МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

Направление подготовки «Прикладная математика и информатика»

**ОТЧЕТ**

по лабораторной работе

**Просматриваемые, упорядоченные и хэш таблицы.**

**Выполнила:** студентка группы 381603-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_В.В.Жилина

**Проверила:** ст.преп., к.т.н.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_В.Д.Кустикова

Нижний Новгород  
2018

Содержание

Введение 3

1 Постановка задачи 4

2 Руководство пользователя 5

3 Руководство программиста 6

3.1 Структура программы 6

3.2 Структуры данных 6

3.2.1 Абстрактный класс таблиц 6

3.2.2 Шаблон класса неупорядоченных таблиц 6

3.2.3 Шаблонный класс упорядоченных таблиц 7

3.2.4 Шаблон класса хэш-таблиц 8

3.3 Программная реализация 8

3.3.1 Класс TabRecord 8

3.3.2 Абстрактный класс Table 8

3.3.3 Класс ScanTable 9

3.3.4 Класс SortedTable 9

3.3.5 Класс HashTable 9

Заключение 11

Литература 12

Приложение 13

Приложение А. Программная реализация виртуального класса таблиц 13

Приложение Б. Программная реализация просматриваемых таблиц 14

Приложение В. Программная реализация упорядоченных таблиц 16

Приложение Г. Программная реализация хэш таблиц 18

Приложение Д. Основная программа 23

# Введение

В настоящее время таблицы, хранящие какие-либо данные, очень широко используются в современном мире как отличное средство хранения и представления информации. Таблицы широко используются в банковской сфере, бухгалтерском и банковском учете, планировании и распределении ресурсов, обработке больших массивов информации. Они удобны для быстрого поиска информации, её обработки и сортировки по различным фильтрам и пользовательским настройкам.

Таблица-динамическая структура данных. Базисное множество-семейство линейных структур из записей, базисное отношение включения определяется операциями вставки и удаления записей.

Данная лабораторная работа посвящена разработке программы, которая выполняет 3 арифметические операции: вставка, удаление, поиск элемента в различных таблицах, хранящих в себе ключ и значение. В программе реализованы три вида таблиц: несортированные, сортированные и хэш-таблицы.

# Постановка задачи

Цель: Разработать программу, позволяющую хранить данные в трёх различных видах таблиц: упорядоченных, неупорядоченных, хэш-таблицах. Кроме того, необходимо разработать пользовательское консольное приложение, где в качестве хранимых данных выступают полиномы, действия над которыми происходят синхронно в трёх видах таблиц.

Контрольный пример:

Вставленный пользователем полином(рис.1) и полученные таблицы, выведенные на экран(рис.2):

|  |  |
| --- | --- |
|  |  |
| 1. Вставка | 1. Полученные таблицы |

# Руководство пользователя

Для начала работы необходимо запустить файл [sample.exe](../sample/Main.cpp). На экране появится меню(рис.3), при выборе пункта 1-3, появится надпись с просьбой ввести полином (рис 4.) для вставки, удаления или поиска, если оно введено верно появится результат: полученные таблицы или найденный полином, в другом случае выведется ошибка . Пункт 4-распечатать, имеющиеся таблицы. При выборе пункта 5-выход из программы.

|  |  |
| --- | --- |
|  |  |
| 1. Меню программы | 1. Просьба о вводе полинома |

# Руководство программиста

## Структура программы

Программа состоит из 4 проектов:

1. gtest

Содержит в себе [gtest.h](../gtest/gtest.h) и [gtest-all.cpp](../gtest/gtest-all.cc). Файлы содержат google тесты, которые необходимы для проверки корректности работы реализованных классов.

1. polinom\_lib

Содержит в себе:

[Node.h](file:///C:\Users\user\Desktop\mp2-practice-2\ZhilinaL\Lab1\include\Node.h)-заголовочный файл, содержащий объявление шаблонного класса Node и его реализацию.

[Monom.h](file:///C:\Users\user\Desktop\mp2-practice-2\ZhilinaL\Lab1\include\Monom.h)- заголовочный файл, содержащий объявление шаблонного класса monom.

[Polinom.h](file:///C:\Users\user\Desktop\mp2-practice-2\ZhilinaL\Lab1\include\Polinom.h)- заголовочный файл, содержащий объявление шаблонного класса Polinom

[List.h](file:///C:\Users\user\Desktop\mp2-practice-2\ZhilinaL\Lab1\include\List.h)-заголовочный файл, содержащий объявление шаблонного класса list и его реализацию.

[Monom.cpp](file:///C:\Users\user\Desktop\mp2-practice-2\ZhilinaL\Lab1\src\Monom.cpp)-содержит реализацию класса monom

[Polinom.cpp](file:///C:\Users\user\Desktop\mp2-practice-2\ZhilinaL\Lab1\src\Polinom.cpp)-содержит реализацию класса Polinom.

1. tables\_lib

Содержит в себе:

Table.h- заголовочный файл, содержащий объявление абстрактного базового виртуального класса Table и его реализацию.

ScanTable.h- заголовочный файл, содержащий объявление и реализацию неупорядоченных таблиц.

SortedTable.h- заголовочный файл, содержащий объявление и реализацию упорядоченных таблиц.

HashTable.h- заголовочный файл, содержащий объявление и реализацию упорядоченных хэш-таблиц.

1. tables\_test

Содержит в себе [Test\_main.cpp](../test/Test_main.cpp), [test\_polinoms.cpp](../test/test_polinoms.cpp), [test\_list.cpp](../test/test_list.cpp)., test\_Scan\_Tables.cpp, test\_Sorted\_Tables.cpp, test\_Hash\_Tables.cpp. Реализация тестов для проверки корректности работы реализованных классов.

1. sample

Содержит [Main.cpp](file:///C:\Users\user\Desktop\mp2-practice-2\ZhilinaL\Lab1\samples\Main.cpp)– реализация пользовательского интерфейса.

## Структуры данных

### Абстрактный класс таблиц

Для описания разных видов таблиц был создан абстрактный базовый класс Table. Для его реализации был написан шаблонный класс TabRecord - ячейка таблицы. Сам класс представляет собой структуру с четырьмя полями:

-массив ячеек таблицы

-размер таблицы

-текущее количество записей в таблице

-индекс текущего элемента

Абстрактный класс содержит виртуальные методы: увеличение размера таблицы, вставка, поиск и удаление, методы навигации. Часть виртуальных методов определена в абстрактном классе и используется классами-наследниками.

### Шаблон класса неупорядоченных таблиц

Унаследован от класса Table, новых полей не имеет.

Описание алгоритмов:

1) Вставка:

1.Если текущий размер равен максимальному, то переупаковываем таблицу.

2.Проходим таблицу с начала пока она не закончится или ключ не будет равен переданному на вход.

3. Если ключ, переданный на вход, уже присутствует в массиве, то происходит исключение.

4.Иначе записать данные в конец таблицы.

2) Удаление:

1. Проходим таблицу с начала пока она не закончится или ключ не будет равен переданному на вход.

2. Если переданный ключ присутствует в таблице- удаляем.

3.На место удаленного элемента записываем последний элемент массива, при этом уменьшаем количество текущих записей на 1.

4. Иначе-исключение.

3) Поиск:

1.Если таблица пуста-исключение.

2. Проходим таблицу с начала пока она не закончится или ключ не будет равен переданному на вход.

3.Если переданный ключ присутствует в таблице-передаем на выход указатель на данные из ячейки, содержащей ключ.

4. Иначе-исключение.

### Шаблонный класс упорядоченных таблиц

Унаследован от класса Table, новых полей не имеет.

Описание алгоритмов:

1) Вставка:

1. Если текущий размер равен максимальному, то переупаковываем таблицу.

2. Проходим таблицу с начала пока она не закончится или ключ не будет равен переданному на вход, сравнивая элемент, который хотим вставить, с текущим элементом.

3. Если ключ, переданный на вход, уже присутствует в массиве, то происходит исключение.

4.Если текущий больше, — вставляем элемент на его место. Этот элемент и все последующие сдвигаем на одно место вперед.

5.Если текущий меньше — сравниваем со следующим элементом. Алгоритм продолжается до тех пор, пока массив элементов не закончится или элемент не будет вставлен.

2) Удаление:

1.Проходим таблицу с начала пока она не закончится или ключ не будет равен переданному на вход.

2. Если переданный ключ присутствует в таблице- удаляем. При удалении происходит переупаковка массива, чтобы не нарушалась упорядоченность в таблице. Все элементы, следующие за удаленным, сдвигаются на одно место влево.

3. Иначе-исключение.

3) Поиск:

Осуществляется с помощью алгоритма бинарного поиска по упорядоченному массиву.

1.Определение значения элемента в середине структуры данных. Полученное значение сравнивается с ключом.

2.Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.

3.Далее повторяем проделанное уже в половине данных таблице. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.

4. Если переданный ключ присутствует в таблице-передаем на выход указатель на данные из ячейки, содержащей ключ.

5. Иначе-исключение.

### Шаблон класса хэш-таблиц

Унаследован от класса Table**,** к имеющимся полям добавлено одно поле:

-flag - целочисленный массив «маркеров», хранит статус ячейки (-1-удалена, 0-пуста, 1-заполнена)

Описание алгоритмов:

1) Хэш-функция:

Генерирует на основе ключа элемента целочисленное число, которое будет являться индексом строки таблицы для записи этого ключа.

1.Идем по строке и суммируем все символы из строки.

2.Умножаем сумму на , где n – количество символов в строке.

3.Устанавливаем полученное число как зерно для генератора псевдослучайных чисел.

4.Возвращаем сгенерированное число, которое мы ограничили размерами таблицы.

2) Вставка:

1. Если текущий размер равен максимальному, то переупаковываем таблицу.

2. Вызываем хэш-функцию для входного ключа и записываем в текущий индекс.

3. Если flag по этому индексу 0 или -1, то есть ячейка пуста или удалена, то записать новую строку на этот индекс, увеличить текущий размер на единицу.

4. Иначе-исключение.

3) Удаление:

1.Если поиск по входному ключу вернул null, то исключение.

2. Иначе- удаляем найденную запись, меняем значение flag на -1 и уменьшаем текущий размер на 1.

4) Поиск:

1. Если таблица пуста-исключение.

2. Вызываем хэш-функцию для входного ключа и записываем в текущий индекс.

3. Если по полученному индексу строка занята, то возвращаем указатель на данные из ячейки, содержащей ключ.

4.Иначе – исключение;

## Программная реализация

### Класс TabRecord

Поля:

string key – ключ записи в таблице

T\* data – данные, хранящиеся в записи

Методы:

TabRecord() { key = "Empty record"; data = new T(NULL); }-конструктор по умолчанию, создает пустую запись с ключом “Empty record”

TabRecord(string CKey , T CData ) –конструктор, получающий на вход новый ключ и данные

TabRecord(const TabRecord& CTabR) – конструктор копирования

TabRecord(const TabRecord& CTabR) TabRecord<T>& operator=(const TabRecord<T>& TabR) – перегрузка оператора присваивания

### Абстрактный класс Table

Поля:

TabRecord<T>\*\* rows – массив из указателей на TabRecord

unsigned int maxRecord – размер таблицы

unsigned int currRecord – текущее количество элементов в таблице

int currIndex – индекс текущего элемента

Методы:

virtual void Realloc() = 0 – переупаковка таблицы, в случае ее заполненности

Table(unsigned int i = 10)-конструктор, создает таблицу размера i

virtual ~Table() { delete[] rows; } - деструктор

virtual void Insert(const T Data, const string Key) = 0-вставка в таблицу пары данные и ключ

virtual void Delete(const string Key) = 0 – удаление элемента по ключу

virtual T\* Search(const string Key) = 0 – поиск элемента по ключу

virtual void Reset() – смещение на начало таблицы

virtual bool IsTabEnded() const – проверка на то, закончилась ли таблица

bool IsFull() const { return currRecord == maxRecord; } – проверка на заполненность таблицы

bool IsEmpty() const { return currRecord == 0; } – проверка на то, является ли таблица пустой

virtual void SetNext()- увеличение текущего индекса

virtual T\* GetCurr() const – возвращает текущие данные в ячейке

int GetCurrRecords() const { return currRecord; } – возвращает текущий размер таблицы

int GetMaxRecords() const { return maxRecord; } – возвращает размер таблицы

### Класс ScanTable

Класс неупорядоченных таблиц, унаследован от базового абстрактного класса Table, параметризован аналогично.

Поля:

Отсутствуют

Методы:

virtual void Realloc() – переупаковка таблицы, в случае заполненности

ScanTable(unsigned int i = 10) : Table<T>(i) {}- конструктор, аналогичный конструктору класса Table, создает таблицу размера i

ScanTable(const ScanTable<T>& TabToCopy) – конструктор копирования

~ScanTable() { } - деструктор

virtual void Insert(const T Data, const string Key) – вставка в таблицу пары данные и ключ

virtual T\* Search(const string Key) – поиск по ключу

virtual void Delete(const string Key) – удаление по ключу

template<class T> friend std::ostream& operator<< (std::ostream& os, const ScanTable<T>& Tab) - перегрузка оператора вывода

### Класс SortedTable

Класс упорядоченных таблиц, унаследован от базового абстрактного класса Table, параметризован аналогично.

Поля:

Отсутствуют

Методы:

virtual void Realloc() – переупаковка таблицы, в случае заполненности

SoredTable(unsigned int i = 10) : Table<T>(i) {}- конструктор, аналогичный конструктору класса Table, создает таблицу размера i

SortedTable(const ScanTable<T>& TabToCopy) – конструктор копирования

~SortedTable() { } - деструктор

virtual void Insert(const T Data, const string Key) – вставка в таблицу пары данные и ключ

virtual T\* Search(const string Key) – поиск по ключу

virtual void Delete(const string Key) – удаление по ключу

template<class T> friend std::ostream& operator<< (std::ostream& os, const SortedTable<T>& Tab) - перегрузка оператора вывода

### Класс HashTable

Класс упорядоченных таблиц, унаследован от базового абстрактного класса Table, параметризован аналогично c добавлением одного параметра flag.

Поля:

int\* flag – массив «маркеров», хранит статус ячейки (-1-удалена, 0-пуста, 1-заполнена)

Методы:

int HashFunc(string key) const – хэш функция

void Realloc() – переупаковка таблицы, в случае заполненности

HashTable(unsigned int i = 10) - конструктор, создает пустую таблицу размером i

HashTable(const HashTable<T>& TabToCopy) – конструктор копирования

~HashTable() { delete[] flag; } - деструктор

virtual void Insert(const T data, const string key) – вставка в таблицу пары данные и ключ

virtual T\* Search(const string key) – поиск по ключу

virtual void Delete(const string key) – удаление по ключу

virtual void SetNext() – увеличение текущего индекса

virtual T\* GetCurr() const – возвращает текущие данные в ячейке

virtual void Reset() – смещение в начало таблицы

virtual bool IsTabEnded() const – проверка на то, закончилась ли таблица

template<class T> friend std::ostream& operator<< (std::ostream& os, const HashTable<T>& Tab) – перегрузка оператора вывода

# Заключение

В лабораторной работе была разработана программа, выполняющая вставку, поиск, удаления элементов, заданных пользователем в трех видах таблиц. Так же была решена проблема коллизий в хэш-таблицах. Так же алгоритм обеспечивает уникальность ключа во всех таблицах.

# Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, Москва 2013
2. Гергель В.П. Рабочие материалы к учебному курсу «Методы программирования». 2002 г.

# Приложение

## Приложение А. Программная реализация виртуального класса таблиц

Table.h

#pragma once

#include <string>

#include <iostream>

using namespace std;

template <typename T>

class TabRecord

{

public:

string key;

T\* data;

TabRecord() { key = "Empty record"; data = new T(NULL); }

TabRecord(string CKey , T CData ) { key = CKey; data = new T(CData);}

TabRecord(const TabRecord& CTabR) { key = CTabR.key; data = new T(\*(CTabR.data)); }

TabRecord<T>& operator=(const TabRecord<T>& TabR) { data = new T(\*(TabR.data)); key = TabR.key; return \*this; }

};

template <typename T>

class Table

{

protected:

TabRecord<T>\*\* rows;

unsigned int maxRecord;

unsigned int currRecord;

int currIndex;

virtual void Realloc() = 0;

public:

Table(unsigned int i = 10);

virtual ~Table() { delete[] rows; };

//виртуальные

virtual void Insert(const T Data, const string Key) = 0;

virtual void Delete(const string Key) = 0;

virtual T\* Search(const string Key) = 0;

//методы навигации

virtual void Reset();

virtual bool IsTabEnded() const { return currIndex == currRecord || currIndex == -1; };

bool IsFull() const { return currRecord == maxRecord; };

bool IsEmpty() const { return currRecord == 0; };

virtual void SetNext();

virtual T\* GetCurr() const;

int GetCurrRecords() const { return currRecord; };

int GetMaxRecords() const { return maxRecord; };

};

template<typename T>

Table<T>::Table(unsigned int i)

{

maxRecord = i;

currRecord = 0;

currIndex = -1;

rows = new TabRecord<T>\*[maxRecord];

}

template<typename T>

void Table<T>::Reset()

{

if(currRecord > 0)

currIndex = 0;

else

currIndex = -1;

}

template<typename T>

T\* Table<T>::GetCurr() const

{

T\* tmp;

if (currIndex >= 0 && currIndex<currRecord)

{

tmp = rows[currIndex]->data;

}

else

{

throw (string)"Table Is Empty";

}

return tmp;

}

template<typename T>

void Table<T>::SetNext()

{

if (currIndex != -1)

currIndex++;

else

throw (string)"Table Is Empty";

if (IsTabEnded())

Reset();

}

## Приложение Б. Программная реализация просматриваемых таблиц

ScanTable.h

#pragma once

#include "Table.h"

template <typename T>

class ScanTable : public Table<T>

{

protected:

virtual void Realloc();

public:

ScanTable(unsigned int i = 10) : Table<T>(i) {};

ScanTable(const ScanTable<T>& TabToCopy);

~ScanTable() { };

//виртуальные

virtual void Insert(const T Data, const string Key);

virtual T\* Search(const string Key);

virtual void Delete(const string Key);

template<class T> friend std::ostream& operator<< (std::ostream& os, const ScanTable<T>& Tab);

};

template <typename T>

ScanTable<T>::ScanTable(const ScanTable<T>& TabToCopy)

{

this->maxRecord = TabToCopy.maxRecord;

this->currIndex = TabToCopy.currIndex;

this->currRecord = TabToCopy.currRecord;

delete[] rows;

this->rows = new TabRecord<T>\*[maxRecord];

if (!IsEmpty())

for (int i = 0; i < this->currRecord; i++)

this->rows[i] = new TabRecord<T>(\*(TabToCopy.rows[i]));

}

template <typename T>

void ScanTable<T>::Realloc()

{

unsigned int NewMaxRecords = (unsigned int)(this->maxRecord\*1.5);

TabRecord<T>\*\* tmp = new TabRecord<T>\*[NewMaxRecords];

for (int i = 0; i < this->maxRecord; i++)

tmp[i] = this->rows[i];

this->maxRecord = NewMaxRecords;

delete[] this->rows;

this->rows = tmp;

}

template <typename T>

void ScanTable<T>::Insert(const T Data, const string Key)

{

if (this->IsFull())

this->Realloc();

TabRecord<T> Row(Key, Data);

this->Reset();

while (!(this->IsTabEnded()) && Key != this->rows[this->currIndex]->key)

this->currIndex++;

if (this->IsEmpty())

{

this->currIndex++;

}

if (this->currIndex == this->currRecord)

{

this->rows[this->currIndex] = new TabRecord<T>(Row);

this->currRecord++;

this->Reset();

}

else

{

this->Reset();

throw "Your key isn't unique";

}

}

template <typename T>

T\* ScanTable<T>::Search(const string Key)

{

this->Reset();

T\* tmp = nullptr;

if (this->IsEmpty())

throw (string)"Cant Search In Empty Table";

while (!(this->IsTabEnded()) && Key != this->rows[this->currIndex]->key)

this->currIndex++;

if (!(this->IsTabEnded()))

tmp = this->rows[this->currIndex]->data;

else

{

throw "Your key wasn't found";

}

return tmp;

}

template <typename T>

void ScanTable<T>::Delete(string Key)

{

if (Search(Key) == nullptr)

{

throw "Can't Delete From Empty Table";

}

else

{

this->rows[this->currIndex]->data = this->rows[this->currRecord - 1]->data;

this->rows[this->currIndex]->key = this->rows[this->currRecord - 1]->key;

this->rows[this->currRecord - 1] = NULL;

this->currRecord = this->currRecord - 1;

this->currIndex = this->currRecord - 1;

}

/\*this->Reset();

if (this->IsEmpty())

throw "Can't Delete From Empty Table";

this->Search(Key);

if (this->currRecord > 1)

this->rows[this->currIndex] = this->rows[--this->currRecord];

else

this->currRecord = 0;\*/

}

template <typename T>

std::ostream& operator<< (std::ostream& os, const ScanTable<T>& Tab)

{

unsigned int i = 0;

while (i < Tab.currRecord)

{

os << i << ". " << Tab.rows[i]->key << " | " << \*(Tab.rows[i]->data) << std::endl;

i++;

}

if (Tab.currRecord == 0)

os << "Table is Empty";

return os;

}

## Приложение В. Программная реализация упорядоченных таблиц

SortedTable.h

#include "Table.h"

template <typename T>

class SortedTable : public Table<T>

{

protected:

void Realloc();

public:

SortedTable(unsigned int i = 10): Table<T>(i) {};

SortedTable(const SortedTable<T>& TabToCopy);

~SortedTable() {};

//виртуальные

virtual void Insert(const T data, const string key);

virtual T\* Search(const string key);

virtual void Delete(const string key);

template<class T> friend std::ostream& operator<< (std::ostream& os, const SortedTable<T>& Tab);

};

template <typename T>

SortedTable<T>::SortedTable(const SortedTable<T>& TabToCopy)

{

maxRecord = TabToCopy.maxRecord;

currIndex = TabToCopy.currIndex;

currRecord = TabToCopy.currRecord;

delete[] rows;

rows = new TabRecord<T>\*[maxRecord];

if (!IsEmpty())

for (int i = 0; i < currRecord; i++)

rows[i] = new TabRecord<T>(\*(TabToCopy.rows[i]));

}

template <typename T>

void SortedTable<T>::Realloc()

{

unsigned int NewMaxRecords = (unsigned int)(this->maxRecord\*1.5);

TabRecord<T>\*\* tmp = new TabRecord<T>\*[NewMaxRecords];

for (int i = 0; i < this->maxRecord; i++)

tmp[i] = this->rows[i];

this->maxRecord = NewMaxRecords;

delete[] this->rows;

this->rows = tmp;

}

template <typename T>

void SortedTable<T>::Insert(const T data, const string key)

{

if (this->IsFull())

this->Realloc();

TabRecord<T> Row(key, data);

this->Reset();

this->Reset();

while (!(this->IsTabEnded()) && Row.key >= this->rows[this->currIndex]->key)

{

if (this->rows[this->currIndex]->key == Row.key)

{

throw "Your key isn't unique";

}

this->currIndex++;

}

if (this->IsEmpty())

this->currIndex++;

this->currRecord++;

for (int i = this->currRecord - 1; i > this->currIndex; i--)

{

this->rows[i] = this->rows[i - 1];

}

this->rows[this->currIndex] = new TabRecord<T>(Row);

this->Reset();

}

template <typename T>

T\* SortedTable<T>::Search(const string key)

{

this->Reset();

T\* tmp = nullptr;

if (!(this->IsEmpty()))

{

int i = -1, j = this->currRecord;

int mid;

while (i < j - 1)

{

mid = (j + i) / 2;

if (key >= this->rows[mid]->key)

{

i = mid;

}

else

{

j = mid;

}

}

if (key == this->rows[i]->key)

{

this->currIndex = i;

tmp = this->rows[this->currIndex]->data;

}

else

{

throw "Your key wasn't found";

}

}

else

throw "Table Is Empty";

return tmp;

}

template <typename T>

void SortedTable<T>::Delete(const string key)

{

this->Reset();

if (this->IsEmpty())

throw "Can't Delete From Empty Table";

this->Search(key);

if (this->currRecord > 1)

{

this->currRecord--;

for (int i = this->currIndex; i < this->currRecord; i++)

{

this->rows[i] = this->rows[i + 1];

}

this->Reset();

}

else

this->currRecord = 0;

/\*if (this->Search(key) == nullptr)

{

throw "Can't Delete From Empty Table.";

}

else

{

int i = 0;

while (this->rows[i]->key > key)

{

i++;

}

for (int j = i; j < this->currRecord - 1; j++)

{

this->rows[j] = this->rows[j + 1];

}

this->rows[this->currRecord - 1] = NULL;

this->currRecord = this->currRecord - 1;

}\*/

}

template <typename T>

std::ostream& operator<< (std::ostream& os, const SortedTable<T>& Tab)

{

unsigned int i = 0;

while (i < Tab.currRecord)

{

os << i << ". " << Tab.rows[i]->key << " | " << \*(Tab.rows[i]->data) << std::endl;

i++;

}

if (Tab.currRecord == 0)

os << "Table is Empty";

return os;

}

## Приложение Г. Программная реализация хэш таблиц

HashTable.h

#pragma once

#include "table.h"

#pragma once

#include "table.h"

template <typename T>

class HashTable : public Table<T>

{

protected:

int\* flag;

int HashFunc(string key) const;

void Realloc();

public:

HashTable(unsigned int i = 10);

HashTable(const HashTable<T>& TabToCopy);

~HashTable() { delete[] flag; };

//виртуальные

virtual void Insert(const T data, const string key);

virtual T\* Search(const string key);

virtual void Delete(const string key);

//навигация

virtual void SetNext();

virtual T\* GetCurr() const;

virtual void Reset();

virtual bool IsTabEnded() const { return this->currIndex == this->currRecord || this->currIndex == -1; };// return false;

template<class T> friend std::ostream& operator<< (std::ostream& os, const HashTable<T>& Tab);

};

template <typename T>

HashTable<T>::HashTable(unsigned int i) : Table<T>(i)

{

flag = new int[i];

for (int j = 0; j < i; j++)

flag[j] = 0; // 0 пуста, 1 -заполнена, -1 - удалена

}

template <typename T>

HashTable<T>::HashTable(const HashTable<T>& TabToCopy)

{

maxRecord = TabToCopy.maxRecord;

currIndex = TabToCopy.currIndex;

currRecord = TabToCopy.currRecord;

delete[] rows;

delete[] flag;

rows = new TabRecord<T>\*[maxRecord];

flag = new int[maxRecord];

for (int j = 0; j < maxRecord; j++)

flag[j] = TabToCopy.flag[j];

if (!IsEmpty())

for (int i = 0; i < maxRecord; i++)

{

if (flag[i] == 1)

rows[i] = new TabRecord<T>(\*(TabToCopy.rows[i]));

}

}

template <typename T>

void HashTable<T>::Reset()

{

if (!(this->IsEmpty()))

{

this->currIndex = 0;

while (flag[this->currIndex] != 1)

this->currIndex++;

}

else

this->currIndex = -1;

}

template <typename T>

T\* HashTable<T>::GetCurr() const

{

T\* tmp = nullptr;

if (!(this->IsEmpty()))

{

tmp = this->rows[this->currIndex]->data;

}

else

{

throw "Table Is Empty";

}

return tmp;

}

template <typename T>

void HashTable<T>::SetNext()

{

if (!(this->IsEmpty()))

{

this->currIndex++;

while (flag[this->currIndex] != 1)

this->currIndex = (this->currIndex + 1) % this->maxRecord;

}

else

throw "Table Is Empty";

}

template <typename T>

void HashTable<T>::Realloc()

{

unsigned int OldMaxRecords = this->maxRecord;

this->maxRecord \*= 1.5;

TabRecord<T>\*\* tmp = new TabRecord<T>\*[OldMaxRecords];

for (int i = 0; i < OldMaxRecords; i++)

{

tmp[i] = new TabRecord<T>(\*(this->rows[i]));

}

delete[] this->rows;

this->currRecord = 0;

this->currIndex = -1;

delete[] flag;

flag = new int[this->maxRecord];

for (int i = 0; i < this->maxRecord; i++)

flag[i] = 0;

this->rows = new TabRecord<T>\*[this->maxRecord];

for (int i = 0; i < OldMaxRecords; i++)

{

this->Insert(\*(tmp[i]->data), tmp[i]->key); // ?

}

}

template<typename T>

int HashTable<T>::HashFunc(string key) const

{

unsigned int i = 0;

for (int j = 0; j < key.length(); j++)

i += (char)key[j];

for (int j = 0; j < key.length(); j++)

i \*= 13;

srand(i);

unsigned int h = rand();

return h % this->maxRecord;

}

template <typename T>

void HashTable<T>::Insert(const T data, const string key)

{

if (this->IsFull())

this->Realloc();

TabRecord<T> Row(key, data);

this->Reset();

this->currIndex = HashFunc(key);

if (flag[this->currIndex] == 0 || flag[this->currIndex] == -1)

{

this->rows[this->currIndex] = new TabRecord<T>(Row);

this->currRecord++;

flag[this->currIndex] = 1;

this->Reset();

}

else

{

if (this->rows[this->currIndex]->key != key)

{

int ind = this->currIndex;

while (flag[this->currIndex] == 1 && this->currIndex + 1 != ind)

this->currIndex = (this->currIndex + 1) % this->maxRecord; //

this->rows[this->currIndex] = new TabRecord<T>(Row);

this->currRecord++;//

flag[this->currIndex] = 1;

this->Reset();

}

else

{

this->Reset();

throw "Your key isn't unique";

}

}

}

template <typename T>

T\* HashTable<T>::Search(const string Key)

{

this->Reset();

T\* tmp = nullptr;

if (this->IsEmpty())

throw (string)"Can't Search In Empty Table";

this->currIndex = HashFunc(Key);

if(flag[this->currIndex] ==1)

tmp = this->rows[this->currIndex]->data;

else

throw "Your key wasn't found, was deleted or nothing";

}

template <typename T>

void HashTable<T>::Delete(string key)

{

if (Search(key) == nullptr)

{

throw "Can't Delete From Empty Table.";

}

else

{

this->rows[this->currIndex] = NULL;

flag[this->currIndex] = -1;

this->currRecord = this->currRecord - 1;

}

}

template <typename T>

std::ostream& operator<< (std::ostream& os, const HashTable<T>& Tab)

{

unsigned int i = 0;

while (i < Tab.maxRecord)

{

if (Tab.flag[i] == 1)

{

os << i << ". " << Tab.rows[i]->key << " | " << \*(Tab.rows[i]->data) << std::endl;

}

i++;

}

if (Tab.currRecord == 0)

os << "Table is Empty";

return os;

}

}

## Приложение Д. Основная программа

#include "ScanTable.h"

#include "HashTable.h"

#include "SortedTable.h"

#include "Polinom.h"

#include <conio.h>

using namespace std;

int main()

{

ScanTable<Polinom> A(10);

SortedTable<Polinom> B;

HashTable<Polinom> C;

string Key, Str;

Polinom P;

int i = 0;

while (i != 5)

{

cout << "Choose your action: " << endl;

cout << "1.Input " << endl;

cout << "2.Delete" << endl;

cout << "3.Search" << endl;

cout << "4.Print tables" << endl;

cout << "5.End" << endl;

cin >> i;

switch (i)

{

case 1:

{

system("cls");

cout << "Enter your polinom" << endl;

cin >> Key;

P = Polinom(Key);

try { A.Insert(P, Key); }

catch (...) { cout << "not unique key" << endl; }

try { B.Insert(P, Key); }

catch (...) { cout << "not unique key" << endl; }

try { C.Insert(P, Key); }

catch (...) { cout << "not unique key" << endl; }

cout << "ScanTable: " << endl << A << endl;

cout << "SortedTable :" << endl << B << endl;

cout << "HashTable :" << endl << C << endl << endl;

break;

}

case 2:

{

system("cls");

cout << "Enter your polinom:" << endl;

cin >> Key;

try { A.Delete(Key); }

catch (...) { cout << "element doesn't exist" << endl; }

try { B.Delete(Key); }

catch (...) { cout << "element doesn't exist" << endl; }

try { C.Delete(Key); }

catch (...) { cout << "element doesn't exist" << endl; }

cout << "ScandTable: " << endl << A << endl;

cout << "Sortedable :" << endl << B << endl;

cout << "HashTable :" << endl << C << endl << endl;

break;

}

case 3:

{

system("cls");

cout << "Enter your polinom:" << endl;

cin >> Key;

try { cout << "find: " << \*(A.Search(Key)) << endl; }

catch (...) { cout << "element doesn't exist" << endl; }

try { cout << "find: " << \*(B.Search(Key)) << endl; }

catch (...) { cout << "element doesn't exist" << endl; }

try { cout << "find: " << \*(C.Search(Key)) << endl; }

catch (...) { cout << "element doesn't exist" << endl; }

break;

}

break;

case 4:

{

system("cls");

cout << "ScanTable : " << A << endl;

cout << "SortedTable : " << B << endl;

cout << "HashTable : " << C << endl;

break;

}

default:

{

cout << "try again" << endl;

break;

}

}

}

return 0;

}