파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

이진탐색 트리(BST)의 ADT를 학습하고

이에 맞는 Node클래스를 구현할 수 있다

Data Structures in Python
Chapter 7 - 2

- **Binary Search Tree(BST)**
- BST Algorithms
- AVL Tree
- AVL Algorithms

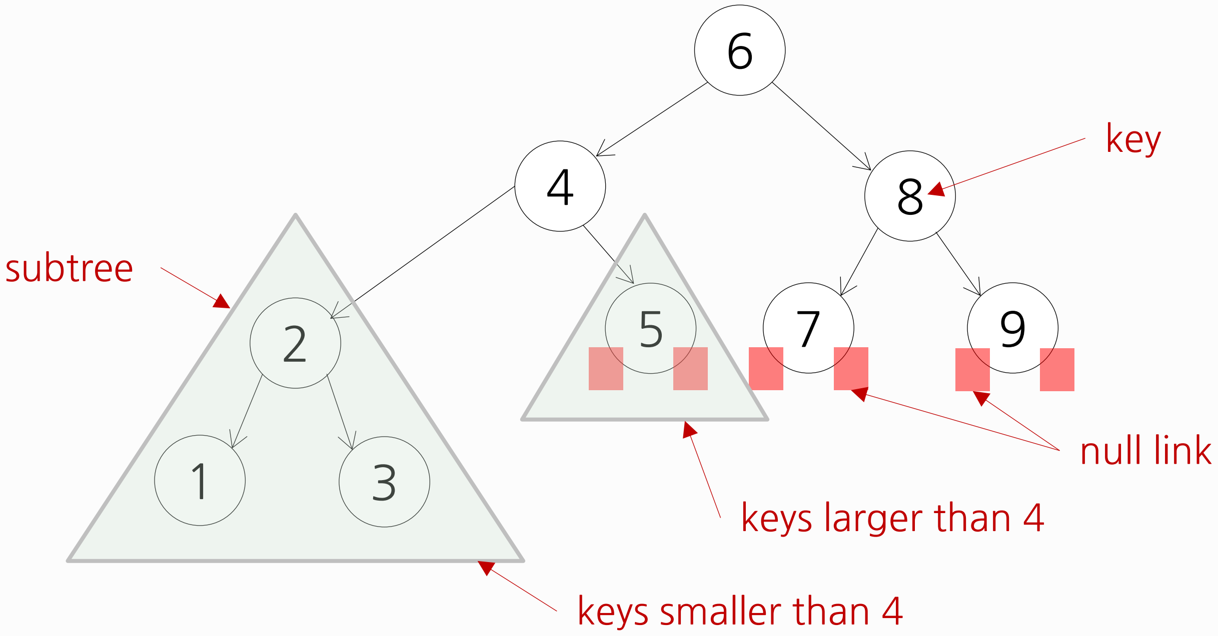
Agenda & Readings

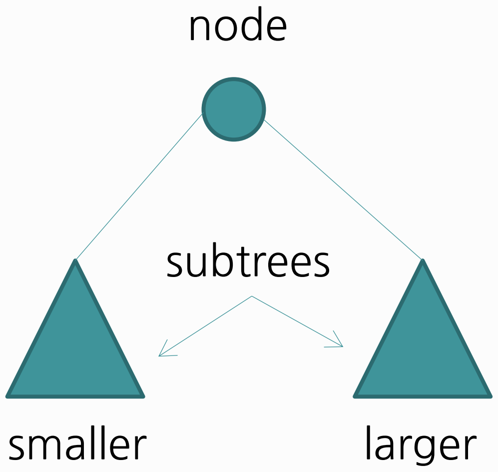
- Binary Search Tree
  - Binary Search Tree Properties
  - Binary Search Tree ADT
  - Node Class BST Class Constructors
  - Search & Insert

- Reference:
  - Problem Solving with Algorithms and Data Structures
  - Chapter 6 - Tree

Binary Search Tree: Definition

- **A binary tree (BT)** is a tree data structure in which each node has **at most two children**, which are referred to as the left child and the right child.
- **A binary search tree (BST)** is an ordered or sorted binary tree.
  - Each node in a binary search tree has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.)
  - Equal keys are ruled out.





Binary Search Tree: Example

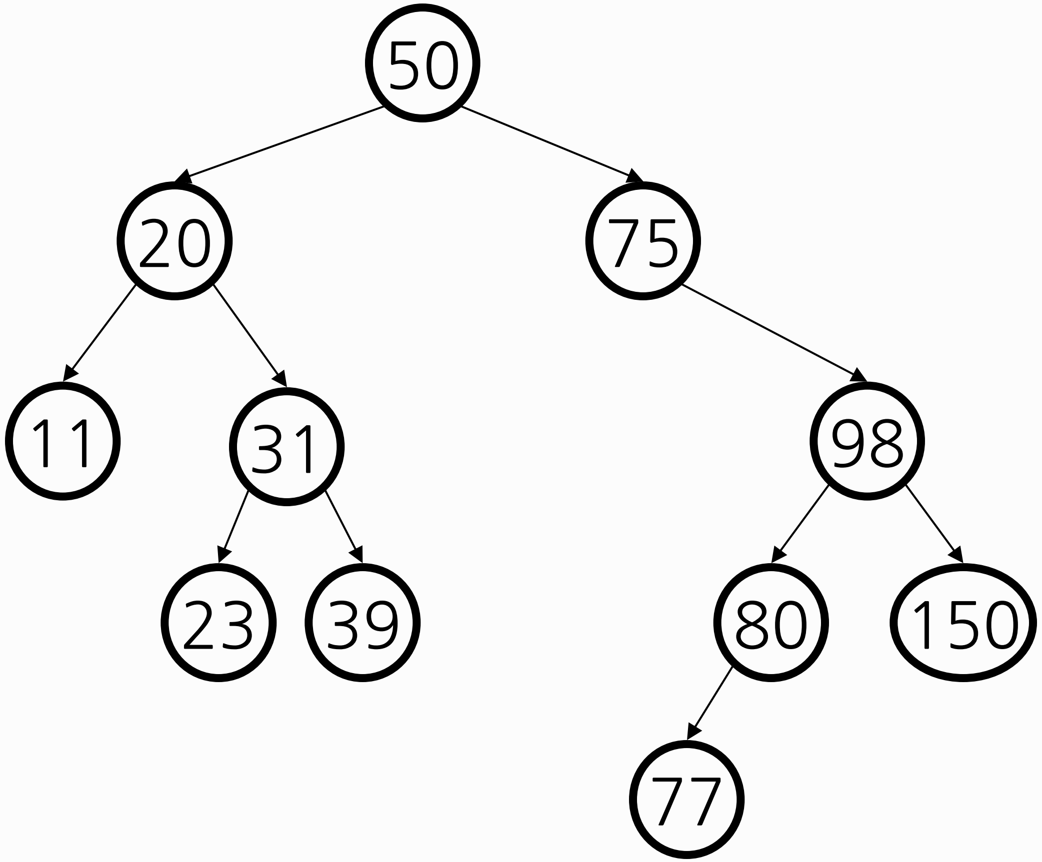
- **Insert -** Drawing a binary search tree as the following values are being added to an initially empty tree.

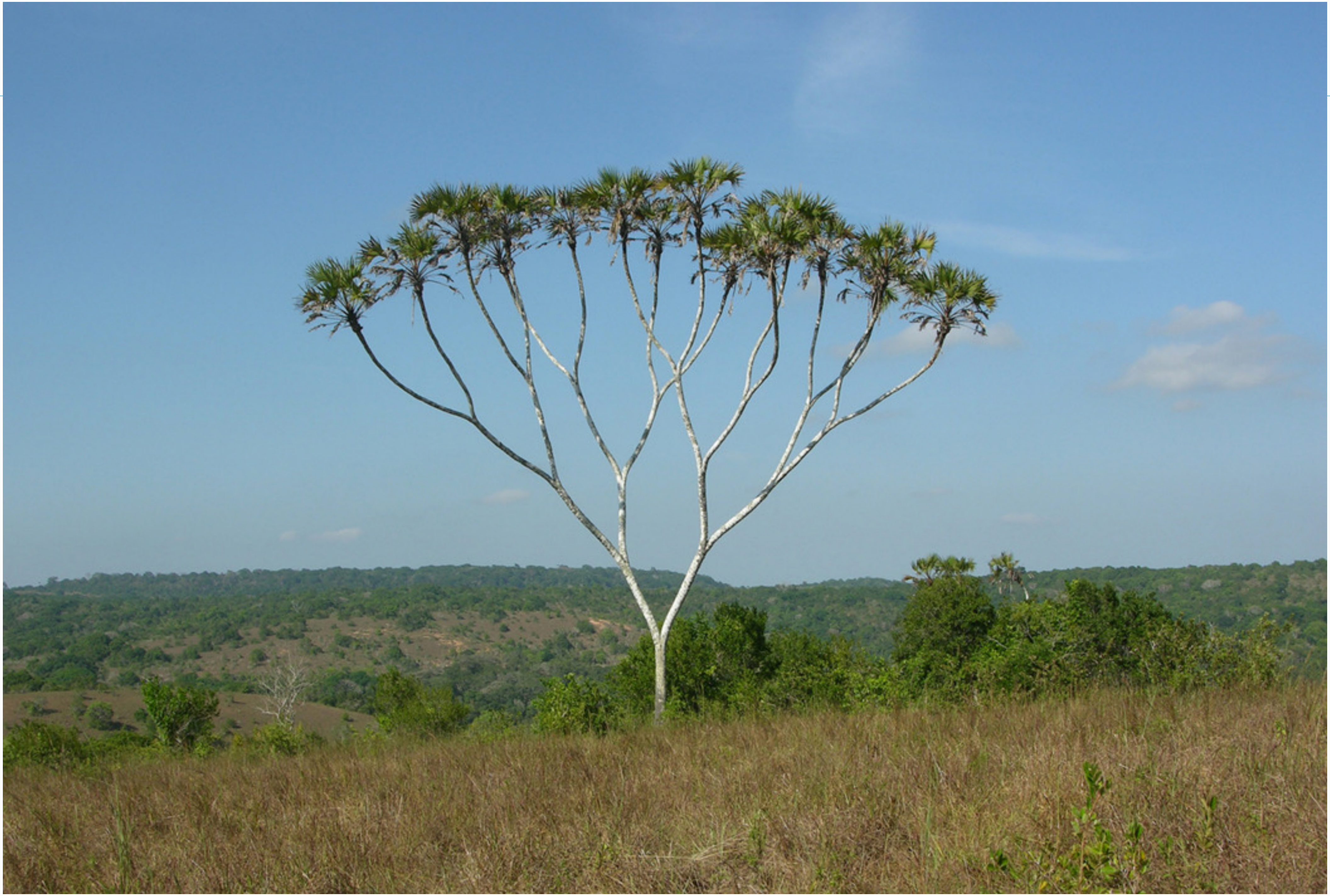
21
28
14
32
25
18
11
30
19
15

Binary Search Tree: Exercise

- **Insert -** Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77





Binary Search Tree: ADT

- get() - search, contains, size, height, min/max, successor, predecessor, distance
- add() - insert, grow
- delete() - delete, trim
- BT to BST conversion
- LCA - Lowest Common Ancestor

Binary Search Tree:

- ## Node and BST Class Constructors

```
class Node():
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right

    def __repr__(self):
        return "Node({})".format(self.key)

if __name__ == '__main__':
    root = Node(99)
    print(root)
```

```
Node(99)
```

Binary Search Tree:

- ## Node and BST Class Constructors

```
class Node():
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right

    def __repr__(self):
        return "Node({})".format(self.key)

if __name__ == '__main__':
    root = Node(99)
    print(root)
```

```
Node(99)
```

```
<__main__.Node object at 0x000002BCE8D22A00>
```

Binary Search Tree:

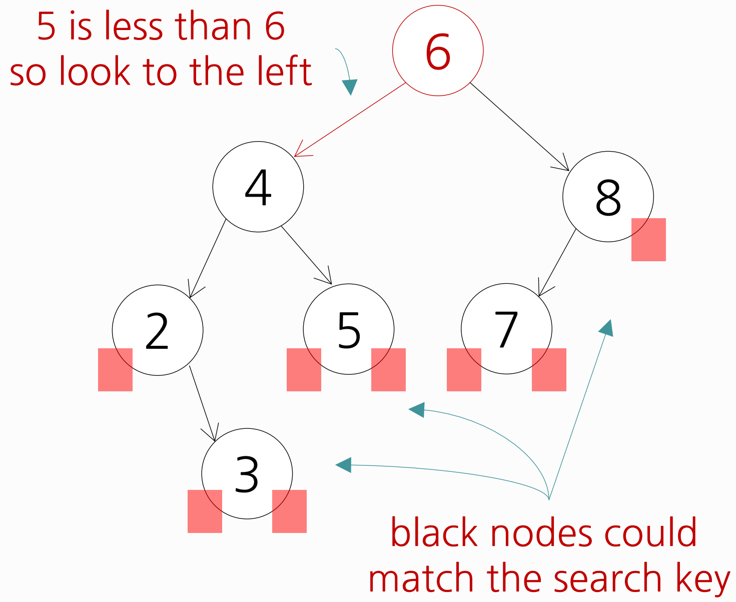
- ## Node and BST Class Constructors

```
class BST:
    def __init__(self, key=None, left=None, right=None):
        if key is None:
            self.root = None
        else:
            self.root = Node(key, left, right)

    def delTree(self):
        self.root = None   # gc will do this for us.

    def __str__(self):
        return self.root.__str__()

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

    def __len__(self):
        return self.root.__len__()

    def len(self):
        return self.__len__()
```

Binary Search Tree: Search

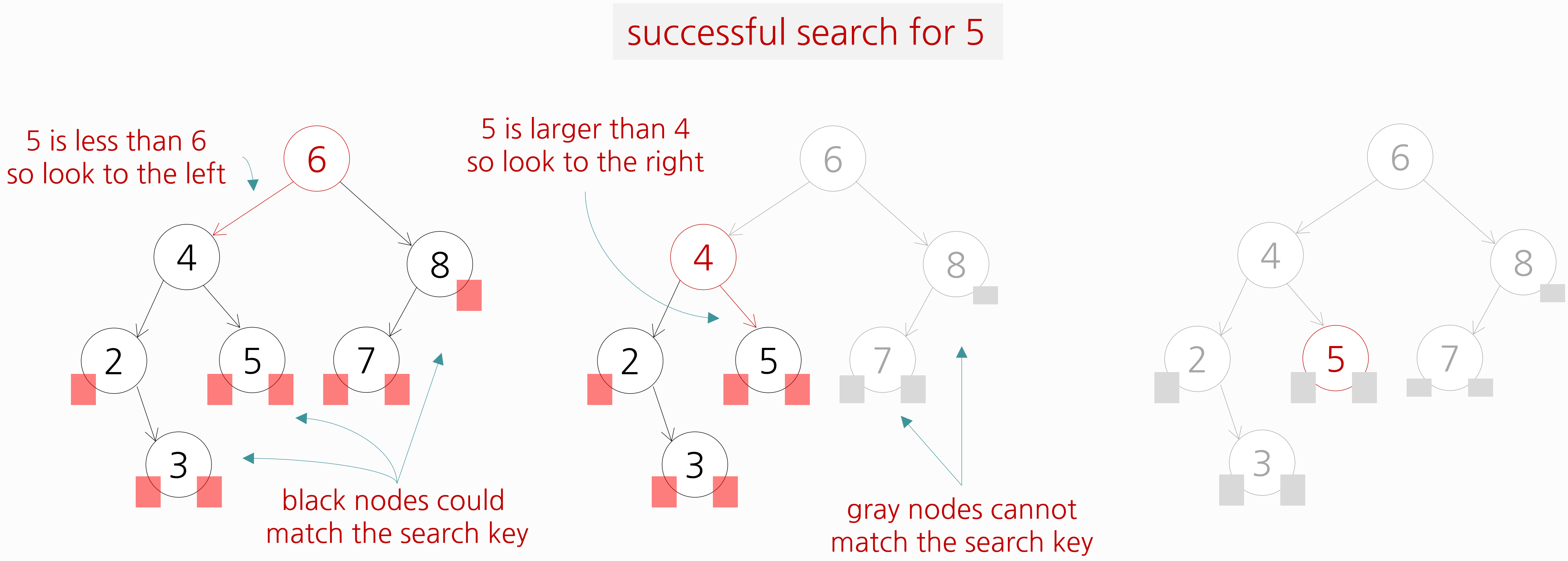
- A recursive algorithm to search for a key in a BST follows immediately from the recursive structure: If the tree is empty, we have a search miss; if the search key is equal to the key at the root, we have a search hit. Otherwise, we search (recursively) in the appropriate subtree.
  - **`get(root, key), contains(root, key), __getitem__(), __contains__(), contains()`**

successful search for 5



Binary Search Tree: Search

- A recursive algorithm to search for a key in a BST follows immediately from the recursive structure: If the tree is empty, we have a search miss; if the search key is equal to the key at the root, we have a search hit. Otherwise, we search (recursively) in the appropriate subtree.
  - **`get(root, key)`, `contains(root, key)`, `__getitem__()`, `__contains__()`, `contains()`**

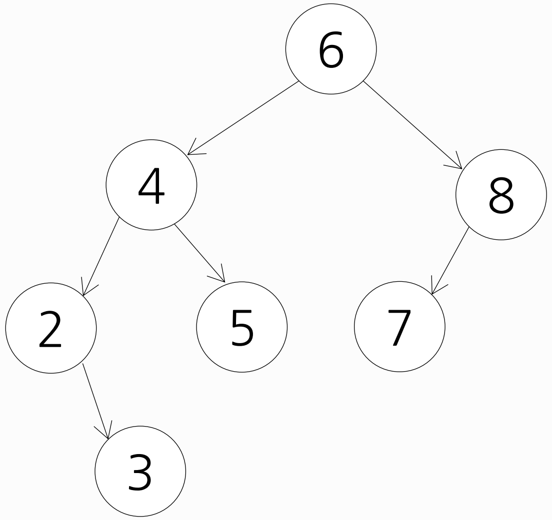


Binary Search Tree: get()

```
class BST:
    ...
    def get(self, key):
        return self._get(self.root, key)

    def _get(self, root, key):
        if root is None or root.key == key:
            return root is not None
        elif key < root.key:
            return _get(root.left, key)
        elif key > root.key:
            return _get(root.right, key)

    def __getitem__(self, key):
        pass

    def __contains__(self, key):
        pass

    def contains(self, key):
        pass
```

Binary Search Tree: get()

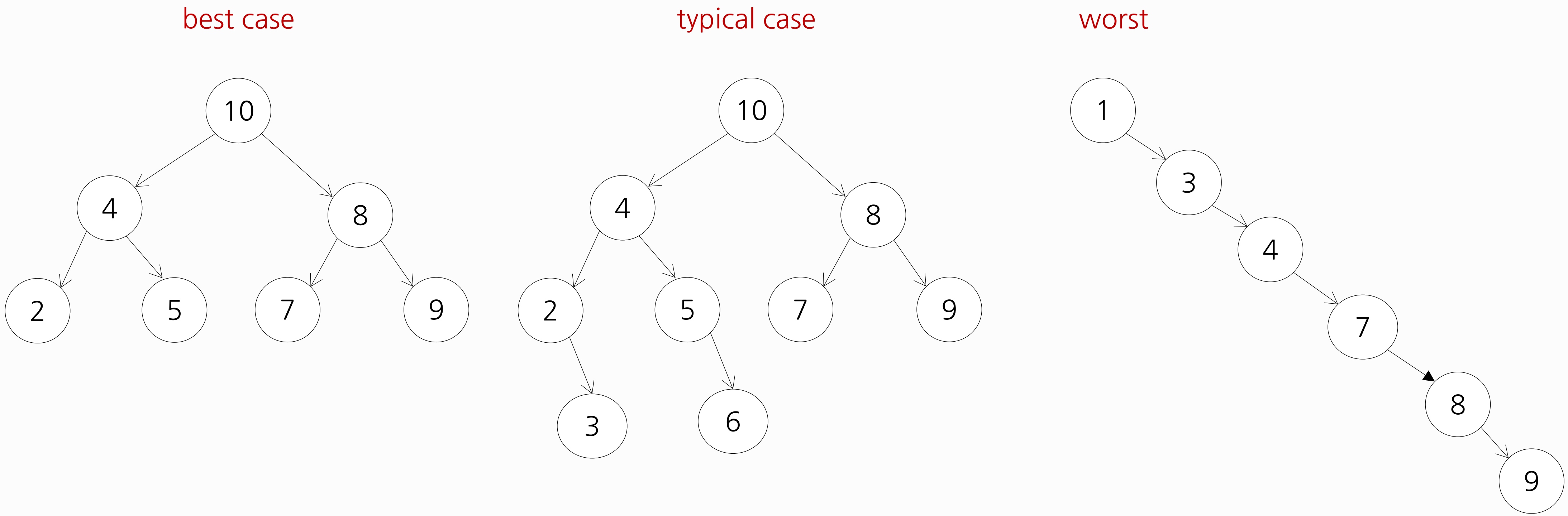
```
def _get(self, root, key):
    if root is None or root.key == key:
        return root is not None
    elif key < root.key:
        return _get(root.left, key)
    elif key > root.key:
        return _get(root.right, key)
```




```
def _get_iter(node, key)
    if node is None: return False
    while node is not None:
        if key == node.key: return True;
        if key  < node.key:
            node = node.left
        else:
            node = node.right
    return False
```

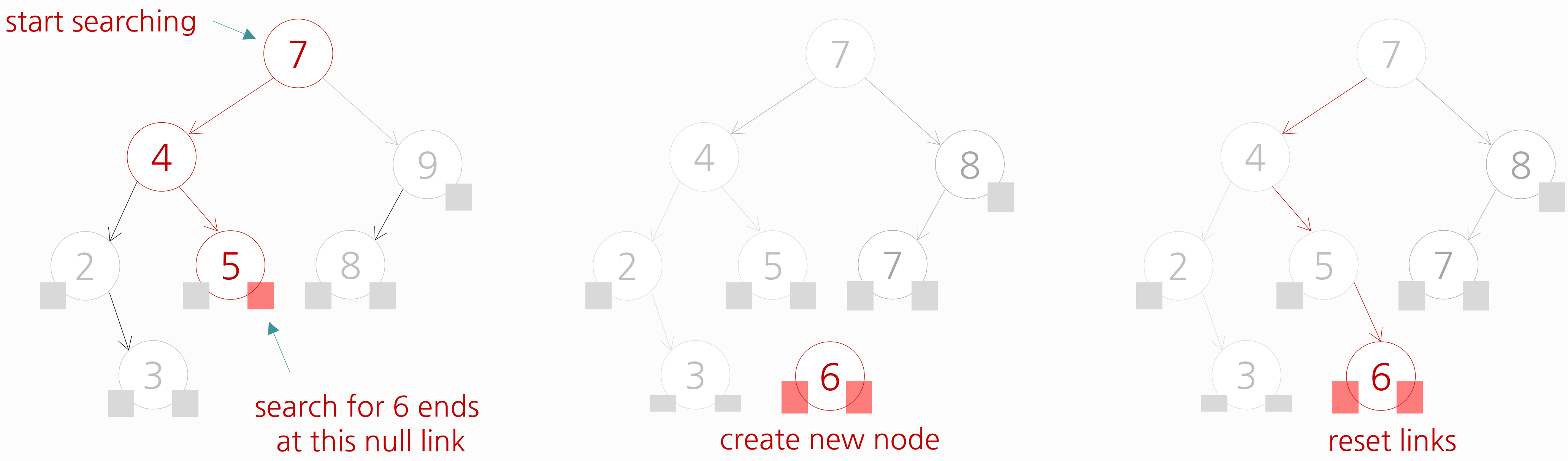
Binary Search Tree: Search Analysis

- The running times of algorithms on binary search trees depend on the shapes of the trees, which, in turn, depends on the order in which keys are inserted.
  - It is reasonable, for many applications, to use the following simple model: We assume that the keys are (uniformly) random, or, equivalently, that they are inserted in random order.
  - Insertion and search misses in a BST built from N random keys requires O(h), where h is the height of a BST. The typical case is ~ 1.39 $\log_2$ N compares on the average.



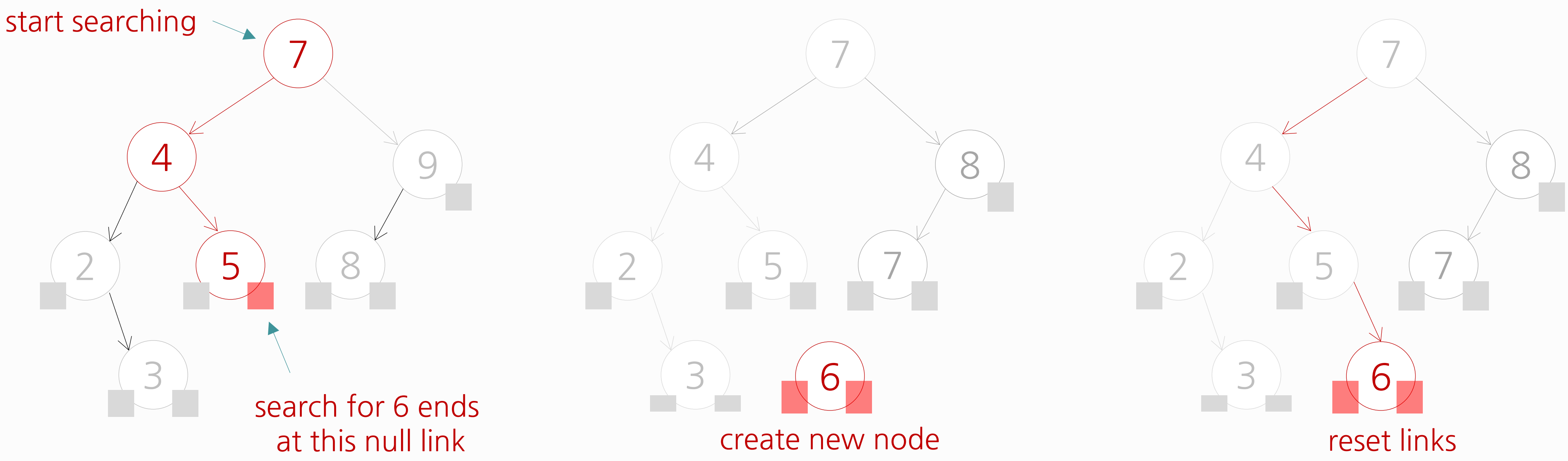
Binary Search Tree: add()

- Insertion operation is very similar to search. Indeed, a search for a key not in the tree ends at a null link, and all that we need to do is replace that link with a new node containing the key. .
  - **`add(root, 6)`**



Binary Search Tree: add()

```
def _add(self, node, key):
    if node is None:
        node = Node(key)
    else:
        if key < node.key:
            node.left = self._add(node.left, key)
        elif key > node.key:
            node.right = self._add(node.right, key)
        else:
            pass
    return node
```

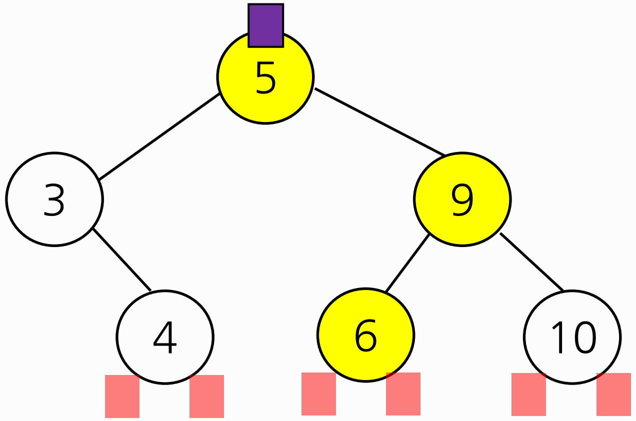


Binary Search Tree: add()

- _add(self, node, key) - Insert a node with key
  - **Step 1**: If the tree is empty, return a new **Node(key)**.
  - **Step 2**: Pretending to search for key in BST, until locating a None.
  - **Step 3**: create a new node(key) and link it.

```
def _add(self, node, key):
    if node is None:
        node = Node(key)
    else:
        if key < node.key:
            node.left = self._add(node.left, key)
        elif key > node.key:
            node.right = self._add(node.right, key)
        else:
            pass
    return node
```

`_add(node, 7)`



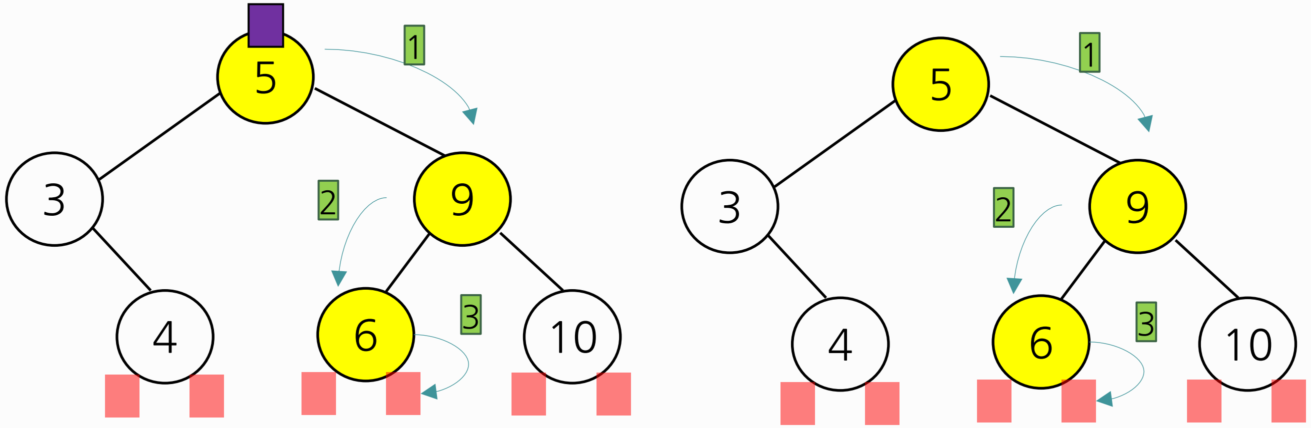
Binary Search Tree: add()

- _add(self, node, key) - Insert a node with key
  - **Step 1**: If the tree is empty, return a new **Node(key)**.
  - **Step 2**: Pretending to search for key in BST, until locating a None.
  - **Step 3**: create a new node(key) and link it.

```
def _add(self, node, key):
    if node is None:
        node = Node(key)
    else:
        if key < node.key:
            node.left = self._add(node.left, key)
        elif key > node.key:
            node.right = self._add(node.right, key)
        else:
            pass
    return node
```

`_add(node, 7)`

The highlight nodes are compared with key 7.



after node(6) & key(7) compared, it calls **_add(None, 7)** 3

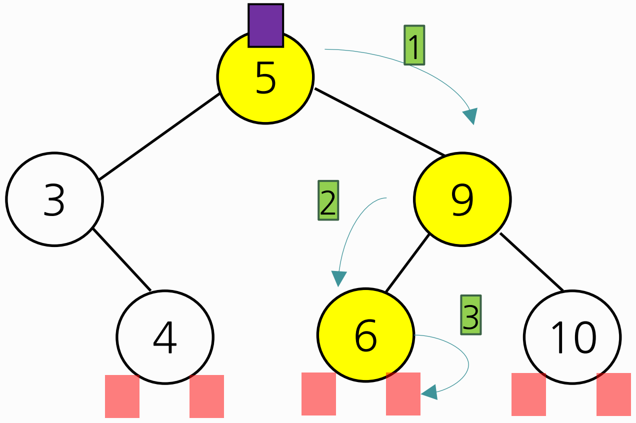
Binary Search Tree: add()

- _add(self, node, key) - Insert a node with key
  - **Step 1**: If the tree is empty, return a new **Node(key)**.
  - **Step 2**: Pretending to search for key in BST, until locating a None.
  - **Step 3**: create a new node(key) and link it.

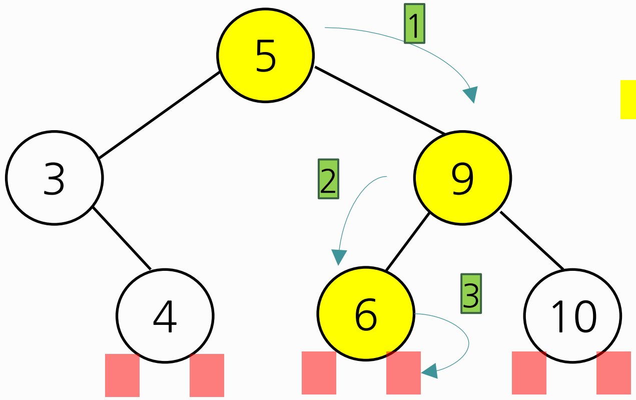
```
def _add(self, node, key):
    if node is None:
        node = Node(key)
    else:
        if key < node.key:
            node.left = self._add(node.left, key)
        elif key > node.key:
            node.right = self._add(node.right, key)
        else:
            pass
    return node
```

`_add(node, 7)`

The highlight nodes are compared with key 7.

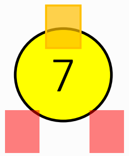


after node(6) & key(7) compared, it calls **_add(None, 7)** 3



Where does it return to?

= Node(7)



Where does it return to?
6 and 7 are **not** linked.

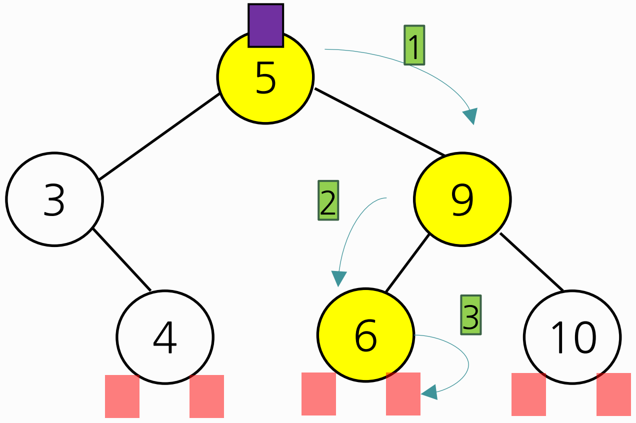
Binary Search Tree: add()

- _add(self, node, key) - Insert a node with key
  - **Step 1**: If the tree is empty, return a new **Node(key)**.
  - **Step 2**: Pretending to search for key in BST, until locating a None.
  - **Step 3**: create a new node(key) and link it.

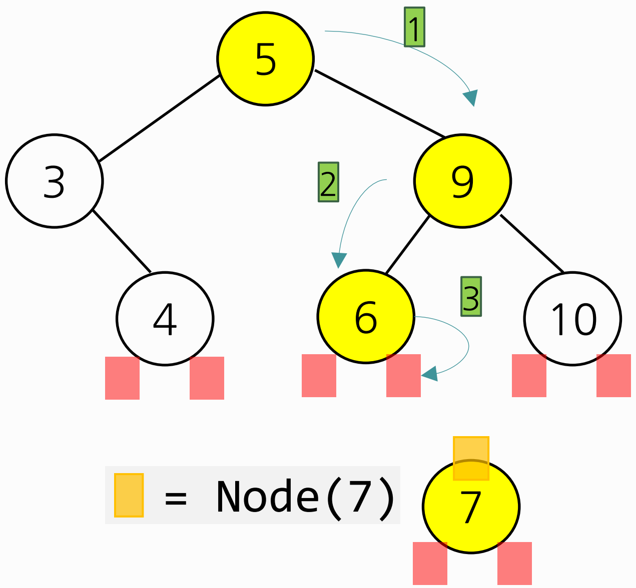
```
def _add(self, node, key):
    if node is None:
        node = Node(key)
    else:
        if key < node.key:
            node.left = self._add(node.left, key)
        elif key > node.key:
            node.right = self._add(node.right, key)
        else:
            pass
    return node
```

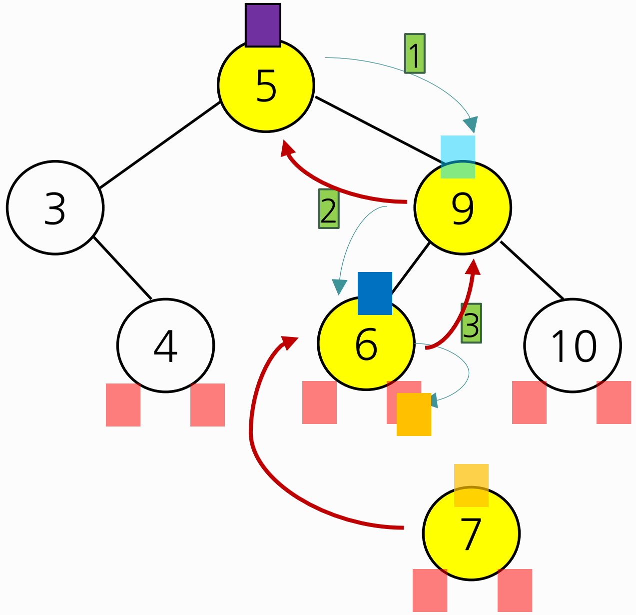
`_add(node, 7)`

The highlight nodes are compared with key 7.



after node(6) & key(7) compared, it calls **_add(None, 7)** 3



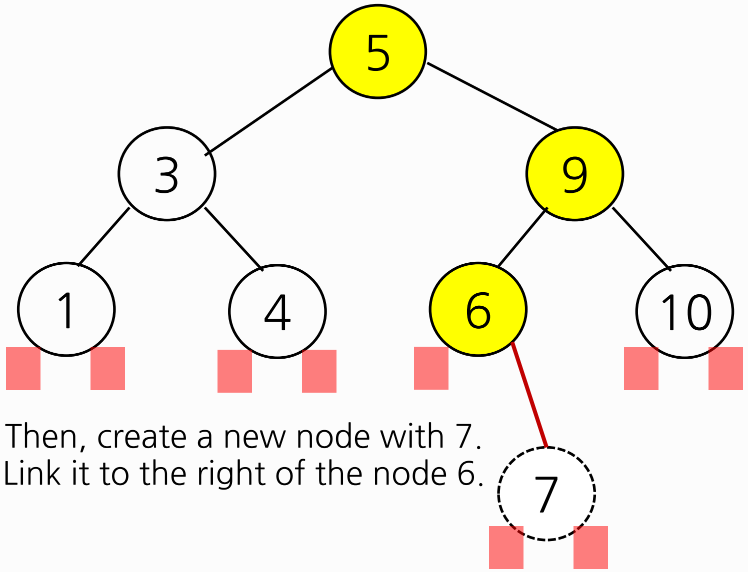


Binary Search Tree: add()

Questions:

1. Explain the differences between the binary tree and binary search tree in this operation.
2. To complete inserting **7**, how many times was **`_add()`** called?
3. How many times " **`if key < node->key`**" called during this process or inserting 7?
4. At the end of this whole process, which **`return`** will be executed and what is the key value of the node?

```
def _add(self, node, key):
    if node is None:
        node = Node(key)
    else:
        if key < node.key:
            node.left = self._add(node.left, key)
        elif key > node.key:
            node.right = self._add(node.right, key)
        else:
            pass
    return node
```



Then, create a new node with 7.
Link it to the right of the node 6.

학습 정리

1) BST의 ADT에는 get(), add(), delete() 등이 있다

2) 균형잡힌 트리에 대한 작업의 시간복잡도는 O($\log n$)이지만,
불균형을 이룬 트리는 최대 O(n)까지 증가할 수 있다

파이썬으로 배우는
데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니
다