МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

**ОТЧЕТ**

**по учебной практике**

Реализовать структуру данных «Бинарные деревья поиска» и основные функции для работы с ним. Разработать тестовое приложение для демонстрации реализованных функции

тема

студента Ванина Константина Евгеньевича группы АВТ-918

ФИО студента(ки)

Место проведения практики: Новосибирский государственный технический

полное наименование предприятия (организации)

университет

Сроки практики по учебному плану: с 1.09.21. по 30.12.21.

Руководитель практики от университета:

подпись , оценка

Перышкова Евгения Николаевна, к.т.н.

ФИО, должность

Оценка по итогам аттестации студента

НОВОСИБИРСК

2021

Содержание

[Введение 3](#_Toc91340346)

[1. Анализ задачи 4](#_Toc91340347)

[1.1. Структуры BST-дерева 4](#_Toc91340348)

[1.2. Операции в BST-дереве 5](#_Toc91340349)

[1.2.1. Вставка элемента в дерево 6](#_Toc91340350)

[1.2.2. Поиск элемента в бинарном дереве 7](#_Toc91340351)

[1.2.3. Удаление элемента из бинарного дерева 8](#_Toc91340352)

[1.3. Обходы дерева 10](#_Toc91340353)

[2. Тестовые данные 11](#_Toc91340354)

[Заключение 15](#_Toc91340355)

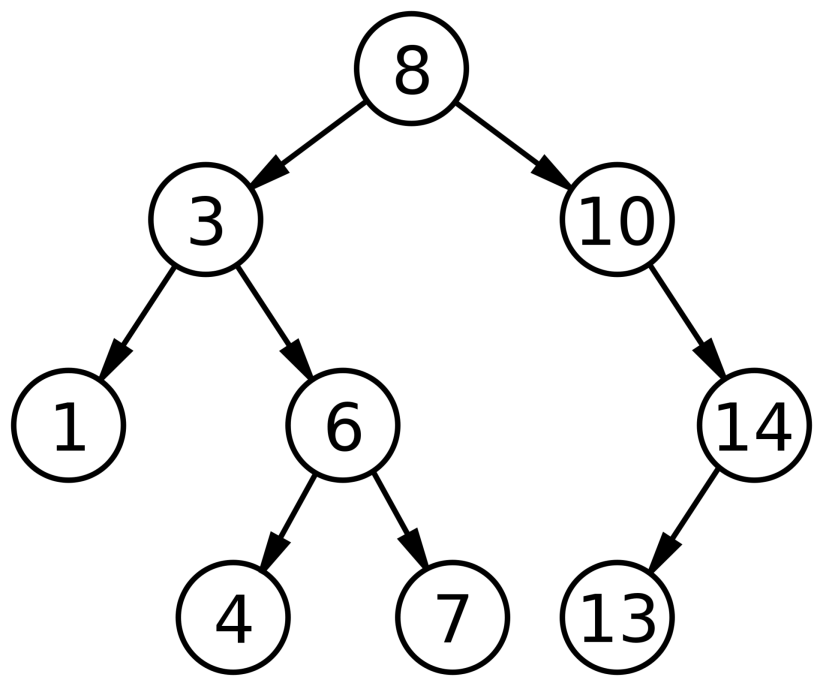
[Список источников 16](#_Toc91340356)

[Приложение 17](#_Toc91340357)

# Введение

Бинарное дерево поиска (BST - дерево) – это упорядоченное дерево, для которого выполняются следующие свойства:

* Каждая вершина (узел) которого имеет не более двух потомков, причем каждый из потомков считается либо «*левый сыном*» либо «*правым сыном*» своего «*родителя*».
* Все вершины обладают *ключами*, на которых определена операция сравнения.
* У всех вершин *левого* поддерева произвольного узла X значение ключей данных меньше, нежели значение ключа данных самого узла X.
* У всех вершин *правого* поддерева произвольного узла X значение ключей данных больше или равны, нежели значение ключа данных самого узла X.



*Рис. 1 – Пример бинарного дерево поиска*

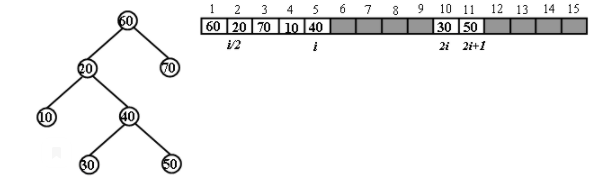
Как абстрактный тип данных, BST-дерево предусматривает операции поиска, вставки и удаления элементов по ключу. А также есть три операции обхода узлов дерева. Используя эти операции можно построить любое бинарное дерево [1].

# Анализ задачи

## Структуры BST-дерева

Для начала разберем структуру BST-дерева. Для хранения BST-деревьев используются две альтернативные структуры – массив элементов дерева и связная структура дерева на базе адресных указателей.

*Массив элементов*  используется в случае, если задано предельное количество элементов (размер дерева), которые будут размещены в BST-дереве. В этом случае для хранения дерева выделяется массив заданного размера, тип которого совпадает с типом элементов множества, хранящегося в дереве. Корень дерева размещается в элементе массива с индексом 1, а его левый и правый сыновья – с индексом 2 и 3 соответственно. По индексу любого элемента легко вычислить индексы его сыновей и родителя в массиве. Если некоторый элемент имеет индекс i, то его левый сын расположен по индексу 2i, а правый сын – по индексу 2i+1. Индекс родителя вычисляется, как целая часть от деления i/2 [1].

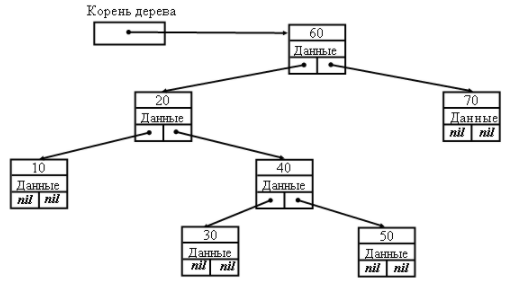


*Рис. 2 – Структура BST-дерева массива элемента.*

Как видно из примера, использование памяти массива хранение дерева неэффективны.

Наиболее распространённой формой для хранения деревьев является *связная структура на базе адресных указателей*. Каждый *узел дерева*  размещается в динамической памяти и содержит помимо ключа и данных два *указателя на левого и правого**сыновей*. Таким образом, структура через указатели на сыновей обеспечивает спуск по дереву от корня к местоположению узла с заданным значением ключа. Для некоторых операций требуется подъём от некоторого узла к родительскому узлу и далее к корню. Для этого в структуру узла можно ввести дополнительный *указатель на родителя*.  Но это приводит к неоправданным затратам памяти на дополнительные указатели в узлах. Поэтому эти указатели используются только в отдельных разновидностях деревьев. Как правило, естественное поведение алгоритма операции, обеспечивает возврат по ранее пройденному пути в дереве [1].

В дерево вводится вспомогательный указатель, содержащий адрес корневого узла. Если дерево пусто, то указатель содержит значение nil .



*Рис. 3 – Структура BST-дерева на базе адресных указателей.*

## Операции в BST-дереве

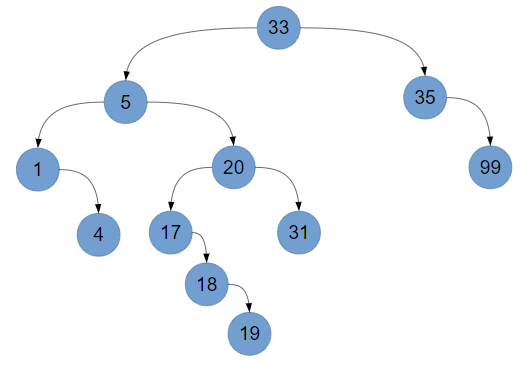
Операции вставки, удаления и поиска элементов для BST – дерева используют правило двоичного поиска при доступе к элементу с заданным значением ключа. Поэтому трудоёмкость этих операций соответствует трудоёмкости бинарного поиска в упорядоченном множестве и имеет нотацию O(log n). Необходимо отметить структурную зависимость BST – дерева от порядка поступления и удаления элементов.

Также важнейшей операцией для бинарного дерева поиска является перебор его элементов в определенном порядке. Для этого существует три основные схемы обхода – прямой, симметричный и обратный. Теперь разберем каждый и них по отдельности на примере.

### Вставка элемента в дерево

Создадим произвольное бинарное дерево с корневым элементом 33 и добавим в него элементы в следующей последовательности: 5, 35, 1, 20, 99, 17, 18, 19, 31, 4. При добавлении элемента x в дерево проверяем значение текущего узла.

* Если значение добавляемого элемента x меньше значения текущего узла, спускаемся к левому поддереву. Если его не существует, то создаем его и присваиваем значение x. Если существует, то обозначим левое поддерево как текущий узел и повторим сначала [2].
* Если значение добавляемого элемента x больше или равно значению текущего узла, спускаемся к правому поддереву. Если его не существует, то создаем его и присваиваем значение x. Если существует, то обозначим правое поддерево как текущий узел и повторим сначала [2].

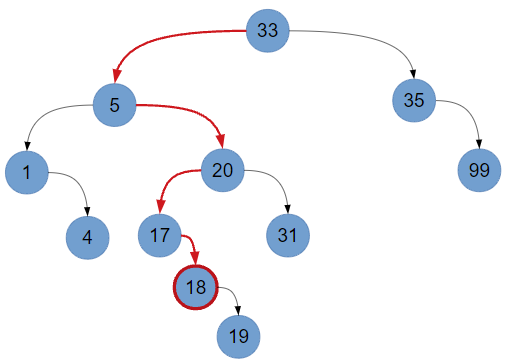


*Рис. 4 – Пример добавления элемента в дерево*

### Поиск элемента в бинарном дереве

Поиск начинаем с родительского элемента. Допустим, мы ищем значение 18 (обозначим его за x*x*). Алгоритм поиска будет иметь следующий вид:

1. x<33*x*<33 — спускаемся в левое поддерево;
2. x>5*x*>5 — спускаемся в правое поддерево;
3. x<20*x*<20 — спускаемся в левое поддерево;
4. x>17*x*>17 — спускаемся в правое поддерево;
5. x=18*x*=18 — мы нашли элемент.



*Рис. 5 – Пример поиска элемента в бинарном дереве*

### Удаление элемента из бинарного дерева

**Удаление листьев**

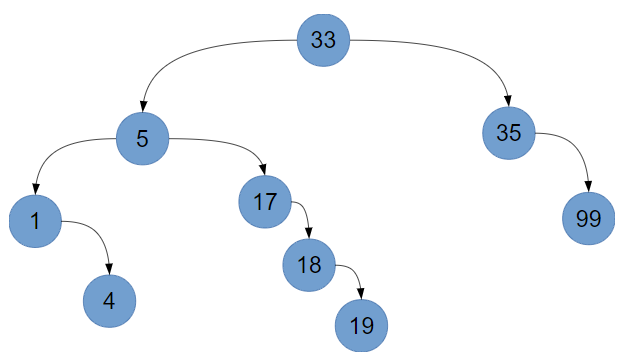
Если удаляемый элемент является листом, то просто удаляем у его родителя ссылку на этот элемент (например на значение 31). Удалим его [2].

**Удаление узла, имеющего левого поддерево, но не имеющее правого поддерева**

После удаления 31 элементом, имеющим левое поддерево, но не имеющим правого поддерева является элемент 20. Удалим его из дерева:

1. Указываем, что родителем элемента 17 теперь будет элемент 5.
2. Указываем, что правым потомком элемента 5 теперь является элемент 17.

После удаления значений 31 и 20 дерево приобретает такой вид:

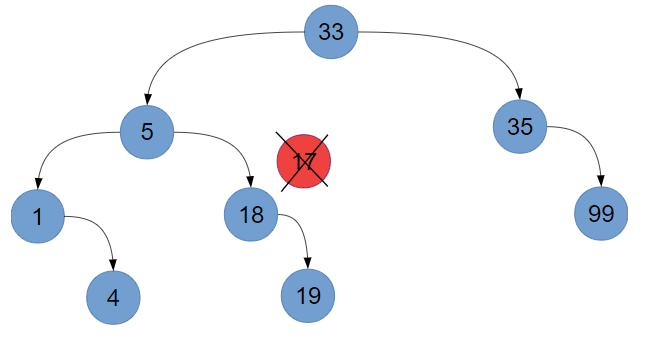


*Рис. 6 – Пример удаление узла, имеющего левого поддерева, но не имеющее правого поддерева*

**Удаление узла, имеющего правого поддерево, но не имеющее левого поддерева**

1. Удалим элемент 17. Присвоим его правому поддереву в качестве родителя элемент 5.
2. Элементу 5 укажем, что его правым поддеревом теперь является элемент 18.

Получим следующую картину:



*Рис. 7 – Пример удаление узла, имеющего правого поддерева, но не имеющее левого поддерева*

## Обходы дерева

Многие алгоритмы, работая с бинарными корневыми деревьями, посещают один раз в определенном порядке каждую вершину дерева. В зависимости от алгоритма операции существуют три наиболее распространенных способа обхода вершин бинарного дерева: *прямой*, *обратный* и *внутренний* [3].

* Прямой (*префиксный*) обход (сверху вниз), при котором обрабатывается значение в узле, а затем значения из левого и правого поддеревьев узла.
* Обратный (*постфиксный*) обход (снизу вверх), при котором обрабатывается значение из левого и правого поддеревьев, а затем значение узла.
* Внутренний (*инфиксный*) обход (слева направо), при котором обрабатываются значения из левого поддерева, затем значения узла, затем значения из правого поддерева.

Разберем пример бинарного дерева поиска из рисунка 1. На этом примере разберем все три обхода.

1. Прямой порядок обхода начинается сверху вниз. Обход заключается в том, что корень некоторого дерева посещается раньше, чем его поддеревья. При выполнении данного обхода вершины выведены следующем порядке: 8, 3, 1, 6, 4, 7, 10, 14, 13.
2. Обратный порядок обхода начинается снизу вверх. Обход заключается в том, что корень дерева посещается после его поддеревьев. При выполнении данного обхода вершины выведены следующем порядке: 1, 4, 7, 6, 3, 13, 14, 10, 8.
3. Внутренний порядок обхода начинается слева направо или справа налево. Обход заключается в том, что корень посещается после посещения одного из его поддеревьев. При выполнении данного обхода вершины выведены следующем порядке: 1, 3, 4, 6, 7, 8, 10, 13, 14.

# Тестовые данные

Операция поиска получает на вход заданный ключ k и указатель t на корень поддерева, в котором производится поиск. Операция возвращает данные с ключом k  или сообщение об ошибке, если узла с таким ключом нет в дереве.

**BST\_Search(*t, k*)**

http://edu.nstu.ru/courses/saod/images/comment.gif ***t*** – корень дерева или поддерева

http://edu.nstu.ru/courses/saod/images/comment.gif ***k*** – значение искомого ключа

**if** ***t*** **=** ***nil***

**then**  **return**"error"

**if** ***k*** = key[***t***]

**then** **return** data[***t***]

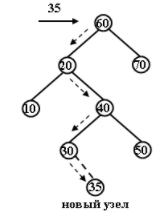
**if** ***k*** < key[t]

**then return** **BST\_Search**(left[***t***], ***k***)

**else return** **BST\_Search**(right[***t***], ***k***)

В процессе поиска операция движется от корня, сравнивая ключ k с ключом key, хранящимся в текущем узле. Если они равны, или адрес узла равен nil, то поиск заканчивается. Если k < key[t], то поиск продолжается в левом поддереве узла t (ключ может быть только там в силу свойства упорядоченности). Если k>key[t], то поиск продолжается в правом поддереве. Длина пути поиска не превосходит высоты дерева h, и время поиска есть О(h). Но, высота дерева h может колебаться от log2n до n в случае вырождения BST- дерева.

Операция вставки добавляет заданный элемент в подходящее место дерева t, сохраняя свойство упорядоченности.



*Рис. 8 – Операция вставки*

Параметрами операции являются указатель на корень дерева или поддерева t, ключ и данные нового элемента, а также переменная insert для записи признака вставки.

**BST\_Insert(t, k, data, inserted)**

http://edu.nstu.ru/courses/saod/images/comment.gif ***t*** – корень дерева или поддерева

http://edu.nstu.ru/courses/saod/images/comment.gif ***k*** –ключ элемента

http://edu.nstu.ru/courses/saod/images/comment.gif ***data*** – данные элемента

http://edu.nstu.ru/courses/saod/images/comment.gif ***inserted*** – возвращаемый признак вставки

**if** ***t*** = ***nil***

**then** ***inserted*** ← TRUE

**return** **Create\_Node**(***k***, ***data***)

**if** ***k*** = key[***t***]

**then** ***inserted*** ← FALSE

**return** ***t***

**if** ***k*** < key[***t***]

**then**  left[***t***] ← **BST\_Insert**(left[***t***], ***k, data***, ins)

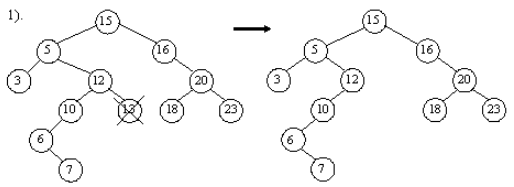
**else** right[***t***] ← **BST\_Insert**(right[***t***], ***k, data***, ins)

***inserted*** ← ins

**return** ***t***

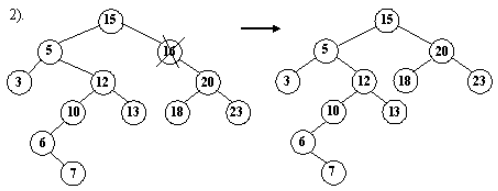
Параметром операции удаления является ключ удаляемого узла. При удалении возможны три случая.

Если у удаляемого узла нет сыновей, для удаления узла достаточно поместить адрес nil в соответствующее поле его родителя вместо адреса удаленного узла.



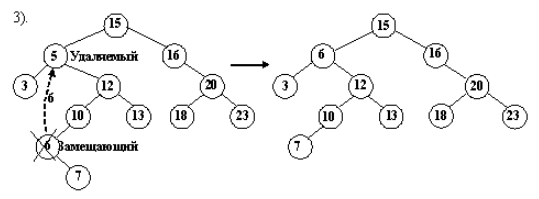
*Рис. 9 – Удаление узла, если нет сыновей.*

Если удаляемого узла  есть один сын, можно его “вырезать”, соединив его родителя с его  сыном.



*Рис. 10 – Удаление узла, соединив его родителя с его сыном*

Если же у удаляемого узла двое сыновей, требуются некоторые действия: сначала находится следующий по порядку ключей замещающий узел.



*Рис. 11 – Удаление узла, путем замены следующей по порядку*

**BST\_Delete(t, k ,deleted)**

http://edu.nstu.ru/courses/saod/images/comment.gif ***t*** – корень дерева или поддерева

http://edu.nstu.ru/courses/saod/images/comment.gif ***k*** – значение удаляемого ключа

http://edu.nstu.ru/courses/saod/images/comment.gif ***deleted*** – возвращаемый признак

**if** ***t*** = ***nil***

**then *deleted*** ← FALSE

**return** ***t***

**if** ***k*** < key[***t***]

        t**hen**  left[***t***] ← **BST\_Delete**(left[***t***], ***k,***del)

***deleted*** ← del

**return** ***t***

**if** ***k*** > key[***t***]

**then**  right[***t***] ← **BST\_Delete**(right[***t***], ***k***, del)

***deleted*** ← del

**return** ***t***

***deleted*** ← TRUE

**if** left[***t***] = ***nil*** **and** right[***t***] = ***nil***

**then** **Delete\_Node** ( ***t )***

**return** ***nil***

**if** left[***t***] **=** ***nil***

**then**  x ← right[***t***]

**Delete\_Node** ( ***t )***

**return** x

**if** right[***t***] = ***nil***

**then** x ← left[***t***]

**Delete\_Node** ( ***t )***

**return** x

 right[***t***] ← **Del**(right[***t***], ***t***)

**return** ***t***

**Del(*t, t*0)**

**if** left[***t***] ≠ ***nil***

**then**  left[***t***] ← **Del**(left[***t***], ***t*0**)

**return** ***t***

     key[***t*0**] ← key[***t***]

     data[***t*0**] ← data[***t***]

     x ← right[***t***]

**Delete\_Node (*t)***

**return** x

# Заключение

В ходе учебной практики были изучены «Бинарные деревья поиска» и их основные операции:

* Поиск
* Вставка
* Удаление

А также были рассмотрены три основные схемы обхода:

* Прямой
* Обратный
* Внутренний

На основе этих операции привели примеры по каждому из них. При проектировании алгоритмов операции для BST-дерева использовали язык программирования C#. Полный код находится в разделе «Приложение».

# Список источников

1. Электронное учебное пособие адресовано студентам, обучающимся в Новосибирском Государственном Техническом Университете. Структуры и алгоритмы обработки данных. [Электронный ресурс]. URL: <http://edu.nstu.ru/courses/saod/index.htm> (Дата обращения: 17.12.2021).
2. Бинарное дерево – двоичное дерево поиска. Основные операции с бинарными деревьями. [Электронный ресурс]. URL: <https://razilov-code.ru/2018/11/02/binarytree-on-csharp-and-java/> (Дата обращения: 18.12.2021).
3. Бинарные поисковые деревья. [Электронный ресурс]. URL: <https://acm.bsu.by/wiki/Бинарные_поисковые_деревья> (Дата обращения: 19.12.2021).

# Приложение

Листинг BinaryTree.cs

using System;

using System.Collections.Generic;

namespace Trees

{

// Структура бинарного дерева

public class BinaryTree<T> where T : IComparable<T>

{

private BinaryTree<T> parent, left, right;

private T key;

private List<T> listForPrint = new List<T>();

public BinaryTree(T key, BinaryTree<T> parent)

{

this.key = key;

this.parent = parent;

}

// Добавление элемента

public void add(T key)

{

if (val.CompareTo(this.key) < 0) {

if (this.left == null) {

this.left = new BinaryTree<T>(key, this);

}

else if (this.left != null)

this.left.add(key);

}

else {

if (this.right == null) {

this.right = new BinaryTree<T>(key, this);

}

else if (this.right != null)

this.right.add(key);

}

}

// Поиск элемента

private BinaryTree<T> \_search(BinaryTree<T> tree, T key)

{

if (tree == null) return null;

switch (key.CompareTo(tree.val)) {

case 1: return \_search(tree.right, key);

case -1: return \_search(tree.left, key);

case 0: return tree;

default: return null;

}

}

public BinaryTree<T> search(T val)

{

return \_search(this, val);

}

// Удаление элемента

public bool remove(T val)

{

//Проверяем, существует ли данный узел

BinaryTree<T> tree = search(val);

if (tree == null) {

//Если узла не существует, вернем false

return false;

}

BinaryTree<T> curTree;

//Если удаляем корень

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;

return true;

}

//Удаление листьев

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;

}

//Удаление узла, имеющего левое поддерево, но не имеющее правого поддерева

if (tree.left != null && tree.right == null) {

//Меняем родителя

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;

}

//Удаление узла, имеющего правое поддерево, но не имеющее левого поддерева

if (tree.left == null && tree.right != null) {

//Меняем родителя

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;

}

//Удаляем узел, имеющий поддеревья с обеих сторон

if (tree.right != null && tree.left != null) {

curTree = tree.right;

while (curTree.left != null) {

curTree = curTree.left;

}

//Если самый левый элемент является первым потомком

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;

}

//Если самый левый элемент НЕ является первым потомком

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(BinaryTree<T> node)

{

if (node == null) return;

\_print(node.left);

listForPrint.Add(node.val);

Console.Write(node + " ");

if (node.right != null)

\_print(node.right);

}

public void print()

{

listForPrint.Clear();

\_print(this);

Console.WriteLine();

}

public override string ToString()

{

return val.ToString();

}

}

}

Листинг Program.cs

using System;

using System.Collections.Generic;

using System.Diagnostics;

namespace Trees

{

class Program

{

public static void Main(string[] args)

{

Random rList = new Random(47);

Random rTree = new Random(47);

int maxVal = 100000;

int n = 100000;

List<int> list = new List<int>();

list.Add(rList.Next(maxVal));

BinaryTree<int> tree = new BinaryTree<int>(maxVal, null);

int[] nabor = new int[90000];

for (int i = 0; i < n; i++)

{

int val = rList.Next(maxVal);

if (!list.Contains(val))

{

list.Add(val);

tree.add(val);

}

}

Console.WriteLine("Data loaded.");

Console.ReadKey();

}

}

}