**Постановка задачи**

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

**S6. Сортировки**

1. Реализуйте указанные преподавателем алгоритмы сортировки (всего не менее 3-х)
   * По крайней мере один из реализованных алгоритмов должен быть применим для реализованного ранее однонаправленного списка
   * По крайней мере два из реализованных алгоритмов должны быть применимы для реализованного ранее двунаправленного списка
   * Все реализованные алгоритмы должны быть применимы для стандартного контейнера std::deque
2. Реализуемая программа должна сортировать числа и выводить результаты сортировки на стандартный вывод:
   * Заполните нужное количество экземпляров реализованных контейнеров: однонаправленного списка и двунаправленного списка, а также стандартный контейнер std::deque сгенерированными числами (одинаковым набором) и отсортируйте данные различными алгоритмами.
   * Исходные данные и результаты сортировок выведите на стандартный вывод на отдельных строках. Например, результаты работы программы могут быть такими:

$ ./lab ascending ints 4

1 3 2 4

1 2 3 4

1 2 3 4

1 2 3 4

1 2 3 4

1 2 3 4

1 2 3 4

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

$ ./lab [ascending|descending] [ints|floats] [size]

Все параметры обязательные

* + - Числа сортируются по возрастанию или по убыванию в зависимости от параметра командной строки (ascending - по возрастанию, descending - по убыванию).
    - Числа должны быть сгенерированы с использованием генератора псевдослучайных чисел в количестве, определяемым параметром командной строки size
    - Тип сортируемых чисел также определяется параметром командной строки (ints - для знаковых целых и floats - для вещественных).

**Приёмочные тесты**

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

|  |  |  |
| --- | --- | --- |
| **#** | **Описание** | **Результат** |
| **1** | Сортировка вещественных чисел по убыванию | **Parameters: “**descending floats 3**”**  **Expected**:  0.8 354.2 121.5  354.2 121.5 0.8  354.2 121.5 0.8  354.2 121.5 0.8  354.2 121.5 0.8  354.2 121.5 0.8  354.2 121.5 0.8 |
| **2** | Сортировка вещественных чисел по возрастанию | **Parameters: “**ascending floats 3**”**  **Expected**:  0.8 354.2 121.5  0.8 121.5 354.2  0.8 121.5 354.2  0.8 121.5 354.2  0.8 121.5 354.2  0.8 121.5 354.2  0.8 121.5 354.2 |
| **3** | Сортировка целых чисел по убыванию | **Parameters: “**descendingints 3**”**  **Expected**:  1 3 0  3 1 0  3 1 0  3 1 0  3 1 0  3 1 0  3 1 0 |
| **4** | Сортировка целых чисел по возрастанию | **Parameters: “**ascending ints 3**”**  **Expected**:  1 3 0  0 1 3  0 1 3  0 1 3  0 1 3  0 1 3  0 1 3 |
| **5** | Сортировка одного числа | **Parameters: “**descending ints 1**”**  **Expected:**  1  1  1  1  1  1  1 |
| **6** | Неизвестные аргументы командной строки | **Parameters**: “unexpected ints 1”  **Expected**:  Сообщение «INVALID COMMAND» и ненулевой код возврата |
| **7** | Некорректное количество аргументов | **Parameters**: “ints 1”  **Expected**:  Сообщение «INVALID COMMAND» и ненулевой код возврата |
| **Parameters**: “ints”  **Expected**:  Сообщение «INVALID COMMAND» и ненулевой код возврата |
| **Parameters**: “”  **Expected**:  Сообщение «INVALID COMMAND» и ненулевой код возврата |
| **Parameters**: “unexpected floats ints 1”  **Expected**:  Сообщение «INVALID COMMAND» и ненулевой код возврата |

**Исходные тексты программы**

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

**./<ROOT>/nikiforova.ekaterina/S6/main.cpp**

﻿#include <iostream>

#include <string>

#include <functional>

#include "errorMessages.h"

#include "commandsS6.h"

#include "dictionary.h"

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

{

if (argc != 4)

{

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

return 1;

}

std::string command = argv[1];

std::string dataType = argv[2];

size\_t size = 0;

try {

size = std::stoull(argv[3]);

}

catch (const std::exception& e) {

std::cerr << e.what() << '\n';

return 1;

}

if ((command != "ascending" && command != "descending") || (dataType != "ints" && dataType != "floats"))

{

nikiforova::invalidCommandMessage(std::cout);

return 1;

}

using command\_t = std::function < void(size\_t, std::ostream&) >;

nikiforova::Dictionary< std::string, command\_t > commands

{

{"ascendingints", nikiforova::ascendingInt},

{"ascendingfloats", nikiforova::ascendingFloat},

{"descendingints", nikiforova::descendingInt},

{"descendingfloats", nikiforova::descendingFloat}

};

std::string typeAndDirection = command + dataType;

auto iter = commands.find(typeAndDirection);

iter->second(size, std::cout);

return 0;

}

**./<ROOT>/nikiforova.ekaterina/S6/commandsS6.h**

#ifndef COMMANDSS6\_H

#define COMMANDSS6\_H

#include <iosfwd>

#include <deque>

#include "bidirectionalList.h"

#include "forwardList.h"

#include "fillRandom.h"

#include "sortings.h"

#include "print.h"

namespace nikiforova {

void ascendingInt(size\_t, std::ostream&);

void ascendingFloat(size\_t, std::ostream&);

void descendingInt(size\_t, std::ostream&);

void descendingFloat(size\_t, std::ostream&);

template< typename DataType, typename Compare >

void printSorted(size\_t size, Compare cmp, std::ostream& out)

{

nikiforova::BidirectionalList< DataType > bList1;

nikiforova::BidirectionalList< DataType > bList2;

nikiforova::ForwardList< DataType > fList;

std::deque< DataType > deque1;

std::deque< DataType > deque2;

std::deque< DataType > deque3;

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

{

DataType value = getRandValue< DataType >();

bList1.pushBack(value);

bList2.pushBack(value);

fList.pushBack(value);

deque1.push\_back(value);

deque2.push\_back(value);

deque3.push\_back(value);

}

nikiforova::print(bList1.begin(), bList1.end(), out);

nikiforova::oddEvenSort(fList.begin(), fList.size(), cmp);

nikiforova::oddEvenSort(bList1.begin(), bList1.size(), cmp);

nikiforova::oddEvenSort(deque1.begin(), deque1.size(), cmp);

nikiforova::shellSort(bList2.begin(), bList2.end(), cmp);

nikiforova::shellSort(deque2.begin(), deque2.end(), cmp);

nikiforova::qSort(deque3.begin(), deque3.end(), cmp);

nikiforova::print(fList.begin(), fList.end(), out);

nikiforova::print(bList1.begin(), bList1.end(), out);

nikiforova::print(deque1.begin(), deque1.end(), out);

nikiforova::print(bList2.begin(), bList2.end(), out);

nikiforova::print(deque2.begin(), deque2.end(), out);

nikiforova::print(deque3.begin(), deque3.end(), out);

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S6/commandsS6.cpp**

#include "commandsS6.h"

void nikiforova::ascendingInt(size\_t size, std::ostream& out)

{

nikiforova::printSorted< int >(size, std::less< int >(), out);

}

void nikiforova::ascendingFloat(size\_t size, std::ostream& out)

{

nikiforova::printSorted< float >(size, std::less< float >(), out);

}

void nikiforova::descendingInt(size\_t size, std::ostream& out)

{

nikiforova::printSorted< int >(size, std::greater< int >(), out);

}

void nikiforova::descendingFloat(size\_t size, std::ostream& out)

{

nikiforova::printSorted< float >(size, std::greater< float >(), out);

}

**./<ROOT>/nikiforova.ekaterina/S6/fillRandom.h**

#ifndef FILLRANDOM\_H

#define FILLRANDOM\_H

#include <cstdlib>

namespace nikiforova {

template< typename DataType >

DataType getRandValue()

{

return static\_cast< DataType >(std::rand() \* 57.0 / 2972);

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S6/print.h**

#ifndef PRINT\_H

#define PRINT\_H

#include <iostream>

#include <iomanip>

namespace nikiforova {

template < typename Iter >

std::ostream& print(const Iter begin, const Iter end, std::ostream& out)

{

out << std::setprecision(1);

Iter temp = begin;

while (temp != end)

{

if (temp != begin)

{

out << " ";

}

out << std::fixed << \*temp;

temp++;

}

return out << "\n";

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S6/sortings.h**

#ifndef SORTINGS\_H

#define SORTINGS\_H

#include <utility>

#include <iterator>

namespace nikiforova {

template < typename BIterator, typename Compare >

void shellSort(BIterator first, BIterator second, Compare cmp)

{

if (std::next(first, 1) == second || first == second)

{

return;

}

auto size = std::distance(first, second);

auto step = size / 2;

while (step > 0)

{

for (auto i = std::next(first, step); i != second; i++)

{

auto j = i;

while (std::distance(first, j) >= step && cmp(\*j, \*std::prev(j, step)))

{

std::iter\_swap(std::prev(j, step), j);

j = std::prev(j, step);

}

}

step /= 2;

}

}

template < typename Iterator, typename Compare >

void oddEvenSort(Iterator first, size\_t size, Compare cmp)

{

if (size == 1)

{

return;

}

bool isSorted = 0;

while (isSorted == 0)

{

isSorted = 1;

auto temp = first;

for (size\_t i = 0; i <= (size - 2); i += 2)

{

auto iter = temp;

auto nextIter = ++temp;

if (!cmp(\*iter, \*nextIter))

{

std::iter\_swap(iter, nextIter);

isSorted = 0;

}

++temp;

}

temp = first;

++temp;

for (size\_t i = 1; i <= (size - 2); i += 2)

{

auto iter = temp;

auto nextIter = ++temp;

if (!cmp(\*iter, \*nextIter))

{

std::iter\_swap(iter, nextIter);

isSorted = 0;

}

++temp;

}

}

}

template < typename BIterator, typename Compare >

void qSort(BIterator first, BIterator second, Compare cmp)

{

if (std::next(first, 1) == second || first == second)

{

return;

}

auto left = first;

auto right = second;

right--;

auto size = std::distance(first, second);

auto pivot = std::next(first, size / 2);

auto dataPivot = \*pivot;

while (left <= right)

{

while (cmp(\*left, dataPivot))

{

++left;

}

while (cmp(dataPivot, \*right))

{

--right;

}

if (left <= right)

{

std::iter\_swap(left, right);

if (pivot != left && dataPivot == \*left)

{

pivot = left;

right--;

}

else if (pivot != right && dataPivot == \*right)

{

pivot = right;

left++;

}

else

{

left++;

}

}

}

qSort(first, pivot, cmp);

qSort(++pivot, second, cmp);

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S6/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/S6/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/S6/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/S6/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/S6/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/S6/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