Лабораторная работа № 4

Выполнила: студент группы 4309

Канькина Ольга

Основное назначение хеширования — проверка информации. Эта задача важна в огромном количестве случаев: от проверки паролей на сайте до сложных вычислений в блокчейне. Так как хеш — это уникальный код определенного набора данных, по нему можно понять, соответствует ли информация ожидаемой. Поэтому программа может хранить хеши вместо образца данных для сравнения. Это может быть нужно для защиты чувствительных сведений или экономии места.

Хеш-табли́ца — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Существуют два основных варианта хеш-таблиц: с цепочками и открытой адресацией.

Хеш-функция— функция, осуществляющая преобразование массива входных данных произвольной длины в выходную битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом, «ключом» или «сообщением».

Под коллизией понимается ситуация, когда при добавлении разных объектов мы попадаем в одну и ту же ячейку массива. Для разрешения коллизий придумано 2 метода: метод цепочек и метод открытой адресации.

Коллизии существуют для большинства хеш-функций, но для «хороших» хеш-функций частота их возникновения близка к теоретическому минимуму. В некоторых частных случаях, когда множество различных входных данных [конечно](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE), можно задать [инъективную](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%8A%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%8C) хеш-функцию, по определению не имеющую коллизий. Однако для хеш-функций, принимающих вход переменной длины и возвращающих хеш постоянной длины, коллизии обязаны существовать, поскольку хотя бы для одного значения хеш-функции соответствующее ему множество входных данных будет бесконечно — и любые два набора данных из этого множества образуют коллизию.

Метод цепочек является наиболее простым методом разрешения коллизий. В ячейке массива мы будем хранить не элементы, а связанный список данных элементов. Потому как добавление в начало списка обладает асимптотикой , мы не испортим общую асимптотику, и она останется равной.

У данной реализации есть проблема: если списки будут очень сильно вырастать (в качестве крайнего случая можно рассмотреть хеш-функцию, которая возвращает константу для любого объекта), то мы получим асимптотику O(m), где m — число элементов во множестве, если размер массива фиксирован. Для избежания таких неприятностей вводится понятие коэффициент заполнения (он может быть равен, например, 1.5). Если при добавлении элемента оказывается, что доля числа элементов, находящихся в структуре данных по отношению к размеру массива, превосходит коэффициент заполнения, то происходит следующее: выделяется новый массив, размер которого превосходит размер старого массива (например в 2 раза), и структура данных перестраивается на новом массиве.

Метод открытой адресации. В данном методе в ячейках хранятся сами элементы, а в случае коллизии происходит последовательность проб, то есть мы начинаем по некоторому алгоритму перебирать ячейки в надежде найти свободную. Это можно делать разными алгоритмами (линейная / квадратичная последовательности проб, двойное хеширование), каждый из которых обладает своими проблемами (например, возникновение первичных или вторичных кластеров).

**Задание**. Синтаксический анализатор размещает распознанные с помощью лексического анализатора слова в таблицы. Количество таблиц соответствует числу типов слов. В нашем случае нужны три таблицы: таблица чисел (первый тип слова), таблица идентификаторов (второй тип слова), таблица составленных из специальных символов слов (третий тип слова). Таблицы организуются методом хеширования. Включить класс «Поиск слова в таблице методом хеширования» в разрабатываемый транслятор.

Код программы:

MyHashFunction.сs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace nsLexMainForm

{

public class MyHashFunction

{

public int HashFunction(string word) //реализация хэш-функции

{ // принимает слово в качестве параметра и вычисляет хэш-код путем суммирования значений символов

int hashValue = 0;

foreach (char c in word)

{

hashValue += (int)c;

}

return hashValue; //Возвращается полученное значение хэша

}

public void AddWord(Dictionary<int, List<string>> hashTable, string word)

{

int hashValue = HashFunction(word); // вычисляется хэш-код для слова

if (!hashTable.ContainsKey(hashValue)) //проверяется, содержится ли уже такой хэш-код в хэш-таблице

{

hashTable[hashValue] = new List<string>(); //в таблице создается новая пустая коллекция

}

else

{

if (hashTable[hashValue].Contains(word))//проверяется, содержится ли уже переданное слово в этой коллекции

{

Console.WriteLine("Слово уже существует: " + word); // выводится сообщение о том, что слово уже существует, и метод завершается

return;

}

}

hashTable[hashValue].Add(word); //добавляется

}

public int SearchWord(Dictionary<int, List<string>> hashTable, string word)

{

int hashValue = HashFunction(word);

if (hashTable.ContainsKey(hashValue)) // проверяется, содержится ли такой хэш-код в хэш-таблице

{

return 1;

}

else

{

return 0;

}

}

public bool RemoveWord(Dictionary<int, List<string>> hashTable, string word)

{

int hashValue = HashFunction(word);

if (hashTable.ContainsKey(hashValue)) //проверяется наличие в хэш-таблице

{

List<string> words = hashTable[hashValue];

if (words.Contains(word))

{

words.Remove(word); // Если такое слово найдено, оно удаляется

if (words.Count == 0) //Если после удаления других слов коллекция становится пустой, она удаляется из хэш-таблицы

{

hashTable.Remove(hashValue);

}

return true;

}

}

return false;

}

}

}

Form1.cs

using nsLex;

using nsLexMainForm;

using nsSynt;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace CHashTableList

{

public partial class Form1 : Form

{

Dictionary<int, List<string>> hashTableIdentifier = new Dictionary<int, List<string>>();

Dictionary<int, List<string>> hashTableDigital = new Dictionary<int, List<string>>();

Dictionary<int, List<string>> hashTableRezerv = new Dictionary<int, List<string>>();

public MyHashFunction hashFunction = new MyHashFunction();

public CHashTableList htl = new CHashTableList(2);

public Form1()

{

InitializeComponent();

tbFSource.AppendText("100\_abcd\_100\_abc" + "\r\n");

int n = tbFSource.Lines.Length;

}

private void Form1\_Load(object sender, EventArgs e)

{

}

public void TablesToMemo(object sender, System.EventArgs e)

{

List<string> listTable = new List<string>();

listBox1.Items.Clear();

listBox2.Items.Clear();

listBox3.Items.Clear();

htl.TableToStringList(0, listTable);

foreach (var entry in hashTableIdentifier)

{

listBox1.Items.Add(string.Join(", ", entry.Value));

}

listTable.Clear();

htl.TableToStringList(1, listTable);

foreach (var entry in hashTableDigital)

{

listBox2.Items.Add(string.Join(", ", entry.Value));

}

listTable.Clear();

foreach (var entry in hashTableRezerv)

{

listBox3.Items.Add(string.Join(", ", entry.Value));

}

listTable.Clear();

}

private void btnFStart\_Click\_1(object sender, EventArgs e)

{

tbFMessage.Clear();

Analyzer Synt = new Analyzer();

Synt.Lex.strPSource = tbFSource.Lines;

Synt.Lex.strPMessage = tbFMessage.Lines;

Synt.Lex.enumPState = TState.Start;

try

{

Synt.Lex.NextToken();

Synt.S();

throw new Exception("Текст верный");

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Synt.Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Synt.Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

private void btnFRecord\_Click\_1(object sender, EventArgs e)

{

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines;

Lex.strPMessage = tbFMessage.Lines;

int x = tbFSource.TextLength;

int y = tbFSource.Lines.Length;

tbFMessage.Text = "";

try

{

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

string s1 = "", s = "";

switch (Lex.enumPToken)

{

case TToken.lxmIdentifier:

{

hashFunction.AddWord(hashTableIdentifier, Lex.strPLexicalUnit);

s1 = "id " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 0, ref b))

{

TablesToMemo(this, e);

}

break;

}

case TToken.lxmNumber:

{

hashFunction.AddWord(hashTableDigital, Lex.strPLexicalUnit);

s1 = "num " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

TablesToMemo(this, e);

}

break;

}

case (TToken.lxmdt):

{

hashFunction.AddWord(hashTableRezerv, ":");

s1 = "rez " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

TablesToMemo(this, e);

}

break;

}

case (TToken.lxmComma):

{

hashFunction.AddWord(hashTableRezerv, ",");

s1 = "rez " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

TablesToMemo(this, e);

}

break;

}

case (TToken.lxmmul):

{

hashFunction.AddWord(hashTableRezerv, "\_");

s1 = "rez " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

TablesToMemo(this, e);

}

break;

}

}

String m = "(" + s + "" + s1 + ")";

tbFMessage.Text += m;

}

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

private void searchBtn\_Click\_1(object sender, EventArgs e)

{

if (hashFunction.SearchWord(hashTableIdentifier, listBox1.SelectedItem.ToString()) == 1)

{

searchBtn.BackColor = Color.Green;

}

else

{

searchBtn.BackColor = Color.Red;

}

}

private void addBtn\_Click\_1(object sender, EventArgs e)

{

hashFunction.AddWord(hashTableIdentifier, textBox1.Text.ToString());

}

private void deleteBtn\_Click\_1(object sender, EventArgs e)

{

if (hashFunction.RemoveWord(hashTableIdentifier, listBox1.SelectedItem.ToString()))

{

deleteBtn.BackColor = Color.Green;

}

else

{

deleteBtn.BackColor = Color.Red;

}

}

private void changeBtn\_Click\_1(object sender, EventArgs e)

{

if (hashFunction.RemoveWord(hashTableIdentifier, listBox1.SelectedItem.ToString()))

{

hashFunction.AddWord(hashTableIdentifier, textBox1.Text.ToString());

changeBtn.BackColor = Color.Green;

}

else

{

changeBtn.BackColor = Color.Red;

}

}

private void reloadBtn\_Click\_1(object sender, EventArgs e)

{

listBox1.Items.Clear();

listBox2.Items.Clear();

listBox3.Items.Clear();

foreach (var entry in hashTableIdentifier)

{

listBox1.Items.Add(string.Join(", ", entry.Value));

}

foreach (var entry in hashTableDigital)

{

listBox2.Items.Add(string.Join(", ", entry.Value));

}

foreach (var entry in hashTableRezerv)

{

listBox3.Items.Add(string.Join(", ", entry.Value));

}

}

}

}









