Файл Node.h

#pragma once

template <typename T>

class Node // шаблонный класс узла списка

{

private:

Node\* next; // указатель на след. узел

public:

T data; // данные узла списка

// конструктор

Node(const T& item, Node<T>\* ptrnext = nullptr);

// методы модификации списка

void InsertAfter(Node<T>\* p);

Node<T>\* DeleteAfter();

// получение адреса следующего узла

Node<T>\* NextNode() const;

};

template <class T>

Node<T>::Node(const T& item, Node<T>\* ptrnext) : data{ item }, next{ ptrnext } {};

// возвратить заркытый член next

template <class T>

Node<T>\* Node<T>::NextNode() const {

return next;

}

// вставить узел p после текущего узла

template <class T>

void Node<T>::InsertAfter(Node<T>\* p) {

// p указывает на следующий за текущим узел

// а текущий узел - на p

p->next = next;

next = p;

}

// удалить узел, следующий за текущим, и возвратить его адрес

template <class T>

Node<T>\* Node<T>::DeleteAfter()

{

// сохранить адрес удаляемого узла

Node<T>\* tempPtr = next;

// если нет следующего узла, возвратить NULL

if (next == nullptr)

return nullptr;

// текущий узел указывает на узел, следующий за tempPtr.

next = tempPtr->next;

// возвратить указатель на несвязанный узел

return tempPtr;

}

Файл List.h

#pragma once

template<class T>

class List

{

//число элементов списка, обновляемое производным классом

int size;

public:

//конструктор устанавливает size в 0

List()

{

this->size = 0;

}

//веврнуть размер списка

int ListSize() {

return this->size;

}

//проверить, пуст ли список

int ListEmpty() {

return this->size == 0;

}

//найти

virtual int Find(T& item) = 0;

//вставить элемент

virtual void Insert(const T& item) = 0;

//удалить элемент

virtual void Delete(const T& item) = 0;

//очистить список

virtual void ClearList() = 0;

};

Файл Array.h

#pragma once

# include <iostream>

# include <stdlib.h>

enum ErrorType

{

invalidArraysize, memoryAllocationError, indexOutOfRange

};

const char\* errorMsg[] = {

"Неверный разыер массива", "Ошибка выделения nамяти", "Неверный индекс: "

};

// шаблонный класс массива

template <class T>

class Array {

private:

// динаыически выделяемый сnисок размером size

T\* alist;

int size;

// метод обработки ошибок

void Error(ErrorType error, int badIndex = 0) const;

public:

// конструктор

Array(int sz = 50);

// конструктор копирования

Array(const Array<T>& А);

// деструктор

~Array();

// перегрузка оператора присваивания

Array<T>& operator = (const Array<T>& rhs);

// перегрузка индексного оператора

T& operator [] (int i);

// перегрузка указателя

operator T\* () const;

// операции с размероы ыассива

int ListSize() const; // читать size

void Resize(int sz); // обновлять size

};

// метод обработки ошибок

template<class T>

void Array<T>::Error(ErrorType error, int badIndex) const {

if (badIndex != 0) {

std::cerr << errorMsg[error] << std::endl;

exit(1);

}

else {

std::cerr << errorMsg[error] << badIndex << std::endl;

exit(1);

}

}

// конструктор

template<class T>

Array<T>::Array(int sz) {

// проверка на наличие параметра неверного размера

if (sz <= 0)

Error(invalidArraysize);

// nрисваивание размера и выделение памяти

size = sz;

alist = new T[size];

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

alist[i] = T();

// убеждаемся в том, что система выделяет необходимую память,

if (alist == nullptr)

Error(memoryAllocationError);

}

// деструктор

template<class T>

Array<T>::~Array() {

delete[] alist;

}

// конструктор копирования

template<class T>

Array<T>::Array(const Array<T>& X) {

// получить разwер объекта Х и присвоить текущеwу объекту

int n = X.size;

size = n;

// выделить новую память дпя объекта с проверкой

// возможных ошибок

alist = new T[n];

if (alist == nullptr)

Error(memoryAllocationError);

// колировать элементы массива объекта Х в текупо,�й объект

T\* srcptr = X.alist; // адрес начала X.aliзt

T\* destptr = alist; // адрес начала alist

while (n--) // копировать список

\*destptr++ = \*srcptr++;

}

// перегрузка оператора присваивания

template<class T>

Array<T>& Array<T>::operator=(const Array<T>& rhs)

{

// получить разwер объекта Х и присвоить текущеwу объекту

int n = rhs.size;

size = n;

// выделить новую память дпя объекта с проверкой

// возможных ошибок

alist = new T[n];

if (alist == nullptr)

Error(memoryAllocationError);

// колировать элементы массива объекта Х в текупо,�й объект

T\* srcptr = rhs.alist; // адрес начала X.aliзt

T\* destptr = alist; // адрес начала alist

while (n--) // копировать список

\*destptr++ = \*srcptr++;

}

// перегрузка индексного оператора

template<class T>

T& Array<T>::operator [] (int n) {

// аыnолнение проверки границ массива

if (n < 0 && n > size - 1)

Error(indexOutOfRange, n);

// возвращается элемент из закрытого списка массива

return alist[n];

}

// перегрузка индексного оператора

template<class T>

Array<T>::operator T\* () const {

// возвращает адрес закрытого массива в текущем оОъекте

return alist;

}

// читать size

template<class T>

int Array<T>::ListSize() const {

return size;

}

// обновлять size

template<class T>

void Array<T>::Resize(int sz) {

// проверка нового размера массива;

// выход из программы при sz <= 0

if (sz <= 0)

Error(invalidArraysize);

// ничего не делать, если размер не изменился

if (sz == size)

return;

// запросить память для нового массива и проверить ответ систеNЫ

T\* newlist = new T[sz];

if (newlist == nullptr)

Error(memoryAllocationError);

// ооъявить n и инициализировать значением sz или зize

int n =(sz <= size) ? sz : size;

// колировать n элементов массива из старой в новую память

T\* srcptr = alist; // адрес начала alist

T\* destptr = newlist; // адрес начала newliзt

while (n--) // колировать список

\*destptr++ = \*srcptr++;

// удалить старый список

delete[] alist;

// переустановить alist, чтобы он указывал на newlist и обновить член класса size

alist = newlist;

size = sz;

}

Файл LinkedList.h

#pragma once

#include <iostream>

#include <stdlib.h>

#include "Node.h"

using namespace std;

template <class T>

class LinkedList

{

private:

// указатели для доступа к началу и концу списка

Node<T>\* front, \* rear;

// используются для извлечения, вставки и удаления данных

Node<T>\* prevPtr, \* currPtr;

// число элементов в списке

int size;

// положение в списке. используется методом Reset

int position;

// создание узла

Node<T>\* GetNode(const T& item, Node<T>\* ptrNext = nullptr);

// очистка узла

void FreeNode(Node<T>\* pl);

// колирует список L в текущий список

void CopyList(const LinkedList<T>& L);

public:

// конструктор

LinkedList();

// конструктор

LinkedList(const LinkedList<T>& L);

// деструктор

~LinkedList();

// проверка состояния списка

int ListSize() const;

// проверка на пустоту

int ListEmpty() const;

// установление position в положение pos

void Reset(int pos = 0);

// переход к следующему узлу

void Next();

// проверка, дошли ли до конца списка

bool EndOfList() const;

// возврат позиции

int CurrentPosition() const;

// вставить вначале

void InsertFront(const T& item);

// вставить в конец

void InsertRear(const T& item);

// вставить на какую-то позицию

void InsertAt(const T& item);

//вставить после

void InsertAfter(const T& item);

// методы удаления

T DeleteFront();

// удаление элемента

void DeleteAt();

// возвратить/изменить данные

T& Data();

// очистка списка

void ClearList();

LinkedList<T>& operator = (const LinkedList<T>& L);

};

// создание узла

template <class T>

Node<T>\* LinkedList<T>::GetNode(const T& item, Node<T>\* ptrNext) {

Node<T>\* temp = new Node<T>(item, ptrNext);

return temp;

};

// очистка узла

template <class T>

void LinkedList<T>::FreeNode(Node<T>\* pl) {

delete pl;

};

// колирует список L в текущий список

template <class T>

void LinkedList<T>::CopyList(const LinkedList<T>& L) {

// р - указатель на L

Node<T>\* p = L.front;

int pos;

// вставлять каждый элемент из L в конец текущего объекта

while (p != nullptr) {

InsertRear(p->data);

p = p->NextNode();

}

// выход, если список пустой

if (position == -1)

return;

// nереустановить prevPtr и currPtr в новом списке

prevPtr = nullptr;

currPtr = front;

for (pos = 0; pos != position; pos++) {

prevPtr = currPtr;

currPtr = currPtr->NextNode();

}

};

// конструктор

template <class T>

LinkedList<T>::LinkedList() {

front = nullptr;

rear = nullptr;

prevPtr = nullptr;

currPtr = nullptr;

size = 0;

position = 0;

};

// конструктор

template <class T>

LinkedList<T>::LinkedList(const LinkedList<T>& L) {

CopyList(L);

};

// деструктор

template <class T>

LinkedList<T>::~LinkedList(void) {

delete front;

delete rear;

};

// оператор присваивания

template <class T>

LinkedList<T>& LinkedList<T>::operator = (const LinkedList<T>& L) {

CopyList(L); //копируем список из входного

return \*this; //возвращаем этот список

};

// проверка состояния списка

template <class T>

int LinkedList<T>::ListSize(void) const {

return size;

};

// проверка на пустоту

template <class T>

int LinkedList<T>::ListEmpty(void) const {

return size == 0;

};

// установление position в положение pos

template <class T>

void LinkedList<T>::Reset(int pos) {

int startPos;

// если список пустой, выход

if (front == nullptr)

return;

// если положение задано не верно, закончить программу

if (pos < 0 || pos > size - 1)

{

cerr << "Reset: Неверно задано положение: " << pos << std::endl;

return;

}

// установить механизм nрохоЖдения в pos

if (pos == 0) {

// перейти в начало списка

prevPtr = nullptr;

currPtr = front;

position = 0;

}

else

// nереустановить currPtr, prevPtr, и startPos

{

currPtr = front->NextNode();

prevPtr = front;

startPos = 1;

// передвигаться вправо до pos

for (position = startPos; position != pos; position++) {

// передвинуть оба указателя прохождения вперед

prevPtr = currPtr;

currPtr = currPtr->NextNode();

}

}

};

// переход к следующему узлу

template <class T>

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

// выйти, если конец списка или // список пустой

if (currPtr != nullptr) {

// nереустановить два указателя на один узел вперед

prevPtr = currPtr;

currPtr = currPtr->NextNode();

position++;

}

};

// проверка, дошли ли до конца списка

template <class T>

bool LinkedList<T>::EndOfList() const {

return position == size;

};

// возврат позиции

template <class T>

int LinkedList<T>::CurrentPosition(void) const {

return position;

};

// вставить вначале

template<typename T>

void LinkedList<T>::InsertFront(const T& item) {

Node<T>\* newNode; //новый узел

newNode = GetNode(item, front); //передаём в него значение и адрес начала

front = newNode; //указываем на новый узел

currPtr = front; //обновляем текущий указатель

position = 0; //обновляем позицию

size++; //увеличиваем размер

}

// вставить в конец

template<typename T>

void LinkedList<T>::InsertRear(const T& item) {

Node<T>\* newNode; //новый узел

this->currPtr = this->front;

this->position = 0;

//если список пуст, вставить item в начало

if (this->currPtr == nullptr)

this->InsertFront(item);

else {

//найти узел с нулевым указателем

while (this->currPtr->NextNode() != nullptr)

this->currPtr = this->currPtr->NextNode();

// создать узел и вставить в конец списка (после currPtr)

newNode = this->GetNode(item, this->rear);

this->currPtr->InsertAfter(newNode);

this->size++;

}

}

// вставить на какую-то позицию

template <class T>

void LinkedList<T>::InsertAt(const T& item) {

Node<T>\* newNode;

// два случая: вставка в начало или внутрь списка

if (prevPtr == nullptr) {

// вставка в начало списка. nоыещает также узел в пустой список

newNode = GetNode(item, front);

front = newNode;

}

else {

// вставка внутрь списка. помещает узел после prevPtr

newNode = GetNode(item);

prevPtr->InsertAfter(newNode);

}

// при prevPtr == rear, имеем вставку в пустой список

// или в хвост нелустого списка; обновляет rear и positioп

if (prevPtr == rear) {

rear = newNode;

position = size;

}

// обновить currPtr и увеличить size

currPtr = newNode;

size++;

};

//вставить после

template<typename T>

void LinkedList<T>::InsertAfter(const T& item) {

Node<T>\* newNode; //новый узел

if (this->front == nullptr) { //если список пуст

//вставляем его в начало

newNode = this->getNode(item, this->front);

this->front = newNode;

}

else {

if (this->currPtr == this->rear) { //если это хвостовой узел

//вставляем в конец списка

newNode = this->getNode(item, this->rear);

this->prev\_ptr->insAfter(newNode);

}

else {

//вставляем после текущего

newNode = this->getNode(item, currPtr->nextNode());

this->currPtr->insAfter(newNode);

}

}

this->next(); //смещаем позицию

this->size++;

}

//удаление

template <class T>

T LinkedList<T>::DeleteFront() {

Node<T>\* p;

T tmp = nullptr;

// ошибка, если список пустой или конец списка

if (currPtr == nullptr) {

cerr << "Ошибка удаления!" << std::endl;

exit(1);

}

if (front != nullptr)

{

p = front;

front = front->NextNode();

tmp = p->data;

}

FreeNode(p);

return tmp;

};

// удаление элемента

template <class T>

void LinkedList<T>::DeleteAt(void) {

Node<T>\* p;

// ошибка, если список пустой или конец списка

if (currPtr == nullptr) {

cerr << "Ошибка удаления!" << std::endl;

exit(1);

}

// удалять можно только в начале и внутри списка

if (prevPtr == nullptr) {

// сохранить адрес начала, но не связывать его. если это - последний узел, присвоить front значение NULL

p = front;

front = front->NextNode();

}

else {

// не связывать внутренний узел после prevPtr, запомнить адрес

p = prevPtr->DeleteAfter();

}

// если хвост удален, адрес нового хвоста в prevPtr, // а position уыеньwается на 1

if (p == rear) {

rear = prevPtr;

position--;

}

// установить currPtr на последний удаленный узел. Если р - последний узел в списке, currPtr становится равным NULL

currPtr = p->NextNode();

// освободить узел и уменьшить значение size

FreeNode(p);

size--;

};

// возвратить/изменить данные

template <class T>

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

// ошибка, если список пустой или прохождение закончено

if (size == 0 || currPtr == nullptr) {

cerr << "Data: Неверная ссылка!" << std::endl;

exit(1);

}

return currPtr->data;

};

// очистка списка template <class T>

void LinkedList<T>::ClearList() {

Node<T>\* currPosition, \* nextPosition;

currPosition = front;

while (currPosition != nullptr) {

// получить адрес следующего узла и удалить текущий

nextPosition = currPosition->NextNode();

FreeNode(currPosition);

currPosition = nextPosition; // перейти к следующему узлу

}

front = rear = nullptr;

prevPtr = currPtr = nullptr;

size = 0;

position = -1;

};

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 HashTable.h

#pragma once

#include "Iterator.h"

#include "List.h"

#include "Array.h"

#include "LinkedList.h"

#include <iostream>

template<class T>

class HashTableIterator;

template <class T>

class HashTable : public List<T>

{

protected:

// число болков; предаставляют размер таблицы

int numBuckets;

//хеш-таблица есть массив связанных списков

Array<LinkedList<T>> buckets;

// хеш функция и адрес элемента данных

// к которому обращались последний раз

unsigned long (\*hf)(T);

T\* current;

public:

// конструтор копирования

HashTable(HashTable<T> &hashTable) {

numBuckets = hashTable.numBuckets;

buckets = hashTable.buckets;

hf = hashTable.hf;

}

// конструктор с параметрами, включающими

// размер таблицы и хеш-функцию

HashTable(int nbucket, unsigned long (\*hashf)(T));

// методы обработки списков

virtual void Insert(const T& key);

virtual int Find(const T& key);

virtual void Delete(const T& key);

virtual void ClearList();

void Update(const T& key);

//дружественный итератор, имеющий доступ к

// данным-членам

friend class HashTableIterator<T>;

};

template<class T>

HashTable<T>::HashTable(int nbucket, unsigned long (\*hashf)(T)) : List<T>(), numBuckets{ nbucket }, hf{ hashf }, buckets{ nbucket }{

current = NULL;

}

//вычисляет значение хэш-функции (индекс блока) и ищет объект типа LinkedList,

//чтобы проверить, есть ли уже такой элемант данных в таблице или нет

template<class T>

void HashTable<T>::Insert(const T& key) {

// hashval - индекс блока (связанного списка)

int hashval = int(hf(key) % numBuckets);

// lst - псведоним для buckets[hashval].

// помогает обойтись без индексов

LinkedList<T>& lst = buckets[hashval];

if (!lst.ListEmpty() ) {

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

// если ключ совпал, обновить данные и выйти

if (lst.Data() == key)

{

current = &lst.Data();

return;

}

}

lst.InsertRear(key);

current = &lst.Data();

}

//применяет хеш-функциюи просматривает указанный в результате список на предмет

//совпадения с входным параметром

template <class T>

int HashTable<T>::Find(const T& key)

{

// вычислить значение хеш-функции и установить lst

// на начало соотвествующего связанного списка

int hashval = int(hf(key) % numBuckets);

LinkedList<T>& lst = buckets[hashval];

// просматривать узлы связаного списка в поиска key

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

if (lst.Data() == key)

{

current = &lst.Data();

return 1; // вернуть true

}

return 0; // иначе вернуть false

}

//просматривает указанный список и удаляет узел, если совпадение произошло

template <class T>

void HashTable<T>::Delete(const T& key)

{

int hashval = int(hf(key) % numBuckets);

LinkedList<T>& lst = buckets[hashval];

lst.Reset();

while (!lst.EndOfList()) {

if (lst.Data() == key) {

lst.DeleteAt();

return;

}

else

lst.Next();

}

cout << "Елемента нет в списке!";

}

// очищение списка по ключу одного из элементов списка

template<class T>

void HashTable<T>::ClearList() {

int hashval = int(hf(\*current) % numBuckets);

LinkedList<T>& lst = buckets[hashval]; //лист блока

lst.ClearList(); // очищение листа

}

//обновление

template <class T>

void HashTable<T>::Update(const T& key) {

int hashval = int(hf(key) % numBuckets); // индекс блока

LinkedList<T>& lst = buckets[hashval]; //лист блока

for (lst.Reset(0); !lst.EndOfList(); lst.Next())

{

if (lst.Data() == key)

{

current = &lst.Data();

lst.Data() = key;

return;

}

}

}

Файл HashTableIterator.h

#pragma once

#include "HashTable.h"

#include "Iterator.h"

template<class T>

class HashTableIterator : public Iterator<T>

{

private:

// указатель таблицы, подлежащий обходу

HashTable<T>\* hashTable;

// индекс текущего просматриваемого блока

// и указатель на связанный список

int currentBucket;

LinkedList<T>\* currBucketPtr;

// утилита для реализации метода Next

void SearchNextNode(int cb);

public:

// конструктор

HashTableIterator(HashTable<T>& hf);

// базоыве методы итератора

virtual void Next(void);

virtual void Reset(void);

virtual T& Data(void);

// подготовить итератор для сканирования другой таблицы

void SetList(HashTable<T>& lst);

};

//обноружение очередного списка, подлежащего прохождению

template <class T>

void HashTableIterator<T>::SearchNextNode(int cb) {

currentBucket = -1;

// если индекс cb больше размера таблицы, прекратить поиск

if (cb > hashTable->numBuckets)

return;

// иначе искать, начиная с текущего списка до конца таблицы,

// непустой блок и обновить частные элементы данных

for (int i = cb; i < hashTable->numBuckets; i++) {

if (!hashTable->buckets[i].ListEmpty()) {

// перед те как вернуться, установить currentBucket равным 1

// и в currBucketPtr поместить адрес нового непутого списка

currBucketPtr = &hashTable->buckets[i];

currBucketPtr->Reset();

currentBucket = i;

return;

}

}

}

//продвижение вперед по текущему списку на один элемент

template<class T>

void HashTableIterator<T>::Next() {

// продвинуться к следующему узлу текущего списка

currBucketPtr->Next();

// по достижению конца списка вызвать SearchNextNode

// для поиска следующего непустого блока в таблице

if (currBucketPtr->EndOfList())

SearchNextNode(++currentBucket);

// установить флажок iterationComlpete, если непустых списков

// больше нет

this->iterationComplete = currentBucket == -1;

}

template<class T>

T& HashTableIterator<T>::Data() {

if (currBucketPtr == nullptr) {

cout << "Data: Ошибка указатель пустой";

exit(1);

}

return currBucketPtr->Data();

}

template<class T>

void HashTableIterator<T>::Reset() {

SearchNextNode(0);

}

template<class T>

void HashTableIterator<T>::SetList(HashTable<T>& lst) {

this->hashTable = lst;

}

// конструктор. инициализирует базовый класс и класс HashTable

// SearchNextNode идентифицирует первый непустой блок в таблице

template<class T>

HashTableIterator<T>::HashTableIterator(HashTable<T>& hf) : Iterator<T>(), hashTable{ &hf } {

SearchNextNode(0);

};

Файл lab12.cpp

#include <iostream>

#include <fstream>

#include "HashTable.h"

#include "HashTableIterator.h"

using namespace std;

struct NameRecord

{

string name;

int count;

NameRecord() {

name = "";

count = 0;

}

int operator ==(const NameRecord& rhs) {

return name == rhs.name;

}

};

unsigned long hash1(NameRecord elem)

{

unsigned long hashval = 0;

for (int i = 0; i < elem.name.length(); i++)

hashval = (hashval << 3) + elem.name[i];

return hashval;

};

int main()

{

//ввести символьные строки из входного потока

ifstream fin;

NameRecord rec;

string token;

HashTable<NameRecord> HF(101, hash1);

setlocale(0, "ru"); //руссификация

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

if (!fin) {

cerr << "Невозможно открыть \"strings.dat\"!" << endl;

exit(1);

}

while (fin>>rec.name)

{

//искать строку в таблице. если найдена, обновить поле count

if (HF.Find(rec)) {

rec.count += 1;

HF.Update(rec);

}

else

{

rec.count = 1;

HF.Insert(rec);

}

}

//печать символьных строк вместе с частотами

HashTableIterator<NameRecord> hiter(HF);

for (hiter.Reset(); !hiter.EndOfList(); hiter.Next()) {

rec = hiter.Data();

cout << rec.name << ": " << rec.count << endl;

}

}

Файл strings.dat



Результат

