**Постановка задачи**

Решить поставленную задачу с использованием языка программирования С++, системы контроля версий Git и средств непрерывной интеграции, предоставляемых GitLab.

**S3. Списки II**

1. Реализовать шаблон класса BidirectionalList и соответствующие итераторы

|  |  |
| --- | --- |
| 1 | template< typename T > |
| 2 | class BidirectionalList { |
| 3 | ... |
| 4 | }; |

Разрабатываемый интерфейс должен быть безопасным

1. Параметром командной строки задётся имя файла filename, который содержит внутри себя данные по некоторому количеству списков • Файл со списками имеет следующий вид:

<list-1> <value-1-1> <value-1-2> ...

<list-2> <value-2-2> <value-2-2> ...

...

* + - list представляет собой имя списка; value целочисленное значение - элемент списка. Например:

first 1 3 2 1

second 6 0 -1 1

* + - Пустые строки игнорируются

1. Реализуемая программа должна считывать списки, содержащиеся в файле, и обрабатывать команды, принимаемые со стандартного ввода от пользователя.
   * Каждая строка содержит ровно одну команду. Должны поддерживаться следующие команды:

print <list>

replace <destination-list> <value-1> <value-2>

replace <destination-list> <value-1> <source-list>

remove <destination-list> <value-1>

remove <destination-list> <source-list-2>

concat <new-list> <1st-list> <2nd-list> <3rd-list> ...

equal <1st-list> <2nd-list> <3rd-list> ...

* + Команда print <list> выводит данные списка с соответствующим именем. Например, для команды:

print first

Должен быть результат:

first 1 3 2 1

Если список пуст, то команда должна вывести сообщение <EMPTY>

* + Команда replace заменяет заданные элементы в списке. В качестве последнего параметра может быть передано значение или другой список. Например, для команд:

replace first 1 2

replace first 3 second

print first

Должен быть результат:

first 2 6 0 -1 1 2 2

* + Команда remove удаляет из списка заданные элементы. Удаляемые элементы могут быть заданы значением или другим списком. Например, для команд:

remove first 2

remove first second print first

Должен быть результат:

first 3

* + Команда concat созадёт новый список конкатенацией списков, перечисленных в качестве параметров. Например, для команд:

concat third first second first

concat yathird second first second first

print third print yathird

Должен быть результат:

third 1 3 2 1 6 0 -1 1 1 3 2 1

yathird 6 0 -1 1 1 3 2 1 6 0 -1 1 1 3 2 1

* + Команда equal сравнивает списки, переданные в качестве параметров. Например, для команды:
  + equal first first

Должен быть результат:

<TRUE>

А для команды:

equal first first second first

Должен быть результат:

<FALSE>

* + Если команда по каким-то причинам некорректна, то команда должна вывести сообщение <INVALID COMMAND>
  + Других команд реализовывать не требуется
  + Признаком конца ввода команд является EOF (на Linux: Ctrl + D | на Windows Ctrl + Z затем Enter)

1. Работа должна быть выполнена в виде 1-го исполняемого файла, принимающего параметры следующим образом:

$ ./lab filename

filename представляет собой обязательный параметр. Поведение программы меняется в зависимости от того передан он или нет. Если параметр filename не задан, программа должна завершаться с ненулевым кодом возврата и сообщением об ошибке

**Приёмочные тесты**

Средствами непрерывной интеграции GitLab осуществляется автоматическая проверка результатов работы программы. В рамках настоящей работы предусмотрены следующие интеграционные тесты:

|  |  |  |
| --- | --- | --- |
| # | **Описание** | **Результат** |
| 1 | 1. Concat Command With Empty Lists |  |
| 2 | Конкатенация с пустым списком | **Input**:  concat fourth second third first second  print fourth  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  fourth 6 0 -1 1 1 3 2 1 6 0 -1 1 |
| 3 | Конкатенация с недостаточным количеством аргументов | **Input**:  concat  concat fourth  concat fourth third  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <INVALID COMMAND>  <INVALID COMMAND>  <INVALID COMMAND>  first 1 3 2 1 |
| 4 | Сравнение разных списков | **Input**:  equal first second  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <FALSE> |
| 5 | Сравнение списков с одинаковым набором элементов | **Input**:  equal first second first second first second first second first second **Expected**:  **With data**:  first 1 3 2 1  second 1 3 2 1  third  **expected result is**:  <TRUE> |
| 6 | Сравнение списков с недостаточным количеством аргументов | **Input**:  equal  equal first  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <INVALID COMMAND>  <INVALID COMMAND>  first 1 3 2 1 |
| 7 | Сравнение списка с самим собой | **Input**:  equal first first first first first first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <TRUE> |
| 8 | Печать пустого списка | **Input**:  print third  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <EMPTY> |
| 9 | Печать с недостаточным количеством аргументов | **Input**:  print  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <INVALID COMMAND> |
| 10 | Удаление из списка с недостаточным количеством аргументов | **Input**:  remove  remove first  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <INVALID COMMAND>  <INVALID COMMAND>  first 1 3 2 1 |
| 11 | Удаление из списка с аргументом – названием списка | **Input**:  remove first second  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 3 -1 1  third  **expected result is**:  first 2 |
| 12 | Удаление из списка с аргументами – значениями | **Input**:  remove first 1  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  first 3 2 |
| 13 | Замена элементов списка другим списком | **Input**:  replace first 2 second  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  first 1 3 6 0 -1 1 1 |
| 14 | Замена элементов списка с недостаточным количеством аргументов | **Input**:  replace  replace first  replace first 2  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  <INVALID COMMAND>  <INVALID COMMAND>  <INVALID COMMAND>  first 1 3 2 1 |
| 15 | Замена элементов списка другим элементом | **Input**:  replace first 1 2  print first  **Expected**:  **With data**:  first 1 3 2 1  second 6 0 -1 1  third  **expected result is**:  first 2 3 2 2 |
| 16 | Пропуск пустых строк при считывании списков из файла | **Input**:  print first  **Expected**:  **With data**:    first 1 3 2 1    third  **expected result is**:  first 1 3 2 1 |
| 17 | Неожиданная команда | **Input**:  unexpected second  **Expected**:  **With data**:  second 6 0 -1 1  **expected result is**:  <INVALID COMMAND> |

**Исходные тексты программы**

Файлы с исходными текстами лабораторной работы располагаются в корне общего проекта (полагаем <ROOT> для папки локального репозитория)

**./<ROOT>/nikiforova.ekaterina/S3/main.cpp**

﻿#include <iostream>

#include <fstream>

#include <string>

#include <functional>

#include "bidirectionalList.h"

#include "dictionary.h"

#include "commandsWithDictsOfLists.h"

#include "errorMessages.h"

int main(int argc, char\*\* argv)

{

if (argc != 2)

{

std::cerr << "Incorrect number of arguments";

return 1;

}

std::ifstream fInput(argv[1]);

if (!fInput)

{

std::cerr << "File open error";

return 1;

}

try

{

nikiforova::Dictionary< std::string, nikiforova::BidirectionalList< long long > > dict = nikiforova::readAllLists(fInput);

fInput.close();

nikiforova::Dictionary< std::string, std::function< void(std::string&, nikiforova::DictOfLists&) > > commands

{

{"print", nikiforova::print},

{"replace", nikiforova::replace},

{"remove", nikiforova::remove},

{"concat", nikiforova::concat},

{"equal", nikiforova::equal}

};

while (!std::cin.eof())

{

std::string command = "";

std::cin >> command;

if (command.empty())

{

continue;

}

auto iter = commands.find(command);

if (iter == commands.end())

{

std::string temp = "";

std::getline(std::cin, temp);

nikiforova::invalidCommandMessage(std::cout);

}

else

{

std::string str = "";

std::getline(std::cin, str);

iter->second(str, dict);

}

}

}

catch (const std::exception& e)

{

std::cerr << e.what();

}

return 0;

}

**./<ROOT>/nikiforova.ekaterina/S3/commandsWithDictsOfLists.h**

#ifndef COMMANDSWITHDICTSOFLISTS\_H

#define COMMANDSWITHDICTSOFLISTS\_H

#include "dictionary.h"

#include "bidirectionalList.h"

namespace nikiforova {

using DictOfLists = Dictionary< std::string, BidirectionalList< long long > >;

DictOfLists readAllLists(std::istream&);

BidirectionalList< long long > convertStringToList(std::string&);

void print(std::string&, DictOfLists&);

void replace(std::string&, DictOfLists&);

void remove(std::string&, DictOfLists&);

void concat(std::string&, DictOfLists&);

void equal(std::string&, DictOfLists&);

std::ostream& doPrint(std::ostream&, std::string&, DictOfLists);

void doReplace(const std::string&, long long, long long, DictOfLists&);

void doReplace(const std::string&, long long, const std::string&, DictOfLists&);

void doRemove(const std::string&, long long, DictOfLists&);

void doRemove(const std::string&, const std::string&, DictOfLists&);

std::ostream& doEqual(std::ostream&, std::string&, DictOfLists&);

}

#endif

**./<ROOT>/nikiforova.ekaterina/S3/commandsWithDictsOfLists.cpp**

#include "commandsWithDictsOfLists.h"

#include <string>

#include "errorMessages.h"

#include "operationsWithStrings.h"

nikiforova::DictOfLists nikiforova::readAllLists(std::istream& in)

{

DictOfLists result;

while (!in.eof())

{

std::string nameOfList = "";

in >> nameOfList;

DictOfLists::ConstIterator cIter = result.find(nameOfList);

if (cIter != result.cend())

{

throw std::logic\_error("List with the same name already exists");

}

else

{

std::string str = "";

std::getline(in, str);

nikiforova::BidirectionalList< long long > temp = nikiforova::convertStringToList(str);

result.push(nameOfList, temp);

}

}

return result;

}

nikiforova::BidirectionalList< long long > nikiforova::convertStringToList(std::string& str)

{

BidirectionalList< long long > result;

while (!str.empty())

{

long long value = 0;

std::string temp = str.substr(1, str.find(" ", 1) - 1);

if (!nikiforova::isNumber(temp))

{

throw std::logic\_error("Value is not a number");

}

value = std::stoll(temp);

str = str.erase(0, temp.length() + 1);

result.pushBack(value);

}

return result;

}

void nikiforova::print(std::string&str , DictOfLists& dict)

{

nikiforova::doPrint(std::cout, str, dict);

}

std::ostream& nikiforova::doPrint(std::ostream& out, std::string& str, DictOfLists dict)

{

std::string nameOfList = nikiforova::getWord(str);

if (nameOfList.empty())

{

return nikiforova::invalidCommandMessage(out);

}

nikiforova::BidirectionalList< long long > newList;

DictOfLists::ConstIterator cIter = dict.find(nameOfList);

if (cIter == dict.cend())

{

return nikiforova::invalidCommandMessage(out);

}

nikiforova::BidirectionalList< long long > temp = cIter->second;

if (temp.isEmpty())

{

return nikiforova::emptyMessage(out);

}

out << nameOfList;

for (auto&& pair: temp)

{

out << " " << pair;

}

out << "\n";

return out;

}

void nikiforova::replace(std::string& str, DictOfLists& dict)

{

int countArgs = 0;

std::string nameOfList = nikiforova::getWord(str);

if (!nameOfList.empty())

{

countArgs++;

}

nikiforova::DictOfLists::ConstIterator cIter = dict.find(nameOfList);

if (cIter == dict.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::BidirectionalList< long long > newList;

std::string arg1 = "";

std::string arg2 = "";

long long value = 0;

if (!str.empty())

{

arg1 = nikiforova::getWord(str);

countArgs++;

if (!nikiforova::isNumber(arg1))

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

value = std::stoll(arg1);

if (!str.empty())

{

arg2 = nikiforova::getWord(str);

countArgs++;

}

}

if (countArgs != 3)

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

if (nikiforova::isNumber(arg2))

{

long long newValue = std::stoll(arg2);

nikiforova::doReplace(nameOfList, value, newValue, dict);

}

else

{

nikiforova::doReplace(nameOfList, value, arg2, dict);

}

}

void nikiforova::doReplace(const std::string& nameOfList, long long value, long long newValue, DictOfLists& dict)

{

nikiforova::BidirectionalList< long long > list = dict.get(nameOfList);

for (auto&& item: list)

{

if (item == value)

{

item = newValue;

}

}

dict.find(nameOfList)->second = list;

}

void nikiforova::doReplace(const std::string& nameOfList, long long value, const std::string& sourceList, DictOfLists& dict)

{

nikiforova::DictOfLists::ConstIterator cIter = dict.find(sourceList);

if (cIter == dict.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::BidirectionalList< long long > destList = dict.find(nameOfList)->second;

nikiforova::BidirectionalList< long long >::Iterator destIter = destList.begin();

while (destIter != destList.end())

{

if (\*destIter == value)

{

destIter = destList.erase(destIter);

nikiforova::BidirectionalList< long long >::Iterator sourceIter = dict.find(sourceList)->second.begin();

while (sourceIter != dict.find(sourceList)->second.end())

{

destList.insert(\*sourceIter, destIter);

++sourceIter;

}

}

++destIter;

}

dict.find(nameOfList)->second = destList;

}

void nikiforova::remove(std::string& str, DictOfLists& dict)

{

int countArgs = 0;

std::string nameOfList = nikiforova::getWord(str);

if (!nameOfList.empty())

{

countArgs++;

}

std::string arg = "";

if (!str.empty())

{

arg = nikiforova::getWord(str);

countArgs++;

}

if (countArgs != 2)

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::DictOfLists::ConstIterator cIter = dict.find(nameOfList);

if (cIter == dict.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

if (nikiforova::isNumber(arg))

{

nikiforova::doRemove(nameOfList, std::stoll(arg), dict);

}

else

{

cIter = dict.find(arg);

if (cIter == dict.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::doRemove(nameOfList, arg, dict);

}

}

void nikiforova::doRemove(const std::string& nameOfList, long long value, DictOfLists& dict)

{

nikiforova::BidirectionalList< long long > destList = dict.find(nameOfList)->second;

nikiforova::BidirectionalList< long long >::Iterator destIter = destList.begin();

while (destIter != destList.end())

{

if (\*destIter == value)

{

destIter = destList.erase(destIter);

}

if (destIter != destList.end())

{

++destIter;

}

}

dict.find(nameOfList)->second = destList;

}

void nikiforova::doRemove(const std::string& nameOfList, const std::string& nameOfSourceList, DictOfLists& dict)

{

nikiforova::BidirectionalList< long long > destList = dict.find(nameOfList)->second;

nikiforova::BidirectionalList< long long >::Iterator destIter = destList.begin();

nikiforova::BidirectionalList< long long > sourceList = dict.find(nameOfSourceList)->second;

nikiforova::BidirectionalList< long long >::Iterator sourceIter = sourceList.begin();

while (sourceIter != sourceList.end())

{

destIter = destList.begin();

while (destIter != destList.end())

{

if (\*destIter == \*sourceIter)

{

destIter = destList.erase(destIter);

}

if (destIter != destList.end())

{

++destIter;

}

}

++sourceIter;

}

dict.find(nameOfList)->second = destList;

}

void nikiforova::concat(std::string& str, DictOfLists& dict)

{

int countArgs = 0;

std::string nameOfList = nikiforova::getWord(str);

if (!nameOfList.empty())

{

countArgs++;

}

nikiforova::BidirectionalList< long long > newList;

while (!str.empty())

{

std::string word = nikiforova::getWord(str);

countArgs++;

auto iter = dict.find(word);

if (iter == dict.end())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

BidirectionalList< long long > temp = dict.get(word);

for (auto&& item: temp)

{

newList.pushBack(item);

}

}

if (countArgs < 3)

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

dict.push(nameOfList, newList);

}

void nikiforova::equal(std::string& str, DictOfLists& dict)

{

nikiforova::doEqual(std::cout, str, dict);

}

std::ostream& nikiforova::doEqual(std::ostream& out, std::string& str, DictOfLists& dict)

{

int countArgs = 0;

std::string word1 = nikiforova::getWord(str);

if (!word1.empty())

{

countArgs++;

}

DictOfLists::ConstIterator cIter1 = dict.find(word1);

if (cIter1 == dict.cend())

{

return nikiforova::invalidCommandMessage(out);

}

while (!str.empty())

{

std::string word2 = nikiforova::getWord(str);

countArgs++;

BidirectionalList< long long > list1 = dict.get(word1);

DictOfLists::ConstIterator cIter2 = dict.find(word2);

if (cIter2 == dict.cend())

{

return nikiforova::invalidCommandMessage(out);

}

BidirectionalList< long long > list2 = dict.get(word2);

if (list1 != list2)

{

return out << "<FALSE>\n";

}

word1 = word2;

}

if (countArgs < 2)

{

return nikiforova::invalidCommandMessage(std::cout);

}

return out << "<TRUE>\n";

}

**./<ROOT>/nikiforova.ekaterina/S3/errorMessages.h**

#ifndef ERRORMESSAGES\_H

#define ERRORMESSAGES\_H

#include <ostream>

namespace nikiforova {

std::ostream& invalidCommandMessage(std::ostream& out);

std::ostream& emptyMessage(std::ostream& out);

}

#endif

**./<ROOT>/nikiforova.ekaterina/S3/errorMessages.cpp**

#include "errorMessages.h"

std::ostream& nikiforova::invalidCommandMessage(std::ostream& out)

{

return out << "<INVALID COMMAND>\n";

}

std::ostream& nikiforova::emptyMessage(std::ostream& out)

{

return out << "<EMPTY>\n";

}

**./<ROOT>/nikiforova.ekaterina/S3/** **operationsWithStrings.h**

#ifndef DIFFERENTUSEFULFUNCTIONS\_H

#define DIFFERENTUSEFULFUNCTIONS\_H

#include <string>

namespace nikiforova {

bool isNumber(const std::string&);

std::string getWord(std::string&);

}

#endif

**./<ROOT>/nikiforova.ekaterina/S3/operationsWithStrings.cpp**

#include "operationsWithStrings.h"

bool nikiforova::isNumber(const std::string& str)

{

bool isNumber = 1;

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

{

if (!std::isdigit(str[i]) && !((str[i] == '-') && (i == 0)))

{

isNumber = 0;

}

}

return isNumber;

}

std::string nikiforova::getWord(std::string& str)

{

std::string word = "";

if (str[0] == ' ')

{

str.erase(0, 1);

}

word = str.substr(0, str.find(" "));

str = str.erase(0, str.find(" "));

return word;

}

**./<ROOT>/nikiforova.ekaterina/S3/dictionary.h**

#ifndef DICTIONARY\_H

#define DICTIONARY\_H

#include <utility>

#include "forwardList.h"

namespace nikiforova {

template < typename Key, typename Value, typename Compare = std::less< Key > >

class Dictionary {

public:

using Iterator = typename ForwardList< std::pair< Key, Value > >::Iterator;

using ConstIterator = typename ForwardList< std::pair< Key, Value > >::ConstIterator;

using pairIterBool = std::pair< typename Dictionary< Key, Value, Compare >::Iterator, bool >;

Dictionary() = default;

Dictionary(std::initializer\_list< std::pair< Key, Value > >);

Dictionary(const Dictionary&) = default;

Dictionary(Dictionary&&) = default;

~Dictionary() = default;

bool isEmpty() const noexcept;

size\_t getSize() const noexcept;

void push(const Key& k, const Value& v);

Iterator find(const Key& k);

ConstIterator find(const Key& k) const;

Value get(const Key& k);

void drop(Key k);

pairIterBool insert(const std::pair< Key, Value >&);

Iterator erase(Iterator);

Iterator begin() noexcept;

ConstIterator cbegin() const noexcept;

Iterator end() noexcept;

ConstIterator cend() const noexcept;

private:

ForwardList< std::pair< Key, Value > > list\_;

bool isLess(const Key&, const Key&);

bool isEqual(const Key&, const Key&);

};

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::Iterator Dictionary< Key, Value, Compare >::begin() noexcept

{

return list\_.begin();

}

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::ConstIterator Dictionary< Key, Value, Compare >::cbegin() const noexcept

{

return list\_.cbegin();

}

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::Iterator Dictionary< Key, Value, Compare >::end() noexcept

{

return list\_.end();

}

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::ConstIterator Dictionary< Key, Value, Compare >::cend() const noexcept

{

return list\_.cend();

}

template< typename Key, typename Value, typename Compare >

Dictionary< Key, Value, Compare >::Dictionary(std::initializer\_list< std::pair< Key, Value > > list)

{

for (auto&& pair: list)

{

push(pair.first, pair.second);

}

}

template< typename Key, typename Value, typename Compare >

bool Dictionary< Key, Value, Compare >::isEmpty() const noexcept

{

return list\_.isEmpty();

}

template< typename Key, typename Value, typename Compare >

size\_t Dictionary< Key, Value, Compare >::getSize() const noexcept

{

return list\_.size();

}

template< typename Key, typename Value, typename Compare >

bool Dictionary< Key, Value, Compare >::isLess(const Key& lhs, const Key& rhs)

{

return Compare()(lhs, rhs);

}

template< typename Key, typename Value, typename Compare >

bool Dictionary< Key, Value, Compare >::isEqual(const Key& lhs, const Key& rhs)

{

return (!isLess(lhs, rhs)) && (!isLess(rhs, lhs));

}

template< typename Key, typename Value, typename Compare >

void Dictionary< Key, Value, Compare >::push(const Key& k, const Value& v)

{

if (isEmpty())

{

list\_.pushFront(std::pair< Key, Value >(k, v));

}

else

{

Iterator iter = list\_.begin();

while (iter != list\_.end())

{

if (isEqual(iter->first, k))

{

throw std::logic\_error("Can't push");

}

if (!isLess(iter->first, k))

{

break;

}

iter++;

}

const std::pair< Key, Value > p(k, v);

list\_.insert(p, iter);

}

}

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::Iterator Dictionary< Key, Value, Compare >::find(const Key& k)

{

Iterator iter = list\_.begin();

while (iter != list\_.end())

{

if (isEqual(iter->first, k))

{

return iter;

}

iter++;

}

return iter;

}

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::ConstIterator Dictionary< Key, Value, Compare >::find(const Key& k) const

{

return ConstIterator(find(k));

}

template< typename Key, typename Value, typename Compare >

Value Dictionary< Key, Value, Compare >::get(const Key& k)

{

ConstIterator iter = find(k);

if (iter != end())

{

return(iter->second);

}

else

{

throw std::logic\_error("Key doesn't exist");

}

}

template< typename Key, typename Value, typename Compare >

void Dictionary< Key, Value, Compare >::drop(Key k)

{

ConstIterator iter = this->find(k);

if (iter == end())

{

throw std::logic\_error("Key doesn't exist");

}

list\_.erase(iter);

}

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::pairIterBool Dictionary< Key, Value, Compare >::insert(const std::pair< Key, Value >& p)

{

Iterator iter = begin();

if (isEmpty())

{

list\_.pushBack(p);

return { list\_.begin(), true };

}

while ((iter != end()) && (isLess(iter->first, p.first)))

{

iter++;

}

if ((iter != end()) && isEqual(p.first, iter->first))

{

return { iter, false };

}

list\_.insert(p, iter);

return { iter, true };

}

template< typename Key, typename Value, typename Compare >

typename Dictionary< Key, Value, Compare >::Iterator Dictionary< Key, Value, Compare >::erase(Iterator iter)

{

if (iter == end())

{

throw std::logic\_error("Empty list");

}

auto tempKey = iter->first;

list\_.erase(iter);

if (isEmpty())

{

return end();

}

Iterator tempIter = begin();

while (tempIter != end() && (isLess(tempIter->first, tempKey)))

{

tempIter++;

}

return tempIter;

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S3/bidirectionalList.h**

#ifndef BIDIRECTIONALLIST\_H

#define BIDIRECTIONALLIST\_H

#include <cassert>

#include <stdexcept>

#include <iterator>

#include <cstddef>

#include <utility>

namespace nikiforova {

namespace detail {

template< typename T >

struct biNode\_t {

T data\_;

biNode\_t\* prev\_;

biNode\_t\* next\_;

};

}

template< typename T >

class BidirectionalList {

public:

BidirectionalList();

BidirectionalList(const BidirectionalList&);

BidirectionalList(BidirectionalList&&) noexcept;

~BidirectionalList();

BidirectionalList& operator= (const BidirectionalList&);

BidirectionalList& operator= (BidirectionalList&&) noexcept;

bool operator==(BidirectionalList&);

bool operator!=(BidirectionalList&);

size\_t size() const noexcept;

void pushFront(const T&);

void popFront();

void pushBack(const T&);

void popBack();

void swap(BidirectionalList&) noexcept;

void clear();

bool isEmpty() const noexcept;

const T& getFront() const;

const T& getBack() const;

class Iterator: public std::iterator< std::bidirectional\_iterator\_tag, T > {

public:

friend class BidirectionalList< T >;

Iterator():

node\_(nullptr)

{}

~Iterator() = default;

Iterator(const Iterator&) = default;

Iterator& operator=(const Iterator&) = default;

Iterator& operator++()

{

assert(node\_ != nullptr);

node\_ = node\_->next\_;

return \*this;

}

Iterator operator++(int)

{

assert(node\_ != nullptr);

Iterator result(\*this);

++(\*this);

return result;

}

Iterator& operator--()

{

assert(node\_ != nullptr);

node\_ = node\_->prev\_;

return \*this;

}

Iterator operator--(int)

{

assert(node\_ != nullptr);

Iterator result(\*this);

--(\*this);

return result;

}

T& operator\*()

{

assert(node\_ != nullptr);

return node\_->data\_;

}

T\* operator->()

{

assert(node\_ != nullptr);

return std::addressof(node\_->data\_);

}

const T& operator\*() const

{

assert(node\_ != nullptr);

return node\_->data\_;

}

const T\* operator->() const

{

assert(node\_ != nullptr);

return std::addressof(node\_->data\_);

}

bool operator==(const Iterator& rhs) const

{

return node\_ == rhs.node\_;

}

bool operator!=(const Iterator& rhs) const

{

return !(rhs == \*this);

}

private:

detail::biNode\_t< T >\* node\_;

Iterator(detail::biNode\_t< T >\* rhsNode):

node\_(rhsNode)

{}

};

class ConstIterator: public std::iterator< std::bidirectional\_iterator\_tag, T > {

public:

friend class BidirectionalList< T >;

ConstIterator():

node\_(nullptr)

{}

~ConstIterator() = default;

ConstIterator(const ConstIterator&) = default;

ConstIterator& operator=(const ConstIterator&) = default;

ConstIterator& operator++()

{

assert(node\_ != nullptr);

node\_ = node\_->next\_;

return \*this;

}

ConstIterator operator++(int)

{

assert(node\_ != nullptr);

ConstIterator result(\*this);

++(\*this);

return result;

}

ConstIterator& operator--()

{

assert(node\_ != nullptr);

node\_ = node\_->prev\_;

return \*this;

}

ConstIterator operator--(int)

{

assert(node\_ != nullptr);

ConstIterator result(\*this);

--(\*this);

return result;

}

const T& operator\*()

{

assert(node\_ != nullptr);

return node\_->data\_;

}

const T\* operator->()

{

assert(node\_ != nullptr);

return std::addressof(node\_->data\_);

}

bool operator==(const ConstIterator& rhs) const

{

return node\_ == rhs.node\_;

}

bool operator!=(const ConstIterator& rhs) const

{

return !(rhs == \*this);

}

private:

const detail::biNode\_t< T >\* node\_;

};

Iterator insert(const T&, Iterator);

Iterator erase(Iterator);

Iterator begin() noexcept;

Iterator end() noexcept;

ConstIterator cbegin() const noexcept;

ConstIterator cend() const noexcept;

private:

detail::biNode\_t< T >\* head\_;

detail::biNode\_t< T >\* tail\_;

size\_t size\_;

};

template< typename T >

BidirectionalList< T >::BidirectionalList():

head\_(nullptr),

tail\_(nullptr),

size\_(0)

{}

template< typename T >

BidirectionalList< T >::BidirectionalList(const BidirectionalList& x):

head\_(nullptr),

tail\_(nullptr),

size\_(0)

{

if (!x.isEmpty())

{

detail::biNode\_t< T >\* srcPtr = x.head\_;

try

{

while (srcPtr)

{

pushBack(srcPtr->data\_);

srcPtr = srcPtr->next\_;

}

}

catch (...)

{

clear();

throw;

}

}

}

template< typename T >

BidirectionalList< T >::BidirectionalList(BidirectionalList&& rhs) noexcept:

head\_(rhs.head\_),

tail\_(rhs.tail\_),

size\_(rhs.size\_)

{

rhs.tail\_ = nullptr;

rhs.head\_ = nullptr;

rhs.size\_ = 0;

}

template< typename T >

BidirectionalList< T >::~BidirectionalList()

{

clear();

}

template< typename T >

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

{

if (this != std::addressof(rhs))

{

BidirectionalList< T > temp(rhs);

swap(temp);

}

return \*this;

}

template< typename T >

BidirectionalList< T >& BidirectionalList< T >::operator=(BidirectionalList< T >&& rhs) noexcept

{

if (this != std::addressof(rhs))

{

BidirectionalList< T > temp(std::move(rhs));

swap(temp);

}

return \*this;

}

template< typename T >

bool BidirectionalList< T >::operator==(BidirectionalList& rhs)

{

BidirectionalList< long long >::Iterator lhsIter = begin();

BidirectionalList< long long >::Iterator rhsIter = rhs.begin();

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

{

return false;

}

while (lhsIter != end())

{

if (\*lhsIter != \*rhsIter)

{

return false;

}

++lhsIter;

++rhsIter;

}

return true;

}

template< typename T >

bool BidirectionalList< T >::operator!=(BidirectionalList& rhs)

{

return !(\*this == rhs);

}

template< typename T >

size\_t BidirectionalList< T >::size() const noexcept

{

return size\_;

}

template< typename T >

void BidirectionalList< T >::pushFront(const T& val)

{

detail::biNode\_t< T >\* temp = new detail::biNode\_t< T >{ val, nullptr, head\_ };

if (!isEmpty())

{

head\_->prev\_ = temp;

}

else

{

tail\_ = temp;

}

head\_ = temp;

size\_++;

}

template< typename T >

void BidirectionalList< T >::popFront()

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

detail::biNode\_t< T >\* newHead = head\_->next\_;

if (head\_ == tail\_)

{

tail\_ = nullptr;

}

delete head\_;

if (newHead != nullptr)

{

newHead->prev\_ = nullptr;

}

head\_ = newHead;

size\_--;

}

template< typename T >

void BidirectionalList< T >::pushBack(const T& val)

{

if (isEmpty())

{

head\_ = new detail::biNode\_t< T >{ val, nullptr, nullptr };

tail\_ = head\_;

}

else

{

tail\_->next\_ = new detail::biNode\_t< T >{ val, tail\_, nullptr };

tail\_ = tail\_->next\_;

}

size\_++;

}

template< typename T >

void BidirectionalList< T >::popBack()

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

detail::biNode\_t< T >\* newTail = tail\_->prev\_;

tail\_->prev\_->next\_ = nullptr;

delete tail\_;

tail\_ = newTail;

size\_--;

}

template< typename T >

void BidirectionalList< T >::swap(BidirectionalList& rhs) noexcept

{

std::swap(head\_, rhs.head\_);

std::swap(tail\_, rhs.tail\_);

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

}

template< typename T >

void BidirectionalList< T >::clear()

{

while (!isEmpty())

{

popFront();

}

}

template< typename T >

bool BidirectionalList< T >::isEmpty() const noexcept

{

return !size\_;

}

template< typename T >

const T& BidirectionalList< T >::getFront() const

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

return head\_->data\_;

}

template< typename T >

const T& BidirectionalList< T >::getBack() const

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

return tail\_->data\_;

}

template< typename T >

typename BidirectionalList< T >::Iterator BidirectionalList< T >::insert(const T& data, Iterator iter)

{

if (iter == begin())

{

pushFront(data);

return iter;

}

if (iter == end())

{

pushBack(data);

return iter;

}

else

{

Iterator temp = begin();

detail::biNode\_t< T >\* tempNode = head\_;

while (++temp != iter)

{

tempNode = tempNode->next\_;

}

tempNode->next\_ = new detail::biNode\_t< T >{ data, tempNode, tempNode->next\_ };

tempNode->next\_->next\_->prev\_ = tempNode->next\_;

size\_++;

return tempNode->next\_;

}

}

template < typename T >

typename BidirectionalList< T >::Iterator BidirectionalList< T >::erase(Iterator iter)

{

if (iter == begin())

{

popFront();

return begin();

}

if (iter == end())

{

throw std::logic\_error("Can't erase");

}

if (++iter == end())

{

popBack();

return end();

}

Iterator tempIter = begin();

detail::biNode\_t< T >\* tempNode = head\_;

while (++tempIter != iter)

{

tempNode = tempNode->next\_;

}

detail::biNode\_t< T >\* tempTempNode = tempNode->next\_;

tempTempNode->prev\_ = tempNode->prev\_;

tempNode->prev\_->next\_ = tempTempNode;

delete tempNode;

size\_--;

return tempTempNode;

}

template< typename T >

typename BidirectionalList< T >::Iterator BidirectionalList< T >::begin() noexcept

{

return Iterator(head\_);

}

template< typename T >

typename BidirectionalList< T >::Iterator BidirectionalList< T >::end() noexcept

{

return Iterator(nullptr);

}

template< typename T >

typename BidirectionalList< T >::ConstIterator BidirectionalList< T >::cbegin() const noexcept

{

return ConstIterator(head\_);

}

template< typename T >

typename BidirectionalList< T >::ConstIterator BidirectionalList< T >::cend() const noexcept

{

return ConstIterator(nullptr);

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S3/forwardList.h**

#ifndef FORWARDLIST\_H

#define FORWARDLIST\_H

#include <cassert>

#include <stdexcept>

#include "list.h"

namespace nikiforova {

template< typename T >

class ForwardList: public nikiforova::detail::List< T > {

public:

ForwardList();

ForwardList(const ForwardList&);

ForwardList(ForwardList&&) noexcept;

~ForwardList();

size\_t size() const noexcept;

void pushFront(const T&);

void popFront();

void pushBack(const T&);

void swap(ForwardList&) noexcept;

void clear();

bool isEmpty() const noexcept;

const T& getFront() const;

const T& getBack() const;

class Iterator {

public:

friend class ForwardList< T >;

Iterator():

node\_(nullptr)

{}

Iterator(detail::node\_t< T >\* rhsNode):

node\_(rhsNode)

{}

~Iterator() = default;

Iterator(const Iterator&) = default;

Iterator& operator=(const Iterator&) = default;

Iterator& operator++()

{

assert(node\_ != nullptr);

node\_ = node\_->next\_;

return \*this;

}

Iterator operator++(int)

{

assert(node\_ != nullptr);

Iterator result(\*this);

++(\*this);

return result;

}

T& operator\*()

{

assert(node\_ != nullptr);

return node\_->data\_;

}

T\* operator->()

{

assert(node\_ != nullptr);

return std::addressof(node\_->data\_);

}

const T& operator\*() const

{

assert(node\_ != nullptr);

return node\_->data\_;

}

const T\* operator->() const

{

assert(node\_ != nullptr);

return std::addressof(node\_->data\_);

}

bool operator==(const Iterator& rhs) const

{

return node\_ == rhs.node\_;

}

bool operator!=(const Iterator& rhs) const

{

return !(rhs == \*this);

}

private:

detail::node\_t< T >\* node\_;

};

class ConstIterator {

public:

friend class ForwardList< T >;

ConstIterator():

iterator\_(nullptr)

{}

ConstIterator(Iterator iter):

iterator\_(iter)

{}

~ConstIterator() = default;

ConstIterator(const ConstIterator&) = default;

ConstIterator& operator=(const ConstIterator&) = default;

ConstIterator& operator++()

{

++iterator\_;

return \*this;

}

ConstIterator operator++(int)

{

return ConstIterator(iterator\_++);

}

const T& operator\*()

{

return \*iterator\_;

}

const T\* operator->()

{

return std::addressof(\*iterator\_);

}

bool operator==(const ConstIterator& rhs) const

{

return iterator\_ == rhs.iterator\_;

}

bool operator!=(const ConstIterator& rhs) const

{

return !(rhs == \*this);

}

private:

Iterator iterator\_;

};

void insert(const T&, ConstIterator);

void erase(ConstIterator);

Iterator begin() noexcept

{

return Iterator(detail::List< T >::head\_);

}

Iterator end() noexcept

{

return Iterator(nullptr);

}

ConstIterator cbegin() const noexcept

{

return ConstIterator(detail::List< T >::head\_);

}

ConstIterator cend() const noexcept

{

return ConstIterator(nullptr);

}

};

template< typename T >

void ForwardList< T >::insert(const T& data, ConstIterator iter)

{

if (iter == this->begin())

{

pushFront(data);

}

else

{

ConstIterator temp = this->cbegin();

detail::node\_t< T >\* tempNode = detail::List< T >::head\_;

while (++temp != iter)

{

tempNode = tempNode->next\_;

}

if (tempNode->next\_ == nullptr)

{

tempNode->next\_ = new detail::node\_t< T >{ data, tempNode->next\_ };

detail::List< T >::tail\_ = tempNode->next\_;

}

else

{

tempNode->next\_ = new detail::node\_t< T >{ data, tempNode->next\_ };

}

detail::List< T >::size\_++;

}

}

template< typename T >

void ForwardList< T >::erase(ConstIterator iter)

{

ConstIterator tempIter = this->cbegin();

detail::node\_t< T >\* tempNode = detail::List< T >::head\_;

if (tempIter == iter)

{

tempNode = tempNode->next\_;

}

else

{

while (++tempIter != iter)

{

tempNode = tempNode->next\_;

}

}

if (tempIter == this->cbegin())

{

popFront();

}

else

{

detail::node\_t< T >\* tempTempNode = tempNode->next\_->next\_;

delete tempNode->next\_;

tempNode->next\_ = tempTempNode;

if (tempTempNode == nullptr)

{

detail::List< T >::tail\_ = tempNode;

}

detail::List< T >::size\_--;

}

}

template< typename T >

ForwardList< T >::ForwardList():

detail::List< T >::List()

{}

template< typename T >

ForwardList< T >::ForwardList(const ForwardList< T >& x):

detail::List< T >::List(x)

{}

template< typename T >

ForwardList< T >::ForwardList(ForwardList< T >&& rhs) noexcept:

detail::List< T >::List(rhs)

{}

template< typename T >

ForwardList< T >::~ForwardList()

{

clear();

}

template< typename T >

size\_t ForwardList< T >::size() const noexcept

{

return detail::List< T >::size();

}

template< typename T >

void ForwardList< T >::pushFront(const T& val)

{

detail::List< T >::pushFront(val);

}

template< typename T >

void ForwardList< T >::popFront()

{

detail::List< T >::popFront();

}

template< typename T >

void ForwardList< T >::swap(ForwardList< T >& x) noexcept

{

detail::List< T >::swap(x);

}

template< typename T >

void ForwardList< T >::clear()

{

detail::List< T >::clear();

}

template< typename T >

void ForwardList< T >::pushBack(const T& val)

{

detail::List< T >::pushBack(val);

}

template< typename T >

bool ForwardList< T >::isEmpty() const noexcept

{

return detail::List< T >::isEmpty();

}

template< typename T >

const T& ForwardList< T >::getFront() const

{

return detail::List< T >::getFront();

}

template< typename T >

const T& ForwardList< T >::getBack() const

{

return detail::List< T >::getBack();

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S3/list.h**

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

namespace nikiforova {

namespace detail {

template< typename T >

struct node\_t

{

T data\_;

node\_t\* next\_;

};

template< typename T >

class List {

public:

List();

List(const List&);

List(List&&) noexcept;

~List();

List& operator= (const List&);

List& operator= (List&&) noexcept;

size\_t size() const noexcept;

void pushFront(const T&);

void popFront();

void pushBack(const T&);

void swap(List&) noexcept;

void clear();

bool isEmpty() const noexcept;

const T& getFront() const;

const T& getBack() const;

protected:

node\_t< T >\* head\_;

node\_t< T >\* tail\_;

size\_t size\_;

};

template< typename T >

List< T >::List():

head\_(nullptr),

tail\_(nullptr),

size\_(0)

{}

template< typename T >

List< T >::List(const List< T >& x):

head\_(nullptr),

tail\_(nullptr),

size\_(0)

{

if (!x.isEmpty())

{

node\_t< T >\* srcPtr = x.head\_;

try

{

while (srcPtr)

{

pushBack(srcPtr->data\_);

srcPtr = srcPtr->next\_;

}

}

catch (...)

{

clear();

throw;

}

}

}

template< typename T >

List< T >::List(List< T >&& rhs) noexcept:

head\_(rhs.head\_),

tail\_(rhs.tail\_),

size\_(rhs.size\_)

{

rhs.tail\_ = nullptr;

rhs.head\_ = nullptr;

rhs.size\_ = 0;

}

template< typename T >

List< T >::~List()

{

clear();

}

template< typename T >

List< T >& List< T >::operator=(const List< T >& x)

{

if (this != std::addressof(x))

{

List< T > temp(x);

swap(temp);

}

return \*this;

}

template< typename T >

List< T >& List< T >::operator=(List< T >&& rhs) noexcept

{

if (this != std::addressof(rhs))

{

List< T > temp(std::move(rhs));

swap(temp);

}

return \*this;

}

template< typename T >

size\_t List< T >::size() const noexcept

{

return size\_;

}

template< typename T >

void List< T >::pushFront(const T& val)

{

head\_ = new node\_t< T >{ val, head\_ };

size\_++;

}

template< typename T >

void List< T >::popFront()

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

node\_t< T >\* newHead = head\_->next\_;

if (head\_ == tail\_)

{

tail\_ = nullptr;

}

delete head\_;

head\_ = newHead;

size\_--;

}

template< typename T >

void List< T >::pushBack(const T& val)

{

if (isEmpty())

{

head\_ = new node\_t< T >{ val, nullptr };

tail\_ = head\_;

}

else

{

tail\_->next\_ = new node\_t< T >{ val, nullptr };

tail\_ = tail\_->next\_;

}

size\_++;

}

template< typename T >

void List< T >::swap(List< T >& x) noexcept

{

std::swap(head\_, x.head\_);

std::swap(tail\_, x.tail\_);

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

}

template< typename T >

void List< T >::clear()

{

while (!isEmpty())

{

popFront();

}

}

template< typename T >

bool List< T >::isEmpty() const noexcept

{

return !size\_;

}

template< typename T >

const T& List< T >::getFront() const

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

return head\_->data\_;

}

template< typename T >

const T& List< T >::getBack() const

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

return tail\_->data\_;

}

}

}

#endif