**Министерство науки и высшего образования РФ  
ФГБОУ ВО  
Московский авиационный институт**

**(национальный исследовательский университет)**

Институт №3

Системы управления, информатики и электроэнергетики

Кафедра 304

**Отчет по лабораторной работе №2**

**по учебной дисциплине**

**«Структуры и алгоритмы обработки данных»**

**на тему:**

***«Бинарные деревья поиска»***

*Группа: М30-209Б-19*

*Варианты №4*

*Выполнили:*

*Кузнецов И.И.*

*Приняли:*

*Ким Р. В.*

*Ведьманов И. С.*

*Москва 2021*

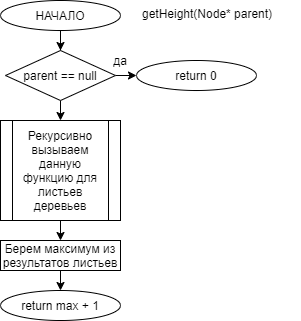
**Задание**  
1. Реализовать функции вставки, поиска, удаления узла, обхода дерева, вывода дерева на экран, нахождения высоты дерева и количества узлов.

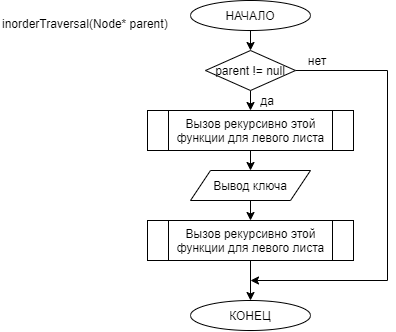
2. Реализовать заданную функцию в соответствии с вариантом: T – тип ключей, D – диапазон изменения значений ключей.

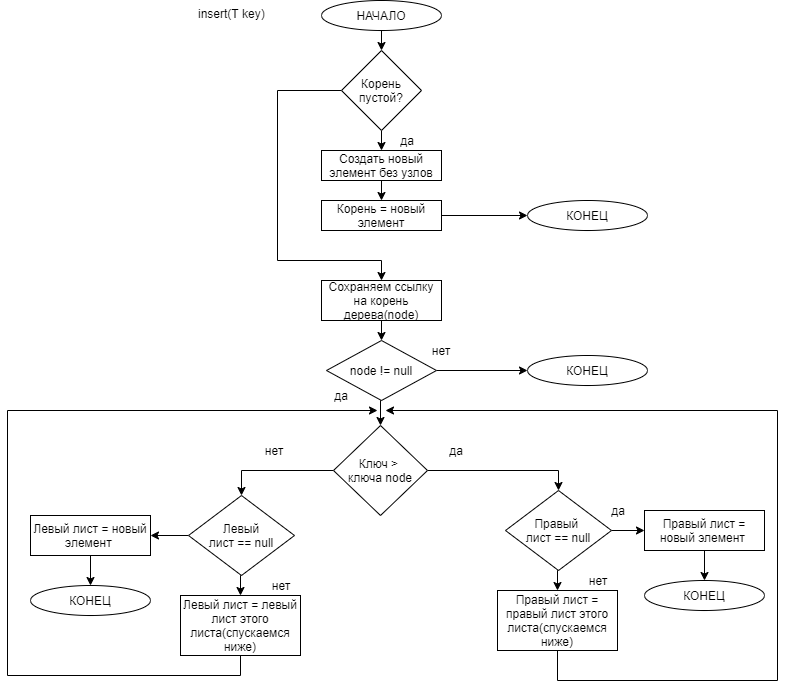
Вариант №14

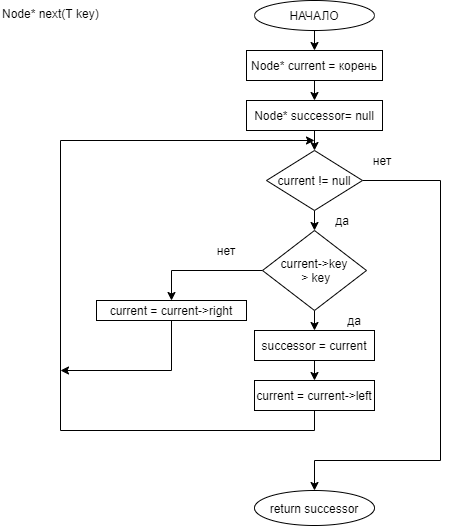
|  |  |  |  |
| --- | --- | --- | --- |
| № | T | D | Функция |
| 14 | char | [а…я, А…Я] | Подсчет количества согласных на 2 и 3 уровнях |

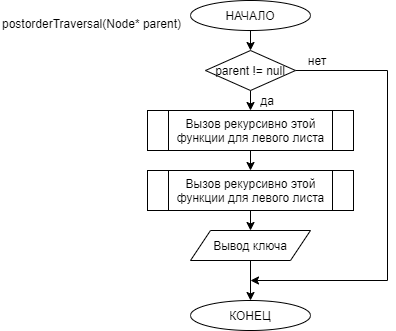
**Блок схема**

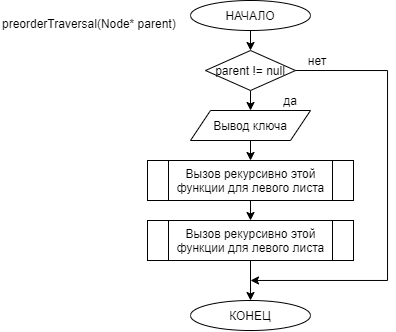
****

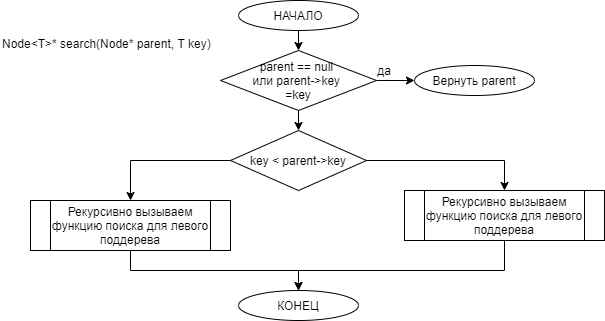
****

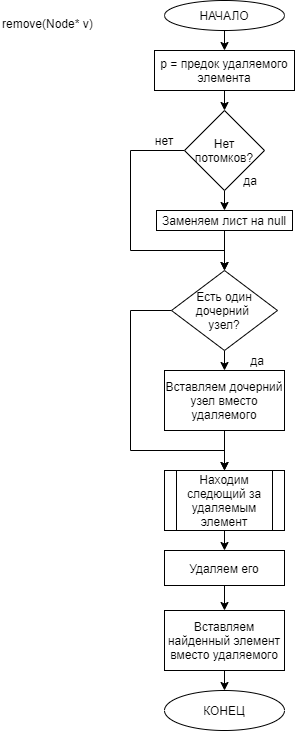
****

****

****

****

****

****

**Код программы**

**Заголовок элемента**

**template<typename T>  
class Node {  
public:  
 T key;  
 Node \*parent;  
 Node \*left;  
 Node \*right;  
  
 explicit Node(  
 T key = T(),  
 Node \*parent = nullptr,  
 Node \*left = nullptr,  
 Node \*right = nullptr  
 ) {  
 this->key = key;  
 this->parent = parent;  
 this->left = left;  
 this->right = right;  
 }  
};**

**Заголовок дерева**

**template<typename T>  
class Tree {  
public:  
 Tree();  
 ~Tree();  
 int getSize();  
 int getHeight();  
 void inorderTraversal();  
 void preorderTraversal();  
 void postorderTraversal();  
 void print();  
 void insert(T key);  
 void search(T key);  
 void remove(T key);  
 int count();  
private:  
 Node<T> \*ROOT; //корень  
 int SIZE;  
 int getHeight(Node<T> \*parent);  
 void inorderTraversal(Node<T> \*parent);  
 void preorderTraversal(Node<T> \*parent);  
 void postorderTraversal(Node<T> \*parent);  
 void print(Node<T> \*node);  
 void print(Node<T> \*parent, int indent);  
 Node<T> \*search(Node<T> \*parent, T key);  
 void remove(Node<T> \*v);  
 Node<T> \*next(T key);  
};**

#include "Tree.h"

#include <iostream>

#include <iomanip>

#include <queue>

using namespace std;

template<typename T>

Tree<T>::Tree() {

SIZE = 0;

ROOT = nullptr;

}

template<typename T>

Tree<T>::~Tree() {

// remove(ROOT);

}

template<typename T>

int Tree<T>::getSize() {

return SIZE;

}

template<typename T>

int Tree<T>::getHeight() {

return getHeight(ROOT);

}

template<typename T>

int Tree<T>::getHeight(Node<T>\* parent) {

if (!parent) return 0;

return 1 + max(getHeight(parent->left), getHeight(parent->right));

}

template<typename T>

void Tree<T>::inorderTraversal() {

if (!ROOT) {

cout << "Tree is empty!" << endl;

return;

}

inorderTraversal(ROOT);

}

template<typename T>

void Tree<T>::inorderTraversal(Node<T>\* parent) {

if (parent != nullptr) {

inorderTraversal(parent->left);

cout << parent->key << " ";

inorderTraversal(parent->right);

}

}

template<typename T>

void Tree<T>::preorderTraversal() {

if (!ROOT) {

cout << "Tree is empty!" << endl;

return;

}

preorderTraversal(ROOT);

}

template<typename T>

void Tree<T>::preorderTraversal(Node<T>\* parent) {

if (parent != nullptr) {

cout << parent->key << " ";

preorderTraversal(parent->left);

preorderTraversal(parent->right);

}

}

template<typename T>

void Tree<T>::postorderTraversal() {

if (!ROOT) {

cout << "Tree is empty!" << endl;

return;

}

postorderTraversal(ROOT);

}

template<typename T>

void Tree<T>::postorderTraversal(Node<T>\* parent) {

if (parent != nullptr) {

preorderTraversal(parent->left);

preorderTraversal(parent->right);

cout << parent->key << " ";

}

}

template<typename T>

void Tree<T>::print() {

if (!ROOT) {

cout << "Tree is empty!" << endl;

return;

}

print(ROOT, 0);

}

template<typename T>

void Tree<T>::print(Node<T>\* parent, int indent) {

if (parent != nullptr) {

if (parent->right) {

print(parent->right, indent + 4);

}

if (indent) {

cout << setw(indent) << ' ';

}

if (parent->right) {

cout << " /\n" << setw(indent) << ' ';

}

cout << parent->key << "\n ";

if (parent->left) {

cout << setw(indent) << ' ' << " \\\n";

print(parent->left, indent + 4);

}

}

}

template<typename T>

void Tree<T>::print(Node<T>\* node) {

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

cout << "left: ";

if (node->left != nullptr) {

cout << node->left->key << endl;

}

else {

cout << "nullptr" << endl;

}

cout << "right: ";

if (node->right != nullptr) {

cout << node->right->key << endl;

}

else {

cout << "nullptr" << endl;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<typename T>

void Tree<T>::insert(T key) {

if (!ROOT) { //если дерево пустое

ROOT = new Node<T>(key, nullptr, nullptr, nullptr);

SIZE++;

return;

}

Node<T>\* node = ROOT;

while (node != nullptr) {

if (key > node->key) {

if (node->right == nullptr) {

node->right = new Node<T>(key, node, nullptr, nullptr);

break;

}

else {

node = node->right;

}

}

else {

if (node->left == nullptr) {

node->left = new Node<T>(key, node, nullptr, nullptr);

break;

}

else {

node = parent->left;

}

}

}

SIZE++;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<typename T>

void Tree<T>::search(T key) {

Node<T>\* res = search(ROOT, key);

if (!res) {

cout << "Not found!";

return;

}

print(res);

}

/\*\*

\* Для каждого узла функция сравнивает значение его ключа с искомым ключом.

\* Если ключи одинаковы, то функция возвращает текущий узел,

\* в противном случае функция вызывается рекурсивно для левого или правого поддерева.

\*/

template<typename T>

Node<T>\* Tree<T>::search(Node<T>\* parent, T key) {

if ((!parent) || (parent->key == key))

return parent;

if (key < parent->key)

return search(parent->left, key);

else

return search(parent->right, key);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<typename T>

void Tree<T>::remove(T key) {

Node<T>\* v = search(ROOT, key);

remove(v);

}

/\*\*

\* Для удаления узла из бинарного дерева поиска нужно рассмотреть три возможные ситуации.

\* 1) Если у узла нет дочерних узлов, то у его родителя нужно просто заменить указатель на null.

\*

\* 2) Если у узла есть только один дочерний узел,

\* то нужно создать новую связь между родителем удаляемого узла и его дочерним узлом.

\*

\* 3) Если у узла два дочерних узла, то нужно найти следующий за ним элемент

\* (у этого элемента не будет левого потомка), его правого потомка подвесить

\* на место найденного элемента, а удаляемый узел заменить найденным узлом.

\* Таким образом, свойство бинарного дерева поиска не будет нарушено.

\*/

template<typename T>

void Tree<T>::remove(Node<T>\* v) {

Node<T>\* p = v->parent; // предок удаляемого элемента

if ((v->left == nullptr) && (v->right == nullptr)) { // первый случай: удаляемый элемент - лист

if (p->left == v) {

//delete (p->left);

p->left = nullptr;

}

if (p->right == v) {

//delete (p->right);

p->right = nullptr;

}

}

else if ((v->left == nullptr) || (v->right == nullptr)) { // второй случай: удаляемый элемент имеет одного потомка

if (v->left == nullptr) {

if (p->left == v) {

p->left = v->right;

}

else {

p->right = v->right;

}

v->right->parent = p;

}

else {

if (p->left == v) {

p->left = v->left;

}

else {

p->right = v->left;

}

v->left->parent = p;

}

}

else { // третий случай: удаляемый элемент имеет двух потомков

Node<T>\* successor = next(v->key);

v->key = successor->key;

if (successor->parent->left == successor) {

successor->parent->left = successor->right;

if (successor->right != nullptr) {

successor->right->parent = successor->parent;

}

}

else {

successor->parent->right = successor->right;

if (successor->right != nullptr) {

successor->right->parent = successor->parent;

}

}

}

}

template<typename T>

Node<T>\* Tree<T>::next(T key) {

Node<T>\* current = ROOT;

Node<T>\* successor = nullptr;

while (current != nullptr) {

if (current->key > key) {

successor = current;

current = current->left;

}

else {

current = current->right;

}

}

return successor;

}

template<typename T>

int Tree<T>::count() {

int counter = 0;

string soglasnie = "abcdfghklmnprstvwxzABCDFGHKLMNPRSTVWXZ";

queue<Node<T>\*> tmp\_queue;

Node<T>\* node;

if (ROOT == NULL)

return 0;

tmp\_queue.push(ROOT);

while (!tmp\_queue.empty()) {

int maximum = getHeight(ROOT);

node = tmp\_queue.front();

tmp\_queue.pop();

int level = maximum - getHeight(node) + 1;

if (level == 2 || level == 3) {

for (int i = 0; i < soglasnie.size()-1; ++i) {

if(node->key == soglasnie[i]){

counter++;

break;

}

}

}

cout << "Level = " << maximum - getHeight(node) + 1 << " Data = " << node->key << endl;

if (node->left) {

tmp\_queue.push(node->left);

}

if (node->right) {

tmp\_queue.push(node->right);

}

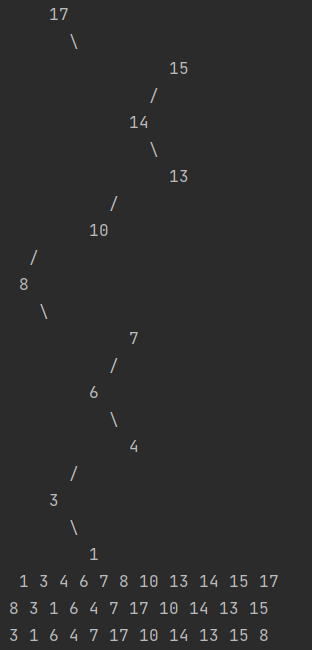
}

return counter;

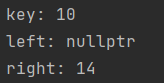
}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

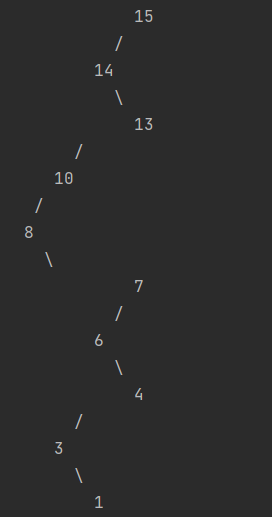
**Результаты работы программы**



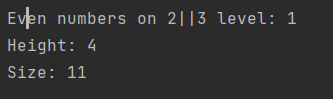
Построенное дерево и все виды обхода его



Поиск ключа, и вывод данных его потомков



Дерево после удаления элемента 17



**Вывод**

Во время выполнения лабораторной работы мы изучили принцип работы бинарного дерева поиска. Проделали все типичные операции, свойственные этой структуре данных.