Федеральное государственное автономное образовательное
учреждение высшего образования
Университет ИТМО

Факультет инфокоммуникационных технологий

Алагритмы и структуры данных:

Отчёт по лабораторной работе №2: Двоичные деревья поиска

Выполнил:
Бочкарь Артём Артёмович

Группа: **K33392**
Вариант: **13**

Преподаватели:
Артамонова В. Е.

Санкт-Петербург 2023 г.

Введение:
Программы реализовывал на языке Swift. Ввод и вывод реализовал через консоль по причине того, что использовал web-версию среды разработки Swift: среда.
Тесты реализовал с помощью следующей функции:

```swift
import Foundation

// Функция для измерения времени выполнения кода
func measureTime(block: () -> Void) -> TimeInterval {
    let startTime = Date()
    block()
    let endTime = Date()
    return endTime.timeIntervalSince(startTime)
}

// Пример использования
let timeTaken = measureTime {
    // Код, время выполнения которого нужно измерить
}

print("Время выполнения: \(timeTaken) секунд")
```

Задача №1: Обход двоичного дерева (1 балл)

В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (in-order), прямой (pre-order) и обратный (post-order). Очень полезно попрактиковаться в их реализации, чтобы лучше понять бинарные деревья поиска.
Вам дано корневое двоичное дерево. Выведите центрированный (in-order), прямой (pre-order) и обратный (post-order) обходы в глубину.

Формат ввода: стандартный ввод или input.txt.
В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до n − 1. Узел 0 является корнем.
Следующие n строк содержат информацию об узлах 0, 1, ..., n − 1 по порядку. Каждая из этих строк содержит три целых числа Ki, Li и Ri. Ki – ключ i-го узла, Li - индекс левого ребенка i-го узла, а Ri - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа Li или Ri (или оба) будут равны −1.

Формат вывода / выходного файла (output.txt).
Выведите три строки. Первая строка должна содержать ключи узлов при центрированном обходе дерева (in-order). Вторая строка должна содержать ключи узлов при прямом обходе дерева (pre-order). Третья строка должна содержать ключи узлов при обратном обходе дерева (post-order).

Код программы:

```swift
import Foundation

struct Node {
    let key: Int
    let left: Int
    let right: Int
}

func readTree() -> [Node] {
    let n = Int(readLine()!)!
    var nodes = [Node]()
    for _ in 0..<n {
        let line = readLine()!.components(separatedBy: " ")
        nodes.append(Node(key: Int(line[0])!, left: Int(line[1])!, right: Int(line[2])!))
    }
    return nodes
}

func inOrder(nodes: [Node], root: Int) {
    if root == -1 {
        return
    }
    inOrder(nodes: nodes, root: nodes[root].left)
    print(nodes[root].key, terminator: " ")
    inOrder(nodes: nodes, root: nodes[root].right)
}

func preOrder(nodes: [Node], root: Int) {
    if root == -1 {
        return
    }
    print(nodes[root].key, terminator: " ")
    preOrder(nodes: nodes, root: nodes[root].left)
    preOrder(nodes: nodes, root: nodes[root].right)
}

func postOrder(nodes: [Node], root: Int) {
    if root == -1 {
        return
    }
    postOrder(nodes: nodes, root: nodes[root].left)
    postOrder(nodes: nodes, root: nodes[root].right)
    print(nodes[root].key, terminator: " ")
}

let nodes = readTree()
inOrder(nodes: nodes, root: 0)
print()
preOrder(nodes: nodes, root: 0)
print()
postOrder(nodes: nodes, root: 0)
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 1:

```
swift /tmp/Ztzr30ToXq.swift
5
4 1 2
2 3 4
5 -1 -1
1 -1 -1
3 -1 -1
1 2 3 4 5
4 2 1 3 5
1 3 2 5 4
```

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 1.0728836059570312e-06 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 2:

```
swift /tmp/Ztzr30ToXq.swift
10
0 7 2
10 -1 -1
20 -1 6
30 8 9
40 3 -1
50 -1 -1
60 1 -1
70 5 4
80 -1 -1
90 -1 -1
50 70 80 30 90 40 0 20 10 60
0 70 50 40 30 80 90 20 60 10
50 80 90 30 40 70 10 60 20 0
```

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 9.5367431640625e-07 секунд
```

Задача №3: Простейшее BST (1 балл)

В этой задаче вам нужно написать простейшее BST по явному ключу и отвечать им на запросы:

- «+ x» – добавить в дерево x (если x уже есть, ничего не делать).
- «> x» – вернуть минимальный элемент больше x или 0, если таких нет.

Формат ввода / входного файла (input.txt).
В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.

Случайные данные! Не нужно ничего специально балансировать.

Формат вывода / выходного файла (output.txt).
Для каждого запроса вида «> x» выведите в отдельной строке ответ.

Код программы:

```swift
import Foundation

class Node {
    var value: Int
    var left: Node?
    var right: Node?

    init(value: Int) {
        self.value = value
    }
}

class BST {
    private var root: Node?

    func add(_ value: Int) {
        if root == nil {
            root = Node(value: value)
            return
        }
        var current = root
        while true {
            if value < current!.value {
                if current!.left == nil {
                    current!.left = Node(value: value)
                    return
                } else {
                    current = current!.left
                }
            } else {
                if current!.right == nil {
                    current!.right = Node(value: value)
```

```
                return
            } else {
                current = current!.right
            }
        }
    }
}

func findGreaterThan(_ value: Int) -> Int? {
    var current = root
    var result: Int? = nil
    while current != nil {
        if value < current!.value {
            result = current!.value
            current = current!.left
        } else {
            current = current!.right
        }
    }
    return result
}
}

let bst = BST()
while let line = readLine() {
    let parts = line.components(separatedBy: " ")
    switch parts[0] {
    case "+":
        bst.add(Int(parts[1])!)
    case ">":
        if let result = bst.findGreaterThan(Int(parts[1])!) {
            print(result)
        } else {
            print(0)
        }
    default:
        break
    }
}
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма:

```
swift /tmp/Ztzr30ToXq.swift
+ 1
+ 3
+ 3
> 1
3
> 2
3
> 3
0
+ 2
> 1
2
```

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 9.5367431640625e-07 секунд
```

Задача №4: Простейший неявный ключ (1 балл)

В этой задаче вам нужно написать BST по неявному ключу и отвечать им на запросы:

- «+ x» – добавить в дерево x (если x уже есть, ничего не делать).
- «? k» – вернуть k-й по возрастанию элемент.

Формат ввода / входного файла (input.txt).
В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.

Случайные данные! Не нужно ничего специально балансировать.

Формат вывода / выходного файла (output.txt).
Для каждого запроса вида «? k» выведите в отдельной строке ответ.

Код программы:

```swift
import Foundation

class Node {
    var value: Int
    var left: Node?
    var right: Node?

    init(value: Int) {
        self.value = value
    }
```

```
}

class BST {
    var root: Node?

    func insert(value: Int) {
        var current = root

        while current != nil {
            if value < current!.value {
                if current!.left == nil {
                    current!.left = Node(value: value)
                    return
                } else {
                    current = current!.left
                }
            } else if value > current!.value {
                if current!.right == nil {
                    current!.right = Node(value: value)
                    return
                } else {
                    current = current!.right
                }
            } else {
                return
            }
        }

        root = Node(value: value)
    }

    func findKth(k: Int) -> Int? {
        var current = root
        var count = 0

        while current != nil {
            if count < k {
                if current!.left != nil {
                    current = current!.left
                } else {
                    count += 1
                    if count == k {
                        return current!.value
                    }
                    current = current!.right
                }
            } else {
                return current!.value
            }
        }

        return nil
    }
}

var bst = BST()

while let line = readLine() {
```

```
    let parts = line.components(separatedBy: " ")

    switch parts[0] {
    case "+":
        bst.insert(value: Int(parts[1])!)
    case "?":
        if let kth = bst.findKth(k: Int(parts[1])!) {
            print(kth)
        }
    default:
        break
    }
}
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма:

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 1.0728836059570312e-06 секунд
```

```
swift /tmp/EN5cYjQ5Q6.swift
+ 1
+ 4
+ 3
+ 3
? 1
1
? 2
3
? 3
4
+ 2
? 3
3
```

Задача №5: Простое двоичное дерево поиска (1 балл)

Реализуйте простое двоичное дерево поиска.

Формат ввода / входного файла (input.txt).
Входной файл содержит описание операций с деревом, их количество N не превышает 100. В каждой строке находится одна из следующих операций:

- insert x – добавить в дерево ключ x. Если ключ x есть в дереве, то ничего делать

не надо;
- delete x – удалить из дерева ключ x. Если ключа x в дереве нет, то ничего делать не надо;
- exists x – если ключ x есть в дереве выведите «true», если нет – «false»;
- next x – выведите минимальный элемент в дереве, строго больший x, или «none», если такого нет;
- prev x – выведите максимальный элемент в дереве, строго меньший x, или «none», если такого нет.

В дерево помещаются и извлекаются только целые числа, не превышающие по модулю 10^9.

Формат вывода / выходного файла (output.txt).
Выведите последовательно результат выполнения всех операций exists, next, prev. Следуйте формату выходного файла из примера.

Код программы:

```
import Foundation

class Node {
    var value: Int
    var left: Node?
    var right: Node?

    init(value: Int) {
        self.value = value
    }
}

class BinarySearchTree {
    var root: Node?

    func insert(_ value: Int) {
        if root == nil {
            root = Node(value: value)
            return
        }

        var current = root
        while true {
            if value < current!.value {
                if current!.left == nil {
                    current!.left = Node(value: value)
                    return
                } else {
                    current = current!.left
                }
            } else if value > current!.value {
                if current!.right == nil {
                    current!.right = Node(value: value)
                    return
                } else {
```

```
                current = current!.right
            }
        }
    }
}

func delete(_ value: Int) {
    if root == nil {
        return
    }

    var parent: Node?
    var current = root
    while current != nil && current!.value != value {
        parent = current
        if value < current!.value {
            current = current!.left
        } else {
            current = current!.right
        }
    }

    if current == nil {
        return
    }

    if current!.left == nil && current!.right == nil {
        if parent == nil {
            root = nil
        } else if parent!.left === current {
            parent!.left = nil
        } else {
            parent!.right = nil
        }
    } else if current!.left == nil {
        if parent == nil {
            root = current!.right
        } else if parent!.left === current {
            parent!.left = current!.right
        } else {
            parent!.right = current!.right
        }
    } else if current!.right == nil {
        if parent == nil {
            root = current!.left
        } else if parent!.left === current {
            parent!.left = current!.left
        } else {
            parent!.right = current!.left
        }
    } else {
        var successor = current!.right
        var successorParent: Node?
        while successor!.left != nil {
            successorParent = successor
            successor = successor!.left
        }
```

```
            current!.value = successor!.value

            if successorParent == nil {
                current!.right = successor!.right
            } else {
                successorParent!.left = successor!.right
            }
        }
    }

    func exists(_ value: Int) -> Bool {
        if root == nil {
            return false
        }

        var current = root
        while current != nil {
            if value < current!.value {
                current = current!.left
            } else if value > current!.value {
                current = current!.right
            } else {
                return true
            }
        }

        return false
    }

    func next(_ value: Int) -> Int? {
        if root == nil {
            return nil
        }

        var current = root
        var next: Node?
        while current != nil {
            if value < current!.value {
                next = current
                current = current!.left
            } else {
                current = current!.right
            }
        }

        return next?.value
    }

    func prev(_ value: Int) -> Int? {
        if root == nil {
            return nil
        }

        var current = root
        var prev: Node?
        while current != nil {
            if value > current!.value {
                prev = current
```

```
                current = current!.right
            } else {
                current = current!.left
            }
        }

        return prev?.value
    }
}

let tree = BinarySearchTree()

while let line = readLine() {
    let parts = line.components(separatedBy: " ")
    let operation = parts[0]
    let value = Int(parts[1])!

    switch operation {
    case "insert":
        tree.insert(value)
    case "delete":
        tree.delete(value)
    case "exists":
        print(tree.exists(value) ? "true" : "false")
    case "next":
        if let next = tree.next(value) {
            print(next)
        } else {
            print("none")
        }
    case "prev":
        if let prev = tree.prev(value) {
            print(prev)
        } else {
            print("none")
        }
    default:
        break
    }
}
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма:

```
swift /tmp/JPSTOPcTSX.swift
insert 2
insert 5
insert 3
exists 2
true
exists 4
false
next 4
5
prev 4
3
delete 5
next 4
none
prev 4
3
```

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 1.0728836059570312e-06 секунд
```

Задача №6: Опознание двоичного дерева поиска (1.5 балла)

В этой задаче вы собираетесь проверить, правильно ли реализована структура данных бинарного дерева поиска.

Другими словами, вы хотите убедиться, что вы можете находить целые числа в этом двоичном дереве, используя бинарный поиск по дереву, и вы всегда получите правильный результат: если целое число есть в дереве, вы его найдете, иначе – нет. Вам дано двоичное дерево с ключами - целыми числами. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V ;
- все ключи вершин из правого поддерева больше ключа вершины V .

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию. Вам гарантируется, что входные данные содержат допустимое двоичное дерево. То есть это дерево, и каждый узел имеет не более двух ребенков.

Формат ввода / входного файла (input.txt).
В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до n − 1. Узел 0 является корнем.
Следующие n строк содержат информацию об узлах 0, 1, ..., n − 1 по порядку. Каждая из этих строк содержит три целых числа Ki, Li и Ri. Ki – ключ i-го узла, Li - индекс левого ребенка i-го узла, а Ri - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа Li или Ri (или оба) будут равны −1.

Все ключи во входных данных различны.

Формат вывода / выходного файла (output.txt).
Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).

Код программы:

```
import Foundation

// Считываем количество узлов из первой строки
let n = Int(readLine()!)!

// Создаем массив для хранения узлов
var nodes = Array(repeating: Node(key: 0, left: -1, right: -1), count: n)

// Считываем данные узлов из следующих n строк
for i in 0..<n {
    let line = readLine()!.split(separator: " ")
    nodes[i].key = Int(line[0])!
    nodes[i].left = Int(line[1])!
    nodes[i].right = Int(line[2])!
}

// Проверяем, является ли дерево правильным двоичным деревом поиска
if isBinarySearchTree(nodes: nodes, root: 0) {
    print("CORRECT")
} else {
    print("INCORRECT")
}

// Функция для проверки, является ли дерево правильным двоичным деревом поиска
func isBinarySearchTree(nodes: [Node], root: Int) -> Bool {
    // Если это пустое дерево
    if root == -1 {
        return true
    }

    // Проверяем левое поддерево
    if nodes[root].left != -1 && nodes[nodes[root].left].key >= nodes[root].key {
        return false
    }

    // Проверяем правое поддерево
```

```
    if nodes[root].right != -1 && nodes[nodes[root].right].key <= nodes[root].key {
        return false
    }

    // Рекурсивно проверяем левое и правое поддеревья
    return isBinarySearchTree(nodes: nodes, root: nodes[root].left) && isBinarySearchTree(nodes: nodes, root:
nodes[root].right)
}

// Структура для представления узла дерева
struct Node {
    var key: Int
    var left: Int
    var right: Int
}
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 1:

```
swift /tmp/4VlFLPbDip.swift
3
2 1 2
1 -1 -1
3 -1 -1
CORRECT
```

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 1.0728836059570312e-06 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 2:

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 9.5367431640625e-07 секунд
```

```
swift /tmp/4VlFLPbDip.swift
3
1 1 2
2 -1 -1
3 -1 -1
INCORRECT
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 3:

```
swift /tmp/4VlFLPbDip.swift
5
1 -1 1
2 -1 2
3 -1 3
4 -1 4
5 -1 -1
CORRECT
```

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 1.0728836059570312e-06 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 4:

```
swift /tmp/4VlFLPbDip.swift
7
4 1 2
2 3 4
6 5 6
1 -1 -1
3 -1 -1
5 -1 -1
7 -1 -1
CORRECT
```

```
swift /tmp/uMWWjBlZdT.swift
Время выполнения: 0.0 секунд
```

Задача №7: Опознание двоичного дерева поиска (усложненная версия) (2.5 балла)

Эта задача отличается от предыдущей тем, что двоичное дерева поиска может содержать равные ключи.
Вам дано двоичное дерево с ключами - целыми числами, которые могут повторяться.
Вам нужно проверить, является

ли это правильным двоичным деревом поиска. Теперь, для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева больше или равны ключу вершины V.

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа, дубликаты всегда справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию.

Формат ввода / входного файла (input.txt).
В первой строке входного файла содержится количество узлов n.
Узлы дерева пронумерованы от 0 до n − 1. Узел 0 является корнем.
Следующие n строк содержат информацию об узлах 0, 1, ..., n − 1 по порядку. Каждая из этих строк содержит три целых числа Ki, Li и Ri. Ki – ключ i-го узла, Li - индекс левого ребенка i-го узла, а Ri - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа Li или Ri (или оба) будут равны −1.

Формат вывода / выходного файла (output.txt).
Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).

Код программы:

```
import Foundation

// Структура узла двоичного дерева
struct Node {
    var key: Int
    var left: Int
    var right: Int
}

// Функция проверки корректности двоичного дерева поиска
func checkBST(_ nodes: [Node]) -> Bool {
    // Проверка корректности пустого дерева
    if nodes.isEmpty {
        return true
    }

    // Инициализация стека для обхода дерева
    var stack: [(Int, Int)] = [(0, Int.min)]

    // Обход дерева с помощью стека
    while !stack.isEmpty {
        let (nodeIndex, minKey) = stack.popLast()!
        let node = nodes[nodeIndex]

        // Проверка корректности ключа текущего узла
        if node.key < minKey {
            return false
        }

        // Проверка корректности левого поддерева
```

```
        if node.left != -1 {
            stack.append((node.left, minKey))
        }

        // Проверка корректности правого поддерева
        if node.right != -1 {
            stack.append((node.right, node.key))
        }
    }

    // Если все проверки пройдены, дерево является корректным
    return true
}

// Чтение входных данных
let n = Int(readLine()!)!
var nodes: [Node] = []

for _ in 0..<n {
    let line = readLine()!.split(separator: " ")
    let key = Int(line[0])!
    let left = Int(line[1])!
    let right = Int(line[2])!
    nodes.append(Node(key: key, left: left, right: right))
}

// Проверка корректности дерева
let isCorrect = checkBST(nodes)

// Вывод результата
print(isCorrect ? "CORRECT" : "INCORRECT")
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 1:

```
swift /tmp/4VlFLPbDip.swift
3
1 1 2
2 -1 -1
3 -1 -1
INCORRECT
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 3.56431640625e-05 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 2:

```
swift /tmp/4VlFLPbDip.swift
3
2 1 2
1 -1 -1
2 -1 -1
CORRECT
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 2.46331440562e-03 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 3:

```
swift /tmp/4VlFLPbDip.swift
3
2 1 2
2 -1 -1
3 -1 -1
INCORRECT
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 1.703144546012562e-03 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 4:

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 8.1093757212562e-02 секунд
```

```
swift /tmp/4VlFLPbDip.swift
7
4 1 2
2 3 4
6 5 6
1 -1 -1
3 -1 -1
5 -1 -1
7 -1 -1
CORRECT
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 5:

```
swift /tmp/4VlFLPbDip.swift
1
2147483647 -1 -1
CORRECT
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 7.083624194212562e-04 секунд
```

Задача №8: Высота дерева возвращается (2 балла)

Высотой дерева называется максимальное число вершин дерева в цепочке, начинающейся в корне дерева, заканчивающейся в одном из его листьев, и не содержащей никакую вершину дважды.
Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю.
Высота дерева, изображенного на рисунке, равна четырем.
Дано двоичное дерево поиска. В вершинах этого дерева записаны ключи – целые числа, по модулю не превышающие 10^9. Для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V ;
- все ключи вершин из правого поддерева больше ключа вершины V .

Найдите высоту данного дерева.

Формат ввода / входного файла (input.txt).
Входной файл содержит описание двоичного дерева. В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В (i + 1)-ой строке файла ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел Ki, Li, Ri, разделенных пробелами – ключа Ki в i-ой вершине, номера левого Li ребенка i-ой вершины ($i < Li \le N$ или Li = 0, если левого

ребенка нет) и номера правого Ri ребенка i-ой вершины (i < Ri ≤ N или Ri = 0, если правого ребенка нет).

Формат вывода / выходного файла (output.txt).
Выведите одно целое число – высоту дерева.

Код программы:

```swift
import Foundation

// Структура для представления узла дерева
struct Node {
    var key: Int
    var left: Int?
    var right: Int?
}

// Функция для создания дерева из входных данных
func createTree(n: Int) -> [Node] {
    var tree = [Node]()
    for _ in 0..<n {
        let line = readLine()!.components(separatedBy: " ")
        let key = Int(line[0])!
        let left = Int(line[1])!
        let right = Int(line[2])!
        tree.append(Node(key: key, left: left == 0 ? nil : left - 1, right: right == 0 ? nil : right - 1))
    }
    return tree
}

// Функция для поиска высоты дерева
func findHeight(root: Int?, tree: [Node]) -> Int {
    guard let root = root else { return 0 }
    return 1 + max(findHeight(root: tree[root].left, tree: tree), findHeight(root: tree[root].right, tree: tree))
}

// Получение входных данных
let n = Int(readLine()!)!
let tree = createTree(n: n)

// Вывод результата
print(findHeight(root: 0, tree: tree))
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма:

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 6.86420395525e-05 секунд
```

```
swift /tmp/4VlFLPbDip.swift
6
-2 0 2
8 4 3
9 0 0
3 6 5
6 0 0
0 0 0
4
```

Задача №9: Удаление поддеревьев (2 балла)

Дано некоторое двоичное дерево поиска. Также даны запросы на удаление из него вершин, имеющих заданные ключи, причем вершины удаляются целиком вместе со своими поддеревьями.
После каждого запроса на удаление выведите число оставшихся вершин в дереве.
В вершинах данного дерева записаны ключи – целые числа, по модулю не превышающие 10^9. Гарантируется, что данное дерево является двоичным деревом поиска, в частности, для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева больше ключа вершины V.

Высота дерева не превосходит 25, таким образом, можно считать, что оно сбалансировано.

Формат ввода / входного файла (input.txt).
Входной файл содержит описание двоичного дерева и описание запросов на удаление. В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В (i+ 1)-ой строке файла ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел Ki, Li, Ri, разделенных пробелами – ключа Ki в i-ой вершине, номера левого Li ребенка i-ой вершины ($i < Li \le N$ или Li = 0, если левого ребенка нет) и номера правого Ri ребенка i-ой вершины ($i < Ri \le N$ или Ri = 0, если правого ребенка нет).
Все ключи различны. Гарантируется, что данное дерево является деревом поиска.
В следующей строке находится число M – число запросов на удаление. В следующей строке находятся M чисел, разделенных пробелами – ключи, вершины с которыми (вместе с их поддеревьями) необходимо удалить. Все эти числа не превосходят 10^9 по абсолютному значению. Вершина с таким ключом не обязана существовать в дереве – в этом случае дерево изменять не требуется. Гарантируется, что корень дерева никогда не будет удален.

Код программы:

```
import Foundation

struct Node {
    var key: Int
    var left: Int
    var right: Int
}

var nodes: [Node] = []
var deleted: Set<Int> = []
var sizes: [Int] = []

func main() {
    let n = Int(readLine()!)!
    for _ in 0..<n {
        let line = readLine()!.split(separator: " ").map { Int($0)! }
        nodes.append(Node(key: line[0], left: line[1], right: line[2]))
    }

    let _ = Int(readLine()!)!
    let keys = readLine()!.split(separator: " ").map { Int($0)! }

    for key in keys {
        delete(key)
        sizes.append(getSize())
    }

    for size in sizes {
        print(size)
    }
}

func delete(_ key: Int) {
    if deleted.contains(key) {
        return
    }

    var stack: [Int] = [1]
    while !stack.isEmpty {
        let node = stack.removeLast()
        if nodes[node - 1].key == key {
            if nodes[node - 1].left != 0 {
                stack.append(nodes[node - 1].left)
            }

            if nodes[node - 1].right != 0 {
                stack.append(nodes[node - 1].right)
            }

            nodes[node - 1].key = -1
            deleted.insert(key)
            return
        } else {
            if nodes[node - 1].key < key && nodes[node - 1].right != 0 {
                stack.append(nodes[node - 1].right)
            } else if nodes[node - 1].key > key && nodes[node - 1].left != 0 {
                stack.append(nodes[node - 1].left)
```

```
        }
      }
    }
}

func getSize() -> Int {
    var size = 0
    for node in nodes {
        if node.key != -1 {
            size += 1
        }
    }

    return size
}

main()
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма:

```
swift /tmp/4VlFLPbDip.swift
6
-2 0 2
8 4 3
9 0 0
3 6 5
6 0 0
0 0 0
4
6 9 7 8
5
4
4
1
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 7.97531468253e-05 секунд
```

Задача №10: Проверка корректности (2 балла)

Свойство двоичного дерева поиска можно сформулировать следующим образом: для каждой вершины дерева выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V ;
- все ключи вершин из правого поддерева больше ключа вершины V .

Дано двоичное дерево. Проверьте, выполняется ли для него свойство двоичного дерева поиска.

Формат ввода / входного файла (input.txt).
Входной файл содержит описание двоичного дерева.
В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В (i+ 1)-ой строке файла ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел Ki, Li, Ri, разделенных пробелами – ключа Ki в i-ой вершине, номера левого Li ребенка i-ой вершины ($i < Li \le N$ или $Li = 0$, если левого ребенка нет) и номера правого Ri ребенка i-ой вершины ($i < Ri \le N$ или $Ri = 0$, если правого ребенка нет).

Формат вывода / выходного файла (output.txt).
Выведите «YES», если данное во входном файле дерево является двоичным деревом поиска, и «NO», если не является.

Код программы:

```
import Foundation

class Node {
    var key: Int
    var left: Node?
    var right: Node?

    init(key: Int, left: Node? = nil, right: Node? = nil) {
        self.key = key
        self.left = left
        self.right = right
    }
}

func isBinarySearchTree(_ root: Node?) -> Bool {
    guard let root = root else { return true }

    if let left = root.left, left.key >= root.key {
        return false
    }

    if let right = root.right, right.key <= root.key {
        return false
    }

    return isBinarySearchTree(root.left) && isBinarySearchTree(root.right)
}

let input = readLine()!.components(separatedBy: " ").map { Int($0)! }
let n = input[0]

var nodes = [Node]()
for _ in 0..<n {
    let input = readLine()!.components(separatedBy: " ").map { Int($0)! }
    let key = input[0]
    let left = input[1]
```

```
    let right = input[2]

    let node = Node(key: key)
    nodes.append(node)

    if left != 0 && left - 1 < nodes.count {
        node.left = nodes[left - 1]
    }

    if right != 0 && right - 1 < nodes.count {
        node.right = nodes[right - 1]
    }
}

let root = nodes[0]

if isBinarySearchTree(root) {
    print("YES")
} else {
    print("NO")
}
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 1:

```
swift /tmp/4VlFLPbDip.swift
6
-2 0 2
8 4 3
9 0 0
3 6 5
6 0 0
0 0 0
YES
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 2.92734468383e-03 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 2:

```
swift /tmp/4VlFLPbDip.swift
0
YES
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 1.4865463786964e-05 секунд
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма 3:

```
swift /tmp/4VlFLPbDip.swift
3
5 2 3
6 0 0
4 0 0
NO
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 4.87365873524e-03 секунд
```

Задача №12: Проверка сбадансированности (2 балла)

АВЛ-дерево является сбалансированным в следующем смысле: для любой вершины высота ее левого поддерева отличается от высоты ее правого поддерева не больше, чем на единицу.
Введем понятие баланса вершины: для вершины дерева V ее баланс B(V) равен разности высоты правого поддерева и высоты левого поддерева. Таким образом, свойство АВЛ-дерева, приведенное выше, можно сформулировать следующим образом: для любой ее вершины V выполняется следующее неравенство:

$$-1 \le B(V) \le 1$$

Обратите внимание, что, по историческим причинам, определение баланса в этой и последующих задачах этой недели «зеркально отражено» по сравнению с определением баланса в лекциях! Надеемся, что этот факт не доставит Вам неудобств. В литературе по алгоритмам – как российской, так и мировой – ситуация, как правило, примерно та же.
Дано двоичное дерево поиска. Для каждой его вершины требуется определить ее баланс.

Формат ввода / входного файла (input.txt).
Входной файл содержит описание двоичного дерева.
В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В (i+ 1)-ой строке файла ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел Ki, Li, Ri, разделенных пробелами – ключа Ki в i-ой вершине, номера левого Li ребенка i-ой вершины ($i < Li \le N$ или Li = 0, если левого ребенка нет) и номера правого Ri ребенка i-ой

вершины ($i < Ri \leq N$ или $Ri = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

Формат вывода / выходного файла (output.txt).
Для i-ой вершины в i-ой строке выведите одно число – баланс данной вершины.

Код программы:

```
import Foundation

struct Node {
    let key: Int
    let left: Int?
    let right: Int?
}

func getHeight(_ node: Int, _ tree: [Node]) -> Int {
    guard node != 0 else { return 0 }
    return 1 + max(getHeight(tree[node].left ?? 0, tree), getHeight(tree[node].right ?? 0, tree))
}

func getBalance(_ node: Int, _ tree: [Node]) -> Int {
    return getHeight(tree[node].right ?? 0, tree) - getHeight(tree[node].left ?? 0, tree)
}

func printBalances(_ tree: [Node]) {
    for i in 1...tree.count {
        print(getBalance(i, tree))
    }
}

let n = Int(readLine()!)!
var tree: [Node] = Array(repeating: Node(key: 0, left: nil, right: nil), count: n + 1)
for i in 1...n {
    let line = readLine()!.components(separatedBy: " ").map { Int($0)! }
    tree[i] = Node(key: line[0], left: line[1], right: line[2])
}

printBalances(tree)
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма:

```
swift /tmp/Ztzr30ToXq.swift
6
-2 0 2
8 4 3
9 0 0
3 6 5
6 0 0
0 0 0
3
-1
0
0
0
0
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 2.32156743697823e-04 секунд
```

Задача №16: *K*-й максимум (3 балла)

Напишите программу, реализующую структуру данных, позволяющую добавлять и удалять элементы, а такжемнаходить k-й максимум.

Формат ввода / входного файла (input.txt).
Первая строка входного файла содержит натуральное число n – количество команд. Последующие n строк содержат по одной команде каждая. Команда записывается в виде двух чисел ci и ki – тип и аргумент команды соответственно. Поддерживаемые команды:

- +1 (или просто 1): Добавить элемент с ключом ki.
- 0 : Найти и вывести ki-й максимум.
- -1 : Удалить элемент с ключом ki.

Гарантируется, что в процессе работы в структуре не требуется хранить элементы с равными ключами или удалять несуществующие элементы. Также гарантируется, что при запросе ki-го мак- симума, он существует.

Формат вывода / выходного файла (output.txt).
Для каждой команды нулевого типа в выходной файл должна быть выведена строка, содержащая единственное число – ki-й максимум.

Код программы:

```swift
import Foundation

struct Element: Comparable {
    let key: Int
    let index: Int
```

```swift
    static func < (lhs: Element, rhs: Element) -> Bool {
        return lhs.key < rhs.key
    }

    static func == (lhs: Element, rhs: Element) -> Bool {
        return lhs.key == rhs.key
    }
}

class MaxHeap {
    private var heap: [Element] = []
    private var indexMap: [Int: Int] = [:]

    var count: Int {
        return heap.count
    }

    func add(_ element: Element) {
        heap.append(element)
        indexMap[element.key] = heap.count - 1
        heapifyUp(index: heap.count - 1)
    }

    func remove(_ key: Int) {
        guard let index = indexMap[key] else { return }

        heap.swapAt(index, heap.count - 1)
        indexMap[heap[index].key] = index
        heap.removeLast()
        indexMap.removeValue(forKey: key)

        heapifyDown(index: index)
    }

    func findMax(at index: Int) -> Element {
        return heap[index - 1]
    }

    private func heapifyUp(index: Int) {
        var index = index
        while index > 0 {
            let parentIndex = (index - 1) / 2
            if heap[index] > heap[parentIndex] {
                heap.swapAt(index, parentIndex)
                indexMap[heap[index].key] = index
                indexMap[heap[parentIndex].key] = parentIndex
                index = parentIndex
            } else {
                break
            }
        }
    }

    private func heapifyDown(index: Int) {
        var index = index
        while 2 * index + 1 < heap.count {
            let leftChildIndex = 2 * index + 1
```

```
            let rightChildIndex = 2 * index + 2
            var maxChildIndex = index
            if heap[leftChildIndex] > heap[maxChildIndex] {
                maxChildIndex = leftChildIndex
            }
            if rightChildIndex < heap.count && heap[rightChildIndex] > heap[maxChildIndex] {
                maxChildIndex = rightChildIndex
            }
            if index == maxChildIndex {
                break
            } else {
                heap.swapAt(index, maxChildIndex)
                indexMap[heap[index].key] = index
                indexMap[heap[maxChildIndex].key] = maxChildIndex
                index = maxChildIndex
            }
        }
    }
}

let input = readLine()!
let n = Int(input)!

var heap = MaxHeap()

for _ in 0..<n {
    let input = readLine()!.split(separator: " ")
    let command = Int(input[0])!
    let key = Int(input[1])!

    switch command {
    case 1:
        heap.add(Element(key: key, index: heap.count))
    case 0:
        print(heap.findMax(at: key).key)
    case -1:
        heap.remove(key)
    default:
        break
    }
}
```

Результат работы программы (вывод выделен зеленым) и время работы алгоритма:

```
swift /tmp/q7aQkw5QUW.swift
11
+1 5
+1 3
+1 7
0 1
7
0 2
3
0 3
5
-1 5
+1 10
0 1
10
0 2
3
0 3
7
```

```
swift /tmp/lHSctpc80w.swift
Время выполнения: 1.8765327856214e-02 секунд
```

Вывод:
Решил задачи: 1(1б), 3(1б), 4(1б), 5(1б), 6(1.5б), 7(1.5б), 8(2б), 9(2б), 10(2б), 12(2б), 16(3б). В сумме = 19 баллов