

Checking Linked Data Structures*

Nancy M. Amato[†]

Michael C. Loui[‡]

Coordinated Science Laboratory, University of Illinois, 1308 W. Main St., Urbana, IL 61801

Abstract

In the program checking paradigm, the original program is run on the desired input, and its output is checked by another program called a checker. Recently, the notion of program checking has been extended from its original formulation of checking functions to checking a sequence of operations which query and alter the state of an object external to the program, e.g., checking the interactions between a client and the manager (server) of a data structure. In this expanded paradigm, the checker acts as an intermediary between the client, which generates the requests, and the server, which processes them. The checker is allowed a small amount of reliable memory and may provide a probabilistic guarantee of correctness for the client. We present off-line and on-line checkers for data structures such as linked lists, trees, and graphs. Previously, the only data structures for which such checkers existed were random access memories, stacks, and queues.

1 Introduction

The traditional methods used to ensure the correctness of programs are testing and verification. Testing certifies that the output of the program is correct for a particular set of test input values; however, no guarantee can be offered about the correctness of the output of the program for input values different from those of the test set. Formal verification determines whether a program produces the correct output for all input/output pairs, but is usually infeasible to implement for all but the simplest of programs. Both testing and formal verification are performed before the program is distributed for use; thus, neither of these methods is fault tolerant. The practical infeasibility of formal verification, the incomplete guarantees provided by testing, and the fact that neither of these methods is fault tolerant have led researchers to pro-

pose alternative methods for verifying the output of computations.

Algorithm-based fault tolerance is a method that checks an operation by tailoring the fault-tolerance scheme to the algorithm used to perform the operation. Introduced by Huang and Abraham [10], this method has since been studied extensively (see, e.g., [2, 3, 4]). Generally, in an application of this technique, the input data are encoded, and the algorithm is designed to operate on encoded input data and produce encoded output data; for example, many matrix operations can be checked by adding a checksum row and a checksum column to the input and output matrices [10]. Another interesting method for achieving fault tolerance is the *certification-trail* technique proposed by Sullivan and Masson [17, 18]. Briefly, the certification-trail technique consists of two phases. In the first phase, a modified version of the original program is run, producing both the expected output and a trail of data called a certification trail. In the second phase, another program uses the certification trail created by the first phase to determine whether an error has occurred.

A technique called *program checking*, which bears conceptual similarities to both algorithm-based fault tolerance and the certification-trail technique, has evolved in an environment largely isolated from the traditional fault tolerance community. In the program checking paradigm proposed by Blum and Kannan [6], the original program is run on the desired input, and its output is checked by another program called a *checker*. In contrast to algorithm-based fault tolerance and the certification-trail technique, the checker treats the original program as a black box controlled by an adversary. Also, the checker is allowed to make calls to the original program (with any input of its choice) and may provide a probabilistic evaluation of the correctness of the output of the original program—as long as the checker’s probability of an incorrect evaluation can be made arbitrarily small, i.e., as small as desired. This basic model of program result checking was extended by Blum, Luby, and Rubinfeld [7] to include the idea of using several different programs (a library) to check another program, and the concept of a self-testing/correcting pair of programs that enable one to

*This paper appeared in the *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing*, 1994, pp. 164–173. This work supported in part by NSF Grant CCR-93-15696.

[†]Supported in part by an AT&T Bell Laboratories Graduate Fellowship. Email: amato@cs.uiuc.edu.

[‡]Email: m-loui@uiuc.edu.

use a program that is not too faulty on average to compute a correct output. Program checkers can be used to achieve hardware and/or software fault-tolerance. For example, they can be used as acceptance tests in the recovery-block approach [15], or as alternatives to N -version programming [1], algorithm-based fault-tolerance, or the certification-trail technique.

Recently, Blum *et al.* [5] expanded the concept of program checking to include an important class of non-functional problems: checking the interactions between a user (client) and the manager (server) of some resource. In this paradigm, the checker acts as an intermediary between the client, which generates the requests, and the server, which processes them. Many software systems follow the client-server paradigm, e.g., database systems and operating systems. Often, the reliability of such non-functional programs is extremely important because they are used to perform safety-critical tasks such as monitoring intensive care patients in hospitals. Note that an important requirement of checkers for these programs is that they be on-line since the programs they check operate continuously or in real-time environments. In addition to the uses mentioned above for program result checking, checkers of this type could be used for purposes such as developing robust data structures [12, 19]. In this expanded version of program checking, the checker might use some limited amount of reliable memory—achieved through hardware fault-tolerance techniques such as hardware or information redundancy [11]. Then, if the resource resides in some larger unreliable memory, the checker can use its own smaller reliable memory to detect faults in the unreliable memory, i.e., faults in the larger memory can be detected without incurring the cost of making it fault-tolerant.

In this paper, using the checking model proposed by Blum *et al.* [5], we provide checkers, both off-line and on-line, for linked data structures such as lists, trees, and graphs. Previously, such checkers were only known for random access memories, stacks, and queues [5]. Our checkers resemble those previous checkers, but are necessarily more complex because we are interested in verifying the dynamic relationships (e.g., links) among the elements in the data structure, in addition to checking the data values stored in each element. For example, since an element can be inserted or deleted anywhere, checking a linked list is more difficult than checking a stack (queue), in which only the top (first or last) element can be accessed. We defer further discussion of our results until we have formally defined the program checking model. After we describe our checkers, we observe in Section 6 that the off-line data structure checkers presented in this paper and

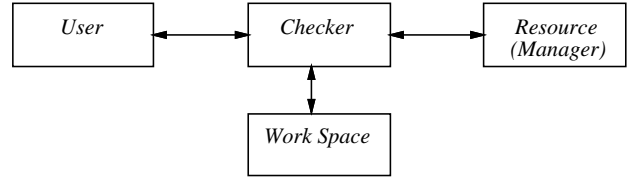


Figure 1: The checker is the intermediary between the user and the manager of the resource.

in [5] bear striking similarities to the certification-trail methods of Sullivan and Masson [17, 18]. Due to space constraints, some details have been omitted from this paper; they will be included in the full version.

2 The Program Checking Model

In this paper we adopt the program checking model proposed by Blum *et al.* [5] for checking the interactions between a user (client) and the manager (server) of a resource. We review this model here. There are three participants in the checking process: the *user* \mathcal{U} , the (manager of the) *resource* \mathcal{R} , and the *checker* \mathcal{C} . For convenience, we sometimes use the term resource \mathcal{R} , when in actuality we refer to the manager of the resource. The checker \mathcal{C} acts as an intermediary between the user \mathcal{U} , and the manager of the resource \mathcal{R} as follows (see Figure 1). The user \mathcal{U} presents operations to be performed on \mathcal{R} to \mathcal{C} , and \mathcal{C} passes these requests along to \mathcal{R} . The manager of \mathcal{R} performs the operations requested by \mathcal{C} and sends the resulting output, if any, to \mathcal{C} , which in turn passes the relevant output, if any, on to \mathcal{U} . The checker is required to pass the output of each operation (*nil* if the operation produces no output) or “BUGGY” back to \mathcal{U} before accepting another operation request from \mathcal{U} .

The checker \mathcal{C} is allowed a small amount of reliable memory, and treats the resource \mathcal{R} as if it resides in a large, unreliable memory that is controlled by an adversary. The checker’s goal is to detect an error in the behavior of \mathcal{R} when performing the operations requested by \mathcal{U} . The resources available to \mathcal{C} are a large, unreliable memory (which also contains \mathcal{R}) and a private, reliable memory. For all implementations of \mathcal{R} , and for all request sequences, the checker \mathcal{C} is required to function as follows: if \mathcal{R} ’s output is correct for all operations in the sequence, then \mathcal{C} ’s output is correct for all operations with probability at least p , and if \mathcal{R} ’s output is incorrect for any operation, then \mathcal{C} outputs “BUGGY” at least once with probability at least p , for some $3/4 < p \leq 1$. Our checkers will have $p = 1 - 1/2^k$, for any desired integer $k > 1$. Note that if the checker

has a reliable memory of size $O(m)$, then it can trivially check resources \mathcal{R} of size m by keeping its own copy of \mathcal{R} , performing each requested operation on its private copy, and comparing this output with the output produced by \mathcal{R} . Therefore, we are interested in checkers that have “small” reliable workspaces, e.g., $O(\log m)$.

A checker may be either *off-line* or *on-line*. An on-line checker outputs “BUGGY” immediately after an errant operation, and an off-line checker specifies “BUGGY” at some time after the errant operation. For example, an off-line checker may wait until after processing a sequence of requests before determining whether or not an error has occurred.

In their paper introducing this new checking model, Blum *et al.* study the fundamental problem of checking a sequence of stores to and retrieves from memory. Their results include off-line and on-line checkers for random access memories (RAMs), stacks, and queues. All the off-line checkers use $O(\log d + k)$ bits of reliable workspace to check a data structure with d elements. Their on-line checkers process sequences of n operations in $O(n \log d)$ time and use $O(k \log d)$ space (except for the on-line RAM checker, which uses $O(\log d + k)$ space). All the checkers detect errors with probability at least $1 - 1/2^k$, for any desired integer $k > 1$. The on-line RAM checker is the only checker that uses any cryptographic assumptions (e.g., a secret workspace). Blum *et al.* also establish that every checker needs a reliable memory of $\Omega(\log d)$ bits to check sequences which store d bits of data.

In this paper we present space-optimal off-line checkers for linked lists, trees, and graphs that use $O(\log d + k)$ bits of reliable memory, where d is the maximum number of elements in the data structure. The efficiency of our on-line checkers differs for each of the data structures studied. The on-line linked list checker handles a sequence of n user operations in $O(n \log d)$ time using a reliable memory of $O(k \log d)$ bits. The on-line tree checker incurs a larger amortized cost of $O(d^{1/\sqrt{\log d}}/\sqrt{\log d})$ for each user operation and uses $O(k\sqrt{d})$ bits of reliable memory. The on-line RAM checkers of Blum *et al.* are used to check graphs on-line (this is the only case in which we use cryptographic assumptions). All our checkers detect errors with probability at least $1 - 1/2^k$, for any desired integer $k > 1$.

Let b denote the maximum number of bits needed to represent any single value stored in the data structure. For simplicity, we assume that $b = O(\log d)$, where d is the maximum number of elements ever present in the data structure. If this is not the case, then an additional term of $O(b)$ bits must be added to the sizes reported for the checker’s reliable memory.

3 Data Structures

To enable the checker to monitor a data structure \mathcal{D} , we require that \mathcal{D} be accessed only by its manager.

Linked Lists. We model each element of the list as a record with three fields: *data*, and the unique identifiers of the element (*id*) and its successor (*sid*), if any. *The identifier fields can be altered only by the checker.* Linked lists are traversed using *head()* and *next(e)* operations, which return a *copy* of the requested element. *Write(e)* copies the record e to the element with identifier $e.id$. *Insert(pred, e)* inserts an element in the list after *pred*, and copies the record e to the new element. *Delete(pred, e)* deletes e , the successor of *pred*, from the list. If *pred* is *nil*, then *insert(pred, e)* (*delete(pred, e)*) inserts (deletes) the element at the head of the list.

Trees. It is assumed that trees are rooted. Each node v of the tree is a record with the five fields: *data*, and the unique identifiers of the node (*id*), its parent (*pid*), its right sibling (*rsid*), and its leftmost child (*lcid*), if they exist. If the tree is binary, or if there is some small number of children (e.g., search trees such as tries or 2-3-4 trees), then each node can store the identifiers of all its children. As with linked lists, *only the checker can alter the identifier fields.* Trees are traversed using *root()*, *parent(v)*, *rightsib(v)*, and *leftchild(v)* operations, each of which returns a *copy* of the requested element. *Write(v)* copies the record v to the node with identifier $v.id$. *Delete(par, lsib, v)* deletes the node v from the tree; the children of v become children of *par*, the parent of v . *Move(par, lsib, v, npar)* changes v ’s parent to *npar* from *par*. *Splice(par, lsib, v)* adds the subtree rooted at v to the tree so that *par* and *lsib* are v ’s parent and left sibling, respectively. *Split(par, lsib, v)* deletes the subtree rooted at v from the tree. If the *par* parameter is *nil* in any operation, then the node v refers to root. If any of the other parameters are *nil*, then it is assumed that the corresponding node does not exist, e.g., if *lsib* is *nil* then v is assumed to be the leftmost child of its parent.

Graphs. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E , where $|V| = g$. Without loss of generality, we assume that the vertices in V are indexed from 1 to g . We consider two alternative representations of graphs: adjacency matrices and adjacency lists. The following operations are available on a graph: *add_vertex(v_i)*, *delete_vertex(v_i)*, *add_edge(v_i, v_j)*, *delete_edge(v_i, v_j)*, *neighbors(v_i)* (returns the set of vertices incident to v_i), *read(v_i)*, and *write(v_i)*.

4 Off-Line Checkers

The off-line checkers given here resemble the off-line checkers for RAMs, stacks, and queues of Blum *et al.* [5]. In its reliable and private (but not necessarily secret) memory, the checker holds:

1. *The description of an ϵ -biased hash function h .*

An ϵ -biased hash function h [13] can be described using $O(\log d + k)$ random bits and can distinguish between two d -bit binary strings A and B as follows: if $A \neq B$, then $h(A) = h(B)$ with probability at most $1/2^k$. The values $h(A)$ and $h(B)$ each require $O(k)$ bits of storage. An important property of ϵ -biased hash functions is that the hashed values can be *updated incrementally*, e.g., if A represents a set, then $h(A)$ can be updated as new elements are added to A ; moreover, each such update can be performed in $O(k)$ time. (ϵ -biased hash functions were also used by Blum *et al.*)

2. *The hashed values of logs recording all the operations performed on the data structure.*

The log of operations is partitioned into the *insert* and *delete* logs, I and D , respectively. Each time the data structure is modified, a log entry recording the incremental change is made.

- (a) An entry representing the state of the data structure before the change is added to D .
- (b) An entry representing the new state of the data structure is added to I .

3. *Other information that facilitates verification of the data structure.*

In order to make the above description more concrete, we give a simple example illustrating the off-line queue checker of Blum *et al.* [5]. The checker maintains two counters, n_e and n_d , that record the number of items that have (ever) been enqueued or dequeued, respectively. After receiving an enqueue request $eq(a)$ from \mathcal{U} , the checker increments n_e , adds the tuple (a, n_e) to the insert log I , passes $eq(a)$ to \mathcal{R} , and sends *nil* back to \mathcal{U} . Similarly, after receiving a dequeue request $dq()$ from \mathcal{U} , the checker passes $dq()$ to \mathcal{R} , receives b (the result of the operation) from \mathcal{R} , increments n_d , adds (b, n_d) to the delete log D , and sends b back to \mathcal{U} . An example is shown in Figure 2. Note that if the queue is empty (i.e., if $n_e = n_d$), then the contents of the insert and delete logs are identical (as long as \mathcal{R} has functioned properly). Thus, in order to determine whether an error has occurred, the checker can simply dequeue all remaining items in the queue and check whether $I = D$ (actually, since the

Op	Queue	n_e	n_d	I : insert Log	D : delete Log
	empty	0	0	\emptyset	\emptyset
$eq(a)$	a	1	0	$\{(a, 1)\}$	\emptyset
$dq()$	empty	1	1	$\{(a, 1)\}$	$\{(a, 1)\}$
$eq(b)$	b	2	1	$\{(a, 1)(b, 2)\}$	$\{(a, 1)\}$
$eq(c)$	cb	3	1	$\{(a, 1)(b, 2)(c, 3)\}$	$\{(a, 1)\}$
$dq()$	c	3	2	$\{(a, 1)(b, 2)(c, 3)\}$	$\{(a, 1)(b, 2)\}$
$dq()$	empty	3	3	$\{(a, 1)(b, 2)(c, 3)\}$	$\{(a, 1)(b, 2)(c, 3)\}$

Figure 2: An off-line queue checker.

checker stores only the hashed values $h(I)$ and $h(D)$, it checks whether $h(I) = h(D)$).

The above example illustrates a general strategy used by the checkers of Blum *et al.* [5], which is also adopted by our checkers. In particular, the checker makes entries to the I and D logs so that $I = D$ if and only if \mathcal{R} performs all of the operations correctly. More precisely, over the lifetime of the data structure, I and D contain identical entries (but not necessarily the same sequence of entries) if and only if all operations performed on the data structure are correct. Thus, I and D are actually sets. However, it is useful to preserve the conceptual view of them as logs, which provides an ordering among their elements. Generally, the above requirement of balancing the logs will be fulfilled by “destroying” the data structure after all requested operations have been performed, e.g., by deleting all elements left in the linked list.

4.1 An Off-Line Checker for Linked Lists

To check that a linked list is functioning correctly, we need to verify that (i) the elements of the list contain the most recent values placed in them, and (ii) the links of the list reflect the current relationships between the elements. To verify condition (i), we augment each element of the linked list with a time-stamp field ts , which is updated upon each access of the element. The element identifiers are used to verify condition (ii).

Entries in the insert and delete logs record information about *links* and are of the form: $(e.id, e.sid, e.data, ts)$, where e is the origin of the link and ts is a time-stamp. We represent the logs by binary strings, and we adopt the method suggested by Blum *et al.* [5] for encoding log entries in these strings: an entry of the form $(e.id, e.sid, e.data, ts)$ corresponds to a 1 bit in the $(e.data + e.id * v + e.sid * v^2 + ts * v^3)^{th}$ bit of the string, where v is the largest possible value stored in any element of the linked list. As mentioned in Section 2, we assume $v = O(d)$, where d is the maximum number of elements in the data structure—if not, then a term of $O(\log v)$ must be added to the size reported for the checker’s memory. Thus, the lengths of the strings representing I and D are polynomial in d .

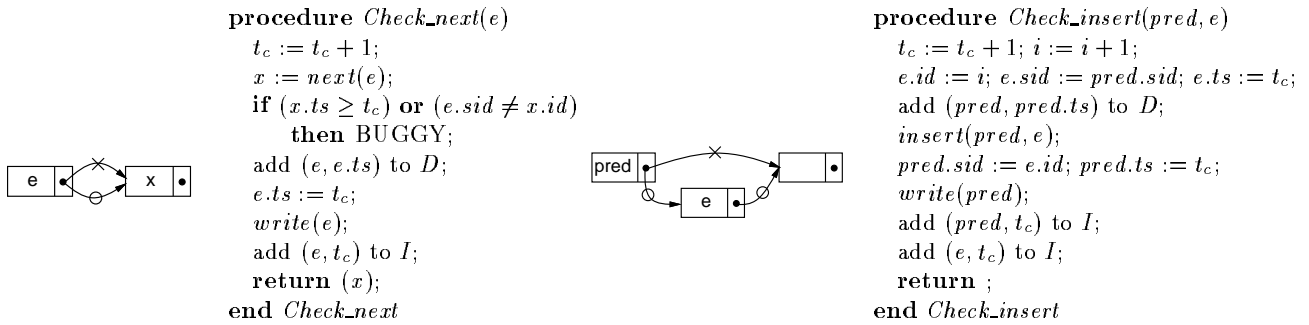


Figure 3: The links recorded in D (I) are marked with an \times (\circ).

In addition to the description of the unbiased hash function h , and the hashed logs $h(I)$ and $h(D)$, the checker also stores a copy of the head element, and two counters i and t_c in its reliable memory. The head is used to check the `head()` operation. The counter i represents the number of elements that have (ever) been inserted into the list; also i is the source of element *ids*. The counter t_c is used to generate time-stamps. The actions taken by the checker on the operations `next` and `insert` are shown in Figure 3; the other operations are handled similarly. For convenience, the tuple $(e.id, e.sid, e.data, ts)$ is represented by (e, ts) . To avoid boundary conditions, the elements involved are assumed to be internal to the list; the modifications required for the general case are straightforward. The checker is said to make entries to I and D , when in reality it updates $h(I)$ and $h(D)$; referring to the logs gives a more intuitive idea of how the checker operates.

We assume that the linked list is originally empty and that $h(I)$, $h(D)$, i , and t_c are initialized to zero. To check the functioning of the list after a sequence of operations, the checker deletes all elements remaining in the list and tests whether $h(I) = h(D)$. Actually, it is sufficient for the checker to perform a *mock deletion* by *reading* each element in the list and making the appropriate entries to D , but not the entries to I ; the entries to D are needed to balance the the corresponding entries (supposedly) previously made to I . Let O denote the sequence of all operations performed on the list.

Lemma 4.1 *If any $op \in O$ malfunctions, then $I \neq D$.*

Proof: Assume some $op \in O$ malfunctions, let l be a link involved in the error, and let e be the origin of the link l . There are two ways that an error could occur: (i) link l did not exist when op was performed, or (ii) the values read from e were not the last values written to e . However, these are both errant reads.

Select the first $op \in O$ which performs an errant read. Let e be the first element encountered during op that contains errant data, i.e., $(e.id, e.sid, e.data, e.ts)$

is the first entry made to D during op that contains erroneous data. We now argue that the tuple $(e.id, e.sid, e.data, e.ts)$ will never appear in I . It is easy to verify that the time-stamps of the entries made to I will be correct, unique, and strictly increasing since the checker is responsible for updating and storing these values. Thus, it must be that $(e.id, e.sid, e.data, e.ts)$ was not present in I when e was read during op —otherwise, e did not contain errant data, or op was not the first operation to perform an errant read. It is also easy to see that the entry $(e.id, e.sid, e.data, e.ts)$ can not be made to I after op since all such entries will receive time-stamps larger than t_c , and the checker verifies that $e.ts < t_c$. ■

Therefore, we have the following theorem.

Theorem 4.1 *For a linked list with at most l elements there exists an off-line checker which uses $O(\log l + k)$ bits of reliable memory and detects errors with probability at least $1 - 1/2^k$, for any integer $k > 1$.*

The linked list checker described above keeps several things in its reliable memory: the description of h , the hashed logs $h(I)$ and $h(D)$, a copy of the head, and the counters i and t_c . If there are multiple linked lists, each checked by the above scheme, then only the description of the hash function can be reused, i.e., $O(k)$ bits of additional memory would be required for the logs of each list, so that $O(\log l + mk)$ bits of reliable memory would be used in total to check m linked lists. However, the checker can easily be modified so that $O(l)$ linked lists can be verified simultaneously by a checker with only $O(\log l + k)$ bits of reliable memory as follows. Each list is assigned a list identifier (*lid*); this *lid* is added to the tuples entered in the I and D logs so that a single pair of I and D logs is sufficient to check all linked lists. Instead of keeping the head and the counter i of each list in its reliable memory, the checker stores these values as records in an array indexed by *lid*. The array can be checked by a RAM checker [5], which requires an additional $O(k)$ bits of reliable memory for the hashed values of its logs.

```

procedure Check_split(par, lsib, v)
  tc := tc + 1;
  add E(lsib, lsib.ts) to D;
  add E(v, v.ts) to D;
  Process_split(leftchild(v));
  split(par, lsib, v);
  lsib.rsid := v.rsid;
  lsib.ts := tc;
  add E(lsib, tc) to I;
  write(lsib);
  return
end Check_split
procedure Process_split(v)
  if (v ≠ nil) then
    add E(v, v.ts) to D;
    process_split(rightsib(v));
    process_split(leftchild(v));
  return ;
end Process_split

```

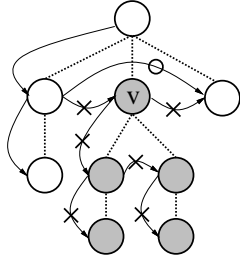


Figure 4: Tree edges are indicated by dotted lines, and the nodes deleted by the *split* operation are shaded. Links recorded in *D* (*I*) are marked with \times (\circ).

4.2 An Off-Line Checker for Trees

The off-line checker for trees is similar to the linked list checker. We include a time-stamp field in each tree node, and the checker stores in its reliable memory the description of the hash function h , the hashed logs $h(I)$ and $h(D)$, a copy of the root, and counters i and t_c . The I and D logs contain entries of the form $(v.id, v.pid, v.rsid, v.lcid, v.data, ts)$ which are represented by (v, ts) .

With the exception of the *split* and *splice* operations, the actions taken by the checker on tree operations are similar to those of the off-line linked list checker. Operations *split* and *splice* must be handled differently because they deal with *subtrees* rather than a single node of the tree, and must process every node of the relevant subtree. (We note that *split* and *splice* can be redefined so that they deal only with individual nodes, and then they could be checked in the same manner as the other tree operations.) The checking version of *split* (shown in Figure 4) calls the recursive subroutine *Process_split*, which visits each node in the subtree that is being deleted from the tree. We avoid boundary conditions by assuming that the root of the subtree is an internal node that has a left sibling; the changes required for the general case are straightforward. The *splice* operation is handled similarly.

To check the functioning of the tree at the end of a sequence of operations, a *split*(*nil*, *nil*, *root*) is performed; this is analogous to the mock deletion performed when checking linked lists. It is easily seen

that the proof of Lemma 1 ($I \neq D$ if an error occurs) continues to hold for trees.

Theorem 4.2 *For a tree with at most t nodes there exists an off-line checker which uses $O(\log t + k)$ bits of reliable memory, and detects errors with probability at least $1 - 1/2^k$, for any integer $k > 1$.*

The modification to the off-line linked list checker that enables it to check $O(l)$ linked lists with the same resources necessary to check a single linked list applies to trees as well, i.e., $O(t)$ trees can be verified off-line by a checker with $O(\log t + k)$ bits of reliable memory.

4.3 Off-Line Checkers for Graphs

If the graph $G = (V, E)$, $|V| = g$, is stored in an adjacency matrix, then we use the RAM checker of Blum *et al.* [5]. If G is stored in an adjacency list, then we use the technique discussed in Section 4.1 that checks $O(g)$ linked lists. Both methods use $O(\log g + k)$ bits of reliable memory, and detect errors with probability at least $1 - 1/2^k$, for any integer $k > 1$.

Theorem 4.3 *For a graph with at most g vertices, represented by an adjacency matrix or an adjacency list, there exists an off-line checker which uses $O(\log g + k)$ bits of reliable memory and detects errors with probability $\geq 1 - 1/2^k$, for any integer $k > 1$.*

5 On-Line Checkers

Thus far, all our checkers have been off-line, i.e., the checker is able to give an answer about the functioning of the data structure only after an entire sequence of operations has been performed. We now describe checkers that determine after *each* operation whether the data structure manager has performed correctly.

5.1 An On-Line Checker for Linked Lists

The off-line checker for linked lists (Section 4.1) checks correctness when the linked list is empty. Thus, a trivial way to verify correctness on-line is to perform a *mock deletion* of the entire list after each operation; recall that in a mock deletion, each element of the list is *read*, and the usual entries are made to the D log, but no entries are made to the I log (of course, it is also necessary to maintain a copy of the D log before the mock deletion). However, this strategy might use $O(l)$ checking operations for each linked list operation on a list with l elements.

Consider a list L containing l elements. Note that L can be viewed as the concatenation of $r = \log l$ linked lists L_0, L_1, \dots, L_{r-1} , where each L_i (except possibly

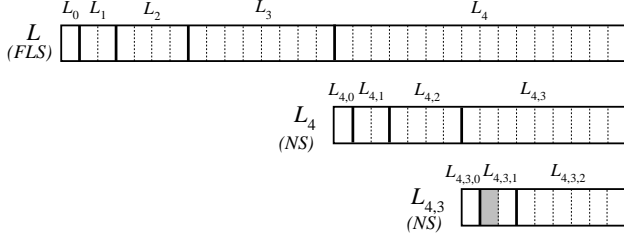


Figure 5: The FLS of the list L , and a portion of the (recursive) NS of list L_4 of the FLS.

L_{r-1}) contains 2^i elements, and the tail of L_i is linked to the head of L_{i+1} , $\forall 0 \leq i < r - 1$; this conceptual partitioning of L will be called the *logarithmic subdivision* of L . If logs I_i and D_i are maintained for each list L_i , $0 \leq i < r$, a linked list operation op can be verified on-line by mock deleting only the list L_j that is involved in op . Although in its present form this method could still require $O(nl)$ time to check $O(n)$ linked list operations, it can be refined to produce a more efficient on-line checker.

The basic idea is to recursively subdivide the lists L_i so that only the first access to L_i requires $O(2^i)$ operations. Let us call the original logarithmic subdivision of L the *first level subdivision* (FLS), and the logarithmic subdivision of any sublist of L a *nested subdivision* (NS). Let L_i , $0 \leq i < r$, denote a list of the FLS of L , where $r = \log l$ (see Figure 5). Let $L_{i,j}$, $0 \leq j < i$, denote a list of the NS of list L_i (of the FLS), let $L_{i,j,k}$, $0 \leq k < j$, denote a list of the NS of list $L_{i,j}$, etc. Suppose list $L_{i,j}$ needs to be checked. When performing the mock deletion of $L_{i,j}$, the checker creates the logs $I_{i,j,k}$ and $D_{i,j,k}$ for each list $L_{i,j,k}$ in the NS of $L_{i,j}$, $0 \leq k < j$. Although only $L_{i,j}$ is checked by the present mock deletion, entries will also be made to the logs for the list L_i (the list of the FLS that contains $L_{i,j}$) so that L_i 's logs can be balanced when necessary.

There are two potential problems with the above proposed on-line linked list checker. The first difficulty is that a large number of recursive NSs might be created, and each such subdivision creates I and D logs for every list in the NS. In fact, it is not difficult to see that there could be as many as $O(l)$ such logs created so that the checker would need $O(lk)$ reliable memory; the technique of simultaneously verifying numerous linked lists mentioned in Section 4.1 is not applicable since the on-line requirement means that we must be able to verify each list *individually* in time proportional to its length. The second difficulty arises from the fact that no provision is made for up-

dating the FLS as the structure of the list changes, i.e., since the FLS remains static there is no guarantee that there are always 2^i elements in list L_i . (Note that no similar problem occurs for the NS since it is dynamically maintained.) To address these concerns we use the following concept:

Definition 5.1 An element e is *active* at time t if at some time $t' > t$ the user performs a linked list operation involving e , and e 's predecessor is not accessed in the intervening interval $[t, t']$. The element involved in the current operation is the *current active element*.

Note that the head is always active since it has no predecessor. Since it is not possible for the checker to determine by itself which elements are active, there must be some mechanism of informing the checker which elements are active. For example, the user could explicitly designate which elements are active, or the checker might classify as active the last m elements accessed, and the user would be prohibited from accessing any other element without first accessing its predecessor. The following discussion assumes that the checker knows which elements of the list are active.

Lists with at most one active element. If, at all times, there is at most one active element e in the list (in addition to the head, which is always active), then it is easy to see that only the logs of the lists of the NSs whose heads lie beyond e (i.e., closer to the tail) need to be maintained by the checker. Consider an element e' that precedes e in the list. Note that e' cannot be accessed until all elements preceding e' in the list have been active. Thus, the NSs of L that contain e' will be formed again before e' becomes active. Therefore, only the logs of the lists of the NSs whose heads lie beyond e need to be maintained by the checker. For example, if the first element of $L_{4,3,1}$ (Figure 5) is the only active element, then the checker needs to maintain logs only for the lists $L_{4,3,1}$ and $L_{4,3,2}$. After the checker discards the logs of the lists of the NSs whose heads precede the active element e , it is a simple matter to verify that the number of logs maintained by the checker is $O(\log l)$, rather than $O(l)$.

In order to maintain the structure of the FLS in the presence of insertions and deletions, it is convenient to relax the definition of a logarithmic subdivision for the FLS as follows. A *relaxed first level subdivision* (RFLS) of L consists of the sublists $L_0, L_1, \dots, L_{r'}$, where $|L_i| < 2^i + 2^{i+1}$, and is maintained as follows.

If an element is inserted into L_i , and the new length of L_i is $< 2^i + 2^{i+1}$ then the checker does not alter the RFLS. If, however, when an element is inserted into L_i , the new length of L_i is $\geq 2^i + 2^{i+1}$, then the checker subdivides L_i into two lists, L_i' and L_i'' , where L_i' contains the first 2^i elements of L_i , and L_i'' contains

the rest of the elements of L_i . The sublist L'_i becomes the new list L_i , and the sublist L''_i is prepended to L_{i+1} ; if the new length of L_{i+1} is $\geq 2^{i+1} + 2^{i+2}$, then L_{i+1} is subdivided analogously and L''_{i+1} is prepended to L_{i+2} , and so on as necessary. Although this process may make a single insertion quite costly, the amortized cost is easily seen to be $O(\log l)$ per insertion; this follows from the fact that a list L_i is subdivided at most once for every 2^{i+1} consecutive user operations.

Deletions may cause some L_i s to become too small. In order for there to be $O(\log l)$ amortized operations for each user operation, we need the fact that sublists of size $O(2^i)$ will be accessed (and thus mock deleted) at most once for every 2^i consecutive user operations, i.e., we need to be sure L_i does not have fewer than 2^i elements very often. This problem is easily dealt with by “filling up” the lists of the RFLS as the elements traverse the list, i.e., if a list L_i contains fewer than 2^i elements, then, as the current active element traverses L_{i+1} , elements are transferred from L_{i+1} to L_i . The cost of the redistribution can be charged to the (already) deleted elements of the list.

Lists with multiple active elements. When there are multiple active elements in the list, a single RFLS is shared by all elements, but a NS is maintained for each active element. The RFLS can be maintained in exactly the same manner, and with the same cost in terms of time and space, as for lists with only one active element. However, since more than one active element can be contained in the the same list L_i of the RFLS, there may be some interaction between the NSs for different elements. In order to keep track of the interactions among the NSs for the active elements, the checker can maintain a list of the active elements sorted by position in L . The basic idea is that if there is more than one active element in some $L_i \in RFLS$, then the NSs for these elements are not be allowed to overlap, i.e., L_i is partitioned according the positions of the active elements that it contains. When an active element e “passes” another active element e' in L_i , the NS for e' becomes the NS for e , and a new NS is computed for e' when it becomes active.

It is fairly easy to see that this scheme results in the same amortized cost of $O(\log l)$ operations to check each user operation. However, it requires more reliable memory since, the checker stores $h(I)$ and $h(D)$ for each of the $O(\log l)$ lists in the RFLS, and for the $O(\log l)$ lists of the NSs for each active element.

Theorem 5.1 *For a linked list with at most l elements and m active elements, there exists an on-line checker that can check n operations in time $O(n \log l)$, uses $O(mk \log l)$ bits of reliable memory, and detects errors with probability at least $1 - 1/2^k$, for any integer $k > 1$.*

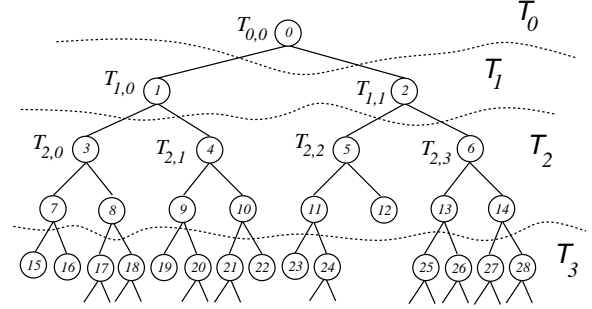


Figure 6: The FLP of the tree T .

5.2 An On-Line Checker for Trees

The structure of the on-line checker for trees is similar to that of the on-line checker for linked lists given in the previous section. Let T be a tree with root r and t nodes. As with linked lists, this checker uses the fact that operations on T can be checked on-line by performing a “mock deletion” after each operation.

We use a scheme called *logarithmic partitioning*, similar to the logarithmic subdivision used for linked lists. The basic idea is as follows. The partitioning of T is performed according to a breadth-first-search (BFS) of T . Without loss of generality, assume that the nodes of T are labeled with their BFS number, where nodes at the same level are visited in left to right order. Note that we can think of each tree as a binary tree since in our chosen representation a node is only incident to its parent, its leftmost child, and its right sibling. The tree T is partitioned into $r = O(\sqrt{\log t})$ sets of trees; the i th such set is denoted by \mathcal{T}_i , $0 \leq i < r$. The set \mathcal{T}_i contains the nodes $\{v_j | N_{i-1} \leq j < N_{i-1} + n_i\}$, where $n_k = |\mathcal{T}_k|$, and $N_j = \sum_{k=0}^j n_k$. Let $f(i) = 2^{i(i+1)}/2$, and let $n_0 = N_0 = 1$ so that \mathcal{T}_0 is the root. The nodes in the set \mathcal{T}_i , $0 < i < r$, are determined as follows. \mathcal{T}_i is initially assigned nodes numbered $N_{i-1} + 1$ through $N_{i-1} + f(i)$, and if the $f(i)$ th node in \mathcal{T}_i is at a distance of d_i from the root, then all other nodes of T that are at a distance d_i from the root are also assigned to \mathcal{T}_i . Note that $n_i \leq 2f(i)$, $0 \leq i < r$. Each set \mathcal{T}_i is partitioned into subtrees $\mathcal{T}_{i,j}$ so that $|\mathcal{T}_{i,j}| < 2^{i+1}$, $\forall j$. (See Figure 6.) It is easy to see that in total there are $O(\sqrt{t})$ subtrees in a logarithmic partition of a tree T of size t . Corresponding to the RFLS and the NSs of linked lists, the logarithmic partition of the entire tree T is called the *relaxed first level partition* (RFLP), and a logarithmic partition of a subtree of T is called a *nested partition* (NP).

The actions of the on-line tree checker are analogous to those of the on-line linked list checker, i.e., the

RFLP is shared by all active nodes in the tree (defined analogously to an active element of a linked list), and a NP is maintained for each active node. Since the partitioning is made according to a breadth-first-search traversal of the tree, a subtree of size $O(2^i)$ will be accessed at most once in every i consecutive user operations, in the worst case, with the exception of *split* and *splice* operations, which can remove or add entire subtrees to the tree, respectively. We take the view that a *split* or *splice* is allowed to be considered as $O(s)$ user operations when adding or deleting a subtree of size s ; this view is not unreasonable when one notes that the checker has verified the structures of these subtrees, and must be able to check them in their entirety, on a node by node basis. With this caveat, the on-line tree checker performs $O(t^{1/\sqrt{\log t}}/\sqrt{\log t})$ amortized operations when checking a user operation.

In addition to requiring more amortized time to check each user operation than the on-line linked list checker, the on-line tree checker also requires more reliable memory. The reason for this is that the RFLP (and subsequent NPs) of a tree T with t nodes partitions T into $O(\sqrt{t})$ subtrees. Thus, in total, for each active node the on-line tree checker may maintain $O(\sqrt{t})$ logs to check as many subtrees.

Theorem 5.2 *For a tree with at most t nodes and m active nodes, there exists an on-line checker that can check n operations in time $O(nt^{1/\sqrt{\log t}}/\sqrt{\log t})$, uses $O(mk\sqrt{t})$ bits of reliable memory, and detects errors with probability at least $1 - 1/2^k$, for any integer $k > 1$.*

We note that it is possible to construct an on-line tree checker with a smaller reliable memory if one is willing to incur a larger amortized cost for each user operation. For example, if the checker has only $O(mk \log t)$ bits of memory, then each user operation can be checked in $O(t/\log t)$ amortized operations. Finally, the on-line graph checker discussed next gives an alternative on-line tree checker which uses less space, but requires cryptographic assumptions.

5.3 An On-Line Checker for Graphs

Since graph algorithms tend to require random access to vertices, it appears that the RAM checkers of Blum *et al.* [5] are needed for an on-line graph checker. The on-line RAM checkers given in [5] differ from the checkers thus far described in this paper because they use cryptographic assumptions. In particular, they use either pseudorandom functions [9] or universal one-way hash functions [14, 16]. In both cases, the on-line RAM checker uses $O(\log n)$ bits of reliable memory and checks n user operations in $O(nf \log r)$ time, where r

is the number of cells in the RAM and f is the time required to evaluate the pseudorandom function or the universal one-way hash function.

Theorem 5.3 *For a graph with at most g vertices, there exists an on-line checker that can check n operations in time $O(nf \log g)$, uses $O(\log g + k)$ bits of reliable memory, and detects errors with probability at least $1 - 1/2^k$, for any integer $k > 1$, where f is the time necessary to evaluate a pseudorandom function or a universal one-way hash function.*

6 The Certification-Trail Technique

The *certification-trail* fault-tolerance technique, recently proposed by Sullivan and Masson [17, 18], bears striking similarities to the data structure checkers described in this paper and in [5]. Briefly, the certification-trail technique consists of two phases. In the first phase, a modified version of the original program is run, producing both the expected output and a trail of data called a *certification trail*. In the second phase, another program, called the *certifier*, uses the certification trail created during the first phase to determine whether an error has occurred.

The logs kept by the data structure checker and the certification trail serve similar purposes in the two techniques. In fact, any off-line data structure checker which adheres to the general paradigm described in Section 4 can be converted into a deterministic certification-trail technique (without reliable memory for the checker). The original program outputs the raw unhashed log entries as the certification trail. The certifier processes these to insure that every entry in D previously appeared in I . The certifier first stores the entries of the insert log in an array A_i indexed by element id ; multiple entries with the same id could be stored, e.g., in linked lists ordered by time-stamp. Then, in constant time, the certifier checks that an entry (e, ts) of D appeared in I by inspecting $A_i[e.id]$. (Multiple entries with the same element id would appear in order of time-stamp in D .) Thus, the certifier could process n data structure operations in $O(n)$ time. Note that this same technique could be performed on-line, and that each data structure operation would be processed and checked in constant time. We remark that, to our knowledge, all proposed certification-trail methods [17, 18, 8] have been off-line, i.e., a determination of correctness can be made only after the entire certification trail has been output by the modified original program and processed by the certifier. Some of this latency could be removed by, e.g., checking subsequences of operations [8], but the proposed methods remain primarily off-line.

Finally, we note that certification-trail methods cannot necessarily be transformed into checkers: the checker trusts only its small reliable memory, but the certifier implicitly trusts the certification trail created by the modified original program.

7 Open Problems

It would be interesting to find an on-line tree checker with the same efficiency as that of the on-line linked list checker, i.e., one that uses only $O(k \log t)$ bits of reliable memory and can check n user operations in total time $O(n \log t)$, where t is the size of the tree. Another, seemingly more difficult, problem is that of designing an on-line graph checker that does not use cryptographic assumptions. Note that if the graph does not have to support random access to any vertex, but can require that all searches of the graph begin at a specific vertex (or even a few such vertices), then techniques analogous to those used for trees can be employed to design checkers with similar time and space requirements. Thus, the apparent cause of the difficulty is the fact that graph algorithms tend to require random access to any vertex.

References

- [1] A. Avizienis. The n -version approach to fault-tolerant software. *IEEE Transactions Software Engineering*, 11(12):1491–1501, 1985.
- [2] V. Balasubramanian and P. Banerjee. Compiler-assisted synthesis of algorithm-based checking in multiprocessors. *IEEE Transactions on Computers*, 39(4):436–459, April 1990.
- [3] P. Banerjee and J. A. Abraham. Bounds on algorithm-based fault tolerance in multiple processor systems. *IEEE Transactions on Computers*, 35(4):296–306, April 1986.
- [4] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *IEEE Transactions on Computers*, 39(9):1132–1245, September 1990.
- [5] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proceedings of the 32nd Symposium on the Foundations of Computer Science*, pages 90–99, 1991.
- [6] M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 86–97, 1989.
- [7] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of the 22nd ACM Symposium on Theory of Computing*, pages 73–83, 1990.
- [8] Jonathon Bright and Gregory F. Sullivan. Checking mergeable priority queues. In *Digest of the 1994 International Symposium on Fault-Tolerant Computing*, June 1994, to appear.
- [9] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.
- [10] K. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, June 1984.
- [11] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison Wesley, Reading, MA, 1989.
- [12] K. Kant and A. Ravichandran. Synthesizing robust data structures - an introduction. *IEEE Transactions on Computers*, 39(2):161–173, February 1990.
- [13] J. Naor and M. Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM Journal on Computing*, 22(4):838–856, August 1993.
- [14] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 33–43, 1989.
- [15] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.
- [16] J. Rompel. One way hash functions are necessary and sufficient for secure signatures. In *Proceedings of the 22nd ACM Symposium on Theory of Computing*, pages 387–394, 1990.
- [17] G. Sullivan and G. Masson. Using certification trails to achieve software fault tolerance. In *Digest of the 1990 International Symposium on Fault-Tolerant Computing*, pages 423–431, 1990.
- [18] G. Sullivan and G. Masson. Certification trails for data structures. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, pages 240–247, 1991.
- [19] D. J. Taylor. Error models for robust storage structures. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 416–422, 1990.