МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Институт информационных технологий, математики и механики**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

**Отчет по учебной практике**

**«Создание и обработка текстов»**

**Выполнил:** студент группы 381706-1

Денисов Владислав Львович

\_\_\_\_\_\_\_\_\_\_\_ Подпись

**Научный руководитель:**

ассистент каф. МОСТ ИИТММ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лебедев И.Г

Нижний Новгород

2019.

Содержание

[1. Введение 3](#_Toc8297211)

[2. Постановка задачи 4](#_Toc8297212)

[3. Руководство пользователя 5](#_Toc8297213)

[4. Руководство программиста 6](#_Toc8297214)

[4.1 Описание структуры программы 6](#_Toc8297215)

[4.2 Описание структур данных 6](#_Toc8297216)

[4.3 Описание алгоритмов 9](#_Toc8297217)

[5. Заключение 12](#_Toc8297218)

[6. Литература 13](#_Toc8297219)

# Введение

Лабораторная работа направлена на разработку оптимальной системы для хранения текстов.

Необходимость в такой системе вызвана тем, что хранить большие объемы информации с помощью компьютеров – задача не из простых. Так, при использовании подхода, основанном на линейном массиве данных символьного типа, возникнут проблемы при доступе к последним элементам массива. Ведь для выполнения некоторых операций будет необходимо обработать все предыдущие элементы. Например, в случае добавления нового элемента внутрь текста придется сдвигать оставшуюся часть, что может оказаться очень затратным как в плане ресурсов, так и в плане времени.

В связи с этим, для эффективной обработки текстовой информации требуется использовать такую структуру, которая будет обеспечивать оптимальное структурирование и быстрый̆ доступ к различным элементам текста.

Для представления данных в данной лабораторной работе будет использован иерархический связный список. Подобная структура данных применяется при компьютерной реализации деревьев, что и будет использовано для хранения текстов в данной лабораторной работе.

# Постановка задачи

В рамках лабораторной работы ставится задача эффективной реализации системы для создания и обработки текстов.

Для работы будут разработаны:

* Методы создания текста;
* Методы вставки новых элементов в текст,
* Методы поиска элементов в тексте,
* Методы удаления элементов из текста,
* Системный метод вывода текста на консоль.

Программное решение будет выглядеть следующим образом:

1. Класс звеньев текста – TNode.
2. Класс итератор по звеньям текста – TNodeIter.
3. Класс текста – Ttext.
4. Класс для обработки исключений – TException, которые могут возникнуть при выполнении различных операций.
5. Программа, демонстрирующая работу класса TText.
6. Набор автоматических тестов с использованием Google C++ Testing Framework.

# Руководство пользователя

В качестве примера использования разработанной библиотеки предлагается следующее.

После запуска тестовой программы создается структура текста. Пользователю предлагается ввести свой текст

Затем выполняется добавление нового текста в уже существующий, начиная с указанной позиции.

Наконец, пользователю предлагается удалить из текста некоторый кусок, начиная с указанной позиции.

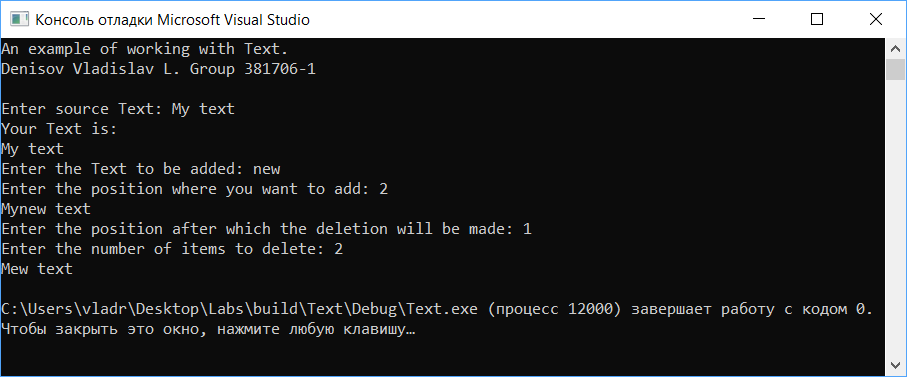


Рисунок 1 Пример работы демонстрационной программы.

# Руководство программиста

## 4.1 Описание структуры программы

Программа состоит из следующих модулей:

* Модуль Text. Содержит пример использования текстов. Реализация в файле *main\_Text.cpp.*
* Модуль TextLib – статическая библиотека. Содержит файлы *Text.h*, *Node.h*, *NodeIter.h* в которых описан интерфейс классов *TText, TNode* и *TNodeIter* соответственно. А также файлы *Text.cpp*, *Node.cpp* и *NodeIter.cpp* содержащие реализацию указанных классов.
* Модуль TextTest. Содержит суммарно 57 тестов (18 для класса текста, 28 для класса звена текста и 11 для итератора по звеньям текста), описанных в файле *TextTest.cpp* и разработанных с помощью использования Google C++ Testing Framework.
* Модуль ExceptionLib – библиотека, позволяющая создавать собственные исключения.

## Описание структур данных

#### Класс TException – класс исключений.

Класс содержит 1 **private** поле *std::string msg* – переменная, хранящая сообщение об ошибке в виде строки.

И содержит 2 **public** элемента:

*TException(std::string \_str) : msg(\_msg)* – конструктор с одним параметром.

*void Print()* – метод отображения ошибки на консоль.

#### Класс TNode – класс звена текста.

Рассмотрим класс *TNode* подробно.

**Элементы класса, объявленные со спецификатором protected:**

TNode\* same\_level – указатель на следующий элемент текста.

TNode\* next\_level – указатель на внутренний элемент в текущем элементе текста.

char letter – буква.

int level – текущий уровень (0 – текст, 1 – строка, 2 – слово, 3 – буква).

static char\* memory – память под элементы дерева.

static TNode\* start – указатель на начало памяти, выделенной под дерево.

static TNode\* end – указатель на конец памяти, выделенной под дерево.

static TNode\* cur\_free – указатель на текущий элемент в общей памяти.

static int node\_size – максимально доступное число элементов.

static int busy\_node\_size – число занятых элементов.

**Элементы класса, объявленные со спецификатором public:**

TNode(const int \_level) – конструктор инициализатор для звена дерева, принимающий уровень, с которым будет создано звено.

TNode(const char\* word) – конструктор инициализатор, принимающий слово.

TNode(const char \_letter = 0) – конструктор инициализатор, принимающий букву.

TNode(const TNode& node) – конструктор копирования.

~TNode() – деструктор.

void SetSameLevel(TNode\* \_same\_level) – метод изменения значения указателя на звено того же уровня.

void SetNextLevel(TNode\* \_next\_level) – метод изменения значения указателя на звено следующего уровня.

void SetLetter(const char \_letter) – метод изменения буквы в звене.

void SetLevel(const int \_level) – метод изменения номера уровня звена.

static void SetNodeSize(const int size) – метод изменения максимального числа элементов.

TNode\* GetSameLevel() – метод получения указателя на звено того же уровня.

TNode\* GetNextLevel() – метод получения указателя на звено следующего уровня.

char GetLetter() – метод получения буквы в звене.

int GetLevel() – метод получения значения уровня звена.

static int GetNodeSize() – метод получения максимального размера звена.

static int GetNodeBusySize() – метод получения занятого числа элементов.

void Initialization(const int size = 100) – метод выделения памяти.

void GarbageCollector() – метод сборки мусора.

static void ClearMemory(void) – метод очистки памяти.

TNode\* Clone() – метод копирования звена.

char\* ToString() – метод перевода звена в строку.

void Output() – метод вывода звена на консоль.

**Перегруженные операторы:**

TNode& operator=(const TNode& node) – оператор присваивания.

TNode& operator+=(const TNode& node) – оператор увеличения с одновременным присваиванием при добавлении другого звена.

TNode& operator+=(const char \_letter) – оператор увеличения с одновременным присваиванием при добавлении буквы.

TNode& operator+=(const char\* word) – оператор увеличения с одновременным присваиванием при добавлении слова.

void\* operator new (const size\_t size) – оператор выделения памяти.

void operator delete (void\* node) – оператор освобождения памяти.

**Дружественная функция:**

friend ostream& operator<<(ostream& os, TNode& text) – перегруженный оператор вывода звена текста на консоль.

#### Класс TNodeIter – класс итератора по звеньям текста.

Рассмтрим класс *TNodeIter* подробно.

**Элементы класса, объявленные со спецификатором protected:**

TNode\* root – указатель на звено, с которого начнется работа итератора (корень дерева, с которым работает итератор)

TNode\* current – указатель на текущую позицию итератора.

TStackList<TNode\*> stack – стек пройденных итератором звеньев (используется обход в глубину).

**Элементы класса, объявленные со спецификатором public:**

TNodeIter(TNode\* \_root) – конструктор инициализатор, принимающий указатель на обходимое дерево.

TNode\* GoNext() – переход к следующему звену по итератору.

void Reset() – вернуться к корню.

void PutInStack(TNode\* node) – добавление звена в стек.

bool IsEnd() – условный оператор проверки на окончание просмотра дерева.

**Перегруженные операторы:**

TNode\* operator()() – возврат указателя на текущее просматриваемое звено.

TNodeIter& operator++() – инкремент.

#### Класс TText – класс текста.

Рассмотрим класс *TText* подробно.

**Элементы класса, объявленные со спецификатором protected:**

TNode\* root – указатель на корень дерева.

**Элементы класса, объявленные со спецификатором public:**

TText(const char\* string) – конструктор преобразования строки в текст.

TText(TText& text) – конструктор копирования текста.

TText(TNode& node) – конструктор преобразования типа.

TNode\* GetRoot() – получение указателя на корень дерева.

void Insert(const int pos, const char\* string) – вставка строки по указанному номеру позиции в тексте.

void Insert(TNode\* start, TNode\* string) – вставка строки после указанного элемента в тексте.

int Find(const char\* string) – поиск позиции строки в тексте.

TNode\* FindNode(const char\* string) – поиск элемента в тексте по указанной строке.

char\* Copy(const int start, const int len) – копирование после указанной позиции start символов в количестве len.

TNode\* Copy(TNode\* start, const int len) – копирование после указанного элемента текста start символов в количестве len.

void Delete(const int start\_del, const int lenght) – удаление с указанной позиции start\_del символов в количестве lenght.

void Delete(TNode\* start\_del, const int lenght) – удаление с указанного элемента текста start\_del символов в количестве lenght.

## Описание алгоритмов

Рассмотрим некоторые алгоритмы, работа которых не очевидна на первый взгляд.

**Работа конструктора класса TText.**

Создаем корень дерева и первую строку в тексте. Создаем указатель tmp на первую строку дерева. Создаем указатель word на массив символьного типа. Вычисляем длину len создаваемого текста. Создаем массив space\_pos, хранящий позиции пробелов в тексте, а также заводим целочисленную переменную space\_count = 0, хранящую число пробелов в тексте. Для удобства создаем iter = 0 – итератор по отдельно взятому слову текста.

Проходим по вставляемому тексту до конца и запоминаем позиции пробелов и их число.

Выделяем память под первое слово. Обнуляем число пробелов в тексте. Выполняем повторный просмотр полученного текста.

Заполняем слово, пока не будет достигнут пробел. Иначе вставляем в добавляемое слово ‘0’, который будет являться разделителем

Если добавляли первое слово, то в строке tmp создаем слово, с помощью word. Указателю tmp спускаемся с уровня строки на уровень слова, т.е. присваиваем значение указателя на слово в тексте.

Иначе создавалось очередное слово в той же строке, связываем их как соседей, продвигаемся по тексту.

Повторяем действия до тех пор, пока не закончим обработку полученного на создание текста.

**Вставка нового текста в позицию с указанным номером.**

Вычисляем длину вставляемого текста.

Создаем первую букву вставляемого текста и сохраняем указатель на нее в str – он будет использовать в качестве итератора по добавляемому слову. Сохраняем также в start\_str указатель на только что созданную букву – начало добавляемого слова.

Затем в цикле выполняем выделение памяти под последующие буквы вставляемого слова.

Создаем итератор для исходного текста. В цикле, пока не достигнем конца текста, ищем подходящую позицию. Число уже пройденных букв при поиске позиции сохраняем в len. Внутри цикл устроен следующим образом.

С помощью итератора переходим к следующему элементу и сохраняем указатель на него в tmp. Рассматриваются различные случаи.

1. Если уровень tmp на текущей итерации цикла равен 2 и полученная позиция pos равна 0, то выполняется вставка в начало. Значит, помещаем tmp в стек, чтобы при выполнении перехода к следующему мы оказались на бывшем начале текста. Выходим из цикла.
2. Если уровень tmp на текущей итерации цикла равен 3, то мы находимся на уроне буквы. Увеличиваем число пройденных букв на 1. Продолжаем выполнение цикла.
3. Если len = pos-1, т.е. мы находимся на предыдущей от вставляемой позиции, то выходим из цикла.

С помощью итератора переходим к вставляемой позиции point и выполняем проверку на вставляемую позицию pos.

Если вставка в начало, т.е.pos=0, то после последней буквы вставляемой строки устанавливаем первую букву бывшей первой строки, изменяем начало дерева.

В противном случае после последней буквы вставляемой буквы устанавливаем букву с места вставки, а в позицию, в которую производилась вставка, устанавливаем вставляемое слово.

**Поиск позиции начала искомого текста.**

Создаем итератор по исходному тексту. Вычисляем длину len искомого слова string.

В цикле с помощью итератора проходим по тексту, пока не будет достигнут его конец. Внутри цикл устроен следующим образом.

С помощью итератора переходим к следующему элементу и сохраняем указатель на него в tmp. Рассматриваем несколько случаев.

Блок (\*). Если наткнулись на первую букву искомого слова, то сравниваем последующие. В случае если очередная буква текста не совпала с буквой слова, проверяем условие.

Блок (\*\*) внутри блока (\*). Если хоть немного успели пройти по слову, то помещаем указатель на предыдущее начало поиска в стек, чтобы потом продолжить со следующего после него. Искомую позицию сдвигаем на число уже пройденных букв. В противном случае, если по слову еще не проходили, то идем до первой подходящей позиции для начала поиска.

После выхода из блока (\*) выполняем проверку на то, пройдена ли целиком искомая строка или нет. Если да, то выходим из цикла.

После выхода из цикла – возвращаем позицию pos искомой строки.

.

# Заключение

В результате лабораторной работы была разработана библиотека, реализующая классы для работы со структурой данной – текст. Она позволяет создать объект класса TText и выполнить с ним те операции, задача реализации которых была поставлена в начале данной лабораторной работы.

Были разработаны и доведены до успешного выполнения тесты, разработанные для данного программного проекта с использованием Google C++ Testing Framework.

Программное решение было продемонстрировано с помощью простейшего набора операций над полиномами. Описание примера работы с полнимомами было представлено в разделе «Руководство пользователя».

# Литература

1. Лабораторный практикум. Составители:Барышева И.В., Мееров И.Б., Сысоев А.В.,

Шестакова Н.В. Под редакцией Гергеля В.П. Учебно-методическое пособие. – Нижний

Новгород: Нижегородский госуниверситет, 2017. – 105с.