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**

**ЛАБОРАТОРНАЯ РАБОТА №15**

**«Таблица на АВЛ (сбалансированном) дереве»**

**Выполнил:** студент группы 381706-2

Жбанова Надежда Сергеевна

\_\_\_\_\_\_\_\_\_\_\_ Подпись

**Руководитель:**

Ассистент кафедры МОСТ

Лебедев Илья Геннадьевич

\_\_\_\_\_\_\_\_\_\_\_ Подпись

Нижний Новгород

2019

Содержание

[1.Введение 3](#_Toc9681356)

[2. Цели и задачи 4](#_Toc9681357)

[2.1. Используемые инструменты 4](#_Toc9681358)

[3. Руководство пользователя 6](#_Toc9681359)

[4. Руководство программиста 9](#_Toc9681360)

[4.1. Описание структуры программы 9](#_Toc9681361)

[4.2. Описание функций и процедур, их алгоритмов 9](#_Toc9681362)

[6. Заключение 13](#_Toc9681363)

[7. Литература 14](#_Toc9681364)

[8. Приложения 15](#_Toc9681365)

[8.1. Приложение 1:Класс TExсeption 15](#_Toc9681366)

[8.2. Приложение 2:Класс TString 15](#_Toc9681367)

[8.3. Приложение 3:Класс TElemTree 17](#_Toc9681368)

[8.4. Приложение 4:Класс TTableTree 19](#_Toc9681369)

[8.5. Приложение 5:Код программы тестирования 23](#_Toc9681370)

[8.6. Приложение 6:Тесты для классов 24](#_Toc9681371)

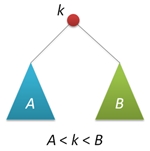
# 1.Введение

Понятие таблицы:

**Таблица**— набор элементов одинаковой организации, каждый из которых можно представить в виде двойки <K, V>, где K— ключ, а V— тело (информационная часть) элемента.

Ключ уникален для каждого элемента, то есть в таблице нет элементов с одинаковыми ключами. Ключ используется для доступа к элементам при выполнении операций.

Понятие АВЛ-дерева

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу*.* Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от log2(n + 1) до 1.44 log2(n + 2) − 0.328. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем гарантированную логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается не равной нулю.

# 2. Цели и задачи

В рамках лабораторной работы ставится задача эффективной реализации структуры данных – таблица на сбалансированном дереве и выполнение основных операций над ней:

* инициализация
* добавление/удаление элемента
* поиск элемента

Для реализации этой структуры данных будем использовать АВЛ-дерево.

В процессе выполнения лабораторной работы требуется использовать систему контроля версий [Git](https://git-scm.com/book/ru/v2) и фрэймворк для разработки автоматических тестов [Google Test](https://github.com/google/googletest).

Выполнение работы предполагает решение следующих задач:

1. Реализация методов шаблонного класса элемента таблицы TElemTree согласно заданному интерфейсу, с использованием класса строки TString.
2. Реализация методов шаблонного класса таблицы на сбалансированном дереве TTableTree согласно заданному интерфейсу, с использованием ранее реализованного класса TStackList.
3. Реализация класса для обработки исключений– TException, которые могут возникнуть при выполнении различных операций, согласно заданному интерфейсу.
4. Обеспечение работоспособности тестов и примера использования.
5. Реализация заготовок тестов, покрывающих все методы созданных классов.
6. Модификация примера использования в тестовое приложение, позволяющее осуществлять основные операции над таблицей.

## 2.1. Используемые инструменты

* Система контроля версий [Git](https://git-scm.com/book/ru/v2). Рекомендуется использовать один из следующих клиентов на выбор студента:
  + [Git](https://git-scm.com/downloads)
  + [GitHub Desktop](https://desktop.github.com/)
* Фреймворк для написания автоматических тестов [Google Test](https://github.com/google/googletest).
* Среда разработки Microsoft Visual Studio (2008 или старше).
* Опционально. Утилита [CMake](http://www.cmake.org/) для генерации проектов по сборке исходных кодов. Может быть использована для генерации решения для среды разработки, отличной от Microsoft Visual Studio 2008 или 2010.

# 3. Руководство пользователя

Запускаем программу из файла sample\_ttabletree.cpp.

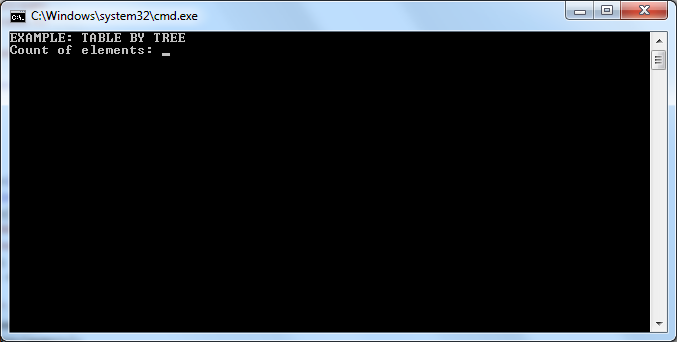


Рис.1. Ввод кол-ва элементов таблицы.

Пользователю предлагается ввести кол-во элементов создаваемой таблицы (Рис.1) и последовательно ее заполнить. (Рис.2):



Рис.2.Заполнение.

После заполнения таблицы элементами, предлагается добавить еще один элемент. (Рис.3):

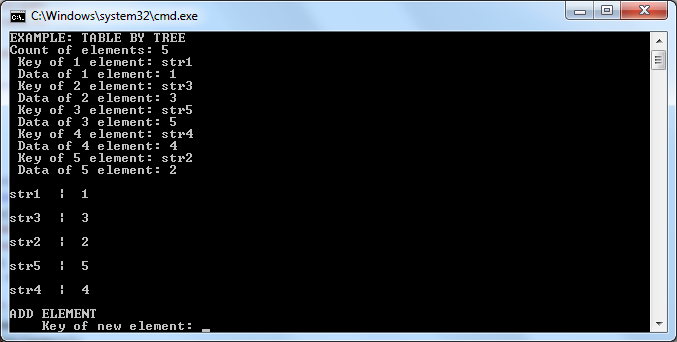


Рис.3.Добавление элемента.

Далее предлагается удалить элемент таблицы по его ключу. (Рис.4,5):

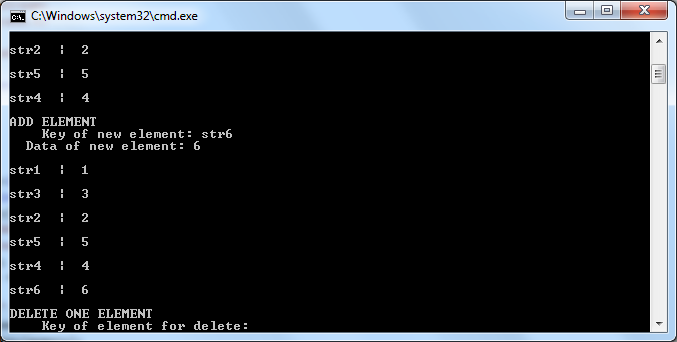


Рис.4.Удаление элемента из таблицы по ключу.

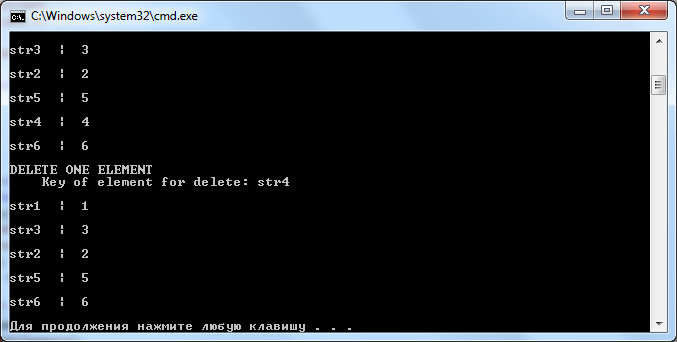


Рис.5.Итоговый результат.

Алгоритм можно повторить несколько раз.

# 4. Руководство программиста

Разработка системы вычисления проводились в среде “Microsoft Visual Studio 2010”.

В данной работе будет использовано несколько классов:

* Класс «Строка» (TString).
* Класс «Элемент таблицы» (TElemTree).
* Класс «Таблица на сбалансированном дереве» (TTableTree).
* Класс «Стек на списке» (TStackList).
* Класс исключения (TExсeption).

## 4.1. Описание структуры программы

Модульная структура программы:

1. tstacklist.h, tlist.h, telement.h - модуль с классом TStackList, в котором определен интерфейс шаблонного класса «Стек на списке» и реализация его методов.
2. ttabletree.h– модуль с классом TTableTree, в котором определен интерфейс шаблонного класса «Таблица на сбалансированном дереве» и реализация его методов.
3. telemtree.h– модуль с классом TElemTree, в котором определен интерфейс шаблонного класса «Элемент таблицы» и реализация его методов.
4. tstring.h– модуль с классом TString, в котором определен интерфейс класса «Строка» и реализация его методов.
5. exсeption.h – модуль с классом исключения TExсeption.
6. sample\_ttabletree.cpp – модуль программы тестирования, с которым работает пользователь.
7. test\_main.cpp, test\_ttabletree.cpp – модуль с функциями тестирования для созданных классов. Содержат 10 тестов для класса TString, 15 - для класса TElemTree и 8- для класса TTableTree.

## 4.2. Описание функций и процедур, их алгоритмов

Рассмотрим реализацию методов класса TString:

class TString

char\* memory – указатель на массив символов строки.

int count – кол-во символов в строке.

1. TString() - конструктор по умолчанию.

2. TString(const char\* A) - конструктор с параметрами. Принимает указатель на массив символов.

3. TString(const TString& A) - конструктор копирования.

Принимает ссылку на объект класса TString.

4. ~TString() - деструктор.

5. int GetCount() const – метод, возвращающий кол-во символов строки.

6. char\* GetMemory() – метод, возвращающий указатель на массив символов строки.

7. TString& operator=(const TString& A) - перегрузка оператора присваивания.

8. bool operator==(const TString& A) const - перегрузка оператора сравнения“==”.

9. bool operator!=(const TString& A) const - перегрузка оператора сравнения“!=”.

10. bool operator<(const TString& A) const - перегрузка оператора “<”.

11. bool operator>(const TString& A) const - перегрузка оператора “>”.

12. char& operator[](const int a) const - перегрузка оператора индексации.

Принимает номер элемента массива символов.

Рассмотрим реализацию методов шаблонного класса TElemTree:

template <typename ElemTreeType>

class TElemTree

TString key - ключ элемента таблицы.

ElemTreeType data - значение элемента таблицы.

TElemTree<ElemTreeType>\* left - указатель на левого потомка.

TElemTree<ElemTreeType>\* right - указатель на правого потомка.

TElemTree<ElemTreeType>\* parent - указатель на родителя(предка).

1. TElemTree() - конструктор по умолчанию.

2. TElemTree(const ElemTreeType& \_data, const TString& A, TElemTree<ElemTreeType>\* \_left = NULL, TElemTree<ElemTreeType>\* \_right = NULL, TElemTree<ElemTreeType>\* \_parent = NULL) - конструктор с параметрами.

Принимает пять параметров. Значение элемента таблицы и его ключ указываются обязательно. Параметры указателей выставятся в null по умолчанию.

3. TElemTree(const TElemTree<ElemTreeType>& A) - конструктор копирования.

Принимает ссылку на объект класса TElemTree.

4. ~TElemTree() – деструктор класса.

5. TElemTree<ElemTreeType>& operator=(const TElemTree<ElemTreeType>& A) - перегрузка оператора присваивания.

6. bool operator==(const TElemTree<ElemTreeType>& A) const - перегрузка оператора сравнения“==”.

Возвращает true, если элементы таблицы равны, и false – если не равны.

7. bool operator!=(const TElemTree<ElemTreeType>& A) const - перегрузка оператора сравнения“!=”.

Возвращает true, если элементы таблицы не равны, и false – если равны.

8. ElemTreeType& GetData() - метод, возвращающий значение элемента таблицы.

9. TString& GetKey() - метод, возвращающий ключ элемента таблицы.

10. TElemTree<ElemTreeType>\* GetLeft() - метод, возвращающий указатель на левого потомка.

11. TElemTree<ElemTreeType>\* GetRight() – метод, возвращающий указатель на правого потомка.

12. TElemTree<ElemTreeType>\* GetParent() - метод, возвращающий указатель на родителя.

13. void SetData(const ElemTreeType& A) - метод, устанавливающий значение элемента таблицы.

14. void SetKey(const TString& A) - метод, устанавливающий ключ элемента таблицы.

15. void SetLeft(TElemTree<ElemTreeType>\* A) - метод, устанавливающий указатель левого потомка.

16. void SetRight(TElemTree<ElemTreeType>\* A) - метод, устанавливающий указатель правого потомка.

17. void SetParent(TElemTree<ElemTreeType>\* A) - метод, устанавливающий указатель родителя.

18. template <typename Type>

friend ostream& operator<<(ostream& ostr, TElemTree<Type>& A) - дружественный метод, перегрузка оператора потокового вывода элемента таблицы.

Рассмотрим реализацию методов шаблонного класса TTableTree:

template <typename TableTreeType>

class TTableTree

static TElemTree<TableTreeType> ttt - первоначальный пустой элемент таблицы.

int count - кол-во элементов таблицы

TElemTree<TableTreeType>\* node - указатель на массив элементов таблицы.

1. TTableTree() - конструктор по умолчанию.

2. TTableTree(const TTableTree<TableTreeType>& A) - конструктор копирования.

Принимает ссылку на объект класса TTableTree.

3. ~TTableTree() - деструктор.

4. int GetCount() const - метод, возвращающий кол-во элементов таблицы.

5. TElemTree<TableTreeType>\* GetNode() const - метод, возвращающий указатель на массив элементов таблицы.

6. TElemTree<TableTreeType>\* Add(TElemTree<TableTreeType>& A) - метод, добавляющий элемент в таблицу.

7. bool Delete(const TString& A) - метод, удаляющий элемент таблицы.

В качестве аргумента принимает его ключ.

8. TElemTree<TableTreeType>& Search(const TString& A) const - метод, осуществляющий поиск элемента в таблице по его ключу.

9. TableTreeType& operator[](const TString& A) const - перегрузка оператора индексации.

Возвращает элемент таблицы по его ключу.

10. template <typename Type>

friend ostream& operator<<(ostream& ostr, const TTableTree<Type>& A) - дружественный метод, перегрузка оператора потокового вывода таблицы.

# 

# 6. Заключение

В результате лабораторной работы была разработана структура данных – Таблица на сбалансированном дереве, а также освоены такие инструменты разработки программного обеспечения, как система контроля версий [Git](https://git-scm.com/book/ru/v2) и фрэймворк для разработки автоматических тестов [Google Test](https://github.com/google/googletest).

Созданные классы был протестированы с использованием Google Tests.

# 7. Литература

1. Лабораторный практикум: Учебно-методическое пособие / Мееров И.Б. [и др.] – Нижний Новгород: Нижегородский госуниверситет , 2017. – 105с.
2. Тестирование с использованием Google Test

(http://neerc.ifmo.ru/wiki/index.php?title=Тестирование\_с\_использованием\_Google\_Test#.D0.A4.D1.83.D0.BD.D0.BA.D1.86.D0.B8.D1.8F\_main.28.29)

1. Касперски К. Техника оптимизации программ. Эффективное использование памяти. — СПб.: БХВ-Петербург, 2003. — 464 с.: ил.
2. <https://habr.com/ru/post/150732/>

# 8. Приложения

## 8.1. Приложение 1:Класс TExсeption

|  |
| --- |
| **exception.h** |
| #pragma once  #include <iostream>  #include <string>  class TException  {  private:  std::string str;  public:  TException(std::string \_str);  void Show();  };  TException::TException(std::string \_str) : str(\_str) {}  void TException::Show()  {  std::cout << "\nWarning! \nMessage: " << str << std::endl;  } |

## 8.2. Приложение 2:Класс TString

|  |
| --- |
| **tstring.h** |
| #pragma once  #include <iostream>  #include <cstring>  #include "exception.h"  class TString  {  protected:  char\* memory;  int count;  public:  TString();  TString(const char\* A);  TString(const TString& A);  ~TString();  int GetCount() const { return count; }  char\* GetMemory() { return memory; }  TString& operator=(const TString& A);  bool operator==(const TString& A) const;  bool operator!=(const TString& A) const;  bool operator<(const TString& A) const;  bool operator>(const TString& A) const;  char& operator[](const int a) const;  };  //----------------------------------------------------------------------  TString::TString() : count(0), memory(NULL)  {  }  //----------------------------------------------------------------------  TString::TString(const char\* A)  {  count = strlen(A);  memory = new char[count + 1];  for (int i = 0; i < count; i++)  memory[i] = A[i];  memory[count] = '\0';  }  //----------------------------------------------------------------------  TString::TString(const TString& A)  {  count = A.count;  memory = new char[count + 1];  for (int i = 0; i < count; i++)  memory[i] = A.memory[i];  memory[count] = '\0';  }  //----------------------------------------------------------------------  TString::~TString()  {  count = 0;  delete[] memory;  }  //----------------------------------------------------------------------  TString& TString::operator=(const TString& A)  {  count = A.count;  delete[] memory;  memory = new char[count + 1];  for (int i = 0; i < count; i++)  memory[i] = A.memory[i];  memory[count] = '\0';  return \*this;  }  //----------------------------------------------------------------------  bool TString::operator==(const TString& A) const  {  if (count != A.count)  return false;  for (int i = 0; i < count; i++)  if (memory[i] != A.memory[i])  return false;  return true;  }  //----------------------------------------------------------------------  bool TString::operator!=(const TString& A) const  {  if (count != A.count)  return true;  for (int i = 0; i < count; i++)  if (memory[i] != A.memory[i])  return true;  return false;  }  //----------------------------------------------------------------------  bool TString::operator<(const TString& A) const  {  bool res = false;  if (count < A.count)  return true;  else if (count > A.count)  return false;  else  {  for (int i = 0; i < count; i++)  if (memory[i] < A.memory[i])  {  res = true;  break;  }  }  return res;  }  //----------------------------------------------------------------------  bool TString::operator>(const TString& A) const  {  bool res = false;  if (count < A.count)  return false;  else if (count > A.count)  return true;  else  for (int i = 0; i < count; i++)  if (memory[i] > A.memory[i])  {  res = true;  break;  }  return res;  }  //----------------------------------------------------------------------  char& TString::operator[](const int a) const  {  if (a < 0 || a > count - 1)  throw TException("Error");  return memory[a];  } |

## 8.3. Приложение 3:Класс TElemTree

|  |
| --- |
| **telemtree.h** |
| #pragma once  #include "tstring.h"  #include <iostream>  using namespace std;  template <typename ElemTreeType>  class TElemTree  {  protected:  TString key;  ElemTreeType data;  TElemTree<ElemTreeType>\* left;  TElemTree<ElemTreeType>\* right;  TElemTree<ElemTreeType>\* parent;  public:  TElemTree();  TElemTree(const ElemTreeType& \_data, const TString& A, TElemTree<ElemTreeType>\* \_left = NULL,  TElemTree<ElemTreeType>\* \_right = NULL, TElemTree<ElemTreeType>\* \_parent = NULL);  TElemTree(const TElemTree<ElemTreeType>& A);  ~TElemTree();  TElemTree<ElemTreeType>& operator=(const TElemTree<ElemTreeType>& A);  bool operator==(const TElemTree<ElemTreeType>& A) const;  bool operator!=(const TElemTree<ElemTreeType>& A) const;  ElemTreeType& GetData() { return data; }  TString& GetKey() { return key; }  TElemTree<ElemTreeType>\* GetLeft() { return left; }  TElemTree<ElemTreeType>\* GetRight() { return right; }  TElemTree<ElemTreeType>\* GetParent() { return parent; }  void SetData(const ElemTreeType& A);  void SetKey(const TString& A);  void SetLeft(TElemTree<ElemTreeType>\* A);  void SetRight(TElemTree<ElemTreeType>\* A);  void SetParent(TElemTree<ElemTreeType>\* A);  template <typename Type>  friend ostream& operator<<(ostream& ostr, TElemTree<Type>& A);  };  //----------------------------------------------------------------------  template <typename ElemTreeType>  TElemTree<ElemTreeType>::TElemTree() : left(NULL), right(NULL), parent(NULL)  {  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  TElemTree<ElemTreeType>::TElemTree(const ElemTreeType& \_data, const TString& A, TElemTree<ElemTreeType>\* \_left,  TElemTree<ElemTreeType>\* \_right, TElemTree<ElemTreeType>\* \_parent) : data(\_data), key(A), left(\_left), right(\_right), parent(\_parent)  {  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  TElemTree<ElemTreeType>::TElemTree(const TElemTree<ElemTreeType>& A) : data(A.data), key(A.key), left(NULL), right(NULL), parent(NULL)  {  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  TElemTree<ElemTreeType>::~TElemTree()  {  left = NULL;  right = NULL;  parent = NULL;  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  TElemTree<ElemTreeType>& TElemTree<ElemTreeType>::operator=(const TElemTree<ElemTreeType>& A)  {  if (this != &A)  {  left = A.left;  right = A.right;  parent = A.parent;  key = A.key;  data = A.data;  }  return \*this;  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  bool TElemTree<ElemTreeType>::operator==(const TElemTree<ElemTreeType>& A) const  {  return (key == A.key && data == A.data) ? true : false ;  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  bool TElemTree<ElemTreeType>::operator!=(const TElemTree<ElemTreeType>& A) const  {  return !(\*this == A);  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  void TElemTree<ElemTreeType>::SetData(const ElemTreeType& A)  {  data = A;  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  void TElemTree<ElemTreeType>::SetKey(const TString& A)  {  key = A;  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  void TElemTree<ElemTreeType>::SetLeft(TElemTree<ElemTreeType>\* A)  {  left = A;  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  void TElemTree<ElemTreeType>::SetRight(TElemTree<ElemTreeType>\* A)  {  right = A;  }  //----------------------------------------------------------------------  template <typename ElemTreeType>  void TElemTree<ElemTreeType>::SetParent(TElemTree<ElemTreeType>\* A)  {  parent = A;  }  //----------------------------------------------------------------------  template <typename Type>  ostream& operator<<(ostream& ostr, TElemTree<Type>& A)  {  ostr << A.GetKey().GetMemory() << " " << "| ";  ostr << A.GetData() << endl;  return ostr;  } |

## 8.4. Приложение 4:Класс TTableTree

|  |
| --- |
| **ttabletree.h** |
| #pragma once  #include "telemtree.h"  #include "tstacklist.h"  using namespace std;  template <typename TableTreeType>  class TTableTree  {  protected:  static TElemTree<TableTreeType> ttt;  int count;  TElemTree<TableTreeType>\* node;  public:  TTableTree();  TTableTree(const TTableTree<TableTreeType>& A);  ~TTableTree();  int GetCount() const { return count; }  TElemTree<TableTreeType>\* GetNode() const { return node; }  TElemTree<TableTreeType>\* Add(TElemTree<TableTreeType>& A);  bool Delete(const TString& A);  TElemTree<TableTreeType>& Search(const TString& A) const;  TableTreeType& operator[](const TString& A) const;  template <typename Type>  friend ostream& operator<<(ostream& ostr, const TTableTree<Type>& A);  };  //----------------------------------------------------------------------  template <typename TableTreeType>  TElemTree<TableTreeType> TTableTree<TableTreeType>::ttt;  //----------------------------------------------------------------------  template <typename TableTreeType>  TTableTree<TableTreeType>::TTableTree() : node(NULL), count(0)  {  }  //----------------------------------------------------------------------  template <typename TableTreeType>  TTableTree<TableTreeType>::TTableTree(const TTableTree<TableTreeType>& A)  {  count = A.count;  node = new TElemTree<TableTreeType>(\*(A.node));  TStackList<TElemTree<TableTreeType>\*> st1;  TStackList<TElemTree<TableTreeType>\*> st2;  st1.Put(node);  st2.Put(A.node);  while (st2.IsEmpty() != true)  {  TElemTree<TableTreeType>\* temp1 = st1.Get();  TElemTree<TableTreeType>\* temp2 = st2.Get();  if (temp2->GetLeft() != NULL)  {  temp1->SetLeft(new TElemTree<TableTreeType>(\*(temp2->GetLeft())));  st1.Put(temp1->GetLeft());  st2.Put(temp2->GetLeft());  temp1->GetLeft()->SetParent(temp1);  }  if (temp2->GetRight() != NULL)  {  temp1->SetRight(new TElemTree<TableTreeType>(\*(temp2->GetRight())));  st1.Put(temp1->GetRight());  st2.Put(temp2->GetRight());  temp1->GetRight()->SetParent(temp1);  }  }  }  //----------------------------------------------------------------------  template <typename TableTreeType>  TTableTree<TableTreeType>::~TTableTree()  {  TStackList<TElemTree<TableTreeType>\*> st1;  st1.Put(node);  while (st1.IsEmpty() != true)  {  TElemTree<TableTreeType>\* temp = st1.Get();  if (temp != NULL)  {  if (temp->GetLeft() != NULL)  st1.Put(temp->GetLeft());  if (temp->GetRight() != NULL)  st1.Put(temp->GetRight());  delete temp;  }  }  }  //----------------------------------------------------------------------  template <typename TableTreeType>  TElemTree<TableTreeType>\* TTableTree<TableTreeType>::Add(TElemTree<TableTreeType>& A)  {  if (count == 0)  {  node = new TElemTree<TableTreeType>(A);  count++;  return node;  }  else  {  TElemTree<TableTreeType>\* a;  TElemTree<TableTreeType>\* b = node;  while (b != NULL)  {  a = b;  if (A.GetKey() > b->GetKey())  b = b->GetRight();  else if (A.GetKey() < b->GetKey())  b = b->GetLeft();  else  throw TException("Error");  }  count++;  if (A.GetKey() > a->GetKey())  {  a->SetRight(new TElemTree<TableTreeType>(A));  a->GetRight()->SetParent(a);  b = a->GetRight();  }  else  {  a->SetLeft(new TElemTree<TableTreeType>(A));  a->GetLeft()->SetParent(a);  b = a->GetLeft();  }  return b;  }  }  //----------------------------------------------------------------------  template <typename TableTreeType>  bool TTableTree<TableTreeType>::Delete(const TString& A)  {  bool flag = false;  TElemTree<TableTreeType>\* a;  TElemTree<TableTreeType>\* b = node;  while (b != NULL)  {  if (A > b->GetKey())  {  a = b;  b = b->GetRight();  }  else if (A < b->GetKey())  {  a = b;  b = b->GetLeft();  }  else  {  flag = true;  break;  }  }  if (b == node && flag == true)  {  node = node->GetRight();  TElemTree<TableTreeType>\* i = b->GetRight();  if (i != NULL)  {  while (i->GetLeft() != NULL)  i = i->GetLeft();  i->SetLeft(b->GetLeft());  if (i->GetLeft() != NULL)  i->GetLeft()->SetParent(i);  }  }  else if (flag == true && A == a->GetLeft()->GetKey())  {  a->SetLeft(b->GetRight());  TElemTree<TableTreeType>\* i = b->GetRight();  if (i != NULL)  {  while (i->GetLeft() != NULL)  i = i->GetLeft();  i->SetLeft(b->GetLeft());  if (i->GetLeft() != NULL)  i->GetLeft()->SetParent(i);  }  }  else if (flag == true && A == a->GetRight()->GetKey())  {  a->SetRight(b->GetRight());  TElemTree<TableTreeType>\* i = b->GetRight();  if (i != NULL)  {  while (i->GetLeft() != NULL)  i = i->GetLeft();  i->SetLeft(b->GetLeft());  if (i->GetLeft() != NULL)  i->GetLeft()->SetParent(i);  }  }  if (flag == true)  {  delete b;  count--;  }  return flag;  }  //----------------------------------------------------------------------  template <typename TableTreeType>  TElemTree<TableTreeType>& TTableTree<TableTreeType>::Search(const TString& A) const  {  if (count == 0)  return ttt;  TElemTree<TableTreeType>\* temp = node;  while (temp != NULL)  {  if (temp->GetKey() < A)  temp = temp->GetRight();  else if (temp->GetKey() > A)  temp = temp->GetLeft();  else  return \*temp;  }  return ttt;  }  //----------------------------------------------------------------------  template <typename TableTreeType>  TableTreeType& TTableTree<TableTreeType>::operator[](const TString& A) const  {  return Search(A).GetData();  }  //----------------------------------------------------------------------  template <typename Type>  ostream& operator<<(ostream& ostr, const TTableTree<Type>& A)  {  TStackList<TElemTree<Type>\*> st1;  st1.Put(A.GetNode());  while (st1.IsEmpty() != true)  {  TElemTree<Type>\* temp = st1.Get();  if (temp->GetRight() != NULL)  st1.Put(temp->GetRight());  if (temp->GetLeft() != NULL)  st1.Put(temp->GetLeft());  ostr << \*temp << endl;  }  return ostr;  } |

## 8.5. Приложение 5:Код программы тестирования

|  |
| --- |
| **sample\_ttabletree.cpp** |
| #include <iostream>  #include <locale.h>  #include "ttabletree.h"  using namespace std;  int main()  {  int data;  int max;  TTableTree<int> look;  TElemTree<int> element;  char memory[100];  cout << "EXAMPLE: TABLE BY TREE"<< endl;  cout << "Count of elements: ";  cin >> max;  for (int i = 0; i < max; i++)  {  cout << " Key of "<< i + 1 <<" element: ";  cin >> memory;  TString key(memory);  element.SetKey(key);  cout << " Data of "<< i + 1 <<" element: ";  cin >> data;  element.SetData(data);  look.Add(element);  }  cout << endl << look;  cout << "ADD ELEMENT\n ";  cout << " Key of new element: ";  cin >> memory;  TString key1(memory);  element.SetKey(key1);  cout << " Data of new element: ";  cin >> data;  element.SetData(data);  look.Add(element);  cout << endl << look;  cout << "DELETE ONE ELEMENT\n ";  cout << " Key of element for delete: ";  cin >> memory;  TString key2(memory);  look.Delete(key2);  cout << endl << look;  return 0;  } |

## 8.6. Приложение 6:Тесты для классов

|  |
| --- |
| **test\_main.cpp** |
| #include <gtest.h>  int main(int argc, char \*\*argv)  {  ::testing::InitGoogleTest(&argc, argv);  return RUN\_ALL\_TESTS();  } |

|  |
| --- |
| **test\_tabletree.cpp** |
| #include "tablesee.h"  #include <gtest.h>  //TESTS FOR CLASS TString  TEST(TString, can\_create\_string\_without\_symbols)  {  ASSERT\_NO\_THROW(TString string);  }  TEST(TString, can\_create\_string\_with\_symbols)  {  ASSERT\_NO\_THROW(TString string("Program"));  }  TEST(TString, can\_create\_copy\_of\_string)  {  TString string("Program");  TString string2(string);  EXPECT\_EQ(string.GetMemory()[0], 'P');  EXPECT\_EQ(string.GetMemory()[6], 'm');  }  TEST(TString, can\_get\_count)  {  TString string("Program");  EXPECT\_EQ(string.GetCount(), 7);  }  TEST(TString, can\_get\_memory)  {  TString string("Program");  ASSERT\_NO\_THROW(string.GetMemory());  EXPECT\_EQ(string.GetMemory()[0], 'P');  EXPECT\_EQ(string.GetMemory()[6], 'm');  }  TEST(TString, can\_use\_assign\_operator)  {  TString string("Program");  TString string2;  string2 = string;  EXPECT\_EQ(string.GetMemory()[0], 'P');  EXPECT\_EQ(string.GetMemory()[6], 'm');  }  TEST(TString, can\_compare\_strings\_part1)  {  TString string("One");  TString string2("Two");  TString string3("One");  EXPECT\_TRUE(string == string3);  EXPECT\_TRUE(string2 != string3);  EXPECT\_FALSE(string == string2);  EXPECT\_FALSE(string != string3);  }  TEST(TString, can\_compare\_strings\_part2)  {  TString string("AAA");  TString string2("AAAAAAA");  EXPECT\_TRUE(string < string2);  EXPECT\_FALSE(string > string2);  EXPECT\_TRUE(string2 > string);  EXPECT\_FALSE(string2 < string);  }  TEST(TString, throw\_when\_use\_wrong\_index\_of\_element)  {  TString string("Program");  ASSERT\_ANY\_THROW(string[-1]);  ASSERT\_ANY\_THROW(string[10]);  }  TEST(TString, can\_return\_element)  {  TString string("Program");  EXPECT\_EQ(string[0], 'P');  EXPECT\_EQ(string[6], 'm');  }  //TESTS FOR CLASS TElemTree  TEST(TElemTree, can\_create\_element\_of\_tabletree\_without\_parametres)  {  ASSERT\_NO\_THROW(TElemTree<int> elem);  }  TEST(TElemTree, can\_create\_element\_of\_tabletree\_with\_parametres)  {  TString string("old");  TString string2("new");  ASSERT\_NO\_THROW(TElemTree<int> element(1, string));  TElemTree<int> element(1, string);  ASSERT\_NO\_THROW(TElemTree<int> element1(2, string2, &element, &element, &element));  }  TEST(TElemTree, can\_create\_copy\_of\_element\_of\_tabletree)  {  TString string("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(element);  EXPECT\_EQ(element.GetData(), 1);  EXPECT\_EQ(element.GetKey(), "new");  }  TEST(TElemTree, can\_use\_assign\_operator)  {  TString string("new");  TElemTree<int> element(1, string);  TElemTree<int> element2;  element2 = element;  EXPECT\_EQ(element.GetData(), 1);  EXPECT\_EQ(element.GetKey(), "new");  }  TEST(TElemTree, can\_get\_data)  {  TString string("new");  TElemTree<int> element(1, string);  EXPECT\_EQ(element.GetData(), 1);  }  TEST(TElemTree, can\_get\_key)  {  TString string("new");  TElemTree<int> element(1, string);  EXPECT\_EQ(element.GetKey(), "new");  }  TEST(TElemTree, can\_get\_right)  {  TString string("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(2, string2, NULL, &element);  EXPECT\_EQ(element2.GetRight(), &element);  }  TEST(TElemTree, can\_get\_left)  {  TString string("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(2, string2, &element);  EXPECT\_EQ(element2.GetLeft(), &element);  }  TEST(TElemTree, can\_get\_parent)  {  TString string("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(2, string2, NULL, NULL, &element);  EXPECT\_EQ(element2.GetParent(), &element);  }  TEST(TElemTree, can\_set\_data)  {  TString string("new");  TElemTree<int> element(1, string);  element.SetData(2);  EXPECT\_EQ(element.GetData(), 2);  }  TEST(TElemTree, can\_set\_key)  {  TString string("old");  TString string1("new");  TElemTree<int> element(1, string);  element.SetKey(string1);  EXPECT\_EQ(element.GetKey(), string1);  }  TEST(TElemTree, can\_set\_right)  {  TString string("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(2, string2);  element2.SetRight(&element);  EXPECT\_EQ(element2.GetRight(), &element);  }  TEST(TElemTree, can\_set\_left)  {  TString string("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(2, string2);  element2.SetLeft(&element);  EXPECT\_EQ(element2.GetLeft(), &element);  }  TEST(TElemTree, can\_set\_parent)  {  TString string("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(2, string2);  element2.SetParent(&element);  EXPECT\_EQ(element2.GetParent(), &element);  }  TEST(TElemTree, can\_compare\_elements)  {  TString string("old");  TString string1("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element1(1, string1);  TElemTree<int> element2(2, string2);  EXPECT\_TRUE(element == element1);  EXPECT\_FALSE(element == element2);  EXPECT\_TRUE(element != element2);  EXPECT\_FALSE(element != element1);  }  //TESTS FOR CLASS TTableTree  TEST(TTableTree, can\_initial\_table)  {  ASSERT\_NO\_THROW(TTableTree<int> table);  }  TEST(TTableTree, can\_create\_copy\_of\_table)  {  TTableTree<int> table;  TString string("old");  TString string2("new");  TElemTree<int> element(1, string);  TElemTree<int> element2(2, string2);  table.Add(element);  table.Add(element2);  TTableTree<int> table2(table);  EXPECT\_EQ(table2.GetCount(), 2);  }  TEST(TTableTree, can\_get\_count)  {  TTableTree<int> table;  TString string("old");  TElemTree<int> element(1, string);  table.Add(element);  TTableTree<int> table2(table);  EXPECT\_EQ(table2.GetCount(), 1);  }  TEST(TTableTree, can\_get\_node)  {  TTableTree<int> table;  EXPECT\_TRUE(table.GetNode() == NULL);  }  TEST(TTableTree, can\_add\_element)  {  TString string("new");  TElemTree<int> element(1, string);  TTableTree<int> table;  table.Add(element);  EXPECT\_EQ(table[string], 1);  }  TEST(TTableTree, can\_delete\_element)  {  TString string("new");  TElemTree<int> element(1, string);  TString string2("two");  TElemTree<int> element2(2, string2);  TTableTree<int> table;  table.Add(element);  EXPECT\_TRUE(table.Delete(string));  }  TEST(TTableTree, can\_search\_element)  {  TString string("one");  TElemTree<int> element(1, string);  TString string2("two");  TElemTree<int> element2(1, string2);  TTableTree<int> table;  table.Add(element);  table.Add(element2);  EXPECT\_TRUE(table.Search(string) == element);  }  TEST(TTableTree, can\_use\_operator\_for\_index)  {  TString string("new");  TElemTree<int> element(1, string);  TTableTree<int> table;  table.Add(element);  EXPECT\_EQ(table[string], 1);  } |