Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Кафедра информационных систем и технологий

**Контрольная работа №1**

по дисциплине

«Алгоритмы и структуры данных»

|  |  |
| --- | --- |
|  | Выполнил студент группы 181073  Шерстнёв Артур Александрович |
|  | Проверил ассистент кафедры ИСиТ Потоцкий Дмитрий Сергеевич |

Минск 2022

**Введение**

**Цель работы:**

1. Научиться строить и обходить различными способами бинарные деревья, а также выполнять на них операции с данными.

2. Научиться прошивать бинарные деревья и обходить их заданным способом.

3. Научиться строить сбалансированные АВЛ-деревья и выполнять на них заданные операции с данными.

**Задание:** ввести 10-15 целых чисел и построить из них с помощью указателей бинарное дерево поиска. Обойти его прямым, симметричным и обратным способами. Реализовать процедуры поиска, вставки и удаления элементов бинарного дерева поиска.

**Код программы:**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

struct Node {

Node\* Left = NULL;

Node\* Right = NULL;

Node\* Parent = NULL;

int Value = NULL;

};

/// Вставка нового значения

void SetValue(Node\*\* p\_rootNode, int value) {

// Получение левого или правого потомка

Node\* nextNode = value < (\*p\_rootNode)->Value ? (\*p\_rootNode)->Left : (\*p\_rootNode)->Right;

// Если потомок не пуст, перейти к его потомкам

if (nextNode != NULL) {

SetValue(&nextNode, value);

return;

}

// Если потомок пуст, создать его

if (value < (\*p\_rootNode)->Value) {

(\*p\_rootNode)->Left = new Node();

(\*p\_rootNode)->Left->Value = value;

(\*p\_rootNode)->Left->Parent = (\*p\_rootNode);

} else {

(\*p\_rootNode)->Right = new Node();

(\*p\_rootNode)->Right->Value = value;

(\*p\_rootNode)->Right->Parent = (\*p\_rootNode);

}

}

/// Поиск узла

Node\* FindNode(Node\*\* p\_rootNode, int value) {

if (value == (\*p\_rootNode)->Value) {

return (\*p\_rootNode);

}

// Если удаляемого значения в дереве нет

if ((\*p\_rootNode)->Left == NULL && (\*p\_rootNode)->Right == NULL) {

return NULL;

}

// Получение левого или правого потомка

Node\* nextNode = value < (\*p\_rootNode)->Value ? (\*p\_rootNode)->Left : (\*p\_rootNode)->Right;

// Если потомок не пуст, перейти к его потомкам

if (nextNode != NULL) {

return FindNode(&nextNode, value);

}

return NULL;

}

// Удаление значения

void DelValue(Node\*\* p\_rootNode, int value) {

// Если удаляется не текущий узел

if (value != (\*p\_rootNode)->Value) {

// Если удаляемого значения в дереве нет

if ((\*p\_rootNode)->Left == NULL && (\*p\_rootNode)->Right == NULL) {

return;

}

Node\* nextNode = value < (\*p\_rootNode)->Value ?

((\*p\_rootNode)->Left != NULL ? (\*p\_rootNode)->Left : (\*p\_rootNode)->Right) :

((\*p\_rootNode)->Right != NULL ? (\*p\_rootNode)->Right : (\*p\_rootNode)->Left);

DelValue(&nextNode, value);

return;

}

// Если узел - единственный в дереве

if ((\*p\_rootNode)->Right == NULL && (\*p\_rootNode)->Left == NULL && (\*p\_rootNode)->Parent == NULL) {

(\*p\_rootNode) = NULL;

return;

}

// Если узел - лист

if ((\*p\_rootNode)->Right == NULL && (\*p\_rootNode)->Left == NULL) {

// Отсоединение от предка

if ((\*p\_rootNode) == (\*p\_rootNode)->Parent->Left) {

(\*p\_rootNode)->Parent->Left = NULL;

} else {

(\*p\_rootNode)->Parent->Right = NULL;

}

return;

}

// Если узел - корень дерева и имеет только левого потомка

if ((\*p\_rootNode)->Right == NULL && (\*p\_rootNode)->Left != NULL && (\*p\_rootNode)->Parent == NULL) {

(\*p\_rootNode)->Value = (\*p\_rootNode)->Left->Value;

(\*p\_rootNode)->Left = (\*p\_rootNode)->Left->Left;

return;

}

// Если узел имеет только левого потомка

if ((\*p\_rootNode)->Right == NULL && (\*p\_rootNode)->Left != NULL) {

// Присоединение к предку левого потомка

(\*p\_rootNode)->Parent->Left = (\*p\_rootNode)->Left;

return;

}

// Поиск ближайшего большего значения

Node\* currentNode = (\*p\_rootNode)->Right;

while (currentNode->Left != NULL) {

currentNode = currentNode->Left;

}

// Замена узла

(\*p\_rootNode)->Value = currentNode->Value;

// Если новый узел - не лист

if (currentNode->Right != NULL) {

currentNode->Parent->Right = currentNode->Right;

return;

}

if (currentNode == currentNode->Parent->Left) {

currentNode->Parent->Left = NULL;

} else {

currentNode->Parent->Right = NULL;

}

}

/// Прямой обход дерева

void NLR(Node\* rootNode) {

if (rootNode == NULL) {

return;

}

// Вывод значения

cout << rootNode->Value << " ";

// Обход левой ветки

if (rootNode->Left != NULL) {

NLR(rootNode->Left);

}

// Обход правой ветки

if (rootNode->Right != NULL) {

NLR(rootNode->Right);

}

}

/// Обратный обход дерева

void LRN(Node\* rootNode) {

if (rootNode == NULL) {

return;

}

// Обход левой ветки

if (rootNode->Left != NULL) {

LRN(rootNode->Left);

}

// Обход правой ветки

if (rootNode->Right != NULL) {

LRN(rootNode->Right);

}

// Вывод значения

cout << rootNode->Value << " ";

}

/// Симметричный обход дерева

void LNR(Node\* rootNode) {

if (rootNode == NULL) {

return;

}

// Обход левой ветки

if (rootNode->Left != NULL) {

LNR(rootNode->Left);

}

// Вывод значения

cout << rootNode->Value << " ";

// Обход правой ветки

if (rootNode->Right != NULL) {

LNR(rootNode->Right);

}

}

int main() {

cout << "Enter the numbers and end with a space with a dot: " << endl;

// Ввод значений

vector<int> values;

string c\_value = " ";

while (true) {

cin >> c\_value;

if (c\_value == ".") {

break;

}

values.push\_back(stoi(c\_value));

}

// Проверка на пустоту списка

if (values.empty()) {

return 0;

}

// Заполнение дерева

Node\* rootNode = new Node();

rootNode->Value = values.front();

values.erase(values.begin());

Node\*\* p\_rootNode = &rootNode;

for (auto value : values) {

SetValue(p\_rootNode, value);

}

cout << "\nDirect tree traversal: " << endl;

NLR(rootNode);

cout << "\nReverse tree traversal: " << endl;

LRN(rootNode);

cout << "\nSymmetric tree traversal: " << endl;

LNR(rootNode);

int n;

cout << "\n\nEnter the value to delete: ";

cin >> n;

DelValue(p\_rootNode, n);

cout << "Direct traversal of the tree after deleting the value: " << endl;

NLR(rootNode);

cout << "\n\nEnter a value to search for: ";

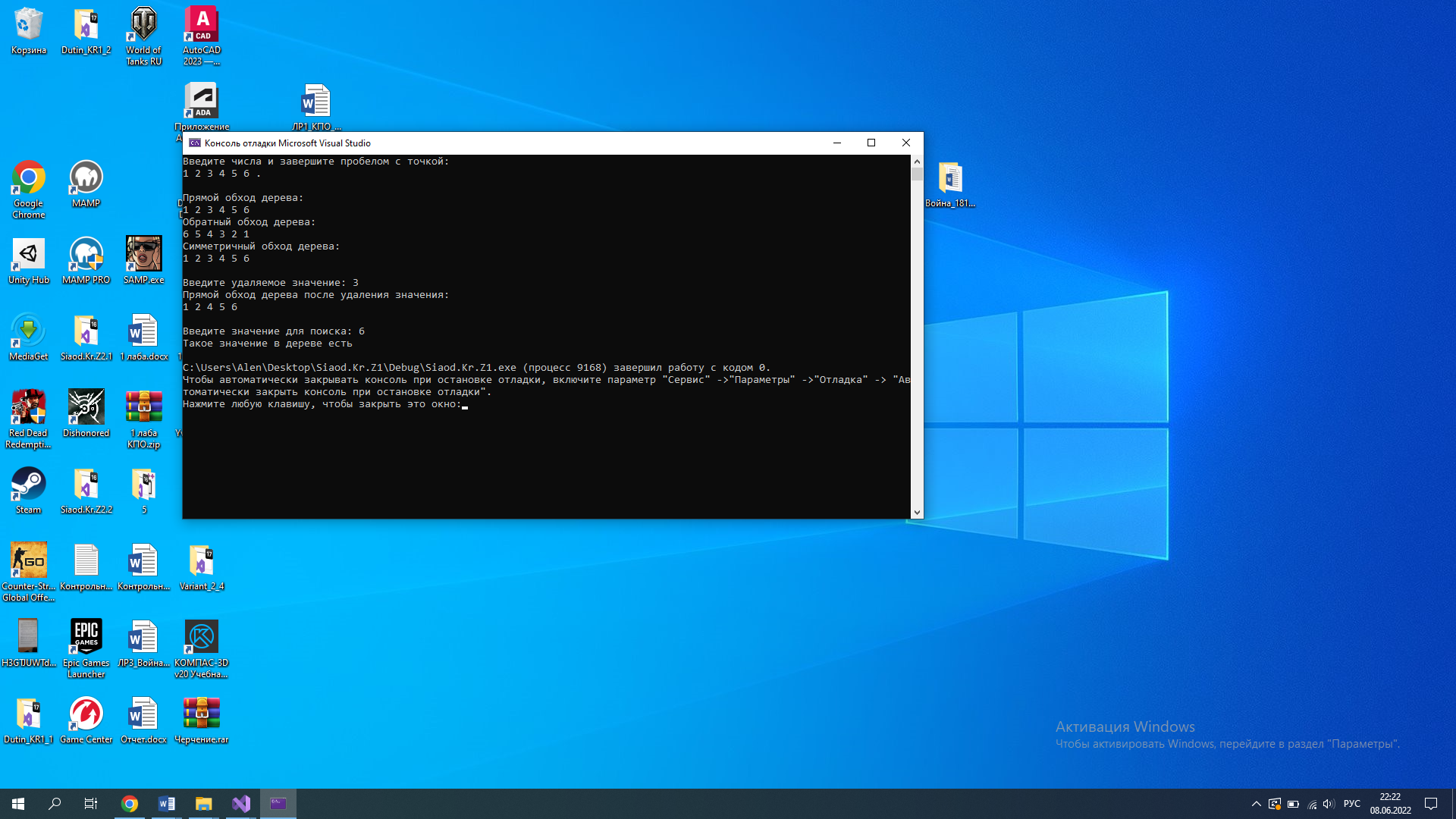
cin >> n;

cout << (FindNode(p\_rootNode, n) != NULL ? "There is such a value in the tree" : "There is no such value in the tree") << endl;

return 0;

}

**Результат:**



**Задание 2:** Ввести 10-15 целых чисел и построить из них бинарное дерево поиска. Выполнить симметричную прошивку бинарного дерева поиска. Обойти его согласно симметричному порядку следования элементов. Реализовать поиск и вставку элементов симметрично прошитого бинарного дерева.

**Код программы:**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

struct Node {

bool LIsThread = false;

Node\* Left = NULL;

bool RIsThread = false;

Node\* Right = NULL;

int Value = NULL;

};

void SetValue(Node\*\* p\_rootNode, int value)

{

// Получение левого или правого потомка

Node\* nextNode = value < (\*p\_rootNode)->Value && !(\*p\_rootNode)->LIsThread ? (\*p\_rootNode)->Left:

value >= (\*p\_rootNode)->Value && !(\*p\_rootNode)->RIsThread ? (\*p\_rootNode)->Right:

NULL;

// Если потомок не пуст, перейти к его потомкам

if (nextNode != NULL)

{

SetValue(&nextNode, value);

return;

}

if (value < (\*p\_rootNode)->Value)

{

Node\* newNode = new Node();

newNode->Value = value;

newNode->LIsThread = true;

newNode->Left = (\*p\_rootNode)->Left;

newNode->RIsThread = true;

newNode->Right = (\*p\_rootNode);

(\*p\_rootNode)->LIsThread = false;

(\*p\_rootNode)->Left = newNode;

}

else

{

Node\* newNode = new Node();

newNode->Value = value;

newNode->LIsThread = true;

newNode->Left = (\*p\_rootNode);

newNode->RIsThread = true;

newNode->Right = (\*p\_rootNode)->Right;

(\*p\_rootNode)->RIsThread = false;

(\*p\_rootNode)->Right = newNode;

}

}

/// Поиск узла

Node\* FindNode(Node\*\* p\_rootNode, int value)

{

if (value == (\*p\_rootNode)->Value)

return (\*p\_rootNode);

// Если удаляемого значения в дереве нет

if ((\*p\_rootNode)->Left == NULL && (\*p\_rootNode)->Right == NULL)

return NULL;

// Получение левого или правого потомка

Node\* nextNode = value < (\*p\_rootNode)->Value ? (\*p\_rootNode)->Left : (\*p\_rootNode)->Right;

// Если потомок не пуст, перейти к его потомкам

if (nextNode != NULL)

return FindNode(&nextNode, value);

return NULL;

}

/// Поиск узла (на основе прошивки)

Node\* FindThreatedNode(Node\* rootNode, int value)

{

// Начало обхода с корня дерева

Node\* currentNode = rootNode;

while (true)

{

// Нахождение самого левого узла в поддереве

while (currentNode->Left != NULL && !currentNode->LIsThread)

currentNode = currentNode->Left;

while (true)

{

// Возвращение найденного узла

if (currentNode->Value == value)

return currentNode;

// Если правая нить ведет к корню дерева и имеет значение больше, чем корень, то достигнут конец списка

if (currentNode->RIsThread && currentNode->Right == rootNode && currentNode->Value > currentNode->Right->Value)

return NULL;

// Если правая ссылка - нить, то перейти, иначе перейти и искать самый левый узел

if (currentNode->RIsThread)

currentNode = currentNode->Right;

else

{

currentNode = currentNode->Right;

break;

}

}

}

}

/// Симметричный обход дерева

void LNR(Node\* rootNode, vector<int>& values)

{

// Обход левой ветки

if (rootNode->Left != NULL && !rootNode->LIsThread)

LNR(rootNode->Left, values);

// Сохранение значения

values.push\_back(rootNode->Value);

// Обход правой ветки

if (rootNode->Right != NULL && !rootNode->RIsThread)

LNR(rootNode->Right, values);

}

/// Симметричный обход прошитого дерева

void ThreadedLNR(Node\* rootNode)

{

// Начало обхода с корня дерева

Node\* currentNode = rootNode;

while (true)

{

// Нахождение самого левого узла в поддереве

while (currentNode->Left != NULL && !currentNode->LIsThread)

currentNode = currentNode->Left;

while (true)

{

// Вывод узла

cout << currentNode->Value << " ";

// Если правая нить ведет к корню дерева и имеет значение больше, чем корень, то достигнут конец списка

if (currentNode->RIsThread && currentNode->Right == rootNode && currentNode->Value > currentNode->Right->Value)

return;

// Если правая ссылка - нить, то перейти, иначе перейти и искать самый левый узел

if (currentNode->RIsThread)

currentNode = currentNode->Right;

else

{

currentNode = currentNode->Right;

break;

}

}

}

}

/// Симметричная прошивка дерева

void LNRThread(Node\*\* p\_rootNode)

{

// Получение значений дерева при симметричном обходе

vector<int> values;

LNR((\*p\_rootNode), values);

// Проверка на пустоту списка

if (values.empty())

return;

// Начало со второго значения и до предпоследнего

for (int i = 1; i < values.size() - 2; i++)

{

// Получение указателя на узел с таким значением

Node\* currentNode = FindNode(p\_rootNode, values[i]);

// Если нет левого потомка, то указать нить на узел с предыдущим значением

if (currentNode->Left == NULL)

{

currentNode->LIsThread = true;

currentNode->Left = FindNode(p\_rootNode, values[i - 1]);

}

// Если нет правого потомка, то указать нить на узел со следующим значением

if (currentNode->Right == NULL)

{

currentNode->RIsThread = true;

currentNode->Right = FindNode(p\_rootNode, values[i + 1]);

}

}

// Связь первого узла с корнем дерева, и, при отсутствии правого потомка, указать нить на узел со следующим значением

Node\* firstNode = FindNode(p\_rootNode, values[0]);

firstNode->LIsThread = true;

firstNode->Left = (\*p\_rootNode);

if (firstNode->Right == NULL)

{

firstNode->RIsThread = true;

firstNode->Right = FindNode(p\_rootNode, values[1]);

}

// Связь последнего узла с корнем дерева, и, при отсутствии левого потомка, указать нить на узел с предыдущим значением

Node\* lastNode = FindNode(p\_rootNode, values[values.size() - 1]);

lastNode->RIsThread = true;

lastNode->Right = (\*p\_rootNode);

if (lastNode->Left == NULL)

{

lastNode->LIsThread = true;

lastNode->Left = FindNode(p\_rootNode, values[values.size() - 2]);

}

}

int main()

{

setlocale(LC\_ALL, "Russian");

cout << "Введите числа, в конце точка: " << endl;

// Ввод значений

vector<int> values;

string c\_value = " ";

while (true)

{

cin >> c\_value;

if (c\_value == ".")

break;

values.push\_back(stoi(c\_value));

}

// Проверка на пустоту списка

if (values.empty())

return 0;

// Заполнение дерева

Node\* rootNode = new Node();

rootNode->Value = values.front();

values.erase(values.begin());

Node\*\* p\_rootNode = &rootNode;

for (auto value : values)

{

SetValue(p\_rootNode, value);

}

LNRThread(p\_rootNode);

cout << "\nОбход прошитого дерева: " << endl;

ThreadedLNR(rootNode);

int n;

cout << "\n\nВведите вставку: ";

cin >> n;

SetValue(p\_rootNode, n);

cout << "Обход дерева после вставки: " << endl;

ThreadedLNR(rootNode);

cout << "\n\nВведите значение для поиска: ";

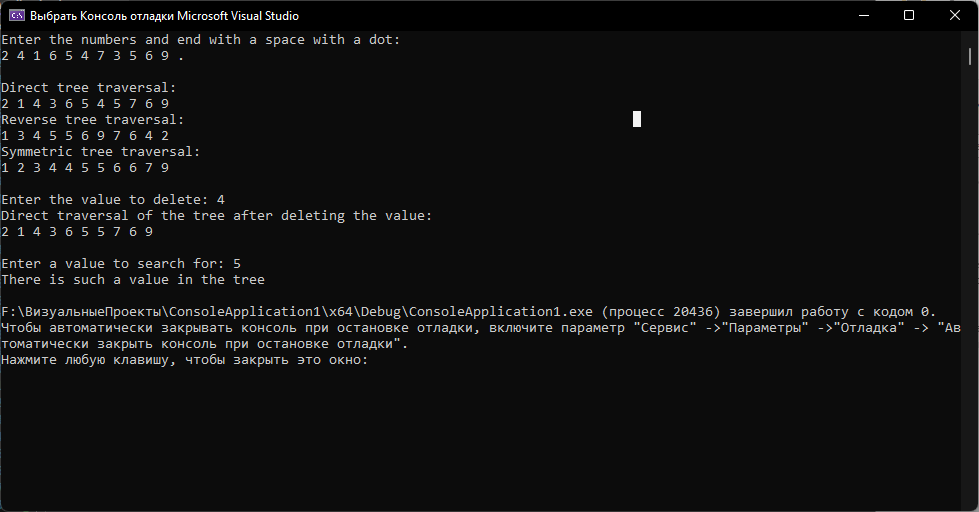
cin >> n;

cout << (FindThreatedNode(rootNode, n) != NULL ? "Присутствует" : "Отсутствует") << endl;

return 0;

}

**Результат:**



**Контрольные вопросы:**

1. Дайте определение абстрактному типу данных «дерево».

Дерево – это абстрактный тип данных для хранения информационных элементов имеющих нелинейные отношения. За исключением элемента, который находится во главе дерева, каждый элемент (вершина) имеет родителя (предка), а также ноль или более дочерних элементов (потомков).

2.  Дайте определение бинарному дереву и бинарному дереву поиска.

Бинарное (двоичное) дерево – это динамическая структура данных, представляющее собой дерево, в котором каждая вершина имеет не более двух потомков.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде.

3.  Назовите три основных способа обхода бинарных деревьев и укажите порядок следования вершин, соответствующий каждому из обходов.

Прямой обход: сначала обходится данная вершина, левое поддерево данной вершины, затем правое поддерево данной вершины.

Симметричный обход: сначала обходится левое поддерево данной вершины, затем данная вершина, затем правое поддерево данной вершины. Вершины при этом будут следовать в неубывающем (по ключам key) порядке.

Обратный обход: сначала обходится левое поддерево данной вершины, затем правое, затем данная вершина.

4.  Перечислите достоинства и недостатки прошивки бинарных деревьев.

Преимущества: обход без использования рекурсии или стека, быстрота, можно определить предшественника и преемника вершины.

Недостатки: включение новой вершины в дерево занимает больше времени, удаление и включение новых вершин могут потребовать прошивки дерева заново.

5.  Чем АВЛ-дерево отличается от идеально сбалансированного бинарного дерева?

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Идеально сбалансированное дерево – это дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более чем на 1.

Различие: в АВЛ-дереве количество вершин в левом и правом поддереве может отличаться более чем на 1.

**Заключение по работе:** в ходе данной лабораторной работы я закрепил знания в написании бинарных деревьев, научился обходить деревья различными способами и заданным способом. Научился строить сбалансированные АВЛ-деревья и выполнил над ними заданные операции.