
SUPERB
wallpapers

Recursion

2

AND

BINARY SEARCH TREE

Overview

- What is recursion?
- Recursive versus Iterative
- “Optimisation”… tail recursion elimination

Definition

- From Wikipedia:
  In mathematical logic and computer science, recursive definition (or inductive definition) is used to define an object in terms of itself .
  (Aczel 1978:740ff).

Definition

- In mathematics and computer science, a class of objects or methods exhibit recursive behaviour when they can be defined by two properties:

1. A simple **base case** (or cases), and
2. A set of rules which **reduce** all other cases **toward the base case**.

Factorial: n!

- the factorial function n! is defined by the rules
  1. $0! = 1$
  2. $(n+1)! = (n+1) \cdot n!$

- Recursive approaches rely on the fact that it is usually simpler to solve a smaller problem than a larger one.

- The recursive case usually contains a call to the very function being executed.
  - This call is known as a *recursive* call.

Code

Code Recursive

```
def factorial_rec(n):
      return n * factorial_rec(n - 1)
```

WRONG

Code Recursive

```
def factorial_rec(n):
  if (n == 0): ## Base case
      return 1
  else:
      return n * factorial_rec(n - 1) ## Recursive call
```

Code Iterative

```
def factorial_iter(n):
   total = 1
   for i in range(1,n+1):
       total = total * i
   return total
```

Recursion vs Iteration

All **recursion** **<u>can</u>** be written in an **iterative** form

Recursion vs Iteration

Question: Does using recursion usually make your code faster?

Answer: **No**.

Question: Does using recursion usually use less memory?

Answer: **No**.

Recursion vs Iteration

Question: Then **why** use recursion?

Answer: It sometimes makes your code **much simpler**!

How does it work?

- When a recursive call is made, the method clones itself, making new copies of:
  - the code
  - the local variables (with their initial values),
  - the parameters
- Each copy of the code includes a marker indicating the current position.
  - When a recursive call is made, the marker in the old copy of the code is just after the call;
  - the marker in the "cloned" copy is at the beginning of the method.
- When the method returns, that clone goes away
  - but the previous ones are still there, and know what to execute next because their current position in the code was saved (indicated by the marker).
- See Will's lecture 13 for another description

Example: factorial_rec(2)

Code Recursive

```
factorial_rec(2):
  if (2 == 0): ## Base case
    return 1
  else:
    return 2 *
```

```
factorial_rec(1):
  if (1 == 0): ## Base case
    return 1
  else:
    return 1 *
```

```
factorial_rec(0):
  if (0 == 0): ## Base case
    return 1
  else:
    return 1 * factorial_rec(0 - 1)
```

Example: factorial_rec(2)

Code Recursive

```
factorial_rec(2):
  if (2 == 0): ## Base case
    return 1
  else:
    return
```

2

```
  if   == 0): ## Base case
    r turn 1
  els :
    r turn 1 *
```

1

```
  if (0 == 0): ## Base case
    return 1
  else:
    return 1 * factorial_rec(0 - 1)
```

Tail recursion

- a **tail call** is a subroutine call that happens inside another function as its final action
- **Tail-call optimization** is where you are able to avoid allocating a new stack frame for a function
  - the calling function will simply return the value that it gets from the called function.
- Traditionally, **tail call elimination** is optional
  - in **functional** programming languages, tail call elimination is often guaranteed by the language standard
  - Python **does not** implement tail call elimination

How to use recursion in the LinquedQueue data structure developed during the practical

- Implementing `_contains__(element)`:

Code Iterative

```
def __contains__(self, element):
        currentNode = self._head
        while currentNode is not None:
            if currentNode.data == element:
                return True
            else:
                currentNode = currentNode.next
        return False
```

Summary

- Use recursion for clarity,
  - and (sometimes) for a reduction in the time needed to write and debug code, not for space savings or speed of execution.
- Remember that every recursive method must have a base case (rule #1).
- Also remember that every recursive method must make progress towards its base case (rule #2).

Summary

- Every recursive function can be written as an iterative function
- Recursion is often simple and elegant, can be efficient, and tends to be underutilized. Consider using recursion when solving a problem!

Binary Search Tree

AN IMPLEMENTATION

USING

OBJECT ORIENTED PROGRAMMING (OOP)

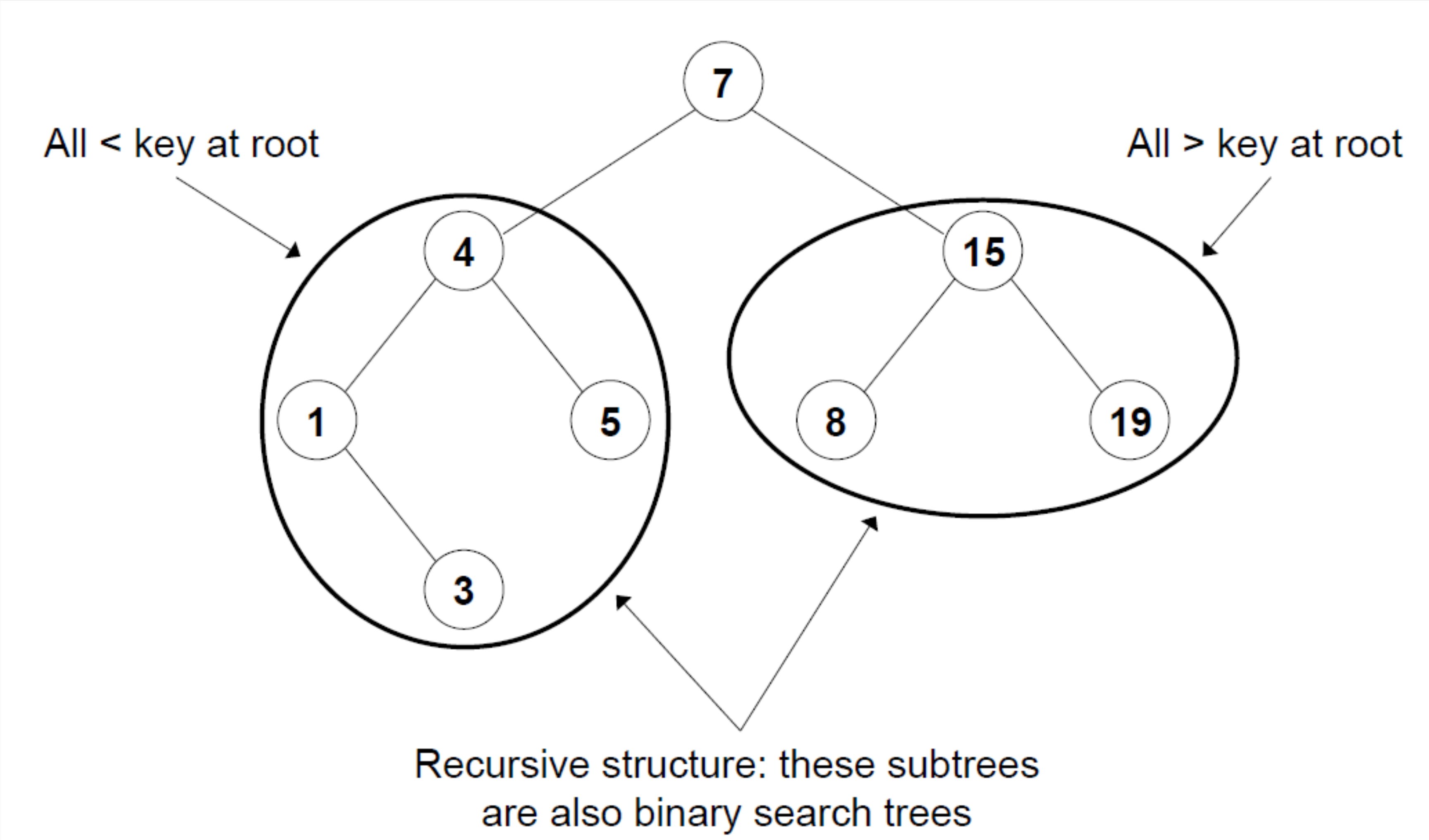
Review BST

- A ***binary tree*** is a special case of a tree in which nodes have a degree of at most 2

- The two children of a node are called the left and right child

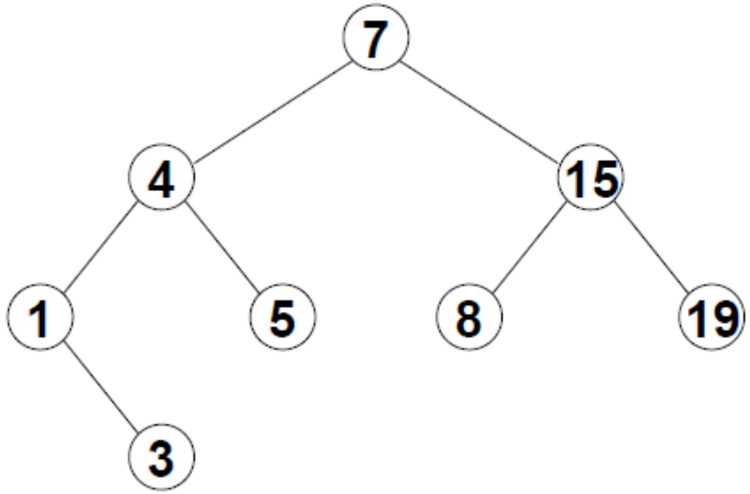
- ***Binary search trees*** are a type of binary tree which are extremely useful for storing and retrieving ordered data:
  - Each node is labelled with a key
  - Nodes in the left subtree have keys smaller than that of the root
  - Nodes in the right subtree have keys greater than that of the root

Review BST

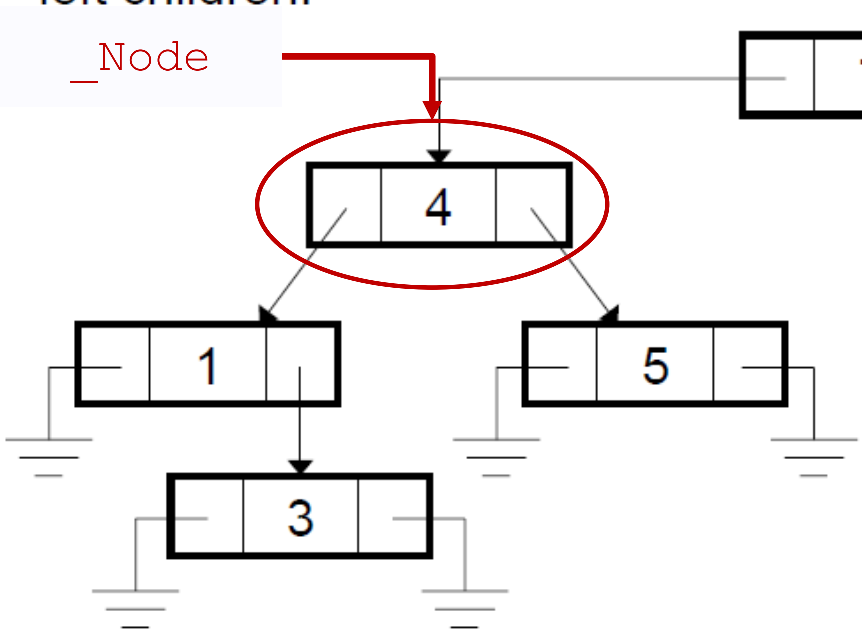

7
All < key at root
All > key at root
4
15
1
5
8
19
3
Recursive structure: these subtrees
are also binary search trees

Review BST

To store a binary tree in a linked structure:



Each node consists of a record containing the key/datum plus pointers to records representing its right and left children:



From this point we assume a linked implementation is used

OOP Implementation

- Encapsulation as Information hiding
  - Data belonging to one object is hidden from other objects.
  - Know what an object can do, not how it does it.
    - NOT allowed to change key value
    - allowed to change datum/data, and left/right children

Code OOP

```
class _Node:
    def __init__(self, key, data):
        assert key is not None,
                'BST_Node constructor: Illegal Argument '
        self._key = key
        self._data = data
        self._left = None
        self._right = None
```

OOP Implementation

- What _Node **can do:** (see binary_search_tree.py)

Code OOP

```
class _Node:
    def __init__(self, key, data): # the constructor

    def key(self): # returns the key

    def getData(self): # returns the data

    def getLeft(self): # return the left child

    def getRight(self): #return the right child

    def setData(self, data): # change the data

    def setLeft(self, bstNode): # change the left child

    def setRight(self, bstNode): # change the right child

    def isleaf(self): # returns True if the node is a leaf
```

OOP Implementation

- **What** BSTree **can do:** (see binary_search_tree.py)

Code OOP

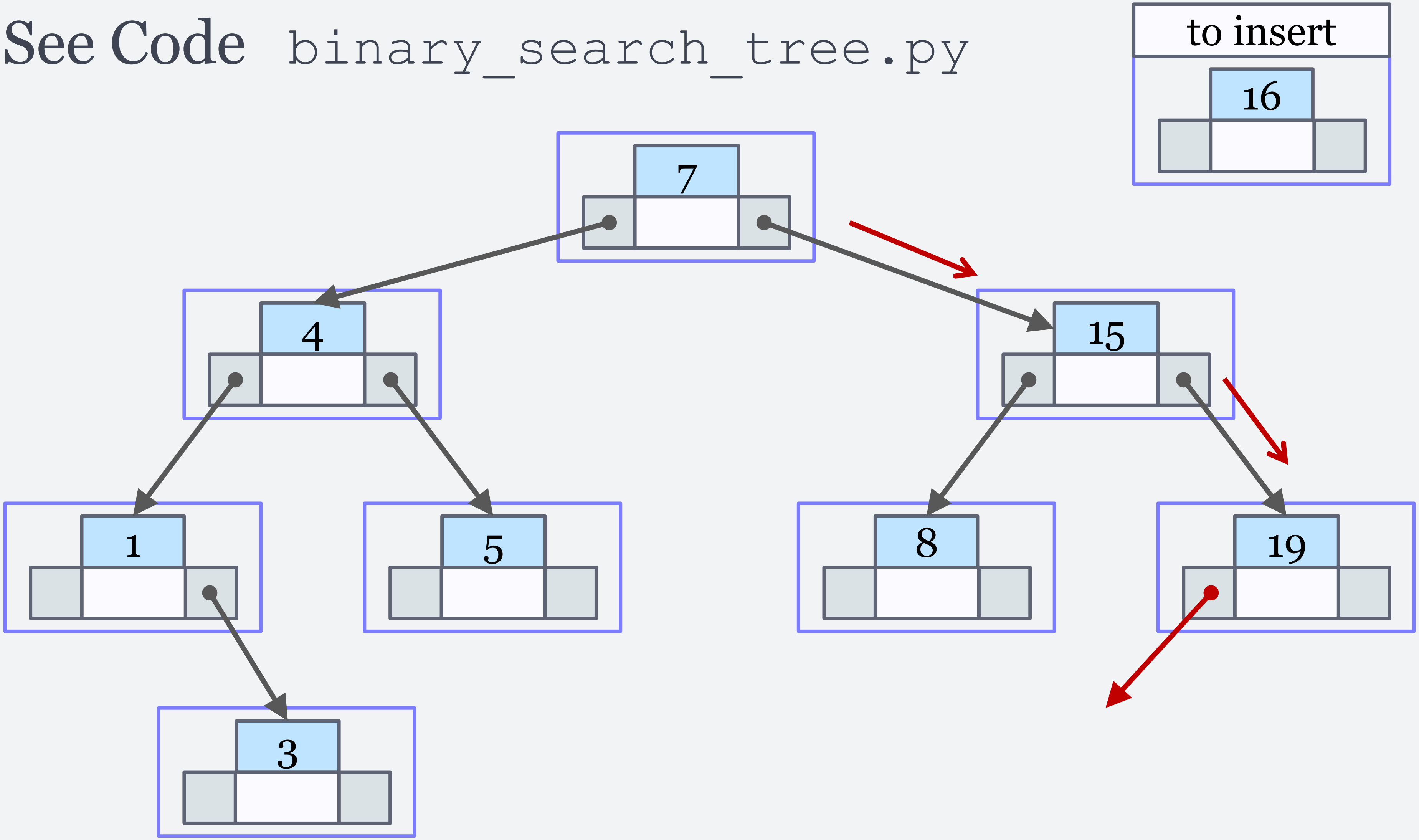
```
class BSTree:

    def __init__(self, bst_node = None): # the constructor
    def isempty(self):
    def insert(self, key, data): # insert a new node
    ## AND NORE…


    ## Helpers methods, also available for right branch
    def _getLeftTree (self):# returns the left child as a BSTree
    def _setLeftBranch(self, bst):
```

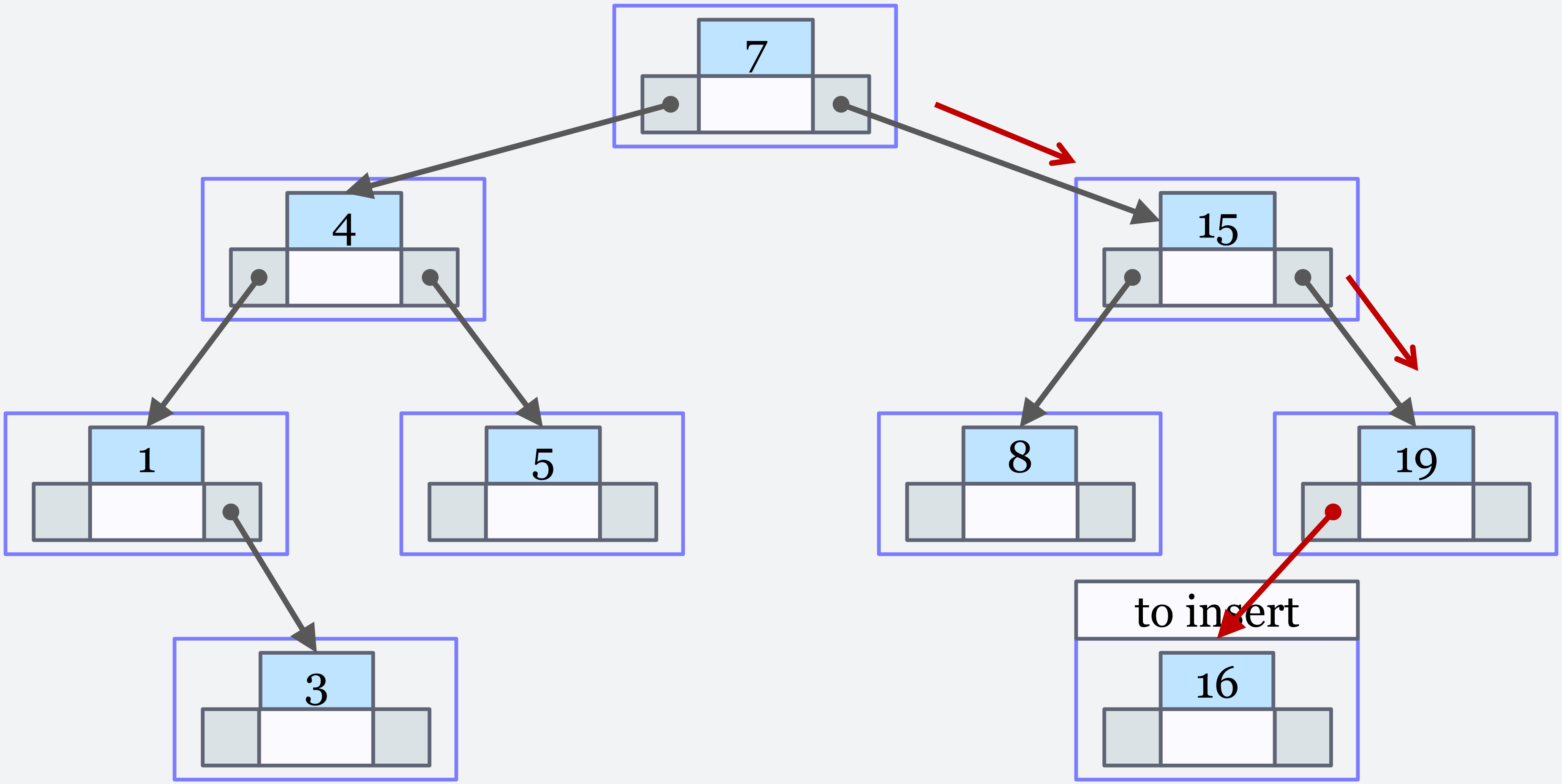
Insertion

- See Code `binary_search_tree.py`



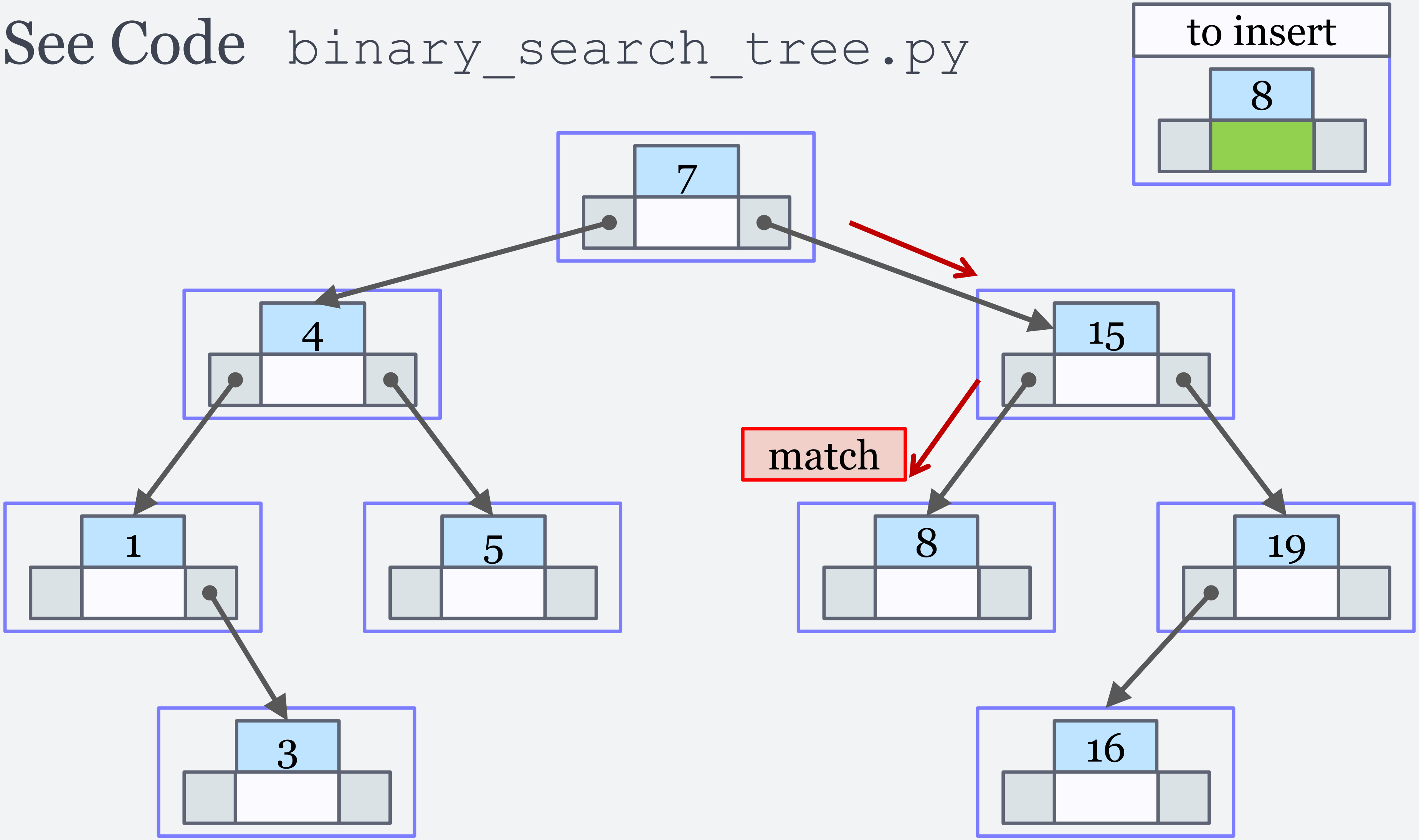
Insertion

- See Code `binary_search_tree.py`



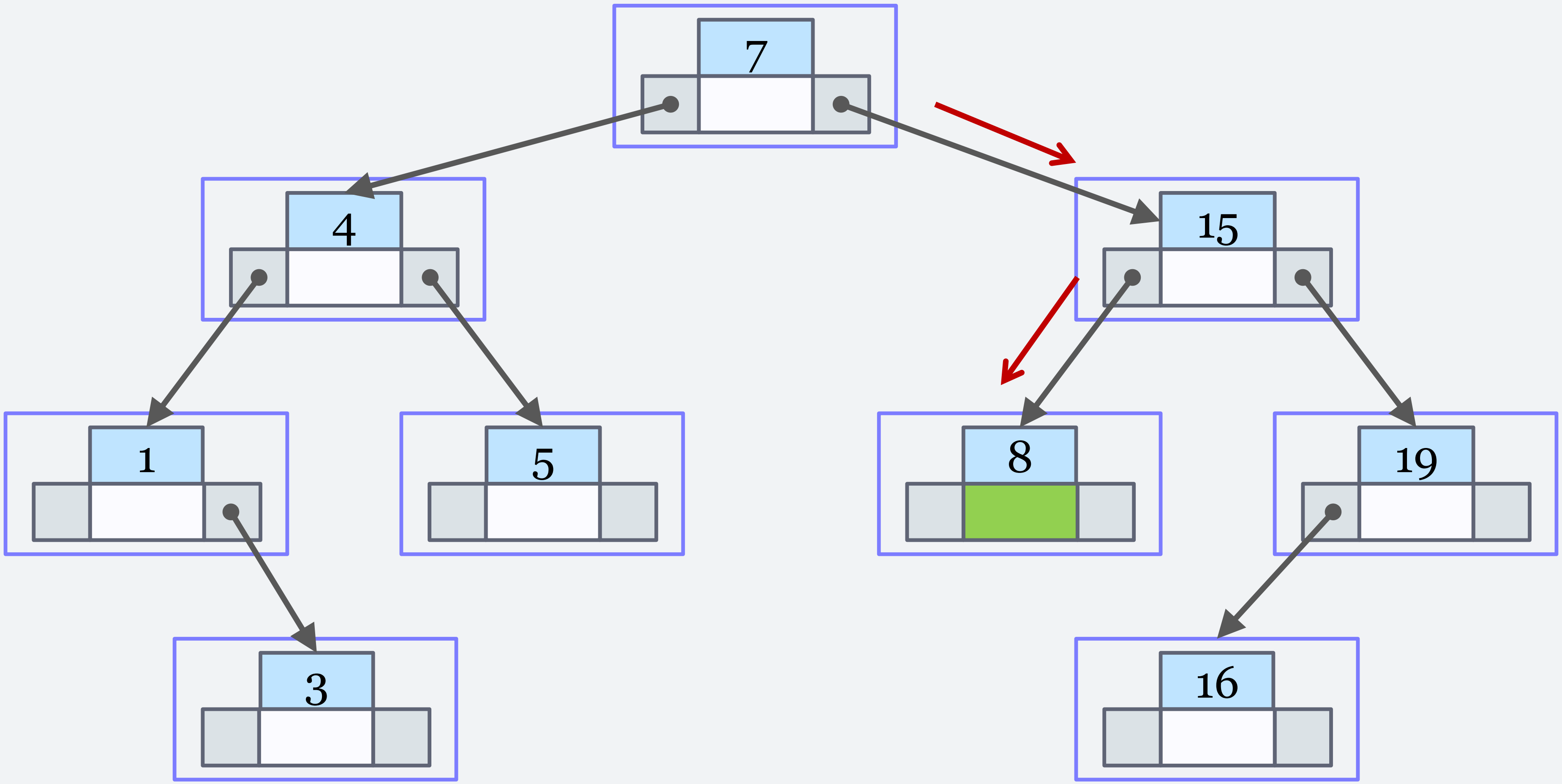
Insertion

- See Code `binary_search_tree.py`



Insertion

- See Code `binary_search_tree.py`



Application

How to use recursion in the BST data structure?

- Implementing `insert(key, element)`:

Code Iterative

```
def insertIterative(self, key, data):
    """
    iterative sample code to insert a data associated with a key
    """
    if self.isempty():
        self._root = self._Node(key,data)
    else:
        currentNode = self._root
        while True:
            if(currentNode.key() == key):
                currentNode.setData(data)
                break
            elif(currentNode.key() > key):
                if(currentNode.getLeft() is None):
                    currentNode.setLeft(self._Node(key,data))
                    break
                else:
                    currentNode = currentNode.getLeft()
            else:
                if(currentNode.getRight() is None):
                    currentNode.setRight(self._Node(key,data))
                    break
                else:
                    currentNode = currentNode.getRight()
```

Summary

- You should be familiar with
  - Encapsulation and information hiding
  - Implementing classes in OOP
  - Recursive algorithm
  - Binary Search tree implementation

Final Thought

- What does the following code, and in particular the last line of code DO?

Code

```
def _getRightTree(self):
    """
    returns the root's right branch as a BSTree object (possibly
    empty), None if the tree is empty.
    """
    if self.isempty():
        return None
    return BSTree(self._root.getRight())
```

The code wrap a _Node object into a BSTree object. This means that we can now use BSTree methods with this _Node

Exercise: the greatest common divisor

- Consider finding the greatest common divisor, the *gcd*, of two numbers.
  - For example, the *gcd* of 30 and 70 is 10, since 10 is the largest number that divides both 30 and 70 evenly.

- Euclid Algorithm:
  1. gcd(a,b) is b — if a divided by b has a remainder of zero
  2. gcd(a,b) is gcd(b, a % b) — otherwise