August 1995

UILU-ENG-95-2223
ACT-135

Applied Computation Theory

Implementing a Program Checker for Linked Lists

D. Deavours

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Field	Entry
1a. REPORT SECURITY CLASSIFICATION	Unclassified
1b. RESTRICTIVE MARKINGS	None
2a. SECURITY CLASSIFICATION AUTHORITY	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
3. DISTRIBUTION / AVAILABILITY OF REPORT	Approved for public release; distribution unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	
5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION	Coordinated Science Lab University of Illinois
6b. OFFICE SYMBOL (If applicable)	N/A
6c. ADDRESS (City, State, and ZIP Code)	1308 W. Main St. Urbana, IL 61801
7a. NAME OF MONITORING ORGANIZATION	
7b. ADDRESS (City, State, and ZIP Code)	
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	
8b. OFFICE SYMBOL (If applicable)	
8c. ADDRESS (City, State, and ZIP Code)	
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
10. SOURCE OF FUNDING NUMBERS	PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification)	Implementing a Program Checker for Linked Lists
12. PERSONAL AUTHOR(S)	D. Deavours
13a. TYPE OF REPORT	Technical
13b. TIME COVERED	FROM ________ TO ______
14. DATE OF REPORT (Year, Month, Day)	August 3, 1995
15. PAGE COUNT	33
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES	FIELD / GROUP / SUB-GROUP
18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	multiplication algorthms, program checking, program verification, implementation, software fault-tolerance, linked lists, hash functions,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A program checker verifies that for a certain input, the corresponding output of a program is correct. We present implementations of some probabilistic program checkers and examine how well they work. First, we discuss an implementation of a probabilistic checker for sorting that uses epsilon-biased hash functions. We show how these hash functions may be computed with efficient methods for multiplying large integers. We apply the hash functions to finally present a probabilistic on-line checker for linked lists and prove its correctness.

Field	Entry
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS
21. ABSTRACT SECURITY CLASSIFICATION	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	
22b. TELEPHONE (Include Area Code)	
22c. OFFICE SYMBOL	

Implementing a Program Checker for Linked Lists

D. Deavours*

August 3, 1995

Abstract

A program checker verifies that for a certain input, the corresponding output of a program is correct. We present implementations of some probabilistic program checkers and examine how well they work. First, we discuss an implementation of a probabilistic checker for sorting that uses epsilon-biased hash functions. We show how these hash functions may be computed with efficient methods for multiplying large integers. We apply the hash functions to finally present a probabilistic on-line checker for linked lists and prove its correctness.

1 Introduction to Program Checking

Because people increasingly rely on computers for critical applications, it is important for the output of a program to be correct. The purpose of a program checker is to verify that the output of a program is correct. To illustrate this concept, we introduce a model for program checking (see Figure 1). In the program checking model, the user may be a program, which uses the facilities of the resource manager as a black box, such as an abstract data type. In order to check that the values returned by the resource manager are correct, the user always makes calls to the checker, which in turn makes calls to the resource manager. If the resource manager returns an incorrect value, then the checker declares an error; otherwise, the checker returns the value returned by the resource manager. The checker should be transparent to the user, except when an error occurs. Furthermore, the design of the checker should be independent of the design of the resource manager.

The checker is allowed a small, reliable workspace. The resource manager may use a large workspace, but the workspace is not guaranteed to be reliable. A program checker may be classified into several categories, depending on its characteristics. A *deterministic* checker always declares an error whenever an error has occured. A *probabilistic* checker declares an error with a certain probability whenever an error has occured. The probabilistic checker is useful if the checker is efficient and the probability is high. An *on-line* checker declares an error immediately after the error occurs, and an *off-line* checker declares an error sometime after one has occured.

The resource manager provides the procedures for performing the operations requested by the user. It contains procedures for manipulating data structures, and it may have a

*Supported by the National Science Foundation under Grant CCR-9315696.

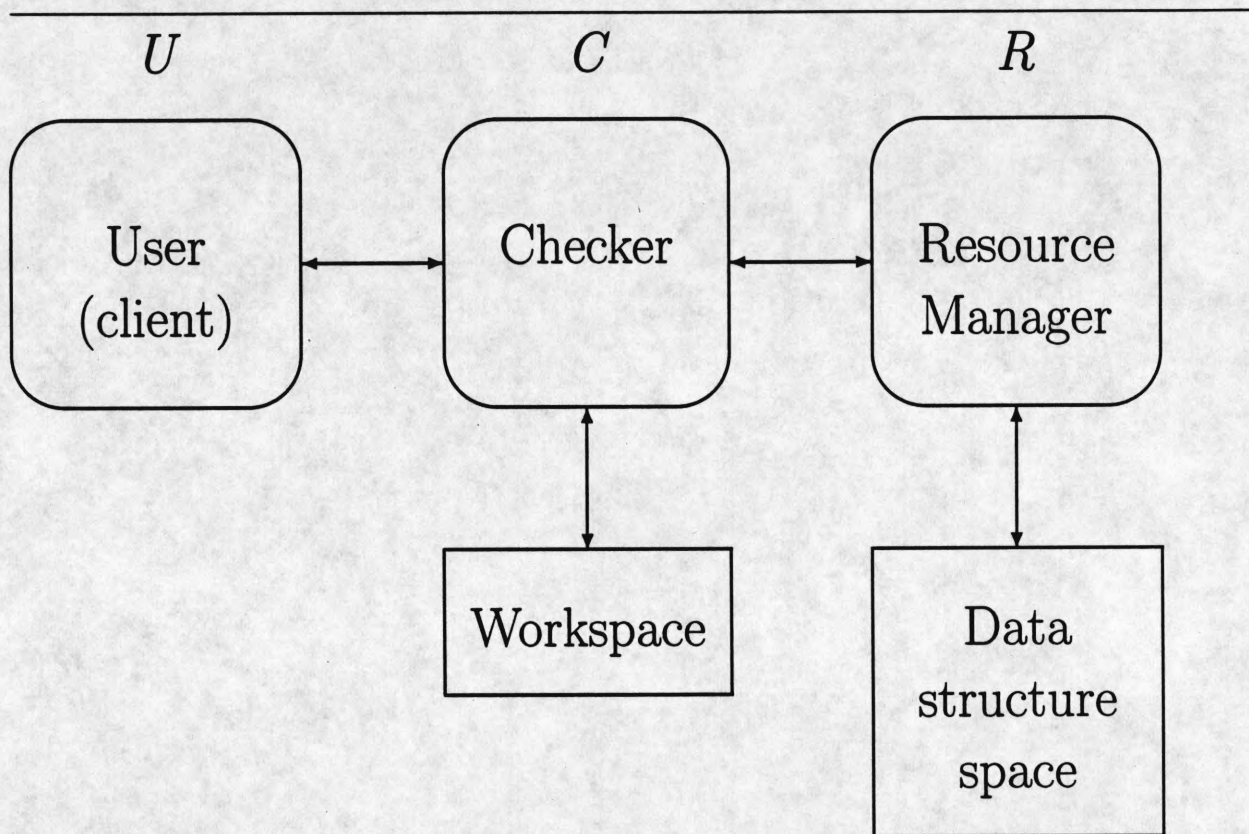

Figure 1: Program Checker Model.

large, unreliable workspace. From the perspective of the checker, the resource manager may be viewed as a black box. The checker may consider the resource manager as an adversary that tries to sneak errors past the checker.

In this paper, we present an implementation of a probabilistic on-line linked list checker. We first introduce hash functions in Section 2 as a tool in solving the set equality problem in Section 3. In Section 4, we discuss the efficient multiplication of large numbers, which is necessary for implementing the hash functions. We then apply these tools to check the Unix `sort` command in Section 5. Finally, in Section 6, we explain how to implement the linked list checker.

2 Hash Functions

Fix n and k. Let $A = \{0, \ldots, 2^n - 1\}, B = \{0, \ldots, 2^k - 1\}$. Consider $h : A \rightarrow B$. If $n > k$, then h maps more than one element in A to the same element in B. Our goal in constructing a hash function h is to minimize the probability that two elements in A are mapped to the same element in B. We can state more precisely that given $x \neq y$, for a randomly chosen hash function h, ideally $h(x) = h(y)$ with probability $1/2^k$. We shall construct h from a family of simpler functions. Let $\mathcal{F}$ be a collection of functions $f : A \rightarrow \{0, 1\}$ so that given $x \neq y$, for randomly chosen function f, $f(x) = f(y)$ with probability 1/2. By choosing k functions from $\mathcal{F}$ independently and randomly without replacement, we can define $h(x) = \sum_{i=0}^{k-1} f_i(x)2^i$.

We can construct a function f by first generating an array of 2^n independently random bits. The value of $f(x)$ is the value of the xth bit in the array. The result is that given $x \neq y$, for a random choice of f, $f(x) = f(y)$ with probability 1/2. The drawback to using this method is that each function f requires 2^n bits of memory.

In order to use less memory, we may define a new function f' which constructs a bit for $f'(x)$ by using fewer than 2^n random bit variables. The result is that when $x \neq y$, then $f'(x) = f'(y)$ with probability less than $1 - \beta$. The construction of f' which we will examine has $\beta = 1/8$, and requires only $O(n)$ memory for each f'. To construct h, we must use more f' functions than f functions. The number of f' functions needed comes from solving the equation $(1/2)^k = (1/8)^m$ for m. In this case, $m \approx 5.2k$, so for $h : A \rightarrow \{0, \ldots, 2^m - 1\}$, if $x \neq y$, then $h(x) = h(y)$ with probability $1/2^k$.

The construction of f' can be conceptualized as generating a random bit vector r which is 2^n bits long. At any instant, only one bit of r is needed, so we will show how to construct this bit. We denote $r[x]$ to be the bit vector of r in the x-th dimension. Naor and Naor [1] suggest the following method for constructing $r[x]$:

- At the start of the program, assign independent random values uniformly distributed in A to the variables $a_0, a_1, b_0, b_1, b_2, b_3, b_4, b_5, b_6$, and c.

- Each time we wish to compute $r[x]$, we execute the procedure given in Figure 2.

  Throughout this paper, we use the following font conventions for programs:

`typewriter` font is used for checker procedures, linked list procedures, and class names. `Read` is the checker procedure, while `read` is the linked list procedure.

```
declare array of words a_0, a_1, b_0, b_1, b_2, b_3, b_4, b_5, b_6, c
(* These are assigned random values previous to calling hash. *)

procedure hash(array of words x)
declare array of words s
declare word j, i
declare bit r
s := ((((((b_6 * x + b_5) * x + b_4) * x + b_3) * x + b_2) * x + b_1) * x + b_0)
if (parity(s) = 1) then
    return (0)
else
    s := a_1 * x + a_0
    j := log n - highbit(s)              (* Log base 2 *)
    r := 0
    for i := 0 to j - 1 do
        r := r ⊕ c[i]
        (* The operation ⊕ is bitwise XOR *)
        (* c[i] is the i'th significant bit in c *)
    return r
endif
```

Figure 2: Implementation of Hash Function f'.

italics are used for other procedures and variable names.

bold is used for key words, declarations, and flow of control.

(* comment *) Comments are enclosed between (* and *).

The declaration type *word* is the natural word size of the computer. When a variable is larger than the natural word size of the computer, it is represented by an array of words. The function $highbit(x)$ is $\max\{i|x_i = 1\}$, where x_i is the i-th bit position in the binary representation of x.

The motivation for and correctness of this construction are given by Naor and Naor [1]. An implementation is given in Figures 2 and 3.

The complexity of computing a single bit $r[x]$ is largely determined by the parameter n. The time required to compute a bit is $O(1)$ for n–bit sized words. However, n may be much larger than the word size of a computer. The real computational cost is proportional to multiplying words of n–bits. An analysis of multiplying long words is presented in Section 4.

The exact amount of memory required to compute the bit $r[x]$ is $9n$ bits. One method to create m such r vectors is to use $9mn$ random bits. Another method, suggested by Naor and Naor, uses expander graphs. However, the explicit construction of an expander graph is prohibitively complex [3], and so we did not implement it.

```
procedure Hash(array of words x, array of words y)
(* x is the element to hash into the set *)
(* y is the hashed value which we update, passed by reference *)
declare integer i
for i := 0 to size of y do
    y_i := y_i ⊕ hash(x)
    (* The operator ⊕ is bitwise XOR *)
    (* y_i is ith bit in y *)
    (* hash is the hash procedure in figure 2 *)
endfor
```

Figure 3: Implementation of Hash Function h.

3 Set Equality

Consider the finite sets I and D, both containing only distinct elements. We say that I is equal to D if they contain the same elements, that is, $I \subseteq D$ and $D \subseteq I$.

One way to solve the set equality problem is to create a large hash table. Then for each element in I, insert the element into the hash table while keeping track of the number of elements in I. Next, for each element in D, delete the element from the hash table. If we attempt to delete an element which which is not in the hash table, or the size of I is not the size of D, then the two sets are not equal. Otherwise, $I = D$. If $n = |I|$, then this method uses $\Omega(n)$ memory and takes $O(n)$ time to compute.

A probabilistic method using hash functions shown in Figure 4 may also be used to solve this problem. Let $h : A \to \{0, \ldots, 2^{5.2k} - 1\}$ be a hash function described in Section 2. The pseudo code given in Figure 4 shows how this hash function can be used. The parameter k passed to the procedure *SetEq* is the confidence parameter, so that if the two sets are not equal, then the procedure returns *Equal* with probability less than $1/2^k$. Let m be the maximum size of an element in bits. This procedure runs in $O(n)$ time for a word size of $5.2k$ bits, but uses only $O(km)$ bits of memory. We show how the running time changes with a word size smaller than $5.2k$ in Section 4.

The insertion and deletion procedure does the same thing, namely invoking $Hash(e, H)$, because $Hash$ is an involution because of $\oplus$.

4 Multiplying Large Numbers

Multiplying large numbers is a problem computers are often required to do, so finding an efficient method is important. One application of multiplying large numbers is in generating the hash function described in Section 2. Several different methods are known for accomplishing this. The Schönhage and Strassen algorithm can multiply m–word numbers in $O(m \log m)$ time on a random access machine [4]. Another simpler method, though asymptotically less

```
declare procedure initialize(array of words x, word i)
(* Initializes each word in x to i *)

procedure SetEq(set I, D, word k)
declare array of words H[5.2 × k/wordsize]
(* k is the confidence parameter desired *)
declare element e
declare word count
initialize(H, 0)
foreach e ∈ I do
    Hash(e, H)
    (* Element e may be larger than one machine word, so cast Element into array of words *)
    (* The procedure Hash is defined in Figure 3 *)
    count := count + 1
endfor
foreach e ∈ D do
    Hash(e, H)
    count := count − 1
endfor
if (count = 0 and H = 0) then
    return Equal
else
    return NotEqual
endif
```

Figure 4: Probabilistic Set Equality.

efficient, takes advantage of some properties of multiplication, which makes it solvable in $O(m^{1+\epsilon})$ time, where ϵ may be arbitrarily small.

In this study of program checking, values of m stay relatively small, but are still large enough that using an efficient method is important. We examine two methods here: a straightforward method using $O(m^2)$ time, and a recursive method which uses $O(m^{1.585})$ time.

The first method is similar to one we are accustomed to when multiplying multi–digit numbers by hand. Example code is given in Figure 5. This procedure requires $O(m^2)$ words of intermediate memory. A modification to this method uses less intermediate memory, and example code for this is given in Figure 6. Each multiplication and addition operation is performed in a single step using this method, and this modified method requires only a small amount of intermediate memory ($O(1)$). We will call this Method 1. This improved method still requires $O(m^2)$ time.

The second method uses recursion and some algebraic properties. First, we introduce the notation $\langle x_1, x_0 \rangle = x_1 2^{w/2} + x_0$ if x_0 is $w/2$ bits long. For an example, we wish to multiply $\langle a_1, a_0 \rangle$ by $\langle b_1, b_0 \rangle$, with with a_1, a_0, b_1, and b_0 each $w/2$ bits long. The number of bits w must be even. First, we compute some intermediate values.

$p := a_1 b_1$
$q := a_0 b_0$
$t := a_1 + a_0$
$u := b_1 + b_0$
$r := tu$

Then, $\langle a_1, a_0 \rangle \langle b_1, b_0 \rangle = p2^w + (r - q - p)2^{w/2} + q$. We call this Method 2.

Notice that there are only three multiplications of $(w/2)$–bit numbers in this step. Therefore, two numbers represented by m words can be multiplied recursively using three multiplications of $m/2$–word numbers. It follows that an m–word multiplication may be computed in $m^{\log_2 3}$, or $m^{1.585}$ word multiplications. Example code for this is given in Figure 7. To ensure that the recursive multiplications are limited to $m/2$ words in length, the computation of $(a_1 + a_0)$ is done in a temporary memory location of size $m/2$–words, which leaves the possibility of an overflow. To compensate, we keep two carry bits (c_1 and c_2) and perform further additions whenever necessary.

The caller passes to *Multiply* two m–word arrays representing the two numbers to be multiplied, and one $2m$–word array *res* to place the product of x and y. Since they don't overlap, p and q may be computed and written directly to the product array *res*. As a result, the memory used in one step is by t, u, r, and the memory required to multiply two $m/2$ word sized numbers. The variables t and u are $m/2$ words long, and r is m words long, so the memory used by this algorithm ($M(m)$) to multiply two m–word numbers is

$$\begin{aligned} M(m) &= m/2 + m/2 + m + M(m/2) + O(1) \\ &= 4m + O(1). \end{aligned}$$

Since Method 2 requires a considerable amount of intermediate memory allocations and many $O(m)$ operations, the overhead involved in multiplying small numbers may be sub-

procedure *Multiply*(array of words x, y, res, word m)
(* x and y are factors, res is the product *)
(* m is the size of the x and y, and res is 2-m words *)
declare word i, j, c, d
(* i and j are index variables, c and d are temporary carry variables *)
declare array of words $q[0 \dots m-1][0 \dots 2m-1]$
(* q stores the intermediate values in the computation *)
$initialize(q, 0)$
$initialize(res, 0)$
(* Do all multiplications and store in intermediate memory *)
for $i := 0$ **to** $m - 1$ **do**
    $c := 0$ (* c is the carry to the next significant word *)
    for $j := 0$ **to** $m - 1$ **do**
        $q[i][i+j]$:= low half of $(x[i] \times y[j] + c)$
        c := high half of $(x[i] \times y[j] + c)$
    endfor
    $q[i][i+j+1] := c$
endfor
(* Add up all intermediate values and put result in res *)
for $i := 0$ **to** $m - 1$ **do**
    $c := 0$
    for $j := 0$ **to** $2m - 1$ **do**
        d := high half of $(res[i] + q[i][j] + c)$
        $res[i]$:= low half of $(res[i] + q[i][j] + c)$
        $c := d$
    endfor
endfor

Figure 5: Naive method, res = x × y.

```
procedure Multiply(array of words x, y, res, word m)
declare word i, j, c, d
initialize(res, 0)
for j := 0 to m − 1 do
    c := 0                              (* C is the carry to the next significant word *)
    for i := 0 to m − 1 do
        d := high half of (x[i] × y[j] + res[i + j] + c)
        res[i + j] := low half of (x[i] × y[j] + res[i + j] + c)
        c := d
    endfor
    res[j + m] := c
endfor
```

Figure 6: Method 1.

stantial. In order to determine whether Method 1 or Method 2 is better to use for a certain range of lengths, we implemented these methods and took performance measurements.

This experiment was done on a SPARCstation IPC, compiled using the GNU C compiler with a word size of 32 bits. The results are given in Table 1 and plotted in Figure 8. A moderate amount of effort was used to make the procedures efficient.

The results are interesting, but can be misleading. The numbers reflect both the quality of the algorithm and the particular implementation. For example, when the input is two words long, Method 1 is faster even though it is the base case of Method 2. Method 2 could be changed to optimize the base case so it would be faster. One of the first versions read 32 bits at a time from the array and then did 16 bit manipulations on it. Subsequent versions read 16 bits at a time for simplicity, and the results were a substantial (constant) loss in performance. Also, a check for multiplication by zero may or may not yield a performance gain, depending on the frequency of zeros.

Basically, the programs follow theoretical results quite closely. As the input size doubles, the time to compute the result triples (Method 2) or quadruples (Method 1.) Method 1 has time complexity $O(m^2)$, and the time can be closely approximated as $t = (2.764\mu s) \times m^2$, where m is the length in words. The time taken by Method 2 can also be closely approximated by $t = (16\mu s) \times m^{1.585}$.

For our purposes, this experiment yielded valuable results. For 32 to 512 bit numbers, Method 1 provides better performance, and for numbers longer than 512 bits, Method 2 is faster.

```
declare procedure MultiwordAdd(array of words x, y, res, word size)
(* res := x + y. Return carry bit (0 or 1) *)
declare procedure MultiwordSub(array of words x, y, res, word size)
(* res := x - y *)

procedure Multiply(array of words x, y, res, word length)
(* length must be power of 2 *)
(* res is passed by reference *)
if (length = 1) then
    res[0] := x × y/BASE
    res[1] := x × y mod BASE
else
    declare array of words r[length + 1], t[length/2], u[length/2]
    declare word halflen, c1, c2
    halflen := length/2
    Multiply(x, y, res, halflen)
    Multiply(x + halflen, y + halflen, res + length, halflen)
    c1 := MultiwordAdd(x, x + halflen, t, halflen)
    c2 := MultiwordAdd(y, y + halflen, u, halflen)
    Multiply(t, u, r, halflen)
    if (c1 = 1 and c2 = 0) then
        c1 := MultiwordAdd(r + halflen, u, r + halflen, halflen)
        r[length] := c1
    else if (c1 = 0 and c2 = 1) then
        c2 := MultiwordAdd(r + halflen, t, r + halflen, halflen)
        r[length] := c2
    else if(c1 = 1 and c2 = 1) then
        c1 := MultiwordAdd(r + halflen, t, r + halflen, halflen)
        c2 := MultiwordAdd(r + halflen, u, r + halflen, halflen)
        r[length] := c1 + c2 + 1
    endif
    MultiwordSub(r, res, length)
    MultiwordSub(r, res + length, length)
    c1 := MultiwordAdd(res + halflen, r, res + halflen, length)
    res[length + halflen] := res[length + halflen] + c1
endif
```

Figure 7: Method 2.

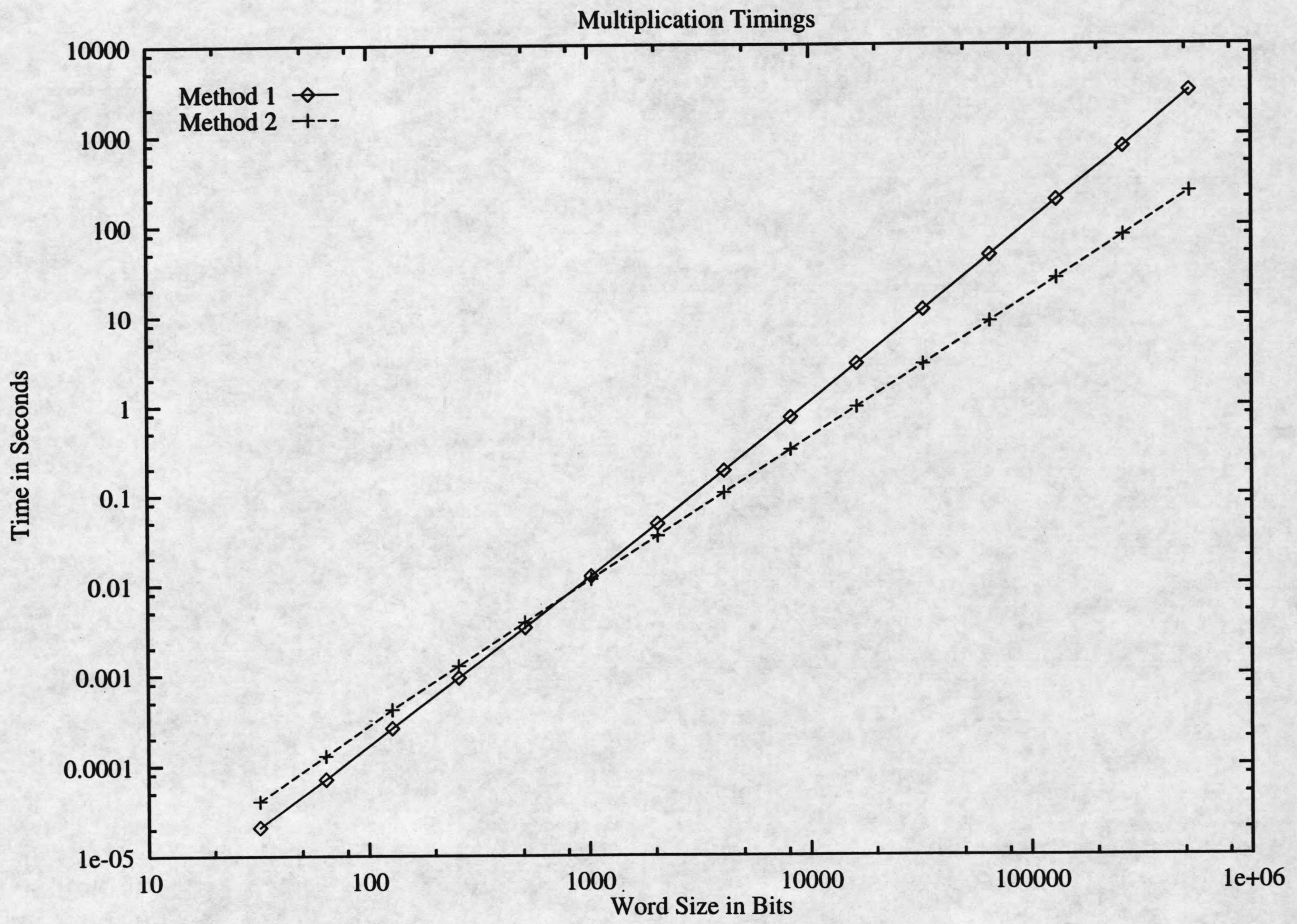

Figure 8: Time plots for methods 1 and 2.

Length (bits)	Method 1	Method 2
32	20.91 μs	40.64 μs
64	70.71 μs	127.8 μs
128	259.1 μs	414.0 μs
256	932.7 μs	1.256 ms
512	3.346 ms	3.804 ms
1024	12.34 ms	11.41 ms
2048	47.14 ms	34.61 ms
4096	184.0 ms	104.8 ms
8192	727.0 ms	316.9 ms
16384	2.891 s	948.6 ms
32768	11.52 s	2.849 s
65536	46.38 s	8.548 s
131072	187.6 s	25.68 s
262144	733.7 s	77.13 s
524288	3036.6 s	232.0 s

Table 1: Methods 1 and 2.

5 Checking the Unix sort program

Many sorting programs use algorithms which run in $O(n \log n)$ time, where n is the number of elements to sort. To determine that the output of a sorting program is correct, a checker verifies that the output is in sorted order and that the input and output sets are equal. Determining that the output is in sorted order is a simple problem that can be solved in $O(n)$ time with $O(1)$ memory. Our checker determines whether the input and output sets are equal by using the probabilistic technique described in Section 3. Using this method, the `sort` program can be checked in $O(kn)$ time using $O(km)$ memory, where m is the maximum size of an element and k is the confidence parameter. This checker detects errors with probability $1 - 1/2^k$.

Our Unix `sort` checker is simply an application of the set equality algorithm discussed in Section 3 with an additional check that the output is in sorted order. It is important to note that it is common for text files to have identical lines. For example, all blank lines are identical. In order for the elements of the sets (which are text lines) to be distinct, we append a line number to each line before it is hashed and sorted, and then strip the line numbers before printing sorted text.

The program checker works as follows:

1. Open files and pipes, and fork a subprocess to run `sort`.

2. Read in the input, append a line number to each line, insert the line into I set, and pipe lines to `sort`. Continue until end–of–file.

3. Read in each line of the output of `sort`, insert the line into D set, determine that the output is sorted, strip the line numbers, and send line to standard out. Continue until end–of–file.

4. Determine whether $I = D$.

For n input lines, `sort` runs in $O(n \log n)$ time, and the checker runs in $O(kn)$ time. With a fixed k and a sufficiently large n, the checker should run faster than `sort`. For practical sizes of n, however, this is not the case. The constants for `sort` are much smaller than the constants for the checker. With the input lines bounded to 118 characters, the checker can check approximately $350/k$ lines per minute. For $n < 100,000$, `sort` can sort 7500 lines per second. With $k = 1$, the checker will run as fast as `sort` with $n \approx 10^{700,000}$.

The primary benefit of using this checker is that it uses a small amount of memory ($5.2k \log m$ bits). Any `sort` program can be checked using this method, and the output can be checked within any desired probability. The drawback is that the constants of operation are so large that it is impractical to use in many cases. If a `sort` program needs to be checked efficiently, using a deterministic checker, while using $O(n)$ memory, may be more practical since most `sort` programs use $O(n)$ intermediate memory as well.

6 On-line Linked List Checker

6.1 Introduction

The linked list is an important fundamental data structure. Since implementations of linked lists are often simple, it is possible to formally verify the correctness of a linked list program. Why, then, implement a program checker for linked lists?

There are several reasons for designing and implementing a linked list checker. Formal verification, though a powerful academic tool, can not detect compiler bugs, operating system bugs, or hardware faults. Program checkers provide a mechanism for detecting errors wherever they occur, and on-line checkers catch any errors immediately. Whenever data reliability is paramount, a program checker assures that errors are detected.

Though linked lists are inefficient for many applications, they are simple. It is this simplicity which allows us to easily study, implement, and present a program checker. The same principles demonstrated here may be applied to more complex and useful data structures.

To implement the program checker efficiently, we use a probabilistic checker, utilizing the tools explained earlier in this paper. These ideas are based on a paper of Amato and Loui [2].

6.2 Linked List Checker Overview

In this section, we define the procedures used in the linked list checker and the restrictions that apply to them. In Section 6.7, we discuss how the checker's data structures are implemented and explain how each procedure modifies the data structures.

Each element of the linked list L contains the following fields:

e.id = unique identifier
e.sid = identifier of the successor of e
e.data = data field

The e.id field may contain the address of the element plus some unique information so the address may be efficiently extracted from the id and yet be unique.

The resource manager provides the following procedures:

`head()`:	returns a copy of the head element
`next`(e):	returns a copy of the successor of e
`read`(e):	returns a copy of the element with identifier e.id
`write`(e):	copies element e to the element in the list with identifier e.id
`insert`($pred, e$):	inserts element e into list after *pred*
`delete`($pred, e$):	deletes e, successor of *pred*, from list

These procedures may be called by the checker.

The checker provides the user with the same procedures, which we denote `Next`, `Head`, `Read`, `Write`, `Insert`, and `Delete`. When the user calls one of these procedures (which are implementations of user operations), the checker may call procedures in the resource manager. If the resource manager makes an error, then with high probability the checker detects and declares the error. The checker also provides the procedure `Deactivate`(e), which is explained here.

To implement a program checker efficiently, the checker must introduce some restrictions. We allow the user to invoke linked list procedures only on *active* elements. This means that every element passed as a parameter must be active, except for e in the `Insert`($pred, e$) procedure. The head element is always considered active. The element returned by `Next`(e) is considered active. An element remains active until either e is deleted or `Deactivate`(e) is called. No other operation changes the activity of an element. The reason for restricting user operations to active elements is discussed in Section 6.3.1.

6.3 Internal Structures

In the program checking model, the checker has a small, reliable memory. Our linked list checker keeps several structures in its reliable memory. First, the checker keeps a copy of the head element of the list. The checker also keeps a list of (hashed) sets for each active element, as discussed in Section 6.3.1.

6.3.1 Logarithmic Subdivisions

The checker keeps a *logarithmic subdivision* (LSD) for each active element. An LSD is simply a collection of sets. Each set contains elements which are copies of contiguous elements in the linked list. Each element in the linked list L is duplicated in exactly one set. The sets are ordered $S_0, S_1, S_2, \ldots, S_n$. In general, S_{i+1} contains twice as many elements as S_i, and S_0 always contains one element.

Consider the case with one active element e_0. We will label subsequent elements in the L as $e_1, e_2, \ldots$. Each $|S_i| = 2^i$, except the last set, which may contain fewer elements. Therefore

$$S_i = \{e_{2^i-1}, \ldots, e_{2^{i+1}-2}\}.$$

If more than one element is active, then the LSD includes all elements starting from the active element and extending to (but not including) the next active element.

Let a be the maximum number of active elements in the list, d be the total number of elements in the list, and m be the maximum size of an element. There are at most $\lceil \log d \rceil$ sets per LSD, and $O(a \log d)$ sets per list. If a becomes large, for example, if a is proportional to d, then the number of sets becomes $O(d)$. This is why a must be small relative to d. Since each hashed set takes $O(km)$ where k is the confidence factor and m is the maximum element size, and the checker keeps $O(a \log d)$ sets, the total amount of space the checker uses is $O(akm \log d)$.

6.3.2 Relaxed Logarithmic Subdivisions

To maintain a rigid $1, 2, 4, 8, \ldots$ structure for the size of sets in a LSD is computationally expensive: each insertion or deletion requires $O(d)$ work. To avoid this, we relax the restriction on the size of the sets. We require

$$2^i \leq |S_i| \leq 3 \times 2^i.$$

We call an LSD observing these restrictions a *relaxed logarithmic subdivision* (RLSD). If S_i gets too large, then the checker moves 2^i elements to S_{i+1}, or if S_i gets too small, then the checker moves 2^i elements from S_{i+1} to S_i. The checker moves elements so that elements in the set are contiguous elements in the list. We also require that $|S_0| = 1$ always.

For a set S_i, moving elements can occur at most once every 2^i user operations. This is easy to see because after moving elements from one set to another, $|S_i| = 2 \times 2^i$. It takes 2^i insertions or deletions to make the set too large or too small again. Since the checker does $O(2^i)$ work at most once in every 2^i user operations for each of $a \log d$ sets, the checker averages $O(a \log d)$ work per `Insert` or `Delete` operation. The number of sets in an RLSD is at most one more than the number of sets in an LSD, so memory usage is approximately the same.

6.4 Checker Procedures

The checker maintains several invariants. First, the checker always keeps hashed sets, which include elements in L as they were last written. Second, the checker reads only elements from L that comprise an entire set so the checker can compare the set of elements read from L with the (hashed) set kept by the checker for equality. The checker keeps a copy of the head element of L and maintains an RLSD for each active element as described in Section 6.3.2. Finally, whenever a new element is inserted into L, the checker gives the element a unique identifier `id`. To simplify computation, the `id` may be the address of the element in memory concatenated with a unique number. In this section, we explain how each user procedure modifies the checker's data structures.

For the procedure `Read`(e), we insist that e be an active element. Let S_0 be the first set in the RLSD for e, so $S_0 = \{e\}$. In the `Read`(e) procedure, the checker does the following:

1. Create a new, empty set D.
2. `read`(e) and add e to D.
3. Determine whether $D = S_0$.

```
procedure Insert(element pred, e)
(* This procedure does not conform to Section 6.7.1 *)
Insert e into S_1
(* S_1 is second set in RLSD for pred *)
if (|S_1| > 3 × 2^i) then
    ForceRLSD(1)
endif
```

Figure 9: Insertion algorithm,which uses the *ForceRLSD* procedure shown in Figure 10.

Therefore, a **Read**(e) procedure can be checked in $O(k)$ time.

The **Head**() procedure differs from the **Read**() procedure because the checker keeps a copy of the head element in reliable memory. To check the statement e = **head**(), the checker simply determines whether e is identical to the copy of the head element stored in the checker's reliable memory. The time to check the **head**() procedure is $O(1)$.

The **Write**(e) procedure makes simple changes to the data structures. Let S_0 be the set which contains e. The checker does the following:

1. Create a new empty set D.
2. Add e to D.
3. Replace S_0 with D.
4. **write**(e).

If the resource manager fails to write e correctly, then we do not consider the write fault an error. Only the subsequent erroneous read of e will be designated as an error.

In Section 6.3.2, we explained how the checker moves elements between sets to maintain the RLSD structure. The **Insert** operation requires the checker to move elements from S_i to S_{i+1} when $|S_i| > 3 \times 2^i$. For the operation **Insert**(*pred*, e), the element *pred* must be an active element, so $S_0 = \{pred\}$. Since we require $|S_0| = 1$, we may insert e into S_1. If $|S_1| > 6$, then the checker moves elements according to algorithm shown in Figures 9 and 10.

For the **Delete** operation, we require that for **Delete**(*pred*, e), both *pred* and e be active elements. Let $S_0 = \{pred\}$, $S'_0 = \{e\}$, and $S'_0, S'_1, S'_2, \ldots$ be the RLSD for e. Since e is deleted, it is no longer active. The checker simply makes the RLSD for *pred* to be $S_0, S'_1, S'_2 \ldots$

The **Next**(e) procedure is more complex than the other procedures. Let e be active, f be inactive, and suppose the user executes $f :=$ **Next**(e). After the **Next** operation, e remains active and f is active. The change of f forces a change in the structure of the RLSD. For example, let the sets in the RLSD for e be ordered $S_0, S_1, S_2, \ldots$. Since e is active, $S_0 = \{e\}$. Assume $S_1 = \{e_1, e_2, e_3\}$. In order to subdivide the set S_1, the checker creates new sets $S'_0 = \{e_1\}$ and $S'_1 = \{e_2, e_3\}$. In this example, e_1 would become the new active element, S'_0 would be the first set in the list, and S'_1 would become the second set.

```
procedure ForceRLSD(integer i)
if (|S_i| > 3 × 2^i) then
    if S_{i+1} does not exist then
        Create a new, empty S_{i+1}
        MoveLast(i)                (* Move the last 2^i elements from S_i to S_{i+1} *)
    else
        MoveLast(i)                (* Move the last 2^i elements from S_i to S_{i+1} *)
        ForceRLSD(i + 1)           (* Re-balance S_{i+1} *)
    endif
else if (|S_i| < 2^i) then
    if S_{i+1} exists then
        MoveFirst(i)               (* Move the first 2^i elements from S_{i+1} to S_i *)
        ForceRLSD(i + 1)           (* Re-balance S_{i+1} *)
    endif
endif
```

Figure 10: RLSD balancing algorithm,which uses procedures in Figures 12 and 11.

Let the sets in the RLSD be ordered $S_0, S_1, \ldots$. Here is how the checker may implement the `Next`(e) operation:

1. Let $f :=$ `next`(e).
2. Check whether f.id $= e$.sid.

   This is necessary to ensure the correctness of the checker, as shown in Section 6.6.
3. Subdivide S_1 (Figure 13).

   Remember that insertions and deletions of elements in a hashed set can be done in $O(k)$ time.
4. Make f active.

The pseudo code for subdividing is given in Figure 13. Each set S_i has a size of approximately 2^i elements, and `Next` requires calling *MoveFirst* only once per 2^i successive `Next` invocations. Since there are $O(\log d)$ sets per list, the amortized time for each `Next` operation is $O(\log d)$.

6.5 Variations

To improve the performance of the checker, we suggest several variations. First, instead of having each RLSD keep sets of minimum size 1, 2, 4, 8, ... the RLSD could keep sets of minimum size 1, 1, 2, 4, 8, ..., which has the property that $|S_i| = \sum_{j=0}^{i-1} |S_j|$. Several procedures may take advantage of this change, as we shall now see.

```
(* Move last 2^i elements from S_i to S_{i+1} to maintain RLSD *)
procedure MoveLast(integer i)
declare new, empty set D, S'_i
declare integer j
declare element e
e := first element in S_i
for j := 0 to |S_i| - 2^i - 2 do
    Insert e into S'_i                    (* Actually uses hashed sets *)
    Insert e into D
    e := next(e)
endfor
for j := |S_i| - 2^i - 1 to |S_i| - 1 do
    Insert e into S_{i+1}
    Insert e into D
    e := next(e)
endfor
if D ≠ S_i then "BUGGY"                   (* Uses probabilistic set equality procedure *)
Set new S_i := S'_i
```

Figure 11: *MoveLast* procedure.

(* Move first 2^i elements from S_{i+1} to S_i to maintain the RLSD *)
procedure $MoveFirst$(integer i)
declare new, empty set D, S'_{i+1}
declare integer j
declare element e
e := first element in S_{i+1}
for $j := 0$ **to** $2^i - 1$ **do**
    Insert e into S_i
    Insert e into D
    $e :=$ **next**(e)
endfor
for $j := 2^i$ **to** $|S_i| - 1$ **do**
    Insert e into S'_{i+1}
    Insert e into D
    $e :=$ **next**(e)
endfor
if $D \neq S_{i+1}$ **then** "BUGGY" (* Uses probabilistic set equality procedure *)
Set new $S_{i+1} := S'_{i+1}$

Figure 12: MoveFirst procedure.

(* Subdivide S_1 and force RLSD structure. *)
(* This will not work with relaxed `Next` and `Delete` restrictions of Section 6.5.*)
procedure *Subdivide*(element e)
(* e is the active element in `Next` operation *)
declare new and empty sets D, S'_0, S'_1
declare elements f, g
declare integer i
$f :=$ `Next`(e)
if $f.id \neq e.sid$ **then** "BUGGY"
for $i := 1$ **to** $|S_1|$ **do**
    Insert f into S'_1
    Insert f into D
    $g :=$ `Next`(f)
    if $g.id \neq f.sid$ **then**"BUGGY"
    $f := g$
endfor
if $D \neq S_1$ **then** "BUGGY"
Set new $S_0 := S'_0$
Set new $S_1 := S'_1$
(* Note: $|S'_1| = |S_1| - 1$, so S'_1 may be too small. *)
$ForceRLSD(1)$

Figure 13: Subdivide S_1 and re-balance the RLSD.

Several improvements rely on lazy evaluation, which means the checker waits as long as possible to do work and then updates the data structures only where necessary. For the `Delete` or `Next` procedures, the checker need only make $|S_0| = 1$ by subdividing the next set in the RLSD, instead of re-balancing the rest of the RLSD. If a set becomes empty in the process, the checker deletes the set from the RLSD. For example, let the sets in the RLSD be $S_0, S_i, S_{i+1}, \ldots$ then execute the `Delete` operation. The set S_i becomes subdivided into sets $S_0, \ldots, S_{i-1}$, and the sets in the RLSD will be $S_0, S_1, \ldots, S_{i-1}, S_{i+1}, \ldots$. The subdivision of S_i into $S_0, \ldots, S_{i-1}$ takes advantage of $|S_i| = \sum_{j=0}^{i-1} |S_j|$.

Every user operation except `Write` requires the checker to execute `read` on the active element and insert e into a hashed set D; then compare D with S_0. The checker may instead keep a copy of the active element in reliable memory and compare copies instead of hashed sets; this avoids dealing with any sets. This method uses $O(akm \log d + am)$ memory where m is the maximum element size (see Section 6.3.1), or simply $O(akm \log d)$.

6.6 Correctness of the Checker

Lemma. If a user operation $op \in O$ malfunctions, then with high probability the checker immediately detects the error.

Proof. Assume some $op \in O$ malfunctions. Let e be the element involved in the error. There are two ways that an error can occur.

1. `next`(e) fails to return the next element
2. the value read from the list for e are not the same as the value last written to e.

Both are read errors.

The program checker reads from the linked list in only two circumstances:

1. a `read` occurs on an active element, or
2. all the elements of a subdivision are read while subdividing.

We will see that the checker catches the error in both circumstances.

Let e^w be e as it was last written to, and e^r be e as it was read.

Case 1: If the checker reads from an active element e, then the subdivision contains a single element in S_0, which is $\{e^w\}$. We create a new set $D = \{e^r\}$. If $e^r \neq e^w$, then clearly $D \neq S_0$, and the checker detects this error (with high probability).

Case 2: If the checker reads an entire subdivision, then let S_i be the set of elements in the subdivision $\{e_0^w, e_1^w, e_2^w, \ldots\}$. Let $D = \{e_0^r, e_1^r, e_2^r, \ldots\}$. If some $e_i^r \neq e_i^w$, then two possibilities exist.

1. $S_i \neq D$, therefore the checker outputs "BUGGY" with high probability.
2. $S_i = D$, but as the checker was traversing the list, elements were read in the wrong order. Let e be the first element read out of order, and *pred* be the predecessor. When subdividing the list for the operation $e :=$ `next`$(pred)$, the checker finds that $pred.\mathsf{sid} \neq e.\mathsf{id}$ and declares an error. □

This algorithm is an improvement over the algorithm presented by Amato and Loui [2] because it does not require time stamps.

Lemma. If the checker declares an error, then a malfunction occurred.

Proof. We declare an error in two circumstances

1. If *pred*.sid $\neq$ *e*.id, then the checker declares an error.
2. If $S_i \neq D$, then the checker declares an error. Each element in S_i is unique and correct. Some element $e_i^w \in S_i$ but $e_i^w \notin D$. This means $e_i^w \neq e_i^r$, so e_i^r is incorrect. □

6.7 Implementation

In this section, we discuss an implementation based on Section 6.

6.7.1 Activity of Elements

We loosely say in Section 6.2 that an element is active if it may be used as a parameter for one of the checker procedures. For our implementation, we must define the word *active* more precisely.

`Head` The head element is always active, so executing this procedure does not change the activity of any element.

`Next` Let f := `Next`(e). Previous to the execution of this statement, e must be active and f may or may not be active. After executing this statement, if f is active (and e remains active).

`Read` Let f := `Read`(e). Previous to executing this statement, e must be active. After execution of this statement, e remains active.

`Write` Previous to executing `Write`(e), e must be active. After executing the procedure, e remains active.

`Insert` Remember that `Insert`(*pred*, e) inserts e after *pred*. Before executing this statement, *pred* must be active. After executing this statement, both *pred* and e are active.

`Delete` Remember that `Delete`(*pred*, e) deletes e, the successor of *pred*. Before executing this statement, both *pred* and e must be active. After executing this statement, *pred* remains active.

`Deactivate` Before executing `Deactivate`(e), e must be active. After executing the procedure, e is not active.

The tendency of these procedures is to make a large number of active elements. This requires careful use of the `Deactivate` procedure. Since, for example, most of the time the user may want execute `Deactivate`(e) after f := `Next`(e), the user may wish to add an additional layer between the checker and the user to more carefully manage active elements.

```
class Active {
declare RLSD rlsd := NULL              (* RLSD S_0 *)
    Active prev := NULL                (* Previous Active element in list *)
    boolean active_var := false
    integer id := 0                    (* The id of the active element *)
}
```

Figure 14: The `Active` object.

```
class RLSD {
declare RLSD next := NULL
        prev := NULL
        tail := NULL
    integer size := 0                  (* Size of the set *)
        head := 0                      (* id of head element *)
    HashedSet set := empty HashedSet;
}
```

Figure 15: The `RLSD` object.

6.7.2 Internal Data Structures

The checker keeps track of active elements by keeping an array of `Active` objects. The `Active` object is the highest level object which contains references to all other information, such as the active element and RLSDs. A graphical representation of the data structures is given in Figure 16. The `Active` object is given in Figure 14. Our implementation of the linked list checker is written in C++, so the code in the figures for this section resembles C++ style. All the class definitions are cumulative.

The `RLSD` class is specified in Figure 15. It is a doubly linked list with a pointer to the tail of the list. Each instance of the `RLSD` class contains a set of elements in an RLSD, and the whole structure of linked `RLSD` classes makes up an RLSD. The `RLSD` must know the size of the set, the sequence number (n of S_n in Section 6.3.1), and the head element of the set. It must also keep the hashed sets S_i (see Section 6.3.1).

The `HashedSet` class is shown in in Figure 17. The `HashedSet` object is similar to the *SetEq* procedure shown in Figure 4. Instead of using two sets and comparing them, we may insert and delete elements from the set, and then we may (probabilistically) determine whether the set is empty. This is equivalent to adding elements into the insertion set, adding

Figure 16: Graphical representation of checker data structures.

elements to the deletion set, and then determining if the two sets are equal.

6.7.3 Checker Procedures

We wish to design the procedures for the checker so they operate efficiently and manipulate the data structures correctly. Here we describe how each procedure changes the data structures.

First, there is a useful subroutine called *subdivide*. This subroutine, given in Figure 18, differs from the subroutine given in Figure 13 in that *subdivide* works with the relaxed techniques in Section 6.5, but is only used for `Next`.

The implementation of the `head` procedure is simple. The code is given in Figure 19.

The `Read` procedure is also simple. Figures 20 and 21 shows pseudo code for our implementation. Determining whether e is active requires checking e.id against all $active[i].id$ for i in $[1..MaxActive]$.

The `Write` procedure shown in Figure 22 is similar to the `Read` procedure.

The `Insert` and `Delete` procedures are more involved than the `Read` and `Write` procedures. The `Delete` procedure is shown in Figure 23. The procedure begins by determining that *pred* and e are active elements, then finding the `RLSD` which contains *pred* and e. Next, the `Active` object containing e is removed from the *active* array, which simply involves changing pointers. Finally, since the `delete` procedure changes *prev* in the linked list, we ensure that the `RLSD` containing *prev* has an accurate copy of it in its (hashed) set. The code

```
class HashedSet {
declare integer count                    (* Number of elements in set *)
    array of words H                     (* Similar to Figure 4 *)
    procedure Insert(element e)
        Hash(e, H)                       (* See Figure 3 *)
        count := count + 1
    procedure Delete(element e)
        Hash(e, H)
        count := count − 1
    procedure Empty()
        if (count = 0 and H = 0) then
            return true
        else
            return false
}
```

Figure 17: The HashedSet object.

shown in Figure 23 is simplified in that it assumes that e is not the head element of the list.

The `Insert` procedure is similar to the `Delete` procedure except that e is obviously not active. The `Insert` procedure begins by determining that *pred* is active, then assigns a unique id to e, which is an important feature for correctness (Section 6.6). Next, it does the equivalent of `Read`(e), which is necessary because *prev.sid* is changed by the `insert` procedure. After `insert`, it makes sure that the set in the `RLSD` containing *pred* contains an accurate value for *pred*.

The `Next` procedure is more complex. Remember that the statement $f :=$ `Next`(e) begins with e active and f possibly active, and leaves f active. The procedure must first make sure e is active. Since e is modified, the record is read and therefore must be checked using the probabilistic set equality checker (check if $\{e\} = S_0$). Next, S_1 must be subdivided so that $|S_1| = 1$ (see Figure 18). The checker then checks whether f is the successor of e, and inserts f into the *active* array.

6.8 Checking the Checker

The purpose of using a program checker is to ensure reliability of a system, but it is not obvious whether the implementation of the checker is correct. To ensure the checker is correct, we would require a checker for the checker, and there are obvious problems with that. We use a simple heuristic to check the checker: after every user operation, check all the internal data structures to see if they are consistent and valid. By internally consistent, we mean that the data structures make sense with respect to basic data structure rules, so that pointers point to valid data, $e.next.pred = e$, etc. By valid, we mean that the data

```
class Checker {
declare
    procedure RLSD subdivide(RLSD rlsd)
        declare RLSD new_rlsd, old_rlsd
            element e, f
            int i, j, newsize
        if (rlsd.size = 1) then         (* Trivial case *)
            e.id := rlsd.head           (* Determine {e^w} = {e^r} in case 1 of Section 6.6 *)
            e := read(e)
            if (not rlsd.set.Empty()) then ERROR
                                        (* Determine if D = S_0 *)
            rlsd.set.Insert(e)          (* Put e back into RLSD set *)
            return rlsd
        else
            e.id = rlsd.head            (* Case 2 of Section 6.6 *)
            e := Read(e)
            new_rlsd.set.Insert(e)
            Link new_rlsd into head of list.
            i := 1
            newsize := 1
            while (i < rlsd.nextsize) do
                old_rlsd := new_rlsd
                new_rlsd := New RLSD
                old_rlsd.next := new_rlsd
                new_rlsd.prev := old_rlsd
                new_rlsd.tail := rlsd.tail
                for j = 1 to Min((rlsd.nextsize − i), newsize) do
                    f := Next(e)
                    if (f.id ≠ e.sid) then ERROR
                    e := f
                    new_rlsd.set.Insert(e)
                    rlsd.set.Delete(e)
                endfor
                i := i + j
                newsize := newsize × 2
            endwhile
        if (not rlsd.set.Empty()) then ERROR
        rlsd.next.prev := new_rlsd
        endif
}
```

Figure 18: The *subdivide* procedure.

```
class Checker {
declare element headcopy
    procedure element Head()
        declare element e

        e := head()
        if (e = headcopy) then ERROR
        return e
}
```

Figure 19: The head procedure.

```
declare procedure integer DetermineActivity(e)
(* Returns index of element e in active array *)
declare integer i
for i := 1 to MAXACTIVE do
    if (active[i].id = e.id) then
        return i
    endif
endfor
return INVALID
```

Figure 20: Determine if element e is active.

```
class Checker {
define MaxActive := Maximum number of active elements
declare Array of Active active[1..MaxActive]
declare procedure Read(element e)
        declare RLSD rlsd
            integer i
            boolean found := false

        i := DetermineActivity(e)
        if (i = INVALID) then ERROR
                                        (* e must be active *)
        rlsd := active[i].rlsd
        e := read(e)
        rlsd.set.Delete(e)
        if (not rlsd.set.Empty()) then ERROR
        rlsd.set.Insert(e)
        return e
}
```

Figure 21: The Read procedure.

```
class Checker {
declare procedure Write(element e)
        declare RLSD rlsd
            integer i
            boolean found := false
            element f
        i := DetermineActivity(e)
        if (i = INVALID) then ERROR
                                            (* e must be active *)
        rlsd := active[i].rlsd
        f := read(e)
        rlsd.set.Delete(f)
        if (not rlsd.set.empty()) then ERROR
        f.data = e.data
        write(f)
        rlsd.set.Insert(f)
        return
}
```

Figure 22: The `Write` procedure.

```
class Checker {
declare procedure Delete(element pred, e)
(* Assume e is not the head of the list *)
        declare RLSD rlsd_pred, rlsd_e
            element f
            integer i
            boolean found := false

        i := DetermineActivity(pred)
        if (i = INVALID) then ERROR            (* pred must be active *)
        rlsd_pred := active[i].rlsd

        Find rlsd_e similarly.
        Remove e from the array of active elements
        rlsd_prev.next := rlsd_e.next
        rlsd_prev.tail := rlsd_e.tail
        rlsd_e.next.prev := rlsd_prev

        (* delete procedure will change pred *)
        f := read(pred)
        rlsd_prev.set.Delete(f)
        if (not rlsd_prev.set.Empty()) then ERROR
        f.sid := e.sid
        delete(pred, e)
        pred := read(pred)
        if (pred ≠ f) then ERROR (* Assume delete(pred, e) changes only pred.sid *)
        rlsd_prev.set.Insert(pred)
        Delete rlsd_e                   (* Garbage collection on rlsd_e *)
        return
}
```

Figure 23: The `Delete` procedure.

```
class Checker {
declare integer IDcounter = 0
    procedure Insert(element pred, element e)
        declare RLSD rlsd_pred
            element f
            integer i
            boolean found = false

        i := DetermineActivity(e)
        if (i = INVALID) then ERROR
                                            (* e must be active *)
        rlsd_pred = active[i].rlsd
        e.id := IDcounter                   (* Or address of e concatenated with IDcounter *)
        IDcounter := IDcounter + 1
        pred := read(pred)
        rlsd_prev.set.Delete(pred)
        if (not rlsd_prev.set.Empty()) then ERROR
        f := pred
        insert(pred, e)
        f.sid := e.id
        pred := read(pred)
        if (pred ≠ f) then ERROR (* Assume insert(pred, e) changes only pred.sid *)
        (* This check is not necessary, but it may declare errors sooner *)
        rlsd_prev.set.Insert(pred)     (* If avoiding the check, rlsd_prev.set.Insert(f) *)
        rlsd_prev.next.set.Insert(e)

}
```

Figure 24: The `Insert` procedure.

```
class Checker {
declare procedure element NEXT(element e)
        declare RLSD rlsd_e, rlsd_f
            element f
            integer i

        i := DetermineActivity(e)
        if (i = INVALID) then ERROR        (* e must be active *)
        rlsd_e := active[i].rlsd

        e := read(e)
        rlsd_e.set.Delete(e)
        if (not rlsd_e.set.Empty()) then ERROR
        rlsd_e.set.Insert(e)

        subdivide(rlsd_e.next)

        rlsd_f := rlsd_e.next
        f := next(e)
        if (f.id ≠ e.sid) then ERROR        (* Check the next procedure *)
        rlsd_f.set.Delete(f)
        if (not rlsd_f.set.Empty()) then ERROR
        rlsd_f.set.Insert(f)

        if (f is not already active) then
            Insert f into active array with index i
            active[i].rlsd := rlsd_f
        endif
        return f
```

Figure 25: The Next procedure.

is accurate and reflective of the state of the linked list. For example, the S_0 for e is $\{e\}$, the size of the hashed sets follow the RLSD rules within the variations of Section 6.5, and all the elements in the *active* array are truly active. These are necessary but not sufficient conditions for the correctness of the checker, however they do provide a higher degree of confidence.

The checker's checker is not presented here in detail because it does not provide significant insight to the working of the checker, and it was not developed to the state where it could find any errors within the checker. However, a nearly complete implementation gives us a useful and powerful debugging tool.

7 Acknowledgments

Several important people helped me in this work. I would like to thank Professor Loui for helping me with the endless revisions, helpful suggestions, countless explanations, and patience with this report. Nancy Amato provided me with some useful dialogues and encouragement. Finally, thanks to the guys who hang out at Hessil Park and play volleyball for the wonderful distraction and sore muscles.

References

[1] J. Naor and M. Naor, "Small-bias probability spaces: efficient construction and applications," *SIAM J. Comput.* vol. 22, no. 4, Aug. 1993, pp. 838–856.

[2] N. M. Amato and M. C. Loui, "Checking linked data structures," *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing*, Austin, Texas, June 15–17, 1994, pp. 164–173.

[3] S. Jimbo and A. Marouka, "Expanders obtained from affine transformations," *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, Providence, Rhode Island, May 6–8, 1985, pp. 88–97.

[4] D. E. Knuth, *The Art of Computer Programming*, vol. 2, Addison-Wesley, 1969.