МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное 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 учебной практике

**«Организация доступа к данным по ключу»**

**Выполнил**: студент группы 381603-1

Сарынин С.В.

**Проверила**: к.т.н., доцент каф. МОСТ

института ИТММ

Кустикова В.Б.

Нижний Новгород

2018

**Содержание**

[Введение 3](#_Toc515881413)

[1. Постановка задачи 4](#_Toc515881414)

[1. Руководство пользователя 5](#_Toc515881415)

[2. Руководство программиста 6](#_Toc515881416)

[1. Структура программы 6](#_Toc515881417)

[2. Структура данных 7](#_Toc515881418)

[Шаблонный класс TabRecord 7](#_Toc515881419)

[Абстрактный класс Table 7](#_Toc515881420)

[Шаблонный класс неупорядоченной таблицы 9](#_Toc515881421)

[Шаблонный класс упорядоченной таблицы 9](#_Toc515881422)

[Шаблонный класс хэш-таблицы 9](#_Toc515881423)

[3. Описание алгоритмов 10](#_Toc515881424)

[шаблонный класс неупорядоченной таблицы 10](#_Toc515881425)

[шаблонный класс упорядоченной таблицы 10](#_Toc515881426)

[шаблонный класс хэш-таблицы 10](#_Toc515881427)

[Заключение 11](#_Toc515881428)

[Литература 12](#_Toc515881429)

[Приложение 13](#_Toc515881430)

Введение

Данная лабораторная посвящена теме “Таблицы”. Целью работы является разработка консольного приложения, выполняющего следующие действия: вставка, удаление и поиск в таблице. Таблица в данной лабораторной работе представляет собой пару элементов ключ–данные. Ключ уникален для каждого элемента.

В отчёте приводится постановка задачи, разработанные алгоритмы и структуры данных, использующиеся в программе.

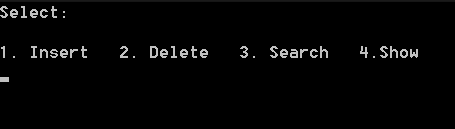
1. Постановка задачи

Разработать программу, позволяющую хранить данные в трёх различных видах таблиц: упорядоченных, неупорядоченных, хэш-таблицах. Таблицы должны поддерживать три операции по работе с данными: вставка, поиск и удаление.

В качестве данных, хранящихся в таблицах выступают полиномы а в качестве ключа строковое представление полинома.

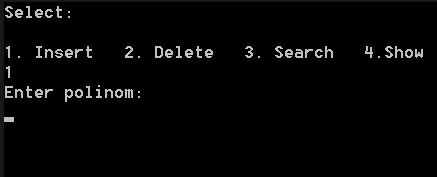
1. Руководство пользователя

Для начала работы с программой необходимо запустить файл polinom.exe. На экране появится меню в котором необходимо выбрать нужный пункт: вставку, удаление, поиск или просмотр содержимого таблиц.



1. начало работы с программой

При выборе пункта 1,2,3 программа предложит ввести полином, далее для выхода из программы необходимо нажать .esc, для продолжения любую клавишу.



1. работа с вводом, удалением, поиском

При выборе пункта 4 программа выведет содержимое таблиц, далее для выхода из программы необходимо нажать .esc, для продолжения любую клавишу.

1. Руководство программиста
2. Структура программы

Программа состоит из четырех проектов.

*gtest1*

Необходим для работы с Google Test.

*polinom\_lib*

Содержит:

1. node.h содержит объявление шаблонного класса Node и его реализацию.
2. List.hсодержит объявление шаблонного класса List и его реализацию.
3. monom.h содержит объявление класса Monom и его реализацию.
4. polinom*.*h содержит объявление класса Polinom.
5. table.h содержит объявление и реализацию абстрактного класса Table.
6. unsorted\_table.h содержит объявление и реализацию класса неупорядоченных таблиц.
7. sorted\_table.h содержит объявление и реализацию класса упорядоченных таблиц.
8. hash\_table содержит объявление и реализацию класса хэш – таблиц.
9. polinom.cpp содержит реализацию класса Polinom.
10. monom.cpp содержит реализацию класса Monom.

polinom\_tests

Включает в себя тесты для проверки функций содержащихся в polinom\_lib:

1. test\_main.cpp
2. test\_polinom.cpp
3. test\_list.cpp
4. test\_hash\_t.cpp.
5. test\_sort\_t.cpp
6. test\_unsort\_t.cpp

polinom\_ sample

Содержит main\_table.cpp – реализация меню программы.

1. Структура данных

Шаблонный класс TabRecord

поля:

string Key–ключ

T Data–поле для записи данных

методы:

TabRecord() – конструктор по умолчанию

TabRecord(string K , T D ) – конструктор

TabRecord(const TabRecord& TR) – конструктор копирования

TabRecord<T>& operator=(const TabRecord<T>& TR) – перегрузка оператора равно

Абстрактный класс Table.

поля:

TabRecord<T>\*\* Records – массив указателей на TabRecord

int MaxRecords – максимальное число записей в таблице

int CurrSize – текущее количество записей в таблице

int CurrIndex – текущий индекс

методы:

Table(int i = 10) – конструктор таблицы

Table(const Table<T>& TabCopy) – конструктор копирования

virtual ~Table() – деструктор

virtual void Insert(const T Data, const string Key) = 0 – функция вставки в таблицу

virtual void Delete(const string key) = 0 – функция удаления из таблицы

virtual T\* Search(const string Key) = 0 – функция поиска по таблице

virtual void Reset() – выставляет CurrInex на начало

virtual bool IsEnded() const

bool IsFull() const – проверяет не заполнена ли таблица

bool IsEmpty() const – проверка на пустоту таблицы

virtual void SetNext() – перемещает индекс следующую запись в таблице

virtual T\* GetCurr() const – возвращает данные текущей записи в таблице

Шаблонный класс неупорядоченной таблицы.

поля:

Унаследованы от класса Table

методы:

Унаследованы от класса Table

Шаблонный класс упорядоченной таблицы.

поля:

Унаследованы от класса Table

методы:

Унаследованы от класса Table

Шаблонный класс хэш-таблицы.

поля:

Помимо унаследованных от Table добавлен целочисленный массив в котором сохраняется состояние записи в таблице.

методы:

Унаследованы от класса Table

int HashFunc(string Key) const –формирует индекс строки таблицы, на основе ключа, в которую будут записаны данные и ключ.

1. Описание алгоритмов

шаблонный класс неупорядоченной таблицы

Вставка–элементы просто записываются друг за другом

Удаление – находим необходимую запись, удаляем, перезаписываем последний элемент на место удаленного, уменьшаем количество записей на 1.

Поиск – полным обходом находим нужную запись.

шаблонный класс упорядоченной таблицы

Вставка – происходит упорядоченно. Пока массив не закончился, поочередно сравниваем элемент с текущим до тех пор пока он не будет больше или равен текущему. Тогда все элементы за еущим сдвигаем вправо и вставляем элемент.

Удаление – при помощи алгоритма бинарного поиска находится необходимая запись таблицы, удаляется и все элементы стоящие после удаленного сдвигаются

Поиск – находим необходимую запись таблицы, выводим ее содержимое.

шаблонный класс хэш-таблицы

Вставка – запоминаем значение хэш-функции это будет номером элемента таблицы указателей. Если коллизии не возникает выполняем вставку.

Удаление – запоминаем значение хэш-функции. Удаляем если коллизии нет.

Поиск – запоминаем значение хэш-функции, , если коллизии нет, выводим данные.

Заключение

В данной работе были реализованы три вида таблиц: сортированные, несортированные и хэш-таблицы, кроме этого сделано меню для работы с таблицами. Можно выполнять вставку, удаление, поиск записей и вывод содержимого в реализованных таблицах.

Литература

1. Рабочие материалы к учебному курсу «Методы программирования». *Гергель В.П*. 2002 г.

Приложение

#pragma once

#include <string>

#include <iostream>

using std::string;

using std::endl;

using std::cout;

template <typename T>

class TabRecord

{

public:

string Key;

T Data;

TabRecord() { Key = "EMPTY"; Data = 0;}

TabRecord(string K , T D ) { Key = K; Data = D;}

TabRecord(const TabRecord& TR) { Key = TR.Key; Data = TR.Data; }

TabRecord<T>& operator=(const TabRecord<T>& TabR) { Data = TabR.Data; Key = TabR.Key; return \*this; }

};

template <typename T>

class Table

{

protected:

TabRecord<T>\*\* Records;

int MaxRecords;

int CurrSize;

int CurrIndex;

virtual void Realloc() = 0;

public:

Table(int i = 10);

Table(const Table<T>& TabCopy);

virtual ~Table()

{

//for (int i = 0; i < MaxRecords; i++) delete Records[i];

delete[] Records;

};

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

virtual void Delete(const string key) = 0;

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

virtual void Reset();

virtual bool IsEnded() const { return CurrIndex == CurrSize || CurrIndex == -1; };

bool IsFull() const { return CurrSize == MaxRecords; };

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

virtual void SetNext();

virtual T\* GetCurr() const;

};

template<typename T>

Table<T>::Table(int i)

{

MaxRecords = i;

CurrSize = 0;

CurrIndex = -1;

Records = new TabRecord<T>\*[MaxRecords];

}

template<typename T>

void Table<T>::Reset()

{

if(CurrSize > 0)

CurrIndex = 0;

else

CurrIndex = -1;

}

template<typename T>

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

{

T\* tmp;

if (CurrIndex >= 0 && CurrIndex<CurrSize)

{

tmp = &(Records[CurrIndex]->Data);

}

else

{

throw (string)"empty table";

}

return tmp;

}

template<typename T>

void Table<T>::SetNext()

{

if (CurrIndex != -1)

CurrIndex++;

else

throw (string)"empty table";

if (IsEnded())

Reset();

}

template <typename T>

Table<T>::Table(const Table<T>& TabCopy)

{

MaxRecords = TabCopy.MaxRecords;

CurrIndex = TabCopy.CurrIndex;

CurrSize = TabCopy.CurrSize;

delete[] Records;

Records = new TabRecord<T>\*[MaxRecords];

if (!IsEmpty())

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

Records[i] = new TabRecord<T>(\*(TabCopy.Records[i]));

}

#pragma once

#include "table.h"

template <typename T>

class UnsortedTable : public Table<T>

{

protected:

void Realloc();

public:

UnsortedTable(int i = 10) : Table(i) {};

UnsortedTable(const UnsortedTable<T>& TabCopy) : Table(TabCopy) {};

~UnsortedTable() { };

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

T\* Search(const string Key);

void Delete(const string Key);

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

};

template <typename T>

void UnsortedTable<T>::Realloc()

{

int NewMaxRecords = (int)(MaxRecords\*1.5);

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

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

tmp[i] = Records[i];

MaxRecords = NewMaxRecords;

delete[] Records;

Records = tmp;

}

template <typename T>

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

{

if (IsFull())

Realloc();

TabRecord<T>\* Row = new TabRecord<T>(Key, Data);

Reset();

while (!IsEnded() && Key != Records[CurrIndex]->Key)

CurrIndex++;

if (IsEmpty())

{

CurrIndex++;

}

if (CurrIndex == CurrSize)

{

Records[CurrIndex] = Row;

CurrSize++;

Reset();

}

else

{

Reset();

string s = "Key: " + Key + " - isn`t unique";

throw s;

}

}

template <typename T>

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

{

Reset();

T\* tmp = nullptr;

if (IsEmpty())

throw (string)"can't search in empty table";

while (!IsEnded() && Key != Records[CurrIndex]->Key)

CurrIndex++;

if (!IsEnded())

tmp = &(Records[CurrIndex]->Data);

else

{

string s = Key + " not found";

throw s;

}

return tmp;

}

template <typename T>

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

{

Reset();

if (IsEmpty())

throw (string)"can't search in empty table";

Search(Key);

if (CurrSize > 1)

Records[CurrIndex] = Records[--CurrSize];

else

CurrSize = 0;

}

template <typename T>

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

{

int i = 0;

while(i < Tab.CurrSize)

{

os << i << ". " << Tab.Records[i]->Key<< " | " << Tab.Records[i]->Data << std::endl;

i++;

}

if (Tab.CurrSize == 0)

os << "empty table";

return os;

}

#include "table.h"

template <typename T>

class SortedTable : public Table<T>

{

protected:

void Realloc();

public:

SortedTable(int i = 10) : Table(i) {};

SortedTable(const SortedTable<T>& TabCopy) : Table(TabCopy) {};

~SortedTable() {};

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

T\* Search(const string Key);

void Delete(const string Key);

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

};

template <typename T>

void SortedTable<T>::Realloc()

{

int NewMaxRecords = (int)(MaxRecords\*1.5);

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

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

tmp[i] = Records[i];

MaxRecords = NewMaxRecords;

delete[] Records;

Records = tmp;

}

template <typename T>

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

{

if (IsFull())

Realloc();

TabRecord<T>\* Row = new TabRecord<T>(Key, Data);

Reset();

/////

while (!IsEnded() && Row->Key >= Records[CurrIndex]->Key)

{

if (Records[CurrIndex]->Key == Row->Key)

{

string s = "Key: " + Key + " - isn`t unique";

throw s;

}

CurrIndex++;

}

if (IsEmpty())

CurrIndex++;

CurrSize++;

for (int i = CurrSize-1; i > CurrIndex; i--)

{

Records[i] = Records[i - 1];

}

Records[CurrIndex] = Row;

Reset();

}

template <typename T>

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

{

Reset();

T\* tmp = nullptr;

if (!IsEmpty())

{

int i = -1, j = CurrSize;

int mid;

while (i < j -1 )

{

mid = (j + i) / 2;

if (Key >= Records[mid]->Key)

{

i = mid;

}

else

{

j = mid;

}

}

if (Key == Records[i]->Key)

{

CurrIndex = i;

tmp = &(Records[CurrIndex]->Data);

}

else

{

string s = Key + " not found";

throw s;

}

}

else

throw (string)"emty table";

return tmp;

}

template <typename T>

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

{

Reset();

if (IsEmpty())

throw (string)"can't search in empty table";

Search(Key);

if (CurrSize > 1)

{

CurrSize--;

for (int i = CurrIndex; i < CurrSize; i++)

{

Records[i] = Records[i + 1];

}

Reset();

}

else

CurrSize = 0;

}

template <typename T>

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

{

int i = 0;

while (i < Tab.CurrSize)

{

os << i << ". " << Tab.Records[i]->Key << " | " << Tab.Records[i]->Data << std::endl;

i++;

}

if (Tab.CurrSize == 0)

os << "Tempty table";

return os;

}

#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(int i = 10);

HashTable(const HashTable<T>& TabCopy);

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

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

T\* Search(const string Key);

void Delete(const string Key);

void SetNext();

T\* GetCurr() const;

void Reset();

bool IsEnded() const { return false; };

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

};

template <typename T>

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

{

flag = new int[i];

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

flag[j] = 0;

}

template <typename T>

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

{

MaxRecords = TabCopy.MaxRecords;

CurrIndex = TabCopy.CurrIndex;

CurrSize = TabCopy.CurrSize;

delete[] Records;

delete[] flag;

Records = new TabRecord<T>\*[MaxRecords];

flag = new int[MaxRecords];

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

flag[j] = TabCopy.flag[j];

if (!IsEmpty())

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

{

if(flag[i] == 1)

Records[i] = new TabRecord<T>(\*(TabCopy.Records[i]));

}

}

template <typename T>

void HashTable<T>::Reset()

{

if (!IsEmpty())

{

CurrIndex = 0;

while (flag[CurrIndex] != 1)

CurrIndex++;

}

else

CurrIndex = -1;

}

template <typename T>

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

{

T\* tmp = nullptr;

if(!IsEmpty())

{

tmp = &(Records[CurrIndex]->Data);

}

else

{

throw (string)"empty table";

}

return tmp;

}

template <typename T>

void HashTable<T>::SetNext()

{

if (!IsEmpty())

{

CurrIndex++;

while (flag[CurrIndex] != 1)

CurrIndex = (CurrIndex+1) % MaxRecords;

}

else

throw (string)"empty table";

}

template <typename T>

void HashTable<T>::Realloc()

{

int OldMaxRecords = MaxRecords;

MaxRecords \*= 1.5;

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

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

{

tmp[i] = Records[i];

}

delete[] Records;

CurrSize = 0;

CurrIndex = -1;

delete[] flag;

flag = new int[MaxRecords];

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

flag[i] = 0;

Records = new TabRecord<T>\*[MaxRecords];

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

{

Insert(tmp[i]->Data, tmp[i]->Key);

}

}

template<typename T>

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

{

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);

int h = rand();

return h % MaxRecords;

}

template <typename T>

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

{

if (IsFull())

Realloc();

TabRecord<T>\* Row = new TabRecord<T>(Key, Data);

Reset();

CurrIndex = HashFunc(Key);

if (flag[CurrIndex] == 0 || flag[CurrIndex] == -1 )

{

Records[CurrIndex] = Row;

CurrSize++;

flag[CurrIndex] = 1;

Reset();

}

else

{

if (Records[CurrIndex]->Key != Key)

{

int ind = CurrIndex;

while (flag[CurrIndex] == 1 && CurrIndex+1 != ind)

CurrIndex = (CurrIndex+1) % MaxRecords;

Records[CurrIndex] = Row;

CurrSize++;

flag[CurrIndex] = 1;

Reset();

}

else

{

Reset();

string s = "Key: " + Key + " - isn`t unique";

throw s;

}

}

}

template <typename T>

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

{

Reset();

T\* tmp = nullptr;

if (IsEmpty())

throw (string)"can't search in empty table";

CurrIndex = HashFunc(Key);

if(Records[CurrIndex]->Key == Key)

tmp = &(Records[CurrIndex]->Data);

else

{

int ind = CurrIndex;

while (flag[CurrIndex] == 1 && CurrIndex+1 != ind)

{

CurrIndex = (CurrIndex + 1) % MaxRecords;

if (Records[CurrIndex]->Key == Key)

{

tmp = &(Records[CurrIndex]->Data);

break;

}

}

if (tmp == nullptr)

{

Reset();

string s = Key + " not found";

throw s;

}

}

return tmp;

}

template <typename T>

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

{

Reset();

if (IsEmpty())

throw (string)"can't search in empty table";

Search(Key);

TabRecord<T> A;

flag[CurrIndex] = -1;

Records[CurrIndex] = new TabRecord<T>(A);

CurrSize--;

Reset();

}

template <typename T>

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

{

int i = 0;

while (i < Tab.MaxRecords)

{

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

{

os << i << ". " << Tab.Records[i]->Key << " | " << Tab.Records[i]->Data << std::endl;

}

i++;

}

if (Tab.CurrSize == 0)

os << "empty table";

return os;

}