Implementing Binary Search Trees (BSTs)

Please take a handout &
Sit in row H or forward

Binary:

every node has a most two children

Search:

everything to the left: <

everything to the right: >

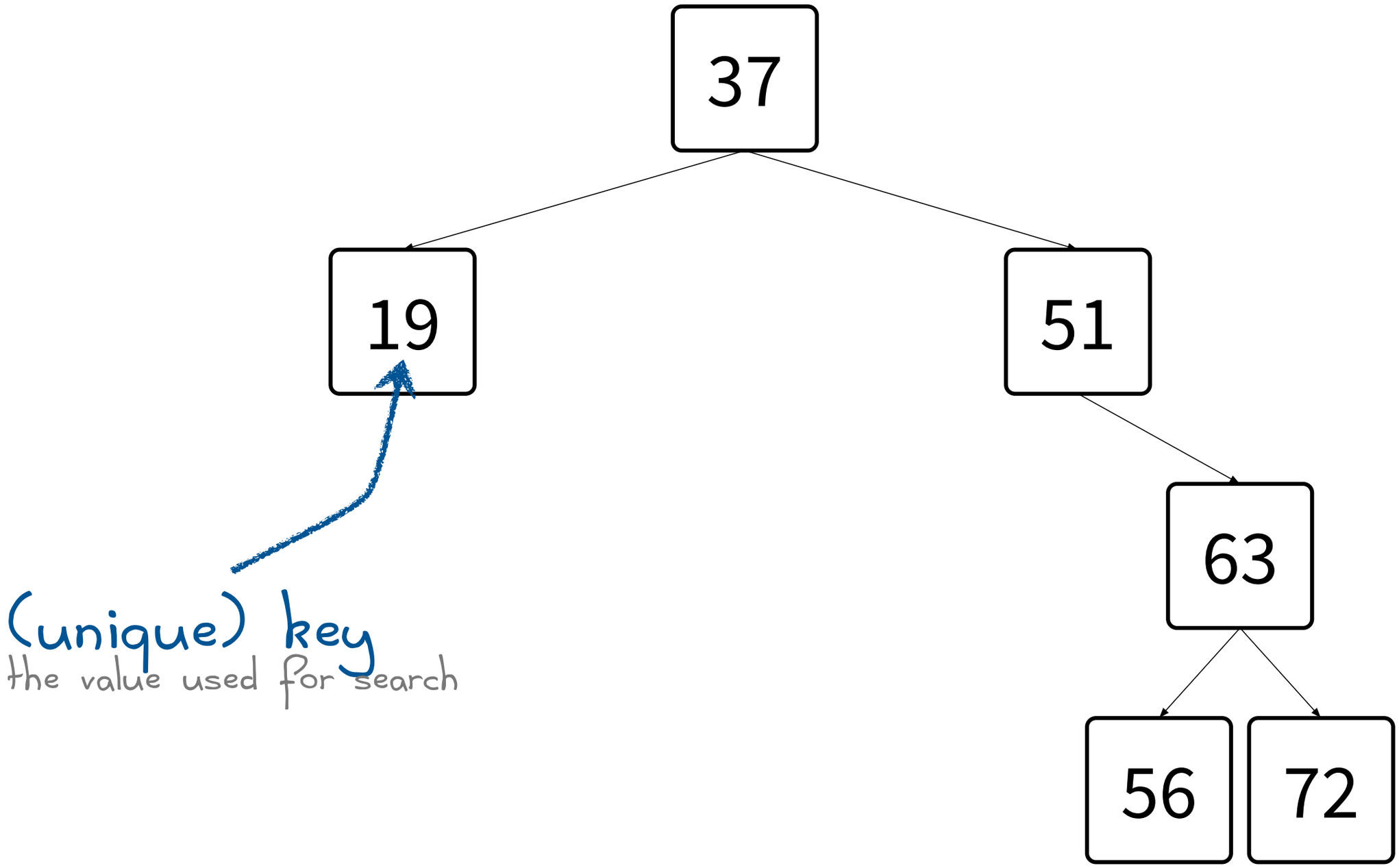
Trees:

unique root node

exactly one path from the root to each node

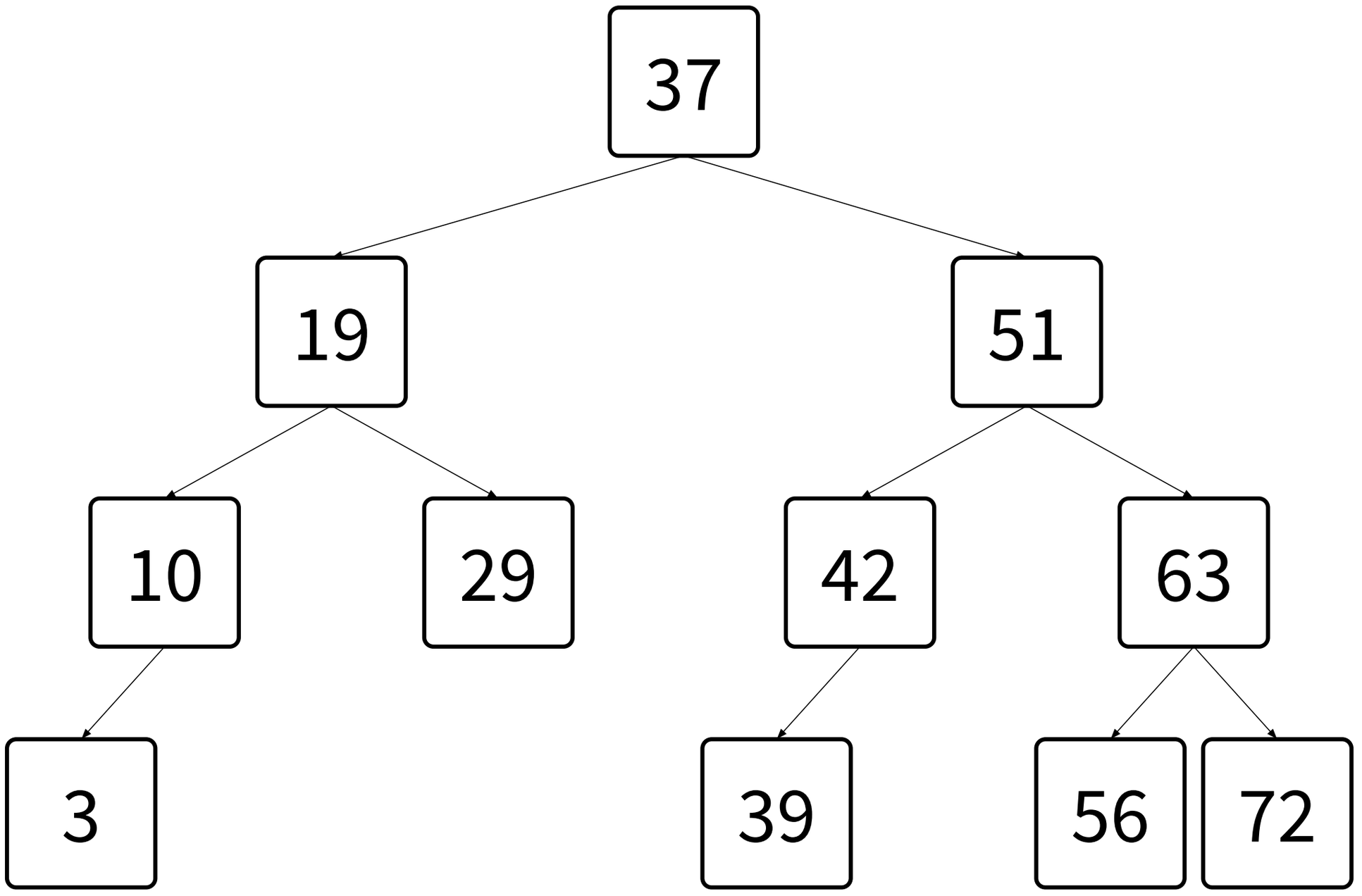
Binary *search* trees (BSTs)

order: every parent is greater than all the nodes in its left subtree and less than all the nodes in the right



Balanced binary search trees

structure: every subtree has roughly the same number of nodes as its sibling



Perfect trees

structure: all leaves are at the same level

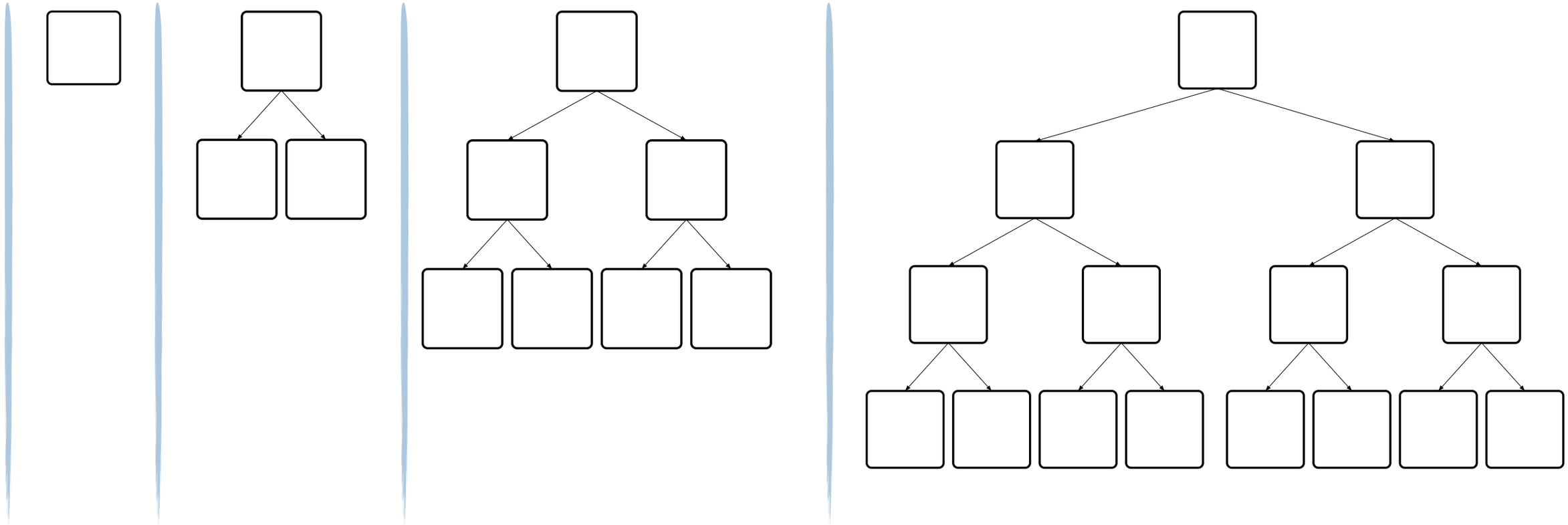
n = 0
h = -1

n = 1
h = 0

n = 3
h = 1

n = 7
h = 2

n = 15
h = 3



$$h = \lfloor \log_2(n) \rfloor$$

- Most trees aren’t perfect (why not?)
- But perfect trees are useful for analyzing balanced trees.

Recall: worst-case analysis

How bad can it get?

Given a collection of size N and an operation:

What's the worst input for the operation?

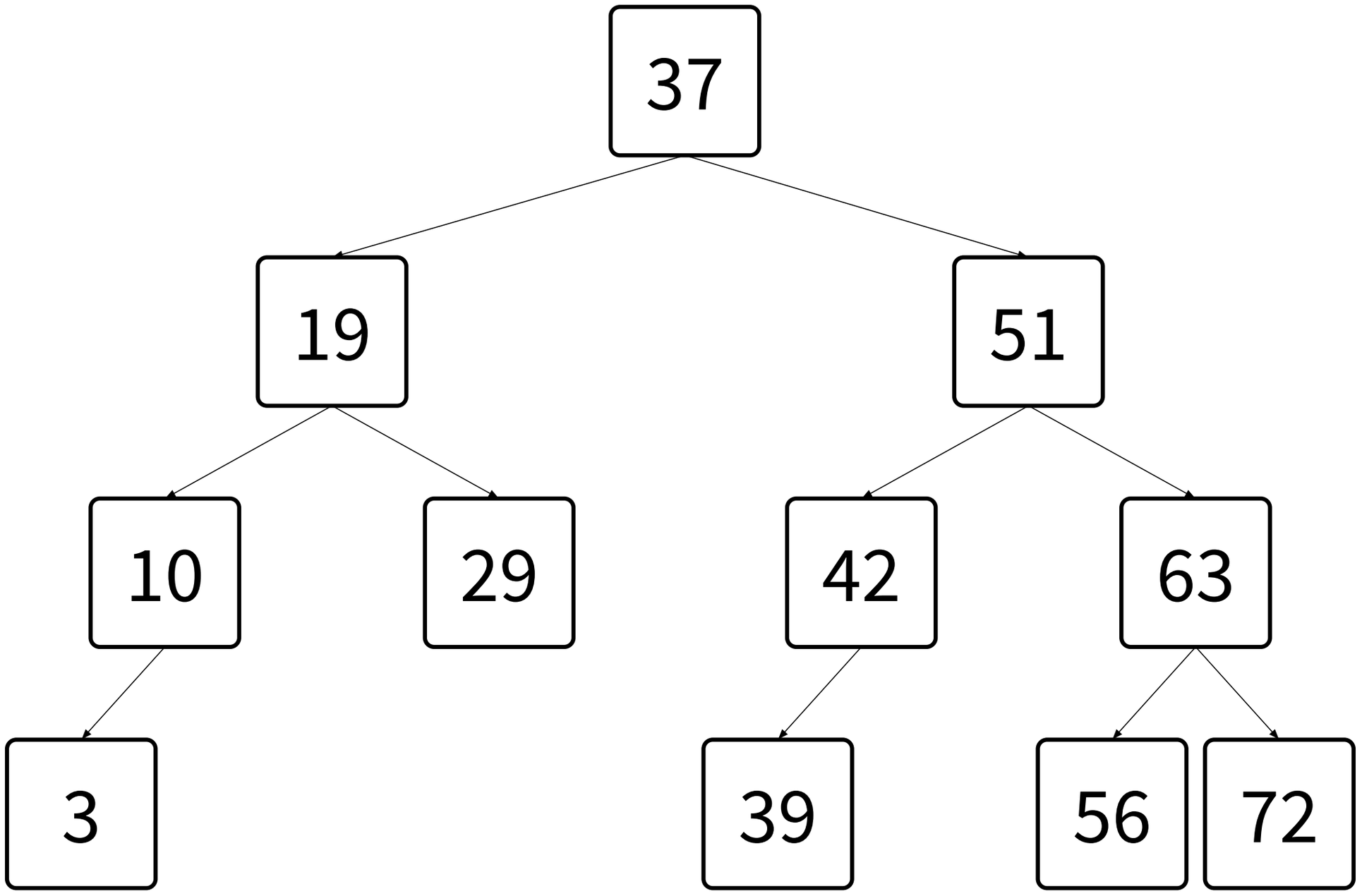
How expensive is the operation, for that input? (cost = # of elements accessed)

	find	insert	max	list elements in order
unsorted array	O(n)	O(n) if array is full	O(n)	O(n log n)
sorted array	O(log n)	O(n)	O(1)	O(n)
unsorted linked list	O(n)	O(1)	O(n)	O(n log n)
sorted linked list	O(n)	O(n)	O(n)	O(n)
hash table	O(1)	O(1)	O(n)	O(n log n)
balanced binary search tree (BST)	O(log n)	O(log n)	O(log n)	O(n)

This week's theme: BSTs

- We'll implement **unbalanced** BSTs
- Our goals are to:
  - practice implementing data structures and algorithms
    functional (Racket) **and** imperative (Java) implementations
  - know when to use the right data structure
    If we're doing a lot of inserts and lookups and we care about order, we use a tree. Otherwise, we probably use a hash table.

BST algorithm: find



Write pseudocode for find

here's the pseudocode for binary search, to use as a template.

Given a **sorted array** *values* with N numbers and a number *i*

for simplicity, assume N is a power of 2 (e.g., 2, 4, 8, 16, 32, …)

```
binarySearch(i, values):
    If the list is empty, return false.
    Find the element mid at the middle of values.
    If i == mid, return true.
    If i < mid, call binarySearch on the front half of values
    If i > mid, call binarySearch on the back half of values
```

BST algorithm: find

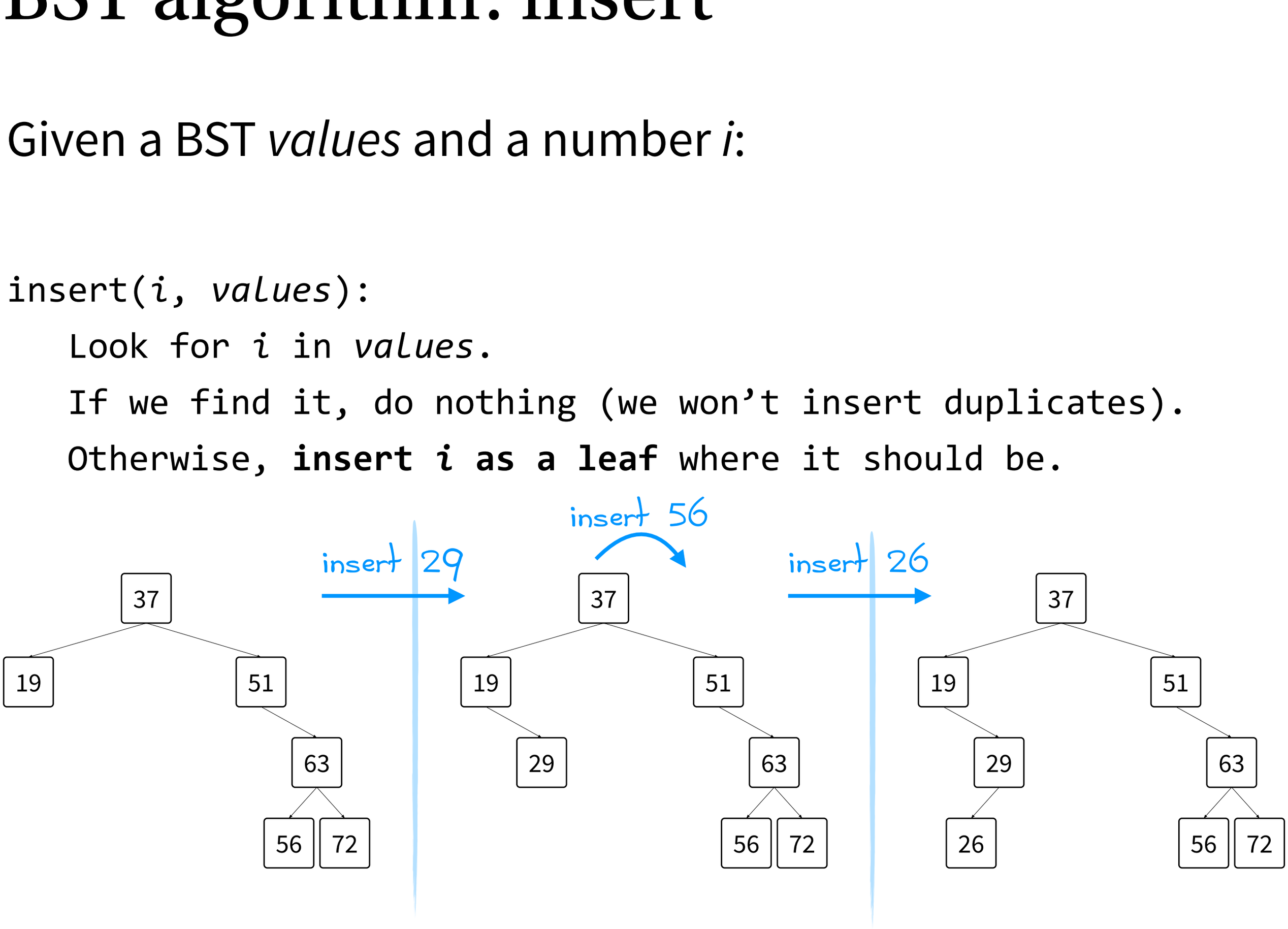
Given a BST *values* and a number *i*:

```
find(i, values):
    If the tree is empty, return false.
    Let key be the value at the root of the tree.
    If key is i, return true.
    If i < key, call find on the left subtree.
    If i > key, call find on the right subtree.
```

BST algorithm: insert

Given a BST *values* and a number *i*:

```
insert(i, values):
    Look for i in values.
    If we find it, do nothing (we won't insert duplicates).
    Otherwise, insert i as a leaf where it should be.
```



BST algorithm: ~~find~~ insert

to insert

Given a BST *values* and a number *i*:

insert

~~find~~(*i*, *values*):

add i as a new node here.

If the tree is empty, ~~return false~~.

Let *key* be the value at the root of the tree.

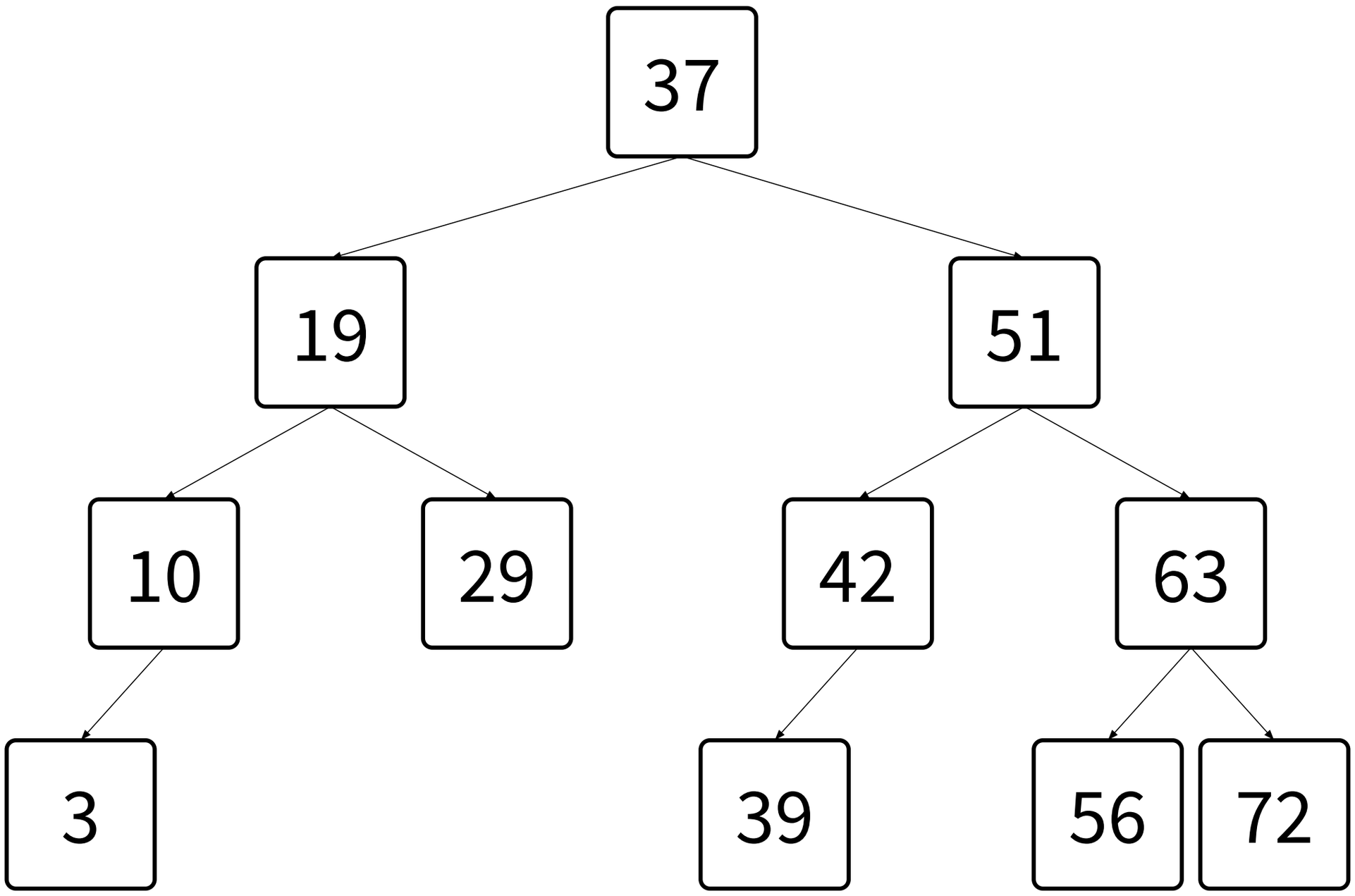
If *key* is *i*, ~~return true~~. do nothing.

If *i* < *key*, call insert ~~find~~ on the left subtree.

If *i* > *key*, call insert ~~find~~ on the right subtree.

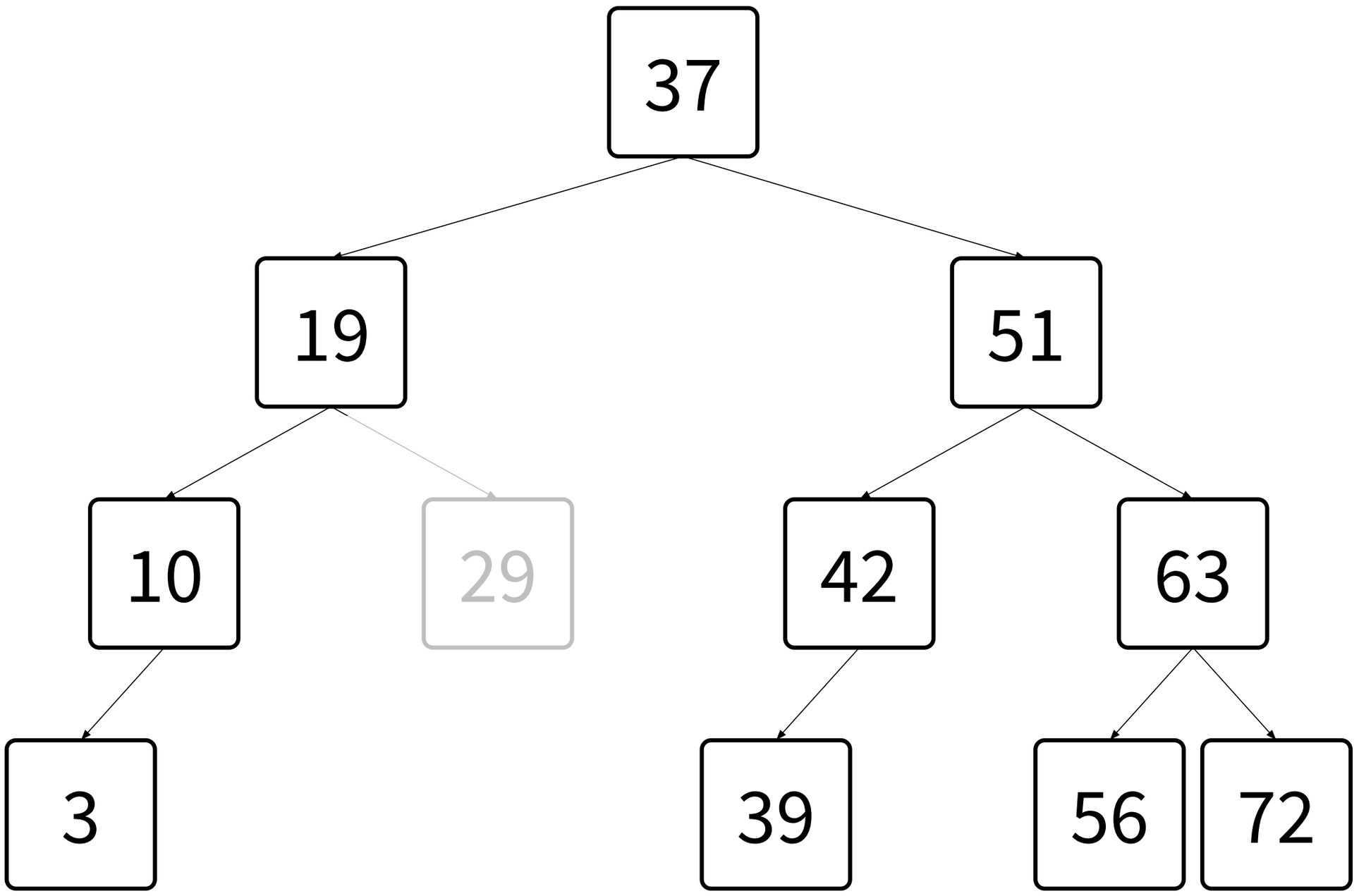
BUT: it's not always easy to implement this idea.

BST algorithm: delete



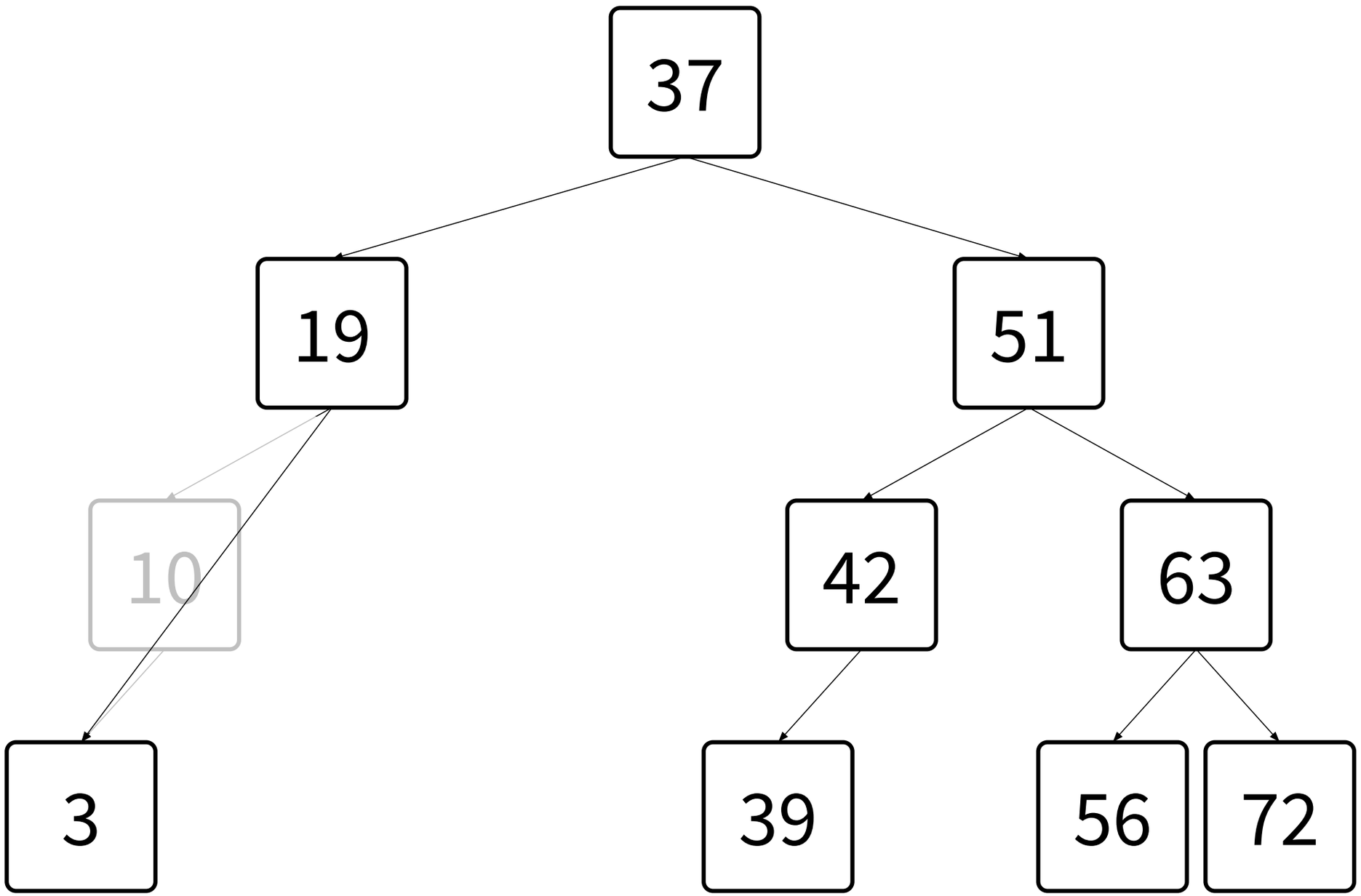
BST algorithm: delete

Zero children



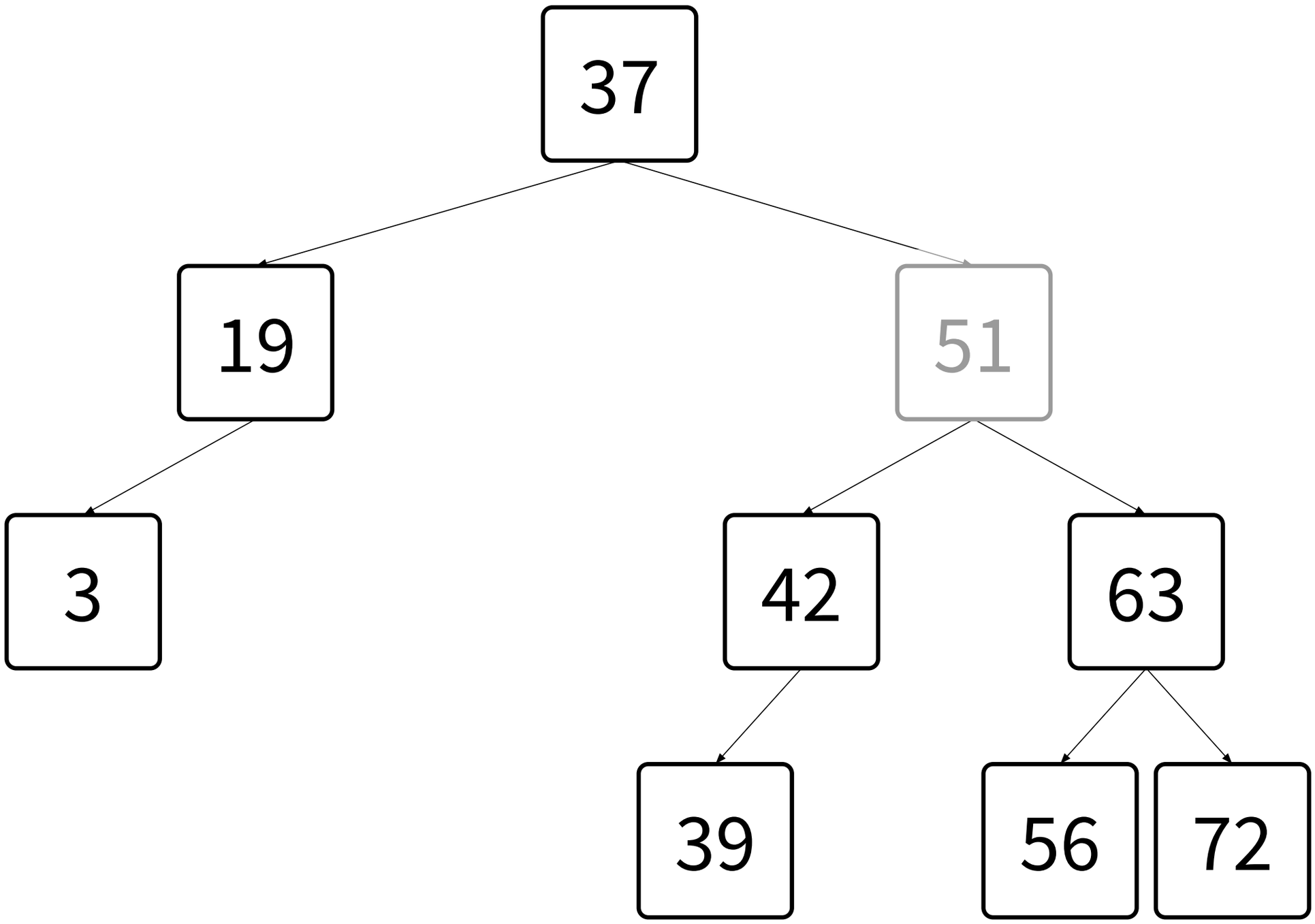
BST algorithm: delete

One child



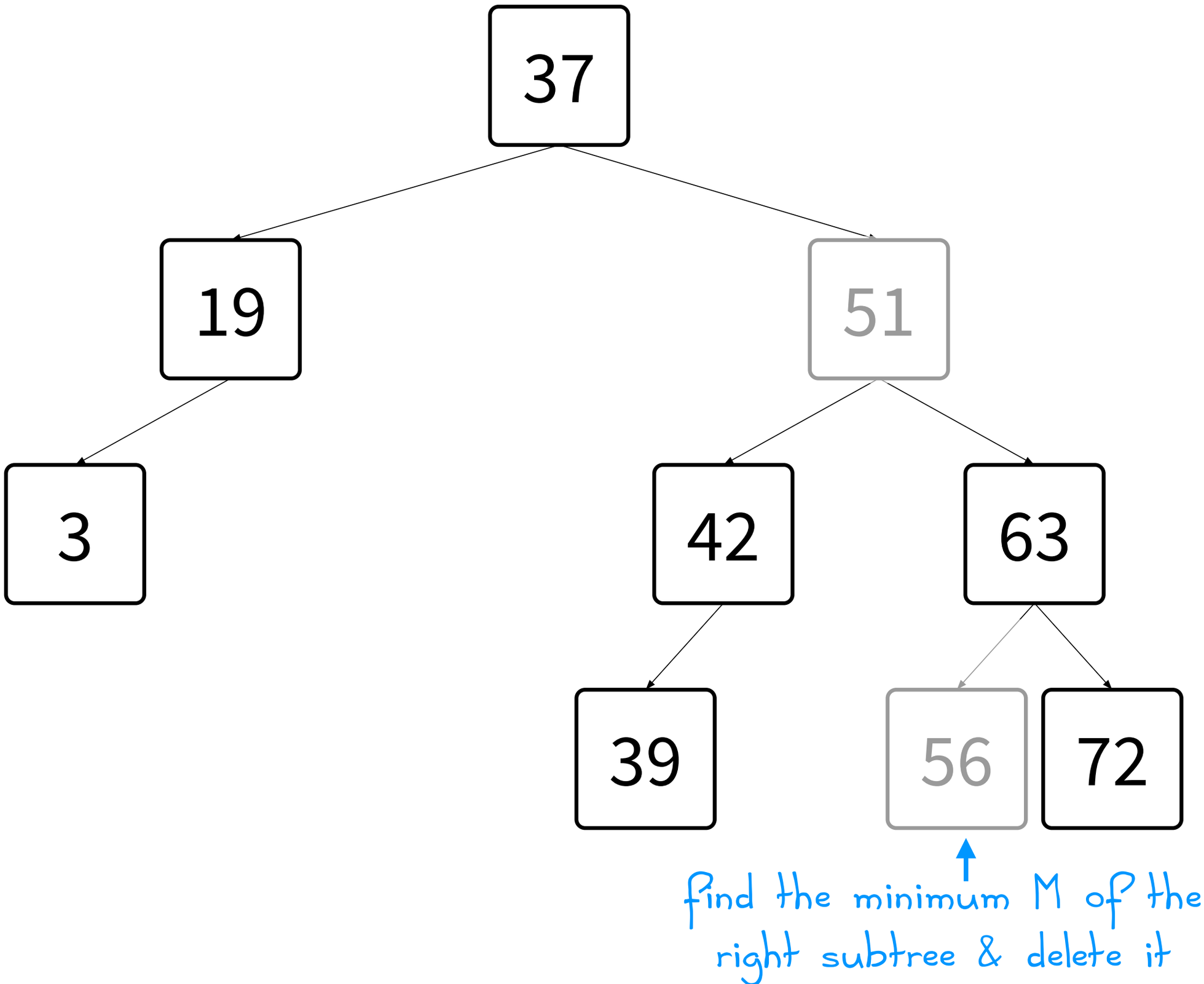
BST algorithm: delete

Two children



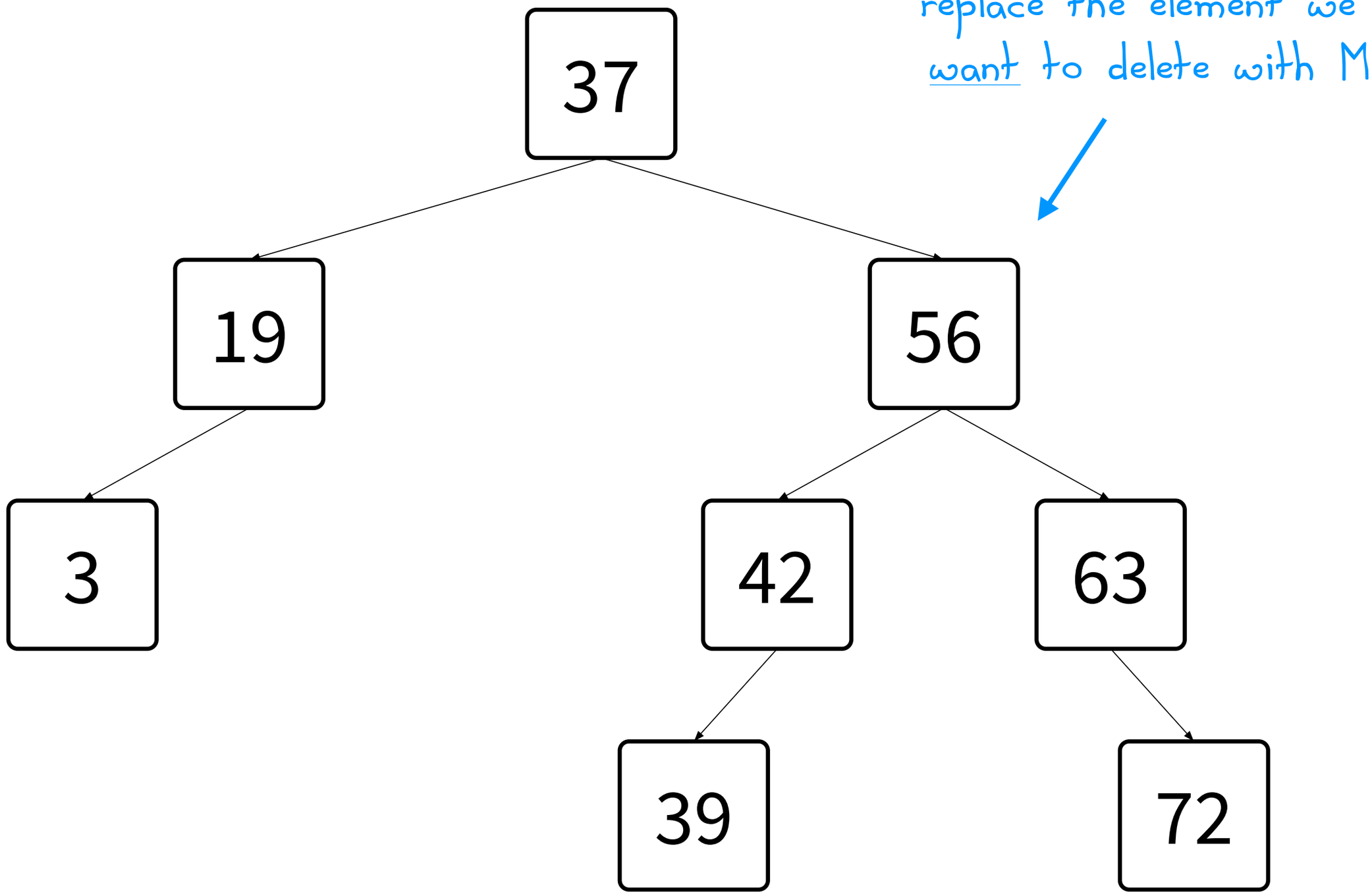
BST algorithm: delete

Two children



BST algorithm: delete

Two children



BST algorithm: delete

Given a BST *values* and a number *i*:

```
delete(i, values):
    If the tree is empty, do nothing.
    Otherwise, let key be the value at the root of the tree.
  If i < key, call delete on the left subtree.
  If i > key, call delete on the right subtree.
  If i = key:
    If the root has 1 child, that child is the new root.
    If the root has 2 children:
        1. Let m be the smallest value in the right subtree.
        2. Delete m from the right subtree.
        3. Make m the new root value.
```

upload.wikimedia.org/wikipedia/commons/8/8c/Key_break.jpg

BSTs in Racket

Designing and implementing a new data structure

Interface and implementation

- Interface

  Describes: **what** this data structure can do

- Implementation: encoding

  Describes: **how** the structure is stored, using existing data structures

- Implementation: operations

  Defines: **how** the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.

Our BST will be unbalanced and functional

Our Racket trees: Interface

Inductive data structure, manipulated via constructors, accessors, and operations

Constructors

```
(define (make-BST key left right)...
(define (make-empty-BST) ...
(define (make-BST-leaf key) ...
```

Read-only operations

```
(define (emptyTree? tree) ...
(define (leaf? tree) ...
(define (key tree) ...
(define (leftTree tree) ...
(define (rightTree tree) ...
```

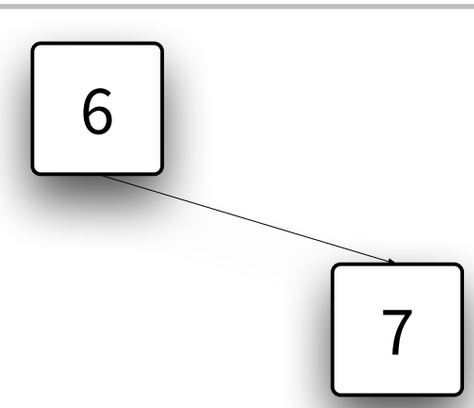
Our Racket trees: list encoding

Tree constructors

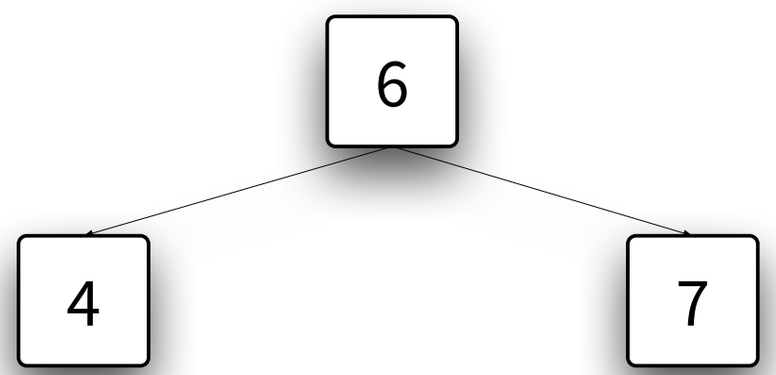
```
> (make-empty-BST)
'()
```

```
> (make-BST-leaf 3)
'(3 () ())
```


```
> (make-BST 6
            (make-empty-BST)
            (make-BST-leaf 7))
'(6 () (7 () ()))
```




```
> (make-BST 6
            (make-BST-leaf 4)
            (make-BST-leaf 7))
'(6 (4 () ()) (7 () ()))
```



Our Racket trees: Implementation

How would you encode a BST as a list?

```
(define (make-empty-BST)
                        )
```

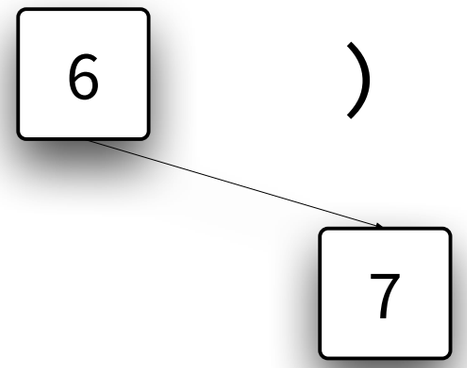
```
(define (make-BST key left right)
                                 )
```

Our Racket trees: list encoding

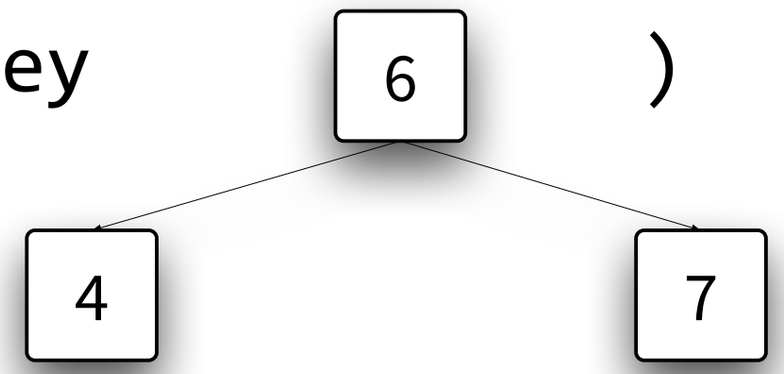
Tree operations

```
> (key  3 )   → 3
```

```
> (key  6 )   → 6
```




```
> (key  6 )   → 6
```



BSTs in Java

Designing and implementing a new data structure

Interface and implementation

- Interface

  Describes: **what** this data structure can do

- Implementation: encoding

  Describes: **how** the structure is stored, using existing data structures

- Implementation: operations

  Defines: **how** the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.

Our BST will be an unbalanced TreeMap

Interface

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`clear()`** Removes all of the mappings from this map (optional operation).
`boolean`	**`containsKey(Object key)`** Returns `true` if this map contains a mapping for the specified key.
`boolean`	**`containsValue(Object value)`** Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	**`entrySet()`** Returns a **`Set`** view of the mappings contained in this map.

Based on Java's `map` interface

bonus: generic trees!

Encoding

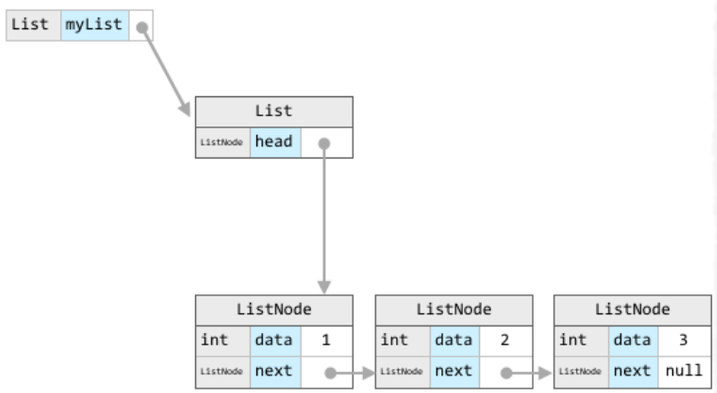
We divide the representation into two classes: `List` and `ListNode`

`List`

purpose	signature
a reference to the first element of the list	`ListNode head`

`ListNode`

purpose	signature
the list element	`int data`
a reference to the rest of the list	`ListNode next`



Based on our `LinkedList` encoding

using nodes

Implementation

This week's assignment :)

How can we use our BST?

We want to ask: “What’s X’s favorite number?”

for some person X

The **interface** is a map: `String` → `Integer`

```
favoriteNumbers.put("Chris", 30);
favoriteNumbers.put("Ben", 26);
favoriteNumbers.get("Chris");   // returns 30
```

We’ll **encode** the information as a BST of key/value pairs:

key = person’s name

value = person’s favorite number

Let's take this step-by-step

1. Encoding for a BST that stores numbers
   Based on our previous encoding for a linked list that stores numbers
2. Encoding for a BST that stores pairs: name + number
   So we can ask for someone's favorite number
3. Bonus: Encoding for a BST that stores arbitrary pairs
   So we can can use this data structure to store lots of things
4. Giving our BST a map-like interface
   So that it behaves (almost) like the built-in map!

Let's take this step-by-step

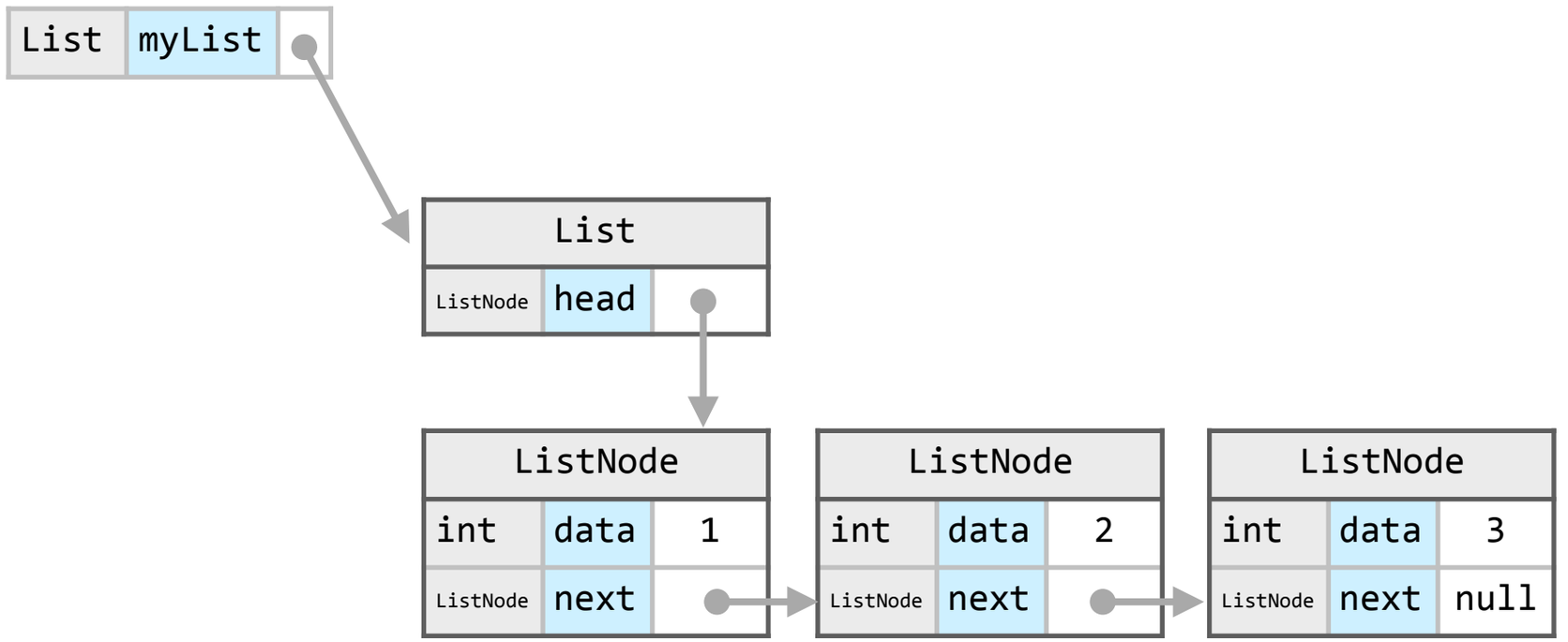
1. Encoding for a BST that stores numbers
   Based on our previous encoding for a linked list that stores numbers
2. Encoding for a BST that stores pairs: name + number
   So we can ask for someone's favorite number
3. Bonus: Encoding for a BST that stores arbitrary pairs
   So we can can use this data structure to store lots of things
4. Giving our BST a map-like interface
   So that it behaves (almost) like the built-in map!

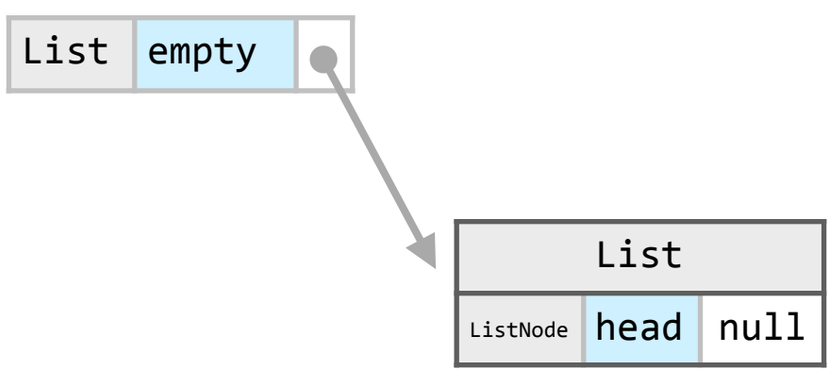
Recall: a linked list's representation

We divide the representation into two classes: `List` and `ListNode`

`List`	
purpose	**signature**
a reference to the first element of the list	`ListNode head`

`ListNode`	
purpose	**signature**
the list element	`int data`
a reference to the rest of the list	`ListNode next`



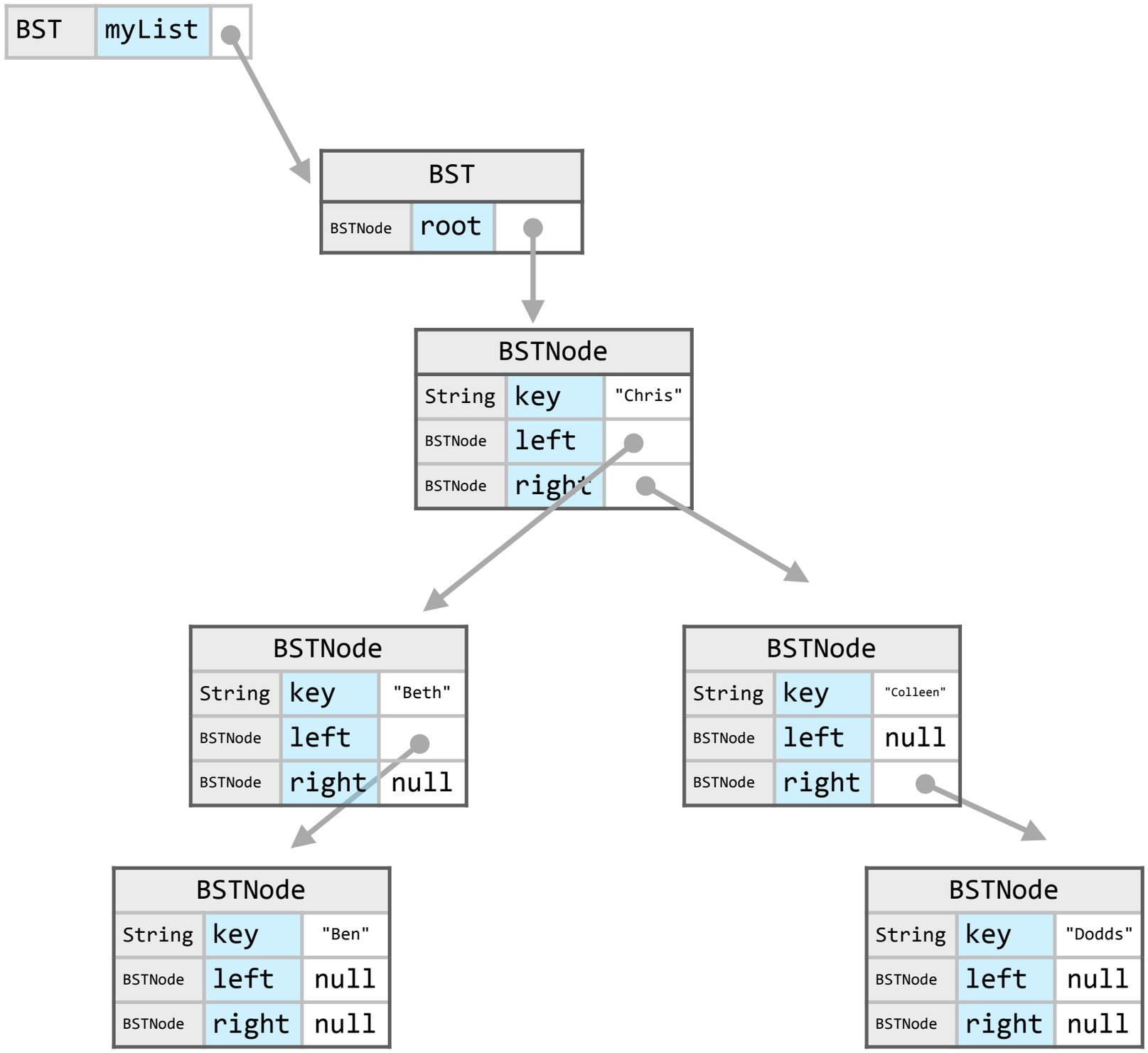


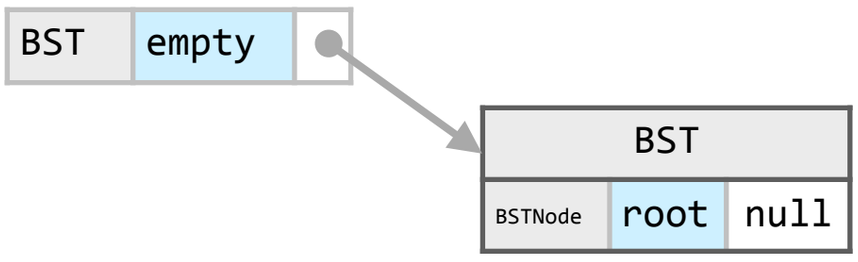
A BST's representation

We divide the representation into two classes: BST and BSTNode

BST	
purpose	**signature**
a reference to the top of the BST	`BSTNode root`

BSTNode	
purpose	**signature**
the node element	`String key`
a reference to the left subtree	`BSTNode left`
a reference to the right subtree	`BSTNode right`





Let's take this step-by-step

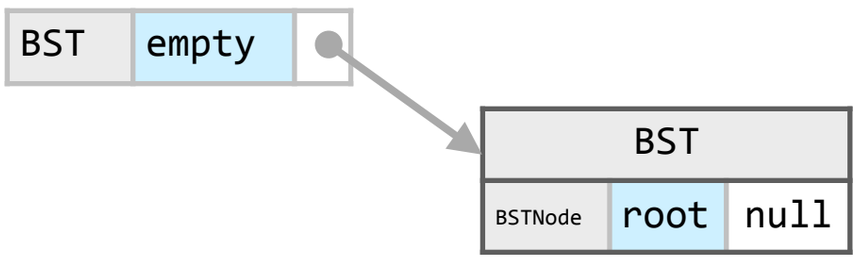
1. Encoding for a BST that stores numbers
   Based on our previous encoding for a linked list that stores numbers
2. Encoding for a BST that stores pairs: name + number
   So we can ask for someone's favorite number
3. Bonus: Encoding for a BST that stores arbitrary pairs
   So we can can use this data structure to store lots of things
4. Giving our BST a map-like interface
   So that it behaves (almost) like the built-in map!

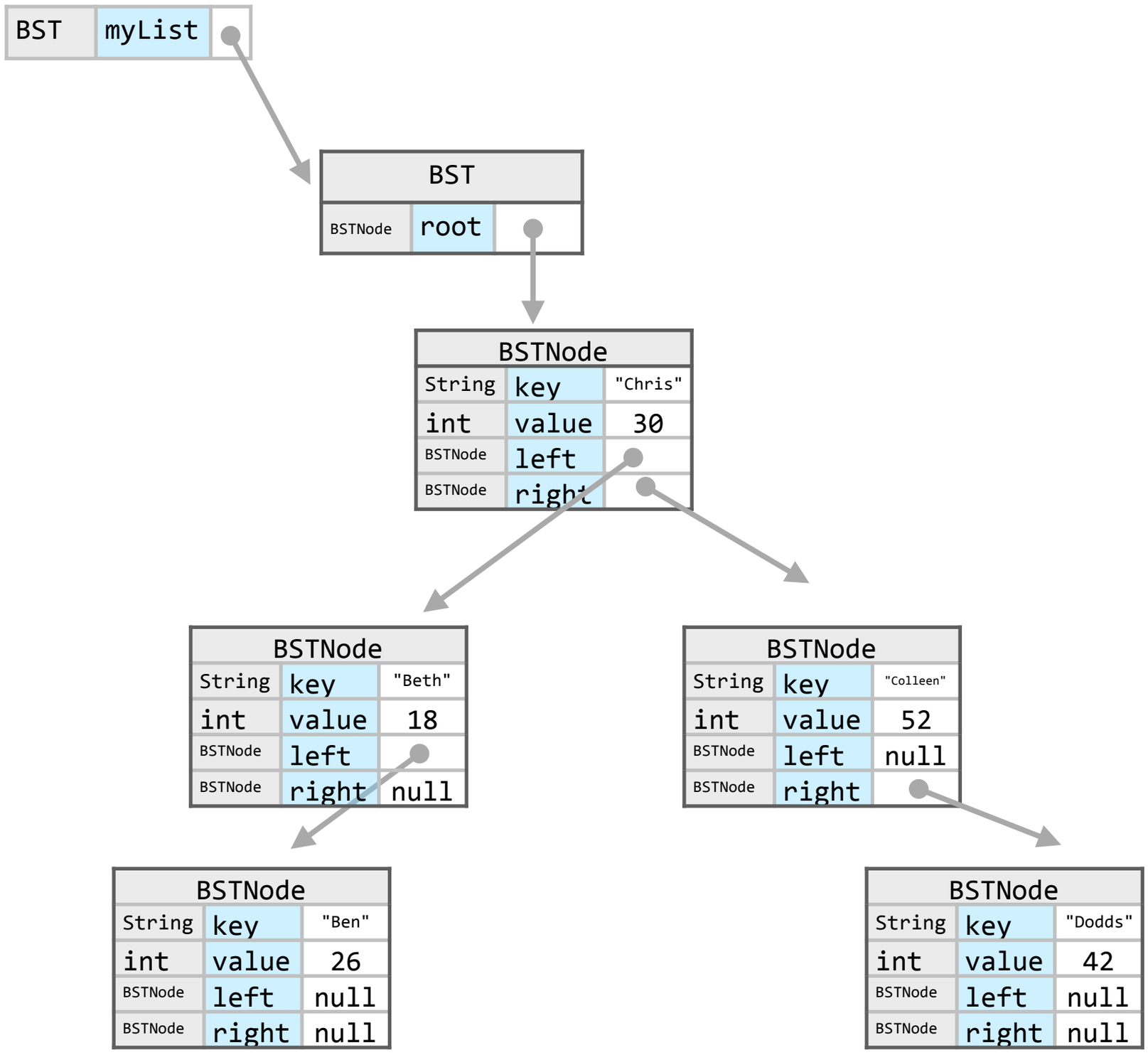
A BST's representation

We divide the representation into two classes: BST and BSTNode

BST	
purpose	**signature**
a reference to the top of the BST	`BSTNode root`

BSTNode	
purpose	**signature**
used for search	`String key`
the data to store	`int value`
a reference to the left subtree	`BSTNode left`
a reference to the right subtree	`BSTNode right`





Our code so far...

```
public class UnbalancedBSTMap {

    /** a reference to the root of the tree (null if the tree is empty) **/
    BSTNode rootNode;

    private class BSTNode {
        private String key;
        private Integer value;
        private BSTNode leftTree;
        private BSTNode rightTree;

        private BSTNode(String inputKey, Integer inputValue,
                        BSTNode inputLeftT, BSTNode inputRightT) {
            if (inputKey == null || inputValue == null) {
                throw new IllegalArgumentException(
                        "Inserted keys and values must be non-null");
            }
            this.key = inputKey;
            this.value = inputValue;
            this.leftTree = inputLeftT;
            this.rightTree = inputRightT;
        }
    }
}
```

Let's take this step-by-step

1. Encoding for a BST that stores numbers
   Based on our previous encoding for a linked list that stores numbers
2. Encoding for a BST that stores pairs: name + number
   So we can ask for someone's favorite number
3. Bonus: Encoding for a BST that stores arbitrary pairs
   So we can can use this data structure to store lots of things
4. Giving our BST a map-like interface
   So that it behaves (almost) like the built-in map!

An inflexible BSTNode

We can only map `Strings` to `Integers`

```
public class UnbalancedBSTMap {

    /** a reference to the root of the tree (null if the tree is empty) **/
     BSTNode rootNode;

     private class BSTNode {
         private String key;
         private Integer value;
         private BSTNode leftTree;
         private BSTNode rightTree;

         private BSTNode(String inputKey, Integer inputValue,
                         BSTNode inputLeftT, BSTNode inputRightT) {
             if (inputKey == null || inputValue == null) {
                 throw new IllegalArgumentException(
                         "Inserted keys and values must be non-null");
             }
             this.key = inputKey;
             this.value = inputValue;
             this.leftTree = inputLeftT;
             this.rightTree = inputRightT;
         }
     }
}
```

A flexible BSTNode

Now we can store lots of kinds of things!

type parameters
aka "generics"

```
public class UnbalancedBSTMap<KeyType, ValueType> {

    /** a reference to the root of the tree (null if the tree is empty) **/
     BSTNode rootNode;

     private class BSTNode {
         private KeyType key;
         private ValueType value;
         private BSTNode leftTree;
         private BSTNode rightTree;

         private BSTNode(KeyType inputKey, ValueType inputValue,
                         BSTNode inputLeftT, BSTNode inputRightT) {
             if (inputKey == null || inputValue == null) {
                 throw new IllegalArgumentException(
                         "Inserted keys and values must be non-null");
             }
             this.key = inputKey;
             this.value = inputValue;
             this.leftTree = inputLeftT;
             this.rightTree = inputRightT;
         }
     }
}
```

```
UnbalancedBSTMap<Integer, String> numbers =
    new UnbalancedBSTMap<Integer, String>();
```

A flexible BSTNode

But we can't compare keys :(

```
public class UnbalancedBSTMap<KeyType, ValueType> {

    /** a reference to the root of the tree (null if the tree is empty) **/
     BSTNode rootNode;

     private class BSTNode {
         ...
     }
}
```

```
find(i, values):
   If the tree is empty, return false.
   Let key be the value at the root of the tree.
   If key is i, return true.
   If i < key, call find on the left subtree.
   If i > key, call find on the right subtree.
```

```
String s1 = "Ben";
String s2 = "Chris";
System.out.println(s1 < s2);
```

The operator < is undefined for the argument type(s) java.lang.String, java.lang.String

Java's `Comparable` interface to the rescue!

We use the `compareTo` method to determine <, ==, or >

```
public class UnbalancedBSTMap<KeyType, ValueType> {

    /** a reference to the root of the tree (null if the tree is empty) **/
     BSTNode rootNode;

     private class BSTNode {
         ...
     }
}
```

```
find(i, values):
   If the tree is empty, return false.
   Let key be the value at the root of the tree.
   If key is i, return true.
   If i < key, call find on the left subtree.
   If i > key, call find on the right subtree.
```

```
String s1 = "Ben";
String s2 = "Chris";
System.out.println(s1.compareTo(s2));
```

A flexible BSTNode

Now we can store lots of kinds of comparable things!

```
public class UnbalancedBSTMap<KeyType extends Comparable<KeyType>, ValueType> {

    /** a reference to the root of the tree (null if the tree is empty) **/
    BSTNode rootNode;

    private class BSTNode {
        ...
    }

    private boolean inOrderKeys(KeyType key1, KeyType key2) {
        return key1.compareTo(key2) < 0;
    }
}
```

```
find(i, values):
    If the tree is empty, return false.
    Let key be the value at the root of the tree.
    If key is i, return true.
    If inOrderKeys(i, key), call find on the left subtree.
    If inOrderKeys(key, i), call find on the right subtree.
```

Let's take this step-by-step

1. Encoding for a BST that stores numbers
   Based on our previous encoding for a linked list that stores numbers
2. Encoding for a BST that stores pairs: name + number
   So we can ask for someone's favorite number
3. Bonus: Encoding for a BST that stores arbitrary pairs
   So we can can use this data structure to store lots of things
4. Giving our BST a map-like interface
   So that it behaves (almost) like the built-in map!

We want our tree to behave like a map

```
public class UnbalancedBSTMap<KeyType extends Comparable<KeyType>, ValueType> {

    /** a reference to the root of the tree (null if the tree is empty) **/
     BSTNode rootNode;

     private class BSTNode {
         ...
     }
}
```

Java's Map interface

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`clear()`** Removes all of the mappings from this map (optional operation).
`boolean`	**`containsKey(Object key)`** Returns `true` if this map contains a mapping for the specified key.
`boolean`	**`containsValue(Object value)`** Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	**`entrySet()`** Returns a **`Set`** view of the mappings contained in this map.
`boolean`	**`equals(Object o)`** Compares the specified object with this map for equality.
`V`	**`get(Object key)`** Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the ke

We want our tree to behave like a map

So our class needs to implement the Map interface

```
public class UnbalancedBSTMap<KeyType extends Comparable<KeyType>, ValueType>
    implements Map<KeyType, ValueType> {

    /** a reference to the root of the tree (null if the tree is empty) **/
     BSTNode rootNode;

     private class BSTNode {
         ...
     }
     ...
     @Override
     public boolean isEmpty() {
         // TODO Auto-generated method stub
         return false;
     }

     @Override
     public ValueType get(Object key) {
         // TODO Auto-generated method stub
         return null;
     }

     ...
}
```

The next assignment

- Given a partial tree implementation in Racket, write a few more tree operations (including remove!)
- Provide a Big-O analysis of some of your Racket code
- Given a partial tree implementation in Java, write several more tree operations (including remove!)

Start Early.

Read everything.

Ask questions.