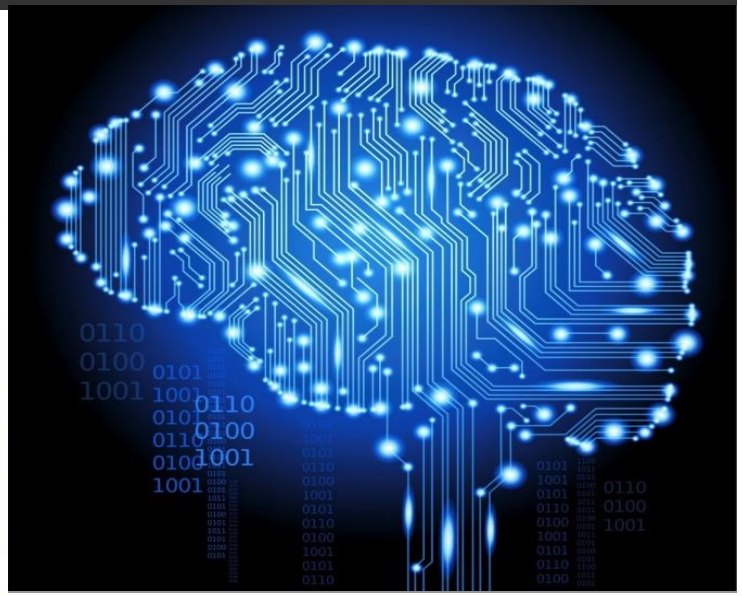
MONASH University

Information Technology

# Trees II

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor



# Objectives for these two lectures

- **To look at Expression Trees and their usefulness**
- **To understand:**
  - The concept of binary search tree
  - The basic operations on these trees (insert and search), and to be able to implement and modify them using linked nodes
  - The advantages and disadvantages of binary search trees over sorted lists (array/linked)
  - To make Binary Trees iterable

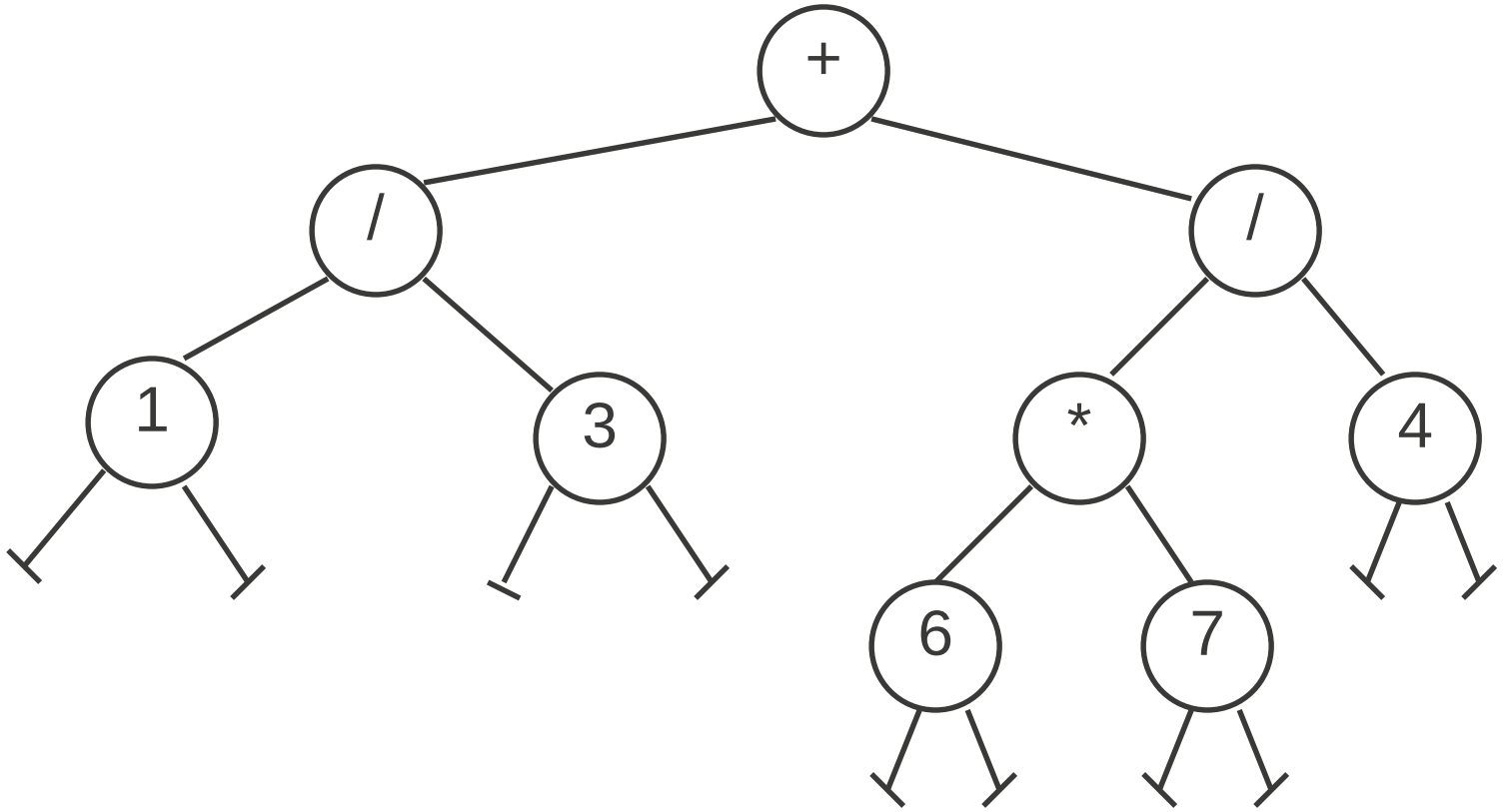
MONASH University

# Expression Trees

# Another kind of tree: Expression Tree

- **A Binary Tree built with**
  - Operands (leaves)
  - Operators (inner nodes)
- **Also known as a parse tree**
- **Used in compilers to represent expressions such as algebraic or Boolean expressions**

# Example: Expression Tree

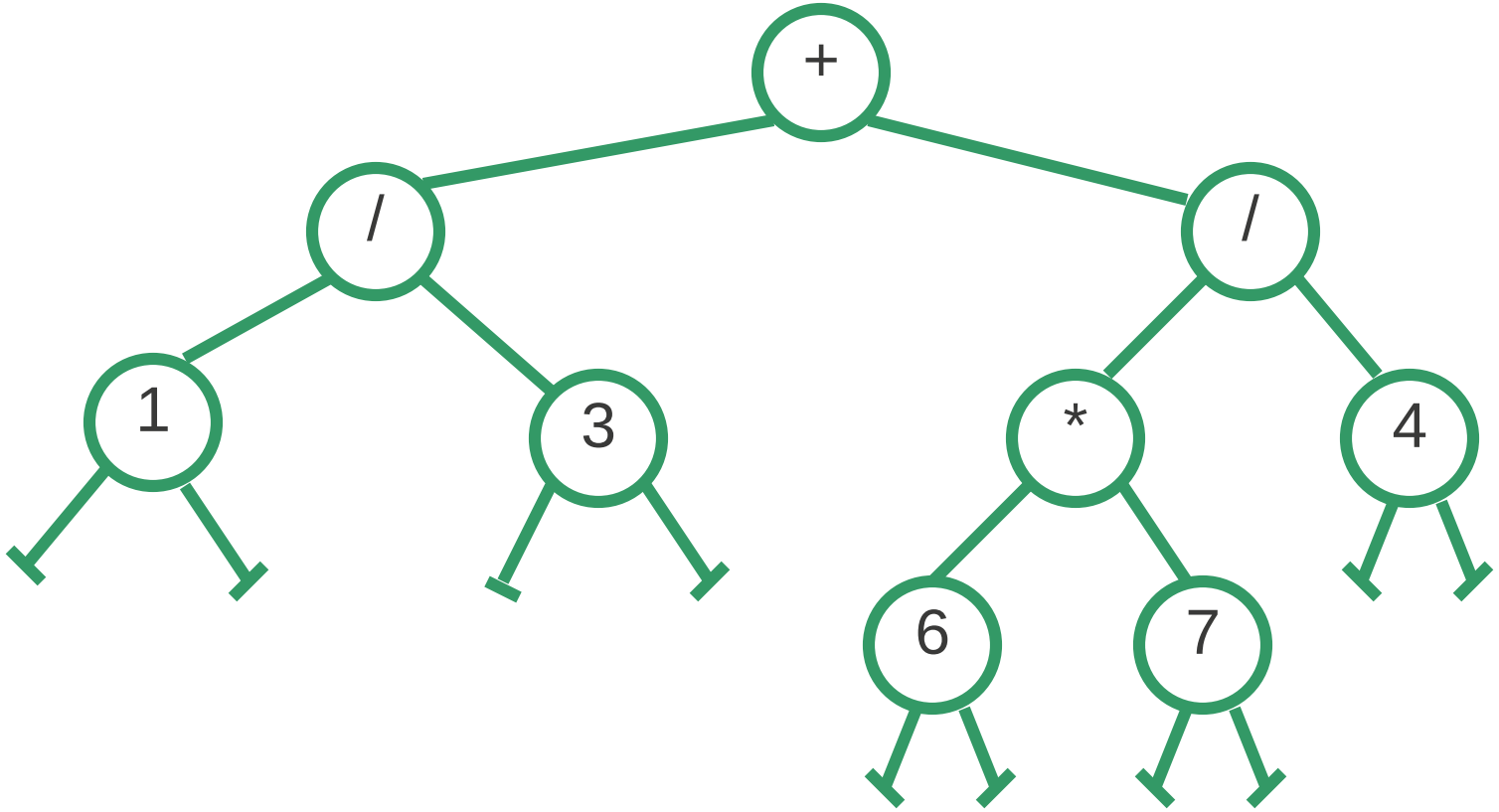



(1/3) + ((6*7) / 4)

# Tree Traversals and Expression Notations

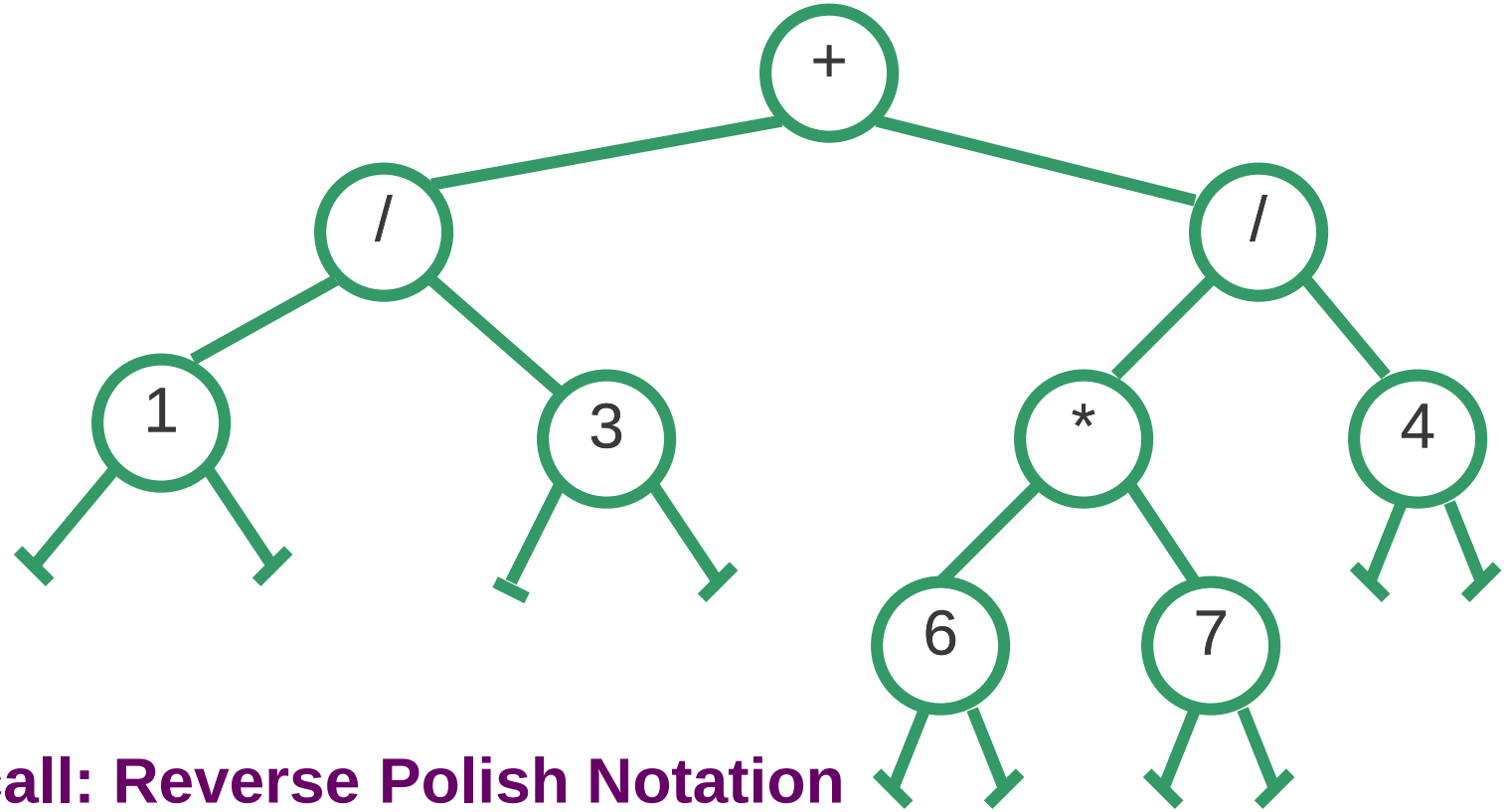
- **Preorder traversal corresponds to:**
  - Prefix Notation
- **Inorder traversal to:**
  - Infix Notation
- **Postorder traversal to:**
  - Postfix Notation

# Example: Infix




| 1 | / | 3 | + | 6 | * | 7 | / | 4 |
|---|---|---|---|---|---|---|---|---|

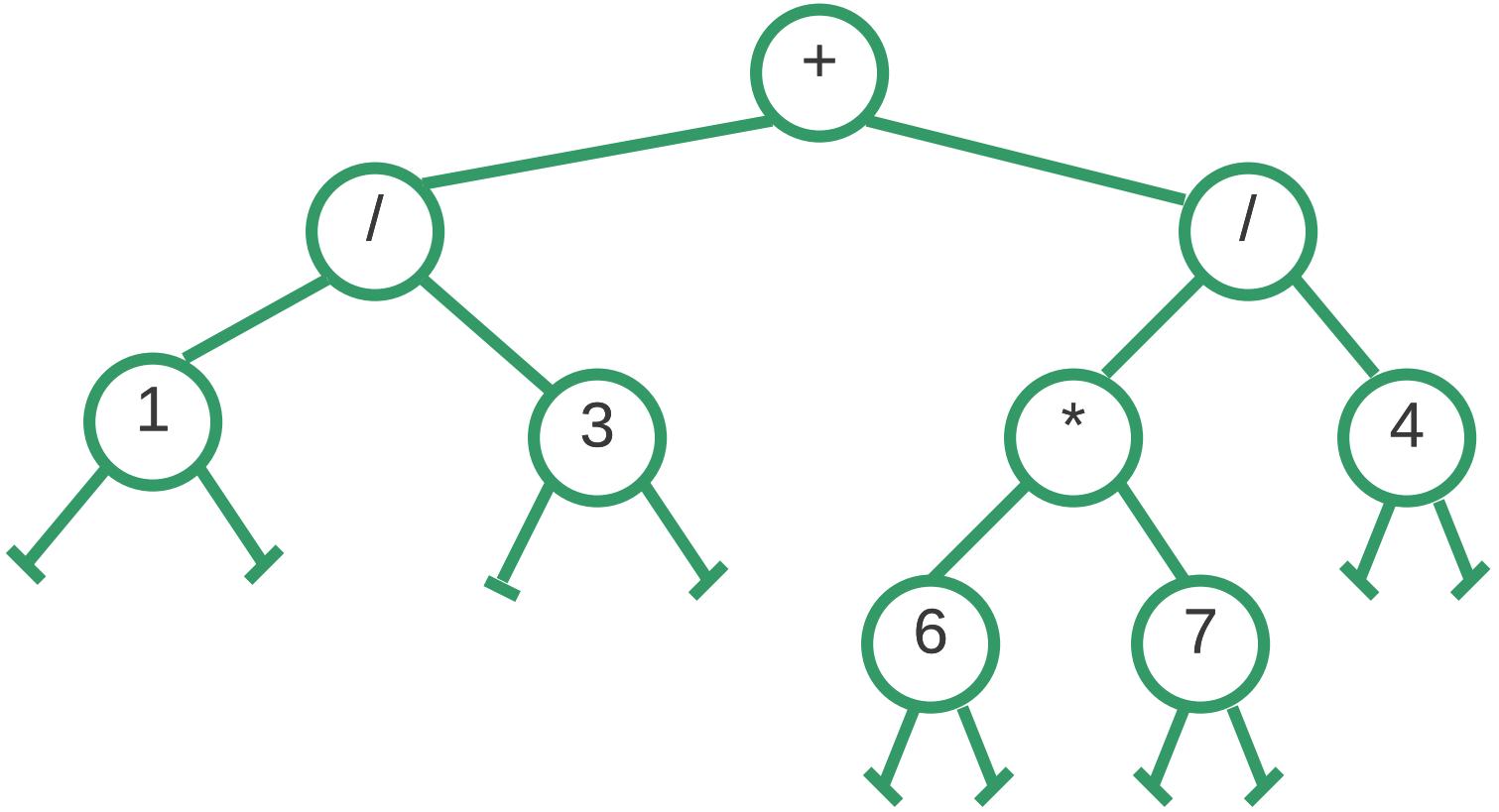
# Example: Postfix




**Recall: Reverse Polish Notation**

| 1 | 3 | / | 6 | 7 | * | 4 | / | + |
|---|---|---|---|---|---|---|---|---|

# Example: Prefix




| + | / | 1 | 3 | / | * | 6 | 7 | 4 |
|---|---|---|---|---|---|---|---|---|

# Recall Python dictionaries

```
>>> x = dict()
>>> x[1152]="Maria"
>>> x[4563]="Julian"
>>> x[1324]="Pierre"
>>> x
{1152: 'Maria', 4563: 'Julian', 1324: 'Pierre'}
>>> x[132]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 132
>>> x[1324]
'Pierre'
>>>
```

insert

search

Python dictionaries are implemented using Hash Tables

But you could also use a Binary Search Tree!

MONASH University

# Binary Search Trees (BST)

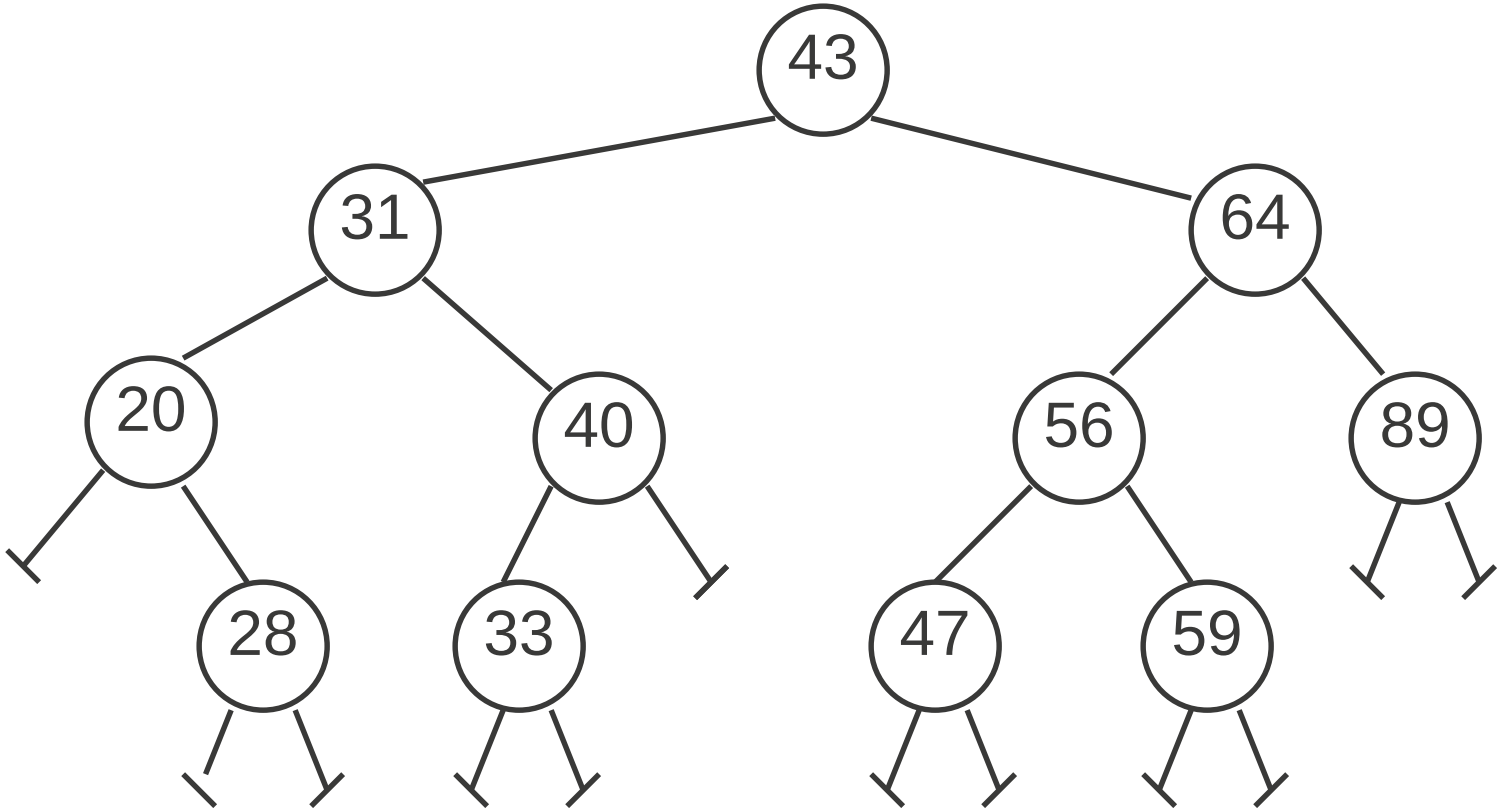
# Binary Search Tree

**A Binary Tree in which:**

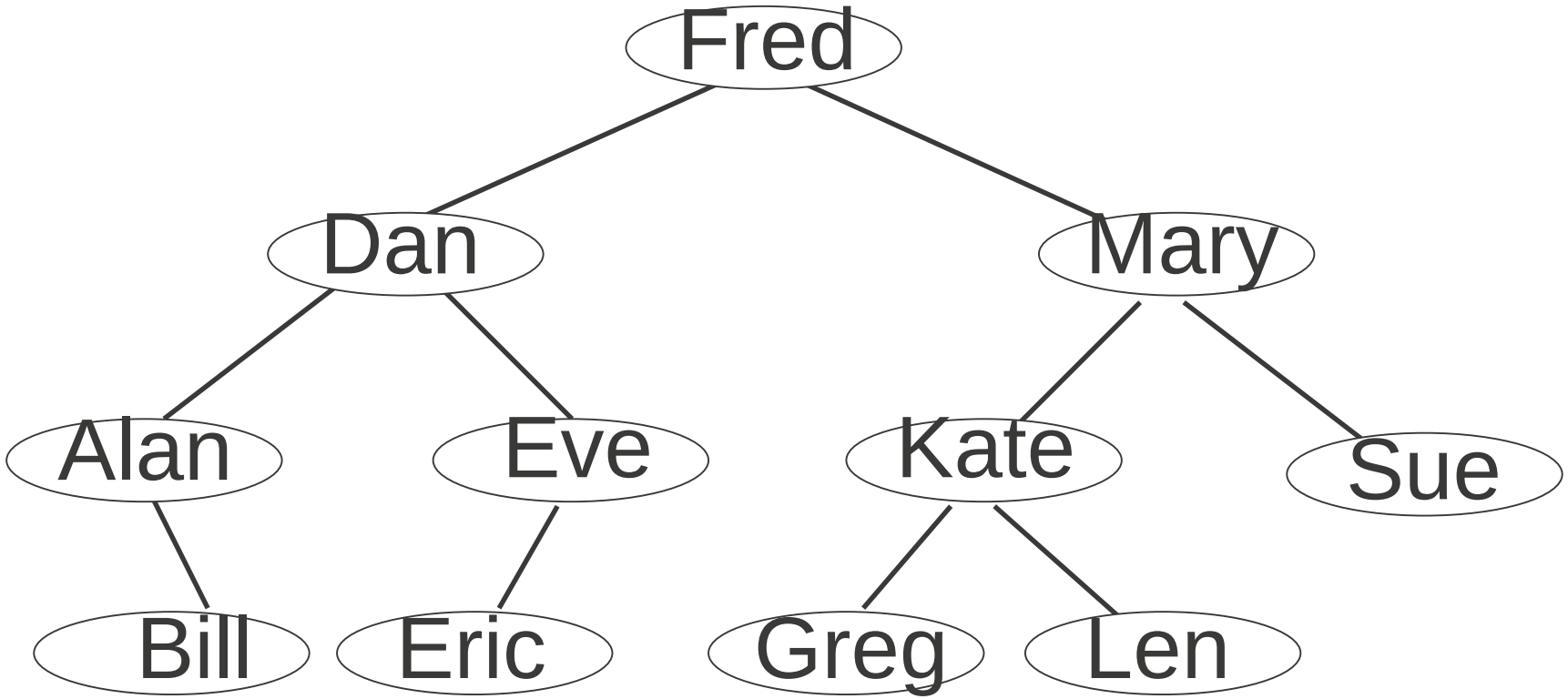
- **Every node entry has a key**
- **All keys in the left subtree of a node are less than the key of the node**
- **All keys in the right subtree of a node are greater than the key of the node**

Again, all keys are unique

# Example 1: the key is an integer



# Example 1: the key is a string



MONASH University

# BST Search

# Motivation: Why should we use binary search trees?

- **Sorted list using arrays:**
  - Good for search -- O(log N) [binary search]
  - Bad for inserting and deleting – O(N) [shuffling]
- **Sorted list using linked nodes:**
  - Good for inserting and deleting – O(1) [modifying links] once found
  - Bad for searching – O(N) [linear search]
- **Binary Search Trees are good for searching AND for inserting and deleting**
  - The search is always binary (complexity will depend on the depth)
  - Modifying the links is constant (once you find the element)
- **What about Hash tables?**
  - Main advantage of BSTs: can be traversed in particular orders; so they are more versatile for a reasonable time cost

# Possible class for Binary Search Trees

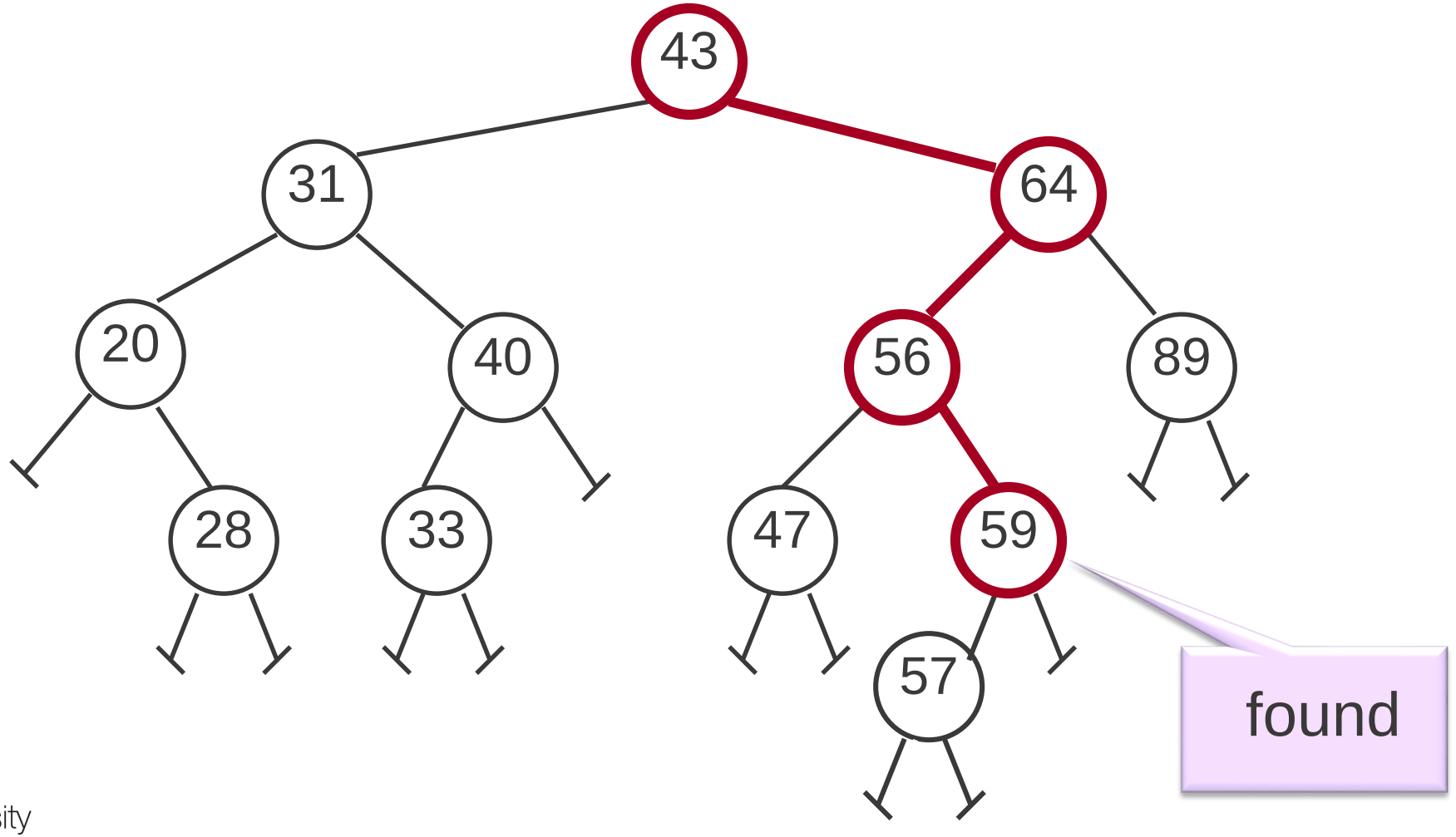
```
from typing import Callable, TypeVar, Generic
K = TypeVar('T')
I = TypeVar('T')

class BinarySearchTreeNode(Generic[K, I]):
    def __init__(self, key: K, item: I = None) -> None:
        self.key = key
        self.item = item
        self.left = None
        self.right = None

    def __str__(self):
        return "(" + str(self.key) + "," + str(self.item) + ")"

class BinarySearchTree(Generic[K, I]):
    def __init__(self) -> None:
        self.root = None

    def is_empty(self) -> bool:
        return self.root is None
```

Separation between key and data is appropriate for any searchable data structure, not only trees

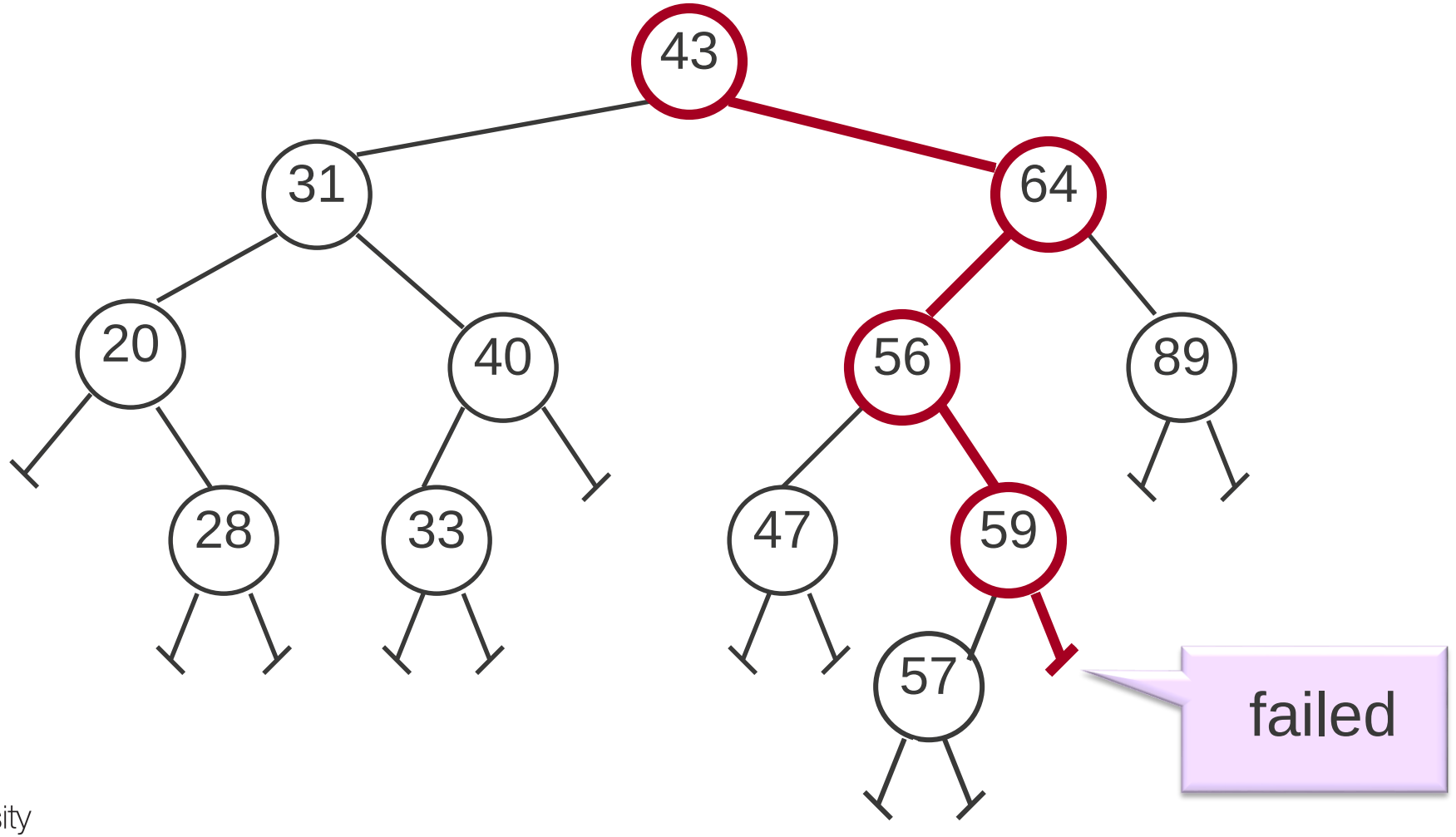
# Search algorithm

- If we reach an empty tree, we have not found it and return `False`
- Else, if target key is equal to the current node's key, we found it and return `True`
- Else, if target key is less than current node's key, search the left sub-tree
- Else, if target key is greater than current node's key, search the right sub-tree

# Search example: find 59



# Search example: find 61



# Search method

```
def find(self, key: K) -> bool:
    return self.find_aux(self.root, key)


def find_aux(self, current: BinarySearchTreeNode[K, I], key: K) -> bool:
    if current is None: # base case: empty
        return False
    elif key == current.key: # base case: found
        return True
    elif key < current.key:
        return self.find_aux(current.left, key)
    else:#key > current.key
        return self.find_aux(current.right, key)
```

This is the definition of __contains__

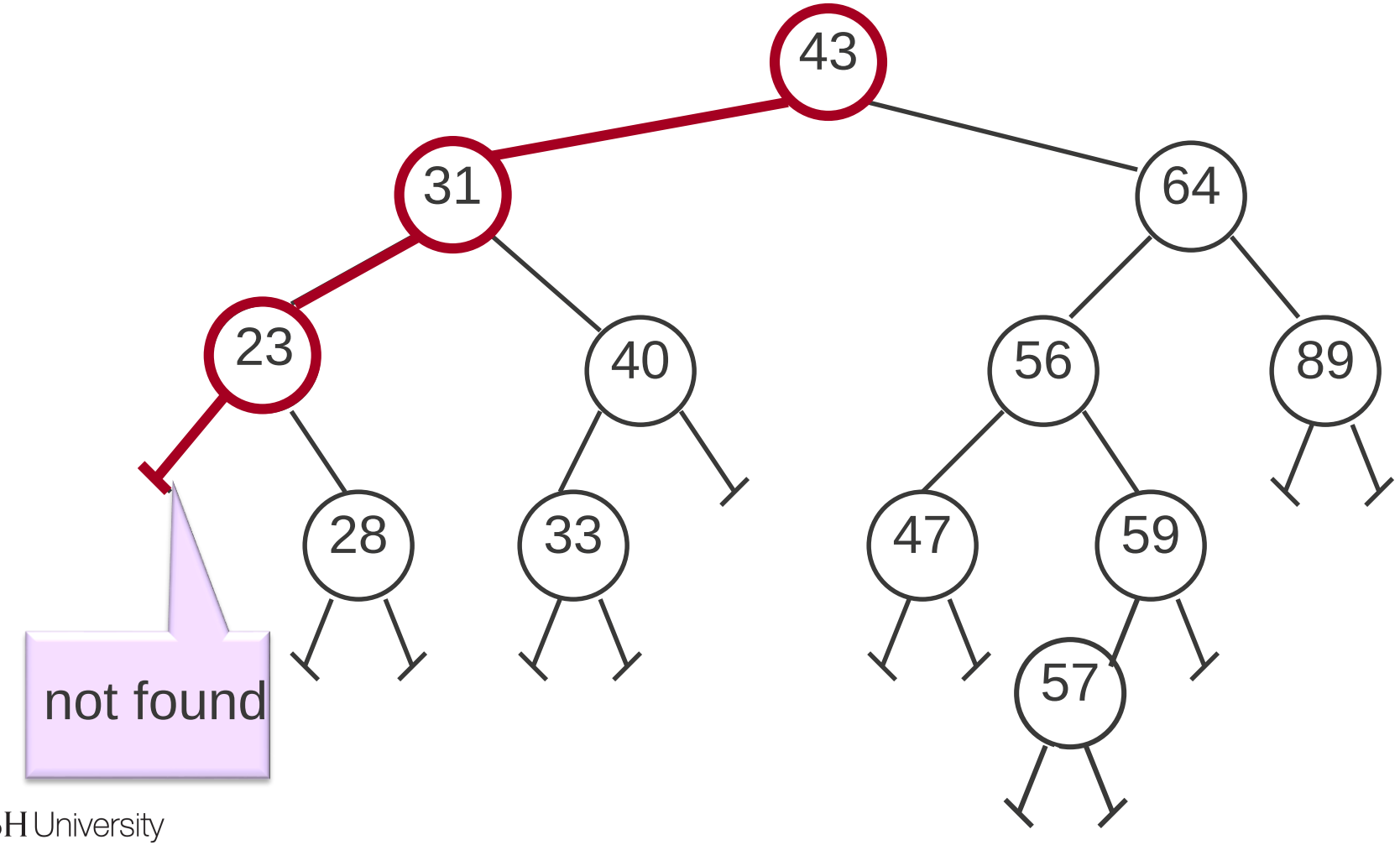
# Search method

```
def __contains__(self, key: K) -> bool:
    return self.find_aux(self.root, key)


def find_aux(self, current: BinarySearchTreeNode[K, I], key: K) -> bool:
    if current is None: # base case: empty
        return False
    elif key == current.key: # base case: found
        return True
    elif key < current.key:
        return self.find_aux(current.left, key)
    else:#key > current.key
        return self.find_aux(current.right, key)
```

This is the definition of __contains__

# Search example: find 22

You avoid all checks for ==



# Complexity

- **Best case: O(1)*Comp== when the element is at the root**
  - Comp== is the complexity of ==
- **Worst case: O(Depth)*(Comp== + Comp<), where Depth is the tree depth**
  - Note that the depth varies depending on how balanced the tree is

```
def find(self, key: K) -> bool:
    return self.find_aux(self.root, key)

def find_aux(self, current: BinarySearchTreeNode[K, I], key: K) -> bool:
    if current is None: # base case: empty
        return False
    elif key == current.key: # base case: found
        return True
    elif key < current.key:
        return self.find_aux(current.left, key)
    else:#key > current.key
        return self.find_aux(current.right, key)
```

# Balanced/Unbalanced tree (cont)

- **The depth of a balanced tree with N nodes is log N**
- **Thus, in a balanced binary search tree, the Big O time complexity of search is O(log N)*(Comp== + Comp<)**
- **In the extreme case the unbalanced tree is equivalent to a list (depth N-1)**
- **Thus, the time complexity becomes O(N)*(Comp==+Comp<) when the tree is unbalanced**
- **This can be solved by ensuring during insertion (and deletion) that the tree remains balanced:**
  - Many trees, like avl trees, red-black trees, 2-3-4 trees, etc, are designed for this

# What about `__getitem__`?

- **Recall: returns the item associated to a key. If it is not there raises KeyError.**

```
def __getitem__(self, key: K) -> T:
    return self.getitem_aux(self.root, key)


def getitem_aux(self, current: BinarySearchTreeNode[K, I], key: K) -> I:
    if current is None: # base case: empty
        raise KeyError("Key not found")
    elif key == current.key: # base case: found
        return current.item
    elif key < current.key:
        return self.getitem_aux(current.left, key)
    else:#key > current.key
        return self.getitem_aux(current.right, key)
```

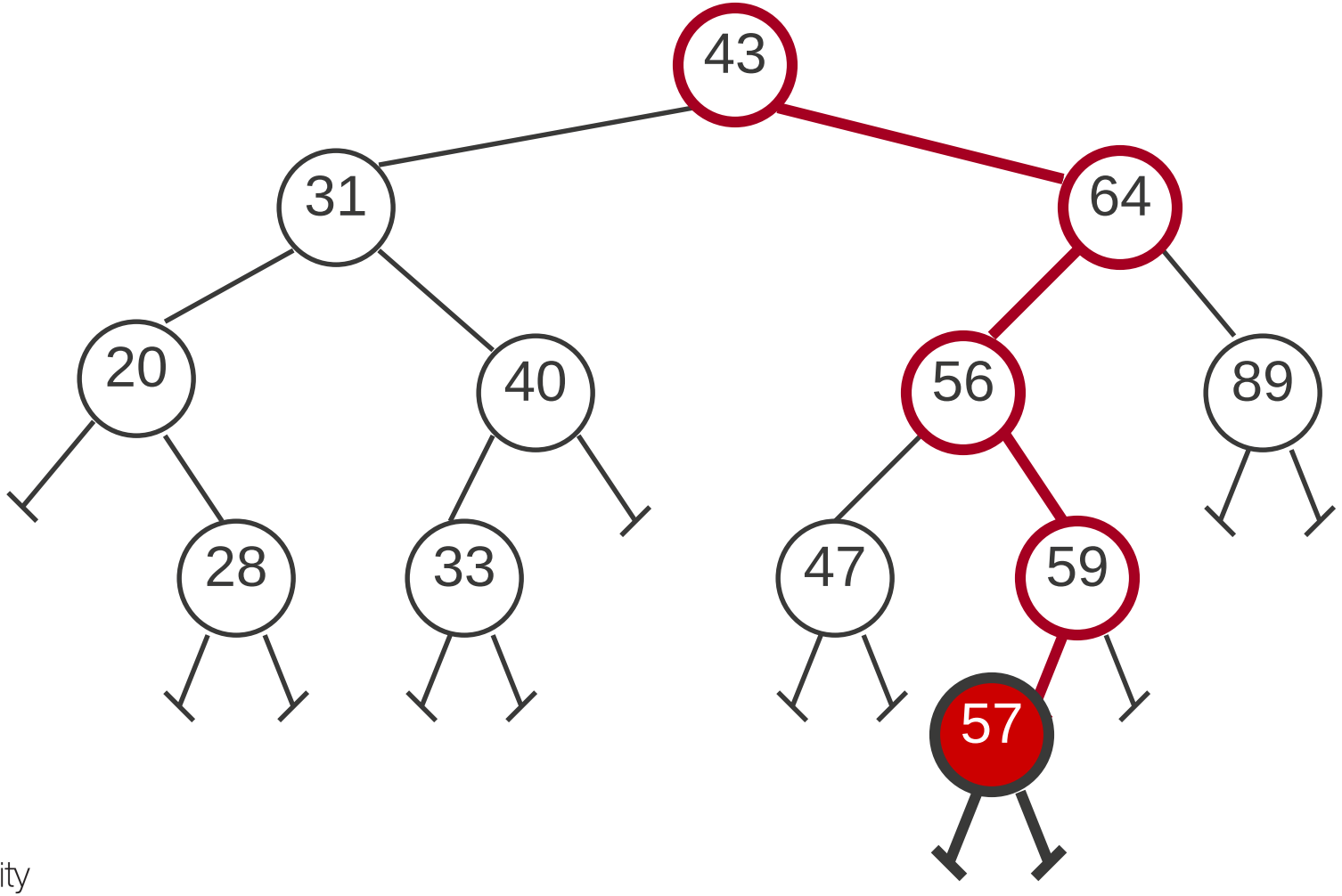
MONASH University

# BST Insertion

# Insert algorithm (insert at leaf)

- **We are given a key and associated item to insert**
- **We do not care whether the tree becomes unbalanced or not…**
- **As usual, it can be thought of a find+insert**
- **We first try to find the given key**
  - If we find it, we raise an exception:
    - No duplicates are allowed in binary search trees
    - We would have updated (if it was `__setitem__`)
  - Otherwise, we have reached a `None` link, its parent should be the parent of our new node
- **Then, we attach the new node as a leaf :**
  - Create a new node for the key and item with `None` links
  - Attach it to the parent node instead of its `None` link

# Insert example: insert 57 (and some item)

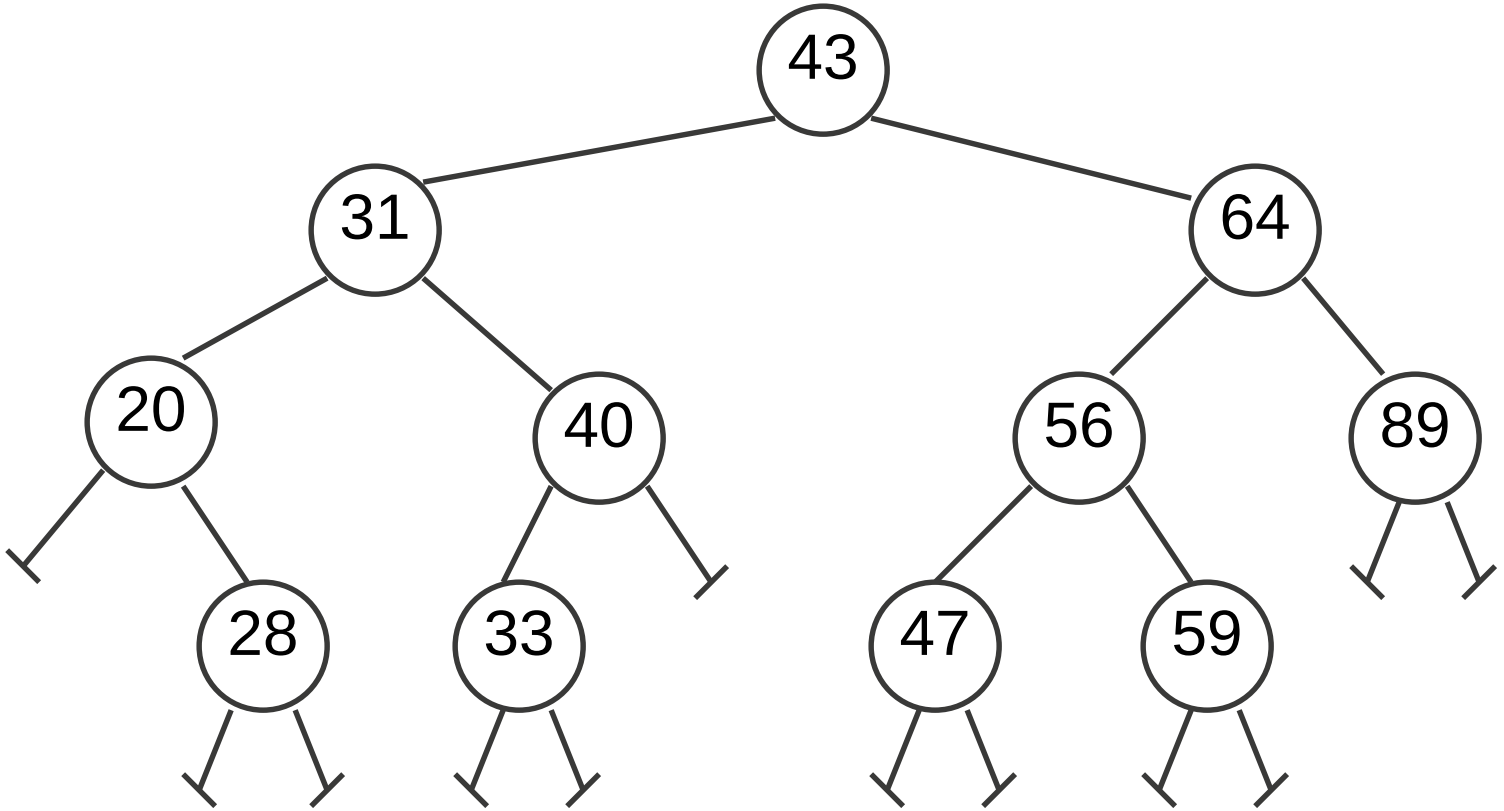


# Insert method

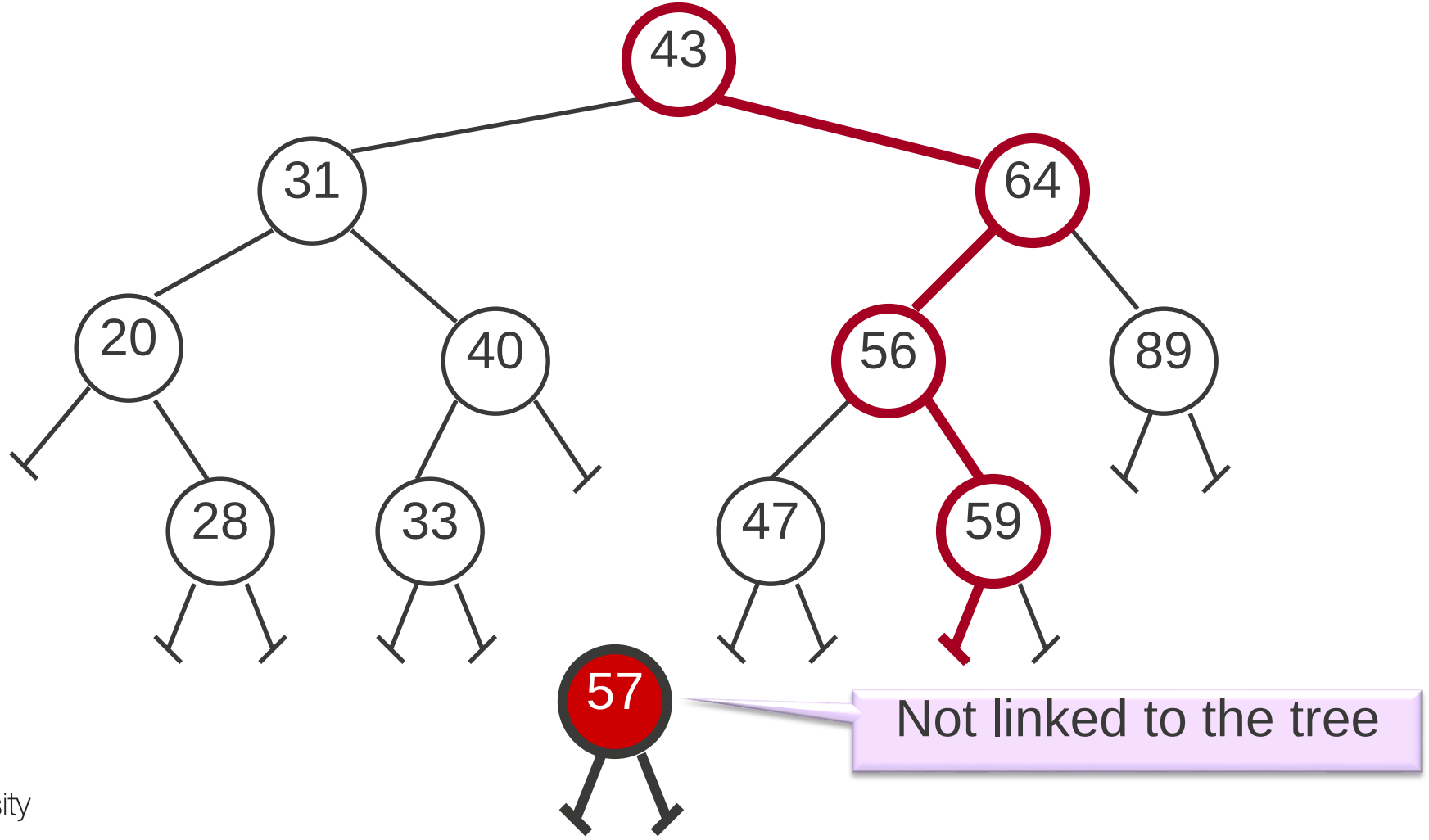
```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)


def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

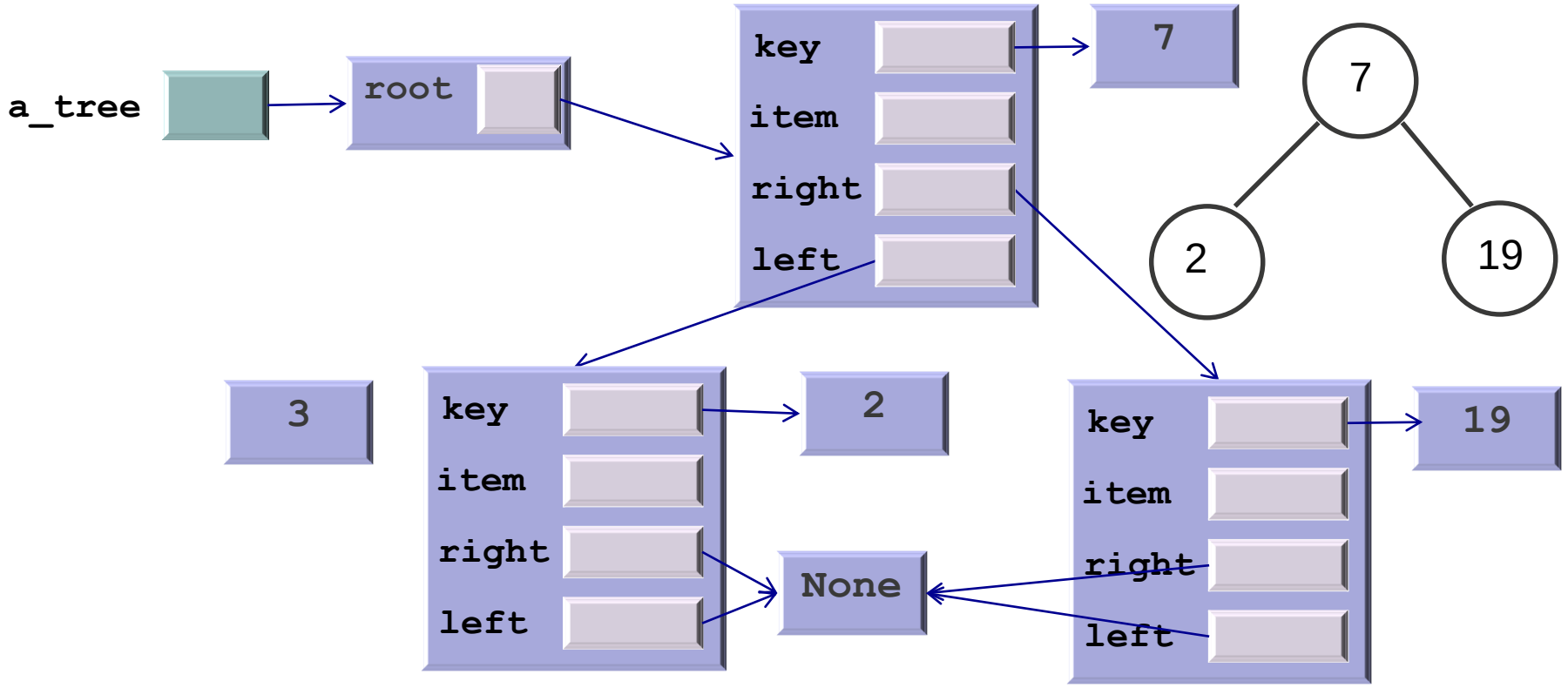
This does not work!

# Insert example: insert 57 (and some item)



# Insert example: insert 57 (and some item)

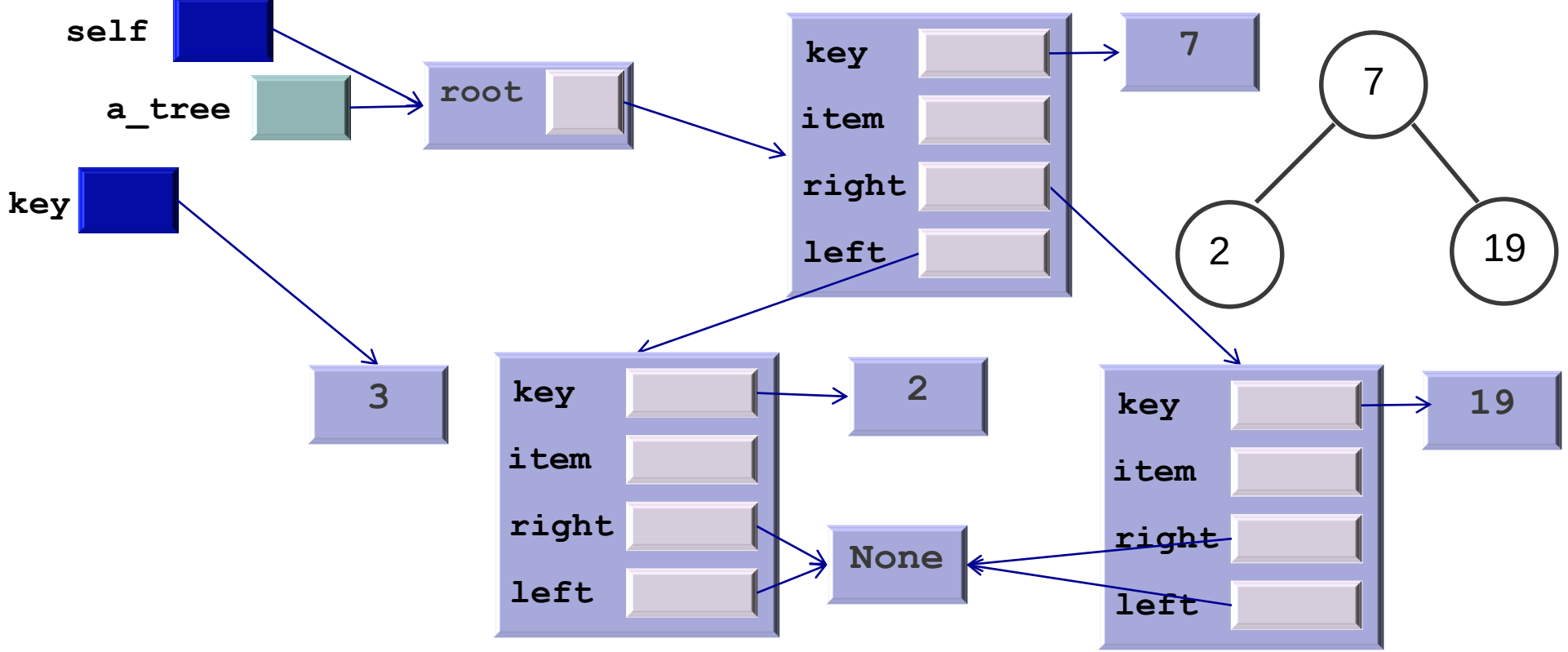





a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

Not representing the item properly…

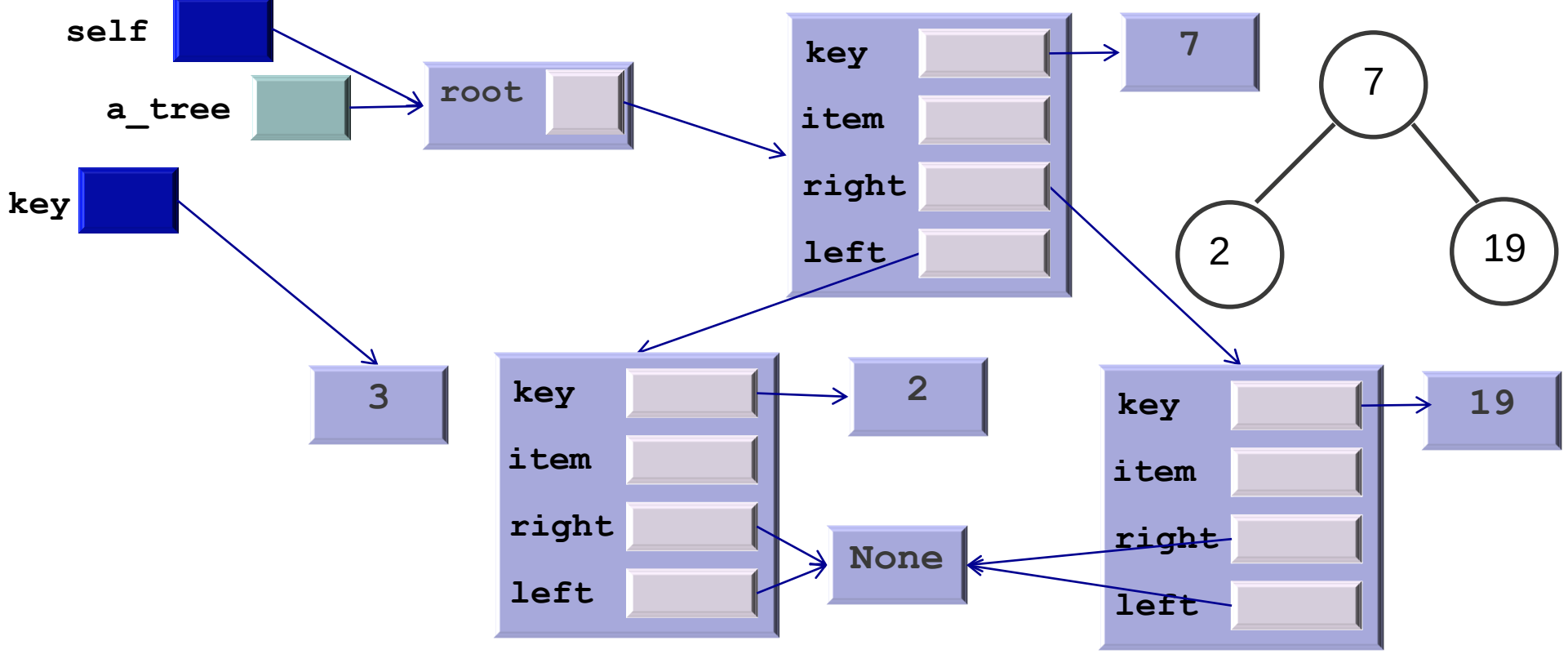



a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

Not representing the item properly…

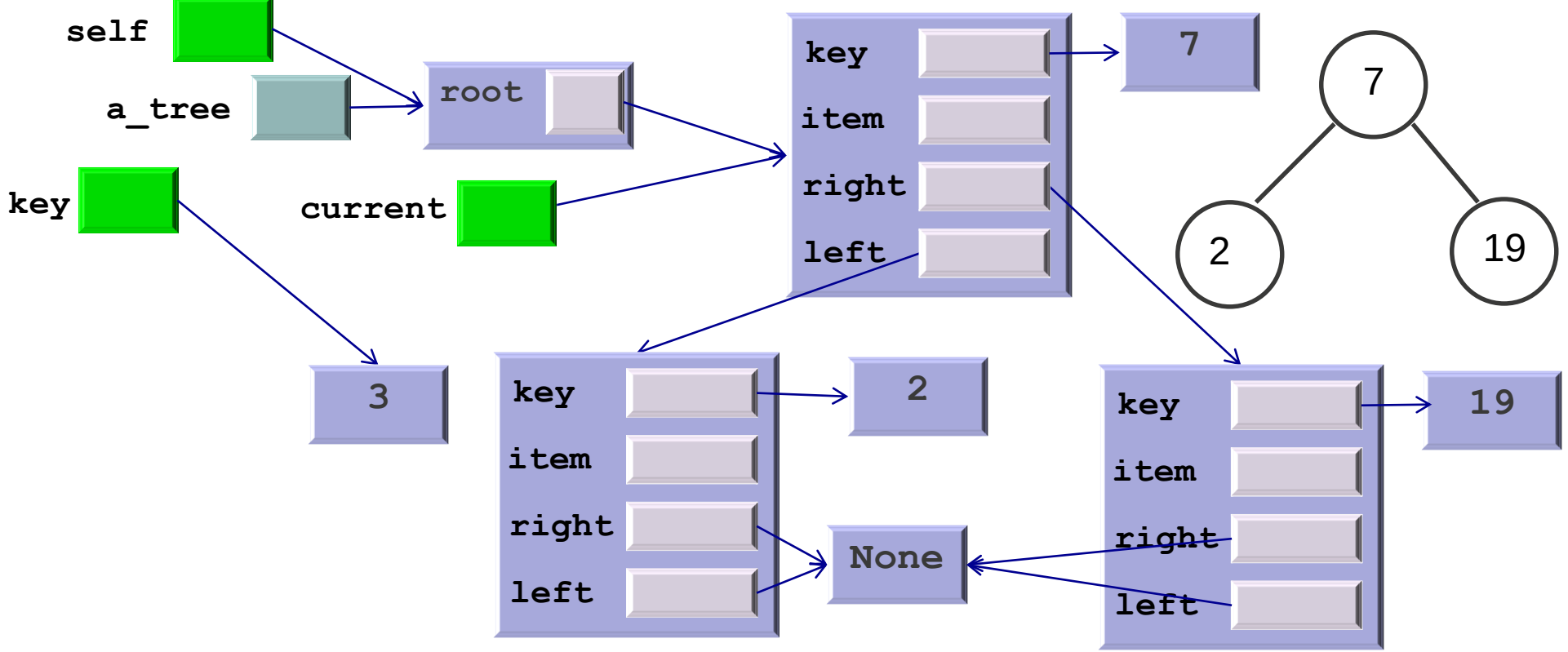
Changed the color to represent a different namespace




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

Not representing the item properly…

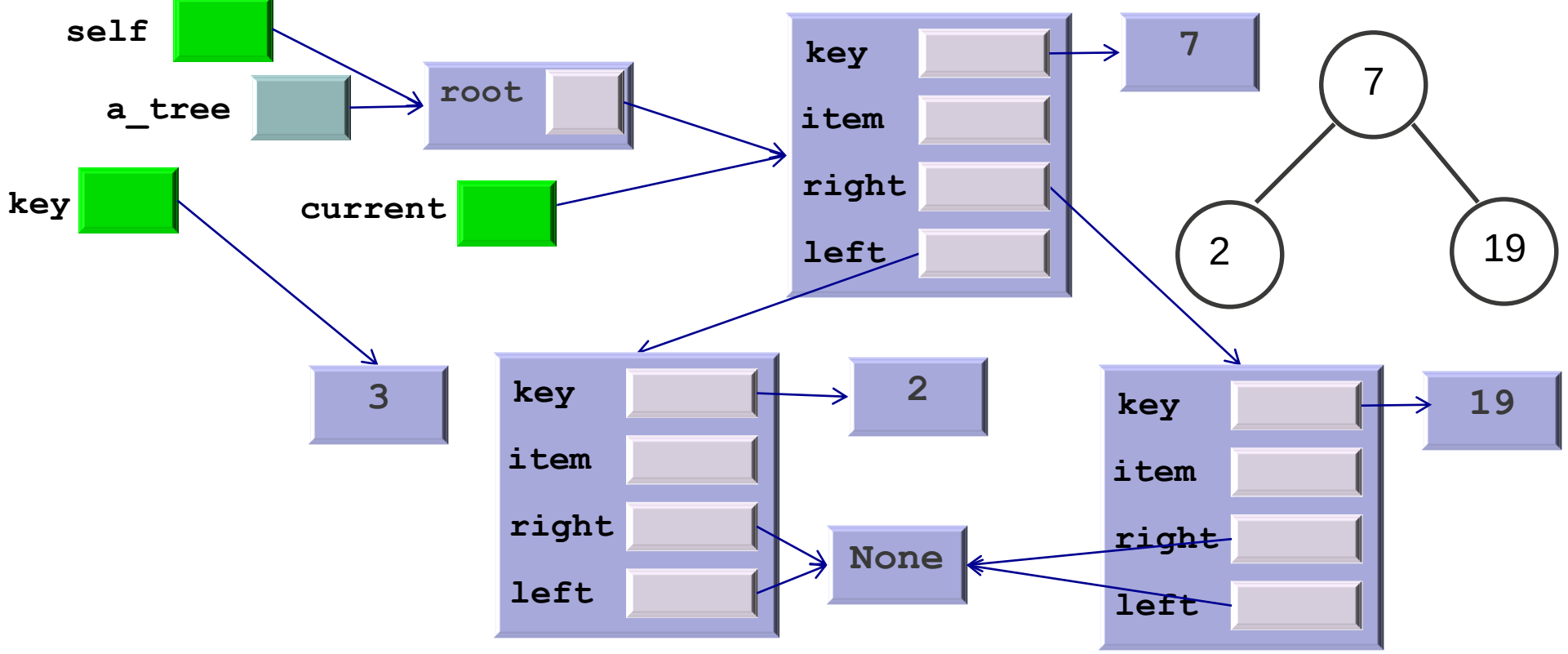



a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

Not representing the item properly…

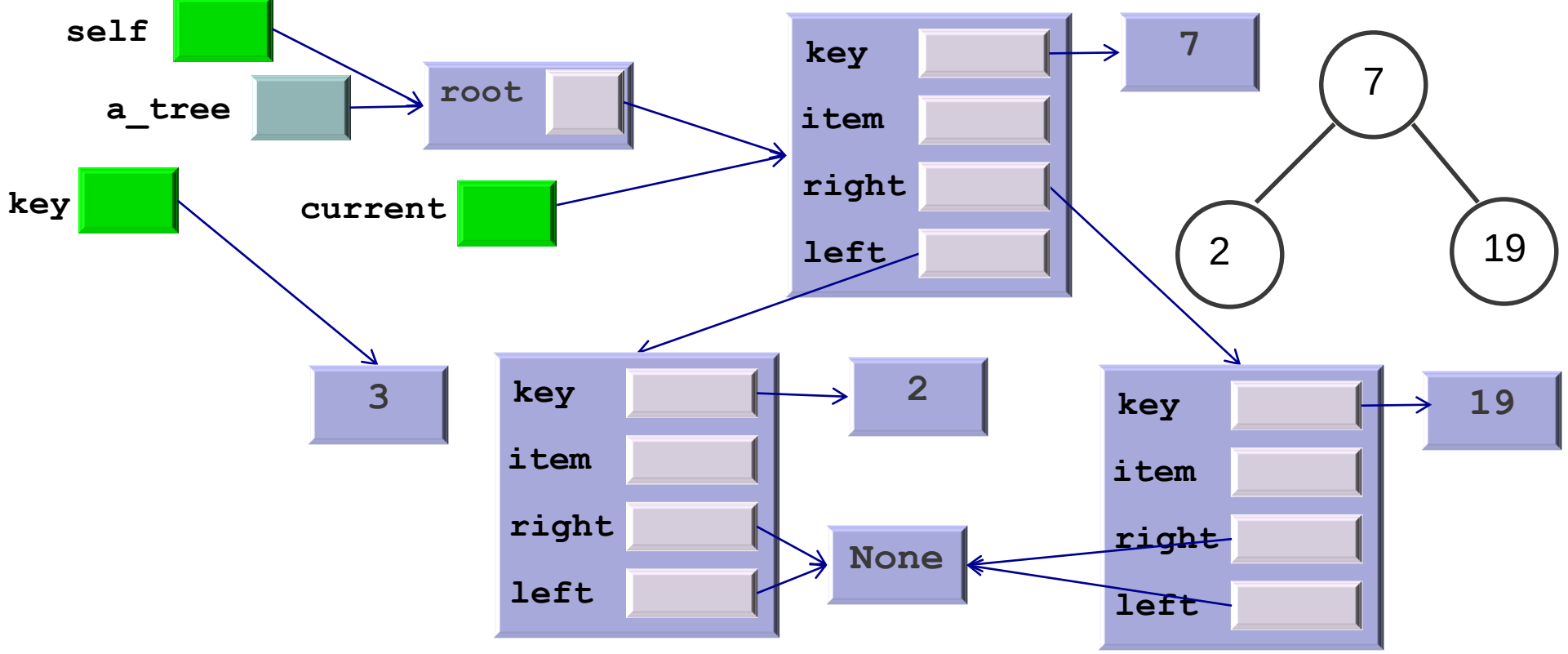
Changed again the color to represent a different namespace




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

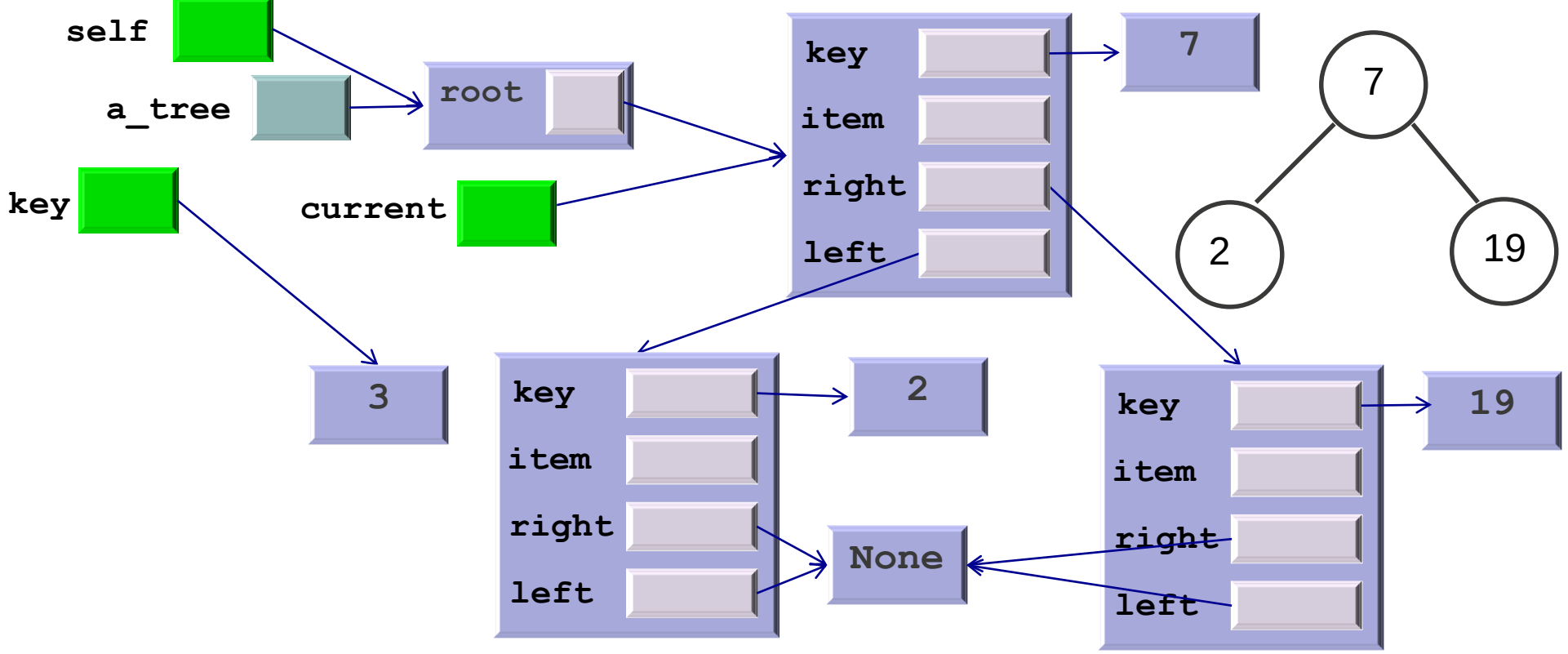
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

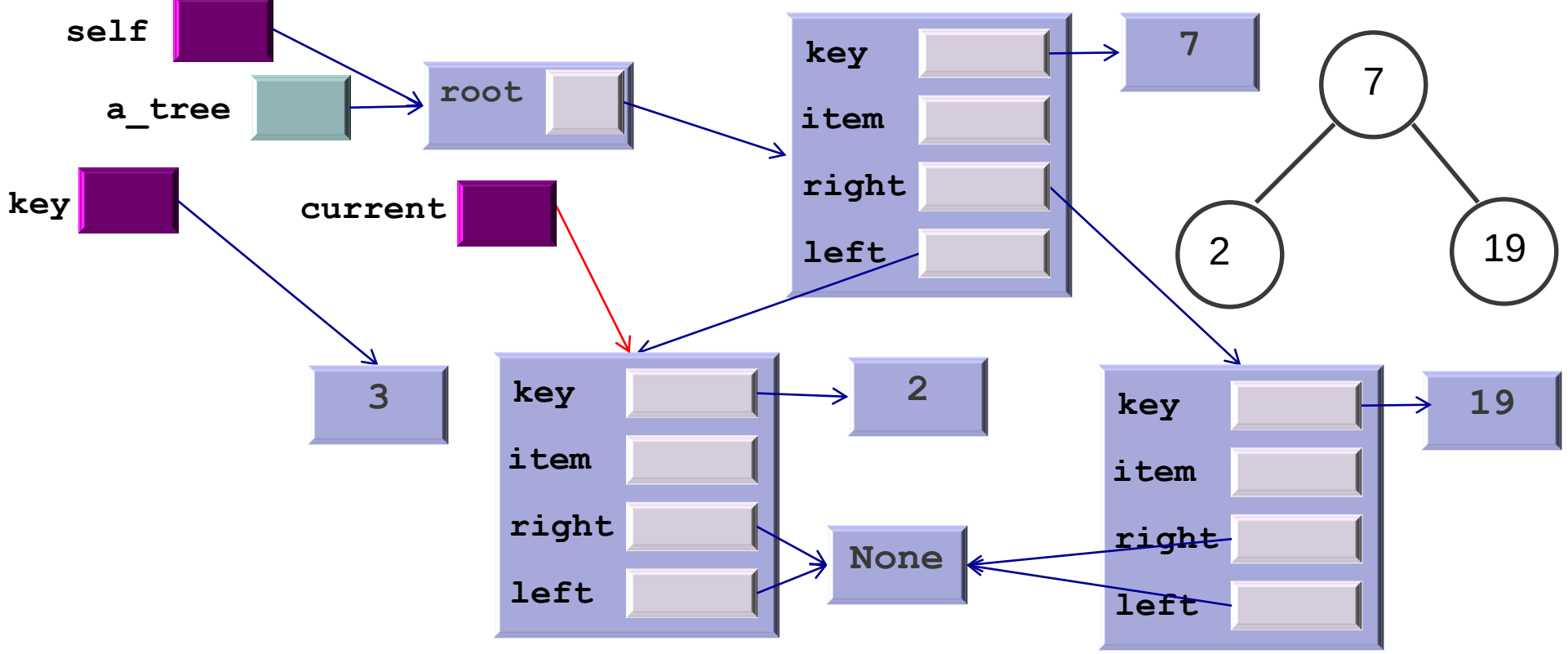
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

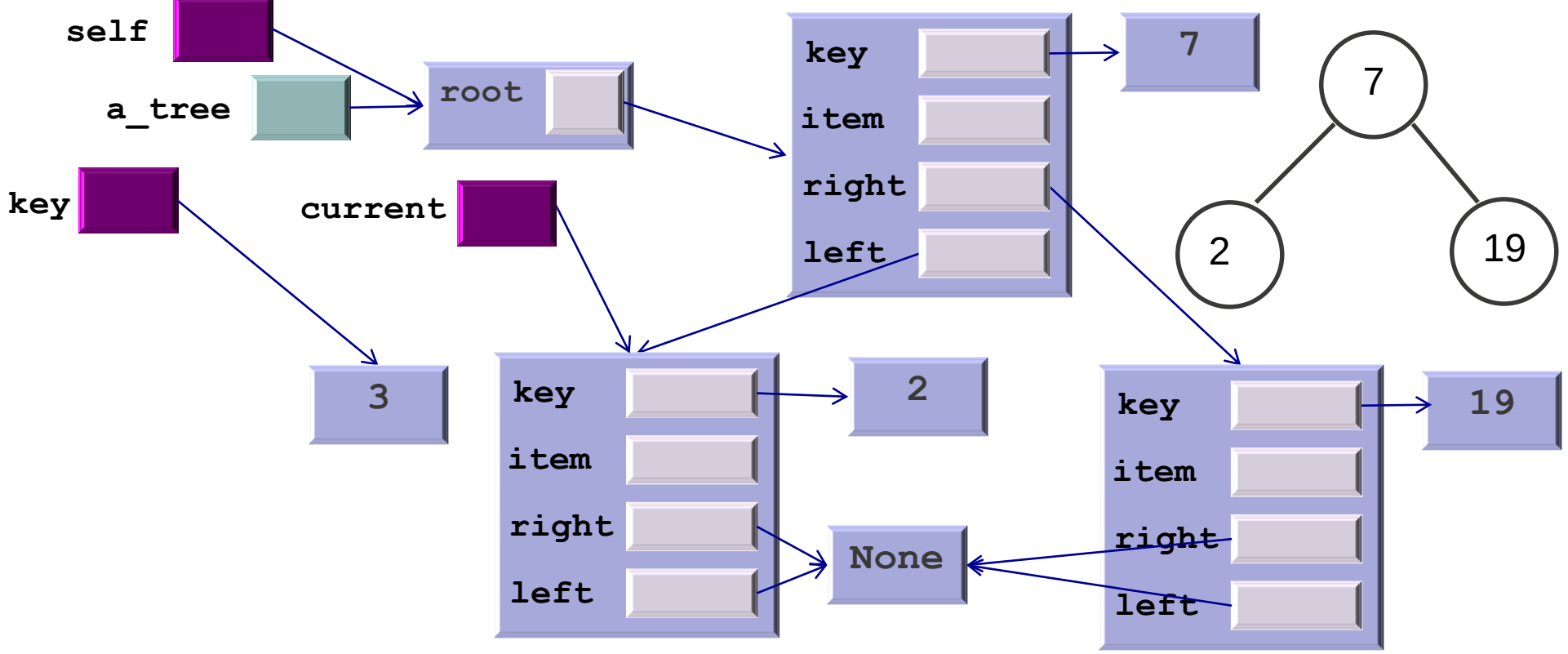
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

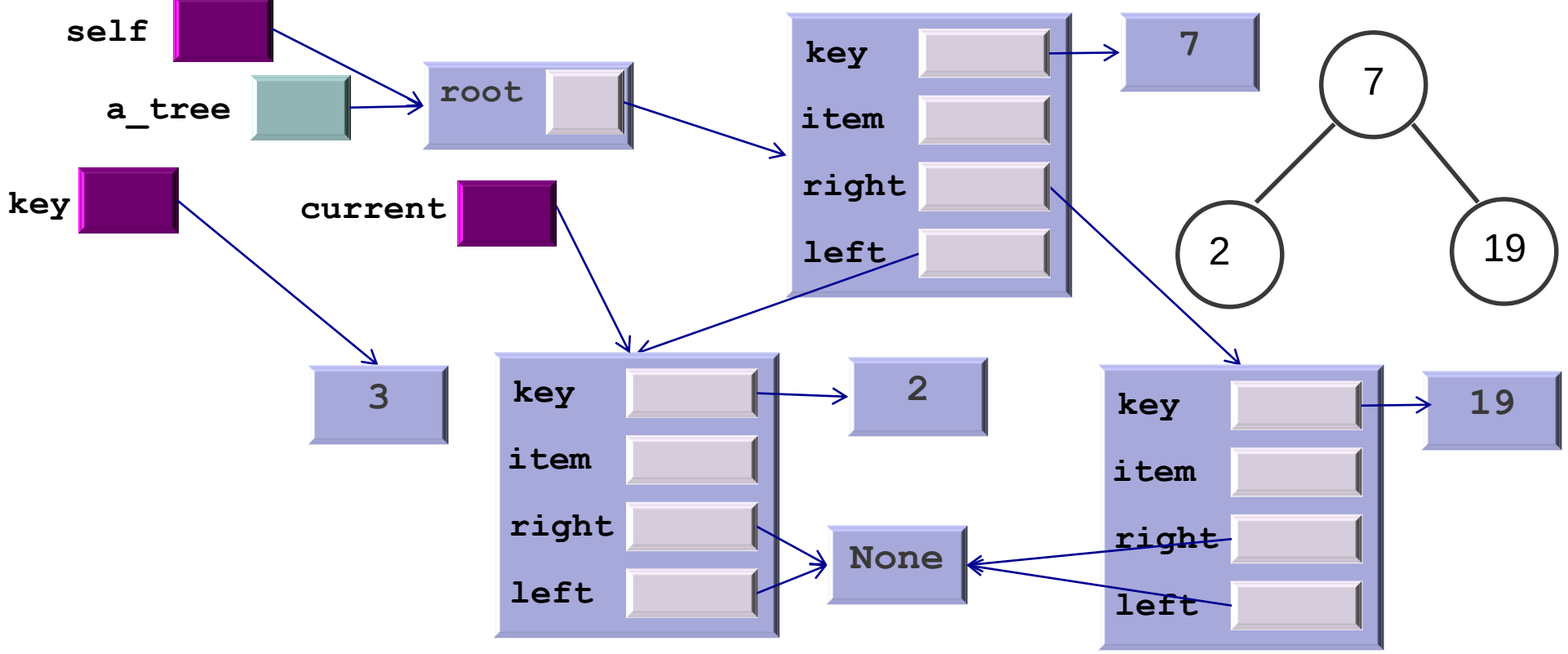
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

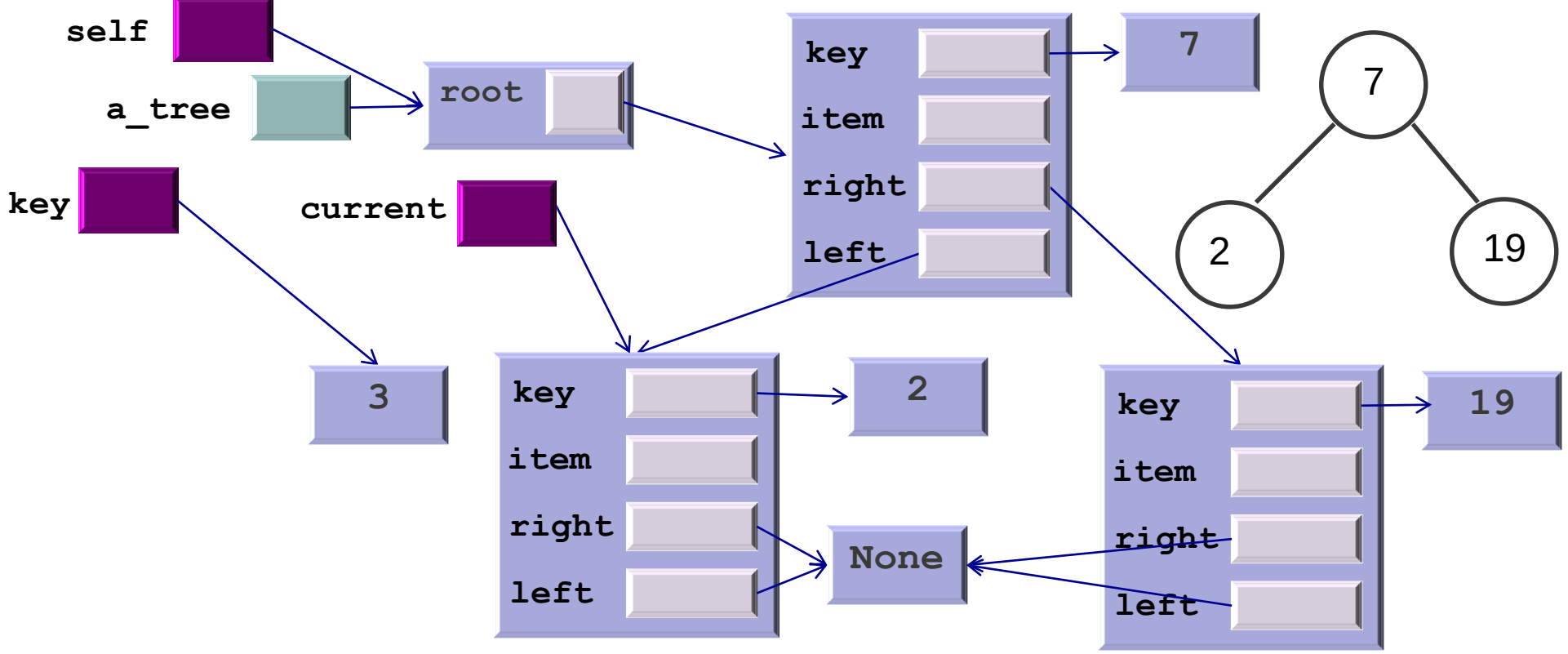
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

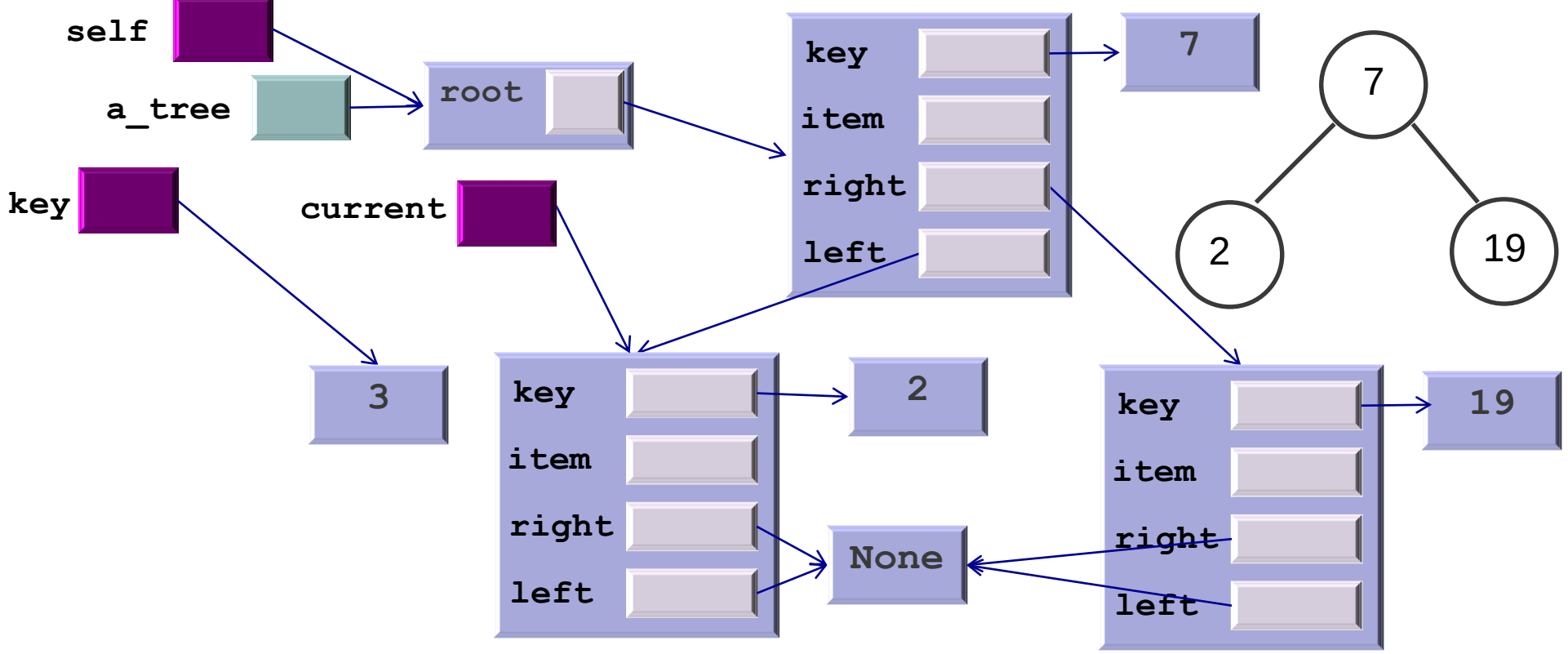
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

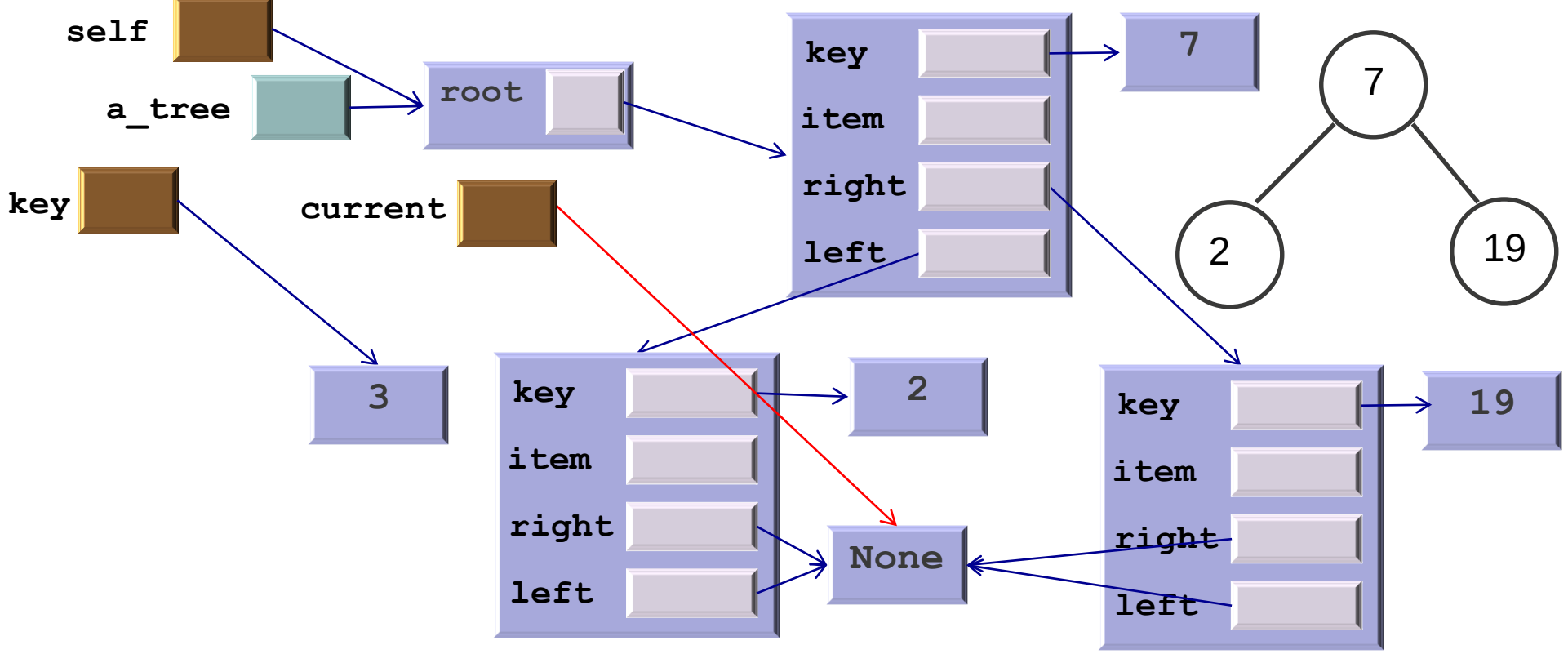
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

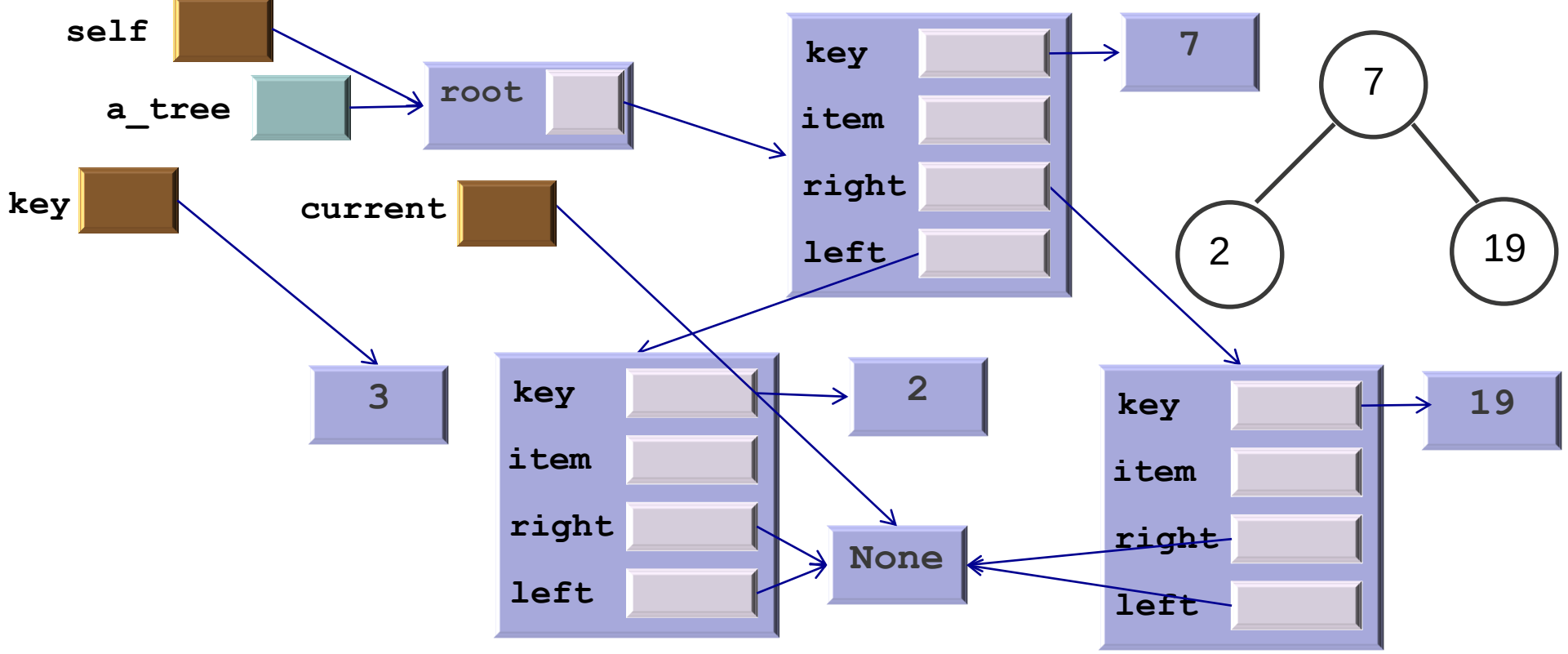
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

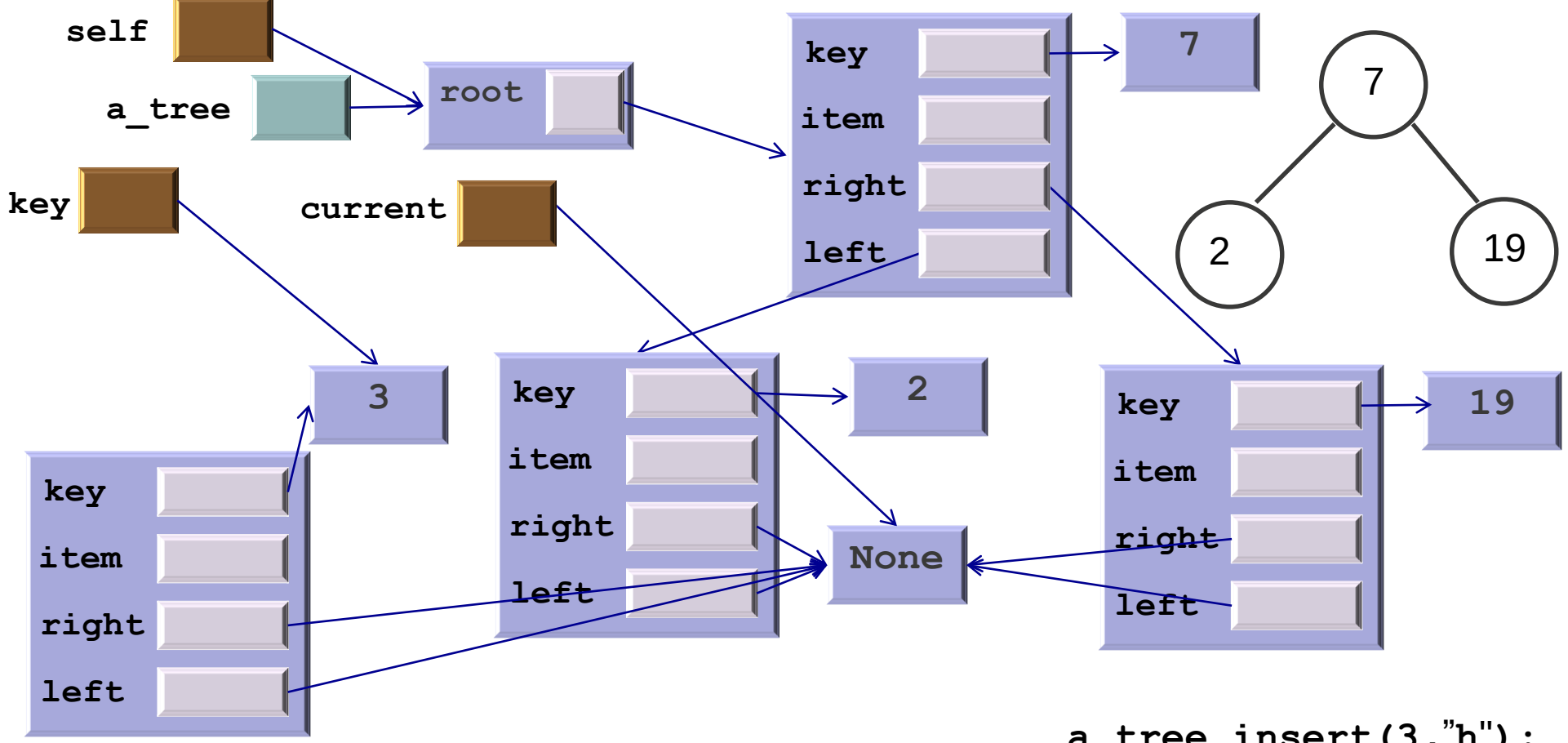
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

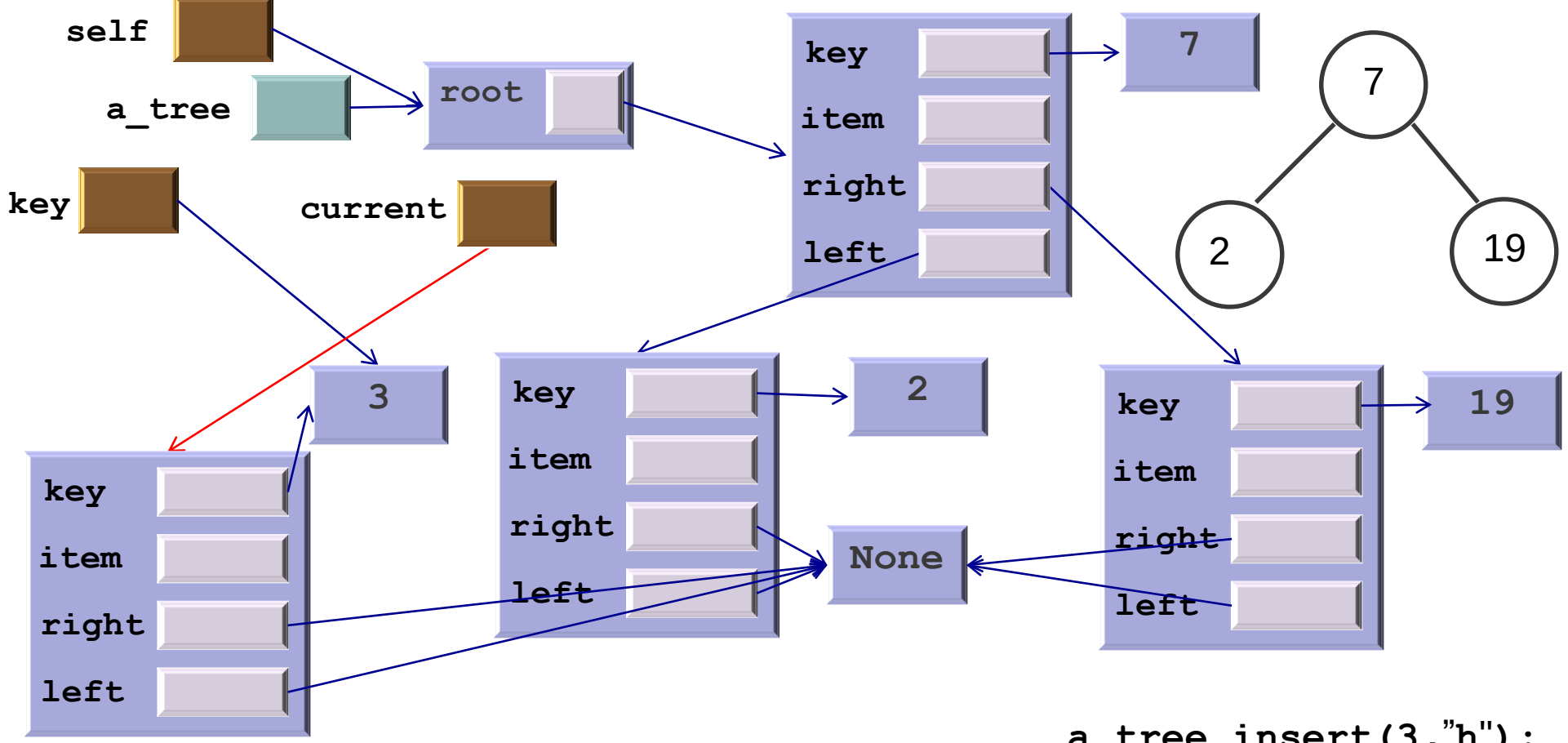
Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

Not representing the item properly…




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        self.insert_aux(current.left, key, item)
    elif key > current.key:
        self.insert_aux(current.right, key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
```

Not representing the item properly…

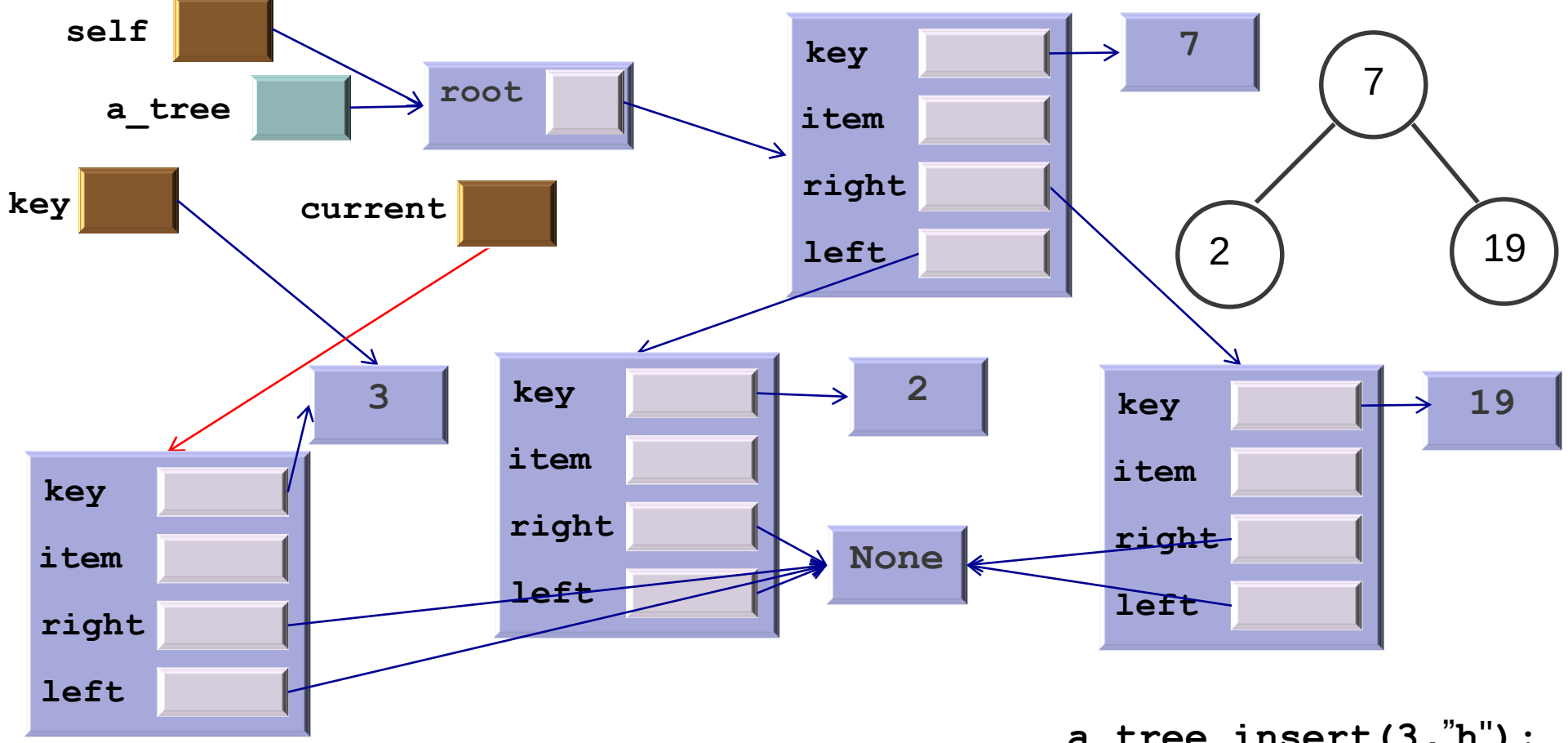
Finishes without modifying the tree!!

# Insert method

- **Main mistake:**
  - By the time `current is None`, the link is lost
  - We never linked the new node from one of the existing tree nodes
- **We could solve this by keeping the parent node around (if not None), as we did in linked lists with `previous`**
- **But since we are already in a recursive function, it is easier to do things "in the way back" (i.e., once the recursive call finishes)**
  - Always return the (possibly updated) current
  - Use this returned item to update the link (left, right or root) in the current node

# Insert method

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)


def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) ->
  None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

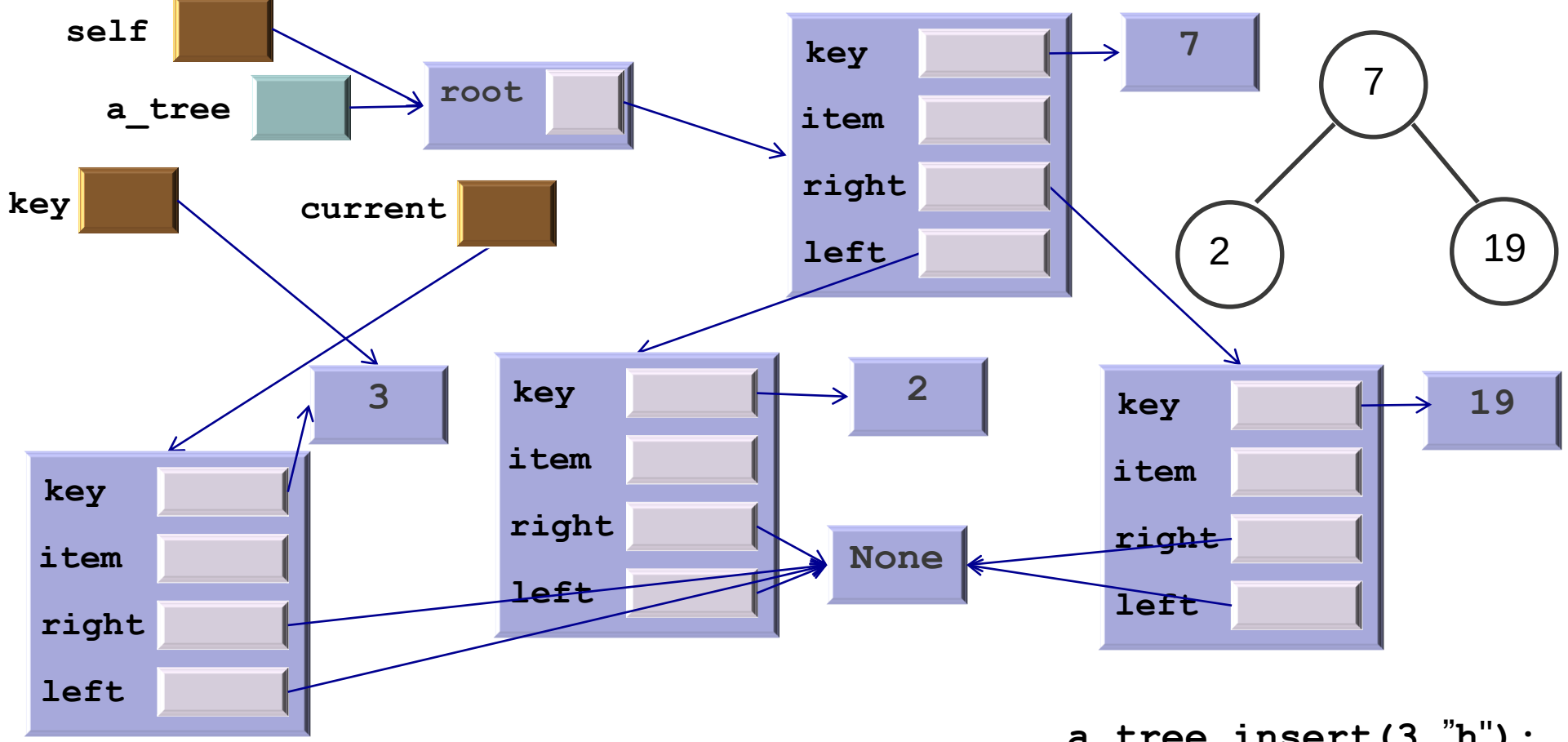



a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

We were here

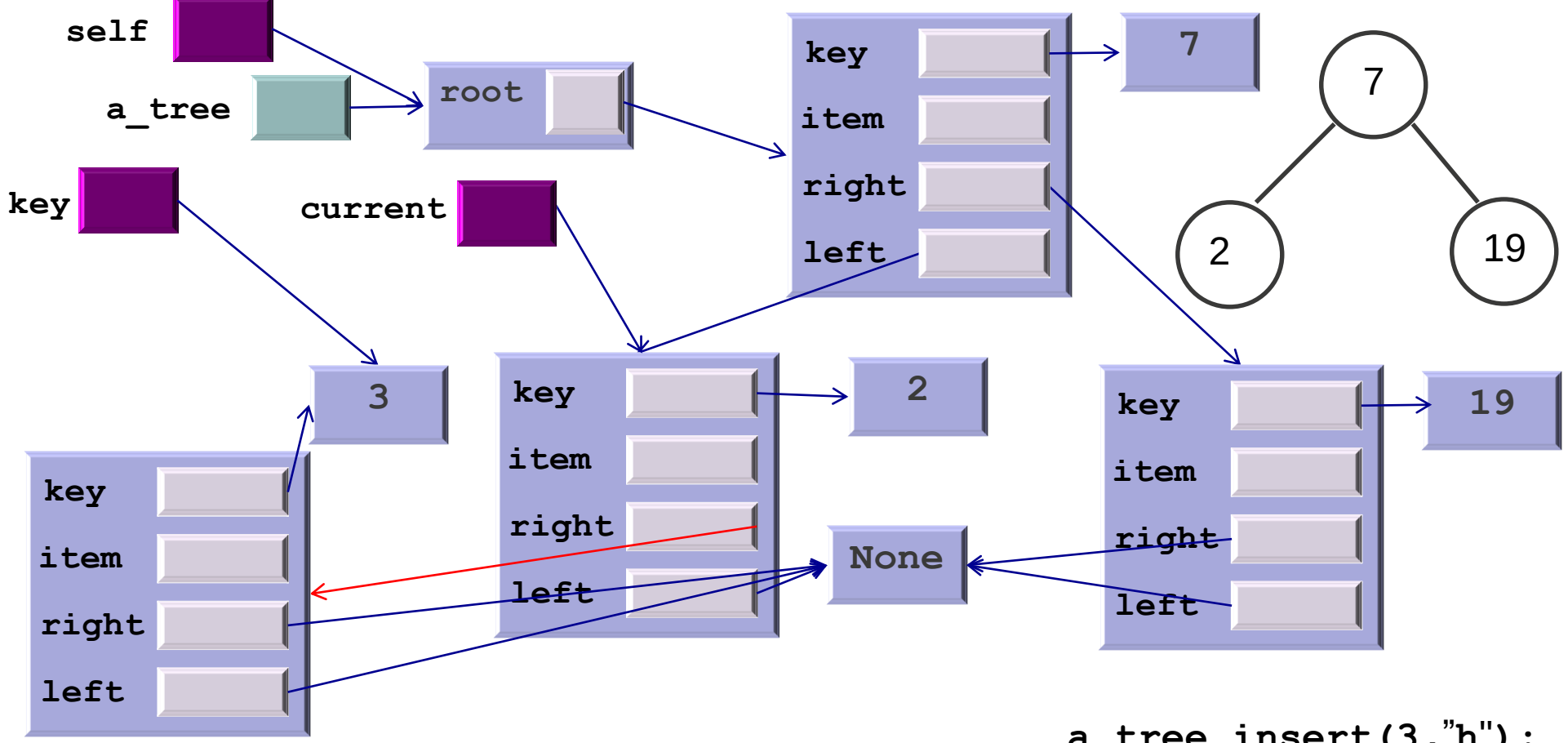
Let's see what happens in the way back




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

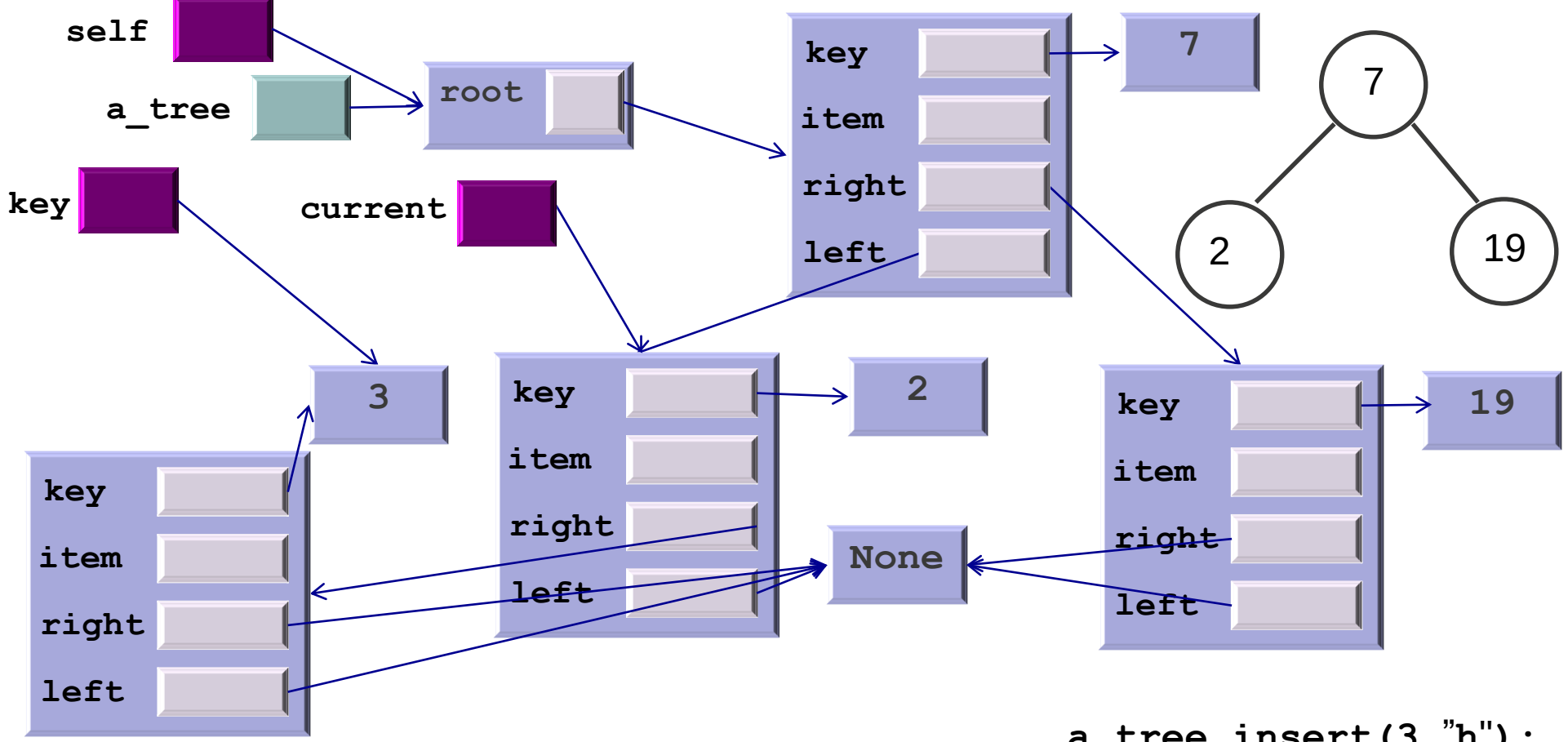
We were here




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

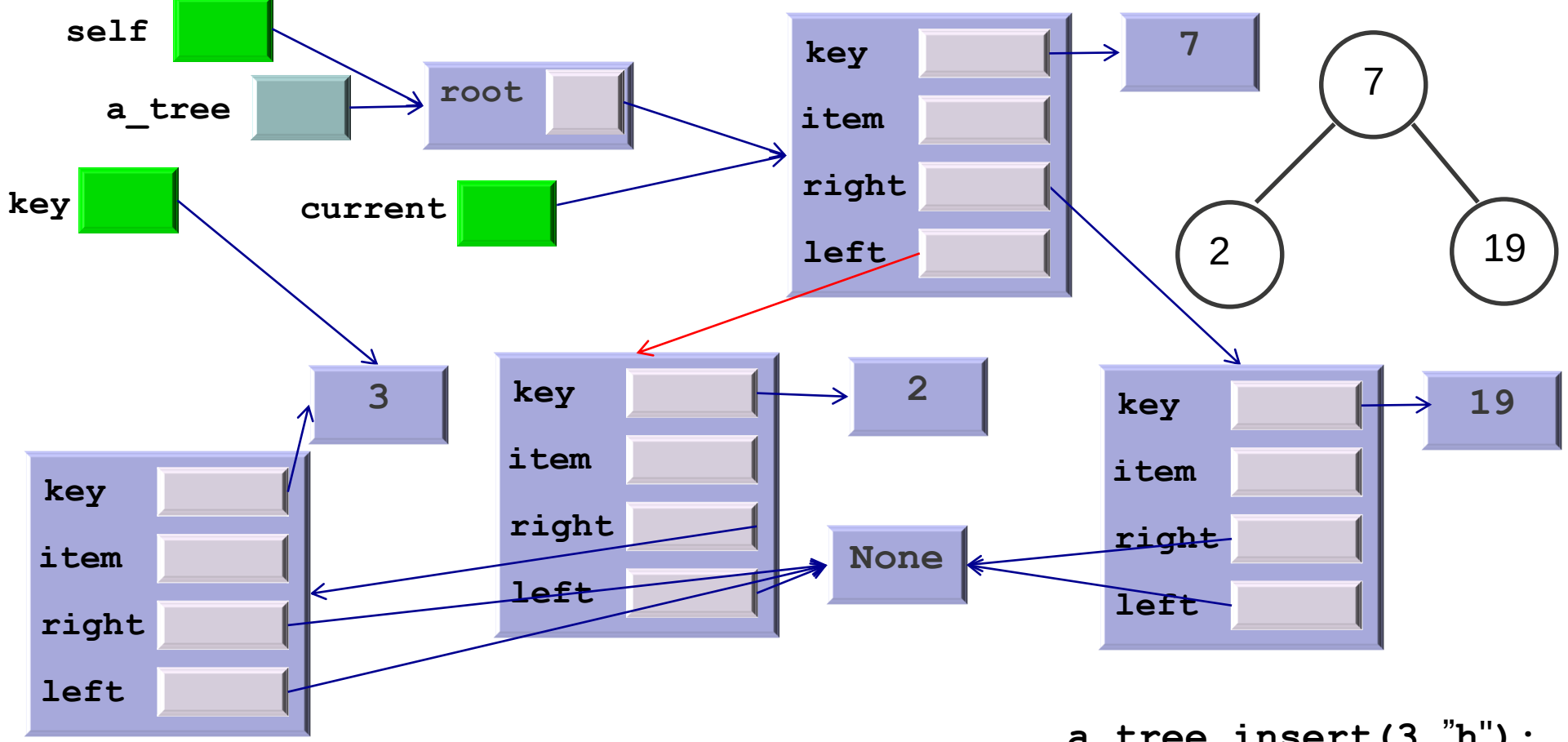
We were here




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

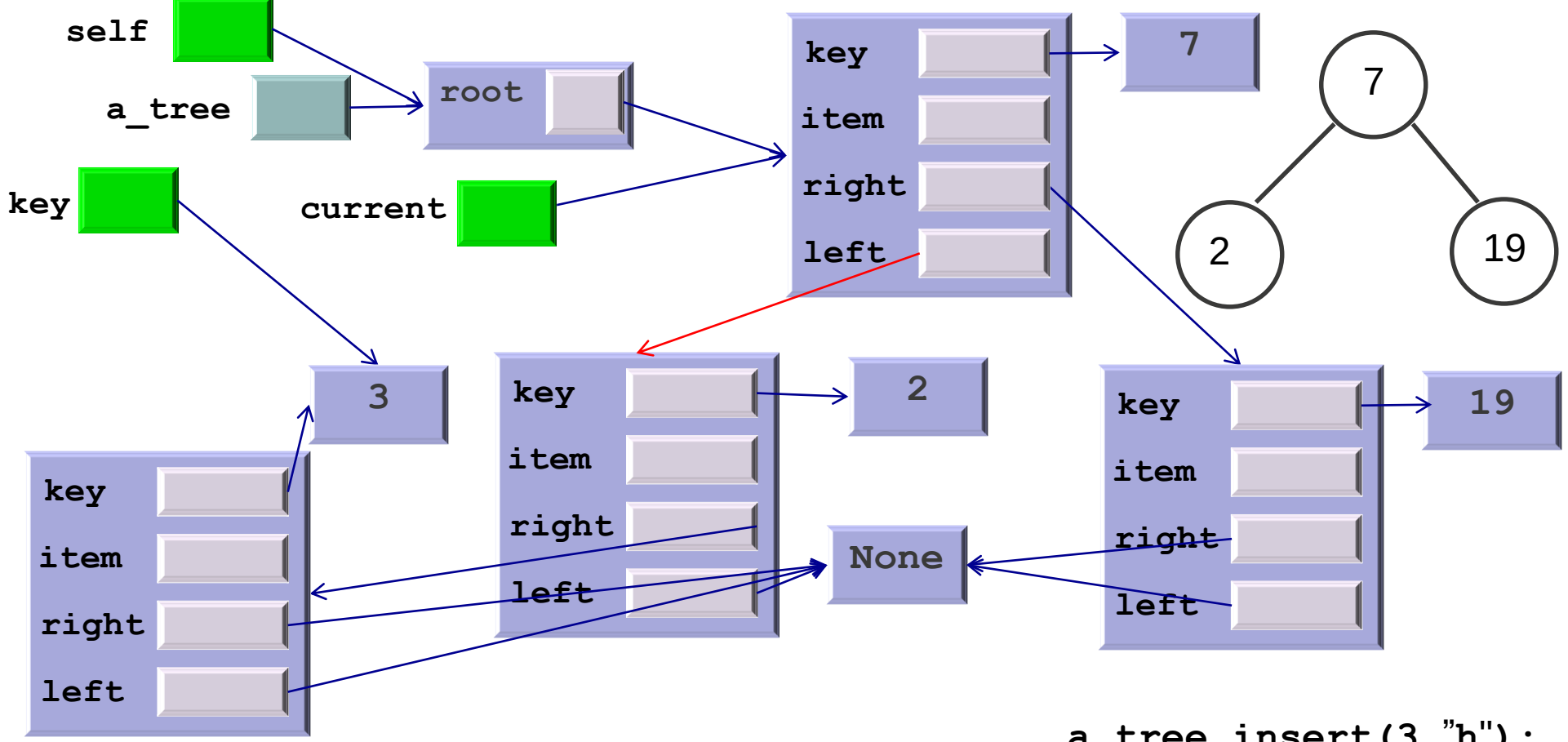
We were here




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

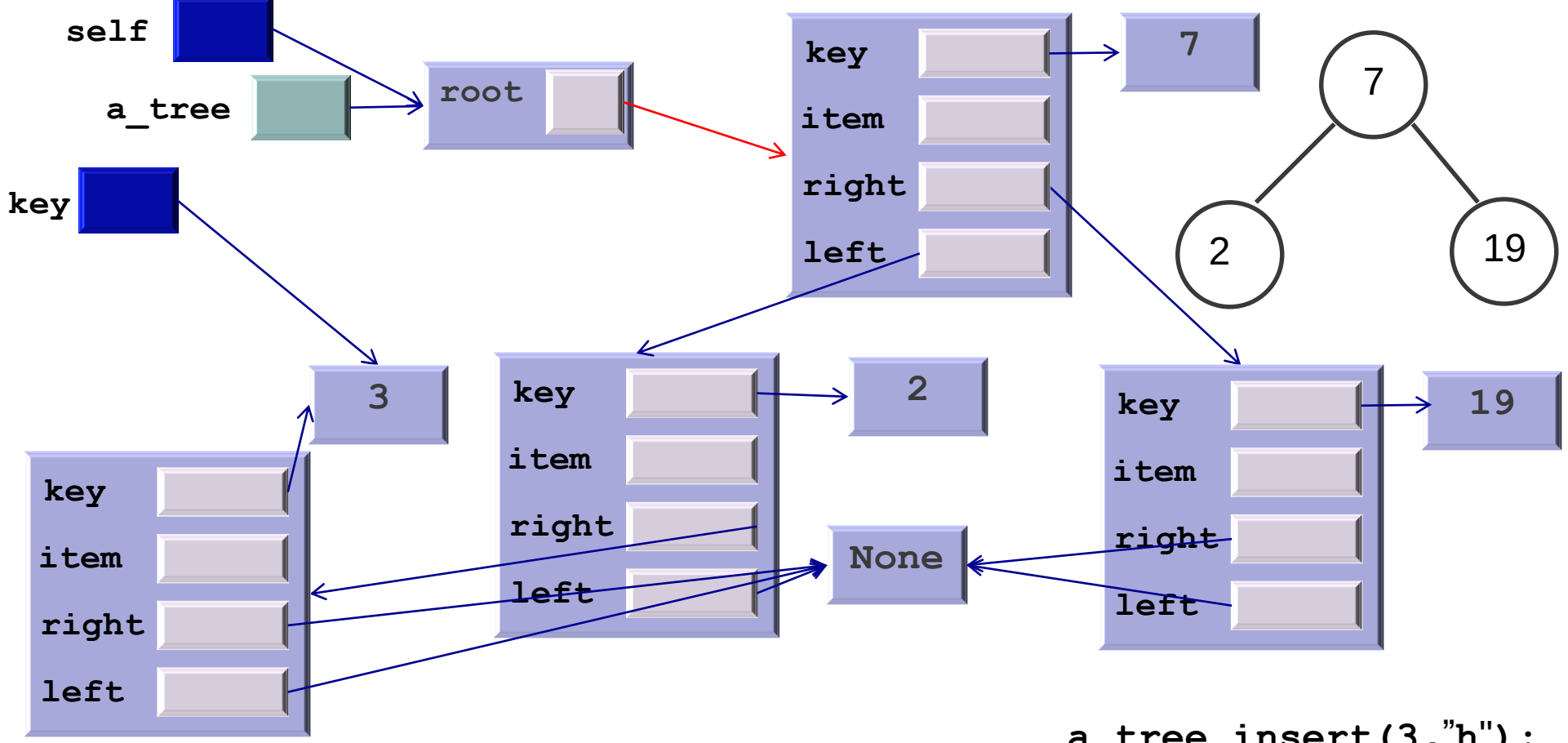
We were here




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

We were here




a_tree.insert(3,"h"):

```
def insert(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)
def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left,key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right,key, item)
    else: # key == current.key
        raise ValueError("Inserting duplicate item")
    return current
```

We were here

The new node is now properly connected

# Complexity of insert

- **Best case:**
  - Tempting to say: O(1)*(Comp< + Comp >) when the key is at the root (we raise an exception)
    - But errors are not considered for Big O: they are not supposed to happen!
  - Best case is O(1)*Comp< when the N nodes in the tree are all in the right sub-tree (or O(1)*Comp> when the N nodes are all in the left sub-tree)
  - If balanced, best=worst so O(log N)*(Comp< + Comp>)
- **Worst case: O(Depth)*(Comp<+Comp>), so if**
  - Balanced: O(log N)*(Comp< + Comp>)
  - Unbalanced: O(N)*(Comp< + Comp>)

# With a few changes, it is __setitem__

```
def __setitem__(self, key: K, item: I) -> None:
    self.root = self.insert_aux(self.root, key, item)


def insert_aux(self, current: BinarySearchTreeNode[K, I], key: K, item: I) -> None:
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key, item)
    elif key < current.key:
        current.left = self.insert_aux(current.left, key, item)
    elif key > current.key:
        current.right = self.insert_aux(current.right, key, item)
    else: # key == current.key
        current.item = item
    return current
```

# Correctness of insertion

- **Can we prove this algorithm is correct?**
- **Proving the correctness of algorithms is a cornerstone of computer science**
- **What are the invariants of the binary search tree class?**
  - the tree is binary,
  - keys in a left subtree are less than that of their root
  - keys in a right subtree are greater than that of their root
- **To prove the correctness of insert we have to prove that it maintains the invariants**
  - i.e., that if insert is applied to a binary search tree, it returns a binary search tree

# Correctness of insertion (cont)

- **We can use many different techniques to prove it**
- **For example, we could prove it by construction:**
  - We first prove that each iteration adds a single (BST) node
    - The algorithm only traverses one branch and the node addition is performed at the very end (base case)
  - Then that the addition is to an empty left/right child of current (current.left or current.right), which is a BST node
  - And then that if added to the left (right) its key is smaller (greater) than the parent

# Correctness of insertion (cont)

- **Or we can prove it by contradiction:**
  - We first assume that the tree is no longer a BST
  - This could happen either if we created a node with more than 2 subtrees
    - Impossible given the definition of the data structure
  - Or if we introduced a problem with the keys
    - Not with an old key (the algorithm does not modify them)
    - So the new key must be greater than its parent but appear in  the node in the left tree
      - You would then prove this is impossible given the code
    - The same reasoning applies to the opposite case (smaller in the right)

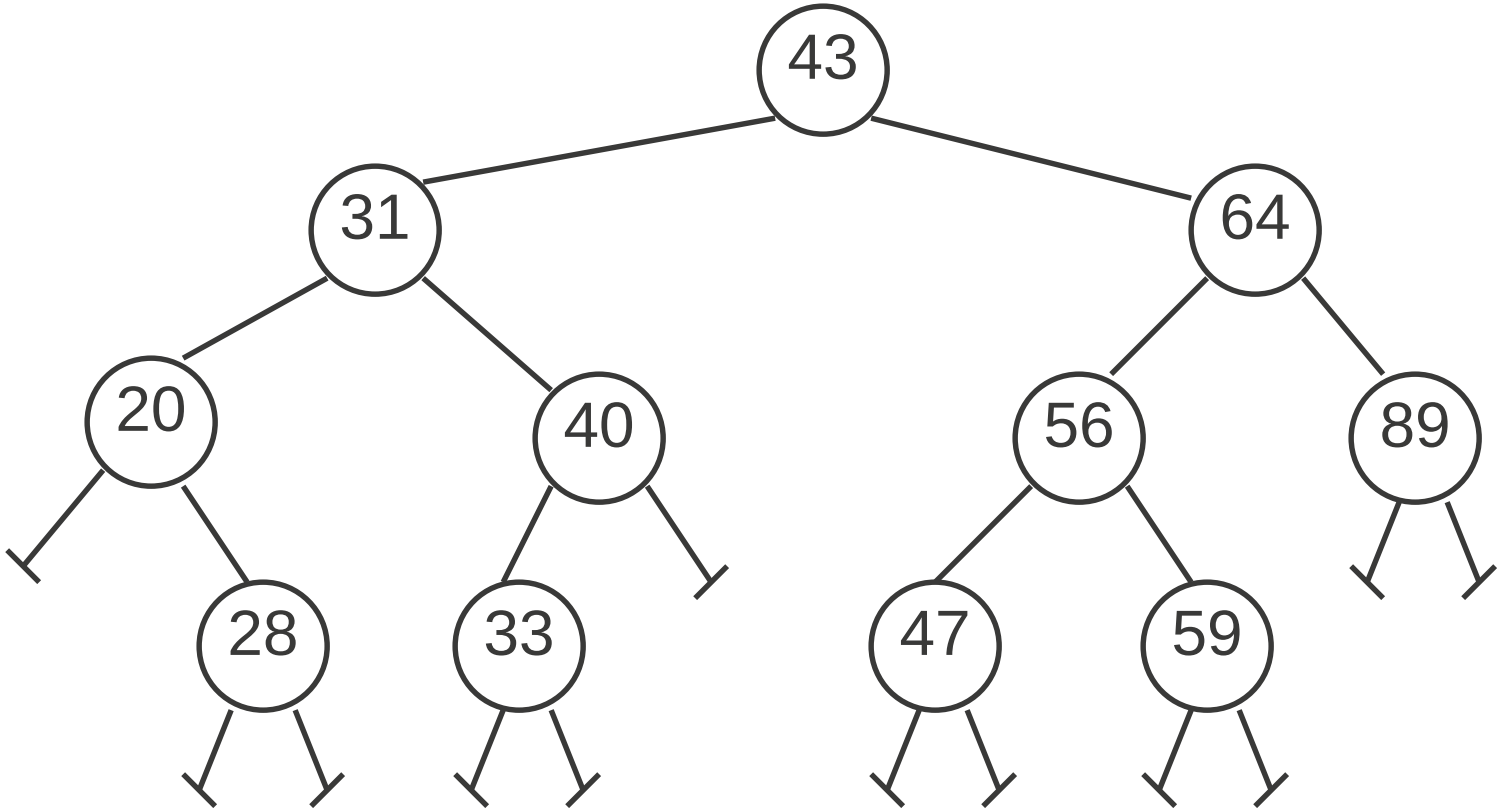
MONASH University

# BST Deletion

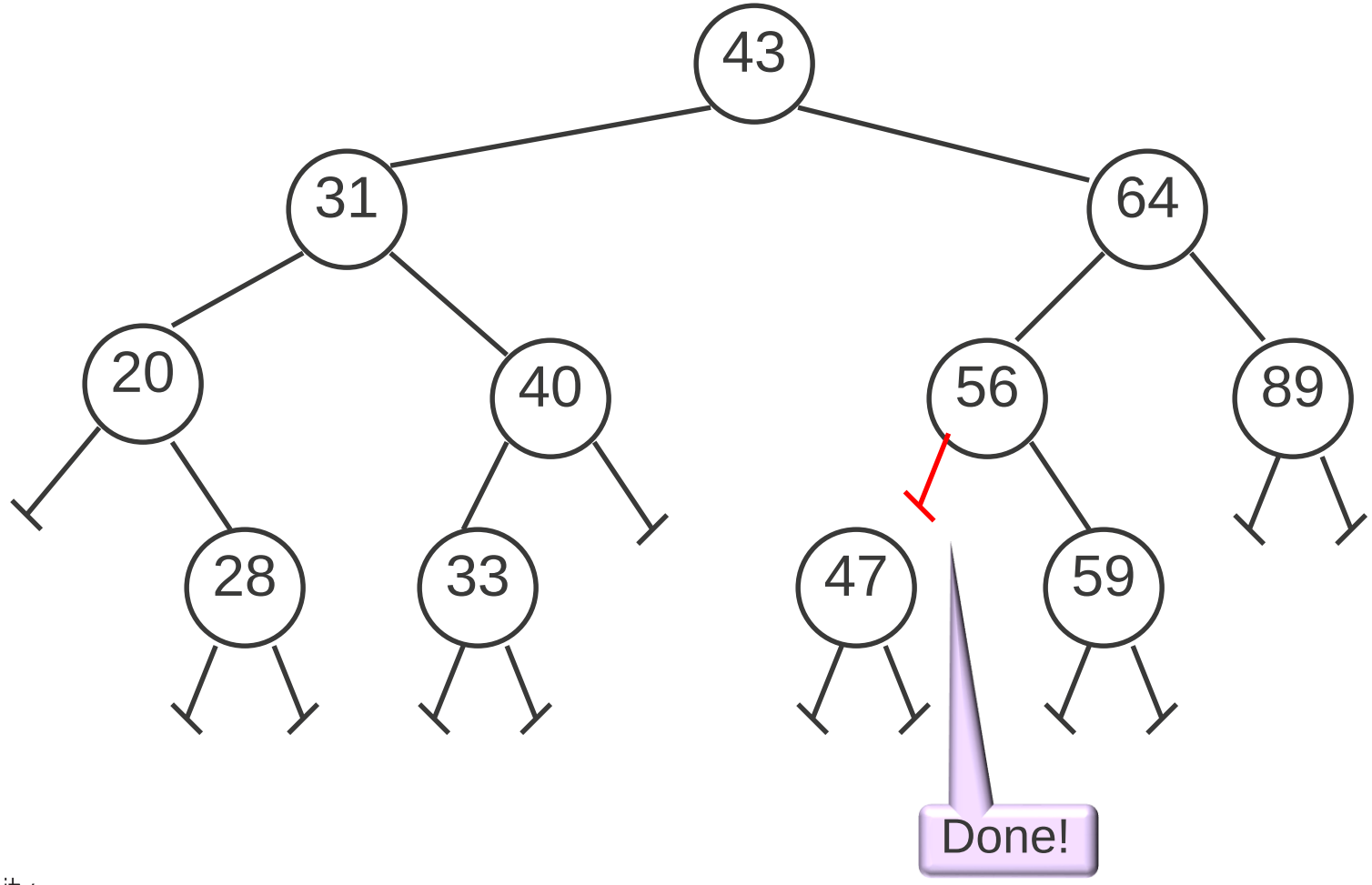
# Deleting a node

- **Find the node you want to delete**
- **If the node is:**
  - A leaf:

# Delete example: node with key 47 (leaf)



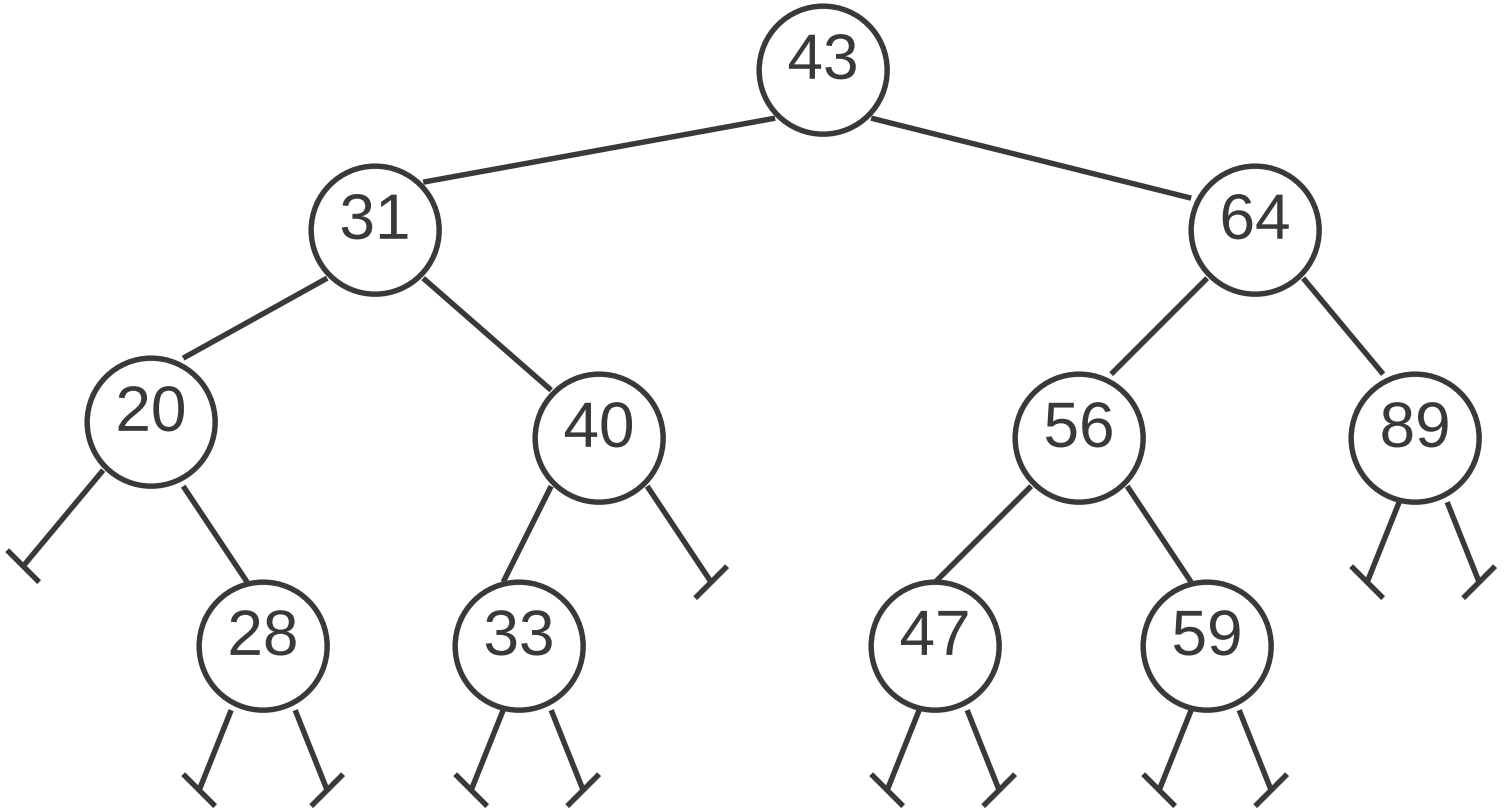
# Delete example: node with key 47 (leaf)



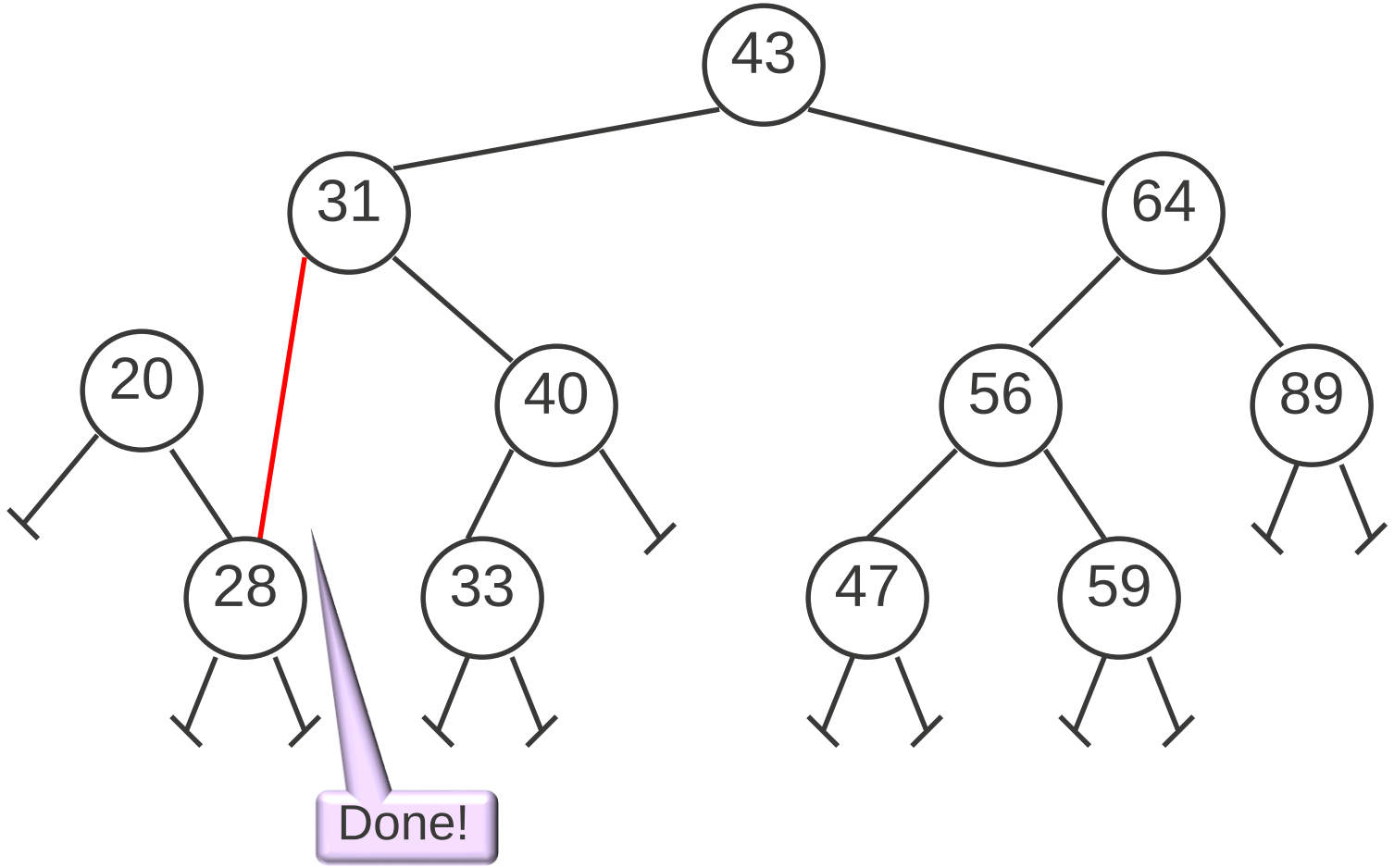
# Deleting a node

- **Find the node you want to delete**
- **If the node is:**
  - A leaf: easy, the parent's reference is set to `None`
  - A node with only one child:

# Delete node with key 20 (has single child)



# Delete node with key 20 (has single child)

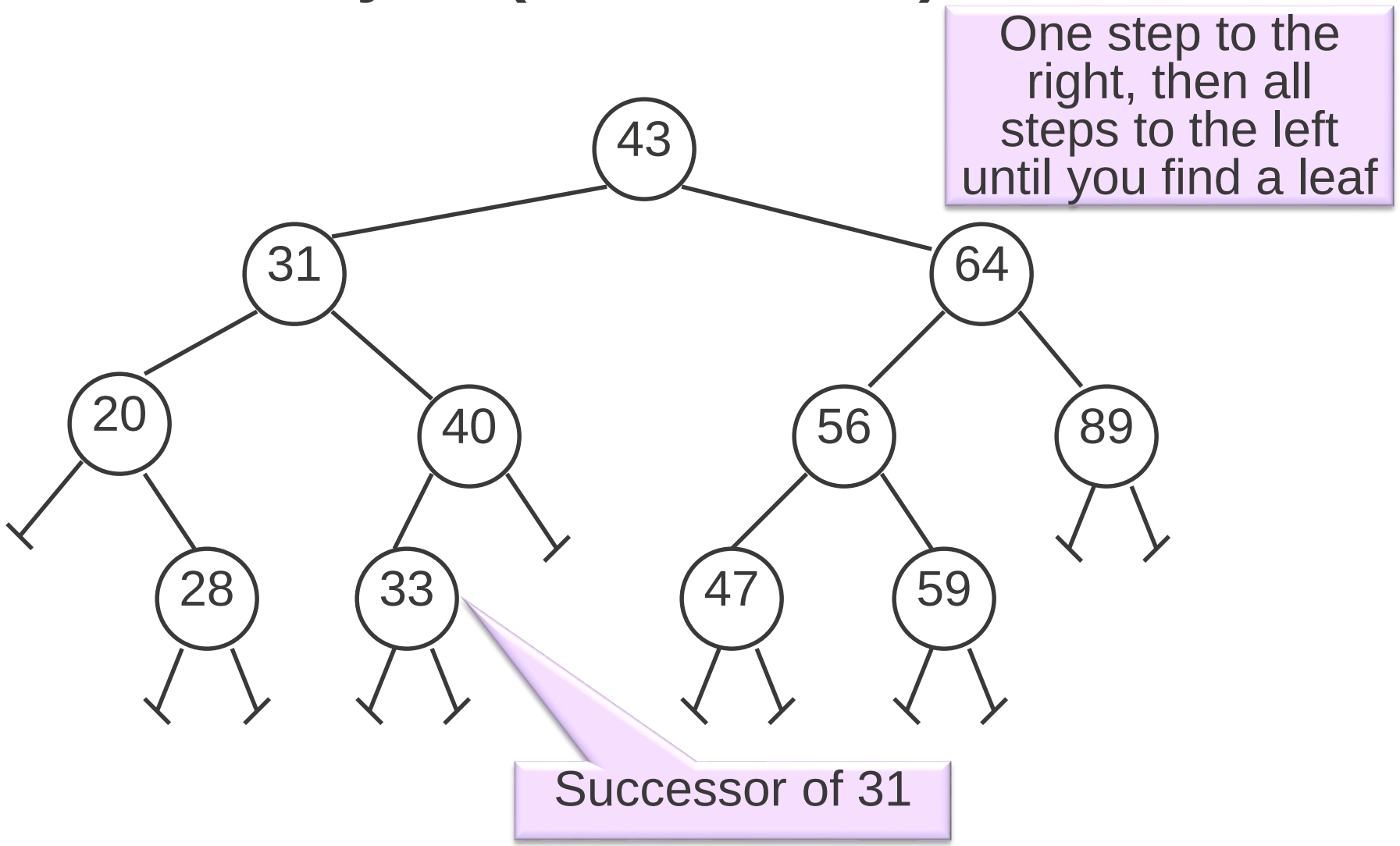


# Deleting a node

- **Find the node you want to delete**
- **If the node is:**
  - A leaf: easy, the parent's reference is set to **`None`**
  - A node with only one child: easy, the parent's reference is set to the child (bypassing the node)
  - Otherwise, it is more complicated
    - Requires finding the node's successor, i.e., the node with the next largest key
    - How do you find the successor in a BST?

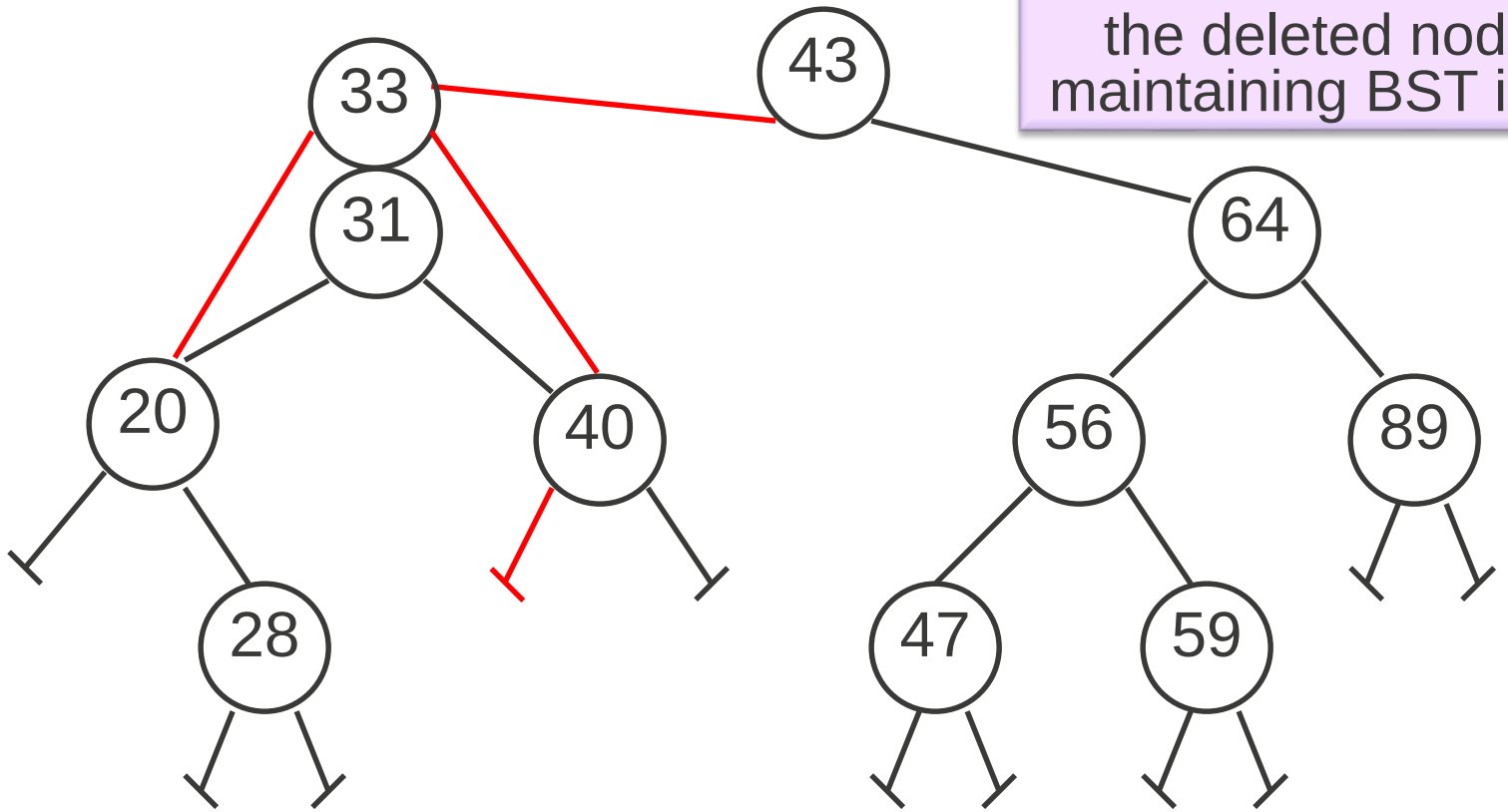
Alternatively, you can also use the *predecessor*

# Delete node with key 31 (two children)



# Delete node with key 31 (two children)

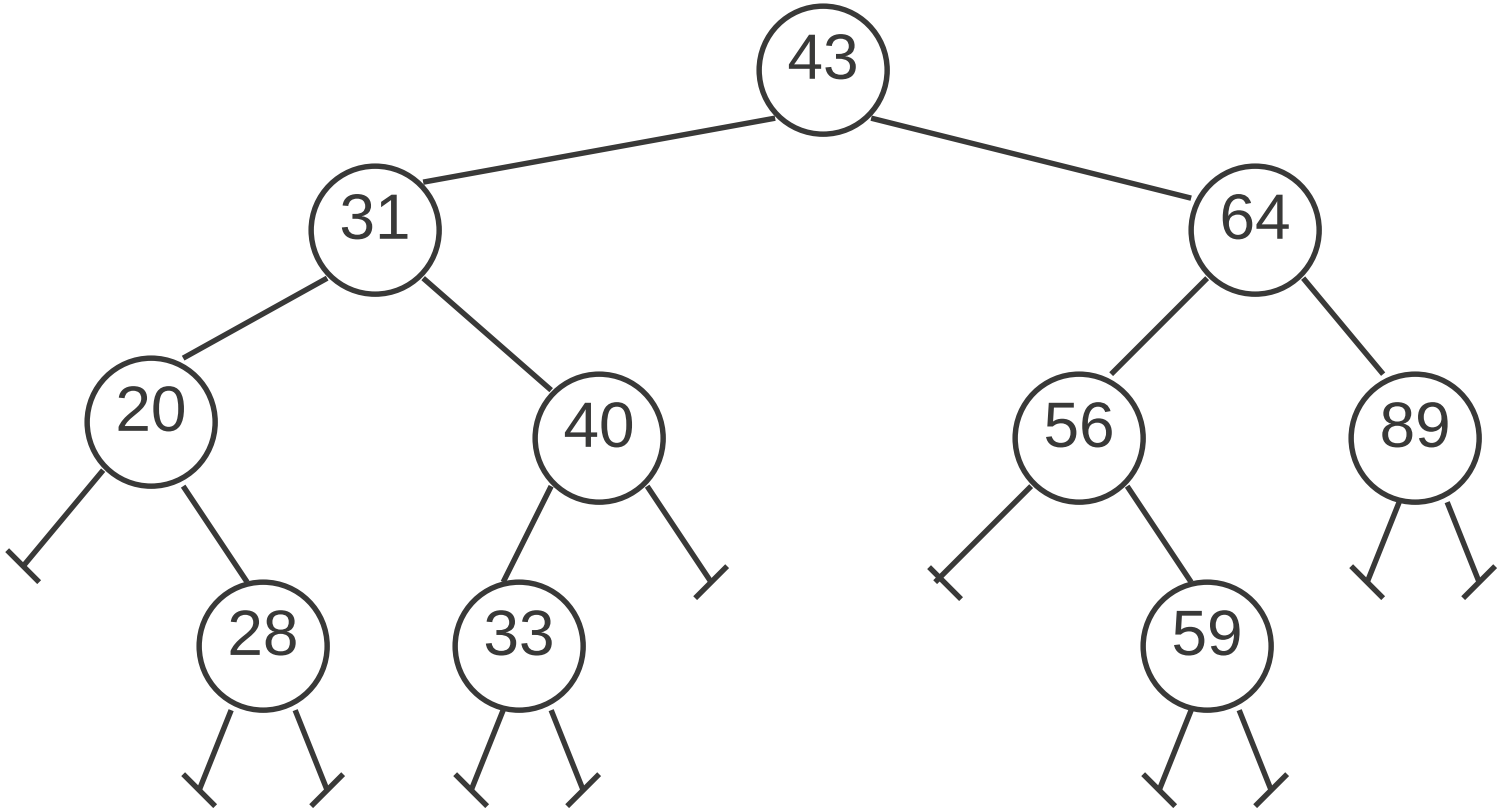
The successor is useful because it can substitute the deleted node while maintaining BST invariants



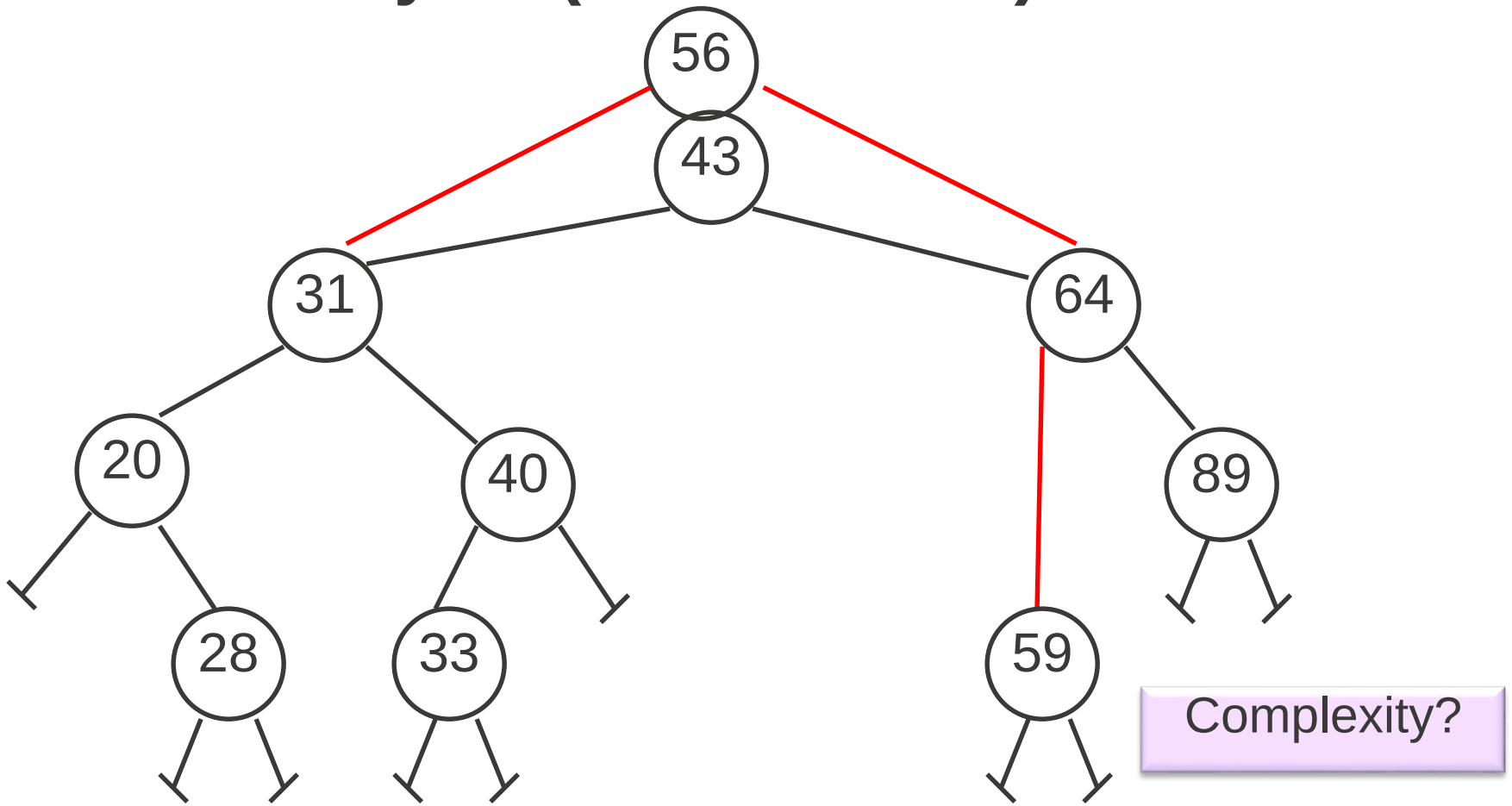
# Deleting a node

- **The previous case was easy:**
  - Successor had no children
- **The successor can never have a left child**
  - Why?
  - If so, the left child would be the successor
- **But what if it has a right child?**

# Delete node with key 43 (two children)



# Delete node with key 43 (two children)




Complexity?

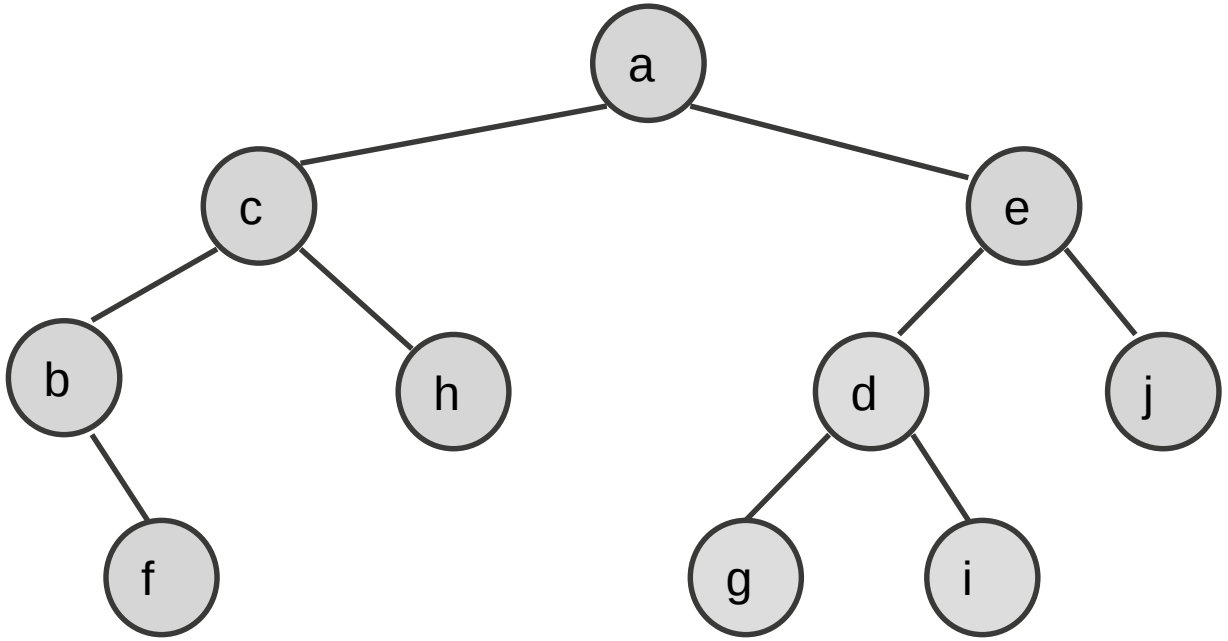
This case can be implemented as the one with no children (we link with its left child, even if it is empty)

# Correctness again

- **Is this algorithm correct too?**
  - Does it maintain the binary search tree invariants?
  - Think about it

# Example: collecting the leaves of a tree

Add a recursive method to `BinaryTree` that returns a `List` with all the leaves in the tree in left-to-right order (without modifying the tree), assume you can use the `append` method for lists.




For example, the above tree will return the list `[f, h, g, i, j]` where `f` is at the head of the list

Convergence? Same (but needs to pass the list around – accumulator!)

Base case? Two: empty (does nothing) and leaf (gets added)

Combination of solutions? Immediate, just pass the list around.

# Example: collecting the leaves of a tree

```
def get_leaves(self) -> LinkList:
    a_list = LinkList(len(self))
    self.get_leaves_aux(self.root, a_list)
    return a_list


def get_leaves_aux(self, current: BinarySearchTreeNode[K, I], a_list: LinkList) -> None:
    if current is not None:
        if current.left is None and current.right is None:
            a_list.append(current.item)
        else:
            self.get_leaves_aux(current.left, a_list)
            self.get_leaves_aux(current.right, a_list)
```

Better as auxiliary method like `is_leaf(current)`

# Example: collecting the leaves of a tree

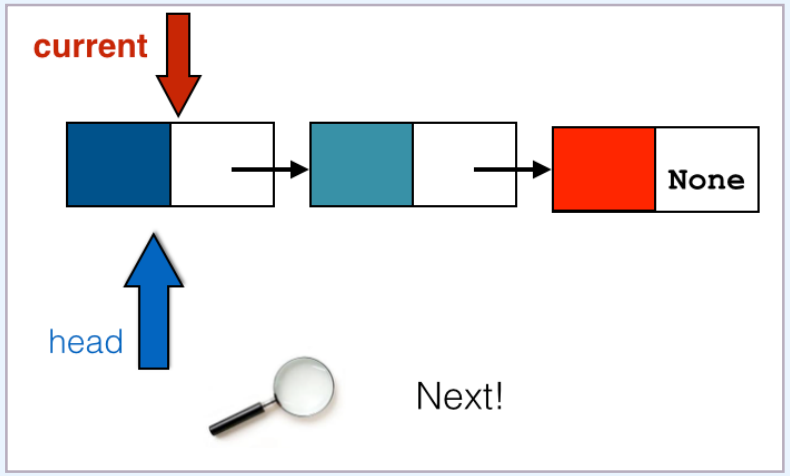
```
def get_leaves(self) -> List:
    a_list = List(len(self))
    self.get_leaves_aux(self.root, a_list)
    return a_list


def get_leaves_aux(self, current: BinarySearchTreeNode[K, I], a_list: List) -> None:
    if current is not None:
        if self.is_leaf(current):
            a_list.add_last(current.item)
        else:
            self.get_leaves_aux(current.left, a_list)
            self.get_leaves_aux(current.right, a_list)

def is_leaf(self, current: BinarySearchTreeNode) -> bool:
    return current.left is None and current.right is None
```

Better as auxiliary method like `is_leaf(current)`

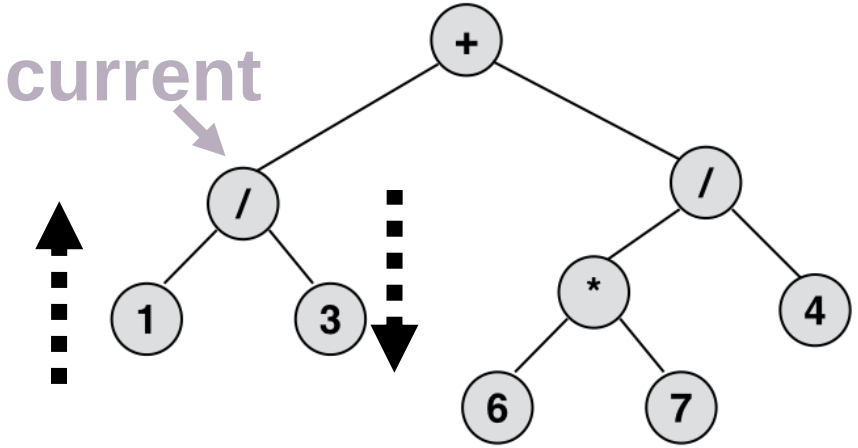
MONASH University

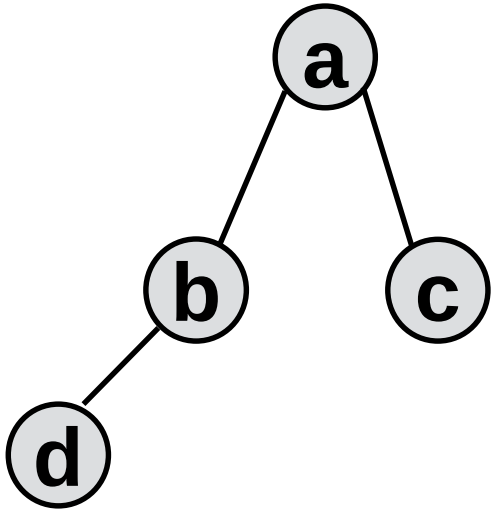
# BST Iterator

```python
class LinkListIterator(Generic[T]):
    def __init__(self, node: Node[T]) -> None:
        self.current = node

    def __iter__(self) -> 'LinkListIterator':
        return self

    def __next__(self) -> T:
        if self.current is not None:
            item = self.current.item
            self.current = self.current.link
            return item
        else:
            raise StopIteration
```




?

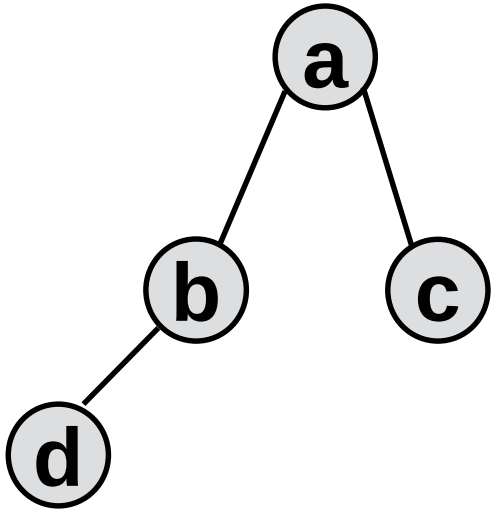





Iterate in what order?

Lets say pre-order

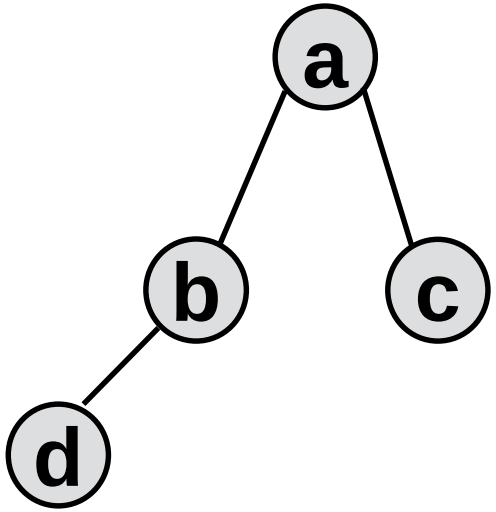
# State of the **Iterator** on creation

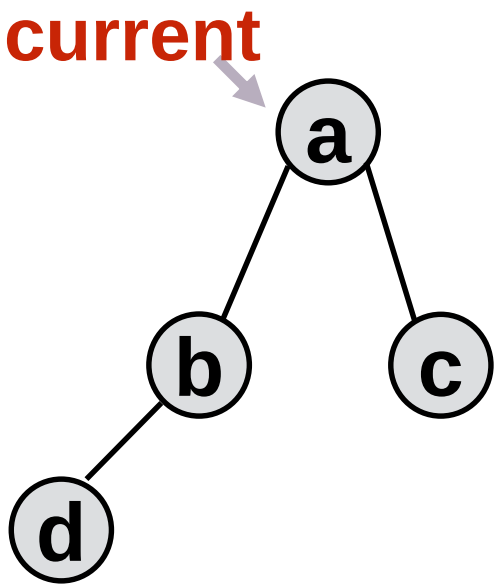


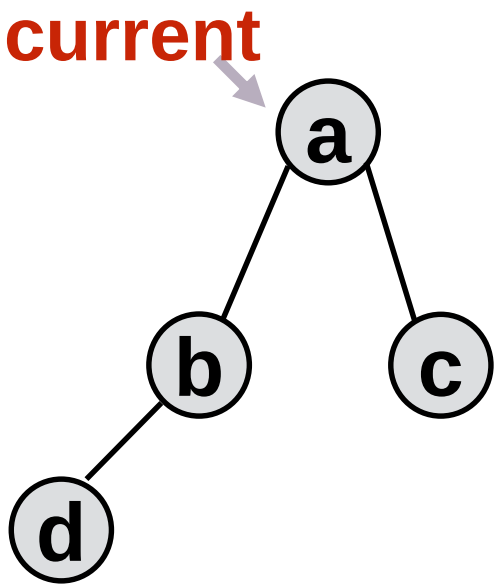


self.stack

# Next!

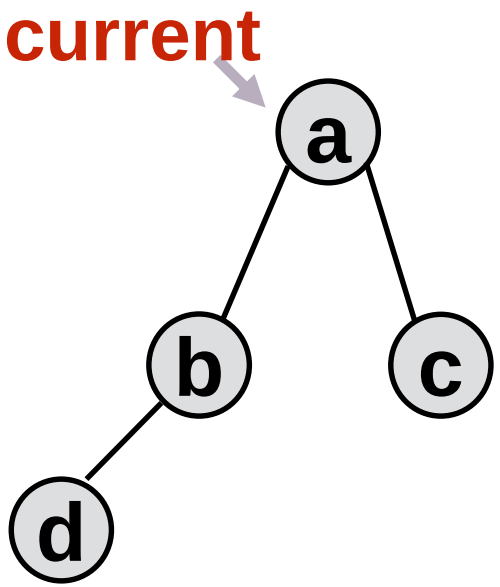


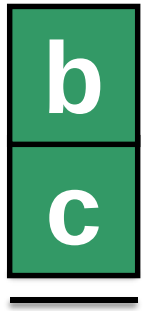


current
a
b
c
d





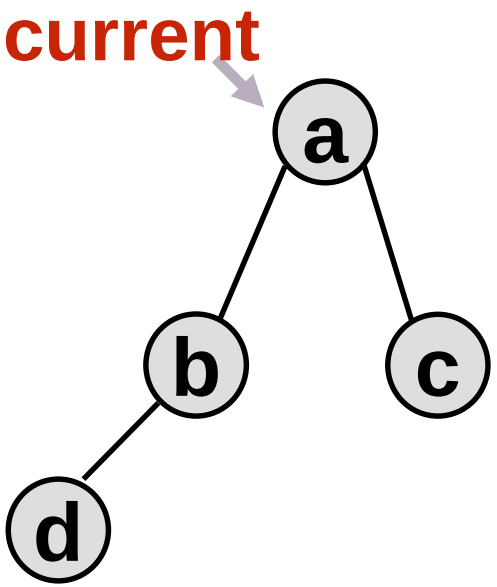
First push what is to the right of current

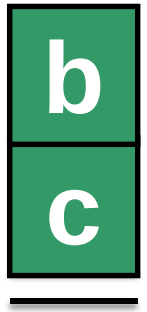



And then push what is to the left of current

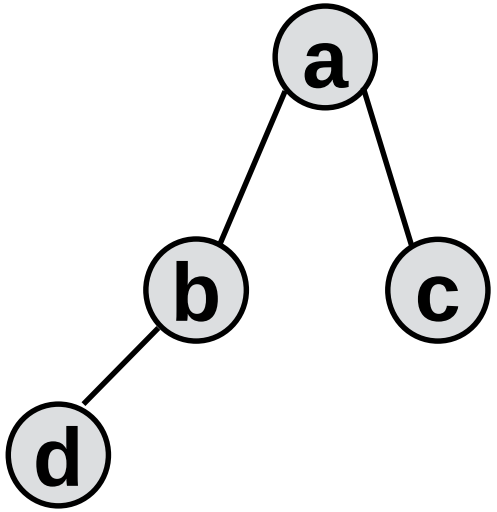
**return** current.item

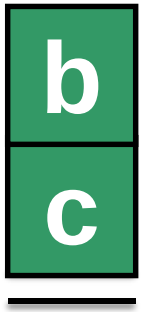


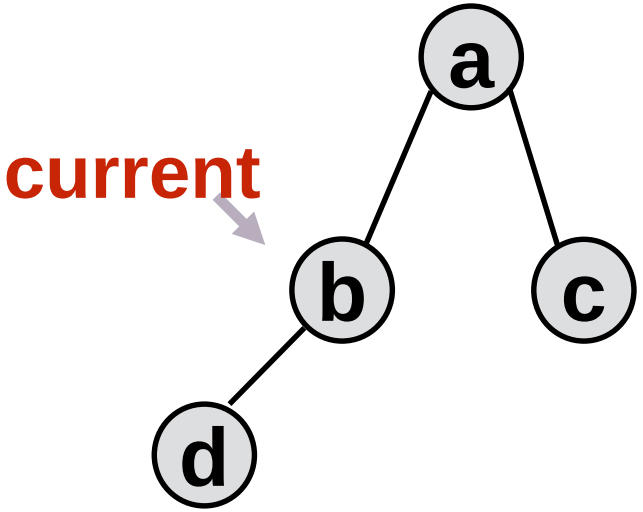

**a**

Next!



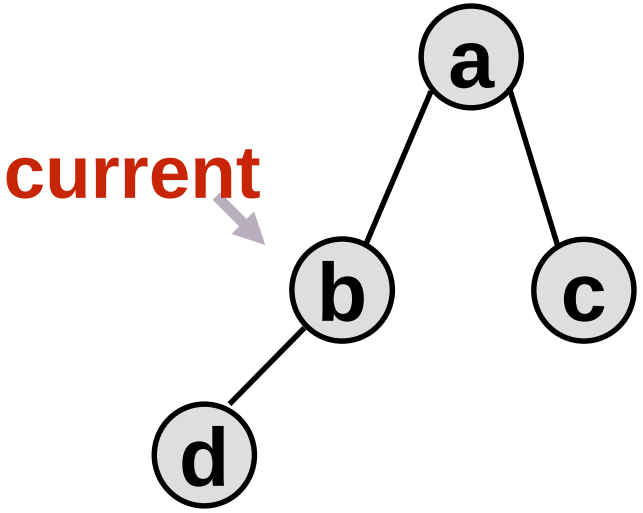

a





a

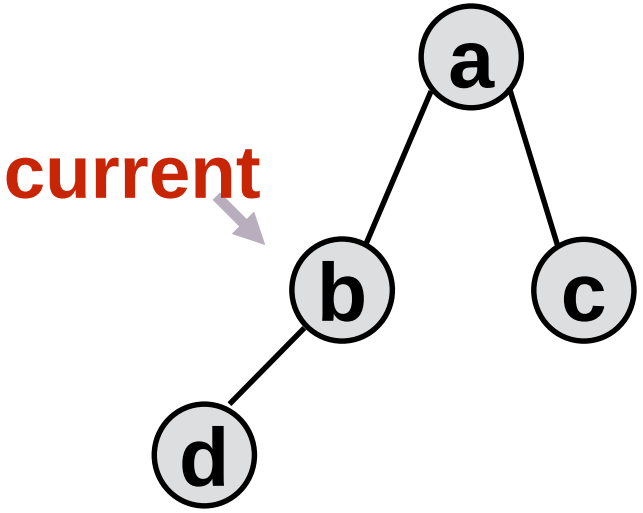
Nothing to push on right

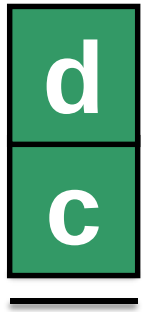




a

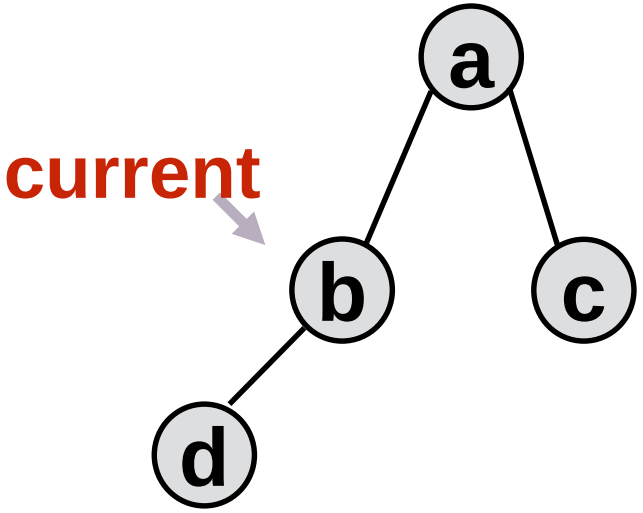
Push what is to the left of current.

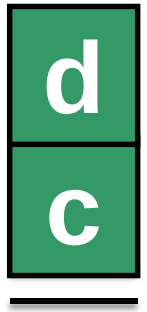


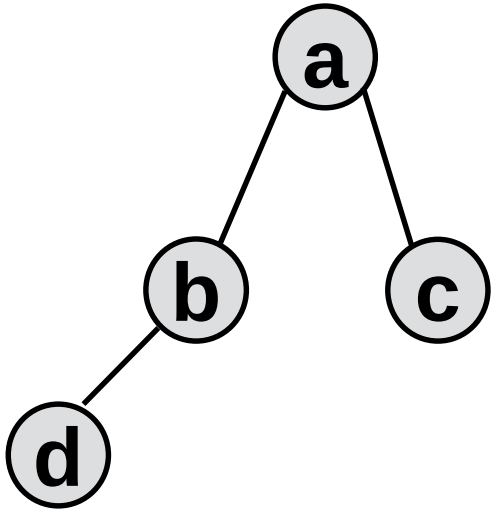

a

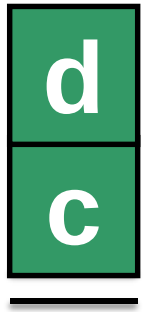
**return** current.item



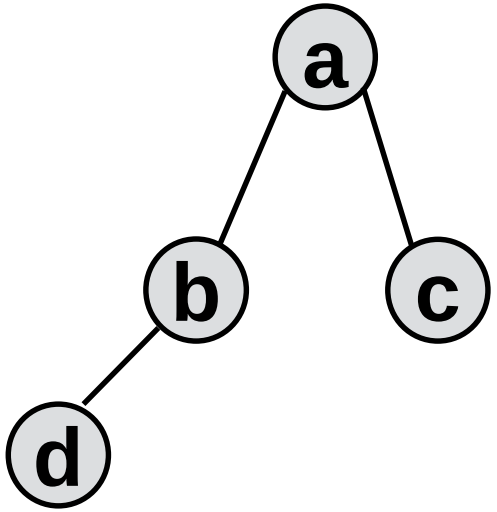

**ab**

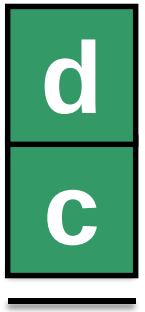


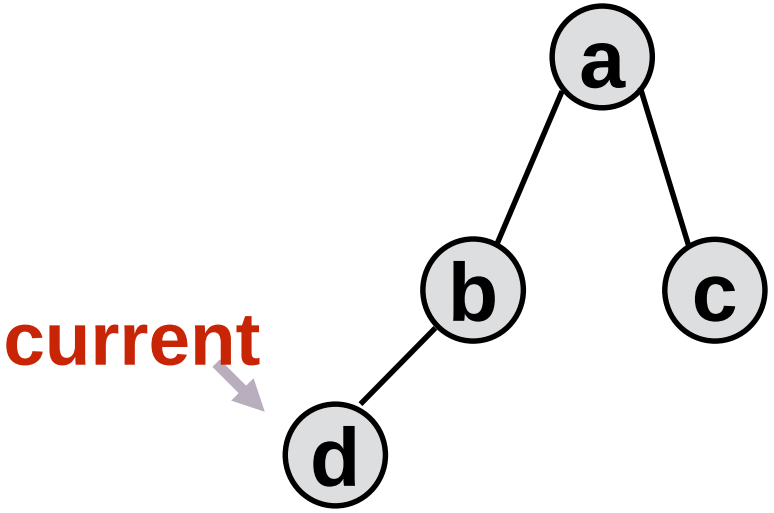

**ab**

Next!



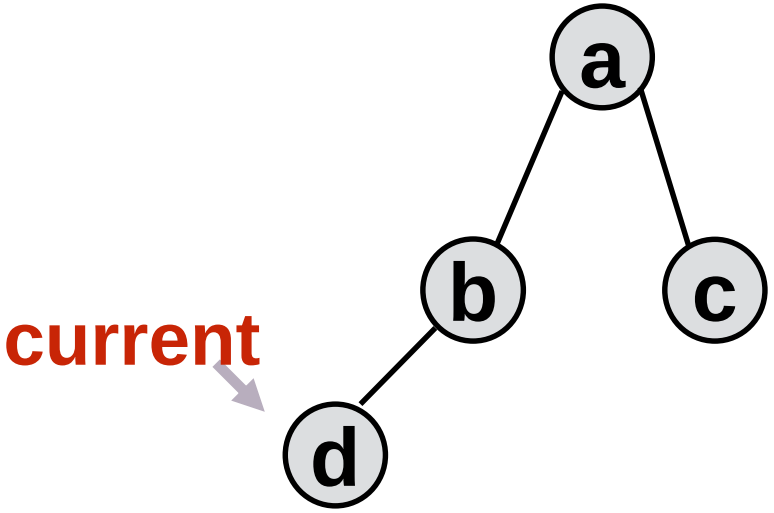




**ab**



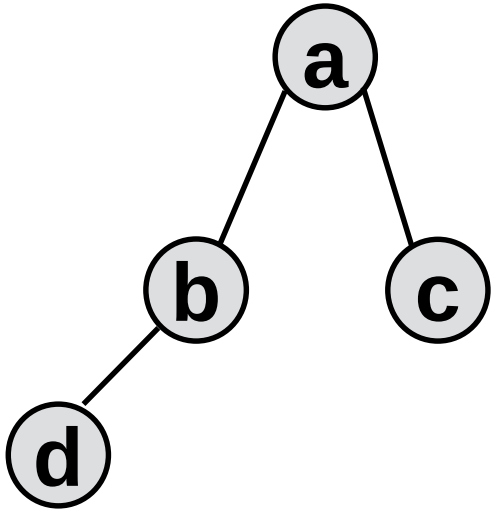


**ab**





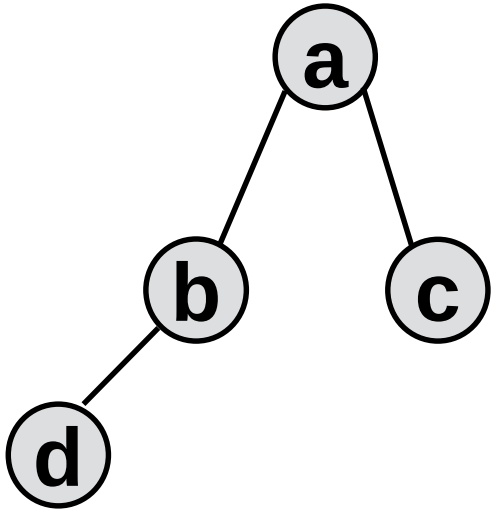
**a b d**





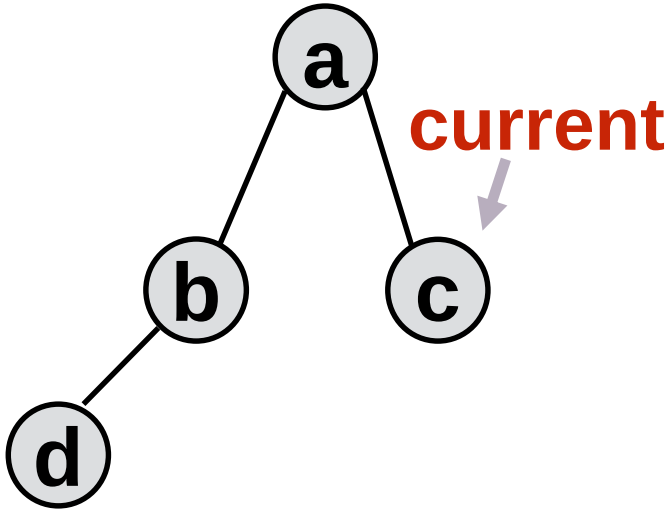
**a b d**

Next!



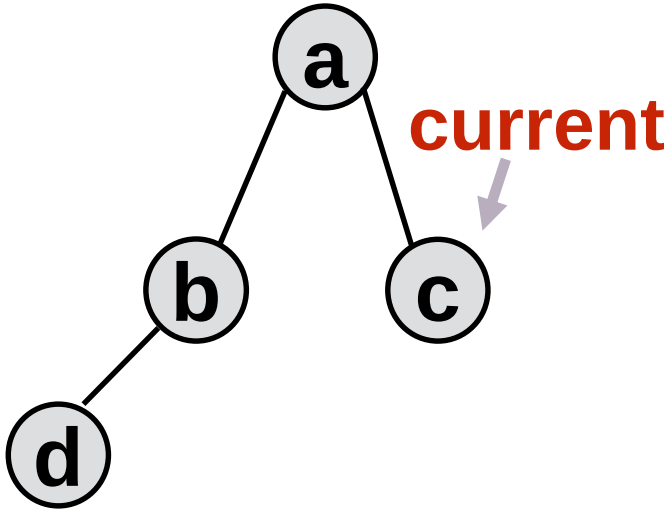


**a b d**

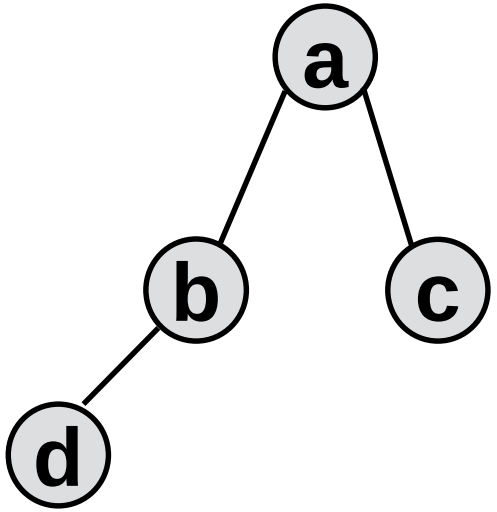



# a b d

**return** current.item

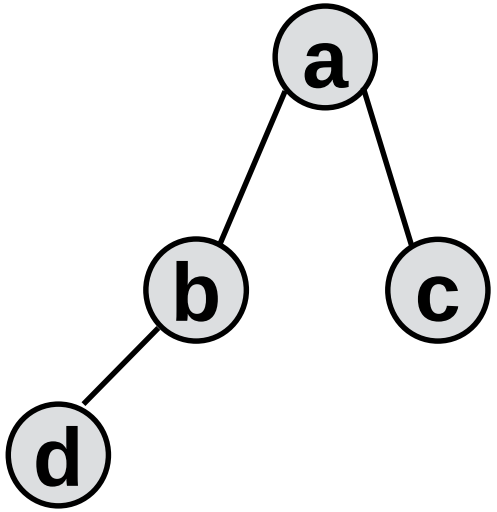



**a b d c**

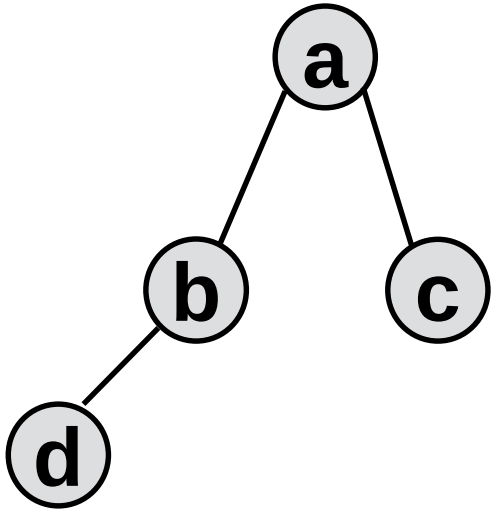



**a b d c**

Next!




**a b d c**




**StopIteration**

**a b d c**

**preorder!**

# How does the core algorithm look like?

- **Something like:**

```
self.current = self.stack.pop()
self.stack.push(self.current.right)
self.stack.push(self.current.left)
return current
```

- **But with some checks:**
  - Not to add nodes when they are None
  - Raise **StopIteration** when all nodes have been traversed

# PreOrder Iterator class

```
def __init__(self, root: BinarySearchTreeNode[K, I]) -> None:
    self.stack = Stack()
    self.stack.push(root)
def __iter__(self):
    return self
def __next__(self) -> T:
    if self.stack.is_empty():
        raise StopIteration
    current = self.stack.pop()
    if current.right is not None:
        self.stack.push(current.right)
    if current.left is not None:
        self.stack.push(current.left)
    return current.item
```

```
class LinkListIterator(Generic[T]):
    def __init__(self, node: Node[T]) -> None:
        self.current = node

    def __iter__(self) -> 'LinkListIterator':
        return self

    def __next__(self) -> T:
        if self.current is not None:
            item = self.current.item
            self.current = self.current.link
            return item
        else:
            raise StopIteration
```

Might need to return (key, item) if a BST

# And now we can use it!

```
my_tree.print_preorder()
```

```
2
5
3
```

```
for i in my_tree:
    print(i)
```

```
2
5
3
```

**In BinaryTree:**

```
def __iter__(self):
    return PreOrderIteratorStack(self.root)
```

# Summary

- **Expression trees: prefix, infix,  postfix**
- **Binary search trees: search, insertion and deletion**
- **Iterators for Binary Trees**