Файл TreeNode.h

#pragma once

#include <cstdlib>

#include <iostream>

#include<fstream>

#include <math.h>

#include<queue>

using namespace std;

//BinSTree зависит от TreeNode

template<class T>

class BinSTree;

//класс узла бинарного дерева

template <class T>

class TreeNode

{

private:

//указатели левого и правого дочерних узлов

TreeNode<T>\* left;

TreeNode<T>\* right;

public:

//открытый элемент, допускающий обновление

T data;

//конструктор

TreeNode(const T& item, TreeNode<T>\* lptr = NULL,

TreeNode<T>\* rptr = NULL) {

data = item;

left = lptr;

right = rptr;

}

//методы доступа к полям указателей

TreeNode<T>\* Left() const {

return left;

}

TreeNode<T>\* Right() const {

return right;

}

//сеттеры

void set\_Left(TreeNode<T>\* newNode) {

left = newNode;

}

void set\_Right(TreeNode<T>\* newNode) {

right = newNode;

}

//для доступа к полям

friend class BinSTree<T>;

};

//освободить память, занимаемую узлом

template<class T>

void FreeTreeNode(TreeNode<T>\* p) {

delete p;

}

//создание и инициализация узла бинарного дерева

template<class T>

TreeNode<T>\* GetTreeNode(const T& item, TreeNode<T>\* lptr = NULL,

TreeNode<T>\* rptr = NULL) {

TreeNode<T>\* p;

p = new TreeNode<T>(item, lptr, rptr);

//проверка памяти

if (p == NULL)

{

cout << "Error!" << endl;

exit(1);

}

return p;

}

//симметричное рекурсивное прохождение узлов дерева

template<class T>

void Inorder(TreeNode<T>\* t, void visit(T& item)) {

if (t != NULL) {

Inorder(t->Left(), visit);

visit(t->data);

Inorder(t->Right(), visit);

}

}

//обратное рекурсивное прохождение узлов дерева

template<class T>

void Postorder(TreeNode<T>\* t, void visit(T& item)) {

if (t != NULL) {

Postorder(t->Left(), visit);

Postorder(t->Right(), visit);

visit(t->data);

}

}

//прямой метод прохождения узлов дерева

template<class T>

void Preorder(TreeNode<T>\* t, void visit(T& item)) {

if (t != NULL) {

visit(t->data);

Postorder(t->Left(), visit);

Postorder(t->Right(), visit);

}

}

//функция подсчета листьев дерева

//count предусматривает себя как глобальный параметр и передается по ссылке

template<class T>

void CountLeaf(TreeNode<T>\* t, int& count) {

if (t != NULL)

{

//используем обратный метод прохождения

CountLeaf(t->Left(), count);

CountLeaf(t->Right(), count);

//является ли узел листовым

if (t->Left() == NULL && t->Right() == NULL)

count++;

}

}

//количество элементов дерева

template <class T>

void Size(TreeNode<T>\* t, int& count) {

if (t != NULL) {

count++;

Size(t->Left(), count);

Size(t->Right(), count);

}

}

//функция подсчета глубины дерева

template <class T>

int Depth(TreeNode<T>\* t) {

int depthLeft, depthRight, depthval;

if (t == NULL)

//если корневой узел пустой, то глубина дерева -1

depthval = -1;

else

{

//используем обратный метод прохождения

depthLeft = Depth(t->Left());

depthRight = Depth(t->Right());

depthval = 1 + (depthLeft > depthRight ? depthLeft : depthRight);//прибавляем

наибольшего сына

}

return depthval;

}

//функция вставки для пострения дерева

template <class T>

void Insert(TreeNode<T>\* root, const T item) {

//t - текущий узел, parent - предыдущий узел

TreeNode<T>\* t = root, \* parent = NULL;

//создание узла из переданного значения

TreeNode<T>\* newNode = GetTreeNode(item);

//закончить на пустом дереве

while (t != NULL)

{

parent = t; //обновить указатель parent

//выбор пути

if (newNode->data < t->data) //если новые данные меньше текущего узла

t = t->Left(); //идём влево

else

t = t->Right(); //а иначе - вправо

}

if (parent == NULL) //если родителя нет

root = newNode; //вставить в качестве корневого узла

else if (newNode->data < parent->data) //если новые данные меньше родительского узла

parent->set\_Left(newNode); //то вставить в качестве левого потомка

else

parent->set\_Right(newNode); // иначе вставить в качестве правого потомка

}

//создание бинарного дерева из очереди

template <class T>

TreeNode<T>\* CreateBinTree(queue<T> Q\_data) {

TreeNode<T>\* root = GetTreeNode(Q\_data.front());

Q\_data.pop();

while (!Q\_data.empty()) {

Insert(root, Q\_data.front());

Q\_data.pop();

}

return root;

}

//вставка пробелов

void IndentBlanks(int num) {

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

cout << " ";

}

}

//горизонтальная печать дерева на экран

template <class T>

void PrintTree(TreeNode<T>\* t, int level) {

if (t != NULL) { //пока указатель не пуст

PrintTree(t->Right(), level + 1); //печатаем правое поддерево

IndentBlanks(6 \* level); //вставка пробелов и выравнивание

cout << t->data << endl;

PrintTree(t->Left(), level + 1); //печатаем левое поддерево

}

}

//горизонтальная печать дерева в файл

template <class T>

void PrintTreeFile(TreeNode<T>\* t, int level, string s1) {

if (t != NULL) { //пока указатель не пуст

ofstream f(s1, ios\_base::app);

PrintTreeFile(t->Right(), level + 1, s1); //печатаем правое поддерево

for (int i = 0; i < 6 \* level; i++) {

f << " ";

}

f << t->data << endl;

PrintTreeFile(t->Left(), level + 1, s1); //печатаем левое поддерево

}

}

//функция удаления дерева

template <class T>

void DeleteTree(TreeNode<T>\* t) {

if (t != NULL) {

DeleteTree(t->Left());

DeleteTree(t->Right());

FreeTreeNode(t);

}

}

//функция удаления дерева под корень

template <class T>

void ClearTree(TreeNode<T>& t) {

DeleteTree(t);

//очищаем корень

t = NULL;

}

//функция создания дубликата дерева с возвращением указателя корня на новое дерво

template<class T>

TreeNode<T>\* CopyTree(TreeNode<T>\* t) {

TreeNode<T>\* newlptr, \* newrptr, \* newnode;

if (t == NULL)

return NULL;

if (t->Left() != NULL)

newlptr = CopyTree(t->Left());

else

newlptr = NULL;

if (t->Right() != NULL)

newrptr = CopyTree(t->Right());

else

newrptr = NULL;

//новое дерво строится снизу вверх

newnode = GetTreeNode(t->data, newlptr, newrptr);

return newnode;

}

//тип координат узла

struct Info {

int xIndent, yLevel;

};

template <class T>

void Print(queue<T>& Q, queue<Info>& QI) {

int lastDepth = 0; int lastIndent = 0;

while (!Q.empty()) {

Info Position = QI.front();

QI.pop();

if (Position.yLevel > lastDepth) {

cout << endl;

lastDepth = Position.yLevel;

lastIndent = 0;

}

IndentBlanks((Position.xIndent - lastIndent));

lastIndent = Position.xIndent;

cout << Q.front();

Q.pop();

}

}

template <class T>

void PrintFile(queue<T>& Q, queue<Info>& QI, string s1) {

ofstream f(s1, ios\_base::app);

int lastDepth = 0; int lastIndent = 0;

while (!Q.empty()) {

Info Position = QI.front();

QI.pop();

if (Position.yLevel > lastDepth) {

f << endl;

lastDepth = Position.yLevel;

lastIndent = 0;

}

//IndentBlanks((Position.xIndent - lastIndent));

for (int i = 0; i < (Position.xIndent - lastIndent); i++) {

f << " ";

}

lastIndent = Position.xIndent;

f << Q.front();

Q.pop();

}

}

//прохождение дерева по уровням

template <class T>

void LevelScan(TreeNode<T>\* t, queue<T>& out, queue<Info>& inf, int dataWidth, int screenWidth) {

//запомнить сыновей каждого узла в очереди чтобы их

//можно было посетить в этом порядке

queue<TreeNode<T>\*> Q; //очередь из узлов

queue<Info> QI; //очередь из координат узлов

Info Position; //текущие координаты

TreeNode<T>\* p; //указатель на обрабатываемый узел

//инициализируем очередь, вставив туда корень

Q.push(t);

//инициализируем очередь , вставляя координаты корня

QI.push({ screenWidth / 2 - (dataWidth / 2),0 });

//пока очередь не опустеет

while (!Q.empty()) {

//удалить первый узел и посетить

p = Q.front();

Q.pop();

Position = QI.front();

QI.pop();

out.push(p->data); //вставляем данные из узла в очередь

inf.push(Position);

if (p->Left() != NULL) {

Q.push(p->Left());

int div = pow(2, (Position.yLevel + 2));

QI.push({ Position.xIndent - (screenWidth / div) - (dataWidth / 2),Position.yLevel + 1 });

}

if (p->Right() != NULL) {

Q.push(p->Right());

int div = pow(2, (Position.yLevel + 2));

QI.push({ Position.xIndent + (screenWidth / div) - (dataWidth / 2),Position.yLevel + 1 });

}

}

}

//вертикальная печать дерева на экран

template <class T>

void PrintVTree(TreeNode<T>\* t, int dataWidth, int screenWidth, void f(queue<T>& Q\_data, queue<Info>& Q\_info)) {

queue<T> Q;

queue<Info> QI;

LevelScan(t, Q, QI, dataWidth, screenWidth);

f(Q, QI);

}

//вертикальная печать дерева в файл

template <class T>

void PrintVTreeFile(TreeNode<T>\* t, int dataWidth, int screenWidth, void f(queue<T>& Q\_data, queue<Info>& Q\_info, string s1), string s) {

queue<T> Q;

queue<Info> QI;

LevelScan(t, Q, QI, dataWidth, screenWidth);

f(Q, QI, s);

}

Файл BinSTree.h

#pragma once

#include"TreeNode.h"

#include<fstream>

using namespace std;

template <class T>

class BinSTree

{

protected:

//указатели на корень и текущий узел

TreeNode<T>\* root, \* current;

//число элементов дерева

int size;

//распределение/освобождение памяти

TreeNode<T>\* GetTreeNode(const T& item,

TreeNode<T>\* lptr,

TreeNode<T>\* rptr) {

TreeNode<T>\* p;

p = new TreeNode<T>(item, lptr, rptr);

//проверка памяти

if (p == NULL)

{

cout << "Error!" << endl;

exit(1);

}

return p;

}

void FreeTreeNode(TreeNode<T>\* p) {

delete p;

}

//используется конструктором копирования и оператором присвания

void DeleteTree(TreeNode<T>\* t) {

if (t != NULL) {

DeleteTree(t->Left());

DeleteTree(t->Right());

FreeTreeNode(t);

}

}

TreeNode<T>\* FindNode(const T& item,

TreeNode<T>\*& parent) const {

//пробежать по узлам

TreeNode<T>\* t = root;

//нет родителя

parent = NULL;

while (t != NULL) {

if (item == t->data)

break;

else

{

//обновить указатель и идти право/лево

parent = t;

if (item < t->data)

t = t->left;

else

t = t->right;

}

}

return t;

}

public:

//конструктор

BinSTree() {

root = nullptr;

current = nullptr;

size = 0;

}

//конструктор

BinSTree(const BinSTree<T>& tree) {

root = new TreeNode<T>(tree, nullptr, nullptr);

current = root;

size = 0;

}

//деструктор

~BinSTree() {

}

//оператор присваивания

BinSTree<T>& operator= (const BinSTree<T>& rhs) {

//проверка

if (this == &rhs)

return \*this;

//очистить текущее дерево,скопировать новое

ClearTree(this->root);

root = CopyTree(rhs.root);

current = root;

size = this->size;

return \*this;

}

//стандартные методы обработки списков

int Find(T& item) {

//принимает параметр parent

TreeNode<T>\* parent;

//искать значение, назначить совпавший узел текущим

current = FindNode(item, parent);

if (current != NULL) {

item = current->data;

return 1;

}

else

return 0;

}

//вставить item в дерево поиска

void Insert(const T& item) {

TreeNode<T>\* t = root, \* parent = NULL, \* newNode = new TreeNode<T>(item,

nullptr, nullptr); //t- текущий узел, parent- пердыдущий

while (t != NULL) { //закончить на пустом дереве

//обновить указатель parent и идти направо или налево

parent = t;

if (item < t->data)

t = t->left;

else

t = t->right;

}

if (parent == NULL) //если родителя нет, вставить в качестве корневого

узла

root = newNode;

else if (item < parent->data) //если item меньше родительского узла, вставить в

качестве левого сына

parent->left = newNode;

else //если item больше или равен родительскому узлу

parent->right = newNode; //вставить в качестве левого сына

//присвоить указателю current адрес нового узла и увеличить size на единицу

current = newNode;

size++;

}

void Delete(const T& val) {

remove\_helper(root, val);

}

TreeNode<T>\* remove\_helper(TreeNode<T>\* node, const T& val)

{

if (node == NULL)

return node;

if (val == node->data) {

TreeNode<T>\* tmp;

if (node->right == NULL)

tmp = node->left;

else {

TreeNode<T>\* ptr = node->right;

if (ptr->left == NULL) {

ptr->left = node->left;

tmp = ptr;

}

else {

TreeNode<T>\* pmin = ptr->left;

while (pmin->left != NULL) {

ptr = pmin;

pmin = ptr->left;

}

ptr->left = pmin->right;

pmin->left = node->left;

pmin->right = node->right;

tmp = pmin;

}

}

delete node;

return tmp;

}

else if (val < node->data)

node->left = remove\_helper(node->left, val);

else

node->right = remove\_helper(node->right, val);

return node;

}

void ClearList() {

ClearTree(this->root);

}

int ListEmpty() const;

int ListSize() const;

//методы, специфичные для деревьев

void Update(const T& item) {

if (current != NULL && current->data == item)

current->data = item;

else

Insert(item);

}

TreeNode<T>\* GetRoot() const {

return root;

}

void PrintTreeB(int level) {

current = root;

PrintTree(root, level);

}

void PrintTreeBFile(int level, string s1) {

current = root;

PrintTreeFile(root, level, s1);

}

void PrintVTreeB(int dataWidth, int screenWidth) {

current = root;

PrintVTree(root, dataWidth, screenWidth, Print);

}

void PrintVTreeBFile(int dataWidth, int screenWidth, string s1) {

current = root;

PrintVTreeFile(root, dataWidth, screenWidth, PrintFile, s1);

}

void CountLeafB(int& count) {

if (root == nullptr) count = 0;

CountLeaf(root, count);

}

int DepthB() {

if (root == nullptr)

return -1;

else

return Depth(root);

}

int SizeB() {

return size;

}

};

Файл MyStack.h

#pragma once

#include <iostream>

#include <cassert> //для assert

#include <iomanip> //для setw

using std::cout; using std::setw; using std::endl;

template <typename T>

class MyStack {

private:

T\* stacklist; //указатель на стек

const int size; //размер массива

int top; //индекс вершины

public:

MyStack(int = 10); //по умолчанию размер стека равен 10 элементам

MyStack(const MyStack<T>&); //конструктор копирования

~MyStack(); //деструктор

const T& peek(int) const; //показать n-й элемент от вершины стека

bool isEmpty() const; //проверка на пустой стек

int getSize() const; //получить размер стека

T\* getPtr() const; //получить указатель на стек

int getTop() const; //получить номер текущего элемента в стеке

void push(const T&); //поместить элемент в вершину стека

T pop(); //удалить элемент из вершины стека и вернуть его

void printStack(); //вывод стека на экран

void clear(); //очистка стека

};

//конструктор стека

template <typename T>

MyStack<T>::MyStack(int maxSize) : size(maxSize) { //инициализация константы

stacklist = new T[size + 1]; //выделить память под стек

top = 0; //инициализируем текущий элемент нулем;

}

//конструктор копирования

template <typename T>

MyStack<T>::MyStack(const MyStack<T>& otherStack) :

size(otherStack.getSize()) { //инициализация константы

stacklist = new T[size]; //выделить память под новый стек

top = otherStack.getTop(); //передаём индекс вершины

for (int i = 0; i < top; i++) //пробег по стеку

stacklist[i] = otherStack.getPtr()[i]; //копируем значения

}

//деструктора cтека

template <typename T>

MyStack<T>::~MyStack() {

delete[] stacklist; //удаляем стек

}

//функция добавления элемента в стек

template <typename T>

void MyStack<T>::push(const T& value) {

//assert проверяет размер стека

assert(top < size); //номер текущего элемента должен быть меньше размера стека

stacklist[top++] = value; //помещаем элемент в стек

}

//функция удаления элемента из стека

template <typename T>

T MyStack<T>::pop() {

//assert проверяет размер стека

assert(top > 0); //номер текущего элемента должен быть больше 0

return stacklist[--top]; //удаляем элемент из стека

}

//функция возвращает n-й элемент от вершины стека

template <typename T>

const T& MyStack<T>::peek(int nom) const {

//проверка диапазона индекса

assert(nom <= top);

return stacklist[top - nom]; // вернуть n-й элемент стека

}

//функция проверает стек на пустоту

template <typename T>

bool MyStack<T>::isEmpty() const {

return (top == 0);

}

//вывод стека на экран

template <typename T>

void MyStack<T>::printStack() {

for (int i = top - 1; i >= 0; i--) //пробег с вершины

cout << "|" << setw(4) << stacklist[i] << endl; //печатаем

}

//очистка стека

template <typename T>

void MyStack<T>::clear() {

top = 0;

}

//вернуть размер стека

template <typename T>

int MyStack<T>::getSize() const {

return size;

}

//вернуть указатель на стек (для конструктора копирования)

template <typename T>

T\* MyStack<T>::getPtr() const {

return stacklist;

}

//вернуть размер стека

template <typename T>

int MyStack<T>::getTop() const {

return top;

}

Файл Iterator.h

#pragma once

template<class T>

class Iterator

{

protected:

int iterationComplete;

public:

Iterator()

{

iterationComplete = 0;

}

virtual void Next() = 0;

virtual void Reset() = 0;

virtual T& Data() = 0;

virtual int EndOfList() const {

return iterationComplete;

}

};

Файл InorderIterator.h

#pragma once

#include "MyStack.h"

#include "TreeNode.h"

#include "Iterator.h"

template <class T>

class InorderIterator : public Iterator<T> {

private:

//стек адресов узлов

MyStack<TreeNode <T>\*> S;

//корень дерева и текущий узел

TreeNode<T>\* root, \* current;

//сканирование левого поддерева. используется функцией Next

TreeNode<T>\* GoFarLeft(TreeNode<T>\* t) {

//если t=NULL, вернуть NULL

if (t == NULL)

return NULL;

//пока не встретится узел с нулевым левым указателем,

while (t->Left() != NULL) {

S.push(t); //запоминая в стеке S адреса пройденных узлов

t = t->Left(); //спускаться по левым ветвям

}

return t; //возвратить указатель на этот узел

}

public:

//конструктор

InorderIterator(TreeNode<T>\* tree, int n);

//реализации базовых операций прохождения

virtual void Next(void);

virtual void Reset(void);

virtual T& Data(void);

//назначение итератору нового дерева

void SetTree(TreeNode<T>\* tree);

};

//конструктор

template<class T>

inline InorderIterator<T>::InorderIterator(TreeNode<T>\* tree, int n) : Iterator<T>(), root(tree), S(n){

this->iterationComplete = (this->root == NULL); //дерево может оказаться пустым

this->current = GoFarLeft(this->root); //узлом сканирования является самый левый узел

}

//движение по дереву

template<class T>

void InorderIterator<T>::Next(void) {

//ошибка, если все узлы уже посещались

if (this->iterationComplete) {

std::cerr << "Next: EndOfList!" << std::endl;

exit(1);

}

//если есть правое поддерево, спуститься до конца по его левой ветви,

//попутно запоминая в стеке адреса пройденных узлов

if (this->current->Right() != NULL)

this->current = GoFarLeft(this->current->Right());

//правого поддерева нет, но в стеке есть другие узлы,

//подлежащие обработке. вытолкнуть из стека новый текущий адрес

else if (!this->S.isEmpty()) { // продвинуться вверх по дереву

this->current = this->S.pop();

}

// нет ни правого поддерева, ни узлов в стеке

else this->iterationComplete = 1;

}

//установка начальных значений

template<class T>

void InorderIterator<T>::Reset(void) {

//дейсвтия как у конструктора

this->iterationComplete = (this->root == NULL);

this->current = GoFarLeft(this->root);

this->S.clear(); //и также очищаем стек

}

//геттер данных

template<class T>

T& InorderIterator<T>::Data(void) {

return this->current->data;

}

//сеттер дерева

template<class T>

void InorderIterator<T>::SetTree(TreeNode<T>\* tree) {

this->root = tree;

}

Файл lab10.cpp

#include <iostream>

#include <fstream>

#include <locale>

#include <chrono>

#include "InorderIterator.h"

#include "BinSTree.h"

using namespace std;

//заполнение массива

int\* create\_mas(int n, int min, int max) {

int\* res = new int[n];

srand(time(0)); //задаём условие рандома

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

res[i] = rand() % (max - min) + min; //заполняем в диапазоне

return res;

}

//печать массива

void print\_mas(int\* arr, int n, ostream& strm) {

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

if (i != 0 && i % 10 == 0) //когда 10 элементов в строке

strm << endl;

strm << arr[i] << " ";

}

strm << endl;

}

//сортировка деревом

template <class T>

void TreeSort(T\* arr, int n) {

//бинарное дерево поиска, в которое копируется массив

BinSTree<T> sortTree;

int i;

//включить каждый элемент массива в поисковое дерево

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

sortTree.Insert(arr[i]);

//объявить итератор симметричного прохождения для sortTree

InorderIterator<T> treeSortIter(sortTree.GetRoot(), sortTree.DepthB());

//выполнить симметричное прохождение дерева.

//скопировать каждый элемент снова в массив

i = 0;

//начинаем обход дерева

while (!treeSortIter.EndOfList()) {

arr[i++] = treeSortIter.Data();

treeSortIter.Next();

}

}

int main() {

setlocale(0, "");

ofstream fout; //файл с результатом

fout.open("Result.txt");

//по заданию 100 элементов

int n = 100;

int\* mas = create\_mas(n, 0, 1000); //получаем случайный массив чисел от 0 до 1000

//отображаем начальный массив

cout << "Start mas" << endl;

fout << "Start mas" << endl;

print\_mas(mas, n, cout);

print\_mas(mas, n, fout);

auto begin = std::chrono::steady\_clock::now();

TreeSort(mas, n);

auto end = std::chrono::steady\_clock::now();

auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);

cout << "End mas" << endl;

fout << "End mas" << endl;

print\_mas(mas, n, cout);

print\_mas(mas, n, fout);

cout << "Time = " << elapsed\_ms.count() << " milliseconds" << endl;

fout << "Time = " << elapsed\_ms.count() << " milliseconds" << endl;

fout.close();

}

