Файл TreeNode.h

#pragma once

template<class T>

class BinSTree;

// Объявление объекта для узла бинарного дерева

template <class T>

class TreeNode {

private:

// укзатели 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 = nullptr, TreeNode<T>\* rptr = nullptr) {

this->data = item;

this->left = lptr;

this->right = rptr;

}

//вернуть правый

TreeNode<T>\* Right() const {

return this->right;

}

//вернуть левый

TreeNode<T>\* Left() const {

return this->left;

}

friend class BinSTree<T>;

};

Файл BinSTree.h

#pragma once

#include <iostream>

#include "TreeNode.h"

template <class T>

class BinSTree {

private:

// текущий элемент дерева

TreeNode<T>\* current;

// корень дерева

TreeNode<T>\* root;

//число элементов дерева

int size;

public:

//текущий элемент

TreeNode<T>\* getCurrent() {

if (this->current != nullptr)

return this->current;

else {

std::cerr << "Current node is empty!";

exit(1);

}

}

//размер

int getSize()

{

return this->size;

}

//корень

TreeNode<T>\* getRoot() {

if (this->root != nullptr)

return this->root;

else {

std::cerr << "Root is empty!";

exit(1);

}

}

//распределение/освобождение памяти

TreeNode<T>\* getTreeNode(T item, TreeNode<T>\* lptr = nullptr, TreeNode<T>\* rptr = nullptr) {

TreeNode<T>\* p; // объвление нового узла

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

нового узла

if (p == nullptr) // если узел пустой то вернуть

ошибку

{

std::cerr << "Error while memory allocation!\n";

exit(1);

} // иначе вернуть ссылку на узел узел

return p;

}

// удаление узла

void deleteTreeNode(const T& item)

{

TreeNode<T>\* dNodePtr, \* pNodePtr, \* rNodePtr;

rNodePtr = nullptr;

dNodePtr = findNode(item, pNodePtr);

if (dNodePtr == nullptr)

return;

//если нет обоих потомков

if (dNodePtr->left == nullptr && dNodePtr->right == nullptr && pNodePtr != nullptr) {

if (dNodePtr->data < pNodePtr->data)

pNodePtr->left = rNodePtr;

else

pNodePtr->right = rNodePtr;

freeTreeNode(dNodePtr);

this->size--;

return;

}

//если есть только левый сын

else if (dNodePtr->right == nullptr)

rNodePtr = dNodePtr->left;

//если есть только правый сын

else if (dNodePtr->left == nullptr)

rNodePtr = dNodePtr->right;

//если есть оба потомка

else {

TreeNode<T>\* pOfRNodePtr = dNodePtr;

rNodePtr = dNodePtr->left;

while (rNodePtr->right != nullptr)

{

pOfRNodePtr = rNodePtr;

rNodePtr = rNodePtr->right;

}

if (pOfRNodePtr == dNodePtr)

rNodePtr->right = dNodePtr->right;

else

pOfRNodePtr->right = rNodePtr->left;

}

// если нечем заменить то это корневой узел удаляем его

if (rNodePtr == nullptr) {

this->root = rNodePtr;

}

else if (pNodePtr != nullptr && dNodePtr->data < pNodePtr->data)

pNodePtr->left = rNodePtr;

else if (pNodePtr != nullptr && dNodePtr->data >= pNodePtr->data)

pNodePtr->right = rNodePtr;

else if(dNodePtr->data > rNodePtr->data) {

rNodePtr->right = dNodePtr->right;

this->root = rNodePtr;

}

else {

rNodePtr->left = dNodePtr->left;

this->root = rNodePtr;

}

freeTreeNode(dNodePtr);

this->size--;

}

//перегрузка оператора =

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

{

// нельзя копировать дерево в само себя

if (this == &rhs)

return \*this;

// очистить текущее дерево. скопировать новое дерево в текущее объект

clearList(this->root);

this->root = copyTree(rhs->root);

// присвоить текущему указателю значению корня и задать размер дерева

this->current = this->root;

this->size = rhs->size;

// возвратить ссылку на текущий объект

return \*this;

}

//очищение

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

if (t != nullptr)

{

clearList(t->left);

clearList(t->right);

freeTreeNode(t);

}

}

//копирование

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

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

if (t == nullptr)

return nullptr;

if (t->left != nullptr)

newlptr = copyTree(t->left);

else

newlptr = nullptr;

if (t->right != nullptr)

newrptr = copyTree(t->right);

else

newrptr = nullptr;

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

return newnode;

}

// вставить item в дерево поиска

void insert(const T& item)

{

//t -текущий узел, parent - предыдуший узел

TreeNode<T>\* t = root, \* parent = nullptr, \* newNode = new TreeNode<T>(item);

// закончить на пустом дереве

while (t != nullptr)

{

// обновить тещущий указатель

parent = t;

if (item < t->data)

t = t->left;

else

t = t->right;

}

// если родитель пустой, то в ставить в корень

if (parent == nullptr)

root = newNode;

// если item меньше родительскоог

else if (item < parent->data)

parent->left = newNode;

else

parent->right = newNode;

current = newNode;

size++;

}

//освободить узел

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

delete p;

}

//поиск узла

TreeNode<T>\* findNode(const T& item, TreeNode<T>\* parent = nullptr) const {

TreeNode<T>\* t = this->root;

parent = nullptr;

while (t != nullptr)

{

if (item == t->data)

break;

else

{

parent = t;

if (item < t->data)

t = t->left;

else

t = t->right;

}

}

return t;

}

BinSTree() {

this->current = nullptr;

this->size = 0;

this->root = nullptr;

}

BinSTree(TreeNode<T>\* root) {

this->current = root;

this->size = 1;

this->root = root;

}

BinSTree(const T &item) {

this->root = new TreeNode<T>(item);

this->current = root;

this->size = 1;

}

};

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:

//указатель на стек

T\* stacklist;

//размер массива

const int size;

//индекс вершины

int top;

public:

//по умолчанию размер стека равен 10 элементам

MyStack(int = 10);

//конструктор копирования

MyStack(const MyStack<T>&);

//деструктор

~MyStack();

//показать n-й элемент от вершины стека

const T& peek(int) const;

//проверка на пустой стек

bool isEmpty() const;

//получить размер стека

int getSize() const;

//получить указатель на стек

T\* getPtr() const;

//получить номер текущего элемента в стеке

int getTop() const;

//поместить элемент в вершину стека

void push(const T&);

//удалить элемент из вершины стека и вернуть его

T pop();

//вывод стека на экран

void printStack();

//очистка стека

void clear();

};

//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:

//флаг, показывающий, достиг ли итератор концв списка

//должен поддерживаться производными классами

int iterationComplete;

public:

//конструктор

Iterator()

{

iterationComplete = 0;

}

//обязательные методы итератора

virtual void Next() = 0;

virtual void Reset() = 0;

//методы выборки/модификации данных

virtual T& Data() = 0;

//проверка конца списка

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 = 100);

//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) {

//дейсвия как у конструктора

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

this->S.clear(); //очищаем стек

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

}

//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 KeyValue.h

#pragma once

template <class K, class T>

class KeyValue

{

protected:

K key;

public:

// словарные данные являются общедоступными

T value;

KeyValue(K keyValue, T dataValue);

KeyValue() {

}

// операторы присваивания. не изменяют ключ

KeyValue<K, T>& operator= (const KeyValue<K, T>& rhs);

// операторы сравнения. сравнивают два ключа

int operator== (const KeyValue<K, T>& value) const;

int operator== (const K& keyval) const;

int operator< (const KeyValue<K, T>& value) const;

int operator< (const K& keyval) const;

// метод доступа к ключу

K Key() const;

};

template <class K, class T>

KeyValue<K, T>::KeyValue(K keyValue, T datavalue) : key(keyValue), value(datavalue) {};

template <class K, class T>

KeyValue<K, T>& KeyValue<K, T>::operator= (const KeyValue<K, T>& rhs) {

if (key != rhs.key) {

key = rhs.key;

value = rhs.value;

return \*this;

}

else return \*this;

}

template <class K, class T>

int KeyValue<K, T>::operator== (const KeyValue<K, T>& value) const {

return this->key == value.key;

}

template <class K, class T>

int KeyValue<K, T>::operator== (const K& keyval) const {

return this->key == keyval;

}

template <class K, class T>

int KeyValue<K, T>::operator< (const KeyValue<K, T>&value) const {

return this->key < value.key;

}

template <class K, class T>

int KeyValue<K, T>::operator< (const K& keyval) const {

return this->key < keyval;

}

template <class K, class T>

K KeyValue<K, T>::Key() const {

return key;

}

Файл Dictionary.h

#pragma once

#include "BinSTree.h"

#include "KeyValue.h"

template <class K, class T>

class Dictionary : public BinSTree<KeyValue<K,T>>

{

private:

//значение, присваиваемое эдементу словаря по умолчанию

//используется оператором индеусирования, а также методами

//InDictionary и DeleteKey

T defaultValue;

public:

//конструктор

Dictionary(const T& defaultval) : BinSTree < KeyValue<K, T>>(), defaultValue(defaultval) {};

//оператор индексирования

T& operator[] (const K& index);

///////////////дополнительные словарные методы//////////////

//проверяет наличие в словаре пары ключ-значение с ключем keyval

int InDictionary(const K& keyval);

//удаляет словарную статью, имеющую ключ keyval

void DeleteKey(const K& keyval);

};

//оператор индексированию. здесь делается почти вся работа

template <class K, class T>

T& Dictionary<K, T>::operator[] (const K& index) {

//определить целевой объект типа KeyValue, содержащий

//данные задаваемые по умолчанию

KeyValue<K, T> targetKey(index, defaultValue);

//искать ключ. если найден, вставить targetKey

if (!this->findNode(targetKey))

this->insert(targetKey);

//возвратить ссылку на найденные и всталенные данные

return this->getCurrent()->data.value;

}

//проверить существует ли объект типа KeyValue

//с данным ключем

template <class K, class T>

int Dictionary<K, T>::InDictionary(const K& keyval) {

//определить целевой объект типа KeyValue, содержащий

//данные задаваемые по умолчанию

KeyValue<K, T> tmp(keyval, defaultValue);

//искать tmp на дереве. вернуть результат

return !this->findNode(tmp);

}

//удалить объект типа KeyValue с данным ключем из словаря

template <class K, class T>

void Dictionary<K, T>::DeleteKey(const K& keyval) {

KeyValue<K, T> tmp(keyval, defaultValue);

this->deleteTreeNode(tmp);

}

Файл DictionaryIterator.h

#pragma once

#include "InorderIterator.h"

#include "Dictionary.h"

#include "KeyValue.h"

template<class K, class T>

class DictionaryIterator : public InorderIterator<KeyValue<K,T>>

{

public:

//конструктор

DictionaryIterator(Dictionary<K,T>& dict) : InorderIterator<KeyValue<K, T>>(dict.getRoot()) {}

//начать итерацию нового словаря

void SetList(Dictionary<K, T>& dict) {

this->SetTree(dict.GetRoot());

}

};

Файл lab15.cpp

#include <iostream>

#include "Dictionary.h"

#include "DictionaryIterator.h"

#include "KeyValue.h"

#include <fstream>

using namespace std;

//

string remove(string str, int i, int j)

{

int k = 1;

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

{

if (j + k > str.length())

{

while (i <= j) {

str.pop\_back();

i++;

}

}

else {

str[i] = str[j + k];

k++;

}

}

return str;

}

//распечатать объект KeyValue, содержащий ключевое слово word

void PrintEntry(const KeyValue<string, string>& word)

{

KeyValue<string, string> w = word;

cout << w.Key() << " - " << w.value;

}

//строка из опредедения

int ReadString(string& str, istream& istr, char delimiter = '.') {

char tmp[256];

int size;

if (istr.getline(tmp, 256, delimiter)) {

size = strlen(tmp);

str = tmp;

return size;

}

else

return -1;

}

int main()

{

setlocale(0, "ru");

//входной поток данных

ifstream fin;

string word, definition;

//словарь

Dictionary<string, string> wordDictionary("");

//открыть файл defs.dat ключевых слов и их дефиниций

fin.open("defs.dat", ios::in | ios::\_Nocreate);

if (!fin) {

cerr << "Файл defs.dat не найден" << endl;

exit(1);

}

//прочитать слово и его дефиницию. с помощью оператора индексирования

//включить статью в словарь или обновить существующую дефеницию,

//дополнив ее текущей

while (fin >> word) {

if (fin.eof())

break;

//прочитать пробел, следующий за ключевым словом

ReadString(definition, fin);

wordDictionary[word] += definition;

}

//обновить итератор дли нисходящего обхода словаря

DictionaryIterator<string, string> dictIter(wordDictionary);

//просматривать словарь. распечатать каждое ключевое слово

//и его дефиницию

cout << "Тольковый словарь:" << endl << endl;

for (dictIter.Reset(); !dictIter.EndOfList(); dictIter.Next())

{

PrintEntry(dictIter.Data());

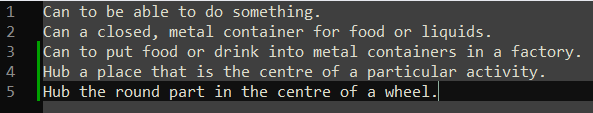
cout << endl;

}

wordDictionary.clearList(wordDictionary.getRoot());

}

Файл defs.dat



Файл Результат

