STORAGE AND ACCESS COSTS

FOR IMPLEMENTATIONS OF VARIABLE - LENGTH LISTS

by

Donna Jean Brown

April 1979

This research was supported in part by the Army Research Office under contract no. DAAG29-77-C-0012.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

This empty page was substituted for a blank page in the original document.

STORAGE AND ACCESS COSTS

FOR IMPLEMENTATIONS OF VARIABLE - LENGTH LISTS

by

Donna Jean Brown

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 1978 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

ABSTRACT

Consider a machine with a cellular memory used to store a list X^i, where X is a finite alphabet and $i \in \mathbb{N}$. We investigate the machine representation of such a list and the implementation of common list operations such as determining the i^{th} element and adding or deleting an element. Information-theoretic arguments are used in order to obtain lower bounds on storage and access costs for implementing variable-length lists and, in particular, stacks. Representations are discussed which attain these bounds separately and can sometimes attain both, although it is shown that some common representations for stacks cannot simultaneously achieve both. On the constructive side, we show that it is possible to implement a stack of any finite length so as to achieve Kraft storage and so that the number of memory cell accesses required to perform a PUSH or a TOP operation is always $O(\log n)$ but where, assuming a nonincreasing probability distribution on stack lengths, a POP operation requires on the average only a constant number of accesses.

THESIS SUPERVISOR: Peter Elias
TITLE: Edwin S. Webster Professor of Electrical Engineering

Keywords: list, stack, queue, pointer, implementation, storage cost, access cost, Kraft inequality

This empty page was substituted for a blank page in the original document.

ACKNOWLEDGEMENTS

I would like to express my gratitude

To my thesis advisor, Peter Elias, who inspired this research and whose patience and guidance made this work possible.

To Frederick C. Hennie, III, for his interest and help throughout my graduate student career, including a careful reading of this thesis during its final preparation.

To Ronald L. Rivest, whose questions and insights have presented me with many valuable research ideas.

To Richard F. Martin, Jr., for constant friendship and encouragement and especially for the many tireless hours he spent making this manuscript readable.

To Michael L. Van De Vanter for assistance with the computing facilities used in producing this document.

To my family for their love and encouragement always.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

With the present-day widespread use of computers, it is important to be able to efficiently store information and execute operations. For a given problem, depending on the structural relationships between the data elements, we choose to use a particular type of data structure. In this thesis, we shall consider only the simplest information structure, a list; in particular, we discuss stacks and briefly mention some work with queues.

1.1 The Data Model

The data model I will use for studying list structures is based on the model of a storage and retrieval problem developed by Elias [5] and Welch [23]. A retrieval problem consists of a collection of data bases, any one of which may be observed at a given time, and a set of retrieval questions which may be asked of any data base. It may also be desired to perform updates; i.e., to transform the currently observed data base into some other data base from the domain, the set of possible data bases.

A retrieval system which solves a retrieval problem must have several components:

(1) a method of representing any observed data base,

(2) a method for answering any retrieval question about the observed data base,

(3) a method for performing updates on the observed data base.

For a given question, the method for answering the question must be independent of the observed data base; to allow the method to depend on the observed data base would presuppose some knowledge of the observed data base by the user in order to determine which method is appropriate. Thus, the method must give the correct answer no matter what the current data base is.

The following example illustrates what we mean by a storage and retrieval problem. We delay discussion of how a data base might be stored and how a query or update might be implemented until the next section, where we shall reconsider this example.

Example 1.1. Consider the problem of Rotary Fan Manufacturing Co., R.F.M., receiving mail orders for fans. Somehow R.F.M. must keep track of these orders to be filled. Exactly what information is needed depends on the questions and updates that will be executed. A data base corresponds to the current list of orders to be filled. The domain is the set of all possible data bases; i.e., all possible lists of fan orders. Notice that there are data bases of different sizes; in fact, it may be possible for a data base to have any integral size greater than or equal to zero. Of course, if R.F.M. wants to stay in business for long it had better be the case that shorter data bases are more probable than larger ones.

Because old orders are continually being filled and new orders received, it must be possible to update the current order list; in particular, R.F.M. needs to be able to perform the following two updates.

u_1: Process an order from the order list. This involves mailing the desired fans and deleting the order from the current list. Thus, the size of the data base is decremented by one.

u_2: A new order arriving must be placed on the order list, which results in the size of the current data base being incremented by one.

R.F.M. must also be prepared to answer queries concerning the current data base, such as whether or not John Doe's order is on the list, or whose order will be filled next. For instance, we might have the following set of questions.

q_1: Is ⟨name⟩ a customer waiting to have his order processed?

q_2: Who is the k^{th} customer in line; i.e., what order will be the k^{th} to be served?

q_3: What are the i most recently placed orders?

Exactly what information needs to be stored on the order list depends on the particular queries that will be made. ∎

1.2 List Structures

We consider a list problem to be a type of storage and retrieval problem, where each data base is a particular list. In general the size of the list may vary, and exactly how the list will be implemented depends on the specific questions and updates to be performed. In this section we introduce the basic list structures we will be concerned with in this thesis: stacks, queues, and dequeues. The appropriate operations will be formally defined later.

A linear *list* is just an ordered sequence of items chosen from a particular set of elements (see e.g. Knuth [14], Aho, Hopcroft, and Ullman [1]). In many instances, accessing of a list is restricted to the first and last elements; in particular, it may be the case that items can be added or deleted only at the ends of the list. Because these lists are frequently encountered, they have special names: stacks, queues, dequeues.

A *stack*, also known as a push-down store or a LIFO (last-in/first-out) list, is a linear list for which all insertions and deletions are made at one end of the list, the *top*. For example, consider an initially empty stack; i.e., there are no elements in the list. Suppose we then insert two elements onto the stack:

Element 1, Element 2.

Since Element 1 was the first item put onto the stack, it occupies the *bottom* stack position and is the least accessible item; it cannot be removed until all other elements on the stack have been removed. To add, PUSH, a third element onto the stack, we locate the top of the stack and insert this new element, Element 3:

Element 1, Element 2, Element 3.

Element 3 is now at the top of the stack, and so if we delete, POP, an element from the stack, we are left with:

Element 1, Element 2.

Of course, if the stack had been empty we would not have been able to perform a POP operation, so there must be some way of detecting an empty stack.

Exactly how one might choose to implement a stack is one of the issues discussed in this thesis. Figure 1.1 should help picture how the stack operations work and corresponds to one common type of implementation, where each item in the stack has a pointer which indicates the location of the previous stack item. An additional pointer always points to the top of the stack. Such a storage arrangement allows the stack operations to be performed in a straightforward way. In particular, a TOP operation is performed by reading the pointer in order to locate the top of the stack and then simply reading what the TOP value is. To perform a POP we locate the top of the stack, use this element to locate the second stack element, and then reset the top of stack pointer to this second element, which becomes the TOP element. Similarly, a PUSH operation can be implemented by first locating some free memory cell, into which the appropriate new stack value is inserted. This new cell has a pointer which is set to the same location as the top of stack pointer, and then the top of stack pointer is changed so that it points to the newly filled cell, our new top of stack. The pointers involved in these implementations are indicated in Figure 1.1. Notice that the directions of the pointers between the stack elements make reading "down" the stack straightforward, but there would be no way to read back "up" the stack. Of course, if the stack occupied a contiguous section of memory, there would be no need at all for pointers between the stack elements.

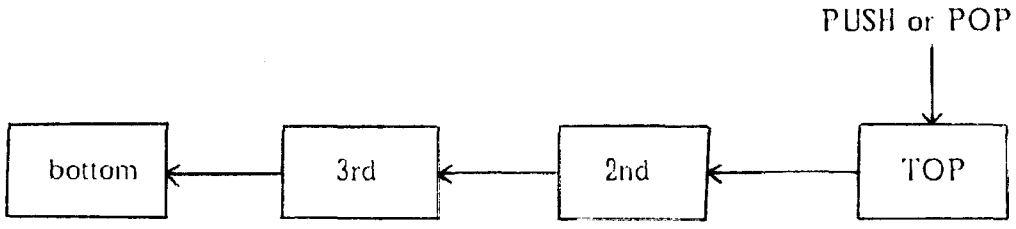


Figure 1.1. Stack Operations

A *queue*, also known as a FIFO (first-in/first-out) list, or a circular list, is a linear list for which all insertions are made at one end of the list, the rear, and all deletions are made at the other, the front. Thus, elements leave the list in the same order in which they entered. Suppose we insert, ENQUEUE, three elements onto an initially empty queue, first element 1, then element 2, then element 3:

Element 3, Element 2, Element 1.

If we now delete, DEQUEUE, one element, we are left with:

Element 3, Element 2.

Figure 1.2 illustrates the queue operations. Notice that if the arrows between elements in Figure 1.2 were reversed, then after performing a DEQUEUE operation we would have no way to keep track of the location of the front of the queue. Of course, we might choose to store pointers going in both directions, but this would involve greater storage costs.

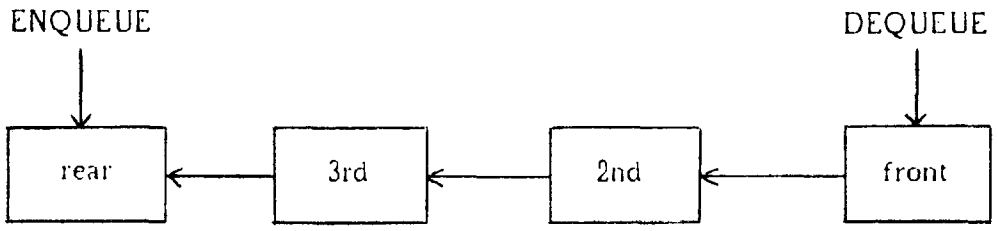


Figure 1.2. Queue Operations

A *dequeue* is a linear list for which all insertions and deletions are made at the ends of the list. Thus, a stack and a queue can each be viewed as a particular type of dequeue. One may also distinguish *output-restricted* or *input-restricted* dequeues, in which deletions or insertions, respectively, are allowed to take place at only one end. The ends are commonly referred to as left and right, although either an insertion or a deletion may occur at either end (see Figure 1.3). We shall not in this report discuss any results specifically concerning dequeues, but it appears that a

dequeue can be viewed as a straightforward extension of a queue.

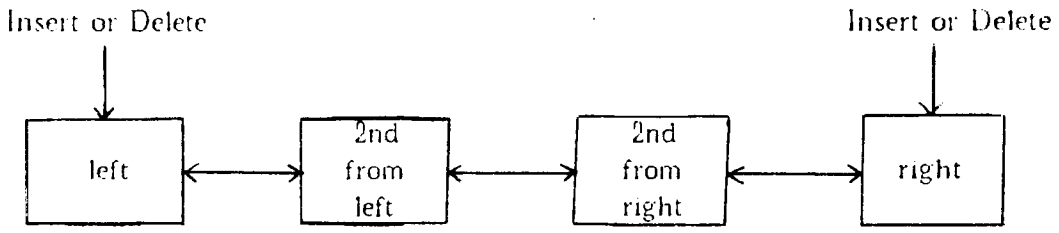


Figure 1.3. Dequeue Operations

Now that we have discussed these simple list structures, let us reconsider the issue of developing a solution to the system of Example 1.1.

Example 1.2. How R.F.M. develops a system to solve its order problem depends not only on finding an efficient means to store any data base, but also on what queries and updates it expects to be making most often. Thus, finding an "optimal" solution would depend on knowing some rather precise probabilities. On the other hand, we can at least make some general comments. The representation of a data base must include the names of the persons who ordered fans, as well as the other necessary information such as quantity ordered, address, payment, etc. It would probably make sense to store a data base as some sort of list structure. For simplicity, let us consider only a list of names and assume that each name also contains a pointer to the relevant corresponding information. In other words, we access any element in the list by reading the appropriate name. We have decided that each data base is to be represented by a list structure, but the type would be determined by R.F.M.'s desired processing order. Let us discuss several possible implementations.

One reasonable scheme would be to process orders FIFO; i.e., in the same order in which they arrived. This would correspond to implementing some sort of queue, perhaps as in Figure 1.2. In this case we always keep track of the next

order to be processed and the last order received. Presumably, updates u_1 and u_2 would be easy to perform. On the other hand, returning the answer to question q_1 requires searching the queue for a particular name. Unless we have more information, this could require searching through the entire list. For a queue implementation, it would probably be straightforward to answer question q_2, by tracing backward k items from the front. On the other hand, q_3 would probably be difficult to answer. To determine the one most recently placed order would require only a single access to the rear of the queue. But to determine the second most recently placed order is not as easy. Unless there is some way of knowing the "reverse pointers", then it would be necessary to read all items from the front, keeping track of each previous item read, until we reach the rear of the queue. Of course, if we expected q_3 to be asked frequently, we might wish to alter our implementation scheme and store both forward and reverse pointers. At the price of increased storage, we could decrease the expense of answering q_3.

Another possible scheme would be to try to process orders as they are received, using a stack representation. Of course, R.F.M. Co. might lose a lot of business this way, because if it gets at all behind in processing orders, then some poor souls would be stuck indefinitely at the bottom of the stack. (And R.F.M. hasn't even considered the issue of cancelling an order from the middle of the list!) With such a FILO implementation, we would expect q_3 to be easier to answer than it was with a queue implementation, but now q_2 doesn't even make sense, because there is no way to know when an order will be processed. Question q_1 would probably be no more or less difficult than it was for the queue.

If we expected to spend most of our time answering question q_1, we might want to sort the list of names alphabetically. (This would also make it easier to cancel an order.) But then we would need some additional means of indicating the processing order, such as a number field associated with the name. Unless we want to mail out the fans according to some alphabetical order, we would either need pointers to indicate the processing order or else updates might be very expensive. ∎

1.3 Computer-Implemented List Problems

In this thesis we are concerned with computer-implemented solutions of list problems. Recall that in Section 1.1 we mentioned the three components that any such system must possess. Note that requiring the algorithmic method for answering a question (or performing an update) be independent of the observed data base implies a strict separation of "program" and "data". The "program" to answer a question must remain constant, while presumably the computer memory state (representing the observed "data") differs for different observed data bases.

A computing system which finds the values of a function $f:D_f \rightarrow R_f$ can be viewed information-theoretically as a deterministic communications channel with input $d \in D_f$ and output value $f(d) \in R_f$. In [6], Elias considered the strictly informational limits on computer performance and obtained lower bounds on storage and access required in the computation of a single function. This was done by allowing freedom of choice of representation of the input and decoding of the output. Viewing the contents of a computer's memory as a codeword, Elias [7] dealt with questions about the use of codewords which are not sequences but are sets of bits at addresses scattered throughout a shared memory. The next step was to extend these results to the computation of a family of functions defined on a common domain. An overview of much of this work is given by Elias [9], and an analysis of the complexity of some simple retrieval problems with update was given by Elias and Flower [10]. Warner [22] has investigated the performance of retrieval systems for tables of entries.

Let us note that information-theoretic approaches have been taken to other problems as well. The work of Kolmogorov [15] using minimal program length as a measure of computational complexity has an informational flavor. Also Chaiten [4] viewed the contents of memory as a program to be executed. Other work has been done relating to problems of exact and partial match and their storage and access costs (Minsky and Papert [19], Rivest [20], [21]).

This thesis extends work that Elias has done, in which he has considered many issues concerned with storage and retrieval problems using a fixed size linear array. To allow the natural representation and manipulation of data, variable size arrays such as stacks, queues, dequeues, lists, and trees are frequently used. The fact that they have variable size makes different storage representations and accessing techniques appropriate; for instance, we must consider the basic operations of insertion of new elements and deletion of existing elements.

We are interested in investigating certain costs associated with solving computer-implemented list problems. In particular, we are concerned with lower bounds on the cost of storing a data base and on the cost of implementing a question or an update on the currently observed data base. The storage cost we measure in terms of the number of memory cells required for the data base representation. The implementation cost we measure in terms of the number of memory accesses required, which is in general directly related to the time taken to perform an operation.

We begin by in Chapter 2 discussing the formalism of our machine model and what it means to solve a list problem. Chapter 3 discusses storage and access costs and explains the notions of Kraft storage and access, indicating the types of cost bounds we might expect to obtain. In Chapter 4 we consider the entire set of table lookup questions and investigate consequences of achieving Kraft storage and access. Possible implementations for the table lookup question set are explored in Chapter 5, where we discuss three types of representations: fixed length, endmarker, and pointer. These same representation classes are analyzed in Chapter 6 with respect to implementing stacks. Finally, we summarize our results, discuss how the techniques we have developed can also be used to help obtain storage and access bounds for queues and dequeues, and point out directions for future work.

CHAPTER 2

SOLUTION OF A LIST PROBLEM

In this chapter we discuss our formal machine model and what it means to solve a list problem. This work is based on the model of a storage and retrieval problem developed by Elias [5], [6], [8]. We shall here introduce much of the terminology and notation that is used throughout the thesis. We first define a storage and retrieval problem, and then define our machine model and what it means for a machine to answer a question correctly. We discuss the distinction between the problem and machine domains and then define the machine representation of a problem domain. At this point we are finally in a position to state precisely what it means for a machine to solve a storage and retrieval problem. In the last section we summarize some of the ideas presented in the chapter, in order to clarify what we mean by the solution of a list problem.

2.1 Definition of a Storage and Retrieval Problem

Let F be a family of functions (operations) defined on a common domain $\mathbb{D}$, and indexed by some index set $J \subseteq \mathbb{N}$, $F = \{f_i \mid i \in J\}$. An operation $f_i \in F$ is an ordered pair of functions $f_i = (q_i, u_i)$, where $\mathrm{dom}(f_i) = \mathbb{D}$ and $\mathrm{ran}(u_i) \subseteq \mathbb{D}$. We refer to an element $d \in \mathbb{D}$ as a data base. Executing operation f_i on data base $d \in \mathbb{D}$ returns the value $q_i(d)$ and has the side effect of updating d to the new value $u_i(d)$; we denote this by $f_i(d) = (q_i(d), u_i(d))$. $Q = \{q_i \mid (q_i, u_i) \in F\}$ is called the question set and $U = \{u_i \mid (q_i, u_i) \in F\}$ is called the update set of F. We refer to $(F, \mathbb{D})$ as a *storage and retrieval problem.* If the data base d is not changed as a consequence of executing f_i (i.e., if $u_i(d) = d$), then $(F, \mathbb{D})$ is said to be a *static problem*, and we may write it as $(Q, \mathbb{D})$. In general, however, the data base may change with time, in which case $(F, \mathbb{D})$ is a *dynamic problem*, or a *problem with update.*

In this thesis, we shall consider storage and retrieval problems which represent list data structures; we refer to these as *list problems*, or simply *problems*. In Section 2.7 we will be in a better position to explain precisely what we have in mind when we discuss the solution of a list problem. Let us begin by presenting a simple example of a storage and retrieval problem, which will illustrate some of the above terminology. Examples 2.2, 2.3, and 2.7 are extensions of this example.

Example 2.1. Let $\mathbb{D} = \{d_i \mid 0 \leq i \leq 6\}$ where each $d_i \in \mathbb{D}$ is a string of symbols from the set $X = \{0,1\}$; i.e., each $d_i \in X^*$:

$$d_0 = \lambda \qquad d_4 = 01$$
$$d_1 = 0 \qquad d_5 = 10$$
$$d_2 = 1 \qquad d_6 = 11$$
$$d_3 = 00$$

Note that we write $d_0 = \lambda$ to indicate that d_0 is the null string, the string with no elements. Now consider two operations on $\mathbb{D}$, f_1 and f_2. The function $f_1 = (q_1,u_1)$ is simply the identity question and update:

$$q_1(d_i) = d_i$$
$$u_1(d_i) = d_i$$

Since u_1 causes no change to the data base d_i, f_1 effectively has only a question component and so is a static operation. We define f_2, however, to be a dynamic operation: $f_2 = (q_2,u_2)$, where

$$q_2(d_0) = a_0 \qquad u_2(d_0) = d_0$$
$$q_2(d_1) = a_1 \qquad u_2(d_1) = d_0$$
$$q_2(d_2) = a_1a_1 \qquad u_2(d_2) = d_1$$
$$q_2(d_3) = a_1a_0a_0 \qquad u_2(d_3) = d_1$$
$$q_2(d_4) = a_1a_2a_1 \qquad u_2(d_4) = d_2$$
$$q_2(d_5) = a_2a_2a_1 \qquad u_2(d_5) = d_2$$
$$q_2(d_6) = a_1a_1a_0a_0 \qquad u_2(d_6) = d_3$$

Thus, executing the operation f_2 on data base d_3 gives the answer $a_1a_0a_0$ and changes the current data base, d_3, to the data base d_1. Notice that $\mathrm{dom}(f_1) = \mathrm{dom}(f_2) = \mathbb{D}$, $\mathrm{ran}(u_1) = \mathbb{D}$, and $\mathrm{ran}(u_2) = \{d_0, d_1, d_2, d_3\} \subseteq \mathbb{D}$.

So if we were to execute the sequence of operations f_2,f_1,f_2,f_2 on d_3, then we would expect the sequence of answers to be $a_1a_0a_0$, d_1, a_1, a_0 and the resulting data base to be d_0. ∎

We frequently denote the domain of a function, dom(f), by D_f and similarly ran(f) by R_f. Where we have a set $F = \{f_i | i \in \mathcal{I}\}$ of operations, we may find it convenient to write D_i and R_i for D_{f_i} and R_{f_i}, respectively. If there is no possibility of confusion, we may simply omit the subscripts and write D and R. For instance, $D(S)$ denotes the domain of the set S. Note that when we discuss a problem $(F, \mathbb{D})$, we write $\mathbb{D}$ to refer to the problem domain, which happens to be the common domain of each function $f \in F$.

2.2 Definition of the Machine Model

Our machine model is a deterministic, sequential, random access cell-addressable, halting automaton $\mathfrak{M}$, with a memory m consisting of L cells (where L may be infinite). The set of all possible contents of a memory cell, $\mathcal{B}$, corresponds to $\mathfrak{M}$'s finite input alphabet, and $\mathcal{B}^L$ denotes the set of possible memory states. Via its memory, $\mathfrak{M}$ stores a sequence $b \in \mathcal{B}^L$, which it reads in some order determined by the structure of $\mathfrak{M}$ and the values in b. $\mathfrak{M}$ may or may not rewrite values as it reads the cells, but it eventually prints a sequence of output symbols chosen from some finite output alphabet $\mathcal{E}$. Since $\mathfrak{M}$ is deterministic, a given input (initial state of memory) always causes $\mathfrak{M}$ to print the same output (if $\mathfrak{M}$ halts), so $\mathfrak{M}$ computes a partial function ω from inputs in $\mathcal{B}^L$ to outputs in $\mathcal{E}^*$. If we let $\mathfrak{D}(\mathfrak{M}) \subseteq \mathcal{B}^L$ be the set of inputs for which $\mathfrak{M}$ halts in finite time and $\mathfrak{R}(\mathfrak{M}) \subseteq \mathcal{E}^*$ be the set of outputs which $\mathfrak{M}$ prints before it halts, then each automaton $\mathfrak{M}$ defines a "characteristic function" $\omega:\mathfrak{D}(\mathfrak{M}) \rightarrow \mathfrak{R}(\mathfrak{M})$. The only functions which $\mathfrak{M}$ can actually compute are restrictions of its characteristic function to some subset of its acceptance set.

2.3 Machine Computation of a Static Function

Now that we have in mind a machine definition, let us investigate in what sense a machine $\mathfrak{M}$ with L memory cells can compute a static function $q:D_q \to R_q$. Technically, a machine $\mathfrak{M}$ can compute the values of a question $q:D_q \to R_q$ only when $D_q \subseteq \mathfrak{D}(\mathfrak{M})$ and $R_q \subseteq \mathfrak{R}(\mathfrak{M})$. It is often claimed, however, that a machine $\mathfrak{M}$ computes a function q even when the machine alphabets and the problem alphabets are not identical. In such a case, the user also has in mind two non-machine components: a coder and a decoder. The coder consists of some encoding relation $\tau:D_q \to \mathfrak{D}_q$, from the domain of q onto a subset $\mathfrak{D}_q \subseteq \mathfrak{D}(\mathfrak{M})$; each $d \in D_q$ is taken into a subset $\tau(d) \subseteq \mathcal{B}^L$, and any string $b \in \tau(d)$ is said to "represent" d. (We shall later use the symbol ρ to stand for an encoding function, as explained in sections 2.5 and 2.6. Using that terminology, our encoding relation τ will be seen to correspond to a relation $\bar{\rho}$.) The decoding function $\delta:\mathfrak{R}_q \to R_q$ maps the subset $\mathfrak{R}_q = \omega(\mathfrak{D}_q) \subseteq \mathfrak{D}(\mathfrak{M})$ onto the range of q. The machine is said to *compute* q correctly if, for any $d \in D_q$, when any $b \in \tau(d)$ is supplied to $\mathfrak{M}$ and gives output $e = \omega(b) = \omega \circ \tau(d)$, the decoding $\delta(e)$ of e satisfies

$$q(d) = \delta(e) = \delta \circ \omega \circ \tau(d), \quad d \in D.$$

In particular, $\omega \circ \tau$ must be a function. These conditions are summarized in the following diagram, where all arrows denote total and onto functions or relations:

$$\begin{array}{ccccc} & & \tau & & \\ D_q & \to & \mathfrak{D}_q & \subseteq & \mathfrak{D}(\mathfrak{M}) \subseteq \mathcal{B}^L \\ q \downarrow & & \downarrow \text{ restriction of } \omega & & \downarrow \omega \\ & & \delta & & \\ R_q & \leftarrow & \mathfrak{R}_q & \subseteq & \mathfrak{R}(\mathfrak{M}) \subseteq \mathcal{E}^* \end{array}$$

To help us understand all of this terminology, we consider the computation of question q_2 from the previous example.

Example 2.2. Recall the question q_2 from Example 2.1, where $\mathbb{D}_{q_2} = \{d_i \mid 0 \le i \le 6\}$, $R_{q_2} \subseteq \{a_0,a_1,a_2\}^*$. Let $\mathfrak{M}$ be a deterministic, sequential halting automaton with a memory m consisting of three cells. Let $\mathcal{B} = \{0,1,2,\Diamond\}$, $\mathcal{E} = \{0,1,2\}$. $\mathfrak{M}$ operates as follows: it reads the string of inputs until it encounters a $\Diamond$, reading in order memory cell 0, then cell 1, then cell 2; it interprets the string of characters from $\{0,1,2\}$ as the ternary representation of a natural number; $\mathfrak{M}$ computes, also in ternary, the square of this number, prints it, and then halts. So

$$\mathfrak{D}(\mathfrak{M}) = \bigcup_{i=1}^{2} \{0,1,2\}^i\Diamond\{0,1,2,\Diamond\}^{2-i}$$

$$= \{0\Diamond\Diamond, 0\Diamond1, 0\Diamond2, 0\Diamond\Diamond, 1\Diamond\Diamond, 1\Diamond1, 1\Diamond2, 1\Diamond\Diamond, 2\Diamond\Diamond,$$
$$2\Diamond1, 2\Diamond2, 2\Diamond\Diamond, 1\Diamond\Diamond, 11\Diamond, 12\Diamond, 2\Diamond\Diamond, 21\Diamond, 22\Diamond\}$$

$$\mathfrak{R}(\mathfrak{M}) = \{0, 1, 11, 100, 121, 221, 1100, 1211, 2101\}.$$

$\mathfrak{M}$ computes q_2 correctly, if we choose our encoding and decoding relations appropriately. Let $\tau: D_q \to \mathcal{B}^3$ be defined as follows:

$\tau(d_0) = \{0\Diamond\Diamond, 0\Diamond1, 0\Diamond2, 0\Diamond\Diamond\}$ $\quad$ $\tau(d_4) = \{11\Diamond\}$

$\tau(d_1) = \{1\Diamond\Diamond, 1\Diamond1, 1\Diamond2, 1\Diamond\}$ $\quad$ $\tau(d_5) = \{12\Diamond\}$

$\tau(d_2) = \{2\Diamond\Diamond, 2\Diamond1, 2\Diamond2, 2\Diamond\Diamond\}$ $\quad$ $\tau(d_6) = \{20\Diamond\}$

$\tau(d_3) = \{10\Diamond\}$

Thus, $\mathfrak{D}_{q_2} = \mathfrak{D}(\mathfrak{M}) - \{21\Diamond, 22\Diamond\}$ and $\mathfrak{R}_{q_2} = \mathfrak{R}(\mathfrak{M}) - \{1211, 2101\}$. Now define $\delta: \mathfrak{R}_{q_2} \to R_{q_2}$ by

$\delta(0) = a_0$ $\quad$ $\delta(121) = a_1a_2a_1$

$\delta(1) = a_1$ $\quad$ $\delta(221) = a_2a_2a_1$

$\delta(11) = a_1a_1$ $\quad$ $\delta(1100) = a_1a_1a_0a_0$

$\delta(100) = a_1a_0a_0$

So the machine $\mathfrak{M}$ with encoding τ and decoding δ computes q_2 correctly. For instance,

$$\delta \circ \omega \circ \tau(d_0) = \delta \circ \omega(0\Diamond\Diamond) = \delta(0) = a_0 = q_2(d_0)$$

$$\delta \circ \omega \circ \tau(d_6) = \delta \circ \omega(20\Diamond) = \delta(1100) = a_1a_1a_0a_0 = q_2(d_6)$$ ∎

2.4 The Problem Domain

Because in this thesis we are concerned with representing list structures, we consider a data base $d \in \mathbb{D}$ to be a string of characters chosen from the *problem alphabet* X. For notational convenience, we formally represent d as a set of $|d|$ ordered pairs, containing one value $d(n)$ from the alphabet X for each $n \in \mathbb{N}$ less than $|d|$:

$$d = \{(n,d(n)) \mid 0 \leq n < |d|,\ d(n) \in X\}.$$

When there is no chance of ambiguity, we may write $d = x_1x_2x_1x_3$ to stand for

$$d = \{(0,x_1), (1,x_2), (2,x_1), (3,x_3)\},$$

where each $x_i \in X$. Thus, what the formal ordered pair notation does is to explicitly state the implied order of characters in the string d. In an obvious way, the definition of d could be extended to include countably infinite strings; i.e., we may wish to consider the size of a data base $d \in \mathbb{D}$ to be unbounded. In this thesis, we shall consider only problem domains $\mathbb{D}$ where for all $d_1, d_2 \in X^k$, $d_1 \in \mathbb{D}$ if and only if $d_2 \in \mathbb{D}$. Thus, if we allow a string $d_1 \in X^k$ to be in the domain $\mathbb{D}$, then all strings in X^k are included in $\mathbb{D}$. Certainly there might be instances where we would want to restrict character sequences, but unless we consider specific applications it would be difficult to characterize the domain. Therefore, we consider only *problem domains* $\mathbb{D}$ of the form $\mathbb{D} = \bigcup_{i \in J} X^i$, for some $J \subseteq \mathbb{N}$.

Example 2.3. In Example 2.1, the problem domain consists of seven data bases, $\mathbb{D} = \{d_i \mid 0 \leq i \leq 6\}$. The problem alphabet is $X = \{0,1\}$, and each $d_i \in X^*$. In particular,

$$\mathbb{D} = \bigcup_{i \in \{0,1,2\}} X^i = \{\lambda\} \cup X \cup X^2.$$

The data base d_4, for example, is the string $01 \in X^2$, which can be formally written as $\{(0,0), (1,1)\}$. Similarly, we can denote each $d_i \in \mathbb{D}$:

$d_0 = \{\ \} = \lambda$ $\qquad$ $d_4 = \{(0,0), (1,1)\} = 01$
$d_1 = \{(0,0)\} = 0$ $\qquad$ $d_5 = \{(0,1), (1,0)\} = 10$
$d_2 = \{(0,1)\} = 1$ $\qquad$ $d_6 = \{(0,1), (1,1)\} = 11$
$d_3 = \{(0,0), (1,0)\} = 00$

Notice that the data base d_0 is just the empty string, λ. When we view d_0 as being represented by a set of ordered pairs, then $d_0 = \{\ \} = \varnothing$. Thus, we might either say that $d_0 = \lambda$ or that $d_0 = \varnothing$, depending on our viewpoint at the moment. ∎

2.5 Machine Representation of the Problem Domain

As we have observed, a data base itself cannot be stored in memory. Instead, we store some encoding of the data base, a string of values from the alphabet $\mathcal{B}$. Each $d \in \mathbb{D}$ is mapped by τ into some subset of $\mathcal{B}^L$. It is unnecessarily restrictive, however, to require that an encoding τ specify values for every memory cell. In fact, most computer systems allocate only certain sections of memory to a given user, and other users may write in the remaining cells of memory in ways unknown to the first user. In order to model practical memory allocation schemes such as linked lists (recall Section 1.2), it is necessary to allow an encoding to specify values for only some of the memory cells.

Thus, we view $\tau(d)$ as some set of codewords, a subset of the code $C = \tau(\mathbb{D})$ (see Elias [8]). Each codeword $\mathbf{c} \in C$ is itself a finite set

$$\mathbf{c} = \{(j, \mathbf{c}(j)) \mid j \in D(\mathbf{c})\}$$

of $|\mathbf{c}|$ ordered pairs. The first coordinate of each pair $(j, \mathbf{c}(j))$ is the integer address $j \in \mathbb{N}$ of a cell in memory, and the second coordinate is the value $\mathbf{c}(j) \in \mathcal{B}$ assigned by $\mathbf{c}$ to be stored at that address. Thus, each codeword in C is a partial function $\mathbf{c}:\mathbb{N} \to \mathcal{B}$ from integer addresses to values in $\mathcal{B}$; its domain, $D(\mathbf{c})$, is a finite subset of $\mathbb{N}$.

We denote by $\mathcal{B}^\dagger$ the class of all such partial functions from $\mathbb{N}$ to $\mathcal{B}$ that are each defined on a finite domain. Thus, a codeword set C is just a subset $C \subset \mathcal{B}^\dagger$. The domain $D(C)$ of a set $C \subset \mathcal{B}^\dagger$ is the union of the domains of its members:

$$D(C) = \bigcup_{\mathbf{c} \in C} D(\mathbf{c}).$$

Example 2.4. Let $\mathcal{B} = \{0,1\}$, and consider the code $C_1 = \{\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2\}$, where

$$\mathbf{c}_0 = \{(0,0), (2,1)\}$$
$$\mathbf{c}_1 = \{(0,1), (1,0)\}$$
$$\mathbf{c}_2 = \{(1,1), (2,0)\}$$

Each codeword $\mathbf{c}_i$ is a partial function $\mathbf{c}_i : \mathbb{N} \to \{0,1,2,\phi\}$, so $\mathbf{c}_i \in \mathcal{B}^\dagger$ and $C_1 \subset \mathcal{B}^\dagger$. Notice that $D(\mathbf{c}_0) = \{0,2\}$, $D(\mathbf{c}_1) = \{0,1\}$, $D(\mathbf{c}_2) = \{1,2\}$, and $D(C_1) = \{0,1,2\}$. We may find it convenient to represent C_1 as an array, as in Figure 2.1, where the i^{th} row represents codeword $\mathbf{c}$. The entries in each row correspond to the contents of the corresponding memory cells. The j^{th} entry in row $\mathbf{c}_i$ is the value $\mathbf{c}_i(j)$ if $j \in D(\mathbf{c}_i)$ and is blank if $j \notin D(\mathbf{c}_i)$. Each column corresponds to a memory cell address, here 0, 1, or 2. ∎

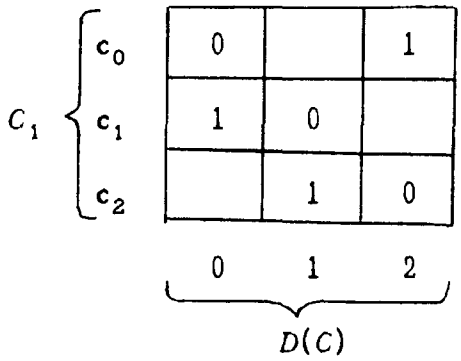


Figure 2.1. Representation of Code C_1 as an array.

Recall that we write $\mathcal{B}^L$ to denote the set of all L-celled memories. Then a memory state m is in $\mathcal{B}^L$ if $m \in \mathcal{B}^\dagger$ and its domain is $D(m) = \{0,1,2, \ldots, L\text{-}1\}$, so that

$$m = \{(0,m(0)), (1,m(1)), \cdots, (L\text{-}1,m(L\text{-}1))\},$$

where the first member of each pair $(n, m(n))$ is the integer address $n \in \mathbb{N}$ of a cell in memory, and the second is the contents $m(n) \in \mathcal{B}$ of cell n. (Recall that it is

possible that L be infinite.) A codeword $\mathbf{c} \in C$ is stored in a memory $m \in \mathcal{B}^L$ by setting $m(j) = \mathbf{c}(j)$, for all $j \in D(\mathbf{c})$. Other users may fill in the values of the $L - |\mathbf{c}|$ cells not occupied by $\mathbf{c}$ but must leave $\mathbf{c}$ itself undisturbed.

For any string $b \in \mathcal{B}^+$ we can define its *L-closure*, $\bar{b}_L$, as the set

$$\bar{b}_L = \{m \in \mathcal{B}^L \mid b \subseteq m\}$$

of memories in $\mathcal{B}^L$ that store b, in the sense that the (address, value) pairs in b are included among those in m. For $L < \max D(b)$, $\bar{b}_L = \varnothing$. Where the value L is understood, we frequently write $\bar{b}$ to mean $\bar{b}_L$. Note that $|\bar{b}_L| = |\mathcal{B}|^{L-|b|}$.

Define the set

$$\mathcal{B}^* = \bigcup_{L \geq 0} \mathcal{B}^L$$

of all finite memories that store values from $\mathcal{B}$. Then for $b \in \mathcal{B}^*$, $D(b) = \{0,1, \ldots, L\}$ for some $L \in \mathbb{N}$. So the L-closure of b contains all sequences in $\mathcal{B}^L$ with prefix b: $\bar{b} = b \cdot \mathcal{B}^{(L-|b|)}$.

Example 2.5. Recall code C_1 from Example 2.4, where $\mathcal{B} = \{0,1\}$. Since $|(\bar{\mathbf{c}}_i)_L| = |\mathcal{B}|^{L-|\mathbf{c}_i|}$, then $|(\bar{\mathbf{c}}_i)_3| = 2^{3-|\mathbf{c}_i|} = 2$. So for $L = 3$ there are two memory states which contain the codeword $\mathbf{c}_i$. In particular,

$$\begin{aligned}(\bar{\mathbf{c}}_0)_3 &= \{m \in \mathcal{B}^3 \mid \mathbf{c}_0 \subseteq m\} \\ &= \{\{(0,0), (1,0), (2,1)\}, \{(0,0), (1,1), (2,1)\}\}.\end{aligned}$$

We can represent the 3-closures of $\mathbf{c}_0$, $\mathbf{c}_1$, $\mathbf{c}_2$ in array form, as in Figure 2.2. Notice that no matter how other users may fill in memory cells n where $n \notin D(\mathbf{c}_i)$, it is always possible to tell precisely what codeword $\mathbf{c}_i$ is being stored. Since $L = 3$ and $\mathcal{B} = \{0,1\}$, there are eight possible memory states, six of which store codewords from C_1.

Also note that

$$\begin{aligned}(\bar{\mathbf{c}}_0)_2 &= \varnothing \\ (\bar{\mathbf{c}}_0)_4 &= \{\{(0,0), (1,0), (2,1), (3,0)\}, \{(0,0), (1,0), (2,1), (3,1)\}, \\ &\quad \{(0,0), (1,1), (2,1), (3,0)\}, \{(0,0), (1,1), (2,1), (3,1)\}\}.\end{aligned}$$

Since $\mathbf{c}_1 \in \mathcal{B}^2$, $\mathbf{c}_1 \in \mathcal{B}^*$, but $\mathbf{c}_0, \mathbf{c}_2 \notin \mathcal{B}^*$. ∎

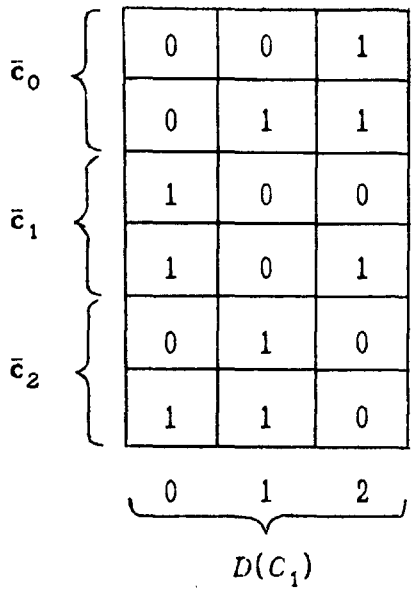


Figure 2.2. Representation of the closures of codewords in C_1.

Having discussed what we mean by an encoding $\tau:\mathbb{D} \to \mathcal{B}^L$ and a code $C \subset \mathcal{B}^+$, we can now explain what we shall mean by a representation $\rho:\mathbb{D} \to \mathcal{B}^+$. Throughout the thesis, unless otherwise specified, we always make the assumption that ρ is a one-to-one function. Thus $\rho(d)$ is a single codeword in $\mathcal{B}^+$, and

$$(\forall d_i, d_j \in \mathbb{D})(i \neq j \Rightarrow \rho(d_i) \neq \rho(d_j)).$$

The one-to-one condition guarantees that distinct data bases d_i and d_j map to distinct codewords. Since

$$\bar{\rho}_L(d) = \{m \in \mathcal{B}^L \mid \rho(d) \subseteq m\},$$

we can see that the relation $\bar{\rho}$ corresponds to the relation τ in Section 2.3. When $\mathfrak{M}$'s memory contains precisely L cells, a specification of a representation ρ indicates, for any $d \in \mathbb{D}$, that the cells in $D(\rho(d))$ be filled in as specified and the remaining cells can be filled in any possible way by other users.

For instance, suppose we have some representation ρ, for which $\rho(d_0) = \{(0,1), (2,0)\}$; i.e., $d_0 \in \mathbb{D}$ is represented by any memory state in which $m(0) = 1$ and $m(2) = 0$. Since the value $m(2)$ to be stored in cell 2 is not specified, cell 2 corresponds to a "don't care". For $L = 3$, we shall find it

convenient to write $\rho(d_0) = 1\_0$ to mean $\rho(d_0) = \{(0,1), (2,0)\}$. Where L is understood, we may even write $\rho(d_0) = 1\_0$ rather than $\rho(d_0) = 1\_0\_\_$ for $L = 5$; i.e., we may suppress all trailing "don't cares", which serve simply as place holders.

We saw in Example 2.5 that if $\mathbf{c}_i \in C_1$ is stored in memory, then it is always possible to distinguish $\mathbf{c}_i$, no matter what other users have done with cells not in $D(\mathbf{c}_i)$. In other words, there is no memory state in $\mathcal{B}^L$ that stores both $\mathbf{c}_i$ and $\mathbf{c}_j$, for $i \neq j$. When this is the case, we say that $\mathbf{c}_i$ and $\mathbf{c}_j$ are distinguishable.

Definition. Let $\rho:\mathbb{D} \to \mathcal{B}^+$, and let $d_1, d_2 \in \mathbb{D}$. Then $\rho(d_1)$ and $\rho(d_2)$ are said to be *distinguishable* if and only if

$$\overline{\rho}_L(d_1) \cap \overline{\rho}_L(d_2) = \varnothing$$

for any $L \geq \max\{\max D(\rho(d_1)), \max D(\rho(d_2))\}$.

In other words, a code $C \subset \mathcal{B}^+$ is distinguishable if and only if the closures of its members are pairwise disjoint (see Elias [8]).

If there exist $d_1, d_2 \in \mathbb{D}$ such that $\rho(d_1)$ and $\rho(d_2)$ are not distinguishable, then for some memory state m_0 it is not possible to tell whether d_1 or d_2 is stored; in fact, m_0 represents both d_1 and d_2. We do not want to allow this loss of information and so make the following formal definition of a representation.

Definition. We say that a function $\rho:\mathbb{D} \to \mathcal{B}^+$ is a *representation* if and only if for all $d_1, d_2 \in \mathbb{D}$, where $d_1 \neq d_2$, $\rho(d_1)$ and $\rho(d_2)$ are distinguishable.

Example 2.6. Let $\mathbb{D} = \{d_0, d_1, d_2\}$, $\mathcal{B} = \{0,1\}$, and $L = 3$. Consider the function $\rho:\mathbb{D} \to \mathcal{B}^+$ defined by

$$\begin{aligned} \rho(d_0) &= 0\_1 \\ \rho(d_1) &= 10\_ \\ \rho(d_2) &= \_00 \end{aligned}$$

Then ρ is not a representation, because it does not have disjoint 3-closures. In particular, $\rho(d_1)$ and $\rho(d_2)$ are not distinguishable:

$$\overline{\rho}(d_1) \cap \overline{\rho}(d_2) = \{100, 101\} \cap \{000, 100\} = \{100\}$$ ∎

Example 2.7. Let us define the function $\rho:\mathbb{D} \to \mathcal{B}^3$ by

$$\rho(d_0) = 0\Diamond\_ \qquad \rho(d_4) = 11\Diamond$$
$$\rho(d_1) = 1\Diamond\_ \qquad \rho(d_5) = 12\Diamond$$
$$\rho(d_2) = 2\Diamond\_ \qquad \rho(d_6) = 20\Diamond$$
$$\rho(d_3) = 10\Diamond$$

Notice that

$$\bar{\rho}_3(d_0) = \overline{\{(0,0), (1,\Diamond)\}}_3 = \{000, 001, 002, 000\}.$$

Thus, there are four memory states that correspond to a representation of d_0, and the relation $\bar{\rho}$ is identical to the relation τ of Example 2.2. ∎

From now on, we define an encoder by specifying a representation function ρ. Then any string $b \in \bar{\rho}_L(d)$ represents the data base d.

2.6 Solution of Dynamic Problems

In Section 2.3 we explained what it means for a machine $\mathfrak{M}$ to answer correctly a question q. Now that we have also discussed what we mean by a representation, we can explicitly state what we mean when we say that a machine $\mathfrak{M}$ solves some list problem.

We can extend the notion of the computation of a function (question) q to include the solution of a set of questions $Q = \{q_i | i \in \mathcal{I}\}$, where each $q_i:\mathbb{D} \to R_i$ maps a common domain $\mathbb{D}$ onto its own range R_i. Since the ranges are in general different for different questions, a set $\Delta = \{\delta_i | i \in \mathcal{I}\}$ of different decodings is allowed. For the solution of the family of questions Q, we introduce a set $\mathfrak{M} = \{\mathfrak{M}_i | i \in \mathcal{I}\}$ of machines with a family $\Omega = \{\omega_i | i \in \mathcal{I}\}$ of different characteristic functions, where $\omega_i:\mathfrak{D}_i \to \mathfrak{R}_i$. We can consider $\mathfrak{M}$ to be a single device, with a set $S_{\mathcal{I}} = \{s_i | i \in \mathcal{I}\}$ of distinct initial states, or programs. $\mathfrak{M}_i$ is the submachine corresponding to $\mathfrak{M}$ started in the initial state s_i. We say that $(\mathfrak{M}, \rho, \Delta)$ solves $(Q, \mathbb{D})$ if, for all $i \in \mathcal{I}$, $\mathfrak{M}_i$ computes q_i correctly. In other words, if $(\mathfrak{M}_i, \rho, \delta_i)$ computes q_i, then for any $m \in \bar{\rho}_L(d)$, $q_i(d) = \delta_i \circ \omega_i(m)$.

Having seen what it means for a machine to solve a static problem $(Q, \mathbb{D})$, let us now extend this to include updates. Recall that in our discussion of the machine model, it was mentioned that $\mathfrak{M}$ may rewrite some of its memory cells. Thus, when given some input m_0, $\mathfrak{M}$ may halt in a new memory state m_1. For a machine $\mathfrak{M}$ which computes a single function f, if we want to be able to compute f several times in succession, then it is natural to require that this new memory state be in $\mathfrak{M}$'s acceptance set. In fact, if $\mathfrak{M}_i$ solves (q_i, u_i) correctly, then performing an update function on any memory state containing $\rho(d)$ leaves us with a memory state that is a representation of the problem domain update function $u_i(d)$. In general, we want a machine $\mathfrak{M}$ to compute a family of functions F, and so we represent our update function in the machine domain by the family of functions $\Upsilon = \{v_i | \ i \in \mathcal{J}\}$, where $v_i: \mathfrak{D}(\mathfrak{M}) \to \mathfrak{D}(\mathfrak{M})$ for $\mathfrak{D}(\mathfrak{M}) = \bigcup_{i \in \mathcal{J}} \mathfrak{D}(\mathfrak{M}_i) \subseteq \mathcal{B}^L$.

Definition. Consider the machine $\mathfrak{M} = \{\mathfrak{M}_i | \ i \in \mathcal{J}\}$ with the family $\Omega = \{\omega_i | \ i \in \mathcal{J}\}$ of characteristic functions and the family $\Upsilon = \{v_i | \ i \in \mathcal{J}\}$ of update functions. We say that $(\mathfrak{M}, \rho, \Delta)$ *solves* the dynamic problem $(F, \mathbb{D})$ if the following conditions are satisfied for all $f_i = (q_i, u_i) \in F$:

(1) $q_i(d) = \delta_i \circ \omega_i \circ \bar{\rho}_L(d)$

(2) $v_i(\bar{\rho}_L(d)) \subseteq \bar{\rho}_L(u_i(d))$.

2.7 Solution of a List Problem

In this section we merely want to summarize what we shall mean when we talk about the solution of a list problem.

First, recall from Section 2.1 that a list problem is a storage and retrieval problem $(F, \mathbb{D})$ where the domain elements have some list structure, e.g., they may be stacks. In any case the problem domain $\mathbb{D}$ consists of strings of characters chosen from the problem alphabet X and is of the following form: $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$ for some $\mathcal{J} \subseteq \mathbb{N}$. For any $d \in \mathbb{D}$, we want to be able to perform the operations in F; e.g., TOP (return the value at the top of the stack) and POP.

If a machine $\mathfrak{M}$ is to solve the list problem $(F, \mathbb{D})$, then there must be some way to represent each $d \in \mathbb{D}$ in the cells of $\mathfrak{M}$'s memory with machine alphabet $\mathcal{B}$. In particular, there is some one-to-one representation function $\rho: \mathbb{D} \rightarrow \mathcal{B}^{+}$, and any $\rho(d)$ stored in m can be viewed as some sort of codeword. The representation has the property that it is always possible to determine what (if any) codeword is currently stored in memory. What other users do cannot interfere with this determination.

Suppose the current memory state is m_0, where $m_0 \in \bar{\rho}_L(d)$. Then $\mathfrak{M}_i$ will output the answer $\omega_i \circ \rho_L(d)$ and halt in the new memory state $\upsilon_i(\bar{\rho}_L(d))$. If we claim that $\mathfrak{M}_i$ computes the function $f_i = (q_i, u_i) \in F$, then $\upsilon_i(\bar{\rho}_L(d)) \subseteq \bar{\rho}_L(u_i(d))$ and there must be some sort of decoding function δ_i such that $q_i(d) = \delta_i \circ \omega_i \circ \bar{\rho}_L(d)$. In other words, $\mathfrak{M}_i$ outputs the machine representation of $q_i(d)$ and halts in a memory state which is included in the set of memory states that represent $u_i(d)$.

We say that $(\mathfrak{M}, \rho, \Delta)$ solves the list problem $(F, \mathbb{D})$ if the above conditions are satisfied for all $f_i \in F$ and for all $d \in \mathbb{D}$. For simplicity, we shall also assume that each decoding function $\delta_i \in \Delta$ is one-to-one. Thus, we speak of a system $(\mathfrak{M}, \rho)$ solving a problem $(F, \mathbb{D})$.

When we discuss the machine solution of a problem $(F, \mathbb{D})$, we have in mind a representation of the domain $\mathbb{D}$ in memory and some collection $\mathcal{A}$ of algorithms or programs which compute the functions F. Any algorithm $\mathcal{A}_i$ that we discuss can be implemented by a machine $\mathfrak{M}_i$ as defined above. Since we do not, however, always want to concern ourselves with all the details of the machine itself, we shall henceforth speak of a system $(\mathcal{A}, \rho)$ solving a problem $(F, \mathbb{D})$. Thus we specify an implementation by defining the function ρ and by, in some (usually program-like) form, presenting the set of algorithms $\mathcal{A}$ (which can be implemented by machine $\mathfrak{M}$).

CHAPTER 3

STORAGE AND ACCESS COSTS

In Section 3.1 we introduce various system costs involved in solving a problem. Since in this thesis we are concerned with obtaining lower bounds on storage and access costs, these costs are discussed more fully in sections 3.2 and 3.3, respectively. We first define our cost measures and then present some basic results. For further information the interested reader is referred to Elias [6], [9], [10].

3.1 System Costs

Many different systems can be used to solve the same problem, and the choice among them depends on their relative costs. There are three basic components of system cost:

(1) Storage cost. There is always some sort of purchase or rental cost for the memory used to store the representation of a data base.

(2) Access cost. This refers to the number of memory cell accesses made by an algorithm or machine and is a partial indication of the time required by a system to answer a question or perform an update.

(3) Processor cost. This involves the costs in memory and logic of the algorithm or machine $\mathfrak{M}$ itself.

For several reasons, we do not in this thesis consider the processor cost. First, any such measure would reflect characteristics of the particular machine, and it is therefore difficult to determine an appropriate measure. We have deliberately tried to let our machine model be as general as possible. Second, the list implementations we do consider are in general quite straightforward and therefore a system which does well for both storage and access costs probably would not have a prohibitive processor cost. Third, the storage-access trade-off is easier to recognize and we do not want the current analysis to become too complex.

3.2 Storage Costs

One measure of the memory requirements of a retrieval system $(\mathcal{A}, \rho)$ solving a problem $(F, \mathbb{D})$ is the number of memory cells dedicated to the storage of a representation in memory.

Definition. Consider a system $(\mathcal{A}, \rho)$ solving a problem $(F, \mathbb{D})$, and assume that ρ is a function. The memory *storage cost*, $|\rho(d)|$, associated with any data base $d \in \mathbb{D}$ is the number of memory cells for which representation ρ specifies a value when representing d:

$$|\rho(d)| \triangleq |D(\rho(d))|.$$

Thus, we define $|\rho(d)|$ to be the number of memory cells occupied by the codeword $\rho(d)$. There is, however, no requirement that the set of occupied cells be contiguous; i.e., there may be "gaps" or "holes" in the representation. Because we are essentially concerned with obtaining lower bounds, we charge only for the cells actually occupied by $\rho(d)$ and do not charge for these gaps.

Example 3.1. Let $\mathcal{B} = \{0,1\}$ and define the code $C_2 = \{\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4\}$ as follows:

$$\mathbf{c}_0 = 0\_1 \qquad \mathbf{c}_3 = 000$$
$$\mathbf{c}_1 = 10\_ \qquad \mathbf{c}_4 = 111$$
$$\mathbf{c}_2 = \_10$$

Suppose that $\mathbb{D} = \{d_0, d_1, d_2, d_3, d_4\}$ and the representation $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$ is defined by $\rho(d_i) = \mathbf{c}_i$. Then

$$|\rho(d_0)| = |\rho(d_1)| = |\rho(d_2)| = 2$$

and

$$|\rho(d_3)| = |\rho(d_4)| = 3. \qquad \blacksquare$$

Certainly the issue of memory management is an important one, because it may be difficult to efficiently allocate to a single user the unspecified memory cells corresponding to holes in another user's memory space. Elias [9] has addressed the

problem of assigning a contiguous section of memory, defining the span of a representation ρ to be the smallest set of contiguous memory cells capable of holding the representation of any domain element. Many representation schemes we shall construct will be able to avoid such gaps, at least when the problem alphabet is of the appropriate size.

Our storage cost measure does not indicate the complexity of the encoding ρ. For a static problem, storing a representation would be only a one-time task. When we consider dynamic problems, the complexity of the representation will evidence itself in the costs of performing updates. In general, a complicated encoding results in higher access costs.

Consider a code $C \subset \mathcal{B}^+$ that has the property that for each $\mathbf{c} \in C$, $D(\mathbf{c}) = \{0,1, \ldots, |\mathbf{c}|-1\}$; i.e., $C \subset \mathcal{B}^*$. Then C is said to be a *prefix code*, or to be prefix-free, if none of its members is a prefix of any other. In other words, a prefix-free set $C \subset \mathcal{B}^*$ has the property that

$$(\forall \mathbf{c}_1, \mathbf{c}_2 \in C)\ (\mathbf{c}_1 \not\subseteq \mathbf{c}_2).$$

As noted by Elias [8], a code $C \subset \mathcal{B}^*$ is distinguishable if and only if it is a prefix code.

The well known *Kraft inequality* [2], [12], [16] states that a necessary and sufficient condition for the existence of a prefix code with codeword lengths $\ell_1, \ell_2, \ldots, \ell_k$ and codeword characters chosen from the alphabet $\mathcal{B}$ is that:

$$\sum_{i=1}^{k} |\mathcal{B}|^{-\ell_i} \leq 1.$$

This result is probably most easily seen by recalling the simple correspondence between prefix codes and labeled trees. Each node corresponds to a memory cell number, and the branch labels correspond to the cell contents; i.e., there are $|\mathcal{B}|$ branches from each node. Each codeword is associated with a distinct leaf. We adopt the convention that the leftmost branch of each node always corresponds to the same element $b_0 \in \mathcal{B}$, and similarly for each of the other branches. For full trees this convention eliminates the need for writing the labels on branches emanating from non-root nodes. In particular, for $\mathcal{B} = \{0,1\}$, we always let a

leftward branch correspond to a zero and a rightward branch to a one.

Example 3.2. Recall the representation $\rho:\mathbb{D} \to \mathcal{B}^*$ from Example 2.7. The code

$$\rho(\mathbb{D}) = \{0\emptyset, 1\emptyset, 2\emptyset, 10\emptyset, 11\emptyset, 12\emptyset, 20\emptyset\}$$

is a prefix code and satifies the Kraft inequality because

$$\sum_{c \in \rho(\mathbb{D})} |\mathcal{B}|^{-|c|} = 3 \cdot 4^{-2} + 4 \cdot 4^{-3} = \frac{1}{4} < 1$$

The tree corresponding to the code $\rho(\mathbb{D})$ is illustrated in Figure 3.1. ∎

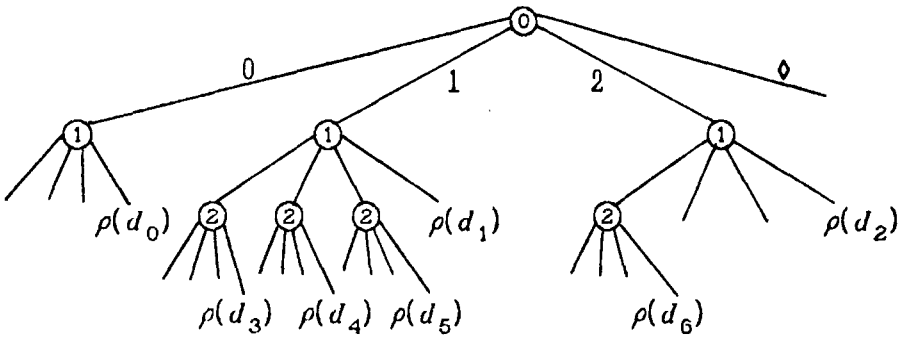


Figure 3.1. Tree corresponding to ρ from Example 3.2.

Elias has extended the Kraft inequality to any distinguishable code $C \subset \mathcal{B}^+$.

Theorem 3.1. (Elias [8]). Let $C \subset \mathcal{B}^+$ be distinguishable. Then

$$\sum_{c \in C} |\mathcal{B}|^{-|c|} \leq 1. \tag{3.1}$$

Equivalently, consider any representation $\rho:\mathbb{D} \to \mathcal{B}^+$. Then

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} \leq 1. \tag{3.2}$$

Proof: Let

$$C_L = \{\mathbf{c} \in C \mid L \geq \max D(\mathbf{c})\}$$

be the subset of the code C whose elements can be stored in an L-cell memory. Since C is distinguishable, the closures of its members are disjoint and we have

$$\bigcup_{c \in C_L} \bar{\mathbf{c}}_L \subseteq \mathcal{B}^L.$$

Recalling also that $|\bar{c}_L| = |\mathcal{B}|^{L-|c|}$, we obtain

$$\sum_{c \in C_L} |\mathcal{B}|^{L-|c|} \leq |\mathcal{B}|^L$$

Now dividing through by $|\mathcal{B}|^L$ gives

$$\sum_{c \in C_L} |\mathcal{B}|^{-|c|} \leq 1.$$

Since $C_L \subseteq C_{L+1}$,

$$\sum_{c \in C_L} |\mathcal{B}|^{-|c|} \leq \sum_{c \in C_{L+1}} |\mathcal{B}|^{-|c|} \leq 1,$$

and so

$$\lim_{L \to \infty} \left(\sum_{c \in C_L} |\mathcal{B}|^{-|c|} \right) = 1.$$

This proves (3.1). Since any representation ρ is by definition distinguishable, the Kraft inequality also holds for representation storage costs and thus (3.2) follows. ∎

Theorem 3.1 is a statement about distributions of the storage measure $|\rho(d)|$ for any representation ρ of domain $\mathbb{D}$. Not all data bases in $\mathbb{D}$ can have short representations, since a small value of $|\rho(d)|$ corresponds to a large term in the Kraft sum. If some of the data bases have relatively short representations then others must have relatively long representations. If, in fact, we have equality in the Kraft sum, then no data base representation can be shortened without lengthening another data base representation.

Definition. We say that a *representation ρ achieves Kraft storage* if and only if the Kraft sum of equation (3.2) is satisfied with equality:

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = 1 \qquad (3.3)$$

Similarly, a *code C achieves Kraft storage* if the Kraft sum of equation (3.1) is equal to one.

We can also extend our usage of trees to correspond to any distinguishable code $C \subset \mathcal{B}^+$. However, since we do not restrict ourselves to prefix codes (i.e., we allow scattered representations), we would not necessarily choose to have the memory cells read in order 0, 1, 2, ... on the path to every leaf. This and the result of Theorem 3.1 are illustrated in the following example.

Example 3.3. a) For code C_1 of Example 2.4, $|\mathcal{B}| = 2$ and

$$\sum_{c \in C_1} |\mathcal{B}|^{-|c|} = 2^{-2} + 2^{-2} + 2^{-2} = \frac{3}{4} < 1.$$

A tree corresponding to C_1 is given in Figure 3.2a, with the memory cells listed in order 0, 1, 2. On the other hand, we might choose to represent C_1 by the tree in Figure 3.2b. In any case, C_1 does not achieve Kraft storage.

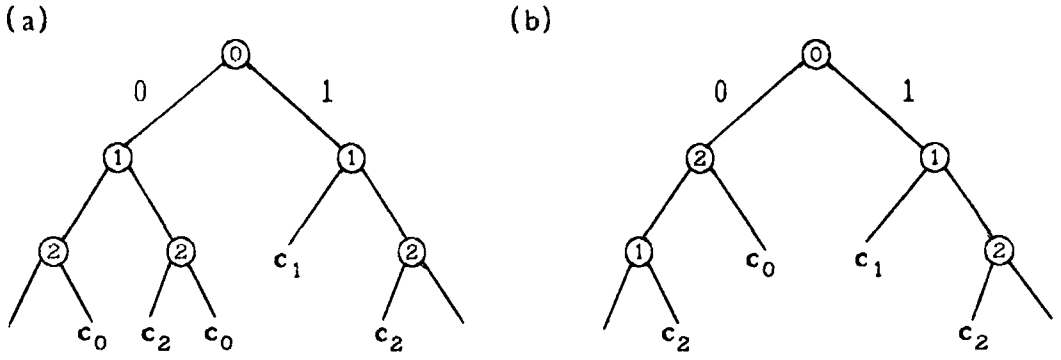


Figure 3.2. Trees corresponding to code C_1.

b) For code C_2 of Example 3.1,

$$\sum_{c \in C_2} |\mathcal{B}|^{-|c|} = 3 \cdot 2^{-2} + 2 \cdot 2^{-3} = 1$$

and so C_2 achieves Kraft storage. A tree for code C_2 is given in Figure 3.3

c) Recall once again the representation $\rho: \mathbb{D} \to \mathcal{B}^*$ from examples 2.7 and 3.2. Then since each $d \in \mathbb{D}$ has a unique representation $\rho(d)$:

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = \sum_{c \in \rho(\mathbb{D})} |\mathcal{B}|^{-|c|} = 3 \cdot 4^{-2} + 4 \cdot 4^{-3} = \frac{1}{4} < 1. \quad \blacksquare$$

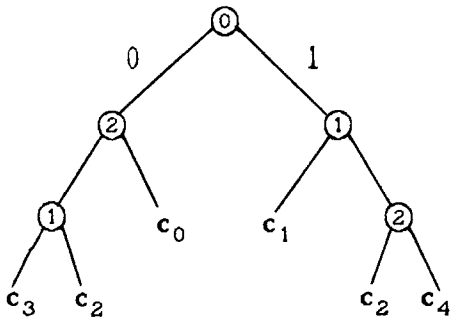


Figure 3.3. Tree corresponding to code C_2.

When we solve some problem we would like to find a representation that does not result in high storage costs. We say that a representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ is optimal in storage if no other representation requires less storage for some data base without requiring more storage for another.

Definition. A representation function $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ achieves *optimal storage* if and only if for any $\rho':\mathbb{D} \rightarrow \mathcal{B}^+$

$$(\forall d_1 \in \mathbb{D})[(|\rho'(d_1)| < |\rho(d_1)|) \Rightarrow (\exists d_2 \in \mathbb{D})(|\rho'(d_2)| > |\rho(d_2)|)].$$

Thus, we use the term optimal storage for a representation if no other representation can uniformly do better. There may, of course, be many representations that are storage optimal, and which would be preferred depends on the particular problem and is conditional on the probabilities of the various data bases in $\mathbb{D}$. In fact, one might not choose to use a storage optimal representation at all if such a representation resulted in higher access or other system costs. However, these involve details of particular problems and, for the general framework we are considering, we shall not usually prefer one optimal representation over another.

If a representation ρ meets the Kraft sum with equality, then ρ is storage optimal. This condition makes it easy to recognize certain storage optimal representations.

Theorem 3.2. Consider the representation function $\rho:\mathbb{D} \to \mathcal{B}^+$. If

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = 1,$$

then ρ is storage optimal. In other words, if ρ achieves Kraft storage then ρ is storage optimal.

Proof: If ρ is not storage optimal, then there exists some representation $\rho':\mathbb{D} \to \mathcal{B}^+$ such that $(\forall d \in \mathbb{D})(|\rho'(d)| \leq |\rho(d)|)$ and $(\exists d_1 \in \mathbb{D})(|\rho'(d_1)| < |\rho(d_1)|)$. But this says that

$$\begin{aligned} 1 &= \sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} \\ &= \sum_{d \in \mathbb{D}-\{d_1\}} |\mathcal{B}|^{-|\rho(d)|} + |\mathcal{B}|^{-|\rho(d_1)|} \\ &\leq \sum_{d \in \mathbb{D}-\{d_1\}} |\mathcal{B}|^{-|\rho'(d)|} + |\mathcal{B}|^{-|\rho(d_1)|} \\ &< \sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho'(d)|} \end{aligned}$$

which contradicts the Kraft inequality of Theorem 3.1. ∎

Example 3.4. Recall Example 2.7 where $\mathcal{B} = \{0,1,2,\Diamond\}$, and consider the alternative encoding $\rho_2:\mathbb{D} \to \mathcal{B}^+$ defined by

$$\begin{aligned} \rho_2(d_0) &= 0 & \rho_2(d_4) &= \Diamond 1 \\ \rho_2(d_1) &= 1 & \rho_2(d_5) &= \Diamond 2 \\ \rho_2(d_2) &= 2 & \rho_2(d_6) &= \Diamond\Diamond \\ \rho_2(d_3) &= \Diamond 0 \end{aligned}$$

and also the encoding $\rho_3:\mathbb{D} \to \mathcal{B}^+$ defined by

$$\begin{aligned} \rho_3(d_0) &= 00 & \rho_3(d_4) &= 1 \\ \rho_3(d_1) &= \Diamond & \rho_3(d_5) &= 2 \\ \rho_3(d_2) &= 02 & \rho_3(d_6) &= 01 \\ \rho_3(d_3) &= 0\Diamond \end{aligned}$$

By Theorem 3.2, both ρ_2 and ρ_3 are storage optimal because

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho_2(d)|} = 3 \cdot 4^{-1} + 4 \cdot 4^{-2} = 1 = \sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho_3(d)|}.$$

On the other hand, ρ as defined in Example 2.7 is not storage optimal because ρ_2 does better; in fact, ρ_2 takes less storage everywhere:

$$(\forall d \in \mathbb{D})(|\rho_2(d)| < |\rho(d)|).$$

The representation ρ_3 also does better than ρ, because it never uses more storage and sometimes uses less.

If we were forced to pay a very high price for storage, we would probably choose to solve the problem (F, $\mathbb{D}$) of Example 2.1 using representation ρ_2 or ρ_3 rather than ρ. However, ρ corresponds to a simple ternary representation (with $\diamond$ serving as an endmarker) and might be more desirable than ρ_2 or ρ_3 in terms of other costs. ∎

We have seen that a code $\rho(\mathbb{D})$ achieves optimal storage if we get equality in the Kraft sum. Let us examine the conditions under which this equality is attained. We first define a distinguishable code $C \subset \mathcal{B}^+$ to be *complete* if and only if for all $\mathbf{c}' \in \mathcal{B}^+$, $C \cup \{\mathbf{c}'\}$ is not distinguishable. Elias [8] has shown that a finite distinguishable code $C \subset \mathcal{B}^+$ is complete if and only if the L-closure of its members partitions $\mathcal{B}^L$ (for $L = \max D(\mathbf{c})$) which is true if $\sum_{\mathbf{c} \in C} |\mathcal{B}|^{-|\mathbf{c}|} = 1$. The converse is not true, i.e., a code C may be complete even if $\sum_{\mathbf{c} \in C} |\mathcal{B}|^{-|\mathbf{c}|} \neq 1$.

Example 3.5. Recalling Example 3.3, we see that C_1 is not complete, since $C_1 \subset C_2$. However, C_2 is complete. If we look at the trees for C_1 and C_2, given in figures 3.2 and 3.3, it is easy to see that C_1 does not partition $\{0,1\}^3$, since there are some leaves in the tree for C_1 that correspond to no codeword. Also, by Example 3.3 we know that C_1 does not achieve Kraft storage and thus cannot be complete (since it is finite); C_2 does achieve Kraft storage and is therefore complete. ∎

We can conclude that, as illustrated in the above example, a finite $|\mathcal{B}|$-ary code C is complete if and only if every leaf in a full $|\mathcal{B}|$-ary tree for C corresponds to some codeword $\mathbf{c} \in C$.

Using the terminology of representations, we can show that if a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ achieves Kraft storage, then $\rho(\mathbb{D})$ is complete.

Theorem 3.3. Let $\rho:\mathbb{D} \to \mathcal{B}^+$ be some representation which achieves Kraft storage. Then for all $b \in \mathcal{B}^+$, there is some $d \in \mathbb{D}$ such that $b \subseteq \bar{\rho}_L(d)$.

Proof: Let ρ achieve Kraft storage and assume that there is some $b_0 \in \mathcal{B}^+$ such that, for all $d \in \mathbb{D}$, $b_0 \not\subseteq \bar{\rho}_L(d)$. In other words, b_0 and d are distinguishable, for every d. Then

$$\sum_{d \in \mathbb{D} \cup \{b\}} |\mathcal{B}|^{-|\rho(d)|} \leq 1 \quad \Rightarrow \quad \sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} < 1,$$

which contradicts the fact that ρ achieves Kraft storage. ∎

The converse is not true (see, once again, Elias [8]). However, if $\rho(\mathbb{D})$ is complete for $\mathbb{D}$ finite, then we do know that ρ achieves Kraft storage.

Let us briefly mention two results concerning worst case and average storage costs. The first result follows from well-known tree properties (see e.g. Gallager [12]) and states that for any representation $\rho:\mathbb{D} \to \mathcal{B}^+$, there is some data base whose representation specifies values for at least $\lceil \log_{|\mathcal{B}|} |\mathbb{D}| \rceil$ memory cells. On the other hand, for any domain $\mathbb{D}$ there is some representation which never requires more than $\lceil \log_{|\mathcal{B}|} |\mathbb{D}| \rceil$ memory cells.

Theorem 3.4. (Elias [6]). (i) For any representation function $\rho:\mathbb{D} \to \mathcal{B}^+$,

$$\max_{d \in \mathbb{D}} |\rho(d)| \geq \lceil \log_{|\mathcal{B}|} |\mathbb{D}| \rceil$$

(ii) There is some representation function $\rho:\mathbb{D} \to \mathcal{B}^*$ such that

$$\max_{d \in \mathbb{D}} |\rho(d)| = \lceil \log_{|\mathcal{B}|} |\mathbb{D}| \rceil$$

This result can be interpreted in terms of any tree corresponding to the $|\mathcal{B}|$-ary

distinguishable code $\rho(\mathbb{D})$, where there must be at least $|\mathbb{D}|$ leaves (since ρ is one-to-one). Since the tree is $|\mathcal{B}|$-ary, the depth of the tree (i.e., the length of the longest codeword) must be at least $\lceil \log_{|\mathcal{B}|} |\mathbb{D}| \rceil$. Also, a complete, full $|\mathcal{B}|$-ary tree with $|\mathbb{D}|$ leaves has all of its leaves at either depth $\lceil \log_{|\mathcal{B}|} |\mathbb{D}| \rceil$ or depth $\lceil \log_{|\mathcal{B}|} |\mathbb{D}| \rceil - 1$.

The second result involves average storage costs. There will be occasions where we wish to consider some sort of probability distribution P on the members of our domain $\mathbb{D}$:

$P(d) \triangleq$ the fraction of time a user expects to consider data base $d \in \mathbb{D}$.

Thus, it makes sense to look at the average storage cost:

$$\sum_{d \in \mathbb{D}} P(d) \cdot |\rho(d)|.$$

We can use a procedure such as Huffman encoding [13], [12] to construct a representation ρ for which very probable data bases have short representations and less probable data bases have longer representations. Other preconstructed universal codes perform almost as well as Huffman codes, provided the shorter preconstructed representations are assigned to the more probable data bases (see Elias [7]).

Theorem 3.5. (Elias [6]). Consider a domain $\mathbb{D}$ and assume there is some probability distribution P on $\mathbb{D}$. Define the entropy $H(\mathbb{D})$ by

$$H(\mathbb{D}) = -\sum_{d \in \mathbb{D}} P(d) \log_{|\mathcal{B}|} P(d).$$

(i) For any representation function $\rho: \mathbb{D} \to \mathcal{B}^+$, the average storage cost is

$$\sum_{d \in \mathbb{D}} P(d) \cdot |\rho(d)| \geq H(\mathbb{D}).$$

(ii) There is some representation function $\rho: \mathbb{D} \to \mathcal{B}^+$ such that

$$\sum_{d \in \mathbb{D}} P(d) \cdot |\rho(d)| < H(\mathbb{D}) + 1.$$

3.3 Access Costs

A user is necessarily concerned with the amount of time it takes to perform an operation f on some $d \in \mathbb{D}$. The number of memory cell accesses made by an algorithm before halting is one direct indication of the performance time. This memory access measure has been used by Minsky and Papert [19] and Elias [5]. The number of accesses made to memory will depend not only on the algorithm used but also on the particular data base which is stored.

There are various ways in which we could define an access, but we use the notion commonly used in Turing machine theory. A machine or algorithm reads a cell and, depending on that cell's contents, may rewrite the value stored there; this corresponds to only one access. We also choose to allow an algorithm to possibly read a cell in another user's memory space, but the algorithm certainly cannot rewrite such a cell (without being charged for it in storage).

> **Definition.** Consider a system (α, ρ) solving a problem $(F, \mathbb{D})$. A memory cell *access* is made each time α moves to a new cell. Once α references a cell, it may read and/or rewrite the cell contents; this constitutes a single access.

Depending on the hardware of an actual machine, this reading and then rewriting action might require two accesses, in which case our results could be off by a factor of two. Flower [11] has investigated update costs and shown that it is necessary for an access measure to involve both reads and writes; considering either reads or writes alone does not give reasonable lower bounds.

We present the following example in order to illustrate some of the terminology we shall use when we discuss the implementation of a function. We frequently find it convenient to describe an algorithm using a program-like description.

Example 3.6. Recall examples 2.1 and 2.7 and consider the problem of performing the update operation u_2 on some data base $d \in \mathbb{D}$. The following algorithm, α_{u_2}, performs the update. (For simplicity, we do not here consider the question component of the function f_2.)

```
α_u2:   if m(0) = 0 then return
        if m(0) = 1 then if m(1) = 0 then m(1) ← ◊
                                          return
                         if m(1) = 1 then m(1) ← ◊
                                          m(0) ← 2
                                          return
                         if m(1) = 2 then m(1) ← ◊
                                          m(0) ← 2
                                          return
                         if m(1) = ◊ then m(0) ← 0
                                          return
        if m(0) = 2 then m(0) ← 1
                         return
```

For instance, suppose we have $\rho(d_0)$ in memory. Given that we know there is some $\rho(d_i)$ stored, when we access cell 0 and discover that $m(0) = 0$, then we know that it is d_0 stored. Since $u_2(d_0) = d_0$, we do not need to rewrite any memory cells. Thus, performing the u_2 operation on $\rho(d_0)$, using algorithm α_{u_2}, involves only a reading of cell 0.

Suppose d_5 is stored in memory with representation ρ. Using algorithm α_{u_2}, we first access cell 0. Since $m(0) = 1$, we next access cell 1. Since $m(1) = 2$, we rewrite cell 1, setting it to the new value ◊, and then backtrack and set $m(0) \leftarrow 2$. ∎

Because we spend a great deal of time discussing algorithms for performing various operations, we find it convenient to make some notational definitions for dealing with memory access costs.

Definition. Suppose a system (α, ρ) solves a problem $(F, \mathbb{D})$. Then for each $d \in \mathbb{D}$ we can define the following.

$[\alpha_i(\rho(d))]$ ≜ the sequence of memory cell accesses made by algorithm α_i in computing $f_i(d)$ using representation ρ.

$\#[\alpha_i(\rho(d))]$ $\triangleq$ $|[\alpha_i(\rho(d))]|$, the number of memory cell accesses made by algorithm α_i in computing $f_i(d)$ using representation ρ.

$\{[\alpha_i(\rho(d))]\}$ $\triangleq$ the set of memory cells accessed by algorithm α_i in computing $f_i(d)$ using representation ρ; i.e., the access set for $f_i(d)$ corresponding to algorithm α_i.

We may sometimes write $[f_i(\rho(d))]$ to denote the access sequence which an algorithm α_i uses to compute $f_i(\rho(d))$.

We refer back to Example 3.6 to illustrate the above definition.

Example 3.7. Recall the algorithm α_{u_2} of Example 3.6. In computing $u_2(d_5)$, α_{u_2} first reads cell 0, then reads and rewrites cell 1, and then backtracks and writes cell 0. Thus, the access sequence is 0, 1, 0. For notational convenience, when we give an access sequence we shall underline any memory cell accesses which correspond to writes:

$[\alpha_i(\rho(d_0))] = 0$ $\quad$ $[\alpha_i(\rho(d_4))] = 0\underline{1}0$
$[\alpha_i(\rho(d_1))] = 0\underline{1}0$ $\quad$ $[\alpha_i(\rho(d_5))] = 0\underline{1}0$
$[\alpha_i(\rho(d_2))] = \underline{0}$ $\quad$ $[\alpha_i(\rho(d_6))] = \underline{0}$
$[\alpha_i(\rho(d_3))] = 0\underline{1}$

Then for the number of memory cell access in each case we clearly have:

$$\#[\alpha_{u_2}(\rho(d_0))] = \#[\alpha_{u_2}(\rho(d_6))] = \#[\alpha_{u_2}(\rho(d_2)] = 1$$
$$\#[\alpha_{u_2}(\rho(d_1))] = \#[\alpha_{u_2}(\rho(d_4))] = \#[\alpha_{u_2}(\rho(d_5)] = 3$$
$$\#[\alpha_{u_2}(\rho(d_3))] = 2$$

Note also that the access sets are just:

$$\{[\alpha_{u_2}(\rho(d_0))]\} = \{[\alpha_{u_2}(\rho(d_2))]\} = \{[\alpha_{u_2}(\rho(d_6)]\} = \{0\}$$
$$\{[\alpha_{u_2}(\rho(d_1))]\} = \{[\alpha_{u_2}(\rho(d_3))]\} = \{[\alpha_{u_2}(\rho(d_4)]\} = \{[\alpha_{u_2}(\rho(d_5))]\} = \{0,1\}$$ ∎

Since our algorithms are sequential and deterministic, we find it convenient to model them by *access trees*. Access trees are basically the same as the trees we used in Section 3.2, where each internal node corresponds to a memory cell access. An

access tree corresponding to the algorithm for a question q will label each leaf by the appropriate answer $q(d)$, if there is one. We speak of the access tree for q_i (or u_i) to mean the access tree for an algorithm α_i solving q_i (or u_i).

Example 3.8. Consider the static problem $(F, \mathbb{D})$ where $F = \{f_1, f_2\}$ and $\mathbb{D} = \{d_0, d_1, d_2\}$. Define the representation function $\rho: \mathbb{D} \to \{0,1\}^+$ by:

$$\rho(d_0) = 0\_0$$
$$\rho(d_1) = 1\_0$$
$$\rho(d_2) = \_\_1$$

Let q_1 and q_2 be defined as follows:

$$q_1(d_0) = a \qquad q_2(d_0) = a$$
$$q_1(d_1) = b \qquad q_2(d_1) = a$$
$$q_1(d_2) = b \qquad q_2(d_2) = b$$

where $a, b \in \mathcal{E}$. An access tree corresponding to the obvious algorithm for q_1 is given in Figure 3.4a. Notice that, in fact, two accesses are necessary to distinguish $\rho(d_0)$ from $\rho(d_1)$ or $\rho(d_2)$ and thus two accesses are required to determine the leaf that can be labelled a. Question q_2, however, can be answered after a single access, to cell 2. ∎

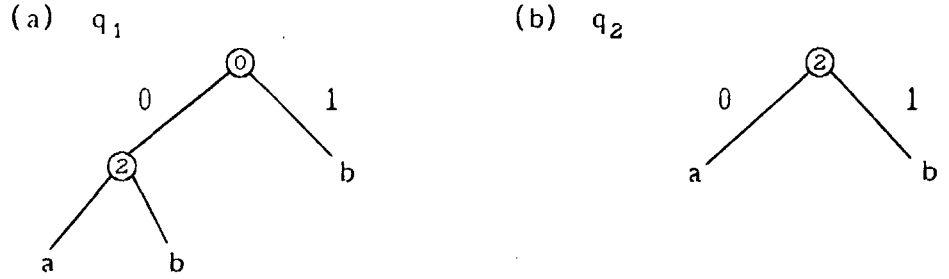


Figure 3.4. Access trees for q_1 and q_2 of Example 3.8

Each output corresponds to some leaf on the access tree for q_i, and we define $\alpha_i(r)$ to be the minimum depth of any leaf labelled r.

Definition. Suppose a system $(\mathcal{A}, \rho)$ solves a static problem $(Q, \mathbb{D})$, and let
$$\mathbb{D}_i(r) = \{d \in \mathbb{D} \mid q_i(d) = r\}.$$
Then
$$\alpha_i(r) \triangleq \min_{d \in \mathbb{D}_i(r)} \#[\mathcal{A}_i(\rho(d))].$$

Similar to our storage result, we have a Kraft inequality for access.

Theorem 3.6. (Elias [6]). If the $|\mathcal{B}|$-ary system $(\mathcal{A}, \rho)$ solves a static problem $(Q, \mathbb{D})$, then for all $q_i \in Q$:
$$\sum_{r \in q_i(\mathbb{D})} |\mathcal{B}|^{-\alpha_i(r)} \leq 1. \tag{3.4}$$

Corresponding to each answer $r \in q_i(\mathbb{D})$, the range of q_i, there is one term in the summation with negative exponent $\alpha_i(r)$. This theorem is a statement about distributions on the numbers of accesses to return the answers $r \in R$ and tells us that not all operations in $q_i(\mathbb{D})$ can have short retrieval times. In fact, equation (3.4) can be strengthened; it holds not only for $\alpha_i(r)$, the minimum number of accesses to return the value r, but also for the number of accesses to return the value r for any $d \in q_i^{-1}(r)$. In other words, if we let $d_i \in q_i^{-1}(r)$, then we have
$$\sum_{i=1}^{|Q|} |\mathcal{B}|^{-\#[\mathcal{A}_i(\rho(d_i))]} \leq 1.$$

Definition. Suppose a $|\mathcal{B}|$-ary system $(\mathcal{A}, \rho)$ solves a static problem $(Q, \mathbb{D})$. Then $\mathcal{A}_i$ is said to *achieve Kraft access* if
$$\sum_{r \in q_i(\mathbb{D})} |\mathcal{B}|^{-\alpha_i(r)} = 1. \tag{3.5}$$
In fact, if (3.5) holds and $(\mathcal{A}_i, \rho)$ is understood, we shall frequently say simply that q_i achieves Kraft access.

If we "assume q_i achieves Kraft access", we mean that we are considering some system $(\mathcal{A}, \rho)$ where $\mathcal{A}_i$ achieves Kraft access and answers q_i on domain $\mathbb{D}$.

In accessing a cell we read some $b_i \in \mathcal{B}$. Information-theoretically, one access distinguishes among $|\mathcal{B}|$ possibilities, and if it is not the case that each of these $|\mathcal{B}|$ possible cell contents leads to a different answer, then we have in some sense obtained more information than is needed. Thus, if an algorithm $\mathcal{A}$ achieves Kraft access, then its access tree must be a full tree where every leaf corresponds to a distinct $r \in R$. In particular, we have the following result.

Theorem 3.7. Suppose a system $(\mathcal{A}, \rho)$ solves a problem $(Q, \mathbb{D})$. If $\mathcal{A}_i$ achieves Kraft access, then for all $d_1, d_2 \in \mathbb{D}_i(r)$,

$$\#[\mathcal{A}_i(\rho(d_1))] = \#[\mathcal{A}_i(\rho(d_2))].$$

Let's look again at the problem from the previous example.

Example 3.9. Recall Example 3.8, and let R_1 and R_2 denote $q_1(\mathbb{D})$ and $q_2(\mathbb{D})$, respectively. For q_1:

$$\sum_{r \in R_1} |\mathcal{B}|^{-\alpha_1(r)} = \sum_{r \in \{a,b\}} 2^{-\alpha_1(r)}$$
$$= 2^{-\alpha_1(a)} + 2^{-\alpha_1(b)} = 2^{-2} + 2^{-1} = \frac{3}{4} < 1.$$

Notice that the access tree for q_1 in Figure 3.4a does not have a distinct label for each leaf and so cannot achieve Kraft access. For q_2:

$$\sum_{r \in R_2} |\mathcal{B}|^{-\alpha_2(r)} = 2^{-1} + 2^{-1} = 1,$$

and so does achieve Kraft access, which is what we would expect by observing Figure 3.4b. ∎

As we did for storage costs, we define an implementation or algorithm to be optimal in access if no other implementation of the operation requires fewer accesses for some data base representation without requiring more accesses for some other data base representation.

Definition. An implementation $(\mathcal{A}_i, \rho)$ is *access optimal* if and only if for any other implementation $(\mathcal{A}_i', \rho)$:

$$(\forall d_1 \in \mathbb{D})[(\#[\mathcal{A}_i'(\rho(d_1))] < \#[\mathcal{A}_i(\rho(d_1))]) \Rightarrow (\exists d_2 \in \mathbb{D})(\#[\mathcal{A}_i'(\rho(d_2))] > \#[\mathcal{A}_i(\rho(d_2))])]$$

Similar to our result for Kraft storage, if $\mathcal{A}_i$ achieves Kraft access then $\mathcal{A}_i$ is access optimal.

Theorem 3.8. Suppose the $|\mathcal{B}|$-ary system $(\mathcal{A}, \rho)$ solves the static problem $(Q, \mathbb{D})$. If

$$\sum_{r \in q_i(\mathbb{D})} |\mathcal{B}|^{\alpha_i(r)} = 1$$

then $\mathcal{A}_i$ is access optimal.

Unless we allow the trivial question, which always returns the same value no matter what data base is stored in memory, then it is always necessary to make at least one access to answer a question.

Theorem 3.9. Given any implementation $(\mathcal{A}, \rho)$, assume that $\mathcal{A}_i(\rho(d))$ is not a constant function. Then, for all $d \in \mathbb{D}$,

$$\#[\mathcal{A}_i(\rho(d))] \geq 1.$$

Corollary 3.9.1. If $\#[\mathcal{A}_i(\rho(d))] = 1$ for all $d \in \mathbb{D}$, then $\mathcal{A}_i$ is access optimal.

If $|R| < |\mathcal{B}|$, then when we access one cell we can distinguish $|\mathcal{B}|$ characters, whereas we only have $|R|$ distinct answers. Therefore we have in some sense obtained more information than we can use, giving us an inequality in the Kraft sum, as the next theorem shows.

Theorem 3.10. Consider a $|\mathcal{B}|$-ary system $(\mathcal{A}, \rho)$ which answers the question $q: \mathbb{D} \to R$. If $\mathcal{A}$ achieves Kraft access, then $|R| \geq |\mathcal{B}|$.

Proof: Assume α meets Kraft with equality. Then by Theorem 3.9 it is always the case that $\alpha(r) \geq 1$, and so

$$1 = \sum_{r \in R} |\mathcal{B}|^{-\alpha(r)} \leq \sum_{r \in R} |\mathcal{B}|^{-1} = \frac{|R|}{|\mathcal{B}|}$$

If $|R| < |\mathcal{B}|$ then we get a contradiction. ∎

Notice that this theorem does not depend on the representation used.

Assume we have an implementation that achieves Kraft access for some set Q of questions. This then tells us something about the possible relative range sizes of questions in Q. We first recall a lemma about trees (see e.g. Knuth [14]).

> **Lemma 3.1.** There is a full $|\mathcal{B}|$-ary tree with k leaves if and only if there is some $n \in \mathbb{N}$ such that $k = (|\mathcal{B}| - 1) \cdot n + 1$. (The number n corresponds to the number of internal nodes in the tree.)

From this lemma and recalling that we have equality in the Kraft sum only when the exponents correspond to the depths of the leaves in a full tree, we have the following theorem.

> **Theorem 3.11.** (Gallager [12]). Let $f: \mathcal{J} \to \mathbb{N}$. If $\sum_{i \in \mathcal{J}} |\mathcal{B}|^{-f(i)} = 1$, then $|\mathcal{J}| = n \cdot (|\mathcal{B}| - 1) + 1$ for some $n \in \mathbb{N}$.

This now tells us something about the possible pairwise relative sizes of the ranges of questions that each achieve Kraft access.

> **Theorem 3.12.** Consider a $|\mathcal{B}|$-ary system (α, ρ) which answers the questions $q_1: \mathbb{D} \to R_1$ and $q_2: \mathbb{D} \to R_2$, and assume both q_1 and q_2 achieve Kraft access. Then there is some integer n such that $|R_1| - |R_2| = n \cdot (|\mathcal{B}| - 1)$.

Proof: Since both q_1 and q_2 achieve Kraft access,

$$\sum_{r \in R_1} |\mathcal{B}|^{-\alpha_1(r)} = 1 \qquad \text{and} \qquad \sum_{r \in R_2} |\mathcal{B}|^{-\alpha_2(r)} = 1$$

By Theorem 3.11 we thus know that there exist $n_1, n_2 \in \mathbb{N}$ such that

$$|R_1| = 1 + n_1(|\mathcal{B}| - 1)$$

and

$$|R_2| = 1 + n_2(|\mathcal{B}| - 1)$$

Therefore,

$$|R_1| - |R_2| = (n_1 - n_2)(|\mathcal{B}| - 1). \qquad \blacksquare$$

We find this theorem useful for some of the results we shall prove later.

As was the case when we discussed storage, it is difficult to understand what the Kraft inequality of Theorem 3.8 tells us about access costs of interest to the user, except when we actually do achieve Kraft access. Thus, we mention two results concerning access costs; these correspond to the storage theorems 3.4 and 3.5.

First, if we need to distinguish $|R_i|$ answers with a $|\mathcal{B}|$-ary tree, it is clear that the access tree must have maximum depth at least $\lceil \log_{|\mathcal{B}|} |R_i| \rceil$. Also, it is always possible to answer a question q_i in such a way that the corresponding access tree has maximum depth exactly $\lceil \log_{|\mathcal{B}|} |R_i| \rceil$.

Theorem 3.13. (Elias [6]). Consider a problem $(F, \mathbb{D})$.

(i) If the $|\mathcal{B}|$-ary system (α_i, ρ) answers the question q_i, then

$$\max_{r \in R_i} \alpha_i(r) \geq \lceil \log_{|\mathcal{B}|} |R_i| \rceil$$

(ii) There is some $|\mathcal{B}|$-ary system (α_i, ρ) that answers question q_i such that

$$\max_{r \in R_i} \alpha_i(r) = \lceil \log_{|\mathcal{B}|} |R_i| \rceil.$$

The bound in (ii) can be attained by using a representation ρ which stores in memory the answers to each question in Q. Thus, to answer q_i, α_i simply reads the i^{th} answer (see Elias and Flower [10]).

If there is some known probability distribution P on $\mathbb{D}$, this induces a probability distribution P_i on R_i defined by

$$P_i(r) = \sum_{d \in \mathbb{D}_i(r)} P(d)$$

where $\mathbb{D}_1(r) = \{d \in \mathbb{D} \mid q_1(d) = r\}$.

Theorem 3.14 (Elias [6]). **Consider a problem $(\Gamma, \mathbb{D})$, and assume there is a probability distribution P on $\mathbb{D}$. Define the entropy $H(R_1)$ by**

$$H(R_1) = -\sum_r P_1(r) \log_{|\mathcal{A}|} P_1(r).$$

(i) If the $|\mathcal{A}|$-ary system (Q_1, ρ) answers the question q_1, then

$$\operatorname*{avg}_{d \in \mathbb{D}} |\rho(Q_1(\rho(d)))| \geq H(R_1).$$

(ii) There is some $|\mathcal{A}|$-ary system (Q_1, ρ) that answers question q_1 such that

$$\operatorname*{avg}_{d \in \mathbb{D}} |\rho(Q_1(\rho(d)))| < H(R_1) + 1.$$

4.1 Definition

If for all i we know the i^{th} element in a list, then we have determined the list. Thus, in some sense this forms a complete set of questions on any domain $\mathbb{D}$, because answering these allows us to answer any other question.

Definition. Define the table lookup question set

$$\Gamma = \{\gamma_i \mid i \geq \max_{d \in \mathbb{D}} |d|\}$$

which has as its i^{th} member the function $\gamma_i : \mathbb{D} \to X$ defined by $\gamma_i(d) = d(i)$.

For $i > |d|$, we say $\gamma_i(d) = \Re$.

Thus, each data base $d \in \mathbb{D}$ is mapped onto the value of its i^{th} element. When $i > |d|$, then we want $\gamma_i(d)$ to return a null answer, which we denote by $\Re$.

Consider a system (α, ρ) solving $(\Gamma, \mathbb{D})$. As was mentioned in Section 3.3, if we say that γ_i achieves Krait access, we mean that α_i solving γ_i achieves Krait access. In general, although γ_i is defined in the problem domain, we may informally refer to γ_i in the machine domain; in particular, we say that $\gamma_i(\rho(d))$ accesses cell k to mean that $k \in \{[(\alpha_i(\rho(d)))]\}$.

CHAPTER 4

THE TABLE LOOKUP QUESTION SET

In the previous chapter we discussed what is meant by Kraft storage and access. In this chapter we shall examine more closely under what conditions Kraft storage and Kraft access can be achieved. In particular, we consider the table lookup question set and attempt to understand the implications of Kraft storage and access and to get a feel for some storage-access tradeoffs.

4.1 Definition

If for all i we know the i^{th} element in a list, then we have determined the list. Thus, in some sense this forms a complete set of questions on any domain $\mathbb{D}$, because answering these allows us to answer any other question.

Definition. Define the *table lookup question set*

$$\Gamma = \{\gamma_i \mid 1 \leq i \leq \max_{d \in \mathbb{D}} |d|\}$$

which has as its i^{th} member the function $\gamma_i : \mathbb{D} \to X$ defined by $\gamma_i(d) = d(i)$. For $i > |d|$, we say $\gamma_i(d) \triangleq \varnothing$.

Thus, each data base $d \in \mathbb{D}$ is mapped onto the value of its i^{th} element. When $i > |d|$, then we want $\gamma_i(d)$ to return a null answer, which we denote by $\varnothing$.

Consider a system $(\mathcal{A}, \rho)$ solving $(\Gamma, \mathbb{D})$. As was mentioned in Section 3.3, if we say that γ_i achieves Kraft access, we mean that $\mathcal{A}_i$ solving γ_i achieves Kraft access. In general, although γ_i is defined in the problem domain, we may informally refer to γ_i in the machine domain; in particular, we say that $\gamma_i(\rho(d))$ accesses cell k to mean that $k \in \{[\mathcal{A}_i(\rho(d))]\}$.

Example 4.1. Recall Example 2.3, where $\mathbb{D} = \{\lambda\} \cup X \cup X^2$ for $X = \{0,1\}$. Then we have, for instance,

$$\begin{aligned}
\gamma_1(d_0) &= \gamma_1(\lambda) = \varnothing = \gamma_2(d_0) \\
\gamma_1(d_1) &= \gamma_1(0) = 0 \\
\gamma_2(d_1) &= \gamma_2(0) = \varnothing \\
\gamma_1(d_6) &= \gamma_1(11) = 1 = \gamma_2(d_6).
\end{aligned}$$

Alternatively, we may informally write, using the representation ρ given in Example 2.7:

$$\begin{aligned}
\gamma_1(\rho(d_0)) &= \gamma_1(0\phi) = \varnothing = \gamma_2(\rho(d_0)) \\
\gamma_1(\rho(d_6)) &= \gamma_1(20\phi) = 1 = \gamma_2(\rho(d_6))
\end{aligned}$$ ∎

If we are going to achieve Kraft access for all questions in the table lookup question set, then for $|\mathcal{B}| > 2$ the ranges of all the questions must be the same.

Theorem 4.1. Let $\gamma_i, \gamma_j \in \Gamma$ and assume that $|\mathcal{B}| > 2$. If γ_i and γ_j both achieve Kraft access, then $R_i = R_j$, where $R_i = R(\gamma_i(\mathbb{D}))$.

Proof: Consider a table lookup question γ on $\rho:\mathbb{D} \to \mathcal{B}^+$. Since $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$, where $\mathcal{J} \subseteq \mathbb{N}$, then either $R(\gamma_i(\mathbb{D})) = X$ or $R(\gamma_i(\mathbb{D})) = X \cup \{\varnothing\}$. Suppose $|R_i| \neq |R_j|$. Then $|R_i| - |R_j| = \pm 1$. By Theorem 3.12 we know that $|R_i| - |R_j| = n \cdot (|\mathcal{B}| - 1) = \pm 1$, and so the only solution is for $|\mathcal{B}| = 2$, $n = \pm 1$. Thus, if $|\mathcal{B}| > 2$ we obtain a contradiction, proving that $|R_i| = |R_j|$, which implies that $R_i = R_j$. ∎

It is easy to show that the condition $|\mathcal{B}| > 2$ is necessary in the above theorem.

Example 4.2. Let $\mathbb{D} = X \cup X^2$ where $X = \{a,b\}$ and define the representation $\rho:\mathbb{D} \to \{0,1\}^+$ by:

$\rho(a) = 00\_$	$\rho(ab) = 011$
$\rho(b) = 10\_$	$\rho(ba) = 110$
$\rho(aa) = 010$	$\rho(bb) = 111$

Then the table lookup question set $\Gamma = \{\gamma_1, \gamma_2\}$ can be solved by algorithms with

access trees as shown in Figure 4.1. It is clear by observation of these trees that both γ_1 and γ_2 achieve Kraft access, and yet $R_1 = X$ whereas $R_2 = X \cup \{\varnothing\}$. ∎

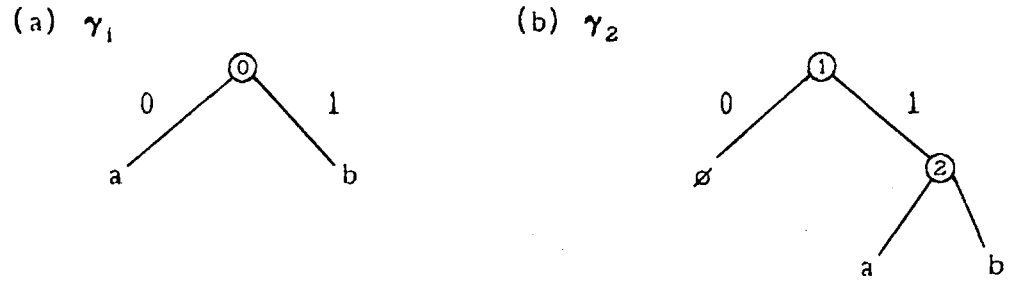


Figure 4.1. Access trees for γ_1 and γ_2 of Example 4.2.

It immediately follows from the previous theorem that if we have Kraft access for the set of table lookup questions, and $|\mathcal{B}| > 2$, then $\lambda \in \mathbb{D}$ except when $\mathbb{D} = X^n$ for some n.

Theorem 4.2. Let $|\mathcal{B}| > 2$. If all $\gamma_i \in \Gamma$ achieve Kraft access, then either $\lambda \in \mathbb{D}$ or else $\mathbb{D} = X^n$ for some $n \in \mathbb{N}^+$.

Proof: Let $\mathbb{D} \neq X^n$. Then there exist $d_1, d_2 \in \mathbb{D}$ such that $|d_1| < |d_2|$. So $\gamma_{|d_2|}(d_1) = \varnothing$ and $R(\gamma_{|d_2|}(\mathbb{D})) = X \cup \{\varnothing\}$. Now assume that $\lambda \notin \mathbb{D}$. Then $R(\gamma_1(\mathbb{D})) = X$. But by Theorem 4.1 this says that γ_1 and $\gamma_{|d_2|}$ can't both achieve Kraft access, a contradiction. Therefore $\lambda \in \mathbb{D}$. ∎

Thus, if $|\mathcal{B}| > 2$, then $\lambda \notin \mathbb{D}$ implies that $\mathbb{D} = X^n$ for some n. Because $R_i = X \cup \{\varnothing\}$, we know that if $\mathbb{D} \neq X^n$, then $\varnothing \in R_i$.

Corollary 4.2.1. Let $|\mathcal{B}| > 2$. If all $\gamma_i \in \Gamma$ achieve Kraft access and there is no $n \in \mathbb{N}^+$ such that $\mathbb{D} = X^n$, then $R(\gamma_i(\mathbb{D})) = X \cup \{\varnothing\}$, for all $\gamma_i \in \Gamma$.

4.2 Kraft Access with Overlapping Access Sets

In this section we discuss achieving Kraft access for the set of table lookup questions Γ and frequently refer to the set of memory cells accessed in order to answer some $\gamma_i \in \Gamma$.

Definition. Let ρ be a representation $\rho:\mathbb{D} \to \mathcal{B}^+$, and let $\gamma_i, \gamma_j \in \Gamma$. Then we say that γ_i and γ_j *have overlapping access sets* if, for some $d \in \mathbb{D}$,

$$\{[\gamma_i(\rho(d))]\} \cap \{[\gamma_j(\rho(d))]\} \neq \varnothing.$$

We shall show that, for $|\mathcal{B}| > 2$, if all $\gamma_i \in \Gamma$ achieve Kraft access then there can be no overlapping access sets (see Theorem 4.4). For the case $|\mathcal{B}| = 2$, two access sets $\{[\gamma_i(\rho(d))]\}$ and $\{[\gamma_j(\rho(d))]\}$ can overlap, but in at most one cell and only when $X^k \notin \mathbb{D}$, for all $i \leq k < j$ (see Theorem 4.8). Where all $\gamma_i \in \Gamma$ achieve Kraft access we also show that

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)| + |\Gamma| - 1,$$

and if the γ_i do not have overlapping access sets then

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)|.$$

(see corollaries 4.7.1 and 4.5.1.).

Consider any representation ρ and suppose that $\gamma_1, \gamma_2 \in \Gamma$ meet Kraft access. Our first theorem says that if $\gamma_1(\rho(d_1))$ and $\gamma_2(\rho(d_1))$ access some cell in common, then $\mathbb{D}$ does not include all strings of the form

$$d_1(i) \cdot R_2$$

or all strings

$$R_1 \cdot d_1(j).$$

Theorem 4.3. Consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ and let $\gamma_i, \gamma_j \in \Gamma$ each achieve Kraft access. Suppose there exists $d_1 \in \mathbb{D}$ such that $\gamma_i(\rho(d_1))$ and $\gamma_j(\rho(d_1))$ access some cell in common. Then

$$\neg(\forall r \in R_j)(\exists d_2 \in \mathbb{D})(d_2(i) = d_1(i) \text{ and } d_2(j) = r)$$

and $$\neg(\forall r \in R_i)(\exists d_2 \in \mathbb{D})(d_2(i) = r \text{ and } d_2(j) = d_1(j)).$$

Proof: For and $d_1 \in \mathbb{D}$, let $\rho(d_1) \subseteq m_1$. Suppose there is some $d_1 \in \mathbb{D}$ such that $\gamma_i(\rho(d_1))$ and $\gamma_j(\rho(d_1))$ both access cell k. Let $m_1(k) = b_1 \in \mathcal{B}$. Since $\gamma_i, \gamma_j \in \Gamma$ achieve Kraft access and access cell k then, for all $d_2 \in \mathbb{D}$,

$$d_2(i) = d_1(i) \Rightarrow m_2(k) = b_1$$
$$d_2(j) = d_1(j) \Rightarrow m_2(k) = b_1.$$

Since γ_j achieves Kraft access, we know there is some string $d_3 \in \mathbb{D}$ such that $m_3(k) \neq b_1$, and γ_j accesses cell k. So there is no way to represent a string d_4 where

$$d_4(i) = d_1(i) \text{ and } d_4(j) = d_3(j)$$

Similarly, there is no way to represent a string d_5 where

$$d_5(i) = d_3(i) \text{ and } d_5(j) = d_1(j).$$ ∎

The intuition behind the preceding theorem can perhaps best be seen by picturing the access trees for two table lookup questions, as we do in the following example. This gives us an example of overlapping storage, although we obviously can't represent all strings in the product of the ranges.

Example 4.3. Let $\mathcal{B} = \{0,1,2\}$, and let $X = \{x_i \mid 1 \leq i \leq 9\}$; i.e., $|\mathcal{B}| = 3$ and $|X| = 9$. Suppose γ_1 and γ_2 have the ternary access trees as shown in Figure 4.2 and therefore achieve Kraft access. These trees indicate that, for instance, $\rho(x_2 \cdot x_6) = 0121\_$ and $\rho(x_8 \cdot x_5) = 12202$. The only time we have overlapping access sets is for $d \in \mathbb{D}$ such that $d(1) = x_6$, x_7, or x_8; i.e., $\gamma_1(\rho(d)) = x_6$, x_7, or x_8. So we can certainly represent any pairs of strings in $x_i \cdot X$, where $x_i \notin \{x_6, x_7, x_8\}$. It is also possible to represent the pairs of strings $x_6 \cdot x_1$, $x_7 \cdot x_2$, and $x_8 \cdot x_j$ where $x_j \notin \{x_1, x_2\}$. ∎

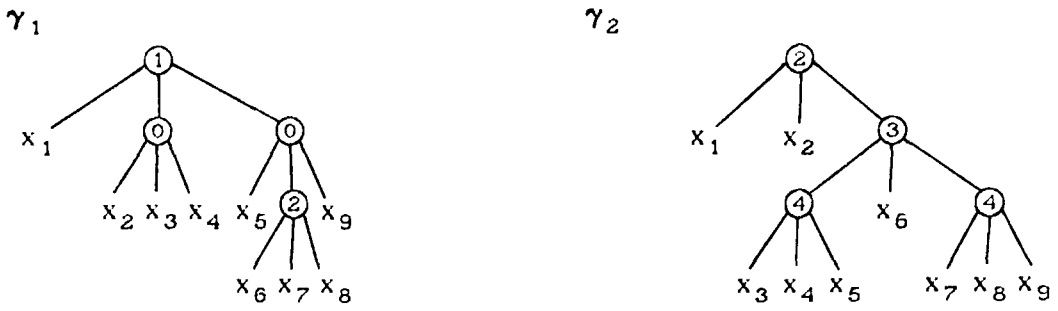


Figure 4.2. Access Trees for γ_1 and γ_2 of Example 4.3.

So if γ_i and γ_j do overlap in access of cell k, then it is not possible for $\mathbb{D}$ to include a string d_1, such that $\rho(d_1(i))$ has some value b_1 in cell k and $\rho(d_1(j))$ has some value $b_2 \neq b_1$ in cell k. If γ_i meets Kraft access, then its access tree is full, so there will be at least $|\mathcal{B}|$ elements $d \in \mathbb{D}$ such that $\gamma_i(\rho(d))$ accesses cell k. Similarly for γ_j. Let S be the set of strings in the domain that agree with d_1 in every position except the j^{th}:

$$S = \{d \in \mathbb{D} \mid d(n) = d_1(n) \text{ for all } n \neq j\}.$$

Then $|S| \leq |R_j| - (|\mathcal{B}| - 1)$, since there must be at least $|\mathcal{B}| - 1$ characters r in R_j such that we cannot represent any string in X^* whose i^{th} component is $d_1(i)$ and whose j^{th} component is r.

Lemma 4.1. Consider any representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ and let $\gamma_i, \gamma_j \in \Gamma$ achieve Kraft access. Suppose that for $d_1 \in \mathbb{D}$, γ_i and γ_j access some cell in common. Then

$$\left| \bigcup_{d \in \mathbb{D}} \{d(j)\} \right| \leq |R_j| - |\mathcal{B}| + 1.$$

Proof: $\gamma_i(\rho(d_1))$ and $\gamma_j(\rho(d_1))$ access some cell in common. Since γ_i meets Kraft access, then for $\rho(d_1) \subseteq m_1$, $\gamma_i(\rho(d_1))$ corresponds to $m_1(k) = b \in \mathcal{B}$. Since γ_j also meets Kraft access, there are at least $|\mathcal{B}| - 1$ values for $d(j)$ that do not have $m(k) = b$. Thus, $\left| \bigcup_{d \in \mathbb{D}} \{d(j)\} \right| \leq |R_j| - |\mathcal{B}| + 1$. ∎

Recall that for a pair of table lookup questions γ_i and γ_j, where $i < j$, then $R_i = R_j$ if γ_i and γ_j achieve Kraft access and $|\mathcal{B}| > 2$. If $d(i) = x \in X$, then all we know is that $d(j) \in X \cup \{\varnothing\}$. On the other hand, we know that if $d(i) = \varnothing$, then $d(j) = \varnothing$; in this case there are $|X|$ combinations of $d(i)$ and $d(j)$ that do not exist for any $d \in \mathbb{D}$. So perhaps there could be some representation scheme that would allow us to overlap accesses. The next theorem follows from Lemma 4.1 and shows that there is no such scheme.

Theorem 4.4. Consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$, where $|\mathcal{B}| > 2$, and let γ_i, $\gamma_j \in \Gamma$ each achieve Kraft access. Then, for all $d \in \mathbb{D}$, $\gamma_i(\rho(d))$ and $\gamma_j(\rho(d))$ access no cells in common.

Proof: Assume there exists $d_1 \in \mathbb{D}$ such that $\gamma_i(\rho(d_1))$ and $\gamma_j(\rho(d_1))$ each access cell k, $i < j$. Then since all γ_i achieve Kraft access, for all $b \in \mathcal{B}$ there is some $d_2 \in \mathbb{D}$ such that $\gamma_i(\rho(d_2))$ causes cell k to be accessed and $m_2(k) = b$, where $\rho(d_2) \subseteq m_2$. Since not all leaf descendants of node k in the access tree for γ_i can be labelled $\varnothing$, there is some $d_3 \in \mathbb{D}$ such that $d_3(i) \neq \varnothing$ and $\gamma_i(\rho(d_3))$ accesses cell k. If we let $\mathbb{D}_i = \{d \in \mathbb{D} \mid d(i) = d_3(i)\}$, then we have

$$\left| \bigcup_{d \in \mathbb{D}_i} \{d(j)\} \right| \leq |R_j| - |\mathcal{B}| + 1 \leq |X| + 1 - |\mathcal{B}| < |X|.$$

But by the way we have defined a problem domain, there are $|X|$ data bases $d \in \mathbb{D}$ that differ from d_3 only in the j^{th} position. This gives a contradiction and so, for all $d \in \mathbb{D}$, $\gamma_i(\rho(d))$ and $\gamma_j(\rho(d))$ do not have overlapping access sets. ∎

Since for any $d \in \mathbb{D}$ each γ_i accesses a distinct set of cells, the total number of accesses made by the various γ's cannot be more than $|\rho(d)|$.

Theorem 4.5. Consider any representation $\rho: \mathbb{D} \to \mathcal{B}^+$ and assume all $\gamma_i \in \Gamma$ achieve Kraft access. If γ_i and γ_j access no cells in common, then

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)|.$$

From theorems 4.4 and 4.5 we can immediately get the following result.

Corollary 4.5.1. Consider any representation ρ: $\mathbb{D} \to \mathcal{B}^+$, where $|\mathcal{B}| > 2$, and let all $\gamma_i \in \Gamma$ achieve Kraft access. Then

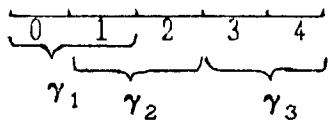
$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)|.$$

Unfortunately, Theorem 4.4 does not hold for $|\mathcal{B}| = 2$. In other words, it is possible for γ_i and γ_j to achieve Kraft access and also access some cell in common.

Example 4.4. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b\}$, and $\mathbb{D} = \{\lambda\} \cup X^2 \cup X^3$. Consider the representation ρ:$\mathbb{D} \to \mathcal{B}^+$ defined as follows:

d	$\rho(d)$
λ	10_0_
aa	0100_
ab	0110_
ba	1100_
bb	1110_
aaa	01010
aab	01011
aba	01110
abb	01111
baa	11010
bab	11011
bba	11110
bbb	11111

Since $\lambda \in \mathbb{D}$, $R_i = \{a,b,\varnothing\}$ for $i \in \{1,2,3\}$. Possible access trees for γ_1, γ_2, γ_3 are shown in Figure 4.3. Notice that γ_1 and γ_2 may both access cell 1, and we have the following storage allocation:



Without altering the access trees, we could extend ρ and $\mathbb{D}$ so as to also include the element $a \in X$, by letting $\rho(a) = 00\_0$. It would not, however, be possible to similarly include b in the domain, because $\rho(b)$ would require cell 1 to be set to 1

and also to 0.

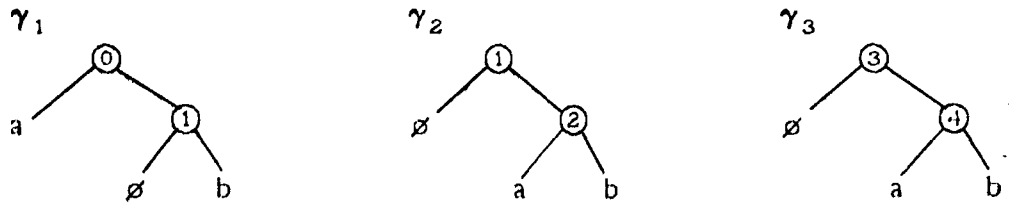


Figure 4.3. Access trees for γ_1, γ_2, γ_3 of Example 4.4.

We can see that Corollary 4.5.1 does not hold for $|\mathcal{B}| = 2$, since for d = bab, ρ(bab) = 11011 and:

$$\sum_{i=1}^{3} \#[\gamma_i(11011)] = 2 + 2 + 2 = 6 > |\rho(\text{bab})|.$$

Notice also that ρ does not achieve Kraft storage:

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = 4 \cdot 2^{-4} + 2^{-3} + 8 \cdot 2^{-5} = \frac{5}{8} < 1. \qquad \blacksquare$$

The following lemma shows for $|\mathcal{B}| = 2$ that if γ_i and γ_j each have Kraft access, and if they both access cell k, then the access trees for γ_i and γ_j each have a node labelled k leading to a leaf ø via a branch labelled b $\in \mathcal{B}$.

Lemma 4.2. Let $|\mathcal{B}| = 2$ and let b, b′ $\in \mathcal{B}$, b $\neq$ b′. Consider a representation $\rho: \mathbb{D} \to \mathcal{B}^+$, and assume that γ_i, $\gamma_j \in \Gamma$ achieve Kraft access and that

$$k \in \Big(\bigcup_{d \in \mathbb{D}} \{[\gamma_i(\rho(d))]\} \cap \bigcup_{d \in \mathbb{D}} \{[\gamma_j(\rho(d))]\} \Big).$$

Choose elements $x_1, x_2 \in R_i$ and $x_3, x_4 \in R_j$, such that $m_1(k)$ = b, $m_2(k)$ = b′, $m_3(k)$ = b, $m_4(k)$ = b′, where $m_i \supseteq \bar{\rho}_L(x_i)$. Then either $x_1 = x_3 =$ ø or $x_2 = x_4 =$ ø.

Proof: Clearly ρ cannot represent a string d_1 where $d_1(i) = x_1$ and $d_1(j) = x_4$ or a string d_2 where $d_2(i) = x_2$ and $d_2(j) = x_3$. There are two cases to consider:
(i) If $x_1 \in X$ then $x_4 =$ ø, since we do not necessarily need to represent $d(i) \in X$

and $d(j) = \varnothing$, but we must be able to represent $d(i) \in X$ and $d(j) \in X$. This tells us that $x_3 \neq \varnothing$ and so $x_3 \in X$. Since we cannot represent d_1, then $x_2 = \varnothing$.
(ii) If $x_1 = \varnothing$, then $x_4 \in X$ and $x_2 \in X$. Since we cannot represent d_1, then $x_3 = \varnothing$. ∎

In Example 4.4, the access sets for γ_1 and γ_2 each included the cell 1. Notice that in each of their access trees, the left branch from the node labelled 1 led to the leaf $\varnothing$; using the terminology of Lemma 4.2, $x_1 = x_3 = \varnothing$.

Lemma 4.2 allows us to prove that at most one cell can be in two access sets, if we achieve Kraft access.

Theorem 4.6. Assume $\gamma_i, \gamma_j \in \Gamma$ achieve Kraft access. Then the access sets for γ_i and γ_j contain at most one cell in common.

Proof: If γ_i and γ_j access two cells in common then by Lemma 4.2 each tree has two leaves $\varnothing$, which violates our assumption of Kraft access. ∎

We can, in fact, make the even stronger statement that if we achieve Kraft access for all of Γ then any table lookup question $\gamma_i \in \Gamma$ can access only one cell that any other $\gamma_j \in \Gamma$ accesses. The following theorem formalizes this.

Theorem 4.7. Consider a representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$, and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. If γ_i, γ_j both access cell k_1 and γ_i, γ_k both access cell k_2, then $k_1 = k_2$.

Proof: By Lemma 4.2, we know that node k_1 in γ_i's access tree leads to a leaf labelled $\varnothing$. But also node k_2 in the tree for γ_i must lead to a leaf $\varnothing$. Since γ_i achieves Kraft access, there can be at most one leaf labelled $\varnothing$, and so $k_1 = k_2$. ∎

This gives us a result similar to Theorem 4.5, for the case where we allow access

overlap.

Corollary 4.7.1. Consider any representation $\rho:\mathbb{D} \to \mathcal{B}^{\dagger}$ and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. If we allow access overlap, then

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)| + |\Gamma| - 1.$$

Proof: From Theorem 4.5 we recall that where there is no access overlap, then

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)|.$$

Now from Theorem 4.7 we know that each γ_i can have at most one cell in common with any other γ_j. So

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)| + |\Gamma| - 1. \quad \blacksquare$$

Example 4.5. Recall Example 4.4, where

$$\sum_{i=1}^{3} \#[\gamma_i(\rho(\text{bab}))] = 2 + 2 + 2 = 6 \leq |\rho(\text{bab})| + |\Gamma| - 1 = 5 + 3 - 1 = 7. \quad \blacksquare$$

The next example verifies that, in fact, the bound in the above corollary is the best possible. We achieve this bound when all $\gamma_i \in \Gamma$ access some cell in common.

Example 4.6. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b\}$, and $\mathbb{D} = \{\lambda\} \cup X^3$. Consider the representation $\rho:\mathbb{D} \to \mathcal{B}^{\dagger}$ defined as follows:

d	$\rho(d)$
λ	10_0
aaa	0101
aab	0100
aba	0111
abb	0110
baa	1101
bab	1100
bba	1111
bbb	1110

Consider the access trees for γ_1, γ_2, γ_3 shown in Figure 4.4. Then it is easy to see that

$$\sum_{i=1}^{3} \#[\gamma_i(\rho(d))] \leq |\rho(d)| + |\Gamma| - 1.$$

In particular,

$$\sum_{i=1}^{3} \#[\gamma_i(\rho(\lambda))] = 5 \leq 3 + 3 - 1$$

and

$$\sum_{i=1}^{3} \#[\gamma_i(\rho(bba))] = 6 \leq 4 + 3 - 1. \quad \blacksquare$$

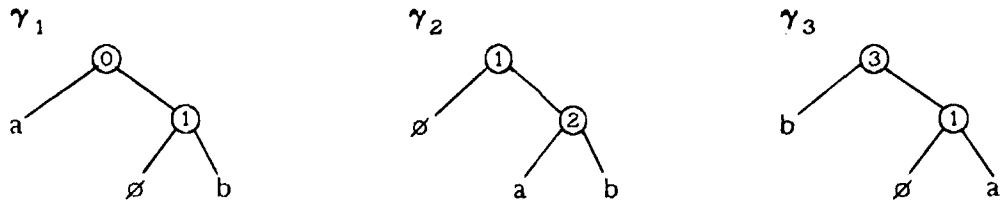


Figure 4.4. Access trees for γ_1, γ_2, γ_3 of Example 4.6.

Essentially, we were able to allow access overlap in Example 4.4 because we did not need to represent the strings $a\varnothing\varnothing$ or $b\varnothing\varnothing$. This was because we restricted $\mathbb{D}$ so that $X^1 \notin \mathbb{D}$. If it is necessary, however, to represent the situation where $\gamma_i(\rho(d)) \neq \varnothing$ and $\gamma_j(\rho(d)) = \varnothing$, then no overlap between γ_1 and γ_2 is possible. In fact, for $|\mathcal{B}| = 2$, this works in both directions, as the next theorem shows.

Theorem 4.8. Let $|\mathcal{B}| = 2$ and let $\gamma_i, \gamma_j \in \Gamma$ each achieve Kraft access. There exists a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ such that γ_i and γ_j access some cell in common if and only if $X^k \notin \mathbb{D}$ for all $i \leq k < j$.

Proof: ($\Rightarrow$) As in the proof of Theorem 4.4, we can assume without loss of generality that $\gamma_i(\rho(d_1)) \neq \varnothing$. Then $|R(\gamma_j(d_1))| = |X| + 1$. But by Lemma 4.1, if γ_i and γ_j access some cell in common, then $\gamma_j(\rho(d))$ can take on at most

$$|R_j| - |\mathcal{B}| + 1 \leq |X| + 2 - |\mathcal{B}| \leq |X| < |X| + 1$$

values. So γ_i and γ_j access no cells in common.

($\Leftarrow$) If there exists no k such that $i < k \leq j$, then

$$\gamma_i(\rho(d)) \in X \Rightarrow \gamma_j(\rho(d)) \in X$$

and $$\gamma_i(\rho(d)) = \varnothing \Rightarrow \gamma_j(\rho(d)) = \varnothing.$$

We can always construct a representation ρ such that γ_i and γ_j will both access some cell k. Let the access tree for γ_i have exactly one node corresponding to an access of cell k, and let this node be at a greater depth than any other nonleaf node. Let the left branch from this node lead to a leaf labeled $\varnothing$ and the right branch lead to some other leaf $x_1 \in X$. Then construct the access tree for γ_j such that the root is labeled k, and its left branch leads directly to a leaf labeled $\varnothing$. This allows us to represent all strings $X \cdot X$ and $\varnothing \cdot \varnothing$, and yet γ_i and γ_j both access cell k. ∎

4.3 Achieving Kraft Storage and Kraft Access

We have seen in Example 4.4 that it is possible to have Kraft access with overlapping access sets, although that particular representation did not achieve Kraft storage. This leads us to wonder whether it is even possible to achieve both Kraft storage and Kraft access; the following example shows us that it is.

Example 4.7. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b\}$, $\mathbb{D} = \{\lambda\} \cup X^2$, and define $\rho:\mathbb{D} \to \mathcal{B}^+$ by

d	$\rho(d)$
λ	0
aa	100
ab	101
ba	110
bb	111

Now consider the access trees for γ_1 and γ_2 as shown in Figure 4.4. Clearly γ_1 and γ_2 each achieve Kraft access. It is also the case, however, that ρ achieves Kraft storage, since

$$\sum_{d \in \mathbb{D}} 2^{-|\rho(d)|} = 2^{-1} + 4 \cdot 2^{-3} = 1$$

Now notice that

$$\sum_{i=1}^{2} \#[\gamma_i(\rho(ab))] = \#[\gamma_1(\rho(ab))] + \#[\gamma_2(\rho(ab))] = 2 + 2 = 4 > |\rho(d)|$$

and so Corollary 4.5.1 does not hold for $|\mathcal{B}| = 2$, even when we achieve both Kraft storage and Kraft access. ∎

The main results of this section, theorems 4.9 and 4.10, tell us that if we achieve both Kraft storage and Kraft access then our domain must be of the form $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X^n$.

We are now in a position to prove our first of two main results of this section: if we are to have Kraft storage and access and not allow overlapping access sets, then $\mathbb{D} = X^n$. We first prove the following lemma.

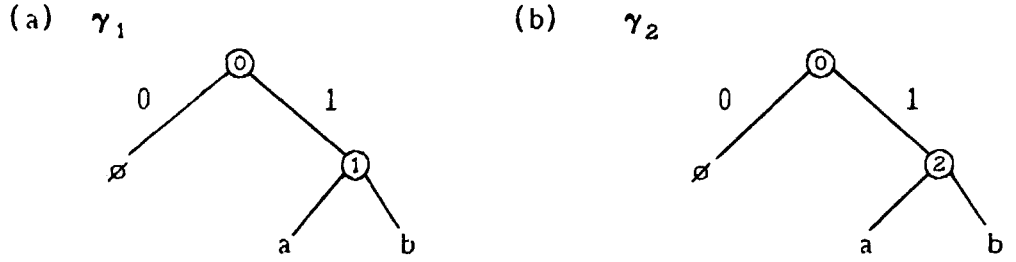


Figure 4.4. Access trees corresponding to γ_1 and γ_2 of Example 4.7.

Lemma 4.3. Consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. Then, for $k \leq |\Gamma|$,

$$\sum_{s \in R^k} |\mathcal{B}|^{-\sum_{i=1}^{k} \#[\gamma_i(\rho(s))]} = 1, \tag{4.1}$$

where $R^k \triangleq R_1 \circ R_2 \circ \ldots \circ R_k$.

Proof: We prove this result by induction on k.

Basis: Since γ_1 achieves Kraft access, by Theorem 3.7 we have

$$\sum_{s \in R^1} |\mathcal{B}|^{-\#[\gamma_1(\rho(s))]} = \sum_{s \in R^1} |\mathcal{B}|^{-\alpha_1(s)} = 1.$$

Induction step: Let $R_{k+1} = \{r_1, r_2, \ldots, r_n\}$, and assume that (4.1) holds for R^k. Then

$$\begin{aligned}\sum_{s \in R^{k+1}} |\mathcal{B}|^{-\sum_{i=1}^{k+1} \#[\gamma_i(\rho(s))]} &= \sum_{s \in R^k \cdot r_1} |\mathcal{B}|^{-\sum_{i=1}^{k} \#[\gamma_i(\rho(s))] - \#[\gamma_{k+1}(\rho(s))]} \\ &+ \sum_{s \in R^k \cdot r_2} |\mathcal{B}|^{-\sum_{i=1}^{k} \#[\gamma_i(\rho(s))] - \#[\gamma_{k+1}(\rho(s))]} \\ &+ \ldots + \sum_{s \in R^k \cdot r_n} |\mathcal{B}|^{-\sum_{i=1}^{k} \#[\gamma_i(\rho(s))] - \#[\gamma_{k+1}(\rho(s))]}\end{aligned}$$

Since γ_{k+1} achieves Kraft access, then for $r \in R^k$ and $r \in R$, we have

$$\#[\gamma_{k+1}(\rho(r \cdot r_i))] = \alpha_{k+1}(r_i)$$

This gives us

$$\sum_{s\in R^{k+1}} |\mathcal{B}|^{-\sum_{i=1}^{k}\#[\gamma_i(\rho(s))]} = |\mathcal{B}|^{-\alpha_{k+1}(r_1)} \sum_{s\in R^k} |\mathcal{B}|^{-\sum_{i=1}^{k}\#[\gamma_i(\rho(s))]} + |\mathcal{B}|^{-\alpha_{k+1}(r_2)} \sum_{s\in R^k} |\mathcal{B}|^{-\sum_{i=1}^{k}\#[\gamma_i(\rho(s))]} + \ldots + |\mathcal{B}|^{-\alpha_{k+1}(r_n)} \sum_{s\in R^k} |\mathcal{B}|^{-\sum_{i=1}^{k}\#[\gamma_i(\rho(s))]}$$

By our inductive assumption and since we are given that γ_{k+1} achieves Kraft access for $k + 1 \leq |\Gamma|$, this becomes:

$$\sum_{s\in R^{k+1}} |\mathcal{B}|^{-\sum_{i=1}^{k+1}\#[\gamma_i(\rho(s))]} = |\mathcal{B}|^{-\alpha_{k+1}(r_1)} + |\mathcal{B}|^{-\alpha_{k+1}(r_2)} + \ldots + |\mathcal{B}|^{-\alpha_{k+1}(r_n)} = 1. \quad \blacksquare$$

We now prove our desired theorem.

Theorem 4.9. Consider a representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$ which achieves Kraft storage and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. If for all $\gamma_i, \gamma_j \in \Gamma$

$$\bigcup_{d\in\mathbb{D}} \{[\gamma_i(\rho(d))]\} \cap \bigcup_{d\in\mathbb{D}} \{[\gamma_j(\rho(d))]\} = \varnothing,$$

then $\mathbb{D} = X^n$.

Proof: Let $R_i^{|\Gamma|}$ denote the set of strings of length $|\Gamma|$, where each element is chosen from R_i. Define the one-to-one function $g:\mathbb{D} \to R_i^{|\Gamma|}$ by

$$g(d) = \gamma_1(d)\cdot\gamma_2(d)\cdot\ldots\cdot\gamma_{|\Gamma|}(d).$$

Assume that $\lambda \in \mathbb{D}$. Then, by Corollary 4.2.1, $R(\gamma_i(\mathbb{D})) = X \cup \{\varnothing\}$ for all i. But for all $\gamma_i \in \Gamma$, $\gamma_1(\rho(d)) = \varnothing$ implies that $\gamma_i(\rho(d)) = \varnothing$. So $g(\mathbb{D}) \neq R_i^{|\Gamma|}$ since, e.g., $\varnothing\cdot X^{|\Gamma|-1} \notin g(\mathbb{D})$. Because we have Kraft access and no overlapping access sets,

$$\sum_{d\in\mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} \leq \sum_{d\in\mathbb{D}} |\mathcal{B}|^{-\sum_{i=1}^{|\Gamma|}\#[\gamma_i(\rho(d))]} \quad \text{by Theorem 4.5}$$

$$= \sum_{s\in g(\mathbb{D})} |\mathcal{B}|^{-\sum_{i=1}^{|\Gamma|}\#[\gamma_i(\rho(s))]} \quad \text{since g is 1-1}$$

$$< \sum_{s \in R_i^{|\Gamma|}} |\mathcal{B}|^{-\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(s))]} \qquad \text{since } g(\mathbb{D}) \subset R_i^{|\Gamma|}$$

$$= 1 \qquad \text{by Lemma 4.3.}$$

This gives a contradiction, since we know that ρ achieves Kraft storage. So $\lambda \neq \mathbb{D}$, which by Theorem 4.2 says that $\mathbb{D} = X^n$. ∎

As we saw in Example 4.7, the condition that there be no access set overlap is necessary in the above theorem.

From theorems 4.9 and 4.4, we have the following corollary.

Corollary 4.9.1. Let $|\mathcal{B}| > 2$, and consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage. If all $\gamma_i \in \Gamma$ achieve Kraft access, then $\mathbb{D} = X^n$.

Because we shall frequently consider domains of the form $\mathbb{D} = \bigcup_{i=0}^{k} X^i$, it is worth noting that with a domain in this form, it is not possible to attain both Kraft storage and Kraft access.

Corollary 4.9.2. Let $\mathbb{D} = \bigcup_{i=0}^{k} X^i$, for $k > 0$, and consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$. Assume that all $\gamma_i \in \Gamma$ achieve Kraft access. Then ρ does not achieve Kraft storage.

Although we have proved that Kraft access, Kraft storage, and no access set overlap implies that $\mathbb{D} = X^n$, we know by Example 4.7 that it is also possible to have, for some domain $\mathbb{D} \neq X^n$, both Kraft storage and access with access overlap. Example 4.7 is not an isolated case; i.e., the next example illustrates that it is not necessary that $|X| = 2$ or that $\mathbb{D} = \{\lambda\} \cup X^2$.

Example 4.8. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b,c,d\}$, $\mathbb{D} = \{\lambda\} \cup X^3$, and define $\rho:\mathbb{D} \to \mathcal{B}^+$ as indicated in Figure 4.5. Such a definition is possible because only cell 0 is in two access sets, and $m(0) = 1$ for all $d \in \mathbb{D}$ except $d = \lambda$. For instance,

d	$\rho(d)$
λ	0
aaa	10_ _0011
aba	10_ _0111
aca	10_ _1011
ada	10_ _1111
bac	110_0000
bbc	110_0100
bcc	110_1000
bdc	110_1100
cad	11100010
cbd	11100110
ccd	11101010
cdd	11101110

This system has overlapping access sets and achieves Kraft access. In fact, we also have Kraft storage, since

$$\sum_{d \in \mathbb{D}} 2^{-|\rho(d)|} = 2^{-1} + 4^2 \cdot 2^{-6} + 4^2 \cdot 2^{-7} + 2 \cdot 4^2 \cdot 2^{-8} = 1. \quad \blacksquare$$

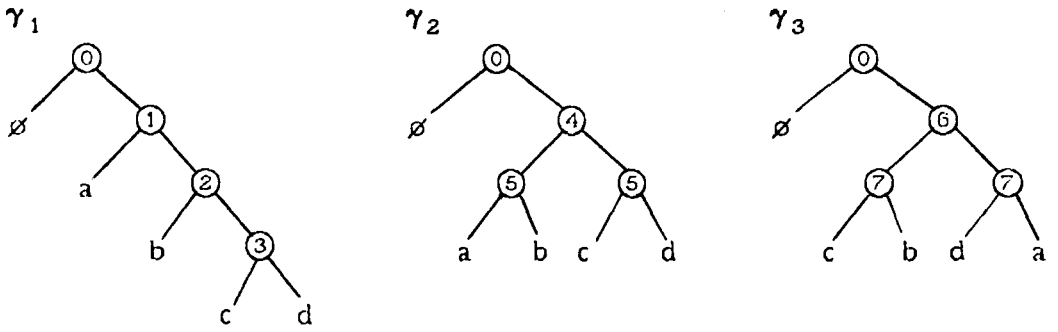


Figure 4.5. Access trees corresponding to γ_1, γ_2, and γ_3 of Example 4.8.

Now we want to determine for what possible domains $\mathbb{D}$ we can get Kraft storage and access if we allow overlapping access sets. Certainly we know that $|\mathcal{B}| = 2$, and recalling examples 4.7 and 4.8 we might suppose that $\mathbb{D}$ is of the form $\{\lambda\} \cup X^n$, as is indeed the case.

Lemma 4.4. Let $|\mathcal{B}| = 2$ and consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage. Assume that $|\Gamma| > 1$ and $\gamma_i, \gamma_j \in \Gamma$ achieve Kraft access, and

$$\bigcup_{d \in \mathbb{D}} \{[\gamma_i(\rho(d))]\} \cap \bigcup_{d \in \mathbb{D}} \{[\gamma_j(\rho(d))]\} = \{k\}.$$

Then the access trees for γ_i and γ_j each have root node labelled k.

Proof: Let $i < j$. By Lemma 4.2, we know that $\varnothing \in R_i$ and $\varnothing \in R_j$. Assume that the access tree for γ_i has root with label $t_1 \neq k$ and that the access tree for γ_j has root t_2. Without loss of generality, let the leaf in tree γ_i with label $\varnothing$ have $t_1 = 0$; i.e., $\rho(\varnothing)$ has $m(t_1) = 0$. Since $t_1 \neq k$ the node k must be a descendant of t_1, and there exists $x_1 \in X$ such that $\rho(x_1)$ also has $m(t_1) = 0$. Clearly there is some $x_2 \in X$ such that $\rho(x_2)$ has $m(t_1) = 1$. From Lemma 4.2 we know that in the tree γ_j we must also have the $\varnothing$ leaf a descendant of node k, with $m(k) = 0$. Thus $d_1 \notin \mathbb{D}$, where $d_1(i) = x_1$ and $d_1(j) = \varnothing$ since ρ would require setting $m(k) = 1$ and $m(k) = 0$. Since γ_i, γ_j achieve Kraft access, then by Theorem 4.8 it must be the case that $X^p \notin \mathbb{D}$ for $i \leq p < j$. On the other hand, we know that ρ does achieve Kraft storage. So by Theorem 3.3 there is some $d_2 \in \mathbb{D}$ such that $d_2(i) = x_2$ and $d_2(j) = \varnothing$, which contradicts the fact that $X^p \notin \mathbb{D}$ for $i \leq p < j$. Thus, $t_1 = k$ and we can similarly show that $t_2 = k$. ∎

This lemma allows us to prove our second main result of the section: If we have Kraft access, Kraft storage, and access overlap, then $\mathbb{D} = \{\lambda\} \cup X^n$.

Theorem 4.10. Consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage, and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. If there exist $\gamma_i, \gamma_j \in \Gamma$ such that γ_i and γ_j have overlapping access sets, then $\mathbb{D} = \{\lambda\} \cup X^n$.

Proof: Assume γ_i and γ_j both access cell k, and assume there is some $\gamma_m \in \Gamma$ that does not access cell k. Then we can represent $d_1(i) = \varnothing$, $d_1(j) = \varnothing$, $d_1(m) \in X$, indicating we don't have Kraft storage, a contradiction. So if γ_i and γ_j both access cell k, then for all $\gamma_m \in \Gamma$, γ_m accesses cell k. By Lemma 4.4, γ_m has root node k with one branch to leaf $\varnothing$. Thus, we can represent exactly the strings $\varnothing^{|\Gamma|}$ and $X^{|\Gamma|}$, and so $\mathbb{D} = \{\lambda\} \cup X^n$. ∎

In Theorem 4.5 we showed that if we meet Kraft access and have no access set overlap, then $|\rho(d)|$ is an upper bound on the total number of accesses made in reading all the elements in $\rho(d)$. We now show that for any $|\mathcal{B}| \geq 2$, if we achieve Kraft storage then every cell must be accessed in answering some question γ_i. Thus $|\rho(d)|$ is a lower bound on the total number of accesses to read $\rho(d)$.

Theorem 4.11. If the representation $\rho:\mathbb{D} \to \mathcal{B}^+$ achieves Kraft storage, then for all $d \in \mathbb{D}$:

$$k \in D(\rho(d)) \Rightarrow k \in \bigcup_{\gamma_i \in \Gamma} \{[\gamma_i(\rho(d))]\}$$

Proof: We define S to be the set of cells accessed by asking of some $d_1 \in \mathbb{D}$ each of the questions γ_i: $S = \bigcup_{\gamma_i \in \Gamma} \{[\gamma_i(\rho(d_1))]\}$. We want to prove that

$$k \in D(\rho(d_1)) \Rightarrow k \in S.$$

Define the representation $\rho_1:\mathbb{D} \to \mathcal{B}^+$ by:

$$\rho_1(d) = \begin{cases} \rho(d) & \text{for } d \neq d_1 \\ \{(k, m(k)) \mid k \in S\} & \text{for } d = d_1 \end{cases}$$

Then ρ_1 is a representation because ρ is: for $d_2, d_3 \in \mathbb{D}$ where $d_2 \neq d_1$, $d_3 \neq d_1$, we have

$$\bar{\rho}_1(d_2) \cap \bar{\rho}_1(d_3) \neq \varnothing \Rightarrow \bar{\rho}(d_2) \cap \bar{\rho}(d_3) \neq \varnothing \Rightarrow d_2 = d_3,$$

and for $d_2 \in \mathbb{D}$, $d_2 \neq d_1$, we have

$$\bar{\rho}_1(d_2) \cap \bar{\rho}_1(d_1) \neq \varnothing \Rightarrow (\forall \gamma_i \in \Gamma)(\gamma_i(\rho(d_1)) = \gamma_i(\rho(d_2))) \Rightarrow d_2 = d_1.$$

Assume there exists $k \in D(\rho(d_1))$ such that $k \notin S$. Then $|\rho_1(d_1)| = |S| < |\rho(d_1)|$ and

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho_1(d)|} = \sum_{d \in \mathbb{D}-\{d_1\}} |\mathcal{B}|^{-|\rho_1(d)|} + |\mathcal{B}|^{-|\rho_1(d_1)|}$$

$$> \sum_{d \in \mathbb{D}-\{d_1\}} |\mathcal{B}|^{-|\rho(d)|} + |\mathcal{B}|^{-|\rho(d_1)|}$$

$$= \sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = 1$$

This violates the fact that ρ achieves Kraft storage, so $k \in D(\rho(d_1)) \Rightarrow k \in S$. ∎

Corollary 4.11.1. If the representation $\rho:\mathbb{D} \to \mathcal{B}^+$ achieves Kraft storage, then for all $d \in \mathbb{D}$:

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \geq |\rho(d)|.$$

From theorems 4.5 and 4.11, we immediately have the following result.

Theorem 4.12. Consider a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. If there is no access set overlap, then for all $d \in \mathbb{D}$:

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] = |\rho(d)|.$$

Since we are in general considering list problems where $\mathbb{D} = \bigcup_{i=0}^{n} X^i\}$, Theorem 4.9 holds for the cases of particular interest to us.

Corollary 4.12.1. If $\mathbb{D} = \bigcup_{i=0}^{n} X^i$ and the representation $\rho:\mathbb{D} \to \mathcal{B}^+$ achieves Kraft storage, and all $\gamma_i \in \Gamma$ achieve Kraft access, then for all $d \in \mathbb{D}$:

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] = |\rho(d)|.$$

Thus, for list problems where $\mathbb{D} = \bigcup_{i=0}^{n} X^i$, if all $\gamma_i \in \Gamma$ achieve Kraft access then γ_i and γ_j access no cells in common.

4.4 Storage Consequences of Kraft Access

We conclude this chapter by examining some consequences of Kraft access for the set of table lookup questions. In particular, achieving Kraft access tells us something about the minimum and maximum possible values of $|\rho(d)|$:

$$\max_{d \in \mathbb{D}} |\rho(d)| > |\Gamma| \cdot (\lceil \log_{|\mathcal{B}|} |R| \rceil - 1)$$

and

$$\min_{d \in \mathbb{D}} |\rho(d)| \geq |J| - 1.$$

In general we have even better bounds.

In order to lower bound $|\rho(d)|$, we first prove two lemmas.

Lemma 4.5. Let $\rho: \mathbb{D} \rightarrow \mathcal{B}^\dagger$ be any representation. Then

$$(\forall \gamma_i \in \Gamma)(\exists d \in \mathbb{D})(\#[\gamma_i(\rho(d))] \geq \lceil \log_{|\mathcal{B}|} |R_i| \rceil).$$

Proof: By Theorem 3.13, $\max_{r \in R_i} \alpha_i(r) \geq \lceil \log_{|\mathcal{B}|} |R_i| \rceil$

so $$\max_{d \in \mathbb{D}} \#[\gamma_i(\rho(d))] \geq \lceil \log_{|\mathcal{B}|} |R_i| \rceil,$$

and this immediately gives our desired result. ∎

Lemma 4.6. Let $\mathbb{D} = \bigcup_{i \in J} X^i$ and let $\rho: \mathbb{D} \rightarrow \mathcal{B}^\dagger$ be any representation. Then

$$(\exists d_1 \in \mathbb{D})(\forall \gamma_i \in \Gamma)(\#[\gamma_i(\rho(d_1))] \geq \lceil \log_{|\mathcal{B}|} |R_i| \rceil).$$

Proof: Let $d_1 \in \mathbb{D}$ be the database defined as follows:

$$d_1 \triangleq \{d_1(i) = r_i \mid (r_i \in X) \wedge (\alpha_i(r_i) = \max_{r \in R_i} \alpha_i(r)) \wedge (0 \geq i \leq |\Gamma|)\}$$

It is always possible to define such a d_1. Now recalling Theorem 3.13,

$$\#[\gamma_i(\rho(d_1))] = \max_{r \in R_i} \alpha_i(r) \geq \lceil \log_{|\mathcal{B}|} |R_i| \rceil.$$ ∎

We now show it is always the case that

$$\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil - |\Gamma| + 1$$

and almost always the case that

$$\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil.$$

Theorem 4.13. Let $\mathbb{D} = \bigcup_{i \in J} X^i$ and $\rho:\mathbb{D} \to \mathcal{B}^+$ be any representation. Assume that all $\gamma_i \in \Gamma$ achieve Kraft access. Then we can conclude the following, where we write $|R|$ to denote $\min_{i \in J} |R_i|$.

(a) $\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot (\lceil \log_{|\mathcal{B}|} |R| \rceil - 1) + 1$

(b) If there are no overlapping access sets for $\gamma_i \in \Gamma$ or if there is no $j \in \mathbb{N}^+$ such that $|X| = 2^j$, then

$$\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil.$$

Proof: (a) By Corollary 4.7.1,

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \geq |\rho(d)| + |\Gamma| - 1.$$

From Lemma 4.6, there exists a $d_1 \in \mathbb{D}$ such that

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d_1))] \geq \sum_{i=1}^{|\Gamma|} \lceil \log_{|\mathcal{B}|} |R_i| \rceil.$$

Combining these, we get

$$\begin{aligned} |\rho(d_1)| &\geq \sum_{i=1}^{|\Gamma|} \lceil \log_{|\mathcal{B}|} |R_i| \rceil - |\Gamma| + 1 \\ &\geq |\Gamma| \cdot (\lceil \log_{|\mathcal{B}|} |R| \rceil - 1) + 1 \end{aligned}$$

and so $$\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil - 1) + 1$$

(b) (i) If there are no overlapping access sets, then Theorem 4.5 tells us that

$$\sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \leq |\rho(d)|,$$

and so we conclude that

$$\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil.$$

(ii) If we do have overlapping access sets, then by Theorem 4.4 we know that $|\mathcal{B}| = 2$ and by Lemma 4.2 we know $|R| = |X| + 1$. Assume there exists $j \in \mathbb{N}^+$ such that $2^j < |X| < 2^{j+1}$. So in each γ_i tree there is some $x_i \in X$ which labels a leaf at depth $j + 1$. Now define $d_1 \in \mathbb{D}$ so that $d_1(i) = x_i$ for all $1 \leq i \leq |\Gamma|$. Then

$$|\rho(d_1)| \geq (j + 1) \cdot |\Gamma|.$$

Since $$\lceil \log_{|\mathcal{B}|} |R| \rceil = j + 1,$$

we have $$\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil.$$ ∎

In the following example we verify that if we allow overlapping access sets for $|X| = 2^J$, then we may have

$$\max_{d \in \mathbb{D}} |\rho(d)| < |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil.$$

Also, the bound in Theorem 4.13(a) is tight.

Example 4.9. (a) In Example 4.8 we clearly have

$$\max_{d \in \mathbb{D}} |\rho(d)| = 8 < 3 \cdot \lceil \log_2 5 \rceil = |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil,$$

since $\rho(d)$ only occupies cells in the set $\{0,1,2,3,4,5,6,7\}$. Note, however, that

$$\max_{d \in \mathbb{D}} |\rho(d)| = 8 > 3 \cdot (\lceil \log_2 5 \rceil - 1).$$

(b) In Example 4.7 we have

$$\max_{d \in \mathbb{D}} |\rho(d)| = 3 < 2 \cdot \lceil \log_2 3 \rceil = 4.$$

However, since

$$\max_{d \in \mathbb{D}} |\rho(d)| = 3 > 2 \cdot (\lceil \log_2 3 \rceil - 1) = 2,$$

the bound in Theorem 4.13(a) is best possible. ∎

On the other hand, it is sometimes the case that we have overlapping access sets, $|X| = 2^J$, and also

$$\max_{d \in \mathbb{D}} |\rho(d)| \geq |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil.$$

Example 4.10 illustrates this.

Example 4.10. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b,c,d\}$, and $\mathbb{D} = \{\lambda\} \cup X^2$. There exists (as the reader may verify) a a representation $\rho: \mathbb{D} \to \mathcal{B}^+$ such that the trees shown in Figure 4.6 implement γ_1 and γ_2, respectively. For instance,

$$\begin{aligned} \rho(\lambda) &= 10\_00\_\_ \\ \rho(cd) &= 10\_1111 \\ \rho(ca) &= 10\_10 \\ \rho(bc) &= 0\_1110 \end{aligned}$$

In this case

$$\max_{d \in \mathbb{D}} |\rho(d)| = 6 = |\Gamma| \cdot \lceil \log_{|\mathcal{B}|} |R| \rceil.$$ ∎

Let us now say something about the minimum size a representation can have. In

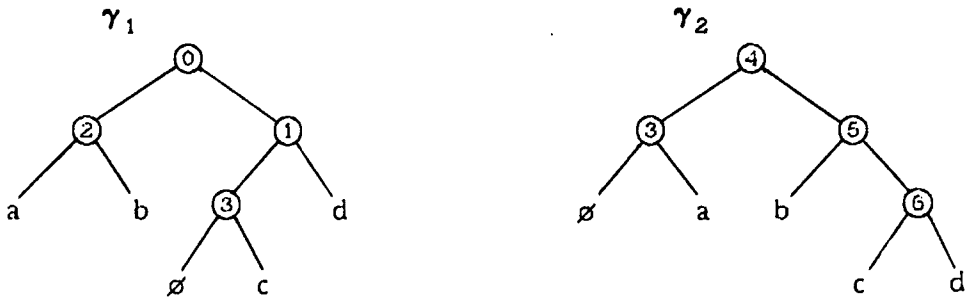


Figure 4.6. Trees for γ_1 and γ_2 of Example 4.10.

particular, it is always the case that $|\rho(d)| \geq |\mathcal{J}| - 1$. Where there is no access set overlap, then it follows from Theorem 4.5 that $|\rho(d)| \geq |\Gamma|$.

Theorem 4.14. Let $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$ and let $\rho: \mathbb{D} \to \mathcal{B}^+$ be some representation. Assume that all $\gamma_i \in \Gamma$ achieve Kraft access.

(a) Then for all $d \in \mathbb{D}$:

$$|\rho(d)| \geq |\mathcal{J}| - 1.$$

(b) If there are no overlapping access sets, then for all $d \in \mathbb{D}$:

$$|\rho(d)| \geq |\Gamma|.$$

Proof: (b) By Theorem 3.9, $\#[\gamma_i(\rho(d))] \geq 1$, and so if there is no access overlap, then Theorem 4.5 tells us that

$$|\rho(d)| \geq \sum_{i=1}^{|\Gamma|} \#[\gamma_i(\rho(d))] \geq \sum_{i=1}^{|\Gamma|} 1 = |\Gamma|.$$

(a) On the other hand, suppose we allow overlapping access sets. If $|\rho(d)| < |\mathcal{J}| - 1$, then there are at most $|\mathcal{J}| - 2$ root node labels. So for $j \in \mathbb{N}^+$ not all of the access trees γ_j where $j \in \mathcal{J}$ can have distinct root node labels. Pick $j_1, j_2 \in \mathcal{J}$, $j_1 \leq j_2$, such that γ_{j_1} and γ_{j_2} have the same root node label. Then by Theorem 4.8, $X^k \notin \mathbb{D}$ for any $j_1 \leq k < j_2$. But we know that $X^{j_1} \subseteq \mathbb{D}$, a contradiction. Therefore, $|\rho(d)| \geq |\mathcal{J}| - 1$. ∎

Example 4.11 shows us that the bound in Theorem 4.14 is best possible.

Example 4.11. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b\}$, and $\mathbb{D} = \bigcup_{i \in J} X^i$, for $J = \{0,3,5,6\}$. Consider the representation $\rho:\mathbb{D} \to \mathcal{B}^+$ that corresponds to the set of access trees shown in Figure 4.7. Then all $\gamma_i \in \Gamma$ achieve Kraft access, and

$$|\rho(\lambda)| = 3 = |J| - 1. \qquad \blacksquare$$

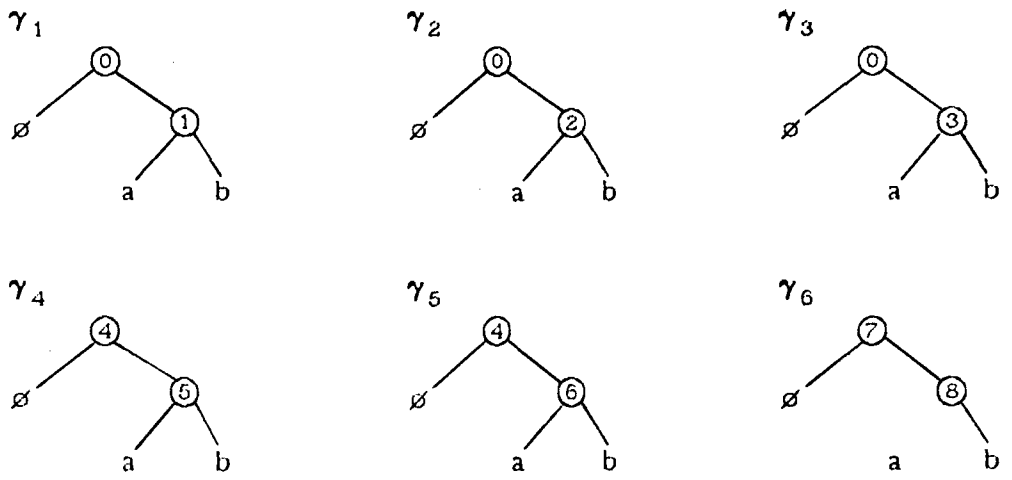


Figure 4.7. Access trees for γ_1, γ_2, γ_3 of Example 4.11.

Note that the bound in Theorem 4.14b may also apply to a table lookup question set that has overlapping access sets; recall Example 4.4.

From Theorem 4.14 it immediately follows that if all $\gamma_i \in \Gamma$ achieve Kraft access and $\max_{d \in \mathbb{D}} |d|$ is unbounded, then infinite storage is required to represent each $d \in \mathbb{D}$.

Corollary 4.14.1. Let $\rho:\mathbb{D} \to \mathcal{B}^+$ be any representation, and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. If $\neg(\exists k_1 \in \mathbb{N})(\max_{d \in \mathbb{D}} |d| \leq k_1)$, then for all $d \in \mathbb{D}$, $\neg(\exists k_2 \in \mathbb{N})(|\rho(d)| \leq k_2)$.

CHAPTER 5

IMPLEMENTING THE TABLE LOOKUP QUESTION SET

In Chapter 4 we discussed the set Γ of table lookup questions and consequences of achieving Kraft access for each $\gamma_i \in \Gamma$. In this chapter we introduce three major classes of representation schemes and then examine the table lookup question set in the contexts of these three basic representations: fixed length, endmarker, and pointer. The fixed length representation was chosen because it sometimes allows us to achieve both Kraft storage and Kraft access. The endmarker and pointer representations were chosen because they illustrate techniques commonly used for implementing variable length lists. In Chapter 6 we reconsider these representations in order to implement stacks.

5.1 Classes of Representations

In this section we briefly discuss some basic definitions and representation techniques and thereby motivate the formal definitions for fixed length, endmarker, and pointer representations, which are presented formally in sections 5.2, 5.3, and 5.4, respectively.

We begin with two notational definitions.

Definition. Consider a function $b \in \mathcal{B}^\dagger$ and recall that

$$b = \{(n, m_b(n)) \mid n \in D(b)\},$$

where $b \subseteq m_b$. For $k \in \mathbb{N}$, we define

$$\{b\}_k \triangleq \{(n + k, m_b(n)) \mid n \in D(b)\}.$$

Thus, $\{b\}_k$ is the set $b \in \mathcal{B}^\dagger$ "displaced" by k, as illustrated in the following example.

Example 5.1. Consider a function $f:S \to \{0,1\}^\dagger$ and let $s_1 \in S$. If

$$f(s_1) = \{(1,0), (3,1), (5,0), (6,1)\},$$

then $$\{f(s_1)\}_0 = f(s_1)$$

and $$\{f(s_1)\}_2 = \{(3,0), (5,1), (7,0), (8,1)\}.$$ ∎

Also, we shall frequently have occasion to refer to the concatenation of two strings in $\mathcal{B}^*$.

> **Definition.** Let f_1 be a function $f_1:S \to \mathcal{B}^*$, let f_2 be a function $f_2:S \to \mathcal{B}^*$, and let $s_1, s_2 \in S$. We write $f_1(s_1) \cdot f_2(s_2)$ to denote the *concatenation* of the strings $f_1(s_1)$ and $f_2(s_2)$, where
>
> $$f_1(s_1) \cdot f_2(s_2) \triangleq f_1(s_1) \cup \{f_2(s_2)\}_{|f_1(s_1)|} \in \mathcal{B}^*.$$

Thus, $$|f_1(s_1) \cdot f_2(s_2)| = |f_1(s_1)| + |f_2(s_2)|,$$

and $$D(f_1(s_1) \cdot f_2(s_2)) = \{0,1, \ldots, |f_1(s_1)| + |f_2(s_2)| - 1\}.$$

Notice that when $f_1(s_1) = \lambda$, then $|f_1(s_1)| = 0$ and $f_1(s_1) \cdot f_2(s_2) = f_2(s_2)$; in particular, $\lambda \cdot \lambda = \lambda$. In an obvious way, the definition can be extended to the concatenation of any countable number of strings.

Example 5.2. Define the function $f:\{a,b,c\} \to \{0,1\}^*$ by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 10 \\ f(c) &= 11 \end{aligned}$$

Then $$\begin{aligned} f(a) \cdot f(b) &= \{(0,0)\} \cup \{(0,1), (1,0)\}_1 \\ &= \{(0,0), (1,1), (2,0)\} = 010 \end{aligned}$$

and $$\begin{aligned} f(c) \cdot f(c) &= \{(0,1), (1,1)\} \cup \{(0,1), (1,1)\}_2 \\ &= \{(0,1), (1,1), (2,1), (3,1)\} = 1111. \end{aligned}$$ ∎

Many commonly used representation schemes involve the concatenation of encodings of a set X. For instance, given a function $f:X \to \mathcal{B}^*$ it would seem natural to encode $x_1x_2 \ldots x_k \in X^k$ as $f(x_1) \cdot f(x_2) \cdot \ldots \cdot f(x_k)$. Similarly, we

could encode X^k by placing each of $f(x_1), \ldots, f(x_k)$ into a fixed field. We illustrate these schemes in Example 5.3a and 5.3b.

Example 5.3. Let $X = \{a,b,c,d\}$, $\mathcal{B} = \{0,1\}$, and consider a function $f:X \to \mathcal{B}^*$ defined by

$$\begin{aligned} f(a) &= 00 \\ f(b) &= 010 \\ f(c) &= 011 \\ f(d) &= 10 \end{aligned}$$

Assume that the domain is of the form $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$, and we want to define a mapping from $\mathbb{D}$ to $\mathcal{B}^+$

(a) Consider the function $f_1:\mathbb{D} \to \mathcal{B}^*$, where

$$f_1(d) = f(d(1)) \cdot f(d(2)) \cdot \ldots \cdot f(d(|d|)).$$

For instance

$$\begin{aligned} f_1(abad) &= f(a) \cdot f(b) \cdot f(a) \cdot f(d) = 000100010 \\ f_1(bdb) &= f(b) \cdot f(d) \cdot f(b) = 01010010 \\ f_1(\lambda) &= \lambda \end{aligned}$$

Notice that, for $|\mathcal{J}| > 1$, f_1 is not a representation because there is no way to recognize the end of the string $f_1(d)$, e.g., $f_1(b)$ and $f_1(ba)$ are indistinguishable since $f_1(b) \subseteq f_1(ba)$.

(b) Consider the function $f_2:\mathbb{D} \to \mathcal{B}^+$, where

$$f_2(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{3(i-1)}$$

Then

$$\begin{aligned} f_2(abad) &= 00\_01000\_10\_ \\ f_2(bdb) &= 01010\_010 \\ f_2(\lambda) &= \lambda. \end{aligned}$$

As in the case of f_1, the function f_2 is a representation if and only if $|\mathcal{J}| = 1$. ∎

In the previous example, f_1 and f_2 would be representations, even for $|\mathcal{J}| > 1$, if there were some way of detecting the ends of codewords $f_1(d)$ and $f_2(d)$. In particular, we might reserve some symbol to mark the end of the list or we might

give some specification of the length $|d|$ or the length $|\rho(d)|$.

Many representations that we consider are what we call concatenation-preserving, where the encoding of a list includes the encodings of the individual elements in the list. We now generalize the familiar notion of concatenation of encodings of list elements to not necessarily imply a "left to right" ordering, only that the encodings are in disjoint sets of memory cells. Thus, if we know where to look then it is possible to determine $d(i)$ and obtain no information about $d(j)$, for $1 \leq i,j \leq |d|$.

Definition. Let $\mathbb{D} = \bigcup_{i \in J} X^i$ and consider a function $f:X \to \mathcal{B}^\dagger$. Define the function $f':\mathbb{D} \to \mathcal{B}^\dagger$ by

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)}$$

where $n_i:\mathbb{D} \to \mathbb{N}$. Then f' is said to be a *concatenation-preserving function* if, for all $i \neq j$,

$$D(\{f(d(i))\}_{n_i(d)}) \cap D(\{f(d(j))\}_{n_j(d)}) = \varnothing$$

Let g be any function $g:\mathbb{D} \to \mathcal{B}^\dagger$ and let f' be the function defined above. Consider the function $\rho:\mathbb{D} \to \mathcal{B}^\dagger$ defined by the union

$$\rho(d) = \{f'(d)\}_{n^1(d)} \cup \{g(d)\}_{n^2(d)},$$

where $n^1:\mathbb{D} \to \mathbb{N}$, $n^2:\mathbb{D} \to \mathbb{N}$. If ρ is, in fact, a representation and if

$$D(\{f'(d(i))\}_{n^1(d)})) \cap D(\{g(d)\}_{n^2(d)}) = \varnothing,$$

then ρ is said to be a *concatenation-preserving representation.*

The condition that the domains of $\{f(d(i))\}_{n_i(d)}$ and $\{f(d(j))\}_{n_j(d)}$ not intersect guarantees that f' is, in fact, a concatenation of encodings of the list elements and that the representations of the list elements do not overlap. Notice that the function g can be chosen in any way whatsoever, so long as the resulting union, ρ, is a representation. We now reconsider Example 5.3 and see that f_1 and f_2 are concatenation-preserving functions.

Example 5.4. The functions f_1 and f_2 from Example 5.3 are concatenation-preserving functions, since they fit the form of the above definition.

(a) Given the function $f:X \to \mathcal{B}^*$ as in Example 5.3, we can define $f_1:\mathbb{D} \to \mathcal{B}^+$ by

$$f_1(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)},$$

where $$n_i(d) = \sum_{j=1}^{i-1} |f(d(j))|.$$

Since $n_{i+1}(d) - n_i(d) = |f(d(i))|$, it is clear that

$$D(\{f(d(i))\}_{n_i(d)}) \cap D(\{f(d(j))\}_{n_j(d)}) = \varnothing.$$

(b) Recall that we defined $f_2:\mathbb{D} \to \mathcal{B}^+$ by

$$f_2(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{3(i-1)}.$$

Since $\max_{x \in X} D(f(x)) = 2$ the domains do not intersect, and it is clear that f_2 is a concatenation-preserving function. ∎

Recalling Example 5.3, when $\mathbb{D} = X^k$ we know that $|d| = k$ and f_1 and f_2 are, in fact, representations. When we wish to allow $\mathbb{D} \neq X^k$, however, then we may wish to consider one of the following three representation schemes.

(i) If $|\rho(d)|$ is of fixed size for all $d \in \mathbb{D}$, then there is no need to specify $|\rho(d)|$. Fixed length representations are discussed in detail in Section 5.2.

(ii) An endmarker representation reserves some symbol or set of symbols $b \in \mathcal{B}^+$ to indicate the end of the list $f(d)$. A formal definition is given in Section 5.3.

(iii) We can encode the length $|d|$ itself and use this as a pointer. A pointer representation is defined formally in Section 5.4.

We illustrate endmarker and pointer representations in Example 5.5a and 5.5b, by extending the function f_1 from examples 5.3 and 5.4.

Example 5.5. (a) Recall from examples 5.3 and 5.4 the function $f:X \to \mathcal{B}^+$ and the function $f_1:\mathbb{D} \to \mathcal{B}^+$. We can then define the representation $\rho_1:\mathbb{D} \to \mathcal{B}^+$ by

$$\rho_1(d) = \{f_1(d)\}_0 \cup \{g(d)\}_{n^2(d)},$$

where $g:\mathbb{D} \to \mathcal{B}^+$ is defined by

$$g(d) = 11^{|d|}$$

and where $$n^2(d) = \sum_{j=1}^{|d|} |f(d(j))|.$$

Since we already know that f_1 is a concatenation-preserving function, we need only note that

$$D(\{g(d)\}_{n^2(d)}) \cap D(\{f_1(d)\}_0) = \varnothing$$

in order to verify that ρ is, in fact, a concatenation-preserving representation. For instance,

$$\begin{aligned}
\rho_1(abad) &= \{f_1(abad)\}_0 \cup \{g(abad)\}_{n^2(d)} \\
&= \{f(a)\cdot f(b)\cdot f(a)\cdot f(d)\}_0 \cup \{11\}_9 = 00010001011 \\
\rho_1(bdb) &= f(b)\cdot f(d)\cdot f(b)\cdot g(bdb) = 0101001011 \\
\rho_1(c) &= 011 \\
\rho_1(\lambda) &= 11.
\end{aligned}$$

Notice that $$|\rho_1(d)| = \sum_{j=1}^{|d|} |f(d(j))| + |g(d)|.$$

Since $g(d)$ and $f(x)$ are distinguishable for all $x \in X$, the string $g(d) = 11$ serves as an endmarker, allowing us to detect when the end of the list has been reached. However, since we also have, e.g., $\rho(c) = 011$, not every occurrence of the string 11 corresponds to the endmarker. It is necessary to somehow decode $\rho(d)$ as we read it.

(b) Recall the functions $f:X \to \mathcal{B}^*$ and $f_2:\mathbb{D} \to \mathcal{B}^+$ from Example 5.3. Define the concatenation-preserving representation $\rho_2:\mathbb{D} \to \mathcal{B}^+$ by

$$\rho_2(d) = \{f_2(d)\}_{|d|+1} \cup \{g(d)\}_0,$$

where $g:\mathbb{D} \to \mathcal{B}^+$ is defined by

$$g(d) = 1^{|d|}0.$$

Notice that $g(d)$ corresponds to the length $|d|$. Thus, after reading $g(d)$, we shall always be able to tell when we are at the end of the list representation $f_2(d)$. For

instance,

$$\rho_2(\text{abad}) = 11110000100010$$
$$\rho_2(\text{bdb}) = 111001010010$$
$$\rho_2(\lambda) = 0$$

We shall later discuss more "efficient" pointer representations. ∎

If in a concatenation-preserving function the functions n_i are all constant functions (i.e., the values of n_i are not functions of the particular d being represented) then we say that the function has fixed position fields. Intuitively, this says that if we were to ask the question γ_i, for $i \leq |d|$, then we would always know where in the representation to begin reading.

Definition. Let $\mathbb{D} = \bigcup_{i \in J} X^i$ and let f be a function $f: X \to \mathcal{B}^+$. Consider a concatenation-preserving function $f': \mathbb{D} \to \mathcal{B}^+$ defined by

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)}$$

where $n_i: \mathbb{D} \to \mathbb{N}$. If for all $d_1, d_2 \in \mathbb{D}$ and for all j, $1 \leq j \leq \max_{i \in J} i$,

$$n_j(d_1) = n_j(d_2)$$

then the function f' is said to have *fixed position fields*. We define an n_i *field* to be the set

$$\bigcup_{x \in X} D(f(x)) + n_i,$$

for $1 \leq i \leq |d|$, where we use the notation

$$\{s_1, s_2, \ldots, s_p\} + k \triangleq \{s_1+k, s_2+k, \ldots, s_p+k\}.$$

Clearly neither of the extensions of f_1 in Example 5.5 gives us a function with fixed position fields. The function f_2 from Example 5.3 is, however, a fixed position field function, since $n_i(d) = 3\cdot(i-1)$ for all $d \in \mathbb{D}$ and thus each n_i is a constant function. Since each n_i field consists of all cells which may be occupied by $\rho(d(i))$, the n_i field for ρ of Example 5.3 is just $\{n_i, n_i + 1, n_i + 2\}$. Notice that it is not necessary that an n_i field consist of contiguous memory cells, although for simplicity most of our examples will be of this form. In fact, it is possible for two

n_i fields to "cross"; e.g., we might have

$$k_1, k_1 + k_2 \in \bigcup_{x \in X} D(f(x)) + n_i$$

and

$$k_1 + k_3 \in \bigcup_{x \in X} D(f(x)) + n_{i+1},$$

for $1 \leq k_3 < k_2$. Example 5.6 gives an example of a concatenation-preserving representation with fixed position fields, where a field does not consist of contiguous cells.

Example 5.6. Let $X = \{a,b,c\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$. Define the function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$ by

$$\begin{aligned} f(a) &= 0\_0 \\ f(b) &= 0\_1 \\ f(c) &= 1\_0 \\ f(\varnothing) &= 1\_1 \end{aligned}$$

Consider the representation $\rho: \mathbb{D} \to \mathcal{B}^+$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{1\_1\}_{n_{|d|+1}},$$

where

$$n_i = \begin{cases} 2 \cdot i - 3 & \text{for } i \text{ even} \\ 2 \cdot i - 2 & \text{for } i \text{ odd} \end{cases}$$

Thus, $d(1)$ occupies cells 0 and 2, $d(2)$ occupies cells 1 and 3, $d(3)$ occupies cells 4 and 6, $d(4)$ occupies cells 5 and 7, $d(5)$ occupies cells 8 and 10, etc. For instance,

$$\begin{aligned} \rho(\lambda) &= 1\_1 \\ \rho(abaa) &= 000100001\_1 \\ \rho(bacba) &= 001010010101. \end{aligned}$$

So an n_i field is not a set of contiguous cells. In fact, the n_3 field is

$$\bigcup_{x \in X} D(f(x)) + n_3 = \bigcup_{x \in X} D(f(x)) + 4 = \{4, 6\}$$

and the n_4 field is

$$\bigcup_{x \in X} D(f(x)) + n_4 = \bigcup_{x \in X} D(f(x)) + 5 = \{5, 7\}.$$

Notice that the n_3 field and the n_4 field "cross", since

$$4, 6 \in \bigcup_{x \in X} D(f(x)) + n_3$$

and

$$5 \in \bigcup_{x \in X} D(f(x)) + n_4.$$

∎

By definition it is, of course, not possible for two η_3 fields to actually overlap.

In sections 5.3 and 5.4 we extend the notion of a fixed position field function to fixed position field endmarker and pointer representations. For instance, we shall refer to the representation γ in Example 5.6 as a fixed position field endmarker representation.

representation with fixed position fields, where a field does not consist of contiguous cells.

Example 5.6. Let $X = \{a,b,c\}$, $B = \{0,1\}$, and $ID = \bigcup_{i=0}^{\infty} X^i$. Define the function $f: X \cup \{\rho\} \to B^+$ by

$$\begin{aligned} f(a) &= 0\_0 \\ f(b) &= 0\_1 \\ f(c) &= 1\_0 \\ f(\rho) &= 1\_1 \end{aligned}$$

Consider the representation $\gamma: ID \to B^+$ defined by

$$\gamma(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}^{\eta_1} \cup \{1\_1\}^{\eta_{|d|+1}},$$

where

$$\eta_i = \begin{cases} 2\cdot i - 2 & \text{for } i \text{ odd} \\ 2\cdot i - 3 & \text{for } i \text{ even} \end{cases}$$

Thus, $d(1)$ occupies cells 0 and 2, $d(2)$ occupies cells 1 and 3, $d(3)$ occupies cells 4 and 6, $d(4)$ occupies cells 5 and 7, $d(5)$ occupies cells 8 and 10, etc. For instance,

$$\begin{aligned} \gamma(\lambda) &= 1\_1 \\ \gamma(abaa) &= 0001000011\_1 \\ \gamma(bacba) &= 0010010101010\_1. \end{aligned}$$

So an η_1 field is not a set of contiguous cells. In fact, the η_3 field is

$$\bigcup_{x \in X} D(f(x)) + \eta_3 = \bigcup_{x \in X} D(f(x)) + 4 = \{4, 6\}$$

and the η_4 field is

$$\bigcup_{x \in X} D(f(x)) + \eta_4 = \bigcup_{x \in X} D(f(x)) + 5 = \{5, 7\}.$$

Notice that the η_3 field and the η_4 field "cross", since

$$4, 6 \in \bigcup_{x \in X} D(f(x)) + \eta_3$$

and

$$5 \in \bigcup_{x \in X} D(f(x)) + \eta_4.$$

5.2 Fixed Size Representations

In theorems 4.9 and 4.10 we showed that if a representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ achieves Kraft storage and also achieves Kraft access for all $\gamma \in \Gamma$, then $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X^n$. In this section we show that it is possible, where $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X$, to have Kraft storage and access with a fixed size representation. In fact, if the relative sizes of the problem and machine alphabets are chosen correctly, and if the domain is of one of the two appropriate forms, then there is always a fixed size representation which achieves Kraft storage and access (see Theorem 5.5 and Corollary 5.5.1).

Recalling Section 5.1, a representation ρ is said to be of fixed size if it maps all strings in $\mathbb{D}$ into strings of the same length.

Definition. A representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ is a *fixed size representation* function of *size* r if and only if

$$(\forall d \in \mathbb{D})(|\rho(d)| = r)$$

Notice that the definition makes no requirement that $D(\rho(d)) = \{0, 1, \ldots, |d|-1\}$, and in general $\rho(d)$ might occupy any r cells of memory, not necessarily contiguous. Of course, we frequently consider a representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^r$, where each $d \in \mathbb{D}$ is mapped onto a sequence $m = m(1)m(2)\ldots m(r) = \rho(d)$, for $m(i) \in \mathcal{B}$. For any fixed size representation, however, it is known that each $\rho(d)$ occupies exactly r cells, and so it is not necessary to store any additional information concerning the length of the representation. Let us look at two examples of fixed size representations.

Example 5.7. Let $\mathbb{D} = \{\lambda\} \cup X \cup X^2$, $X = \{a,b\}$, and $\mathcal{B} = \{0,1\}$. Define the fixed size representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^3$ as follows:

$$\begin{aligned} \rho(\lambda) &= 000 \\ \rho(a) &= 001 \\ \rho(b) &= 010 \end{aligned}$$

$$\begin{aligned}\rho(aa) &= 011\\ \rho(ab) &= 100\\ \rho(ba) &= 101\\ \rho(bb) &= 110\end{aligned}$$

Since there is no $d \in \mathbb{D}$ such that $\rho(d) = 111$, ρ does not achieve Kraft storage. Also, it is not possible, using representation ρ, to implement any $\gamma_i \in \Gamma$ so as to achieve Kraft access. (If γ_i did achieve Kraft access, then the tree for γ_i would have three leaves and therefore two internal nodes. So one answer among a, b, $\varnothing$ would be determined in a single access, but by inspection we can see that this cannot happen.) ∎

Example 5.8 illustrates a procedure for constructing a fixed size representation for which, if $\mathbb{D} \neq X^n$ and $|X| = |\mathcal{B}|^k - 1$, we can attain Kraft access (although not Kraft storage). Notice that $r = k \cdot |\Gamma|$, and we answer γ_i by first accessing cell $(i-1) \cdot k$.

Example 5.8. Let $\mathbb{D} = \bigcup_{i=0}^{n} X^i$, $X = \{a,b,c\}$, and $\mathcal{B} = \{0,1\}$. Define the fixed size concatenation-preserving representation $\rho:\mathbb{D} \to \mathcal{B}^{2n}$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{2(i-1)} \cup \bigcup_{i=|d|+1}^{n} \{f(\varnothing)\}_{2(i-1)}$$

where $f:X \cup \{\varnothing\} \to \mathcal{B}^2$ is defined by

$$\begin{aligned}f(a) &= 00\\ f(b) &= 01\\ f(c) &= 10\\ f(\varnothing) &= 11\end{aligned}$$

In particular, for $n = 2$ we have

$$\begin{aligned}\rho(\lambda) &= 1111\\ \rho(a) &= 0011\\ \rho(b) &= 0111\\ \rho(c) &= 1011\\ \rho(aa) &= 0000\\ \rho(ab) &= 0001\\ \rho(ac) &= 0010\\ \rho(ba) &= 0100\\ \rho(bb) &= 0101\end{aligned}$$

$$\rho(bc) = 0110$$
$$\rho(ca) = 1000$$
$$\rho(cb) = 1001$$
$$\rho(cc) = 1010$$

Notice that $|\mathcal{B}| = 2$ and $2^2 - 1 = 3 = |X|$. So $r = 2 \cdot 2$ and to answer γ_i we first access cell $2 \cdot (i-1)$. Figure 5.1 illustrates access trees for γ_1 and γ_2, and it is clear that we achieve Kraft access. On the other hand, ρ does not have Kraft storage because

$$\sum 13 \cdot 2^{-4} = \frac{13}{16} \neq 1.$$

Intuitively, we would have achieved Kraft storage if we had altered the definition of ρ by letting $\rho(\lambda) = 11\_\_$; this would have made $\rho(\mathbb{D})$ a complete code. Instead, we chose to specify values for $m(2)$ and $m(3)$ so we could always answer γ_2 in two accesses. This illustrates a trade-off between Kraft storage and Kraft access. ∎

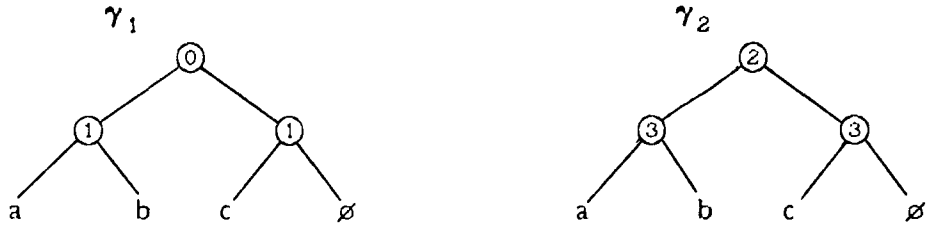


Figure 5.1. Access trees for γ_1 and γ_2 of Example 5.8.

Notice that when we define some fixed size representation ρ, we have not explicitly said anything about the elements in the problem domain $\mathbb{D}$. If, however, we meet Kraft storage, then we know by the following theorem that there are $|\mathcal{B}|^r$ elements in the domain.

Theorem 5.1. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be a fixed size representation of size r. ρ achieves Kraft storage if and only if $|\mathbb{D}| = |\mathcal{B}|^r$.

Proof: Since $|\rho(d)| = r$ for all $d \in \mathbb{D}$, then

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = |\mathbb{D}| \cdot |\mathcal{B}|^{-r}$$

and we have Kraft storage if and only if $|\mathbb{D}|\cdot|\mathcal{B}|^{-r} = 1$; that is, if and only if $|\mathbb{D}| = |\mathcal{B}|^r$. ∎

Notice that we could, of course, be representing any $|\mathcal{B}|^r$ strings in $\mathbb{D}$.

We know by Theorem 3.10 that we cannot achieve Kraft access for $|X| < |\mathcal{B}|$. Unfortunately, even for $|X| \geq |\mathcal{B}|$, the conditions $|\mathbb{D}| = |\mathcal{B}|^r$ and $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$ do not guarantee that there is a fixed size representation that attains Kraft access.

Example 5.9. Let $|\mathcal{B}| = 3$, $|X| = 4$, and $\mathbb{D} = \{\lambda\} \cup X^2 \cup X^3$. Then for $r = 4$, we have $|\mathcal{B}|^r = 3^4 = 4^3 + 4^2 + 1 = |\mathbb{D}|$, and a fixed size representation $\rho:\mathbb{D} \to \mathcal{B}^4$ is storage optimal. On the other hand, by theorems 4.9 and 4.10 we know that there is no representation, fixed size or otherwise, that achieves both Kraft storage and Kraft access for the table lookup question set $\Gamma = \{\gamma_1, \gamma_2, \gamma_3\}$. ∎

In the last chapter, we have already shown that in order to possibly achieve Kraft storage and Kraft access, it must be the case that $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X^n$. If we wish a fixed size representation to have Kraft storage and access, then either $\mathbb{D} = X^n$ or else we have the less interesting situation where $\mathbb{D} = \{\lambda\} \cup X^1$.

> **Lemma 5.1.** Let $\mathbb{D} = \{\lambda\} \cup X^n$ and consider a fixed size representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$, of size r, which achieves Kraft storage. Assume also that each $\gamma_i \in \Gamma$ achieves Kraft access. Then $|\mathcal{B}| = 2$ and $|\Gamma| = 1$; i.e., $\mathbb{D} = \{\lambda\} \cup X^1$ and $|X| = 1$.

Proof: By Theorem 4.9 and Corollary 4.9.1, since $\mathbb{D} \neq X^n$ then the only way we can achieve both Kraft storage and access is to have $|\mathcal{B}| = 2$ and for there to be some $\gamma_i, \gamma_j \in \Gamma$ such that γ_i and γ_j have overlapping sets. As a consequence of Theorem 4.8, we know that γ_i, γ_j access some cell in common if and only if $X^k \notin \mathbb{D}$ for all $i \leq k < j$. Since $\mathbb{D} = \{\lambda\} \cup X^n$, then any pair of table lookup questions has overlapping access sets. Now lemmas 4.2 and 4.4 allow us to conclude

that each $\gamma_i \in \Gamma$ has the same root node label and each has a leaf labelled $\varnothing$ at depth 1. But this says that $|\rho(\lambda)| = 1$, and so if ρ is a fixed size representation then it is a fixed size representation of length 1. Thus $|\Gamma| = 1$. If γ_1 has a leaf labelled a at depth greater than 1, then $|\rho(\lambda)| \neq |\rho(a)|$ and ρ could not be a fixed size representation. Thus γ_1 has its only two leaves at depth one and so we have the trivial case $\mathbb{D} = \{\lambda\} \cup X^1$ and $|X| = 1$. ∎

Of course, the above lemma simply says that if a fixed size representation achieves Kraft storage and access, then $\mathbb{D} = \{\lambda\} \cup X$. The following example shows that it is, in fact, possible to have $\mathbb{D} = \{\lambda\} \cup X$ for a fixed size representation which does have Kraft storage and access.

Example 5.10. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b,c\}$, and $\mathbb{D} = \{\lambda\} \cup X$. Define the representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$ by

$$\rho(\lambda) = 0\_1$$
$$\rho(a) = 0\_0$$
$$\rho(b) = 10\_$$
$$\rho(c) = 11\_$$

Clearly ρ is a storage optimal fixed size representation of size 2, and from Figure 5.2 we see that it is possible to implement γ_1 so that it has Kraft access. ∎

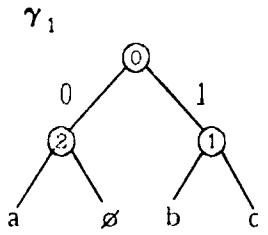


Figure 5.2. Access tree for γ_1 of Example 5.10.

Now from Lemma 5.1 and theorems 4.9 and 4.10 we obtain the following result.

Theorem 5.2. Consider a fixed size representation $\rho:\mathbb{D} \to \mathcal{B}^+$. Assume ρ achieves Kraft storage and each $\gamma_i \in \Gamma$ achieves Kraft access. Then $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X$.

In fact, achieving Kraft storage and access with a fixed size representation tells us something about the relative sizes of the problem and machine alphabets.

Lemma 5.2. Consider a fixed size representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage, and assume that all $\gamma_i \in \Gamma$ achieve Kraft access. Then, for all $\gamma_i \in \Gamma$, the access tree for γ_i has uniform depth.

Proof: By Theorem 5.2, there are two cases to consider:
(i) $\mathbb{D} = \{\lambda\} \cup X$. In this case we know by Lemma 5.1 that $|\Gamma| = 1$ and $|X| = 1$, so γ_1 clearly has uniform depth and ρ is a fixed size representation.
(ii) $\mathbb{D} = X^k$. Then by Theorem 4.10 there are no overlapping access sets. Assume there is some γ_n whose access tree does not have uniform depth; in particular, let leaves labelled $x_1, x_2 \in X$ be at different depths. Then there exist $d_1, d_2 \in \mathbb{D}$ such that $d_1(n) = x_1$ and

$$d_2(i) = \begin{cases} d_1(i) & \text{for } i \neq n \\ x_2 & \text{for } i = n \end{cases}$$

By Theorem 4.12,

$$|\rho(d_1)| = \sum_{i \neq n} \#[\gamma_i(\rho(d_1))] + \#[\gamma_n(\rho(d_1))]$$

and

$$|\rho(d_2)| = \sum_{i \neq n} \#[\gamma_i(\rho(d_2))] + \#[\gamma_n(\rho(d_2))].$$

But

$$\#[\gamma_n(\rho(d_1))] = \#[x_1] \neq \#[x_2] = \#[\gamma_n(\rho(d_2))].$$

Thus $|\rho(d_1)| \neq |\rho(d_2)|$, implying ρ is not a fixed size representation, a contradiction. So each γ_i has a tree of uniform depth. ∎

This lemma allows us to prove that $|X| = |\mathcal{B}|^k$ or $|X| = |\mathcal{B}|^k - 1$ if we are to attain Kraft storage and access with a fixed size representation.

Theorem 5.3. Consider a fixed size representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage, and assume all $\gamma_i \in \Gamma$ achieve Kraft access. If $\mathbb{D} = X^n$ then $|\mathcal{B}|^k = |X|$ for some $k \in \mathbb{N}$, and if $\mathbb{D} = \{\lambda\} \cup X$ then $|\mathcal{B}|^k = |X| + 1$.

Proof: Let $\mathbb{D} = X^n$. By Lemma 5.2, we know that the access tree for γ_i has uniform depth, say k, and so $|\mathcal{B}|^k = |R(\gamma_i(\mathbb{D}))| = |X|$. Similarly, for $\mathbb{D} = \{\lambda\} \cup X$, $|R(\gamma_i(\mathbb{D}))| = |X| + 1 = |\mathcal{B}|^k$. ∎

The following example illustrates, however, that attaining Kraft storage and access, even where $\mathbb{D} = X^n$ and $|X| = |\mathcal{B}|^k$, does not necessarily mean our representation has fixed size.

Example 5.11. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b,c,d\}$, and $\mathbb{D} = X^2$. Define the representation $\rho:\mathbb{D} \to \mathcal{B}^+$ as illustrated in Figure 5.3. More specifically, for $x_1, x_2 \in X$, we can let $\rho(x_1 \cdot x_2) = f_1(x_1) \cdot f_2(x_2)$, where the representation tree for f_i has the same form as the access tree γ_i. For instance,

$$\begin{aligned} \rho(ac) &= 0\_\_10 \\ \rho(ad) &= 0\_\_11 \\ \rho(bb) &= 10\_01 \\ \rho(dc) &= 11110. \end{aligned}$$

From the trees it is clear that γ_1 and γ_2 achieve Kraft access. Also, since

$$|\rho(x_1 \cdot x_2)| = |f_1(x_1)| + |f_2(x_2)| = |f_1(x_1)| + 2$$

then the reader can verify that

$$\sum_{d \in X^2} 2^{-|\rho(d)|} = 4 \cdot 2^{-3} + 4 \cdot 2^{-4} + 8 \cdot 2^{-5} = 1$$

and so ρ achieves Kraft storage. ∎

On the other hand, the following example shows that we could have defined ρ in the above example to be a fixed size representation and still have attained Kraft

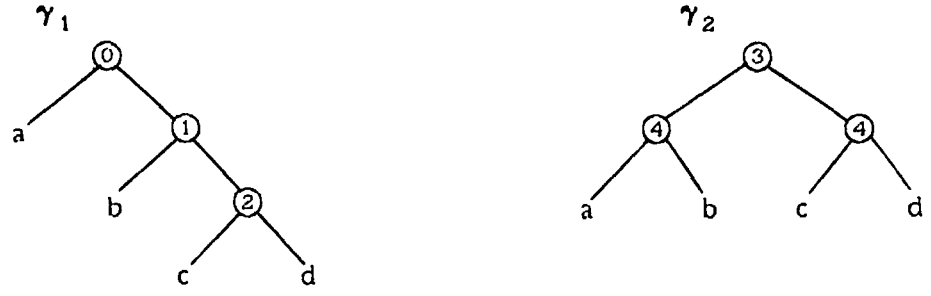


Figure 5.3. Access trees for γ_1 and γ_2 of Example 5.11.

storage and access. In fact, there would always be such a fixed size representation.

Example 5.12. Let $\mathcal{B}$, X, and $\mathbb{D}$ be the same as in Example 5.11 and define the representation $\rho:\mathbb{D} \to \mathcal{B}^+$ as illustrated in Figure 5.10. For instance,

$$\rho(ac) = 0010$$
$$\rho(ad) = 0011$$
$$\rho(bb) = 0101$$
$$\rho(dc) = 1110$$

Figure 5.4. Access trees for γ_1 and γ_2 of Example 5.12.

Clearly we achieve both Kraft access and Kraft storage. ∎

To help motivate some further discussions, we first prove the following simple lemma.

Lemma 5.3. Let $\mathbb{D} = X^n$. Then the following statements are equivalent.

(1) There is some representation $\rho: X \to \mathcal{B}^+$ which attains Kraft storage.

(2) There is some implementation for which each $\gamma_i \in \Gamma$ achieves Kraft access.

(3) There is some $k \in \mathbb{N}$ such that $|X| = k\cdot(|\mathcal{B}| - 1) + 1$.

Proof: For $\mathbb{D} = X^n$, $R(\gamma_1(\mathbb{D})) = X$. There is some representation ρ which attains Kraft storage for $x \in X$ if and only if there is a $|\mathcal{B}|$-ary tree with $|X|$ leaves if and only if $|X| = k\cdot(|\mathcal{B}| - 1) + 1$. Also, γ_1 achieves Kraft access if and only if its $|\mathcal{B}|$-ary tree has $|X|$ leaves if and only if $|X| = k\cdot(|\mathcal{B}| - 1) + 1$. ∎

It is not the case, however, that Kraft storage for a representation $\rho: X^n \to \mathcal{B}^+$ implies Kraft storage for some representation $\rho: X \to \mathcal{B}^+$.

Example 5.13. Let $|\mathcal{B}| = 5$ and $|X| = 7$. To get Kraft storage for X, we would need $i\cdot(|\mathcal{B}| - 1) + 1 = 4i + 1 = 7$, which is not possible. But for X^2, $i = 12$ gives us $i\cdot(|\mathcal{B}| - 1) + 1 = 4i + 1 = 49$. ∎

We are now ready to prove the main results of this section. The proof of the following lemma is essentially the same as the proof of Lemma 4.3.

Lemma 5.4. Let $X = \{x_1, x_2, \ldots, x_k\}$ and $\mathbb{D} = X^n$. Consider a representation $f: X \to \mathcal{B}^+$. If f achieves Kraft storage, then a concatenation-preserving representation $\rho: \mathbb{D} \to \mathcal{B}^+$ defined by

$$\rho(d) = \bigcup_{i=1}^{n} \{f(d(i))\}_{n_i(d)},$$

where $n_i: \mathbb{D} \to \mathbb{N}$, also achieves Kraft storage.

Proof: By induction on n we prove that

$$\sum_{d \in X^n} |\mathcal{B}|^{-|\rho(d)|} = 1. \tag{5.1}$$

Basis: For $n = 1$, $|\rho(d)| = |f(d)|$ and so

$$\sum_{d\in X} |\mathcal{B}|^{-|\rho(d)|} = \sum_{d\in X} |\mathcal{B}|^{-|f(d)|} = \sum_{x\in X} |\mathcal{B}|^{-|f(x)|} = 1.$$

Induction step: Assume that (5.1) holds for n. Then

$$\begin{aligned}
\sum_{d\in X^{n+1}} |\mathcal{B}|^{-|\rho(d)|} &= \sum_{d\in X^{n+1}} |\mathcal{B}|^{-\sum_{i=1}^{n+1}|f(d(i))|} \\
&= \sum_{d\in X^{n}\cdot x_1} |\mathcal{B}|^{-(\sum_{i=1}^{n}|f(d(i))| + |f(x_1)|)} \\
&\quad + \sum_{d\in X^{n}\cdot x_2} |\mathcal{B}|^{-(\sum_{i=1}^{n}|f(d(i))| + |f(x_2)|)} \\
&\quad + \ldots + \sum_{d\in X^{n}\cdot x_k} |\mathcal{B}|^{-(\sum_{i=1}^{n}|f(d(i))| + |f(x_k)|)} \\
&= |\mathcal{B}|^{-|f(x_1)|} \sum_{d\in X^n} |\mathcal{B}|^{-\sum_{i=1}^{n}|f(d(i))|} \\
&\quad + |\mathcal{B}|^{-|f(x_2)|} \sum_{d\in X^n} |\mathcal{B}|^{-\sum_{i=1}^{n}|f(d(i))|} \\
&\quad + \ldots + |\mathcal{B}|^{-|f(x_k)|} \sum_{d\in X^n} |\mathcal{B}|^{-\sum_{i=1}^{n}|f(d(i))|}
\end{aligned}$$

By our inductive hypothesis this then gives us

$$\sum_{d\in X^{n+1}} |\mathcal{B}|^{-|\rho(d)|} = \sum_{i=1}^{k} |\mathcal{B}|^{-|f(x_i)|} = \sum_{x\in X} |\mathcal{B}|^{-|f(x)|} = 1. \quad \blacksquare$$

Theorem 5.4. Let $\mathbb{D} = X^n$. If there exists some $k \in \mathbb{N}$ for which $|X| = k\cdot(|\mathcal{B}| - 1) + 1$, then there is an implementation (α, ρ) solving $(\Gamma, \mathbb{D})$ such that $\rho:\mathbb{D} \to \mathcal{B}^+$ achieves Kraft storage and each $\gamma_i \in \Gamma$ achieves Kraft access.

Proof: Since $|X| = k\cdot(|\mathcal{B}| - 1) + 1$, we know by Lemma 3.1 that there is a $|\mathcal{B}|$-ary tree T with $|X|$ leaves and node labels chosen from the set $\{0,1, \ldots r\}$, for $r \leq k-1$. We can use this tree T to define the storage optimal representation $f:X \to \mathcal{B}^*$. Now define the concatenation-preserving representation $\rho:\mathbb{D} \to \mathcal{B}^+$ by

$$\rho(d) = \bigcup_{i=1}^{n} \{f(d(i))\}_{(r+1)(i-1)}$$

By Lemma 5.4, since f achieves Kraft storage so does ρ. Also, if we implement $\gamma_i \in \Gamma$ by the same tree T except replacing node label j by label $j + (r+1)\cdot(i-1)$, then each $\gamma_i \in \Gamma$ achieves Kraft access. ∎

From Lemma 5.3, Theorem 5.4 holds if instead of the condition $|X| = i\cdot(|\mathcal{B}| - 1) + 1$ we have the condition that there be some representation $\rho: X \to \mathcal{B}^+$ which attains Kraft storage or that there be some implementation for γ_i which achieves Kraft access. Trivially, the above theorem also holds for $\mathbb{D} = \{\lambda\} \cup X$, when $|X| + 1 = i\cdot(|\mathcal{B}| - 1) + 1$.

> **Corollary 5.4.1.** Let $\mathbb{D} = \{\lambda\} \cup X$. If there exists some $k \in \mathbb{N}^+$ for which $|X| + 1 = k\cdot(|\mathcal{B}| - 1) + 1$, then there is an implementation (α, ρ) solving $(\Gamma, \mathbb{D})$ such that $\rho: \mathbb{D} \to \mathcal{B}^+$ achieves Kraft storage and $\gamma_1 \in \Gamma$ achieves Kraft access.

We present an example to illustrate how ρ and f in the proof of Theorem 5.4 might be chosen.

Example 5.14. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b,c,d,e\}$, and $\mathbb{D} = X^n$. Then $|X| = i\cdot(|\mathcal{B}| - 1) + 1$ is satisfied by $i = 4$, and there is a binary tree with five leaves and four internal nodes whose labels are in $\{0,1,2,3\}$. In fact, there are many such trees, and we (arbitrarily) pick T to be the tree shown in Figure 5.5a. Using T, we define the representation $f: X \to \mathcal{B}^+$ by

$$\begin{aligned} f(a) &= 00\_\_ \\ f(b) &= 01\_\_ \\ f(c) &= 1\_00 \\ f(d) &= 1\_01 \\ f(e) &= 1\_1\_ \end{aligned}$$

By inspection, f attains Kraft storage. We define the concatenation-preserving representation $\rho: \mathbb{D} \to \mathcal{B}^+$ by

$$\rho(d) = f(d(1))\cdot f(d(2)).$$

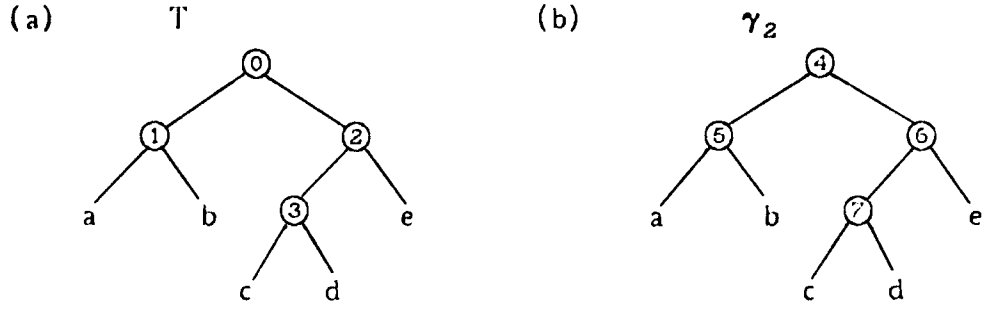


Figure 5.5. Trees for T and γ_2 of Example 5.14.

So T is also the access tree for γ_1, and the access tree for γ_2 is the same as T but has each node label j replaced by the label 2i + j. The tree for γ_2 is illustrated in Figure 5.5b. Then we have, for instance, $\rho(bc) = 01\_\_1\_00$. The representation ρ achieves Kraft storage because

$$\sum_{d \in X^2} 2^{-|\rho(d)|} = 9 \cdot 2^{-4} + 12 \cdot 2^{-5} + 4 \cdot 2^{-6} = 1.$$

By inspection of the trees for γ_1 and γ_2, we also attain Kraft access. ∎

Notice, however, that the representation ρ in Theorem 5.4 has many "gaps" in it. Even if we had constructed the tree T so that each node at depth j had label j, we would still have had gaps, unless T were of uniform depth. If we require that ρ be located in consecutive cells, then we cannot obtain Kraft access unless for all $d_1, d_2 \in X^n$, $|\rho(d_1)| = |\rho(d_2)|$; i.e., $\#[\gamma_i(\rho(d))] = \#[\gamma_j(\rho(d))]$, for all $\gamma_i, \gamma_j \in \Gamma$. We now show that if in Theorem 5.4 it had also been the case that $|X| = |\mathcal{B}|^k$, then there would have been an implementation achieving Kraft storage and access with a fixed size representation and without any "gaps".

Theorem 5.5. Let $\mathbb{D} = X^n$ and $|X| = |\mathcal{B}|^k$ for $n,k \in \mathbb{N}^+$. Then there is an implementation $(\mathcal{A}, \rho)$ solving $(\Gamma, \mathbb{D})$ such that $\rho: \mathbb{D} \to \mathcal{B}^{nk}$ is a fixed size representation achieving Kraft storage, and each $\gamma_i \in \Gamma$ achieves Kraft access.

Proof: Since we are given that $|X| = |\mathcal{B}|^k$, the equation $|X| = i\cdot(|\mathcal{B}| - 1) + 1$ is satisfied for $i = \sum_{j=0}^{k-1} |\mathcal{B}|^j$. Theorem 5.4 immediately tells us that there is some representation with Kraft storage and access, but we want to show that there is, in fact, such a fixed size representation. As in the proof of Theorem 5.4, we define the concatenation-preserving representation $\rho:\mathbb{D} \to \mathcal{B}^{nk}$ by

$$\rho(d) = f(d(1))\cdot f(d(2))\cdot \ldots \cdot f(d(n))$$

where $f:X \to \mathcal{B}^k$ corresponds to a tree T of uniform depth k where each internal node at depth j has label j. Certainly f and therefore ρ both achieve Kraft storage, as verified by

$$\sum_{d\in\mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = \sum_{d\in X^n} |\mathcal{B}|^{-nk} = |X|^n\cdot |\mathcal{B}|^{-nk} = |X|^n\cdot |X|^{-n} = 1.$$

Also, we implement $\gamma_m \in \Gamma$ by the same tree T, with labels j replaced by mk + j. Each $\gamma_m \in \Gamma$ achieves Kraft access, since $\#[\gamma_m(\rho(d))] = k$ and

$$\sum_{r\in X} |\mathcal{B}|^{-\alpha(r)} = \sum_{r\in X} |\mathcal{B}|^{-k} = |X|\cdot|\mathcal{B}|^{-k} = 1. \quad \blacksquare$$

We can give an example, similar to Example 5.12, which illustrates this theorem.

Example 5.15. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b,c,d\}$, and $\mathbb{D} = X^2$. Notice that $|X| = |\mathcal{B}|^2$, and Figure 5.6a shows a tree T of uniform depth two corresponding to the representation $f:X \to \mathcal{B}^2$. Then we define the representation $\rho:\mathbb{D} \to \mathcal{B}^4$ by $\rho(d) = f(d(1))\cdot f(d(2))$. For instance,

$$\begin{aligned} \rho(ac) &= 0010 \\ \rho(ad) &= 0011 \\ \rho(bb) &= 0101 \\ \rho(dc) &= 1110 \end{aligned}$$

The tree T of Figure 5.6a is the access tree for γ_1, and the access tree for γ_2 is shown in Figure 5.6b. $\blacksquare$

Analogous to Theorem 5.4, we have the following corollary.

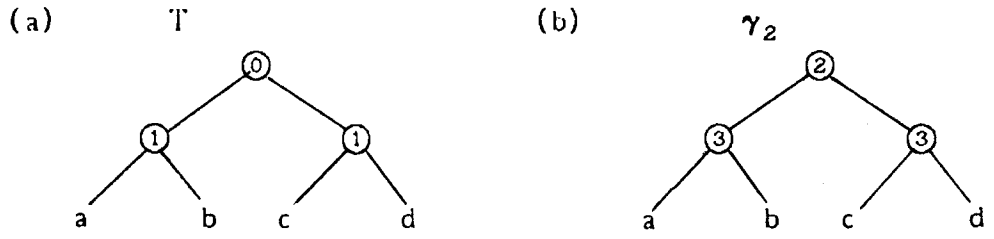


Figure 5.6. Trees for T and γ_2 of Example 5.15.

Corollary 5.5.1. Let $\mathbb{D} = \{\lambda\} \cup X$ and $|X| + 1 = |\mathcal{B}|^k$ for some $k \in \mathbb{N}^+$. Then there is an implementation (α, ρ) solving $(\Gamma, \mathbb{D})$ such that $\rho:\mathbb{D} \to \mathcal{B}^k$ is a fixed size representation achieving Kraft storage, and each $\gamma_1 \in \Gamma$ achieves Kraft access.

What we have proved in this section is a weak equivalence between the requirements that $\mathbb{D} = X^n$ (or $\mathbb{D} = \{\lambda\} \cup X$) and that there be some fixed size implementation in which we achieve Kraft storage and access. More precisely, Theorem 5.2 told us that if there is a fixed size representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage and for which each $\gamma_1 \in \Gamma$ achieves Kraft access, then $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X$. Conversely, Theorem 5.5 and Corollary 5.5.1 essentially tell us that if $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X$, then there is some fixed size representation which achieves Kraft storage and access. The condition $|X| = |\mathcal{B}|^k$ (or $|X| + 1 = |\mathcal{B}|^k$) was put in to avoid "rounding errors". If we do not have $|X| = |\mathcal{B}|^k$ for $\mathbb{D} = X^n$, then either we do not have Kraft storage or else our tree must have leaves at (at least) two depths, j and j + 1. This would cause $nj \leq |\rho(d)| \leq n(j + 1)$ and so ρ would not be of exactly fixed size. Or else we could let $\rho:\mathbb{D} \to \mathcal{B}^{n(j+1)}$ be fixed size and then we would not quite attain Kraft storage. Thus, theorems 5.2 and 5.5 allow us to prove the following result.

Theorem 5.6. Let $|X| = |\mathcal{B}|^k$. Then $D = \bigcup_{i=0}^{\infty} X^i$ has a fixed size representation $\rho: D \to \mathcal{B}^+$ that achieves Kraft storage and for which each $\gamma_1 \in \Gamma_\rho$ achieves Kraft access.

Recall from Section 5.1 that an endmarker representation has some fixed symbol or sequence of symbols in $\mathcal{B}^+$ which are always at the "end" of the list. Example 5.5a is an example of an endmarker representation. The representation ρ in Example 2.7 is also an endmarker representation, with endmarker 0. We now give a formal definition.

Definition. Let f be a total function $f: D \to \mathcal{B}^+$, and let $\phi \in \mathcal{B}^+$ $(\phi \neq \alpha)$. For each $d \in D$, let $n(d) \in \mathbb{N}$ such that $n(d) > \max D(f(d))$. Then a representation $\rho: D \to \mathcal{B}^+$ which is defined by

$$\rho(d) = f(d) \cup \{\phi\}^{n(d)}$$

is an endmarker representation. The relation ϕ is known as the endmarker, and the function f is the list component of ρ.

To illustrate what this definition says, we present the following example.

Example 5.16. Let $X = \{a,b,c\}$, $\mathcal{B} = \{0,1\}$, and $D = \bigcup_{i \in J} X^i$. Define the function $f: X \cup \{\alpha\} \to \mathcal{B}^+$ by

$$\begin{aligned} f(a) &= 0\_0 \\ f(b) &= 10\_ \\ f(c) &= 11\_ \\ f(\alpha) &= 0\_1 \end{aligned}$$

If we then define $f': D \to \mathcal{B}^+$ by

$$f'(d) = \bigcup_{i=1}^{|d|} \{(f(d(i)))^{3(i-1)}\},$$

then the representation

$$\rho(d) = f'(d) \cup \{f(\alpha)\}^{3|d|}$$

is a concatenation-preserving endmarker representation. For $D = \bigcup_{i=0}^{\infty} X^i$, it is easy to verify that ρ achieves Kraft storage, since

$$|\rho(d)| = 2 \cdot |d| + 2.$$

5.3 Endmarker Representations

Recall from Section 5.1 that an endmarker representation has some fixed symbol or sequence of symbols in $\mathcal{B}^+$ which are always at the "end" of the list. Example 5.5a is an example of an endmarker representation. The representation ρ in Example 2.7 is also an endmarker representation, with endmarker $\Diamond$. We now give a formal definition.

Definition. Let f be a total function $f:\mathbb{D} \to \mathcal{B}^+$, and let $\Diamond \in \mathcal{B}^+$ $(\Diamond \neq \varnothing)$. For each $d \in \mathbb{D}$, let $n(d) \in \mathbb{N}$ such that $n(d) > \max D(f(d))$. Then a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which is defined by

$$\rho(d) = f(d) \cup \{\Diamond\}_{n(d)}$$

is an *endmarker representation.* The relation $\Diamond$ is known as the *endmarker*, and the function f is the *list component* of ρ.

To illustrate what this definition says, we present the following example.

Example 5.16. Let $X = \{a,b,c\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \bigcup_{i \in J} X^i$. Define the function $f:X \cup \{\varnothing\} \to \mathcal{B}^+$ by

$$\begin{aligned} f(a) &= 0\_0 \\ f(b) &= 10\_ \\ f(c) &= 11\_ \\ f(\varnothing) &= 0\_1 \end{aligned}$$

If we then define $f':\mathbb{D} \to \mathcal{B}^+$ by

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{3(i-1)},$$

then the representation

$$\rho(d) = f'(d) \cup \{f(\varnothing)\}_{3|d|}$$

is a concatenation-preserving endmarker representation. For $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, it is easy to verify that ρ achieves Kraft storage, since

$$|\rho(d)| = 2 \cdot |d| + 2.$$

Thus,

$$\sum_{d \in |D} 2^{-|\rho(d)|} = \sum_{i \in J} \sum_{d \in X^i} 2^{-(2|d| + 2)}$$

$$= \sum_{i=0}^{\infty} |X|^i \cdot 2^{-(2i + 2)}$$

$$= \frac{1}{4} \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i$$

$$= 1 .$$

Note that no finite $|J|$ will give us Kraft storage.

Now consider answering a table lookup question $\gamma_i \in \Gamma$. For γ_1 we need only access $m(0)$ and $m(1)$, or else $m(0)$ and $m(2)$. On the other hand, to answer the question γ_2, accessing just $m(3)$ and $m(4)$ (or else $m(3)$ and $m(5)$) may not give the correct answer. In particular, unless we have already determined that the answer is $\varnothing$, then we must verify that $|d| \geq 1$. This requres accessing $m(0)$ and possibly $m(2)$. Possible access trees T_i for each γ_i can be constructed as indicated in Figure 5.7, where we write $\{T_i\}_k$ to denote the tree T_i with each node label j replaced by the label j + k. These trees correspond to reading the necessary memory cells in a left to right order. It would also be possible to read the cells essentially from right to left. For either method, once $f(\varnothing)$ is encountered for γ_i, then it is known that $|d| < i$. ∎

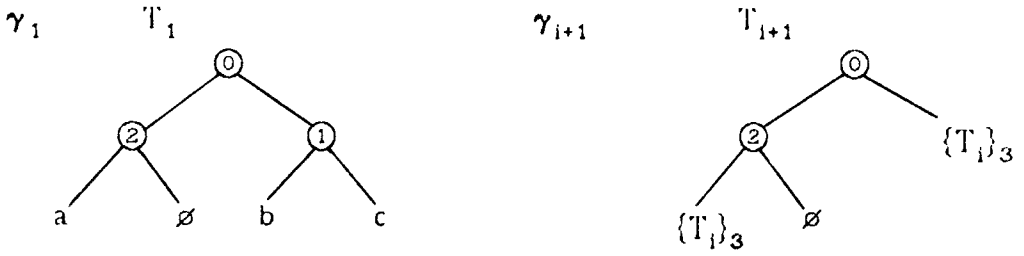


Figure 5.7. Trees for γ_i of Example 5.16.

The endmarker representations we have thus far seen are all concatenation-preserving representations, but there is no such requirement in the definition. In fact, there is not even any requirement that the endmarker be necessary; i.e., for an endmarker representation $\rho(d) = f(d) \cup \{\Diamond\}_{n(d)}$ it may be the case that $f(\mathbb{D})$ itself is a representation and thus the endmarker $\Diamond$ is superfluous. Also, there is no restriction that the endmarker not appear in $f(d)$. Even if the pattern $\Diamond \in \mathcal{B}^+$ does not appear in $f(d)$, there may be "holes" in $f(d)$, which allow the possibility of another user writing $\Diamond$. Thus, it may not be the case that the first occurrence of $\Diamond$ serves as the endmarker.

Example 5.17. Let $\mathcal{B} = \{0,1\}$ and, for $\mathbb{D} = \{d_1, d_2, d_3, d_4, d_5\}$, define the function $f:\mathbb{D} \to \mathcal{B}^+$ by

$$\begin{aligned} f(d_1) &= 0\_0 \\ f(d_2) &= 0\_1 \\ f(d_3) &= 10\_ \\ f(d_4) &= \_10 \\ f(d_5) &= 11\_ \end{aligned}$$

If we let $\Diamond = \{(0,1), (1,0)\}$, we can then define the endmarker representation $\rho:\mathbb{D} \to \mathcal{B}^+$ by

$$\begin{aligned} \rho(d_1) &= 0\_0\_10 \\ \rho(d_2) &= 0\_1\_10 \\ \rho(d_3) &= 1010 \\ \rho(d_4) &= \_1010 \\ \rho(d_5) &= 1110 \end{aligned}$$

The endmarker here is not superfluous because it does enable us to distinguish between $\rho(d_1)$ and $\rho(d_4)$ and between $\rho(d_4)$ and $\rho(d_5)$. On the other hand, even if we were to eliminate d_4 from the domain, ρ would still be an endmarker representation. Notice also that $\rho(d_1) = 0\_0\_10$, and thus if another user sets $m(1) = 1$ then the actual endmarker is not the first occurrence of $\Diamond$. In fact, $f(d_3)$ itself contains the set $\Diamond$. ∎

We usually have in mind a more restricted notion of an endmarker representation, where we require $\Diamond$ to be distinguishable and reserve $\Diamond$ solely to indicate the end of the list. (Of course, if the representation has holes in it, then it is still possible for other users to write $\Diamond$.) Thus, if we read a list representation from left to right and access no cells not in the representation, then encountering $\Diamond$ immediately tells us when we've reached the end. Most of our examples will be of this form.

Notice that the function f' in Example 5.16 has fixed position fields. However, the endmarker in the representation ρ has a displacement function $n(d) = 3 \cdot |d|$. So n is not a constant function, and the endmarker is not always in the same memory position. In fact, if the endmarker were always at the same location, then there would be no point in having an endmarker at all; there is no such concatenation-preserving endmarker representation. We make the following definition.

Definition. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be a concatenation-preserving endmarker representation, with endmarker $\Diamond$, and formed from a concatenation-preserving function f' with fixed position fields n_i. If ρ is of the form

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{|d|+1}}$$

then ρ is said to be a *fixed position field endmarker representation.*

Thus, the representation ρ in Example 5.16 is a fixed position field endmarker representation.

In Example 5.16 we saw an endmarker representation that achieves Kraft storage when $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$. We can, in fact, show that achieving Kraft storage implies that $\max_{d \in \mathbb{D}} |\rho(d)|$ is unbounded.

Theorem 5.7. If an endmarker representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ achieves Kraft storage, then $\neg(\exists n \in \mathbb{N})(\forall d \in \mathbb{D})(|\rho(d)| \leq n)$.

Proof: Assume that $(\exists n \in \mathbb{N})(\forall d \in \mathbb{D})(|\rho(d)| \leq n)$.

Then it is possible to choose $d_k \in \mathbb{D}$ such that

$$\max D(\rho(d_k)) = \max_{d \in \mathbb{D}} D(\rho(d)).$$

Thus, no $\rho(d)$ occupies a larger memory cell location than $\rho(d_k)$. By the definition of an endmarker representation, there is some function f such that

$$\rho(d_k) = f(d_k) \cup \{\Diamond\}_{n(d_k)}.$$

Now let $r = \min D(\Diamond)$, and choose $b_0 \in \mathcal{B}$ such that b_0 is not a prefix of $\Diamond$. (Since $|\mathcal{B}| \geq 2$, there must always be such a b_0.) Consider the string

$$b = f(d_k) \cup \{(n(d_k) + r, b_0)\} \in \mathcal{B}^+$$

For all $d_i \in \mathbb{D}$, b and $\rho(d_i)$ are distinguishable. In other words, there is no $d_i \in \mathbb{D}$ such that $b \subseteq \bar{\rho}_L(d_i)$. So by Theorem 3.3 ρ does not achieve Kraft storage. Thus, our original assumption must have been wrong, and we conclude that

$$\neg(\exists n \in \mathbb{N})(\forall d \in \mathbb{D})(|\rho(d)| \leq n). \quad \blacksquare$$

It immediately follows that if an endmarker representation achieves Kraft storage, then the domain $\mathbb{D}$ must be infinite and also that the index set $\mathcal{J}$ must be infinite.

Corollary 5.7.1. If an endmarker representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ achieves Kraft storage, then $\neg(\exists n \in \mathbb{N})(|\mathbb{D}| \leq n)$.

Corollary 5.7.2. If an endmarker representation $\rho:\mathbb{D} \rightarrow \mathcal{B}^+$ achieves Kraft storage, then $\neg(\exists n \in \mathbb{N})(\max_{i \in \mathcal{J}} i \leq n)$.

Thus, when we are discussing endmarker representations, we frequently consider $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$.

Notice that since achieving Kraft storage tells us that the domain must be infinite, we immediately know that no endmarker representation can achieve both Kraft storage and Kraft access.

Theorem 5.8. There is no endmarker representation that achieves Kraft storage and also achieves Kraft access for all $\gamma_i \in \Gamma$.

Proof: From theorems 4.9 and 4.10, we know that if a representation ρ achieves Kraft storage and Kraft access for all $\gamma_i \in \Gamma$, then $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X^n$. But by Corollary 5.7.1 we know that $|\mathbb{D}|$ cannot be finite for an endmarker representation that achieves Kraft storage. Thus, there is no endmarker representation that achieves both Kraft storage and Kraft access. ∎

Recall again the representation ρ in Example 5.16, which achieved Kraft storage. We can show that this result generalizes. In particular, given any representation $f: X \cup \{\varnothing\} \to \mathcal{B}^+$ which achieves Kraft storage, a concatenation-preserving endmarker representation ρ formed from f also achieves Kraft storage. Before we prove this, however, we introduce some terminology and prove a lemma. We begin with the following definition.

Definition. Consider a full $|\mathcal{B}|$-ary tree T' with $|\mathcal{B}|^k$ nodes at depth k, for all $k \in \mathbb{N}$. Assume that some of the (internal) nodes are labelled $\varnothing$ but that T' has the property that if a node is labelled $\varnothing$ then none of the descendants of that node is labelled. We use the term *$\varnothing$-node* to refer to a node labelled $\varnothing$ or the descendant of a node labelled $\varnothing$. We then let ξ denote the fraction of the nodes in T' at depth k that are $\varnothing$-nodes.

Since ξ is a fraction of nodes that are $\varnothing$-nodes, it is clear that $0 \leq \xi(k) \leq 1$. Also $\xi(k+1) \geq \xi(k)$, since a $\varnothing$-node at some level leads to the same fraction of $\varnothing$-node descendants at the next level. The following example should clarify what is

meant by a $\varnothing$-node and by $\xi(k)$.

Example 5.18. Consider the tree T' in Figure 5.8. For simplicity we have deleted the node labels indicating memory cell locations. We have, however, retained the external label $\varnothing$ on certain nodes and marked each $\varnothing$-node with an "x". Notice that all descendants of nodes labelled $\varnothing$ are themselves $\varnothing$-nodes. There are 1 $\varnothing$-node at depth 2, 3 $\varnothing$-nodes at depth 3, 8 $\varnothing$-nodes at depth 4, 19 $\varnothing$-nodes at depth 5, etc. Thus

$$\xi(0) = \xi(1) = 0$$
$$\xi(2) = \frac{1}{4}$$
$$\xi(3) = \xi(2) + \frac{1}{8} = \frac{3}{8}$$
$$\xi(4) = \xi(3) + \frac{2}{16} = \frac{1}{2}$$
$$\xi(5) = \xi(4) + \frac{3}{32} = \frac{19}{32}$$

We shall have occasion to refer back to this tree T' in a later example. ∎

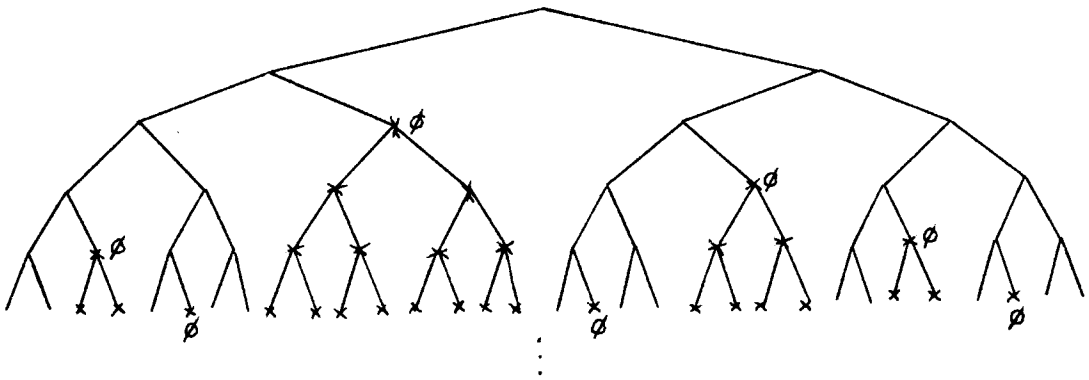


Figure 5.8. Tree T' from Example 5.18.

In order to motivate some of the terminology used in the next lemma, let us consider another example.

Example 5.19. Let $X = \{a,b\}$, $\mathcal{B} = \{0,1\}$, and consider the representation $f: X \cup \{\varnothing\} \to \mathcal{B}^*$ defined by

$$\begin{aligned} f(a) &= 1 \\ f(b) &= 00 \\ f(\varnothing) &= 01 \end{aligned}$$

Then f achieves Kraft storage and corresponds to the tree T_f shown in Figure 5.9a. Now, for $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, define a concatenation-preserving endmarker representation $\rho: \mathbb{D} \to \mathcal{B}^*$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)} \cup \{(0,0), (1,1)\}_{n(d)}$$

where $n_i(d) = \sum_{j=1}^{i-1} |f(d(j))|$ and $n(d) = \sum_{j=1}^{|d|} |f(d(j))|$. Then we can construct a tree T for representation ρ as in Figure 5.9b. ∎

We can now prove the following lemma.

Lemma 5.5. Consider a prefix representation $f: X \cup \{\varnothing\} \to \mathcal{B}^*$ which achieves Kraft storage. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a concatenation-preserving endmarker representation $\rho: \mathbb{D} \to \mathcal{B}^+$ defined by

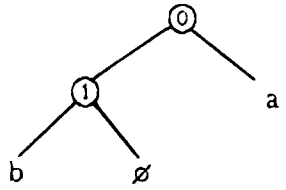
$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)} \cup \{f(\varnothing)\}_{n(d)},$$

where $n_i: \mathbb{D} \to \mathbb{N}$, $n: \mathbb{D} \to \mathbb{N}$. Let T be a $|\mathcal{B}|$-ary tree corresponding to ρ, and let T′ be an extension of T which keeps the $\varnothing$-node labels of T but extends the tree so that T′ has $|\mathcal{B}|^k$ nodes at depth k, for all $k \in \mathbb{N}$, and the $\varnothing$ labels now label internal nodes. Then

$$\lim_{k \to \infty} \xi(k) = 1.$$

Proof: Since the prefix representation f achieves Kraft storage, there is a corresponding full $|\mathcal{B}|$-ary tree T_f, as shown in Figure 5.9a. Assume that $|f(\varnothing)| = r$ and that $\max_{x \in X \cup \{\varnothing\}} |f(x)| = p$. Then T_f has (maximum) depth p and the depth of its $\varnothing$-node is r. The tree T corresponding to ρ is formed from T_f by

(a) T_f



(b) T

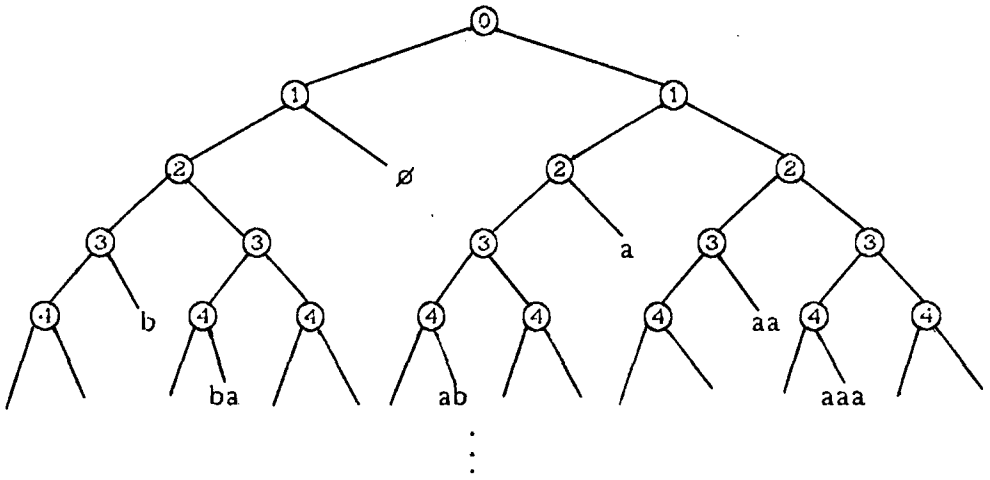


Figure 5.9. Trees T_f and T from Example 5.19.

placing a copy of T_f at each leaf not labelled $\varnothing$ and doing this indefinitely. (The memory cells to be accessed need to be altered according to the values of $n_i(d)$ and $n(d)$. Since we know, however, that no path will contain the same memory location twice, we choose to ignore these access labels and are concerned only with the external labels at a $\varnothing$-node indicating the d such that $\rho(d)$ leads to this node.) T' is the extension of T where we keep the $\varnothing$-node leaf labels but extend from each of these leaves a full $|\mathcal{B}|$-ary tree. Thus, for all $k \in \mathbb{N}$, T' has $|\mathcal{B}|^k$ nodes at depth k.

We are now ready to determine $\lim_{k\to\infty} \xi(k)$. It is clear that $\xi(i+1) \geq \xi(i)$, because if there are j $\varnothing$-nodes at depth i, then there are $|\mathcal{B}|\cdot j$ descendants of these $\varnothing$-nodes at depth $i+1$. Thus, the fraction of these $\varnothing$-nodes cannot decrease.

Also, there may be more $\varnothing$-nodes at depth $i+1$, corresponding to copies of T_f with leaves labelled $\varnothing$ at depth $i+1$. Each node at depth i which is not a $\varnothing$-node will have a descendant within depth p which is a $\varnothing$-node. Thus, at least $|\mathcal{B}|^{-p}$ descendants of non $\varnothing$-nodes at depth i will themselves be $\varnothing$-nodes at depth $i+p$. Since the fraction of non $\varnothing$-nodes at depth i is $1 - \xi(i)$,

$$\begin{aligned}\xi(i+p) &\geq \xi(i) + |\mathcal{B}|^{-p}(1 - \xi(i)) \\ &= \frac{1}{|\mathcal{B}|^p} + \frac{|\mathcal{B}|^p - 1}{|\mathcal{B}|^p} \cdot \xi(i)\end{aligned}$$

If we look at the values of $\xi(k)$ at depths 0, p, $2\cdot p$, etc., we find that

$$\begin{aligned}\xi(k\cdot p) &\geq \frac{1}{|\mathcal{B}|^p} \cdot \sum_{j=0}^{k-1} \left(\frac{|\mathcal{B}|^p - 1}{|\mathcal{B}|^p}\right)^j \\ &= 1 - \left(\frac{|\mathcal{B}|^p - 1}{|\mathcal{B}|^p}\right)^k\end{aligned}$$

Of course we know that $\xi(k\cdot p) \leq 1$, and so we conclude that

$$\lim_{k\to\infty} \xi(k) = 1. \qquad \blacksquare$$

To illustrate the method used in proving Lemma 5.5, we refer back to Example 5.19.

Example 5.20. Recall the representation ρ from Example 5.19. The extension of T to a tree T' with $|\mathcal{B}|^k$ nodes at depth k is the tree T' of Example 5.18, shown in Figure 5.8. The tree T_f, from which T and T' were constructed, has maximum depth 2, and the depth of its $\varnothing$-node is 2. We want to verify that

$$\xi(i+2) \geq \xi(i) + \frac{1}{4} \cdot (1 - \xi(i)).$$

The fraction of non $\varnothing$-nodes at depth i is $1 - \xi(i)$. Every non $\varnothing$-node at depth i serves either as a root of another copy of T_f (see node A in Figure 5.10a) or else is an internal node of some T_f copy (see node B of Figure 5.10b). In the former case, we get a new $\varnothing$-node at depth $i+2$. In the latter case, we get a new $\varnothing$-node at depth $i+1$, which gives us two additional $\varnothing$-nodes at depth $i+2$. $\blacksquare$

Lemma 5.5 allows us to prove the following result.

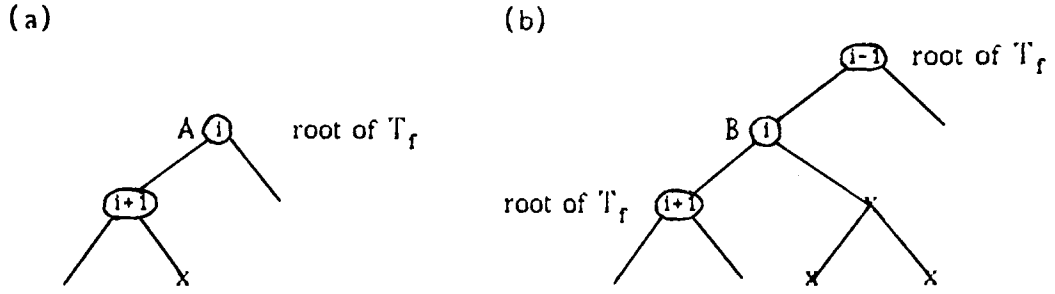


Figure 5.10. Origination of new $\varnothing$-nodes at depth i + 2.

Theorem 5.9. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, and consider a representation $f: X \cup \{\varnothing\} \to \mathcal{B}^*$ which achieves Kraft storage. Assume that the set $f(X \cup \{\varnothing\})$ forms a prefix code, and let $\rho: \mathbb{D} \to \mathcal{B}^+$ be a concatenation-preserving endmarker representation defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)} \cup \{f(\varnothing)\}_{n(d)},$$

where $n_i: \mathbb{D} \to \mathbb{N}$, $n: \mathbb{D} \to \mathbb{N}$. Then ρ achieves Kraft storage.

Proof: Let $\psi(i)$ be the distribution function

$$\psi(i) \triangleq |\{\rho(d) \mid |\rho(d)| = i\}|.$$

So $\psi(i)$ corresponds to the number of $\varnothing$-nodes at depth i that have no $\varnothing$-node ancestors. Then

$$\begin{aligned} \sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} &= \sum_{i=0}^{\infty} \psi(i) \cdot |\mathcal{B}|^{-i} \\ &= \lim_{k \to \infty} \sum_{i=0}^{k} \psi(i) \cdot |\mathcal{B}|^{-i} \\ &= \lim_{k \to \infty} |\mathcal{B}|^{-k} \sum_{i=0}^{k} \psi(i) \cdot |\mathcal{B}|^{k-i} \end{aligned}$$

A $\varnothing$-node at depth i is an ancestor of $|\mathcal{B}|^{k-i}$ descendants at depth k, and so there are $\sum_{i=0}^{k} \psi(i) \cdot |\mathcal{B}|^{k-i}$ $\varnothing$-nodes at depth k. Since at depth k there are a total of $|\mathcal{B}|^k$ nodes, the fraction of nodes at depth k which are $\varnothing$-nodes is

$$|\mathcal{B}|^{-k} \sum_{i=0}^{k} \psi(i) \cdot |\mathcal{B}|^{k-i} = \xi(k).$$

Applying Lemma 5.5 gives the desired result:

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = \lim_{k \to \infty} \xi(k) = 1. \quad \blacksquare$$

The above theorem still holds if we do not require that $f(X \cup \{\varnothing\})$ be a prefix code.

Theorem 5.10. Consider a representation $f: X \cup \{\varnothing\} \to \mathcal{B}^+$ which achieves Kraft storage. For $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, let $\rho: \mathbb{D} \to \mathcal{B}^+$ be a concatenation-preserving endmarker representation, where

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_1(d)} \cup \{f(\varnothing)\}_{n(d)},$$

for $n_1: \mathbb{D} \to \mathbb{N}$, $n: \mathbb{D} \to \mathbb{N}$. Then ρ achieves Kraft storage.

Proof: Consider any representation $f_1: X \cup \{\varnothing\} \to \mathcal{B}^+$, and recall from Chapter 3 the statement of the Kraft inequality. If f_1 achieves Kraft storage, then

$$\sum_{x \in X \cup \{\varnothing\}} |\mathcal{B}|^{-|f_1(x)|} = 1$$

and the Kraft inequality is satisfied (with equality). Thus, there is some function $f_2: X \cup \{\varnothing\} \to \mathcal{B}^+$ such that $f_2(X \cup \{\varnothing\})$ is a prefix code and $|f_2(x)| = |f_1(x)|$ for all $x \in X \cup \{\varnothing\}$. By Theorem 5.9 we know that for any concatenation-preserving representation ρ_1 formed from f_2,

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho_1(d)|} = 1 = \sum_{d \in \mathbb{D}} |\mathcal{B}|^{-(\sum |f_2(d(i))| + |f(\varnothing)|)}$$

Since $$|\rho(d)| = \sum_{i=1}^{|d|} |f_1(d(i))| + |f(\varnothing)| = \sum_{i=1}^{|d|} |f_2(d(i))| + |f(\varnothing)|,$$

we can conclude that

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = 1$$

and so ρ achieves Kraft storage. $\blacksquare$

We can verify directly that the representation ρ from examples 5.19 and 5.20 achieves Kraft storage.

Example 5.21. What we want to show is that

$$\sum_{d \in \mathbb{D}} 2^{-|\rho(d)|} = \sum_{i=0}^{\infty} \psi(i) \cdot 2^{-i} = 1.$$

Referring to the tree T of Figure 5.9b, we see that

$$\begin{aligned} \psi(0) &= \psi(1) = 0 \\ \psi(2) &= 1 \\ \psi(3) &= 1 \\ \psi(4) &= 2 \\ \psi(5) &= 3 \end{aligned}$$

In fact, whenever a copy of T_f terminates at depth i, then there is a leaf from T_f at depth i-1 which serves as the root of another copy of T_f, one which has a $\varnothing$-leaf at depth i + 1. Similarly, if a copy of T_f terminates at depth i - 1, then there is a leaf of T_f also at depth i - 1 which serves as the root of a copy of T_f, leading to a $\varnothing$-leaf at depth i + 1. Thus, we can define the distribution function ψ by

$$\begin{aligned} \psi(1) &= 0 \\ \psi(2) &= 1 \\ \psi(i+1) &= \psi(i) + \psi(i-1) \end{aligned}$$

Solving this Fibonacci expression, we find that

$$\psi(i) = \frac{5 - \sqrt{5}}{10} \cdot \left(\frac{1 + \sqrt{5}}{2}\right)^i + \frac{5 + \sqrt{5}}{10} \cdot \left(\frac{1 - \sqrt{5}}{2}\right)^i$$

for $i \geq 1$. Thus, we can directly show that ρ achieves Kraft storage.

$$\begin{aligned} \sum_{i=0}^{\infty} \psi(i) \cdot 2^{-i} &= \sum_{i=1}^{\infty} \left[\frac{5 - \sqrt{5}}{10} \cdot \left(\frac{1 + \sqrt{5}}{2}\right)^i + \frac{5 + \sqrt{5}}{10} \cdot \left(\frac{1 - \sqrt{5}}{2}\right)^i\right] \cdot 2^{-i} \\ &= \frac{5 - \sqrt{5}}{10} \cdot \sum_{i=1}^{\infty} \left(\frac{1 + \sqrt{5}}{4}\right)^i + \frac{5 + \sqrt{5}}{10} \cdot \sum_{i=1}^{\infty} \left(\frac{1 - \sqrt{5}}{4}\right)^i \\ &= \frac{5 - \sqrt{5}}{10} \cdot \frac{1 + \sqrt{5}}{3 - \sqrt{5}} + \frac{5 + \sqrt{5}}{10} \cdot \frac{1 - \sqrt{5}}{3 + \sqrt{5}} \\ &= 1. \end{aligned}$$

∎

As an aside for interested number theorists, notice that the sum in Example 5.21 holds for $\psi(i)$ any extended Fibonacci sequence.

Corollary 5.10.1. Let $fib_n(i) \triangleq fib_n(i-1) + fib_n(i-2) + \ldots + fib_n(i-n)$.

Then $$\sum_{i=0}^{\infty} fib_n(i) \cdot 2^{-i} = 1.$$

Proof: Consider a binary tree T_f of the form shown in Figure 5.9a, which has internal node labels 0, 1, . . . , n-1 (for $0 \leq i < n$, there is one node at depth i, and that node has label i) and has one leaf at each of the depths 1, 2, 3, . . . , n-1 and two leaves at depth n. Consider the extension T of T_f, as in Figure 5.9b. If a copy of T_f has a $\varnothing$-node at depth i - k, for $1 \leq k \leq n$, then that copy of T_f has its root at depth i - n - k and thus has a node at depth i - n which is not a $\varnothing$-node. This node, not itself a $\varnothing$-node, must serve as the root of yet another copy of T_f and this new copy of T_f has a $\varnothing$-leaf at depth i. Thus

$$\psi(i) = \psi(i-1) + \psi(i-2) + \ldots + \psi(i-n).$$

But by Theorem 5.10 we know that the extension T of T_f corresponds to a representation ρ which achieves Kraft storage. Thus

$$\sum_{i=0}^{\infty} \psi(i) \cdot 2^{-i} = 1 = \sum_{i=0}^{\infty} fib_n(i) \cdot 2^{-i}. \qquad \blacksquare$$

5.4 Pointer Representations

Recall from Section 5.1 that a pointer representation has some function $\ell:\mathcal{J} \to \mathcal{B}^+$ which serves as a pointer and indicates the length $|d|$. Example 5.5b gave an example of a pointer representation, and we now give a formal definition.

Definition. Let $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$, let f be a total function $f:\mathbb{D} \to \mathcal{B}^+$, and let $\ell:\mathcal{J} \to \mathcal{B}^+$ be a representation. Then a representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which is defined by

$$\rho(d) = \{f(d)\}_{n_1(d)} \cup \{\ell(|d|)\}_{n_2(d)}$$

is a *pointer representation* if

$$D(\{f(d)\}_{n_1(d)}) \cap D(\{\ell(|d|)\}_{n_2(d)}) = \varnothing,$$

where n_1, n_2 are functions, $n_1:\mathbb{D} \to \mathbb{N}$, $n_2:\mathbb{D} \to \mathbb{N}$. The function f is the *list component* of ρ and ℓ is the *pointer component* of ρ. We refer to $\ell(|d|)$ as the *pointer* of $\rho(d)$.

Note that the functions n_1, n_2 in the above definition are not the same functions as the n_i in the definition of a concatenation-preserving function. Before discussing the pointer representation in more detail, let us present the following example in order to illustrate the definition.

Example 5.22. Let $X = \{a,b,c,d\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$. Define the function $f:X \to \mathcal{B}^+$ by

$$\begin{aligned} f(a) &= 0\_0 \\ f(b) &= 10\_ \\ f(c) &= 11\_ \\ f(d) &= 0\_1 \end{aligned}$$

and then define the concatenation-preserving function $f':\mathbb{D} \to \mathcal{B}^+$ by

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{3(i-1)}.$$

The pointer $\ell:\mathcal{J} \to \mathcal{B}^+$ is defined by

$$\ell(i) = 1^i0.$$

Then the representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$ where

$$\rho(d) = \{\ell(|d|)\}_0 \cup \{f'(d)\}_{|d|+1}$$

is a pointer representation. For instance,

$$\begin{aligned} \rho(\text{abad}) &= 111100\_010\_0\_00\_1 \\ \rho(\text{bdb}) &= 111010\_0\_110\_ \\ \rho(\lambda) &= 0. \end{aligned}$$

Notice that

$$|\rho(d)| = |d| + 1 + |f'(d)| = 3\cdot|d| + 1.$$

For $\mathcal{J} = \mathbb{N}$, then $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and it is easy to verify that ρ achieves Kraft storage:

$$\begin{aligned} \sum_{d\in\mathbb{D}} 2^{-|\rho(d)|} &= \sum_{i\in\mathcal{J}} \sum_{d\in X^i} 2^{-(3i+1)} \\ &= \sum_{i=0}^{\infty} |X|^i \cdot 2^{-(3i+1)} \\ &= \frac{1}{2}\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \\ &= 1 . \end{aligned}$$

Now consider answering the table lookup question $\gamma_i \in \Gamma$. The answer to the question γ_i is essentially found at memory locations beginning with cell $3\cdot(i-1)$, except that we have stored the pointer in front of $f(d)$, and so $f(d)$ has been displaced by $|d| + 1$ cells. Thus, the answer to γ_i, for $i \le |d|$, is found by reading $m(|d|+1+3(i-1))$ and then reading either $m(|d|+1+3(i-1)+1)$ or $m(|d|+1+3(i-1)+2)$. When $i > |d|$, we need only read the pointer to determine that the answer is $\varnothing$. One possible algorithm to answer the question γ_i therefore has the memory cell access sequences:

$0, 1, \ldots, \lvert d\rvert-1, \lvert d\rvert, \lvert d\rvert+3(i-1)+1, \lvert d\rvert+3(i-1)+2$	if $m(\lvert d\rvert+3i-2) = 1, \lvert d\rvert \ge i$
$0, 1, \ldots, \lvert d\rvert-1, \lvert d\rvert, \lvert d\rvert+3(i-1)+1, \lvert d\rvert+3(i-1)+3$	if $m(\lvert d\rvert+3i-2) = 0, \lvert d\rvert \ge i$
$0, 1, \ldots, \lvert d\rvert-1, \lvert d\rvert$	if $\lvert d\rvert < i$

This immediately tells us the total number of accesses made:

$$\#[\gamma_i(\rho(d))] = \begin{cases} |d| + 3 & \text{if } |d| \geq i \\ |d| + 1 & \text{if } |d| < i \end{cases}$$

∎

The intuition behind the definition of a pointer representation is that we encode the length so that in order to answer a question γ_i we need only read the pointer and can then look up the answer. In the case of the endmarker representation, we were forced to actually read the list. The question remains, however, why we chose to allow the pointer to encode $|d|$ rather than $|\rho(d)|$. If we wish to be able to access individual list elements, as by asking the questions in Γ, then it is reasonable to encode $|d|$. Reading the pointer will then at least tell us immediately whether the answer to γ_i is $\varnothing$ or not. On the other hand, if we wish to perform the update operation of appending an element to the end of the list, then it would be advantageous to know $|\rho(d)|$. If

$$(\forall d_1, d_2 \in \mathbb{D})(|d_1| = |d_2| \Rightarrow |f(d_1)| = |f(d_2)|)$$

then it of course makes no difference whether the pointer encodes $|d|$ or $|f(d)|$, since we can determine one from the other.

Example 5.23. Reconsider the functions f and g from Example 5.22 but define a partial function $\ell' : \mathbb{N} \to \mathcal{B}^+$ such that $D(\ell') = \{2i \mid i \in \mathcal{J}\}$, the even natural numbers, and $\ell'(n) = 1^{(\frac{n}{2})}0$. Notice that ℓ' is a representation. So the representation $\rho' : \mathbb{D} \to \mathcal{B}^+$ defined by

$$\rho'(d) = \{\ell'(|f(d)|)\}_0 \cup \{f(d)\}_{|d|+1}$$

is equivalent to the representation ρ in Example 5.22 because

$$\ell'(|f(d)|) = \ell(|d|).$$

Technically, however, the representation ρ' is not a pointer representation because $\ell' : \{2i \mid i \in \mathcal{J}\} \to \mathcal{B}^+$, whereas the definition requires that $\ell : \mathcal{J} \to \mathcal{B}^+$. But since $|D(\ell')| = |D(\ell)| = |\mathcal{J}|$, we often find it convenient to loosely refer to ρ' as a pointer representation itself. ∎

We could have written the formal definition of a pointer to allow a mapping $\ell' : \mathbb{N} \to \mathcal{B}^\dagger$ where $|D(\ell')| = |\mathcal{J}|$, but we chose not to since the added generality would make the definition statement more complex and would not improve our results.

We shall, however, allow one conceptual extension for pointer representations. Since we require the pointer and list components, $\ell(|d|)$ and $f(d)$, to be placed in memory so as to not overlap, we may want to view them as being stored in separate sections of memory. In other words, we could view $f(d)$ as being stored in memory as usual and $\ell(|d|)$ as being stored in an auxiliary section of memory, perhaps some sort of register. However, we shall not in general want to bound the size of the pointer and we do not differentiate between the cost of a pointer access vs. the cost of a list access, so it is easier to view the pointer as also being in memory. We simply assume the memory manager allocates the list and the pointer separate areas. Perhaps they are even interspersed, but we do not want to have to alter our coding schemes to take this into account. Therefore for numbering simplicity we may choose to allow both the list and the pointer to begin at cell number 0 and just note that the representations are separate and therefore disjoint. In this way, the storage of $f(d)$ in memory does not have to depend on the memory location of $\ell(|d|)$.

> **Definition.** Let f be a total function $f : \mathbb{D} \to \mathcal{B}^\dagger$, and let ℓ be a representation $\ell : \mathcal{J} \to \mathcal{B}^\dagger$. Assume that $f(d)$ and $\ell(|d|)$ are stored in separate sections of memory. Then we refer to a pointer representation $\rho : \mathbb{D} \to \mathcal{B}^\dagger$ formed from ℓ and f as a *separate pointer representation* and write
>
> $$\rho(d) = f(d) \cup \ell(|d|)$$

In order to avoid possible confusion, when we have in mind a separate pointer representataion, we shall explicitly say so.

Example 5.24. Reconsider the pointer representation ρ of Example 5.22, but assume that the pointer and the list components are stored in separate memory sections. So ρ is a separate pointer representation, and we denote it by

$$\rho(d) = \ell(|d|) \cup f'(d).$$

Certainly we have not altered the storage costs from those of Example 5.22, but it is possible to implement each γ_i in such a way that we decrease the access costs. Possible access sequences for $\gamma_i \in \Gamma$ are:

$0, 1, \ldots, i\text{-}1, 3(i\text{-}1), 3(i\text{-}1)+1$	if $m(3(i\text{-}1)) = 1$, $\|d\| \geq i$
$0, 1, \ldots, i\text{-}1, 3(i\text{-}1), 3(i\text{-}1)+2$	if $m(3(i\text{-}1)) = 0$, $\|d\| \geq i$
$0, 1, \ldots, \|d\|\text{-}1, \|d\|$	if $\|d\| < i$

Thus we have for the total number of accesses:

$$\#[\gamma_i(\rho(d))] = \begin{cases} i + 2 & \text{if } |d| \geq i \\ |d| + 1 & \text{if } |d| < i \end{cases}$$

Notice that this represents an improvement over the access costs we previously had. ∎

Although we shall not in general concern ourselves with the way in which separate memory sections are allocated, let us note, in the context of this same example, one possible scheme.

Example 5.25. Let f be defined as in Example 5.22, but now define the concatenation-preserving representation $f_1: \mathbb{D} \to \mathcal{B}^+$ by

$$f_1(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{4(i-1)}.$$

If we view the pointer ℓ_1 as being "scattered", we may define

$$\ell_1(i) = \{(3+4j, 1) \mid 0 \leq j < i\} \cup \{(3+4i, 0)\}.$$

Then the pointer representation $\rho_1: \mathbb{D} \to \mathcal{B}^+$ is defined by

$$\rho_1(d) = \{\ell_1(|d|)\}_0 \cup \{f_1(d)\}_0.$$

For instance,

$$\rho_1(\text{abad}) = 0\_0110\_10010\_11\_\_\_0$$
$$\rho_1(\text{bdb}) = 10\_10\_1110\_1\_\_\_0$$
$$\rho_1(\lambda) = \_\_\_0.$$

Since we are counting only the actual number of cells occupied, the storage has not been altered. For $\gamma_1 \in \Gamma$ we have the following access sequences:

3, 7, 11, . . . , 3+4(i-1), 4i-4, 4i-3	if $m(4i\text{-}4) = 1$, $\|d\| \geq i$
3, 7, 11, . . . , 3+4(i-1), 4i-4, 4i-2	if $m(4i\text{-}4) = 0$, $\|d\| \geq i$
3, 7, 11, . . . , 3+4(|d|-1), 3+4|d|	if $\|d\| < i$

This gives us the same total number of accesses as we had in Example 5.24, where we simply made the assumption that we had separate memory sections. ∎

Example 5.25 illustrates an encoding for a separate pointer scheme. Notice that this encoding did not affect the order in which memory cell contents were determined; it simply altered the memory cell numbers in which this information was found. We can show in general that there is no harm in using a separate pointer scheme if it makes our coding job easier, because for any separate pointer representation ρ there is a pointer representation ρ' without a separate pointer that achieves the same storage and access costs.

Theorem 5.11. Given any pointer representation ρ with a separate pointer, there exists a pointer representation ρ' without a separate pointer such that

$$|\rho(d)| = |\rho'(d)| \quad \text{for all } d \in \mathbb{D}$$

and $$\#[f_i(\rho(d))] = \#[f_i(\rho'(d))] \quad \text{for any operation } f_i$$

Proof: Suppose the representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$ has a separate pointer and is defined by

$$\rho(d) = f(d) \cup \ell(|d|).$$

We can define a representation $\rho':\mathbb{D} \to \mathcal{B}^\dagger$ without a separate pointer by

$$\rho'(d) = f'(d) \cup \ell'(|d|)$$

where $$f'(d) = \{(2n, m(n)) \mid n \in D(f(d))\}$$

and $$\ell'(|d|) = \{(2n + 1, m(n)) \mid n \in D(\ell(|d|))\}.$$

Since $$D(f'(d)) \cap D(\ell'(|d|)) = \varnothing,$$

it is clear that $|\rho(d)| = |\rho'(d)|$ for all $d \in \mathbb{D}$. Also, any access sequence to perform an operation f_i using ρ' can be mapped in an obvious way to an access sequence to perform f_i using representation ρ. ∎

Recall that no endmarker representation can achieve Kraft storage for finite $\mathbb{D}$. This is not the case for pointer representations, as the following example shows.

Example 5.26. Let $X = \{a,b\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \bigcup_{i \in \{0,1,2,3\}} X^i$. Define the function $f: X \to \mathcal{B}^*$ by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \end{aligned}$$

and the concatenation-preserving function $f': \mathbb{D} \to \mathcal{B}^+$ by

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{i-1}$$

Let the pointer $\ell: \{0,1,2,3\} \to \mathcal{B}^*$ be defined so that

$$\begin{aligned} \ell(0) &= 00 \\ \ell(1) &= 01 \\ \ell(2) &= 10 \\ \ell(3) &= 11 \end{aligned}$$

Then we define the representation $\rho: \mathbb{D} \to \mathcal{B}^*$ by

$$\rho(d) = \{\ell(|d|)\}_0 \cup \{f'(d)\}_2$$

For instance,

$$\begin{aligned} \rho(\lambda) &= 00 \\ \rho(a) &= 010 \\ \rho(b) &= 011 \\ \rho(aa) &= 1000 \\ \rho(abb) &= 11011 \end{aligned}$$

The representation ρ achieves Kraft storage, because

$$\sum_{d\in\mathbb{D}} 2^{-|\rho(d)|} = \sum_{i\in\mathcal{J}} \sum_{d\in X^i} 2^{-(i+2)}$$
$$= \sum_{i=0}^{3} 2^i \cdot 2^{-(i+2)}$$
$$= 1 . \qquad \blacksquare$$

So we know from examples 5.22 and 5.26 that a pointer representation may achieve Kraft storage for $\mathbb{D}$ infinite or finite.

Let us try to determine under what conditions a pointer representation ρ does achieve Kraft storage. The following theorem shows that the pointer ℓ must itself achieve Kraft storage in order for ρ to achieve Kraft storage.

Theorem 5.12. Let f be a total function $f:\mathbb{D} \to \mathcal{B}^\dagger$ and let $\ell:\mathcal{J} \to \mathcal{B}^\dagger$ be a representation which does not achieve Kraft storage. Then the pointer representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$, where

$$\rho(d) = \{f(d)\}_{n_1(d)} \cup \{\ell(|d|)\}_{n_2(d)}$$

does not achieve Kraft storage.

Proof: Since ℓ is a representation which does not achieve Kraft storage,

$$\sum_{i\in\mathcal{J}} |\mathcal{B}|^{-|\ell(i)|} < 1.$$

We first show that the theorem holds for a separate pointer representation $\rho':\mathbb{D} \to \mathcal{B}^\dagger$, where

$$\rho'(d) = f(d) \cup \ell(|d|).$$

Assume that the representation ρ' does attain Kraft storage. Then

$$1 = \sum_{i\in\mathcal{J}} \left(|\mathcal{B}|^{-|\ell(i)|} \sum_{d\in X^i} |\mathcal{B}|^{-|f(d)|} \right)$$

Thus, there exists $k \in \mathcal{J}$ such that

$$\sum_{d\in X^k} |\mathcal{B}|^{-|f(d)|} > 1.$$

So f is not a representation and there exist $d_1, d_2 \in X^k$ such that $f(d_1)$ and $f(d_2)$ are indistinguishable. But since $|d_1| = |d_2| = k$,

$$\rho'(d_1) = f(d_1) \cup \ell(k)$$

and

$$\rho'(d_2) = f(d_2) \cup \ell(k)$$

are indistinguishable, contradicting the fact that ρ' is a representation. Thus, ρ' cannot achieve Kraft storage if ℓ doesn't. Since

$$|\rho'(d)| = |f(d)| + |\ell(|d|)| = |\rho(d)|,$$

then

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho'(d)|} \neq 1$$

implies that

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} \neq 1.$$

So the pointer representation ρ cannot achieve Kraft storage if ℓ doesn't. ∎

Thus, the pointer ℓ achieving Kraft storage is a necessary, although certainly not sufficient, condition for the pointer representation ρ to achieve Kraft storage.

We frequently consider a pointer representation formed from a concatenation-preserving function f' and a pointer ℓ. We now show that whenever that concatenation-preserving function f' is based on a function $f: X \to \mathcal{B}^+$ which itself is a representation and attains Kraft storage, then the pointer representation ρ also achieves Kraft storage, assuming, of course, that the pointer ℓ achieves Kraft storage.

Theorem 5.13. Let $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$ and consider a representation function $f: X \to \mathcal{B}^+$ which achieves Kraft storage. Let $f': \mathbb{D} \to \mathcal{B}^+$ be a concatenation-preserving function formed from f and defined by

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)}.$$

If the representation $\ell: \mathcal{J} \to \mathcal{B}^+$ attains Kraft storage, then the pointer representation $\rho: \mathbb{D} \to \mathcal{B}^+$ also achieves Kraft storage, where

$$\rho(d) = \{f'(d)\}_{n_1(d)} \cup \{\ell(|d|)\}_{n_2(d)}.$$

Proof: Since $|\rho(d)| = |f'(d)| + |\ell(|d|)|$,

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = \sum_{i \in \mathcal{I}} \sum_{d \in X^i} |\mathcal{B}|^{-(|f'(d)| + |\ell(i)|)}$$

$$= \sum_{i \in \mathcal{I}} (|\mathcal{B}|^{-|\ell(i)|} \sum_{d \in X^i} |\mathcal{B}|^{-|f'(d)|}).$$

Since f achieves Kraft storage we can make use of Lemma 5.4, which gives us

$$\sum_{d \in \mathbb{D}} |\mathcal{B}|^{-|\rho(d)|} = \sum_{i \in \mathcal{I}} |\mathcal{B}|^{-|\ell(i)|}$$

$$= 1$$ ∎

We now want to determine the conditions, if any, under which a pointer representation can achieve Kraft access for the set Γ of table lookup questions and also achieve Kraft storage.

Theorem 5.14. Let $\mathbb{D} = \bigcup_{i \in \mathcal{I}} X^i$. If a pointer representation $\rho: \mathbb{D} \to \mathcal{B}^+$ achieves Kraft storage and also achieves Kraft access for all $\gamma_i \in \Gamma$, then $|\mathcal{B}| = 2$ and $\mathbb{D} = \{\lambda\} \cup X^n$ for some $n \in \mathbb{N}^+$.

Proof: Theorem 5.12 guarantees that if ρ achieves Kraft storage, then its pointer function $\ell: \mathcal{I} \to \mathcal{B}^+$ must also achieve Kraft storage. Since $|\mathcal{B}| \geq 2$, it must be the case that $|\mathcal{I}| \geq 2$. Thus, $\mathbb{D} \neq X^n$. Recalling theorems 4.9 and 4.10, we know that if a representation ρ achieves Kraft storage and Kraft access for all $\gamma_i \in \Gamma$, then $\mathbb{D} = X^n$ or $\mathbb{D} = \{\lambda\} \cup X^n$. Since the former is not true, the only possibility is that $\mathbb{D} = \{\lambda\} \cup X^n$. So if we are to achieve Kraft storage and access at all, then $|\mathcal{I}| = 2$ and therefore $|\mathcal{B}| = 2$. ∎

Theorem 5.14 simply says that if a pointer representation is to achieve Kraft storage and access, then $|\mathcal{B}| = 2$ and $\mathbb{D} = \{\lambda\} \cup X^n$. It does not necessarily say that it is possible to ever achieve both. The following example shows, however, that it is possible.

Example 5.27. Let $X = \{a,b,c\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \{\lambda\} \cup X^3$. We want to construct a pointer representation ρ which achieves Kraft storage and also achieves Kraft access for $\Gamma = \{\gamma_1, \gamma_2, \gamma_3\}$. To do so, we first define a function $f:X \to \mathcal{B}^+$:

$$\begin{aligned} f(a) &= 00 \\ f(b) &= 01 \\ f(c) &= 1\_ \end{aligned}$$

We then let $f':\mathbb{D} \to \mathcal{B}^+$ be the concatenation-preserving function formed from f:

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{2(i-1)}.$$

The pointer function $\ell:\mathbb{J} \to \mathcal{B}^+$ is defined by

$$\begin{aligned} \ell(0) &= 0 \\ \ell(3) &= 1 \end{aligned}$$

Then the pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$ can be defined by

$$\rho(d) = \{\ell(|d|)\}_0 \cup \{f'(d)\}_1.$$

The representation ρ achieves Kraft storage, because

$$\begin{aligned} \sum_{d \in \mathbb{D}} 2^{-|\rho(d)|} &= \sum_{d \in \{\lambda, X^n\}} 2^{-(|f'(d)| + |\ell(|d|)|)} \\ &= 2^{-1} \sum_{d \in \{\lambda, X^n\}} 2^{-|f'(d)|} \\ &= 2^{-1}(2^{-0} + 2^{-3} + 6 \cdot 2^{-4} + 12 \cdot 2^{-5} + 8 \cdot 2^{-6}) \\ &= 1. \end{aligned}$$

We can construct access trees for γ_1, γ_2, γ_3 as shown in Figure 5.11. By observation, each achieves Kraft access. ∎

This example can be generalized, giving us the following theorem.

> **Theorem 5.15.** Consider any domain of the form $\mathbb{D} = \{\lambda\} \cup X^n$, $|X| > 1$, and assume that $|\mathcal{B}| = 2$. Then there is a concatenation-preserving pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$ which achieves Kraft storage and for which it is possible to implement the table lookup questions $\Gamma = \{\gamma_i \mid 1 \le i \le n\}$ so that each γ_i achieves Kraft access.

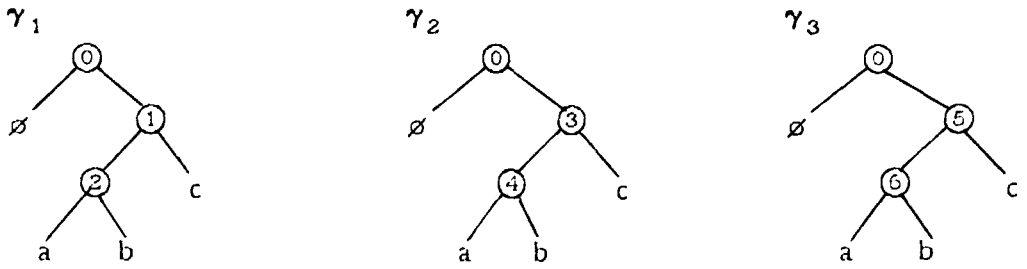


Figure 5.11. Access trees for γ_1, γ_2, γ_3 of Example 5.27.

Proof: The construction is like that in Example 5.27. We first define a function $f:X \to \mathcal{B}^+$ such that f achieves Kraft storage. It is possible to do this since there exists $n_1 \in \mathbb{N}$ such that $|X| = (|\mathcal{B}| - 1) \cdot n_1 + 1 = n_1 + 1$. A tree T for f has n_1 internal nodes, for which we can choose labels from the set $\{0, 1, \ldots, n - 1\}$. We now define the concatenation-preserving function $f':\mathbb{D} \to \mathcal{B}^+$ formed from f:

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n(i-1)}$$

The pointer function $\ell:\mathcal{J} \to \mathcal{B}^+$ is defined by

$$\ell(0) = 0$$
$$\ell(n) = 1$$

From these we define the pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$:

$$\rho(d) = \{\ell(|d|)\}_0 \cup \{f'(d)\}_1.$$

By Theorem 5.13, since f and ℓ achieve Kraft storage, so does ρ. Also, if T is the full tree corresponding to f, then the access tree for any $\gamma_i \in \Gamma$ is of the form shown in Figure 5.12. Thus, ρ achieves both Kraft access and Kraft storage. ∎

We have seen by Theorem 5.14 that only for $|\mathcal{B}| = 2$, $\mathbb{D} = \{\lambda\} \cup X^n$ can a pointer representation achieve Kraft storage and access. Let's try to see when it is at least possible to achieve Kraft access. The following example presents such a scheme, but the resulting pointer storage cost is high: $|\mathcal{J}| - 1$.

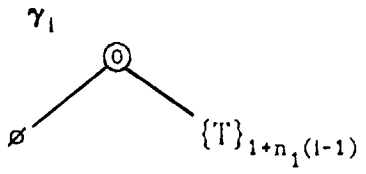


Figure 5.12. Access tree for γ_i in proof of Theorem 5.15.

Example 5.28. Let $\mathcal{B} = \{0,1\}$, $X = \{a,b,c,d\}$, and $\mathbb{D} = \bigcup_{i \in J} X^i$ for $J = \{0,3,5,6\}$. Let $f: X \to \mathcal{B}^+$ be defined by

$$\begin{aligned} f(a) &= 00 \\ f(b) &= 01 \\ f(c) &= 10 \\ f(d) &= 11 \end{aligned}$$

and define from f the concatenation-preserving representation $f': \mathbb{D} \to \mathcal{B}^+$ so that

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d)\}_{2(i-1)}$$

(a) Define a length function $\ell_1: \mathbb{D} \to \mathcal{B}^+$ by

$$\ell_1(|d|) = 1^{|d|}0^{|\Gamma|-|d|} = 1^{|d|}0^{6-|d|}$$

Then a pointer representation $\rho_1: \mathbb{D} \to \mathcal{B}^+$ can be defined by

$$\rho_1(d) = \{f'(d)\}_6 \cup \{\ell_1(|d|)\}_0.$$

Notice that $$|\rho_1(d)| = |f'(d)| + 6 = 2\cdot|d| + 6.$$

Since ℓ_1 does not achieve Kraft storage we know that ρ_1 does not either. On the other hand, we can implement each $\gamma_i \in \Gamma$ so as to achieve Kraft access. We do this by first reading the i^{th} bit of $\ell_1(|d|)$. If $m(i) = 0$, then we know $|d| < i$ and so $\gamma_i(\rho(d)) = \varnothing$. On the other hand, if $m(i) = 1$ then we know $\gamma_i(\rho(d)) \neq \varnothing$ and we look in locations $2(i-1) + 6 = 2i + 4$ and $2i + 5$ in order to determine $\gamma_i(\rho(d))$. Thus, each γ_i can be implemented by an access tree as shown in Figure 5.13.

(b) Recalling Theorem 4.14 leads us to try to find a length function ℓ_2, where $|\ell_2(|d|)| = |J| - 1$. Since we know, for instance, that $X^4 \notin \mathbb{D}$, then $\gamma_4(\rho(d_1)) = \varnothing$ if and only if $\gamma_5(\rho(d_1)) = \varnothing$. So we define the length function

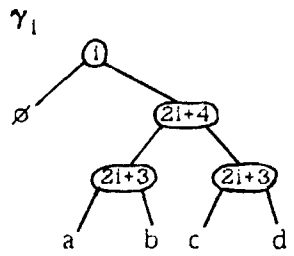


Figure 5.13. Access tree for γ_1 of Example 5.28a.

$\ell_2:\mathbb{D} \to \mathcal{B}^\dagger$ by

$$\ell_2(|d|) = \begin{cases} 000 & \text{if } |d| = 0 \\ 100 & \text{if } |d| = 3 \\ 110 & \text{if } |d| = 5 \\ 111 & \text{if } |d| = 6 \end{cases}$$

Then the pointer representation $\rho_2:\mathbb{D} \to \mathcal{B}^\dagger$ is defined by

$$\rho_2(d) = \{f'(d)\}_3 \cup \{\ell_2(|d|)\}_0.$$

Once again, ρ_2 cannot achieve Kraft storage since ℓ_2 doesn't. But we can implement each $\gamma_i \in \Gamma$ so as to achieve Kraft access, as shown in Figure 5.14. Notice that, for all $d \in \mathbb{D}$, $|\rho_2(d)| \geq 3 = |\mathcal{J}| - 1$, as required by Theorem 4.14. ∎

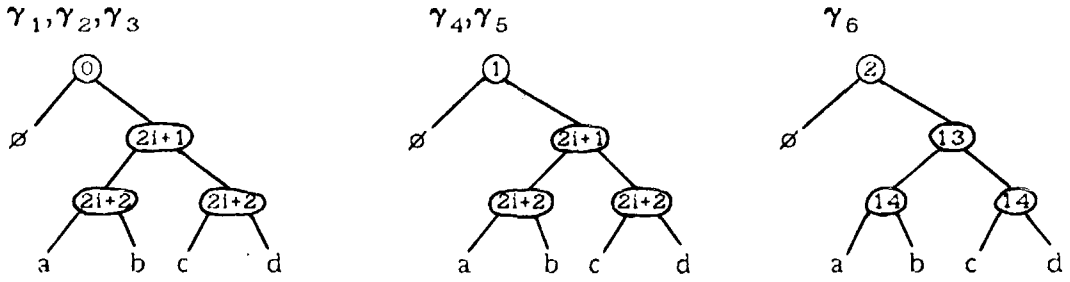


Figure 5.14. Access trees for all $\gamma_i \in \Gamma$ from Example 5.28b.

The method used in Example 5.28b can be generalized so that it is always possible, when $|\mathcal{B}| = 2$, to construct a pointer representation that achieves Kraft access.

Theorem 5.16. Let $\mathbb{D} = \bigcup_{i \in \mathcal{I}} X^i$, and let $|\mathcal{B}| = 2$. Then it is possible to construct a concatenation-preserving pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$ such that each $\gamma_i \in \Gamma$ can be implemented so as to achieve Kraft access.

Proof: Construct some representation $f:X \to \mathcal{B}^*$ such that f achieves Kraft storage. Since $|\mathcal{B}| = 2$, it is always possible to do this; f corresponds to some full tree T. Let $k = \max_{x \in X} |f(x)|$ and define the concatenation-preserving function $f':\mathbb{D} \to \mathcal{B}^+$ by

$$f'(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{k(i-1)}$$

We define the length function $\ell:\mathcal{I} \to \mathcal{B}^+$ in such a way that $|\ell(i)| = |\mathcal{I}| - 1$, for all $i \in \mathcal{I}$. First, index the elements in $\mathcal{I}$ so that $\mathcal{I} = \{i_0, i_1, i_2, \ldots\}$, where $i_j < i_{j+1}$. Then define

$$\ell(i_n) = 1^n 0^{|\mathcal{I}|-1-n}$$

The separate pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$ defined by

$$\rho(d) = f'(d) \cup \ell(|d|)$$

can be implemented so as to achieve Kraft access. For instance, $\gamma_j \in \Gamma$ can be implemented as follows. Determine the least value $i_n \in \mathcal{I}$ such that $j \leq i_n$. Then an access to cell n-1 of the pointer indicates whether or not $\gamma_j(\rho(d)) = \varnothing$:

$$m(n-1) = 0 \Rightarrow \gamma_j(\rho(d)) = \varnothing$$

and

$$m(n-1) = 1 \Rightarrow \gamma_j(\rho(d)) \neq \varnothing$$

If $\gamma_j(\rho(d)) \neq \varnothing$, then we can go to cell k(j-1) of the list function $f'(d)$. Figure 5.15 illustrates an access tree for γ_j, where the nodes of T correspond to memory cells of the list component $f'(d)$. ∎

Although the pointer representation constructed in Theorem 5.16 can achieve Kraft access, this is at a potentially very high storage cost, since for all $d \in \mathbb{D}$, $|\rho(d)| \geq |\mathcal{I}| - 1$. Unfortunately, by Theorem 4.14 we know that we cannot uniformly improve this storage. In other words, if we insist on Kraft access for all $\gamma_i \in \Gamma$, then we are stuck with $|\rho(d)| \geq |\mathcal{I}| - 1$.

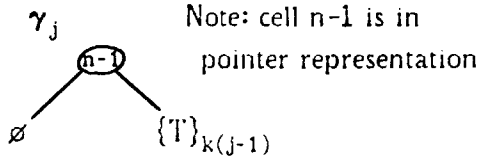


Figure 5.15. Access tree for $\gamma_j \in \Gamma$ in the proof of Theorem 5.16.

Theorem 5.15 presented a method for constructing a pointer representation so as to achieve Kraft access, but it was only for the case $|\mathcal{B}| = 2$. This leads us to wonder whether it is possible to extend the result to $|\mathcal{B}| > 2$. The following theorem shows that, for $|\mathcal{B}| > 2$, it is not possible with a pointer representation to implement each $\gamma_i \in \Gamma$ so as to achieve Kraft access, unless the pointer component is "superfluous".

Theorem 5.17. Let $|\mathcal{B}| > 2$ and consider a function $f:\mathbb{D} \to \mathcal{B}^+$. Let $\rho:\mathbb{D} \to \mathcal{B}^+$ be a pointer representation

$$\rho(d) = f(d) \cup \ell(|d|),$$

where ℓ is a representation $\ell:\mathcal{I} \to \mathcal{B}^+$. If f is not by itself a representation of $\mathbb{D}$, then ρ does not achieve Kraft access for all $\gamma_i \in \Gamma$.

Proof: Let the function f not be a representation, and assume ρ does achieve Kraft access for all $\gamma_i \in \Gamma$. Since f is not a representation, there exists $\gamma_k \in \Gamma$ such that the access tree for γ_k, T, has an internal node labelled $r \in D(\ell(|d|))$. By Theorem 4.1 and Corollary 4.2.1, since γ_k achieves Kraft access, it has $|X| + 1$ leaves with distinct labels from the set $X \cup \{\varnothing\}$ (or $|X|$ leaves if $\mathbb{D} = X^n$) and the node r has $|\mathcal{B}|$ branches. Let one of the branches from node r eventually lead to some leaf labelled $x_1 \in X$ and another branch from r eventually lead to a leaf labelled $x_2 \in X$. There is some $d_1 \in \mathbb{D}$ such that $d_1(k) = x_1$, $r \in \{[\gamma_k(\rho(d_1))]\}$, and $r \in \mathbb{D}(\ell(|d_1|))$ where $m_1(r) = b \in \mathcal{B}$ for $m_1 \supseteq \rho(d_1)$. Let

$$d_2 = \{(n,d_1(n)) \mid 1 \le n \le |d_1|, n \ne k\} \cup \{(k,x_2)\}.$$

In other words d_2 differs from d_1 only in its k^{th} element. By the definition of $\mathbb{D}$,

$d_1 \in \mathbb{D}$ imples that $d_2 \in \mathbb{D}$. Then $m_2(r) = b' \in \mathcal{B}$, where $b' \neq b$, for $m_2 \geq \rho(d_2)$. Since $|d_1| = |d_2|$, $\ell(|d_1|) = \ell(|d_2|)$ and so $m_1(r) = m_2(r)$, since $r \in \mathbb{D}(\ell(|d_1|))$. This gives a contradiction. Thus, ρ cannot achieve Kraft access for all $\gamma_i \in \Gamma$. ∎

Thus, if a pointer representation achieves Kraft access for all $\gamma_i \in \Gamma$, then the list component f was itself a representation and so we need not have stored any pointer at all. Effectively, this says that it is impossible for all $\gamma_i \in \Gamma$ to achieve Kraft access with a pointer representation in which the pointer is in fact needed to store length information. Certainly, it is not possible for a concatenation-preserving pointer representation to achieve Kraft access, since a concatenation-preserving function f′ is not a representation (except in the trivial case where $X^k \not\subseteq \mathbb{D}$ for $k \geq 2$).

Corollary 5.17.1. Let $\mathbb{D} = \bigcup_{i \in \mathcal{J}} X^i$, where $\max_{i \in \mathcal{J}} i > 1$. If $|\mathcal{B}| > 2$, no concatenation-preserving pointer representation can achieve Kraft access for all $\gamma_i \in \Gamma$.

We have seen that for a pointer representation we in general cannot hope to achieve Kraft access. On the other hand, we know that we can actually achieve Kraft storage. So let us discuss how well we can do for access costs if we insist on Kraft storage. This is the approach we take for the rest of this section, and we shall see that pointer representations can, in fact, be quite efficient in terms of access as well as storage costs.

Recall the pointer representation scheme used in Example 5.24. Since the list component f′ had fixed position fields, we could immediately (and with Kraft access) determine the answer to any $\gamma_i \in \Gamma$, as soon as we knew the answer was not $\varnothing$. So in order to answer a table lookup question γ_i, we read enough of the pointer to know whether or not $|d| \geq i$. Since the pointer function $\ell: \mathcal{J} \to \mathcal{B}^+$ was defined by $\ell(|d|) = 1^{|d|}0$, this meant we had to read i bits of the pointer for

$|d| \geq i$ and $|d| + 1$ bits of the pointer for $|d| < i$. We shall present a scheme to reduce the length $|\ell(|d|)|$, which therefore reduces the cost of accessing the pointer.

For $|\mathcal{B}| = 2$ we saw in Example 5.22 the pointer representation, where $\ell(n) = 1^n0$; this is essentially a unary representation of n followed by an endmarker. It would, of course, be desirable to somehow represent n in binary, which would descrease the storage cost but would generate the problem of detecting the end of the string; i.e., we need some way to guarantee that ℓ is a representation. Since $D(\ell) = \mathbb{N}$, we use a universal encoding method as described by Elias [7]. In this scheme we successively encode, in binary, the length of the result of the previous encoding. For instance, we could represent $|d|$ as a binary string s, which would have length $|s| \approx \log_2|d|$. If we were to use, say, a unary encoding to specify $|s|$, then we could write $\ell^1(|d|) = 0^{|s|}1s$, which gives us

$$|\ell^1(|d|)| = 2\cdot|s| + 1 \approx 2\cdot\log_2|d| + 1,$$

an improvement for large $|d|$ over our previous scheme's cost, where we had $|\ell(|d|)| = |d| + 1$. In the following example we present an encoding scheme for the case $|\mathcal{B}| = 2$.

Example 5.29. Recall the fixed position field concatenation-preserving function f' from Example 5.22. Our concern here is with finding an efficient length representation $\ell: \mathcal{I} \to \mathcal{B}^+$. Assume for simplicity that $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$. Rather than defining $\ell(|d|) = 1^{|d|}0$, as we did in Example 5.22, consider representing $|d| = n$ as a binary string as follows:

n	h(n)
0	-
1	0
2	1
3	00
4	01
5	10
6	11
7	000
8	001

More formally, we can define $h:\mathbb{N} \to \mathcal{B}^*$ by letting $h(n)$ be the binary representation of $n + 1$, with the leftmost symbol deleted. For example, to determine $h(21)$, we write 22 in binary, 10110, and then delete the leftmost symbol (always a 1): $h(21) = 0110$ (see Table 5.1). Notice that

$$|h(n)| = \lfloor \log_2(n+1) \rfloor .$$

We now define a pointer representation $\ell^1:\mathbb{N} \to \mathcal{B}^*$ by

$$\ell^1(n) = 0^{|h(n)|}.1 \cdot h(n),$$

as also shown in Table 5.1. The storage cost for the representation ℓ^1 is

$$|\ell^1(n)| = 2 \cdot |h(n)| + 1 = 2 \cdot \lfloor \log_2(n+1) \rfloor + 1.$$

We can show that the representation ℓ^1 achieves Kraft storage by noting that, for each $j \in \mathbb{N}$, $\lfloor \log_2(n+1) \rfloor = j$ for 2^j consecutive values of n:

$$\begin{aligned} \sum_{n=0}^{\infty} 2^{-|\ell^1(n)|} &= \sum_{n=0}^{\infty} 2^{-(2 \cdot \lfloor \log_2(n+1) \rfloor + 1)} \\ &= \sum_{j=0}^{\infty} 2^j \cdot 2^{-(2j+1)} \\ &= 1 . \end{aligned}$$

Thus, a worst case access cost to determine whether or not $\gamma_i(d) = \varnothing$ is just $2 \cdot \lfloor \log_2(n+1) \rfloor + 1$, an improvement over the scheme in Example 5.22 (or Example 5.24), which had a worst case of $n + 1$. In general, we can expect to do even better than this, reading only as much of the pointer as necessary. Because only two accesses of the list representation are required to read the answer $\gamma_i(d)$ for this particular example, we have the following access costs:

$$\#[\gamma_i(d)] = \begin{cases} \lceil \log_2(i+2) \rceil + 2 & \text{for } i \le 2^{\lceil \log_2(n+2) \rceil} \\ 2 \cdot \lfloor \log_2(n+1) \rfloor + 3 & \text{for } 2^{\lceil \log_2(n+2) \rceil - 1} \le i \le 2^{\lceil \log_2(n+2) \rceil} - 2 \\ \lceil \log_2(n+2) \rceil & \text{for } i \ge 2^{\lceil \log_2(n+2) \rceil} - 1. \end{cases}$$

Using the same trick over again, we can encode the length of the length of n by defining the pointer representation $\ell^2:\mathbb{N} \to \mathcal{B}^*$ by

$$\ell^2(n) = 0^{|h(|h(n)|)|}.1 \cdot h(|h(n)|) \cdot h(n),$$

giving a storage cost of

n	$h(n)$	$\ell^1(n)$	$h(\|h(n)\|)$	$\ell^2(n)$
0	-	1	-	1
1	0	010	0	0100
2	1	011	0	0101
3	00	00100	1	01100
4	01	00101	1	01101
5	10	00110	1	01110
6	11	00111	1	01111
7	000	0001000	00	00100000
8	001	0001001	00	00100001
9	010	0001010	00	00100010
10	011	0001011	00	00100011
11	100	0001100	00	00100100
12	101	0001101	00	00100101
13	110	0001110	00	00100110
14	111	0001111	00	00100111
15	0000	000010000	01	001010000
16	0001	000010001	01	001010001
17	0010	000010010	01	001010010
18	0011	000010011	01	001010011
19	0100	000010100	01	001010100
20	0101	000010101	01	001010101
21	0110	000010110	01	001010110
22	0111	000010111	01	001010111
23	1000	000011000	01	001011000
24	1001	000011001	01	001011001
25	1010	000011010	01	001011010
26	1011	000011011	01	001011011
27	1100	000011100	01	001011100
28	1101	000011101	01	001011101
29	1110	000011110	01	001011110
30	1111	000011111	01	001011111
31	00000	00000100000	10	0011000000
32	00001	00000100001	10	0011000001
33	00010	00000100010	10	0011000010

Table 5.1. Construction of pointer representations ℓ^1 and ℓ^2, for $|\mathcal{B}| = 2$.

$$|\ell^2(n)| = 2\cdot|h(|h(n)|)| + |h(n)| + 1$$
$$= 2\cdot\lfloor\log_2(\lfloor\log_2(n+1)\rfloor + 1)\rfloor + \lfloor\log_2(n+1)\rfloor + 1,$$

and, as for ℓ^1, it can also be verified that the pointer representation ℓ^2 achieves Kraft storage:

$$\sum_{n=0}^{\infty} 2^{-|\ell^2(n)|} = \sum_{n=0}^{\infty} 2^{-(2\cdot\lfloor\log_2(\lfloor\log_2(n+1)\rfloor + 1)\rfloor + \lfloor\log_2(n+1)\rfloor + 1)}$$
$$= \sum_{j=0}^{\infty} 2^{-(2\cdot\lfloor\log_2(j+1)\rfloor + 1 + j)}\cdot 2^{+j}$$
$$= \sum_{j=0}^{\infty} 2^{-(2\cdot\lfloor\log_2(j+1)\rfloor + 1)}$$
$$= \sum_{j=0}^{\infty} 2^{-|\ell^1(n)|}$$
$$= 1. \quad \blacksquare$$

The pointer representation construction procedure presented in Example 5.29 can be applied indefinitely, encoding the length of the length of the length of n, etc. It can also be extended to the case where $|\mathcal{B}| > 2$. In order to do this, we make use of a mod-$|\mathcal{B}|$ successor operation, $\oplus_{|\mathcal{B}|}$, on strings. We define $\oplus_{|\mathcal{B}|}$ so that, e.g., $\oplus_2$ corresponds intuitively to addition base 2 with the leftmost 1 deleted:

$$0 \oplus_2 1 = 1,\ 1 \oplus_2 1 = 00,\ 00 \oplus_2 1 = 01,\ \ldots,\ 11 \oplus_2 1 = 000,\ \ldots$$

For $\oplus_3$ we would obtain the sequence of strings

$$1,\ 2,\ 00,\ 01,\ 02,\ 10,\ \ldots,\ 22,\ 000,\ 001,\ \ldots$$

Definition. Consider a binary string

$$s = s_{|s|}\cdot s_{|s|-1}\cdot\ \ldots\ \cdot s_3\cdot s_2\cdot s_1 \in \mathcal{B}^*$$

and let

$$k = \min\{i \mid s_i \neq |\mathcal{B}| - 1\}.$$

(If $s_i = |\mathcal{B}| - 1$ for $1 \leq i \leq |s|$, by convention we have $k = |s| + 1$.) Then we define

$$s' = s \oplus_{|\mathcal{B}|} 1$$

by

$$s_i' = \begin{cases} 0 & \text{for } 1 \le i < k \\ s_k + 1 & \text{for } i = k,\ k \le |s| \\ s_i & \text{for } k < i \le |s| \\ 0 & \text{for } i = k,\ k = |s| + 1 \end{cases}$$

So $|s'| = |s|$ except when $s = \{|\mathcal{B}|-1\}^{|s|}$, in which case $s' = \{0\}^{|s|+1}$ and $|s'| = |s| + 1$. We can now define a function h as a $|\mathcal{B}|$-ary string representation of a natural number n.

Definition. For $|\mathcal{B}| \ge 2$, let $h_{|\mathcal{B}|}(n)$ be the encoding of $n \in \mathbb{N}$ as a $|\mathcal{B}|$-ary string, $h_{|\mathcal{B}|}: \mathbb{N} \to \mathcal{B}^*$, where

$$\begin{aligned} h_{|\mathcal{B}|}(0) &= \lambda \\ h_{|\mathcal{B}|}(1) &= 0 \\ h_{|\mathcal{B}|}(n+1) &= h_{|\mathcal{B}|}(n) \oplus_{|\mathcal{B}|} 1. \end{aligned}$$

For any string $b \in \mathcal{B}^*$, for $|\mathcal{B}| = 1$ we by convention define

$$h_1(b) \triangleq b.$$

We extend our notation and write $h^{k+1}_{|\mathcal{B}|}(n)$ to indicate k + 1 applications of $h_{|\mathcal{B}|}$:

$$h^{k+1}_{|\mathcal{B}|}(n) \triangleq h^{k}_{|\mathcal{B}|}(|h_{|\mathcal{B}|}(n)|).$$

Where the particular $|\mathcal{B}|$ we are considering is clear, we may simply write h rather than $h_{|\mathcal{B}|}$. For instance, the function h in Table 5.1 corresponds to h_2. Notice that $1 \cdot h_2(n+1) = 1 \cdot h_2(n) + 1$, where the addition is in base 2.

Example 5.30. For $|\mathcal{B}| = 3$, Table 5.2 illustrates $h_3(n)$ and $h^2_3(n)$. To see how we can use the above definitions to determine $h_3(n)$, assume we know $h_3(11) = 21$. So

$$h_3(12) = h_3(11) \oplus_3 1 = 21 \oplus_3 1.$$

Letting $s = 21 = s_2 s_1$, then $\min\{i \mid s_i \ne |\mathcal{B}|-1\} = 1$ and so

$$s_i' = \begin{cases} s_1 + 1 & \text{for } i = 1 \\ s_2 & \text{for } i = 2 \end{cases}$$

Thus,

$$s' = h_3(12) = 22.$$

Similarly, $h_3(13) = h_3(12) \oplus_3 1 = 22 \oplus_3 1,$

and for $s = 22 = s_2 s_1$, then $\min\{i \mid s_i \neq 2\} = 3 = |s| + 1$ and

$$h_3(13) = 000.$$

Using the above notation we have, e.g.,

$$h\,{}^{3}_{3}(n) = h\,{}^{2}_{3}(|h_3(n)|) = h_3(|h_3(|h_3(n)|)|)$$

Notice that $|h_3(n)| = 0$ for one value of n, $|h_3(n)| = 1$ for three values of n, $|h_3(n)| = 2$ for nine values of n, etc. ∎

In general, since $|s \oplus_{|\mathcal{B}|} 1| = |s|$ except for $s = \{|\mathcal{B}|-1\}^{|s|}$, we note that the above definitions, by design, give us the following lemma.

Lemma 5.6. For $|\mathcal{B}| \geq 1$, there are $|\mathcal{B}|^r$ values of $n \in \mathbb{N}$ such that $|h_{|\mathcal{B}|}(n)| = r$.

Lemma 5.6 immediately allows us to show the following.

Lemma 5.7. Let $n \in \mathbb{N}$. For $|\mathcal{B}| \geq 2$,

$$|h_{|\mathcal{B}|}(n)| = \lfloor \log_{|\mathcal{B}|}(|\mathcal{B}|-1)(n+1) \rfloor.$$

For $|\mathcal{B}| = 1$, $\qquad |h_1(n)| = n.$

Proof: For $|\mathcal{B}| = 1$, $|h_1(n)| = n$ by definition. So consider $|\mathcal{B}| \geq 2$. Since Lemma 5.6 tells us that there are $|\mathcal{B}|^r$ values of $n \in \mathbb{N}$ such that $|h_{|\mathcal{B}|}(n)| = r$, then we know there are $\sum_{i=0}^{r} |\mathcal{B}|^i$ values of n such that $|h_{|\mathcal{B}|}(n)| \leq r$, and

n	$h_3(n)$	$\ell^1_3(n)$	$h_3(\lvert h_3(n)\rvert)$	$\ell^2_3(n)$
0	-	2	-	2
1	0	020	0	0200
2	1	021	0	0201
3	2	022	0	0202
4	00	1200	1	02100
5	01	1201	1	02101
6	02	1202	1	02102
7	10	1210	1	02110
8	11	1211	1	02111
9	12	1212	1	02112
10	20	1220	1	02120
11	21	1221	1	02121
12	22	1222	1	02122
13	000	002000	2	022000
14	001	002001	2	022001
15	002	002002	2	022002
16	010	002010	2	022010
17	011	002011	2	022011
18	012	002012	2	022012
19	020	002020	2	022020
.	.	.	.	.
.	.	.	.	.
38	221	002221	2	022221
39	222	002222	2	022222
40	0000	0120000	00	12000000
41	0001	0120001	00	12000001
42	0002	0120002	00	12000002
43	0010	0120010	00	12000010
44	0011	0120011	00	12000011
.	.	.	.	.
.	.	.	.	.
120	2222	0122222	00	12002222
121	00000	10200000	01	120100000
122	00001	10200001	01	120100001

Table 5.2. Construction of pointer representations ℓ^1 and ℓ^2, for $|\mathcal{B}| = 3$.

$$\begin{aligned} |h_{|\mathcal{B}|}(n)| &= \min\{k \mid n \le \sum_{i=0}^{k} |\mathcal{B}|^i - 1\} \\ &= \min\{k \mid n + 1 \le \frac{|\mathcal{B}|^{k+1} - 1}{|\mathcal{B}| - 1}\} \\ &= \min\{k \mid (|\mathcal{B}| - 1)(n + 1) \le |\mathcal{B}|^{k+1} - 1\} \\ &= \lfloor \log_{|\mathcal{B}|}(|\mathcal{B}| - 1)(n + 1) \rfloor \end{aligned}$$ ∎

We now define our class of pointer representations, extending the ℓ^1 and ℓ^2 of Example 5.29. For $|\mathcal{B}| = 2$ we want:

$$\ell^1_2(n) = 0^{|h(n)|}.1 \cdot h(n)$$
$$\ell^2_2(n) = 0^{|h(|h(n)|)|}.1 \cdot h(|h(n)|) \cdot h(n)$$
$$\ell^3_2(n) = 0^{|h(|h(|h(n)|)|)|}.1 \cdot h(|h(|h(n)|)|) \cdot h(|h(n)|) \cdot h(n)$$

Notice, however, that for $|\mathcal{B}| > 2$ the first component of $\ell^i_{|\mathcal{B}|}$, $0^{|h^i(n)|}.1$, can in fact be encoded in base $|\mathcal{B}| - 1$, leaving one unused symbol to serve as the endmarker. So we can formalize the class of pointer representations as follows.

Definition. Let $|\mathcal{B}| \ge 2$. We can define a class of pointer representations ℓ^i, for $i > 0$, as follows:

$$\ell^k_{|\mathcal{B}|}(n) = \{h_{|\mathcal{B}|-1}(|h^k_{|\mathcal{B}|}(n)|) \cdot (|\mathcal{B}|-1)^1\}_0 \cup (\bigcup_{i=1}^{k} \{h^i_{|\mathcal{B}|}(n)\}_{n_i})$$

where $$n_i = 1 + |h_{|\mathcal{B}|-1}(|h^k(n)|)| + \sum_{j=i+1}^{k} |h^i(n)|$$

Therefore

$$\ell^k_{|\mathcal{B}|}(n) = h_{|\mathcal{B}|-1}(|h^k_{|\mathcal{B}|}(n)|) \cdot (|\mathcal{B}|-1) \cdot h^k_{|\mathcal{B}|}(n) \cdot h^{k-1}_{|\mathcal{B}|}(n) \cdot \ldots \cdot h^2_{|\mathcal{B}|}(n) \cdot h_{|\mathcal{B}|}(n)$$

Example 5.31. We now verify that the definition behaves as we would like for $|\mathcal{B}| = 3$, writing h to mean h_3. In particular,

$$\ell^1_3(n) = h_2(|h(n)|) \cdot 2 \cdot h(n)$$
$$\ell^2_3(n) = h_2(|h(|h(n)|)|) \cdot 2 \cdot h(|h(n)|) \cdot h(n)$$
$$\ell^3_3(n) = h_2(|h^3(n)|) \cdot 2 \cdot h(|h(|h(n)|)|) \cdot h(|h(n)|) \cdot h(n)$$

Thus, $$\ell^3_3(n) = \{h_2(|h^3(n)|) \cdot 2\}_0 \cup (\bigcup_{i=1}^{3} \{h^i(n)\}_{n_3})$$

where $n_1 = |h_2(|h_3^3(n)|)| + 1 + |h_3^2(n)| + |h_3^3(n)|$

$n_2 = |h_2(|h_3^3(n)|)| + 1 + |h_3^3(n)|$

$n_3 = |h_2(|h_3^3(n)|)| + 1$

So $\ell_3^3(n) = h_2(|h^3(n)|)\cdot 2\cdot h_3^3(n)\cdot h_3^2(n)\cdot h_3^1(n).$ ∎

The length $|\ell_{|\mathcal{B}|}^k(n)|$ should immediately be clear.

Theorem 5.18. Let $|\mathcal{B}| \geq 2$, $k \geq 1$. Then

$$|\ell_{|\mathcal{B}|}^k(n)| = |h_{|\mathcal{B}|-1}(|h_{|\mathcal{B}|}^k(n)|)| + 1 + \sum_{i=1}^{k} |h_{|\mathcal{B}|}^i(n)|.$$

While the exact numerical value of $|\log_{|\mathcal{B}|}^k(n)|$ can be obtained by substituting $\lfloor \log_{|\mathcal{B}|}((|\mathcal{B}|-1)(n+1)) \rfloor$ for $|h_{|\mathcal{B}|}(n)|$ in the expression in Theorem 5.18, we can see that we essentially have:

$$|\ell^1(n)| \approx 2\log n,$$

$$|\ell^2(n)| \approx \log n + 2\log\log n,$$

$$|\ell^3(n)| \approx \log n + \log\log n + 2\log\log\log n.$$

In any case, we can make the following statement.

Corollary 5.18.1. Let $|\mathcal{B}| \geq 2$, $k \geq 1$. Then

$$|\ell_{|\mathcal{B}|}^i(n)| = O(\log_{|\mathcal{B}|} n).$$

We can now show that each of the pointer representations $\ell_{|\mathcal{B}|}^i$ achieves Kraft storage.

Theorem 5.19. For $|\mathcal{B}| \geq 2$, $k \geq 1$, each of the pointer representations $\ell_{|\mathcal{B}|}^k$ achieves Kraft storage:

$$\sum_{n=0}^{\infty} |\mathcal{B}|^{-|\ell^k(n)|} = 1.$$

Proof: The proof is by induction on k. Once again, we write ℓ to denote $\ell_{|\mathcal{B}|}$ and h to denote $h_{|\mathcal{B}|}$.

Basis: For $k = 1$,

$$\sum_{n=0}^{\infty} |\mathcal{B}|^{-|\ell^1(n)|} = \sum_{n=0}^{\infty} |\mathcal{B}|^{-(|h_{|\mathcal{B}|-1}(|h(n)|)| + 1 + |h(n)|)} \quad \text{by Lemma 5.8}$$

$$= \sum_{r=0}^{\infty} |\mathcal{B}|^{-(|h_{|\mathcal{B}|-1}(r)| + 1 + r)} \cdot |\mathcal{B}|^{r} \quad \text{by Lemma 5.6}$$

$$= \frac{1}{|\mathcal{B}|} \sum_{r=0}^{\infty} |\mathcal{B}|^{-|h_{|\mathcal{B}|-1}(r)|}$$

$$= \frac{1}{|\mathcal{B}|} \sum_{j=0}^{\infty} |\mathcal{B}|^{-j} \cdot (|\mathcal{B}| - 1)^{j} \quad \text{by Lemma 5.6}$$

$$= 1$$

Induction step: Assume the result holds for k; i.e., assume that

$$\sum_{n=0}^{\infty} |\mathcal{B}|^{-|\ell^k(n)|} = 1.$$

Then

$$\sum_{n=0}^{\infty} |\mathcal{B}|^{-|\ell^{k+1}(n)|} = \sum_{n=0}^{\infty} |\mathcal{B}|^{-(|h_{|\mathcal{B}|-1}(|h^{k+1}(n)|)| + 1 + \sum|h^i(n)|)}$$

$$= \sum_{n=0}^{\infty} |\mathcal{B}|^{-(|h_{|\mathcal{B}|-1}(|h^k(|h(n)|)|)| + 1 + |h(n)| + \sum|h^{i+1}(n)|)}$$

$$= \sum_{j=0}^{\infty} |\mathcal{B}|^{-(|h_{|\mathcal{B}|-1}(|h^k(j)|)| + 1 + j + \sum|h^i(j)|} \cdot |\mathcal{B}|^{j}$$

$$= \sum_{j=0}^{\infty} |\mathcal{B}|^{-(|h_{|\mathcal{B}|-1}(|h^k(j)|)| + 1 + \sum|h^i(j)|)}$$

$$= \sum_{j=0}^{\infty} |\mathcal{B}|^{-|\ell^k(j)|}$$

$$= 1. \qquad \blacksquare$$

Since each of the pointer schemes ℓ^i achieves Kraft storage, it follows from Theorem 5.13 that a pointer representation which uses ℓ^i also achieves Kraft storage if the list component is storage efficient.

Corollary 5.19.1. Consider a separate concatenation-preserving pointer representation $\rho: \mathbb{D} \to \mathcal{B}^+$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)} \cup \{\ell^{k}_{|\mathcal{B}|}(|d|)\}.$$

If f achieves Kraft storage, then ρ also achieves Kraft storage.

So we have presented a pointer encoding scheme which allows us to represent lists of unbounded length and also achieve Kraft storage. Consider a fixed position field, separate, concatenation-preserving pointer representation ρ, and let us see how well one can do for access. We already know, of course, that we cannot achieve Kraft access. So suppose we want to answer some table lookup question $\gamma_i \in \Gamma$. We can do this by reading the pointer in order to determine whether or not $|d| \geq i$. If it is not, then we immediately return the answer $\varnothing$. If it is, then we go to the appropriate memory location to read the answer. So at worst we need to make

$$|\ell^{i}_{|\mathcal{B}|}(|d|)| \approx \log^{i}_{|\mathcal{B}|}|d| + \sum_{j=1}^{i} \log^{j}_{|\mathcal{B}|}|d| + k$$

accesses, where k is some constant depending on the function f, at most the size of a field n_i. We can often do even better by only reading enough of the pointer to determine if $|d| \geq i$, but, of course, for $|d| = i$ we would be forced to read $|\ell(|d|)| + k = O(\log|d|)$ cells. We shall discuss this encoding in the context of stacks in Section 6.4.

CHAPTER 6

STACKS

In Chapter 1 we discussed what we mean by a stack, a linear list for which all insertions and deletions are made at the top of the stack. Much work has been done to obtain formal specifications of the stack as a data type (see e.g., Liskov and Zilles [18], Lehman and Smyth [17]), but such a formal definition is unimportant for our purposes. Any scheme which captures our intuitive notion of a stack would suffice. It is our goal to apply some of the techniques we have thus far developed to analyze some stack implementations in terms of Kraft storage and access. We first define the basic stack operations and in the following sections we examine endmarker and poionter stack representations. Table 6.3 at the end of the chapter summarizes some of the lower bound results.

6.1 Stack Operations

While there are various operations we might wish to consider, any stack implementations will have PUSH and POP operations. These are presumably the only update operations that we shall want to perform on a stack. We also want some way to read elements in the list; we at least need to be able to read the top stack element. So we begin by formally defining these three stack operations: PUSH, POP, TOP.

Viewed in the problem domain, a PUSH operation causes a new value in X to be inserted at the top of the stack, thereby increasing the stack length by one. So a PUSH is a pure update, provided the stack can grow indefinitely. Where the memory size L is bounded, some sort of "Error" statement must be returned if an attempt is made to PUSH a value onto a stack which has no room to grow. Thus, we define a PUSH operation to consist of both a question and an update.

Because we are considering only domains of the form $\mathbb{D} = \bigcup_{i \in J} \{X^i\}$ and a PUSH operation will cause a stack to increase in size by one, it makes little sense to consider domains where $i, i+2 \in J$ but $i+1 \notin J$. So for simplicity we shall henceforth assume that $\mathbb{D} = \bigcup_{i=0}^{L} \{X^i\}$, where L may be infinite.

For the problem domains we are considering, if $b \in X^i$ and $b \in \mathbb{D}$, then $X^i \subseteq \mathbb{D}$. So any value in X can be pushed onto a stack at any time, and there are in general $|X|$ different PUSH operations. The following definition states more formally what we mean in the problem domain by a PUSH operation.

Definition. In any problem domain $\mathbb{D} = \bigcup_{i=0}^{L} \{X^i\}$, we define the class of PUSH operations

$$F_{PUSH} = \{ f_{PUSHx} \mid x \in X \},$$

where each PUSH operation f_{PUSHx} consists of a question component and an update component:

$$f_{PUSHx} = (q_{PUSHx}, u_{PUSHx}).$$

For any $d \in \mathbb{D}$,

$$q_{PUSHx}(d) = \begin{cases} \varnothing & \text{if } |d| < L \\ \text{Error} & \text{if } |d| = L \end{cases}$$

and

$$u_{PUSHx}(d) = \begin{cases} d \cup \{(|d|,x)\} & \text{if } q_{PUSHx}(d) = 0 \\ d & \text{else} \end{cases}$$

If L is infinite, then any finite stack is allowed, and we always have

$$q_{PUSHx}(d) = 0$$
$$u_{PUSHx}(d) = d \cup \{(|d|,x)\}$$

and so we can view a PUSH operation as a pure update.

Similarly, a POP operation also consists of a question and an update portion. A POP causes the top stack element to be removed; i.e., the stack length is decreased by one. If the stack length is already empty, however, then its length should not be decreased and some sort of "Error" must be returned.

Definition. For any problem domain $\mathbb{D} = \bigcup_{i=0}^{L} \{X^i\}$ and any $d \in \mathbb{D}$, we define a POP operation f_{POP} by

$$f_{POP} = (q_{POP}, u_{POP}),$$

where

$$q_{POP} = \begin{cases} \varnothing & \text{if } |d| > 0 \\ \text{Error} & \text{if } |d| = 0 \end{cases}$$

and

$$u_{POP}(d) = \{(n, d(n)) \mid 0 \leq n < |d| - 1\}.$$

Note that $u_{POP}(d) = \varnothing$ when $|d| = 0$ (as well as when $|d| = 1$). We have defined the POP operation to be a pure update when $|d| \neq 0$.

We read the stack via the top element, using the operation TOP. Since the stack state is not altered, $u_{TOP}(d) = d$ and TOP is defined as a pure question.

Definition. For any problem domain $\mathbb{D} = \bigcup_{i=0}^{L} \{X^i\}$ and any $d \in \mathbb{D}$, we define the TOP operation f_{TOP} as a pure question:

$$f_{TOP}(d) = q_{TOP}(d) = \begin{cases} d(|d|-1) & \text{if } |d| > 0 \\ \text{Error} & \text{if } |d| = 0 \end{cases}$$

We might have chosen to define a POP operation so as to return the value which it deletes from the top of the stack. Instead, we define another operation, TPOP, to serve as a combination TOP and POP operation.

Definition. For any problem domain $\mathbb{D} = \bigcup_{i=0}^{L} \{X^i\}$ and any $d \in \mathbb{D}$, we define the TPOP operation f_{TPOP} by

$$f_{TPOP} = (q_{TPOP}, u_{TPOP}),$$

where $$q_{TPOP}(d) = q_{TOP}(d)$$

and $$u_{TPOP}(d) = u_{POP}(d).$$

So TPOP returns an "Error" message precisely when $|d| = 0$. In general, we choose to discuss separately the component TOP and POP operations and only occasionally make reference to the TPOP operation.

We have defined the basic stack operations that we shall consider. Notice that a PUSH or POP operation causes the stack size to increment or decrement by at most one. It is also possible to execute the composition of a fixed sequence of operations; e.g., to push a sequence of k symbols onto the stack. We might extend this notion and consider the execution of a conditional sequence of operations in which the operation to be executed next (if any) depends on the answer sequence returned by the operations performed so far. For instance, there might be an operation to clear the stack; i.e., POP until stack is empty.

We shall in the rest of the chapter consider several stack representations and see how efficiently it is possible to perform the basic stack operations. Recall that the operation definitions we have presented describe behavior in the problem domain; for a particular representation, the operation behavior in the machine domain might or might not resemble the problem domain behavior.

6.2 The TOS Endmarker Representation

Consider using an endmarker representation to implement a stack and the PUSH, POP, TOP operations. The following example illustrates one possible such implementation.

Example 6.1. Let $X = \{a,b\}$, $\mathcal{B} = \{0,1,2\}$, and $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$. Let the function $f: X \cup \{\varnothing\} \to \mathcal{B}^*$ be defined by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \\ f(\varnothing) &= 2 = \Diamond \end{aligned}$$

Define the concatenation-preserving endmarker representation $\rho: \mathbb{D} \to \mathcal{B}^*$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{i-1} \cup \{\Diamond\}_{|d|}.$$

In this representation, one symbol from $\mathcal{B}$, namely 2, is reserved to tell us when we have reached the top of the stack. For instance,

$$\begin{aligned} \rho(\lambda) &= 2 \\ \rho(abaa) &= 01002 \\ \rho(baabba) &= 1001102 \end{aligned}$$

If we view each $d \in \mathbb{D}$ as a stack, then we might implement the POP, PUSHx, and TOP operations by first reading the stack representation from left to right until we detect the end-of-stack marker 2. For a POP operation, we then back up and put $\Diamond$ in the previous cell. Assuming L is unbounded, this corresponds to the following algorithm.

α_{POP}:

```
i ← 0
while m(i) ≠ 2 do i ← i + 1
if i = 0  then return "Error"
          else m(i - 1) ← 2
```

For PUSHx and TOP operations, we similarly read until we detect the end of stack marker 2, and we can then immediately perform the desired operation.

α_{PUSHx}: $i \leftarrow 0$
while $m(i) \neq 2$ do $i \leftarrow i + 1$
$m(i) \leftarrow f(x)$
$m(i+1) \leftarrow 2$

α_{TOP}: $i \leftarrow 0$
while $m(i) \neq 2$ do $i \leftarrow i + 1$
if $i = 0$ then return "Error"
else if $m(i-1) = 0$ then return "a"
else return "b"

These algorithms give us the following access costs when no Error conditions are encountered:

$$\#[\alpha_{POP}(\rho(d))] = |d| + 2$$
$$\#[\alpha_{PUSHx}(\rho(d))] = |d| + 2$$
$$\#[\alpha_{TOP}(\rho(d))] = |d| + 2$$

We could improve slightly the access cost for α_{TOP} by remembering the previous cell value in some location called "temp", as we make our left to right reading of the stack representation.

α_{TOP}: $i \leftarrow 0$
while $m(i) \neq 2$ do temp $\leftarrow m(i)$
$i \leftarrow i + 1$
if $i = 0$ then return "Error"
else if temp $= 0$ then return "a"
else return "b"

This modified algorithm gives us a memory cell access cost of

$$\#[\alpha_{TOP}(\rho(d))] = |d| + 1$$

Although temp can be viewed as requiring additional cells, we choose to let temp be part of our processor state, and so we do not include it in the memory access cost. ∎

A representation such as ρ in Example 6.1 is a natural one to use if we choose to implement a stack with an endmarker representation. We assume the bottom of the stack is at some fixed (known) location, and we reserve some string $\diamond \in \mathcal{B}^+$ to denote the top of the stack. We shall also require that a TOS endmarker representation have fixed position fields and that $D(\diamond) \subseteq \bigcup_{x \in X} D(f(x))$; the

reasons for these assumptions will be made clear shortly. We now make the following definition.

Definition. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any fixed position field endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{|d|+1}}$$

where $n_i \in \mathbb{N}$, for any $i \in \mathbb{N}^+$, and $f(\varnothing) = \Diamond$. If $D(\Diamond) \subseteq \bigcup_{x \in X} D(f(x))$, then we refer to ρ as a *top of stack (TOS) endmarker representation.*

Clearly the representation ρ in Example 6.1 is a TOS endmarker representation. We use the term TOS because the endmarker $\Diamond$ is always situated in the set of cells which the stack element $d(|d|+1)$ would occupy, if there were one. In other words, $\Diamond$ is in the field at the top of the stack. The representation is easiest to visualize when $n_{i+1} > n_i$ and each field consists of contiguous memory cells. Notice also that it is not necessary that each field have size one. The following example illustrates another TOS endmarker representation and shows that we need not restrict ourselves to the case where $|\mathcal{B}| \geq |X| + 1$.

Example 6.2. Let $X = \{a,b\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$. Define the function $f: X \cup \{\varnothing\} \to \mathcal{B}^*$ by

$$\begin{aligned} f(a) &= 00 \\ f(b) &= 01 \\ f(\varnothing) &= 1 = \Diamond \end{aligned}$$

Then we can define the TOS endmarker representation $\rho: \mathbb{D} \to \mathcal{B}^+$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{2(i-1)} \cup \{\Diamond\}_{2|d|}.$$

For instance, we have

$$\begin{aligned} \rho(\lambda) &= 1 \\ \rho(abaa) &= 000100001 \\ \rho(baabba) &= 0100000101001 \end{aligned}$$

Similar to what we did in Example 6.1, we can implement the POP, PUSHx, and

TOP operations by first reading $\rho(d)$ from left to right until we detect the end of stack marker $\Diamond$. However, since $|f(a)| = |f(b)| = 2$, we can locate $\Diamond$ by reading only cells 0, 2, 4, . . . , until we detect a 1. We then perform the desired operation in a straightforward way. Thus, assuming L is unbounded, we might use the following algorithms.

α_{POP}:
```
i ← 0
while m(i) ≠ 1 do i ← i + 2
if i = 0  then return "Error"
          else m(i-2) ← 1
```

α_{PUSHx}:
```
i ← 0
while m(i) ≠ 1 do i ← i + 2
m(i) ← 0
if x = a then m(i+1) ← 0
if x = b then m(i+1) ← 1
m(i+2) ← 1
```

α_{TOP}:
```
i ← 0
while m(i) ≠ 1 do i ← i + 2
if i = 0  then return "Error"
          else if m(i-1) = 0 then return "a"
                             else return "b"
```

These produce the following access costs, when no Error conditions are encountered:

$$\#[\alpha_{POP}(\rho(d))] = |d| + 2$$
$$\#[\alpha_{PUSHx}(\rho(d))] = |d| + 3$$
$$\#[\alpha_{TOP}(\rho(d))] = |d| + 2$$ ∎

In both examples 6.1 and 6.2 we found that the access costs for the stack operations POP, PUSHx, and TOP grow with $|d|$. This leads us to wonder whether it is ever possible to perform the operations with fewer accesses. We shall prove that the answer is no. In particular, whenever a TOS endmarker representation is used we show that for each $d \in \mathbb{D}$ it must be the case that

$$\#[\alpha_{POP}(\rho(d))] \geq |d| + 1$$
$$\#[\alpha_{PUSHx}(\rho(d))] \geq |d| + 2$$

$$\#[\alpha_{TOP}(\rho(d))] \geq \begin{cases} \lceil \frac{|d|-1}{2} \rceil + 2 & \text{for } |d| > 0 \\ 1 & \text{for } |d| = 0 \end{cases}$$

To aid us in proving these results we prove three lemmas. The first says that when $|d| = 0$ any algorithm for a POP, PUSHx, or TOP operation will access the n_1 field, which is also the $n_{|d|+1}$ field.

Lemma 6.1. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and let $d_0 \in \mathbb{D}$, $|d_0| = 0$. Consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any TOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{0\}_{n_{|d|+1}}.$$

Then any implementation of a POP, a PUSHx, or a TOP operation on data base $d_0 \in \mathbb{D}$ must access some cell in the $n_1 = n_{|d|+1}$ field.

Proof: If $|d_0| = 0$, then $\rho(d_0) = \{0\}_{n_1}$. Thus, if a stack operation is performed without accessing the n_1 field, then no cells in $\rho(d_0)$ were accessed at all. Even if we accessed every one of the (infinite number of) other memory cells, we would get no information concerning whether or not $|d| = 0$. Effectively, this says that we were able to perform the operation with no accesses, an impossibility. ∎

Lemma 6.2 guarantees that performing a stack operation on any $d \in \mathbb{D}$ causes the n_1 field to be accessed.

Lemma 6.2. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any TOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{0\}_{n_{|d|+1}}.$$

Then for all $d \in \mathbb{D}$ any implementation of a POP, a PUSHx, or a TOP operation must access some cell in the field n_1.

Proof: Let d_0 be some stack, $d_0 \in \mathbb{D}$. As a consequence of Lemma 6.1, the result of this lemma clearly holds for $|d_0| = 0$. So consider the case where $|d_0| > 0$, and let m be a memory state which contains the representation of d_0; $m \supseteq \rho(d_0)$. Assume there is an algorithm α_{OP} for the stack operation OP (one of POP, PUSHx, TOP) such that $\{[\alpha_{OP}(\rho(d_0))]\}$ does not contain any cells in the n_1 field. Let m_1 be a memory state which differs from m_0 only in the contents of field n_1:

$$m_1 = \{(n, m_0(n)) \mid n \notin D(\text{field } n_1)\} \cup \{\Diamond\}_{n_1}.$$

So m_1 represents the empty stack d_1, $|d_1| = 0$. Since α_{OP} does not access the n_1 field when applied to memory state m_0, it also does not access the field n_1 for the memory state m_1. Thus, α_{OP} performs the same operation in either case. Let us now look separately at the three stack operations.

(i) Consider the operation POP, and notice that

$$u_{POP}(d_1) = \text{Error}$$

whereas

$$u_{POP}(d_0) \neq \text{Error}.$$

Thus α_{POP} cannot always operate correctly without accessing the n_1 field.

(ii) Similarly, α_{TOP} cannot always give the right answer without accessing the field n_1, because

$$q_{TOP}(d_1) = \text{Error}$$

whereas

$$q_{TOP}(d_0) \neq \text{Error}.$$

(iii) α_{PUSHx} will write a $\Diamond$ in field n_2 if and only if the current memory state contains a representation of the empty stack.

Thus, for all $d \in \mathbb{D}$, an algorithm which implements a POP, a PUSHx, or a TOP operation will access field n_1. ∎

It is also necessary that the endmarker field be accessed, as the following lemma shows.

Lemma 6.3. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any TOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{|d|+1}}.$$

Then for all $d \in \mathbb{D}$ any implementation of a POP, a PUSHx, or a TOP operation must access the $n_{|d|+1}$ field.

Proof: For a PUSHx operation,

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\} \cup \{\Diamond\}_{n_{|d|+1}}$$

and $$\rho(u_{PUSHx}(d)) = \bigcup_{i=1}^{|d|} \{f(d(i))\} \cup \{f(x)\}_{n_{|d|+1}} \cup \{\Diamond\}_{n_{|d|+2}}$$

and so field $n_{|d|+1}$ must be not only accessed but rewritten.

The rest of the proof is similar to that of Lemma 6.2. Lemma 6.1 shows that this lemma holds for any $d_0 \in \mathbb{D}$ such $|d_0| = 0$. So consider the case $|d_0| > 0$, and let m_0 be a memory state such that $m_0 \supseteq \rho(d_0)$. Assume there is an algorithm α_{OP} for the stack operation OP such that performing $\alpha_{OP}(\rho(d_0))$ does not cause any cell in the $n_{|d_0|+1}$ field to be accessed. Choose $k \in \mathbb{N}$ such that the n_k and the n_{k+1} fields are not accessed (e.g., choose $k > |d_0| + 1$). Now define a memory state m_1 that differs from m_0 only in fields $n_{|d|+1}$, n_k, and n_{k+1}:

$$m_1 = \{(n, m_0(n)) \mid n \notin D(\text{field } n_k),\ n \notin D(\text{field } n_{k+1}),\ n \notin D(\text{field } n_{|d|+1})\}$$
$$\cup \{\rho(x_1)\}_{n_{|d|+1}} \cup \{\rho(x_2)\}_{n_k} \cup \{\Diamond\}_{n_{k+1}},$$

where x_1 is any element in X and $x_2 \in X$ such that $x_2 \neq d_0(|d_0|)$. Pick $d_1 \in \mathbb{D}$ such that $\rho(d_1) \subseteq m_1$. Since α_{OP} accesses neither the $n_{|d|+1}$, the n_k field, nor the n_{k+1} field, α_{OP} is not a correct algorithm for either of the stack operations POP or TOP, because no such implementation can perform correctly for both d_0 and d_1. (This same argument also would include the PUSHx operation.) Thus, any algorithm α_{OP} must access field $n_{|d|+1}$. ∎

We can now prove our lower bound results for the number of memory cell accesses

required to perform any POP or PUSHx operation using a TOS endmarker representation.

Theorem 6.1. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^\dagger$. Let $\rho: \mathbb{D} \to \mathcal{B}^\dagger$ be any TOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{|d|+1}}.$$

Then for all $d \in \mathbb{D}$ any implementation of a POP operation requires at least $|d| + 1$ memory cell accesses, and any implementation of PUSHx requires $|d| + 2$ accesses; i.e., for all $d \in \mathbb{D}$,

$$\#[\alpha_{POP}(\rho(d))] \geq |d| + 1$$

$$\#[\alpha_{PUSHx}(\rho(d))] \geq |d| + 2$$

Proof: Any implementation of a PUSHx or a POP operation using ρ will result in:

$$\rho(u_{PUSHx}(d)) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{f(x)\}_{n_{|d|+1}} \cup \{\Diamond\}_{n_{|d|+2}}$$

$$\rho(u_{POP}(d)) = \bigcup_{i=1}^{|d|-1} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{|d|}}$$

Assume there is some algorithm α_{OP}, for POP or PUSHx, for which there is some p, $1 \leq p \leq |d|$, such that no cell in n_p is in $\{[\alpha_{OP}(\rho(d_0))]\}$, for $d_0 \in \mathbb{D}$. Let m_0 be a memory state such that $m_0 \supseteq \rho(d_0)$, and define a memory state m_1 that differs from m_0 only in field n_p:

$$m_1 = \{(n, m_0(n)) \mid n \notin D(\text{field } n_p)\} \cup \{\Diamond\}_{n_p}.$$

Choose $d_1 \in \mathbb{D}$ such that $\rho(d_1) \subseteq m_1$. Since $D(\Diamond) \subseteq \bigcup_{x \in X} D(f(x))$, the endmarker $\Diamond$ is located entirely in the n_p field and so α_{OP} does not distinguish $\rho(d_1)$ from $\rho(d)$. Performing a PUSHx or a POP operation on d_1 would give:

$$\rho(u_{PUSHx}(d_1)) = \bigcup_{i=1}^{p-1} \{f(d(i))\}_{n_i} \cup \{f(x)\}_{n_p} \cup \{\Diamond\}_{n_{p+1}}$$

$$\rho(u_{POP}(d_1)) = \bigcup_{i=1}^{p-2} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{p-1}}$$

Thus, we must be able to distinguish $|d_1|$ from $|d|$ in order for a PUSHx or POP operation to necessarily be performed correctly. Since the argument holds for any

p, $1 \le p \le |d|$, we need to access at least $|d|$ cells. By Lemma 6.3, it is also necessary to detect the endmarker, leading to one additional access and a lower bound of $|d| + 1$ for both POP and PUSHx operations. Notice that for a PUSHx operation, it is, in addition, necessary to write $\Diamond$ in the $n_{|d|+2}$ field, which gives the $|d| + 2$ lower bound for the PUSHx operation. ∎

Whenever f achieves Kraft storage, then $D(\Diamond) \subseteq \bigcup_{x \in X} D(f(x))$, and so we have the following corollary.

Corollary 6.1.1. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any TOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{|d|+1}}.$$

If f achieves Kraft storage, then for all $d \in \mathbb{D}$ any implementation of a POP operation requires at least $|d| + 1$ memory cell accesses, and any implementation of PUSHx requires $|d| + 2$ accesses; i.e.,

$$\#[\Omega_{POP}(\rho(d))] \ge |d| + 1$$

$$\#[\Omega_{PUSHx}(\rho(d))] \ge |d| + 2$$

We have chosen to require that a TOS endmarker representation have fixed position fields and that $D(\Diamond) \subseteq \bigcup_{x \in X} D(f(x))$, because these seem to be natural requirements that are met in most implementations. As Example 6.3 illustrates, however, if we were to eliminate the condition that the fields be in fixed positions, then we might sometimes be able to achieve lower access costs than were specified by Theorem 6.1.

Example 6.3. Let $X = \{a,b\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$. Consider the storage optimal function $f: X \cup \{\varnothing\} \to \mathcal{B}^*$ defined by

$$f(a) = 0$$
$$f(b) = 10$$
$$f(\varnothing) = 11.$$

Construct from f the concatenation-preserving representation $\rho:\mathbb{D} \to \mathcal{B}^*$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{11\}_{n(d)}$$

where

$$n_i(d) = \sum_{j=1}^{i-1} |f(d(j))|$$

and

$$n(d) = \sum_{j=1}^{|d|} |f(d(j))|.$$

Then we have, for instance:

$$\rho(aaaaaa) = 00000011$$
$$\rho(bbbbbb) = 10101010101011$$
$$\rho(aabaab) = 0010001011$$

Notice that the leftmost occurrence of 11 indicates the end of the stack. It is not necessary, however, to read every element in the stack representation. For instance, when $m(i) = 0$ and $m(i+2) = 0$, then there is no need to read $m(i+1)$. Thus, we could implement POP and PUSH as follows.

α_{POP}:
```
        i ← 1
loop:   while m(i) ≠ 1 do i ← i + 2
        if m(i-1) = 0 then  i ← i + 1
                            goto loop
        if i = 1  then return "Error"
                  else m(i-2) ← 1
```

α_{PUSHa}:
```
        i ← 1
loop:   while m(i) ≠ 1 do i ← i + 2
        if m(i-1) = 0 then  i ← i + 1
                            goto loop
        m(i-1) ← 0
        m(i+1) ← 1
```

α_{PUSHb}:
```
        i ← 1
loop:   while m(i) ≠ 1 do i ← i + 2
        if m(i-1) = 0 then  i ← i + 1
                            goto loop
        m(i) ← 0
        m(i+1) ← 1
        m(i+2) ← 1
```

Using these algorithms to perform a POP or a PUSHx operation on $\rho(a^n)$ or on $\rho(b^n)$, we only make $\frac{|d|}{2} + k_1$ accesses, for some constant $k_1 \in \mathbb{N}$. So ρ is a concatenation-preserving endmarker representation for which it is not always necessary to make $|d|$ accesses. Note, however, that for $d = \{ab\}^n$ these algorithms lead us to access every cell in $D(\rho(\{ab\}^n))$, a total of $\frac{3}{2} \cdot |d| + k_2$ accesses. ∎

Although in the above example we were sometimes able to perform a POP or a PUSHx operation in only $\frac{|d|}{2}$ accesses, we at other times were forced to make $\frac{3|d|}{2}$ accesses. Thus, it seems likely that there would still be an average cost of $|d|$ accesses, even though the worst case cost has been improved. If we were to eliminate the requirement that $D(\emptyset) \subseteq \bigcup_{x \in X} D(f(x))$, then we would lose storage optimality but would be able to achieve lower access costs, as Example 6.4 shows.

Example 6.4. Let $X = \{a,b\}$, $\mathcal{B} = \{0,1,2\}$, and define the non storage optimal function $f: X \cup \{\varnothing\} \to \mathcal{B}^*$ by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \\ f(\varnothing) &= 22 \end{aligned}$$

Let $\rho: \mathbb{D} \to \mathcal{B}^*$ be the concatenation-preserving endmarker representation, with fixed position fields, defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{i-1} \cup \{22\}_{|d|}$$

For instance,

$$\begin{aligned} \rho(abaab) &= 0100122 \\ \rho(bbab) &= 110122 \\ \rho(a) &= 022 \end{aligned}$$

Possible algorithms to implement POP and PUSHx operations are as follows:

α_{POP}:

```
i ← 1
while m(i) ≠ 2 do i ← i + 2
if m(i-1) ≠ 2 then m(i-1) ← 2
              else if i = 1 then return "Error"
                            else m(i-2) ← 2
```

α_{PUSHx}: $i \leftarrow 1$
while $m(i) \neq 2$ do $i \leftarrow i + 2$
if $m(i-1) \neq 2$ then $m(i) \leftarrow f(x)$
$m(i+2) \leftarrow 2$
else $m(i-1) \leftarrow f(x)$
$m(i+1) \leftarrow 2$

For all $d \in \mathbb{D}$, these algorithms have the following access costs:

$$\#[\alpha_{POP}(\rho(d))] = \frac{|d|}{2} + k_1$$

$$\#[\alpha_{PUSHx}(\rho(d))] = \frac{|d|}{2} + k_2,$$

for $k_1, k_2 \in \mathbb{N}$. ∎

Theorem 6.1 made no mention of the TOP operation; in fact, the $|d| + 1$ result does not necessarily hold for every $d \in \mathbb{D}$. We can see this by reconsidering Example 6.1, which we do in the following example.

Example 6.5. Recall the representation ρ from Example 6.1. We presented there an algorithm α_{TOP} which required $|d| + 1$ accesses, for all $d \in \mathbb{D}$. We now show that we can sometimes do better than $|d| + 1$. For instance, consider

$$\rho(\text{abbaaba}) = 01100102.$$

From Theorem 6.1, we know that any algorithms for α_{POP} and α_{PUSHx} will access at least $|d| + 1$ memory cells, for all $d \in \mathbb{D}$. Let us construct an algorithm for α_{TOP}. Suppose our algorithm first accesses cell 7. Since $m(7) = 2$, cell 7 must contain the endmarker, if cell 7 is part of $\rho(d)$. By reading $m(6)$, we know that $q_{TOP} = a$ if $7 \in D(\rho(d))$. Of course, if $|d| < 7$ then it is possible that $q_{TOP} = b$. In order to verify that $q_{TOP} = a$ we need only access cells $m(0)$, $m(2)$, $m(3)$, $m(5)$, $m(6)$. In particular, we don't need to access $m(1)$ or $m(4)$, because we already know that $m(0) = m(3) = 0$. So upon locating the occurrence of the endmarker in cell 7, we conjecture that $q_{TOP} = a$. If $m(1) = 2$ or $m(4) = 2$ then we still have $q_{TOP} = a$. Thus, we have an example where it is possible to sometimes determine q_{TOP} in fewer than $|d| + 1$ accesses. Notice, however, that an algorithm such as we presented here would for some d require more than $|d| + 1$ accesses; in particular,

if the $m(7)$ we originally accessed were not in our representation. ∎

In determining $q_{TOP}(d)$, the trick used in Example 6.5 could allow us to access, for some $d \in \mathbb{D}$, as few as $\lceil \frac{|d| - 1}{2} \rceil + 2$ memory cells. In other words, we would always access the endmarker field and the field corresponding to the top stack element. At best we would only have to access half of the remaining $|d| - 1$ cells. The following theorem shows that it is never possible to do better.

Theorem 6.2. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any TOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Diamond\}_{n_{|d|+1}}.$$

Then for all $d \in \mathbb{D}$ such that $|d| \geq 1$ and for any implementation, α_{TOP}, of a TOP operation:

$$\#[\alpha_{TOP}(\rho(d))] \geq \lceil \frac{|d| - 1}{2} \rceil + 2.$$

Proof: By Lemma 6.3, we know that the $n_{|d|+1}$ field must always be accessed. Also, it is necessary to access the $n_{|d|}$ field, since this is the value we want to determine. So the result clearly holds for $|d| = 1$ and, by also using Lemma 6.2, for $|d| = 2$. Consider the case where $|d| > 2$. We know that we must access the fields $n_{|d|+1}$ and $n_{|d|}$. Now assume we have an algorithm for f_{TOP}, α_{TOP}, that for some $d_0 \in \mathbb{D}$ returns the value $q_{TOP}(d_0) = x_1$ for some $x_1 \in X$ and for which there exists $k \in \mathbb{N}$, $1 \leq k < |d| - 1$, such that α_{TOP} accesses neither the n_k nor the n_{k+1} field. Let $m_0 \supseteq \rho(d_0)$. Define a new memory state m_1 such that m_1 differs from m_0 only in the n_k field, which contains $f(x_2)$ (for $x_2 \neq x_1$), and in the n_{k+1} field, which contains $\Diamond$. Then $m_1 \supseteq \rho(d_1)$, where

$$d_1(i) = \begin{cases} d(i) & \text{for } i < k \\ x_2 & \text{for } i = k \end{cases}$$

Then, using the algorithm α_{TOP}, we must get $\alpha_{TOP}(\rho(d_1)) = x_1$. But we know that $f_{TOP}(d_1) = x_2$. This results in a contradiction, which means that for any

valid algorithm α_{TOP} it is never possible to not access two consecutive fields n_k and n_{k+1}, for $1 \leq k < |d| - 1$. Since by Lemma 6.2 we must always access the n_1 field, this says that we must make at least $\lceil \frac{|d| - 1}{2} \rceil + 2$ accesses. ∎

From Theorem 5.10 we immediately know that a TOS endmarker representation achieves Kraft storage when the function f does.

Theorem 6.2. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider the function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^+$. If the function f achieves Kraft storage, then the TOS endmarker representation $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$, defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{0\}_{n_{|d|+1}},$$

also achieves Kraft storage.

Before we conclude this section, let us say something about finite memories, $L < \omega$. In our definition of a TOS endmarker representation we, for simplicity, considered infinite domains and assumed that we would never run out of memory space. Allowing L to be finite would not have changed our results, except perhaps when $|\rho(d)| \approx L$, although our algorithms would, of course, have to be modified. Also, recall from Section 5.3 that an endmarker representation cannot achieve Kraft storage for finite L. If we had wanted to allow finite L we perhaps would have chosen to extend the definition of a TOS endmarker representation as in the following example.

Example 6.6. Recall Example 6.1, where $X = \{a,b\}$, $\mathcal{B} = \{0,1,2\}$, and the function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^*$ is defined by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \\ f(\varnothing) &= 2. \end{aligned}$$

Assume, however, that $L = 4$ and that $\mathbb{D} = \bigcup_{i=0}^{4} X^i$. We could define a representation $\rho: \mathbb{D} \rightarrow \mathcal{B}^*$ by

$$\rho(d) = \begin{cases} \bigcup_{i=1}^{|d|} \{f(d(i))\}_{i-1} \cup \{\diamond\}_{|d|} & \text{for } |d| \leq 3 \\ \bigcup_{i=1}^{|d|} \{f(d(i))\}_{i-1} & \text{for } |d| = 4 \end{cases}$$

Then

$$\begin{aligned} \rho(\lambda) &= 2\_\_\_ \\ \rho(\text{abb}) &= 0112 \\ \rho(\text{babb}) &= 1011 \end{aligned}$$

Notice that, using this definition, every possible memory state is a representation of some stack and ρ achieves Kraft storage. The stack operations can be implemented essentially as they were in Example 6.1, but we have to watch for $|d| = L$.

α_{POP}:
```
i ← 0
while m(i) ≠ 2 do if i = L - 1 then m(i) ← 2
                                    return
                               else i ← i + 1
if i = 0 then return "Error"
m(i-1) ← 2
```

α_{PUSHx}:
```
i ← 0
while m(i) ≠ 2 do if i = L - 1 then return "Error"
                               else i ← i + 1
m(i) ← f(x)
if i ≠ L - 1 then m(i+1) ← 2
```

α_{TOP}:
```
i ← 0
while m(i) ≠ 2 do if i = L - 1 then temp ← m(i)
                                    goto decode
                               else temp ← m(i)
                                    i ← i + 1
decode: if i = 0 then return "Error"
        else if temp = 0 then return "a"
                         else return "b"
```

These algorithms give the following access costs:

$$\#[\alpha_{POP}(\rho(d))] = \begin{cases} 1 & \text{if } |d| = 0 \\ |d| + 2 & \text{if } 0 < |d| < L \\ |d| & \text{if } |d| = L \end{cases}$$

$$\#[\alpha_{PUSHx}(\rho(d))] = \begin{cases} |d| + 2 & \text{if } |d| + 1 < L \\ |d| + 1 & \text{if } |d| + 1 = L \\ |d| & \text{if } |d| = L \end{cases}$$

$$= \min\{|d| + 2, L\}$$

$$\#[\alpha_{TOP}(\rho(d))] = \begin{cases} |d| + 1 & \text{if } |d| < L \\ |d| & \text{if } |d| = L \end{cases}$$

$$= \min\{|d| + 1, L\}$$ ∎

Thus, we certainly could have considered finite memory spaces, but the extra complication in our algorithms would not have increased our understanding of TOS endmarker representations. Similarly, in the next section we always make the assumption that L is infinite. In Section 6.4, where we discuss pointer representations for stacks, we shall consider both finite and infinite L.

In this section we have examined perhaps the most obvious stack endmarker representation scheme, the TOS endmarker representation. We know as a consequence of Theorem 6.2 that it is possible for such a representation to achieve Kraft storage, but we have also shown that any implementation will result in expensive access costs for every $d \in \mathbb{D}$. In particular,

$$\#[\alpha_{POP}(\rho(d))] \geq |d| + 1$$
$$\#[\alpha_{PUSHx}(\rho(d))] \geq |d| + 2$$
$$\#[\alpha_{TOP}(\rho(d))] \geq \lceil \frac{|d| - 1}{2} \rceil + 2,$$

for any algorithms α_{POP}, α_{PUSHx}, α_{TOP} implementing the stack operations POP, PUSHx, and TOP. This leads us to wonder whether some other type of endmarker representation could result in cheaper access costs. The POP and PUSHx operations involve updating the memory contents, but the TOP operation is just a question. Suppose we were to keep the top of the stack at some fixed location. Such a representation scheme is discussed in the next section.

6.3 The BOS Endmarker Representation

Consider an endmarker representation of a stack in which the top of the stack is always at a fixed (known) location and the bottom of the stack is allowed to vary. In this case, the endmarker denotes the bottom of the stack. The following example illustrates one possible such implementation.

Example 6.7. Consider the function f from Example 6.1, where we have $X = \{a,b\}$, $\mathcal{B} = \{0,1,2\}$, $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, and where we define the function $X \cup \{\varnothing\} \rightarrow \mathcal{B}^*$ by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \\ f(\varnothing) &= 2 = \Diamond \end{aligned}$$

Define the concatenation-preserving endmarker representation $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)} \cup \{\Diamond\}_{n(d)},$$

where $$n_i(d) = 2(|d| - i)$$

and $$n(d) = 2|d|$$

In this representation, the endmarker indicates when we have reached the bottom of the stack. Reading the memory contents "from left to right" corresponds to reading the elements in the stack from the top down. For instance,

$$\begin{aligned} \rho(\lambda) &= 2 \\ \rho(abaa) &= 00102 \\ \rho(baabba) &= 0110012 \end{aligned}$$

It is certainly easy to perform a TOP operation, since we need only read $m(0)$.

α_{TOP}: if $m(0) = 2$ then return "Error"
else if $m(0) = 0$ then return "a"
else return "b"

On the other hand, consider performing a PUSHb operation on d = ababa:

$$\rho(d) = \rho(ababa) = 010102$$

$$\rho(u_{PUSHb}(d)) = \rho(ababab) = 1010102$$

Notice that it will certainly be necessary to access $|d| + 2$ cells, since this many cells

are actually rewritten. Intuitively, we want to set $m(0) \leftarrow 1$ and to shift the contents of each cell in $\rho(d)$ right by one. Recalling the notation introduced in Chapter 3, one implementation scheme would have the access sequence

$$\underline{0}, \underline{1}, \underline{2}, \ldots \underline{|d| - 1}, \underline{|d|}, \underline{|d| + 1}.$$

One possible algorithm is the following.

α_{PUSHx}:
```
i ← 0
temp1 ← f(x)
while m(i) ≠ 2 do  temp1 ≒ m(i)
                   i ← i + 1
m(i) ← temp1
m(i+1) ← 2
```

Notice that we have made use of the additional register temp1, as we did in Example 6.1. Recall also that in Chapter 3 we defined a single access to consist of reading and then possibly rewriting a cell. Thus, we have written

$$\text{temp1} \leftrightarrows m(i)$$

to indicate a single access to $m(i)$, where the old contents of $m(i)$ is stored in temp1 and the old contents of temp1 is stored in $m(i)$. We refer to this as an exchange, and might have written it out using a second temporary location, temp2:

```
temp2 ← m(i)
m(i) ← temp1
temp1 ← temp2
```

Now consider performing a POP operation on d = ababab:

$$\rho(d) = \rho(\text{ababab}) = 1010102$$
$$\rho(u_{POP}(d)) = \rho(\text{ababa}) = 010102$$

As for PUSHx, a POP operation will have to rewrite $|d|$ cells and so at least $|d|$ accesses will be required. In this case, we intuitively want to shift the contents of all of the cells in $\rho(d)$ left by one. One scheme for doing this would have the access sequence

$$0, 1, 2, \ldots, |d|\text{-}1, |d|, \underline{|d|\text{-}1}, \underline{|d|\text{-}2}, \ldots, \underline{2}, \underline{1}, \underline{0}$$

and could be implemented using the exchange operation described above.

α_{POP}: if $m(i) = 0$ return "Error"
$i \leftarrow 1$
while $m(i) \neq 2$ do $i \leftarrow i + 1$
temp1 $\leftarrow 2$
while $i > 0$ do $i \leftarrow i - 1$
$m(i) \leftarrow$ temp1 ∎

We refer to a representation such as ρ in Example 6.7 as a BOS endmarker representation, because the endmarker $\Diamond$ is always situated in the field following that field which contains the bottom stack element; i.e., in the set of cells which the bottom stack element would occupy if the stack had another element in it.

Definition. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^+$. Let $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$ be any endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{\Diamond\}_{n_{|d|+1}}$$

where $n_i \in \mathbb{N}$, for any $i \in \mathbb{N}^+$, and $f(\varnothing) = \Diamond$. If $D(\Diamond) \subseteq \bigcup_{x \in X} D(f(x))$, then we refer to ρ as a *bottom of stack (BOS) endmarker representation.*

The definition of a BOS endmarker representation is basically the same as that of a TOS endmarker representation, except that $d(i)$ is located in field $n_{|d|+1-i}$ rather than in field n_i. In other words the order of the representations of the stack elements is reversed. The representation is easiest to visualize when $n_{i+1} > n_i$ and each field consists of contiguous memory cells, but no such requirements are imposed by the definition.

The BOS endmarker representation was motivated by an attempt to decrease the access cost for performing a top operation. As we shall see, however, we have not altered the access cost for PUSHx and we have actually worsened, for all $d \in \mathbb{D}$, the lower bound access cost for POP:

$$\#[\alpha_{TOP}(\rho(d))] \geq 1$$
$$\#[\alpha_{PUSHx}(\rho(d))] \geq |d| + 2$$
$$\#[\alpha_{POP}(\rho(d))] \geq \left\lceil \frac{3|d| + 1}{2} \right\rceil = \left\lfloor \frac{3|d|}{2} \right\rfloor + 1.$$

Theorem 6.3. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^\dagger$. Let $\rho: \mathbb{D} \rightarrow \mathcal{B}^\dagger$ be any BOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{\Diamond\}_{n_{|d|+1}}.$$

Then for all $d \in \mathbb{D}$ any implementation of a PUSHx operation requires at least $|d| + 2$ memory cell accesses; i.e., for all $d \in \mathbb{D}$,

$$\#[\alpha_{PUSHx}(\rho(d))] \geq |d| + 2.$$

Proof: Assume there exists some algorithm α_{PUSHx} for performing a PUSHx operation and some $d_0 \in \mathbb{D}$ such that $\#[\alpha_{PUSHx}(\rho(d_0))] < |d| + 2$. By the definition of a PUSHx operation we know that

$$\rho(d_0) = \bigcup_{i=1}^{|d|} \{f(d_0(i))\}_{n_{|d|+1-i}} \cup \{\Diamond\}_{n_{|d|+1}}$$

$$\rho(u_{PUSHx}(d_0)) = \bigcup_{i=1}^{|d|} \{f(d_0(i))\}_{n_{|d|+2-i}} \cup \{f(x)\}_{n_1} \cup \{\Diamond\}_{n_{|d|+2}}.$$

Certainly the values in fields $n_{|d|+1}$ and $n_{|d|+2}$ must be accessed. Assume that there is some p, $1 \leq p \leq |d|$, such that $\alpha_{PUSHx}(\rho(d_0))$ does not access field n_p. Let m_0 be a memory state such that $m_0 \geq \rho(d_0)$, and as in the proof of Lemma 6.3, let m_1 be a memory state which is identical to m_0 except in the n_p field, where $\Diamond$ is stored. If $m_1 \geq \rho(d_1)$, then the algorithm α_{PUSHx} does not distinguish d_0 and d_1 and thus α_{PUSHx} does not correctly perform a PUSHx operation on d_1, a contradiction. So any algorithm α_{PUSHx} must always access the $|d|$ fields $n_1, n_2, \ldots, n_{|d|}$, well as the fields $n_{|d|+1}$ and $n_{|d|+2}$. ∎

As a consequence of this theorem, we know that the algorithm α_{PUSHx} in Example 6.7 is optimal; in fact, we know that for no $d \in \mathbb{D}$ is it possible to make fewer than $|d| + 2$ accesses.

Let us now consider the construction of an algorithm for the POP operation. Using the scheme presented in Example 6.7, we could read the n_i fields essentially from left to right until we reach the bottom-of-stack endmarker, and then shift the

elements in the representation "left one field". This corresponds to a field access sequence

$$1, 2, \ldots, |d|-1, |d|, |d|+1, \underline{|d|}, \underline{|d|-1}, \ldots, \underline{2}, \underline{1}.$$

If we choose not to read all the way to the endmarker and then backtrack, we could use an algorithm with a field access sequence

$$1, 2, \underline{1}, 3, \underline{2}, 4, \underline{3}, \ldots, |d|, \underline{|d|-1}, |d|+1, \underline{|d|}.$$

Either of these algorithms would, however, require making a total of $2\cdot|d| + 1$ accesses, and we shall show that it is possible to (always) do better. In order to motivate the lower bound we shall obtain for the POP operation, we indicate how the algorithm α_{POP} in Example 6.7 could be improved.

Example 6.8. Recall the representation ρ from Example 6.7 and consider performing a POP operation on d_0 = abaa. We know that

$$\rho(d_0) = 00102$$

and

$$\rho(u_{POP}(\rho(d_0)) = 0102.$$

Recall that our definition of access allows us to read and then, if we choose, rewrite a cell. So suppose we first access cell 1. Since $m(1) = 0$, we put a 0 into cell 0, checking, of course, that cell 0 is not the end of the stack. We then read cell 3. Since $m(3) = 0 \neq 2$, we go back to cell 2, which we now read. Since $m(2) \neq 0$, we write a 0 into cell 2. We already know that $m(1) \neq 1$, and so we set $m(1) \leftarrow m(2)$. At this point we have (correctly) rewritten $m(0)$, $m(1)$, $m(2)$. We now read cell 5. For the case we are considering, $m(5)$ is not included in $\rho(d_0)$, so cell 5 might or might not contain the endmarker 2. In either case, we back up and read cell 4, at which time we find that $m(4) = 2$. Having already read cells 0, 1, 2, 3, we now know that cell 4 contains the BOS endmarker. So we set $m(3) \leftarrow 2$ and are done. Using this procedure we have the memory cell access sequence

$$1, \underline{0}, 3, \underline{2}, \underline{1}, 5, \underline{4}, \underline{3}, 7, \underline{6}, \underline{5}, 9, \underline{8}, \underline{7}, 11, \underline{10}, \underline{9}, \ldots$$

We might write the algorithm out as follows, making use of two temporary locations, temp1 and temp2.

```
α_POP:   temp1 ← m(1)
         if m(0) = 2  then return "Error"
                      else  m(0) ← temp1
                            if temp1 = 2 then return
         i ← 3
         while m(i) ≠ 2 do  temp1 ← m(i)
                            temp2 ← m(i-1)
                            m(i-2) ← temp2
                            if temp2 = 2  then return
                                          else  m(i-1) ← temp1
                                                i ← i + 2
         temp2 ← m(i-1)
         m(i-2) ← temp2
         if temp2 = 2  then return
                       else m(i-1) ← 2
```

This algorithm results in an access cost of

$$\#[\alpha_{POP}(\rho(d))] = \begin{cases} 3\cdot\frac{|d|}{2} + 2 & \text{for } |d| \text{ even} \\ 3\cdot\frac{|d|-1}{2} + 2 & \text{for } |d| \text{ odd.} \end{cases}$$

We shall shortly prove that the algorithm is, in fact, optimal. ∎

In order to derive a lower bound access cost result for performing a POP operation we begin by proving two lemmas. Recall that our definition of access allows us to read, although certainly not rewrite, a memory cell which is being used by another user.

Lemma 6.4. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any BOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{0\}_{n_{|d|+1}}.$$

Then a cell in field n_i, $i \geq 1$, cannot be rewritten unless each of the fields $n_1, \ldots, n_{i-1}$ has been accessed.

Proof: A cell in field n_i cannot be rewritten unless it is known that $|d| \geq i - 1$; i.e., we are not allowed to rewrite field n_i if it is in some other user's memory space. Thus, in order to rewrite a cell in field n_i, it must be the case that no n_j, for $1 \leq j < i$, contains the endmarker $\diamond$. There is no way to guarantee this without accessing each of the fields $n_1, n_2, \ldots, n_{i-1}$. ∎

The following lemma essentially tells us that field n_i cannot be rewritten until field n_{i+1} has been accessed.

Lemma 6.5. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^+$. Let $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$ be any BOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{\diamond\}_{n_{|d|+1}}.$$

Consider any algorithm, α_{POP}, for the operation POP. Then there must be an access to field n_{i+1} made previous to the last rewrite of field n_i, for $1 \leq i \leq |d|$.

Proof: Recalling the definitions of the BOS endmarker and the POP operation,

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{\diamond\}_{n_{|d|+1}}$$

and

$$\rho(u_{POP}(d)) = \bigcup_{i=1}^{|d|-1} \{f(d(i))\}_{n_{|d|-i}} \cup \{\diamond\}_{n_{|d|}}.$$

So $f(d(i))$ gets moved from field $n_{|d|+1-i}$ to field $n_{|d|-i}$. Since we can determine the contents of field $n_{|d|+1-i}$ only by making at least one access to that field, field $n_{|d|+1-i}$ must be read before its value can be put into field $n_{|d|-i}$. ∎

For any algorithm α_{POP} we can consider its corresponding field access sequence. We prove our lower bound result by lower bounding the size of a sequence which meets the conditions presented in lemmas 6.4 and 6.5. We first make the following definition.

Definition. For $k, i \in \mathbb{N}^+$, define a set $S_{k,i}$ as follows:

$$S_{k,i} \triangleq \{k, k+1, \ldots, k+i-1, k+i, \underline{k}, \underline{k+1}, \ldots, \underline{k+i-1}\}.$$

We say that a sequence is an *s(k,i)-sequence* if it contains each of the terms $\underline{k}, \underline{k+1}, \ldots, \underline{k+i-1}, k+i$, if each term in the sequence is in $S_{k,i}$, and if the following conditions are satisfied:

(i) For all r, $k < r < k + i$, the last occurrence of $\underline{r}$ is preceded by $r + 1$ or $\underline{r+1}$.

(ii) For all r, $k < r < k + i$, the last occurrence of $\underline{r}$ is preceded by j or $\underline{j}$, for every element $j \in \{k, k+1, \ldots, r-2, r-1\}$.

We define $\sigma(k,i)$ to be an $s(k,i)$-sequence of minimal length, so that

$$|\sigma(k,i)| \triangleq \min_{s(k,i)} |s(k,i)|.$$

Since $|\sigma(0,|d|)|$ is minimal over all sequences $s(0,|d|)$, $\sigma(0,|d|)$ corresponds to an optimal access order for performing a POP operation.

Lemma 6.6. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^+$. Let $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$ be any BOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{0\}_{n_{|d|+1}}.$$

Then for any algorithm, α_{POP}, which implements the operation POP, and for all $d \in \mathbb{D}$:

$$\#[\alpha_{POP}(\rho(d))] \geq |\sigma(0,|d|)|.$$

Proof: Recalling the definition of a field access sequence, the proof follows directly from lemmas 6.4 and 6.5 and from the definition of $\sigma(0,|d|)$. ∎

Now that we have established the correspondence between a sequence $\sigma(0,|d|)$ and $\#[\alpha_{POP}(\rho(d))]$, we have the notation with which we prove our lower bound result. We prove this as a consequence of three lemmas.

Lemma 6.7. For $k \in \mathbb{N}$, $k \geq 0$ (a) $|\sigma(k,0)| = 1$

(b) $|\sigma(k,1)| = 2$

Proof: (a) We want the minimal length of a sequence $\sigma(k,0)$ containing k and satisfying conditions (i) and (ii) in the definition of σ; k is such a sequence and therefore $|\sigma(k,0)| = 1$.

(b) We want the minimal length of a sequence $\sigma(k,1)$ containing $\underline{k}$, k+1, and satisfying (i) and (ii); k+1, $\underline{k}$ is such a sequence and clearly must be minimal. Thus, $|\sigma(k,1)| = 2$. ∎

Lemma 6.8. For $i \in \mathbb{N}$, $i \geq 0$, $|\sigma(0,i+2)| = 3 + |\sigma(0,i)|$.

Proof: $\sigma(0,i+2)$ is a minimal length sequence containing $\underline{0}$, $\underline{1}$, . . . , $\underline{i}$, $\underline{i+1}$, i+2, and $\sigma(2,i+2)$ is a minimal length sequence containing $\underline{2}$, $\underline{3}$, . . . , $\underline{i}$, $\underline{i+1}$, i+2 (both satisfying the above conditions (i) and (ii)).

(i) We first show that $|\sigma(0,i+2)| \leq 3 + |\sigma(0,i)|$.

Suppose we have some minimal length sequence $\sigma(2,i+2)$. We convert this into a sequence $\sigma(0,i+2)$ by considering two cases:

(a) Assume that $\underline{2}$ is preceded by 2 in the sequence $\sigma(2,i+2)$. Immediately following 2, insert 0, $\underline{1}$, $\underline{0}$ into the sequence.

(b) Assume that $\underline{2}$ is not preceded by 2 in the sequence. (Then it must be the case that $\underline{2}$ is the first field written.) Before $\underline{2}$, insert 1, $\underline{0}$ and after $\underline{2}$ insert $\underline{1}$.

Thus, $|\sigma(0,i+2)| \leq 3 + |\sigma(2,i+2)| = 3 + |\sigma(0,i)|$.

(ii) We now show that $|\sigma(0,i+2)| \geq 3 + |\sigma(0,i)|$.

A minimal sequence $\sigma(0,i+2)$ must contain $\underline{0}$, $\underline{1}$, and a minimal sequence $\sigma(2,i+2)$ will not contain these. But 1 must appear before $\underline{0}$, and 0 (as well as 2) must appear before $\underline{1}$. Therefore, it is necessary to include 0 or 1 in order to have $\underline{0}$, $\underline{1}$.

Thus, $|\sigma(0,i+2)| \geq 3 + |\sigma(0,i)|$. ∎

Using the previous two lemmas, we can compute $|\sigma(0,i)|$.

Lemma 6.9. For $i \in \mathbb{N}$, $i \geq 0$, we have:

(a) $|\sigma(0,2i)| = 3i + 1$

(b) $|\sigma(0,2i+1)| = 3i + 2$.

Proof: From Lemma 6.8, $|\sigma(0,i+2)| = 3 + |\sigma(0,i)|$. Now apply Lemma 6.7. For i even, this gives us

$$|\sigma(0,i+2)| = 3i + |\sigma(0,0)| = 3i + 1,$$

and for i odd we have

$$|\sigma(0,i+2)| = 3i + |\sigma(0,1)| = 3i + 2. \quad \blacksquare$$

We now apply this discussion of sequences σ and recall from Lemma 6.6 the correspondence to the POP operation. This now allows us to lower bound the number of accesses required to perform a POP operation.

Theorem 6.4. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^+$. Let $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$ be any BOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{\Diamond\}_{n_{|d|+1}}.$$

Let α_{POP} be any implementation of the POP operation. Then for all $d \in \mathbb{D}$:

$$\#[\alpha_{POP}(\rho(d))] \geq \begin{cases} 3 \cdot \frac{|d|-1}{2} + 2 & \text{if } |d| \text{ is odd} \\ 3 \cdot \frac{|d|}{2} + 1 & \text{if } |d| \text{ is even} \end{cases}$$

In other words,

$$\#[\alpha_{POP}(\rho(d))] \geq \lceil \frac{3|d| + 1}{2} \rceil.$$

Theorem 6.5 combines the results of theorem 6.3 and 6.4, along with the trivial observation that $\#[\alpha_{TOP}(\rho(d))] \geq 1$.

Theorem 6.5. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider a function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^\dagger$. Let $\rho: \mathbb{D} \rightarrow \mathcal{B}^\dagger$ be any BOS endmarker representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{0\}_0.$$

Let α_{POP} be any implementation of the POP operation, let α_{PUSHx} be any implementation of the PUSHx operation, and let α_{TOP} be any implementation of the TOP operation. Then for all $d \in \mathbb{D}$:

$$\#[\alpha_{POP}(\rho(d))] \geq \lceil \frac{3|d| + 1}{2} \rceil$$

$$\#[\alpha_{PUSHx}(\rho(d))] \geq |d| + 2$$

$$\#[\alpha_{TOP}(\rho(d))] \geq 1.$$

Recalling the algorithm for POP that we presented in Example 6.8, we now know that that algorithm is optimal for $|d|$ odd. Perhaps it would be possible to do one access better, however, when $|d|$ is even. As a consequence of the following lemma, it is impossible to simultaneously achieve the bounds of Theorem 6.4 for both $|d|$ odd and $|d|$ even.

Lemma 6.10. Let i be any even natural number. Suppose we have some minimal length sequence $\sigma_0(0,i)$ and some minimal length sequence $\sigma_1(0,i+1)$. Then $\sigma_0(0,i)$ is not a prefix of $\sigma_1(0,i+1)$.

Proof: The sequence $\sigma_0(0,i)$ must contain $\underline{0}, \underline{1}, \ldots, \underline{i-1}, \underline{i}$, and $\sigma_1(0,i+1)$ must contain $\underline{0}, \underline{1}, \ldots, \underline{i-1}, \underline{i}, \underline{i+1}$. Since i is even, $|\sigma_0(0,i)| = 3 \cdot \frac{i}{2} + 1$. Because $i + 1$ is odd and $\sigma_1(0,i+1)$ also has minimal length, $|\sigma_1(0,i+1)| = 3 \cdot \frac{i}{2} + 2$. Thus,

$$|\sigma_1(0,i+1)| = |\sigma_0(0,i)| + 1.$$

Suppose $\sigma_0(0,i)$ is a prefix of $\sigma_1(0,i+1)$. Since $|\sigma_0(0,i)|$ is minimal, $\sigma_0(0,i)$ does not contain $i + 1$, and therefore also does not contain $\underline{i}$, both of which must be present in $\sigma_1(0,i+1)$. So there is no way to append a sequence to $\sigma_0(0,i)$ in order to obtain a minimal length sequence $\sigma_1(0,i+1)$. ∎

Note that the proof of Lemma 6.10 does not hold if i is an odd number, because $|\sigma_1(0,i+1)| = 2 + |\sigma_0(0,i)|$ for i odd.

Theorem 6.5 gave lower bounds for implementing the stack operations with a BOS endmarker representation. Example 6.7 showed that the bounds for the PUSHx and TOP operations are actually achievable. We can also argue, as a consequence of Lemma 6.10, that the algorithm α_{POP} from Example 6.8 is access optimal. Since α_{POP} has a minimal number of accesses for $|d|$ odd, it cannot possibly achieve $3 \cdot \frac{|d|}{2} + 1$ accesses for $|d|$ even. Thus, the best it could possibly do would be $3 \cdot \frac{|d|}{2} + 2$ accesses for $|d|$ even, which is precisely what it does do. The following example shows that we could have constructed an algorithm for the POP operation which would have been minimal for $|d|$ even.

Example 6.9. Reconsider the representation ρ from examples 6.7 and 6.8. The algorithm α_{POP} from Example 6.8 is access optimal. Let α_{POP}' be an algorithm for the POP operation which has the field access sequence

$$0, 2, \underline{1}, \underline{0}, 4, \underline{3}, \underline{2}, 6, \underline{5}, \underline{4}, 8, \underline{7}, \underline{6}, \ldots$$

Note that α_{POP}' is, in fact, realizable, because this is basically the same algorithm we had before, only with a different starting sequence. This algorithm has for an access cost:

$$\#[\alpha_{POP}'(\rho(d))] \geq \begin{cases} 3 \cdot \frac{|d|}{2} + 1 & \text{if } |d| \text{ is even} \\ 3 \cdot \frac{|d|-1}{2} + 4 & \text{if } |d| \text{ is odd} \end{cases}$$

Thus, α_{POP}' requires a minimal number of accesses for $|d|$ even and is also access optimal. In fact, for Example 6.1, the BOS endmarker representation ρ with TOP and PUSHx implemented as in 6.1 and the POP implemented as in Example 6.8 is a storage and access optimal implementation $(\rho, \alpha_{POP}, \alpha_{PUSHx}, \alpha_{TOP})$.

As was the case for the TOS endmarker representation, Theorem 5.10 immediately tells us that a BOS endmarker representation achieves Kraft storage when the function f does.

Theorem 6.6. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$ and consider the function $f: X \cup \{\varnothing\} \rightarrow \mathcal{B}^+$. If the function f achieves Kraft storage, then the BOS endmarker representation $\rho: \mathbb{D} \rightarrow \mathcal{B}^+$, defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\diamond\}_{n_{|d|+1}},$$

also achieves Kraft storage.

So we have constructed the BOS endmarker as an alternative to the TOS endmarker representation scheme. We thereby decreased the access cost for performing a TOP operation, but in so doing we increased the cost of a POP operation. For a summary of the worst case lower bounds, see Table 6.3 at the end of the chapter.

6.4 The TOS Pointer Representation

Consider using an endmarker representation to implement a stack and the POP, PUSH, and TOP operations. The following example illustrates one possible such implementation.

Example 6.10. Let $X = \{a,b,c,d\}$, $\mathcal{B} = \{0,1,2,3\}$, and $\mathbb{D} = \bigcup_{i=0}^{3} X^i$. Let the function $f:X \to \mathcal{B}^*$ be defined by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \\ f(c) &= 2 \\ f(d) &= 3 \end{aligned}$$

Define the concatenation-preserving pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_i \cup \{\ell(|d|)\}_0,$$

where the pointer component $\ell:\mathcal{J} \to \mathcal{B}^+$ is defined by

$$\begin{aligned} \ell(0) &= 0 \\ \ell(1) &= 1 \\ \ell(2) &= 2 \\ \ell(3) &= 3 \end{aligned}$$

For instance,

$$\begin{aligned} \rho(\lambda) &= 0 \\ \rho(d) &= 13 \\ \rho(cab) &= 3201 \end{aligned}$$

We assume that L is large enough to represent any $d \in \mathbb{D}$; in particular $L \geq 4$. In order to perform a POP operation in this example we need only decrement the pointer. Notice that there is no need to read any stack elements, since decrementing the pointer automatically decreases $|\rho(d)|$ by one. So we could use the following simple algorithm to perform a POP operation.

α_{POP}: if $m(0) = 0$ then return "Error"
$m(0) \leftarrow m(0) - 1$

By our definition of a memory cell access, this algorithm for POP corresponds to a single access; we read the contents of cell 0 and then, depending on its contents, we

may rewrite the value. If $m(0) = 0$, then we return an "Error" message and the second line in the algorithm never gets executed. For a PUSHx or a TOP operation, however, we must read the pointer in order to determine where the top of the stack is, and we then go to the appropriate stack location to perform the operation.

α_{PUSHx}: if $m(0) = 3$ then return "Error"
$m(0) \leftarrow m(0) + 1$
$m(m(0)) \leftarrow f(x)$

α_{TOP}: if $m(0) = 0$ then return "Error"
return $m(m(0))$

These algorithms give the following access costs:

$$\#[\alpha_{POP}(\rho(d))] = 1$$

$$\#[\alpha_{PUSHx}(\rho(d))] = \begin{cases} 2 & \text{if } |d| \neq 3 \\ 1 & \text{else} \end{cases}$$

$$\#[\alpha_{TOP}(\rho(d))] = \begin{cases} 2 & \text{if } |d| \neq 0 \\ 1 & \text{else} \end{cases}$$

∎

Notice that the representation ρ in the above example allowed us to implement the set of stack states $\mathbb{D} = \bigcup_{i=0}^{3} X^i$ with low update costs, lower than was possible with the TOS or BOS endmarker representations.

We extend the pointer scheme illustrated in Example 6.10 and make the following definition.

Definition. Let $\mathbb{D} = \bigcup_{i=0}^{k} X^i$, for $k \in \mathbb{N}$ and consider a function $f: X \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any fixed position field pointer representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\ell(|d|)\}_n$$

where $n, n_i \in \mathbb{N}$ (for any $0 \leq i \leq k$) and where ℓ is a representation $\ell: \mathcal{J} \to \mathcal{B}^+$. Then ρ is a *TOS pointer representation.*

We use the term TOS because reading the pointer component, $\ell(|d|)$, tells us $|d|$ and we can then go directly to the field $n_{|d|}$ in order to determine the top of stack element. Clearly the representation ρ from Example 6.10 is a TOS pointer representation. If $|\mathbb{D}|$ is large, and especially if $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, then the size of the pointer will grow large. Therefore, we may sometimes find it convenient to view the TOS pointer representation as a separate pointer representation.

Restricting our consideration to concatenation-preserving representations is perhaps an obvious thing to do, but let us discuss why we also require that a TOS pointer representation have fixed position fields. The fixed position field assumption is included as a consequence of our definition of a pointer representation, where we chose to encode $|d|$ rather that $|\rho(d)|$. If we were to allow variable position fields, then knowing $\ell(|d|)$ would not necessarily tell us the location of the top of the stack.

Unfortunately, requiring fixed position fields will, in general, result in "gaps" in the representation, unless $|f(x_1)| = |f(x_2)|$ for all $x_1, x_2 \in X$. Thus, if we insist on Kraft storage, a TOS pointer representation must sometimes have gaps when $|X| \neq |\mathcal{B}|$. We could, alternatively, have defined a TOS pointer representation ρ to be a concatenation-preserving representation $\rho:\mathbb{D} \in \mathcal{B}^*$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i(d)} \cup \{\ell(|\rho(d)|)\}_0$$

where

$$n_i = |\ell(|\rho(d)|)| + \sum_{j=1}^{i-1} |f(d(j))|.$$

Such a definition would avoid the problem of having gaps in the storage of $\rho(d)$ and would not affect the storage and access results we obtain. Thus, our original definition of a TOS pointer representation is satisfactory for our purposes.

In Example 6.10, the domain size was small enough that the pointer component was able to fit in a single memory cell. For a larger but bounded domain size, we can still store a stack pointer in a fixed number of memory cells, as we do in the following example.

Example 6.11. Let $X = \{a,b,c\}$, $\mathcal{B} = \{0,1\}$, and $\mathbb{D} = \bigcup_{i=0}^{7} X^i$. Define the function $f:X \to \mathcal{B}^*$ by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 10 \\ f(c) &= 11 \end{aligned}$$

and the pointer component $\ell:\mathcal{J} \to \mathcal{B}^*$ by

$$\begin{aligned} \ell(0) &= 000 & \qquad \ell(4) &= 100 \\ \ell(1) &= 001 & \qquad \ell(5) &= 101 \\ \ell(2) &= 010 & \qquad \ell(6) &= 110 \\ \ell(3) &= 011 & \qquad \ell(7) &= 111. \end{aligned}$$

Then we can define the TOS pointer representation $\rho:\mathbb{D} \to \mathcal{B}^*$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{2(i-1)+3} \cup \{\ell(|d|)\}_0.$$

(We assume, of course, that $L \geq 17$.) Then we have, for instance,

$$\begin{aligned} \rho(\lambda) &= 000 \\ \rho(abc) &= 0110\_1011 \\ \rho(accab) &= 1010\_11110\_10 \end{aligned}$$

Notice that the representation ρ achieves Kraft storage.

We can implement the stack operations roughly as follows. For the TOP operation, we read the three pointer cells and then go to the top of the stack to look up the answer.

```
α_TOP:   temp1 ← m(2) + 2·m(1) + 4·m(0)
         if temp1 = 0 then return "Error"
         temp2 ← 2·(temp1-1) + 3
         if m(temp2) = 0 then return "a"
                         else if m(temp2+1) = 0 then return "b"
                                                else return "c"
```

In order to do a PUSHx operation, we must increment the three pointer cells as we read them, and we then insert the correct item onto the stack.

$\alpha_{\text{PUSH}x}$:
```
        if m(2) = 0 then  m(2) ← 1
                          goto write
        m(2) ← 0
        if m(1) = 0 then  m(1) ← 1
                          goto write
        m(1) ← 0
        if m(0) = 0 then  m(0) ← 1
                          goto write
        m(2) ← 1
        m(1) ← 1
        return "Error"
write:  temp1 ← 2·(m(2) + 2·m(1) + 4·m(0)) + 1
        if x = a then m(temp1) ← 0
                 else m(temp1) ← 1
                      if x = b then m(temp1) + 1 ← 0
                               else m(temp1+1) ← 1
```

For the POP operation, we need only decrement the pointer. Unfortunately, this may require accessing some pointer memory cell more than once. The following simple algorithm is one possibility.

α_{POP}:
```
        if m(2) = 1 then  m(2) ← 0
                          return
        m(2) ← 1
        if m(1) = 1 then  m(1) ← 0
                          return
        m(1) ← 1
        if m(0) = 1 then  m(0) ← 0
                          return
        m(1) ← 0
        m(2) ← 0
        return "Error"
```

Notice that this algorithm causes us to incorrectly change $m(1)$ and $m(2)$ in the case where an "Error" condition is to be returned, thus forcing us to go back and rewrite these cells.

Excluding the cases where we get an Error condition, these three algorithms give us the following access costs, for all $d \in \mathbb{D}$:

$$\begin{aligned} 5 \geq \quad & \#[Q_{TOP}(\rho(d))] \quad \geq 4 \\ 5 \geq \quad & \#[Q_{PUSHx}(\rho(d))] \quad \geq 4 \\ 5 \geq \quad & \#[Q_{POP}(\rho(d))] \quad \geq 4 \end{aligned}$$ ∎

The strategy used in Example 6.11 for implementing the stack could be used with any TOS pointer representation which has a fixed size pointer field.

Definition. Let $\mathbb{D} = \bigcup_{i=0}^{k} X^i$, let $r = \lceil \log_{|\mathcal{B}|}(k+1) \rceil$, and let f be a function $f: X \to \mathcal{B}^+$. Suppose the pointer component ℓ is a one-to-one function $\ell: \{0,1,\ldots,k\} \to \mathcal{B}^r$. Then the TOS pointer representation $\rho: \mathbb{D} \to \mathcal{B}^+$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_1+r} \cup \{\ell(|d|)\}_0,$$

is said to be a TOS pointer representation *with a fixed size pointer field.*

The TOS pointer representations in both examples 6.10 and 6.11 have fixed size pointer fields, and we implemented the stack operations in essentially the same way, first reading the pointer and then, if necessary, accessing the list component.

Theorem 6.7. Let $\mathbb{D} = \bigcup_{i=0}^{k} X^i$ and let $r = \lceil \log_{|\mathcal{B}|}(k+1) \rceil$. Let f be a function $f: X \to \mathcal{B}^*$ such that $\max_{x \in X} |f(x)| = t$. Consider the TOS pointer representation $\rho: \mathbb{D} \to \mathcal{B}^*$, with a fixed size pointer field, defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{t(i-1)+r} \cup \{\ell(|d|)\}_0,$$

where $\ell: \mathcal{J} \to \mathcal{B}^r$. Then it is possible to define the representation ℓ in such a way that the stack operations can be implemented with algorithms which have the following access costs. For all $d \in \mathbb{D}$,

$$\begin{aligned} r + t \quad & \geq \quad \#[Q_{TOP}(\rho(d))] \quad \geq r + 1 \\ r + t \quad & \geq \quad \#[Q_{PUSHx}(\rho(d))] \quad \geq r + 1 \\ 2r - 1 \quad & \geq \quad \#[Q_{POP}(\rho(d))] \quad \geq 1 \end{aligned}$$

Proof: The construction of algorithms for the TOP, PUSHx, and POP operations is the same as in Example 6.11, and we shall not present all of the details here. We define $\ell(i) \in \mathcal{B}^r$ so that when the string $\ell(i)$ is viewed as a number it is the base $|\mathcal{B}|$ representation of i (with preceding 0's, if necessary, since $|\ell(i)| = r$).

We construct α_{TOP} so that it reads the r memory cells in the pointer and then goes to field $n_{|d|}$ to read the top stack element. Thus, α_{TOP} accesses at least r + 1 and at most r + t memory cells, depending on the size of the representation of the element at the top of the stack.

Now consider implementing an α_{PUSHx} algorithm. By the way we have defined the pointer component ℓ, it is possible to increment the pointer as we read it, if we access cells in the order $m(r\text{-}1), m(r\text{-}2), \ldots, 1, 0$. (See Example 6.11 for an illustration.) After reading the r pointer cells, we locate the appropriate field and write f(x), a total of $r + |f(x)|$ accesses.

For the α_{POP} algorithm we need only decrement the pointer. So it would never be necessary to make more than $2 \cdot r - 1$ accesses, because we could just read the pointer in one pass and rewrite it in the next. On the lower bound side, we clearly need to make at least one access. ∎

Notice that, using the method from Example 6.11, the $2 \cdot r - 1$ upper bound on the number of accesses for the POP operation would be attained only when $|d| = 0$ and an "Error" message is returned. For $d \neq \lambda$, r would be an upper bound and we frequently would be able to do even better.

In the proof of Theorem 6.7, the only reference to the particular pointer ℓ we chose was in obtaining the upper bound for the cost of performing a PUSHx operation. As we argued there for the POP operation, it would always be possible to increment the pointer by making $2 \cdot r - 1$ accesses. This gives us the following corollary.

Corollary 6.7.1. Let $\mathbb{D} = \bigcup_{i=0}^{k} X^i$ and let $r = \lceil \log_{|\mathcal{B}|}(k+1) \rceil$. Let f be a function $f: X \to \mathcal{B}^*$ such that $\max_{x \in X} |f(x)| = t$. Consider the TOS pointer representation $\rho: \mathbb{D} \to \mathcal{B}^*$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{t(i-1)+r} \cup \{\ell(|d|)\}_0.$$

Then for any one-to-one pointer function $\ell: \mathcal{J} \to \mathcal{B}^r$, it is possible to implement the stack operations so as to obtain the following access costs. For all $d \in \mathbb{D}$,

$$\begin{aligned} r + t &\geq \#[Q_{TOP}(\rho(d))] \geq r + 1 \\ 2r - 1 + t &\geq \#[Q_{PUSHx}(\rho(d))] \geq r + 1 \\ 2r - 1 &\geq \#[Q_{POP}(\rho(d))] \geq 1 \end{aligned}$$

Theorem 6.7 and Corollary 6.7.1 gave us upper bounds on access costs for performing POP, PUSHx, and TOP operations using a TOS pointer representation with fixed position fields. The bounds depend on r, not on $|d|$, although the size of r itself is dependent on $\max_{d \in \mathbb{D}} |d|$: $r = \lceil \log_{|\mathcal{B}|}(\max_{d \in \mathbb{D}} |d| + 1) \rceil$. Thus, when $|d|$ is small, being forced to read r cells could be relatively expensive (e.g., when r is large and the stacks we are representing are small). Consider, however, where these bounds came from. We can rewrite the result of Corollary 6.7.1. For any TOS pointer representation with a fixed size pointer field, we can implement the stack operations with the following access costs:

$$\begin{aligned} \#[Q_{TOP}(\rho(d))] &\leq |\ell(|d|)| + |f(q_{TOP}(d))| \\ \#[Q_{PUSHx}(\rho(d))] &\leq |\ell(|d|)| + |f(x)| \\ \#[Q_{POP}(\rho(d))] &\leq \begin{cases} 2 \cdot |\ell(|d|)| - 1 & \text{for } |d| = 0 \\ |\ell(|d|)| & \text{for } |d| \neq 0 \end{cases} \end{aligned}$$

assuming that the function f is a representation and achieves Kraft storage.

Let us now extend these results to TOS pointer representations where we do not have fixed size pointer fields. We would also like to allow $\mathbb{D} = \bigcup_{i=0}^{k} X^i$, where $k \leq \omega$. From the above discussion it should be easy to see that the following

theorem holds.

Theorem 6.8. Let $\mathbb{D} = \bigcup_{i=0}^{k} X^i$, where $k \leq \omega$, and let $f:X \to \mathcal{B}^*$ achieve Kraft storage. Consider the TOS pointer representation $\rho:\mathbb{D} \to \mathcal{B}^\dagger$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\ell(|d|)\}_n,$$

where ℓ is any representation $\ell:\mathcal{J} \to \mathcal{B}^\dagger$. Then it is possible to implement the stack operations so as to achieve the following access costs. For all $d \in \mathbb{D}$

$$\begin{aligned} \#[\alpha_{TOP}(\rho(d))] &\leq |\ell(|d|)| + |f(q_{TOP}(d))| \\ \#[\alpha_{PUSHx}(\rho(d))] &\leq 2\cdot|\ell(|d|)| + |f(x)| - 1 \\ \#[\alpha_{POP}(\rho(d))] &\leq 2\cdot|\ell(|d|))| - 1 \end{aligned}$$

Proof: For any TOS pointer representation, reading the pointer immediately tells us the location of the top of the stack. So we can certainly perform a TOP operation, by accessing each pointer cell and then reading enough cells in the list component for us to distinguish $q_{TOP}(d)$. Since f achieves Kraft storage, this is precisely $|\ell(|d|)| + |f(q_{TOP}(d))|$. For the PUSHx and POP operations, it is, in general, necessary to rewrite the pointer, which at worst would require $2\cdot|\ell(|d|)| - 1$ accesses: one pass over $\ell(|d|)$ to read and the next to rewrite. For a POP, we need not access the list component at all, and for a PUSHx, we need to write $f(x)$ into memory. ∎

From Theorem 6.8, the issue is now to see how compact we can make our pointer component $\ell(|d|)$. Recalling the construction of the class of pointers $\ell^{k}_{|\mathcal{B}|}$ from Section 5.4 (see Table 5.2), we have a possible representation scheme, with

$$|\ell(|d|)| = O(\log |d|).$$

Consider using this scheme to perform a PUSHx or a POP. Since each pointer is a representation of a natural number n, we want to be able to increment or decrement by one the number to be represented. For the scheme in Section 5.4, this means we always need to alter the "rightmost" cell in the pointer representation. Since the size

of the pointer component is not fixed (in fact, it may be unbounded), there is no way to know where this rightmost cell is, unless we read (most of) $\ell(|d|)$. Even then, we might be forced to backtrack. We shall construct a new pointer scheme, with the same storage cost as our previous one, but for which it will be easier to increment and decrement the pointer. This makes it, on the average, cheaper to perform a POP operation.

Recall the pointer representation scheme $\ell\,\frac{1}{2}$ as illustrated in Table 5.1:

$$\ell\,\tfrac{1}{2}(n) = 0^{|h(n)|}\cdot 1\cdot h(n),$$

where we write $h(n)$ for $h_2(n)$. For instance, consider

$$\ell\,\tfrac{1}{2}(18) = 000010011.$$

In order to perform a POP operation on a stack of length 18, we need to decrement the pointer, leaving us

$$\ell\,\tfrac{1}{2}(17) = 000010010.$$

Notice that we needed to alter only the last bit in the pointer, but there is no way to locate this last bit without reading the entire pointer. If we could rearrange bits so that we read the last bit (of $h(n)$) early, then whenever n is even we would just change the appropriate bit to 1 and immediately be finished with our POP operation. We can do this by interspersing the bits of $\ell\,\frac{1}{2}(n)$ from the $0^{|h(n)|}$ component with those from the $h(n)$ component (using an extra 1 to denote the end of the pointer representation). Note that these two components each have the same number of bits. Since we would like to be able to read the last bit of $h(n)$ as early as possible, we reverse the bit order of $h(n)$. Such a strategy gives us

$$\Lambda\,\tfrac{1}{2}(18) = 0\underline{1}0\underline{1}0\underline{0}0\underline{0}\underline{1}$$

$$\Lambda\,\tfrac{1}{2}(17) = 0\underline{0}0\underline{1}0\underline{0}0\underline{0}\underline{1}.$$

For clarity, we have underlined the bits that come from the $h(n)$ component. Some additional values of $\Lambda\,\frac{1}{2}$ are given in Table 6.1.

We now give a formal definition of the pointer representation scheme $\Lambda\,\frac{1}{2}$. We begin with the following preliminary definition, based on the definition of the string $h_{|\beta|}(n)$ from Section 5.4.

Definition. For $|\mathcal{B}| \geq 2$, we define the string

$$\Theta_n^{|\mathcal{B}|} \triangleq (h_{|\mathcal{B}|}(n))^R;$$

i.e., the reverse of the string $h_{|\mathcal{B}|}(n)$. For $1 \leq i \leq |\Theta_n^{|\mathcal{B}|}|$, we write $\Theta_n^{|\mathcal{B}|}(i)$ to denote the i^{th} component of the string $\Theta_n^{|\mathcal{B}|}$. For notational simplicity we may simply write Θ_n to stand for Θ_n^2.

So $\Theta_n^{|\mathcal{B}|}(1)$ is the last character in the string $h_{|\mathcal{B}|}(n)$, $\Theta_n^{|\mathcal{B}|}(2)$ is the next to the last character in the string $h_{|\mathcal{B}|}(n)$, etc.

Example 6.12. Since $h_2(18) = 0011$. Then $\Theta_{18} = 1100$, and we have $\Theta_{18}(1) = 1$, $\Theta_{18}(2) = 1$, $\Theta_{18}(3) = 0$, $\Theta_{18}(4) = 0$. ∎

We now define the pointer representation Λ_2^1, in terms of the string Θ_n.

Definition. Let $|\mathcal{B}| \geq 2$. We define the pointer representation scheme Λ_2^1 as follows:

$$\Lambda_2^1(n) \triangleq \bigcup_{i=1}^{|\Theta_n|} \{0 \cdot \Theta_n(i)\}_{2(i-1)} \cup \{1\}_{2|\Theta_n|}.$$

We illustrate the definition with an example.

Example 6.13. Let us determine the pointer $\Lambda_2^1(26)$. Recall from Section 5.4 that $h_2(26) = 1011$. So $\Theta_2(26) = 1101$, and

$$\begin{aligned}\Lambda_2^1(26) &= \{0 \cdot 1\}_0 \cup \{0 \cdot 1\}_2 \cup \{0 \cdot 0\}_4 \cup \{0 \cdot 1\}_6 \cup \{1\}_8 \\ &= 010100011.\end{aligned}$$ ∎

Table 6.1 gives the pointer representations $\Lambda_2^1(n)$ for $0 \leq n \leq 33$.

Now that we have defined the pointer representation scheme Λ_2^1, let us use this scheme and determine access costs for implementing the stack operations.

n	$h(n)$	Θ_n	$\Lambda_{\frac{1}{2}}(n)$
0	-	-	1
1	0	0	001
2	1	1	011
3	00	00	00001
4	01	10	01001
5	10	01	00011
6	11	11	01011
7	000	000	0000001
8	001	100	0100001
9	010	010	0001001
10	011	110	0101001
11	100	001	0000011
12	101	101	0100011
13	110	011	0101001
14	111	111	0101011
15	0000	0000	000000001
16	0001	1000	010000001
17	0010	0100	000100001
18	0011	1100	010100001
19	0100	0010	000001001
20	0101	1010	010001001
21	0110	0110	000101001
22	0111	1110	010101001
23	1000	0001	000000011
24	1001	1001	010000011
25	1010	0101	000100011
26	1011	1101	010100011
27	1100	0011	000001011
28	1101	1011	010001011
29	1110	0111	000101011
30	1111	1111	010101011
31	00000	00000	0000000001
32	00001	10000	0100000001
33	00010	01000	0001000001

Table 6.1. Construction of pointer representation $\Lambda_{\frac{1}{2}}$.

Theorem 6.9. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, let $\mathcal{B} = \{0,1\}$, and consider a function $f: X \to \mathcal{B}^\dagger$. Let $\rho: \mathbb{D} \to \mathcal{B}^\dagger$ be a TOS pointer representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Lambda^{\frac{1}{2}}(|d|)\}_n,$$

where $n, n_i \in \mathbb{N}$. Let $k_1 \in \mathbb{N}$. Then it is possible to implement the stack operations so as to achieve the following access costs for all $d \in \mathbb{D}$.

$$\#[\mathcal{A}_{TOP}(\rho(d))] \leq |\Lambda^{\frac{1}{2}}(|d|)| + k_1$$
$$\#[\mathcal{A}_{PUSHx}(\rho(d))] \leq |\Lambda^{\frac{1}{2}}(|d|)| + |f(x)| + 2$$
$$\#[\mathcal{A}_{POP}(\rho(d))] \leq |\Lambda^{\frac{1}{2}}(|d|)| + 1$$

Proof: As we have previously seen, it is certainly possible to implement the TOP operation by reading the entire pointer and then going to the appropriate location to look up the answer $q_{TOP}(d)$. Although a lookup of this answer might require making more than $|f(q_{TOP}(d))|$ accesses, it cannot take more than some constant number of accesses, depending on details of the function f.

We have constructed the representation scheme $\Lambda^{\frac{1}{2}}$ so that it will be easy to decrement the stack pointer. Consider the following algorithm:

```
A_POP       if m(0) = 1 then return "Error"
            i ← 1
loop:       if m(i) = 1 then  m(i) ← 0
                              return
            m(i) ← 1
            if m(i+1) = 1 then m(i-1) ← 1
                               return
            i ← i + 2
            goto loop
```

In this algorithm, we read the pointer from left to right and never backtrack over more than one cell. This gives the desired bound for POP.

A similar scheme allows us to perform a PUSHx operation.

```
A_PUSHx:    if m(0) = 1 then  m(0) ← 0
                              m(1) ← 0
                              m(2) ← 1
                              return
```

```
        i ← 1
loop:   if m(i) = 0 then  m(i) ← 1
                          return
        m(i) ← 0
        if m(i+1) = 1 then  m(i+1) ← 0
                            m(i+2) ← 0
                            m(i+3) ← 1
                            return
        i ← i+2
        goto loop
```

In this case we read the entire pointer and, although we never need to backtrack, we sometimes need to rewrite two additional cells. Having incremented the pointer, we can insert $f(x)$ in the appropriate field with $|f(x)|$ accesses. ∎

We can see that we have improved our previous access costs, so that each stack operation can be implemented with at most $O(\log|d|)$ accesses in the worst case. In fact, the next result shows that we could expect to do even better for a POP operation because for a very reasonable probability distribution we can expect to make, on the average, only a constant number of accesses.

Theorem 6.10. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, let $\mathcal{B} = \{0,1\}$, and consider a function $f: X \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be a TOS pointer representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Lambda\, \tfrac{1}{2}(|d|)\}_n$$

where $n,\ n_i \in \mathbb{N}$. Assume that there is a monotonically nonincreasing probability distribution P on the stack states:

$$P(|d| = n + 1) \leq P(|d| = n).$$

Then it is possible to implement the POP operation so that

$$\text{avg}\#[Q_{POP}(\rho(d))] \leq k,$$

for some $k \in \mathbb{N}$.

Proof: Consider the algorithm $\mathcal{A}_{POP}$ presented in the proof of Theorem 6.9. Note that 2 accesses are required for $|d| = 2,4,6,8,10, \ldots$, that 4 accesses are required for $|d| = 5,9,13,17, \ldots$, that 6 accesses are required for $|d| = 11,19,27,35, \ldots$, etc. Denote $P(|d| = i)$ by p_i. Since $p_{n+1} \leq p_n$, we know that

$$p_2 + p_4 + p_6 + p_8 + \ldots \leq p_1 + p_3 + p_5 + p_7 + \ldots$$

and so
$$p_2 + p_4 + p_6 + p_8 + \ldots \leq \frac{1}{2}.$$

Similarly,
$$p_5 + p_9 + p_{13} + p_{17} + \ldots \leq \frac{1}{4},$$
$$p_{11} + p_{19} + p_{27} + p_{35} + \ldots \leq \frac{1}{8},$$
$$p_{23} + p_{39} + p_{55} + p_{71} + \ldots \leq \frac{1}{16}, \text{ etc.}$$

Notice that extra work is required to perform the POP whenever $|d| = 1$, $|d| = 3$, $|d| = 7$, $|d| = 15$, etc. (i.e., when $|d| = 2^i - 1$ for some $i \in \mathbb{N}$). Thus,

$$\begin{aligned}\sum_{i=0}^{\infty} p_i \cdot \#[\mathcal{A}_{POP}(\rho(|d| = i))] &\leq \sum_{i=1}^{\infty} \frac{2i}{2^i} + \sum_{k=0}^{\infty} p_{(2^k-1)} \cdot 2(k+1) \\ &\leq \sum_{i=1}^{\infty} \frac{2i}{2^i} + \sum_{k=0}^{\infty} \frac{1}{2^k} \cdot 2(k+1) \\ &= \sum_{i=1}^{\infty} \frac{4i}{2^i} + \sum_{k=0}^{\infty} \frac{1}{2^i} \\ &= 10 \end{aligned}$$

∎

The following theorem summarizes the results we have just derived.

Theorem 6.11. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, let $\mathcal{B} = \{0,1\}$, and consider a function $f: X \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be a TOS pointer representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_i} \cup \{\Lambda_{\frac{1}{2}}(|d|)\}_n$$

where n, $n_i \in \mathbb{N}$. Assume that there is a monotonically nonincreasing probability distribution P on the stack states:

$$P(|d| = n + 1) \leq P(|d| = n).$$

Let $k_2, k_3 \in \mathbb{N}$. Then $\Lambda_{\frac{1}{2}}$ achieves Kraft storage, and it is possible to implement the stack operations so as to achieve the following access costs:

$$\#[Q_{TOP}(\rho(d))] \leq 2\cdot \lfloor \log_2(|d|+1) \rfloor + k_2$$
$$\#[Q_{PUSHx}(\rho(d))] \leq 2\cdot \lfloor \log_2(|d|+1) \rfloor + |f(x)| + 3$$
$$\#[Q_{POP}(\rho(d))] \leq k_3.$$

Proof: The result for the POP operation is the result of Theorem 6.10. We obtain the inequalities for TOP and PUSHx by recalling that $|\Lambda\ ^1_2(n)| = |\ell\ ^1_2(n)|$ and by making use of Lemma 5.7 and theorems 5.18 and 6.9. The Kraft storage follows from Theorem 5.19. ∎

We have chosen to prove these results for the pointer scheme $\Lambda\ ^1_2$, but the scheme can be extended to include $\Lambda\ ^1_{|\mathcal{B}|}$. As it turns out, the access costs we obtain are even better than for $\Lambda\ ^1_2$, although the results are all of the same order of growth. Because the details would tend to obscure an understanding of the class of pointer schemes Λ, we shall not formally define $\Lambda\ ^i_{|\mathcal{B}|}$ for $|\mathcal{B}| > 2$ or $i > 1$. But let us indicate informally how these extensions could be made. Note that we shall always have

$$|\Lambda\ ^i_{|\mathcal{B}|}(n)| = |\ell\ ^i_{|\mathcal{B}|}(n)|,$$

and, in fact, the string $\Lambda\ ^i_{|\mathcal{B}|}(n)$ is just a rearrangement of the elements in the string $\ell\ ^i_{|\mathcal{B}|}(n)$.

Consider $|\mathcal{B}| = 3$ and recall $h_3(n)$ from Table 5.2. Since we want to construct $\Lambda\ ^1_3(n)$ in such a way that it is a rearrangement of the elements in $\ell\ ^1_3(n)$, recall that

$$\ell\ ^1_3(n) = h_2(|h_3(n)|)\cdot 2\cdot h_3(n).$$

In this case the first (pointer) component of $\ell\ ^1_3(n)$ has only about $\log_2(|h_3(n)|)$ elements, whereas the second (list) component has $|h_3(n)|$ elements. So we clearly cannot just use every other cell for the first component, as we did with $\Lambda\ ^1_2(n)$. Referring again to Table 5.2, we see that our pointer component has a 0 when the list component has length 1, has a 1 when the list component has length 2, has 00 when the list component has length 3, etc. So

$$|h_3(n)| = \sum_{i=1}^{|\theta_n|} 2^{i-1}\theta_n(i),$$

where θ_n denotes $\theta\,{}^{3}_{n}$. When $|h_3(n)| = 1$ then $\Lambda\,{}^{1}_{3}(n)$ is of the form 0_, when $|h_3(n)| = 2$ then $\Lambda\,{}^{1}_{3}(n)$ is of the form 1_, etc. This scheme is illustrated in Figure 6.1. The string $\theta\,{}^{3}_{n}$ is written out in blocks of size 2^i and the coefficient of each block, 0 or 1, tells whether there are 2^i or $2\cdot 2^i$ elements, respectively, in that block. Rather than attempt to say more in words, we refer the reader to Table 6.2.

$\lvert\theta_n\rvert$	form of $\Lambda\,{}^{1}_{3}(n)$
1	0_2
2	1_ _2
3	0_0_ _2
4	1_ _0_ _2
5	0_1_ _ _ _2
6	1_ _1_ _ _ _2
7	0_0_ _0_ _ _ _2
8	1_ _0_ _0_ _ _ _2
9	0_1_ _ _ _0_ _ _ _2
10	1_ _1_ _ _ _0_ _ _ _2
11	0_0_ _1_ _ _ _ _ _ _ _2

Figure 6.1. Outline of scheme for $\Lambda\,{}^{1}_{3}(n)$.

It is also possible to construct $\Lambda\,{}^{1}_{|\mathcal{B}|}$ for $i > 1$. The procedure is outlined in Figure 6.2. Notice that we write the initial part of θ_n, as much as possible, in blocks of size 1, 2, 2^2, 2^3, 2^4, etc. Of course, when $|\theta_n| \neq \sum_{i=0}^{k} 2^i$ for some k (i.e., $|\theta_n| \neq 1,3,7,15$, etc.), then some digits in θ_n will be left over. In particular, let

$$r = \min\{j \mid \sum_{i=0}^{j+1} 2^i > |\theta_n|\}$$

Then we can write the first $\sum_{i=0}^{r} 2^i$ elements in blocks of size powers of 2, each block preceded by a 0; a 1 indicates when we do not want to continue reading the next

n	θ_n	Λ_3^1	$\Lambda_2^2(n)$
0	-	2	1
1	0	002	0010
2	1	012	0110
3	00	1002	00110
4	10	1102	01110
5	01	1012	00111
6	11	1112	01111
7	000	000002	00000100
8	100	010002	01000100
9	010	000102	00010100
10	110	010102	01010100
11	001	000012	00001100
12	101	010012	01001100
13	011	000112	00011100
14	111	010112	01011100
15	0000	1000002	000001100
16	1000	1100002	010001100
17	0100	1010002	000101100
18	1100	1110002	010101100
19	0010	1000102	000011100
20	1010	1100102	010011100
21	0110	1010102	000111100
22	1110	1110102	010111100
23	0001	1000012	000001110
24	1001	1100012	010001110
25	0101	1010012	000101110
26	1101	1110012	010101110
27	0011	1000112	000011110
28	1011	1100112	010011110
29	0111	1010112	000111110
30	1111	1110112	010111110
31	00000	00100002	0000010100
32	10000	01100002	0100010100
33	01000	00110002	0001010100

Table 6.2. Construction of pointer representations Λ_3^1 and Λ_2^2.

succesively larger block. We must then represent the remaining $|\Theta_n| - \sum_{i=0}^{} 2^i$ elements

or 1 to indicate its presence or absence. Table 6.2 presents sample values

$\|\Theta_n\|$	form of $\Lambda_g^2(n)$
1	0_10
2	0 = 0_11_
3	1 = 0_0_100
4	2 = 0_0_11_0
5	3 = 0_0_101_ _
6	0_0_11_1
7	0_0_0_1000
8	0 = 0_0_ _11_00
9	1 = 0_0_ _101_ _0
10	2 = 0_0_ _11_1_ _0
11	3 = 0_0_ _1001_ _ _ _

Figure 6.2. Outline of scheme for $\Lambda_g^2(n)$.

6.5 The BOS Pointer Representation

For the sake of completeness, let us briefly mention the bottom of stack pointer representation.

Example 6.14. Recall Example 6.10, where we had $X = \{a,b,c,d\}$, $\mathcal{B} = \{0,1,2,3\}$, and $\mathbb{D} = \bigcup_{i=0}^{3} X^i$. The function $f:X \to \mathcal{B}^*$ is defined by

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \\ f(c) &= 2 \\ f(d) &= 3 \end{aligned}$$

and the pointer $\ell:\mathcal{J} \to \mathcal{B}^+$ is defined by

$$\begin{aligned} \ell(0) &= 0 \\ \ell(1) &= 1 \\ \ell(2) &= 2 \\ \ell(3) &= 3 \end{aligned}$$

Then we can define the concatenation-preserving pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$ by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{|d|-i+1} \cup \{\ell(|d|)\}_0.$$

For instance,

$$\begin{aligned} \rho(\lambda) &= 0 \\ \rho(d) &= 13 \\ \rho(cab) &= 3102 \end{aligned}$$

Assuming L is large enough to represent any $d \in \mathbb{D}$ (i.e., $L \geq 4$), let us construct algorithms to implement the stack operations. In order to perform a POP operation we need not only decrement the pointer but the contents of all of the memory cells will have to be shifted left by one.

α_{POP}:
```
if m(0) = 0 then return "Error"
m(0) ← m(0) - 1
i ← m(0) + 1
while i > 1 do    m(i-1) ← m(i)
                  i ← i - 1
```

Similarly, the PUSHx operation requires that the contents of each cell be shifted

right by one.

α_{PUSHx}: if $m(0) = 3$ then return "Error"
$m(0) \leftarrow m(0) + 1$
temp1 $\leftarrow m(0)$
temp2 $\leftarrow m(1)$
$m(1) \leftarrow f(x)$
$i \leftarrow 2$
while $i \leq$ temp1 then temp2 $\leftrightarrows m(i)$
$i \leftarrow i + 1$

The TOP operation is much easier.

α_{TOP}: if $m(0) = 0$ then return "Error"
return $m(1)$

∎

We can extend the pointer scheme in the previous example and define the BOS pointer representation in the obvious way.

Definition. Let $\mathbb{D} = \bigcup_{i=0}^{\infty} X^i$, for $k \in \mathbb{N}$ and consider a function $f: X \to \mathcal{B}^+$. Let $\rho: \mathbb{D} \to \mathcal{B}^+$ be any pointer representation

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{\ell(|d|)\}_n$$

where $n, n_i \in \mathbb{N}$ (for any $0 \leq i \leq k$) and where ℓ is a representation $\ell: \mathcal{I} \to \mathcal{B}^+$. Then ρ is a *bottom of stack (BOS) pointer representation.*

The types of arguments used in the preceding sections can be used to determine the access costs for implementing the BOS pointer representation. For PUSHx or POP, the elements in the stack will all have to be moved, requiring an access to each n_i field and also reading the entire pointer component (assuming the pointer achieves Kraft storage). A TOP operation is, however, cheap since it is always located in the same field, assuming, of course, that there is a TOP element.

Theorem 6.12. Let $\mathbb{D} = \bigcup_{i=0}^{k} X^i$ for $k \in \mathbb{N}$ and let the function $f:X \to \mathcal{B}^+$ be a representation that achieves Kraft storage. Consider the BOS pointer representation $\rho:\mathbb{D} \to \mathcal{B}^+$ defined by

$$\rho(d) = \bigcup_{i=1}^{|d|} \{f(d(i))\}_{n_{|d|+1-i}} \cup \{\ell(|d|)\}_{n},$$

where $n, n_j \in \mathbb{N}$ and where ℓ is any representation $\ell:\mathcal{J} \to \mathcal{B}^+$. Then any implementation of the stack operations will have the following access costs:

$$\begin{aligned}
\#[\alpha_{TOP}(\rho(d))] &\geq 1 \\
\#[\alpha_{PUSHx}(\rho(d))] &\geq |\ell(|d|)| + |d| + 1 \\
\#[\alpha_{POP}(\rho(d))] &\geq \begin{cases} |\ell(|d|)| + |d| & \text{if } |d| \neq 0 \\ 1 & \text{if } |d| = 0 \end{cases}
\end{aligned}$$

We do not formally prove this theorem because the proof is similar to arguments we have already made and because we can now already see that the stack operations would have higher access costs than we would in general want.

Note that the four stack representations we have discussed may all achieve Kraft storage, but their access costs differ greatly. We summarize in Table 6.3 some of the lower bounds we have determined in this chapter.

CHAPTER 7

QUEUES

The same framework that we have developed in this thesis can also be used to analyze queues. Although we shall not in this chapter prove any results, let us point out some of the complexities inherent in queues that are not present in stacks.

Recall that a queue differs from a stack in that items are inserted at one end and deleted from the other. If we want to achieve Kraft storage in a representation of a queue, we know that we can use only a single pointer. This, however, does not allow multiple representations and so updating operations will necessarily have high access costs. In all of the examples we consider in this chapter, we shall assume a problem domain alphabet $X = \{a,b,c,d\}$ and assume that $|X|$ is large enough so that a pointer always fits in a single cell in the cases we consider. We shall also assume that $a \in X$ is represented by 0 ($\in \mathbb{B}$), b by 1, c by 2, and d by 3.

Example 7.1. Suppose we have a three element queue. Consider implementing such a queue with a single pointer and holding the other end fixed.

a) Let the rear of the queue be fixed; i.e., all insertions are made to the same cell. Thus, the entire contents of the queue must be slid each time an insertion is made. On the other hand, a deletion requires only the deletion of the item from the queue. For instance, suppose our queue initially has three elements inserted: b, a, c. So $m(0) = 2$, $m(1) = 0$, and $m(2) = 1$:

2 0 1 _	Pointer to front: 3

If we DELETE an item we are left with

2 0 _ _	Pointer to front: 2

If we now INSERT(d), we obtain

3 2 0 _	Pointer to front: 3

Notice that each of the elements already on the queue had to be moved when we made an INSERT. With this scheme, a DELETE operation requires only a single

Table 6.3 Lower bounds on the numbers of accesses required to perform the stack operations $Q_{acc}(n(d))$.

CHAPTER 7

QUEUES

The same framework that we have developed in this thesis can also be used to analyze queues. Although we shall not in this chapter prove any results, let us point out some of the complexities inherent in queues that are not present in stacks.

Recall that a queue differs from a stack in that items are inserted at one end and deleted from the other. If we want to achieve Kraft storage in a representation of a queue, we know that we can use only a single pointer. This, however, does not allow multiple representations and so updating operations will necessarily have high access costs. In all of the examples we consider in this chapter, we shall assume a problem domain alphabet $X = \{a,b,c,d\}$ and assume that $|\mathcal{B}|$ is large enough so that a pointer always fits in a single cell in the cases we consider. We shall also assume that $a \in X$ is represented by $0 \in \mathcal{B}$, b by 1, c by 2, and d by 3.

Example 7.1. Suppose we have a three element queue. Consider implementing such a queue with a single pointer and holding the other end fixed.

a) Let the rear of the queue be fixed; i.e., all insertions are made to the same cell. Thus, the entire contents of the queue must be slid each time an insertion is made. On the other hand, we need only decrement the pointer to delete an item from the queue. For instance, suppose our queue initially has three elements inserted: b, a, c. So $m(0) = 2$, $m(1) = 0$, and $m(2) = 1$:

2 0 1 _ Pointer to front: 3

If we DELETE an item we are left with

2 0 _ _ Pointer to front: 2

If we now INSERT(d), we obtain

3 2 0 _ Pointer to front: 3

Notice that each of the elements already on the queue had to be moved when we made an INSERT. With this scheme, a DELETE operation requires only a single

access in order to decrement the pointer. An INSERT operation, however, requires $|d| + 1$ accesses, where $|d|$ is the initial queue size.

b) If the front of the queue were stationary, then it would be an insertion which would be easy to perform. As above, suppose we have intially inserted b, a, c on the queue:

1 0 2 _ Pointer to rear: 3

A DELETE operation requires moving the contents of each element in the queue:

0 2 _ _ Pointer to rear: 2

Now an INSERT(d) is simple:

0 2 3 _ Pointer to rear: 3

Using this second scheme, an INSERT operation requires two accesses, one to the pointer and one to insert the new element. On the other hand, the DELETE operation requires accessing every element in the queue (as well as the pointer), $|d| + 1$ accesses. ∎

The tradeoff in the preceding example suggests that we do not want to consider separately the access costs for the INSERT and DELETE operations; instead, we might want to consider the cost of a DELETE-INSERT pair of operations. In Example 7.1a we found that an INSERT had cost $|d| + 1$ and a DELETE had cost 1, a total cost of $|d| + 2$ accesses for the DELETE-INSERT pair. In Example 7.1b, INSERT had cost 2 and a DELETE had cost $|d| + 1$, a total cost of $|d| + 3$.

The expense involved in the INSERT or DELETE operation in Example 7.1 was due to the fact that we were forced to always maintain one end of the queue fixed. Of course, if we were to allow two pointers, then we would not have this problem. Instead, let us consider a scheme where we allow a queue to have one end in one of, say, two positions.

Example 7.2. Reconsider Example 7.1b but assume that the pointer is large enough that one bit can be reserved to indicate whether the "fixed" end of the queue is in cell 0 or in cell 1. Suppose our initial queue state is, as before:

1 0 2 _ Pointer to rear: 3 Front: 0

Now if we do a DELETE, we do not need to move any of the list elements:

_ 0 2 _ Pointer to rear: 3 Front: 1

An INSERT(d) operation gives:

_ 0 2 3 Pointer to rear: 4 Front: 1

Unfortunately, another DELETE will require moving the queue:

2 3 _ _ Pointer to rear: 2 Front: 0

Finally one more INSERT(a):

2 3 0 _ Pointer to rear: 3 Front: 0

This effectively brings us back to our initial state (although the actual queue elements differ). Notice that these four operations we performed required, in order, 1, 2, $|d| + 2$, and 2 accesses, where $|d|$ refers to the size of our initial queue state before the two pairs of DELETE-INSERT operations were performed. ∎

So in Example 7.2, by reserving one bit of the pointer to indicate the location of the front of the queue, we used a total of $|d| + 7$ accesses, only $\frac{|d| + 7}{2}$ accesses on the average for a DELETE-INSERT. On the other hand, without using this extra bit we in Example 7.1 were forced to make $|d| + 2$ accesses for a DELETE-INSERT. So we were able to not only delay the heavy cost of sliding the queue, but we in fact have decreased the average cost of a DELETE-INSERT pair. Let us use the same trick again and reserve two bits to tell us where the front of the queue is located; i.e., the front of the queue may be in any of cells 0, 1, 2, 3.

Example 7.3. Given an initial queue 2 1 0 3, let us perform a sequence of four DELETE-INSERT pairs of operations, keeping track of the numbers of accesses.

2 1 0 3 _ _ _

DELETE: _ 1 0 3 _ _ _ 1

INSERT(a):	_ 1 0 3 0 _ _	2
DELETE:	_ _ 0 3 0 _ _	1
INSERT(c):	_ _ 0 3 0 2 _	2
DELETE:	_ _ _ 3 0 2 _	1
INSERT(b):	_ _ _ 3 0 2 1	2
DELETE:	0 2 1 _ _ _ _	$\|d\| + 4$
INSERT(a):	0 2 1 0 _ _ _	2

This gives a total of $|d| + 15$ accesses, an average of $\frac{|d| + 15}{4}$ accesses per DELETE-INSERT pair. ∎

In general, if we reserve k bits of the pointer to indicate the location of one end of the queue, then there are 2^k possible representations of each queue, and a DELETE-INSERT pair requires, on the average,

$$\frac{|d| + 2^k + 3\cdot(2^k - 1) + 2}{2^k} = \frac{|d|}{2^k} + 4 - \frac{1}{2^k} = O(\frac{|d|}{2^k})$$

accesses.

Thus, we have seen that a one pointer scheme allows no multiple representations and we may achieve Kraft storage. Using a two pointer scheme, the queue could be located anywhere in memory (within the range of the pointers) and may, in fact, drift throughout memory. An intermediate scheme has a single pointer which has enough room for $|d|$ with one or more extra bits reserved to indicate the location of one end of the queue. In this latter case, we not only defer but actually save in our access cost. This illustrates not only a storage-access tradeoff but also a tradeoff with multiplicity of representation, and we have a nice continuum between the one and two pointer cases.

Suppose we do want to achieve Kraft storage and are using a single pointer. It is interesting to consider how many accesses are required in order to perform a DELETE-INSERT pair of operations. If the queue is always of a fixed size k (i.e., the only operations performed are DELETE-INSERT(a) pairs), then, somewhat surprisingly, it is possible to represent the queues in memory in such a way that the

average number of cells accessed is a constant independent of the length $|d|$. On the other hand, suppose we insist that the representation function ρ have the constraint that $\rho(d)$ is a permutation of d and that $d(i)$ always maps to the same memory cell(s), for all $0 \leq i \leq |d|$. Then it can, in fact, be shown that a DELETE-INSERT pair of operations performed on all queues of a fixed length k will have an average access cost of at least $(\frac{|\mathcal{B}| - 1}{|\mathcal{B}|}) \cdot k$. Thus, for most natural encoding schemes it will be necessary to access essentially $|d|$ cells.

CHAPTER 8

CONCLUSIONS

In this thesis we have explored what it means for a list to be information-theoretically optimal, in the sense that it achieves Kraft storage and Kraft access. We first examined the full set of table lookup questions and showed that if we are considering problem domains of the form $\mathbb{D} = \bigcup_{i \in J} X^i$, then it is possible to achieve both bounds simultaneously only for domains $\mathbb{D} = X^n$ and $\mathbb{D} = \{\lambda\} \cup X^n$. This corresponds to a notion of independence; essentially, it must be the case that no matter what the value $d(i) \in X$, then $d(i+1)$ might take on any value in X. If we were to determine $d(i) = \varnothing$ then it would have to be the case that $d(i+1) = \varnothing$ and we would not have independence. Of course, we did see that there is a perhaps surprising exception, namely, when $\mathbb{D} = \{\lambda\} \cup X^n$ and $|X| = 2$.

As a consequence of this work, we were able to show that it is never possible to achieve both Kraft storage and Kraft access for many common list representation schemes. The only exception was for a fixed size representation, when $\mathbb{D} = X^n$. Since we are here primarily interested in variable-length lists, it is clear that we will not be able achieve both.

We discussed four natural stack representation schemes: TOS endmarker, BOS endmarker, TOS pointer, and BOS pointer. We were able to obtain fairly tight lower bounds on access costs for performing POP, PUSHx, and TOP operations; those results are summarized in Table 6.3. It is shown that endmarker representations are necessarily expensive to update. On the constructive side, we developed a representation scheme for a TOS pointer that is storage optimal and does quite well for access. Assuming a monotomically nonincreasing probability distribution on stack lengths, we were able to obtain the following access costs:

$$\#[\alpha_{TOP}(\rho(d))] \leq 2 \cdot \lfloor \log_2(|d|+1) \rfloor + k_1 = O(\log|d|)$$
$$\#[\alpha_{PUSHx}(\rho(d))] \leq 2 \cdot \lfloor \log_2(|d|+1) \rfloor + k_2 = O(\log(|d|)$$
$$\text{avg } \#[\alpha_{POP}(\rho(d))] \leq k_3,$$

for $k_1, k_2, k_3 \in \mathbb{N}$. The bounds we obtained give an indication as to why pointer representations are so commonly used in the practical implementation of stacks.

In the discussion of stacks, we were forced to examine separately several classes of representations. It would be nice if there were some more general characterization that would allow us to make more general statements. For instance, is it possible for any implementation to perform both a PUSHx and a POP in a constant number of accesses.

The model that we used is capable of more generalization. For instance, instead of considering access costs for performing only a single operation, we might wish to perform a sequence of operations. Also, our definition of access or storage costs could be altered to correspond to the desired application; we might even be able to consider some sort of hierarchical memory structure.

There remains a great deal of work to be done. Perhaps the most obvious is the need to apply the techniques used in this thesis in order to examine other types of lists. We briefly discussed queues, but it is clear that queues raise a lot of issues that were not present with stacks. The flavor of some preliminary results was indicated in that chapter. It appears that dequeues are a straightforward extension of queues, but there remain many other types of lists to be explored. In addition, it would be interesting to know whether similar arguments could be applied to trees. Some of the techniques discussed may also be useful in the analysis of hashing tables.

BIBLIOGRAPHY

[1] A. Aho, J. Hopcroft, and J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Co, Reading, MA, 1974.

[2] N. Abramson, Information Theory and Coding, McGraw-Hill Publishing Co., New York, 1963.

[3] D. J. Brown and P. Elias, "Complexity of Acceptors for Prefix Codes", *IEEE Transactions on Information Theory*, vol. 21, pp. 357-359, May 1976.

[4] G. J. Chaiten, "A Theory of Program Size Formally Identical to Information Theory", *Journal of the Association for Computing Machinery*, vol. 22, pp. 329-340, 1975.

[5] P. Elias, "Efficient Storage and Retrieval by Content and Address of Static Files", *Journal of the Association for Computing Machinery*, vol. 21, no. 2, pp. 246-260, April 1974.

[6] P. Elias, "Minimum Times and Memories Needed to Compute the Values of a Function", *Journal of Computer and System Sciences*, vol. 9, no. 2, pp. 196-212, Oct. 1974.

[7] P. Elias, "Universal Codeword Sets and Representations of the Integers", *IEEE Transactions on Information Theory*, vol. IT-21, no. 2, pp. 194-203, March 1975.

[8] P. Elias, "Distinguishable Codeword Sets for Shared Memory", *IEEE Transactions on Information Theory*, vol. IT-21, no. 4, pp. 392-399, July 1975.

[9] P. Elias, "An Information Theoretic Approach to Computational Complexity", *Colloquia Mathematica Societas Janos Bolyai*, no. 16, pp. 171-198, 1977.

[10] P. Elias and R. A. Flower, "The Complexity of Some Simple Retrieval Problems", *Journal of the Association for Computing Machinery*, vol. 22, no. 3, July 1975.

[11] R. A. Flower, "An Analysis of Optimal Retrieval Systems with Updates", Ph.D. Thesis, Dept. of Electrical Engineering, M.I.T., Cambridge, MA, 1974.

[12] R. Gallager, Information Theory and Reliable Communication, John Wiley and Sons, Inc., New York, 1968.

[13] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proceedings IRE*, vol. 40, no. 10, pp. 1098-1101, Sept. 1952.

[14] D. E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, MA, 1969.

[15] A. N. Kolmogorov, "Three approaches to the quantitative definition of information", (Russian) *Problems of Information Transmission*, vol. 1, pp. 1-7, 1965.

[16] L. G. Kraft, "A Device for Quantizing, Grouping, and Coding Amplitude Modulated Pulses", M.S. Thesis, Dept. of Electrical Engineering, M.I.T., Cambridge, MA, 1949.

[17] D. J. Lehman and Michael B. Smyth, "Data Types" (Extended Abstract), *18th Annual Symposium on Foundations of Computer Science*, pp. 7-12, 1977.

[18] B. Liskov and S. Zilles, "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 1, pp. 7-19, March 1975.

[19] M. Minsky and S. Papert, Perceptrons, M.I.T. Press, Cambridge, MA, (pp. 215-226), 1969.

[20] R. L. Rivest, "Analysis of Associative Retrieval Algorithms", Ph.D. Thesis, Computer Science Department, Stanford University, Stanford, CA, 1974.

[21] R. L. Rivest, "On Hash-Coding Algorithms for Partial-Match Retrieval", *Proc. 1974 Switching and Automata Theory Conference*, pp. 95-103, 1974.

[22] M. K. Warner, "An Analysis of Storage, Retrieval, and Update Costs for Data Bases Which Are Tables of Entries", M.S. Thesis, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, 1978.

[23] T. Welch, "Bounds on Information Retrieval Efficiency in Static File Structures", MAC TR-84, Project MAC, Computer Science, M.I.T., Cambridge, MA, 1971.

Donna Jean Brown was born in Lansing, Michigan on July 28, 1949. She graduated in 1967 from Mount Greylock Regional High School in Williamstown, Massachusetts, and in 1971 received her B.A. from Wellesley College, where she was a Pendleton Scholar. She began graduate work in the department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology and was promoted to an Instructor in 1974. She received there the degrees of S.M. in 1974, E.E. in 1976, and Ph.D. in 1978. Beginning in August 1978, she will be an Assistant Professor of Electrical Engineering at the University of Illinois at Urbana-Champaign.

Donna Brown is a member of Sigma Xi, the Association for Computing Machinery (SIGACT), and the Association for Women in Mathematics. Her interests include music, backpacking, swimming, cooking, and traveling.

BIOGRAPHICAL NOTE

Donna Jean Brown was born in Lansing, Michigan on July 28, 1949. She graduated in 1967 from Mount Greylock Regional High School in Williamstown, Massachusetts, and in 1971 recieved her B.A. from Wellesley College, where she was a Pendleton Scholar. She began graduate work in the department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology and was promoted to an Instructor in 1974. She received there the degrees of S.M. in 1974, E.E. in 1976, and Ph.D. in 1978. Beginning in August 1978, she will be an Assistant Professor of Electrical Engineering at the University of Illinois at Urbana-Champaign.

Donna Brown is a member of Sigma Xi, the Association for Computing Machinery (SIGACT), and the Association for Women in Mathematics. Her interests include music, backpacking, swimming, cooking, and travelling.

CS-TR Scanning Project
Document Control Form

Date : 4/4/96

Report # LCS-TR-217

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 210(214-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): FOLLOW TITLE & ABSTACT PAGES

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1-210) UNNUMBERED TITLE PAGE, UNNUMBERED BLANK, 2, UNNUMBERED BLANK, 3-208	
(211-214) SCANCONTROL TARGETS (3)	

Scanning Agent Signoff:

Date Received: 4/4/96 Date Scanned: 4/4/96 Date Returned: 4/11/96

Scanning Agent Signature: Michael W. Cobb

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**