Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

**Лабораторна робота №** 5

по дисципліні «Теорія алгоритмів»

на тему: Пошукові структури даних. Бінарні дерева пошуку

|  |  |
| --- | --- |
| Виконав:  студент групи ІА-з91  Москаленко В.В.  Дата здачі \_\_\_\_\_\_\_\_\_\_\_\_  Захищено з балом \_\_\_\_\_\_\_\_\_\_\_ | Перевірив:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Київ 2020

**0. Завдання.**

Пошукові структури даних – бінарні дерева пошуку

Заміри, які необхідно здійснити:

1. час наповнення різних дерев елементами. (1000 елементів, або більше)
   1. послідовні елементи
   2. випадкові елементи( широке розподілення випадкових елементів, не від 0 до 100, а з великими числами)
2. час пошуку в
   1. не збалансованому бінарному дереві
   2. збалансованому бінарному дереві
3. **Постановка проблеми.**

Необхідно реалізувати бінарне дерево, наповнити його масивом з 1тис. елементів, отримати заміри часу балансування для послідовних та випадкових елементів, час пошуку в збалансованому та не збалансованому бінарному дереві, час видалення елементів з або без перебалансування. Для даного алгоритму є стандартні користувацькі структури даних та методи.

1. **Побудова моделі.**

Бінарне дерево пошуку будується за наступними правилами. У кожного вузла не більше двох дітей, будь яке, значення менше значення вузла стає лівим нащадком, або нащадка лівого нащадка. Будь яке значення більше або дорівнює значенню вузла стає правим нащадком або нащадком правого нащадка.

1. **Розроблення алгоритму.**

Для початку вибираємо значення, яке стане корнем дерева. Вузли додаються методом add, котрий приймає значення, що додається. Послідовне порівняння значення, що додається з потенціальним батьківським значенням буде продовжуватись до тих пір поки не буде знайдено місце для вставки і повторюється для кожного значення, що вставляється до тих пір, поки не буде побудоване все дерево повністю. Механізм видалення елементу з дерева обернено пропорційний до механізму додавання елементу. Пошук елемента в бінарному дереві починається з батьківського елемента. Алгоритм пошуку буде мати вигляд порівняння елементу, що шукають з кожним нащадком і відповідний зсув вліво або вправо сторони.

1. **Перевірка правильності алгоритму.**

Для того, щоб перевірити, що задане дерево є деревом пошуку, використовується обхід в глубину. Запустивши від корня рекурсивну логічну функцію, котра видає true, якщо дерево є бінарним деревом пошуку. Дерево не є бінарним деревом пошуку, якщо в ньому є хоча б одна вершина, котра не підпадає під визначення дерева пошуку.

1. **Реалізація алгоритму.**

import java.io.Console;  
import java.util.ArrayList;  
import java.util.List;  
  
  
public class Lab5{  
public static void main(String[] args) {  
Tree<Integer> tree = new Tree<>(33, null);  
tree.add(5, 35, 1, 20, 4, 17, 31, 99, 18, 19);  
  
tree.print();  
  
long start = System.nanoTime();  
[tree.search](http://tree.search/)(4);  
long finish = System.nanoTime();  
long ans = finish - start;  
System.out.println("Search speed: "+ ans);  
  
tree.remove(33);  
tree.print();  
  
System.out.println(tree);  
System.out.println(tree.left());  
System.out.println(tree.left().left());  
System.out.println(tree.right().left());  
}  
}  
  
class Tree<T extends Comparable<T>> {  
private T val;  
private Tree left;  
private Tree right;  
private Tree parent;  
private List<T> listForPrint = new ArrayList<>();  
  
public T val() {  
return val;  
}  
  
public Tree left() {  
return left;  
}  
  
public Tree right() {  
return right;  
}  
  
public Tree parent() {  
return parent;  
}  
  
public Tree(T val, Tree parent) {  
this.val = val;  
this.parent = parent;  
}  
  
  
public void add(T...vals){  
long start = System.nanoTime();  
for(T v : vals){  
add(v);  
}  
long finish = System.nanoTime();  
long timeConsumedMillis = finish - start;  
System.out.print("Add speed: ");  
System.out.println(timeConsumedMillis + "\n");  
}  
  
public void add(T val){  
long start = System.nanoTime();  
if(val.compareTo(this.val) < 0){  
if(this.left==null){  
this.left = new Tree(val, this);  
}  
else if(this.left != null)  
this.left.add(val);  
}  
else{  
if(this.right==null){  
this.right = new Tree(val, this);  
}  
else if(this.right != null)  
this.right.add(val);  
}  
long finish = System.nanoTime();  
long ans = finish - start;  
System.out.println(" Add speed current element: " + ans);  
}  
  
private Tree<T> \_search(Tree<T> tree, T val){  
if(tree == null) return null;  
switch (val.compareTo(tree.val)){  
case 1: return \_search(tree.right, val);  
case -1: return \_search(tree.left, val);  
case 0: return tree;  
default: return null;  
}  
}  
  
public Tree<T> search(T val){  
return \_search(this, val);  
}  
  
public boolean remove(T val){  
long start = System.nanoTime();  
Tree<T> tree = search(val);  
if(tree == null){  
return false;  
}  
Tree<T> curTree;  
  
if(tree == this){  
if(tree.right!=null) {  
curTree = tree.right;  
}  
else curTree = tree.left;  
  
while (curTree.left != null) {  
curTree = curTree.left;  
}  
T temp = curTree.val;  
this.remove(temp);  
tree.val = temp;  
long fin1 = System.nanoTime();  
long ans = fin1 - start;  
System.out.println("Remove speed: "+ ans);  
return true;  
}  
  
if(tree.left==null && tree.right==null && tree.parent != null){  
if(tree == tree.parent.left)  
tree.parent.left = null;  
else {  
tree.parent.right = null;  
}  
return true;  
}  
  
if(tree.left != null && tree.right == null){  
tree.left.parent = tree.parent;  
if(tree == tree.parent.left){  
tree.parent.left = tree.left;  
}  
else if(tree == tree.parent.right){  
tree.parent.right = tree.left;  
}  
return true;  
}

if(tree.left == null && tree.right != null){  
tree.right.parent = tree.parent;  
if(tree == tree.parent.left){  
tree.parent.left = tree.right;  
}  
else if(tree == tree.parent.right){  
tree.parent.right = tree.right;  
}  
return true;  
}  
  
if(tree.right!=null && tree.left!=null) {  
curTree = tree.right;  
  
while (curTree.left != null) {  
curTree = curTree.left;  
}  
  
if(curTree.parent == tree) {  
curTree.left = tree.left;  
tree.left.parent = curTree;  
curTree.parent = tree.parent;  
if (tree == tree.parent.left) {  
tree.parent.left = curTree;  
} else if (tree == tree.parent.right) {  
tree.parent.right = curTree;  
}  
return true;  
}  
else {  
if (curTree.right != null) {  
curTree.right.parent = curTree.parent;  
}  
curTree.parent.left = curTree.right;  
curTree.right = tree.right;  
curTree.left = tree.left;  
tree.left.parent = curTree;  
tree.right.parent = curTree;  
curTree.parent = tree.parent;  
if (tree == tree.parent.left) {  
tree.parent.left = curTree;  
} else if (tree == tree.parent.right) {  
tree.parent.right = curTree;  
}  
  
return true;  
}  
}  
return false;  
}  
  
  
private void \_print(Tree<T> node){  
if(node == null) return;  
\_print(node.left);  
listForPrint.add(node.val);  
System.out.print(node + " ");  
if(node.right!=null)  
\_print(node.right);  
}  
  
public void print(){  
listForPrint.clear();  
\_print(this);  
System.out.println();  
}  
  
[@Override](https://web.telegram.org/" \l "/im?p=@Override)  
public String toString() {  
return val.toString();  
}  
}

1. Начало формы
2. Конец формы

**6. Перевірка програми.**

Даний алгоритм протестований за допомогою візуального відображення у консолі. Даний алгоритм реалізовано у вигляді збалансованого та незбалансованого дерев.

**7. Вивід програми**

