Chapter 8: Search Trees (Part 1)

Dr. Sirasit Lochanachit

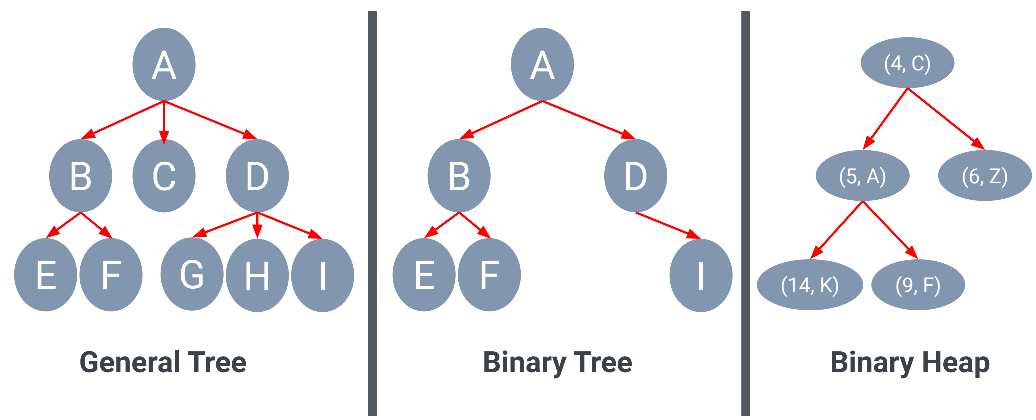
Overall

Outline

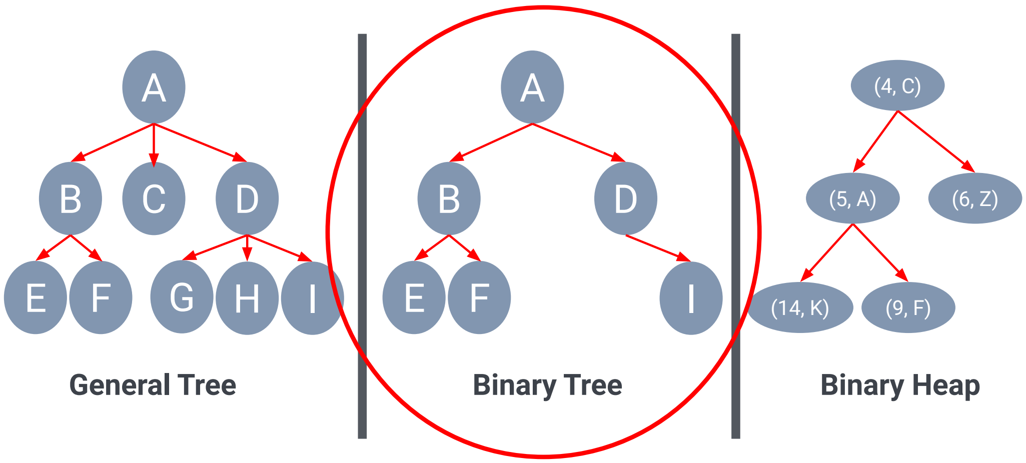
Binary Search Trees:

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

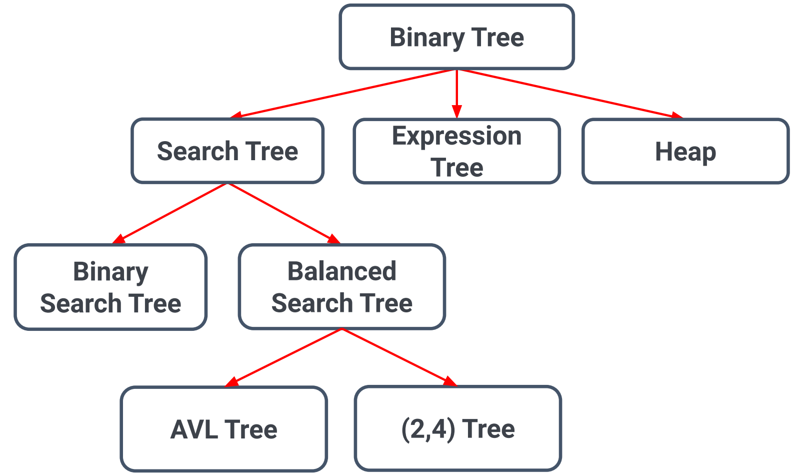
Types of Trees (Revisited)



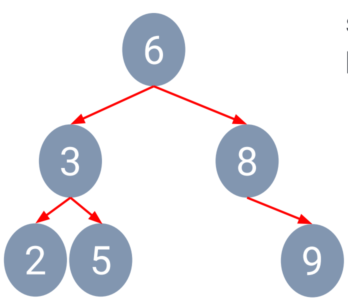
Types of Trees (Revisited)



Types of Binary Trees



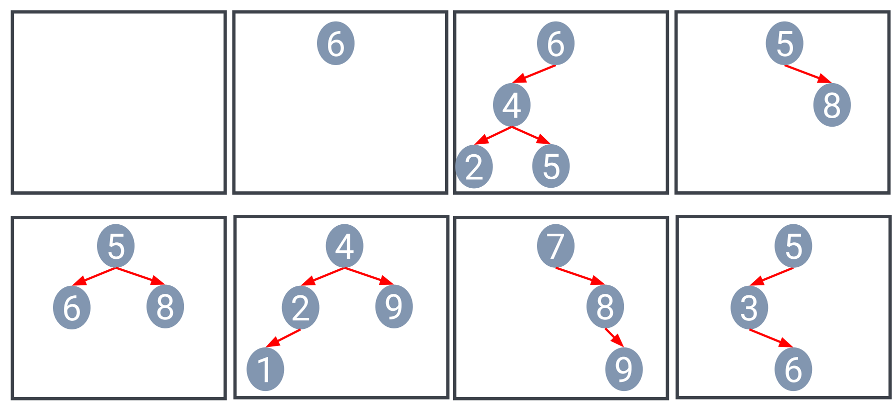
Binary Search Tree



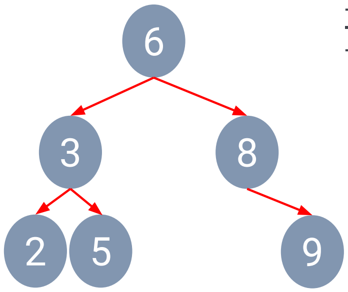
- A **binary search tree (BST)** is a binary tree that stores an ordered sequence of elements or pairs of keys and values and has the following properties [1]:
  - All keys/elements in the ***left subtree*** are **less than** their ***root***.
  - All keys/items in the **right subtree** are **greater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

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

Binary Search Tree



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.
    - For instance, [2, 3, 5, 6, 8, 9].

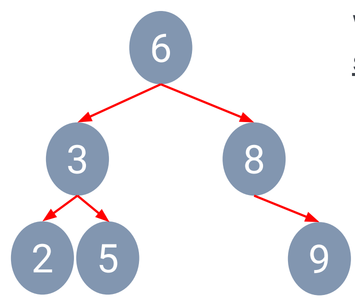
BST Node Implementation


```
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

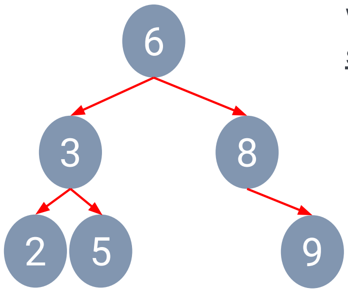

```
class BinarySearchTree:
    def __init__(self):
        self._root = None
        self._size = 0


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

Binary Search Tree



- A **binary search tree (BST)** can be used to find whether a given value/key is stored in a tree by starting at the root.
    - For each position p, the search value are compared with the key stored at position p, which is denoted as p.key().
    - If value < p.key(), then move to the left subtree of p and continue the search.
    - If value > p.key(), then move to the right subtree of p and continue the search.

Binary Search Tree



- A **binary search tree (BST)** can be used to find whether a given value/key is stored in a tree by <u>starting at the root</u>.
  - If value = p.key(), then the value is found and the search is stopped.
  - If the search reach an empty tree, the value is not found and the search is also stopped.
  - Running time of <u>search operation</u> is proportional to the **height of tree** (i.e. $\log_2 n$ or n) == O(h).

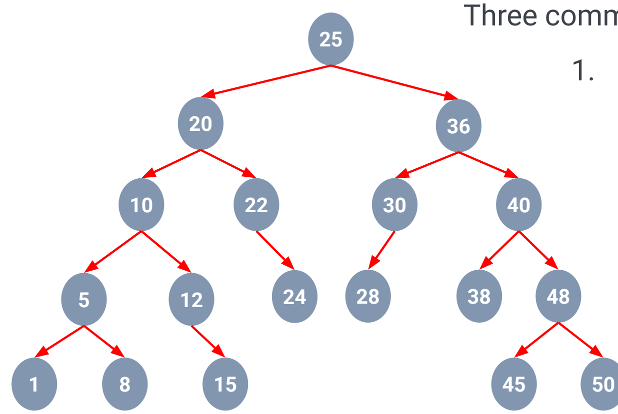
Binary Search Tree Algorithm

```
Algorithm treeSearch(target):
    if rootNode is not empty then                        # Root node exists
        result = _treeSearch(rootNode, target)           # Perform search on the tree
        return result._value                             # Return the node's value (None if not found)
    else:                                                # Root node not exists
        return None
```


Binary Search Tree Algorithm

```
Algorithm _treeSearch(p, target):    # p denotes the current node, starting from the root
    if target == p._key then
        return p                                             # Target is found
    else if target < p.key() and p.getLeftChild() is not None then
        return _treeSearch(p.getLeftChild(), target)         # Recur on left subtree
    else if target > p.key() and p.getRightChild() is not None then
        return _treeSearch(p.getRightChild(), target)        # Recur on right subtree
    return None                                              # Target is not found
```

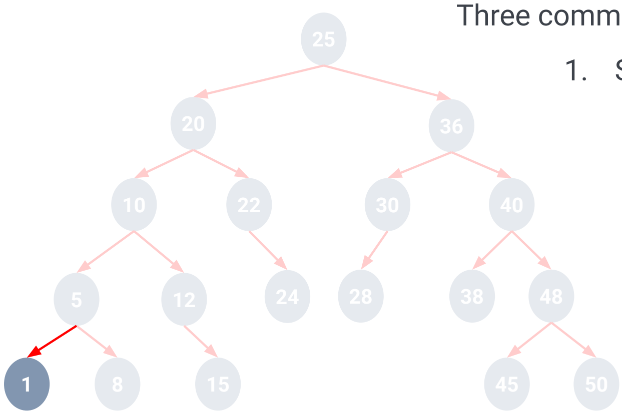

Searching in BST



Three common cases of searching in BST:

1. Smallest key

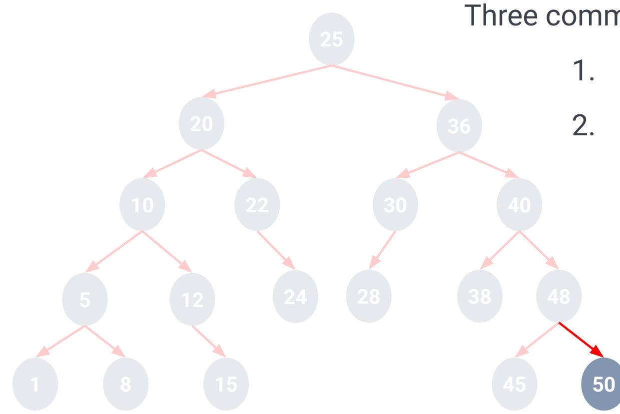
Searching in BST



Three common cases of searching in BST:

1. Smallest key

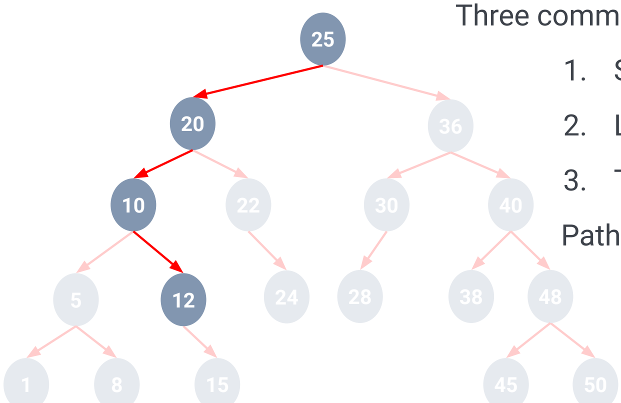
Searching in BST



Three common cases of searching in BST:

1. Smallest key
2. Largest key

Searching in BST

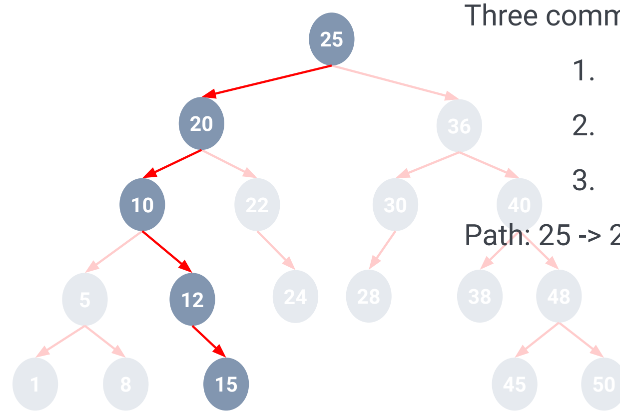


Three common cases of searching in BST:

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

Path: 25 -> 20 -> 10 -> 12

Searching in BST

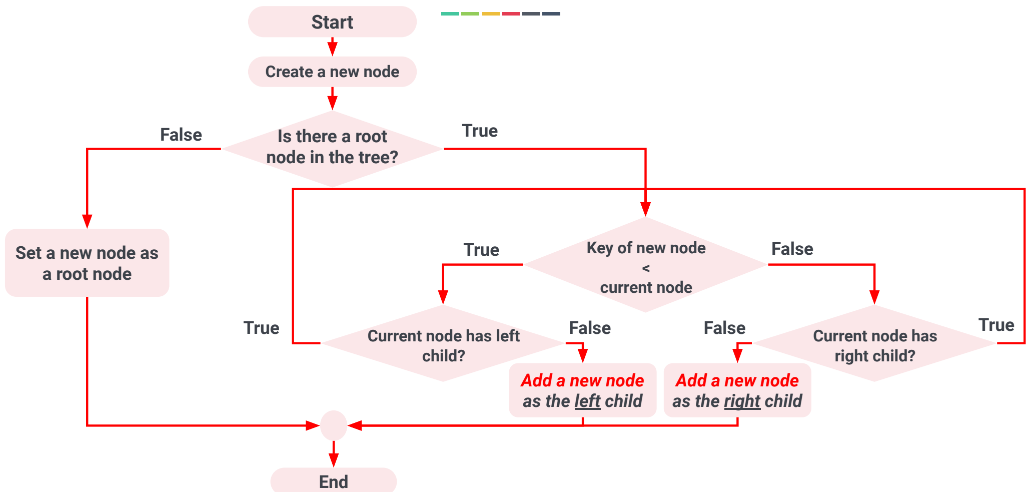


Three common cases of searching in BST:

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

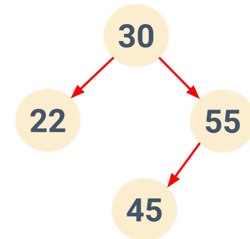
Path: 25 -> 20 -> 10 -> 12 -> 15 -> Not found

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 .....
            .....
```

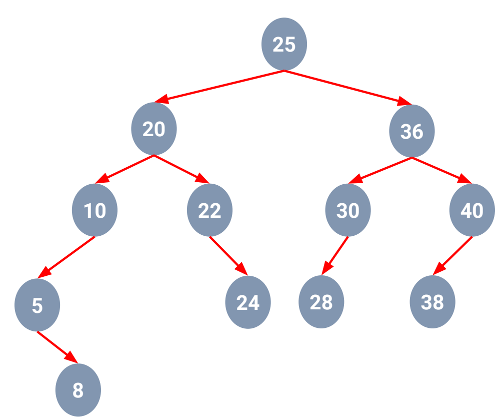

BST Deletion

- Deleting an element from a BST is a bit more challenging than inserting.
    - The deletion of a key can be performed on any node.
    - In contrast, insertion usually occurs at the bottom of a tree path.
- Need to consider whether a tree has only a single node.
    - Removing only the root of the tree if the keys are matched.
    - Otherwise, return None or raise errors
- Once a target node is found, three cases to consider:
    - The node either has zero, one or two children.

BST Deletion Algorithm

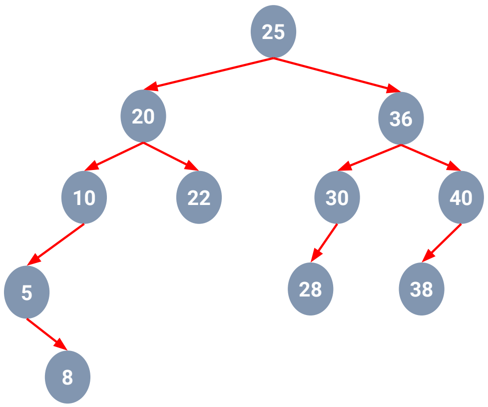
```
Algorithm deleteNode(key):
    if self._size > 1 then                      # Tree has >1 nodes
        node = _treeSearch(self._root, key)     # Find the node
        if node is not empty then               # Node is found in the tree
            _deleteNode(node)
            self._size = self._size - 1
        else:    return None or error           # Node is not found
    else if self._size == 1 and self._root._key == key:    # Only single node
        self._root = None
        self._size = 0
    else:    return None or error
```


BST Deletion Case #1



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

BST Deletion Case #1



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

BST Deletion Algorithm (cont.)

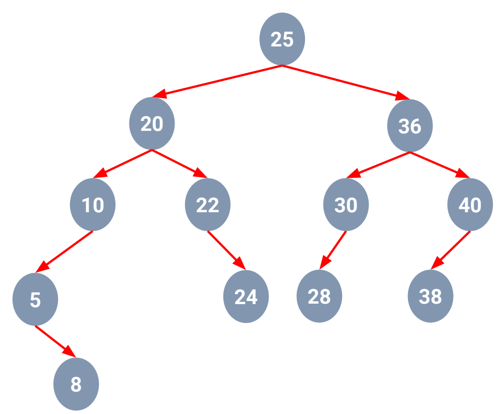
Algorithm _deleteNode(currentNode):

- **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 -> Update its <u>left child</u> to point to its parent
    - A node itself is a left child. -> Update parent's <u>left child</u>
    - A node itself is a right child. -> Update parent's <u>right child</u>
    - A node itself is the root node, no parent. -> replace with the left child
  - A node has a right child -> Update its <u>right child</u> to point to its parent
    - Same as left child case.

BST Deletion Case #2



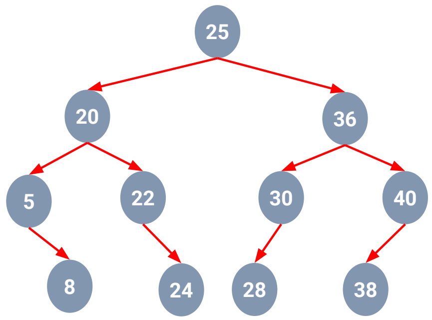
#2: The node has **one** child.

- A node has a left child

  -> Update its <u>left child</u> to point to its parent
  - A node itself is a left child.

  -> Update parent's <u>left child</u>

BST Deletion Case #2



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

BST Deletion Algorithm (cont.)

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

elif currentNode.hasAnyChildren()**:** #2: The node has zero child

if currentNode <u>has</u> a <u>left child</u> **then**

if currentNode <u>is</u> the <u>left child</u> 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 <u>is</u> the <u>right child</u> 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

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.
- 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:
  - The successor has <u>no more than one child</u>, so it can be removed using case #2.
  - Then the successor node replaces the deleted node.

BST Deletion Case #3

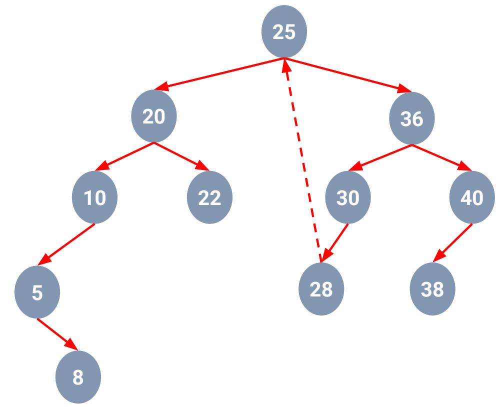
- The successor node could be either:
  - The <u>minimum</u> key node in the <u>right subtree</u>. Or
  - The <u>maximum</u> key node in the <u>left subtree</u>.
- In case of min key, this value is at the leftmost child of the tree.
- To find the successor, follow leftChild references in each node until reaching a node that does not have a left child.

BST Deletion Algorithm (cont.)

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

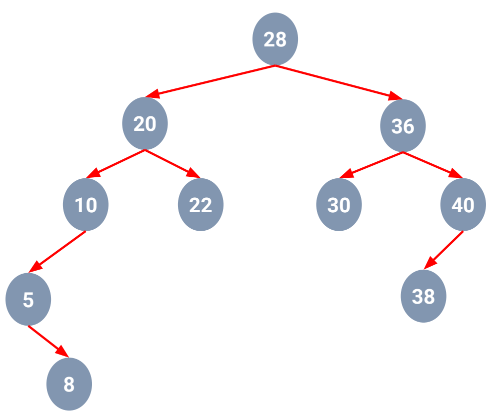
```
elif currentNode.hasBothChildren():        #3: The node has two children
    successor = currentNode._rightChild
    while successor.hasLeftChild():        # find min key in a subtree
        successor = current._leftChild()
    Update the parent and children (if any) nodes of the minimum key node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


BST Deletion Case #3



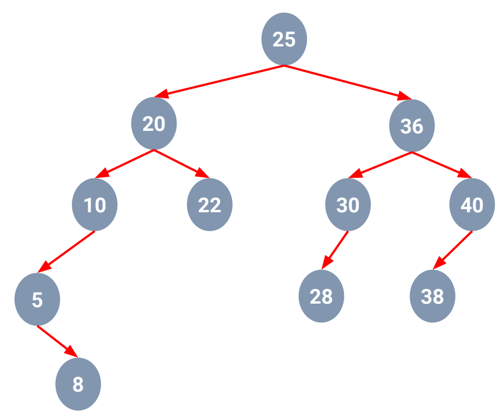
#3: The node has **two** child.

BST Deletion Case #3

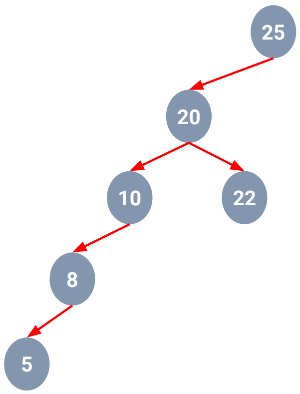


#3: The node has **two** child.

BST Deletion Exercise#1

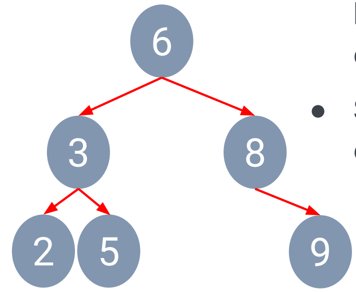


Binary Search Tree



- 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 $O(n)$.
- Other operations' performances are also limited by the height of the tree.

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