Certification Trails for Data Structures

Gregory F. Sullivan[1]

Gerald M. Masson[2]

Dwight S. Wilson

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [19]. In this paper, we significantly generalize the applicability of the certification trail technique. Previously, certification trails had to be customized to each algorithm application, but here we develop trails appropriate to wide classes of algorithms. These certification trails are based on common data-structure operations such as those carried out using balanced binary trees and heaps. Any algorithm using these sets of operations can therefore employ the certification trail method to achieve software fault tolerance. To exemplify the scope of the generalization of the certification trail technique provided in this paper, constructions of trails for abstract data types such as priority queues and union-find structures will be given. These trails are applicable to any data-structure implementation of the abstract data type. It will also be shown that these ideas lead naturally to monitors for data-structure operations.

Keywords: Software fault tolerance, certification trails, error monitoring, design diversity, data structures.

1 Introduction

In this paper we significantly generalize the novel and powerful certification-trail technique for achieving fault tolerance in systems that was introduced in [19]. Although applicable to both hardware and software, we restrict our discussion of the certification-trail technique in the following to software fault tolerance. To explain the essence of the certification-trail technique for software fault tolerance, we will first discuss a simpler fault-tolerant software method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so-called time redundancy [14, 18]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct. The second possibility, of undetected faults, occurs when the output of the execution is unaffected by the faults.

[1]Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.

[2]Research partially supported by NASA Grant NSG 1442.

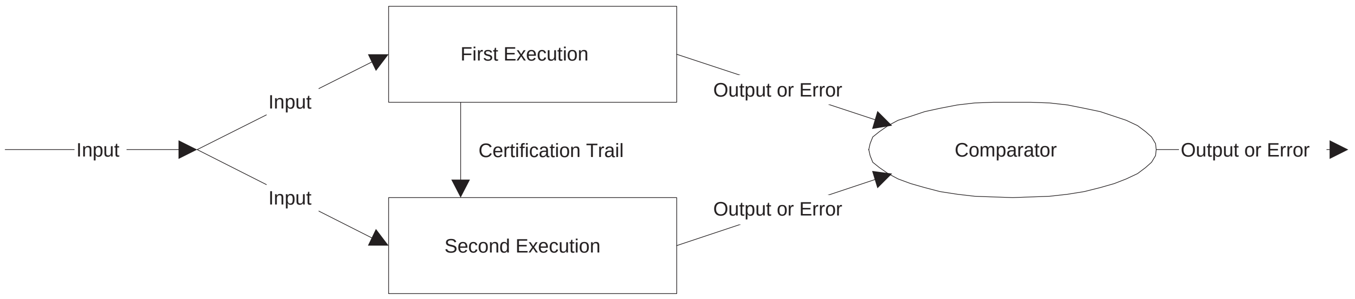

Figure 1: Certification trail method.

The certification-trail technique is designed to obtain similar types of error-detection capabilities but expend fewer resources. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail.* This data is chosen so that it can allow the the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generate a correct answer or signal that an error has been detected in the data trail.

2 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 2.1 A problem $\mathbf{P}$ is formalized as a relation, i.e., a set of ordered pairs. Let $\mathbf{D}$ be the domain (that is, the set of inputs) of the relation $\mathbf{P}$ and let $\mathbf{S}$ be the range (that is, the set of solutions) for the problem. We say an algorithm $\mathbf{A}$ solves a problem $\mathbf{P}$ iff for all $d \in \mathbf{D}$ when d is input to $\mathbf{A}$ then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 2.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. $\mathbf{T}$ is the set of *certification trails.* The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and
there exists $t \in \mathbf{T}$ such that

$F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$

(2) for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$

either $(F_2(d,t) = s$ and $(d,s) \in \mathbf{P})$ or

$F_2(d,t) =$ error.

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. The definitions above assure that the error-detection capability of the certification-trail approach is similar to that obtained with the simple time-redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

Observant readers of our earlier paper [19] in which we introduced the notion of a certification trail might have noticed that our certification-trail solution for the min-spanning tree was generalizable. The generalized technique allows one to generate a certification trail for many algorithms which use a balanced binary tree data structure. However, the technique relies on the efficient execution of the predecessor operation and some data structures such as heaps cannot execute the predecessor operation efficiently. The techniques described in this paper are even more general and powerful, and they do apply to heaps.

The degree of diversity or independence achieved when using certification trails depends on how they are used. A fuller discussion of this and of the relationship between certification trails and other approaches to software fault tolerance is contained in the expanded version of [19]. This current paper presents asymptotic analysis which shows that the certification-trail approach is desirable even when the overhead of generating the certification-trail is included. We are currently working on an experimental analysis of the method and initial results are quite promising.

3 Answer-Validation Problem for Abstract Data Types

Our general approach to applying certification trails uses the concept of an abstract data type. Some examples of abstract data types are given later in this paper. Here we mention some important common properties and give a short illustration. Each abstract data type has a well defined data object or set of data objects, and each abstract data type has a carefully defined finite collection of operations that can be performed on its data object(s). Each operation takes a finite number of arguments (possibly zero), and some but not all operations return answers. An example of an abstract data type is a priority queue. The data object for a priority queue is an ordered pair of the form (i,k) where i is an item number and k is a key value. A priority queue has two operations: insert(i,k) and delmin. The insert operation has two arguments: item number i and key value k. The insert operation does not return an answer. The delmin operation has no arguments, but it does return an answer. The precise semantics of these operations are given later in this paper.

For each abstract data type we define an *answer-validation* problem. Intuitively, the answer validation problem consists of checking the correctness of a sequence of supposed answers to a sequence of operations performed on the abstract data type. More formally, the input to the answer-validation problem is a sequence of operations on the abstract data type together with the arguments of each operation. In addition, the sequence contains the supposed answers for each of the operations which return answers. In particular, each supposed answer is paired with the operation that is supposed to return it. Examples of such inputs are given in the columns labelled "Operation" and "Answer" of table 1 and table 3.

The output for the answer-validation problem is the word "correct" if the answers given in the input match the answers that would be generated by actually performing the operations. The output is the word "incorrect" if the answers do not match. It is also useful to allow the output word to say "ill-formed". This output is used if the sequence of operations is ill-formed, e.g., an operation has too many arguments or an argument refers to an inappropriate object.

The answer-validation problem is similar to the idea of an acceptance test which is used in the recovery-block approach [17, 2] to software fault tolerance. The main difference is that an answer-validation problem is dependent upon a sequence of answers, not just an individual answer. Hence, if an incorrect answer appears in the sequence, it may not be detected immediately. It is guaranteed, however, that an incorrect answer will be detected at some point during the processing of the entire sequence. By allowing for this latency in detection, it is possible to create a much more efficient procedure for solving the answer-validation problem.

In this paper we shall solve the answer-validation problem for two abstract data types. The first data type we shall consider is for the disjoint-set union problem, and the second data type is for the generalized priority-queue problem. The priority-queue example will be presented in a sequence of stages, in which each stage allows for more data-structure operations.

The most important aspect of the answer-validation problem is that it is often possible to check the correctness of the answers to a sequence of operations much more quickly than actually calculating what the answers should be from scratch. In other words, the answer-validation problem has a smaller time complexity than the original abstract-data-type problem. For example, to calculate the answers to a sequence of n priority-queue operations takes $\Omega(n \log(n))$ time, however it is possible to check the correctness of the answers in only $O(n)$ time. This speedup is very useful in fault-detection applications.

It is possible to run an answer-validation algorithm for some abstract data type concurrently with some algorithm which uses the abstract data type. The answer-validation algorithm could act as a monitor making sure that all interactions with the abstract data type are handled correctly. This is valuable because many algorithms spend a large fraction of their time operating on abstract data types. Note, the overhead of this monitor is less than the overhead of actually performing the data-type operations a second time.

One possible application of the answer-validation problem occurs when it is used in conjunction with a repairable data structure which allows for repair but does not automatically attempt to detect faults [24]. Suppose an abstract data type is implemented with a repairable data structure. One can use an answer-validation procedure to detect errors in the answers generated by the abstract data type. When an error is detected, a repair of the data structure can be attempted. In some cases, recovery and continued execution will be possible.

In the next section, we will show how to create certification trails for programs which use abstract data types when those data types have efficient solutions for their answer-validation problems.

4 Schema for using Certification Trails

Suppose that we have developed an efficient solution to the answer-validation problem for some abstract data type. By efficient we mean the time complexity of the answer-validation problem is smaller than the time complexity of the original abstract-data-type problem. Further, suppose that we wish to run an algorithm, say A, which uses that abstract data type. To apply the certification trail method we can use the following schema to yield the two executions:

First Execution:

Execute algorithm A.
Each time an abstract-data-type operation is performed, append to the certification trail the identity of the operation, the arguments and the answer.

Second execution:

Phase One:
Validate the correctness of the operations and supposed answers given in the certification trail. If the validation returns "incorrect" or "ill-formed" then output "error" and stop. Otherwise, continue.

Phase Two:
Execute algorithm A.
Each time an abstract-data-type operation is performed, read the next entry in the certification trail. Make sure that the operation and the arguments in the certification trail agree with those requested in the algorithm. If not output "error" and stop. Otherwise, use the answer given in the certification trail and continue.

In the final step the outputs from the two executions are compared and the output is accepted or an error is signaled. This schema can yield execution times which are significantly faster than the execution time obtained by running algorithm A twice, yet these two methods give similar fault detection capabilities. That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct. Note, the first execution can be slower than a simple execution of algorithm A since it must output a certification trail. However, the second execution can be significantly faster than a simple execution of the algorithm since the interactions with the abstract data type take less time overall. The net effect can be a major speedup.

Suppose an algorithm uses multiple abstract data types and suppose there are efficient answer-validation algorithms for each of these abstract data types. It is easy to see how our method generalizes. We can leave behind a generalized certification trail which consists of a separate certification trail for each of the abstract data types. The effect on the speedup of the second execution will be cumulative.

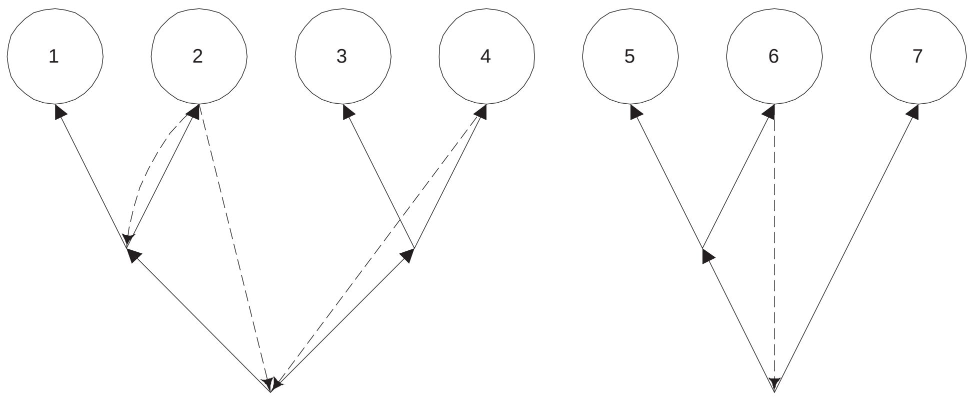

Figure 2: Union Tree with Find Edges

5 Answer Validation for Disjoint-Set Union

As our first example we will discuss the disjoint-set union problem. This problem concerns a dynamic collection of sets in which pairs of sets can be combined to yield new sets. The underlying universe of set elements consists of the integers from 1 to n inclusive. Also, the universe of set names consists of the integers from 1 to n inclusive. There are three operations that can be performed:

create(A,x) creates a singleton set named A which contains element x. Since sets must be disjoint we require that x not already be in some set.

union(A,B) creates a new set which is the union of the sets named A and B. This new set is called A and the set named B becomes undefined. It is required that the sets named A and B are originally defined and are disjoint.

find(x) returns the name of the set which contains element x. It is required that x be a member of some unique set.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

In table 1 we give an example of a sequence of disjoint-set-union operations together with the answers for find operations. In addition, the collection of sets is depicted as it is changed by the operations. For simplicity, in this example each set name corresponds to the integer originally contained in the set when it is created. Sets are listed by first giving the name of the set followed by a colon and then the contents of the set.

The disjoint-set-union problem is a classic problem which has many applications [9] such as the off-line min problem, connected components, least-common ancestors, and equivalence of finite automata. Of particular interest is the time-complexity of performing a sequence of operations. Let us say the total number of operations is m, which is assumed to be greater than or equal to n. Recall, n is the number of set elements and set names.

Tarjan gave the tight upper bound of $O(m\alpha(m,n))$ [21, 22] for this problem. The α refers to the inverse of Ackermann's function which is a very slowly growing function. His solution

Operation	Answer	Status of sets
create(1,1)		1:{1}
create(2,2)		1:{1},2:{2}
union(1,2)		1:{1,2}
find(2)	1	
create(3,3)		1:{1,2},3:{3}
create(4,4)		1:{1,2},3:{3},4:{4}
create(5,5)		1:{1,2},3:{3},4:{4},5:{5}
union(5,3)		1:{1,2},4:{4},5:{3,5}
union(5,1)		4:{4},5:{1,2,3,5}
find(2)	5	
find(5)	5	
create(6,6)		4:{4},5:{1,2,3,5},6:{6}
union(4,6)		4:{4,6},5:{1,2,3,5}
create(7,7)		4:{4,6},5:{1,2,3,5},7:{7}
union(4,7)		4:{4,6,7},5:{1,2,3,5}
find(6)	4	

Table 1: Sequence of operations for a Disjoint Set Union

and earlier solutions used a path-compression heuristic [23]. Fredman and Saks gave a lower bound of $\Omega(m\alpha(m,n))$ [10] in a general cell-probe model. Gabow and Tarjan show how to solve some important special cases of this problem in O(m) time [11].

We now consider the answer-validation problem for the disjoint-set-union data type. We will show that this problem can be solved in $O(m)$ time where m is the number of operations. Note, this time complexity is superior to the complexity of actually performing the sequence of operations as discussed above. One method for solving this problem in $O(m)$ time uses the powerful techniques of Gabow and Tarjan [11]. However, we shall present a simpler method with a small constant of proportionality that is tailored to this problem.

To solve this problem we will build a forest based on the union operations in the sequence. In addition, we shall add edges to this forest based on the find operations. As a final step we will perform a traversal of the forest and perform appropriate checks. The solid edges in figure 2 indicate the forest we would build for the set of operations given in table 1. The dashed edges indicate the edges we would add to the forest based on the find operations.

Algorithm for Answer Validation for Disjoint-Set Union

Input: sequence of m operations together with arguments and supposed answers for the disjoint-set union data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: Type *treenode* has fields left and right. Type *treeleaf* contains a list of pointers such that each pointer points to a treenode or a treeleaf. Array *activeset* is indexed by set name. Each array element is a pointer to a treenode or a treeleaf. Array *whereis* is indexed by an element number. Each array element is a pointer to a treeleaf. Initially, all pointers are nil and lists are null.

In the first phase of the algorithm we process each operation as it appears serially using the following rules:

create(A,x): If activeset[A] or whereis[x] are non-nil then output "ill-formed" and stop. Otherwise, allocate a treeleaf and set activeset[A] and whereis[x] to the allocated node.

union(A,B): If activeset[A] or activeset[B] are nil then output "ill-formed" and stop. Otherwise, allocate a treenode and set left to activeset[A] and right to activeset[B]. Next set activeset[A] to the treenode and set activeset[B] to nil.

find(x) **A**: (where A is the supposed answer to the find.) If whereis[x] is nil then output "ill-formed". Otherwise, whereis[x] points to some treeleaf. Call it tleaf. If activeset[A] is nil then output "ill-formed". Otherwise, activeset[A] points to some treeleaf or treenode. Call it t. Add a pointer to t to the list of pointers contained in treeleaf.

In the second phase of the algorithm we shall traverse the structure we have built.

Scan thru the array activeset to find non-nil pointers. It is not hard to see that each non-nil pointer points to the root of a tree made up of nodes of type tnode and tleaf. The tree uses the edges in the left and right fields of tnode.

For each such tree perform a depth-first search. Whenever the search reaches a node of type tleaf traverse the list of pointers that it contains. Check that each pointer points to a node which is currently on the stack which is used to perform the depth-first search. This is equivalent to checking that each pointer in tleaf points to a node which is an ancestor of tleaf in the tree.

If some pointer does not point to an ancestor then output "incorrect" and stop. Otherwise, output "correct" and stop.

Theorem 5.1 *The algorithm for answer validation of the disjoint-set-union abstract data type is correct.*

Theorem 5.2 *The answer validation algorithm for disjoint set union has a time complexity of $O(m)$ for processing a sequence of m operations.*

We omit these two theorems which overall are not difficult to show. We comment on one aspect of implementation. In the second phase of the answer validation algorithm it is necessary to determine if certain nodes are on the stack during the tree traversal. This can be done efficiently as follows: First, each treenode and each treeleaf can be assigned a unique identifier in the range 1 to m as it is allocated. Next, a boolean vector of size m indexed by the unique identifiers described above can be allocated. This vector can be used to keep track of which nodes are on the stack during tree traversal by turning bits on and off. This modified tree traversal algorithm still takes $O(m)$ time.

6 Simple Priority Queue

In the next three sections we present four algorithms for answer validation of priority queues. The first is a simple, but inefficient, algorithm that illustrates some of the concepts used in the second two algorithms. The second algorithm is a linear time algorithm for a simple priority queue supporting only insert, deletemin, and member operations above. Third is a linear time algorithm for answer validation of a priority queue also supporting delete and changekey operations. Finally, we show how the third algorithm may be used to construct an algorithm for validating sequences of insert, changekey, delete, min, deletemin, max, and deletemax operations.

In discussing the first algorithm, we shall make the simplifying assumption that each item number is used at most once. This assumption will be removed when the other two algorithms are discussed.

6.1 Simple Priority Queue Definition

A *simple priority queue* contains $(item, value)$ pairs, where *item* is an item number from the set $[1..N]$ and *value* is a member of an arbitrary completely ordered set. At any given time, the pairs in the priority queue have distinct item numbers though the value field may be the same for multiple pairs. It is, however, possible for an item number to be reused. That is, the pair (i, v_1) could be added to the priority queue, removed at some later time, and then the pair (i, v_2) added, where v_1 and v_2 may or may not be distinct. Pairs are ordered as follows: $(i_1, v_1) < (i_2, v_2)$ iff $v_1 < v_2$ or $v_1 = v_2$ and $i_1 < i_2$. It is not possible for two pairs in the priority queue at the same time to be equal, since their item numbers must be distinct.

The following operations are supported by a simple priority queue:

insert(i,v): Adds the pair (i, v) to the priority queue as long as the item number i is not currently in use.

(i,v) = min(): Returns the smallest pair (i, v) contained in the priority queue.

(i,v) = deletemin(): Returns the smallest pair (i, v) in the priority queue and removes it from the queue.

6.2 A Simple but Inefficient Answer Validation Algorithm

Given a sequence of simple priority queue operations, along with answers to the min and deletemin operations, we wish to check that the answers are correct. This is done by three classes of checks.

First, technical checks are performed to verify that item numbers are not duplicated and that the answers given for min and deletemin operations correspond to elements actually in the priority queue at the time those operations are performed. We say that the input is *ill-formed* if either of these conditions is violated and that it is *well-formed* otherwise. These conditions are easily verified by a linear scan through the input operations and answers, so in the text below we assume that the input is well-formed.

To perform the other checks, we first associate a time stamp with each operation by numbering the operations sequentially. The insert time of an element is the time of the insert operation that placed it in the priority queue. Since we are assuming that item numbers are not reused, we may also speak of the insert time of an item number, which is the insert time of the pair having that item number.

Next, we scan the list of operations and create two data structures. The first structure is an array, *InsertTime*, indexed by item number and containing the insert time for every element ever placed in the priority queue. The second structure is *Answers*, a list of triples of the form $(i, v, atime)$, where (i, v) is the answer given for the operation at time *atime*. We form one such answer triple for each min and deletemin operation.

We now perform the following check on answers, or more formally on answer triples. For every pair of answers, check that either the earlier answer is less than or equal to the later one, or that the pair given as the later answer was not inserted until after the first answer was returned. Formally, for every pair of answer triples $(i_1, v_1, atime_1)$ and $(i_2, v_2, atime_2)$ in *Answers* with $atime_1 < atime_2$, we check that $(i_1, v_1) \leq (i_2, v_2)$ or $InsertTime[i_2] > atime_1$. If this check fails, then some answer on the input must be incorrect. Assume that all answers given before time $atime_1$ are correct. Then the answer (i_1, v_1) given at $atime_1$ cannot be correct, because the smaller pair (i_2, v_2) is in the priority queue at that time. Thus either the answer (i_1, v_1) at time $atime_1$ is incorrect, or some earlier answer was incorrect.

If the priority queue is empty after the operations are executed, these check are sufficient. Otherwise, they generally will not be. Intuitively, if some pair (i, v) is not given as the answer to any min or deletemin operation, then it will not be involved in any of the above checks and we won't detect an error if it should have been then answer to one of the operations. A similar problem exists for a pair (i, v) that is given as the answer to one or more min operations but not to any deletemin operation. If t_{last} is the last time (i, v) is given as an answer, the checks above do not rule out the possibility that (i, v) should be the answer to some operation occurring after t_{last}.

The final set of checks guard against these possibilities. Let Let S be the set of pairs that are inserted into the queue but never returned as an answer to a deletemin operation. Note that such a pair may have been returned as an answer to a min operation. For each such pair and each answer, we check that either the answer is less than or equal to to the pair, or that the pair was not inserted until after the answer was given. Formally, for each answer triple $(i_1, v_1, atime_1)$ and each element (i_2, v_2) in S, we check that $(i_1, v_1) \leq (i_2, v_2)$ or $InsertTime[i_2] > atime_1$. If this check fails, then some answer on the input is incorrect. As above, assume that all answers before time $atime_1$ are correct. Then the answer (i_1, v_1) given at $atime_1$ is incorrect because the smaller pair (i_2, v_2) is present in the priority queue at that time.

These three sets of checks provide an algorithm for validating the answers of simple priority queue operations. A formal proof of the correctness of this algorithm will not be given. It is similar, though simpler, than the proof we present for the linear time algorithm given in the following section.

The problem with this simple algorithm is that it requires $O(n^2)$ checks, and thus takes more time than well-known priority queue implementations. We now show that not all of these checks are required and that by carefully maintaining a subset of answers, a linear number of checks will suffice.

6.3 Linear Time Answer Validation

Input: Sequence of priority queue operations (insert, min, deletemin) with answers to min and deletemin operations Output: “correct”, “incorrect”, or “ill-formed”.
Variables:

CurrentTime A variable indicating which operation is being processed. $CurrentTime = n$ indicates that the n-*th* operation from the sequence is being processed. Initialized to 1.

$InsertTime[1..N]$ $InsertTime[i]$ contains the time of the insert operation that added the pair with item number i. Each element of this array is initialized to the special value *unused.*

Value[1..N] For each pair (i, v) currently in the queue, $Value[i] = v$. For all other item numbers i, $Value[i]$ is undefined.

AnswerStack $(i, v, atime)$ where (i, v) is a pair returned by a min or deletemin operation and *atime* is the time of the operation returning that pair. Our stack structure supports the following operations:

isempty(S) Returns *true* if the stack S is empty and *false* otherwise.

push($S, i, v, atime$) Adds the triple $(i, v, atime)$ to the top of the stack S.

$(i, v, atime)$ =pop(S) Removes the top triple from the stack and returns it.

$(i, v, atime)$ =top(S) Returns the top triple without removing it from the stack.

We will frequently speak of one stack element $(i_1, v_1, atime_1)$ being “above” or “below” another element $(i_2, v_2, atime_2)$. $(i_1, v_1, atime_1)$ above $(i_2, v_2, atime_2)$ means that it is closer to the top of the stack, i.e., that $(i_2, v_2, atime_2)$ was already on the stack when $(i_1, v_1, atime_1)$ was pushed. $(i_1, v_1, atime_1)$ below $(i_2, v_2, atime_2)$ means that $(i_2, v_2, atime_2)$ is above $(i_1, v_1, atime_1)$. The terms “immediately above” and “immediately below” mean that there are no stack elements between them. We may drop the word “immediately” if it is clear from the context).

Finally, we define the result of comparison two stack triples as follows: $(i_1, v_1, atime_1) < (i_2, v_2, atime_2)$ iff $(i_1, v_1) < (i_2, v_2)$ or $(i_1, v_1) = (i_2, v_2)$ and $atime_1 < atime_2$. This ordering has independent of the order of triples on the stack. Furthermore, the third field will always be different for different triples (since each operation produces at most one triple), and therefore two triples will never be equal.

We now describe and present pseudo-code for the steps performed by the answer validation algorithm. The validation precedes in two phases. During the first phase, the input operations and answers read, and the associated routines below are executed. Between operations, the *CurrentTime* variable is incremented. After all operations are complete, the FinalPhase procedure is executed. Table 2 provides an example of these routines. The top of the stack is on the left in table 2.

insert(i,v): Check whether the item number i is currently in use. If so, the input is ill-formed and we halt immediately. If not, set $InsertTime[i]$ to *CurrentTime* and $Value[i]$ to v.

Time	Operation	Answer	Insert time	Stack used in validation
1	insert(6,300)			
2	insert(2,404)			
3	insert(3,250)			
4	deletemin	(3,250)	3	(3,250,4)
5	insert(10,248)			
6	insert(12,245)			
7	insert(4,260)			
8	min	(12,245)	6	(12,245,8), (3,250,4)
9	insert(13,140)			
10	insert(5,142)			
11	deletemin	(13,140)	9	(13,140,11), (12,245,8), (3,250,4)
12	deletemin	(5,142)	10	(5,142,12), (12,245,8), (3,250,4)
13	deletemin	(12,245)	6	(12,245,13),(3,250,4)
14	deletemin	(10,248)	5	(10,248,14),(3,250,4)
15	deletemin	(4,260)	7	(4,260,15)

Table 2: Sequence of Priority Queue operations illustrating answer validation algorithm

```
insert($i$,$v$)              /* $(i, v)$ is the pair to insert */
{
    if ($InsertTime[i] \neq unused$) output "ill-formed" and halt
    $InsertTime[i] = CurrentTime$
    $Value[i] = v$
}
```

(i,v) = min(): First check if *InsertTime*[i] is set to *unused* and if *Value*[i] is not equal to v. If either of these is the case, the input is ill-formed and we halt. Otherwise, pop elements off the top of the stack until $(i, v, CurrentTime)$ is less than the top stack element (it is possible that no elements are popped). If the stack is empty, push $(i, v, CurrentTime)$ onto the stack. Otherwise let $(i_2, v_2, atime_2)$ be the top stack element and compare $atime_2$ with insertion time of (i, v). If (i, v) was inserted before $atime_2$ output "incorrect" and halt (in this case, the answer (i_2, v_2) was returned while the smaller element (i, v) was in the queue). Otherwise, push $(i, v, CurrentTime)$ onto the stack.

```
min(i,v)                     /* (i,v) is the answer given in the input for this min */
{
    if ($InsertTime[i] = unused$ or $Value[i] \neq v$)
        output "ill-formed" and halt

    while (not empty(AnswerStack)) {
        $(i_2, v_2, atime_2) = top(AnswerStack)$
        if $(i, v, CurrentTime) > (i_2, v_2, atime_2)$
            pop($AnswerStack$)
        else if ($InsertTime[i] < atime_2$) output "incorrect" and halt          (1)
        else exit while loop
    }
    push($AnswerStack, i, v, CurrentTime$)
}
```

(i,v) = deletemin(): We perform the same operations as for min, and in addition set *InsertTime*[i] to *unused*.

```
deletemin(i,v)          /* (i,v) is answer */
{
    min(i,v)
    $InsertTime[i] = unused$
}
```

In the final phase, we examine the elements remaining in queue, i.e., those for which $InsertTime[i] \neq unused$. For each such pair (i, v) form the triple $(i, v, InsertTime[i])$. Use bucket sort to order these triples by insertion time in linear time, and call the resulting list *SOrdered*. Pop the remaining stack elements and place them in a list sorted by answer time. Call this list *AOrdered*. Compare each triple $(i_1, v_1, InsertTime[i_1])$ in *SOrdered* with the the triple $(i_2, v_2, atime_2)$ in *AOrdered* having the smallest $atime_2$ such that $atime_2 > InsertTime[i_1]$. If $(i_1, v_1) < (i_2, v_2)$ then output "incorrect" and halt. If $(i_1, v_1) \geq (i_2, v_2)$ for all pairs compared, then output "correct". Note that since *SOrdered* and *AOrdered* are sorted by the time field, the comparisons may be performed in linear time by marching down both lists in parallel.

```
FinalPhase()          /* executed after all operations have been performed */
{
    $S$ = Set of triples $(i_1, v_1, InsertTime[i_1])$ for each item number $i_1$
        where $InsertTime[i_1] \neq unused$
    $SOrdered = S$ sorted by the third field (insertion times)
    $AOrdered$ = triples $(i_2, v_2, atime_2)$ remaining on the stack in order of $atime_2$.

    For each triple $(i_1, v_1, InsertTime[i_1])$ in $SOrdered$ {
        Let $(i_2, v_2, atime_2)$ be the triple from $AOrdered$ with
        the smallest answer time such that $atime_2 > InsertTime[i_1]$

        If no such triple exists, examine the next triple from $SOrdered$.
        else if $(i_1, v_1) < (i_2, v_2)$                                    (2)
            then output "incorrect" and halt.
        else
            continue
        }
        output "correct"
}
```

The following lemma lists three properties of *AnswerStack* that are maintained by the above operations. These properties are used to prove the correctness of the answer validation algorithm.

Lemma 6.1 *The following stack properties are maintained throughout the algorithm. We*

1. The answer time field of stack triples are in strictly decreasing order from the top to the bottom of the stack.

2. Let $(i_1, v_1, atime_1)$ and $(i_2, v_2, atime_2)$ be two adjacent stack triples with $(i_1, v_1, atime_1)$ immediately above $(i_2, v_2, atime_2)$. Then $(i_1, v_1, atime_1) < (i_2, v_2, atime_2)$. More generally, the stack triples are in strictly increasing order from the top to the bottom of the stack. Furthermore, since the answer time fields are in decreasing order, this implies that the pairs formed by the first two elements of each element are in strictly increasing order. Note that this implies that for any given pair (i, v), there is only one stack triple with item number i and value v. This triple will have an answer time field equal to the number of the most recent min or deletemin operation with answer (i, v).

3. Let $(i_1, v_1, atime_1)$ and $(i_2, v_2, atime_2)$ be two adjacent stack triples with $(i_1, j_1, atime_1)$ above $(i_2, j_2, atime_2)$. Let t_{ins} be the insert time for the instruction that inserted the pair (i_1, j_1) corresponding to the stack triple $(i_1, v_1, atime_1)$. (Recall that if (i_1, j_1) has been inserted multiple times, this corresponds to the last instance such insertion before the current time. Also note that if this instance of (i_1, j_1) is still in the priority queue, then $t_{ins} = InsertTime[i_1]$). Then $t_{ins} > atime_2$.

Proof: The first property is clear because when an element is added to the stack, the value of *CurrentTime* is greater than the answer time of any element on the stack. Since elements may only be added to the top of the stack, answer times must decrease from the top to the bottom of the stack.

The second property is trivially true for a stack with fewer than two triples, and is therefore true at the start of the algorithm.

Suppose that the stack has this property before a min or deletemin operation is performed. Let $(i_1, v_1, atime_1)$ be the answer triple for that operation. Let $(i_2, v_2, atime_2)$ be the smallest stack triple s.t. $(i_1, v_1, atime_1) < (i_2, v_2, atime_2)$. If there is no such element, then all stack triples will be popped by the min or deletmin operation and $(i_1, v_1, atime_1)$ pushed, in which case the property remains true. Otherwise, all triples above $(i_2, v_2, atime_2)$ will be popped, since by assumption they are all smaller than this triple, and hence smaller than $(i_1, v_1, atime_1)$. Similarly, all triples below $(i_2, v_2, atime_2)$ are greater than it are in strictly increasing order. Therefore when the elements above $(i_2, v_2, atime_2)$ are popped and $(i_1, v_1, atime_1)$ is pushed, the ordering of the triples is maintained.

Since the answer time of the triple being pushed on the stack is larger than that of any stack triple, if there is a triple $(i_1, v_1, atime_2)$, then that triple will is smaller than $(i_1, v_1, atime_1)$, so it is popped. Thus, only one triple with item i_1 and value v_1 will be on the stack, and the answer time field corresponds to the most recent operation returning that pair.

Property 3 is checked by min and deletemin operations when a triple $(i_1, v_1, atime_1)$ is first pushed on the stack. Since triples may only be added to the top of the stack, and no element in the stack may be modified, this property is maintained throughout the algorithm. Note that $InsertTime[i_1]$ may change if the pair (i_1, v_1) is removed from the priority queue and a new pair with item number i_1 is inserted. This is not important, since checks involving stack triple $(i_1, v_1, atime_1)$ do not depend on $InsertTime[i_1]$. Note that $InsertTime[i_1]$ may be examined if the other pair involved in such a comparison also has item number i_1. This does not cause problems since only one pair with item number i_1 can be in the priority queue at any time. $InsertTime[i_1]$ will be valid for that pair, and the insert time of any earlier pair with item number i_1 is not used in any check. ∎

Theorem 6.2 *The algorithm for answer validation of the simple priority queue terminates on all input. It outputs "correct" if the answers on the input are correct and "incorrect" or "ill-formed" if they are not.*

Proof: The proof is in two parts:

Part I: Suppose the answers given in the input are correct.

First we must check that each operation successfully executes. The checks against $InsertTime[]$, and $Value[]$ are technical checks that verify that the item numbers used by insert operations are not already in use and that the pairs given as answers are actually in the priority queue at the time the associated operations are performed. These checks will clearly succeed if the answers given on the input are correct.

Therefore, only min or deletemin operations can fail, and only if the check given at (1) in the pseudocode for min fails. Let (i_1, v_1) and (i_2, v_2) be any two answers from the input sequence and let $(i_1, v_1, atime_1)$ and $(i_2, v_2, atime_2)$ be the associated answer triples. WLOG, assume $atime_1 > atime_2$. Let $t_{ins} = InsertTime[i_1]$ at time $atime_1$, i.e., the largest insertion time of pair (i_1, v_1) less than $atime_1$. Then either $(i_1, v_1, atime_1) \geq (i_2, v_2, atime_2)$ or $t_{ins} > atime_2$. If not, then $(i_1, v_1) < (i_2, v_2)$ and (i_1, v_1) was present in the priority queue at time $atime_2$. But this contradicts the assumption that (i_2, v_2) is the correct answer at $atime_2$. Since this is true for any two answer in the sequence, the check given at (1) cannot fail if all the answers are correct.

The same argument shows that the check at (2) in FinalPhase will also always succeed if all answers are correct. It is clear from the pseudocode that processing of each priority queue operation terminates and also that the FinalPhase routine terminates, thus the algorithm will terminate and output "correct".

Part II: Assume that there is at least one incorrect answer in the input. Then we show that the algorithm will output "incorrect" or "ill-formed".

Again, the initial checks in min and insert against $InsertTime[]$ and $Value[]$ will catch an attempt to use an item number currently in use or to return a pair that is not currently in the priority queue. If this happens "ill-formed" will be output. Thus, we may assume that the input is well-formed.

As a point of clarification, when we refer to the "answer" of an operation we mean the answer given in the input for that operation. The term "correct answer" refers to the answer that would be given by a correct execution of the operations. An "incorrect answer" is an answer on the input that is not a correct answer.

Let t_{wrong} be the time of the first operation for which an incorrect answer is given. Let (i_1, v_1) be the pair that is the correct answer to that operation. Since all previous answers are correct, we know that (i_1, v_1) is smaller than the incorrect answer given.

Let t_{ins} be the largest time for any insert(i_1,v_1) instruction with $t_{ins} < t_{wrong}$, i.e., the insertion corresponding to the instance of (i_1, v_1) that is in the priority queue at time t_{wrong}. Let t_{del} be the first deletemin operation after time t_{wrong} with an answer of (i_1, v_1). If there is no such operation, then $t_{del} = \mathit{infinity}$.

Let $(i_2, v_2, atime_2)$ be the largest answer triple occurring with $t_{ins} < atime_2 < t_{del}$, i.e., (i_2, v_2) is the largest answer that is returned while (i_1, v_1) is in the priority queue. We know that $(i_2, v_2) > (i_1, v_1)$ because (i_2, v_2) is at least as large as the incorrect answer returned at time t_{wrong}. Finally, note that the pair (i_2, v_2) may be given as answer more than once between times t_{ins} and t_{del}. Since we have selected the largest answer triple, it corresponds to the last time (i_2, v_2) was given as an answer during that interval.

We will now show that either check done at (1) or the check done at (2) must eventually fail. There are two cases, depending on whether or not (i_1, v_1) is given as answer.

Case 1: Suppose that (i_1, v_1) is given as the answer to some operation occurring between some time $atime_1 > atime_2$ This could be the result of a min operation, so this case may apply even if $t_{del} = \mathit{infinity}$. Clearly $atime_1 \leq t_{del}$.

At time $atime_1$ the triple $(i_2, v_2, atime_2)$ must be in the answer stack. This is because it was placed on the stack at time $atime_2$ and no larger element has been returned as an answer between $atime_1$ and t_{del}. Suppose that $(i_2, v_2, atime_2)$ is actually the topmost stack element after popping any triples smaller than $(i_1, v_1, atime_1)$. Then check (1) will fail since $InsertTime[i_1] = t_{ins} < atime_2$.

Otherwise, let $(i_3, v_3, atime_3)$ be the topmost element of the stack after popping. Then stack property 1 implies that $atime_2 < atime_3$, so $InsertTime[i] = t_{ins} < atime_2 < atime_3$, so once again check (1) fails.

There is an interesting subtlety at this point. We have identified a specific comparison that will fail, however there is no guarantee that the algorithm will actually reach this comparison. It is possible that some earlier comparison will have failed, stopping execution of the algorithm. What we can say is that if the algorithm reaches this comparison, then it will fail. If it does not reach this comparison, it must be because an earlier comparison failed. In either case, the algorithm will output "incorrect" and halt.

Case 2: (i_1, v_1) is not given as answer after time $atime_2$.

Then (i_1, v_1) must be in the priority queue after all operations are complete because there is no deletemin after time t_{ins} with (i_1, v_1) as its answer. This means that the triple (i_1, v_1, t_{ins}) will be in the list of remaining elements considered during FinalPhase. The same reasoning as in Case 1 implies that $(i_2, v_2, atime_2)$ will be in *AnswerStack* after all operations are executed.

Now let, $(i_3, v_3, atime_3)$ be the smallest triple remaining in *AnswerStack* that is larger than (i_1, v_1, t_{ins}). Then, if $(i_3, v_3) \neq (i_2, v_2)$ the triple $(i_3, v_3, atime_3)$ must be above $(i_2, v_2, atime_2)$ in the stack, so $atime_3 \geq atime_2$

The triples (i_1, v_1, t_{ins}) and $(i_3, v_3, atime_3)$ will fail check (2) in FinalPhase, since $(i_1, v_1) < (i_3, v_3)$ and $InsertTime[i] = t_{ins} < atime_2 < atime_3$.

As in the first case, we cannot guarantee that the comparisons considered above will ever be reached. If they are not, it can only be because an earlier comparison failed, ending the algorithm.

Thus, the FinalPhase routine will output “incorrect” and halt.

▮

7 Priority Queue

7.1 Priority Queue Definition

A *priority queue* is similar to the simple priority queue structure described previously. In addition to the insert, min, and, deletemin operations, an operation *delete(i)* which removes the pair with item number i is supported. This operation fails if there is no such element currently in the priority queue. This structure also supports the operation *changekey(i,w)*. This operation find the pair (i, v) with and changes its value field to w. Since this operation may be implemented as a delete(i) followed by an insert(i,w), we need not considered it in the material below.

7.2 Answer validation algorithm

Input: Sequence of operations with answers to min and deletemin operations Output: “correct”, “incorrect”, and “ill-formed” Variables:

CurrentTime: Same as for the simply priority queue.

InsertTime$[1..N]$: Same as for the simply priority queue.

Value$[1..N]$ Same as for the simply priority queue.

AnswerStack AnswerStack is similar to the variable of the same name in the previous algorithm, but somewhat more complex. This stack consists of quadruples $(i, v, atime, s)$, where i, v, and $atime$ are the same as in the previous algorithm, and s is a set of item numbers. In addition to the stack operations described above, we require the operations:

$(i, v, atime, s) = find(i)$ Finds the stack element whose set s contains the pair with item number i.

$istop((i, v, atime, s))$ Returns true iff the argument is the top element of the stack.

$(i_2, v_2, atime_2, s_2) = up((i_1, v_1, atime_1, s_1))$ Returns the stack element immediately above $(i_1, v_1, atime_1, s_1)$.

$add((i_1, v_1, atime_1, s_1), i_2)$ adds the item number i_2 to the set s_1.

$remove(i)$ Removes i from the set s containing it. Note that s is not an argument to this operation.

An efficient implementation need not store the actual set in the stack element (a pointer to the set suffices) but the explanation is simplified if we describe sets as being part of the stack elements.

Time	Operation	Answer	Insert time		Stack Used in validation	
1	insert(5,310)					(0,-∞,-1,{5})
2	insert(6,210)					(0,-∞,-1,{5,6})
3	insert(8,280)					(0,-∞,-1,{5,6,7})
4	min	(6,210)	2			(6,210,4,{5,6,8})
5	insert(9,190)					(6,210,4,{5,6,8,9})
6	min	(9,190)	5		(9,190,6,$\emptyset$),	(6,210,4,{5,6,8,9})
7	insert(2,275)				(9,190,6,{2}),	(6,210,4,{5,6,8,9})
8	delete(8)		3		(9,190,6,{2}),	(6,210,4,{5,6,9})
9	insert(12,170)				(9,190,6,{2,12}),	(6,210,4,{5,6,9})
10	insert(14,400)				(9,190,6,{2,12,14}),	(6,210,4,{5,6,9})
11	deletemin	(12,170)	9	(12,170,11,$\emptyset$),	(9,190,6,{2,14}),	(6,210,4,{5,6,9})
12	insert(3,290)			(12,170,11,{3}),	(9,190,6,{2,14}),	(6,210,4,{5,6,9})
13	insert(7,330)			(12,170,11,{3,7}),	(9,190,6,{2,14}),	(6,210,4,{5,6,9})
14	insert(15,200)			(12,170,11,{3,7,15}),	(9,190,6,{2,14}),	(6,210,4,{5,6,9})
15	delete(9)		5	(12,170,11,{3,7,15}),	(9,190,6,{2,14}),	(6,210,4,{5,6})
16	deletemin	(15,200)	14		(15,200,16,{2,3,7,14}),	(6,210,4,{5,6})
17	delete(7)		13		(15,200,16,{2,3,14}),	(6,210,4,{5,6})
18	deletemin	(6,210)	2			(6,210,18,{2,3,5,14})
19	delete(14)		10			(6,210,18,{2,3,5})
20	deletemin	(2,275)	7			(2,275,20,{3,5})
21	deletemin	(3,290)	12			(3,290,21,{5})
22	deletemin	(5,310)	1			(5,310,1,$\emptyset$)

Table 3: Sequence of Priority Queue operations illustrating answer validation algorithm

Initially, *AnswerStack* contains the single quadruple $(0, -inf, -1, null)$, where $(0, -inf)$ is guaranteed to be smaller than any pair.

We now describe and present pseudo-code for the answer validation algorithm. As for the previous algorithm, this consists of routines for each data structure operation and a FinalPhase routine. The variable *CurrentTime* is incremented after each operation. An example of these routines is presented in table 3.

insert(i,v): This is the same as the previous insert algorithm with the additional step of adding i to the set s belonging to the top stack element.

```
insert(i,v)     /* $(i, v)$ is the pair to be inserted */
{
    if ($InsertTime[i] \neq unused$ ) output "ill-formed" and halt
    $InsertTime[i] = CurrentTime$
    $Value[i] = v$

    add(top($AnswerStack$), $i$)
}
```

(i,v) = min(): Perform the same steps as for the previous algorithm. In addition, take the union of the sets contained in stack elements that were popped of the stack and assign this to the fourth element of the quadruple pushed on the stack.

```
min($i_1$,$v_1$) /* $(i_1, v_1)$ is the answer given in the input for this min */
{
    if ($InsertTime[i_1] = unused$ or $Value[i_1] \neq v$)
        output “ill-formed” and halt

    $s_1 = null$

    while (not empty($AnswerStack$)) {
        $(i_2, v_2, atime_2, s_2)$ = top($AnswerStack$)
        if $(i_1, v_) \leq (i_2, v_2)$
            pop($AnswerStack$)
            $s_1 = s_1$ union $s_2$
        else if ($InsertTime[i_1] < atime_2$) output “incorrect” and halt          (1)
        else e}xit from while loop
    push(AnswerStack, i, k, CurrentTime, s')
}
```

(i,v) = deletemin: We perform the same operations as for min. In addition, we remove the item number i from the set containing it and set *InsertTime*[i] to *unused.*

```
deletemin(i,v) /* (i,v) is answer */
{
    min($i$,$v$)
    remove($i$)
    $InsertTime[i] = unused$
}
```

delete(i_1)**:** First, check that there is a pair (i_1, v_1) in the priority queue. If not, output “ill-formed” and halt. Otherwise, let $(i_2, v_2, atime_2, s_2)$ be the stack element with s_2 containing i_1. Remove i_1 from s_2. Now, if the pair (i_1, v_1) is smaller than (i_2, v_2) check that it wasn't inserted until after the answer (i_2, v_2) was given. If not, output “incorrect” and halt. Next, if $(i_2, v_2, atime_2, s_2)$ is the top stack element we are done. Otherwise let $(i_3, v_3, atime_3, s_3)$ be the element immediately above $(i_2, v_2, atime_2, s_2)$. If (i_1, v_1) is smaller than (i_3, v_3) we output “incorrect” and halt. Otherwise the operation succeeds.

```
delete($i_1$)
{
    if ($InsertTime[i_1] = unused$) output “ill-formed” and halt

    $v_1 = Value[i_1]$
```

```
        $(i_2, v_2, atime_2, s_2)$ =find($i_1$)
        remove($i_1$)

        if $atime_2 > InsertTime[i_1]$ and $(i_1, v_1) < (i_2, v_2)$          (2)
            output "incorrect" and halt
        if $atime_2 < InsertTime[i_1]$ and !istop($(i_2, v_2, a_2, s_2)$) {
            $(i_3, v_3, a_3, s_3)$ =up($i_2$,$v_2$,$a_2$,$s_2$)
            if $(i_1, v_1) < (i_3, v_3)$ output "incorrect" and halt       (3)
        }
}

FinalPhase() /* executed after all operations have been performed */
{
    Form triples $(i_1, v_1, InsertTime[i_1])$ for each pair remaining in the queue.
    RemainderList = these triples sorted by $InsertTime[i]$
    For each $(i_2, v_2, atime_2, s_2)$ on AnswerStack {
        If there is an $(i_1, v_1, InsertTime[i_1])$ on RemainderList s.t.
            $InsertTime[i_1] < atime_2$ and
            $(i_1, v_1) < (i_2, v_2)$ then output "incorrect" and halt      (4)
    }
    output "correct"
}
```

7.3 Stack Properties

Lemma 7.1 *The three properties from before still hold. In addition the following properties hold:*

4. The union of the sets in the stack consists of the set of item numbers of pairs in the priority queue.

5. Given adjacent stack elements $(i_1, v_1, atime_1, s_1)$ above $(i_2, v_2, atime_2, s_2)$, for any i_3 in s_2, $InsertTime[i_3] < atime_1$. For any i_3 in s_1, $InsertTime[i_3] > atime_2$.

Proof:

The proof from the previous section applies to the stack in this algorithm and demonstrates the three previous properties.

Property 4 is easy to demonstrate. An insert instruction adds the appropriate item number to the top element. A delete or deletemin removes the appropriate item number from its containing set. A min or deletemin operation that pops stack elements will push a stack element with a set consisting of the union of popped sets, so the union of all sets on the stack does not change.

Consider the first part of property 5. There are two ways that i_3 could have been placed in s_2. First, the pair (i_3, v_3) could have been inserted while $(i_2, v_2, atime_2, s_2)$ was the top element, or s_2 could have initially been formed by the union of sets, one of which contained

i_3. In either case, there is some time t that is the earliest at which $(i_2, v_2, atime_2, s_2)$ was the top stack element and i_3 was in s_2. Note that a stack element that is not the top of stack can never become the top of stack since the only operations that remove elements from the stack are min and deletemin and they end by pushing a new quadruple. Thus $(i_1, v_1, atime_1, s_1)$ must have been pushed onto the stack at some time after t, since eventually it is immediately above $(i_2, v_2, atime_2, s_2)$. Therefore $InsertTime[i_3] < atime_1$.

The second part of property 5 is simpler. Item numbers may only be placed in the set on top of the stack (either by inserting into an existing set or from a merge forming a new set). Thus i_3 was either added to s_1 when $(i_1, v_1, atime_1, s_1)$ was already top of the stack, or it was placed in s_1 during the operation that pushed (i_1, v_1, a_1, s_1). In the former case we have $InsertTime[i_3] > atime_1 > atime_2$. In the latter case, some stack element $(i_4, v_4, atime_4, s_4)$ was originally on the top of the stack at time $InsertTime[i_3]$ and must have been above $(i_2, v_2, atime_2, s_2)$ (thought not necessarily immediately above it). Thus $InsertTime[i_3] > atime_4 > atime_2$. In either case case the second half of property 5 holds. ∎

7.4 Proof of correctness

Theorem 7.2 *Theorem: The algorithm for answer validation of the priority queue terminates on all input. It outputs "correct" if the answers on the input are correct and "incorrect" or "ill-formed" if they are not.*

Proof: Clearly the algorithm terminates since each of the routines given above terminates. The initial checks against $InsertTime[]$ and $Value[]$ detect ill-formed input, so we will assume that the input is well formed.

Part 1: First we must show that if the input answers are correct, then the algorithm will output "correct".

We now check that min, deletemin, and delete operations do not fail on correct input. The checks given for min and deletemin are the same as for the simple priority queue, so the same reasoning implies that they will not fail. Examine check (2) in the delete routine. (i_1, v_1) is the pair being deleted $(i_2, v_2, atime_2, s_2)$ is he stack element s.t., s_2 contains (i_1, v_1), which must exist by stack property 4. We return "incorrect" if $atime_2 > InsertTime[i_1]$ and $(i_1, v_1) < (i_2, v_2)$. But this means that (i_2, v_2) was incorrectly given as an answer at time $atime_2$ since the smaller pair (i_1, v_1) was present in the priority queue at that time. Therefore this check cannot fail if all the answers are correct. Examine check (3). Let $(i_3, v_3, atime_3, s_3)$ be the stack element directly above $(i_2, v_2, atime_2, s_2)$, which must exist since we do not perform check (3) if $(i_2, v_2, atime_2, s_2)$ is the top element. The check returns "incorrect" if $(i_1, v_1) < (i_3, v_3)$.

By stack property 5, $InsertTime[i_1] < atime_3$. This if this check fails, the answer (i_3, v_3) given at $atime_3$ is incorrect since the smaller pair (i_1, v_1) is in the priority queue at that time.

Finally, the FinalPhase routine is identical to that for the simple priority queue, so the same argument shows that check (4) cannot fail.

Part 2: Now we must prove that the if any input answers are not correct, the algorithm will output "ill-formed" or "incorrect".

Again, the initial checks against *InsertTime*[], and *Value*[] check for ill-formed input, so we may assume that the input is well-formed.

The proof is similar to the earlier proof. Let t_{wrong} be the time of the first operation with an incorrect answer. Let (i_1, v_1) be the pair that is the correct answer for that operation. Let t_{ins} be the time of the last insert(i_1,v_1) operation occurring before t_{wrong} Let t_{del} be the time of the first delete(i_1) operation or deletemin operation with answer (i_1, v_1) occurring after t_{wrong}. If there is no such operation, let t_{del} = *infinity*. Then during our execution of the answer validation algorithm, (i_1, v_1) is marked as being in the queue between times t_{ins} and t_{del}. Note that in a correct execution of the operations this might not be the case.

Let $(i_2, v_2, atime_2)$ be the largest answer triple occurring with $t_{ins} < atime_2 < t_{del}$. We know that $(i_2, v_2) > (i_1, v_1)$ because it (i_2, v_2) is at least as large as the incorrect answer given at time t_{wrong}. There may be several operations during that time period that return (i_2, v_2), but our ordering on triples guarantees that we pick the last such operation.

There are now three cases, the first two of which are identical to the simple priority cases.

Case I: (i_1, v_1) returned as an answer at some time $atime_1 > atime_2$. This case is identical to case I for simple priority queues.

Case II: (i_1, v_1) is not deleted and is never returned as an answer. This is identical to case II for simple priority queues.

Case III: (i_1, v_1) is not returned as answer before or at time t_{del}, but is deleted from the priority queue at time t_{del}. Note that some another instance (i_1, v_1) could be inserted and returned as an answer after t_{del}. This is irrelevant.

At time t_{del}, $(i_2, v_2, atime_2, s_2)$ must be on the stack, since it is the largest answer triple with time larger than t_{ins}.

If i_1 is in s_2 then check (2) will fail, i.e., cause "incorrect" to be output, since $(i_1, v_1) < (i_2, v_2)$.

If not, let $(i_3, v_3, atime_3, s_3)$ be the stack element s.t. i_1 is in s_3. $InsertTime[i_1] < atime_2$, so this stack element must be below (not necessarily immediately below) $(i_2, v_2, atime_2, s_2)$, for otherwise stack property (5) would require $InsertTime[i_1] > atime_2$. Suppose check (2) succeeds, that is, does not cause "incorrect" to be output. Then either $InsertTime[i_1] > atime_3$ or $(i_3, v_3) < (i_1, v_1)$. However $(i_3, v_3) > (i_2, v_2)$ by stack property 2, so we must have $InsertTime[i_1] > atime_3$, in which case check (3) will be performed.

Let $(i_4, v_4, atime_4, s_4)$ be the stack element immediately above $(i_3, v_3, atime_3, s_3)$, which must exist since $(i_2, v_2, atime_2, s_2)$ is above $(i_3, v_3, atime_3, s_3)$.

Then we have $InsertTime[i_1] < a_4$, by stack property 5, but $(i_1, v_1) < (i_2, v_2) < (i_4, v_4)$ by stack property 2, so check (3) will fail. ∎

There is an important subtlety in the above theorem. We have not shown that all priority queue operations were correctly performed, only that the answers given are the same as those that would have been given if all operations had been correcty performed. In particular, since delete operations do not return answers, it is possible that a delete operation removed the wrong element during the original execution of operations. If that error does not affect the answers to the other operations, we will not detect it.

Note also that we could define the delete the operation to return the pair being deleted. It is trivial to modify the procedure for delete in the above algorithm to validate those answers.

8 Generalized Priority Queue

We can define max and deletemax operations analogous to the min and deletemin operations defined previously. A *generalized priority queue* is a structure supporting the priority queue operations defined in the previous section and the operations max and deletemax. As before, the changekey(i,w) operation may be implemented as a delete(i) followed by an insert(i,w) so for simplicity we do not consider it in the material below.

It is obvious that the technique in the preceding section provides linear time validation for the operations insert, delete, max, and deletemax. We now show how to validate the generalized priority queue operations.

Definition 8.1 Consider a sequence of generalized priority queue operations together with the supposed answers. Based on this sequence we derive a new sequence of operations called the *minimum sequence*. This sequence is derived from the original sequence by:

i. Removing every max operation and the corresponding answer.

ii. Replacing every deletemax operation and corresponding answer by a delete(i) operation, where i is the item number given in the answer to the deletemax operation.

Every other operation from the original sequence is copied to the minimum sequence without change. We say that two operations, O_1 the original sequence and O_2 from the minimum sequence, are corresponding operations if

i. The operation O_1 is a deletemax operation, O_2 is a delete operation, and O_2 was created from O_1 by the replacement rule given above. OR,

ii. O_1 and O_2 are the same operation, with the same arguments but not necessarily the same answer, and O_2 is the unchanged copy of O_1.

The *maximum sequence* is defined analogously.

Theorem 8.2 *Let S be a sequence of generalized priority queue operations with supposed answers. Let S_{min} and S_{max} be the minimum and maximum sequences, respectively. Then the answers in S are correct iff, the answers on both S_{min} and S_{max} are correct.*

Proof:

Let P_i be the set of elements in the priority queue after correct execution of the first i operations in S. Similarly, define $P_{min,i}$ and $P_{max,i}$ for the sequences S_{min} and S_{max}. We shall show that if O_1 and O_2 are corresponding operations in S and S_{min}, then P_{O_1} and P_{min,O_2} contain the same elements. The same statement is true for corresponding operations in S and S_{max}.

Suppose that this is true for all operation before O_1 in S and the corresponding operation O_2 in S_{min}. The if O_1 is an insert operation, O_2 will insert the same element so the two queues will still store identical sets. Similarly if O_1 and O_2 are corresponding deletes or deletemins, the same element will be deleted from each queue. Finally, if O_1 is a deletemax operation, the derivation of S_{min} guarantees that the same element is removed by the corresponding delete operation O_2. No other operations change the contents of the queues, so we have shown that P_{O_1} and P_{min,O_2} contain the same elements. The proof for S and S_{max} is analogous.

Now, suppose that the answers given in S are correct. This means that they are the answer that would be given by a correct execution of the operations in S. Since P_{O_1} and P_{min,O_2} contain the same elements after corresponding operation O_1 and O_2, the correct answers to corresponding min and deletemin operations must be the same. Thus, the answers on the sequence S_{min} are correct. Similarly for S_{max}.

Now, suppose that the answers on both S_{min} and S_{max} are correct. We must show that the answers given in S are correct. We know that P_{O_1} and P_{min,O_2} contain contain the same elements after corresponding operations O_1 and O_2. Similarly P_{O_1} and P_{max,O_3} contain the same elements after corresponding operations O_1 and O_3.

Suppose that the answers in S are correct up to operation O_1. We may assume that O_1 is a min, max, deletemin, or deletemax operation since the other operations do not return answers. Suppose O_1 is a min or deletemin operation. Since the priority queues $P_{O_1 - 1}$ and P_{min,O_2-1} contain the same elements, the correct answer for both min operations must be the same. Therefore the answer given in S for O_1 is correct because it is the same as the answer given in S_{min} for O_2.

Similarly, if O_1 is a max or deletemax operation, then the priority queues associated with S and S_{max} will contain the same elements before O_1 and the corresponding operation O_3. The correct answers to those operations must therefore be the same, so the answer given in S is correct.

Therefore all answers given in S are correct.

∎

Note that neither of the sequences S_{min} nor S_{max} is sufficient by itself.

Corollary 8.3 *The set of answers to generalized priority queue operations (insert, delete, min, deletemin, max, deletemax) may be validated in linear time.*

Proof: Clearly the minimum and maximum sequences can be formed in linear time. The algorithm in the preceding section may be used to validate the answers in the minimum and maximum subsequences in linear time. ∎

9 Experimental Results

In this section we evaluate the use of certification trails for data structures as applied to four well-known and significant problems in computer science: sorting, the shortest path tree problem, the Huffman tree problem, and the skyline problem. We have implemented basic algorithms for these problems and applied the techniques described in Section 4 to implement algorithms which generate and use certification trails. Timing data was collected using a SPARCstation ELC running SunOS 4.1.

The timing information reported in the tables consists of the run time of the basic algorithm (i.e., no certification trail), the run time of the trail-generating algorithm, the run time of the trail-using algorithm, the percentage savings of using certification trails, and the speedup achieved by the second phase of the certification trail method. The percentage savings is computed by comparing the total run time of algorithms for generating and using trails against twice the run time of the basic algorithm. The speedup is computed by dividing the run time of the basic algorithm by the run time of the algorithm that uses the certification trail.

Apart from the data structures, the implementation of both phases of the certification trail version of each algorithm is nearly identical to the implementation of the basic version. The only difference in the code for the two phases is a parameter passed to the data structure code indicating whether a certification trail should be generated or used. All code implementing the certification trails is localized to the modules implementing the data structures, allowing the generation and use of the trail to be transparent to the user of these modules. Due to space constraints only an abbreviated discussion of the algorithms is given.

9.1 Heapsort

Sorting is a fundamental operation in computer systems, and there exist several sorting algorithms. Sorting may be implemented with a priority queue (or more specifically, a heap) by inserting all elements and performing deletemin operations until the queue is empty.

Input data was generated by creating sets of integers chosen uniformly from the interval $[0, 10000000]$. Timing results are based on fifty executions at each input size.

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
10000	0.44	0.45	0.11	4.00	36.36
20000	0.98	1.00	0.23	4.26	37.24
50000	2.71	2.80	0.60	4.52	37.27
100000	5.87	6.05	1.23	4.77	37.99
200000	12.71	12.91	2.47	5.15	39.50
300000	19.67	20.25	3.73	5.27	39.04

Table 4: Heapsort

9.2 Huffman Tree

Given a sequence of frequencies (positive integers), we wish to construct a Huffman tree, i.e., a binary tree with frequencies assigned to the leaves, such that the sum of the weighted path lengths is minimized. This is a classic algorithmic problem and one of the original solutions was found by Huffman [13]. It has been used extensively in data compression algorithms through the design and use of so called Huffman codes. The tree structure and code design are based on frequencies of individual characters in the data to be compressed. In this paper we are concerned only with the Huffman tree, the interested reader should consult [13] for information about the coding application.

The Huffman tree is built from the bottom up and the overall structure of the algorithm is based on the greedy "merging" of subtrees. An array of pointers, ptr, is used to point to the subtrees as they are constructed. Initially, n single vertex subtrees are constructed, each one associated with a frequency number in the input. The algorithm repeatedly merges the two subtrees with the smallest associated frequency values, assigning the sum of these frequencies to the resulting tree. A priority queue data structure allows the algorithm to quickly find the subtrees to merge at each step.

Data for the timing experiments was generated by choosing integer frequencies uniformly from the range $[0, 100000]$. Timing results are based on fifty executions for each input size.

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
5000	0.38	0.41	0.14	2.71	27.63
10000	0.83	0.87	0.29	2.86	30.12
20000	1.79	1.90	0.61	2.93	29.89
50000	4.93	5.30	1.53	3.22	30.73
100000	10.75	11.47	3.12	3.45	32.14
150000	16.70	17.87	4.66	3.58	32.54

Table 5: Huffman Tree

9.3 Shortest Path

Given a graph with non-negative edge weights and a source vertex, we wish to find the shortest paths from the source vertex to each of the other vertices. This is another classic problem and has been examined extensively in the literature. Our approach is applied to Dijkstra's algorithm.

Dijkstra's algorithm is a greedy algorithm. At each step, there exists a set of vertices S to which shortest paths are known, and a set T of vertices adjacent to members of this set. The best paths known to members of T are examined, and the vertex v, with the minimum path length is removed from T and added to S. A data structure that supports insert, delete, and deletemin can be used to implement this algorithm.

Input graphs of $|V|$ vertices and $|E|$ edges were generated by choosing a set of $|E|$ distinct edges uniformly from all possible such sets, then rejecting graphs that were not connected. $|E|$ was chosen sufficiently large that each selection is connected with high probability, resulting in few rejections. The input sizes were chosen to keep the ration $|E|/|V|$ constant, for in practice the running time of the algorithm is affected by this ratio. Timing results are based on fifty executions at each input size. The size column of Table 6 contains an ordered pair indicating the number of vertices and edges.

9.4 Skyline

Given a set of rectangles with with collinear bottom edges, the *skyline* is the figure resulting from removing all hidden edges. The problem of computing the skyline of a set of rectangular

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
500,5000	0.38	0.41	0.22	1.73	17.11
1000,10000	0.86	0.91	0.45	1.91	20.93
1500,15000	1.39	1.48	0.69	2.01	21.94
2000,20000	1.94	1.97	0.90	2.16	26.03

Table 6: Shortest Path

buildings by eliminating hidden lines is discussed in [15]. The method used is divide and conquer and it constructs a skyline in $O(n \log(n))$ time. In this paper we use a plane sweep algorithm that can be easily implemented in terms of operations on priority queues. Plane sweep algorithms are widely used for computational geometry problems [16], and typically use a priority queue for event scheduling, and may be amenable to use of certification trail techniques.

Using a plane sweep algorithm, we compute the skyline as follows. Initialize a vertical sweep line to the left of all the rectangles (we may assume that all rectangle are to the right of the y-axis). As we sweep the line to the right we maintain a collection of the heights of the rectangles encountered. For each rectangle R, the height of R is added to the collection when we encounter R's left edge and removed when we encounter its right edge. The height of the skyline at any point x_0, is the maximum height in the collection when the sweepline is at $x = x_0$. Details are given below. A structure supporting insert and deletemin is all that is needed to order the events, and a structure supporting insert, max, and delete is required to store the rectangle heights. A priority queue (supporting insert and can be used to order the sweepline events, and a generalized priority queue to store the rectangle heights.

Input data was generated by choosing integral rectangle heights uniformly over the range $[0, 100000]$. The x-coordinates of the left edges were chosen uniformly over the range $[0, 90000]$ and the width of each rectangle was chosen uniformly over the range $[1, 10000]$. Timing results are based on twenty executions for each input size.

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
1000	0.25	0.27	0.11	2.27	24.00
2000	0.56	0.59	0.22	2.55	27.68
5000	1.71	1.79	0.58	2.95	30.70
10000	3.86	4.01	1.17	3.30	32.90
20000	8.39	8.76	2.36	3.56	33.73
30000	13.29	14.02	3.55	3.74	33.90

Table 7: Skyline

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] Andrews, D., "Software fault tolerance through executable assertions," *Rec. 12th Asilomar Conf. Circuits, Syst., Comput.*, pp. 641-645, 1978, Nov. 6-8.

[4] Andrews, D., "Using excutable assertions for testing and fault tolerance," *Dig. 9th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 102-105, 1979, June 20-22.

[5] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[6] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[7] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[8] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[9] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

[10] Fredman, M. L., and Saks, M. E., "The cell probe complexity of dynamic data structures," *Proc. 21st ACM Symp. on Theo. Comp. 1989*, pp. 109-122, 2, 1986.

[11] Gabow, H. N., and Tarjan, R. E., "A linear-time algorithm for a special case of disjoint set union," *J. of Comp. and Sys. Sci.*, 30(2), pp. 209-221, 1985.

[12] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[13] Huffman, D., "A method for the construction of minimum redundancy codes", *Proc. IRE*, pp 1098-1101, 40, 1952.

[14] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[15] Manber U., *Introduction to Algorithms: A Creative Approach* Addison-Wesley, Reading, MA, 1989.

[16] Preparata F. P., and Shamos M. I., *Computational geometry: an introduction*, Springer-Verlag, New York, NY, 1985.

[17] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[18] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[19] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[20] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Department of Computer Science Technical Report JHU 90/17*, Johns Hopkins University, Baltimore, Maryland, 1990.

[21] Tarjan, R. E., "Efficiency of a good but not linear set union algorithm," *J. ACM*, 22(2), pp. 215-225, 1975.

[22] Tarjan, R. E., "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. of Comp. and Sys. Sci.*, 18(2), pp. 110-127, 1979.

[23] Tarjan, R. E., and Leeuwen, J. van, "Worst-case analysis of set union algorithms," *J. ACM*, 31(2), pp. 245-281, 1984.

[24] Taylor, D., "Error Models for robust data structures," *Dig. 20th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 416-422, 1990 June 26-28.

[25] Williams, J. W. J, "Algorithm 232 (heapsort)," *Commun. of ACM*, vol.7, pp. 347-348, 1964.