МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра прикладной математики и информатики

ОТЧЁТ

по лабораторной работе №3 «Рефакторинг лабораторной работы №3»

по дисциплине

Структуры данных

(наименование дисциплины)

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Санников А.Н.

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Зырянов Е.А.

(подпись) (фамилия, и.,о.)

22-ПМ-1

(шифр группы)

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Задание на лабораторную работу:*** Осуществить рефакторинг лабораторной работы №1 «Списки. Стеки. Очереди».

***Задание проверяемой лабораторной работы****:* Дано бинарное дерево. Найти ветви с мах числом ветвлений.

|  |  |  |
| --- | --- | --- |
| Ошибка | Было | Стало |
| Неправильное комментирование кода | Комментарии отсутствовали | Добавлены комментарии к функциям |
| Использование разных стилей кода | Неправильный змеиный регистр | Исправленный змеиный регистр |
| Использование разных стилей расстановки скобок | Отсутствие какого либо стиля расстановки скобок | Применение стиля Кернигена и Ричи |
| Отсутствие пробелов в операторах выражения | В операторах отсутствовали пробелы | Добавлены пробелы |
| Непонятные символы | В коде были переменные, название которых подобрано не правильно | Замена на нормальные имена |

До ревью:

#include <iostream>

struct NodeList {

private:

int data;

NodeList\* next;

public:

NodeList() {

next = NULL;

}

int get\_data() {

return data;

}

NodeList\* get\_next() {

return next;

}

void set\_data(int data) {

this->data = data;

}

void set\_next(NodeList\* next) {

this->next = next;

}

int\* get\_data\_link() {

return &data;

}

};

class list {

private:

NodeList\* head = NULL;

public:

void push\_back(int data) {

if (head == NULL) {

head = new NodeList;

head->set\_data(data);

return;

}

NodeList\* tmp = head;

while (tmp->get\_next()) {

tmp = tmp->get\_next();

}

tmp->set\_next(new NodeList);

tmp->get\_next()->set\_data(data);

}

void print() {

NodeList\* tmp = head;

while (tmp) {

std::cout << tmp->get\_data() << ' ';

tmp = tmp->get\_next();

}

std::cout << '\n';

}

int len() {

NodeList\* tmp = head;

int c = 0;

while (tmp) {

c += 1;

tmp = tmp->get\_next();

}

return c;

}

int\* operator[](int i) {

NodeList\* tmp = head;

for (int j = 0; j < i; j++) {

tmp = tmp->get\_next();

}

return tmp->get\_data\_link();

}

};

class BinaryTree {

public:

struct NodeTree {

int value;

int count;

NodeTree\* left;

NodeTree\* right;

};

private:

NodeTree\* root;

NodeTree\* AddNode(NodeTree\* root, int value) {

if (root == nullptr) {

root = new NodeTree;

root->value = value;

root->count = 1;

root->left = root->right = nullptr;

}

else if (root->value > value) {

root->left = AddNode(root->left, value);

}

else if (root->value < value) {

root->right = AddNode(root->right, value);

}

else {

root->count++;

}

return root;

}

void PrintTree(NodeTree\* root) {

if (root == nullptr)

return;

PrintTree(root->left);

for (int i = 0; i < root->count; ++i)

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

PrintTree(root->right);

}

void DeleteTree(NodeTree\* root) {

if (root == nullptr)

return;

else {

DeleteTree(root->left);

DeleteTree(root->right);

delete root;

}

}

int maxDepth(NodeTree\* node) {

if (node == nullptr)

return 0;

int left = maxDepth(node->left);

int right = maxDepth(node->right);

return std::max(left, right) + 1;

}

void printMaxBranches(NodeTree\* Node,list path, int pathLen) {

if (Node == nullptr)

return;

if (path.len() <= pathLen)

path.push\_back(Node->value);

else

\*(path[pathLen]) = Node->value;

pathLen++;

if (Node->left == nullptr && Node->right == nullptr && pathLen == maxDepth(root))

path.print();

else {

printMaxBranches(Node->left, path, pathLen);

printMaxBranches(Node->right, path, pathLen);

}

}

public:

BinaryTree() : root(nullptr) {}

~BinaryTree() {

if (root)

DeleteTree(root);

}

void Add(int value) {

root = AddNode(root, value);

}

void Print() {

PrintTree(root);

}

void Clear() {

if (root) {

DeleteTree(root);

root = nullptr;

}

}

void PrintMaxBranch() {

int max\_branches = maxDepth(root);

std::cout << "Максимальное число ветвлений: " << max\_branches << std::endl;

}

void PrintMaxBranches() {

list path;

std::cout << "Максимальные ветки: " << '\n';

printMaxBranches(root, path, 0);

}

};

int main() {

using namespace std;

setlocale(LC\_ALL, "Russian");

BinaryTree Tree;

int a,n;

cout << "Введите количество узлов: ";

cin >> n; cout << '\n';

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

cin >> a;

Tree.Add(a);

}

Tree.PrintMaxBranch();

Tree.PrintMaxBranches();

return 0;

}

После:

#include <iostream>

using namespace std;

// Структура для узлов односвязного списка

struct Node\_List {

private:

int Data; // данные узла

Node\_List\* Next; // указатель на следующий узел

public:

Node\_List() { // конструктор по умолчанию

Next = nullptr; // инициализация указателя на следующий узел

}

int Get\_Data() { // метод для получения данных узла

return Data; // возвращает данные узла

}

Node\_List\* Get\_Next() { // метод для получения указателя на следующий узел

return Next; // возвращает указатель на следующий узел

}

void Set\_Data( int Data ) { // метод для установки данных узла

this->Data = Data; // устанавливает данные узла

}

void Set\_Next( Node\_List\* Next ) { // метод для установки указателя на следующий узел

this->Next = Next; // устанавливает указатель на следующий узел

}

int\* Get\_Data\_Link() { // метод для получения ссылки на данные узла

return &Data; // возвращает ссылку на данные узла

}

};

// Класс для односвязного списка

class List {

private:

Node\_List\* Head = nullptr; // указатель на начало списка

public:

void Push\_Back( int Data ) { // метод для добавления нового узла в конец списка

if ( Head == nullptr ) { // если список пуст

Head = new Node\_List; // выделяем память под новый узел

Head->Set\_Data( Data ); // устанавливаем данные узла

return; // выходим из метода

}

Node\_List\* Current\_Node = Head; // создаем временный указатель и устанавливаем его на начало списка

while ( Current\_Node->Get\_Next() ) { // пока есть следующий узел

Current\_Node = Current\_Node->Get\_Next(); // двигаемся по списку

}

Current\_Node->Set\_Next( new Node\_List ); // выделяем память под новый узел и устанавливаем его как следующий узел

Current\_Node->Get\_Next()->Set\_Data( Data ); // устанавливаем данные нового узла

}

void Print() { // метод для вывода содержимого списка

Node\_List\* Current\_Node = Head; // создаем временный указатель и устанавливаем его на начало списка

while ( Current\_Node ) { // пока не достигнут конец списка

cout << Current\_Node->Get\_Data() << ' '; // выводим данные текущего узла

Current\_Node = Current\_Node->Get\_Next(); // двигаемся по списку

}

cout << '\n'; // выводим символ новой строки

}

int Len() { // метод для определения длины списка

Node\_List\* Current\_Node = Head; // создаем временный указатель и устанавливаем его на начало списка

int Count = 0; // переменная для подсчета количества узлов

while ( Current\_Node ) { // пока не достигнут конец списка

Count += 1; // увеличиваем счетчик узлов

Current\_Node = Current\_Node->Get\_Next(); // двигаемся по списку

}

return Count; // возвращаем количество узлов

}

int\* operator[]( int Index ) { // оператор индексации для доступа к данным узла по индексу

Node\_List\* Current\_Node = Head; // создаем временный указатель и устанавливаем его на начало списка

for ( int i = 0; i < Index; i++ ) { // цикл по списку до указанного индекса

Current\_Node = Current\_Node->Get\_Next(); // двигаемся по списку

}

return Current\_Node->Get\_Data\_Link(); // возвращаем ссылку на данные узла

}

};

// Класс для бинарного дерева

class Binary\_Tree {

public:

struct Node\_Tree {

int Value; // данные узла

int Count; // количество повторений значения узла

Node\_Tree\* Left; // указатель на левого потомка

Node\_Tree\* Right; // указатель на правого потомка

};

private:

Node\_Tree\* Root; // Указатель на корневой узел бинарного дерева

Node\_Tree\* Add\_Node( Node\_Tree\* Root, int Value ) { // Приватный метод для добавления узла в дерево

if ( Root == nullptr ) { // Если корневой узел пустой

Root = new Node\_Tree; // Создаем новый узел

Root->Value = Value; // Устанавливаем значение узла

Root->Count = 1; // Инициализируем количество повторений значения узла

Root->Left = Root->Right = nullptr; // Инициализируем указатели на потомков узла

}

else if ( Root->Value > Value ) { // Если значение корневого узла больше вставляемого значения

Root->Left = Add\_Node( Root->Left, Value ); // Рекурсивно добавляем узел в левое поддерево

}

else if ( Root->Value < Value ) { // Если значение корневого узла меньше вставляемого значения

Root->Right = Add\_Node( Root->Right, Value ); // Рекурсивно добавляем узел в правое поддерево

}

else { // Если значение уже существует в дереве

Root->Count++; // Увеличиваем количество повторений значения узла

}

return Root; // Возвращаем указатель на корневой узел

}

void Print\_Tree( Node\_Tree\* Root ) { // Приватный метод для вывода значений узлов дерева в порядке возрастания

if ( Root == nullptr ) // Если текущий узел пуст

return; // Завершаем выполнение метода

Print\_Tree( Root->Left ); // Рекурсивно вызываем метод для левого поддерева

for ( int i = 0; i < Root->Count; ++i ) // Печатаем значение текущего узла несколько раз в зависимости от его количества

cout << Root->Value << " "; // Выводим значение узла

Print\_Tree( Root->Right ); // Рекурсивно вызываем метод для правого поддерева

}

void Delete\_Tree( Node\_Tree\* Root ) { // Приватный рекурсивный метод для удаления всех узлов дерева

if ( Root == nullptr ) // Если текущий узел пуст

return; // Завершаем выполнение метода

else { // Иначе

Delete\_Tree( Root->Left ); // Рекурсивно удаляем левое поддерево

Delete\_Tree( Root->Right ); // Рекурсивно удаляем правое поддерево

delete Root; // Удаляем текущий узел

}

}

int Max\_Depth( Node\_Tree\* Node ) { // Приватный метод для определения максимальной глубины дерева

if (Node == nullptr) // Если текущий узел пуст

return 0; // Возвращаем ноль

int Left = Max\_Depth( Node->Left ); // Рекурсивно определяем максимальную глубину левого поддерева

int Right = Max\_Depth( Node->Right ); // Рекурсивно определяем максимальную глубину правого поддерева

return max( Left, Right ) + 1; // Возвращаем максимальное значение глубины плюс один

}

void Print\_Max\_Branches( Node\_Tree\* Node, List Path, int Path\_Length ) { // Приватный рекурсивный метод для вывода максимальных веток дерева

if ( Node == nullptr ) // Если текущий узел пуст

return; // Завершаем выполнение метода

if ( Path.Len() <= Path\_Length ) // Если длина пути меньше или равна текущей длине пути

Path.Push\_Back( Node->Value ); // Добавляем значение текущего узла в путь

else

\*( Path[ Path\_Length ] ) = Node->Value; // Иначе перезаписываем значение в пути на значение текущего узла

Path\_Length++; // Увеличиваем текущую длину пути

if ( Node->Left == nullptr && Node->Right == nullptr && Path\_Length == Max\_Depth( Root ) ) // Если текущий узел - лист и текущая длина пути равна максимальной глубине дерева

Path.Print(); // Выводим путь

else { // Иначе

Print\_Max\_Branches( Node->Left, Path, Path\_Length ); // Рекурсивно вызываем метод для левого поддерева

Print\_Max\_Branches( Node->Right, Path, Path\_Length ); // Рекурсивно вызываем метод для правого поддерева

}

}

public:

Binary\_Tree() : Root( nullptr ) {} // Конструктор класса для создания пустого бинарного дерева

~Binary\_Tree() { // Деструктор класса для освобождения памяти, занимаемой узлами дерева

if ( Root )

Delete\_Tree( Root ); // Удаляем все узлы дерева, если они существуют

}

void Add( int Value ) { // Метод для добавления нового узла в дерево

Root = Add\_Node( Root, Value ); // Вызываем приватный метод для добавления узла

}

void Print() { // Метод для печати значений узлов дерева в порядке возрастания

Print\_Tree( Root ); // Вызываем приватный метод для печати значений узлов дерева

}

void Clear() { // Метод для очистки дерева (удаления всех узлов)

if ( Root ) { // Если дерево не пустое

Delete\_Tree( Root ); // Удаляем все узлы дерева

Root = nullptr; // Обнуляем указатель на корневой узел

}

}

void Print\_Max\_Branch() { // Метод для вывода максимальной глубины дерева (максимального числа ветвей)

int Max\_Branches = Max\_Depth( Root ); // Получаем максимальную глубину дерева

cout << "Максимальное число ветвей: " << Max\_Branches << endl; // Выводим результат

}

void Print\_Max\_Branches() { // Метод для вывода максимальных веток дерева

List Current\_Path; // Создаем объект класса List для хранения текущего пути

cout << "Максимальные ветки: " << '\n'; // Выводим заголовок

Print\_Max\_Branches( Root, Current\_Path, 0 ); // Вызываем приватный метод для вывода максимальных веток

}

};

int main() {

setlocale( LC\_ALL, "Russian" );

Binary\_Tree Tree; // Создаем объект класса Binary\_Tree для представления бинарного дерева

int Value, Nodes;

cout << "Введите количество узлов: "; // Запрос пользователю на ввод количества узлов

cin >> Nodes; cout << '\n'; // Чтение количества узлов

for ( int i = 0; i < Nodes; ++i ) { // Цикл для ввода значений узлов

cin >> Value; // Чтение значения узла

Tree.Add( Value ); // Добавление узла в дерево

}

Tree.Print\_Max\_Branch(); // Вывод максимальной глубины дерева

Tree.Print\_Max\_Branches(); // Вывод максимальных веток дерева

return 0;

}