Государственное образовательное учреждение высшего профессионального образования
Санкт-Петербургский национальный исследовательский университет
Информационных Технологий, Механики и Оптики
Факультет инфокоммуникационных технологий

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

По алгоритмам и структурам данных
На тему: «Деревья и обходы»

Проверил(а):

_____

Дата: «___»_____201__г.

Оценка: _____

Работу выполнил(а):

Никончук А.П.
Студент(ка) группы К3242
Дневного отделения

_____

2018 г.

Цель:
Получить начальные знания о деревьях, как структуры данных.

Задание:

1. Освоить синтаксис нового языка программирования;
2. написать алгоритм создания связного упорядоченного дерева
3. написать алгоритмы обхода дерева

Задание согласно варианту:
Напишите программу, которая создает связное упорядоченное дерево и совершает по нему симметричный и обратный симметричный обходы. Побочно решается задача создания дерева и отрисовка.

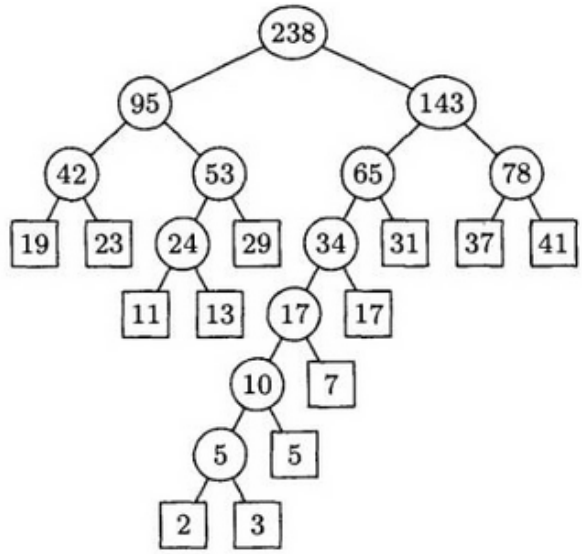
Реализация:

1. Описание принципа работы алгоритма

Дерево – связный граф без циклов, характеризующийся наличием корня, из которого по цепочке указателей можено перейти к любой другой вершине до листьев. Невзвешанный и неориетированный (от корня к листьям).
Связное обозначает, что все элементы переданные дереву сообщаются через какое-то количество ребер, т. е. между любыми двумя узлами всегда будет существовать путь.
Бинарные (двоичные) деревья являются одним из самых востребованных вариантов данной структуры данных, так как широко используются в поисковых алгоритмах и для решения других вычислительных задач, поэтому будем создавить бинарные упорядоченные деревья. При помещении нового элемента сравнение с корневым, а затем родительскими узлами будет происходит по заложенному в алгоритм правилу – в этом заключается упорядоченность.
Бинарное дерево поиска – это структура, которая упорядочивает элементы посредством отношения “<”.

1.1.a. Создание бинарного дерева из неупорядоченного массива
Создание дерева из массива-комбинации неповторяющихся чисел – функция creteBinaryTree. В качестве дерева (tree) в неё передается пустой массив элементов ([]), если нет задачи довбавить элементов в существующее дерево.
Присваиваем значение ссылке на ветку начало дерева – предварительно добавленный в качестве корня первый элемент массива. Затем циклически выбираем из 2х “кейсов”. Происзводим сравнение корневого элемента выбранной ветки. Если корневой элемент больше элемента массива, который нужно вставить, то он пойдет в левое поддерево, иначе в правое. Внутри проверка существования этой подветки. Если существует, то подменяем выбранную текущую ветку, после происходит новая интерация цикла. Иначе элементо вставляется в дерево. (Ветка левая или правая обозначается как пустой массив, в её коррень помещается элемент массива, которые следующим шагом удаляется из своего массива. Новая текущая ветка – ссылка на все дерево.)

1.1.b. Создание бинарного дерева из упорядоченного массива методом Хаффмана
Другой вариант – рекурсивное решение алгоритма Хаффмана. Алгоритм заключается в выборе двух наименьших двух элементов, объединении их в поддерево и помешение в качестве ветки этого поддерева в следующую выбранную пару. Таким образом элементы массива собираются в дерево от листьев к корню. Все листья (терминальные элементы) будут представлять собой элементы переданного массива. createHaffleBinaryTree требует на вход два массива: первый, пустой, в теле функции клонирует второй – массив элементов для добавления в дерево, затем будет преобразован в дерево. При этом важно в javascript, чтобы переменные не оставались в одном поле видимости, и выходит, что под разными именами мы обращаемся к одной и той же области памяти.




```
2 3 5 7 11 13 17 19 23 29 31 37 41
  5 5 7 11 13 17 19 23 29 31 37 41
   10 7 11 13 17 19 23 29 31 37 41
     17   24 17 19 23 29 31 37 41
          24 34 19 23 29 31 37 41
          24 34   42 29 31 37 41
                  42 53 65 37 41
                  42 53 65   78
                     95 65   78
                     95      143
                             238
```

1.2. Обходы

Пошаговый перебор элементов дерева (перебор всех элементов дерева;) по связям между узлами-предками и узлами-потомками называется обходом дерева.

Пусть примером будет выражение 1+ 2 * 3.

Прямой (preorder) – корень выводится раньше всех прочих элементов дерева, за ним левое поддерево, в заключении правое.

Посетить корень → Обойти левое поддерево → Обойти правое поддерево

Симметричный (поперечный, in-order) – вначале выводится левое поддерево, затем его родителя, затем правое поддерево. Инфикстная запись выражения: 1 + 2 * 3

Обойти левое поддерево → Посетить корень → Обойти правое поддерево

Обратный симметричный (в обратном порядке или поступорядоченный, postorder) – вывод поддерева раньше его родителя. Постфиксный вывод 1 2 3 * +. Вначале выводятся листья, в конце – корень.

Обойти левое поддерево → Обойти правое поддерево → Посетить корень

1.3. Отрисовка / отображение дерева

Определение количества узлов дерева

Определение ширины дерева

Определение глубины дерева

(вычисление масштаба дерева, горизонтальное или вертикальное его отображение – сколько места понадобится)

2. Код программы на 2 языках программирования

2.1. JavaScript

```
function createBinaryTree(tree, array)
{
        console.log("\n");
        if (tree.length<1){
                var tree = [array.shift()];
        }
        var currentNode = tree;

        while(array.length>0){
                //console.log("_____");
                //console.log("THIS: "+tree.toString());
                console.log(currentNode);
                if (currentNode[0]>array[0]){
```

```
                        // left branch
                        if (currentNode[1]){
                        // left branch exists
                                console.log(currentNode[0] + ">" + array[0] + "  leftChild exists ");
                                console.log(currentNode);
                                currentNode = currentNode[1];
                                //break;
                        }else{
                        // left branch doesn't exist
                                console.log(currentNode[0] + ">" + array[0] + " leftChild doesn exist ");
                                currentNode[1] = [];
                                currentNode[1].push(array[0]);
                                array.shift();
                                currentNode = tree;
                        }
                }else{
                // right branch
                        if (currentNode[2]){
                        // both branches exists
                                console.log(currentNode[0] + "<" + array[0] + " rightChild exists ");
                                currentNode = currentNode[2]
                        } else{
                        // right branch doesn't exist
                                console.log(currentNode[0] + ">" + array[0] + " rightChild doesn exist");
                                currentNode[2] = []
                                currentNode[2].push(array[0]);
                                array.shift();
                                currentNode = tree;
                        }
                }
        }
        console.log("\n");
        return tree;
}

function createHaffleBinaryTree(tree, dublicate)
{
        if (tree.length==0){
                for (i=0; i<dublicate.length; i++){
                        tree.push([dublicate[i]]);
                }
        }
        if(dublicate.length>1){
                //console.log("_____");
                //console.log(tree);
                //console.log(dublicate+" "+dublicate.length);
                var x1 = getMinValue(dublicate);
                var index = dublicate.indexOf(x1);
                var y1 = tree[index];
                //console.log("x1="+x1+" y1="+y1);
                //console.log("index"+index);
                dublicate.splice(index, 1);
                tree.splice(index, 1);
                //console.log(dublicate+" "+dublicate.length);
                var x2 = getMinValue(dublicate);
                var index = dublicate.indexOf(x2);
                var y2 = tree[index];
```

```
                //console.log("x2="+x2+" y2="+y2);
                //console.log("index"+index);
                dublicate.splice(index, 1);
                tree.splice(index, 1);
                //console.log(dublicate+" "+dublicate.length);
                dublicate.push(x1+x2);
                tree.push([x1+x2, y1, y2]);
                //console.log(dublicate+" "+dublicate.length);
                createHaffleBinaryTree(tree, dublicate);
        }else{
                return tree;
        }
        return tree;
}

function preorder(tree)
{
        if (tree!=undefined){
                console.log(getRoot(tree));
                preorder(getLeftChild(tree));
                preorder(getRightChild(tree));
        }
}

function inorder(tree)
{
        if (tree!=undefined){
                inorder(getLeftChild(tree));
                console.log(getRoot(tree));
                inorder(getRightChild(tree));
        }
}

function postorder(tree)
{
        if (tree!=undefined){
                postorder(getLeftChild(tree));
                postorder(getRightChild(tree));
                console.log(getRoot(tree));
        }
}
```

2.2. Java

```
package com.company;

public class Node <A1, A2, A3> {
     int index;
     int value;
     Node left_child;
     Node right_child;

     public Node TakeLeftChild(){
          return this.left_child;
     }
     public Node TakeRightChild(){
          return this.left_child;
     }
     Node(int index, int value, Node parent){
```

```
        this.index = index;
        this.value = value;
    }
}

package com.company;

class Tree { // package-private
    Node root;
    Tree(Node node){
        this.root = node;
    }
    public Node get(Tree tree, int value){
        if (tree.root.value == value){
            return tree.root;
        }
        if (tree.root.value > value){
            return get(new Tree(tree.root.right_child), value);
        } else {
            return get(new Tree(tree.root.left_child), value);
        }
    }
    private Tree addNode(Tree tree, int value) {
        Node node;
        // root is null then new value put at the root
        if (tree.root == null) {
            node = new Node(0, value, null);
            tree.root = node;
        } else {
            // if new value is smaller then root
            if (value < tree.root.value){
                // if left child is empty it puts in left child
                if (tree.root.left_child == null){
                    node = new Node(tree.root.index+1, value, tree.root);
                    tree.root.left_child = node;
                // in the other ocasion we use recursion to compare left branch
                } else {
                    Tree left_branch = new Tree(tree.root.left_child);
                    addNode(left_branch, value);
                }
            // if new value is bigger then root
            } else {
                // if left child is empty it puts in left child
                if (tree.root.right_child == null){
                    node = new Node(tree.root.index+2, value, tree.root);
                    tree.root.right_child = node;
                // in the other ocasion we use recursion to compare right branch
                } else {
                    Tree right_branch = new Tree(tree.root.right_child);
                    addNode(right_branch, value);
                }
            }
        }
        return tree;
    }
    Tree createTree(int[] mass){
        Tree tree = new Tree(null);
        for (int i=0; i < mass.length; i++){
            tree.addNode(tree, mass[i]);
        }
        return tree;
    }

    void preorder(Tree tree){
```

```
        if (tree.root!=null) {
            System.out.println(tree.root.value);
            System.out.print('|');
            Tree left_branch = new Tree(tree.root.left_child);
            Tree right_branch = new Tree(tree.root.right_child);
            System.out.print('-');
            preorder(left_branch);
            System.out.print('-');
            preorder(right_branch);
        }
    }
    void inorder(Tree tree){
        if (tree.root!=null) {
            Tree left_branch = new Tree(tree.root.left_child);
            Tree right_branch = new Tree(tree.root.right_child);
            inorder(left_branch);
            System.out.println(tree.root.value);
            inorder(right_branch);
            System.out.print(' ');
        }
    }
    void postorder(Tree tree){
        if (tree.root!=null) {
            Tree left_branch = new Tree(tree.root.left_child);
            Tree right_branch = new Tree(tree.root.right_child);
            System.out.print('-');
            postorder(left_branch);
            postorder(right_branch);
            System.out.println(tree.root.value);
        }
    }
}
```

3. Пример входных данных

На каждом языке написана функция генерации массивов случайных чисел. В JS происходит перемешивание созданного масива в заданном промежутке. В Java генерируется число в промежутке, если в статическом массиве его нет, добавляется, иначе откатываетя шаг цикла.

3.1. JavaScript

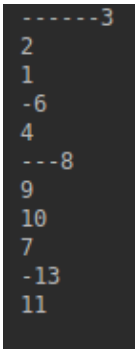
```
function getAllNumbers(max_num)
// create ordered mass from 0 to max_number
{
        var array = [];
        for (var i=0; i <= max_num; i++){
                array.push(i);
        }
        return array;
}

function shuffle(array, count_elem)
// shuffle part of array
{
        var pool = [];
        for (var i = 0; i<=count_elem; i++){
                var randomNumber = array.splice(Math.floor(Math.random() * ((count_elem-i)-1) +1), 1);
                pool.push(randomNumber.pop());
                //console.log(array)
        }
        return pool;
}
```

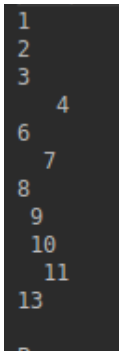
3.2. Java

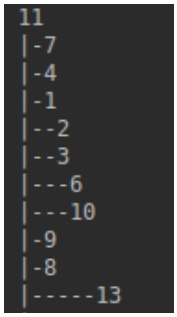
```
    private static int getRandomBetween(int max){
        return (int)(Math.random()*((max-1)+1)+1);
        //Random random = new Random();
        //return random.nextInt(max);
    }
    private static boolean checkNumberInMass(int[] mass, int elem){
        for (int i: mass) {
            if (i == elem){
                return false;
            }
        }
        return true;
    }
    private static int[] createRandomMass(int count_elem, int max_elem) {
        //if (max_elem > count_elem && count_elem>1){
        int[] mass = new int[count_elem];
        for ( int i=0; i < count_elem; i++){
        mass[i] = 0;
                int number = getRandomBetween(max_elem);
                if (checkNumberInMass(mass, number)){
                    mass[i] = number;
                }
                else i--;
        }
        return mass;
    }
    private static void printMass(int[] mass){
        System.out.print("[");
        for (int i=0; i < mass.length; i++) {
            System.out.print(" " + mass[i]);
        }
        System.out.print(']');
    }
}
```

4. Примеры результатов работы, скриншоты

postorder, inorder, preorder на массиве `{ 11, 7, 4, 10, 1, 9, 13, 2, 8, 6, 3 };`







Click the button to create a new ordered tree.

5 13 Create tree

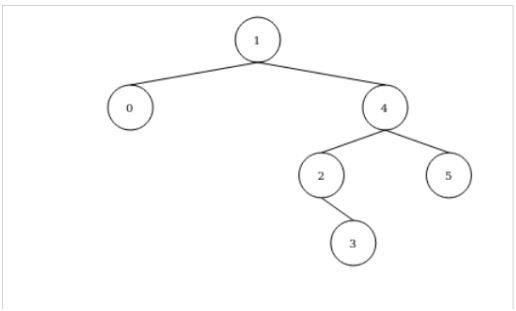
numbers: 1,4,2,3,5,0

pre-order: 1,0,4,2,3,5

in-order: 0,1,2,3,4,5

post-order: 0,3,2,5,4,1

tree characters deep=5 weight=2



Click the button to create a new ordered tree.

9 57 Create tree

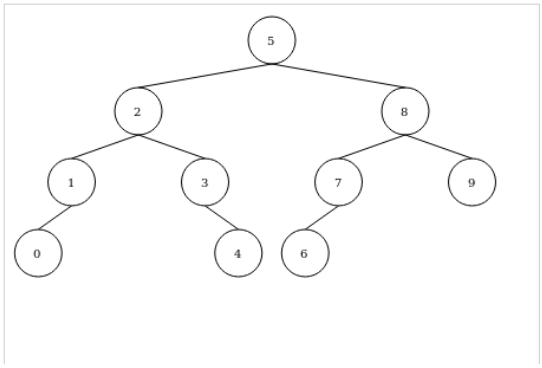
numbers: 5,2,8,7,6,3,1,4,9,0

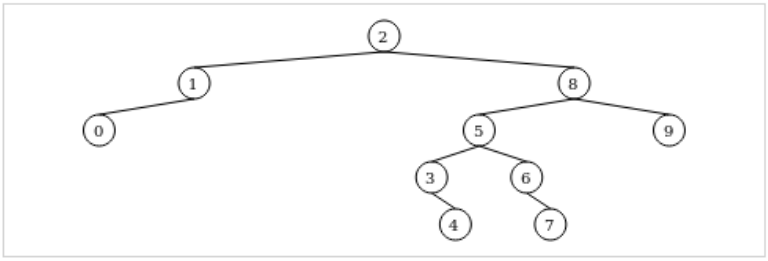
pre-order: 5,2,1,0,3,4,8,7,6,9

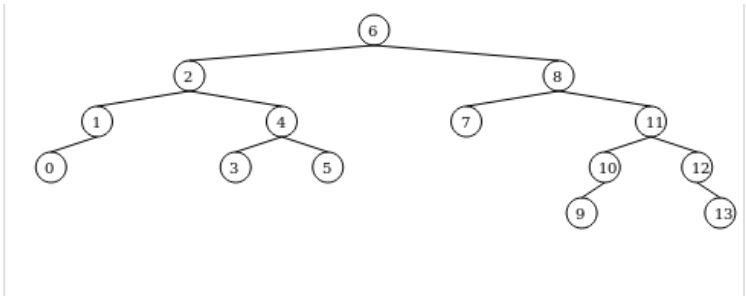
in-order: 0,1,2,3,4,5,6,7,8,9

post-order: 0,1,4,3,2,6,7,9,8,5

tree characters deep=5 weight=4





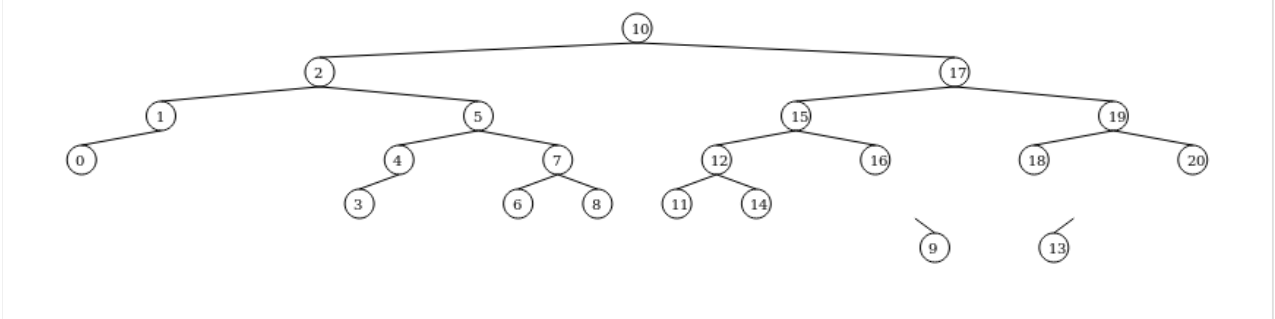


pre-order: 10,2,1,0,5,4,3,7,6,8,9,17,15,12,11,14,13,16,19,18,20

in-order: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20

post-order: 0,1,3,4,6,9,8,7,5,2,11,13,14,12,16,15,18,20,19,17,10

tree characters deep=7 weight=7



Выводы:
При выполнении работы были выполнены реализации деревьев в виде списков списков и связных списков, выявлено, что связные списки из элементов дерева проще реализовать и с ними проще работать. Реализовано создание структуры деревьев на двух языках программирования – java и javascript. Получен навык визуализации на javascript с использованием canvas, получены варианты визуализации ренерированных деревьев. Изучены прямой обход (NLR preorder), центрированный обход (LNR inorder), обратный обход (LRN postorder) как варианты поиска в глубину. Сделана попытка реализации поиска в ширину.
Общее применение

- управление иерархией данных;
- упрощение поиска информации (см. обход дерева);
- управление сортированными списками данных;
- синтаксический разбор арифметических выражений, оптимизация программ;
- в качестве технологии компоновки цифровых картинок для получения различных визуальных эффектов;
- форма принятия многоэтапного решения (деловые шахматы).

https://aliev.me/runestone/Trees/TreeTraversals.html
Отрисовка на с http://www.drdobbs.com/database/ternary-search-trees/184410528
Ternary search tree https://www.abc.se/~re/code/tst/tst_docs/classcontainers_1_1ternary__tree.html