МИНИСТРЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОУ НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ИМ. Р.Е. АЛЕКСЕЕВА

ИНСТИТУТ ЭКОНОМИКИ И УПРАВЛЕНИЯ

КАФЕДРА " Прикладная математика и информатика"

Дисциплина "Структуры данных"

Отчет

по лабораторной работе № 10

Выполнил: Иванов Михаил Егорович

студент группы: 21 САИ

Проверил: Моисеев Антон Евгеньевич

Нижний Новгород

2023

**Цель работы:** реализовать двоичное дерево.

**Практическая часть.**

Двоичное дерево - это структура данных, в которой каждый узел имеет не более двух потомков, называемых левым и правым. Двоичное дерево позволяет эффективно выполнять поиск, вставку и удаление элементов, если оно сбалансировано.

Для реализации двоичного дерева была создана структура Node, которая представляет узел дерева, и структура Tree, которая представляет само дерево. Структура Node содержит поле data для хранения данных типа int и поля left и right для хранения указателей на левого и правого потомка типа \*Node. Структура Tree содержит поле root для хранения указателя на корневой узел дерева типа \*Node.

Для работы с двоичным деревом были созданы следующие методы:

* Метод insert для добавления нового узла в дерево. Метод принимает данные для нового узла и создает его с помощью оператора &. Затем метод проверяет, пусто ли дерево, и если да, то делает новый узел корневым. Иначе метод ищет место для вставки нового узла в дереве, начиная с корня и сравнивая данные нового узла с данными текущего узла. Если данные нового узла меньше данных текущего узла, то метод переходит к левому потомку, иначе к правому. Когда место для вставки найдено, метод связывает новый узел с родительским узлом.
* Метод search для поиска узла в дереве по заданным данным и возвращения его и его родителя. Метод принимает данные для поиска и проверяет, пусто ли дерево. Если да, то метод возвращает nil, nil. Иначе метод ищет узел в дереве, начиная с корня и сравнивая данные для поиска с данными текущего узла. Если данные совпадают, то метод возвращает текущий узел и его родителя. Если данные для поиска меньше данных текущего узла, то метод переходит к левому потомку, иначе к правому. Если узел не найден, то метод возвращает nil, nil.
* Метод remove для удаления узла из дерева по заданным данным. Метод принимает данные для удаления и ищет узел и его родителя в дереве по этим данным с помощью метода search. Если узел не найден, то метод ничего не делает. Иначе метод определяет, является ли удаляемый узел левым или правым потомком своего родителя. Затем метод рассматривает три случая:
  + Если удаляемый узел не имеет потомков, то метод просто отсоединяет его от родителя.
  + Если удаляемый узел имеет только одного потомка (левого или правого), то метод заменяет его на этого потомка, связывая его с родителем.
  + Если удаляемый узел имеет обоих потомков, то метод находит минимальный элемент в правом поддереве (самый левый элемент) и его родителя. Затем метод копирует данные минимального элемента в удаляемый узел и удаляет минимальный элемент из правого поддерева, связывая его родителя с его правым потомком (если он есть).
* Метод print для вывода элементов дерева на экран. Метод использует вспомогательную функцию printTree, которая принимает узел, отступ и символ связи и рекурсивно выводит данные узла и его потомков в древовидной форме.

**Код:**

package main  
  
import "fmt"  
  
// Определяем структуру Node, которая представляет узел двоичного дерева  
type Node struct {  
 data int // Данные, хранящиеся в узле  
 left \*Node // Указатель на левого потомка  
 right \*Node // Указатель на правого потомка  
}  
  
// Определяем структуру Tree, которая представляет двоичное дерево  
type Tree struct {  
 root \*Node // Указатель на корневой узел дерева  
}  
  
// Создаем метод insert для добавления нового узла в дерево  
func (t \*Tree) insert(data int) {  
 // Создаем новый узел с данными  
 newNode := &Node{data: data}  
 // Если дерево пусто, то делаем новый узел корневым  
 if t.root == nil {  
 t.root = newNode  
 return  
 }  
 // Иначе ищем место для вставки нового узла в дереве, начиная с корня  
 current := t.root  
 var parent \*Node  
 for current != nil {  
 // Запоминаем родительский узел  
 parent = current  
 // Если данные нового узла меньше данных текущего узла, то переходим к левому потомку  
 if data < current.data {  
 current = current.left  
 } else {  
 // Иначе переходим к правому потомку  
 current = current.right  
 }  
 }  
 // Связываем новый узел с родительским узлом в соответствии с его данными  
 if data < parent.data {  
 parent.left = newNode  
 } else {  
 parent.right = newNode  
 }  
}  
  
// Создаем метод search для поиска узла в дереве по заданным данным и возвращения его указателя и указателя на его родителя  
func (t \*Tree) search(data int) (\*Node, \*Node) {  
 // Если дерево пусто, то возвращаем nil, nil  
 if t.root == nil {  
 return nil, nil  
 }  
 // Иначе ищем узел в дереве, начиная с корня  
 current := t.root  
 var parent \*Node  
 for current != nil {  
 // Если данные текущего узла равны заданным данным, то возвращаем указатель на текущий узел и его родителя  
 if data == current.data {  
 return current, parent  
 }  
 // Запоминаем родительский узел  
 parent = current  
 // Если данные меньше данных текущего узла, то переходим к левому потомку  
 if data < current.data {  
 current = current.left  
 } else {  
 // Иначе переходим к правому потомку  
 current = current.right  
 }  
 }  
 // Если узел не найден, то возвращаем nil, nil  
 return nil, nil  
}  
  
// Создаем метод remove для удаления узла из дерева по заданным данным  
func (t \*Tree) remove(data int) {  
 // Ищем узел и его родителя в дереве по заданным данным  
 node, parent := t.search(data)  
 // Если узел не найден, то ничего не делаем  
 if node == nil {  
 return  
 }  
 // Определяем, является ли узел левым или правым потомком своего родителя  
 var isLeft bool  
 if parent != nil {  
 isLeft = parent.left == node  
 }  
 // Если узел не имеет потомков, то просто удаляем его из дерева, обнуляя указатель на него у родителя  
 if node.left == nil && node.right == nil {  
 if parent == nil {  
 t.root = nil  
 } else if isLeft {  
 parent.left = nil  
 } else {  
 parent.right = nil  
 }  
 return  
 }  
 // Если узел имеет только левого потомка, то заменяем его на левого потомка, связывая его с родителем  
 if node.left != nil && node.right == nil {  
 if parent == nil {  
 t.root = node.left  
 } else if isLeft {  
 parent.left = node.left  
 } else {  
 parent.right = node.left  
 }  
 return  
 }  
 // Если узел имеет только правого потомка, то заменяем его на правого потомка, связывая его с родителем  
 if node.left == nil && node.right != nil {  
 if parent == nil {  
 t.root = node.right  
 } else if isLeft {  
 parent.left = node.right  
 } else {  
 parent.right = node.right  
 }  
 return  
 }  
 // Если узел имеет обоих потомков, то ищем наименьший узел в правом поддереве и заменяем им удаляемый узел, сохраняя его потомков  
 minNode := node.right  
 var minParent \*Node  
 for minNode.left != nil {  
 minParent = minNode  
 minNode = minNode.left  
 }  
 node.data = minNode.data  
 if minParent == nil {  
 node.right = minNode.right  
 } else {  
 minParent.left = minNode.right  
 }  
}  
  
// Создаем метод print для вывода элементов дерева на экран в виде дерева с отступами и символами  
func (t \*Tree) print() {  
 // Создаем вспомогательную функцию printTree, которая будет рекурсивно выводить узлы дерева с заданным отступом и символом связи  
 var printTree func(node \*Node, indent string, link string)  
 printTree = func(node \*Node, indent string, link string) {  
 if node == nil {  
 return  
 }  
 // Выводим данные узла с отступом и символом связи  
 fmt.Println(indent + link + fmt.Sprint(node.data))  
 // Выводим левого потомка с увеличенным отступом и символом /  
 printTree(node.left, indent+" ", "/ ")  
 // Выводим правого потомка с увеличенным отступом и символом \  
 printTree(node.right, indent+" ", "\\ ")  
 }  
 // Вызываем вспомогательную функцию с корневым узлом дерева, пустым отступом и пустым символом связи  
 printTree(t.root, "", "")  
}  
  
func main() {  
 // Создаем пустое дерево  
 tree := &Tree{}  
 // Добавляем несколько элементов в дерево  
 tree.insert(5)  
 tree.insert(3)  
 tree.insert(7)  
 tree.insert(2)  
 tree.insert(4)  
 tree.insert(6)  
 tree.insert(8)  
 tree.print()  
 // Выводим дерево на экран  
 fmt.Println("Дерево после добавления элементов:")  
 tree.print()  
 // Удаляем элемент из дерева и выводим дерево на экран  
 fmt.Println("Удаляем элемент 5 из дерева:")  
 tree.remove(5)  
 fmt.Println("Дерево после удаления элемента:")  
 tree.print()  
}

### Вывод: