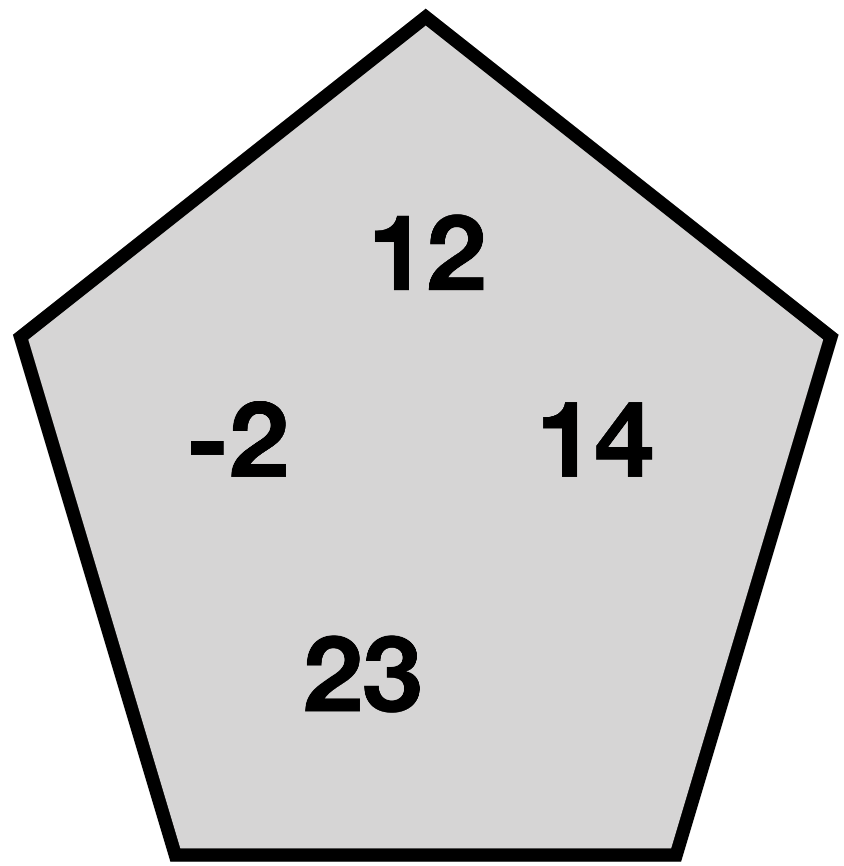
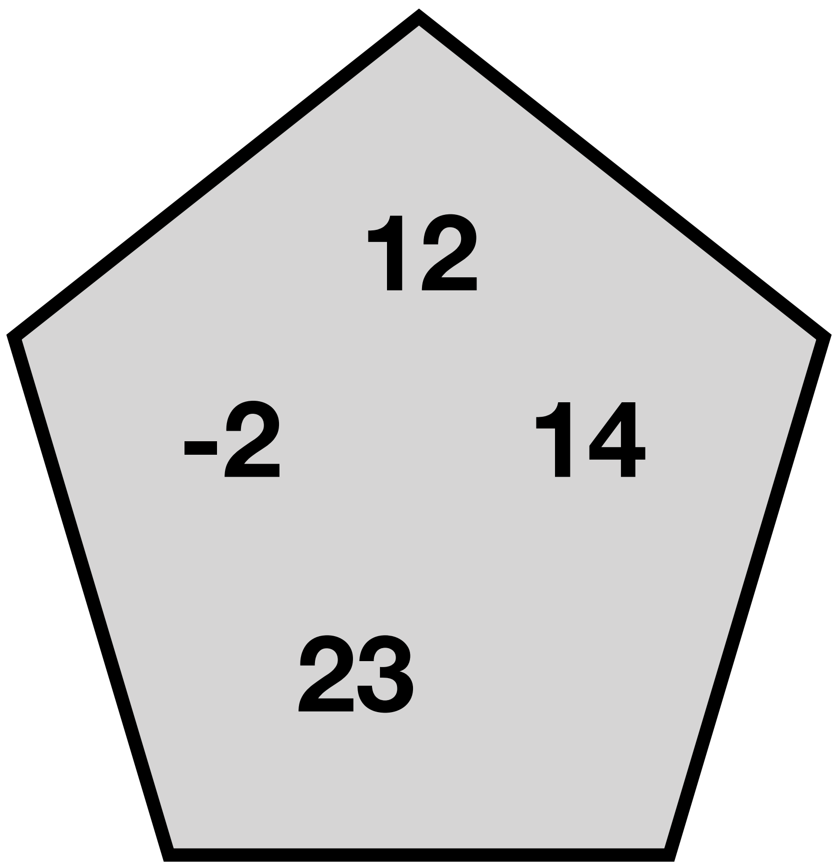
Binary Search Trees

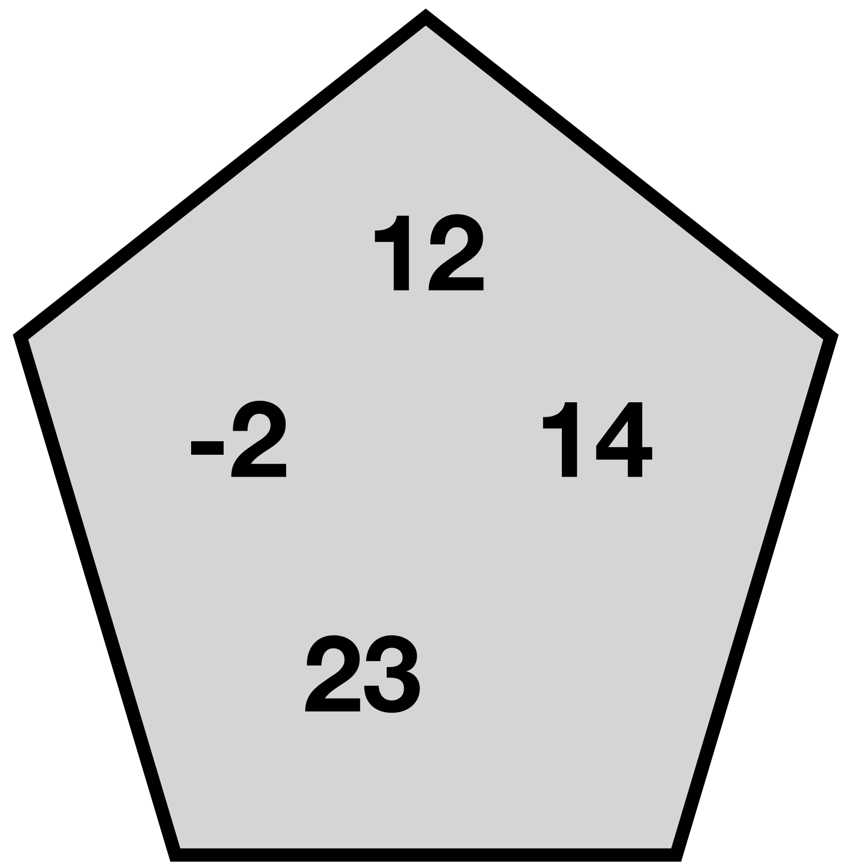
Let's say that I have a collection of numbers…



There are a few fundamental operations I can perform

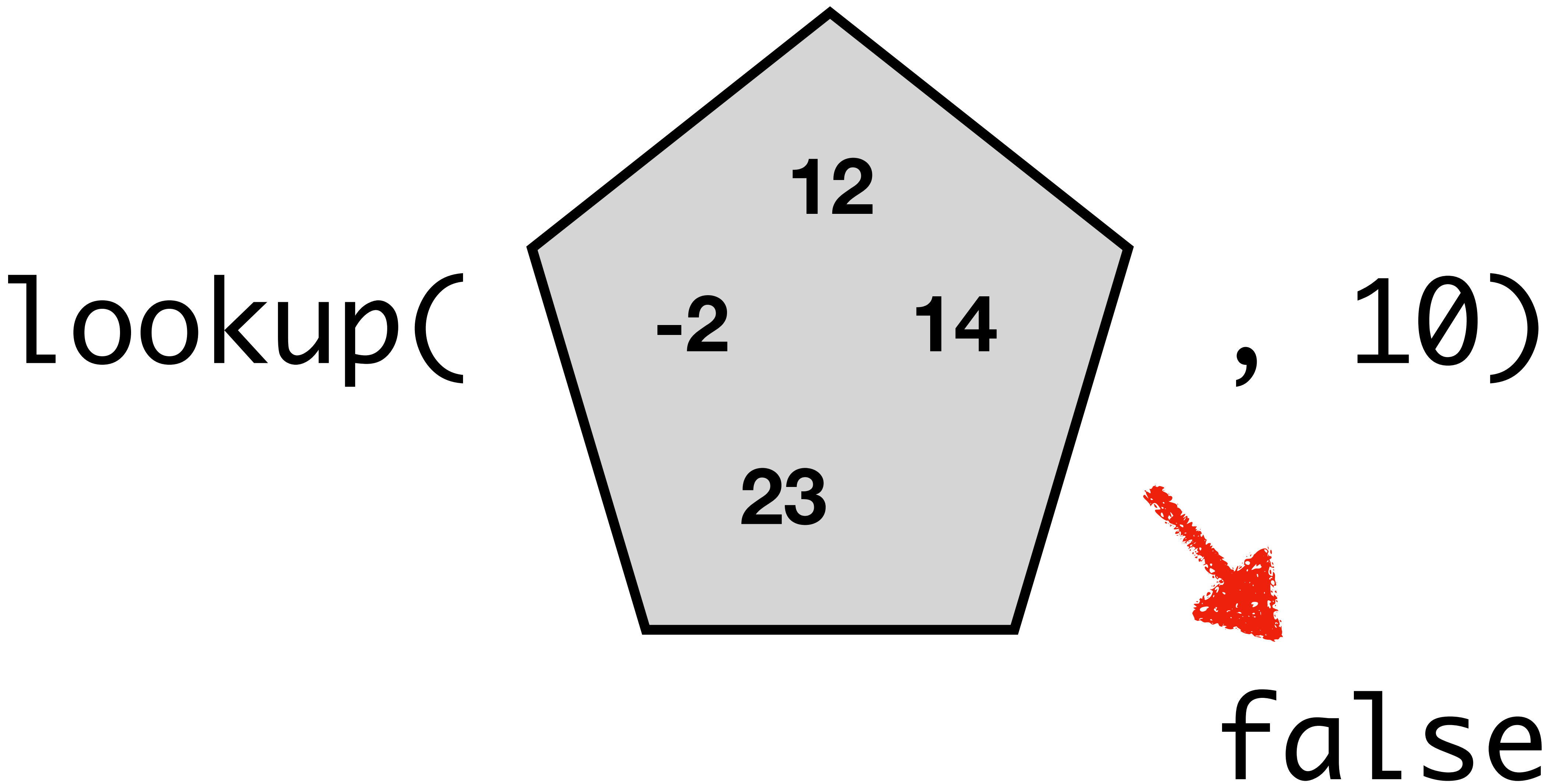


I can **lookup** an item to see if it exists

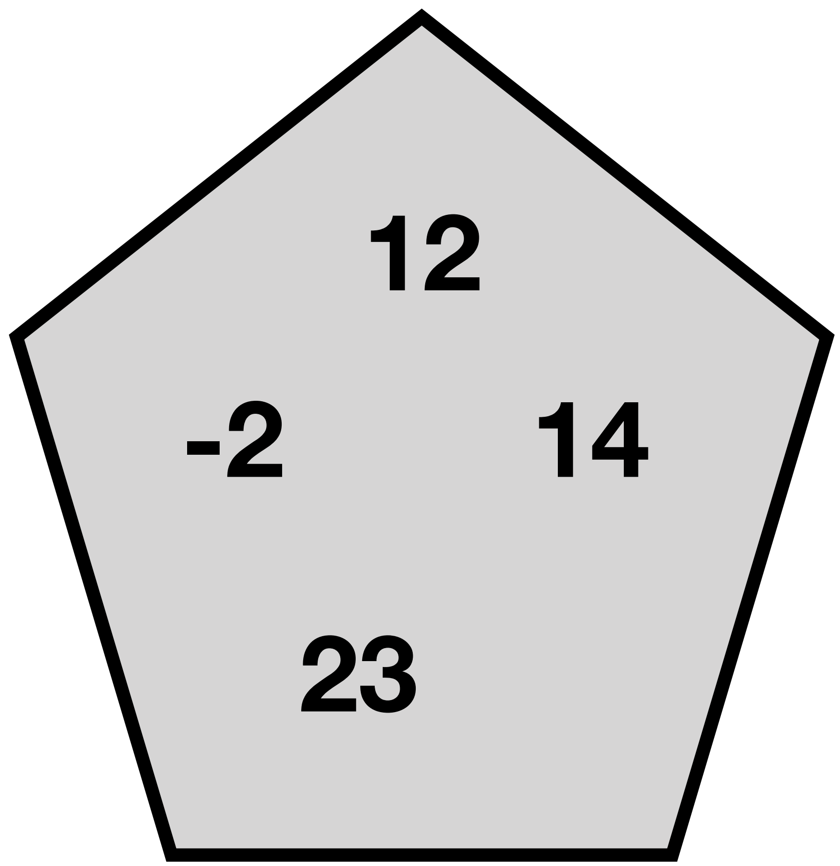
lookup( , 12)

true

I can **lookup** an item to see if it exists

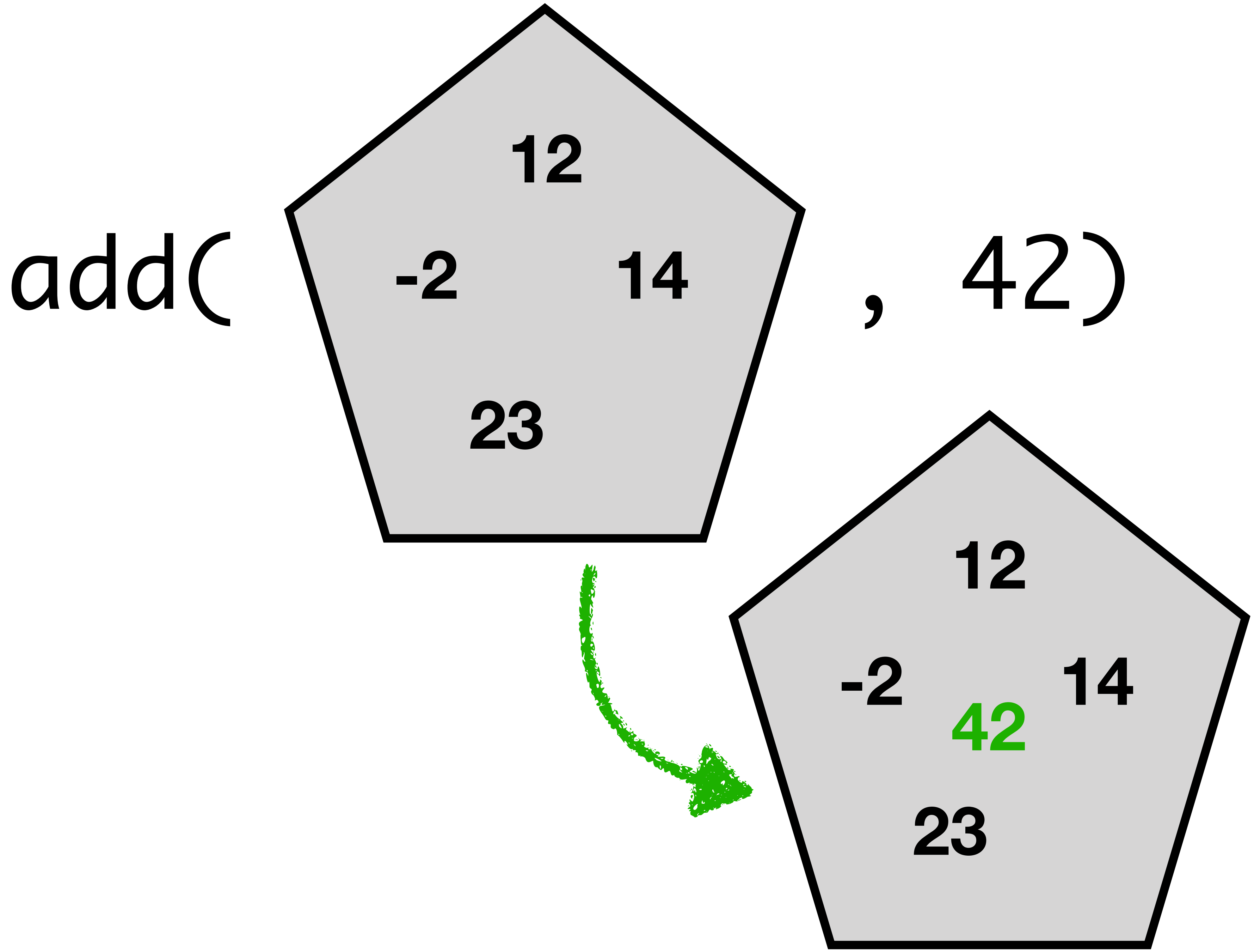


I can **add** an item to the collection

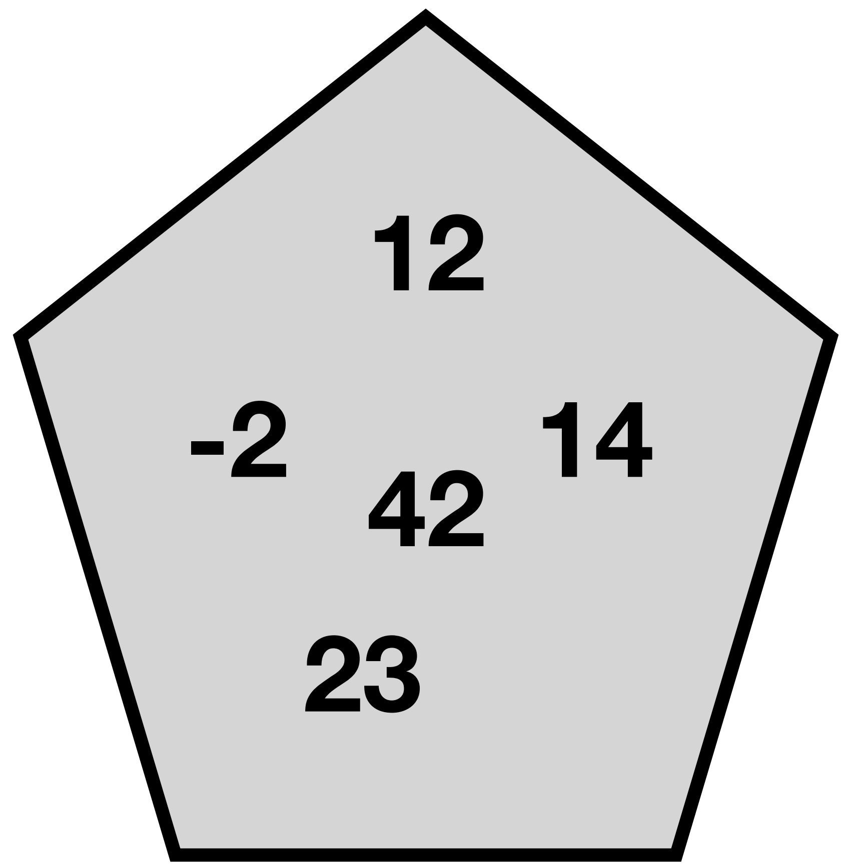
add(

, 42)

I can **add** an item to the collection



For now, I haven't talked about how we actually **implement** the collection



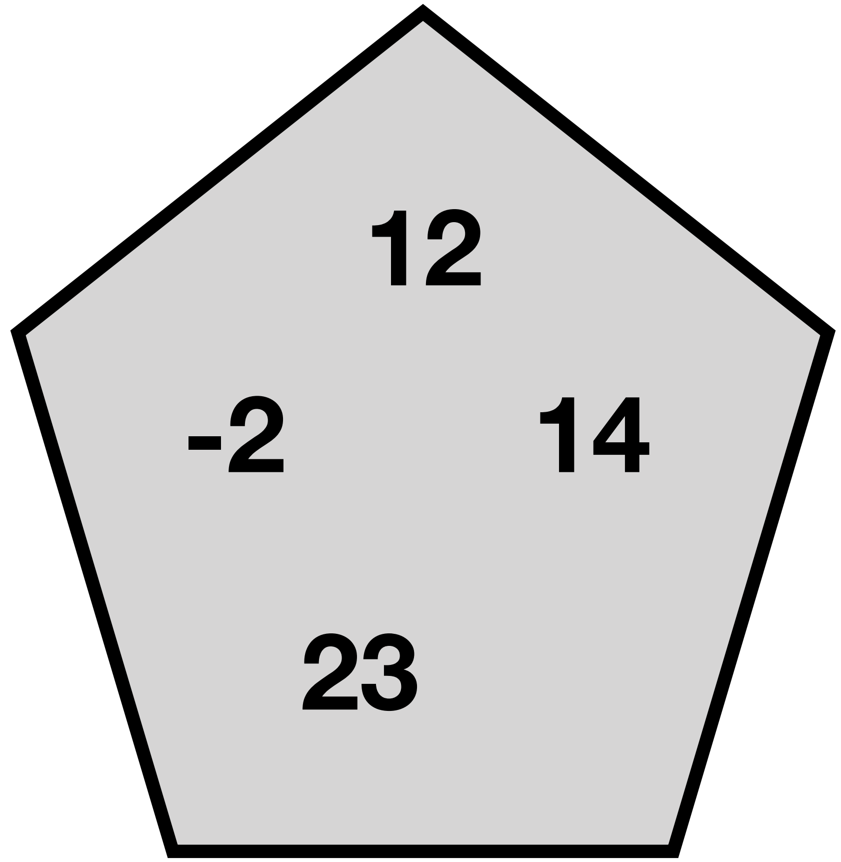
Only it's **specification**

Ask yourself: what's the

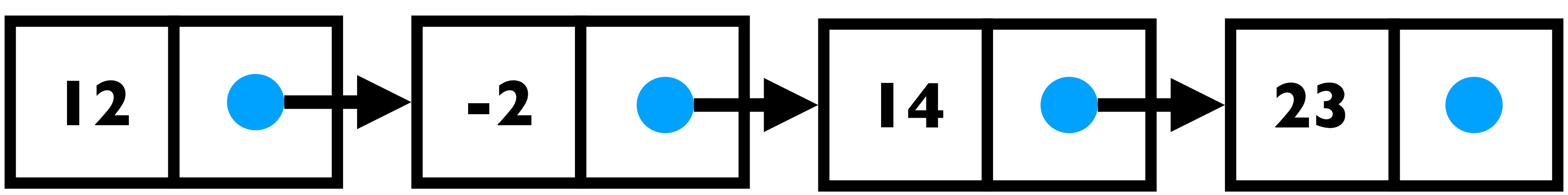
simplest possible implementation?

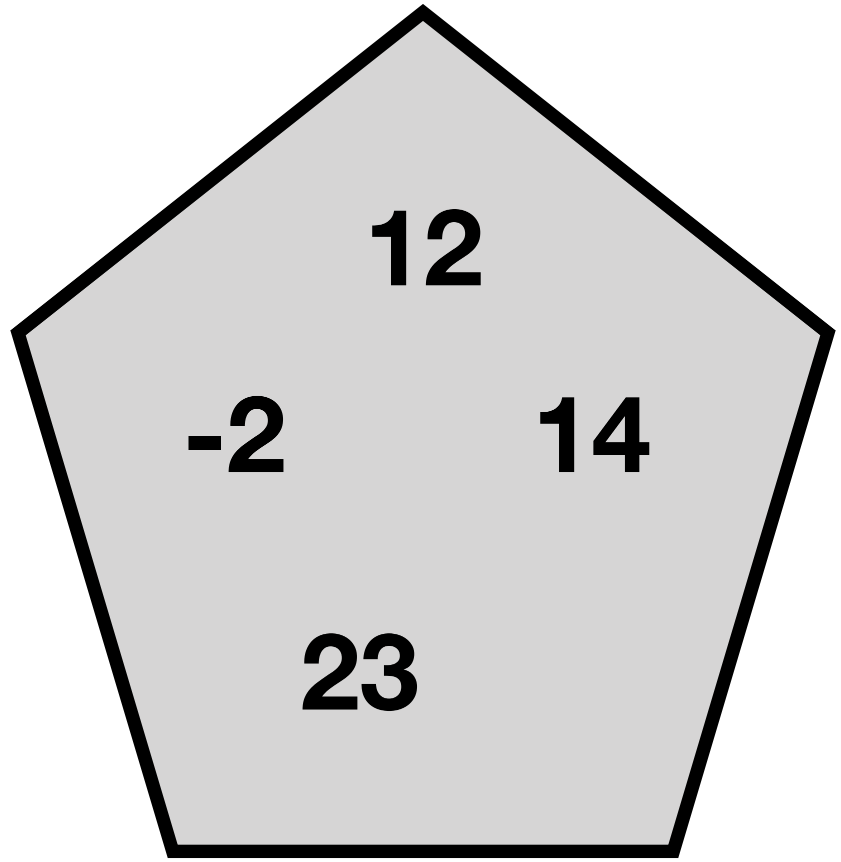
Ask yourself: what’s the
simplest possible implementation?

For me that would be a **list**



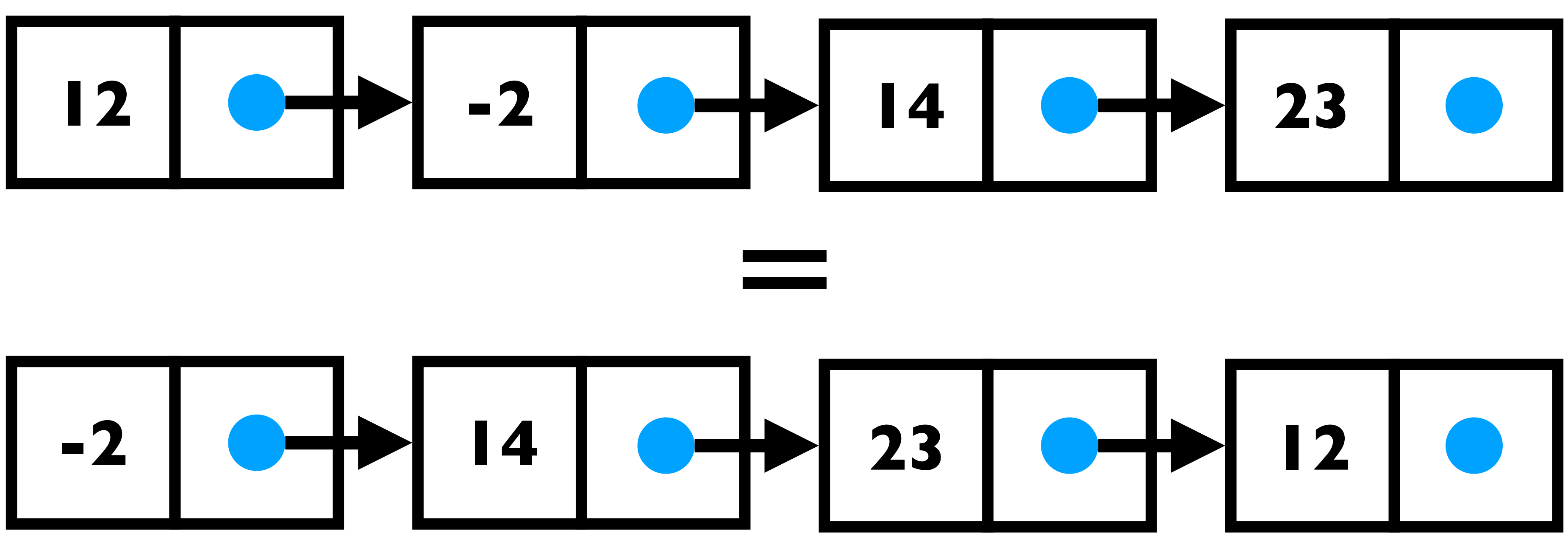
Implemented as a list…





Note: collection is unordered!

Implemented as a list...



Can implement these two operations

```
 def insert(l,n):
     return l.insert(0,n) # Extend front

def lookup(l,n):
    for i in l:
       if (i == n): return true # Found!
    return false
```

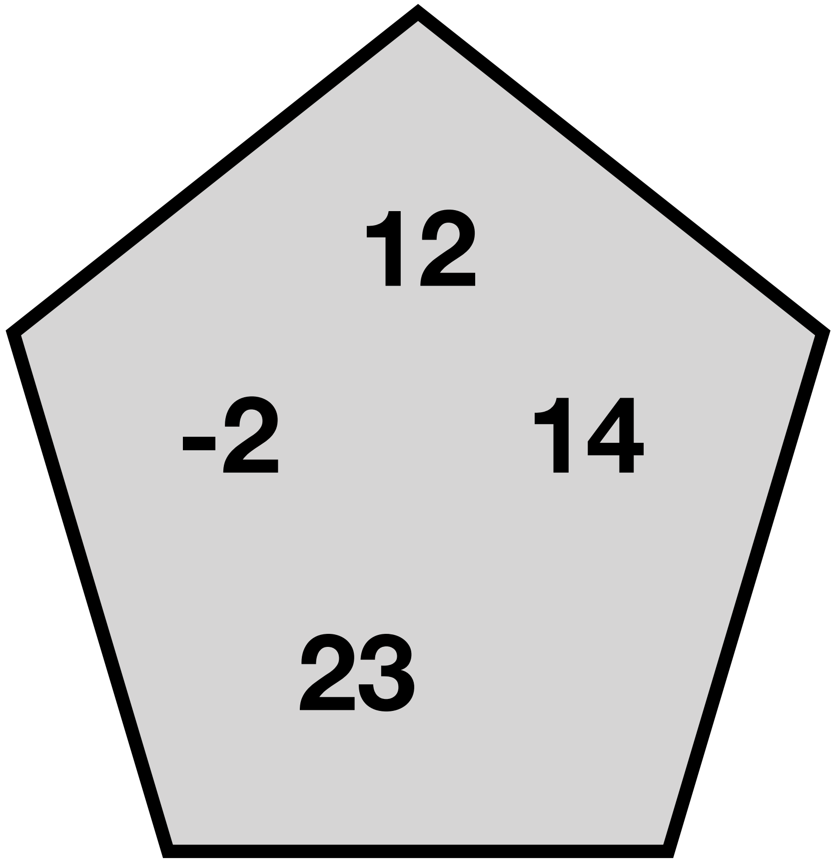
Can implement these two operations

O(1)

```
def insert(l,n):
    return l.insert(0,n) # Extend front
```

O(n)

```
def lookup(l,n):
    for i in l:
        if (i == n): return true # Found!
    return false
```



Implementation: list

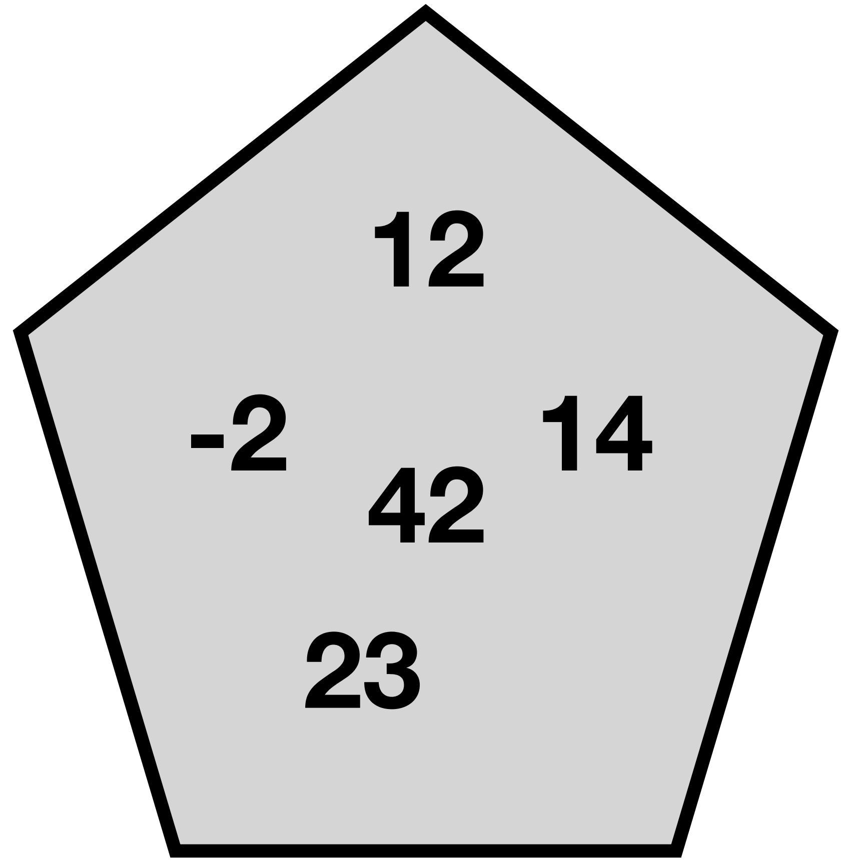
Operations	
`insert`	O(1)
`lookup`	O(n)

Can we do better for lookup?

Answer: **yes**, using slightly smarter data-structure

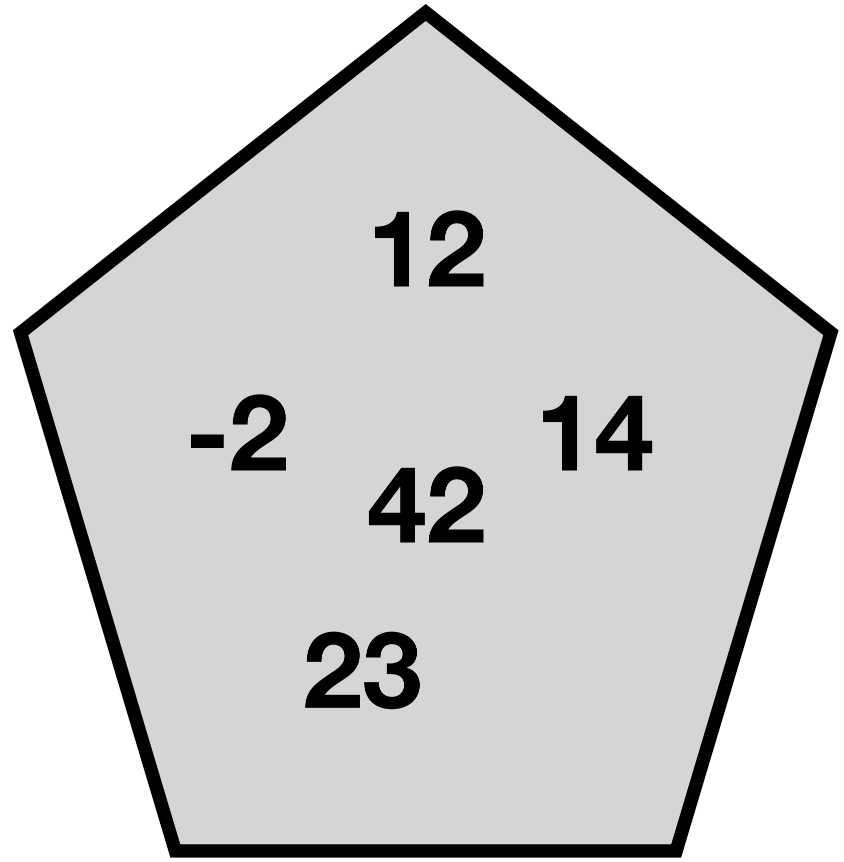
But all data structures have trade-offs

First some intuition…

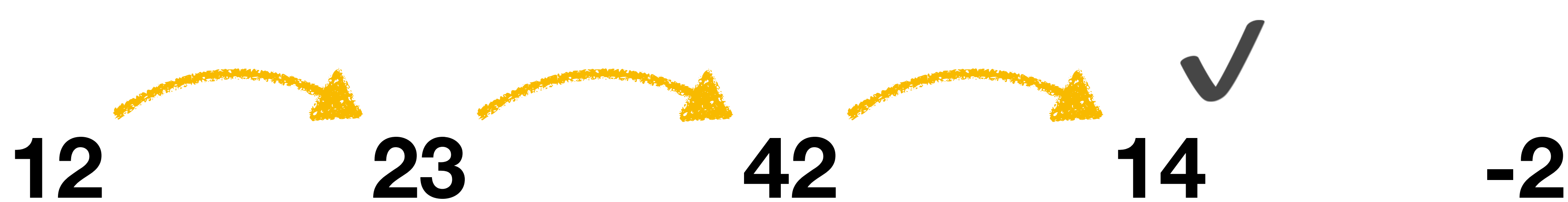


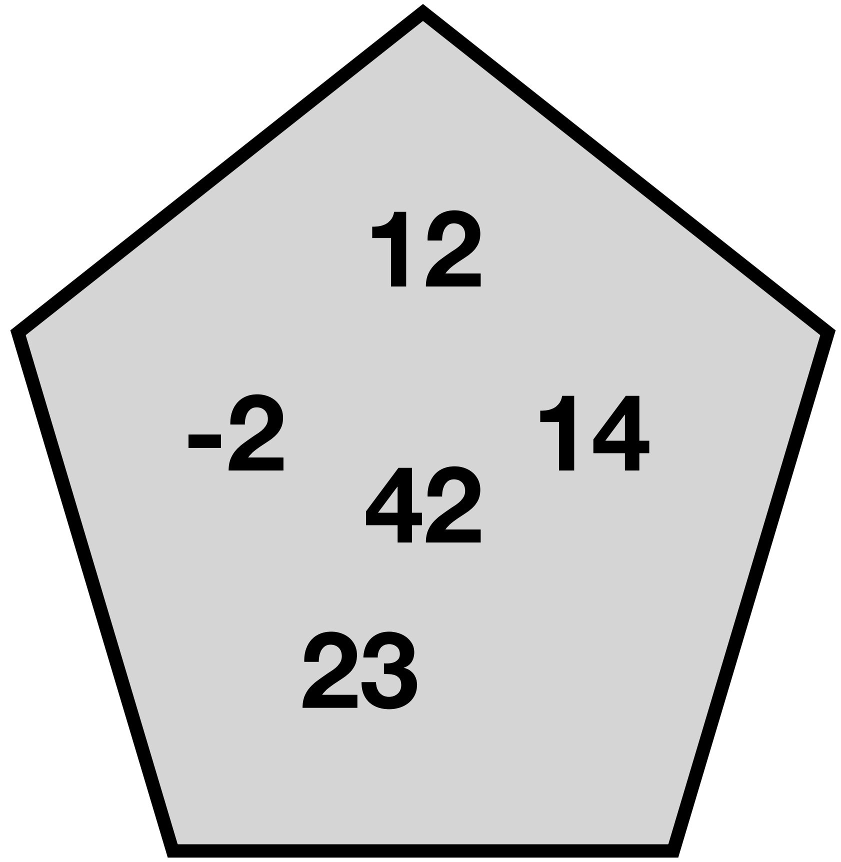
When I lookup, I go through list one-by-one…

12	**23**	**42**	**14**	**-2**

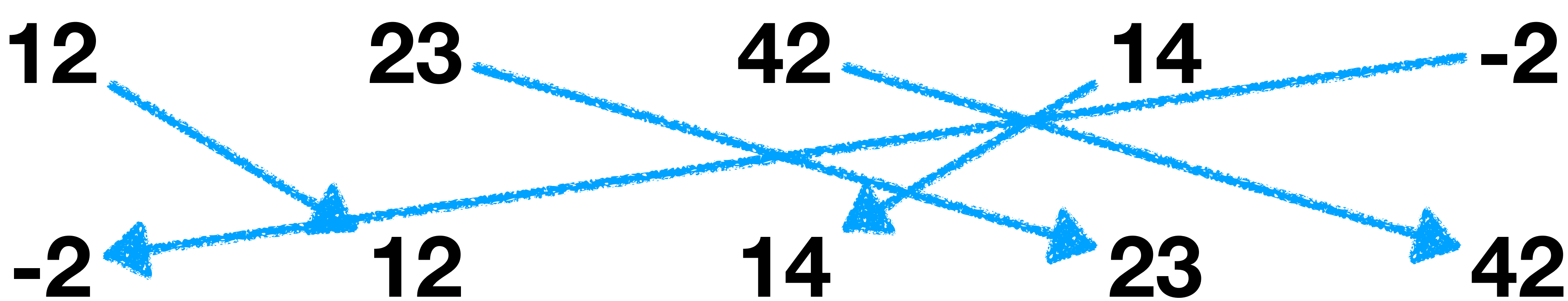


When I lookup, I go through list one-by-one…





The *crucial trick* is to **sort**



Is 12 in this sequence?

Start looking here

-2	**12**	**14**	**23**	**42**

Is 12 in this sequence?

Start looking here

-2	**12**	**14**	**23**	**42**

14 > 12, so **I know it must be in lower half**

Is 12 in this sequence?

Keep looking here…

-2	**12**	**14**	**23**	**42**

To lookup i in a sequence l:

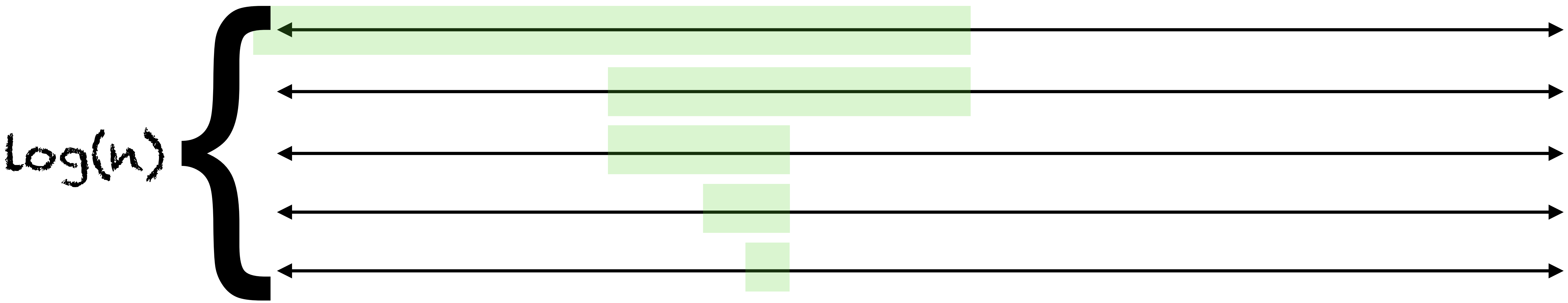
- Start at the middle element, call it j
  - If i = j, then return true
  - If i < j, then repeat with the lower half of the list
  - If i > j, then repeat with the upper half

Logarithmic work! Every step eliminates **half**!

To lookup i in a list l:

- Start at the middle element, call it j
  - If i = j, then return true
  - If i < j, then repeat with the lower half of the list
  - If i > j, then repeat with the upper half

Logarithmic work! Every step eliminates **half**!*



Logarithmic work! Every step eliminates **half**!*

*But if we use a list, we **don't** get log(n) lookup!

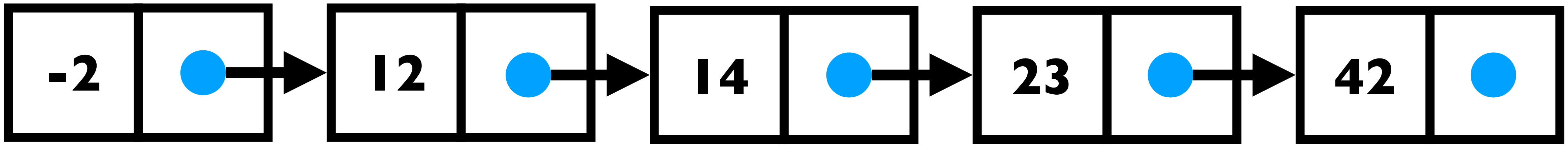
Lists aren't **random access**, so first step takes linear time!

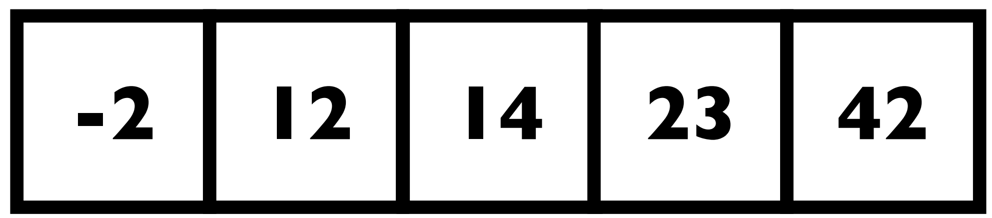
To lookup i in a sequenced:

- Start at the middle element, call it j
  - If i = j, then return true
  - If i < j, then repeat with the lower half of the list
  - If i > j, then repeat with the upper half

Therefore, actually n! :(

One possible solution: represent as an **array**

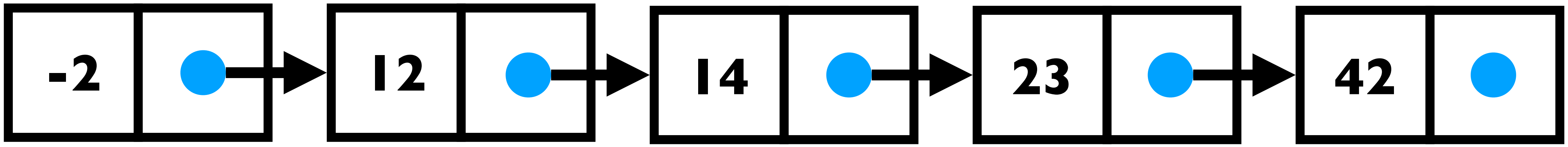




Arrays **are** random access

`lookup` now log(n)!

One possible solution: represent as an **array**



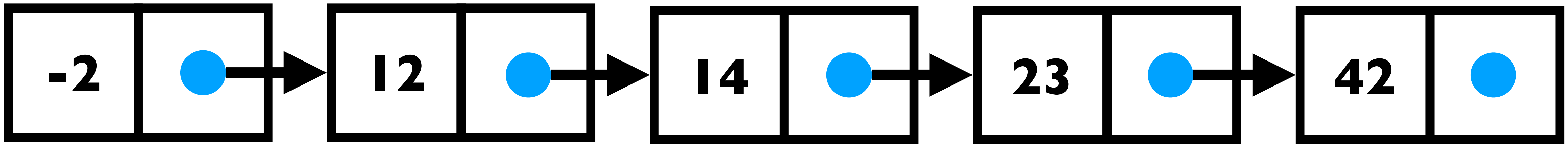
-2	12	14	23	42

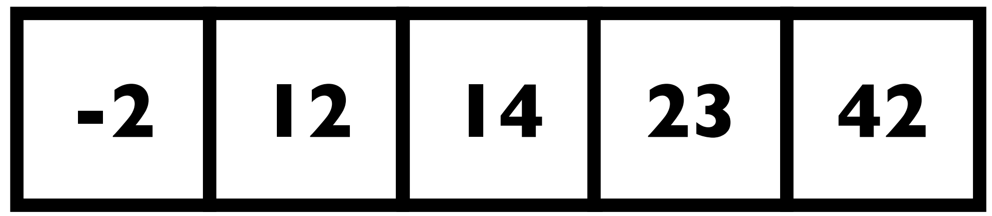
Downside: insertions must copy the **whole array**

Insert 13

-2	12	14	23	42

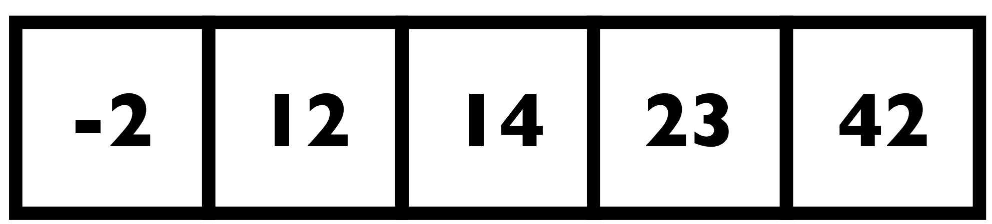
One possible solution: represent as an **array**

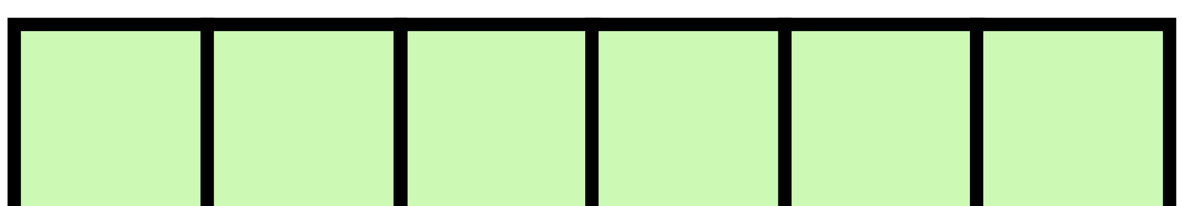




Downside: insertions must copy the **whole array**

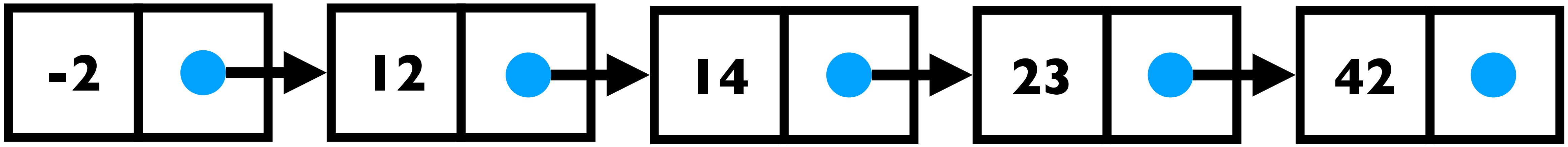
Insert 13

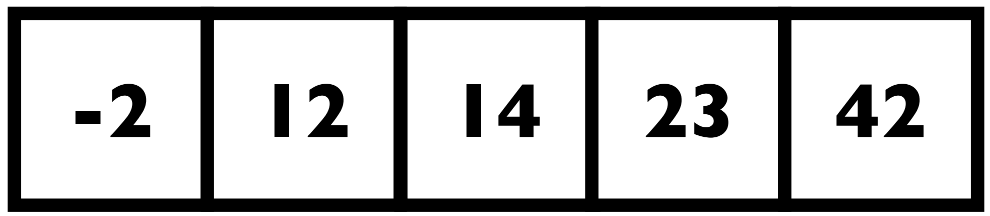




Newly allocated space

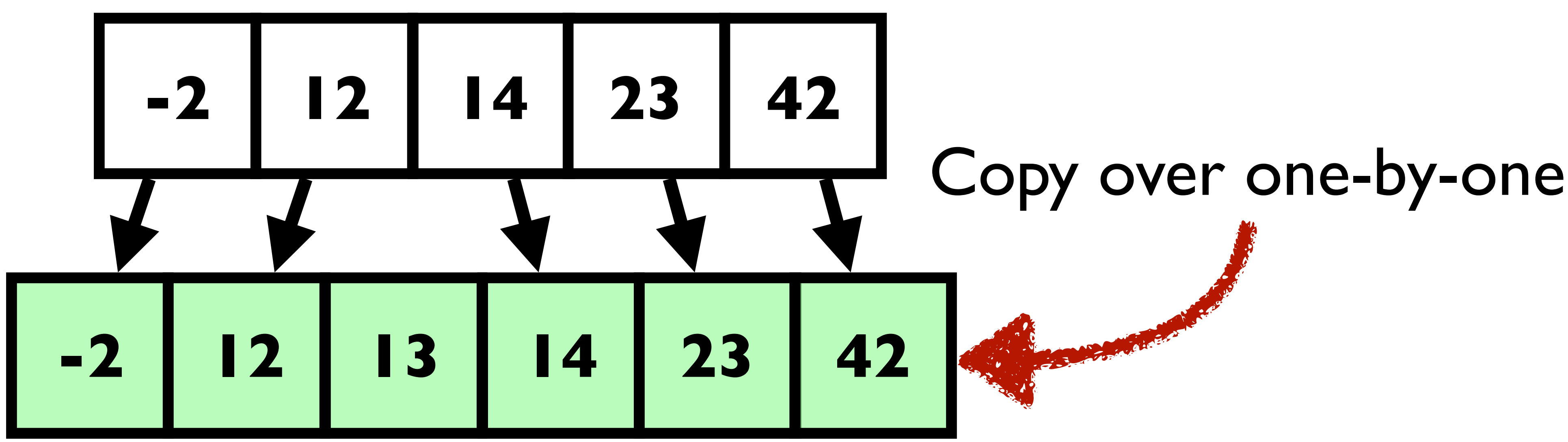
One possible solution: represent as an **array**





Downside: insertions must copy the **whole array**

Insert 13



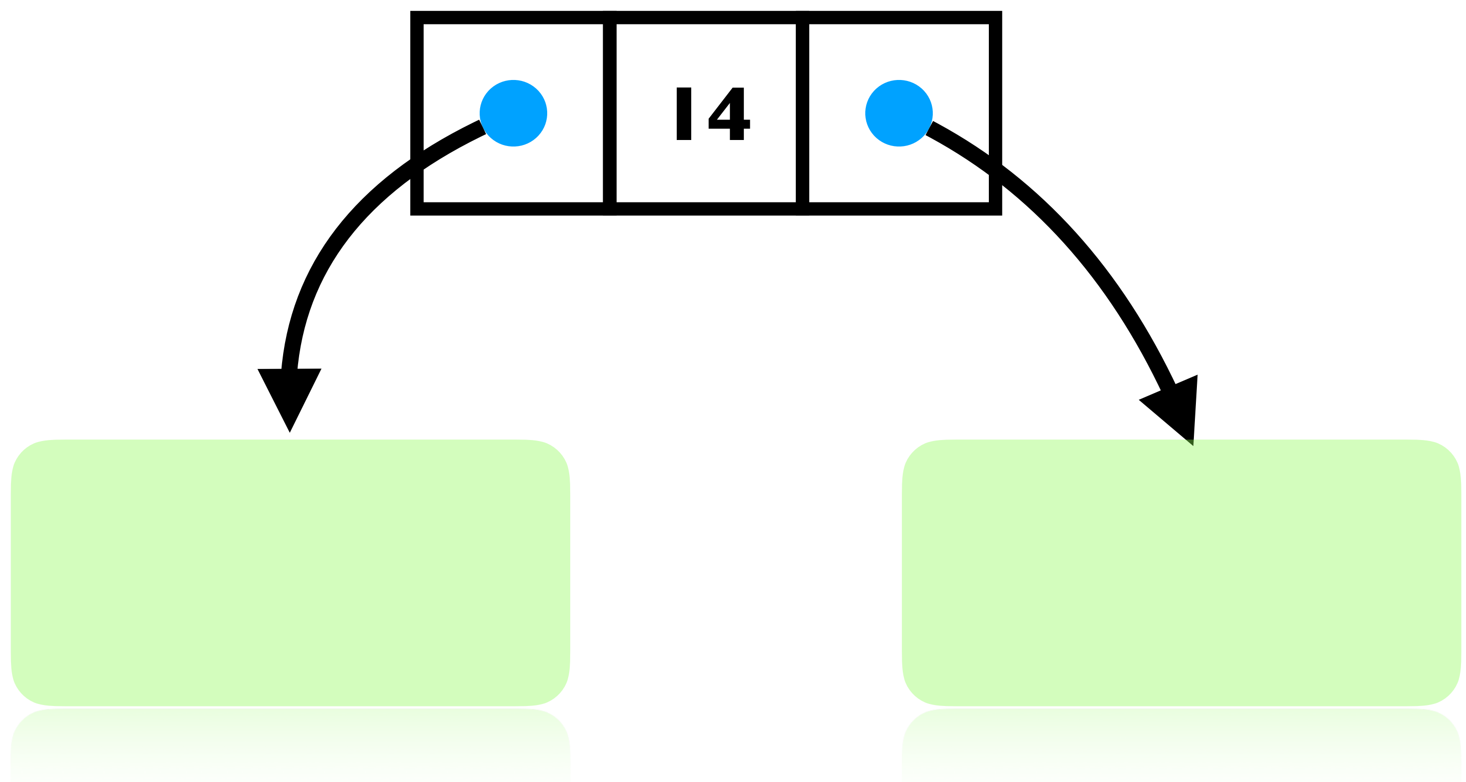
Lesson: trade-offs *nuanced*, no silver bullet

Let’s apply this intuition to **trees**

BST: Binary Tree that has the…

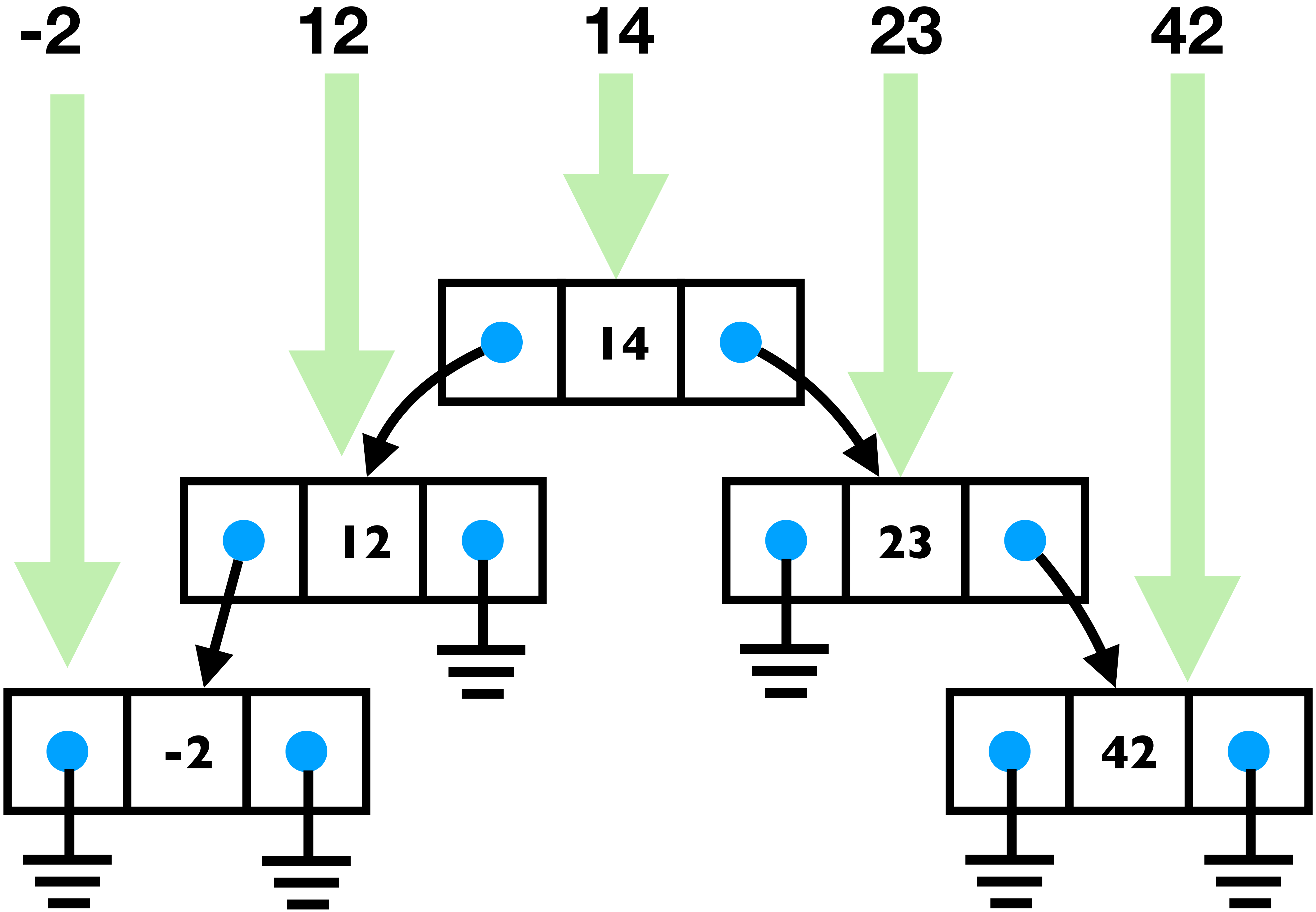
Binary Search Property

Every item in left child < parent, vice versa



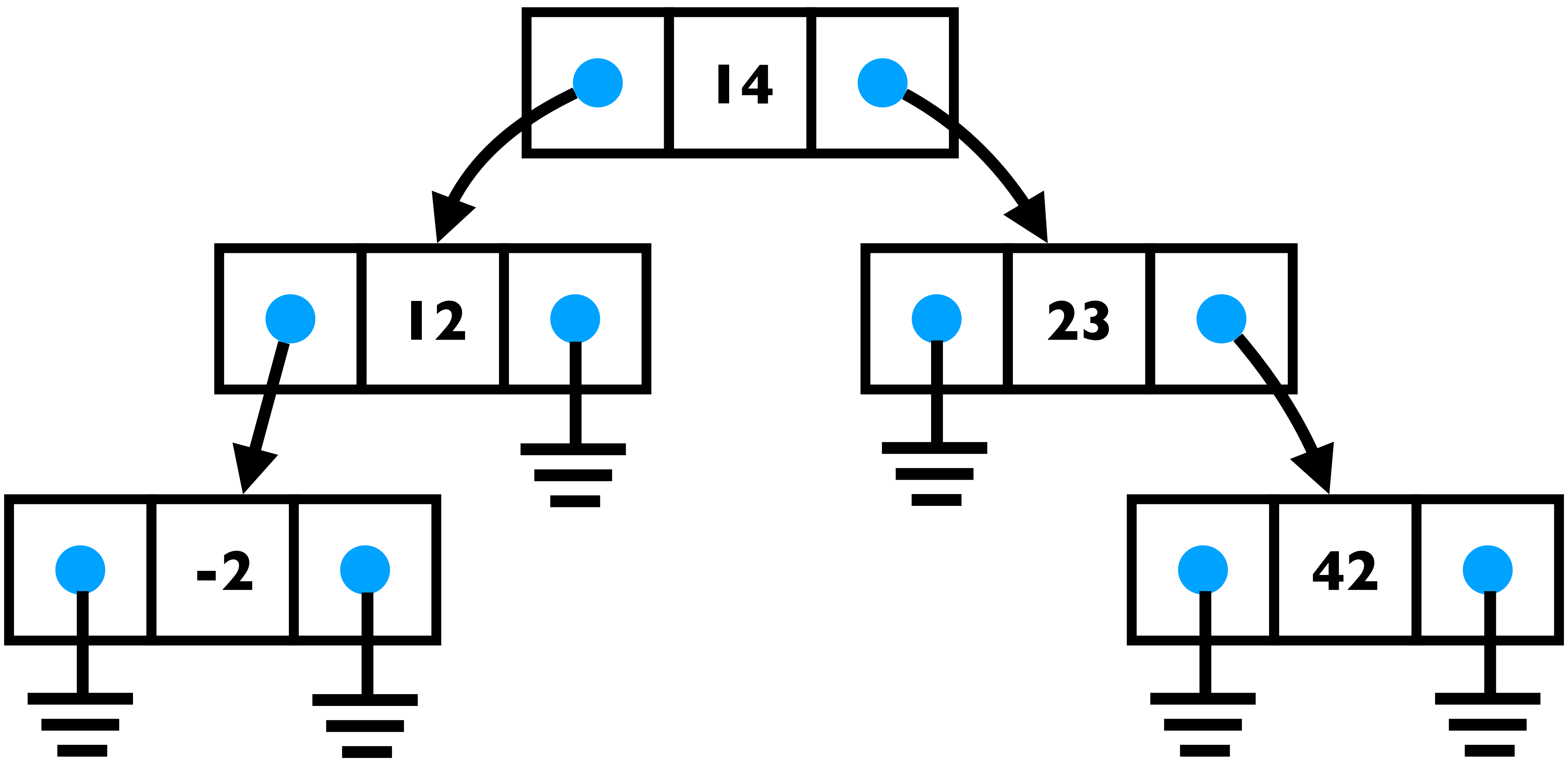
Everything over here had better be < 14

(Even in children of this node)

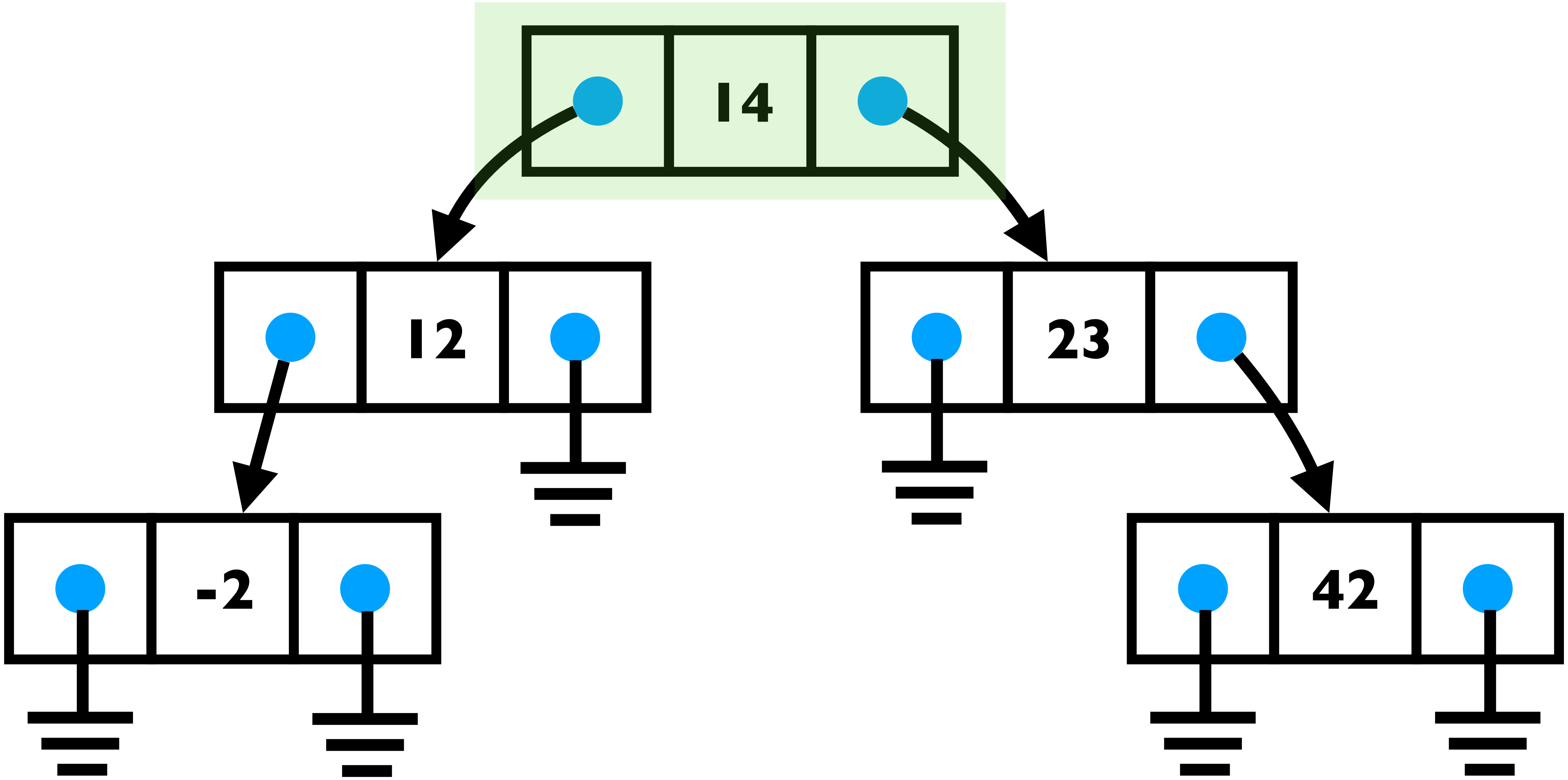


-2
12
14
23
42
14
12
23
-2
42

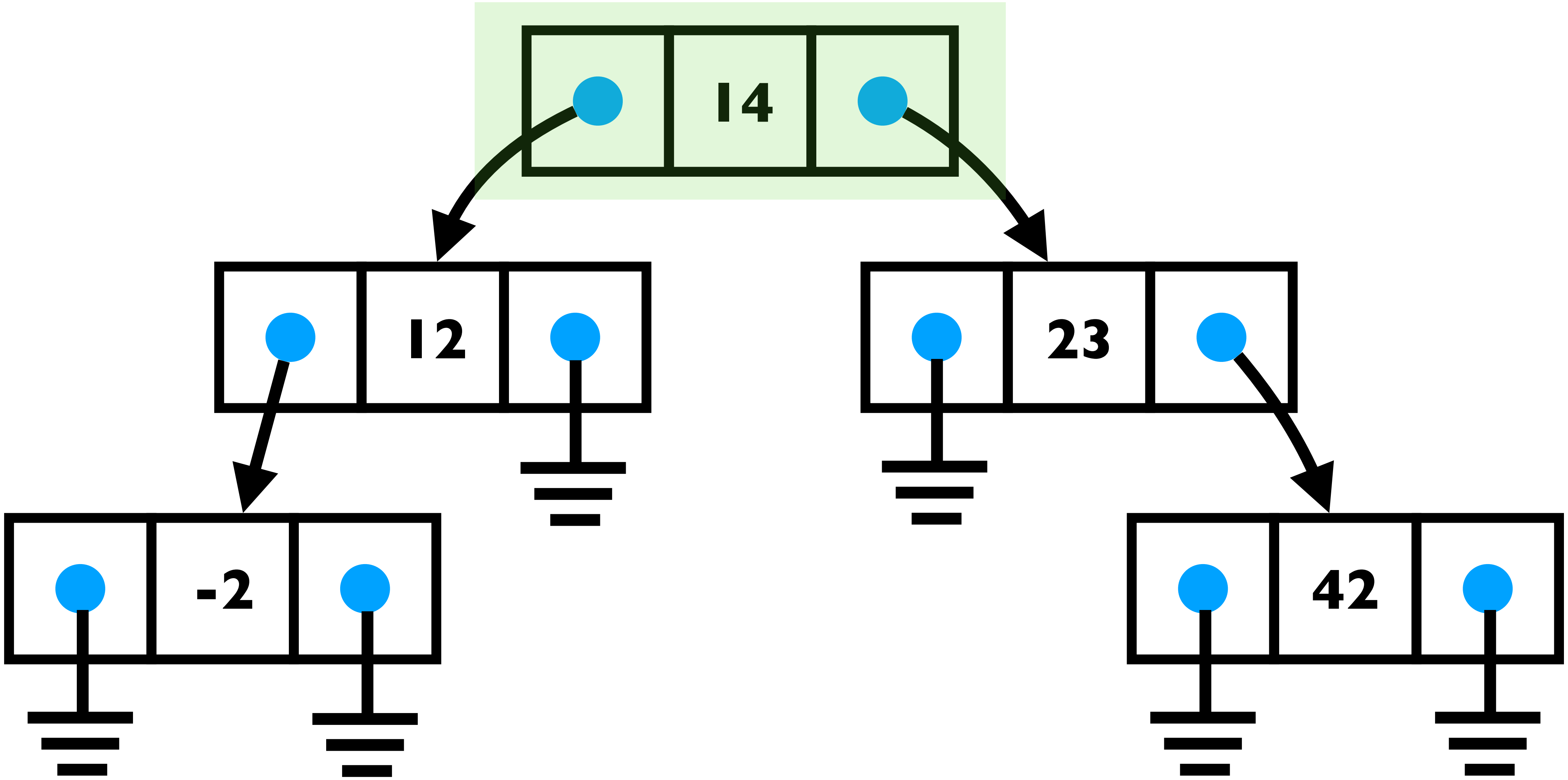
Let’s say we want to look for 12…

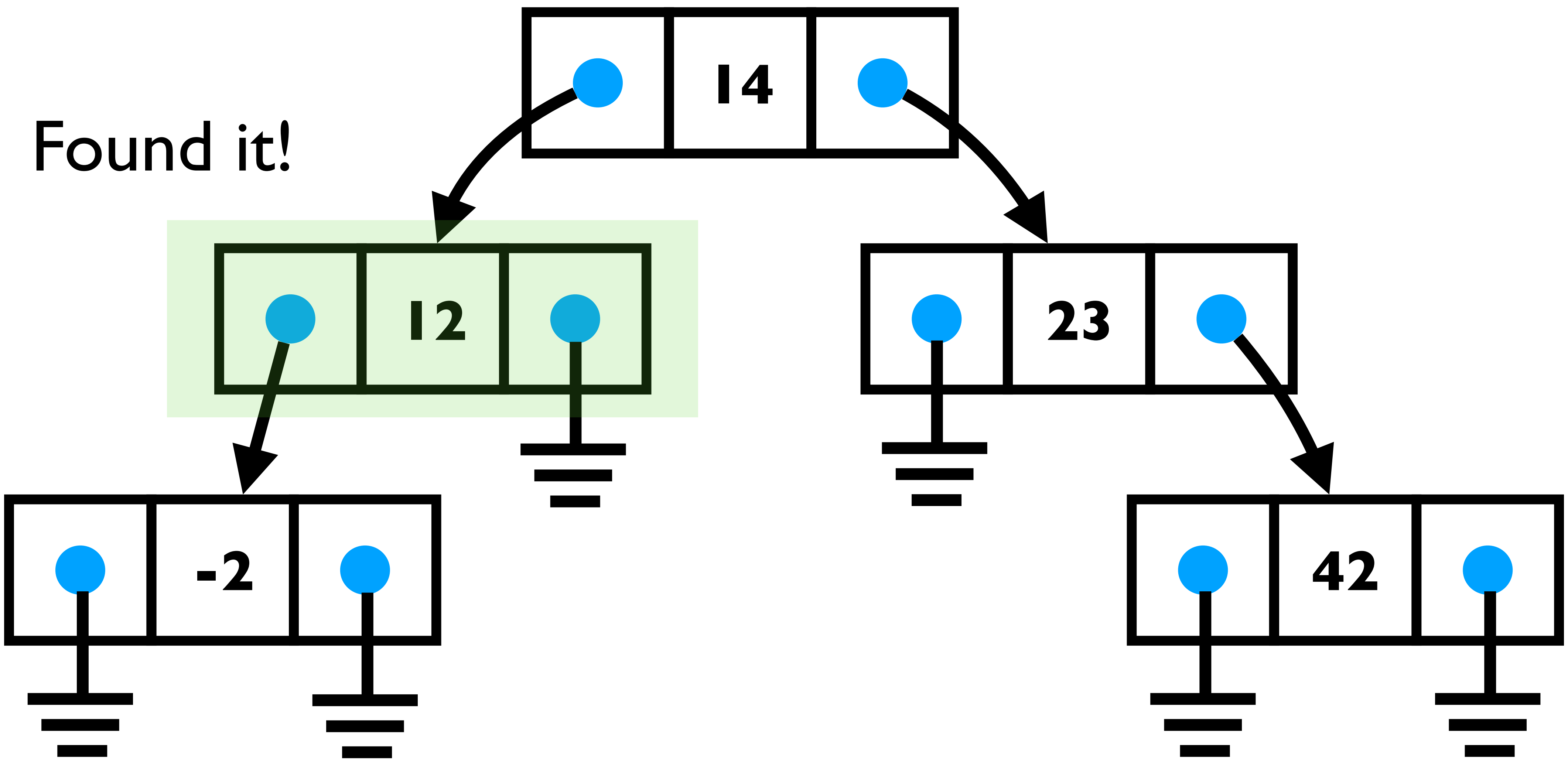


Start at the top of the tree



12 < 14, so look down left child

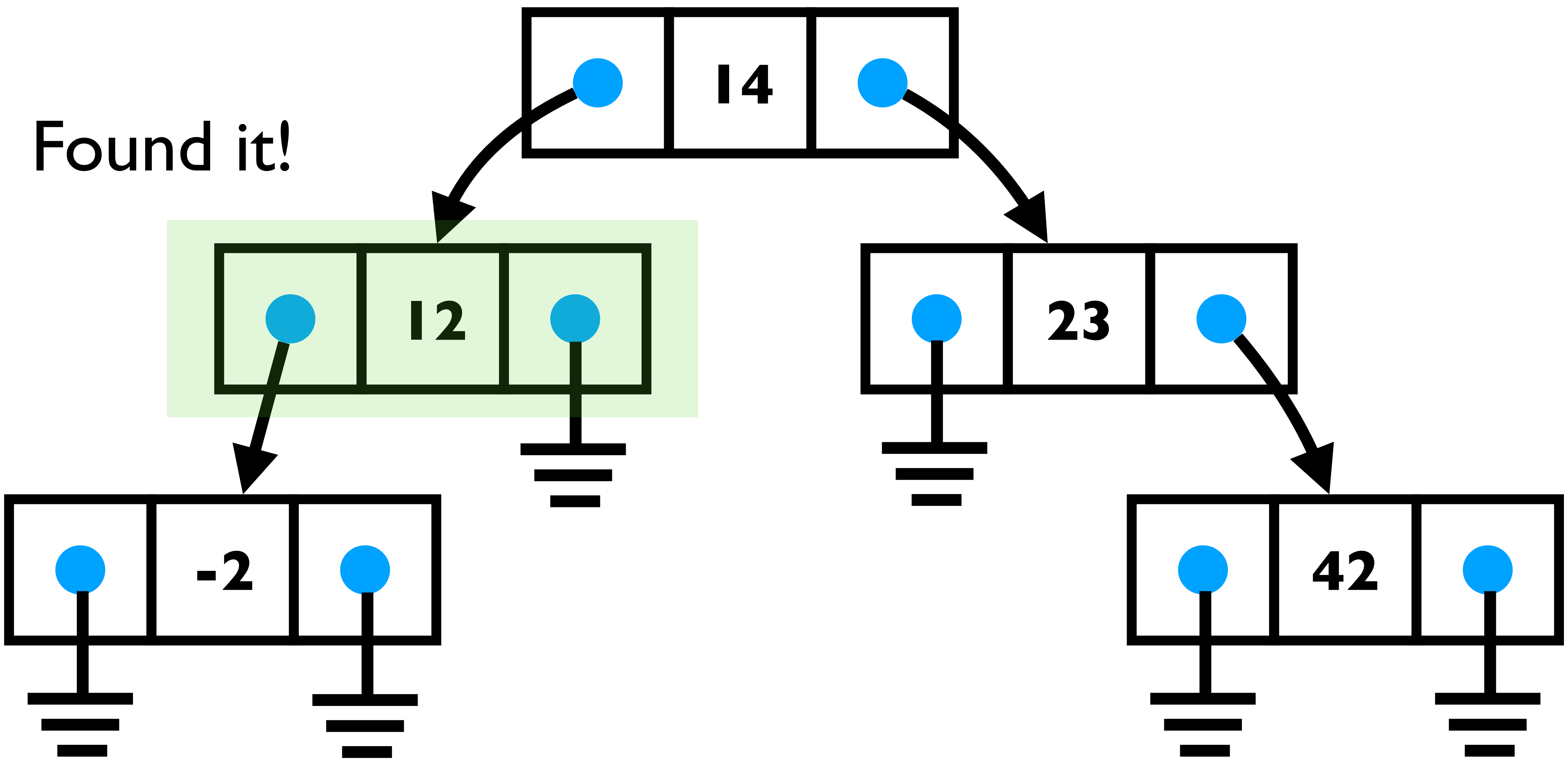




Found it!
14
12
23
-2
42

At a high level: Walk down tree left / right until you find what you're looking for

The **structure** of the tree tells you *which way* to go

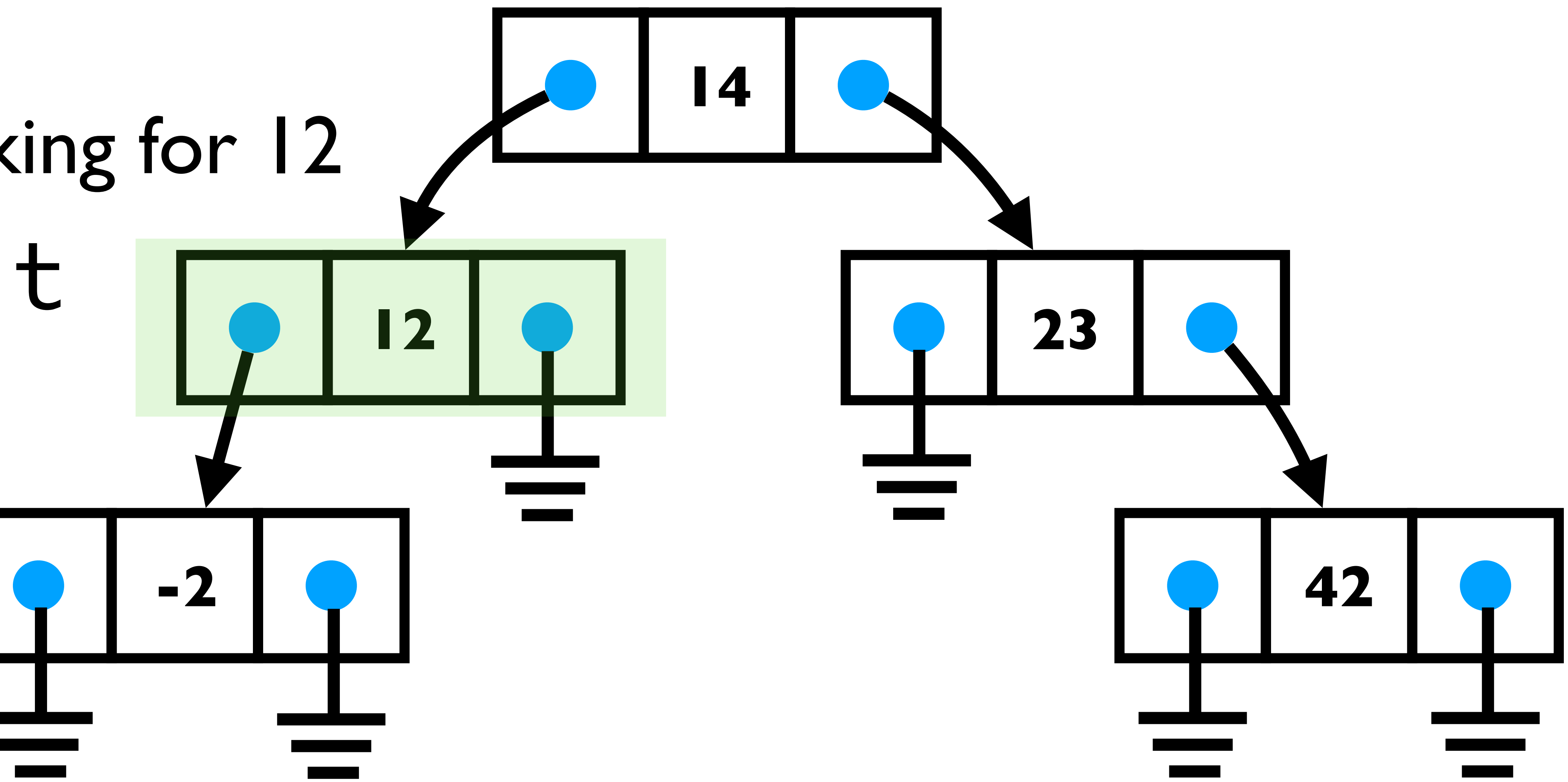


Implementing `lookup`

```
# Assume t is a tree with .left, .right, and .elem
def lookup(t,i):
    if t == null: return false
    if t.elem == i: return true
    else if t.elem < i: return lookup(t.left, i)
    else if t.elem > i: return lookup(t.right, i)
```

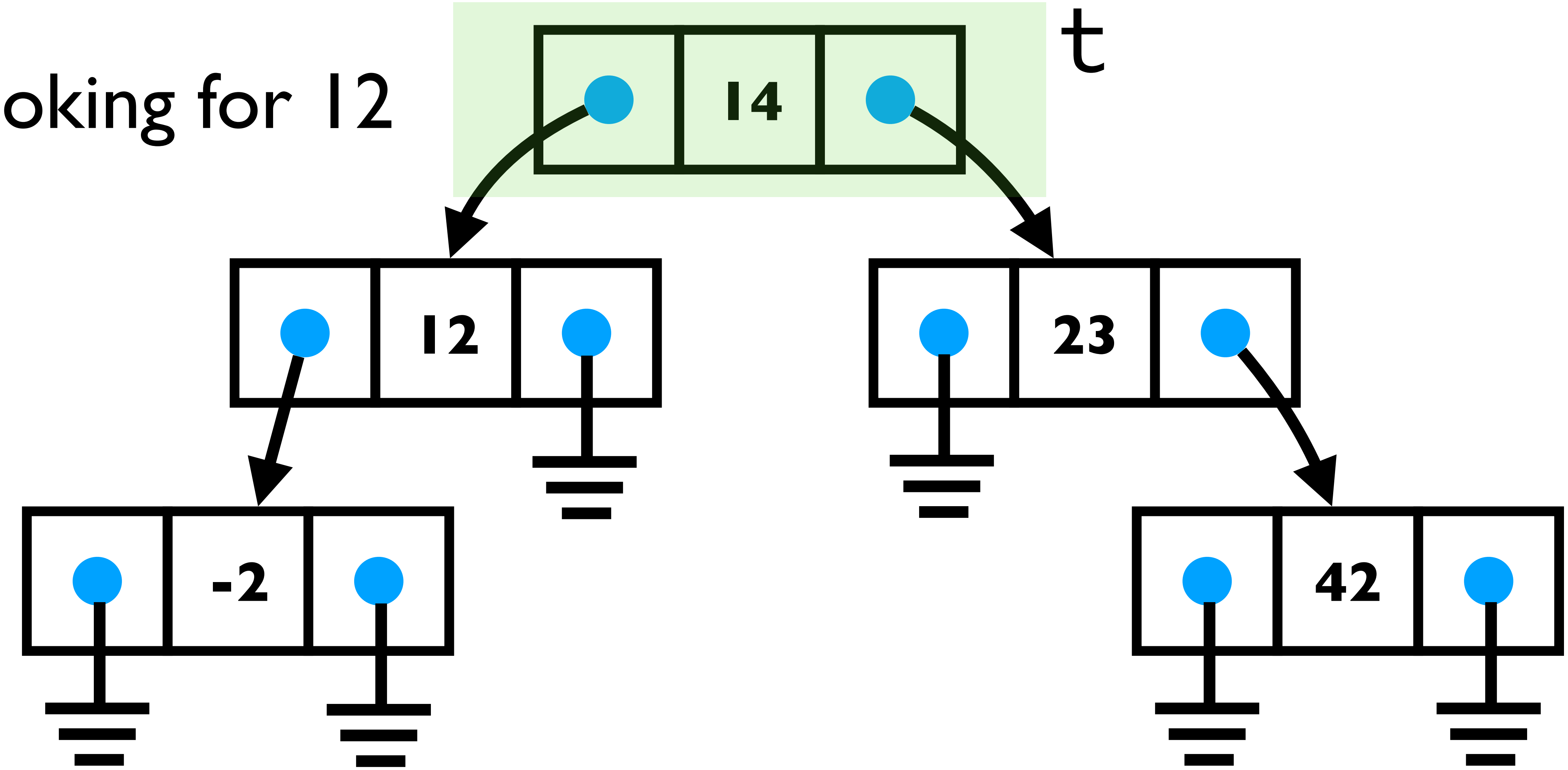
```
# Assume t is a tree with .left, .right, and .elem
def lookup(t,i):
    if t == null: return false
    if t.elem == i: return true
    else if t.elem < i: return lookup(t.left, i)
    else if t.elem > i: return lookup(t.right, i)
```

Looking for 12



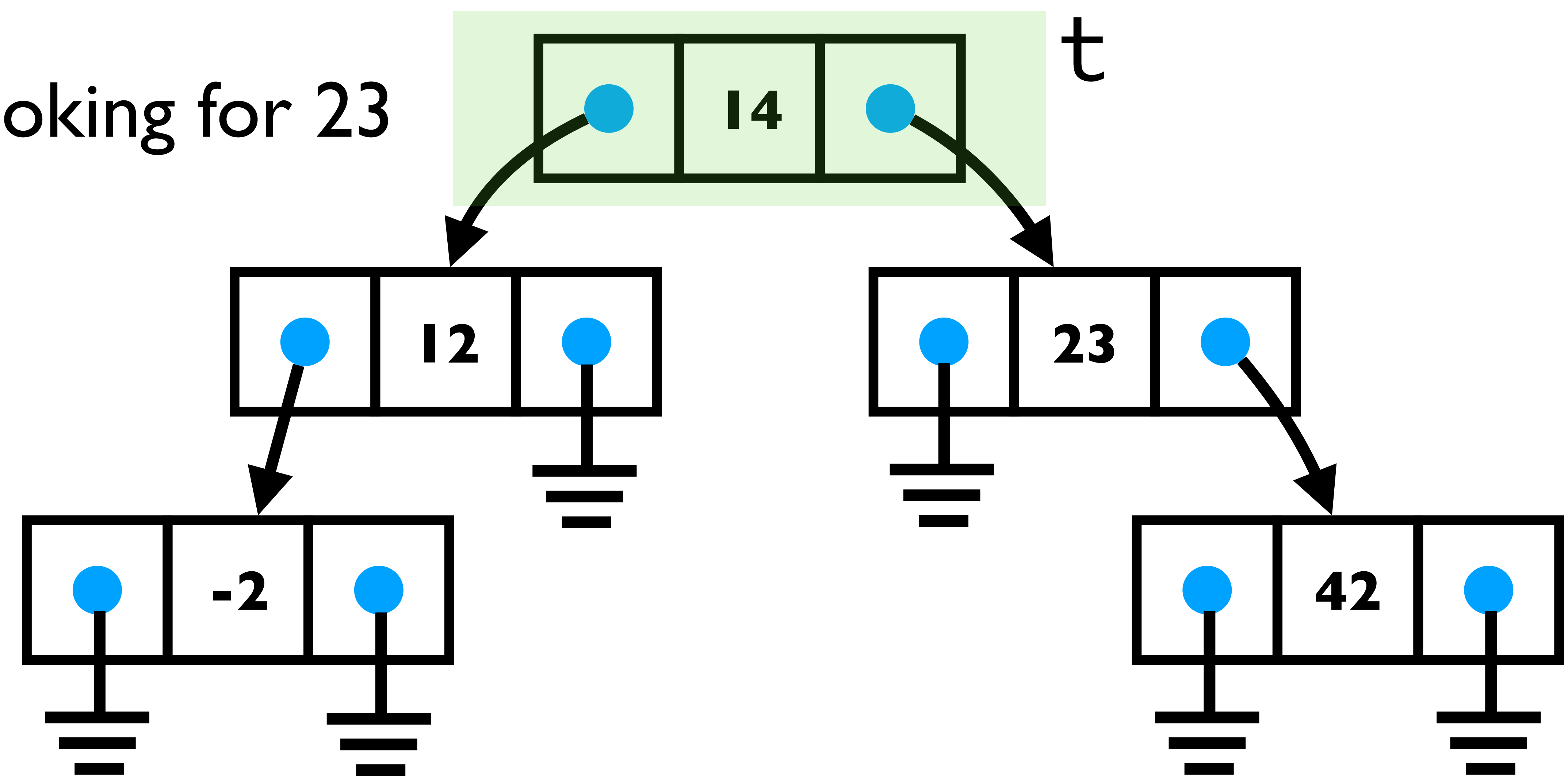
```
# Assume t is a tree with .left, .right, and .elem
def lookup(t,i):
    if t == null: return false
    if t.elem == i: return true
    else if t.elem < i: return lookup(t.left, i)
    else if t.elem > i: return lookup(t.right, i)
```

Looking for 12



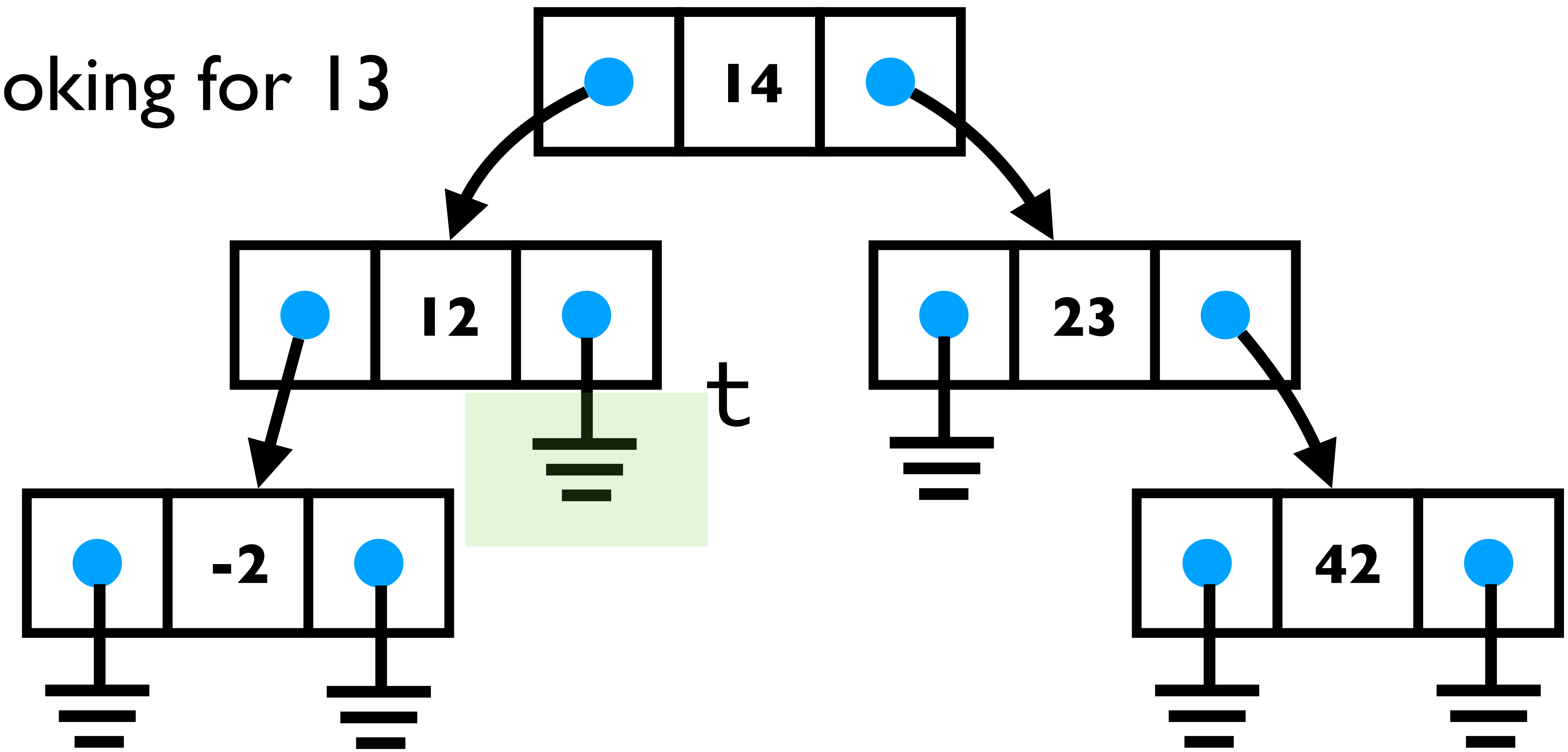
```
# Assume t is a tree with .left, .right, and .elem
def lookup(t,i):
    if t == null: return false
    if t.elem == i: return true
    else if t.elem < i: return lookup(t.left, i)
    else if t.elem > i: return lookup(t.right, i)
```

Looking for 23

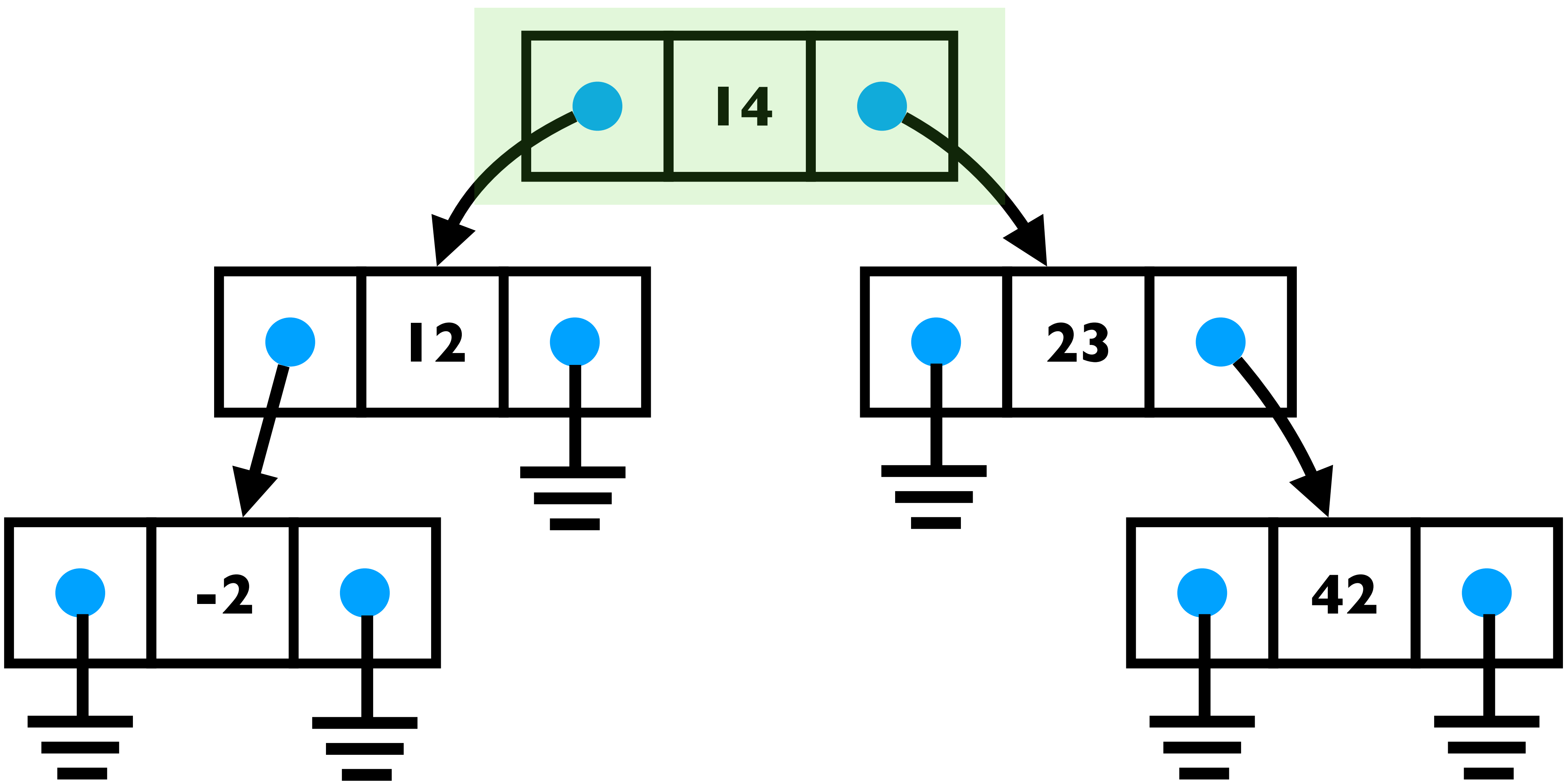


```
# Assume t is a tree with .left, .right, and .elem
def lookup(t,i):
    if t == null: return false
    if t.elem == i: return true
    else if t.elem < i: return lookup(t.left, i)
    else if t.elem > i: return lookup(t.right, i)
```

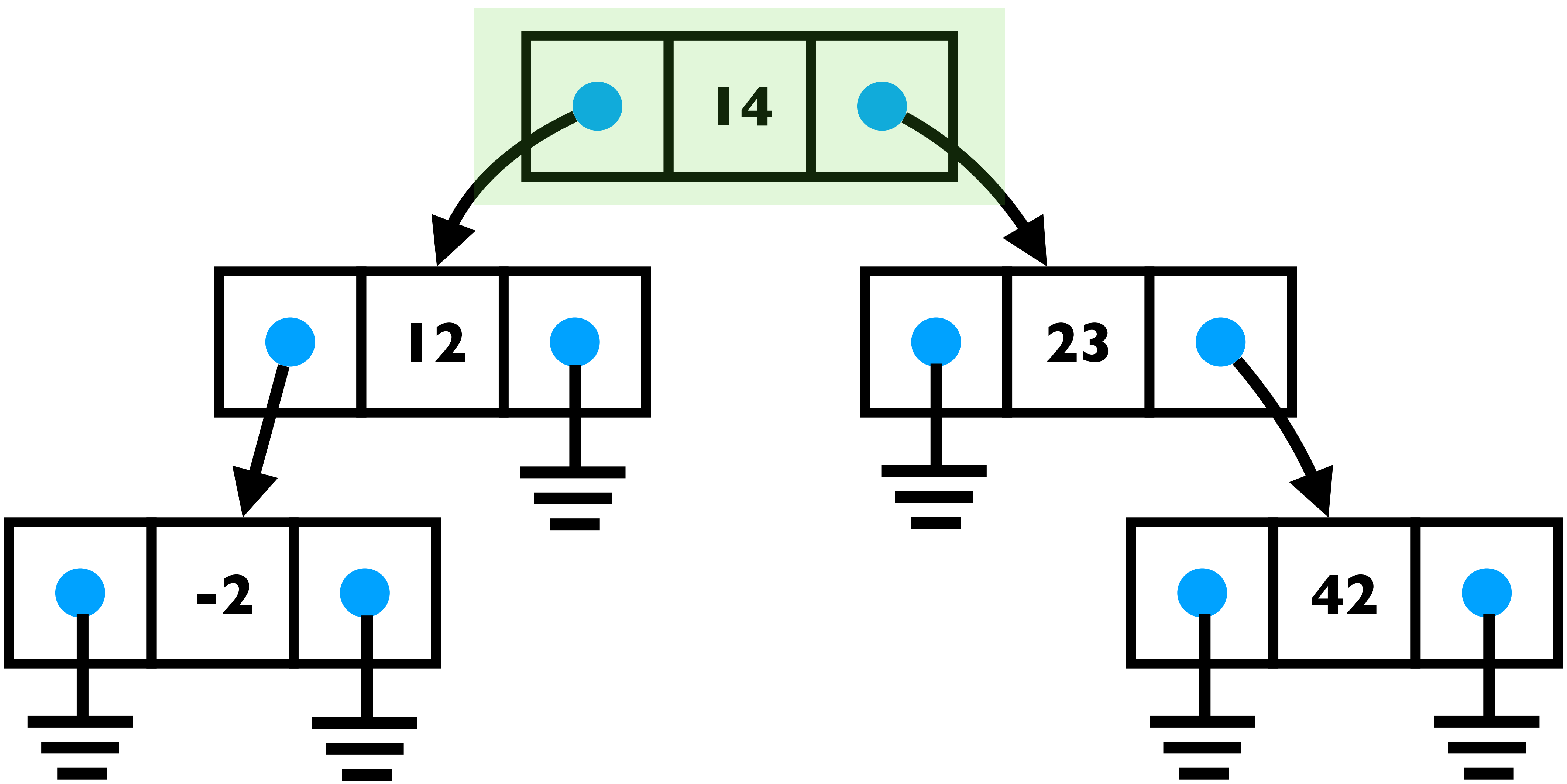
Looking for 13



Now let's say I want to insert 13…

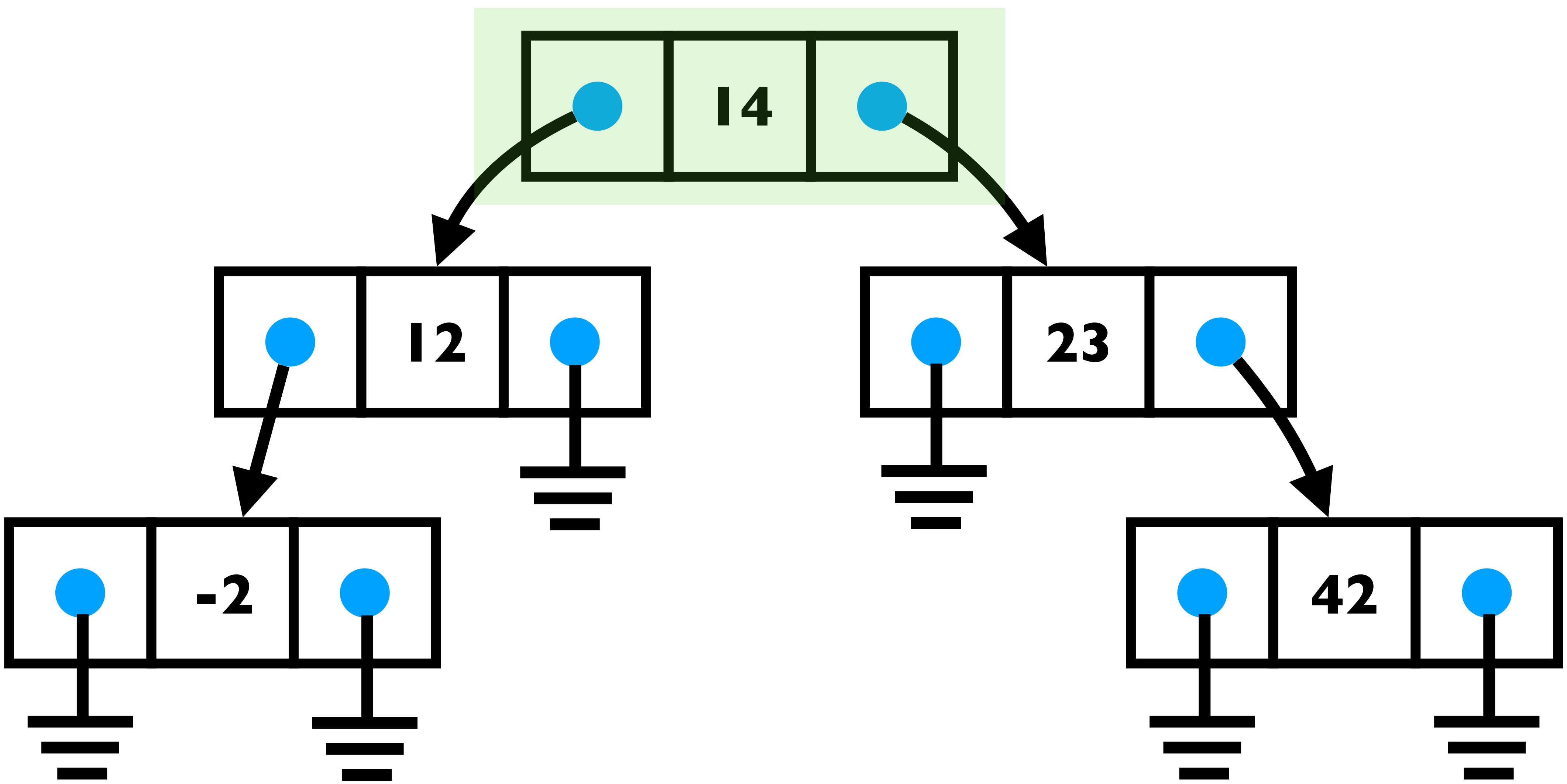


Start at top

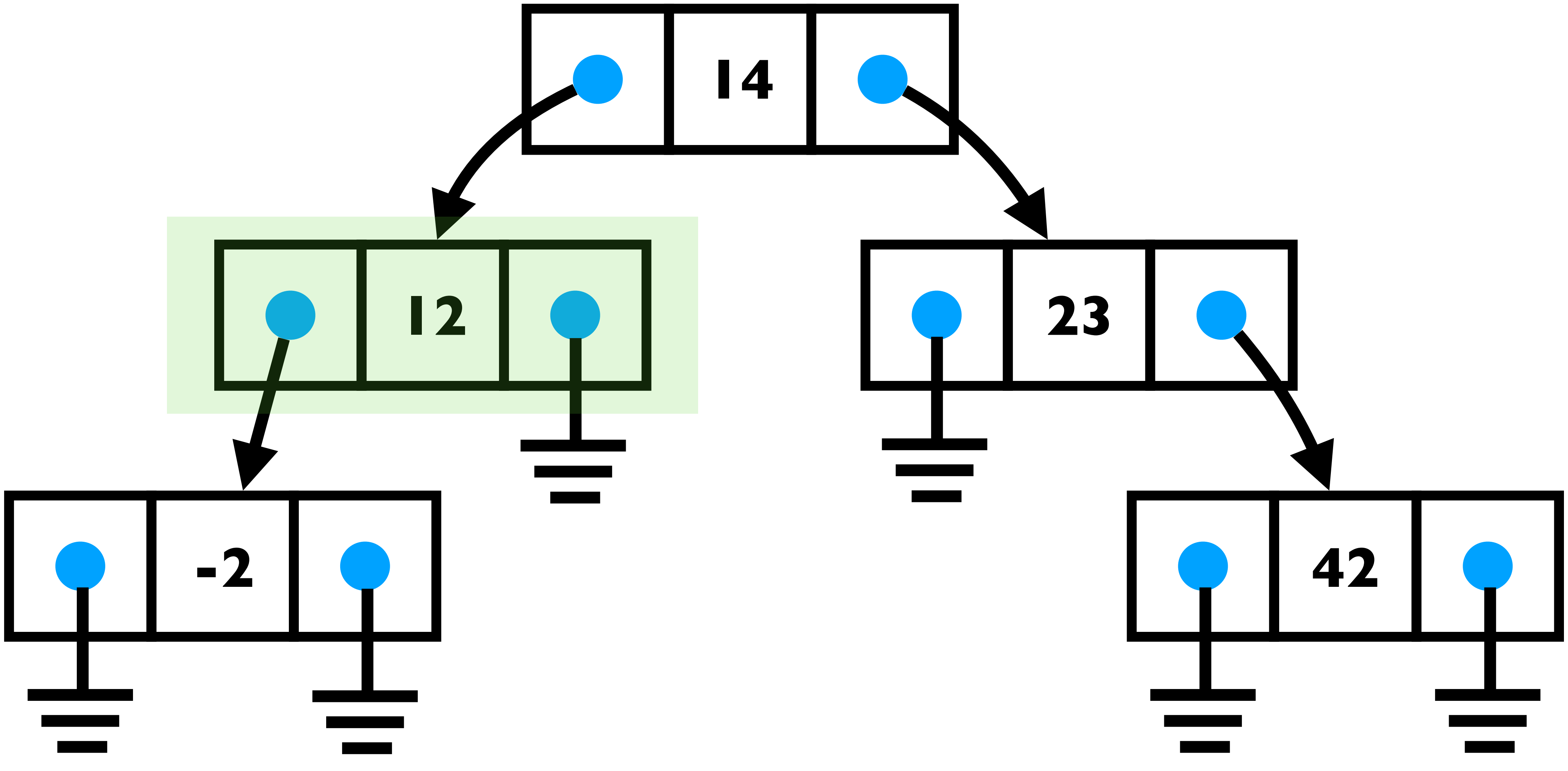


13 < 14, so must add to left

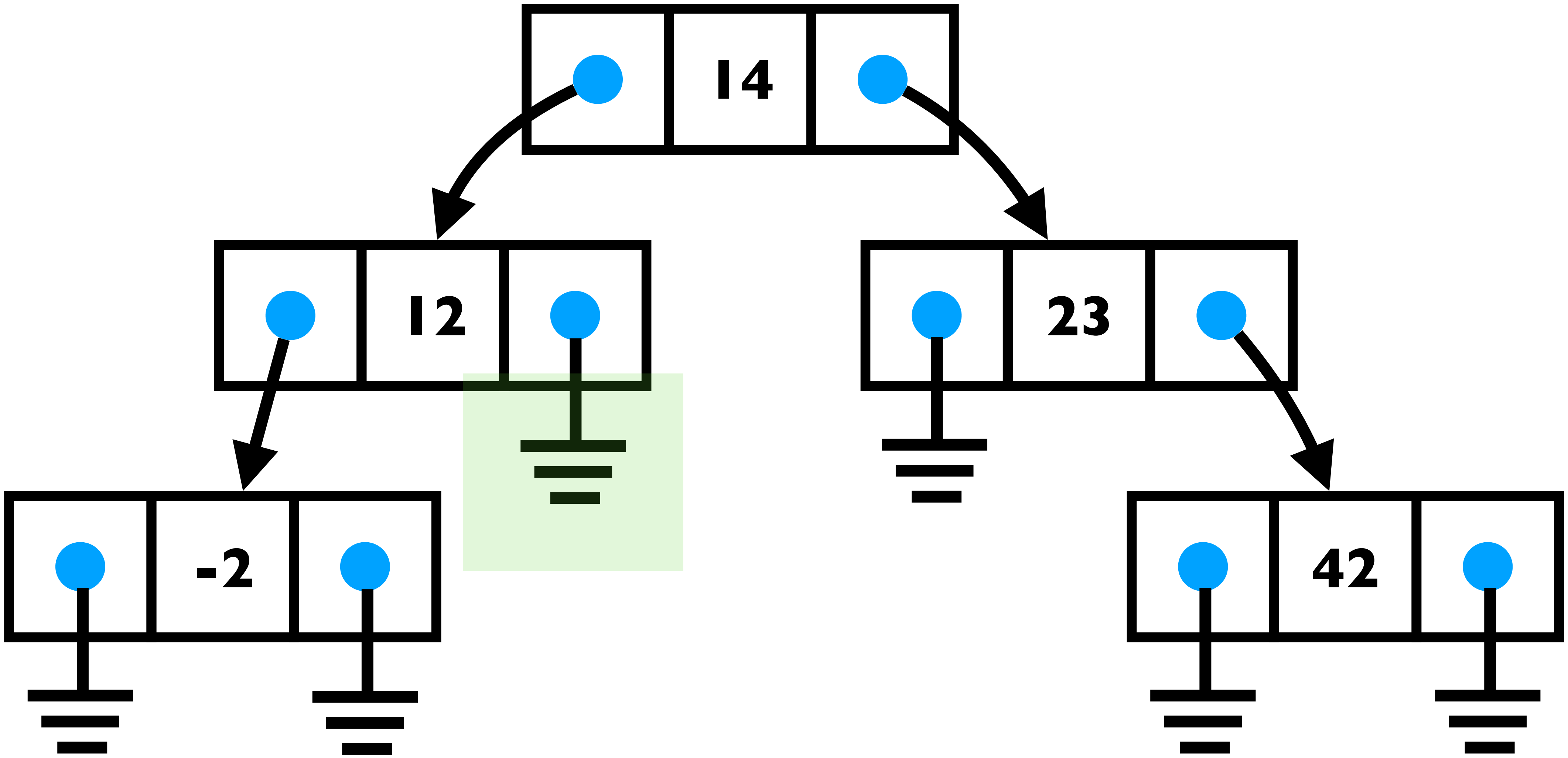
Start at top



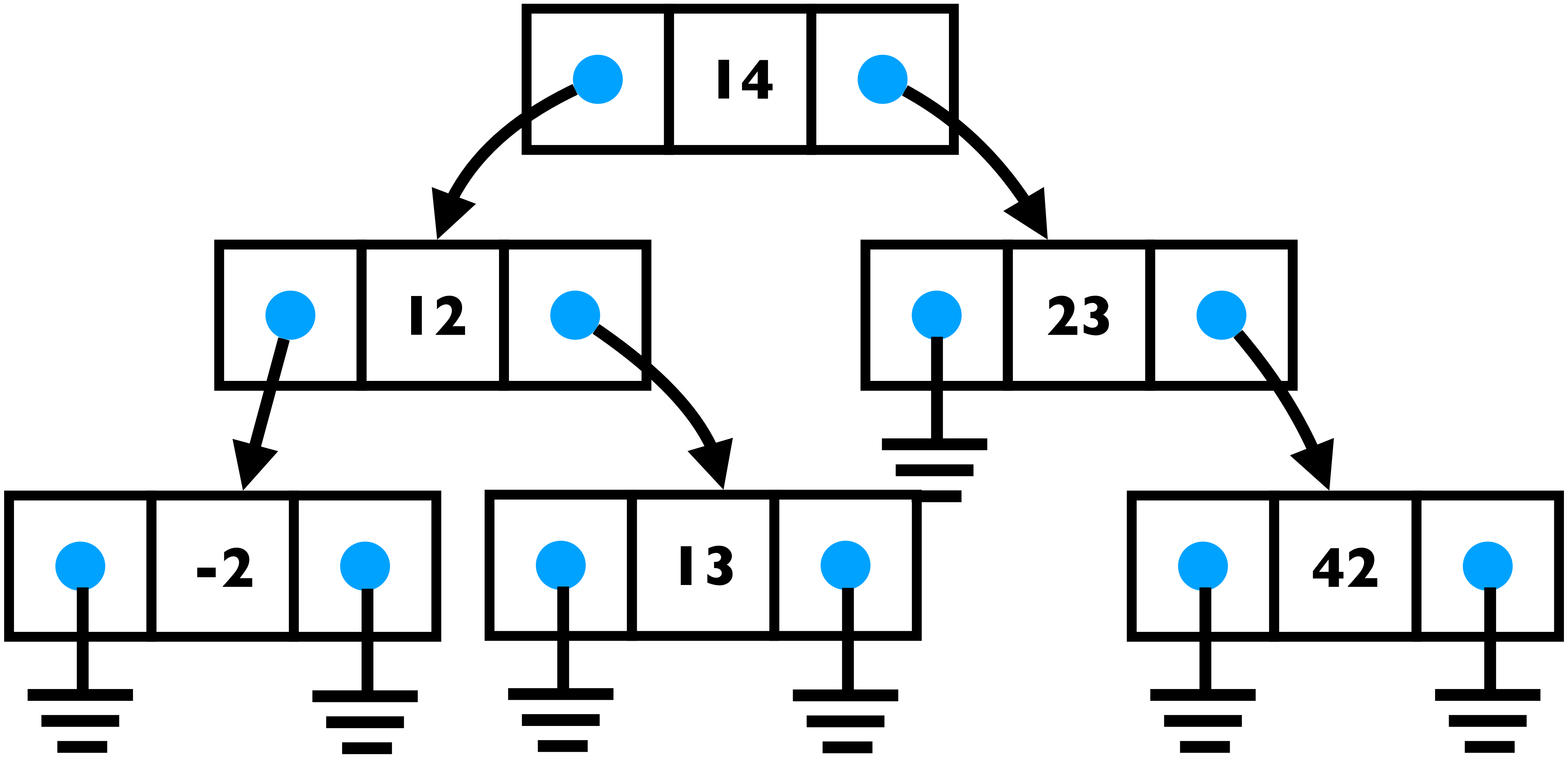
13 > 12, so must add to right



Oops! Nothing here. Add **new** node



Oops! Nothing here. Add **new** node



```
# Assume node(elem,left,right) is a constructor
def add(t,i):
    if t == null: new node(i,null,null)
    if t.elem == i: return t
    else if t.elem < i:
        return node(t.elem,add(t.left,i),t.right)
    else if t.elem > i:
        return node(t.elem,t.right,add(t.right,i))
```

Observation: BSTs can store more than just numbers

Only need **total ordering** (any two can be compared)

- ➡Strings
- ➡Doubles
- ➡Other user defined types
  - ➡Some langs allow overloading <

Observation: BSTs can store more than just numbers

Only need **total ordering** (any two can be compared)

- ➡Strings
- ➡Doubles
- ➡Other user defined types
  - ➡Some langs allow overloading <

Can also use as basis for other data structures (e.g., dictionary: nodes key/value pairs) `insert` O(height)

`lookup` O(height)

O(log(size)) when *balanced*

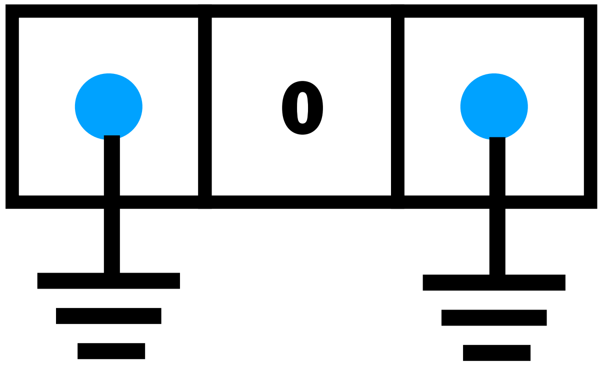
Naive insertion does not balance tree :(

`insert` O(height)

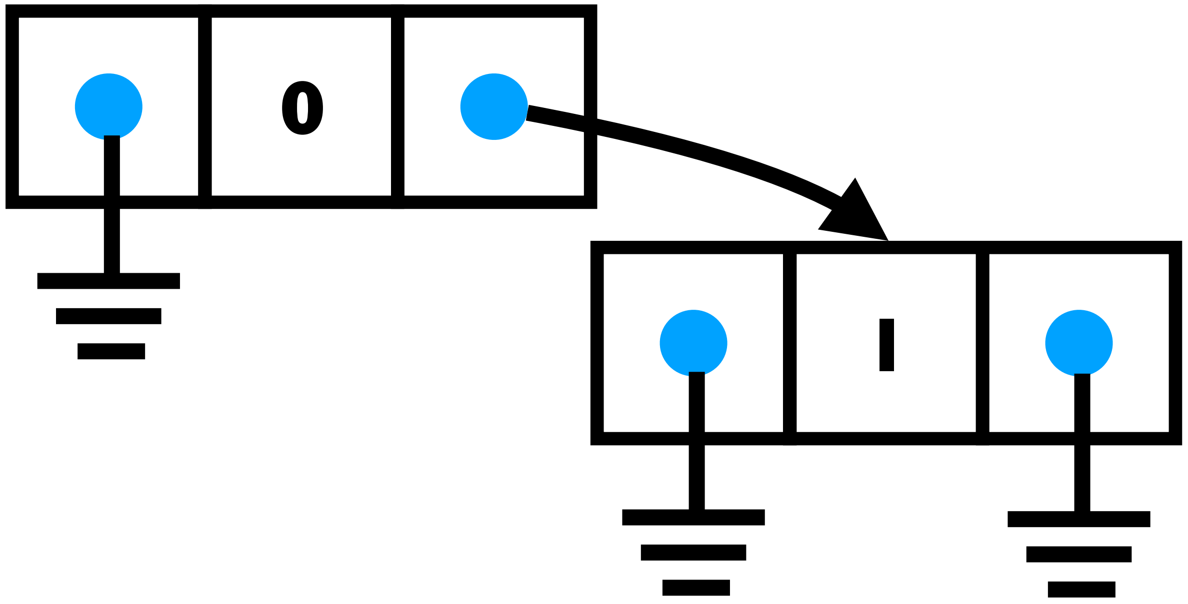
`lookup` O(height)

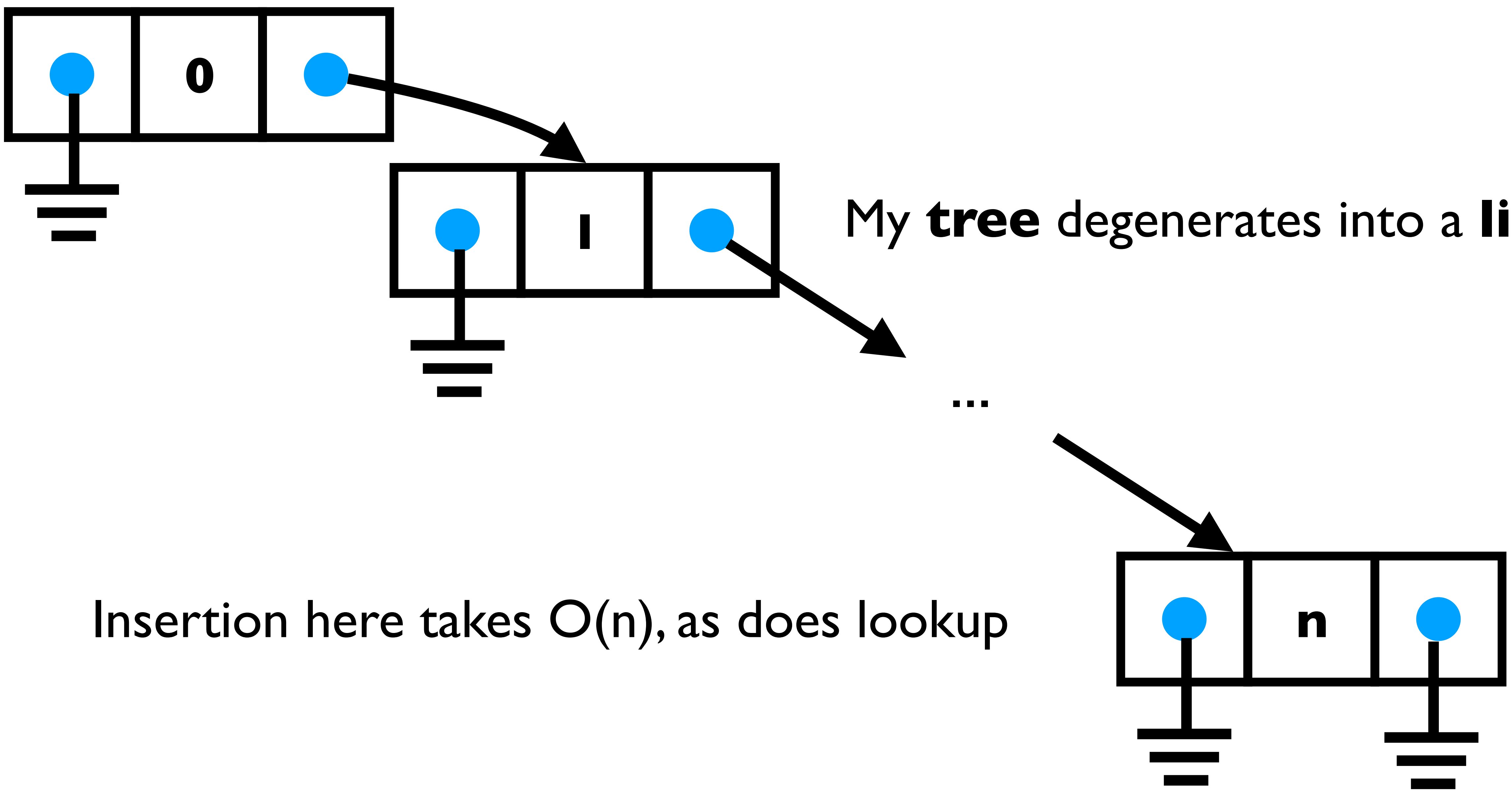
O(log(size)) when *balanced*

Let's say I start with a 1-element tree…



Then extend it…





My **tree** degenerates into a **list**

Insertion here takes O(n), as does lookup

Question

Can we ensure good performance generally?

- Precompute **best** BST (dynamic programming)
- Randomize insertion order
- Build *even smarter* data structures:
  - Red-Black trees maintain "balanced-ish" trees
  - AVL trees "rebalance" the tree

(Beyond scope, today)

List

`insert`	O(1)
`lookup`	O(n)

👍 Simple

👍 Insertions frequent

👎 Lookups frequent

List

`insert`	O(1)	👍	Simple
`lookup`	O(n)	👍	Insertions frequent
		👎	Lookups frequent

Sorted Array

Also allocates lots of memory

`insert`	O(n)	👍	Lookups frequent
`lookup`	O(log(n))	👎👎	Insertions frequent

List

		👍	Simple
`insert`	O(1)	👍	Insertions frequent
`lookup`	O(n)	👎	Lookups frequent

Sorted Array

Also allocates lots of memory

`insert`	O(n)	👍	Lookups frequent
`lookup`	O(log(n))	👎👎	Insertions frequent

BST

balanced

		👍👍	Lookups frequent
`insert`	~O(log(n))	👎	Insertions frequent
`lookup`	~O(log(n))	👎	Maintaining balance hard