Binary Search Tree

package main

import "fmt"

type Node struct {

data int

left \*Node

right \*Node

}

type BST struct {

root \*Node

}

func (t \*BST) Insert(value int) {

newNode := &Node{data: value}

if t.root == nil {

t.root = newNode

return

}

temp := t.root

for {

if temp.data == value {

return

}

if temp.data > value {

if temp.left == nil {

temp.left = newNode

return

}

temp = temp.left

}

if temp.data < value {

if temp.right == nil {

temp.right = newNode

return

}

temp = temp.right

}

}

}

func (t \*BST) Contains(value int) bool {

temp := t.root

for temp != nil {

if value == temp.data {

return true

} else if value < temp.data {

temp = temp.left

} else {

temp = temp.right

}

}

return false

}

func (t \*BST) Remove(value int) {

t.removeHelper(value, t.root, nil)

}

func (t \*BST) removeHelper(value int, currentNode, parentNode \*Node) {

for currentNode != nil {

if value < currentNode.data {

parentNode = currentNode

currentNode = currentNode.left

} else if value > currentNode.data {

parentNode = currentNode

currentNode = currentNode.right

} else {

if currentNode.left != nil && currentNode.right != nil {

currentNode.data = t.GetMinValue(currentNode)

t.removeHelper(currentNode.data, currentNode.left, currentNode)

} else {

if parentNode == nil {

if currentNode.left == nil {

t.root = currentNode.right

} else {

t.root = currentNode.left

}

} else {

if parentNode.left == currentNode {

if currentNode.left == nil {

parentNode.left = currentNode.right

} else {

parentNode.left = currentNode.left

}

} else {

if currentNode.left == nil {

parentNode.right = currentNode.right

} else {

parentNode.right = currentNode.left

}

}

}

}

return

}

}

fmt.Println("not a value in the tree")

}

func (t \*BST) GetMinValue(currentNode \*Node) int {

if currentNode.left == nil {

return currentNode.data

}

return t.GetMinValue(currentNode.left)

}

func (t \*BST) InOrder() []int {

arr := []int{}

t.inOrderHelper(t.root, &arr)

return arr

}

func (t \*BST) inOrderHelper(node \*Node, arr \*[]int) {

// arr := []int{}

if node != nil {

t.inOrderHelper(node.left, arr)

\*arr = append(\*arr, node.data)

t.inOrderHelper(node.right, arr)

}

}

func (t \*BST) PreOrder() {

t.preOrderHelper(t.root)

}

func (t \*BST) preOrderHelper(node \*Node) {

if node != nil {

fmt.Printf("%d ", node.data)

t.preOrderHelper(node.left)

t.preOrderHelper(node.right)

}

}

func (t \*BST) PostOrder() {

t.postOrderHelper(t.root)

}

func (t \*BST) postOrderHelper(node \*Node) {

if node != nil {

t.postOrderHelper(node.left)

t.postOrderHelper(node.right)

fmt.Printf("%d ", node.data)

}

}

func (t \*BST) ValidBST() bool {

arr := t.InOrder()

for i := 0; i < len(arr)-1; i++ {

if arr[i] > arr[i+1] {

return false

}

}

return true

}

func (t \*BST) FindHeight(node \*Node) int {

if node == nil {

return -1

}

left := t.FindHeight(node.left)

right := t.FindHeight(node.right)

max := 0

if left > right {

max = left

} else {

max = right

}

return max + 1

}

var flag = 1

func (t \*BST) IsBalanced(node \*Node) bool {

t.IsBalanceHelper(node)

return flag == 1

}

func (t \*BST) IsBalanceHelper(node \*Node) int {

if node == nil {

return -1

}

left := t.IsBalanceHelper(node.left)

right := t.IsBalanceHelper(node.right)

if left-right > 1 || right-left > 1 {

flag = 0

}

max := 0

if left > right {

max = left

} else {

max = right

}

return max + 1

}

func main() {

t := &BST{}

t.Insert(2)

t.Insert(1)

t.Insert(9)

t.Insert(8)

t.Insert(7)

t.Insert(4)

t.Insert(3)

t.Insert(5)

fmt.Println(t.Contains(11))

t.Remove(2)

arr := t.InOrder()

fmt.Println(arr)

fmt.Println(t.FindHeight(t.root))

fmt.Println(t.IsBalanced(t.root))

fmt.Println(t.ValidBST())

t.PreOrder()

t.PostOrder()

}