**KATHMANDU UNIVERSITY**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**

****

**Lab Sheet 3**

**Binary Search Tree**

**Algorithm and Complexity**

**[Course Code: COMP 314]**

**Submitted by:**

Sajag Silwal

Roll no: 48

**Submitted to:**

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

**Submission Date:** 16-04-2022

# Implementation of Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

1. The left subtree of a node contains only nodes with keys lesser than the node’s key.
2. The right subtree of a node contains only nodes with keys greater than the node’s key.
3. The left and right subtree each must also be a binary search tree.



Figure : Binary Tree

## Creating class for Node and binary Search tree

A class representing binary search tree and its nodes is created.

**Source Code:**

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.\_size = 0

        self.root = None

    class Node:

        def \_\_init\_\_(self,key,value):

            self.key = key

            self.value = value

            self.left = None

            self.right = None

## Addition of Node:

**Source Code:**

    def add(self,key,value):

        if self.root == None:

            self.root = self.Node(key, value)

            self.\_size += 1

        else:

            self.addRec(self.root,key,value)

    def addRec(self,subtree:Node,key,value):

        if subtree.key > key:

            if subtree.left == None:

                subtree.left = self.Node(key,value)

                self.\_size += 1

            else:

                self.addRec(subtree.left,key,value)

        elif subtree.key < key:

            if subtree.right == None:

                subtree.right = self.Node(key,value)

                self.\_size += 1

            else:

                self.addRec(subtree.right,key,value)

## Searching an element

**Source Code:**

    def search(self, key):

        return self.search\_recursion(self.root,key)

    def searchRec(self, subtree:Node, key):

        if subtree == None:

            return False

        elif subtree.key == key:

            return subtree.value

        elif subtree.key < key:

            return self.searchRec(subtree.right,key)

        else:

            return self.searchRec(subtree.left,key)

## Traversal

The Implementation of Inorder, PreOrder, and PostOrder Traversal is given below:

**Source Code:**

    def inorder\_walk(self):

        visited\_node = []

        self.inorder\_walkRec(self.root, visited\_node)

        return visited\_node

    def inorder\_walkRec(self,subtree:Node,visited\_node):

        if subtree:

            self.inorder\_walkRec(subtree.left,visited\_node)

            visited\_node.append(subtree.key)

            self.inorder\_walkRec(subtree.right,visited\_node)

    def preorder\_walk(self):

        visited\_node = []

        self.preorder\_walkRec(self.root, visited\_node)

        return visited\_node

    def preorder\_walkRec(self,subtree:Node,visited\_node):

        if subtree:

            visited\_node.append(subtree.key)

            self.preorder\_walkRec(subtree.left,visited\_node)

            self.preorder\_walkRec(subtree.right,visited\_node)

    def postorder\_walk(self):

        visited\_node = []

        self.postorder\_walkRec(self.root, visited\_node)

        return visited\_node

    def postorder\_walkRec(self,subtree:Node,visited\_node):

        if subtree:

            self.postorder\_walkRec(subtree.left,visited\_node)

            self.postorder\_walkRec(subtree.right,visited\_node)

            visited\_node.append(subtree.key)

## Smallest Element

The source code for finding the smallest element in the binary search tree is given below:

    def smallest(self):

        return self.smallestRec(self.root)

    def smallestRec(self,subtree:Node):

        if subtree.left == None:

            return (subtree.key, subtree.value)

        else:

            return self.smallestRec(subtree.left)

## Largest Element

The source code for finding the largest element in the binary search tree is given below:

    def largest(self):

        return self.largestRec(self.root)

    def largestRec(self,subtree:Node):

        if subtree.right == None:

            return (subtree.key, subtree.value)

        else:

            return self.largestRec(subtree.right)

## Remove

The source Code to remove a node from the Binary Search Tree is given Below:

    def remove(self, key):

        if self.search(key):

            self.root = self.removeRec(self.root, key)

            self.\_size -= 1

        else:

            return self.search(key)

    def removeRec(self, subtree, key):

        if subtree is None:

            return subtree

        elif key < subtree.key:

            subtree.left = self.removeRec(subtree.left, key)

            return subtree

        elif key > subtree.key:

            subtree.right = self.removeRec(subtree.right, key)

            return subtree

        else:

            if subtree.left is None and subtree.right is None:

                return None

            elif subtree.left is None or subtree.right is None:

                if subtree.left is not None:

                    return subtree.left

                else:

                    return subtree.right

            else:

                successor = self.largestRec(subtree.left)

                subtree.key = successor[0]

                subtree.value = successor[1]

                subtree.left = self.removeRec(subtree.left, successor[0])

                return subtree

## Output:

The output of the source code above using the test case provided is given below:

