#include <iostream>

#include <clocale>

#include <ctime>

using namespace std;

struct Tree

{

int info;

Tree\* left, \* right;

};

Tree\* List(int inf)

{

Tree\* t = new Tree; // Захват памяти

t->info = inf; // Формирование информационной части

t->left = t->right = NULL; // Формирование адресных частей

return t; // Возврат созданного указателя

}

void Add\_List(Tree\* root, int key)

{

Tree\* prev, \* t; // prev – указатель предка нового листа

bool find = true;

t = root;

prev = t;

while (t && find)

{

prev = t;

if (key == t->info)

{

find = false; // Ключ должен быть уникален

cout << "Dublucate Key!";

}

else

if (key < t->info) t = t->left;

else t = t->right;

}

if (find)

{ // Нашли нужное место

t = List(key); // Создаем новый лист

if (key < prev->info) prev->left = t;

else prev->right = t;

}

}

int unique\_el(int\* ms, int i)

{

int tmp, fl = 1;

while (fl)

{

fl = 0;

tmp = rand() % 20 - 10;

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

{

if (ms[j] == tmp)fl = 1;

}

}

return tmp;

}

void Show\_rev(Tree\*& tree)

{

if (NULL == tree) return; //Если дерева нет, выходим

Show\_rev(tree->left); //Обошли левое поддерево

Show\_rev(tree->right); //Обошли правое поддерево

cout << tree->info << endl; //Посетили узел

}

void Show(Tree\*& tree)

{

if (NULL == tree) return; //Если дерева нет, выходим

cout << tree->info << endl; //Посетили узел

Show(tree->left); //Обошли левое поддерево

Show(tree->right); //Обошли правое поддерево

}

void Make\_Blns(Tree\*\* p, int n, int k, int\* a)

{

if (n == k)

{

\*p = NULL;

return;

}

else

{

int m = (n + k) / 2;

\*p = new Tree;

(\*p)->info = a[m];

Make\_Blns(&(\*p)->left, n, m, a);

Make\_Blns(&(\*p)->right, m + 1, k, a);

}

}

void Del\_Tree(Tree\* t)

{

if (t != NULL)

{

Del\_Tree(t->left); // На левую ветвь

Del\_Tree(t->right); // На правую ветвь

delete t;

}

}

Tree\* Del\_Info(Tree\* root, int key)

{

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

// Del, Prev\_Del – удаляемый узел и его предыдущий (предок);

// R, Prev\_R – элемент, на который заменяется удаленный узел, и его пре\_док;

Del = root;

Prev\_Del = NULL;

//-------- Поиск удаляемого элемента и его предка по ключу key ---------

while (Del != NULL && Del->info != key)

{

Prev\_Del = Del;

if (Del->info > key) Del = Del->left;

else Del = Del->right;

}

if (Del == NULL)

{ // Элемент не найден

cout << "NOT Key!";

return root;

}

//-------------------- Поиск элемента R для замены --------------------------------

if (Del->right == NULL) R = Del->left;

else

if (Del->left == NULL) R = Del->right;

else

{

//---------------- Ищем самый правый узел в левом поддереве -----------------

Prev\_R = Del;

R = Del->left;

while (R->right != NULL)

{

Prev\_R = R;

R = R->right;

}

//----------- Нашли элемент для замены R и его предка Prev\_R -------------

if (Prev\_R == Del) R->right = Del->right;

else

{

R->right = Del->right;

Prev\_R->right = R->left;

R->left = Prev\_R;

}

}

if (Del == root) root = R; // Удаляя корень, заменяем его на R

else

//------- Поддерево R присоединяем к предку удаляемого узла -----------

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

Prev\_Del->left = R; // На левую ветвь

else Prev\_Del->right = R; // На правую ветвь

delete Del;

return root;

}

Tree\* find(Tree\* r, int d)

{

if (r == NULL)

{

return NULL; // не найден

}

if (r->info == d)

{

return r; // нашли!!!

}

Ilya, [Вс 05.06.22 11:08]

if (d <= r->info)

{

// left

if (r->left != NULL)

return find(r->left, d); // рекурсивный поиск влево

else

{

return NULL; // не найден

}

}

else

{

//right

if (r->right)

return find(r->right, d);// рекурсивный поиск вправо

else

{

return NULL; // не найден

}

}

}

void treeprint(Tree\* tree)

{

if (tree != NULL) { //Пока не встретится пустой узел

treeprint(tree->left); //Рекурсивная функция вывода левого поддерева

cout << tree->info << endl; //Отображаем корень дерева

treeprint(tree->right); //Рекурсивная функция вывода правого поддерева

}

}

bool isLeaf(Tree\* node)

{ //Функция проверки, что узел дерева является листом

if (!node->left && !node->right)

{

return true;

}

return false;

}

int height(Tree\* node)

{ //Функция определения высоты дерева (число уровней)

if (node == NULL)

{

return 0;

}

else

{

int lheight = height(node->left);

int rheight = height(node->right);

if (lheight > rheight)

{

return (lheight + 1);

}

else

{

return (rheight + 1);

}

}

}

int printGivenLevel(Tree\* node, int level)

{ //Функция подсчета листьев на заданном уровне

if (node == NULL)

return 0;

static int k = 0;

if (level == 1)

{

if (isLeaf(node))

{

k++;

}

}

else if (level > 1)

{

printGivenLevel(node->left, level - 1);

printGivenLevel(node->right, level - 1);

}

return k;

}

void printLevelOrder(Tree\* root)

{

int h = height(root); //Количество уровней (высота дерева)

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

{

cout << "Level #" << i << " => ";

int curr = printGivenLevel(root, i);

cout << printGivenLevel(root, i) - curr;

cout << "\n";

}

}

int main()

{

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

Tree\* root;

int tmp;

int\* ms = new int[10];

for (int i = 0; i < 10; i++)

{

ms[i] = unique\_el(ms, i);

}

for (int i = 0; i < 10; i++)

{

for (int i = 0; i < 9; i++)

{

if (ms[i] > ms[i + 1])

{

int tmp = ms[i];

ms[i] = ms[i + 1];

ms[i + 1] = tmp;

}

}

}

Make\_Blns(&root, 0, 10, ms);

cout << "Прямой обход сбалансированного дерева:\n";

Show(root);

cout << "Введите элемент, который вы хотите добавить: ";

int key;

cin >> key;

Add\_List(root, key);

cout << "Прямой обход сбалансированного дерева:\n";

Show(root);

cout << "Введите ключ, который вы хотите найти: ";

cin >> key;

if (find(root, key) == NULL) cout << "ключ не найден!";

else cout << "Найденный элемент: " << find(root, key)->info;

cout << "\nВведите ключ, который вы хотите удалить: ";

cin >> key;

Del\_Info(root, key);

cout << "Прямой обход сбалансированного дерева:\n";

Show(root);

cout << "Обратный обход сбалансированного дерева:\n";

Show\_rev(root);

cout << "Обход сбалансированного дерева по возрастанию:\n";

treeprint(root);

cout << "Кол-во листьев на каждом уровне:\n";

printLevelOrder(root); //Вывод числа листьев дерева по уровням

return 0;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <iostream>

#include <cstring>

#include <queue>

#pragma warning(disable:4996);

using namespace std;

struct Tree

{

int key = 0;

char str[500] = { 0 };

Tree\* left = NULL, \* right = NULL;

};

Tree\* AddTree(Tree\*, int, char\*);

Tree\* FindNode(Tree\*, int, Tree\*\* = NULL);

void ViewTree(Tree\*);

void DeleteTree(Tree\*);

int DeleteBranch(Tree\*\*, int);

int main()

{

setlocale(LC\_ALL, "rus");

Tree\* tree = NULL;

bool go = true;

while (go)

{

cout << "\n" << "1 - Добавить\n" << "2 - Показать\n" << "3 - Удалить дерево\n"

<< "4 - Удалить ветку\n" << "5 - Найти\n" << "0 - Выход\n";

int k = 0;

cin >> k;

cout << endl;

switch (k)

{

case 0:

{

go = false;

break;

}

case 1:

{

int x = 0;

cout << "Ключ: ";

cin >> x;

cout << "Информация: ";

cin.ignore(1000, '\n');

char inp[500];

cin.getline(inp, 500);

tree = AddTree(tree, x, inp);

break;

}

case 2:

{

ViewTree(tree);

break;

}

case 3:

{

DeleteTree(tree);

tree = NULL;

cout << "Дерево удалено\n";

break;

}

case 4:

{

int x = 0;

cout << "Ключ для удаления: ";

cin >> x;

if (DeleteBranch(&tree, x) == 1)

cout << "Ключ не найден\n\n";

else

cout << "Ветка удалена\n\n";

break;

}

case 5:

{

int x = 0;

cout << "Ключ для поиска: ";

cin >> x;

Tree\* t = FindNode(tree, x);

if (t == NULL)

{

cout << "Ключ не найден.\n";

break;

}

cout << "Адрес: " << t << "\nКлюч: " << t->key << "\nДанные: " << t->str << endl;

break;

}

}

}

DeleteTree(tree);

return 0;

}

Tree\* AddTree(Tree\* t, int x, char\* s)

{

Tree\* tmp = new Tree;

tmp->key = x;

strcpy(tmp->str, s);

if (t == NULL)

return tmp;

Tree\* search = t;

while (1)

{

if (x < search->key)

{

if (search->left)

{

search = search->left;

continue;

}

search->left = tmp;

break;

}

if (search->right)

{

search = search->right;

continue;

}

search->right = tmp;

break;

}

return t;

}

void ViewTree(Tree\* tree)

{

if (tree == NULL)

return;

queue<Tree\*> Q;

Q.push(tree);

Q.push(NULL);

while (Q.size())

{

Tree\* t = Q.front();

Q.pop();

if (t == NULL)

{

cout << endl;

if (Q.empty())

break;

Q.push(NULL);

continue;

}

cout << t->key << ":" << t->str << "\t";

if (t->left)

Q.push(t->left);

if (t->right)

Q.push(t->right);

}

return;

}

void DeleteTree(Tree\* t)

{

if (t == NULL)

return;

DeleteTree(t->left);

DeleteTree(t->right);

delete t;

}

Tree\* FindNode(Tree\* t, int x, Tree\*\* parent)

{

if (parent)

\*parent = NULL;

while (t)

{

if (t->key == x)

break;

if (parent)

\*parent = t;

if (x < t->key)

t = t->left;

else

t = t->right;

}

return t;

}

int DeleteBranch(Tree\*\* t, int x)

{

Tree\* parent = NULL;

Tree\* node = FindNode(\*t, x, &parent);

if (node == NULL)

return 1;

DeleteTree(node);

if (parent)

{

if (parent->left == node)

parent->left = NULL;

else

parent->right = NULL;

}

else

\*t = NULL;

return 0;

}