МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра Вычислительной техники



**ОТЧЁТ**

**по лабораторной работе № 1**

«Разработка класса структуры данных двоичное дерево»

**по дисциплине: «***Технология разработки программного обеспечения***»**

Выполнили:Проверил:

Студенты гр. АММ2-21

*Арнольд Э.В. «Гаврилов А.В.»*

*Антонов С.С..*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (подпись)

Новосибирск

2021

1. ***Цели и задачи лабораторной работы***

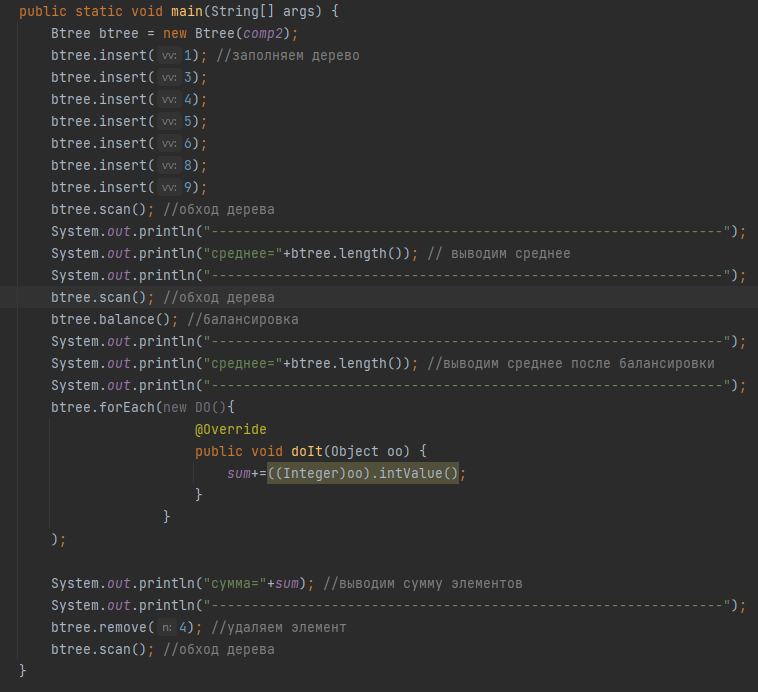
**Вариант 1:** Двоичное дерево.

Набор операций для структуры данных:

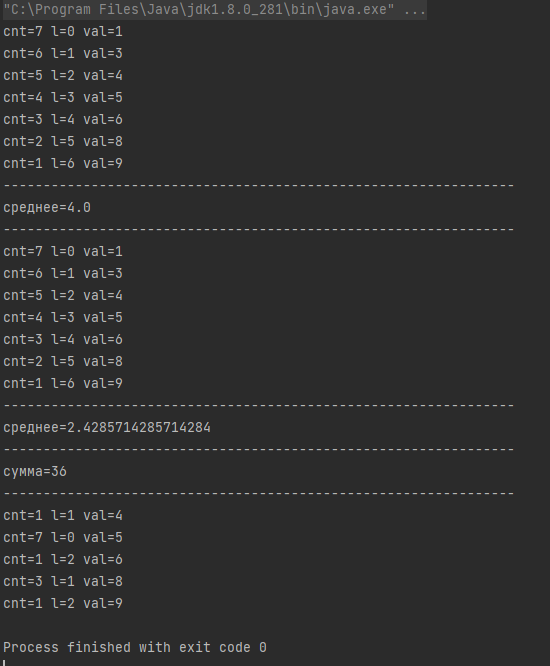
* добавление в конец, получение, вставка и удаление по логическому номеру (индексу). Логический номер – порядковый номер хранимого элемента в соответствии с последовательностью обхода структуры данных (перебора элементов)
* для исходно упорядоченной (например, «двоичное дерево») вставка с сохранением порядка, получение и удаление по по логическому номеру (индексу)
* Итератор forEach – для Java: интерфейс обратного вызова с методом void toDo(T v), итератора получает ссылку на интерфейсный объект (анонимный класс), обходит структуру данных и вызывает метод для каждого выбранного элемента. В Scala – формальный параметр = функция.
* сортировка, для исходно упорядоченной – балансировка = создание экземпляра структуры данных с равными размерностями компонент (например, сбалансированное дерево, либо список списков со списками нижнего уровня равной длины)

1. **Результат работы**

Запрос из Main:



Результат:



**Приложение 1**

package com.company;  
  
import java.util.Comparator;  
  
public class Node {  
 int cnt; //Количество вершин в дереве  
 Object val;  
 Node l, r; // Левый и правый потомки  
  
 Node(Object vv){  
 cnt=1;  
 val = vv;  
 r=l=null;  
 }  
 void scan(int level){  
 if (l!=null)  
 l.scan(level+1);  
 System.*out*.println("cnt="+cnt+" l="+level+" val="+val);  
 if (r!=null)  
 r.scan(level+1);  
 }  
 void forEach(DO fun){  
 if (l!=null)  
 l.forEach(fun);  
 fun.doIt(val);  
 if (r!=null)  
 r.forEach(fun);  
 }  
 int sum(int level){  
 int s = level;  
 if (r!=null)  
 s+=r.sum(level+1);  
 if (l!=null)  
 s+=l.sum(level+1);  
 return s;  
 }  
 void insert(Comparator comp,Object vv){  
 cnt++;  
 if(comp.compare(vv,val)<0){  
 if (l==null)  
 l = new Node(vv);  
 else  
 l.insert(comp,vv);  
 }  
 else{  
 if (r==null)  
 r = new Node(vv);  
 else  
 r.insert(comp,vv);  
 }  
 }  
  
 int createArray(Node pp[], int n){  
 if (l!=null)  
 n = l.createArray(pp, n); // В правое поддерево - собрать вершины  
 pp[n++] = this;  
 if (r!=null)  
 n = r.createArray(pp,n); // В правое поддерево - собрать вершин  
 return n;  
 }  
 // Балансировка дерева  
 static Node balance(Node pp[], int a, int b){  
 if (a > b) return null;  
 Node q;  
 if (a == b){  
 q = pp[a];  
 q.l = q.r = null;  
 q.cnt = 1;  
 return q;  
 }  
 int m = (a+b)/2;  
 q = pp[m]; // Корневая вершина – середина интервала  
 q.l = *balance*(pp, a, m-1); // Создание поддеревьев на правом и левом интервалах  
 q.r = *balance*(pp, m+1, b);  
 q.cnt = b - a + 1;  
 return q;  
 }  
  
  
 //---------------------------------------------------------------------------------------  
}

package com.company;  
  
public interface DO {  
 public void doIt(Object oo);  
}

package com.company;  
  
import java.util.Comparator;  
  
public class Btree {  
 Node root=null;  
 Comparator comparator;  
 Btree(Comparator cmp){  
 comparator = cmp;  
 }  
  
  
 void insert(Object vv){  
 if (root==null)  
 root = new Node(vv);  
 else  
 root.insert(comparator,vv);  
 }  
 void remove(Object n) {  
  
 Node currentNode = root;  
 Node parentNode = root;  
 boolean isLeftChild = true;  
 while (*comp2*.compare(n, currentNode.val) < 0) {  
 parentNode = currentNode;  
 if (*comp2*.compare(n, currentNode.val) < 0) {  
 isLeftChild = true;  
 currentNode = currentNode.l;  
 }  
 else { // или движение вправо?  
 isLeftChild = false;  
 currentNode = currentNode.r;  
 }  
 if (currentNode == null)  
 return; // yзел не найден  
 }  
 if (currentNode.l == null && currentNode.r == null) { // узел просто удаляется, если не имеет потомков  
 if (currentNode == root) // если узел - корень, то дерево очищается  
 root = null;  
 else if (isLeftChild)  
 parentNode.l = null; // если нет - узел отсоединяется, от родителя  
 else  
 parentNode.r = null;  
 }  
 else if (currentNode.r == null) { // узел заменяется левым поддеревом, если правого потомка нет  
 if (currentNode == root)  
 root = currentNode.l;  
 else if (isLeftChild)  
 parentNode.l = currentNode.l;  
 else  
 parentNode.r = currentNode.l;  
 }  
 else if (currentNode.l == null) { // узел заменяется правым поддеревом, если левого потомка нет  
 if (currentNode == root)  
 root = currentNode.r;  
 else if (isLeftChild)  
 parentNode.l = currentNode.r;  
 else  
 parentNode.r = currentNode.r;  
 }  
 else { // если есть два потомка, узел заменяется преемником  
 Node heir = receiveHeir(currentNode);// поиск преемника для удаляемого узла  
 if (currentNode == root)  
 root = heir;  
 else if (isLeftChild)  
 parentNode.l = heir;  
 else  
 parentNode.r = heir;  
 }  
 return; // элемент успешно удалён  
 }  
 // метод возвращает узел со следующим значением после передаваемого аргументом.  
 // для этого он сначала переходим к правому потомку, а затем  
 // отслеживаем цепочку левых потомков этого узла.  
 private Node receiveHeir(Node node) {  
 Node parentNode = node;  
 Node heirNode = node;  
 Node currentNode = node.r; // Переход к правому потомку  
 while (currentNode != null) // Пока остаются левые потомки  
 {  
 parentNode = heirNode;// потомка задаём как текущий узел  
 heirNode = currentNode;  
 currentNode = currentNode.l; // переход к левому потомку  
 }  
 // Если преемник не является  
 if (heirNode != node.r) // правым потомком,  
 { // создать связи между узлами  
 parentNode.l = heirNode.r;  
 heirNode.r = node.r;  
 }  
 return heirNode;// возвращаем приемника  
 }  
 void scan() {  
 if (root != null)  
 root.scan(0);  
 }  
 double length(){  
 if (root==null)  
 return 0;  
 return ((double)root.sum(1))/root.cnt;  
 }  
 static void insert(Node node,Comparator comp,Object vv){ // ЦИКЛ !!!!!!!!!!!!!!!!!!!!!!!!!!!!  
 while (true){  
 node.cnt++;  
 if(comp.compare(vv,node.val)<0){  
 if (node.l==null){  
 node.l = new Node(vv);  
 return;  
 }  
 else  
 node = node.l;  
 }  
 else{  
 if (node.r==null){  
 node.r = new Node(vv);  
 return;  
 }  
 else  
 node = node.r;  
 }  
 }  
 }  
 void balance(){  
 if (root==null) return;  
 Node pp[] = new Node[root.cnt];  
 root.createArray(pp,0);  
 root = Node.*balance*(pp,0,pp.length-1);  
 }  
 void forEach(DO fun){  
 if (root!=null)  
 root.forEach(fun);  
 }  
 //-----------------------------------------------------------------  
 static Comparator *comp* = new Comparator() {  
 @Override  
 public int compare(Object o1, Object o2) {  
 return ((Integer)o1).compareTo((Integer) o2);  
 }  
 };  
 static Comparator *comp2* = new Comparator() {  
 @Override  
 public int compare(Object o1, Object o2) {  
 return ((Integer)o1).compareTo((Integer) o2);  
 }  
 };  
 static int *sum*=0;  
  
  
 public static void main(String[] args) {  
 Btree btree = new Btree(*comp2*);  
 btree.insert(1); //заполняем дерево  
 btree.insert(3);  
 btree.insert(4);  
 btree.insert(5);  
 btree.insert(6);  
 btree.insert(8);  
 btree.insert(9);  
 btree.scan(); //обход дерева  
 System.*out*.println("----------------------------------------------------------------");  
 System.*out*.println("среднее="+btree.length()); // выводим среднее  
 System.*out*.println("----------------------------------------------------------------");  
 btree.scan(); //обход дерева  
 btree.balance(); //балансировка  
 System.*out*.println("----------------------------------------------------------------");  
 System.*out*.println("среднее="+btree.length()); //выводим среднее после балансировки  
 System.*out*.println("----------------------------------------------------------------");  
 btree.forEach(new DO(){  
 @Override  
 public void doIt(Object oo) {  
 *sum*+=((Integer)oo).intValue();  
 }  
 }  
 );  
  
 System.*out*.println("сумма="+*sum*); //выводим сумму элементов  
 System.*out*.println("----------------------------------------------------------------");  
 btree.remove(4); //удаляем элемент  
 btree.scan(); //обход дерева  
 }  
}