Часть 1. Дек, как совокупность классов

Цель работы: разработать контейнер типа дек, как совокупность классов, предусмотреть исключения, разработать графический интерфейс тестирующей программы.

Ход работы: диаграмма классов изображена на рисунке 1

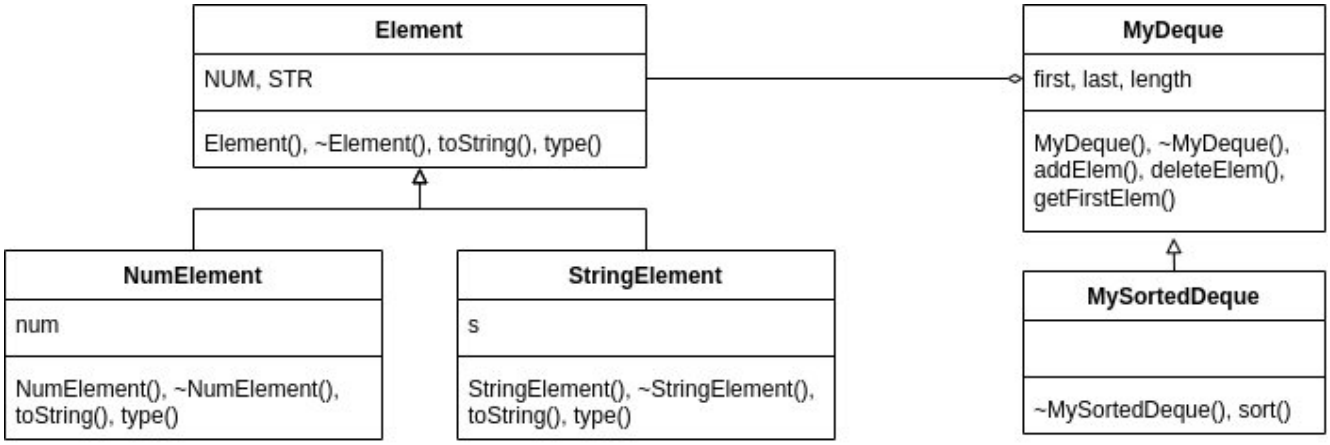

Рисунок 1 — Диаграмма классов

Данная структура данных была выбрана для возможности хранения различных типов данных, а также лёгкой масштабируемости. Получившаяся программа, её исходный код, а также результаты тестирования изображены на рисунках 2-7.

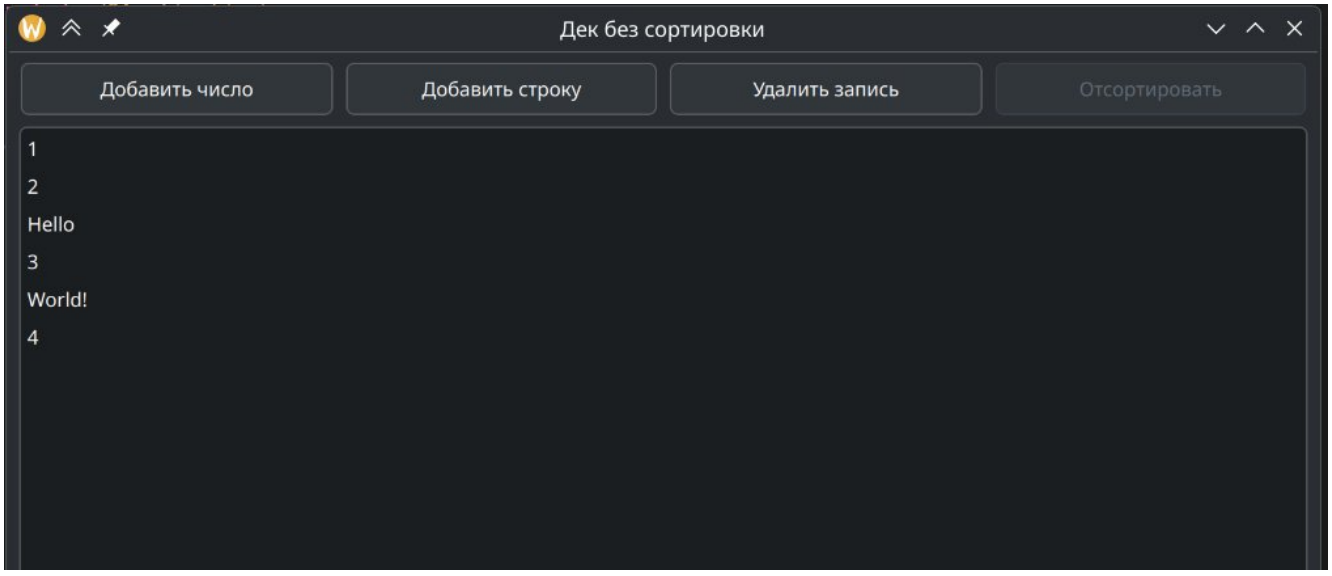

Рисунок 2 — Окно программы

```cpp
#ifndef ELEMENT_H
#define ELEMENT_H

#include <QString>

class Element {
public:
  const int NUM = 1;
  const int STR = 2;
  Element() {};
  virtual ~Element() {};
  virtual QString toString() = 0;
  virtual int type() = 0;
};

class NumElement : public Element {
public:
  NumElement(const int &n) : num(n) {};
  int num;
  QString toString() override;
  int type() override;
};

class StringElement : public Element {
public:
  StringElement(const QString &str) : s(str) {};
  QString s;
  QString toString() override;
  int type() override;
};

#endif // ELEMENT_H
```

Рисунок 3 — Код классов Element, NumElement, StringElement

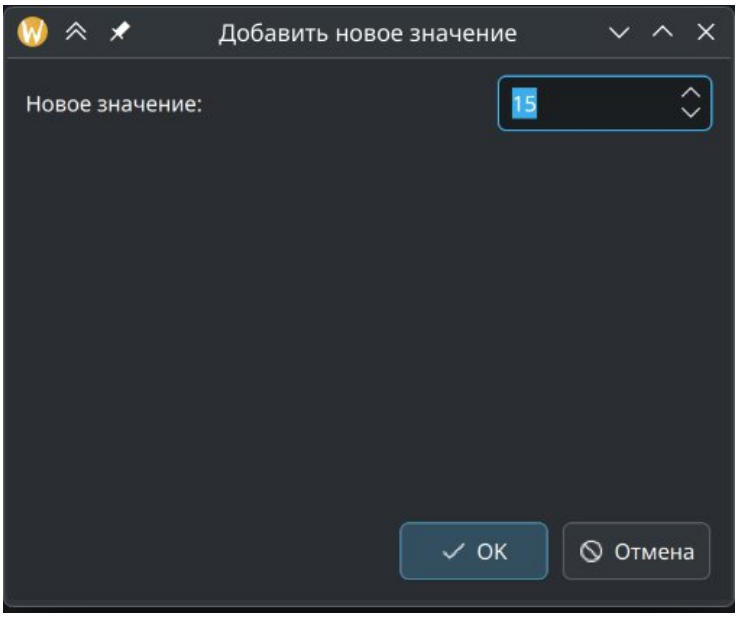

Рисунок 4 — Окно добавления нового значения (число)

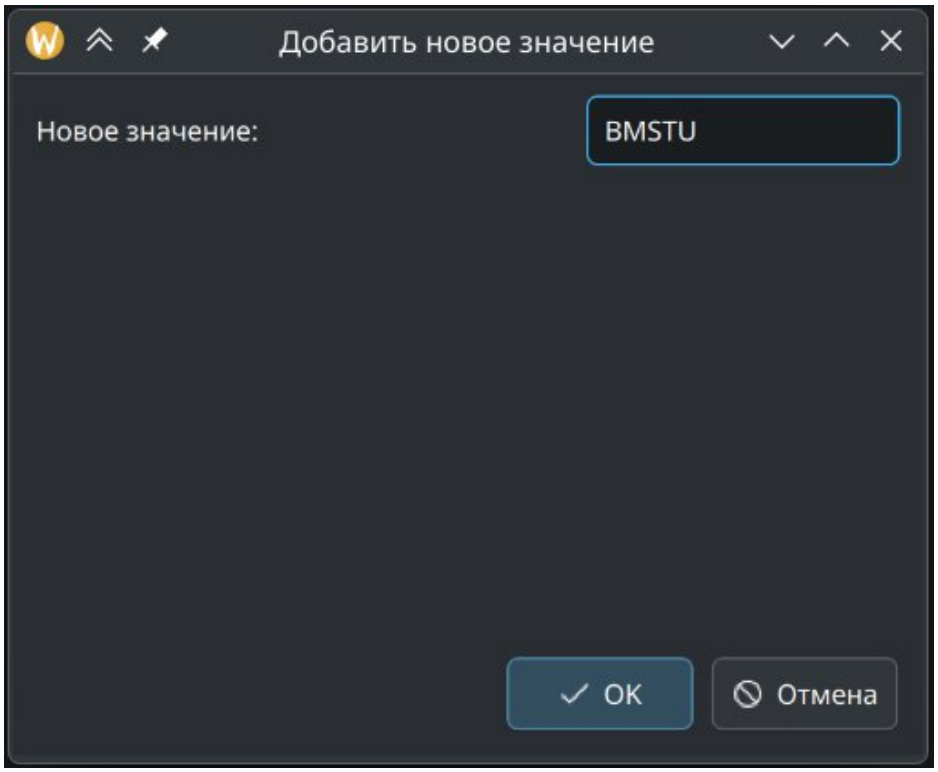

Рисунок 5 — Окно добавления нового значения (строка)

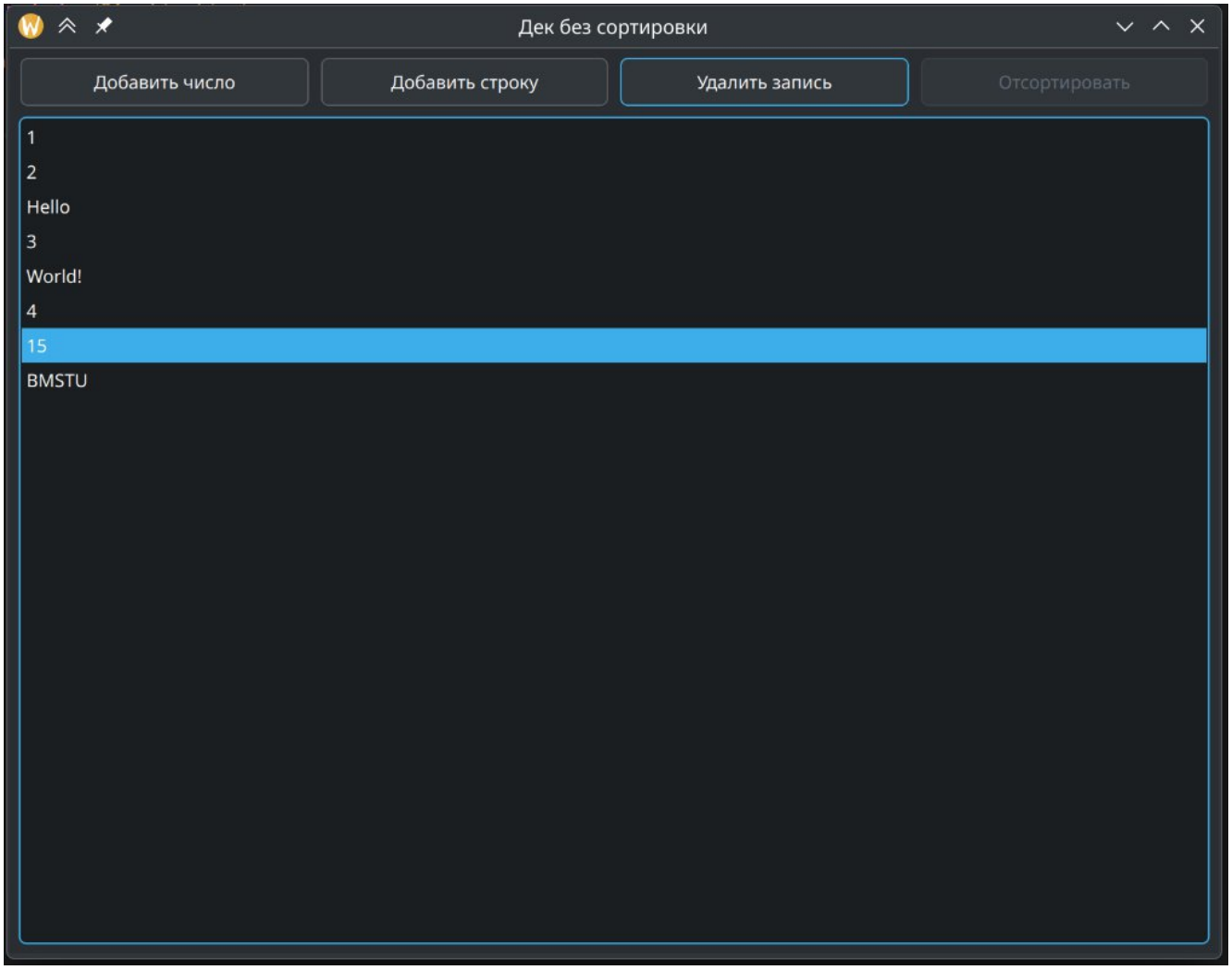

Рисунок 6 — Способ удаления элементов

```
void MyDeque::addElem(Element *new_elem) {
  entry *new_entry = new entry();
  new_entry->element = new_elem;
  try {
    if (first == nullptr) {
      first = new_entry;
      last = new_entry;
    } else {
      new_entry->previous = last;
      last->next = new_entry;
      last = new_entry;
    }
    length++;
  } catch (...) {
  }
}

void MyDeque::deleteItem(const int &index) {
  auto t = first;
  try {
    for (int i = 0; i < index; ++i) {
      t = t->next;
    }
  } catch (...) {
    return;
  }
  if (t == nullptr)
    return;

  if (t == first) {
    first = first->next;
    if (first == nullptr)
      last = nullptr;
    else
      first->previous = nullptr;
  } else if (t == last) {
    last = last->previous;
    last->next = nullptr;
  } else {
    t->previous->next = t->next;
    t->next->previous = t->previous;
  }

  delete t->element;
  delete t;
  length--;
}
```

Рисунок 7 — Реализация добавления и удаления элементов из дека

Дальше был создан класс MySortedDeque – потомок класса MyDeque. Он добавляет возможность сортировки значений. Так как в MyDeque возможны как целочисленные значения, так и строковые, был выбран следующий способ сортировки: сначала числа по возрастанию, затем строки по лексикографическому возрастанию. Код сортировки изображен на рисунке 8. Результат сортировки изображен на рисунке 9.

```
bool MySortedDeque::compare(entry *e1, entry *e2) {
  if (e1->element->type() != e2->element->type()) {
    return e1->element->type() < e2->element->type();
  }
  if (e1->element->type() == e1->element->NUM)
    return e1->element->toString().toInt() < e2->element->toString().toInt();
  return e1->element->toString() <= e2->element->toString();
}

void MySortedDeque::swap(entry *e1, entry *e2) {
  auto temp = e1->element;
  e1->element = e2->element;
  e2->element = temp;
}

MySortedDeque::~MySortedDeque() { MyDeque::~MyDeque(); }

void MySortedDeque::sort() {
  if (first == nullptr)
    return;
  auto it1 = first;
  while (it1 != last) {
    auto it2 = it1;
    auto mn = it1;
    while (it2 != nullptr) {
      if (!compare(mn, it2))
        mn = it2;
      it2 = it2->next;
    }
    if (mn != it1)
      swap(mn, it1);
    it1 = it1->next;     ⚠ Access to field 'next' results in a dereference of
  }
}
```

Рисунок 8 — Код сортировки

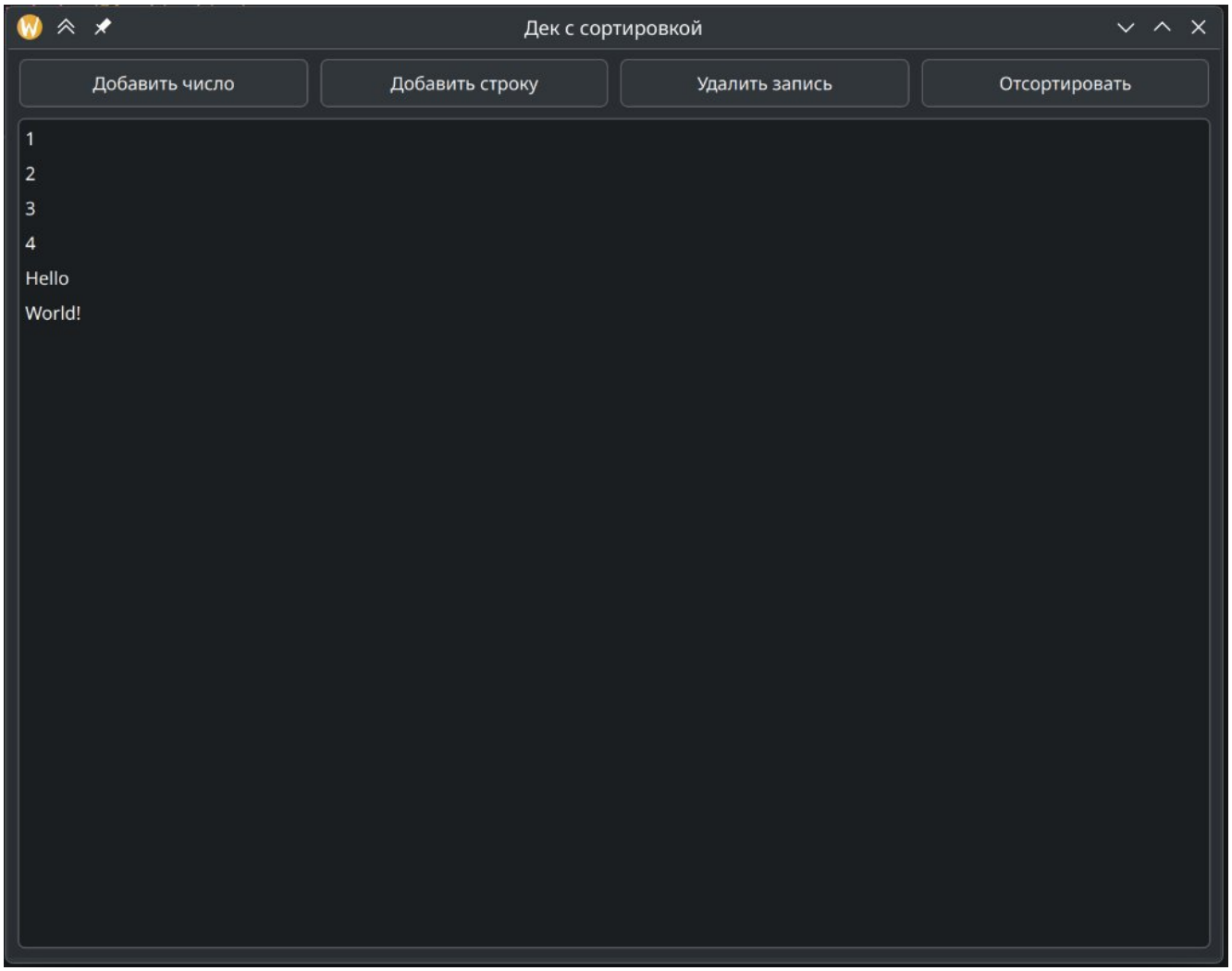

Рисунок 8 — Результат сортировки

Вывод: в ходе работы был разработан контейнер в виде совокупности классов, обеспечивающий хранение и обработку данных. Разработана тестирующая программа с графическим интерфейсом, позволяющая удобно взаимодействовать с функционалом контейнера.

Часть 2. Дек, как шаблон класса

Цель работы: преобразовать разработанный контейнер в шаблон класса.

Ход работы: для изображения различий было решено хранить double в MyUniversalDeque, который не является потомком Element. Исходный код изменения, а также результаты тестирования изображены на рисунках 9-10.

```
template <class T> class MyUniversalDeque {
protected:
  struct entry {
    T *element;
    entry *previous = nullptr;
    entry *next = nullptr;
  };
  entry *first = nullptr;
  entry *last = nullptr;
  int length = 0;

  void swap(entry *e1, entry *e2) {
    auto temp = e1->element;
    e1->element = e2->element;
    e2->element = temp;
  }

public:
  MyUniversalDeque() {}
  ~MyUniversalDeque() {
    for (int i = 0; i < length; i++) {
      last = first;
      last = last->next;
      delete first->element;
      delete first;
      first = last;
    }
  }
```

Рисунок 9 — Часть реализации MyUniversalDeque (основной код идентичен MyDeque)

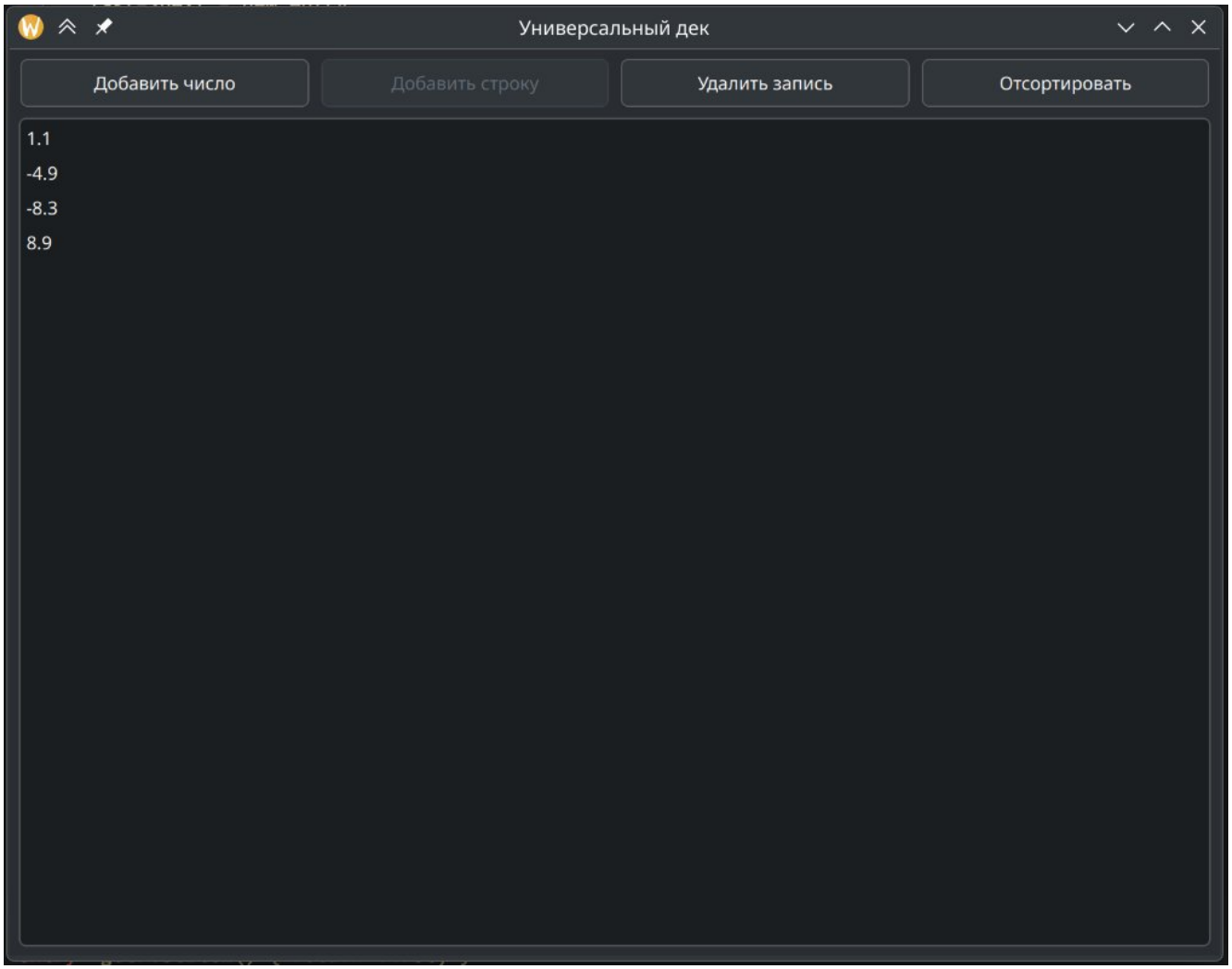

Рисунок 10 — Окно тестирующей программы

Вывод: были разработаны две реализации одной структуры данных – дека с использованием двух разных подходов: совокупность классов и шаблон класса. Также были использованы исключения – способа обработки ошибок.