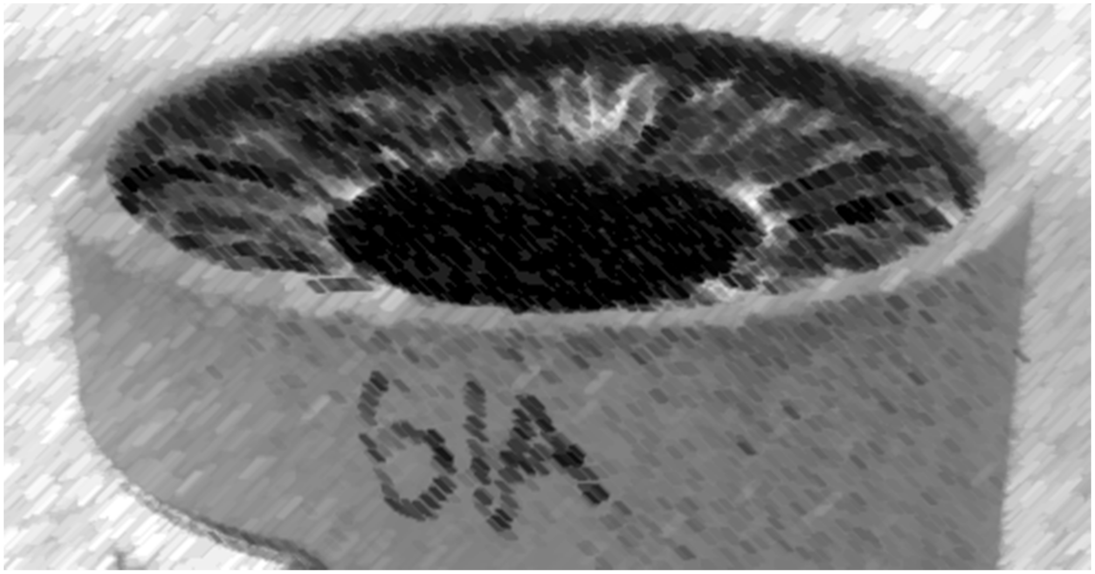


CS61A Lecture 26

Amir Kamil and Hamilton Nguyen

UC Berkeley

March 22, 2013

Announcements

- HW9 out tonight, due 4/3

- Ants extra credit due 4/3
  - See Piazza for submission instructions

Data Structure Applications

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

More about data structures in 61B

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

More about data structures in 61B

Example: recursive lists (also called *linked lists*)

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

More about data structures in 61B

Example: recursive lists (also called *linked lists*)

- Operating systems

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

More about data structures in 61B

Example: recursive lists (also called *linked lists*)

- Operating systems
- Interpreters and compilers

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

More about data structures in 61B

Example: recursive lists (also called *linked lists*)

- Operating systems
- Interpreters and compilers
- Anything that uses a queue

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

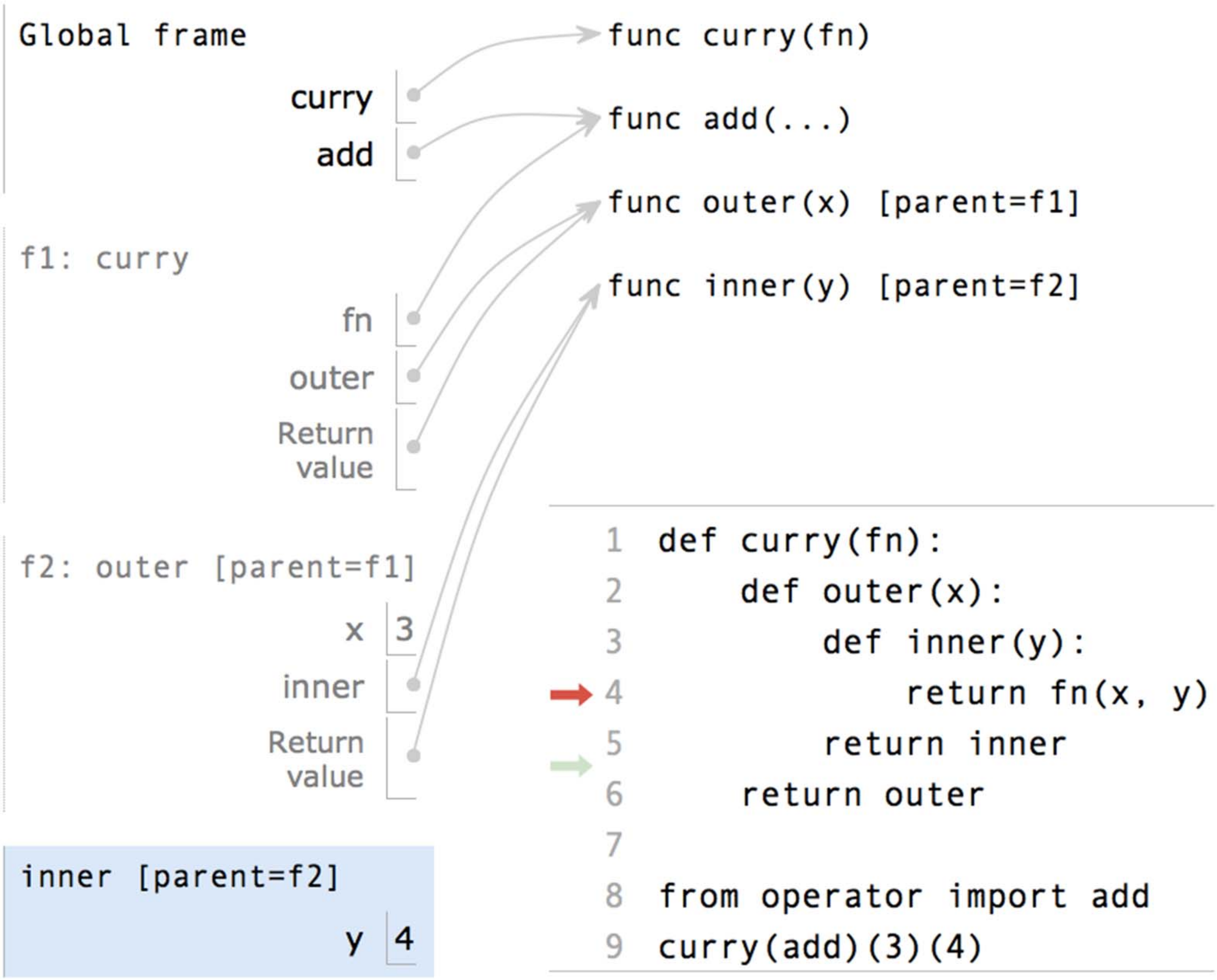
More about data structures in 61B

Example: recursive lists (also called *linked lists*)

- Operating systems
- Interpreters and compilers
- Anything that uses a queue

The Scheme programming language, which we will learn soon, uses recursive lists as its primary data structure

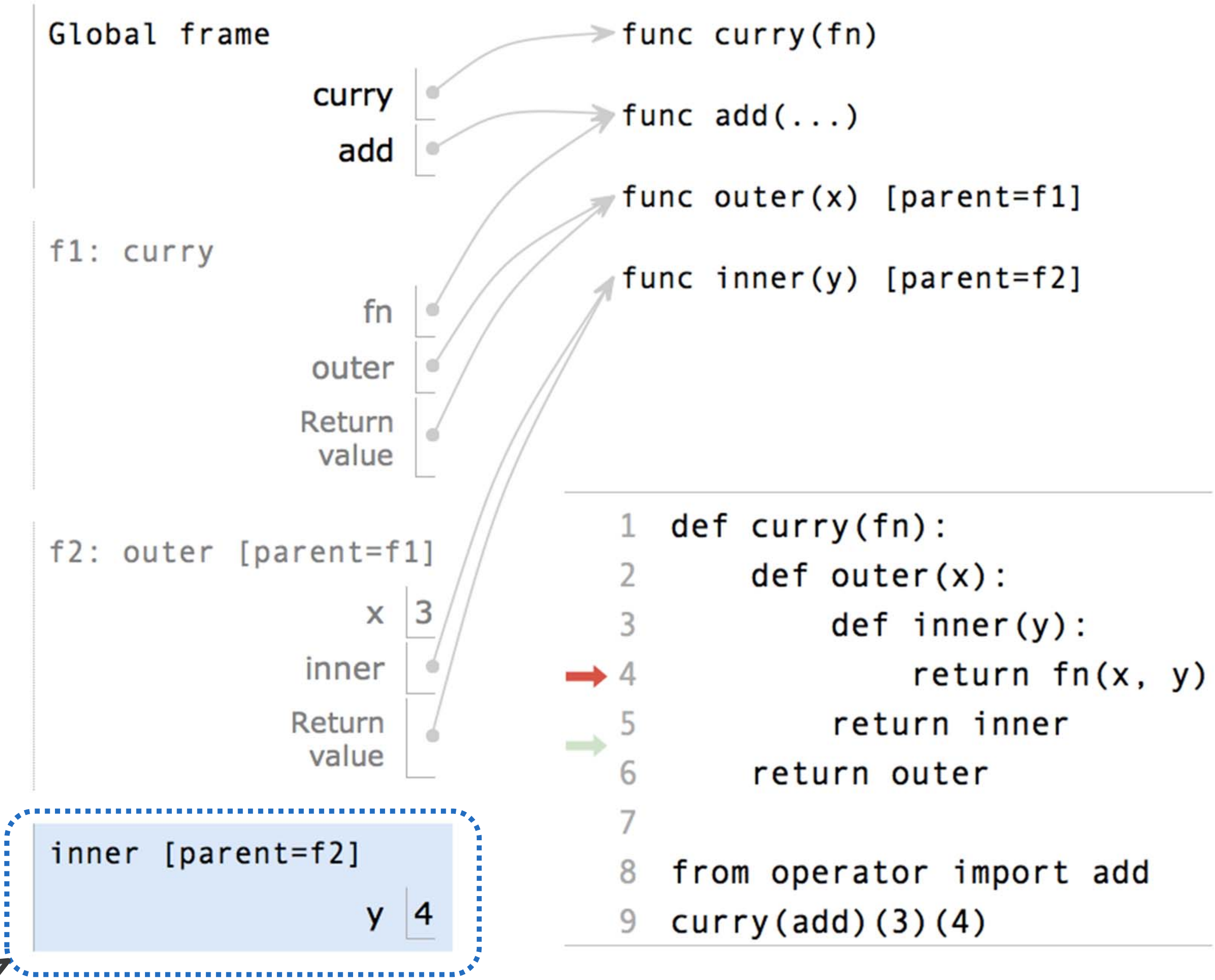
Example: Environments



```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Example: http://goo.gl/8DNY1

Example: Environments

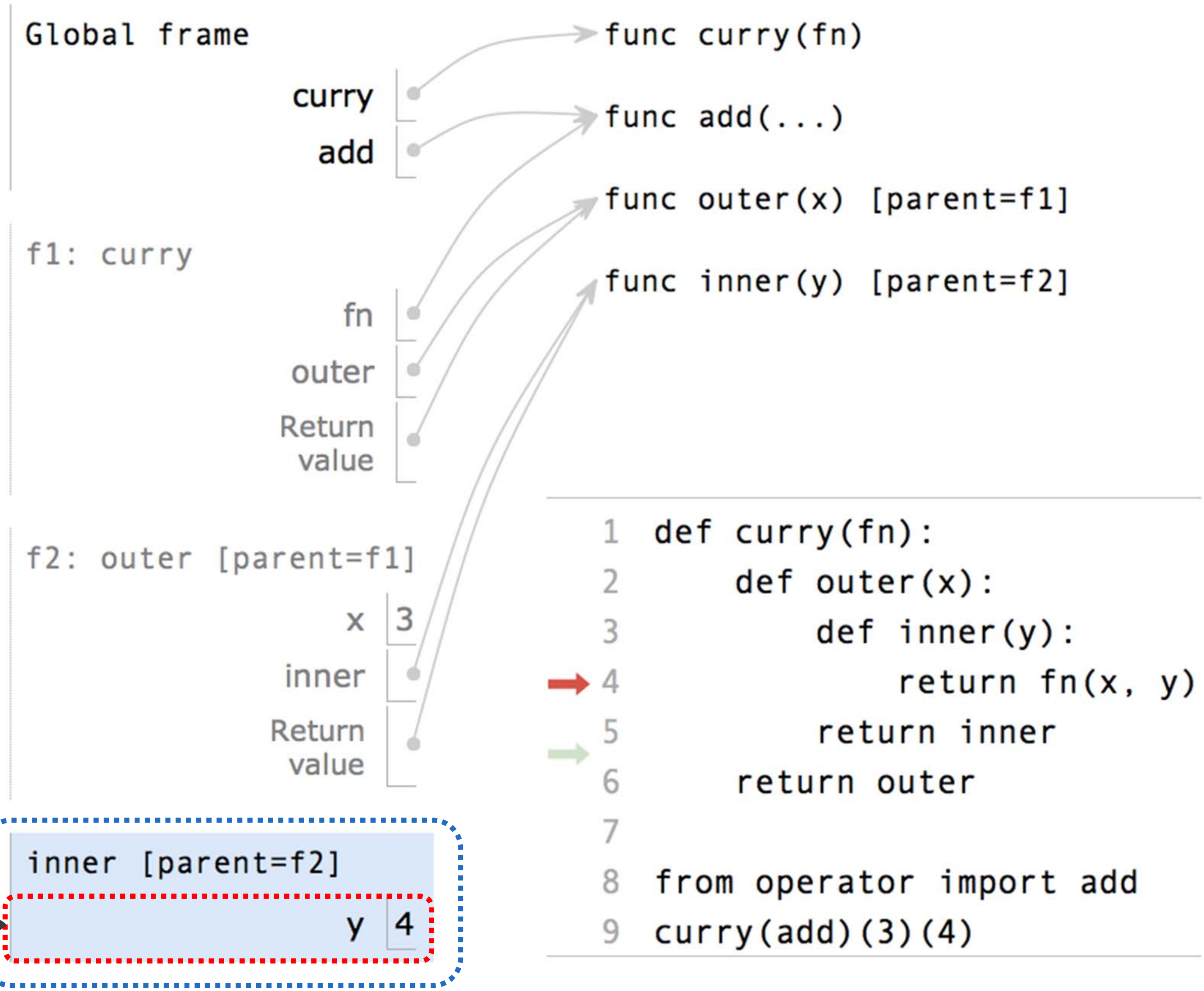


```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Recursive List

Example: http://goo.gl/8DNY1

Example: Environments



Global frame
curry
add

f1: curry
fn
outer
Return value

f2: outer [parent=f1]
x 3
inner
Return value

inner [parent=f2]
y 4

func curry(fn)
func add(...)
func outer(x) [parent=f1]
func inner(y) [parent=f2]

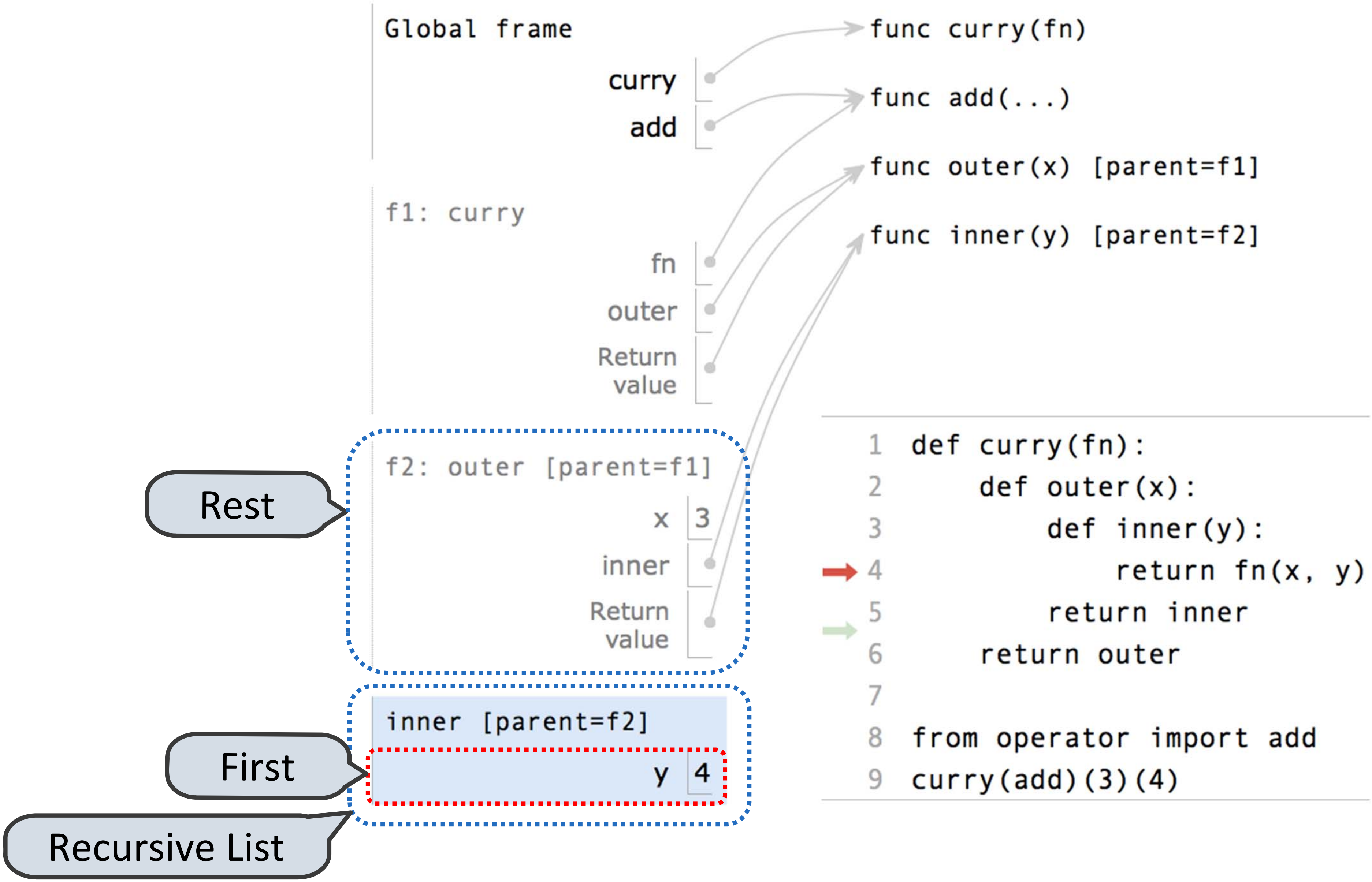
```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

First

Recursive List

Example: http://goo.gl/8DNY1

Example: Environments



```python
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

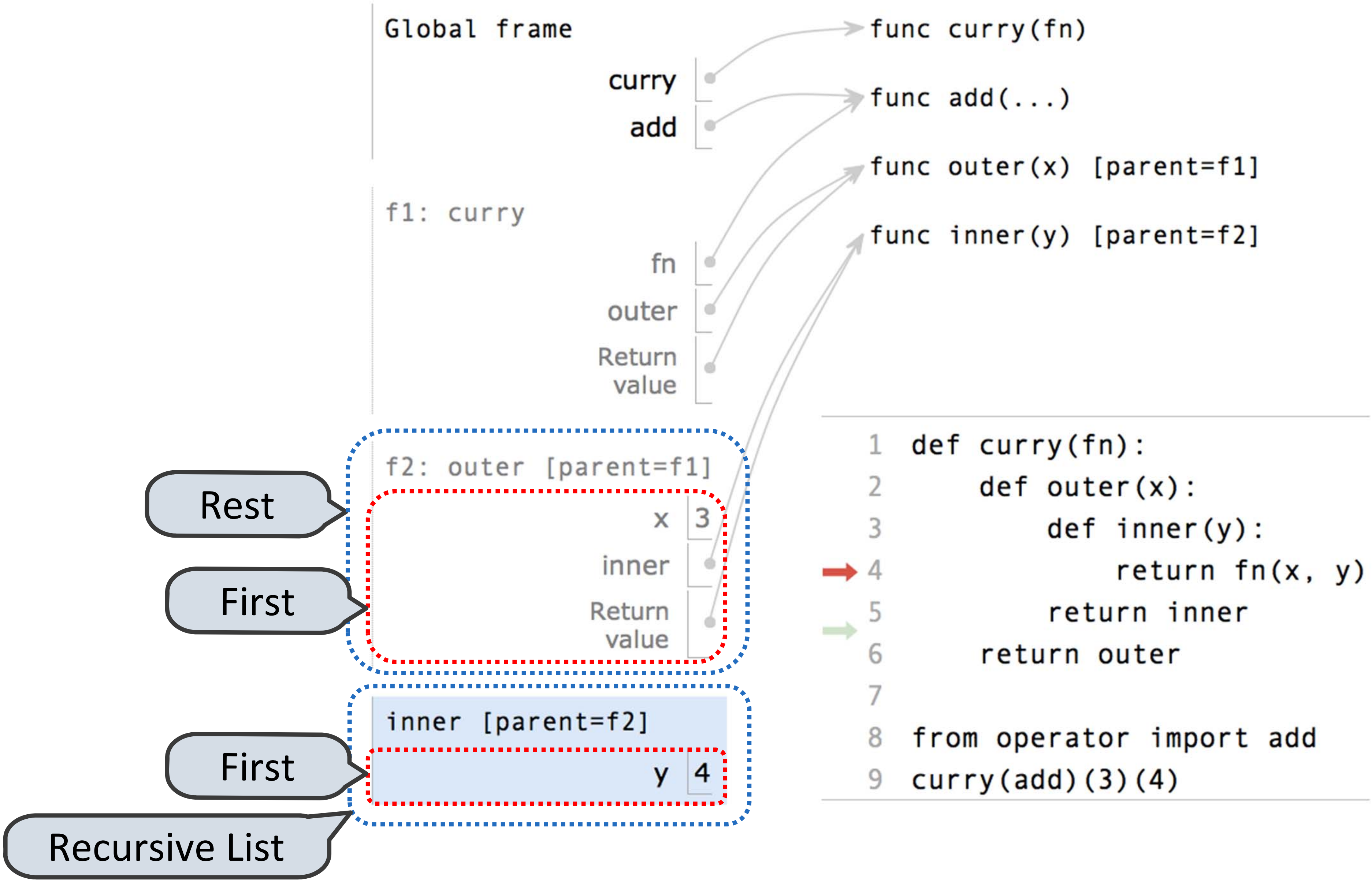
Rest

First

Recursive List

Example: http://goo.gl/8DNY1

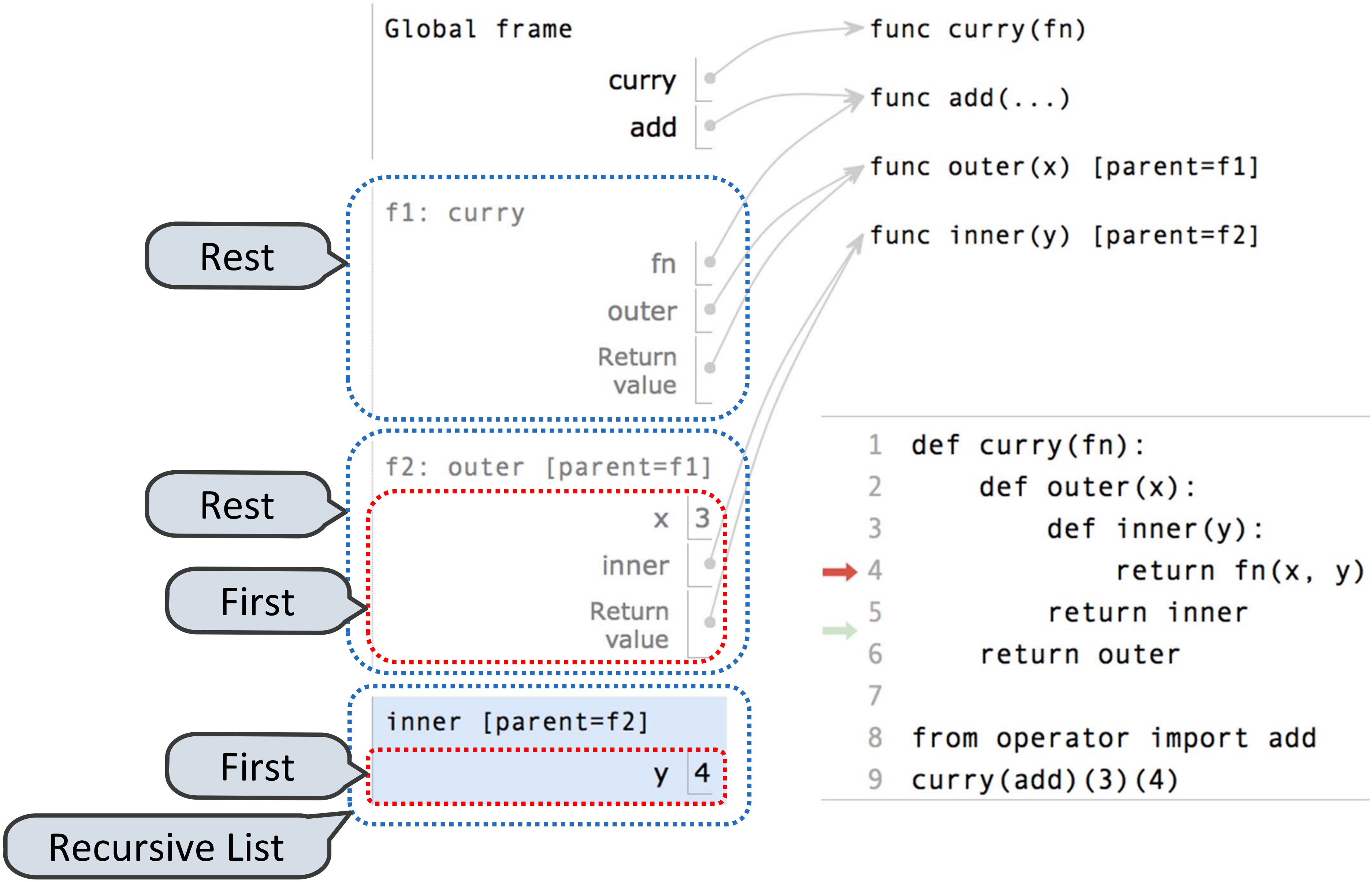
Example: Environments



```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Example: http://goo.gl/8DNY1

Example: Environments



```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Rest

Rest

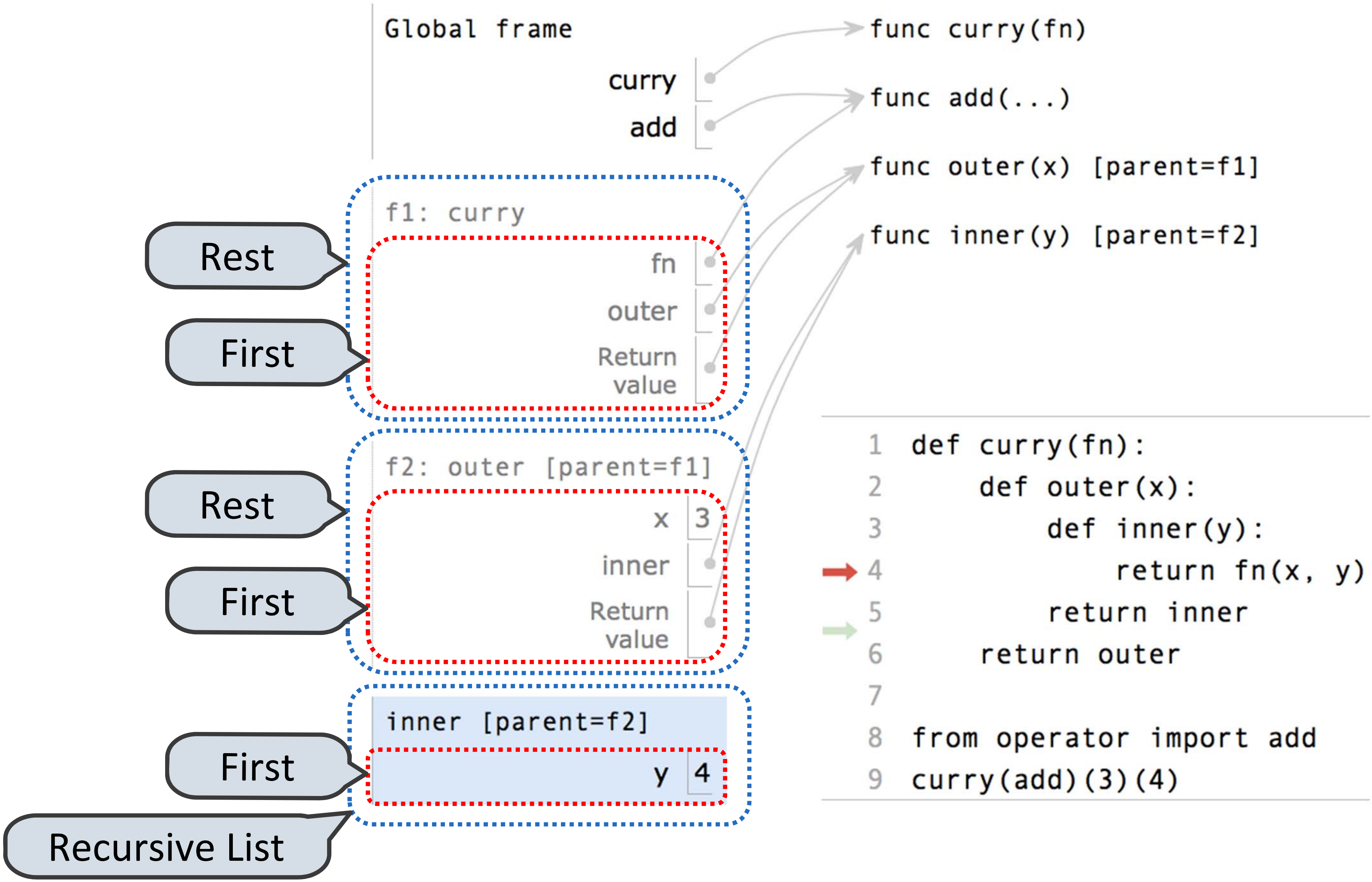
First

First

Recursive List

Example: http://goo.gl/8DNY1

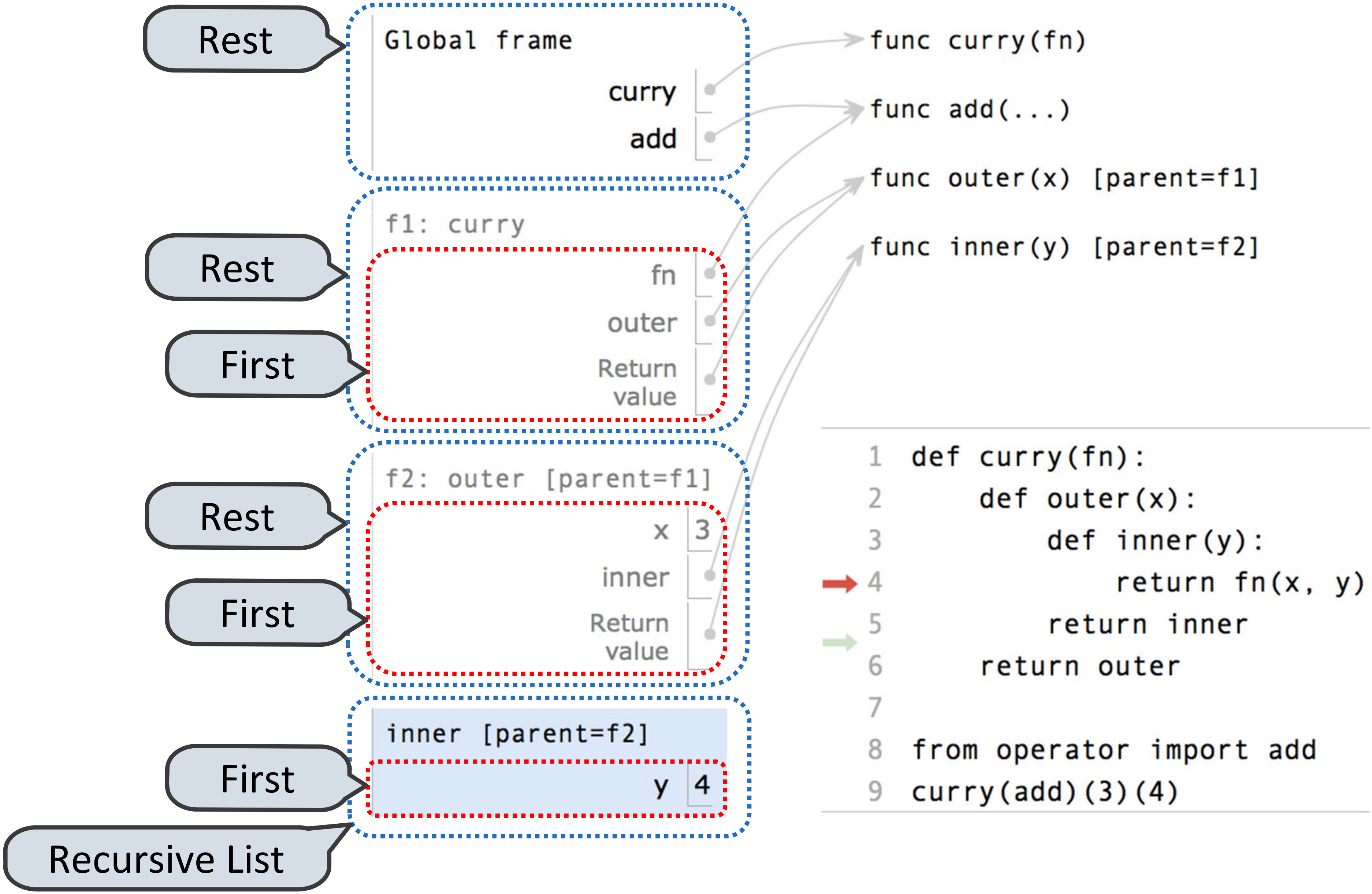
Example: Environments



```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Example: http://goo.gl/8DNY1

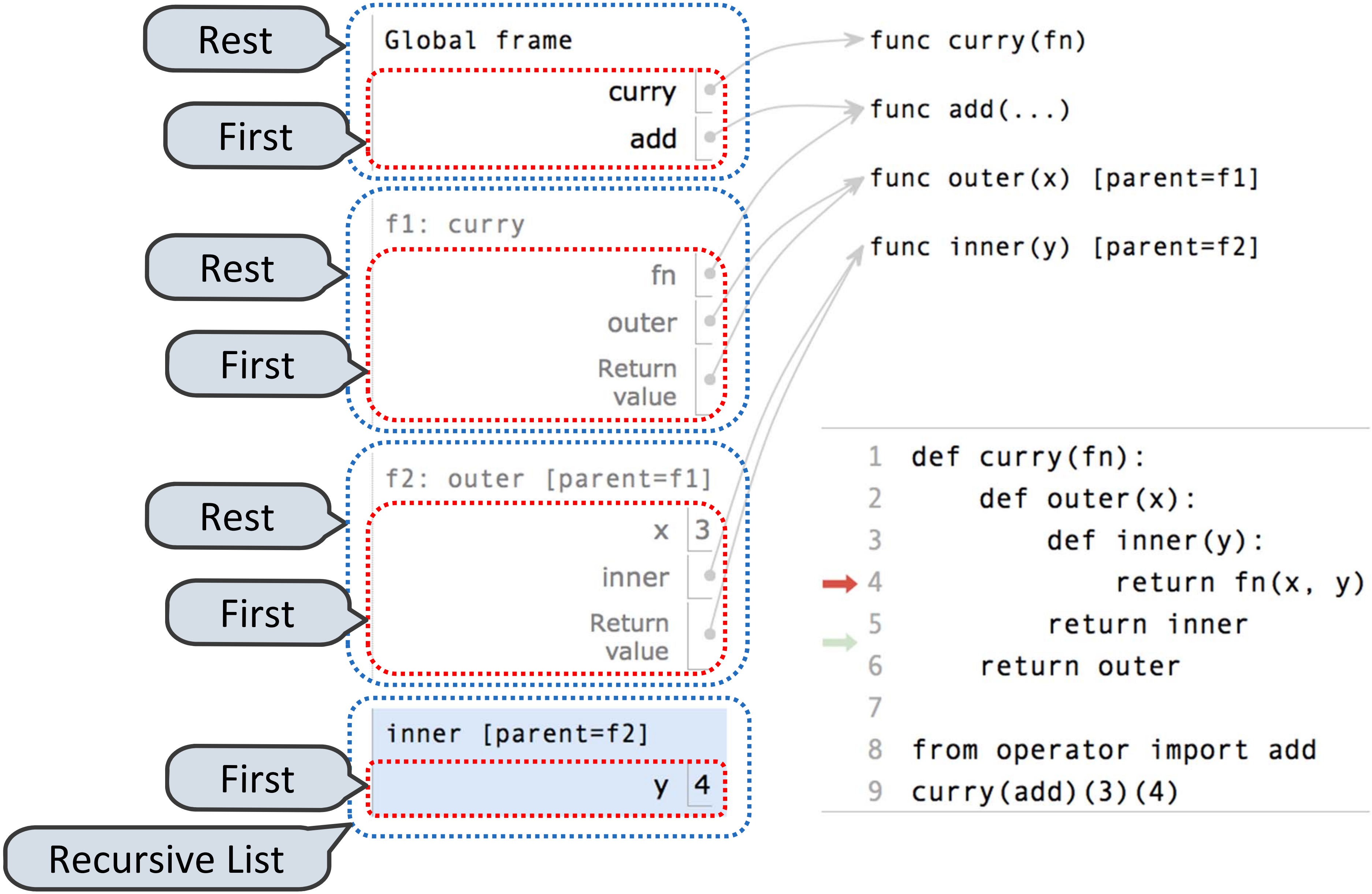
Example: Environments



```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Example: http://goo.gl/8DNY1

Example: Environments



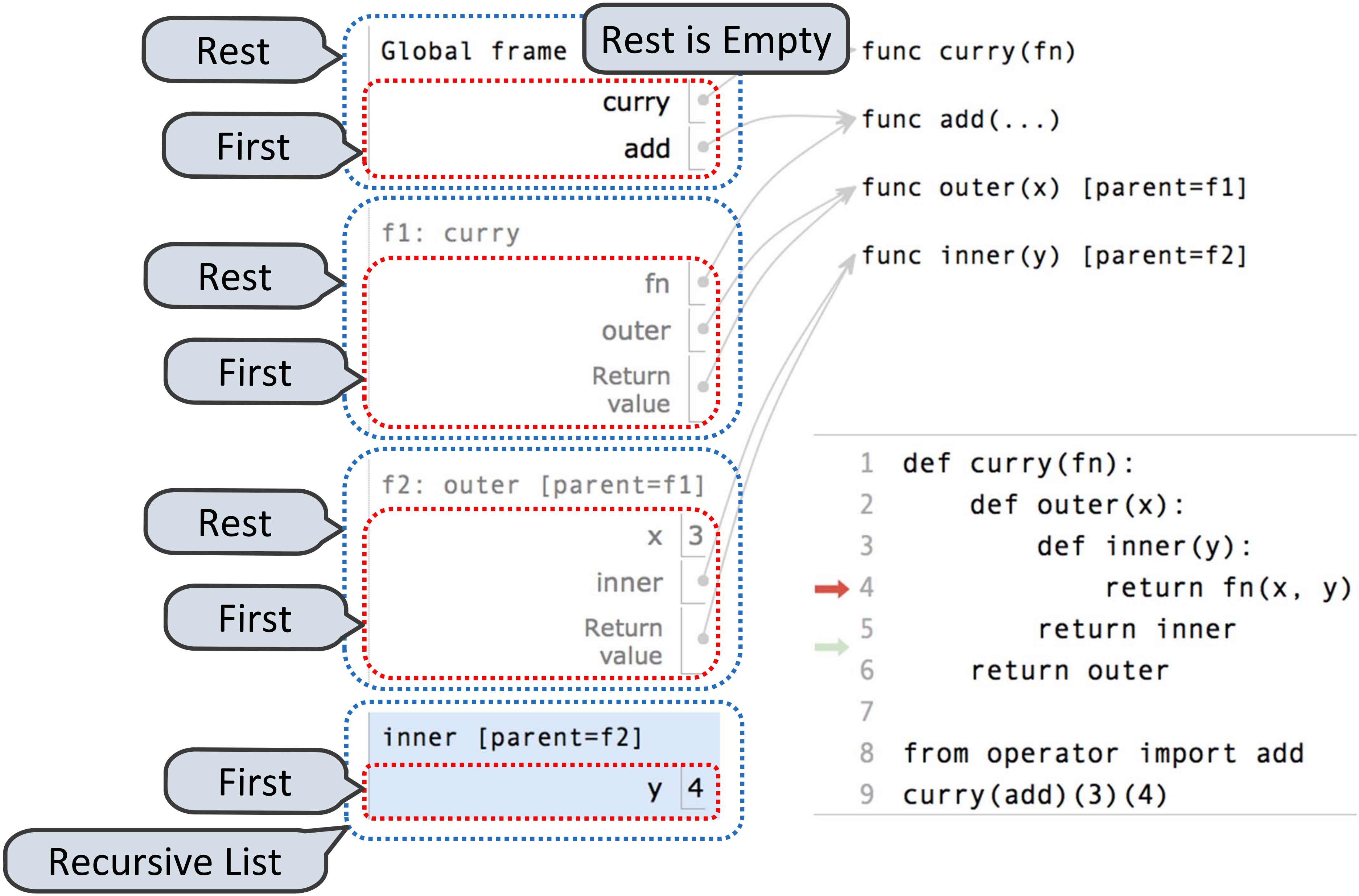
```
Global frame
        curry  •
          add  •

f1: curry
           fn  •
        outer  •
 Return value  •

f2: outer [parent=f1]
            x  3
        inner  •
 Return value  •

inner [parent=f2]
            y  4
```

```
func curry(fn)
func add(...)
func outer(x) [parent=f1]
func inner(y) [parent=f2]
```

```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Example: http://goo.gl/8DNY1

Example: Environments



```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

Example: http://goo.gl/8DNY1

Trees with Internal Node Values

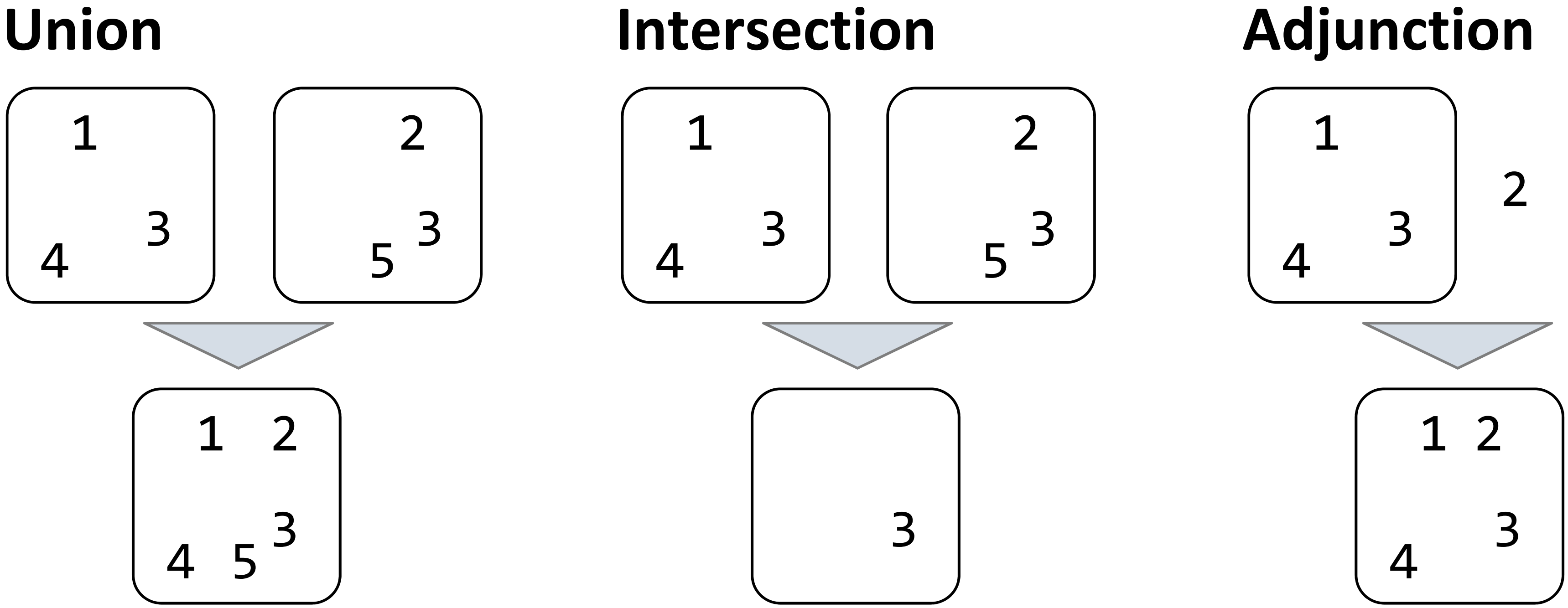
Trees can have values at internal nodes as well as their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right


def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n - 2)
    right = fib_tree(n - 1)
    return Tree(left.entry + right.entry, left, right)
```

Implementing Sets

What we should be able to do with a set:

- Membership testing: Is a value an element of a set?
- Union: Return a set with all elements in *set1* **or** *set2*
- Intersection: Return a set with any elements in *set1* **and** *set2*
- Adjunction: Return a set with all elements in *s* and a value *v*



Union

Intersection

Adjunction

Sets as Unordered Sequences

Proposal 1: A set is represented by a recursive list that contains no duplicate items

This is how we implemented dictionaries

```
def empty(s):
    return s is Rlist.empty

def set_contains(s, v):
    if empty(s):
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
```

Sets as Unordered Sequences

Sets as Unordered Sequences

```
def adjoin_set(s, v):
```

Sets as Unordered Sequences

```
def adjoin_set(s, v):
    if set_contains(s, v):
```

Sets as Unordered Sequences

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
```

Sets as Unordered Sequences

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

Sets as Unordered Sequences

Time order of growth

$\Theta(n)$

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

Sets as Unordered Sequences

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

Time order of growth

$\Theta(n)$

The size of the set

Sets as Unordered Sequences

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

Time order of growth

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
```

Sets as Unordered Sequences

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

Time order of growth

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Assume sets are the same size

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Assume sets are the same size

```
def union_set(set1, set2):
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Assume sets are the same size

```
def union_set(set1, set2):
    f = lambda v: not set_contains(set2, v)
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Assume sets are the same size

```
def union_set(set1, set2):
    f = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, f)
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Assume sets are the same size

```
def union_set(set1, set2):
    f = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, f)
    return extend_rlist(set1_not_set2, set2)
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Assume sets are the same size

```
def union_set(set1, set2):
    f = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, f)
    return extend_rlist(set1_not_set2, set2)
```

$\Theta(n^2)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

Order of growth?

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

Order of growth? $\Theta(n)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains2(s, v):
```

Order of growth? $\Theta(n)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains2(s, v):
    if empty(s) or s.first > v:
```

Order of growth? $\Theta(n)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
```

Order of growth? $\Theta(n)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
```

Order of growth? $\Theta(n)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
```

Order of growth? $\Theta(n)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

Order of growth?

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
    elif e1 < e2:
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
    elif e1 < e2:
        return intersect_set2(set1.rest, set2)
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
    elif e1 < e2:
        return intersect_set2(set1.rest, set2)
    elif e2 < e1:
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
    elif e1 < e2:
        return intersect_set2(set1.rest, set2)
    elif e2 < e1:
        return intersect_set2(set1, set2.rest)
```

Order of growth? $\Theta(n)$

Tree Sets

Tree Sets

Proposal 3: A set is represented as a Tree. Each entry is:

Tree Sets

Proposal 3: A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and

Tree Sets

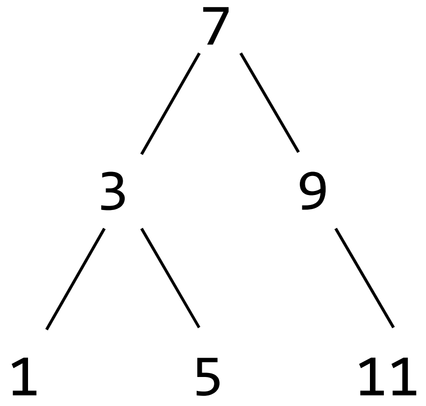
Proposal 3: A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and
- Smaller than all entries in its right branch

Tree Sets

Proposal 3: A set is represented as a Tree. Each entry is:

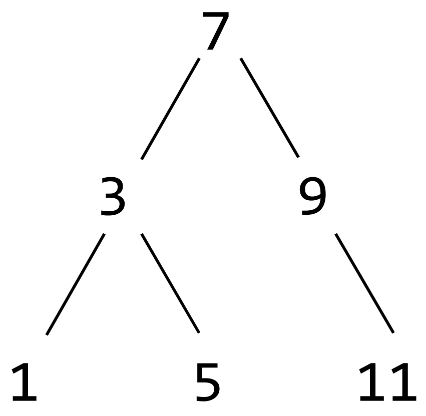
- Larger than all entries in its left branch and
- Smaller than all entries in its right branch

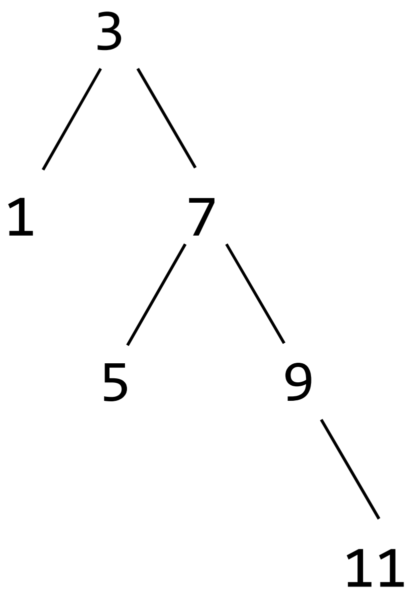


Tree Sets

Proposal 3: A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and
- Smaller than all entries in its right branch

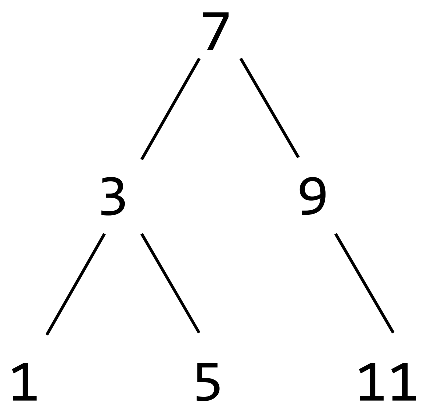


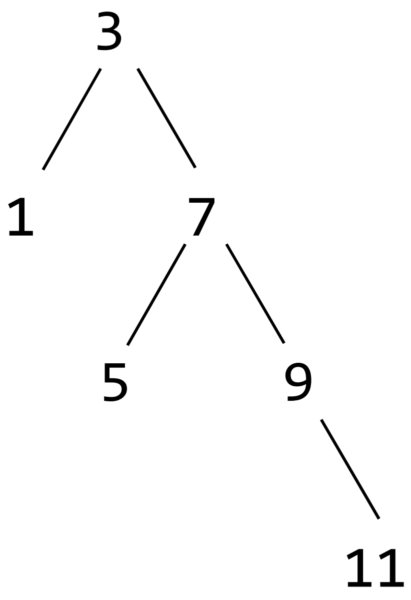


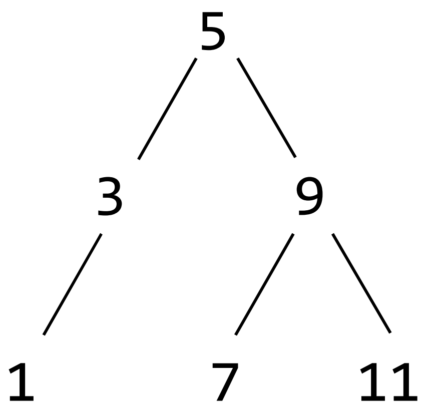
Tree Sets

Proposal 3: A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and
- Smaller than all entries in its right branch







Membership in Tree Sets

Membership in Tree Sets

Set membership tests traverse the tree

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

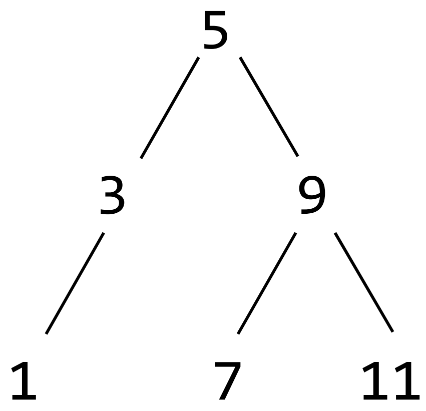
```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
        return set_contains3(s.left, v)
```

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
        return set_contains3(s.left, v)
```

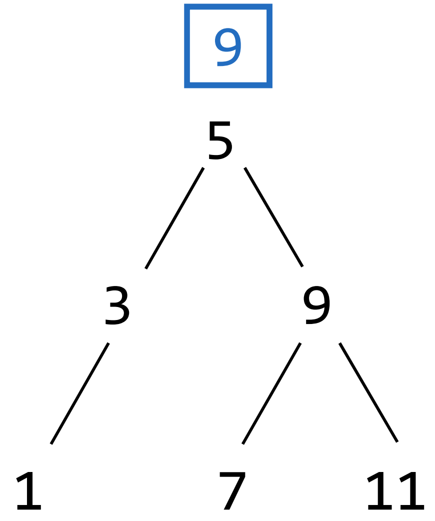


Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
        return set_contains3(s.left, v)
```

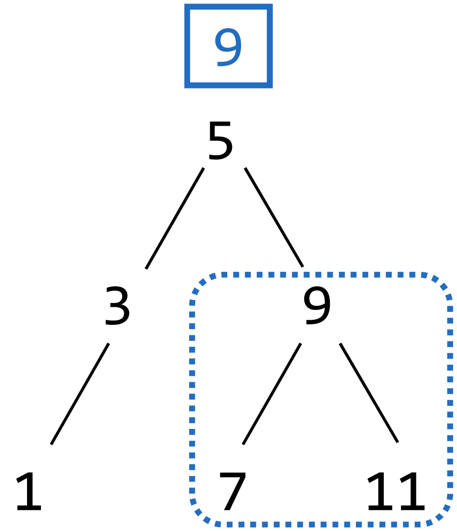


Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
        return set_contains3(s.left, v)
```



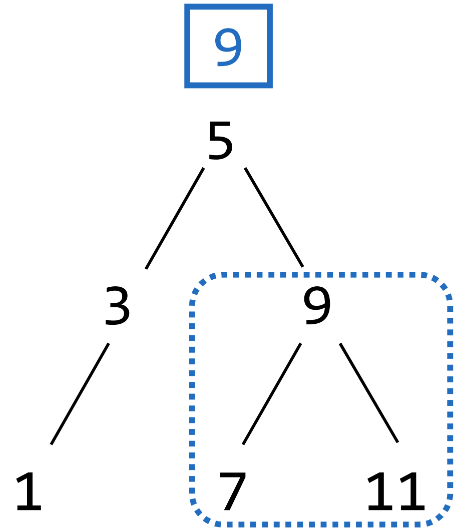
If 9 is in the set, it is in this branch

Membership in Tree Sets

Set membership tests traverse the tree

- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
        return set_contains3(s.left, v)
```

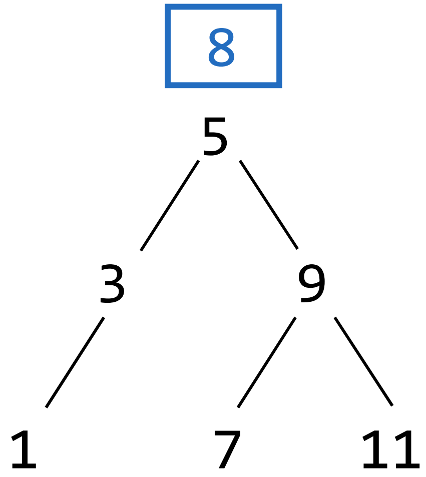


If 9 is in the set, it is in this branch

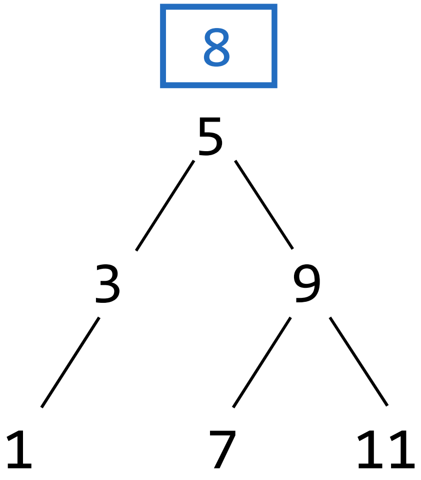
Order of growth?

Adjoining to a Tree Set

Adjoining to a Tree Set

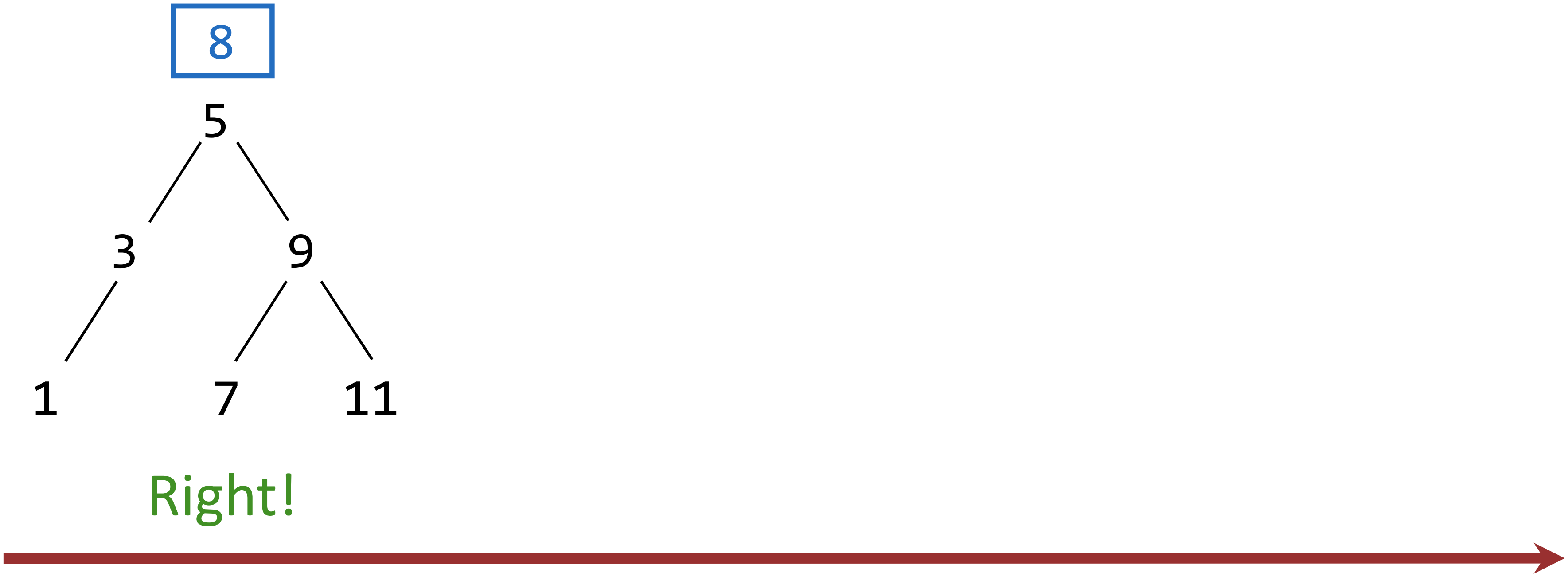


Adjoining to a Tree Set

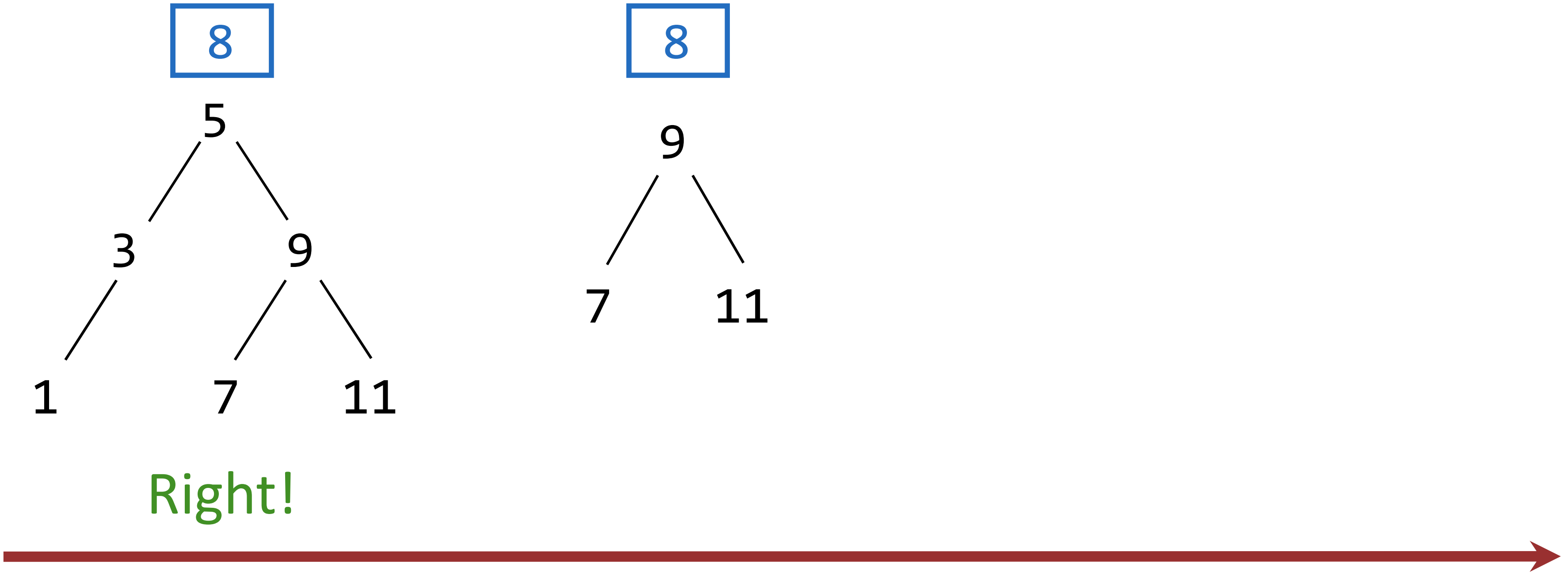


Right!

Adjoining to a Tree Set

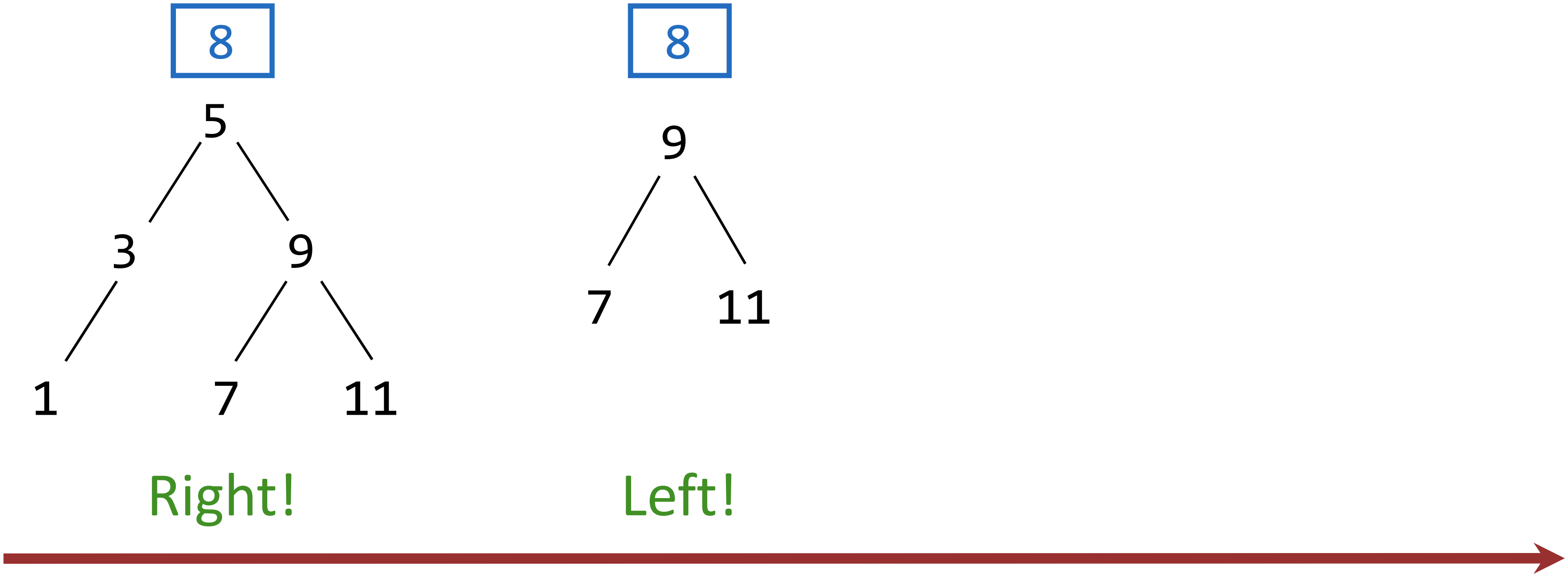


Adjoining to a Tree Set

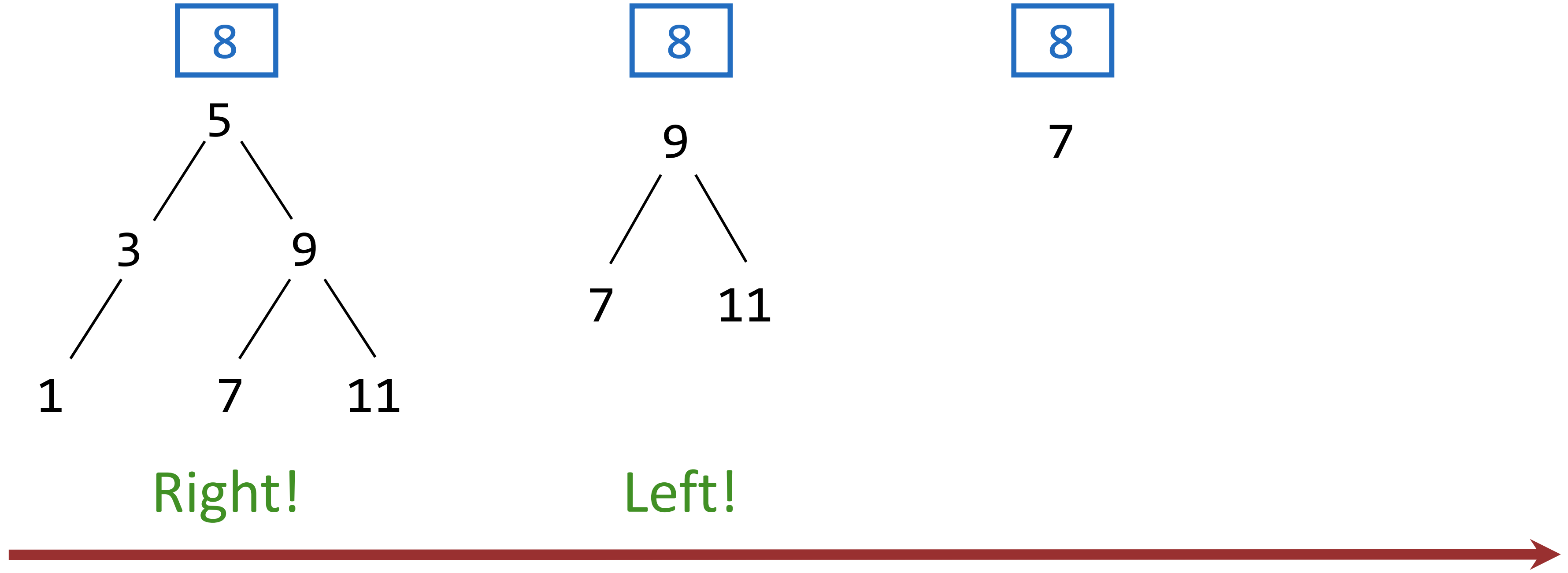


Right!

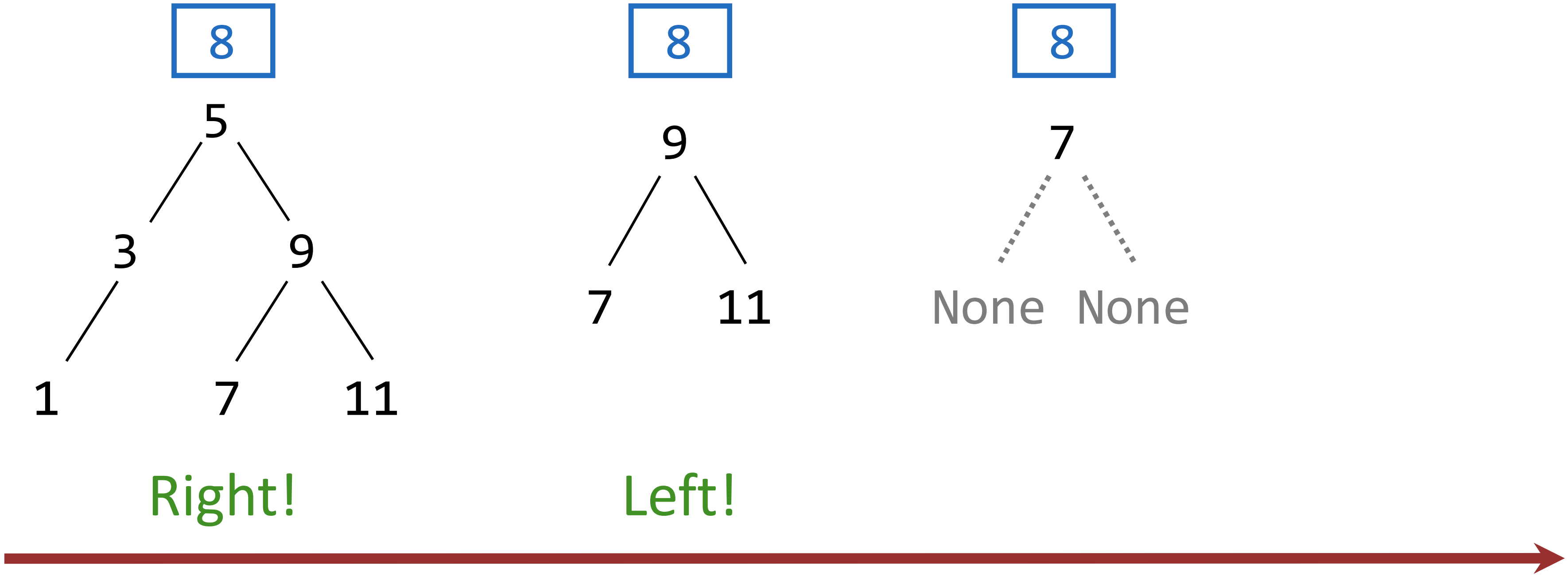
Adjoining to a Tree Set



Adjoining to a Tree Set



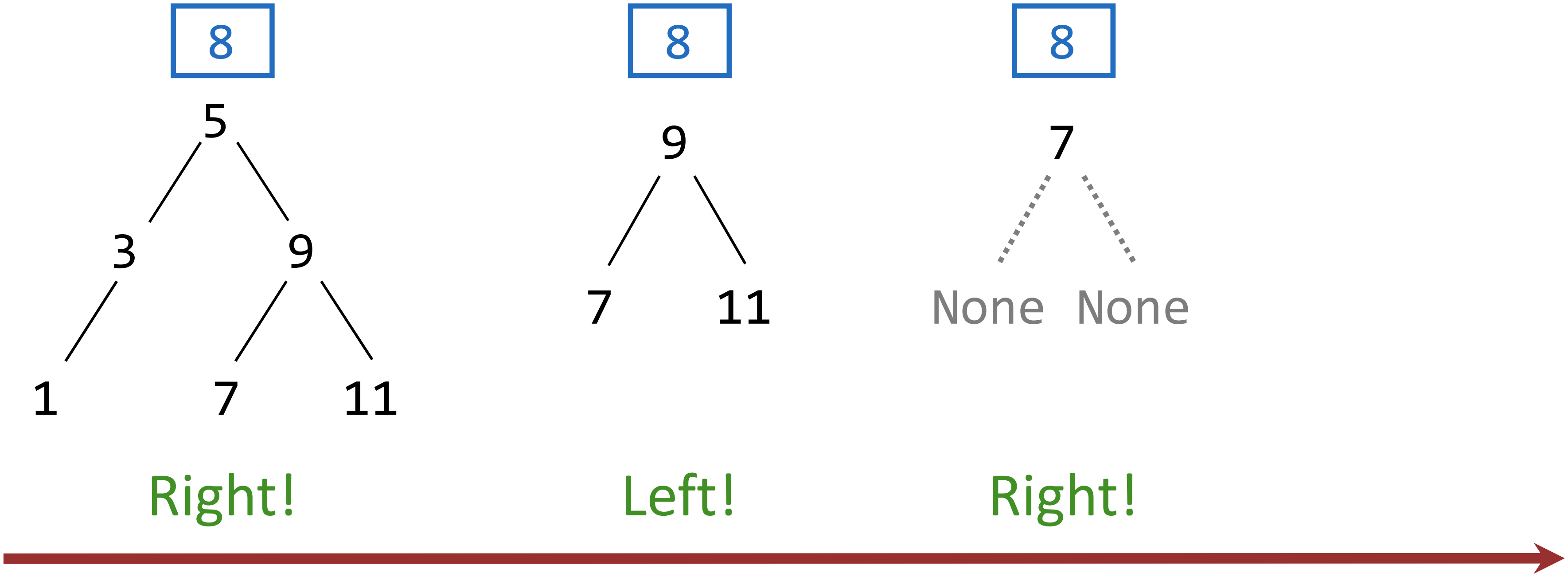
Adjoining to a Tree Set



Right!

Left!

Adjoining to a Tree Set

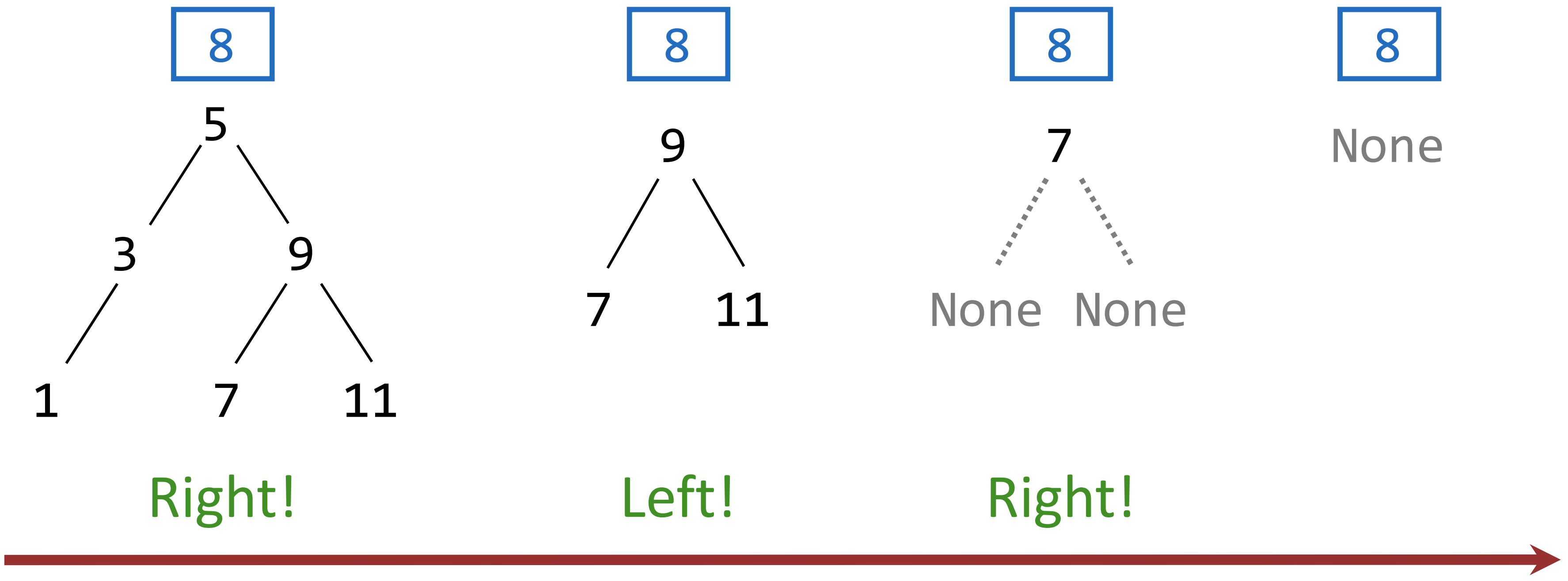


Right!

Left!

Right!

Adjoining to a Tree Set



8 8 8 8

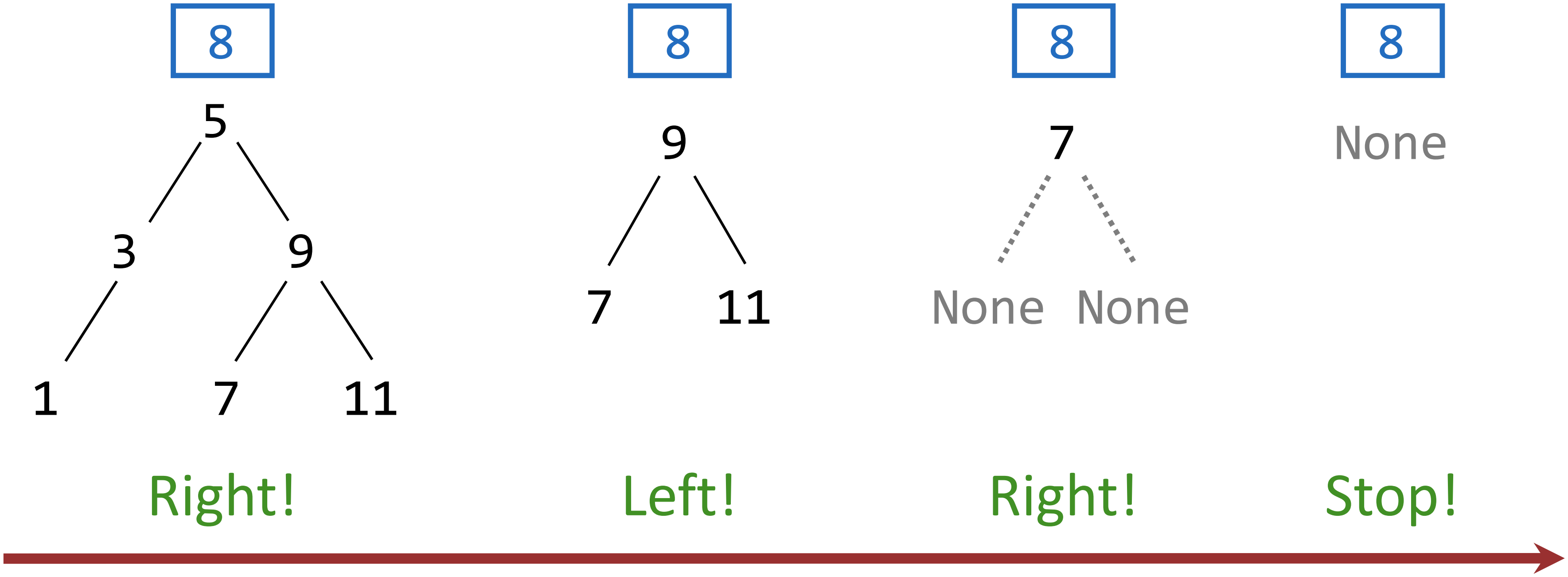
5 9 7 None

3 9 7 11 None None

1 7 11

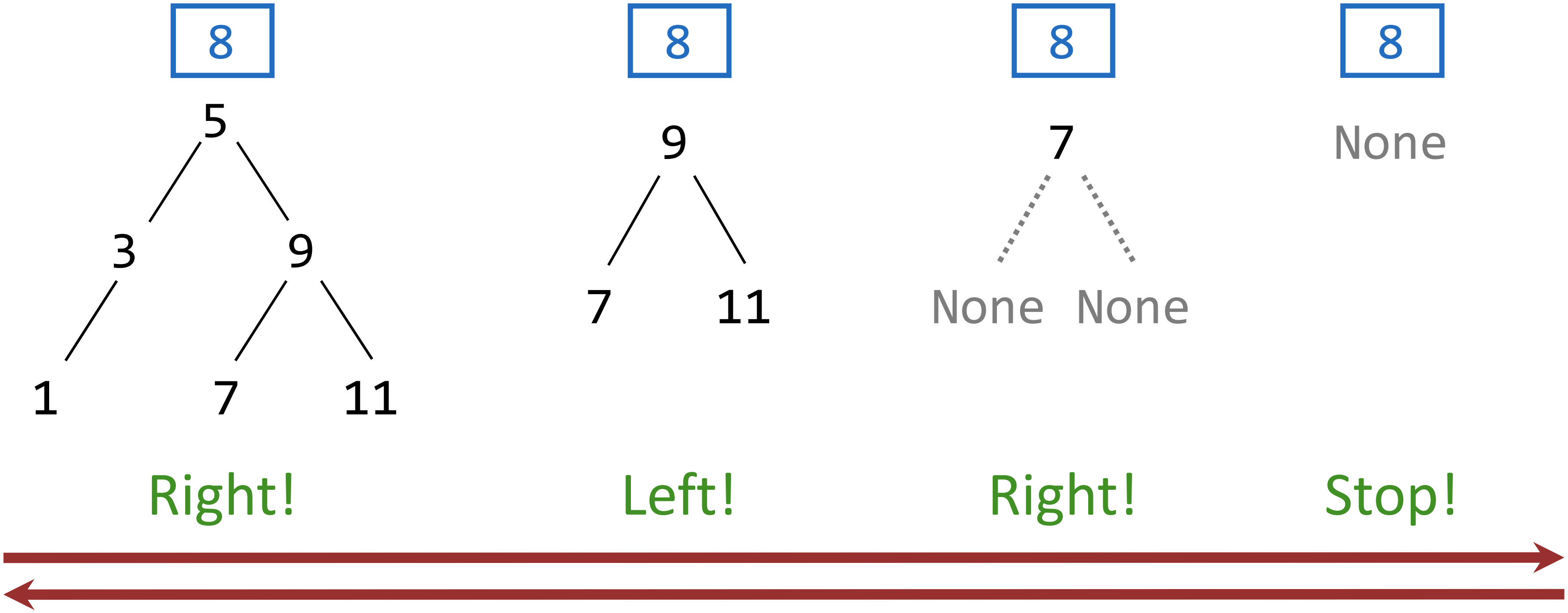
Right! Left! Right!

Adjoining to a Tree Set

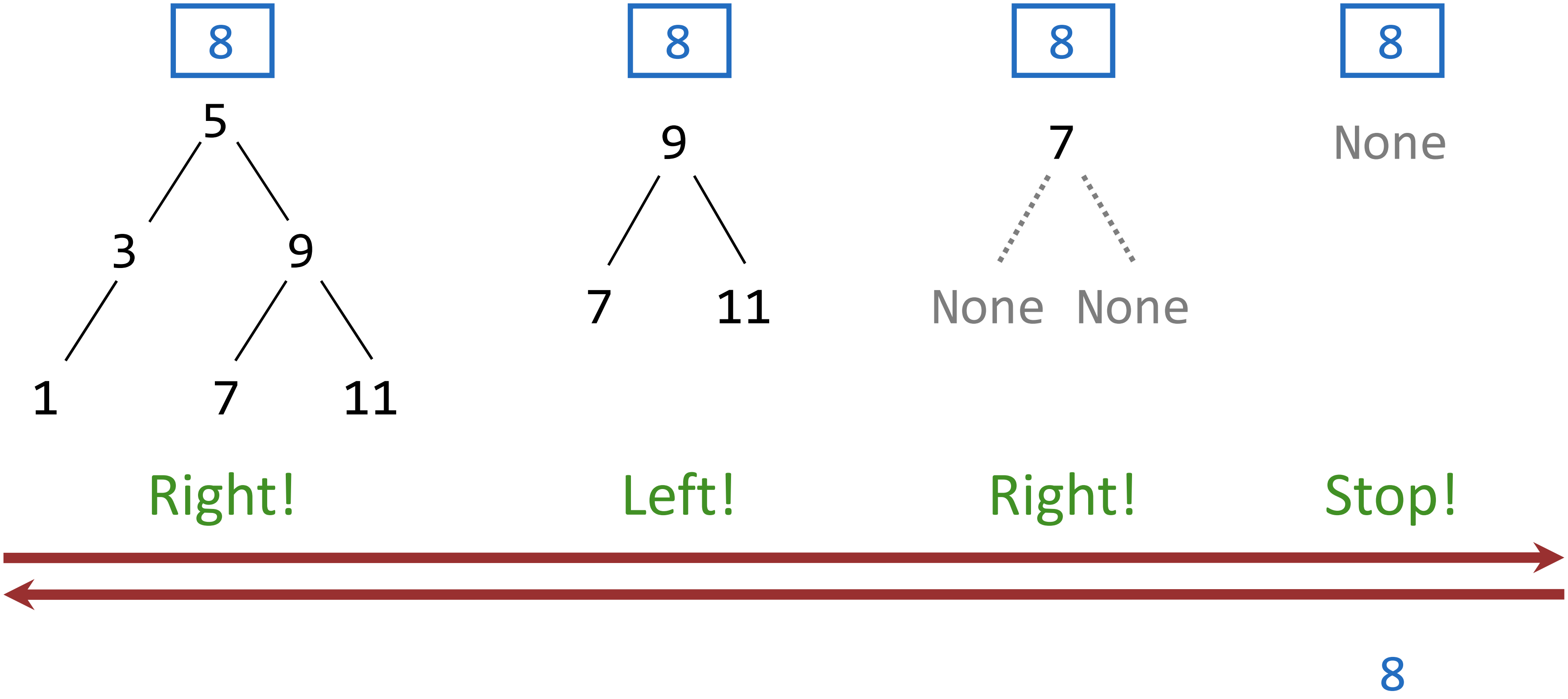


Right! Left! Right! Stop!

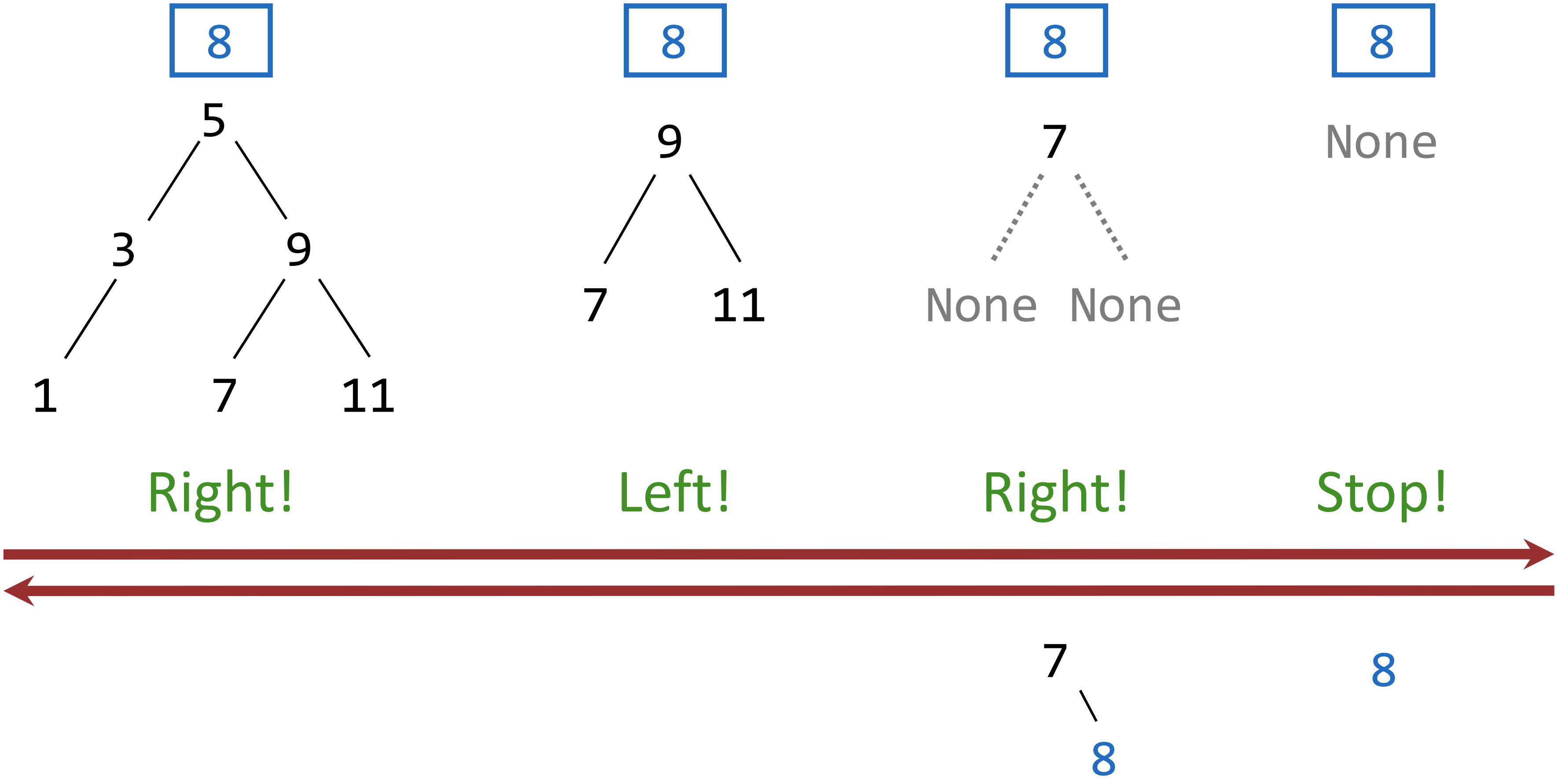
Adjoining to a Tree Set



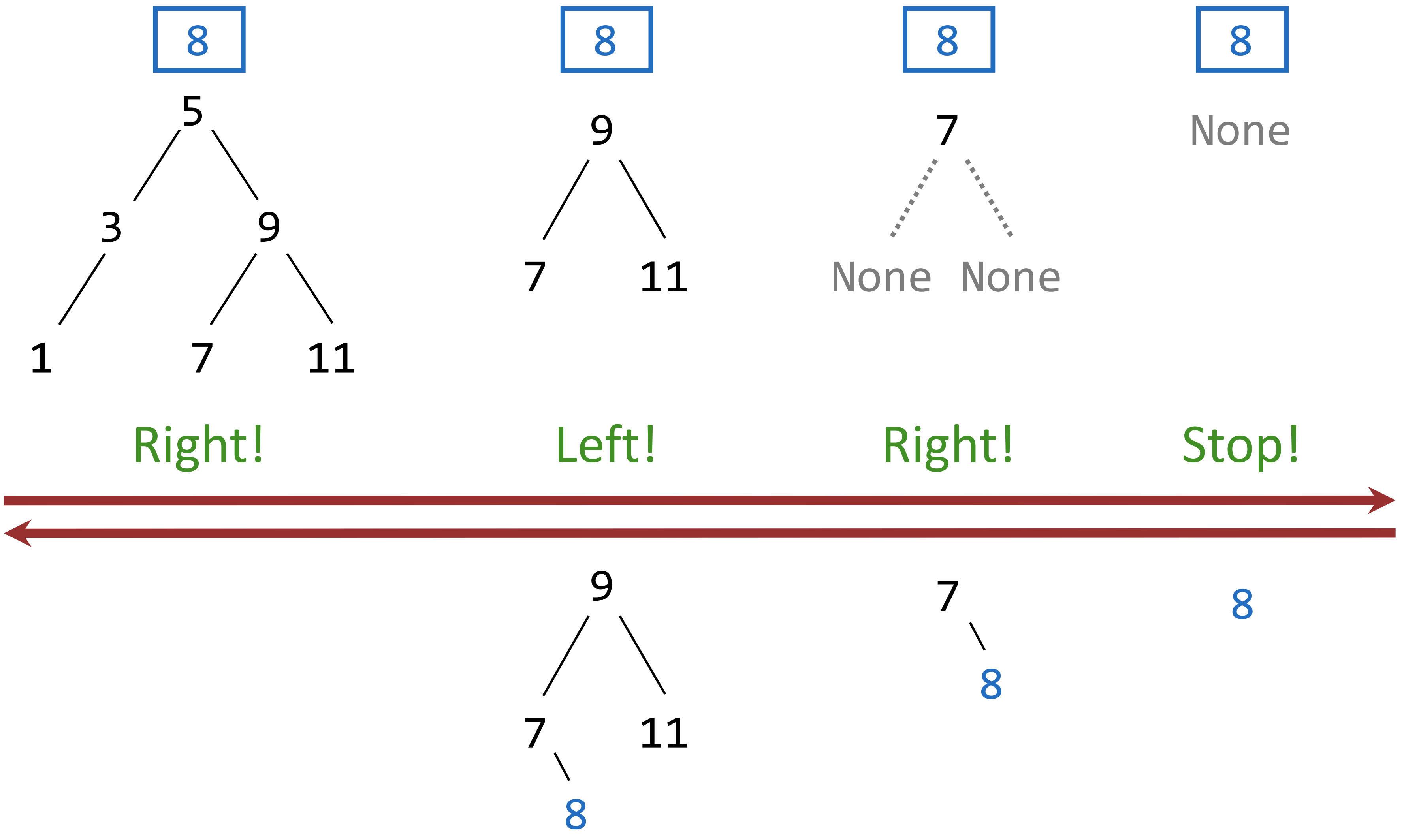
Adjoining to a Tree Set



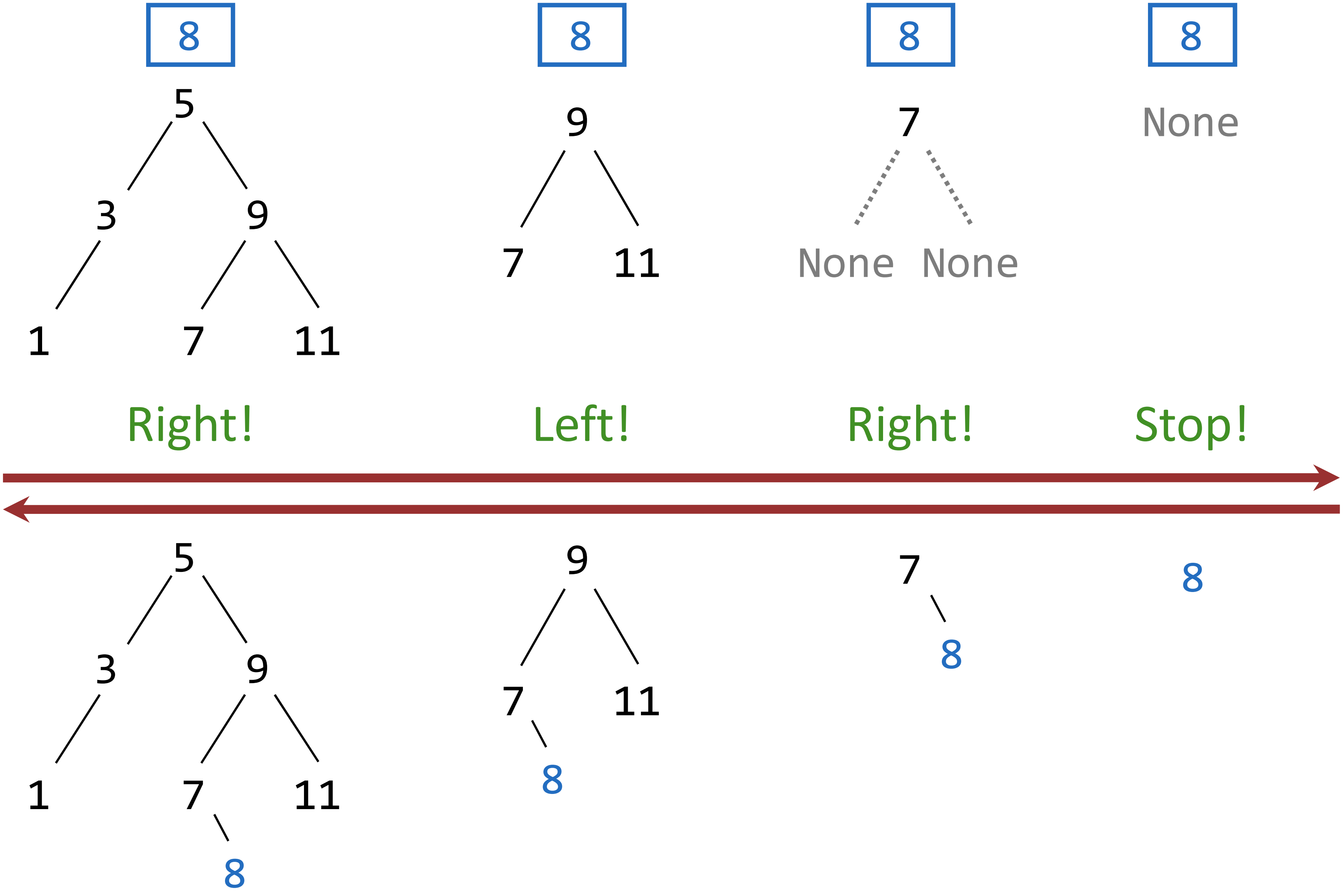
Adjoining to a Tree Set



Adjoining to a Tree Set



Adjoining to a Tree Set



What Did I Leave Out?

What Did I Leave Out?

Sets as ordered sequences:

What Did I Leave Out?

Sets as ordered sequences:

- Adjoining an element to a set

What Did I Leave Out?

Sets as ordered sequences:

- Adjoining an element to a set
- Union of two sets

What Did I Leave Out?

Sets as ordered sequences:

- Adjoining an element to a set
- Union of two sets

Sets as binary trees:

What Did I Leave Out?

Sets as ordered sequences:

- Adjoining an element to a set
- Union of two sets

Sets as binary trees:

- Intersection of two sets

What Did I Leave Out?

Sets as ordered sequences:

- Adjoining an element to a set
- Union of two sets

Sets as binary trees:

- Intersection of two sets
- Union of two sets

What Did I Leave Out?

Sets as ordered sequences:

- Adjoining an element to a set
- Union of two sets

Sets as binary trees:

- Intersection of two sets
- Union of two sets

That's homework 9!

Social Implications / Programming Practices

- Why things go wrong
- What can we do about this

Therac-25 Case Study

Therac-25 Case Study

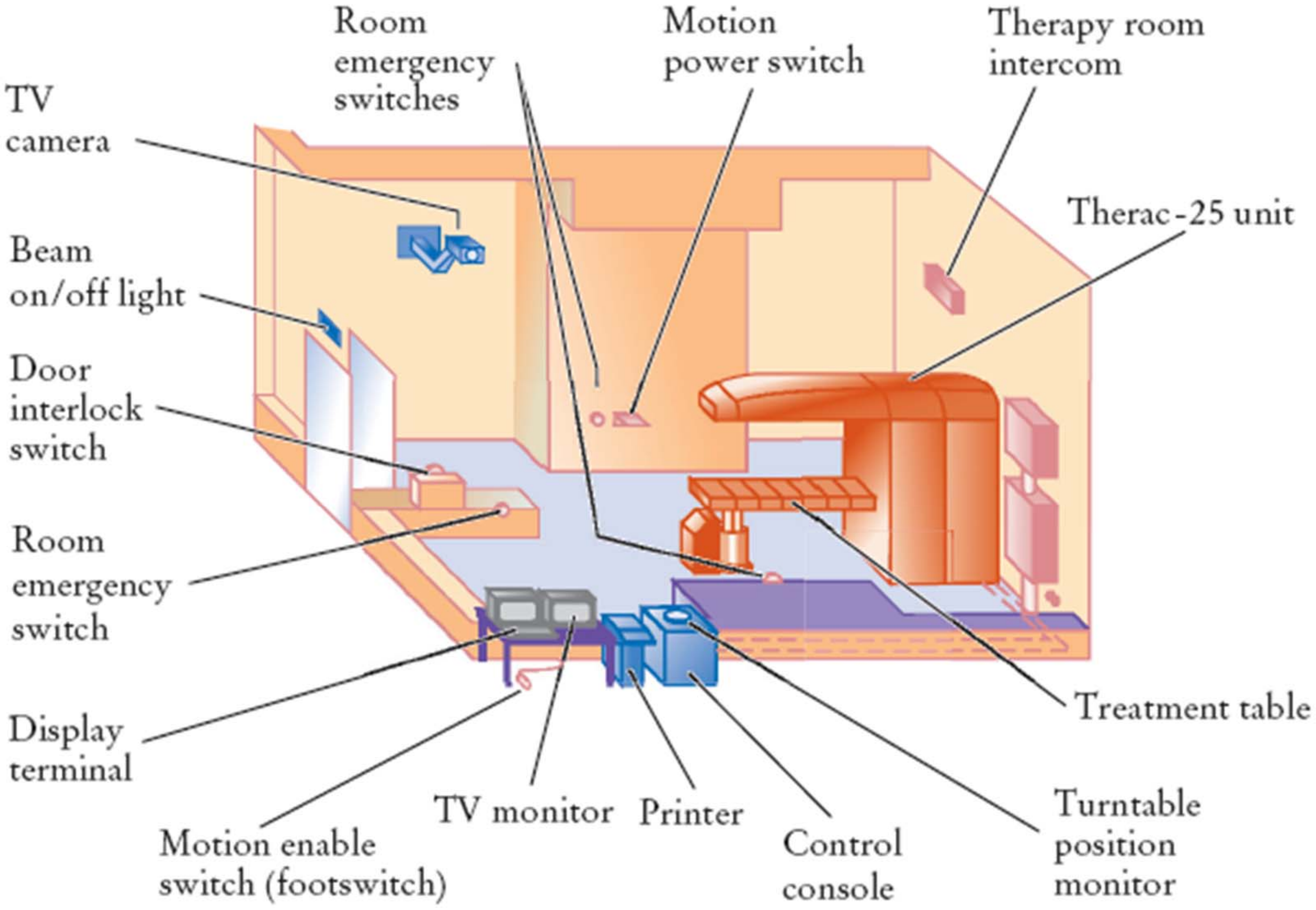


Figure 9 Typical Therac-25 Facility

Therac-25 Case Study

- Medical imaging device

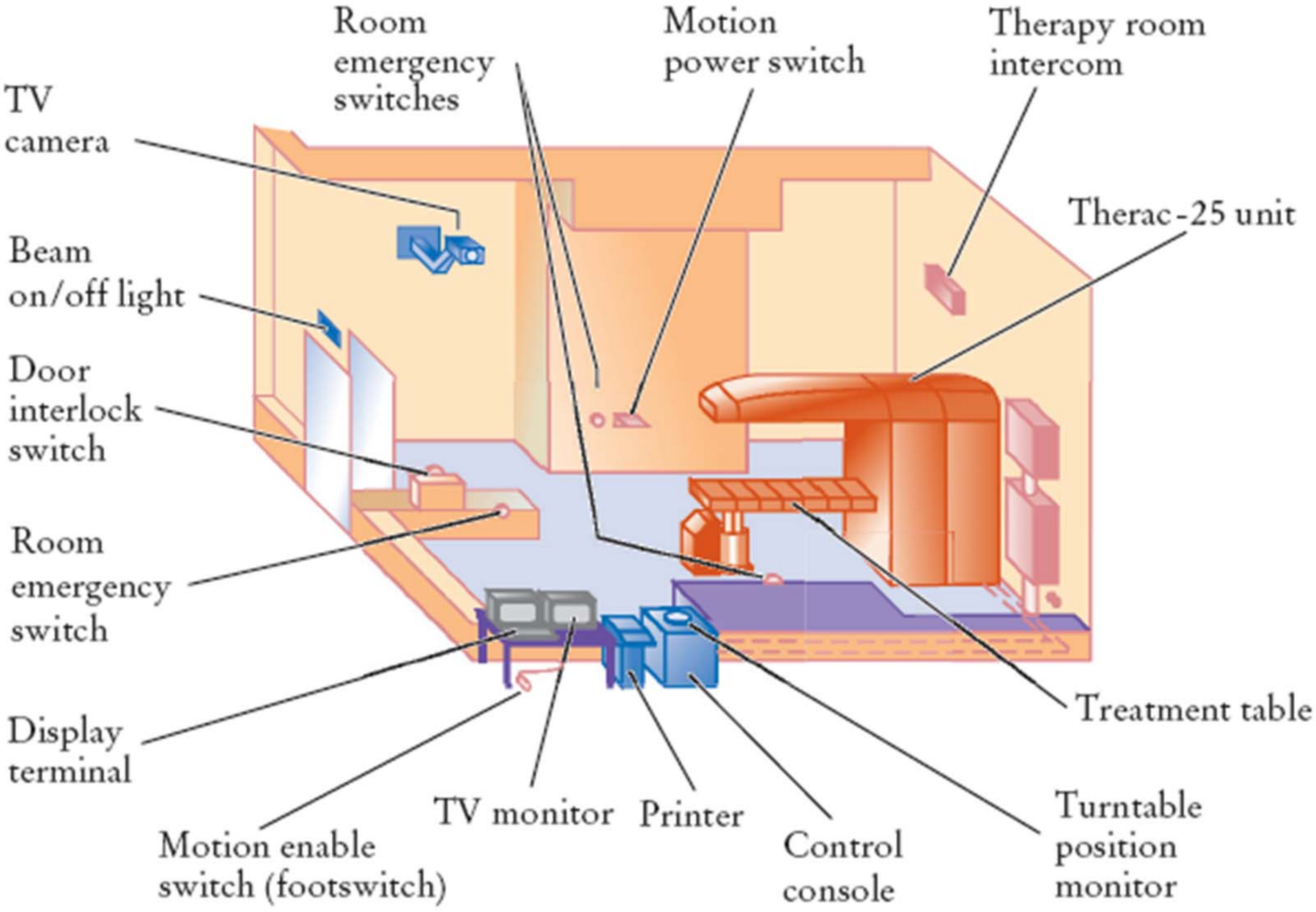


Figure 9 Typical Therac-25 Facility

Therac-25 Case Study

Therac-25 Case Study

- What happened?

Therac-25 Case Study

- What happened?
- 6 serious injuries

Therac-25 Case Study

- What happened?
- 6 serious injuries
- 4 deaths

Therac-25 Case Study

- What happened?
- 6 serious injuries
- 4 deaths
- Otherwise effective – saved hundreds of lives

Lesson to be learned

Lesson to be learned

- Social responsibility in engineering

Lesson to be learned

- Social responsibility in engineering
- First real incident of fatal software failure

Lesson to be learned

- Social responsibility in engineering
- First real incident of fatal software failure
- Bigger issue
  - No bad guys
  - Honestly believed there was nothing wrong

“Software Rot”

- Other engineering fields: clear sense of degradation and decay
- Can software become brittle or fractured?

A bigger picture

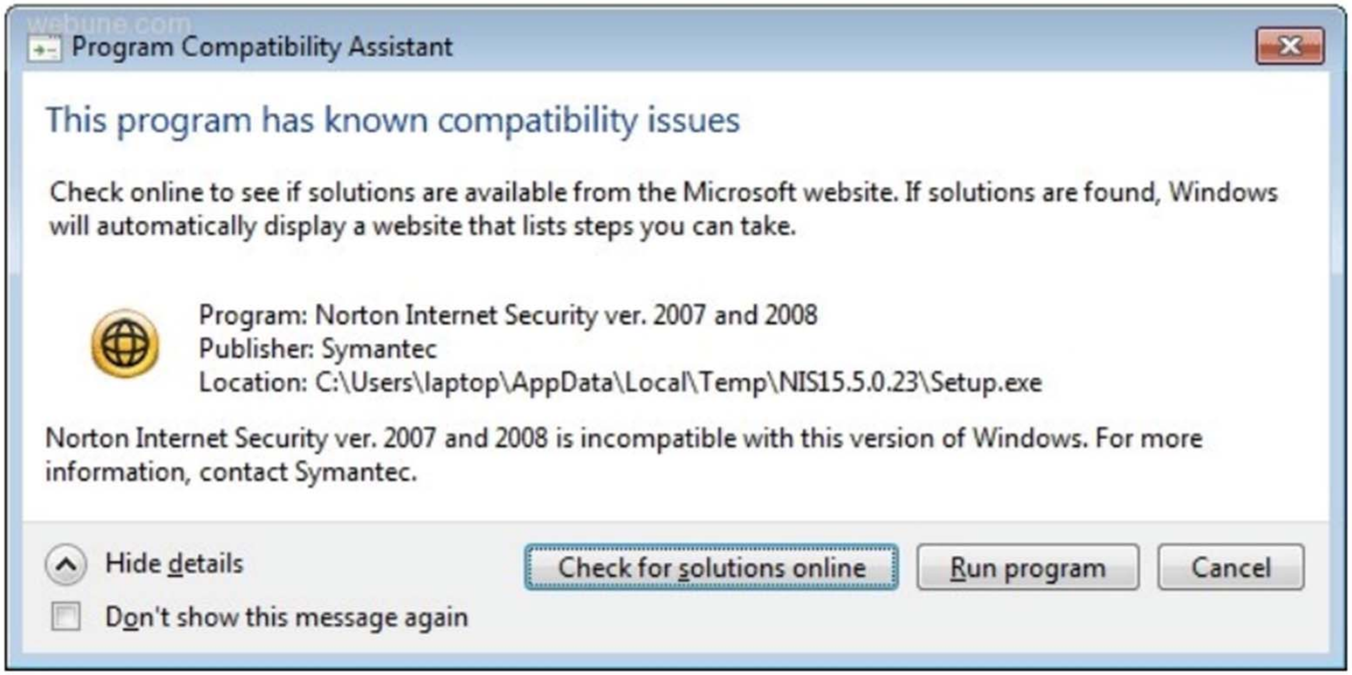
A bigger picture

- **All software is part of a bigger system**

A bigger picture

- **All software is part of a bigger system**
- Software degrades because:
  - Other piece of software changes
  - Hardware changes
  - Environment changes

Ex: Compatibility Issues



Program Compatibility Assistant

This program has known compatibility issues

Check online to see if solutions are available from the Microsoft website. If solutions are found, Windows will automatically display a website that lists steps you can take.

Program: Norton Internet Security ver. 2007 and 2008
Publisher: Symantec
Location: C:\Users\laptop\AppData\Local\Temp\NIS15.5.0.23\Setup.exe

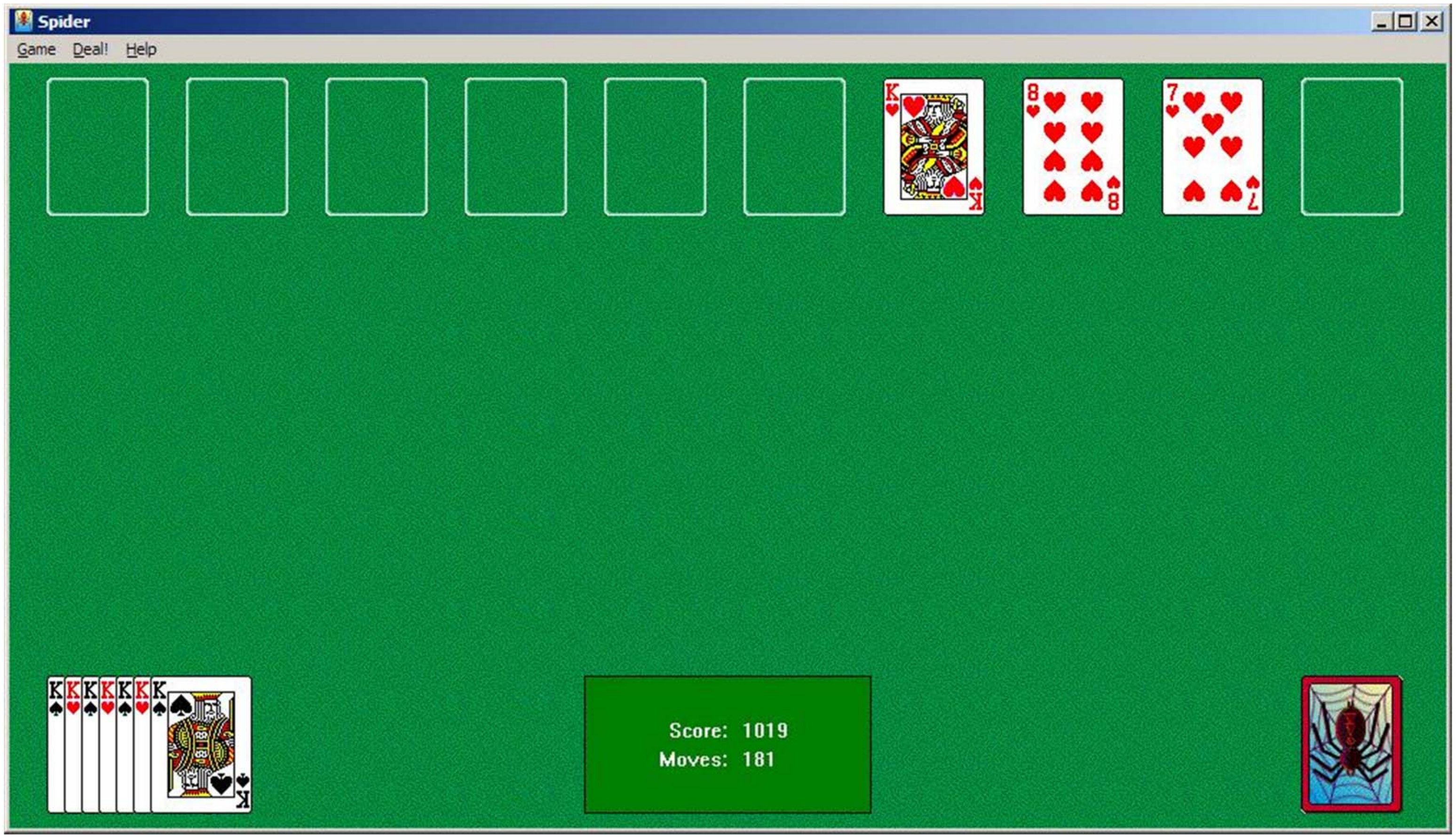
Norton Internet Security ver. 2007 and 2008 is incompatible with this version of Windows. For more information, contact Symantec.

A bigger issue

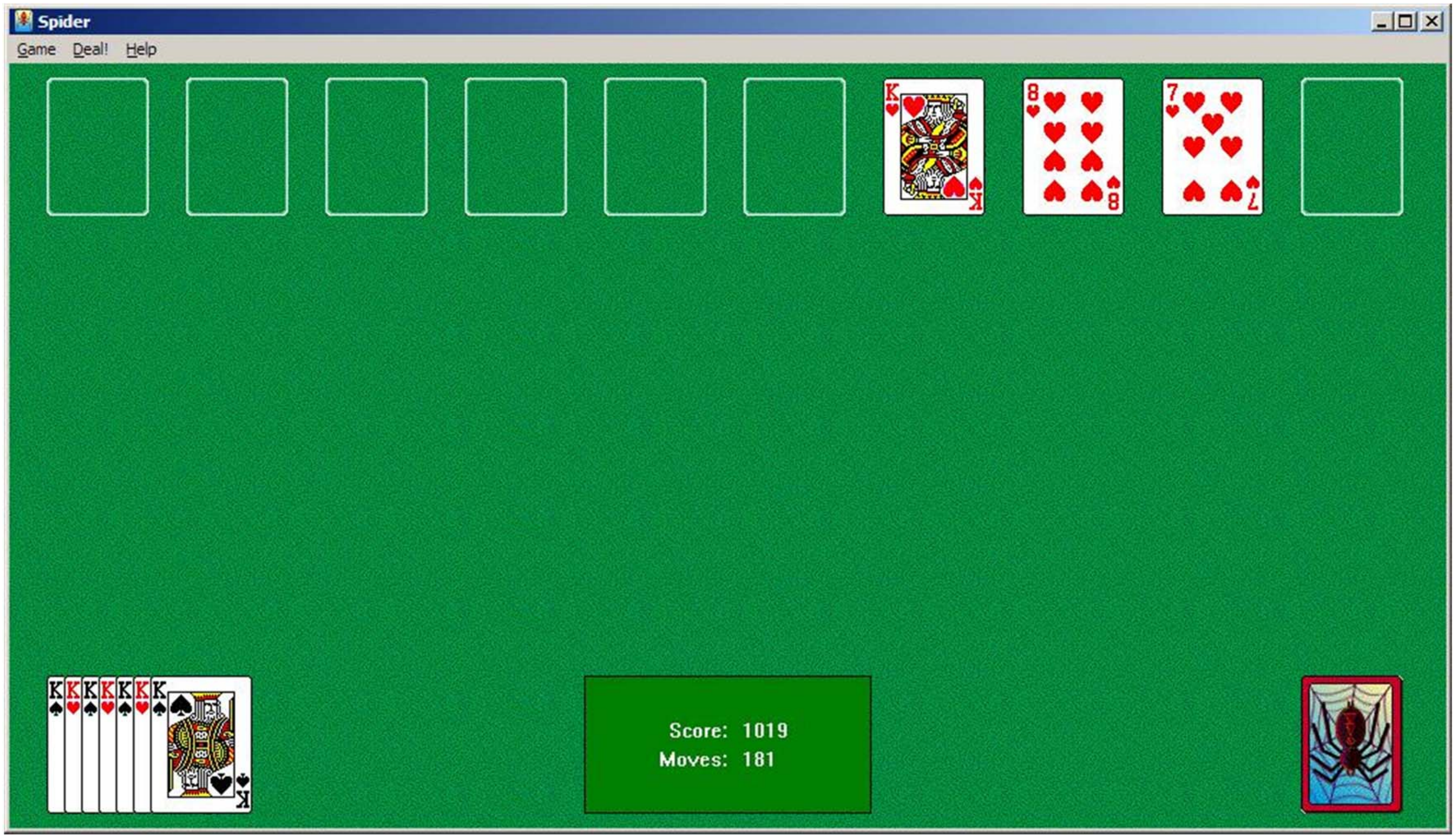
- The makers of the Therac did not fully understand the **complexity** of their software
- Complexity of constructs in other fields more apparent

A “simple” program

A “simple” program



A “simple” program



Score: 1019
Moves: 181

- □ This program can delete any file you can

Complexity in the Therac-25

Complexity in the Therac-25

- Abundant user interface issues

Complexity in the Therac-25

- Abundant user interface issues

- Cursor position and field entry

Complexity in the Therac-25

- Abundant user interface issues

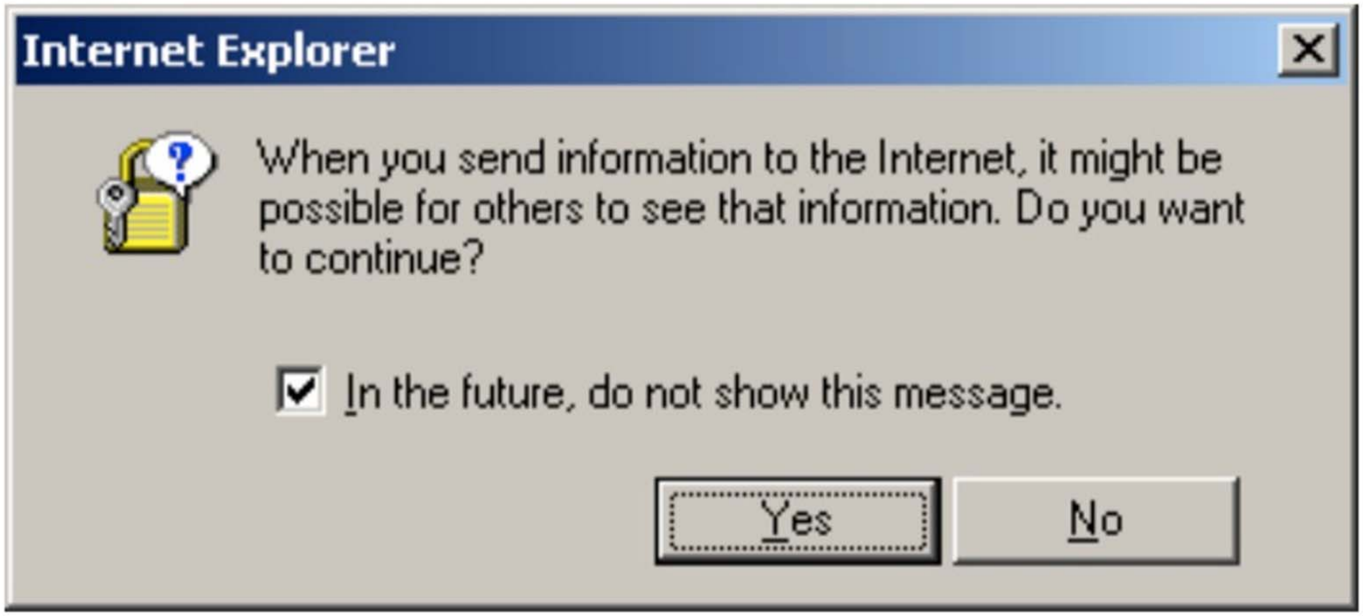
- Cursor position and field entry
- Default values

Complexity in the Therac-25

- □ Abundant user interface issues

- □ Cursor position and field entry
- □ Default values
- □ Too many error messages

Too many error messages

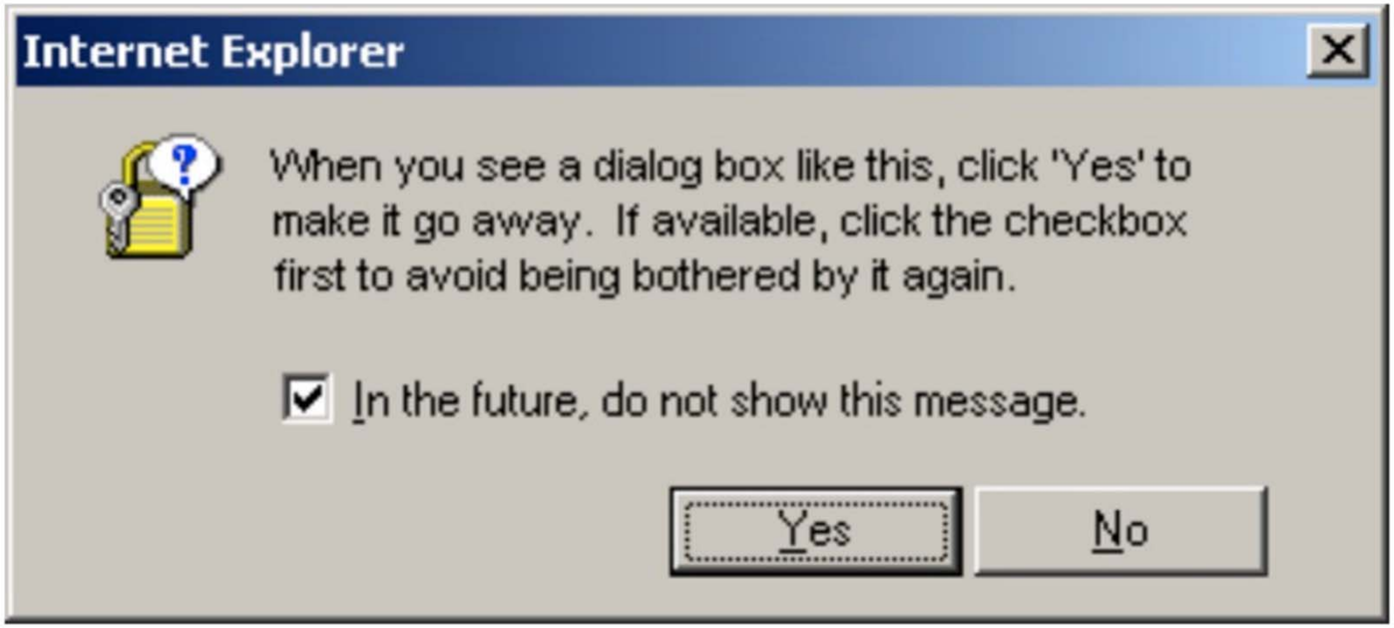


Internet Explorer

When you send information to the Internet, it might be possible for others to see that information. Do you want to continue?

☑ In the future, do not show this message.

Too many error messages



When you see a dialog box like this, click 'Yes' to make it go away. If available, click the checkbox first to avoid being bothered by it again.

☑ In the future, do not show this message.

(More) Complexity in the Therac-25

- No atomic test-and-set
- No hardware interlocks

How can we solve these things?

- □ Know your user
- □ Fail-Soft (or Fail-Safe)
- □ Audit Trail
- □ Correctness from the start
- □ Redundancy

Fail-Soft (or Fail-Safe)

```
def mutable_rlist():
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'first':
            return first(contents)
        if message == 'rest':
            return rest(contents)
        if message == 'len':
            return len_rlist(contents)
        ...


    return dispatch
```

Fail-Soft (or Fail-Safe)

```
def mutable_rlist():
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'first':
            return first(contents)
        if message == 'rest':
            return rest(contents)
        if message == 'len':
            return len_rlist(contents)
        ...
        else:
            print('Unknown message')
    return dispatch
```

Correctness from the start

- Edsger Dijkstra: “On the Cruelty of Really Teaching Computing Sciences”
- CS students shouldn’t use computers
- Rigorously prove correctness of their programs

- Correctness proofs
- Compilation (pre-execution) analysis

On debugging

□ Black box debugging

□ Glass box debugging

□ Don’t break what works

□ Golden rule of debugging…

Golden rule of debugging

- “Debug by subtraction, not by addition”
  - Prof. Brian Harvey