**Постановка задачи**

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

**S4. Бинарные деревья поиска I**

1. Реализуйте шаблон класса BinarySearchTree и соответствующие итераторы

|  |  |
| --- | --- |
| 1 | template< typename Key, typename Value, typename Compare > |
| 2 | class BinarySearchTree { |
| 3 | ... |
| 4 | }; |

Интерфейс класса должен быть безопасным. Дерево должно быть сбалансированным.

5. Исправьте реализацию класса «Словарь»: используйте реализованное дерево поиска вместо списка в качестве контейнера.

3. Реализуйте программу, описанную в задании Списки I с использованием реализованного дерева поиска

4. Избегайте дублирования кода. При необходимости сделайте правки в предыдущих работах

**Приёмочные тесты**

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

|  |  |  |
| --- | --- | --- |
| # | **Описание** | **Результат** |
| 1 | Вычитание словарей | **Input**:  complement third second first  print third  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is:**  third 4 mouse |
| 2 | Вычитание словарей с сохранением нового в уже существующий словарь | **Input**:  complement second second first  print second  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  second 4 mouse |
| 3 | Вычитание с несуществующим словарем | **Input**:  complement third third second  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  <INVALID COMMAND> |
| 4 | Вычитание словаря из самого себя | **Input**:  complement third second second  print third  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  <EMPTY> |
| 5 | Пересечение словарей | **Input**:  intersect third second first  print third  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  third 1 name 2 keyboard |
| 6 | Пересечение словарей с сохранением нового в уже существующий словарь | **Input**:  intersect second second first  print second  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  second 1 name 2 keyboard |
| 7 | Пересечение словарей, не имеющих общих элементов | **Input**:  intersect third first second  print third  **Expected**:  **With data**:  first 3 name 5 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  <EMPTY> |
| 8 | Пересечение с несуществующим словарем | **Input**:  intersect third third second  **Expected**:  **With data**:  first 3 name 5 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  <INVALID COMMAND> |
| 9 | Пересечение словаря с самим собой | **Input**:  intersect third second second  print third  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  third 1 name 2 keyboard 4 mouse |
| 10 | Пересечение с пустым словарем | **Input**:  intersect third first second  print third  **Expected**:  **With data**:  first  second 4 mouse 1 name 2 keyboard  **expected result is**:  <EMPTY> |
| 11 | Печать пустого словаря | **Input**:  print first  **Expected**:  **With data**:  first  **expected result is**:  <EMPTY> |
| 12 | Печать не пустого словаря | **Input**:  print second  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  second 1 name 2 keyboard 4 mouse |
| 13 | Печать не существующего словаря | **Input**:  print third  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  <INVALID COMMAND> |
| 14 | Несколько команд подряд | **Input**:  union third first second  complement fourth first first  intersect first third fourth  print first  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  <EMPTY> |
| 15 | Неожиданная команда | **Input**:  unexpected second  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  <INVALID COMMAND> |
| 16 | Объединение словарей | **Input**:  union third second first  print third  **Expected**:  **With data**:  first 1 name 3 machine 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  third 1 name 2 keyboard 3 machine 4 mouse |
| 17 | Объединение словарей с сохранением нового в уже существующий словарь | **Input**:  union first second first  print first  **Expected**:  **With data**:  first 1 name 3 machine 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  first 1 name 2 keyboard 3 machine 4 mouse |
| 18 | Объединение пустых словарей | **Input**:  union third first second  print third  **Expected**:  **With data**:  first  second  **expected result is**:  <EMPTY> |
| 19 | Объединение словаря с самим собой | **Input**:  union third first first  print third  **Expected**:  **With data**:  first 1 name 3 machine 2 surname  second 4 mouse 1 name 2 keyboard  **expected result is**:  third 1 name 2 surname 3 machine |
| 20 | Объединение с одним пустым словарем | **Input**:  union third first second  print third  **Expected**:  **With data**:  first 1 name 3 machine 2 surname  second  **expected result is**:  third 1 name 2 surname 3 machine |
| 21 | Объединение с несуществующим словарем | **Input**:  union third third second  **Expected**:  **With data**:  first 1 name 2 surname  second 4 mouse 1 name 2 keyboard  **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/S4/main.cpp**

﻿ ﻿#include <iostream>

#include <string>

#include <fstream>

#include <functional>

#include "forwardList.h"

#include "2-3Tree.h"

#include "errorMessages.h"

#include "commandsWithDictsOfDicts.h"

#include "dictionary.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::Tree< std::string, nikiforova::Tree< long long, std::string > > dict = nikiforova::readAllDictsFromStream(fInput);

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

{

{"print", nikiforova::print},

{"complement", nikiforova::complement},

{"intersect", nikiforova::intersect},

{"union", nikiforova::myUnion}

};

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/S4/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/S4/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/S4/commandsWithDictsOfDicts.h**

#ifndef COMMANDSWITHDICTSOFDICTS\_H

#define COMMANDSWITHDICTSOFDICTS\_H

#include "2-3Tree.h"

namespace nikiforova {

using Dict = Tree< long long, std::string >;

using DictOfDicts = Tree< std::string, Dict >;

DictOfDicts readAllDictsFromStream(std::istream&);

Dict convertStringToDict(std::string&);

std::ostream& doPrint(std::ostream&, const std::string&, const Dict&);

void doComplement(const std::string&, const std::string&, const std::string&, DictOfDicts&);

void doIntersect(const std::string&, const std::string&, const std::string&, DictOfDicts&);

void doUnion(const std::string&, const std::string&, const std::string&, DictOfDicts&);

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

void complement(std::string&, DictOfDicts&);

void intersect(std::string&, DictOfDicts&);

void myUnion(std::string&, DictOfDicts&);

}

#endif

**./<ROOT>/nikiforova.ekaterina/S4/commandsWithDictsOfDicts.cpp**

#include "commandsWithDictsOfDicts.h"

#include <string>

#include <iostream>

#include "2-3Tree.h"

#include "forwardList.h"

#include "errorMessages.h"

#include "operationsWithStrings.h"

nikiforova::DictOfDicts nikiforova::readAllDictsFromStream(std::istream& in)

{

nikiforova::DictOfDicts result;

while (!in.eof())

{

std::string str = "";

std::getline(in, str);

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

nikiforova::DictOfDicts::ConstIterator cIter = result.cfind(nameOfDict);

if (cIter != result.cend())

{

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

}

else

{

nikiforova::Dict temp = nikiforova::convertStringToDict(str);

result.push(nameOfDict, temp);

}

}

return result;

}

nikiforova::Dict nikiforova::convertStringToDict(std::string& str)

{

nikiforova::Dict dict;

while (!str.empty())

{

long long key = 0;

std::string value = "";

std::string temp = "";

if (str[0] == ' ')

{

str.erase(0, 1);

}

temp = str.substr(0, str.find(" ", 1));

if (!nikiforova::isNumber(temp))

{

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

}

key = std::stoll(temp);

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

value = str.substr(0, str.find(" ", 1));

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

dict.push(key, value);

}

return dict;

}

std::ostream& nikiforova::doPrint(std::ostream& out, const std::string& dataset, const Dict& dict)

{

if (dict.isEmpty())

{

return nikiforova::emptyMessage(out);

}

out << dataset;

for (auto&& pair: dict)

{

out << " " << pair.first << " " << pair.second;

}

out << "\n";

return out;

}

void nikiforova::print(std::string& str, DictOfDicts& dicts)

{

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

nikiforova::DictOfDicts::ConstIterator cIter = dicts.cfind(nameOfDict);

if (cIter == dicts.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::Dict dict = cIter->second;

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

}

void nikiforova::doComplement(const std::string& newDataset, const std::string& dataset1, const std::string& dataset2, DictOfDicts& dict)

{

nikiforova::DictOfDicts::ConstIterator cIter1 = dict.cfind(dataset1);

nikiforova::DictOfDicts::ConstIterator cIter2 = dict.cfind(dataset2);

if ((cIter1 == dict.cend()) || cIter2 == dict.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::Dict result = Dict(cIter1->second);

auto listIter = cIter2->second.cbegin();

while (listIter != cIter2->second.cend())

{

auto key = listIter->first;

if (result.find(key) != result.end())

{

result.drop(key);

}

listIter++;

}

if (dict.find(newDataset) != dict.end())

{

dict.drop(newDataset);

}

dict.push(newDataset, result);

}

void nikiforova::complement(std::string& str, DictOfDicts& dict)

{

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

std::string dataset1 = nikiforova::getWord(str);

std::string dataset2 = nikiforova::getWord(str);

nikiforova::doComplement(newDataset, dataset1, dataset2, dict);

}

void nikiforova::doIntersect(const std::string& newDataset, const std::string& dataset1, const std::string& dataset2, DictOfDicts& dict)

{

nikiforova::DictOfDicts::ConstIterator cIter1 = dict.cfind(dataset1);

nikiforova::DictOfDicts::ConstIterator cIter2 = dict.cfind(dataset2);

if ((cIter1 == dict.cend()) || cIter2 == dict.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::Dict result;

auto listIter1 = cIter1->second.cbegin();

while (listIter1 != cIter1->second.cend())

{

auto key1 = listIter1->first;

auto listIter2 = cIter2->second.cbegin();

while (listIter2 != cIter2->second.cend())

{

auto key2 = listIter2->first;

if ((key1 == key2) && (result.find(key1) == result.end()))

{

result.push(key1, listIter1->second);

}

listIter2++;

}

listIter1++;

}

if (dict.find(newDataset) != dict.end())

{

dict.drop(newDataset);

}

dict.push(newDataset, result);

}

void nikiforova::intersect(std::string& str, DictOfDicts& dict)

{

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

std::string dataset1 = nikiforova::getWord(str);

std::string dataset2 = nikiforova::getWord(str);

nikiforova::doIntersect(newDataset, dataset1, dataset2, dict);

}

void nikiforova::doUnion(const std::string& newDataset, const std::string& dataset1, const std::string& dataset2, DictOfDicts& dict)

{

nikiforova::DictOfDicts::ConstIterator cIter1 = dict.cfind(dataset1);

nikiforova::DictOfDicts::ConstIterator cIter2 = dict.cfind(dataset2);

if ((cIter1 == dict.cend()) || cIter2 == dict.cend())

{

nikiforova::invalidCommandMessage(std::cout);

return;

}

nikiforova::Dict result = Dict(dict.get(dataset1));

auto listIter = cIter2->second.cbegin();

while (listIter != cIter2->second.cend())

{

auto key = listIter->first;

if (result.find(key) == result.end())

{

result.push(key, listIter->second);

}

listIter++;

}

if (dict.find(newDataset) != dict.end())

{

dict.drop(newDataset);

}

dict.push(newDataset, result);

}

void nikiforova::myUnion(std::string& str, DictOfDicts& dict)

{

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

std::string dataset1 = nikiforova::getWord(str);

std::string dataset2 = nikiforova::getWord(str);

nikiforova::doUnion(newDataset, dataset1, dataset2, dict);

}

**./<ROOT>/nikiforova.ekaterina/S4/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/S4/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/S4/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/S4/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/S4/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

**./<ROOT>/nikiforova.ekaterina/S4/queue.h**

#ifndef QUEUE\_H

#define QUEUE\_H

#include "list.h"

namespace nikiforova {

template< typename T >

class Queue {

public:

void push(const T&);

void drop();

const T& getFront() const;

const T& getBack() const;

bool isEmpty() const noexcept;

size\_t getLenght() const;

private:

nikiforova::detail::List< T > list\_;

};

template< typename T >

void Queue< T >::push(const T& x)

{

list\_.pushBack(x);

}

template< typename T >

void Queue< T >::drop()

{

if (isEmpty())

{

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

}

list\_.popFront();

}

template< typename T >

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

{

if (isEmpty())

{

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

}

return list\_.getFront();

}

template< typename T >

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

{

if (isEmpty())

{

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

}

return list\_.getBack();

}

template< typename T >

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

{

return list\_.isEmpty();

}

template< typename T >

inline size\_t Queue< T >::getLenght() const

{

return list\_.size();

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S4/stack.h**

#ifndef STACK\_H

#define STACK\_H

#include "list.h"

namespace nikiforova {

template< typename T >

class Stack {

public:

const T& getTop() const;

void push(const T&);

void drop();

bool isEmpty() const noexcept;

size\_t getSize() const;

private:

nikiforova::detail::List< T > list\_;

};

template< typename T >

const T& Stack< T >::getTop() const

{

if (isEmpty())

{

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

}

return list\_.getFront();

}

template< typename T >

void Stack< T >::push(const T& rhs)

{

list\_.pushFront(rhs);

}

template< typename T >

void Stack< T >::drop()

{

if (isEmpty())

{

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

}

list\_.popFront();

}

template< typename T >

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

{

return list\_.isEmpty();

}

template< typename T >

inline size\_t Stack< T >::getSize() const

{

return list\_.size();

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S4/2-3Tree.h**

#ifndef TREE\_H

#define TREE\_H

#include <utility>

#include <cassert>

#include <iterator>

#include <stdexcept>

#include "stack.h"

#include "queue.h"

namespace nikiforova {

namespace detail {

template< typename T >

struct treeNode\_t {

T data\_[3];

size\_t size\_;

treeNode\_t\* parent\_;

treeNode\_t\* first\_;

treeNode\_t\* second\_;

treeNode\_t\* third\_;

treeNode\_t\* fourth\_;

treeNode\_t():

size\_(0),

parent\_(nullptr),

first\_(nullptr),

second\_(nullptr),

third\_(nullptr),

fourth\_(nullptr)

{}

treeNode\_t(const T& x):

size\_(1),

parent\_(nullptr),

first\_(nullptr),

second\_(nullptr),

third\_(nullptr),

fourth\_(nullptr)

{

data\_[0] = x;

}

treeNode\_t(const treeNode\_t& rhs):

size\_(rhs.size\_),

parent\_(nullptr),

first\_(nullptr),

second\_(nullptr),

third\_(nullptr),

fourth\_(nullptr)

{

for (auto i = 0; i < rhs.size\_; i++)

{

data\_[i] = rhs.data\_[i];

}

}

~treeNode\_t() = default;

};

}

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

class Tree {

public:

class ConstIterator;

class Iterator;

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

using thisTreeNode\_t = detail::treeNode\_t< std::pair< Key, Value > >;

Tree();

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

Tree(const Tree&);

Tree(Tree&&) noexcept;

~Tree();

Tree& operator= (const Tree&);

Tree& operator= (Tree&&) noexcept;

bool isEmpty() const noexcept;

void clear();

void swap(Tree&) noexcept;

const Value get(const Key&);

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

void drop(const Key&);

void fixErase(thisTreeNode\_t\*);

void rotate(thisTreeNode\_t\*);

void rotateAndMerge(thisTreeNode\_t\*);

void merge(thisTreeNode\_t\*);

thisTreeNode\_t\* findMinNode(thisTreeNode\_t\*);

thisTreeNode\_t\* findMaxNode(thisTreeNode\_t\*);

void clearNode(thisTreeNode\_t\*);

thisTreeNode\_t\* splitNode(thisTreeNode\_t\*);

void sortKeys(thisTreeNode\_t\*);

Iterator find(const Key&);

thisTreeNode\_t\* searchNode(const Key&) const;

ConstIterator cfind(const Key&) const;

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

pairIterBool insert(const Key&, const Value&);

Iterator begin() const noexcept;

Iterator end() const noexcept;

ConstIterator cbegin() const noexcept;

ConstIterator cend() const noexcept;

template < typename F >

F traverse\_lnr(F f) const;

template < typename F >

F traverse\_rnl(F f) const;

template < typename F >

F traverse\_breadth(F f) const;

class ConstIterator: public std::iterator< std::bidirectional\_iterator\_tag, std::pair< Key, Value > >

{

public:

friend class Tree< Key, Value, Compare >;

ConstIterator(const ConstIterator&) = default;

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

~ConstIterator() = default;

ConstIterator& operator++()

{

assert(treeNode\_ != nullptr);

if (treeNode\_->first\_)

{

if (treeNode\_->size\_ == 1)

{

treeNode\_ = treeNode\_->second\_;

index\_ = 0;

while (treeNode\_->first\_)

{

treeNode\_ = treeNode\_->first\_;

index\_ = 0;

}

return \*this;

}

if (treeNode\_->size\_ == 2)

{

if (index\_ == 0)

{

treeNode\_ = treeNode\_->second\_;

index\_ = 0;

}

else

{

treeNode\_ = treeNode\_->third\_;

index\_ = 0;

}

while (treeNode\_->first\_)

{

treeNode\_ = treeNode\_->first\_;

}

return \*this;

}

}

else

{

if (treeNode\_->size\_ == 2 && index\_ == 0)

{

index\_ = 1;

return \*this;

}

if (!treeNode\_->parent\_)

{

if ((treeNode\_->size\_ == 1 && index\_ == 0) || (treeNode\_->size\_ == 2 && index\_ == 1))

{

\*this = ConstIterator(nullptr);

return \*this;

}

}

else

{

while (treeNode\_ == treeNode\_->parent\_->third\_ || (treeNode\_->parent\_->size\_ == 1 && treeNode\_ == treeNode\_->parent\_->second\_))

{

treeNode\_ = treeNode\_->parent\_;

if (!treeNode\_->parent\_)

{

\*this = ConstIterator(nullptr);

return \*this;

}

}

if (treeNode\_ == treeNode\_->parent\_->first\_)

{

treeNode\_ = treeNode\_->parent\_;

index\_ = 0;

return \*this;

}

else if ((treeNode\_->parent\_->size\_ == 2) && (treeNode\_ == treeNode\_->parent\_->second\_))

{

treeNode\_ = treeNode\_->parent\_;

index\_ = 1;

return \*this;

}

}

}

\*this = ConstIterator(nullptr);

return \*this;

}

ConstIterator operator++(int)

{

assert(treeNode\_ != nullptr);

ConstIterator result(\*this);

++(\*this);

return result;

}

ConstIterator& operator--()

{

assert(treeNode\_ != nullptr);

if (!treeNode\_->second\_)

{

if ((treeNode\_->size\_ == 2) && index\_ == 1)

{

index\_ = 0;

return \*this;

}

else if (treeNode\_ == treeNode\_->parent\_->third\_)

{

treeNode\_ = treeNode\_->parent\_;

index\_ = 1;

}

else if (treeNode\_ == treeNode\_->parent\_->second\_)

{

treeNode\_ = treeNode\_->parent\_;

index\_ = 0;

}

else

{

if (treeNode\_->parent\_->parent\_ && treeNode\_->parent\_ == treeNode\_->parent\_->parent\_->third\_)

{

treeNode\_ = treeNode\_->parent\_->parent\_;

index\_ = 1;

}

else if (treeNode\_->parent\_ == treeNode\_->parent\_->parent\_->second\_)

{

treeNode\_ = treeNode\_->parent\_->parent\_;

index\_ = 0;

}

}

return \*this;

}

else

{

if (index\_ = 0)

{

treeNode\_ = treeNode\_->first\_;

}

else

{

treeNode\_ = treeNode\_->second\_;

}

if (treeNode\_->size\_ == 2)

{

index\_ = 1;

}

else

{

index\_ = 0;

}

ConstIterator iter = treeNode\_;

while (treeNode\_->first\_)

{

iter++;

}

\*this = iter;

return \*this;

}

}

ConstIterator operator--(int)

{

assert(treeNode\_ != nullptr);

ConstIterator result(\*this);

--(\*this);

return result;

}

const std::pair< Key, Value >& operator\*() const

{

assert(treeNode\_ != nullptr);

return treeNode\_->data\_[index\_];

}

const std::pair< Key, Value >\* operator->() const

{

assert(treeNode\_ != nullptr);

return std::addressof(treeNode\_->data\_[index\_]);

}

bool operator==(const ConstIterator& rhs) const

{

return (treeNode\_ == rhs.treeNode\_) && ((index\_ == rhs.index\_) || (treeNode\_ == nullptr));

}

bool operator!=(const ConstIterator& rhs) const

{

return !(rhs == \*this);

}

private:

const thisTreeNode\_t\* treeNode\_;

int index\_;

ConstIterator(thisTreeNode\_t\* rhs)

{

treeNode\_ = rhs;

index\_ = 0;

}

};

class Iterator: public std::iterator< std::bidirectional\_iterator\_tag, std::pair< Key, Value > >

{

public:

friend class Tree< Key, Value, Compare >;

Iterator(const Iterator&) = default;

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

~Iterator() = default;

Iterator(const ConstIterator& x):

cIter\_(x)

{}

Iterator& operator++()

{

++cIter\_;

return \*this;

}

Iterator operator++(int)

{

Iterator result(\*this);

++(\*this);

return result;

}

Iterator& operator--()

{

--cIter\_;

return \*this;

}

Iterator operator--(int)

{

Iterator result(\*this);

--(\*this);

return result;

}

std::pair< Key, Value >& operator\*() const

{

return const\_cast<std::pair< Key, Value >&>(\*cIter\_);

}

std::pair< Key, Value >\* operator->() const

{

return const\_cast<std::pair< Key, Value >\*>(std::addressof(\*cIter\_));

}

bool operator==(const Iterator& rhs) const

{

return cIter\_ == rhs.cIter\_;

}

bool operator!=(const Iterator& rhs) const

{

return !(rhs == \*this);

}

private:

ConstIterator cIter\_;

};

private:

thisTreeNode\_t\* root\_;

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

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

};

template< typename Key, typename Value, typename Compare >

Tree< Key, Value, Compare >::Tree():

root\_(nullptr)

{}

template< typename Key, typename Value, typename Compare >

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

root\_(nullptr)

{

for (auto&& pair: list)

{

insert(pair);

}

}

template< typename Key, typename Value, typename Compare >

Tree< Key, Value, Compare >::Tree(const Tree& rhs):

root\_(nullptr)

{

if (!rhs.isEmpty())

{

Iterator iter = rhs.begin();

try

{

while (iter != rhs.end())

{

insert(\*iter);

iter++;

}

}

catch (...)

{

clear();

throw;

}

}

}

template< typename Key, typename Value, typename Compare >

Tree< Key, Value, Compare >::Tree(Tree&& rhs) noexcept:

root\_(rhs.root\_)

{

rhs.root\_ = nullptr;

}

template< typename Key, typename Value, typename Compare >

Tree< Key, Value, Compare >& Tree< Key, Value, Compare >::operator=(const Tree< Key, Value, Compare >& rhs)

{

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

{

Tree< Key, Value, Compare > temp(rhs);

swap(temp);

}

return \*this;

}

template< typename Key, typename Value, typename Compare >

Tree< Key, Value, Compare >& Tree< Key, Value, Compare >::operator=(Tree< Key, Value, Compare >&& rhs) noexcept

{

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

{

Tree< Key, Value, Compare > temp(std::move(rhs));

swap(temp);

}

return \*this;

}

template< typename Key, typename Value, typename Compare >

Tree< Key, Value, Compare >::~Tree()

{

clear();

}

template< typename Key, typename Value, typename Compare >

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

{

return !root\_;

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::clear()

{

clearNode(root\_);

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::swap(Tree& rhs) noexcept

{

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

}

template< typename Key, typename Value, typename Compare >

const Value Tree< Key, Value, Compare >::get(const Key& key)

{

auto iter = cfind(key);

return (\*iter).second;

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::push(const Key& key, const Value& val)

{

insert(key, val);

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::drop(const Key& key)

{

thisTreeNode\_t\* node = searchNode(key);

if (node == nullptr)

{

return;

}

else if (node->size\_ == 2 && !node->first\_)

{

if (node->data\_[0].first == key)

{

node->data\_[0] = node->data\_[1];

}

node->size\_--;

return;

}

if (node->first\_)

{

thisTreeNode\_t\* minNode = nullptr;

if (key == node->data\_[0].first)

{

minNode = findMinNode(node->second\_);

}

else

{

minNode = findMinNode(node->third\_);

}

std::pair< Key, Value >& data = (key == node->data\_[0].first) ? node->data\_[0] : node->data\_[1];

auto temp = minNode->data\_[0];

minNode->data\_[0] = data;

data = temp;

node = minNode;

}

if (node->size\_ == 2 && key == node->data\_[0].first)

{

node->data\_[0] = node->data\_[1];

}

node->size\_--;

if (node->size\_ == 1 && !node->first\_)

{

return;

}

if (node->size\_ == 0)

{

fixErase(node);

}

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::fixErase(thisTreeNode\_t\* node)

{

if (node->size\_ == 0 && node->parent\_ == nullptr && !node->first\_)

{

delete node;

node = nullptr;

root\_ = nullptr;

}

else

{

thisTreeNode\_t\* parent = node->parent\_;

if (node->parent\_)

{

if (parent->size\_ == 1)

{

if (node == parent->second\_ && parent->first\_->size\_ == 1)

{

merge(node);

}

else if (node == parent->first\_ && parent->second\_->size\_ == 1)

{

merge(node);

}

else if (node == parent->first\_ && parent->second\_->size\_ == 2)

{

rotate(node);

}

else if (node == parent->second\_ && parent->first\_->size\_ == 2)

{

rotate(node);

}

}

else if (parent->size\_ == 2)

{

if (parent->second\_->size\_ == 1 || parent->second\_->size\_ == 0)

{

if (node == parent->first\_ || node == parent->third\_)

{

rotateAndMerge(node);

return;

}

else if (parent->first\_->size\_ == 1 && parent->third\_->size\_ == 1)

{

rotateAndMerge(node);

return;

}

}

rotate(node);

}

}

else

{

thisTreeNode\_t \*tmp = nullptr;

if (node->first\_ != nullptr)

{

tmp = node->first\_;

}

else

{

tmp = node->second\_;

}

tmp->parent\_ = nullptr;

delete node;

root\_ = tmp;

}

}

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::rotate(thisTreeNode\_t\* node)

{

thisTreeNode\_t\* parent = node->parent\_;

if (node == parent->first\_)

{

node->data\_[0] = parent->data\_[0];

node->size\_++;

parent->data\_[0] = parent->second\_->data\_[0];

parent->second\_->data\_[0] = parent->second\_->data\_[1];

parent->second\_->size\_--;

if (parent->second\_->first\_)

{

node->second\_ = parent->second\_->first\_;

node->second\_->parent\_ = node;

parent->second\_->first\_ = parent->second\_->second\_;

parent->second\_->second\_ = parent->second\_->third\_;

parent->second\_->third\_ = nullptr;

}

}

else if (node == parent->second\_)

{

if (parent->first\_->size\_ == 2)

{

node->data\_[0] = parent->data\_[0];

node->size\_++;

node->parent\_->data\_[0] = parent->first\_->data\_[1];

parent->first\_->size\_--;

if (node->first\_)

{

node->second\_ = node->first\_;

node->first\_ = parent->first\_->third\_;

if (node->first\_)

{

node->first\_->parent\_ = node;

}

parent->first\_->third\_ = nullptr;

}

}

else if (parent->third\_->size\_ == 2)

{

node->data\_[0] = parent->data\_[1];

node->size\_++;

parent->data\_[1] = parent->third\_->data\_[0];

parent->third\_->data\_[0] = parent->third\_->data\_[1];

parent->third\_->size\_--;

if (node->first\_)

{

node->second\_ = parent->third\_->first\_;

node->second\_->parent\_ = node;

parent->third\_->first\_ = parent->third\_->second\_;

parent->third\_->second\_ = parent->third\_->third\_;

parent->third\_->third\_ = nullptr;

}

}

}

else

{

node->data\_[0] = parent->data\_[1];

node->size\_++;

parent->data\_[1] = parent->second\_->data\_[1];

parent->second\_->size\_--;

if (node->first\_)

{

node->second\_ = node->first\_;

node->first\_ = parent->second\_->third\_;

node->first\_->parent\_ = node;

parent->third\_ = nullptr;

}

}

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::rotateAndMerge(thisTreeNode\_t\* node)

{

thisTreeNode\_t\* parent = node->parent\_;

if (node == parent->first\_)

{

parent->second\_->data\_[1] = parent->second\_->data\_[0];

parent->second\_->data\_[0] = parent->data\_[0];

parent->data\_[0] = parent->data\_[1];

parent->second\_->size\_++;

parent->second\_->third\_ = parent->second\_->second\_;

parent->second\_->second\_ = parent->second\_->first\_;

if (node->first\_)

{

parent->second\_->first\_ = node->first\_;

parent->second\_->first\_->parent\_ = parent->second\_;

}

delete node;

parent->first\_ = parent->second\_;

parent->second\_ = parent->third\_;

}

else

{

if (node == parent->second\_)

{

parent->third\_->data\_[1] = parent->third\_->data\_[0];

parent->third\_->data\_[0] = parent->data\_[1];

parent->third\_->parent\_ = parent;

if (node->first\_)

{

parent->third\_->third\_ = parent->third\_->second\_;

parent->third\_->second\_ = parent->third\_->first\_;

parent->third\_->first\_ = node->first\_;

parent->third\_->first\_->parent\_ = parent->third\_;

}

delete node;

parent->second\_ = parent->third\_;

}

else

{

delete node;

parent->second\_->data\_[1] = parent->data\_[1];

}

parent->second\_->size\_++;

}

parent->size\_--;

parent->third\_ = nullptr;

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::merge(thisTreeNode\_t\* node)

{

thisTreeNode\_t\* parent = node->parent\_;

if (node == parent->first\_)

{

parent->second\_->data\_[1] = parent->second\_->data\_[0];

parent->second\_->data\_[0] = parent->data\_[0];

parent->size\_--;

parent->second\_->size\_++;

if (node->first\_)

{

parent->second\_->third\_ = parent->second\_->second\_;

parent->second\_->second\_ = parent->second\_->first\_;

parent->second\_->first\_ = node->first\_;

parent->second\_->first\_->parent\_ = parent->second\_;

node->first\_ = nullptr;

}

delete node;

parent->first\_ = parent->second\_;

parent->second\_ = nullptr;

}

else

{

parent->first\_->data\_[1]=parent->data\_[0];

parent->first\_->size\_++;

parent->size\_--;

if (node->first\_)

{

parent->first\_->third\_ = node->first\_;

parent->first\_->third\_->parent\_ = parent->first\_;

node->first\_ = nullptr;

}

delete node;

parent->second\_ = nullptr;

}

return fixErase(parent);

}

template< typename Key, typename Value, typename Compare >

typename Tree< Key, Value, Compare >::thisTreeNode\_t\* Tree< Key, Value, Compare >::findMinNode(thisTreeNode\_t\* node)

{

thisTreeNode\_t\* res = node;

if (!res)

{

return res;

}

while (res->first\_)

{

res = res->first\_;

}

return res;

}

template< typename Key, typename Value, typename Compare >

typename Tree< Key, Value, Compare >::thisTreeNode\_t\* Tree< Key, Value, Compare >::findMaxNode(thisTreeNode\_t\* node)

{

thisTreeNode\_t\* res = node;

while (res->first\_)

{

if (res->size\_ == 2)

{

res = res->third\_;

}

else

{

res = res->second\_;

}

}

return res;

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::clearNode(thisTreeNode\_t\* node)

{

if (node)

{

clearNode(node->first\_);

clearNode(node->second\_);

clearNode(node->third\_);

delete node;

}

}

template< typename Key, typename Value, typename Compare >

typename Tree< Key, Value, Compare >::thisTreeNode\_t\* Tree< Key, Value, Compare >::splitNode(thisTreeNode\_t\* node)

{

if (node->size\_ != 3)

{

return node;

}

thisTreeNode\_t\* left = new thisTreeNode\_t;

left->data\_[0] = node->data\_[0];

left->parent\_ = node->parent\_;

left->first\_ = node->first\_;

left->second\_ = node->second\_;

if (left->first\_)

{

left->first\_->parent\_ = left;

}

if (left->second\_)

{

left->second\_->parent\_ = left;

}

left->size\_ = 1;

thisTreeNode\_t\* right = new thisTreeNode\_t;

right->data\_[0] = node->data\_[2];

right->parent\_ = node->parent\_;

right->first\_ = node->third\_;

right->second\_ = node->fourth\_;

if (right->first\_)

{

right->first\_->parent\_ = right;

}

if (right->second\_)

{

right->second\_->parent\_ = right;

}

right->size\_ = 1;

if (node->parent\_)

{

node->parent\_->data\_[node->parent\_->size\_] = node->data\_[1];

node->parent\_->size\_++;

sortKeys(node->parent\_);

if (node == node->parent\_->first\_)

{

node->parent\_->first\_ = left;

node->parent\_->fourth\_ = node->parent\_->third\_;

node->parent\_->third\_ = node->parent\_->second\_;

node->parent\_->second\_ = right;

}

else if (node == node->parent\_->second\_)

{

node->parent\_->second\_ = left;

node->parent\_->fourth\_ = node->parent\_->third\_;

node->parent\_->third\_ = right;

}

else

{

node->parent\_->third\_ = left;

node->parent\_->fourth\_ = right;

}

thisTreeNode\_t\* temp = node->parent\_;

delete node;

return splitNode(temp);

}

left->parent\_ = node;

right->parent\_ = node;

node->data\_[0] = node->data\_[1];

node->size\_ = 1;

node->parent\_ = nullptr;

node->first\_ = left;

node->second\_ = right;

node->third\_ = nullptr;

node->fourth\_ = nullptr;

return node;

}

template< typename Key, typename Value, typename Compare >

void Tree< Key, Value, Compare >::sortKeys(thisTreeNode\_t\* node)

{

if (!isLess(node->data\_[0].first, node->data\_[1].first))

{

std::swap(node->data\_[0], node->data\_[1]);

}

if (node->size\_ == 3)

{

if (!isLess(node->data\_[0].first, node->data\_[2].first))

{

std::swap(node->data\_[0], node->data\_[2]);

}

if (!isLess(node->data\_[1].first, node->data\_[2].first))

{

std::swap(node->data\_[1], node->data\_[2]);

}

}

}

template< typename Key, typename Value, typename Compare >

typename Tree< Key, Value, Compare >::Iterator Tree< Key, Value, Compare >::find(const Key& key)

{

return Iterator(cfind(key));

}

template< typename Key, typename Value, typename Compare >

typename Tree< Key, Value, Compare >::thisTreeNode\_t\* Tree< Key, Value, Compare >::searchNode(const Key& key) const

{

if (!root\_)

{

return root\_;

}

thisTreeNode\_t\* node = root\_;

while (node && !(isEqual(key, node->data\_[0].first) || (node->size\_ == 2 && isEqual(key, node->data\_[1].first))))

{

if (isLess(key, node->data\_[0].first))

{

node = node->first\_;

}

else if (node->size\_ == 1 && !isLess(key, node->data\_[0].first))

{

node = node->second\_;

}

else if (node->size\_ == 2 && isLess(key, node->data\_[1].first))

{

node = node->second\_;

}

else if (node->size\_ == 2 && !isLess(key, node->data\_[1].first))

{

if (node->first\_)

{

node = node->third\_;

}

else

{

return node;

}

}

}

return node;

}

template< typename Key, typename Value, typename Compare >

typename Tree< Key, Value, Compare >::ConstIterator Tree< Key, Value, Compare >::cfind(const Key& key) const

{

if (!root\_)

{

return cend();

}

thisTreeNode\_t\* node = root\_;

while (node)

{

if (isEqual(key, node->data\_[0].first))

{

auto res = ConstIterator(node);

res.index\_ = 0;

return res;

}

else if (isEqual(key, node->data\_[1].first))

{

auto res = ConstIterator(node);

res.index\_ = 1;

return res;

}

if (isLess(key, node->data\_[0].first))

{

node = node->first\_;

}

else if ((node->size\_ == 1 && !isLess(key, node->data\_[0].first)) || (node->size\_ == 2 && isLess(key, node->data\_[1].first)))

{

node = node->second\_;

}

else if (node->size\_ == 2 && !isLess(key, node->data\_[1].first))

{

if (node->first\_)

{

node = node->third\_;

}

else

{

return ConstIterator(nullptr);

}

}

}

return ConstIterator(nullptr);

}

template< typename Key, typename Value, typename Compare >

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

{

return insert(data.first, data.second);

}

template< typename Key, typename Value, typename Compare >

typename Tree< Key, Value, Compare >::pairIterBool Tree< Key, Value, Compare >::insert(const Key& key, const Value& value)

{

std::pair< Key, Value > pair = std::make\_pair(key, value);

thisTreeNode\_t\* prevNode = root\_;

if (isEmpty())

{

root\_ = new thisTreeNode\_t(pair);

return std::make\_pair(Iterator(root\_), true);

}

else

{

auto size = prevNode->size\_;

while (prevNode != nullptr && !(isEqual(key, prevNode->data\_[0].first) || (size == 2 && isEqual(key, prevNode->data\_[1].first))))

{

if (isLess(key, prevNode->data\_[0].first))

{

if (!prevNode->first\_)

{

prevNode->data\_[prevNode->size\_] = pair;

prevNode->size\_++;

sortKeys(prevNode);

splitNode(prevNode);

return std::make\_pair(Iterator(prevNode), true);

}

else

{

prevNode = prevNode->first\_;

size = prevNode->size\_;

}

}

else if ((!isLess(key, prevNode->data\_[0].first) && isLess(key, prevNode->data\_[1].first) && size == 2 ) || size == 1)

{

if (!prevNode->first\_)

{

prevNode->data\_[prevNode->size\_] = pair;

prevNode->size\_++;

sortKeys(prevNode);

splitNode(prevNode);

return std::make\_pair(Iterator(prevNode), true);

}

else

{

prevNode = prevNode->second\_;

size = prevNode->size\_;

}

}

else if (size == 2 && !isLess(key, prevNode->data\_[1].first))

{

if (!prevNode->first\_)

{

prevNode->data\_[prevNode->size\_] = pair;

prevNode->size\_++;

sortKeys(prevNode);

splitNode(prevNode);

return std::make\_pair(Iterator(prevNode), true);

}

else

{

prevNode = prevNode->third\_;

size = prevNode->size\_;

}

}

}

}

return std::make\_pair(Iterator(prevNode), false);

}

template< typename Key, typename Value, typename Compare >

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

{

return Iterator(cbegin());

}

template< typename Key, typename Value, typename Compare >

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

{

return Iterator(cend());

}

template< typename Key, typename Value, typename Compare >

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

{

if (!root\_)

{

return ConstIterator(root\_);

}

else

{

thisTreeNode\_t\* temp = root\_;

while (temp->first\_)

{

temp = temp->first\_;

}

return ConstIterator(temp);

}

}

template< typename Key, typename Value, typename Compare >

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

{

return ConstIterator(nullptr);

}

template< typename Key, typename Value, typename Compare >

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

{

return Compare()(lhs, rhs);

}

template< typename Key, typename Value, typename Compare >

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

{

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

}

template< typename Key, typename Value, typename Compare >

template< typename F >

F Tree< Key, Value, Compare >::traverse\_lnr(F f) const

{

auto iter = cbegin();

while (iter != cend())

{

f(\*iter);

iter++;

}

return f;

}

template< typename Key, typename Value, typename Compare >

template< typename F >

F Tree< Key, Value, Compare >::traverse\_rnl(F f) const

{

auto iter = cbegin();

Stack< std::pair< Key, Value > > stack;

while (iter != cend())

{

stack.push(\*iter);

iter++;

}

while (!stack.isEmpty())

{

f(stack.getTop());

stack.drop();

}

return f;

}

template< typename Key, typename Value, typename Compare >

template< typename F >

F Tree< Key, Value, Compare >::traverse\_breadth(F f) const

{

if (!root\_)

{

return f;

}

Queue< thisTreeNode\_t\* > queue;

queue.push(root\_);

while (!queue.isEmpty())

{

thisTreeNode\_t\* tempNode = queue.getFront();

f(tempNode->data\_[0]);

if (tempNode->size\_ == 2)

{

f(tempNode->data\_[1]);

}

if (tempNode->first\_)

{

queue.push(tempNode->first\_);

}

if (tempNode->second\_)

{

queue.push(tempNode->second\_);

}

if (tempNode->third\_)

{

queue.push(tempNode->third\_);

}

queue.drop();

}

return f;

}

}

#endif