

University of Montana

## ScholarWorks at University of Montana

---

Graduate Student Theses, Dissertations, &  
Professional Papers

Graduate School

---

1987

### Algorithm for character recognition based on the trie structure

Mohammad N. Paryavi  
*The University of Montana*

Follow this and additional works at: <https://scholarworks.umt.edu/etd>

**Let us know how access to this document benefits you.**

---

#### Recommended Citation

Paryavi, Mohammad N., "Algorithm for character recognition based on the trie structure" (1987). *Graduate Student Theses, Dissertations, & Professional Papers*. 5091.  
<https://scholarworks.umt.edu/etd/5091>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact [scholarworks@mso.umt.edu](mailto:scholarworks@mso.umt.edu).

COPYRIGHT ACT OF 1976

THIS IS AN UNPUBLISHED MANUSCRIPT IN WHICH COPYRIGHT SUBSISTS. ANY FURTHER REPRINTING OF ITS CONTENTS MUST BE APPROVED BY THE AUTHOR.

MANSFIELD LIBRARY

UNIVERSITY OF MONTANA

DATE: 1987

AN ALGORITHM FOR CHARACTER RECOGNITION  
BASED ON THE TRIE STRUCTURE

By

Mohammad N. Paryavi

B. A., University of Washington, 1983

Presented in partial fulfillment of the requirements

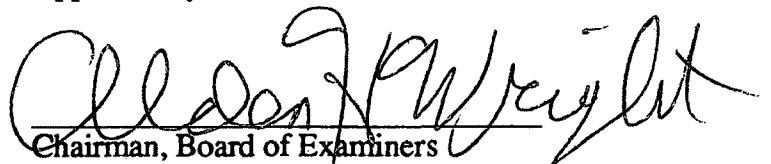
for the degree of

Master of Science


University of Montana

1987

Approved by

  
Chairman, Board of Examiners

  
Dean, Graduate School

  
Date

UMI Number: EP40555

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.

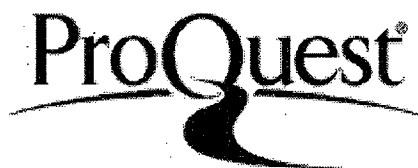


UMI EP40555

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

Paryavi, Mohammad N., M.S., Mar. 1987

Computer Science

An Algorithm for Character Recognition Based on The Trie Structure

Director: Dr. Alden H. Wright



Character recognition is the process of attempting to complete a partially typed string of characters by comparing it with the list of strings stored within the system. Character recognition can be used in a user-friendly interface for a computer system.

In this thesis an algorithm for character recognition is developed based on the trie structure. A trie is an  $m$ -ary tree composed of two types of nodes; branch nodes that contain link fields associated with the elements of an alphabet, and information nodes that are the leaf nodes. For reasons of manipulation and implementation simplicity a binary tree representation of the trie structure was chosen. In this representation a trie branch node is represented by a list of binary tree nodes. This allows us to simulate a branch node with variable number of link fields.

An algorithm for the character recognition operation on the trie is presented first in a general form and later in detail. Also, an algorithm is developed for the insertion operation into a trie structure. These algorithms are analyzed for run time and storage space efficiency. Finally, an alternative representation and an alternative algorithm are discussed.

## Table of Contents

Abstract .....	ii
Table of Contents .....	iii
List of Figures .....	v
Acknowledgements .....	vi
1. Introduction .....	1
1.1. Background .....	1
1.2. Statement of The Problem .....	4
1.3. The Proposed Research .....	4
2. Related Work and Literature .....	6
3. Algorithm Overview .....	8
3.1. The Data Structure .....	8
3.1.1. The General Tree .....	8
3.1.2. The Trie Structure .....	9
3.2. The General Algorithm .....	15
3.2.1. The Recognition Algorithm .....	16
3.2.2. The Insertion Algorithm .....	18
4. The Algorithm in Detail .....	21
4.1. The Representation .....	21
4.2. Notations, Symbols, and Conventions .....	23
4.3. Description of Variables .....	25
4.4. The Algorithm .....	26
4.4.1. The Recognition Algorithm .....	26
4.4.1.1. The Process_Recog Algorithm .....	27
4.4.1.2. The Process_Other Algorithm .....	29

4.4.1.3. The Process_Rubout Algorithm .....	32
4.4.1.4. The Process_Newline Algorithm .....	33
4.4.1.5. The Process_Listop Algorithm .....	34
4.4.1.6. The Traverse_Trie Algorithm .....	35
4.4.2. The Insertion Algorithm .....	36
4.4.2.1. The Add-To-Trie Alg. ....	36
4.4.2.2. The Process-Info-Node Alg. ....	39
4.4.2.3. The Insert-To-Sibling-List Alg. ....	40
4.4.2.4. The Insert-Short-Strings Alg. ....	42
4.4.2.5. The Insert-Long-Strings Alg. ....	43
5. Analysis of the Algorithm .....	47
5.1. General Discussion .....	47
5.2. Time Efficiency of the Alg. ....	47
5.2.1. Analysis of the Recognition Alg. ....	48
5.2.2. Analysis of the Insertion Alg. ....	51
5.3. Space Efficiency of the Algorithm .....	52
5.4. Extensions to the Trie Structure .....	54
5.5. An Alternative Tree Representation .....	55
5.6. An Alternative Algorithm .....	56
6. Summary and Conclusions .....	59
6.1. Summary .....	59
6.2. Conclusions .....	62
BIBLIOGRAPHY .....	64

## List of Illustrations

Figure 3.1 .....	9
Figure 3.2 .....	11
Figure 3.3 .....	12
Figure 3.4 .....	14
Figure 4.1 .....	22
Figure 5.1 .....	50
Figure 5.2 .....	56



## Acknowledgments

This research paper is solely dedicated to Dr. Alden Wright without whose inspirational and technical support it would not have been possible. His expertise was the technical backbone of this paper and his personality the motivating force behind it.

I would also like to thank Professors William Ballard and Spencer Manlove for their invaluable guidance.

My colleagues and fellow FIRESYS team members, Greg Hume, Bruce McTavish, and Jim Mitchell, at the University of Montana - Thanks to all of you.

Special thanks are also due to my good friend Babak Shahpar for being so helpful in criticizing my work to make it better.

Last but not least, my deepest gratitude and love to my wife Somieh for her constant patients and understanding. Without her, none of this would have been possible.

Funding for the FIRESYS project was provided in part through a grant from the Intermountain Fire Sciences Laboratory, in Missoula, Montana.

## Chapter One

### Introduction

#### 1.1. Background

Computers are magnificent creations of man. They are capable of storing large amounts of information and performing otherwise complex and boring tasks with tremendous speed and accuracy. As more and more complex and exotic machines and the programs that drive them are developed, it becomes harder for humans to interact with them without adequate interfaces.<sup>1</sup> For this reason, simple and user-friendly interfaces are becoming essential parts of computer systems.<sup>2</sup> So much so, that a great amount of effort in today's computer science research is spent on user-friendly interfaces. In this area of research, new tools and techniques are sought to diminish the gap between man and machine by simplifying the interface to computer systems.

An *information system*<sup>3</sup> is an example of a computer system that requires a highly user-friendly interface. The reason is that most users of such systems are unfamiliar with the technical aspects of computer systems. Typically, these users are managers, staff members, and customers of an enterprise.

Recently (1985-86), the researcher had a chance to work with such an information system. The research project was in conjunction with the Intermountain Fire Sciences Laboratory, in Missoula, Montana, and involved the building of a fire effects information system using artificial intelligence techniques. The building of this information system

---

<sup>1</sup>These more powerful computers provide more power for developing better user interfaces, and that gives rise to the need for new ideas and techniques in this area.

<sup>2</sup>A computer system usually consists of the hardware and the software that drives the hardware.

<sup>3</sup>The term *information system* is often used synonymously with the term *database system*. However, an information system differs from a database system, in that, it is capable of storing not only the data, but also some of the semantics of the data as well.

was perceived by the Fire Lab as one of the initial steps towards the ultimate goal of constructing a fire effects expert system.

The completed fire effects information system consists of four major parts. The first two parts are the database, where the actual fire effects information is stored, and the interface to the database. The third part is the database builder, which is an interactive system for manipulating the information in the database. The last component of the information system is the query system. The query system is responsible for providing novice users easy access to the information in the database.

One of the most important requirements of the information system, particularly of the query component, was user-friendliness. In order to access information on a specific species of plant or animal, the user most often has to enter the name of the species in question.<sup>1</sup> This name is usually long, very difficult to remember, and next to impossible to spell correctly. For this reason a mechanism was needed to simplify the selection of a species name.

The researcher proposed the utilization of a technique known as *character recognition* to solve the aforementioned problem. Character recognition is the process of attempting to complete a partially typed string of characters by comparing it with the list of strings stored within the system. This is an extremely useful feature, in that it saves the user time and effort by providing the ability to enter a very long string by just supplying the initial few characters of that string. The interface system will perform a search to find a unique string that begins with the characters that the user entered - the prefix. If a unique string is not located, the user will be able to view the strings that are possible with the given prefix and be able to continue the search from this point by supplementing the prefix with one or more additional characters.

---

<sup>1</sup>Currently a menu system allows the user to choose from a list of available species names. This process could become time consuming and bothersome if the list were to become very large.

Perhaps an example would help in understanding the process of character recognition. Assume that the following strings are stored in some data structure in the computer's memory: ALLOCATE, BEGIN, BOOK, BY, BYE, CHECK, and CHAULK. Now suppose the user enters an 'A' followed by the escape character (to signal a recognition request) at the prompt:

→ A<esc>

The computer then searches its memory for a unique word that starts with the letter 'A', and then completes the string for the user:

→ ALLOCATE

Now suppose the user enters a 'B' followed by the escape character:

→ B<esc>

The computer attempts to complete the string for the user, but it finds that there are more than one string that begin with the letter 'B'. Therefore, it simply sounds the bell on the user's terminal and awaits further input:

→ B

At this point the user may decide he/she wants to see what are the possible strings that start with a B by supplying a special character to signal his/her request (in our case a '?'):

→ B?

The computer detects the request and lists the following string on the users terminal and then waits for further input:

BEGIN

BOOK

BY

BYE

→ B

The user may now supply an 'O' and signal for recognition:

→ BO<esc>

And the computer would respond with the completed word:

→ BOOK

## 1.2. Statement of The Problem

Initially the goal of this research was to study some of the existing algorithms for character recognition by analyzing, comparing, and contrasting them. Although the possibilities of introducing a new algorithm were to be considered, the actual development of such an algorithm was not predicted.

That initial goal, however, was quickly changed in light of the fact that the researcher was not able to locate any previous formalizations of algorithms for character recognition. Thus, the initiative was made to work towards the development of an algorithm.

## 1.3. The Proposed Research

The process of the development of a formal algorithm consists of a number of steps. First, a problem gives rise to the basic idea for an algorithm. Next, the development begins, requiring a tremendous amount of thought. During this phase, appropriate data structures need to be considered, different parts of the algorithm have to be characterized, and attempts must be made to discover special cases and conditions. After the completion of, or in conjunction with the development of the algorithm, an implementation environment (language and computer) need to be chosen. Implementation allows the researcher to verify the correctness of the algorithm and gives him/her a chance to "fine-tune" the algorithm.

The formalization of the algorithm is the next step. The algorithm needs to be described in some formal notation for clarity and easy understanding of others. As the last step, the algorithm has to be analyzed mathematically and evaluated to assess efficiency and complexity. This evaluation is necessary if the algorithm is to be used as part of a computer system. Before using an algorithm, system designers usually want to know just how efficient it is, both in terms of memory usage and in terms of speed. The complexity of the implementation is also a fair consideration. An algorithm that is overly complex to implement may not be well suited for some applications.

The thesis of this paper is to develop a character recognition algorithm using the above steps. In addition, the researcher will investigate and present further research ideas such as alternate data structures, implementation strategies and languages, and possibly alternative algorithms.

## Chapter Two

### Related Work and Literature

As mentioned in chapter one, the researcher was unable to locate any previous formalization of character recognition algorithms. Most computer systems that have implemented the idea of character recognition have done so in an ad-hoc manner, i.e., the designers of such systems never described their algorithms in any formal way. Also, these implementations are application specific and not general algorithms. Two examples are the TOPS-20 operating system monitor running on DEC-20 computers and the UNIX Shell.<sup>1</sup>

Unfortunately the code for the TOPS-20 monitor was restricted and unavailable at the time of the research; thus investigation of this implementation was impossible. However, the code for the UNIX Shell was available. The UNIX Shell implementation of character recognition is indeed very application specific. The algorithm does not call for a specific data structure; rather, it is implemented on top of the existing hierarchical UNIX file structure. The recognition is performed on file names at different levels of the file structure. Each time a string of characters is to be matched, as a prefix, with a partial file name, the directories are searched in a given sequence and a character-by-character match is performed on each file name in each directory.

On the surface, this implementation seems to be very inefficient because of the brute-force matching of each file name in each possible directory in the given path, but since it blends very nicely with the structure of the file system and the fact that the number of files in each directory is fairly small, the implementation actually is very fast. However, the important observation to be made is that if the same technique were to be used with a

---

<sup>1</sup>See sh.file.c, code for Cshell by Ken Greer, version 1.4, Dec. 1981, UC Berkley, California.

flat file structured operating system in which the number of files in a single directory is very large, it would not be nearly as efficient in terms of execution time.

A general algorithm for character recognition would provide a fast execution speed that would not vary greatly from applications with a few strings to those with a large number of strings. What will be presented in this paper is such an algorithm (the evaluation of the algorithm will be discussed in later chapters).

The main sources of inspiration for this algorithm were reference books on data structures and algorithms. These include [Tremblay & Sorenson 1984], [Baron & Shapiro 1980], [Knuth 1973], and [Horowitz & Sahni 1983]. In all of these books, strings and data structures and algorithms dealing with strings are presented in a rigorous manner. In particular [Horowitz & Sahni 1983] discuss the idea of *Trie Indexing*, whose main data structure is the *Trie* (to be discussed in detail in later chapters). The researcher was inspired mainly by this discussion and thus chose to develop the character recognition algorithm upon the trie data structure.



## Chapter Three

### Algorithm Overview

#### 3.1. The Data Structure

The main data structure used in the character recognition algorithm is the *trie*. This data structure is based on the notion of a *tree*. We start by a discussion about trees in general and then introduce a special type of tree called the *trie*.

##### 3.1.1. The General Tree

A *tree* [Knuth 1973] is a hierarchical structure made up of cells called *nodes*. Every node of a tree is the parent of zero or more *child* nodes which are connected to the parent. In Figure 3.1, nodes  $\beta$ ,  $\delta$ , and  $\phi$  are children of the root node  $\alpha$ .

Formally, a tree  $T$  is a triple  $(N, r, \tau)$  where

- a)  $N$  is a finite nonempty set of nodes called the *node set* of  $T$ ,
- b)  $r \in N$  is a distinguished node called the root of  $T$ , and
- c)  $\tau = \{ T_1, T_2, \dots, T_n \}$  is a collection of trees whose node sets partition the set  $N - \{ r \}$ . The trees  $T_1, T_2, \dots, T_n$  are called *subtrees* of the root.

The children of one parent node form a set of nodes called *siblings*. In Figure 3.1,  $\beta$ ,  $\delta$ , and  $\phi$  are siblings and  $\gamma$ ,  $\Gamma$ , and  $\Lambda$  are siblings. The level of a child is equal to the level of the parent plus one. The number of subtrees of a node is called the *degree* of that node. A *leaf node* is one that has a degree of zero and a node that has a degree greater than zero is called a *branch node*.

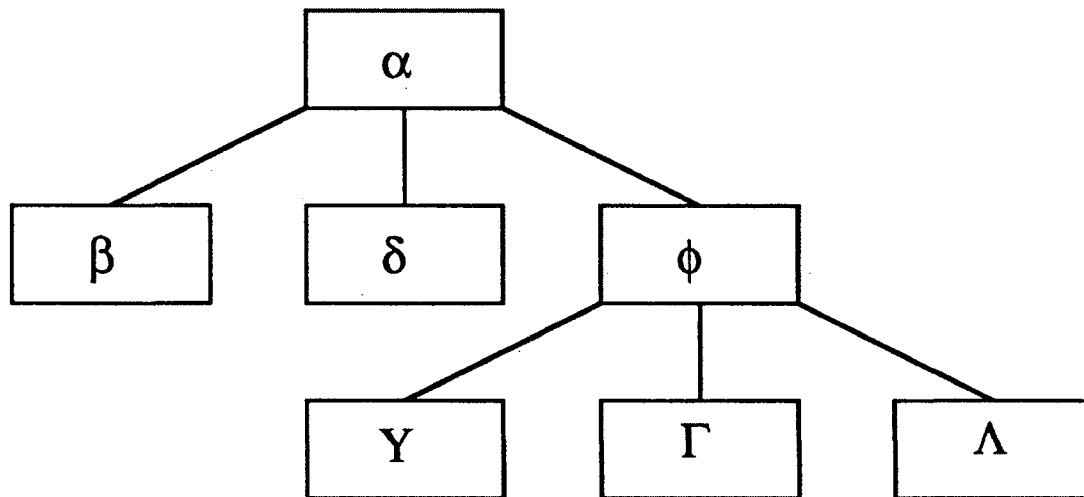


Figure 3.1  
A tree with seven nodes

---

The above description is that of a general tree or an  $m$ -ary tree where  $m$  is the maximum of the degrees of all the nodes. In the following section, we will present a standard definition of the *trie* structure and then explain how we need to modify it for the purposes of the character recognition algorithm.

### 3.1.2. The Trie Structure

A *trie* structure [Horowitz & Sahni 1983] is an  $m$ -ary tree in which each branch node consists of  $m$  components ( $m \geq 2$ ). Typically, these components correspond to letters and digits. The trie structure is an index structure, often used in information organization and retrieval, that is particularly useful when key values (the values to be searched for) are strings of varying size. During a search in a trie, branching at each node of level  $i$  depends on the  $i$ th character of a key.

A formal definition would describe a trie  $T$  as a 5-tuple  $(\Sigma, I, B, r, \tau)$ , where

- a)  $\Sigma$  is an alphabet<sup>1</sup>, and  $|\Sigma| > 0$ ,
- b)  $I$  is a finite set of information nodes (nodes that do not have any children),
- c)  $B$  is a finite set of branch nodes (nodes that have branch and/or information nodes as their children),  $b_1, \dots, b_n$ ,
- d) If  $(B \cup I) \neq \emptyset$ , there exists a distinguished node  $r \in (B \cup I)$  called the root of  $T$ ,
- e) If  $r \in B$ ,  $\tau = \{ T_\sigma : \sigma \in \Sigma \}$  is a collection of tries whose node sets  $B_\sigma, I_\sigma$  partition  $B - \{r\}, I$ . Each  $T_\sigma$  is called a *subtrie* of the root, and
- f) If  $r \in I$ , then  $B = \emptyset$ ,  $\tau = \emptyset$ , and  $|I| = 1$ .

From the above definition it can be inferred that a trie may be empty and that a branch node is allowed to have no children. If a subtrie contains only a single string, then it will be replaced by an information node.

Figure 3.2 illustrates a trie structure. As mentioned above, there are two types of nodes in a trie: *branch* nodes which contain elements of  $\Sigma$  called *link fields*; and *information* nodes which are the leaf nodes of the trie. In Figure 3.2,  $B = \{\alpha, \beta, \gamma, \delta, \sigma, \lambda\}$  and  $I = \{a, b, c, d, e, f, g, h, i, j, k, l\}$ . The root node  $r$  of the trie is  $\alpha$ .

Each branch node consists of 27 link fields: 26 fields associated with the letters of the English alphabet and one field designated for the '\*' which is the terminating character of a string (to be explained later). Thus the alphabet is  $\Sigma = \{\text{English alphabet}\} \cup \{*\}$ . Each information node is or has a *string*. A string as usual, is a finite sequence of elements of  $\Sigma$ , i.e. a member of  $\Sigma^*$  (in standard notation).

At level  $i$  of the trie, strings are partitioned into 27 disjoint classes depending on their  $i^{\text{th}}$  character. Thus, at level  $i$ , the  $j^{\text{th}}$  link of a branch node points to a subtrie containing all the strings whose  $i^{\text{th}}$  character is equal to the character represented by the

---

<sup>1</sup>An alphabet is a finite set of symbols. In our case, the set of symbols is the set of allowable characters.

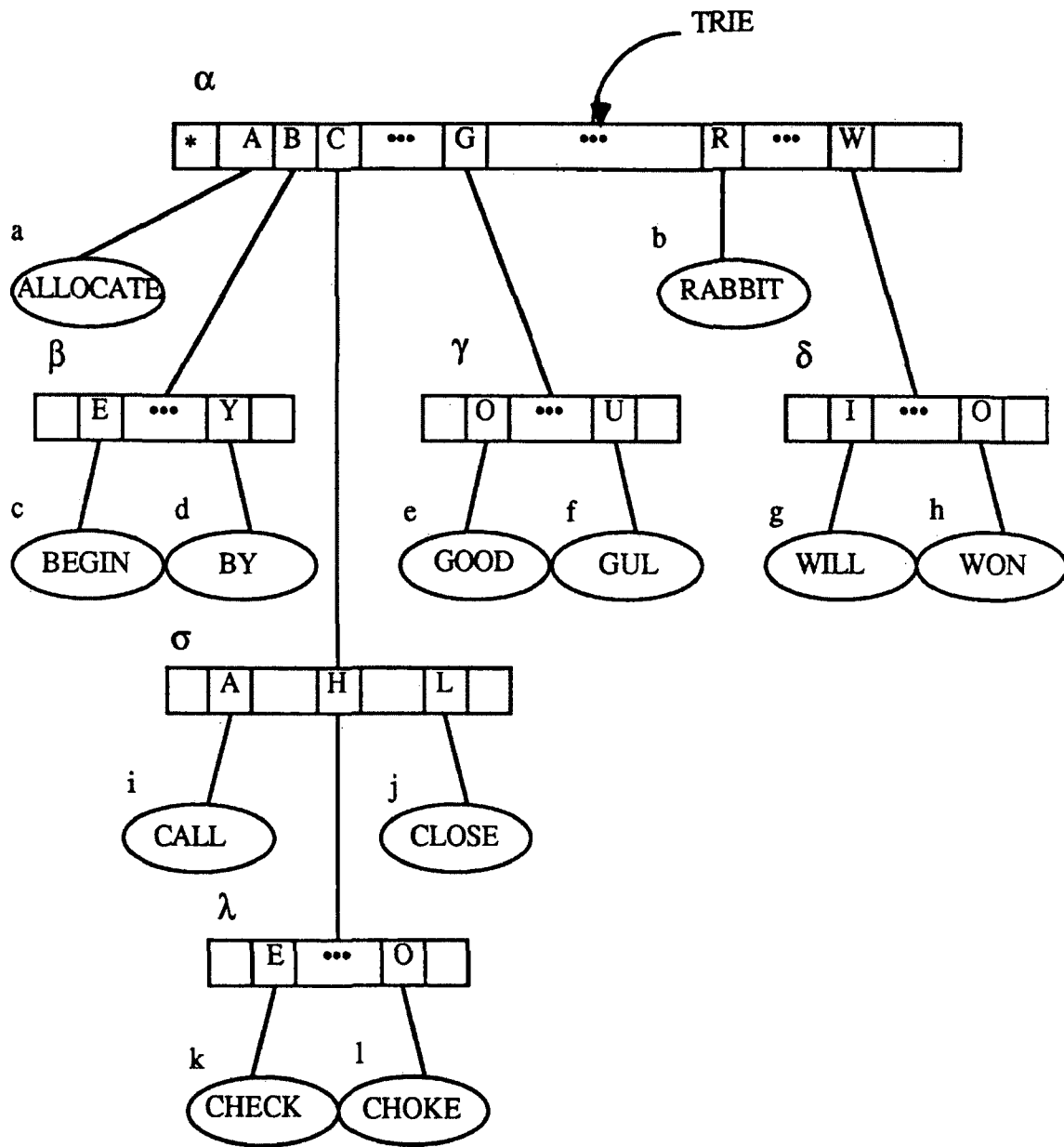


Figure 3.2  
A trie with four levels

$j^{th}$  link of that branch node. For instance, in Figure 3.2 the link representing the character G in the root node points to the subtrie with root  $\gamma$  that contains the two strings GOOD and GULL. G is the *first* character of these two strings, thus they are placed in the subtrie  $T_G$  beneath the root node. The root node of  $T_G$ , namely  $\gamma$ , has links to the two subtries  $T_O$  (information node labeled e) and  $T_U$  (information node labeled f). Furthermore,  $\{ T_k : k \in (\Sigma - \{O, U\}) \} = \emptyset$ .

The terminating character '\*' is necessary to resolve situations in which one string ends on a character that was part of the common prefix with another string. To see this more clearly, we present the example in Figure 3.3. The two strings "DO" and "DOES" have the same first two characters and "DO" terminates on the second common character, namely 'O'.

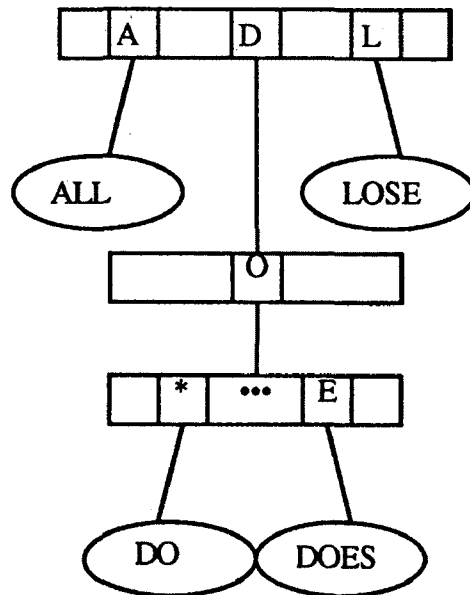


Figure 3.3  
A trie with a string that requires a terminating character

---

Normally, a trie is searched in the following way: a string to be searched for is supplied (the keyword), the keyword is broken up into its constituent characters and the search is performed following the branching patterns determined by these characters. The important observation here is that the entire keyword is supplied before the search can begin.

The trie structure that we have assumed in the character recognition algorithm is different from the above definition in two ways. First, we allow branch nodes to vary in size. That is, it is not required that every branch have a link field corresponding to every possible (allowable) character. Instead, the node will have one link field associated with each subtrie beneath it. This modification to the simple trie results in a more practical implementation, given a limited amount of memory and a large number of strings to be stored.

The second modification is that, in our version of the trie, a complete string is not stored in the corresponding information node unless there is no other string that shares prefix characters with it. This is possible considering that the letters in a string can be collected along the path starting from the root of the trie, and the part that cannot be collected will be stored in the final information node. Figure 3.4 shows the trie of Figure 3.2 with the two modifications that we have discussed.

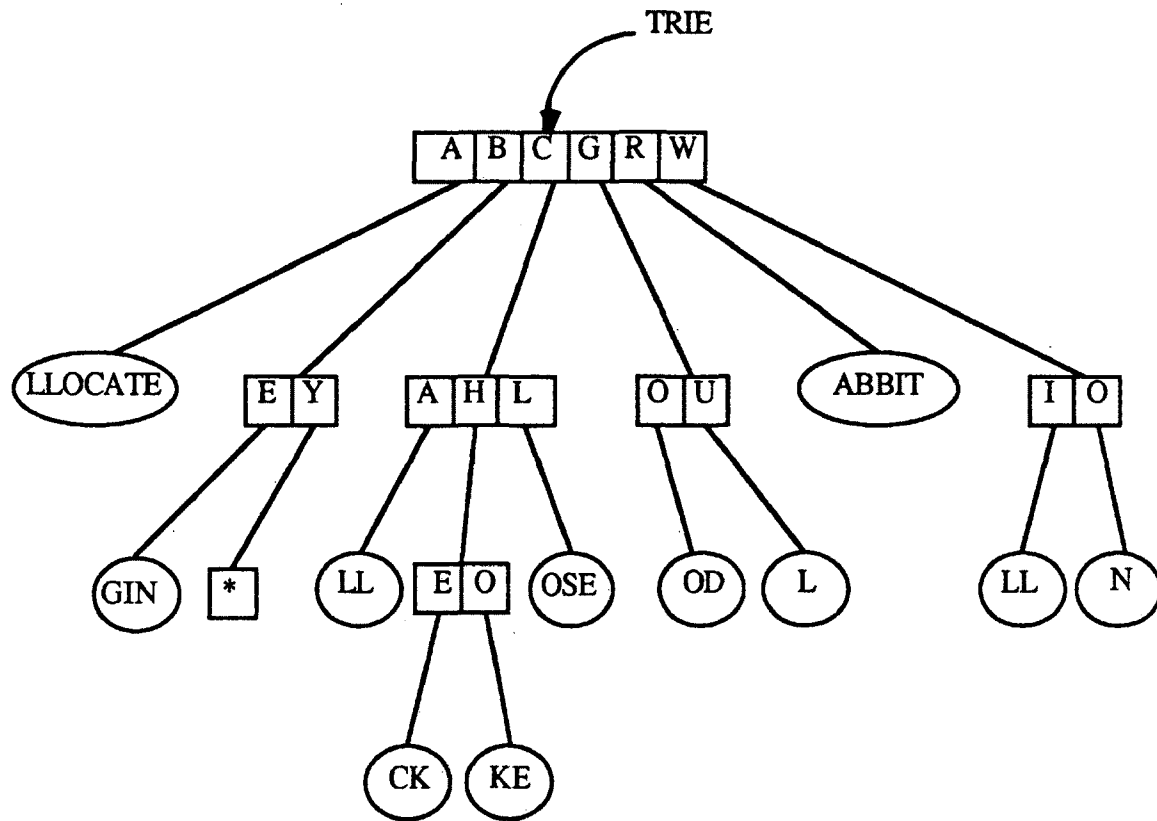


Figure 3.4  
A modified trie with four levels

---

Of course, our modifications to the trie introduce more complications. However, in the opinion of the researcher, the additional complexity is small enough to make the storage conservation well worth the effort. This feature would be extremely helpful in a situation where there are a large number of very long strings that have many characters in common. The reason is that the final string in the information node to be matched is much shorter than the entire string. Thus, the storage space needed for the remaining characters and the time needed to check for a match are greatly reduced.

### 3.2. The General Algorithm

The main data structure for the character recognition algorithm was chosen to be the trie structure. The primary reason behind this decision was the easy and natural adaptability of the trie to the problem. If one were given a list of strings and asked to perform character recognition by hand, with a keyword that would be supplied one character at a time, one would probably proceed by applying the process of elimination. Each time the  $i^{th}$  character was supplied, the strings that did not match in their  $i^{th}$  character to this new character would be crossed out from the list of possible strings.

The same process of elimination applies to the character recognition on a trie structure. Each time the  $i^{th}$  character in the keyword is read, a branch corresponding to that character is followed down to the next subtrie and the other subtrees at the previous level are ignored. Therefore, it seems very natural to employ the trie structure to perform character recognition.

It must be mentioned that the character recognition algorithm was developed based on the assumption that modifications such as addition and deletion of strings to and from the trie would be rare. Furthermore it is assumed that the main operations on the trie are search and traversal of the trie. Based on this assumption, the emphasis was placed on



the efficiency of the more important operations, namely, the search and traversal of the trie.

The algorithm can be viewed as a multi-operation algorithm. That is, a number of recognition operations and update operations on the trie are possible (other operations may be possible but are not represented by our algorithm). The recognition operations usually requested by the user are processing of a recognition character or request, processing of a rubout character, processing of a newline (end of keyword) character, and processing of requests to list possible options (strings). These operations are activated by detecting a special pre-determined character corresponding to each operation request. If the character entered by the user is not one of the special ones, then the character is taken as the next character of the keyword string being entered. The update operations consist of adding a new string to the trie and deleting a string from the trie. The latter operation will not be discussed in this paper.

### 3.2.1 The Recognition Algorithm

The character recognition algorithm differs from the general trie search (described in section 3.1.2), mainly in that the general trie search requires the entire keyword to be supplied before the search can begin, whereas in the recognition algorithm characters are processed as they are entered interactively by the user. The following is a general algorithm for character recognition on a trie.

1. - Position a search pointer to the root of the trie
2. - Repeat the following for every input character

- 2.1. - If the character is a recognition request then
  - 2.1.1. - If there has been any mismatch so far, output a bell and go to step 2.
  - 2.1.2. - If the search pointer is pointing to a branch node with only one subtrie, then output the characters in each node, each time moving the search pointer down a node until one of the following has been reached: an information node, a branch node with more than one subtrie, or a branch node that contains a terminating character (\*). Then go to step 2.
  - 2.1.3. - If the search pointer points to an information node then output the string in that node starting with the character after the last matched character, then go to step 2.
  - 2.1.4. - If the search pointer points to a branch node that contains a terminating character, then go to step 2.
  - 2.1.5. - If the search pointer points to a branch node that has more than one child output a bell and go to step 2.
- 2.2. - If the character is a rubout then
  - 2.2.1. - If there has been any mismatch so far, decrease the number of mismatches by one and go to step 2.
  - 2.2.2. - If the search pointer is pointing to a branch node, then move the pointer up one level to the parent of the branch node and go to step 2.
  - 2.2.3. - If the search pointer points to an information node then unmatch one character from the string in that node and go to step 2.
  - 2.2.4. - If the search pointer points to an information node and none of the characters of the string have been matched yet, move the pointer up to the parent of that information node and go to step 2.
- 2.3. - If the character is a newline then
  - 2.3.1. - If there has been any mismatch so far or if the string is not fully matched with a string in the trie then a match was not possible -- flag error and exit.
  - 2.3.2. - If the keyword string matches completely with a string in the trie, return the keyword string.
- 2.4. - If the character is a list-option (i.e., one that indicates the user wants a list of strings available at some point in the trie) then
  - 2.4.1. - If there has been any mismatch so far then flag error and exit
  - 2.4.2. - If the search pointer points to a branch node, then traverse the trie starting from the head node each time printing the strings found on the way, and go to step 2 when done.
  - 2.4.3. - If the search pointer points to an information node, output the string in that node starting with the character after the last matched character, then go to step 2.

- 2.5. - If the character is not special
  - 2.5.1. - If there has been any mismatch so far then output bell, increase the number of mismatches by one, then go to step 2.
  - 2.5.2. - If the search pointer points to a branch node, then if there is a subtrie corresponding to the input character, move the pointer down to the next subtrie and go to step 2, otherwise output bell and increase the number of mismatches by one and then go to step 2.
  - 2.5.3. - If the search pointer points to an information node, match the next unmatched character in the string with the incoming character and go to step 2 - if no match is possible, output bell and increase the number of mismatches by one, then go to step 2.

In this algorithm, a variable keeps track of the number of characters that the user has entered but have not matched any character. This number is increased each time a new mismatch is encountered and decreased each time a character is taken back by the user using the rubout operation. The detailed algorithm to be presented in the next chapter will clear up many of the ambiguities that may exist in this general form of the algorithm.

### 3.2.2. The Insertion Algorithm

Basically, insertion into a trie is made by first performing a search on the keyword to be added to the trie. If the string does not already exist in the trie, the search ends, with the detection of a mismatch, either at a branch node or at an information node. In either case appropriate steps need to be taken to insert the new string at this location.

The following is a general algorithm for insertion into the trie used for character recognition:

1. - Position a search pointer to the root of the trie
2. - If the trie is empty, insert an information node and place the new string in it, and exit.
3. - Find the position of the new string in the trie by taking each character of the new string and walking down the trie while there is a match. This is continued until the end of a branch (an information node) is reached, a mismatch has occurred, or no more characters are left in the new string.

4. - If the end of a branch has been reached, then insert the new string at this point in the trie by breaking it up into appropriate branch and information nodes and then exit.
5. - If the search pointer points to a branch node, then
  - 5.1. - If the node has a terminating character then the string already exists - signal an error and exit.
  - 5.2. - If the node does not contain a terminating character, then make a branch node with a terminating character and insert it at this point in the trie, and then exit.
6. - If the search pointer points to an information node, then
  - 6.1. - If the unmatched part of the new string is equal to the string in this node, then the string already exists - signal an error and exit.
  - 6.2. - Match the unmatched characters of the new string with the characters of the string in the information node until the end of one or both of the strings is reached or a mismatch occurs. Every time a match is seen, a new branch node needs to be constructed to hold the character that was matched.
  - 6.3. - If the end of the new string is reached first, then
    - 6.3.1. - Construct two branch nodes and link them together as siblings.
    - 6.3.2. - Place the next character of the string in the information node into the first branch node and a terminating character in the second one.
    - 6.3.3. - If the end of the string in the information node has been reached, then
      - 6.3.3.1. - Make a branch node with a terminating character and connect it to the first branch node above as its child, then go to 6.3.5.
    - 6.3.4. - Place the remaining part of the string in the information node back into the information node and connect the information node to the first branch node of 6.3.1.
    - 6.3.5. - Connect the second branch node of 6.3.1 to the trie and exit.
  - 6.4. - If the end of the string in the information node is reached first, then
    - 6.4.1. - Construct two branch nodes and link them together as siblings.
    - 6.4.2. - Place the next character of the new string into the first branch node and a terminating character in the second one.
    - 6.4.3. - If the end of the new string has been reached, then
      - 6.4.3.1. - Make a branch node with a terminating character and connect it to the first branch node above as its child, then go to 6.4.5.
    - 6.4.4. - Place the remaining part of the new string into the information node and connect the information node to the first branch node of 6.4.1.
    - 6.4.5. - Connect the second branch node of 6.4.1 to the trie and exit.
  - 6.5. - If the end of neither of the strings is reached, then
    - 6.5.1. - Construct two branch nodes and link them together as siblings.
    - 6.5.2. - Place the next character of the new string into the first branch node and the next character of the string in the information node into the second one.

- 6.5.3. - If the end of the new string is reached, then
  - 6.5.3.1. - Make a new branch node with a terminating character and connect it to the first branch node above, as its child. Go to 6.5.4.
  - 6.5.3.2. - Make an information node and place the remaining part of the new string in it. Connect this node to the first branch node of 6.5.1.
- 6.5.4. - If the end of the string in the information node is reached, then
  - 6.5.4.1. - Make a new branch node with a terminating character and connect it to the first branch node of 6.5.1, as its child. Go to 6.5.5.
  - 6.5.4.2. - Make an information node and place the remaining part of the string in the original information node in it. Connect this node to the second branch node of 6.5.1.
- 6.5.5. - Connect the two branch nodes of 6.5.1 to the trie in such a way that the alphabetical ordering of the characters is preserved.

## Chapter Four

### The Algorithm in Detail

#### 4.1. The Representation

In order to be able to describe an algorithm in detail, in a manner that is easy to communicate to others, one should represent the main data structure assumed in terms of an already known and simple data structure. In our case, the primary data structure is the trie. We chose to use a *binary tree* [Baron & Shapiro 1980] representation of our trie. The reason is the manipulation and implementation simplicity of the binary tree. A binary tree is a tree in which each node has a degree of two. In other words, each node of a binary tree has two subtrees, the *left subtree* and the *right subtree*.

The trick to the binary representation of the trie is that a trie branch node may correspond to several binary tree nodes. Essentially, each trie branch node is represented as a linked list of binary tree nodes. This allows us to explicitly represent only those fields of the trie branch node that have nonempty children. Each binary tree branch node has two link fields: the *child* field and the *next-sibling* field. The child field of a branch node points to its first (leftmost) child. The next-sibling field of a branch node points to its next sibling node. Thus the next-sibling field of each of the children of one parent points to the next child of the same parent, and the next-sibling field of the last child is null. Figure 4.1 shows the trie of Figure 3.4 (excluding the subtrees corresponding to the letters 'R' and 'W') in binary tree representation.

As mentioned previously, there are two different types of nodes in the trie: branch nodes and information nodes. Each branch node is represented, in the algorithm, by a node with four fields: two link fields as described above, one field that determines the type of the node (BRANCH or INFO), and one field that holds the character associated

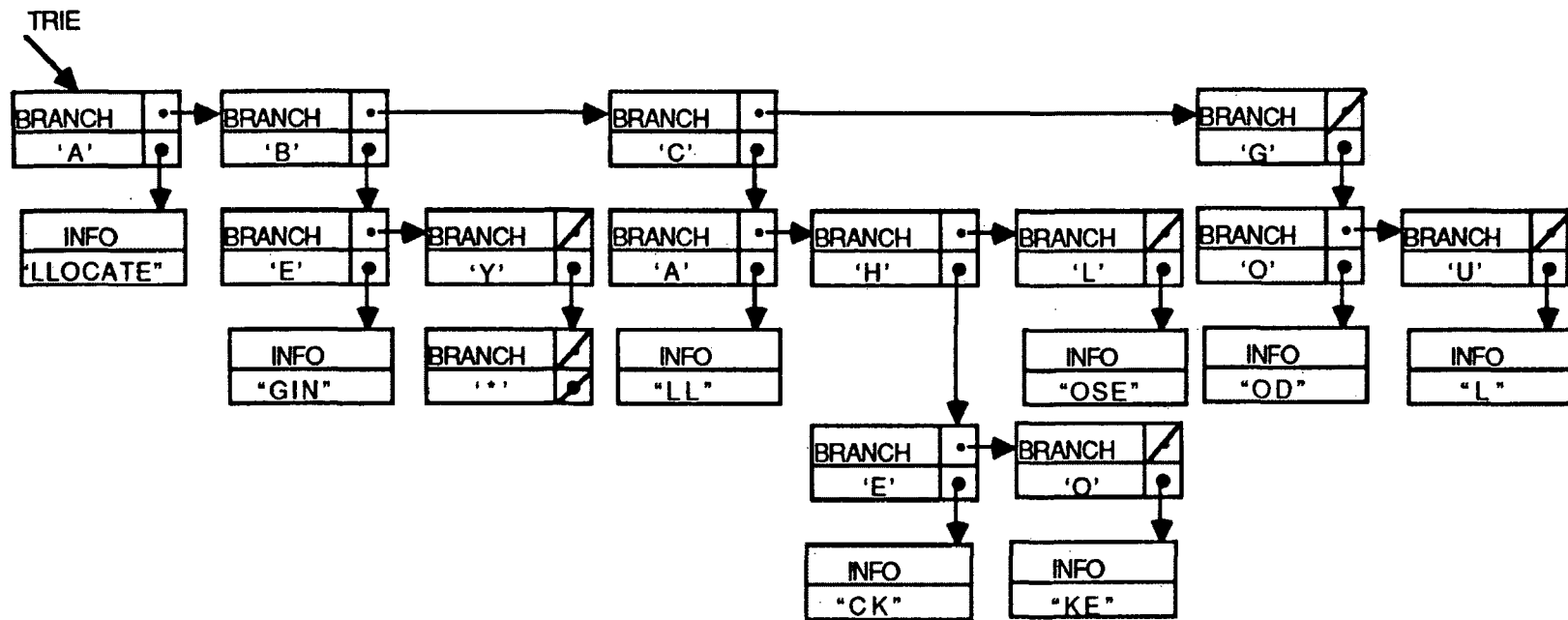


Figure 4.1

with the subtrie beneath the branch node. The information nodes are represented by nodes with two fields. The first field holds the type of the node (BRANCH or INFO) and the second field holds the remaining characters of a string in the trie. A null link in a branch node is represented by a line through the link field.

#### 4.2. Notations, Symbols, and Conventions

The algorithm to be presented has a Pascal-like structure. Anyone familiar with Pascal or most other Algol-like structured languages should be able to follow the line of the algorithm easily. Comments are placed inside square brackets. Keywords are shown in bold letters.

If a pointer  $p$  is pointing to a branch node, then  $p^{\wedge}.type$  accesses the type field of that node (which would have a value of BRANCH),  $p^{\wedge}.child$  accesses the child pointer of that node,  $p^{\wedge}.letter$  accesses the character that is represented by this node, and  $p^{\wedge}.next\text{-}sibling$  accesses the next sibling of the first (leftmost) child. If  $p$  is pointing to an information node, then  $p^{\wedge}.type$  accesses the type field of that node (which would have the value of INFO) and  $p^{\wedge}.string$  accesses the field in which the string is stored. In either case,  $p^{\wedge}$  refers to the node that  $p$  is pointing to.

An assignment operator is represented by an arrow ( $\leftarrow$ ), as in  $X \leftarrow 5$ . The concatenation of two strings is performed by the **concat** function. For example, **concat**("abcd","efgh") would give the result of "abcdefgh". In order to extract part of a string, the **substring** function will be employed. Substring will take a string and an index value (the first character of the string is assumed to be in position zero) as arguments and will return the part of the string starting from the character at the position indicated by the given index value to the end of the string. For instance, **substring**("abcdefg",3) will give the resulting string "defg". Relational operators are used as in most programming languages. However, for the sake of simplicity, an



important assumption has been made throughout the algorithm; namely that the boolean operators **and** and **or** operate as their corresponding conditional boolean operators **cand** and **cor**. The following example shows how the conditionals **cand** and **cor** work:

```

if e1 cand e2 cand e3 then
    s1;
    s2;
else if e4 cor e5 then
    s3;
else
    s4;
endif;

```

In this example, the expressions e1, e2, and e3 are evaluated in order from left to right unless one of them evaluates to false - as soon as an expression evaluates to false, evaluation of the remaining expressions is suspended and statements s1 and s2 will not be executed. In the second conditional statement, e4 is evaluated first; if it results in a true value, then e5 will not be evaluated and statement s3 will be executed.

The beauty of conditional boolean operators is that if the evaluation of an expression depends on the truth or falsity of another expression, then nested **if** statements do not have to be used. Instead, the two expressions may be checked in one conditional statement by placing the dependent expression to the right of the other expression and an operator in between them.

A number of symbols have been used in the algorithm to represent special characters. These are:

```

BELL : a bell character,
/endstr/: a string termination character ('*'),
/recog/ : a recognition request character,

```

/rubout/ : a rubout request character,  
 /listop/ : a listing of options request character,  
 /newline/ : a newline (end of keyword) character, and  
 /quit/ : a termination request character.

#### 4.3. Description of Variables

Even though the algorithm is broken up into separate modules, there are a number of variables used in most modules that have the same type and meaning. Thus, it is convenient to describe them only once. The following is a list of the variables with some explanation for each:

- **mismatch**

This variable is an integer that keeps track of the number of mismatches that have occurred between characters. The variable is incremented by one every time a new mismatched character is encountered or when a character is entered and the number of previous mismatches was not zero. **Mismatch** is decremented by one (if it is not zero) every time a *rubout* request is processed.

- **search**

This variable is used to walk down the trie. It acts as a pointer to the nodes in the trie. In actuality, **search** is a record with two fields; the first is a link pointer field that points to the nodes of the trie (**search.ptr**), and the second field is a counter used to keep track of the number of characters in an information field that have been matched (**search.index**).

- **ptr-stack & str-stack**

These are two stack abstract data structures. The operations allowed on these structures are:

```

Clearstack (stack)
    -- clears the stack of any items,

Emptystack (stack)
    -- returns true if the stack is empty, and false otherwise

Push (stack, item(s))
    -- pushes either a single item or, in the case the stack is a
    str-stack, a number of items (characters) onto the stack,

Pop (stack, item)
    -- pops a single item off of the stack and returns it in the
    variable item

```

As our **search** pointer moves down the trie to a subtrie of a branch node, the pointer to the parent branch node is pushed onto the **ptr-stack**. This pointer can later be used for walking back up the trie.

The **str-stack** is used to collect individual characters either from the input or by walking down the trie. The characters gathered in this stack constitute the string that the user is entering.

#### 4.4. The Algorithm

The algorithm has been developed in parts for the reason of clarity, readability, and most importantly, for modularity. The modules communicate using parameters, as in the Pascal programming language. In the next few sections, each module is presented with a description and appropriate explanations.

##### 4.4.1. The Recognition Algorithm

This module is the top level in the character recognition algorithm. It is basically in charge of reading characters from the input stream and calling the appropriate modules depending on what the characters are. It will terminate either by detecting a /quit/ character or after a Newline request has been completed.

For this algorithm to work, a trie containing the predetermined strings will have to be built ahead of time. Then, the algorithm uses a pointer, pointed at the root of the trie, to perform the recognition. The original pointer (at the root of the trie) is not modified; instead a **search.ptr** (see section 4.3) is used to traverse the trie. This pointer is initially pointed at the root of the trie.

### Recognition Algorithm:

**Inputs:** trie [a pointer to the root of the trie]

**Outputs:** str-stack [the recognized string]

1. [Initialize variables]
  - done  $\leftarrow$  false [Local boolean loop variable]
  - mismatch  $\leftarrow$  0
  - search.ptr  $\leftarrow$  trie
  - search.index  $\leftarrow$  0
  - Clearstack (ptr-stack)
  - Clearstack (str-stack)
2. [Process every character as it is read in]
  - while not done do**
    - read**(cur-char)
    - case** cur-char **of**
      - /recog/ : Process\_Recog( mismatch, search, str-stack, ptr-stack )
      - /rubout/: Process\_Rubout( mismatch, search, str-stack, ptr-stack)
      - /listop/ : Process\_Listop( mismatch, search, str-stack )
      - /newline/ : done  $\leftarrow$  true  
 Process\_Newline( mismatch, search, str-stack )  
 return(str-stack)
      - /quit/ : done  $\leftarrow$  true
      - others : Process\_Other( mismatch, search, str-stack,  
 ptr-stack, cur-char )
    - end case**
  - end while**

**End of Recognition Algorithm.**

#### 4.4.1.1. The Process\_Recog Algorithm

The following algorithm is the procedure called upon when the recognition algorithm detects a recognition request character in the input stream.

**Algorithm for Process\_Recog:****Inputs:** mismatch, search, str-stack, ptr-stack**Outputs:** search, str-stack, ptr-stack

1. [Check for mismatch]
    - if mismatch > 0 then
      - output(BELL)
      - return
    - end if
  2. [Output the unique part of the string]
    - while (search.ptr^.type = BRANCH) and
      - (search.ptr^.next-sibling = nil) and
      - (search.ptr^.letter <> /endstr/) do
        - output(search.ptr^.letter)
        - Push(str-stack, search.ptr^.letter)
        - Push(ptr-stack, search.ptr)
        - search.ptr ← search.ptr^.child
    - end while
  3. [Search pointer is pointing to a INFO node — output that string]
    - if (search.ptr^.type = INFO) then
      - output(Substring(search.ptr^.str, search.index+1))
      - Push(str-stack, Substring(search.ptr^.str, search.index+1))
      - search.index ← Length(Substring(search.ptr^.str, search.index+1))
  4. [Search pointer is at the end of a string that ends at /endstr/]
    - else if (search.ptr^.type = BRANCH) and
      - (search.ptr^.letter = /endstr/) then
        - [ do nothing ]
  5. [No part of the string is unique]
    - else
      - output(BELL)
    - end if
- End of Process\_Recog Algorithm.**

In step 1, the procedure checks to see if there have been any mismatches so far; if so, then it outputs a bell and returns. Step 2 determines if the search pointer is pointing to a BRANCH node that does not have any siblings. If true, it repeatedly outputs the characters found in each BRANCH node, saves the character on the **str-stack**, saves the pointer to the BRANCH node on the **ptr-stack**, and moves the pointer down to the next subtree. This process continues until an INFO node is reached, or a BRANCH node is

reached that has a next-sibling or it has a string termination character in its *string* field. Step 3 determines if the search pointer is pointing to an INFO node, if so, the procedure outputs the string in that node, saves the string on the *str-stack*, and increments the *index* field of the *search* variable by the length of the string that was output. If the search pointer points to a BRANCH node that has a terminating character, then the condition of step 4 holds true and thus no operations are needed. If none of the conditions of steps 1, 2, 3, and 4 hold true, then the pointer must be pointing to a BRANCH node that has next siblings and therefore no part of the string is unique in the trie. In this case the procedure outputs a bell and returns (step 5).

#### 4.4.1.2. The Process\_Other Algorithm

The following algorithm processes characters that are not among the special request characters. These characters are assumed to be a part of the keyword that the user is entering.

##### Algorithm for Process-Other:

**Inputs:** mismatch, search, str-stack, ptr-stack, cur-char

**Outputs:** mismatch, search, str-stack, ptr-stack

1. [Echo the entered character on the user's monitor]  
    **output**(cur-char)
2. [Check for mismatch]  
    **if** mismatch > 0 **then**  
        **output**(BELL)  
        mismatch  $\leftarrow$  mismatch + 1  
        **return**  
    **end if**

3. [Check to see if we have a list of BRANCH nodes to choose from]
 

```

      if (search.ptr^.type = BRANCH) then
        prev ← search.ptr
        Get-Next-Node(search.ptr, parent, cur-char)
        if search.ptr <> nil then
          Push(str-stack, cur-char)
          Push(ptr-stack, prev)
        else
          output(BELL)
          mismatch ← mismatch + 1
          search.ptr ← prev
        end if
      
```
4. [Search pointer is pointing to a INFO node — walk down that string]
 

```

      else if (search.index < length(search.ptr^.str)) and
        (cur-char = Getchar(search.ptr^.str, search.index + 1)) then
        search.index ← search.index + 1
        Push (str-stack, cur-char)
      
```
5. [Character does not match anything]
 

```

      else
        output(BELL)
        mismatch ← mismatch + 1
      end if
    
```

#### **End of Process\_Other Algorithm.**

Here is how the Process\_Other algorithm works: First the entered character is echoed to the user's monitor, then a check is made in step 2 to see if there have been any mismatches so far, if so, a bell is outputted and the number of mismatches is incremented by one. The procedure is then exited. Step 3 determines if the search pointer is at a BRANCH node. If this is true, then a search is performed to find a node that contains the character just read in, among the siblings of this BRANCH node. This search is performed by a module called Get-Next-Node. An algorithm of this module will follow shortly. If a node is located, then the current character is saved on the **str-stack**, the pointer to the original BRANCH node is saved on the **ptr-stack**, and the search pointer is advanced to the child of the newly found BRANCH node. If a node was not found, then a mismatch has occurred -- ouput a bell and increase the number of mismatched

characters by one. At step 4 it is certain that the search pointer points to an INFO node. If the length of the string stored in the INFO node is greater than the value of the index field of the *search* variable, and the next character in the INFO node is the same as the current character, then a match is possible -- the index field of the *search* variable is incremented by one and the current character is saved on the *str-stack*. Otherwise, a mismatch occurs -- output a bell and increment the number of mismatches by one.

The following is the algorithm for the Get-Next-Node process. The algorithm basically searches the linked list of next-siblings starting from the BRANCH node that *ptr* points to. If it finds a node that has a character equal to *ch*, then it returns a pointer to this node and a pointer to the child of this node, otherwise it returns a nil pointer.

**Algorithm for Get-Next-Node:**

**Inputs:** ptr [Pointer to the first BRANCH node],  
ch [The character to be searched for in the siblings of the first  
BRANCH node]

**Outputs:** ptr [Pointer to the child of the BRANCH node located]  
parent [Pointer to the parent of node pointed to by ptr]

1. [Search the next-siblings for a node that contains ch]
  - while** (ptr <> nil) **and** (ptr^.letter < ch) **do**
  - ptr ← ptr^.next-sibling
  - end while**
2. [Nothing found, return nil in ptr -- value of parent is not significant]
  - if** (ptr = nil) **or** (ptr^.letter > ch) **then**
  - ptr ← nil
  - return**
  - end if**
3. [A node was found, return a pointer to it and a pointer to its parent]
  - parent ← ptr
  - ptr ← ptr^.child

**End of Get-Next-Node Algorithm.**



#### 4.4.1.3. The Process\_Rubout Algorithm

The processing of a rubout request is performed by the following algorithm. The last character that was entered by the user is erased from the input device.

**Algorithm for Process-Rubout:**

**Inputs:** mismatch, search, str-stack, ptr-stack

**Outputs:** mismatch, search, str-stack, ptr-stack

1. [Remove the last character entered from the user's monitor]  
    **delchar;**
2. [Check for mismatch]  
    **if mismatch > 0 then**  
        mismatch  $\leftarrow$  mismatch - 1  
    **return**  
    **end if**
3. [Remove the last character from str-stack]  
    **if Emptystack(str-stack) then**  
        **return**  
    **else**  
        Pop (str-stack)  
    **end if**
4. [Search pointer is pointing to a BRANCH node — move the search pointer up to the parent of that node]  
    **if (search.ptr.type = BRANCH) and (not Emptystack(ptr-stack)) then**  
        search.ptr  $\leftarrow$  Pop(ptr-stack)  
        search.index  $\leftarrow$  0  
    **return**
5. [Search pointer is pointing to a INFO node and some of the characters of the string in that node have already been matched — unmatch one character]  
    **else if search.index > 0 then**  
        search.index  $\leftarrow$  search.index - 1  
    **return**
6. [Search pointer is pointing to a INFO node and none of the characters of the string in that node have been matched yet — move the search pointer up to the parent of that node]  
  
        search.ptr  $\leftarrow$  Pop(ptr-stack)  
        search.index  $\leftarrow$  0

**End of Process\_Rubout Algorithm.**

The first operation in this module is to erase the last character entered from the user's monitor. Again, as in the other modules, a check is performed for previous mismatches. In this case however, if there have been any mismatches, the number is decreased by one since the user has decided to take back the last character that was entered. After that, no further processing is needed since mismatched characters are not saved on the stack. Step 3 pops one character from the *str-stack*. The search pointer is then adjusted accordingly. Step 4 moves the search pointer up to the parent of the node it is pointing to, if this node is a *BRANCH* node. If the node is an *INFO* node and at least one of the characters in that node have been matched, then step 5 simply decrements the index field of the *search* variable by one. Otherwise, step 6 just moves the search pointer up to the parent node.

#### 4.4.1.4. The Process\_Newline Algorithm

In case a newline character is detected by the recognition process, the following procedure is activated. A newline character actually means that the user has completed entering the keyword. The algorithm should check to make sure that the completed keyword is one that exists in the trie.

##### **Algorithm for Process-Newline:**

**Inputs:** mismatch, search, str-stack

**Outputs:** str-stack

1. [If the string in str-stack does not match a string in the trie, output error message]
  - if (mismatch > 0) or
  - ((search.ptr^.type = *BRANCH*) and
  - (search.ptr^.letter <> /endstr/)) or
  - ((search.ptr^.type = *INFO*) and
  - (length(search.ptr^.str) <> search.index)) then
  - Recog-Error

2. [Return the matched string]
  - else
  - return(str-stack)
  - end if

#### End of Process\_Newline Algorithm.

If there have been any mismatches at all, then the keyword is not valid, thus step 1 makes a call to some error routine. Step 1 further checks to see if the search pointer points to a BRANCH node that does not have a terminating character in it or it points to an INFO node that has not matched all of the characters in its *string* field. If either of the two conditions is true, then the keyword is not found in the trie -- the error routine needs to be called. Otherwise, the keyword does exist in the trie. Thus, in step 2 the string accumulated in the *str-stack* is returned.

#### 4.4.1.5. The Process\_Listop Algorithm

This procedure displays the strings found in the subtrie pointed to by the search pointer. It makes use of the Traverse-Trie algorithm to be presented in the next section. A local variable, *prefix-str-stack*, is used to collect characters from the nodes, as the trie is traversed. This variable is of the same type as *str-stack* discussed in section 4.3.

##### Algorithm for Process-Listop:

Inputs: mismatch, search, str-stack

Outputs: none

1. [Place the input string into the prefix stack]
  - prefix-str-stack ← str-stack
2. [Check for mismatch]
  - if mismatch > 0 then
  - Recog-Error
3. [traverse the subtrie starting from a BRANCH node]
  - else if search.ptr.type = BRANCH then
  - Traverse-Trie (search.ptr, prefix-str-stack)
  - output(str-stack)

4. [search pointer is pointing to an INFO node — output that string]
  - else
    - output(concat(str-stack, search.ptr^.str))
  - end if

#### **End of Process\_Listop Algorithm.**

If there have been any mismatches thus far, it is not possible to list any options. In this case an error routine is called at step 2. Step 3 determines if the search pointer is pointing to a BRANCH node, if so, the subtrie beneath this node is traversed and the strings displayed. The traversal is performed by the Traverse-Trie module which will be presented in the next section. If the search pointer is pointing to an INFO node, then all that needs to be done is to output the string in that node.

#### **4.4.1.6. The Traverse\_Trie Algorithm**

Traverse-Trie is presented in a recursive fashion in order to simplify the algorithm. A depth first traversal approach is followed so that the strings are displayed in alphabetical order.

#### **Algorithm for Traverse-Trie (p, prefix-str-stack):**

**Inputs:** p [Pointer used to move through the trie],  
 prefix-str-stack  
**Outputs:**

1. [Return if at the end of a branch]
  - if p = nil then
    - return
  - end if
2. [Search pointer is pointing to a INFO node — output that string]
  - else if p^.type = INFO then
    - output(concat(prefix-str-stack, p^.str))
    - return
3. [String terminates with a /endstr/ marker — output that string]
  - else if p^.letter = /endstr/ then
    - output(prefix-str-stack)

4. [String does not terminate — traverse the child of the current node]
  - else
    - Push (prefix-str-stack, p^.letter)
    - Traverse-Trie (p^.child, prefix-str-stack)
    - Pop (prefix-str-stack)
  - end if
5. [Traverse the next-sibling of the current node]
  - Traverse-Trie (p^.next-sibling, prefix-str-stack)

**End of Traverse-Trie Algorithm.**

The algorithm works in the following general way: Characters of the BRANCH nodes are pushed onto the prefix-str-stack and the children of the BRANCH nodes are followed recursively down the trie (step 4) until either an INFO node or a BRANCH node with a terminating character is reached. At this point (step 2 or 3), the contents of the prefix-str-stack are output. In case the terminating node were an INFO node, the string of the INFO node would be concatenated to the contents of the prefix-str-stack before it is output (step 2). Once the end of one branch has been reached, the next-siblings of the branch nodes are processed recursively (step 5).

#### 4.4.2. The Insertion Algorithm

As with the character recognition algorithm, the insertion algorithm is broken up into modules. The next section presents the top level or the main driver module for the insertion algorithm and the other sections following the next one describe the lower level modules of the algorithm. In the algorithms, two operations are used in order to create new nodes. Make-Branch-Node creates a branch node and Make-Info-Node creates a branch node. Both of these operations return a pointer to the node that they create.

##### 4.4.2.1. The Add-To-Trie Algorithm

The main driver for the insertion of a new string into a trie requires that two parameters be passed to it. The first is a pointer *trie*, to the root of the trie structure and

the second is the new string *str* , to be inserted into the trie. The modified trie is then returned.

The algorithm uses an *index* variable to keep track of which character of the new string is currently being processed. As in the recognition algorithm, a **search** (see section 4.3) variable is used to move through the trie. The pointer field of this variable, **search.ptr** is initially set to the root of the trie. Two pointers are employed to make the actual insertions of new nodes into the trie. These are *prev* and *parent*. *Parent* is set by the Get-Next-Node (see section 4.4.1.2) module used in this algorithm. *Prev* is used to keep track of the previous position of the search pointer after it has been moved.

#### Algorithm for Add-To-Trie:

**Inputs:** trie, str

**Outputs:** trie

1. [Initialize variables]
  - index  $\leftarrow$  1
  - search.ptr  $\leftarrow$  trie
  - search.index  $\leftarrow$  0
  - prev  $\leftarrow$  trie
  - parent  $\leftarrow$  nil
2. [If the trie is empty then insert an information node]
  - if** trie = nil **then**
    - trie = Make-Info-Node
    - trie^.str  $\leftarrow$  str
    - return**
  - end if**
3. [Locate the insertion point in the trie]
  - while** search.ptr  $\neq$  nil **and**
    - search.ptr.type = BRANCH **and**
    - index  $\leq$  Length(str) **do**
      - prev  $\leftarrow$  search.ptr
      - Get-Next-Node(search.ptr,parent,Getchar(str, index))
      - index  $\leftarrow$  index + 1
  - end while**

4. [End of a branch has been reached]
  - if search.ptr = nil then
    - Insert-To-Sibling-List(trie, prev, parent, Substring(str,index-1))
5. [The new string ended at a branch node - insert a terminating node]
  - else if search.ptr^.type = BRANCH then
    - if search.ptr^.letter <> /endstr/ then
      - new-node ← Make-Branch-Node
      - new-node^.letter ← /endstr/
      - new-node^.next-sibling ← search.ptr
      - new-node^.child ← nil
      - prev^.child ← new-node
    - else
      - output("string already exists")
      - exit
    - end if
6. [Search pointer points to an information node - break up the information node and insert the new string]
  - else if search.ptr^.str = Substring(str,index) then
    - output("string already exists")
    - exit
  - else
    - Process-Info-Node(trie,str,index,search,parent)
  - end if

#### End of Add-To-Trie Algorithm.

If the trie is initially empty, the algorithm simply builds a new information node and places *str* in it. It then sets the trie to point to this new node and exists (step 1). If the trie already has some nodes in it, then in step 2 a search is performed on the characters of *str* until either the search pointer becomes nil, the search pointer reaches an information node, or *index* becomes bigger than the length of *str*. Step 4 determines if the search pointer is nil (i.e., it has reached the end of a branch), if so then the Insert-To-Sibling-List module is activated (see section 4.4.2.3). If the search pointer points to a branch node, then step 5 determines if *str* already exists in the trie or not. If it exists, a message is displayed and the module is exited. Otherwise, a new branch node is constructed with a terminating character in it. This branch node is then inserted at the location where search pointer is. Step 6 determines if the search pointer points to an

information node. It further checks to see if the string in this information node matches the unmatched part of *str* . If so, then the string already exists - display a message and exit. Otherwise, the Process-Info-Node routine is activated to handle the breaking up of this information node and inserting of the new string (see next section).

#### 4.4.2.2. The Process-Info-Node Algorithm

##### Algorithm for Process-Info-Node:

**Inputs:** trie, str, index, search, parent

**Outputs:** trie

1. [Match the characters of the string in the information node with the unmatched characters of *str* ]
  - while** Substring(str, index) <> "empty string" and  
 Substring(search.ptr^.str, search.index+1) <> "empty string" and  
 Getchar(str, index) = Getchar(search.ptr^.str, search.index+1) **do**  
     node1 ← Make-Branch-Node  
     node1^.next-sibling ← nil  
     node1^.letter ← Getchar(str, index)  
     **if** parent = nil **then**  
         trie ← node1  
     **else**  
         parent^.child ← node1  
     **end if**  
     parent ← node1  
     index ← index + 1  
     search.index ← search.index + 1  
**end while**
2. [Either end of *str* or end of the string in the information node has been reached]
  - if** Substring(str, index) = "empty string" **or**  
 Substring(search.ptr^.str, search.index + 1) = "empty string" **then**  
     Insert-Short-Strings(trie, str, index, search, parent)
3. [Neither end of *str* or end of the string in the information node has been reached]
  - else**  
     Insert-Long-Strings(trie, str, index, search, parent)  
**end if**

**End of Process-Info-Node Algorithm.**



Step 1 of the algorithm builds a branch node for each character matched between *str* and the string in the information node. This process is continued until the end of one or both of the strings is reached or a mismatched character is detected. Step 2 determines if the end of either of the two strings has been reached. If so, it activates the Insert-Short-Strings routine. If not, it activates the Insert-Long-Strings process.

#### 4.4.2.3. The Insert-To-Sibling-List Algorithm

##### Algorithm for Insert-To-Sibling-List:

**Inputs:** trie, ptr, parent, str

**Outputs:** trie

prev  $\leftarrow$  ptr

1. [Locate the insertion position in the siblings]
  - while** (ptr  $\neq$  nil) **and** (ptr^.letter < Getchar(str,1)) **do**
    - prev  $\leftarrow$  ptr
    - ptr  $\leftarrow$  ptr^.next-sibling
  - end while**
2. [Create a branch node with the first character of *str* ]
  - node1  $\leftarrow$  Make-Branch-Node
  - node1^.letter  $\leftarrow$  Getchar(str,1)
  - node1^.next-sibling  $\leftarrow$  ptr
3. [Insert the new node as the first node in list of siblings]
  - if** ptr = prev **then**
    - if** parent = nil **then**
      - trie  $\leftarrow$  node1
    - else**
      - parent^.child  $\leftarrow$  node1
    - end if**
4. [Insert the new node in the middle of list of siblings]
  - else**
    - prev^.next-sibling  $\leftarrow$  node1
  - end if**

5. [The end of *str* has been reached]
  - if Substring(*str*,2) = "empty string" then
    - node2  $\leftarrow$  Make-Branch-Node
    - node2^.letter  $\leftarrow$  /endstr/
    - node2^.child  $\leftarrow$  nil
    - node2^.next-sibling  $\leftarrow$  nil
6. [The end of *str* has not been reached]
  - else
    - node2  $\leftarrow$  Make-Info-Node
    - node2^.str  $\leftarrow$  Substring(*str*,2)
  - end if
7. [Connect the node with remaining part of *str* to the node inserted in step 3 or 4]
  - node1^.child  $\leftarrow$  node2

**End of Insert-To-Sibling-List Algorithm.**

This module first locates the insertion position of the new string among the siblings of the node pointed to by *ptr*. *Prev* is kept at the node before the node pointed to by *ptr*. Step 2 creates a new branch node and places the first character of *str* in it. Then it connects the next-sibling link of this node to the node pointed to by *ptr*. Step 3 determines if the node found in step 1 is the first node in the list of siblings, if so, and if the node is the root node, then the pointer to the root needs to be moved to point to this new node. If *ptr* was not pointing to the root node, then the parent of the node pointed to by *ptr* is connected to this new node as its child. In step 4, the new node is inserted in the middle of a list of siblings. Now we need to store the remaining part of *str*. Step 5 determines if there is only one character left in *str*. If so, then that character is placed inside a new branch node and, in step 7, the branch node is made the child of the node created in step 1. If more than one character is left in *str*, then the remaining part of the string is placed in an information node and then, in step 7, it is made the child of the node created in step 1.

#### 4.4.2.4. The Insert-Short-Strings Algorithm

##### Algorithm for Insert-Short-Strings:

**Inputs:**       trie, str, index, search, parent

**Outputs:**     trie

1. [The end of *str* has been reached]
  - if** Substring(str, index) = "empty string" **then**
    - long\_str ← search.ptr^.str
    - long\_index ← search.index + 1
2. [The end of the string in the information node pointed to by the search pointer has been reached]
  - else**
    - long\_str ← str
    - long\_index ← index
  - end if**
3. [Make two branch nodes, one for the next unmmatched character of *long\_str* and one for the terminating character]
  - node1 ← Make-Branch-Node
  - node2 ← Make-Branch-Node
  - node2^.next-sibling ← node1
  - node1^.letter ← Getchar(long\_str, long\_index)
  - node2^.letter ← /endstr/
  - node1^.next-sibling ← nil
4. [Only one character left in the string, place it in a branch node]
  - if** Substring(long\_str, long\_index+1) = "empty string" **then**
    - node3 ← Make-Branch-Node
    - node3^.letter ← /endstr/
    - node3^.child ← nil
    - node3^.next-sibling ← nil
    - node1^.child ← node3
5. [More than one character is left in the string, place the remaining characters back into the information node]
  - else**
    - search.ptr^.str ← Substring(long\_str, long\_index+1)
    - node1^.child ← search.ptr
  - end if**

6. [Connect the new nodes to the trie via the parent of the information node]  
 $\text{parent}^{\wedge}.\text{child} \leftarrow \text{node2}$

**End of Insert-Short-Strings Algorithm.**

Step 1 of this algorithm determines if the end of *str* has been reached. If so, the string in the information node is assigned to a temporary variable called *long\_str* and the value of the index field of the *search* variable is assigned to the variable *long\_index*. If the end of *str* has not been reached, then the end of the string in the information node must have been reached. In this case, in step 2, the value of *str* and *index* are assigned to *long\_str* and *long\_index* respectively. In step 3, two branch nodes, *node1* and *node2* are created. The next unmatched character of *long\_str* is placed in *node1* and a terminating character is placed in *node2*. *Node2* is then connected to *node1* via its *next-sibling* link and the *next-sibling* link of *node1* is set to *nil*. This assures that the node with the terminating character is always the first node in a list of siblings. At this point a check is made in step 4 to see if the end of *long\_str* has been reached. If it has not, then the string in the information node is replaced with the remaining part of *long\_str* and the node is connected to *node1* as its child, otherwise, a branch node is created with a terminating character and is connected to *node1* as its child. At the end, in step 6, the new nodes are connected to trie by linking the node pointed to by the parent of the original information node to *node2*.

**4.4.2.5. The Insert-Long-Strings Algorithm**

**Algorithm for Insert-Long-Strings:**

**Inputs:**       trie, str, index, search, parent  
**Outputs:**     trie

1. [Make two branch nodes, one for the next unmatched character of *str* and one for the next unmatched character of the string in the information node pointed to by the search pointer]
  - node1  $\leftarrow$  Make-Branch-Node
  - node2  $\leftarrow$  Make-Branch-Node
  - node1^.letter  $\leftarrow$  Getchar(str,index)
  - node2^.letter  $\leftarrow$  Getchar(search.ptr^.str, search.index+1)
  - index  $\leftarrow$  index + 1
  - search.index  $\leftarrow$  search.index + 1
2. [The end of *str* has been reached]
  - if Substring(str,index) = "empty string" then
    - node3  $\leftarrow$  Make-Branch-Node
    - node3^.letter  $\leftarrow$  /endstr/
    - node3^.next-sibling  $\leftarrow$  nil
    - node3^.child  $\leftarrow$  nil
    - node1^.child  $\leftarrow$  node3
3. [The end of *str* has not been reached]
  - else
    - node3  $\leftarrow$  Make-Info-Node
    - node3^.str  $\leftarrow$  Substring(str,index)
    - node1^.child  $\leftarrow$  node3
  - end if
4. [The end of the string in the information node pointed to by the search pointer is reached]
  - if Substring(search.ptr^.str,search.index+1) = "empty string" then
    - node3  $\leftarrow$  Make-Branch-Node
    - node3^.letter  $\leftarrow$  /endstr/
    - node3^.next-sibling  $\leftarrow$  nil
    - node3^.child  $\leftarrow$  nil
    - node2^.child  $\leftarrow$  node3
5. [The end of the string in the information node pointed to by the search pointer has not been reached]
  - else
    - node3  $\leftarrow$  Make-Info-Node
    - node3^.str  $\leftarrow$  Substring(search.ptr^.str, search.index + 1)
    - node2^.child  $\leftarrow$  node3
  - end if

6. [Arrange the two branch nodes so that the alphabetical ordering of their characters is preserved]
 

```

        if node1^.letter > node2^.letter then
            node2^.next-sibling ← node1
            node1^.next-sibling ← nil
            connect ← node2
        else
            node1^.next-sibling ← node2
            node2^.next-sibling ← nil
            connect ← node1
        end if
      
```
7. [*trie* points to an information node - connect *trie* to the new nodes]
 

```

        if (parent = nil) or (parent^.type = nil) then
            trie ← connect
        end if
      
```
8. [Connect the new nodes to the node pointed to by the parent of the original information node]
 

```

        else
            parent^.child ← connect
        end if
      
```

#### End of Insert-Long-Strings Algorithm.

In step 1 two branch nodes are created. Node1 holds the next unmatched character of *str* and *node2* holds the next unmatched character of the string in the information node pointed to by the search pointer. If the end of *str* has been reached, step 2 makes a branch node with a terminating string and makes it the child of *node1*, otherwise, in step 3, an information node is built and the remaining part of *str* is placed in it. This node is then made the child of *node1*. Step 4 determines if the end of the string in the information node has been reached. If true, a branch node with a terminating character is constructed and is connected to *node2* as its child, otherwise, an information node is built and the remaining part of the string is placed in it. This node is then linked to *node2* as its child. Nodes *node1* and *node2* are connected as siblings in such a way as to maintain the alphabetical ordering of the characters (step 6). If the *trie* has only an information node in it, then the pointer to the *trie* needs to be connected to the new nodes

(step 7), otherwise, the new nodes are linked to the node pointed to by the parent of the original information node via its child link.

## Chapter Five

### Analysis of the Algorithm

#### 5.1. General Discussion

The purpose of analysing the character recognition algorithm is to predict how much time or space is required by a computer to execute an implementation of the algorithm. Our analysis will be broken up into two types of efficiency considerations; that of *time*, and that of *space* (storage). The discussion on time efficiency will be broken up into separate analyses of the algorithms presented in chapter 4.

#### 5.2. Time Efficiency of the Algorithm

The running time of a program is usually a function of the input or the length of the input, the time complexity of the algorithm underlying the program, the quality of code generated by the compiler, and the nature and speed of machine instructions. We can express an approximation of the relationship between the length of the input and the running time of an algorithm using a mathematical notation called *order of magnitude* or *Big-O* notation [Aho & Others 1983]. We say the run time complexity of some algorithm is  $O(f(n)) = g(n)$ . This means, if the actual run time of an algorithm is  $g(n)$  where  $n$  is a measure of the size of the input, we say the run time complexity is  $O(f(n))$  if there exist positive constants  $c$  and  $n_0$ , such that

$$g(n) \leq c f(n), \quad n \geq n_0.$$

For instance, if

$$g(n) = 2n^2 + 6n + 19,$$

then for  $c = 3$  and  $n_0 = 25$ ,  $g(n) \leq c n^2$ ,  $n \geq n_0$  holds. Thus,  $f(n) = n^2$  (or in Big-O notation,  $O(n^2)$ ). In other words, for large values of  $n$ , the term  $2n^2$ , or more



specifically  $n^2$ , will dominate the function  $g(n)$ . Since this notation ignores constant factors in the run time, it is independent of such things as the speed of the hardware and the quality of the compiler.

A constant or bounded running time or complexity is referred to as  $O(1)$ . This is the best complexity that an algorithm can have. A complexity of  $O(\log_2 N)$  is slightly worse than  $O(1)$ . A running time of  $O(\log_2 N)$  is better than  $O(N)$  (linear time), which in turn is better than  $O(N \log N)$ . A complexity of  $O(N \log N)$  is better than  $O(N^2)$  (quadratic time).  $O(N^3)$  (cubic time) is worse than  $O(N^2)$ , and  $O(2^N)$  (exponential) is worse than  $O(N^m)$  for every positive  $m$  (actually unacceptable for most applications).

In the following subsections we shall discuss the complexities of the recognition and the trie insertion algorithms using the Big-O notation described above.

### 5.2.1. Analysis of the Recognition Algorithm

In our recognition algorithm there are actually two types of input. The first is the trie containing the predetermined strings, and the second is the keyword that will be received, from the user, one character at a time. Thus, one would presume that the efficiency of the character recognition algorithm depends on these two types of input.

The complexity of our algorithm is not that of a search through a trie.<sup>1</sup> The major difference is that in the case of a search through a trie, a keyword to be searched for is supplied and then the trie is searched for that keyword. In our case, however, the search is broken up into small pieces corresponding to the individual characters of the keyword. The character recognition search pointer moves through the trie each time a new character is entered or each time an old character is erased (taken back) by the user. Therefore, the efficiency that we need to concern ourselves with is that of processing individual

---

<sup>1</sup>In [Horowitz & Sahni 1983], complexity of the search algorithm for tries is given as  $O(k)$ , where  $k$  is the number of levels in the trie.

characters instead of an entire keyword. However, the overall time complexity of the algorithm is actually the sum of the time complexities associated with processing each character of a keyword.

The modules, **Process\_Rubout** (section 4.4.4) and **Process\_Newline** (section 4.4.5) are of constant complexity. These modules have a linear structure and contain no loops.

The most important part of the algorithm is **Process\_Recog** (section 4.4.2). The major complexity of processing in this module is hidden in the *while* loop. This loop is executed in the case that the search pointer points to a **BRANCH** node that has no siblings and has a subtree with one or more **BRANCH** nodes with no siblings. The loop is executed once for each of these **BRANCH** nodes in the subtree. Figure 5.1 shows such a situation. In this trie the two words *ALLOCATES* and *ALLOCATED* are represented. The first eight characters of these two are equivalent and thus if our search pointer is initially pointing to the root of the trie and we signal a recognition request, the first eight characters, *ALLOCATE*, have to be displayed. This requires that the loop be executed eight times. The worst case complexity of this module is therefore proportional to the number of levels in the trie minus one, or  $O(k)$ , where  $k$  is the number of levels in the trie. Of course,  $k$  is bounded by the length of the longest string stored in the trie.

The complexity of the **Process\_Other** (section 4.4.3) module is really in the sub-module that it activates, namely **Get-Next-Node** (section 4.4.3). This sub-module searches the siblings of a given **BRANCH** node for a node containing a specific character. Therefore, the time complexity is proportional to the number of siblings that a **BRANCH** node could have. In other words, **Get-Next-Node** is of  $O(|\Sigma|)$  complexity, where  $|\Sigma|$  is the magnitude or cardinality of the alphabet (see section 3.1.2). Since we consider  $\Sigma$  to be fixed, this algorithm is of constant complexity.

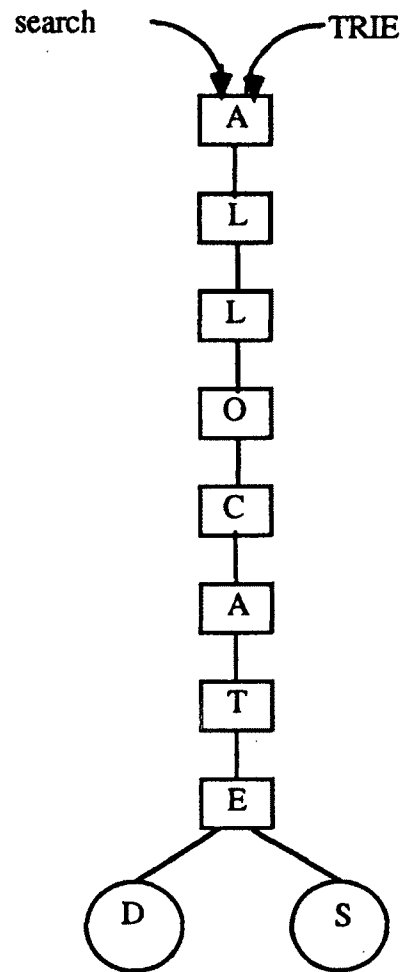


Figure 5.1  
A nine level trie with ALLOCATED & ALLOCATES

---

**Traverse-Trie** (section 4.4.7) is the recursive module activated by **Process\_Listop** (section 4.4.6) and is the main contributor to the overall complexity of these two modules. Basically, **Traverse-Trie** visits every node contained in the subtrie beneath the **BRANCH** node that search pointer is pointing to. In the worst case, the search pointer is initially pointing to the root of the trie. The complexity of the algorithm is therefore  $O(m)$ , where  $m$  is bounded by the number of nodes in the trie to be traversed (see section 5.3).

The overall complexity of the character recognition algorithm can be  $O(n m)$  if during an interactive session the user chooses to perform **Process\_Listop** after entering each character, where  $n$  is the total number of characters entered by the user. In the case that the user performs recognition on each character the overall complexity would be  $O(n k)$ , and if the user simply types in all the characters without choosing any other operations, the overall complexity is  $O(n)$ .

### 5.2.2. Analysis of the Insertion Algorithm

Since the insertion of a new string into the trie structure is a single operation, it is possible to calculate a running time or complexity for the entire algorithm. There are basically three areas where a significant complexity may reside. These three areas of complexity are characterized by three types of loops in the insertion algorithm. The first type of loop (step 3 of Add-To-Trie) searches down the trie in order to locate the insertion position for the new string. The complexity of this loop is on the order of the number of levels in the trie, or  $O(k)$ , because in the worst case, the trie needs to be followed from the root down to the lowest leaf node. As before,  $k$  is the number of levels in the trie and is bounded by the length of the longest string stored in the trie.

The second type of loop in the insertion algorithm is one that searches a list of sibling nodes for a given branch node. These loops appear in **Get-Next-Node**, that is called from **Add-To-Trie** and in step 1 of **Insert-To-Sibling-List**. As discussed earlier in section 5.2.1, this part of the algorithm is of constant complexity since it does not depend on the size of the input.

The third type of loop, namely the loop in step 1 of **Process-Info-Node**, is responsible for breaking up a string in an information node and constructing a branch node corresponding to each character in the new string that matches a character in the string in the information node. The maximum number of branch nodes constructed by this loop is of course equivalent to the maximum allowable size of a string in the trie

(more specifically in an information node). Therefore, the worst case complexity of this type of loop is  $O(s)$ , where  $s$  is bounded by the maximum size of the string to be inserted.

The overall worst case complexity of the insertion algorithm can be described as the  $\text{Max}(O(s), O(k))$ . In either case, the complexity is linear.

### 5.3. Space Efficiency of the Algorithm

The representation of the data structure for our character recognition algorithm is the binary form of a trie. The trie (section 3.1.2) supports variable size nodes and requires the linking of two different types of nodes; branch nodes and information nodes.

It would be extremely difficult to figure out an exact upper bound for the amount of storage space needed for a trie. In this paper, however, an attempt is made to find a very rough estimate of the number of binary tree nodes needed for a trie.

Let  $S = \{ s_i : s_i \text{ is a string in the trie } \}$ ,

$B = \{ b_i : b_i \text{ is a binary tree branch node } \}$ , and

$I = \{ i_j : i_j \text{ is a binary tree information node } \}$

for  $0 \leq j \leq |S|$ , then from a simplistic point of view, in the worst case,

$$|B| \leq \sum_{1 \leq j \leq |S|} |s_j|, \quad (5.1)$$

and

$$|I| \leq |S|. \quad (5.2)$$

That means a trie would contain at most one branch node for every character in each string and at most one information node corresponding to each string stored in the trie.

The total number of nodes  $N$  in a trie is therefore

$$N = |I| + |B| \leq |S| + \sum_{1 \leq j \leq |S|} |s_j|. \quad (5.3)$$

We then notice that a branch node exists only when there are at least two strings that share the character in that branch node. We can further realize that an information node exists only if a string does not terminate at a branch node. Considering these two facts, it can be seen that the actual number of nodes contained within a trie is usually much less than the number we arrived at from the simplistic point of view (equation 5.3), or

$$|B| \leq 1/2 \sum_{1 \leq j \leq |S|} |s_j|, \quad (5.4)$$

and

$$|I| \leq |S|. \quad (5.5)$$

In our representation of the trie a branch node is composed of four fields; one for the type of the node, two for child and next-sibling pointers, and one for the character that the branch node represents. If an implementation language is capable of distinguishing nodes of different types then we might not have any need for the *type* field of the branch node. The amount of storage needed by the individual fields of a branch node depends on the implementation environment.

As with a branch node, the amount of storage needed by an information node greatly depends on the implementation environment of the trie structure. If the implementation language allocates storage space for strings dynamically, then there is no waste of space. If a language does not have this feature, then storage space has to be allocated ahead of time to accommodate the largest possible string. This usually leads to great amounts of waste. Of course, in many programming languages, one is able to simulate dynamically allocated string storage space by constructing a string space that

would hold all the strings in one linear structure. This technique does solve the storage problem to some extent, but it does so at the cost of a much more complex program.

The other data structures that are worth analyzing are the three stacks used in the character recognition algorithm (see section 4.3). These are the **str-stack** (see section 4.3) and the **prefix-str-stack** (see section 4.4.1.5) which are used to collect individual characters of a string, and the **ptr-stack** (see section 4.3) which is used to stack the pointers that are used to walk back through the trie. The sizes of **str-stack** and **prefix-str-stack** are directly proportional to the size of the input string. Thus their maximum size is the maximum allowable size of a string in the trie. The **ptr-stack** collects pointers to the nodes previously pointed to by the search pointer as the search pointer moves down a branch of the trie. The farthest that the search pointer could go is to the end of the longest branch in the trie. Therefore, the maximum size of **ptr-stack** is equivalent to the number of levels of the trie.

#### 5.4. Extensions to the Trie Structure

The trie structure that we have discussed thus far is capable of storing strings of characters in an efficient manner. Most often however, a string of characters is not the only information one is after. Strings in a computer system usually have associated with them a command, a value, or an index of some sort to some memory or disk location. Ideally, one would like to be able to access such information using the algorithm of this paper.

In order to accomodate this capability, we can suggest at least two possible extensions to the trie structure. The first is to allow more than one field in an information node. The extra fields would serve as a holding place for the needed additional information. Of course, one would have to make certain that each string in the trie has an information node associated with it. This would require minor modifications to the

algorithms that were developed in this research. Currently, strings whose characters terminate at a branch node are ended with a branch node which contains a terminating symbol. The change would dictate that every string would terminate at an information node.

Another possible extension to our trie structure would be to provide pointers to the additional information for each information node. In other words, some or all of the actual data which was stored in an information node, in the above extension, could be stored outside of the trie in some other data structure. What would be stored in the information nodes would be pointers or indexes to this data structure. Again, this extension would require that there is one information node associated with each string in the trie.

The first extension described above, provides a more practical (more usable) data structure for storing a large number of keywords. With the second extension to the trie structure, a programmer has more flexibility to implement the storage of the information associated with each character string.

### 5.5. An Alternative Tree Representation

From the beginning of the development of the character recognition algorithm, we have assumed a binary tree representation of the trie structure (see section 4.1). That representation was convenient and allowed us to describe the algorithm in much detail. Also, the binary tree representation gives us the advantage of directly implementing the algorithm in most programming languages. However, there is at least one other possible representation.

The alternative is to use a general tree (m-ary tree) with variable length nodes<sup>2</sup>. This is a natural representation for the trie structure. Of course, the disadvantage of this

---

<sup>2</sup>The *length* of a node is the number of fields in that node.



alternative is that most languages do not support variable length nodes, thus the implementation is bound to be either inefficient or not close to the original representation. The advantage of the m-ary tree representation is that it simplifies the algorithm and leaves more implementation flexibility to the programmer.

### 5.6. An Alternative Algorithm

The character recognition algorithm presented in this paper utilizes the trie structure. The researcher proposes an alternative algorithm based on a sequential structure. This algorithm would require that the strings be stored in a sequential and sorted form. Figure 5.2 illustrates such a structure. The algorithm is briefly outlined below.

---

String
ALLOCATE
BEGIN
BY
CALL
CHECK
CHOKER
CLOSE
GOOD
GULL
•
•
•

Figure 5.2  
A sequential structure for storing strings.

---

To perform character recognition on this sequential structure, one would performed a search on each character entered by the user. To be more precise, as characters are entered, a search would try to locate the first string in the sequential structure whose  $i^{\text{th}}$  character is equal to the  $i^{\text{th}}$  character entered. If the search locates the same string that the search pointer was pointing to previously, then a *character pointer* is merely moved down that string by one character position. Otherwise, a *string pointer* is pointed at the new string location and a character pointer is pointed at the matched character within that string.

At least on the surface, it seems that a binary search algorithm could be employed for this purpose, since the strings are stored in a sorted order. For the  $i^{\text{th}}$  character entered, the binary search would only be performed in the range of the strings that have their first  $i-1$  characters equal to the first  $i-1$  characters entered by the user. Of course, one would have to find some method of determining the bounds of the search at each level.

If the user signals a recognition request and the character after the one pointed at by the *character pointer* is equal to the corresponding character of the string following the string pointed at by the *string pointer*, the request for recognition cannot be granted, otherwise, the remaining characters of the string pointed at by the *string pointer* would be displayed (from the character after the one pointed at by *character pointer* to the end of the string). The other character recognition operations discussed in the previous chapters could be performed just as easily.

If we let  $n$  be the maximum number of strings and  $m$  be the size of the largest string in the sequential structure, we can guess at this time that the overall complexity of the character recognition algorithm may be  $O(m \log n)$ , considering the fact that a binary search algorithm is usually  $O(\log n)$ .<sup>3</sup> That means, in the worst case, all of the  $n$  strings

---

<sup>3</sup>An analysis of the binary search algorithm is given in [Purdom & Brown 19885], pages 19-20.

in the structure have to be binary searched for every one of their  $m$  characters. The detailed algorithm and analysis for this sequential structure would be the scope of additional research and will not be presented in this paper.

## Chapter Six

### Summary and Conclusions

#### 6.1. Summary

The scope of this research has been the development of an algorithm for character recognition. The idea for the algorithm developed out of a research project that the researcher was involved in. The project was sponsored by the Intermountain Fire Sciences Laboratory and involved the building of an information system. The information system called for a user-friendly interface. In order to access information in the system, one would most often have to enter a keyword that is usually long and very difficult to remember or spell correctly. To alleviate this problem, the researcher proposed the utilization of the character recognition technique for entry of keywords.

Character recognition provides the user with the ability to enter a very long string by supplying just the initial few characters of that string. The character recognition system would perform a search through a given data structure that houses the available strings to find a unique string that begins with the characters that the user has entered.

Because of its natural adaptability to the problem, the *trie* structure was chosen as the main data structure used in the character recognition algorithm. A trie structure is an  $m$ -ary tree composed of two types of nodes; *branch* nodes and *information* nodes. A formal definition of the trie structure was presented in section 3.1.2. Each branch node of a trie consists of  $m \geq 2$  components corresponding to the elements of the alphabet set.

For the purpose of our application, two modifications were made to the general trie. First, branch nodes were allowed to vary in size; and second, the decision was made not to store a complete string in the corresponding information node unless there is no other string that shares prefix characters with it. This decision results in conservation of

storage space that is well worth the additional complexity it introduces. For reasons of manipulation and implementation simplicity, a binary tree representation of the trie structure was chosen. This means that each branch node is represented as a linked list of binary tree branch nodes where the next sibling is the next binary tree branch node in the list that represents the trie branch node. In this representation, each branch node has two link fields: a *child* field that points to its child, and a *next-sibling* field that points to its next sibling node.

The character recognition algorithm was developed based on the assumption that the operations performed on a trie mainly consist of search and traversal of the trie. Thus, the emphasis was placed on the efficiency of these types of operations as opposed to operations that modify the trie.

The algorithm was presented in a modular fashion, with each module representing an operation that the user might request. The recognition operations for which algorithms were developed are processing of a recognition request, a rubout request, a newline request, and processing of a request to list possible options. The only update (modification) operation presented in this paper is the insertion of a new string into the trie structure. A general algorithm and then a detailed algorithm were presented for each of the above operations on a trie.

The algorithm modules were analyzed for time and space efficiency. The results of the time efficiency analysis were shown in the Big-O notation. In analyzing the character recognition algorithm it was discovered that the time efficiency depends only on the trie structure and is independent of the keyword string entered by the user. The only nonconstant complexity in the character recognition algorithm was found to be in processing of a recognition request and processing of a request to list possible options. In the former case, the worst case complexity is  $O(k)$ , where  $k$  is the number of levels in

the trie; and in the latter case, the worst case complexity is  $O(m)$ , where  $m$  is the number of nodes in the trie to be traversed.

The insertion algorithm was discovered to have an overall worst case complexity of  $\text{Max}(O(k), O(s))$ , where  $k$  is the number of levels in the trie, and  $s$  is the size of the string to be inserted into the trie. In either case the complexity is linear with respect to the input.

As for space efficiency of the algorithm, some approximations of measurements were presented in the paper. An upper bound for the maximum storage requirements for the trie structure is one branch node for every character in each string, and one information node for each string stored in the trie. An important point to consider is that the storage efficiency of information nodes that contain strings depends greatly on the implementation environment.

The only other structures that needed to be analyzed for space efficiency were the three stacks **str-stack**, **prefix-stack**, and **ptr-stack**, used to perform character recognition. The maximum needed size of the first two was discovered to be equal to the maximum allowable size of a string in the trie, and for the **ptr-stack** it was equal to the number of levels in the trie.

Two extensions to the trie structure have been proposed to facilitate storage of other relevant information along with a string. The first is to allow more than one field in an information node and the second extension is to provide pointers to information spaces in or in place of each information node. These extensions would require modifications to the algorithm to make sure that each string has an information node associated with it.

As mentioned before, we have used a binary tree representation of the trie structure for the purpose of our algorithm. There is however at least one other possible representation. The alternative is to use a general tree ( $m$ -ary tree) with variable length

nodes. This representation results in a less detailed algorithm than the binary tree representation.

As a final note, we presented an idea for a different algorithm for character recognition. This idea is based on a sequential structure for storing strings. The development and analysis of a detailed algorithm on this structure would probably constitute another research project.

## 6.2. Conclusions

As it was mentioned early in this paper, the main goal of this research was to develop an algorithm for character recognition that would be efficient for search and traversal operations but not necessarily for modification operations. This goal has been achieved to a great extent. The recognition algorithm developed has a time efficiency of  $O(\text{number of levels in trie})$  for handling recognition requests,  $O(\text{number of nodes in the trie})$  for processing of requests for listing possible options, and a constant running time for all other recognition operations. In the case of the first two operations, the complexity is linear or  $O(N)$ . Of course this is the worst case complexity, which means if we have a large number of long strings that have many common characters, then the search for recognition will be close to  $O(N)$ , otherwise it will be better than  $O(N)$ .

As far as storage efficiency of the algorithm is concerned, the trie representation used for the purpose of our character recognition algorithm promotes the conservation of much storage space depending on the implementation environment. If the algorithm is implemented in a language that dynamically allocates storage space for strings, then the fact that the complete strings need not be stored in their corresponding information fields will save storage space; otherwise if the implementation language pre-allocates space for strings, then our modification to the trie saves no space whatever.

The correctness of the algorithm was tested by applying it to many different examples covering all known cases. This was done both by hand and through implementation of the algorithm on the computer. The implementation further showed the practicality of the algorithm, in that the processing of a recognition request was almost instantly performed even with a large number of strings.

The algorithms developed in this research provide an efficient and practical means of performing character recognition that is suitable for applications that do not have a very dynamic set of strings or keywords. That is to say, the best area of application for our algorithm is in an environment that requires frequent entry of predetermined keywords.

As a final note, it should be mentioned that a comparison between the character recognition algorithm developed in this research and other previous algorithms would have been desirable. But unfortunately, no other algorithms were available at the time of the research. However, we have proposed a character recognition algorithm based on a sequential structure. The complexity of that algorithm was estimated to be  $O(m \log n)$ . It can be seen that the advantage of the trie character recognition algorithm over the sequential algorithm is that the trie algorithm is independent of the number of strings stored in the data structure, whereas, in the sequential case the complexity is directly influenced by the number of the strings stored in the structure. That is of course if we do not count the traversal of the trie as a part of the recognition process.



## BIBLIOGRAPHY

- Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass., 1985.
- Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Data Structures and Algorithms*. Addison-Wesley Pub. Co., Reading, Mass., 1983.
- Baron, R. J., Shapiro, L. G. *Data Structures and Their Implementation*. Van Nostrand Reinhold, New York, NY, 1980.
- Cooper, D. *Standard Pascal User Reference Manual*. W.W.Norton & Co., New York, NY, 1983.
- Gries, David. *The Science of Programming*. Springer-Verlag, New York, NY, 1981.
- Horowitz, E., Sahni, S. *Fundamentals of Data Structures*. Computer Science Press, Rockville, MD, 1983.
- Knuth, D. E. *The Art of Computer Programming*, 2nd ed. Addison-Wesley Pub. Co. Reading, Mass., 1973.
- Lewis, H. R., and Papadimitriou, C. H. *Elements of The Theory of Computation*. Printice-Hall, Inc., Englewood Cliffs, NJ, 1981.
- Manna, Z., Waldinger, R. *The Logical Basis for Computer Programming*. Addison-Wesley Pub. Com., Reading, Mass., 1985.
- Purdom, P. W. Jr., Brown, C. A. *The Analysis of Algorithms*. Holt, Rinehart and Winston, New York, NY, 1985.
- Tremblay, J., Sorenson, P. *An Introduction to Data Structures With Applications*. McGraw-Hill, Inc., New York, NY, 1984.