Государственное образовательное учреждение

высшего профессионального образования

«Уфимский Государственный Авиационный Технический Университет»

Кафедра «Вычислительной математики и кибернетики»

Отчет

По лабораторной работе

по дисциплине: Методы построения трансляторов

Лабораторная работа № 1.

Организация таблиц идентификаторов

Вариант 3

Проверил

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Выполнил:

студент гр. ПРО-301в

Доронин С.Г.

Уфа 2014

# Задание

Изучить основные методы организации таблиц идентификаторов, получить представление о преимуществах и недостатках, присущих различным методам организации таблиц идентификаторов.

Для выполнения лабораторной работы требуется написать программу, которая получает на входе набор идентификаторов, организует таблицы идентификаторов с помощью заданных методов, позволяет осуществить **многократный** поиск произвольного идентификатора в таблицах и сравнить эффективность методов организации таблиц. Список идентификаторов считать заданным в виде текстового файла. Длина идентификаторов ограничена 32 символами.

# Описание выбранной хэш функции

В качестве хэш-функции используется: остаток от деления суммы всех символов идентификатора на размер таблицы.

**protected** **def** getHash(string: String) **=** string.getBytes.sum.abs **%** MaxTableSize

# Схема организации таблиц идентификаторов

В качестве первой схемы используется: простое рехеширование. В качестве второй – бинарное дерево

# Описание алгоритмов поиска в таблицах идентификаторов

## Простое рехеширование

Для организации поиска по таблице используются две функции, первая – **find()** и **findRec()** – рекурсивная.

Функция **find(idName: String)**, принимает имя искомого идентификатора, запускает счетчик количества проверок и запускает рекурсивную функцию **findRec(idName: String, el: Node, list: Seq[Node])**, передавая первым параметром имя искомого идентификатора, вторым – первый элемент из таблицы идентификаторов (голова), третьим – список идентификаторов без первого элемента (хвост).

/\*\* Поиск элемента в таблице по имени \*/

**override** **def** find(idName: String) **=** {

// Инкремент счетчика кол-ва поисков

findStat.newElement()

// Рекурсивный поиск элемента

findRec(idName, hashTable.head, hashTable.tail)

}

Функция **findRec(idName: String, el: Node, list: Seq[Node])**, принимает параметры: имя искомого идентификатора, текущий сравниваемый идентификатор, список не проверенных идентификаторов.

Функция увеличивает счетчик, количества сравнений, проверяет совпадает ли имя текущего идентификатора, с искомым, если да, то возвращает его. Если не совпадает, то проверяет пустой ли список оставшихся идентификаторов, если пустой, то возвращает None (ничего). Если список не пустой, то функция вызывает сама себя, передавая параметры: имя искомого идентификатора, первый элемент из списка еще не проверенных идентификаторов (голова), оставшийся список не проверенных идентификаторов без первого (хвост).

/\*\* Рекурсионный поиск элемента в таблице по имени \*/

**@**tailrec

**private** **def** findRec(idName: String, el: Node, list: Seq[Node]): Option[Node] **=** {

// Инкремент счетчика кол-ва итераций поиска элемента

findStat.inc()

// Если найден, то возвращаем его

**if** (el **!=** **null** **&&** el.name **==** idName) Some(el)

// Если список пустой, то возвратить None

**else** **if** (list.isEmpty) None

// Иначе запуск этой же функции, со списком без головы

**else** findRec(idName, list.head, list.tail)

}

## Бинарное дерево

Для организации поиска по таблице используются две функции, первая – **find()** и **findRec()** – рекурсивная.

Функция **find(idName: String)**, принимает имя искомого идентификатора, запускает счетчик количества проверок и запускает рекурсивную функцию **findRec(idName: String, node: Option[Node])**, передавая первым параметром имя искомого идентификатора, вторым – корневой элемент бинарного дерева.

/\*\* Поиск элемента в таблице по имени \*/

**override** **def** find(idName: String): Option[Node] **=** {

// Инкремент счетчика кол-ва поисков

findStat.newElement()

// Рекурсивный поиск элемента

findRec(idName, root)

}

Функция **findRec(idName: String, node: Option[Node])**, принимает параметры: имя искомого идентификатора, текущий сравниваемый узел дерева, содержащий идентификатор.

Функция увеличивает счетчик, количества сравнений, проверяет пустой-ли узел текущего элемента, если да, значит искомый идентификатор не обнаружен в текущем узле, возвращает None (ничего).

Если узел не пустой, проверяет совпадает-ли имя идентификатора в узле с искомым, если совпадает возвращает текущий узел. Если имя не совпадает, то функция запускает поиск идентификатора в левом и правом поддеревьях вызывая саму себя. Далее функция возвращает результат поиска в левом поддереве, если он не пуст, иначе результат поиска в правом поддереве.

/\*\* Рекурсивный поиск элемента в таблице по имени \*/

**private** **def** findRec(idName: String, node: Option[Node]): Option[Node] **=** {

// Инкремент счетчика кол-ва итераций поиска элемента

findStat.inc()

// Поиск элемента

**if** (node.isEmpty) {

None // Если текущий узел пустой, то возвращаем None

} **else** **if** (node.get.name **==** idName) node // Если элемент найден, то возвращаем его

**else** {

// Иначе запускаем поиск для левого и прового поддеревьев

**val** left **=** findRec(idName, node.get.left)

**val** right **=** findRec(idName, node.get.right)

// Еслил рез-т поиска в левом поддереве не пустой, то возвращаем его, иначе рез-т правого

**if** (**!**left.isEmpty) left **else** right

}

}

# Текст программы

## Таблица идентификаторов (метод: Простое рехеширование)

*/\*\**

*\* Элемент таблицы*

*\*/*

**class** Node(name: String, hash: Index) **extends** NodeAbstract(name, hash)

*/\*\**

*\* Организация таблиц идентификаторов*

*\* Метод: Простое рехэширование*

*\*/*

**class** RehashTable(MaxTableSize: Index) **extends** IdTableAbstract(MaxTableSize) {

*/\*\* Минимальный индекс таблицы \*/*

**private val** MinIndex = IndexType.Zero

*/\*\* Хранилище элементов таблицы \*/*

**private val** hashTable = **new** ArrayBuffer[Node](MaxTableSize)

*///////////////////////////////////////////*

init()

*///////////////////////////////////////////*

*/\*\* Инициализация \*/*

**override def** init() = clear()

*/\*\* Добавление элемента в таблицу \*/*

**override def** add(idName: String) = {

*// Если в таблице есть хоть одно свободное место*

**if** (addStat.elementsCount < MaxTableSize) {

*// Изменить статистику и добавить элемент*

addStat.newElement()

**val** idHash = getHash(idName)

addRec(idName, idHash)

} **else** {

*// Если нет, то выдать сообщение и возвратить None*

println(**">> Table is full"**) *//TODO*

None

}

}

*/\*\* Рекурсивный поиск свободной ячейки и добавление туда \*/*

@tailrec

**private def** addRec(idName: String, hash: Index): Option[Node] = {

*// Инкремент счетчика кол-ва итераций добавления элемента*

addStat.inc()

*// Элемент таблицы под индексом == hash*

**val** el = hashTable(hash)

**if** (el == **null**) {

*// Если ячейка пуста, то добавляем туда*

**val** node = **new** Node(idName, getHash(idName))

hashTable(hash) = node

Some(node)

} **else if** (el.name == idName) {

*// Если в ячейке уже есть данный Id, то выводим оообщение*

println(**">> Already exists!"**) *//TODO*

Some(el)

} **else** {

*// Иначе запуск этой же функции с другим хэшем*

**val** newHash = rehash(hash)

addRec(idName, newHash)

}

}

*/\*\* Функция рехэширования \*/*

**private def** rehash(hash: Index) = (hash + 1) % MaxTableSize

*/\*\* Поиск элемента в таблице по имени \*/*

**override def** find(idName: String) = {

*// Инкремент счетчика кол-ва поисков*

findStat.newElement()

*// Рекурсивный поиск элемента*

findRec(idName, hashTable.head, hashTable.tail)

}

*/\*\* Рекурсионный поиск элемента в таблице по имени \*/*

@tailrec

**private def** findRec(idName: String, el: Node, list: Seq[Node]): Option[Node] = {

*// Инкремент счетчика кол-ва итераций поиска элемента*

findStat.inc()

*// Если найден, то возвращаем его*

**if** (el != **null** && el.name == idName) Some(el)

*// Если список пустой, то возвратить None*

**else if** (list.isEmpty) None

*// Иначе запуск этой же функции, со списком без головы*

**else** findRec(idName, list.head, list.tail)

}

*/\*\* Очистка таблицы \*/*

**override def** clear() = {

hashTable.clear()

**for** (i <- MinIndex until MaxTableSize) hashTable += **null**

}

*/\*\* Возврат таблицы идентификаторов \*/*

**override def** getIdTable = {

**for** (index <- MinIndex until MaxTableSize **if** hashTable(index) != **null**) **yield s"$**index**: $**{hashTable(index)}**"**

}

}

## Таблица идентификаторов (метод: Бинарное дерево)

*/\*\**

*\* Элемент дерева*

*\*/*

**class** Node(name: String, hash: Index, **var** left: Option[Node], **var** right: Option[Node]) **extends** NodeAbstract(name, hash) {

**def this**(name: String, hash: Index) = **this**(name, hash, None, None)

**override def** toString = **s"hash: $**hash**, name: $**name**, "** +

**s"L: $**{**if** (!left.isEmpty) left.get.name **else "-"**} **"** +

**s"R: $**{**if** (!right.isEmpty) right.get.name **else "-"**} **"**

*/\*\* Вывод поддерева начиная с текущего элемента \*/*

**def** toSubtreeString: String = **s"hash: $**hash**, name: $**name**"** +

**s"\nL: $**{**if** (!left.isEmpty) left.get.toSubtreeString **else "-"**} **"** +

**s"\nR: $**{**if** (!right.isEmpty) right.get.toSubtreeString **else "-"**} **"**

}

*/\*\**

*\* Организация таблиц идентификаторов*

*\* Метод: Бинарное дерево*

*\*/*

**class** BinaryTree(MaxTableSize: Index) **extends** IdTableAbstract(MaxTableSize) {

*/\*\* Корневой элемент дерева \*/*

**private var** root: Option[Node] = None

*///////////////////////////////////////////*

init()

*///////////////////////////////////////////*

*/\*\* Инициализация таблицы идентификаторов \*/*

**override def** init() = clear()

*/\*\* Добавление элемента в таблицу \*/*

**override def** add(idName: String): Option[Node] = {

*// Инкремент счетчика кол-ва добавленных элементов*

addStat.newElement()

*// Добавление элемента в дерево*

**val** hash = getHash(idName)

root = addRec(idName, hash, root)

*// Поиск добавленного элемента и его возврат*

find(idName)

}

*/\*\* Рекурсивное добавление элемента в дерево \*/*

**private def** addRec(idName: String, hash: Index, node: Option[Node]): Option[Node] = {

*// Инкремент счетчика кол-ва итераций добавления элемента*

addStat.inc()

*// Добавление элемента в дерево*

**if** (node.isEmpty) { *// Если узел пустой*

**new** Some(**new** Node(idName, hash)) *//Создаем новый узел, знач-е узла берем из idName*

} **else if** (node.get.name == idName) { *// Если элемент уже есть, не добавляем его*

println(**">> Already exists!"**) *//TODO*

node

} **else if** (hash > node.get.hash) { *// Если hash > текущего узла*

node.get.right = addRec(idName, hash, node.get.right)

node

} **else if** (hash <= node.get.hash) { *// Если hash <= текущего узла*

node.get.left = addRec(idName, hash, node.get.left)

node

} **else** {

assert(assertion = **false**, **">> The condition is not provided"**) *//TODO*

None

}

}

*/\*\* Поиск элемента в таблице по имени \*/*

**override def** find(idName: String): Option[Node] = {

*// Инкремент счетчика кол-ва поисков*

findStat.newElement()

*// Рекурсивный поиск элемента*

findRec(idName, root)

}

*/\*\* Рекурсивный поиск элемента в таблице по имени \*/*

**private def** findRec(idName: String, node: Option[Node]): Option[Node] = {

*// Инкремент счетчика кол-ва итераций поиска элемента*

findStat.inc()

*// Поиск элемента*

**if** (node.isEmpty) {

None *// Если текущий узел пустой, то возвращаем None*

} **else if** (node.get.name == idName) node *// Если элемент найден, то возвращаем его*

**else** {

*// Иначе запускаем поиск для левого и прового поддеревьев*

**val** left = findRec(idName, node.get.left)

**val** right = findRec(idName, node.get.right)

*// Еслил рез-т поиска в левом поддереве не пустой, то возвращаем его, иначе рез-т правого*

**if** (!left.isEmpty) left **else** right

}

}

*/\*\* Очистка таблицы \*/*

**override def** clear() {

root = None

}

*/\*\* Возврат таблицы идентификаторов \*/*

**override def** getIdTable: Seq[String] = {

println(**">> root: "** + root.get.toSubtreeString) *//TODO*

**val** list = ListBuffer[String]()

buildIdTableRec(root, list)

list

}

*/\*\* Рекурсивный возврат таблицы идентификаторов \*/*

**private def** buildIdTableRec(node: Option[Node], list: ListBuffer[String]) {

**if** (!node.isEmpty) {

buildIdTableRec(node.get.left, list)

list += node.get.name

buildIdTableRec(node.get.right, list)

}

}

}

## Обработчики кнопок формы

…

*// Таблицы ид-ов*

**val** MaxTableSize = 500

**val** binaryTree = **new** BinaryTree(MaxTableSize)

**val** rehashTable = **new** RehashTable(MaxTableSize)

*// Обработчики событий формы*

listenTo(generateIdsButton, clearIdsButton)

listenTo(searchButton, resetButton, findAllButton)

listenTo(exitButton)

reactions += {

*// Генерация ид-ов*

**case** ButtonClicked(`generateIdsButton`) =>

**val** ids = generateIds()

idsTextArea.append(ids.mkString(**"\n"**))

*// Загрузка id-ов в таблицу с рехэшированием*

rehashTable.clear()

ids.foreach(rehashTable.add)

*// Загрузка id-ов в бинарное дерево*

binaryTree.clear()

ids.foreach(binaryTree.add)

*// Поиск элемента*

**case** ButtonClicked(`searchButton`) =>

**val** rNode = rehashTable.find(searchTextField.text)

rFoundLabel.text = **if** (rNode.isDefined) **"Id found" else "Id not found"**

**val** bNode = binaryTree.find(searchTextField.text)

bFoundLabel.text = **if** (bNode.isDefined) **"Id found" else "Id not found"**

updateStat()

*// Сброс статистики поиска*

**case** ButtonClicked(`resetButton`) => resetSearchStat()

*// Очистка поле ввода ид-во*

**case** ButtonClicked(`clearIdsButton`) => idsTextArea.text = **""**

*// Выход из программы*

**case** ButtonClicked(`exitButton`) => System.exit(0)

}

*// Статистика поиска*

**val** rFindStat = rehashTable.getStat.get(**"find"**).get

**val** bFindStat = binaryTree.getStat.get(**"find"**).get

*/\*\* Сброс статистики поиска \*/*

**private def** resetSearchStat() {

searchTextField.text = **""**

rFindStat.reset()

bFindStat.reset()

updateStat()

}

*/\*\* Обновить статистику на форме \*/*

**private def** updateStat() {

**val** format = **"%.3f"**

*// Статистика для таблици рехэширвония*

rEqualsLabel.text = rFindStat.currentElementCounter.toString

rAllEqualsLabel.text = rFindStat.allElementsCounter.toString

rAvgEqualsLabel.text = rFindStat.avg().formatted(format)

*// Статистика для бинарного дерева*

bEqualsLabel.text = bFindStat.currentElementCounter.toString

bAllEqualsLabel.text = bFindStat.allElementsCounter.toString

bAvgEqualsLabel.text = bFindStat.avg().formatted(format)

}

*/\*\* Генерация строки с ид-ми \*/*

**private def** generateIds = () => {

**for** (i <- 1 to IdsCount) **yield** generateRandomString(IdSymbolsCount)

}

*/\*\* Генерация рандомной строки из латинских больших и малых букв \*/*

**private def** generateRandomString = (size: Int) => {

*// Кодировка символов*

**val** CharsetType = **"utf8"**

*// Начальный символ, сооветсвует: A*

**val** StartSymbol = 65

*// Конечный символ, сооветсвует: z*

**val** EndSymbol = 122

**val** bytes = **for** (i <- 1 to size) **yield** (Random.nextInt(EndSymbol - StartSymbol + 1) + StartSymbol).toByte

**new** String(bytes.toArray, CharsetType)

}