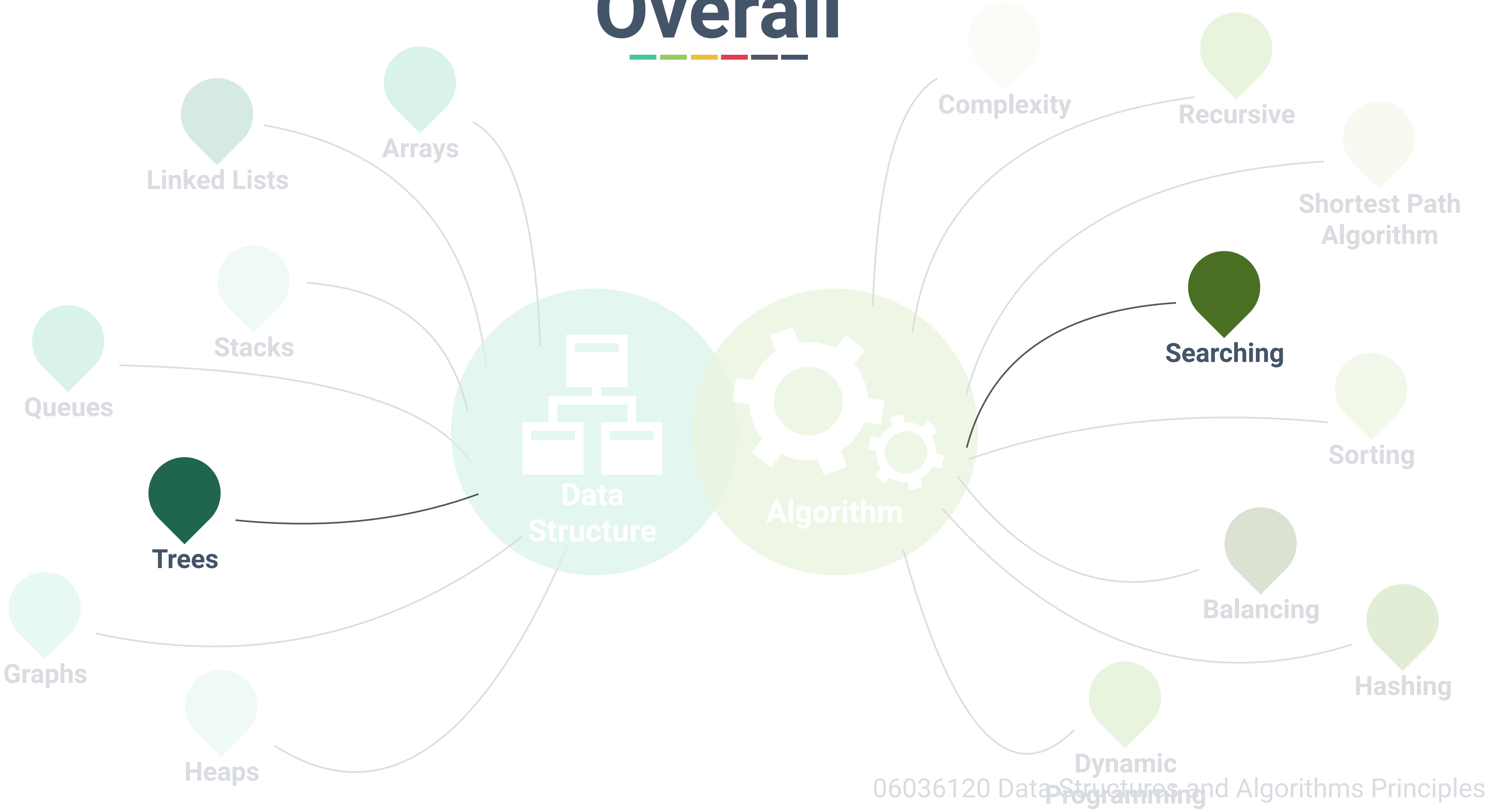
Chapter 8: Search Trees

Dr. Sirasit Lochanachit

Overall

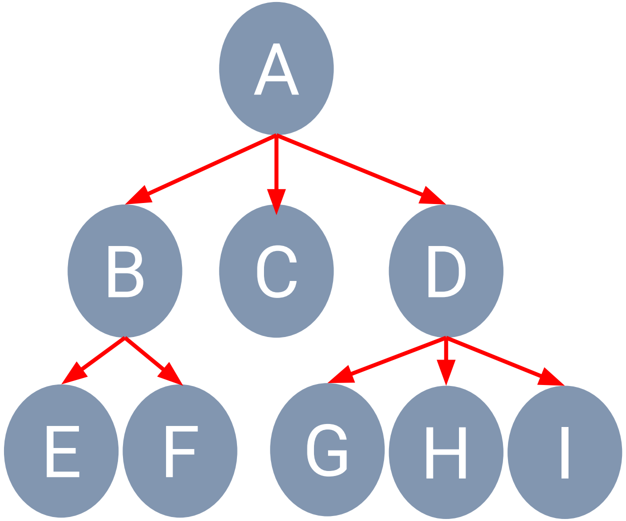

Linked Lists
Arrays
Stacks
Queues
Trees
Graphs
Heaps
Data Structure
Algorithm
Complexity
Recursive
Shortest Path Algorithm
Searching
Sorting
Balancing
Hashing
Dynamic Programming

Outline

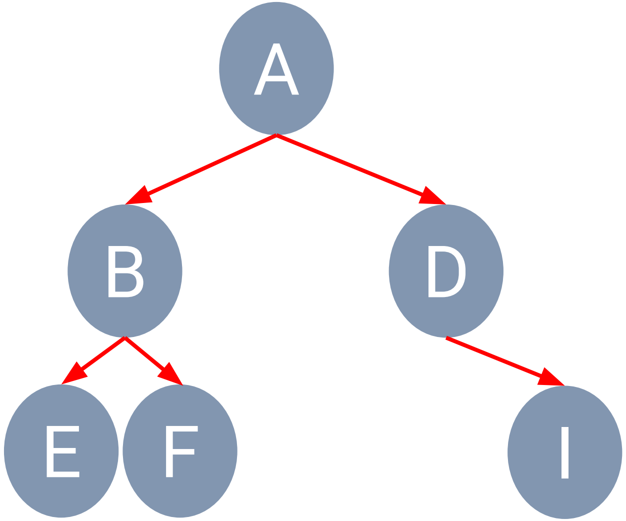
Binary Search Trees:

- Definition, properties and methods (search, add, delete)
- Algorithms and Operation examples

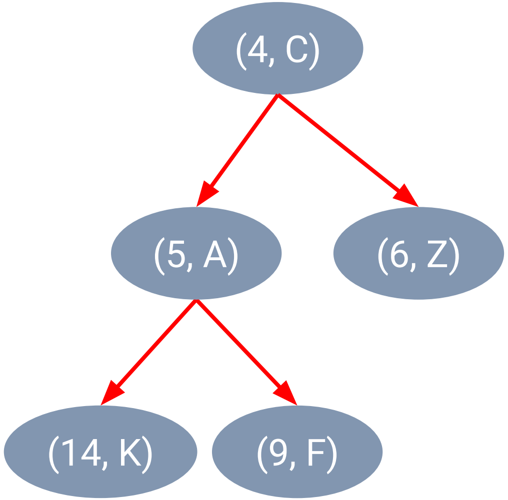
Types of Trees (Revisited)



General Tree

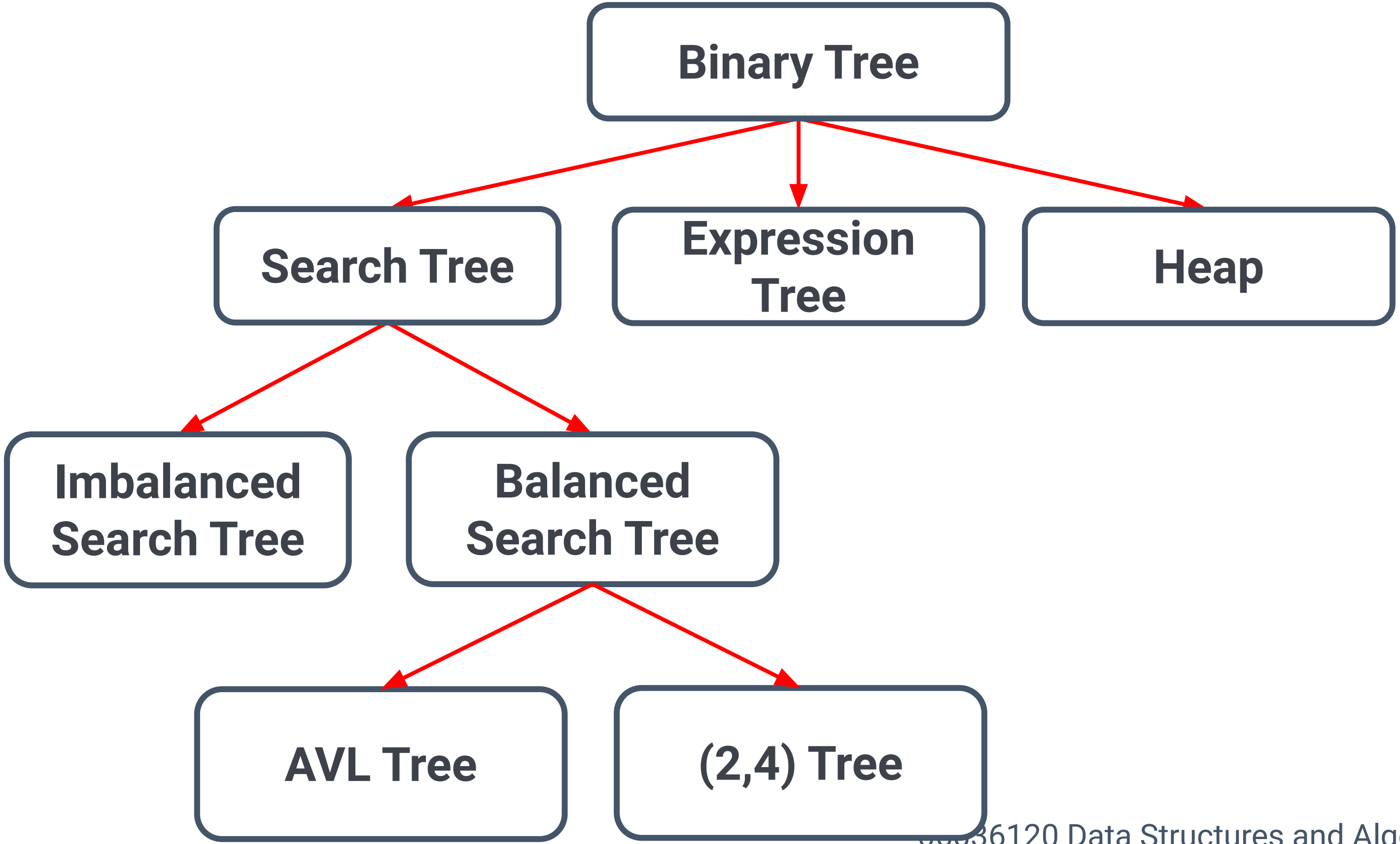


Binary Tree

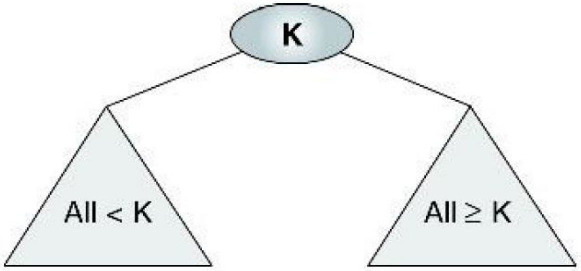


Binary Heap

Types of Binary Trees



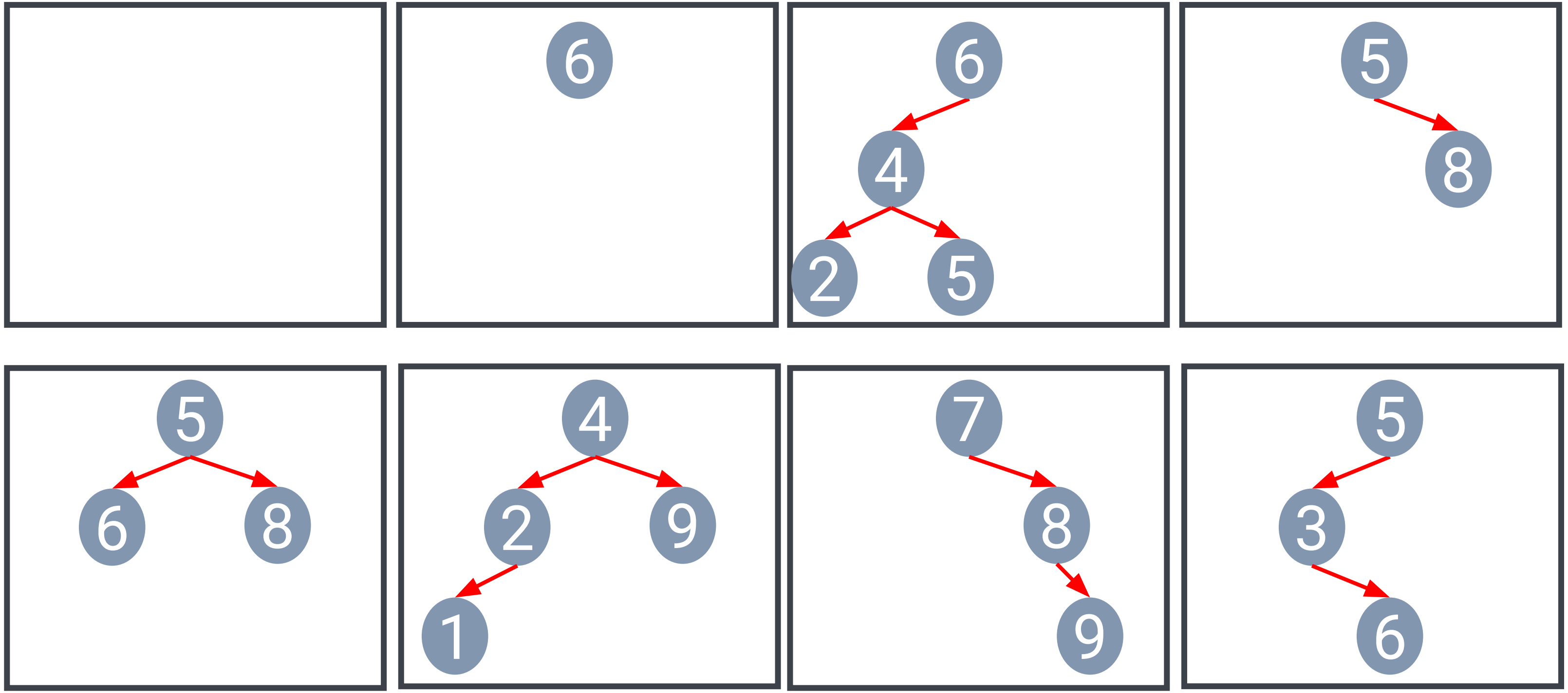
What is a Binary Search Tree?



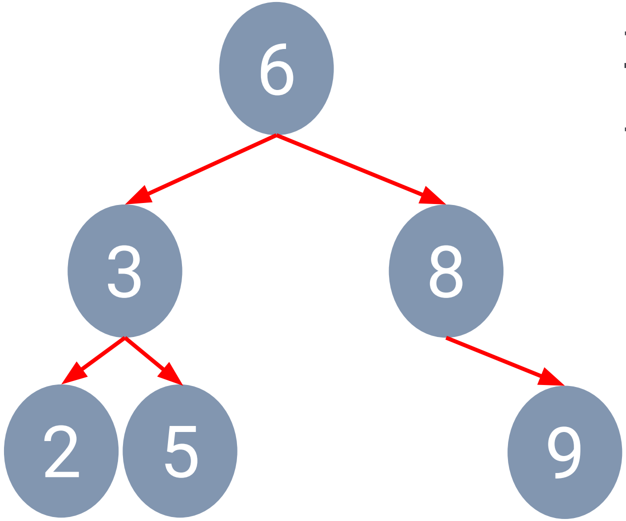
- A **binary search tree (BST)** is a binary tree that stores an ordered sequence of elements or pairs of keys and values and has the following properties [1]:
    - All keys/elements in the ***left subtree*** are **less than** their ***root***.
    - All keys/items in the **right subtree** are **greater than** or **equal to** their ***root***.
    - Each subtree itself is a binary search tree.

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Valid/Invalid Binary Search Trees?



Binary Search Tree

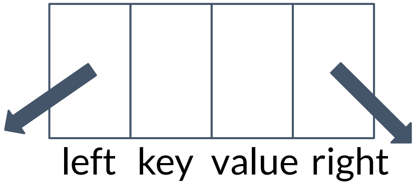


- A **binary search tree (BST)** applies the inorder traversal algorithm to insert keys and navigate the tree.
    - Produces a sorted keys in linear time.

BST Node Implementation

```
class BST_Node:
    def __init__(self, key, val, left=None, right=None, parent=None):
        self._key = key
        self._value = val
        self._leftChild = left
        self._rightChild = right
        self._parent = parent

    def ....
```

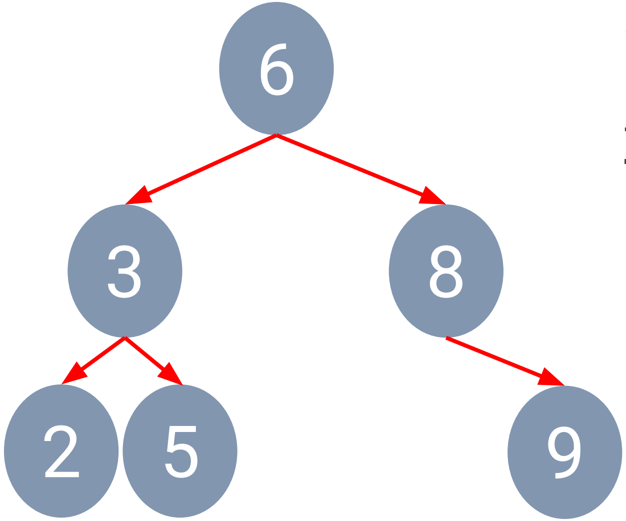


BST Implementation

```
class BinarySearchTree:
    def __init__(self):
        self._root = None
        self._size = 0

    def ....
    def ....
```

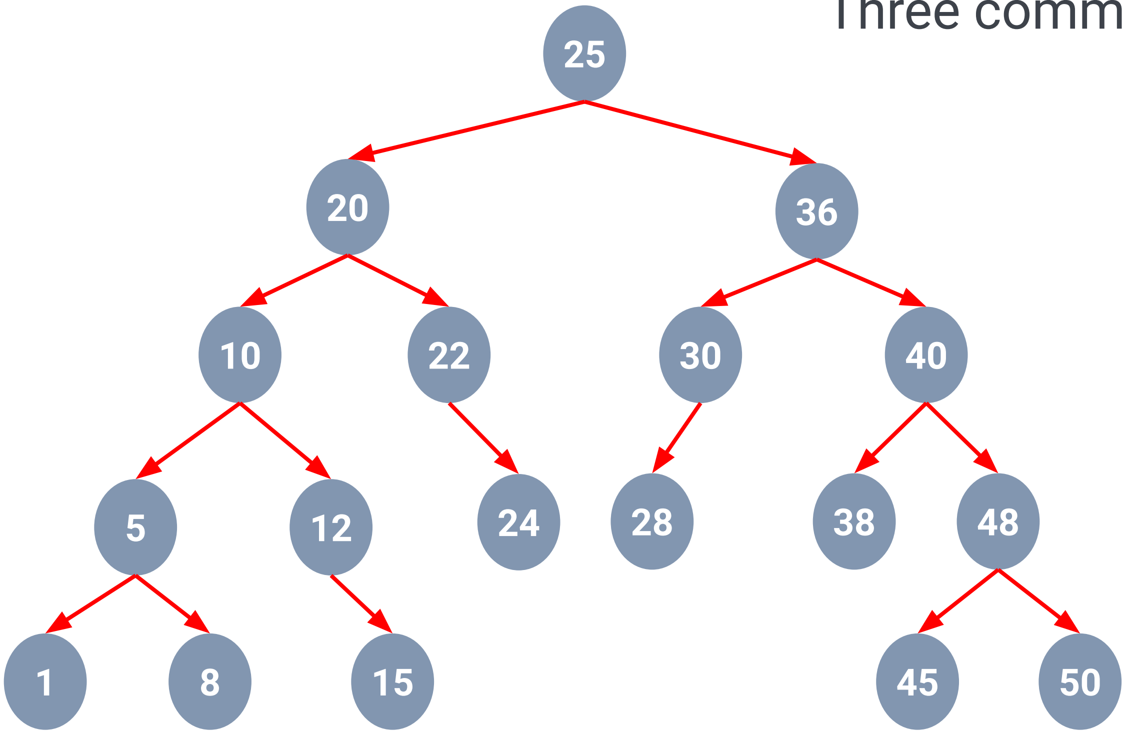
Search in Binary Search Tree



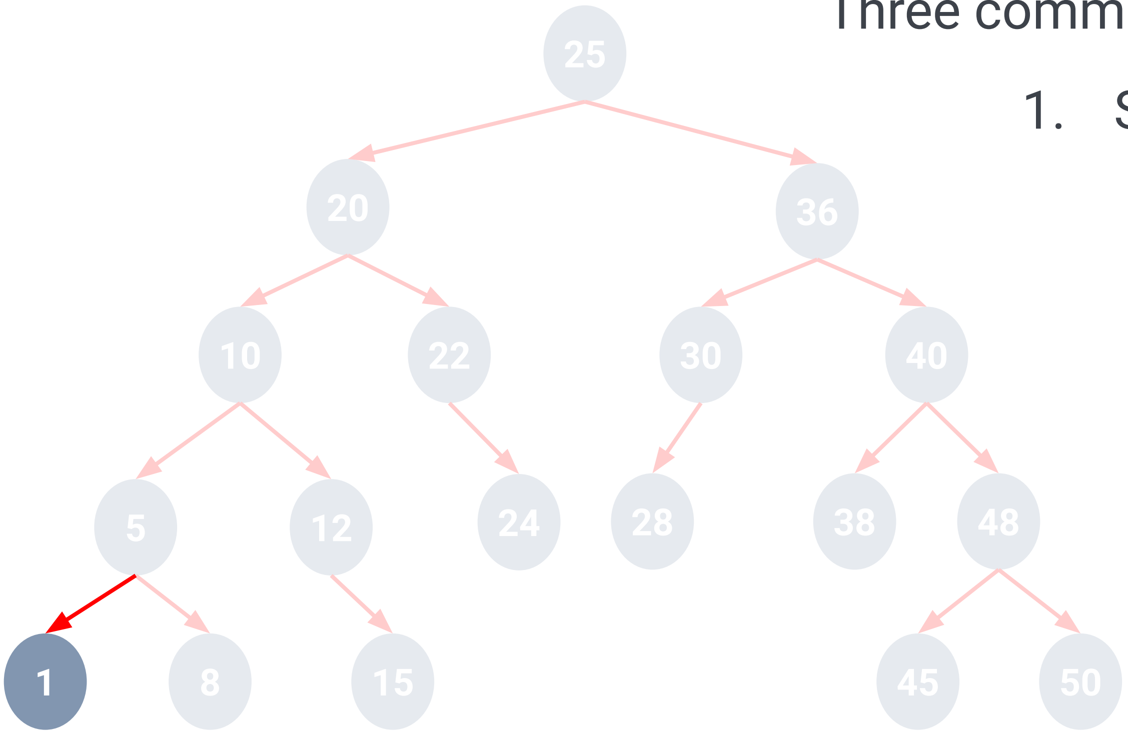
- A **binary search tree (BST)** can be used to find whether a given key is stored in a tree by starting at the root.
    - For each position *p*, the searched *key* are compared with the key stored at position *p*, which is denoted as *p.key*().

Searching in BST

Three common cases of searching in BST:



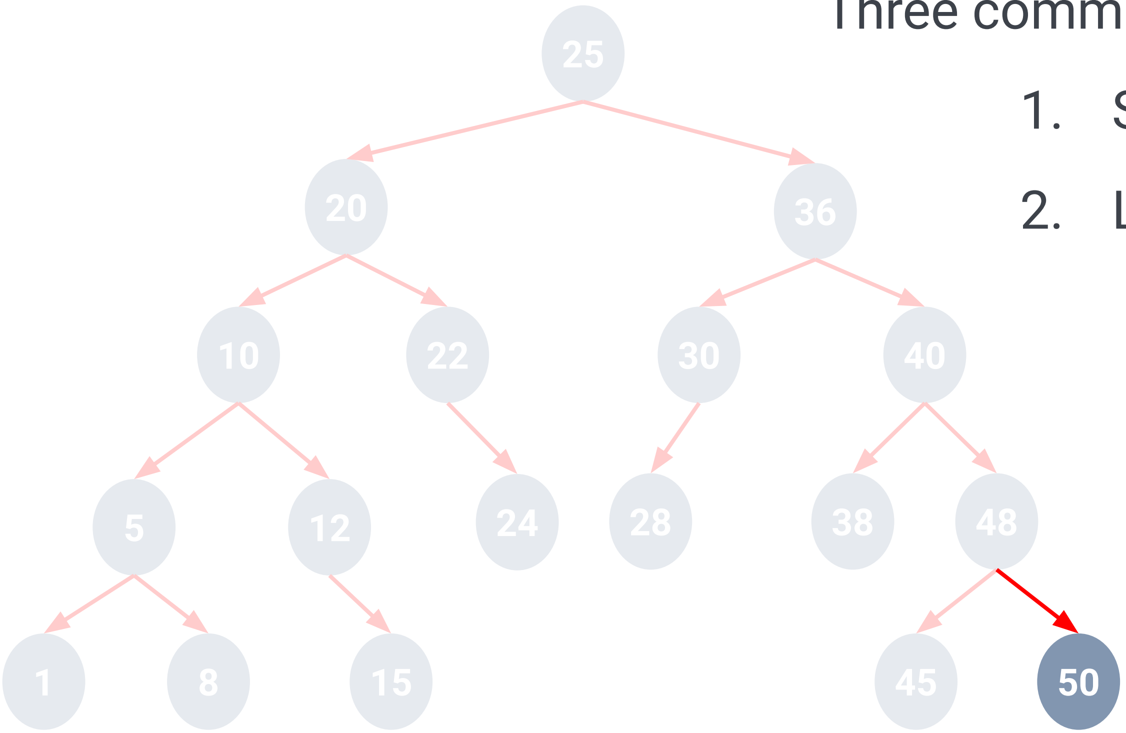
Find the Smallest Node



Three common cases of searching in BST:

1. Smallest key

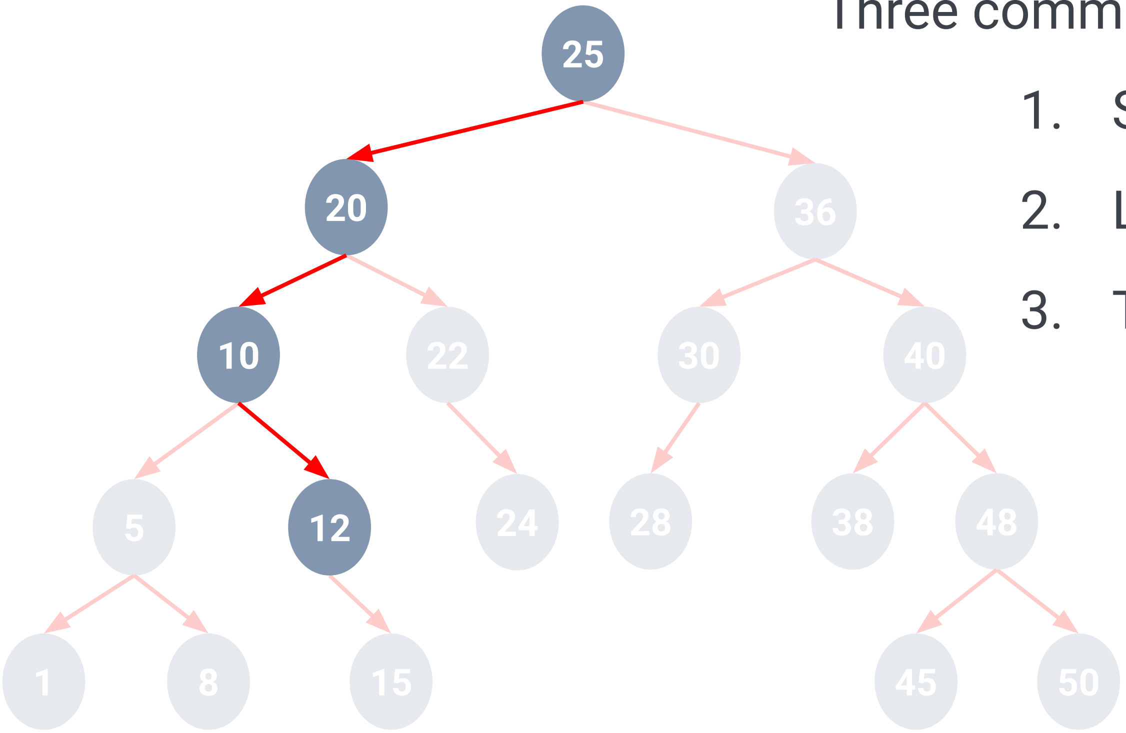
Find the Largest Node



Three common cases of searching in BST:

1. Smallest key
2. Largest key

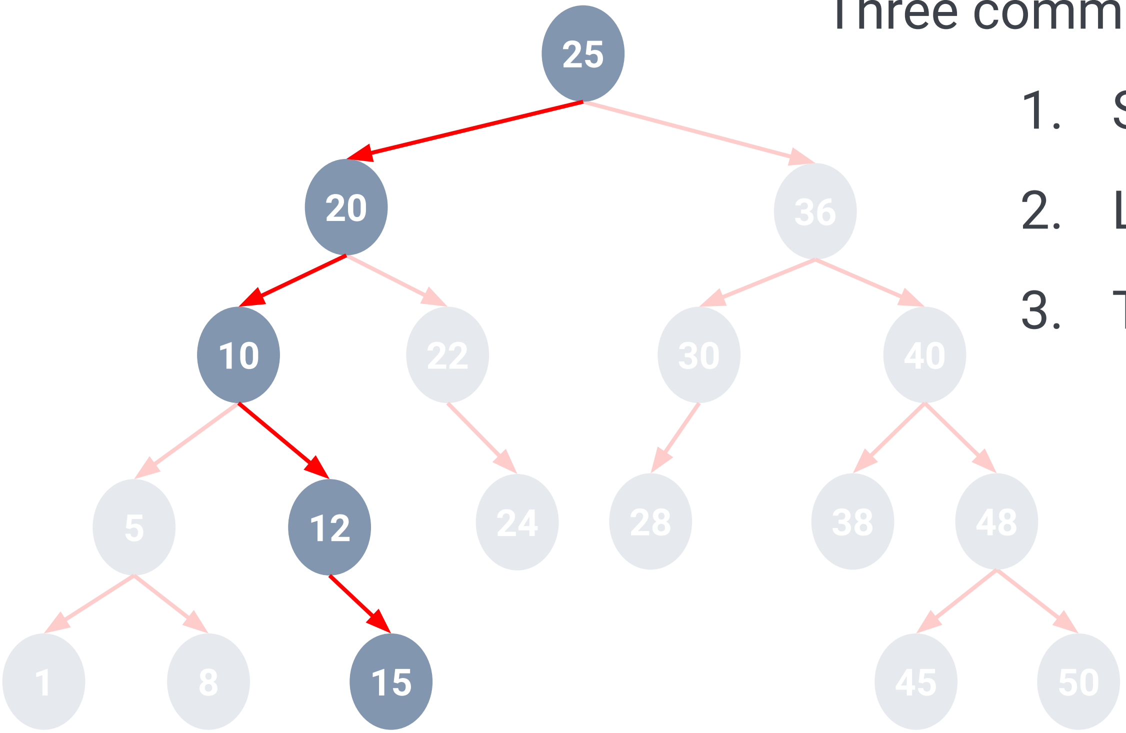
Searching a Target in a BST



Three common cases of searching in BST:

1. Smallest key
2. Largest key
3. Target key - **12**

Searching a Target in a BST

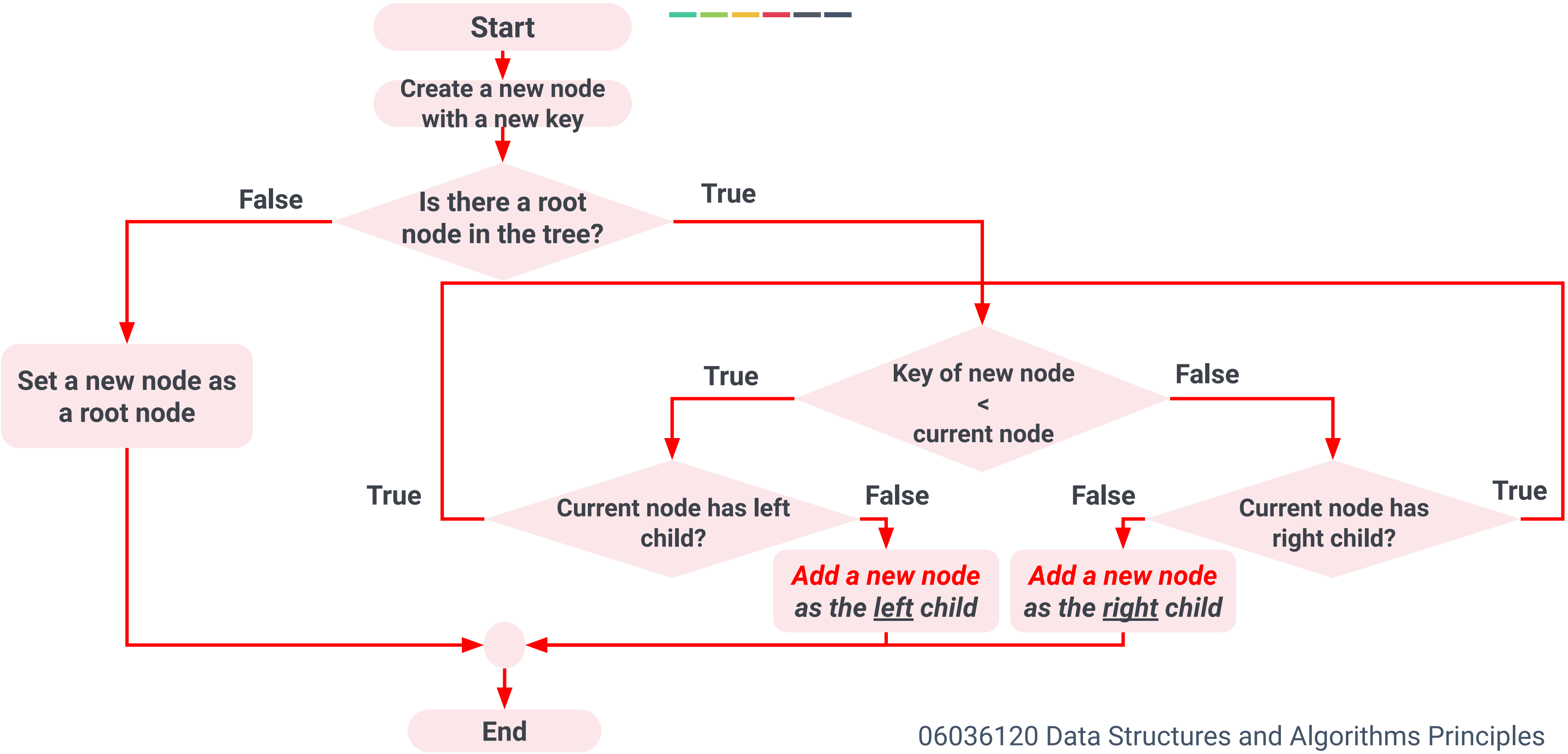


Three common cases of searching in BST:

1. Smallest key
2. Largest key
3. Target key - **18**

Binary Search Tree Algorithm

BST Construction and Insertion



BST Construction and Insertion

30 22 55 45

BST Insertion Exercise 1

23 18 44 52 12 20 8

BST Construction and Insertion Algorithm

Algorithm treeInsert(key, value):

    if rootNode is not empty **then**

        _treeInsert(key, value, rootNode)   # Root node exists

    else:

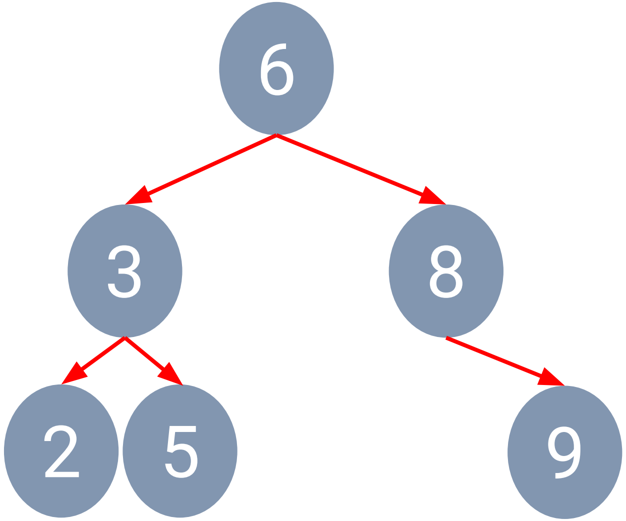
        create a new tree and put key & value as the root #Root node not exists

    self._size = self._size + 1

BST Construction and Insertion Algorithm

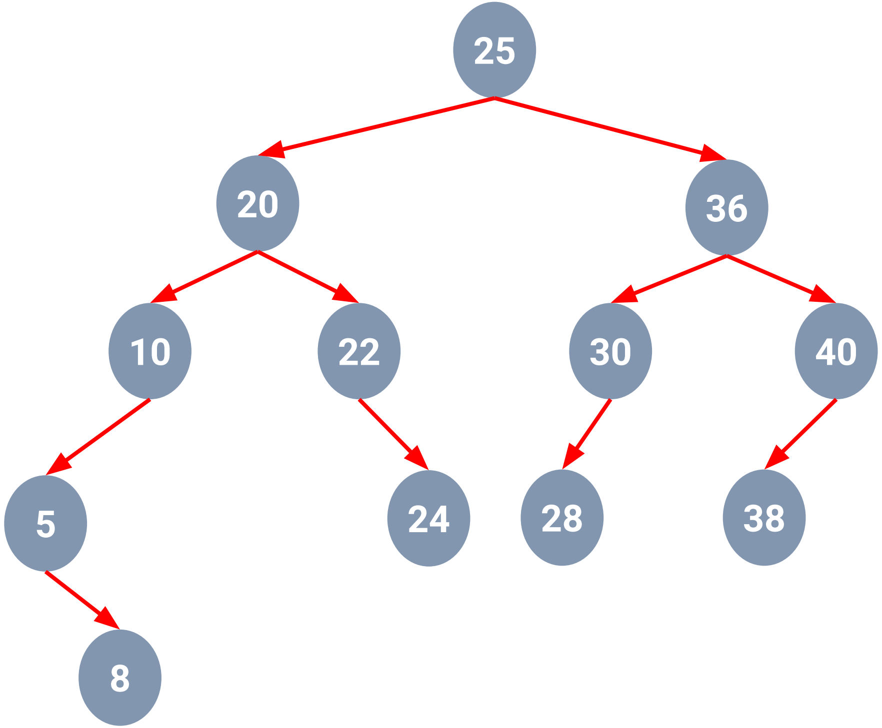
```
Algorithm _treeInsert(key, value, currentNode):
    if key < currentNode.key then
        if .....                    # Recur on left subtree
            .....
        else .....
            .....
    else:
        if .....                    # Recur on right subtree
            .....
        else .....
            .....
```

BST Deletion



- The deletion of a key from a BST can be performed on any node.
- Once a target node is found, three cases to consider:

BST Deletion Case #1



#1: The node has **zero** child.

BST Deletion Algorithm (cont.)

Algorithm _deleteNode(currentNode):

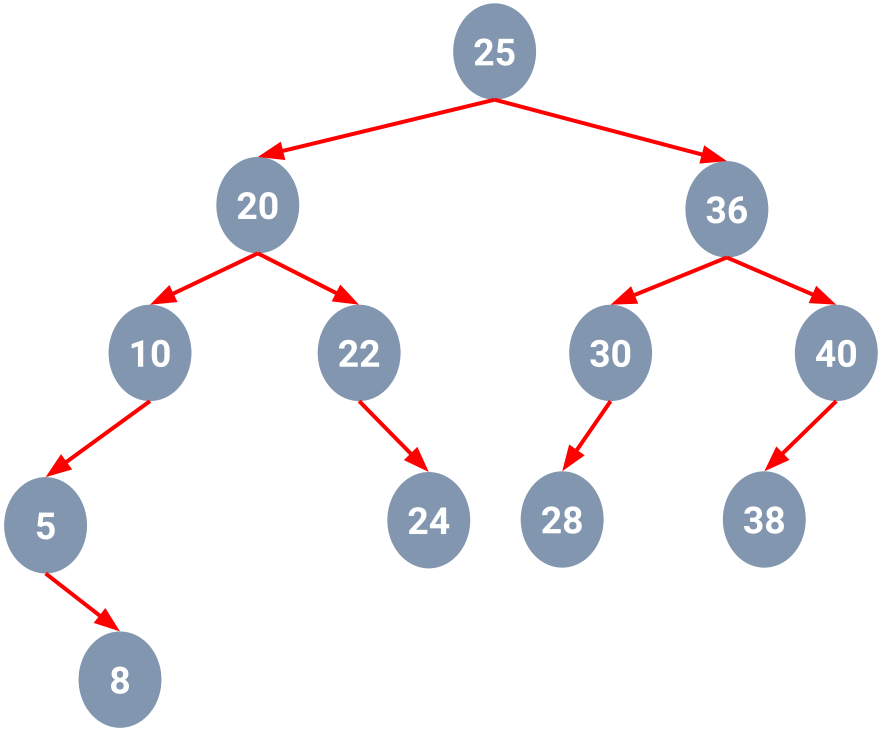
```
if currentNode is a leaf node then            #1: The node has zero child
    if currentNode is the left child of the parent node then
        Set the parent node's left child to None
    else
        Set the parent node's right child to None
```

BST Deletion Case #2

If a node has only a **single** child, the child could be promoted and replace its parent.

- 6 Sub-cases to consider:
  - A node has a left child
    - A node itself is a left child.
    - A node itself is a right child.
    - A node itself is the root node, no parent.

BST Deletion Case #2



#2: The node has **one** child.

- A node has a left child
  - A node itself is a left child.

BST Deletion Algorithm (cont.)

Algorithm _deleteNode(currentNode) *(cont.)*:

```
elif currentNode.hasAnyChildren():                #2: The node has one child
    if currentNode has a left child then
        if currentNode is the left child of the parent node then
            Update the parent pointer of its left child to currentNode's parent
            Set the parent node's left child to currentNode's left child
        else if currentNode is the right child of the parent node then
            Update the parent pointer of its left child to currentNode's parent
            Set the parent node's right child to currentNode's left child
        else
            Replace the currentNode with the left child
    if currentNode has a right child then
        ...
```

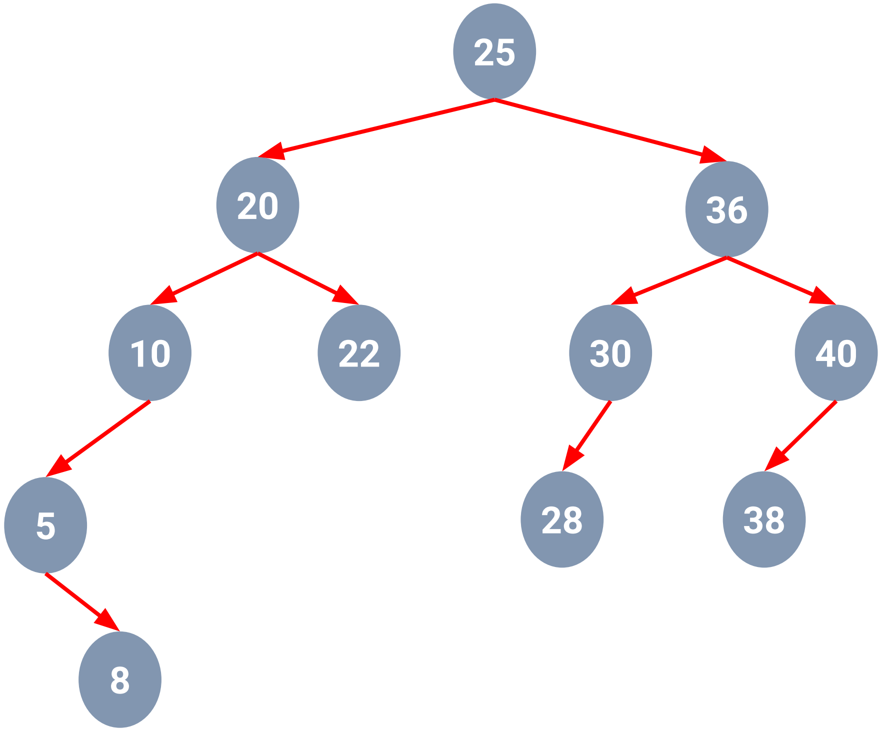
BST Deletion Case #3

- If a node has **two** children, one of the child could not be simply promoted and replace its parent since the other child would be left out of the tree.

BST Deletion Case #3

- To preserve a binary tree property, need to search the tree for a **successor** node, which is the next largest key after the deleted node:
  - If the successor has one child, it can be removed using case #1 or #2.
  - Then the successor node replaces the deleted node.
- The successor node could be either:
  - The minimum key node in the right subtree. Or
  - The maximum key node in the left subtree.

BST Deletion Case #3



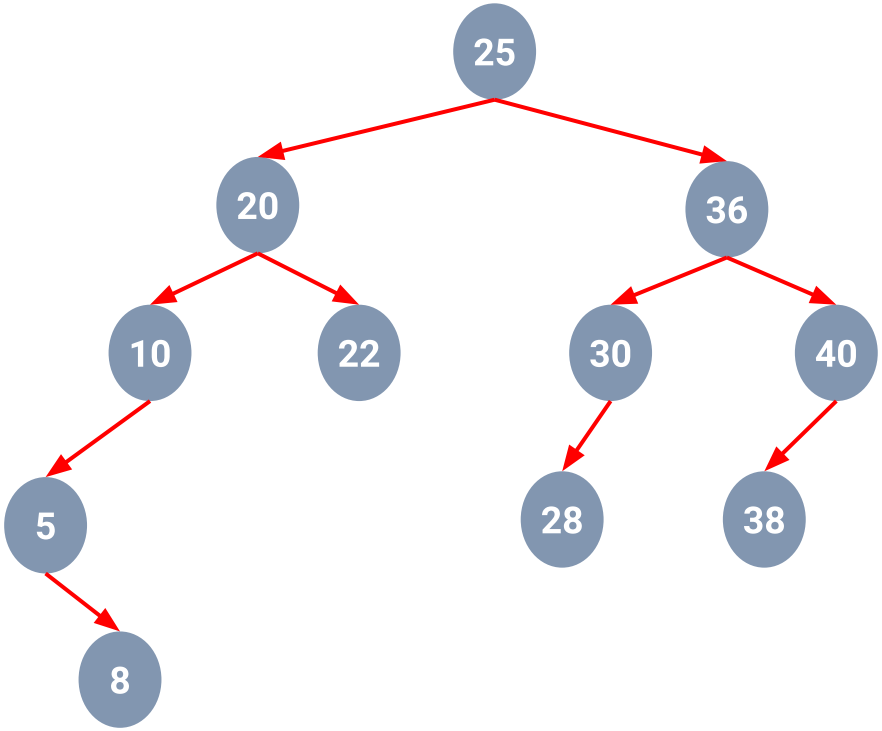
#3: The node has **two** child.

BST Deletion Algorithm (cont.)

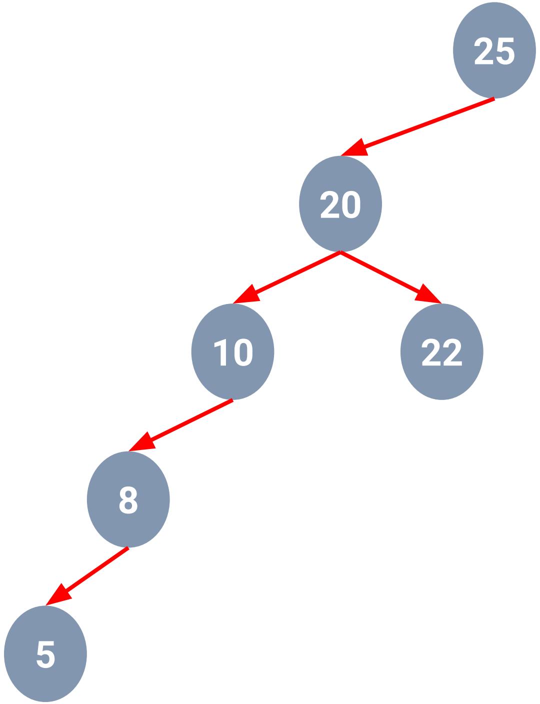
Algorithm _deleteNode(currentNode) *(cont.)*:

```
elif currentNode.hasBothChildren():        #3: The node has two children
    successor = currentNode._rightChild
    while successor.hasLeftChild():         # find min key in a right subtree
        successor = current._leftChild()
    Update the parent and children (if any) nodes of the minimum key node
    Replace the currentNode with the minimum key node
```

BST Deletion Exercise#1

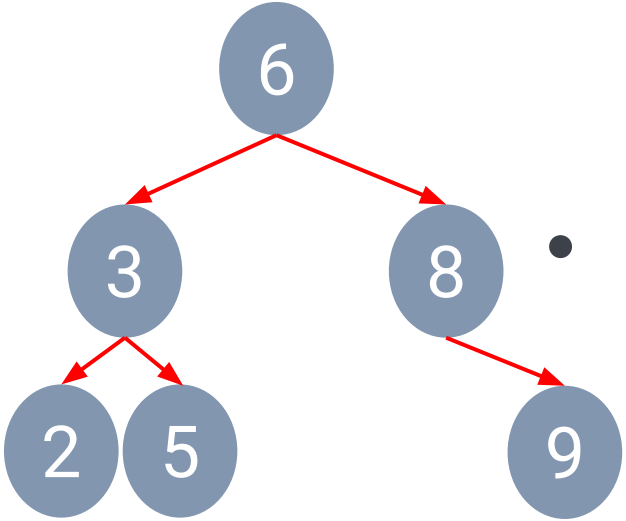


Binary Search Tree



- Running time of inserting node is also proportional to the **height of tree** (i.e. $\log_2 n$ or n) == O(h).
- A **balanced search tree** has the same number of nodes in both left and right subtree.
  - Worst-case performance is $O(\log_2 n)$.
- Inserting keys in sorted order would construct an **imbalanced tree**.
  - Provides poor performance of $O(n)$.

Balanced Binary Search Tree



- A **balanced binary search tree (BST)** maintains the balance through a rotation operation which consequently provides a better performance.
- Several types of binary tree that automatically ensure balance
  - AVL tree
  - Splay tree
  - Red-black tree