Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Лабораторная работа №11

По дисциплине «Основы алгоритмизации и программирования»

На тему « **Бинарные деревья**»

Выполнил:

Студент 1 курса 10 группы

Мандрик Алексей Иванович

Преподаватель: асс. Андронова М.В.

Минск, 2024

Разработать программу работы с ***бинарным деревом поиска***, в которую включить основные функции манипуляции данными и функцию в соответствии со своим вариантом из таблицы, представленной ниже.

**Вариант 8**



**Код программы:**

#include <iostream>

using namespace std;

// Структура для представления узла бинарного дерева

struct TreeNode {

int key;

TreeNode\* left;

TreeNode\* right;

// Конструктор для узла дерева

TreeNode(int value) : key(value), left(nullptr), right(nullptr) {}

};

// Класс для работы с бинарным деревом поиска

class BinarySearchTree {

private:

TreeNode\* root;

// Вставка узла в дерево

TreeNode\* insert(TreeNode\* node, int key) {

// Если дерево пустое, создаем новый узел с заданным ключом

if (node == nullptr) {

return new TreeNode(key);

}

// Иначе рекурсивно ищем место для вставки ключа в левое или правое поддерево

if (key < node->key) {

node->left = insert(node->left, key);

}

else if (key > node->key) {

node->right = insert(node->right, key);

}

return node;

}

// Поиск узла с максимальной суммой

TreeNode\* findMaxSumNode(TreeNode\* node) {

if (node == nullptr) return nullptr;

TreeNode\* maxNode = node;

TreeNode\* rightMaxNode = findMaxSumNode(node->right);

if (rightMaxNode != nullptr && (rightMaxNode->key + node->key) > (maxNode->key + node->key)) {

maxNode = rightMaxNode;

}

return maxNode;

}

// Удаление узла из дерева

TreeNode\* deleteNode(TreeNode\* node, int key) {

if (node == nullptr) return nullptr;

if (key < node->key) {

node->left = deleteNode(node->left, key);

}

else if (key > node->key) {

node->right = deleteNode(node->right, key);

}

else {

if (node->left == nullptr) {

TreeNode\* temp = node->right;

delete node;

return temp;

}

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

TreeNode\* temp = node->left;

delete node;

return temp;

}

TreeNode\* temp = findMaxSumNode(node->left);

node->key = temp->key;

node->left = deleteNode(node->left, temp->key);

}

return node;

}

// Обход дерева в порядке возрастания

void inorderTraversal(TreeNode\* node) {

if (node != nullptr) {

inorderTraversal(node->left);

cout << node->key << " ";

inorderTraversal(node->right);

}

}

// Рекурсивная функция для очистки дерева

void clearTree(TreeNode\* node) {

if (node != nullptr) {

clearTree(node->left);

clearTree(node->right);

delete node;

}

}

public:

BinarySearchTree() : root(nullptr) {}

// Деструктор для очистки памяти

~BinarySearchTree() {

clearTree(root);

}

// Вставка значения в дерево

void insert(int key) {

root = insert(root, key);

}

// Удаление узла с максимальной суммой

void deleteMaxSumNode() {

if (root == nullptr) return;

root = deleteNode(root, findMaxSumNode(root)->key);

}

// Вывод дерева в порядке возрастания

void inorderTraversal() {

inorderTraversal(root);

cout << endl;

}

// Очистка дерева от узлов

void clear() {

clearTree(root);

root = nullptr;

}

};

int main() {

setlocale(LC\_ALL, "Rus");

BinarySearchTree bst;

int choice, key;

do {

cout << "Выберите операцию:\n";

cout << "1. Вставить узел\n";

cout << "2. Удалить вершину с максимальной суммой\n";

cout << "3. Вывести дерево в порядке возрастания\n";

cout << "4. Очистить дерево\n";

cout << "0. Выйти\n";

cout << "Ваш выбор: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Введите значение для вставки: ";

cin >> key;

bst.insert(key);

break;

case 2:

bst.deleteMaxSumNode();

cout << "Вершина с максимальной суммой удалена.\n";

break;

case 3:

cout << "Дерево в порядке возрастания:\n";

bst.inorderTraversal();

break;

case 4:

bst.clear();

cout << "Дерево очищено.\n";

break;

case 0:

cout << "Программа завершена.\n";

break;

default:

cout << "Некорректный выбор. Пожалуйста, попробуйте еще раз.\n";

break;

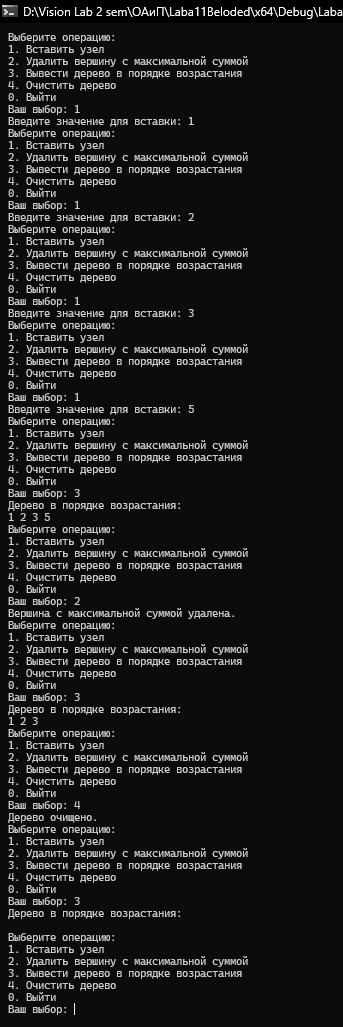
}

} while (choice != 0);

return 0;

}

}**Результат выполнения:**

****

**Дополнительные задания:**

**Вариант 7:**



**Код программы:**

#include <iostream>

using namespace std;

// Структура для представления узла в бинарном дереве поиска

struct TreeNode {

int key;

TreeNode\* left; // Левый потомок

TreeNode\* right; // Правый потомок

// Конструктор для создания нового узла с заданным ключом

TreeNode(int value) : key(value), left(nullptr), right(nullptr) {}

};

// Класс, представляющий бинарное дерево поиска

class BinarySearchTree {

private:

TreeNode\* root; // Корень дерева

// Приватная рекурсивная функция для вставки ключа в дерево

TreeNode\* insert(TreeNode\* node, int key) {

if (node == nullptr) {

return new TreeNode(key);

}

if (key < node->key) {

node->left = insert(node->left, key);

}

else if (key > node->key) {

node->right = insert(node->right, key);

}

return node;

}

// Приватная рекурсивная функция для обхода дерева в порядке возрастания

void inorderTraversal(TreeNode\* node) {

if (node != nullptr) {

inorderTraversal(node->left);

cout << node->key << " ";

inorderTraversal(node->right);

}

}

// Приватная рекурсивная функция для очистки дерева от узлов

void clearTree(TreeNode\* node) {

if (node != nullptr) {

clearTree(node->left);

clearTree(node->right);

delete node;

}

}

// Приватная рекурсивная функция для подсчета листьев

int countLeaves(TreeNode\* node) {

if (node == nullptr) return 0; // Условие выхода из рекурсии: если узел пустой, возвращаем 0

if (node->left == nullptr && node->right == nullptr) return 1; // Если узел является листом, возвращаем 1

// Рекурсивно вызываем функцию для левого и правого поддеревьев и суммируем результаты

return countLeaves(node->left) + countLeaves(node->right);

}

public:

// Конструктор для создания пустого дерева

BinarySearchTree() : root(nullptr) {}

// Деструктор для очистки памяти

~BinarySearchTree() {

clearTree(root);

}

// Публичная функция для вставки ключа в дерево

void insert(int key) {

root = insert(root, key);

}

// Публичная функция для вывода дерева в порядке возрастания

void inorderTraversal() {

inorderTraversal(root);

cout << endl;

}

// Публичная функция для очистки дерева от всех узлов

void clear() {

clearTree(root);

root = nullptr;

}

// Публичная функция для подсчета количества листьев в дереве

int countLeaves() {

return countLeaves(root);

}

};

// Основная функция

int main() {

setlocale(LC\_ALL, "Rus");

BinarySearchTree bst;

int choice, key;

// Основной цикл программы

do {

cout << "Выберите операцию:\n";

cout << "1. Вставить узел\n";

cout << "2. Вывести дерево в порядке возрастания\n";

cout << "3. Очистить дерево\n";

cout << "4. Посчитать количество листьев\n";

cout << "0. Выйти\n";

cout << "Ваш выбор: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Введите значение для вставки: ";

cin >> key;

bst.insert(key);

break;

case 2:

cout << "Дерево в порядке возрастания:\n";

bst.inorderTraversal();

break;

case 3:

bst.clear();

cout << "Дерево очищено.\n";

break;

case 4:

cout << "Количество листьев в дереве: " << bst.countLeaves() << endl;

break;

case 0:

cout << "Программа завершена.\n";

break;

default:

cout << "Некорректный выбор. Пожалуйста, попробуйте еще раз.\n";

break;

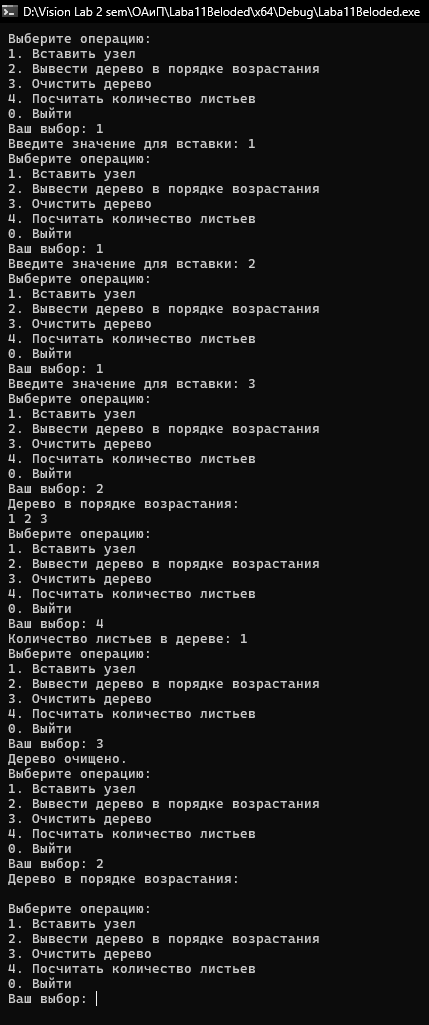
}

} while (choice != 0);

return 0;

}

**Результат выполнения:**



**Вариант 10:**

**Код программы:**

#include <iostream>

#include <windows.h>

using namespace std;

struct Tree //дерево

{

int key; //ключ

Tree\* Left, \* Right;

};

Tree\* makeTree(Tree\* Root); //Создание дерева

Tree\* list(int i); //Создание нового элемента

Tree\* insertElem(Tree\* Root, int key); //Добавление нового элемента

Tree\* delet(Tree\* Root, int key); //Удаление элемента по ключу

void view(Tree\* t, int level); //Вывод дерева

int count(Tree\* t); //Подсчет количества листьев

void delAll(Tree\* t); //Очистка дерева

int c = 0, rightLen = 0;; //количество слов и длина правой ветки соответ.

Tree\* Root = NULL; //указатель корня

int main()

{

setlocale(0, "Russian");

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

int key, choice;

for (;;)

{

cout << "1 - создание дерева\n";

cout << "2 - добавление элемента\n";

cout << "3 - удаление элемента\n";

cout << "4 - вывод дерева\n";

cout << "5 - очистка дерева\n";

cout << "6 - выход\n";

cout << "ваш выбор?\n";

cin >> choice;

cout << "\n";

switch (choice)

{

case 1: Root = makeTree(Root); break;

case 2: cout << "\nВведите ключ: "; cin >> key;

insertElem(Root, key); break;

case 3:

cout << "\nВведите удаляемый ключ: "; cin >> key;

Root = delet(Root, key); break;

case 4: if (Root->key >= 0)

{

cout << "Дерево повернуто на 90 град. влево" << endl;

view(Root, 0);

cout << "Сумма всех вершин = " << count(Root) << endl;

}

else cout << "Дерево пустое\n"; break;

case 5: delAll(Root); break;

case 6: exit(0);

}

}

return 0;

}

Tree\* makeTree(Tree\* Root) //Создание дерева

{

int key;

cout << "Конец ввода - отрицательное число\n\n";

if (Root == NULL) // если дерево не создано

{

cout << "Введите ключ корня: "; cin >> key;

Root = list(key); // установка указателя на корень

}

while (1) //добавление элементов

{

cout << "\nВведите ключ: "; cin >> key;

if (key < 0) break; //признак выхода (ключ < 0)

insertElem(Root, key);

}

return Root;

}

Tree\* list(int i) //Создание нового элемента

{

Tree\* t = new Tree[sizeof(Tree)];

t->key = i;

t->Left = t->Right = NULL;

return t;

}

Tree\* insertElem(Tree\* t, int key) //Добавление нового элемента

{

Tree\* Prev = t; // Prev - элемент перед текущим

int find = 0; // признак поиска

while (t && !find) {

Prev = t;

if (key == t->key)

find = 1; //ключи должны быть уникальны

else

if (key < t->key) t = t->Left;

else t = t->Right;

}

if (!find) //найдено место с адресом Prev

{

t = list(key); //создается новый узел

if (key < Prev->key) // и присоединяется либо

Prev->Left = t; //переход на левую ветвь,

else

Prev->Right = t; // либо на правую

}

return t;

}

Tree\* delet(Tree\* Root, int key) //Удаление элемента по ключу

{ // Del, Prev\_Del - удаляемый элемент и его предыдущий ;

// R, Prev\_R - элемент, на который заменяется удаленный, и его родитель;

Tree\* Del, \* Prev\_Del, \* R, \* Prev\_R;

Del = Root;

Prev\_Del = NULL;

while (Del != NULL && Del->key != key)//поиск элемента и его родителя

{

Prev\_Del = Del;

if (Del->key > key)

Del = Del->Left;

else

Del = Del->Right;

}

if (Del == NULL) // элемент не найден

{

puts("\nНет такого ключа");

return Root;

}

if (Del->Right == NULL) // поиск элемента R для замены

R = Del->Left;

else

if (Del->Left == NULL)

R = Del->Right;

else

{

Prev\_R = Del; //поиск самого правого элемента в левом поддереве

R = Del->Left;

while (R->Right != NULL)

{

Prev\_R = R;

R = R->Right;

}

if (Prev\_R == Del) // найден элемент для замены R и его родителя Prev\_R

R->Right = Del->Right;

else

{

R->Right = Del->Right;

Prev\_R->Right = R->Left;

R->Left = Prev\_R;

}

}

if (Del == Root) Root = R; //удаление корня и замена его на R

else

// поддерево R присоединяется к родителю удаляемого узла

if (Del->key < Prev\_Del->key)

Prev\_Del->Left = R; // на левую ветвь

else

Prev\_Del->Right = R; // на правую ветвь

int tmp = Del->key;

cout << "\nУдален элемент с ключом " << tmp << endl;

delete Del;

return Root;

}

int count(Tree\* t) //Подсчет количества слов

{

if (t) {

count(t->Right);

c += t->key;

count(t->Left);

}

return c;

}

void view(Tree\* t, int level) //Вывод дерева

{

if (t)

{

rightLen++;

view(t->Right, level + 1); //вывод правого поддерева

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

cout << " ";

int tm = t->key;

cout << tm << ' ' << endl;;

view(t->Left, level + 1); //вывод левого поддерева

}

}

void delAll(Tree\* t) //Очистка дерева

{

if (t != NULL)

{

delAll(t->Left);

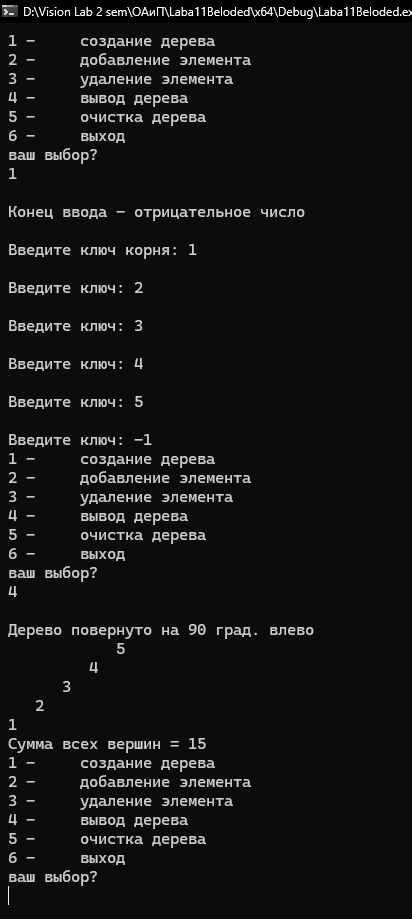
delAll(t->Right);

delete t;

}

}

**Результат выполнения:**



**Доп 12:**

**Код программы:**

#include <iostream>

using namespace std;

// Структура для представления узла в бинарном дереве поиска

struct TreeNode {

int key;

TreeNode\* left; // Левый потомок

TreeNode\* right; // Правый потомок

// Конструктор для создания нового узла с заданным ключом

TreeNode(int value) : key(value), left(nullptr), right(nullptr) {}

};

// Класс, представляющий бинарное дерево поиска

class BinarySearchTree {

private:

TreeNode\* root; // Корень дерева

// Приватная рекурсивная функция для вставки ключа в дерево

TreeNode\* insert(TreeNode\* node, int key) {

if (node == nullptr) {

return new TreeNode(key);

}

if (key < node->key) {

node->left = insert(node->left, key);

}

else if (key > node->key) {

node->right = insert(node->right, key);

}

return node;

}

// Приватная рекурсивная функция для обхода дерева в порядке возрастания

void inorderTraversal(TreeNode\* node) {

if (node != nullptr) {

inorderTraversal(node->left);

cout << node->key << " ";

inorderTraversal(node->right);

}

}

// Приватная рекурсивная функция для очистки дерева от узлов

void clearTree(TreeNode\* node) {

if (node != nullptr) {

clearTree(node->left);

clearTree(node->right);

delete node;

}

}

// Приватная рекурсивная функция для подсчета листьев

int countLeaves(TreeNode\* node) {

if (node == nullptr) return 0; // Условие выхода из рекурсии: если узел пустой, возвращаем 0

if (node->left == nullptr && node->right == nullptr) return 1; // Если узел является листом, возвращаем 1

// Рекурсивно вызываем функцию для левого и правого поддеревьев и суммируем результаты

return countLeaves(node->left) + countLeaves(node->right);

}

// Приватная рекурсивная функция для подсчета узлов с четными ключами

int countEvenNodes(TreeNode\* node) {

if (node == nullptr) return 0; // Условие выхода из рекурсии: если узел пустой, возвращаем 0

int count = 0;

if (node->key % 2 == 0) { // Если ключ узла четный, увеличиваем счетчик

count = 1;

}

// Рекурсивно вызываем функцию для левого и правого поддеревьев и суммируем результаты

return count + countEvenNodes(node->left) + countEvenNodes(node->right);

}

public:

// Конструктор для создания пустого дерева

BinarySearchTree() : root(nullptr) {}

// Деструктор для очистки памяти

~BinarySearchTree() {

clearTree(root);

}

// Публичная функция для вставки ключа в дерево

void insert(int key) {

root = insert(root, key);

}

// Публичная функция для вывода дерева в порядке возрастания

void inorderTraversal() {

inorderTraversal(root);

cout << endl;

}

// Публичная функция для очистки дерева от всех узлов

void clear() {

clearTree(root);

root = nullptr;

}

// Публичная функция для подсчета количества листьев в дереве

int countLeaves() {

return countLeaves(root);

}

// Публичная функция для подсчета количества узлов с четными ключами в дереве

int countEvenNodes() {

return countEvenNodes(root);

}

};

// Основная функция

int main() {

setlocale(LC\_ALL, "Rus");

BinarySearchTree bst;

int choice, key;

// Основной цикл программы

do {

cout << "Выберите операцию:\n";

cout << "1. Вставить узел\n";

cout << "2. Вывести дерево в порядке возрастания\n";

cout << "3. Очистить дерево\n";

cout << "4. Посчитать количество листьев\n";

cout << "5. Посчитать количество узлов с четными ключами\n";

cout << "0. Выйти\n";

cout << "Ваш выбор: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Введите значение для вставки: ";

cin >> key;

bst.insert(key);

break;

case 2:

cout << "Дерево в порядке возрастания:\n";

bst.inorderTraversal();

break;

case 3:

bst.clear();

cout << "Дерево очищено.\n";

break;

case 4:

cout << "Количество листьев в дереве: " << bst.countLeaves() << endl;

break;

case 5:

cout << "Количество узлов с четными ключами: " << bst.countEvenNodes() << endl;

break;

case 0:

cout << "Программа завершена.\n";

break;

default:

cout << "Некорректный выбор. Пожалуйста, попробуйте еще раз.\n";

break;

}

} while (choice != 0);

return 0;

}

**Результат выполнения:**

