ФИО: Никифорова Екатерина Алексеевна
Группа: 3530904/10001
Лабораторная работа: FA «1.3.2. Словарь. 2-3 дерево»

Постановка задачи

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

FA «1.3.2. Словарь. 2-3 дерево»

1. Разработать и реализовать алгоритм формирования словаря; построить лексикографическое дерево из всех слов, встречающихся в тексте. Для реализации задания использовать 2-3 дерево
2. Параметром командной строки задаётся имя файла filename, который содержит внутри себя данные по некоторому количеству списков
3. Файл со списками имеет следующий вид:

```
<name of dict 1> <word 1 > < word 2> < word 3> < word 4> < word 5> ...
<name of dict 2> <word 1 > < word 2> < word 3> ...
...
```

   - name of dict представляет собой имя словаря; word – слово, элемент списка.

   Например input.txt:

```
first keyboard carrot key car knowledge mouse
second name carrot key
third
```

   - Пустые строки игнорируются. Данные в строке разделены ровно одним пробелом
4. Реализуемая программа считывает данные словарей из файла и выполнять команды, принимаемые от пользователя со стандартного ввода.
   - Каждая строка содержит ровно одну команду. Должны поддерживаться следующие команды, все аргументы которых обязательны:

```
print [имя словаря]
contain [имя словаря] [слово]
words [имя словаря] [префикс]
union [имя нового словаря] [словарь1] [словарь2]
intersect [имя нового словаря] [словарь1] [словарь2]
complement [имя нового словаря] [словарь1] [словарь2]
save [имя словаря] [файл]
```

   - print - выводит все слова словаря в лексикографичкском порядке
   - contain - выводит True, если слово есть в словаре, false если нет
   - words - выводит все слова из словаря, начинающиеся на указанный префикс
   - union - создаёт словарь, объединяя словарь 1 и 2
   - intersect - создаёт словарь из пересечения словаря 1 и 2
   - complement - создаёт словарь вычитанием словаря2 из словаря1
   - save - запись словаря в файл

   - Если команда по каким-то причинам некорректна, то команда выводит сообщение <INVALID COMMAND>
   - Признаком конца ввода команд является EOF

Приёмочные тесты

#	Описание	Результат
1	Неверное количество аргументов командной строки	**Expected:** Сообщение об ошибке: «Incorrect number of arguments» и ненулевой код возврата
2	В файле два словаря с одинаковым именем	**Expected:** Сообщение об ошибке: «Dictionary with the same name already exists» и ненулевой код возврата
3	Вывод всех слов указанного словаря	**Input:** print first **Expected:** first car carrot key keyboard knowledge mouse
		Input: print third **Expected:** <EMPTY>
4	Вывод всех слов словаря, начинающихся на указанный префикс	**Input:** words first ca **Expected:** car carrot
		Input: words first as **Expected:** Words with prefix = as doesn't exist
5	Проверка есть ли слово в словаре	**Input:** contain first car **Expected:** true
		Input: contain first card **Expected:** False
6	Пересечение словарей	**Input:** intersect fourth first second print fourth **Expected:** fourth carrot key
7	Объединение словарей	**Input:** union fifth first second print fifth **Expected:** fifth car carrot key keyboard knowledge mouse name
8	Вычитание словарей	**Input:** complement sixth first second print sixth **Expected:** sixth car keyboard knowledge mouse
9	Сохранение словаря в файл	**Input:** intersect fourth first second save fourth input.txt **Expected in “input.txt”:** first keyboard carrot key car knowledge mouse second name carrot key third fourth carrot key
		Input: intersect fifth first second save fifth newFile.txt **Expected in “newFile.txt”:** fifth carrot key

Исходные тексты программы

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

./<ROOT>/Nikiforova.ekaterina/FA/2-3Set.h

```
#ifndef TREESET_H
#define TREESET_H
#include "2-3Tree.h"

namespace nikiforova {

  template < typename Key, typename Value = Key, typename Compare = std::less< Key > >
  class TreeSet {
  public:
    using Iterator = typename Tree< Key, Value, Compare >::Iterator;
    using ConstIterator = typename Tree< Key, Value, Compare >::ConstIterator;
    using pairIterBool = typename std::pair< Iterator, bool >;
    using thisTreeNode_t = typename detail::treeNode_t< std::pair< Key, Value > >;
    pairIterBool insert(const Value&);
    pairIterBool insert(const std::pair< Key, Value >&);
    pairIterBool insert(const Key&, const Value&);
    void push(const Value& val);
    void push(const Key&, const Value&);
    void drop(const Key&);
    bool isEmpty() const;
    Iterator find(const Key&);
    ConstIterator cfind(const Key&) const;
    Iterator begin() const noexcept;
    Iterator end() const noexcept;
    ConstIterator cbegin() const noexcept;
    ConstIterator cend() const noexcept;
    bool isLess(const Key& lhs, const Key& rhs) const;
    bool isEqual(const Key& lhs, const Key& rhs) const;

  private:
    Tree< Key, Value, Compare > tree_;
    thisTreeNode_t* searchNode(const Key&) const;
  };

  template < typename Key, typename Value, typename Compare >
  bool TreeSet< Key, Value, Compare >::isLess(const Key& lhs, const Key& rhs) const
  {
    return Compare()(lhs, rhs);
  }

  template < typename Key, typename Value, typename Compare >
  bool TreeSet< 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 >
  typename TreeSet< Key, Value, Compare >::pairIterBool TreeSet< Key, Value, Compare
>::insert(const Value& val)
  {
    return tree_.insert(val, val);
  }

  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::pairIterBool TreeSet< Key, Value, Compare
>::insert(const std::pair< Key, Value >& data)
  {
    return tree_.insert(data);
  }
```

```
  template< typename Key, typename Value, typename Compare >
  void TreeSet< Key, Value, Compare >::push(const Value& val)
  {
    tree_.push(val, val);
  }

  template< typename Key, typename Value, typename Compare >
  void TreeSet< Key, Value, Compare >::push(const Key& key, const Value& val)
  {
    tree_.push(key, val);
  }

  template< typename Key, typename Value, typename Compare >
  void nikiforova::TreeSet< Key, Value, Compare >::drop(const Key& key)
  {
    tree_.drop(key);
  }

  template< typename Key, typename Value, typename Compare >
  bool TreeSet< Key, Value, Compare >::isEmpty() const
  {
    return tree_.isEmpty();
  }

  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::Iterator nikiforova::TreeSet< Key, Value, Compare
>::find(const Key& key)
  {
    return tree_.find(key);
  }

  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::ConstIterator nikiforova::TreeSet< Key, Value, Compare
>::cfind(const Key& key) const
  {
    return tree_.cfind(key);
  }

  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::thisTreeNode_t* nikiforova::TreeSet< Key, Value,
Compare >::searchNode(const Key& key) const
  {
    return tree_.searchNode(key);
  }

  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::Iterator TreeSet< Key, Value, Compare >::begin() const
noexcept
  {
    return tree_.begin();
  }

  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::Iterator TreeSet< Key, Value, Compare >::end() const
noexcept
  {
    return tree_.end();
  }

  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::ConstIterator TreeSet< Key, Value, Compare >::cbegin()
const noexcept
  {
    return tree_.cbegin();
  }
```

```
  template< typename Key, typename Value, typename Compare >
  typename TreeSet< Key, Value, Compare >::ConstIterator TreeSet< Key, Value, Compare >::cend()
const noexcept
  {
    return tree_.cend();
  }

}

#endif
```

./<ROOT>/Nikiforova.ekaterina/FA/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*);
    void clearNode(thisTreeNode_t*);
    void sortKeys(thisTreeNode_t*);
    Iterator find(const Key&);
    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;
  thisTreeNode_t* findMinNode(thisTreeNode_t*);
  thisTreeNode_t* findMaxNode(thisTreeNode_t*);
  thisTreeNode_t* splitNode(thisTreeNode_t*);
  thisTreeNode_t* searchNode(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
```

./<ROOT>/Nikiforova.ekaterina/FA/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/FA/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/FA/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/FA/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/FA/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/FA/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/FA/errorMessages.cpp

```
#include "errorMessages.h"

std::ostream& nikiforova::invalidCommandMessage(std::ostream& out)
{
  return out << "<INVALID COMMAND>";
}
std::ostream& nikiforova::emptyMessage(std::ostream& out)
{
  return out << "<EMPTY>";
}
```

./<ROOT>/Nikiforova.ekaterina/FA/operationsWithStrings.h

```
#ifndef DIFFERENTUSEFULFUNCTIONS_H
#define DIFFERENTUSEFULFUNCTIONS_H
#include <string>

namespace nikiforova {
  bool isNumber(const std::string&);
  std::string getWord(std::string&);
  bool hasPrefix(const std::string&, const std::string&);
}
#endif
```

./<ROOT>/Nikiforova.ekaterina/FA/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;
}

bool nikiforova::hasPrefix(const std::string& str, const std::string& pref)
{
  bool res = 1;
  for (auto n = 0; n < str.length(); n++)
  {
    if (n >= pref.length())
    {
      break;
    }
    if(str[n] != pref[n])
    {
      return 0;
    }
  }
  return 1;
}
```

./<ROOT>/Nikiforova.ekaterina/FA/commandsWithDictsOfDicts.h

```
#ifndef COMMANDSWITHDICTSOFDICTS_H
#define COMMANDSWITHDICTSOFDICTS_H
#include "2-3Tree.h"
#include "2-3Set.h"

namespace nikiforova {
```

```
  using Dict = TreeSet< 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&);
  std::ostream& getWordsWithPrefix(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&);
  std::ostream& isContain(std::ostream&, const std::string&, const Dict&);
  void doSave(const std::string&, const std::string&, const Dict&);

  void print(std::string&, const DictOfDicts&);
  void complement(std::string&, DictOfDicts&);
  void intersect(std::string&, DictOfDicts&);
  void myUnion(std::string&, DictOfDicts&);
  void wordsWithPrefix(std::string&, const DictOfDicts&);
  void contain(std::string&, const DictOfDicts&);
  void save(std::string&, const DictOfDicts&);

}
#endif
```

./<ROOT>/Nikiforova.ekaterina/FA/commandsWithDictsOfDicts.cpp

```
#include "commandsWithDictsOfDicts.h"
#include <string>
#include <iostream>
#include <fstream>
#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);
    if (!str.empty())
    {
      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())
  {
    std::string temp = nikiforova::getWord(str);
    dict.push(temp);
```

```
  }
  return dict;
}

std::ostream& nikiforova::doPrint(std::ostream& out, const std::string& dataset, const Dict&
dict)
{
  if (dict.isEmpty())
  {
    nikiforova::emptyMessage(out);
    out << "\n";
    return out;
  }
  out << dataset;
  for (auto&& pair: dict)
  {
    out << " " << pair.second;
  }
  out << "\n";
  return out;
}

void nikiforova::print(std::string& str, const DictOfDicts& dicts)
{
  std::string nameOfDict = nikiforova::getWord(str);
  nikiforova::DictOfDicts::ConstIterator cIter = dicts.cfind(nameOfDict);
  if (cIter == dicts.cend())
  {
    nikiforova::invalidCommandMessage(std::cout);
    std::cout << "\n";
    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);
    std::cout << "\n";
    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);
    std::cout << "\n";
    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(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);
    std::cout << "\n";
    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(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);
}

std::ostream& nikiforova::getWordsWithPrefix(std::ostream& out, const std::string& prefix, const
Dict& dict)
{
  if (dict.isEmpty())
  {
    nikiforova::emptyMessage(out);
    out << "\n";
    return out;
  }
  TreeSet< std::string >::Iterator iter = dict.begin();
  std::string temp = iter->first;
  while (dict.isLess(temp, prefix))
  {
    iter++;
    temp = iter->first;
  }
  if (!nikiforova::hasPrefix(temp, prefix))
  {
    return out << "Words with prefix = " << prefix << " doesn't exist" << "\n";
  }
  bool flag = 1;
  while (!dict.isLess(temp, prefix))
  {
    if (flag)
    {
      out << temp;
      flag = 0;
    }
    else
    {
      out << " " << temp;
    }
    iter++;
    if (iter == dict.end())
    {
      return out << "\n";
    }
    temp = iter->first;
    if (!nikiforova::hasPrefix(temp, prefix))
    {
      return out << "\n";
    }
```

```
  }
  return out;
}

void nikiforova::wordsWithPrefix(std::string& str, const DictOfDicts& dicts)
{
  std::string nameOfDict = nikiforova::getWord(str);
  std::string prefix = nikiforova::getWord(str);
  nikiforova::DictOfDicts::ConstIterator cIter = dicts.cfind(nameOfDict);
  if (cIter == dicts.cend())
  {
    nikiforova::invalidCommandMessage(std::cout);
    std::cout << "\n";
    return;
  }
  nikiforova::Dict dict = cIter->second;
  nikiforova::getWordsWithPrefix(std::cout, prefix, dict);
}

std::ostream& nikiforova::isContain(std::ostream& out, const std::string& word, const Dict& dict)
{
  TreeSet< std::string >::ConstIterator iter = dict.cfind(word);
  return out << std::boolalpha << (iter != dict.cend()) << "\n";
}

void nikiforova::contain(std::string& str, const DictOfDicts& dicts)
{
  std::string nameOfDict = nikiforova::getWord(str);
  std::string word = nikiforova::getWord(str);
  nikiforova::DictOfDicts::ConstIterator cIter = dicts.cfind(nameOfDict);
  if (cIter == dicts.cend())
  {
    nikiforova::invalidCommandMessage(std::cout);
    std::cout << "\n";
    return;
  }
  nikiforova::Dict dict = cIter->second;
  nikiforova::isContain(std::cout, word, dict);
}

void nikiforova::save(std::string& str, const DictOfDicts& dicts)
{
  std::string nameOfDict = nikiforova::getWord(str);
  std::string nameOfFile = nikiforova::getWord(str);
  nikiforova::DictOfDicts::ConstIterator cIter = dicts.cfind(nameOfDict);
  if (cIter == dicts.cend())
  {
    nikiforova::invalidCommandMessage(std::cout);
    std::cout << "\n";
    return;
  }
  nikiforova::Dict dict = cIter->second;
  nikiforova::doSave(nameOfDict, nameOfFile, dict);
}

void nikiforova::doSave(const std::string& nameOfDict, const std::string& nameOfFile, const Dict&
dict)
{
  std::fstream fOut(nameOfFile, std::ios::app);
  fOut << "\n";
  nikiforova::doPrint(fOut, nameOfDict, dict);
  fOut.close();
}
```

./<ROOT>/Nikiforova.ekaterina/FA/main.cpp

```
#include <iostream>
#include <string>
#include <fstream>
#include <functional>
#include "2-3Tree.h"
#include "2-3Set.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::TreeSet< std::string > > dict = nikiforova::readAllDictsFromStream(fInput);
    using command_t = std::function< void(std::string&, nikiforova::DictOfDicts&) >;
    using constCommand_t = std::function< void(std::string&, const nikiforova::DictOfDicts&) >;

    nikiforova::Dictionary< std::string, command_t > commands
    {
      {"complement", nikiforova::complement},
      {"intersect", nikiforova::intersect},
      {"union", nikiforova::myUnion}
    };
    nikiforova::Dictionary< std::string, constCommand_t > constCommands
    {
      {"print", nikiforova::print},
      {"words", nikiforova::wordsWithPrefix},
      {"contain", nikiforova::contain},
      {"save", nikiforova::save}
    };
    while (!std::cin.eof())
    {
      std::string command = "";
      std::cin >> command;
      if (command.empty())
      {
        continue;
      }
      auto iter = commands.find(command);
      if (iter == commands.end())
      {
        auto cIter = constCommands.find(command);
        if (cIter == constCommands.end())
        {
          std::string temp = "";
          std::getline(std::cin, temp);
          nikiforova::invalidCommandMessage(std::cout);
          std::cout << "\n";
        }
        else
        {
```

```
          std::string str = "";
          std::getline(std::cin, str);
          cIter->second(str, dict);
        }
      }
      else
      {
        std::string str = "";
        std::getline(std::cin, str);
        iter->second(str, dict);
      }
    }
  }
  catch (const std::exception& e)
  {
    std::cerr << e.what();
  }
  return 0;
}
```