LUT University

BM40A1500 DATA STRUCTURES AND ALGORITHMS

BINARY TREES

2024

RECURSION

- The process of solving a large problem by reducing it to sub-problems
  - Subproblems are identical in structure to the original problem and simpler to solve.
  - Solved using functions that call themselves.

- A recursive algorithm has two parts:
  - **The base case**, which handles a simple input that can be solved.
  - **The recursive part** which contains one or more recursive calls to the algorithm.
    - In every recursive call, the parameters must be in some sense “closer” to the base case than those of the original call.

RECURSION

- Example:
  - Multiplication of two numbers

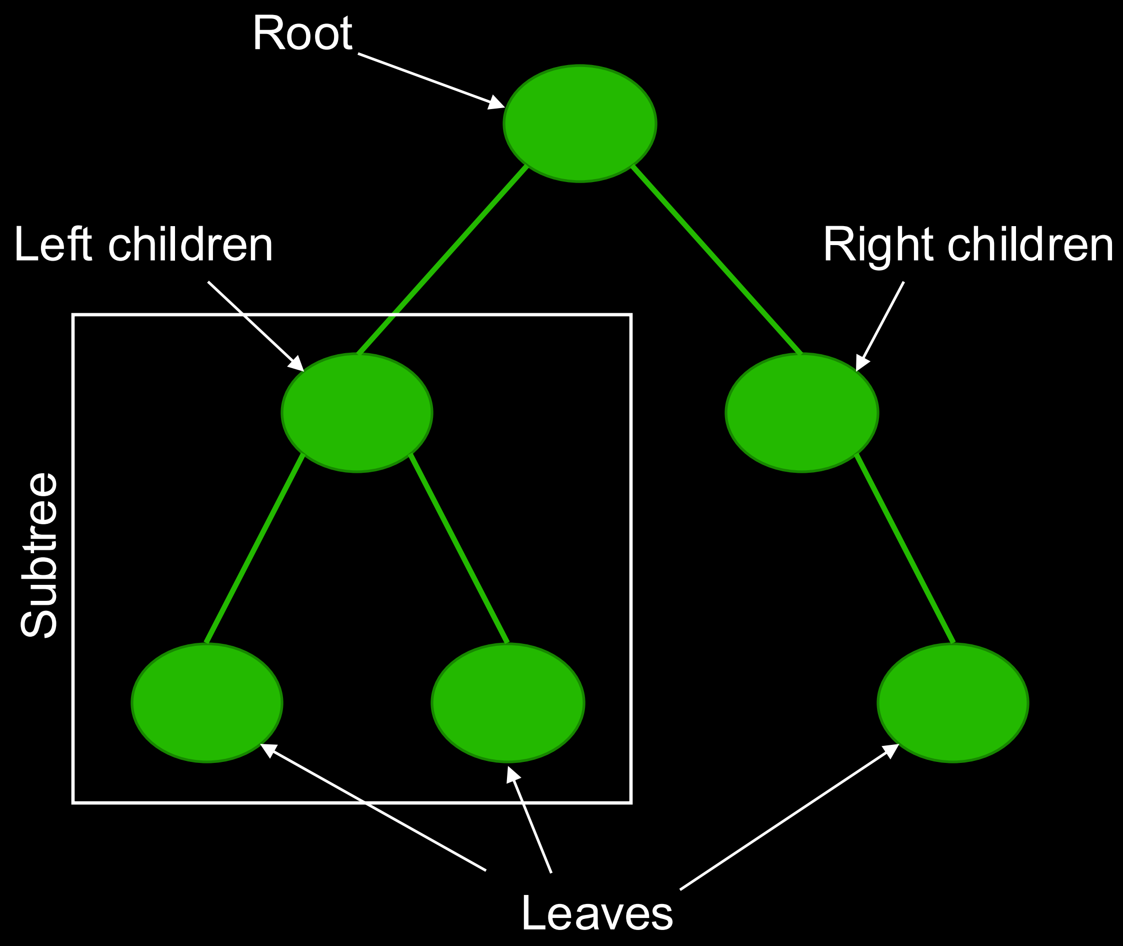
```
def multiply(x,y):
        if x == 1:
                return y
        else:
                return y + multiply(x - 1, y)
```

$1 \cdot y = y$

$$x \cdot y = \overbrace{y + \underbrace{y + \dots + y}_{x-1}}^{x} = (x-1)y + y$$

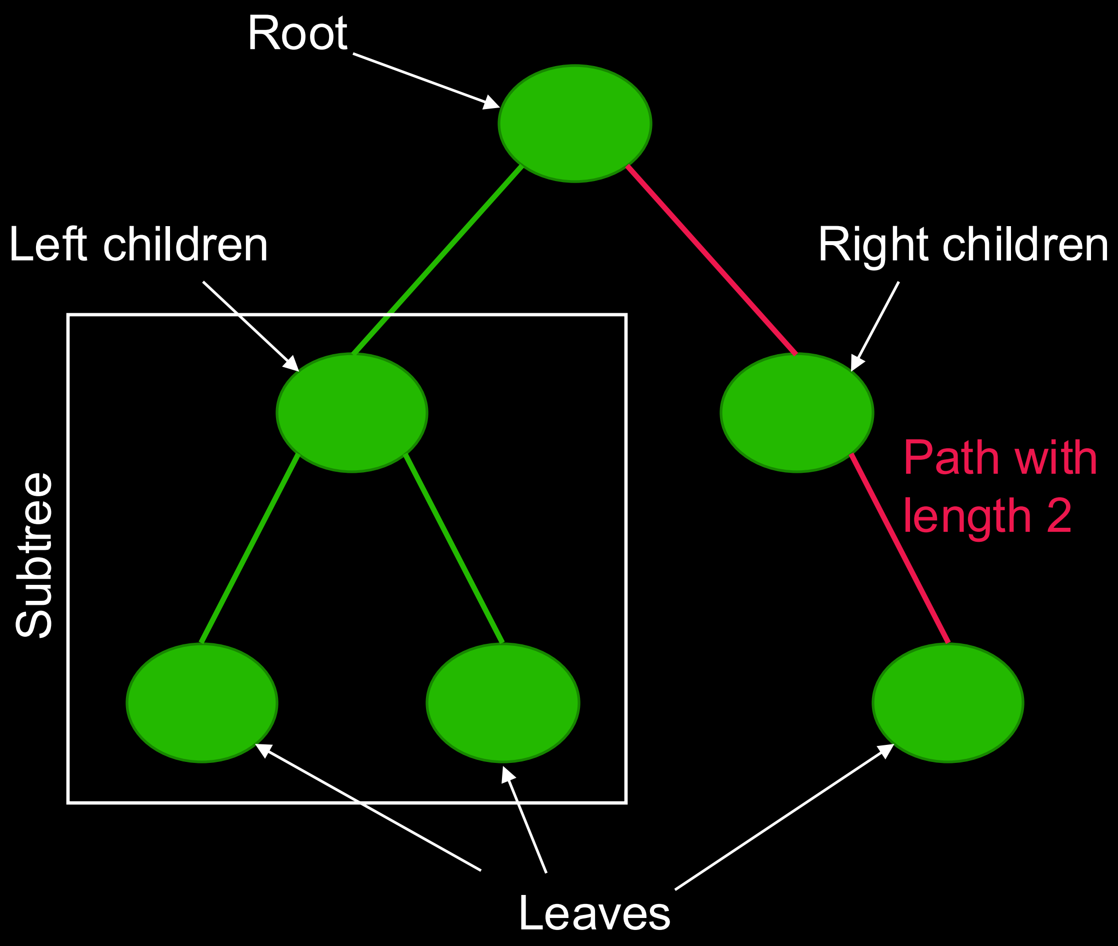
BINARY TREE

- Data structure consisting of nodes.
- Each node has at most two children.
- Terminology:
  - Root
  - Subtree
  - Children – parent (ancestor and descendant)
  - Leaf



BINARY TREE

- Data structure consisting of nodes.
- Each node has at most two children.
- Terminology:
  - Root
  - Subtree
  - Children – parent (ancestor and descendant)
  - Leaf
  - Path (length)
  - Depth of a node
    - The length of the path from the root to the node
  - Height of the tree
    - The depth of the deepest node

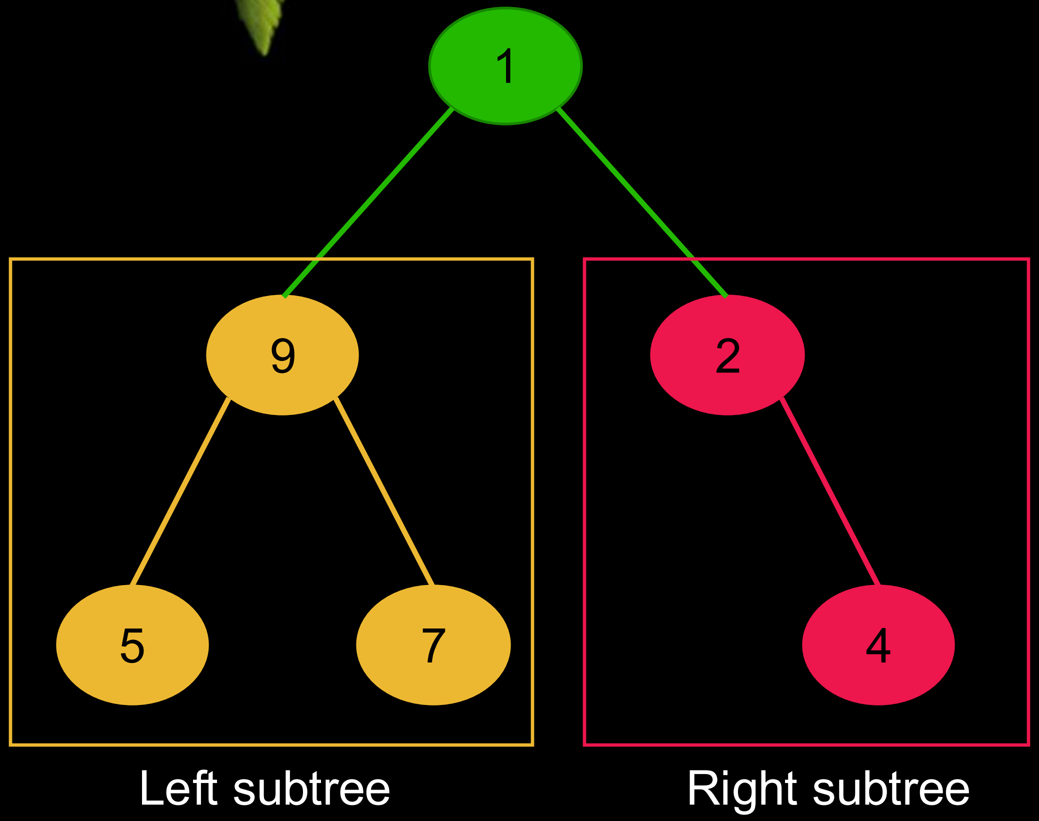


BINARY TREE

- Recursive structure
  - (Left and right) subtrees are also binary trees
  - Many of the operations can be performed recursively.
- Example: the sum of key values

S = root + left + right

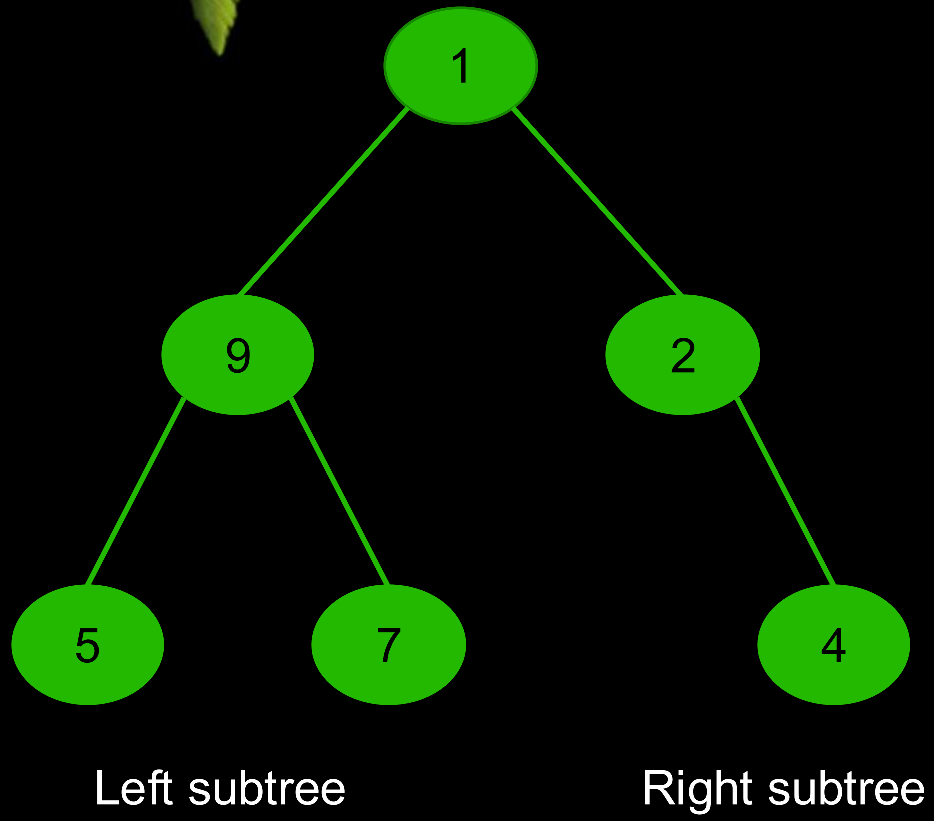
root + left + righ



BINARY TREE

- Tree traversal
  - Visiting all nodes of the tree
  - Preorder traversal
    - visit any given node before we visit its children.

```
procedure preorder(node):
        visit(node)
        preorder(node->left)
        preorder(node->right)
```



1 9 5 7 2 4

BINARY TREE

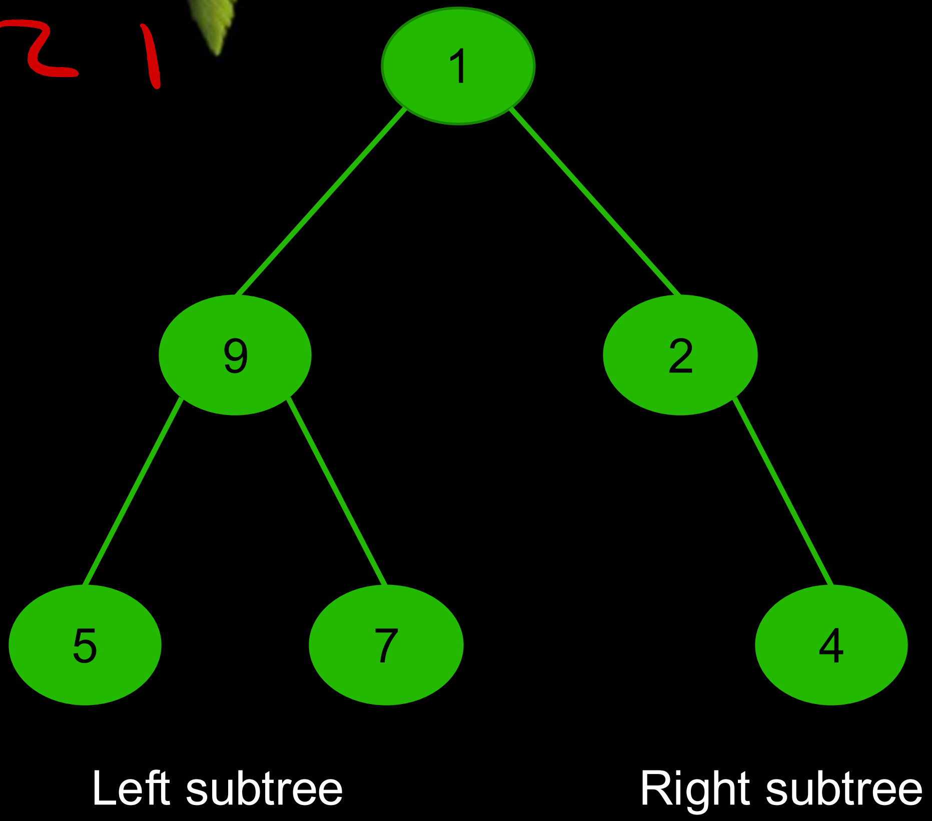
5 7 9 4 2 1

- Tree traversal
  - Postorder traversal
    - visit each node only after we visit its children

```
procedure postorder(node):
        postorder(node->left)
        postorder(node->right)
        visit(node)
```

  - Inorder traversal
    - first visit the left child, then the node, and finally the right child

```
procedure inorder(node):
        inorder(node->left)
        visit(node)
        inorder(node->right)
```



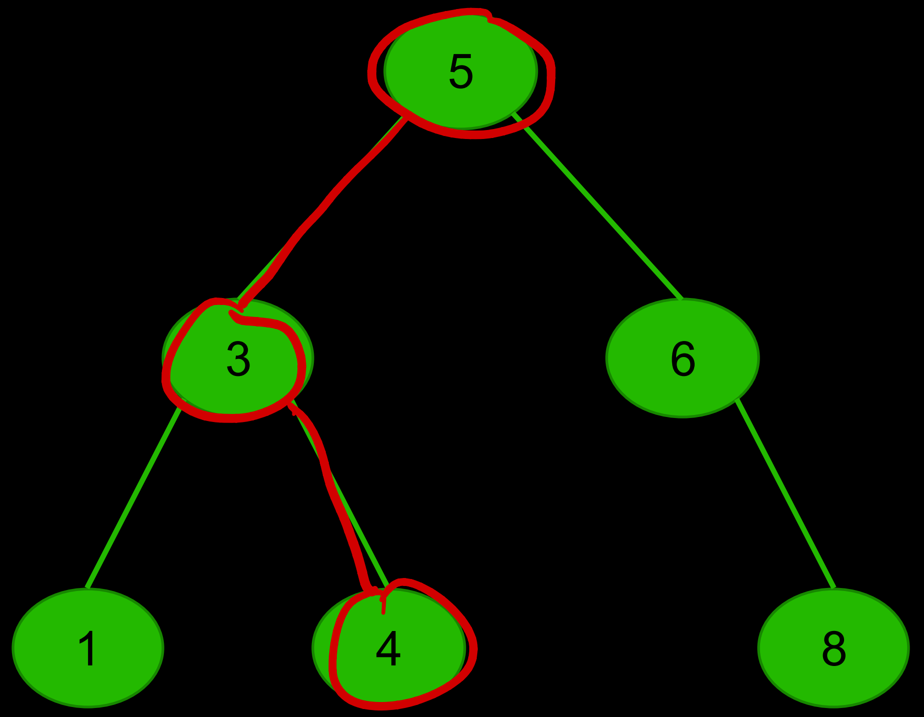
5 9 7 1 2 4

BINARY SEARCH TREE (BST)

search(4)

- All nodes stored in the left subtree of a node whose key value is *K* have key values less than to *K.*
- Basic operations can be implemented efficiently using recursion.
  - Search
  - Insert
  - Remove

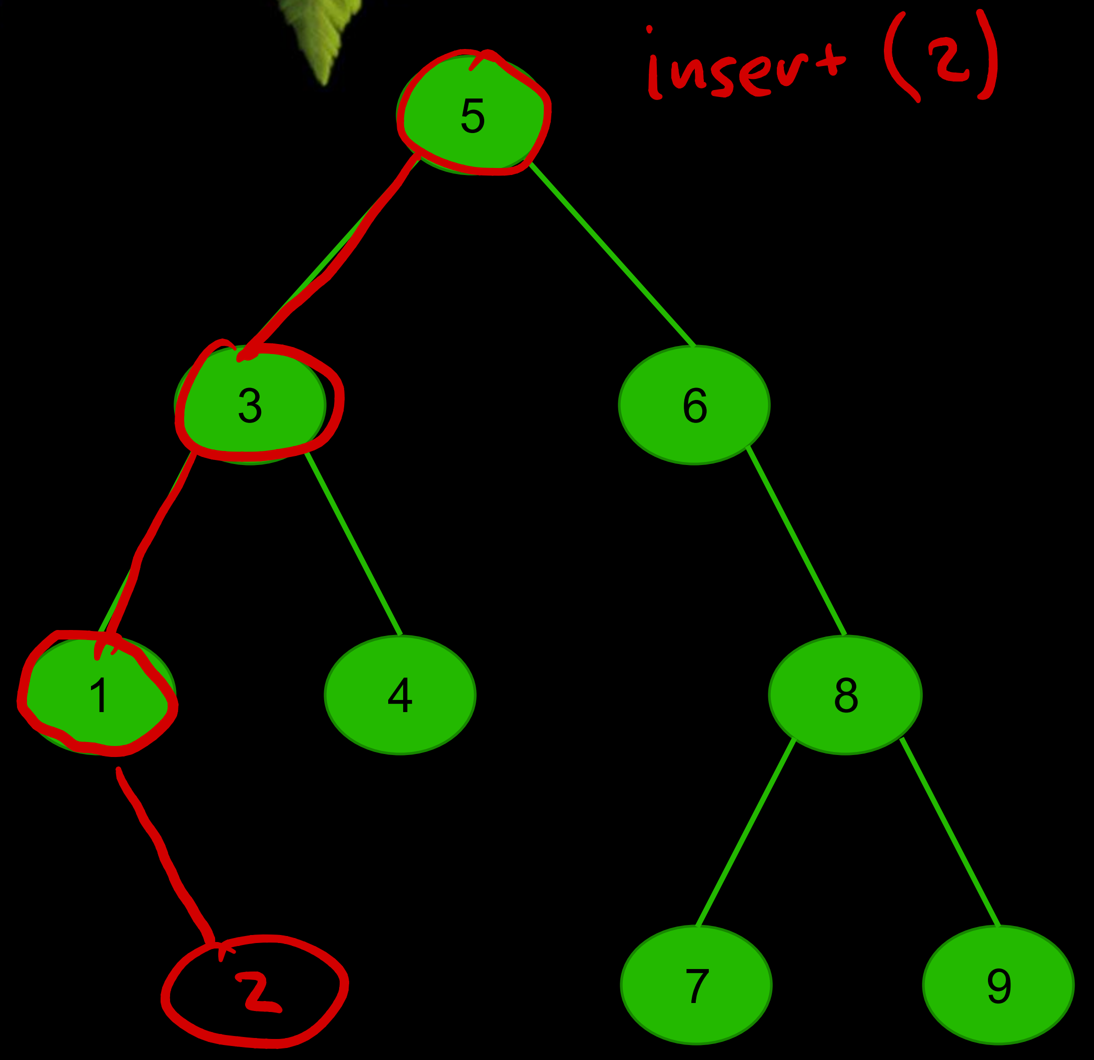
```
procedure search(node, key):
        if key == node.key
                # key found
        elseif key < node.key
                search(node->left, key)
        else search(node->right, key)
```



BINARY SEARCH TREE (BST)

- Insert operation
  - A new key is always inserted at the leaf while maintaining the property of the BST.

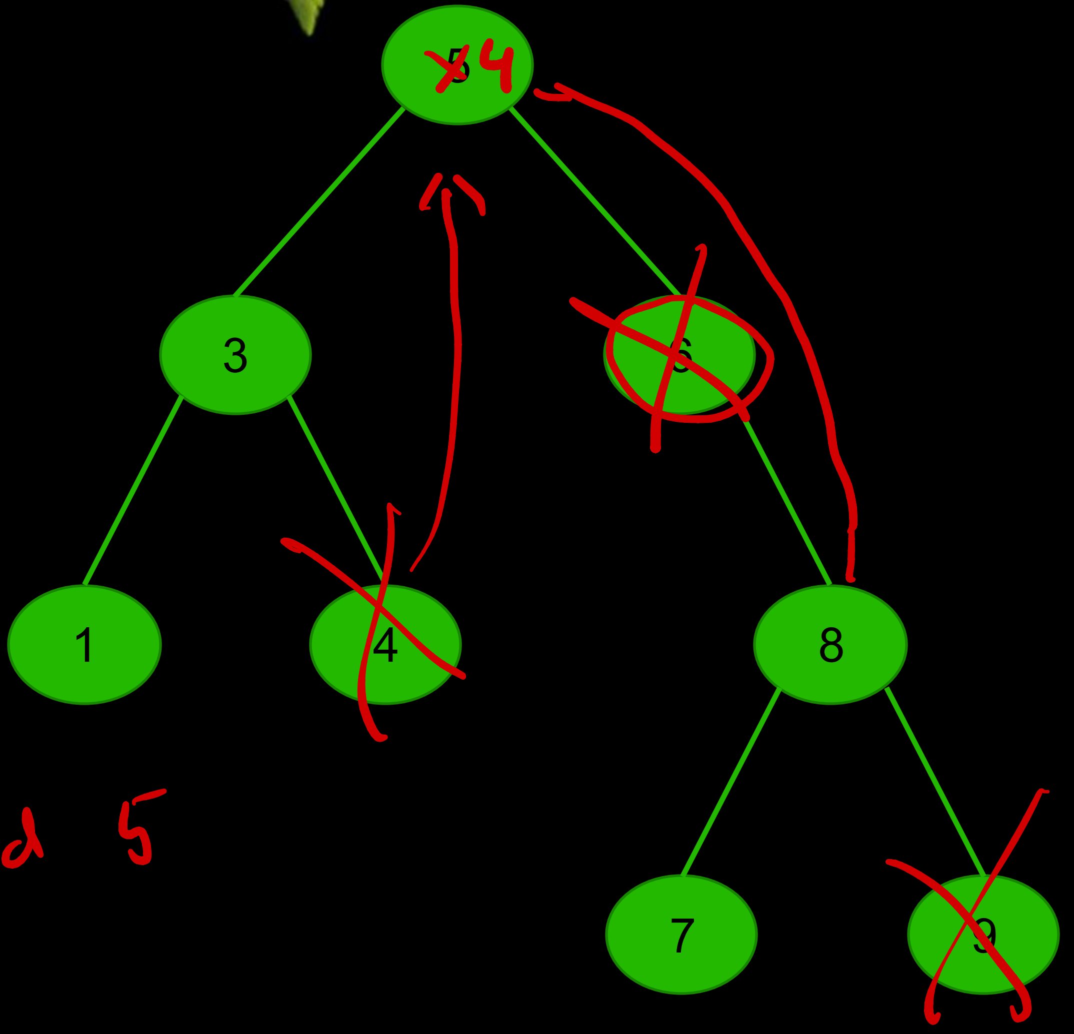
```
procedure insert(node, key):
        if not node
                # create new node
                node.key = key
        else if key == node.key
                # key is already in the tree
                stop
        elseif key < node.key
                search(node->left, key)
        else search(node->right, key)
```



BINARY SEARCH TREE (BST)

- Remove operation
  - If the node (being removed) is a leaf node it can be removed without further operations.
  - If the node has only one child, we can substitute the removed node with the child.
  - If the node has two children, we have 2 options:
    - substitute the node with the one having the largest key value in the left subtree, or
    - substitute the node with the one having the smallest key value in the right subtree.

use this in PA5.1 →



remove 9, 6 and 5

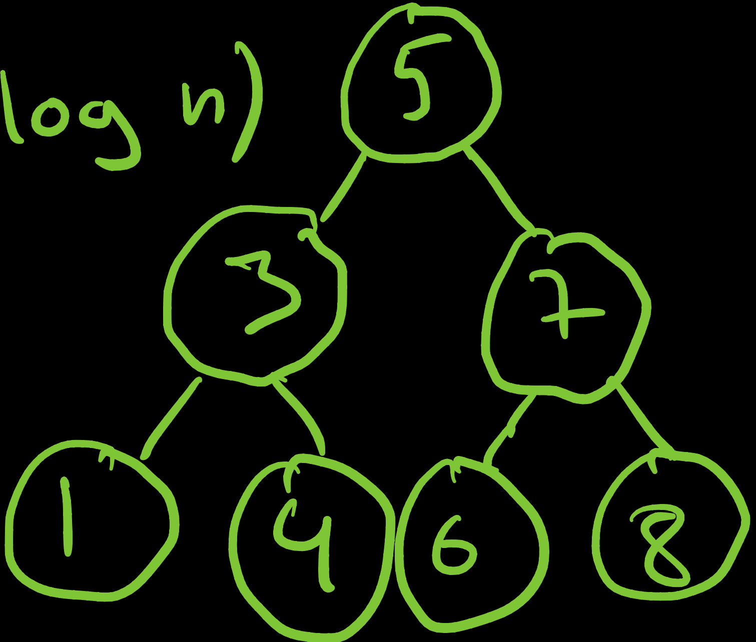
IMPLEMENTING BST IN PYTHON

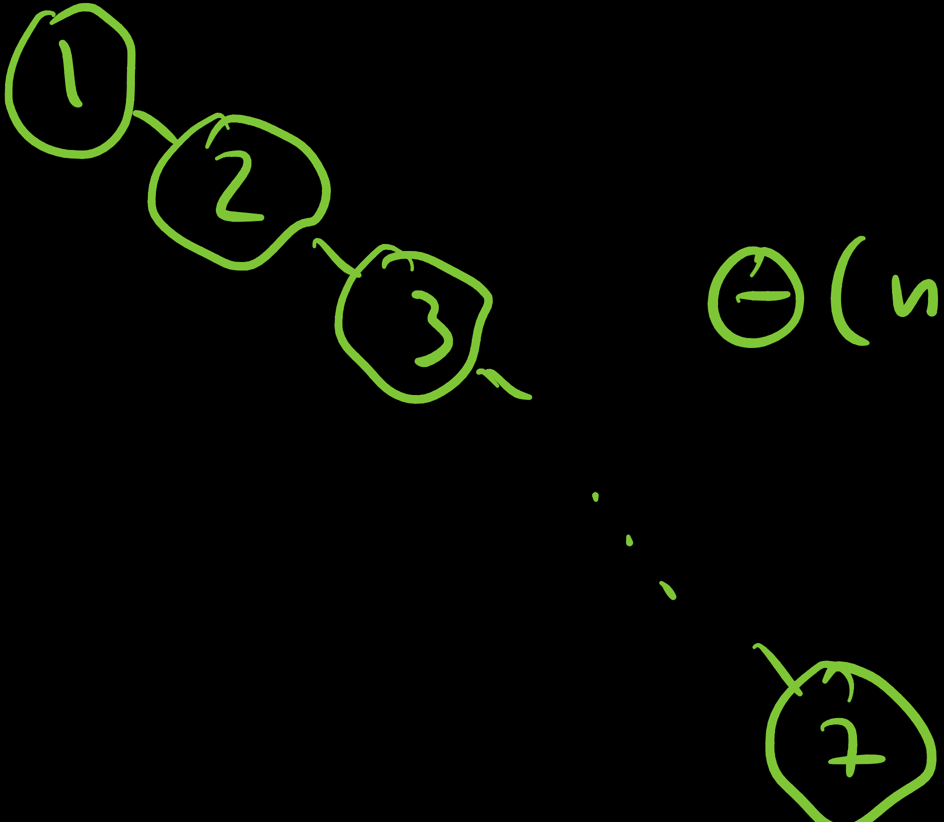
```
class Node:
    def __init__(self, key: int):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def search(self, key):
        return self.search_help(self.root, key)

    def search_help(self, node, key):
        if not node:    # The node is not in the tree
            return False
        elif node.key > key:
            return self.search_help(node.left, key)
        elif node.key < key:
            return self.search_help(node.right, key)
        return True     # The node stores the key
```

BALANCED AND UNBALANCED BST

- Efficiency of the basic operations depend on the height of the tree
- Example: Insert the following keys to a BST:
    1. 5, 3, 7, 1, 6, 8, 4
    2. 1, 2, 3, 4, 5, 6, 7

$\Theta(\log n)$





$\Theta(n)$


LUT
University