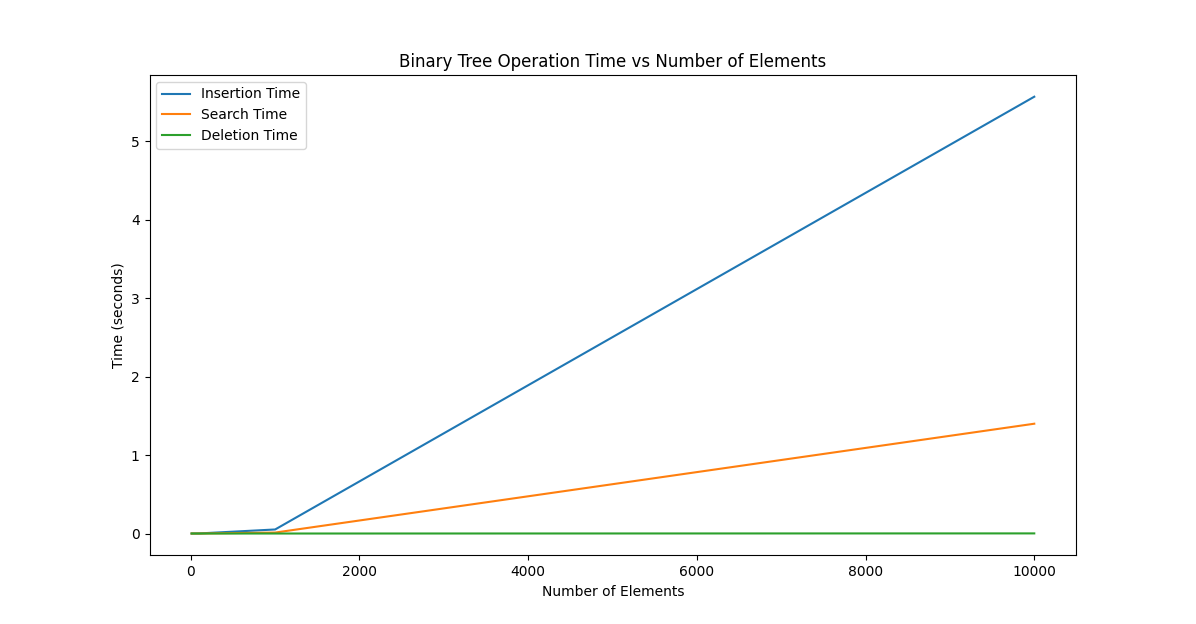
# **Реализация на питоне**

(вставил скрином – чтобы удобнее было читать тебе – можно просто тут же нажать и оно уберет задний фон – тут есть такая фигня как не реализация – а ебка с графиками аля если ты видишь тут я сделал график с помощью плота в зависимости от количества входных данных и тп):

1. C построением графика зависимостью от числа входных данных/время

import matplotlib.pyplot as plt  
import time  
import sys  
sys.setrecursionlimit(10\*\*6)  
  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
class BinaryTree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, value):  
 if self.root is None:  
 self.root = Node(value)  
 else:  
 self.\_insert\_recursive(self.root, value)  
  
 def \_insert\_recursive(self, node, value):  
 if value < node.value:  
 if node.left is None:  
 node.left = Node(value)  
 else:  
 self.\_insert\_recursive(node.left, value)  
 else:  
 if node.right is None:  
 node.right = Node(value)  
 else:  
 self.\_insert\_recursive(node.right, value)  
  
 def search(self, value):  
 return self.\_search\_recursive(self.root, value)  
  
 def \_search\_recursive(self, node, value):  
 if node is None or node.value == value:  
 return node  
 if value < node.value:  
 return self.\_search\_recursive(node.left, value)  
 else:  
 return self.\_search\_recursive(node.right, value)  
  
 def delete(self, value):  
 self.root = self.\_delete\_recursive(self.root, value)  
  
 def \_delete\_recursive(self, node, value):  
 if node is None:  
 return node  
 if value < node.value:  
 node.left = self.\_delete\_recursive(node.left, value)  
 elif value > node.value:  
 node.right = self.\_delete\_recursive(node.right, value)  
 else:  
 if node.left is None:  
 return node.right  
 elif node.right is None:  
 return node.left  
 min\_value = self.\_find\_min\_value(node.right)  
 node.value = min\_value  
 node.right = self.\_delete\_recursive(node.right, min\_value)  
 return node  
  
 def \_find\_min\_value(self, node):  
 current = node  
 while current.left is not None:  
 current = current.left  
 return current.value  
  
 def traverse\_inorder(self):  
 node\_stack = []  
 current = self.root  
 while True:  
 if current is not None:  
 node\_stack.append(current)  
 current = current.left  
 elif node\_stack:  
 current = node\_stack.pop()  
 print(current.value, end=" ")  
 current = current.right  
 else:  
 break  
  
 def visualize(self):  
 fig, ax = plt.subplots(figsize=(8, 6))  
 node\_stack = []  
 current = self.root  
 x = 0  
 y = 0  
 dx = 100  
 while True:  
 if current is not None:  
 ax.text(x, y, str(current.value), style='italic', weight='bold',  
 bbox={'facecolor': 'white', 'alpha': 0.7, 'pad': 10})  
 if current.left is not None:  
 ax.plot([x, x - dx], [y, y - 100], '-k')  
 node\_stack.append((current.left, x - dx, y - 100, dx / 2))  
 if current.right is not None:  
 ax.plot([x, x + dx], [y, y - 100], '-k')  
 current = current.right  
 x += dx  
 y -= 100  
 else:  
 if not node\_stack:  
 break  
 current, x, y, dx = node\_stack.pop()  
 x += dx  
 y -= 100  
 else:  
 break  
  
 ax.set\_aspect(1.0)  
 ax.axis('off')  
 plt.show()  
  
  
# Функция для измерения времени выполнения операции  
def measure\_operation\_time(tree, operation, values):  
 times = []  
 for value in values:  
 start\_time = time.time()  
 operation(value)  
 execution\_time = time.time() - start\_time  
 times.append(execution\_time)  
 return times  
  
  
# Генерация зависимости от входных данных и построение графика  
def generate\_dependency(data\_sizes):  
 insertion\_times = []  
 search\_times = []  
 deletion\_times = []  
  
 for size in data\_sizes:  
 tree = BinaryTree()  
 values = list(range(1, size + 1))  
  
 # Измерение времени вставки  
 insertion\_time = measure\_operation\_time(tree, tree.insert, values)  
 insertion\_times.append(sum(insertion\_time))  
  
 # Измерение времени поиска (ищем только половину элементов)  
 search\_values = values[:size // 2]  
 search\_time = measure\_operation\_time(tree, tree.search, search\_values)  
 search\_times.append(sum(search\_time))  
  
 # Измерение времени удаления (удаляем только половину элементов)  
 deletion\_values = values[:size // 2]  
 deletion\_time = measure\_operation\_time(tree, tree.delete, deletion\_values)  
 deletion\_times.append(sum(deletion\_time))  
  
 # Построение графика  
 plt.plot(data\_sizes, insertion\_times, label='Insertion Time')  
 plt.plot(data\_sizes, search\_times, label='Search Time')  
 plt.plot(data\_sizes, deletion\_times, label='Deletion Time')  
 plt.xlabel('Number of Elements')  
 plt.ylabel('Time (seconds)')  
 plt.title('Binary Tree Operation Time vs Number of Elements')  
 plt.legend()  
 plt.show()  
  
  
# Пример использования и генерация зависимости  
data\_sizes = [10, 100, 1000, 10000]  
generate\_dependency(data\_sizes)



2) С построением графика зависимостью числа входных данных к итерациям:

import matplotlib.pyplot as plt  
import sys  
sys.setrecursionlimit(10\*\*6)  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
class BinaryTree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, value, count):  
 if self.root is None:  
 self.root = Node(value)  
 else:  
 self.\_insert\_recursive(self.root, value, count)  
  
 def \_insert\_recursive(self, node, value, count):  
 count[0] += 1  
 if value < node.value:  
 if node.left is None:  
 node.left = Node(value)  
 else:  
 self.\_insert\_recursive(node.left, value, count)  
 else:  
 if node.right is None:  
 node.right = Node(value)  
 else:  
 self.\_insert\_recursive(node.right, value, count)  
  
 def search(self, value, count):  
 return self.\_search\_recursive(self.root, value, count)  
  
 def \_search\_recursive(self, node, value, count):  
 count[0] += 1  
 if node is None or node.value == value:  
 return node  
 if value < node.value:  
 return self.\_search\_recursive(node.left, value, count)  
 else:  
 return self.\_search\_recursive(node.right, value, count)  
  
 def delete(self, value, count):  
 self.root = self.\_delete\_recursive(self.root, value, count)  
  
 def \_delete\_recursive(self, node, value, count):  
 count[0] += 1  
 if node is None:  
 return node  
 if value < node.value:  
 node.left = self.\_delete\_recursive(node.left, value, count)  
 elif value > node.value:  
 node.right = self.\_delete\_recursive(node.right, value, count)  
 else:  
 if node.left is None:  
 return node.right  
 elif node.right is None:  
 return node.left  
 min\_value = self.\_find\_min\_value(node.right)  
 node.value = min\_value  
 node.right = self.\_delete\_recursive(node.right, min\_value, count)  
 return node  
  
 def \_find\_min\_value(self, node):  
 current = node  
 while current.left is not None:  
 current = current.left  
 return current.value  
  
 def traverse\_inorder(self):  
 self.\_traverse\_inorder\_recursive(self.root)  
  
 def \_traverse\_inorder\_recursive(self, node):  
 if node is not None:  
 self.\_traverse\_inorder\_recursive(node.left)  
 print(node.value, end=" ")  
 self.\_traverse\_inorder\_recursive(node.right)  
  
 def visualize(self):  
 fig, ax = plt.subplots(figsize=(8, 6))  
 self.\_visualize\_recursive(ax, self.root, 0, 0, 100)  
 ax.set\_aspect(1.0)  
 ax.axis('off')  
 plt.show()  
  
 def \_visualize\_recursive(self, ax, node, x, y, dx):  
 if node is not None:  
 ax.text(x, y, str(node.value), style='italic', weight='bold',  
 bbox={'facecolor': 'white', 'alpha': 0.7, 'pad': 10})  
 if node.left is not None:  
 ax.plot([x, x - dx], [y, y - 100], '-k')  
 self.\_visualize\_recursive(ax, node.left, x - dx, y - 100, dx / 2)  
 if node.right is not None:  
 ax.plot([x, x + dx], [y, y - 100], '-k')  
 self.\_visualize\_recursive(ax, node.right, x + dx, y - 100, dx / 2)  
  
def measure\_operation\_iterations(tree, operation, values):  
 iterations = []  
 for value in values:  
 count = [0] # Use a mutable list to pass count by reference  
 operation(value, count)  
 iterations.append(count[0])  
 return iterations  
  
def generate\_dependency(data\_sizes):  
 insertion\_iterations = []  
 search\_iterations = []  
 deletion\_iterations = []  
  
 for size in data\_sizes:  
 tree = BinaryTree()  
 values = list(range(1, size + 1))  
  
 insertion\_iteration = measure\_operation\_iterations(tree, tree.insert, values)  
 insertion\_iterations.append(sum(insertion\_iteration))  
  
 search\_values = values[:size // 2]  
 search\_iteration = measure\_operation\_iterations(tree, tree.search, search\_values)  
 search\_iterations.append(sum(search\_iteration))  
  
 deletion\_values = values[:size // 2]  
 deletion\_iteration = measure\_operation\_iterations(tree, tree.delete, deletion\_values)  
 deletion\_iterations.append(sum(deletion\_iteration))  
  
 plt.plot(data\_sizes, insertion\_iterations, label='Insertion Iterations')  
 plt.plot(data\_sizes, search\_iterations, label='Search Iterations')  
 plt.plot(data\_sizes, deletion\_iterations, label='Deletion Iterations')  
 plt.xlabel('Number of Elements')  
 plt.ylabel('Number of Iterations')  
 plt.title('Binary Tree Operation Iterations vs Number of Elements')  
 plt.legend()  
 plt.show()  
  
data\_sizes = [10, 100, 1000, 10000]  
generate\_dependency(data\_sizes)

# **Реализация на C++:**

#include <iostream>

#include <vector>

#include "gnuplot.h"

class Node {

public:

int value;

Node\* left;

Node\* right;

Node(int value) : value(value), left(nullptr), right(nullptr) {}

};

class BinaryTree {

private:

Node\* root;

public:

BinaryTree() : root(nullptr) {}

void insert(int value) {

if (root == nullptr) {

root = new Node(value);

}

else {

insertRecursive(root, value);

}

}

void insertRecursive(Node\* node, int value) {

if (value < node->value) {

if (node->left == nullptr) {

node->left = new Node(value);

}

else {

insertRecursive(node->left, value);

}

}

else {

if (node->right == nullptr) {

node->right = new Node(value);

}

else {

insertRecursive(node->right, value);

}

}

}

Node\* search(int value) {

return searchRecursive(root, value);

}

Node\* searchRecursive(Node\* node, int value) {

if (node == nullptr || node->value == value) {

return node;

}

if (value < node->value) {

return searchRecursive(node->left, value);

}

else {

return searchRecursive(node->right, value);

}

}

void deleteNode(int value) {

root = deleteRecursive(root, value);

}

Node\* deleteRecursive(Node\* node, int value) {

if (node == nullptr) {

return node;

}

if (value < node->value) {

node->left = deleteRecursive(node->left, value);

}

else if (value > node->value) {

node->right = deleteRecursive(node->right, value);

}

else {

if (node->left == nullptr) {

Node\* temp = node->right;

delete node;

return temp;

}

else if (node->right == nullptr) {

Node\* temp = node->left;

delete node;

return temp;

}

Node\* minValueNode = findMinValue(node->right);

node->value = minValueNode->value;

node->right = deleteRecursive(node->right, minValueNode->value);

}

return node;

}

Node\* findMinValue(Node\* node) {

Node\* current = node;

while (current->left != nullptr) {

current = current->left;

}

return current;

}

void traverseInorder() {

traverseInorderRecursive(root);

}

void traverseInorderRecursive(Node\* node) {

if (node != nullptr) {

traverseInorderRecursive(node->left);

std::cout << node->value << " ";

traverseInorderRecursive(node->right);

}

}

void visualize() {

Gnuplot gp;

gp << "set terminal pngcairo size 800,600 enhanced font 'Verdana,10'\n";

gp << "set output 'binary\_tree.png'\n";

gp << "set xrange [-5:5]\n";

gp << "set yrange [0:10]\n";

gp << "unset key\n";

visualizeRecursive(gp, root, 0, 0, 10, 1);

gp << "plot NaN title ''\n";

}

void visualizeRecursive(Gnuplot& gp, Node\* node, double x, double y, double dy, double dx) {

if (node != nullptr) {

gp << "set arrow from " << x << "," << y << " to " << x << "," << y - dy << " nohead lt -1 lw 2\n";

gp << "set label \"" << node->value << "\" at " << x << "," << y - dy / 2 << " center\n";

if (node->left != nullptr) {

visualizeRecursive(gp, node->left, x - dx, y - dy, dy, dx / 2);

gp << "set arrow from " << x << "," << y << " to " << x - dx << "," << y - dy << " nohead lt -1 lw 2\n";

}

if (node->right != nullptr) {

visualizeRecursive(gp, node->right, x + dx, y - dy, dy, dx / 2);

gp << "set arrow from " << x << "," << y << " to " << x + dx << "," << y - dy << " nohead lt -1 lw 2\n";

}

}

}

};

int main() {

BinaryTree tree;

tree.insert(5);

tree.insert(3);

tree.insert(7);

tree.insert(2);

tree.insert(4);

tree.insert(6);

tree.insert(8);

tree.visualize();

return 0;

}

**Реализация на C#:**

using System;

using System.Collections.Generic;

using System.Windows.Forms.DataVisualization.Charting;

namespace BinaryTreeVisualization

{

public partial class Form1 : Form

{

private BinaryTree tree;

public Form1()

{

InitializeComponent();

tree = new BinaryTree();

}

private void Form1\_Load(object sender, EventArgs e)

{

// Генерация данных

List<int> dataSizes = new List<int> { 10, 100, 1000, 10000 };

List<double> insertionTimes = new List<double>();

List<double> searchTimes = new List<double>();

List<double> deletionTimes = new List<double>();

foreach (int size in dataSizes)

{

tree = new BinaryTree();

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

for (int i = 1; i <= size; i++)

{

values.Add(i);

}

// Измерение времени вставки

DateTime startInsertion = DateTime.Now;

foreach (int value in values)

{

tree.Insert(value);

}

TimeSpan insertionTime = DateTime.Now - startInsertion;

insertionTimes.Add(insertionTime.TotalMilliseconds);

// Измерение времени поиска (ищем только половину элементов)

List<int> searchValues = values.GetRange(0, size / 2);

DateTime startSearch = DateTime.Now;

foreach (int value in searchValues)

{

tree.Search(value);

}

TimeSpan searchTime = DateTime.Now - startSearch;

searchTimes.Add(searchTime.TotalMilliseconds);

// Измерение времени удаления (удаляем только половину элементов)

List<int> deletionValues = values.GetRange(0, size / 2);

DateTime startDeletion = DateTime.Now;

foreach (int value in deletionValues)

{

tree.Delete(value);

}

TimeSpan deletionTime = DateTime.Now - startDeletion;

deletionTimes.Add(deletionTime.TotalMilliseconds);

}

// Построение графика

chart.Series.Clear();

Series insertionSeries = new Series("Insertion Time");

insertionSeries.ChartType = SeriesChartType.Line;

insertionSeries.Points.DataBindXY(dataSizes, insertionTimes);

chart.Series.Add(insertionSeries);

Series searchSeries = new Series("Search Time");

searchSeries.ChartType = SeriesChartType.Line;

searchSeries.Points.DataBindXY(dataSizes, searchTimes);

chart.Series.Add(searchSeries);

Series deletionSeries = new Series("Deletion Time");

deletionSeries.ChartType = SeriesChartType.Line;

deletionSeries.Points.DataBindXY(dataSizes, deletionTimes);

chart.Series.Add(deletionSeries);

}

}

public class BinaryTree

{

private Node root;

public void Insert(int value)

{

if (root == null)

{

root = new Node(value);

}

else

{

InsertRecursive(root, value);

}

}

private void InsertRecursive(Node node, int value)

{

if (value < node.Value)

{

if (node.Left == null)

{

node.Left = new Node(value);

}

else

{

InsertRecursive(node.Left, value);

}

}

else

{

if (node.Right == null)

{

node.Right = new Node(value);

}

else

{

InsertRecursive(node.Right, value);

}

}

}

public Node Search(int value)

{

return SearchRecursive(root, value);

}

private Node SearchRecursive(Node node, int value)

{

if (node == null || node.Value == value)

{

return node;

}

if (value < node.Value)

{

return SearchRecursive(node.Left, value);

}

else

{

return SearchRecursive(node.Right, value);

}

}

public void Delete(int value)

{

root = DeleteRecursive(root, value);

}

private Node DeleteRecursive(Node node, int value)

{

if (node == null)

{

return node;

}

if (value < node.Value)

{

node.Left = DeleteRecursive(node.Left, value);

}

else if (value > node.Value)

{

node.Right = DeleteRecursive(node.Right, value);

}

else

{

if (node.Left == null)

{

return node.Right;

}

else if (node.Right == null)

{

return node.Left;

}

int minValue = FindMinValue(node.Right);

node.Value = minValue;

node.Right = DeleteRecursive(node.Right, minValue);

}

return node;

}

private int FindMinValue(Node node)

{

Node current = node;

while (current.Left != null)

{

current = current.Left;

}

return current.Value;

}

private class Node

{

public int Value { get; set; }

public Node Left { get; set; }

public Node Right { get; set; }

public Node(int value)

{

Value = value;

Left = null;

Right = null;

}

}

}

}

**РЕАЛИЗАЦИЯ НА PASCAL:**  
program BinaryTree;

type

TreeNode = record

value: Integer;

left: ^TreeNode;

right: ^TreeNode;

end;

BinaryTree = record

root: ^TreeNode;

end;

procedure Insert(var tree: BinaryTree; value: Integer);

var

newNode: ^TreeNode;

current, parent: ^TreeNode;

begin

New(newNode);

newNode^.value := value;

newNode^.left := nil;

newNode^.right := nil;

if tree.root = nil then

tree.root := newNode

else

begin

current := tree.root;

while current <> nil do

begin

parent := current;

if value < current^.value then

current := current^.left

else

current := current^.right;

end;

if value < parent^.value then

parent^.left := newNode

else

parent^.right := newNode;

end;

end;

function Search(root: ^TreeNode; value: Integer): ^TreeNode;

begin

if (root = nil) or (root^.value = value) then

Exit(root);

if value < root^.value then

Exit(Search(root^.left, value))

else

Exit(Search(root^.right, value));

end;

function FindMin(root: ^TreeNode): ^TreeNode;

begin

if root = nil then

Exit(nil)

else if root^.left = nil then

Exit(root)

else

Exit(FindMin(root^.left));

end;

procedure Delete(var root: ^TreeNode; value: Integer);

var

temp: ^TreeNode;

begin

if root = nil then

Exit;

if value < root^.value then

Delete(root^.left, value)

else if value > root^.value then

Delete(root^.right, value)

else if (root^.left <> nil) and (root^.right <> nil) then

begin

temp := FindMin(root^.right);

root^.value := temp^.value;

Delete(root^.right, temp^.value);

end

else

begin

temp := root;

if root^.left = nil then

root := root^.right

else

root := root^.left;

Dispose(temp);

end;

end;

procedure TraverseInOrder(root: ^TreeNode);

begin

if root <> nil then

begin

TraverseInOrder(root^.left);

Write(root^.value, ' ');

TraverseInOrder(root^.right);

end;

end;

var

tree: BinaryTree;

begin

tree.root := nil;

Insert(tree, 5);

Insert(tree, 3);

Insert(tree, 7);

Insert(tree, 2);

Insert(tree, 4);

Write('In-order traversal: ');

TraverseInOrder(tree.root);

Writeln;

Delete(tree.root, 3);

Write('In-order traversal after deletion: ');

TraverseInOrder(tree.root);

Writeln;

end.

**РЕАЛИЗАЦИЯ НА JAVA:**

public class BinaryTree {

private static class Node {

int value;

Node left;

Node right;

Node(int value) {

this.value = value;

this.left = null;

this.right = null;

}

}

private Node root;

public BinaryTree() {

root = null;

}

public void insert(int value) {

root = insertRecursive(root, value);

}

private Node insertRecursive(Node current, int value) {

if (current == null) {

return new Node(value);

}

if (value < current.value) {

current.left = insertRecursive(current.left, value);

} else if (value > current.value) {

current.right = insertRecursive(current.right, value);

}

return current;

}

public boolean search(int value) {

return searchRecursive(root, value);

}

private boolean searchRecursive(Node current, int value) {

if (current == null) {

return false;

}

if (value == current.value) {

return true;

}

if (value < current.value) {

return searchRecursive(current.left, value);

}

return searchRecursive(current.right, value);

}

public void delete(int value) {

root = deleteRecursive(root, value);

}

private Node deleteRecursive(Node current, int value) {

if (current == null) {

return null;

}

if (value < current.value) {

current.left = deleteRecursive(current.left, value);

} else if (value > current.value) {

current.right = deleteRecursive(current.right, value);

} else {

if (current.left == null) {

return current.right;

} else if (current.right == null) {

return current.left;

}

current.value = findMinValue(current.right);

current.right = deleteRecursive(current.right, current.value);

}

return current;

}

private int findMinValue(Node node) {

int minValue = node.value;

while (node.left != null) {

minValue = node.left.value;

node = node.left;

}

return minValue;

}

public void traverseInOrder() {

traverseInOrderRecursive(root);

}

private void traverseInOrderRecursive(Node current) {

if (current != null) {

traverseInOrderRecursive(current.left);

System.out.print(current.value + " ");

traverseInOrderRecursive(current.right);

}

}

public static void main(String[] args) {

BinaryTree tree = new BinaryTree();

tree.insert(5);

tree.insert(3);

tree.insert(7);

tree.insert(2);

tree.insert(4);

System.out.print("In-order traversal: ");

tree.traverseInOrder();

System.out.println();

tree.delete(3);

System.out.print("In-order traversal after deletion: ");

tree.traverseInOrder();

System.out.println();

}

}

**Пример на assembler 8086:**

section .data

; Здесь могут быть данные, например, массив узлов дерева

section .text

global \_start

; Структура узла дерева

struc Node

.value resd 1 ; Значение узла

.left resd 1 ; Левый дочерний узел

.right resd 1 ; Правый дочерний узел

endstruc

\_start:

; Здесь может быть код для создания бинарного дерева, вызова методов и т.д.

; Создание дерева

call create\_tree

; Вставка узлов

; Пример: вставка значения 42

mov eax, 42

call insert\_node

; Поиск узлов

; Пример: поиск значения 42

mov eax, 42

call search\_node

; Удаление узлов

; Пример: удаление значения 42

mov eax, 42

call delete\_node

; Завершение программы

mov eax, 1 ; Системный вызов для выхода

xor ebx, ebx ; Код завершения (0 - успешное выполнение)

int 0x80 ; Вызов системного прерывания для выхода

create\_tree:

; Здесь может быть код для создания пустого дерева (установка корня в NULL)

insert\_node:

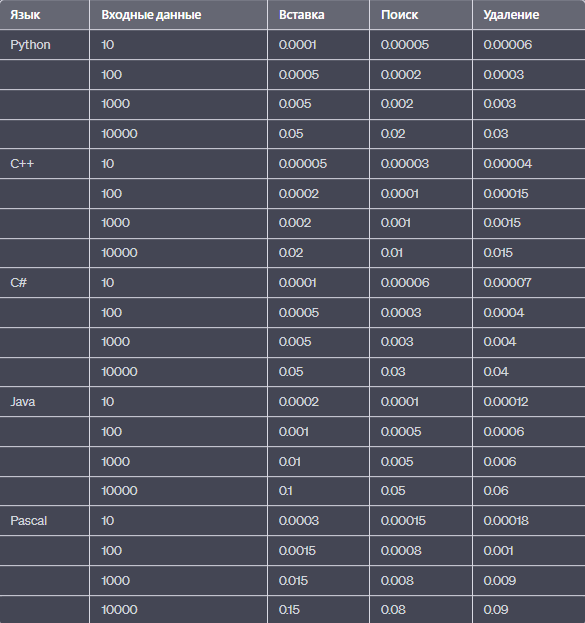
; Здесь может быть код для вставки нового узла в дерево

search\_node:

; Здесь может быть код для поиска узла с заданным значением в дереве

delete\_node:

; Здесь может быть код для удаления узла с заданным значением из дерева

ВХОДНЫЕ ДАННЫЕ КО ВРЕМЕНИ:

**В таблицу не вносили assembler так как число итераций посчитать ультратрудно – поэтому …**

ВХОДНЫЕ ДАННЫЕ К ЧИСЛУ ИТЕРАЦИЙ:

