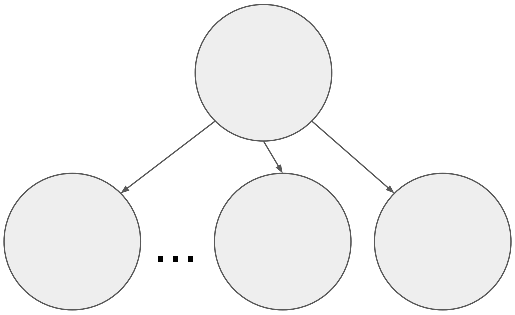
Binary Trees

CPE202

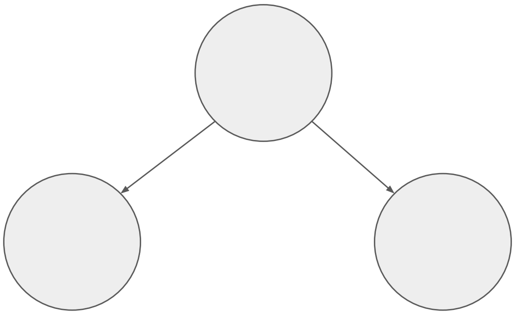
Tree

- Each node has 2 or more fields pointing to other nodes.
  - Linked list is a kind of tree but has only one pointer.



Binary Tree

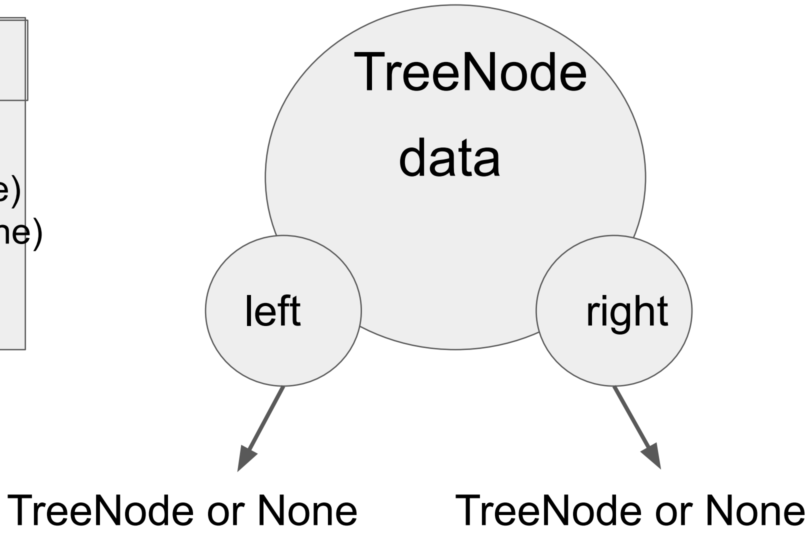
- Each node has 2 fields pointing to 2 other nodes.

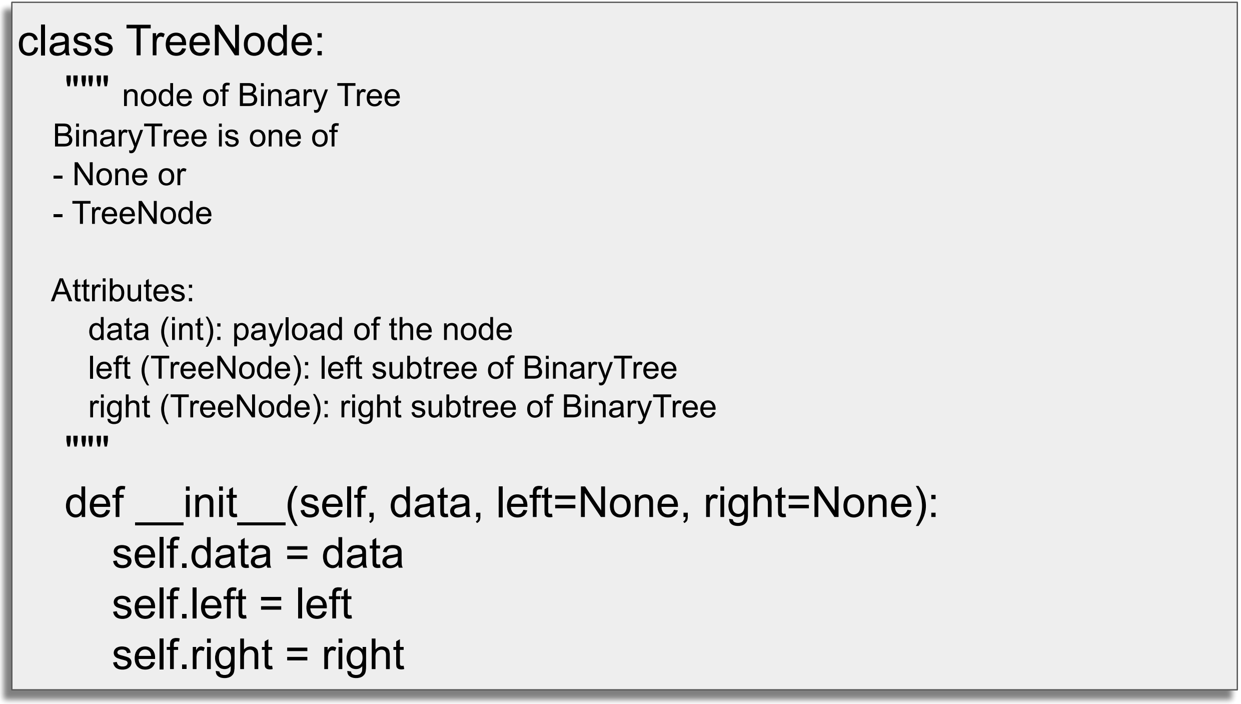


Node for Binary Tree

TreeNode

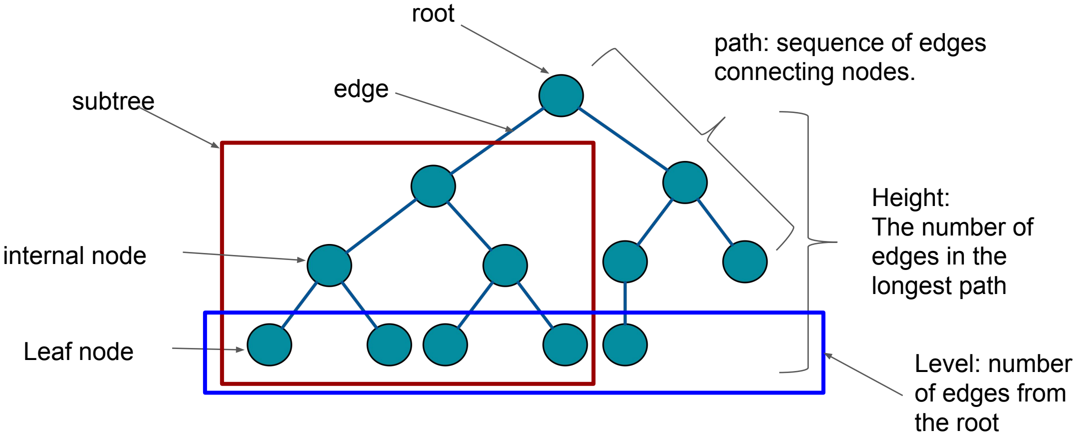
data
left(TreeNode or None)
right(TreeNode or None)





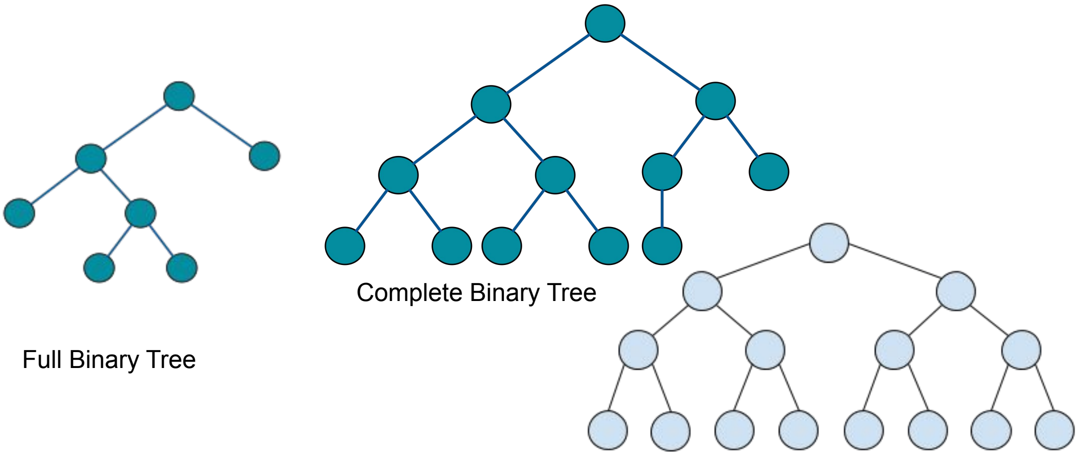
```
class TreeNode:
    """ node of Binary Tree
   BinaryTree is one of
   - None or
   - TreeNode

   Attributes:
      data (int): payload of the node
      left (TreeNode): left subtree of BinaryTree
      right (TreeNode): right subtree of BinaryTree
    """
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

Terminologies of Binary Tree



root
path: sequence of edges connecting nodes.
edge
subtree
Height: The number of edges in the longest path
internal node
Leaf node
Level: number of edges from the root

Examples



Full Binary Tree

Complete Binary Tree

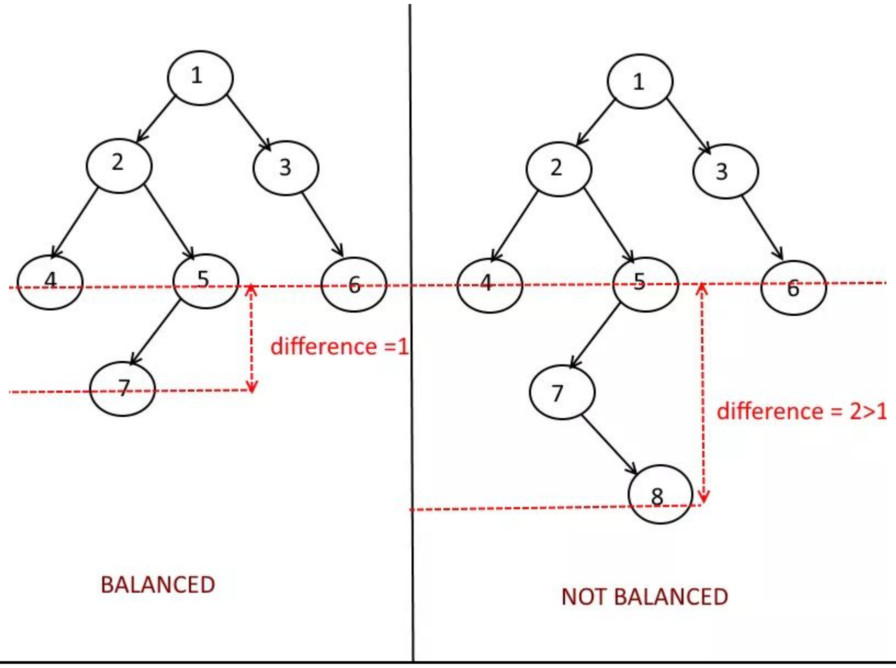
Perfect Binary Tree

Types of Binary Tree

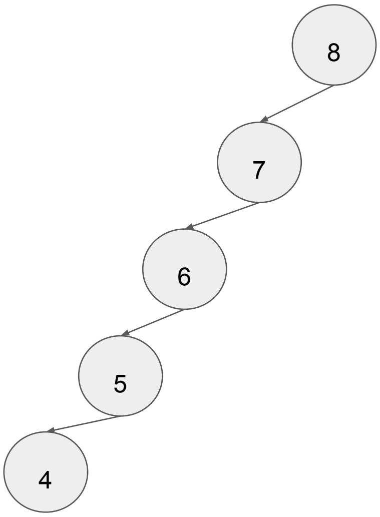
- A full binary tree
  - a binary tree in which every node has either 0 or 2 children.
- A complete binary tree
  - a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes at the last level h.

Types of Binary Tree

- A perfect binary tree
  - a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.
- A balanced binary tree
  - is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1
- A degenerate (or pathological) tree
  - each parent node has only one associated child node. The tree will behave like a linked list.



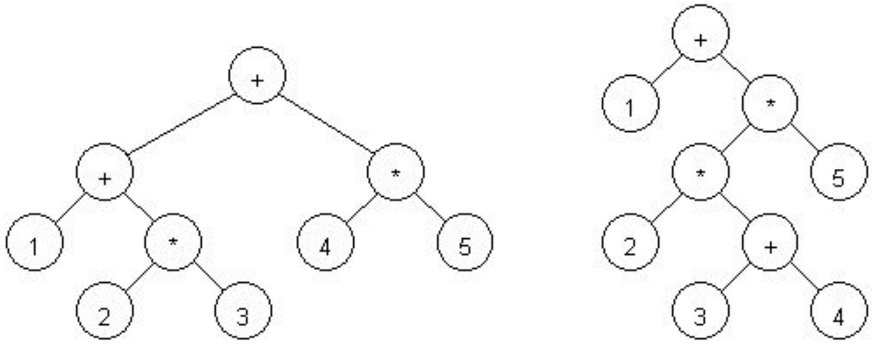
A degenerate (or pathological) tree



Properties of Binary Tree

- The number of nodes n
  - in a full binary tree, n is at least **n=2h+1** and at most **n=(2**(h+1))-1**, where h is the height of the tree. A tree consisting of only a root node has a height of 0.
- The number of leaf nodes l
  - l in a perfect binary tree, is **l=(n+1)/2**.

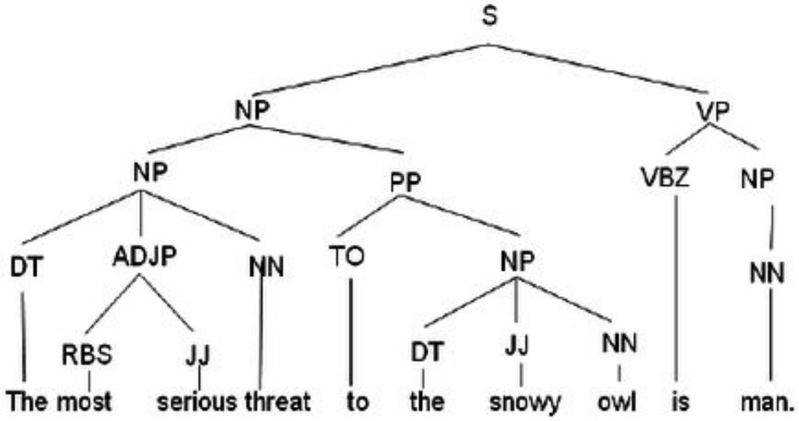
Applications



Parse Tree for Expression 1+ 2 * 3 + 4 * 5

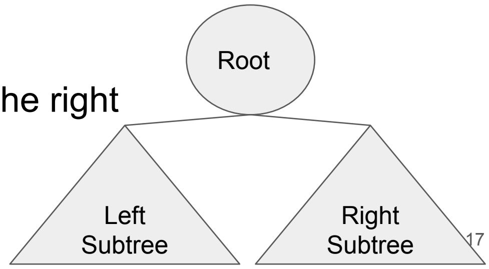
Parse Tree for Expression 1+ 2 * (3 + 4) * 5

More Examples (Not Binary): Parse Tree



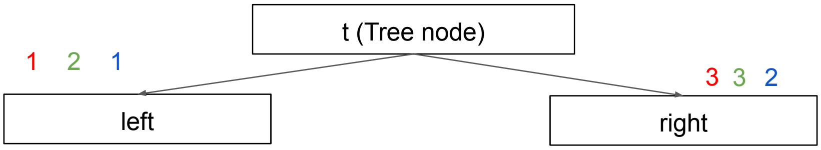
Tree Traversal

- Peorder
  - Visit the root first, then visit its left subtree, and visit the right subtree last
- Inorder
  - Visit the root's left subtree first, then the root, and visit the right subtree last
- Postorder
  - Visit the root's left subtree first, then the right subtree, and visit the root last

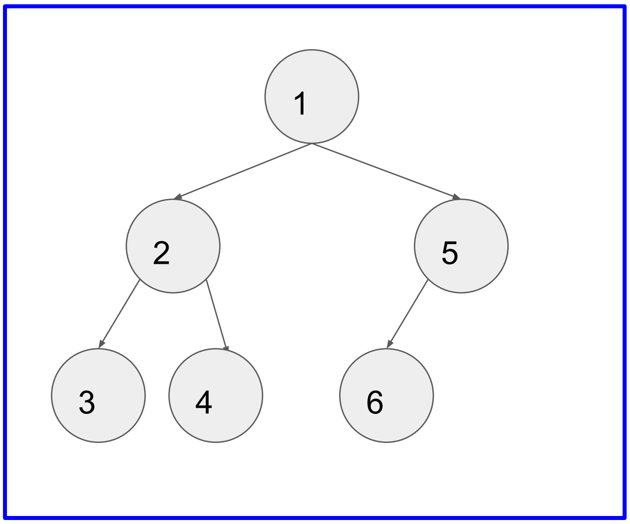


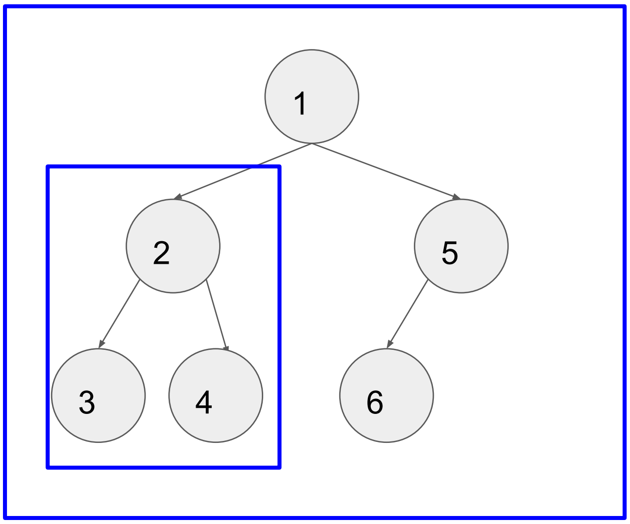
Tree Traversal

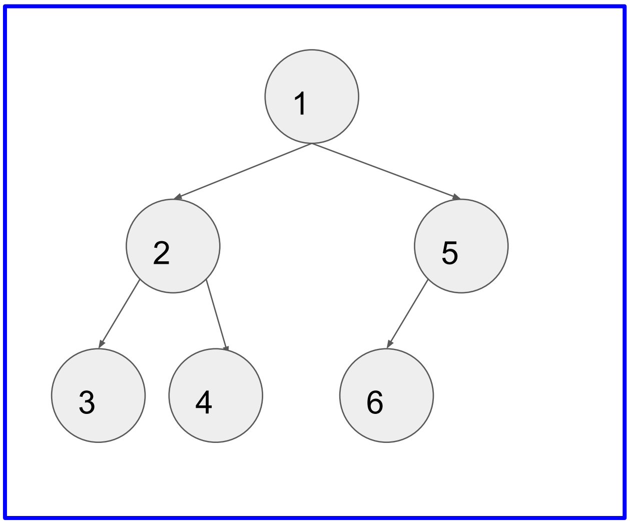
2 1 3 in-order pre-order post-order

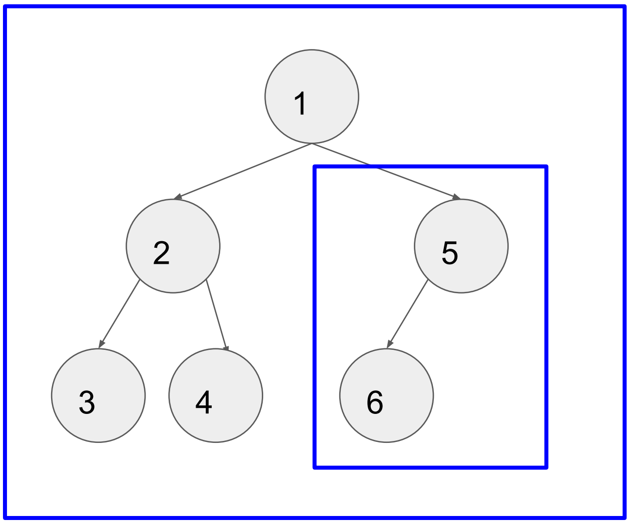


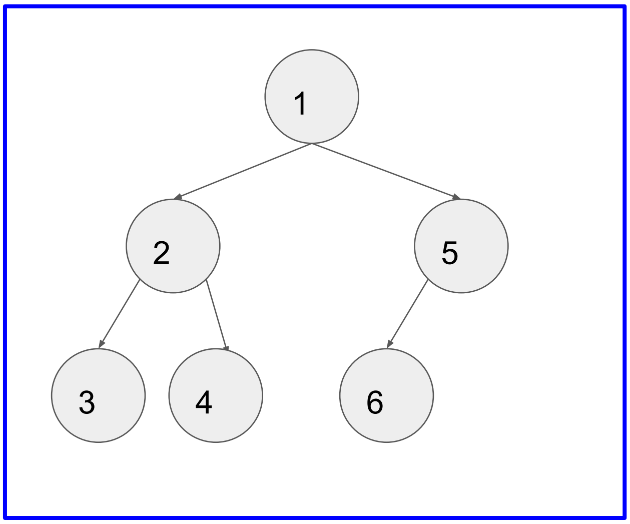
```
def traverse(t):
   if t == None:
      …
   else:  2 1  3       1    2    1               3  3   2
      … t.data … traverse(t.left) … traverse(t.right) ...
```


1
2
5
3
4
6

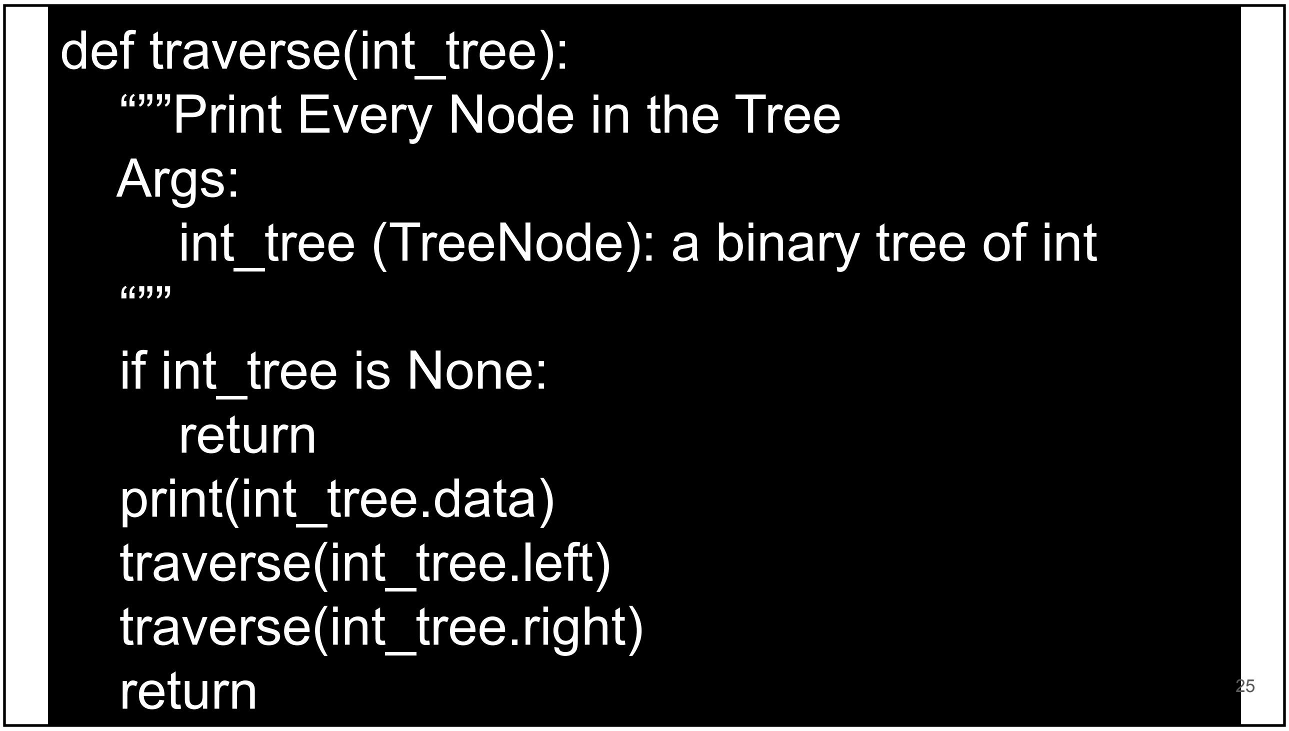

1
2
5
3
4
6


1
2
5
3
4
6

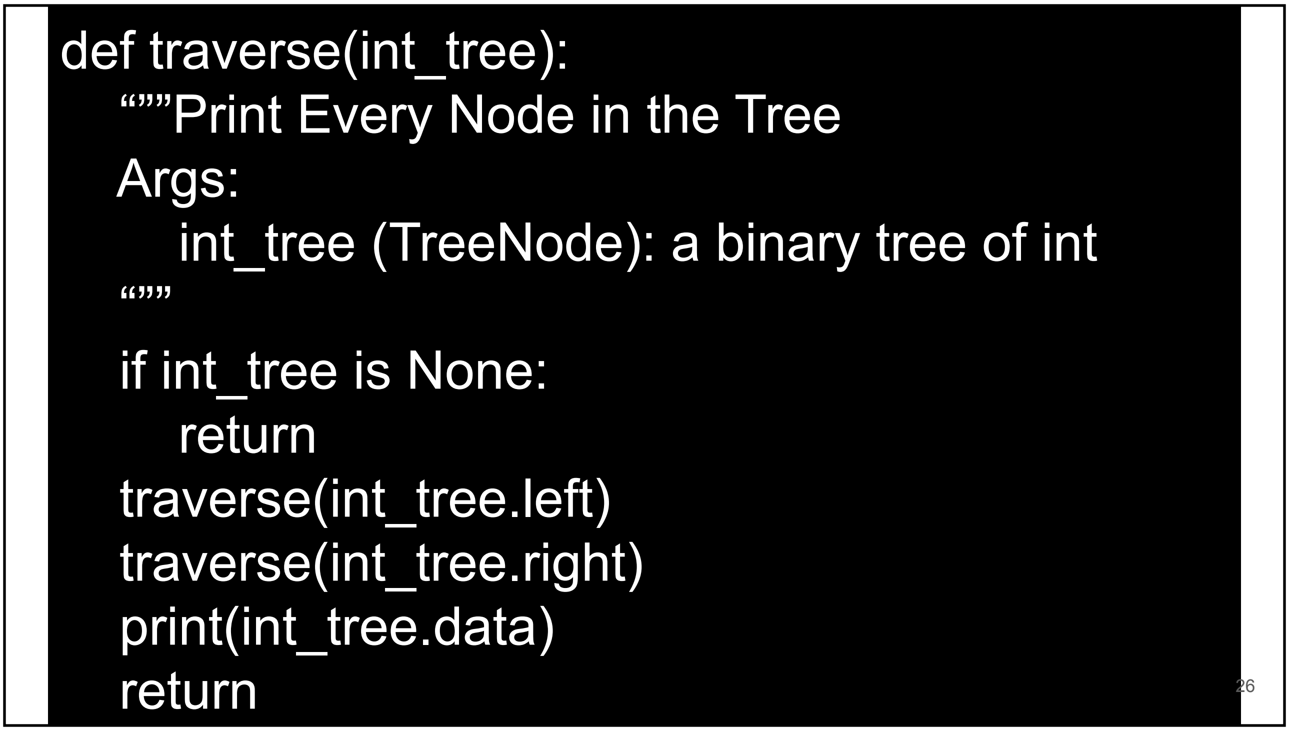

1
2
5
3
4
6



```
def traverse(int_tree):
    “””Print Every Node in the Tree
    Args:
        int_tree (TreeNode): a binary tree of int
    “””
    if int_tree is None:
        return
    traverse(int_tree.left)
    print(int_tree.data)
    traverse(int_tree.right)
    return
```



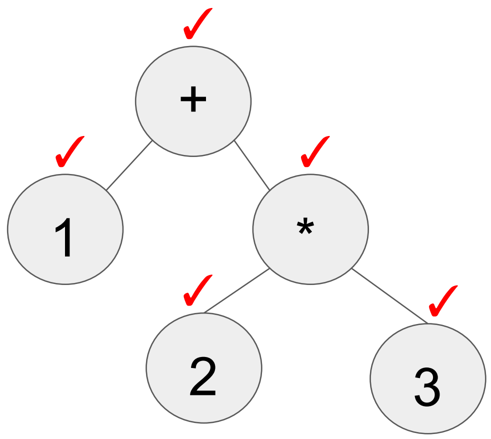
```
def traverse(int_tree):
    “””Print Every Node in the Tree
    Args:
        int_tree (TreeNode): a binary tree of int
    “””
    if int_tree is None:
        return
    print(int_tree.data)
    traverse(int_tree.left)
    traverse(int_tree.right)
    return
```



```
def traverse(int_tree):
    “””Print Every Node in the Tree
    Args:
        int_tree (TreeNode): a binary tree of int
    “””
    if int_tree is None:
        return
    traverse(int_tree.left)
    traverse(int_tree.right)
    print(int_tree.data)
    return
```

Tree Traversal Example

1 + 2 * 3



Pre-order

+	1	*	2	3

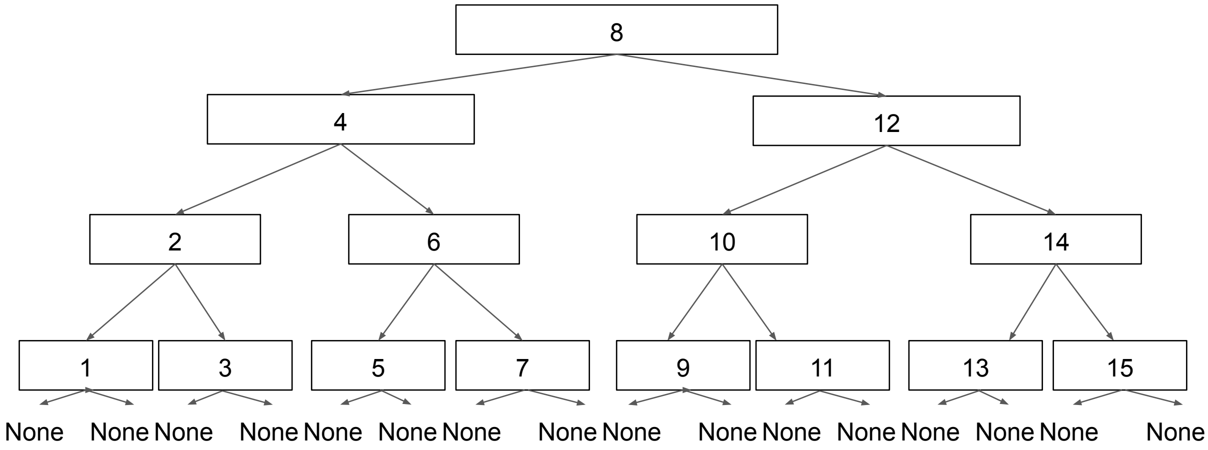
In-order

1	+	2	*	3

Post-order

1	2	3	*	+

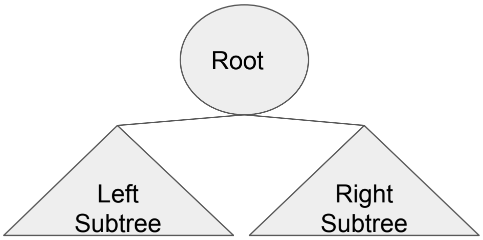
Binary Search Tree (BST)



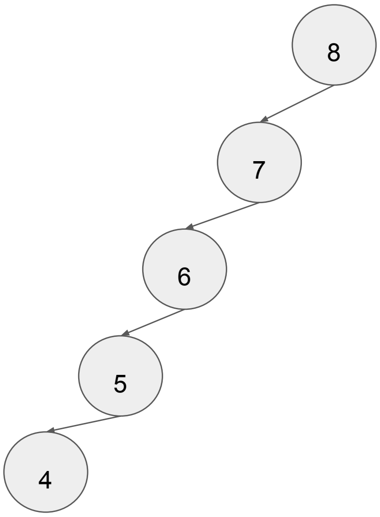
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Properties of BST

- Binary Search Tree,
  - Values in left subtree are smaller than the root’s
  - Values in right subtree are larger than the root’s
- When the tree is balanced, the time complexity of search is O(logN)



BST Not Balanced



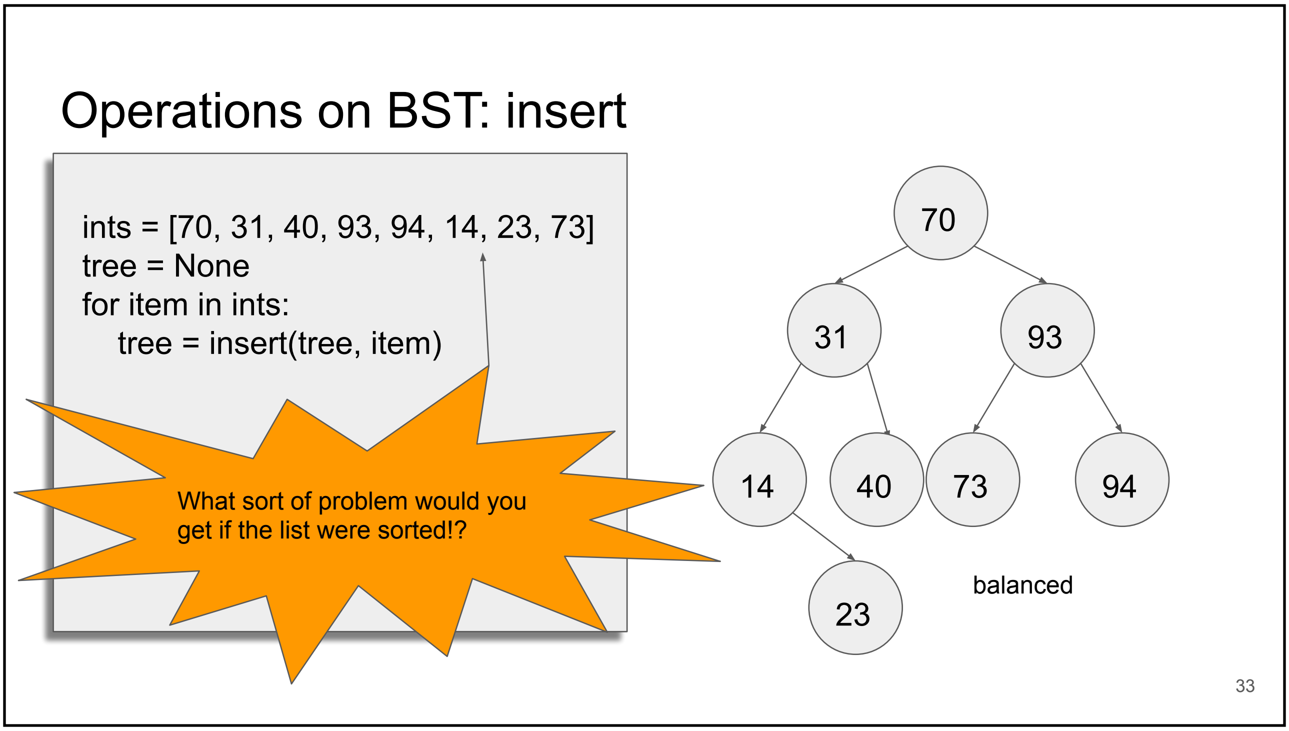
BST Data Definition

```
class BSTNode:
    """ node of Binary Search Tree

    BinarySearchTree is one of
    - None or
    - BSTNode

    Attributes:
        data (int): payload of the node
        left (BSTNode): left subtree of BinarySearchTree
        right (BSTNode): right subtree of BinarySearchTree
    """
    def __init__(self, data, left, right):
        self.data = data
        self.left = left
        self.right = right
```

```
    def __eq__(self, other):
        return isinstance(other, type(self))\
                and self.data == other.data\
                and self.left == other.left\
                and self.right == other.right

    def __repr__(self):
        return "BSTNode{data: %s, left: %s, right: %s}"\
                % (self.data, self.left, self.right)
```

Operations on BST: insert

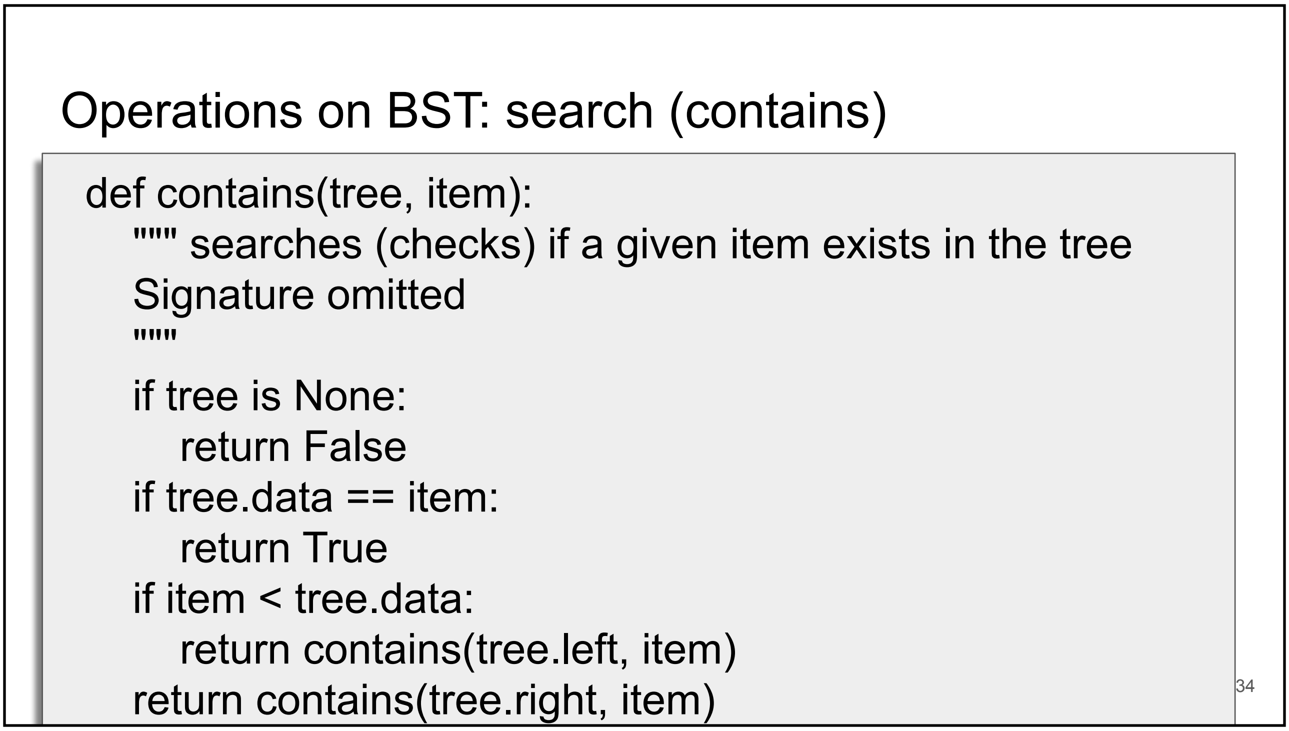
```
def insert(tree, item):
    """Docstring omitted
    """
    if tree is None:
        return BSTNode(item, None, None)
    if item < tree.data:
        tree.left = insert(tree.left, item)
    else:
        tree.right = insert(tree.right, item)
    return tree
```



Operations on BST: insert

```
ints = [70, 31, 40, 93, 94, 14, 23, 73]
tree = None
for item in ints:
    tree = insert(tree, item)
```

What sort of problem would you get if the list were sorted!?



Operations on BST: search (contains)

```
def contains(tree, item):
    """ searches (checks) if a given item exists in the tree
    Signature omitted
    """
    if tree is None:
        return False
    if tree.data == item:
        return True
    if item < tree.data:
        return contains(tree.left, item)
    return contains(tree.right, item)
```

Operations on BST: delete

Template

1. If Tree is None, raise error.
2. If found:
   a. If the node has no children, just remove the node by returning None
   b. If it has a child, make the child become the node's parent's child by returning the child
   c. If it has two children, search for the next larger value node on the right subtree and make it replace the node.
      i. If the replacement has no children, its parent needs to abandon it.
      ii. else, the replacement's child must be adopted by its parent.
      iii. Return the replacement
3. If the target value is smaller, continue on to the left subtree.
4. Else, continue on to the right subtree
5. Return the tree

Operations on BST: delete 1/3

```
def delete(tree, item):
    """delete a given item in the tree

    Args:
        tree (BSTNode): BinarySearchTree
        item (int): the item to be deleted

    Returns:
        BSTNode: the root of a BinarySearchTree
    """
    #the base case
    if tree is None:
        raise KeyError("%s not found in this tree!")
```

Operations on BST: delete 2/3

```
#found
if tree.data == item:
    #case 1: a node with no children
    if tree.left is None and tree.right is None:
        return None
    #case 2: a node with one child
    if tree.left is None:
        return tree.right
    if tree.right is None:
        return tree.left
    #case 3: a node with two children
    #we need a special care when both children are present
    replacement, tree.right = get_replacement(tree.right)
    #replace children
    replacement.left = tree.left
    replacement.right = tree.right
    return replacement
```

Operations on BST: delete 3/3

```
#not the current node. continue the search
if tree.data > item:
    #go left
    tree.left = delete(tree.left, item)
else:
    #go right
    tree.right = delete(tree.right, item)
#return the root of a tree or subtree
return tree
```

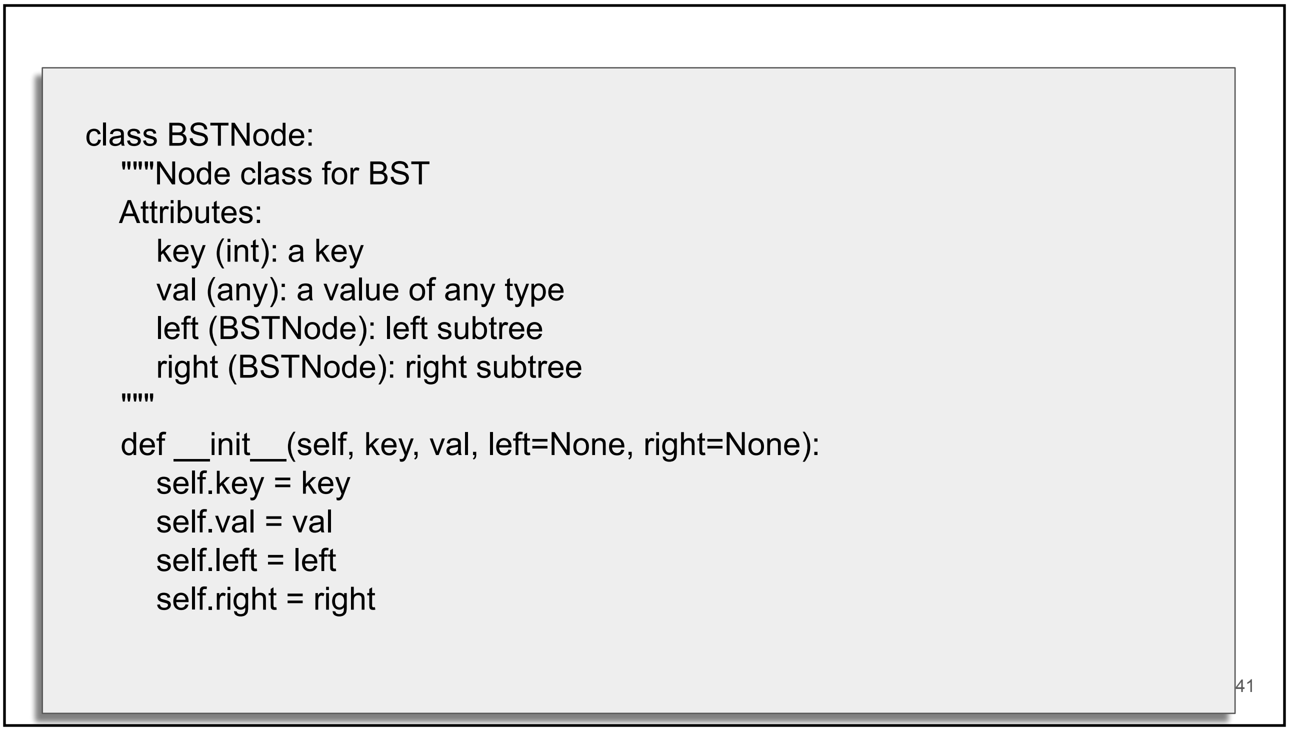
Helper function for getting a replacement node

```
def get_replacement(current):
    """a helper function to get a replacement node for to be deleted
    A replacement node is the node with next larger value.
    Args:
        current (BSTNode): current node
    Returns:
        BSTNode: the replacement node
        BSTNode: the new current node
    """
    #base case: found the target node
    if current.left is None:
        child = current.right
        #abondon the child
        current.right = None
        #return the node and ask the parent to adopt the child
        return current, child
    #otherwise continue the search on the left
    replacement, current.left = get_replacement(current.left)
    return replacement, current
```

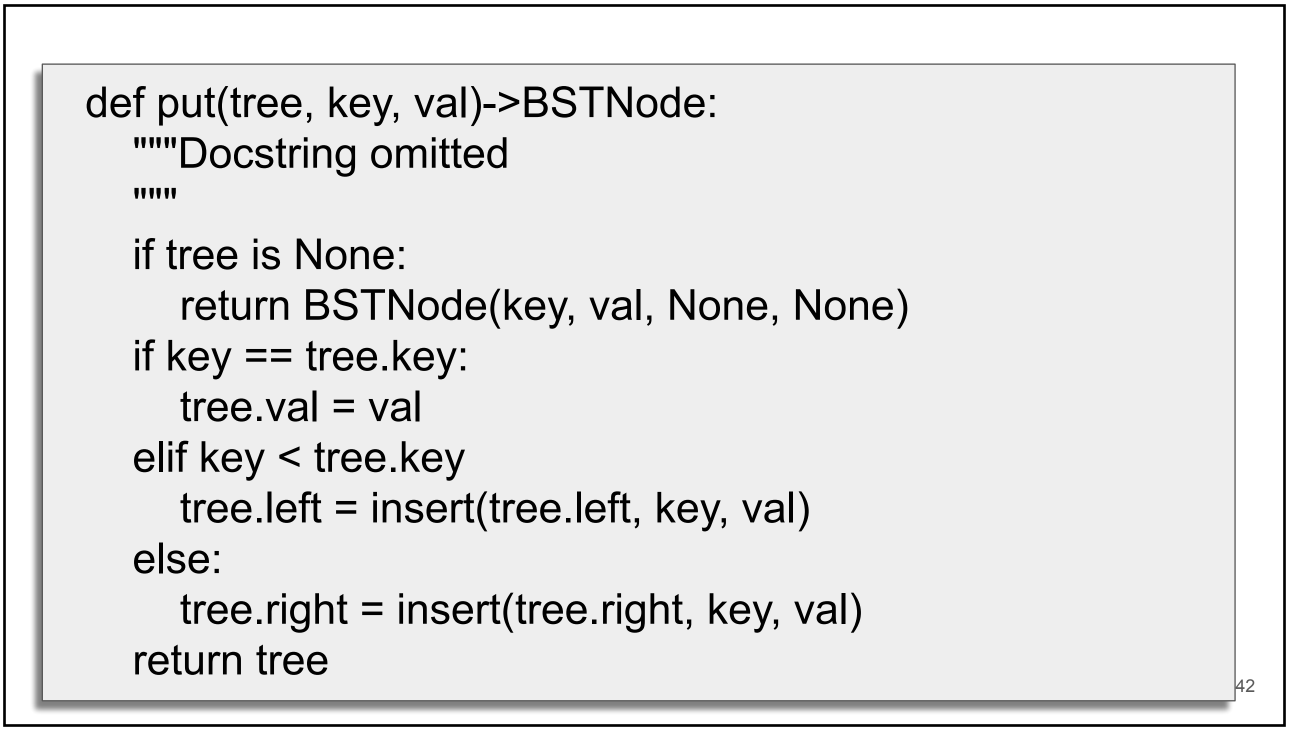
MAP Abstract Data Type

Binary Search Tree is used for implementing MAP ADT

- Map() Create a new, empty map. Map associates a key with a value.
- put(key,val) Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- get(key) Given a key, return the value stored in the map or None otherwise.
- delete(key) Delete the key-value pair from the map.
- size() Return the number of key-value pairs stored in the map.
- contains(key) Return True for a statement of the form key in map, if the given key is in the map.



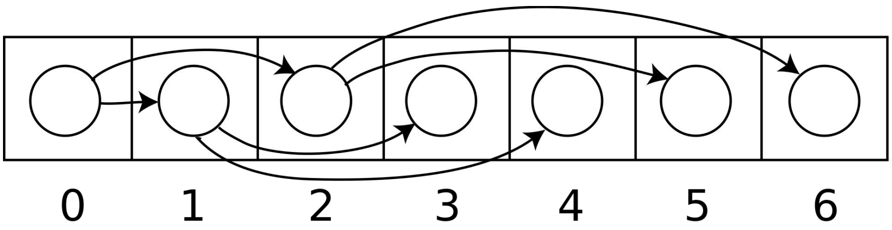
```
class BSTNode:
    """Node class for BST
    Attributes:
        key (int): a key
        val (any): a value of any type
        left (BSTNode): left subtree
        right (BSTNode): right subtree
    """
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```



```
def put(tree, key, val)->BSTNode:
    """Docstring omitted
    """
    if tree is None:
        return BSTNode(key, val, None, None)
    if key == tree.key:
        tree.val = val
    elif key < tree.key
        tree.left = insert(tree.left, key, val)
    else:
        tree.right = insert(tree.right, key, val)
    return tree
```

```
def get(tree, key)->any:
    """Docstring omitted
    """
    if tree is None:
        raise KeyError()
    if key == tree.key:
        return tree.val
    if key < tree.key
        return get(tree.left, key)
    return get(tree.right, key)
```

Implementing Tree with Array



N’s children are at 2N + 1 and 2N + 2
For example, 0’s children is at 1 and 2