Node.h

#pragma once

#include <iostream>

#include <ctime>

using namespace std;

template <class T>

class Tree

{

private:

template <class T>

class Node

{

public:

T information;

Node<T>\* left = NULL;

Node<T>\* right = NULL;

Node(Node<T>\* left = NULL, Node<T>\* right = NULL) :

left(left),

right(right) {}

Node(const T& information\_, Node<T>\* left = NULL, Node<T>\* right = NULL) :

left(left),

right(right),

information(information\_){}

//рекурсивыные функции Delete, OtpuingLR, Copy нужны для создания методов

friend void Delete(Node<T>\*& top)

{

if (top)

{

Delete(top->left);

Delete(top->right);

delete top;

}

top = NULL;

}

friend void OutputingLR(Node<T>\* top, std::ostream& os)

{

if (top)

{

OutputingLR(top->left, os);

os << top->information << ' ';

OutputingLR(top->right, os);

}

}

friend void Copy(Node<T>\*& top, Node<T>\* other)

{

if (other)

{

top = new Node<T>;

top->information = other->information;

Copy(top->left, other->left);

Copy(top->right, other->right);

}

}

};

Node<T>\* top;

public:

Tree():

top(NULL)

{}

explicit Tree(T information)

{

top = new Node<T>(information);

}

explicit Tree(const Node<T>& other):

Node()

{

Copy(top, other.top);

}

~Tree()

{

delate();

}

bool empty() const

{

return top == NULL;

}

//добавение элемента. если больше либо равно, т направо, если меньше, то налево

void add\_element(T newInformation)

{

if (top)

{

Node<T>\* temp = top;

while (temp)

{

if (newInformation >= temp->information)

{

if (temp->right)

temp = temp->right;

else

{

temp->right = new Node<T>(newInformation);

break;

}

}

else

{

if (temp->left)

temp = temp->left;

else

{

temp->left = new Node<T>(std::move(newInformation));

break;

}

}

}

}

else

{

top = new Node<T>(newInformation);

}

}

//зеполнение случайными числами

template<class T = int>

void random(int size, int random)

{

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

{

add\_element(rand() % random + 1);

}

}

//удаление дерева с помощью рекурсивной функции

void delate()

{

Delete(top);

}

//вывод с помощью рекурсивной функции в порядке возрастания

friend ostream& operator<< (ostream& os, const Tree<T>& top)

{

OutputingLR(top.top,os);

return os;

}

//индивидуальное задание с удалением узла с макисмальным элементом

void Del\_max()

{

Node<T>\* temp = top;

if (!(top->right))

{

top = top->left;

temp->left = NULL;

delete temp;

}

else

{

while (temp->right->right)

{

temp = temp->right;

}

Node<T>\* temp1 = temp->right;

temp->right = temp->right->left;

temp1->left = NULL;

delete temp1;

}

}

//функция нахождения элемента, если не найдено, то NULL

Node<T> find(T information)

{

Node<T>\* temp = top;

while (temp != NULL && temp->information != information)

{

if (information > temp->information)

{

temp = temp ->right;

}

else

{

temp = temp->left;

}

}

if (temp != NULL)

{

return temp;

}

else

{

throw std::exception("Not found");

}

}

};

//ввод с помощью функции добавления элемента

template<class T>

std::istream& operator>> (std::istream& is, Tree<T>& top)

{

top.delate();

cout << "Введите количество чисел в дереве:" << endl;

size\_t sizeTree;

is >> sizeTree;

T number;

cout << "Введите дерево:" << endl;

for (size\_t i = 0; i < sizeTree; ++i)

{

is >> number;

top.add\_element(number);

}

return is;

}

String.h

#pragma once

#include <cstddef>

#include <algorithm>

class String

{

private:

char\* str\_;

size\_t size\_;

public:

String():

str\_(NULL),

size\_(0){}

String(const char\* str) :

size\_(strlen(str))

{

str\_ = new char[size\_ + 1];

for (size\_t i = 0; i < size\_; ++i)

{

str\_[i] = str[i];

}

str\_[size\_] = '\0';

}

String(String&& str) noexcept:

str\_(NULL),

size\_(0)

{

std::swap(str\_, str.str\_);

std::swap(size\_, str.size\_);

}

String(const String& str) :

size\_(str.size\_),

str\_(new char[str.size\_ + 1])

{

for (size\_t i = 0; i < size\_; ++i)

{

str\_[i] = str.str\_[i];

}

str\_[size\_] = '\0';

}

String& operator=(const String& str)

{

if (str\_ == str.str\_)

{

return \*this;

}

delete[]str\_;

size\_ = str.size\_;

str\_ = new char[size\_ + 1];

for (size\_t i = 0; i < size\_; ++i)

{

str\_[i] = str.str\_[i];

}

str\_[size\_] = '\0';

return \*this;

}

size\_t getSize() const

{

return size\_;

}

~String()

{

delete []str\_;

str\_ = NULL;

size\_ = 0;

}

bool operator<(const String& str) const

{

auto s = std::min(str.size\_, size\_);

for (size\_t i = 0; i < s; ++i)

{

if (str\_[i] < str.str\_[i])

{

return true;

}

else

{

if (str\_[i] > str.str\_[i])

{

return false;

}

}

}

return s == size\_;

}

bool operator<=(const String& str) const

{

return \*this < str || \*this == str;

}

bool operator>(const String& str) const

{

return !(\*this <= str);

}

bool operator>=(const String& str) const

{

return !(\*this < str);

}

bool operator==(const String& str) const

{

if (str.size\_ != size\_)

{

return false;

}

for (size\_t i = 0; i < size\_; ++i)

{

if (str\_[i] != str.str\_[i])

{

return false;

}

}

return true;

}

bool operator!=(const String& str) const

{

return !(\*this == str);

}

friend std::ostream& operator<<(std::ostream& o, const String& str)

{

o << str.str\_;

return o;

}

};

10.cpp

#include <iostream>

#include "String.h"

#include "Node.h"

int main()

{

Tree<String> t;

t.add\_element("dsadsdf");

t.add\_element("bdvsvsd");

t.add\_element("advsvdscdscd");

t.add\_element("sdvsvdscdscd");

std::cout << t << std::endl;

const auto str = "bdvsvsd";

try

{

t.find(str);

std::cout << str << " is founded!" << std::endl;

}

catch (const std::exception& e)

{

std::cerr << str << " " << e.what() << std::endl;

}

return 0;

}