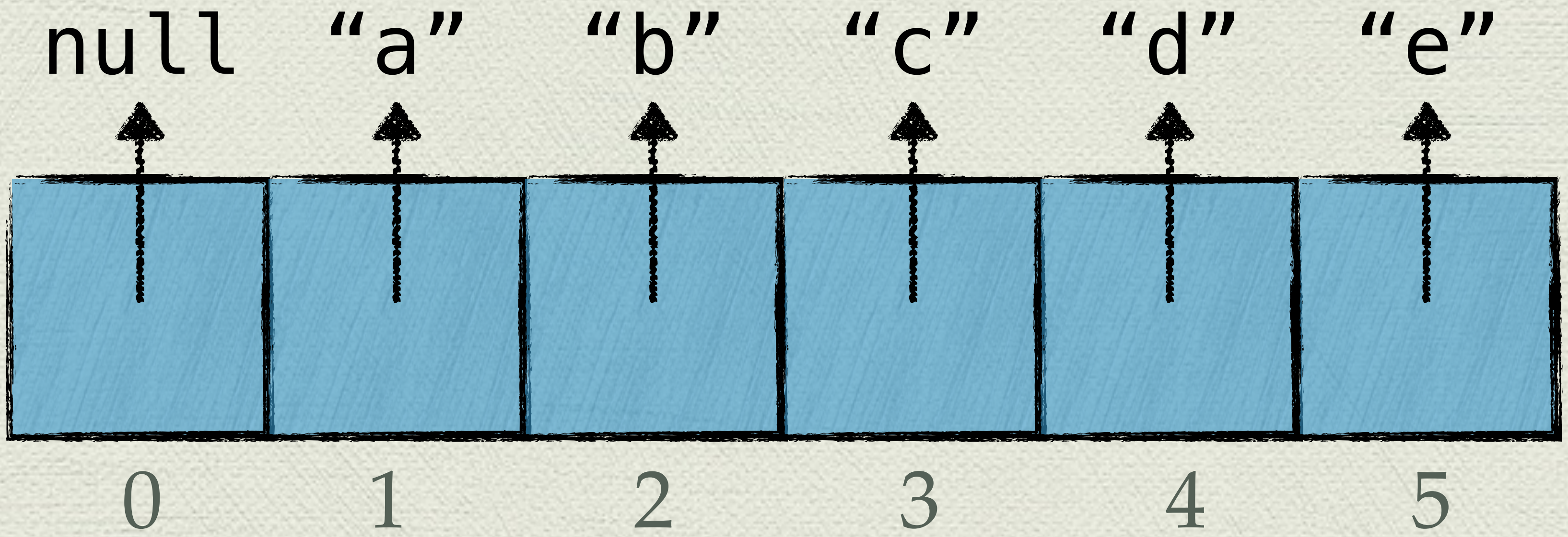


Is this really a tree??

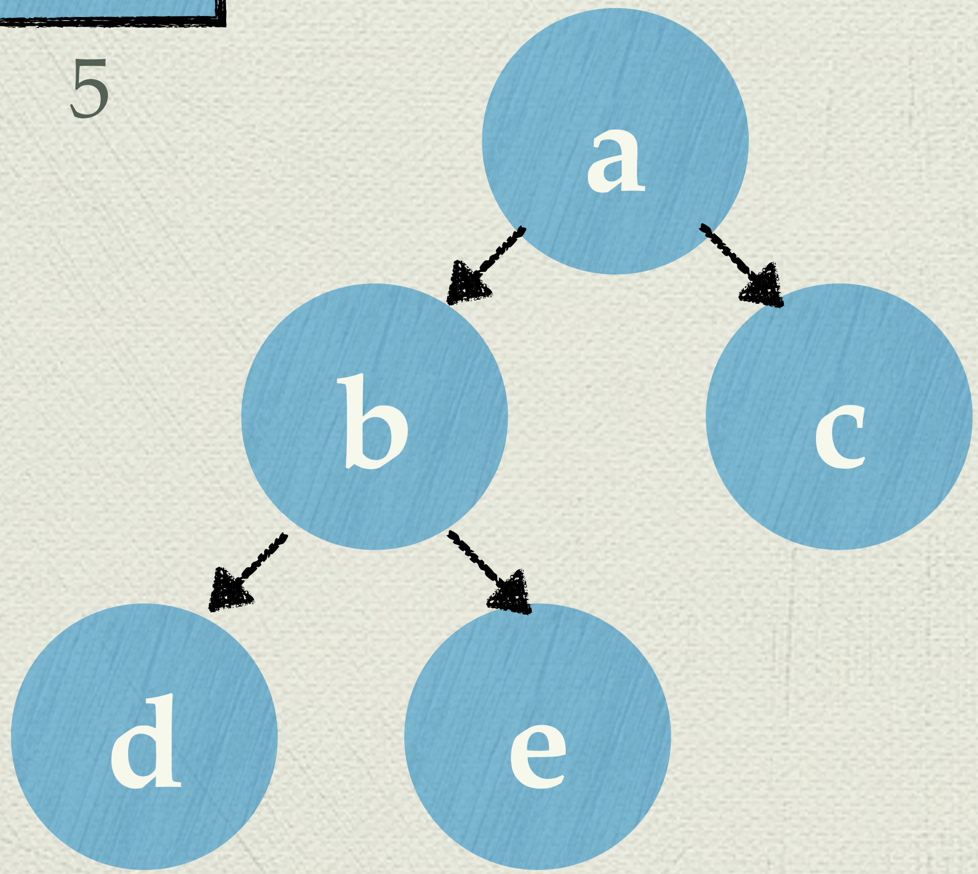
A tree with an array?!

- The secret:
  - `myItems[0]` is always null
  - `myItems[1]` is the root
  - the left child of `myItems[i]` is at `myItems[2*i]`
  - the right child of `myItems[i]` is at `myItems[2*i + 1]`

A tree with an array?!



The array represents this theoretical tree

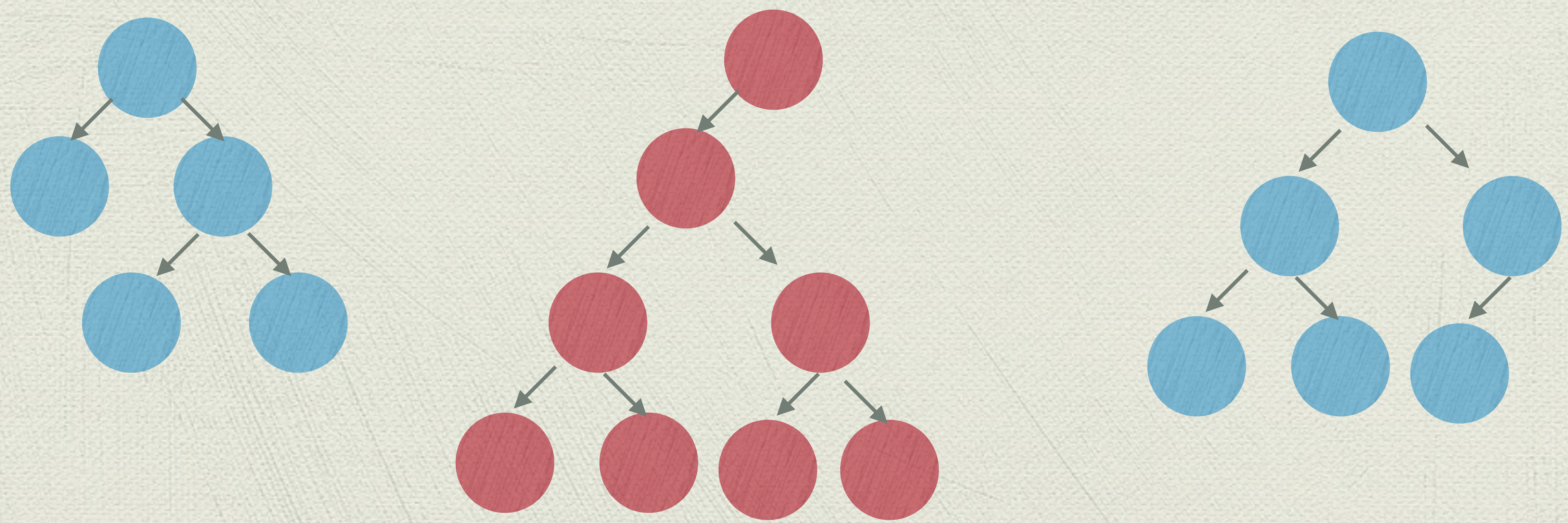


A tree with an array?!

- *Why?*
- Memory efficient *if there are no holes* in the middle of the array

A tree with an array?!

- Which tree below, if any, wouldn't have a "hole" in the array?
- Can you come up with a general rule?



Tree processing styles

Quiz time!

```
public class Tree {
   private TreeNode myRoot;

   private int shortestOddPath() {
      // TODO
   }

   /** Returns the min of any number of
   arguments. */
   private static int min(int... nums)
  {…}

   private class TreeNode {
      private int myItem;
      private TreeNode myLeft;
      private TreeNode myRight;
   }
}
```

- Complete `shortestOddPath`
- The length of the path is the sum of the myItems of nodes from root to leaf
- Only considers nodes at odd depths
- Find the shortest path from root to any leaf
- Pay attention to null checks and variable scope!!

Quiz time!

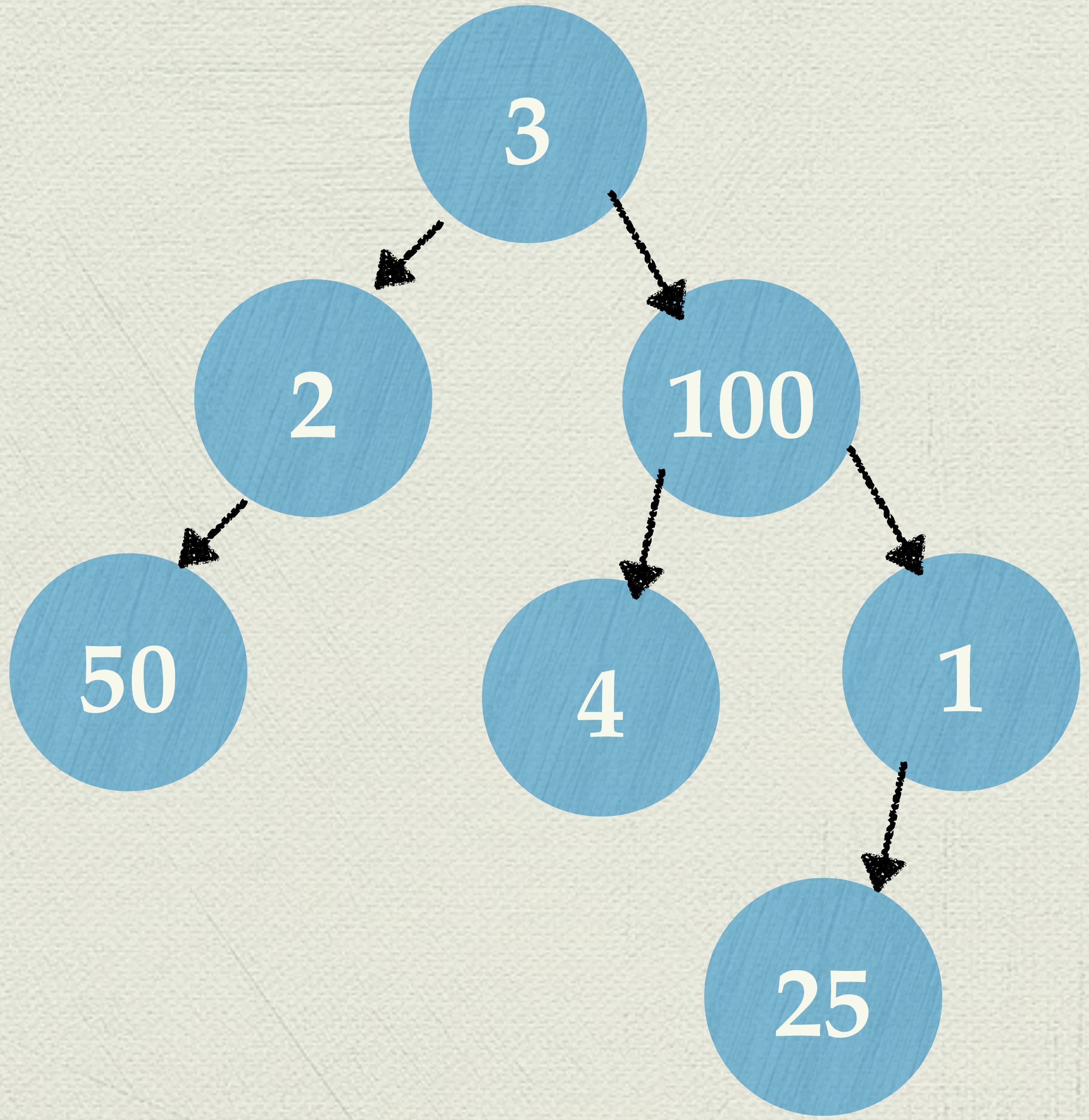
shortestOddPath is 4:

3 —> 1

Other paths are

3 —> 50

3 —> 4



Tree processing styles

- The point of this question was actually not the logic, but the style of your solution
- There are roughly *three* distinct stylistic approaches

Tree processing styles

- You essentially have three choices:
  - Null checks everywhere
  - Static helper methods
  - EmptyTreeNodes

Tree processing styles

- You essentially have three choices:
  - Null checks everywhere
  - Static helper methods
  - EmptyTreeNodes
- The more complicated your code gets, the more appealing EmptyTreeNode is. But for simple code, the former are appropriate

break

Map as a Tree

Maps

Before, we implemented a map using the concept of *hashing*…

- ```
  Map<String, Integer> h = new HashMap<String, Integer>();
  ```

Is there another option?
  ```

Introducing the tree map

- `Map<String, Integer> h = new HashMap<String, Integer>();`
- `Map<String, Integer> t = new` **`TreeMap`**`<String, Integer>();`

Introducing the tree map

- How is the tree map implemented?
- Well, a map is basically a **set** of key-value pairs, so let's see how a tree set is implemented…

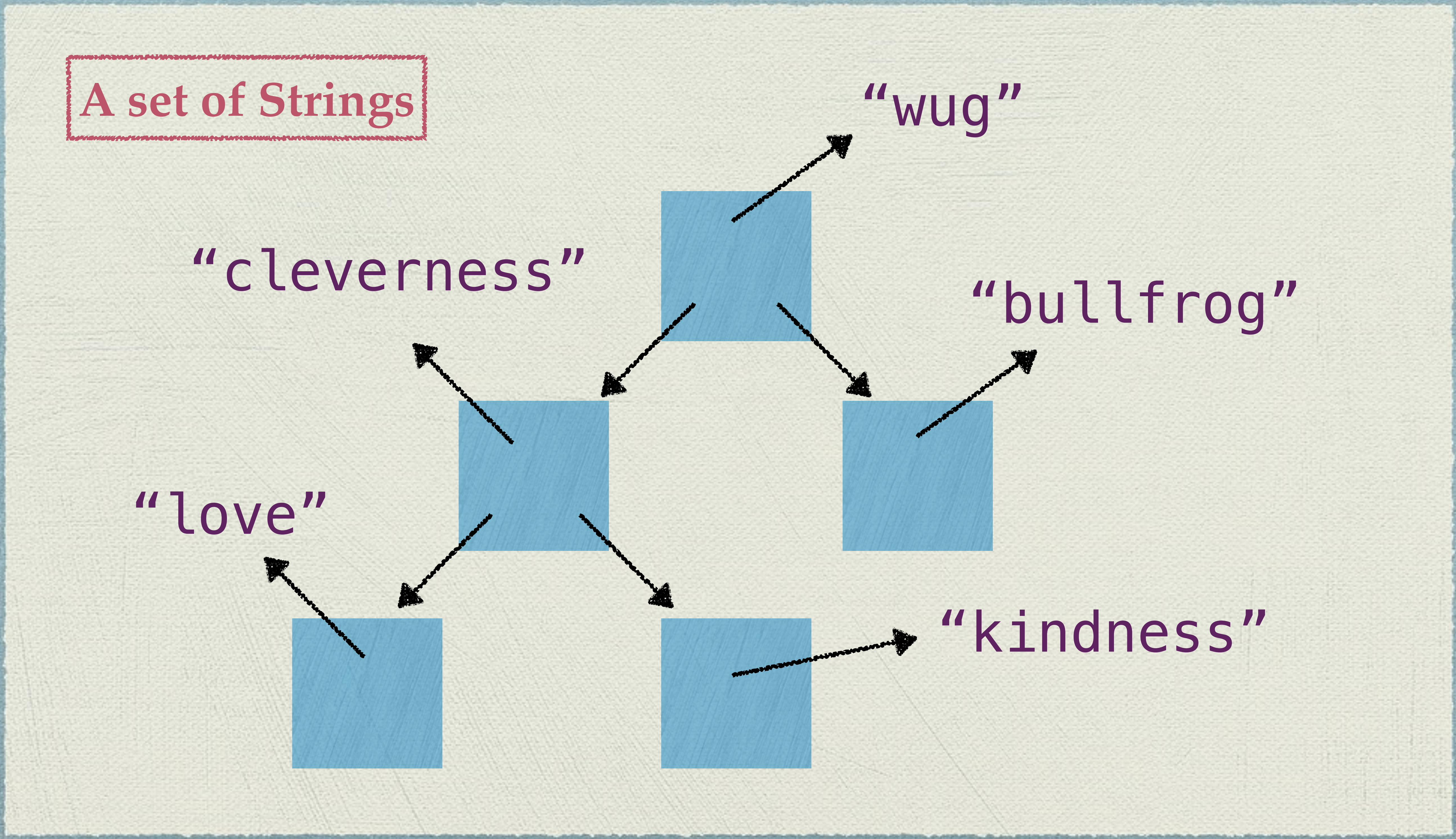
Yes, there is a tree set

- `Set<String> s = new TreeSet<String>();`

A tree… is a set?

- Remember, the main functionality of a set is to have an **add** and **contains** method (and **remove**)

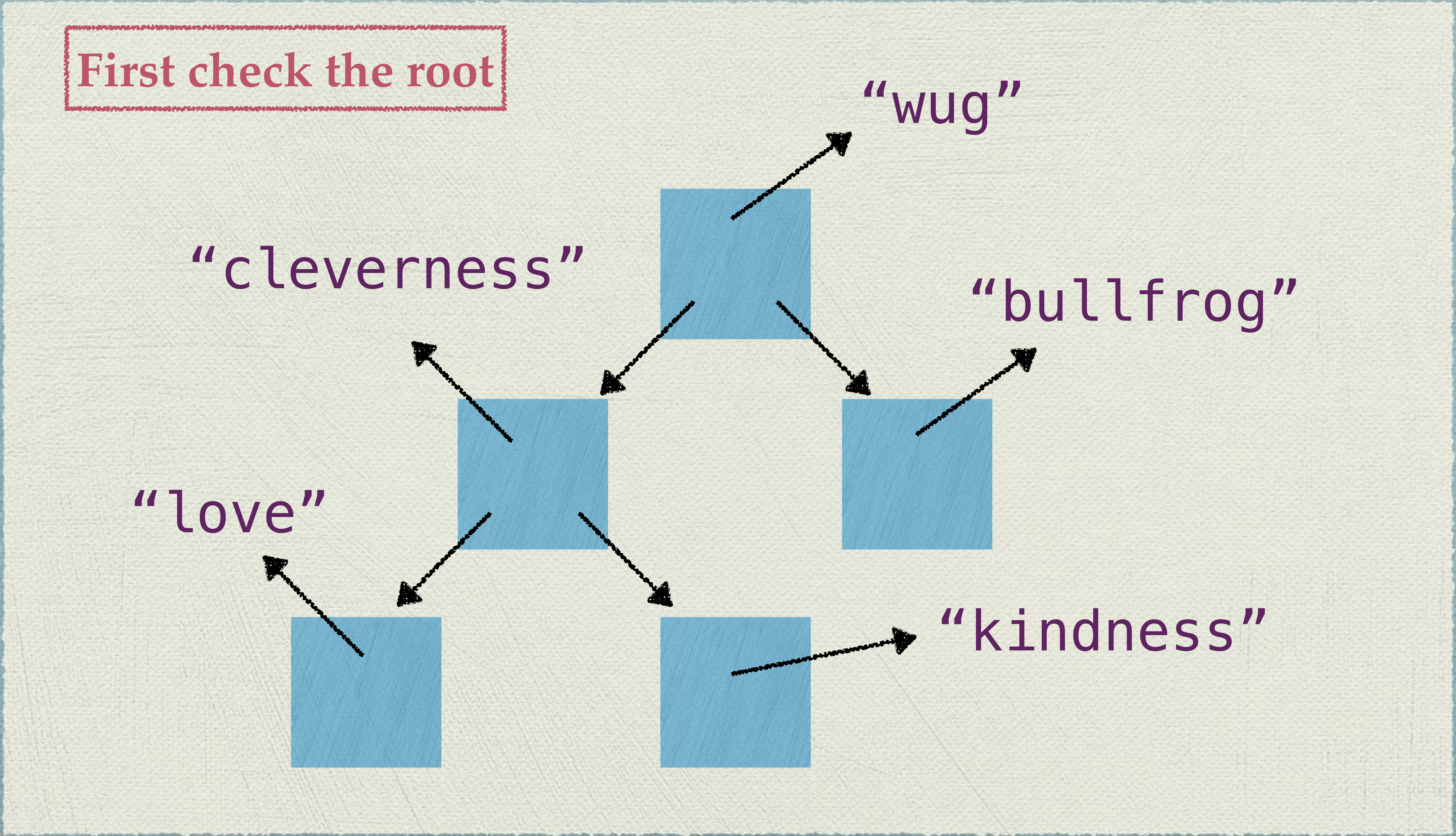
A tree… is a set?



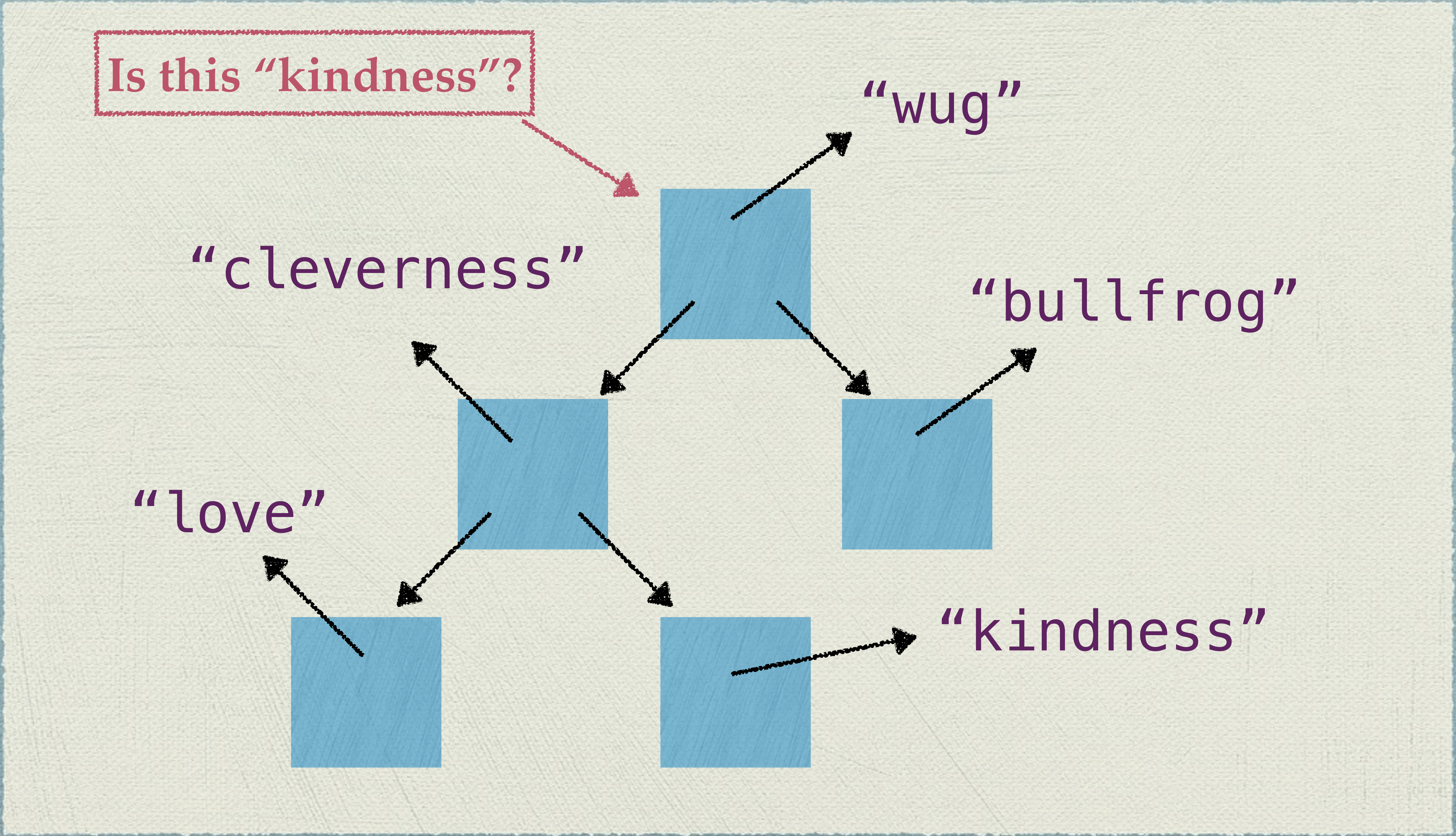
A tree… is a set?

- How do we check if it **contains** something?
- No option except to **search** the whole tree

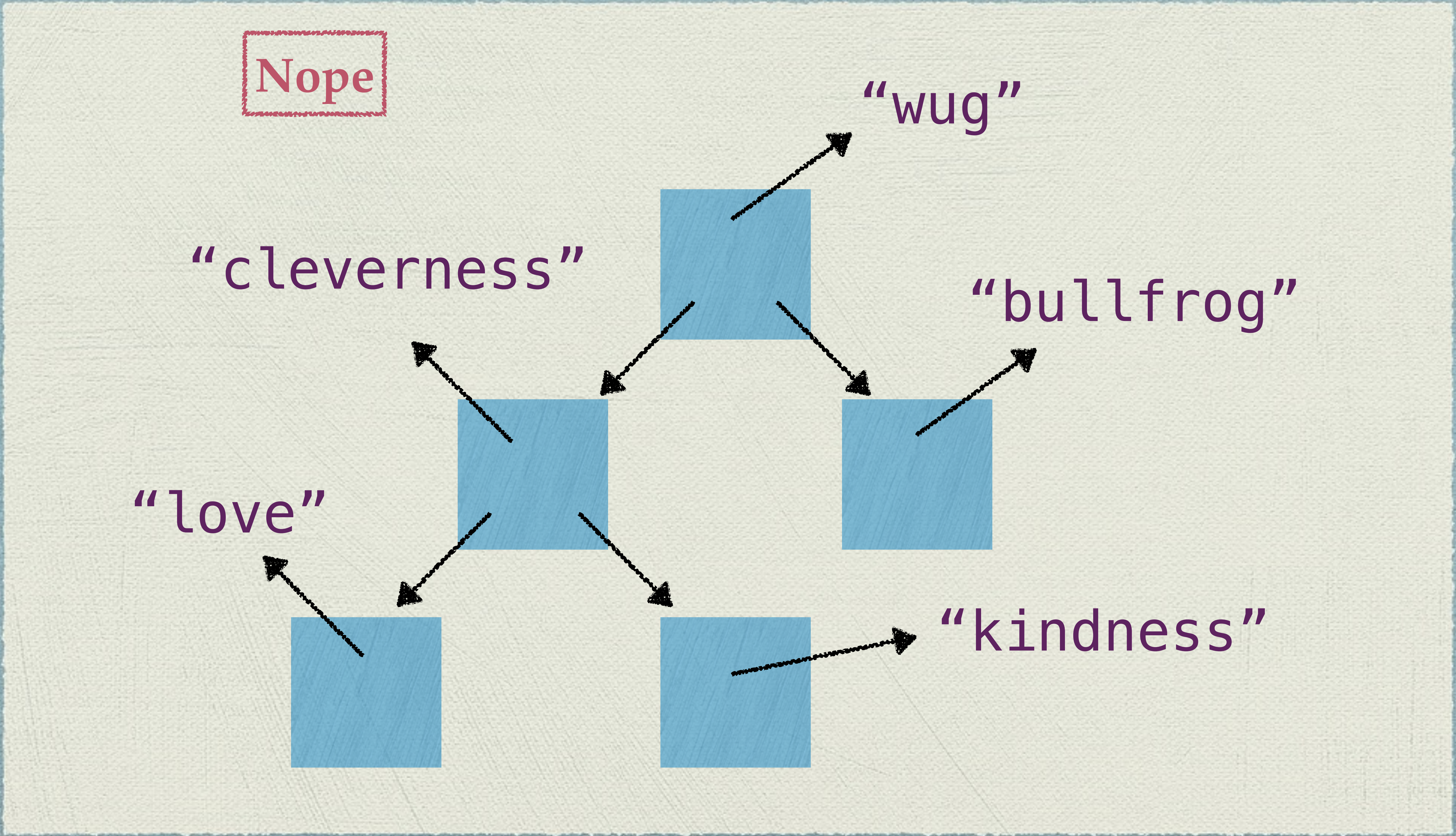
Does it contain "kindness"?



Does it contain “kindness”?

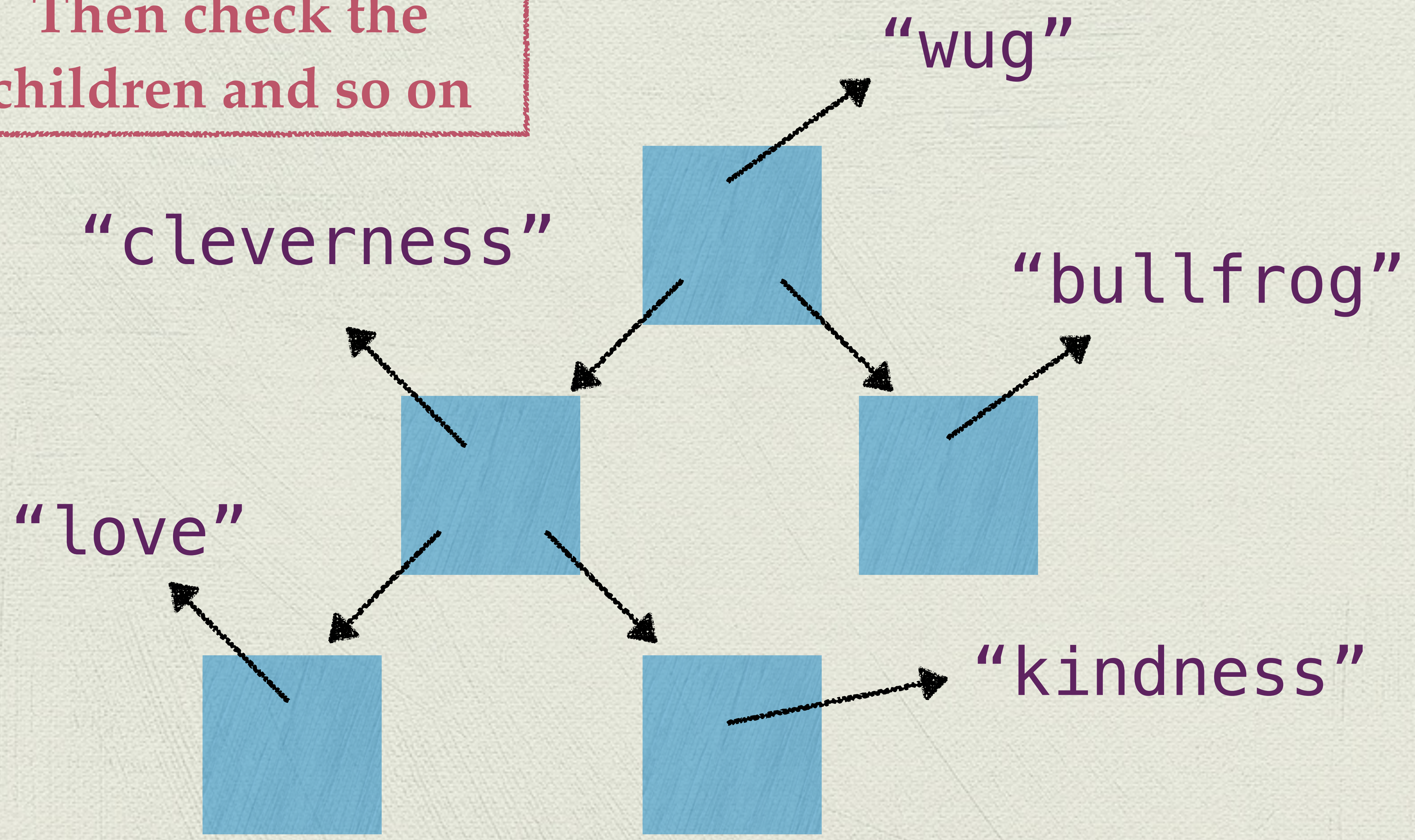


Does it contain “kindness”?

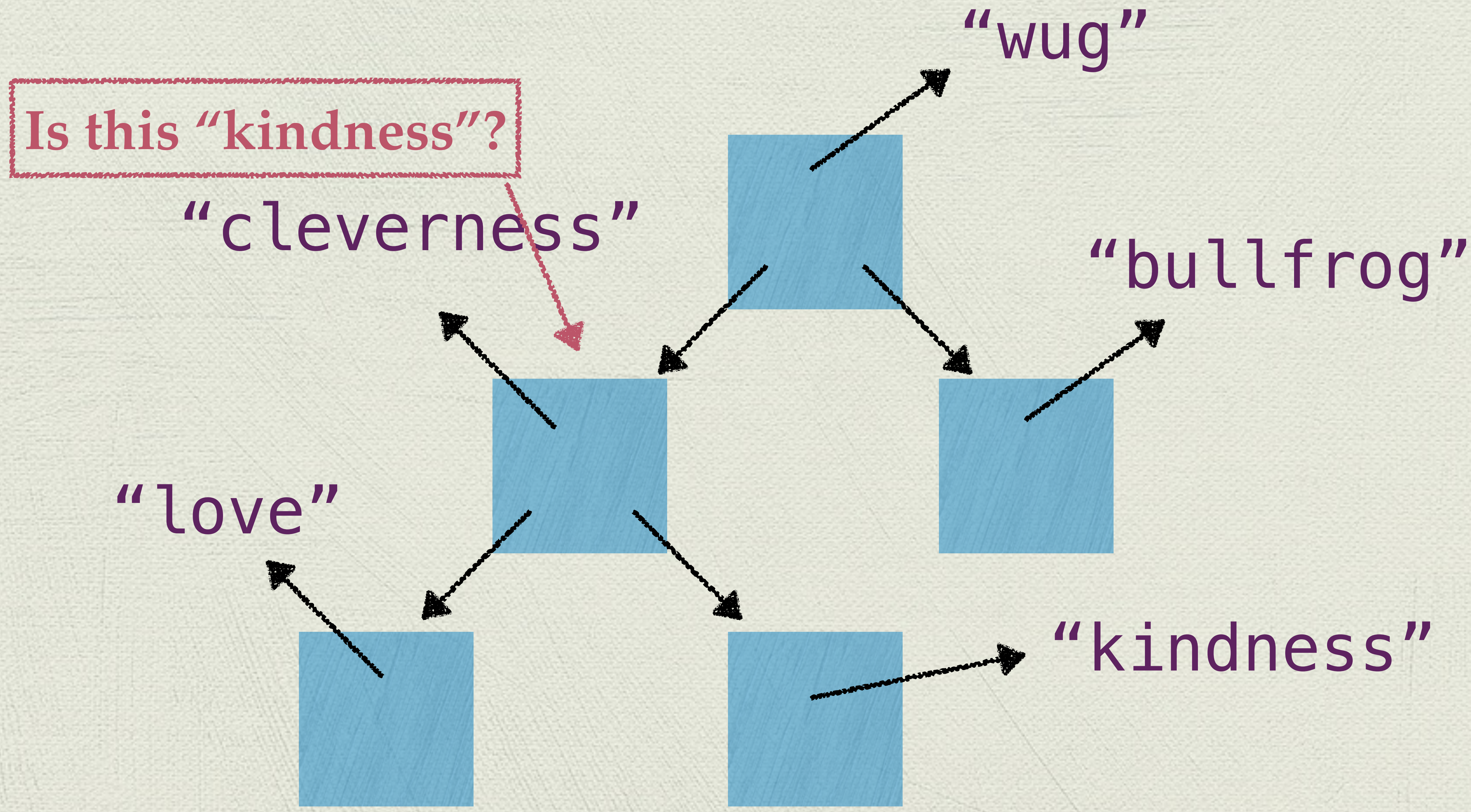


Does it contain “kindness”?

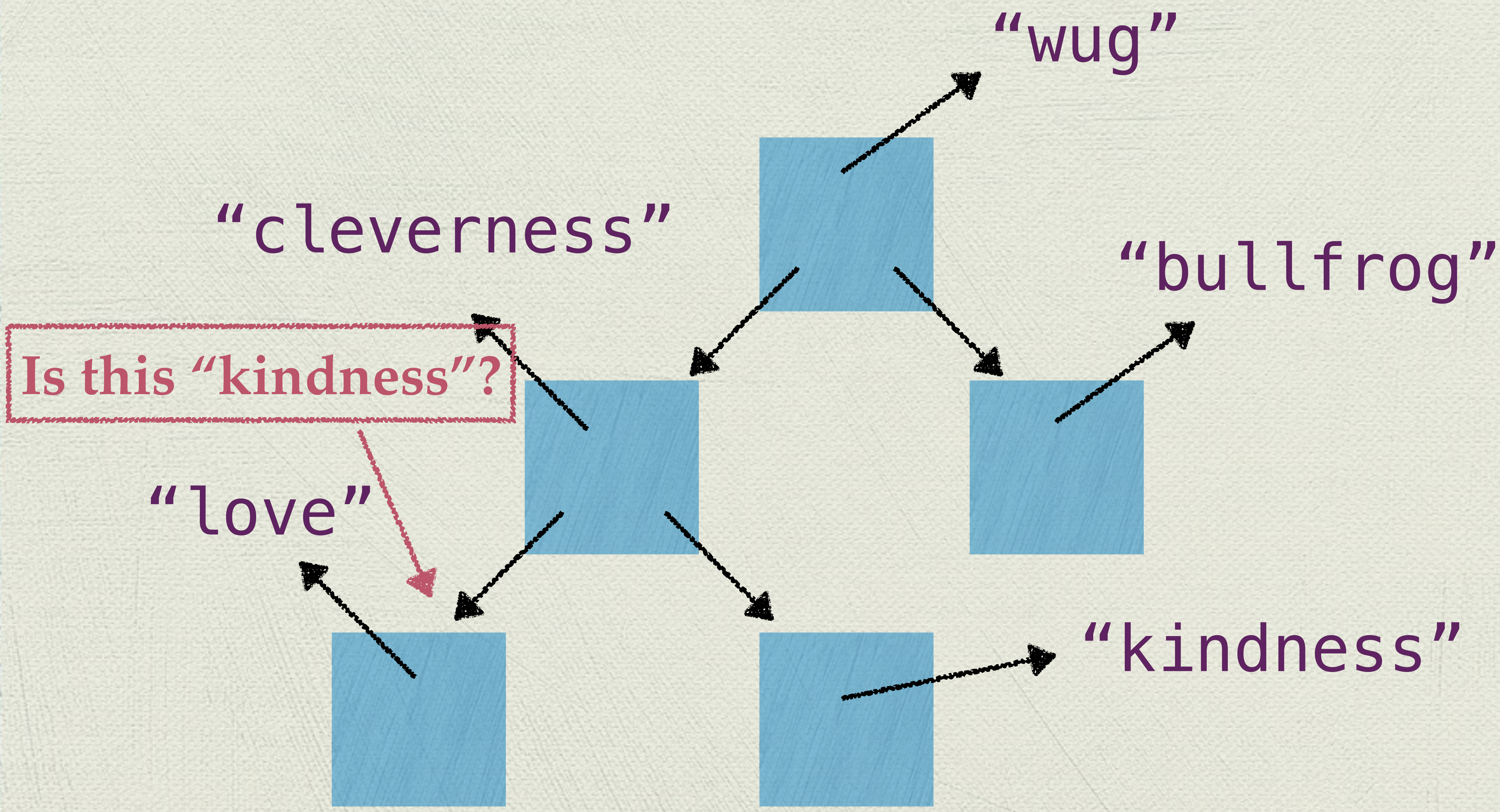
Then check the children and so on



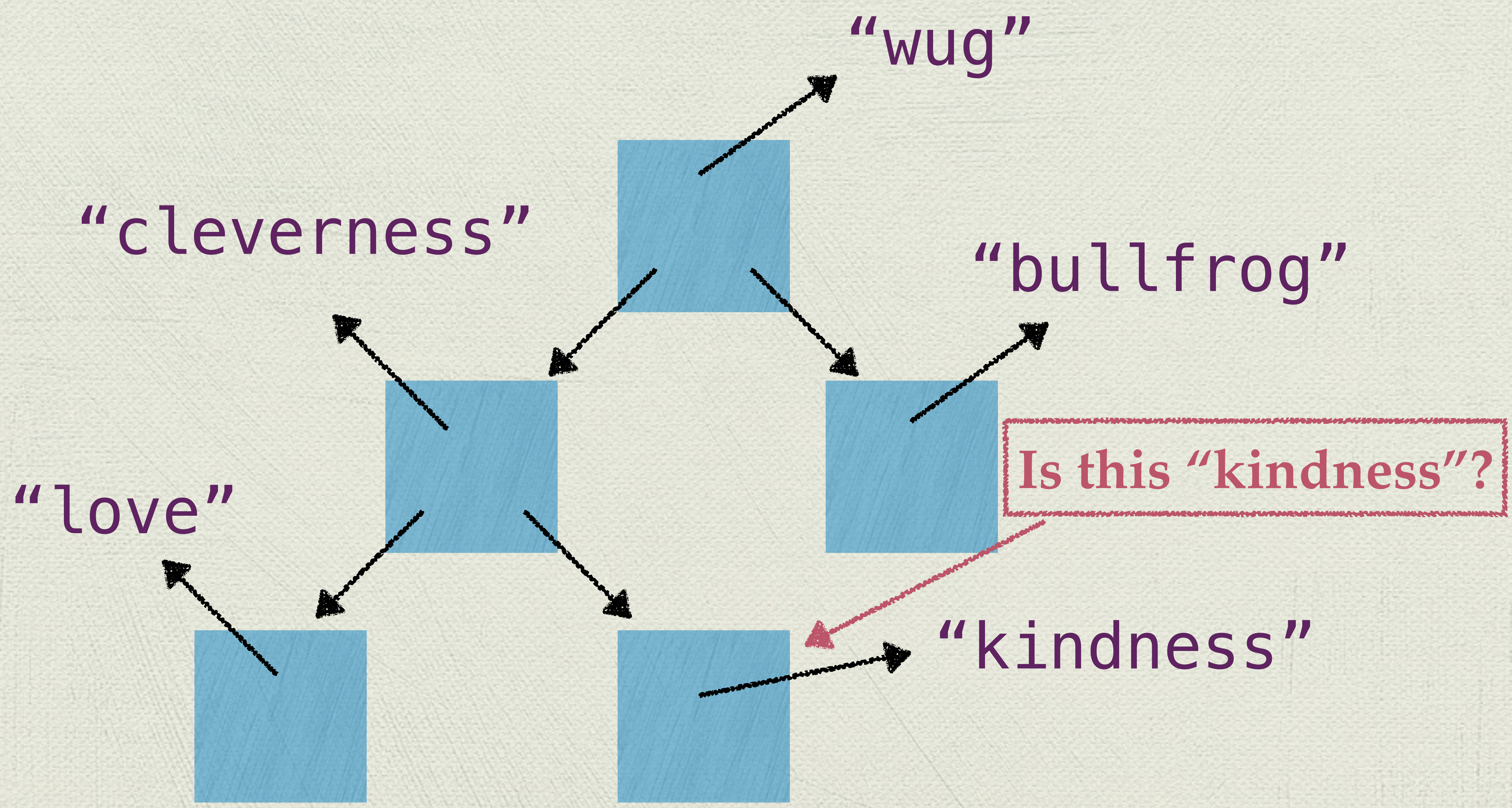
Does it contain "kindness"?



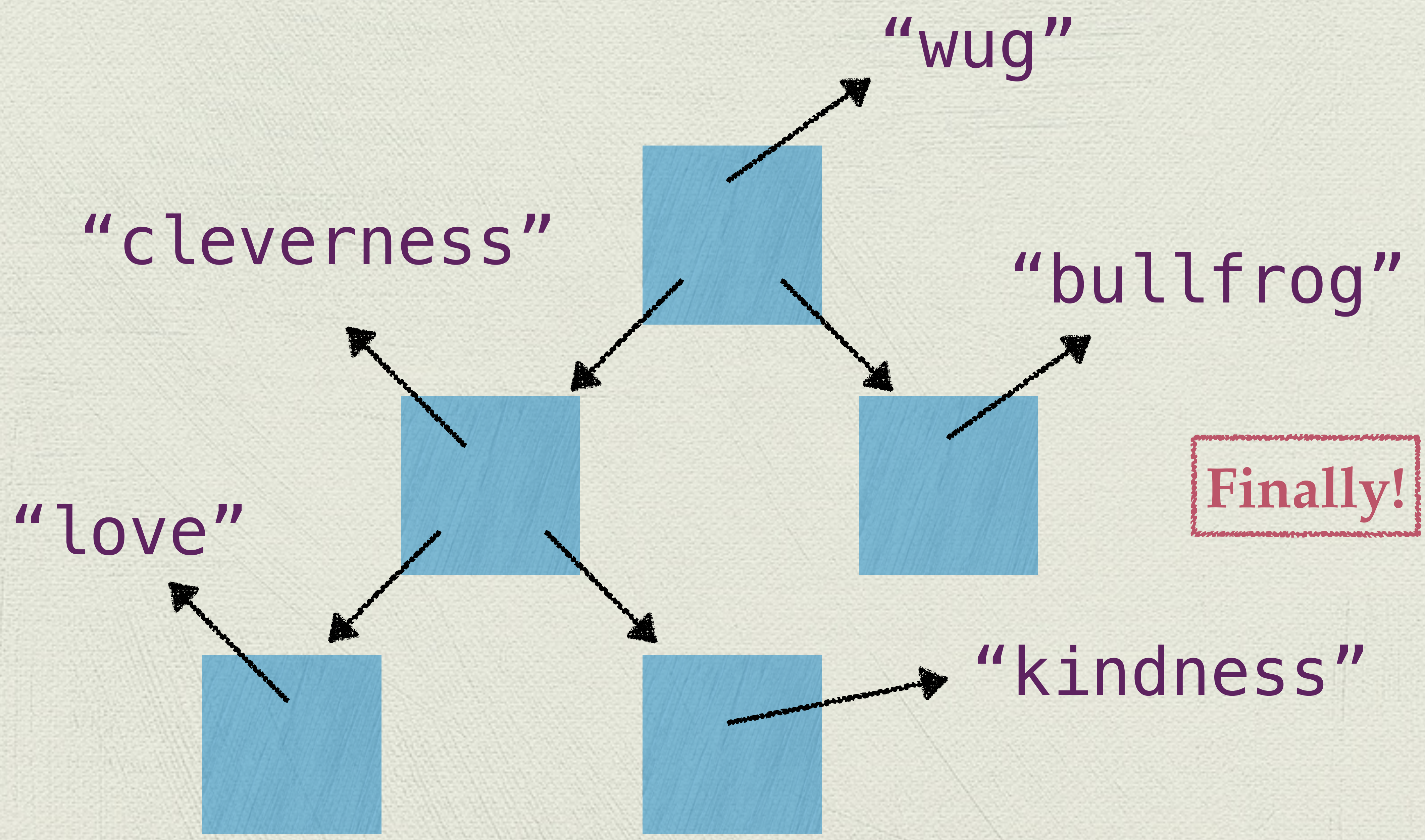
Does it contain “kindness”?



Does it contain "kindness"?



Does it contain “kindness”?



DFS

- We just performed a **depth-first traversal** of the tree
- Because we were looking for something, it's called **depth-first search**, or **DFS**
- In general, this is how we check if a tree contains something (alternatively: BFS)

Runtime?

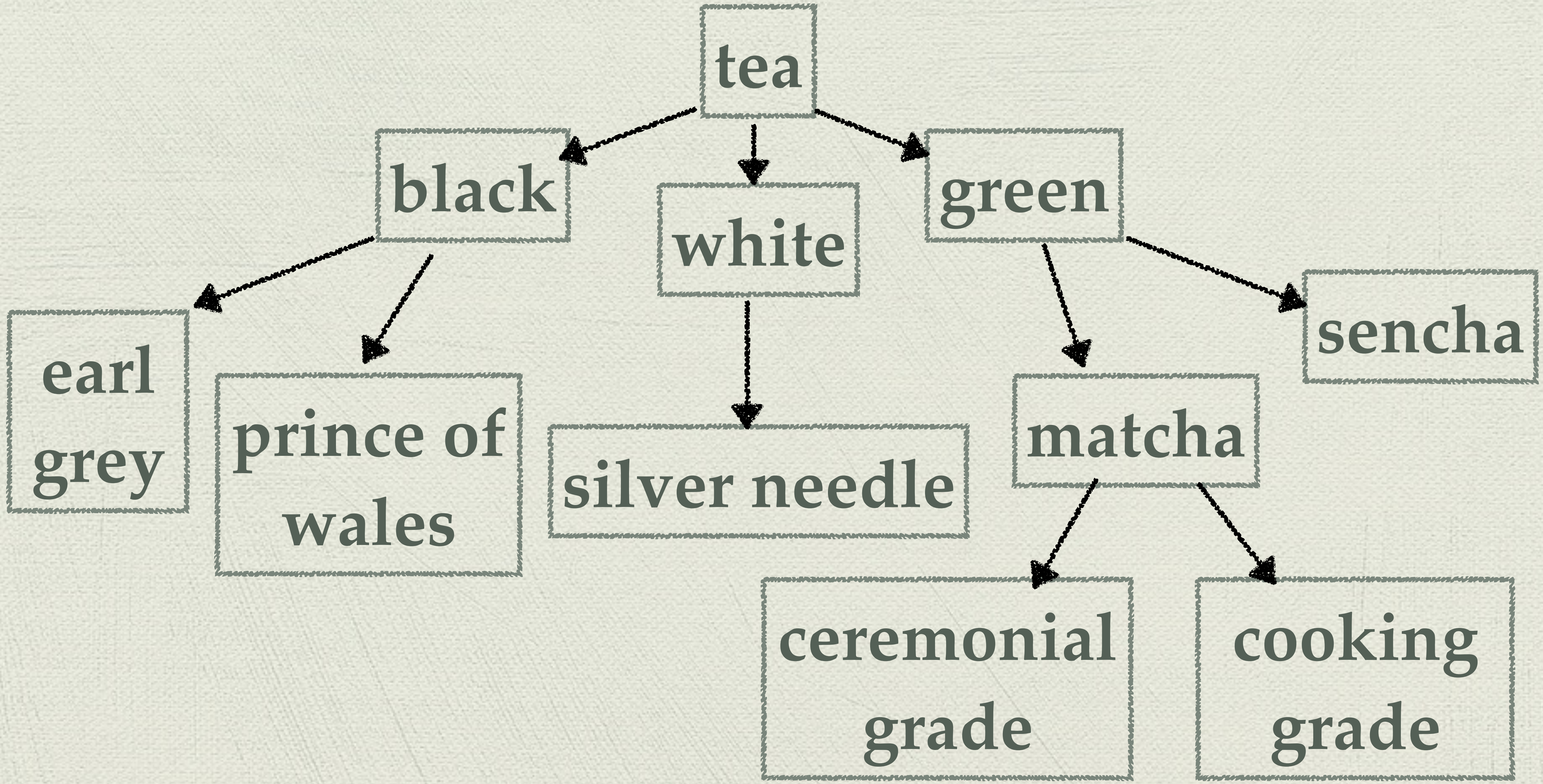
- **Check** if the set contains something
  - Use DFS: **O(*N*)** worst case, if there are N nodes in the tree. We just look at each node one-by-one

This is really bad

- A **tree** seems no better than using a **list** as a set
- And we already decided **hashing** was better than using a list

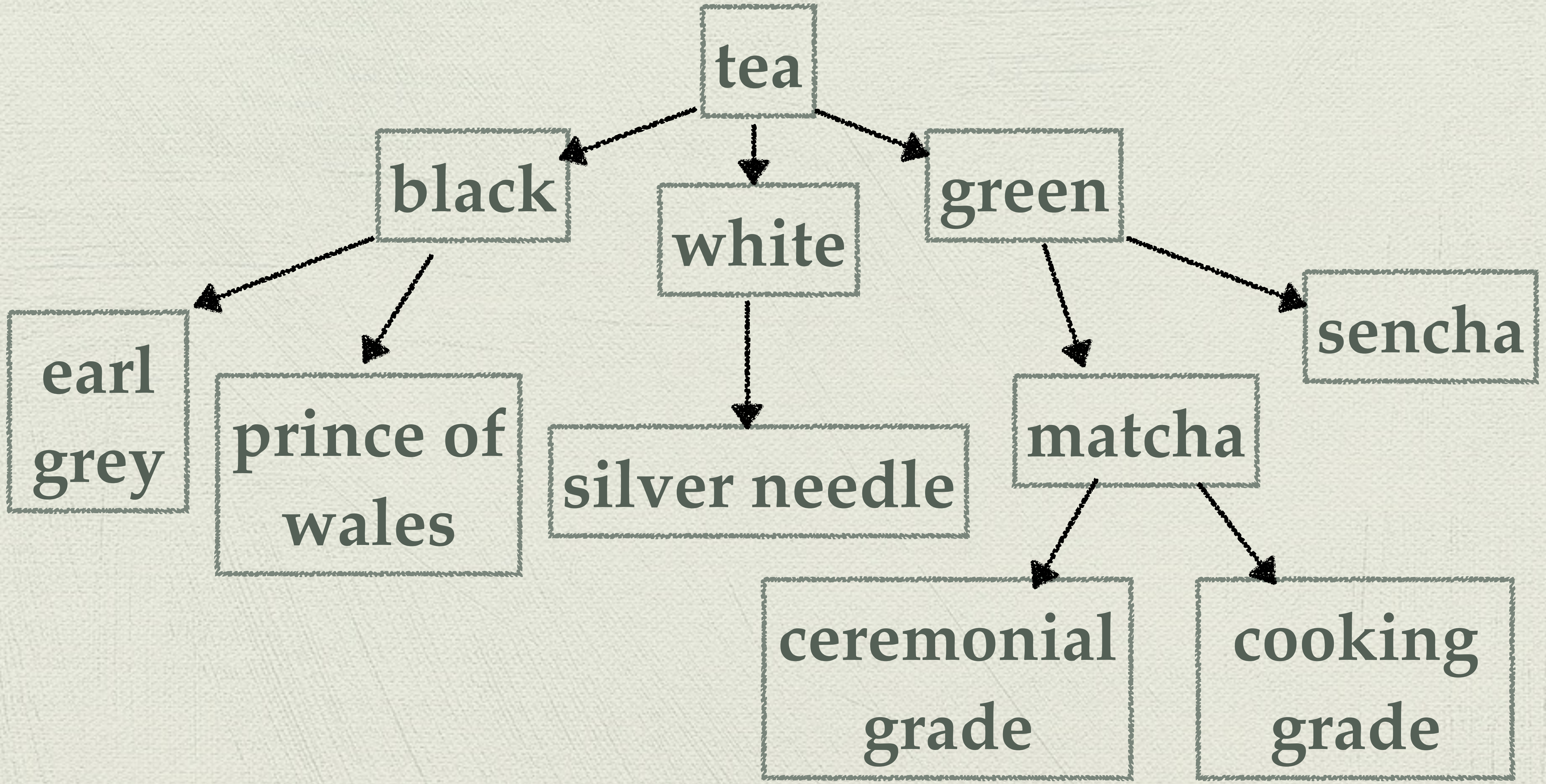
A hint that we could do better

- Here's a **set** of teas



A hint that we could do better

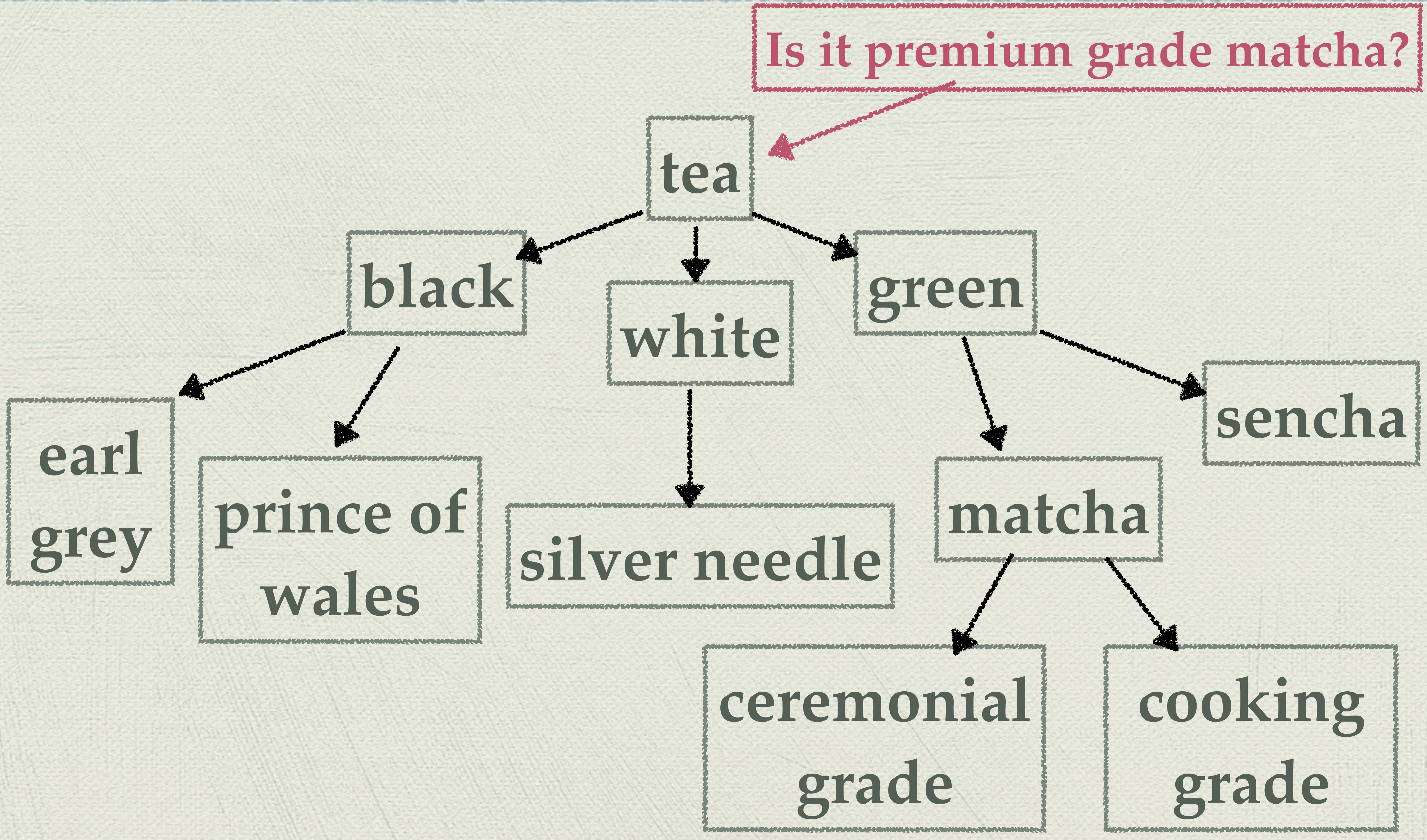
- Does it contain **premium grade matcha**?



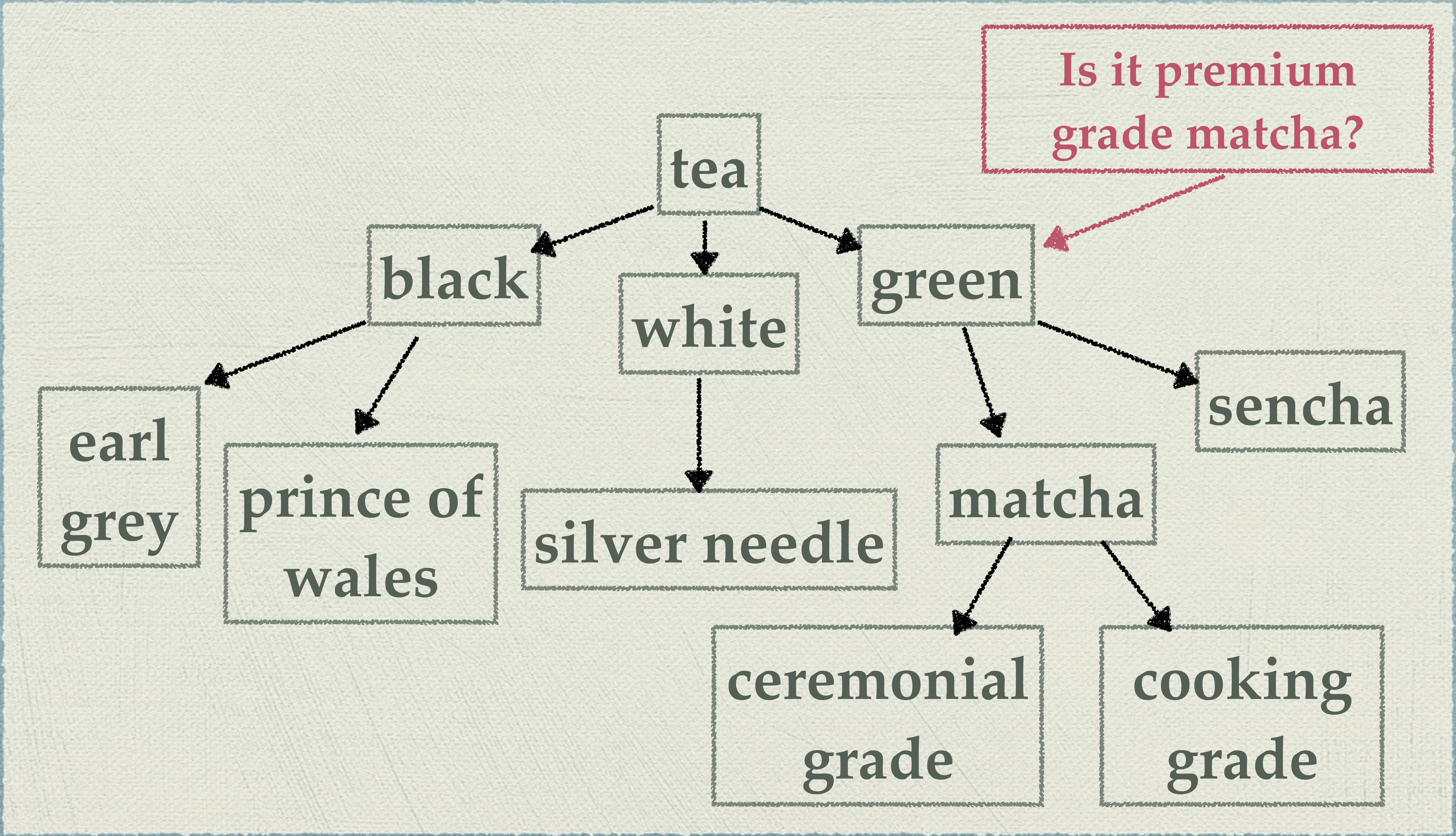
A hint that we could do better

- Does it contain **premium grade matcha**?
- I happen to know that premium grade matcha is a type of matcha, which is a type of green tea

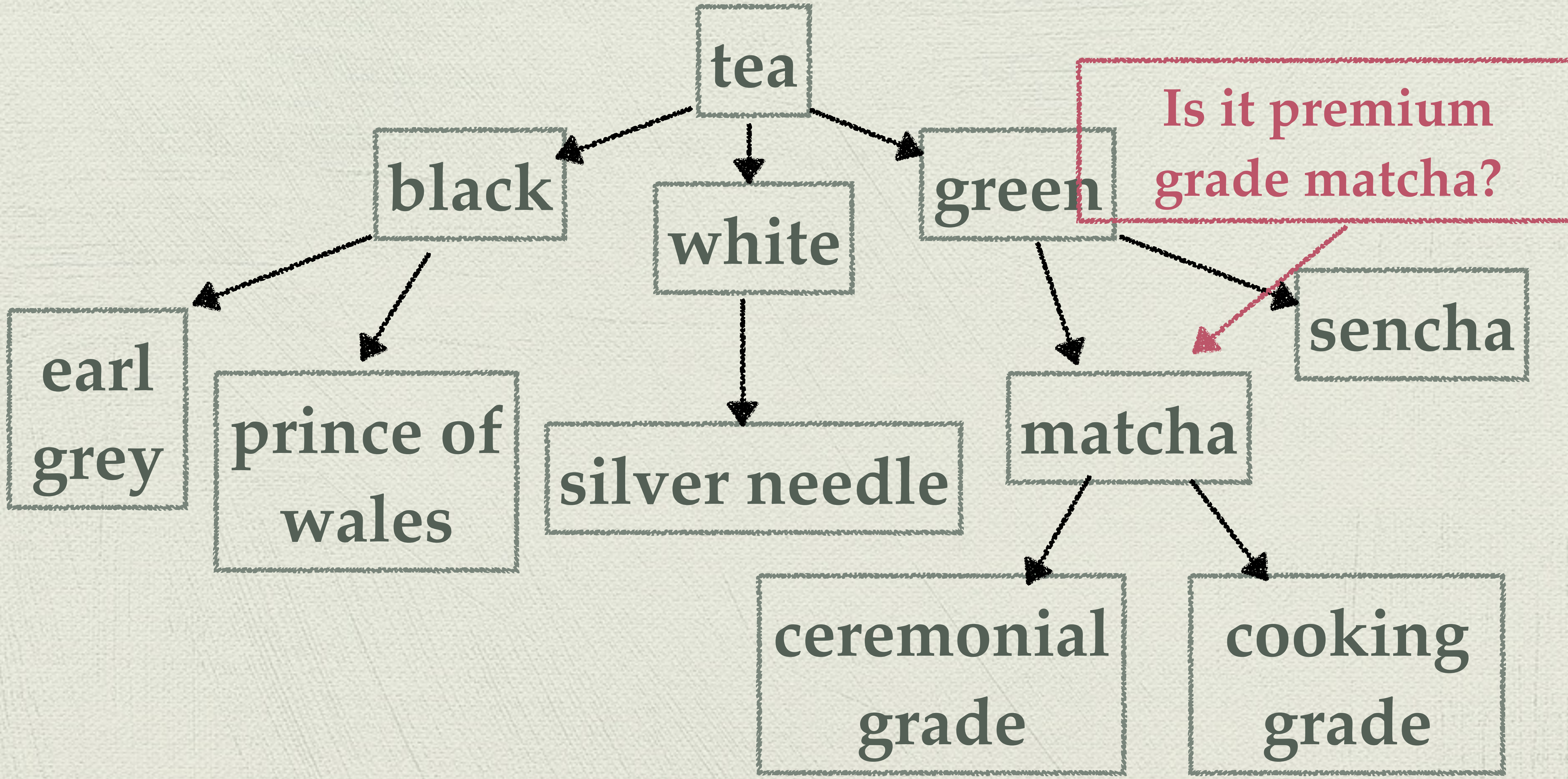
A hint that we could do better



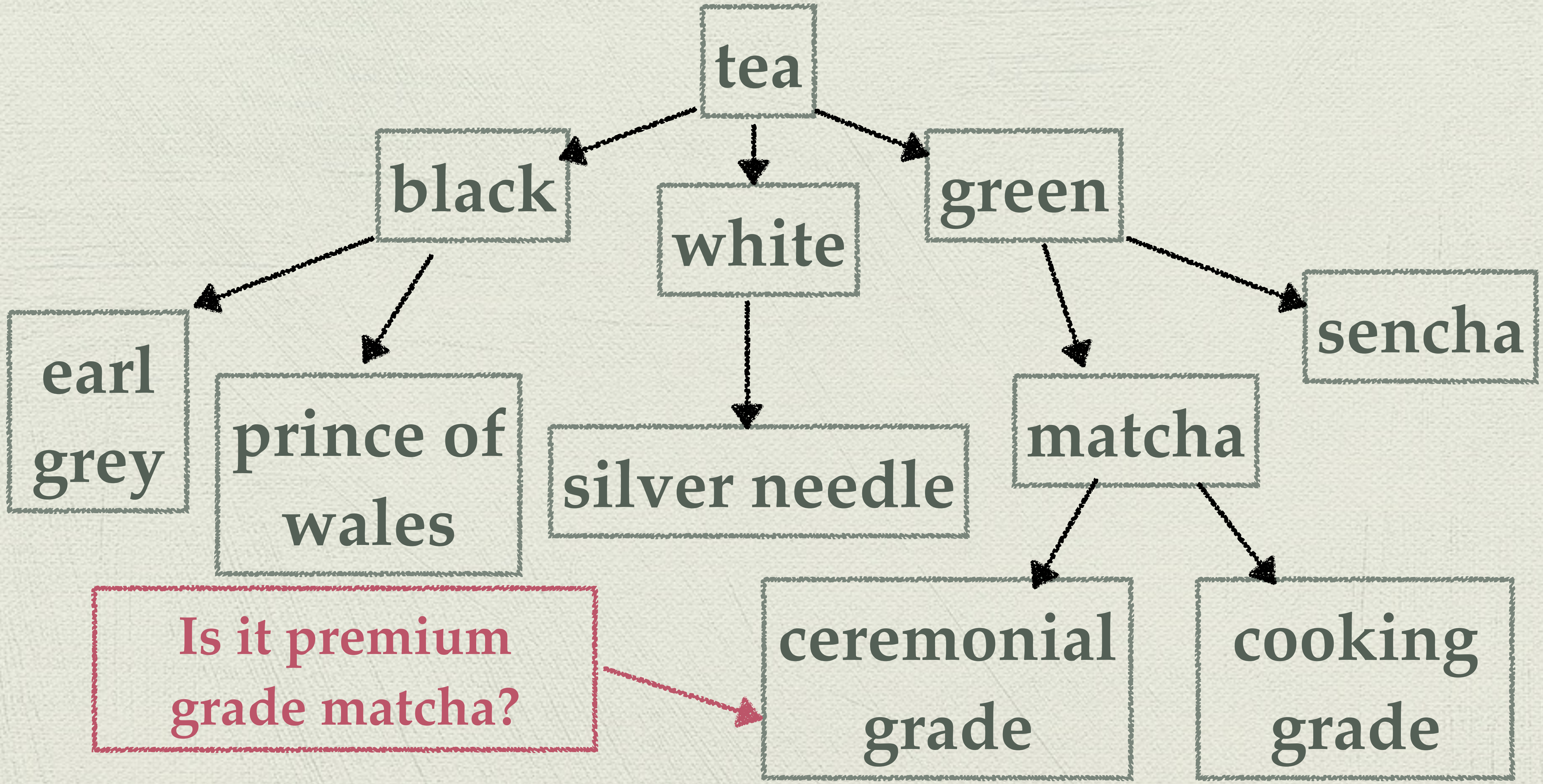
A hint that we could do better



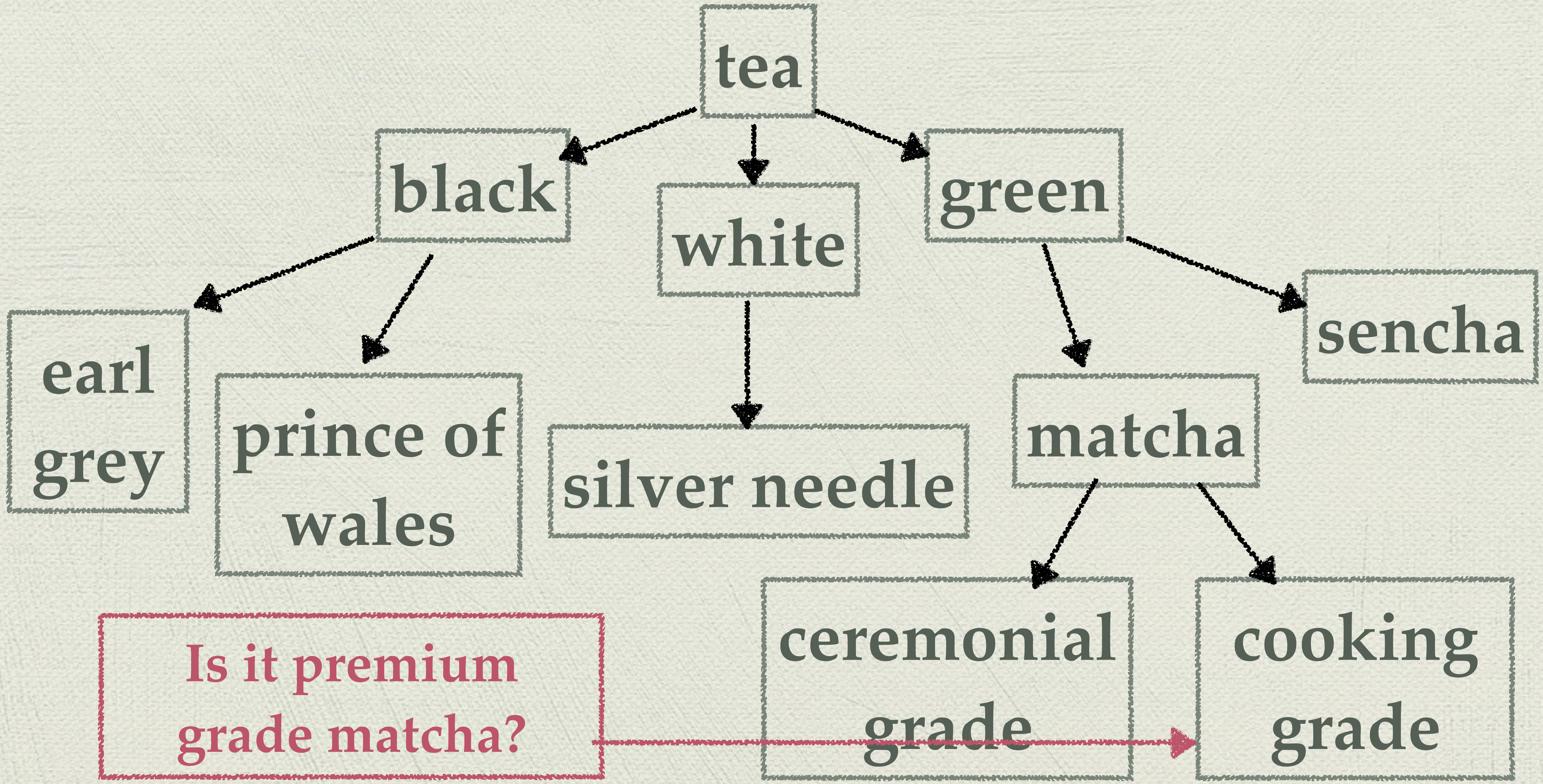
A hint that we could do better



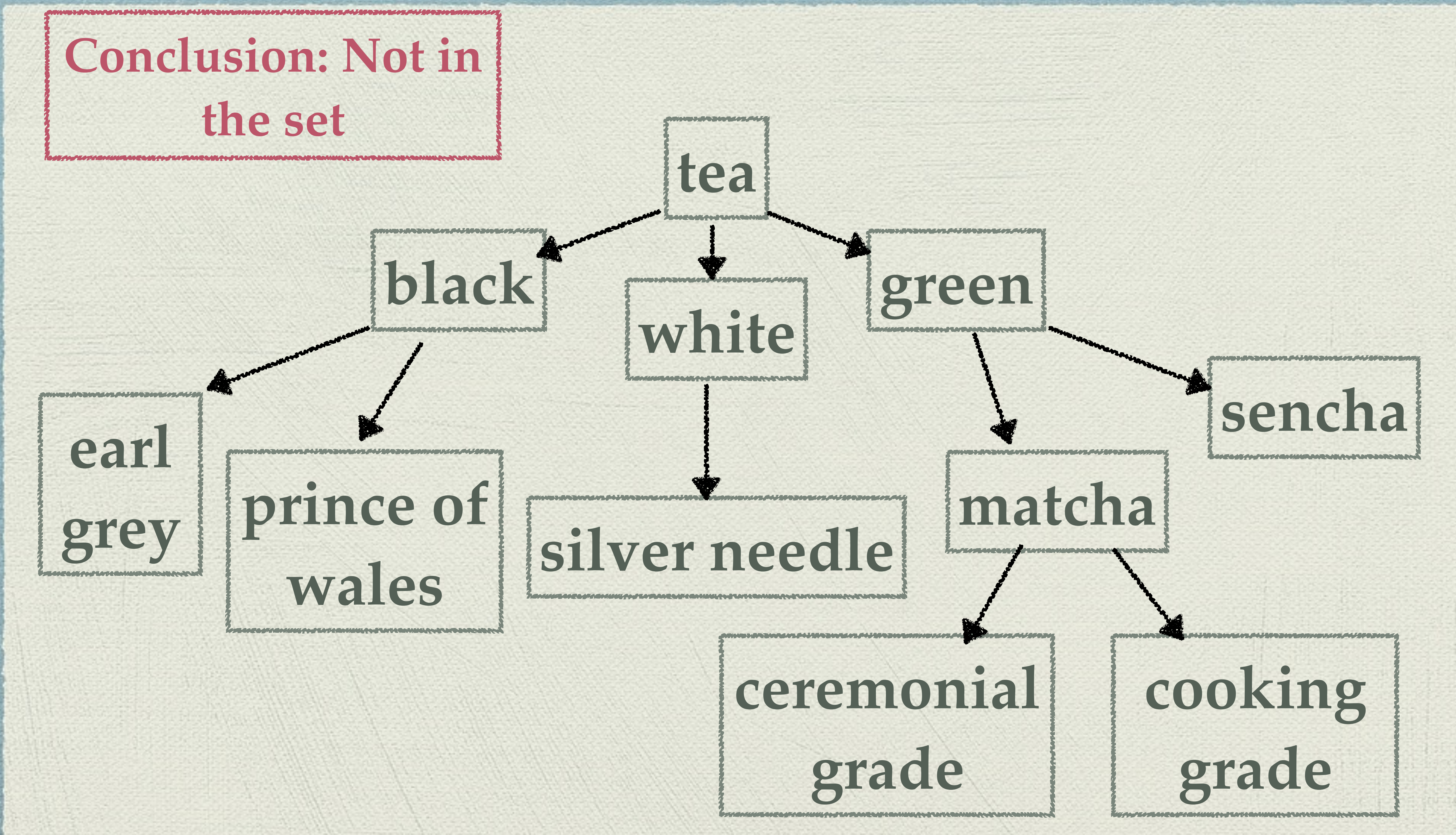
A hint that we could do better



A hint that we could do better



A hint that we could do better



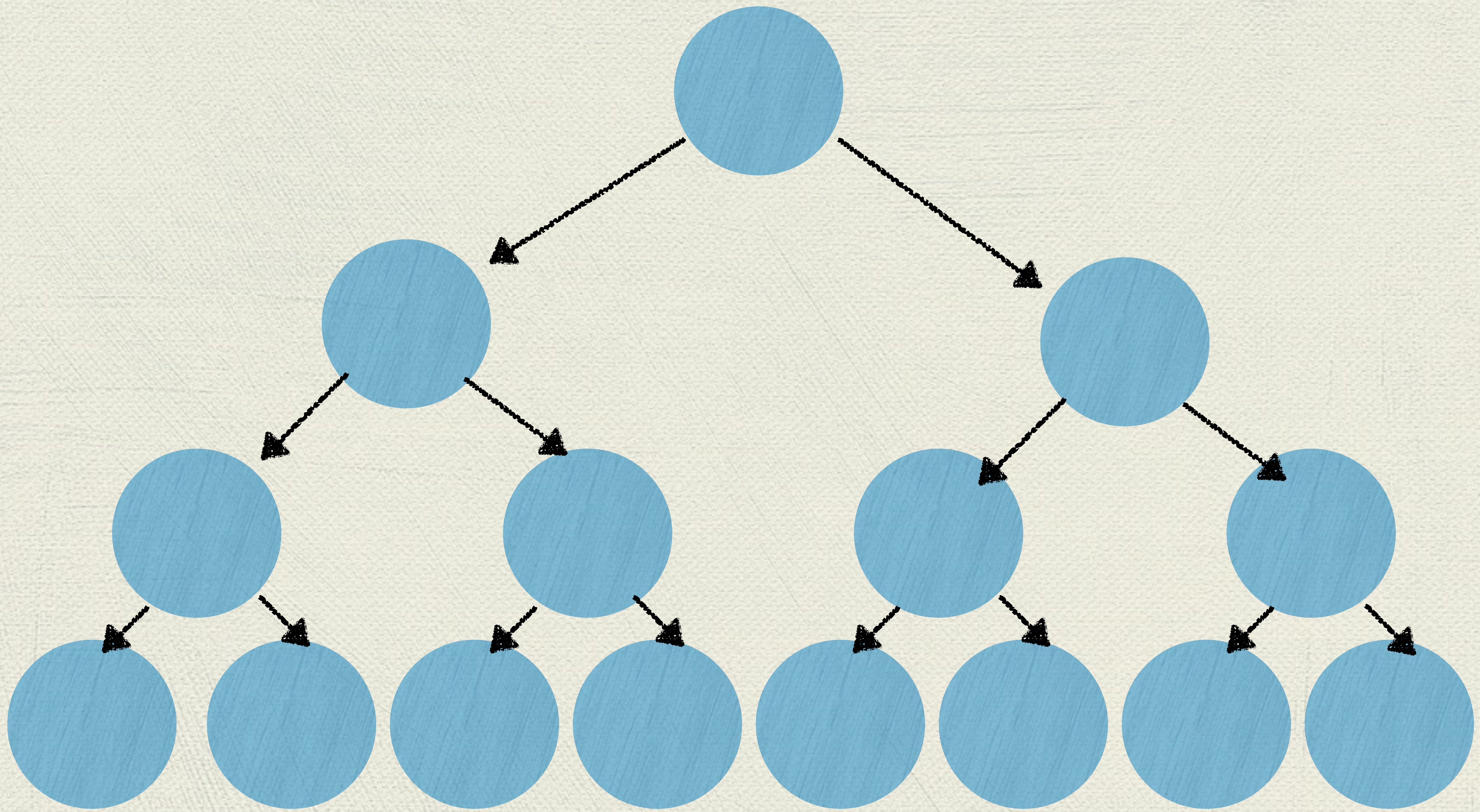
The point

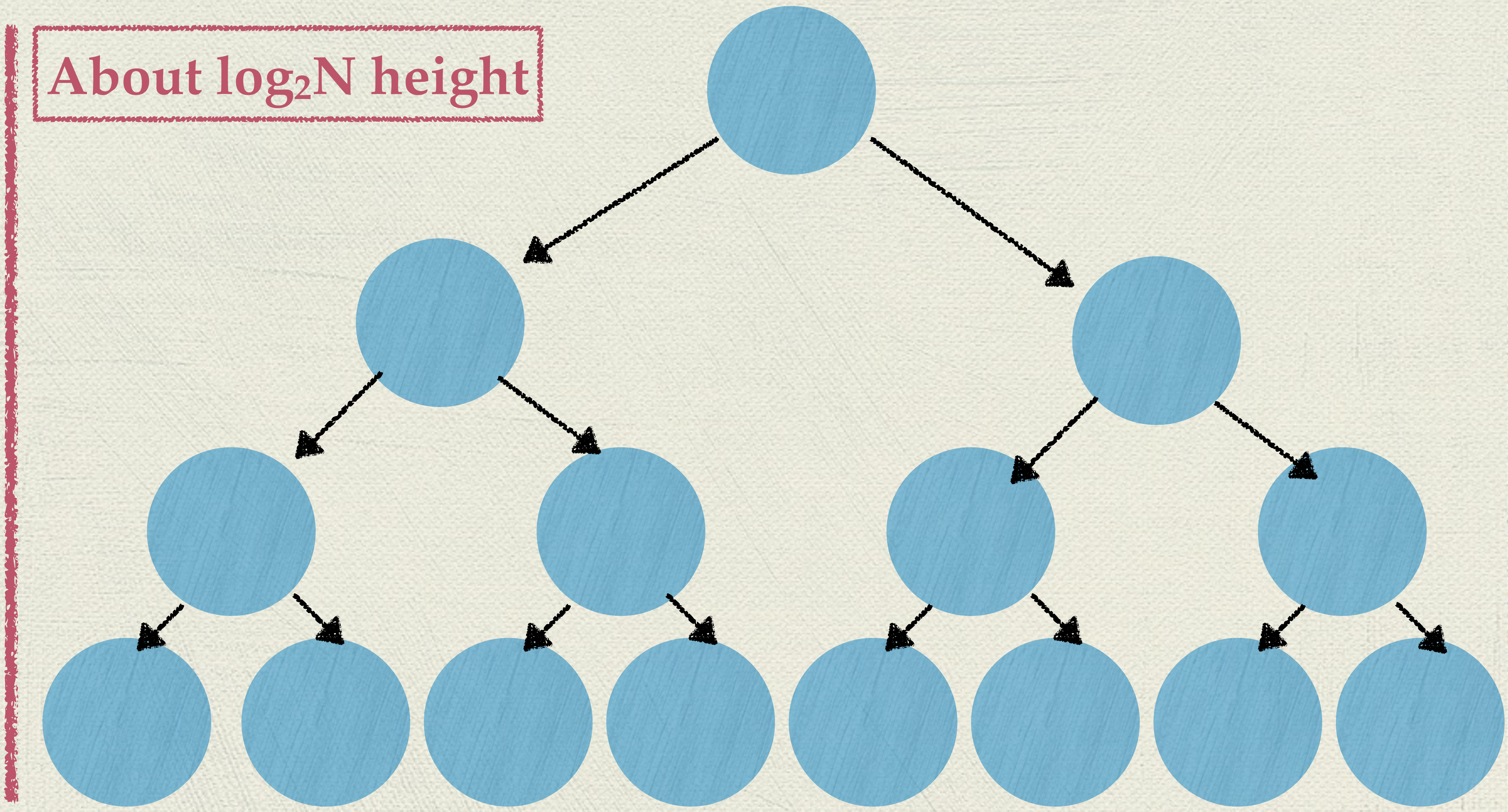
- We knew we wouldn't have to check anywhere down the black branch or the white branch
- We can take advantage of the **organizational structure** of the tree to improve search time

Runtime now?

- We have to check every node from root to a leaf, but *not every node in the tree*
- How many nodes are there from root to leaf?
- In other words, what is the height of tree?

Say there are N nodes total. How many nodes does it take to get to the bottom?




About $\log_2 N$ height

What is log?

$$log_B(N)$$

- is the number of times you have to divide N by B before you get 1

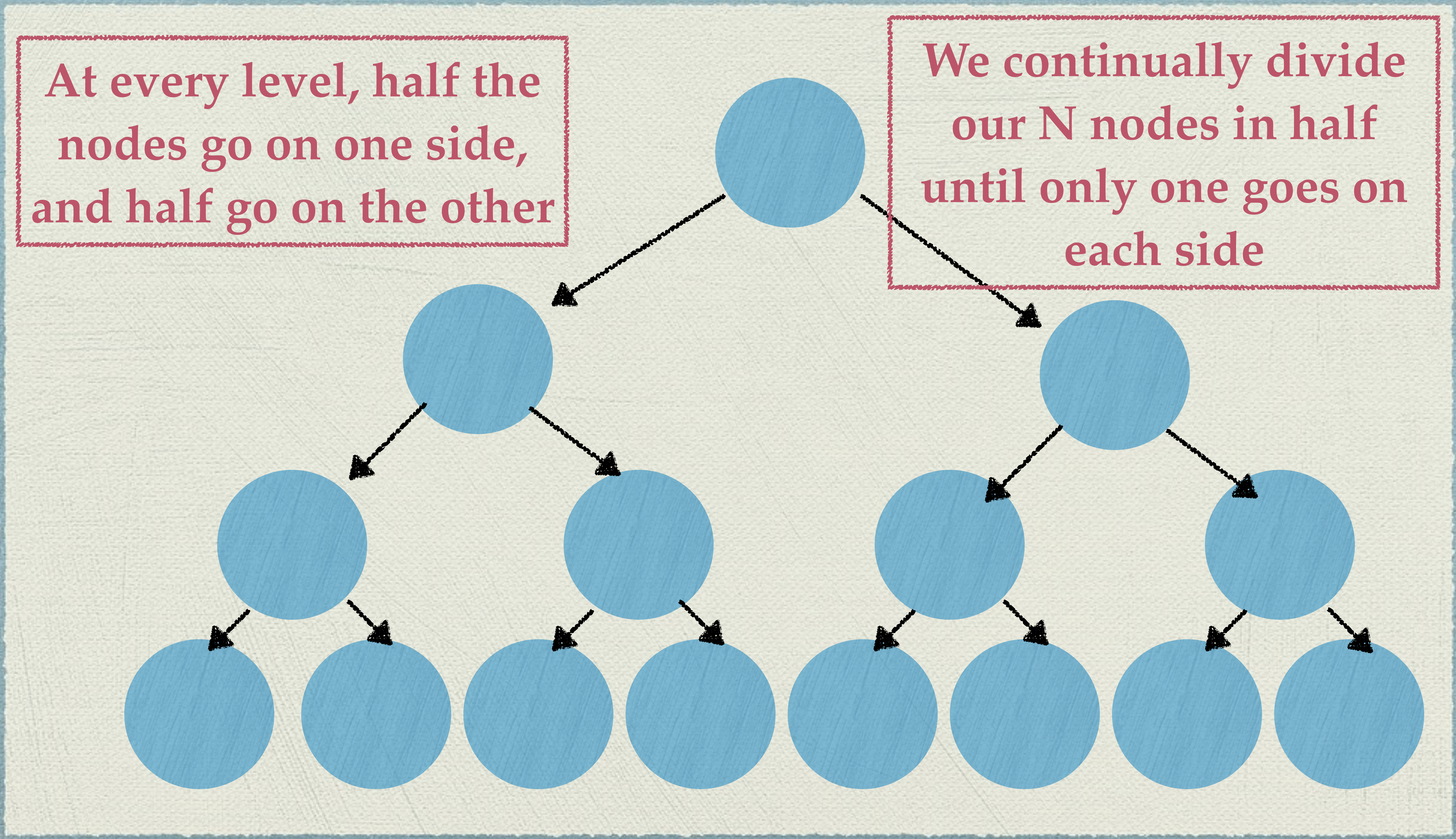
Example

$$log_2(16) = 4$$ because

$$16/2 = 8$$
$$8/2 = 4$$
$$4/2 = 2$$
$$2/2 = 1$$

} 4 steps

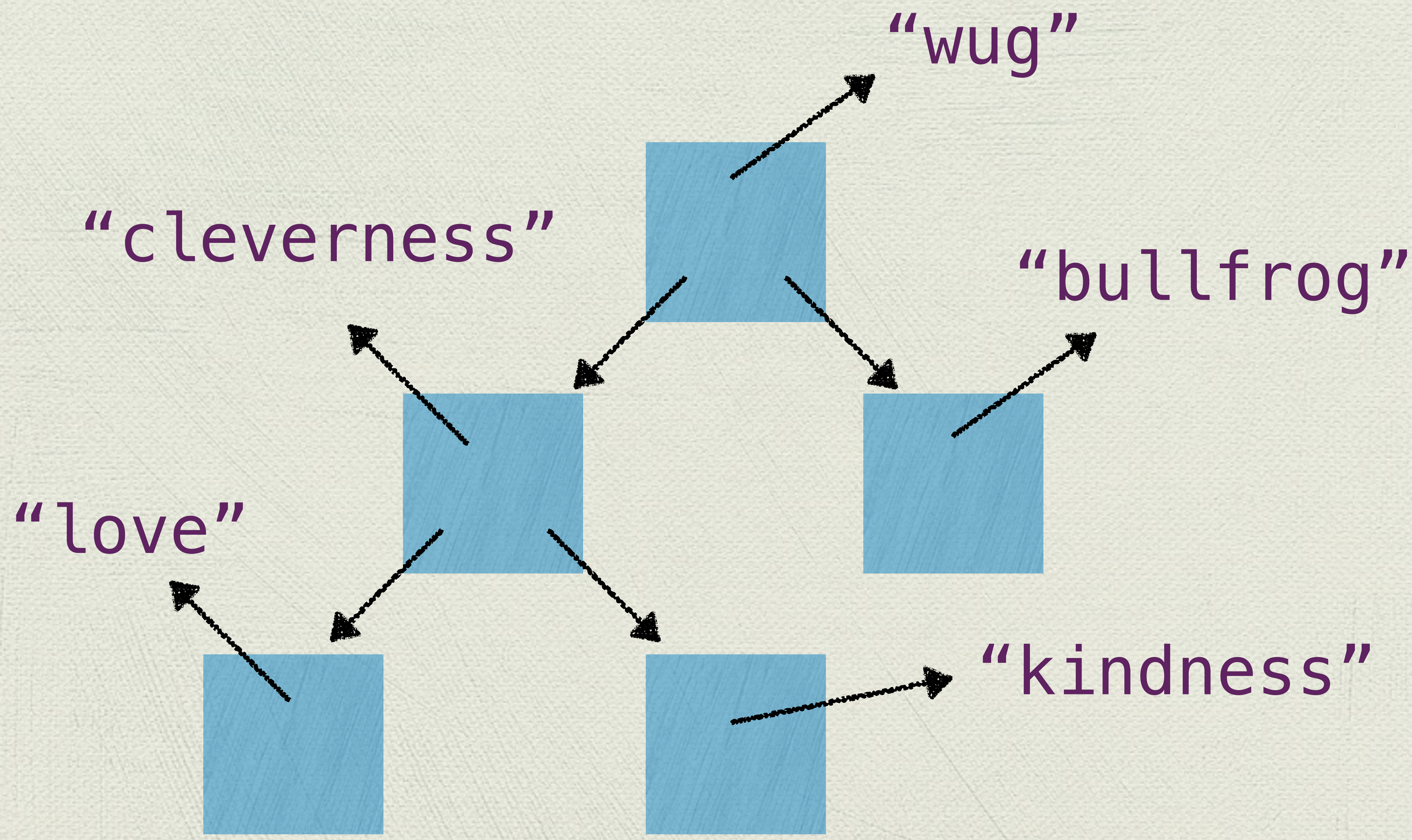
What does this have to do with trees?



Conclusion

- **Contains** in our tree set runs in **O(logN)** time, which is significantly less than **O(N)**

Why didn't it work here?



Why didn't it work before?

- Because the tree was completely **unorganized**. They were just random Strings placed in there

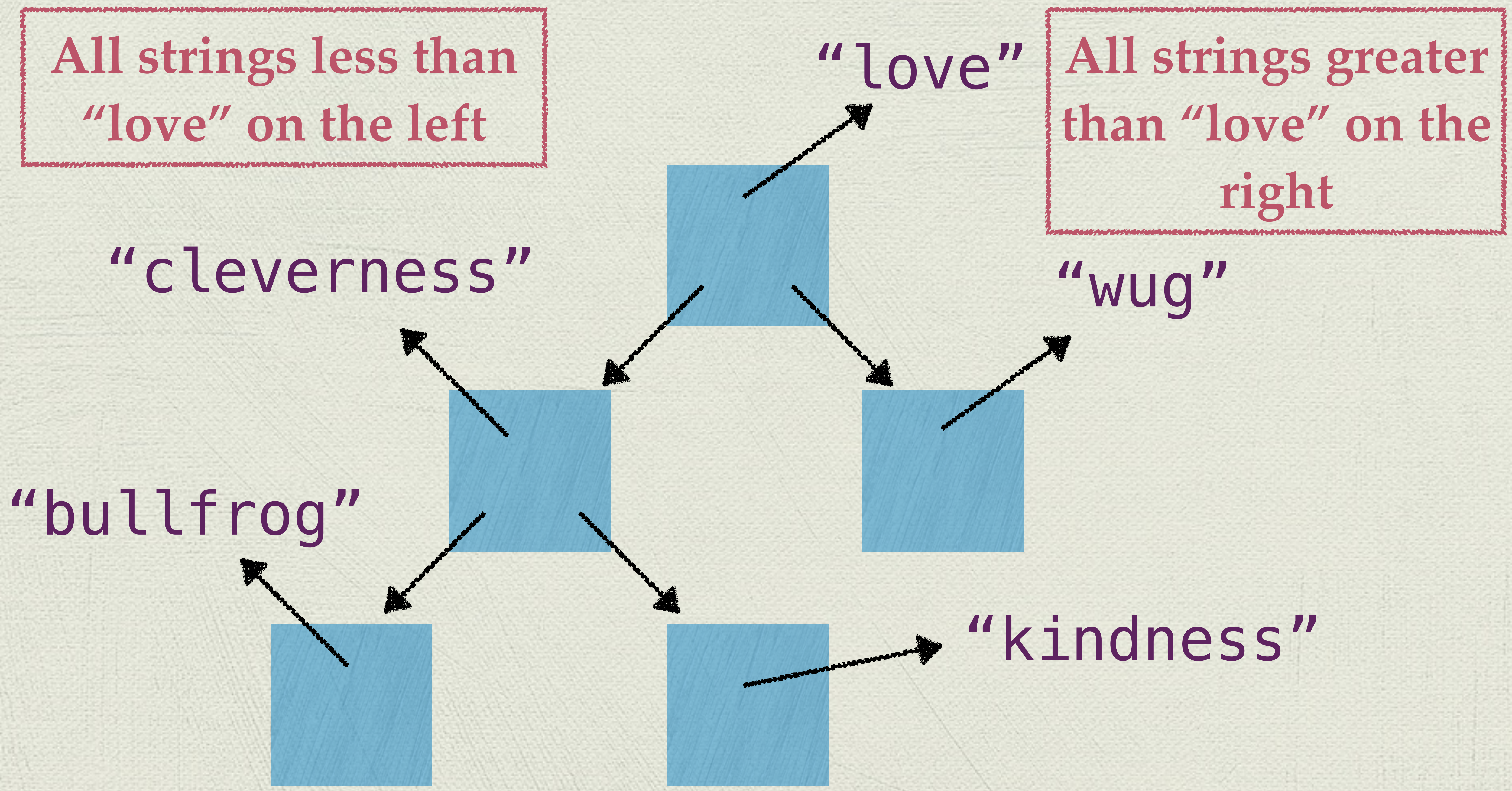
An idea

- We can organize arbitrary strings **alphabetically**
- This strategy will allow us **O(logN) contains** time on a set of any strings

The binary search tree

- A binary search tree (**BST**) is a special kind of tree that organizes strings alphabetically, or integers by size, etc.

BST version



All strings less than "love" on the left
"love"
All strings greater than "love" on the right
"cleverness"
"wug"
"bullfrog"
"kindness"

The binary search tree

- A BST is a tree with one more **special rule** (invariant)
  - Consider a TreeNode **t**. All nodes in the left subtree of **t** are less than t. All nodes in the right subtree of **t** are greater than **t**.
  - This rule holds recursively for all nodes in the tree

Congratulations

- We built a set with **O(log N)** contains time

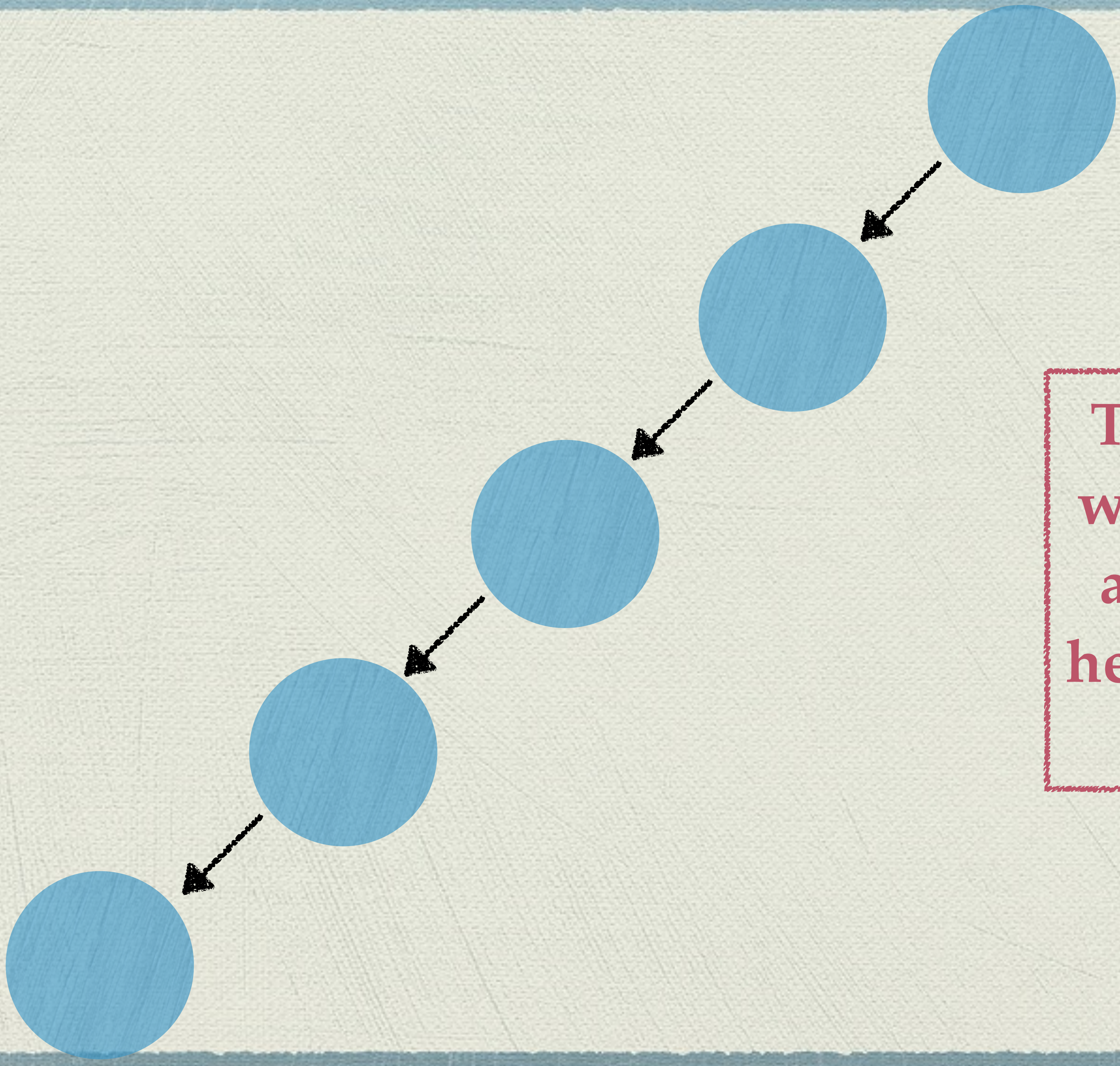
Congratulations

- We built a set with **O(log N)** contains time **NOT**

Actually

- We built a set with **O(H)** contains time, where **H** is the height of the tree
- Normally H is logN, as we showed, but…

What about this tree?



Technically a tree, but we don't divide in half at each step. Now the height of the tree is just N, not logN

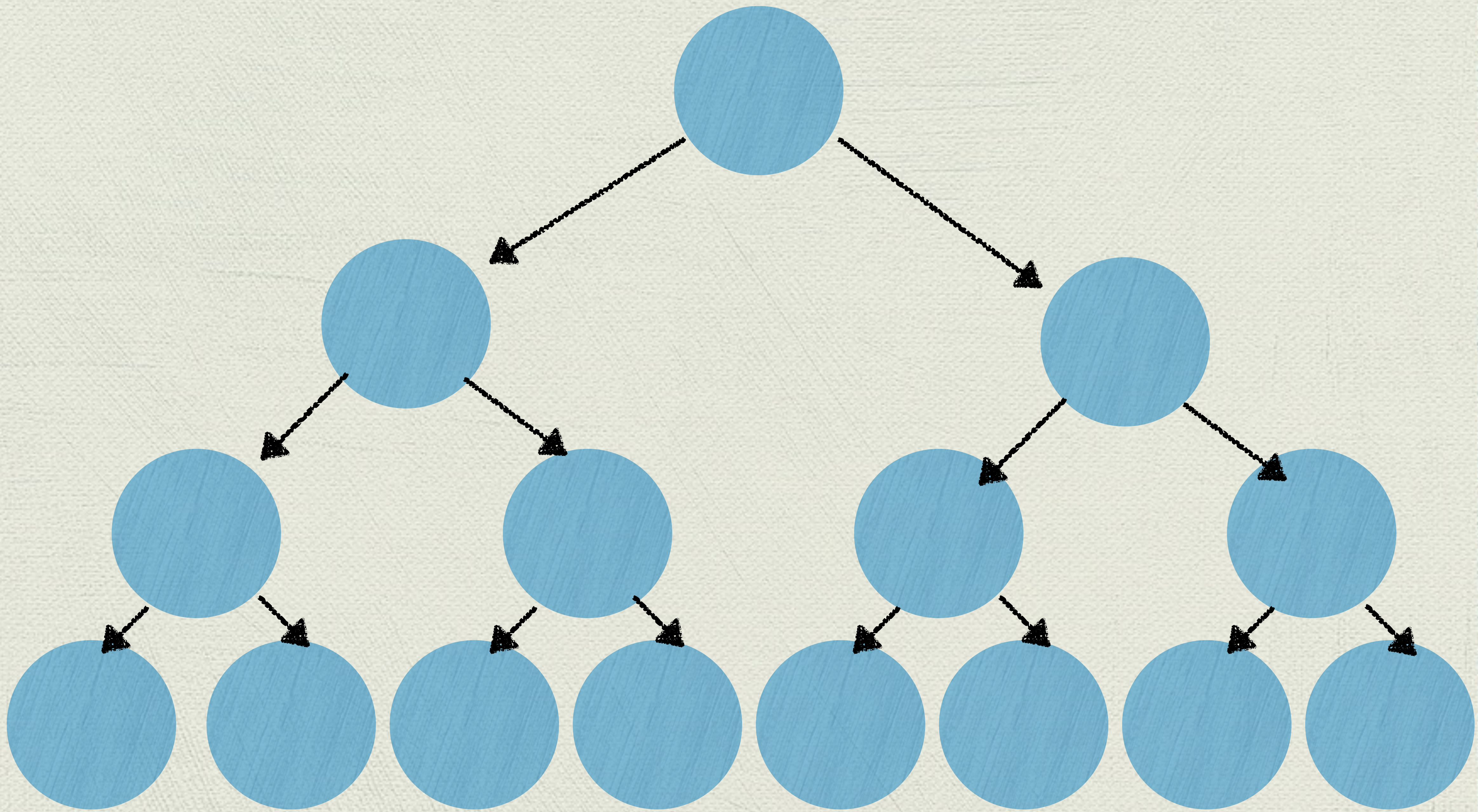
Problem

- The BST only has good **contains** time if the tree is relatively **balanced**, or close to

Balance

- We'll develop three notions of balance
  - Completely balanced
  - Maximally balanced
  - Almost balanced
- These are three technical terms

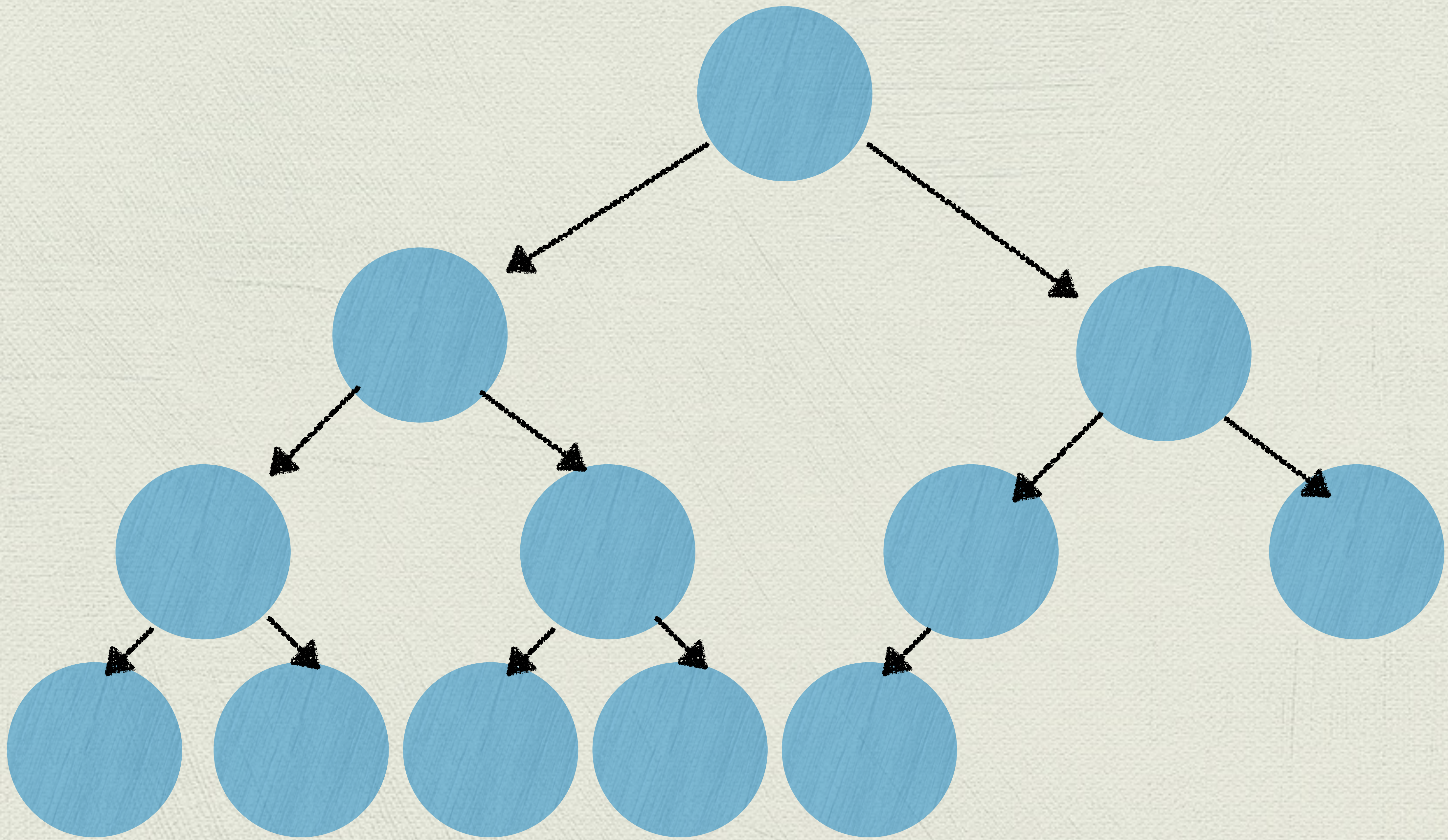
Completely balanced



Maximally balanced

- Every row is filled, except possibly the last, which is filled left-to-right
- This is equivalent to the condition that the array tree has no holes in it

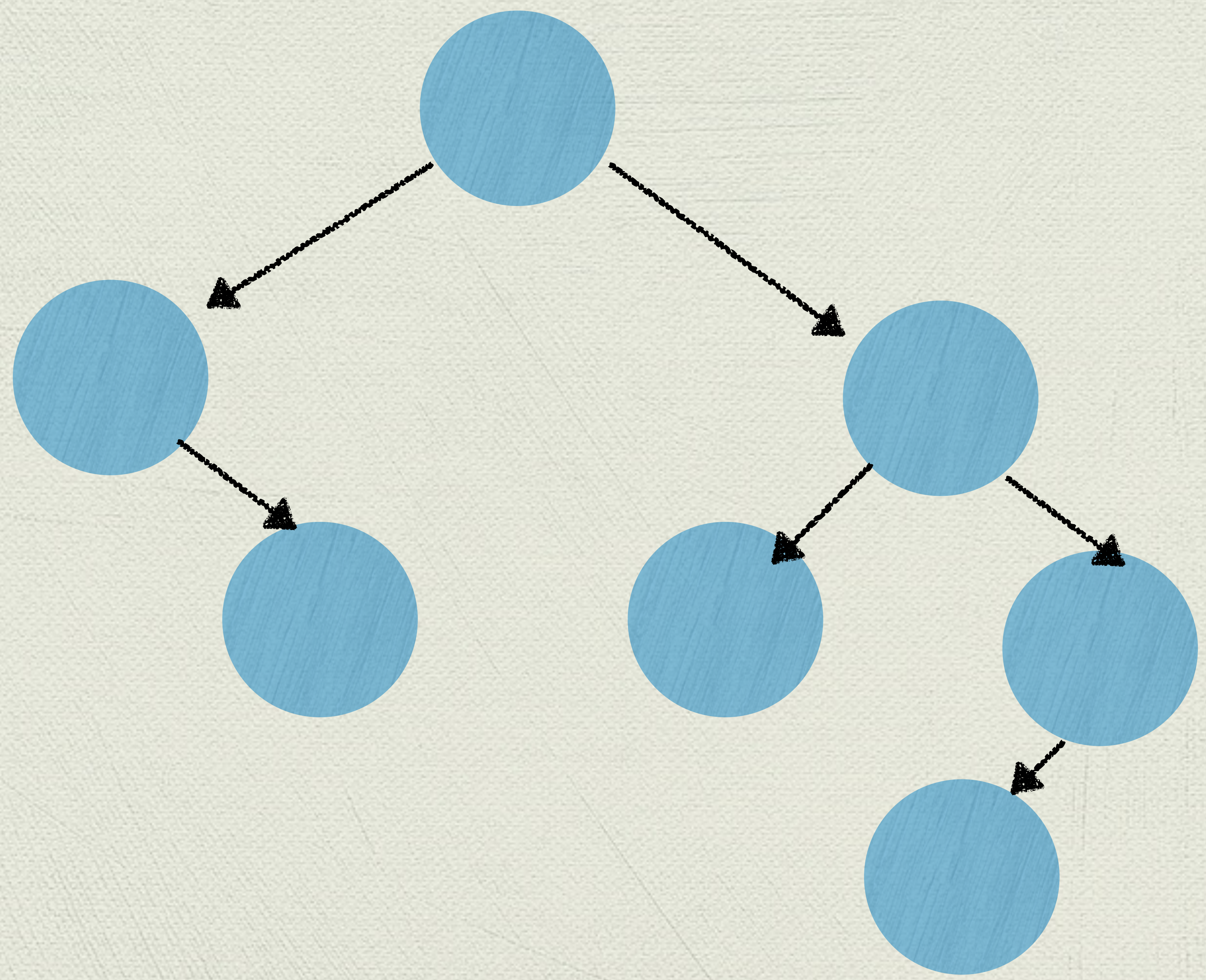
Maximally balanced



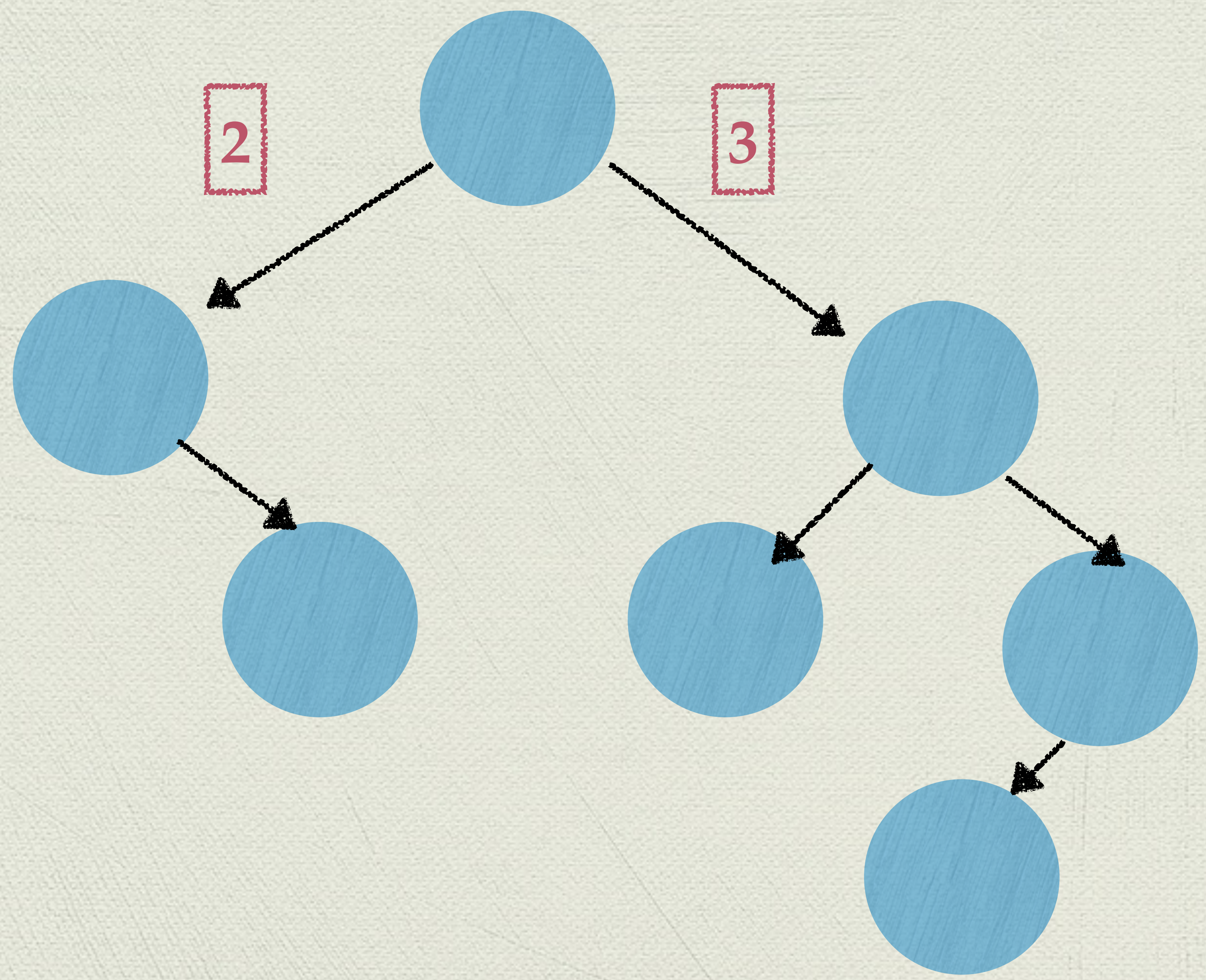
Almost balanced

- The heights of two sister subtrees cannot differ by more than one

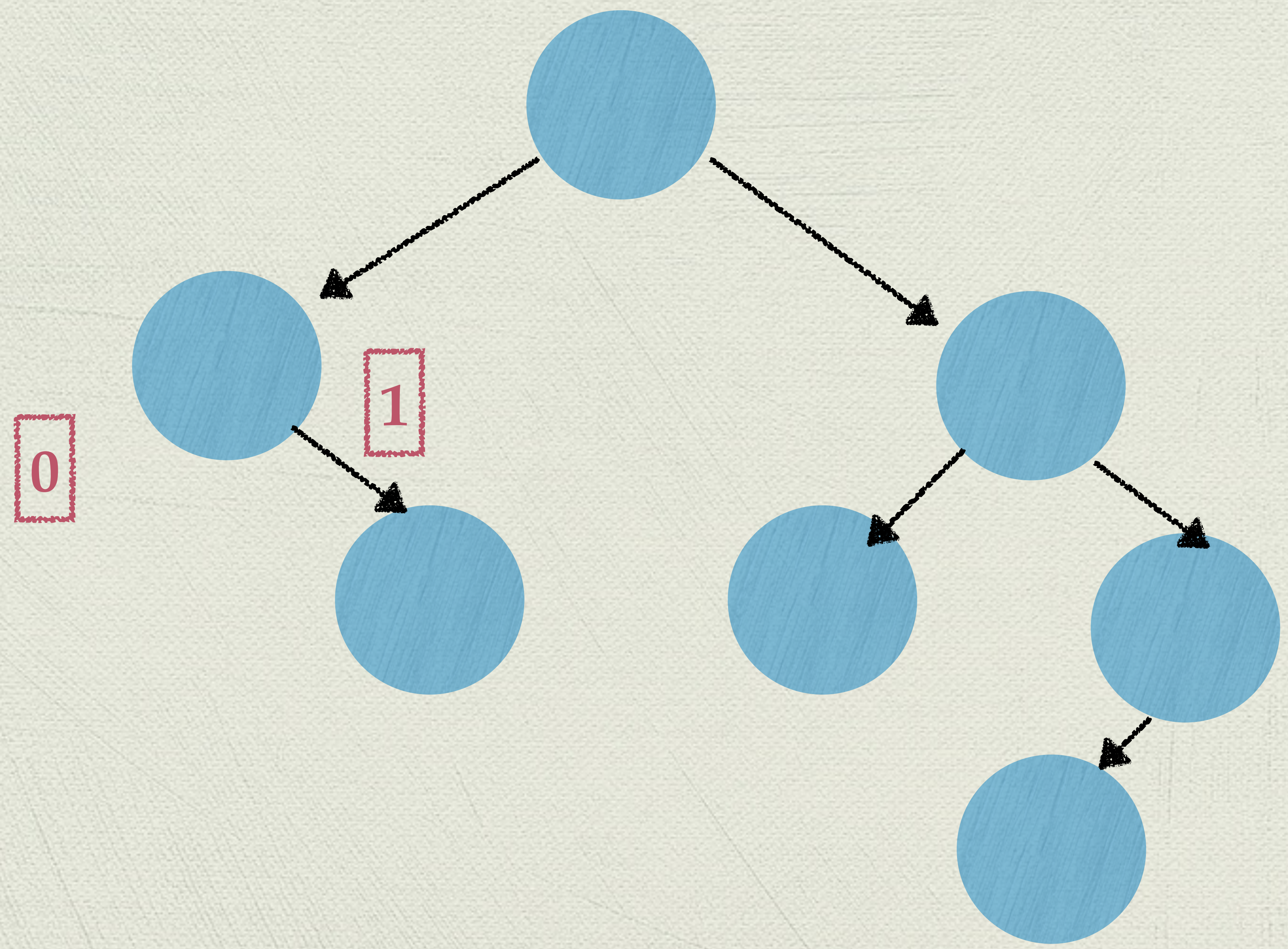
Almost balanced



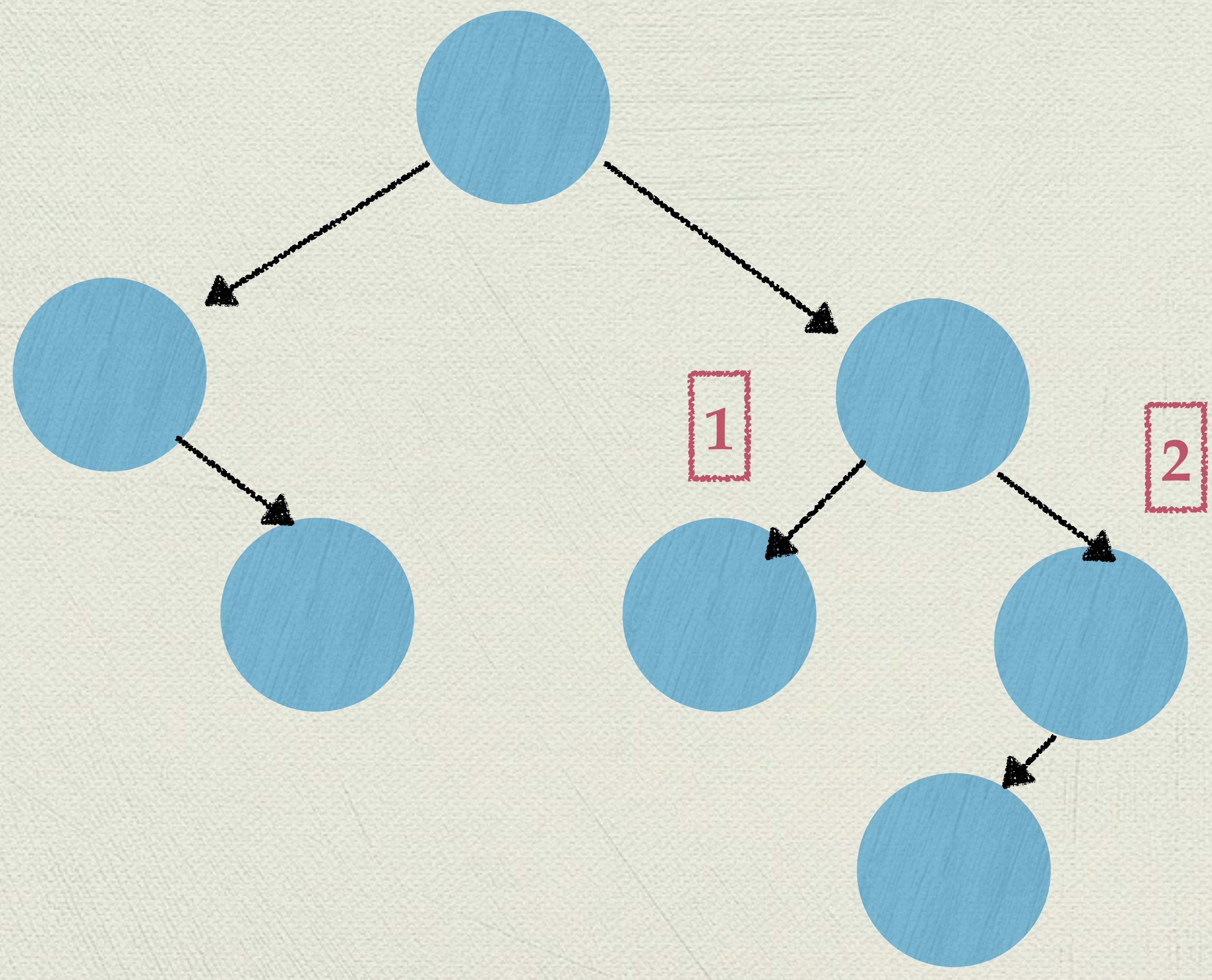
Almost balanced



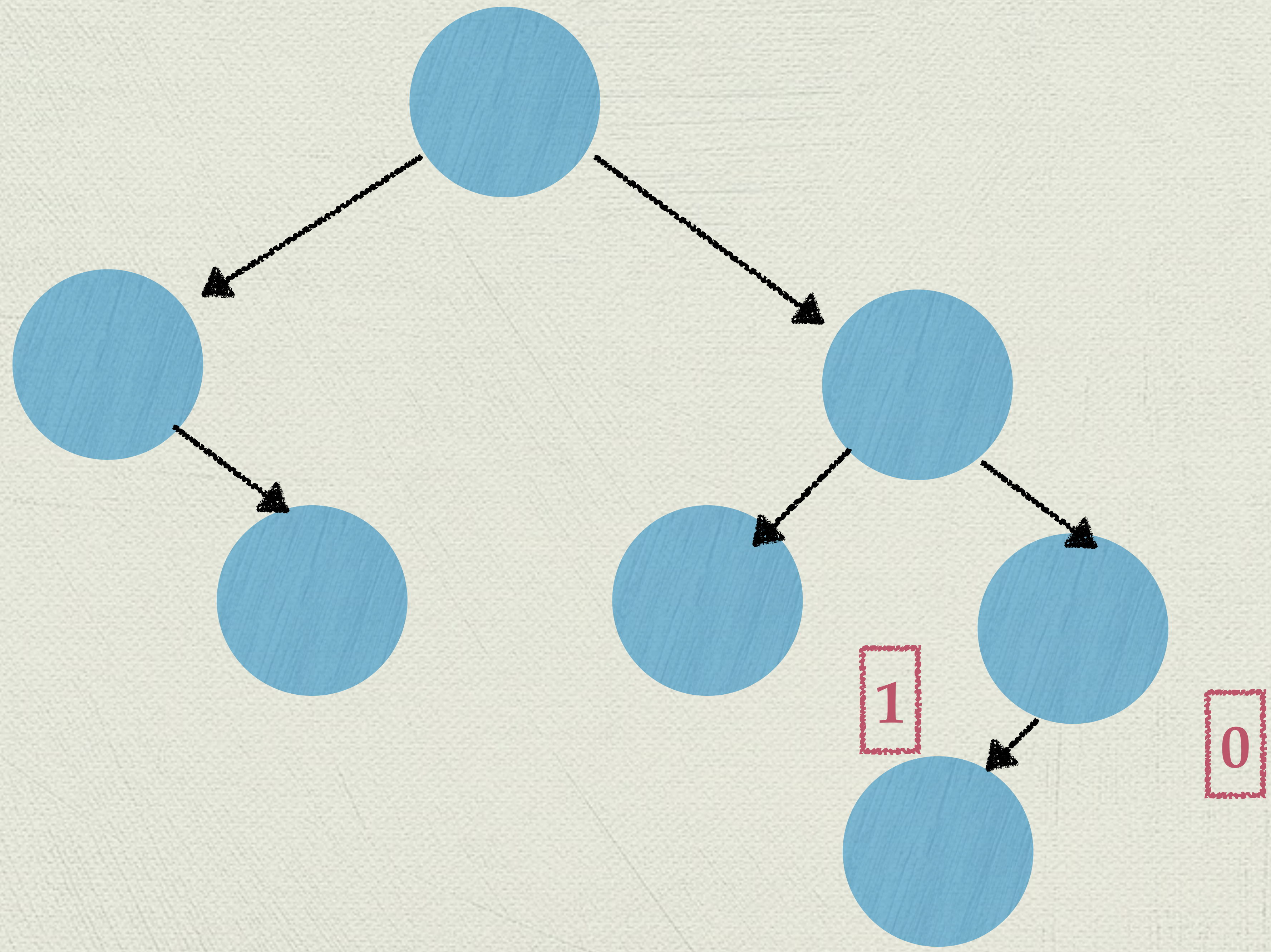
Almost balanced



Almost balanced



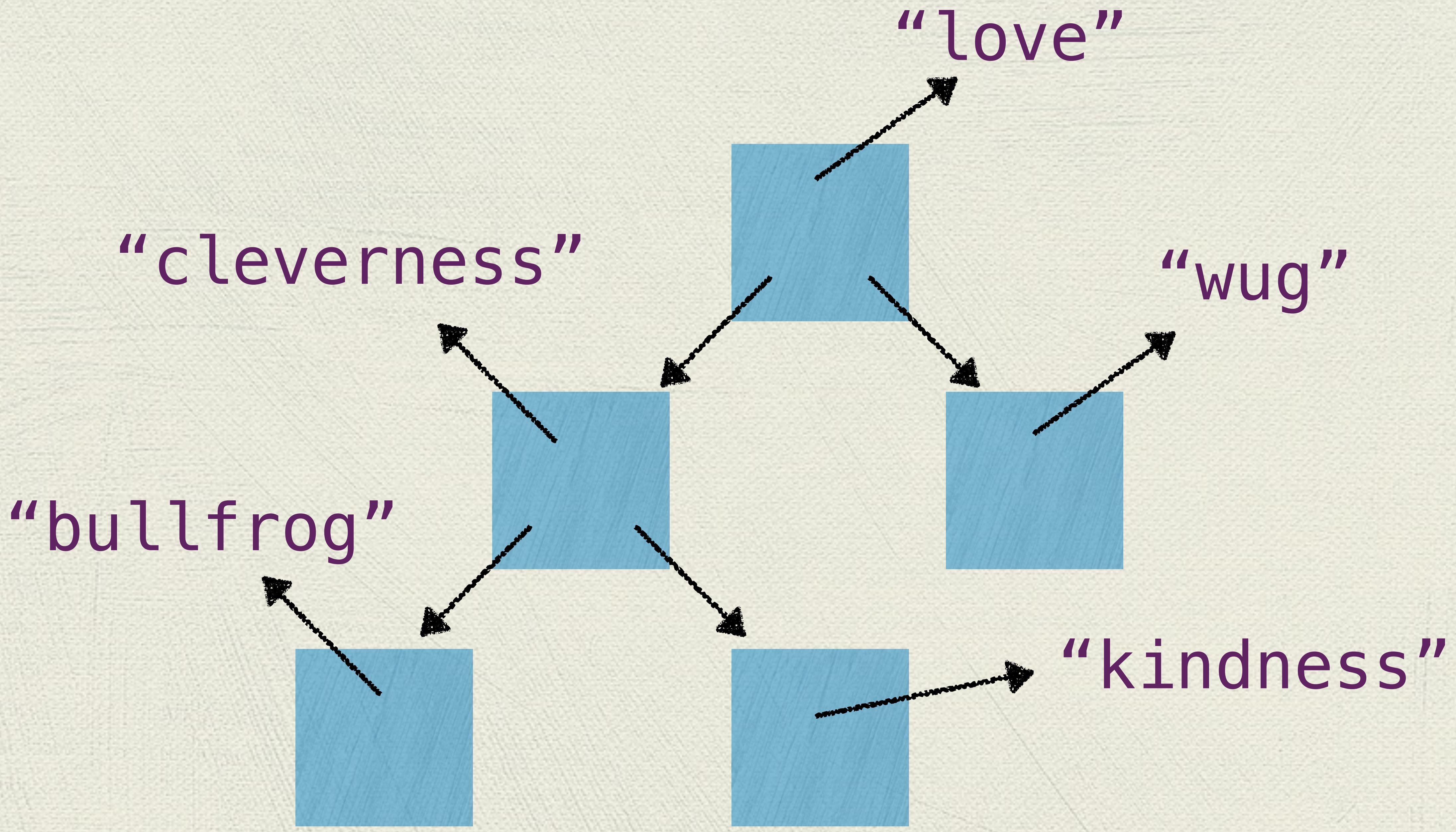
Almost balanced



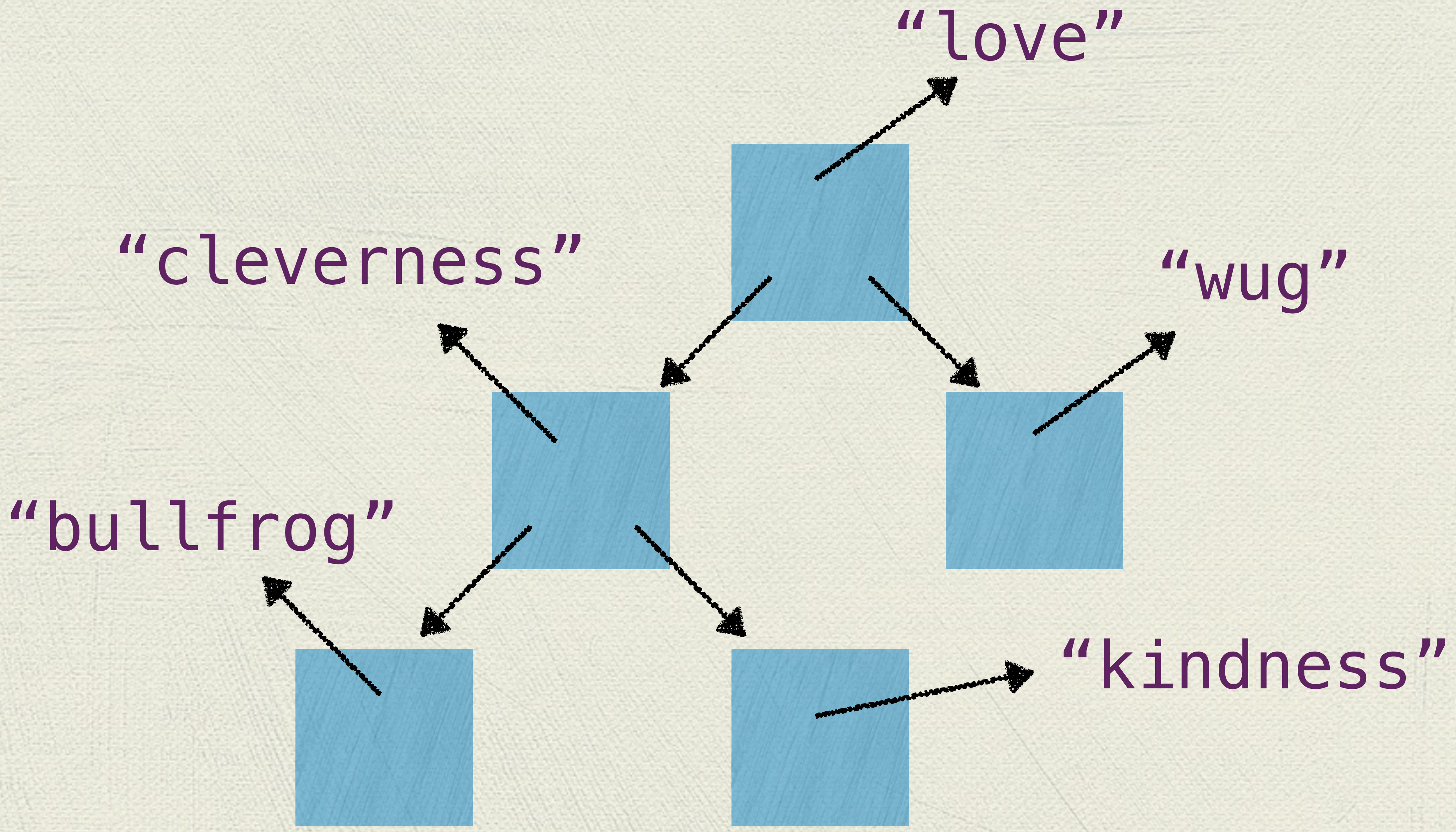
Balanced BST

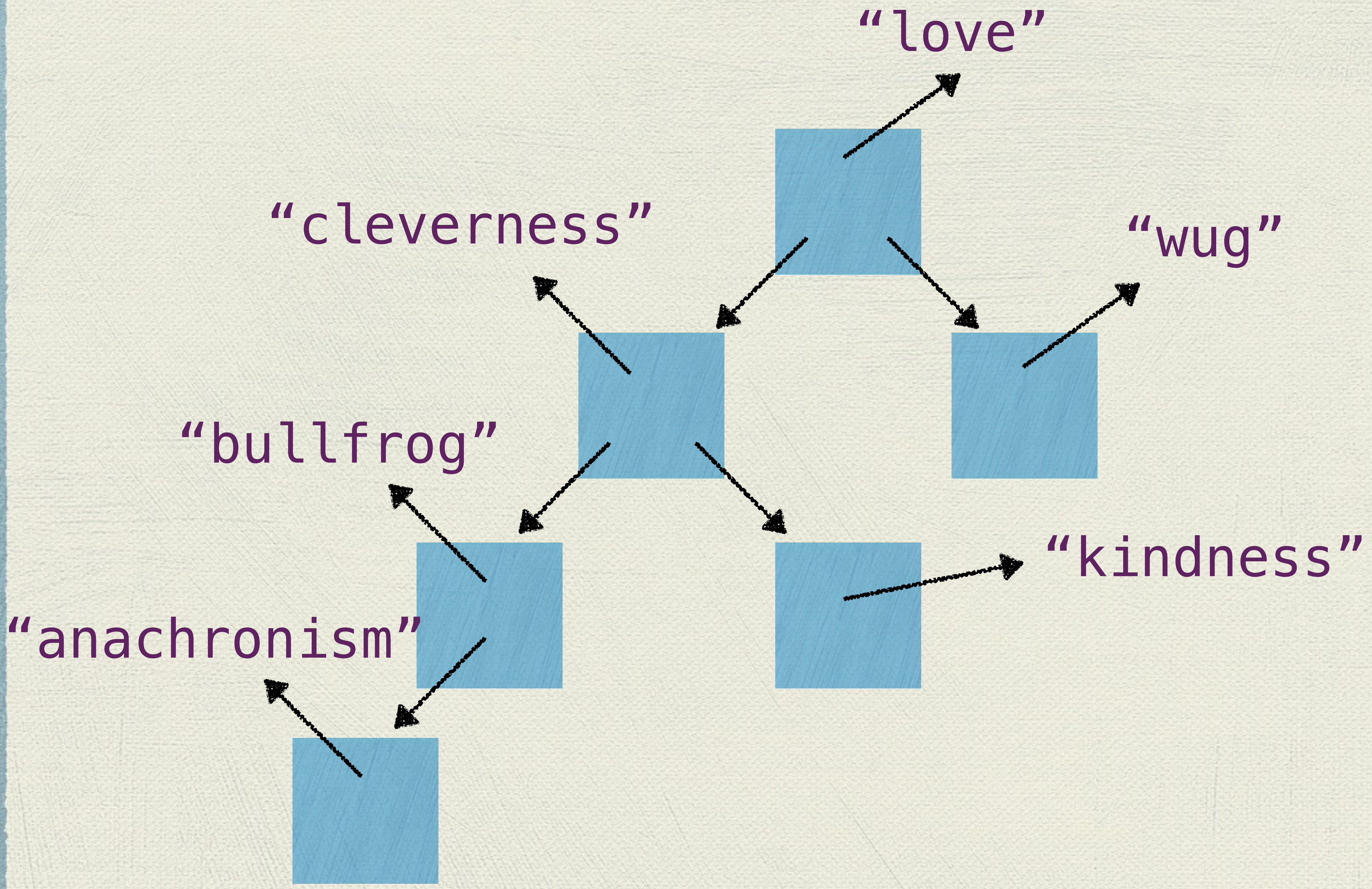
- Our BST would have fast contains if only it supported one more **invariant** — that it is balanced in some sense
- But, how can we ensure this?

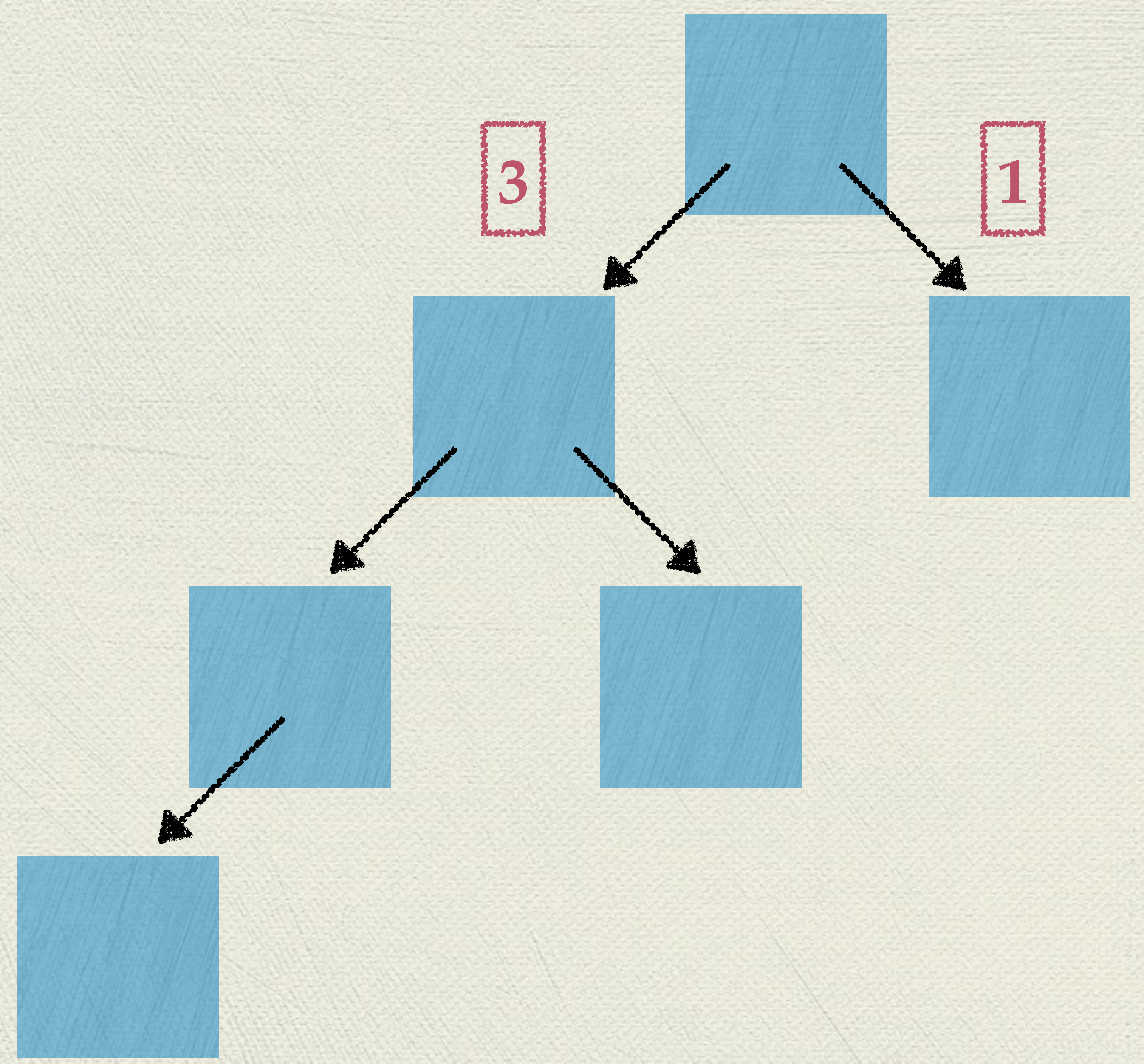
It’s almost balanced



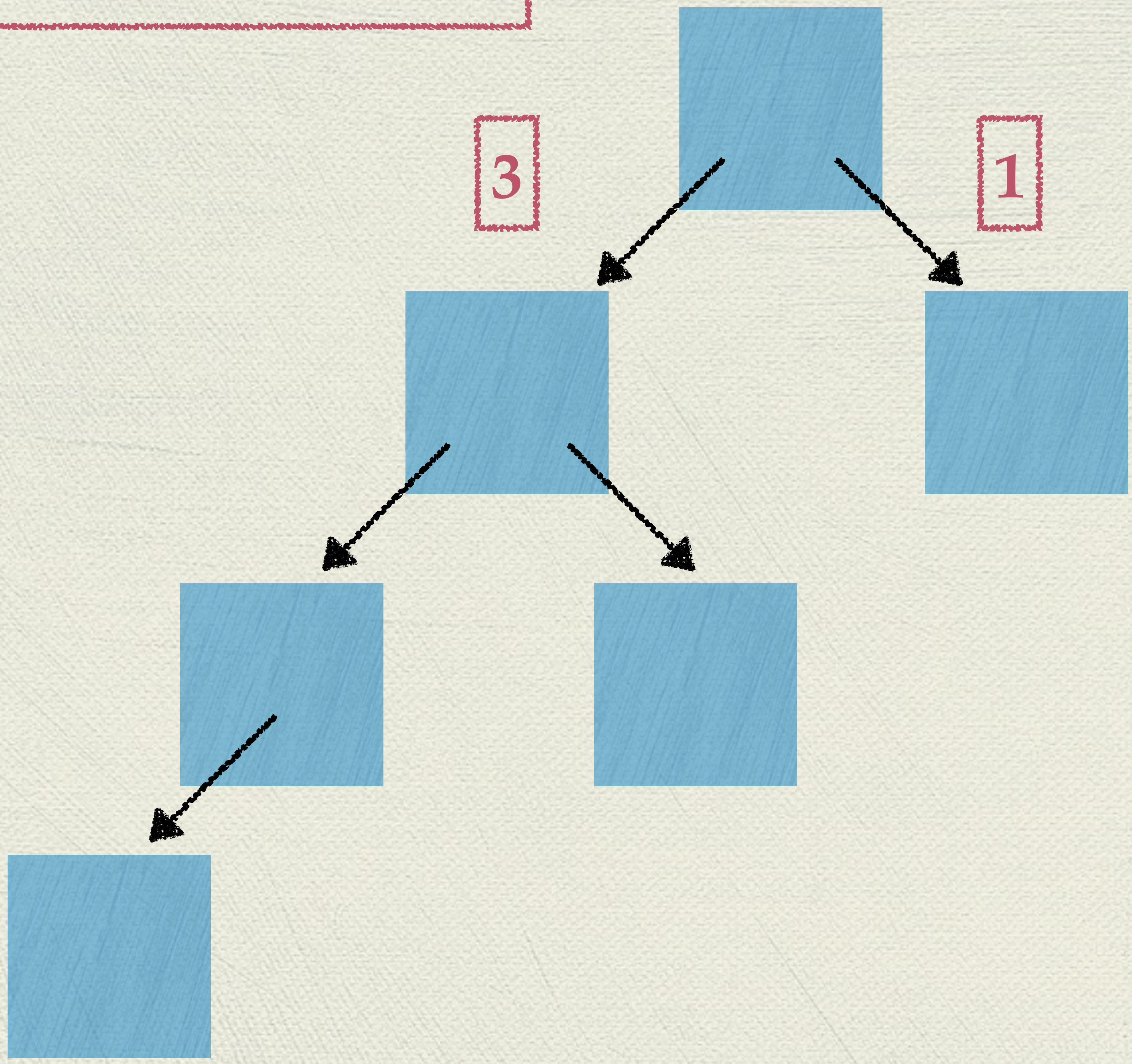
What if we insert "anachronism"?




“love”
“cleverness”
“wug”
“bullfrog”
“kindness”
“anachronism”


3
1

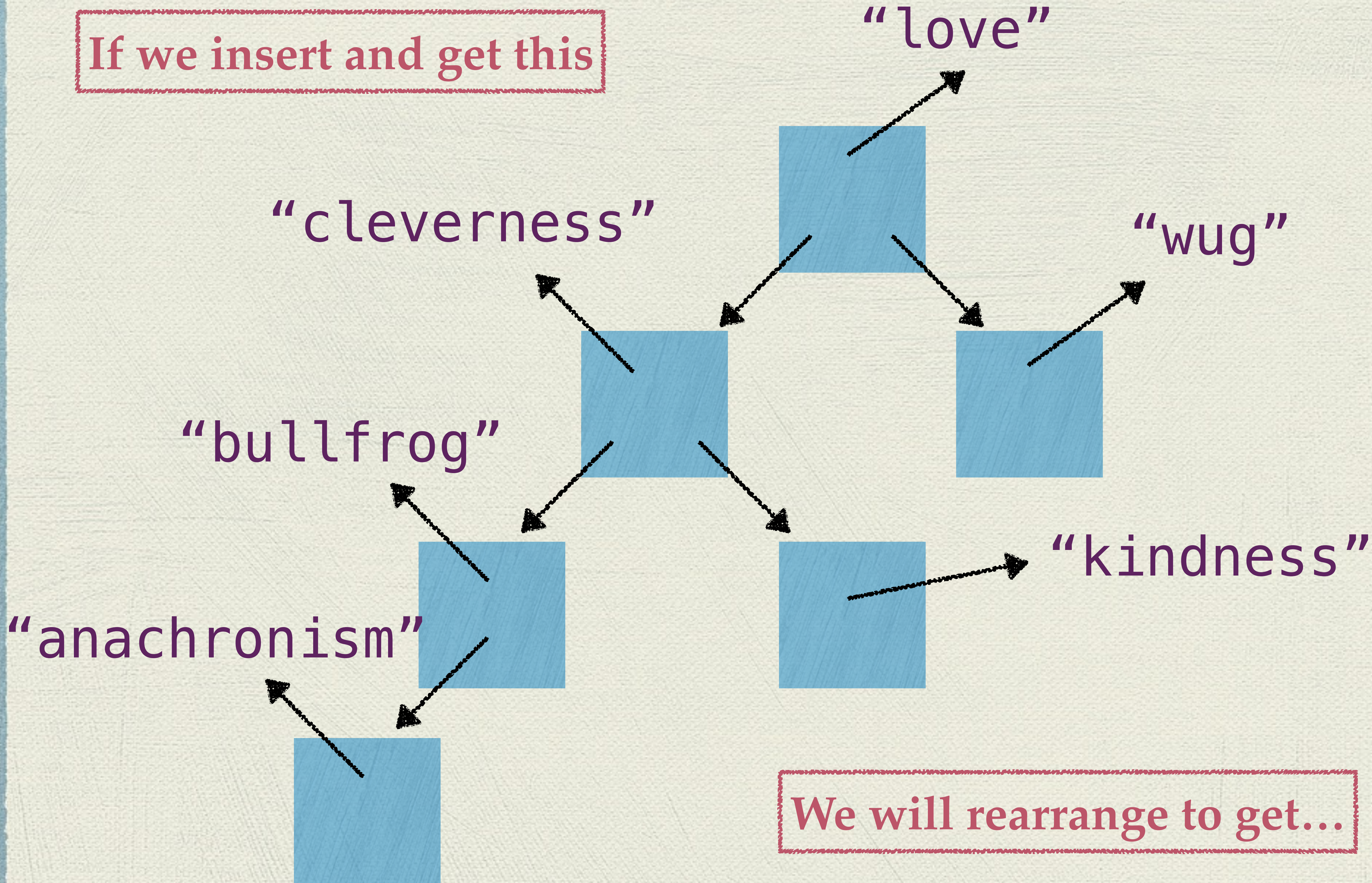
No longer almost balanced!



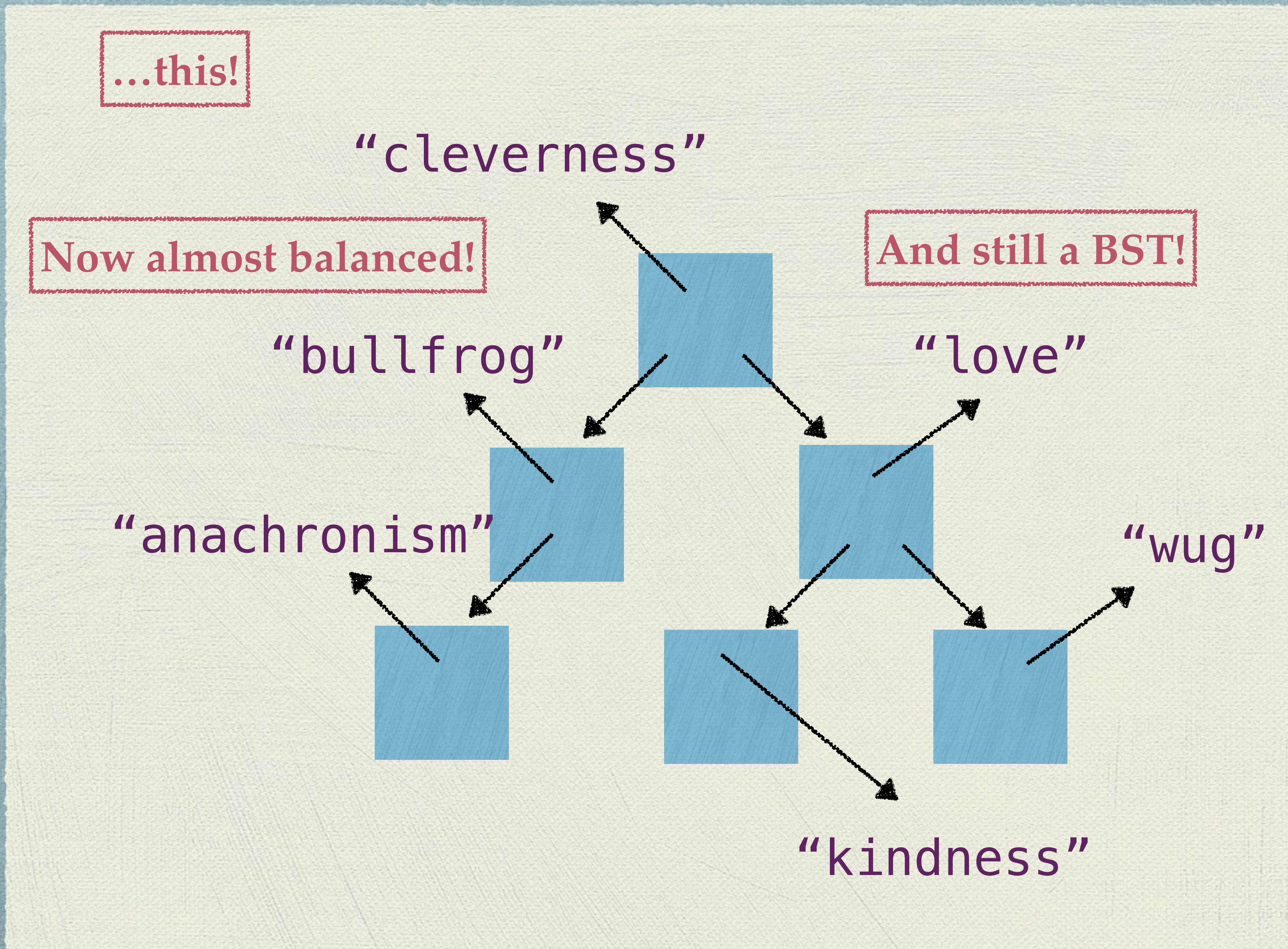
Enter the AVL tree

- The **AVL tree** (named for inventors Adelson-Velsky and Landis) is a **BST** that is always **almost balanced**
- How?
- After inserting a new item, *rearrange the tree to be more balanced*

If we insert and get this



We will rearrange to get…

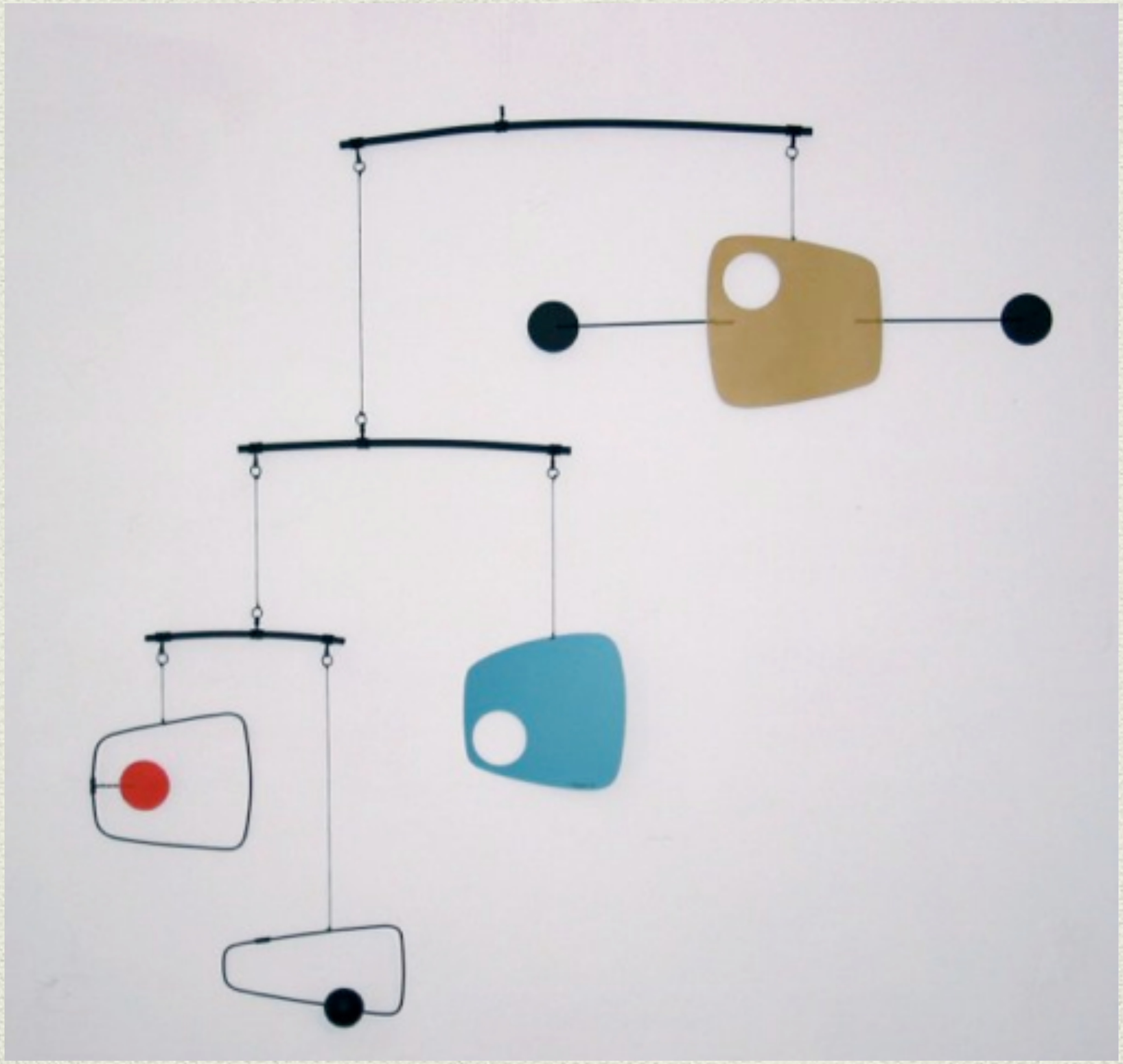

...this!
"cleverness"
Now almost balanced!
And still a BST!
"bullfrog"
"love"
"anachronism"
"wug"
"kindness"

AVL balance

- The specific operation that balances an AVL tree is called a **rotation**

Rotation intuition

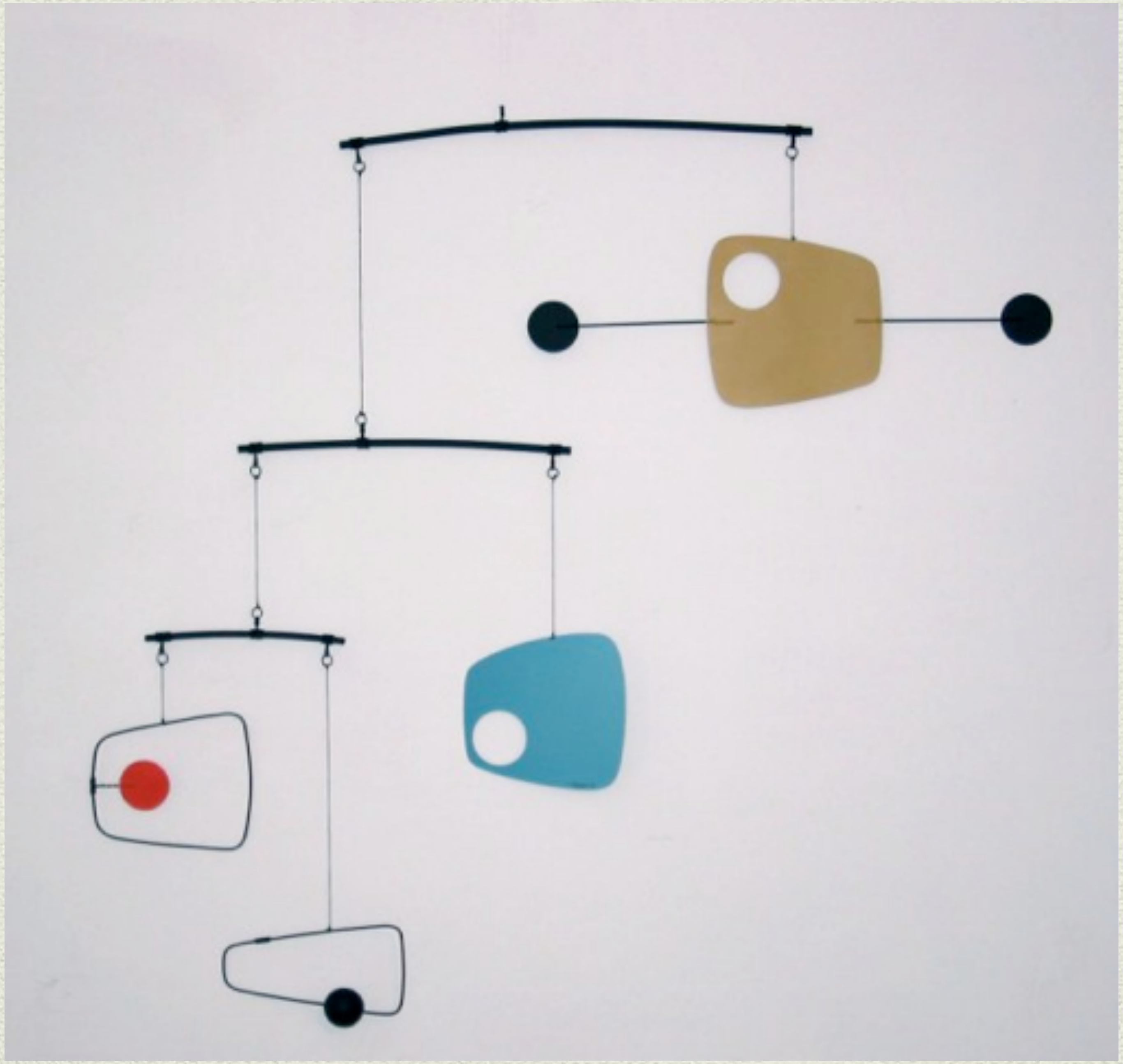
- Imagine a tree is like a hanging mobile



Credit: http://www.the-mobile-factory.com/

Rotation intuition

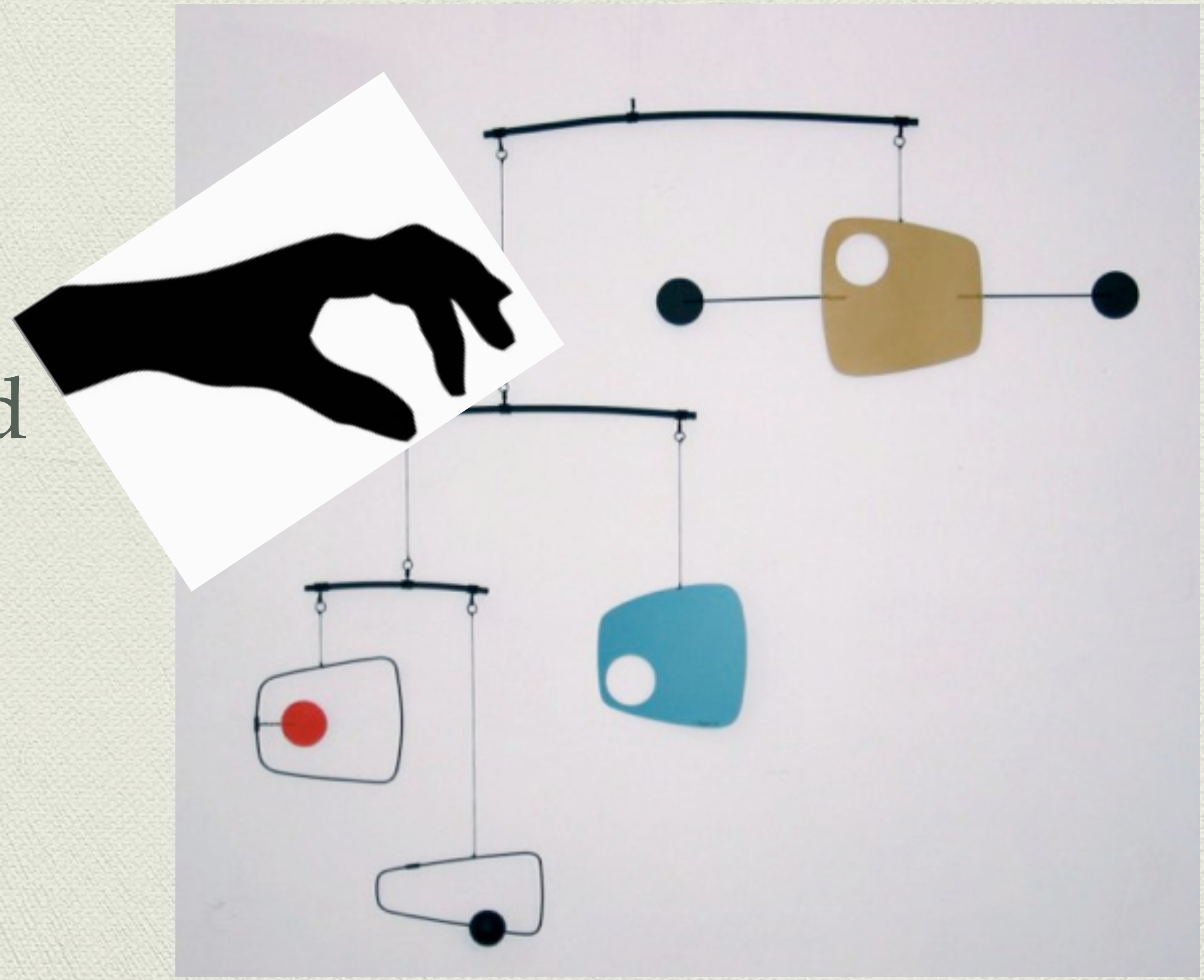
- How would you balance it?



Credit: http://www.the-mobile-factory.com/

Rotation intuition

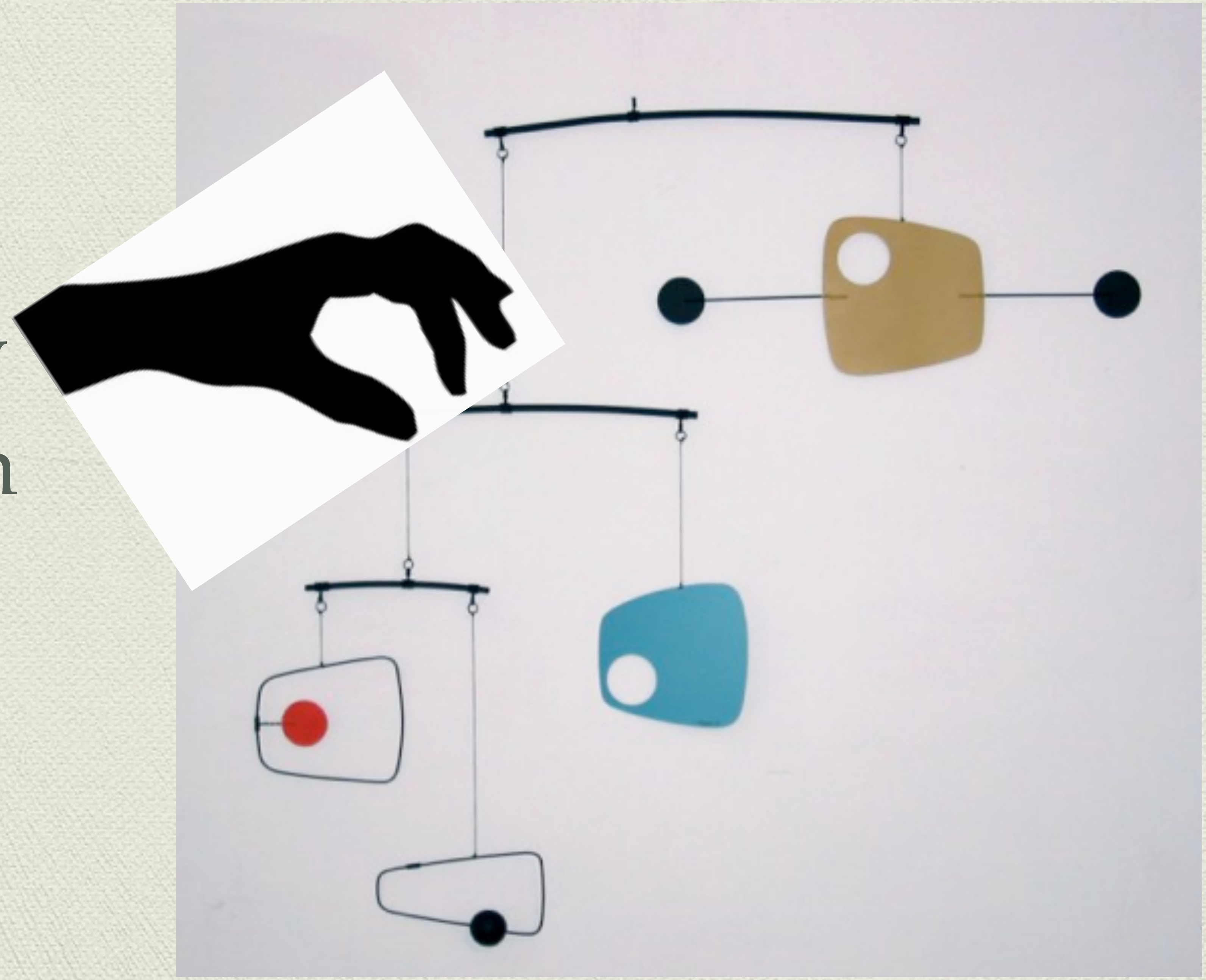
- Pinch here, and pull up!



Credit: http://www.the-mobile-factory.com/

Rotation intuition

- This is roughly what a rotation is



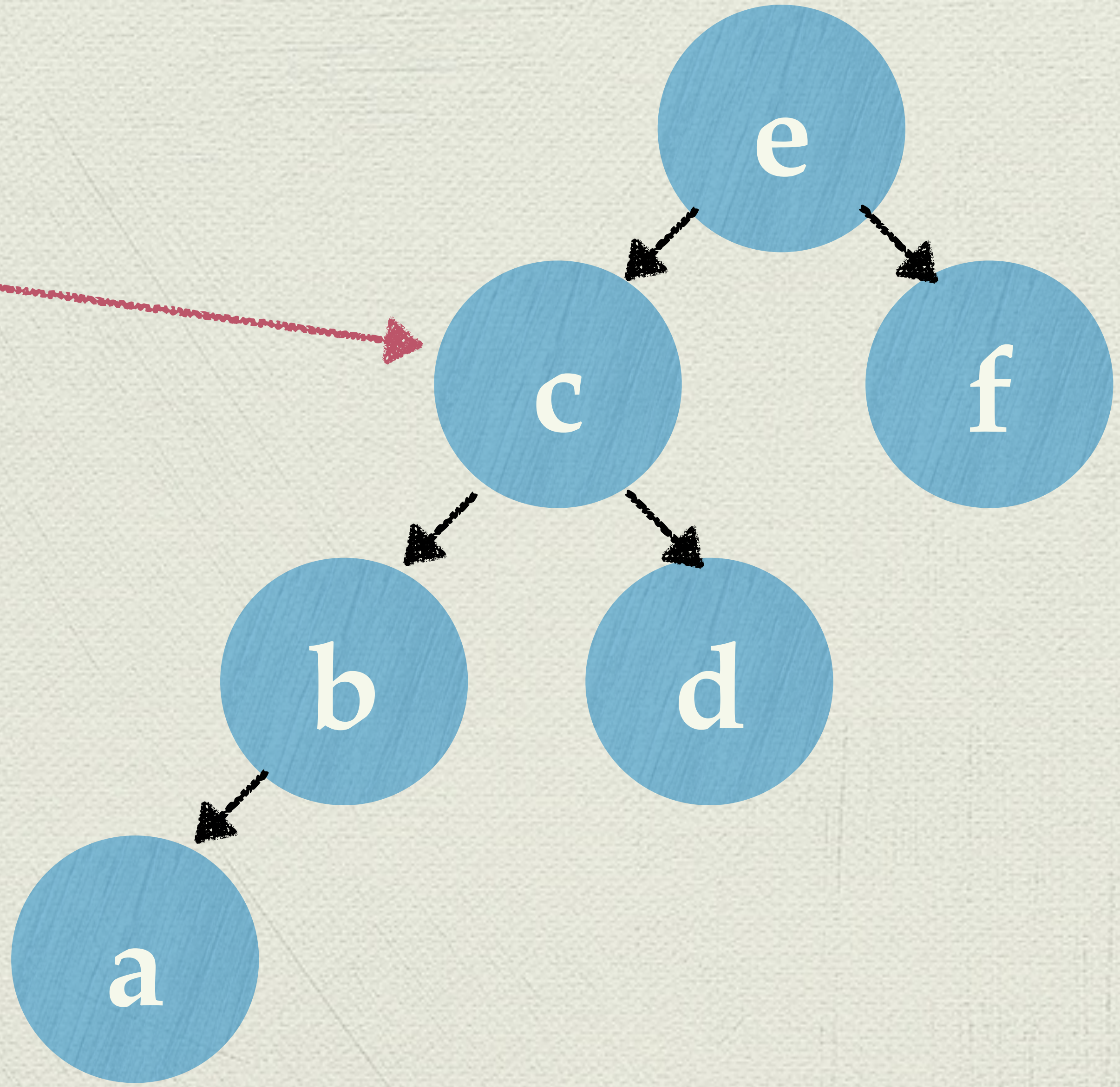
Credit: http://www.the-mobile-factory.com/

Rotate!

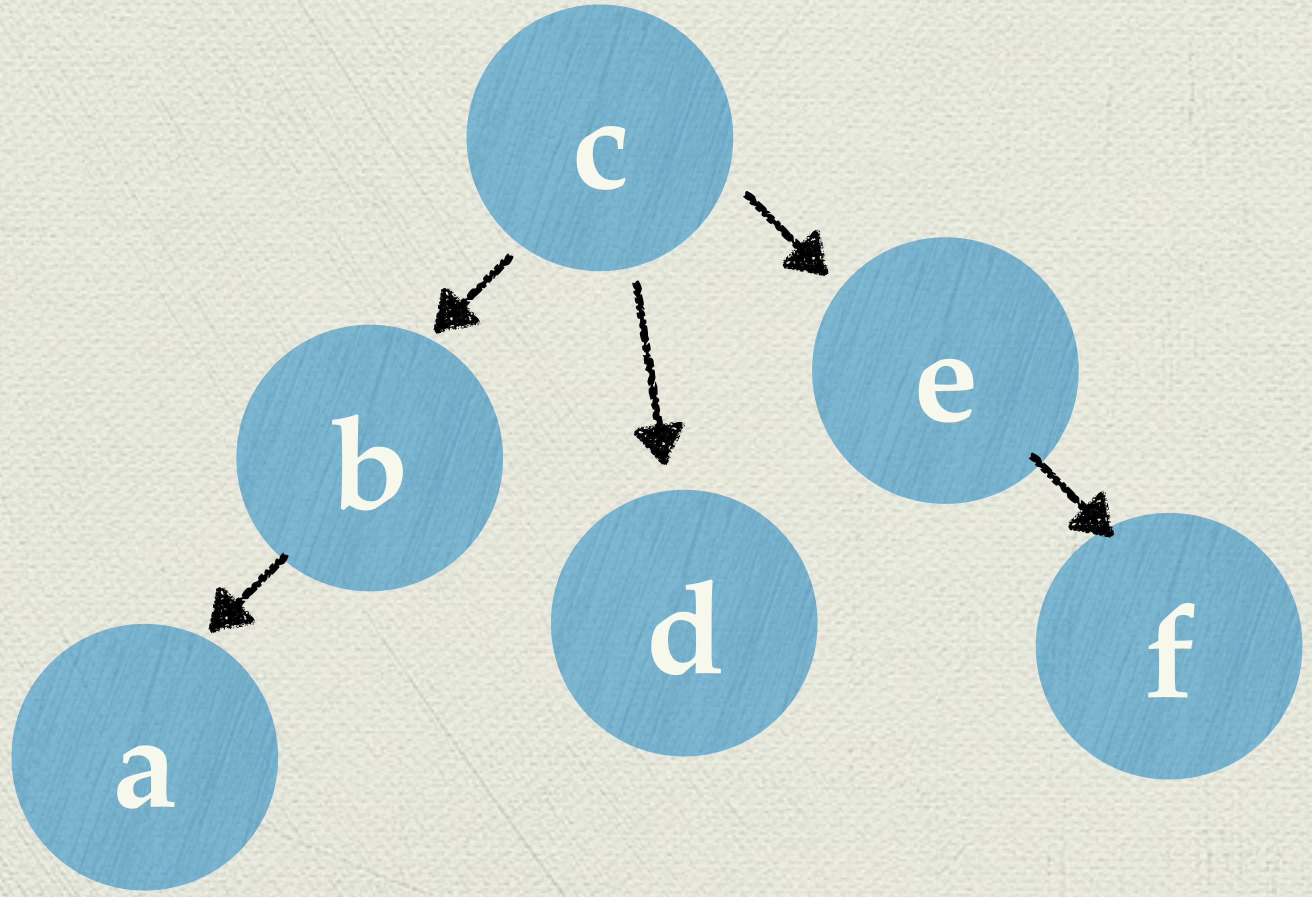
It's unbalanced!

Pull up on this node

This means its parent will become its child

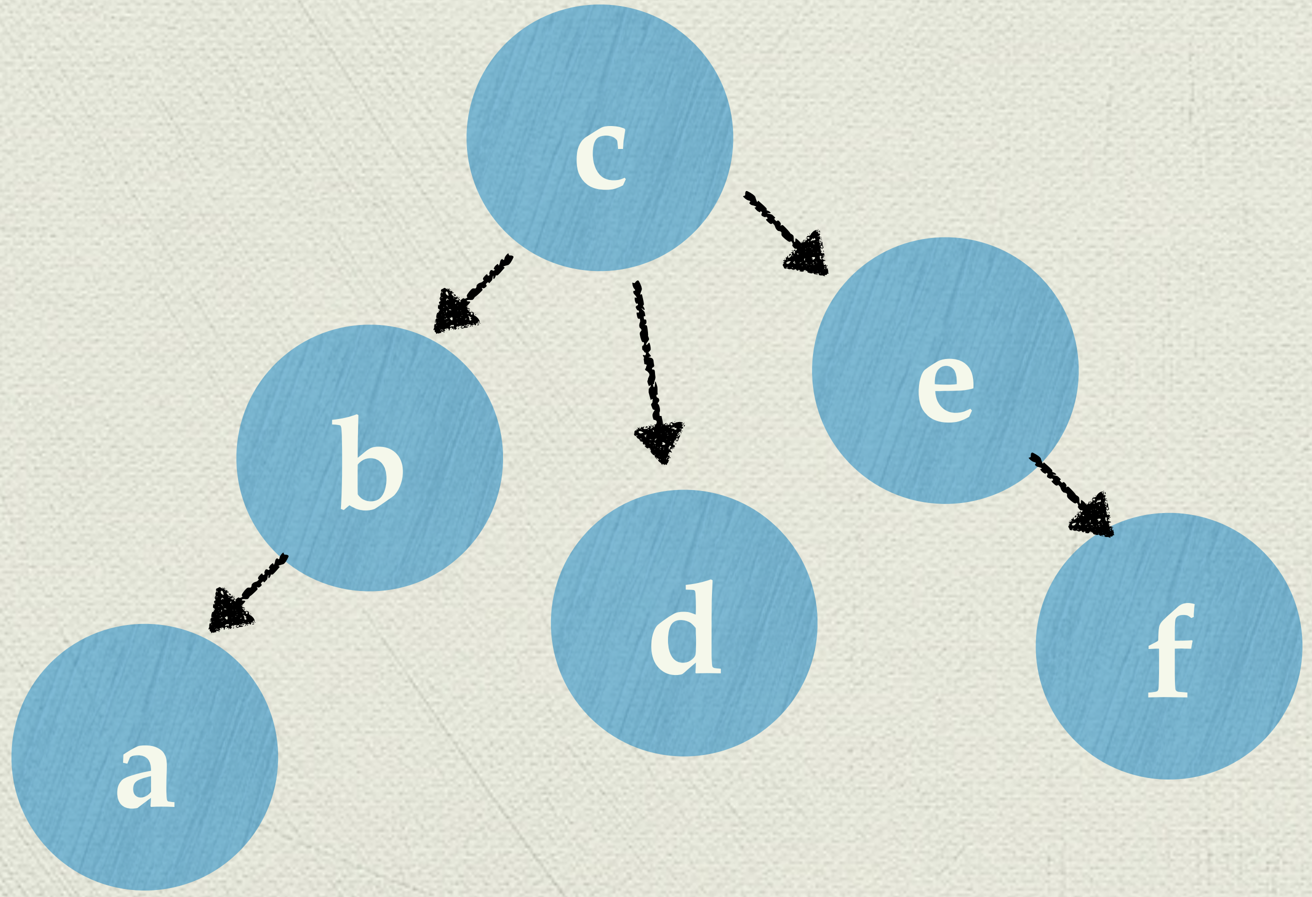


Basically this



Basically this

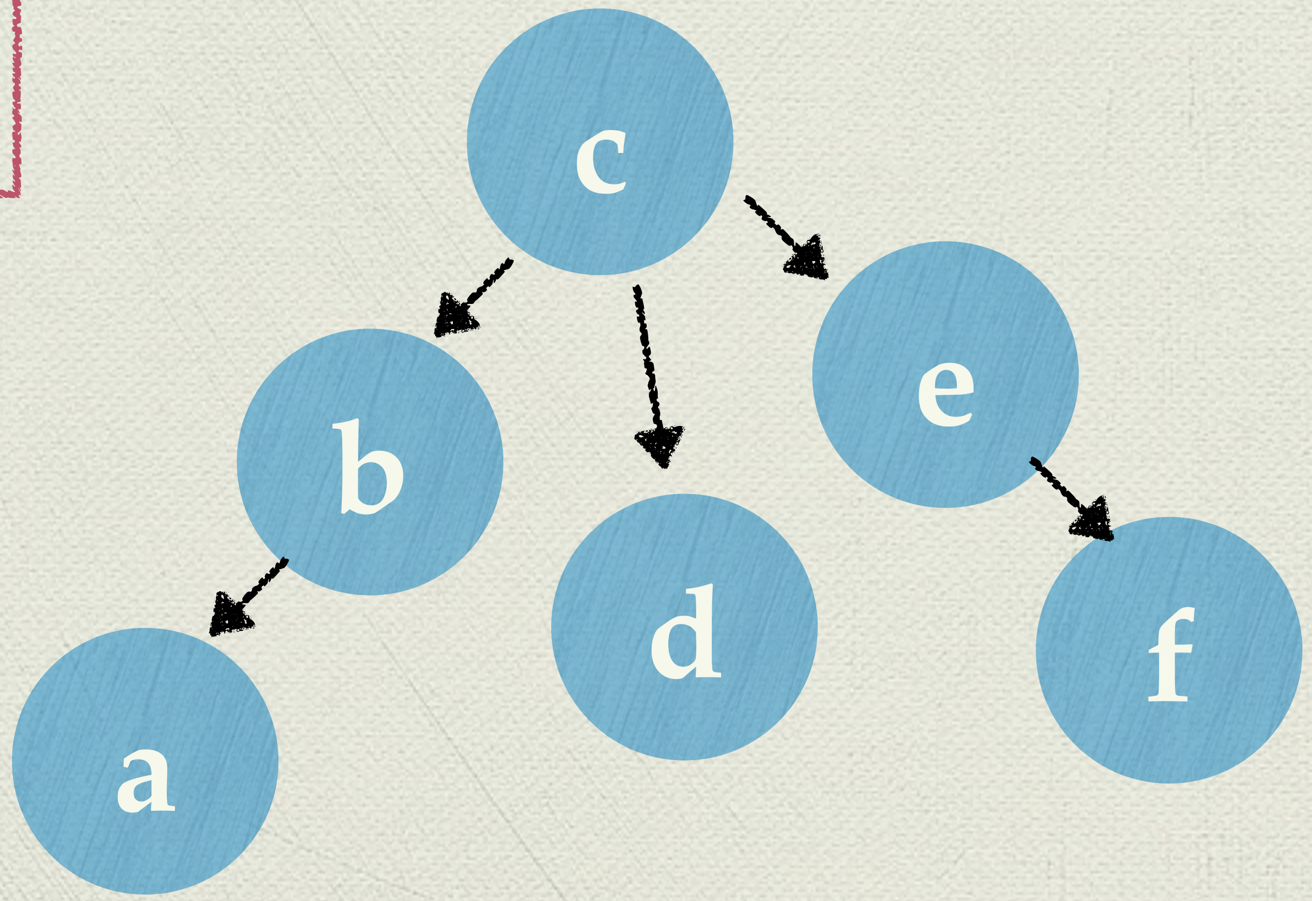
But now c has three children



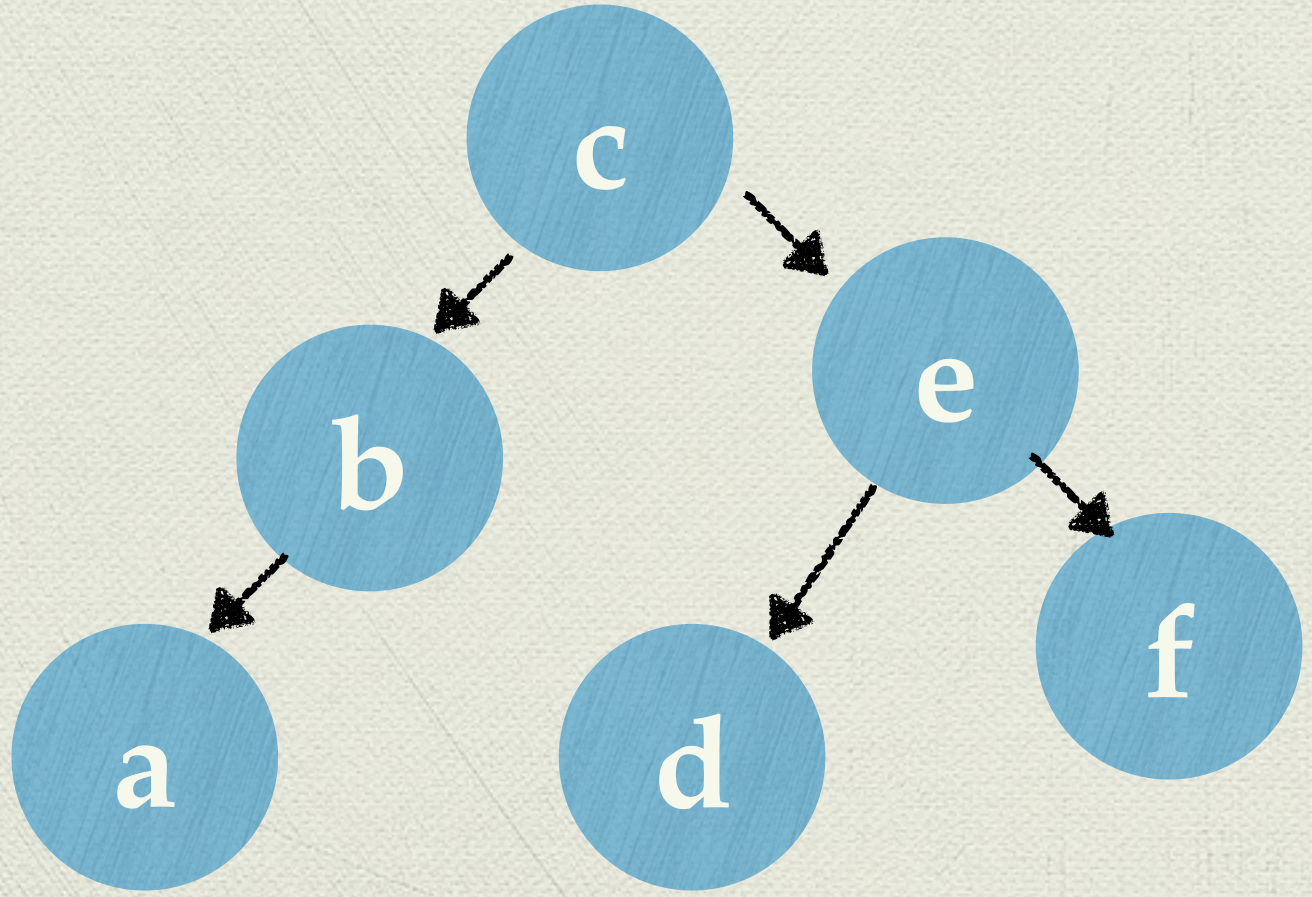
Basically this

But now c has three children

Luckily, e only has 1, because c used to be its child, but isn't anymore



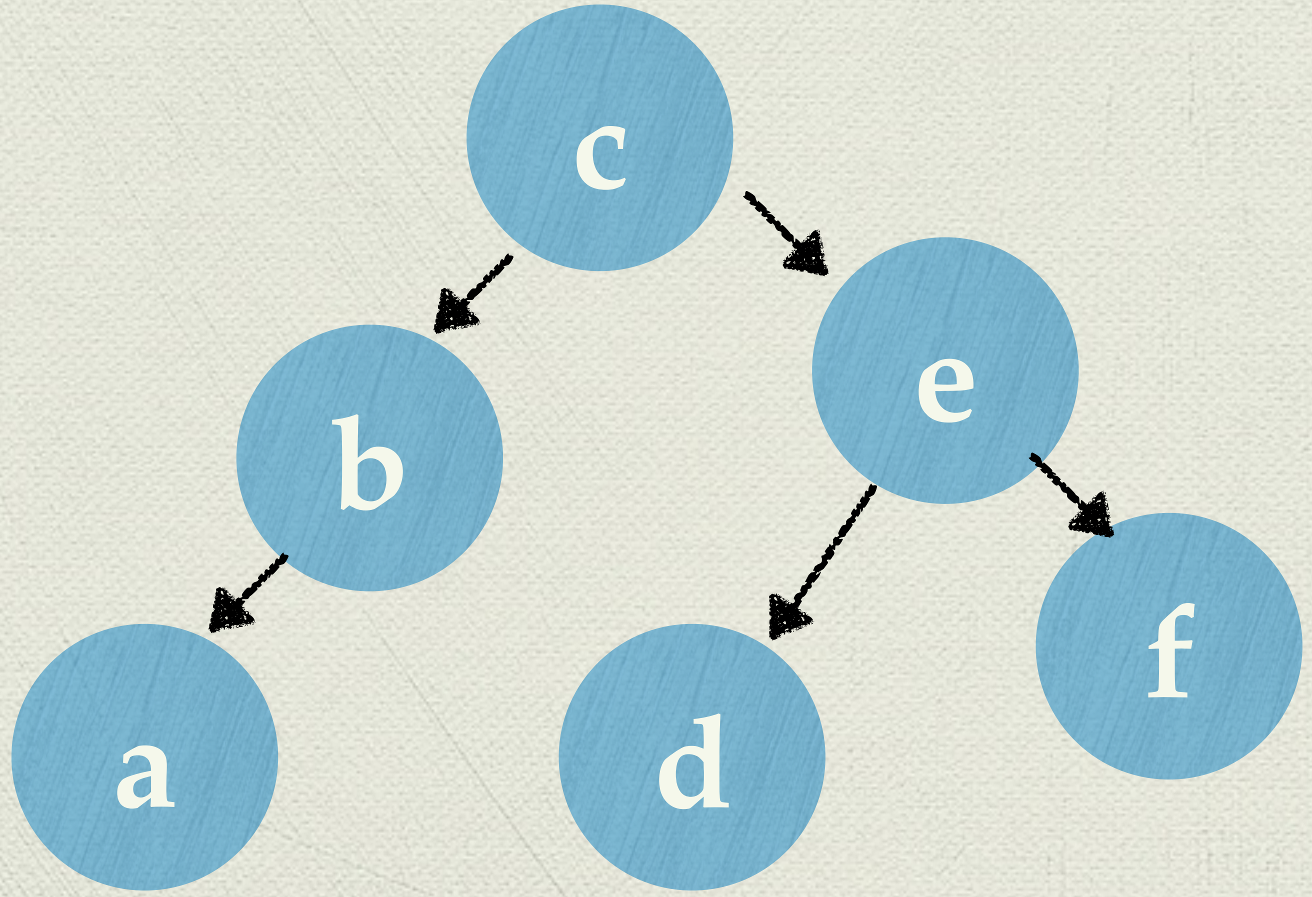
So actually this



So actually this

Notice!!

We preserved the BST ordering property

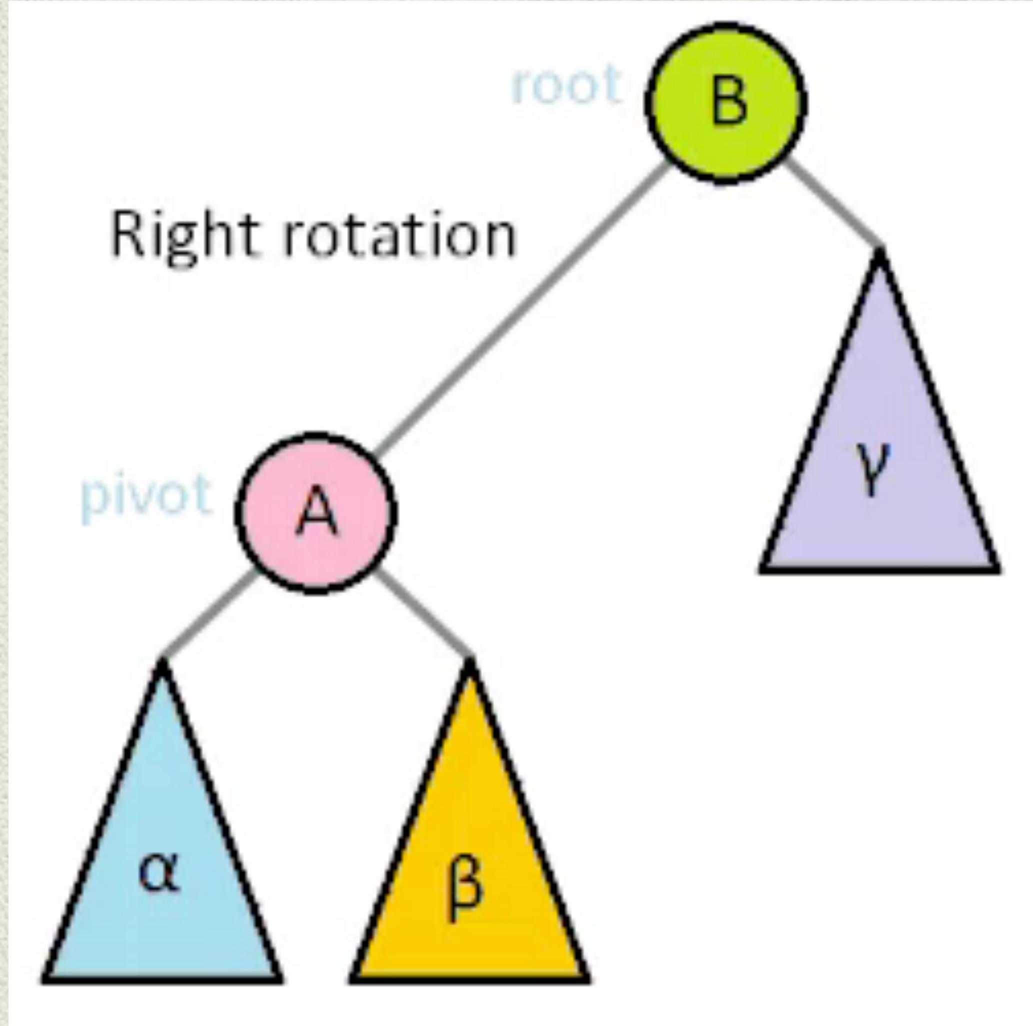


Rotations

- We just **rotated C right**… this means C’s parent becomes its right child
- If the tree had been misbalanced the other way, we might have **rotated a node left** instead

Rotations

- Wikipedia has a wonderful animation

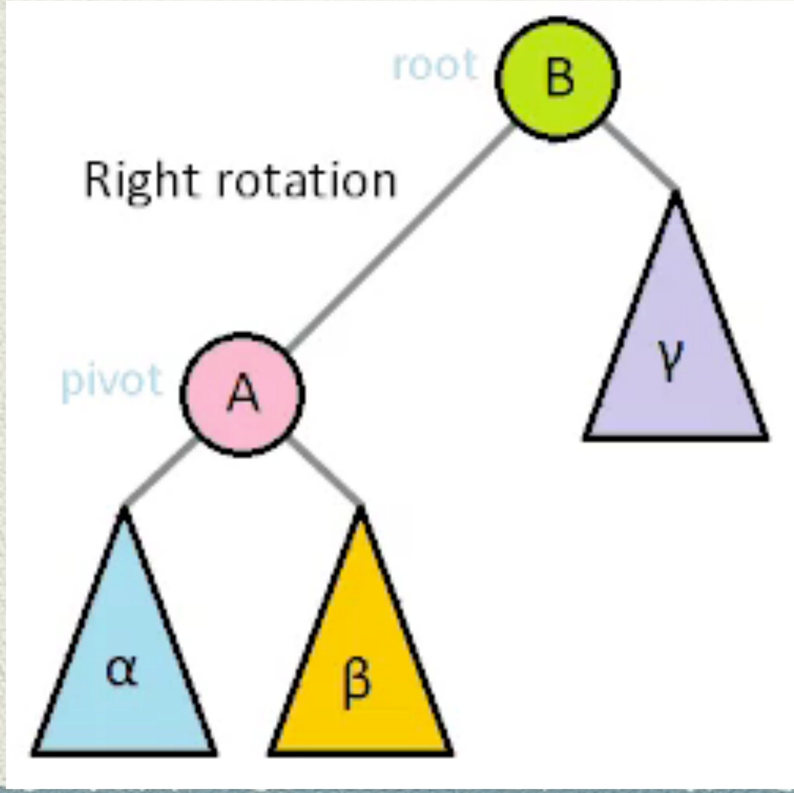


Rotations

- The triangles in the previous animation indicate **subtrees**. That is, there could be a lot more nodes under them

Rotations

- Technical definition of rotation: if after inserting you find that **alpha** is too tall relative to **beta**, then turn **B** into **A**'s child, and make **beta** a child of **B**



Rotations

- Technical definition of rotation: if after inserting you find that **alpha** is too tall relative to **beta**, then turn **B** into **A**′s child, and make **beta** a child of **B**
- There is a mirror-image case, as well

Unfortunately

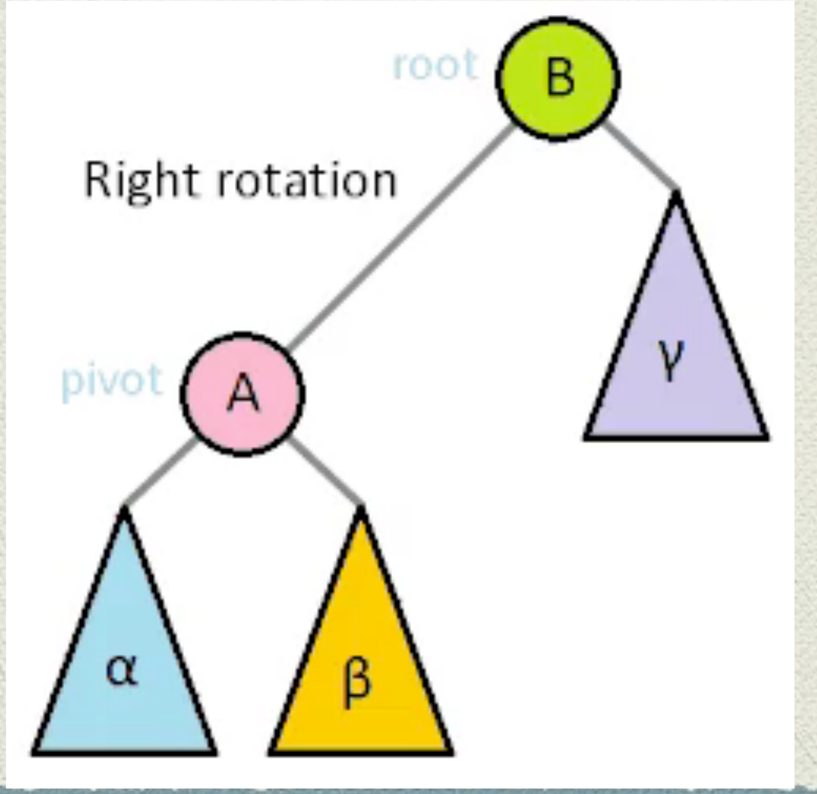
- It’s not quite that simple
- (Are you kidding me?!)

Rotations in more detail

- I described the process of rotation accurately to you
- However, sometimes a single rotation is not enough to balance an AVL tree after an insertion
- Sometimes two rotations are needed (but no more)

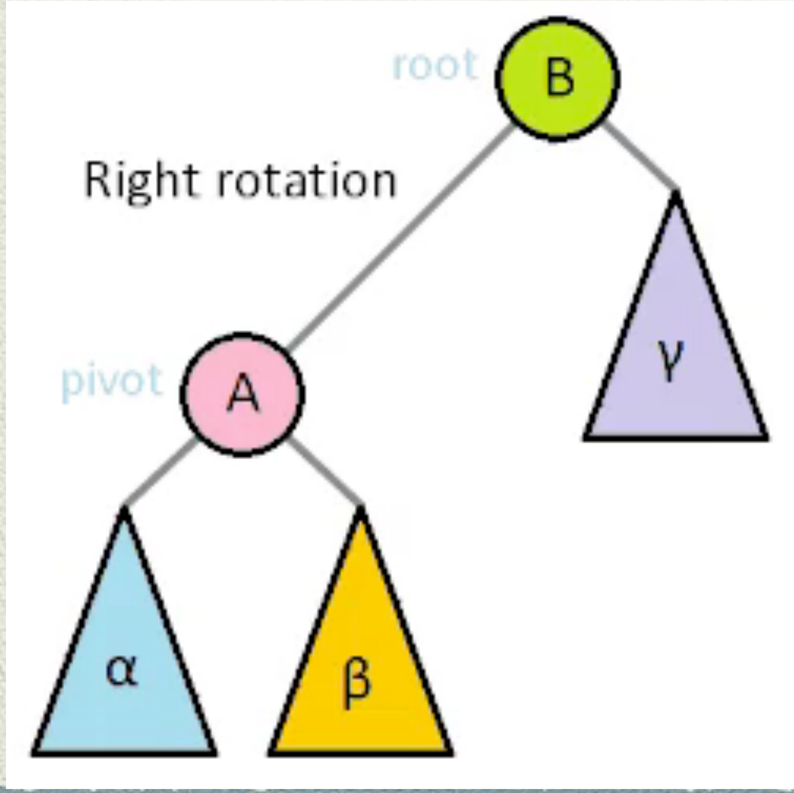
Rotations

- If **alpha** was heavier than **Beta**, we rotated **A** right and were done
- What if **Beta** was heavier than **alpha**?



Rotations

- If **Beta** is heavier than **alpha…**
  - first rotate the root of Beta left. Now root of Beta takes A's place
  - Then rotate the root of Beta right. Now root of Beta is the highest node



Phew!

- So that was rotations, huh?

Runtimes

- During a rotation, we only reassign ~6 references, no matter how many nodes in the tree there are
- If we always rebalance as soon as the tree becomes unbalanced, we only have to do max of 2 rotations to fix things
- This means: fixing balance is **O(1)** time!!!

Runtimes — AVL tree set

- **add** an item to the set
  - Traverse down to the correct spot, put the item: **O(logN)**
  - Maybe apply rotations to fix balance **O(1)**
- **check** if set contains an item
  - Traverse down to the correct spot: **O(logN)**

The score board

- HashSet:
  - **add**: Guaranteed **O(1)**, if fast hashCode
  - **contains**: Average **O(1)**, worst-case **O(N)**
- TreeSet:
  - **add**: Guaranteed **O(logN)**
  - **contains**: Guaranteed **O(logN)**

The score board

- HashSet generally outdoes TreeSet, and so is far more common
- But, TreeSet does beat HashSet in the worst-case, if you need to be worried about that

The score board

- TreeSet has one other major advantage over HashSet
- In HashSet, items were *hashed*, or scrambled
- In TreeSet, items are *organized* in sorted order

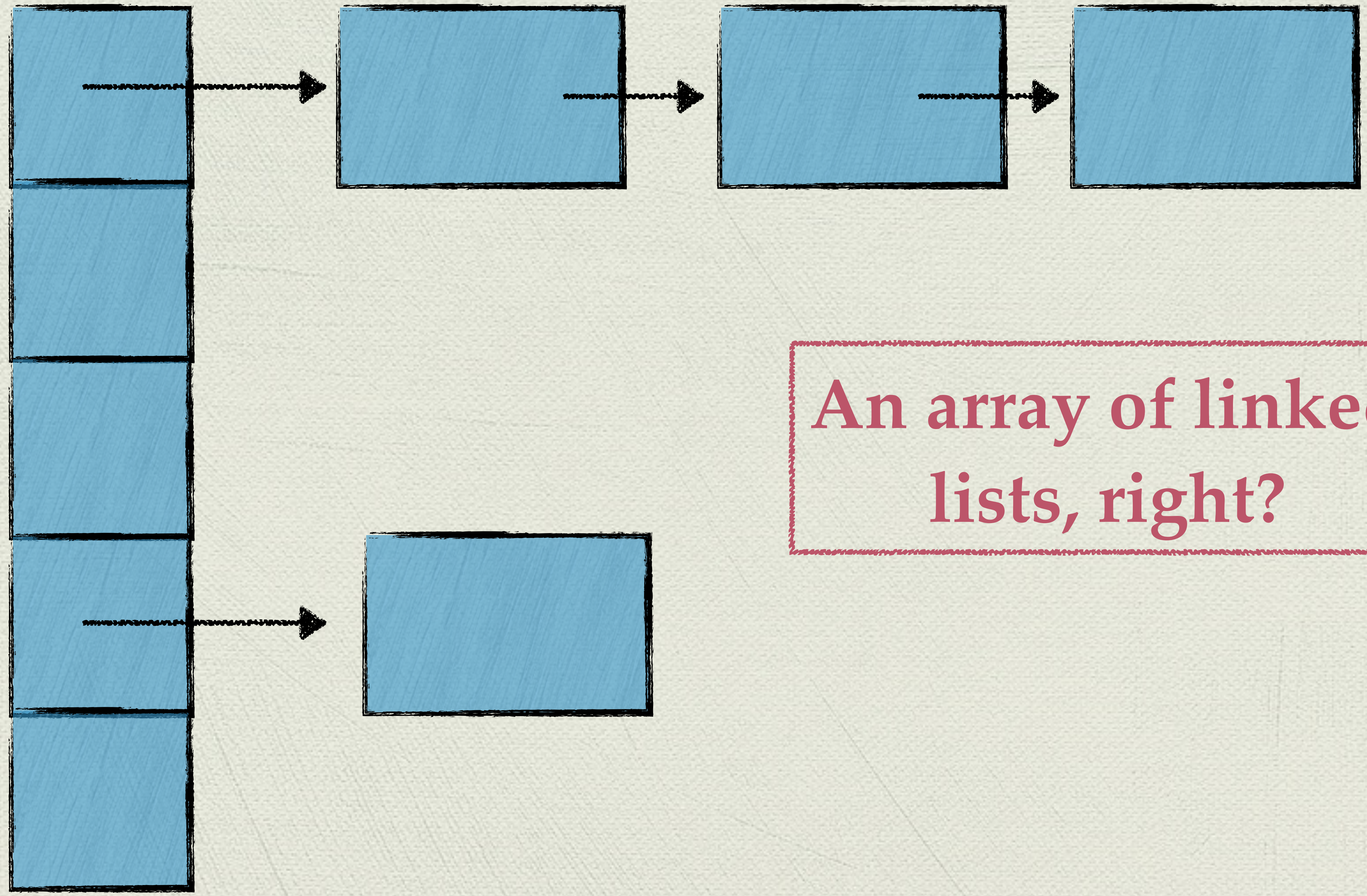
The score board

- In TreeSet, items are *organized* in sorted order
- This means you could also use TreeSet to find items close to a given item, among other things
- For example, TreeSet has a method `higher`, that returns the closest element in the set higher than an input one

And we're not done yet!!

- Although TreeSet tends to be a worse set than HashSet…
- It's definitely better than using a LinkedList as a set (asymptotically)

Remember the HashSet?



An array of linked lists, right?

And we're not done yet!!

- Since Java 8 (released last year), Java's HashMap will use a tree in each bucket instead of an array, if there are too many items in the bucket
- Cool, right?

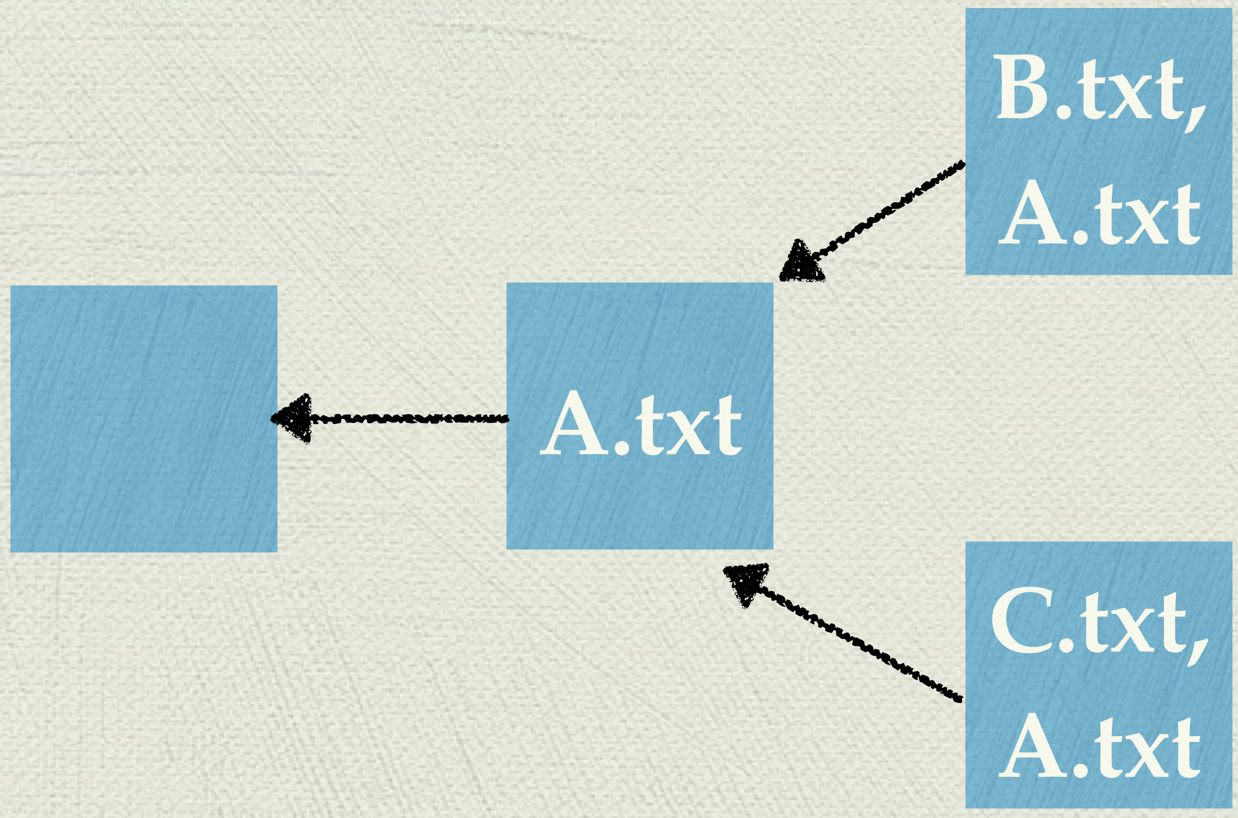
One final note

- We learned about AVL trees, the first balanced BST invented
- In practice, Java uses something called a **red-black tree**, which is similar in concept to AVL tree, but slightly more complicated

Let’s talk about gitlet

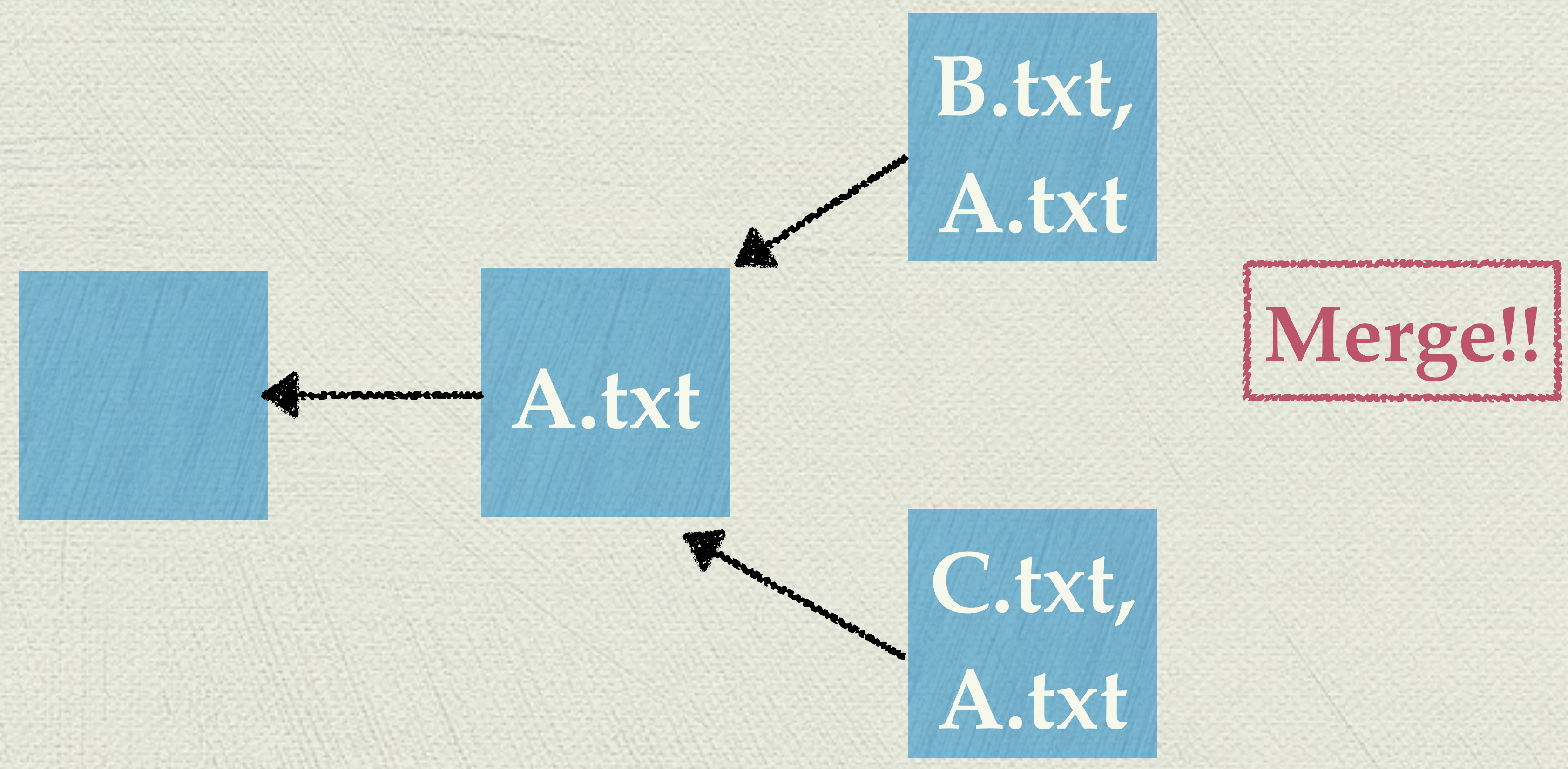
Merge

- When we merge, we want to create a new commit with files from both branches



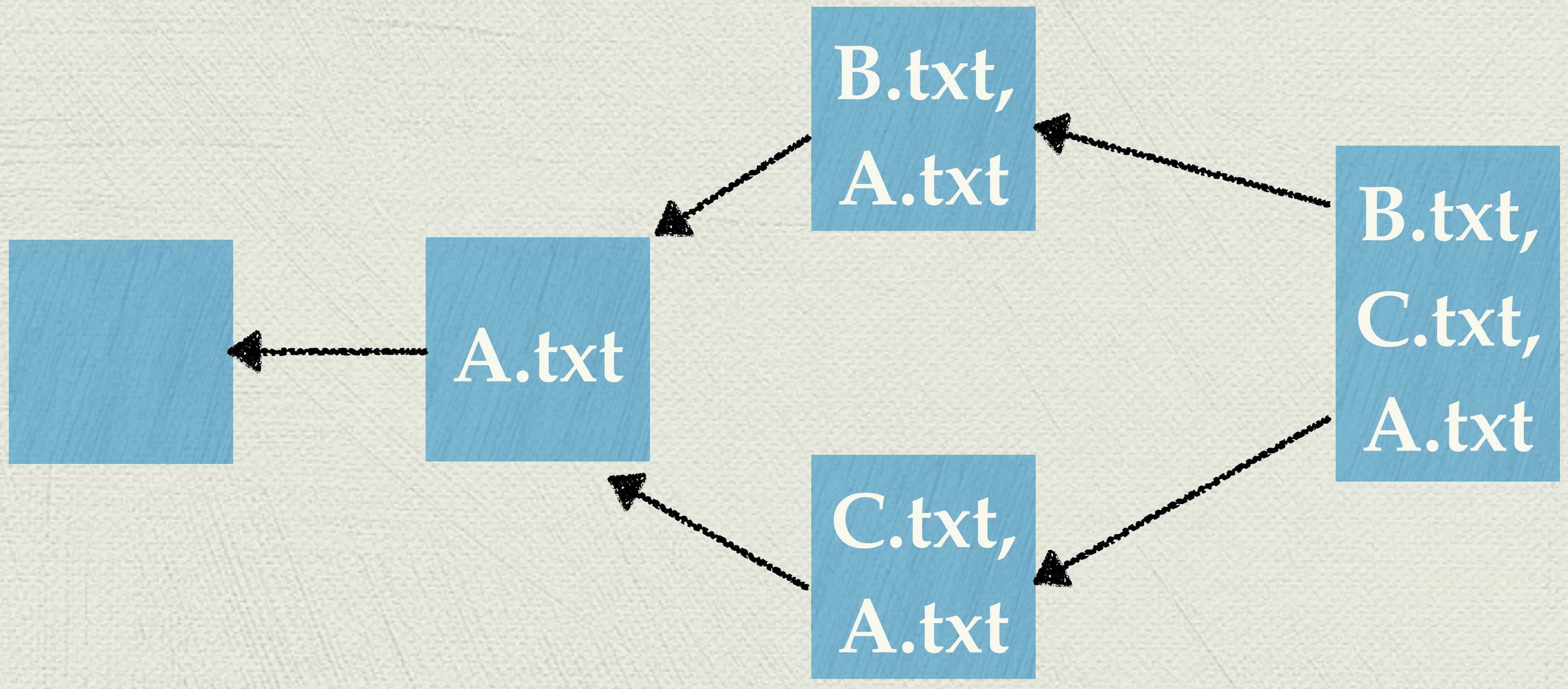
Merge

- When we merge, we want to create a new commit with files from both branches



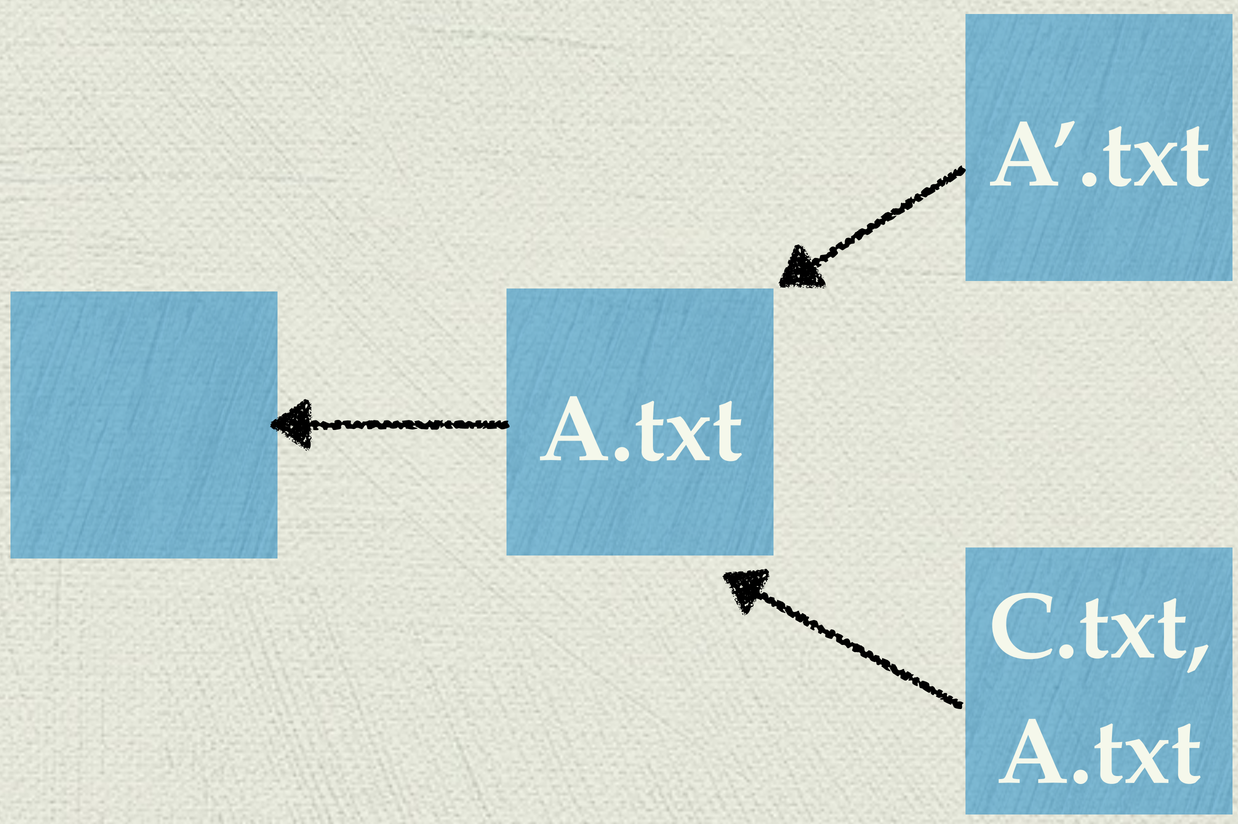
Merge

- When we merge, we want to create a new commit with files from both branches



Merge

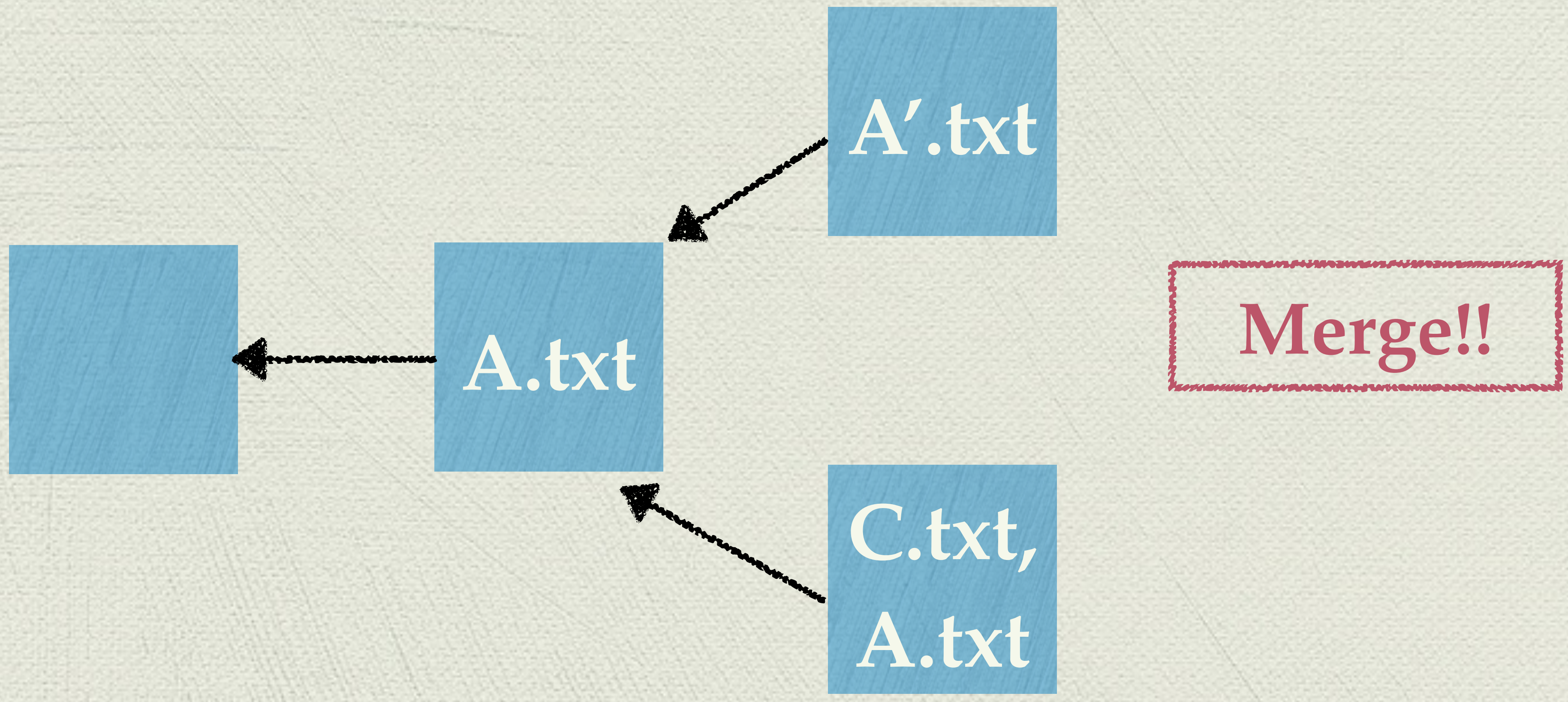
- What if we have different versions of A?



Imagine A′.txt actually has the same name as A.txt. The ′ is just supposed to indicate A is changed.

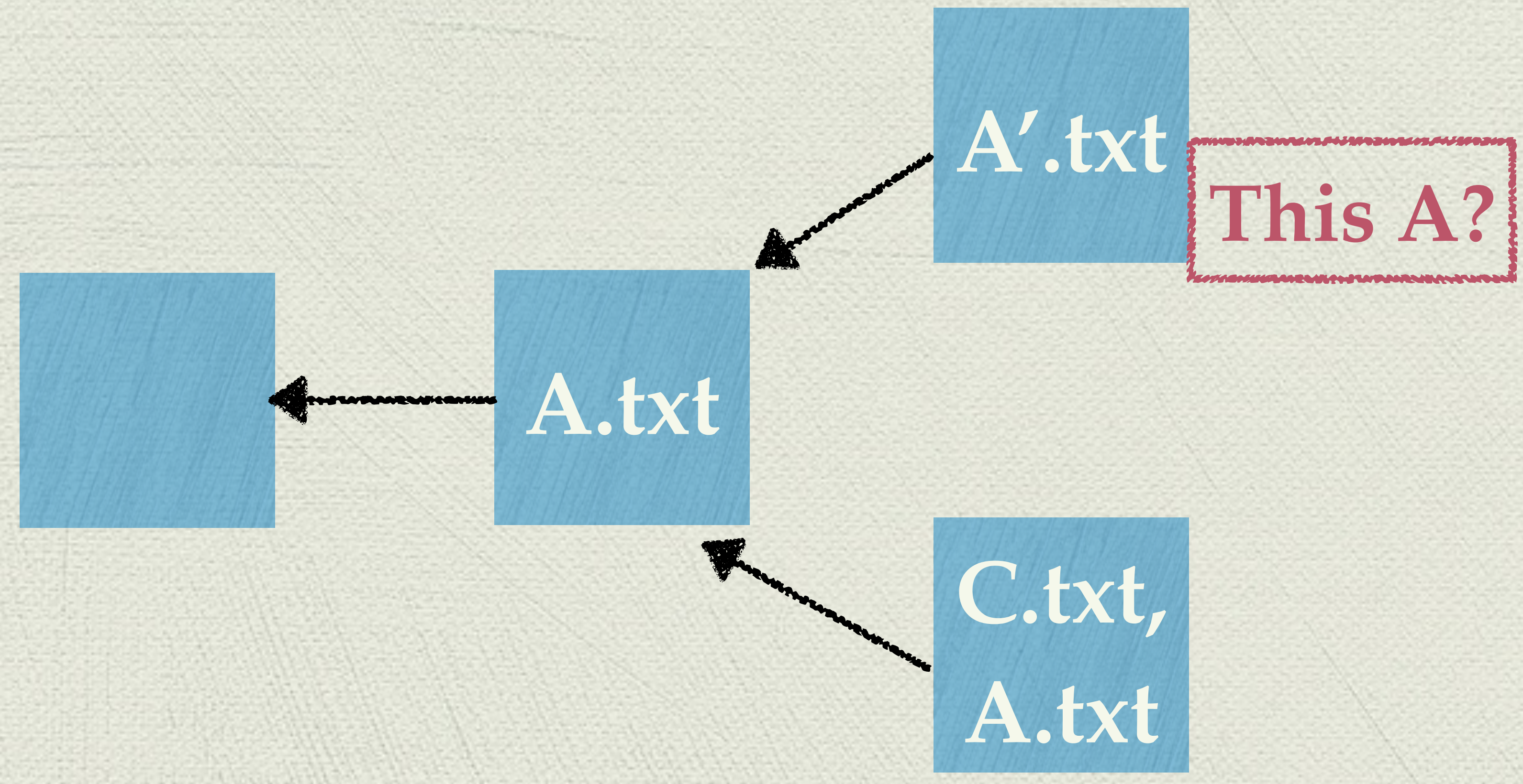
Merge

- What if we have different versions of A?



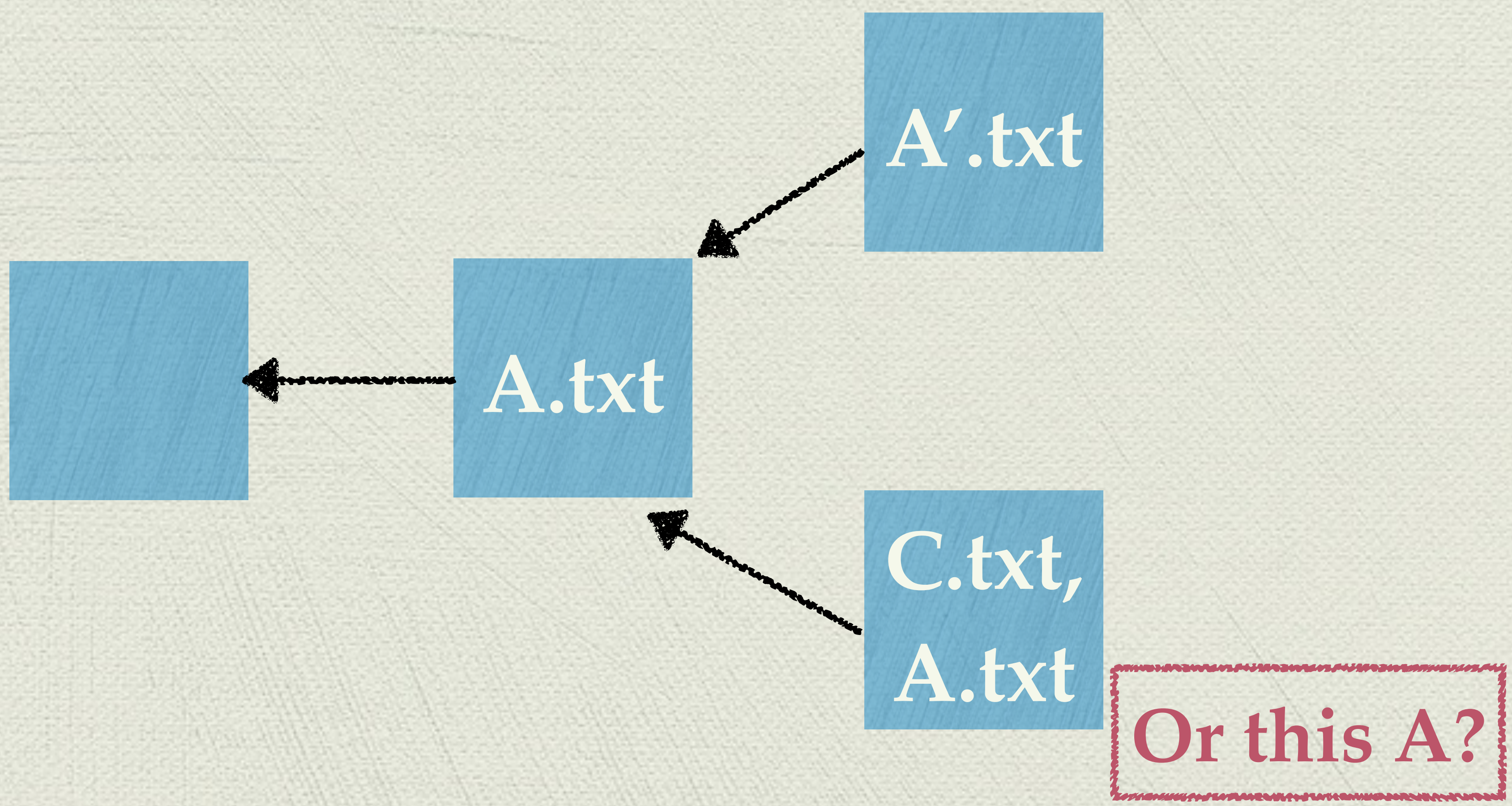
Merge

- Which version of A should we keep in the new commit?



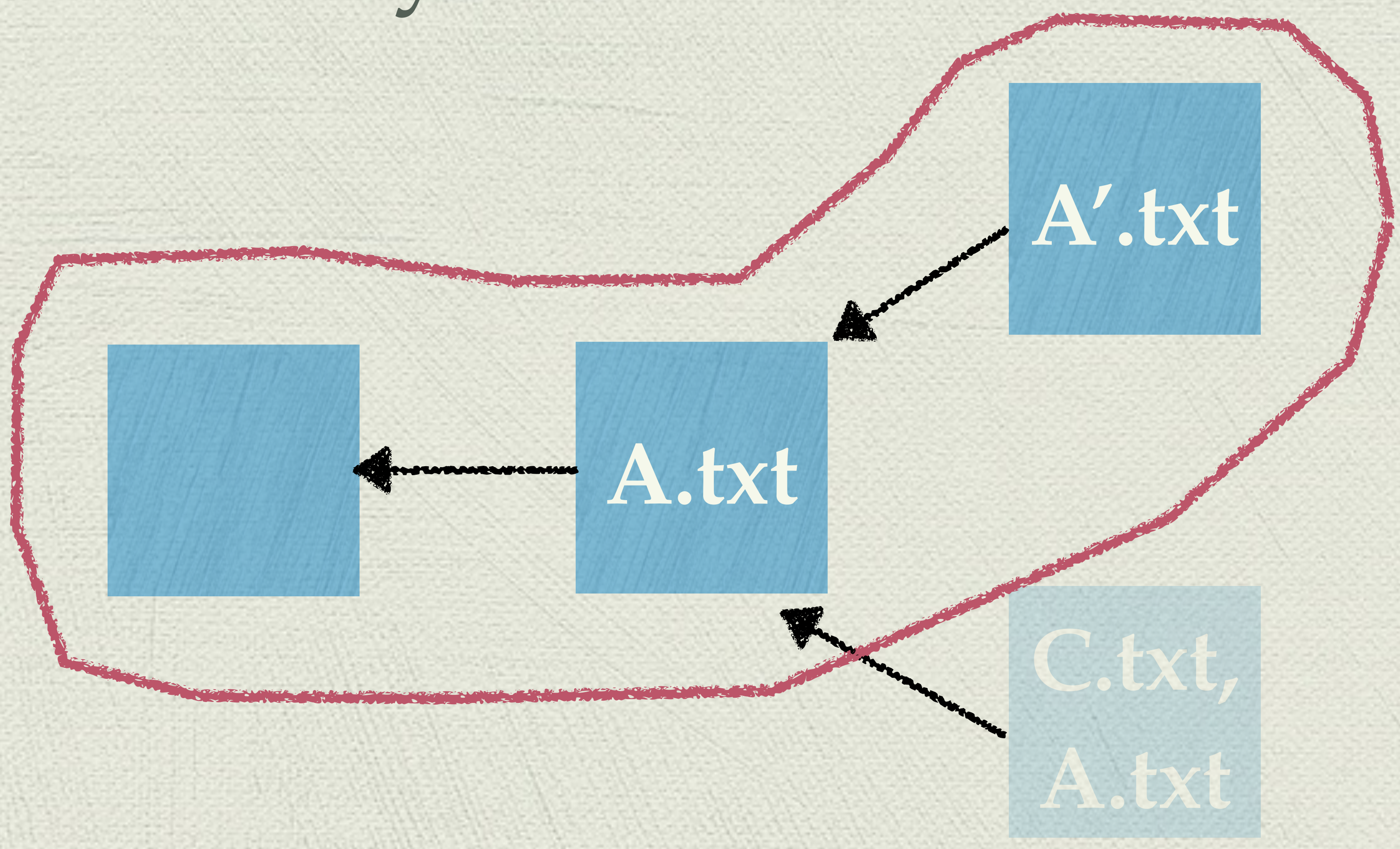
Merge

- Which version of A should we keep in the new commit?



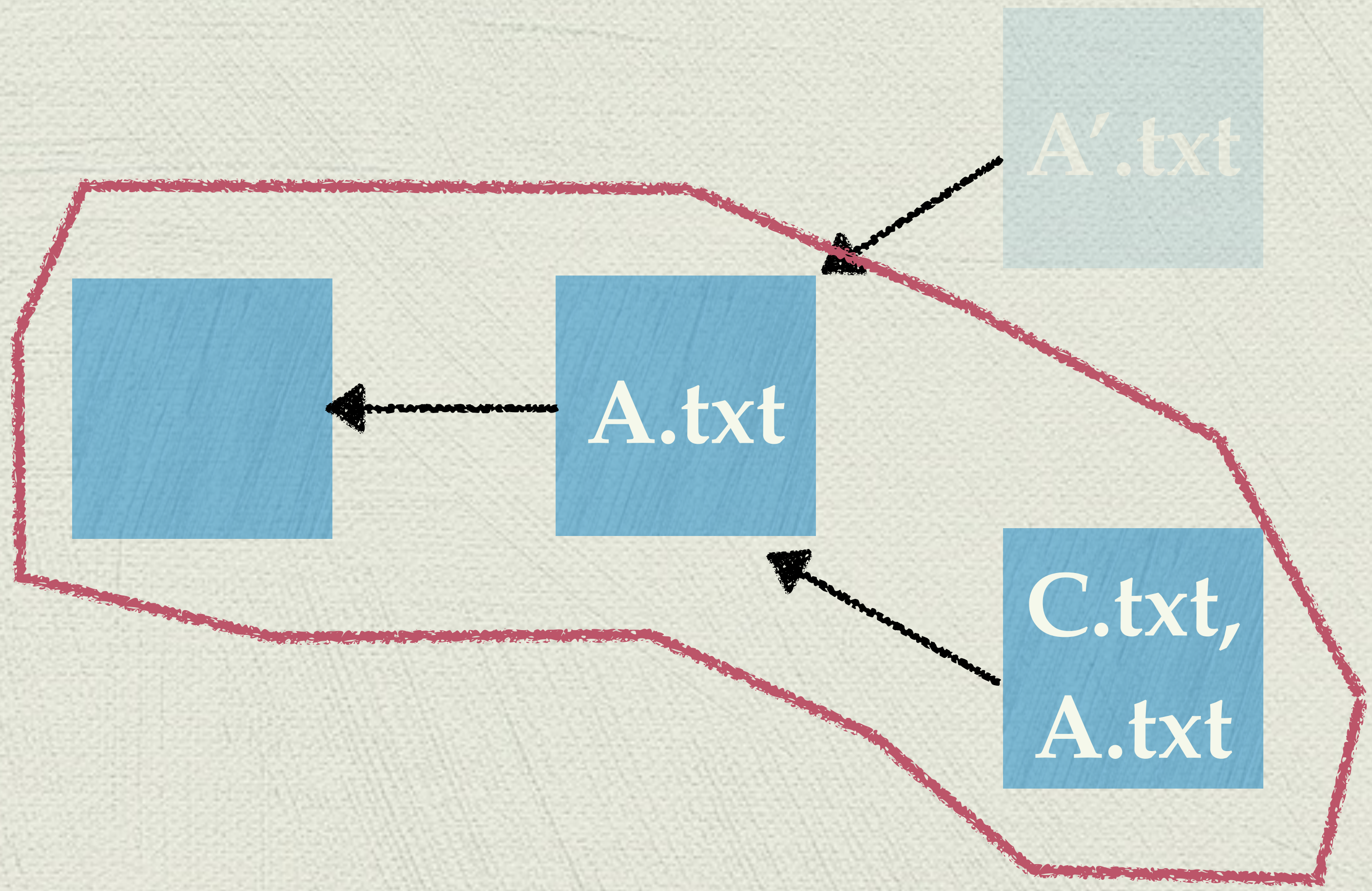
Merge

- Ignoring the other branch, we see that A′ is a strictly newer version than A



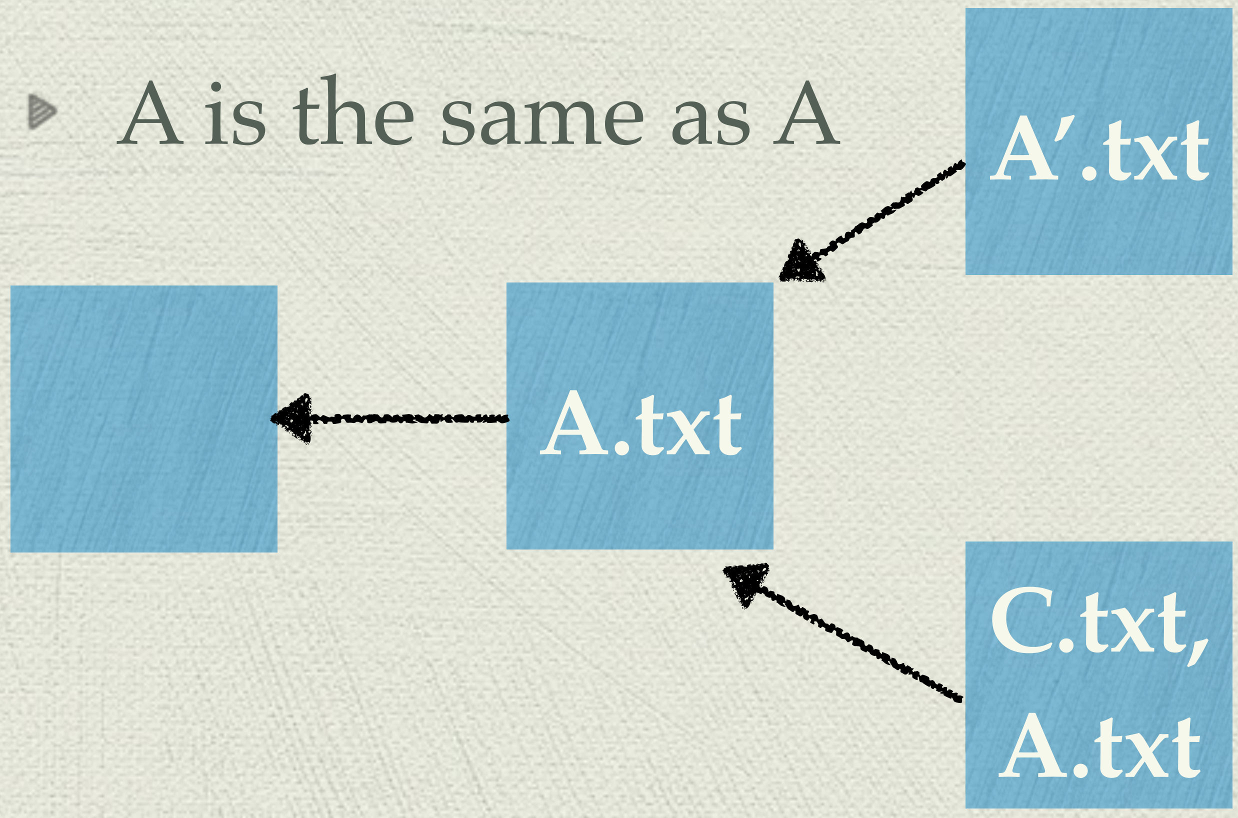
Merge

- But this A is the same A



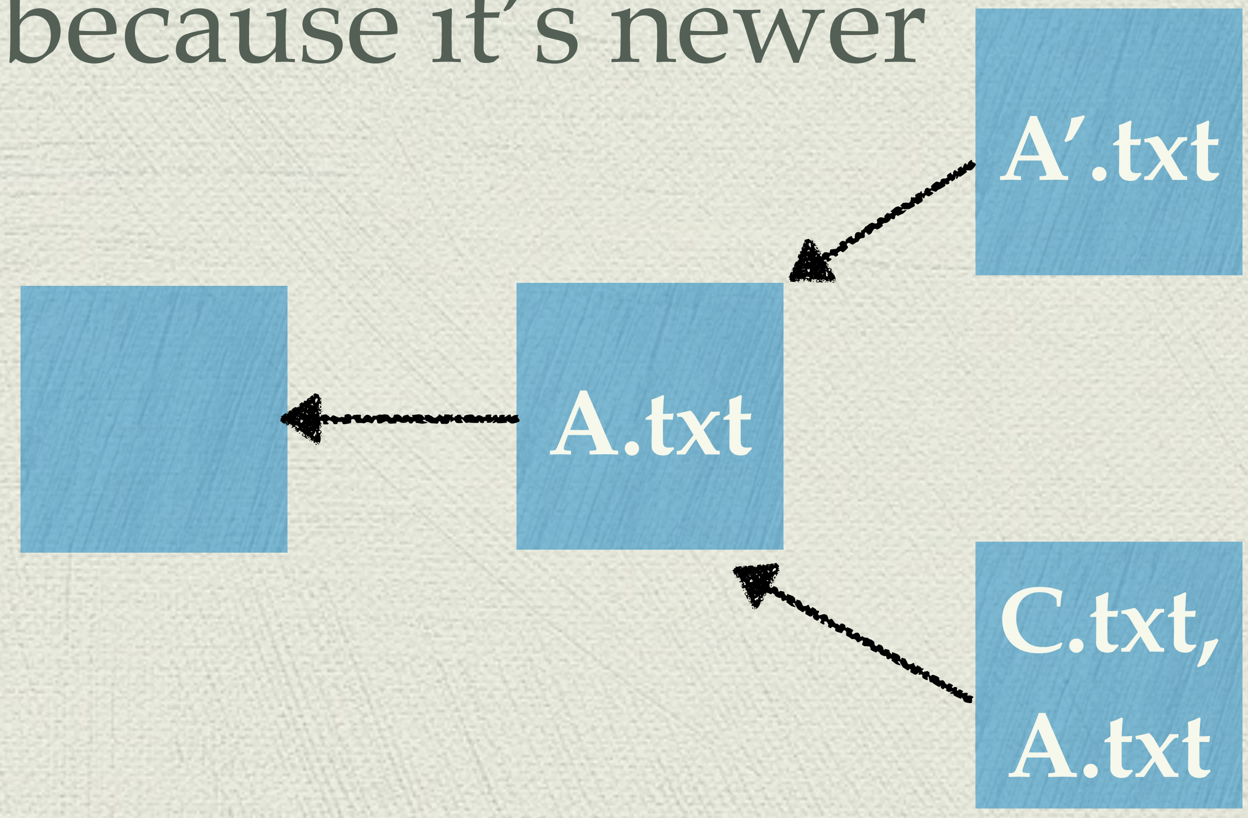
Merge

- Facts:
  - A′ is newer than A
  - A is the same as A

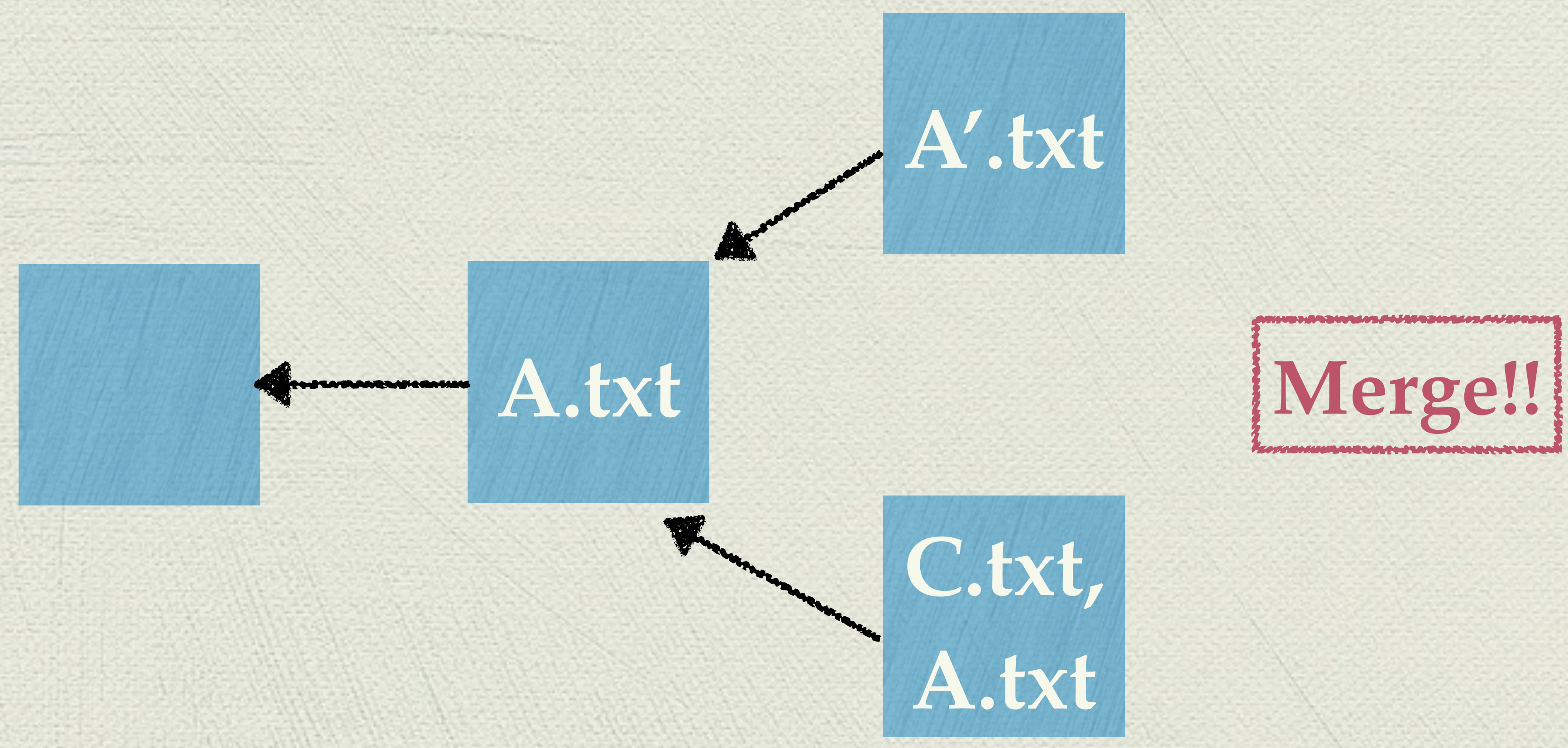


Merge

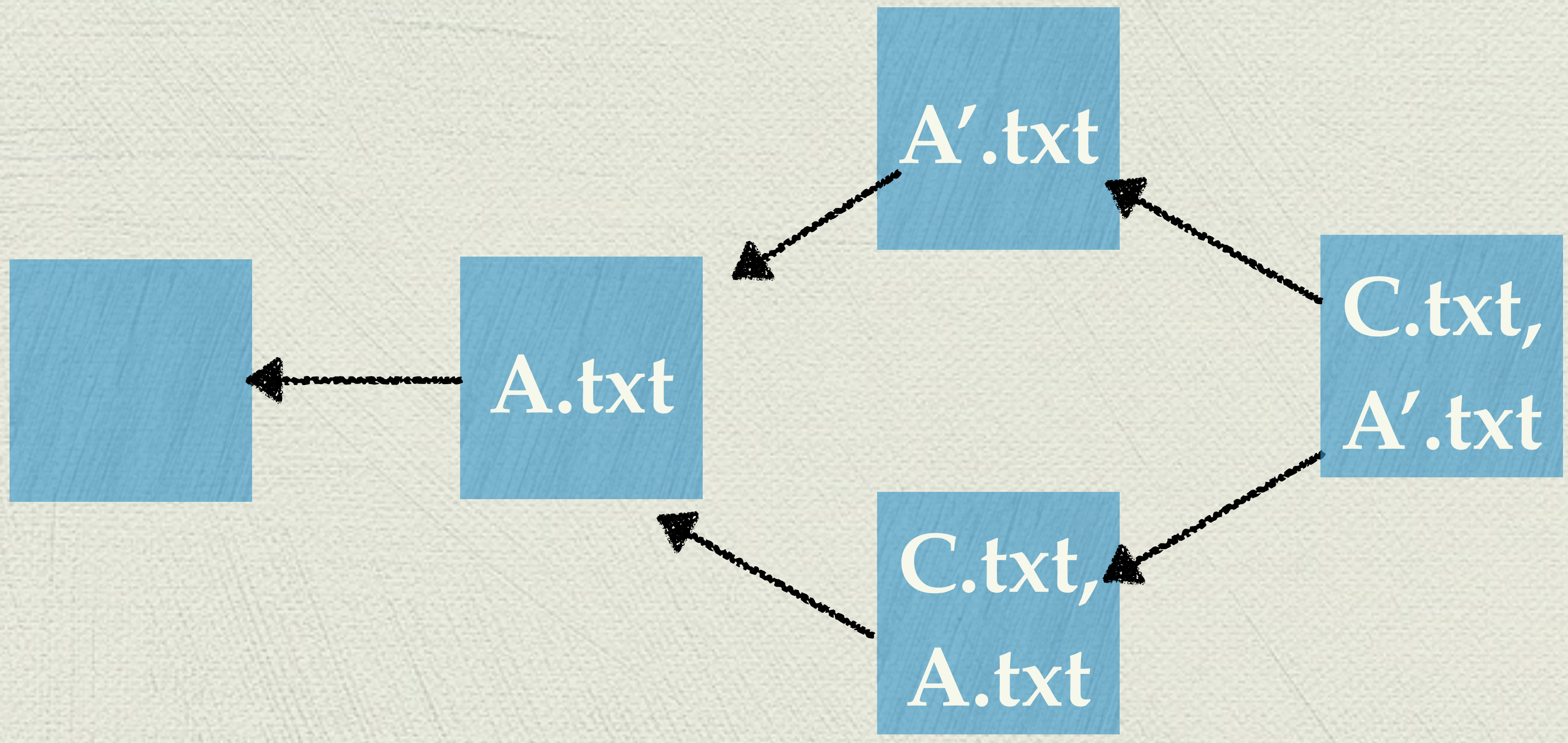
- So when we merge, and we have to decide whether to keep A′ or A, we should keep A′, because it′s newer



Merge

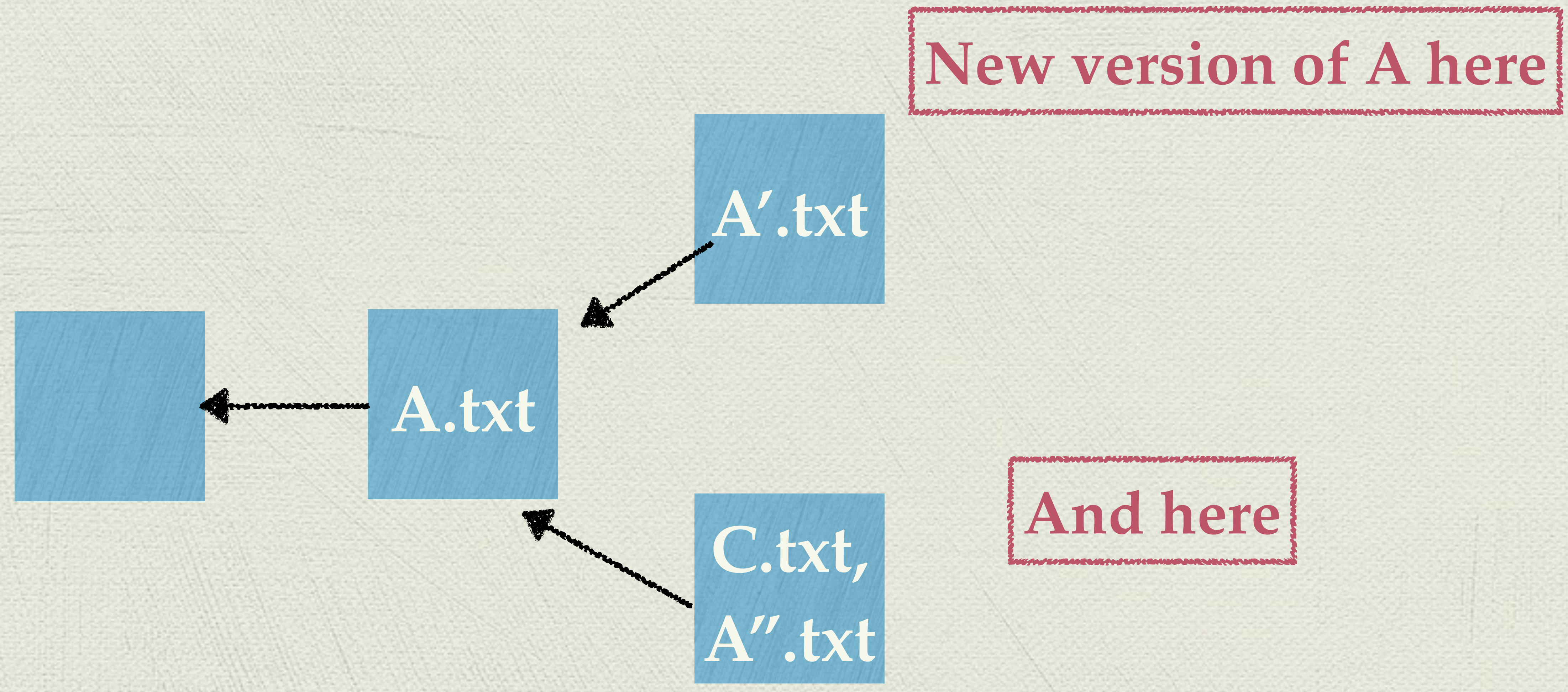


Merge



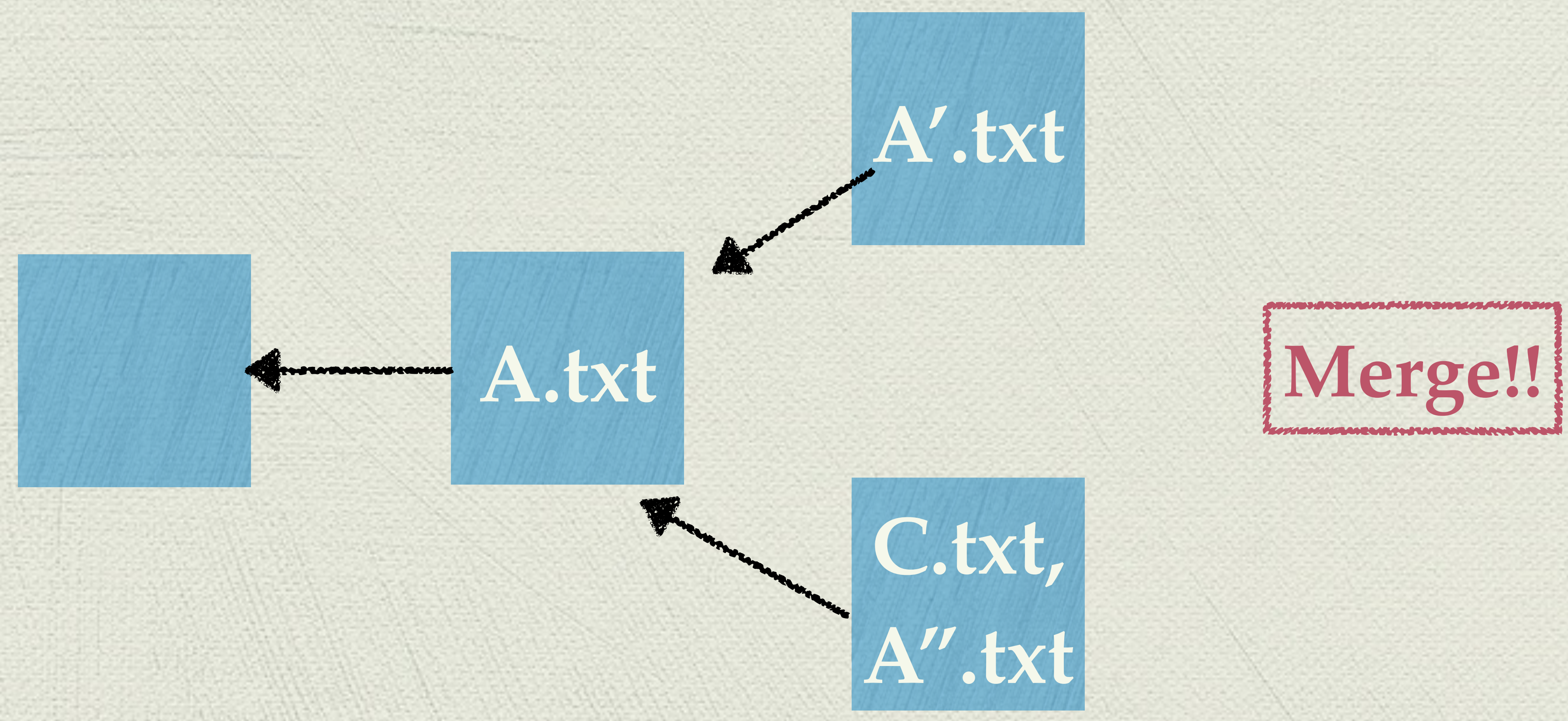
Merge

- New scenario



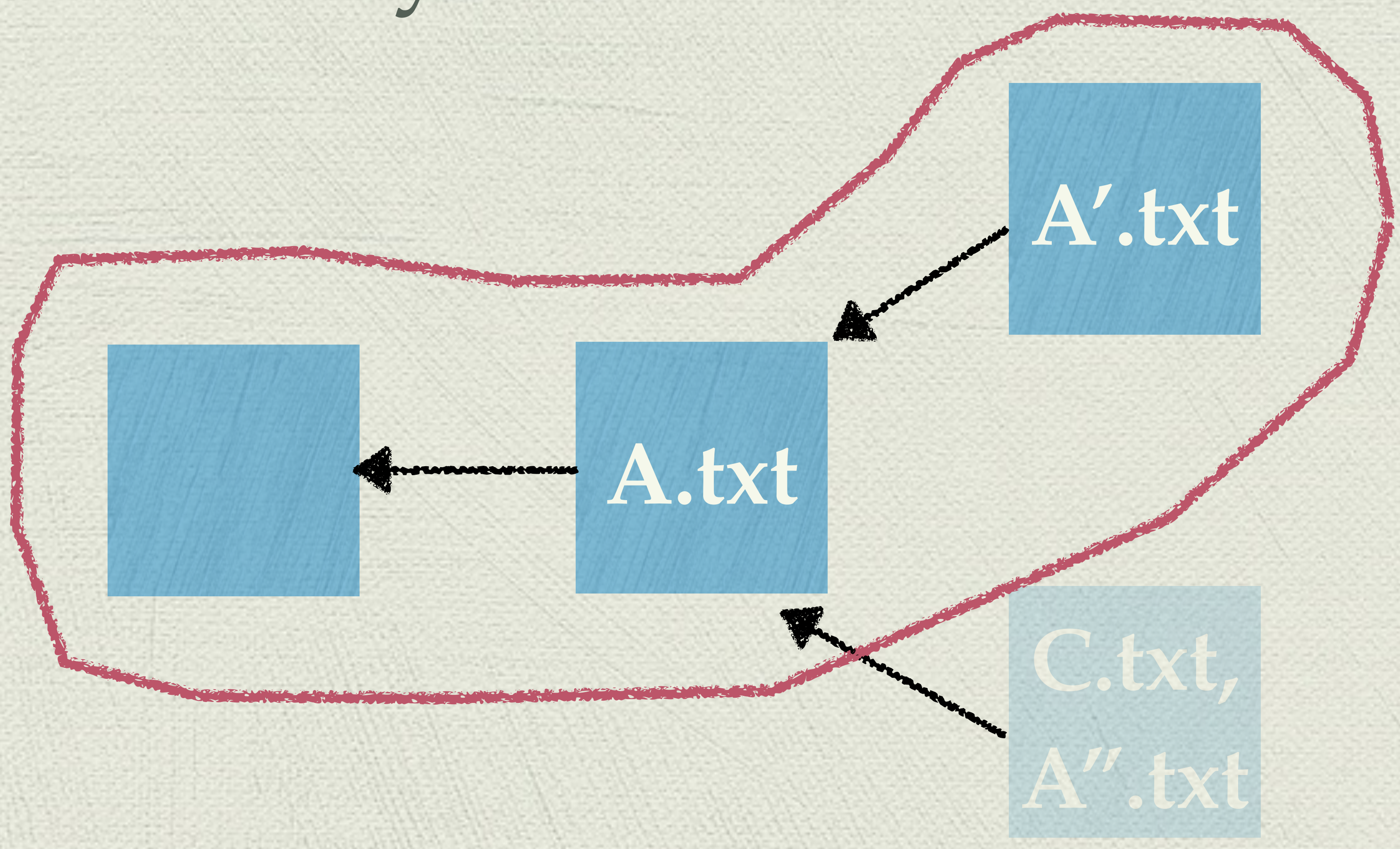
Merge

- Which one do we keep now?



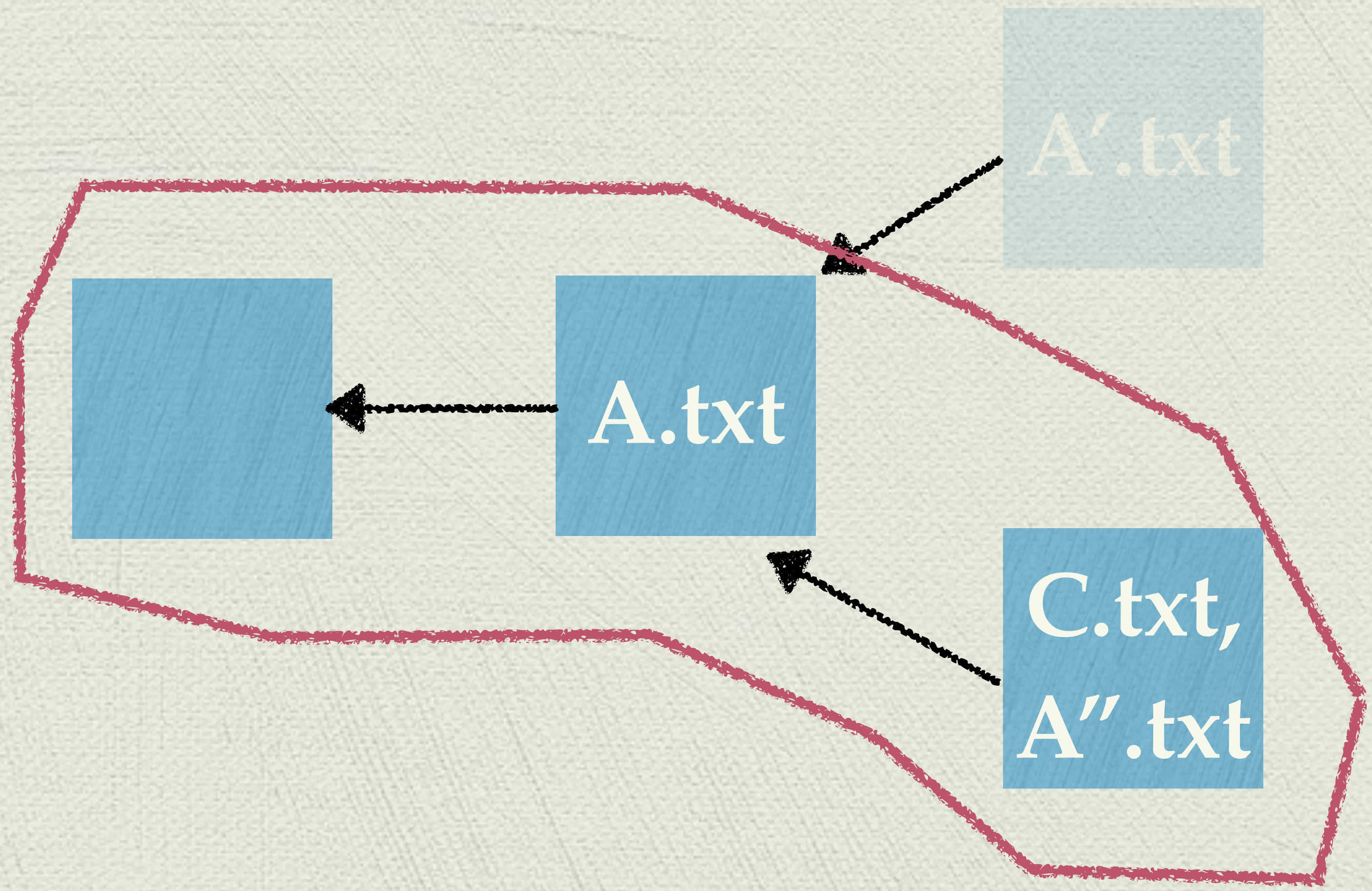
Merge

- Ignoring the other branch, we see that A′ is a strictly newer version than A



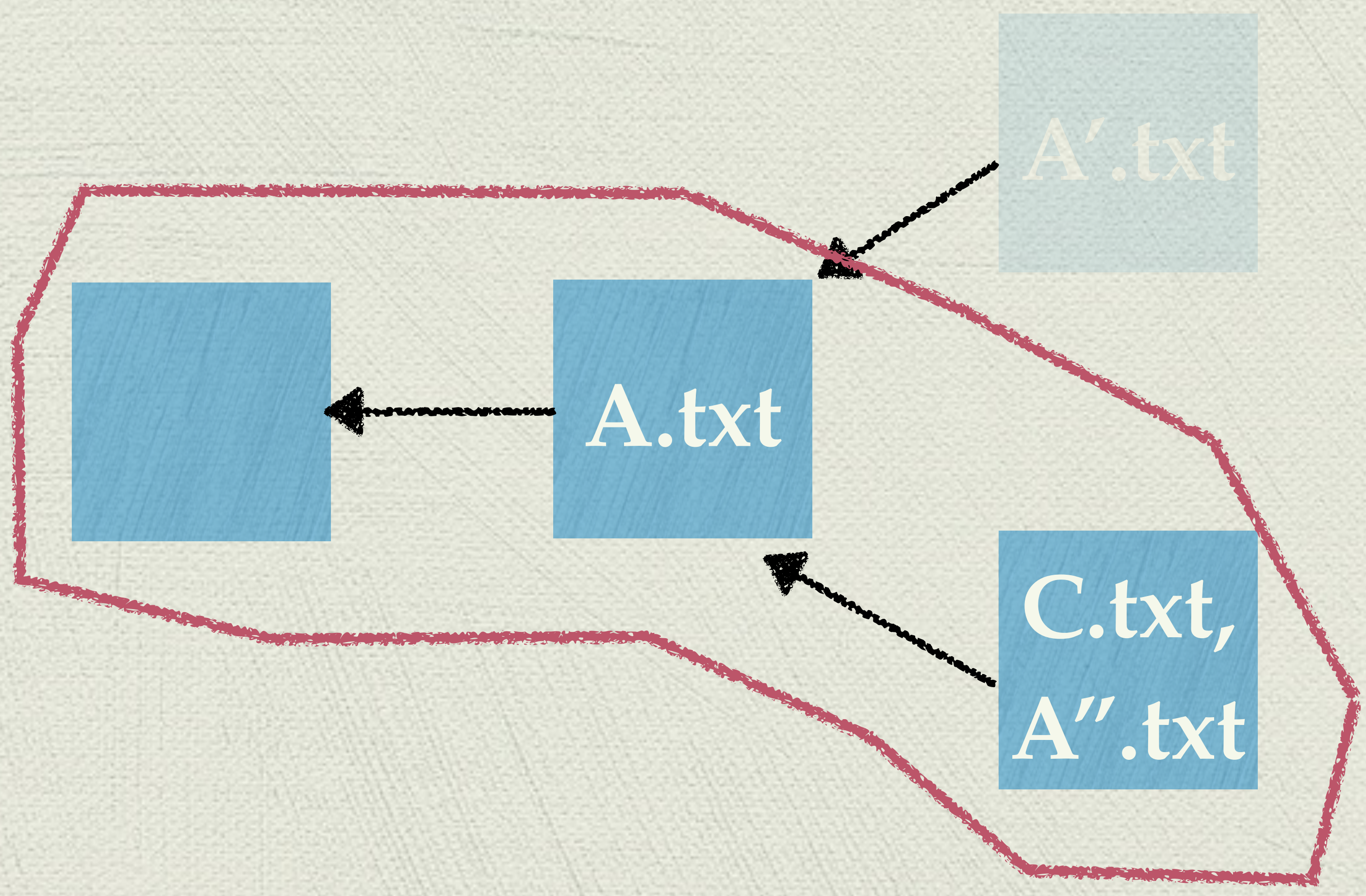
Merge

- Ignoring the other branch, we see A″ is a newer version than A



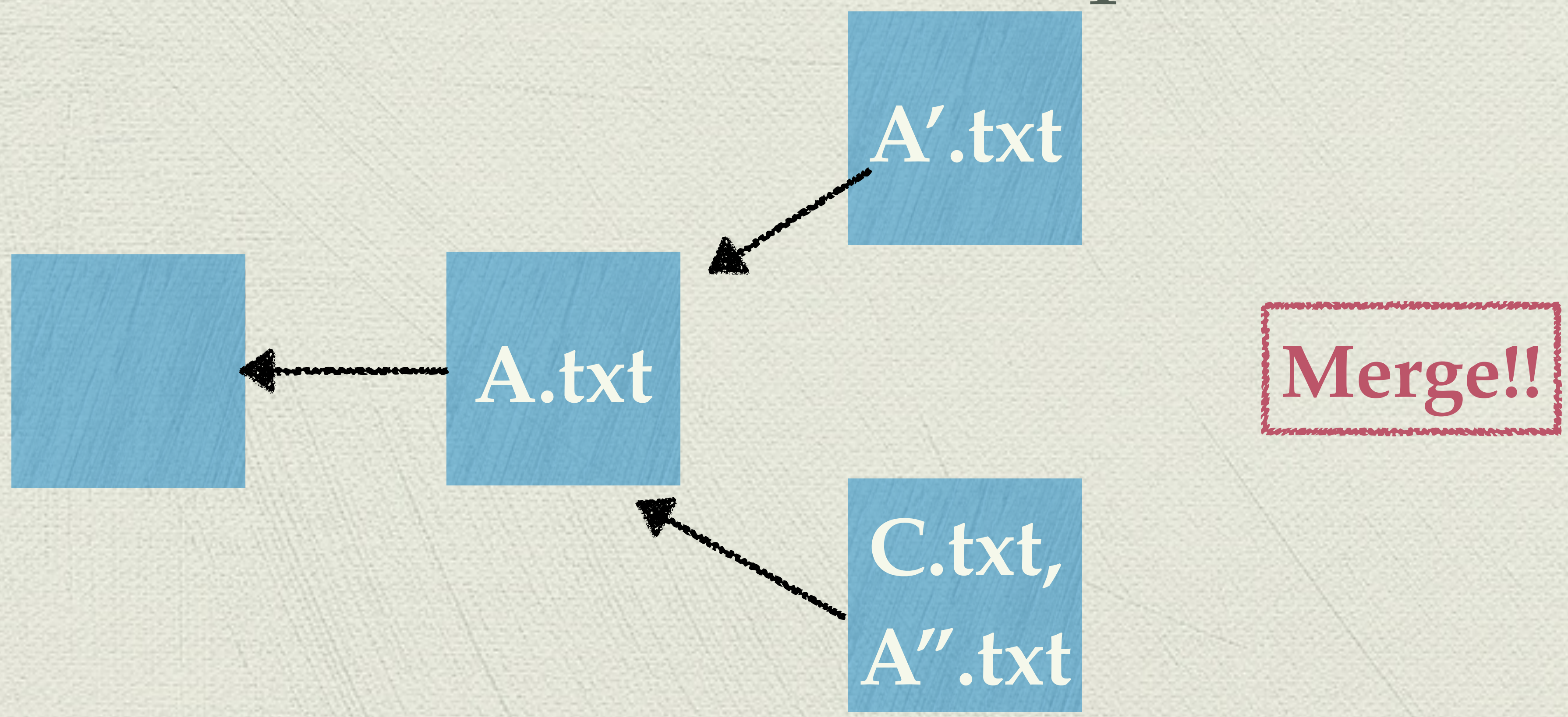
Merge

- But A″ doesn′t know anything about A′. It is not obviously newer than A′



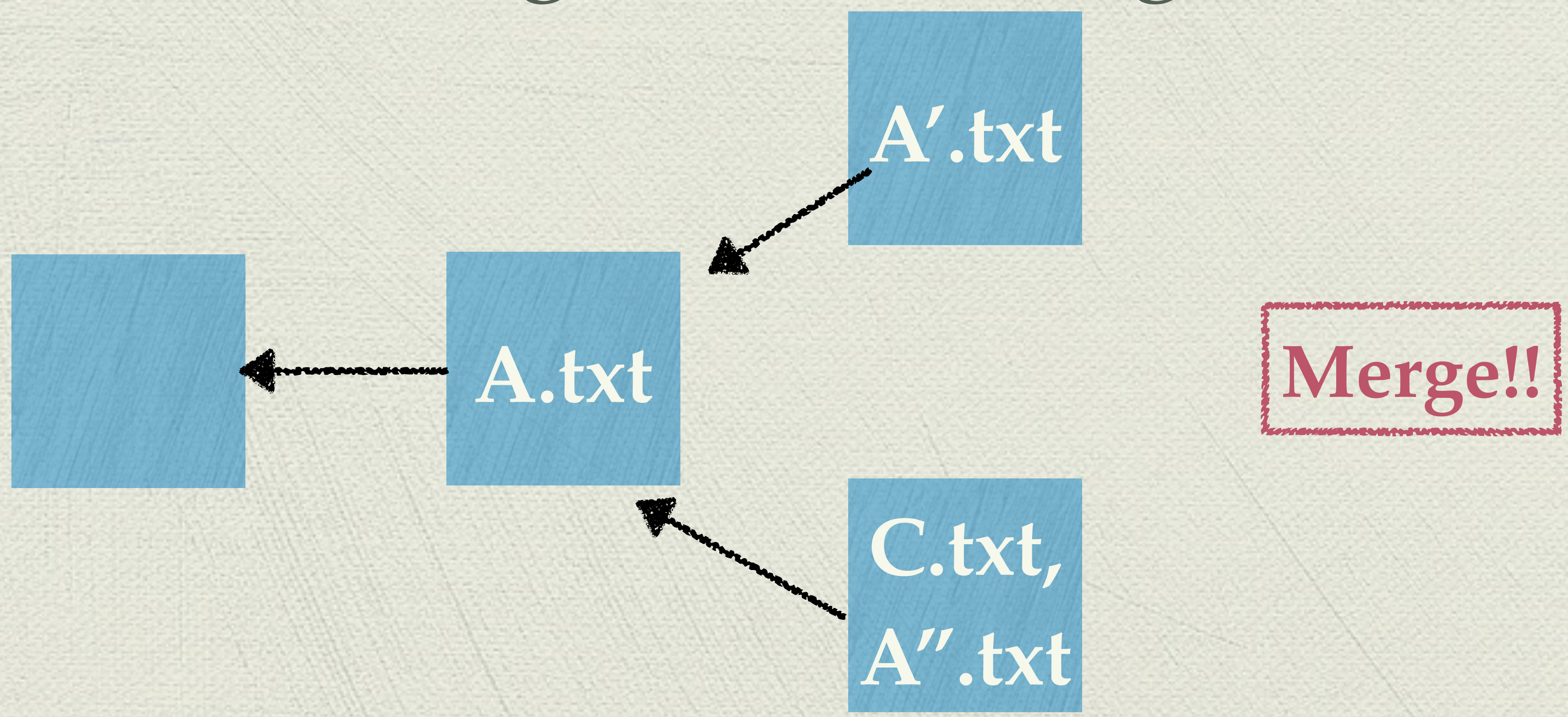
Merge

- Facts:
  - A′ is newer than A
  - A″ is newer than A
  - No known relationship between A′ and A″

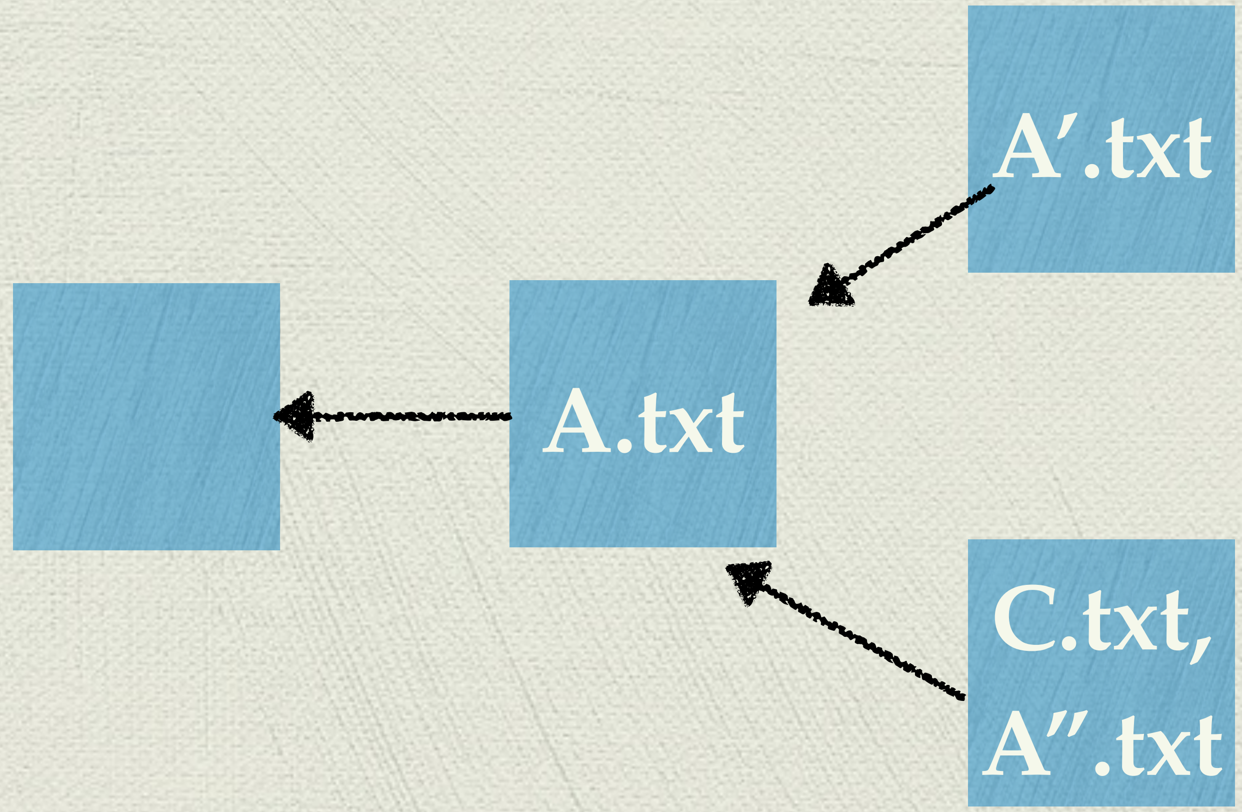


Merge

- Therefore, it's not clear which of A′ or A″ we should keep in the merged commit
- So gitlet will let the user decide manually rather than gitlet deciding automatically



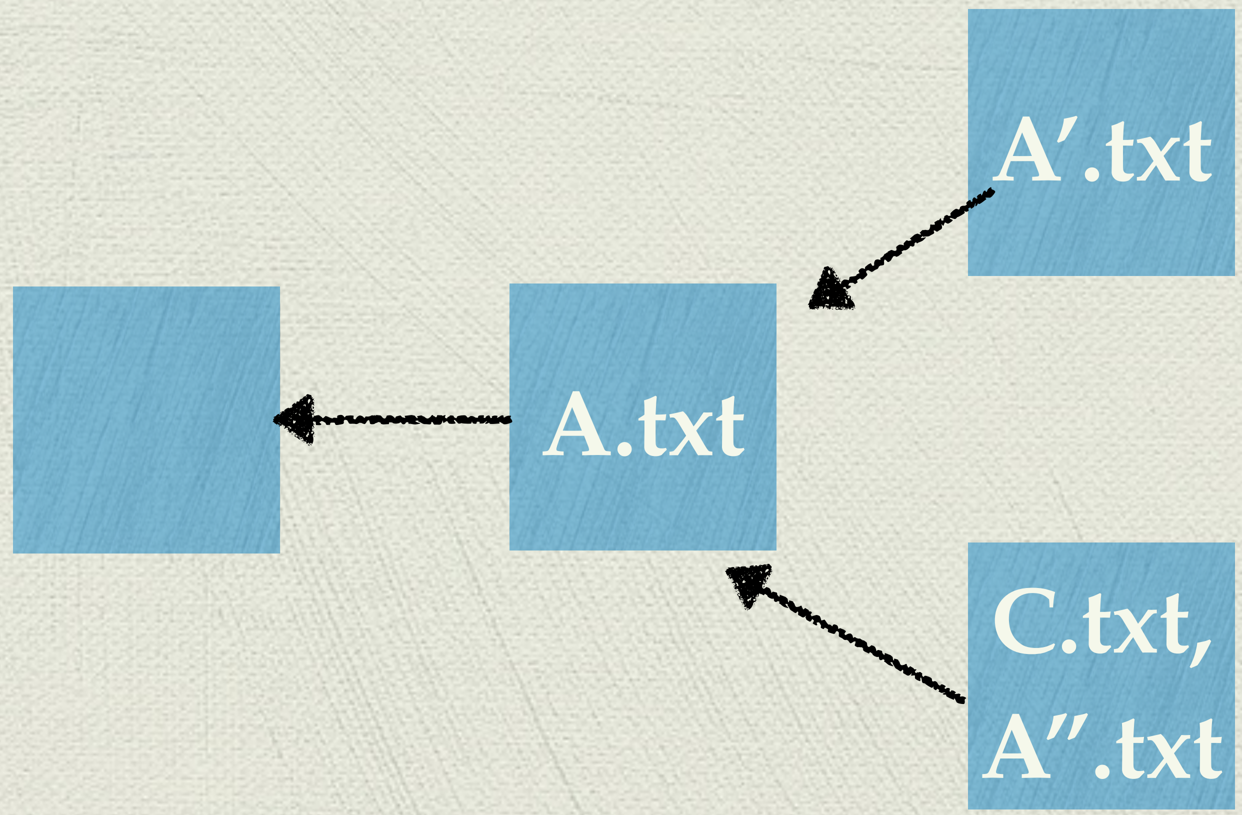
Merge



Merge commit will not happen temporarily.

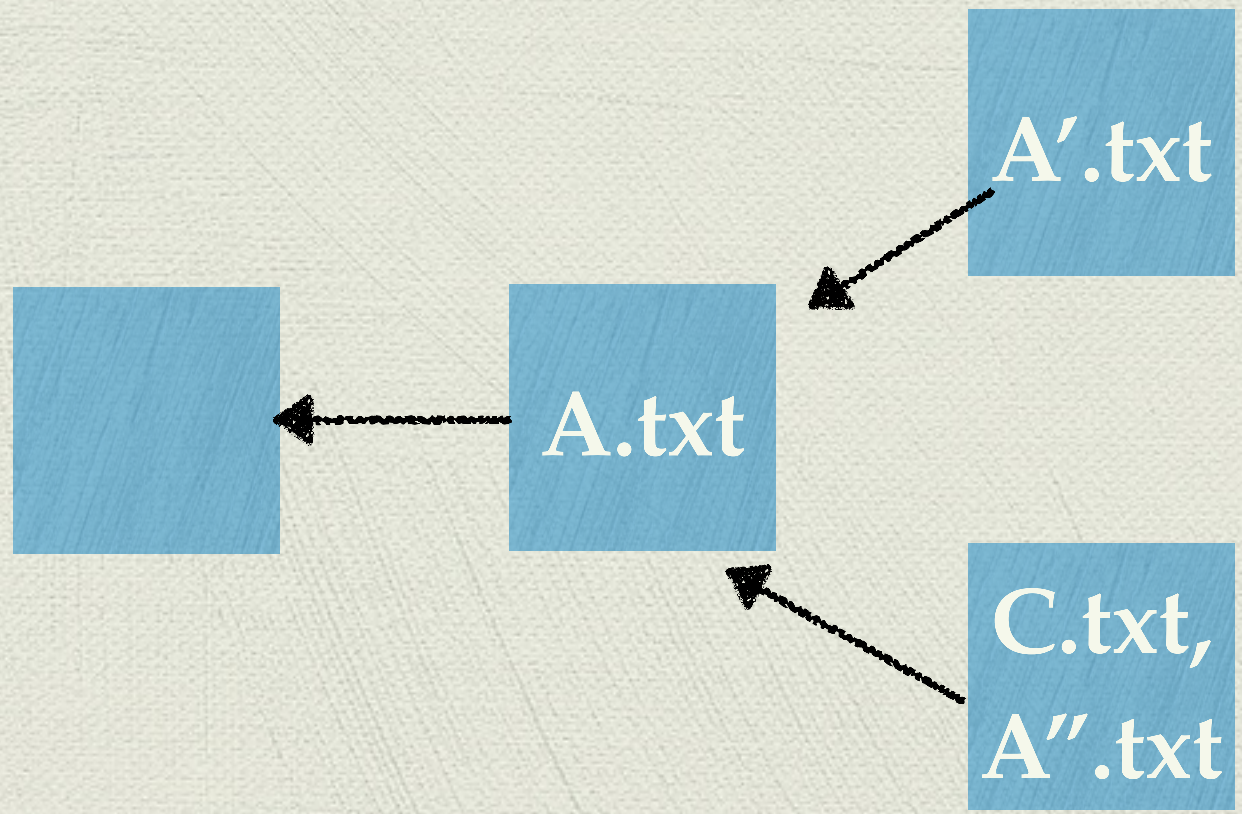
Merge

- Instead, both A′ and A″ appear in the working directory. One has name A.txt, the other has name A.txt.conflicted



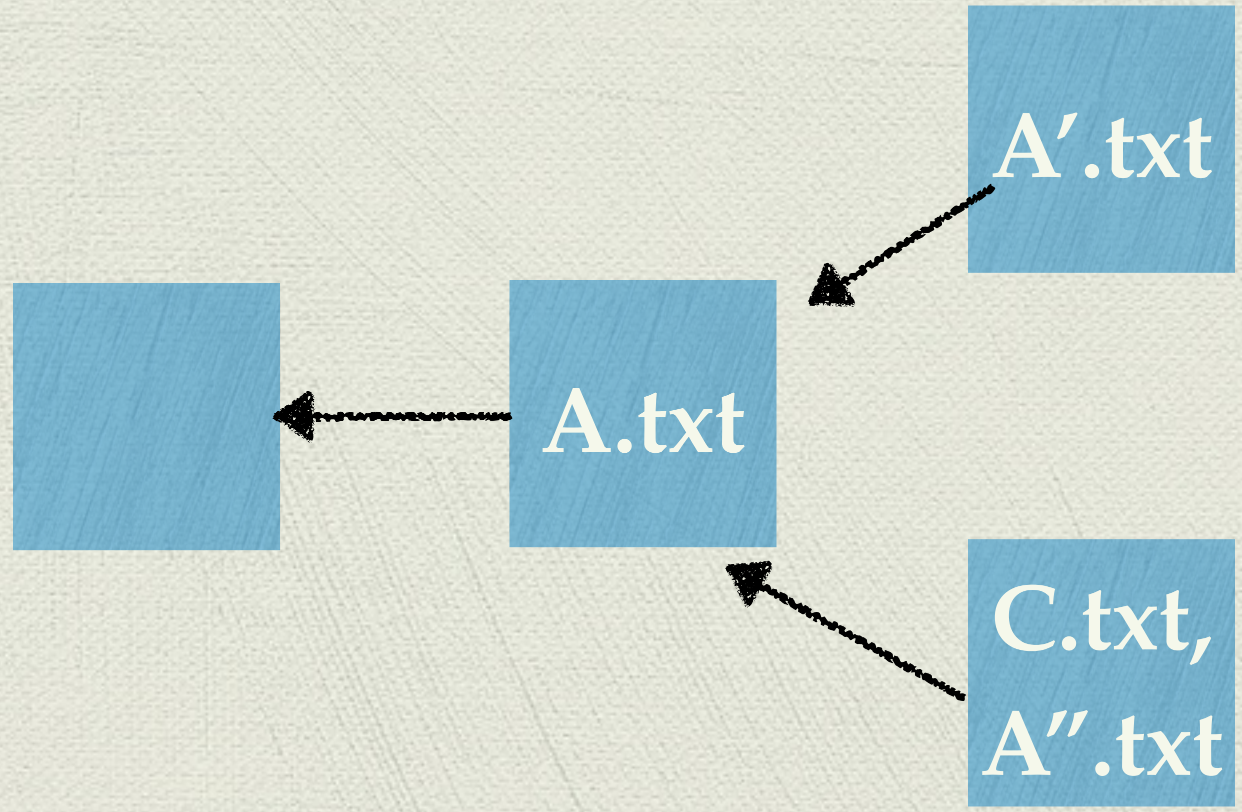
Merge

- User decides which one they want, deletes the other, and makes sure the name of the one they want is A.txt



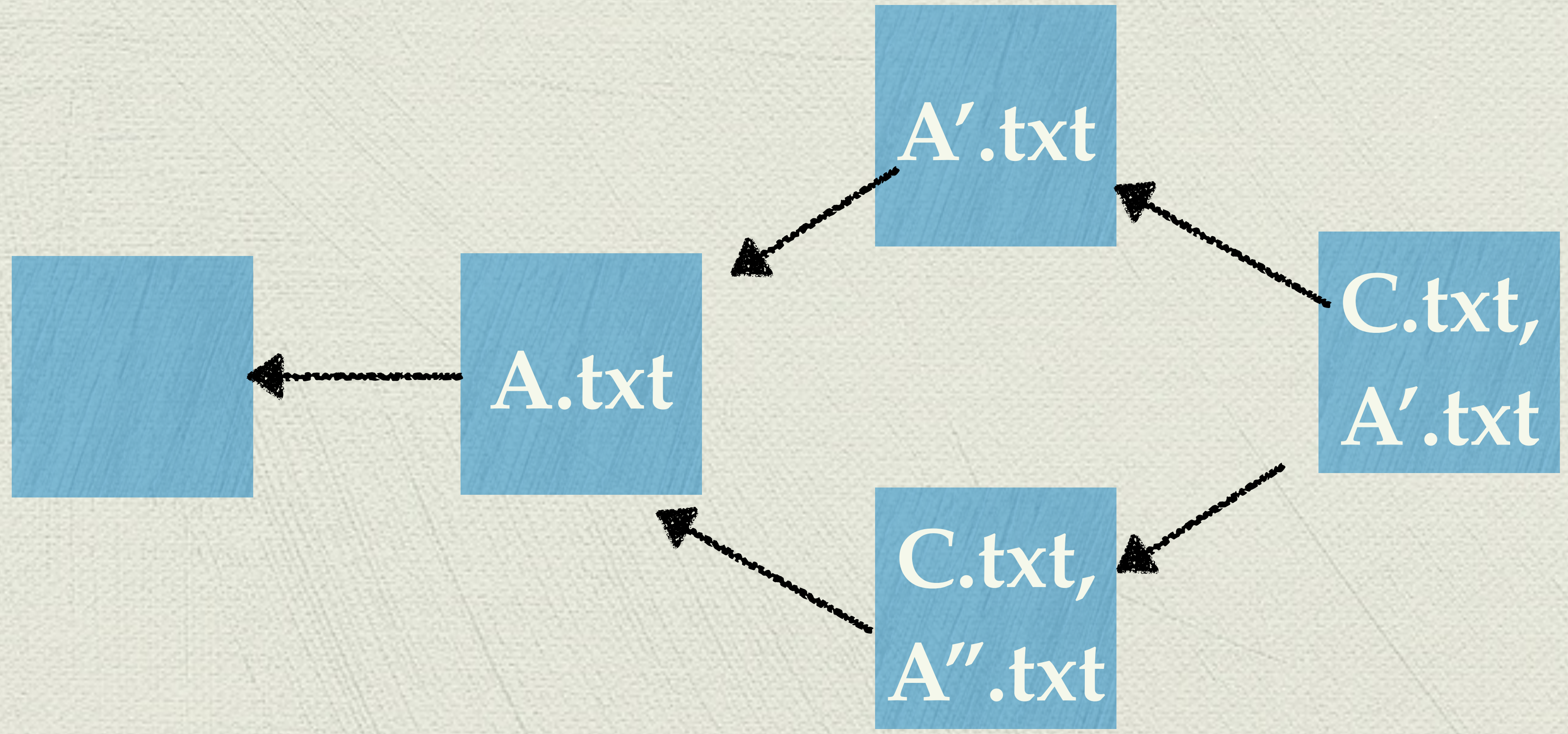
Merge

- Say user decides they wanted A′.txt. So they delete A″.txt, then add A′.txt, then commit.



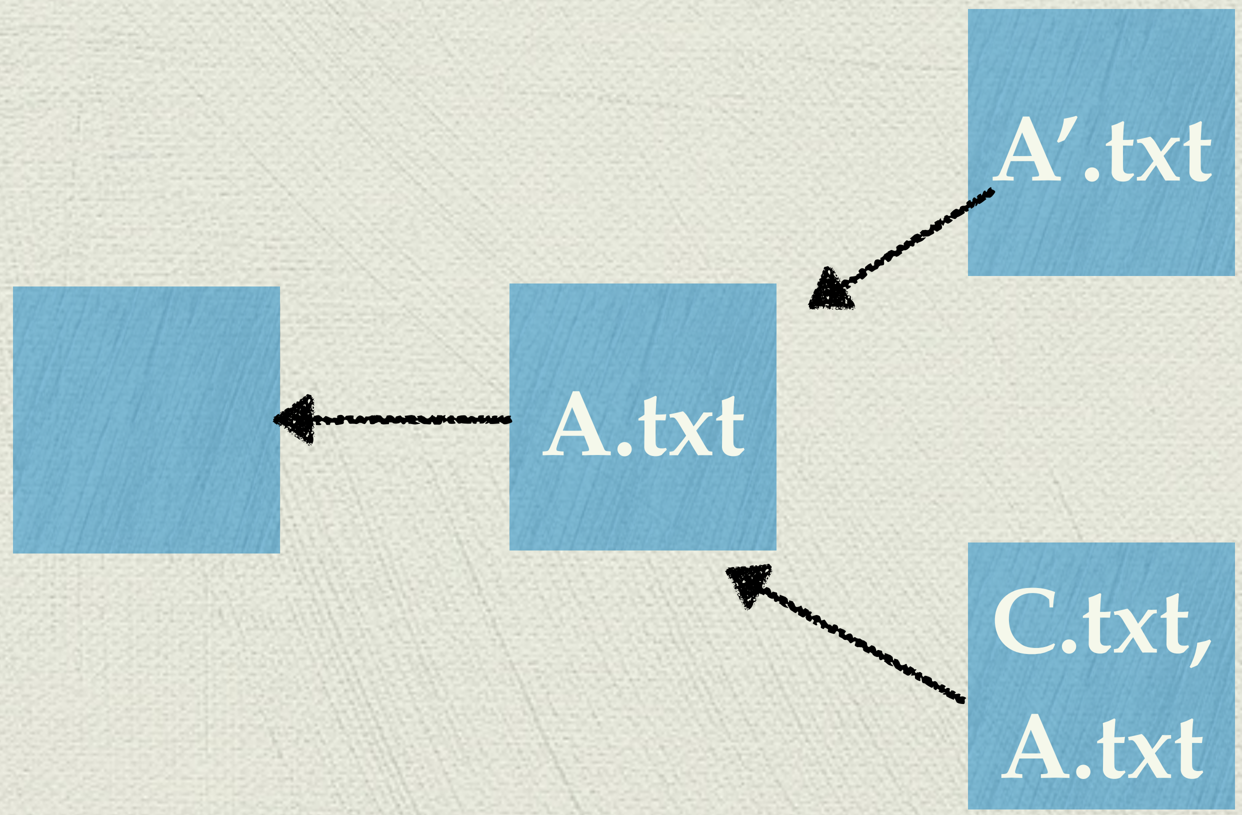
Merge

- Merge complete

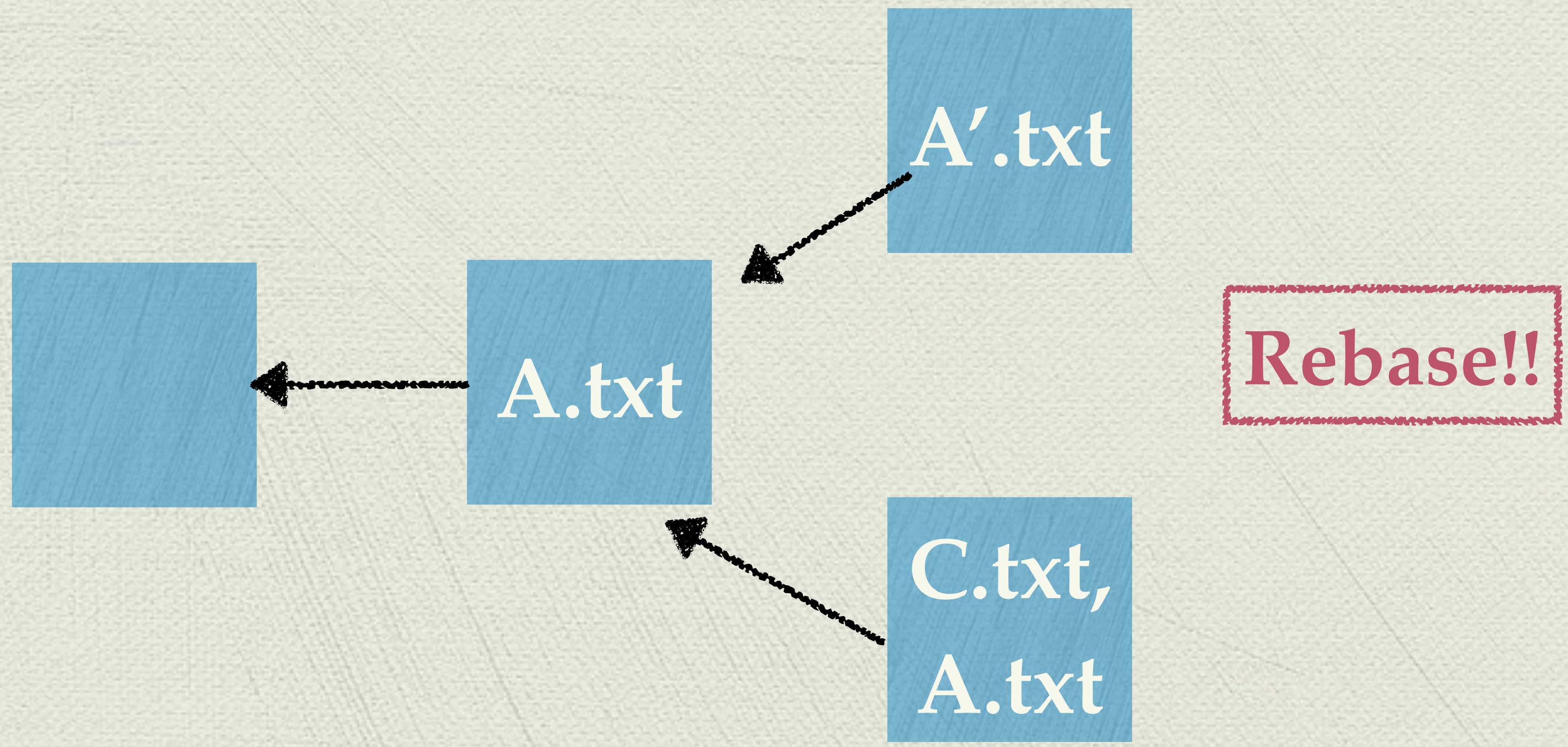


Rebase

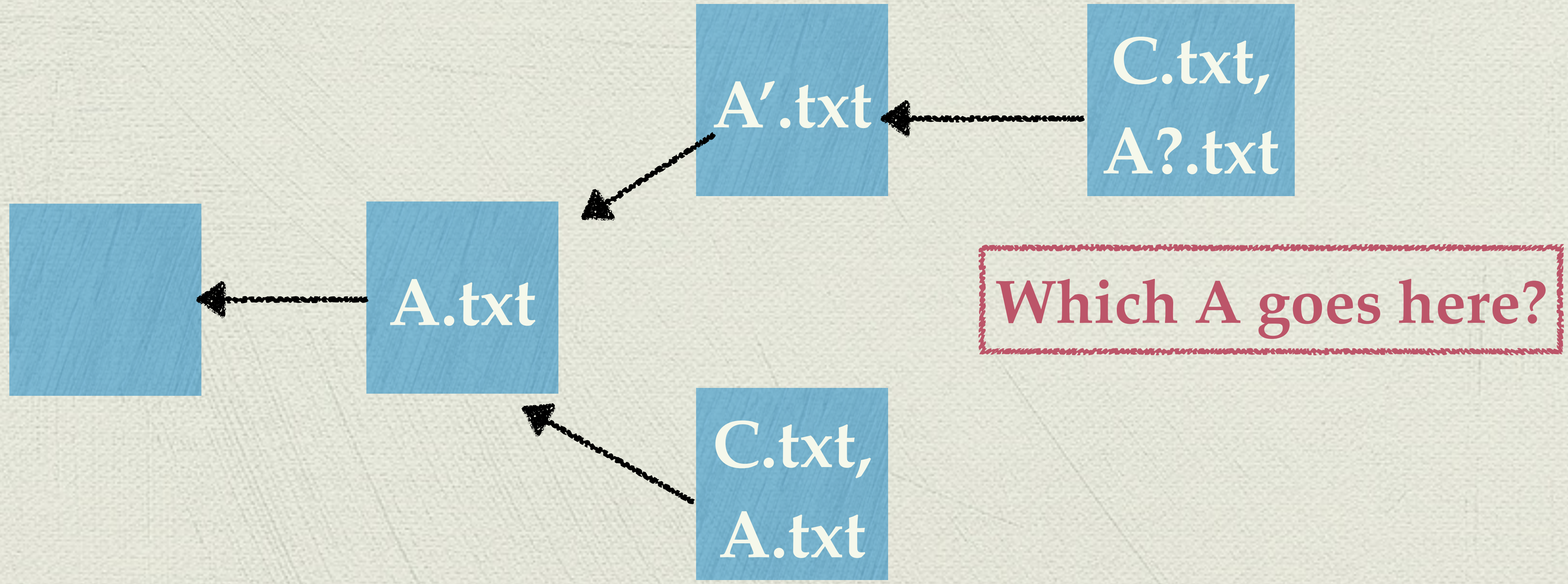
- Follows almost the exact same logic as merge



Rebase

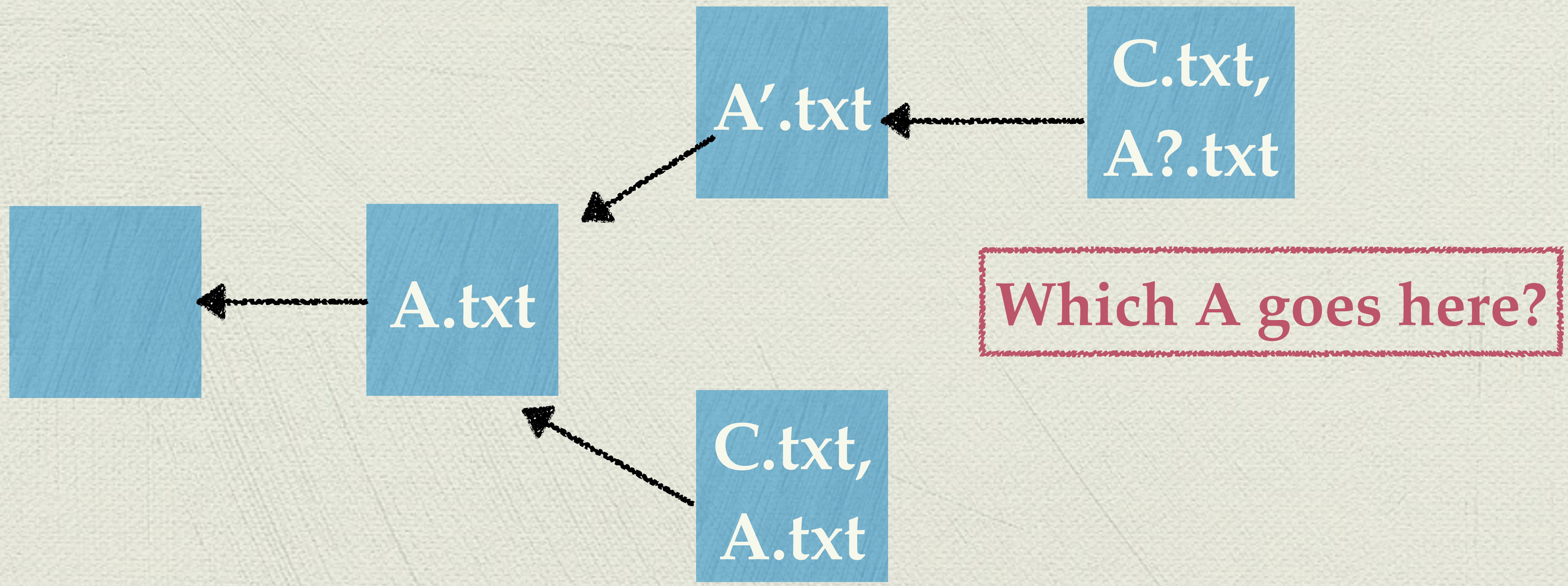


Rebase



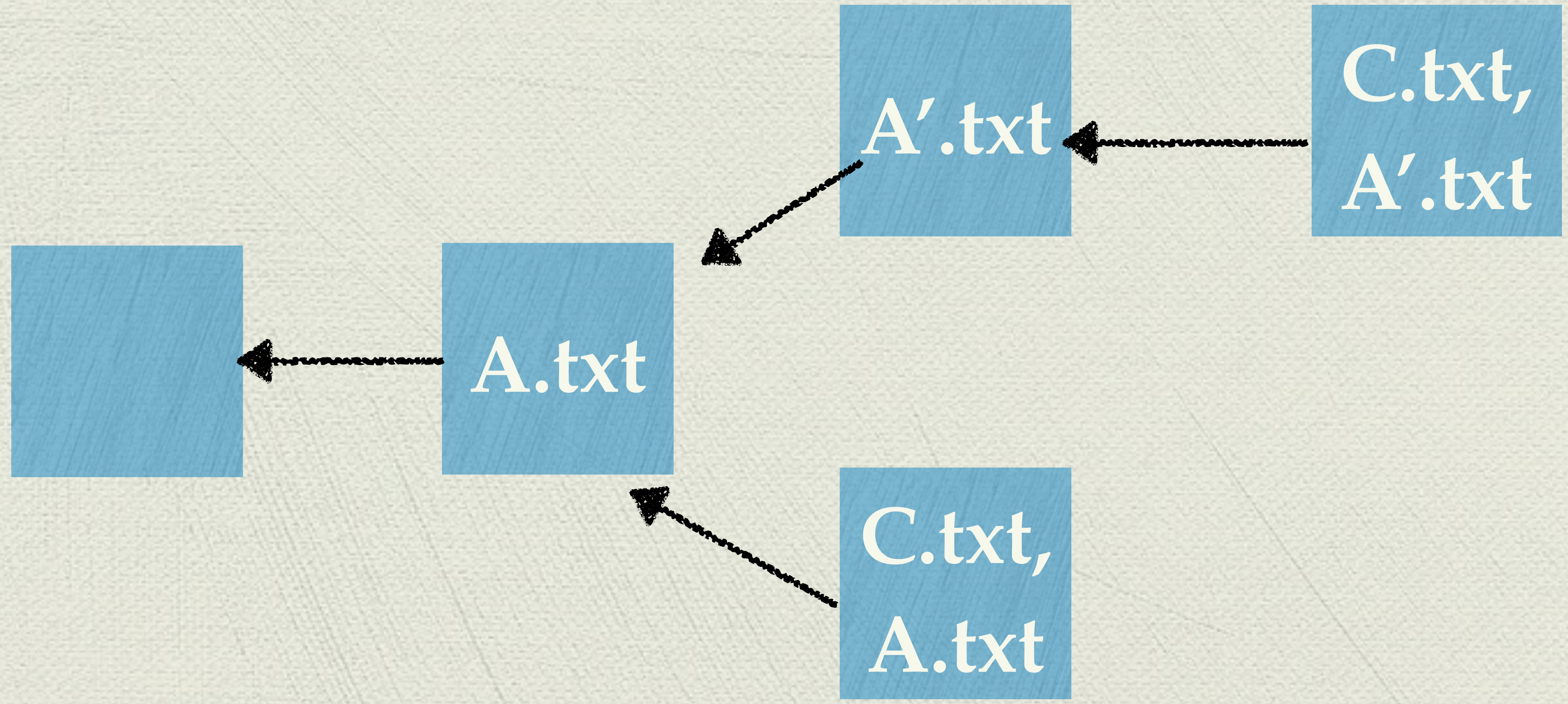
Rebase

- Facts are the same: A′ is newer than A, and A is the same as A



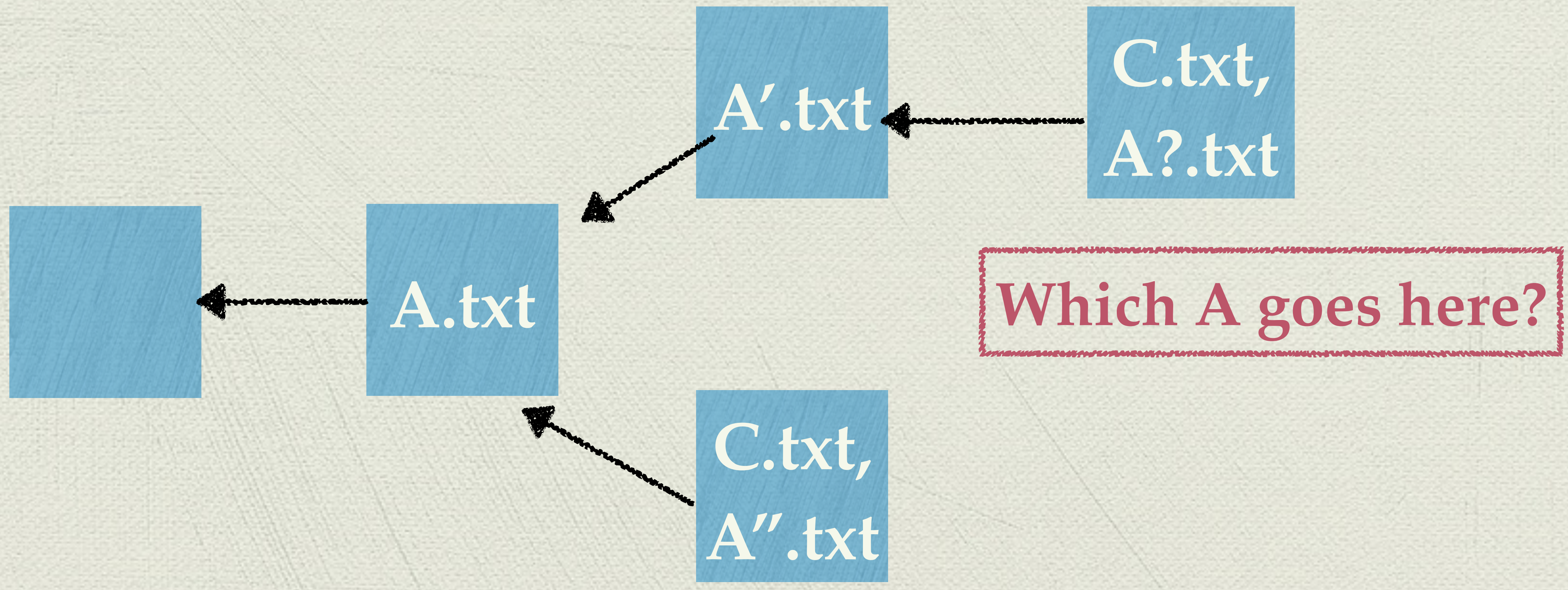
Rebase

- Facts are the same: A′ is newer than A, and A is the same as A
- Therefore, keep A′



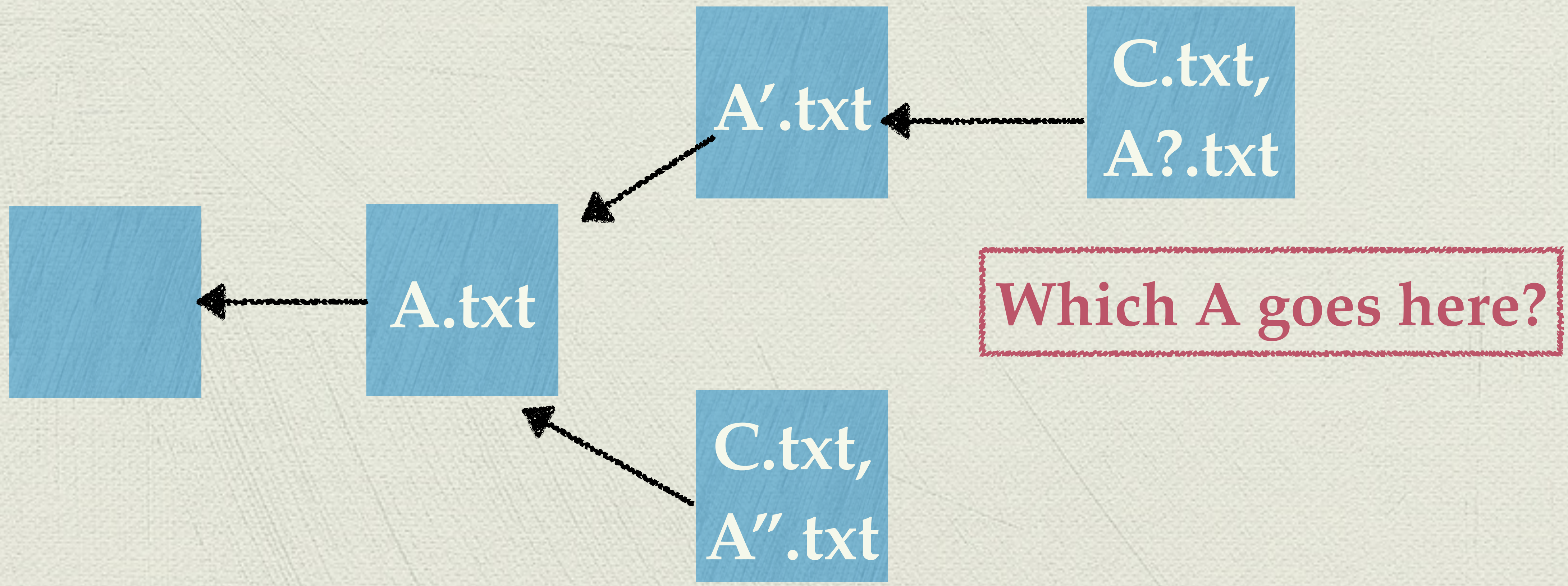
Rebase

- Facts: A′ is newer than A, A″ is newer than A, A′ and A″ have no relation



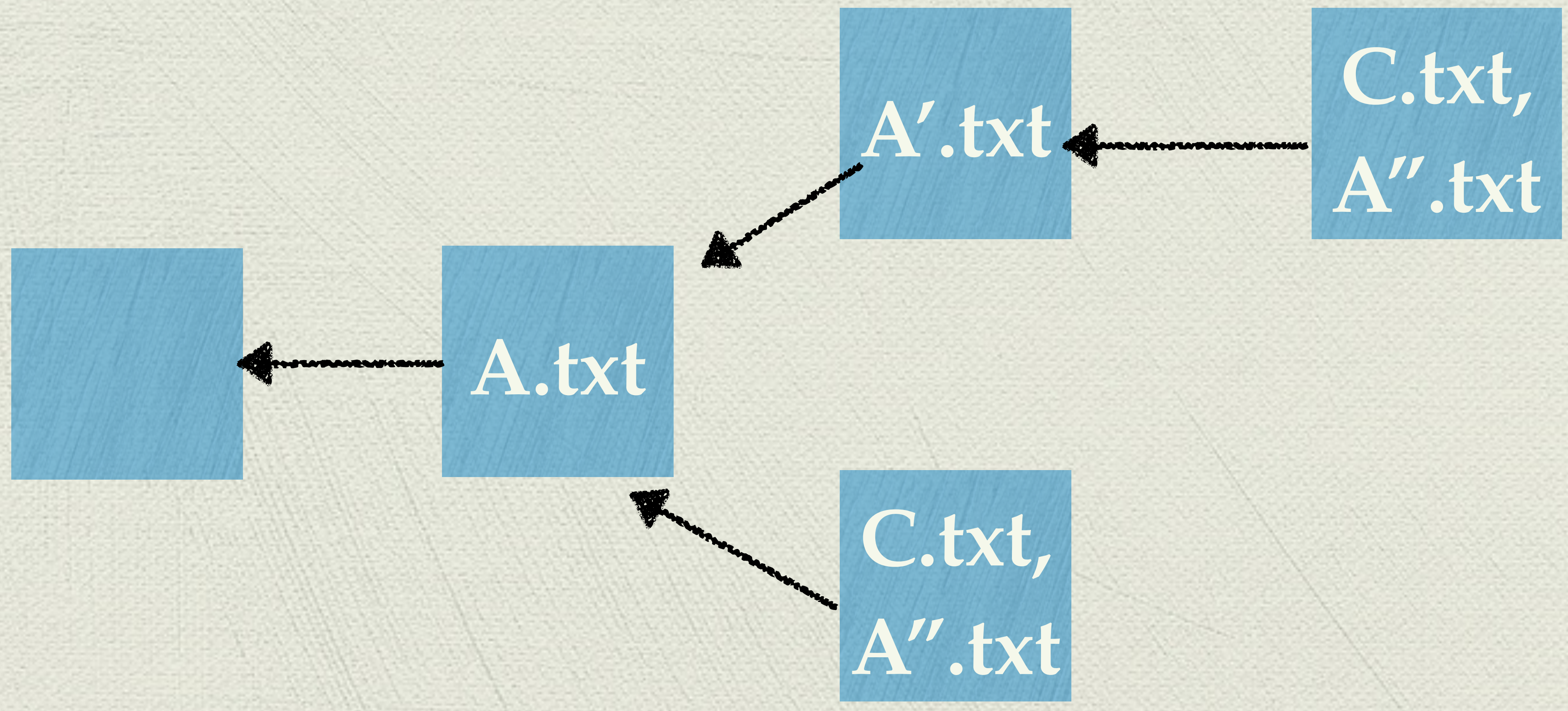
Rebase

- Conclusion: We *should* conflict between A′ and A″



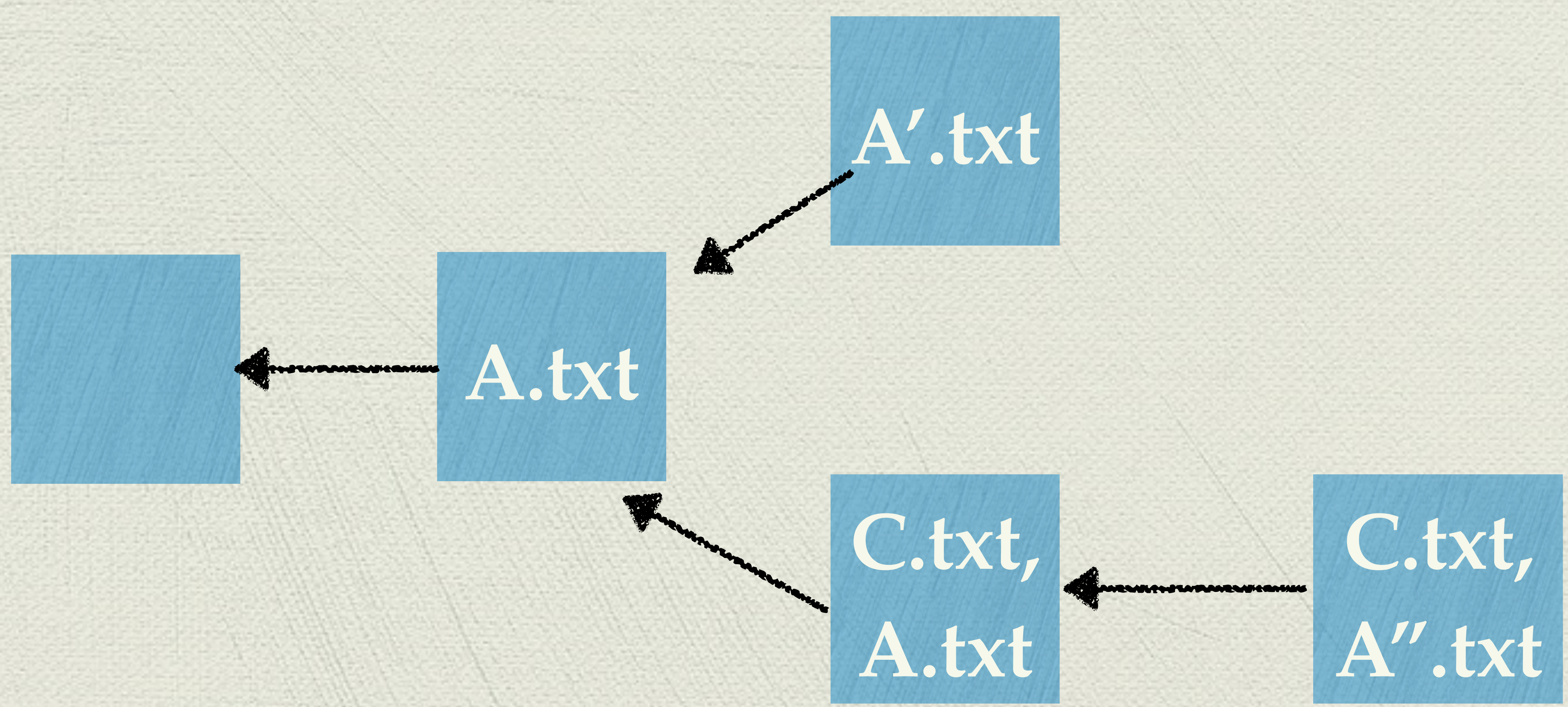
Rebase

- But, in the name of simplicity, just keep A″

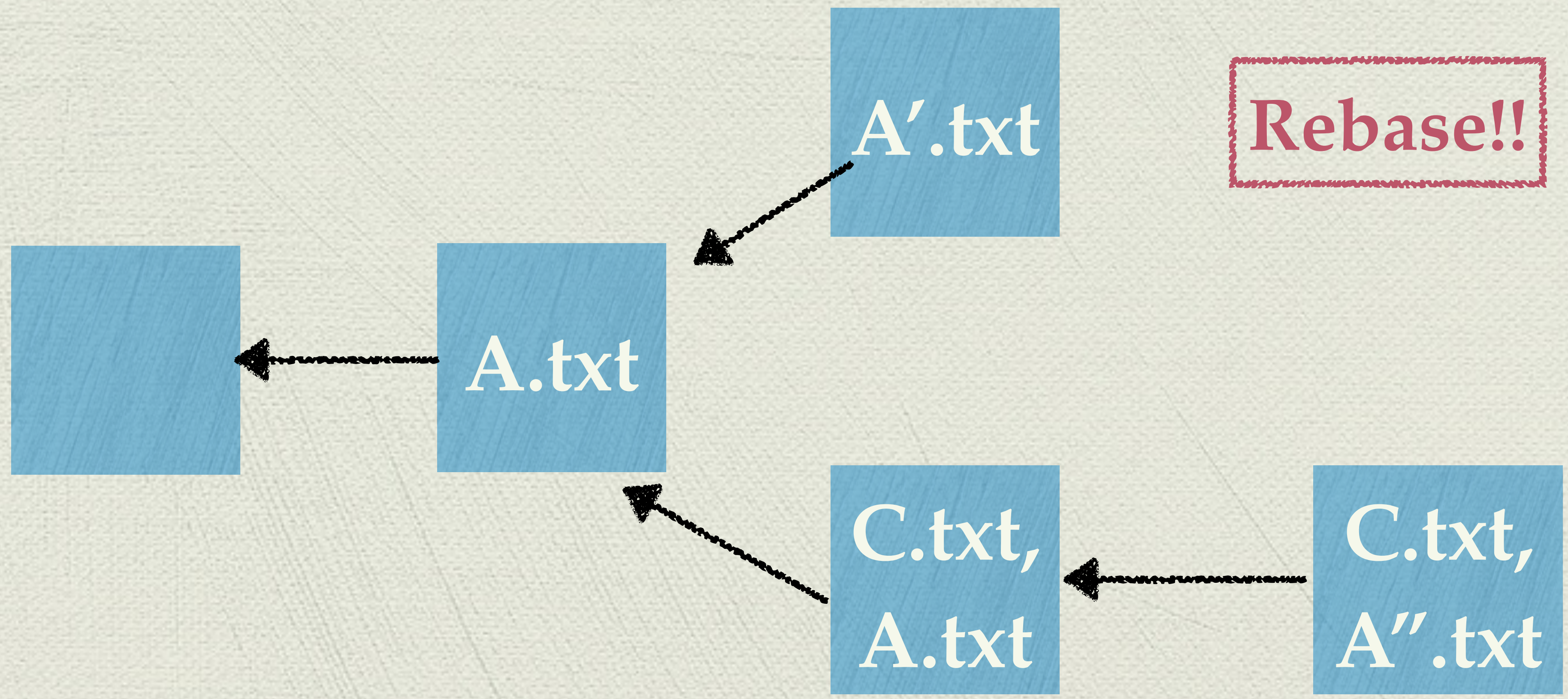


Rebase — weird

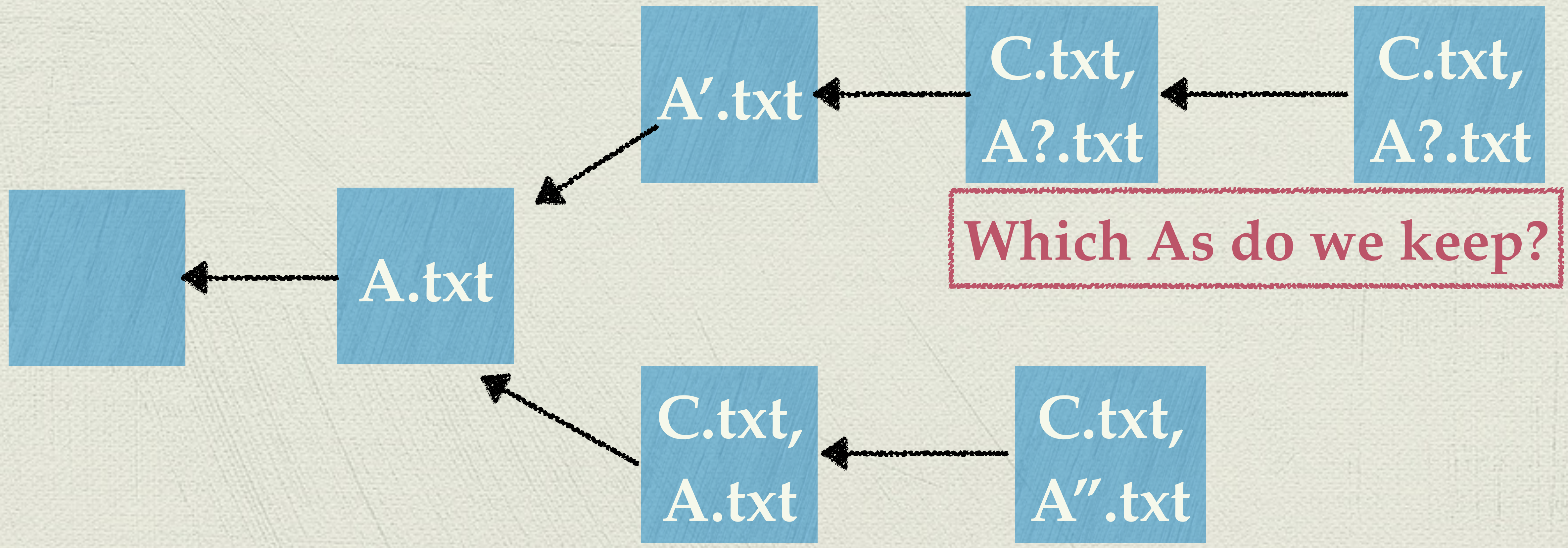
- What if it looks like this before rebase?



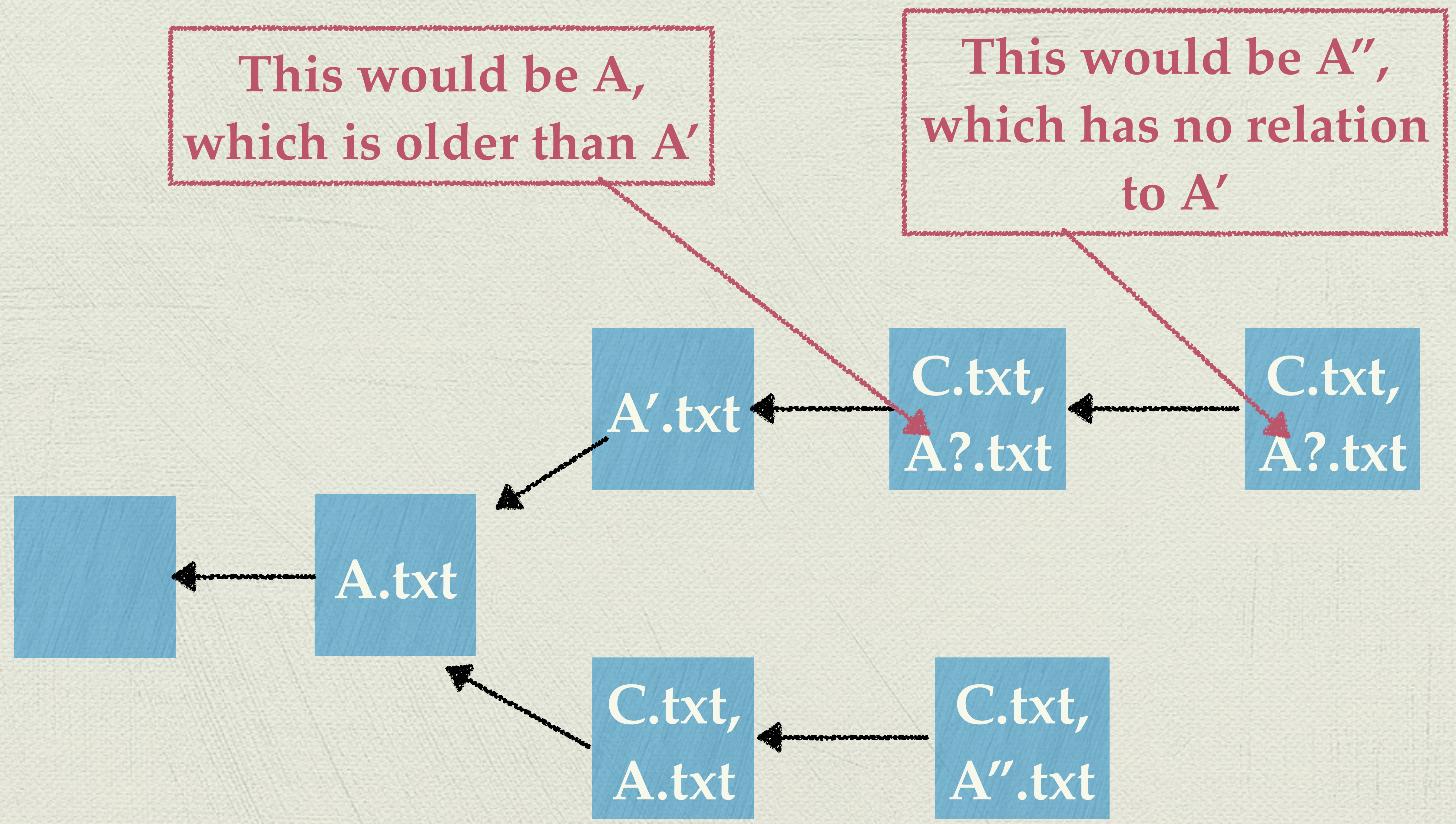
Rebase — weird



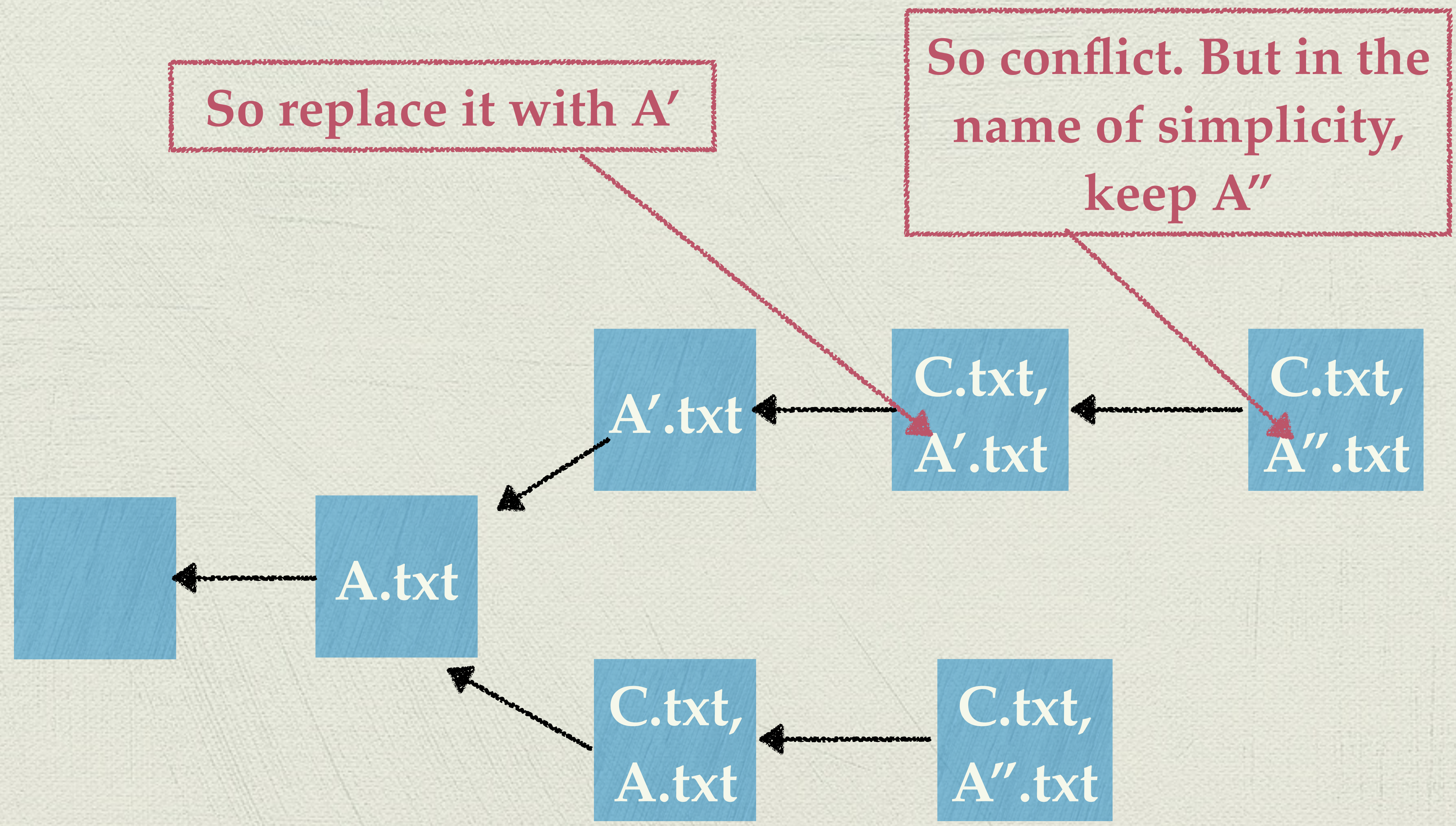
Rebase — weird



Rebase — weird



Rebase — weird



Traversals
(section of lecture I decided to skip)

- Traversal
  - Depth-first
    - Pre-order
    - Post-order
    - In-order
  - Breadth-first
  - Best-first

Let’s get working with trees!

- The different tree traversals themselves are a tree…

- Traversal
  - Depth-first
    - Pre-order
    - Post-order
    - In-order
  - Breadth-first
  - Best-first

Better explained with recursion

- Traversal
  - Depth-first
    - Pre-order
    - Post-order
    - In-order
  - Breadth-first
  - Best-first

Better explained with loops

Pre-order traversal

- As you traverse through the tree, **process** the parents before the children
- **Process** means do some computation with the node. Print it out, add it to some total, etc.

Pre-order traversal

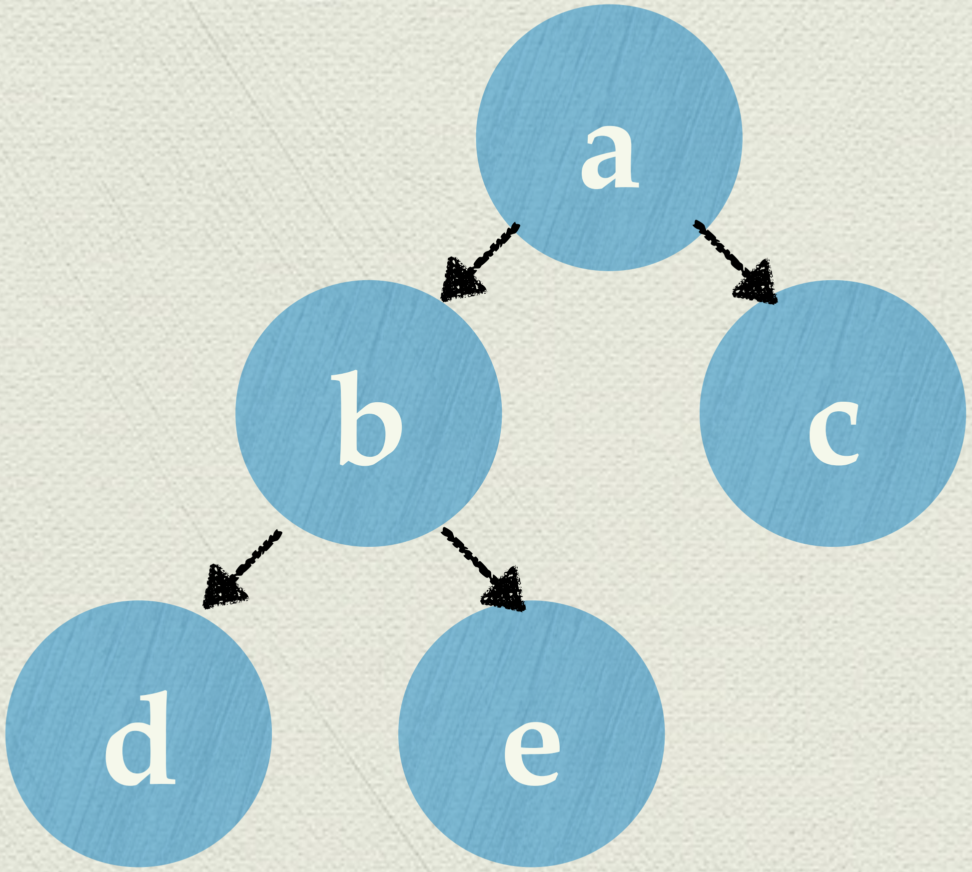
```java
public class TreeNode {
 Object myItem;
 TreeNode myLeft;
 TreeNode myRight;


   public void preOrderTraversal() {
     process(this);
     myLeft.preOrderTraversal();
     myRight.preOrderTraversal();
   }


}
```

Pre-order traversal example

```
public void printPreOrder() {
  System.out.println(this.myItem);
  myLeft.printPreOrder();
  myRight.printPreOrder();
}
```

Pre-order traversal example

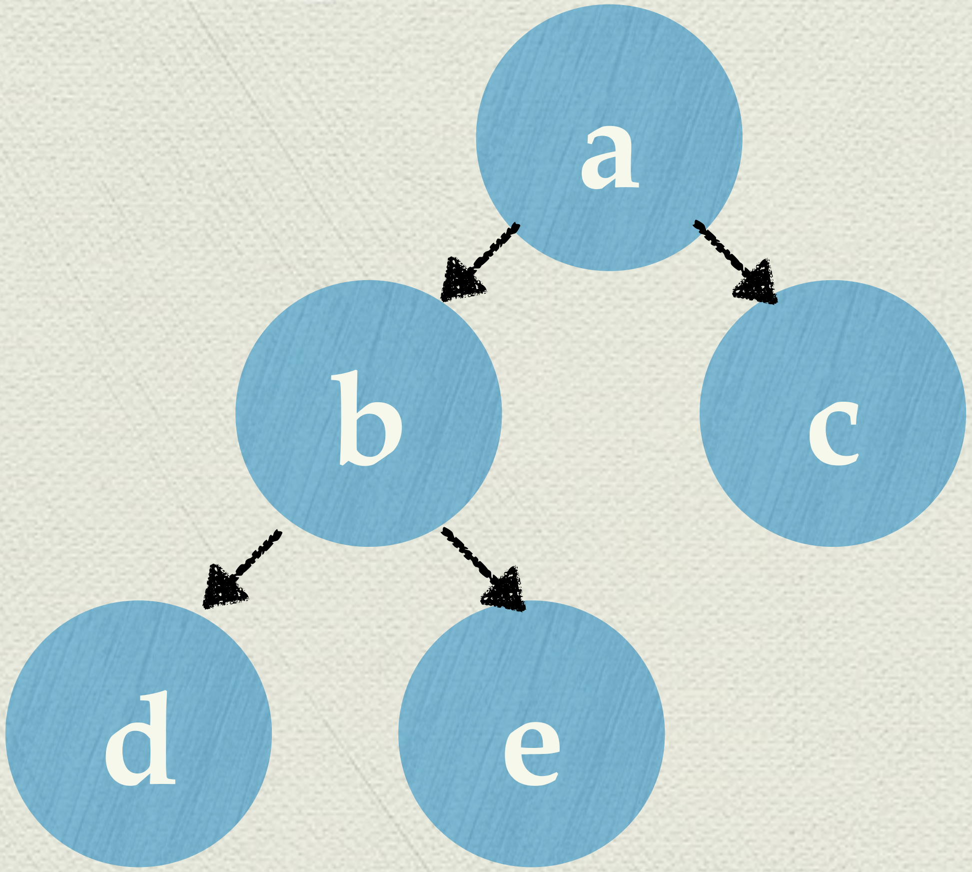
- What would it print?



Pre-order traversal example

- What would it print?

a b d e c



Post-order

- **Process** the children before the parent

Post-order traversal

```
public void postOrderTraversal() {
  myLeft.postOrderTraversal();
  myRight.postOrderTraversal();
  process(this);
}
```

Pre-order traversal

```
public void preOrderTraversal() {
  process(this);
  myLeft.preOrderTraversal();
  myRight.preOrderTraversal();
}
```

Post-order traversal example

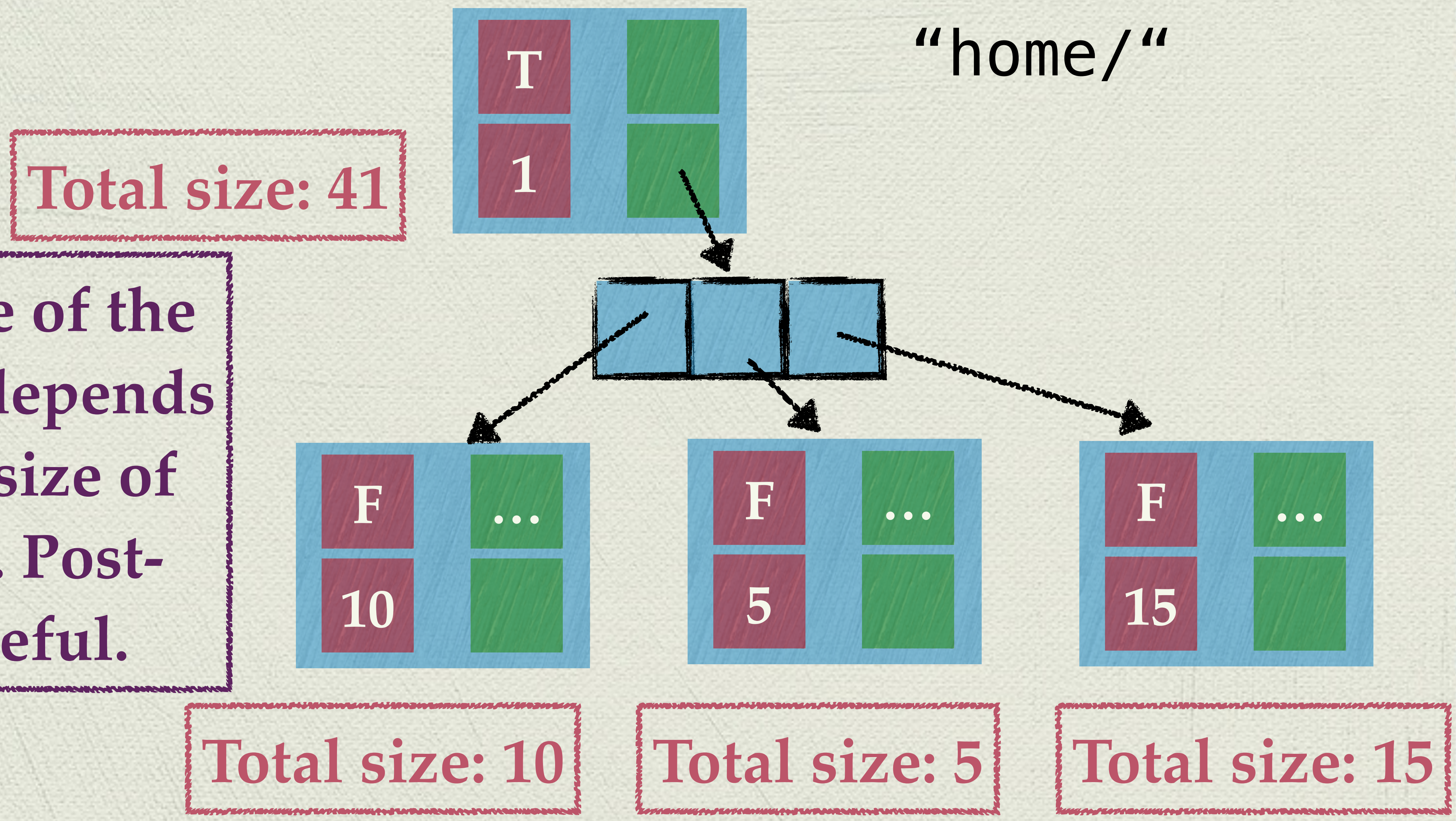
- Compute the total size of a folder

```
public int totalSize() {
  int totalSize = 0;
  for (File child : myContainedFiles) {
    totalSize += child.totalSize();
  }
  totalSize += mySize;
  return totalSize;
}
```

We finish figuring out the size of our children…

Before finishing our own size.

Post-order traversal example



The total size of the parent node depends on the total size of its children. Post-order is useful.

In-order traversal

- **Process** left child, then parent, then right parent

In-order traversal

```
public void inOrderTraversal() {
  myLeft.inOrderTraversal();
  process(this);
  myRight.inOrderTraversal();
}
```

In-order traversal example

- Print out the expression

```
public String toString() {
  String results = "(";
  if (myLeftOperand != null) {
    results += myLeftOperand.toString();
  }
  results += myItem;
  if (myRightOperand != null) {
    results += myRightOperand.toString();
  }
  return results + ")";
}
```

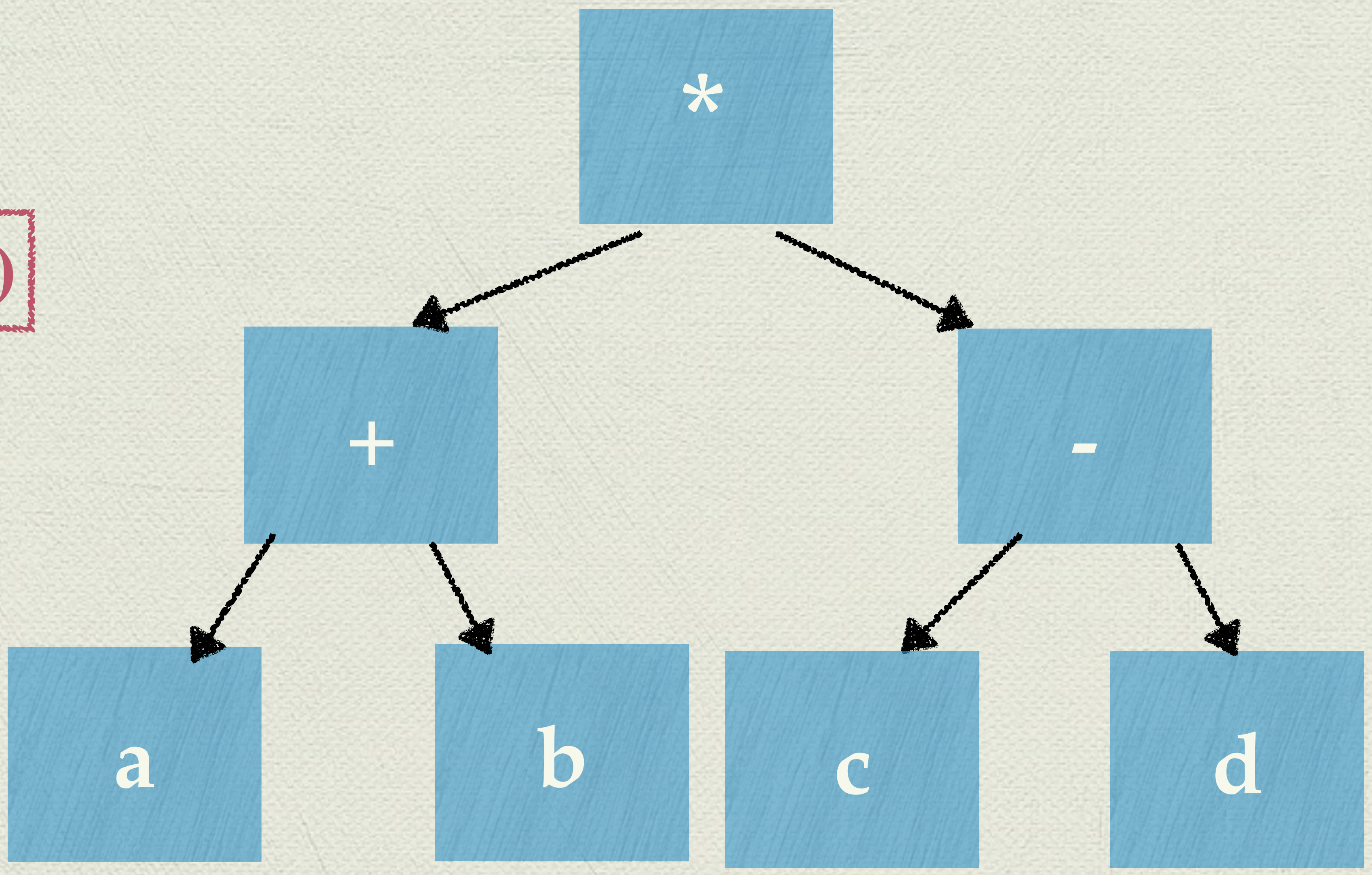
Finish left

Add in this node

Finish right

In-order traversal example

Prints ((a+b)*(c-d))



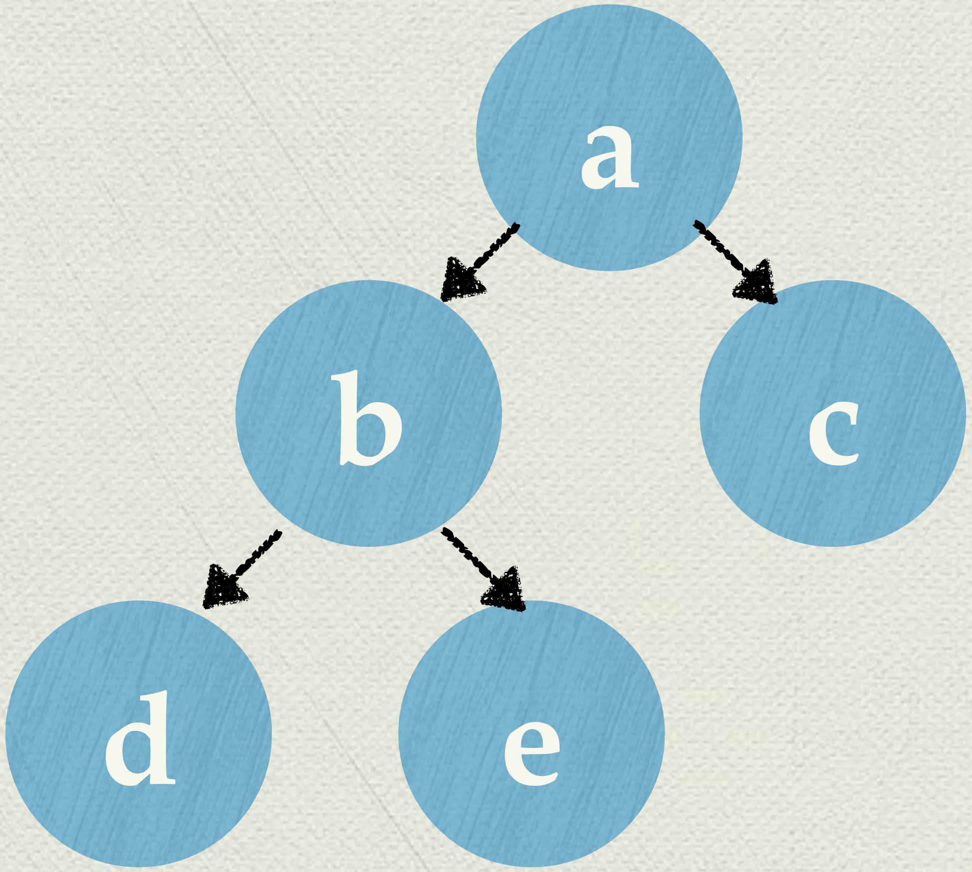
Depth-first traversal to breadth-first traversal

- All the traversal we looked at so far were **depth-first**, meaning they went all the way down a branch before moving onto another branch

- **Breadth-first** instead explores *all* the nodes that are at depth ***D*** *before* going to any at depth ***D + 1***

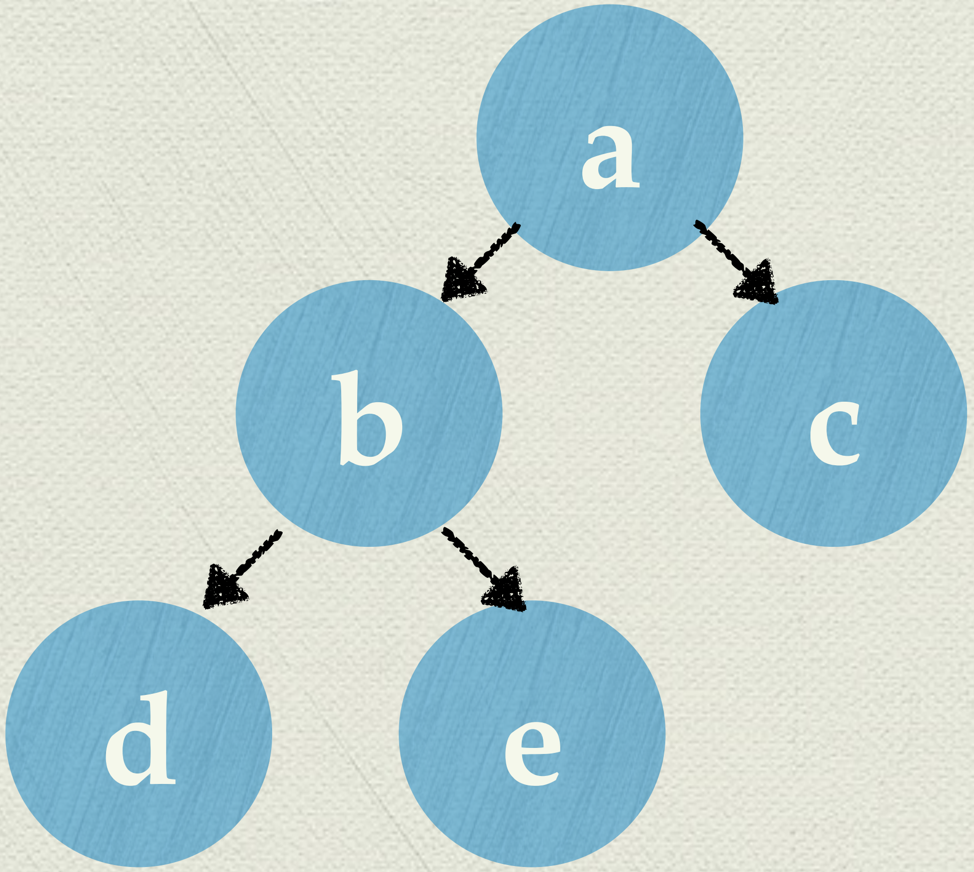
Depth first

a b d e c



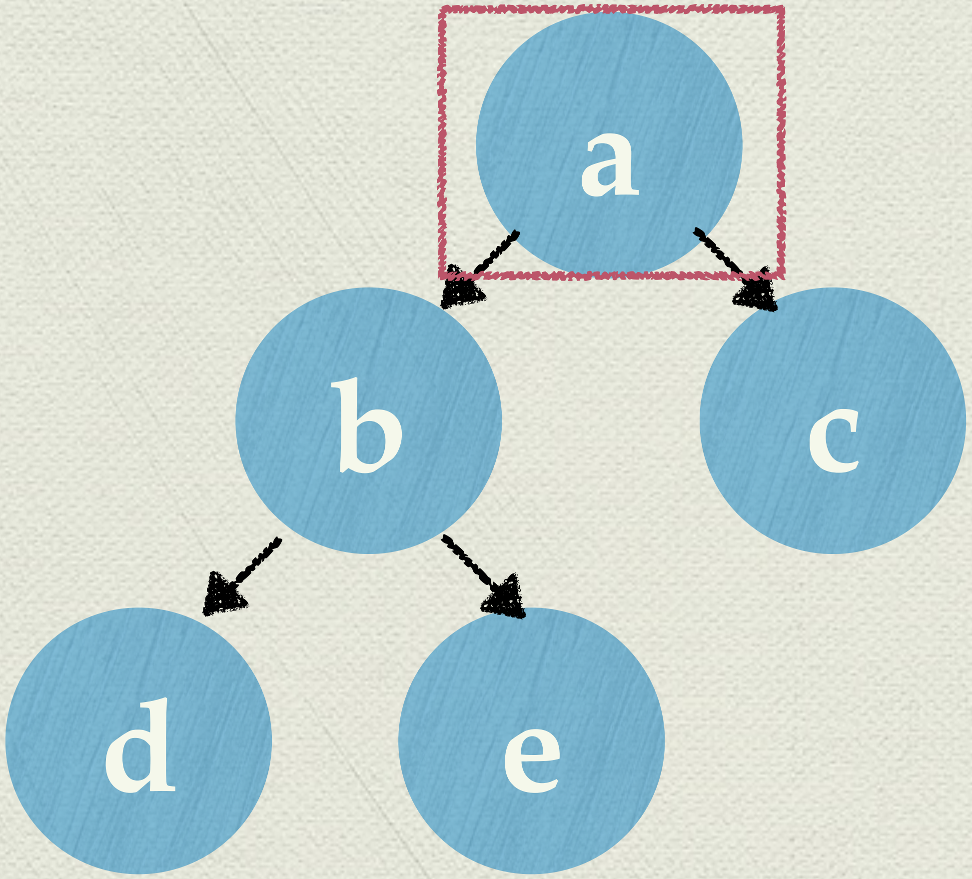
Breadth first

a b c d e



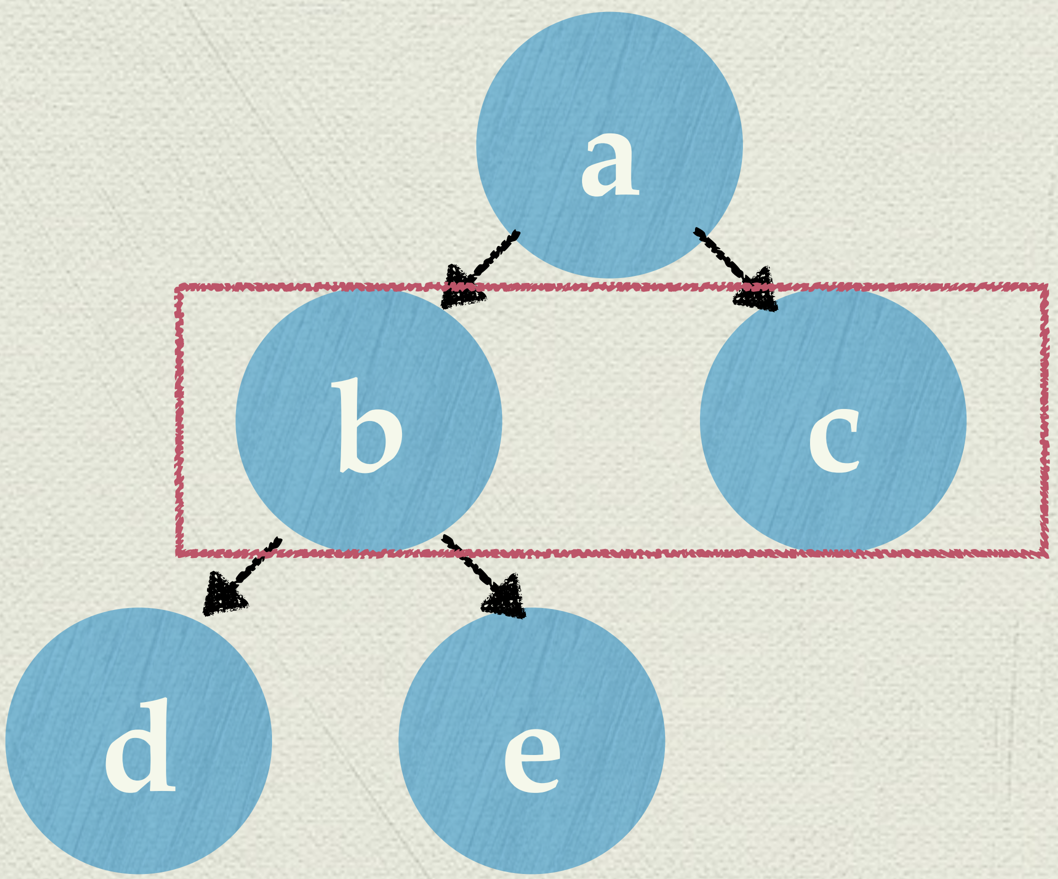
Breadth first

a b c d e



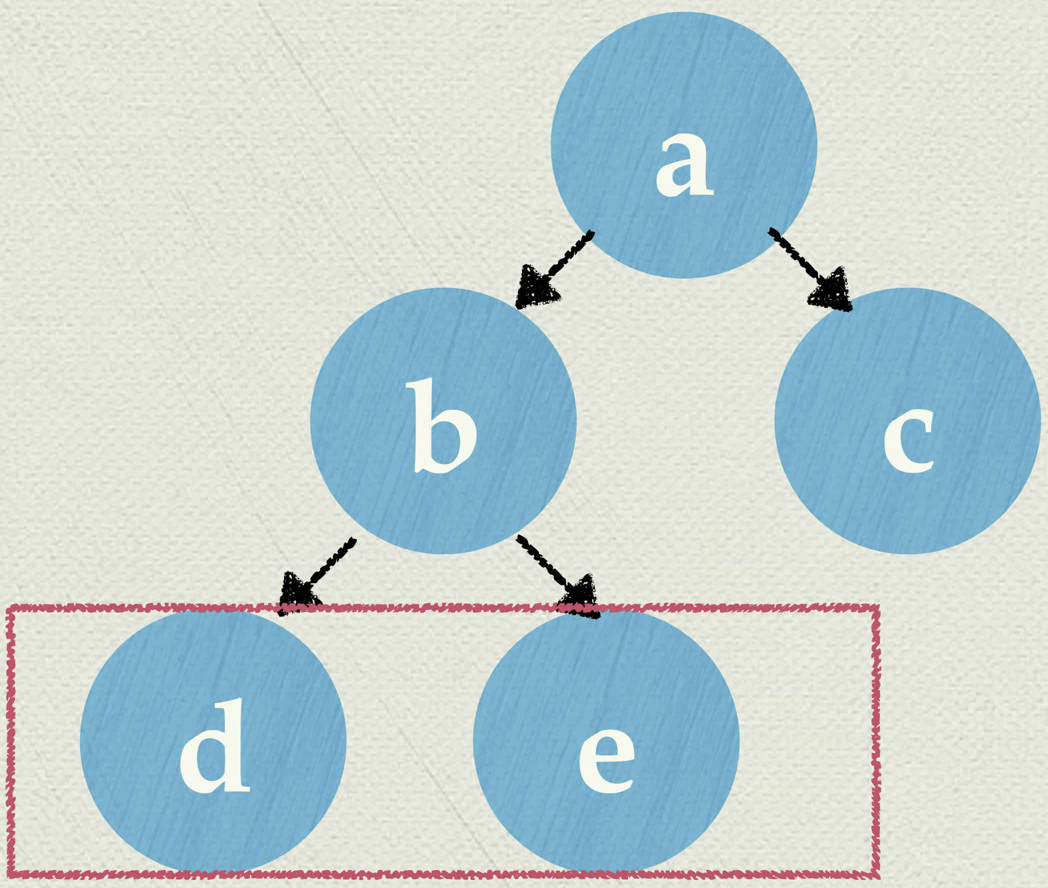
Breadth first

a b c d e



Breadth first

a b c d e



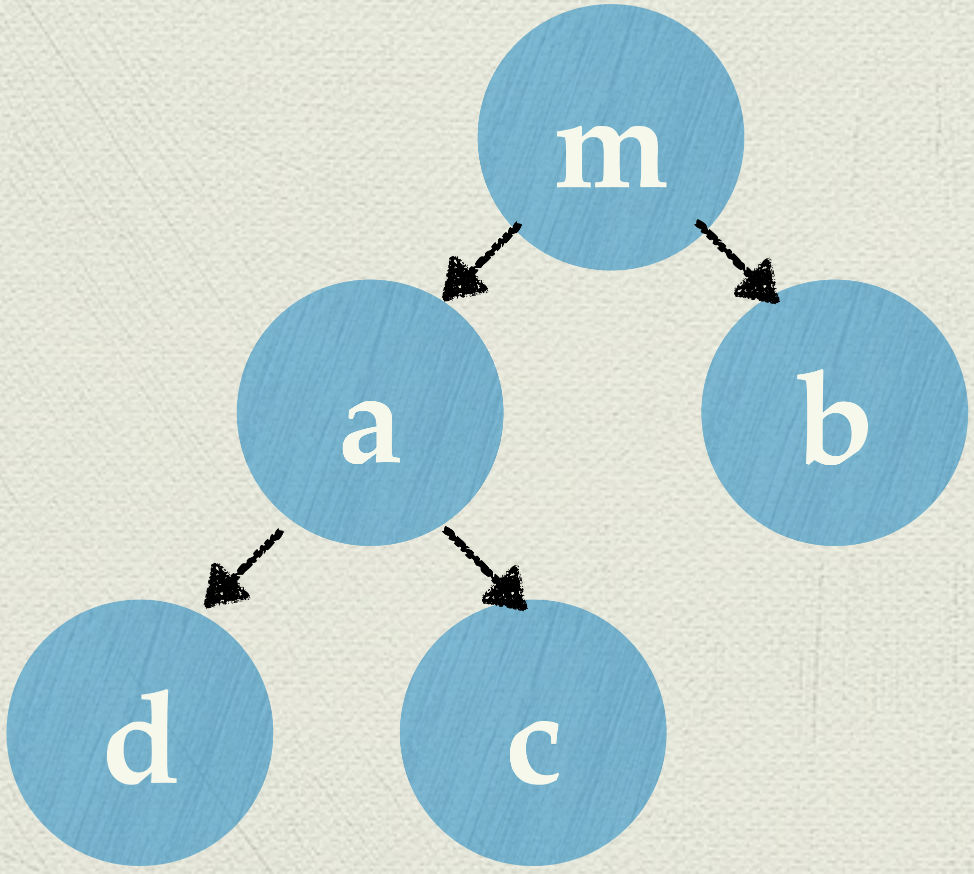
Best first

- **Best first** traversal is not a standard term, but is extremely common
- Choose which node to go to next by some notion of **priority**

Best first

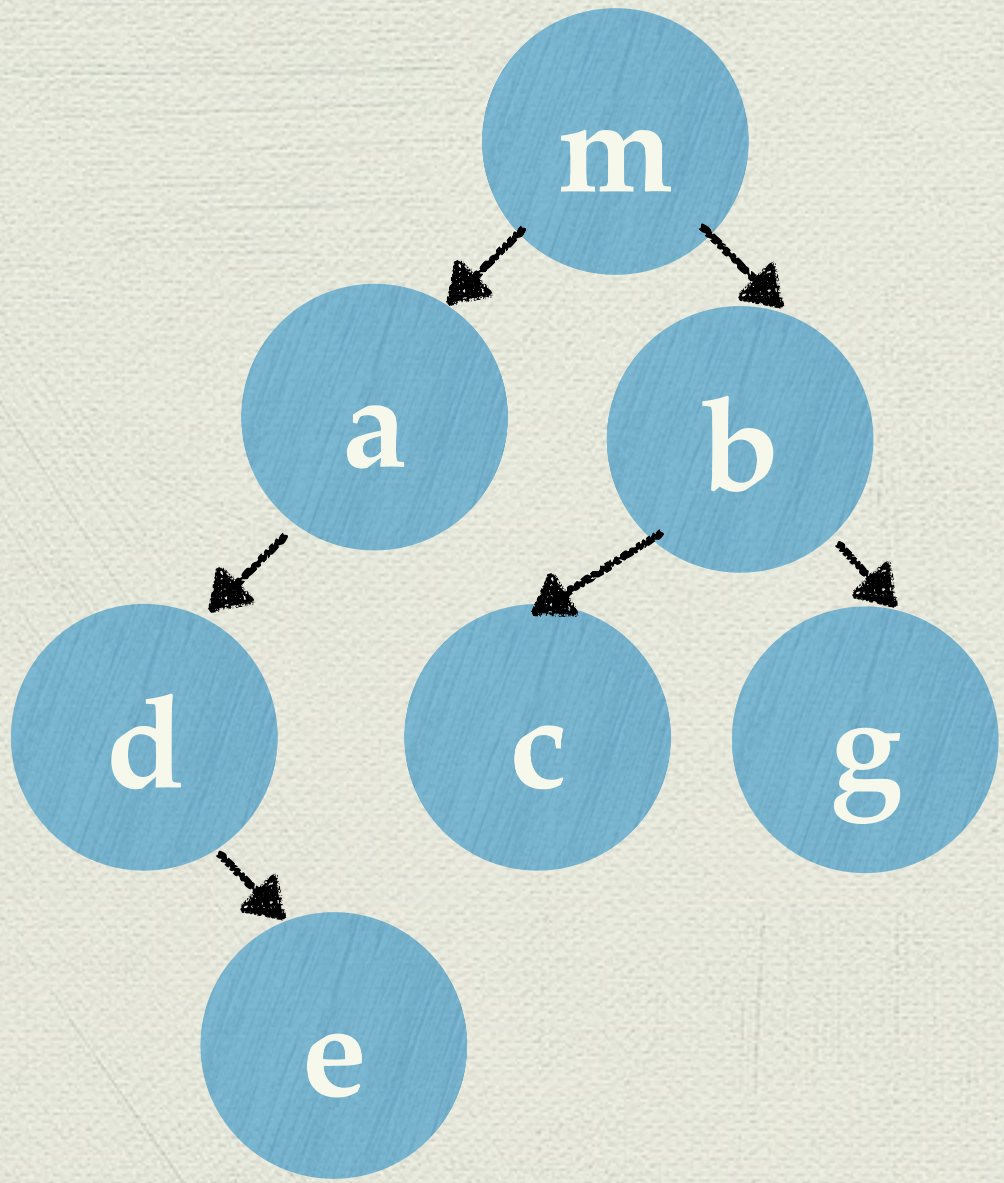
- Example: Go to lower letters alphabetically, when you have a choice

m a b c d



Intuitive tree traversals!

- For the tree, write down the order nodes would be printed out
  - Preorder
  - Postorder
  - Inorder
  - Breadth-first



break

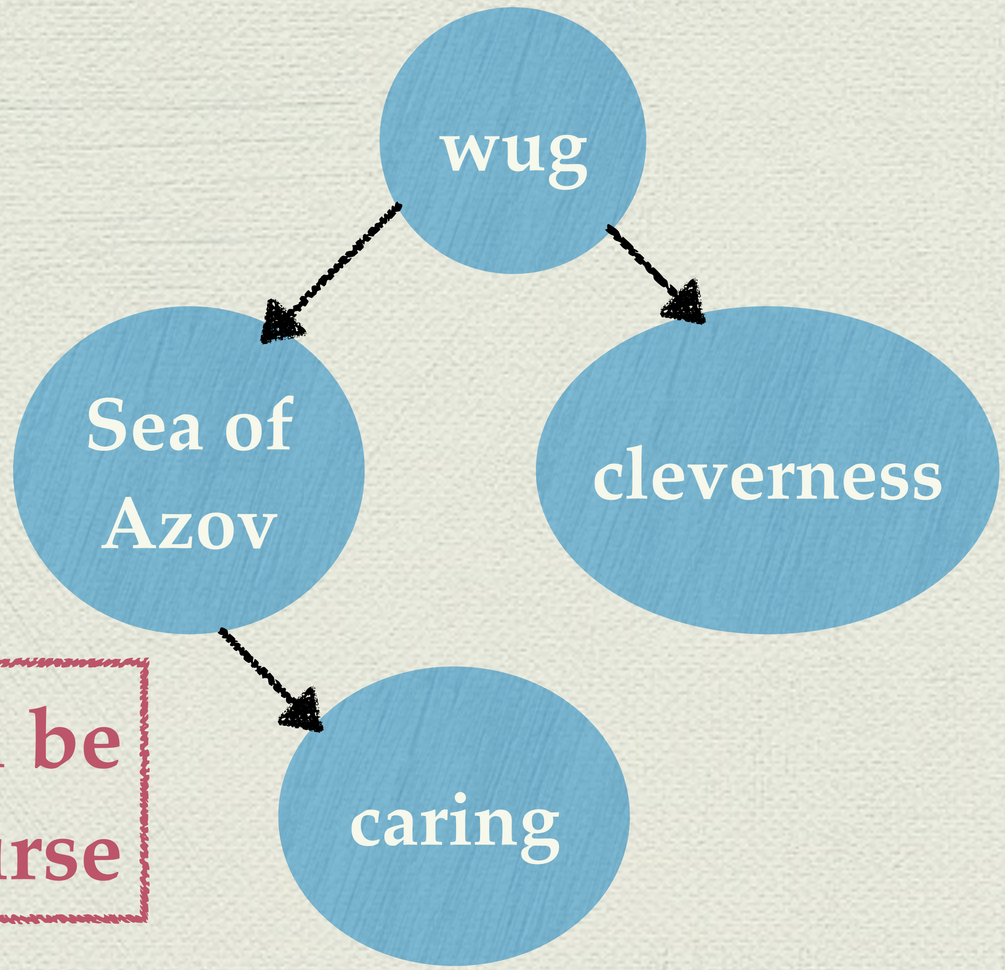
Try out a traversal yourself…

- …with a quiz!

Shallow things

- Here's a class.

```
public class TreeNode {
  String myItem;
  List<TreeNode> myChildren;
}
```



Answer here would be "cleverness", of course

- Write a method: `public String shallowestC`, that returns the shallowest String in the tree that starts with the letter "c"

Breadth-first search (BFS)

- Solve this problem with a breadth-first traversal, also called *breadth-first search* (**BFS**)

BFS

- How did it go? Did you try recursion? It turns out iteration is much easier…

Iterative BFS

- How do we make this happen?
- Let’s start by examining the iterative DFS case

Iterative DFS

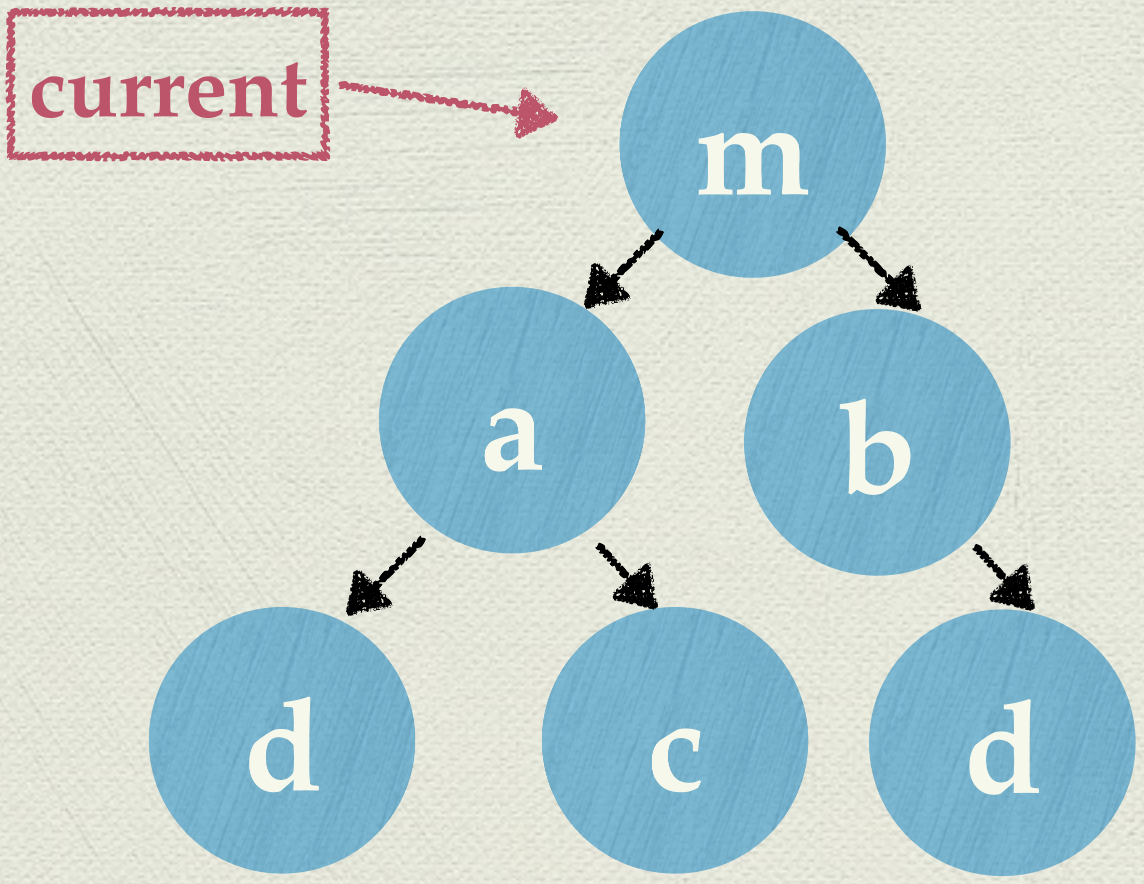
- Iteration for a linked list would have looked like this:

```
TreeNode current = this;
while (current != null) {
  if (current.myItem.startsWith("c")) {
    return current.myItem;
  }
  current = current.myChildren.get(0);
}
```

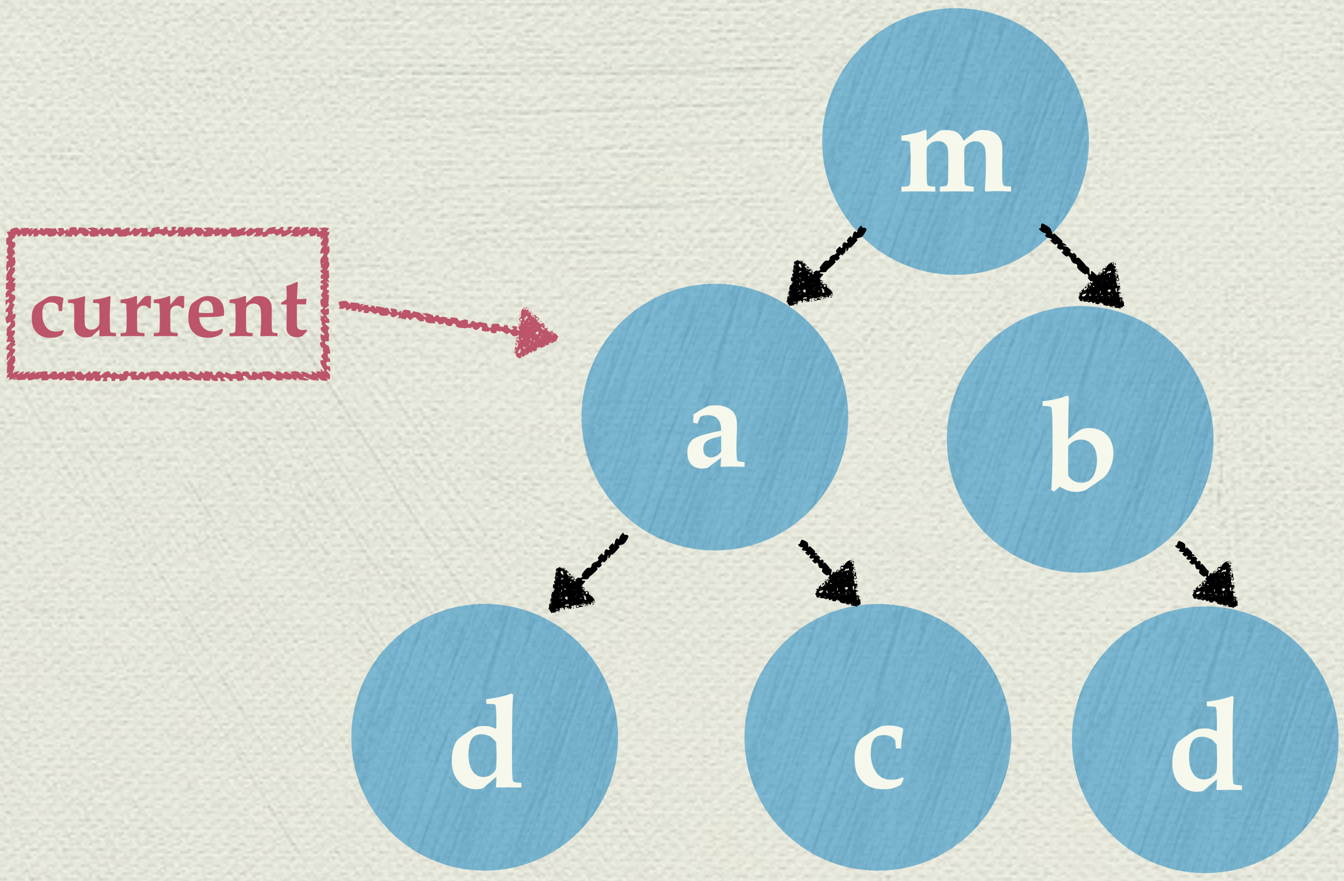
- This *will* take us down the depth of the tree

- But what about the nodes we left behind? DFS doesn't mean *only* go deep, it means go deep *first*

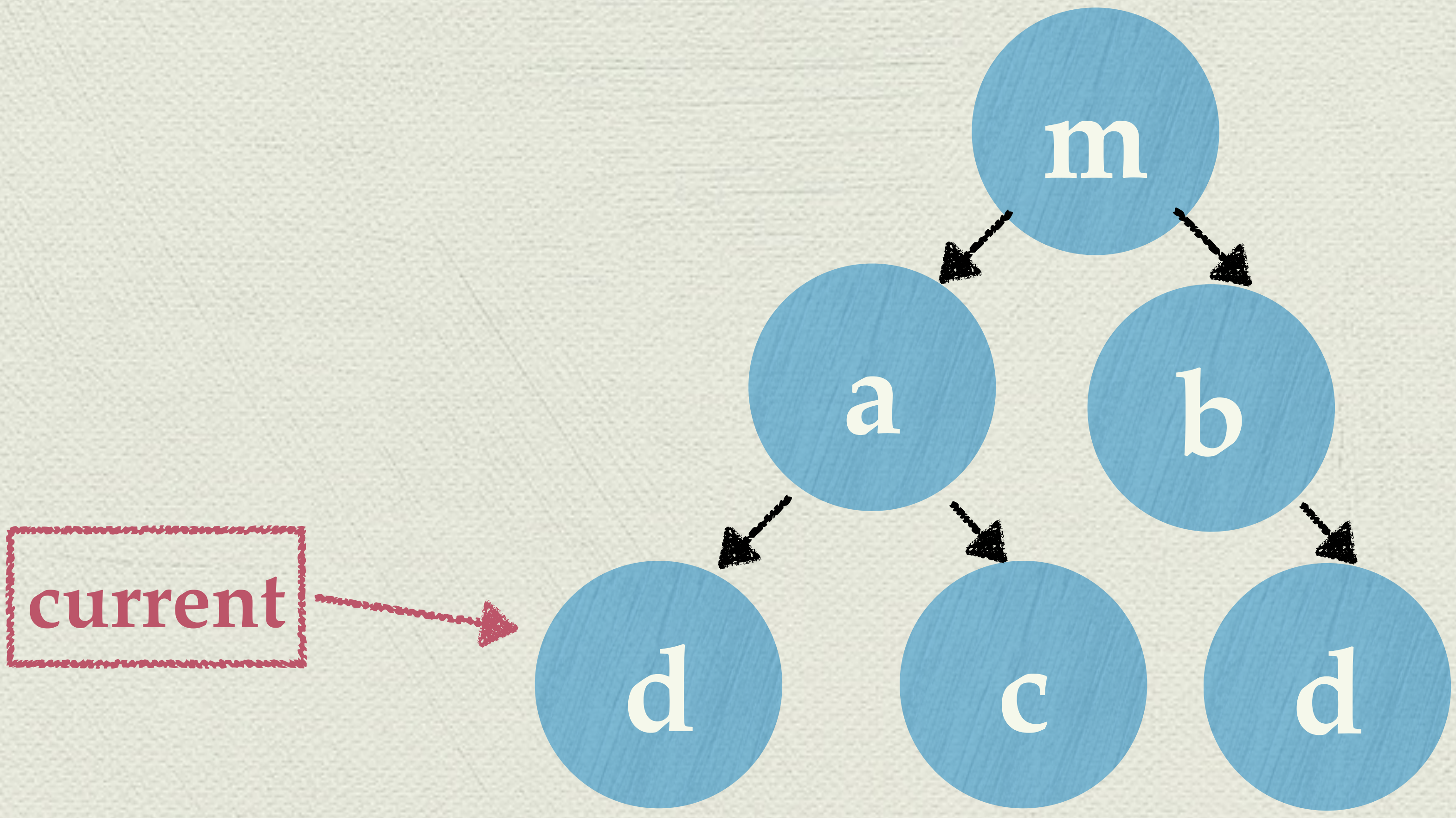
That kind of DFS



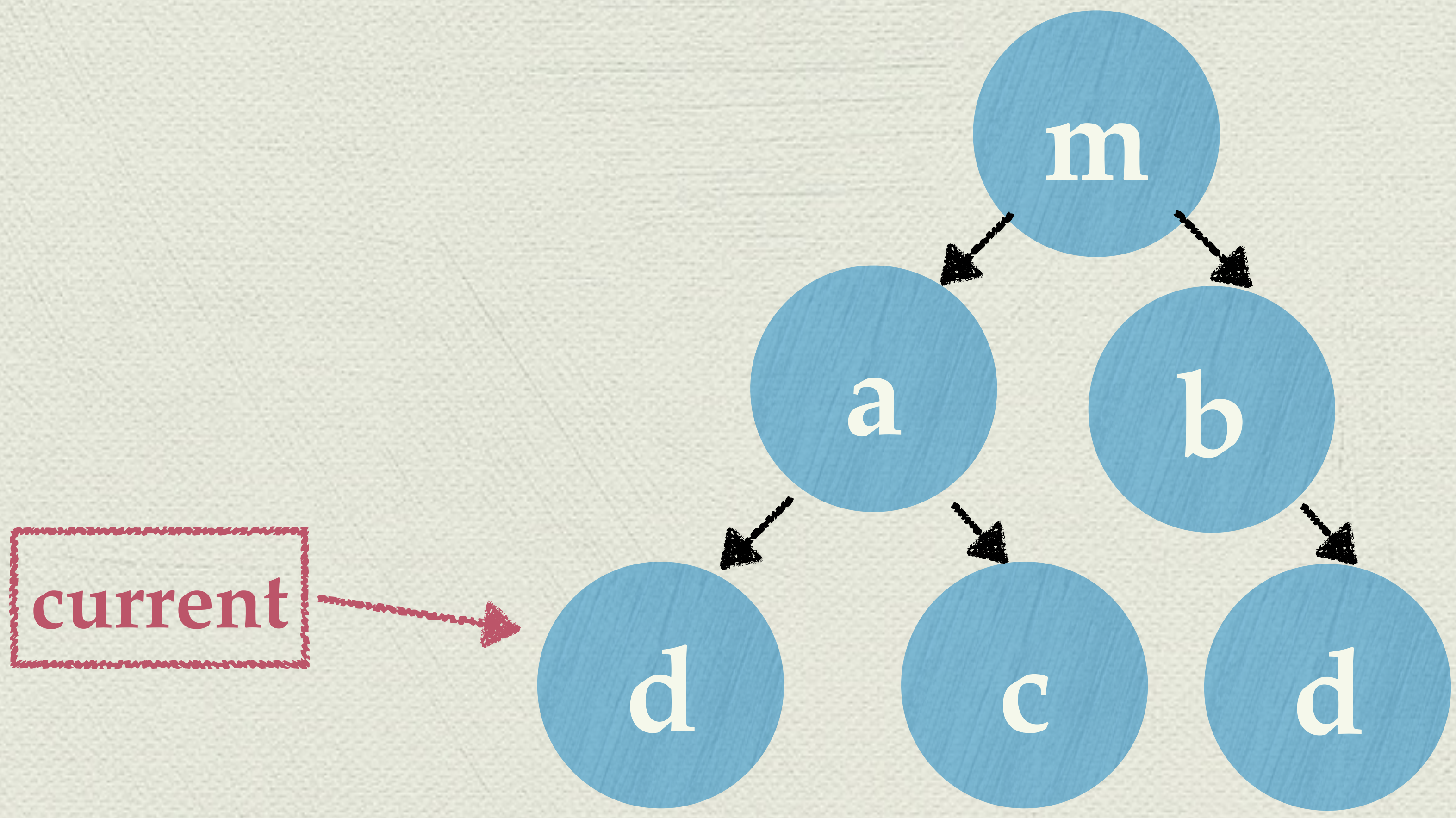
That kind of DFS



That kind of DFS



That kind of DFS



Now what?

DFS

- **Idea**: store the nodes we left behind in a list
- After we're done going down, which one do we want next? Another one that's also deep…
- …or nearby us, or the *latest* one we added to the list
- Seems like a job for a **stack**

DFS (the full story)

```
public String dfsForC()
  Stack<TreeNode> fringe = new Stack<>();
  fringe.push(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.pop();
    if (current.myItem.startsWith("c")) {
      return current.myItem;
    }
    for (TreeNode child: current.myChildren) {
      fringe.push(child);
    }
  }
  return null;
}
```

DFS (the full story)

```
public String dfsForC()
  Stack<TreeNode> fringe = new Stack<>();
  fringe.push(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.pop();
    if (current.myItem.startsWith("c")) {
      return current.myItem;
    }
    for (TreeNode child: current.myChildren) {
      fringe.push(child);
    }
  }
  return null;
}
```

Current starts at **this**

DFS (the full story)

```
public String dfsForC()
  Stack<TreeNode> fringe = new Stack<>();
  fringe.push(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.pop();
    if (current.myItem.startsWith("c")) {
      return current.myItem;
    }
    for (TreeNode child: current.myChildren) {
      fringe.push(child);
    }
  }
  return null;
}
```

If current is what we want, we're done

DFS (the full story)

```
public String dfsForC()
  Stack<TreeNode> fringe = new Stac
  fringe.push(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.pop();
    if (current.myItem.startsWith('
      return current.myItem;
    }
    for (TreeNode child: current.myChildren) {
      fringe.push(child);
    }
  }
  return null;
}
```

Otherwise, we need to look further in the tree. So add our children as possible places to go next

DFS (the full story)

```
public String dfsForC()
  Stack<TreeNode> fringe = new Stack<>();
  fringe.push(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.pop();
    if (current.myItem.startsWith(
      return current.myItem;
    }
    for (TreeNode child: current.m
      fringe.push(child);
    }
  }
  return null;
}
```

Choose a node to try next. Take the most recent one we added, because it is the deepest

BFS

- Great, so we made DFS
- We really wanted BFS, though
- Only a small change away!!

BFS (the full story)

```
public String bfsForC()
  Queue<TreeNode> fringe = new LinkedList<>();
  fringe.offer(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.poll();
    if (current.myItem.startsWith("c")) {
      return current.myItem;
    }
    for (TreeNode child: current.myChildren) {
      fringe.push(child);
    }
  }
  return null;
}
```

Only a few changes…

BFS (the full story)

```
public String bfsForC()
  Queue<TreeNode> fringe = new LinkedList<>();
  fringe.offer(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.poll();
    if (current.myItem.startsWith("
      return current.myItem;
    }
    for (TreeNode child: current.my
      fringe.push(child);
    }
  }
  return null;
}
```

Choose a node to try next. Take the *least* recent one we added, because it is the shallowest

Best-first search

```
public String bestfsForC()
  PriorityQueue<TreeNode> fringe = new PriorityQueue<>();
  fringe.offer(this);
  TreeNode current = null;
  while (!fringe.isEmpty()) {
    current = fringe.poll();
    if (current.myItem.startsWith("c")) {
      return current.myItem;
    }
    for (TreeNode child: current.myChildren) {
      fringe.push(child);
    }
  }
  return null;
}
```