Binary Search Trees

Mustafa Daraghmeh

Outline

- Introduction
- Binary Search Tree (BST) Concepts
- BST Insertion Operation
- BST Search Operation
- Practical Exercises
- Summary
- Practice Questions
- Q&A

Overview and Objectives

- In this lecture, you will learn:
  - Understand binary and binary search trees.
  - Perform insertion and search operations.

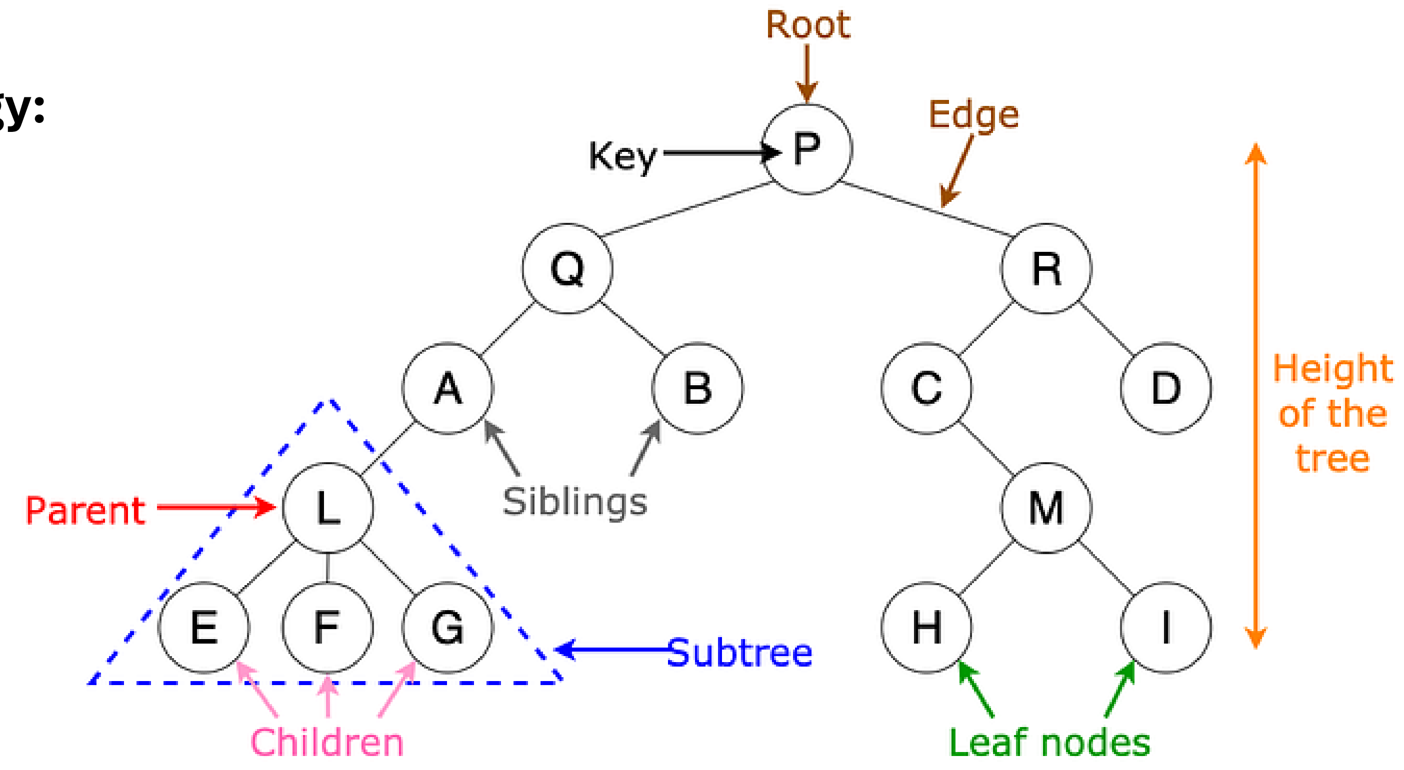
Motivation

- **Limitations of Lists**
  - Linear search in lists requires **O(n)** time in the worst case.
- **Demand for Efficient Search**
  - Applications require faster **lookup, insertion**, and **deletion** operations.
- **Why Trees?**
  - Trees organize data **hierarchically**, enabling efficient access.
  - Balanced binary search trees can achieve **O(log n)** time complexity.

What is a Tree?

- A **tree** is a hierarchical data structure consisting of nodes connected by edges.
- It represents relationships in a non-linear form.

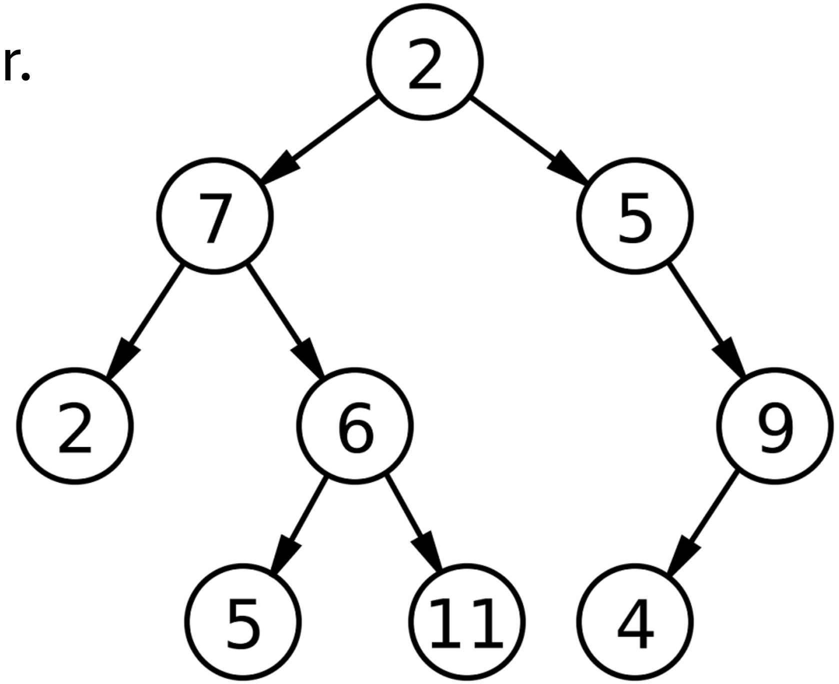
- **Basic Terminology:**



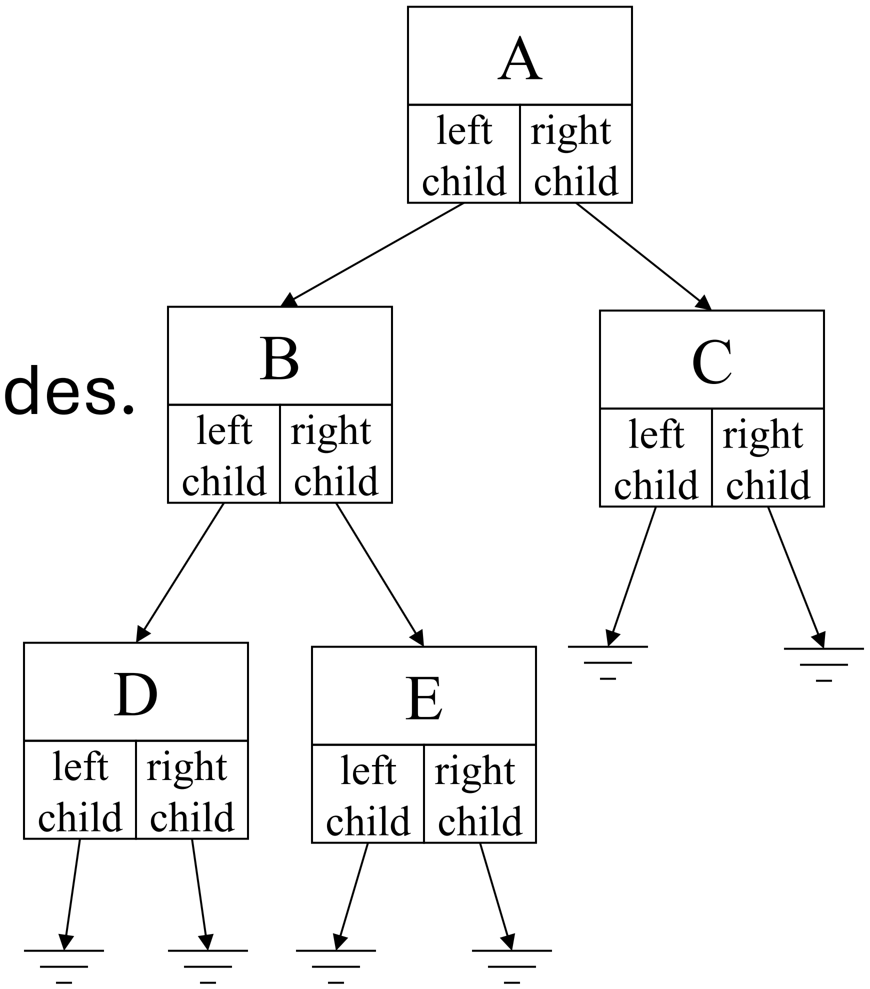
Binary Trees

- **Binary Tree:**
  - A special type of tree where **each node has at most two children**, called the **left** and **right** child.
  - It has no specific data order.

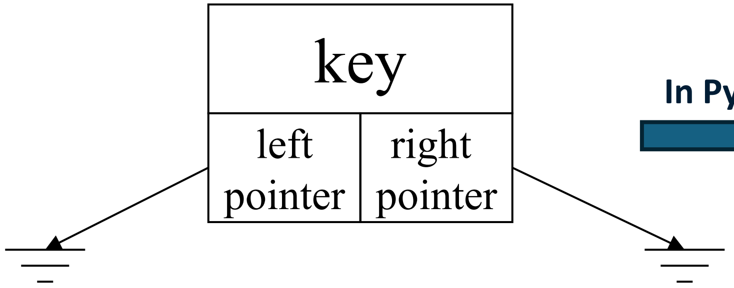
Is this a binary tree?



Binary Tree Representation



- Represented by a linked data structure of nodes.
- Each node contains fields:
  - **key**: Represents the node value.
  - **left**: Pointer to left child (*maybe empty*).
    - *Root of the left subtree.*
  - **right**: Pointer to the right child (*maybe empty*)
    - *Root of the right subtree.*



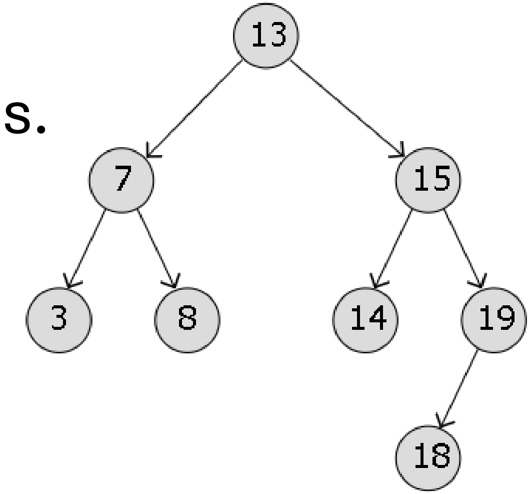
In Python

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

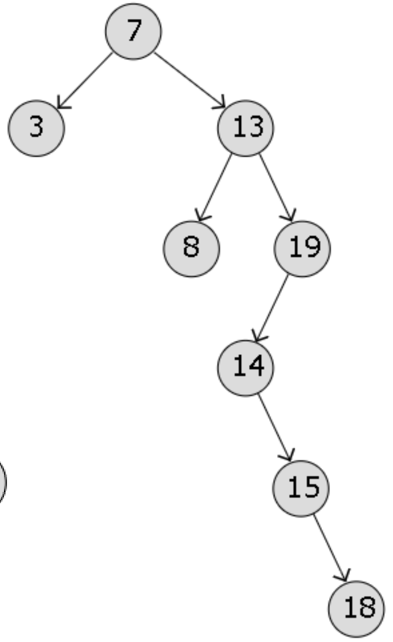
Binary Search Trees

- A **Binary Search Tree (BST)** is a binary tree where each node satisfies:
  - All values in the **left subtree** are **less than** the node's value.
  - All values in the **right subtree** are **equal to or greater than** the node's value.
  - **Recursive structure**: A subtree of a BST is also a BST.

- **Advantages:**
  - Efficient search, insertion, and deletion operations.
  - Time complexity:
    - Proportional to the **height of the tree – O(h)**.
    - Best/Average: **O(log n)** (**balanced**)
    - Worst: **O(n)** (**unbalanced**)



Balanced BST



Unbalanced BST

Why Study Binary Search Trees?

- **Foundation for Advanced Structures**
  - Forms the basis for more complex data structures like **AVL trees**, **Red-Black trees**, and **others**.
- **Real-World Applications**
  - Used in: **Databases** (e.g., indexing), **Compilers** (syntax trees), and **File systems** (hierarchical structure)
- **Supports Ordered Data**
  - Enables **in-order traversal** for producing sorted sequences without additional memory or sorting operations.

Insertion Operation

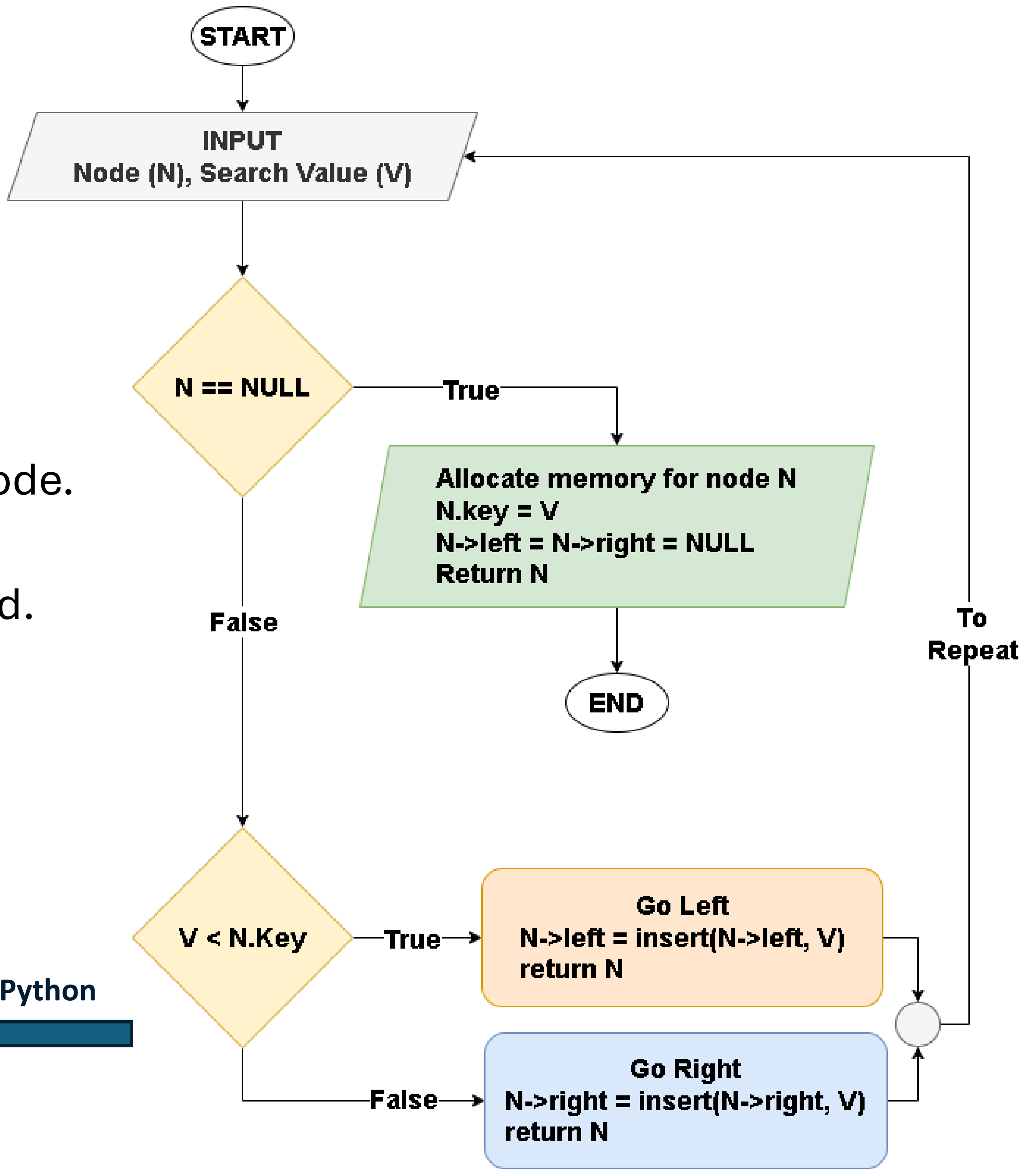
- **Objective**: To add a new element with a specific key into the binary search tree while maintaining its ordered properties.
  - All values in the left subtree are strictly less than the current node's key.
  - All values in the right subtree are greater than or equal to the current node's key.
- **Core Principle:** A new key value will always be added at a leaf node.

Insertion Operation

- **Steps to Insert a New Node:**
  1. Start at the root.
  2. Compare the new value with the current node.
  3. If smaller, go left; if larger, go right.
  4. Repeat until an empty child pointer is found.
  5. Insert the new node there.

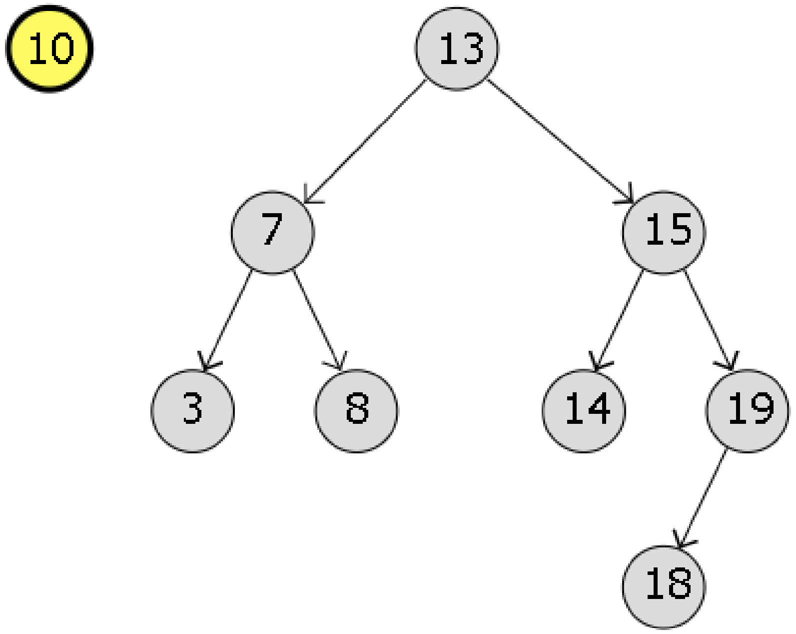
```python
def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.val:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root
```

In Python

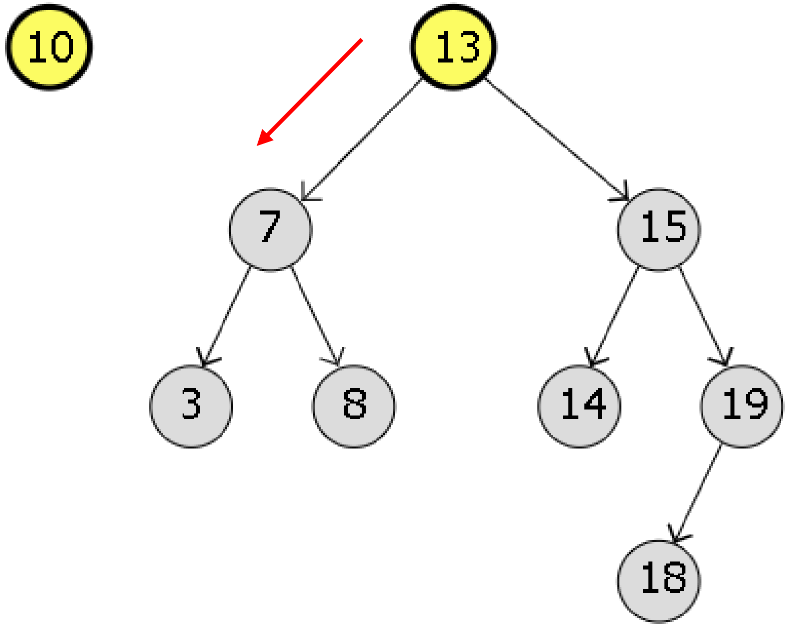


Insertion Operation – Example 1

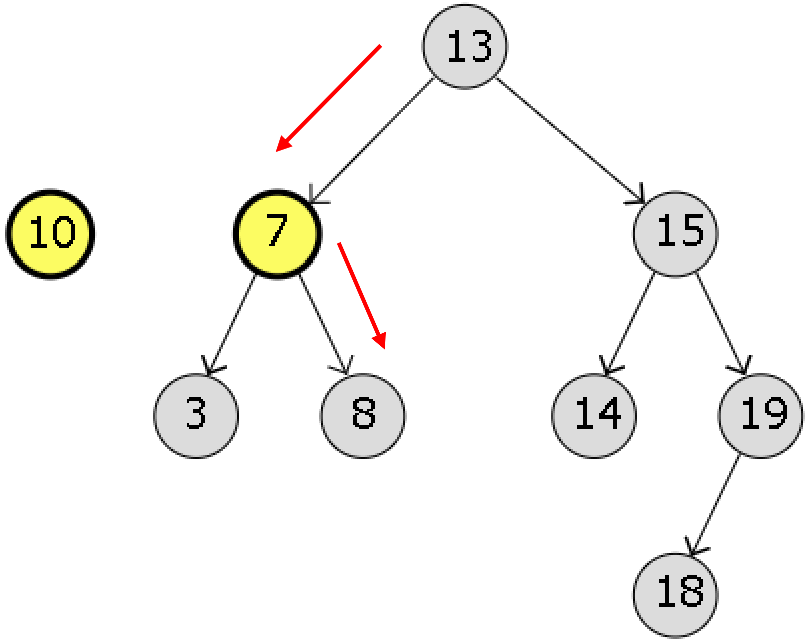
Inserting node 10



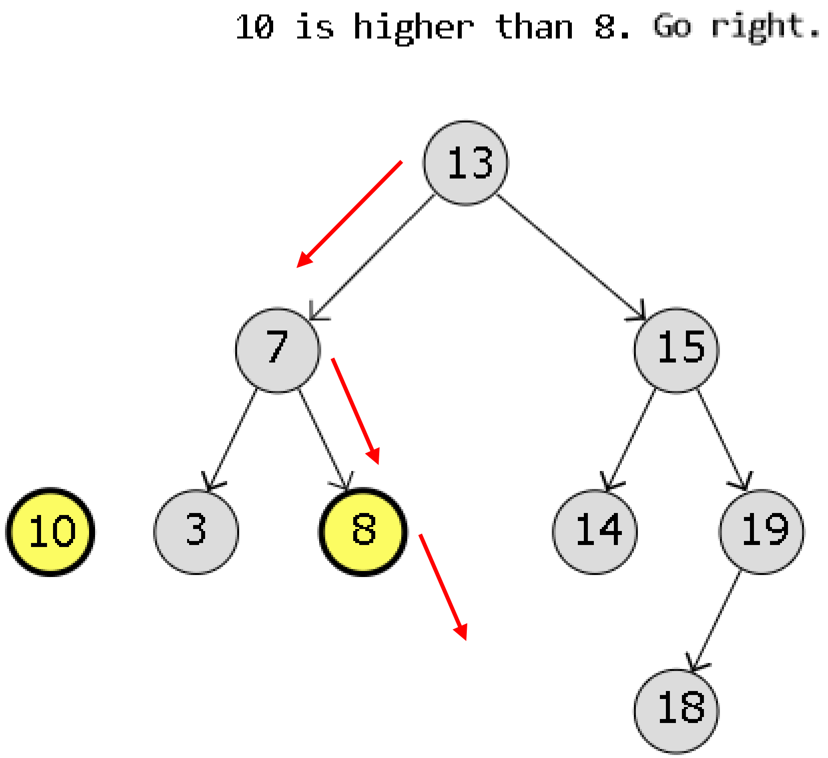
10 is lower than 13. Go left.

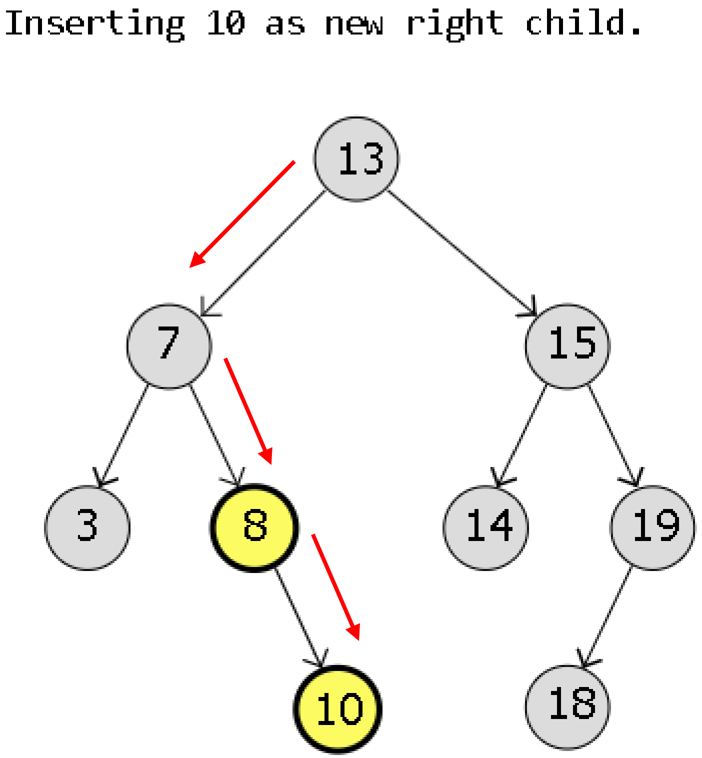


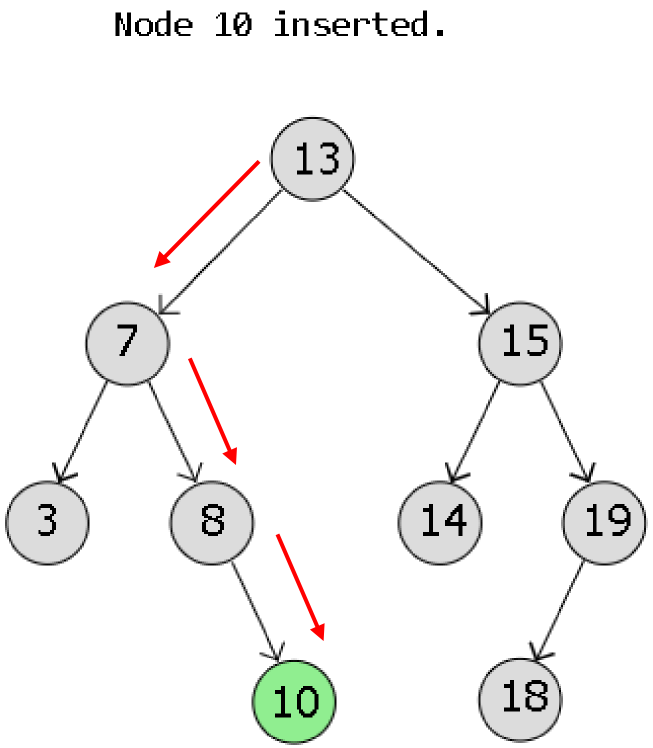
10 is higher than 7. Go right.



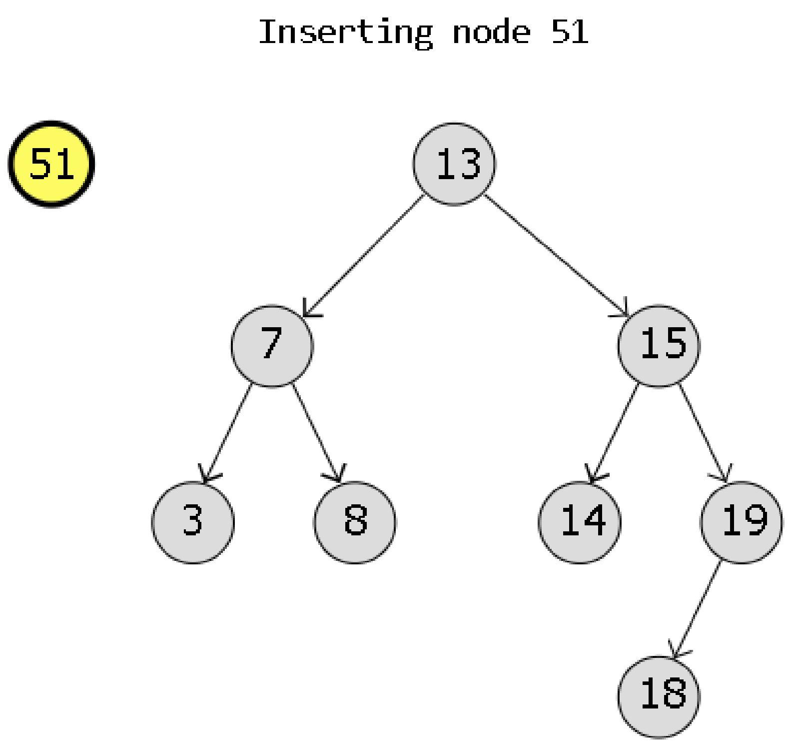
Insertion Operation – Example 1 (Cont.)

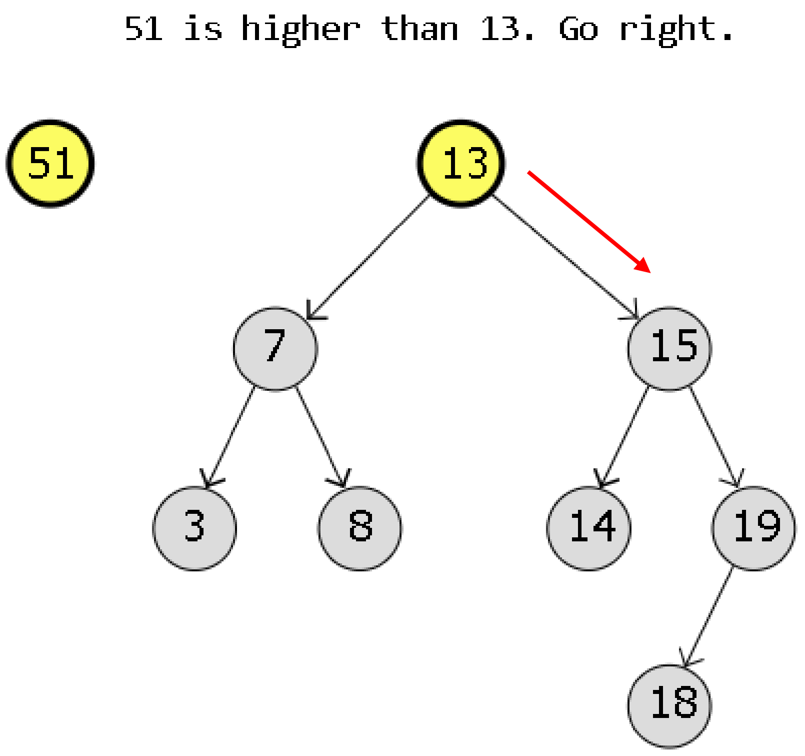


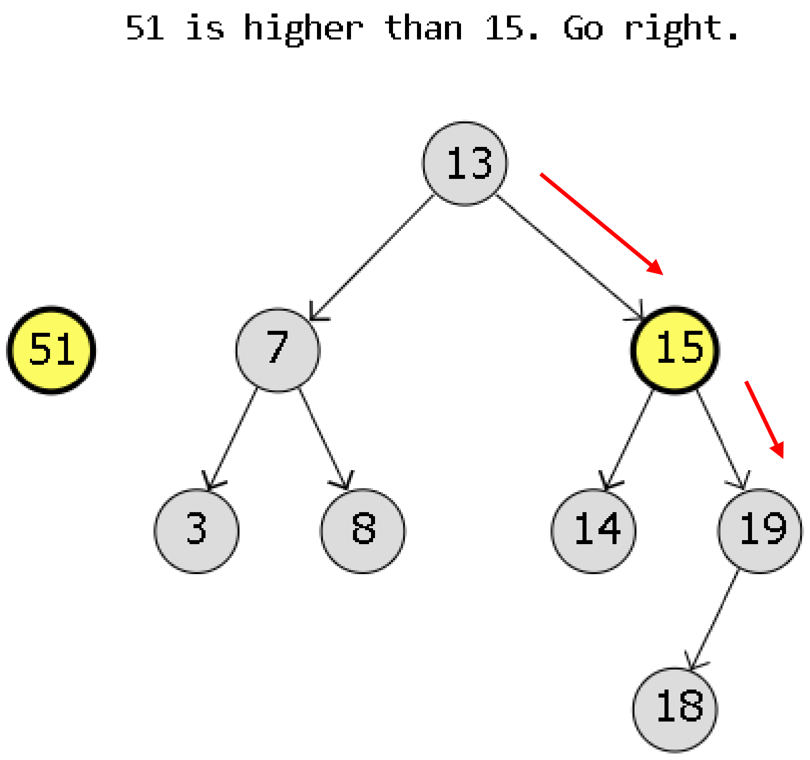




Insertion Operation – Example 2

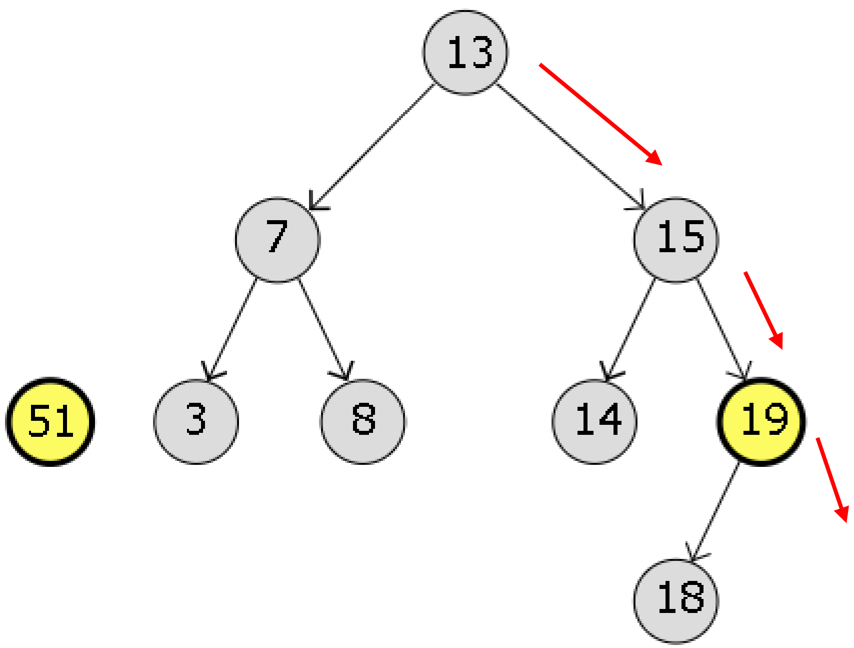




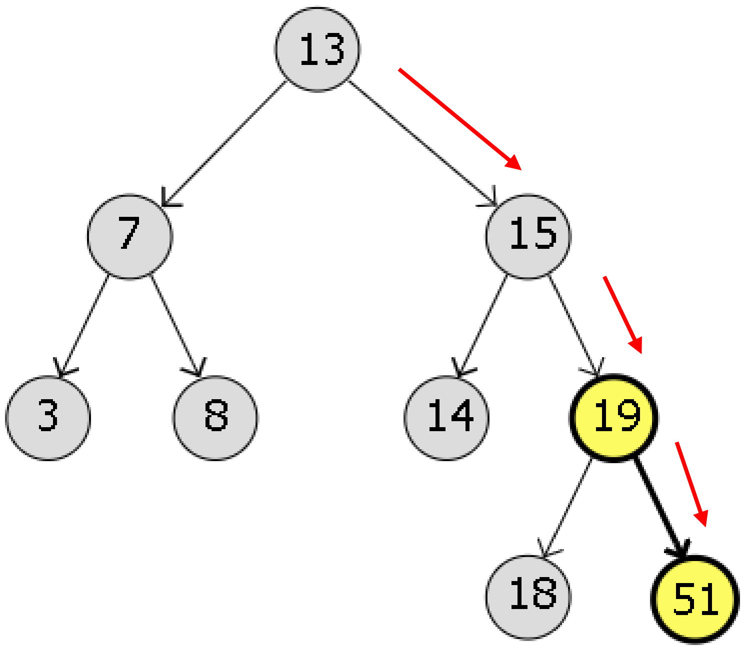


Insertion Operation – Example 2 (Cont.)

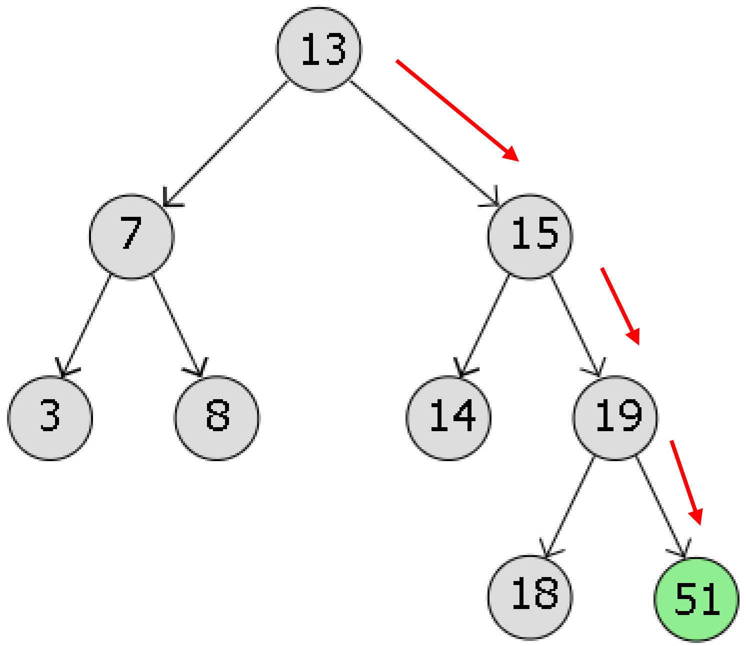
51 is higher than 19. Go right.



Inserting 51 as new right child.



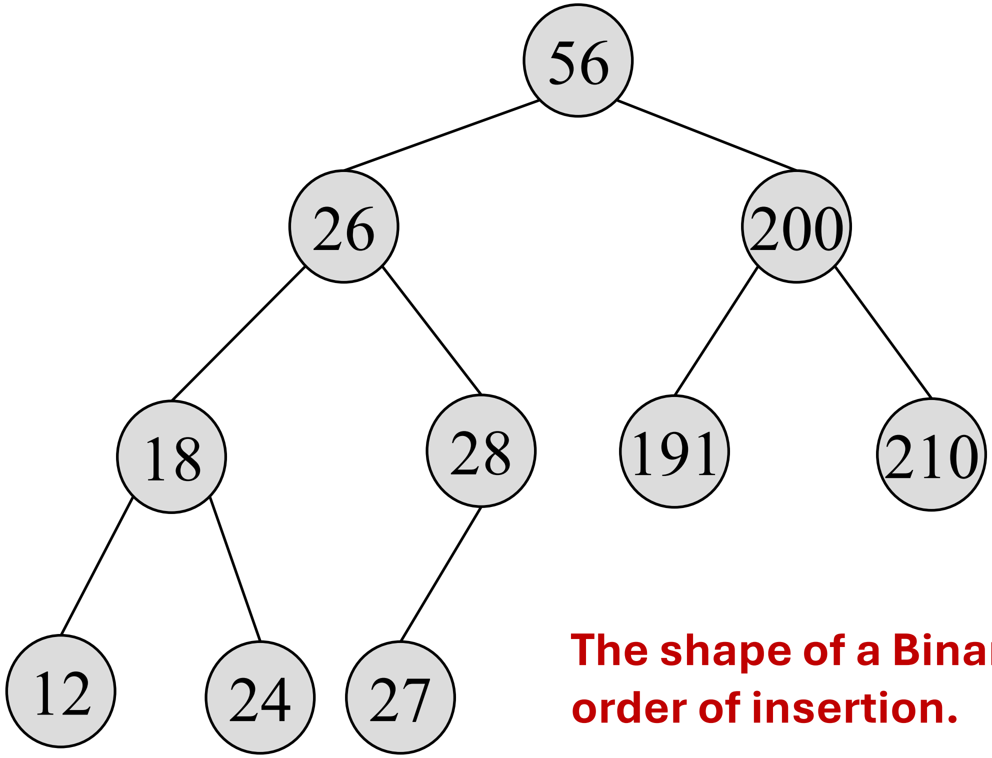
Node 51 inserted.



Insertion Operation – Example 3

- Construct a BST using the given keys:

56 26 200 18 28 12 24 27 191 210



The shape of a Binary Search Tree depends entirely on the order of insertion.

Search Operation

- **Objective**:
  - To retrieve and determine whether a specific value exists in a binary search tree or not.

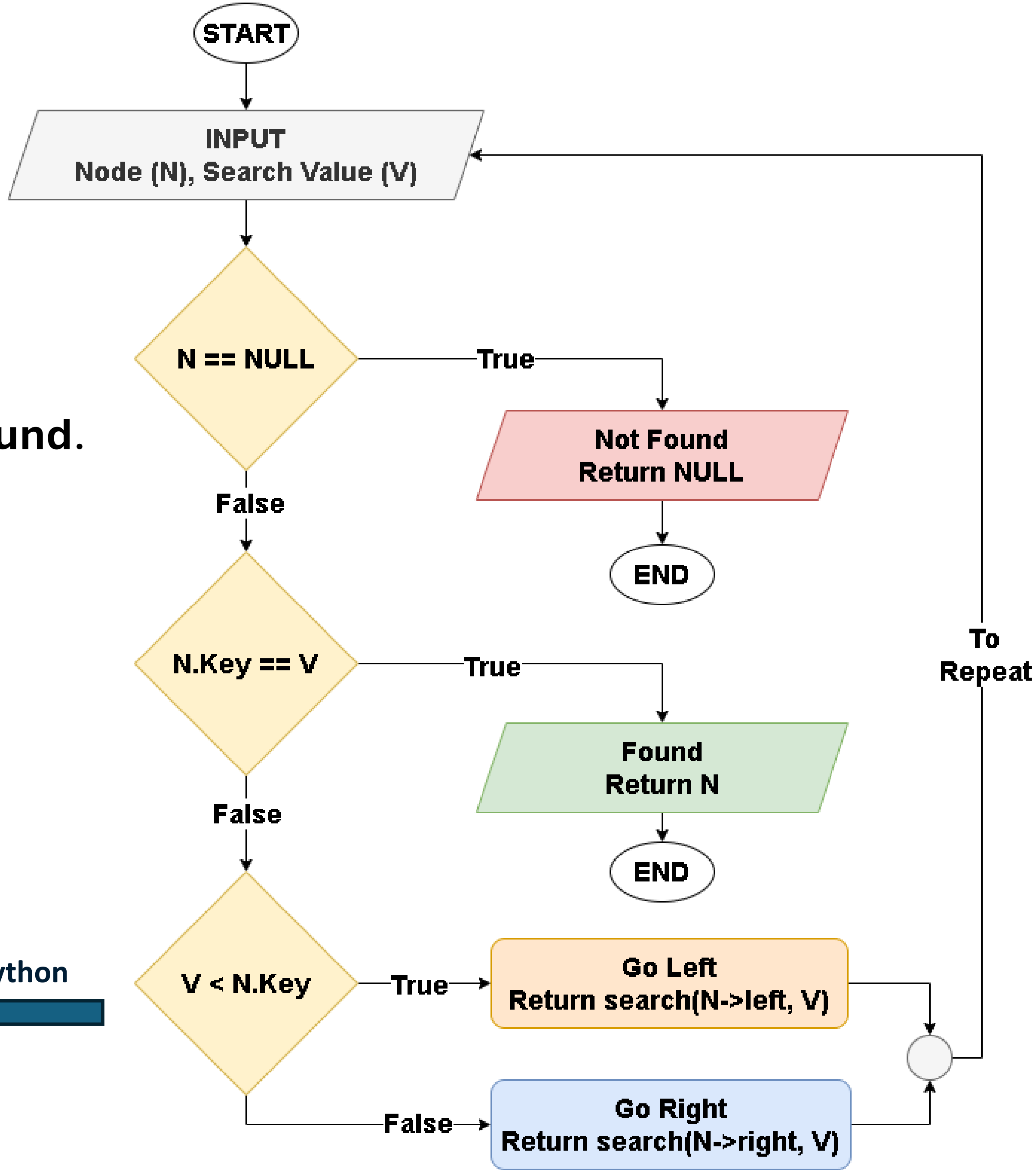
- **Core Principle:**
  - The search operation utilizes the binary search property to eliminate half of the remaining nodes, if balanced, at each comparison step.
  - This leads to an efficient search time of O(h), where h is the height of the tree.

Search Operation

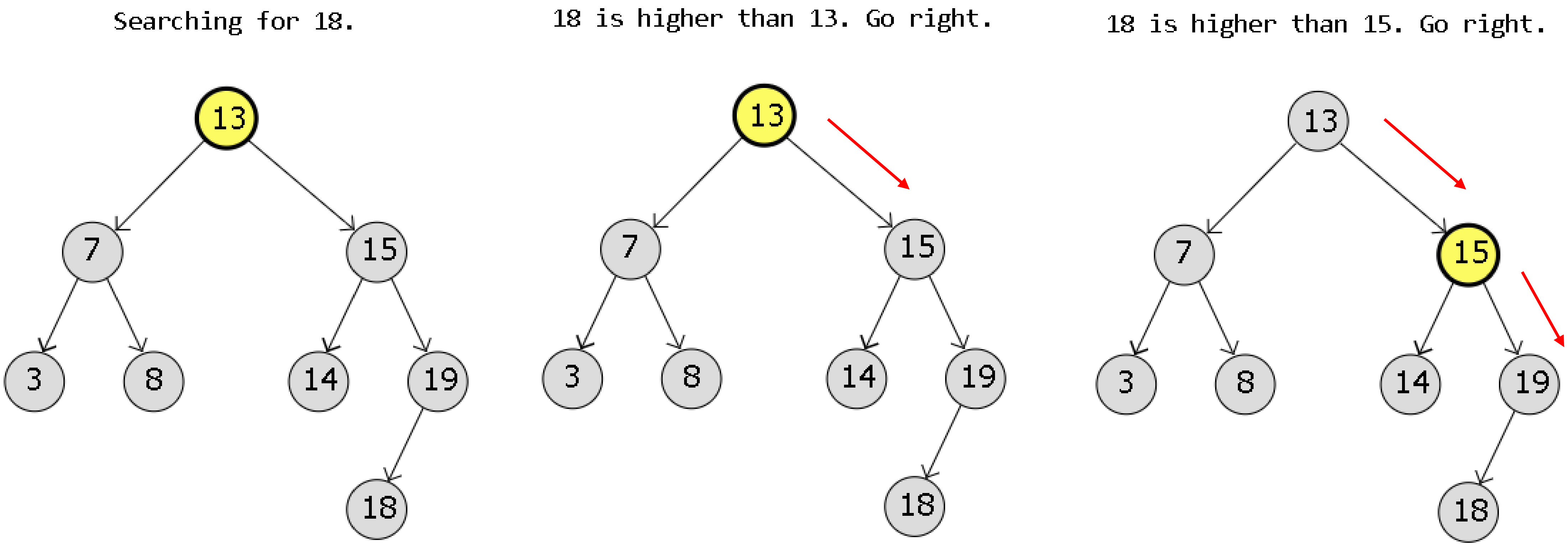
- **Steps to Search in a BST:**
    1. Start at the root.
    2. If the key equals the current node's value → **found**.
    3. If the key is less → go to the **left child**.
    4. If the key is greater → go to the **right child**.
    5. Repeat until found or null (not found).

```python
def search(root, key):
    if root is None or root.val == key:
        return root
    if key < root.val:
        return search(root.left, key)
    return search(root.right, key)
```

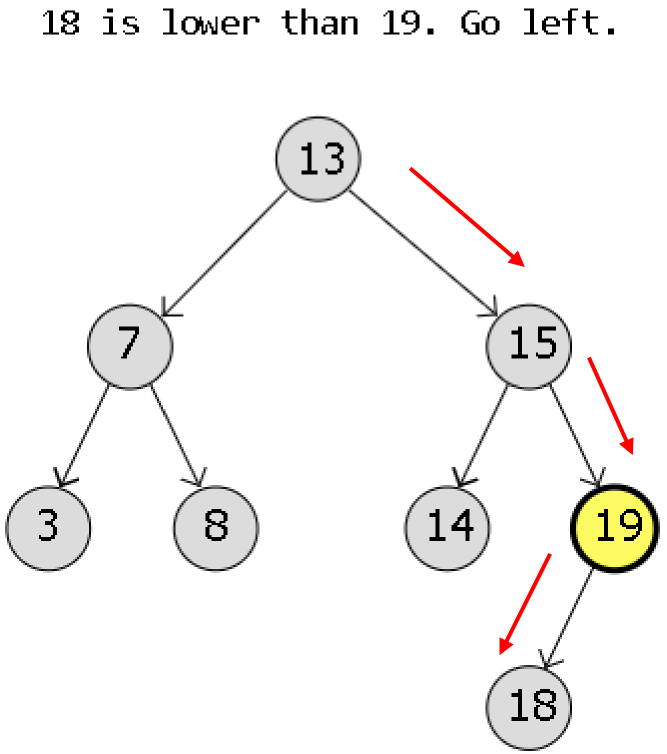
In Python

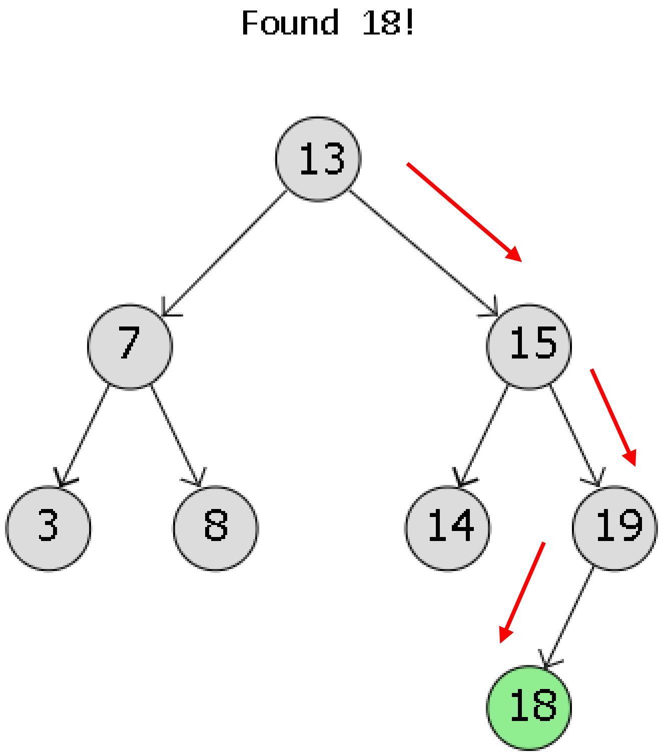


Search Operation – Example 1



Search Operation – Example 1 (Cont.)

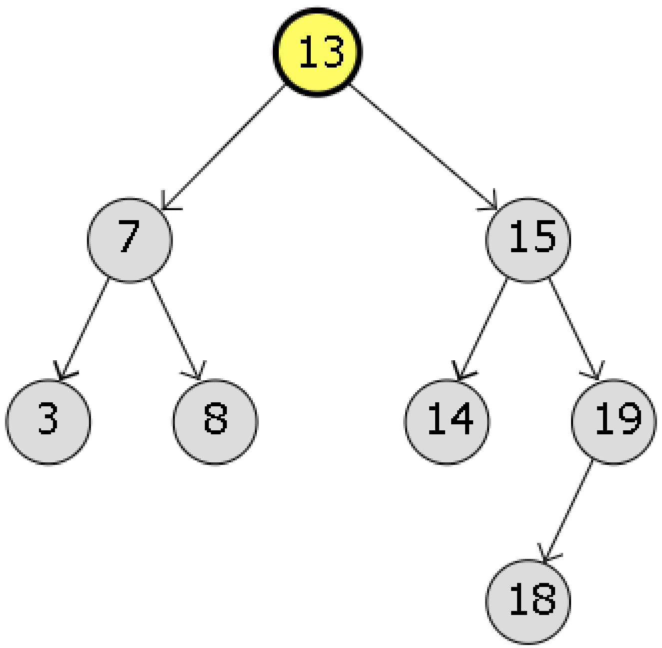




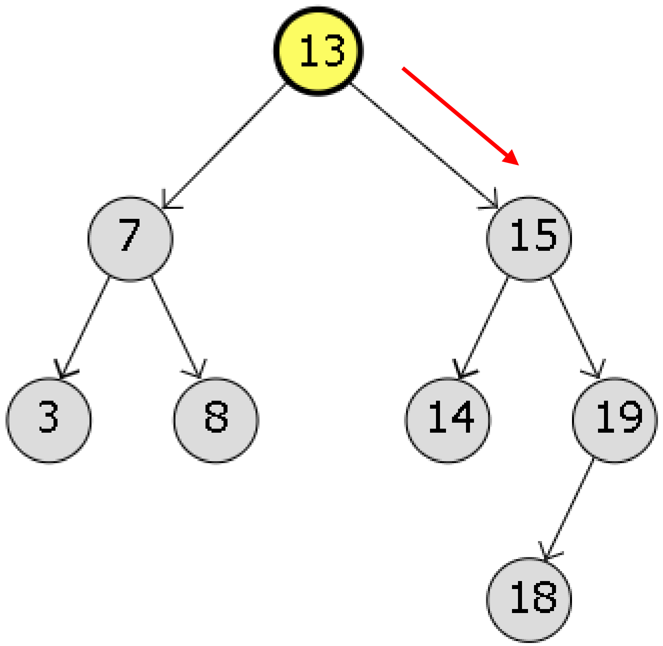
Path: 13 → 15 → 19 → 18.

Search Operation – Example 2

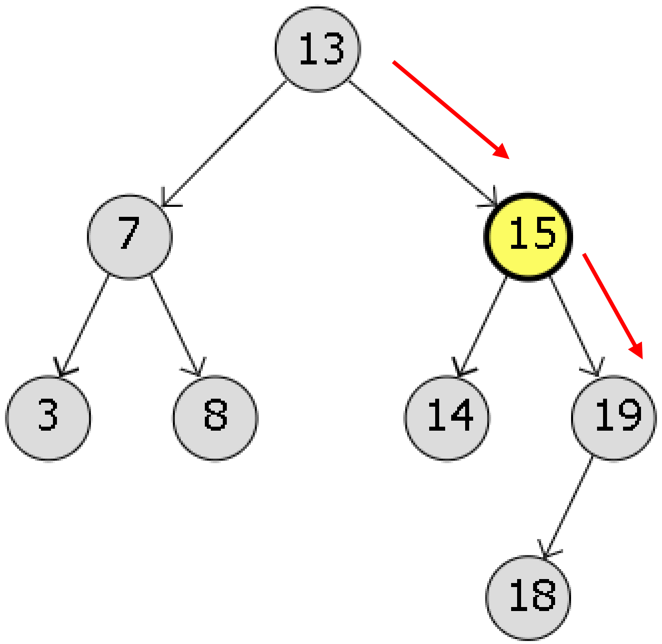
Searching for 51.



51 is higher than 13. Go right.

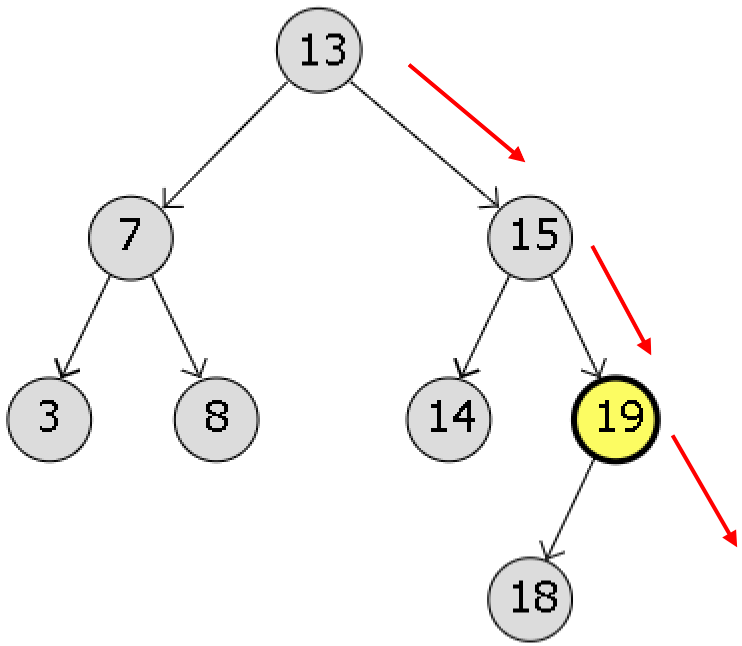


51 is higher than 15. Go right.

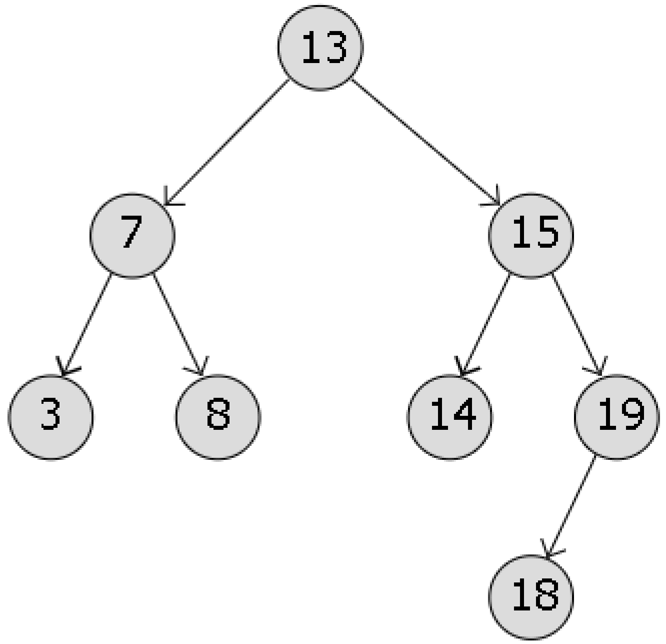


Search Operation – Example 2 (Cont.)

51 is higher than 19. Go right.



51 is not found!



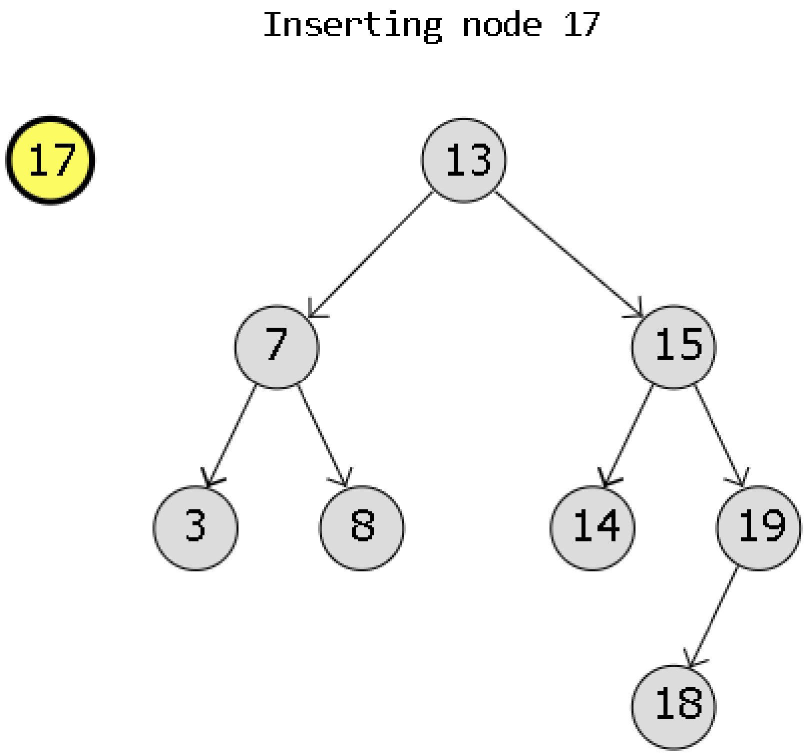
Summary

- BSTs are a fundamental data structure in computer science that organizes data hierarchically based on a defined order.
- Their primary advantage lies in the efficiency of search, insertion, and deletion operations, which are O(log n) if balanced

- Key takeaways:
  - BSTs maintain sorted data with structured access.
  - Structure degrades in worst-case (O(n)) if unbalanced.
  - Foundation for self-balancing trees (AVL, Red-Black Trees).

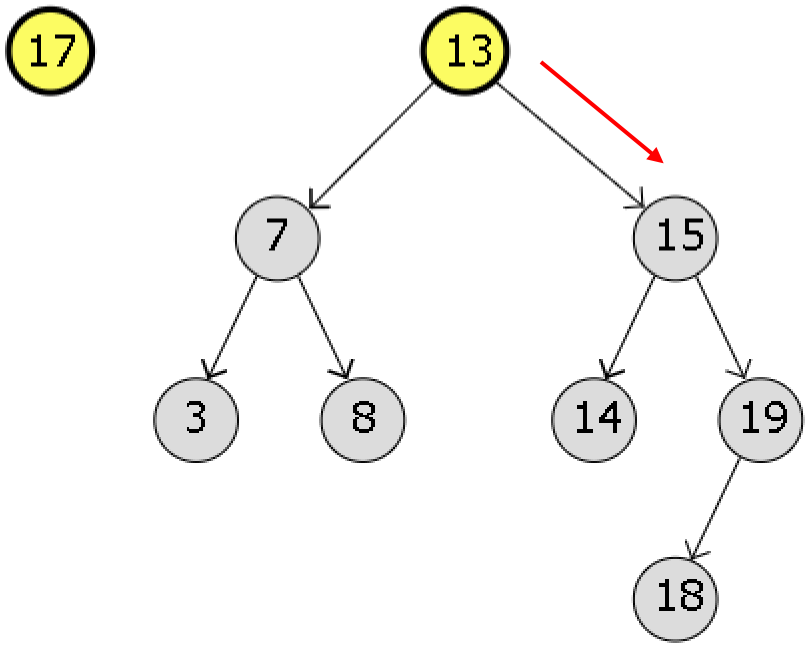
Practice Questions

Exercise 1

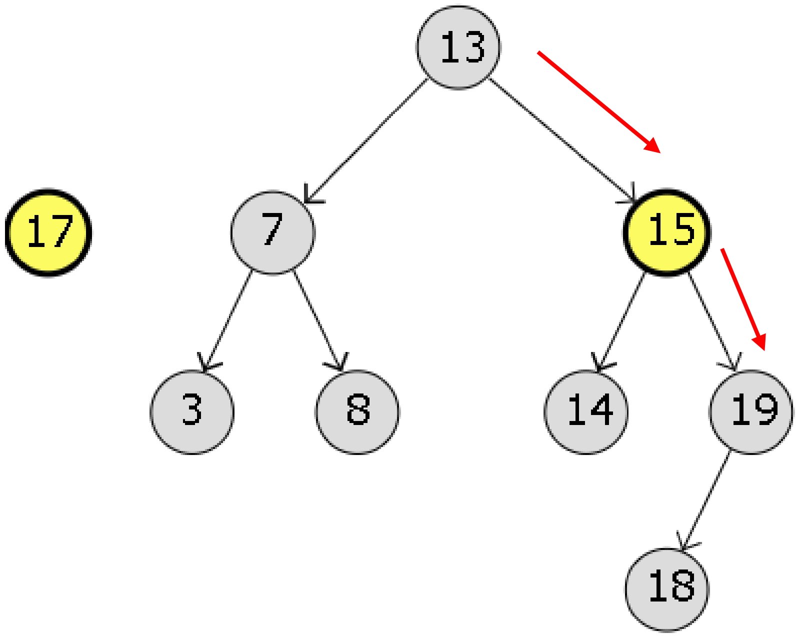


Exercise 1 – Solution

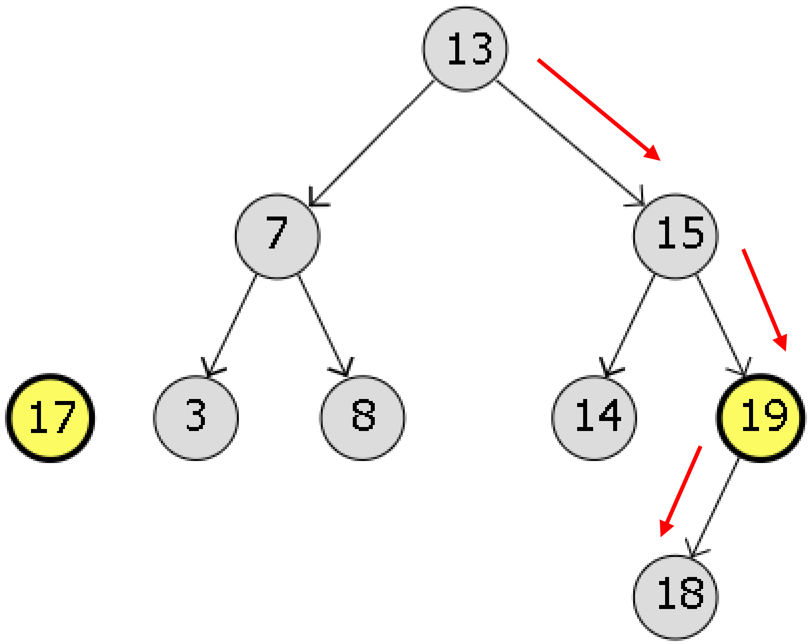
17 is higher than 13. Go right.



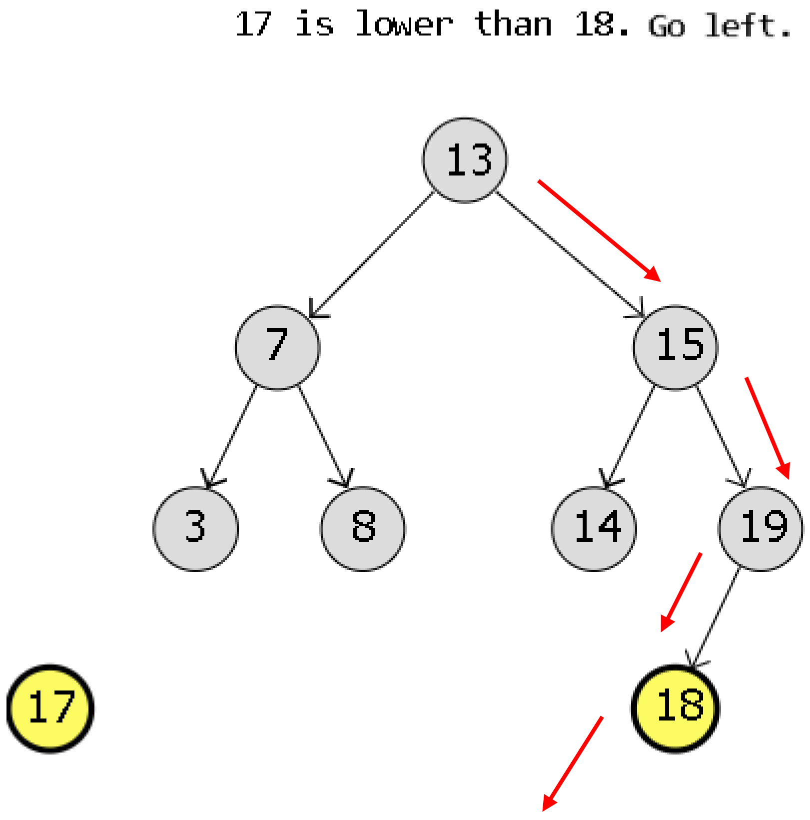
17 is higher than 15. Go right.

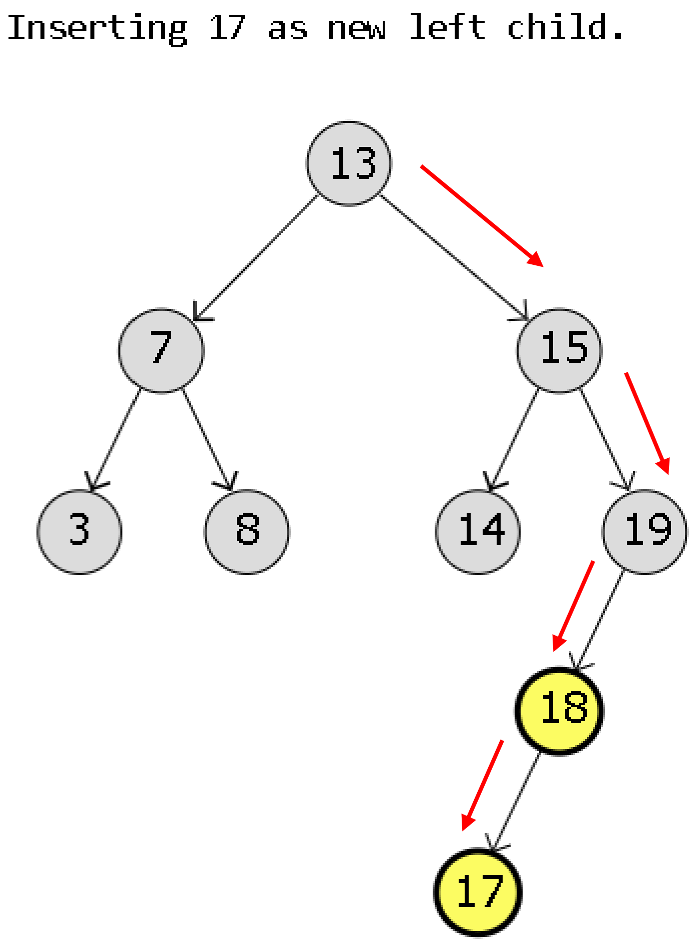


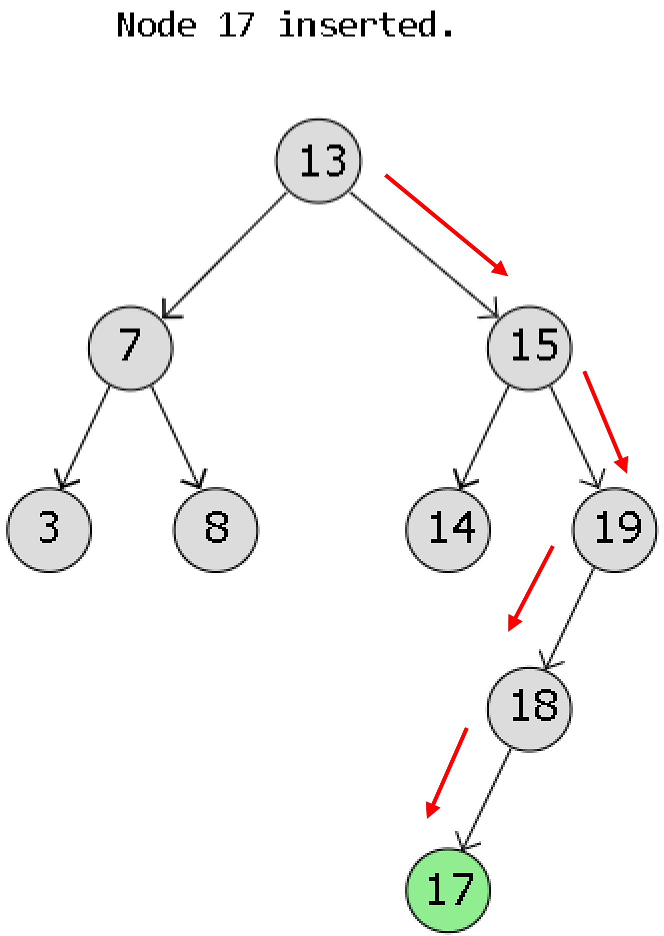
17 is lower than 19. Go left.



Exercise 1 – Solution (Cont.)



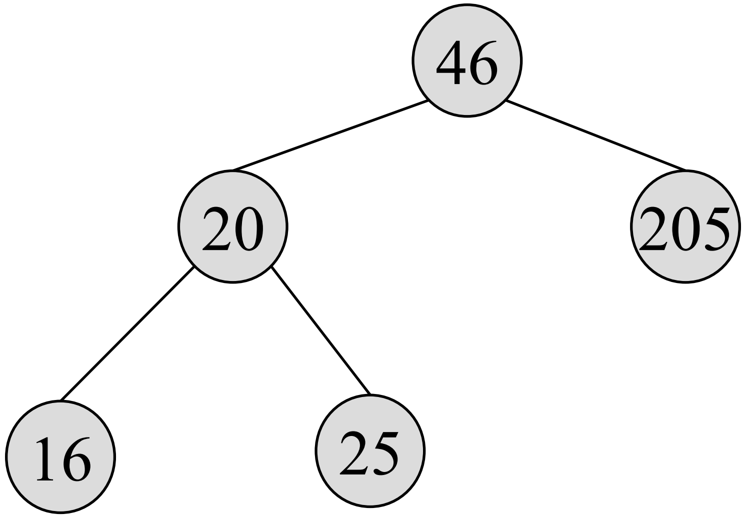




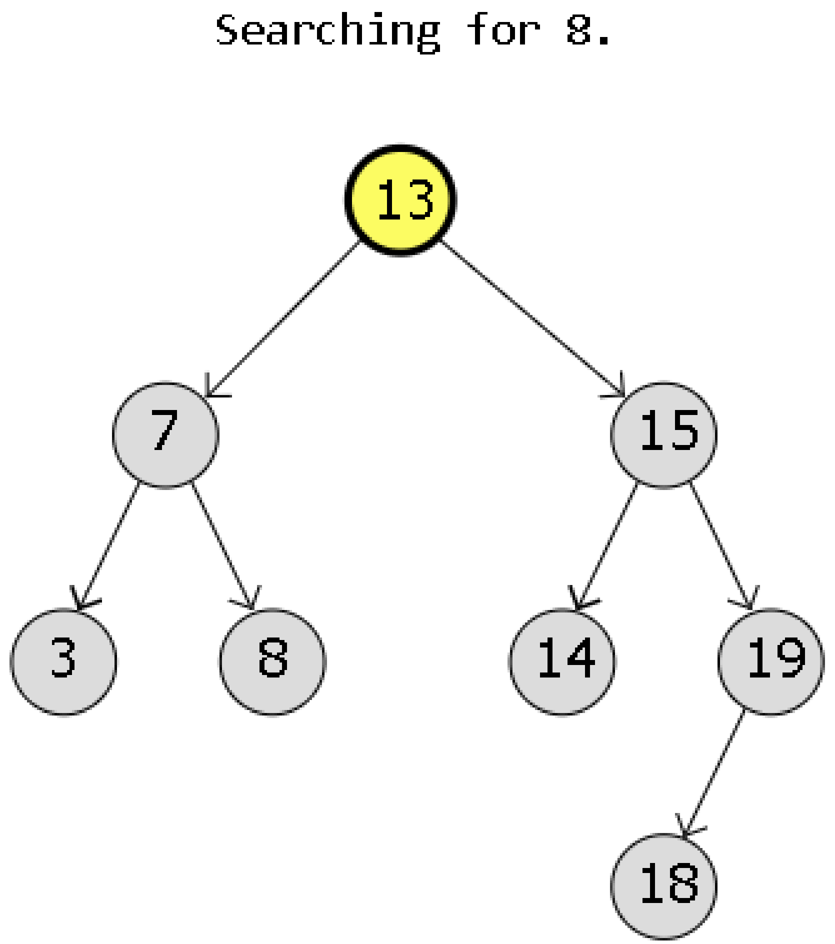
Exercise 2 – Solution

- Construct a BST using the following keys:

  46 20 205 16 25

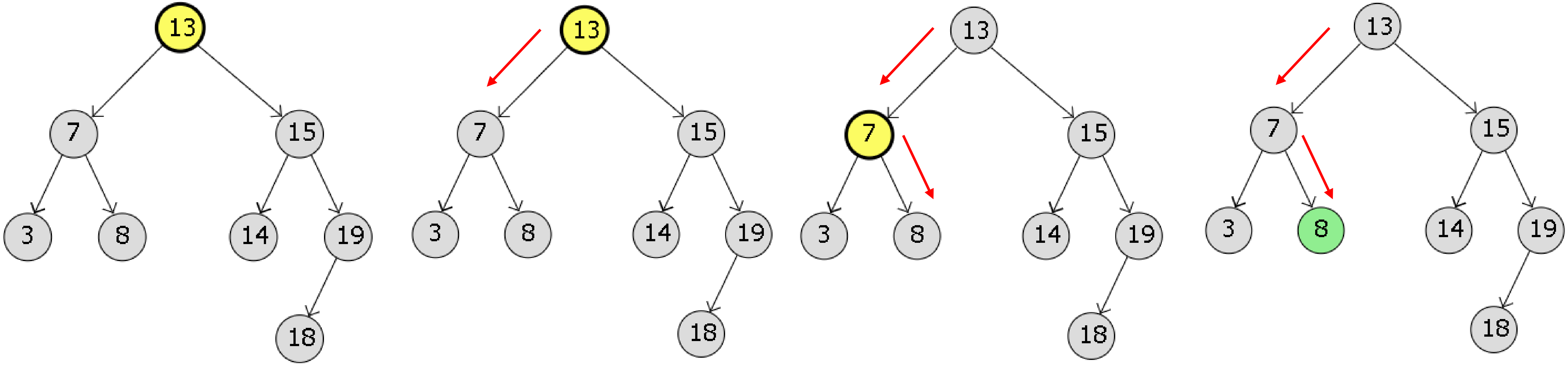


Exercise 3



Exercise 3 – Solution

Searching for 8. | 8 is lower than 13. Go left. | 8 is higher than 7. Go right. | Found 8!



Path: 13 → 7 → 8

References

- BST slides:
  - https://github.com/MustafaDaraghmeh/BST/blob/main/BST.pdf
- BST code in Python:
  - https://github.com/MustafaDaraghmeh/BST/blob/main/BST.py
- BST simulators:
  - https://bohr.wlu.ca/trees/#bst
  - https://www.cs.usfca.edu/~galles/visualization/BST.html
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd edition). MIT Press and McGraw-Hill.

Any Questions?