Answer Validation for Abstract Data Types *

Dwight S.Wilson, Sophie Y. Qiu, Gregory F. Sullivan, and Gerald M. Masson
Department of Computer Science, Johns Hopkins University
Baltimore, MD 21218

Abstract

Run-time certificates can be used to efficiently check the correctness of computational results with relatively small time and software redundancy. However, certificates heretofore have had to be customized to specific algorithms. In this paper we provide both theoretical and experimental results regarding the use of certificates for wide classes of algorithms based on common data-structure operations. Using an answer-validation monitoring approach, certificates can be created for a variety of algorithms. Specifically, this paper discusses answer-validation monitoring of disjoint-set-union and priority-queue operations. The correctness of the answer validation approach is rigorously demonstrated. Experimental results are presented for four well-known algorithms using priority queues: shortest path, heapsort, Huffman tree, and skyline. It will be seen that the certificate utilized for each algorithm is generic in the sense that it is applicable to any algorithm using a priority queue. The experimental results compare the certificate approach using answer-validation monitoring to a two-version programming approach. It is seen that the certificate-based approach has significantly lower overhead. Furthermore, the advantage of using the certificate approach becomes greater as the size of the algorithm input increases, making the approach particularly attractive for low-overhead enhancement of the dependability of operations associated with computationally intensive applications.

Keywords: Answer validation, abstract data type, run-time certificate, priority queue

1 Introduction

Software must be highly reliable, continuously available, and extremely safe [1]. However, the complexity of software continues to increase, and correspondingly the existence of software faults during system operation also increases, which in turn makes it very difficult and costly – if not infeasible – to perform comprehensive and

* Research partially supported by NSF Grants CCR-8910569 and CCR-8908092

rigorous fault avoidance or fault removal techniques. Indeed, even though such techniques might be employed exhaustively during software development phases, there is still no guarantee that the software is not error-prone, since, for example, many run-time faults are transient and unpredictable regarding their sources, and therefore might only manifest themselves during run-time operations. As a result, practical software systems can rarely be assured to be free of faults or totally resilient to error-prone behavior. Accordingly, various software fault tolerance techniques have gained importance in applications of dependable systems because of their purported capabilities in delivering acceptable results in the presence of software/hardware faults (especially those of a transient nature). Such fault tolerance techniques can dynamically evaluate answers provided during the run time of software applications to determine whether correct results are being provided.

As examples of software fault tolerance techniques, the recovery block approach [2] uses acceptance tests and alternative procedures to produce a correct output from a program, which is structured into blocks of operations. Formal methodologies for the definition and generation of acceptance tests, however, have not been fully established. A program result checking approach [3] develops an algorithm to check the output of another algorithm for correctness and thus is similar to an acceptance test in a recovery block. The checking algorithm may be called multiple times, based on the input/output specifications of a problem only, not the specific algorithm being checked. Another popular software fault tolerance approach is N-version programming [4, 5, 6], where N functionally equivalent software versions are independently developed and executed given the same input, and their outputs are subsequently compared to detect faults. Like result checking, N-version programming has large time and software redundancy.

A different yet more general and efficient fault tolerance approach has been described in [7, 8] using the concept of a *certificate.* In essence, a primary program is modified to not only produce the results, but also leave behind a trail of information, referred to as a certificate. A secondary program takes the same input as the primary program plus the certificate and either produces the same results if they are correct or indicates that an error has occurred in the primary program. The certificate has to be customized for each specific algorithm and be extracted again for other algorithms, even if they are implemented using the same data structure.

In this paper, we formulate an *answer validation* problem for abstract data types that can be used by a wide range of algorithms (i.e., not restricted for use with a specific algorithm). We then describe a two-phase schema to solve this problem by using a significantly generalized certificate-based approach in the system that was initially introduced in [7]. We develop this schema for disjoint-set-union and priority queue abstract data types, and further demonstrate its effectiveness and efficiency in such algorithm applications as shortest path, heapsort, Huffman tree, and skyline that utilize priority queue data structures. Our empirical evidence shows that the certificate-based approach, without a need of customizing the certificates to any of the four specific algorithms being considered but enabling them to be generally applicable to any algorithms using priority queues, can achieve up to 3.5 folds of run-time speed-up in comparison to the two-version programming approach for validating the answers of various

algorithms with abstract data types. Moreover, as the size of the data sets employed in the algorithms increases, the advantage of certificate-based approach is even greater, making itself particularly attractive for low-overhead enhancement of the dependability of operations associated with computationally intensive applications.

The rest of this paper is organized as follows: Section 2 reviews the concept of two-version programming and certificate-based approaches. Section 3 formulates and solves the answer validation problem for abstract data types by using certificate, based on an example of disjoint-set-union data type. Section 4 and Section 5 illustrate the application to answer validation of priority queue data type. Section 6 compares the experimental results of answer validation for shortest path, heapsort, Huffman tree, and skyline problems using certificate with those using two-version programming. Section 7 concludes this paper with discussions on future work.

2 Two-Version Programming and Certificate-based Approach

2.1 Two-Version Programming

Without loss of generality, we consider N-version programming with $N = 2$ for the simplicity of presentation. Two-version programming consists of two independently developed programs to solve the same problem and compare the two outputs generated for a given input. If the two outputs are equal then the output is correct. If the two outputs are different then an error has been detected. It is easy to see that two-version programming has both time and software redundancy. In addition, there might be errors during both executions which cause both outputs to be incorrect, and yet equal coincidentally. Therefore, we must constrain the class of errors under consideration and one natural constraint states that errors are restricted to modify only the execution of one of the programs. The material below presents these ideas with more precision.

Definition 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 *Let* $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ *be a problem. A solution to this problem using two-version programming consists of algorithms implementing two functions* F_1 *and* F_2 *with the following domains and ranges* $F_1 : \mathbf{D} \rightarrow \mathbf{S}$ *and* $F_2 : \mathbf{D} \rightarrow \mathbf{S}$. *The functions must satisfy the following three properties:*

(1) for all $d \in \mathbf{D}$ *there exists* $s \in \mathbf{S}$ *such that* $F_1(d) = s$ *and* $(d, s) \in \mathbf{P}$
(2) for all $d \in \mathbf{D}$ *there exists* $s \in \mathbf{S}$ *such that* $F_2(d) = s$ *and* $(d, s) \in \mathbf{P}$
(3) for all $d \in \mathbf{D}$ $F_1(d) = F_2(d)$

Hence, F_1 and F_2 both solve problem P and they agree on the answers even when multiple answers are possible. Note that this means F_1 and F_2 are the same function. We use two function names in order to allow the careful statement of the theorem given below. The theorem also uses variant functions which are marked with primes. These primed functions are intended to capture the notion of the occurrence of errors.

Theorem 1 *Let F_1 and F_2 satisfy the definition of 2-version programming.*

Let F_1' be an arbitrary function with domain $\mathbf{D}$ and range $\mathbf{S}$.

For all $d \in \mathbf{D}$

let $F_1'(d) = s'$ and $F_2(d) = s$

if $s = s'$ then $(d, s') \in \mathbf{P}$ i.e., answer is correct

if $s \neq s'$ then $F_1'(d) \neq F_1(d)$ i.e., error is detected

Let F_2' be an arbitrary function with domain $\mathbf{D}$ and range $\mathbf{S}$.

For all $d \in \mathbf{D}$

let $F_1(d) = s$ and $F_2'(d) = s'$

if $s = s'$ then $(d, s') \in \mathbf{P}$ i.e., answer is correct

if $s \neq s'$ then $F_2'(d) \neq F_2(d)$ i.e., error is detected

Proof: Part One: Let $d \in \mathbf{D}$ be an arbitrary element of the domain, let $F_1'(d) = s'$, and let $F_2(d) = s$. By property (2) of the definition of two-version programming, $(d, s) \in \mathbf{P}$. Now, suppose $s = s'$ then $(d, s') \in \mathbf{P}$, and, in this case, the answer specified by either s or s', is correct. Suppose instead $s \neq s'$. By property (3) of the definition of two-version programming, $F_1(d) = F_2(d)$. But, $F_2(d) = s$, $F_1'(d) = s'$ and $s \neq s'$, so, $F_1'(d) \neq F_1(d)$. In this case, an error has been detected.

Part Two: Let $d \in \mathbf{D}$ be an arbitrary element of the domain, let $F_1(d) = s$, and let $F_2'(d) = s'$. By property (1) of the definition of two-version programming, $(d, s) \in \mathbf{P}$. Now, suppose $s = s'$ then $(d, s') \in \mathbf{P}$, and, in this case, the answer specified by either s or s', is correct. Suppose instead $s \neq s'$. By property (3) of the definition of two-version programming, $F_1(d) = F_2(d)$. But, $F_1(d) = s$, $F_2'(d) = s'$ and $s \neq s'$, so, $F_2'(d) \neq F_2(d)$. In this case, an error has been detected.

Many elements can operate incorrectly when a computer system composed of hardware and software is used to solve a problem. For example, there might be a hardware design error, or a software design error. There might be a hardware fault during execution, or the operating system might malfunction during execution. Further, any combination of these difficulties might occur. In the theorem above, we used primed functions to model the effect of these myriad possible deviations from correct functioning. Informally, the first part of the theorem concerns the

situation when F_2 is computed correctly, yet anomalous system behavior causes F_1 to be computed incorrectly as F_1'. In this case, the comparison test performed in two-version programming allows a given correct answer to be determined for a given input, or it allows the detection of the error causing behavior. The second part of the theorem is similar, but concerns the situation when F_1 is computed correctly, yet F_2 is computed incorrectly as F_2'. In this case, the comparison test also allows a given correct answer to be determined, or it allows the detection of the error.

2.2 Certificate-based Approach

It is also natural to just try to modify one of the two programs to make the task of determining the correctness of its output easier. For example, we suggest modifying a program so that it generates additional information (a trail of data) which we call a *certificate.* The central idea is to modify the first algorithm so that it leaves behind a certificate such that it can allow the the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. This mechanism improves the efficiency and simplicity of ascertaining the correctness of the output answer. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, that we must be careful in defining this approach 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 certificate 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 certificate 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 certificate 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 certificate.

The formal definition below describes the modification of a program so that it generates a certificate. Refer to the previous section for the definition of problem $\mathbf{P}$.

Definition 3 *Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a certificate consists of algorithms implementing two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{C}$ and $F_2 : \mathbf{D} \times \mathbf{C} \rightarrow \mathbf{S} \cup \{error\}$. $\mathbf{C}$ is the set of certificates. The symbol $error$ is chosen such that $error \notin \mathbf{S}$. The functions must satisfy the following two properties:*

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{C}$ 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 $t \in \mathbf{C}$
either ($F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$) or $F_2(d, t) = error$.

In contrast with two-version programming where each version of the algorithm is executed independently without any information about the execution of the other algorithm and there is no relationship between the executions of two different versions of the algorithm other than they both use the same input, the certificate-based approach allows the primary systems to generate a trail of information while executing its algorithm that is critical to the secondary system's execution of its algorithm. Moreover, the certificate is designed to allow the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. This mechanism helps reduce the time and hardware redundancy associated with two-version programming approach. It also expends fewer resources. We will further show that both approaches have similar error-detection capabilities.

On the other hand, N-version programming can be in effect thought of relative to the certificate-based approach as the employment of a *null certificate.* The informal interpretation of the following theorem is very similar to the one given for the two-version programming theorem. The first part of the theorem concerns the situation when F_2 is computed correctly, yet F_1 is computed incorrectly as F_1'. In this case, the comparison test allows a correct answer to be determined, or it allows the detection of an error. The second part of the theorem concerns the situation when F_1 is computed correctly, yet F_2 is computed incorrectly as F_2'. Once again, the comparison test allows the determination of a correct answer or the detection of an error.

Theorem 2 *Let F_1 and F_2 satisfy the definition of a certificate.*

Let F_1' be an arbitrary function with domain $\mathbf{D}$ and range $\mathbf{S} \times \mathbf{C}$.

For all $d \in \mathbf{D}$

let $F_1'(d) = (s', t')$ and $F_2(d, t') = s''$

if $s' = s''$ then $(d, s') \in \mathbf{P}$ i.e., answer is correct

if $s' \neq s''$ then $F_1'(d) \neq F_1(d)$ i.e., error is detected

Let F_2' be an arbitrary function with domain $\mathbf{D}$ and range $\mathbf{S} \cup \{error\}$.

For all $d \in \mathbf{D}$

let $F_1(d) = (s, t)$ and $F_2'(d, t) = s'$

if $s = s'$ then $(d, s') \in \mathbf{P}$ i.e., answer is correct

if $s \neq s'$ then $F_2'(d, t) \neq F_2(d, t)$ i.e., error is detected

Proof: Part One: Let $d \in \mathbf{D}$ be an arbitrary element of the domain, let $F_1'(d) = (s', t')$, and let $F_2(d, t') = s''$. Since F_1' has range $\mathbf{S} \times \mathbf{C}$ we know $s' \in \mathbf{S}$ and $t' \in \mathbf{C}$. By property (2) of the definition of certificate, either $(d, s'') \in \mathbf{P}$ or $s'' = error$. Now, suppose $s' = s''$ then either $(d, s') \in \mathbf{P}$ or $s' = error$. But $s' \in \mathbf{S}$ and the $error$

symbol is chosen to be outside $\mathbf{S}$, so $s' \neq error$. In this case, $(d, s') \in \mathbf{P}$ and the answer specified by either s' or s'', is correct.

Suppose instead $s' \neq s''$. Let $F_1(d) = (s, t)$ and suppose by way of contradiction that $F_1'(d) = F_1(d)$. This means $s = s'$ and $t = t'$. Further, $F_2(d, t') = F_2(d, t) = s$ by property (1) of the definition of certificate. Recall, $F_2(d, t') = s''$ which implies $s'' = s = s'$. This is a contradiction since we were assuming $s' \neq s''$. We conclude $F_1'(d) \neq F_1(d)$. Therefore, in this case, an error has been detected.

Part Two: Let $d \in \mathbf{D}$ be an arbitrary element of the domain, let $F_1(d) = (s, t)$, and let $F_2'(d, t) = s'$. By property (1) of the definition of certificate, $(d, s) \in \mathbf{P}$. Now, suppose $s = s'$ then $(d, s') \in \mathbf{P}$, and in this case, the answer specified by either s or s', is correct.

Suppose instead $s \neq s'$. By property (1) of the definition of certificate, $F_2(d, t) = s$. Since $F_2'(d, t) = s'$ and $s \neq s'$ we conclude $F_2'(d, t) \neq F_2(d, t)$. Hence, in this case, an error has been detected.

The theorem above is somewhat surprising, because it shows that the error detection properties of the certificate-based approach and two-version programming are very similar. This is despite the fact that additional data, the certificate, is being generated and used to enable faster and simpler computation. The certificate-based approach was deliberately defined to have desirable error-detection properties. We do not claim that the approach is a panacea. but we do believe that it always merits substantive consideration.

We also note that the theorems above do not capture all possible different types of error behavior. For example, a hardware or software fault might cause a program to enter an 'infinite loop' in which case no output would be generated. A fault might cause all available computing resources to be consumed, e.g., all free memory might be allocated. The output might be too large, e.g., a very large string might be generated. These problems are not precisely modeled by the variant primed functions used in the theorems above. But note that these behaviors are relevant to each of the error detection schemes we discuss: two-version programming, certificate, program checkers, and others. Clearly, there must be alternative error detection facilities such as watchdog timers for 'infinite loops', and resource utilization monitors. In addition, if a function value lying outside its proper range or domain is encountered then an error detection flag should be raised and execution should enter an error handling routine. These auxiliary detection mechanisms are needed for each of the approaches mentioned. Our proposal concentrates on the types of errors which we feel are prevalent, and which are the most difficult to detect and remove, i.e., the ones that result in variant functions.

Moreover, the theorems above do not explicitly treat the situation where errors occur during both executions. It is possible that the errors might cause the outputs of the two executions to differ, in which case the comparison test would still be able to detect the presence of errors. This desirable detection property is present in both two-version programming and certificate-based approaches. One can construct a probabilistic model to help explore this issue, but that is beyond the scope of this discussion. Nevertheless, we argue that the certificate-based approach can do

better than two-version programming approach in the aforementioned scenario. The reason is what follows: in the certificate-based approach, the second execution typically uses a different and simpler program than the first execution, and it can often be executed more quickly. Therefore, the probability of error detection can be high even when errors are present in both executions.

In [7, 8] the certificate-based fault tolerance approach uses time and software redundancy and can be outlined as follows. In the initial phase, a program is run to solve a problem and store the result. In addition, this program leaves behind a certificate. In the second phase, another program is run which solves the original problem again. This program, however, has access to the certificate left by the first program. Because of the availability of the certificate, the second phase can be performed by a less complex program and can execute more quickly. In the final phase, the two results are compared and if they agree the results are accepted as correct; otherwise an error is indicated. An essential aspect of this approach is that the second program must always generate either an error indication or a correct output even when the certificate it receives from the first program is incorrect. However, the above certificate can only be applied to a specific algorithm and has to be extracted again for other algorithms, even if they are implemented using the same data structure, which severely limits its use in practice. From the next section on, we shall generalize the above approach by developing certificate-based approach for a number of different algorithms that use the same abstract data types.

3 Answer Validation Problem for Abstract Data Types

3.1 Problem Formulation

Our general approach to applying certificates uses the concept of abstract data type. Each abstract data type has a well defined data object or a set of data objects, as well as a 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. Two of the priority queue operations are **insert(**i**,**k**)** and **deletemin**. The **insert** operation has two arguments: item number i and key value k. The **insert** operation does not return an answer. The **deletemin** operation has no arguments, but it does return an answer. The precise definition and semantics of these operations are given later in Section 4.

Intuitively, an *answer validation* problem consists of checking the correctness of a sequence of supposed answers to a sequence of operations performed on an abstract data type. For each abstract data type we formulate the answer validation problem as follows:

Formulation 1 *The input to the answer validation problem is a sequence of operations on the abstract data type together with the arguments of each operation. The sequence contains the supposed answers for each of the*

operations which return answers and each supposed answer is paired with the operation that is supposed to return it.[1] *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 will be the word "incorrect" if the answers do not match. The output will be the word "ill-formed" 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 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. In [9], for example, to calculate the answers to a sequence of n priority queue operations takes $\Omega(n \log(n))$ time. It is possible to check the correctness of the answers in only $O(n)$ time. This speed-up is very useful in transient-fault-detection applications.

In what follows we shall show how to create certificates for programs which use abstract data types when those data types have efficient solutions to their answer validation problems.

3.2 Answer Validation Schema Using Certificates

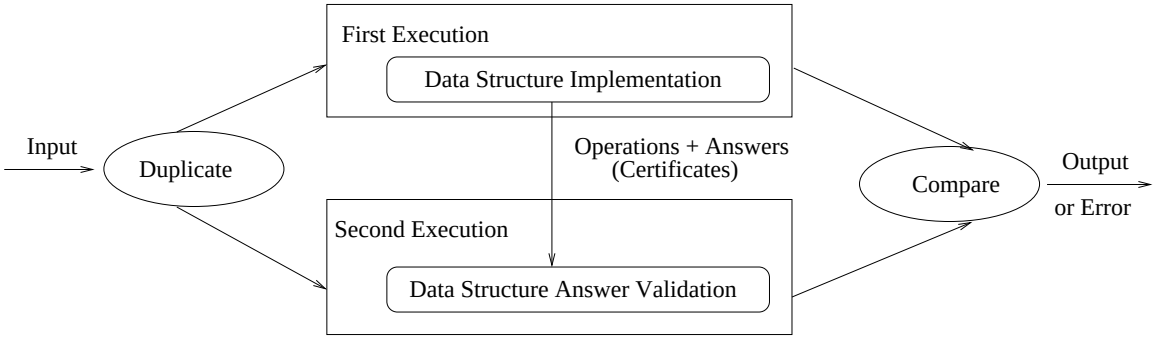

Figure 1: Answer validation schema

The answer validation schema is depicted in Figure 1. Suppose that we have developed an efficient solution to the answer validation problem for some abstract data type.[2] Further suppose that we wish to run an algorithm, say A, which uses that abstract data type. To apply the certificate-based approach we can use the following schema to yield the two executions:

First Execution:

[1] Examples of such inputs are given in the columns labeled "Operation" and "Answer" of Table 2.

[2] By "efficient" we mean that the time complexity of the answer validation problem is smaller than that of the original abstract-data-type problem.

Execute algorithm A. Each time an abstract-data-type operation is performed, append to the certificate the identity of the operation, the arguments, and the answer.
Second Execution:
Execute algorithm A using the certificate instead of a data structure implementation. When the algorithm executes a data structure operation, verify that the operation and arguments match those in the certificate and return the answer (if any) from the certificate. Validate the answers provided by the certificate (see Section 4). If at any point a mismatch of operations is detected or the validation fails, output "error" and stop.

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 time which is significantly less than the execution time obtained by running algorithm A twice, yet these two approaches give similar fault-detection capabilities. That is, if transient faults occur during only one of the executions then either an error will be detected or the output will still be correct. Note that the first execution can be slower than a simple execution of algorithm A since it must output a certificate. 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 speed-up.

3.3 A Simple Example: Answer Validation for Disjoint-Set-Union

In order to better explain the schema for using certificates, in this sub-section we will discuss the answer validation for disjoint-set-union problem. Dynamic sets are fundamental data structures and can grow, shrink, or otherwise change over time by algorithmic manipulations. The disjoint-set-union problem concerns a collection of disjoint dynamic 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. Set A must be undefined before creation. Since sets must be disjoint we require that x not yet 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 be currently defined and be 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 content of the set.

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(4,3)		1:{1,2},4:{4,3},5:{5}
union(4,1)		4:{1,2,3,4},5:{5}
find(2)	4	
find(4)	4	
create(6,6)		4:{1,2,3,4},5:{5},6:{6}
union(5,6)		4:{1,2,3,4},5:{5,6}
create(7,7)		4:{1,2,3,4},5:{5,6},7:{7}
union(5,7)		4:{1,2,3,4},5:{5,6,7}
find(6)	5	

Table 1: Sequence of operations for a disjoint-set-union

We may build a forest to represent the three operations on the disjoint-set-union. Every node represents a set after certain operations. Every solid arc points from the new set to the old set in **union** operation. Every dashed arc represents the **find** operation and points from the set of operation argument to the set of returning answer. In addition, there is no arc for **create** operation and all leaf nodes are created sets. As a final step we will perform a traversal of the forest and perform appropriate checks. The **union** forest corresponding to the operations given in Table 1 is shown in Figure 2

The disjoint-set-union problem is a classic problem which has many applications [10] 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 gives the tight upper bound of O($m\alpha(m, n)$) [11, 12] for this problem. The α refers to the inverse of Ackermann's function which is a very slowly growing function. His solution and earlier solutions used a path-compression heuristic [13]. Fredman and Saks give a lower bound of $\Omega(m\alpha(m, n))$ [14] in a general cell-probe model. Gabow and Tarjan show how to solve some important special cases of this problem in O(m) time [15].

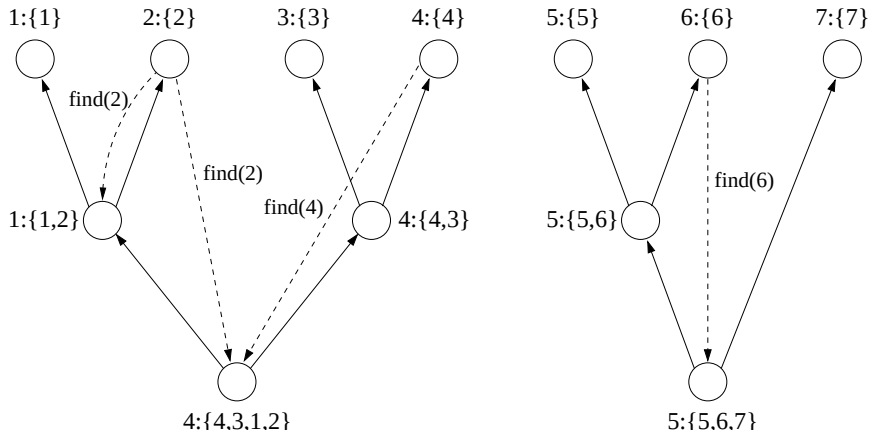

Figure 2: Disjoint-set-union forest.

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 that this time complexity is superior to the complexity of actually performing the sequence of operations as discussed above. One approach to this problem in $O(m)$ time uses the powerful algorithm of Gabow and Tarjan [15]. However, we shall present a simpler approach with a small constant of proportionality that is tailored to this problem.

Algorithm for Answer Validation of 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] is 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] is 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** operation.) If *whereis*[x] is nil then output "ill-formed". Otherwise, *whereis*[x] points to some *treeleaf*, denoted as *tleaf.* If *activeset*[A] is nil then output "ill-formed".

Otherwise, *activeset*[A] points to some *treeleaf* or *treenode*, denoted as t. As a result, every time a **find** operation is performed, a pointer to the corresponding t is added to the list of pointers contained in the corresponding *tleaf*.

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

Scan through the array *activeset*[] to find non-nil pointers. It is not difficult to see that each non-nil pointer points to the root of a tree made up of nodes of type *treenode* and type *treeleaf*. The tree uses the edges in the *left* and *right* fields of *treenode*.

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

Proof: A full proof of correctness is not difficult, but is detailed. First, we show that if the answer to each **find** operation is correct, then the algorithm outputs "correct". To demonstrate this, we need only show that for each **find** operation, the pointer added to a *treeleaf* node always points to an answer of that node. First we show that the following two properties hold after each step in the construction of the forest of trees.

1. The non-nil pointers in *activeset*[] correspond exactly to the sets in existence after this step.
2. Each non-nil pointer *activeset*[A] points to the root of a tree. This tree may consist of a single *treeleaf* or be made up of *treenodes* and *treeleaves*. The treeleaves in this tree correspond to the original sets that were merged together to form the set A.

These properties are trivially true after the first **create** operation is executed, establishing the base case. Suppose that they are true after the first k operations. We may assume that the (k+1)-th operation is a **create** or **union** operation since **find** operations add pointers to *treeleaves* but do not otherwise change the forest of trees. If the (k+1)-th operation is a **create** operation the two properties are clearly maintained since a new set is created and pointed to by a formerly nil *activeset*[] pointer and the rest of the forest is not modified. If the (k+1)-th operation is a **union(A,B)** operation, then a new *treenode* is created with children *activeset*[A] and *activeset*[B]. Since those are the roots of trees containing the *treeleaves* representing sets merged to create A and B respectively, the new *treenode* is the root of a tree that contains all the *treeleaves* for the union of A and B. This, in addition to the updates to the *activeset* array, preserves the conditions.

Now consider a **find(**x**)** operation with answer A. Since this answer is correct, we know that A is either the set in which x was created or a set resulting from merging that set with other sets. Let *tleaf* be the *treeleaf* pointed to by *whereis*[x]. In the first case, *activeset*[A] also points to *tleaf.* In the latter case, the tree pointed to by *activeset*[A] will contain the *tleaf.* Thus the pointer added to *tleaf* processing the **find** operation points either to *tleaf* or to an ancestor of *tleaf.*

Thus, we have proven that if the **find** answers are correct then the algorithm outputs "correct". We will now prove that if the algorithm outputs "correct" then the answers given for **find** operations are correct. We prove the contrapositive, that is, if the answer to some **find** operation is not correct then the algorithm does not output "correct".

Consider the first **find(**x**)** operation with an incorrect answer A. Suppose *whereis*[x] points to a *treeleaf,* call it *tleaf,* which is part of a larger tree. Then if the correct answer had been given, a pointer would have been added to *tleaf* pointing to the root of the tree. It might initially appear that an incorrect answer could cause the addition of a pointer to a *treenode* other than the root, but that is still an ancestor of *tleaf.* We shall see that this is impossible.

The reason that this cannot occur is that we have demonstrated above that each of the non-nil pointers in the *activeset*[] points to the root of some tree, not to internal nodes of a tree. Since the pointer we attempt to add to *tleaf* is to the node pointed to by *activeset*[A], there are two possibilities. First, *activeset*[A] may be nil in which "ill-formed" is output and we are done. Otherwise *activeset*[A] is non-nil and thus points to the root of a tree. Since A is not the correct answer, this is not the tree containing *tleaf.* Thus, the check performed for this pointer during the depth first traversal will fail and "incorrect" will be the output.

Theorem 4 *The answer validation algorithm for disjoint-set-union has a time complexity of O(m) for processing a sequence of m operations.*

Proof: The construction of the forest of trees clearly requires only linear time. Similarly, aside from the pointer checks performed at *treeleaf* nodes, the depth first traversal of the trees requires only linear time. Thus it remains to show that each pointer stored in a *treeleaf* may be checked in constant time. This may be done as follows. During construction of the forest of trees, each *treenode* or *treeleaf* created is assigned an unique identifier between 1 and m. An array of m booleans is created for use in the depth first traversal. During the depth first traversal, when node i is placed on the stack the i-th array element is set to true, and when it is taken off the stack that element is set to false. This array enables each pointer check to be performed in constant time. Since there are only a linear number of pointers to check, one per **find** operation, only linear time is required for all pointer checks. Finally, performing the array updates during the depth first traversal requires only a linear amount of work. Therefore, the entire algorithm runs in $O(m)$ time.

4 Answer Validation for Priority Queue

In this section, the answer validation problem and algorithm using certificates are to be discussed in detail for priority queue data type. Firstly we shall give formal definition of priority queue. Secondly we shall present its answer validation algorithm. Thirdly some important properties will be revealed and proved regarding some stack operations used in the algorithm. Finally these properties will be used to further prove the correctness of the answer validation algorithm.

4.1 Priority Queue Definition

A *priority queue* contains a sequence of $(item, value)$ pairs, where $item$ is a number from the integer set $[1...N]$. At any given time, the pairs in the priority queue have distinct item numbers though the value fields may be the same for multiple pairs. It is possible, however, 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 $i_1 < i_2$). We may also define $(i_1, v_1) = (i_2, v_2)$ iff $v_1 = v_2$ and $i_1 = i_2$, but in this case the two pairs cannot be in the priority queue at the same time. In this paper, the following operations are supported by a priority queue:

insert(i**,**v**)**: Add the pair (i, v) to the priority queue if the item number i is not currently in use.

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

(i**,**v**) = deletemin()**: Return the smallest pair (i, v) in the priority queue and remove it from the queue.

delete(i**)**: Remove the pair with item number i. This operation fails if currently there is no such element in the priority queue.

changekey(i**,**w**)**: Find the pair with item number i and change its value field to w. Since this operation may be implemented as a **delete(**i**)** followed by an **insert(**i**,**w**)**, we do not consider it for the rest of this paper.

4.2 Answer Validation Algorithm

Input: Sequence of operations 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 in the sequence is being processed. It is 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 the other item numbers i, $Value[i]$ is undefined.

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

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

push(S, $atime, i, v, s$*)*: Add the element $(atime, i, v, s)$ to the top of the stack S.

$(atime, i, v, s) =$ *pop(S)*: Remove the top stack element and return it.

$(atime, i, v, s) =$ *top(S)*: Return the top element without removing it from the stack.

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

$istop((atime, i, v, s))$: Return *true* iff the argument is the top stack element and *false* otherwise.

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

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

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

Initially, $AnswerStack$ contains the single quadruple $(-1, 0, -\infty, null)$, where $(0, -\infty)$ is guaranteed to be smaller than any pair. We will frequently speak of one stack element $(atime_1, i_1, v_1, s_1)$ being "above" or "below" another element $(atime_2, i_2, v_2, s_2)$. $(atime_1, i_1, v_1, s_1)$ above $(atime_2, i_2, v_2, s_2)$ means that it is closer to the top of the stack, i.e., that $(atime_2, i_2, v_2, s_2)$ was already on the stack when $(atime_1, i_1, v_1, s_1)$ was pushed. $(atime_1, i_1, v_1, s_1)$ below $(atime_2, i_2, v_2, s_2)$ means that $(atime_2, i_2, v_2, s_2)$ is above $(atime_1, i_1, v_1, s_1)$. 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.

We now describe the pseudo-code for the answer validation algorithm, which consists of routines for each operation and a **FinalPhase** routine. Note that 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 the set as being part of the stack element. The variable *CurrentTime* is incremented after each operation. An example of how these routines function is presented in Table 2. In the column labeled "Stack used in validation", the top stack element is at the leftmost.

insert(i**,**v**)** (shown in Figure 3): Check whether the item number i is currently in use. If so, output "ill-formed" and halt. If not, set $InsertTime[i]$ to *CurrentTime* and $Value[i]$ to v. Add i to the set s belonging to the top stack element.

(i_1**,**v_1**) = min()** (shown in Figure 4): First check if *InsertTime*[i_1] is set to *unused* and if *Value*[i_1] is not equal to v_1. If either of these is the case, output "ill-formed" and halt. Otherwise, pop elements off the top of the stack

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

Table 2: Sequence of priority queue operations illustrating answer-validation algorithm

insert(*i*,*v*) { /* (i,v) is the pair to be inserted */

if ($InsertTime[i] \neq unused$) then output "ill-formed" and halt

$InsertTime[i] = CurrentTime$

$Value[i] = v$

add(*top*($AnswerStack$),i)

}

Figure 3: **insert** operation

until the pair (i_1, v_1) is less than that of the top stack element (it is possible that no elements are popped). If the stack is initially empty, just push $(CurrentTime, i_1, v_1, s_1)$ onto the stack, where s_1 is an empty set. Otherwise, let $(atime_2, i_2, v_2, s_2)$ be that top stack element and compare $atime_2$ with the insertion time of (i_1, v_1). If (i_1, v_1) 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_1, v_1) was in the queue). Otherwise, push $(CurrentTime, i_1, v_1, s_1)$ onto the stack, where s_1 is the union of the sets contained in stack elements that were popped off the stack.

```
min(i1,v1) {      /* (i1, v1) is the answer given in the input for this min */
    if (InsertTime[i1] = unused or Value[i1] ≠ v1)
        output "ill-formed" and halt
    s1 = null
    while (not empty(AnswerStack)) {
        (atime2, i2, v2, s2) = top(AnswerStack)
        if ((i1, v1) > (i2, v2)) {
            pop(AnswerStack)
            s1 = s1 union s2
        }
        else if (InsertTime[i1] < atime2)
            output "incorrect" and halt          (1)
        else
            exit from while loop
    }
    push(AnswerStack, CurrentTime, i1, v1, s1)
}
```

Figure 4: **min** operation

(*i*,*v*) = deletemin() (shown in Figure 5): Perform the same operations as for **min**. In addition, remove the item number i from the set containing it and set *InsertTime*[i] to *unused.*

```
deletemin(i,v) {      /* (i,v) is the answer given in the input for this deletemin */
    min(i,v)
    remove(i)
    InsertTime[i] = unused
}
```

Figure 5: **deletemin** operation

delete(i_1) (shown in Figure 6): First check that there is a pair (i_1, v_1) in the priority queue. If not, output "ill-formed" and halt. Otherwise, let $(atime_2, i_2, v_2, s_2)$ be the stack element with s_2 containing i_1 and 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 $(atime_2, i_2, v_2, s_2)$ is the top stack element we are done.

Otherwise let $(atime_3, i_3, v_3, s_3)$ be the element immediately above $(atime_2, i_2, v_2, s_2)$. If (i_1, v_1) is smaller than (i_3, v_3), output "incorrect" and halt. Otherwise the operation succeeds.

```
delete(i_1) {      /* remove the pair with item number i_1 */
    if (InsertTime[i_1] = unused)
        output "ill-formed" and halt
    v_1 = Value[i_1]
    (atime_2, i_2, v_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((atime_2, i_2, v_2, s_2))) {
        (atime_3, i_3, v_3, s_3) = up(atime_2, i_2, v_2, s_2)
        if ((i_1, v_1) < (i_3, v_3))
            output "incorrect" and halt                                 (3)
    }
}
```

Figure 6: **delete** operation

FinalPhase() (shown in Figure 7): In the final phase, we examine the pairs remained in the queue. For each such pair (i_1, v_1) form a corresponding triple $(InsertTime[i_1], i_1, v_1)$. Sort these triples by $InsertTime$ and obtain the resulting list *RemainderList.* For each element $(atime_2, i_2, v_2, s_2)$ in *AnswerStack,* check whether there is any triple $(InsertTime[i_1], i_1, v_1)$ in *RemainderList* such that the pair (i_1, v_1) is smaller than (i_2, v_2) and (i_1, v_1) is inserted before $atime_2$. If yes, output "incorrect" and halt. Otherwise, output "correct".

4.3 Stack Properties

In this section, we list five properties of $AnswerStack$ that are maintained by the aforementioned operations. These properties will be used to prove the correctness of the answer validation algorithm.

For a stack element $(atime, i, v, s)$, we refer to $(atime, i, v)$ as its corresponding stack triple (or triple for short). The stack triples are ordered as follows: $(atime_1, i_1, v_1) < (atime_2, i_2, v_2)$ iff $(i_1, v_1) < (i_2, v_2)$ or $((i_1, v_1) = (i_2, v_2)$ and $atime_1 < atime_2)$. $(atime_1, i_1, v_1)$ is (immediately) above (or below) $(atime_2, i_2, v_2)$ if $(atime_1, i_1, v_1, s_1)$ is (immediately) above (or below) $(atime_2, i_2, v_2, s_2)$. The field $atime$ is always different for different triples (since each operation produces at most one triple), and therefore two triples are never equal. The following stack properties are maintained throughout the algorithm.

```
FinalPhase() {      /* executed after all operations have been performed */
    For each pair remaining in the queue
        form triple (InsertTime[i1], i1, v1)
    RemainderList = these triples sorted by InsertTime[i]
    For each (atime2, i2, v2, s2) on AnswerStack {
        if (there is an (InsertTime[i1], i1, v1) on RemainderList
          such that InsertTime[i1] < atime2 and (i1, v1) < (i2, v2))
            output "incorrect" and halt          (4)
    }
    output "correct"
}
```

Figure 7: **FinalPhase** routine

Property 1 *The answer-time fields of stack elements are in strictly descending order from the top to the bottom of the stack.*

Proof: 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 time must decrease from the top to the bottom of the stack. □

Property 2 *Let* $(atime_1, i_1, v_1)$ *and* $(atime_2, i_2, v_2)$ *be two adjacent stack triples with* $(atime_1, i_1, v_1)$ *immediately above* $(atime_2, i_2, v_2)$*, then* $(atime_1, i_1, v_1) < (atime_2, i_2, v_2)$*. More generally, the stack triples are in strictly ascending order from the top to the bottom of the stack. Furthermore, since the answer-time fields are in descending order, this implies that the pairs formed by the second two elements of each triple are in strictly ascending order.*

Proof: The second property is trivially true for a stack with fewer than two elements, 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 $(atime_1, i_1, v_1)$ be the answer triple for that operation. Let $(atime_2, i_2, v_2)$ be the smallest stack triple such that $(atime_1, i_1, v_1) < (atime_2, i_2, v_2)$. If there is no such triple, then all stack elements will be popped by the **min** or **deletemin** operation and $(atime_1, i_1, v_1, s_1)$ pushed, in which case the property remains true. Otherwise, all elements above $(atime_2, i_2, v_2, s_2)$ will be popped, since by assumption their corresponding triples are all smaller than $(atime_2, i_2, v_2)$, and hence smaller than $(atime_1, i_1, v_1)$. Similarly, all triples below $(atime_2, i_2, v_2)$ are greater than it and are in strictly ascending order. Therefore when the elements above $(atime_2, i_2, v_2, s_2)$ are popped and $(atime_1, i_1, v_1, s_1)$ pushed, the ordering of the triples is maintained. □

This property also 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).

Property 3 *Let $(atime_1, i_1, v_1)$ immediately above $(atime_2, i_2, v_2)$, and t_{ins} the insert time for the operation that inserted the pair (i_1, v_1) corresponding to the triple $(atime_1, i_1, v_1)$, then $t_{ins} > atime_2$.*[3]

Proof: Property 3 is checked by **min** and **deletemin** operations when the stack element containing the triple $(atime_1, i_1, v_1)$ is first pushed onto the stack. Since elements 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 triple $(atime_1, i_1, v_1)$ do not depend on $InsertTime[i_1]$. Also 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. □

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

Proof: An **insert** operation adds an appropriate item number to the top element. A **delete** or **deletemin** operation removes an 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. Therefore the union of all sets on the stack does not change. □

Property 5 *Given adjacent stack elements $(atime_1, i_1, v_1, s_1)$ above $(atime_2, i_2, v_2, s_2)$, 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: Considering 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 $(atime_2, i_2, v_2, s_2)$ 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 $(atime_2, i_2, v_2, s_2)$ was the top stack element and i_3 was in s_2. Note that a stack element that is not on the top of the stack can never become the top of the stack since the only operations that remove elements from the stack are **min** and **deletemin** and they end by pushing a new quadruple. Thus $(atime_1, i_1, v_1, s_1)$ must have been pushed onto the stack at some time after t, since eventually it is immediately above $(atime_2, i_2, v_2, s_2)$. Therefore $InsertTime[i_3] < atime_1$.

[3]Recall that if (i_1, v_1) has been inserted multiple times, this corresponds to the last instance of such insertion before the current time. Also note that if this instance of (i_1, v_1) is still in the priority queue, then $t_{ins} = InsertTime[i_1]$).

Considering the second part of Property 5, item numbers may only be placed in the set on the 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 $(atime_1, i_1, v_1, s_1)$ was already on the top of the stack, or it was placed in s_1 during the operation that pushed $(atime_1, i_1, v_1, s_1)$. In the former case we have $InsertTime[i_3] > atime_1 > atime_2$. In the latter case, some stack element $(atime_4, i_4, v_4, s_4)$ was originally on the top of the stack at time $InsertTime[i_3]$ and must have been above $(atime_2, i_2, v_2, s_2)$, thus $InsertTime[i_3] > atime_4 > atime_2$. In either case case the second half of Property 5 holds. □

4.4 Proof of Correctness of the Algorithm

Theorem 5 *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 $InsertTime[\cdot]$ and $Value[\cdot]$ detect ill-formed input, so we assume that the input is well formed. The proof is in two parts:

Part 1: We show that if all 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. First for check (1) in the **min** routine, let (i_1, v_1) and (i_2, v_2) be any two answers from the input sequence and let $(atime_1, i_1, v_1)$ and $(atime_2, i_2, v_2)$ be the associated answer triples. Without loss of generality, 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 $(atime_1, i_1, v_1) > (atime_2, i_2, v_2)$ or $t_{ins} > atime_2$. If not, $(i_1, v_1) < (i_2, v_2)$ 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 answers in the sequence, the check given at (1) cannot fail if all the answers are correct. Second for check (2) in the **delete** routine, (i_1, v_1) is the pair being deleted and $(atime_2, i_2, v_2, s_2)$ is a stack element such that s_2 contains (i_1, v_1), which must exist by 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. Third for check (3) in the **delete** routine, let $(atime_3, i_3, v_3, s_3)$ be the stack element immediately above $(atime_2, i_2, v_2, s_2)$, which must exist since we do not perform check (3) if $(atime_2, i_2, v_2, s_2)$ is the top element. The check returns "incorrect" if $(i_1, v_1) < (i_3, v_3)$. By Property 5, $InsertTime[i_1] < atime_3$. Thus 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. The same argument as examining the check (1) shows that the check (4) in **FinalPhase** routine will also always succeed if all answers are correct.

Part 2: We prove that the if any input answer is not correct, the algorithm will output "ill-formed" or "incor-

rect".

Again, the initial checks against $InsertTime[\cdot]$ and $Value[\cdot]$ check for ill-formed input, so we may assume that the input is well-formed. Let t_{wrong} be the time of the first operation with an incorrect answer, (i_1, v_1) the pair that is the correct answer for that operation, t_{ins} the time of the last **insert**(i_1,v_1) operation occurring before t_{wrong}, and t_{del} the time of the first **delete**(i_1) 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 time t_{ins} and time t_{del}. Note that in a correct execution of the operations this might not be the case. Let $(atime_2, i_2, v_2)$ be the largest answer triple occurring with $t_{ins} < atime_2 < t_{del}$. $(i_2, v_2) > (i_1, v_1)$ because (i_2, v_2) is not smaller than the incorrect answer given at time t_{wrong}. During the time period there may be several operations that return (i_2, v_2), but our ordering on triples guarantees that we select the last such operation.

There are now three cases.

Case I: (i_1, v_1) is returned as an answer at some time $atime_1 > atime_2$. This could be the result of a **min** operation, so this case may apply even if $t_{del} = infinity$. Clearly $atime_1 \leq t_{del}$. At time $atime_1$ the element $(atime_2, i_2, v_2, s_2)$ must be in the answer stack, because it was placed on the stack at time $atime_2$ and no larger pair has been returned as an answer between $atime_1$ and t_{del}. Suppose that $(atime_2, i_2, v_2)$ is actually the topmost triple after popping any element with its triple smaller than $(atime_1, i_1, v_1)$, then check (1) will fail since $InsertTime[i_1] = t_{ins} < atime_2$. Otherwise, let $(atime_3, i_3, v_3)$ be the topmost triple of the stack after popping. Then Property 1 implies that $atime_2 < atime_3$, so $InsertTime[i] = t_{ins} < atime_2 < atime_3$ and once again check (1) fails. When it reaches the above specified comparison, the algorithm will fail. If not, it must be due to an earlier comparison failure, stopping the execution of the algorithm. In either case, the algorithm will output "incorrect" and halt.

Case II: (i_1, v_1) is not deleted and is never returned as an answer. 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 (t_{ins}, i_1, v_1) will be in the list of remaining elements considered during **FinalPhase**. The same reasoning as in *Case I* implies that $(atime_2, i_2, v_2)$ will be in $AnswerStack$ after all operations are executed. Now let $(atime_3, i_3, v_3)$ be the smallest triple remaining in $AnswerStack$ that is larger than (t_{ins}, i_1, v_1). Then, if $(i_3, v_3) \neq (i_2, v_2)$ the triple $(atime_3, i_3, v_3)$ must be above $(atime_2, i_2, v_2)$ in the stack, so $atime_3 \geq atime_2$. The triples (i_1, v_1, t_{ins}) and $(atime_3, i_3, v_3)$ will fail in check (4) in **FinalPhase** routine, since $(i_1, v_1) < (i_3, v_3)$ and $InsertTime[i] = t_{ins} < atime_2 \leq atime_3$. As in the first case, if the above comparisons can ever be reached, the algorithm will fail. Otherwise, it can only be because that an earlier comparison failed, ending the algorithm. In either case, the algorithm will output "incorrect" and halt.

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

is irrelevant. At time t_{del}, $(atime_2, i_2, v_2, s_2)$ must be on the stack, since it is the largest answer triple with time greater than t_{ins}. If i_1 is in s_2, then check (2) will fail, i.e., output "incorrect", since $(i_1, v_1) < (i_2, v_2)$. If not, let $(atime_3, i_3, v_3, s_3)$ be the stack element where i_1 is in s_3. $InsertTime[i_1] < atime_2$, so this stack element must be below $(atime_2, i_2, v_2, s_2)$, because otherwise Property 5 would require $InsertTime[i_1] > atime_2$. Suppose check (2) succeeds, i.e., it does not output "incorrect", 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 Property 2, so we must have $InsertTime[i_1] > atime_3$, in which case check (3) will be performed. Let $(atime_4, i_4, v_4, s_4)$ be the stack element immediately above $(atime_3, i_3, v_3, s_3)$, which must exist because $(atime_2, i_2, v_2, s_2)$ is above $(atime_3, i_3, v_3, s_3)$, then we have $InsertTime[i_1] < atime_4$ by Property 5, but $(i_1, v_1) < (i_2, v_2) \leq (i_4, v_4)$ by Property 2, so check (3) will fail. □

We have not shown that all priority queue operations are correctly performed, only that the answers given are the same as those that would have been given if all operations had been correctly performed. In particular, since it does not return answers, a **delete** operation may remove the wrong element. If that error does not affect the answers to the other operations, we will not be able to detect it. However we could define the **delete** operation to return the pair being deleted. It is trivial to modify the procedure for **delete** in the above algorithm to validate those answers.

5 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**. It is obvious that the approach 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 4 *Based on a sequence of generalized priority queue operations together with the supposed answers, we derive a new sequence of operations called the minimum sequence by:*

1. *Removing every* **max** *operation and the corresponding answer; and*
2. *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 from the original sequence and O_2 from the minimum sequence, are corresponding operations if

1. *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*

2. *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 6 *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 in both S_{min} and S_{max} are correct.*

Proof: Let P_O, $P_{min,o}$, and $P_{max,o}$ be the sets of elements in the priority queue after correct executions of up to operation O in S, S_{min}, and S_{max}, respectively. According to Definition 4, it is easy to see that if O_1 and O_2 are corresponding operations in S and S_{min}, 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}.

If the answers given in S are correct, this means that they are the answers 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 operations O_1 and O_2, the correct answers to the corresponding **min** and **deletemin** operations must be the same. Thus, the answers on the sequence S_{min} are correct. Similar reasoning holds for S_{max}.

On the other hand, if 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 the same elements after the corresponding operations O_1 and O_2. 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 and O_1' is the operation immediately before O_1. Operation O_2' can be defined in a similar fashion with respect to O_2. Since the priority queues $P_{O_1'}$ and $P_{min,O_2'}$ contain the same elements, the correct answers 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, the above statement still holds if O_1 is a **max** or **deletemax** operation. □

6 Experimental Results

In this section we evaluate the use of certificates for data structures as applied to four well-known and significant problems in computer science: the shortest-path problem, sorting, the Huffman-tree problem, and the skyline problem. We have implemented basic algorithms for these problems and applied the approaches described in

Section 4 and Section 5 to implement algorithms which generate and use certificates. Timing data was measured in CPU seconds and collected using a Sun Ultra 5 running Solaris 2.6.

The timing information reported in the tables consists of the run time of the basic algorithm (i.e., no certificate), the run time of the execution that generates the certificate, the run time of the execution that uses the certificate, the percent of savings by using certificates, and the speed-up achieved by the second execution of the certificate-based approach. The percent of savings is computed by comparing the total run time of the two executions for generating and using certificates against twice the run time of the basic algorithm. The speed-up is computed by dividing the run time of the basic algorithm by the run time of the execution that uses the certificate. At each input size, between fifty and one hundred inputs were generated and the execution timed.

During experimentation it was discovered that the overhead of generating the certificate was sometimes small enough to be within the timing error. This resulted in a few trials in which the reported run time for the first execution of the certification-based approach was slightly *less* than the time for the basic algorithm. To avoid these anomalous results, timings are based on twenty-five to one hundred repetitions of the algorithm on each input. A smaller number repetitions was used for the larger sets because of the increased time required for experimentation. The run time reported in the table is the total run time divided by the number of repetitions on each input.

Apart from the data structures, the implementation of both executions of the certificate-based approach is nearly identical to the implementation of the basic algorithm. The only difference is a parameter passed to the data-structure code indicating whether a certificate should be generated or used. All code implementing the certificates is localized to the modules implementing the data structures, allowing the generation and use of the certificate to be transparent to the user of these modules. Due to space constraints only an abbreviated discussion of the algorithms is given.

6.1 Shortest Path

Given a graph $G = (V, E)$ with non-negative edge weights and a source vertex v_0, we wish to find the shortest paths from the source vertex to each of the other vertices in G. This is a classic problem and has been examined extensively in the literature [10]. Our approach is applied to Dijkstra's algorithm which 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 removed from T and added to S.

As shown in Figure 8, a data structure that supports **insert**, **delete**, and **deletemin** operations 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 and 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 ratio $|E|/|V|$ constant, for in practice the run time of the algorithm is affected by this ratio. The column labeled "Size" in Table 3 contains an ordered pair indicating the number of vertices and edges.

```
ShortestPath(G = (E, V),v0) {
/* d[V] = array of the best known path lengths from v0, indexed by the vertices.
   S = set of vertices for which the shortest paths from v0 have been determined.
   Q = priority queue of vertices in V − S, with d[V] values used as keys */
   d[v0] = 0
   for v ≠ v0
      d[v] = ∞
   S = ∅
   for v ∈ V
      Q.insert(d[v], v)
   while (Q ≠ ∅)
      u = Q.deletemin()
      S = S ∪ {u}
      for each vertex v in Adj[u]
         if d[v] > d[u] + w(u, v)
            Q.delete(d[v], v)
            d[v] = d[u] + w(u, v)
            Q.insert(d[v], v)
}
```

Figure 8: Shortest-path algorithm

Size	Basic Algorithm	First Execution (generates certificates)	Second Execution (uses certificates)	Speed-up	Percent of Savings
20000,200000	0.83	0.88	0.39	2.13	23.49
40000,400000	1.90	2.00	0.88	2.16	24.21
60000,600000	3.04	3.19	1.42	2.14	24.18
80000,800000	4.38	4.56	1.99	2.20	25.23

Table 3: Shortest path

6.2 Heapsort

Sorting is a fundamental operation in computer systems [10] and may be implemented with a priority queue (or more specifically, a heap) by **insert**-ing 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 $[0, 10000000]$. Figure 9 shows a plot of time taken by a two-version programming solution and a solution using a certificate generated by the answer validation approach for priority queues.

Size	Basic Algorithm	First Execution (generates certificates)	Second Execution (uses certificates)	Speed-up	Percent of Savings
1000000	2.85	3.15	1.19	2.39	23.86
2000000	6.62	7.49	2.52	2.63	24.40
3000000	10.85	12.40	3.88	2.80	24.98
4000000	15.42	17.80	5.26	2.93	25.23

Table 4: Heapsort

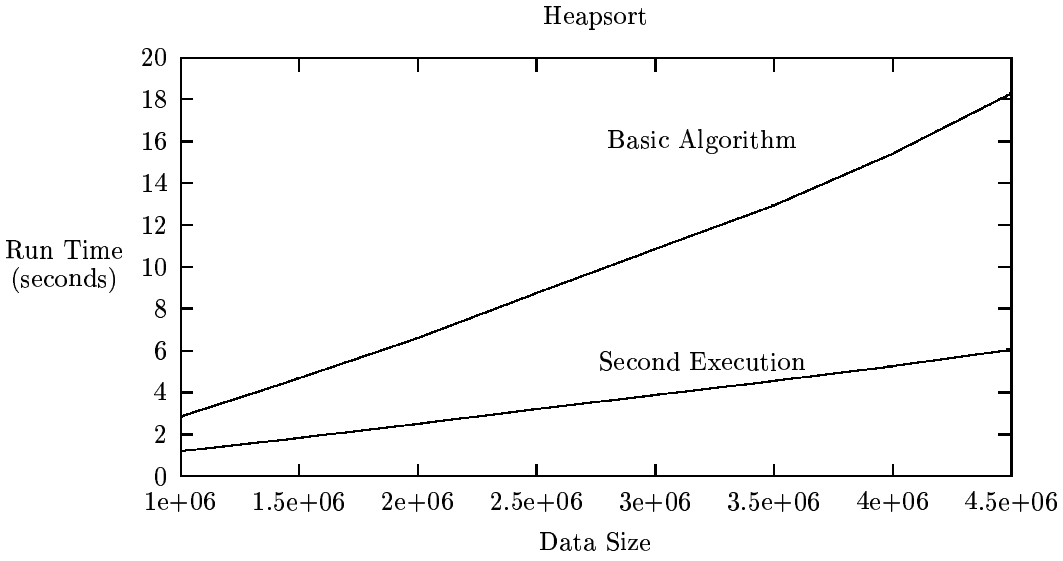

Figure 9: Heapsort

6.3 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 another

classic algorithmic problem [10] and has been used extensively in data-compression algorithms through the design and use of 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, interested readers should consult [10] 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 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 $[0, 100000]$.

Size	Basic Algorithm	First Execution (generates certificates)	Second Execution (uses certificates)	Speed-up	Percent of Savings
100000	0.40	0.41	0.21	1.90	22.50
500000	2.73	2.94	1.25	2.18	23.26
1000000	6.51	6.82	2.61	2.49	27.57
1500000	10.53	11.51	3.64	2.89	28.06

Table 5: Huffman tree

6.4 Skyline

Given a set of rectangles with collinear bottom edges, the *skyline* is the figure resulting from removing all hidden edges. The problem of computing the skyline for a set of rectangular buildings by eliminating hidden lines is discussed in [17]. We use a plane-sweep algorithm that can be easily implemented and is widely used for computational-geometry problems [18]. It typically uses a priority queue for event scheduling and may be amenable to certificate-based approaches.

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 sweep-line is at $x = x_0$. A structure supporting **insert** and **deletemin** operation is all that is needed to order the events, and a structure

supporting **insert**, **max**, and **delete** operations is required to store the rectangle heights. A priority queue can be used to order the sweep-line 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 $[1, 10000]$.

Size	Basic Algorithm	First Execution (generates certificates)	Second Execution (uses certificates)	Speed-up	Percent of Savings
100000	2.53	2.58	0.83	3.05	32.61
200000	6.00	6.13	1.72	3.49	34.48
300000	9.36	9.63	2.64	3.55	34.46
400000	13.42	13.82	3.74	3.59	34.58

Table 6: Skyline

6.5 Experimental Summary

The data clearly indicates that the certificate-based approach results in significant practical time savings. The second execution typically required only 1/2 to 1/3 the CPU time of the basic algorithm. For three of the problems examined, the speed-up was greater for larger data sets. This is not a surprise since the worst-case run time for the basic algorithms is $O(n \log n)$, whereas the second execution of the certificate-based approach requires only linear worst-case time. Indeed, it is somewhat surprising that the speed-up for the shortest path experiment was nearly constant across the range of input sizes. This appears to be due to the fact that the algorithm is executed on random graphs rather than graphs that produce worst-case behavior.

7 Conclusion and Discussion

Run-time certificates can be used to efficiently check the correctness of computational results with relatively small time overhead and software redundancy. In the past, a certificate had to be customized for a specific algorithm. In this paper we have formulated and solved the answer-validation problem using *generalized certificates* for abstract data structures such as disjoint-set-union and priority queues. These data structures are applicable to wide classes of algorithms. Specifically we have applied answer validation to produce certificates for four algorithms using priority queues: shortest path, heapsort, Huffman tree, and skyline. The experimental results indicate that the certificate-based answer validation has fault-detection capability similar to that of two-version programming, but with far less overhead and up to a 3.5-fold speed-up in execution time. Furthermore, as the size of the data

sets employed in the algorithms increases, the advantage of certificate-based approach is even more obvious, making itself particularly attractive for low-overhead enhancement of the dependability of operations associated with computationally intensive applications.

The certificate generation is encapsulated by the priority-queue implementation. Thus, the certificates are not tailored to the specific algorithms, and this technique may be applied to any algorithm using priority queues, either in isolation or in combination with other data structures.

If the original algorithm uses multiple abstract data types and there is an efficient answer validation for each of them, we can generalize our approach by leaving behind a generalized certificate which consists of a separate certificate for each of the abstract data types. The effect on the speed-up of the second execution will be cumulative. Furthermore, the certificate-based approach can be used jointly with other fault tolerance techniques. Multiple answer validation algorithms can be developed to generate and read multiple (but different) certificates. These algorithms can be implemented by separate teams of individuals. A general architecture for the interaction of these algorithms is an important future research topic. The ideas developed in N-version programming can be used as guidance in exploring such issues.

Another application of the answer-validation technique is to run an answer-validation algorithm concurrently with the algorithm that uses the abstract data type. The answer validation algorithm acts as a monitor ensuring 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.

The above techniques also support fault-detection. One approach to a fault-tolerant solution is to implement the abstract data type with a repairable data structure [19]. When the answer-validation algorithm detects an error a repair may be attempted. If successful, continued execution will be possible.

References

[1] Lyu, M. R. “Software Fault Tolerance”, *John Wiley & Sons Ltd*, 1995

[2] Horning, J.J., and et al, “A program structure for error detection and recovery”, *Lectur Notes in Computer Science*, vol.16, pp.177-193, 1974.

[3] Wasserman, H., and Blum, M., “Software Reliability via Run-Time Result-Checking”, *Journal of the ACM*, vol. 44(6), pp. 826-849, 1997.

[4] Chen, L., and Avizienis A., “N-version programming: a fault tolerant approach to reliability of software operation,” in *Proc. Int. Fault-Tolerant Comput. Symp.*, 1978.

[5] Avizienis, A., and Kelly, J., "Fault tolerance by design diversity: concepts and experiments," *Computer*, vol. 17, Aug., 1984.

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

[7] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," in *Proc. Intl. Fault-Tolerant Comput. Symp.*, 1990.

[8] Jin, H., "Run-time result verification using program certificates,", *PhD Thesis*, Johns Hopkins Univeristy, Dept. of Computer Science, 1999

[9] Sullivan, G.F., and Masson, G.M., "Certification Trails for Data Structures", *Proc. Intl. Fault-Tolerant Comput. Symp.*, 1991

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

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

[12] 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.

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

[14] 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.

[15] 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.

[16] Huffman, D., "A method for the construction of minimum redundancy codes", in *Proc. IRE*, 1952.

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

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

[19] Taylor, D., "Error Models for robust data structures," in *Proc. Int. Fault-Tolerant Comput. Symp.*, 1990