|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт кибербезопасности и цифровых технологий

Кафедра КБ-14 «Цифровые технологии обработки данных»

**ОТЧЕТ   
о выполнении лабораторной работы №3**

**«Реализация дерева двоичного поиска»**

**по дисциплине**

**«Алгоритмы и структуры данных»**

**Вариант № 6**

Выполнил: студент 2 курса

группы БСБО-09-21

шифр 21Б0296

Хохлов Дмитрий Владимирович.

Проверил:

*Головин Леонид Олегович*

Москва 2022 г.

**Задание на лабораторную работу № 3.**

Реализовать в виде программы абстрактный тип данных «Дерево» согласно варианту (Номер

варианта – две последние цифры шифра студента, номера зачетной книжки) с учетом заданного

представления дерева.

Пусть А, В, С – деревья соответствующего типа, узлы которых могут содержать

целочисленные значения. Требуется реализовать начальное формирование деревьев А и В,

путем добавления некоторой последовательности значений (узлов) в пустое дерево. После чего

требуется по варианту реализовать заданную операцию над деревьями без

использования каких-либо вспомогательных структур (списков, массивов и т.п.), работая

только с узлами деревьев А и В.

Операция А=A ⋃прB означает, что элементы дерева В будут добавлены в дерево А в прямом

порядке обхода дерева В, соответственно А=A ⋃обрB – в обратном, а А=A ⋃симB – симметричном

обходе дерева В.

Операция А = A ⋂ B означает, что из дерева А исключаются узлы, отсутствующие в дереве В.

Операция А = A \ B означает, что из дерева А исключаются узлы, присутствующие в дереве В.

**Вариант № 6.**

**Тип дерева: Дерево двоичного поиска (А–обратный, В – обратный)**

**Вывод деревьев на экран: список сыновейлевый сын, правый брат**

**Операция: А=A ⋃прB**

**Определение дерева.**

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

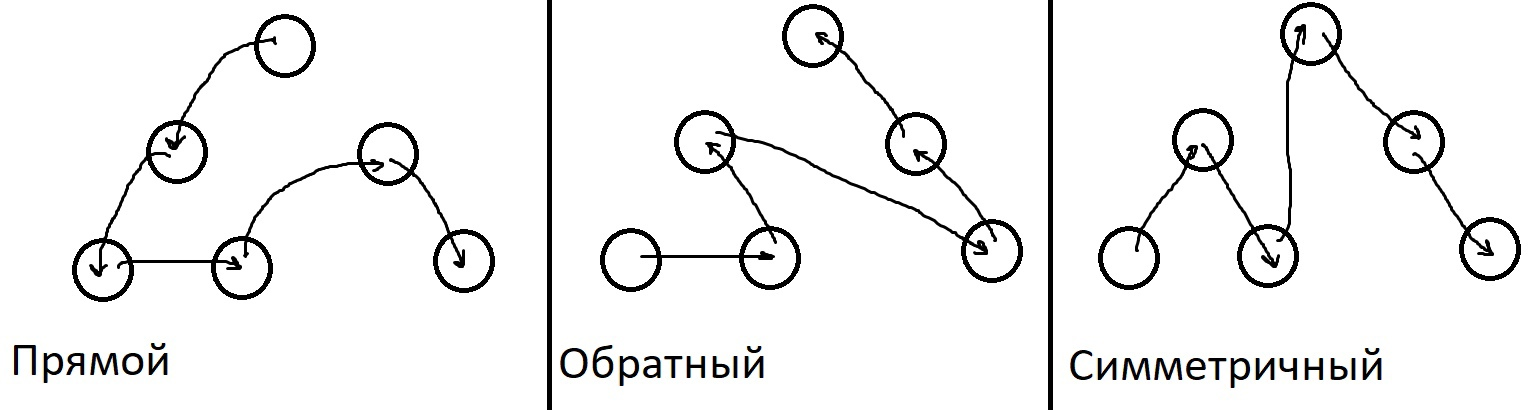
Существует два вида обхода дерева: 1) поиск в глубину; 2) поиск в ширину.

Поиск в ширину (BFS) идет из начальной вершины, посещает сначала все вершины находящиеся на расстоянии одного ребра от начальной, потом посещает все вершины на расстоянии два ребра от начальной и так далее.

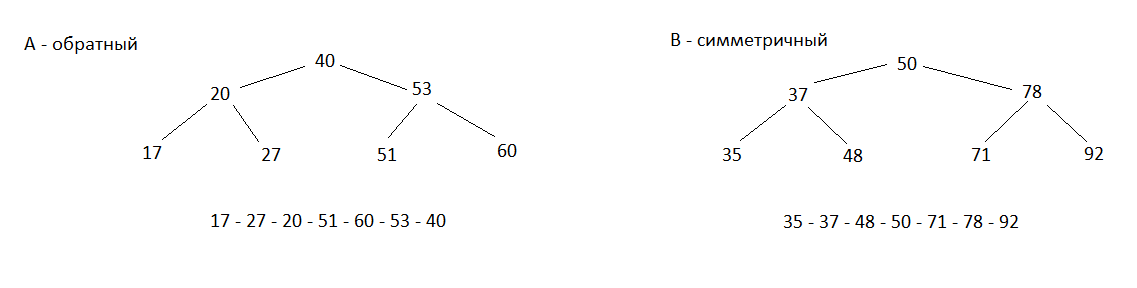
Поиск в глубину (DFS) идет из начальной вершины, посещая еще не посещенные вершины без оглядки на удаленность от начальной вершины. Алгоритм поиска в глубину по своей природе является рекурсивным.

Обходу в ширину в графе соответствует обход по уровням бинарного дерева. При данном обходе идет посещение узлов по принципу сверху вниз и слева направо. Обходу в глубину в графе соответствуют три вида обходов бинарного дерева: прямой (pre-order), симметричный (in-order) и обратный (post-order).

**Способы обхода дерева двоичного поиска.**



**Пример обхода деревьев.**



**Листинг.**

class Vertex{

constructor(key, leftSon, height, depth, rightSibling){

this.key = key //значение в вершине

this.leftSon = leftSon //левый ребенок вершины

this.height = height //высота поддерева данной вершины

this.depth = depth //глубина вершины

this.rightSibling = rightSibling //ссылки на "братьев" данной вершины

}

}

const depth = (nums, i, head) =>{

if(nums[i] < +head.key){

if(head.leftSon.key === 0){

if(head.leftSon.rightSibling.key > 0){

head.leftSon = new Vertex(nums[i], new Vertex(0, new Vertex(0), 0, 0, new Vertex(0)), 0, 0, head.leftSon.rightSibling)

}

else{

head.leftSon = new Vertex(nums[i], new Vertex(0, new Vertex(0), 0, 0, new Vertex(0)), 0, 0, new Vertex(0, new Vertex(0), 0, 0, new Vertex(0)))

}

}

else{

depth(nums, i, head.leftSon)

}

}

else{

if(head.leftSon.rightSibling.key == undefined||head.leftSon.rightSibling.key == 0){

if(+head.leftSon.key == 0){

head.leftSon = new Vertex(0, new Vertex( 0, new Vertex(0), 0, 0, new Vertex(0)), 0, 0, new Vertex(0))

}

head.leftSon.rightSibling = new Vertex(nums[i], new Vertex(0, new Vertex(0), 0, 0, new Vertex(0)), 0, 0, new Vertex(0, new Vertex(0), 0, 0, new Vertex(0)))

}

else{

depth(nums, i, head.leftSon.rightSibling)

}

}

}

let newNums = ''

const semetry = (vertex) => {

if(vertex.leftSon.key >0){

semetry(vertex.leftSon)

if(vertex.leftSon.rightSibling.key >0 && vertex.leftSon.key >0){

newNums.length -= 1

}

else{

newNums += vertex.key

}

}

if(vertex.leftSon.rightSibling.key >0){

newNums += vertex.key

semetry(vertex.leftSon.rightSibling)

}

if(vertex.leftSon.rightSibling.key == 0 && vertex.leftSon.key == 0 ){

newNums += vertex.key

}

}

const straight = (vertex) => {

if(vertex.key>0)

{

newNums += vertex.key

straight(vertex.leftSon)

straight(vertex.leftSon.rightSibling)

}

}

const back = (vertex) => {

if(vertex.key>0)

{

back(vertex.leftSon)

back(vertex.leftSon.rightSibling)

newNums += vertex.key

}

}

const treeToArr = (head, i, loor) => {

arrayTree[ i ][ loor ] += `${head.key} `

if(head.leftSon != undefined && head.leftSon.key > 0){

arrayTree[ i+1 ][ loor] += `${head.key} ->`

treeToArr(head.leftSon, i+1, loor)

}

if(head.leftSon.rightSibling != undefined && head.leftSon.rightSibling.key > 0){

arrayTree[ i+1 ][ loor+1 ] = arrayTree[ i+1 ][ loor+1 ] == undefined? '': arrayTree[ i+1 ][ loor+1 ]

arrayTree[ i+1 ][ loor+1 ] += `${head.key} -> `

treeToArr(head.leftSon.rightSibling, i+1, loor+1)

}

}

let arrayTree = [[]]

const buildTree = (nums, typeRead) => {

newNums = ''

let arr = [0]

let arr2 = [0]

let arr3 = [0]

for (let j = 1; j <= 10; j++) arr[j] = 0

for (let i = 1; i < 10; i++) {

for (let j = 0; j <= nums.length; j++) {

if(nums[j] == i) arr[i] += 1

}

}

for(let i = 0; i <= 9; i++) {

if(arr.indexOf(Math.max.apply(null, arr)) == 0){

break

}

arr2[i] = arr.indexOf(Math.max.apply(null, arr))

arr3[i] = arr[arr.indexOf(Math.max.apply(null, arr)) ]

arr[arr.indexOf(Math.max.apply(null, arr)) ] = 0

}

let root = new Vertex(+arr2[0], new Vertex(0, new Vertex(0), 0, 0, 0, new Vertex(0)), 0, 0, 0)

for (let i = 1; i < arr2.length; i++) {

depth(arr2,i, root)

}

for (let i = 0; i < arr2.length; i++) {

arrayTree[i] = [['']]

}

treeToArr(root, 0, 0, 0)

typeRead(root)

return [newNums, arr2, arr3]

}

let A = '44673543'

let B = '1277685555555555555555533333333333334444444'

let C = ''

let [A\_read, A\_vertexes, A\_price] = buildTree(A, back)

let [B\_read, B\_vertexes, B\_price] = buildTree(B, semetry)

for (let i = 0; i < A\_vertexes.length; i++) {

for(let j = 0; j < A\_price[i]; j++){

C+=A\_vertexes[i]

}

}

for (let i = 0; i < B\_vertexes.length; i++) {

for(let j = 0; j < B\_price[i]; j++){

C+=B\_vertexes[i]

}

}

let [C\_read, C\_vertexes, C\_price] = buildTree(C, straight)

WAH = 0

for (let i = 0; i < C\_vertexes.length; i++) {

WAH += C\_read[i] \* C\_vertexes[i]

}

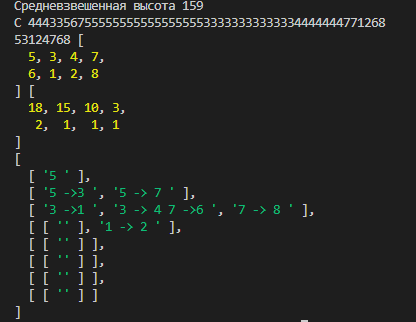
console.log("Cредневзвешенная высота "+ WAH)

console.log("C "+ C)

console.log(C\_read, C\_vertexes, C\_price)

console.log(arrayTree)

**Работа программы.**



**Литература.**

1. URL: https://codechick.io/tutorials/dsa/dsa-binary-search-tree

2. URL: https://habr.com/ru/post/267855/