МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Алгоритмы и структуры данных»
Тема: Бинарные деревья

Студент гр. 9304 _____ Арутюнян В.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с понятием бинарного дерева, реализовать его, решить поставленную задачу на его основе.

Постановка задачи.

Вариант *13y*.

Формулу вида:

<формула> ::= <терминал> | (<формула> <формула> <знак>)

*<знак> ::= + | – | **

<терминал> ::= 0 | 1 | ... | 9 | a | b | ... | z

можно представить в виде бинарного дерева.

– формула из одного терминала представляется деревом из одной вершины с этим терминалом;

– формула вида (f_1 f_2 s) представляется деревом, в котором корень – это знак s, а левое и правое поддеревья – соответствующие представления формул f_1 и f_2.

Требуется:

– построить дерево t из строки, задающей формулу в постфиксной форме (перечисление узлов t в порядке ЛПК);

– упростить дерево-формулу t, выполнив в нем все операции вычитания, в которых уменьшаемое и вычитаемое – цифры. Результат вычитания – цифра или формула вида (*0 – цифра*).

Выполнение работы.

Программа на вход ожидает строку для анализа сразу после флага “*-s*”, иначе ожидается путь до файла со строкой для обработки. Возможно совместное использование:

./lab3 some/path/1 -s “some_string_1” -s “some_string_2” some/path/2

При таком вызове программа обработает строку из файла *some/path/1*, затем строку *some_string_1*, после строку *some_string_2*, далее строку из файла *some/path/2*.

Класс BinaryTree.

Класс представляет собой бинарное дерево. В нём хранятся следующие поля:

1. *head_* – умный указатель на головной элемент (корень) дерева;
2. *is_correct_binary_tree_* – флаг того, что дерево было построено корректно;

Для начала анализа теста, достаточно создать экземпляр класса, передав в конструктор входную строку, и вызвать метод *Simplify()* для упрощения строки. В конструкторе от строки с формулой происходит вызов метода *CreateBinaryTree()* и обработка возможных ошибок при построении строки. Последнее происходит с помощью еще одного реализованного класса *MyException*.

Внутри *CreateBinaryTree()* происходит проверка введенной строки на синтаксическую корректность и построение дерева.

Краткий алгоритм:

Происходит посимвольная проверка, если вдруг введен недопустимый символ или введена операция, пока в дереве присутствует меньше двух формул, то генерируется исключение. Создается *curr* – умный указатель указывающий на ту же область, что и head_.

1. Если введен не <*знак*> и при этом количество введенных формул до этого момента равно нулю, то у дерева создается корень, в который помещается введенный символ, после происходит *curr = head_*;
2. Иначе, если введен не <*знак*>, то создается элемент дерева *temp*, в который помещается так называемая «неизвестная» операция, левым потомком этого элемента становится указатель *curr*, а правым – новый элемент, в который записывается введенный <*терминал*>. При этом,

если у *curr* не было родителя, то *temp* записывается в *head_*, иначе правым потомком родителя *curr* становится *temp*. Родителем *temp* становится бывший родитель *curr*. Также обновляется родитель у потомков *temp* на *temp*, а *curr* начинает указывать на правового потомка *temp*;

3. Введен <*знак*>, то записывается определенная операция в родителя *curr*, после *curr* начинает указывать на своего родителя;
4. В конце введенная строка проверяется на наличие необходимого количества элементов <*знак*> и <*терминал*>, в случае неверного количества, генерируется исключение.

Также в данном классе присутствуют следующие методы:

1. InfixTraverse() – ЛКП обход, построенного дерева, RecursiveInfixTraverse() – вспомогательный метод, в котором представлена реализация обхода;
2. PrefixTraverse() – КЛП обход, построенного дерева, RecursivePrefixTraverse() – вспомогательный метод, в котором представлена реализация обхода;
3. PostfixTraverse() – ЛПК обход, построенного дерева, RecursivePostfixTraverse() – вспомогательный метод, в котором представлена реализация обхода;
4. IsCorrectBinaryTree() – метод возвращающий информацию о флаге is_correct_binary_tree_;
5. Simplify() – упрощение построенного дерева, где выполняются все операции вычитания, в которых уменьшаемое и вычитаемое – цифры, RecursiveSimplify() – вспомогательный метод, в котором представлена реализация упрощения;
6. IsOperation() – метод, позволяющий узнать, является ли символ элементом <знак>;
7. ToOperation() – преобразование символа в тип Operation;

8. ToTryToReplace() – вспомогательный метод для метода Simplify(), пытается произвести сокращение для текущего элемента дерева;
9. ConvertToICO() – конвертирует символ в элемент, который можно поместить в элемент дерева.

Класс Node.

Данный класс является реализацией элемента бинарного дерева.

В нём хранятся следующие поля:

1. *data_* – данные (операция, число или символ);
2. *left_* – указатель на левого потомка элемента дерева;
3. right_ – указатель на правого потомка элемента дерева;
4. parent_ – указатель на родителя элемента дерева;

В классе есть методы, необходимые для проверки синтаксической корректности введенной строки:

1. *IsNumber()* – установление того, является ли элемент, который хранится в *data_* числом;
2. *IsChar()* – установление того, является ли элемент, который хранится в *data_* символом;
3. *IsOperation()* – установление того, является ли элемент, который хранится в *data_* операцией;
4. *GetNumber()* – получение числа, хранящего в *data_*;
5. *GetChar()* – получение символа, хранящего в *data_*;
6. *GetOperation()* – получение операции, хранящейся в *data_*;

enum class Operation.

Хранит возможные виды операций.

Класс MyException.

Данный класс необходим для хранения кода сгенерированной ошибки и дополнительной строки с пояснениями. Здесь же определен метод получения кода ошибки – *GetCode()* и перегружен оператор вывода.

enum class ErrorCode необходим для понятной записи кода ошибки.

Программа выводит упрощенную строку, если входная строка являлась корректной, иначе выводит “ERROR”.

Исходный код находится в приложении А.

Далее будет проиллюстрировано хранение входной строки в бинарном дереве.

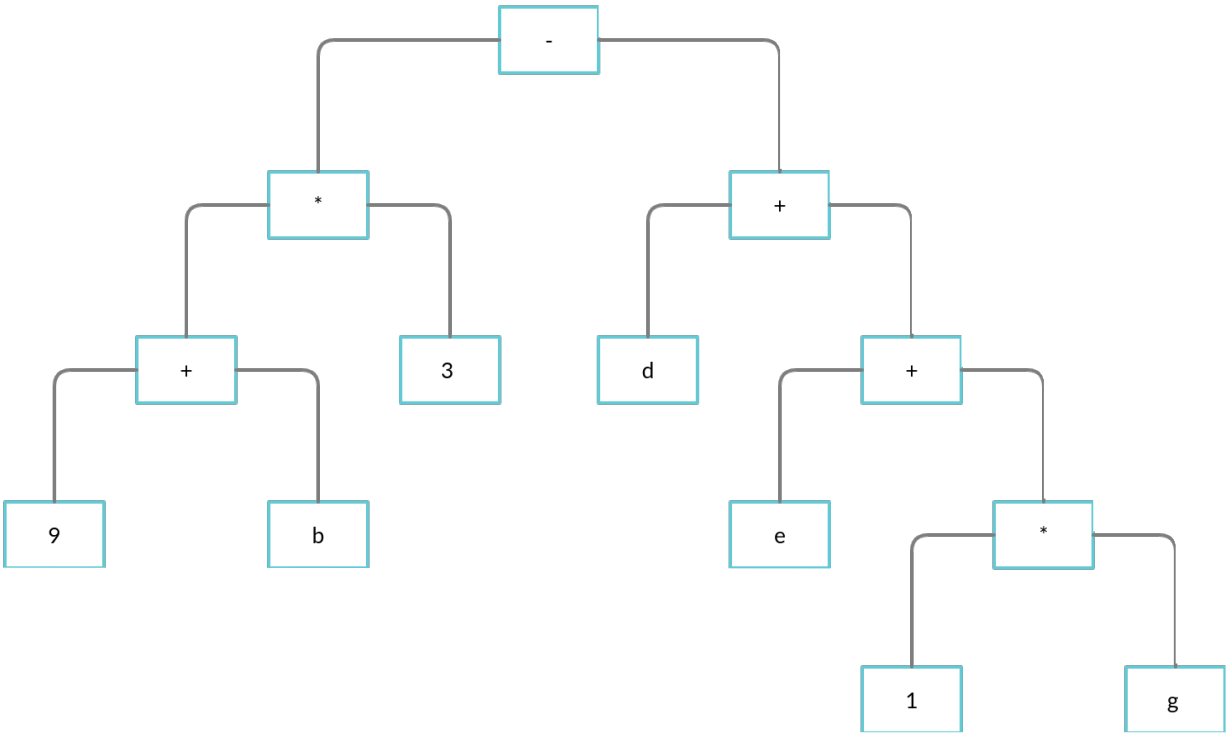

Рисунок 1 – Иллюстрация хранения входного выражения в бинарном дереве

На Рисунке 1 представлено следующее выражение: "9 b + 3 * d e 1 g * + + –".

Тестирование.

Программу можно собрать через *Makefile* командой *make*, после этого создается исполняемый файл *lab3*. Существует несколько вариантов провести тестирование:

1. Вызвать *lab3*, указав путь до файла с тестом, либо передать флаг "-*s*", а затем строку, которую требуется проанализировать, в кавычках;
2. Запустить python-скрипт – *testing.py*, в котором можно изменять параметры для тестирования, например, количество тестов, их расположение, расположение эталонных ответов, расположение ответов, полученных от программы.

Далее будет представлена таблица тестирования с несколькими тестами.

Таблица 1. Примеры входных и выходных данных

№	Входные данные	Выходные данные
1	- * + a b c - d + e * f g	ERROR
2	a b + c * d e f g * + - -	a b + c * d e f g * + - -
3	b 1 k 3 2 - + 3 - * 1 - +	b 1 k 1 + 3 - * 1 - +
4	a 1 + 1 2 - 0 0 - +	ERROR
5	8 3 - 1 0 0 - - -	4
6	t * 1	ERROR
7	t 1 *	t 1 *
8	* t 1	ERROR
9	b 0 7 - * 9 9 - 1 - *	b 0 7 - * 0 1 - *
10	2 2 - 0 + 1 2 3 - + -	0 0 + 1 0 1 - + -

Выводы.

В ходе выполнения лабораторной работы было реализовано бинарное дерево и решена поставленная задача на его основе. Была разработана программа, упрощающая выражение, введенное в ЛПК виде. Использование бинарного дерева, основанного на указателях, оправдывается более удобной обработкой элементов в нескольких видах операций, а также небольшим потреблением памяти относительно реализации бинарного дерева на основе массива.

ПРИЛОЖЕНИЕ А

main.cpp

```
#include <fstream>
#include <iostream>
#include <string>

#include "../lib/binary_tree.h"

void PrintTraverse(std::vector<std::variant<int, char, Operation>>
vec) {
  for (const auto& it : vec) {
    if (std::holds_alternative<int>(it)) {
      std::cout << std::get<int>(it);
    } else if (std::holds_alternative<char>(it)) {
      std::cout << std::get<char>(it);
    } else {
      char c = ' ';
      Operation oper = std::get<Operation>(it);
      if (oper == Operation::kPlus) {
        c = '+';
      } else if (oper == Operation::kMinus) {
        c = '-';
      } else if (oper == Operation::kMultiply) {
        c = '*';
      } else {
        c = '?';
      }
      std::cout << c;
    }
    std::cout << ' ';
  }
  std::cout << '\n';
}

void PrintResult(BinaryTree& tree) {
  if (tree.IsCorrectBinaryTree()) {
    PrintTraverse(tree.PostfixTraverse());
    PrintTraverse(tree.PrefixTraverse());
    PrintTraverse(tree.InfixTraverse());
  } else {
    std::cout << "ERROR\n";
  }
}
```

```
int main(int argc, char** argv) {
  if (argc == 1) {
    std::cout << "Too small arguments.\n"
              << "A algebraic expression is expected after \"-s\"
flag, "
              << "otherwise a file path is expected.\n\n"
              << "example: ./lab3 -s \"- * + a 2 c - 3 + e * f
g \" Tests/test/test1.txt\n";
  } else {
    bool is_string = false;

    for (int i = 1; i < argc; ++i) {
      std::string arg = argv[i];

      if (is_string) {
        is_string = false;
        BinaryTree expr(arg);
        expr.Simplify();
        PrintResult(expr);

      } else if (arg == "-s") {
        is_string = true;

      } else {
        std::ifstream file_in(arg);

        if (file_in.is_open()) {
          std::string str;
          std::getline(file_in, str);
          BinaryTree expr(str);
          expr.Simplify();
          PrintResult(expr);
          file_in.close();

        } else {
          std::cout << "Couldn't open the file.\n";
        } // else
      } // else
    } // for
  } // else
  return 0;
} // main
```

binary_tree.h

```
#ifndef BINARY_TREE_H_
```

```
#define BINARY_TREE_H_

#include <vector>

#include "node.h"

class BinaryTree {
  using vector_traverse = std::vector<std::variant<int, char,
Operation>>;

 public:
  BinaryTree() = default;
  BinaryTree(std::string_view str);
  BinaryTree(const BinaryTree& object);
  BinaryTree(BinaryTree&& object);

  BinaryTree& operator=(const BinaryTree& object);
  BinaryTree& operator=(BinaryTree&& object);

  vector_traverse InfixTraverse();
  vector_traverse PrefixTraverse();
  vector_traverse PostfixTraverse();

  bool IsCorrectBinaryTree();
  void Simplify();

  ~BinaryTree() = default;

 private:
  std::shared_ptr<Node> head_ = nullptr;
  bool is_correct_binary_tree_ = false;

  void CreateBinaryTree(std::string_view str);
  bool IsOperation(const char c);
  Operation ToOperation(const char c);
  void RecursiveSimplify(std::shared_ptr<Node>& object);
  void ToTryToReplace(std::shared_ptr<Node>& object);
  std::variant<int, char, Operation> ConvertToICO(const char c);

  void RecursiveInfixTraverse(vector_traverse& vec,
                              std::shared_ptr<Node>& object);
  void RecursivePrefixTraverse(vector_traverse& vec,
                               std::shared_ptr<Node>& object);
  void RecursivePostfixTraverse(vector_traverse& vec,
                                std::shared_ptr<Node>& object);
  void RecursiveCopy(std::shared_ptr<Node>& curr_object,
```

```
                    const std::shared_ptr<Node>& other_object);
};

#endif  // BINARY_TREE_H_
```

binary_tree.cpp

```
#include "../lib/binary_tree.h"

using vector_traverse = std::vector<std::variant<int, char,
Operation>>;

BinaryTree::BinaryTree(std::string_view str) {
  try {
    is_correct_binary_tree_ = true;
    CreateBinaryTree(str);
  } catch (MyException& err) {
    is_correct_binary_tree_ = false;
  }
}

BinaryTree::BinaryTree(const BinaryTree& object)
    : head_(nullptr),
is_correct_binary_tree_(object.is_correct_binary_tree_) {
  if (object.head_ != nullptr) {
    head_ = std::shared_ptr<Node>(new Node(object.head_->data_));
    RecursiveCopy(head_, object.head_);
  }
}

BinaryTree::BinaryTree(BinaryTree&& object)
    : head_(std::move(object.head_)),

is_correct_binary_tree_(std::move(object.is_correct_binary_tree_))
{}

BinaryTree& BinaryTree::operator=(const BinaryTree& object) {
  if (this == &object) {
    return *this;
  }
  is_correct_binary_tree_ = object.is_correct_binary_tree_;
  head_ = nullptr;
  if (object.head_ != nullptr) {
    head_ = std::shared_ptr<Node>(new Node(object.head_->data_));
    RecursiveCopy(head_, object.head_);
  }
```

```
  return *this;
}

BinaryTree& BinaryTree::operator=(BinaryTree&& object) {
  if (this == &object) {
    return *this;
  }
  is_correct_binary_tree_ =
std::move(object.is_correct_binary_tree_);
  head_ = std::move(object.head_);
  return *this;
}

vector_traverse BinaryTree::InfixTraverse() {
  vector_traverse vec;
  RecursiveInfixTraverse(vec, head_);
  return vec;
}

vector_traverse BinaryTree::PrefixTraverse() {
  vector_traverse vec;
  RecursivePrefixTraverse(vec, head_);
  return vec;
}

vector_traverse BinaryTree::PostfixTraverse() {
  vector_traverse vec;
  RecursivePostfixTraverse(vec, head_);
  return vec;
}

bool BinaryTree::IsCorrectBinaryTree() { return
is_correct_binary_tree_; }

void BinaryTree::Simplify() {
  if (is_correct_binary_tree_) {
    RecursiveSimplify(head_);
  }
}

void BinaryTree::CreateBinaryTree(std::string_view str) {
  int quantity_of_trees = 0;
  bool is_next_word = true;
  int index;
  std::shared_ptr<Node> curr = head_;
  for (index = 0; index < str.size(); ++index) {
```

```
    if (std::isspace(str[index])) {
      is_next_word = true;
      continue;
    } else if (!is_next_word) {
      throw MyException(ErrorCode::kSyntaxError, "invalid
construction");
    } else if (!(std::isdigit(str[index]) ||
std::isalpha(str[index]) ||
                  IsOperation(str[index]))) {
      throw MyException(ErrorCode::kSyntaxError, "invalid
symbol");
    } else if (IsOperation(str[index]) && quantity_of_trees <= 1)
{
      throw MyException(ErrorCode::kSyntaxError,
                        "the operation was received too early");
    }

    bool is_operation = IsOperation(str[index]);

    if (!is_operation && !quantity_of_trees) {
      head_ = std::shared_ptr<Node>(new
Node(ConvertToICO(str[index])));
      curr = head_;
      ++quantity_of_trees;

    } else if (!is_operation) {
      std::variant<int, char, Operation> var =
Operation::kUnknown;
      auto temp = std::shared_ptr<Node>(new Node(var));
      temp->left_ = curr;
      temp->right_ = std::shared_ptr<Node>(new
Node(ConvertToICO(str[index])));

      if (curr->parent_.expired()) {
        head_ = temp;
      } else {
        curr->parent_.lock()->right_ = temp;
      }

      temp->parent_ = curr->parent_;
      temp->left_->parent_ = temp;
      temp->right_->parent_ = temp;
      curr = temp->right_;
      ++quantity_of_trees;

    } else {
```

```
      curr->parent_.lock()->data_ = ToOperation(str[index]);
      curr = curr->parent_.lock();
      --quantity_of_trees;
    }

    is_next_word = false;
  }

  if (quantity_of_trees != 1) {
    throw MyException(ErrorCode::kSyntaxError, "invalid syntax");
  }
}

bool BinaryTree::IsOperation(const char c) {
  return c == '+' || c == '-' || c == '*';
}

Operation BinaryTree::ToOperation(const char c) {
  Operation oper;
  if (c == '+') {
    oper = Operation::kPlus;
  } else if (c == '-') {
    oper = Operation::kMinus;
  } else if (c == '*') {
    oper = Operation::kMultiply;
  } else {
    oper = Operation::kUnknown;
  }
  return oper;
}

void BinaryTree::RecursiveSimplify(std::shared_ptr<Node>& object)
{
  if (object == nullptr) {
    return;
  }
  ToTryToReplace(object);
  RecursiveSimplify(object->left_);
  ToTryToReplace(object);
  RecursiveSimplify(object->right_);
  ToTryToReplace(object);
}

void BinaryTree::ToTryToReplace(std::shared_ptr<Node>& object) {
  if (object->IsOperation() && object->GetOperation() ==
Operation::kMinus &&
```

```
      object->left_->IsNumber() && object->right_->IsNumber()) {
    int diff = object->left_->GetNumber() - object->right_-
>GetNumber();
    if (diff < 0) {
      object->left_->data_ = 0;
      object->right_->data_ = -1 * diff;
    } else {
      object->data_ = diff;
      object->left_ = std::shared_ptr<Node>(nullptr);
      object->right_ = std::shared_ptr<Node>(nullptr);
    }
  }
}

std::variant<int, char, Operation> BinaryTree::ConvertToICO(const
char c) {
  std::variant<int, char, Operation> variant;
  if (std::isdigit(c)) {
    variant = c - '0';
  } else if (IsOperation(c)) {
    variant = ToOperation(c);
  } else {
    variant = c;
  }
  return variant;
}

void BinaryTree::RecursiveInfixTraverse(vector_traverse& vec,
                                        std::shared_ptr<Node>&
object) {
  if (object == nullptr) {
    return;
  }
  RecursiveInfixTraverse(vec, object->left_);
  vec.push_back(object->data_);
  RecursiveInfixTraverse(vec, object->right_);
}

void BinaryTree::RecursivePrefixTraverse(vector_traverse& vec,
                                         std::shared_ptr<Node>&
object) {
  if (object == nullptr) {
    return;
  }
  vec.push_back(object->data_);
  RecursivePrefixTraverse(vec, object->left_);
```

```
  RecursivePrefixTraverse(vec, object->right_);
}

void BinaryTree::RecursivePostfixTraverse(vector_traverse& vec,
                                          std::shared_ptr<Node>&
object) {
  if (object == nullptr) {
    return;
  }
  RecursivePostfixTraverse(vec, object->left_);
  RecursivePostfixTraverse(vec, object->right_);
  vec.push_back(object->data_);
}

void BinaryTree::RecursiveCopy(std::shared_ptr<Node>& curr_object,
                               const std::shared_ptr<Node>&
other_object) {
  if (other_object == nullptr) {
    return;
  }
  curr_object->left_ = nullptr;
  curr_object->right_ = nullptr;

  if (other_object->left_ != nullptr) {
    curr_object->left_ = std::shared_ptr<Node>(
        new Node(other_object->left_->data_, curr_object));
  }
  if (other_object->right_ != nullptr) {
    curr_object->right_ = std::shared_ptr<Node>(
        new Node(other_object->right_->data_, curr_object));
  }

  RecursiveCopy(curr_object->left_, other_object->left_);
  RecursiveCopy(curr_object->right_, other_object->right_);
}
```

node.h

```
#ifndef NODE_H_
#define NODE_H_

#include <memory>
#include <variant>

#include "my_exception.h"
```

```
#include "operation.h"

class Node {
 public:
  Node(std::variant<int, char, Operation> data,
       std::shared_ptr<Node> parent = nullptr,
       std::shared_ptr<Node> left = nullptr,
       std::shared_ptr<Node> right = nullptr);

  bool IsNumber();
  bool IsChar();
  bool IsOperation();
  int GetNumber();
  char GetChar();
  Operation GetOperation();
  ~Node() = default;

 private:
  friend class BinaryTree;
  std::variant<int, char, Operation> data_;
  std::shared_ptr<Node> left_ = nullptr;
  std::shared_ptr<Node> right_ = nullptr;
  std::weak_ptr<Node> parent_;
};

#endif  // NODE_H_
```

node.cpp

```
#include "../lib/node.h"

Node::Node(std::variant<int, char, Operation> data,
           std::shared_ptr<Node> parent, std::shared_ptr<Node>
left,
           std::shared_ptr<Node> right)
    : data_(data), left_(left), right_(right), parent_(parent) {}

bool Node::IsNumber() { return std::holds_alternative<int>(data_);
}

bool Node::IsChar() { return
std::holds_alternative<char>(data_); }

bool Node::IsOperation() { return
std::holds_alternative<Operation>(data_); }
```

```
int Node::GetNumber() {
  if (!IsNumber()) {
    throw MyException(ErrorCode::kValueError, "Trying to get a
different type");
  }
  return std::get<int>(data_);
}

char Node::GetChar() {
  if (!IsChar()) {
    throw MyException(ErrorCode::kValueError, "Trying to get a
different type");
  }
  return std::get<char>(data_);
}

Operation Node::GetOperation() {
  if (!IsOperation()) {
    throw MyException(ErrorCode::kValueError, "Trying to get a
different type");
  }
  return std::get<Operation>(data_);
}
```

my_exception.h

```
#ifndef MY_EXCEPTION_H_
#define MY_EXCEPTION_H_

#include <iostream>
#include <stdexcept>

enum class ErrorCode {
  kNone = 0,
  kIndexError,
  kValueError,
  kSyntaxError,
  kRuntimeError
};

class MyException : public std::runtime_error  {
 public:
  MyException(const ErrorCode code, const std::string&& str = "");
  MyException(const MyException& err);
  ErrorCode GetCode();
```

```
  friend std::ostream& operator<<(std::ostream& out, const
MyException object);
  ~MyException() = default;

 private:
  ErrorCode code_;
};

#endif // MY_EXCEPTION_H_
```

my_exception.cpp

```
#include "../lib/my_exception.h"

MyException::MyException(const ErrorCode code, const std::string&&
str)
    : std::runtime_error(str), code_(code) {}

MyException::MyException(const MyException& err)
    : std::runtime_error(err.what()), code_(err.code_) {}

ErrorCode MyException::GetCode() { return code_; }

std::ostream& operator<<(std::ostream& out, const MyException
object) {
  switch (object.code_) {
    case ErrorCode::kNone:
      out << "None";
      break;
    case ErrorCode::kIndexError:
      out << "IndexError: " << object.what();
      break;
    case ErrorCode::kValueError:
      out << "ValueError: " << object.what();
      break;
    case ErrorCode::kSyntaxError:
      out << "SyntaxError: " << object.what();
      break;
    case ErrorCode::kRuntimeError:
      out << "RuntimeError: " << object.what();
      break;
  }

  return out;
}
```

operation.h

```
#ifndef OPERATION_H_
#define OPERATION_H_

enum class Operation {
  kUnknown = 0,
  kPlus,
  kMinus,
  kMultiply
};

#endif  // OPERATION_H_
```

Makefile

```
CXX = g++
TARGET = lab2
CXXFLAGS = -c -std=c++17
CXXOBJFLAGS = -std=c++17
LIBDIR = source/lib
SRCDIR = source/src
SRCS = $(wildcard $(SRCDIR)/*.cpp)
OBJS = $(SRCS:.cpp=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
     $(CXX) $(CXXOBJFLAGS) $(OBJS) -o $(TARGET)

%.o: $(SRCDIR)/%.cpp $(LIBDIR)/*.h
     $(CXX) $(CXXFLAGS) $<

clean:
     rm -rf $(SRCDIR)/*.o $(TARGET)
```