УДК 519.7

Н.А. Валенда, О.В. Калиниченко, А.В. Саламаха, С.В. Яворский

Харьковский национальный университет радиоэлектроники, Харьков

АНАЛИЗ СПОСОБОВ ОРГАНИЗАЦИИ СЛОВАРЯ СЛОВОФОРМ

Статья посвящена анализу способов организации словарей большого объема, которые являются основой работы систем машинного перевода и обработки естественного языка. Проводится сравнение эффективности алгоритмов доступа к элементам словаря и поиска слов с ошибками для различных способов организации словаря. Рассматриваются структуры данных для реализации словаря на основе нагруженных деревьев.

Ключевые слова: *словарь, асимптотическая оценка, хеш-таблицы, бинарные деревья, нагруженные деревья, вектор переходов, список переходов.*

Введение

Актуальной задачей в области информационных технологий является обработка постоянно возрастающих объемов информации на различных языках. Основной составляющей систем обработки текстов на естественном языке является лингвистический процессор. В зависимости от задач, решаемых системой, структура процессора и сложность составляющих элементов могут варьироваться. Основой работы лингвистического процессора являются словари [1].

Основными требованиями к современным словарям для систем машинного перевода (МП) и анализа естественного языка являются:

– максимальное быстродействие;

– словарный объем должен полностью покрывать тексты по заданной тематике;

– анализ слов, которые не найдены в словаре;

– анализ возможных ошибок во введенном слове.

Необходимо отметить, что для систем МП требование нахождения любого слова текста в словаре является обязательным, поскольку определение только морфологических характеристик по флексии не позволяет найти переводной эквивалент. Требование максимального быстродействия обусловлено тем, что последующие этапы анализа – синтаксический и семантический, являются значительно более сложными и емкими по времени. Морфологический анализ (МА) должен предоставлять полную информацию о лексемах текста за максимально короткое время.

Если рассматривать с этой точки зрения два традиционных подхода к организации словарной информации на базе словаря основ и словаря словоформ, то анализ, основанный на использовании словаря основ, работает в несколько раз медленнее. Это обусловлено большей сложностью алгоритмов МА и необходимостью многократного поиска по словарю при выделении основы из состава изменяемого слова. Таким образом, с точки зрения быстродействия предпочтительнее МА на основе словаря словоформ.

Известно, что в русском языке число различных словоформ значительно больше числа различных основ слов. Если фиксировать объем словаря основ и потребовать, чтобы словарь словоформ содержал все формы слов, которые могут быть образованы на базе словаря основ, то отношение числа словоформ к числу основ слов определяется выражением:

$$K = \sum_{i=1}^{n} M_i P_i ,$$

где n – количество флективных классов слов в русском языке;

M_i – количество различных форм у слов i-го флективного класса;

P_i – вероятность появления i-го флективного класса в словаре. Исследования словарей показывают, что $K \approx 8$ [2].

Поскольку современные системы обработки ЕЯ информации основываются на работе со словарем, то создание эффективных и быстродействующих алгоритмов является важным вопросом. В данной работе исследуются способы организации словаря на базе словаря словоформ. Этот подход выбран за его быстродействие. Необходимо исследовать возможные методы организации словаря и сравнить скорость работы алгоритмов поиска информации для каждого метода. Кроме того, необходимо рассмотреть, существуют ли удобные и эффективные алгоритмы для поиска неправильно написанных слов.

1. Методы организации словаря

С точки зрения теории множеств можно рассматривать словарь как абстрактный тип данных (АТД), для которого реализуются операторы Insert – вставка элемента, Delete – удаление элемента, Mem-

© Н.А. Валенда, О.В. Калиниченко, А.В. Саламаха, С.В. Яворский

ber – поиск заданного элемента. Существует несколько подходов к реализации АТД Словарь – это хеш-таблицы, упорядоченные списки, нагруженные деревья [3, 4]. Рассмотрим сущность каждого из этих методов и эффективность выполнения на них операции поиска слов. Кроме того, рассмотрим возможность организации поиска слов, записанных с ошибками.

Пусть имеется список из M слов, расположенных в алфавитном порядке и хранящихся в последовательных ячейках. Для некоторого входного слова необходимо определить принадлежит ли оно этому множеству слов. Наиболее эффективным методом поиска в упорядоченном списке является бинарный поиск.

При первом сравнении либо обнаруживается совпадение, либо список, где необходимо продолжать поиск, сокращается вдвое. Если исходный список состоит из M элементов, то можно уменьшить его вдвое не более чем $\log_2 M$ раз, при этом получится список, состоящий из одного элемента. К этому моменту либо слово будет найдено, либо выяснится, что его нет в списке. Количество сравнений, производимых алгоритмом, можно выразить асимптотической функцией $\theta(\log_2 M)$.

При реализации этой процедуры требуются некоторые действия для вычисления середины списка. Поэтому на каждом шаге уходит несколько больше времени, чем в случае непосредственного доступа к списку. Из перечисленных операций для АТД Словарь наибольший интерес представляет Member. Скорость выполнения операции Member для упорядоченного списка можно определить следующим соотношением:

$$T(M) = \theta(\log_2 M) * (t_1 + t_2),$$

где t_1 – время вычисления середины списка;

t_2 – время сравнения двух ключей.

Для хранения упорядоченного списка можно использовать массив типизированных записей. Операции добавления и удаления слов будут весьма трудоемким процессом, поскольку в памяти необходимо сдвигать все записи после места вставки или удаления слова. Для удобства работы с массивом необходимо ввести дополнительные поля, что еще больше увеличит размер занимаемого пространства.

Для поиска слов, содержащих ошибки, необходима разработка специальных алгоритмов, направленных на предсказание возможных ошибок. В соответствии с этой гипотезой искомое слово изменяется, после этого поиск производится снова. К недостаткам такого подхода относятся: невозможность предсказания любых ошибок или просто опечаток, слабая надежность метода, дополнительные затраты времени.

Метод упорядоченного списка позволяет быстро находить слово – не более, чем за $\theta(\log_2 M)$ шагов. Если список будет храниться во внешней памяти, то функция задает количество обращений к жесткому диску, а каждая такая операция имеет достаточно большой вес. Кроме того, такая организация словаря не позволяет построить эффективный алгоритм для нахождения слов с ошибками.

Для реализации АТД Словарь часто используется хеширование. Этот метод требует фиксированного времени (в среднем) на выполнение операторов. В самом худшем случае этот метод требует времени, пропорционального размеру множества, так же, как и в случае реализации с помощью неупорядоченных списков. Но при тщательной разработке алгоритмов можно сделать так, что вероятность выполнения операторов за время, большее фиксированного, будет сколь угодно малой [4].

Хеш-таблицу можно рассматривать как обобщение обычного массива. Если достаточно памяти для массива, число элементов которого равно числу всех возможных ключей, то для каждого возможного ключа можно отвести ячейку в массиве и иметь возможность получить доступ к любой записи за время $\theta(1)$. Однако, если реальное количество записей значительно меньше, чем количество возможных ключей, то эффективнее применять хеширование – вычислять позицию записи в массиве, исходя из ключа.

При хешировании элемент с ключом x записывается в позицию номер h(x) в хеш-таблице $T[0 \ldots M-1]$, где $h : U \to \{0,...,M-1\}$ – некоторая хеш-функция. Число h(x) является хеш-значением ключа x. Хеш-значения двух различных ключей могут совпадать. Это означает, что произошла коллизия. По способу обработки коллизий хеширование делится на открытое и закрытое.

При открытом хешировании потенциальное множество разбивается на конечное число классов. Для B классов, пронумерованных от 0 до B–1, строится хеш-функция h такая, что для любого элемента x исходного множества функция h(x) принимает целочисленное значение из интервала 0,…,B-1, которое соответствует классу, которому принадлежит элемент x. Элементы множества, которым соответствует одно и то же хеш-значение, связываются в цепочку-список. Массив, называемый таблицей сегментов и проиндексированный номерами сегментов 0,…,B–1, содержит заголовки для B списков. В позиции номер j хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно j, если таких элементов в множестве нет, в позиции j записан NULL.

Пусть T – хеш-таблица с B позициями, в которую занесено M элементов. Коэффициентом заполнения таблицы называется число α=M/B. Оценка стоимости операций будет происходить в терминах α. Средняя стоимость поиска зависит от того, на-

сколько равномерно хеш-функция распределяет хеш-значения по позициям таблицы. Будем условно предполагать, что каждый элемент может попасть в любую из B позиций таблицы с равной вероятностью – это предположение «равномерного хеширования». Определим среднее время поиска отсутствующего в таблице элемента.

В предположении равномерного хеширования все позиции таблицы для данного ключа равновероятны. Среднее время поиска отсутствующего элемента совпадает со средним временем полного просмотра одного из B списков, то есть пропорционально средней длине списков. Эта средняя длина есть α. Необходимо учесть время $\theta(1)$ на вычисление хеш-значения. Тогда при поиске элемента, отсутствующего в таблице, будет просмотрено в среднем α элементов, а среднее время такого поиска будет равно $\theta(1+\alpha)$. При равномерном хешировании среднее время успешного поиска в хеш-таблице такое же.

При закрытом хешировании в таблице хранятся непосредственно элементы словаря, а не заголовки списков. Каждая ячейка содержит либо элемент динамического множества, либо NULL. Число хранимых элементов не может быть больше размера таблицы: коэффициент заполнения не больше 1. Чтобы добавить новый элемент в таблицу с закрытой адресацией, ячейки которой занумерованы целыми числами от 0 до M–1, просматриваем ее, пока не найдем свободное место. Порядок просмотра таблицы зависит от ключа. К хеш-функции добавляется второй аргумент – номер попытки. Хеш-функция имеет вид:

$$h : U \times \{0,...,M-1\} \to \{0,...,M-1\}.$$

Последовательность испробованных мест для данного ключа x имеет вид:

$$\langle h(x,0), h(x,1), ..., h(x,M-1) \rangle,$$

функция h должна быть такой, чтобы каждое из чисел от 0 до M–1 встретилось в ней ровно один раз.

Если мы попытаемся поместить элемент x в сегмент с номером h(x,0), который уже занят другим элементом, то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов h(x,1), h(x,2) и т.д., в которые можно поместить элемент x. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то таблица заполнена, элемент x вставить нельзя.

При поиске элемента с ключом x в таблице с закрытым хешированием ячейки таблицы просматриваются в том же порядке, что и при добавлении в нее элемента с ключом x. Если при этом наталкиваемся на ячейку, в которой записан NULL, то можно говорить, что искомого элемента в таблице нет (иначе он был бы занесен в эту ячейку).

Такая организация хеш-таблицы не позволяет удалить из нее слово. Если просто записать на его место NULL, то в дальнейшем невозможно будет найти те элементы, в момент добавления которых в таблицу это место было занято, т.к. из-за этого был выбран более далекий элемент в последовательности испробованных мест. Возможное решение состоит в том, чтобы записывать на место удаленного элемента не NULL, а специальное значение DELETED, и при добавлении рассматривать ячейку с записью DELETED как свободную, а при поиске – как занятую, и продолжать поиск. Недостаток этого подхода в том, что время поиска может оказаться большим даже при низком коэффициенте заполнения. Поэтому, если требуется удалять записи из хеш-таблицы, предпочтение обычно отдают хешированию с помощью открытой адресации.

Вычисление времени поиска элемента в хеш-таблице велось в предположении, что время вычисления хеш-функции – $\theta(1)$. На самом деле оно отлично от константы. Это время зависит от выбранного способа хеширования. Чтобы быть действительно «случайной», хеш-функция должна хешировать каждый символ слова. Время работы такой функции будет пропорционально длине слова x. Организация словаря с помощью хеш-таблицы создает трудности при нахождении слова, записанного с ошибкой. В этом случае поиск будет даже более сложным, чем для упорядоченного списка. Сначала необходимо вносить в слово изменения, которые предположительно приведут его к правильному варианту, а потом пытаться найти это измененное слово в хеш-таблице, решая по ходу все коллизии. Необходимо каждый раз заново осуществлять поиск, чтобы получить близкие по словарному составу словоформы. Эффективность такого алгоритма весьма низкая.

Нагруженные деревья – специальная структура для представления множеств, состоящих из символьных строк (рис. 1). Такая структура удобна, когда большинство слов начинается с одной и той же последовательности букв или, другими словами, когда количество различных префиксов значительно меньше общей длины всех слов [3]. В нагруженном дереве каждый путь от корня к листу соответствует одному слову из множества. Для обозначения конца слова вводится специальный маркер – ┣.

Нагруженные деревья обладают следующими свойствами:

- каждая вершина дерева может иметь степень по выходу M, где M – это количество символов в алфавите распознаваемого языка;
- большинство вершин будут иметь степень по выходу меньше M;
- листьями дерева могут быть только верши-

ны, помеченные признаком конца слова – ├;

- каждый путь от корня к листу соответствует одному слову из множества.

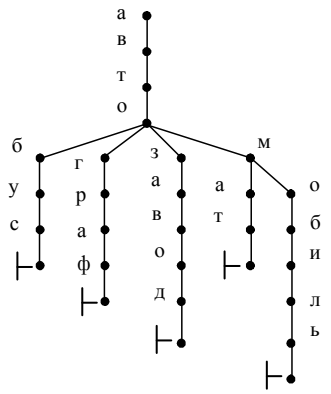

Рис. 1. Представление множества слов с помощью нагруженного дерева

Для представления узлов нагруженного дерева могут быть использованы два метода: вектора переходов и списка переходов. При использовании метода вектора переходов узел дерева представляет собой массив переходов, по одному переходу на каждый символ входного алфавита. Если перехода из данного символа не существует, в ячейку записывается константа NULL. Очередной символ входного слова служит индексом, по которому выбирается элемент вектора, дающий нужный переход. Дерево представляет собой массив размером N×M , где N – количество узлов в нагруженном дереве. Поиск в таком дереве можно будет осуществлять в среднем за время $\log_M N$. Поиск слова, которое находится в словаре и имеет длину n, производится за n+1 шаг. Размер памяти, занимаемой словарем, пропорционален N×M. При большом количестве слов словарь будет иметь очень большой размер. Узлы нижних уровней дерева будут иметь большинство переходов в NULL.

В целях экономии памяти можно использовать метод списка переходов. Каждый узел дерева представляет собой связанный список, состоящий из элементов алфавита к которым возможен переход из данного узла. Элемент списка имеет три поля: поле символа, по которому возможен переход, поле с адресом узла на который идет переход и указатель на следующий элемент связанного списка. Такое дерево легко может быть преобразовано в бинарное дерево. По определению бинарного дерева каждая вершина имеет поле ключа и две ссылки: переход к вершине левого поддерева – LLINK и переход к вершине правого поддерева – RLINK. В бинарном дереве каждый элемент списка представляется отдельной вершиной. LLINK соответствует переходу по совпадению поля ключа и искомого символа. RLINK соответствует переходу к следующему элементу списка, если ключ не совпал. Такая структура очень удобна для расширяющегося набора данных. Поиск в дереве осуществляется по совпадению или не совпадению с символом информационного поля. Пример дерева представлен на рис. 2.

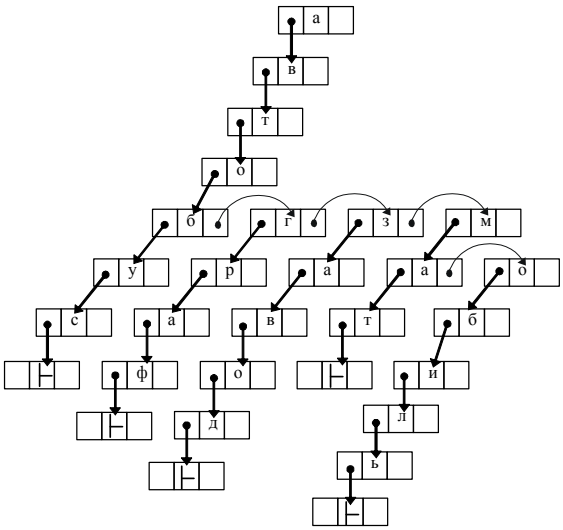

Рис. 2. Бинарное дерево для представления словарной информации

Максимальную высоту бинарного дерева можно определить как

$$H = M \times n_{max},$$

где n_{max} – максимальная длина слова.

Значение H получается достаточно большим, поскольку бинарное дерево, построенное на основе словарной информации, будет не сбалансированным и оценка H – это максимально возможная высота. В худшем случае, чтобы достигнуть листа для любого слова потребуется шагов меньше H. Оценка H для времени нахождения слова была бы справедлива, если бы мы искали слово максимальной длины, состоящее из букв, стоящих на последнем месте в полностью заполненном связном списке для каждого узла дерева. Но таких слов не существует, и реальное количество шагов будет значительно меньше.

Средняя оценка для высоты произвольного не сбалансированного бинарного дерева может быть выражена следующей формулой:

$$H_{ср} = t \times n_{max},$$

где t – среднее количество переходов из одной вершины.

Значение t зависит от количества вершин в дереве. Чем больше заполнено дерево, тем больше будет значение t. На основании данных, полученных при построении экспериментальных словарей, можно утверждать, что эта зависимость носит логарифмический характер

$$t = \theta(\log_M N).$$

Функция зависимости времени прохождения пути в дереве от размера дерева имеет следующий вид

$$T(N) = \theta(\log_M N) \times n_{max}.$$

В общем случае она пропорциональна длине искомого слова.

Если сравнить два рассмотренных подхода, то метод вектора переходов работает быстрее, но применить его для большого объема данных невозможно из-за больших затрат памяти. Метод списка переходов имеет худшие показатели времени поиска, но требует меньше памяти. Оптимальное решение состоит в сочетании этих двух методов. Несколько верхних уровней дерева, которые наиболее плотно заполнены, необходимо представить с помощью векторов переходов. Вершины остальных уровней, которые содержат значительно меньше переходов, можно представить с помощью списков переходов. Время поиска в такой структуре будет пропорционально длине искомого слова.

Нагруженные деревья позволяют эффективно организовывать поиск слов, записанных с ошибками. Поиск в дереве ведется по совпадению текущей буквы слова и вершины в дереве. Если отсутствует соответствующая вершина, то необходимо сделать переходы по имеющимся вершинам и проверить на совпадение со следующей буквой слова. Если один из путей приведет к совпадению всех оставшихся букв слова с графом, то можно считать, что ошибка найдена. Ни один из предыдущих методов не позволял производить такой анализ.

Были рассмотрены три способа представления АДТ Словарь. Оптимальными с точки зрения быстродействия и удобства представления информации, а так же организации алгоритмов поиска, являются нагруженные деревья. Они позволяют построить более эффективные алгоритмы поиска слов с ошибками, чем упорядоченные списки. Любые алгоритмы поиска выполняются для них быстрее, чем для хеш-таблиц.

2. Комбинированные структуры данных для представления бинарного дерева

Исходя из приведенного анализа, словарь реализуется на основе нагруженных деревьев. Первые три уровня дерева задаются с помощью векторов переходов. Эта часть дерева размещается в оперативной памяти. Оставшаяся часть дерева реализуется списками переходов и может размещаться как в оперативной памяти, так и на жестком диске [5].

Согласно методу вектора переходов, адреса или метки тех процедур переходов, на которые должно передаваться управление, хранятся в виде вектора в последовательных ячейках памяти, по одной ячейке для каждого входного символа. Очередной входной символ служит индексом, по которому выбирается элемент вектора, дающий нужный переход. Вектор представляет собой массив из 34 ячеек, которые соответствуют алфавиту русского языка, последняя ячейка отводится для признака конца слова – ├.

Общий вид вектора переходов приведен на рис. 3.

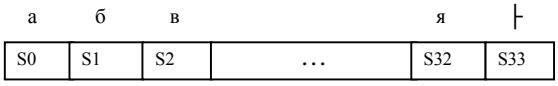

Рис. 3. Вектор переходов

В верхней строке рисунка представлен алфавит допустимых символов. S0, S1,…, S33 – переходы на векторы уровня 2. Пустые ячейки означают вызов процедуры обработки ошибок. Первые три уровня представляют собой иерархическое дерево (рис. 4).

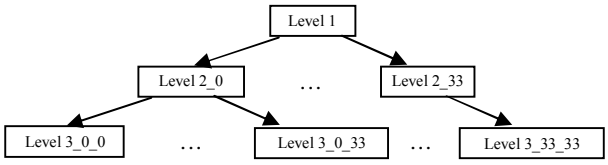

Рис. 4. Иерархия уровней векторов

Каждая ячейка в векторах третьего уровня является переходом на поддерево, реализуемое списком переходов. Эта ячейка выступает в качестве корня дерева.

Согласно методу списка переходов, входные символы делятся на два класса: каждому входному символу первого класса приписывается индивидуальный переход, а все символы второго класса имеют общий переход в состояние ошибки. Для первого класса соответствие между входным символом и адресом процедуры перехода задается в виде списка упорядоченных пар. Общий переход для символов из второго класса запоминается отдельно и называется переходом по неудаче.

При поступлении нового входного символа происходит поиск в списке этого символа и соответствующего ему перехода. Если поиск заканчивается неудачей, делается переход на процедуру, соответствующую неудаче.

Т.к. запись должна иметь фиксированную длину, то необходимо задать количество пар: символ – № перехода. Для каждого перехода максимальное количество пар может быть равно количеству букв в алфавите. В этом случае, большая часть позиций таблицы остается не заполненными, т.е. предложенная структура записи не является эффективной и приемлемой. Для того чтобы записи имели фиксированную длину, необходимо, чтобы узлы дерева имели фиксированное количество переходов. Таким

деревом является бинарное.

Бинарное дерево – это конечное множество узлов, которое или пусто, или состоит из корня и двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня [6, 7]. Любое дерево может быть представлено в виде соответствующего ему бинарного дерева.

Левая связь узла соответствует первой или единственной дуге исходного дерева, а правая связь – переходу по следующей альтернативе из вершины предыдущего уровня.

Если буква анализируемого слова совпадает с символом, то переход по первой ссылке, иначе продолжаем поиск символа и переходим по второму переходу.

При такой структуре записи не важно количество переходов из данного символа, все они будут иметь свою запись, каждый предыдущий будет ссылаться на следующего. Последний из символов альтернативы будет иметь пустой второй переход. Для символов, у которых вообще нет альтернатив, второй переход так же будет пуст.

Словарь, записанный с помощью записей такого типа, легко пополняется, т.к. при внесении нового слова создаются записи для новых альтернатив, при этом не возникает пустых мест в записях. Для построения словаря важным является порядок следования альтернатив. В примере с рис. 1 альтернативы упорядочены слева на право по алфавиту. Для распознавания слова "автомат" понадобиться сделать больше переходов, чем для слова "автобус". Обход дерева ведется сверху вниз и слева на право. При переборе альтернатив одного уровня поиск можно остановить, если дошли до символа, имеющего по алфавиту больший порядковый номер, чем искомый символ.

Существует и другой способ распределения альтернатив – частотный. Чем чаще символ встречается, тем раньше он идет в альтернативе. При таком подходе, если искомого символа нет в альтернативе, то потребуется полный перебор, но если символ есть, то этот метод будет работать быстрее, чем предыдущий. Поскольку в дереве после третьего уровня ветвлений становится меньше, то удобнее использовать альтернативы, упорядоченные по алфавиту.

3. Алгоритмы на деревьях

Словарь должен предусматривать такие основные операции над множеством как Insert, Delete, Member. Должна быть реализована операция поиска неверно заданного слова. В тексте часто случаются опечатки, пользователь может ошибиться при наборе и т.д.

Необходимо создать алгоритм, который позволит производить поиск в словаре возможных вариантов правильного слова, если исходное слово не найдено.

Рассмотрим по порядку соответствующие алгоритмы.

Алгоритм записи слова в словарь состоит из двух частей. Сначала происходит заполнение векторов соответствующих первым трем уровням. На третьем уровне в ячейку записывается указатель на бинарное поддерево. Если слово длиннее 3 символов, то остаток записывается в бинарное поддерево, указатель на которое записан в соответствующую ячейку вектора третьего уровня.

Последним символом слова заносится признак конца – « ├». В ячейке этого символа храниться указатель на комплекс информации, соответствующей данному слову. Это весь комплекс морфологической информации (КМИ), которая может быть приписана данной словоформе. Каждой позиции КМИ соответствует свой указатель на информацию семантического словаря.

Алгоритм поиска слова в словаре аналогичен алгоритму записи. Первые три буквы слова ищутся в векторах, а остаток слова в бинарном поддереве. Если ссылка, по которой должен произойти переход, на данном уровне пуста, то искомого слова в словаре нет. В этом случае необходимо провести поиск для слов с ошибками. Слово найдено, если удалось дойти по графу до признака конца слова. Осуществляется переход по ссылке, считывается приписанная ему морфологическая информация.

Алгоритм удаления слова из словаря существенно отличается от алгоритма записи слова. Нельзя просто обнулять соответствующие позиции в векторах и бинарном дереве, поскольку они могут использоваться для других слов. Следовательно, сначала необходимо просмотреть все записи, относящиеся к данному слову, и отметить среди них те, которые имеют альтернативы. После чего можно будет удалять позиции, относящиеся только к заданному слову.

Рассмотрим пример удаления слова «автограф» из дерева, изображенного на рис. 2. Удалять переходы из заполненных векторов нельзя. Если дерево заполнено в достаточной степени, то эти переходы использует очень много слов. Достаточно удалить переход на бинарное поддерево, если оно состоит только из удаляемого слова.

При удалении слова необходимо убрать лишние переходы из бинарного дерева. Будем рассматривать этот алгоритм в предположении, что слово длиннее 3 символов.

Ищем слово и запоминаем буквы, для которых есть переход по альтернативе. В данном слове такая буква только одна – “г”. Чтобы удалить слово, необходимо удалить все буквы после “г”, поскольку они относятся только к этому слову, изменить пере-

ход для буквы, предшествующей “г” в альтернативе, и удалить ее саму.

Алгоритм удаления слова из бинарного дерева состоит в следующем: ищется последняя буква слова, для которой существует альтернатива. Ее позиция Right, копируется в соответствующую позицию предыдущей буквы в альтернативе. После этого все буквы, начиная с этой и до признака конца слова, можно удалить.

Алгоритм поиска слова с ошибками ориентирован в первую очередь на поиск опечаток, он позволяет эффективно искать ошибки в слове, когда неверно задана или пропущена одна из букв. Данный алгоритм не производит анализ слова на предмет наиболее распространенных ошибок или дублирования букв. Такие алгоритмы уже существуют и реализованы, например, в текстовом редакторе Word.

В качестве примера рассмотрим дерево на рис. 2.

Допустим вместо слова «автозавод», на вход подано ошибочное слова «автохавод». В дереве нельзя найти пятую букву, поскольку она ошибочна. Выделяем оставшуюся часть слова после нераспознанной буквы и пытаемся найти, начиная с первой буквы альтернативы, в которой не оказалось “х”. Сначала проверяем “б”, потом “г”, потом “з”. Для буквы “з” остаток слова может быть распознан. Значит можно распознать это слово как «автозавод». Если в альтернативе найдена буква, с которой начинается остаток слова, то сравнение начинается с нее, а не с буквы, в которую идет переход. При совпадении остатка слова, распознается слово с пропущенной буквой.

В результате работы данного алгоритма формируется множество возможных вариантов слова. Все они передаются ЛА, который формирует на их основании лексемы, соответствующие слову.

Выводы

Произведено сравнение различных способов организации словарей большого объема. Выбран способ организации словаря на основе нагруженного дерева. Рассмотрены структуры, которые используются для физической организации словаря. Приведены алгоритмы пополнения и удаления элементов словаря. Разработан алгоритм поиска слов введенных с ошибками, который позволяет находить ошибки в любой части слова, а не только в окончаниях. Разработанная структура словаря позволяет производить быстрый поиск слов и обрабатывать слова с опечатками.

Список литературы

1. Хайрова Н.Ф. Машинный перевод / Н.Ф. Хайрова, И.В. Замаруева. – Х.: ОКО, 1998. – 80 с.

2. Белоногов Г.Г. Языковые средства автоматизированных информационных систем / Г.Г. Белоногов, Б.А. Кузнецов. – М.: Наука. Главная редакция физико-математической литературы, 1983. – 288 с.

3. Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Издательский дом «Вильямс», 2001. – 384 с.

4. Кормен Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М.: МЦНМО, 2001. – 960 с.

5. Льюис Ф. Теоретические основы проектирования компиляторов / Ф. Льюис, Д. Розенкранц, Р. Стирнз. – М.: Наука, 1983. – 360 с.

6. Седжвик Р. Фундаментальные алгоритмы на С++. Анализ. Структуры данных. Сортировка. Поиск / Р. Седжвик. – К.: «ДиаСофт», 2001. – 688 с.

7. Кнут Д. Искусство программирования, Т 1. Основные алгоритмы / Д. Кнут. – М.: Издательский дом «Вильямс», 2000. – 720 с.

Поступила в редколлегию 30.03.2010

Рецензент: д-р техн. наук, проф. И.В. Гребенник, Харьковский национальный университет радиоэлектроники, Харьков.

АНАЛІЗ СПОСОБІВ ОРГАНІЗАЦІЇ СЛОВНИКА СЛОВОФОРМ

Н.А. Валенда, О.В. Калиниченко, А.В. Саламаха, С.В. Яворський

Стаття присвячена аналізу способів організації словників великого об'єму, які є основою роботи систем машинного перекладу і обробки природної мови. Проводиться порівняння ефективності алгоритмів доступу до елементів словника і пошуку слів з помилками для різних способів організації словника. Розглядаються структури даних для реалізації словника на основі навантажених дерев.

Ключові слова: *словник, асимптотична оцінка, хеш-таблиці, бінарні дерева, навантажені дерева, вектор переходів, список переходів.*

ANALYSIS OF WAYS TO ORGANIZE A DICTIONARY

N.A. Valenda, O.V. Kalynychenko, A.V. Salamah, S.V. Jaworski

The article is devoted the analysis of methods of organization of dictionaries of large volume, which are basis of work of the machine translation and treatment of human language systems. A comparison of efficiency of algorithms of access is made to the elements of dictionary and search of words with errors for the different ways of organization of dictionary. The structures of data are examined for realization of dictionary on the basis of the loaded trees.

Keywords: *dictionary, asymptotic estimation, hash-tables, binary trees, loaded trees, vector of transitions, list of transitions.*