

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

«Дальневосточный федеральный университет»

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Департамент программной инженерии и искусственного интеллекта

|  |
| --- |
| **О Т Ч Е Т** |
| по лабораторной работе №2  дисциплина «Фундаментальные структуры данных и алгоритмы» |

|  |  |  |
| --- | --- | --- |
|  |  | Студент группы Б9123-09.03.04  \_\_\_\_\_\_\_\_\_\_\_\_\_А. В. Ельчанинов  (подпись) |
|  |  | Руководитель  Ассистент ДПИиИИ  Шулятьев А.А. |

Владивосток

2024

Неформальная постановка задачи

Реализовать программу, решающую следующую задачу:

Сделать красно черное дерево с возможностью:

1. Вставки элементов.
2. Инициализацию
3. Удаление элемента.
4. Поиск элемента.
5. Печать дерева.

**Описание типа + спецификация подпрограмм + тесты**

* function RBTree.add(series: Int, number: Int): Unit

Вставляет в дерево с корнем root типа Node элемент типа Passport.

Входные данные: класс RBTree, значение, значение.

Выходные данные: класс RBTree c элементом с элементом Passport(series, number)

|  |  |  |
| --- | --- | --- |
| **Описание** | **Входные данные** | **Выходные данные** |
| Вставка элемента в дерево | series=1 number=2  [tree = ] | series=1 number=2  [tree = ] |
| Вставка элемента в дерево | series=1 number=2  [tree = ] | series=1 number=2  [tree = ] |
| Вставка элемента в дерево | series=1 number=2  [tree = ] | series=1 number=2  [tree = ] |

* function RBTree.delete(series: Int, number: Int): Unit

Удаляет в дереве с корнем root типа Node элемент со значением Passport(series: Int, Int number: Int)

Входные данные: класс RBTree , значение, значение.

Выходные данные: класс RBTree без элементов со значением Passport(series, number)

|  |  |  |
| --- | --- | --- |
| **Описание** | **Входные данные** | **Выходные данные** |
| Удаление по значению из дерева | series=1 number=2  [tree = ] | series=1 number=2  [tree = ] |
| Удаление по значению из дерева | series=1 number=2  [tree = ] | series=1 number=2  [tree = ] |
| Удаление по значению из дерева | series=1 number=2  [tree = ] | series=1 number=2  [tree = ] |

* function RBTree.search(series: Int, number: Int): Unit

Ищет и возвращает элемент типа Node в дереве с корнем root типа Node по Passport(series: Int, Int number: Int)

Входные данные: класс RBTree, значение, значение.

Выходные данные: Node

|  |  |  |
| --- | --- | --- |
| **Описание** | **Входные данные** | **Выходные данные** |
| Поиск по значению из дерева | series=1 number=2  [tree = ] | Node(  passport = Passport(series=,number=), color, left, right, parent  ) |
| Поиск по значению из дерева | series=1 number=2  [tree = ] | Node(  passport = Passport(series=,number=), color, left, right, parent  ) |
| Поиск по значению из дерева | series=1 number=2  [tree = ] | Node(  passport = Passport(series=,number=), color, left, right, parent  ) |
| Поиск по значению из дерева | series=1 number=2  [tree = ] | Node(  passport = Passport(series=,number=), color, left, right, parent  ) |

* function RBTree.lrPrint ():

Выводить дерево с корнем root типа Node обходом с лево на право

Входные данные: класс RBTree

Выходные данные: -

|  |  |  |
| --- | --- | --- |
| **Описание** | **Входные данные** | **Выходные данные** |
| Вывод дерева | [list = 1 3 4 4 5] | -  [list =] |
| Вывод дерева | [list = 1 1 1 1] | -  [list =] |

Алгоритм программы на языке PDL

class RBTree(private var root: Node? = null) {

enum class Color {

RED, BLACK

}

class Passport(series: Int, number: Int) {

var series: Int = series % 10000

var number: Int = number % 1000000

}

data class Node(

var passport: Passport,

var color: Color = Color.RED,

var left: Node? = null,

var right: Node? = null,

var parent: Node? = null

) {

operator fun compareTo(other: Node): Int {

return when {

this.passport.series < other.passport.series -> -1

this.passport.series > other.passport.series -> 1

this.passport.number < other.passport.number -> -1

this.passport.number > other.passport.number -> 1

else -> 0

}

}

override operator fun equals(other: Any?): Boolean {

if (this === other) return true

if (other !is Node) return false

return this.passport.series == other.passport.series && this.passport.number == other.passport.number

}

override fun toString(): String {

return "${this.passport.series} " +

"${this.passport.number} ${this.color} ${this.left?.passport?.series} " +

"${this.right?.passport?.series} ${this.parent?.passport?.series}"

}

}

private fun Node.uncleL(): Node? = this.parent?.parent?.left

private fun Node.uncleR(): Node? = this.parent?.parent?.right

private fun Node.grand(): Node? = this.parent?.parent

private fun Node.bratR(to: Node? = null): Node? {

if (to != null) this.parent?.right = to

return this.parent?.right

}

private fun Node.bratL(to: Node? = null): Node? {

if (to != null) this.parent?.left = to

return this.parent?.left

}

private fun cutNode(before: Node, after: Node?) {

when {

before.parent == null -> root = after

before == before.parent?.left -> before.parent?.left = after

else -> before.parent?.right = after

}

after?.parent = before.parent

}

private fun minimum(node: Node): Node {

var current = node

while (current.left != null) {

current = current.left!!

}

return current

}

private fun leftRotate(x: Node) {

val y = x.right ?: return

x.right = y.left

y.left.let { it?.parent = x }

y.parent = x.parent

when {

x.parent == null -> root = y

x == x.bratL() -> x.bratL(y)

else -> x.bratR(y)

}

y.left = x

x.parent = y

}

private fun rightRotate(y: Node) {

val x = y.left ?: return

y.left = x.right

x.right.let { it?.parent = y }

x.parent = y.parent

when {

x.parent == null -> root = x

y == y.bratL() -> y.bratL(x)

else -> y.bratR(x)

}

x.right = y

y.parent = x

}

fun add(series: Int, number: Int) {

val newNode = Node(Passport(series, number))

var x: Node? = root

while (x?.left != null || x?.right != null) {

x = if (newNode < x) x.left else x.right

}

newNode.parent = x

if (x == null) root = newNode

else if (newNode < x) x.left = newNode

else x.right = newNode

insert(newNode)

}

private fun insert(node: Node) {

var z = node

while (z.parent?.color == Color.RED) {

if (z.parent == z.uncleL()) {

val y = z.uncleR()

if (y?.color == Color.RED) {

z.parent?.color = Color.BLACK

y.color = Color.BLACK

z.grand()?.color = Color.RED

z = z.grand() ?: return

} else {

if (z == z.parent?.right) {

z = z.parent!!

leftRotate(z)

}

z.parent?.color = Color.BLACK

z.grand()?.color = Color.RED

rightRotate(z.grand()!!)

}

} else {

val y = z.grand()?.left

if (y?.color == Color.RED) {

z.parent?.color = Color.BLACK

y.color = Color.BLACK

z.grand()?.color = Color.RED

z = z.grand() ?: return

} else {

if (z == z.parent?.left) {

z = z.parent!!

rightRotate(z)

}

z.parent?.color = Color.BLACK

z.grand()?.color = Color.RED

leftRotate(z.grand()!!)

}

}

}

root?.color = Color.BLACK

}

fun delete(series: Int, number: Int) {

search(series, number)?.let { exclude(it) }

}

private fun exclude(node: Node) {

val temp: Node?

if (node.left == null || node.right == null) {

temp = node.right ?: node.left

cutNode(node, temp)

} else {

val successor = minimum(node.right!!)

node.passport = successor.passport

temp = successor.right

cutNode(successor, temp)

}

temp.let { this.fixDelete(it) }

}

private fun fixDelete(node: Node?): Node? {

var z = node

while (z != root && z?.color == Color.BLACK) {

println(z.toString())

z = if (z == z.bratL()) {

fixByBrat(z)(true)

} else {

fixByBrat(z)(false)

}

}

z?.color = Color.BLACK

return z

}

private fun fixByBrat(z: Node): (Boolean) -> Node? = { isLeft ->

var brat = if (isLeft) z.bratR() else z.bratL()

val action = { z: Node? -> if (isLeft) z?.bratR() else z?.bratL() }

val rotate1 = { z: Node -> if (isLeft) leftRotate(z) else rightRotate(z) }

val rotate2 = { z: Node -> if (isLeft) rightRotate(z) else leftRotate(z) }

val son1 = if (isLeft) brat?.right else brat?.left

val son2 = if (isLeft) brat?.left else brat?.right

if (brat?.color == Color.RED) {

brat.color = Color.BLACK

z.parent?.color = Color.RED

rotate1(z.parent!!)

brat = action(z)

}

if (son1?.color == Color.BLACK && son2?.color == Color.BLACK) {

brat?.color = Color.RED

z.parent

} else {

if (son1?.color == Color.BLACK) {

son2?.color = Color.BLACK

brat?.color = Color.RED

rotate2(brat!!)

brat = action(z)

}

brat?.color = z.parent?.color!!

z.parent?.color = Color.BLACK

son1?.color = Color.BLACK

rotate1(z.parent!!)

root

}

}

fun search(series: Int, number: Int): Node? {

val node = Node(Passport(series, number))

var z = root

while (z != null && node != z) {

z = if (node < z) z.left else z.right

}

return z

}

fun print() = printTree(root, " ", true)

fun lrPrint() = leftRightOrder(root)

private fun leftRightOrder(root: Node?) {

if (root != null) {

leftRightOrder(root.left)

print("${root.passport.series}->")

leftRightOrder(root.right)

}

}

private fun printTree(node: Node?, indent: String, isLeft: Boolean) {

if (node == null) return

println("$indent${if (isLeft) "L--" else "R--"}$node")

printTree(node.left, "$indent ", true)

printTree(node.right, "$indent ", false)

}

}

fun main() {

val tree = RBTree()

for (i in 1..8) {

tree.add(i, i \* i)

}

tree.delete(8, 64)

tree.print()

tree.lrPrint()

println(tree.search(4, 16)?.passport?.series)

}